THE (NEW) NEW FILESTORES

George D.M. Ross 26/02/85

This discussion paper is intended to air some thoughts on the facilities which should be provided by the Department's network file servers. It is clear that the facilities which are currently provided are severly limited in a number of respects, and it is desirable that the servers should be upgraded to correspond with current requirements.

A historical overview

In the beginning was the Interdata filestore. It had two 67 Mbyte CDC disc drives and a Dataproducts lineprinter attached. Clients were connected via point-to-point links, either directly or via a multiplexor (another Interdata). This system went into service around 1976. Slight protocol changes were made around 1981 and a connection to the 2Meg ether was added to allow additional clients to be supported on a "virtual link" basis. This original system then remained virtually unchanged until it was replaced by the current generation of filestores.

It had become clear by 1983 that the Interdata-based filestore would have to be replaced. The hardware itself was nearing the end of its working life, and the fact that the ether station was unable to handle the load placed on it by the ever-increasing number of clients meant that reliability was a perpetual problem. The solution adopted was to produce an interim implementation of a filestore based on the hardware envisaged for the new filestores, viz. a Fred-machine with a 300 Mbyte Winchester disc drive, but running a system which would be a re-implementation of the specification of the original filestore. This system was phased into service with a half-size disc around the end of 1983, and the old Interdata filestore was gradually phased out. Subsequently two more filestores were built, each with a 300 Mbyte drive, and the small filestore was transferred to the Appleton Tower where it now supports a cluster of Sirii via asynchronous lines. In addition, a filestore-protocol interpreter was written for the Department's VAX-11/780 (running VMS).

This brings us to the situation at present. There are two 300 Mbyte filestores at K.B., and another 160 Mbyte filestore in the Appleton Tower, with a third 300 Mbyte filestore envisaged for K.B. These filestores support a large number of personal machines, all of which rely entirely on the filestores for their permanent storage.

It is important to remember that the current situation was originally intended as an <u>interim</u> measure until such time as the design for a new super-improved filestore could be worked out. This situation has now lasted for about a year, during which time it has become apparent that the original specification does not allow for some of the functionality which is desired of our file servers. With the prospect of a new (or, at least, different) operating system for the Fred-machines it is now time that the filestore specifiction was revised.

Deficiencies in the current filestores

The greatest limitation of the current filestores is the directory structure, which is flat with a one-to-one correspondence between directories and owners. The upshot of this is that it is not possible for users to group related files together in the same directory easily, as creating a new owner is a system management function. For those facilities which have been deemed important enough to invent a dummy owner for, there is the question of access: would-be software maintainers must either log on as the dummy user or log on as themselves and quote the dummy user's password. If several facilities are being worked on this can be highly inconvenient. Ordinary mortals must try to make the best of it, naming their files so as to distinguish between those belonging to different projects.

The directory structure also imposes a limit on the total number of files which an owner may have. The upper bound on this is due to the fixed size of directories (2048 bytes), but as extent information is held in the directories as well as naming and access-rights information the limit is reduced by fragmentation of the disc.

Multiple versions of files are not well supported. A file can exist in no more than two versions at any time, and then only when one of the versions is currently being created (when the filename is distinguished by having a '!' appended). This mechanism is intended to allow safe abandonment of programs which were creating new files, as in that case both the old and new versions of the files are available for the user (though it is rather harder to access the new '!' version than to access the old non-'!' version). When the new file is closed the old version is deleted. This mechanism does not correspond well with, say, behaviour while editing, where it may be desirable to keep several previous versions around for some time.

The protection mechanism is not sufficiently flexible. There are only two categories of file access, viz. owner authority and world authority. Owner authority is acquired by quoting the password for the directory (quoting the system password gives either owner authority or read access, whichever is less restrictive, in addition to the ability to perform filestore-global operations such as creating and deleting users). All other users have world authority.

Removable media (e.g. floppy discs) are not well supported.

This list is not exhaustive. There are a number of other deficiencies which have less impact on the usability of the filestores. The points raised here are merely the ones which are most obvious at a first glance.

