

File Systems

File systems provide applications with permanent storage. More than that they organize and protect data, and provide a clean interface to allow manipulation of that data. It's no exaggeration to say that providing a file system is one of the major services of general purpose operating systems, and less general ones as well. (Even the palm pilot has permanent storage).

Files

A file is a persistent, hardware-independent, named, protected collection of bits and a collection of operations that can be executed on them. The access operations generally impose an order on the bits. The attributes attributes define what files are used for.

Persistence implies that the bytes have a meaning that extends in time. Memory used in calculating intermediate results doesn't have that attribute. One wouldn't store the memory used in a computation in a file because it has no long-term use. Because the data in files has this long term significance, files are stored on more permanent media. These days, the most common medium is still magnetic disk, although several others are making bids. Some other media that can contain files are memory, flash memory¹, tapes, CD-ROMS, and more esoteric media. Basically anything that can hold information permanently and be read by a computer has held a file system, or will eventually.

By definition, files are largely medium-independent. The same operations are generally allowed on files regardless of the underlying storage medium. There are obvious exceptions - you can write to a CD-ROM at most once, and there are obvious drawbacks to trying to move to a byte at the front of a tape to the back. In general, however, code that manipulates files on one medium will work on others. This saves a lot of programmer time, as we'll see.

Finally, file systems provide a way to name files. This is a seemingly simple function that turns out to be enormously powerful.² File systems provide ways to name files that span multiple media on the same machine (the UNIX® file system), loosely connected local area networks (the Network File System (NFS)) and even global name spaces (the Andrew File System (AFS)). Providing a name space outside the confines of memory addresses allows processes to share data and communicate.

Because files are outside memory, they are also outside the protection of the memory protection system³. As a result, the file system has to impose ideas of user identity and related privileges on the data.

The File Abstraction

Although the idea of an abstract, named collection of bytes is easy enough to grasp, the Devil is in the details. Despite the fairly simple idea of what a file is, files on different operating systems can be remarkably different. We'll discuss the variations in the file abstraction along the following axes:

- Naming
- Data Structure and Access Patterns
- File Types
- Attributes
- Operations

¹ Flash memory holds the data stored in it even when the power is off. So does core memory, but your generation will only see that in museums.

² Magicians and conjurers have long believed that to know a thing's name gives a person power over that thing. So it is with computers. Naming information and the operations thereon is the heart of computer science.

³ Unless we put them there, like Multics does. Even then the initial permissions on the memory segments are derived from the files themselves.

Naming

A file generally has a name, a string of bits that (usually) correspond to human-readable letters. The operating system defines what characters are valid in file names, and any equivalence classes between them. For example, UNIX allows any byte except hex 0x2f (ASCII for /) to appear in a filename. MS-DOS limits the character set to uppercase letters and a few symbols (letters are converted to uppercase in all file names). AmigaDOS allows you to specify any capitalization, but internally ignores case. "File", "file", and "FiLe" all refer to the same file, although it will appear in directory listings spelled as the creator of the file spelled it.⁴

Beyond the character set, operating systems impose a structure on the names of files. Under UNIX the restraints are minor - filenames can't contain /⁵ Compare this to MS-DOS (and the Windows\d\d systems that basically sit atop them) that requires an 8 character file name and three character extension. (Systems other than MS-DOS have the idea of an extension, or a naming convention for related files.)

Different operating systems depend to differing extents on the structure of the file names. MS-DOS defines an executable file by its extension, while UNIX generally takes it as a hint. Other programs, like compilers, place varying degrees of emphasis on file names. For example gcc uses the file extension to determine which of the languages it supports should be used to compile the source file, although the behavior can be overridden.

File Structure

In its simplest form, often called a *flat file*, a file is a collection of ordered bytes. Some systems, however, place additional structure on files. For example files under some operating systems consist of records, or collections of bytes. Rather than reading single bytes or seeking to arbitrary offsets, files are always accessed in terms of records. The records can be fixed length or variable.

Arranging a file as records implies the existence of a *schema* (or description of the record) either embedded in the file in a manner that the OS can read or separately in the system (perhaps elsewhere in the file system).

Many people think of record-based files as database entries, and that's one common use of them. Another common type of record based file was the card file - a file that was a sequence of 80 character records that was the electronic equivalent of a stack of punched cards, or of printer lines.

Record-based files may display an ordering that is independent of the way the bits are ordered on the underlying storage medium. The internal structure of the file may reflect this possibility and be significantly more complex than a flat file. We will discuss the details of this when we discuss implementation of file systems.

