Chapter 2

The Local File System

In this chapter we discuss the local disc file system in general terms. This is but one of the possible file systems which can coexist in any given machine,[1] though it is the primary one as far as the file servers are concerned. All these file systems present a common, message-based, interface to the user-level run-time support; this is discussed in section 2.4. From the point of view of the file system, a remote client accessing files *via* some file access protocol with a local interpreter has exactly the same status as a co-resident client whose run-time support is speaking directly to the various local file systems.

The local disc file system is constructed in two main layers, with a packaging layer on top to stitch everything together. The lowest of these layers is the file system proper which manages the disc space and performs access control on the files held thereon. Each file is identified at this level only by a 30-bit "file-ID." The facilities provided include the ability to create and delete files, to open and close them, and to read and write their data blocks.

The middle layer is responsible for maintaining the system's hierarchic directory structure. At this level the structure applies strictly to files in the local file system, though the addition of "redirectors," which are interpreted by the client, can extend the user's preception of the tree to encompass a number of file systems and, indeed, autonomous servers. A directory is considered to be merely a list of names and corresponding file-IDs. User filenames are translated by considering each component in turn and, starting at the root of the directory tree, searching for the next component in the directory whose ID has been found by translating all the previous path components in turn. The directory layer uses the "spare" two bits in the file-ID for its own purposes.

The topmost packaging layer essentially takes each user-request and performs the necessary sequence of directory and file-system operations. In the simplest cases, such as writing a block of data, these may translate one-for-one, while creating, renaming or copying, for example, will each involve several more primitive operations.

2.1 The File System proper

The file system layer is responsible for disc layout management and file protection enforcement. The facilities exported are summarised in figure 2.1 on page 5. Each pro-

[1] The others currently comprise the "special" filesystem, an "old-style" importer and an NFS importer.

```
%integerfn fsys open file  (%record(fsys access fm)%name access,
                            %integer ID, mode, compatible,
                            %integername token, size, flags)
%integerfn fsys close file (%record(fsys access fm)%name access,
                            %integer token, flags)
%integerfn fsys read file block %c
                           (%record(fsys access fm)%name access,
                            %integer token, block,
                            %integername bytes,
                            %record(*)%name buffer)
%integerfn fsys write file block %c
                           (%record(fsys access fm)%name access,
                            %integer token, block, bytes,
                            %record(*)%name buffer)
%integerfn fsys truncate open file %c
                           (%record(fsys access fm)%name access,
                            %integer token, bytes)
%integerfn fsys create file(%record(fsys access fm)%name access,
                            %string(255) creation name,
                            %integer pn, benefactor ID,
                            %integer initial allocation,
                            %integername ID)
%integerfn fsys bump refcount %c
                           (%record(fsys access fm)%name access,
                            %integer ID, increment)
%integerfn fsys obtain attributes %c
                           (%record(fsys access fm)%name access,
                            %integer file ID,
                            %record(attributes list fm)%name a)
%integerfn fsys modify attributes %c
                           (%record(fsys access fm)%name access,
                            %integer file ID,
                            %record(attributes list fm)%name a)
%integerfn fsys exchange   (%record(fsys access fm)%name access,
                            %integer ID1, ID2)
```

Figure 2.1: Exported file system operations

semaphore control. Each chunk slot has an associated status word, with a wait queue for those processes attempting to access a chunk while it is still in transit from the disc.

2.1.2 File-Structured Partitions

The key to a file-structured partition is its collection of file headers. These are grouped together into an "index file," the first block of which contains the header for the index file itself. The 24-bit part of the file ID which identifies the file within partition is subdivided into two: the low order 16 bits are used as to select the block within the index file containing the header; while the high order eight bits are used as a sequence number to catch "dangling" directory entries, being incremented by one, independently for each index file slot, each time a new file is created. The format of the file header is shown in figure 2.2 on page 8. Each header is protected by a checksum, intended mostly to catch software bugs and occasional hardware errors as the disc controllers' ECC algorithms are considerably more powerful. No transaction mechanism is provided for file headers; instead the file system is careful not to start any operation unless it intends completing it.

A file's ID is stored in its header: if, on a header access, this is found not to match the ID specified, this is assumed to be due to a dangling directory entry, and a "no such file" error is returned. A slot-part of zero in the stored file-ID indicates that the header is free for reallocation.

