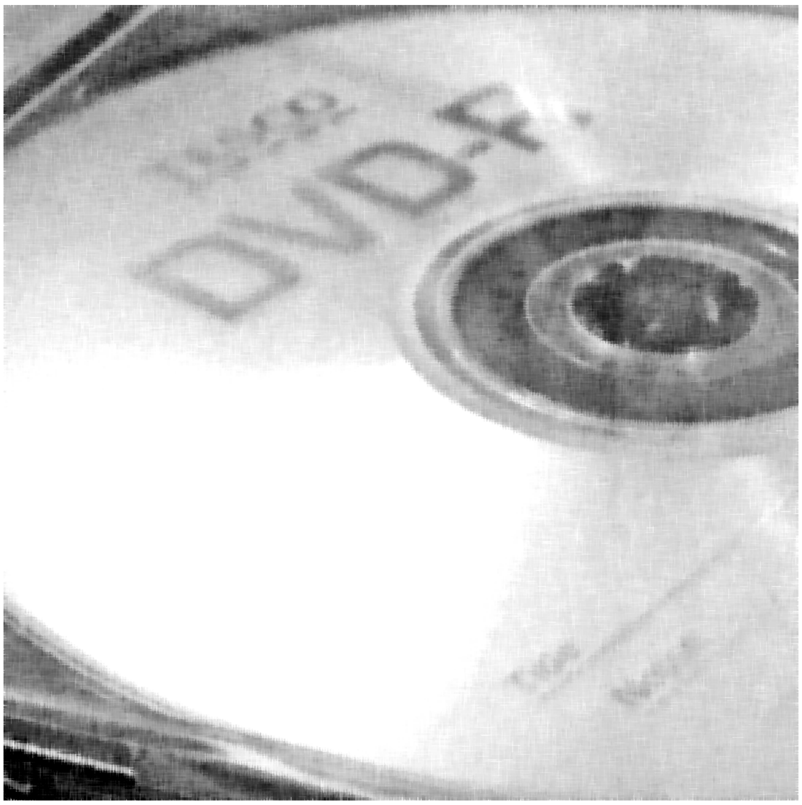
File System Management 12



DVD storage media.

12.1 Foreword

Though computer systems have experienced many significant changes and evolutionary steps in the past five or six decades, the nature of data has perhaps changed more than any other aspect. We can say this because of the huge changes in data volume being managed by computer systems, the explosion in the diversity of types of data, and the way it is stored and retrieved, both in local systems and via networks and remote resources. As a result, the concept of file management has evolved alongside operating systems technology in tandem. In this chapter we will explore the concept of file systems, the nature of data, and some important aspects of file and file-system management.

12.2 The concept of a file system

In computer systems, as we have discovered in earlier chapters, there is a concept of data storage, typically using disk systems. The hard-disk, and more recently the SSD, provide large scale storage of data in the form of many separate files. We also store files on removable devices such as USB memory sticks. These devices all look the same at the level of the file-system, even though the underlying hardware may differ.

In a file system, each file is a separate data entity. A file might be a document, a program, an image, a piece of audio, a spreadsheet, and so on: the possibilities are almost as limitless as the inventiveness of the programmer and user.

We might imagine that, on one level, every file is simply a sequence of binary values, or a stream of bytes. This is correct, certainly at the low-level structural state of data as stored on a disk unit. However, the arrangement of these bytes within the file dictates a particular file-format, and often there is more organisation to this than first thought might suggest.

However, let us stay with the idea of a file at a low level, as a sequence of bytes, just for the moment. Remember that a disk unit has a concept of storage organised into tracks, and sectors. Also remember that a sector is of a particular fixed size, such as 4096 bytes. Consequently, a single file must consist of one or more data sectors. If by pure good luck a file requires precisely 4096 bytes of storage, it might fully occupy a single sector. If by sheer bad luck a file requires exactly 4097 bytes, then it must occupy two sectors, even though the second one is almost entirely empty of valid data.

Every file will therefore waste (on average) 0.5 sectors of data storage capacity, which is 2048 bytes for a sector size of 4096 bytes. We can reduce this wastage by making sectors smaller, but this means that the number of sectors on the disk becomes larger, and this has negative implications for performance and other aspects of file management.

Now, if a single file typically consists of a number of sectors, each of which could be anywhere on the disk, then we need a way to keep track of which sectors belong to a given file, and this is achieved using a **file allocation table**. And because there are usually thousands of files, then we also need to keep track of every one of those files and its associated sectors. The smaller the sectors are, the larger the number of sectors that need to be kept track of per file, and in total for the entire disk. This requirement is one of several which drive the need for a well designed file-system.

We can now perhaps begin to define what a file system is, and add a few additional expectations we may wish it to fulfil:

- It keeps track of the individual identity of files on a disk.
- It knows which sectors relate to which files.
- It conforms to a standard to make disks interchangeable across systems.
- It achieves a degree of efficiency, balancing competing requirements.
- It has some fault tolerance of disk problems.
- It (may) provide a degree of security assurance.

So a file-system has to respect **efficiency**, and also the concept of **interchangeability**. The latter point is important: If a disk is configured with a given file system, then it can only be read by an operating system that also understands that file system. This creates the expectation of a **file-system standard**, which allows many computers to utilise the same storage devices. For example, one of the most widely known file systems is the FAT file system, but there are numerous others. One of the problems with standardisation is that there are many standards! However, at least these are well defined and able to be reproduced by software engineers reliably when required.

The aspect of **fault tolerance** is also important. Since disks can be damaged, and may develop faults over time, the file system must be aware of bad sectors and ensure these are never used for storage. This is relatively easy to achieve, and where there are huge numbers of sectors on a typical disk and rare faults, the impact on the user is usually unnoticeable. Other kinds of faults, such as mechanical failure, can be more catastrophic, but systems exist to try to mitigate these problems too.

We should also remember that the file system not only keeps track of where files are on the disk, but it also facilitates deletion of files and reuse of the space created by doing so. Keeping track of unused sectors is also important, then when a new file is created, the file system will know which sectors to use, and perhaps even which sectors represent the most efficient choices.

