FILES IN THE SYSTEM

These are general comments about all sorts of file. They can be seen as an abstract view of the file system, or, equivalently, a set of specifications which should be met by all actual file systems. Notice, too, that we do mean *file* systems – we are not talking about streams. These are essentially persistent collections of data which we intend to store for some time. We shall address the question of how streams and files can sensibly be combined into a single useful system later.

As we are not worrying about what sort of file we shall be using, our concern here is to accept some instruction for an operation on a file, and to make sure that is sent, in some appropriate form, to some other part of the system which does know about the nasty practical details. The first task is clearly to decide how the system is to interpret the instructions, and how it is to convert the file specifications they contain into a sequence of actions which will identify the intended file. We shall describe this process as interpreting the file name – but we shall define "file name" as any data or actions which identify a file. A file name might therefore be a string of text, or an icon, or a sequence of selections from collections of items.

USING FILE NAMES.

The basic requirement is the ability to find a requested file, and we have seen two major ways to express the request : we can either provide the name ourselves (typically by typing), or we can select a name from a list (menu or icons) provided by the system. Both of these are common everyday activities. For example, we could go into a bookshop and ask for a copy of *A practical approach to operating systems* by Lane and Mooney – or we could start by asking to see what books on operating systems were in stock. In either case, though, the system must use some sort of *file table* which it maintains, and which holds all the information it has about the files. If the name is explicitly stated, the system will simply search for it in the file table; for a selection method, the system must display those parts of the table which are appropriate for the selection.

FILE TABLES : Do not be confused – that which we call a "file table" is in most systems called something like the "directory". We use the less familiar name for two reasons : first, to emphasise the analogy between the file system's list of files and the tables maintained by many other parts of the operating system; and, second, to avoid implying any sort of commitment to a particular structure until we've designed it.

Once the table entry has been identified, we can in principle find out anything we like about the file – provided that the table contains the information we need, and it isn't hidden from us for security reasons. The table is therefore likely to contain some information in the nature of addresses which we can use to find all the file components we mentioned earlier in this section – attributes and data for most systems, attributes, data, and resources for the Macintosh. In practice, it is common to keep all or some of the attributes in the file table itself, but that doesn't change the principle.

The details must depend on precisely what we want the name to do for us. It isn't self-evident that designers put a lot of thought into this question, though perhaps that's a reflection of the rather good performance, on the whole, of the common sort of organisation. Just to make the point, though, here's a question : can I take my { 340, files } file and just add another descriptive term to the set, making { 340, files, old } ? – and, if I do, will it be listed with { 340, files } so that I can see them both at once ? It's quite likely that your computer's file name structure won't let you do that; though you will probably be able to achieve the same effect by other means, the system will probably not give you any explicit help. The Burroughs MCP file system was constructed so that you could have a file called `somefilename` and another called `somefilename/oldversion`, which was very convenient.

That says something about what's in the table, but we still haven't decided anything about the table itself, except that we must be able to find an entry given the file's name. The obvious way to do that is just to include the filenames in the table, then we can simply search the table until we find the name we want – or that it isn't there. That works well for very small systems with very few files, and was widely used in many monitor systems; but once your system has to look after many thousands of files belonging to different people who want to use structured names of the sort we discussed earlier, it isn't so satisfactory.

In the particular case of a system which works by selection, each display must show a subset of what's available (so you don't go into a bookshop and ask for a list of the complete stock). Even in this case, therefore, *some* information must be supplied with which the system can work out which subset to display. This is commonly supplied by the system to start things off (using Macintosh terminology, you begin with the folders on the desktop), but while using the system you give more information step by step (by opening folders which you can see), narrowing down the selection until you find the file you want. In terms of the relationship between file names and descriptions, this process amounts to a strategy by which we can build up the description item by item until we have constructed the complete specification.

Why is that easier than remembering the complete name ? Because we don't have to remember the order of the components of the name, as they are presented for us one by one; and because we don't have to remember just which words were used in the name ("Did I call it *current* or *present ?*"). It's interesting that in practice we often use textual systems in just the same way; unless we're very sure of the names, we display the contents of directories before committing ourselves, and often climb through a sequence of directories to find a required file.

THE FILE TABLE.

Once the need for a permanent file table is established, we can start to work out what it should contain. Some obvious suggestions have appeared in the previous remarks, but we can make the idea more precise : the file table is the system's only way of reaching the file, so quite generally all information about the file should be accessible in, or through, the file table.

