7

The NT Cache Manager II

In this chapter:

* *Cache Manager Structures*
* *Interaction with Clients (file Systems and Network Redirectors)*
* *Cache Manager Interfaces*

In the previous chapter, you were introduced to the NT Cache Manager module, which provides a global cache for file streams, along with read-ahead and delayed-write functionality. As was noted in that chapter, the Cache Manager cannot provide such functionality by itself, but must work in conjunction with the Virtual Memory Manager, the I/O Manager, and each file system or network redirector driver to boost throughput and increase system performance.

In this chapter, as well as in the next one, the interfaces presented by the Cache Manager are examined in much greater detail. First I present an overview of Cache Manager data structures used internally by the Cache Manager to maintain state information for cached file streams. You were exposed to some of these structures in the previous chapter; in this chapter, you will see how the Cache Manager tries to maintain a consistent in-memory representation of all information associated with the cached file streams.

Then I describe further the interactions between the Cache Manager and its clients, specifically file system drivers and network redirectors. This includes an introduction to the resource acquisition constraints that must be followed by the Cache Manager as well as by the client, followed by a detailed examination of the steps involved in initiating caching for file streams; code examples are used to make the material more concrete and applicable in real-world development environments.

Although the routines exported by the Cache Manager are the primary means of interaction between the Cache Manager and the file system drivers, there are also callback routines exported by Cache Manager clients, which are in turn invoked by the Cache Manager. I present some information on these routines in this chapter. Further discussion on callbacks exported by file system drivers will also be presented in Chapter 11, *Writing a File System Driver III.*

A detailed listing (with descriptions and examples) of the copy interface, the pinning interface, and the MDL interface concludes the chapter.

Cache Manager Structures

The Cache Manager maintains information for each file stream on which caching has been initiated. Before examining the interactions between the Cache Manager and other system components, it will be useful to understand some of the data structures used internally by the Cache Manager to maintain the required state information for cached file streams. Very little information is currently publicly available on the data structures used by the Cache Manager, and it is also likely that these data structures will continue to change and evolve in new releases of the Windows NT operating system. However, it is quite instructive to get an overall sense of the manner in which the Cache Manager keeps track of cached file streams.

The I/O Manager creates a file object structure for every successful open operation on a file stream. For every file object on which caching has been initiated, the Cache Manager maintains caching-related state information:

- A private cache map structure for each file object
- The shared cache map structure, which is shared by all file objects representing the same file stream

The private cache map structure is allocated by the Cache Manager for each file object when caching is initiated using that file object. It is unique to the file object, and therefore multiple private cache maps can exist concurrently for an open file stream. On the other hand, only one shared cache map structure is allocated by the Cache Manager when caching is first initiated for a file stream via some file object. This shared cache map is used by all open instances for the file stream. The shared cache map is accessible indirectly via the `SectionObjectPointer` field in the file object structure.

Recall from the previous chapter that the Cache Manager provides caching services by mapping views of the file stream. Each mapped view of the file is represented internally by the Cache Manager in a structure called the *Virtual Address Control Block (VACB).* The mapping granularity—or the size of each mapped view for every file stream—is set to a constant value by the Cache Manager and therefore is the same for each VACB. This constant value determines how large the Cache Manager makes each *window* into the file stream. The Cache Manager maintains a global array of VACB structures and allocates VACBs to a specific file stream on an as-needed basis.

The shared cache map structure is the primary repository of caching information for a file stream and is maintained by the Cache Manager.

All VACBs associated with the same file stream are accessible to the Cache Manager using the shared cache map structure. Each VACB contains the virtual address associated with the view, as well as the starting offset in the file stream. This allows the Cache Manager to quickly determine whether a mapped view already exists containing the byte range requested by the user. If no such view exists, the Cache Manager can create a new view and allocate a VACB to represent it.* The list of VACBs associated with the file stream is accessible using an array of VACB pointers associated with the shared cache map. Since VACB structures are allocated from a fixed-size global pool of VACBs, it is possible that the Cache Manager may not have any free VACBs to allocate to a file stream when a view needs to be created. In this case, the Cache Manager may need to unmap a previously mapped view for a file stream (this could be from the same file stream that requires a new view to be mapped in or it could be from another file stream), remove the VACB from the linked list of VACBs allocated for the file stream, and then reassign the VACB to the new file stream. However, this operation is typically not required, since VACBs are freed whenever file close operations are performed, and a free VACB is generally available whenever required.

As shown in Figure 7-1, all private cache map structures for a cached file stream are linked together, and this list of private cache maps is anchored by a field in the shared cache map structure for the cached file stream.

This layout also allows the Cache Manager to keep track of all file objects that have the file stream cached, since the private cache map structure is always associated with a corresponding file object that represents an instance of the file stream opened for cached data access. As will be explained later in this chapter, in some situations the Cache Manager might need to forcibly terminate caching previously initiated using different file objects for a specific file stream. The Cache Manager must be able to get to each file object that has the file stream cached.

Now that you have some understanding of the structures used internally by the Cache Manager, we can examine the various routines exported by the Cache Manager and the interactions between the Cache Manager and file system drivers or network redirectors.

---

* Note that a byte range accessed by a user application may be quite large and may span multiple VACBs (since the size of the view associated with a VACB is a constant). However, the Cache Manager can still quickly determine what portions of the requested byte range are already contained in a mapped view of the file (if any such view exists) and what subset needs to be mapped in.

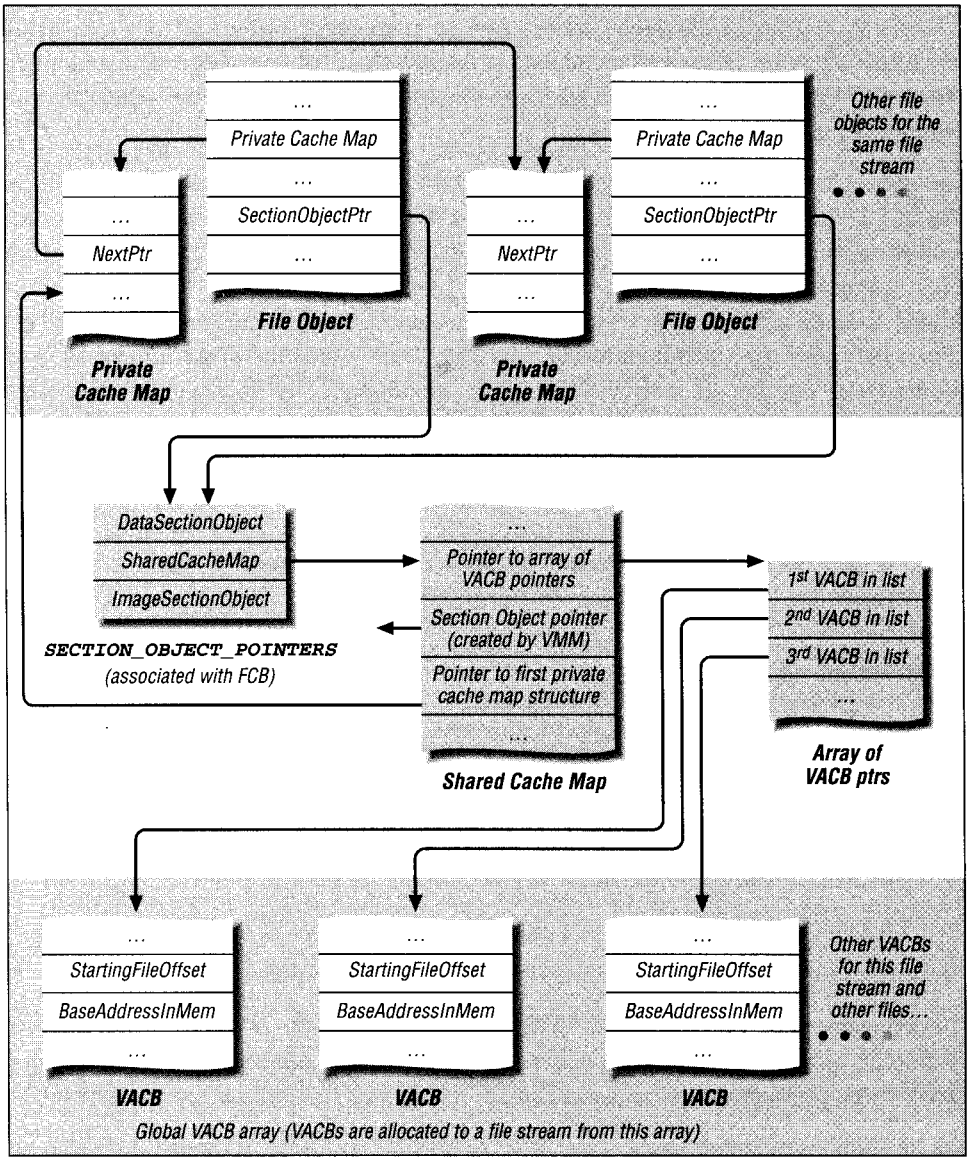

Figure 7-1. State maintained by Cache Manager for a cached file stream

Interaction with Clients (File Systems and Network Redirectors)

File system and network redirector drivers interact heavily with the Cache Manager; they must initiate caching for every file object for each file stream that

can be buffered, use an appropriate Cache Manager interface routine to transfer data to and from the system cache, service page fault requests from the Virtual Memory Manager (caused by the Cache Manager), flush or purge data belonging to a file stream from the cache, and finally, terminate caching when the file stream is no longer being accessed.

To perform these operations, file system and network redirector clients use the interface routines made available by the Cache Manager. Nearly all interface routines available to file system or network redirector drivers result in an operation being performed on the cached information for a specific file stream. Since many threads could concurrently attempt to manipulate data for a file stream, any file system using Cache Manager services must correctly synchronize all such concurrent operations. Synchronization is maintained by following well-defined rules describing how mutual exclusion can be maintained whenever data for a file stream is modified. At the same time, applications that share data for read operations should be allowed to proceed concurrently only if no other thread is modifying the data. Therefore, many Cache Manager interface routines can also be invoked concurrently on behalf of multiple threads reading data for the same file stream.

Resource Acquisition

As you know, each file stream is uniquely represented in memory by a File Control Block (FCB) structure. In the previous chapter, you saw that each FCB must be associated with a unique structure of type FSRTL_COMMON_FCB_ HEADER. There are two important fields contained within this structure:

- `MainResource`
- `PagingloResource`

Both of these fields contain pointers to objects of type ERESOURCE.

In order to synchronize correctly with the Cache Manager and the Virtual Memory Manager, all I/O operations to a file stream, including reading or writing file data or file size changes, must be synchronized using one or both of these resources.*

* For some third-party file system or network driver implementations, there may be additional synchronization primitives associated with a file stream that may need to be acquired. Although the NT environment does not prohibit the existence of such additional primitives, these .should be acquired (and released) in some manner compatible with the requirements placed by the Cache Manager on the two ERESOURCE type objects. For example, some file systems might have a third resource that may have to be acquired exclusively to provide mutual exclusion between threads when the file stream is being modified. In this case, this third resource would have to be acquired in addition to the predefined resources (i.e., the MainResource and/or the PagingloResource) mentioned here.

NOTE In any multithreaded or multiprocessor environment, shared objects that are accessed in the context of more than one thread or process must be protected using a synchronization primitive. This ensures that the state of the shared object does not change unexpectedly in the midst of an operation involving the object.

Synchronization primitives of type ERESOURCE (as described in Chapter 3, *Structured Driver Development)* are read/write locks that help provide multiple reader, single modifier semantics. By acquiring the synchronization primitive exclusively, a thread is able to ensure that no other thread can concurrently access the shared data object. On the other hand, by acquiring the synchronization primitive shared, multiple threads can concurrently read the data comprising the shared object, but no thread can acquire the synchronization primitive exclusively and modify the shared object.

Since starvation is a possibility for threads requiring exclusive access, the NT operating system typically grants waiting requests for exclusive access over requests for shared access.

A final note: in order to ensure data integrity and consistency, all threads accessing the shared data object must follow the resource acquisition rules described here. None of the synchronization is automatic and failure to observe the rules by any single thread could potentially lead to data corruption. Therefore, it is the file system driver's responsibility to ensure that resources are acquired correctly in the context of the thread requesting the cached I/O operation.

For each interface routine exported by the Cache Manager, there are well-defined options describing how the file stream should be acquired:

- Resources for the file stream should be acquired exclusively.
- Resources should be acquired shared.
- Resources should not be acquired (or should be *unowned).*
- The Cache Manager is not affected by the state of the resources.

Although the Cache Manager requires that synchronization be performed using the two resources associated with the FCB, there are not any clear, specific rules governing how these resources should actually be used to provide the required synchronization. For example, acquiring an FCB representing a file stream exclusively may consist of one of the following actions:

- Acquire the MainResource exclusively
- Acquire the PagingloResource exclusively
- Acquire both the MainResource and the PagingloResource exclusively

In this case, to prevent deadlock, a locking hierarchy must be defined between the two resources. Typically, most file systems define a hierarchy in which the MainResource must be acquired before the PagingloResource is acquired.

