In this chapter:

- ***Flushing the Cache***
- ***Termination of Caching***
- *Miscellaneous File Stream Manipulation Functions*
- *Interactions with the VMM*
- ***Interactions with the I/O Manager***
- ***The Read-Ahead Module***
- ***Lazy-Write Functionality***

8

The NT Cache Manager III

This chapter explains the remaining file stream manipulation functions that were listed in Chapter 6, *The NT Cache Manager I.* These include flushing the cache on demand, purging pages from the system cache, changing file sizes (and informing the Cache Manager of such changes), and terminating caching for a file object.

Following the description of the file stream manipulation functions, I describe some of the interactions the Cache Manager has with both the Virtual Memory Manager and the I/O Manager. I then present a discussion of the lazy-writer and the read-ahead components of the Cache Manager.

Flushing the Cache

Modified cached data is typically written asynchronously to secondary storage media by the lazy-writer component. Also, if the system is running low on available physical memory, the Virtual Memory Manager may flush modified pages to secondary storage. However, any thread that opens a file stream for write access can request that cached data be flushed, and it then has the option of waiting until the flush operation completes before continuing with further processing.

Another method that a thread can use to force modified data to be written to secondary storage is to use write-through operations. This is accomplished by specifying the FILE_WRITE_THROUGH flag when the file stream is opened.

NOTE By requesting write-through mode, a thread informs the file system that a system call (resulting in a call to the file system) to write modified data must ensure that the data has been written to secondary storage before returning control back to the thread. Then, in the event of unexpected system crashes, the thread can guarantee that data was not lost due to in-memory buffering.

An interesting situation arises when multiple open operations are concurrently performed on a file stream, some requesting cached access and others specifying write-through mode. Typically, when the file system receives a write request using a file object that specifies write-through, the file system has to ensure that all modified data for the file stream in the system cache is written to secondary storage, including the newly modified data.* Therefore, requests for access to data using file objects that were created with write-through specified typically result in frequent flush operations performed on the file stream.

The routine used by file systems to request that file stream data be flushed is defined as follows:

```
VOID
CcFlushCache (
    IN PSECTION_OBJECT_POINTERS        SectionObjectPointer,
    IN PLARGE_INTEGER                  FileOffset OPTIONAL,
    IN ULONG                           Length,
    OUT PIO_STATUS_BLOCK               loStatus OPTIONAL
);
```

Resource Acquisition Constraints:

The file system can choose from one of two acquisition options:

- The FCB for the file stream can be acquired exclusively.
- The FCB for the file stream is left unowned. The file system should guarantee in this case that no resources are acquired before invoking the Cache Manager.

* It is indeed possible that a file system may flush only the region specified in the write-through request. Typically, however, most files are relatively small (many are less than 64KB in length), and it might make sense for the file system to request that the entire file be flushed out to secondary storage—hopefully, in a single I/O operation. Only modified pages will ever be written out; therefore, most file systems simply request that the Cache Manager flush the entire file and subsequently let the Cache Manager and the Virtual Memory Manager figure out the pages to be actually written.

Parameters:

SectionObjectPointer

The file system allocates a section object pointer structure when caching is first initiated for the file stream. As noted in Chapter 6, the SharedCacheMap field is used by the Cache Manager to store a pointer to an allocated shared cache map structure uniquely associated with the file stream. The Cache Manager can uniquely identify the file stream that should be flushed using this pointer.

Since a pointer to the section object pointers structure is required, caching must have been previously initiated on the file stream.

FileOffset

This is an optional argument. If supplied, the offset specifies the starting offset of the byte range to be flushed. If not supplied, the Cache Manager assumes that the starting offset is *byte 0* in the file stream. Also, if the file offset argument is omitted, the Cache Manager ignores the Length argument and also assumes that the entire file should be flushed to secondary storage.

Note that the large integer structure is not pushed onto the stack and that a pointer to the large integer structure is required instead.

Length

This is the number of bytes that should be flushed. This argument is ignored if no file offset is supplied to the Cache Manager.

IoStatus

The Cache Manager returns the status code for this operation in the Status field of the IoStatus structure. This is an optional argument and the caller can supply a NULL pointer if the client does not need to know the result of the operation.

Functionality Provided:

The CcFlushCache() routine accepts a request to flush the modified in-memory data to secondary storage. The flushing is performed synchronously, and hence the calling thread should be prepared to block, waiting for the I/O operation to complete.

The implementation of this routine is conceptually very simple: the Cache Manager receives this request and decides if the entire file (beginning at *file offset 0)* should be flushed or if a specific byte range in the file should be flushed. This is determined based on whether the caller supplied a file offset argument or not. If a file offset is supplied, then the requested byte range is flushed; otherwise, the entire file is flushed. If a byte range is supplied, the Cache Manager checks that a valid range has been requested.

The Cache Manager then asks the Virtual Memory Manager to flush the section object (representing the file stream mapping object) to secondary storage. The results of the operation are then returned to the caller, if the caller supplies an loStatus argument.

Note that modified buffers that are currently pinned in memory are not flushed when this routine is invoked. These buffers are flushed asynchronously by the lazy-writer thread after they are unpinned.

Termination of Caching

Once caching has been initiated for a file object, the user can access data directly out of the system cache and also enjoy the benefits obtained from read-ahead and lazy-write operations on the cached data. As was noted in the previous chapter, in response to a file system request to initiate caching, the Cache Manager allocates the shared cache map data structure. Once all processes in the system complete processing data for the file stream, this structure should be deallocated by the Cache Manager and memory pages used to cache data for the file stream should be freed.

After I/O operations on the file stream have ceased, a close operation is performed on the file handle representing the file stream. This operation indicates that the particular process no longer needs to access the data for that file stream, and the file system should terminate caching of data using the associated file handle.

After all processes that opened the file stream close their respective handles to the file, all references to file objects for the file stream are removed. At this time, all data structures used to maintain cache state information can be deallocated and data for the file stream can be purged from system memory.

To understand the sequence of operations that leads to termination of caching for a file stream, let us examine the cleanup and close requests handled by file system drivers.

Once a process completes all desired I/O operations on a file stream, it performs a close operation on the handle representing that file stream. When the last user handle corresponding to the file object is closed, the I/O Manager invokes the file system driver with an IRP containing the major function IRP_MJ_CLEANUP. This is known as a cleanup request to the file system driver.

NOTE The terminology is a little confusing here; you may wonder why a *dose* operation by a process on a handle to the file stream results in a *cleanup* request (IRP) to the file system. And then, at some point, the file system receives a *close* request (IRP) as well for the file stream. The simple answer: someone at Microsoft picked these non-intuitive names! Alternative names for these IPRs could include IRP_MJ_FILE_OBJ_USERS_HANDLES_CLOSEDhmn, for the cleanup request, and IRP_MJ_FILE_ALL_REFERENCES_GONE for the close request. Hopefully, the discussion in this chapter and in Part 3 will help clarify the situation.

