DISC SYSTEMS : AN EXAMPLE

The major part of this chapter is taken from an account of a fairly simple and practical file system[IMP25]. It was originally intended as a server designed for use in a distributed system, but that doesn't affect its design significantly. Its most significant unusual feature is its avoidance of imposing any particular file structure on the material stored; that is particularly appropriate for a server which might be required to provide resources to different operating systems on different processors.

CFS is the Cambridge File Server, one of the very few successful network file servers at the time.

4. THE ARCA FILE SERVER

The Arca file server is a stand alone system which provides a file store to be shared by other systems, henceforth called clients, on the network. It allows clients to create, destroy, read and write files from a distance. It is important when designing a file server to strike the correct balance between what the file server is expected to do for the clients and what they must do themselves. In the case of the Arca file server we had three main aims:

(1) To do as much for the client as possible without either imposing any kind of restriction on its clients regarding file structure or taking on time consuming operations that will hold up other clients.
(2) To allow each client to impose his structure on files and to let several different file structures co-exist on the one filing system.
(3) The file server should not lose data when a crash occurs and should be able to recover automatically.

In order to satisfy the third requirement the file server has to ensure that certain types of file will be updated atomically, that is, if either the hardware or software fails during a transaction, the file server will return the file to either its final or original state depending on whether the file was closed or not (see section 7). Because of the overheads involved in the atomic update sequence, it is adopted for a file only if the client explicitly requests it when the file is created. This type of file is called 'special'. Although the client could be allowed to change the 'special' attribute of an object, we decided it was not a useful thing to do because objects which are easily re-creatable (ie compiler listings) will always be re-creatable and vice-versa for non re-creatable objects (ie program sources). As indicated in section 3 we decided to base our file server on CFS, which has the following desirable properties:

- high speed transfers to random access word-addressed files.
- the ability to perform atomic updates to files.
- a capability-like access control mechanism.
- automatic reclamation of unused storage.
- attention to the integrity of stored data.

In the next section the underlying file structure of our implementation, based strongly on that of CFS, is explained in detail.

5. FILE SYSTEM STRUCTURE

Each object is uniquely identified by a PUID (Permanent Unique IDentifier) which is created with the object and is never re-used or destroyed. A UID has the format shown in figure 2.

When an object is opened, (ie disk held information about the file is entered into a table), the file server returns a TUID (Temporary Unique IDentifier) synonymous with its PUID. This TUID is valid only until the object is closed and is never re-used. The random bit pattern is the only kind of protection the file server uses. It makes use of the fact that 32 random bits will be very difficult to guess and will take too long to try out all combinations. Two types of object exist on the File Server: files and indices. A *file* is an array of bytes, whereas an *index* is an array of PUIDs, each representing a file or index. Some (or all) entries in an index may be the null PUID (all zeroes). Note that the client is not allowed to write explicitly to an index. Any index may contain a PUID for any object provided only that the object is in existence. Thus, the structure is that of

an undirected graph (figure 3). One index, however, is distinguished from the rest – that of the *root-index*. This index is important because any object not reachable by any chain of indices starting from the root is eligible for deletion. No further structure is imposed on the file store other than that mentioned above. Thus, although creation of objects is controlled by the clients, their deletion is controlled by the file server.

