In this chapter:

- *Functionality*
- *File Streams*
- *Virtual Block Caching*
- *Caching During Read and Write Operations*
- *Cache Manager Interfaces*
- *Cache Manager Clients*
- *Some Important Data Structures*
- *File Size Considerations*

6

The NT Cache Manager I

Although constant advances in storage technologies have led to faster and cheaper secondary storage devices, accessing data off secondary storage media is still much slower than accessing data buffered in system memory. Therefore, to achieve greater performance with applications that manage large amounts of data (e.g., with database management applications), it becomes important to have data brought into system memory before it is accessed (*read-ahead functionality),* to retain such information in memory until it is no longer needed (*caching of data),* and possibly to defer writing of modified data to disk to obtain greater efficiency *(write-behind* or *delayed-write* functionality).

Most modern operating systems provide support for some form of file data caching.* This task is traditionally performed by individual file systems or by modules such as the systemwide buffer cache on UNIX systems. In the Windows NT operating system, the NT Cache Manager encapsulates the functionality required to cache file data.t In order to perform this task, the Cache Manager interacts with file system drivers and with the NT Virtual Memory Manager. The Cache Manager is an integral component of the Windows NT environment. By simply using Windows NT to access file data, each of us utilizes the services provided by the Cache Manager. If our requests to access data seem to be satisfied fairly quickly, without even accessing the disk drive, we know that the Cache Manager

* Even the maligned Microsoft DOS environment featured the (in)famous SmartDrive caching module.

t Actually, the Cache Manager caches *byte streams* (without interpretation), which can be stored on disk using any layout defined by the file system. Therefore, file system metadata can also be cached by the NT Cache Manager.

worked hard to preread our data into system memory. If requests to copy files or modify them return almost instantaneously, it is probably because the modified data was buffered in memory. When we notice that the hard disk shows activity periodically (every few seconds), we realize that modified data is being lazy-written to disk. And finally, when we lose data as a result of a system crash, it is quite evident that the Cache Manager must be to blame.*

In this chapter, as well as in the next two, I will present the NT Cache Manager in detail, focusing on the responsibilities of the Cache Manager, the methodology used by it to buffer data, and also the interactions of the Cache Manager with the NT file system drivers and the NT Virtual Memory Manager.

Functionality

The NT Cache Manager is a distinct component of the NT Executive, and it is closely affiliated with the Virtual Memory Manager.

It provides a consistent systemwide cache for data stored on secondary storage devices.

This cache is managed in conjunction with the appropriate file system drivers, and with the cooperation of the Virtual Memory Manager and the I/O Manager.

It performs read-ahead on file data.

The Cache Manager attempts to tune its read-ahead policy per file based on the pattern of data access performed by user applications. Since all I/O requests on buffered files get routed through the Cache Manager, the Cache Manager can keep track of the access pattern for the data belonging to the file. Therefore, if a user application reads (say) the first 10K bytes for a file, the NT Cache Manager will typically try to read ahead the next 64K bytes of the file into memory. Subsequently, if the application attempts to obtain this data, it can simply be copied over from the system cache, thereby avoiding making the user application wait until the data can be read from secondary storage. For sequentially accessed files, the *read-ahead* functionality provided by the Cache Manager can result in significant performance gains, since data will have already been read into system volatile memory before the application requests access to such data.

---

* By accepting (and requiring) greater throughput via caching in volatile system RAM, users accept the risks associated with such caching. Typically, unplanned system outages (perhaps due to failure of hardware components or errors in the software) result in the loss of modified data that had not been flushed to secondary storage. Although it is possible to use nonvolatile memory to cache data, the associated costs with such usage are prohibitive for most environments.

It provides delayed-write functionality for modified cached data.

By keeping modified data in memory for some time before actually writing it to disk, the Cache Manager provides greater responsiveness to the user applications that actually perform the write. It can also batch multiple contiguous write operations in memory and write all the modified bytes out in a single I/O operation, which is typically more efficient than performing each smaller write operation individually. Finally, it is possible that a user application may repeatedly modify the same byte range. By deferring I/O to disk, such modifications are made only in memory, avoiding completely the overhead of repeated write operations to the media.

File Streams

Each instance of an open file is represented by a file object structure in Windows NT. Any linear stream of bytes associated with a file object can be defined as a *file stream.* Examples of file streams include the data for the given file,* a directory (containing information about other files stored on disk), file system *metadata* (such as volume information), Access Control Lists (ACLs) associated with the file, and extended attributes stored with the file.

NT file systems create, delete, and manipulate file streams as the result of either externally generated user requests to read or write file data, or internally generated requests to manipulate file-system-specific data structures. File systems identify file streams that they wish to support and cache. For example, unless directed otherwise by a user, file systems cache user data contained within a file. For each file stream to be cached, the file system typically supports both cached and noncached access.

The Cache Manager provides support for the *caching of file streams* by using memory mapping, and it also integrates caching with the memory manager's policies for other uses of pageable memory. From the perspective of the Cache Manager, the stream is simply a random sequence of bytes representing information that should be kept in memory. Therefore, the same set of services offered by the Cache Manager can be used by file system drivers to cache user file data or file system metadata.

* Some files could have multiple data streams if the file system supports this feature. For example, NTFS supports multiple data streams. NTFS uses two distinct byte streams (for the same named file) to store the resource and data forks associated with Macintosh files stored on NTFS file systems on NT servers.

Virtual Block Caching

Some operating systems use physical offsets (or disk block addresses) to cache file data in system memory. Instead of using disk block addresses, the NT Cache Manager provides a *virtual block cache* by using the *file mapping method* for caching file streams. Figure 6-1 illustrates the difference between these two methods for data caching (for buffered data). Note that the numbering indicates the logical sequence in which the operations are performed.

In operating systems that use physical block addressing for cached data (the *old buffer cache* implementation in UNIX SVR4), the file system or caching module must first convert virtual byte offsets in a file to physical block offsets on disk before checking whether data is available in the system cache, since the caching module—the buffer cache—keeps track of cached data by using physical disk addresses. However, as shown in Figure 6-1, the NT Cache Manager only uses virtual byte offsets in a file to keep track of cached information. The Cache Manager does not need to understand physical block addresses for the data being accessed. Therefore, file system drivers in the Windows NT operating system generally translate virtual byte offsets in a file to physical block offsets on disk only if the data could not be obtained from the in-memory cache being managed by the Cache Manager.

The advantages of using a virtual block cache (as compared to a physical block cache) follow:

- Some applications may use native NT system calls to access file data, e.g., `NtReadFile()` or `NtWriteFile()`,* while other applications executing concurrently may map the file data into their address space for read or read/write access. By using virtual block caching, via file mapping, and by using proper synchronization, it is possible for all such applications to see the most current data.t
- Conceptually, there is no difference between file data mapped in by the NT Cache Manager compared to file data mapped in by an application. By using the file mapping model, all physical memory becomes available for data caching. As mentioned before, the allocation of physical memory is controlled by the NT Virtual Memory Manager; the number of physical pages allocated to

---

* Typically, applications use the interfaces provided by a subsystem (e.g., the `ReadFile()` interface provided by the Win32 subsystem) to perform read/write operations. Invoking such interface routines eventually results in calls to native NT system services. For a comprehensive listing of system services provided by the I/O Manager for data access, see Appendix A, *Windows NT System Services.*

t Note that neither the FASTFAT nor the NTFS native file system implementations currently **guarantee** that applications using conventional system calls will always obtain the most current data if other applications have also mapped the file for read/write access. However, in most cases, the file systems go to considerable lengths to ensure that this is indeed the case.