The cleanup request notifies the file system that no additional user processes will attempt to access the file stream using the specific file object (an argument to the file system receiving the cleanup request). In response, the file system performs a well-defined sequence of operations; these operations are explained in further detail in Part 3. However, regarding interfacing with the Cache Manager, the file system driver typically does the following:

- The file system flushes all the buffers associated with the file stream.*

  Once the IRP for the cleanup request is completed by the file system, the calling process expects that modified data should have been written to secondary storage, or at the very least, it should be scheduled to be written fairly soon.

- The file system terminates caching for the passed-in file object.

You have already seen the Cache Manager routine used by the file system to flush buffers for a file stream. The routine to terminate caching for a file object associated with a file stream is defined as follows:

```
BOOLEAN
CcUninitializeCacheMap (
    IN PFILE_OBJECT                 FileObject,
    IN PLARGE_INTEGER               Truncatesize OPTIONAL,
    IN PCACHE_UNINITIALIZE_EVENT    UninitializeCompleteEvent OPTIONAL
);
```

The CACHE_UNINITIALIZE_EVENT structure is defined below:

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

' The Cache Manager routine to uninitialize the cache map for a file object also ensures that data for the file stream gets flushed to secondary storage. However, that flush operation is typically performed asynchronously, and invoking the flush call explicitly could be useful to file systems that wish to ensure that modified buffers are written to secondary storage each time a user handle is closed.

Resource Acquisition Constraints:

The FCB for the file stream must be acquired exclusively before invoking this routine.

Parameters:

`FileObject`

This is a pointer to the file object structure for which caching is being terminated by the file system.

`TruncateSize`

This is an optional argument. If the file stream has been deleted, the delete actually occurs only when the final cleanup call for the file stream is received by the file system driver, i.e., when the last user handle is closed.* At this time, the Cache Manager purges all pages from the system cache and forces the section representing the file mapping to be closed if the value of the TruncateSize argument is set to 0.

Alternatively, the file system may wish to truncate the file stream even when there are other open handles for the file stream. In this case, specifying a valid truncate size results in this truncation and pages are purged when the last user handle is closed.

`UninitializeCompleteEvent`

The name for this optional argument is somewhat of a misnomer (maybe UninitializeAndFlushCompleteEvent might have been a better choice). Since the Cache Manager might choose to lazy-write the file stream data to secondary storage and/or lazy-delete the section object representing the file mapping, this argument allows the caller to request that it be notified when the actual flush of cached data and the subsequent uninitialization of the cache map is completed.

Functionality Provided:

The CcUninitializeCacheMapO routine is used by file systems for each file object when a cleanup IRP is received for a file object. Note that this routine should be invoked for every file object, regardless of whether caching had ever been invoked for the file object. This is because truncation related to deletion of a

* This is a peculiarity of the Windows NT system. As you will see in Chapter]0, *Writing A File System Driver II,* to delete a file stream (more specifically, to delete a link/name-entry in a directory associated with a file stream), a process must first open the link for the file stream, mark it for deletion, and finally close the handle. When all file handles for the file stream are closed, the directory entry will actually be deleted (and so will the file stream if the link count for the file was 1). For cached files, when the last user handle for the file stream is closed, the Cache Manager purges all the pages associated with the file stream from system memory and also forces the section to be closed. In other operating systems, it is not always required that a file stream be opened in order to delete it.

file is only performed when the last cleanup operation is invoked for a file stream; i.e., when all user file handles (and therefore all corresponding file objects) have been closed. Similarly, truncation specified for a file stream opened by other processes is performed when all user handles to the file stream have been closed.

Invoking this routine for a file object on which caching has not been initialized has a benign effect.

WARNING Although the above statement is mostly true, if you write a file system driver, be careful to ensure that the SectionObjectPointer field in the file object structure has been initialized prior to invoking this routine. Failure to do so might lead to an exception being raised because the Cache Manager dereferences this field to get to the shared cache map field within the structure. The shared cache map structure in turn is used to determine whether caching is in progress at all for the file stream associated with the file object.

You should ensure that the file control block for the file stream has been acquired exclusively prior to invoking the routine. If caching has been initiated for the file object on which this operation is being performed, caching will be uninitialized. You should note that after returning from this operation, the PrivateCacheMap field in the file object structure will have been reset to NULL.

If the last open user handle to the file stream is being closed, invoking this routine will result in the following:

- If a valid TruncateSize argument was supplied, the pages starting at the supplied offset will be purged from the system cache.
- Modified (but unpurged) pages in the system cache are flushed to secondary storage.
- The shared cache map for the file stream is deleted (actually a lazy-delete will be initiated, since modified pages may be lazy-flushed to secondary storage).

As was noted in Chapter 6, the Cache Manager does not interpret the contents of the byte streams that it caches for other system components. In particular, the Cache Manager is used by file systems to cache not only user data but also file system metadata, such as volume information, extended attributes, directory contents, and other similar information. To initiate caching for such file streams, file systems use the IoCreateStreamFileObject() routine to request that the I/O Manager create a file object representing the file stream. Once this file object has been created, the file system can itself initiate caching on the returned file object and use the system cache to cache nonuser data.

The IoCreateStreamFileObject() routine creates a file object and references it. It then executes a close operation on the referenced file object before returning the file object pointer to the caller. This close operation on the handle for the newly created file object results in a cleanup IRP being dispatched to the file system. The file system should recognize that this is a cleanup request for a special stream file object data structure and simply no-op the call (instead of trying to uninitialize caching for the file object).

You should also note that receipt of a cleanup request on a file object by the file system does not mean that no further I/O requests will be received by the file system using that file object. Although the cleanup request does indicate that all user handles associated with the file object have been closed, it is indeed possible that the Cache Manager (and/or the Virtual Memory Manager) may have referenced the file object and might send read or write-behind requests to the file system using that file object.

Typically, once a file system receives a cleanup request on a file object, further I/O requests should be expected if the following conditions hold true:

- The file object was the first one used to initiate caching for the file stream (i.e., this was the first file object—corresponding to the first open instance among many possible file open instances—that was used in a call to CcInitializeCacheMap() to initiate caching).
- The file system did not invoke CcFlushCache() explicitly when receiving the cleanup IRP and there is modified data in the system cache (you should note that the lazy-writer would then try to write-behind this modified data), or there are other open instances for the same file stream and one or more is resulting in modified data in the system cache (this means that some other thread/process seems to be modifying data for the file stream).

