The Virtual-Memory Manager in Windows NT

Randy Kath
Microsoft Developer Network Technology Group

Created: December 21, 1992

Abstract

This article provides an in-depth survey of the memory management system in Windows NT™. Specifically, these topics are explored in detail:

- Virtual memory in Windows NT
- 32-bit virtual addresses
- Page directory, page tables, and page frames
- Translating a virtual address
- Process integrity
- Reserved and committed memory
- Translation lookaside buffers
- Page-table entry structure
- Page faults
- Sharing pages across process boundaries
- Prototype page-table entries
- Copy-on-write page optimization
- Characteristics of the virtual-memory manager
- The page-frame database
- Managing a working set of pages for each process

This article does not discuss the Win32™ memory management application programming interface (API). Instead, several other technical articles on the Microsoft Developer Network CD should be referenced for issues related to understanding how to manage memory with the Win32 API. Those articles provide both insight into the system and understanding of the functions themselves. While this article primarily deals with Windows NT-specific memory management issues, it does refer to some of the memory objects in the Win32 subsystem (like memory-mapped files and dynamic heaps) in an attempt to shed some light on the age-old dilemma of performance vs. resource usage as it applies to applications written for the Win32 subsystem in Windows NT.

Introduction

As the size of applications and the operating systems that run them grow larger and larger, so do their demands on memory. Consequently, all modern operating systems provide a form of virtual memory to applications. Being the newest of the operating systems to hit the main stream, Windows NT™ will likely have applications ported to it that will evolve into larger monstrosities that require even more memory than they did on the last operating system on which they ran. Even applications being written exclusively for Windows NT will be written with the future in mind and will no doubt take advantage of all the memory that is available to them.

Fortunately, Windows NT does, in fact, offer virtual memory to its applications (or processes) and subsystems. Windows NT provides a page-based virtual memory management scheme that allows applications to realize a 32-bit linear address space for 4 gigabytes (GB) of memory. As a result, each application has its own private address space from which it can use the lower 2 GB—the system reserves the upper 2 GB of every process's address space for its own use.
The Virtual-Memory Manager in Windows NT (Memory Technical Art... http://msdn.microsoft.com/library/en-us/dngenlib/html/msdn_ntvmm.a...}

Figure 1. A process in Windows NT has a 4-GB linear address space, of which the lower 2 GB is available for applications to use.

As illustrated in Figure 1, each process can address up to 4 GB of memory using 32-bit linear addresses. The upper half of the address space is reserved for use by the system. Because the system is the same for each process, regardless of the subsystem it runs on, similar pages of system memory are typically mapped to each process in the same relative location for efficiency.

Note The Win32™ subsystem provides user services that are loaded as dynamic-link libraries (DLLs) into the lower portion of the address space of a process. These DLLs exist in addition to the system DLLs that occupy the upper portion of the address space. Depending on which DLLs an application links or loads, the DLLs that are mapped into the lower portion of a process’s address space will vary from one application to the next within a subsystem.

If only we had PCs with similar memory capacities... Actually, a computer doesn’t really need 4 GB of physical memory for Windows NT to operate effectively—though the general rule of virtual memory systems is the more physical memory, the better the performance. Windows NT’s memory management system virtualizes memory such that to each application it appears as though there is 2 GB of memory available, regardless of how much physical memory actually exists. In order to do this, Windows NT must manage memory in the background without regard to the instantaneous requests that each application makes. In fact, the memory manager in Windows NT is a completely independent process consisting of several threads that constantly manage available resources.

Windows version 3.x has realizable limitations to the maximum amount of memory available to it and all of its applications; these are often barriers to large applications for this environment. Windows NT’s limits are far more theoretical. Windows NT employs the PC’s hard disk as the memory-backing store and, as such, has a practical limit imposed only by available disk space. So, it is reasonable to assume that a Windows NT system could have an extremely large hard disk or array of disks amounting to 2 GB or more of physical memory and provide that much virtual memory to each of its applications (minus the portions used by the system, occupied by the file system, and allocated by files stored within the file system). In short, Windows NT provides a seemingly endless supply of memory to all of the applications running on it.