12.3 A file-system example

The FAT file system[150] is, as mentioned, a very widely known file system based on the concept of a **file allocation table**. There are a number of variants of FAT. However, they all have the same concept at their foundation. A file allocation table is simply a look-up table, typically stored on a reserved track of the disk unit, and containing information about every storage sector on the disk. In early systems, the number of sectors was limited to 65536 or even a puny 4096 sectors per disk, whereas modern FAT systems may typically support 4 billion sectors. One of the advantages of having a large number of sectors is that the sector size can be chosen to given optimal file storage and retrieval performance rather than being restricted by the need to keep the number of sectors low.

[150] This is sometimes abbreviated to FAT-FS. There are many implementations of FAT-FS, so this term is widely used.

12.4 The boot sector

As mentioned in earlier sections, the idea of a boot sector, as a portion of the disk containing startup code, is a well established aspect of file systems, including FAT. This is an integral part of the file-system: when a disk is formatted, the disk is configured to conform to a given file-system standard, with certain sector size, and other parameters. In order for the disk to be bootable it must have a valid boot sector. This is set up during a process called **disk formatting**, which prepares a disk for use with a particular file format.

The boot sector is responsible for ensuring that the computer system can start up and begin loading the operating system, which is stored on the disk using the chosen file format. Note that disks do not have to be bootable, in which case they can only be used as secondary disks, to provide extra storage space, but must exist alongside at least one bootable disk unit also installed on that system.

Note that a non-bootable disk can still contain applications, it just cannot initiate the startup process of the computer. Once the computer has started up by other means, it can happily run programs stored on the secondary non-bootable disk, should it need to do so.

12.5 Volumes and partitions

Sometimes it is convenient to have more than one drive in a system, though having several physical drives may be somewhat cumbersome. It is however possible to divide the total capacity of a drive unit into several subsections, known as **partitions**. Each of these can be formatted to appear as if it is a separate drive, and not necessarily using the same file system as other partitions on the same disk. It is possible for example to create two partitions, for example: one Linux OS, and one Windows OS, and have both of them capable of being bootable, giving the user a choice at startup[151] .

[151] This is known as a **dual-boot system**, though they can have more than two boot choices. There are a number of software tools available to manage these kind of systems conveniently. APPLE computer users might be familiar with BootCamp utility, for example.

Meanwhile, we can invert the argument we have just presented. Rather than subdividing a drive into smaller partitions, we may want to create the impression that a drive is bigger than a single physical drive, in other words, aggregate several drives into a single entity. This can be done by using the concept of a **spanned volume**.

It is worth noting that when a volume is smaller or the same size as a single drive then it is no different than a partition in practice. It is normal to talk about partitioning a drive and configuring it to have several volumes. However, a volume can span more than one physical drive.

So, for example, two 4 TeraByte drives could theoretically be formed into an 8 Terabyte volume that looks like a single drive to the operating system, its users, and their applications. These are sometimes also referred to as **spanned volumes**. One of the benefits of file system and operating system functionality is that the complexities of these issues are hidden: all drives look the same at the file access level.

Examples of a spanned volume and a partitioned drive are illustrated in Figure 12.1, where we can see (a) a single physical drive partitioned into

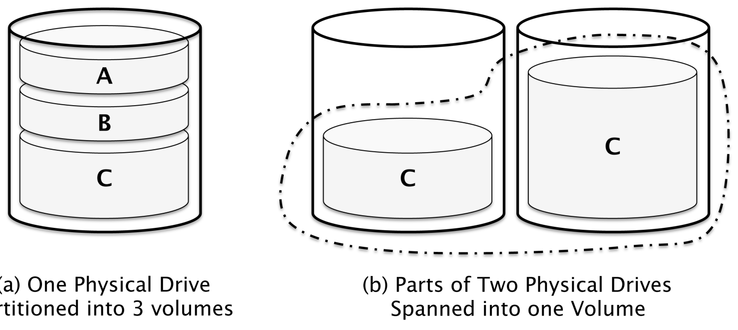

A
B
C
C
C
(a) One Physical Drive Partitioned into 3 volumes
(b) Parts of Two Physical Drives Spanned into one Volume

Figure 12.1: Examples of a partitioned drive, volumes and spanned volumes. Example (a) shows one drive partitioned into multiple volumes. Example (b) shows several drives spanned to create one larger volume occupying part or all of the two drives.

three non-spanning volumes, A, B, and C. We also see (b) a spanned volume where parts of two physical drives are combined to create a single larger volume.

12.6 The root directory

All file systems have a root directory. The root directory is the first place the operating system is able to look in order to find files stored on that disk. Simplistically, from the user point of view, this is where all of the files are stored. However, this would not be very convenient if this was all that happens.

Consider a system in which 10,000 customer records have to be stored as separate files, and at the same time the same computer needs to keep 300 spreadsheets with various accounts, stock records, employee records, etc. Add upon this the additional load of 1000's of operating system files containing kernel, code, utilities, and so on. Clearly, if these were all just dumped in the root directory in an entirely arbitrary fashion, things could get messy very quickly: files could get lost, accidentally deleted, mixed up, etc. To avoid this recipe for disaster, we have the concept of a hierarchical file system.

12.7 File-system hierarchy

To ensure some kind of organisational sensibility in the way files are managed, the vast majority of file systems employ the concept of file-system hierarchy. Let us consider again the example above, but impose some order out of the chaos.

Whilst the root directory[152] notionally contains all of the stored files, it can contain them hierarchically. This is achieved by the concept of sub-directories. If root is the first directory, and we then create further directories within the root, then these will be known as sub-directories. A directory is simply a named container used to hold a group (a subset) of files belonging to the file system. Therefore, in our rather unwieldy example above, we might decide to reorganise things into directories as follows:

[152] Directory is one term, another more recently popularised term is 'folder'.

