

# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Computers never seem to have enough memory, no matter how much memory is installed. One of the most complex and difficult tasks Windows NT faces is managing the limited physical memory present on a computer. This challenge is heightened by the fact that NT must divide physical memory among many processes that might be running simultaneously, giving each process an appropriate memory share. Further, NT must be able to adjust its behavior within a wide range of memory sizes, from as little as 16MB to as much as 1GB or more.

This month I'll introduce the concept of virtual memory, which is based on hardware-supported memory paging. This column will lay a foundation for understanding how NT defines process address spaces. I'll discuss how NT allocates virtual-memory addresses and the internal data structures that record allocation. I'll also describe two powerful features of NT memory management: memory sharing and copy-on-write. Next month, I'll describe more internal data structures, how NT implements shared memory, and working set tuning.

## Virtual Memory

As do almost all modern operating systems (OSs), including Windows 3.x, Windows 95, Win98, and UNIX, NT relies on hardware support to provide processes with the illusion that a computer has more memory than it actually has. The mechanism that implements this illusion is known as paged virtual memory. The Memory Manager (or Virtual Memory Manager) executive subsystem is the NT component responsible for NT's paged virtual memory and for exporting functions that other subsystems and applications can use to interact with paged virtual memory. Processes access memory through their virtual memory address space. In NT, the size of a process' address space is defined by the number of bytes that can be described in 32 bits, which is 2<sup>32</sup>, or 4GB (64-bit NT, which will be available on Alpha and Intel Merced--Deschutes--processors, will have 2<sup>64</sup> bytes of addressability). Thus, barring other resource limitations, a process can theoretically allocate 4GB of code and data. However, most computers today have less than 128MB of physical memory. The 4GB address space of a process is therefore known as virtual memory, because it doesn't directly correspond to physical memory.

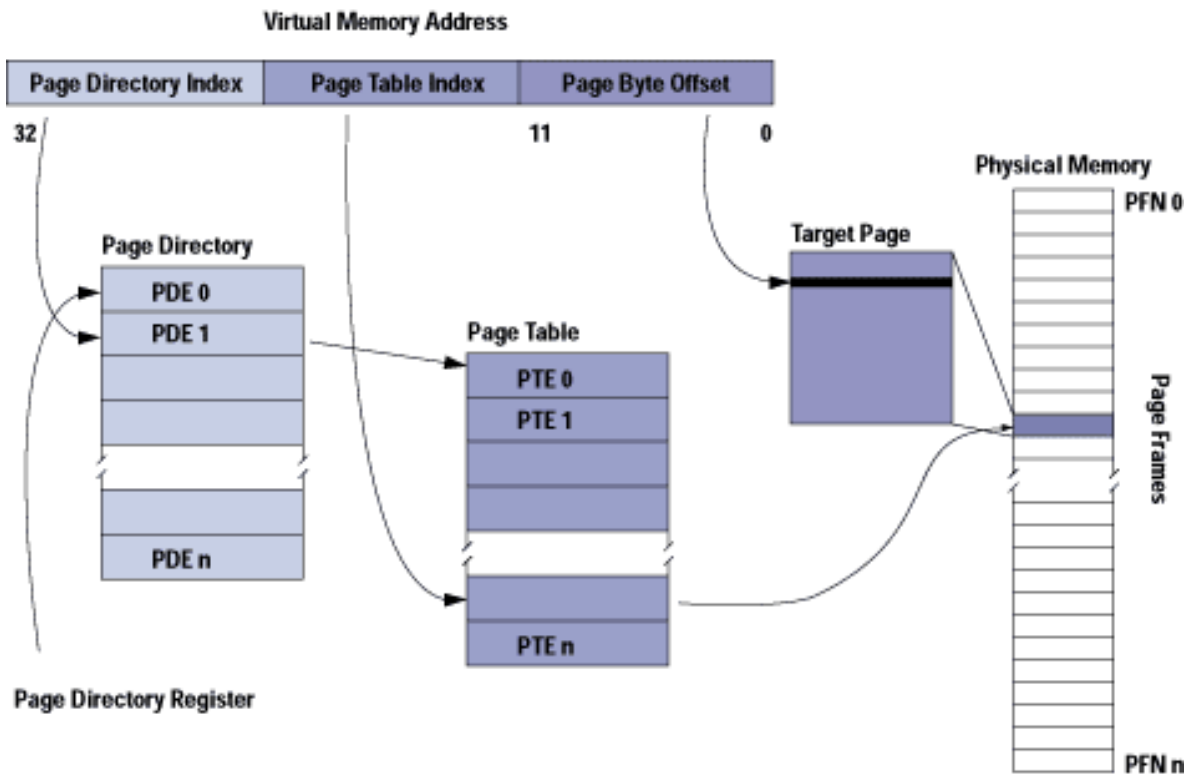
Before I describe how a computer's OS and hardware collude to trick processes into thinking they have 4GB of memory, I'll review memory address translation. The memory management unit (MMU) on Alpha and x86 processors manages physical and virtual memory in blocks called pages. On x86 processors, the size of a page is 4KB, whereas Alpha processors maintain 8KB pages. A computer's physical memory pages are known as page frames, which the processor numbers consecutively with page frame numbers (PFNs).

When a process references its virtual memory, it does so with a 32-bit pointer, and the MMU's job is to determine the location in physical memory to which the virtual address maps. The MMU determines this location by dividing the address into three separate indexes, as Figure 1, page 68, shows: page directory index, page table index, and page byte offset. The MMU uses each index in a three-step address resolution process.

# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)



The first step begins with the MMU locating a table known as the page directory. The MMU locates the page directory by reading the contents of a special hardware processor register. On the x86 this register is the Control Register 3 (CR3), and on the Alpha the register is the Page Directory Register (PDR). Each process has its own private page directory, and the address of that directory is stored in the process' control block data structure. Whenever the scheduler switches from one process to another, NT updates the page directory register with the value stored in the control block of the process that takes over the CPU. The first step in the resolution process continues with the MMU using the page directory index from the virtual address to index to the page directory table. The MMU retrieves from the page directory the page frame number of the location of another table, the page table. Entries in a page directory table are called page directory entries (PDEs) and are 32 bits in size.

In the second step of the resolution process, the MMU uses the page table index from the virtual address on the page table the MMU located. The entry the MMU finds at the page table index identifies a page frame in physical memory. Finally, in the third step, the MMU uses the page byte offset as an index into the physical page and isolates the data that the process wants to reference. Entries in a page table are called page table entries (PTEs). Similar to a PDE, a PTE is 32 bits wide.

