Table of Contents

SOME IDEAS ABOUT THE NEW FILESTORE

1. The virtue of simplicity

1.1. The general argument for simplicity

The design of a filestore is not independent of the design of the operating system(s) that will use the filestore. The capabilities and structure of the filestore affect the system(s) in many ways.

Suppose a filestore FLAP includes a useful facility known as booping (not defined here). FLAP will accept requests to boop a nominated set of files. An operating system OSTRICH wants to use the FLAP filestore. OSTRICH has four choices of what to do about booping. It can present the facility of booping directly to its users; it can use booping to support some feature which is already in the design of OSTRICH; it can invent a new facility which it will map onto FLAP's booping mechanism; or it can ignore booping altogether, so that OSTRICH users cannot use booping.

Giving users direct access to booping is a simple choice, but it is only sensible if booping is compatible and consistent with all the other things that OSTRICH does. Otherwise, either booping will not "work" at all for OSTRICH users, or it will seem to the OSTRICH user to be inconsistent and confusing. So this choice is only practical if OSTRICH happens, by coincidence or design, to conform to the expectations of the people who designed FLAP.

Using booping to support an existing OSTRICH facility also requires that OSTRICH should be compatible to some degree with FLAP, but it does not demand such complete consistency as giving users direct access to booping. The user interface, for instance, can be "mapped" so that the user gives commands which feel natural on OSTRICH, and his commands are translated into the appropriate form for FLAP. This scheme has obvious attractions, but it also has a serious disadvantage if OSTRICH is not the only system attached to FLAP. Users of a different system OSCAR may have an entirely different interface to booping, and indeed booping may be used by OSTRICH and OSCAR in quite incompatible ways. This can easily lead to great difficulty and confusion in mutual understanding, and can make it hard or even impossible to share work and data. Thus one of the advantages of a shared filestore is lost.

The same difficulties can arise, of course, if OSTRICH "invents" a new facility to allow its users access to booping, instead of using booping to support an inherent feature of OSTRICH.

The last choice is for OSTRICH not to use booping at all. This is no real problem, except that the efforts of the designer of FLAP have been wasted and the end users of OSTRICH get no benefit.

1.2. Simplicity in the filestore hierarchy

A straightforward example of this kind of difficulty is the organisation of files into a hierarchy. Clearly, there is no problem if the hierarchy provided in the filestore is exactly the same as the hierarchy defined by the operating system. Where there is more than one operating system involved, there will still be no difficulty provided that they all use the same hierarchic organisation: but that is not very likely, for it

demands that the "permissions" systems should be the same, and that the syntax (and semantics) of file names should be the same.

If filenames have a different form on OSTRICH and OSCAR, then any user who has dealings with both operating systems will probably need to know how to "translate" an OSTRICH-name into an OSCAR-name: a nuisance, but not intolerable. He will probably also need to understand FLAP-names, if they are different. What will be much more troublesome will be understanding the interactions between the permissions systems of OSTRICH, OSCAR and FLAP. Permission mechanisms vary so much in basic purpose and design (let alone in detail) that it is very hard to map one onto another. If OSCAR and OSTRICH and FLAP also operate "generation" or "version number" schemes, or "atomic replacement" of old files by new files, or automatic housekeeping and tidying, then the opportunities for conflict and confusion are multiplied again.

In this example, it seems to me that it would be reasonable for OSTRICH and OSCAR to perform automatic mappings of the syntax of file names, but it would probably be better not to try to map the permissions, nor any generation and version mechanism. Consequently I would expect OSTRICH users to have to deal directly with the FLAP permission system, and they would have to understand the differences between OSTRICH's and FLAP's permission systems. Those who had also to deal with OSCAR would have still more to remember. Given all the possibilities for misunderstanding and mistakes, the best service that FLAP could do for users would be to keep its own permission system as plain and simple as possible.

1.3. An example: supporting UNIX

If UNIX is to be one of the systems served by the filestore, then obviously it would be helpful if the filestore supported the UNIX style of organisation. An implementation which is "almost like UNIX", unfortunately, is likely to be very little better than a filestore which is totally non-UNIX-like. Partial compatibility is hardly ever a worthwhile objective, and it usually causes trouble by leading people to expect and assume too much. So implementing a filestore for the benefit of UNIX would involve emulating the UNIX filestore in considerable detail, including the permissions system and perhaps the internal structure of directory files. That would impose the same mechanism on all the other systems which wanted to use the filestore. Since the design of a filestore has implications in many areas of an operating system, the other operating systems would be constrained to behave like UNIX in more ways than simply using similar file names. The danger is that those other systems would look like unsatisfactory variants of UNIX rather than having their own distinctive virtues.

If the filestore was not organised in the UNIX style, it might be possible for the UNIX systems to support what looks to the users like a UNIX-style filestore on top of the facilities actually implemented by the filestore. In practice, such attempts usually run into difficulties because of relatively minor technicalities, and they are eventually made to work by simply avoiding the areas of difficulty and hence not using many of the available facilities. In the case of UNIX, one might find that the central filestore's hierarchic organisation was unsatisfactory, so that files held on behalf of UNIX would have to be held as a flat unstructured set, and some of those files would represent UNIX directories, and the UNIX systems would do all the work of maintaining the hierarchic organisation of the files according to UNIX conventions. Apart from being wasteful and complicated, this sort of scheme can make life very difficult for people who need to access files on the central filestore from more than one operating system, for the different systems and the central filestore may all use different names for the same file.

The last option, and probably the least unsatisfactory, is for UNIX to let its users see the central filestore as it is, and leave the users to cope with the differences between UNIX-style organisation and filestore-type organisation. This might involve explicit renaming of a file when it is moved from the filestore to UNIX, and changing the access permissions on the file. If one chooses this approach, then the best service one can perform for the end users is to keep the central filestore as simple as possible. Every extra facility in the filestore is an extra potential incompatibility.