Close Request

When the last user handle associated with a file object is closed, the file system receives a cleanup request. In response, the file system flushes the cached file stream data, uninitializes the cache map, and performs other housekeeping functionality for that file object.

It is important to note that, although all user handles associated with a file object may be closed, there may be references to the particular file object. As long as one or more references exist to a file object, the file object structure cannot be deallocated. However, once the last reference to the file object structure has been removed, the file system receives a close IRP (IRP_MJ_CLOSE). At this time, the file system can perform any final housekeeping associated with the file object before it gets deallocated.

Although most file systems do not interact with the Cache Manager when a close IRP is received for a file stream, it is important to note that the Cache Manager retains a reference to the first file object for a file stream on which caching has been initiated. This may result in a close operation on a file object being delayed until after the cleanup request for the file object has been received and completed.

To clarify this further, consider a file stream for a file *foo* on disk. When *process-1* opens this file, a file object is created to represent the open instance for the file stream. Now, imagine that *process-2* also opens file *foo*. At this time, another file object representing the second open for the file stream is created. Now, let *process-1* initiate the first I/O operation (either read or write) on the file stream. The file system driver initiates caching for the file object, and this request to initiate caching is received by the Cache Manager. While initiating caching for the file object, the Cache Manager notices that this is the first occurrence of caching being initiated for the file stream *foo*. Therefore, the Cache Manager retains a reference to the file object. Although, at some later time, *process-2* might also perform buffered I/O, which causes caching to be initiated for the second file object associated with the file stream *foo*, the Cache Manager does not reference any other file object for the same file stream.

After both processes have closed their respective handles, the file system will get a close IRP only when the Cache Manager (and any other component that references the file object) releases its reference to the file object structure.

NOTE You know that the Cache Manager invokes the Virtual Memory Manager to create a section object representing the file mapping for each file that is cached. When the VMM *is* invoked for the first time on a file stream (to create a section object or file mapping), the VMM also references the passed-in file object. Therefore, the first file object on which caching is initiated for a file stream is referenced at least twice due to the act of caching being initiated—once by the Cache Manager and a second time by the Virtual Memory Manager. Both of these references need to be removed before a close IRP is received by the file system for this particular file object.

This method of referencing the file object and thereby delaying the close operation for a file object results in cached data being kept around in the system cache across user file open and close operations. Therefore, if you open a Microsoft Word document, then close it and then quickly open it once again, the second open and subsequent I/O operations will typically access cached data, and should be a lot quicker than the first one.

Miscellaneous File Stream Manipulation Functions

In Chapter 7, *The NT Cache Manager II,* as well as in this chapter, I presented in detail some file stream manipulation functions used by Cache Manager clients. For example, you now know how to request that the Cache Manager initialize caching for a file stream, how to flush the cache, and how to uninitialize caching. In this section, the remaining file stream manipulation functions made available by the Cache *Manager are* presented.

CcSetFileSizes()

```
VOID
CcSetFileSizes (
    IN PFILE_OBJECT         FileObject,
    IN PCC_FILE_SIZES      FileSizes // See the previous chapter
                                     // for the type definition
);
```

Resource Acquisition Constraints:

The FCB for the file must be acquired exclusively before invoking this routine.

Parameters:

`FileObject`
: This argument contains a pointer to a file object structure associated with the file stream whose size is being modified.

`FileSizes`
: This is an initialized structure with the correct AllocationSize (may be different from the current one), FileSize (i.e., the end-of-file value, which might be changed), and the ValidDataLength. Note that the value in the ValidDataLength field is not used.

Functionality Provided:

When the file system changes either the allocation size for a file or the current end-of-file mark for a file stream on which caching has been initiated, it must inform the Cache Manager of the new sizes. This is done using the CcSetFile-Sizes() routine.

By acquiring the file stream exclusively, the file system ensures that no other thread can concurrently access the data contained within the stream until the file size change operation has been completed. This ensures that users see a consistent view of the data.

The functionality provided by this routine is as follows:

1. If the new allocation size is greater than the previous allocation size, the Cache Manager will extend the section size for the mapped data section object created for the file stream.

   Remember that the Cache Manager provides caching services by mapping the file stream data. Mapping of a file stream is performed by requesting that the Virtual Memory Manager create a section object for the file stream. Therefore, the Cache Manager (once again) asks the VMM to increase the size of the section object to correspond to the new allocation size for the file stream.

   Note that this section object extension operation could result in a recursive callback into the file system.

2. The Cache Manager will update the end-of-file with the new file size value.

   If the valid data length value is being maintained (remember that the file system can decide whether valid data length should be maintained or not), the Cache Manager will also update the valid data length field for the file stream. If the new end-of-file mark is less than the previous end-of-file value, the Cache Manager may purge the cache of all extraneous pages.

   You should note that in certain cases, the NT Cache Manager may actually flush some dirty data to disk before purging the pages from the cache. These flush operations typically cause a recursion back into the file system driver at this time. The flush operations are usually performed when the file system driver has not yet initiated caching on the file stream, yet the user has mapped the file into the process' virtual address space.

CcPurgeCacheSection ()

```
BOOLEAN
CcPurgeCacheSection (
    IN PSECTION_OBJECT_POINTERS        SectionObjectPointer,
    IN PLARGE_INTEGER                  FileOffset OPTIONAL,
    IN ULONG                           Length,
    IN BOOLEAN                         UninitializeCacheMaps
);
```

Resource Acquisition Constraints:

The FCB for the file must be acquired exclusively before invoking this routine.

Parameters:

`SectionObjectPointer`

The Cache Manager uses the SectionObjectPointer to uniquely identify the cached file stream on which the purge operation is being performed.

`FileOffset`

The caller can specify that data be purged beginning at this file offset. If the FileOffset value is nonnull, the Length argument (described below) will be used; otherwise the Length argument will be ignored. Note that if the FileOffset pointer value is set to NULL, all cached pages associated with the file stream will be purged from memory.

Length

The client file system can request that the supplied number of bytes should be purged, beginning at the FileOffset value described above. Note that the Length field is ignored if the value of the FileOffset pointer is set to NULL. If the supplied Length is not a multiple of the PAGE_SIZE for the system, then the value will be adjusted upward to a multiple of the page size.

For example, if the FileOffset is 0, signifying that the purge should begin at the beginning of the file stream, and the Length is 5, then at least one page will be purged. Note that typically the page size is 4K bytes or greater.

`UninitializeCacheMaps`

If set to TRUE, the Cache Manager will force uninitialization of caching for all file objects associated with the file stream.

Functionality Provided:

A file system uses this routine when a file stream is being truncated, but not deleted. This routine causes previously written data to be discarded from memory without being flushed to secondary storage (although a flush might have taken place already due to asynchronous I/O initiated by either the lazy-writer or the modified page/block writer).

