PROPERTIES OF FILES

We can't use a data structure without knowing its properties, and the most obvious evidence of this in computer programmes is the appearance of structure declarations, in which the composition of the data structure is defined. Structure declarations as such are not commonly found in the context of file systems, but the consequent properties of each file – most obviously the size of the structure and the access mode it requires – are more prominent. What is there to know about a file ? Generally, anything that makes sense for a collection of information which exists independently of a programme.

Just what that is depends to some extent on the nature of the information itself. A picture might be represented as an $m \times n$ array of records, each containing whatever information is needed to define the colour of a small area; a computer code file might be just a long sequence of bytes with no internal structure; a textual document might be a sequence of text strings, each associated with information on layout, typeface, and so on.

Just how much of this structure is the concern of the operating system is a matter of system design. Generally speaking, the more the system knows about a file, the better the service it can give. On the other hand, the traditional view of files has been that the structure of the data carried by a file is none of the system's business; we'll see shortly (*FILES – FROM THE BEGINNING*) that the stack of punched cards formed the common metaphor for a file, and that any more detailed structure was by implication the programme's responsibility. In effect, the file system was designed to store long thin things; if you were so perverse as to wish to store any awkward or knobbly data structures, then it was your fault, and up to you to find a way to transform the awkward and knobbly things into long thin things.

We don't know that anyone questioned this view at the time, though the importance of more complex data structures was well recognised – most notably, perhaps, in the tree-structured data declarations provided in Cobol. (The early "scientific" languages – Fortran and Algol – provided no data structures except multi-dimensioned arrays; Lisp went to the other extreme, but people didn't seem to want to save Lisp structures.) The Cobol structure declarations were therefore carefully designed to imply a mapping of the structure onto a linear record. Here's an example[SUP1] :

```
01 THE-CARD.
     02 NAME PICTURE X(15).
     02 ADDRESS-1 PICTURE X(15).
     02 ADDRESS-2 PICTURE X(15).
     02 ADDRESS-3 PICTURE X(15).
     02 REF-NUM PICTURE 9(5).
     02 SALARY PICTURE 9(5)V99.
```

The level numbers (`01`, `02`) denote nesting, so the items with names labelled `02` are thereby identified as components of the `01`-labelled item `THE-CARD`. The `PICTURE`s identify both the items' types (`X` means any character, `9` means a numeric character, `V` shows the position of the decimal point) and the amounts of memory they require. Nested items can themselves be nested, so, as the maximum level number was 49, quite elaborate tree structures could be defined. From the point of view of the file system, though, only the outermost name was significant; it had to deal only with `THE-CARD`, which is – as required – a long thin thing.

That's not too bad, and works for any fixed tree structure. But what if your data are of variable size, or have different structures ? Then its up to you to spend time writing code to decompose your structures into linear pieces, and to store those in the file together with any information you need to show how they were tied together, and then to write more code to read these items from the file and reconstruct the original structure.

Is that really helping us to get our work done ? Only in the sense that giving you a bicycle and a map is helping you to travel from Auckland to Wellington : you can do it, but it's hard work, and things might well go wrong. In practice, the difficulties in travelling from Auckland to Wellington are well understood, and specialised devices

(buses, trains, aeroplanes) have been provided to make it easier*. Similarly, it isn't forbiddingly difficult to devise a generally applicable means of storing an arbitrary data structure which can support automatic storage and retrieval, the details of which can be handled (like all other transactions between programme and file) by cooperation between operating system and compiler.

Such systems haven't developed for general use, and we don't propose to flog this particular dead horse any longer, but our point is that the file systems which are found in current use have not really been designed to make it easy for us to get our work done; they've been designed to make it *possible* for us to get our work done, but only provide something close to the simplest possible implementation with which we can use the storage devices. (There has been some work on systems which can support several different types of file[SUP2]. The idea is to use the primitive file systems to support a collection of more specialised file types, which you could use as required.)

WHAT HAPPENS IN PRACTICE : FILE ATTRIBUTES.

Our top-down analysis has led us to a position which is not that found in the common operating systems. That doesn't mean that someone's right and someone's wrong, but rather reflects a difference between systematic design and development by evolution. (We will even admit, under pressure, that there are other ways to work through the systematic design, and that they don't all give the same result; if you prefer another view, we wish you well, but we hope that you know the principles behind it and why they lead to your favoured conclusion.)

