Image Not Available

Chapter 5
FileDir Design Description

Abstract

FileDir provides file system services including file, directory, and synchronization functions. This module contains the cache for the file system and support for auto-reconnection of file system resources.

Contents

Introduction

FileDir contains three areas of file system functionality: files (APIs beginning with "FILE"), directory (DIR), and synchronization (SYNC) functions. The file and directory functions use memory caching for improved performance. Caching is vital because memory is shared among the system as a whole.

FileDir is session-protocol-independent and transport-protocol-independent, and is written entirely in "C," making it portable to other operating systems and platforms.

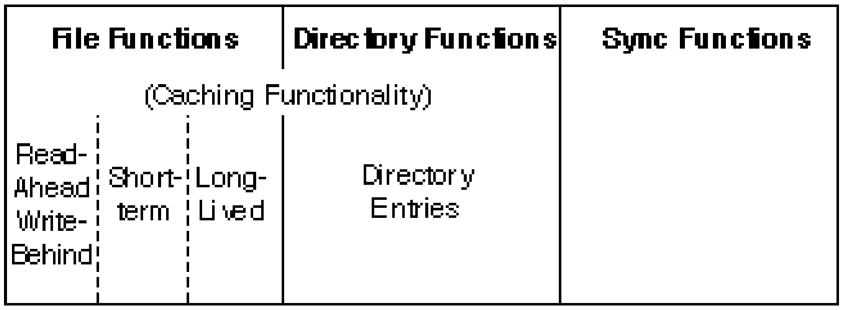

Figure 1. FileDir functional architecture. Both FILE and DIR functions including caching capabilities, as shown.

Overall Design Description

The FILE APIs perform opens, reads, writes, seeks, and closes, as well as getting and setting file information (for example, date, time, size).

DIR APIs operate on directory entries and logical volume information for a given directory level. Functions include retrieving and setting attributes, and creating, deleting, renaming, and searching directory entries.

SYNC APIs operate on physical records, logical records, and semaphores. SYNC functionality includes logging, locking, releasing, and clearing of physical and logical records, and opening, creating, examining, waiting, signaling, and closing semaphores.

FileDir functions are fully reentrant and can be called from a foreground context.

Memory for caching file and directory data is shared with the entire system through the NIOS Memory Pool NLM (MemPool) functions (see the *NetWare I/O Subsystem (NIOS) Design Specification*). MemPool functions allocate and return memory, and when the system needs more memory and there isn't any free, the module that allocated the memory is "called back" and asked to return it.

Auto-Reconnect Capabilities

A server connection may become invalidated by events such as the server going down, the server's watchdog facility timing out a connection, or as a result of a mobile operation. For a more detailed discussion of mobile operations see the *Disconnected Operations Design Specification*.

When a connection becomes invalid, the connection must be re-validated through the process of auto-reconnection. Auto-reconnection events are initiated by lower Requester layers that tell FileDir which connection needs its resources reestablished. Information for each resource is stored during its creation and modification so that it can be reestablished on-the-fly.

FileDir auto-reconnects files, locks, drive mappings, and semaphores.

During the time a connection is invalid, previously-accessed resources may be deleted by other users or altered such that resynchronization is not possible. (An example of when resynchronization is not possible is when the user is reading a file, loses connection to the server, is reconnected, and discovers that the directory the file was in has been deleted.)

When this occurs, a pop-up window indicates which resource is unable to be reestablished. (Note: The pop-up will remain visible until acknowledged by the user or until a configurable time limit expires, but the pop-up will reappear later and this cycle will repeat. This allows batch operations to continue unattended but still provides notification if a user was merely away from the computer temporarily.)

All file system resources have a handle associated with them. The handle is a 32-bit value, and access to the resource is limited to the correct process group and/or process. During access to the resource, the process group and process is compared against the creator's identifiers and must match before access is allowed. All file system functions support long filenames (up to 255 bytes). Long filenames must be associated with a name space since there are different file systems supported by different servers. Therefore, all calls to the FileDir module that specify paths also must specify the name space. Windows 95 uses the OS/2 name-space.