The file system supplies a pointer to the section object structure associated with the file stream. The Cache Manager purges the entire file (i.e., all pages in memory for the file stream) if the supplied FileOffset pointer is NULL or if the FileOffset value is 0 and Length is 0. Otherwise, it purges beginning at the supplied offset value for Length number of bytes. Note that if Length is set to 0, then the remainder of the file, beginning at FileOffset, will be purged from memory.

An important point to note here is that user-mapped files cannot be purged or truncated as long as the file is mapped by some process. Therefore, if a user process previously mapped the file (see Chapter 5, *The NT Virtual Memory Manager*, for details), the purge request fails and potentially stale data continues to reside in the system cache. If the purge is unsuccessful, this routine returns FALSE; otherwise it returns TRUE.

WARNING The fact that a purge could fail simply because a user previously mapped the file is a big problem for distributed file systems. As an example, consider a remote file system (e.g., NFS or DPS) that is being accessed by processes on multiple nodes on a network. If some process on *node-1* maps the file into its virtual address space and then a request to truncate the file stream is received from another process on another node, the mapped pages cannot be purged from *node-1* until the process that mapped the file into memory unmaps it. We -will discuss this problem further in a later chapter.

The client can also request that all file objects with caching initiated for this file stream have their cache maps uninitialized. Note that typically, uninitialization of a cache map is only performed by a file system upon receiving a cleanup request. Uninitialization of the cache maps forces all file objects to reinitiate caching whenever new I/O operations are received. If UninitializeCacheMaps is set to TRUE, the Cache Manager will force uninitialization of all cache maps on all file objects associated with this file stream, regardless of whether the purge operation succeeds or fails.

CcSetDirtyPageThreshold()

```
VOID
CcSetDirtyPageThreshold (
    IN PFILE_OBJECT          FileObject,
    IN ULONG                 DirtyPageThreshold
);
```

Resource Acquisition Constraints:

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

Parameters:

FileObject
: This is a file object associated with the file stream on which a restriction is being placed. The file object must have caching initialized.

DirtyPageThreshold
: This is the maximum number of modified pages that can be outstanding at any time for this file stream.

Functionality Provided:

In order to help provide good overall system performance, a file system may restrict the maximum total number of outstanding modified pages associated with a file stream. An example of when this may be necessary is if some process starts rapidly modifying pages for a file stream at a rate faster than the system can cope

with, resulting in pages for other file streams being discarded from memory, to make room for this one particular file stream. This situation leads to unnecessary thrashing of pages in and out of memory and degrades overall system responsiveness and performance to other processes.

By restricting the total number of outstanding modified pages for a file stream, and subsequently using the CcCanlWrite() and CcDeferWrite() routines described in the previous chapter, the file system can ensure that no rogue process can seriously degrade overall system performance by flooding the system cache with data belonging to a single file stream.

CcZeroData ()

```
BOOLEAN
CcZeroData {
IN PFILE_OBJECT        FileObject,
IN PLARGE_INTEGER    StartOffset,
IN PLARGE_INTEGER    EndOffset,
IN BOOLEAN            Wait
);
```

Resource Acquisition Constraints:

The FCB for the file must be acquired exclusively before invoking this routine.

Parameters:

`FileObject`
: This argument contains a pointer to a file object structure for which a range of bytes should be zeroed.

`StartOffset`
: This is the starting offset for the range of bytes to be zeroed.

EndOffset
: This is the corresponding ending offset.

Wait
: This is set to TRUE if the file system is prepared to block in the context of the thread used to invoke this routine. Otherwise, it should be set to FALSE.

Functionality Provided:

This routine can be used by the Cache Manager client to zero a range of bytes within a file stream. The StartOffset and EndOffset arguments determine the actual range of bytes that will be modified (set to zero).

The CcZeroData() routine can be invoked regardless of whether or not caching has been initiated on the concerned file object. If caching has not been previously initiated on the file object or if the file object has been marked for

write-through, i.e., the FO_WRITE_THROUGH flag was set, then the byte range is zeroed directly on-disk.

Note that it is possible that other file objects for the same file stream may have caching initiated (even though the one being used to zero data might not), or that other file objects for the same file stream may not have write-through specified. In such situations, the cached byte range might not be consistent with the newly zeroed range on disk. Therefore, file system developers should be especially careful when invoking this routine if they want to present a consistent view of data to all processes accessing the file stream.

The Wait argument allows the file system to specify whether the file system is prepared to block in the context of the thread used to invoke the CcZero-Data ( ) routine. Writing to secondary storage is potentially a blocking operation, and if write-through is set or if the file object does not have caching initiated and if Wait is set to FALSE, no zeroing of data will be performed. In general, if Wait is set to FALSE, the Cache Manager will be able to successfully zero the specified byte range only if the required space for the byte range is immediately accessible in the system cache. If Wait is set to TRUE, however, the Cache Manager attempts to zero as much of the byte range in the system cache as possible, and the remainder of the specified byte range is zeroed directly on disk.

File systems should note that if the Cache Manager decides to zero data directly on disk, invoking this routine leads to a recursive callback into the file system in the form of a paging I/O write operation. See Chapter 10 for a discussion of the implications on FSD processing when the zeroing operation is performed directly on-disk.

If the Cache Manager successfully zeroes the specified byte range, the call to this routine returns TRUE; otherwise the Cache Manager returns FALSE. This routine raises an exception (e.g., STATUS_INSUFFICIENT_RESOURCES) in the event of an error while allocating resources or while performing I/O to secondary storage.

CcGetFileObjectFromSectionPtrs()

```
PFILE_OBJECT
CcGetFileObjectFromSectionPtrs (
    IN PSECTION_OBJECT_POINTERS     SectionObjectPointer
);
```

Resource Acquisition Constraints:

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

Parameters:

SectionObjectPointer

This is a pointer to the section object associated with the FCB representing the file stream.

Functionality Provided:

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

This routine is typically used when the file system needs to perform an operation requiring a file object pointer that might not be conveniently available at that time.

CcSetLogHandleForFile()

```
VOID
CcSetLogHandleForFile (
    IN PFILE_OBJECT         FileObject,
    IN PVOID                LogHandle,
    IN PFLUSH_TO_LSN        FlushToLsnRoutine
);
```

where:

```
typedef
VOID (*PFLUSH_TO_LSN) (
    IN PVOID                LogHandle,
    IN LARGE_INTEGER        Lsn
);
```

Resource Acquisition Constraints:

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

Parameters:

FileObject

This is a file object for a file stream with which the log handle is being associated.

LogHandle

This is an opaque value (from the Cache Manager's perspective) associated with the file stream identified by the passed-in file object.

FlushToLsnRoutine

This routine is invoked before the Cache Manager flushes buffers (or any BCB) for the file.