OS : the directory containing all of the OS related files. CUST : directory containing all customer records. EMP : directory containing all employee records. STOCK : Directory containing stock spreadsheets ACCT: directory containing account spreadsheets

It should be clear that this is much more sensible already. We can easily find particular groups of files simply by remembering the directory name in question. Files cannot get mixed up easily.

In some operating systems, it is possible to restrict access to certain directories. For example, only the person who maintains the computer system might be allowed to have access to 'OS', so that other users do not accidentally mess up the operating system settings. We could take this a step further, and subdivide the directory structure to another level. We might end up with a structure as represented by the diagram of Figure 12.2. Here the root directory contains three top level directories, and some of those directories contain sub-directories. In this way, file-system content can be organised into a nested structure, with any number of sub-directories that may realistically be needed for most purposes.[153]

[153] There are of course limits to all systems. However, for the vast majority of uses, this is unlikely to be encountered, especially as disk systems continue to grow in capacity.

12.8 Special file cases

The file-system hierarchy allows for several special capabilities to be implemented, to create the illusion of a particular file configuration, when in fact that file is elsewhere. There are two particular cases worth mentioning at this point:

Symbolic Links: Some operating systems support the concept of linked files and directories. With symbolic links, it is possible to create the illusion of a file or directory existing in a particular place, but the reality hidden from the user is that this link actually points to another place in the system. This is usually done for convenience and consistency in file management arrangements in a given system.

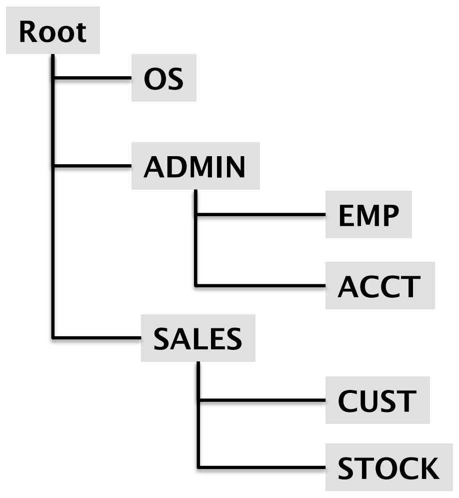


Figure 12.2: Example hierarchical file system directory structure. Where the Root directory contains three sub-directories, several of which have further nested sub-directories.

For example, taking Figure 12.2 as our model again, each of the accountants might find it easier to access the accounts if they had their own folders 'sam', 'joe', 'tina', and so on. Inside each of those folders a sub-directory exists called 'accounts'. However, 'accounts' is actually a symbolic link in this case, which simply links to: root/ADMIN/ACCT, or perhaps even links to a particular set of files that are the responsibility of that particular accountant. An example of this is shown in Figure 12.3

There are some advantages in doing this: for example, accounts can be stored anywhere, and moved around from time to time, and, as long as the symbolic links are kept up to date, the accountant never needs to know where the files are actually stored. They only need to look in their own local folder and the files magically always appear there.

Symbolic links are also necessary from time to time to manage various operating system tweaks that may result from various versions of software coexisting or running with different versions on a system.

Virtual Files: Where computer systems are operating in a networked environment, it is possible for files and directories to exist on one machine, but to be visible on another machine, via a network and suitable operating system enabled capability. In a similar way to a symbolic link, a file or directory on a remote system can be made to appear as if it exists in a local folder on the user's system.

Again, using our company example, this can be useful: the store room of the company may have four separate stock control terminals in different

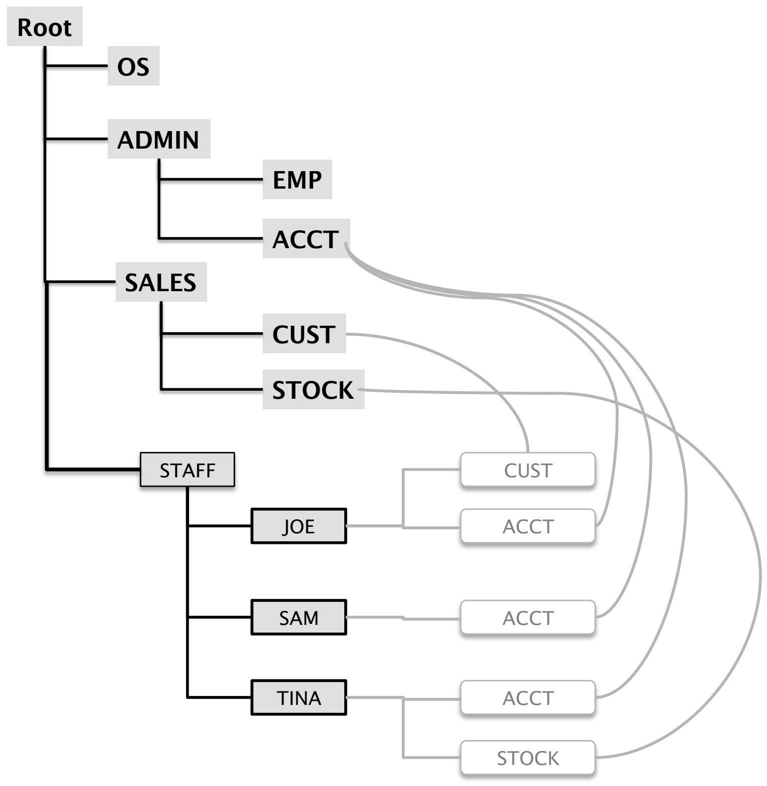


Figure 12.3: An example of a linked file configuration. Joe, Sam, and Tina each see their own file space, but this is actually part of a larger file system.

parts of the warehouse. Each of these may see the stock files as if they are in a local folder, but in fact that set of files may be elsewhere, accessed via the network and looked after by a system technician at that location who never has to come down to the warehouse, or the four terminals, to manage those files.

12.9 File attributes

Most file systems support the concept of files having attributes. These help to control how files can be used, allow users and administrators to track such things as when a file was created, when it was last modified, last read, and so on. This and other capabilities make file management much easier.