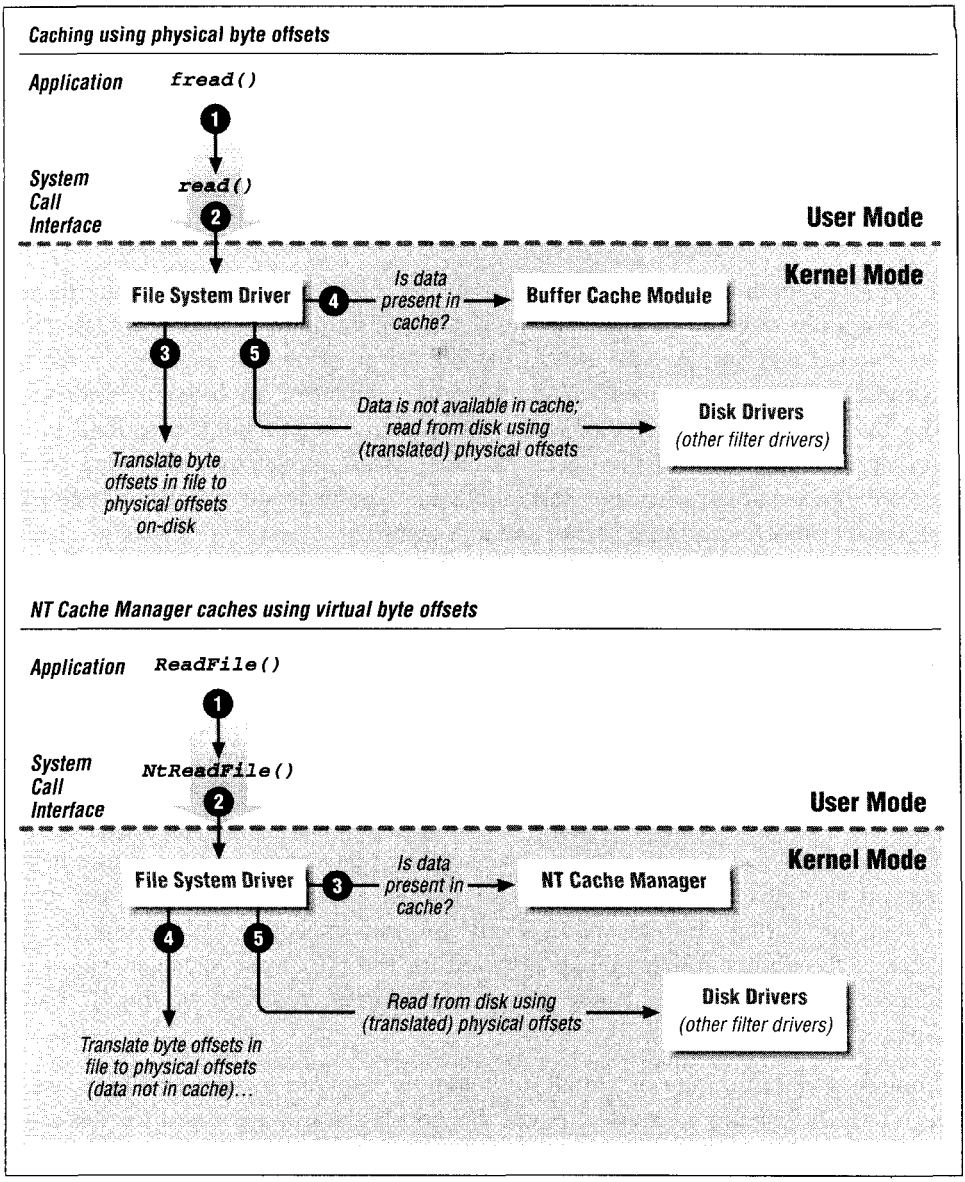

Figure 6-1. Comparison of virtual block address caching with physical block address caching

the Cache Manager depends on changing needs for memory by other components in the system (e.g., memory allocated for image file pages versus data file pages).

- Often, the I/O Manager invokes the Cache Manager directly, bypassing the file system driver or the network redirector driver completely. In such cases, it

is possible for the Cache Manager to resolve the file access via a single hardware virtual address lookup.* This is considerably more efficient than the process of converting a virtual address to a physical disk address before checking whether data is available in system buffers.

Caching During Read and Write Operations

In the NT operating system, user processes are allowed to specify at the time of opening a file whether data for the file should be buffered in memory. Only those files opened without the IRP_NOCACHE flag—to indicate that data for the file can be buffered—have their data cached in system memory. In order to understand how the NT Cache Manager provides the caching functionality described in the previous section, think of the Cache Manager as an application, executing on the system, which happens to open the very same files as those opened by all of the other applications executing on the same system.

In order to cache data, the Cache Manager has to utilize system memory. As was noted in Chapter 5, *The NT Virtual Memory Manager,* each process executing in the Windows NT environment has 4GB of virtual address space available to it. The lower half of this address space is process-specific, while the upper 2GB are reserved for the operating system and are shared for every process executing in the system. This virtual address model applies also to the system process, which is a special process created at system initialization time. At system initialization, the Cache Manager reserves a range of virtual addresses within the upper 2GB of the system process virtual address space. Since this virtual address range that is reserved for the exclusive use of the NT Cache Manager exists within the upper 2GB of the virtual address space, every process executing on the system has access to the virtual address range reserved for the NT Cache Manager. Figure 6-2 depicts the location of the range of virtual addresses reserved by the NT Cache Manager.

Although a certain range of virtual addresses is reserved for the exclusive use of the NT Cache Manager, physical pages are not necessarily allocated for this range of virtual addresses. The number of physical pages that are allocated to the Cache Manager is determined, and constantly adjusted, by the NT Virtual Memory Manager. In the absence of demand for physical memory from other user processes or system components, the Virtual Memory Manager may choose to increase the amount of physical memory allocated to the Cache Manager. On the

* Virtual address translation can be immediately performed using the Translation Lookaside Buffer (TLB). A TLB hit results in extremely efficient translation to the corresponding physical memory address.

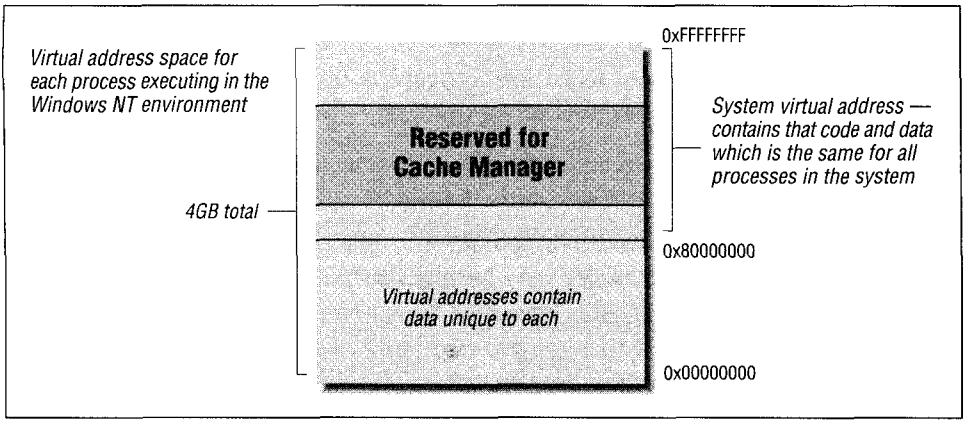

Figure 6-2. Virtual address range reservedfor the NT Cache Manager

other hand, on heavily loaded systems with scarce available physical memory, the memory manager may decrease the amount of physical memory allocated to the Cache Manager for caching file data.

