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.

cedure has as one of its parameters an access rights record, detailing the caller's ID, privileges and group memberships, as described in section 2.1.3 below.

The basic sequence of operations is to open a file, to read and/or write one or more blocks, and then to close it again. This stateful approach was chosen over the alternative stateless method whereby the file's ID would be specified with each read or write request, because it reduces the cost for these data-transfer operations, because it allows for files to be opened by suitably endowed agents on behalf of other less-well endowed agents, and because it allows the file system to enforce file-grained concurrency control. This latter requirement implies that some state must be maintained somewhere, and the file system, which is in any case responsible for enforcing access control, is the obvious place to put it. Supported access modes are `read`, `modify` which allows any block of the file to be written, and append, enforced here but implemented mainly in the packaging layer, which allows new data to be written beyond the end of the current file but not the alteration of existing data.

When a file is opened, the file system returns a token for the file, the size of the file (in bytes) and a flags word. The flags include some from the header, such as whether the file has been improperly closed or is marked for backup, and some which are generated, such as whether the file has world-read access (because access control is enforced by the file system, any higher-level caches must either be user-specific, or contain only world-readable information).

Reading and writing blocks require only that the token issued when the file was opened be presented along with a data buffer. Files are extended automatically by writing, one block at a time, beyond the area which already contains data. Only the last block written may be short; an error status will be returned should a short write to any other block of the file be attempted.

Closing a file invalidates the token supplied when it was opened and removes any concurrency constraints. Optionally, the allocated space may be truncated to exactly that required to hold the data content.

Files grow automatically as they are written. In order that they may be shortened, a procedure is provided to truncate a currently-open file. The new size is specified in bytes, and any data which existed beyond that size becomes inaccessible. For reasons of security the converse operation is not supported.

A separate procedure for creating files is provided, as this was felt to result in a cleaner division of the file system's tasks than would automatic creation in the "open" procedure. The caller specifies a creation name, which is recorded (possibly truncated) in the file header for the benefit of file system structure management utilities, a partition number, the initial space allocation (note that these blocks are not accessible until written) and the file-ID of a "benefactor." This last is used to supply defaults for all ownership and access fields, with the exception of the "creator" which is taken from the supplied access record. The purpose of these ownership and access fields is explained in section 2.1.3.

Each file has a reference count associated with it which is incremented or decremented in steps of one at the request of the directory layer. Files are automatically deleted when their reference count goes to zero, implying that they are no longer needed.

The procedures for querying or modifying a file's miscellaneous attributes (basically, everything except data content) take linked lists of attribute records, each of which contains an attribute code, a status, and pointers to one or two buffers. These are

```
%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

processed, independently of each other, in the order in which they appear in the list, the entire request being an atomic operation on the file. Files are referred to directly by ID, rather than being required first to be opened, as attribute operations are generally of a one-shot nature and do not require to be interlocked with other concurrent users.

The "exchange" operation is provided for the benefit of applications implementing transactions by mechanisms such as differential files. It provides a means whereby the data content of two files can be logically interchanged as an atomic operation. The operation is performed "carefully" in the sense that in the unlikely event of a total system failure partway through, the data content of both of the files is guaranteed to be preserved, albeit perhaps not with the intended IDs; the critical window is of the order of milliseconds, however.

2.1.1 Logical Partitions

The file system does not, in fact, deal directly with raw discs. Instead, each disc is divided up into one or more logical partitions. There are a number of advantages in this approach: these include the ability to mix both file-structured and non-file-structured areas, such as the disc header (describing the partition layout on the disc) and the bootstrap area; the ability to influence disc layout on a coarse scale, for example by placing commonly-used system utilities in a partition near the centre of the disc; the provision of coarse-grained access control; and independent quota enforcement.

Each 30-bit file-ID is divided into two parts: the high-order six bits are used to identify the logical partition, while, in the case of file-structured partitions, the low-order 24 bits identify the file within the partition. The partition identification part is currently further sub-divided into two equal parts, the high order three bits identifying the disc, and the low-order three bits identifying the partition within the disc. The partition module provides a defaulting service to the directory layer, with procedures[2] to supply missing partition and disc numbers for one file-ID based on those of a (fully specified) second file-ID, and conversely to strip out the partition and/or disc numbers from a given file-ID if they correspond with those of a second. The directory layer uses this mechanism to make files' directory entries relative to their respective parent directories, allowing directory trees to be more loosely coupled with discs and partitions.

The interface between the partition sub-layer and the file system above it takes the form of block-read and block-write requests, each of which specify the partition ID and the block within the partition.

As well as controlling coarse-grain disc layout, the partition module maintains a cache of disc blocks. This is organised in "chunks," currently of eight blocks each, read from the disc in a single operation. Writes are done one block at a time, first to the disc and then copied into the appropriate cache chunk as required: this approach is sufficient, in the sense that it does not make any concurrent read from the same block any less risky. Note that it is not safe to attempt the cache update before writing to disc, as the disc driver is at liberty to schedule any subsequent chunk read request before the write, giving rise to cache inconsistency. The cache replacement strategy is LRU. A prefetch request entry is provided, together with an indication to the caller whenever a read is from the last block in a chunk. With a $\frac{1}{2}$-Megabyte cache, the hit rate is typically over 90%, and often in excess of 95%. Concurrent access to the cache tables is under

[2] At least it should be done like this!

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

