5

The NT Virtual Memory Manager

In this chapter:
* *Functionality*
• *Process Address Space*
• *Physical Memory Management*
"*Virtual Address Support*
* *Shared Memory and Memory-Mapped File Support*
• *Modified and Mapped Page Writer*
• *Page Fault Handling*
• *Interactions with File System Drivers*

An important functionality provided by modern day operating systems is the management of physical memory on the node. Typically, the amount of available volatile RAM on a machine is less than that required by all the applications and by the operating system itself. Therefore, the operating system has to intervene and facilitate sharing this limited memory resource, given the often conflicting demands placed by all components on the node.

Furthermore, with multiple applications executing concurrently on the same machine, the operating system has the task of ensuring that these applications can perform their tasks independently of each other. Therefore, code and data structures for each of the applications must be managed such that they do not interfere with code and data from any other application. The operating system must also protect its own in-memory resources (both code and data), used to manage the system, from all of the applications executing on the system. This is required to guarantee the integrity and security of the machine itself. Finally, sophisticated applications on the same machine (and sometimes on networked clusters of machines) often need to share in-memory data with each other. The operating system has to facilitate orderly sharing of data such that only those applications that are given permission to access the shared data are allowed to do so.

The Virtual Memory Manager (VMM) has the responsibility for providing all of this functionality. The VMM is so named because it helps provide an abstraction to each application executing on the system: each application can perform its tasks believing all of the memory resources on that system are available for the sole use of that application. Furthermore, the application can execute believing that it has

an infinite amount of memory resource available to it. This abstraction of an infinite amount of memory reserved solely for the use of a specific application is called virtual memory. The VMM is the kernel-mode component responsible for providing this abstraction.

Functionality

The NT VMM provides the following functionality to the other components of the system:

- The Virtual Memory Manager provides a demand-paged (with clustering support), virtual memory system. Each process has a private virtual address space associated with it.* This virtual address space is backed by physical pages, allocated on demand from the total pool of physical pages available on the machine.
- Management of the virtual address space associated with a process is separated from the manipulation of physical pages. The VMM provides support for application control of the virtual address space allocation, commitment, manipulation, and deallocation.
- Virtual memory support is provided with the help of the local file systems. In order to provide the illusion of a large amount of available memory (greater than the amount of actual RAM), the contents of memory are backed-up to storage allocated on secondary storage media. Memory backed by on-disk storage is called *committed memory.* Committed memory is backed either by page files, which can be dynamically resized, or by data/image files on secondary storage.
- The VMM provides support for memory-mapped files. These files can be arbitrarily large; files larger than 2GB can be mapped using partial *views* of the file.
- It supports sharing memory between different processes on the system. This is also used as a method of interprocess communication.
- It implements per-process quotas.
- It determines the policies for working-set management of physical memory allocated to processes.

---

* Currently (for Windows NT Version 4.0 and earlier), the Virtual Address Space associated with a process is limited to 4 GB. It is inevitable that this support will be extended to 2^{64} bytes when Windows NT becomes a true 64-bit operating system. At the time this book went to press, Microsoft announced its intention to support a 64-bit address space with Version 5.0 of the operating system.

Furthermore, all physical memory allocation/deallocation decisions are performed by the NT VMM. This is done irrespective of whether memory is allocated to user-mode applications or for kernel-mode file data caching.

- It provides support for the protection of memory using Access Control Lists (ACLs).
- It provides support for the POSIX *fork* and *exec* operations, thereby enabling compliance with the POSIX standard.
- It provides support for *copy-on-write* pages. It has the ability to establish guard pages and to set page level protection.*

Process Address Space

An address is simply a value that points to a memory location contained within

In this chapter:
- *I/O Revisited: Who Called?*
- *Asynchronous I/O Processing*
- *Dispatch Routine: File Information*
- *Dispatch Routine: Directory Control*
- *Dispatch Routine: Cleanup*

10

Writing A File System Driver II

on the machine is typically much less than 4GB. Assume that each element of the virtual address space really mapped to an equivalent physical address. Therefore, each system would have to install 4GB of memory for each process that would run on that system. Instead, the VMM tricks each process into believing that it has 4GB of addressable memory available for its usage. The process takes this proposition at face value and tries to access memory using the range of available virtual addresses. It is the responsibility of the VMM to translate (or map) each virtual address into a corresponding physical address.

Why does a process need to access memory? In order to provide any useful functionality, each process has most of the following components associated with it:

Close

- Uninitialized global data
- Heap (dynamically allocated memory)
- Shared memory
- Shared libraries

These components must be stored somewhere in physical memory, though not all of these components need to always exist in physical memory all of the time. If these components are brought into physical memory when needed, the process must have some way of accessing the memory locations where this information is stored. Therefore, an address space (or a range of addresses) must be associated with each process.

Some amount of physical memory in the system must also be devoted to the operating system code and data. So now, not only does the VMM have to provide virtual addresses for process-specific information, it must provide virtual addresses to refer to operating system components, including addresses that refer to its own code used to manage the memory in the system. This is achieved by the creation of a special system process, which, like any other process, has 4GB of virtual memory available to it. However, creation of the system process is not sufficient in itself.

You know that user processes executing on the system often have to request services from the operating system. These requests might be related to I/O operations, allocation and manipulation of memory, creation of new processes and other similar operations. The NT operating system provides system services that receive user requests and execute code in kernel mode to handle such requests. This leads to the situation where operating system code executing in kernel mode must perform some tasks in the context of the user process that issued the request.

Figure 5-1 illustrates a typical process address space on an Intel x86 hardware platform.*

To perform such tasks, operating system code and data must be addressable from within the context of the user process; i.e., there must be a range of virtual addresses that refer to operating system code and data, but actually "reside" within the 4GB limit set by the underlying hardware. To achieve this, the NT VMM divides the 4GB range of addresses allocated to each process into two halves, a 2GB range dedicated to user-mode virtual addresses, called *user space* (a

* Some hardware platforms support a segmented addressing scheme, which divides physical memory into contiguous chunks known as *segments.* However, the view presented by the NT VMM to the entire system is that of a linear (or a *flat)* virtual address space. Any segmentation issues (if they exist on a particular architecture) are handled transparently by the VMM.

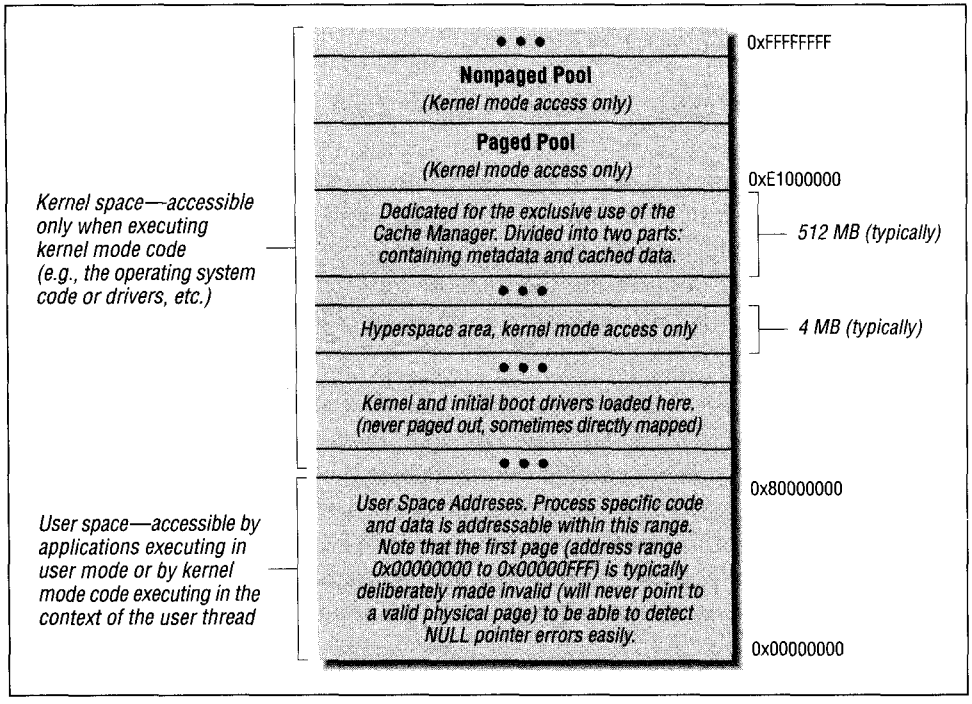

Figure 5-1. Virtual address space for a typical process

user process executing in user mode can only access this 2GB range) and another 2GB range containing kernel-mode virtual addresses, called *kernel space.*

NOTE Let me reiterate the following concept: a processor can execute code in user mode (typically *Ring 3* on Intel x86 architectures) or it can execute code in kernel/privileged mode (typically *Ring 0* on Intel x86 architectures). Therefore, although stated otherwise, user-mode or kernel-mode states are associated with a processor, not with any code (or process) executing on that processor. The distinction, though subtle, must be well understood by all kernel developers.

Although the 2GB of user-mode virtual addresses refer to process-private data (not accessible by other processes in the system), the 2GB of kernel-mode addresses always refer to the same physical pages* on the system (regardless of the thread context in which they are accessed) containing operating system code and data.

---

* Pages are explained later in this chapter.

Another concept that you should understand is that of a *hyperspace area* within the 4GB virtual address space associated with each process. This hyperspace area is a range of virtual addresses actually reserved from within the 2GB kernel space area, but specially designated, since it typically contains process-specific internal data structures maintained by the NT VMM. Whenever a context switch occurs, the VMM refreshes this virtual address space to refer to information specific to the new process. These data structures include page table pages for the process, w, and other such VMM data structures.

If you develop kernel-mode drivers, you always have to be aware of the thread context in which your code operates. For example, if you design a file system driver, your dispatch entry points will typically be executed in the context of the user process that invoked the corresponding system call.* If this is the case, your driver can use addresses (passed in the IRP) within the lower 2GB of the process's virtual address space to refer to user-space memory (e.g., user buffers). However, if you write intermediate or lower-level drivers (e.g., device drivers), your dispatch routines will typically be invoked in the context of an arbitrary thread, defined as the thread that is executing on that processor at that particular time. In this case, you cannot assume that any user-space virtual addresses that might be contained in the I/O Request Packet are still valid, because your code is not executing in the context of the user thread originating the request; hence the lower 2GB of the virtual address space now map to physical pages belonging to some other process.

On the other hand, if you develop a kernel-mode driver and allocate memory, the returned memory pointer will typically be a virtual address in kernel space. Since the kernel-space virtual address is the same for all processes in the system, the allocated memory can be referred to (using the returned pointer) in the context of any thread in which your code might be executing.

How do you ensure that a user-space buffer pointer passed in via an IRP is accessible from within your driver, if code within your driver might execute in an arbitrary thread context? The VMM provides support to map user-space memory into kernel virtual address space precisely for this purpose (MmGetSystemAddressForMdl( )).t

One final point: it might be necessary for your kernel-mode driver to occasionally access the virtual address space of some other process. One of the ways that this

---

* This is not always true. Sometimes, the subsystem (e.g., the Win32 subsystem) will invoke file system entry points in the context of a worker thread belonging to some Win32-specific process. Furthermore, the NT VMM and the NT Cache Manager often originate calls into the file system read/write routines in the context of a thread belonging to the system process.

t This routine is described in further detail later in this chapter.

can be accomplished is by using the KeAttachProcess() kernel support routine. This routine is not documented in the Windows NT DDK, but is defined as follows:

```
VOID
KeAttachProcess (
    IN PEPROCESS        Process
);
```

Parameters:

Process

A pointer to the process you wish to attach to. This can be obtained by an invocation to loGetCurrentProcess().