It is important to note that these decisions concerning physical memory allocation are the sole prerogative of the NT Virtual Memory Manager.

The Cache Manager application uses *file mapping* to buffer file data. Caching is initiated on a file stream by a file system driver through a call to the Cache Manager. Upon receiving such a request, the Cache Manager, invokes the Virtual Memory Manager to create a section object representing the file mapping—this is done for the entire file stream. Subsequently, when a process attempts to access data belonging to the stream, the Cache Manager dynamically maps views of the file stream into portions of the virtual address space reserved for itself in the system virtual address space. Note that since the range of virtual addresses reserved for the Cache Manager is fixed, the Cache Manager may have to unmap one or more previously mapped views in order to be able to create a new view.

In order to better understand the role played by the Cache Manager in servicing I/O requests, let's examine the typical sequence of steps executed in response to user-initiated read and write operations.

Cached Read Operation

Consider a read operation initiated by a user application. This read operation is passed on to the file system by the NT I/O Manager.* Figure 6-3 illustrates the

* As shown later, the file system is bypassed by the I/O Manager in many cases. However, for simplicity, let's assume that I/O operations are first sent to the file system driver by the I/O Manager subsystem.

sequence of operations executed to satisfy the read request (using the *copy interface** provided by the Cache Manager).

An explanation for each step listed in the figure is provided below. Note that the arrows in the figure represent flow of control.

1. The user application executes a read operation, which causes control to be transferred to the I/O Manager in the kernel.
2. The I/O Manager directs the read request to the appropriate file system driver using an IRP. The user buffer may be mapped into the system virtual address space, or the I/O Manager may allocate a Memory Descriptor List representing the buffer and lock pages associated with this MDL, or the virtual address for the buffer may be passed-in unmodified by the I/O Manager. In Part 3 you will see that the file system driver has control over which of these operations is performed by the I/O Manager.
3. The file system driver receives the read request and notices that the read operation is directed to a file that is opened for buffered access. If caching has not yet been initiated for this file, the file system driver initiates caching on the file by invoking the Cache Manager. In turn, the Cache Manager requests the Virtual Memory Manager to create a file mapping (section object) for the file to be cached.
4. The file system driver passes the read request to the NT Cache Manager using the CcCopyRead ( ) t Cache Manager call. The Cache Manager is now responsible for executing all the necessary steps to transfer data into the user's buffer.
5. The Cache Manager examines its data structures to determine whether there is a mapped view of the file containing the range of bytes requested by the user. If no mapped view exists, the Cache Manager creates one.
6. The Cache Manager simply performs a memory copy operation from the mapped view into the user's buffer.
7. If the mapped view of the file is not backed by physical pages containing the required data, a page fault occurs and control is transferred to the Virtual Memory Manager.
8. The VMM allocates physical pages that will be used to contain the requested data‡ for which the page fault occurred and then issues a *noncached paging I/O*

---

* Later in this chapter, I will discuss the various interface methods presented by the NT Cache Manager to other system components. The *copy interface* is one of the four available interfaces.

t See the next two chapters for a detailed discussion on all the routines exposed by the Cache Manager.

‡ In order to free up physical memory, the Virtual Memory Manager may need to write modified pages to disk. For now, assume that unmodified free pages are available.

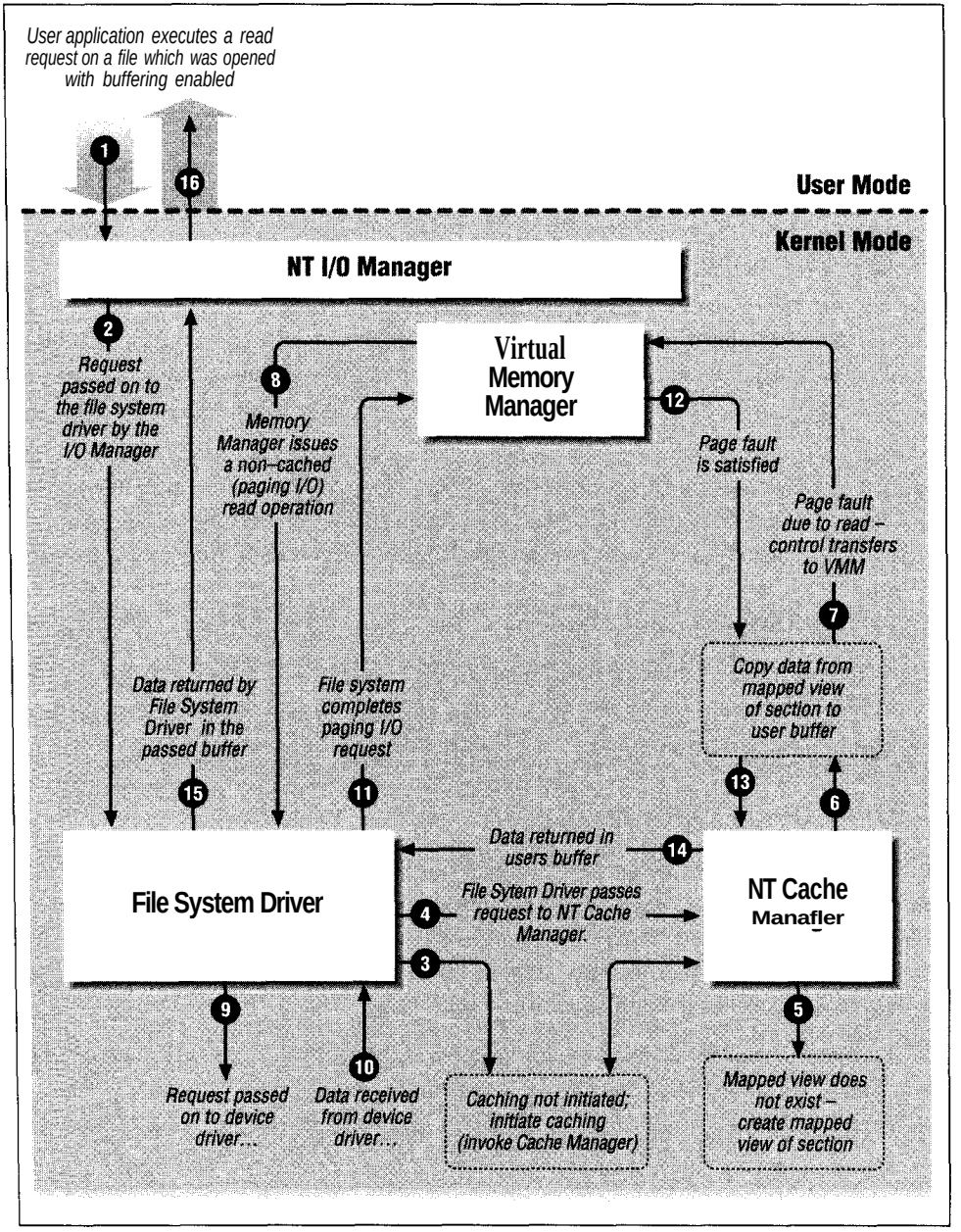

Figure 6-3. Sequence of steps executed to satisfy a user read request for a cached file

read operation to the file system driver via the NT I/O Manager. Note that although the figure above does not indicate that the paging I/O request is routed via the NT I/O Manager, that is indeed what happens.