Dependencies

The FileDir module depends on NIOS functions, the Session Multiplexor (SessMux) module with its corresponding NCP module, and the ConnMan module.

FILE Functions

FILE read and write functions use a cache to buffer network access. Open, create, close, remote-copy, seek, and flush functions are monitored to ensure file cache data integrity.

Caching

Because user reads and writes are the most frequently requested network operations, network performance improves by buffering these reads and writes, thereby reducing network access. This is called *caching*.

The integrated cache buffers network reads and writes so that network access is made less frequently and in bigger blocks. Instead of having every user read and write go directly to the server, the requests are instead sent to a cache buffer.

Goals of Integrated Caching

The goals of the integrated cache are threefold:

- To buffer small reads and writes (small meaning smaller than the size of the cache blocks).
- To buffer frequently accessed data so that network traffic and delays are reduced.
- To maintain cached data across file open/close operations for frequently accessed files.

The integrated cache monitors open, create, close, remote-copy, and flush functions, altering cache status accordingly.

Cache memory consists of data blocks, each 4096 bytes of data plus a 64-byte header.

Available blocks are maintained in an LRU linked list. That is, when a request is made for a cache data block and none is available, the memory module locates the least-recently-used block of memory, notifies the owner of the block to free the block, and appropriates the block for cache memory.

There are five caching modes: read-ahead/write-behind, short-term cache, long-lived cache, warehouse cache, and no cache. Each of these is defined in the *Definitions* section that follows.

Caching Definitions

Dirty Buffer. A buffer that contains a block of data which has been modified (written to) but not actually written to the network.

Long-Lived Cache. A caching mode wherein used data blocks are placed on a "potentially free" list for possible re-use. Because an image of a data block remains in cache memory even after the file is closed, it is possible to access that data again without the overhead of re-reading it into memory. This is true only if no changes have been made to the file since the last access.

Read-Ahead. A caching mode wherein, on a read request, an entire block of data (up to 4Kb) is read from the network rather than just reading the size requested. The assumption is that network access will be reduced because other nearby data (within the 4K block) is likely to be accessed next.

Read-Ahead mode is most efficient if the file is being accessed sequentially. Random accesses to large files causes performance to deteriorate, so Read-Ahead is turned off when random access is detected.

Redundant Write Removal. An algorithm whereby only the actual bytes that were modified are written to the network, rather than writing an entire cache block.

For example, a database record 6K in length is accessed and read into cache memory, and only one byte is modified, say a single digit of a phone number. Rather than write the entire 6K bytes back to the network, a comparison is done between the new and old record and only a single byte is written.

Short-Term Cache. A caching mode that is used when a request is made for a file that is small enough to fit entirely within cache memory.

Unique Write Mode. The mode wherein Redundant Write Removal is practiced.

Warehouse Cache. A caching mode that combines long-lived caching with caching on the local disk. This is used when local disk access is faster than the network link.

.

Write-Behind. A caching mode wherein data is written into a cached data block until an entire data block is filled, then the entire block is written at once. (As opposed to writing directly to the network in smaller chunks.)

Cases Where Data May Be Cached

Files that are being shared may not be cached, thus ensuring that no one gets bad data (for example, no one reads data that's being written-over by someone else).

Note: If a file is write-blocked (others may not write), the client may cache read operations.

If a file is read-blocked (others may not read), the client may cache write operations.

A client determines the access mode (read- or write-locks) of a file in one of two ways: by noting the mode in which the file was opened, or by using a bi-directional protocol.

This bi-directional protocol allows a client to cache data even though others are granted access to that data. As long as actual access has not occurred (that is, no one has read the file even though they could), the client may continue caching. As soon as the server notifies the client that actual access has been made to the file (someone opened it to read or write), the client flushes any dirty buffers (in the case of a write) or invalidates read buffers (in the case of a read) before continuing.

For a complete list of cases where caching can be used, see Appendix B of this chapter.