The file system maintains a cache of file headers to improve performance, in conjunction with the concurrent access code. Only one copy of any header is in the cache, shared amongst the various users who may have the file open. When a user opens a file, both the requested access mode and also a list of modes which the user is prepared to allow to other shared users of the file are supplied. These are checked against the modes specified by any other users who might already have the file open, and access is granted only if there are no conflicts. File access requests which are blocked by this test are not queued by the file system, but instead result in an immediate error reponse. "Control" access, allowing modification of a file's attributes, is always implicitly enabled for all files, except for the (short) duration of an "exchange" operation. This file-grained locking scheme is not discretionary: it is enforced for all file access requests. The header-cache replacement strategy is LRU, though only those headers which are not currently in use are candidates for replacement. Semaphores are used to maintain integrity and atomicity: each header is protected by an associated semaphore, while one common semaphore interlocks access to the global tables. In order to avoid deadlocks, no attempt is ever made to claim more than one of these semaphores at a time.

An extent-based allocation scheme is used; assuming that the partition is not too badly fragmented, this will result in a more compact representation than would be the case if each block of the file had its own pointer. Extent slots are allocated from the end of the header, growing back towards the start. Not all the allocated blocks need necessarily be used: indeed, as a file is extended new blocks will be allocated several at a time though only one will be written with each request. For security reasons, users are prevented from reading allocated blocks which they have not previously written. If possible, the file system will try to allocated new blocks contiguously with those in the final extent, and if successful will merely update the extent record to incorporate the newly-allocated blocks rather than allocating a new extent slot for them. Any

unused blocks are, optionally, released when a file is closed. A cache prefetch is initiated whenever a read access pattern is noted to be sequential and the block just transferred was either the last block of the current extent or the last block of the cache chunk, as signalled by the status return from the read operation.

The "exchange" operation is performed by copying the extent records between the two files, using an additional "anonymous" file header as an intermediary. This operation is guaranteed to be atomic and risk-free, and is considerably more efficient than would be the copying of large numbers of data blocks. Both files must reside on the same partition, of course. The operation is performed as follows: the intermediary file, I say, is created (but with no blocks allocated); the extent records of A are copied to I's header which is then flushed to disc; B's extent records are copied to A's header which then is flushed to disc; I's extent records, originally from A, are copied to B's header and it is flushed to disc; and finally the intermediary I is deleted. Note that the data content of both A and B will always be preserved by this sequence of operations; even in the unlikely event of a system failure partway through, the system manager can find I using the lost-files utility and either complete the exchange or delete the intermediary as required.

Timestamps note when a file was created, when it was last modified, and, if enabled for the partition, approximately when the file was last accessed. This last timestamp, which could be used by an archiving system or to delete infrequently-used files, is updated no more than one every 20 minutes or so, in order to minimise "unnecessary" disc traffic for heavily-used system utilities.

2.1.3 Access Control

Access control is based on 32-bit tokens: at the level of the file system there is no other form of user identification. These tokens are treated purely as bit-patterns, with any conventional structure imposed by the system manager being for higher-level consumption only. Each user possesses a "user-ID" token, which would normally be unique but need not necessarily be so; in addition, each user may be granted the right to assert a number of additional tokens, referred to as "group-IDs" in recognition of their conventional purpose. When a file is accessed, the user's list of tokens is compared against the list associated with the file, the resulting access permission being the union of those permissions associated with tokens which matched. For the moment we assume merely that the appropriate tokens and privileges are being asserted: we postpone any discussion as to how this is done to later (section 2.5).

The tokens associated with a file fall into four categories:

- The "world" token, implicitly granted to all users, with its associated access rights (`world access` in figure 2.2).
- The "local" token, implicitly granted to all co-resident users, with associated access rights (`local access`).
- A variable number of "group" access records (the array `access`), growing from the front of the file header towards the extent list, with associated access rights.
- The "owner" token, the "supervisor" token and the "creator" token, with `owner` access plus implicit "control" access.

requested number of blocks out of the first free area which is big enough are allocated; while if there are no free areas of the requested size, the largest is allocated.

The bitmaps are not stored on disc; rather they are built from scratch each time the file system is initialised. This simplifies manipulation, removes one possible source of inconsistency, and, incidentally, would make it easier to replace the entire free space manager with one using a different philosophy such as an explicit free list.