Virtual Memory in Windows NT

The virtual-memory manager (VMM) in Windows NT is nothing like the memory managers used in previous versions of the Windows operating system. Relying on a 32-bit address model, Windows NT is able to drop the segmented architecture of previous versions of Windows. Instead, the VMM employs 32-bit virtual addresses for directly manipulating the entire 4-GB process. At first this appears to be a restriction because, without segment selectors for relative addressing, there is no way to move a chunk of memory without having to change the address that references it. In reality, the VMM is able to do exactly that by implementing virtual addresses. Each application is able to reference a physical chunk of memory, at a specific virtual address, throughout the life of the application. The VMM takes care of whether the memory should be moved to a new location or swapped to disk completely independently of the application, much like updating a selector entry in the local descriptor table (LDT).
Windows versions 3.1 and earlier employed a scheme for moving segments of memory to other locations in memory both to maximize the amount of available contiguous memory and to place executable segments in the location where they could be executed. An equivalent operation is unnecessary in Windows NT's virtual memory management system for three reasons. One, code segments are no longer required to reside in the 0-640K range of memory in order for Windows NT to execute them. Windows NT does require that the hardware have at least a 32-bit address bus, so it is able to address all of physical memory, regardless of location. Two, the VMM virtualizes the address space such that two processes can use the same virtual address to refer to distinct locations in physical memory. Virtual address locations are not a commodity, especially considering that a process has 2 GB available for the application. So, each process may use any or all of its virtual addresses without regard to other processes in the system. Three, contiguous virtual memory in Windows NT can be allocated discontiguously in physical memory. So, there is no need to move chunks to make room for a large allocation.

The foundation for the system provides the answer to how VMM is able to perform these seemingly miraculous functions. VMM is constructed upon a page-based memory management scheme that divides all of memory into equal chunks called pages. Each page is 4096 bytes (4K) in size with no discrimination applied as to how a page is used. Everything in Windows NT—code, data, resources, files, dynamic memory, and so forth—is implemented using pages of physical memory.

Because everything in the system is realized via pages of physical memory, it is easy to see that pages of memory become scarce rather quickly. VMM employs the use of the hard disk to store unneeded pages of memory in one or more files called pagefiles. Pagefiles represent pages of data that are not currently being used, but may be needed spontaneously at any time. By swapping pages to and from pagefiles, the VMM is able to make pages of memory available to applications on demand and provide much more virtual memory than the available physical memory. Also, pagefiles in Windows NT are dynamic in size, allowing them to grow as the demands for pages of memory grow. In this way, Windows NT is able to provide virtually unlimited memory to the system.

**Note** A detailed discussion on how the virtual-memory manager performs the functions mentioned here is presented later in this article in the section "The Virtual-Memory Manager (VMM)."

### 32-Bit Virtual Addresses

One of the conveniences of the 32-bit linear address space is its continuity. Applications are free to use basic arithmetic on pointers based on 32-bit unsigned integers, which makes manipulating memory in the address space relatively easy. Though this is how addresses are viewed by an application, Windows NT translates addresses in a slightly different manner.

To Windows NT, the 32-bit virtual address is nothing more than a placeholder of information used to find the actual physical address. Windows NT separates each 32-bit virtual address into three groups. Each group of bits is then used independently as an offset into a specific page of memory. Figure 2 shows how the 32-bit virtual address is divided into three offsets, two containing 10 bits and one containing 12.

![Figure 2. A 32-bit virtual address in Windows NT is divided into page offsets that are used for translating the address into a physical location in memory.](image)

### Page Directory, Page Tables, and Page Frames

The first step in translating the virtual address is to extract the higher-order 10 bits to serve as the first offset. This offset is used to index a 4-byte value in a page of memory called the page directory. Each process has a single, unique page directory in the Win32 subsystem. The page directory is itself a 4K page, segmented into 1024 4-byte values called page-directory entries (PDEs). The 10 bits provide the exact number of bits necessary to index each PDE in the page directory (210 bits = 1024 possible combinations).