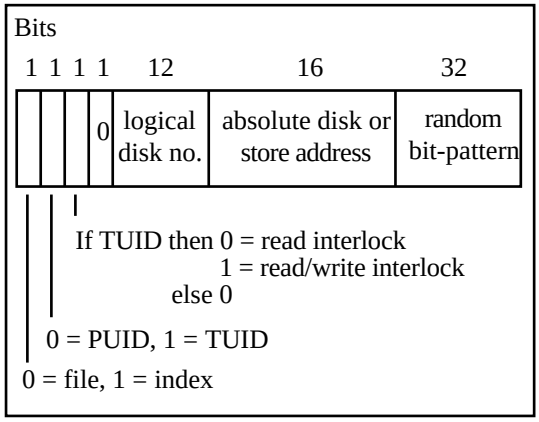

Fig 2. Format of a uid

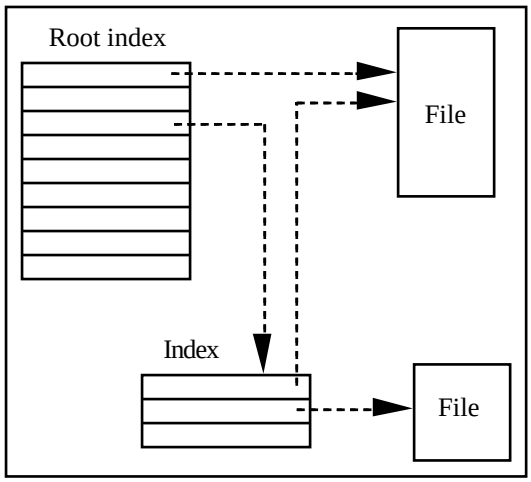

Fig 3. File store graph structure

So far we have described the client's view of the filing system. Its internal structure is now explained. Each object (file or index) has certain attributes, namely whether it is a file or index, special or non-special (indices are always special), its PUID, its 'uninitialised' value and its logical size. The 'uninitialised' value is the value returned for any byte which has never been written, and is always zero for indices. The logical size determines *only* the highest word address which may be written and has little bearing on the actual physical size. Thus, increasing an object's logical size will not change its physical size (except if a change in the object's 'depth' is caused – see later), but a reduction of its logical size may cause some physical blocks to be removed from that object. Reduction of the logical size of an index may also cause references to PUIDs in that index to be lost. Each object also has a 'high water mark' indicating the highest address ever written. The logical file size and the high water mark are the only dynamic attributes of an object. Transparent to the client is the object's 'depth' which can be one, two or three depending on its logical size. The following gives the object's depth, D, in terms of the logical file size (in bytes), L:

$$L \leq 504 \qquad D = I$$
$$504 < L \leq 252*512 \qquad D = 2$$
$$252*512 < L \leq 252*256*512 \qquad D = 3$$

Note that, for an index, L is always a multiple of eight (because a PUID is eight bytes), but the client sees its logical size only in terms of the number of entries it holds. There are 512 bytes in each disk block. All objects consist of at least one block, called the 'first block', which contains some of the object's attributes in the first eight bytes. The meaning of the remaining 504 bytes depends on the object's depth. For a depth of one, these bytes contain the data itself; for a depth of two they contain 252 block addresses, each of which contains 512 bytes of data; for a depth of three they give the addresses of indirect blocks, each of which contains 256 block addresses, each of which in turn contains 512 data bytes. When an object is created, it always occupies one physical block regardless of its logical size; if its depth is greater than one, the remaining 504 bytes of its first block are null pointers. Further blocks become allocated as necessary; until then, they are read as the appropriate number of 'uninitialised' bytes. The last block of each cylinder of the disk is used by the file server as a 'cylinder map'. This map contains four words of status for each block on that cylinder. The information contained in the cylinder maps and the tree structure of each object are mutually redundant: thus, each may be rebuilt from the other. This is the basis of the file server's ability to perform atomic updates (see section 7). If the cylinder map becomes bad, a new disk will need to be obtained. The information held in

the cylinder map for each block is as follows:

word 1:

1. allocation state (1 = allocated, 0 = de-allocated) – 1 bit
2. intention state (1 intending to change allocation state) – 1 bit
3. first (1 first block of an object) – 1 bit
4. index (1 = index, 0 = file) – 1 bit
5. commit (1 = commit, 0 = don't commit) – 1 bit
6. level1 (1 this block contains pointers to data blocks) – 1 bit
7. level2 (1 this block contains pointers to level1 blocks) – 1 bit
8. (9 bits unused)

word 2:

sequence number (0..255) of block within its parent (the block that contains its address) or zero if none

words 3 and 4:

if this is the first block of an object, these words contain the random part of its PUID, otherwise they contain the address of its parent block and the address of the object's first block.

Finding a free block to extend an existing object is performed by looking first on the cylinder containing its 'first block', then examining cylinders in both directions from this point. It is hoped that this algorithm will minimise the head movement when accessing an object.