Record-based files impose a structure on the data and allows the operating system to keep that structure intact. The flip-side of that is that record-based files are less flexible. Translation between formats or adding a new format is frequently difficult.

I should note that record-based access is provided by and enforced by the operating system. It's not an application convention (although many applications create the illusion of record-based files in a flat file system.) Record based file systems cannot seek to a specific byte in the file, or ask for a set of bytes that spans a record boundary. Records are the fundamental building block of files in such systems in the same way that bytes are the fundamental building blocks of flat files. (Alternatively, flat files are files with 1 byte unformatted records).

Related to the building blocks of the files is the access method that a file supports. The access methods are an abstraction of the underlying hardware.

A file system that supports only *sequential* access requires all files to be read or written from the first to the last byte, only. Such access methods are appropriate file files residing on a magnetic tape for example. A file that can be accessed in any order supports random access.⁶ Random access is appropriate for

⁴ I think that's confusing behavior, but others disagree with me.

⁵ Historically, some flavors of UNIX have limited filename lengths, but most modern versions do not.

⁶ In *The Devil's DP Dictionary* Stan Kelly-Bootle refers to these access methods as allowing items to be lost or ruined sequentially or in any order.

files on disk or CD-ROM.

Note well the distinction between an application's access pattern and the access method allowed by the OS. An application may read a configuration file that supports random access sequentially. That means that the operating system allows any access pattern, but the application chose to read it sequentially. The reverse cannot be done; a file that supports only sequential access cannot have the bytes read in another order.

Other access methods include *read-only* for unmodifiable files or indexed, for record based files that have been sorted multiple ways (the OS must support that, of course).⁷

File Types

Files have various uses: they hold data for various programs, programs themselves, free-form text intended to be read by humans, and other things. Some operating systems allow the contents of a file to be directly encoded as a file type.

File types can be an explicit piece of information remembered by the operating system (a file attribute - see below) or can be a combination of name, permissions (another attribute - see below) and contents.

How types are encoded in the system and how rigidly type restrictions are enforced control how type-based the file system is. Strongly-typed operating systems require file types to be encoded in every file, and enforce restrictions. As a result files generally have only one function, and their use can be easily controlled. Older business computers, especially mainframes had strongly-typed systems. The advantage is that file use, and the procedures that underly them can be tightly constrained. If files that can be printed as checks can only be created by a few trusted programs, it makes forging a check more difficult. It also makes creating one for a legitimate purpose that the system hasn't been programmed for more difficult.

Other general purpose systems, like UNIX and Windows, rely on a combination of filenames, permissions, and file contents to provide typing. These systems generally only care about determining if a file is *executable* (that is contains a program that can be run) or not, leaving the business of discriminating between data types to the applications that use them. (Applications use similar criteria to differentiate). UNIX considers a file executable if the user has the right to execute the file encoded in the permissions and if the file is correctly formatted as an executable. Formatting is generally checked by a *magic number* in the first few bytes of the file. MS-DOS relies on the file extension and the internal format. Check out the `file` command in UNIX for a list of some of the magic numbers used by modern versions of UNIX.

How intrinsic types are in the file system is a tradeoff between codifying practices in the OS and allowing adaptability.

Permissions and Attributes

Permissions encode the operations allowable on a file and what users⁸ are allowed to perform them. You can think of this as a list of all the possible operations on a file, and what users are allowed to perform them. In practice, the representations are smaller and the listings less exhaustive. Consider the UNIX file permissions: each file has an owning user and group, and the rights to read, write, and execute a file is controlled for each of those entities and other users. For example a file might be readable and writable by its owner, readable by members of its group, and not accessible to other users.

We will talk more about permissions in a few days, but right now, the owner, group and permissions are interesting as attributes of the file. Attributes are meta-data, that is data about the file itself, not the data within it. Permissions are a good example: they control what processes may access the underlying data of the file, but are independent of that data.

some other common attributes are:

- creator

⁷ Again, this is an OS feature, not a simulation by the application. The `read` system calls return the records in sorted order.

⁸ Well, processes acting on behalf on what users. We'll discuss it in a little while.

- owner
- system file flag
- hidden flag
- temporary flag
- creation time
- modification time
- information about the last backup
- lock information
- current size
- maximum size

Meta-data is used for a variety of reasons. Some of it is for human use: a data provenance. Some of it is for internal OS use, for example the backup flags. The ability of the file system to store both data and data about the data is an important aspect of the system. For example the `make` utility would be useless without the meta-data telling the program the relative ages of the files.