Each PDE is then used to identify another page of memory called a page table. The second 10-bit offset is subsequently used to index a 4-byte page-table entry (PTE) in exactly the same way as the page directory does. PTEs identify pages of memory called page frames. The remaining 12-bit offset in the virtual address is used to address a specific byte of memory in the page frame identified by the PTE. With 12 bits, the final offset can index all 4096 bytes in the page frame.
Through three layers of indirection, Windows NT is able to offer virtual memory that is unique to each process and relatively independent of available physical resources. Also, embedded within this structure is the basis for managing all of memory based on 4K pages. Every page in the system can be categorized as either a page directory, page table, or page frame.

**Realizing 4 GB of Address Space**

Translating a virtual address from page directory to page frame is similar to traversing a b-tree structure, where the page directory is the root; page tables are the immediate descendants of the root; and page frames are the page table’s descendants. Figure 3 illustrates this organization.

![Figure 3. Translating a virtual address is similar to traversing a b-tree structure.](image)

A page directory has up to 1024 PDEs or a maximum of 1024 page tables. Each page table contains up to 1024 PTEs with a maximum of 1024 page frames per page table. Each page frame has its own 4096 one-byte locations of actual data. All totaled, the 32-bit virtual address can be translated into 4 GB of address space (1024 * 1024 * 4096). Yet, there is still the question of the pages that are used to represent the page tables and page directory.

Looking closely at Figure 3 reveals that a considerable amount of overhead is required to completely realize all of the page frames in memory. In fact, to address each location in the 4-GB address space would require one page directory and 1024 page tables. Because each page is 4K of memory, 4 MB (approximately) of memory would be needed just to represent the address space ( [1024 page tables + 1 page directory] * 4096 bytes/page).

Although that may seem like a high price to pay, it really isn’t, for two reasons: First, 4 MB is less than 0.1 percent of the entire 4-GB address space, which is a reasonably small amount of overhead when you consider comparable operating systems. Second, Windows NT realizes the address space as it is needed by the application, rather than all at once, so page tables are not created until the addresses they are used to translate are needed.

**Translating a Virtual Address**

Specifically, how does Windows NT translate a 32-bit virtual address into a specific memory location? As an example, take a look at an address in the process of any Win32-based application. This is easily performed by running the Windows NT debugger, WINDBG.EXE. Simply load an application, step into the `WinMain` code for the application, and choose to view local variables. A typical address is 043612FF16 or 0000 0100 0011 0110 0001 0010 1111 11112.

The first 10-bit offset is 00 0001 00002 or 0x01016. Shift the bits left two spaces to form a 12-bit value where the lower-order significant bits are padded with zeros, resulting in the value 0000 0100 00002 or 0x06016. The bit shifting provides an easy mechanism for indexing the page on 4-byte boundaries. Use this value to index into the 4K page directory. The 4-byte PDE at this location identifies the page table for this address.

Repeat this method for the second 10-bit sequence, using the page table instead of the page directory. The index shifted left two spaces is 1101 1000 01002 or D8616. The PTE at this index identifies the specific page frame. To index a one-byte location in the page frame, use the final 12 bits just as they are. Figure 4 demonstrates this process pictorially.
The Virtual-Memory Manager in Windows NT (Memory Technical Art... http://msdn.microsoft.com/library/en-us/dngenlib/html/msdn_ntvmm.a... sur 12 17/05/2006 12:27

Figure 4. Pages of memory are used to represent the page directory, page tables, and page frames for each process.

Individual Process Integrity

Because every process has its own page directory, Windows NT is able to preserve the integrity of each process's address space. This is both good news and bad news for those programming applications destined for the Windows NT operating system. The good news is that an application is secure against unwarranted, perhaps accidental, intrusion. Also, as stated earlier, you are free to use as much of the address space as you need without regard to the impact it has on other processes.

Now for the bad news. Because two processes can use the same virtual address to refer to different locations in memory, processes are not able to communicate addresses to one another. This makes sharing memory much more difficult than in the past and requires that you use specific mechanisms to share memory with other processes. For more information on sharing memory in Windows NT, refer to the section "Sharing Pages Across Process Boundaries" later in this article.

