Portable Systems Group

Caching Design Note

Author: *Tom Miller*

Revision 1.3, October 31, 1991

1. Overview

This design note describes the Cache Manager for Windows NT. The Cache Manager uses a file mapping model, which is closely integrated with memory management.

The file mapping model or virtual block cache, has been chosen over a logical block cache for the following reasons:

- o Virtual block caching is more compatible with the ability of user programs to map files. It is possible for some programs to do NtReadFile and NtWriteFile at the same time that other programs have the file mapped with read-only or read/write access. With proper synchronization, both types of programs are able to see the most current data.
- o By using a file mapping model, all of physical memory becomes available for data caching, with the allocation of pages reacting dynamically to the changing needs for image file pages versus data file pages.
- o Cache hits are processed more efficiently by handling virtual block hits directly in a mapped file. In most cases an I/O request is able to access the data directly in the cache, without calling the file system at all (see Section 0). The I/O system makes a subroutine call to access the cache, and the Cache Manager resolves the access via a single hardware virtual address lookup.
- o For the a recoverable file system such as NTFS, it is necessary to have caching closely synchronized with logging. This requires that all cache entries be directly identifiable by the recoverable file to which they belong.

The Cache Manager also provides a simple mechanism for dealing with unaligned buffers. If a file has been opened with caching disabled (FILE_NO_INTERMEDIATE_BUFFERING specified in the Create/Open options), then an NtReadFile or NtWriteFile *will fail* if the alignment and size of the specified transfer is less than that required by the target disk. The assumption is, that if a program specified a request with caching disabled, then it really does not want to pay the cost of having the transfer go to an intermediate buffer and be copied.

1.1 File Streams and Cache Maps

The Cache Manager is a central system component which may be thought of as being layered closely on top of the Memory Management support. Key to understanding the Cache Manager is the concept of *File Streams*.

A File Stream is a linear stream of bytes associated with a File Object. Each File System creates, deletes and manipulates File Streams both for external use via NT File System APIs, as well as for internal use by the File System itself. Examples of File Streams maintained by File Systems are the data of a given file, the EAs of a file, the Acl of a file, a directory, or any other file system metadata. How virtual byte offsets within the File Stream are mapped to physical locations in nonvolatile store is strictly an opaque operation determined by the File System, and may vary for different types of file streams.

Once a file system has identified which streams it wishes to support, it needs to decide which of these streams it wishes to cache. For all streams which are to be cached, the file system must actually support both cached and noncached access. Noncached access is always issued via a read or write I/O Request Packet (IRP), in which the **IRP_NOCACHE** flag is set in the Irp flags. (See the *NT I/O System Specification.*) For streams which may be accessed by normal user programs, such as the data of a file, the file system will also receive cached I/O requests via read or write IRPs with the **IRP_NOCACHE** flag not set. Also for internal use a file system may perform cached access to any of the streams it defines via direct calls to the Cache Manager.

As mentioned earlier, the Cache Manager uses mapping to implement the caching of streams, and to integrate caching with Memory Management's policy with other uses of pageable memory. Thus when a file system calls the Cache Manager to intitiate caching of a stream, the Cache Manager immediately maps all or a portion of the stream via a call to memory management. For larger streams, the Cache Manager may subsequently find it necessary to map additional portions of the stream on an as-needed basis. To keep track of which portions of a file stream the Cache Manager currently has mapped, it uses private data structures which it refers to as *Cache Maps*. For each stream being cached, the Cache Manager maintains a single *Shared Cache Map*. For each File Object through which the cached stream is being accessed, the Cache Manager also maintains a *Private Cache Map*. The Shared Cache Map describes an initial portion of the file stream which is mapped for common access via all File Objects for this stream. Each Private Cache Map optionally describes an additional nonoverlapping portion of the stream mapped on an as-needed basis to access bytes in the stream which were not mapped by the Shared Cache Map.

Again, the Cache Maps are private structures maintained by the Cache Manager, and a further understanding of these structures is not required by a person writing a file system. However, a file system writer does have to be aware of the respective relationships between a file system, the Cache Manager, and Memory Management. For example, when an attempted cache access results in a "miss", this miss results in a page fault which is serviced by Memory Management who subsequently makes a (recursive) call back to the file system with a noncached I/O request.

1.2 Target Clients of the Cache Manager

The Cache Manager interfaces have been primarily designed to support the following clients:

o Normal file systems such as FAT, HPFS and CDFS. File systems may create and cache File Streams for normal data files, the EAs associated with a file, the volume structure of a volume, etc. Note that the Cache Manager knows nothing about different types of streams; it only knows about File Objects and different modes of access.

  For example, HPFS creates File Streams to cache normal file data, the first time the data is actually accessed. It also creates a File Stream for a "Volume File", which is a compressed mapping of the volume structure on a HPFS volume. If the EAs or ACL for a given file fit in the Fnode, then they are simply cached with the Fnode in the Volume File. The other case HPFS has is that the EA or ACL is too large to fit in the Fnode, and is described by one or more runs of contiguous sectors external to the Fnode. In this case, a separate stream is created to cache the EA or ACL the first time they are accessed.

Interfaces are provided for File Systems to access data by copying, or accessing it directly in the cache.

- o Network File System clients, such as the Lan Manager Redirector. For starters, a Network File System looks like any other File System, with normal data streams, and potentially other types of streams associated with files. However, a Network File System client would normally not be maintaining any "volume" structure of its own.

- o Network File Servers, such as the Lan Manager Server. A file server is not expected to look like a file system at all. However, it also may be considered a "client" of the Cache Manager via the host file system(s) which it calls. Indeed, some of the file system calls which are ultimately supported by the Cache Manager (such as the Mdl interfaces defined later), were designed with Network File Servers in mind.

1.3 Cache Manager Interfaces

The Cache Manager has four sets of interfaces. One is for basic File Stream maintenance, and the other three implement different access methods for the cache. The three access methods share common support routines, but acknowledge the different ways in which the cache will be used.

Following is a brief description of the four sets of interfaces supported by the cache manager, which are described in detail in the following sections:

- o File Stream maintenance functions.

  The File Stream maintenance functions are implemented in the Cache Manager module fssup.c. These routines are for initializing and uninitializing cached operation for a stream, extending and truncating cached streams and file sizes, flushing pages to disk, purging pages from the cache without flushing, zeroing file data, and so on.

- o Copy Interface.

  The Copy Interface is implemented by the Cache Manager module copysup.c. The copy interface is the simplest form of cached access. It supports copying a range of bytes from a specified offset in a cached file stream to a buffer in memory, or from a buffer in memory to a specifiedd offset in a cached file stream. The copy interface also has a related call to initiate read ahead.

- o Mdl Interface.

The Mdl Interface is implemented by the Cache Manager module mdlsup.c. The Mdl interface supports direct access to the cache via DMA. For example, a network file server can efficiently support large client reads via DMA of the desired bytes directly out of the cache to a network device. Similarly a network file server is able to support large client writes by DMA directly into the cache. The Mdl interface shares the same Read Ahead call as the copy interface.

- A *Pinning* Interface.

  The Pinning Interface is implemented by the Cache Manager module pinsup.c. The pin interface may be used to lock (pin) data in the cache and access it directly via a pointer, and then unpin the data when the pointer is no longer required. Pinning is a database concept, and it is the optimal way for a File System to deal with the caching of file system metadata:

The following table summarizes which of the Cache Manager's clients are intended to use which of the four interface classes. Note that Network File Servers never call the Cache Manager directly, but rather benefit from the specified interfaces via associated calls to local file systems.

	Local File Systems	Network FS Clients	Network File Servers
FS Maint.	x	x	
Copy Int.	x	x	x
Mdl Int.		x	x
Pin Int.	x		

The next section walks through what a file system has to do to set up for and use the Cache Manager. Then, subsequent sections will document the individual routines belonging to the four classes of interfaces presented above.

2. WalkThrough of Cache Manager Interaction

This section attempts to present all of the background information which is important to understand when about to write a File System (including a Network File System client) or File Server which intends to use the Cache Manager. All of the following subsections but the last one relate only to file systems, but may provide some insight to someone writing a file server.

The final subsection describes how a file server accesses cached file streams. The final section should also be understood by anyone writing a local file system.

The following include files, present in \nt\private\inc, define the data structures and procedure calls described in this section and the rest of this document:

cache.h	Cache Manager structures and routines
fsrtl.h	File System Rtl structures and routines
io.h	I/O system structures and routines
ex.h	Executive structures and routines

2.1 Setting up the File Object on Create

When a file system is called at its Create Fsd entry point, one of the important fields in the Irp is a pointer to a File Object (see io.h) for the file being opened. There are three pointers in the File Object which must be initialized in a particular way for a file system which wishes to use the Cache Manager. These fields are FsContext, SectionObjectPointer, and PrivateCacheMap. (A fourth pointer, FsContext2, has no significance to the Cache Manager, and is usually used to point to a per file object context called the Channel Control Block or CCB.) The following subsections describe how these fields are to be initialized.