Determining Which Cache Mode to Use

There are three modes of file caching: the file being accessed is too big to fit into cache memory, the file is small enough to fit into cache memory, and the file is accessed after it is closed.

1. A file is larger than the memory allocated.

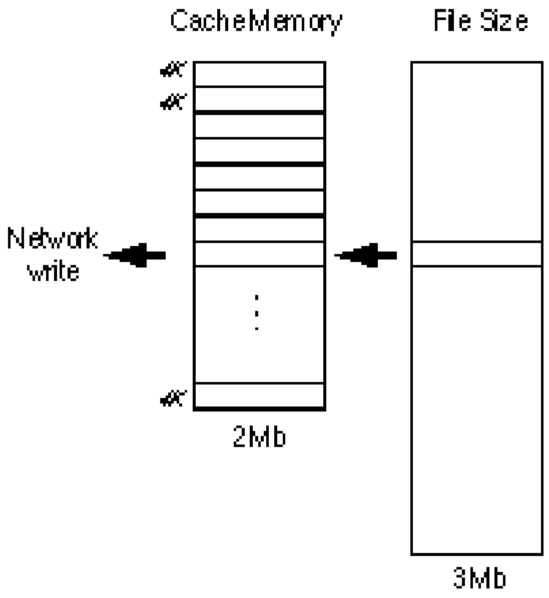

Figure 2. Read-Ahead/Write-Behind Mode.

In this case, the FileDir will use only the read-ahead/write-behind mode. That is, instead of attempting to read the entire file into cache memory (and wiping out everything else in cache memory), the file will be accessed only 4K at a time. When a read is finished, the cache buffer is re-used for the next read.

If, while in this mode, the file is accessed at random locations in the file, read-ahead will be turned off for this file. Random access is determined by having two (or some other configurable number) read cache misses in a row. (That is, the subsequent read is for a location outside of the 4K that has been read into memory.)

This method of spotting random access is only meant to weed out large database or index files which tend to be randomly accessed. It will still allow caching for multimedia files which, though very large, are generally "seeked" to a location and then read sequentially for awhile.

2. **The entire file fits into cache memory.**
   In this case, the entire file is read into cache memory. When a read is finished, the buffer is not re-used for the next read; rather, a new buffer is used for the next read, and so on until the entire file is in cache memory. This is called short-term cache.

3. **A file is accessed after it has been closed.**

   When a file is opened, the file system records the last time the file was modified. If the file is closed and re-opened, the file system compares the last-modified date of the file being opened with the information previously stored on the file.

   If the times are the same (that is, no modifications have been made to the file since the last time it was in cache memory), the file system uses the cache buffers already in memory. This is called long-lived cache.

Underlying Cache Design and Integrity Considerations

The boundaries (in the file) for cache block data start and end will always be on multiples of the cache block size. That means, for example, that the second cache block's data corresponds to offset 4096 to 8191 in the file. This helps maintain cache integrity and allows the caching code to run faster, with no size or location variables to consider.

An attempt to write data in a non-overlapping or non-contiguous fashion within the same cache block will cause any "dirty" data within that cache block to be flushed before the write actually takes place. This makes it so that the region between the non-sequential writes does not have to be read from the network to make a contiguous "dirty" region. This will not occur if the data between those regions has already been read. In this case, the whole spanning write regions would become the new "dirty" region.

This optimization will not occur if the spanning region is larger than the maximum packet size used by the conection, or if the spanning region is larger than the overhead of two separate write requests while on a slow link (less than 100 KBps).

Background Operation

The client cache supports a background operation that wakes up when an idle state is detected and will perform flushes of any "dirty" buffers and will read-ahead the rest of the files that are cacheable as short-term.

Because files sometimes are opened and closed frequently, the FileDir module supports an optimization where close requests are delayed for a short period, and if the file is opened again during that period it is reused without hitting the network. The delay period is specified in ticks and is configurable (Close Behind Ticks, see NET.CFG information).