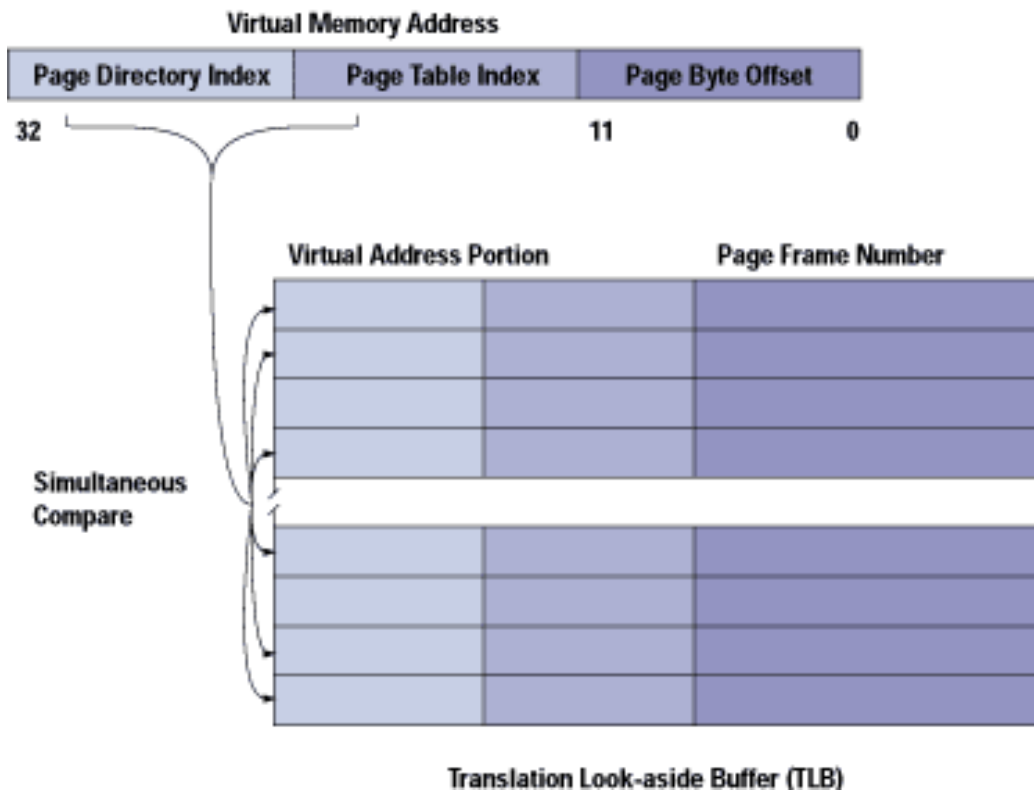
You might wonder why the MMU goes through this three-step gyration. It does so to save memory. Consider a one-step translation in which a virtual address is divided into two components, one locating a PTE in a page table and the other serving as a page offset. To map a 4GB address space, the page table would have to consist of 1,048,576 entries. Four bytes (32 bits equals four 8-bit bytes) per entry would mean that the page table would require 4MB of physical memory to map each process' address space. With a two-level scheme such as the one the x86 and Alpha use, only a process' page directory must be fully defined--page tables are defined only as necessary. Thus, if the majority of a process' 4GB address space is unallocated, a significant saving in memory results because page tables are not then allocated to define the unused spaces.

# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Nevertheless, the three-step translation process would cause a system's performance to be unbearably poor if the process occurred on every memory access. Therefore, x86 and Alpha processors come with an address translation cache such as the one Figure 2 shows, which stores the most recent virtual page to physical page translations. When a process makes a memory reference, the MMU takes the virtual page number and simultaneously compares it with the virtual page number of every translation pair stored in the cache (this type of simultaneous-compare memory is known as associative memory). If there's a match, the MMU can bypass the page directory and page table lookups because it has already obtained the page frame number from the simultaneous compare. Address translation caches are known as Translation Look-aside Buffers (TLBs) or Translation Buffers (TBs). One of the costs associated with the scheduler switching from one thread to another is that, if a newly scheduled thread is from a different process, the TLB must clear the mappings that belong to the old process. Then, the three-step translation is required to fill the TLB with mapping pairs for the new process.



## Paging

What I've described so far is the address translation that occurs when a process references a valid virtual memory address. A process can also make several types of invalid memory references. I'll review error cases first, then I'll discuss situations in which NT considers an invalid memory reference legal and correct--this type of reference is necessary to implement true virtual memory.

Not many processes today require 4GB of address space. Therefore, the address map of most processes is almost entirely empty or undefined. When a process references undefined parts of its virtual memory map, the MMU detects the undefined space by finding, in the first step of translation, a PDE marked invalid in the page directory, or by finding in the second step of translation a PTE marked invalid in a page table. PDEs and PTEs contain enough space in addition to the indexes they

# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

store to keep several bits that serve as bookkeeping information. One bit of a PDE and PTE is the valid bit, which is set only when the translation through the PDE or PTE is configured as legal. If this bit is not set (i.e., it's off), the MMU will stop translation and raise a processor exception called a page fault.

Besides actively playing a role in defining the contents of PDEs and PTEs to define address spaces, NT must respond to page faults and react to them appropriately. When the MMU invokes NT's page-fault handling code, the Memory Manager must check to see if the reference that raised the exception is to an undefined address. If the reference is to an undefined address, the reference raises an access violation (which usually results in the termination of the process) if the processor was executing in user mode, or the blue screen if it was executing in kernel mode. (See my column "Inside the Blue Screen," December 1997, for more information about the difference between user mode and kernel mode).

Because the MMU raises a page fault when a process references an invalid PDE or PTE, paged virtual memory can rely on a nifty trick. To support the illusion that an application process has access to more data and code than physical memory can hold, NT's Memory Manager will move parts of the application to a file on disk called a paging file. NT marks as invalid the PTEs that would otherwise map the pages in the process' address space that correspond to the paged-out data. Thus, when a process tries to reference a paged-out part of its virtual memory, the MMU generates a page-fault.

The Memory Manager's page-fault handler then looks into its internal data structures and discovers that the reference that triggered the page fault was not to an undefined address but rather to an address whose data is stored temporarily in the paging file. The Memory Manager then makes room in physical memory for the page from the paging file that the process is requesting. This operation often means that another page of data from the current process or from another process is sent out to the paging file (a page-out operation). Once the Memory Manager creates space in physical memory for the requested page, the process reads the requested page from the paging file (a page-in operation).

After the page-in operation completes, NT updates the page table of the process that raised the page fault, and the page table points at the new page frame. The processor instruction that caused the memory reference restarts, and on the second time through, the translation succeeds without a page fault and accesses the requested physical data. For both page-in and page-out operations, the Memory Manager works with a disk driver to perform the I/O.

Thus using invalid bits to its advantage, the Memory Manager makes a computer appear to have a total amount of memory that is equal to the size of physical memory plus the sizes of all the paging files. You can create up to 16 paging files in NT, placing each on a separate logical drive.