Similarly, acquiring an FCB for shared access might be implemented by the Cache Manager client as acquiring any one or both of the resources shared. A determination of the exact usage of these resources is made by each file system or network redirector, based on the requirements of the particular driver.

Typically, the PagingloResource is acquired only while servicing paging read operations or during delayed write (paging I/O write) operations. For example, if the file system driver read routine is invoked to service a page fault request, the FCB for the file stream is acquired shared, by acquiring the PagingloResource shared. The MainResource, on the other hand, is typically used by the Cache Manager client to service requests that execute in the context of user threads (or as a result of direct user requests). For example, a write request executing in the context of the originating user thread is synchronized by acquiring the MainResource for the file exclusively.

NOTE Each FSD has unique requirements that influence -when and how resources should be acquired to ensure correct synchronization of FSD data structures. In general, however, the Windows NT environment appears to favor usage of the PagingloResource to synchronize most modifications to file state (e.g., file size changes) while the MainResource appears to be used mostly to synchronize user-initiated I/O requests with each other.

Sometimes, the Cache Manager client may acquire both resources before performing an action of the file stream. For example, truncation of a file stream is performed only after both the MainResource and the PagingloResource have been exclusively obtained. This prevents any unwanted side effects from taking place, since file size changes are typically not otherwise synchronized with background delayed-write or read-ahead activity that might be in progress. As mentioned previously, whenever both resources need to be acquired simultaneously, a well-defined locking hierarchy should dictate the order in which the two resources are acquired. For the remainder of this book, we will define the hierarchy such that the MainResource is acquired before the PagingloResource.

In Part 3, the rules governing resource acquisition for file streams will be discussed in greater detail.

Prerequisites to Initiation of Caching

Now that you have a fair idea of how caching is provided by the Cache Manager, it is time to begin exploring the sequence of steps undertaken by Cache Manager clients to interact with the Cache Manager and provide higher performance to user applications. The previous chapter lists the various kinds of modules that interact with the Cache Manager; the two specific clients that use Cache Manager services directly are file systems and network redirector drivers.

Fundamentally, both disk-based file systems and network redirectors provide similar functionality to user applications, namely, access to data streams stored as files on media. The difference is that network redirectors obtain data from servers residing on other nodes across the network, while local file systems simply use the services of disk drivers to obtain data from media directly attached to the node on which the request was initiated. For the remainder of this chapter, we will not differentiate between the two kinds of modules, except where absolutely necessary, and will refer to both types of drivers genetically as file system drivers.

At driver initialization: fast I/O support

Typically, I/O requests for a file are conveyed by the I/O Manager to the file system driver using I/O Request Packets (IRPs). However, the overhead associated with the creation, completion, and destruction of IRPs is sometimes an inhibitor of good performance. Also, if data is cached by the Cache Manager, it is possible that such data could be directly obtained from the system cache by directly issuing a request to the Cache Manager instead of going through the file system driver.- Since the Cache Manager can then directly access data within the system cache, such access is as fast as a single hardware lookup (using the *Translation Lookaside Buffer* to convert the virtual address into a physical memory address), which is extremely efficient. The desire to achieve better system performance by taking into consideration the factors mentioned here led to the creation of the fast I/O method for obtaining cached file data in the Windows NT environment.*

Fast I/O is only performed if the file stream is cached and it is always a synchronous operation. An interesting point to note is that if data transfer is not possible using the fast I/O path for a specific operation on a file stream, the I/O Manager simply resorts to using the standard IRP method to retry the operation. This is no

---

* It could legitimately be argued that the entire fast I/O interface was a last minute hack or addition to the I/O subsystem in response to some serious performance problems encountered during testing by the Windows NT development group. Whether this is true is difficult to say, unless confirmed or denied by engineers at Microsoft. However, the fast I/O interface seems to have measurably enhanced throughput in the I/O path, and will continue to exist for the foreseeable future unless some major revamping of the Cache Manager module is undertaken by Microsoft.

worse than the original method of always creating an IRP to communicate with the file system driver to service a user request. Figure 7-2 illustrates the flow of execution when fast I/O is used to satisfy user requests.

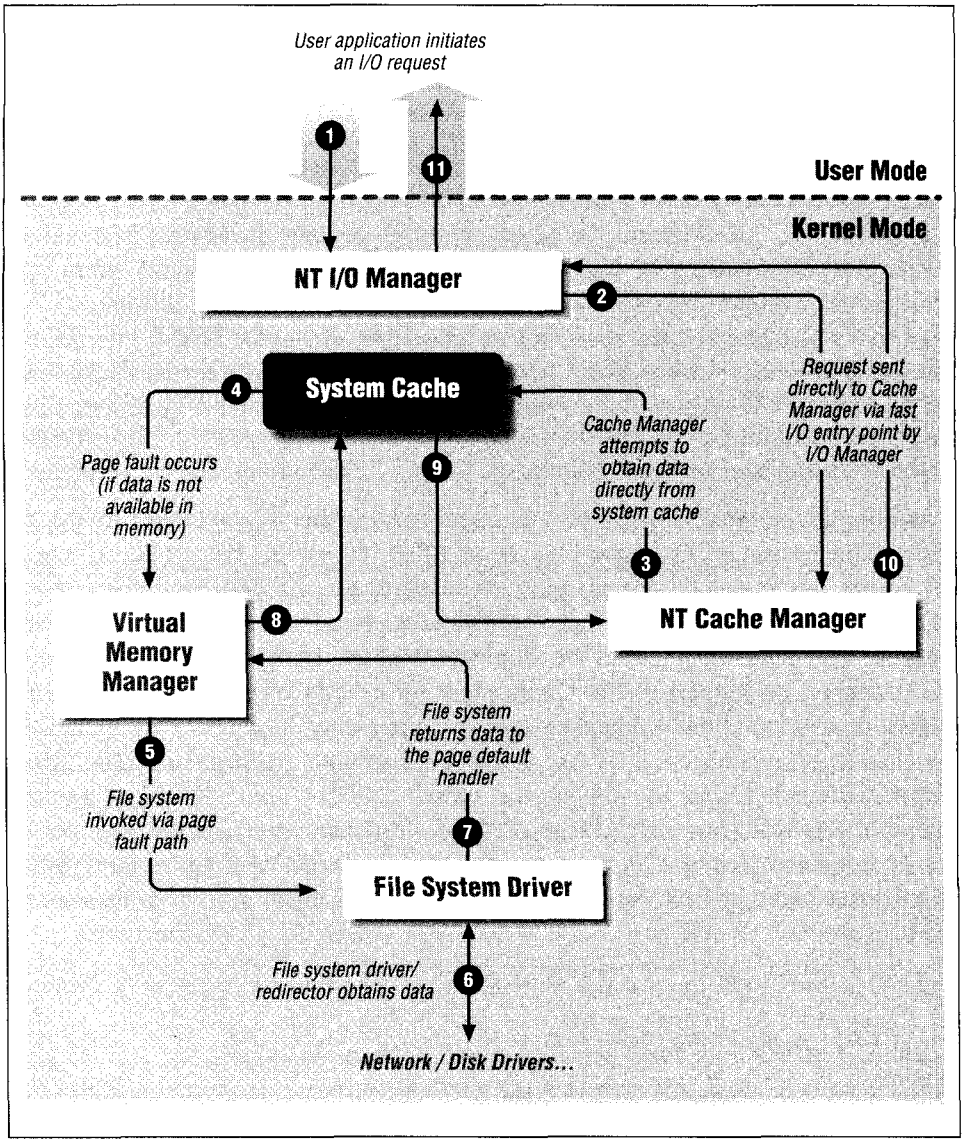

Figure 7-2. I/O requests using the fast I/O path

In Figure 7-2, the following steps are performed:

1. The I/O Manager receives a user request to read or write data for a specific file object. The file object represents an open instance for a file stream.
2. The I/O Manager invokes the fast I/O read or write entry point, which causes the corresponding Cache Manager entry point to be invoked. Note that typically the Cache Manager copy interface is used to obtain the data.
3. The Cache Manager attempts to transfer data from or to the system cache. If data exists in the system cache and is present in memory, Step 9 is executed. Otherwise, execution continues with Step 4 below.
4. A page fault occurs, causing the memory manager page fault handler routine to be invoked.
5. The page fault handler routine calls into the file system driver entry point using an I/O Request Packet. Although the figure does not show this, the actual call into the file system driver entry point is via the I/O Manager loCallDriver ( ) routine.
6. The file system driver uses the services of disk and network drivers to transfer data.
7. The file system satisfies the page fault request and control returns to the page fault handler.
8. The page fault is satisfied and the Cache Manager data transfer operation is restarted.
9. The Cache Manager completes the data transfer.
10. The Cache Manager returns control back to the I/O Manager (via the fast I/O entry point).
11. The I/O Manager completes the user request synchronously.

As you might have noticed, data that is physically present in memory can be transferred extremely quickly to or from the user's buffer. However, if data is not already physically present in memory, a trip through the file system will eventually result as a consequence of the page fault that must be resolved. This is not conducive to quick response times and is typically not required, due to the read-ahead performed by the Cache Manager.

Providing support for fast I/O is not required from file system drivers, and file systems have the option of not supporting fast I/O or of disabling fast I/O support for certain file streams dynamically. However, the resulting performance degradation is evident, especially when data transfer rates are compared with file systems that do provide fast I/O support.

To provide fast I/O support, the file system driver must perform the following actions, generally at driver initialization time:

- Initialize a global/static structure of type FAST_IO_DISPATCH. This structure contains a list of pointers that must be initialized to functions implementing each of the fast I/O entry points.
- Initialize a pointer within the DRIVER_OBJECT structure to refer to the fast I/O dispatch table described above.

There are specific operations that can be executed using the fast I/O method. The list of possible operations differs between the various NT versions. Specifically, Windows NT Version 4.0 supports more operations using the fast I/O method than Windows NT Version 3.51. Further information on the implementation of fast I/O support is given in Part 3.

The following code fragment illustrates the two steps described above:*

```
// Declare a static global fast I/O structure that contains function
// pointers. The fast I/O structure here is contained within a global
// data structure type declaration.
typedef struct _SFsdData {
    SFsdIdentifier           NodeIdentifier;

    // Other fields that you will read about in subsequent chapters.
    //  ....

    // The NT Cache Manager, the I/O Manager, and this code will
    // conspire to bypass IRP usage using the function pointers
    // contained in the following structure
    FAST_IO_DISPATCH            SFsdFastIoDispatch;

    // Still more fields . . .
} SFsdData, *PtrSFsdData;

// Declare all the functions that we will implement to support the
// fast I/O path. In this example, only read and write operations are
// supported via the fast I/O method.
extern BOOLEAN SFsdFastIoCheckIfPossible (
    IN FILE_OBJECT             *FileObject,
    IN PLARGE_INTEGER           FileOffset,
    IN ULONG                    Length,
    IN BOOLEAN                  Wait,
    IN ULONG                    LockKey,
    IN BOOLEAN                  CheckForReadOperation,
    OUT PIO_STATUS_BLOCK        IoStatus,
    IN DEVICE_OBJECT           *DeviceObject
    );
extern BOOLEAN SFsdFastIoRead (
```

* All of the routines are prefixed with *SFsd* to conform to the convention used by the sample file system driver code provided in Part 3.

```
    IN FILE_OBJECT             *FileObject,
    IN PLARGE_INTEGER           FileOffset,
    IN ULONG                    Length,
    IN BOOLEAN                  Wait,
    IN ULONG                    LockKey,
    OUT PVOID                   Buffer,
    OUT PIO_STATUS_BLOCK        loStatus,
    IN DEVICE_OBJECT           *DeviceObject
);
extern BOOLEAN SFsdFastloWrite (
    IN FILE_OBJECT             *FileObject,
    IN PLARGE_INTEGER           FileOffset,
    IN ULONG                    Length,
    IN BOOLEAN                  Wait,
    IN ULONG                    LockKey,
    IN PVOID                    Buffer,
    OUT PIO_STATUS_BLOCK        loStatus,
    IN DEVICE_OBJECT           *DeviceObject
);

// Driver Entry routine - this is where all of the initialization takes
// place.
NTSTATUS SFsdDriverEntry(
IN PDRIVER_OBJECT DriverObject,  // created by the I/O subsystem
IN PUNICODE_STRING RegistryPath) // path to registry key for the driver
{
    // Initialize the global data structure. Note that we will
    // end up zeroing out the fast I/O dispatch structure as well.
    // This will save us setting individual fields to NULL.
    RtlZeroMemory(&SFsdGlobalData, sizeof(SFsdGlobalData) );

    // Other initialization operations . . .

    // Initialize the IRP major function table, and the fast I/O table.
    SFsdlnitializeFunctionPointers(DriverObject);

    // Still more initialization stuff . . .
}

void  SFsdlnitializeFunctionPointers (
PDRIVER_OBJECT        DriverObject)        /* created by the I/O sub-
system */
t
   PFAST_IO_DISPATCH    PtrFastloDispatch = NULL;

    // Initialize dispatch function table here. See Part 3
    // and accompanying disk for details.

    // Now, it is time to initialize the fast I/O stuff . . .
    // Note that I am initializing the "FastloDispatch" field in
    // the DriverObject below.
    PtrFastloDispatch = DriverObject->FastIoDispatch =
       &(SFsdGlobalData.SFsdFastloDispatch);
```

```
    II Initialize the global fast I/O structure
    // NOTE: The fast I/O structure has undergone a substantial
    // revision in Windows NT Version 4.0. The structure has been
    // extensively expanded.
    // Therefore, if your driver needs to work on both V3.51 and V4.0+,
    // you will have to be able to distinguish between the two versions
    //at compile time.
    PtrFastIoDispatch->SizeOfFastIoDispatch = sizeof(FAST_IO_DISPATCH) ;
    PtrFastIoDispatch->FastIoCheckIfPossible =
                                        SFsdFastloChecklfPossible;
    PtrFastIoDispatch->FastIoRead        = SFsdFastloRead;
    PtrFastIoDispatch->FastIoWrite       = SFsdFastloWrite;

    // See Part 3 for other initialization steps performed here.
}
```

In this example, the test driver only supports read and write operations using the fast I/O method. Therefore, other fields in the FAST_IO_DISPATCH data structure are initialized to NULL. An explanation for the SFsdFastloChecklfPossible!) routine, as well as other information on the implementation of the fast I/O routines is provided in Part 3.