Let us now turn to consider what the current interim filestore system should be replaced with. We consider mainly the facilities offered to clients, and the structure of the filestore system which will provide these facilities. We defer discussion of the protocol(s) used by clients to communicate with the filestores to another paper.

The new filestores: directory structure

One of the problems of the current filestore directory structure is that it blurs the distinction between what a file is called and what its attributes (including contents) are. One of the features of UNIX is that it makes this distinction clear: directories are merely lists of name-pairs, each entry consisting of the name the user has given to the file and the name by which the system knows it. It is proposed that the same distinction be drawn in the new filestores. Directories will be lists of name-pairs, each consisting of the user's name for the file and either a system filename or another pathname which is substituted for the path interpreted so far. In 4.2BSD terms these would be considered a hard link and a symbolic link, respectively.

Filename (pathname) interpretation is performed in much the same way as is done by UNIX. There is a path separator character, which should be user-selectable. Pathnames which start with this separator are interpreted relative to the root directory (a distinguished directory whose system name is known to the interpretation code). Pathnames which do not start with the separator are interpreted relative to some default directory which the user has told the filestore about. User-supplied system names, introduced by some other special character (probably a "control" character), are passed on directly without path interpretation.

For the convenience of applications programmers there should probably also be a second path-separator, again a "control" character, which would be guaranteed to work even if the user altered the "normal" path-separator to some non-standard character. Indeed, it can be argued that all distinguished characters should be non-alterable "control" characters, with the client system being required to perform any translation required from the user's preferred non-"control" characters.

Interpretation proceeds by considering only the frontmost component of the pathname at each step. The component name is looked up in the current directory (or the root directory if the pathname started with the path separator). If the result is another pathname then it is prepended onto the remainder of the original path name and interpretation proceeds using that new path name. Otherwise the result must have been a system name. There are then two cases: either the component currently being interpreted was the last part of the pathname, in which case the resulting system name is the translation of the pathname, or there is some part of the pathname remaining, in which case the system name just obtained is used as the current directory with respect to which the remainder of the pathname is interpreted.

Pathname components may consist of any sequence of printing characters whatsoever, though the user should be aware that some characters may be pre-empted for use as distinguished characters, for example as the pathname separator. Case preservation and significance in directory entries should probably be selectable on a per-directory basis. Parity will be stripped.

What we have done, in effect, is to split the filestore apart into two distinct layers. The upper layer is responsible for interpreting the directory structure, while the lower layer is essentially a flat file system which performs the remaining functions required of the filestore, including space allocation, quota management, authorisation and access control. We could, indeed, build this directory structure on top of the current filestore system, though a lot of messy juggling would be required in order to make the access mechanisms work sensibly. We consider now the design of a flat file system which will provide a more secure base for supporting the higher layers.

The underlying (flat) file system

One of the requirements of the underlying file system is that it should be able to support the use of removable media, such as floppy discs. This implies that part of the disc address space of the filestore may not always be present. Furthermore, the file structure of the removable medium may not even be the same as that of the rest of the file system. It is essential, therefore, that the "permanent" part of the file system does not rely on the contents of the transitory parts in any way. This is most easily achieved by partitioning the disc address space and treating each separate partition as though it had its own, independent, file system. There is no need to require that there is a one-to-one correspondence between logical partitions and physical media, though of course our primary purpose would be defeated if partitions were allowed to straddle removable media.

This partitioning has a number of advantages other than allowing for removable media. Write-protected partitions can provide an added barrier against accidental scribbling or deletion. Partition access controls can provide a coarse measure of security. Quotas may be enforced separately for separate partitions, or not at all for some, resulting in a better control of resource usage. The capacity of the partition itself provides a limit on user profligacy. Dismounting a partition effectively makes it inaccessible until it is mounted again. Mounting a non-file-structured partition provides a convenient way of allowing direct access to special areas of the disc address space, such as the filestore boot area, as well as to foreign removable media.