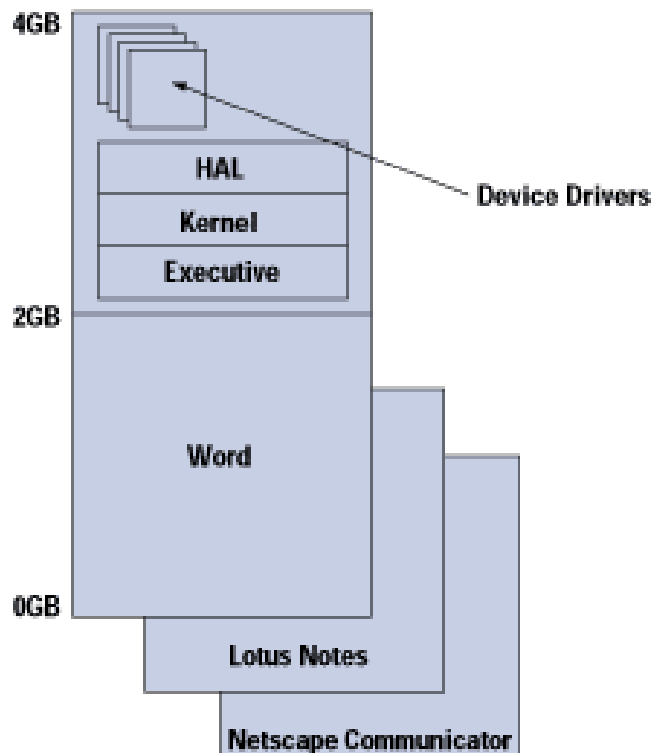
## Process Address Spaces

Now that you understand the mechanics of paging, let's talk about process address spaces and how the Memory Manager defines and keeps track of them. As I've stated, each process has a 4GB virtual address space. This space is divided into two areas, in which the lower addresses map to data and code that are private to the process, and the upper addresses map to system data and code that are common to all processes, as Figure 3 shows. When the scheduler changes execution from one process to another, the page directory register in the process changes so that the process-private portion of the processor's mappings is updated. The system mappings are kept global through common page tables that every process' page directories point to.

# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)



The permissions that apply to pages in each region also reflect the split in mappings. The PTEs of x86 and Alpha processors contain a permissions bit. This bit specifies whether a page is accessible from a program executing in user mode. If a process refers to a page that is not accessible from user mode, the MMU generates a page fault, and the Memory Manager generates an access violation for the process (which is caught by Dr. Watson). As the Memory Manager sets up address spaces, it marks all system PTEs as accessible only from kernel mode. Thus, the Executive subsystems, device drivers, hardware abstraction layer (HAL), and Kernel can reference system memory, but user applications cannot touch system memory. This restriction protects key OS data structures and makes the system secure.

Figure 3 shows that on pre-Service Pack 3 (SP3) systems, the boundary between the lower address space (user mode) and the upper address space (kernel mode) is at 2GB. On SP3 and NT 4.0, Enterprise Edition, the boundary can be moved with a switch on a boot.ini line (/3GB) so that user programs have access to 3GB and system memory can access 1GB. This change enables memory-intensive applications such as database servers to directly address more memory, thus relying less on shuffling data to and from disk in a manner similar to paging.

Memory allocation in NT is a two-step process--virtual memory addresses are reserved first, and committed second. The reservation process is simply a way NT tells the Memory Manager to reserve a block of virtual memory pages to satisfy other memory requests by the process. However, Memory Manager makes no changes to a process at the time of a reservation because the reservation does not use actual memory. When a process wants to use addresses it has reserved, it must commit them. Access to uncommitted reserved memory will typically result in an access violation. When a process wants to commit addresses, the Memory Manager ensures that the process has enough memory quota to do so. The Memory Manager also checks to see that there is enough commit memory (physical memory plus the size of all the paging files) for the commit request. There are many cases in which an application will want to reserve a large block of its address space for a particular purpose

# Inside Memory Management

Mark Russinovich

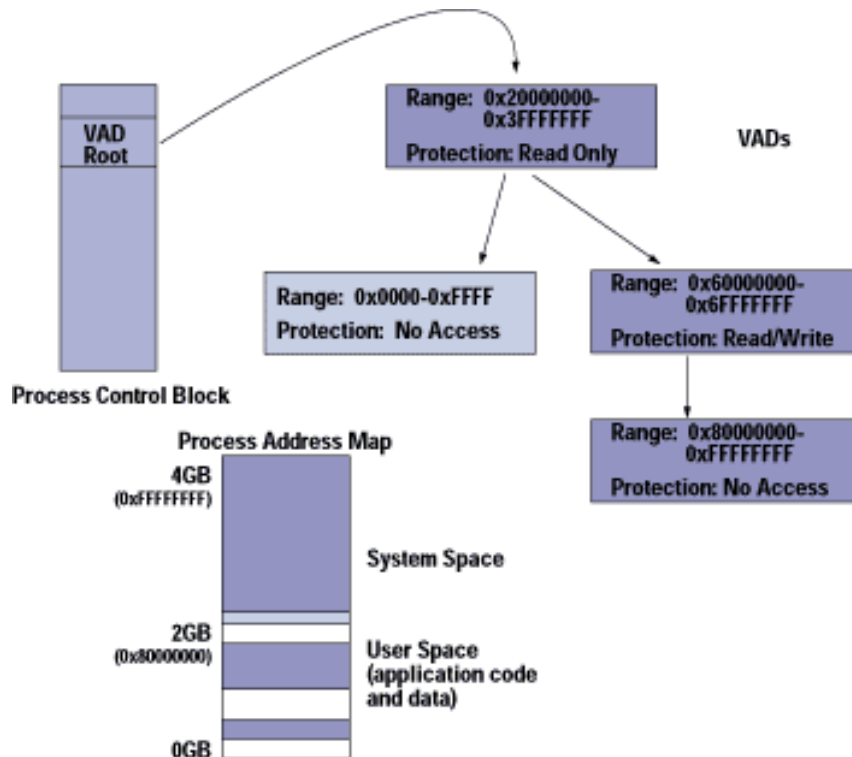
(Reprinted from WindowsItPro Magazine)

(keeping data in a contiguous block makes the data easy to manage) but might not want to use all of the space. The application can specify both reservation and commit in a single request to the Memory Manager with a specific API, and this is the way most applications allocate memory.

One example of the Memory Manager's use of the single-request functionality is in the management of a thread's call stack. The Memory Manager reserves a range of memory (usually 1MB) in a process' address space for each thread created in a process. However, the Memory Manager actually commits only one page of each stack. As a thread uses the stack and touches reserved pages, the Memory Manager commits the reserved pages, increasing the size of the stack.