Functionality Provided:

As described in the previous chapter, the Cache Manager helps the Log File Service assist file systems that use on-disk logging to help guarantee data consistency and to provide fast recovery from system crashes. The file system can associate a handle with a file stream for a data file using this routine; typically this handle represents a log file associated with the data file.

The file system can also specify a callback routine, which is invoked before the Cache Manager flushes a BCB (Buffer Control Block) to disk. By specifying a callback routine, the file system is informed of the newest Logical Sequence Number (associated with a data record) being flushed, giving the file system an opportunity to ensure that the contents of the log file are written to before the data is written out. Typically, this is required by logging file systems to guarantee data consistency in the event of system crashes. See the previous chapter, especially the discussion on CcSetDirtyPinnedData ( ) , for additional information.

CcSetAdditionalCacheAttributes()

```
VOID
CcSetAdditionalCacheAttributes (
    IN PFILE_OBJECT        FileObject,
    IN BOOLEAN             DisableReadAhead,
    IN BOOLEAN             DisableWriteBehind
);
```

Resource Acquisition Constraints:

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

Parameters:

`FileObject`
: This is a pointer to a file object structure for the file stream for which read-ahead and/or write-behind is being disabled. Caching must have been initiated for the file stream using the passed-in file object, or an exception will be raised.

DisableReadAhead
: If set to TRUE, read-ahead is being disabled.

`DisableWriteBehind`
: If set to TRUE, write-behind (or lazy-write) will be disabled.

Functionality Provided:

Typically, read-ahead and lazy-write (or write-behind) are enabled for all file streams for which caching is initiated. In the event that a file system wishes to

disable one or both of these features for a particular file stream, this routine can be used to do so.

CcGetDirtyPages()

```
LARGE_INTEGER
CcGetDirtyPages (
    IN PVOID                        LogHandle,
    IN PDIRTY_PAGE_ROUTINE          DirtyPageRoutine,
    IN PVOID                        Contextl,
    IN PVOID                        Context2
);
```

where:

```
typedef
VOID (*PDIRTY_PAGE_ROUTINE) (
    IN PFILE_OBJECT                 FileObject,
    IN PLARGE_INTEGER               FileOffset,
    IN ULONG                        Length,
    IN PLARGE_INTEGER               OldestLsn,
    IN PLARGE_INTEGER               NewestLsn,
    IN PVOID                        Contextl,
    IN PVOID                        Context2
>;
```

Resource Acquisition Constraints:

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

Parameters:

LogHandle
: This is a log handle, previously associated with the file stream, for which dirty pages should be returned.

DirtyPageRoutine
: This is the callback routine to be invoked for each dirty page that is found for the file stream identified by the LogHandle input parameter.

Contextl
: This is an opaque (from the Cache Manager's perspective) value to be passed in to the dirty page callback routine.

Context2
: This is a second opaque value to be passed in to the dirty page callback routine.

Functionality Provided:

For logging file systems, the Cache Manager provides this routine to obtain a list of dirty pages for file streams associated with the specified log handle. Cached file

streams may have been previously associated with a log handle. Each of these cached file streams may also have one or more byte ranges cached in memory, with modified data that has not yet been written to secondary storage.

The Cache Manager checks all cached byte ranges in memory, and if it finds any such range that has dirty data for a file stream that was associated with the specified log handle, the Cache Manager immediately invokes the supplied dirty page routine for this byte range. The dirty page routine is given the starting file offset, length of the cached range (in memory), the oldest and newest logical sequence numbers associated with this range, and the two opaque context values that the file system supplied in the call to CcGetDirtyPages ().

The file system should be aware that the callback is invoked at high IRQL with a spin lock acquired. Therefore, the callback is not allowed to take a page fault and it must perform its tasks quickly before returning control back to the Cache Manager. Also, since the Cache Manager invokes the callback for each modified byte range, the callback could be invoked multiple times for every file stream associated with the specified log handle.

The call to CcGetDirtyPages ( ) returns 0 if no dirty pages are encountered, or else it returns the value of the oldest logical sequence number found for a modified byte range for a file stream associated with the supplied log handle.

CcIsThereDirtyData()

```
BOOLEAN
CdsThereDirtyData (
    IN PVPB       Vpb
);
```

Resource Acquisition Constraints:

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

Parameters:

Vpb
This is a pointer to a mounted Volume Parameter Block structure.

Functionality Provided:

In response to this call, the Cache Manager simply scans through all the cached file streams, looking for those that are associated with the supplied VPB and have some modified—but not flushed—data in the system cache. If any such cached file stream is encountered, the Cache Manager returns TRUE.

Note that this is a quick way for a file system to determine whether dirty data for any file stream on a particular volume exists in the system cache.

So how does the Cache Manager determine whether a cached file stream belongs to the specified volume? Recall that the Cache Manager stores a pointer to the referenced file object used in the very first CcInitializeCacheMap() invocation for a file stream. Also, recall from Chapter 4, *The NT I/O Manager*, that each file object has a pointer to the VPB for the volume on which the file stream for the file object resides. Therefore, the Cache Manager can always obtain the pointer to the VPB from the file object for that file stream.

CcGetLsnForFileObject()

```
LARGE_INTEGER
CcGetLsnForFileObject(
    IN PFILE_OBJECT        FileObject,
    OUT PLARGE_INTEGER     OldestLsn OPTIONAL
);
```

Resource Acquisition Constraints:

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

Parameters:

`FileObject`
: This is the file object for the file stream for which information is being requested.

OldestLsn
: This is an optional argument. If the oldest logical sequence number is also required, this argument will be filled in.

Functionality Provided:

This routine simply returns the newest logical sequence number associated with a file stream among dirty byte ranges. If caching has not been initiated for the file stream, or if all data for the file stream has already been flushed to secondary storage, this routine returns 0.

If the OldestLsn argument is supplied, and if there is any dirty data cached in memory, the routine will also return the oldest logical sequence number for the file stream.

Interactions with the VMM

The Cache Manager depends on the services provided by the Virtual Memory Manager to provide caching functionality. Specifically, all policies related to actual memory management, such as allocation of physical memory, creation of file

mappings, destruction of file mappings (section objects), and flushing data cached in memory are performed with the active assistance of the VMM subsystem.

Dependencies upon the VMM exist throughout the Cache Manager implementation; most of these dependencies are resolved by internal calls to the VMM. Unfortunately, most of the Cache Manager calls to the VMM use routines that are not exposed to other kernel developers. Still, in order to understand the Cache Manager, it is useful to be aware of the dependencies that the Cache Manager has on the VMM. This allows you to be more aware of the dependencies within the NT Executive as a whole, and if you design or develop a kernel-mode driver, you will undoubtedly see stack traces during system crashes that indicate that both the Cache Manager and VMM were involved in calls that ended up in your code.* Let us examine the various points where the Cache Manager requires the assistance of the Virtual Memory Manager.