Having separate logical partitions requires that the system names by which the files in the filestore are ultimately known must also have a distinct partition element. Taking the system names as 32-bit integers, we use the top 8 bits as the partition identifier and the remaining 24 bits as the identifier of the file within the partition (the numbers proposed here seem reasonable, but could easily be adjusted if it was felt necessary). Conventionally, partition number zero is taken to refer to the current partition, so that references to files within the same partition are independent of the actual number of the partition they reside in. This is essential if removable media are used, as a floppy disc may be written as partition 7 on one filestore, say, and read back as partition 9 on another. Cross-partition references would probably be the exception rather than the rule, with most references being to files within the same partition.

Within each partition we have a totally flat file system, with files identified by 24-bit numbers (depending on the actual split between partition numbers and file numbers). As with any other file system, there must be some way of interpreting these names so as to find the files they refer to.

One way of doing this would be to take the name as being the address of the file's header block. This has two disadvantages: potentially, any name which corresponds to some block in the partition might be a file header, with there being no absolutely sure way of guaranteeing that even if the block looks like a file header that it isn't part of a file written specially by some malicious user to compromise security; and the only way of finding all the files in the partition, for backup or during initialisation, is by scanning the directory structure, an inter-layer dependancy which we have already rejected.

The alternative is to use the address as an index, either into a table of header addresses or directly into a table of header blocks. There would seem to be no merit in placing the index blocks anywhere other than in the partition itself, since separating them merely reduces locality of access. We could have two separate areas of the partition, one for file headers and one for data, but this distinction loses the flexibility of being able to allocate the header blocks dynamically as required. It would seem at first sight that indirecting via a table of header block addresses would give added flexibility over directly indexing, since new header blocks could be allocated from anywhere on the partition, though at the expense of some extra complexity. However, by amalgamating all the partition's headers into a single file (which we will call the index file, following VMS terminology) we can have the flexibility of indirection, with the file header acting, essentially, as a rather compressed list of file to partition addresses, together with the conceptual simplicity of using the file's system name as an index directly into a table of header blocks, in this case by treating it as the block address of the header block in the index file.

In practice the index file would be created near the centre of the partition. It would be "reasonably" large and would be contiguous. It could be extended, however, in exactly the same way as any as any other file if it were found to be too small.

Clearly, the file system would have to have some way of locating the header block for the index file, as this is the key to the partition's structure. Since this would be accessed so frequently it should be cached, and hence it does not really matter where it is sited in the partition. Conventionally it could be block zero of the partition. Alternatively, it could be the first block of the (contiguous) index file, with a pointer to the index file held in some conventional location, say block zero, of the partition. This latter scheme has the advantage that the index file is made more contiguous, so less time is wasted in interpreting the extent list. The spare space in the indirection block (the "partition header" block) could be used to hold additional partition-specific information, such as protection to be applied to the partition as a whole.

Since it is so vital, the index file's header block should be duplicated, though as it would appear as part of a backup of the index file it could be reconstructed in extremis.

Although we have allocated 24 bits of the system filename to be the partition's name for the file, we would not really expect to have that number of files in a partition. The (non-partitioned) 300 Mbyte drives on the Department's (VMS) 11/780 can each hold a maximum of 50038 files, which has never proved to be a serious limitation. We can, therefore, subdivide the 24 bits into an index part of, say, 16 bits, and 8 bits for some other purpose. These would best be used as a sequence number for the file header, being incremented each time the file with that index is deleted. This ensures that any dangling pointers to the file left when it is deleted do not undangle and point to the new occupant of the index file slot when a new file reuses it.

Access control

For the purposes of this discussion we will assume only that users of the filestore have access rights in respect of the files held by the file system, and that these rights can be confirmed to the file system in some way. We are not discussing, here, the manner in which these rights are established. We merely note in passing that these might include: quoting a username and password with every filestore operation; exchanging a username and password for a SAP-specific token (the technique employed by the current filestore); and exchanging a username and password for a universal token. We are interested here in the access rights per se.

Basically what we want to be able to do is to grant access rights for our files to other file system users. As well as ourselves and the whole word, we may want to grant rights to both individual users and to groups of users. Both VMS and UNIX offer this facility in some form.

VMS considers that users are members of some particular group, and offers a protection class for that group. It also has the notion of a "system" class of user, with separate access rights. A new addition with Version 4 is the concept of "access control lists", whereby protection may be given to objects with a finer degree of control.