This includes the file attributes, which we have regarded as a part of the file. Should these be kept with the file, or in the file table ? The quick answer is that it doesn't matter. As in most parts of computing, there is no law of nature which dictates that certain things can or cannot be done, and we are left without any clear criterion which will help us to decide what "should" be done. In fact, we are straddling the line between specification and implementation. Provided that we recognise that the attributes are in principle components of the file (because they must move around with the file), it doesn't much matter how they are implemented. In practice, information which pertains to most files and which it is useful to be able to find quickly is sensibly kept in the file table, while less important information, or information only relevant to files of certain types, is sensibly kept separately – and, until recently, that meant in the body of the file itself, where it was inaccessible to the operating system. The appearance of files with more components, such as the Macintosh system's resource fork, makes a more rational approach possible.

The consequences of trying to keep lots of information about files in a traditional directory are well illustrated by the Burroughs MCP, in which all files had over 100 attributes, all kept in the directory ! This enormous number was necessary because, though many attributes were only appropriate for files of particular sorts, there was nowhere else to put them. The result was that perfectly ordinary disc files had attributes (unused, of course) which only made sense for terminal files, *and vice versa. It might have helped significantly if streams and files had been distinguished.*

We already have some candidates for inclusion in the table. Several possibilities are listed in the chapter *FILES*. There might also be information specific to the device on which the file is stored; actual operating systems are not usually as careful as we in distinguishing such accidental details of storage from information about the file proper. More will turn up later; for example, while discussing file protection and security, we shall find several items of information which can well be kept in the table : whether the file has been changed since the last backup, whether the file has been deleted, pointers to several versions of the file for a system implementing file generations, information for protection codes or an access control list.

Clearly, there is no shortage of material. Equally clearly, trying to keep all of it might not be wise. Just what you want to preserve depends on the system facilities which you think should be offered, and that's a decision you should make when you design the system.

OPERATIONS ON WHOLE FILES.

If we are to pursue the idea of a file as a data type which we advocated in the chapter *FILES – FROM THE BEGINNING*, we cannot stop with data structures. A data type includes not only structures but also the operations appropriate to objects of the type; so we must ask what operations can be performed on files. This idea was introduced in the chapter *FILES*; here we shall elaborate on those operations which can sensibly be applied to the whole file. That's as opposed to operations on details of the file data, such as editing text or changing database records; those are not usually considered to be the responsibility of the operating system. On the other hand, finding and starting up the editor or database system are matters for the operating system, as are some other operations which concern the file as a whole.

A reminder is in order before going on. We commented earlier that implementation details inevitably begin to intrude when we try to speak generally about operations on files. Looking forward a little, one of the advantages of magnetic disc as a file storage medium is that it lends itself well to implementing all the file operations we want; but the same cannot necessarily be said about all media. Take the following discussion, therefore, as a description of what we would *like* our system to do; but be ready to compromise for different file media.

There are a number of operations on files which can be managed without any concern for the nature of the material held in the file, and are therefore applicable in principle to any file whatever. Examples are **delete** and **rename**, and operations which affect the file protection and security properties. These operations might manipulate the file table itself, or (subject to obvious restrictions) the file attributes. For example, to delete a file it is necessary to remove its entry from the file table display, while to rename it the table entry must be changed.

Rather few universally applicable operations on the data are likely to be achievable, because few can make sense without any information about the structure of the data. Two obvious awkward examples are **print** and **execute**. One or two easy cases remain : **copy** is the most obvious of these, for there is no need for the system to know anything about the structure of the file body to complete the operation – provided that the new copy contains the same pattern of bits as the old, the copy has been accomplished.

The notion that it should be possible to use the same set of operations on any file is nevertheless attractive; it fits in well with the aim of making the system model simple and straightforward. How can this be extended to files and operations in general ?

OBJECT-BASED SYSTEMS.

From the considerations put forward in the previous paragraph, it is clear that to put such a scheme into practice on any but the very limited scale mentioned there requires that much more information be available to the system. Generally, if it is proposed that any operation be universally available, then the system must know how to perform that operation on *any* sort of file.

Clearly, it is impossible to build that information into the system, as new sorts of file may be invented at any time, and our earlier comment on the Burroughs MCP illustrates another danger. On the other hand, it *is* possible to associate the information with the file. Suppose that we wish the operation **print** to apply to every sort of file. What we must do is decide on the mechanism which the system will use to handle a **print** request, decide what specific information will be needed with each file, then require that every file in the system must provide the required information in some suitable standard form.