9. Upon receiving the noncached read request, the file system driver creates a corresponding I/O request to obtain data off secondary storage media and sends this I/O request to the lower-layer drivers.
10. The device driver(s) below the file system obtain data from secondary storage (or from across the network) and complete the request.
11. The file system driver completes the paging I/O request from the NT Virtual Memory Manager.
12. The instruction that resulted in a page fault is reexecuted.
13. The Cache Manager completes the copy operation from the mapped view for the file to the user's buffer. This time, the copy should complete without incurring a page fault (although it is theoretically possible to have a page fault repeatedly on a page that has just been brought in, practically speaking, this does not occur).
14. The Cache Manager returns control to the file system driver after the cached data has been copied into the user's buffer. Note that this data will also remain cached in the virtual address space reserved for the Cache Manager (however, this data may be discarded from system memory by the NT Virtual Memory Manager at any time).
15. The file system driver completes the original IRP sent to it by the NT I/O Manager.
16. The I/O Manager completes the original user read request.

Cached Write Operation

Now, consider a write operation initiated by a user application. Figure 6-4 illustrates the sequence of operations executed to satisfy the write request (using the *copy interface* provided by the Cache Manager).* As you will see, the sequence of operations is similar to the read operation described previously. An explanation for each step listed in the figure is provided below:

1. The user application executes a write operation, which causes control to be transferred to the I/O Manager in the kernel.
2. The I/O Manager directs the write request to the appropriate file system driver using an IRP. As in the case of the read operation, the buffer may be mapped into the system virtual address space, or an MDL may be created, or

* The figure has been deliberately simplified for the sake of clarity. As you will see in Chapter 9, *Writing a File System Driver I,* in order to account for incomplete block transfers, write operations may cause the file system to actually read data from disk before executing the write.

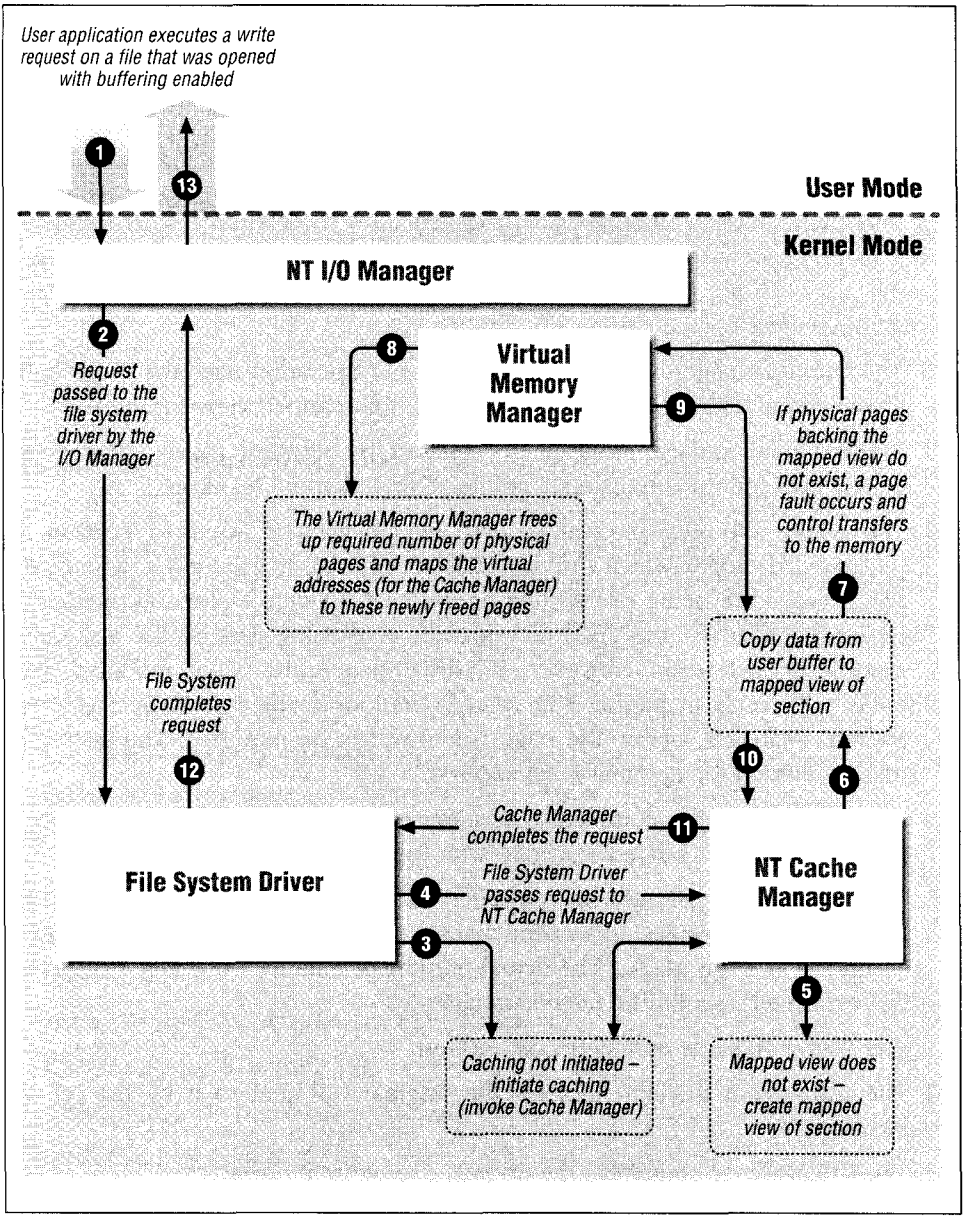

Figure 6-4. Sequence of steps executed to satisfy a write request for a cached file

the virtual address for the buffer may be passed to the file system driver without any modifications.

3. The file system driver notices that the write operation is directed to a file that is opened for buffered access. As shown in the example for a read operation,

if caching has not yet been initiated for this file, the file system driver initiates caching on the file by invoking the Cache Manager. The Virtual Memory Manager creates a file mapping (section object) for the file to be cached.