File open

In Windows NT, to access data for a file stream, the file stream must first be opened. The open* operation performed by an application returns a handle to the application. This handle is used by the application when reading or writing to the file stream and corresponds to a file object structure, created by the I/O Manager, representing an instance of a successful open operation.

From the perspective of the file system driver servicing the open request, a considerable amount of work is performed at file open time to support access to the file stream. The file system constructs all in-memory data structures required to support I/O operations to the file stream, including the construction of any data structures that might be required to support buffered access to file data. The file system driver must also fill in specific fields in the file object structure; these fields were described in the previous chapter.

The file system driver allocates and initializes a file control block (FCB) structure, which is a unique representation of the file stream in memory. This is done only if no such structure currently exists; as it would if the file stream had been previously opened and at least one reference to the FCB were still present. If a new

* Open operations requesting access to previously created file streams and close operations that create new file streams result in the same IRP being dispatched to a file system by the I/O Manager, with a major function of *IRP_MJ_CREATE.* Effectively, a create operation is simply a two-step process, where an entry representing the new file is first created, and subsequently opened. We will refer to both create and open operations together as requests to open a file stream.

file control block is created, most file system drivers also allocate memory for a structure of type FSRTL_COMMON_FCB_HEADER (see the previous chapter for an explanation of the various fields in the CommonFCBHeader structure). Often, this structure, which is required by the Cache Manager to be able to cache file data, is embedded by file system drivers within the file control block representing the file stream.

Note that even if the current open operation specifies noncached access to file data, the file system driver will still end up allocating the FSRTL_COMMON_FCB_HEADER along with the FCB for the file, since subsequent concurrent open operations might require cached file access. Initialization of the individual fields within the structure is performed by the file system at this time as follows:

- The FSD initializes the two ERESOURCE type objects, allocated as part of the CommonFCBHeader from the nonpaged memory pool, with the ExInitializeResourceLite() system call.

  The two resource object fields are the MainResource and the PagingIoResource.

- The enumerated type field IsFastIoPossible is initialized to an appropriate value.

  Typically, FSDs set this to FastIoIsPossible. By doing so, the I/O Manager is encouraged to begin using the fast I/O method for accessing data for the file stream at the very earliest—typically, as soon as caching is initiated for the file.*

- Each of the file size fields—the AllocationSize, ValidDataLength, and FileSize—is initialized to their true values.

  If the file stream has been created as a result of the create operation, then the file size fields will all be initialized to 0. Otherwise, for an existing file stream or if the create operation requested preallocation of space for the file, the file size fields will be initialized to the correct values.

Once the CommonFCBHeader is allocated and initialized, the FsContext field in the file object is initialized to refer to the allocated CommonFCBHeader structure.

The PrivateCacheMap field in the file object structure is initialized by the file system driver to NULL.

Finally, the FSD must also allocate and initialize a structure of type SECTION_OBJECT_POINTERS. A single (unique) instance of the structure is typically associ-

* Caching for the file stream is initiated when the FSD receives the first I/O request for the file stream. Therefore, the first I/O request for the file stream will always be described via an IRP by the I/O Manager.

ated with the FCB. Each of the fields within the structure is initialized to NULL. The SectionObjectPointer field in the file object structure is then initialized to refer to the allocated structure.

The following code extract from a file system driver implementation of a file open operation performs the operations described here (in the code extract, it's assumed that the IRP_NOCACHE flag has not been specified):

```
// There are some fields that must always be associated with an FCB
//to successfully interface with the Cache Manager. The sample FSD
// implementation has extracted these fields into a separate structure.
typedef struct _SFsdNTRequiredFCB {
    FSRTL_COMMON_FCB_HEADER         CommonFCBHeader;
    SECTION_OBJECT_POINTERS        SectionObj ect;
    ERESOURCE                       MainResource;
    ERESOURCE                       PagingloResource;
} SFsdNTRequiredFCB, *PtrSFsdNTRequiredFCB;

// The actual FCB structure is defined by the sample FSD as shown below:
typedef struct _SFsdFileControlBlock {
    SFsdldentifier                          Nodeldentifier;
    // We will go ahead and embed the "NT Required FCB" right here.
    //    Note that it is just as acceptable to simply allocate
    //    memory separately for the other half of the FCB and store a
    //    pointer to the "NT Required" portion here, instead of embedding
    //     it.
    SFsdNTRequiredFCB                    NTRequiredFCB;
    // Other fields go here. See subsequent chapters for details.
    // ...
    // Some state information for the FCB is maintained using the
    // Flags field
    uint32                                      FCBFlags;
    // More fields here ...
} SFsdFCB, *PtrSFsdFCB;

// Some Flag definitions, see accompanying diskette for definitions
// of other flag values.
#define        SFSD_INITIALIZED_MAIN_RESOURCE             (0x00002000)
#define          SFSD_INITIALIZED_PAGING_IO_RESOURCE        (0x00004000)

// Our work is performed while servicing a create/open request.
// The parameters to the SFsdCommonCreate() function will be explained
//in Part 3.
NTSTATOS SFsdCommonCreate(
PtrSFsdlrpContext            PtrlrpContext,
PIRP                         Ptrlrp)
{
    PtrSFsdFCB               PtrNewFCB = NULL;
    LARGE_INTEGER           FileAllocationSize, FileEndOfFile;
    PFILE_OBJECT             PtrNewFileObject = NULL;
    PtrSFsdVCB               PtrVCB = NULL;
    // Other declarations ...
```

```
    try {

        //As you will see in Chapter 9, a lot of information is obtained
        // from the IRP sent to the FSD for a create/open request.
        // The I/O-Manager-created file object structure pointer is also
        // obtained from the current I/O Stack Location in the FCB.
        PtrIoStackLocation = IoGetCurrentIrpStackLocation(PtrIrp);
        PtrNewFileObject   = PtrIoStackLocation->FileObject;

        // The Volume Control Block (VCB) pointer is obtained
        // from the target device object representing the mounted logical
        // volume.

        // ...

        // The create/open operation is fairly complex and is detailed in
        // Part 3. The FSD has to validate all arguments passed-in within
        // the IRP, and then traverse the path supplied within the IRP,
        // eventually leading to the file/directory/link that has to be
        // created/opened.

        // Assume that all the complicated processing has been done and
        // that we have decided to create a new FCB structure.
        // Note that a typical FSD gets the current file stream allocation-
        // size and EOF values from the directory entry for the file
        // stream (obtained from secondary storage) .
        // In this example, we assume that this is the first instance of
        // an "open" operation for a specific file stream. Therefore, we
        // allocate the FCB structure for this file stream.
        RC = SFsdCreateNewFCB(&PtrNewFCB, &FileAllocationSize,
                                          &FileEndOfFile,
                                          PtrNewFileObject,  PtrVCB);

        if ( !NT_SUCCESS(RC) ) {
            try_return(RC) ;
        }

    } finally {
        // All of the cleanup code is executed here.
        ...
    }

    return(RC);
}    // SFsdCommonCreate ()

NTSTATUS SFsdCreateNewFCB(
PtrSFsdFCB                  *ReturnedFCB,
PLARGE_INTEGER              AllocationSize,
PLARGE_INTEGER              EndOfFile,
PFILE_OBJECT                PtrFileObject,
PtrSFsdVCB                  PtrVCB)
{
    NTSTATUS                        RC = STATUS_SUCCESS;
   PtrSFsdFCB                       PtrFCB = NULL;
```

```
PtrSFsdNTRequiredFCB            PtrReqdFCB = NULL;
PFSRTL_COMMON_FCB_HEADER        PtrCoiranonFCBHeader = NULL;

 try {
     // Obtain a new FCB structure.
     // The function SFsdAllocateFCB ( ) will obtain a new structure
     // either from a zone or from memory requested directly from the
     // VMM.  Note that the sample FSD (described in greater detail in
     // Part 3 of this book) allocates the entire FCB from nonpaged pool
     // though you may choose to be "smarter" about your allocation
     // method and possibly break up the FCB into paged and nonpaged
     // portions.
     PtrFCB = SFsdAllocateFCB () ;
     if (! PtrFCB) {
         // Assume lack of memory.
         try_return(RC = STATUS_INSUFFICIENT_RESOURCES) ;
     }

     // Initialize fields required to interface with the NT Cache
     // Manager.  Note that the returned structure has already been
     // zeroed. This means that the SectionObject structure has been
     // zeroed, which is a requirement for newly created FCB structures.
     PtrReqdFCB = & (PtrFCB->NTRequiredFCB) ;

     // Initialize the MainResource and PagingloResource structures now.
     ExInitializeResourceLite(&(PtrReqdFCB->MainResource));
     SFsdSetFlag(PtrFCB->FCBFlags, SFSD_INITIALIZED_MAIN_RESOURCE);

     ExInitializeResourceLite(&(PtrReqdFCB->PagingIoResource));
     SFsdSetFlag(PtrFCB->FCBFlags, SFSD_INITIALIZED_PAGING_IO_RESOURCE);

     // Start initializing the fields contained in the CommonFCBHeader.
     PtrCommonFCBHeader = &(PtrReqdFCB->CommonFCBHeader);

     // Allow fast I/O for now.
     PtrCoiranonFCBHeader->IsFastIoPossible = FastloIsPossible;

     // Initialize the MainResource and PagingloResource pointers in
     // the CommonFCBHeader structure to point to the ERESOURCE
     // structures we have allocated and already initialized above.
     PtrCommonFCBHeader->Resource = &(PtrRegdFCB->MainResource);
     PtrCommonFCBHeader->PagingIoResource =
                                     &(PtrReqdFCB->PagingIoResource)    ;

     // Ignore the Flags field in the CommonFCBHeader for now.  Part 3
     // of the book describes it in greater detail.

     // Initialize the file size values here.
     PtrCommonFCBHeader->AllocationSize = *(AllocationSize);
     PtrCommonFCBHeader->FileSize = *(EndOfFile);

     // The following will disable ValidDataLength support. However,
     // your FSD may choose to support this concept.
     PtrCommonFCBHeader->ValidDataLength.LowPart  = 0xFFFFFFFF;
```

```
        PtrCommonFCBHeader->ValidDataLength.HighPart  =  0x7FFFFFFF;

        // Initialize other fields for the FCB here.
        PtrFCB->PtrVCB = PtrVCB;
        InitializeListHead(&(PtrFCB->NextCCB));

        // Other similar initialization continues ...

        // Initialize fields contained in the file object now.
        PtrFileObject->PrivateCacheMap = NULL;
        // Note that we could have just as well taken the value of
        // PtrReqdFCB directly below. The bottom line, however, is that
        // the FsContext field must point to a FSRTL_COMMON_FCB_HEADER
        // structure.
        PtrFileObject->FsContext =  (void *)(PtrCommonFCBHeader);

        // Other initialization continues here ...

        try_exit:    NOTHING;
    } finally {
        ;
    }

    return (RC) ;
}
```

Initiation of Caching

All file stream operations in NT require that the file stream first be opened. To avoid incurring unnecessary overhead, file system drivers do not initiate caching for a file stream until it can be determined that I/O (read/write of file data) will be performed on the file stream. Therefore, caching is typically initiated only when the first I/O operation (read/write) is received by the file system driver. Note that caching must be initiated for each file object on which I/O can be performed (only if buffered access is allowed by the user). To determine whether caching had been previously initiated for a specific file object, the PrivateCacheMap field in the file object is checked as follows:

```
#define SFsdHasCachingBeenInitiated(PFileObject)     \
        ((PFileObject)->PrivateCacheMap ? TRUE : FALSE)
```

To initiate caching, the FSD uses the CcInitializeCacheMap() interface routine. This routine is defined as follows:

```
void CcInitializeCacheMap  (
    IN PFILE_OBJECT                     PtrFileObject;
    IN PCC_FILE_SIZES                   FileSizes;
    IN BOOLEAN                          PinAccess;
    IN PCACHE_MANAGER_CALLBACKS         CallBacks;
    IN PVOID                            LazyWriterContext
);
```

where:

```
typedef struct _CC_FILE_SIZES {
    LARGE_INTEGER            AllocationSize;
    LARGE_INTEGER            FileSize;
    LARGE_INTEGER            ValidDataLength;
} CC_FILE_SIZES, *PCC_FILE_SIZES;

// The callbacks structure is defined as follows:
typedef struct _CACHE_MANAGER_CALLBACKS {
    PACQUIRE_FOR_LAZY_WRITE    AcquireForLazyWrite ;
    PRELEASE_FROM_LAZY_WRITE    ReleaseFromLazyWrite;
    PACQUIRE_FOR_READ_AHEAD     AcquireForReadAhead;
    PRELEASE_FROM_READ_AHEAD    ReleaseFromReadAhead;
} CACHE_MANAGER_CALLBACKS, *PCACHE_MANAGER_CALLBACKS;
```

Resource Acquisition Constraints:

The above routine requires that the FCB for the file be acquired either shared or exclusive prior to invoking the routine.