In practice, many operating systems preserve a selection of properties, called *file attributes*, for each file. Some examples :

Attributes of a file as a thing in the operating system :

Owner; When first made; When last used; How to find it; Protection and security information.

These are attributes which don't depend on the special nature of files, so could describe anything known to the operating system. In practice, they are most prominent with files.

Attributes of a file as a chunk of information :

Size; What sort of information (code, text, picture).

These are gross features of the file – they don't say anything detailed about the data in the file. Some of this information is commonly summarised as the "file type", but the precise meaning of that term isn't very well defined.

Attributes of a file as a data structure :

Record length; Number of records; File organisation; Fixed or variable length records.

These attributes tell us about the internal structure of the data. While in principle we could extend this information to cover all sorts of data structure, in practice it is rare to find files regarded as anything more complicated than arrays of records. The first two attributes are therefore equivalent to the dimensions of a two-dimensional array. The others show how the records are used – simple file, indexed sequential, etc.

---

* - Analogies are never perfect; if they were, they would be identical with the items to which they were analogous, so there wouldn't be much point. In this case, you should ignore details such as the enjoyment of cycling, its beneficent effects on the environment and your health, and who wants to go to Wellington anyway ?

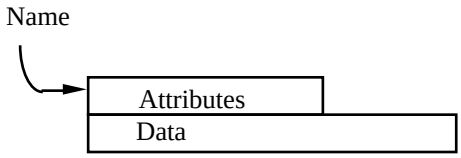
The order of presentation above is, roughly, the order of likelihood of finding the attributes listed in an operating system. The likelihood isn't constant, but depends – also roughly – on the date of the operating system. Early systems saved lots of information about files in the interests of efficient implementation; later, when the cost of storage had decreased significantly, efficiency became less important, and less information was necessary; later still, as the importance of making systems safe and easy to use was recognised, the amount of information increased again.

GENERALISING.

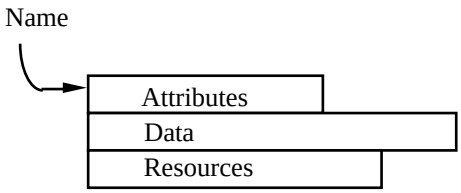
It seems that the operating system will need to know about two sorts of information for each file : the file attributes, and the data stored in the file. Is that all, or should we be able to handle more components ? While most systems manage with the two (and, indeed, some simple systems reduce the attributes to the bare minimum of how to find the file), there are exceptions.

The Macintosh file system is an example[SUP3]. A Macintosh file has three components : attributes, data, and resources. The resources identify things which will be required when the file is used, but which are usually considered to be in the nature of standard parameters – as opposed to the data, which may be read and written as usual. This additional dimension to the file system requires appropriate supporting software, so special procedures to handle the "resource fork" are supplied in the Macintosh system, and there is a "resource editor" to manipulate resources in various ways. Here's a diagram comparing the Macintosh and conventional file structures :

CONVENTIONAL

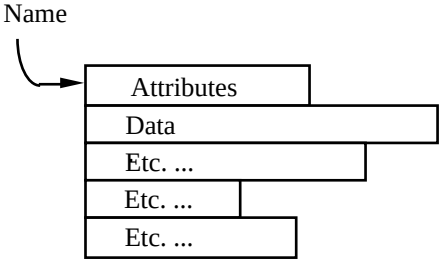


MACINTOSH



Does that exhaust the possibilities ? Probably not, though we can't think of a realistic and more elaborate example. (One could speculate about the access control list.) To be safe, we should regard a file as a collection of an arbitrary number of separate, though related, sets of information, which we shall call *components*. A structure of this sort would be not too difficult to implement in a rather general way, which would leave room for expansions of many sorts. (It begins to sound distinctly like an object, as in object-oriented systems – particularly as resources can include executable code. You can think of that either as an amusing accident, or as an indication that the object model does indeed capture something fairly fundamental about the nature of computing.) In terms of the diagram above, it would look something like this :

GENERALLY ?



We urge that we're discussing the principles, not the practical details. Just what appears in the "Etc." areas depends on circumstances – and there's no very obvious reason why it should always be the same for all files in a system. Our point is that we have been led by our design process to suggest that it might be sensible to design systems with provision for such multiple-component files.