As a general statement, that works well; as a pattern for implementation, it poses certain problems. The major problem is a consequence of the diversity of sorts of file. If you devise a file to hold a certain sort of information, your primary interest is to make it hold the information effectively, and there is absolutely no reason why the file you design for that purpose should be even remotely appropriate for printing without a great deal of work. You might well have means of your own for displaying the file somehow, but short of giving the system your own printing package there might be no practicable way to tell it how to manage the layout.

That points to the answer. If the only way to satisfy the requirements is to use your own printing package, then that's what must happen. Our specification becomes almost trivial from the system's point of view : it must be enough just to issue a **print** instruction for the file, and *all* the necessary information must be associated with the file.

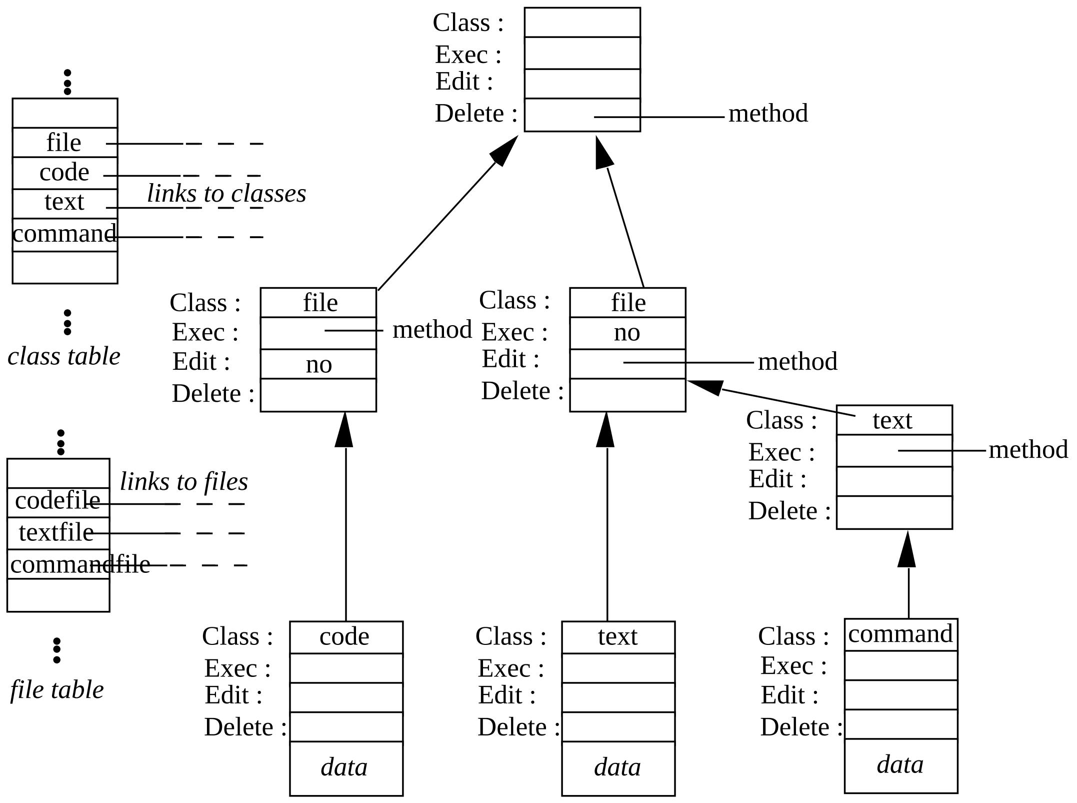
We have invented an *object-based* system. We have seen them called object-oriented systems, but prefer the less emphatic term if only because there is no necessary connection of these systems with object-oriented languages; there's a rather closer link with the object-based interface style we discussed in the chapter *PEOPLE TALKING TO COMPUTERS*, but still no essential dependence. Related ideas are involved in all these cases, but the connection remains on the level of ideas, not of implementation.

In such a system, everything is regarded as an *object*, where that term is used in a rather more technical sense than usual (and has nothing to do with the objects we met when discussing protection and security). Generally, an object can include information, thereby resembling a file, or it can cause actions, resembling a programme, or it can – and commonly does – combine both properties, as our new sort of file does. An important principle of an object-based system is that each object knows what can happen to it, and how it should be done, so to cause some operation to be performed on an object one sends it a message requesting the required action, and the object does the rest – just as we hope to send a **print** message to any sort of file, and expect the file to know how to do it. The object's knowledge of what to do in response to a message is commonly called a *method*. Perhaps now it is easier to see the force of our earlier observation that files constructed according to our new proposal were looking very like objects. They are very well suited to carrying all these bits and pieces, and permitting expansion in unforeseen directions to cope with unforeseen requirements.