Parameters:

`PtrFileObject`

This is the file object for which caching is being initiated.

FileSizes

The Cache Manager requires that the current file sizes be supplied at this time. Note that since the FCB for the file is acquired either shared or exclusively, none of the file size values can change while caching is being initiated for any file object associated with the file stream.*

PinAccess

The caller can specify if the pinning interface will be used to access data. Note that the pinning interface cannot be used concurrently with either the copy interface or the MDL interface to access data for the file stream. Typically, for user file open requests, you should set this to FALSE.

Callbacks

In the Windows NT environment, the file system, Virtual Memory Manager, and the Cache Manager are all highly dependent on each other. I/O operations can be initiated from the file system driver (on behalf of user processes), via the Virtual Memory Manager or from the Cache Manager. To avoid system

* It is highly recommended (in order to avoid data corruption) that the FCB for a file be acquired exclusively whenever there are any modifications resulting in changes to the data or attributes of the file stream. Therefore, if the FCB for the file stream has been acquired shared or exclusively while caching is being initiated, we can be certain that the file sizes will not change from underneath us.

deadlock, a well-defined hierarchy must set the order in which each of these components can acquire their respective resources associated with the file stream(s) on which I/O is being performed. This order is defined as follows:

— File system resources are acquired first.

— Cache Manager resources are acquired next.

— Virtual Memory Manager resources are acquired last.

To help maintain this hierarchy, the file system driver is required to supply the Cache Manager with callback routines that are utilized by the read-ahead and delayed-write threads in the Cache Manager. These callback routines are supplied when caching is initiated by the FSD using this argument. Further details on this topic will be presented in Part 3.

LazyWriterContext

This value is treated as an opaque pointer value by the Cache Manager. It is used as an argument supplied to the file system driver when the Cache Manager uses the AcquireForLazyWrite ( ) and AcquireForRead-Ahead ( ) callback routines. (The name of the argument is somewhat of a misnomer since the same context is used for both the read-ahead and write-behind callbacks; therefore it is not just the lazy writer context, but the read-ahead context as well.) Typically, the FSD will supply a pointer to a Context Control Block (CCB)* as the context.

Functionality Provided:

The CcInitializeCacheMap ( ) routine is responsible for creating all data structures required for the Cache Manager to support caching for the concerned file stream. The first invocation of this routine results in the creation of the shared cache map structure for the file stream. It is extremely important to note that the Cache Manager also references the file object structure at this time to ensure that the file object stays around and that a corresponding uninitialize operation will occur sometime in the future. The Cache Manager also creates a file mapping (section) object for the file using the services of the Virtual Memory Manager. For all subsequent invocations of this routine for the same file stream (with different file object structures), the Cache Manager checks the current size of the mapping section object and extends it if required.

The Cache Manager also allocates a private cache map structure and initializes it. A pointer to the allocated PrivateCacheMap is stored in the Private-CacheMap field within the passed-in file object structure. Since the value of the

* The Context Control Block is a structure created by file system drivers to represent an open instance of a file stream. There is one CCB corresponding to each successful open operation; therefore there is a one-to-one mapping between file object structures created by the I/O Manager (representing a successful open operation on a file stream) and CCBs created by the file system driver. CCB structures arc discussed in detail in Chapter 9, *Writing a File System. Driver I.*

PrivateCacheMap field now becomes nonnull, subsequent I/O requests will check this field using the SFsdHasCachingBeenInitiated() macro, defined above, and determine that caching had been previously initiated for the file stream via the specific file object.

Note that the Cache Manager does not map in any views for the file stream at this time. These views into the file are created only when data transfer is requested using any of the three interfaces provided by the Cache Manager.

Since the initialization routine does not return any status back to the caller, the Cache Manager raises exceptions if something goes wrong while trying to perform the initialization for the file object. Exception handlers in the FSD should be capable of receiving such exceptions and returning an appropriate error back to the application that initiated the cached I/O operation.

WARNING Using Structured Exception Handling is no longer optional if you write a file system driver that interacts with the Windows NT Cache Manager. I would advise that all kernel-mode drivers should incorporate SEH to ensure that system integrity and robustness is not compromised.

Example of Usage:

Caching is initiated by file system drivers when either a read or a write operation is invoked for a file stream. In this code snippet, caching is initiated when a read request is processed for a file stream.

```
//A pointer to a callbacks structure must be passed in to the Cache
// Manager when initializing caching for a file stream. Typically, file
// systems use a single global callbacks structure that has been
// initialized.
typedef struct _SFsdData {
    SFsdIdentifier                Nodeldentifier;
    // Some fields that will be discussed further in Part 3.
    ...
    // The NT Cache Manager uses the following callbacks to ensure
    // correct locking hierarchy is maintained.
    CACHE_MANAGER_CALLBACKS    CacheMgrCallBacks;

    // Some more fields that will also be discussed in Part 3.
    ...
} SFsdData, *PtrSFsdData;

// The arguments to the SFsdCommonRead() function (part of the sample FSD
// provided in this book) will be discussed in Part 3.
NTSTATUS    SFsdCommonRead(
PtrSFsdlrpContext              PtrlrpContext,
PIRP                           Ptrlrp)
```

```
{
    NTSTATUS                    RC = STATUS_SUCCESS;
    PFILE_OBJECT                PtrFileObject = NULL;
    PtrSFsdFCB                  PtrFCB = NULL;
    PtrSFsdCCB                  PtrCCB = NULL;
    PtrSFsdNTRequiredFCB        PtrReqdFCB = NULL;
    BOOLEAN                     NonBufferedlo = FALSE;
    LARGE_INTEGER               ByteOffset;
    uint32                      ReadLength = 0, TruncatedReadLength = 0;
    BOOLEAN                     CanWait = FALSE;
    void                        *PtrSystemBuffer = NULL;
    // Other declarations .. .

    try {

        //As you will see in Chapter 9, a lot of information is obtained
        // from the IRP sent to the FSD for a read request.
        // The I/O-Manager-created file object structure pointer is also
        // obtained from the current I/O Stack Location in the FCB.
        PtrloStackLocation = loGetCurrentlrpStackLocation(Ptrlrp);
        PtrFileObject      = PtrIoStackLocation->FileObject;

        // Get the FCB and CCB pointers.
        //    Typically the FsContext2 field in the file object refers to
        //    the Context Control Block associated with the file object.
        PtrCCB = (PtrSFsdCCB)(PtrFileObject->FsContext2);
        ASSERT(PtrCCB);
        PtrFCB = PtrCCB->PtrFCB;
        ASSERT(PtrFCB);

        // Other arguments are also obtained ...

        NonBufferedlo = ((PtrIrp->Flags & IRP_NOCACHE) ? TRUE : FALSE);
        ByteOffset    = PtrIoStackLocation->Parameters .Read.ByteOffset;
        ReadLength    = PtrIoStackLocation->Parameters.Read.Length;

        // Don't worry about how the following flag is set in the
        // PtrlrpContext structure at this time. Note, however, that
        // the CanWait value determines whether the caller is willing to
        // perform the operation synchronously (CanWait = TRUE), or if the
        // caller prefers asynchronous processing (CanWait = FALSE).
        CanWait = ((PtrIrpContext->IrpContextFlags &
                                SFSD_IRP_CONTEXT_CAN_BLOCK)
                                        ? TRUE : FALSE);

        PtrReqdFCB = &(PtrFCB->NTRequiredFCB);

        //A lot of preprocessing is typically performed that you will
        // read about later in this book.

        // Assume for now that this FSD does not have to worry about
        // paging I/O requests.

        // Try to acquire the FCB MainResource shared. Assume that the call
```

```
II cannot fail. Also assume that the caller does not mind blocking.
ExAcquireResourceSharedLite(&(PtrReqdFCB->MainResource),   TRUE)

// More processing here that will be discussed later in Part 3 of
// this book.

// Branch here for cached vs. noncached I/O.
if (iNonBufferedlo) {

    // The caller wishes to perform cached I/O. Initiate caching if
    // this is the first cached I/O operation using this file
    // object
    if (!SFsdHasCachingBeenInitiated(PtrFileObject)) {
        // This is the first cached I/O operation. You must ensure
        // that the Common FCB Header contains valid sizes at this
        // time
        CcInitializeCacheMap(PtrFileObject, (PCC_FILE_SIZES)
                (&(PtrReqdFCB->CommonFCBHeader.AllocationSize)    )   ,
                FALSE,    //We will not utilize pin access for
                          // this file
                &(SFsdGlobalData.CacheMgrCallBacks),  // callbacks
                PtrCCB);        // The context used in callbacks
    }

    // Check and see if this request requires an MDL returned to
    // the caller.
    if (PtrIoStackLocation->MinorFunction & IRP_MN_MDL) {
        // Caller wants an MDL returned. Note that this mode
        // implies that the caller is prepared to block.
        // CcMdlReadO is discussed later in this chapter.
        CcMdlRead(PtrFileObject, &ByteOffset, TruncatedReadLength,
                        &(PtrIrp->MdlAddress)  ,
                        &(PtrIrp->IoStatus));
        NumberBytesRead = PtrIrp->IoStatus.Information;
        RC = PtrIrp->IoStatus.Status;

        try_return(RC) ;
    }

    // This is a regular run-of-the-mill cached I/O request. Let
    // the Cache Manager worry about it.
    // First though, we need a buffer pointer (address) that is
    // valid.  More on this in Chapter 9.
    PtrSystemBuffer = SFsdGetCallersBuffer(Ptrlrp);
    ASSERT(PtrSystemBuffer);
    if ( !CcCopyRead(PtrFileObject, &(ByteOffset), ReadLength,
                CanWait, PtrSystemBuffer, &(PtrIrp->IoStatus))) {
        // The caller was not prepared to block and data is not
        // immediately available in the system cache.
        // Mark IRP Pending and prepare to post the request for
        // asynchronous processing. I am beginning to sound like a
        // broken record but more on this in Part 3 of the book.
        try_return(RC = STATUS_PENDING);
    }
```

```
            // We have the data
            RC = PtrIrp->IoStatus.Status;
            NumberBytesRead = PtrIrp->IoStatus.Information;

            try_return(RC);

        } else {
            // Noncached processing done here.
        }

        // Other processing ...

        try_exit:    NOTHING;

    } finally {
        // A lot of processing done here before completing the IRP.
        ....
    } // end of "finally" processing

    return(RC);
}
```

Cache Manager Interfaces

Once caching has been initiated for a file stream using a file object, requests to read and write data are satisfied from the system cache. In the previous chapter, three interfaces provided by the Cache Manager to access cached data were listed. Each of the routines comprising the three interfaces is covered in this section.

TIP You may find it useful simply to skim through the material presented below on your first reading and to refer back to it when required as you progress through Part 3 (describing FSD development) and also when you eventually design and debug your kernel-mode file system (or filter) driver.

Copy Interface

The copy interface is most commonly used by FSDs to access data within the system cache. The following routines comprise the copy interface.

CcCopyRead()/CcFastCopyRead()

```
BOOLEAN
CcCopyRead (
    IN PFILE_OBJECT             FileObject,
    IN PLARGE_INTEGER           FileOffset,
    IN ULONG                    Length,
```

```
    IN BOOLEAN                Wait,
    OUT PVOID                 Buffer,
    OUT PIO_STATUS_BLOCK      loStatus
);

VOID
CcFastCopyRead (
    IN PFILE_OBJECT           FileObject,
    IN ULONG                  FileOffset,
    IN ULONG                  Length,
    IN ULONG                  PageCount,
    OUT PVOID                 Buffer,
    OUT PIO_STATUS_BLOCK      loStatus
);
```

Resource Acquisition Constraints:

The FCB for the file must usually be acquired shared before invoking the routine. Acquiring the FCB exclusively will prevent multiple readers from being able to concurrently access file data and should therefore be avoided in the interest of efficiency.