Functionality Provided:

The KeAttachProcess() call allows your kernel-mode thread to attach itself to the target process. Then your thread will execute in the context of that process, allowing it to access the entire virtual address space and all other resources belonging to that process.

NOTE A reason you might wish to access the virtual address space of another process is if memory had been mapped into the virtual address space of the target process and you need to access it. Another reason is if you need to use any resources (e.g., file handles) that belong to the target process.

Be very careful though, since attaching to another process is an extremely expensive operation and will result in two context switches, at the very least, if the target process has been swapped out. Furthermore, it will probably result in flushing of *Translation Lookaside Buffers* on all processors in a symmetric multiprocessor (SMP) system, which can be detrimental to system performance.

Do not invoke this routine at an IRQL greater than DISPATCH_LEVEL. An executive spin lock used to protect internal data structures in the implementation of this routine is acquired at IRQL DISPATCH_LEVEL; therefore, invoking this function at a higher IRQL could lead to a deadlock scenario. Also, do not attempt to attach to a second process if you have already invoked the KeAttachProcess() function without invoking the corresponding detach routine, described next, or a bugcheck will occur.

The corresponding routine to detach from a process to which your thread is attached is defined as follows:

```
VOID
KeDetachProcess (
```

```
    VOID
);
```

Parameters:

None.

Functionality Provided:

The KeDetachProcess() function allows your kernel mode thread to detach itself from a previously attached process. Do not invoke this routine at an IRQL greater than DISPATCH_LEVEL.

Physical Memory Management

To write a kernel-mode driver (especially a file system driver), it helps to broadly understand the method used by the memory manager to manage physical memory. Once you understand how physical memory is manipulated, I will describe how virtual addresses are mapped to physical addresses. This knowledge can be invaluable when debugging NT systems and when attempting to understand why certain things work the way they do.

Page Frames and the Page Frame Database

The NT VMM must manage the available physical memory in the system. The method used by the VMM is the standard page-based scheme used by modern day commercial operating systems such as Solaris, HPUX, or other System V Revision 4 (SVR4)-based UNIX implementations.

The NT VMM divides the available RAM into fixed-size *page frames.* The size of the page frame (page size) supported can vary from 4K to 64K; on Intel x86 architectures, it is currently set to 4K bytes.* Each page frame is represented by an entry in a structure called the *page frame database (PFN database).*† The page frame database is simply an array of entries allocated in nonpaged system memory, one for each page frame of physical memory. For each page frame, the following information is maintained:

---

* Windows NT and most other commercial operation systems currently use fixed-sized pages. However, a considerable amount of research has been performed *on* the implementation of support for variable-sized pages by the underlying hardware architecture and the operating system. Support for variable-sized pages might someday be implemented in commercial operating systems, though one might conjecture that UNIX platforms are likely to implement it sooner than NT. With Windows NT Version 4.0, Microsoft does use 4 MB-sized pages (supported by the Intel Pentium processor *extensions)* to contain kernel mode code on Intel platforms. However, as stated here, truly variable-sized pages are not yet supported by the Windows NT platform.

† This is similar to the *core map* structure on 4.3 BSD-based systems.

- A physical address for the page frame represented by the entry in the PFN database. This physical address is currently limited to a 20-bit field. When combined with a 12-bit page offset, you can see that the resulting 32-bit quantity is limited to supporting a 4GB physical memory system.
- A set of attributes associated with the page frame. These are:
  - A modified bit that indicates whether the contents of the page frame were modified
  - Status indicating whether a read or write operation is underway for the page frame
  - A *page color* associated with the page (on some platforms)

NOTE On systems that have a physically indexed direct-mapped cache, poor allocation of virtual addresses to physical addresses within page frames can lead to contention for the same cache line (i.e., 2 physical pages hash to the same cache line) and hence always cause cache misses if the pages happen to be part of the working set for one process or for two or more processes executing concurrently. Page coloring attempts to address this problem in software. Note that page coloring support is not provided by the NT VMM on x86 based machines. However, such support is provided, for example, for the MIPS R4000 processor.

  - Information on whether this page frame contains a shared page or a private page for a process
- A back pointer to the Page Table Entry/Prototype Page Table Entry (PTE/PPTE)* that points to this page. This pointer is used to perform a reverse mapping from a physical address to the corresponding virtual address.
- Reference count for the page. The reference count value indicates to the VMM whether any PTE refers to the page in the page frame database.
- Forward and backward pointers for any hash lists on which the page frame might be linked.
- An event pointer that refers to an event whenever a paging I/O read operation is in progress; i.e., data is being brought into memory from secondary storage.

Valid page frames are those that have a nonzero reference count. These page frames contain a page of information actively being used by some process (or by the operating system). When a page frame is no longer pointed to by a PTE, the

---

* Page Table Entries and Prototype Page Table Entries are described in more detail later in this chapter.

reference count is decremented. When the reference count is zero, the page frame is considered unused. Each unused page frame is on one of five different lists, reflecting the state of the page frame:

- The bad page list, linking together page frames that have parity (ECC) errors
- The free list, indicating pages that are available for immediate reuse but have not yet been zeroed

  The NT VMM (in order to conform to C2 level security as defined by the US DOD) will not reuse a page frame unless the contents have been zeroed. However, in the interest of keeping low overhead, pages are not zeroed each time they are freed. Once a critical mass of free and not-zeroed pages has been reached, a system worker thread is awakened to asynchronously zero pages on the free list.
- The zeroed list, linking page frames that are available for immediate reuse
- The modified list, linking page frames that are no longer referenced but cannot be reclaimed until the contents of the page have been written to secondary storage

  Writing modified pages to secondary storage is typically performed asynchronously by the *Modified Page Writer/Mapped Page Writer,* a component that I will discuss in detail later in this chapter.
- The standby list, containing page frames with pages that were removed from the process's working set

  The NT VMM aggressively tries to decrease the number of page frames allocated to a given process, based upon the access pattern of the process. This number of pages allocated to the process at any given instant is called the *working set* for the process. By automatically trimming the working set for a process, the NT VMM tries to make better use of the physical memory. However, if a page frame allocated to a process is stolen due to this trimming of the working set, the VMM does not immediately reclaim the page frame. Instead, by placing the page frame on this standby list, the VMM delays the reuse of the page frame, giving the process an opportunity to regain the page frame by accessing an address contained within it. While a page frame is on this list, it is marked as being in a transitional state, since it is not yet free, nor does it really belong to a process.

The NT VMM keeps both a minimum and a maximum for the total of free and standby page frames on the system. Whenever a page frame is linked to the free or standby list and the total is below the minimum or above the maximum, an appropriate VMM global event is signaled. These events are used by the VMM to determine whether sufficient number of pages are available in the system.

Often, the VMM invokes an internal routine to check whether memory is available for a certain operation. For example, your driver might invoke a system routine called MmAllocateNonCachedMemory ( ) . This routine needs free pages that it can allocate to your driver and therefore invokes an internal routine (not directly available to kernel developers) called MiEnsureAvailablePageOrWait ( ) to check whether the number of required pages are available from either the free or the standby list. If not available, the MiEnsureAvailablePageOrWait ( ) routine will block on the two events waiting for sufficient pages to become available. If neither of the two events is set within a fixed period of time, the system will panic by invoking KeBugCheck ( ) .

Note that manipulation of the page frame database is a frequent operation. There has been considerable research on how VMM implementations can achieve greater concurrency by using fine-grained locking for the page frame database (or whatever the equivalent structure is called on some specific platform). However, the NT VMM does not follow any such model of using fine-grained locking. There is a global lock, an Executive spin lock, for the entire page frame database. This spin lock is acquired at an appropriate IRQL (APC_LEVEL or DISPATCH_ LEVEL) when the PFN database is accessed. This might reduce concurrency, since it forces single threading whenever the PFN database must be accessed, but it definitely simplifies the code. Note that no I/O is ever performed (indeed no routine outside the VMM module is ever invoked) with the PFN lock acquired. However, since the lock is acquired at DISPATCH_LEVEL or less, you can now be completely convinced that any page fault by your code at a higher IRQL will lead to a system panic.

Virtual Address Support

The NT Virtual Memory Manager provides virtual address support to the remainder of the system:

- Virtual address ranges can be manipulated independently of the physical memory on the system.
- If a virtual address is backed by either physical memory or on-disk storage, the NT VMM assists the processor hardware in transparently translating the virtual address into the corresponding physical address.
- If the page containing the translated physical address needs to be read from secondary storage, the NT VMM initiates and manages the I/O operation.

  To achieve this transfer of data from disk to memory, the NT VMM uses the support of the appropriate file system driver.

- The VMM determines the paging policies used to control the transfer of information to and from disk and main memory to maximize system throughput.

As noted earlier in this chapter, the VMM provides each process with an address space larger than the amount of physical memory available on the system. Virtual addresses must eventually refer to some code or data residing in physical RAM on the system. Therefore, in order to support this large address space, the VMM and the system hardware must transparently translate virtual addresses into physical addresses. Furthermore, since the total memory requirements of all processes executing on the system will typically be in excess of the total physical memory available, the VMM must be able to move data and code to and from secondary storage as required.

The NT VMM is a core component that determines the perceived performance and cost of the system. RAM, although getting cheaper every day, is still not a costless component. At the same time, users are very demanding of their machines and a poor implementation of the VMM can significantly degrade the overall system throughput. Therefore, the VMM is extremely sensitive to the minimum memory requirements it imposes upon the system. As is the case with every design decision, certain tradeoffs have to be made. Later in this chapter, I will discuss an explicit tradeoff made by the designers of the NT VMM, resulting in problems for implementations of distributed file systems in the NT environment.

Virtual Address Manipulation

To provide a separate virtual address space for a process, the NT VMM maintains a self-balancing binary tree (splay tree) of Virtual Address Descriptors (VADs) for each process in the system. Every block of memory allocated for a process is represented by a VAD structure inserted into this tree. A pointer to the root of this tree is inserted into the process structure. A virtual address descriptor structure contains the following information:

- The starting virtual address for the range represented by the VAD
- The ending virtual address for the VAD range
- Pointers to other VAD structures in the splay tree
- Attributes determining the nature of the allocated virtual address range

  These attributes contain the following information:

  — Information on whether allocated memory has been committed

  For committed memory, the VMM allocates storage space from a page file to back up the allocated memory whenever it needs to be swapped to disk.

— Information specifying whether the range of allocated virtual addresses are private to the process, or whether the virtual address range is shared

— Bits describing the protection associated with the memory backing a range of virtual addresses

The protection is composed of combinations of primitive protection attributes: PAGE_NOACCESS, PAGE_READONLY, PAGE_READWRITE, PAGE_WRITECOPY, PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY, PAGE_GUARD, and PAGE_NOCACHE.

— Whether copy-on-write has been enabled for the range of pages

The copy-on-write feature allows efficient support for POSIX-style fork() operations, in which the address space is initially shared by parent and child processes. If, however, either the parent or children try to modify a page, a private copy is created for the process performing the modification.

— Whether this range should be shared by a child process when a fork() occurs (VIEW_UNMAP = do not share, VIEW_SHARE = shared by parent and child)

This information is valid only for mapped views of a file, which are discussed later in this chapter.