File APIs

FILEAbort	Cleans up file entry without hitting network.
FILEBuildFIB	Returns FIB handle for specified file description
FILEClose	Closes specified file
FILECommit	Commits specified files dirty write buffers
FILEDup	Increments the count of times opened
FILEFindFIB	Finds fib handle by connHandle and 6-byte file handle
FILEGetDateTime	Returns file's date and time
FILEGetInfo	Returns information about a file handle
FILEGetSize	Returns the current file size
FILEOpenCreate	Opens or creates specified file
FILERead	Reads specified file from cache or network
FILERemoteCopy	Copies data from one file to another
FILESeek	Sets current file position
FILESetDateTime	Sets file's date and time
FILEWrite	Writes specified file to cache or network

Algorithms

Explanation of Need for True Commit Option.

The combination of the write-behind algorithm with auto-reconnection introduces a potential file integrity issue. The issue arises in this case:

1. The client writes to the server, and receives an acknowledgement of the write. But the server hasn't actually commited the write to disk, but is holding it in cache.

2. The server, unbeknownst to the client, crashes. The data in server cache is lost, but the client thinks data is in the file.

3. Auto-reconnection to the server occurs, and the client continues to write, not knowing that the file is corrupted.

The only way for a client to ensure that writes make it to the disk is to set the True Commit option. This ensures file integrity across a crash, but is disk-intensive if several workstations are writing files. The background write feature of the FileDir module helps performance for this situation.

By default, written files are not auto-reconnected so that, also by default, "True Commit" need not be on.

DIR Functions

The functions that return directory information also use caching to improve performance. When a search context is opened, a check is made to see if the contents of the cache for that context are still synchronized with the server. If they are, requests are serviced from the cache; otherwise, the context is treated as the first time used and directory entry information is placed in the cache as the entries are enumerated.

Directory APIs

DIRGetAttributes	Gets entry's attributes
DIRSetAttributes	Sets entry's attributes
DIRRename	Renames specified directory entry
DIRMakeDirectory	Creates the specified directory
DIRDelete	Deletes specified directory entry
DIROpenSearch	Returns search context for specified filespec
DIRNextSearch	Returns directory entry information
DIRCloseSearch	Closes specified search context
DIRRedoSearch	Reinitializes search context to new attributes
DIRSetDirectory	Sets directory handle to new path
DIRGetDirectory	Returns UNC path for a dir handle and relative path
DIRAllocDirHandle	Allocates a directory handle to specified path
DIRFreeDirHandle	Deallocates specified directory handle
DIR32To8Bit	Converts a 32-bit directory handle to 8-bit
DIR8To32Bit	Converts an 8-bit directory handle to 32-bit
DIRGetDirectorySpace	Gets directory space information
DIRGetVolumeInfo	Gets volume information
DIRGetVolumeName	Gets volume name
DIRGetVolumeID	Gets volume ID
DIRDup	Returns a duplicate of the specified dir handle
DIREnumerateDirs	Returns directory mapping information

SYNC Functions

Synchronization functions have two general types (see table below): record and semaphore. The record functions in turn have two types: physical and logical. Physical records deal with portions of files or whole files and logical records deal with application-defined names associated with resources (for example, a CD-ROM on a server).

Synchronization Functions		
Record		Semaphore
Physical	Logical	

Physical-record locks on a file, if that file is opened exclusively, will be cached since the file is already protected by the exclusive open. If that file's access status changes, by a bi-directional NCP perhaps, then those physical locks will be issued without application intervention.

Semaphores

Semaphores are like logical record locks in the sense that they are associated with network resources such as files, records, structures, or hardware. Logical record locks limit the number of applications that can simultaneously access the resource to one. Semaphores, however, allow a configurable number of applications to access a network resource at one time.

When an application creates a semaphore, the application assigns a value to the semaphore (for example, 4). The value indicates how many applications can access the resource associated with the semaphore at one time. In the example, five applications can access the resource at one time (0 to 4).