File Operations

File operations are a generally simple and intuitive set of operations on files. Not all of these are supported by all file systems. In some file systems, operations are implemented in terms of other file operations. Still, these give a good feel for file operations.

Create:

Create a new file. This may allocate space for the file, or just reserve the name for future action.

Delete:

Delete an existing file. The ability to delete a file is distinct from (but often related to) the ability to modify its contents. Some operating systems use the existence of a file to start a service, so the existence of such a file is its most important attribute, and should therefore be protected.

Open:

This lets the OS know that the current process will be interested in a file soon. In some sense, it's extraneous, but in cases where the file resides on a medium that has significant startup cost (a robotic tape cabinet) not returning the open call until the file is ready for access is a good idea. Your Nachos work should give you an idea about some of the OS set up work that's done here.

Close:

Let the OS know that the process is done with this file, and that the OS can reclaim the resources allocated to manipulating the file. (The data and meta-data are updated, but remain in the file system, of course.) Some systems delay writes or cache data for future reads. Close is an indication to them that pending writes must be flushed and that cached reads can be discarded.

Seek:

Files have a notion of the current byte (or record) of the file that will next be accessed. On files that can be randomly accessed, seek allows the calling process to set the current byte (or record).

Read:

Get some data from the file. The OS may take this opportunity to predict future behavior, collect statistics or do other support work in addition to bringing the data from secondary storage to the processes address space. Reading occurs at the current byte/record.

Write:

Write some data to the file. Strictly this means to change existing data in the file to a new value, but many systems also use the write system call to append data. Like read, there may be secondary actions associated with this. Writing occurs at the current byte/record.

Note that reading and writing may cache data, and that such caches have to be coordinated so that all processes see a consistent version of the file.

Append:

Add data to the end of the file. This shares aspects with write, but includes the idea that the underlying file is changing size. Append means that the OS must increase the allocation of storage to the file. As I mentioned above, some OSes determine whether a write system call causes an append or a write based on the current byte and the length of the buffer written.

Get/Set Attributes:

For those attributes that can be modified directly by users, for example the backup flag, this provides access.

Rename

Change a name of the file in the file system. This may be an operation on the file or a directory depending on the file system.

Memory Mapping Files

As we discussed in the memory management unit, it is sometimes convenient to move a file into memory and access it directly there. The easiest way to do this is to make the file on disk (or whatever) the backing store for a segment (or section of a paged VM space) and let the paging system handle the writes. The big issue here is consistency - when do changes in memory get reflected to the file in secondary storage? The paging system is probably pretty lazy about getting changes out to a file, compared to a file write, but writing a whole page to memory on every memory write is probably too slow.

One solution is to keep data about which files are memory mapped (as an attribute) and make associated file reads from the memory system rather than the file.

Putting Devices in the File System

The simple, powerful semantics of files lend themselves to controlling a variety of resources. For example, terminal input can be thought of as a sequential read-only file, and terminal output as a sequential write-only file. This means that programs can be written to take an input and output file and transparently run interactively or from disk files. Furthermore, by introducing memory-resident sequential files that are written by one process and read by another, called *pipes*, programs can be linked together in arbitrary ways. The OS has to do extra work to make a terminal look like a file, or create a pipe, but the result is a simpler programming model for developers: I/O is file I/O.

UNIX takes this to an extreme - nearly every OS resource has a file system interface. Physical disk drives are a special type of file, so are terminals, modems and printers. All of these can be accessed through the familiar file system interface. Recently, the data of running procedures has been added to the list of data accessible through the file system.

Furthermore, the access control mechanisms of files can be directly applied to prevent unauthorized manipulations of the hardware. Raw disk drives or modem devices can only be accessed by privileged users, and those accesses are controlled by the same system as file accesses. A unified access control system is easier to use and only having one to debug implies that it will be more secure.

All is not a bed of roses, though. Hardware has features that are not easily expressed as file operations. The result is the introduction of extraneous system calls to access special features. In UNIX these system calls are called `ioctl`s, for I/O controls.⁹ Being forced to add this feature implies that the interface is not as versatile as one might like. There are various attempts to refine the interface to be more expressive without breaking the good parts.

Next time - directories and implementation.

⁹ You can tell something about where and when people learned UNIX by how they pronounce this word.