At system initialization time, the Cache Manager requires a range of addresses within the system virtual address space to be reserved for its exclusive use. This is performed by the VMM automatically, and therefore the Cache Manager can be guaranteed an available fixed-size virtual address byte range.

When the Cache Manager initializes, it needs to determine the number of threads it should create, the maximum number of dirty pages that the entire system cache can contain, and other such configuration parameters. To determine absolute values, the Cache Manager uses a VMM routine called MmQuerySystemSize ( ) (see Chapter 5 for the definition of this routine).

Assume that a file system invokes the CcInitializeCacheMap() routine described earlier to initiate caching for a file stream using a specific file object. To service this request, the Cache Manager checks whether caching was previously initiated for the file stream using any other file object. If this is the first instance of caching being initiated for the file stream using any file object, the Cache Manager has to map the file stream into memory (specifically, into the reserved virtual address range set aside for the Cache Manager). The Cache Manager achieves this by using the MmCreateSection() routine, and although this routine is not exported by the Windows NT Executive for use by any external kernel drivers, the routine is amazingly similar to the NtCreateSection() (also known as the ZwCreateSection ( ) t) system call. The MmCreateSection ( ) routine results in the creation of a section object that represents a file stream mapping to the

---

* My apologies for insinuating that newly developed kernel code by readers could lead to system crashes. Unfortunately, this is a fact of life that all kernel developers either learn to accept and so become better designers/developers, or deny stoically forever, resulting in their customers finding out the effects of the designers intransigence the hard way.

t This routine is defined in Chapter 5.

VMM. A pointer to the section object can subsequently be used by the Cache Manager whenever it needs to manipulate the section or its contents.

When the allocation size of a file stream is extended, the Cache Manager must extend the section associated with the specific file stream if the file stream was previously mapped into the system cache. This is achieved, once again, by invoking the VMM via a routine called MmExtendSection (). Unfortunately, this routine is not defined or exposed by the NT Executive and hence the arguments supplied to this routine are subject to change.

Whenever a file system tries to perform I/O to a cached byte range and the request is transferred to the Cache Manager, the Cache Manager must map a view of the affected byte range into the system virtual address space (using the section object created earlier when caching was initiated). This is achieved by invoking the VMM routine called MmMapViewInSystemCache ( ) . Note that, although this routine is not exposed, the functionality provided is similar to that of ZwMap-ViewOfSection (). The difference, however, is that the requested view is mapped into the specific reserved virtual address range set aside for the Cache Manager.

Correspondingly, whenever the Cache Manager wishes to discard a previously mapped view, it uses the VMM routine MmUnmapViewInSystemCache ( ) .*

Now, when the Cache Manager has to flush the data associated with a file stream, the actual flush is performed by invoking the MmFlushSection () call. The interesting point to note is that, for any data present in the system cache, the Cache Manager never directly invokes the file system or the I/O Manager to write data out, instead, the Cache Manager always requests that the VMM flush out the associated section (and more specifically, a byte range within the section), thereby always synchronizing with the modified page writer thread within the VMM. This is true even when the lazy-writer component within the Cache Manager performs asynchronous write-behind of data.

* Unmapping a mapped view is almost never a cheap operation. If pages are physically assigned to some addresses within the mapped view, invoking this routine leads to TLB (Translation Lookasidc Buffer) flushes. This degrades performance somewhat.

NOTE The way a file system eventually sees this request to write data out is in the form of a noncached, paging I/O -write request that comes via the VMM and the I/O Manager. Again, if you were to see the stack trace, you might be able to see the Cache Manager routine (CcFlushCache ( )) in the trace. Sometimes, if the modified page writer is already in the process of asynchronously flushing out the same byte range, you may not even see the Cache Manager in the trace, since the Cache Manager's request to the VMM would be blocked waiting for the asynchronous flush to complete.

When the Cache Manager has to read data into the system cache, it does not even have to invoke the VMM explicitly. It simply attempts to copy data from the mapped view in the system cache into the user-supplied buffer. This causes a page fault, which is automatically (and normally) handled by the page fault handler component of the VMM. Note that when the read-ahead component of the Cache Manager wishes to bring data asynchronously into the system cache, it, too, simply tries to touch (or access) a byte from each page that is being brought into the system cache. Once again, the act of accessing a byte leads to a page fault (if the data is not already in physical memory) and this page fault is resolved by the VMM. Remember that the page fault will eventually be resolved by a noncached, paging I/O read request to the file system by the VMM, although the file system cannot tell whether the page fault is due to the Cache Manager touching a page that was not in memory or some other process doing so.

NOTE File system drivers (especially those for the Windows NT operating system) have to be fully aware of how an I/O request arrives at either the read or write dispatch entry point. Therefore, file systems work very hard to determine the sequence of operations that caused the dispatch entry point to be invoked. This applies equally well to paging I/O operations. Part 3 discusses this topic extensively.

There are other VMM routines that are available only to the Cache Manager for use during normal operations. For example, the Cache Manager can check whether an address is backed by a physical page (and if not, request that the page be made resident and zeroed) using the MmCheckCachedPageState ( ) routine. The Cache Manager can set an address range to modified (causing the data to be flushed out) using a routine called MmSetAdd.ressRangeMod.ified( ). Also, the Cache Manager can force pages to be purged from the system cache using the MmPurgeSection ( ) routine.*

* The request to purge might be failed by the VMM if the section is mapped by a user process as well as by the Cache Manager.

Note that the Cache Manager is treated as any other (though slightly special) client by the VMM. This means that the VMM maintains a working set for the pages allocated to the Cache Manager and trims or expands the physical memory that is assigned to the Cache Manager, based upon demands made by the Cache Manager and other modules in the system. This allows the VMM to make global allocation decisions wisely and prevents file data caching from overwhelming the system to the extent that all other work becomes impossible.

Although routines listed in this section might not be exported and described in detail for use by other kernel-mode subsystems, it is obvious that special support is provided by the VMM to the Cache Manager. This makes the Cache Manager unique within the NT Executive and allows it, in turn, to provide caching support to other modules, such as file systems.

One final note: although the Cache Manager uses the services of the NT VMM, the VMM never needs to utilize services provided by the Cache Manager.* Therefore, the relationship is mostly a one-directional, client-server relationship.

Interactions with the I/O Manager

The Cache Manager uses the services of the I/O Manager, just like other system modules. For example, the Cache Manager must request that the I/O Manager allocate an IRP for it using the `loAllocatelrp()` system call. The Cache Manager uses `loCallDriver()` when it invokes the file system to notify the file system about changes in the file size. Other I/O Manager routines such as `loAllocateMdl()` and `loRaiselnformationalHardError()` are also used by the Cache Manager.