— Whether the VAD represents a mapped view of a section object

— The amount of committed memory associated with the VAD

Whenever memory is allocated on behalf of a process or whenever a process maps a view of a file into its virtual address space, the NT VMM allocates a VAD structure and inserts it into the splay tree. At allocation time, a process can specify whether it requires committed memory, or whether it simply needs to reserve a range of virtual addresses. Allocating committed memory results in the amount of memory requested being charged against the quota allocated to the process. Reserving a virtual address range, however, is a benign operation in that only a VAD structure is created and inserted into the splay tree, and the starting virtual address is returned to the requesting process. Note that memory must be committed before it is actually used.

The NT VMM allows a process to allocate and deallocate purely virtual address spaces, i.e., the memory need never be committed. If a process allocates a virtual range of addresses and subsequently discovers that it needs to commit only a subset of the range, the NT VMM also allows the process to do so.

There is a native allocation routine supplied by the NT VMM called NtAllocate-VirtualMemory ( ) , which is not available to kernel developers. Kernel-mode drivers have access to the following routine instead:

```
NTSTATUS
ZwAllocateVirtualMemory(
    IN HANDLE               ProcessHandle,
    IN OUT PVOID           *BaseAddress,
    IN ULONG                ZeroBits,
    IN OUT PULONG           RegionSize,
    IN ULONG                AllocationType,
    IN ULONG                Protect
);
```

Parameters:

ProcessHandle

An open handle to the process in whose context the memory is being allocated. For NT kernel-mode drivers that call this routine, it is the context of the system process (e.g., at driver initialization time). You can use the macro NtCurrentProcess ( ) , which simply returns a special handle value of (-1) which identifies the current process as the system process. Note that if you ask for memory to be allocated within the context of a process other than your current process, the NtAllocateVirtualMemory ( ) routine will use the KeAttachProcess ( ) routine described earlier, to attach your process to the target process before allocating the range of virtual addresses.

BaseAddress

Upon a successful return from this routine, the BaseAddress argument will contain the starting virtual address for the allocated memory.

If you supply a nonnull initial value, the VMM will attempt to allocate the memory at the address supplied by you, after rounding it down to a multiple of the page size. If, however, you supply a null initial value, the VMM will simply pick a base address for you.

Note that if the VMM cannot allocate memory at the base address supplied by you (the address has already been used or not enough contiguous memory is available beginning at that address), and if you have specified MEM_RESERVE as AllocationType (defined later), an error will be returned (STATUS_ CONFLICTING_ADDRESSES). The same error will be returned if you supplied a base address without previously reserving it (using this same routine).

Finally, you cannot specify a base address greater than 2GB, and your specified range cannot exceed the 2GB virtual address limit. The important point to note, then, is that if you use this call, you will get a kernel-mode address that will not be valid in the context of any process except the process passed

in (via the handle argument). This call is, therefore, not the preferred way to get kernel memory for your driver (use the ExAllocatePool() routines instead).

ZeroBits

This argument is only valid if the BaseAddress argument discussed above was passed in initialized to NULL (the VMM gets to pick the base address). You can specify the number of high-order bits that must be zero for the base address of the allocated memory.

By doing this, you can ensure that the returned starting address is below a specific value. This argument cannot be greater than 21 (since that would make the starting address less than 4096 bytes). A value of 0 is treated (at least) as a value of 2, since the returned virtual address will always be within the user-space-addressable 2GB of virtual address space.

RegionSize

Note that this is a pointer argument. You must supply the number of bytes to be allocated. You will receive the actual number of bytes allocated, which will be your number rounded up to a multiple of the page size.

AllocationType

You have a choice of MEM_COMMIT, MEM_RESERVE, or MEM_TOP_DOWN. The first option indicates that you wish space to be reserved in the page file (this memory is committed and therefore usable). The second option says that you simply want the virtual address range and that you might commit the memory later. The first two options are therefore mutually exclusive. The third option can be combined with either of the first two and it states that you want the highest possible starting virtual address allocated, given the constraints specified by the ZeroBits argument.

Protect

Your options are one or more of the following primitive protections: PAGE_NOACCESS, PAGE_READONLY, PAGE_READWRITE, PAGE_NOCACHE (cannot be placed into the data cache, this is not allowed for mapped pages), and PAGE_EXECUTE.

Functionality Provided:

This routine can only be used to allocate memory within the lower 2GB of the process virtual address space (even for the system process). Therefore, it is typically not used by kernel-mode drivers, unless you are quite sure that you will to access the memory only in the context of the specified process. If you need to allocate memory that is accessible within the context of any process, use the ExAllocatePool() routines instead.

This routine allows you to do one of three things:

- Reserve a range of virtual addresses but not commit them
- Reserve and commit a range of virtual addresses (in one call)
- Commit a previously reserved range of virtual addresses

The corresponding routine to free the allocated range is defined as follows:

```
NTSTATUS
NTAPI
ZwFreeVirtualMemory(
    IN HANDLE                ProcessHandle,
    IN OUT PVOID            *BaseAddress,
    IN OUT PULONG            RegionSize,
    IN ULONG                 FreeType
);
```

Parameters:

ProcessHandle
: An open handle to the process in whose context previously allocated memory is being freed.

BaseAddress
: The first address of the virtual address range being freed. This value is rounded down to a multiple of the page size.

RegionSize
: Note that this is a pointer argument. You must supply the number of bytes to be freed. You will receive the actual number of bytes freed, rounded up to a multiple of the page size.

FreeType
: Your options are one of the following: MEM_DECOMMIT or MEM_RELEASE. These are mutually exclusive.

Functionality Provided:

You can use this routine to do the following:

- Decommit previously committed pages (but retain the virtual address range allocation)
- Release both the committed memory as well as the virtual address range that was previously allocated

This routine is fairly flexible, in the sense that it allows you to modify a subset of the address range previously allocated by you. Note, however, that you cannot expect to be able to free or release a range that spans two previous invocations to ZwAllocateVirtualMemory( ); i.e., the entire range that you specify must be contained within a single, previously allocated VAD. If you specify a Region-Size value equal to 0, the VMM interprets this to mean that the entire VAD must

be freed/decommited. However, in this case, you must specify the correct BaseAddress (equal to the starting BaseAddress of the VAD, or the BaseAddress specified when you allocated the range earlier).

It might sound strange but there is a possibility that you might get an error indicating that you exceeded your quota for the target process if you try to free a subset of a previously allocated range. The reason for this is that the VMM splits a VAD, if required, into two VADs in order to accommodate your request to free up a range contained within the original allocated range. Of course, this requires allocating a new VAD structure which is charged to the quota assigned to the target process. If this pushes the allocated memory for that process to an amount greater than what is allowed, you will get an error returned.

Translation of Virtual Addresses

In this section, I will briefly discuss virtual to physical address translation. This topic is covered very well in the literature, and I recommend that you consult Appendix E, *Recommended Readings and References,* for further information.

Each virtual address in Windows NT is currently a 32-bit quantity. This virtual address must be transparently translated to refer to some physical byte in memory.* Two system components work together to achieve this translation:

- The Memory Management Unit (MMU) provided in hardware by the processor
- The Virtual Memory Manager implemented by the operating system in software

Translation is not necessarily performed only in one direction, for example, from virtual memory addresses to physical memory addresses. The VMM must also be able to translate in the reverse direction, from a physical address to any corresponding virtual addresses.t Whenever the contents of a physical page are written out to secondary storage to make room for some other data, the corresponding virtual addresses must be marked as "no longer valid in memory." This requires that the physical address be translated back to its corresponding virtual address.

Virtual address translation is typically performed by the MMU in hardware. The VMM is responsible for maintaining appropriate *translation maps* or *page tables* that can subsequently be used by the MMU to do the actual translation. Broadly

---

* Memory-mapped I/O device registers are also addressable via the virtual address space. Therefore, a virtual address could be translated to a physical address that actually corresponds to a mapped register on an I/O bus.

t It is possible for an operating system to implement aliasing, where more than one virtual address refers to the same physical address.

speaking, the following sequence of operations is typically performed to translate a physical address:

1. As part of the context switch procedure that causes a process to begin executing, the VMM sets up appropriate page tables that contain virtual-to-physical address translation information specific to that process.
2. When the executing process accesses a virtual address, the MMU attempts to perform virtual to physical address translation by either using a cache called the Translation Lookaside Buffer (TLB) or, if an entry is not found in the TLB, using the page tables set up by the VMM. Each translated address must be contained within a page that, in turn, might be present in one of the page frames on the system.

NOTE Translating from a virtual to a physical address is a time-consuming operation. Since this operation must be performed for every memory access, most architectures provide efficient translation. One way of speeding up this process is by using an associative cache such as a Translation Lookaside Buffer (TLB). The TLB contains a list of the most recently performed translations, tagged by the process ID. Therefore, if a virtual address is located in the TLB, the corresponding physical address can be immediately obtained and the contents of that address are guaranteed to be in main memory. Software manipulation of the TLB is architecture dependent; some architectures allow the VMM to explicitly load, unload, and flush TLB entries (either one entry at a time or the entire TLB), while other architectures simply load or unload the TLB as a by-product of certain execution sequences.

---

3. If the byte referenced by the translated physical address is currently in main memory, the process is allowed access to the data.
4. If, however, the contents of the page are not contained within a page frame in memory, an exception is raised, a page fault occurs, and control is transferred to the VMM page fault handler that brings the appropriate data into the system memory. An exception could also be raised by hardware if the page protection conflicts with the attempted mode of access or for other similar reasons.

Note that the design of the MMU has far-reaching implications on the design of the VMM subsystem. Naturally, the portion of the VMM subsystem that interfaces with the MMU is very architecture-specific and inherently nonportable.

As described earlier, the VMM maintains a page frame database in nonpaged pool to manage the physical memory available on the system. This database is composed of page frame entries where each page frame represents a chunk of

contiguous physical memory. Since each physical page frame in the system is numbered sequentially (from page frame *0* to page frame *(n-1)* for *n* page frames of physical RAM), computing the PFN database entry for a page frame is relatively trivial. Once a virtual address has been translated into a physical address (composed of a page frame and an offset into the frame), the page frame number is multiplied by the size of the PFN database entry and the resulting address is added to the physical base address assigned to the PFN database. The net result is a physical address pointer to the start of the entry describing the page frame in the PFN database for the translated physical address.

Consider the 32-bit virtual address on a Windows NT platform. Since the page size is 4096 bytes, computing an offset into a page requires 12 bits (the least significant 12 bits). This leaves the MMU with 20 bits to uniquely identify a page frame. Page frames are uniquely identifiable via Page Table Entries (PTEs) in a page table, where a page table is simply an array of PTEs. Note that many architectures (including the Intel x86 architecture) clearly define the structure of a PTE.*

On the Intel platform, each PTE must be 32 bits (or 4 bytes) wide. Given that there are a total of 2^{20} (1 million) possible PTEs and each PTE has a size of 2^2 bytes, the amount of memory required to store translation information for a single 4GB virtual address space is 2^{22} bytes (4MB). Since each page table can itself store one page-size worth of information (2^{12} bytes), 1024 page frames would be required simply to contain all the PTEs for the virtual address space for a single process.†

To avoid consuming this significant amount of memory for translation information,‡ page tables are also paged in and out of memory. To do this, the x86 processor defines a two-level page table scheme. Each process has a page directory that contains PTEs for page tables. This directory is a single page in size and therefore can contain 1024 PTEs, each referencing a page table. A typical virtual

---

* Other architectures, such as the MIPS R3000, provide no hardware support for page tables. Therefore, the MIPS R3000 does not mandate the structure of PTEs either, since the entire responsibility of translating virtual to physical addresses lies with the VMM.