```
%constinteger no       access =  0  { File is inaccessible
%constinteger read     access =  1  { File can be read
%constinteger modify   access =  2  { File can be modified
%constinteger append   access =  4  { File can be appended to
%constinteger exchange access =  8  { File can be (extent) exchanged
%constinteger link     access = 16  { File can be (un)linked
%constinteger control  access = 32  { File attributes can be modified
%constinteger deny     access = 64  { Invert sense of access bits

%recordformat header access fm(%integer ID, access)

%recordformat extent fm(%integer start, size)

%constinteger non extent size = 8 + 2 + 2 + 12 + 8 + %c
                                  12 + 12 + 16 + 4 + 2 + 2

%constinteger extent limit = (512 - non extent size) // 8
%constinteger access table size = extent limit

%recordformat file header fm((%integer checksum, ID,
                              %short header refcount,
                              %short flags,
                              %integer owner, owner access, supervisor,
                              %integer world access, local access,
                              %integer creator, static ID, audit ID,
                              %integer created, modified, accessed,
                              %string(15) creation name,
                              %integer blocks used,
                              %short bytes in last block,
                              %short extent limit,
                              ( %record(header access fm)%array %c
                                    access(1 : access table size) %c
                            %or %record(extent fm)%array %c
                                    extent(1 : extent limit)) %c
                             ) %or %integerarray x(1 : 128))
```

Figure 2.2: File Header Format

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.

When a file is created it normally inherits its access control data, with the exception of the creator-ID, from a "benefactor," chosen by the packaging layer to be either a previous version of the file, if one exists, or the parent directory in which the file is being created.

Two ownership tokens are maintained for each file in order that that both the creator of the file and the owner of the directory (or the previous version of the file, should this be different for some reason) maintain full access rights. The use of only one token would inevitably have led to one or other being disenfranchised.

The "supervisor" token is intended as an aid to class management. The members of a particular class would be assigned to some group, with rights to the token being granted to the lecturers, tutors and demonstrators responsible for that class. This would give them full owner-access rights to the root directory of the tree and any files created as part of that tree, including the implicit right to alter a file's protection attributes. In effect, they have equal power over students' files as do the students themselves. Note that students would inherit ownership of, and hence full access rights to, any files which a supervisor might create in their directories.

In addition to the right to assert a number of access tokens, each user may also be granted various privileges. Those relevant to the file system are:

readall allowing the user to read any file, irrespective of the protection set;

bypass which, as its name suggests, causes the protection checking mechanism to be bypassed completely; and

bootarea which allows the holder to access unstructured partitions, and in particular the system's on-disc bootstrap area.

2.1.4 Free Space Management

Free space management is performed using bitmaps, one for each partition, though as they are manipulated exclusively through a procedural interface it would be straightforward to implement some other scheme. The operations allowed are:

- to claim a specified range of blocks, returning an error if any was already allocated;
- to release a specified range of blocks, returning an error if any was not allocated; and
- to allocate a contiguous range of blocks, guided by a desired size and a suggested starting position.

Bitmaps are scanned 32-bits at a time whenever possible, starting either at the suggested location, wrapping round if necessary, or, if none is specified, from some (prime) number of blocks beyond the end of the area allocated in the previous call. The intention behind this algorithm is to make it more likely that files can be extended contiguously, for efficiency of access and to reduce the number of extent records required to describe them, the hope being that the hole which was left can be allocated at the time the file is next extended. The prime step-size should be less likely to interact constructively with the bitmap size. If contiguous space is available then it is allocated, even if it is smaller than the requested size; if no contiguous space is available, then the 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-