There is more bad news when it comes to stray pointers. Having exclusive access to your own process means that you own the entire space (that is, the lower 2 GB). Consequently, if you have a stray pointer, it is much more difficult to detect. For example, a pointer that runs past the bounds of a designated array only exists in your own address space. Whether you actually committed that memory is another question, but you can be certain that it won't pounce on another process's memory. A stray pointer can point to two things: either a memory location you have committed for some other purpose or an invalid address (one that memory has not been committed to). The latter case generates an access violation exception, which you can handle easily enough through structured exception handling. The former case will simply result in a successful read or write operation. So, there is no way of knowing that a problem even occurred.

Reserved vs. Committed Memory

In Windows NT, a distinction exists between memory and address space. Although each process has a 4-GB address space, rarely if ever will it realize anywhere near that amount of physical memory. Consequently, the virtual-memory manager must keep track of the used and unused addresses of a process, independent of the pages of memory it is actually using. In actuality this amounts to having a structure for representing all of the physical memory in the system and a structure for representing each process's address space.

As part of the process object (the overhead associated with every process in Windows NT), the VMM stores a structure called the virtual address descriptor (VAD) tree to represent the address space of a process. As address space gets used for a process, the VMM updates the VAD tree to reflect which addresses are used and which are not. Fortunately, Windows NT recognizes the value of managing address space independent of memory and extends this capability to subsystems. Further, the Win32 subsystem provides this capability through the VirtualCreate API. With this function, applications can reserve a range of addresses for use at a later time.

Many parts of the Win32 subsystem make use of the reserved memory feature. Take stack space, for example. A Win32-based application can reserve up to 1 MB of memory for the stack while actually committing as little as 4K of
space initially. Memory-mapped files represent another example of reserving a range of addresses for use at a later time. In this case, the address range is reserved with the function `MapViewOfFile`. This permits applications to map a large file (it is possible to load a file 1 GB in size in Windows NT) to a specific range of addresses without having to load the entire file into memory. Instead, portions (views) of the file can be loaded on demand directly to the reserved address space.

One other benefit of reserving address space resides in the fact that the addresses are contiguous by default. Windows NT uses this technique for mapping each application’s code and DLL files to specific addresses. Because the content of code is executed sequentially, the address space that references it must also be contiguous. By reserving address space, Windows NT need load only the pages that are being used; the rest are loaded on demand.

Another nice feature of reserved memory is that Windows NT can reserve address space for page tables, as well as other pages. From a more global perspective, this means that the 4 MB of memory required to realize a process’s 4-GB address space is also not committed until needed. A possible drawback to this feature is that a process could generate a page fault more than once on a single address translation. More discussion on page faults is presented below in the section “Page Faults.”

**Translation Lookaside Buffers (TLBs)**

Considering all the work Windows NT does to retrieve the physical address of a page, addressing in general seems a little on the inefficient side. After all, to translate a single virtual address, the VMM must access memory in three physical pages of memory. However, Windows NT uses another addressing scheme in parallel with the virtual address translation technique described above.

Windows NT exploits the capability of modern CPUs by putting the translation lookaside buffer (TLB) to use. The TLB (often referred to as the internal or on-chip cache) is nothing more than a 64K buffer that is used by hardware to access the location of a physical address. Specifically, Windows NT uses the TLB to provide a direct connection between frequently used virtual addresses and their corresponding page frames. Using it, the VMM is able to go from a virtual address directly to the page frame, thereby avoiding translation through both the page directory and page table.

Because the TLB is a hardware component, its contents can be searched and compared completely in parallel with the standard address translation performed in software. So, the time saved in translating a virtual address comes without a tradeoff. Also, being a hardware mechanism, it is extremely fast in comparison to the software translation described earlier. A big win all the way around. Too bad there isn't room for more than 32 entries! Being a hardware component also has its limitations.

Each entry in the TLB consists of a virtual address and a corresponding PTE to identify the page frame. Consecutively addressing two locations in the same page of memory generates only one entry in the TLB in order to reduce redundancy and save precious TLB space. Every time an address is translated in software and references a new page frame, an entry is added to the TLB. Once the TLB is full, every new entry requires a previous entry to be dropped from the buffer. The algorithm for dropping entries from the TLB is simple: drop the least recently used page from the list.