† Note that the Intel x86 architecture is segmented, where a virtual address is actually composed of a segment and an offset. The Intel hardware converts this virtual address into a 32-bit linear address, which is subsequently translated into a physical address using the method described in this section. Since the Windows NT VMM presents a flat memory model to the system (hiding the segmentation details), we will neglect the virtual to linear address conversion process and assume that user addresses are virtual addresses that simply require a 1-step conversion to a physical address.

‡ Note that rarely will 4GB of virtual address space need to be completely translated, since most address spaces are sparse in nature, i.e., there exist gaps in the virtual address space for addresses that are never used. Reserving memory for PTEs that will probably never be utilized is therefore quite unnecessary.

address for a process has 10 bits reserved to identify a page table from a page directory associated with the process, 10 bits to identify a page frame given a page table, and 12 bits to get to the desired offset within a page.

Figure 5-2 graphically illustrates how virtual to physical address translation is performed on Windows NT systems. Note that even on systems such as the MIPS R3000 where the architecture places no limitations on the structure of the PTEs (and correspondingly provides no hardware virtual address translation support except for TLB lookups), the VMM maintains a similar set of data structures to simplify the design and maintenance of the VMM subsystem.

Everything so far seems to be relatively straightforward. The MMU checks the TLB and if it gets a TLB *hit,* it simply returns the translated physical address. On the other hand, if it gets a TLB *miss,* it must check the page tables for the process to locate the corresponding PTE that determines the physical page frame that might contain the accessed address. Now, if the PTE indicates that the page is resident in memory and the protection attributes match the access mode, the MMU allows the access to continue. Otherwise, an appropriate exception (page-fault or protection-violation) is raised and control transfers to the VMM. However, the observant reader must have noticed the presence of an additional table called the *Prototype Page Table* in Figure 5-2. So where exactly does the PPT fit into this clean model we understand so well by now?

Prototype Page Tables are used to contain page table entries for page frames that contain pages shared by more than one process. Sharing of pages and page frames occurs when more than one process maps in the same byte range for the same mapped object. Therefore, to understand the PPT, you must first understand the concept of shared memory and memory mapped files.

Shared Memory and Memory-Mapped File Support

Accessing memory seems so convenient to application developers these days. An application process can simply issue a `malloc` call (or its equivalent), receive a virtual address from the VMM, and begin using this virtual address to access the allocated block of memory. The operating system is responsible, along with the hardware, for managing physical memory and maintaining the appropriate translation between virtual addresses and physical addresses. Furthermore, the operating system can observe the behavior of all processes executing on the system, allowing it to make rational decisions concerning the allocation of physical memory to specific processes.

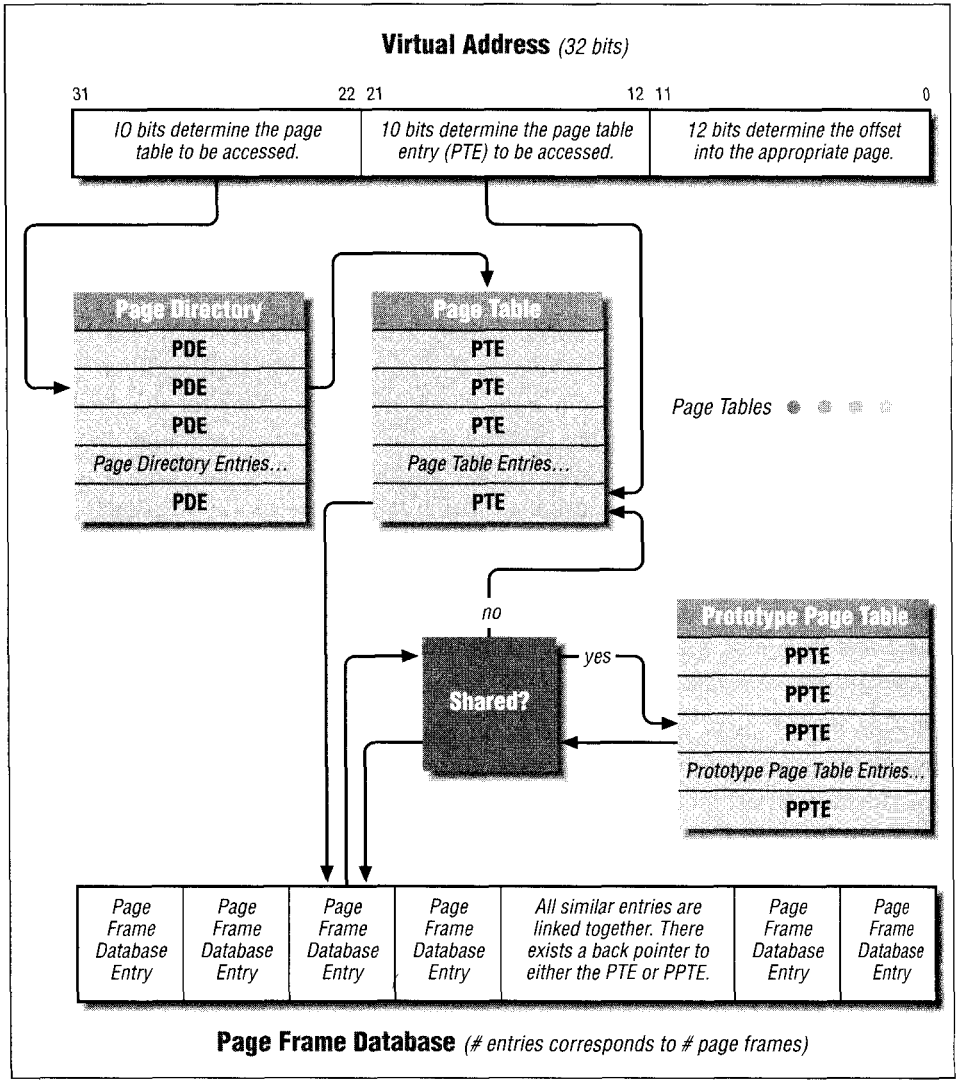

Figure 5-2. Virtual to physical address translation

At the same time, most applications must do other things besides computational activities requiring memory. Notably, all applications need to perform some I/O to and from secondary storage. In addition, sophisticated applications sometimes wish to share in-memory data with each other.

Traditionally, I/O has been performed via read/write system calls handled by the appropriate file system. Servicing these calls requires the execution of a system trap to switch the processor from user mode to kernel mode and vice versa. For a

read request, the file system must first read data into system memory and then copy it into buffers allocated by the application. For a write request, the operating system must first copy data from the application's buffers into system memory. This copying of data to and from system buffers, combined with the overhead of making system calls for I/O requests, can lead to substantial execution overhead for application processes.

Consider now the case where two processes on the same system are accessing the same file. These processes might be accessing the same byte range, but since they have their own private buffers containing the data, where each buffer is backed by physical pages different from those backing the other buffer, each process has potentially a different view of the same data. *Process-1* might have read the data into memory and modified it but not yet written it out to disk; if *process-2* reads the same byte range, it will not see the modified data but will instead be given the original data obtained from disk. This can be a deterrent to efficient sharing of data between the two processes, because each process would have to ensure that its modifications are written out to disk before the other process reads-in the byte range.

Imagine now if each process could simply map the on-disk file into their virtual address space. The VMM provides virtual memory support by swapping data to and from an on-disk page file whenever required. An application allocates some memory, tries to access it and possibly gets a page fault. The page fault is resolved (we will see how later in this chapter), and magically the application can now access some physical memory reserved solely for its use.

Now consider the case where the data is originally read from an on-disk file and is destined to be written out to the same on-disk file. In this case, why not use the file itself as the backing store for data instead of a page file? Instead of making the application issue read/write system calls to access the data, simply let the application reserve a virtual address range associated with an on-disk byte range, try to access this memory (in reality, access the byte range with which the virtual addresses are associated) represented by the virtual address range, get a page fault, and then the operating system will resolve the page fault by allocating some physical memory and obtaining the appropriate data from the on-disk file. Similarly, the application can simply modify the data in-memory and the operating system will—whenever required—write out the modified data to the on-disk file and, possibly, release the physical memory to make room for another process.

The above method of mapping in a file has one additional benefit; all applications that try to map in the same file can now have their respective virtual addresses backed by the same physical pages, so all applications will always see a consis-

tent view of the data, regardless of the fact that any application could modify the data at any time.*

The NT VMM supports file mapping. The mapped object is the on-disk file. When you execute a file (say Microsoft Word), the executable (the mapped object in this case) is mapped into your process's virtual address space and instructions are executed. Now, if some other user, on the same machine, tries to execute Microsoft Word as well, the same executable is mapped into his or her virtual address space, and since the physical pages backing the VADs are probably already in memory, the other user should see a relatively fast response time. See Figure 5-3 for an illustration of file mapping.

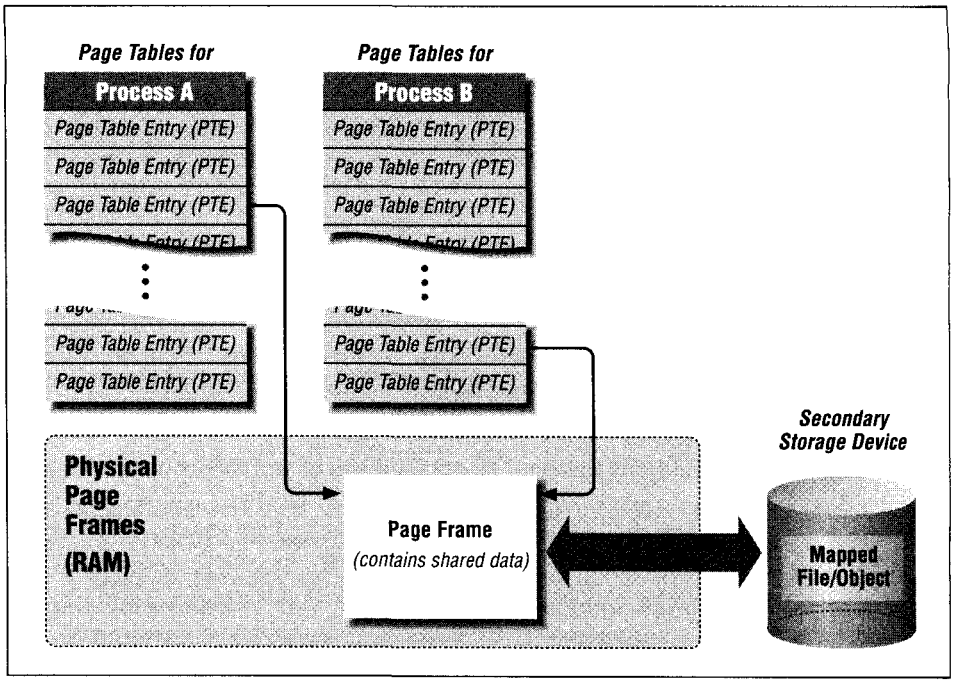

Figure 5-3. Two processes mapping the same page into their virtual address space

Note that file mapping is not the only way to share physical memory between two processes. Since Virtual Address Descriptors (VADs) are manipulated separately from the physical page frames backing the virtual addresses, it is entirely possible for the VMM to allow processes to share memory by simply ensuring that appropriate VADs for the two processes are backed by the same physical page frames. File mapping is simply an extension of this concept wherein the shared

* Each application must synchronize its changes, so that there are no unexpected consequences.

memory object is actually backed by an on-disk, permanent file object, instead of a page file. Just as you can create file-backed shared memory objects, it is also possible to create shared memory objects that will later be backed by one of the system page files. This is typically done when you wish to share memory between two modules or processes in the system. Often, kernel-driver designers need to share memory between some user-space helper processes and the kernel driver. The shared memory support provided by the VMM allows this functionality. When a shared memory object is created (one that is not backed by an on-disk file), the starting virtual address associated with the object represents *offset 0* into the shared object. Therefore, all processes sharing this object can index into the appropriate byte offset and manipulate data. You must note, however, that modifications to shared memory objects that are not backed by an on-disk file will not be permanent; i.e., such modifications will be lost once the shared object is closed by all processes using this object.