1.4. Simplicity in the departmental filestore

These problems have not arisen in the past because machines connected to file stores tended all to run one system, and to have no local file storage of their own, or to use the same mechanisms as the central file store. This will certainly not continue to be the case if APMs come to run divers systems (including UNIX), and if other machines running proprietary systems are to be serviced by central file stores. I am therefore arguing for extremely simple structures and facilities in any new central file store.

2. What a filestore knows about files

I would like to distinguish four things that a filestore needs to keep for each file: the file's name N; administrative details A, such as "where to find the file", "date last accessed", and the "properties" that I have already mentioned - primarily the access permissions; a description S of the internal structure of the file (e.g., the file type, plus - usefully but not typically - a text description of the contents of the file, composed by the owner of the file); and the data D of the file. The reason for this distinction is that all four items need to be handled differently. The name N is clearly rather special: it is essential to be able to locate any of A, S and D from it, but not vice versa. The administrative details A are the preserve of the filestore, and access to read them (let alone alter them) has to be strictly controlled. Even the owner of a file cannot read A directly, and may only be allowed to interrogate some parts of A. A should be the only way to locate S and D, so that location from N would have to be indirect via A. The filestore's handling of a file should depend only on A, so that two files with similar administrative details should be treated identically even if their internal structures S are quite different. S and D are the preserve of the owner of the file (or anyone who has sufficient access rights), who is entitled to examine them and change them freely. When a file is copied, S and D must both be copied together, and the copies associated with a new N and A. When a file is read, you get free access to S and D, but not to N and A (although you probably know N anyway). If a file is held off-line, D is obviously absent (and a copy held elsewhere), and N and A must still be on-line if the file is to be restorable. As a precaution against loss, copies of N, A and S must be kept off-line and preferably close to the copy of D. S need not be retained on-line, but it can be useful to keep it if it contains a textual description of the file.

The use of any of N, A, S and D is subject to the permissions system. However, even if you have the right to access them, it may be impossible to access some of them at a particular instant. The filestore would, for example, probably refuse to allow you to open a file for modifying (overwriting) while someone else has it open for reading. This kind of restriction can apply to S and to D, and in fact so long as it applies to one, it probably applies to the other at the same time. On the other hand, users do not get access direct to N and A, but they use N, or interrogate A, by calls on filestore facilities. Consequently there is no need for N and A ever to be "locked". If the permissions system allows you to use file A, and you ask the filestore for some

information about A, you should get a sensible response regardless of what anyone else is doing to the file. If what they are doing is destroying the file, then you may get told "the file does not exist", or you may get some information about a file which will have disappeared before you next try to access it. You would not expect to be told "your enquiry is out of order just now, please try later". If the filestore were momentarily unable to handle your request, it could simply defer the response briefly.

It is possible to devise schemes in which you can access the S and D of a file without knowing its full name N. This involves either being given an "alias" for the file, or being given access to the file when it is already "opened" by some agency. Mechanisms like this are useful for security if it is made impossible to discover N from S and D. They are, however, complicated, and (I think) unsuitable for a general-purpose central filestore. I believe that it is reasonable for a user to be able to interrogate the environment in which he is working, and that includes the names of the files which he may access. This eliminates one possible reason why the filestore should recognise aliases for filenames, and in fact I believe that it would be much better for the filestore to accept only full hierarchic filenames (and names which can be expanded unambiguously, and independently of context, to a full hierarchic form).

The distinction between N, A, S and D does not imply that they all have to be kept in physically different places. It does suggest S and D could usefully be separated from N and A, and that S and D might well be kept "close" together, and perhaps also that N and D might be kept close to one another. What is definitely not a good idea is to split things up so that (for instance) some subset of A is kept with some part of S, while other parts of A and S are held elsewhere.

In order to make these distinctions clear, I would like to have some more precise terminology than the word "header".

3. Grouping files on the filestore

3.1. A simple scheme

The essential function of a filestore is to keep files associated with names and access permissions. A flat filestore like the present one meets that need. Every extension beyond that must be considered very carefully, not simply as a facility in its own right, but as a feature which must be integrated into other systems. Anything which is liable to present compatibility problems should be left out, no matter how attractive it might be in isolation.

If a hierarchic organisation is considered essential, there are a number of choices to be made. Should we have a simple tree-structured scheme, or one which allows multiple names for (or paths to) one file? Should directories and sub-directories be visible as files? Is compatibility with UNIX the over-riding consideration, or simplicity, or what? What facilities will the filestore provide for dealing with directories or groups of files, as distinct from individual files? Presumably one will be able to set access permissions for a whole directory - will there be anything else? One feature which I do think would cause problems is the interpretation of incomplete file names: it really would be better to insist that every client should expand file names into complete hierarchic names before sending them to the filestore. I am also very doubtful about generations, version numbers, qualifiers and so forth. There is no reason why the syntax of a filestore-type name should exclude those, but I do not think the filestore should interpret those parts of a name, nor do anything about generating or changing them automatically.

If the only objective of a hierarchic organisation of files were to allow users to gather their files into named groups, then a very simple "Scheme A" would suffice, involving:

1. an old-fashioned, unsophisticated, boring flat filestore,
2. a reasonably large maximum length for file names,
3. a conventionally accepted separator to delimit the components of a file name.

In this scheme, a file name might look like "MS1.TELF.ASF13L". As far as the filestore is concerned, that is simply a longish name; the division into components "MS1", "TELF" and "ASF13L" has no significance except to the file's owner who invented the name. To the owner, however, the name would indicated that the file is associated in some way with all the other files whose names begin with "MS1.TELF". That is enough to provide for the user's basic need to group related files together. It allows for "groups within groups", but since a complete name is a file name, it does not provide for "empty groups". Further complications are only justified if there are further needs to be met - and there are, of course.