Another feature of object-based systems is important to make this work in practice. While it is up to each object to know how to print itself, large groups of objects are likely to share methods. That being so, it's silly to provide every object with full instructions on how to print itself; instead, we can collect the similar objects together into a *class*, tell the class about the method, and provide facilities to link the objects to their parent class. We say that the objects *inherit* the printing method from the class. Equivalently, we can say that every object has a method for every action. The method can be explicitly defined in the object, but is otherwise automatically defined as "ask the class for instructions". The idea of organising the objects into a hierarchy of classes is characteristic of object-based systems. It requires a certain amount of additional organisation within the system, but is very powerful.

Here's a mythical example to illustrate the principle involved. Suppose we have three files – a text file, a code file, and a command file. The text file can be edited; the code file can only be executed; the command file can be edited and executed, but the execution of the command file is quite different from that of the code file; and all the files can be deleted.

The diagram below illustrates how files of different types might share methods by inheritance. All files are deleted in the same way, so the delete operation appears as a property of the universal class file; all text files are printed similarly, but code files cannot sensibly be printed, so the print method must appear lower in the hierarchy; different methods of execution must be used for different sorts of executable file, so code files and command files have separate execution methods.



Don't get carried away by the structure – it's a means to an end. The important feature of an object-based file system isn't the classification or inheritance; it's that the methods are associated directly with the files. The classification and inheritance are (very effective) means of implementing this important feature. The class is an entity with which we can associate any properties common to all its members, but different from those of any broader group. These will commonly be properties which depend on the identifying features of the class – so, in this case, they will be properties of (for example) text files which are not shared by files of any other sort. It is certainly plausible that an editing method might fall into this category, as the facilities we need to edit text are not the same as those required to edit pictures, or code, or databases, or other material which can be stored in files.

Notice that all files have methods for all functions – even if some of the methods (marked "no" in the diagram) are error reports, or something of the sort. If you want to execute a code file, the system follows the class tree upwards and eventually finds an execute method – presumably the system's standard procedure for loading and executing a code file – at the code level. Similarly, if you want to execute the command file, the system will find an appropriate, though different, method (the command file interpreter) at the command level. If you want to execute a text file, though, the system finds at the text level that you can't. But if you now make a very special file which has all the characteristics of a text file, but for which execution has some plausible meaning, you can identify it as a text file but give it its very own execution method.

As well as being easier to think about, it's practically useful, as it makes life much easier when you want to install, say, a new editor – you only have to change one thing instead of every file. If the new editor is incompatible with the old, then you can preserve the current structure by introducing a new class with the new editor between the text and file classes, and making this the new standard class for text files. The old files then still find the old editor before the new one, but are otherwise treated in just the same way as the others. And so on.

This is all a good deal more elaborate than the traditional simple file system. Does this apparent complexity really help people using the system ? This is another case where we should remember that the aim of the design is to implement the system metaphor, not to build a simple system. This file organisation simplifies the user interface because it is no longer necessary to remember the details of any procedure used to perform an operation on a specific file. Any plausible instruction can be issued in respect of any file, and, if it's possible, the file itself will "know" how to perform the required action.

We emphasise that this is all still design; the diagram is not intended as a map of how to implement your system – though you might decide that an implementation which sticks rather close to it would be effective. Implementation is still another question, which we shall address later. For the moment, we note that an object-based file system is not the same as an object-based interface; the "select-and-click" interface design can be combined with a quite conventional file system, as in the Macintosh system.

The Macintosh operating system includes some of these features[SUP4]. A small selection of operations – those in the File menu – are in principle applicable to any file, but there is no formal class structure. Instead, the role of a file's class is fulfilled by the programme which made the file in the first place. In effect, every file remembers as one of its attributes the identity of the programme which made it, and calls on that programme for any special service which cannot be provided by the system.

As a final note, it is worth emphasising that there is absolutely no necessary connection between the notions of object-oriented systems and graphical user interfaces. It happens that the two were developed at around the same time, with much of the development going on in the same place (the Xerox Palo Alto Research Centre (PARC)), and that they are in some respects complementary, but each can stand alone without the other.

DON'T FORGET THE SYSTEM METAPHOR.