When a process reserves memory, it must specify the amount of memory but can also request a starting virtual address and specific protections to be placed on the memory. NT uses bits in a PTE that the MMU defines to indicate whether a page can be written to or read from, and whether code can be executed in the page. NT sets the addresses that the API parameters specify. If the PTE bits specify a page as read-only and a process attempts to write to the page, the MMU generates a page fault, and NT's page-fault handler will flag an access violation for the process. The system uses this protection to easily detect errant programs that try to do things such as modify their own code image.

NT keeps track of the reserved or committed address ranges in a process' address space by using a tree data structure, such as the example Figure 4 shows. Each node in the tree represents a range of pages that have identical characteristics with respect to protection and commit state information. This tree structure is a binary splay tree, which means that the depth of the tree (the maximum number of links from the root to any leaf, or bottom, node) is kept to a minimum. The tree's nodes are called Virtual Address Descriptors (VADs), and the process' process control block stores the root of the tree. When a process makes a memory reservation or commit request, the Memory Manager can quickly determine where there are free spaces in the process address map, and whether the request overlaps memory that is already reserved or committed.



# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

## File Mapping and Memory Sharing

A powerful feature of the NT Memory Manager is that it lets programs map files into their virtual address space. This mapping is accomplished in two steps. In the first step, the program creates an object, called a Section Object, to describe a file that the Memory Manager can map. The Section Object holds information about the name of a file, its size, and what portions of it are mapped. The Win32 function `CreateFileMapping` results in a Section Object's creation.

In the second step, the program maps all or part of the file into the process address space. The process invokes the Win32 function `MapViewOfFile` to do this mapping, specifying the start offset in the file to begin mapping and the number of bytes to map. The process specifies the protection (e.g., read-only, read/write) on the view at this time. After the view is mapped, the process can read to and write from the file simply by reading and writing data to the view's addresses in the process address map. The Memory Manager transparently works with the file systems to ensure that all processes accessing the file see any updates the original process makes and that the updates are eventually flushed to disk.

The file-mapping capability makes file I/O extremely straightforward, and NT uses it extensively. When an application launches, the Process Manager maps a view of the application's image to the address space of the application's process. The Process Manager transfers control to the entry point of the image in the address space, and as the application executes, any pages referenced for the first time generate page faults. The page faults result in the Memory Manager reading the applications pages from the file on disk.

NT uses a variation of file mapping to share memory. In this variation, `MapViewOfFile` takes a parameter that specifies memory mapping, rather than file mapping. The Memory Manager backs views of the section with the paging file when memory mapping is specified. The data in the view is no different from regular virtual memory that a process reserves and commits. However, NT can give sections of the data names in the Object Manager namespace so that two or more processes can open the same section, map views to their own address spaces, and then communicate with one another through this shared memory.

## Copy-on-Write

The Memory Manager implements an important optimization of memory sharing called copy-on-write. There are several common scenarios in which a process might want to use the same data as another process but keep any modifications it makes private to itself. For example, if two or more processes start the same application and one process modifies the data in the image, other processes should not see that modification. The obvious way to accomplish this kind of private modification is to load multiple copies of the application into memory; however, this strategy wastes space. Instead, the Memory Manager marks the physical pages containing the image read-only and notes in an internal data structure (which I'll describe next month) that the page is a copy-on-write page. When a process tries to modify the copy-on-write page, that action will generate a page fault (because the page is read-only). The Memory Manager will see that the process referenced a copy-on-write page and will copy the contents of the copy-on-write page to another page that is private to the process that made the reference. Then, when the faulting instruction restarts, the process can modify its private copy of the page. Figure 5 demonstrates this procedure. In the figure, Process 1 and Process 2 share three copy-on-write pages. If Process 1 writes to one of the pages, it will get its own private copy of the page, and Process 2 will retain the original of the page.

# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

NT uses the copy-on-write functionality mostly for executable images. When programs start, the Process Manager maps copy-on-write images. The POSIX subsystem also uses copy-on-write for handling the POSIX fork operation. When a POSIX process forks, the Process Manager creates a child process that will inherit the address space of the parent process. Instead of making a copy of all the parent's memory, the Memory Manager just marks the address-space pages of both the parent and child processes as copy-on-write. When either process makes a modification to its memory, it receives a private copy of the page it wants to modify, and the original of the page will then be private to the other process.

## Stay Tuned

Next month, I'll dig deeper into the Memory Manager to describe the internal data structures it uses to keep track of pages. I'll present the details of how the Memory Manager implements shared memory, and I'll give you an overview of the way working sets--the amount of physical memory assigned to each process--are tuned as the system runs.

Last month, I began this two-part series about memory management in Windows NT by introducing the concept of virtual memory. I reviewed the way x86 and Alpha processors use a two-level virtual address-to-physical address translation scheme. I discussed paging and introduced two powerful features of the Memory Manager: memory-mapped files and copy-on-write memory.

This month, I'll go into more detail about the internal data structures the Memory Manager uses to track the state of memory. I'll discuss working sets and the Page Frame Number (PFN) Database. I'll wrap up with a tour inside the additional data structures the Memory Manager uses to track memory shared by two or more applications, and I'll discuss Section Objects, the data structures the PFN Database uses to implement memory-mapped files.

## Working Sets

The biggest effect the Memory Manager has on individual applications' performance and on the system is in its allocation of physical memory to each active process. The amount of memory the Memory Manager assigns to a process is called the working set. Every process has a working set. A special working set, called the system working set, is physical memory that belongs to parts of the NT Executive, device drivers, and the Cache Manager. If a process' working set is too small, the process will incur a high number of page faults as it accesses page table entries that describe data not present in memory but located either in the process' executable image on disk or in a paging file. For each such access, the Memory Manager must intervene and perform disk I/O to retrieve the data. If a process' working set is too large, the process will not incur page faults, but its physical memory might be holding data that the process will not access for some time, and that other processes might require. Thus, the Memory Manager must try to achieve a balance for all processes according to their memory usage patterns.

The choice NT makes for two memory-management policies--the page fetch policy and the page replacement policy--affects the way NT manages working sets. The page fetch policy is the method NT uses to bring in a process' data. NT uses the most common page fetch policy, demand paging. In demand paging, a process' data is paged-in as a process accesses it. A different page fetch policy, prefetching, requires the Memory Manager to aggressively bring in a process' data before the process asks for it. Prefetching can improve the performance of an application at the expense of other applications, because the Memory Manager's prediction of what data a process will ask for is likely to be imperfect and can waste physical memory in storing unused data. NT therefore implements a slight variation on demand paging: clustered demand paging. Instead of bringing into a process' working set only the page that a process accesses, the Memory Manager will attempt to bring in pages that

# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

surround the requested page. The idea behind clustered demand paging is that if a process accesses a page, it will likely also access the memory just preceding or following that page. The pages the Memory Manager thus brings in are known as a cluster. In NT, clusters can vary between 0 pages and 7 pages, depending on the amount of physical memory present on the system and whether the system is accessing code or data.