```
%recordformat path fm(%record(path fm)%name next, %integer version,
                      %string(*)%name key, %string(255) text)

%integerfn directory lookup one %c
                                (%record(fsys access fm)%name access,
                                 %integer directory ID,
                                 %string(*)%name key,
                                 %integer version,
                                 %integername resulting ID,
                                 %string(*)%name textual translation)
%record(path fm)%map directory penultimate %c
                                (%record(fsys access fm)%name access,
                                 %record(path fm)%name path,
                                 %integername components translated,
                                 %integername resulting ID,
                                 %string(*)%name textual translation,
                                 %integername status)
%integerfn directory lookup %c
                                (%record(fsys access fm)%name access,
                                 %record(path fm)%name path,
                                 %integername components translated,
                                 %integername file ID, penultimate ID,
                                 %string(*)%name textual translation)
%integerfn directory insert ID %c
                                (%record(fsys access fm)%name access,
                                 %integer directory ID,
                                 %string(*)%name inserting key,
                                 %integer inserting ID)
%integerfn directory insert local %c
                                (%record(fsys access fm)%name access,
                                 %integer directory ID,
                                 %string(*)%name inserting key,
                                 %string(*)%name inserting text)
%integerfn directory insert external %c
                                (%record(fsys access fm)%name access,
                                 %integer directory ID,
                                 %string(*)%name inserting key,
                                 %string(*)%name inserting text)
%integerfn directory delete entry %c
                                (%record(fsys access fm)%name access,
                                 %integer directory ID,
                                 %string(255) key, %integer version)
%integerfn directory check empty %c
                                (%record(fsys access fm)%name access,
                                 %integer directory ID)
%record(*)%map directory contents %c
                                (%record(fsys access fm)%name access,
                                 %integer directory ID,
                                 %integername status, flags)
```

Figure 2.3: The interface to the directory module

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 slots, identified by file-ID, each containing name/translation pairs. Directory slot reuse is controlled by a LRU algorithm, while within slots the entries at present remain permanently allocated. Because the directory layer sits at a higher level than that at which file protection is enforced, the cache mechanisms must take care not to compromise file system security. This is achieved by making a cache entry only where the directory concerned is world-readable.

2.3 The B-Tree Module

As has already been mentioned, the directory layer makes use of a separate B-Tree package to perform key management and data storage (the interface is shown in figure 2.4 on page 15). The package, which is also used by other server components such as the authorisation manager, has three parts:

- the I/O interface,
- the key manager, and
- the data record manager.

As well as interfacing to the file system, the I/O section implements transactions using a form of virtual file [16,11,1,9,3,4]. Two two-level indirectory maps are held in the file along with the tree and data blocks. The one-block roots of the maps, which are stamped with an epoch number to show which is the most recent, are in blocks 0 and 1 of the file. The high-order bits of a virtual block number are used to index into the current root block; the resulting entry is the physical block number of the second-level map block; the physical block number corresponding to the original virtual block number is obtained by using the low-order bits of the latter as an index into the former. A transaction is opened by selecting the root block with the more recent epoch number, the other one being ignored meantime. Blocks are read using the current indirectory map. When a (virtual) block is written for the first time a new physical block is selected and the second-level map block is updated; if this was the first time that map block was updated then a new physical block will be chosen for it and the root block will be updated accordingly. The transaction is committed by flushing the map cache, then incrementing the epoch number and writing the root back to the *other* root block site; it is abandoned simply by closing the file without updating the root. Although a small degree of complication is added by this process it is more than made up for by the considerable simplification in the logic of the rest of the B-Tree module and the other system components which make use of it.

The key management section maintains a B-Tree of variable-size keys and corresponding 32-bit data. Insertions follow the usual splitting algorithm. Deletions are complicated slightly by the variable key-size: a non-leaf key deletion requires that the next-highest key be borrowed from the appropriate leaf, possibly resulting in splitting if the borrowed key is larger than that being deleted; the shape of the tree is then readjusted, starting at the leaf from which the key was borrowed. Readjustment, which is carried out only if the node is less than half full, takes the form first of an attempt to merge adjacent blocks, right or left merging being chosen according to which gives the best use of space; if merging is not possible an attempt is made to rotate one or

```
%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

more keys left or right, again depending on use of space; if neither merging nor rotation was possible the attempt is abandoned, the assumption being that some subsequent readjustment will recover the wasted space. Note that empty blocks will never arise, as it would always be possible to merge or rotate in this case. No attempt is made to carry out tree manipulation operations in a "safe" manner, as the transaction system in the I/O interface allows any alterations to be backed out if required.