2.1.5 Quotas

These have not been implemented yet. The intention at the moment is that each partition will have quotas enforced separately.

2.1.6 The Disc Drivers

Though not strictly part of the file system, the disc drivers are described here since the (fixed) discs are universally accessed through one or more file system partitions. Although all drivers present a common interface, each is required at present to "know" the geometry of the drives attached to its controller. Thus each system configured has a slightly different version of the driver tables built in. This slightly unsatisfactory situation has arisen because the only real choice for on-disc geometry tables, *viz* sector zero of track zero of cylinder zero, or at any rate one of the low-numbered sectors, had already been allocated to the processor's primary boot firmware; it may be changed if it proves to be sufficiently annoying. Two interfaces and four drive types are currently supported: "standard" SMD, using a NEC controller chip, with Fujitsu 2284 and 2294 drives; and ST506, using an Ambit Pace controller board, with Fujitsu 2243 and Rodime 204E drives. Any other type of drive could easily be configured.

Disc read and write request messages are ordered by the driver process according to disc address, and are scheduled using an "elevator" algorithm. Transfers can be of any size, though the partition module will not at present ask to read more than eight blocks at a time, nor write more than one. The driver and controller co-operate to continue transfers across track and cylinder boundaries as required. Verification can be enabled for the SMD driver, independently for read and write transfers: only write verification is currently enabled.

2.2 Directories

The directory layer is responsible for maintaining the correspondence between user-supplied filenames and filesystem-generated file-IDs. It presents the user with a fully hierarchic intra-filesystem directory structure, and provides "redirectors" which can be acted upon by the user's client system to unify the view of the various servers' directory structures. The B-Tree module described in section 2.3 below is used to provide key management and data storage facilities. Individual filename components can be up to 127 characters in length; case is preserved but ignored. The interface to the directory module is shown in figure 2.3 on page 12.

Filename paths are passed to the directory layer as a linked list, already split into their constituent components by the user's run-time support package. This has a two-fold benefit: the maximal number of path components is not constrained by any limi-

tation on the size of any buffer holding the whole, unsplit, filename; and the choice of meta-character to use as path separator can be made to suit each run-time environment individually.

Multiple versions of data files are permitted. These are numbered in relative terms, rather than absolute, with the most recent being version zero, the next being version -1, then -2 and so on. Directories and redirectors are constrained to exist in only one version; hence the version number of only the final path component need be considered. At present there is no automatic version limit mechanism: instead the directory module simply refuses to allow a version beyond a fixed reasonably large limit to be created. This may be changed in future releases.

The directory layer layer uses the two "spare" file-ID bits for its own purposes: bit-31 indicates whether a key (filename) translates to a single file-ID, in which case the translation can be used directly, or whether it translates to multiple versions or a redirector, in which case the translation is presented as a token to the B-Tree data storage section; while bit-30 is used to indicate that the file-ID corresponds to a directory (this latter is also known to the packaging layer, which treats attempts to read or write directories as special cases).

Three separate lookup procedures are provided for the benefit of the packaging layer. The basic primitive is `directory lookup one`, which attempts to translate the key supplied in the indicated directory. A successful translation as a file-ID results in a zero status, a translation as a redirector results in a positive status, while failure is indicated by a negative status. The remaining two procedures call the first repeatedly for each element in the path list, starting with a known directory-ID (the "root" directory) and using the result of one step as the input directory-ID for the next. The operation is terminated should any non-zero status be returned, with the status, the textual translation (if any) and the number of successful translations being passed back to the caller. Note that redirectors are not processed in the directory layer, but instead are passed back to the run-time support to be dealt with. The reason for having three different "lookup" procedures is that this makes the operation of the packaging layer considerably clearer.

As well as file-IDs, which if they have the same key as an already-existing entry are inserted as the most recent version, there are two forms of textual redirectors, *viz* `internal` and `external`. These are treated identically by the directory layer but result in a different status value being passed back as the result of a translation, allowing the higher layers to handle them differently. One common entry-deletion procedure is provided, the the type of entry to delete being known from the directory itself. The insertion or deletion of a file-ID entry results in a suitable adjustment of the corresponding file header reference count, with the file being automatically deleted by the file system if the reference count goes to zero (i.e. when all paths to the file have been removed).