One obvious requirement is to be able to ask "what are the members of this group?". There is also a need to do something to "all the members of this group" (although, if a client can ask for the names of all the members, the client could simply repeat the action for each member). Both these needs can be met if the filestore recognises the separator in file names. We would need a flat filestore, longish file names, a reserved "component separator" character, and the filestore to recognise that character.

The mechanism could be improved, but with no extra functionality, if the long filenames could be held compactly, and if scanning for members of a group did not involve a scan of all file names. Given the likely uses of this kind of filename, the obvious compaction is to build a tree-structure of the components of names, so that if files "MS1.TELF.ASF13L" and "MS1.TELF.EOC816" exist, then the components "MS1" and "TELF" would only be held once. However, this is simply an encoding of the names; it does not imply anything about the organisation of the filestore. Information is held only at the leaves of the tree, and refers only to files. So we still have an essentially flat filestore, but it satisfies the basic need for grouping of files and interrogation of the membership of groups.

There are more things that one could reasonably ask of a filestore, but it is worth asking whether features that would be essential in a filestore that is integral with an operating system are equally useful in a remote shared filestore. Nevertheless, I will assume that one more thing is desirable: to be able to specify properties that apply to all the members of the group, including those which are created subsequently. That is, groups should be able to carry properties which are to apply to all their members. To achieve this, we need to carry information at the branch points of our "name tree" as well as at the leaves. This kind of information refers to the group of all files and groups which grow on any of the branches from that point. If we can carry that sort of information separately from information about individual files, then we are in a position to allow for empty groups - a useful facility with no extra cost.

Thus by using a tree-structure within a directory we can support a hierarchic organisation of files which I believe is adequate. It is simpler than the UNIX scheme using a hierarchy of directories, and its main limitation is that you can only have a tree-structured hierarchy of files and groups: a file can have only one full hierarchic name (i.e., there is only one path to a file). I do not see that as a problem - indeed, the simplicity is an advantage. If compatibility with UNIX were a primary consideration, then a more UNIX-like mechanism might be useful, but I believe that it would only be worth doing if it were wholly UNIX-compatible, to the point of using identical structures within the files which represent directories - which implies, for instance, using the UNIX style of access control. The same arguments would apply to emulating other systems.

The other possible reason for using a more complicated organisation of files would be the provision of extra facilities. I can not actually think of any valuable facilities that could not be provided by a simple one-directory mechanism, except possibly the ability to treat a directory as a file. This is a neat trick and elegantly exploited in systems which allow it, but it has no great value in itself: it is simply a way of doing things to a whole group of files, and our simple mechanism allows for that in other ways.

The simple mechanism allows us to store and access files with "hierarchic names" of our own choosing, and the structure of file names indicates the association of files into groups. It allows also for the existence of groups, which may contain other groups and/or files. Files and groups may have properties associated with them. We can create new (empty) groups and files, at the "top level" or within other groups. We can interrogate the properties and contents of files and groups. We can delete files and groups, at any level. We can set the properties of files and groups.

What are these "properties"? The obvious ones are access rights. In fact the only others that I can think of are matters of administration and housekeeping, such as the size of a file and the date it was last changed - and those are properties which one may interrogate but may not change.

3.2. Other knids of organisation

Using a hierarchy of directories held as separate files does also have disadvantages in performance and for the management and control of the filestore. Without going over the technicalities, the grouping of files is the user's view of things, defined by him for his own convenience, and it is of very little interest to the system. The system's main interest is in the set of all files belonging to the user, and not in how the user wants to associate those files with one another. For example, if a user is to be restricted to hold a maximum of 24Mbytes, it would not normally matter which group a file was held in - the user does not want to suffer a smaller limit if he places his files in particular groups, and the system manager does not want the user to evade the limit by putting a file in a different group. So it seems reasonable that the grouping of files should be represented within the directory, where the user has control, and the aggregate of files should be represented by the whole directory, which is under the control of the system.

4. File access permissions

4.1. How to hold permissions

Another aspect of this is how one defines access permissions for groups of files. Systems which have a hierarchy of directories often do not use explicit "group permissions": instead, they have a convention that the access permission on the file which contains the directory of a group will apply to all the files within that group. ("The (new) new filestore" presumably intends such a scheme, for it makes no explicit mention of access control to groups of files.) That sounds simple, but in fact it is too simple: for sound practical reasons, the rule is generally not

> "access permission XX to the directory implies access permission XX to all its members"

but "access permission XX to the directory implies access permission YY to all its members".

In other words, the access permission to a member can be derived from the access

permission to the directory, but the permissions are not necessarily the same. Even this leaves problems. For instance, write access to a directory would allow a user to write different data into the directory, which implies that he could make any change at all to the files which are members of the directory (and, indeed, he could fill it with nonsense and destroy its structure as a directory). This is generally felt to be too drastic, so

1. write access to a directory may not actually allow a user to overwrite the directory, even though the directory is a file, or
2. write access to a directory may imply less access to the members of the group than the user could actually achieve by writing to the directory.

Difficulties of this sort have led some systems to provide a comprehensive permissions system which recognise such distinctions as

- permissions to access individual files (read, overwrite, execute) in ways which involve no change to the directory entry for the file;
- permissions to alter the directory entry for a file (e.g., change its size, replace it, change its permissions);
- permission to destroy a file;
- permissions of those three sorts to apply to all the files which are members of a directory;
- permission to add new directories to a directory;
- permission to add new files to a directory;
- permission to destroy a directory;
- permissions which will apply to all directories which are members of a directory;

and so on. These schemes do arguably "do it right", but they far too complicated and are consequently misunderstood and misused, so in the end they can provide even less security than a simple "illogical" scheme.