**Flushing the Buffer on Context Switches**

Although Windows NT appears to run all threads concurrently, in actuality only one thread can execute at any given time. So, Windows NT schedules each thread to execute for a small amount of time, often referred to as a time slice. When the time slice expires, Windows NT schedules the next thread to run during its slice. This continues until all threads have had a chance to execute during their slice. At that time, Windows NT schedules the first thread again, repeating this process indefinitely. The act of stopping a thread at the end of its time slice and scheduling another thread is called context switching.

When a thread is running, it spends much of its time translating addresses and relies on the TLB to make this as fast as possible. Yet, a single context switch can render the TLB useless. The address translations of one thread will be incorrect for another thread unless the second thread is a thread of the same process. The chances of the thread being from the same process are remote at best, and even if it were from the same process, addresses used by the second thread are likely to be entirely different from those of the first.

Consequently, the buffer is automatically flushed when context switching between threads on the Intel platform—a hardware feature. The millions-of-instructions-per-second (MIPS) implementation of Windows NT does not flush the buffer during context switches, but it does provide a 36-bit address space that Windows NT makes use of for this purpose. Windows NT uses the extra four address bits to identify which process is responsible for each TLB entry.

When you consider the rate at which Windows NT performs context switches, it seems at first that this action would nullify any gains the TLB could offer. In actuality, though, consider that the duration between context switches is on the
order of 17 milliseconds. On a machine capable of 5 million instructions per second (a typical Intel-based 80386 33MHz is on the order of 10-15 MIPS), this will amount to 85,000 instructions per context switch (17 milliseconds * 5 MIPS). Then, take into account the fact that on average an application attributes 30–50 percent of its time addressing memory, and you get 25,500–42,500 address instructions per context switch. Also, the TLB will become completely full after executing 32 address instructions that reference a unique page of memory. So, the TLB will likely be refilled very soon after the context switch occurs so that it again becomes useful for nearly all address instructions.

**Associative Buffer Limitation**

One limitation exists in the translation lookaside buffer due to its tight integration with hardware. Because there are only 32 entries in the TLB, and each of these entries must be able to map to physical addresses, there is some restriction as to which address each entry can contain. Because of this, some overlap exists, creating the possibility that the each of TLB entries capable of containing a specific address could all be in use at one time. In that case, accessing a page of memory that can only be represented in one of these entries forces the least recently used entry to be dropped from the list.

The worst-case scenario for this situation is that repetitively executing code at five different addresses that can be located in only four TLB entries results in complete address translation for all five addresses. What happens is each new address translation ends up replacing the least recently used entry in the TLB. Then, if the next of these five addresses is the one that was just replaced, it must be translated over again because it was just replaced in the list. This is repeated for each of the five addresses in a cyclical fashion, making the TLB useless. Although the possibility of this type of occurrence does exist, it has proven an extremely rare occurrence.

**Page-Table Entry Structure**

Address translation has two aspects: breaking a virtual address into three offsets for indexing into pages of memory and locating an actual physical page of memory. The page directory and page table entries mentioned earlier are used for this purpose.

Once memory is committed for a range of reserved addresses, it exists as either a page in random access memory (RAM) or in a pagefile on disk. A page-table entry identifies the location of the page, its protection, its backing pagefile, and the state of the page, as shown in Figure 5.

![Page-Table Entry](image)

**Figure 5. Page-table entries are used to provide access to physical pages of memory.**

The first 5 bits are dedicated to page protection for each page of memory. The Win32 API exposes PAGE_NOACCESS, PAGE_READONLY, and PAGE_READWRITE protection to applications written for the Win32 subsystem.

Following the protection bits are 20 bits that represent the physical address of the page in memory if it is resident. Note that these 20 bits can address any 4K page of memory in the 4-GB address space (220 * 4096 = 4 GB). If the page of memory is paged to disk, the twenty address lines are used as an offset into the appropriate pagefile to locate the page.

The next four bits are used to indicate which pagefile backs this page of memory. Each of the 16 possible pagefiles can be uniquely identified with these four bits.

The final three bits indicate the state of the page in memory. The first bit is a flag indicating pages in transition (T); the second indicates dirty pages, pages that have been written to but not saved (D); and the third indicates whether each page is present in memory (P). The state table below represents the possible states of a page.