So how does mapping actually work? What data structures are created by the VMM to support mapped/shared objects. Before I address these questions, let me revisit the issue of the Prototype Page Table described back in Figure 5-2.

Prototype Page Table

Page frames that contain shared (mapped in) pages are described by a special structure—the Prototype Page Table (PPT). This structure can be allocated from paged or nonpaged memory.

Whenever the VMM creates a mapping or a shared object for a process, it allocates Prototype Page Table Entries (PPTEs) to describe the physical page frames that will back the file mapping. The PPT for a mapped object is shared by all processes that map in the same object. Each PPTE refers to a page that may or may not be present in memory; i.e., the page may be contained within a physical page frame, or it may need to be brought in from secondary storage when accessed. Since all processes have to use the same PPT (and corresponding PPTEs), it follows that all processes use the same physical page frame and therefore see the same view of the mapped data.

Whenever a page frame is assigned to a PPTE, the PPTE is marked as valid. The page frame entry within the PEN database is then initialized to point back to the PPTE. Note that neither the Intel x86 MMU nor the MIPS R3000 or similar architectures support prototype page tables. How does the VMM arrange things such that the MMU can work with shared memory?

Consider the Intel x86 architecture. The Intel x86 MMU strictly defines the structure of page tables and PTEs. The VMM creates a PPT (with PPTEs) in allocated memory whenever a process creates a file mapping. Imagine now that the

process tries to access a virtual address that is part of a range backed by a mapped file object. The MMU will translate the virtual address into a page directory table offset and then offset into an appropriate page table. On the first access to this virtual address, the page table entry will indicate that the page is not backed by any physical memory.

This will result in a page fault and control will transfer to the VMM page fault handler. The page fault handler notices that the VAD containing the accessed virtual address is marked as being backed by a mapped object. The VMM can then find the appropriate PPTE and fault the page in. At this point, the PPTE is marked as valid and refers to a PFN database entry and correspondingly, a PFN database entry points back to the PPTE. At the same time, the VMM initializes the PTE as valid and makes it refer to the appropriate physical address. The net result is that both the PPTE and the PTE contain information about the physical address, but the corresponding PFN database entry only points back to the PPTE. Now, the memory access is retried, and, since the MMU finds the PTE initialized correctly (it does not know nor does it care about PPTEs), the translation from virtual address to a physical address can be performed.

A Small Problem with the PPT Design

You must note that, since the PFN database entry never refers back to the PTE, the VMM has no way of finding, from a PFN database entry, all the PTEs for all the processes that have mapped that shared object into their virtual address space. The best that the VMM can do is find the PPTE that refers to the PFN database entry (using the back pointer) and thereby manipulate the contents of the PPTE.

There is one serious flaw with this design: imagine that a kernel-mode component wanted to request the VMM to purge certain pages from physical memory,' Normally (for nonshared files), you can certainly ask the VMM to do this and the VMM will respond by marking the PFN database entry invalid. Furthermore, the VMM will use the information stored in the PFN database entry to find the appropriate PTE in the address space of some process that is currently referring to the PFN database entry. It will mark the PTE entry not valid, ensuring that the MMU will have to fault the page back in on the next access to an address contained within the page.

---

* You might ask why would anyone want to *do* this? Suppose you were implementing some complicated distributed data access method across multiple nodes where all consistency guarantees were maintained by your modules. Now, if some process on a remote machine modified shared data that was mapped in on a local node, you might wish to ensure that all nodes accessing this data refreshed their memory with the latest copy of the data. This is precisely what distributed file systems such as the OSF DPS attempt to do. There could be other similar scenarios that might be needed to support certain complicated functionality on distributed architectures.

However, if the page belongs to a mapped object, the VMM has no way of accessing all the PTEs that refer to the page frame containing the shared page. Therefore, if you requested that the VMM purge such a page from memory, the VMM will return an error saying that this functionality is not possible for mapped objects. This is a serious problem for any third-party developer that counts on being able to purge pages from system memory on demand.

Sections and Views

The Windows NT system tends to be strongly object-centric; i.e., most functionality is provided in the form of objects and methods that manipulate such objects. File mappings are created and accessed as a two-step process:

1. A *section object* is created by the VMM in response to a request for a file mapping or a shared memory object.
2. When the process actually needs to access a byte range for a mapped file or a shared memory object, the caller must request the VMM to map a *view* into the file. Conceptually, this view is like a window into the file, allowing access to a limited byte range. Of course, it is possible for a process to request multiple views for the same file concurrently, just as it is possible for multiple processes to have different views concurrently into the same mapped file.

Note that section objects have a set of protection attributes associated with them, just as all other objects in the Windows NT environment can. By specifying a set of protection attributes for the section object, a process can define the manner in which this object (and any data for a file object that might be mapped in and represented by the section object) is manipulated.

Section objects backed by on-disk files fall into two categories:

- Executable image file mappings
- File (nonimage) mappings

When you tell the VMM to create a section object representing a mapped file, you can specify how the mapped file should be treated. The system loader uses file mapping to run executables and specifies that the file mapping be treated as an executable image file mapping. However, it is entirely your prerogative to request that an executable (say, a copy of Microsoft Word) be mapped in as a nonimage file mapping.

Note that the VMM performs tests to verify that any section object created for an executable image file mapping actually does map in a valid executable. If you try to map in a simple text file as an executable image file mapping, you will get an error from the VMM. Also, it is entirely possible for the same executable file to be mapped both as an executable image file and as a simple file mapping; the

address alignments for each of these mappings will probably be quite different though.

A major difference between how executable image file mappings and nonimage file mappings (or simple shared memory) are handled is in how modifications to the mapped range are managed by the VMM. When a nonimage file mapping is modified by a process, the modification is immediately seen by all processes mapping in the same file, because the contents of the shared physical page are changed by the VMM. These modifications will eventually be reflected in the on-disk mapped object when the modifications are flushed to secondary storage. However, when an image file mapping is modified, a private copy of the page is made for the process making the modification. This private page will now be backed by a page file, since the modifications to an image file mapping are never written out to the mapped object (the on-disk file). These modifications are eventually discarded when the process unmaps the file.

To create a shared memory object (a section object), the NT VMM provides a routine called NtCreateSection(). Though this routine is not exported to kernel developers, the ZwCreateSection() routine can be used instead. This routine is defined as follows:

```
NTSTATUS
NTAPI
ZwCreateSection (
    OUT PHANDLE                     SectionHandle,
    IN ACCESS_MASK                  DesiredAccess,
    IN POBJECT_ATTRIBUTES           ObjectAttributes  OPTIONAL,
    IN PLARGE_INTEGER               MaximumSize OPTIONAL,
    IN ULONG                        SectionPageProtection,
    IN ULONG                        AllocationAttributes,
    IN HANDLE                       FileHandle  OPTIONAL
);
```

Parameters:

`Sect!onHandle`

If this routine returns a success status, a handle to the created section object is returned in this argument. Note that this handle is only valid in the context of the process that creates the section object. If you wish to access the section object in the context of other processes as well, you must use the ObReferenceObjectByHandle() object manager routine (described in the DDK) to get a pointer to the actual section object. Subsequently, you can use the ObOpenObjectByPointer() routine to get a handle in the context of some other process.

`DesiredAccess`

This argument allows you to specify the access desired to the section object: SECTION_MAP_EXECUTE, SECTTON_MAP_READ, or SECTION_MAP_WRITE.

ObjectAttributes

This can be NULL, or you can specify an initialized structure (use the InitializeObjectAttributes() macro to do this). Note that if you need to share a piece of memory between two processes (or share memory between a module executed in user mode and a kernel-mode driver), you can use this structure to supply a name for the section object. This named section object can subsequently be opened by other processes, and thus sharing of in-memory data can be achieved without having to use a named file from secondary storage.

This structure can also be used to supply a security descriptor for the section object, which allows you to protect the section object appropriately.

MaximumSize

For a page-file-based section (i.e., you simply wish to create a shared memory object), this value cannot be NULL, since it represents the size of the section.

For a mapped file, it represents the maximum size to which the section might be extended. If the section is for a mapped file and the size of the file is less than this value, the file size is extended at this time.*

Note that any value supplied by you is rounded up to a multiple of the host page size. Finally, if this value is set to NULL for mapped files, the VMM will set the value to the end-of-file at that time (appropriately rounded up).

SectionPageProtection

This defines the protection to be placed on each page contained in the section. Here are the appropriate values:

- PAGE_READONLY
- PAGE_READWRITE
- PAGE_EXECUTE
- PAGE_WRITECOPY

AllocationAttributes

These attributes allow the caller to inform the VMM if this section object represents a shared piece of memory (backed by the page file), a file mapping for

* This is a very important point to note for developers of file systems, since you should he prepared to receive a request for extending the file size when the memory manager is in the process of creating a file mapping. I will discuss this more later in this chapter.

an executable, or a file mapping for some other type of file. Here are the options to use:*

— SEC_IMAGE, indicating that an executable is being mapped into a process virtual address space

— SEC_FILE, indicating that the supplied file handle refers to an open file that must be treated as a regular (nonimage) file mapping

— SEC_RESERVE, indicating that all pages allocated to the section object must be placed into the reserved state (only valid for a shared memory object not backed by an on-disk file)

— SEC_COMMIT, indicating that all pages allocated to the section object must be placed into the committed state (must also be set if SEC_FILE is set or if the shared memory object is not an executable image file mapping)

`FileHandle`

This optional argument indicates that the section object represents a mapped file (the handle must refer to an open file). Otherwise, the VMM will simply create a section object backed by a page file (simple piece of shared memory).

Functionality Provided:

This routine can be used by kernel-mode drivers to create a shared memory object (named or anonymous) or to create a file mapping for an on-disk file. Even if you are a file system driver developer implementing an on-disk or a network file system, you can use this call to create a shared memory object or mapped file object (do not try to create a mapped file object on your own file system using this call unless you really know what you are doing).

Sometimes, kernel-mode driver developers wish to share in-memory data with user-space modules. Or, if you design a kernel-mode driver that obtains data from across the network or transfers data across the network using the services of a user process, you may use this call to create either a simple shared memory object or a file-backed shared memory object in order to facilitate easy and efficient data transfer between the kernel driver and the user-space process (consider using a named object to allow for easy opening of the object by the user-mode service).

* These symbolic definitions do not exist in any of the supplied DDK inelude files, but you can use the symbolic names (or the actual values) in the *winnt.h* include file provided with the Win32 SDK. Since this routine is not documented by Microsoft, they must have figured that it was not necessary to define these symbols in any DDK header file.