4.2. Permissions, tokens and user groups

About access permissions: page 6 of the (revised) "(new) New Filestore" suggests that EMAS is very different from other mechanisms. In fact the proposal is very similar to what EMAS does, except that a process uses (or owns) a token which is not the same as its user name. Indeed, it suggests the use of stylised tokens just like EMAS' stylised user names. Tokens and stylised user names are alternative mechanisms for allowing the definition of groups of users.

It seems clear that one access token is intended to be able to carry more than one access right (to more than one file) - otherwise the "tokens" would simply be passwords.

The essential difference from EMAS is that one user may have several tokens. This can be exploited to give access to files to groups of users in a more flexible way than the EMAS scheme, and that would definitely be useful. It seems to me that that is the sole justification for using tokens, and consequently the token mechanism needs to be able to achieve that and nothing more. In particular, there is no need to issue a new token for every access permission: one token is needed for each user and group of users. None of this requires any particular structure within a token; it is simply an arbitrary value. Random binary patterns might serve, or freely chosen strings of text. It might make things easier for people to understand if one used short text strings - specifically, the username, for a single user, and a usergroup name for a group of users. Then every user would posess one token implicitly (viz., his own username), and that token would not have to be stored explicitly in the description of the user. A user could also posess other tokens, which would be recognisable as the names of usergroups, and those tokens would have to be explicitly associated with the user. It

does not seem to me that new tokens would need to be issued frequently, and I would not anticipate problems with the re-use of old tokens.

Each token would carry access rights to one or more files. There is a question of how that should be represented. Should the filestore have a list of tokens, and for each token a list of files and access permissions? Or should each file have a list of tokens, and with each token a list of access rights? The latter seems altogether simpler, and since I anticipate files being created and destroyed very much more often than tokens, it would also give much less trouble with housekeeping.

4.3. What permissions are useful?

For practical purposes, a rather small number of access permissions will serve adequately, and more elaborate schemes are probably more trouble than they are worth.

For groups -

1. EXECUTE all members of the group (may include permission to find the names of all members which are files of object code, to interrogate the administrative information on each such file, but not to read the structural information on such files; gives no useful access to the structural description and data of members which are not files of object code, but may permit interrogation of names and administrative information).
2. READ the data of all members of the group (includes permission to find the names of all members, to interrogate the administrative information on each file, and to read the structural information on each file).
3. MODIFY the directory: includes the right to DESTROY the directory, and implies all possible access rights to all members of the group.

Access rights to a directory would presumably apply also to any sub-directories which were members of the directory, and so on through the levels of the hierarchy.

For files -

4. EXECUTE the file. Allows the use of the name of the file, and interrogation of its administrative information, but may not permit reading its structural description.
5. READ the data of the file, and its structural description. Allows the use of the name of the file, and interrogation of its administrative information.
6. OVERWRITE the data of the file without changing its administrative information (suitable, for example, for a conventional direct- access file of fixed-length records, or a journal kept as a cyclic buffer). OVERWRITE includes all access rights granted by READ permission.
7. MODIFY the administrative information of the file. Allows the file to be DESTROYed or totally replaced, and all possible access to the data and structural description of the file.

A user should be able to discover the names of all the files which are permitted to him, regardless of the kind of access permitted or of the route by which he has access, and he should be able to use those names, and interrogate the administrative information on all such files. He should be able to read the structural description of any file to which he has at least READ access rights.

4.4. EXECUTE access permission

Although I have included EXECUTE permissions in the list above, they are not really appropriate for a central filestore. Firstly, they are impossible to enforce unless (a) the hardware supports the facility - and not all architectures which claim to do so are actually effective in practice - and (b) the client's operating system can support it. Secondly, there are two possible uses for an EXECUTE access permission. One is to allow people to run code without being able to read it, to prevent software theft; the other is simply to prevent people from accidentally attempting to execute data which is not object code. These two objectives are not compatible, and neither of them is easy to make secure, and neither of them is particularly important for the APM. Thirdly, EXECUTE access permission is the concern of other parts of an operating system apart from its filestore (e.g., the object code loader): the filestore may "know" that user XXX has EXECUTE access rights to file FFF, but when it receives a request from user XXX to deliver file FFF, the filestore has no way to ensure that user XXX will not read the data instead of executing it. With a central filestore serving more than one operating system, the problems are altogether too many, and the benefit of solving them is not worth the effort.

If EXECUTE permission is considered essential, then I would propose the following scheme. Enquiries about "is this file executable code?" should be satisfied by examining the structural description of the file, which requires READ permission. READ permission would allow execution; if it did not, a user could easily bypass the restriction by exploiting READ access to copy the file into a file of his own which he could then execute. The filestore would be able to hold EXECUTE permissions and these would only be practically effective for files to which one had ne READ access rights. If you had EXECUTE-but-not-READ rights to a file, you could (a) discover the name of the file, (b) use that name, and (c) interrogate the administrative information about the file. You would have no access to the structural description of the file. You would have to rely on the loader in your operating system to load the code for you. The loader would need READ access to the file. READ access to any file would entitle any user L to enquire "does user M have EXECUTE access to the file?". The loader would exploit that to check whether it should load the file for you. It would also be able to inspect the structural description of the file to check that it really was object code of a form which suits the hardware and the operating system (this check must be different in different clients, so it cannot be done centrally by the filestore). The loader could then read the file, and load the code into your process, exploiting whatever facilities its hardware provides to prevent you from reading the code.