Parameters:

`FileObject`

This is a pointer to the file object structure representing the open operation performed by the thread. Caching must have been previously initiated by the file system driver on this file object.

Note that if the file system driver has not initiated caching prior to invoking the CcCopyReadO or CcFastCopyRead() routines, an exception will be generated since the Cache Manager assumes that the private cache map and shared cache map structures exist and have been initialized correctly.

`FileOffset`

This is the starting offset in the file, from where the read operation should be performed.

For the CcCopyRead() routine, the starting offset can be anywhere within the allowable range of file offsets—a 64-bit quantity. However, the CcFast-CopyRead() expects the entire range being requested (starting offset + number of bytes) to be contained within 4GB (maximum range allowable for a 32 bit offset).

Length

This field specifies the number of bytes requested in the read operation.

Wait

This argument is only accepted by the CcCopyRead() routine. If the entire byte range requested is not present in the system cache (therefore, the data

would have to read off media using the page fault mechanism), and if Wait is specified as FALSE, the CcCopyRead ( ) routine returns a FALSE value to the caller. The caller can subsequently determine whether to restart the copy operation or to pursue some other course of action.

The CcFastCopyRead() routine assumes that the caller is prepared to wait for the data; i.e., the implied value of Wait is TRUE.

PageCount

This is the number of pages requested in the read operation. Argument is required only by the CcFastCopyRead() routine. The caller can use the COMPUTE_PAGES_SPANNED ( ) macro supplied in the *ntddk.h* header file to determine the value to be passed in.

NOTE It is surprising that the Cache Manager requires this argument, since computing the value could just as easily be done within the routine by the Cache Manager itself.

Buffer

This field contains a pointer to the buffer into which the copy operation should be performed is passed-in. If the buffer pointer becomes invalid (once the Cache Manager is invoked), an exception will be raised by the Cache Manager.

loStatus

The Status field is generally set to STATUS_SUCCESS by the Cache Manager. The Information field contains the number of bytes that were actually transferred.

Functionality Provided:

Fundamentally, both the CcCopyRead ( ) and the CcFastCopyRead() routines perform the same functionality: data is transferred from the system cache to the buffer passed in to either of the two routines. The Cache Manager also schedules *read-ahead* based upon the pattern of accesses detected during multiple invocations of either of these two routines.

The primary difference between the two routines is that the CcFastCopy-Read ( ) routine assumes that the caller is always prepared to block, waiting for the data to be brought into the system cache if it is not already there. In the case of the CcCopyRead ( ) routine, the caller is allowed to specify whether waiting for the data to be brought into the system cache is acceptable or not. If Wait is set to FALSE and file data is not already physically present in memory, the Cache Manager will simply return a status of FALSE to the caller. However, if data is

already physically present in memory, or if Wait is supplied as TRUE, the Cache Manager will return as many bytes as it successfully reads, which can be less than or equal to the number of bytes requested (if the read extends beyond the end-of-file).

A second constraint for the CcFastCopyRead() routine is that it expects to work with byte ranges that are completely contained within a 32-bit quantity. Therefore, the CcFastCopyRead() routine will not accept a byte range with a starting offset greater than or equal to 4GB or an ending offset (= starting offset + length) greater than or equal to 4GB.

For both routines, the Cache Manager expects the file system to have checked that the byte range being requested does not extend beyond the end-of-file mark (based on the size of the file stream). Therefore, the only likely reason for the number of bytes in the Information field to be less than the number of bytes requested is if data was not present in the system cache and there was an error encountered faulting in the data from disk (or from across the network).

If the buffer pointer passed in to either routine is invalid, an exception is raised by the Cache Manager.

The implementation of both routines is conceptually very simple:

- The Cache Manager determines if a mapped view for the desired range exists. If no such view exists, the Cache Manager will create such a view.
- The Cache Manager checks to see if the requested data is already physically present in memory (using the services provided by the Virtual Memory Manager). If data is not already in memory and if Wait is supplied as FALSE (for the CcCopyRead() routine), the Cache Manager immediately returns to the caller with a return status of FALSE, indicating that data transfer was not performed. In the case of the CcFastCopyRead() routine, the Cache Manager expects that the caller is prepared to block, waiting for data to be brought into the system cache. If data is not already present, the Cache Manager also determines the number of pages that should be brought in using a single I/O operation, based upon the number of bytes requested. This information is then conveyed by the Cache Manager to the Virtual Memory Manager, which is responsible for handling the page fault (when it occurs) and actually obtaining data via the page fault path from the file system driver.
- The Cache Manager performs a simple copy operation from the system cache (using the mapped view of the file) to the buffer sent in by the caller. If data is already available in the system cache, the copy operation will immediately complete. Otherwise, a page fault will occur, and the page fault handler in the Virtual Memory Manager obtains the data from disk or from across the network. Note that this results in a recursive operation into the file system driver.

The Cache Manager returns the total number of bytes successfully transferred in the Information field in the loStatus parameter.

CcCopyWrite()/CcFastCopyWrite()

```
BOOLEAN
CcCopyWrite (
    IN PFILE_OBJECT          FileObject,
    IN PLARGE_INTEGER        FileOffset,
    IN ULONG                 Length,
    IN BOOLEAN               Wait,
    IN PVOID                 Buffer
);

VOID
CcFastCopyWrite (
    IN PFILE_OBJECT          FileObject,
    IN ULONG                 FileOffset,
    IN ULONG                 Length,
    IN PVOID                 Buffer
);
```

Resource Acquisition Constraints:

The FCB for the file must be acquired exclusively before invoking either of the two routines. This allows only a single thread to be able to access the file stream and modify it.

Parameters:

FileObject

This argument contains a pointer to the file object structure representing the open operation performed by the thread. Caching must have been previously initiated by the file system driver on this file object.

Note that if the file system driver has not initiated caching prior to invoking the CcCopyWrite ( ) / CcFastCopyWrite ( ) routines, an exception will be generated, since the Cache Manager assumes that the private cache map and shared cache map structures exist and have been initialized correctly.

FileOffset

This is the starting offset in the file, from where the modification operation should be performed.

For the CcCopyWrite ( ) routine, the starting offset can be anywhere within the allowable range for file offsets, which is a 64-bit quantity. However, the CcFast-CopyWrite ( ) routine expects the entire range being modified (starting offset + number of bytes) to be contained within 4GB (maximum range allowable for a 32-bit offset).

Length

This is the number of bytes to be modified.

Wait

This argument is only accepted by the CcCopyWrite () routine. The caller can decide whether blocking for disk I/O is acceptable or not. For example, in order to modify a byte range in memory, free pages are required. To free up physical memory, some data may have to be transferred to disk. This involves disk (or network) I/O, which is a blocking operation. Similarly, if a page is being partially modified, the previous contents of the page must already be present in memory. If not, then the data has to be read off secondary storage. This, too, is a blocking operation.

If Wait is specified as FALSE to the CcCopyWrite () routine and blocking becomes necessary, the routine returns FALSE to the caller.*

The CcFastCopyWrite ( ) routine assumes that the caller is prepared to block to achieve the data transfer; i.e., the (implied) value of Wait is TRUE.

If the file stream was opened for write-through operations, data will have been flushed to secondary storage media before this call returns. By definition, this call therefore will block and hence the Wait argument must be TRUE in this case. Otherwise, a return value of FALSE will result and no data transfer will occur.

Buffer

This argument contains a pointer to the buffer from which the copy operation should be performed. If the buffer pointer becomes invalid (once the Cache Manager is invoked), an exception will be raised by the Cache Manager.

Functionality Provided:

The CcCopyWrite ( ) and the CcFastCopyWrite ( ) functions are similar to their read counterparts. They are responsible for transferring modified data from the user's buffer into the system cache.

As mentioned above in the case of the routines providing read functionality, the primary difference between the CcCopyWrite ( ) and CcFastCopyWrite ( ) routines is that the latter routine assumes the caller is always prepared to block in the context of the requesting thread. The requesting thread may have to block due to any one of the following reasons:

---

* If FALSE is returned, the caller should assume that none of the data has been transferred.

- In the case of partial write requests,* data may first have to be obtained from disk (or from across the network) before it can be modified.
- The file stream may have been opened with write-through mode specified (the FO_WRITE_THROUGH flag was set in the file object structure). In this case, modified data will be physically written out to disk (or across the network) before either of these two routines return control back to the caller. Note that writing to disk is a blocking operation, since it involves a recursive call back into the file system driver (which will then forward the request to the disk drivers/network drivers responsible for the actual transfer of data).
- There may not be a sufficient number of available, unmodified pages of physical memory to contain the new data before it can be lazy-written to disk. To create space in memory for the data, the Virtual Memory Manager has to flush out other previously modified data to disk, discard the data, and reallocate the physical pages to contain the newly modified bytes.

If CcCopyWrite ( ) is used, the caller can specify whether blocking is acceptable. If the caller is not prepared to block and data transfer cannot be immediately completed, the routine returns a FALSE status.

The CcFastCopyWrite ( ) routine expects that the starting and ending offsets for the entire request are contained within a 32-bit quantity. It also assumes that the caller is prepared to block until the write operation can be successfully completed.

Just as in the case of CcCopyRead ( ) described previously, an invalid buffer being passed in to the Cache Manager results in an exception condition being raised. Similarly, any errors encountered in either obtaining original data from secondary store (in the case of a partial write operation) or in writing the new data out (if write-through mode had been specified) will cause an exception to be raised. The exception values include the following:

`STATUS_INVALID_USER_BUFFER`
: This exception is raised if the user buffer is invalid or becomes invalid while the request is being processed.

---

* A *partial write* (as used in this context) is a write operation that does not begin and end on whole page boundaries. Note that the smallest unit of physical memory manipulated by the VMM is a page. The contents of a page are marked as either valid or not valid. It is too expensive for the VMM to keep track of valid ranges within a page. If an entire page is being overwritten (in a write request), the VMM optimizes by not obtaining the original byte range from secondary store—if the old data was not already present in memory. Instead, the VMM simply materializes an empty (zeroed) page into which the new data can be transferred and subsequently, the new contents of the page are marked as valid. If, however, an entire page is not being modified, the VMM must ensure that the original contents of the page have been brought into memory before the modification of a subset of the appropriate byte range is allowed to proceed. Transferring the affected byte range into memory from secondary storage (if it is not already present) is an expensive operation.

STATUS_UNEXPECTED_IO_ERROR or STATUS_IN_PAGE_ERROR

One of these two exceptions is raised if the Cache Manager received an error from the VMM when requesting data transfer. Note that the data transfer requested by the Cache Manager could be a read operation (in the event that the write request is a partial write), or it could be the attempt to write out the contents of the caller-supplied buffer.

STATUS_INSUFFICIENT_RESOURCES

This exception is raised if the Cache Manager could not allocate required memory to complete the request.

The implementation of both routines is similar to that for the read case described earlier:

- The Cache Manager determines if a mapped view for the desired range exists. If no such view exists, the Cache Manager creates such a view.
- The write request may either be contained completely within a page or span multiple pages. For those pages whose contents are being completely overwritten, the Cache Manager recognizes that obtaining the original contents from disk (for the byte range associated with the pages) is not required. Therefore, the Cache Manager requests zeroed pages from the VMM (using a special call provided by the VMM) and transfers the new data there. For those pages that are not being completely overwritten, the Cache Manager will perform a simple copy operation from the user's buffer into the virtual address space associated with the mapped view.

  Note that as a result of the copy operation, a page fault may occur if the byte range being modified is not already present in physical memory. The Virtual Memory Manager will resolve the page fault by bringing the original contents (for the byte range) from disk and restarting the copy operation. The copy operation should then complete successfully.

CcCanIWrite()

This routine is defined as follows:

```
BOOLEAN
CcCanlWrite (
    IN PFILE_OBJECT        FileObject,
    IN ULONG               BytesToWrite,
    IN BOOLEAN             Wait,
    IN BOOLEAN             Retrying
);
```

Resource Acquisition Constraints:

If Wait is TRUE, the file system should ensure that no resources have been acquired. Otherwise, the caller can choose to have the FCB resources unowned, or acquired shared or exclusively.

Parameters:

FileObject
: This argument contains a pointer to the file object structure representing the open operation performed by the thread.

BytesToWrite
: This is the number of bytes to be modified.

Wait
: This argument is used by the Cache Manager to determine whether the caller is prepared to wait in the routine until it is acceptable for the caller to be allowed to perform the write operation.

Retrying
: The file system may have to keep requesting permission to proceed with a write operation (if Wait is supplied as FALSE) until it is allowed to do so. This argument allows the file system to notify the Cache Manager if it had previously requested permission for the same write request or if the current instance was the first time permission was being requested for the specific write operation.