**Table 1. Page-Table Entry Page States**

<table>
<thead>
<tr>
<th>T</th>
<th>D</th>
<th>P</th>
<th>Page state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Invalid page</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>1</td>
<td>Valid page</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>1</td>
<td>Valid dirty page</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Invalid page in transition</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Invalid dirty page in transition</td>
</tr>
</tbody>
</table>
When a page is not present and not in transition, the dirty bit is ignored. Also, only when a page is present, the transition bit is ignored.

The above description of a PTE applies to all pages of memory that are backed by one of the 16 pagefiles. Yet, in Windows NT, not all pages of memory are backed by these pagefiles. Instead, Windows NT backs pages of memory that represent either code or memory-mapped files with the actual file they represent. This provides a substantial savings of disk memory by eliminating redundant information on the disk. When a page of this type is present in memory, the PTE is structured just as described above for present pages and pages in transition. When a page is not present in memory, the PTE structure changes to provide 28 bits that can be used for addressing an entry in a system data structure. This entry references the name of a file and a location within the file for the page of memory. To get 28 bits in the PTE, the four pagefile bits and four of the protection bits are sacrificed, while the three state bits remain intact.

Page Faults

When Windows NT addresses an invalid page (that is, during the course of address translation one of the PTEs identifies a page as not present), a page-fault exception is raised by the processor. A page fault results in switching immediately to the pager. The pager then loads the page into memory and, upon return, the processor re-executes the original instruction that generated the page fault. This is a relatively fast process, but accumulating many page faults can have a drastic impact on performance.

It is possible that during translation of a single virtual address, as many as three page faults can occur. This is due to the fact that, in the worst case, a virtual address may be realized only by accessing a page directory, page table, and page frame where none of these pages is present in memory and each generates a separate page fault when accessed.

This is one instance in which the TLB can really improve performance of memory addressing by avoiding the page fault associated with loading a page table and page directory, reducing multiple fault possibilities to at most a single page fault. A similar reduction of page faults from two to one occurs when, during translation, a page directory is present but the page table and page frame are not. In both cases, the page frame is not present, so one page fault is required to retrieve the page frame. On the other hand, if the page frame is present but the page table and page directory are not, two page faults are avoided.

Sharing Pages Across Process Boundaries

Given that each process has its own page directory, sharing memory between processes is anything but trivial. Because the page directory is the root of an address as shown in Figure 4, an address in one process is essentially meaningless in the context of another process. If, in fact, an address in one process is a valid address in a second process, it is, at best, a coincidence. Yet on the other hand, there is no physical barrier preventing page tables from two processes from having identical PTEs that point to the same page frame. So, it is feasible for Windows NT to provide memory sharing in this way.

However, there is one glaring inefficiency in this scheme: What happens when the state of a shared page is changed? Say, for example, that a shared page is written to by one of four processes sharing that page. Then the system would have to update the four PTEs, one entry in the page table for each of the four processes. This would not only be an expensive performance hit on a single write, but there is no way of determining which page tables reference a specific page frame. There would have to be some type of overhead put in place that reverse-referenced all PTEs that reference a shared page frame.

Prototype PTEs

For Windows NT, a better implementation than the scheme described above was chosen for sharing memory. Rather than having multiple PTEs point to the same physical page, another layer of page tables was put in place exclusively for shared memory. When two or more processes share a page of memory, an additional structure called a prototype page-table entry is used to reference the shared page. Each process's PTE points to the prototype PTE, which, in turn, points to the actual shared page. The prototype PTE is also a 32-bit quantity that directly references the page frame of the shared page. Figure 6 illustrates this new indirect approach.
Prototype page-table entries are used to share pages of memory between processes.

**Performance Hit on Prototype PTEs**

Prototype page-table entries are not without their own performance hit. They are implemented as a global system resource mapped into the upper address space of all processes. A maximum of 8 MB of space is reserved for use by the system to support a prototype PTE data structure for all shared pages. Prototype PTEs are allocated dynamically as they are needed by the system, so no memory is wasted on supporting nonexistent shared pages. The biggest performance hit occurs in accessing one of these nonexistent shared pages. The additional layer of indirection means that translating a virtual address to a shared page could mean as many as four page faults, instead of three.