4. The file system driver simply passes on the write request to the NT Cache Manager via the CcCopyWrite ( ) Cache Manager call, which is part of the copy interface made available by the Cache Manager.
5. The Cache Manager examines its data structures to determine whether there is a mapped view for the file containing the range of bytes being modified by the user. If no such mapped view exists, the Cache Manager creates a mapped view for the file.
6. The Cache Manager performs a memory copy operation from the user's buffer to the virtual address range associated with the mapped view for the file.
7. If this virtual address range is not backed by physical pages, a page fault occurs and control is transferred to the Virtual Memory Manager.
8. The VMM allocates physical pages, which will be used to contain the requested data (for which the page fault occurred). In Figure 6-4, assume that entire pages are being overwritten by the user. In such a scenario, neither the Cache Manager nor the VMM read previously existing data off the disk before modifying such data. However, if partial pages are being modified, page faults will result in paging I/O *read* operations being issued by the Virtual Memory Manager, before the page is allowed to be modified. The instruction that resulted in a page fault is reexecuted.
9. The Cache Manager completes the copy operation from the user's buffer to the virtual address range associated with the mapped view for the file.
10. The Cache Manager returns control to the file system driver. Note that the user data now resides in system memory and has not yet been written to secondary storage media. The actual transfer of data to secondary storage will be performed later by the Cache Manager.*
11. The Cache Manager completes the request.
12. The file system driver completes the original IRP sent to it by the NT I/O Manager.
13. The I/O Manager completes the original user write request.

---

* Either the lazy writer component of the Cache Manager or the modified page writer component of the Memory Manager may initiate the write to secondary storage media. Also, it is possible that a user request to flush system buffers or a flush initiated by the file system driver (due to some reason such as a cleanup operation) may be responsible for instigating the write operation to disk. The lazy writer component will be covered in greater detail in the next chapter. Refer to Chapter 5 for more details on the modified page writer.

Cache Manager Interfaces

Now that we have explored how caching is typically used by file system drivers, let us look at the different ways in which system components can use the NT Cache Manager. File system drivers and other components in the Windows NT operating system can use the services provided by the Cache Manager through four sets of interface routines. The first set of interface routines provides support for basic file stream access and manipulation, while the other three can be used as different access methods for the system cache.

The four sets of interfaces provided by the NT Cache Manager are file stream manipulation functions, the copy interface, the MDL interface, and the pinning interface.

File Stream Manipulation Functions

The Cache Manager provides support for initializing cached operations for a file stream, terminating caching, flushing cached data to disk (on demand), modifying file sizes, purging cached data, zeroing file data, support for logging file systems,* and other common maintenance functions. The functions provided by the Cache Manager within this interface set consist of the following:

- `CcInitializeCacheMap`
- `CcUninitializeCacheMap`
- `CcSetFileSizes`
- `CcPurgeCacheSection`
- `CcSetDirtyPageThreshold`
- `CcFlushCache`
- `CcZeroData`
- `CcGetFileObjectFromSectionPtrs`
- `CcSetLogHandleForFile`
- `CcSetAdditionalCacheAttributes`
- `CcGetDirtyPages`
- `CcIsThereDirtyData`
- `CcGetLsnForFileObject`

* Some file systems (e.g., the NTFS file system) use a method called *logging to* enable faster recovery and ensure metadata integrity upon a system crash (or any unexpected shutdown). These file systems need to ensure a certain sequence in which log entries and corresponding file metadata/data are written to disk. The Cache Manager provides support for such file system drivers via the routines listed above.

Copy Interface

The copy interface is the simplest form of cached access. The client module, using the Cache Manager, can utilize this interface to copy either a range of bytes from a buffer in memory to a specified virtual byte offset in the cached file stream, or a range of bytes from a specified virtual byte offset in the cached file stream to a buffer in memory.

This interface includes a call to initiate read-ahead and also includes calls to support *write throttling.* Write throttling allows the client of the Cache Manager (usually a file system driver) to defer certain write operations if the system is running low on available or unmodified pages. This condition can occur if some applications keep modifying data at an extremely rapid rate, greater than the rate at which the lazy writer or modified page writer can initiate the transfer of modified data to disk or across the network to a storage server. Note that it is also quite possible that the disk or network driver may not be able to keep pace with the rate at which I/O requests to write data to disk are being generated by the modified page writer or the lazy writer. This would also result in a decrease in the number of available, unmodified pages.

The functions provided by the Cache Manager within this interface set consist of the following:

- `CcCopyRead/CcFastCopyRead`
- `CcCopyWrite/CcFastCopyWrite`
- `CcCanlWrite`
- `CcDeferWrite`
- `CcSetReadAheadGranularity`
- `CcScheduleReadAhead`

MDL Interface

A Memory Descriptor List (MDL) is an opaque Memory-Manager-defined data structure that maps a particular virtual address range to one or more paged-based physical address ranges. The MDL interface to the Cache Manager allows direct access to the system cache via Direct Memory Access (DMA).* The set of routines comprising the MDL interface return an MDL to the caller, containing the byte range described in the request, which can be subsequently used by the caller to transfer data directly into or out of the system cache.

* DMA allows a device controller to transfer data directly between system memory and a secondary storage device. The processing unit doesn't get involved in the data transfer, resulting in better performance.

This interface is useful to subsystems that need direct access to the contents of the system cache. For example, network file servers that need to DMA across the network device directly into or out of the Cache Manager's virtual address range use the MDL interface to achieve higher performance. In the absence of this interface, a network driver transferring data out of the system cache might first have to allocate a temporary buffer, copy data from the system cache to this temporary buffer, let the network device perform the transfer, and, finally, deallocate the temporary buffer. The extraneous calls to allocate/deallocate the temporary buffer and the redundant copy can all be avoided if data can be transferred by the network device directly from the system cache across the network. This can indeed be achieved using the CcMdlRead ( ) and CcMdlReadComplete ( ) sequence of calls.*

Note that this interface shares the same read-ahead call as the copy interface. Also, routines comprising the MDL interface and those belonging to the copy interface can be used concurrently on the same file stream. The functions provided by the Cache Manager within this interface set consist of the following:

« CcMdlRead
- `CcMdlReadComplete`
- `CcPrepareMdlWrite`
- `CcMdlWriteComplete`

An interesting point to note here is that, while most of the other Cache Manager routines associated with data transfer (e.g., CcMdlReadO, CcCopyRead()) perform data transfer as part of the functionality provided by the routine, the CcPrepareMdlWrite() routine simply creates an MDL containing original data, which can be subsequently modified by the caller prior to invoking CcMdlWriteComplete ( ). Therefore, although some data transfer might be performed by the Cache Manager when CcPrepareMdlWrite ( ) is invoked (to obtain current file stream data from disk or across the network and place it in the pages described by the MDL), the routine acts more as an enabler routine, allowing the caller to transfer the new data later, using the returned MDL.

Pinning Interface

This interface provided by the Cache Manager can be used to perform two tasks:

- Map data into the system cache for direct access using a buffer pointer
- Pin (or lock) the physical pages that back the mapped data

---

* The terminology used here is important: CcMdlRead ( ) is used when the client wishes to read from the system cache and write to the network (or disk). CcPrepareMdlWrite ( ) is used when the client wishes to transfer directly from the network device *(or* disk) and write to the system cache.

In addition to being able to read data directly using a buffer pointer, the caller can also modify the data directly in the system cache.

When access to the mapped data is no longer required, the data can be unpinned. This will also result in locked pages being unlocked and made available for other uses. Once the data is unpinned, the pointer to the data should no longer be used.

Pinning data is typically used for efficiency reasons when file system drivers or other system components need to access frequently used data structures (or other data associated with the file stream) directly in memory. It is also used to ensure that the data being accessed cannot be removed from system memory. However, locking mapped data consumes physical memory and therefore decreases the amount of memory available to other system components.

Note that the pinning interface cannot currently be used in conjunction with either the copy interface or the MDL interface.