The world, as usual, is a bit slow in catching up, but it's getting there. Something almost identical with our structure has been reported[SUP16] as a means of coping with files for high-speed parallel file systems. In these files, the data are organised into several separate components (which the authors unfortunately call forks), and different forks an be used for different purposes.

OPERATIONS ON FILES.

There are approximately two sorts of operation which make sense with files – "approximately" because it depends on how you choose to make the classification, and others might perhaps discern subdivisions which they would promote to the top rank, but we'll stick to two. These are operations which do things to the file as a whole (and to its file table entry), or to its attributes – such as **delete**, or **rename**, or **set protection codes**, or (recalling our ideas on *MAKING LIFE EASIER*) **make a file pointer** – and operations which need to get inside the file to see what's there – such as **read** or **write**. Operations of the first sort are commonly initiated interactively from a terminal, but the others are usually restricted to the inside of a programme. There is more about these different views in the chapters *FILES IN THE SYSTEM* and *STREAMS IN PROGRAMMES*.

Perhaps we could develop a more systematic treatment by looking at the several components of the file we distinguished earlier – so the "operations which need to get inside a file" could be classified according to the component with which they were concerned. We would then classify **set protection codes** as an operation on the attributes rather than as an operation on the file as a whole. The Macintosh system would then have whole file operations, file attribute operations, file data operations, and file resource operations. This certainly leads to an elegant separation of the operations, so that it becomes straightforward to decide where the resource editor fits in, and we believe that it could profitably be elaborated further. For present purposes, though, we shall stick to the more common current view.

In general, the operations on the file as a whole are concerned either with the file attributes, or with entities in the system outside the file. Protection codes are attributes of the file; but to delete or rename the file some operations must be carried out on the list in which the file name is found. The "file as a whole" operations are analogous to the actions which it would be reasonable enough to perform on a cardboard box containing a number of papers, so long as we don't have to read the papers. We can throw it away, relabel it, move it into a different cupboard, send it to be copied, give it to someone else, lock it up for safety, and so on. All these – or, rather, their analogues – are useful operations which we might expect an operating system to make available.

Once we open the box, a whole new set of operations become possible corresponding to operations on the file's contents. These are harder work : we have to take out and read what's there, or write new papers to put in. Perhaps some of the information is wrong, and we want to change it. The material in the box might be just one long string of text (or of anything else, for that matter); or it might have a lot of internal structure, in the form of records, or a tree structure, or something else. How much of this structure can we reasonably expect to be supported by the operating system ? There's no precise answer; computing is not (at this level) subject to laws of nature, and everything is a matter of design. Some systems (Unix) provide only a minimum of operations, typically **read** or **write** the next element in the file, and perhaps retrieve the element at a specified position in the file. Others, particularly systems designed more for commercial use, provide support for defining record structure in the files. Few, if any, allow for any more complex structures.

Even reading and writing are not quite as straightforward as we might think, if we extend the idea to include all the components of the file. The position of the file data is

clear enough : subject to reasonable security conditions, any part of the file data is readable under some circumstances, and must have been writable once. (You will observe that we are choosing our words with some care, for there are many special cases; it might well not be sensible to write in the middle of a file, and some parts of the file might not be comprehensible without other parts.) When we consider the file attributes, though, it is far from clear that even the owner should have free access to them all.

COMPARE :

Lane and Mooney[INT3] : Chapter 12; Silberschatz and Galvin[INT4] : Chapter 10.

REFERENCES.

SUP1 : G.A. Creak : *Cobol using the Stubol compiler* (Auckland University Computer Centre, 1975).

SUP2 : D. Dean, R. Zippel : "Matching data storage to application needs", *Operating Systems Review* **29#1**, 68 (January, 1995).

SUP3 : *Inside Macintosh*, volume 1 (Addison-Wesley, 1985).

SUP16 : D. Kotz, N. Nieuwejaar : "Flexibility and performance of parallel file systems", *Operating Systems Review* **30#2**, 63-73 (April, 1996).

QUESTIONS.

Consider evidence for and against the desirability of file attributes.

Follow the "cardboard box" analogy further. How far does it go ? Does it suggest any new file operations which should be provided by an operating system ?

How could an operating system provide facilities for storing tree structures in files ? Any proposed method should work with any sort of tree structure, should provide for both reading and writing, and should be significantly easier to use than do-it-yourself methods. When you've solved that, think about cyclic structures.
