# File Types and Attributes

## 1. Introduction

Many operating systems (including Unix derivatives) attempt to represent all data sources and sinks as files. Files are most commonly discussed as byte streams or one-dimensional arrays, and this is a fair description of much of the file processing done by applications software. But many of those byte-streams contain highly structured data (e.g. a load module, an MPEG-4 video or a database). Some very interesting sources of data (e.g. directories and key-value stores), while serializable, are are not even intended to be processed as byte streams, but with very different APIs. Many files have interesting attributes beyond the data they contain.

This is a brief exploration of the range of different types of data that are commonly gathered under the general heading of *files*.

## 2. Ordinary Files

Even the simplest files are understood by imposing structure and semantics on a binary byte stream:

- A text file is a byte stream, but when we process it we generally break it into lines (delimited by *\n* or *\r\n*), and render it as characters (e.g. according to the ASCII or ISO 8859 character set).
- An archive (e.g. zip or tar) is a single file that contains many others. It is an alternating sequence of headers (that describe the next file in the archive) and data blobs (that are the contents of the described file).
- A load module is similar to an archive (in that it is an alternating sequence of section headers and contents) but the different sections represent different parts of a program (code, initialized data, symbol table, ...).
- an MPEG stream is a sequence of audio, video, frames, containing compressed (e.g. Discrete Cosine Transform) program information, which require considerable processing in order to reconstruct the encoded program.

An ordinary file is just a blob of ones and zeroes. They only have meaning when rendered by a program that understands the underlying data format.

### 2.1 Data Types and Associated Applications

If a file can only be interpreted by a program that understands the *meaning* that has been encoded in the byte stream, our first problem is finding the right program to interpret each file (or byte stream). There are a few general approaches:

- Require the user to specifically invoke the correct command to process the data. This is common in Unix-derived systems:
  - to edit a file you type `vi` *filename*
  - to compile a program you type `gcc` *filename*
- Consult a registry that associates a program with a file type:
  - there may be a system-wide registry that associates programs with file types (e.g. Windows).
  - there may be a program-specific registry that associates programs with file types (e.g. configuring browser plug-ins).
  - the owning program may be an attribute of the file (e.g. classic Mac OS).

A registry solution presupposes that we know what the type of the file is. There are a few approaches to *classing* files:

- The simplest approach is based on file name suffix (e.g. `.c`, `.png`, `.txt`). This may be simply an organizing convention (e.g. in Unix-derived systems) or a hard-rule (in Windows it may be impossible to process a file that has been renamed to the wrong suffix).
- Another common approach (very popular with Unix-derived systems) is a *magic number* at the start of the file. Each type of file (or even file system) begins with a reserved and registered magic number that identifies the file's type. For more information on this approach, look at the file(1) command.
- In systems that support extended attributes the file type can be an attribute of the file, not dependent on a suffix or magic-number registry.

### 2.2 File Structure and Operations

Even ASCII text byte streams have structure, and some streams (e.g. an MPEG-4 video) have a very rich structure. In these cases the file can be viewed as a serialized representation of data that is intended to be viewed by a particular program (e.g. a text editor or video player). It is meaningful to talk about doing *read(2)* and *write(2)* operations on these files. But not all files are intended to be accessed via these operations. In some cases, however, the structure of the data is not merely an implementation decision, but fundamental to the manner in which the data is intended to be used:

- the earliest databases were indexed sequential files. These were organized into records, each with a unique *index key*. While these files could be processed sequentially (one record at a time), it was more common to *get* and *put* records based on their keys.
- these evolved into Relational databases, accessed via Structured Query Languages.
- the complexity (and non-scalability) of SQL databases gave rise to much simpler (and more scalable) Key-Value Stores accessed only by *get, put,* and *delete* operations.

While all of these wind up being implemented (usually by some middle-ware layer) on top of byte streams, that is certainly not how they are accessed by their clients.

## 3. Other types of files

It can be argued that all of the above types of ordinary files are still just blobs of data, differing in their binary representations and the operations in terms of which they are written and read back. But there are other types of files that are not merely blobs of data, to be written and re-read.

### 3.1 Directories

Directories do not contain blobs of client data. Rather they represent name-spaces, the association of names with blobs of data. They are, in this respect, somewhat similar to key-value stores, but they the namespace is much more highly structured, and the operations are quite different. One very important difference is that a key-value store, as a single file, typically contains data owned by a single user. The namespace implemented by directories includes files owned by numerous users, each of whom wants to impose different sharing/privacy constraints on the access to each referenced file.