This interface is often used by file system drivers when dealing with cached file system metadata. The pinning interface consists of the following functions:

- CcMapData
- `CcPinMappedData`
- `CcPinRead`
- `CcSetDirtyPinnedData`
- `CcPreparePinWrite`
- `CcUnpinData`
- `CcUnpinDataForThread`
- `CcRepinBcb`
- `CcUnpinRepinnedBcb`
- `CcGetFileObjectFromBcb`

The above functions are described in greater detail in Chapter 7, *The NT Cache Manager II.*

Cache Manager Clients

The following components are typical users of the interfaces provided by the Cache Manager. These components are also known as *clients* of the Cache Manager.

- File system drivers such as NTFS, FASTFAT, CDFS, and other third-party file systems use the copy interface services of the Cache Manager to perform each-

ing on user file data. This allows for greater performance, because once user data is cached in system memory, subsequent access to the data can be satisfied immediately without getting the data again from secondary storage media.

File system drivers also use the Cache Manager to cache file system metadata, including volume structures, directory information, bitmaps for free space on disk, extended attributes associated with a file, and other similar information. Many of these structures are often pinned in memory by the file system driver. Note that the Cache Manager does not interpret the type of data streams being cached; it only knows about file object data structures and data streams associated with such file objects.

File system drivers also typically use the read-ahead and delayed write functionality provided by the Cache Manager, although it is quite possible that certain sophisticated file system implementations may add their own support for read-ahead or delayed write operations. Finally, all file system drivers have to use the file stream manipulation functions provided by the Cache Manager to interface correctly with the Cache Manager.

- Network redirectors are similar to file system driver implementations; however, these modules obtain data from file servers across a network, instead of from a secondary storage medium directly attached to the host system. These components typically cache various data streams in the system cache to provide extremely fast performance comparable to local file systems.

  Network redirectors typically use the copy interface provided by the Cache Manager. They may also use the MDL interface to DMA data directly into or out of the system cache. These components also benefit from the read-ahead and write-behind functionality provided by the Cache Manager. In order to initiate or terminate caching on specific data streams or to perform other cache manipulation functions, network redirectors use the file stream manipulation functions.

- Network File Servers are indirect clients of the Cache Manager, since they use the local file systems to ultimately obtain access to file data. These drivers never invoke Cache Manager routines directly. File servers are often implemented as kernel-mode drivers for performance reasons. They use the copy interface via the file system drivers that serve their requests. Also, file servers typically use DMA to transfer data directly into (or out of) the system cache. To do this, file servers use the MDL interface to the Cache Manager. Since file servers cannot directly invoke the Cache Manager, they use special flags in read/write IRPs sent to file system drivers to request that a memory descriptor list be created for the specified virtual address range in the file stream. After data transfer has been completed, file servers inform file system drivers that previously created memory descriptor lists can now be deleted. Chapter 9,

contains an explanation of the flags used by file servers to request the creation and deletion of MDLs for data buffered in the system cache.

- Filter drivers, or other drivers that use the NT file system interface for specialized purposes, are indirect clients of the Cache Manager. Consider a filter driver that provides hard disk caching for data stored on slower media such as magnetic tape or optical media. Such a driver uses the services of a local file system to store the cached information. Therefore, the filter driver is an indirect client of the Cache Manager, since the file system supporting the filter driver uses the copy interface to transfer data into and from system memory. Similarly, consider a filter driver that provides HSM* functionality. Such a driver has to migrate data from a relatively fast secondary storage device, such as a magnetic disk, to a slower device, such as tape. To help speed up the process, the filter driver uses DMA to transfer data directly from the system cache to tape and, therefore, uses the MDL interface (via special flags in read/write IRPs sent to the file system driver) provided by the Cache Manager. After the transfer process has completed, the filter driver will inform the file system driver that any previously created memory descriptor lists can now be deleted.

Table 6-1 summarizes the way clients of the Cache Manager use its various interfaces.

Table 6-1. Clients of the Cache Manager

	Local File Systems	Network Redirectors	Network File Servers	Filter Drivers
File Stream Manipulation	✓	✓		
Copy Interface	✓	✓	✓	✓
MDL Interface		✓	✓	✓
Pinning Interface	✓			

Some Important Data Structures

The services provided by the Cache Manager are most heavily utilized by file system drivers and network redirectors, which serve user I/O requests. The data

* HSM or Hierarchical Storage Management involves efficient management of available storage using configurations comprising faster and more expensive media along with slower but cheaper media, to minimize cost per byte of stored data and yet have data always available when required. Typically, this is performed by automatically transferring infrequently accessed data to slower, cheaper media, such as tape, from the faster (but more expensive) hard disks. When such data is subsequently accessed, the driver automatically transfers data back from tape to hard disk. There are other aspects to HSM that are outside the scope of this discussion.

structures and fields described below are important to understand to interface correctly with the Cache Manager.

Fields in the File Object

As explained in Chapter 4, *The NTI/O Manager,* each file stream, when created or opened, has a file object structure (of type FILE_OBJECT) created for it by the I/O Manager. Although most of the fields within the file object structure are filled in by the I/O Manager, the file system drivers and network redirectors that are the recipients of the I/O requests on the associated file stream are required to fill in certain specific fields. Three important fields that must be initialized follow:

- The FsContext field
- The SectionObjectPointer field
- The PrivateCacheMap field

This initialization is typically performed at file stream open (or create) time; it is possible, though, for a file system or network redirector to defer this operation to some other time before caching is first initiated for the file stream.

FsContext

If caching via the NT Cache Manager is required for an open file stream (represented by the file object structure), the FsContext field must be initialized to point to a structure of type FSRTL_COMMON_FCB_HEADER. This structure is defined as follows:

```
typedef struct _FSRTL_COMMON_FCB_HEADER {
 CSHORT              NodeTypeCode;
 CSHORT              NodeByteSize;
 UCHAR               Flags;
 UCHAR               IsFastIoPossible;
 // ************************************************************
 // The following two fields are only present in Version 4.0+ of the
 // the Windows NT operating system.
 //  Second Flags Field.
 UCHAR               Flags2;
 //  The following reserved field should always be 0.
 UCHAR               Reserved;
 // ************************************************************
 PERESOURCE          Resource ;
 PERESOURCE          PagingloResource;
 LARGE_INTEGER       AllocationSize;
 LARGE_INTEGER       FileSize;
 LARGE_INTEGER       ValidDataLength;
} FSRTL_COMMON_FCB_HEADER;
```

The above structure will be referred to as the CommonFCBHeader structure. It has to be allocated by the file system or network driver from nonpaged kernel

memory. As you will see in Chapter 9, each file stream is uniquely represented in memory by a File Control Block (FCB) structure.

NOTE For readers with a UNIX background, note that a File Control Block is analogous to a UNIX *vnode* structure representing a file (or directory) in memory.

Although multiple concurrent open operations performed on the same file stream may result in multiple file object structures being created, there is only one unique FCB for the file, and all file object structures must refer to it.

Similarly, only one CommonFCBHeader structure can exist per file stream. Therefore, it is not uncommon to see file system driver or network driver implementations allocate the CommonFCBHeader structure as part of their FCB structure representing the file stream. Note, however, that the file system driver is not required to allocate the CommonFCBHeader as part of the FCB structure as long as a one-to-one (unique) logical association can be created between these two structures.