2.1.1 FsContext

The Cache Manager expects FsContext to point to a structure defined by FsRtl, called the FSRTL_COMMON_FCB_HEADER. This structure must be allocated from nonpaged pool, and must exist only once for the respective file stream, no matter how many times it is opened. So, for example, for normal files, exactly one Common Fcb Header must exist for each open file on the volume, no matter how many times the file is opened. If the same file is opened multiple times, then for each File Object which has the file open, FsContext must point to the same Common Fcb Header. The Common Fcb Header will generally be contained at the beginning of a common structure maintained by the file system for this file, typically called the File Control Block, or Fcb.

> \*Note that currently the Common Fcb Header is actually only used to support Fast I/O, which means it is actually only required for file objects describing user file opens. However, it might be recommedable for a new file system to always point FsContext to this structure, even for stream files.*\\

The Common Fcb Header is defined in fsrtl.h, and at the time of this revision has the following format.

```
typedef struct _FSRTL_COMMON_FCB_HEADER {

    CSHORT NodeTypeCode;
    CSHORT NodeByteSize;

    BOOLEAN IsFastIoPossible;

    LARGE_INTEGER AllocationSize;
    LARGE_INTEGER FileSize;
    LARGE_INTEGER ValidDataLength;

    PERESOURCE Resource;

} FSRTL_COMMON_FCB_HEADER;
typedef FSRTL_COMMON_FCB_HEADER *PFSRTL_COMMON_FCB_HEADER;
```

Here is a brief definition of these fields:

NodeTypeCode - A unique code identifying the Fcb for this particular file system. This field is unused by the Cache Manager.

NodeByteSize - The size of the entire containing Fcb in bytes. This field is also not used by the Cache Manager.

IsFastIoPossible - This boolean contain TRUE (0x01) whenever the file system believes it is acceptable for the I/O system to call the Cache Manager directly to read or write byte ranges directly in the cache, without calling the file system. It must contain FALSE (0x00) whenever it is not acceptable to access cache data directly. In this case, all cached reads and writes must be passed to the file system via Irp.

Examples of cases where this field might contain FALSE, would be if there are active FileLocks or Network Oplocks in the file, or the media is undergoing volume verification.

AllocationSize - The current allocation size of this file in bytes, typically an integral multiple of the underlying device sector size or allocation cluster size. This field must be initialized to the correct value when the Fcb is created, and thereafter the Cache Manager must be notified when it changes.

FileSize - The logical size of the file up to which the file may be read. Reads beginning before this point are truncated, and reads beyond this point return STATUS_END_OF_FILE. This field must be initialized to the correct value when the Fcb is created, and thereafter the Cache Manager must be notified when it changes.

ValidDataLength - The size of the initialized portion of the file. If ValidDataLength is less than FileSize, then reads extending beyond ValidDataLength return binary 0 in the read buffer. This field must be initialized to the correct value when the Fcb is created, and thereafter the Cache Manager will inform the file system when it is safe to change this value for the file on disk (see Section 0).

Resource - Pointer to an ERESOURCE structure (defined in ex.h). The ERESOURCE structure is usually allocated elsewhere in the Fcb. The executive resource structure is a synchronization structure which supports multiple Shared accessors at once, or one Exclusive accessor via the routines ExAcquireResourceShared, ExAcquireResourceExclusive and ExReleaseResource. FsRtl and the Cache Manager require that all file system operations for this stream be synchronized by this resource. Of course, modifying operations generally require that the file system take out exclusive access, and nonmodifying access require that the file system take out shared access. The synchronization requirements of streams will be further discussed later.

2.1.2 SectionObjectPointer

Memory Management and the Cache Manager require that the SectionObjectPointer field of the file object point to a structure call SECTION_OBJECT_POINTERS. This structure must also exist only once for the file stream, and it must also be allocated in nonpaged pool. Generally this structure is also allocated somewhere in the file system's Fcb.

The Section Object Pointers structure is defined in io.h, and at the time of this revision has the following format.

```
typedef struct _SECTION_OBJECT_POINTERS {
    PVOID DataSectionObject;
    PVOID SharedCacheMap;
    PVOID ImageSectionObject;
} SECTION_OBJECT_POINTERS;
typedef SECTION_OBJECT_POINTERS *PSECTION_OBJECT_POINTERS;
```

Here is a brief description of these fields, however the file system only has to initialize this structure by clearing it:

DataSectionObject - This pointer is used by Memory Management whenever a data section has been created for this stream, including when the Cache Manager has done so.

SharedCacheMap - This pointer is used by the Cache Manager to point to its SharedCacheMap structure whenever the file is currently being cached.

ImageSectionObject - This pointer is used by Memory Management whenever an image section has been created for this stream.

2.1.3 PrivateCacheMap field

This pointer must simply be initialized with NULL (0x00000000). It is filled in by the Cache Manager if the stream is cached. The only way for the file system to reliably determine if this file object is currently being cached (**CcInitializeCacheMap** has been called), is to have the Fcb Resource shared or exclusive, and test the PrivateCacheMap field for NULL. It is not valid for the file system to capture this information elsewhere, because under certain circumstances the File Object has to be forcibly uninitialized.

2.2 Initializing Cache Maps for a File Stream

The previous section described how the Cache Manager expects the File Object to be initialized on Create. The Cache Manager however does not expect Create to initiate caching, but rather that this work be deferred to the first read or write of the file. This is basically for two reasons:

First, experience has shown that it is very inefficient to immediately initiate caching on a file when it is opened, since there are quite a few apps which open a file, get or set some file information on the file or mark it for delete, and then close the file without ever accessing its data. These applications may run noticeably slower if the file system is needlessly initializing and uninitializing caching for the files.

More importantly, however, is the fact that under certain circumstances it becomes necessary to forcibly uninitialize the Cache Maps on a file object. Examples are when a file is truncated, or the file system supports removeable media and a volume fails mount verification. A network file system client may decide to call **CcUninitializeCacheMap** on all of its files in the event of an oplock break, or a virtual circuit goes down, and the data can no longer be trusted. Forced uninitialization works because we know that the next read or write, if there is one, will simply initialize the cache again.

Note that if there are multiple accesses to the same file, as represented by multiple file objects, the file system must call **CcInitializeCacheMap** for each file object that attempts to access file data. It must also eventually call **CcUninitializeCacheMap** for each of these file objects.

For further detail on initialization, see Section 0 on **CcInitializeCacheMap**.

Once, caching has been initialized on a file object, and the Fcb resource is still acquired, a file system may access the cache via one of the classes of access routines described in the next section.

2.3 Accessing Data in the Cache

Once the file object is set up and **CcInitializeCacheMap** has been called, a file system may now access data in the cache. There are three methods for accessing the cache, and these methods are described in the following subsections.

2.3.1 Copying Data To and From the Cache

The simplest way of accessing a stream is to copy data into and out of it. The routines **CcCopyRead** and **CcCopyWrite** are provided for this purpose. They both take a previously initialized file object along with a description of the desired byte range in the file and an input or output buffer in memory. It is essential that the

Fcb resource remain acquired from some time before the point where the file object was initialized (if it was not already), until after the copy operation is complete.

In the case of a call to **CcCopyRead**, the caller may also wish to call **CcReadAhead** to see if any Read Ahead is desired. The call to **CcReadAhead** takes some of the same parameters as the call to **CcCopyRead**. **CcReadAhead** automatically determines whether or not read ahead is appropriate, based on the recent history of calls to **CcReadAhead**, and whether or not the data has perhaps already been read ahead. If read ahead is required, it is scheduled to be performed by one of the Cache Manager's worker threads, so as not to hold up the current thread.

For the case of **CcCopyWrite**, all modified pages are lazy written by default. For most cases the file system simply does not have to worry about it, and the data will typically get to disk within about five seconds of when it was modified. For cases where Lazy Writing is not appropriate because the data has to be written through, see Section 0.

2.3.2 DMA Transfer of Data To and From the Cache

Network file servers and network file system clients sometimes have to transfer large amounts of data into or out of the cache from a network device. For such large transfers, it is inefficient to allocate a temporary buffer, call the copy interfaces above, transfer the data on the network device, and then free the temporary buffer. In order to eliminate the large copy and temporary buffer in the above scenario, the Cache Manager provides a second class of interface to the cache, called the Mdl interface.

The Mdl (Memory Descriptor List) contains a physical description of a buffer in memory, according to the physical pages it occupies. This structure should already be familiar to anyone dealing with the network (see the *Windows NT I/O System Specification*).

> \*Please note in the following discussion that the network software does not actually call the CcMdl routines directly, however we describe it that way here for simplicity. See Section 0.*\\

To read from the cache and write to the network, network software may first call **CcMdlRead**, specifying the initialized file object and range of bytes required. **CcMdlRead** returns an Mdl (actually a linked list of Mdls called an Mdl Chain) describing the desired byte range directly in the cache. Note that the reader does not have to specify a transfer that starts on a page or sector boundary, he only needs to make sure he is specifying a file offset with sufficient alignment to satisfy his network device. Once **CcMdlRead** has returned, the pages containing the desired data are locked in memory, and the reader may use the Mdl chain to effect the transfer on the network. Prior to that the network software may wish to prepend an Mdl to the Mdl chain returned by the Cache Manager, in order to describe header information. When the network transfer is complete, **CcMdlReadComplete** must be called to unlock the cache buffers and delete the Mdl chain. Just as described with the Copy interfaces, **CcReadAhead** may be called to have the Cache Manager decide whether he should schedule some data to be read ahead after a **CcMdlRead**.

