DISC FILE SYSTEMS.

We saw earlier (particularly in the chapter *FILES IN THE SYSTEM*) what we wanted the file system to do. So far as the disc is concerned, we have to store the files (which, we recall, consist of at least the two components data and attributes) and the file table, and make it all work. In a very simple system, that's fairly easy, but it gets harder quickly as we try to be cleverer.

FINDING THE FILES.

The conventional files administered by the system appear as a **somestructure of file** – where **somestructure** is determined by the way in which file names are defined and used. Just what the structure is depends on the system, but in almost all systems, the same solution is adopted : a tree of structures, with the files as leaves. Each structure contains information about a collection of items which may be further structures or files. This splits up the job of finding the file into a number of simple consecutive stages, each involving one search through, typically, not too many names. How you use the structure hierarchy is your business, but typically you will use all but one or two of the attributes of the structured name to define names for structures in the hierarchy, reserving the most important item for use as the name of the file itself.

As compared with this organised structure, the files themselves are – so far as the system is concerned – amorphous. A file may contain anything at all; the details are generally regarded as not the operating system's business. A good case can nevertheless be made for knowing something about the information in the file, for protection against doing silly things with it – like printing code files. This brings us back to the idea of a *file type*.

In order to find the files we want, there must be a **file table** of some sort, generally called the **directory**, with an entry for each file. The entry can be found given the file name, and from the entry it must be possible to find all the system knows about the file – the file attributes, the file data, and any other components which might be defined. Where should we keep the directory ? The obvious answer is : on the disc, with the files. We need it when we have the files, and to rely on its preservation somewhere else is asking for trouble. In early systems with small and fixed disc configurations, there was some latitude, and it was sometimes convenient to collect the directory information onto one disc (perhaps to load it all into memory easily). As the systems have grown and become more flexible, it has become more important to keep directories on the discs occupied by the files they describe; that is particularly obvious in the case of removable discs, but in any case can be seen as a safety device, allowing some processing to continue even if one disc is out of action. Most systems in fact do just that.

There is another, rather less obvious, answer : don't keep the directory. Instead, store enough information with the files themselves to reconstruct the directory when the disc is mounted. The main advantage of such a system is robustness. There is no danger of corrupting the directory and thereby losing all the files on the disc; any corruption will only affect the files on the parts of the medium actually spoilt. Against that, the system must read all the way through the disc to build the directory before it can begin to use the files, and as disc pack capacities soar up into the gigabytes and beyond this can take a very long time. Some systems have been built this way, but we don't know of any in current production.

The directory organisation must cater for the file name structure – though it needn't mirror the name structure. The essential feature is that the directory structure should lend itself to a safe and efficient implementation of the functions of the names. We identified those in *NAMES OF FILES* as *description* and *classification*, and suggested that we wanted a file to be identified by a collection of names, each describing some aspect of the file's reason for existence. Most commonly, files are organised hierarchically, and so are directories. A directory contains pointers, each of which may point to a file or to another directory. Clearly, to use this structure it is necessary that the directories must say whether each of the names they contain denotes a file or a directory, but that's just another

file attribute. Within the structure, each file and each directory has a name; reading the names sequentially from the lowest directory to a file defines the full pathname of the file. This structure satisfies our requirements for names reasonably well.

Recall the pathname of the file we mentioned in the *FILES* chapter : `...:Current:340:340 Files`. It happens that in the current Macintosh system the early parts of this compound name are implemented by a directory structure, while the last item is simply a name from a list of names. `340 Files` is a name which appears in a directory called `340`; `340` is a name which appears in a directory called `Current`; and so on. Unix provides an essentially identical structure; indeed, so does MS-DOS, though as it restricts each component of the name to at most eight characters in length (with an optional three-character "extension") the choice of names in that system is more restricted. A big advantage of this structure is that if you are working with a number of files which share some attributes – typically because they are all connected with the same job – you can choose the file names so that they all live in a set of directories which form a compact subtree of the whole file structure. Most traditional systems take advantage of that by providing means to fix attention on a particular directory, from which all file names are thereafter assumed to start, thus relieving you of the need to specify many parts of the name every time you want to use a file. This operation is sometimes called "attaching to a directory" or "setting the working directory".