Directories are just the same as other files as far as the file system is concerned, with no special deletion procedure being required. However, the packaging layer knows about the "directory" bit in the file-ID, and will refuse to delete a directory unless it has been confirmed to be empty.

A list giving the contents of a directory can be obtained using the `directory contents` procedure. This is a rather *ad hoc* mechanism, though it does have the virtue that the time that the directory is required to be open is minimised.

In order to speed up the translation of commonly-used filenames, the directory layer maintains a cache of recently-used names. This is organised as a number of directory

```
%integerfn B tree open by ID  (%record(fsys access fm)%name access,
                               %integer ID, mode,
                               %integername access token, flags)
%integerfn B tree close       (%integer access token, abandon)
%integerfn B tree create      (%record(fsys access fm)%name access,
                               %string(31) name,
                               %integer partition, benefactor ID,
                               %integername ID)

%integerfn B tree add entry   (%integer access token,
                               %string(255) key, %integer data)
%integerfn B tree find entry  (%integer access token,
                               %string(255) key, %integername data)
%integerfn B tree delete entry(%integer access token,
                               %string(255) key)
%integerfn B tree modify entry(%integer access token,
                               %string(255) key, %integer data)

%predicate B tree empty       (%integer access token)

%integerfn B tree data value  (%integer access token, site,
                               %integername size,
                               %record(*)%name target)
%integerfn B tree data replace(%integer access token, site,
                               %record(*)%name source)
%integerfn B tree data insert (%integer access token, size,
                               %record(*)%name source,
                               %integername site)
%integerfn B tree data delete (%integer access token, site)

%recordformat key list fm(%record(key list fm)%name next,
                          %integer value, %string(255) key)
%record(key list fm)%map B tree key list(%integer access token,
                                         %integername status)
```

Figure 2.4: The B-Tree package interface

understand the request to part-translate the path, in this case probably returning a redirector pointing to another file system. This allows non-standard operations to take place through the normal global directory structure.

- user identification will be as described in section 2.5.
- Each request will contain a 32-bit identification tag, allowing the client to distinguish between responses to several concurrent requests.
- Responses will contain a standard status value, a file system specific status code, and a textual message containing either an error message or the data corresponding to a redirector.

Standard request codes will include the following: open file; read data; write data; close file; truncate file; make accessible (to unlock files to which the file system has blocked access because they were not closed cleanly); create directory; remove directory entry; rename file. Non-standard codes recognised by the local packaging layer might include: insert local redirector; insert external redirector.

2.5 User validation

At present user validation is performed either by the protocol interpreters or by a co-resident client's run-time support. This is clearly wrong for two reasons: it requires that the client (or protocol interpreter) know the correct form of credentials to present to whichever file system it happend to be talking to; and it trusts the client not to misrepresent itself. The approach which will shortly be implemented recognises that there are two aspects to the problem, *viz* determining a user's identity and determining a user's access rights and privileges.

User identification will be the responsibility of an identification manager. If there is a local database, the manager will issue tokens in return for a correct username-password pair. On request the manager will validata any token presented to it, and if acceptable will return the identity of the corresponding user. The local manager may, if it chooses, accept the word of a remote manager. Within an administrative domain, such as the Department, the major file servers and multi-access machines would be set up to trust each other; hence users would only require to log on once in order to have all their files accessible, whichever server happened to hold them.

Access rights and privileges will be the responsibility of the server processing the request; in the case of the local file system, the packaging layer will take the user's identity, as supplied by the identification manager, and use that to determine the appropriate user and group IDs and privileges to assert. Initially there will be one database covering the whole file system; later it might be useful to allow supplementary databases for individual partitions, particularly in the case where a disc has been moved from one machine to another. Default access rights will apply should the username not be in the local database.

[13] J. Postel. *Transmission Control Protocol.* RFC 793, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291, September 1981.

[14] J. Postel. *User Datagram Protocol.* RFC 768, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291, August 1980.

[15] G. D. M. Ross. *The New Filestores.* Technical Report, Computer Science Department, University of Edinburgh, 1984.

[16] G. D. M. Ross. *Virtual Files: a Framework for Experimental Design.* PhD thesis, University of Edinburgh, 1983. Available as Technical Report CST-26-83.