Similarly for writing to the cache, network software may call **CcPrepareMdlWrite** to prepare a space in the cache to receive the desired data for a specified byte range in the file. The Mdl returned may then be used to specify a direct DMA transfer of the data into the cache off of the network. When the DMA is complete, **CcMdlWriteComplete** must be called to unlock the buffers and free the Mdl chain. Also as in the Copy case, after receiving the **CcMdlWriteComplete** call, the Cache Manager automatically guarantees that the new data is eventually written to disk.

It is acceptable to mix Copy access and Mdl access to the same file.

2.3.3 Accessing Data Directly in the Cache

Local file systems sometimes wish to access data directly in the Cache, possibly modifying it in place. This is particularly interesting for file streams which have been defined to describe file system metadata, such as directories. For this purpose the Cache Manager provides a third interface class referred to as the Pin interface.

If a file system wants to access a structure directly in a stream, possibly modify it, and then release it, it may start by calling **CcPinRead**. **CcPinRead** takes an initialized file object, and the offset and length of the desired byte range. It returns a virtual address at which the desired file data may be accessed, and an opaque Buffer Control Block address (Bcb) which will be used to free the buffer later. If the file system subsequently modifies the pinned data, then it must call **CcSetDirtyPinnedData** before unpinning it. If the file system knows in advance that it will be modifying an entire range of bytes, then it may call **CcPreparePinWrite** instead of **CcPinRead**, and the data will automatically be set dirty (and optionally zeroed in advance). In any case, when the file system is done with the pinned data, it must call **CcUnpinData**, to release the buffer, and allow it to be written if it is dirty.

If the file system knows in advance that it does not need to modify the desired data, or knows in advance that it may not need to modify the data, then instead of calling **CcPinRead** it can use a faster call which is **CcMapData**. **CcMapData** has the same interface as **CcPinRead**, but it is much cheaper since it does not lock the data in memory. If the caller later decides that he does need to modify the data, then he may call **CcPinMappedData** to lock it in memory (and then call **CcSetDirtyPinnedData**). In any case, when done with the mapped and optionally pinned data, the caller must call **CcUnpinData** when done.

Since pinning is generally used for random access to file system metadata, read ahead is usually not performed. As to modified data, the Cache Manager guarantees that any data that was set dirty will eventually be written to disk, typically within about five seconds.

For reasons relating to Cache Manager implementation details, it is not acceptable to mix Pin access to a file with Copy or Mdl access.

2.4 Uninitializing Cache Maps for a File Stream

When a file system is done accessing a given file on a given file object, it must call **CcUninitializeCacheMap**. This routine should generally be called in the file system's cleanup processing. **CcUninitializeCacheMap** must be called for each file object on which **CcInitializeCacheMap** was called.

By default, **CcUnintializeCacheMap** does not remove the file from the cache, it simply tells the Cache Manager that the file system is no longer accessing that file from the specified file object. The file may still remain in the cache for some time until its pages get reclaimed for caching another file (or program image, etc.).

If for any reason a file system does wish to have all or part of a file removed from the cache, **CcUnitializeCacheMap** provides this capability as well (see Section 0).

2.5 Fast I/O Optimization

There is a module in FsRtl which provides fast access to cached data without calling the file system. The routines in this module may be called by the I/O system when caching has already been initialized on a file object. They may also be called by file servers.

Since the file system is never called on the Fast I/O path, it is important that it have the ability to enable or disable these calls. Fast I/O should generally be left enabled unless some condition exists in a file for which correct handling can only be guaranteed by executing the normal file system read and write paths. For example, if any file locks exist in the file, or network oplocks, then execution of a fast I/O path may not work correctly.

If the file system detects a case which makes Fast I/O unsafe, then it must simply clear the IsFastIoPossible boolean in the common Fcb header. This boolean will be tested while owning the Fcb resource shared, and if it is FALSE, the Fast I/O routine returns FALSE as an indication that Fast I/O is not currently possible.

Once the file system detects that the last condition making Fast I/O impossible has been removed, then it should set the IsFastIoPossible boolean to TRUE again.

2.6 Use of the Wait Input Parameter

A number of the Cache Manager routines and the FsRtl routines implementing Fast I/O take a boolean Wait input parameter, and return a boolean result. Use of the Wait parameter is the same in all cases, and is explained here in detail. By far the most efficient operation is always afforded to synchronous callers, i.e., callers who supply Wait as **TRUE** signifying that it is ok to block. This design encourages callers to be multi-threaded in order to get parallel operation, rather than adding lots of threads to file systems and having to pay the expense of locking down and mapping buffers and then context switching to the next available file system thread.

The Wait parameter should be used as follows.

If the caller does not want to block (such as for disk I/O), then *Wait* should be supplied as **FALSE**. If *Wait* was supplied as **FALSE** and it is currently impossible to supply all of the requested data without blocking, then this routine will return **FALSE**. However, if the data is immediately accessible in the cache and no blocking is required, this routine supplies the data and returns **TRUE**.

If the caller supplies *Wait* as **TRUE**, then this routine is guaranteed to supply the data and return **TRUE**. If the data is immediately accessible in the cache, then no blocking will occur. Otherwise, the the data transfer from the file into the cache will be initiated, and the caller will be blocked until the data can be returned.

File system Fsd's should typically supply *Wait* = **TRUE** if they are processing a synchronous I/O request, or *Wait* = **FALSE** if they are processing an asynchronous request.

File system or Server Fsp threads should supply *Wait* = **TRUE**.

2.7 Use of Stream Files

All of the Cache Manager routines which have been presented take a file object as input in order to tell which file a particular operation is directed to. For normal user file opens, it is the user's own file object, which the file system initializes during create, which may be specified to all of the Cache Manager calls. For the case where a file system wishes to cache file system metadata, there is no user file object at hand.

For this case, the I/O system provides the capability of creating a "stream file object", to represent an arbitrary stream as defined by the file system. The file system simply calls IoCreateStreamFileObject (see the *Windows NT I/O System Specification*), and sets up the file object fields as described in Section 0. Note that in this case the common Fcb header and the section object pointers may generally not be resident in an Fcb, but rather in any structure convenient to the file system.

Once the stream file is created and the various pointer fields initialized, the file system may call **CcInitializeCacheMap** at any time to enable caching on this stream.

When done with the stream file, the file system should call **CcUninitializeCacheMap** to turn off caching on that file, and **ObDereferenceObject** with the address of the file object to cause it to subsequently get deleted.

2.8 File System Cleanup and Close Routines

Now that we have presented a walkthrough of the normal Cache Manager interaction, and presented the special case of how stream files may be used, it is important to complete the picture by explaining exactly what expectations are placed on the file system cleanup and close routines (which respond to the Irps with function codes IRP_MJ_CLEANUP and IRP_MJ_CLOSE).

Cleanup is called each time that the last user file handle to a given file object goes away. For normal user files, the file system is guaranteed to get exactly one cleanup call on a given file for each successful create operation which it performs. If the same file is opened and accessed by multiple users, then each open results in a separate file object, and separate cleanup calls on this file will be received as each user file handle is closed.

Within the I/O system, there are various cases where a system component wishes to guarantee that a file object will not be deleted, even if the user closes its handle. It does this typically by calling **ObReferenceObjectByPointer** for the file object. When the system component no longer needs to rely on this file object it calls **ObDereferenceFileObject**. So, for example, a file object is referenced each time an I/O request is issued on it, and dereferenced each time the request is completed. The Cache Manager references a file object the first time the file is cached, and it dereferences the file object when no file objects have it cached, and there are no more dirty pages to flush. A final example is that Memory Management references a file object when a user or the Cache Manager creates a section for mapping that file, and it dereferences the file object when there are no more sections in existence for the file, and the last page has been removed from memory for the file. Note that regardless of how many times the Cache Manager and Memory Management is called for a given file, they only reference the first file object they were called with.

A file system is called to close a file object when the last reference to that file object goes away. This may not occur until some time after cleanup is received on the file object. For example if a system is idle for hours and memory management still has pages for a file that was once mapped, the close call will not occur during this time.

In order to keep track of all these file objects, and thus assist the cleanup and close routines to do the right thing, the file system is expected to maintain two counters in the Fcb. The first counter is essentially a count of user handles, but has been traditionally referred to as the "UncleanCount". The second count is a count of how many referenced file objects referenced exist for a given file, and it has been traditionally called the "OpenCount".

For normal user files, a file system should increment both the UncleanCount and the OpenCount on each successful create. The UncleanCount should be decremented on each cleanup call for a given file, and the OpenCount should be decremented on each close call for a given file.

For stream files, a file system generally only needs to maintain (at most) an OpenCount. Note that a cleanup call will be issued for a stream file object from within the call to **IoCreateStreamFileObject**. It is important to recognize this cleanup call in the file systems cleanup routine (by the way the stream file object was set up), and expediently dismiss it; i.e., simply Noop all cleanup calls to stream file objects. Generally it is also not necessary to maintain an OpenCount for stream files, as a single close call will be received when the one and only file object for the stream is dereferenced the last time.

The Cache Manager expects to be called at **CcUninitializeCacheMap** for each file object which was initialized. If a file is being truncated or deleted, the TruncateSize parameter should be correctly specified to this routine. It is acceptable to call **CcUninitializeCacheMap** on a file object that was never initialized; the Cache Manager will detect this case and do the right thing. In fact, if the file is being deleted or truncated, the Cache Manager definitely should be unconditionally called with the correct TruncateSize, because otherwise the file may not be purged from the cache properly if it had been earlier cached or otherwise mapped via a different file object.