Functionality Provided:

This routine is part of a group of routines that allow the file system to defer executing a write request until it is appropriate to do so. There are a number of reasons why deferring a write operation is necessary. They include the following:

- The file system may need to restrict the number of dirty pages outstanding for each file stream at any time. This allows the file system to ensure that cached data for other file streams does not get discarded to make space for data belonging to a single file stream. Such a situation may arise if a process keeps modifying data for a specific file stream at a very fast rate.
- The Cache Manager tries to keep the total number of modified pages within a certain limit, for all files that have their data cached. This helps ensure that a sufficient number of free pages are available for other purposes, including memory for loading executable files, memory-mapped files, and memory for other system components.
- The Virtual Memory Manager sets certain limits on the maximum number of dirty pages within the system (based upon the total amount of physical mem-

ory present on the system). If the write operation causes the limit to be exceeded, the VMM would rather defer the write until the modified page writer has flushed some of the existing dirty data to disk.

In order to assist the Cache Manager and the Virtual Memory Manager in managing physical memory optimally, the file system driver can use the CcCanIWrite() routine to determine whether the current write operation should be allowed to proceed. Use of this routine is optional.

The Wait argument allows the file system to specify whether the thread can be blocked until the write can be allowed to proceed. If Wait is FALSE and the write operation should be deferred, the routine returns FALSE. The file system can then determine an appropriate course of action—this might be to postpone the operation using the CcDeferWrite() routine described next in this section.

Setting Wait to TRUE causes the Cache Manager to block the current thread (by putting it to sleep) until the write can be allowed to proceed. Note that the file system should ensure that no resources are acquired by the thread, since this may lead to a system deadlock.

The Retrying argument allows a file system to notify the Cache Manager whether permission is being requested either for the first time or in the case when permission had been previously requested (and denied) at least once before. If set to TRUE, the Cache Manager assigns a slightly higher priority to the current request while determining whether it should be allowed to proceed or not (e.g., if two write requests are pending and one of them is being retried, the Cache Manager will try to allow the one being retried to proceed first). Note, however, that there are no guarantees to ensure that a request being retried will indeed be allowed to proceed before other new requests.

Conceptually, the functionality provided by the Cache Manager in this routine is fairly simple:

- First, check whether the current write operation can proceed based upon criteria including whether the outstanding number of dirty pages associated with a file stream has been exceeded, whether the total number of dirty pages in the system cache has exceeded some limit, or whether the Virtual Memory Manager needs to block this write until enough unmodified pages are available in the system.
- If the write operation can proceed, return TRUE.
- Otherwise, if Wait is set to TRUE, put the current thread to sleep until the write operation can be allowed to proceed. Once the thread is awakened from the sleep, return TRUE. However, if Wait is FALSE, return FALSE immediately.

Note that a value of TRUE, if returned by this function, does not guarantee that conditions will continue to remain amenable to performing the write operation. Therefore, it is quite possible that CcCanlWrite () returns TRUE but by the time the write operation is actually submitted, conditions have changed (other writes may have caused many more pages to become dirty) such that the current write should really be deferred. However, since correctness of the operation is not affected, the caller should not really worry about this possible race condition.

To ensure that no other thread sneaks in to perform a write and thereby increase the number of outstanding modified pages, your FSD can acquire the FCB for the file stream exclusively before invoking CcCanlWrite (). However, Wait should then be set to FALSE.

CcDeferWrite()

```
VOID
CcDeferWrite (
    IN PFILE_OBJECT                         FileObject,
    IN PCC_POST_DEFERRED_WRITE              PostRoutine,
    IN PVOID                                Contextl,
    IN PVOID                                Context2,
    IN ULONG                                BytesToWrite,
    IN BOOLEAN                              Retrying
);
```

where:

```
typedef
VOID  (*PCC_POST_DEFERRED_WRITE)  (
    IN PVOID            Contextl,
    IN PVOID            Context2
);
```

Resource Acquisition Constraints:

No resources should be acquired before invoking this routine.

Parameters:

FileObject
: This argument contains a pointer to the file object structure representing the open operation performed by the thread.

PostRoutine
: The routine to be invoked whenever it is appropriate for the current write request to proceed. Typically, this is a recursive call into the file system write routine.

Context1 and Context2

These are arguments that the PostRoutine will accept. Typically, if the post routine is the same as the generic write routine, these arguments are the DeviceObject and the IRP (for the current request).

BytesToWrite

This is the number of bytes being modified.

Retrying

This allows the file system to specify whether the check (should the write be allowed to proceed?) is being performed for the first time or has already been performed before.

Functionality Provided:

This routine is part of a group of routines that allows the file system to defer executing a write request. As discussed earlier, the CcCanlWrite() routine allows a file system driver to query the Cache Manager to see if the current write request can proceed immediately. If the CcCanlWrite() routine returns FALSE, the file system can use the CcDeferWrite() routine to queue the write until it is appropriate for it to proceed.

The PostRoutine argument allows the file system to specify the routine that will perform the actual write operation when invoked. It is quite possible that the Cache Manager might choose to invoke the post routine immediately (in the context of the thread invoking the CcDeferWrite() routine). Typically, however, the post routine is invoked asynchronously whenever a sufficient number of dirty pages have been flushed to disk.

CcSetReadAheadGranularityO

```
VOID
CcSetReadAheadGranularity (
    IN PFILE_OBJECT         FileObject,
    IN ULONG                Granularity
);
```

Resource Acquisition Constraints:

There are no special resource acquisition constraints associated with this routine.

Parameters:

FileObject

This argument contains a pointer to the file object structure representing the open operation performed by the thread.

Granularity

This is the new granularity to be used in determining the number of additional bytes obtained by the read-ahead thread.

Functionality Provided:

The default read-ahead size is PAGE_SIZE. This simple routine allows the file system to determine an appropriate read-ahead granularity for a file stream. The new granularity should be a power of two and should be greater than or equal to the PAGE_SIZE value.

CcScheduleReadAhead()

```
VOID
CcScheduleReadAhead (
    IN PFILE_OBJECT        FileObject,
    IN PLARGE_INTEGER      FileOffset,
    IN ULONG               Length
);
```

Resource Acquisition Constraints:

There are no special resource acquisition constraints associated with this routine.

Parameters:

FileObject

This argument contains a pointer to the file object structure representing the open operation performed by the thread.

FileOffset

This is the offset from which the last read was initiated.

Length

This is the number of bytes requested in the last read operation.

Functionality Provided:

The CcScheduleReadAhead() routine is shared by both the copy interface and the MDL interface. This routine allows the file system to request that read-ahead be performed (if appropriate) for a file stream.

Using this routine is optional, since read-ahead is automatically initiated by the Cache Manager (unless the file system has requested that read-ahead be disabled for a specific file stream) whenever a read operation is performed, using either the copy interface or the MDL interface. However, this routine allows a file system to initiate read-ahead itself whenever required.

The FileOffset and Length arguments typically describe a read operation that has just been completed (in the case of an MDL read, the read operation may have just been initiated). Since it has been determined empirically by Windows

NT designers that the read-ahead implementation on Windows NT is not particularly beneficial when the original read request is fairly small (performance might actually degrade in some cases where read-ahead is inappropriately invoked), the file system typically does not invoke the read-ahead routine directly. Instead, the file system can use the following system-defined macro to initiate read-ahead if required:

```
•define CcReadAhead(FO,FOFF,LEN) {                    \
    if ((LEN) >= 256) {                                \
        CcScheduleReadAhead((FO),(FOFF),(LEN));        \
    }                                                  \
}
```

Whether read-ahead is actually performed depends on the following factors:

- If the file stream had been opened for sequential access, the Cache Manager will typically read ahead aggressively to ensure that data is always present in the cache to satisfy the (expected) next read operation.
- Even if the file stream is not open for sequential access, the Cache Manager maintains information, associated with the file stream, that allows it to determine the pattern of data access. If data is currently being accessed sequentially or if data is being accessed in a certain recognizable pattern, the Cache Manager will again attempt to read ahead enough data to satisfy the next read operations from the system cache.

The set of routines comprising the copy interface are the most commonly used by file systems when accessing cached data for file streams. Consult Part 3, as well as the accompanying diskette, for code fragments that illustrate the usage of these routines.

Pinning Interface

The pinning interface allows a client to map data into the system cache, lock the data into the system cache if required, and subsequently manipulate the data using a virtual address pointer. Data can be unpinned later when it is no longer required. The following routines comprise the pinning interface.

CcMapData()

```
BOOLEAN
CcMapData (
    IN PFILE_OBJECT        FileObject,
    IN PLARGE_INTEGER      FileOffset,
    IN ULONG               Length,
    IN BOOLEAN             Wait,
    OUT PVOID             *Bcb,
    OUT PVOID             *Buffer
);
```

Resource Acquisition Constraints:

There are no special resource acquisition constraints associated with this routine.

Parameters:

`FileObject`
This argument contains a pointer to the file object structure representing the open operation performed by the thread.

FileOffset
Data should be mapped in beginning at this offset in the file stream.

Length
This is the number of bytes that should be mapped into the system cache.

Wait
This is TRUE, if the caller wishes only that the data be mapped in (as opposed to requiring that the data be physically present in the system cache), otherwise, FALSE.

Bcb
If this routine returns a success code, a pointer to a Buffer Control Block (BCB) structure (allocated by the Cache Manager) is returned in this argument. The memory allocated for the BCB structure is released by the Cache Manager when the CcUnpinData ( ) routine is invoked for the last time. The BCB is also considered to be referenced whenever this routine is successfully invoked. A corresponding invocation of CcUnpinData ( ) will dereference the BCB.

Buffer
This contains the virtual address of the mapped data (if the routine is successful). The pointer is valid until a request to unmap or unpin the data is made.

Functionality Provided:

The CcMapData ( ) routine allows the caller to request that a range of bytes associated with the file stream be mapped into the system cache. This range of bytes will not be unmapped until a subsequent call to CcUnpinData () is made. If successful, this routine returns two values:

- A pointer to a Buffer Control Block (BCB) structure. This pointer should be used by the caller as context to be supplied to the Cache Manager on subsequent calls to manipulate the mapped buffer.
- A virtual address pointer representing the start of the mapped range.

Note that this routine simply maps in the desired byte range—no guarantees are provided that the byte range will be pinned into memory. Therefore, it is entirely

possible that subsequent attempts to access the byte range may cause page faults that will eventually result in the data being brought into memory from secondary storage.

It is important to note that the caller must not use the returned buffer pointer to modify the mapped range of bytes until a call either to CcPinMappedDataO or CcPreparePinWrite ( ) is made. Therefore, the caller can only use the returned buffer pointer to read the mapped range until the range is pinned in memory.

If Wait is TRUE, the Cache Manager will map the data into the cache and return. In this case, the data does not need to be physically present in the cache. If Wait is FALSE, the Cache Manager will return success only if the data is already physically present in the cache. The net result is that setting Wait to TRUE should result in quicker turnaround from the Cache Manager, since it must only ensure that data is mapped into the cache, as opposed to the alternative case, when the Cache Manager must ensure that data is physically present.

It is quite possible that this routine may pin the data into memory before returning a success code to the caller. However, the caller must be careful not to depend on this behavior and to explicitly invoke an appropriate routine to pin the mapped data when required.

Finally, the caller can invoke this routine multiple times for the same byte range. However, a corresponding invocation to CcUnpinData ( ) must be made for each instance that the CcMapData ( ) routine was successfully called.

CcPinMappedDataO

```
BOOLEAN
CcPinMappedData (
    IN PFILE_OBJECT          FileObject,
    IN PLARGE_INTEGER        FileOffset,
    IN ULONG                 Length,
    IN BOOLEAN               Wait,
    IN OUT PVOID            *Bcb
);
```

Resource Acquisition Constraints:

There are no special resource acquisition constraints associated with this routine.

Parameters:

`FileObject`

This argument contains a pointer to the file object structure representing the open operation performed by the thread.

`FileOffset`
Data is mapped in beginning at this offset in the file stream.

Length
This is the number of bytes that were mapped into the system cache.

Wait
This is TRUE if the caller can block, waiting for data to be brought into the system cache.

Bcb
When data was previously mapped into the system cache, a pointer to a BCB structure was returned by the Cache Manager. That pointer must now be used as context in this routine. It is quite possible that the Cache Manager might allocate a new BCB when this routine is invoked, and therefore return a new BCB pointer value to be used as context in subsequent calls for the pinned byte range.*

Functionality Provided:

Upon successful return from the CcPinMappedData ( ) routine, the caller can be assured that the previously mapped data is now pinned in the system cache. Now, the caller is also permitted to modify the pinned data. However, if modifications are performed, the caller must inform the Cache Manager by using the CcSetDirtyPinnedData ( ) routine, described later.

The CcPinMappedData ( ) routine will not do anything and simply return success if any of the previous invocations to CcMapData ( ) resulted in data being pinned in the system cache. Similarly, since it is legitimate to invoke the CcPinMappedData ( ) routine multiple times for the same file stream, this routine will simply return a success if the requested byte range has been pinned before.