UNIX users may be members of several groups, but files are restricted to having only one group protection field. If the group is one of those to which the user belongs then the associated protection is considered when determining whether to grant or deny access.

We note, in passing, that EMAS "group" protection is based on usernames with wild characters. A stylised notion of username is required for this to have any meaning.

The method we propose has elements of both the VMS and UNIX approaches. Each user is granted the use of a number of access tokens. Associated with each file is a list of access tokens and corresponding access rights. When checking a file to determine whether the user is allowed to access it or not these two lists are compared, and for each

of the file's access tokens which matches one of the user's access tokens the corresponding access rights are granted.

Although to the file system there would be no difference between tokens for individual users (a group of size one?) and groups of users, we would probably want to make some distinction between them. If the tokens are represented by integers then we could, conventionally, regard positive ones as belonging to individual users and negative ones as belonging to groups of users. The use of integers also means that token names need never be reused, in practice at any rate. File system security could be compromised by injudicious reissue of token names while there are still files which refer to them.

It would be convenient if users could create new groups themselves, rather than require that management should create them. This ability might be protected by privilege or quota.

The total number of access pairs which each file may have will, of course, be limited by the size of the file header, and a balance will have to be struck between the access list size and the size of the other file header structures.

We will distinguish two of a file's list of tokens. One of them, the world token, is implicitly held by everyone and is present for every file: this defines minimum access rights which all users have to the file. The other is the owner token. Typically, though not necessarily, this would grant more access to the file than any other token. The holder(s) of this token would also have the right to alter the access token list of the file. In addition, the resources consumed by the file are charged to the owner token. Every file must have an owner. Transferring ownership of a file may require some form of privilege.

In a teaching environment it is desirable that course coordinators, lecturers and tutors should have easy access to their students' files. We can do this by allowing for another distinguished access token, the supervisory token. It grants exactly the same access rights as the owner token, including the right to modify the access token list. Anything that the owner of a file can do can also be done by any holder of the supervisory access token.

Every user of the file system is given a token which will be used as the owner token for any files (s)he creates. Changing this token should be a privileged operation. Likewise, each user may have a supervisory token which is applied to any newly-created files; again it should be a privileged operation to change this.

The access rights which a user may be granted should include: read access; write access (to any block); append access; exchange access. The ability to create a file in a directory is determined by the protection on the directory; this may also control the creation of new versions, depending on the scheme adopted (see below).

Transactions and Alternatives

The current filestores are not particularly good at handling multiple concurrent updates to files. The allowed combinations are shown in the following table:

	Read	Modify	Write
Read	Y	N	Y
Modify	N	N	N
Write	Y	N	N

Read implies opening an existing file for reading. Modify implies opening an existing file for modification. Write implies creating a new version of a file. Read and write are compatible since they refer to different versions of the file. Write and modify also refer to different versions of the file, but are deemed harmful since one of the sets of changes must be lost. Concurrent reads are, of course, always safe.

The new filestores should allow any combination of reading and writing if the user (program) asks for it, though the defaults should probably be pretty much as they are now, together with the necessary lock management required to enforce the rules. These locking rules will be applied to all files opened by the filestores for clients. It would be desirable for the filestores to export their lock management ability to clients in order to simplify database implementation, though communications traffic considerations between filestores and clients would probably limit its use to a fairly coarse granularity.

Transactions (in the database sense) are partly supported in the current filestores, but unfortunately not where they are really needed. An unsuccessful close (abandon) option is offered only for files which have been written completely from scratch, which would seldom be the case for a database. UClosing a file opened for modification has pretty much the same effect as Closing it.

As a first step towards rectifying this omission the new filestores should support multiple versions of files. This gives the user the ability to decide which files should be preserved and which can be deleted. These could be implemented either at the directory level or at the file level. The former is probably more convenient from the filestore's point of view, as it avoids the need for a network of pointers cross-linking the file headers, and it also makes it easier for the user to find out about the existence of the different versions.