The directory structure can be used for other purposes too. In most operating systems designed to be shared between different people, it is the basis of the file protection and security systems. Each person is associated with one directory of the tree, and has uncontrolled access only to the subtree which grows from that directory. To share files, some sort of access to other directories, including operating system directories, must be possible, but is carefully guarded to prevent unauthorised access.

Though most systems use a hierarchy of directories, it was not always so; other implementations are possible, and indeed the double appearance of `340` in the pathname used as an example above stems from a time when the Macintosh implementation was quite different. The first Macintosh system provided "folders" as a way of grouping files, but they were not directories. They had no real existence, and were more like file attributes. When asked to display the folder `340`, the Macintosh Finder would inspect all the files in the appropriate disc, and display those tagged as belonging to the 340 folder. This became abundantly clear on trying to store two files each called `Assignment 1` (one for 340 and one for 360). (After all, there really were two real folders on the real desk, each containing a real file called Assignment 1 ! So much for the system illusion !) So far as the system was concerned, they both had the same name – so the course number was included to ensure that they were distinct. This curious (and, we think, unique) way of implementing grouping simply didn't work as bigger discs became available, and was later replaced with a conventional hierarchic file system.

LINKS.

The directory structure can also be bent to cater for multiply named files. Recall the example we mentioned in the *NAMES OF FILES* chapter, where we wanted a file to have two names, `...:Peter:AI` and `..:AI:Peter`. To cater for this requirement, some systems allow a file to be entered with more than one name, so that it appears in two or more file directories; such names can be thought of as *file pointers*. (We mentioned them in the chapter *MAKING LIFE EASIER* as an example of a service provided by the system to make the system easier to use rather than to provide any specific new function.)

From the outside, the two (or more) routes to the file must look like ordinary directory entries, but that says little about the means of implementation. In practice, there is no standard position on the semantics of file pointers, and they come in two varieties. In Unix, a file pointer can be made by a **ln** instruction, and is set up to be precisely equivalent to the original – so if the original is deleted, the pointer constructed by the **ln** remains valid. This is called a *hard link*. The file itself cannot be deleted until all links to it have been cancelled in one way or another, so there must be some provision to count the links as they are made. In the Macintosh system, a file pointer is called an alias, but remains inferior – if the file is deleted, the alias stays there, but you can't open it. (The

same happens if you make an alias to an alias, and delete the first alias; the second-order alias no longer identifies the original.) These are *soft links*; the alias does not point at the file itself, but at the directory entry.

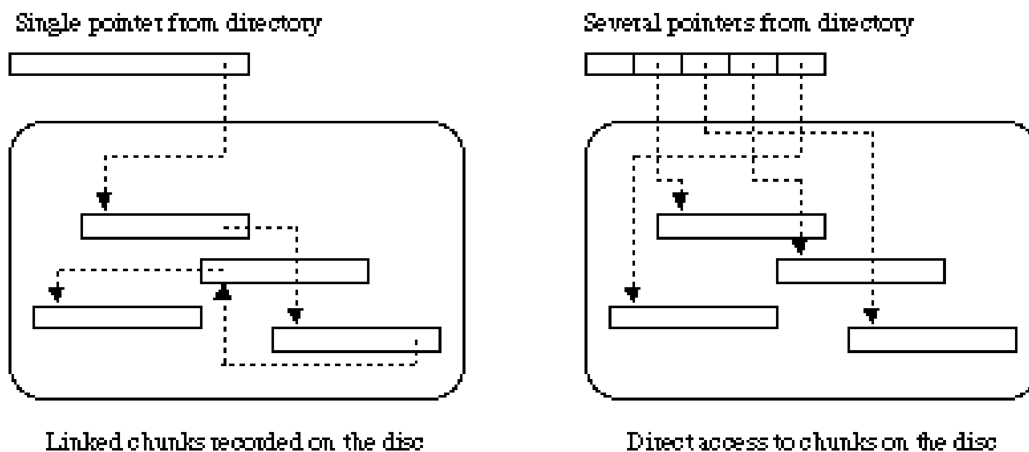
BUILDING THE FILE SYSTEM.