Exactly which attributes are supported is operating system dependent, and these are not universal. Some typical file attributes are described below:

Hidden: A hidden file is one that is not normally visible to the user. Hidden files can be viewed using particular commands, but not all users may have this capability. Often a hidden file is used to hide unnecessary information from general users, and prevent accidental changes to these files.

Executable: Some operating systems will only execute a program stored in a file if the file has been designated as an executable file. Attempting to execute a file that is not an executable could otherwise potentially cause a program crash. Also, a user that does not have the ability to change a file attribute to executable, is unable to write a program and then run it. This might provide additional security to that system.

System: A file that belongs to the operating system or kernel. Generally only a superuser can alter these files. This protects the operating system from being accidentally or deliberately modified.

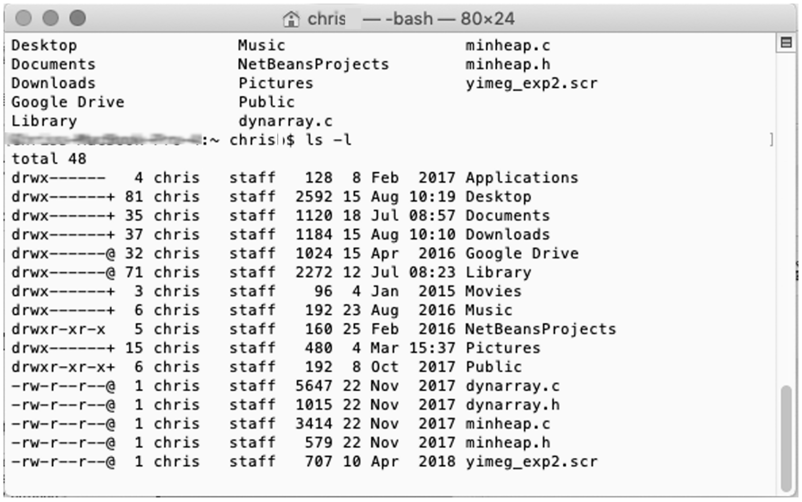
Read-Only: A file that is read-only (or write-protected) can be viewed but not modified or deleted. If a user has the right to change a file to remove the read-only status then they can alter it, but only after changing the attribute. This provides a level of protection against accidental removal or overwriting of files even if the user has the right to do so.

Archive: Used in some operating systems to control the backup policy for particular files.

We can view file attributes using command line commands, an example of which is shown in Figure 12.4, or via a window-based file manager.

Some operating systems control some of these features by designating files as readable and writable. For example a file that is readable, but not writable, is read-only. A file that is not readable is effectively a hidden file as far as the user is concerned. The effect is the same, only the terminology is different.

Some operating systems are more sophisticated in file attribute management. For example, in Linux OS, it is possible to define groups of users, and to set permissions for groups to access, read, write, and execute files, according to their group status. So, for example, a group called 'HR' might allow all HR members to view staff record files, whereas 'employees' cannot.



```
Desktop                 Music                   minheap.c
Documents               NetBeansProjects        minheap.h
Downloads               Pictures                yimeg_exp2.scr
Google Drive            Public
Library                 dynarray.c
[redacted]:~ chris$ ls -l
total 48
drwx------   4 chris   staff   128  8 Feb  2017 Applications
drwx------+ 81 chris   staff  2592 15 Aug 10:19 Desktop
drwx------+ 35 chris   staff  1120 18 Jul 08:57 Documents
drwx------+ 37 chris   staff  1184 15 Aug 10:10 Downloads
drwx------@ 32 chris   staff  1024 15 Apr  2016 Google Drive
drwx------@ 71 chris   staff  2272 12 Jul 08:23 Library
drwx------+  3 chris   staff    96  4 Jan  2015 Movies
drwx------+  6 chris   staff   192 23 Aug  2016 Music
drwxr-xr-x   5 chris   staff   160 25 Feb  2016 NetBeansProjects
drwx------+ 15 chris   staff   480  4 Mar 15:37 Pictures
drwxr-xr-x+  6 chris   staff   192  8 Oct  2017 Public
-rw-r--r--@  1 chris   staff  5647 22 Nov  2017 dynarray.c
-rw-r--r--@  1 chris   staff  1015 22 Nov  2017 dynarray.h
-rw-r--r--@  1 chris   staff  3414 22 Nov  2017 minheap.c
-rw-r--r--@  1 chris   staff   579 22 Nov  2017 minheap.h
-rw-r--r--@  1 chris   staff   707 10 Apr  2018 yimeg_exp2.scr
```

Figure 12.4: Example of file attributes. Example accessed via a Mac-OS command line, showing attribute flags on the left.

12.10 The variety of file systems

There is a wide variety of file systems in use currently, often frequently found being used alongside particular operating systems with which they have co-evolved.

- **FAT** File Allocation Table , FAT32 most commonly in use currently, very widely used and supported across many operating systems.
- **NTFS** New Technology File System, an evolution from FAT, which includes capabilities for much larger drives and files (FAT32 limits files to 4Gbyte), and also includes some fault-tolerance features to prevent data loss due to power outages etc.
- **APFS** APPLE File System, used on APPLE devices such as laptops and desktop machines, smartphones, and so on.
- **EXT** Extended File System (various versions Ext2, Ext3, Ext4 and so on), used with Linux primarily.

There are many others (tens, if not hundreds), too many to list in detail. Many of these were devised to work with fairly specialised operating systems, many of which are no longer (or perhaps never were) widely used. Since operating systems are often driven by proprietary or open standards, the most widely used are often associated with MICROSOFT Windows systems, Unix and Linux systems, and Mac-OS systems, and their relatives (such as IOS, Android, etc).

12.11 File compression

File compression is a technique that permits a portion of data to occupy less space when stored on a disk than it requires when loaded into computer memory for immediate use. This concept is widely used, both to reduce storage capacity of files, and also to reduce the amount of data transmitted by network connectivity, and in theory this can also reduce the delays required to transfer information around systems.[154]

[154] Does this translate into better performance? This very much depends upon how much CPU effort is required to perform compression and decompression.