An important point to note about the interactions between the Cache Manager and the I/O Manager is the existence of the fast I/O path described in the previous chapter. The I/O Manager tries to increase system throughput by bypassing the file system completely and invoking the Cache Manager directly to satisfy user I/O requests for cached file streams. If this fails, the I/O Manager defaults to using the standard I/O path through the file system driver. The fast I/O path is described in detail in the previous chapter and further information is also available in Chapter 11, *Writing a File System Driver III*

---

* As with everything else, this is almost true. It appears that the VMM uses a single routine, CcZeroEndOfLastPage(), provided by the Cache Manager, when mapping a section on behalf of a user process. The purpose of this routine is to check for uninitialized pages at the end of the file stream being mapped, and if found, to zero these pages by freeing them. This routine is exported for use by other kernel developers, but the lack of sufficient documentation explaining this routine seems to deter usage by any other module.

The Read-Ahead Module

The Cache Manager helps enhance system responsiveness and throughput by providing read-ahead functionality. This means that the Cache Manager tries to bring data from secondary storage into the system cache before it is even requested by a user process. Subsequently, when the user process tries to access the byte range that was read-ahead into the system cache, the user I/O request can be immediately satisfied from the data present in the cache, avoiding a time consuming read operation to obtain data from secondary storage or from across the network.

In order to provide read-ahead, the Cache Manager must be able to answer the following questions:

- Should read-ahead be performed for a specific file stream?
- If it is determined that read-ahead should be performed for a cached file stream, when should read-ahead be initiated?
- What should be read-ahead into the system cache?
- Given a user request that was recently satisfied, what would be the byte range that the user process is likely to access in the near future?
- Who does the actual read-ahead operation—one thread, many threads, specially reserved threads, or simply system worker threads?
- If errors occur while trying to read-ahead data into the system cache, what should the Cache Manager do in response to these error conditions?

Let us examine each of the issues listed here to see how the Cache Manager implements read-ahead functionality.

Should Read-Ahead Be Attempted for a File Stream?

The default answer to this question is yes. Read ahead is generally attempted by the Cache Manager for all file streams that are cached in memory. The exception is that read-ahead is not attempted for file streams on which caching was initiated specifying that PinnedAccess would be used to access cached data.

It is possible for file systems to request that read-ahead be disabled for specific file streams. This can be achieved by the CcSetAdditionalCacheAttributes() routine described earlier in this chapter.

When Should the Cache Manager Try Read-Ahead?

Read-ahead is attempted by the Cache Manager either at the explicit request of file system drivers or automatically when I/O requests are serviced by the Cache

Manager. A file system can request that read-ahead be performed by using the following system defined macro:

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

where:

FO = file object pointer
FOFF = file offset from where last read request was initiated
LEN = length in bytes of last read request

As you can see, the system will perform read-ahead (at the explicit request of a module such as a file system) only if the last read operation was greater than 256 bytes. Apparently, invoking read-ahead for smaller read operations actually results in degraded system performance.*

The CcSch.eduleReadAh.ead( ) routine is automatically invoked by the Cache Manager whenever CcCopyRead( ), CcFastCopyRead( ), or CcMdlReadO are invoked. The Cache Manager checks read-ahead is not currently active for the file stream and, if not active, will invoke CcScheduleReadAhead( ). Of course, if read-ahead is disabled for the file stream, it will not be attempted.

NOTE The Cache Manager often schedules read-ahead concurrently with trying to read in the current user request. Typically though, the Cache Manager will not get ahead of itself and the user request will be received by the file system before the read-ahead request makes it to the file system.

What Does the Cache Manager Read-Ahead?

The function of read-ahead is to try to anticipate the byte range the user process might next access and preread into memory. The Cache Manager relies on the property of *locality of reference* to make educated guesses about the byte range that the user process might access next, following the current read request.

Simply stated, this means that a user process is likely to access a byte range that is within a few bytes of the byte range that was just accessed. Therefore, say that a process accessed bytes 1000—5000 within a file with length of 2MB. There is a

* The caller is not required to use the read-ahead macro; it's simply a good idea.

greater probability that the process will next try to access byte offset 10,000 than that the process will next try to access byte offset (1MB + 1).

A process can specify when opening a file whether the file stream will be accessed in a sequential manner by means of the FO_SEQUENTIAL_ONLY flag in the file object. This flag serves as a valuable hint to the Cache Manager, which then tries to keep at least two read-ahead granularities ahead of the current read operation (although the default read-ahead granularity is one PAGE_SIZE, it can be changed using the CcSetReadAheadGranularity ( ) routine described earlier). This means that if the user process has just accessed the first page length in the file stream, approximately two additional pages beyond the ending offset of the first page will be read-ahead by the Cache Manager.

Even if the sequential-only flag is not supplied by a process when opening a file stream, the Cache Manager keeps track of read requests performed via the copy or the MDL interfaces. If the Cache Manager detects a sequential nature in the read operations being performed (e.g., if the previous two read requests were close enough to be considered sequential), the Cache Manager will attempt to read-ahead from the offset where the last read request ended (rounded up to a multiple of the page size).

NOTE The Cache Manager masks off certain *noise bits* when trying to characterize two or more read operations as being sequential or not. For example, if read operation #1 starts at offset 0 and has a length of 4096 bytes, and read operation *#2* starts at offset 5002 and has a length of 1500 bytes, the Cache Manager will disregard the fact that operation #2 starts 6 bytes beyond the end of the first request and will consider the two read requests to be sequential in nature. Therefore, read-ahead will be attempted.

Note that the Cache Manager keeps track of whether sequential accesses are being performed in the forward or in the reverse direction. Read-ahead will also be performed if a process begins reading from the end of a file stream sequentially toward the beginning of the file.

Who Performs the Actual Read-Ahead Operation?

The read-ahead is performed in the context of a system worker thread. As you know, there are worker threads available to asynchronously perform operations that are not time-critical. Therefore, the Cache Manager simply posts a request using the ExQueueWorkItem() system call. Note that the Cache Manager specifies that the request be posted onto the critical work queue.

This ExQueueWorkItem() routine is defined in the documentation for the Device Driver's Kit. It allows a work request to be queued to a global system queue. The work item is subsequently performed in the context of a system worker thread when such a thread becomes available.

There are three categories of work requests that can be queued: *delayed work requests, critical work requests,* and *hypercritical work requests.* The read-ahead operation is queued onto the critical work queue. Note also that, when invoking this routine, the caller has to specify the actual function call that the worker thread must invoke, and the caller must also supply an opaque context pointer that will be supplied as an argument to the specified function call.