Perhaps the easiest way to implement the directory system is to store each subdirectory as a separate file, and many systems use this approach. It's a very uniform system, in which each step – whether working through the directories or having found the desired file – is the same : find a name in the directory, and go and open the file it identifies. There is a cost for the simplicity : it can be very expensive in operation, as a search through a few directories will require opening and closing many files. An alternative is to store the whole directory as a unit, with the contents organised as a tree (or other appropriate) structure. That makes the administration a little more complicated, but potentially very much faster.

It also makes the disc more fragile, in that damage to the directory can destroy the links to all the files. For that reason, many systems adopt a policy of both maintaining a conventional directory structure and keeping information in the files which can be used to identify them if the directory is lost.

Some sort of machinery must be provided to reach the file data from the directory entry. Clearly, this must be based on the disc addresses at which the file data are stored. In old systems, it was necessary to state when first opening a file how much disc space would be required; the system would then reserve exactly that amount of space as a contiguous area of the disc, and put the address of the first sector in the directory. This is obviously less than satisfactory unless you always know exactly how big files will be, which in practice is rather unusual. Modern systems therefore always permit files to grow or shrink as required by the programmes which use them, but to do so must be able to store the files as several separate chunks. These chunks must be linked by structure provided by the system, and there are several common methods.

In a comparatively simple method, file areas are chained together, with each chunk terminating in a pointer to the next chunk; this is very flexible, easily extensible, and very good for files intended to be used sequentially, but very bad for random access. To improve the random access performance, it must be possible to identify the required chunk quickly, without reading through the file from the beginning, which means that the structural information must be readily available. For example, it can be held in the directory as a set of pointers to the different areas, and there must be some means to identify the right pointer : either record numbers (or something similar) can be associated with the pointers, or every chunk might be constrained to have the same size. This method is quite flexible, it works for sequential files, and it is much better for random access. The two methods are illustrated in the diagrams below.



Unix uses a variant of the method shown in the right-hand diagram, with some of the pointers referring to further levels of pointer arrays, so that very large files can be incorporated in the system if necessary.

Many other modifications are possible. For example, some systems also maintain a pointer from the directory to the end of the file; this makes it very easy to implement an **open at end** variant of the **open** instruction, useful for files which keep historical records of any sort of activity.

USING FILES.

Having the files on the disc is one thing, but they're no good if they just lie there. The whole point of having a computer is to do things using the files, so we shall now consider the implications of this requirement on the operation of the file system.

Certain **file operations** must be available on request. A selection of these, such as the standard example **rename**, are operations on the directory with possible side effects on the file attributes, and can be handled completely within the directory structure without reference to the file data. We've discussed these earlier; they're included here for completeness, and to remind you that the system must be able to make the operations work using the structures we're describing. Given the directory structure, the details are fairly obvious. Two examples which do affect the data, but which can be performed with no knowledge about the details of the file, are **copy** and **delete**; once again, the necessary mechanism is obvious enough, though both require services concerned with managing disc space : we discuss these in the next chapter.

The operations **open** and **close** are rather more demanding. All **open** and **close** instructions involve interactions between the device concerned and structures in memory, such as the file information block and file buffer, and we have already discussed these. In the case of disc files, they might also involve operations on the directories – and, in particular, extending the directories to accommodate new files. While there is nothing very mysterious about the procedures which must be followed – once again, they follow from the structures defined – there are differences between the details adopted by different systems.

An interesting divergence of views is seen in the sequence of operations adopted. Some systems construct a new directory entry as soon as a new file is opened; others do not extend the directory until the file is closed. That's possible, because – as we have already pointed out – the file information block contains all we need in order to use the file, so if that is correctly set up we don't need a directory entry. Proceeding in this way, it is easy to use temporary files without the overhead of changing the directory. (In Unix, files are linked into directories when opened; nevertheless, you can achieve the same effect, but without avoiding the cost, by opening a new file and then immediately unlinking it. This is not a pretty way of doing things, but it works.) The disadvantage is that a file is not protected against system failure unless it is linked into the directory, so that a new file made without linking can be lost if anything goes wrong and the **close** instruction isn't executed.