7. ATOMIC UPDATES

The File Server has a mechanism by which an object can be updated atomically in that, if an update fails, the object will be restored to either its original or final state. A file may be given the property 'special' at create time which causes all updates to the file to be performed atomically, otherwise this mechanism will be by-passed. The latter case is useful when a file can be easily re-created and one wishes to avoid the overheads of atomic updates. Note that indices are *always* treated as 'special'. As stated in the previous section, the information pertaining to the consistency of the file store (excluding the contents of data blocks) is recorded twice: once in the graph structure of the indices and the tree structure of each object, and once in the cylinder maps (so called because there is one per cylinder). An atomic update to a 'special' object of depth two is as follows:

When an object is opened for writing, a copy is made of its first block and the new block is marked as a 'new first block'. As the object is written, intending to allocate/de-allocate block pairs are generated. These intending to allocate/de-allocate states are used by the automatic crash recovery program to enable it to ascertain which blocks should be freed and which blocks should become part of the object. When the object is closed, the following steps occur:

1. Set the 'commit' bit associated with the object.
2. Copy the 'new first block' into the 'old first block' and de-allocate the 'new first block'.
3. Change all 'intending to allocate' blocks to 'allocated'. Change all 'intending to de-allocate' blocks to 'de-allocated'.
4. Reset the object's commit bit.

If the file server crashes and restarts, the 'restore' program will examine the disk and will either undo or complete any unfinished atomic transactions. If any cylinder map is found to be unreadable, all cylinder maps will be rebuilt by traversing the undirected graph from the root-index. Otherwise, the commit bit of each object is examined and the allocation state of each block involved is changed as follows:

commit = 0:
intending to allocate → de-allocated
intending to de-allocate → allocated

commit = 1:
intending to allocate → allocated
intending to de-allocate → de-allocated

Note that the 'restore' program undoes/completes atomic transactions in the same way as the file server so that a crash in this program is also recoverable. Note also that this method assumes that a block which was half written will be left detectably bad.

Having set the commit bit of the object, no further cylinder maps are written until the 'new first block' has been copied onto the 'old first block'. Thus, if the cylinder maps need to be rebuilt, the view of the object from its first block will be correct according to what the commit bit would have been. Therefore, the only ways the file store could be damaged are due to software errors or physical damage to the disk.

8. GARBAGE COLLECTION

Garbage collection is performed asynchronously with the normal operation of the File Server. The garbage collection algorithm runs at priority zero and is interrupted by the receipt of any network request. This algorithm is as follows:

1. Mark all objects as 'not found'.
2. Search graph marking objects seen as 'found'.
3 . For each object marked as 'not found' do the following:
   (a) mark its first block as 'intending to delete'
   (b) mark the rest of its blocks as 'intending to delete'
   (c) mark its first block as 'deleting'
   (d) free all its blocks in depth-wise order.
4. Reset the 'intending to delete' bits of every block.

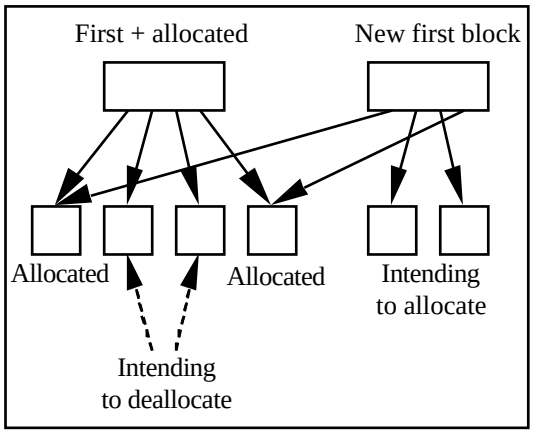

Fig 4. Atomic update

During step 2, a note is made of any first block marked as 'deleting'; if so, the process of deleting that object will be continued immediately before step 3 begins. Thus, there will only be one partially deleted object at any time. No object will be 'retained' in an index if its first block has the 'deleting' bit set. Whenever any object is 'retained' in an index, the 'found' bit is set for that object. This is done to prevent the garbage collector from deleting it. When an index is retained, the garbage-collector is restarted.

REFERENCE.

IMP25 : S. Muir, D. Hutchison, D. Shepherd : "Arca : a local network file server", *Computer Journal* **28**, 243 (1985)