5. Simultaneous file access

Simultaneous access to a file: a file can be OPENed for reading or overwriting. It cannot be OPEN for execution. I am not yet sure about OPENing a file for replacement. If two users have READ access rights to a file, they should both be able to have it open for reading at the same time, and neither should be able to prevent the other from reading the file. If any one user has a file open for overwriting, no-one else should be able to open it for any purpose unless he says explicitly that he is willing to accept someone else writing "unpredictably" into the file. Thus I suggest that "accept shared writing" should be an optional qualifier for OPEN requests (and it should be FALSE by default). A user making such a request would be expected to make his own arrangements for co-operation or synchronisation with any other process accessing the file, and that presupposes some reciprocal arrangements in the other processes. Hence every task which had the file open would have to specify "accept shared writing" if any one of the tasks had it open for overwriting, and that rule would have to apply irrespective of the order in which the tasks opened the file.

If a task opened a file for overwriting without specifying "accept shared writing", another task could not later open the file even if the second task was willing to "accept shared writing".

6. Further facilities

6.1. Atomic replacement

I am doubtful about the proposed facilities for "atomic replacement" of files. The basic mechanism of atomic replacement (or atomic exchange) is clearly essential. I have not met a system before which offers atomic exchange, but (assuming that there are no "internal" problems) it looks very useful and I would prefer it to atomic replacement. What worries me is the naming of the files - the rule that both old and new versions have the same name, with a marker to distinguish them where necessary. I would prefer it to be left to the client systems to choose the name for the new file up to the moment of replacement. Some systems would leave the choice to the end user, and some might generate names by their own conventions (which, given.long names with separators, they could easily do without risk of name clashes).

I like the EMAS NEWGEN mechanism which performs atomic replacement without disturbing users who are currently accessing the "old" version: they continue to get access to the old version until they close the file, and they get the new version next time they open it. This is generally more useful than refusing to perform atomic replacement when anyone has the file open. In those cases where it is a problem (e.g., when several files have to be replaced together), the necessary locking can be arranged by opening all the files for writing (which will not succeed so long as anyone else has one of the files open) and then cancelling all access permissions. This requires no co-operation from other processes as the use of semaphores would do.

6.2. Versions and generations

Multiple versions and generations of files have given me much grief over the years on various systems, and I believe that the filestore should leave it to client systems to make their own arrangements. Implementing its own scheme would only add to the confusions that already exist. The starting point for the whole argument for a hierarchic filestore is the need for long file names with separators: each client system can choose to use some component of the name as a version number. It would not worry me if the filestore used a second (special-purpose) separator to distinguish version numbers, or used a more complicated syntax for file names so that version numbers could be recognised, so long as the filestore itself did not have rules about the behaviour and treatment of versions of files: those are the decisions which should be left for clients. But there are arguments for having as few rules of syntax as possible for filenames.

Differential files are definitely a good idea. Whether they are important enough to justify the effort of implementing them is something to be discussed elsewhere. But I do not think they would be ruled out by my general preference for simplicity. The point is that: they are useful (if you need them!); they are not easy for clients to arrange for themselves; they are not likely to conflict with facilities built into client systems.

6.3. Path selection

"Currently selected directory": the filestore will get requests not only from numerous different client systems, but also from several different tasks within each client system. Every task may have a different selection (and the meaning of the selection may well depend on the operating system running in the client system). So the filestore must get every request with an identification of the client system and the task within that system, if it is to interpret the "current selection". It will also need to know the owner of each task, so that it can find the appropriate "root" for filenames, and so that it can check access permissions. Different systems will create and destroy tasks in different ways, with different types of names, and in some cases rather frequently. The filestore will probably need to know about the deletion of tasks, so that it can discard its records of "current selections" for deceased tasks (otherwise its tables may fill up). This is a complication that we could well do without.

The Kernel within the APM

Conclusions first!

Functions required in the software kernel (** means that a conventional single-machine mechanism needs generalisation to support distribution):

Basic hardware management

** Hardware grope and reconfiguration

Hardware fault handling and basic diagnostic support

Maintaining the environment (including virtual memory) for individual tasks

** Resource allocation

** Message passing between tasks

Access to shared resources

Task synchronisation with external events

Error trapping for tasks

Filestore access

Creation and deletion of tasks

Basic monitoring and logging mechanisms

These have to divided into global control functions and local control functions.

John Wexler
26th. September 1985

In my back-of-envelope sketches, I envisage a distributed system which is represented in each APM by a small limited kernel of software. Some of the functions of this kernel might be:

Basic hardware management.
Essential.

Hardware "groping" and reconfiguration.
Very useful, but not absolutely essential.

Hardware fault handling.
Essential.

Multiprogramming several independent tasks on the single processor.
Essential.

Maintaining the separate environment for each task.
Essential. This heading covers "virtual memory support" - see below.

Protection between tasks.
Essential.

Allocation of resources to tasks.
This might be left to a special task (e.g., accepting requests in the form of messages from other tasks and sending response messages when the resource is available), or it could be done at kernel level. The function is essential, so the "special task" would have to be a permanent companion of the kernel in any case.

Message passing between tasks.
Essential (except that some people prefer equivalent mechanisms under different names).

Providing interfaces (and access discipline) to resources which are shared rather than allocated.
This can be very useful, but there seems to be a tendency to present all resources through an allocation scheme even if one is only allocating a momentary permission to access an essentially permanent shared object. If one wants simpler mechanisms for shared resources, the kernel is the right place for them.

Task synchronisation with external events.
Essential, but need not be distinct from the message-passing mechanism.

Error trapping for tasks.
Essential.

Filestore access.
Could be done at library level, but it seems so universally useful that the kernel is the more appropriate place.

Creation and deletion of tasks.
Essential.

Monitoring.
The kernel should provide basic facilities to be switched on and off and controlled by higher-level (library) software.

Logging.
If the kernel maintained an in-store buffer of records of recent activity, it could be a higher-level (library) function to make occasional permanent (filestore) copies of the buffer.