What will we do when we no longer have a distinct disc file system, but instead simply regard the disc as our permanent memory as we suggested in *VIRTUAL MEMORY* ? It seems likely that it won't make a lot of difference at the higher levels of organisation. Whatever happens, we'll need file tables of some sort to find stored material, because we can't reliably remember long numeric addresses. Also, most of the file attributes we use now will still be useful, so that bit doesn't change. The only new feature will be material which we can use to map the data straight into the virtual memory (until we get memories which can hold 2^{64} bytes, and that will be a little while yet – work it out), and that will be something like a piece out of one of the addressing tables we discussed in the context of virtual memory. It begins to sound almost plausible.

DISC DIRECTORIES AND SYSTEM DIRECTORIES.

We have been discussing the disc directory so far as though it was all that we needed to consider, but in practice this is not so. We have already mentioned several cases which

demonstrate this fact : earlier in this chapter, we remarked on the desirability that each disc should have its own directory; in the chapter *DEVICES* we introduced the idea of *mounting* additional directories or devices in an existing directory; in the chapter *DISTRIBUTED SYSTEMS*, we argued that the file table must be able to cope with files distributed throughout a network, and with the appearance and disappearance – perhaps in different places at different times – of parts of the file system. Clearly, there is more to say about the directory.

Or, to be more precise, there *should* be more to say about the directory, but we can get away without it. The cost of doing so is that everyone who uses anything but the simplest computer system must be aware of some details of the file implementation. For example, the file into which this chapter is being entered is called **C:\courses\340\book\implem.bts**. (Yes, it is not the same system as was used for *NAMES OF FILES* : versatile people, we. This is Microsoft Windows 3.1, which lives on top of MS-DOS.) The "C:" part of the name identifies the disc drive currently occupied by the disc which carries the file. Notice that's the name of the *disc drive*, not the name of the *disc* : so the name of the file is in part determined by which drive you happen to be using, and, with a removable disc, might change from session to session. The name of the disc itself, which is at least fairly constant, doesn't appear. (In the example, "C:" is the internal hard disc, so is rather special, but the principle is clear.)

Identification by disc drive is easy, and common, and has been used for a long time in many systems. The Macintosh system is slightly more accommodating : different discs are identified by name, but the systems are still separate. (We have already seen in *DEFINING A SYSTEM INTERFACE* that this can lead to confusion, as the distinction affects the semantics of operations such as dragging icons from directory to directory.) In each of these cases, we avoid facing the question of how to organise a single universal file directory by treating each device separately. The Unix notion of mounting one directory as a subtree of another does produce a single directory structure – but it is arbitrary, in that a file name depends on just where in the tree the mounting is effected. When used for distributed systems, it is common to mount remote discs as subtrees of local discs, so that file names change depending on where you are. The Unix directory encompasses more than a single device, so either it must be based in primary memory, or we must be willing to insert links into the disc directories which identify objects outside the disc. Unix keeps the information in the disc directories, and works out which disc to use from a set of *mount tables*, which identify the physical storage media to be used at each point.

We see that the disc directories we have been discussing are essential as maps which let us find our way about on the disc, but they are not necessarily the structures which we really want to see in the user interface. Whatever these structures are, though, it seems likely that our basic device-handling model will be able to cope. Provided that we know what we want, the details of management can be encoded in the device's own procedures accessible from the device table, and will automatically be used as required.

This is true even if we use devices which don't strictly exist in fact. A good example of this is the (increasingly common) use of network file servers to provide distributed file services to many individual computers. In such a system, "the disc" might in fact be many separate discs which are distributed over a wide area, and administered by local software which knows just enough to direct enquiries by remote procedure calls or other means to the correct destination for service. In this case, the procedures in the device table will be those required to pass on the information to the local software, and to receive results from there as they are returned.

The Computer Science department's Unix system is operated, in part, as a distributed system. If you are a favoured person, you can log in to any Unix machine and the system looks much the same. The trick is worked by the *Network Information Service* (NIS), which acts as a name server, maintaining a central list of names (including userdata, aliases, and so on) to which each processor can refer if a name is not found locally.