There are a multitude of compression schemes, some very specific to particular data types, whilst others are more general purpose, or certainly used as such. Within the scope of compression, there are also other factors: most importantly the ability to reconstruct the data content in its exact original form (**lossless compression**), as compared to a format that can only regenerate an approximation of the original data (**lossy compression**).

However, to be precise, when we consider file-system compression as it is found integrated into operating system and file storage practices, we are almost certainly talking about lossless methods, specifically to reduce file size in transmission and storage, but without altering the reproducible file content in any way. Data compression in a broader sense is slightly different, because it can apply any method to the original data, before it is stored, and could be lossless or lossy in nature, depending upon what can be tolerated.

12.11.1 File compression example

One of the the most widely used file-system compression methods is known as **Huffman compression**, often used in tandem with **LZSS compression**[155]. The **ZIP** file format is a file archive format,[156] with optional compression, which supports many compression methods and options within its range of capabilities (including Huffman-LZSS), and allows (with care) a compression outcome that is particularly good for a given file or data type.

[155] Lempel–Ziv–Storer–Szymanski (LZSS) compression algorithm.

[156] File archiving, using TAR or ZIP is first of all a method of collecting together groups of files into single entities. Most archive tools offer compression as an option (though it is nearly always the reason they are used).

An easy way to understand Huffman compression is as follows. Suppose instead of bytes, we have letters of the alphabet A to Z. Now consider that with 26 letters, and each letter being assigned a code, it will require two digits to represent each letter, (A being 01, B being 02, up to Z being 26). So the word 'EDEN' could be represented by 8 digits:

E (05) **D** (04) **E** (05) **N** (14) = 05,04,05,14

So far, all we have done is assigned an equally sized value to each character, and this offers no benefit. But now suppose that we observe that certain letters occur much more frequently than others in any piece of text, such that E,D,O,S,C etc are very frequent, whilst N,B,X,Z,K etc are much less frequent. In a particular Huffman scheme, the most frequent letters might require only a single digit, less frequent letters require two digits, and the least frequent letters require three, four, perhaps more digits. Consequently we might find that:

E might have the code **1**,
D might have the code **3**,
N might have the code **109**.

Now we can represent our word as follows:

$$\textbf{E}\ (1)\ \textbf{D}\ (3)\ \textbf{E}\ (1)\ \textbf{N}\ (109) = \ 1,3,1,109$$

We should be able to see that the new encoding requires only 6 digits, a saving of 25%. We can express a compression saving in another way, described as a compression ratio. We can define this as follows:

Definition 12.11.1 ***Compression Ratio (CR)***

$$CompressionRatio = \frac{OriginalDataSize}{CompressedDataSize}$$

Using this formula we find that $\text{CR} = 8/6 = 1.33$. Any compression ratio greater than 1.0 represents a reduction in file size.[157]

[157] We have used decimal digits here for simplicity. If you prefer an example closer to a real hardware storage example, consider each letter initially as a 4-bit binary code, with values ranging from 0000_2 to 1001_2, then consider what happens if we replace these with optimal length bit codes such as 01, 110, 11, etc.

When this method is applied to data in a more general sense, it can be used to achieve significant data compression savings. Often the reason this is possible is that data contains many repetitions, and these can be encoded as frequent cases, using fewer digits to represent them.

It is important to note that there is a cost to using any compression method. It requires CPU time to compress, and then later to decompress such files. Also, we are actually reducing the total information content of the data by applying compression, even though its effective information content is still present. In other words, redundancy within the data representation is reduced by compression. But redundancy helps to improve fault tolerance. If a plain text file is slightly corrupted it may be possible to recover the meaning of the text:

Cam you sti£l wo3k out whaM lhis says?

However, if a compressed text file is corrupted even by a single bit, then it will very likely decompress to a potentially complex mis-decoded and usually unintelligible file content.

12.12 Automatically compressed file systems

Because there are some significant benefits in using file compression, some disk units and/or operating systems have been designed to apply this automatically to all files stored on that disk unit.[158] This is in contrast to the user-directed file compression which is applied on an ad hoc basis by a user only as and when it is felt to be desirable.

[158] This is also known as transparent compression. It may actually apply to an entire disk or to designated parts of the file-system hierarchy, depending upon the sophistication of the system employed.

There are considerations to be made here, some of which are competing against each other. Consider the following :

- Compressing a file requires CPU effort,
- CPU effort uses power and time,
- Compressed files are smaller so use less disk space
- Less data means less effort moving it about,
- Smaller files can be accessed faster and have less fragmentation.

So we see that actually, whilst enabling a compressed file-system to operate continuously will save disk space, and speed up file access, it will also take time to compress files when writing, and to decompress files when reading.

Remembering the estimates we made for disk performance, in Chapter 7, let us suppose for example that we have a disk system in which the average seek time to locate a file is 5 milliseconds, and the average file size is 100 sectors, taking 0.1 milliseconds per sector to read. Then the average file read time is 15 milliseconds: 5 milliseconds for seek time plus 10ms (100 x 0.1ms) for sector reads.

Now, suppose that by using file encryption the average file size can be reduced to 60% of its original size, (a compression ratio of 1.66[159]), and assume also that the decompression effort when reading back the file requires 2 milliseconds. Now we can calculate average file time as seek time 5ms, 60 sector reads of 0.1ms each (6ms) and decompression time 2 ms, a total average access time of 13 milliseconds.

[159] Compression Ratio is calculated as original size divided by new size. In this case, 100 divided by 60 = 1.66.

Interestingly, using compression, in this example, we have achieved an increase in file-system read performance of around 15%, whilst also reducing file storage requirements by 40%. In this example, as long as the compression effort takes 4ms or less (the **break-even point**) then we get the reduced storage requirement with no net penalty in speed terms, and potentially a speedup.