The first two fields in the CommonFCBHeader—NodeTypeCode and NodeByteSize—are unused by the Cache Manager. The fields comprising this structure are described below. Note that many of these fields require the understanding of concepts explained in later chapters (specifically Chapters 9-11); the issue of initialization of each of these fields will be revisited when all such required concepts have been presented:

Flags

The CommonFCBHeader structure has pointers to two synchronization ERESOURCE type structures. The PagingloResource is acquired by the modified page writer thread. By setting an appropriate value in the Flags field, the file system driver or network redirector is allowed to specify to the MPW thread that the MainResource (see below) should be acquired instead of the PagingloResource. In Chapter 11, *Writing a File System Driver III,* reasons why a file system driver or a network redirector may set such a flag will be discussed.

Flags2

This field was added with Version 4.0 of the operating system. As discussed later in this book, it is possible for an FSD to specify that lazy-write operations not be performed for a cached file stream. However, if the Flags2 field has the FSRTL_FLAG2_DO_MODIFIED_WRITE flag set (defined as 0x01), the Cache Manager will ignore the FSD request to disallow delayed operations and perform lazy-write I/O for the file stream.

IsFastIoPossible

For efficiency reasons, the I/O Manager attempts to bypass the file system driver or network redirector for cached files and tries to obtain file data directly from the Cache Manager. This process is called the fast I/O process. The IsFastIoPossible field allows the file system driver or network redirector to control whether fast I/O operations should be allowed to proceed for the specific file stream. The contents of this field are set by the file system driver or network redirector and can be one of the following three enumerated types: FastIoIsNotPossible, FastIoIsPossible, or FastIoIsQuestionable.

Resource and PagingIoResource

Access to data associated with a file stream must be synchronized using these ERESOURCE structures.*

This is a requirement for file system drivers and network redirectors in order to be able to interface correctly with the Cache Manager and Memory Manager components.

Memory for both resources must be allocated by the file system or network redirector from nonpaged pool, and the fields in the CoiranonFCBHeader must be initialized to point to the allocated structures. These structures must also have been initialized by the FSD via the ExInitializeResourceLite() executive support routine.

Since these resources provide shared reader and exclusive writer semantics, the Cache Manager expects the file system driver or network redirector to synchronize all modifying operations for the file stream by obtaining the MainResource exclusively. Similarly, read operations can be synchronized by obtaining the MainResource shared.

AllocationSize

This is the actual amount of on-disk storage space allocated for the file stream. Typically, this is a multiple of the media sector size or file system cluster size.† This field must be initialized by the file system driver or network redirector to the appropriate value. Subsequently, the Cache Manager must be notified each time this value changes. In the next chapter, you will see how the file system driver notifies the Cache Manager of changes in the allocation size.

---

* See Chapter 3, *Structured Driver Development,* for a discussion on various synchronization structures available under Windows NT including a discussion on ERESOURCE type structures.

† Space is allocated on secondary storage devices in units called sectors. Each sector is composed *of* a fixed number of bytes—for example, one sector may equal 512 bytes. To avoid fragmentation, some file system drivers allocate storage space using clusters as units, where each cluster is some number of sectors. For example, one cluster may equal 8 physical sectors.

FileSize

This is the size of the file as presented to the user; this value indicates the number of bytes contained within the file stream. Any read operations beyond this value will result in an end-of-file (STATUS_END_OF_FILE) error message being returned to the application process. Any read operations that overlap this value will be truncated at this value.

For example, if the FileSize is 45 bytes and the reader wishes to obtain (say) 30 bytes beginning at offset 40 in the file stream, only 5 bytes will actually be returned to the reader by the file system driver (or the Cache Manager). However, if the same reader wishes to read 30 bytes beginning at offset 45 (assuming that offsets are counted beginning at offset 0), an error STATUS_END_OF_FILE will be returned to the reader.

The file system driver or network redirector initializes this field to an appropriate value and informs the Cache Manager whenever this value changes.

ValidDataLength

Consider a situation where the FileSize for a file stream is 100 bytes. However, only the first 10 bytes of the file stream have valid data and the last 90 bytes were never written to by any process. The ValidDataLength for this file stream is then set to 10. Any read operations that attempt to access bytes beyond this range will automatically get zeroes returned to them. This helps avoid unnecessary I/O operations from disk and also helps provide data security (since older information stored on the media from some previous file stream is not inadvertently returned to the user).

Few file systems maintain the concept of a ValidDataLength stored on disk associated with a file stream. The NTFS and the HPFS file system drivers supplied with the NT operating system do support this concept. However, regardless of whether the file system driver supports the valid data length concept, the Cache Manager expects the file system driver or network redirector to initialize this field to an appropriate value.

SectionObjectPointer

This field has to be initialized to point to a structure of type SECTION_OBJECT_POINTERS.* This structure must be allocated from nonpaged kernel memory by the file system driver or network redirector and is shared by the Virtual Memory Manager and the Cache Manager. It stores file-mapping and caching-related information for a file stream. This structure has the following format:

```
typedef struct _SECTION_OBJECT_POINTERS {
```

* This structure is also required by the Virtual Memory Manager to provide support for memory-mapped files. See Chapter 5 for details on memory mapped files.

```
    PVOID       DataSectionObject;
    PVOID       SharedCacheMap;
    PVOID       ImageSectionObject;
} SECTION_OBJECT_POINTERS;
typedef SECTION_OBJECT_POINTERS *PSECTION_OBJECT_POINTERS;
```

Only one structure of this type can be associated with a given file stream at any time. However, it is entirely possible, and very probable in the case of user-opened files, that multiple file objects, each representing an open instance of a given file stream, can exist simultaneously on the node. In this case, all of the SectionObjectPointer fields in each file object structure must be initialized with the address of the single allocated structure of this type. Therefore, this structure is typically associated with the FCB for the file stream.

Upon allocation, it is the responsibility of the client of the Cache Manager to clear all fields within the SECTION_OBJECT_POINTERS data structure. After clearing the structure, the client does **not** need to be concerned anymore with the manipulation of any of the fields. An explanation of fields contained in this structure follows (remember that only the VMM or Cache Manager can manipulate these fields):

DataSectionObj ect
: This pointer is used by the Virtual Memory Manager to refer to an internal data structure representing a data section object created for the file stream. Therefore, this field is initialized by the Virtual Memory Manager when caching is initiated for the file stream.

SharedCacheMap
: The Cache Manager creates private data structures called cache maps to keep track of the views mapped for the specific data stream. This field is initialized by the Cache Manager with the address of the SharedCacheMap structure (described later in this section) when caching is initiated for the file stream.

ImageSectionObject
: The Virtual Memory Manager initializes this field with the address of a private data structure whenever an image section is created for the file stream.

PrivateCacheMap

The client of the Cache Manager is expected to initialize this field to NULL for each file object structure. Note that multiple file object structures may exist concurrently in memory for a given file stream. It is also possible that caching may have been initiated by some, but not all, file object structures.

We know that file system drivers, network redirectors, and other clients of the Cache Manager work in cooperation with the Cache Manager to present a consistent view of the data to all users; this is done for those threads that access data

using the cached path as well as for those who do not. The only way for a file system driver or network redirector to determine whether caching has been initiated using a specific file object for a given file stream is to examine whether the PrivateCacheMap field is nonnull. This check must only be performed after acquiring the MainResource, either shared or exclusively.