After opening an existing semaphore, an application first checks the value using **SYNCExamineSemaphore**. If the value is greater than or equal to zero, the application can access the associated network resource. The application decrements the value by calling **SYNCWaitOnSemaphore** and then accesses the resource. When the application finishes accessing the resource, the application increments the semaphore value by calling

SYNCSignalSemaphore, and then **SYNCExitSemaphore**.

If, after opening a semaphore, an application discovers that the value is negative, the application cannot access the resource immediately. The application can either wait a specified timeout interval until the resource becomes accessible, or the application can retry later.

The currentOpenCount returned by **SYNCExamineSemaphore** indicates the number of processes using the semaphore. **SYNCOpenSemaphore** increments that count and **SYNCCloseSemaphore** decrements it.

The following algorithm illustrates semaphore usage:

```
SYNCOpenSemaphore ()
SYNCExamineSemaphore ()
If (semaphoreValue >= 0) {
    If ((SYNCWaitOnSemaphore ()) == 0) {
        Access the associated resource
        SYNCSignalSemaphore ()
    }
}
SYNCCloseSemaphore
```

Synchronization APIs

SYNCFileName	Provides services for file-based semaphores
SYNCFileSet	Sets active all logged file-based semaphores
SYNCPhysRecord	Provides services for file region-based semaphores
SYNCPhysRecordSet	Sets active all logged file region-based semaphores
SYNCLogicalRecord	Provides services for string-based semaphores
SYNCLogicalRecordSet	Sets active all logged string-based semaphores
SYNCOpenSemaphore	Opens or creates a named semaphore
SYNCExamineSemaphore	Examines the current count of a semaphore
SYNCWaitOnSemaphore	Waits on the semaphore for the specified timeout
SYNCSignalSemaphore	Signals the specified semaphore
SYNCCloseSemaphore	Closes the specified semaphore

Events Generated by FileDir

The following events are produced by FileDir.:

DriveCreatedEvent	Produced when a directory handle has been allocated. This event passes a pointer to the *DriveUpdateEvent* structure.
DriveChangedEvent	Produced when a directory handle has been changed to a new location. This event passes a pointer to the *DriveUpdateEvent* structure.
DriveDestroyedEvent	Produced when a directory handle has been destroyed. This event passes a pointer to the *DriveUpdateEvent* structure.
FileAbortEvent	Produced when a file is aborted with FILEAbort or an auto-reconnection failure.

NET.CFG Parameters

The following FileDir parameters are configurable by the user at load- time in the NET.CFG file:

Load-time configurable parameters

Parameter	Default	Range
Cache blocks	16	no limit
FILE Cache Level 0: Disabled 1: Read-ahead and write-behind only 2: Short-lived caching (don't cache beyond closes) 3: Long-lived caching 4: Warehouse caching	3	4
Checksum Cache	OFF	

Run-time configurable parameters

Parameter	Default	Range
BACKGROUND_WAKE_UP_TIME	5000 ms	56-64Kms
BACKGROUND_INTER_WORK_TIME	56ms	56-64K ms
CACHE_READ_MISS_LIMIT	2	1-4 billion
Read Only Compatibility	ON	
Redundant Write Removal	ON	
True Commit	OFF	
Auto-reconnect level	2	0-3
Close-Behind ticks	36	0-64K
Cache writes	ON	
Delay writes	ON	

Deliverables Information

Executables	FILEDIR.NLM
Product Source	FILECORE.C, FILECORE.H
	DIRCORE.C, DIRCORE.H
	SYNC.C, SYNC.H
	API.H
	INTERNAL.H

Unit Tests

DEV.NLM	Tests all of the Device registry functions.
DIR8-32.NLM	Tests the 8-to-32 and 32-to-8 dir handle conversion functions
FC.NLM	Tests drive alloc, make, change, get, rename, remove directory, get attributes, set attributes, (search init, continue, redo, close), drive free functions.
FIO.NLM	Tests file open, create, read, write, close, get info, seek, flush, and delete functions.
SHOWDIR.NLM	Tests the DIREnumerate and DIRGetDirectory functions.
SYNCTEST.NLM	Tests the all syncronization functions.
TN.NLM	Tests the path and name parsing routines.