Secondly, the filestore should support an "exchange extents" operation between two files. This would allow the data in the files to be swapped over quickly and atomically. It should, of course, be implemented so as to be restartable if the filestore should crash partway through. This might be used as follows: file A is opened, processed, and written out to a new version B; once B has been closed

the extents of A and B are exchanged, so that A now contains the data formerly in B, and vice versa; B, now being the old version, can be deleted. To be most useful, this operation should be able to deal with several pairs of files at once. If differential files are implemented (see below) then the relevant pointers should also be exchanged.

Thirdly, the filestore should implement differential files, allowing for a hierarchy of alternatives of a file to exist. As well as its CAD applications, this would allow for more efficient teaching databases, for example, as the (large) static part would be shared, with only small differential parts being owned by individual students. Transactions are abandoned merely by deleting the differential part; they are committed by exchanging extents and differential file pointers between the differential and static parts' headers. The differential part can be merged into the static part at a convenient time, possibly after creating an inverse differential file, and the extent information of the static and differential parts exchanged again. For more details of differential files and transactions see "Virtual files: a Framework for Experimental Design" (CST-26-83).

Audit files

For accounting and security purposes it could be desirable that any accesses to particular files should be logged. It is proposed that the filestores should implement audit files into which they will write (at least) information concerning who has accessed what and in what mode. Other information may also be desirable.

Interfaces and protocols

Throughout this document we have avoided any discussion of the protocols by which clients should converse with the filestore. The protocol currently employed was developed when the primary mode of connection was by means of a serial link: characters which are written in at one end appear at the other end where they may be read; flow control is performed on a per-character basis. This protocol has been adopted in an essentially unchanged form for the current ether-based filestores. Whether it is still the most suitable in terms of the packet layout and the parameter-passing capability is, however, open to question.

Likewise, we have avoided any mention of the particular interfaces presented by the filestores. Clearly these would include the following in some form: create file; open file; close file; read block; write block; read header; modify header. Some operations on otherwise inaccessible files should also be provided: change password is an obvious example. Some of these might be merged together. The precise details depend, inter alia, on the method of access rights confirmation.

Multiple filestores

Throughout this paper we have been dealing with the case of one single autonomous file server. In the Department we already have several file servers, all of which currently work totally independently of all the others. The question must be asked: is this the best approach, or should we be aiming to distribute the functionality amongst all the servers on the network? Given the nature of the file servers discussed here, there would seem to be three possible approaches to integrating the file servers.

We could have a fully distributed file system, from the lowest level upward. This loses us some redundancy -- at present if one of the filestores is unavailable we can still work just as well with the remaning ones, since there are copies of all the vital system files on all of the filestores. It also adds considerably to the complexity of the tasks the file server must perform and adds inter-filestore traffic to the network in addition to filestore-client traffic. This option would also require a distributed lock management system.

We could maintain independent servers at the file system level -- indeed the partition structure described maintains essentially independent file systems even within the same server -- but distribute the directory structure among the filestores.

We could keep the filestores totally independent. This last scheme makes it harder to maintain consistency between versions of files held on different filestores, but does add valuable redundancy. We can enhance the directory structure so as to allow for cross-filestore references, but these should be passed back to the client to interpret. For example, the directory entry /CS3 on filestore B might contain information to the effect that "the filestore you want is C and the path you want there is /CS3". Note that although the pathnames on both filestores are the same here, this need not always be the case: /P/Q on filestore B might point the client to /Z/X/Y on filestore C. The resulting pathname should be absolute (relative to the root) for this to be meaningful, of course.

The distinction between the second and third possibilities is not great. The only real differences are: that in the second case the filestores must redirect client requests themselves, while in the third case they merely inform the client; and that in the second case a fault may result in a fatal hole in the (global) file system, while in the third case users of the remaining filestores are unaffected if one goes down. The difference is more one of operating practice than design principle.

Envoi

This discussion paper was not intended to provide a description of the new filestores complete in every detail. It proposes some of the facilities which this particular user would find useful in the new

filestores, but leaves many loose ends. It is essential that the user community should make its wishes known, otherwise things will either remain as they are, or drift slowly along with the current filestores growing by accretion.

Trademarks

VAX and VMS are trademarks of Digital Equipment Corporation. UNIX is a trademark of AT&T. Apologies are hereby made for any others which have been missed.