The page replacement policy affects the way the Memory Manager decides which page to remove from a working set. The Memory Manager must remove pages whenever physical memory is full of in-use pages and space is necessary to accommodate a page of data a process is accessing. Two characterizations for replacement policies are global and local. In a global replacement policy, the Memory Manager considers all pages of physical memory as replacement candidates, without regard to which working set the pages belong to. In a local replacement policy, the Memory Manager considers as replacement candidates only pages in the working set of the process that is accessing the page to be brought in. The disadvantage of global policy is that a rogue process can, by accessing large amounts of memory very quickly, adversely affect all other processes in the system by forcing their data out of memory. As with its page fetch policy, NT implements a variation of the replacement policy. This variation is local, because it first considers the pages in the working set of the process that is accessing the data. But the variation is also global in that it removes pages from the working sets of other processes if their memory requirements are low.

Replacement policies are further characterized by the method with which they choose a page to replace out of the set of local or global candidates. One policy, first in, first out (FIFO), removes pages in the order in which they were added to the working set (i.e., the first page added is the first page removed). An algorithm that has a more positive effect on the performance of applications is least recently used (LRU). The LRU algorithm requires the aid of the MMU to give the Memory Manager an idea of how recently a process accessed a particular page. LRU replaces first those pages that processes have not accessed for the longest period of time. On uniprocessors, NT uses the clock algorithm, a simple form of LRU replacement that is based on the limited access tracking x86 and Alpha MMUs provide. Whenever a process references a page, these MMUs set the Accessed flag bit in the page table entries (PTEs) of the page. If a page's Accessed flag is not set, the Memory Manager knows that no processes have accessed the page since the last time the Memory Manager examined the page's PTE.

The working set of every process, including the system working set, has a minimum size, a current size, and a maximum size. On a typical NT system with 64MB of physical memory, the default minimum size of a working set is 200KB, and the default maximum size is 1.4MB. Processes with the `PROCESS_SET_QUOTA` privilege can override these values using a Win32 API. When a process starts, it generates a large number of page faults as it references data in its virtual memory map (most often in its executable image) that it must bring into physical memory. The Memory Manager lets a process' current working set size grow as needed to its working set maximum. When a working set reaches its maximum size, the Memory Manager will allow additional pages into the working set only if plenty of free (unused) pages are available. If free pages are not available, the Memory Manager uses the replacement policies I just described to determine which page in the working set to replace. The pages in the working set are linked in a list that the Memory Manager scans. On a uniprocessor, if the Memory Manager finds a page with its Accessed flag set, the Memory Manager clears the flag and proceeds to the following pages, selecting for replacement the next page it finds with a cleared Accessed flag. Thus, the Memory Manager avoids pages that the process has accessed since the previous scan, because the Memory Manager assumes that the process will access those pages again. If the Memory Manager finds no pages to select for replacement, it restarts the scan and

# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

almost certainly finds a page whose Accessed flag it cleared on the previous pass and the process has not accessed since the first scan.

On a multiprocessor system, the Memory Manager does not clear Accessed flags during scans, because any modification to a PTE on a multiprocessor system invalidates Translation Look-aside Buffer (TLB) entries (which I described last month) on all processors. If the Memory Manager invalidates an address' cached translation by clearing its Accessed flag, the address must again undergo the slow three-step virtual memory translation process the next time a process references it. The result is an effectively random page-replacement algorithm on multiprocessors. Microsoft is developing more effective replacement algorithms for the multiprocessor version of NT 5.0.

## The Balance Set Manager

The Memory Manager adjusts the sizes of working sets once every second, in response to page-in operations or when free memory drops below a certain threshold. The Memory Manager also examines all working sets to determine whether the Balance Set Manager thread needs to trim them. If free memory is plentiful, the Balance Set Manager removes pages from only the working sets of processes whose current size is above minimum that have not recently incurred many page faults. However, if the Memory Manager awakens the Balance Set Manager thread because free memory is scarce, the Balance Set Manager can trim pages from any process until it creates an adequate number of free pages--even to the point of dropping working sets below their minimum size.

In addition to trimming working sets, once every 4 seconds the Balance Set Manager can swap out pages that belong to the call stacks of threads that have been sleeping for more than 7 seconds (3 seconds on systems with less than 19MB of memory). The call stack is a special type of memory that records the function invocations a thread makes. If a thread has been asleep (e.g., while waiting for a user to enter a keystroke) for a long time, the Memory Manager assumes the thread will sleep for a longer time. If the Balance Set Manager swaps out the stacks of all the threads in a process, it trims the process' working set to nothing, and the system assumes that the process is inactive. Swapping out the stacks of all the threads in a process minimizes the footprints of processes that don't require memory because they are waiting for infrequent events to occur before they become active.

## The Page Frame Database

Last month, I described how NT organizes physical memory as a list of page frames, each of which is one page in size (4096 bytes on x86 processors, 8192 bytes on Alphas). The Memory Manager mirrors this list of page frames with its own list--the PFN Database. Each entry in the PFN Database corresponds to a physical page of memory (e.g., entry 5 in the PFN Database corresponds to page frame 5 in physical memory). Entries in the PFN Database record information about the state of data stored in corresponding physical pages.

A physical page can exist in one of eight states, which Table 1, page 64, describes. Figure 1 illustrates how the Memory Manager adds to a list all the PFN Database entries that belong to pages in the same state (except pages in the valid or transition states). For example, all modified pages go on the modified page list. A given page undergoes several status changes during its lifetime; therefore, its PFN entry moves among different lists. Figure 1 shows how working sets are associated with valid pages in the PFN Database.