The data storage section provides a means whereby variable-size records may be stored in the same database file as the keys. When a record is inserted a token is returned to the caller (usually to be stored itself as the datum corresponding to some key). The record can subsequently be read, modified (provided its size is not changed) or deleted on presentation of the token. A size change requires that the record be deleted and reinserted, probably resulting in a different token being returned.

Both the key management and the data storage sections are self-contained units, and could easily be packaged up in an I/O section suitable for a user application. Indeed, the only changes required to the current I/O section would be to open the database file by name rather than by file-ID, and to adapt the block-read, block-write and close calls appropriately.

2.4 User Requests and the Packaging Layer

The packaging layer is responsible for taking user requests in the standard form and performing the appropriate sequence of file and directory operations. These operations may map directly, may involve several more primitive operations, may be ignored or may be rejected. Requests are serviced by a pool of procecsses, sharing access to common tables and all waiting on a single request queue. Each request is dealt with in its entirety by whichever process happens to have picked it up before the process returns to the tail of the queue to wait its turn for the next request.

Note that the packaging layer does not interpret redirectors; instead it passes the resulting text back to the user-level run-time support together with an indication as to the type of the redirector. A co-resident client would probably want to interpret both local and external redirectors, the former resulting in a new request to the local file system, with the latter resulting in a request to a different file system. A protocol interpreter acting on behalf of an external client, however, would only interpret local redirectors, returning external redirectors to the client for further processing. This means that the remote client can obtain the same view of the global directory structure as would a co-resident client without the need for the protocol interpreter to act as a proxy.

The current standard form is a preliminary version, and will shortly be replaced. The new version will include the following features:

- Each (32-bit) request code will consist of two parts. The least significant 16 bits will specify the operation, while the most significant 16 bits will indicate whether the operation is common to all file systems or whether it is a specific extension supported by one particular file system.

- For specific extension codes, bit 15, if set, will indicate that there is a path in the "usual place." This will allow file systems which are otherwise unable to 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.

Bibliography

[1] M. M. Astrahan *et al.* System R: relational approach to database management. *ACM Transactions on Database Systems*, 1:97–137, June 1976.

[2] G. Brebner and F. King. *The Evolution of the Fred Machine.* Technical Report CSR-246-87, Computer Science Department, University of Edinburgh, 1987.

[3] M. F. Challis. Database consistency and integrity in a multi-user environment. In B. Schneiderman, editor, *Databases: Improving Usability and Responsiveness*, pages 245–270, Academic Press, 1978.

[4] M. F. Challis. Version management—or how to implement transactions without a recovery log. In M. P. Atkinson, editor, *Database*, pages 435–458, Pergamon Infotech, 1981.

[5] H. Dewar, V. Eachus, K. Humphry, and P. McLellan. *The Filestore.* Technical Report, Computer Science Department, University of Edinburgh, 1977. Second Revision: August 1983.

[6] H. M. Dewar and M. R. King *et al. APM Reference Manual.* Technical Report, Computer Science Department, University of Edinburgh, 1983.

[7] W. P. Enos, I. B. Hansen, and R. W. Thönnes. *Edinburgh Local Area Network.* Technical Report, Computer Science Department, University of Edinburgh, 1981.

[8] W. P. S. Enos. *Ethernet Protocols: Design and Implementation.* Master's thesis, University of Edinburgh, Computer Science Department, 1981.

[9] J. Gray *et al.* The recovery manager of the *System R* database manager. *ACM Computing Surveys*, 13:223–242, June 1981.

[10] P. M. McLellan. *The Design of a Network Filing System.* PhD thesis, University of Edinburgh, 1981. Available as Technical Report CST-12-81.

[11] P. M. McLellan. *Shrines.* Technical Report, Computer Science Department, University of Edinburgh, 1982.

[12] J. Postel. *Internet Protocol.* RFC 791, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California 90291, September 1981.

[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.