Shared memory in Windows NT has several uses, the most common of which is code sharing. Code is shared by default so that running multiple instances of an application reuses as much of the existing resources as possible. Also, memory-mapped files are implemented as shared memory. Finally, the Win32 subsystem provides a feature that enables processes to share data via a DLL. For more information on the memory sharing capabilities provided in the Win32 subsystem, refer to the article "Managing Memory Mapped Files in Win32" on the Developer Network CD (Technical Articles, Win32).

**Copy-on-Write Optimization**

By default, all code pages have PAGE_READWRITE protection in Windows NT. This characteristic makes life easy for applications like debuggers. Because a debugger can write code pages, it is relatively easy to embed break points and single-step execution instructions in the code itself. Yet, this also raises another issue. What if the code being debugged was also being executed by another process simultaneously? The act of writing a break-point instruction to the code page would affect both the process being debugged and the other process. On the other hand, having duplicate copies of code, one for each instance of a process, would be redundant and wasteful.

The solution to this problem is in an optimization called **Copy-on-write**. I have already discussed how a prototype PTE is used for all code pages to make them capable of being shared among different processes. In addition to the shareability of code pages, Windows NT gives code pages another special characteristic that enables them to be copied, if necessary, and backed by the pagefile. Copying would only occur if and when a write ever occurred to a code page. The optimization resides in the fact that copying does not occur unless necessary, as determined by the act of writing to a page. Consequently, only pages that are written to are copied, saving precious memory resources.

**The Virtual-Memory Manager (VMM)**

The virtual-memory manager in Windows NT is a separate process that is primarily responsible for managing the use of physical memory and pagefiles. To do this, it must track each page of physical memory, tune the working set of all active processes, and swap pages to and from disk both on demand and routinely. The VM manager is an executive component of Windows NT that runs exclusively in kernel mode. Because of the time-critical nature of the code that is executed by the virtual-memory manager, the VMM code resides in the small section of memory called **nonpaged pool**. This memory is never paged to disk.
The Page-Frame Database

The virtual-memory manager uses a private data structure for maintaining the status of every physical page of memory in the system. The structure is called the page-frame database. The database contains an entry for every page in the system, as well as a status for each page. The status of each page falls into one of the following categories:

- **Valid**: A page in use by an active process in the system. Its PTE is marked as valid.
- **Modified**: A page that has been written to, but not written to disk. Its PTE is marked as invalid and in transition.
- **Standby**: A page that has been removed from a process's working set. Its PTE is marked as invalid and in transition.
- **Free**: A page with no corresponding PTE and available for use. It must first be zeroed before being used unless it is used as a read-only page.
- **Zeroed**: A free page that has already been zeroed and is immediately available for use by any process.
- **Bad**: A page that has generated a hardware error and cannot be used by any process in the system.

Most of the status types are common to most paged operating systems, but the two transitional page status types are unique to Windows NT. If a process addresses a location in one of these pages, a page fault is still generated, but very little work is required of the VMM. Transitional pages are marked as invalid, but they are still resident in memory, and their location is still valid in the PTE. The VMM merely has to change the status on this page to reflect that it is valid in both the PTE and the page-frame database, and let the process continue.

The page-frame database associates similar pages based on each page's status. All the pages of a given type are linked together via a linked list within the database; see Figure 7. These lists are then traversed directly according to status. This enables the VM manager to locate three pages marked Free, for example, without having to search the entire database independently for each Free page. Another way of thinking of the database entries is to consider them as existing in six independent lists, one for each type of page status.

![Figure 7. The page-frame database records the status of pages of physical memory.](http://msdn.microsoft.com/library/en-us/dnenlib/html/msdn_ntvmm.a...)

Each page-frame entry in the database also reverse-references its corresponding PTE. This is necessary so that the VM manager can quickly return to the PTE to update its status bits when the status of a page changes. The VM manager is also able to reverse-reference prototype PTEs to update their status changes, but note that the prototype PTE does not reverse-reference any of its corresponding PTEs.