Clearly, there are issues here, in that an over-complicated compression algorithm might squeeze a few percent more compression from a file system, but with a large CPU time demand, the question of how much compression is enough is not easy to answer, but so-called **lightweight compression algorithms** appear to gain the best of high speed, low CPU demand, and reasonable compression ratios.

12.12.1 File encryption

Often, in the process of performing file compression, it is beneficial to also apply encryption. This is a technique which transforms the data content into a form, using a mathematical process, that can only be recovered by reversing that mathematical process when in possession of some key information[160].

[160] Coincidentally, this is often referred to as the encryption key.

The precise methods of encryption vary, and there are multiple algorithms, which we examine further in Chapter 14. Increasingly, those encryption techniques are becoming more and more complex, in order to keep up with the increasing power of computers to break existing weaker encryption methods.

For everyday users, a file encryption is unlikely to be compromised within a short period of time. If data is required to be kept confidential for long time periods, then this is an issue that should be considered more possible. Files may have to be re-encrypted at some point with stronger encryption methods to remain secure.

Some disk systems also support inbuilt encryption, and this is seen in removable hard disks, and particularly in USB flash drives. Because flash drives are easily lost or stolen, they represent a security risk in some scenarios, and therefore using an encrypted drive makes sense.

No encryption is unbreakable, but the amount of effort required to break encryption for the vast majority of data files is simply too much effort for the gain that might be obtained in an average scenario. Only in highly sensitive areas such as governments, security services, etc, are there likely to be the resources to break such file encryptions in a reasonable time frame. Even so, this time-frame might be hours or days of supercomputer time, and not minutes on a laptop.

12.13 File-system resilience

As mentioned earlier, the physical integrity of data can be protected, to a lesser or greater extent, by using suitable disk hardware configurations, such as RAID storage arrays. However, this is only part of the picture. Ensuring file-system resilience also requires additional aspects of operating system functionality and design.

At a lower level of kernel functionality, the file system needs to have features that ensure data integrity under unexpected conditions. For example, suppose a file is being moved from one place to another, and in the middle of that transfer, the power fails. What happens to the partially created file? What happens to the original file that we thought we were moving? With careful design, the file system and OS kernel can manage these scenarios in ways that allow errors to be recoverable in sensible ways that guarantee no data loss.

For example, a system where overwriting of a file is achieved by creating a duplicate, and only deleting the old version once writing has been completed and fully verified, will ensure that the worst that can happen is the latest version of the file may be lost, but not the previous version. A weakly resilient system might simply overwrite existing sectors of the file, meaning that if the system fails during the process, neither the old or the new file are intact.

This cautious philosophy has to be extended to the effects of virtual memory, file-caching in memory, cache modules within the hardware, and on the disk system, and so on. A fully resilient file system must ensure that any data written to a disk is flushed through to the physical disk as soon as possible, and any data not at that point is tracked so that errors can be flagged after something like a power outage, or component failure.

This is of course quite a difficult task to achieve in a watertight way. In most computer systems, the degree of resilience is a compromise between what is cost-efficient, good for performance, technically achievable, and also tolerable in terms of consequences. Most users know that if they pull out the power plug whilst typing in a document on a desktop PC, they will lose some data. But, they also expect systems to minimise the data loss by using features such as autosave to regularly update the hard-disk content to be as close as possible to the most recent work typed in.

Meanwhile, the ability of a user (and by extension any task started by that user) to modify files, is controlled by the concept of file attributes introduced earlier. So, if a particular file is read-only for a particular user, then any program that they run will also have that restriction. This ensures that rogue programs cannot start to do unexpected things by overwriting or deleting important files. This ensures that all tasks within the operating system follow the rules (permissions) relating to the right to read, write, or execute the content of any file present on the system.

A further extension of this model is the idea of **root control** as a restriction upon a process or set of processes, whereby the process being started is given a **virtual root directory** in which its own local files are visible, but outside of which it can see nothing of the wider filesystem. For example, if a system had a true root directory containing sub-folders 'OS', 'Users', and 'Private', then a process can be configured to see the root of the system as any desired path[161], such as **(root)/Users/user3**, meaning in this case that the process can only see files within user3 sub-directory, and not Private, OS, or other colleagues' directories that might exist in 'Users'. Essentially everything 'above' the point designated as virtual root is inaccessible. In this case, the process would be unable to even see the operating system files in OS, let alone do anything to the file content. This prevents deliberate or accidental process misbehaviour damaging unrelated file-system content.

[161] This is often achieved using the Linux **chroot** command or equivalent in non-Linux systems.

12.14 Internal file formats

File formats are far too numerous to investigate in detail individually in this text. However, there are some basic concepts of file formats that are worth investigating further.

ASCII TEXT FILE: The **ASCII**[162] text file contains a sequence of bytes, where each byte corresponds to a character, a letter, number, or other printable symbol, along with a few special characters (non-printable symbols). A text file can of course represent many things: the source code of a program, a list of items representing a record of a customer or employee, a simple text document, and so on. A text file can represent a series of words, parameters or commands that allow a data structure to be reconstructed. The format is entirely dictated by the program designer.

[162] ASCII: American Standard Code for Information Interchange, a system where typically one byte is used to represent one character or symbol in a data stream or file. ASCII has been the dominant standard for many decades (since 1960 in fact) but is now being challenged by the newer and varied UTF (Universal Transformation Format) encodings, which were first defined in 1992. The first 128 ASCII codes and their corresponding characters are listed below.

0-31 Special Control codes.

32	(space)	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_	127	

XML: Extensible Markup Language is a file format that permits a wide range of textually readable parameters and data values to be specified within a file, which is also well defined in its layout in such a way that it can be consistently read by computer programs. An example might appear as follows:

Simple XML example

```
<diary>
    <meeting>
        <date>03.04.19</date>
        <location>Office4</location>
    </meeting>
    <lunch>
        <date>05.04.19</date>
        <location>Luigi's</location>
    </lunch>
</diary>
```

We can see with a bit of effort that a collection of items exists within a group called **diary**, and that these include different types of events such as **meeting**, **lunch**, and anything else we wish to add. A program reading this file would then need to know how to deal with each type of event. Note that the indentations are entirely optional, and are there for human readability.