The only way that a file system really knows if both the Cache Manager and memory management (or potentially anyone else) are done with a file, is when the OpenCount finally goes to 0 in the close routine. This is the only time that it is safe for the file system to delete its Fcb, or whatever other structure the file system has associated with a given stream file.

2.9 Using Write Through and Cache Flushing

So far we have only discussed the Cache Manager's default method of lazy writing all dirty data. There are two different ways for either the user program or a file system to force dirty data out to disk and know when it is safely out there. These two methods are write through or flushing.

A user program specifies that it wants all operations on a given file performed write through by specifying FILE_WRITE_THROUGH in its Create options when it opens a file, which the file system can later see in the file object via the FO_WRITE_THROUGH flag in the file object flags. Once this flag is set in the file object, the copy write and Mdl write routines automatically perform write through. As a result, the Lazy Writer will never see dirty data modified through this file object and will never attempt to write any.

Now the only question that remains is, how is write through dealt with in conjunction with pin access? The current file systems in NT have chosen to write through all structure information that is modified as the result of performing an operation on a file object with FO_WRITE_THROUGH set. For such a file object, each time that a pinned Bcb is set dirty, **CcRepinBcb** is called at the same time to guarantee that the Bcb will not be deleted when it is unpinned. In addition, the file systems remember all Bcbs that they have repinned. When the file system request is complete, and all Bcbs have been unpinned and all resources have been released (both very important to prevent deadlocks), and just before completing the Irp, the file systems loop to call **CcUnpinRepinnedBcb** for each Bcb that was repinned. This call is made with the WriteThrough flag specified as TRUE. An unpinned Bcb causes the Bcb to be flushed, and the resulting I/O status is returned. This write through is synchronized with the Lazy Writer, and the Lazy Writer will not lazy write this page a second time.

Flushing is considerably simpler. A user request to flush file buffers results in a flush Irp to the file system. **CcFlushCache** may be called to immediately flush all dirty data to the file. In addition, a file system may choose to flush buffers in any cached file or stream file at any time by also calling **CcFlushCache**. Unlike write through, flushing is not synchronized with the Lazy Writer. However, this only means two things:

- o Any buffer which is currently dirty and pinned will not be flushed. If the file system does not eliminate this possibility by synchronizing this properly within itself, then the affected buffer will still eventually get Lazy Written.
- o If a dirty buffer which is waiting to be flushed by the Lazy Writer is flushed first, then the Lazy Writer will eventually go through the motions of flushing this buffer anyway, but the flush will be nooped by Memory Management, when it realizes that the buffer is no longer dirty.

The current NT file systems only use flushing to mark volumes clean when they have been idle for a while. For this case flushing activity is serialized with everything else on a volume and the first case above does not occur. The second case above can occur, but is benign.

2.10 Valid Data Length and File Size Considerations

Nearly every file system has system has separate concepts of Allocation Size (how much space is allocated to a file) and File Size (how far may a caller read in the file). Allocation Size will typically be a multiple of the disk sector size or allocation quantum (aka cluster size), while File Size may be any number of bytes.

Some file systems (such as HPFS) have, in addition, a concept called Valid Data Length, which is an indication of how much of the file has actually been initialized. Reading beyond Valid Data Length is allowed (unlike reading beyond File Size), however all zeros are returned in the buffer, regardless of what may actually be present on the respective allocated sectors on disk. Returning 0's is both an optimization (we do not have to read the sectors) as well as a security feature (the caller does not get to read the data that used to be in those sectors from some previous file).

It is a very good idea, even for file systems that do not have a concept of Valid Data Length, to present and maintain a concept of Valid Data Length in their implementation and in their interaction with the Cache Manager. This is advisable for both the optimization and security related reasons discussed above. Consider the frequent case where a user creates a file and is sequentially writing to the file. As each user write comes in, the file system typically has to check if it needs to extend the file allocation, and it also may want to advance the File Size early on for internal reasons. When it comes time to call the Cache Manager, say at **CcCopyWrite**, the Cache Manager has to get a page ready to receive the data, and the only way to do that is to fault the page in. This now results in a page fault read back to the file system from within the write path. Fortunately Resources, such as the one synchronizing the Fcb, allow recursive acquisition, so the read proceeds fine. The File Size may already be advanced, but clearly what the file system wants to do in this case is detect that the read is beyond Valid Data Length, so that no real read is required. The file system in this case should simply map the buffer and clear it, and complete the request.

Now, for file systems that actually record Valid Data Length on disk, this field should be updated in a reliable fashion such that even if the system dies, there are still no windows where someone will get to see uninitialized data after the system reboots. This is necessary to really make the file system secure. However, because of the Lazy Writer, the file system can not easily and reliably keep track of when it is safe to advance Valid Data Length, because it cannot make any assumptions about what order the Lazy Writer will flush data to disk. Therefore, the Lazy Writer calls the file system to inform it when it is safe to update Valid Data Length. It does this by issuing a **IRP_MJ_SET_INFORMATION** Irp on the file with **SetEndOfFileInformation** as the operation code and the **AdvanceOnly** flag set. (The **AdvanceOnly** flag can only be set by the Lazy Writer. This call instructs the file system that it can safely update ValidDataLength for the file to the specified size, but only if that would make the new ValidDataLength greater than the current value (someone could have done a WriteThrough or a flush in the meantime which would already advance the ValidDataLength). In some cases, such as for stream files containing file system metadata, the file system simply wishes to consider the entire file to be valid, and it never wants to get the **SetEndOfFileInformation** calls described above. For this case it may specify a **NULL** pointer for ValidDataLength in the **CcInitializeCacheMap** call for this file. This will disable the Cache Manager's ValidDataLength processing just described. For normal files, however, file systems are recommended to support a concept of ValidDataLength in their implementation.

One final note about FileSize. In general paging I/O requests (**IRP_PAGING_IO** set in the Irp Flags) are unsynchronized with File Size Changes. This is true whether these requests emanate from the Cache Manager (especially the Lazy Writer) or whether they occur from user mapped files. Fortunately the rules a file system must follow are simple. On reads, paging I/O requests must obey end of file like anyone else; thus reads extending beyond FileSize should be truncated to the nearest allocation boundary beyond FileSize, and reads totally beyond FileSize should receive **STATUS_END_OF_FILE**. Paging I/O writes are not allowed to extend AllocationSize or FileSize; they are handled similarly to reads. Paging I/O writes extending beyond end of file should be truncated to the nearest allocation boundary beyond FileSize. Paging I/O writes starting beyond FileSize should be nooped with an immediate completion with **STATUS_SUCCESS**. Complete all successful writes with the Information field of the I/O status containing the requested byte count, whether all the bytes really were transferred or not.

2.11 Resource Locking Rules

Doing a caching strategy with a mapped file model is a fairly complex problem. The file system calls the Cache Manager, the Cache Manager calls Memory Management, at which point Memory Management sometimes has to call the file system again. Generally all of this activity stays within the same file. In spite of this complexity, at the time of this writing two disk-based file systems (FAT and HPFS), the CDRom file system, the Lan Manager Redirector, and the Lan Manager Server (through the calls in the next subsection) are all completed and running reliably using the Cache Manager.

Through the experiences gained with the above implementations, a set of resource locking rules has been refined, which seems to allow for good parallelism without deadlock. These rules are as follows:

o Since most activity begins in the file system, the first rule of preventing deadlock is that resources must be acquired in the order: file system resources, Cache Manager resources, Memory Management resources.

  Since some activity begins in the Lazy Writer as it processes its work queue, and since it is necessary for the Lazy Writer to own some its resources across calls to the file systems, the Cache Manager requires some very simple callbacks to allow it to acquire file system resources first before beginning to acquire its own resources. (See Section 0.)

  Some activity also begins in Memory Management, such as in the Modified

Page Writer, or in the servicing of MM services. In general, Memory Management attempts to own no resources at all when it calls the file system.

- The next rule is that the file system resources must support recursive acquisition, since some Cache Manager calls that the file system makes will sometimes result in recursive calls to the file system within the same thread. It's important to note that the recursive calls are never random, but rather logical consequences of the Cache Manager call being made; otherwise recursive resource acquisition could actually be dangerous! One example of a worst-case scenario: in the process of servicing a cached write request, the file system calls CcCopyWrite, which results in a recursive call to the file system for a noncached read to fault in the page to be written, then subsequently a call for a noncached write of the page if the file object is Write Through.

  All of the file systems currently use the executive resource package, which supports single-thread exclusive access or multi-thread shared access. Both exclusive and shared access support recursive acquisition. If an exclusive user recursively requests a resource shared, this is transparently turned into a recursive exclusive acquisition (there is only one release call). Finally, a non-recursive exclusive acquisition can be converted to shared access to allow greater sharing after completing a critical section. (Code which attempts to convert shared to exclusive is almost certain to cause deadlocks.) The calls are: **ExAcquireResourceExclusive**, **ExAcquireResourceShared**, **ExReleaseResource**, and **ExConvertExclusiveToShared**.

  Note that the file systems use some of the other synchronization mechanisms available in NT, but never across calls to the Cache Manager.