Because of how important directories are to the system operation and integrity, all directory operations tend to be implemented within the operating system. To ensure the correctness and security of the directory structure there are only a few supported update operations (e.g. *mkdir(2), rmdir(2), link(2), unlink(2), create(2)* and *open(2)*), each of which involves considerable permission checking and integrity assurance.

But despite these differences, directories exist in same namespace as files, and have the same notions of user/group ownership, and file protection. They can even be accessed (if you know what you are doing) with *open(2)* and *read(2)*.

### 3.2 Inter-Process Communications Ports

An inter-process communications port (e.g. a pipe) is not so much a container in which data is stored, as it is a channel through which data is passed. That data is exchanged via *write(2)* and *read(2)* system calls on file descriptors that can be manipulated with the *dup(2)* and *close(2)* operations. And in the case of named pipes they can be accessed via the *open(2)* system call.

With directories we saw something that was implemented as on-disk byte streams, but accessed with very different operations. With inter-process communications ports, we have something with a very different implementation, but that is accessed (as a byte stream) with normal file I/O operations.

### 3.3. I/O Devices

I/O devices connect a computer to the outside world. Many operating systems put devices in the file namespace. In Unix/Linux systems the special file associated with a device can be anywhere and have any name.

Many sequential access devices (e.g. keyboards and printers) are fit naturally into the byte-stream *read(2)/write(2)* model. Other random access devices (e.g. disks) easily fit into a *read(2)/write(2)/seek(2)* access model. Communications interfaces often behave like byte streams (or perhaps message streams) that also support additional control functions (like controlling line speed, setting MAC address, etc), which we can handle with *ioctl(2)* operations.

But not all I/O devices can be fit into a byte-stream access model. Consider a 3D rendering engine comprised of a few thousand GPUs. Rather than deal with this as a byte stream, we simply map gigabytes of display memory and control registers into our address space and maniuplate them directly. With more primitive devices, we may choose to deal directly with digital and analog signals that sample the state of the external world and control external actuators. Beyond the fact that we used the *open(2)* system call to get access to them, these devices may bear no relation to persistent byte streams.

## 4. File Attributes

Most of the above have treated files as containers (or access portals) for data. But even a file of ASCII text (e.g. this HTML) is more than than the bytes it contains. In addition to *data*, files also have *meta-data* ... data that describes data.

### 4.1 System Attributes

Unix/Linux files all have a standard set of attributes:

- type: regular, directory, pipe, device, symbolic link, ...
- ownership: identity of the owning user and owning group
- protection: permitted access (read, write execute), by the owner, the owning group, and others
- when the file was created, last updated, last accessed
- size (for regular files) ... the number of bytes in the file (which may be sparse).

Other operating systems may support more or different attributes. What is important about this list is:

- all files have these attributes
- the operating system depends on these attributes (e.g. to correctly implement access control)
- the operating system maintains these attributes

### 4.2 Extended Attributes

There may be other information (beyond the basic system attributes) that is vitally important to correct file processing. Examples might be:

- if the file has been encrypted or compressed, by what algorithm(s)?
- if a file has been signed, what is the associated certificate?
- if a file has been check-summed, what is the correct check-sum?
- if a program has been internationalized, where are its localizations?

All of the above examples are *meta-data*. They are not part of the file's contents ... but rather descriptive information that may be necessary to properly process the file's contents. There have been two basic approaches taken to supporting extended attributes:

- associate a limited number/size of *name*=*value* attributes with each file.
- pair each file with one or more shadow files (sometimes called *resource forks*) that contain additional resources and information.

The operating system may even need to make use of extended attributes. If we were required to extend Unix/Linux owner/group/other protection to support generalized Access Control Lists we might choose store the additional information in (protected) extended attributes.

# 5. Diversity of Semantics

The Posix operations for file and directory operations are standardized and relatively universal (although some file systems do not support some types of links). The Posix standards even include a *readdir(3)* operation for file-system and operating system independent scanning of directories. It is not difficult to write portable software that operates on ordinary files and directories.

The operations on and behavior of other types of files (e.g. inter-process communications ports and devices) are not at all standardized.

While the need for extended attributes is widely recognized, there are many different implementations, and not (yet) any widely accepted standard.