Binary file: Any file that is a continuous sequence of bytes with no human readable context. The vast majority of file formats contain some binary structure, since this is the most efficient way to store data in compact form: for example, the number 123 requires three ASCII character codes (3 bytes) to be stored as text, but in binary the same number requires only 1 byte at most. A graphical image might be described by a sequence of lines drawn by an artist using a file format that supports descriptions of drawings. However, an image might also simply be a large array of data bytes corresponding to pixels within the image. The latter case is typically a binary file format. Some examples of binary file formats include **BMP** (bitmap), **PNG**, **JPEG**, **WMV**, **MPEG**.

12.14.1 File structures and encodings

As mentioned, file formats vary according to the file type. Usually the file format is indicated by the **file extension** portion of the filename (the part after the main filename, after the dot). So **dog.bmp** is a BMP file type, and **shopping.txt** is a text file. Most file extensions are so well known that they are used universally only for that file type. However, it is worth cautioning that occasionally file types can be ambiguous because software developers have reused existing, less well-established, file extensions for other purposes.

A file extension is used to tell an application firstly if the file is one it can recognise and read, and secondly which type it is. Many applications accept multiple file types. A graphics editor for example, might desirably be expected to support all possible graphics file types, or at least the majority of widely used ones.

Usually a file has an initial portion of data called a **header**, containing essential information about the file, and this is then followed by the actual encoded file data, which we might call the payload.

A header block may tell the application which version of the file format is in use. A newer version of the file may have features that an older version does not support, or a feature that has been discontinued. It is generally necessary for software to support older **legacy** versions of file formats. It is not usually acceptable to just say 'sorry that is an old file type, it can't be opened'. This concept is also known as **backward-compatibility**.

Let us take an example: The **BMP** file format (known as the **device independent bitmap**), as defined by MICROSOFT. It consist of the following sections:

- **Header:** Basic information such as file size and other data. Also contains a space for application specific information.
- **Image Information:** A short block of data defining things like image dimensions, number of colours per pixel, and additional options.
- **Optional Colour Table:** a table defining colours used in the image if they are not the default ones.
- **PixelData:** A block of data containing 3 bytes for every pixel in the image.

All of these data values are encoded as binary data, so BMP is not human readable as if it were text. This is important, as pixel data can be compressed into a smaller number of binary bits than the textual equivalents.

The BMP file format has the option of algorithmic compression to further reduce the number of bytes that need to be stored. Many other file formats also use compression algorithms within their file encodings in order to achieve data compression.

Every file encoding is ideally defined by a well defined and maintained standard, such that any developer wishing to make their software compatible just has to implement their code according to that standard.

Where developers deviate from these standards they create problems for interoperability which may arise in the future. It is therefore important that standards are well defined, updated by a single standard originator, and properly documented. Here are some of the most well known file formats:

- **BMP**: Bitmap graphics file for 2D images.
- **CSV**: Comma Separated Values, used to store arrays of data.
- **TXT**, **TEX**: Text file, usually ASCII text characters.
- **GIF**: Graphics Interchange Format, 2D images.
- **PNG**: Portable Network Graphic.
- **WAV**: Audio Waveform file.
- **JPEG**: Joint Photographic Experts Group, lossy compressive graphics file format.
- **MPEG**: Moving Picture Experts Group, lossy compressive video file format.
- **DOC**: Document file (typically Micrsoft Word Documents).
- **XLS**: Spreadsheet binary file format (typically MICROSOFT Excel).
- **XLSX**: As for XLS but newer format that uses XML structure.
- **ODT**: Open Office Document.
- **EXE**: Executable file
- **OBJ**: Object File (intermediate file generated by a compiler).
- **ASM**: Assembly language file format.
- **C**, **CPP**, **java**, **py**, HLL program source file formats.
- **ZIP**, **TAR**: File archive, multiple files can be grouped together as a single 'tar' or 'zip' file, most often with the option of compression also applied.
- **PS**: Postscript, printed page content.
- **PDF**: Portable Document Format, as for PS.
- **SQL**: Structured Query Language (Database) file.
- **CGI**: Common Gateway Interface (script file for webservers).
- **HTML**: HyperText Markup Language (web page content).

We can find details of any of these formats via a web search, and can typically find a standard definition for version(s) of each file type, including detail of the internal structure of the file including header format, payload sections, etc. A competent programmer should therefore be able to write a program to read or write any of these file formats with suitable documentation and patience.

12.14.2 File format tradeoffs: an example

As we have just discussed, some file formats are very human friendly, but relatively inefficient, whereas others are distinctly unintelligible without program support, but compact and more efficient for the computer system. An example may now help to highlight the pros and cons of these two approaches. Suppose a car dealership wants a computer file to record their car stock. They need to know the *model*, the *year*, the *mileage*, and the *price* of each vehicle. Let us look at how this might work in several different file format choices.

OPTION 1: BINARY Format

Each piece of information may have a fixed length, so we might allow the following number of characters for each item:

MODEL: 8 characters
YEAR: 4 characters
MILEAGE: 6 characters
PRICE: 5 characters.

Now, there are three cars encoded in the fixed length binary format: (note . represents a blank space here)

FORD....2008..9223.5995CHRYSLER
2010100978.8000JAGUAR..2012.9923110995

You should be able to work out from the encoded content and the specification of the data fields (though not easily) that the car records are as follows:

MAKE	**YEAR**	**MILEAGE**	**PRICE**
FORD	2008	9223	5995
CHRYSLER	2010	100978	8000
JAGUAR	2012	99231	10995

This file format is reasonably efficient on storage space[163], and you will be able to see that it requires 23 characters per car - a total of 69 characters. There are other advantages, for example a computer program can skip over car records by skipping over multiples of 23 characters without having to read the content of each entry.

[163] For the sake of simplicity of the example we have actually ignored a more efficient option, where each character of the file can represent an 8-bit byte. In that case, year and price only requires two bytes each (two file characters), and mileage requires 3 bytes. So the car record could use as few as 15 bytes per car.