Diagnostics.
Diagnosis of hardware trouble would be desirable (if it is practical), and perhaps some basic diagnosis of software problems within tasks but most diagnosis of program errors should be handled by library software.

Providing simple interfaces for facilities whose low-level operations are unnecessarily complicated.
This should only be provided for mechanisms which are universally required; the library is the proper place for most simple interfaces.

Providing common interfaces for conceptually similar facilities whose low-level operations are unnecessarily different.
This kind of thing should be in the library.

External communications.
A high-level facility, not appropriate at the kernel level.

Spooling.
A high-level facility, not appropriate at the kernel level.

Each task should run in its own independent virtual memory (rather than in its own part of a system-wide virtual memory). The support of virtual memory is thus part of "maintaining the separate environment for each task".

The kernel as described so far is an ordinary operating system kernel for a single machine, with no special provision for "distributed operating". Distributed or not, there has to be some mechanism to look after what goes on in an individual machine, and it seems natural for that mechanism to reside in the machine itself: does it make sense to try to run the kernel for machine A on the processor of remote machine B? The provision for distribution should be made by extending certain kernel functions; it should not be necessary to add totally new functions. The areas which would need generalisation include:

1. Hardware groping (to recognise remote machines, etc.)

2. Allocation of resources

3. Message passing

4. Access to shared resources (but simple sharing is probably not appropriate for remote resources)

5. Synchronisation (but message passing is probably the appropriate mechanism for remote events)

So far, I have written of the "kernel" and "tasks" as if they are totally separate, suggesting a simple scheme with a single privileged kernel looking after a number of non-privileged tasks. This is rather too simple. What we are aiming for is to run independent streams of non-privileged user code within tasks; and we can expect to find a single unit of privileged "global control" code at the heart of the system; but the "protection boundary" between kernel and user code need not be the same as the boundary between global control code and tasks. It is reasonable (and useful) to have some privileged kernel code running within individual tasks, so that the protection boundary divides each task into a "local control" section and a user code section. (On the other hand, I can see no sense in putting the protection boundary "below" the task boundary, but someone might find some interest in the idea.)

In a structure like this, the kernel is divided into global control and local control code. The list of kernel functions must therefore be divided into global control functions and local control functions.

If we could work with multiple levels of protection, more elaborate schemes could be devised. For instance, library facilities might be more privileged than user code but less privileged than the kernel. For the time being, however, I am assuming that library facilities would run as user code.

Building in facilities

I would like to distinguish the environment of a running task from the library of facilities that are available to it. By the "environment", I mean those facilities which are always available , provided and supported and guaranteed by the operating system. I use "library" to mean the aggregate of optional facilities which the task may call on if it chooses. The execution of a single stream of non-privileged instructions will be a facility in the environment of most operating systems, whereas mathematical functions are commonly found as part of the library. A facility may be in the environment of one system and in the library of another: for instance, the BASIC language on home computers and on mainframes.

An environment will only work reliably if the running tasks conform to the rules. If tasks bypass the defined interfaces, fiddle about with the underlying structures, take short cuts or attempt to implement their own parallel facilities, then trouble arises. For reliability, especially where one system supports the environments for several independent tasks, the implementation of the environment has to run behind some barrier of protection. Typically, control of the use of interfaces leads to a "layered design" in the operating system.

Not surprisingly, the richer the environment the more restrictive it is. If "freedom", as opposed to "versatility", is important for the use of the APM, then a rich environment would not be appropriate. What is needed is a simple environment with "richness" provided in the form of a library.

Libraries, however, have their own problems - or rather, the same fundamental problems appear in a different guise. Each facility in the library should work reliably for any user - that is, when called in any task. A facility must therefore expect to find nothing more than the basic environment guaranteed by the operating system. It must construct everything that it needs out of the basics provided by the standard environment. No surprises so far: that is plainly the essential purpose of a library facility. However, a task is entitled to call upon more than one facility from the library, and no-one can predict which facilities will be called together. So each facility must be able to co-exist with every other. That means that a facility must make no changes in the environment when it is called, or alternatively that all the facilities must be carefully designed not to interfere with one another.

Now if the facilities operate outside the protection barriers of the system, neither the hardware nor the operating system can detect conflicts between facilities, so the unfortunate user is left to cope with peculiar behaviour in his task and no help in diagnosing the problem. We have no practical available methods for avoiding these hazards apart from planning and engineering.

In the case of an APM system, there is a further level of difficulty, since "freedom" demands that tasks should have access not only to the intended user interface of a facility but also to its lower levels and internals. Nevertheless, I feel that the "library" approach will be right for the APM, and consequently that the design and coordination of the library should be accepted as an important and fairly large task.

Here are two discussion notes identifying essential questions which must be answered before any serious plans can be laid, and identifying a limited set of objectives for a serviceable APM system. The conclusions come first, to save you reading further if you have nothing to add or disagree with.

John Wexler
19th. September 1985

Conclusions:

The most important decision in the development of an APM distributed system will be whether to treat it as research or a service requirement. The choice depends on the answer to two questions:

who wants to be involved in the design and implementation?
what are the prospects for UNIX on the APM?

Looking at the requirements for a service system, the following features would be primary targets -

facilities for the "ordinary user", but no fancy command language;
a free environment for running programs, no richer than UNIX;
access to the filestore;
a very limited kernel of software and a rich library of optional facilities;
failure recovery and diagnostics;
monitoring and measurement;
system-wide management facilities;
modest, inexpensive, levels of security and instantaneous data consistency.

I expect that the omissions from this list will be more contentious than the inclusions.

If, on the other hand, the system is a subject of research, there is much more freedom to choose objectives, and a list like that can not be draw up yet. Instead, we have a list of possible developments which would have some research interest, such as -

a "blackboard"-based system;
a system based on "electronic rumour";
a system exploiting Centrenet.