Information on whether caching has been initiated on a file stream via a specific file object cannot be maintained elsewhere by a client. This is because the Cache Manager retains the right to forcibly terminate caching via some or all file objects associated with the file stream. Therefore, as mentioned earlier, the fact that the PrivateCacheMap field is nonnull is the only reliable indicator for the client that caching is currently initiated via the file object structure being examined.

Cache Maps

The Cache Manager must maintain information about each file stream for which it helps to cache data. This information is maintained using *Cache Maps.* For each file stream, the Cache Manager allocates a *Shared Cache Map* structure that serves as the anchor for all information regarding views mapped for the file stream and other information associated with the file stream. This shared cache map structure is allocated when caching is first initiated for the file upon the request of a file system driver or a network redirector.

In addition to the shared cache map structure that is unique for each file stream and therefore allocated only when caching is first initiated for a file stream, each time a client issues a request to initiate caching using a specific file object structure, the Cache Manager allocates a *Private Cache Map* structure. This structure serves as a marker for the Cache Manager, establishing the fact that caching has been initiated using the specific file object. It also contains some private information for the Cache Manager for read-ahead control and other such data.

Note that both the private cache map structure and the shared cache map structure are allocated and maintained by the Cache Manager.

Buffer Control Blocks

One of the interfaces presented by the Cache Manager and mentioned previously is the pinning interface. Clients of the Cache Manager that use this interface must use the *Buffer Control Block* structure. This structure is divided into two parts: a *public BCB,* that is exposed to clients of the Cache Manager, and a *private BCB* that is internal to the Cache Manager.

The public BCB is defined as follows:

```
typedef struct _PUBLIC_BCB {
```

```
    CSHORT              NodeTypeCode;
    CSHORT              NodeByteSize;
    ULONG               MappedLength;
    LARGE_INTEGER       MappedFileOffset;
} PUBLIC_BCB, *PPUBLIC_BCB;
```

The public BCB is extremely simple and serves as a context to the Cache Manager client—to be used in the pinning and subsequent unpinning of data. Upon return from a successful request to the Cache Manager by the file system driver or network redirector to pin data for a file stream, a pointer to the BCB structure is returned by the Cache Manager. Memory for this BCB structure is allocated by the Cache Manager.

The file system driver uses the pointer to the BCB structure in an opaque manner: the MappedLength and MappedFileOffset provide information to the client about the actual offset, beginning where the data has been pinned in memory and the number of bytes of data that were pinned.

Subsequent requests by the client to repin the memory structures or to unpin the memory must be performed using the BCB pointer as a context, which is returned to the Cache Manager. As will be explained in the next chapter, it's possible for the BCB returned by the Cache Manager to change across different Cache Manager invocations when the BCB is passed in as context. Therefore, the client must not attempt to make and use a copy of the returned BCB structure. The private portion of the BCB is not exposed by the Cache Manager.

File Size Considerations

There are three different file size values:

- The AllocationSize for a file stream is a value that reflects the actual on-disk space reserved for the file stream, which is a multiple of the minimum allocation unit for the media on which the file stream resides.
- The FileSize for a file stream is the value beyond which all read operations return an end-of-file error.*
- The ValidDataLength is the amount of valid data contained within a file stream.

  Any bytes accessed beyond this value (up to the FileSize) contain invalid data and should result in zeroes being returned to the application trying to read this information.

* Note that it is entirely possible that, for certain file system implementations, the FileSize may be greater than the AllocationSize. This happens when the file system driver supports sparse file implementations. None of the file systems supplied with Windows NT currently support sparse files.

There are two important considerations for Cache Manager clients who change one or more of these file sizes.

One cardinal rule all clients must follow is that changing the AllocationSize or the FileSize must be synchronized with other read/write requests and that the Cache Manager must be immediately informed of any changes.

Synchronizing changes in the FileSize with other read/write requests is accomplished by ensuring that the FCB for the file stream has been acquired exclusively while performing such a change. Both the MainResource as well as the PagingIoResource must be acquired exclusively before changing either of the file size values. The CcSetFileSizes() routine, which is invoked with the FCB for the file acquired will inform the Cache Manager.

The rationale behind the above rule is simple: the file system driver (or network redirector) is often bypassed by the I/O Manager, which tries to transfer data to or from a file stream directly, using the Cache Manager via the fast I/O path. In such cases, if the Cache Manager is not correctly notified of FileSize changes, invalid results may be returned to the application trying to perform the data transfer.* For example, if the current file size is extended by an application but the Cache Manager is not informed of the new file size, it is quite possible that the application will receive a STATUS_END_OF_FILE error when trying to read information from beyond the old end-of-file offset. This is incorrect and could result in data corruption.

A second important point to note is that changes in the FileSize are generally not synchronized with paging I/O read or write requests. Note that paging I/O requests generally originate either from the lazy writer or modified block writer components, or are a result of direct user read/write operations on mapped files. While paging I/O requests are dealt with in greater detail in Chapters 9-11, the reader should be cognizant of the following:

- Paging I/O read requests starting beyond end-of-file are completed with a STATUS_END_OF_FILE error.
- Paging I/O read requests that start before the current end-of-file but extend beyond current end-of-file are truncated to the current end-of-file byte offset. However, the client must be careful to set the number of bytes written to be the same as the number of bytes initially requested (although no I/O was actually performed).

* In certain cases, when the file is being truncated, the Cache Manager or the Memory Manager may refuse to allow the operation to proceed. This topic will be dealt with in greater detail in Chapter 10, *Writing A File System Driver II*. However, it is important to note that the file system driver or network redirector must coordinate changes in the file size for a file stream with the Cache Manager module.

- Paging I/O write requests that start beyond the current end-of-file must be voided by the file system driver or network redirector and STATUS_SUCCESS should be returned.

The ValidDataLength concept is supported by few file system drivers on disk. If the file system driver supports and records the ValidDataLength value on disk, it should initialize the CommonFCBHeader with the current value when the file stream is first opened. Subsequently, the Cache Manager will inform the file system driver when this value changes and the file system driver or network redirector can then record the modified value on disk. Note that the Cache Manager may have been invoked directly by the I/O Manager to service a user write request that could have resulted in a change in the valid data length.

The Cache Manager informs the client of the change in the valid data length via the SetFileInformation IRP. This IRP and the method used by the Cache Manager to notify the client will be discussed in greater detail in Chapter 10.

If the client does not support the concept of a valid data length on disk and therefore does not wish to receive notification from the Cache Manager about changes in this value, the client must initialize the ValidDataLength field as follows: the low 32 bits of the valid data length must be initialized to OxFFFFFFFF and the high 32 bits of the field must be initialized to Ox7FFFFFFF.

Even if the client does not record the valid data length on disk, it might still be useful to the client to maintain the valid data length while the file stream stays open. Consider the situation where a user process extends the file length. Subsequently, the user process issues a write request beyond the old end-of-file byte offset. This request will be directed to the Cache Manager, which will first try to fault the page in while trying to get a page ready to receive the user data. This page fault will eventually need to be serviced by the file system driver or network redirector. If the file system driver maintains the concept of the valid data length in-memory, it could recognize that no read operation was required since the file stream had just been extended and zeroes can be returned immediately to complete the page fault request.