- As further assistance, the following table attempts to summarize how the Cache Manager expects the the Fcb to be acquired when it is called at its various entry points. This table was built from the actual usage in HPFS and FAT. Note that the file systems should always attempt to own no other resources exclusive (such as a resource synchronizing allocation on the volume) across calls to the Cache Manager.

  Multiple options in the table below are separated by "/". In the table E = exclusive, S = shared, 0 = unowned, and - = don't care.

Routine	Fcb Res
CcInitializeCacheMap	E/S
CcUninitializeCacheMap	E
CcExtendCachedFileSize	E
CcExtendCacheSection	E
CcFlushCache	E/0
CcPurgeFromWorkingSet	-
CcPurgeCacheSection	-
CcTruncateCachedFileSize	E
CcZeroData	E
CcRepinBcb	-
CcUnpinRepinnedBcb	0
CcIsFileCached	-
CcReadAhead	S
CcSetAdditionalCacheAttributes	E
CcCopyRead	S
CcCopyWrite	E/S
CcMdlRead	S
CcMdlReadComplete	-
CcPrepareMdlWrite	E/S
CcMdlWriteComplete	0
CcPinRead	-
CcMapData	-
CcPinMappedData	-
CcPreparePinWrite	-
CcSetDirtyPinnedData	-
CcUnpinData	-

In addition, the caller should have nothing pinned (repinned is ok) when calling **CcExtendCacheSection** or **CcUnpinRepinnedBcb**.

2.12 Network File Server Interfaces

There is not a lot to say here about how a network file server should use the Cache Manager, as this occurs primarily by virtue of the fact that the file server calls a local file system which is already using the Cache Manager. However, it is quickly worth mentioning that there are basically two ways for a Server to access cached files. Note that in any case servers will tend to open files, close files, and do all other operations except read and write by calling the same file APIs that any other local program would call.

The first alternative for reading and writing file data in a server is to also issue the normal **NtReadFile** and **NtWriteFile** operations. This is not a bad approach, as the server will still benefit from the Fast I/O operations implemented in these services. However, note that all data will be copied into and out of the cache; there is no opportunity to get at the Mdl interfaces at this level.

The second alternative assumes that the Server is running as a kernel-mode process, just like the current Lan Manager Server and all of the file systems. Running in kernel mode the server is able to directly call the FsRtl Fast I/O interfaces layers to either the Cache Manager copy interfaces or Mdl interfaces. The FsRtl interfaces are nearly identical to the respective Cc interfaces documented in this paper; the names are the same except that the Cc prefix is replaced by FsRtl. The difference is that the FsRtl interfaces perform the necessary synchronization with the file system via the Fcb resource, and they also perform a few simple checks (such as IsFastIoPossible as described in Section 0). If the FsRtl routine cannot perform the specified request, then it returns FALSE. If the Server receives FALSE from an FsRtl Fast I/O routine, then it should build the same request in the form of an Irp and queue it directly to the file system via **IoCallDriver** (see the *Windows NT I/O System Specification* and Section 0).

3. File System Maintenance Functions (FSSUP)

3.1 CcInitializeCacheMap

This routine is intended to be called by File Systems only. It initializes the Cache Manager Data structures for data caching. It should be called the first time a File Stream which is to be cached is read or written, or any time the stream is about to be written and it is not already cached (FileObject->PrivateCacheMap == NULL).

The Fcb should be acquired either shared or exclusive when this routine is called.

The three size parameters passed will be captured in the Shared Cache Map, and they must be updated as described later if they change.

If a window to the file cannot be mapped in the normal system cache, then it will be mapped to the specified process, which should presumably be the file system's Fsp process.

The callbacks are described in the next subsection.

```
VOID
CcInitializeCacheMap (
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER AllocationSize,
    IN PLARGE_INTEGER FileSize,
    IN PLARGE_INTEGER ValidDataLength OPTIONAL,
    IN PEPROCESS Process,
    IN BOOLEAN PinAccess,
    IN PCACHE_MANAGER_CALLBACKS Callbacks,
    IN PVOID LazyWriteContext,
    IN PVOID CloseContext
    );
```

Parameters:

FileObject - A pointer to the file object for the stream to be cached.

AllocationSize - The size of the file to be cached. This must be greater than or equal to the actual size of the file. It might be greater, for example, if the file is being created, or may be extended. If supplied as 0, it will be defaulted by the Cache Manager.

FileSize - The exact File Size of the file, beyond which it may not be read.

ValidDataLength - The initialized portion of the file, beyond which 0's must be returned if read (up to FileSize). This number also controls when the Lazy Writer should call Set Information File to advance valid data length. If the caller wants to consider all data valid and does not want callbacks, it can specify **NULL** for this pointer (please refer to Section 0).

Process - Pointer to the process to which the view should be mapped, if it cannot be mapped in system space. This should typically be the Fsp process itself.

PinAccess - FALSE if file will be used exclusively for Copy and Mdl access, or TRUE if file will be used for Pin access. (Files for Pin access are mapped entirely in one view, as it is assumed that the caller must access multiple areas of the file at once. Therefore, it is a good idea organize files for Pin Access into a number of small files.)

Callbacks - Pointer to a vector of Callbacks used by the Lazy Writer (see next subsection)

LazyWriteContext - Parameter to pass to Lazy Write and Read Ahead callbacks.

CloseContext - Parameter to pass to Close callbacks

3.1.1 Cache Manager Callbacks

The Cache Manager must set rules for locking order in order to prevent deadlocks. At a high level these rules are that first the file system is allowed to acquire its resources, then the Cache Manager is allowed to acquire resources, and finally in the worst case Memory Management may also acquire resources. These rules work perfectly well in most cases, since most activity starts in the file system to begin with. One case where there is a problem, however, is in the Lazy Writer. The Lazy Writer must own some Cache Manager resources prior to calling the file system.

To keep this case from producing deadlocks, the Cache Manager requires a set of callbacks to allow the Lazy Writer to acquire any necessary file system resources first, before it begins to acquire its own. This allows the Lazy Writer to continue to follow the locking rules, and prevent deadlocks.

To this end, **CcInitializeCacheMap** takes a pointer to a vector of callback addresses, and two different callback parameters, as defined below:

This routine is called by the Lazy Writer prior to calling **CcUninitializeCacheMap**, since this may result in a Close call to the file system. The context parameter supplied is whatever the file system passed as the CloseContext parameter when it called CcInitializeCacheMap.

```
typedef
BOOLEAN (*PACQUIRE_FOR_CLOSE) (
        IN PVOID Context,
        IN BOOLEAN Wait
        );
```

This routine releases the Context acquired above.

```
typedef
VOID (*PRELEASE_FROM_CLOSE) (
            IN PVOID Context
            );
```

This routine is called by the Lazy Writer prior to doing a write, since this will require some file system resources associated with this cached file. The context parameter supplied is whatever the FS passed as the LazyWriteContext parameter when it called **CcInitializeCacheMap**.

```
typedef
BOOLEAN (*PACQUIRE_FOR_LAZY_WRITE) (
            IN PVOID Context,
            IN BOOLEAN Wait
            );
```

This routine releases the Context acquired above.

```
typedef
VOID (*PRELEASE_FROM_LAZY_WRITE) (
            IN PVOID Context
            );
```

This routine is called by the Lazy Writer prior to doing a readahead. It also uses the LazyWriteContext parameter.

```
typedef
BOOLEAN (*PACQUIRE_FOR_READ_AHEAD) (
            IN PVOID Context,
            IN BOOLEAN Wait
            );
```

This routine releases the Context acquired above.

```
typedef
VOID (*PRELEASE_FROM_READ_AHEAD) (
            IN PVOID Context
            );
```

Finally, this is the complete callback vector, a pointer to which must be passed to **CcInitializeCacheMap**.

```
typedef struct _CACHE_MANAGER_CALLBACKS {

    PACQUIRE_FOR_CLOSE AcquireForClose;
    PRELEASE_FROM_CLOSE ReleaseFromClose;
    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;
```

3.2 CcUninitializeCacheMap

This routine uninitializes the previously initialized File Stream. This routine is only intended to be called by File Systems. It should be called when the File System receives a cleanup call on the File Object.

A File System which supports data caching must always call this routine whenever it closes a file that it is trying to delete, whether it cached the file on the given file object or not. This is because the final cleanup of a file related to truncation or deletion of the file, can only occur on the last close, whether the last closer cached the file or not. Any time **CcUninitializeCacheMap** is called on a file object for which **CcInitializeCacheMap** was never called, the call is benign.

CcUninitializeCacheMap does the following:

- o If a File Stream was initialized on this File Object, it is uninitialized (unmap any views, delete section, and delete Cache Manager structures).

- o On the last Cleanup, if the file has been deleted, the Section is forced closed. If the file has been truncated, then the truncated pages are purged from the cache.

Some times a file system may want pages of the file removed from the cache, even though the file is still open. Examples in the case of a local file system might be if the file has been truncated, a file's media has been removed from the drive. For a network file system client, examples might be if an opportunistic locking protocol dictates that a file may no longer be cached, or perhaps if a virtual circuit goes down. For this purpose **CcUnitializeCacheMap** takes a TruncateSize parameter, which, if specified, causes all pages from the specified file offset on to be purged (removed) from the cache.