TIP In the description of the ZwCreateSection() routine, I mentioned the existence of an Object Manager routine that can be used to obtain a handle to an object in the context of any arbitrary process, given a pointer to that object. This routine is called ObOpenObjectByPointer() and is defined as follows (note that this routine is not ordinarily documented in the DDK):

```
NTSTATUS
ObOpenObjectByPointer(
    IN PVOID                Object,
    IN ULONG                HandleAttributes,
    IN PACCESS_STATE        PassedAccessState OPTIONAL,
    IN ACCESS_MASK          DesiredAccess OPTIONAL,
    IN POBJECT_TYPE         ObjectType OPTIONAL,
    IN KPROCESSOR_MODE      AccessMode,
    OUT PHANDLE             Handle
);
```

Typically, you can pass in NULL for PassedAccessState and for the ObjectType. Be careful to request only the type of access in the DesiredAccess argument permitted by the original open operation (from which you obtained a pointer to the object). The HandleAttributes can be obtained from the previous invocation to ObReferenceObjectByHandle(). That routine returns HandleInformation, which in turn contains the returned HandleAttributes.

There is also a routine called ObReferenceObjectByPointer(), which simply increments the object reference count for the specified object. This function is defined in the Windows NT IPS kit as follows:

```
NTSTATUS
ObReferenceObjectByPointer(
    IN PVOID                Object,
    IN ACCESS_MASK          DesiredAccess OPTIONAL,
    IN POBJECT_TYPE         ObjectType OPTIONAL,
    IN KPROCESSOR_MODE      AccessMode,
);
```

There are other routines, well documented in the DDK, to open and close a previously created section object and to map and unmap a view using a section object. Consult the available documentation for the following system support routines:

- ZwOpenSection()
- ZwMapViewOfSection()
- ZwUnmapViewOfSection()

File-Mapping Structures

When a process creates a file mapping, the process must specify whether an executable file or another type of file is being mapped. Although both types of file mappings eventually result in the file contents being mapped into the virtual address space of a process, the NT VMM treats the map requests differently.

As mentioned earlier, any modifications made to pages belonging to image file mappings will not be reflected in the on-disk mapped executable. The page will be backed by a page file instead, and all changes made to the page will be discarded once the mapping is closed.

Internally, the NT VMM maintains two types of section objects (and associated data structures) for each mapped file object. For each type of mapping, the VMM maintains a SEGMENT data structure that describes the mapping. Therefore, there are two possible segment data structures associated with each mapped file: the image segment and the data segment. Each segment data structure, in turn, points to the prototype page table for a mapped object.

Although the segment data structure is opaque to kernel-mode developers, the point to note here is that both types of mappings can exist concurrently. An executable can be mapped both as an image file and as a nonimage file. For each type of mapping, the VMM will create and maintain a segment data structure associated with the representation of the file in memory. Because there are two separate data structures created, depending on the type of mapping performed, the same byte range in a file contained within a page could exist in two separate page frames concurrently in memory! This is possible because each type of mapping has its own segment data structure and its own prototype page tables with different PPTEs.

Modified and Mapped Page Writer

As discussed earlier, the NT Virtual Memory Manager has the task of presenting the illusion of a large amount of available virtual memory to each process, even though the amount of physical memory on the system is limited. To perform this task, the NT VMM must use secondary storage devices as a backing store for in-memory data and page data in and out. This paging is performed transparently to the processes executing on the system.

The NT VMM automatically flushes dirty or modified pages to secondary storage to reclaim page frames for use by other threads in the system. Modified data within a page frame will be written either to one of the 16 possible page files, or to a named file on disk if the page frame was allocated to a mapped section

object. Unless modified page frames are flushed to disk, the NT VMM cannot reuse the page frames, as doing so would cause data loss.

In order to ensure that sufficient RAM is available whenever required, the NT VMM always keeps a certain number of page frames available. These page frames must not contain any modified data and therefore, they can be reallocated whenever the VMM decides to do so. If the VMM did not maintain this pool of available page frames, it might need to make processes block, waiting for modified data to be flushed to secondary storage before it could reassign page frames to them. Making processes block is not conducive to good system performance.

Therefore, the NT VMM creates at least two special dedicated threads called *Modified* and *Mapped Page Writer* threads. Note that it is quite possible that the number of threads created could be greater than two. At least one modifier page writer thread is created to asynchronously write modified page frames to the page files. At least one other thread, called the mapped page writer thread, is assigned to asynchronously write out modified page frames to mapped files. Both of these threads essentially perform the same functionality and therefore throughout this book, the terms mapped page writer thread and modified page writer thread are used interchangeably.

The sole purpose of these dedicated threads is to flush modified page frames out to secondary storage, thereby keeping a certain number of page frames available for reassignment. Each of these threads is a real-time thread with a priority set to at least LOW_REALTIME_PRIORITY + 1.

The algorithm used by the modified page writer threads is shown below. Note that the following pseudocode is based on the operations performed by the mapped and modified page writer threads flushing page frames to memory-mapped or page files; differences in operations between these two threads is clearly indicated whenever required:

```
// The following routine summarizes the MPW code executed by a dedicated
// worker thread. Note, however, that although the specific method used
// in various versions of the operating system might be different, the
// fundamental methodology described here should be consistent.
MiModifiedPageWriterWorker() {
    for (;;) {
        // Wait for event to get set, indicating that insufficient "free"
        // (not modified) pages exist. This event is set when the system
        // is running low on available pages and the VMM wants some
        // modified pages written out so they can be reassigned.
        // This event is also set when the total number of modified pages
        // in the system becomes greater than a pre-determined
        // threshold value (the "threshold value" in turn depends on
        // whether the system is configured as a workstation or as a server
        // and on how much RAM is present on the system).
        KeWaitForSingleObject(ModifiedPageWriterEvent, ...);
```

```
        II Now, lock the PFN database.

        …

        for(;;){

            // The event was set indicating that some pages need to
            // be flushed. Pick a page frame to be flushed (the first on
            // the modified pages list from the PFN database?) and invoke
            // an appropriate routine.
            MiGatherMappedPages(PageFramelndex, ...);

            // The above routine is responsible for the actual flush.
            // To perform the flush I/O, the PFN database
            // lock will have been dropped and reacquired by the
            // MiGatherMappedPages() routine. Therefore, check whether
            // adequate clean pages are now available and if so, stop
            // flushing.
            if (enough free pages are available) {
                // Unlock PFN database.
                …
                break;
            }
        } // End of loop in which the MPW thread flushed modified pages to
          // disk.

    } // End of infinite loop awaiting event to be set so that the
      // MPW thread can begin flushing pages.

} // end of MiModifiedPageWriterWorker() routine

// The following routine is responsible for collecting a bunch of
// contiguous modified pages and writing them out to the page file.
// The similar routine responsible for writing out mapped file pages is
// called MiGatherMappedPages().
MiGatherMappedPages(...) / MiGatherPageFilePages(...) {
    // Find a paging file for page file backed pages only.
    if (paging file not available) {
        // Nothing can be done as some I/O is already in progress
        //    to all paging files.
        return;
    }

    //    Find a contiguous chunk of available paging file space using a
    //    bitmap per paging file.
    //    OR
    //    If this is a mapped file, ensure that the mapped file is not an
    //    image file.

    //    Initialize a MDL (Memory descriptor List) to be used in the
    //    I/O operation

    // Scan both backward and forward, starting from the sent-in page frame
```

```
//    index, to find a contiguous set of modified pages that can be
//    written out to the page file or to the mapped file.
for (each candidate PTE) {
    if (PTE is modified and backed by the page file or by the mapped
        file){
        // Increment reference count on this PTE
        PTE->reference_count++ ;

        // Mark this PTE as "not modified," anticipating that our write
        // will succeed.
        PTE->modified = FALSE;

        // Mark the fact that this PTE is being flushed. Any flush
        // requests for this PTE (say from a file system or from the
        // Cache Manager will be blocked until this I/O completes) .
        PTE->being_flushed = TRUE;

        // Put the page file page address into the PTE.
        // Add this page frame into the physical page list described by
        // the MDL.
}

// OK, so now we have a list of page frames that we wish to flush.

if (number of pages reserved in the page file > number of contiguous
        modified page frames encountered) {
    // Release extra space pre-allocated from the page file
    //(if any).
}

// Unlock the PFN database lock.
…
/*****************************************************************/

// NOTE: If this were the routine handling mapped files, some
// additional processing would be performed here. This processing is
//as follows:
{
    // Only for mapped files.
    if (this file is marked as "fail all i/o, " forget it and return) {
        return;
    }

    // Make a callback into the file system advising the file system
    // that a paging I/O is on its way.
    // THIS IS VERY IMPORTANT:
    // The file system must - in response to this callback - acquire
    // all resources that might be needed to satisfy the paging-IO
    // operation.  We will cover this call-back in detail later in
    // this chapter and in Part 3 of this book.
    if (FsRtlAcquireFileForModWrite(...)) {
        // Call-back succeeded, issue I/O here
        …
```

```
            IoAsynchronousPageWrite(...)
        } else {
            // Call-back failed.
            // Return error locally = STATUS_FILE_LOCK_CONFLICT;
            // Note that pages will stay marked dirty and the operation
            // will be retried sometime later.
        }

        // Return

    } // End of code that is executed only for mapped files.

    /*********************************************************************/

    // NOTE:  The following code is only executed for pagefile backed pages

    {
        // Perform an asynchronous, paging-IO operation. This operation is
        // a special request handled by the I/O manager who quickly
        // redirects it to the appropriate file system on which the page
        // file is located . . .
        loAsynchronousPageWrite (...)  ;

        // Return;
    } // end of code executed only for page files.

    /*********************************************************************/

    // Lock the page frame database lock.
    …
} // end of MiGatherPageFilePages ( ) / MiGatherMappedPages ( )

// The following routine is invoked as an Asynchronous Procedure Call
// (APC) when the asynchronous paging I/O is completed by the file system.
// Note that the file system might choose to handle the I/O
// synchronously though that is not recommended …

MiWriteComplete(Context, StatusOfOperation, Reserved) {
    BOOLEAN        FailAllIoWasSet = FALSE;

    // The Context is actually the MDL that was sent to the file system
    MdlPointer = Context;
    …
    // Lock the PFN database
    …

    for (each page that comprised the MDL that was written out) {

        // Set write-in-progress to false
        PTE->being_flushed = FALSE;

        // If an error was encountered …
        if (error AND this was a write to a mapped file AND the mapped
                file belongs to a networked file system) {
```

```
            // THIS IS IMPORTANT TO FILE SYSTEM DEVELOPERS WRITING
            // REDIRECTORS.
            // The VMM assumes that if a paging I/O to a file across the
            // network has failed, the network MUST BE DOWN.
            // In this case, the VMM marks the file as "fail all I/O" and
            // all modified data to the file will now be discarded!
            FailAllIOWasSet = TRUE;
        }

        // Dereference the page.
        PTE->reference_count—;

        if (error AND not file on networked file system) {
            // Mark page as modified once again so write will be retried
            //    later.
            PTE->modified = TRUE;
        }
    } // Loop for each page.

    // FOR MAPPED FILES ONLY ...
    ReleaseFileResources(); // Resources acquired using file system
                            // callback

    // Unlock PFN database.
    …

    if (FailAllIOWasSet) {
        // The user sees the famous error message
        //    "system lost write-behind data" now.
        loRaiselnformationalHardError(STATUS_LOST_WRITE_BEHIND_DATA,
                                                FileName, Status);
    }
} // end of MiWriteComplete()
```

Note that in this pseudocode, the VMM uses an I/O Manager function loAsynchronousPageWrite() to flush modified data to secondary storage. This call will be quickly routed by the I/O Manager to the file system driver managing the mounted file system on which the target page file or mapped file resides.