# Inside Memory Management

Mark Russinovich

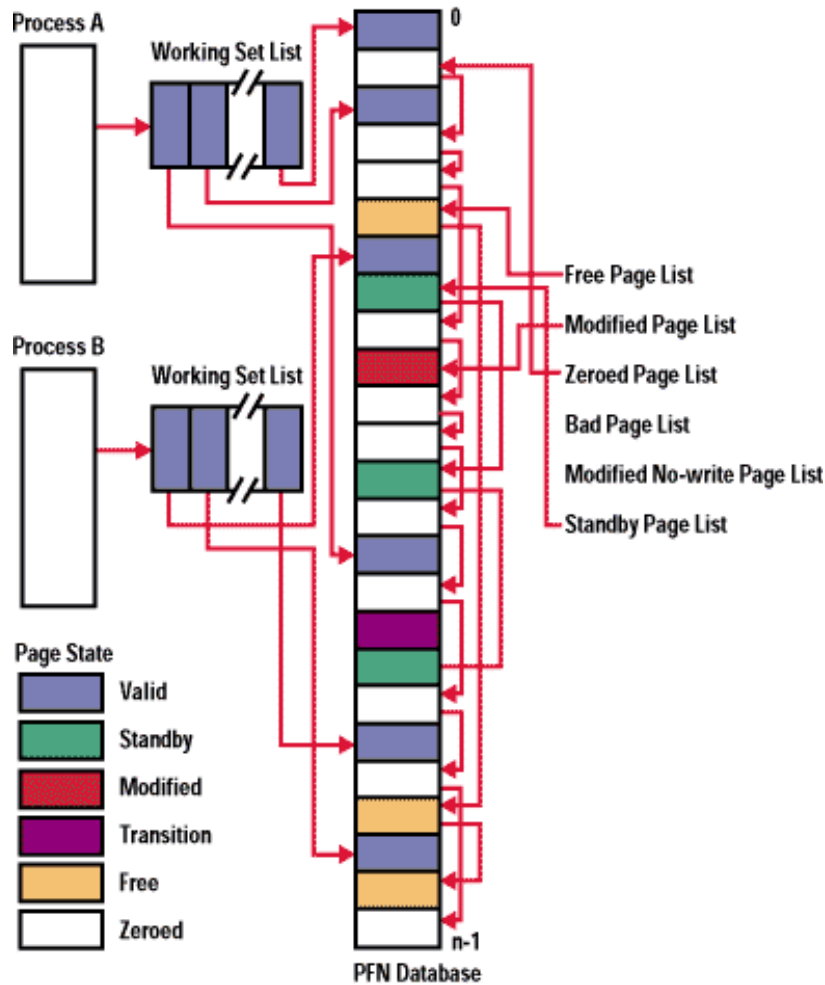
(Reprinted from WindowsItPro Magazine)

TABLE 1: Physical Page States	
Physical Page State	Description
Valid (or Active)	The page is part of at least one process' working set.
Transition	The page is not on any page list, and the Memory Manager is in the process of reading the page in from or writing the page out to a file (e.g., a paging file).
Standby	The Memory Manager has recently removed the page from a working set, and any copy of the page on disk is consistent with the in-memory copy.
Modified	The Memory Manager has recently removed the page from a working set, but the process modified the page, and a copy of the file on disk (e.g., in a paging file) does not have the new content.
Modified No-write	The Memory Manager recently removed the page from a working set, but something, typically a file system driver, marked the page so that it didn't automatically write back to a file, even if a process modified the page. A file system indicated that the Memory Manager need not write the data to disk yet.
Free	The page is not part of any working set, and the process that referenced it removed any other references to it (e.g., it was deallocated). The zero-page thread has not cleared the page (i.e., filled it with zeroes).
Zeroed	The Memory Manager freed the page, and the zero-page thread cleared it.
Bad	The MMU detected the page as faulty, and the page is now off-limits.

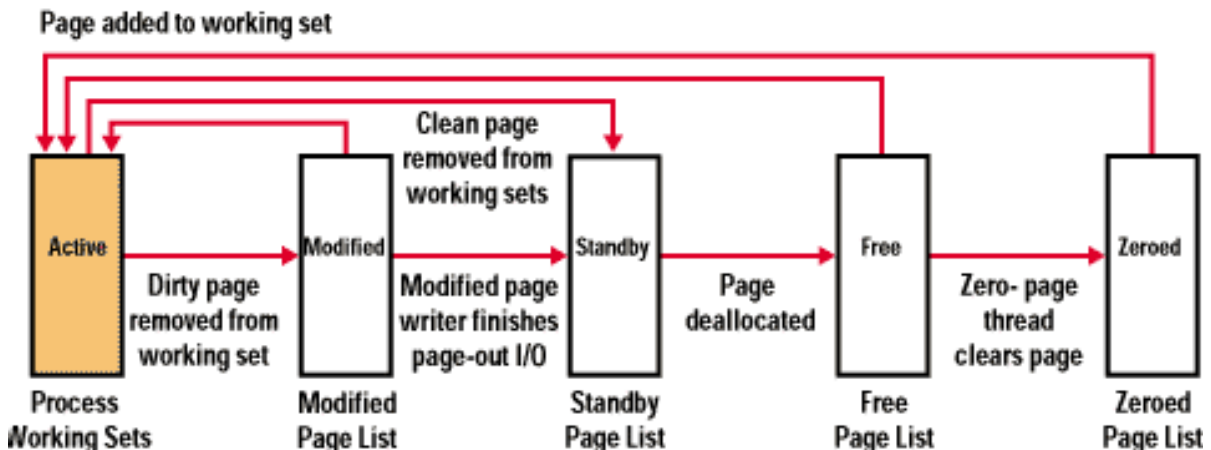
# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)



Let's look at some of these state changes, beginning with a valid page. Figure 2 depicts the state changes I'll discuss. A page is in the valid state if it is currently part of a process working set. A valid page can be dirty or clean--a dirty page is one a process has written to, so that the page's content in physical memory might be different from its content in a backing file (either a paging file or a memory-mapped file). The MMU sets a bit in a page's PTE when a process writes to that page.



# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

When the Balance Set Manager trims a valid page from a working set, the page moves to one of three lists. If the page is clean, it will move to the standby list. If the page is dirty, it almost always moves to the modified page list. In some cases, as when a file system accesses pages backed by files the file system creates to represent its metadata (such as the FAT on FAT file systems, or the MFT--Master File Table--on NTFS file systems), a dirty page moves to the modified no-write page list instead. NTFS takes advantage of this type of marking so that it can ensure that pages belonging to NTFS logging files write to disk before file metadata writes to disk.

A page on the modified page list moves to the standby page list after the Memory Manager's modified page writer writes the page's data to disk. However, while the modified page writer is writing its data to disk, the page makes a stop in the transition state (which has no list). A page on the modified no-write page list moves to the standby page list when the file system instructs the Memory Manager to move the page. This instruction comes after a log file is safely written to disk, for example. When the number of pages on the modified page list exceeds a threshold, or when free memory drops below a threshold (based on the amount of physical memory on the system and determined during the boot), one of two modified page writer threads wakes up to perform disk I/O that sends the data to a paging file or a data file.

A page on the standby page list moves to the free page list when the process that had the page in its working set either frees the virtual memory associated with the page, or the process terminates (which is another way of freeing the virtual memory). An interesting optimization based on the lists I've already discussed is possible. If a page moves from the working set of a process to the standby page, modified page, or modified no-write page lists, and if that process subsequently accesses the page before the Memory Manager assigns the page to a different process, the Memory Manager can place the page back in the process' working set. This operation is known as soft-faulting (or soft-page faulting) memory back to a process. Soft-faulting is a way the Memory Manager can tentatively remove memory from a process but give it back quickly if the process reaccesses the memory in a short amount of time.