There must be plenty more ideas to add to this list.

Service or research?

The first question that has to be settled is whether the distributed system development should be a research and development project or whether the objective is to deliver something that will serve users in the department. (I don't mean that research cannot produce something of practical use; but it does allow you to try things which are not immediately and obviously useful.) So: who wants to work on the design and construction, and who would want to use it when it is built? If the first group is large and the second is small, we have clearly got a research interest. If there are few potential workers and many prospective users, it looks more like a

development for service. If both groups are small, then research would be the only justification for doing it. If both groups are large ... but I don't think they are.

"Who wants to build the system?" is a question to ask people. "Who wants to use it?" would not get a clear answer, but we can make some reasonable guesses.

As I understand things, there is a positive confirmed decision that UNIX should be mounted on the APMs as a service vehicle, using the Newcastle Connection to form an integrated system. I don't know how realistic that objective is, but supposing that it is practical, I want to consider several possible ways of using the APMs.

All APMs run UNIX:

```
   advantages - a standard system
                many well-known virtues (mostly flexibility)
                a good deal of software available
                well-known and well-liked (by some)
                works reliably and predictably
disadvantages - some well-known problems
                ugly and unfriendly (to others)
                doesn't exploit the APM
                makes it difficult to exploit the APM even if you try
                an upheaval for users
```

```
All APMs run present system
   advantages - no change or disruption
                total freedom for APM users
disadvantages - system is feeble
                system leaves it to the user to exploit APM features
                impossible to maintain continuity of service
                non-standard
```

```
All APMs run a new system
   advantages - a marvellous system
                reliable
                disciplined development to avoid upsetting users
                exploiting APM harware properly
                completing the APM project
disadvantages - some time, maybe never
                some restrictions on private exploitation of the APM
                no packages/application software
                non-standard
                an upheaval for users
```

```
APMs run divers systems
   advantages - everyone can please himself
                wider range of available facilities
disadvantages - more effort to support, develop and maintain services
                problems in sharing work, data and software
                possible incompatibilities
                complication
```

Reviewing all those, I think it is extremely probable that UNIX and the present APM system would both be used, and would satisfy between them so much of the demand that there would be little need for a new system. I envisage the APMs being used as

(a) very flexible hardware, mostly using the present system, and
(b) readily available computer power, mostly under UNIX.

On the other hand, if UNIX is not going to be available as an effective service vehicle, then there will be a need for a new APM system. Consequently, the prospects for installing UNIX must be a dominant consideration in planning for a new APM system.

I assume that the UNIX scheme is a matter of policy and will not be dropped unless it proves impractical, so the question is "can UNIX really be installed?", not "will the department change its mind?"

Supposing once more that UNIX will be available, the conclusion is that a distributed APM system would be a project for research. Nevertheless, the existing APM system would need some limited development, mostly in the communications area, at least to the point where it could co-exist with UNIX on the Ethernet and the filestore. It might also benefit from some discipline and organisation in its development, but on the whole I feel it would be best left in the hands of the free-style developers and used mainly by them. Users who want a predictable and stable environment would choose UNIX rather than the APM system.

If UNIX is not going to be a serviceable system, things would look very different. There would be a real need for a better system, and the existing system could probably not be modified and extended to meet the need. There is a limit to what you can achieve by bolting new goodies onto an old system. A new design is required. "Backward compatibility" should not be demanded, for it would probably be a crippling constraint. That may mean that the old system has to remain in use to support some old applications and for the benefit of hackers.

Objectives for a distributed operating system on the APMs

One approach is to consider the purposes of an ordinary system and see how relevant they are for the APM.

Providing an interface and facilities for the user:
this is essential; computing power for users is a primary reason for the existence of the APMs.

Providing an interface and facilities for the program developer:
there is no clear distinction between "users" and "program developers" on the APM, but they probably want divers programming support environments rather than a system standard.

Providing an interface and facilities for the operator:
there is no "operator" involved in normal APM use, and although one can imagine an operator looking after the whole distributed system, it is not how I would expect the system to work.

Providing an interface and facilities for the manager:
there is no "manager" involved in normal APM use, but I do envisage some such function for the whole of a distributed system; since the APMs have no local filestore, management would be largely concerned with the central filestore.

Providing an interface and facilities for the engineer:
APM users are usually their own engineers except for maintenance;

present arrangements for engineering seem to work well, and the distributed system would not need to give extra support.

Reducing dependency on human help and intervention:
this is not so much an objective as a description; in any case, APMs are used by knowledgeable enthusiasts who do not want too much kept out of sight.

Maintaining the environment for running programs:
this must not go so far as to limit users' freedom to use their APMs as they please; restrictive features should be as few as possible, and optional facilities come under the "library" heading below.

Providing diagnostics and recovery for failures:
this is particularly important when each unit of hardware may be serving several independent users.

Making up deficiencies in hardware:
hardly relevant, apart from the question of protection, and in any case the APM style is to modify the hardware; gaps in the hardware are seen as opportunities, not shortcomings.

Providing a filing system:
APMs generally have no local file storage, and this fundamental requirement is supplied by the filestore; the distributed system has only to support the interface between user and filestore.

Providing a library of commonly used facilities:
an important objective, since the need for flexibility means that features should where possible be optional (i.e., in the library) rather than built in to the system.

Monitoring and measurement:
this could be very useful in future use of the APM for research.

In summary, there are six objectives:

1. User interface and facilities.
2. Free environment for running programs.
3. Interface to the filestore.
4. Rich library of optional facilities.
5. Failure recovery and diagnostics.
6. Monitoring and measurement.
7. System-wide management facilities.