The file system driver can easily recognize that the write request is a paging I/O request because the I/O Request Packet sent to the file system has the IRP_PAGING_IO and the IRP_NOCACHE flags set. Note that the file system is *not* permitted to take another page fault while resolving the paging I/O write request. The I/O Manager handles asynchronous page writes differently when performing postprocessing on IRP structures that described such paging I/O requests. Essentially, the I/O Manager invokes the MiWriteComplete() routine by means of a kernel APC upon completion of the asynchronous paging I/O IRP. The routine is invoked in the context of the MPW thread.

Page Fault Handling

The NT VMM is responsible for handling the case when contents referred to by a virtual address are not present in physical memory. Although the hardware MMU typically translates virtual addresses into physical addresses, when the MMU discovers that the PTE indicates that the page is not in memory, the MMU will turn the problem over to the VMM for resolution. The VMM routine invoked when a page fault occurs, either in kernel mode or in user mode, is called MmAccessFault(). This routine takes three arguments:

- The virtual address that caused the page fault
- A boolean argument that indicates whether a store/write operation caused the page fault (a FALSE value indicates that this was a read/load operation)
- The mode (kernel or user) in which the fault occurred

First, the MInAccessFault() routine checks the current IRQL. If it is greater than APC_LEVEL, and if either the page directory or the Page Table Entry indicates that the page is not valid, the VMM will bugcheck the system and the following message will be printed on your debugger screen:

```
MM:***PAGE FAULT AT IRQL > 1 Va %x, IRQL %x
```

The routine within the VMM that resolves page faults is appropriately called MiDispatchFault(). The MmAccessFault() routine invokes MiDispatchFault() to resolve the fault and make the contents of the page frame valid. This routine handles page faults for access to addresses in both system address space (the upper 2GB of the virtual address space) and in user process address space. Faults are dispatched for further processing to an appropriate subroutine based upon the nature of the faulting address:

- If the faulting address is backed by a page file, the routine MiResolvePageFileFault() is invoked.

  This routine performs the following tasks:

  — Allocate enough page frames in memory so that data can be read from the page file.

  — Note that this routine uses the MiEnsureAvailablePageOrWait() routine mentioned earlier in this chapter.

  — Figure out the page file to which the read operation should be directed from the PTE.

  — Build a Memory Descriptor List (MDL) containing the list of available physical pages.

— Mark the PTEs for the pages being brought into memory as being "in transition."

— Return a special status OxC0033333 to the caller, MiDispatchFault ().

Because MiResolvePageFileFault ( ) returned a status of OxC0033333, MiDispatchFault ( ) will then invoke a paging I/O read operation using the IoPageRead() call exported by the I/O Manager. Just as in the case of the paging I/O write request described in the Modified Page Writer discussion, the file system driver invoked by the I/O Manager to satisfy the page read request will recognize the request as a paging read, because of the presence of the IRP_PAGING_IO and the IRP_NOCACHE flags. Note that the file system cannot incur any page faults while trying to satisfy the paging I/O read request.

The VMM then waits for the page fault read request to be completed, and if successful, adds the page to the working set of the active process.

- If the PTE for the faulting address indicates that the page is in transition, then the MiResolveTransitionFault ( ) routine is invoked. A transition page is marked as being in-transition for one of the following reasons:

  — The page frame contains valid data, but the page was placed on the free list because it was automatically trimmed.

  — The page frame contains valid data, but it was placed on the modified list as a result of being automatically trimmed from the working set of a process.

  — The page is being actively read from secondary storage; this is a *collided page fault.*

  This routine performs the following tasks:

  — For pages that are being actively read from secondary storage, the MiResolveTransitionFault () routine will block, awaiting I/O completion. If an error occurs, it will mark the PTE invalid and return success, forcing the caller to undergo another page fault, for which the PTE will now no longer be marked as *in-transition.*

  — Otherwise, this routine will mark the transition PTE valid and will add it to the working set for the current process.

  Note that this routine will not return the status OxC0033333 since there is no page read operation to be initiated by MiDispatchFault ( ) .

- The MmAccessFault ( ) routine invokes MiDispatchFault ( ) with a prototype PTE (PPTE) to fault into memory if the faulting virtual address belongs to a shared memory range or to a memory-mapped file. In this case, MiDis-

patchFault() invokes the MiResolveProtoPteFau.lt() subroutine, which in turn performs the following tasks:

— If the PPTE belongs to a mapped file, the MiResolveMappedFile-Fault() routine is invoked to determine the set of pages to be faulted into memory, allocate an MDL and return OxC0033333. Note that the VMM attempts to cluster pages together to improve performance.

— If the PPTE was created to back up shared memory contained within a page file, the MiResolvePageFileFault() routine is invoked. This routine determines the page file number from which to perform the paging I/O read operation, builds an MDL structure that will subsequently be used to perform the read, and returns OxC0033333.

— If the PPTE indicates that it is in transition, this routine will itself invoke the MiResolveTransitionFault() subroutine discussed above.

— If a zeroed page is required, the MiResolveDemandZeroFault() subroutine is invoked.

Once an appropriate subroutine has been invoked successfully, the Mi-ResolveProtoPteFault() routine will make the PTE reflect the contents of the PPTE. Now the PTE for the process will refer to the PEN database entry for the page frame whose contents either will be read in (if OxC0033333 is returned) or are already valid if a transition fault was resolved.

- Sometimes, the VMM simply needs to materialize a page frame containing zeroes in response to a page fault. This may happen when a thread tries to extend a file on disk, or if a thread tries to access some newly allocated, committed memory. In this case, the MiDispatchFault() routine simply invokes the MiResolveDemandZeroFault() subroutine, which in turn allocates a zeroed page frame from the list of available page frames. If such a page frame is not available, the MiResolveDemandZeroFault() routine returns OxC7303001, which simply causes the fault to recur and at that time a page should become available (remember that the MPW thread is always trying to ensure that there are enough free and unmodified page frames available to be reallocated).

As you can see, the NT VMM supports the MMU in resolving virtual addresses to physical addresses by faulting in pages that are not present in system memory. If you develop a kernel-mode driver that takes a page fault at an IRQL greater than or equal to DISPATCH_LEVEL, you will cause the system to bugcheck, since the VMM will not satisfy page faults at such a high IRQL. Ensure that all code and data that is accessed at high IRQLs has been previously locked into nonpageable system memory.

Interactions with File System Drivers

The NT Virtual Memory Manager and file system drivers have mutual dependencies between them. The VMM depends on file system drivers to provide support for page file I/O and also to provide support for section objects representing memory-mapped files. The file system, in turn, depends upon the NT VMM to resolve page faults that occur within the file system driver; to manipulate user and system buffers; to be able to allocate, manipulate, and deallocate memory; and to help cache file stream data.*

Here is a list of functionality provided by the VMM to the file system drivers on NT platforms:

- The file system driver is an executable, dynamically loadable driver that is loaded into system virtual address space with the assistance of the VMM and the file system driver that contains the executable. By default, code for the file system and other kernel-mode drivers is not pageable; i.e., these drivers reside in RAM as long as they are loaded. Similarly, all global memory associated with kernel-mode drivers is never paged-out by default. There is a compiler directive that your driver can specify that will cause portions of the driver code to be marked as pageable. This pragma is defined with any NT compatible compiler as follows:

  ```
  #pragma  alloc_text (PAGExxxx,  NameOfRoutine)
  ```

  Note that *xxxx* should be a unique sequence of four characters that identifies a pageable portion (also referred to a pageable section) of code. Furthermore, at run-time, it is possible for your driver to invoke the MmLockPageableDataSection( ) or the MmLockPageableCodeSection ( ) routines to dynamically lock code or data. These routines and the corresponding unlock routines are well documented in the DDK documentation. Some information on making drivers pageable is also provided in Chapter *2, File System Driver Development.*

- File system drivers, filter drivers and device drivers all need memory that they allocate at run-time. Typically, your driver will invoke a version of the `ExAllocatePoolWithTag()` routine to request pageable, nonpageable, or cache-aligned memory. You can even request memory with the condition that failure to allocate memory should result in an automatic system panic. Although the Executive support routines manage these *pools* from which your driver obtains memory, the physical memory and its manipulation is performed only by the VMM. Any virtual address pointers (for memory) returned

* Chapter 6, *The NT Cache Manager I,* defines file streams more formally. For now, you can substitute the word *file* for *file stream* if you like.

using one of the ExAllocatePool ( ) routines is guaranteed to be in kernel virtual address space.

Note that your driver can also invoke the ZwAllocateVirtualMemoryO routine to directly request memory from the VMM, although the returned virtual address will be in the lower 2GB of the process virtual address space; therefore, such memory will only be accessible in the context of the allocating thread/process.

- Since your kernel-mode driver must be accessible while executing in the context of any process in the system, the VMM manipulates the virtual address space of every process in the system such that the lower 2GB are unique (and private) to that process while the upper 2GB are reserved as the system virtual address space and are mapped to the same physical addresses in the context of all processes executing on the system.
- As a file system or as a kernel-mode driver, your code will often need to use buffers that are passed in from user-mode code (e.g., a thread that executes in user mode allocates memory and passes this buffer down to your driver). Your driver must use this buffer to transfer data either into or out of the buffer. However, there are two problems here that your kernel-mode driver must address:
  - — Unless your driver is always guaranteed to execute in the context of the user-mode thread, your driver cannot use the virtual addresses passed in by the user-space thread, since they are only valid in that particular thread's context.
  - — Sometimes, your driver might need to access the passed in buffer at an IRQL greater than APC_LEVEL. In this case, you must ensure that the buffer is backed by locked physical pages, because a page fault will certainly result in a system crash.

  The VMM assists you in addressing both of the issues listed. Any buffer can have its associated physical pages locked in memory by invoking any of the VMM routines such as MmProbeAndLockPages(), MmBuildMdl ( ) , and other similar routines. These request the VMM to create a Memory Descriptor List (MDL), an opaque structure that describes the list of physical page frames backing your allocated virtual address range. Optionally, depending upon the VMM routine invoked, the pages will also be locked in memory; the page frames allocated to the buffer will not be reclaimed until they are unlocked. If you need to map the passed in addresses into system virtual address space, you can use the MmGetSystemAddressForMdl ( ) VMM routine.

TIP A Memory Descriptor List (MDL) is a system-defined structure that describes a virtual address range (buffer) in terms of physical pages. It contains an array, each element of which refers to a page frame index for the frame backing the virtual address range. The array is allocated immediately after the MDL structure; i.e., the MDL structure and the array (both of which are allocated from nonpaged pool) are physically contiguous in memory.

Typically, your kernel-mode driver will often request the VMM to create such an MDL for a user buffer and will usually map the buffer to system virtual address space. This ensures that the pages stay locked until you have finished processing them and that you can access the virtual addresses in the context of any arbitrary process.

The *ntddk.h* include file, supplied as part of the DDK, contains the description of the MDL data structure. Note that your driver ideally must not access the fields within the data structure directly, since they could be changed by the system.