BOOLEAN
CcUninitializeCacheMap (
  IN PFILE_OBJECT *FileObject,*
  IN PLARGE_INTEGER *TruncateSize* **OPTIONAL**,
  IN PCACHE_UNINITIALIZE_EVENT *UninitializeCompleteEvent* **OPTIONAL**
  );

Parameters:

FileObject - File Object which was previously supplied to **CcInitializeCacheMap**.

TruncateSize - If specified, all pages should be purged (removed) from the cache starting at, and including, the specified address.

UninitializeCompleteEvent - If specified, this event will be set when the uninitialize is complete, since it may not be complete upon return. If the caller wishes to wait on this event, he must absolutely guarantee that he owns no resources, as this could lead to deadlocks. The format of this structure is:

```
typedef struct _CACHE_UNINITIALIZE_EVENT {
struct _CACHE_UNINITIALIZE_EVENT *Next;
KEVENT Event;
} CACHE_UNINITIALIZE_EVENT, *PCACHE_UNINITIALIZE_EVENT;
```

Returns:

FALSE - if Section was not closed. In this case, if the caller really cares, it may wish to specify and wait on the UninitializeCompleteEvent.

TRUE - if Section was closed.

3.3 CcExtendCachedFileSize

This routine must be called whenever a file has been extended to reflect this extension in the Cache Manager data structures and the underlying section. Calling this routine has a benign effect if the current size of the file is already greater than or equal to FileSize. The Cache Manager must know the correct file size to make the fast read paths work correctly.

VOID
CcExtendCachedFileSize (
  IN PFILE_OBJECT *FileObject,*
  IN PLARGE_INTEGER *FileSize*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileSize - Supplies the new file size for the file.

3.4 CcExtendCacheSection

This routine must be called whenever the allocation for a file has been extended to reflect this extension in the Cache Manager data structures and the underlying section. Calling this routine has a benign effect if the current allocation size of the file is already greater than or equal to NewSize. The Cache Manager must know the correct allocation size in order to insure that the underlying section is large enough.

```
BOOLEAN
CcExtendCachedFileSize (
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER NewSize,
    IN BOOLEAN Wait
    )
```

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

NewSize - Supplies the new allocation size for the file.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Returns:

FALSE - if Wait was supplied as **FALSE**, and the extend was not possible without blocking.

TRUE - if the extend was successfully completed.

3.5 CcFlushCache

This routine may be called to flush dirty data from the cache to the cached file on disk. Any byte range within the file may be flushed, or the entire file may be flushed by omitting the FileOffset parameter.

This routine does not take a Wait parameter; the caller should assume that it will always block.

VOID
CcFlushCache (
    IN PFILE_OBJECT *FileObject,*
    IN PLARGE_INTEGER *FileOffset* **OPTIONAL**,
    IN ULONG *Length,*
    OUT PIO_STATUS_BLOCK *IoStatus* **OPTIONAL**
)

<u>Parameters:</u>

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - If this parameter is supplied (not NULL), then only the byte range specified by FileOffset and Length are flushed.

Length - Defines the length of the byte range to flush, starting at FileOffset. This parameter is ignored if FileOffset is specified as NULL.

IoStatus - The I/O status resulting from the flush operation.

3.6 CcPurgeFromWorkingSet

This routine which may optionally be used to purge all of the pages of a file from the system cache or Fsp working set. The pages do not immediately leave memory, but simply become eligible for replacement.

BOOLEAN
CcPurgeFromWorkingSet (
    IN PFILE_OBJECT *FileObject,*
    IN BOOLEAN *Wait*
)

<u>Parameters:</u>

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

<u>Returns:</u>

FALSE - if Wait was supplied as **FALSE**, and the extend was not possible without blocking.

TRUE - if the extend was successfully completed.

3.7 CcPurgeCacheSection

This routine forcibly purges pages from the cache, automatically uninitializing *all* file objects which have cached this file if necessary. It is meant for infrequent use when dealing with such things as removeable media. Note that this routine is called automatically if the Cache Manager is notified of a file truncation via **CcTruncateCachedFileSize**.

VOID
CcPurgeCacheSection (
  IN PFILE_OBJECT *FileObject,*
  IN PLARGE_INTEGER *PurgeSize*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

PurgeSize - The offset at which the purge is to begin. If not on a page boundary, the page at PurgeSize is first flushed.

3.8 CcTruncateCachedFileSize

This routine must be called any time a local file system truncates a file. It informs the Cache Manager of the new size. If any of AllocationSize, FileSize, or ValidDataLength are larger than this number, they are reduced. Any pages beyond this point are purged from the cache.

VOID
CcTruncateCachedFileSize (
  IN PFILE_OBJECT *FileObject,*
  IN PLARGE_INTEGER *NewSize*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

NewSize - Supplies the new size for the file.

3.9 CcZeroData

This routine may be called to zero a given byte range in a file. As a general service, it may even be called by file systems to zero byte ranges in files which are not cached.

Up to some reasonable amount, this routine will simply attempt to zero data in the cache, and let it be lazy written out. However, beyond a certain size, or for the entire range if the file is not cached, the pages of the file are zeroed by writing to them directly on disk. Note that for files which are not cached, the caller must guarantee that the specified StartOffset is on a physical sector boundary for the underlying disk, otherwise the disk driver will return an error and this routine will raise that error status.

BOOLEAN
CcZeroData (
    IN PFILE_OBJECT *FileObject,*
    IN PLARGE_INTEGER *StartOffset,*
    IN PLARGE_INTEGER *EndOffset,*
    IN BOOLEAN *Wait,*
    OUT PIO_STATUS_BLOCK *IoStatus*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

StartOffset - Supplies the file offset at which zeroing is to begin. If the file is not cached, this offset must be on a hardware sector boundary.

EndOffset - Supplies the file offset at which zeroing is to end.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

IoStatus - Returns the I/O status from the zeroing operation.

Returns:

FALSE - if Wait was supplied as **FALSE**, and the extend was not possible without blocking.

TRUE - if the extend was successfully completed.

3.10 CcRepinBcb

This routine may be called to guarantee that the specified Bcb does not go away. This Bcb address must be one previously returned by either **CcPinRead**, **CcPreparePinWrite**, or **CcPinMappedData**. The caller must subsequently call **CcUnpinRepinnedBcb** for this Bcb. This sequence is usually done in connection with a write through file object, however it may also be done to insure that a buffer does not leave memory to facilitate possible error recovery.

BOOLEAN
CcRepinBcb (
    IN PBCB *Bcb*
)

Parameters:

Bcb - A previously returned pinned Bcb.

3.11 CcUnpinRepinnedBcb

This routine must be called to release a Bcb which was previously specified in **CcRepinBcb**. It releases the Bcb, optionally writing it through to disk first.

VOID
CcUnpinRepinnedBcb (
    IN PBCB *Bcb,*
    IN BOOLEAN *WriteThrough,*
    OUT PIO_STATUS_BLOCK *IoStatus*
)

Parameters:

Bcb - Address of the Bcb

WriteThrough - Specified as **TRUE**, if the data represented by the Bcb should first be written through

IoStatus - Returns the I/O status of the write, if WriteThrough was specified

3.12 CcIsFileCached

This routine is the approved way to determine if a file is cached by any FileObject, whether it is cached by the input file object or not.

Note, if the caller wishes to determine if a given file object itself has been initialized for caching, he should simply test FileObject->PrivateCacheMap. If this field is not NULL, then the file object has been initialized for caching.

BOOLEAN
CcIsFileCached (
    IN PFILE_OBJECT *FileObject*
)

Parameters:

FileObject - The file object in question.

Returns:

FALSE - if no file object has the file cached.

TRUE - if at least one file object has the file cached.

3.13 CcReadAhead

This routine is intended to be called by file systems, after a successful **CcCopyRead** or **CcMdlRead**. The caller essentially specifies information about the previous read. **CcReadAhead** maintains history information about a small number of recent calls for this file object, and attempts to detect if read ahead would currently be adviseable, and if so, whether or not the determined read ahead has already been performed.

If the routine decides that it should perform some read ahead, then a read ahead work request is queued off to one of the Cache Manager's worker threads, in order to not tie up the current thread.

```
VOID
CcReadAhead (
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN ULONG Length,
    IN ULONG StillNeed
    )
```

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file where the last read was just performed.

Length - The number of bytes successfully returned to the reader.

StillNeed - If the read was **CcCopyRead**, this parameter should specify 0. If the read was **CcMdlRead**, then the caller had specified both a Length and MinimumLength that he desired, and we may therefore have given him less than Length. If so, this parameter should specify the Length requested in **CcMdlRead** minus the length we returned to him.

3.14 CcSetAdditionalCacheAttributes

This routine may be called to disable read ahead or lazy write on a file object. Disabling read ahead is always safe. The caller must guarantee that if it disables lazy write, that it will write all dirty pages eventually for the entire file by flushing. A file system should clearly not disable lazy write just because someone opens the file write through, because disabling lazywrite applies to the file itself (not a given file object), and someone else may open the file without write through. Note also that write through is properly synchronized with the Lazy Writer anyway.

VOID
CcSetAdditionalCacheAttributes (
    IN PFILE_OBJECT *FileObject*,
    IN BOOLEAN *DisableReadAhead*,
    IN BOOLEAN *DisableLazyWrite*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

DisableReadAhead - If specified as **TRUE**, read ahead will be disabled.