For another view, consider the differences between known systems and the possible APM system:

1. Individual machines have no local file storage.

2. A central file store is available.

3. Machines must share the network with others which are not part of the system.

4. Hardware assistance and modification is a normal and acceptable solution to problems.

5. Users can be assumed to be intelligent, but not necessarily tolerant.

6. Standardisation and compatibility are secondary considerations, and there is no pressure to "stick with what the supplier supports".

7. Individual machines must be able to behave differently and to support "irregular" tasks; part of the purpose of the APM is to be a "hacker's machine".

8. Applications will not demand a high level of security (but loss of data on the filestore is not acceptable).

9. Applications will not demand the highest levels of certainty about instantaneous data consistency.

10. Non-stop running is not essential, if recovery from failures is prompt and reliable.

11. Commercial exploitation is not envisaged.

12. The number of stations will not exceed a few tens.

13. The system is not intended to be the only, or the major, computing resource for a large community.

14. Money, time and manpower to implement the system are strictly limited.

15. Mounting packages is not likely to be an important requirement.

16. The "normal" environment for a running task need not be very rich. Richness should be found in libraries, not in the standard system.

All these points have interesting consequences.

(1) and (2) together are unusual except in systems like Econet at the lower end of the market. If we consider the aggregate of data used by the system to be one data base, we have to implement a distributed data base in which permanent copies of data can be held in only one place (not the same as a central data base with intelligent terminals). Obviously there will be a Consistency Control Protocol, but point (9) and some others suggest that we should choose carefully when to use the CCP and when not.

Point (7) suggests that the essential parts of the system should cover the network and filestore and a very limited kernel in each APM. Higher level facilities should be supplied as options to be chosen from a library. The usual "user's view" of a system, that it consists of very little more than its user interface, will not be applicable. Interfaces at all levels should be documented and accessible, and should be flexible enough to support unexpected uses. To achieve this and still to maintain a conventional "layered" structure is quite possible, but requires some care.

(15) and (16) mean that we are aiming for something nearer to UNIX than to

VME/B. The point is that the "normal" environment must be supported and dependable for any task, and the system must guarantee the environment no matter how the individual user configures and adapts things to suit his own tastes: otherwise, it is impossible to write utility software and packages which will work for all users. It follows that a more complicated (richer) environment implies more restrictions on the user's freedom than a simple environment. It provides extra facilities, but it obliges the user to follow the rules in using them. For example: in a system with a flat filestore, a user may implement his own hierarchy of files if he pleases, and a second user could implement a different structure, and both users can expect all the standard utilities and packages to handle their files. If the system itself supported a hierarchic filestore, then the users could still implement their own ideas, but they would also have to implement their own versions of utilities and packages.

An APM system will differ from SPICE in points 1, 2, 3, 7, 12, 13 and 14, and possibly others. Point 4 is an interesting similarity with SPICE's use of a microprogrammable engine.

The existence of a single central filestore (point 2) offers the possibility of a "blackboard" mechanism in addition to the normal facilities of point-to-point and broadcast messages.

Point 10 is simply an observation of how the APM is actually used at present: people seem to accept having to reset and restart the system from time to time. When one processor may be serving several users, rapid reliable restarting will be important.

Points 5, 6 and 7 suggest that a sophisticated command/control mechanism is not required for the user interface.

Points 8, 9 and 11 suggest that it is not essential to have a high level of protection against malicious and criminal hacking.

Some technical issues to decide

1. How should work be distributed?

2. What kind of global control structure shall we use?

3. What kind of virtual memory support mechanism can we use?

4. Is there any system which we could usefully borrow or plagiarise?

5. How important is performance? What is known about performance in distributed systems?

John Wexler
8th. October 1985

1. How should work be distributed?

   a. Should things reside at fixed sites? No, except where hardware requires it: for example, compilations might be done on any machine, but file storage must be where the discs are.

   b. Who should decide where a service is to be provided? The client, the server(s), or some overseer?

   c. How large should the unit of distribution be? Clearly, larger than the single machine instruction; presumably smaller than the complete job or session. "Remote procedure call" is an example of a fairly small unit. Despatching a complete compilation is an example of a larger unit. Some systems have imposed constraints on units to make them "small" or otherwise manageable - e.g., time limits, or (more sensibly) limits on their demands for other resources such as memory or network traffic, or restricted environments.

   d. How wide should be the repertoire of distributable actions? A well-defined set of primitives? A set of primitives with enough parameters to make them very general? Any actions which the user may choose to define?

2. What kind of global control structure shall we use?
   I don't know enough possibilities to be worth enumerating. At the moment I prefer "message passing" schemes to "remote procedure

call"; I like the "blackboard" approach; the idea of despatching a task to wander around the network until it finds a home is interesting. That contrasts with mechanisms which describe the task to a central co-ordinator who decides where the task will be performed.

With a "wandering task" mechanism, there is the question of whether the task should migrate from one processor to another until it finds one willing to accept it, or whether the task should be broadcast to all processors (or, equivalently, be "written up on the blackboard"). The first option means that the task may be "accepted" by a processor which would not be the best choice, simply because the task comes to that processor early in its travels. There is also the problem of how a processor is to decide where to send a task next if it cannot accept the task. The second (broadcast or blackboard) option would require the resolution of "disputes" when two processors simultaneously take on the same task.

Another question about the global control structure is whether we have a group of processors all running similar systems and obeying some protocol to ensure that the whole group functions as one co-ordinated system; or whether we have one system with many processors, in which case different parts of the system may run in different processors, and the individual processors are not all running the same software.

3. What kind of virtual memory support mechanism can we use?
The choice is between the conventional mechanism where the VM of a running process is backed up by a temporary allocation of disc space, and the EMAS/MULTICS mechanism. I very much prefer the latter.