This routine is used only to pin previously mapped data. As was mentioned earlier, a successful return from a call to CcMapData ( ) requires that a subsequent call to CcUnpinData ( ) be made. However, note that no additional calls to CcUnpinData ( ) are required if the CcPinMappedData ( ) routine is successfully invoked for previously mapped data. Therefore, the following rules should be followed in this regard:

- If you invoke CcMapData ( ) successfully for a specific byte range, you must subsequently invoke CcUnpinData ( ) .

---

* If a new BCB pointer value is returned from this call, you (the caller) should assume that the old BCB has been dereferenced and deallocated.

- If you invoke CcMapData ( ) and then you use CcPinMappedData ( ) , you will invoke CcUnpinData ( ) only once, to correspond to the CcMapData ( ) call. Specifically, you should not invoke CcUnpinData ( ) twice.
- If you invoke CcMapData ( ) more than once for the same byte range (i.e., using the same BCB pointer), you must invoke CcUnpinData ( ) for each instance when CcMapData ( ) was successfully invoked.
- Regardless of the number of times you invoke CcPinMappedData ( ) for the same byte range (i.e., using the same BCB pointer), you do not have to invoke CcUnpinData ( ) to correspond to any of these calls (since the calls are effectively turned into NULL operations).

CcPinRead()

```
BOOLEAN
CcPinRead (
    IN PFILE_OBJECT         FileObject,
    IN PLARGE_INTEGER       FileOffset,
    IN ULONG                Length,
    IN BOOLEAN              Wait,
    OUT PVOID              *Bcb,
    OUT PVOID              *Buffer
);
```

Resource Acquisition Constraints:

There are no special resource acquisition constraints associated with this routine.

Parameters:

`FileObject`

This argument contains a pointer to the file object structure representing the open operation performed by the thread. The caller should have initialized caching for the file stream using this file object.

`FileOffset`

The caller wishes to have data pinned in memory beginning at this file offset.

Length

This is the number of bytes that should be pinned in the system cache.

Wait

This is TRUE if the caller can block, waiting for data to be brought into the system cache.

Bcb

If this routine returns a success code, a pointer to a BCB structure (allocated by the Cache Manager) is returned in this argument. The BCB structure must be used as context when invoking other routines for the buffer returned

below. The memory allocated for the BCB structure is released by the Cache Manager when the CcUnpinData ( ) routine is invoked for the last time.

Buffer

This contains the virtual address of the mapped data (if the routine is successful). The pointer is valid until a request to unmap or unpin the data is made.

Functionality Provided:

A call to CcPinRead ( ) is functionally equivalent to calling CcMapData ( ) followed by a call to CcPinMappedData ( ) . The net result is that the requested byte range is pinned in the system cache. The caller is allowed to modify the byte range that is pinned, as long as the caller informs the Cache Manager that data has been modified (via the CcSetDirtyPinnedData ( ) call).

The CcPinRead ( ) routine returns TRUE if it successfully pins the requested byte range in the system cache. If successful, the routine also returns the following (just as in the case of the CcMapData ( ) routine described earlier):

- A pointer to a Buffer Control Block (BCB) structure. This pointer should be used by the caller as context to be supplied to the Cache Manager on subsequent calls to manipulate the pinned buffer.
- A virtual address pointer representing the start of the pinned range.

If the Wait argument is set to FALSE, the CcPinRead ( ) routine checks to see if the requested byte range is immediately available in the system cache. If the byte range is not present in the system cache, the routine will return an unsuccessful (FALSE) return code. However, if data is immediately available or if Wait is supplied as TRUE, this routine returns success.

This routine may be invoked multiple times for the same byte range belonging to the same file stream. However, each successful invocation of CcPinRead ( ) must be later followed by a corresponding call to CcUnpinData ( ) .

CcSetDirtyPinnedDataO

```
VOID
CcSetDirtyPinnedData (
    IN PVOID                Bcb,
    IN PLARGE_INTEGER       Lsn OPTIONAL
);
```

Resource Acquisition Constraints:

There are no special resource acquisition constraints associated with this routine.

Parameters:

Bcb

This is the BCB pointer used as context. This pointer was obtained from a previous invocation to either CcPinMappedData ( ) or CcPinRead ( ) .

Lsn

This is a Logical Sequence Number (LSN)* associated with this dirty data.

Functionality Provided:

Once data has been pinned in memory using either CcPinRead ( ) or CcPin-MappedData ( ) , the file system is free to modify the data. However, once this data is modified, the Cache Manager must be informed that the byte range contains dirty (modified) data that has yet to be written to secondary media. The file system uses CcSetDirtyPinnedData ( ) to inform the Cache Manager that the pinned data has been modified.

In the descriptions for CcPinMappedData ( ) and CcPinRead ( ) , it's mentioned that the BCB pointer returned by the Cache Manager should be used as context when invoking the Cache Manager to perform operations on the pinned byte range. The CcSetDirtyPinnedData ( ) routine also requires the BCB pointer, so that the Cache Manager can identify the byte range that has to be marked dirty.

The Cache Manager allows the file system to request that a Logical Sequence Number (LSN) be associated with the modified, pinned byte range. If your driver wishes to associate a unique number with the pinned byte range, it can pass in the optional second argument to the Cache Manager. This number can be used to determine the sequence in which data is eventually written to secondary media.

When CcSetDirtyPinnedData ( ) is invoked, the Cache Manager marks as dirty the BCB for the pinned byte range. This call also results in the lazy-writer thread being signaled if the lazy writer is not currently active. In time, the lazy-writer component will write the modified data out to secondary storage. There are two important points that must be noted here:

- No I/O is attempted in the context of the thread invoking this routine.

---

* NT provides a Log File Service (LFS) component that can be used by file systems or other modules (apparently, the LFS has yet to be extended to become generically usable by components other than kernel-mode file systems). This component provides logging and recovery services to users. Currently, NTFS is the only client of the Log File Service. The LFS provides logging and recovery services to NTFS, via the use of log files associated with file objects. Records written by the LFS to the log files are identified using Logical Sequence Numbers (LSNs). These LSNs are used in a monotonically increasing fashion, and the file system can identify the oldest record describing a transaction that has not yet been updated on secondary media using the Logical Sequence Number associated with this record. The Cache Manager provides the service where a client can associate a Logical Sequence Number with a byte range that has been pinned in memory.

- None of the data that is pinned in memory will ever be written until it is unpinned (and no other references to pin the data are outstanding). Therefore, all data that is dirty and pinned will have to wait until it is completely unpinned before it can either be explicitly flushed or lazy-written to secondary storage.

Finally, if the byte range being marked dirty extends beyond current valid data length, the Cache Manager updates the valid data length for the file stream. At some point, the Cache Manager will then inform the file system that the valid data length for the file stream has been changed.

CcPreparePin Write()

```
BOOLEAN
CcPreparePinWrite (
    IN PFILE_OBJECT        FileObject,
    IN PLARGE_INTEGER      FileOffset,
    IN ULONG               Length,
    IN BOOLEAN             Zero,
    IN BOOLEAN             Wait,
    OUT PVOID             *Bcb,
    OUT PVOID             *Buffer
);
```

Resource Acquisition Constraints:

There are no special resource acquisition constraints associated with this routine.

Parameters:

FileObject
: This argument contains a pointer to the file object structure representing the open operation performed by the thread. The caller should have initialized caching for the file stream via this file object.

FileOffset
: The caller wishes to have data pinned in memory beginning at this file offset. The caller will then begin writing the byte range, presumably beginning at this offset.

Length
: This is the number of bytes that should be pinned in the system cache.

Zero
: If TRUE, the Cache Manger will zero out the contents of the buffer before returning successfully from this routine.

Wait
: This is TRUE if the caller can block, waiting for data to be brought into the system cache.

Bcb

If this routine returns a success code, a pointer to a BCB structure (allocated by the Cache Manager) is returned in this argument. The BCB structure must be used as context when invoking other routines for the buffer returned below. The memory allocated for the BCB structure is released by the Cache Manager when the CcUnpinData ( ) routine is invoked for the last time.

Buffer

This contains the virtual address of the mapped data (if the routine is successful). The pointer is valid until a request to unmap or unpin the data is made.

Functionality Provided:

The CcPreparePinWrite () is used when the file system knows that it will modify a byte range for the file stream. Upon successful completion of this call, the file system can immediately begin transferring data into the buffer reserved for the byte range.

Functionally, this call is similar to the CcPinRead() routine; the Cache Manager maps in the desired byte range and then ensures that data is present in memory. If Wait is set to FALSE and the Cache Manager cannot return all the data requested within the byte range, the Cache Manager will return FALSE from this routine. However, if either Wait is set to TRUE or all the requested data is immediately available in the cache, the Cache Manager will pin the requested byte range in memory and return TRUE to the caller.

As a user of this routine, you should be aware of an important optimization performed by the Cache Manager: if the requested byte range contains pages that will be completely overwritten, the Cache Manager will not bother to read the original data contained in those pages from secondary media. Instead, the Cache Manager simply returns zeroed pages. Therefore, the caller of this routine must be careful not to use the CcPreparePinWrite () call in lieu of the CcPinReadO routine, since the buffer returned by the latter can indeed have data read from it. However, the buffer returned by CcPreparePinWrite () must only be used to transfer new data to secondary media.

Just as was described for the CcPinRead ( ) routine, this function returns the following:

- A pointer to a Buffer Control Block (BCB) structure. This pointer should be used by the caller as context to be supplied to the Cache Manager on subsequent calls to manipulate the pinned buffer.
- A virtual address pointer representing the start of the pinned range.

This routine may be invoked multiple times for the same byte range belonging to the same file stream. However, each successful invocation of CcPreparePinWrite ( ) must be later followed by a corresponding call to CcUnpinData ( ) .

If Zero is set to TRUE, the Cache Manager will zero out the entire buffer before returning from this routine. Finally, the buffer returned by the Cache Manager is marked as dirty (internally). Therefore, at some time, the lazy-writer thread will begin writing the contents of the buffer to secondary storage. However, as noted in the description for CcSetDirtyPinnedData ( ) , the modified byte range will be written to disk only after it has been unpinned.

CcUnpinData()/CcUnpinDataForThread()

```
VOID
CcUnpinData (
    IN PVOID      Bcb
);

VOID
CcUnpinDataForThread (
    IN PVOID                    Bcb,
    IN ERESOURCE_THREAD    ResourceThreadId
);
```

Resource Acquisition Constraints:

There are no special resource acquisition constraints associated with these routines.

Parameters:

Bcb
: BCB pointer used as context. This pointer was obtained from a previous invocation to CcMapData ( ) , CcPinRead ( ) , or CcPreparePinWrite ( ) .

ResourceThreadId
: This is only used in CcUnpinDataForThread ( ) . It identifies the thread performing the operation.

Functionality Provided:

It is extremely important that each successful invocation of CcMapData ( ) , CcPinRead(), and CcPreparePinWrite ( ) be followed by a corresponding call to CcUnpinData ( ) ; this should be done after the operation requiring that data be pinned has been completed. This routine simply unpins (unlocks) the byte range from the system cache.

The byte range is unmapped from memory only after all invocations of CcUnpinData ( ) have been made — one for each invocation of CcMapData ( ) , CcPinReadO, or CcPreparePinWrite ( ) . Data that was modified in the

system cache and has been marked dirty will be written to secondary storage by the lazy-writer thread after the BCB has been completely unmapped. Note that no I/O is performed in the context of the thread invoking the CcUnpinData() routine (all I/O will be performed asynchronously). This can be a problem when the client (file system driver) needs to ensure that all data has indeed been written to secondary storage when the BCB has been completely unmapped (unpinned). A solution to this problem is described later (see CcUnpinRepinnedBcb()).

Functionally, there is no difference between the CcUnpinData() and the CcUnpinDataForThread() routines.

CcRepinBcb()

```
VOID
CcRepinBcb (
    IN PVOID    Bcb
);
```

Resource Acquisition Constraints:

There are no special resource acquisition constraints associated with this routine.

Parameters:

Bcb

This is the BCB pointer used as context. This pointer was obtained from a previous invocation to either CcMapData(), CcPinRead(), or CcPreparePinWriteO.

Functionality Provided:

After the BCB has been completely unpinned (i.e., CcUnpinData() has been invoked for each successful invocation of CcMapData(), CcPinRead(), or CcPreparePinWrite()), the modified data will be asynchronously written to disk via the lazy-writer module. However, this presents a problem for file streams that have also been opened by users with write-through access specified (F0_WRITE_THROUGH set in the flags for the associated file object).

Since the user requires that the data be synchronously written to disk, file systems have to ensure that such write-through functionality is indeed performed before returning to the requesting user process. To ensure this, file systems use the CcRepinBcb() and the CcUnpinRepinnedBcb() routines.

The CcRepinBcb() routine simply references the BCB an additional time. This ensures that the BCB will not be deleted when a subsequent call to CcUnpinData() is made.

NOTE The BCB is deleted only after all references to the BCB are removed. Typically, a BCB is referenced when one of the CcMapRead ( ), etc. routines are invoked. The reference is only removed when CcUnpinData ( ) is subsequently called.

The significance of this operation is explained below (see CcUnpinRepinnedBcb ()).