However, if the record format is ever changed, the entire file format will need to be read and re-written. Consider for example if a new item is added, called 'colour', or if price had an extra digit added. This would require restructuring of the entire file. It may be necessary for a program to support multiple variations of the file format, each with new features. In order to support this it is essential that the file itself includes some information about which version the file format is, thereby necessitating a header section.

OPTION 2: Structured Text File

Suppose that we encode our car data in the following textual format following an XML style

```
<car>
    <make>FORD</make>
    <year>2008</year>
    <mileage>9223</mileage>
    <price>5995</price>
</car>
<car>
    <make>CHRYSLER</make>
    <year>2010</year>
    <mileage>100978</mileage>
    <price>8000</price>
</car>
<car>
    <make>JAGUAR</make>
    <mileage>99231</mileage>
    <year>2012</year>
    <price>10995</price>
</car>
```

With this more structured file format, there are several things to note:

- The file is human readable, and could even be edited manually as a text file to correct problems.
- The size of the data fields are arbitrary. They do not have to be the full length, indeed they can be longer without having to modify the file format.
- Fields do not need to be in the same order each time. Each data item is preceded by a tag (the item in brackets), so the program reading the file will know what it is reading, even if the order changes.

- New tags can be added to the specification, and not all cars need to have that data field if not relevant. (It is extensible and record content can be customised to each car).
- Earlier file format versions are easier to support, and new features can be given default values, where older 'legacy' file formats are being read in and such data features are not present.
- Because each record may be a different length, the program reading the file cannot simply skip over a number of records by skipping a fixed distance. Therefore, the file must be iteratively parsed (read entry by entry) to find a particular record.[164]

[164] For very large sets of records, this becomes impractical, and other methods are used. In particular, **SQL databases**, which organise and access data records in more sophisticated and efficient ways. In effect they reduce unnecessary use of slow data storage systems to give speedier access to data.

This file format offers more flexibility, but at the cost of a larger file size. The tags waste space, though we could make them slightly shorter at the cost of less readability. If we count up the number of characters we find that this file format uses a total of 301 bytes or characters[165]. This equates to an average of about 100 characters per car record.

Note that the programmer still needs to create a bespoke program to interpret this file format. However, if standard XML was used, then the XML file structure (which might look not that dissimilar) accommodates the creation of data fields and parameters and hierarchical structure automatically. It would then become possible for any XML reader to read the file and display the content even though the program might not understand what it means. Programming languages often come with XML function libraries to permit rapid development of file content management and formats in this way: this takes away the hard work and provides ready debugged program code to do the job of reading and writing the data format.

[165] This figure includes an extra non-printable character at the end of each line of the text file, which is not used in the binary file-format case.

12.15 File management

Many system users only ever use their file systems via a window-based under interface (such as MICROSOFT Windows, Ubuntu, Android, APPLE OS, etc). The kind of actions they perform on files are usually moving files from one folder to another, copying them, deleting them (and undeleting!), renaming, perhaps zipping and unzipping, and so on.

The more advanced user can use the shell or command line capability of the operating system to type commands directly into the system, and act directly upon files at a low level. In some respects, the capabilities available are much more varied and sophisticated in this scenario, though the operator needs a deeper system knowledge to make good use of it.

If you wish to be highly proficient in use of shell file management, then you must study the particular operating system in question and learn how to use the command line effectively. Often there are no 'undo' options with the shell, so you must know what you are doing. However, with sufficient expertise, it also then becomes possible to combine commands into text files and have these executed one after another (shell scripting).

Let us consider an example. In the Mac/OS command shell, for instance, the following commands will perform a given task:

```
cd fred
ls
cp file1 file2
mv file2 ../
cd ..
```

These commands perform the following tasks (in order):

> **CD** changes the directory to place the command line context into the directory called fred.
> **LS** performs a listing of the files visible in that directory.
> **CP** copies file1 to a duplicate file called file2.
> **MV** moves file2 to a new location in the directory level above the current one (which is back where we started).
> **CD** moves the command line context back to the original directory (above fred).

So, in entirety, the sequence of commands results in **file1** in the **fred** subdirectory being duplicated, and that copy is then left in the directory where we started off. Clearly there is some learned expertise in doing this: knowing how to use each command allows them to be combined to achieve a particular task. Indeed, this is not the most concise way to perform the action described, we could simply type:

```
CP fred/file1 file2
```

A similar sequence might be achieved in MS-DOS, or Linux, though perhaps with slightly different command keywords and syntax. It is therefore also the case that using command line file management requires knowledge and experience of specific operating systems.

12.16 Summary

In this chapter we have explored the nature of files and file systems in a degree of detail that allows the beginner to appreciate most of the key considerations they are likely to come across. As always, some aspects have been simplified, and some deeper technical knowledge has been avoided.

However, what we should now appreciate is that file systems come in many varieties, and that the way that files are created and managed on disk systems is also quite diverse. File systems imply compatibility issues that need to be understood. The way storage devices are configured will dictate much about how the file system operates, and how efficient the file system is for a given use.

Just as the operating system kernel provides protections for tasks within the memory space, the file system also provides degrees of protection for users within the file space. Privileges, access rights, and file attributes can be used to constrain what users do within the system, and even to ring-fence their own regions of file space that are mutually exclusive of others. This is important for security and resilience, as well as for just making life easier for the users themselves.

12.17 Terminology introduced in this chapter

Archive file attribute	ASCII
Automatic file compression	Backward-compatibility
Disk formatting	Executable file
File allocation table	File header
File system	Hidden file
Huffman compression	Lightweight compression algorithm
Lightweight Process	Lossless compression
Lossy compression	LZSS compression
Partition (disk)	Read-only file
Resilient file system	Root control
Root directory	Spanned volume
Symbolic link	System file
Transparent compression	Virtual file
Virtual root directory	Volume (disk)
XML	

These terms are defined in the glossary, Appendix A.1.