- The VMM manages the stack frames allocated to all threads executing in the system. The stack allocated to a thread executing in kernel mode is of fixed length. In NT 3.51 and previous versions, this stack was limited to two page-frames. In Windows NT 4.0, the stack has been expanded to three 4KB pages of RAM (12288 bytes).
- The VMM assists the file system (and the NT Cache Manager) in caching file data. All of the physical memory manipulation is concentrated in the NT VMM. Therefore, the support of the VMM is actively required in using physical memory to cache byte streams, which eventually enhances system throughput.
- The VMM provides support for clustering when satisfying page faults, which helps improve system performance.

  Typically, the VMM tries to cluster I/O operations into groups of 16 pages. On Intel x86 platforms, this leads to a 64KB I/O size, while on Alpha machines, this translates to 128KB I/O operations.
- Sometimes, filter drivers need to do unusual things, like caching data to a file on a local file system. Or, user-mode code and kernel-mode drivers might need to pass data buffers between them. To solve problems like these, kernel-mode drivers and user-mode applications can use the services of the VMM to create shared memory objects or memory-mapped files.

TIP Although the focus of the book is not on designing and developing NT device drivers, you should be aware that the NT VMM utility routines and data structures (the MDL data structure and routines that manipulate it) are also applicable to device driver designers. There are other supporting routines that the VMM provides to device driver developers, most notably MmMapIoSpace ( ) , which maps a given physical address range into nonpaged system space. Consult the DDK for additional documentation on this routine as well as other supporting routines provided by the Hardware Abstraction Layer (HAL).

Remember, however, that regardless of the nature of the kernel-mode driver that you develop, you -will need to understand the contents of this chapter.

- The NT Virtual Memory Manager provides the MmQuerySystemSize ( ) support routine that can sometimes be useful to file system drivers.

  The MmQuerySystemSize ( ) function takes no arguments. It simply returns an enumerated type result that can take one of the following values:

  — MmSma11System (enumerated type value = 0)

  — MmMediumSystem (enumerated type value = 1)

  — MmLargeSystem (enumerated type value = 2)

  The value returned depends upon the amount of physical memory configured on the system. The VMM initializes a global variable, MmSystemSize, to one of these three values at system initialization time, after determining the amount of physical memory available on the node. MmQuerySystemSizeO returns the contents of this global variable.

  The actual amount of RAM that may result in one value being returned instead of another is subject to change between different versions of Windows NT. For example, if your system has less than 12MB of physical memory, you could expect to get back the MmSmallSystem value when you invoke the MmQuerySystemSize ( ) function. Similarly, if you have less than 20MB of available physical memory, you could expect to get MmMediumSystem returned.

  The MmQuerySystemSize ( ) function call is typically made by kernel-mode components to guide them in making resource allocation decisions. For example, consider the case when the MmSmallSystem value is returned as a result of calling this function. Now your file system driver may not know exactly what a "small system" really means, but you can infer that, relatively speaking, the amount of available physical memory is less than what it would be on medium or large systems. Therefore, your driver could preallocate

smaller-sized zones, or create fewer worker threads as compared to what it may do on medium or large systems. Use this routine to get additional information about the system to help determine the resource utilization within your driver.

There will undoubtedly be other factors that your driver will consider in making the final determination about the amount of resources (physical memory) your driver should consume.

The NT VMM also depends on the file system for the following functionality:

- Page files are created and manipulated on mounted file systems. Therefore, to implement virtual memory support, the VMM needs the file system to perform paging I/O read and write operations.

  As illustrated by sample code in Part 3, the file system driver must completely rely on the VMM when receiving I/O requests directed to a page file. Therefore, the file system should avoid acquiring any resources (to provide any synchronization), should never incur a page fault in processing the read/write request, should never defer the request for asynchronous processing, and should never block the request for any reason. It should simply forward the request immediately (after determining the on-disk parameters for the request) to the appropriate lower-level device drivers.

- In order to provide support for shared memory or for memory-mapped files, the VMM needs the active support of the underlying file system. First, the VMM requires that the file system provide appropriate callbacks to help maintain the locking hierarchy in the NT system. In addition, the VMM requires that the file system be prepared to receive page faults that occur as a direct consequence of the user process accessing mapped memory.

  The callbacks that must be implemented by the file system driver are the `AcquireFileForNTCreateSection()` and `ReleaseFileForNtCreateSection()`. The file system is expected to acquire all resources that might be needed while the NT VMM executes code in support of a *create section* request. I will discuss the implementation of these callbacks in detail in Part 3.

Support Routines Provided for FSD Implementations

The VMM provides two specific routines, MmFlushImageSection() and MmCanFileBeTruncated(), that are very important for file system designers, but they are not well documented. Part 3 has examples using these routines.

MmFlushImageSection ()

This function is used by a file system driver to ask the VMM to discard pages in memory containing information associated with a specified image section object. For example, consider a copy of the Microsoft Word executable file that a user mapped in to memory and now wishes to delete, maybe to upgrade the copy to a later version of the software. The file system driver must ensure, before actually deleting the file, that all pages containing file data are flushed (discarded). During normal execution, these pages may contain file stream data even after all user handles to the file have been closed. However, the file system cannot allow such information to stay around in memory if it plans to delete the file stream.

NOTE Note that the VMM enforces a restriction that a file can't be deleted as long as any user has actively mapped in the file stream; if the file is currently being executed, it cannot be deleted. However, from the discussions presented in this chapter, you have learned that the VMM keeps file data around in memory even after all handles to the mapped file stream have been closed, as long as it does not really need to reuse the physical memory. This helps achieve faster response if the user closes the file handle but reopens it soon after.

It is precisely during these situations that the file system driver must flush the system pages before proceeding with the delete operation.

This function is also invoked by a file system driver before allowing a thread to open a file stream for write access. Also, the VMM will typically not allow a user to open a file for write access if another thread had previously mapped the file into memory as an executable.

The MmFlushImageSection ( ) function is defined as follows:

```
BOOLEAN
MmFlushImageSection (
    IN PSECTION_OBJECT_POINTERS    SectionObjectPointer,
    IN MMFLUSH_TYPE                FlushType
);
```

where

```
typedef enum _MMFLUSH_TYPE {
    MmFlushForDelete,
    MmFlushForWrite
} MMFLUSH_TYPE;
```

Resource Acquisition Constraints:

The file system driver must ensure that the file stream has been acquired exclusively before invoking this function. Typically, the MainResource* for the file stream is acquired exclusively before calling MmFlushImageSection().

Parameters:

`SectionObjectPointer`

In the next chapter, the SECTION_OBJECT_POINTERS structure will be described in detail. For now, note that a unique instance of this structure type is associated with each representation of the file stream in memory. The VMM expects a pointer to this structure to be passed in to the MmFlushImageSection() function.

FlushType

This can assume one of two values, MmFlushForDelete or MmFlushForWrite. When checking whether a user open can be allowed to proceed during processing a create/open request for an on-disk file stream, the file system should pass in the MmFlushForWrite value. Before actually attempting to delete an on-disk file (in the *set file information* dispatch routine and in the *cleanup* dispatch routine, both of which are described in Part 3), the file system should pass in the MmFlushForDelete enumerated type value.

Functionality Provided:

- If the routine receives the argument MmFlushForDelete, and any user thread has mapped the file stream into its virtual address space as a regular data stream (memory-mapped file), the VMM will immediately return FALSE.
- In either case, whether MmFlushForDelete or MmFlushForWrite is passed in, if any thread has mapped the file stream into its virtual address space as an executable, the VMM will reject the request and return FALSE.
- Otherwise, the VMM will grab the page frame database lock and mark the image section object for deletion.

Once the VMM has determined that it is safe to proceed with flushing of the image section, the VMM will actually walk through the list of dirty pages contained within the section and flush them out to secondary storage if they are backed by an on-disk page file. Note that any dirty (modified) pages belonging to mapped files are simply discarded immediately. Before actually starting the flush operation, the VMM will ensure that all asynchronous modified page writer operations on the file stream have been stopped (and will actually block until any

* More information on synchronization objects associated with a file stream is provided in Part 3.

ongoing write operations have been completed). If your file system supports page files and if any of the dirty pages are backed by the page files residing on your file system, your driver should expect to receive recursive paging I/O write requests at this time.

Once the modified pages have been flushed (if required), the VMM will tear down the image section object for the file stream, making it safe for the file system to proceed with a delete or open operation.

There are two extremely important points you must be aware of before invoking this function:

- When trying to flush modified pages for the image section to a page file, the VMM will ignore any I/O errors encountered.
- The VMM will dereference the file object that was referenced when the image section object was created.

  To the file system designer, this means that your driver could receive a close request as part of the processing performed during this call. If your file system is in the middle of processing a create operation, do not be surprised to suddenly receive the last close operation on the file stream as a result of invoking this function.*

MmCanFileBeTruncatedO

This routine is provided by the VMM to help a file system determine whether a truncate operation on a file stream should be allowed to proceed. The VMM imposes certain restrictions on when file size modifications and/or deletions are allowed to proceed. A user is not allowed to truncate a file stream if the file stream is mapped in as an executable; the truncate request will be denied if an image section object has been created by the VMM and is actively being used by a user thread. The rationale is that it would confuse the user executing the file tremendously if a page fault failed because the contents corresponding to the page no longer exist on disk due to the truncate operation, although the contents existed just a moment ago. Also, if any thread has mapped the file stream as a data file (not as an image section), and if the new file size would be less than the currently mapped view length of the file stream, the VMM will disallow the truncate request.

The MmCanFileBeTruncated() function is typically invoked by the file system before allowing a truncate request to proceed. An example using this function is

* Although you will appreciate this more when you actually develop your file system driver, this could cause all sorts of problems for your driver if you have to arbitrate between tearing down file system structures (as a result of the last close being received) and using them because you are processing a create/open request.

provided in Chapter 10, *Writing A File System Driver II.* The function is defined as follows:

```
BOOLEAN
MmCanFileBeTruncated (
IN PSECTION_OBJECT_POINTERS        SectionPointer,
IN PLARGE_INTEGER                  NewFileSize
);
```

Resource Acquisition Constraints:

The file system driver must ensure that the file stream has been acquired exclusively before invoking this function. Typically, the MainResource* for the file stream is acquired exclusively before calling MmCanFileBeTruncated ( ) .

Parameters:

SectionObjectPointer

The SECTION_OBJECT_POINTERS structure is described in detail in the next chapter. Note for now that a unique instance of this structure type is associated with each representation of the file stream in memory.

NewFileSize

A pointer to a large integer containing the proposed new file size.

Functionality Provided:

- Internally, MmCanFileBeTruncated ( ) invokes MmFlushlmageSection (), supplying MmFlushForWrite as the reason for the flush request.

  If the MmFlushlmageSection () function returns FALSE, the MmCanFileBeTruncated () function will also return FALSE and deny the truncate request.

- Otherwise, the function checks if any user thread-mapped views exist; if they do and the new file size would be less than the size of the mapped view, the MmCanFileBeTruncated ( ) function returns FALSE.

- Otherwise, the function returns TRUE, allowing the truncate request.

The basic philosophy followed here is:

- If an image section is in use for the file stream, the VMM will return FALSE.
- If a user data section exists for the file stream, and if the new file size is less than the size of the currently mapped view, the VMM will return FALSE.
- If neither of the two conditions above are found to be TRUE, the VMM will return TRUE.

---

* More information on synchronization objects associated with a file stream is provided in Part 3.

This chapter presented the Virtual Memory Manager. The next three chapters will cover the NT Cache Manager, a kernel-mode component that assists file system drivers in caching data. This component depends heavily upon the NT VMM and is explicitly supported by it.