DisableLazyWrite - If specified as **TRUE**, lazy writing will be disabled.

4. Copy Interface (COPYSUP)

4.1 CcCopyRead

This routine attempts to copy the specified file data from the cache into the output buffer, and deliver the correct I/O status.

BOOLEAN
CcCopyRead (
  IN PFILE_OBJECT *FileObject,*
  IN PLARGE_INTEGER *FileOffset,*
  IN ULONG *Length,*
  IN BOOLEAN *Wait,*
  OUT PVOID *Buffer,*
  OUT PIO_STATUS_BLOCK *IoStatus*
)

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Buffer - Pointer to output buffer to which data should be copied.

IoStatus - Pointer to standard I/O status block to receive the status for the transfer. (**STATUS_SUCCESS** guaranteed for cache hits, otherwise the actual I/O status is returned.)

Returns:

FALSE - if Wait was supplied as **FALSE** and the data was not delivered

TRUE - if the data is being delivered

4.2 CcCopyWrite

This routine attempts to copy the specified file data from the specified buffer into the Cache, and deliver the correct I/O status. If the file object has **FO_WRITE_THROUGH** set, then the data will have been written through to disk upon return.

There is one optimization that is important to note. In **CcCopyWrite**, a fast compare is made to see if the caller happens to be writing the same data that already exists in the file at that point, a common case in certain applications. On the first different byte that is seen, a move of the new data into the cache begins at that point. However, if the buffer is completely the same, then the write is essentially nooped. This optimization does not occur if the buffer was already dirty anyway, or the write is beyond ValidDataLength, or the file does not support ValidDataLength (**NULL** pointer was passed to **CcInitializeCacheMap**). Given these checks, this optimization should always be safe, but the file system should be aware of this optimization none the less.

BOOLEAN
CcCopyWrite (
    IN PFILE_OBJECT *FileObject*,
    IN PLARGE_INTEGER *FileOffset*,
    IN ULONG *Length*,
    IN BOOLEAN *Wait*,
    IN PVOID *Buffer*,
    IN PLSN *Lsn* **OPTIONAL**,
    OUT PIO_STATUS_BLOCK *IoStatus*
)

<u>Parameters:</u>

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file to receive the data.

Length - Length of data in bytes.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Buffer - Pointer to input buffer from which data should be copied.

Lsn - An optional pointer reserved for future support. Should be supplied as **NULL**.

IoStatus - Pointer to standard I/O status block to receive the status for the transfer. (**STATUS_SUCCESS** guaranteed when *WriteThrough* = **FALSE**, otherwise the actual I/O status from the Write is returned.)

<u>Returns:</u>

FALSE - if *Wait* was supplied as **FALSE** and the data was not copied.

TRUE - if the data has been copied.

5. Mdl Interface (MDLSUP)

5.1 CcMdlRead

This routine attempts to lock the specified file data in the cache and return a description of it in an Mdl along with the correct I/O status.

If not all of the data can be delivered, but at least *MinimumLength* can be, then all of the data currently available is still delivered.

If the caller does not want to block, then *Wait* should be supplied as **FALSE**. If *Wait* was supplied as **FALSE** and it is currently impossible to supply the minimum requested data without blocking, then this routine will return **FALSE**. However, if the minimum amount of data is immediately accessible in the cache and no blocking is required, this routine locks the data and returns **TRUE**.

If the caller supplies *Wait* as **TRUE**, then this routine is guaranteed to lock at least *MinimumLength* data and return **TRUE**. If at least *MinimumLength* is immediately accessible in the cache, then no blocking will occur, and all of the available data up to *Length* will be returned. Otherwise, a data transfer from the file into the cache will be initiated for all missing data up to *MinimumLength*, and the caller will be blocked until the data can be returned.

File system Fsd's will typically not use **CcMdlRead**, except to implement the **IRP_MN_MDL** subfunction of read.

File Server threads do not call this routine directly as that is not safe. They may call **FsRtlMdlRead**, which has essentially the same interface. They may also queue an Irp with **IRP_MN_MDL** set in the subfunction of an **IRP_MJ_READ** request. In this case they must pass *MinimumLength* in via the Irp->IoStatus.Information field. They can intercept the Irp completion via a completion routine (see the *Windows NT I/O System Specification*) and read the I/O status and get the Mdl from Irp->MdlAddress. It must then clear this field, or else not allow the Irp completion to continue.

After the caller is done with the data, it must call **CcMdlReadComplete** to free Cache Manager resources.

```
BOOLEAN
CcMdlRead (
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN ULONG Length,
    IN ULONG MinimumLength,
    IN BOOLEAN Wait,
    OUT PMDL *MdlChain,
    OUT PIO_STATUS_BLOCK IoStatus
    )
```

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

MinimimumLength - Minimum data to be guaranteed on return if this routine returns **TRUE**.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

MdlChain - Returns a pointer to an Mdl chain describing the desired data.

IoStatus - Pointer to standard I/O status block to receive the status and byte length of the returned data for the transfer. (**STATUS_SUCCESS** guaranteed for cache hits, otherwise the actual I/O status is returned.)

Returns:

FALSE - if Wait was supplied as **FALSE** and the data was not delivered

TRUE - if the data is being delivered

5.2 CcMdlReadComplete

This routine must be called after the call to **CcMdlRead**, when the Mdl is no longer required. It performs any cleanup that is necessary from the **CcMdlRead**.

Note that this routine does not assume that the calls to **CcMdlReadComplete** will occur in the same order as the calls to **CcMdlRead**, however it does assume that each call to **CcMdlRead** will eventually be followed by a call to this routine.

File systems only use this routine to implement the **IRP_MN_MDL_COMPLETE** subfunction of **IRP_MJ_READ**. Servers may generate this Irp with the *MdlChain* returned from **CcMdlRead** in Irp->MdlAddress, or they may call

FsRtlMdlReadComplete. (Either of these options are available regardless of how they called **CcMdlRead**.)

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

Parameters:

FileObject - Same file object pointer supplied to **CcMdlRead**.

MdlChain - Mdl chain returned from **CcMdlRead**.

5.3 CcPrepareMdlWrite

This routine attempts to lock the specified file data in the cache and return a description of it in an Mdl along with the correct I/O status. Pages to be completely overwritten may be satisfied with empty pages.

File system Fsd's will typically not use **CcMdlWrite**, except to implement the **IRP_MN_MDL** subfunction of write.

File Server threads do not call this routine directly as that is not safe. They may call **FsRtlPrepareMdlWrite**, which has essentially the same interface. They may also queue an Irp with **IRP_MN_MDL** set in the subfunction of an **IRP_MJ_WRITE** request. They can intercept the Irp completion via a completion routine (see the *Windows NT I/O System Specification*) and read the I/O status and get the Mdl from Irp->MdlAddress. It must then clear this field, or else not allow the Irp completion to continue.

After the caller is done with the data, it must call **CcMdlWriteComplete** to free Cache Manager resources.

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

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

MdlChain - On output it returns a pointer to an Mdl chain describing the desired data.

IoStatus - Returns I/O status from potential read required to prepare the data.

Returns:

FALSE - if *Wait* was supplied as **FALSE** and the pages were not delivered

TRUE - if the pages are being delivered

5.4 CcMdlWriteComplete

This routine must be called after a call to **CcPrepareMdlWrite**. The caller supplies the *ActualLength* of data that it actually wrote into the buffer, which may be less than or equal to the *Length* specified in **CcPrepareMdlWrite**.

This call does the following:

- Makes sure the data up to *ActualLength* eventually gets written. If the file object is not write through, the data will not be written immediately and *IoStatus* will simply say *ActualLength* bytes were successfully written on return (even though they were not). This strategy allows the caller to always check the I/O status, and know that if it got an error, the file object must be write through. If the file object is write through, then the data is written synchronously, and the appropriate *IoStatus* is returned.
- Unlocks the pages and deletes the *MdlChain*

File systems only use this routine to implement the **IRP_MN_MDL_COMPLETE** subfunction of **IRP_MJ_WRITE**. Servers may generate this Irp with the *MdlChain* returned from **CcPrepareMdlWrite** in Irp->MdlAddress, or they may call **FsRtlMdlWriteComplete**. (Either of these options are available regardless of how they called **CcPrepareMdlWrite**.)

BOOLEAN
CcMdlWriteComplete (
  IN PFILE_OBJECT *FileObject*,
  IN ULONG *ActualLength*,
  IN PMDL *MdlChain*,
  IN BOOLEAN *Wait*,
  OUT PIO_STATUS_BLOCK *IoStatus*
  );

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

ActualLength - Length of data actually transferred.

MdlChain - Mdl chain returned from **CcPrepareMdlWrite**.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

IoStatus - Returns success, or the actual I/O status from Write Through.

Returns:

FALSE - if *Wait* was supplied as **FALSE** and the pages were not delivered

TRUE - if the pages are being delivered

6. Pin Interface (PINSUP)

The Pin interface is implemented specifically for file systems which are implementing a particular file structure on external media. In a sense it is a more primitive interface than the Copy or Mdl interfaces which always guarantee that data is never left locked in memory, and which automatically deal with cases where the desired data crosses page boundaries, and so on. Therefore there are a couple of special rules that users of this interface must obey.