Pages on the standby page list move to the zeroed page list after a special thread, called the zero-page thread, clears their content. The zero-page thread executes in the background at priority 0. It runs only if no other thread can run, and its job is to move pages from the free page list to the zeroed page list as it clears their content.

When a process accesses a page and the Memory Manager decides that the process' working set can expand, the Memory Manager checks the zeroed page list to find a physical page to assign to the process. If the Memory Manager finds a page in the zeroed page list, it can immediately use the page. If the zeroed page list is empty, the Memory Manager checks the free page list. If the free page list is also empty, the Memory Manager checks the standby page list. Finally, if the standby page list is empty, the Memory Manager's thread waits while the Balance Set Manager trims a page from a working set, or until a page shows up on the standby page, free page, or zeroed page lists (e.g., a page might be in transition, and after it writes to disk it appears on the standby list).

The necessity of zeroing a page before assigning it to the working set of a different process is a C2 security requirement. (For more information about NT's C2 security rating, see "Windows NT Security, Part 1," May 1998.) The operating system (OS) must reinitialize all OS resources (e.g., memory, disk space, objects) before reassigning them to prevent the creation of security holes in which one process can see another process' potentially sensitive data. In some cases, the process does not require zero-filled memory, as when the Memory Manager allocates a page to store data read from a memory-

# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

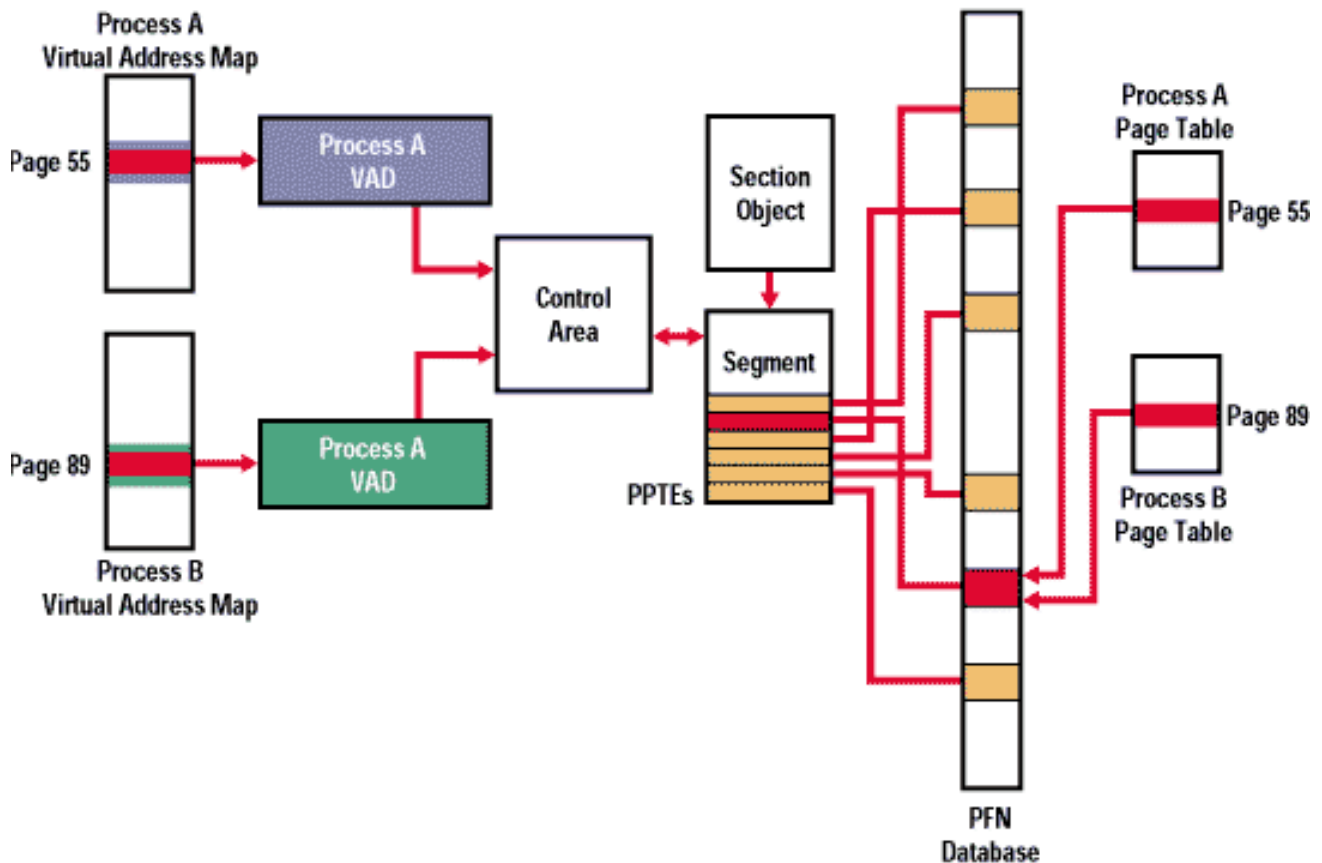
mapped file. In these cases, the Memory Manager checks the free list for an available page before it checks the zeroed list.

The final list is the bad page list. As its name implies, the bad page list is an off-limits holding area in which the Memory Manager, with the support of memory parity error detection, places pages it has detected as faulty.

The number of pages that are on the standby page, free page, and zeroed page lists defines the free memory that various memory-tracking tools (e.g., the Task Manager) report. The commit total is the amount of currently allocated memory that paging file space backs. If the Memory Manager pages the data in commit total memory out of physical memory, the Memory Manager stores that data in a paging file. The commit limit is the amount of commit total memory the Memory Manager can allocate without expanding the sizes of existing paging files.

## Shared Memory and Prototype PTEs

PFN Database entries contain varying information, depending on which state corresponding pages are in. In most cases, a PFN Database entry contains a pointer to a PTE that references a page. However, if two or more processes share the same page, multiple PTEs reference the page: one PTE in the virtual address map of each process sharing the page. Instead of pointing the PFN Database entry at one of these PTEs, the Memory Manager points the PFN Database entry at a data structure the Memory Manager allocates, called a Prototype PTE (PPTe), as Figure 3 shows. I'll describe the way the Memory Manager uses PPTEs to manage shared and mapped memory.



# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

I explained last month that to share memory, a process must create a Section Object. Section Objects hold information about the name of a file, its size, and what portions of it are mapped. Section Object creation defines the shareable data, and each additional process that wants to participate in the sharing must map a portion of the data into its address space. This mapping is known as mapping a view of a section, because processes might map only a portion of the data that a Section Object defines.