CcUnpinRepinnedBcb()

```
VOID
CcUnpinRepinnedBcb (
    IN PVOID                    Bcb,
    IN BOOLEAN                  WriteThrough,
    OUT PIO_STATUS_BLOCK        loStatus
);
```

Resource Acquisition Constraints:

The caller must ensure that no client resources have been acquired when invoking this routine (otherwise, a system deadlock is possible).

Parameters:

Bcb
: This is the BCB pointer used as context. This pointer was obtained from a previous invocation to either CcMapDataO, CcPinReadO, or to CcPreparePinWrite().

WriteThrough
: If set to TRUE, the Cache Manager will synchronously flush modified data to secondary storage before returning from this call.

loStatus
: This is set to STATUS_SUCCESS if WriteThrough is FALSE (i.e., since there was nothing to flush synchronously, the return status must be STATUS_SUCCESS). Otherwise, it returns the actual result of the flush operation.

Functionality Provided:

In the earlier description for CcUnpinData ( ), it's mentioned that modified, pinned data will be asynchronously written to secondary storage by the lazy-writer component of the Cache Manager when the BCB is completely unpinned/unmapped. This happens after the reference count for the BCB structure is equal to 0; i.e., for every successful invocation of CcMapDataO, CcPinReadO, CcPreparePinWriteO, a corresponding invocation of CcUnpinData ( ) has been performed.

Consider the case, however, when a user process that has opened the file stream with write-through access makes a write request for the byte range that has been pinned in memory. Alternatively, the user process may request a write-through operation when the file system has pinned metadata for the file stream in memory (metadata includes file stream date, time, and size information, along with other information pertaining to the file stream). To perform write-through, the file system must ensure that the data has been written to secondary storage before control is returned to the user process.

The file system achieves this by using the CcRepinBcb{) and CcUnpinRepinnedBcb() sequence of calls to the Cache Manager. The CcRepinBcb() call adds a reference to the BCB structure, ensuring that the BCB will not be deleted when CcUnpinData() is invoked (which will be done by the file system as part of processing the user request). Subsequently, before completing the IRP describing the user's write request (typically, the file system does this by requesting that it be invoked by the I/O Manager before the IRP is completed), the file system will invoke CcUnpinRepinnedBcb(). Note that the file system must ensure that no resources have been acquired by the file system when this routine is invoked.

If WriteThrough is set to TRUE by the file system, the Cache Manager will synchronously write the modified data to secondary storage before returning from the routine. This ensures that the resource acquisition hierarchy is maintained, yet the file system can honor the user's desire for write-through operation.

Although there is a Cache Manager interface routine to flush cached data (described in the next chapter), pinned buffers are not flushed when that routine is invoked. Therefore, by using the method described here, the file system can achieve its objective of ensuring synchronous flush/write-through of user data.

CcGetFileObjectFromBcbO

```
PFILE_OBJECT
CcGetFileObjectFromBcb (
    IN PVOID    Bcb
);
```

Resource Acquisition Constraints:

There are no special resource acquisition constraints associated with this routine.

Parameters:

Bcb

This is the BCB pointer used as context. This pointer was obtained from a previous invocation to either CcMapData(), CcPinRead(), or to CcPreparePinWrite().

Functionality Provided:

The Cache Manager returns a pointer to the file object that was used when caching was first initiated for the file stream. Note that the file object is not returned referenced (i.e., the Cache Manager does not reference the file object structure an extra time when returning a pointer to the structure from this routine) and hence the Cache Manager cannot guarantee that the file object structure will not be deallocated at any instant.

MDL Interface

The Memory Descriptor List (MDL) interface is used by clients of the Cache Manager so that they can perform I/O directly into or out of the system cache. This interface can be used concurrently with the copy interface; however, neither the copy interface nor the MDL interface can be used in conjunction with the pinning interface. The following routines comprise the MDL interface.

CcMdlRead()

```
VOID
CcMdlRead (
    IN PFILE_OBJECT             FileObject,
    IN PLARGE_INTEGER           FileOffset,
    IN ULONG                    Length,
    OUT PMDL                   *MdlChain,
    OUT PIO_STATUS_BLOCK        loStatuS
);
```

Resource Acquisition Constraints:

The FCB for the file must usually be acquired shared before invoking the routine Acquiring the FCB exclusively will prevent multiple readers from being able to concurrently access file data.

Parameters:

FileObject
: This argument contains a pointer to the file object structure representing the open operation performed by the thread. Caching must have been previously initiated by the file system driver on this file object.

  Note that if the file system driver did not initiate caching prior to invoking the CcMdlRead() routine, an exception is generated, because the Cache Manager assumes that the private cache map and shared cache map structures exist and have been initialized correctly.

FileOffset
: This is the starting offset in the file. This offset denotes the file position from which the data will be transferred from the system cache. Note that the Cache

Manager does not require that the starting offset be aligned on some boundary (e.g., page boundary or sector boundary). However, the device that eventually uses the returned MDL to perform data transfer may have certain alignment restrictions that the caller should keep in mind.

Length
: This is the number of bytes that will be transferred from the system cache.

MdlChain
: If this routine does not generate an exception condition and if the status field in the returned loStatus argument is set to TRUE, then the Cache Manager will return a pointer to an allocated MDL, describing the requested byte range in this field.

loStatus
: The Cache Manager returns the status code for this operation—in the Status field—as well as the number of bytes that are described by the MDL in the Information field. Typically, if the CcMdlRead() routine does not generate an exception condition, the Status field will be set to STATUS_SUCCESS.

Functionality Provided:

The CcMdlRead() routine returns a Memory Descriptor List (MDL) that describes physical pages allocated for the passed-in byte range. This allows the client to read data directly from the system cache and write it either across the network or to some secondary storage device (that typically supports DMA).

Note that the returned pages are locked; i.e., the specified byte range is guaranteed to continue to be backed by the physical pages described in the MDL. The pages are available for reuse only after the caller invokes the CcMdlReadComplete() routine to signify that the caller no longer has any use for the MDL. Also note that the returned MDL is not necessarily mapped into the system virtual address space. If the caller does require that the pages be mapped into the system virtual address space, the caller can invoke the MmGetSystemAddressForMdl() function to do so. (Note that MmGetSystemAddressForMdl() is actually a macro defined in the *ntddk.h* header file.)

As part of creating an MDL describing the byte range (requested by the caller), the Cache Manager ensures that data is physically present in the requested pages. This is done by faulting the requested byte range into the system cache.

If this routine fails to allocate an MDL or if the data cannot be read-in, an exception will be generated by this routine. Therefore, the client must ensure that an exception handler is prepared to handle any exceptions generated as a result of invoking

this routine (a rare, though typical exception is STATUS_INSUFFICIENT_RESOURCES).

CcMdlReadComplete()

```
VOID
CcMdlReadComplete (
    IN PFILE_OBJECT         FileObject,
    IN PMDL                 MdlChain
);
```

Resource Acquisition Constraints:

There are no special resource acquisition constraints associated with this routine.

Parameters:

`FileObject`
: This argument contains a pointer to the file object structure used when CcMdlRead ( ) was invoked.

MdlChain
: This is the pointer to the MDL chain that was returned by the Cache Manager when CcMdlRead ( ) was invoked.

Functionality Provided:

Once the client has transferred data from the system cache using the MDL created by the Cache Manager (see description of CcMdlRead ( )), the client must invoke this routine to allow the Cache Manager to deallocate the MDL and unlock the physical pages associated with the byte range.

If multiple calls to CcMdlRead ( ) are made for different byte ranges for a file stream, it is not necessary that the calls to CcMdlReadComplete ( ) be made in the same order (or in any particular order) to release the various MDL chains. However, to avoid serious memory leaks, among other problems, the client must ensure that a call to CcMdlRead ( ) is always followed by a corresponding call to CcMdlReadComplete().

CcPrepareMdlWrite()

```
VOID
CcPrepareMdlWrite (
    IN PFILE_OBJECT            FileObject,
    IN PLARGE_INTEGER          FileOffset,
    IN ULONG                   Length,
    OUT PMDL                  *MdlChain,
    OUT PIO_STATUS_BLOCK       IoStatus
);
```

Resource Acquisition Constraints:

The FCB for the file must at least be acquired shared before invoking the routine. Typically, the FCB for the file stream is acquired exclusively by the file system to ensure that data consistency is maintained.

Parameters:

`FileObject`

This argument contains a pointer to the file object structure representing the open operation performed by the thread. Caching must have been previously initiated by the file system driver on this file object.

Note that if the file system driver has not initiated caching prior to invoking the CcPrepareMdlWrite() routine, an exception is generated, because the Cache Manager assumes that the private cache map and shared cache map structures exist and have been initialized correctly.

`FileOffset`

This is the starting offset in the file. This offset denotes the file position at which the data will be transferred into the system cache. Note that the Cache Manager does not require that the starting offset be aligned on some boundary (e.g., page boundary or sector boundary). However, the device that eventually uses the returned MDL to perform data transfer may have certain alignment restrictions that the caller should keep in mind.

Length

This is the number of bytes that will be transferred into the system cache.

MdlChain

If this routine does not generate an exception condition and if the status field in the returned IoStatus argument is set to TRUE, then the Cache Manager will return a pointer to an allocated MDL, describing the requested byte range in this field.

IoStatus

The Cache Manager returns the status code for this operation in the Status field, as well as the number of bytes that are described by the MDL in the Information field. Typically, if the CcPrepareMdlWrite () routine does not generate an exception condition, the Status field will be set to STATUS_ SUCCESS.

Functionality Provided:

The CcPrepareMdlWrite ( ) routine is analogous to the CcMdlReadO routine in that it returns a list of locked physical pages that can subsequently be used by the client to transfer data directly into the system cache. Typically, data is trans-

ferred directly from across the network or from a secondary storage device that supports DMA.

The pages comprising the returned MDL are guaranteed to be resident (locked in memory) until the caller invokes the CcMdlWriteComplete ( ) routine to signify that data has been transferred into the system cache. Just as in the case of CcMdlRead ( ) , the caller must not assume that the pages backing the requested byte range have been mapped into system virtual address space. However, the caller may choose to map these pages into the system virtual address space explicitly as a separate step.

Since the Cache Manager assumes that the byte range for which an MDL has been requested will be modified by the caller, the Cache Manager tries to optimize for the case when entire pages are being overwritten by returning zeroed pages instead of attempting to fault the original data into the system cache. Typically this is done only when the requested byte range extends beyond the current valid data length.*

If this routine fails to allocate an MDL or if the data cannot be read in, it will generate an exception. Therefore, the client must ensure that an exception handler is prepared to handle any exceptions generated as a result of invoking this routine (a rare exception is STATUS_INSUFFICIENT_RESOURCES; this exception would be raised if the Cache Manager could not allocate an MDL or some other similar scenario).

CcMdlWriteCompleteO

```
VOID
CcMdlWriteComplete (
    IN PFILE_OBJECT        FileObject,
    IN PLARGE_INTEGER      FileOffset,
    IN PMDL                MdlChain
);
```

Resource Acquisition Constraints:

The caller must ensure that no client resources have been acquired when invoking this routine, or a system deadlock is possible.

---

* Though the Cache Manager could further optimize for the case when entire pages are presumably being overwritten by returning zeroed pages spanning a byte range contained within current valid data length, it appears as though the Cache Manager does not do so. One possible explanation for this is the fact that if the write operation does not successfully complete, the Cache Manager would then overwrite perfectly valid data with zeroes! The conservative option, in this case, is to fault in all data that is contained within the current valid data length for the file and to return zeroed pages only for that portion of the byte range that extends beyond the valid data length.

Parameters:

`FileObject`
This argument contains a pointer to the file object structure used when CcPrepareMdlWrite ( ) was invoked.

`FileOffset`
This is a starting offset passed in to the CcPrepareMdlWrite ( ) routine.

MdlChain
This is the pointer to the MDL chain that was returned by the Cache Manager when CcPrepareMdlWrite ( ) was invoked.

Functionality Provided:

After data has been transferred into the system cache following a call to CcPrepareMdlWrite (), the client must invoke the CcMdlWriteComplete () routine to inform the Cache Manager that it is now safe to unlock the pages comprising the MDL. In turn, the Cache Manager will unlock the pages backing the requested byte range and also ensure that the modified data is written to disk.

If the file stream was opened with write-through specified, the Cache Manager will not return control from this routine until the data has been written to secondary storage. In this case, any error in writing the data out to media is returned in the form of a raised exception. However, if the file stream was not opened for write-through access, the Cache Manager simply initiates an asynchronous write operation via the lazy writer component. This data will then be written to disk at a later time.

In order to avoid system deadlock (especially in the case where write-through has been specified), it is extremely important that this routine be invoked with none of the client's resources acquired.

In the next chapter, we will continue our detailed exploration of the Cache Manager and examine issues related to termination of caching, flushing and purging of file streams, cleanup and close operations, and truncation of cached streams. We'll also review the interaction of the Cache Manager with the Virtual Memory Manager, the lazy-writer, and the read-ahead components of the Cache Manager.