In general, the Pin interface allows a range of bytes to be locked/pinned in memory (or in the lighter-weight case mapped), and subsequently accessed directly by virtual address. While the data is pinned or mapped system resources are being held. The Cache Manager absolutely relies on the File System to guarantee that it will free these resources by calling **CcUnpinData**.

Forgetting to unpin data is a serious error which can lead to system failure. One approach has proven to be bullit-proof in guaranteeing that file systems never forget to unpin data. By initializing all Bcb variables in a procedure to **NULL**, and nesting all calls which may fail or Raise within a try statement of a try-finally clause, all non-**NULL** Bcbs may be unpinned in the finally clause on the way out. This means they will be unpinned whether the try statement is exited normally, or whether some type of exception occurs which causes the procedure to be unwound.

There is another rule which is a bit more subtle, but for most cases not a problem. Whenever a file system maps or pins a range of bytes in one request that are present on one or more pages, it is invalid for that file system to *ever* make a subsequent request to map or pin a range of bytes in this stream that includes a page from the first request along with a page that was not included in the first request. The reason for this is somewhat due to internal details, but here is a simplified explanation. Once the Cache Manager completes the first request, he has "delivered" this data at a particular range of virtual addresses. If the second request comes along and overlaps the first request, but demands at least one additional page at the beginning or end, it is impossible in general for the Cache Manager to guarantee that it can deliver the new page(s) at contiguous virtual addresses. In the worst case the new page(s) could currently be being accessed as part of another request at a different virtual range. In reality the Cache Manager tries to avoid doing dynamic mapping, but in addition to the potential mapping problems the internal use of Bcbs also restricts overlapping requests.

6.1 CcPinRead

This routine attempts to lock/pin the specified file data in the cache. If successful (returning **TRUE**), a pointer is returned to the desired data in the cache. This routine is intended for File Systems.

If the caller subsequently modifies the data, it should call **CcSetDirtyPinnedData**.

In any case, the caller MUST subsequently call **CcUnpinData**. Naturally if **CcPinRead**, **CcMapData**, or **CcPreparePinWrite** were called multiple times for the same data, **CcUnpinData** must be called the same number of times.

The returned *Buffer* pointer is valid until the data is unpinned, at which point it is invalid to use the pointer further.

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

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Bcb - On the first call this returns a pointer to a Bcb parameter which must be supplied as input on all subsequent calls for this buffer.

Buffer - Returns pointer to desired data, valid until the buffer is unpinned or freed.

IoStatus - Pointer to standard I/O status block to receive the status for the transfer. (**STATUS_SUCCESS** guaranteed for cache hits, otherwise the actual I/O status is returned.)

Returns:

FALSE - if *Wait* was supplied as **FALSE** and the data was not delivered

TRUE - if the data is being delivered

6.2 CcMapData

This routine attempts to map the specified file data in the cache. If successful (returning **TRUE**), a pointer is returned to the desired data in the cache. Mapping data is considerably cheaper than pinning it, however mapped data may not be modified. One either needs to call **CcPinRead** (or **CcPreparePinWrite**) instead if one knows in advance that the data is to be modified, or call **CcPinMappedData** prior to modifying the data and setting it dirty. The caller must not modify the data or set it dirty before calling **CcPinMappedData**.

This routine is intended for File Systems.

The caller MUST subsequently call **CcUnpinData** once with the Bcb returned from this call, or the modified Bcb returned from **CcPinMappedData** if that routine was called. Naturally if **CcPinRead**, **CcMapData**, or **CcPreparePinWrite** were called multiple times for the same data, **CcUnpinData** must be called the same number of times.

The returned *Buffer* pointer is valid until the data is unpinned, at which point it is invalid to use the pointer further.

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

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Bcb - On the first call this returns a pointer to a Bcb parameter which must be supplied as input on all subsequent calls for this buffer.

Buffer - Returns pointer to desired data, valid until the buffer is unpinned or freed.

IoStatus - Pointer to standard I/O status block to receive the status for the transfer. (**STATUS_SUCCESS** guaranteed for cache hits, otherwise the actual I/O status is returned.)

Returns:

FALSE - if *Wait* was supplied as **FALSE** and the data was not delivered

TRUE - if the data is being delivered

6.3 CcPinMappedData

This routine attempts to pin the specified file data which may have previously only been mapped. If successful (returning **TRUE**), a pointer is returned to the desired data in the cache. The data is guaranteed to stay at the same virtual address. If **CcPinMappedData** has already been called for this data or the data was actually pinned in the first place (both cases determined from the *Bcb* IN OUT parameter), then this call is benign. Also note that Bcbs that are either pinned or mapped have to be unpinned, and a call to this routine does not mean that **CcUnpinData** has to be called an additional time, in fact it should not.

Note that *Bcb* is an IN OUT parameter, and that its value may in fact change. If so, it is the new value that must be specified to **CcSetDirtyPinnedData** or **CcUnpinData**; the caller should avoid making copies of the Bcb prior to this call.

This routine is intended for File Systems.

If the caller subsequently modifies the data, it should call **CcSetDirtyPinnedData**.

In any case, the caller MUST subsequently call **CcUnpinData**. Naturally if **CcPinRead**, **CcMapData**, or **CcPreparePinWrite** were called multiple times for the same data, **CcUnpinData** must be called the same number of times.

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

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Bcb - On the first call this returns a pointer to a Bcb parameter which must be supplied as input on all subsequent calls for this buffer.

IoStatus - Pointer to standard I/O status block to receive the status for the transfer. (**STATUS_SUCCESS** guaranteed for cache hits, otherwise the actual I/O status is returned.)

Returns:

FALSE - if *Wait* was supplied as **FALSE** and the data was not delivered

TRUE - if the data is being delivered

6.4 CcPreparePinWrite

This routine attempts to lock the specified file data in the cache and return a pointer to it along with the correct I/O status. Pages to be completely overwritten may be satisfied with empty pages.

When this call returns with **TRUE**, the caller may immediately begin to transfer data into the buffers via the Buffer pointer. The buffer will already be marked dirty.

The caller MUST subsequently call **CcUnpinData**. Naturally if **CcPinRead** or **CcPreparePinWrite** were called multiple times for the same data, **CcUnpinData** (or **CcFreePinnedData**) must be called the same number of times.

The returned *Buffer* pointer is valid until the data is unpinned, at which point it is invalid to use the pointer further.

```
BOOLEAN
CcPreparePinWrite (
    IN PFILE_OBJECT FileObject,
    IN PLARGE_INTEGER FileOffset,
    IN ULONG Length,
    IN BOOLEAN Zero,
    IN BOOLEAN Wait,
    IN PLSN Lsn OPTIONAL,
    OUT PVOID *Bcb,
    OUT PVOID *Buffer,
    OUT PIO_STATUS_BLOCK IoStatus
    )
```

Parameters:

FileObject - A file object for which **CcInitializeCacheMap** has been previously called.

FileOffset - Byte offset in file for desired data.

Length - Length of desired data in bytes.

Zero - If supplied as **TRUE**, the buffer will be zeroed on return.

Wait - **FALSE** if caller may not block, **TRUE** otherwise (see Section 0)

Lsn - An optional pointer reserved for future support. Should be supplied as **NULL**.

Bcb - This returns a pointer to a Bcb parameter which must be supplied as input to **CcPinWriteComplete**.

Buffer - Returns pointer to desired data, valid until the buffer is unpinned or freed.

IoStatus - Pointer to standard I/O status block to receive the status for the transfer. (**STATUS_SUCCESS** guaranteed for cache hits, otherwise the actual I/O status is returned.)

<u>Returns:</u>

FALSE - if *Wait* was supplied as **FALSE** and the pages were not delivered

TRUE - if the pages are being delivered

6.5 CcSetDirtyPinnedData

This routine declares that the data previously read via a call to **CcPinRead** has been modified. It is important to call this routine to insure that the data will eventually be written to disk in a timely manner.

VOID
CcSetDirtyPinnedData (
IN PVOID *Bcb*,
IN PLSN *Lsn* **OPTIONAL**,
)

<u>Parameters:</u>

Bcb - *Bcb* parameter returned from a call to **CcPinRead**.

Lsn - An optional pointer reserved for future support. Should be supplied as **NULL**.

6.6 CcUnpinData

This routine must be called after each call to **CcPinRead**, **CcMapData** or **CcPreparePinWrite**. It unlocks the data from the cache, enabling it to be written if it is dirty. Data will never be written while it is pinned.

VOID
CcUnpinData (
IN PVOID *Bcb*
)

<u>Parameters:</u>

Bcb - Bcb parameter returned from the last call to **CcPinRead**, **CcMapData** (possibly modified by **CcPinMappedData**) or **CcPreparePinWrite**.

7. Revision History

Original Draft 1.0, February 3, 1990

Revision Draft 1.1, March 5, 1990

- Minor changes plus incorporate review comments
- Addition of MmDeclareWsRoutines
- Addition of Section on further Memory Management Requirements
- Addition of entire

Revision Draft 1.2, June 15, 1990

- Complete rewrite except for the first half of Page 1, to reflect the actual implementation driven by the Design Review meeting for Draft 1.1.

Revision Draft 1.3, October 27, 1991

- Greatly expanded detail and update to describe actual implementation for PDK1.