When a process allocates a Section Object, the Memory Manager allocates another data structure called a Segment. Segments contain storage to hold enough PTEs to describe all the pages in the Section Object. Usually, when a page moves from a process' working set to the standby page, modified page, or modified no-write page lists, the Memory Manager marks the page's PTE invalid and sets a bit in the page to indicate that the page can be soft-page faulted, which means the PFN of the page stays in the PTE. PTEs that the Memory Manager marks as invalid for shared pages do not continue to store PFNs; rather, the Memory Manager updates them to point at the shared page's PTE. This trick makes it easy for the Memory Manager to update the PFN of a shared page without manually updating the PTEs that refer to the page in the address spaces of all the processes that reference the page.

Consider an example in which two processes share a page. When the page's data is in memory, each process has a valid PTE that stores the PFN where the page's data resides in physical memory. If the Memory Manager removes the PTEs from the working sets of both processes and sends the page's data to a paging file, the PTEs are both invalid and contain pointers to the PTE. When the Memory Manager brings the page's data back into physical memory, the Memory Manager updates the PTE to reflect the page's new PFN. When one of the processes tries to access the page, it generates a page fault. Then the Memory Manager looks at the PTE, finds the new PFN in the PTE the PTE points to, marks the PTE as valid, and updates its PFN. When the second process accesses the page, the Memory Manager updates that process' PTE similarly. Without this optimization, the Memory Manager needs to track down both PTEs (or more, if a greater number of processes shared the page) and update them when it brings the page back into memory--an expensive and potentially wasteful operation.

A reference count in the PFN Database tracks the number of processes that access a shared page, and the Memory Manager knows that when the count becomes zero, the page is not marked valid in any working set. At that point, the Memory Manager moves the page to the standby page or modified page lists.

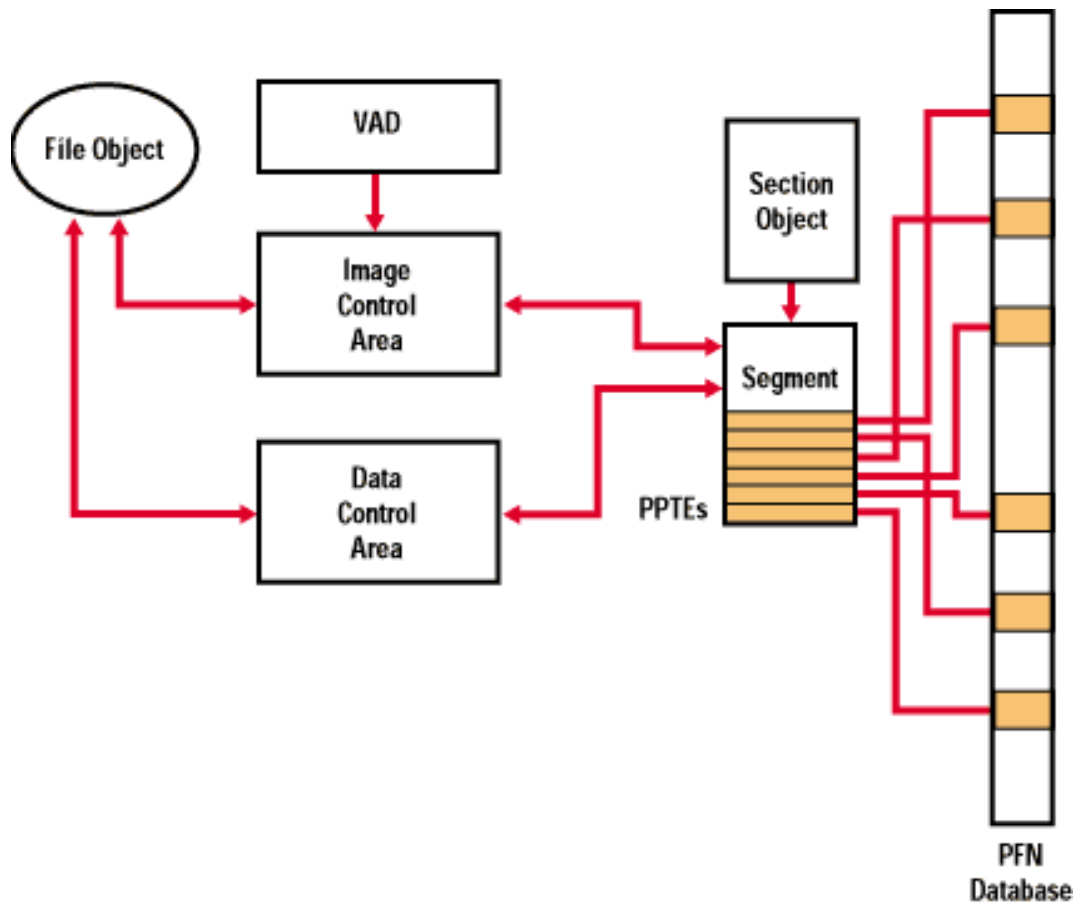
## Memory-Mapped Files

Memory-mapped files are a special form of shared memory. When a process maps a file into its address space, the Memory Manager creates several support data structures that aid the process' interaction with file systems. Figure 4 shows how these data structures are related. A File Object represents the file on disk, and the File Object is the target of all I/O the Memory Manager performs on the file. The Section Object points at its Segment, which contains the PTEs for the Section Object. The Segment points to a control area that the File Object also points at. This control area is the nerve center for a mapped file, so that even if a process creates more than one Section Object for a file, there is still only one shared control area.

# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)



When a process references a virtual address that the process indicates should be backed by a file, the Memory Manager examines the Virtual Address Descriptor (VAD) that describes the memory range, then locates the control area. The Memory Manager allocates a page of physical memory to bring the requested data into and updates the PTE for the virtual address map. Because the Memory Manager finds the File Object through the control area, the Memory Manager can initiate an I/O operation to the file system that owns the disk on which the file resides. The operation reads the page from the file on disk (the page is in the transition state while the I/O is in progress). After the operation reads the page, the Memory Manager marks the PTE as valid and lets the process continue its attempt to access the data, which is now present.

A caveat with regard to memory-mapped files exists: Files can map as data files or as images. Files map as images when the NT Process Manager loads them for execution. The same file can map as a data file and an image, and maintaining separate control areas for data files and images lets NT ensure the consistency of the different mappings that different processes make.

## More on Memory

Last month, I claimed that memory management is one of the most complex tasks an OS faces. In this two-part series on memory management, I've provided only an overview of the policies and mechanisms NT implements to provide applications with memory resources appropriate to their needs and the needs of other concurrently running programs. If you want to learn more about the Memory Manager, I recommend *Inside Windows NT, Second Edition*, by David A. Solomon (Microsoft Press).

# Inside Memory Management

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

Next month, I'll cover a subsystem that's closely tied to the Memory Manager--the Cache Manager. I'll discuss how the Memory Manager sizes system working sets (including the file system cache) differently from process working sets.