While discussing file table operations, we remarked that "to delete a file it is necessary to remove its entry from the file table display". Why did we add "display" to that sentence ? It was to give us an excuse for making an important point. Recall, yet again, that we are still discussing design, not implementation, and that the design is still directed at providing service for people who are using the operating system. What do you mean when you say "I want to delete that file" ? You mean that you've finished with it, and don't want to be bothered with it any more. Very rarely, if ever, do you mean that all traces of the file must instantly be destroyed beyond all hope of redemption – and there are several sound implementation reasons for not doing so. For example, there might be another file pointer to the same file, or you might wish to preserve the file table entry, marked "deleted", until other administrative operations on an archive or a remote system have been completed. There is even good reason from the service provision principle for avoiding precipitate action; if we want to provide "undo" operations as we decided in the *PROTECTION* chapter then we had better not get rid of the file. What we must do, though, to support the system metaphor, is ensure that the file disappears from view when it is "deleted". So the important point we mentioned at the beginning of the paragraph is that the primary purpose of this design is not to produce a system which does what it's told to do, but to produce a system which conforms to the system metaphor.

It is curious that we started with the idea of building an operating system which would "produce results as instructed", and now by pursuing that aim we find ourselves building a system which doesn't do what it's told. We assert that there is no contradiction – for the deletion of a file might be a convenience, but, having no effect outside the computer system, is never one of the results of ultimate interest.

What, never ? No, never. What, never ? Well, hardly ever[SUP8]. The only exception is the case when the physical deletion really is of ultimate interest, and that arises in the context of security. If your file was a map of an obscure island with the precise location of a vast hoard of buried treasure clearly marked, or your meticulous plans for getting at the rather closer hoard of treasure in the local bank's strongroom, then circumstances would be different. You might very well hope that "delete" meant not only "delete" but also "overwrite with something else, and for preference grind the surface off the disc", and you would be aggrieved if your competitors, or the police force, were able to find a perfectly good copy of the file merely by typing "undo" in the right place. How can the system provide the appropriate service for such requirements for high security ?

There are several answers : you can provide a different instruction for "real" deletion, or a variant of the same **delete** instruction, or you can offer a "secure mode" in which the security level is higher throughout, or you can implement a different system to provide operations which guarantee security – in effect, you implement a different metaphor to satisfy the different requirements. The important principle is that, in designing the system, these ramifications of the requirements should be recognised and taken into account.

"DEVICE INDEPENDENCE."

An interesting consequence of our top-down approach to design is that we have, more or less automatically, implemented a property which turned up in the evolution of operating systems only after a lot of trouble and pain. Our system so far will provide our defined set of operations on any file – and this means any file on any device, for we have not yet needed to define specific devices. This turns out to be a very useful provision, and was called *device independence* when it eventually found its way into operating systems. Of course, all we have really done is passed the buck to some lower-level procedures, where we must find a way to implement our requirements; but that's what top-down design is all about.

COMPARE :

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

REFERENCE.

SUP8 : W.S. Gilbert : *H.M.S. Pinafore or The Lass That Loved a Sailor* (May 28, 1878) (or try http://diamond.idbsu.edu/GaS/pinafore/libretto.txt)

QUESTIONS.

Think of some other operations on the file as a whole, comparable to delete, copy, and rename. (Example : **ln** in Unix.)

Consider how the file operations can, or can't, be applied to files on other media : magnetic tape, punched cards, optical disc, any others that strike your fancy.

How do the file operations interact with other activities in the system ? Think about backup, archiving, security, accounting.

The first computer files were called files by analogy with office files; in the Macintosh system, the office files became folders. Why ? (We do *not* think that the early computists were thinking of a desktop metaphor.)

Which of the actions of the Macintosh File menu can you apply to a file when the programme which made it isn't available ? Why ?

The Macintosh File menu is very restricted. For example, you could use any text-processing programme to write a Pascal programme, but you can't then choose an action "Compile this Pascal programme". Why not ? How could you implement such an action as a generally available facility ? Would it be easier with a real object-based system ?

Consider our extended definition of "file name". Suppose your system includes a utility which can search through the file system for files with names including a specified text string and then displays as a menu the names of files which satisfy the request. Is the sequence "search for 'files', followed by a selection from the menu" a possible name for `lectures/340/files/1991/current` ? If not, what's wrong with the definition ?

We suggested that **move** and **copy** were file operations which didn't depend on the details of what was in the file. How far is that true when moving or copying files from one type of file system or machine to another ? Can it be done safely ?