In Chapter 7, I mentioned that a file system has to supply callback routines when initiating caching on a file stream. These callback routines are used to maintain locking hierarchy between the Cache Manager, the VMM, and the file system driver. One of the callback routines that a file system must supply (AccruireFor-ReadAhead()) allows the Cache Manager to acquire file system resources before initiating read-ahead. This callback is invoked by the thread that actually performs the read-ahead operation on a file stream. Upon completion of the read-ahead operation, the thread invokes ReleaseFromReadAhead() to inform the file system that resources previously acquired should now be released.

Further information on the implementation of these callback routines is given in Chapter 11.

What If There Are I/O Errors in Attempting the Read-Ahead?

If there are I/O errors during the read-ahead, the Cache Manager ignores them and simply aborts the current read-ahead operation. It is possible that read-ahead might be retried in the future.

Lazy-Write Functionality

Just as in the case of read-ahead operations, the Cache Manager tries to help enhance system responsiveness and provide greater throughput by implementing write-behind (or lazy-write) functionality. Here, the Cache Manager does not write modified data supplied by a user process, either directly to disk or across the network to a file server. Instead, the Cache Manager buffers the data in memory and periodically flushes modified data asynchronously to nonvolatile storage. This asynchronous, periodic flushing of data is called write-behind (or delayed-write or lazy-write) functionality.

As in the case of read-ahead operations, the Cache Manager must be able to answer the following questions:

- Should lazy-write be performed for a specific file stream?
- If it is determined that lazy-write should be performed for a cached file stream, how is the lazy-write functionality initiated?
- Who does the actual lazy-write operation?
- If errors occur during trying to lazy-write data from the system cache, what should the Cache Manager do in response to these error conditions?

Let us examine each of the issues listed above to see how the Cache Manager implements lazy-write functionality.

Should Lazy-Write Be Attempted for a File Stream?

By default, all cached file streams are lazy-written unless lazy-write has been disabled for a specific file stream, using the CcSetAdditionalCacheAttributes() routine described earlier in this chapter. However, data that is currently pinned in memory is not flushed until the data is unpinned.*

Furthermore, temporary files are not written to secondary storage, since the application has specified that the file be deleted anyway once the last user handle to the file has been closed.

How Is Lazy-Write Functionality Initiated?

The lazy-writer is invoked in one of the following ways:

- The Cache Manager has a DPC (Deferred Procedure Call) timer that pops once every few seconds (between 1—3 seconds). When this timer pops, it schedules a scan through the cache to find candidates that should be flushed to secondary storage.
- The Cache Manager explicitly schedules a scan of all cached byte ranges to search for modified ranges that can be flushed to secondary storage.

Once a scan is initiated, the Cache Manager sets a target amount of data that it would like to flush in that instance. Typically, the Cache Manager determines that one quarter of the currently modified (or dirty) data in the system cache should be flushed. This allows the Cache Manager to sweep through all of the dirty data in four scans through the cache. Note that the scan always begins at the point at

* This is different from the read-ahead case where file streams that specified PinAccess as TRUE would simply not have read-ahead initiated for them. Lazy-write, in contrast, is performed on these file streams, but pinned byte ranges are skipped until unpinned.

which the last scan terminated; this ensures that all dirty pages in the system cache are flushed out to secondary storage in a round-robin fashion.

When searching the cache for candidates to be flushed, the Cache Manager simply looks at each shared cache map that has dirty pages outstanding and schedules an asynchronous write operation for the shared cache map. The Cache Manager continues to schedule such write operations until the targeted limit of 1/4 of the total dirty pages has been exceeded or the Cache Manager runs out of dirty pages to be flushed.

The Cache Manager also tries to adapt the rate at which it flushes data to disk. For example, if the Cache Manager notices that modified pages are being produced at a fast rate, it will try to flush out more data in the current scan to keep the total number of outstanding dirty pages constant in the system cache.

Who Performs the Actual Lazy-Write Operation?

As noted in the preceding section, the Cache Manager periodically scans through all the shared cache maps and schedules asynchronous write operations for those that contain dirty data. Just as in the case of the read-ahead functionality, the actual write-behind operation is performed in the context of a system worker thread. The write-behind requests are posted to the global critical work queue and are picked up by available system worker threads assigned to service that queue.

Before actually posting the write to the file system, via a synchronous call internally to CcFlushCache ( ), the thread performing the write-behind will invoke the file system callback for AcquireForLazyWrite (). After completion of the flush operation, the Cache Manager will invoke a corresponding callback ReleaseFromLazyWrite ( ) to inform the file system that it can release its resources.

The thread performing the write-behind will also check to see if the write operation extended the ValidDataLength associated with the file stream. If the current ValidDataLength is exceeded, the file system will be invoked via the IRP_MJ_SET_INFORMATTON I/O Request Packet (the AdvanceOnly Boolean flag will be set to TRUE),* and informed of the new valid data length for the file stream.

Finally, the thread that performs the lazy-write operation also performs a lazy/delete operation of the shared cache map for the file stream if such a delete had

* A description of this IRP is presented in Part 3. There, you will also find an explanation of this special flag that exists solely to inform the file system that the ValidDataLength for the file stream must be changed.

been requested earlier by the file system. Of course, no file object should be actively referencing the shared cache map so that the delete operation is attempted. If any thread is awaiting the deletion of the shared cache map, the appropriate event will be set in order to inform the thread that the shared cache map was deleted.

What If There Are I/O Errors in Attempting the Write-Behind?

Consider a situation where the system worker thread, performing a lazy-write, encounters an error during the actual write operation. In this case, the thread attempts to retry the write operation—one page at a time. The theory here is to try to write out as much data as possible.

Once the retry operation has been attempted (one write per page being flushed to secondary storage), any I/O errors encountered while retrying are essentially ignored. The thread marks the pages as clean and thereby effectively loses all data that could not be flushed to secondary storage. This is a nasty side effect of the delayed-write method because a user process that opened the file stream wrote data that was buffered, received a successful return code, closed the file stream, and exited can lose the data that it thought had been successfully written out, due to the failure of the write-behind attempt!

However, the Cache Manager does pop up a message on the system console, and also writes out the message to the error log, stating that some data for the specific file stream was lost in the write-behind process. Unfortunately, by the time the system operator receives this message, it is already too late to save the data, since the pages have been marked clean.*

With this chapter, we have concluded our discussion of the NT Cache Manager. In Part 3, you will find code examples and discussions of how file systems and filter drivers take advantage of the services provided by the NT Cache Manager.

---

* It would he wise for system administrators to invest in high availability software (and redundant hardware) that provide for mirrored copies to avoid such nasty surprises. However, this still does not guarantee that such data loss will never occur.