Controlling disc usage might also be an important function. The topics discussed so far in this section – structure, file directory, and operations implemented – all have their counterparts in the other disc areas described in the next chapter, but an additional dimension of control is necessary in the ordinary file system because it is here that the file system most closely approaches the user interface – and, of course, the user. In consequence, the ordinary file system cannot be assumed to operate within rational and reasonably predictable limits, and, particularly in shared systems, it might be desirable to incorporate measures which will encourage people to use the file system sensibly.

A major problem with many systems is that of ever-increasing demand for space. We do it; you probably do it too. If you regularly use a computer system in any but a purely routine manner, it is very likely that the volume of disc files you save, like the entropy of an isolated system, increases with time. That's because anyone who uses a computer very soon finds out that it's easier – and a lot safer – *not* to delete files, ever. We all have a magpie instinct, and we are all well aware of the sheer labour involved in reconstructing a file once it has been deleted. Therefore, unless your computer manager is happy to go on buying more disc ad infinitum, people must be encouraged to tidy up their disc file holdings from time to time. There are two approaches, probably best used in combination :

STICKS :

Disc quotas – limits on how much disc you can use when logged out, and maybe when logged in. You can't log out if you've too much disc.
Charging – make people pay rent for their disc usage.

CARROTS :

Provide alternative cheap storage – automatic or discretionary **archiving**.

One other intriguing possibility deserves mention : do nothing. Disc space is getting cheaper quickly, and with the immediate prospect of optical discs offering enormous increases in storage density it is quite likely that one or two small discs could record all that anyone is likely to write in a lifetime, so you need never delete anything. Of course, you have to be able to find it somehow; and past experience suggests that it will not be long before people find ways of using up the new memory resources much quicker than one could expect on the basis of current practice. Think about high-definition full-colour volume images.

Generally, the more (useful) information is available to the operating system, the more helpful it can be. Early operating systems sometimes carried a lot of information about their files – we mentioned earlier that the Burroughs (pre-Unisys) MCP system carried well over 100 attributes for every file. (That was partly because they followed the empirical "definition" of a file as anything which could be read or written, so their collection included all sorts of details about transmission lines and other streams which were quite irrelevant to media like discs, and vice versa.) Systems to run on minicomputers and microcomputers went to the other extreme, and aimed to be simple at all costs. Their file systems commonly were (or are) only file systems, without any compromise in the direction of streams – particularly with microcomputer systems, streams are frequently dealt with outside the disc file system. In the file systems, they often threw out the baby with (in some cases, instead of) the bathwater, and ended up knowing only that a file was an array of bytes beginning at a certain disc address. This approach has the advantage of unifying the two "definitions" of a file – much as one might unify a cow and an earthworm by chopping off any part of one which cannot be found in the other. While merely accumulating a lot of information doesn't ensure that the file system will improve, it can't make use of information which it doesn't have.

EXAMPLE : Sprite and NFS (Network File Server) are two file servers, developed primarily for use in distributed systems[IMP18]*. Sprite knows the mode*

(read or write) of any open file, but NFS doesn't. Because of that, Sprite can operate more efficiently when it closes files, or in deciding policy for caching; it can also better guarantee consistency in some cases in which a file is simultaneously open for reading and writing.

There is some evidence of a trend back to keeping more information about files in the operating system. One example is a proposal to maintain a hierarchy of file types, which could then be used by a "knowledge-based" operating system to avoid using files in silly ways.

REFERENCE.

IMP18 : V. Srinivasan, J.C. Mogul : "Spritely NFS : experiments with cache-consistency protocols", *Operating Systems Review* **23#5**, 45 (1989).

QUESTIONS.

Work through the design of the file system representing a 2^{64}-byte address space in a little more detail. What (if anything) happens when you open and close the file ? How will you manage file sharing between different people ? Do you need any precaution against using the same addresses twice ? (Probability arguments won't do !)

How do the different sorts of distributed file system fit in with the pattern of file management we described in the *DEVICES* chapter ? (Work through the events which must happen when a file is opened.)

Consider the example of Sprite and NFS. How does it work ? Think of other cases where more information could give more efficient operation.

Consider the two "sticks" – disc quotas and charging rent. How would you implement them in an operating system ? Would it be possible to add them, securely, to an existing system, or would it be necessary to change the system ?