The VMM uses the page-frame database any time a page of memory is moved in or out of memory or its state changes. Take, for example, a process that attempts to address a specific memory location in a page that had been paged to disk. The translation for this virtual address would generate a page fault as soon as an attempt to access the page referenced by the PTE occurred. The VMM would then allocate a physical page of memory to satisfy the request. Depending on the current state of the system, allocating a page may be as easy as changing the PTE for the page to Valid and updating the page-frame database for that page; such is the case for transitional pages as described above. On the other hand, the VMM may be required to steal a Modified page from another process; write the page to disk; update the PTE in the page table of the other process as not in transition; zero the page; read in the new page from a pagefile; update its PTE to indicate a valid page; and update the page-frame database to represent the physical page as Valid.
Periodically, the VMM updates the page-frame database and the state of transitional pages in memory. In an effort to keep a minimum number of pages available to the system at all times, the VMM moves pages (figuratively speaking) from either the Modified or Standby list to the Free list. Modified pages must be written to disk first and then marked as Free. Standby pages do not need to be written because they are not dirty. Free pages are eventually zeroed and moved to the Zeroed list. Pages in the Free and Zeroed lists are immediately available to processes that request pages of memory. Each time a page is moved from one list to the next, the VMM updates the page-frame database and the PTE for the page. It is important to note that pages in either of the transition states are literally in transition from Valid pages to Free pages.

Managing a Working Set of Pages for Each Process

Another part of the VMM gets pages into the transitional state. The thread that gets transitional pages must continually decide what data is most deserving of replacement on a process-by-process basis. The algorithm for deciding which page to replace is typically based on predicting the page that is least likely to be needed next. This prediction is influenced by factors such as what page was accessed least often and what page was accessed the longest time ago. In Windows NT, the component responsible for making these predictions is called the working-set manager.

When a process starts, the VMM assigns it a default working set that indicates the minimum number of pages necessary for the process to operate efficiently (that is, the least amount of paging possible to fulfill the needs of the process without starving the needs of other processes). The working-set manager periodically tests this quota by stealing Valid pages of memory from a process. If the process continues to execute without generating a page fault for this page, the working set is reduced by one, and the page is made available to the system. This test is performed indiscriminately to all processes in the system, providing the basis for the free pool of pages described above. All processes benefit from this pool by being able to allocate from it on demand.

The act of stealing a page from a process actually occurs in two stages. First, the working-set manager changes the PTE for the page to indicate an invalid page in transition. Second, the working-set manager also updates the page-frame database entry for the physical page, marking it as either Modified or Standby, depending on whether the page is dirty or not.

Conclusion

Developers for Windows NT face many new challenges on their way to becoming proficient with the operating system. Understanding how to manage memory effectively is likely to be one of the more difficult challenges—and probably the most important one. Figuring out whether to use virtual memory, memory-mapped files, heap memory, or space on a thread’s stack for implementing specific types of data is representative of the kind of decision that developers routinely face when developing applications for Windows NT. Such a decision depends on knowing how and when the operating system allocates specific resources—such as the virtual address descriptor (VAD) tree, the translation lookaside buffer (TLB), page tables and page-table entries (PTEs), prototype PTEs, the page-frame database, a process’s virtual address space, system pagefiles, and so on. A thorough knowledge of each of these system resources and how they are affected by specific application programming interface (API) functions is the key to mastering Windows NT.

This technical article identifies each of the components of the virtual memory management system in Windows NT, focusing on answers to these questions:

- How does the system go about its business?
- How do the components work together to get things done?
- When do things get done?

Understanding how and when the system does things and what the system resources are is only half the challenge. The other half lies in understanding how each of the API functions affects these resources. With hundreds of memory management functions available, the task is especially challenging. This technical article provides a foundation for three other technical articles on this disc that specifically address the memory functions available in the Win32 API and explain the impact each has on system resources: "Managing Virtual Memory in Win32," "Managing Memory Mapped Files in Win32," and "Managing Heap Memory in Win32" (MSDN Library, Technical Articles). You should examine this article first and then read the other three when you are faced with memory issues in a specific area of the Win32 API.