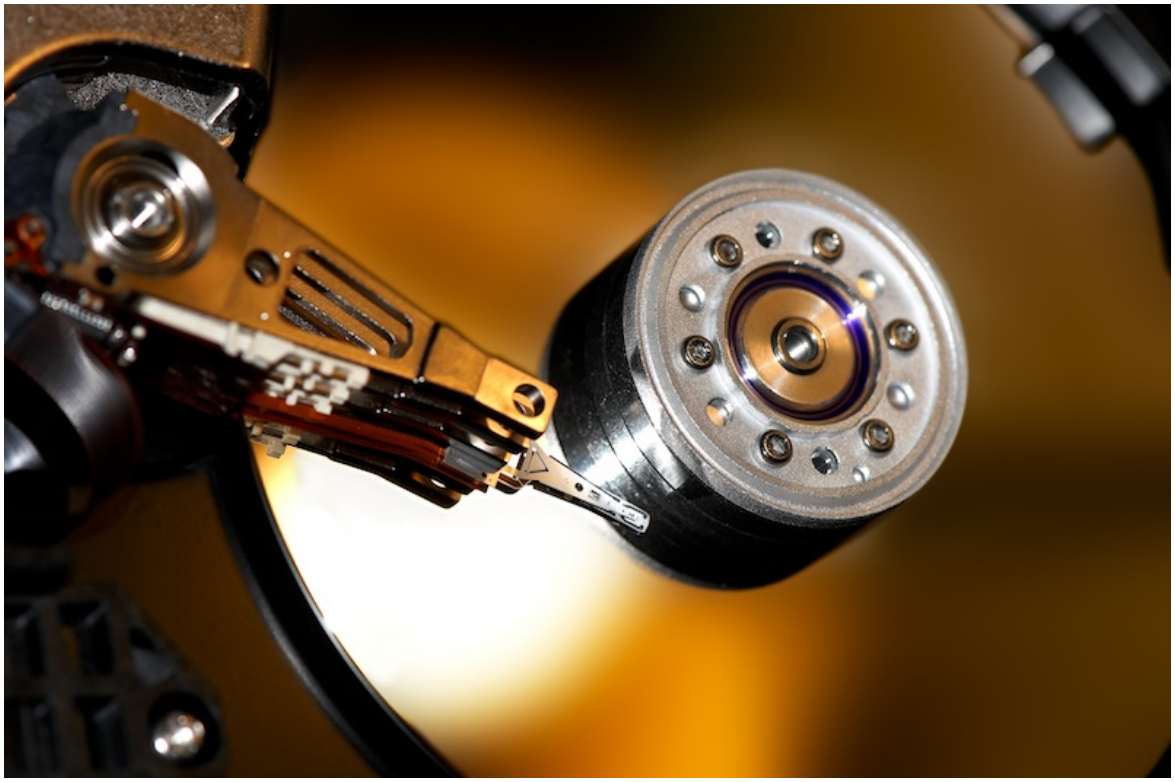
"Files and stuff"

Storage Management in Contemporary (and some fossilized) Operating Systems



Graphics: ilco (sxc.hu)

Abstract

This document was written in 2007. It was updated in 2011 with a cover page, table of contents and this introduction. The document is quite an in-depth treatise on how a file actually is stored on a disk.

[Patrik Sternudd]

2007-04-01

Table of Contents

1 Prerequisites – The Physical Media

If we are to store even a single file, we need somewhere to put it. Preferably a non-volatile area. For that purpose, a magnetic disk drive could be used. It is more suitable than for example a magnetic tape, because we can access every part of it in reasonable time (as opposed to tapes which can only be read sequentially, and furthermore, are more or less append-only). Therefore, near-line technology (e.g. jukeboxes) will not be discussed; nor will FLASH or CD-drives be covered.

Also slightly above the physical layer, we need to prepare the disk with partitions and also a basic (low-level) format.

1.1 What IS a Disk, Anyway?

There are several standards for disk drives (SCSI, IDE, SATA), each with their own variations. I will not go into these, but instead give a brief overview of the internal parts of a typical disk drive.

It will contain several platters mounted on a spindle. Each platter will have its own head which can write the data by modifying the magnetic surface. The head is not in contact with the disc itself; it is positioned slightly above. All heads are mounted on an arms (attached to an arm assembly), which can traverse the width of the platters, thus giving it access to the whole surface.

When the platters revolve, all positions that pass below the head during a complete rotation are called a track. A track is subdivided into sectors to be able to read a smaller portion. All tracks (i.e one track per disc) located on the same distance from the spindle makes a cylinder.

When measuring the speed of a disk drive, we must observe both the positioning time which is the time it takes the disk arm to reach the correct cylinder (seek time) plus the time until the platter has rotated so that the wanted sector is under the disk head (rotational latency).

Finally, it does not matter how fast the disk itself is if the interface between the disk and the host cannot handle it. This is called the transfer rate.

A disk is addressed as arrays of logical blocks, typically between 512 and 1024 bytes per block. The first sector (sector 0) is located on the first track, on outmost cylinder. Although the a track on the rim of a platter physically is longer than at the center, disk drives use something called CAV – constant angular velocity – to keep the data rate constant.

1.2 Partitioning & Formatting

1.2.1 Low-Level Formatting

Before a file system can be put on a disk, it must be a partition where it can be placed. But before even that, the disk must be low-level formatted. The low-level formatting creates the sectors, and it is also possible to specify the sector size. The sector consist of a header, a data area, and a trailer (much like a network packet).

1.2.2 Partitioning

Now, back to the partitioning. This is done by the OS, and each OS has its own format. For example, the Solaris VTOC (Volume Table Of Contents) supports seven slices (partitions). The

partitioning divides the disk into groups of cylinders, and each partition can then be handled as a separate volume (several partitions can also be grouped into a single volume).

1.2.3 High-Level Formatting

The third step is also formatting, but now on a higher level which is file system specific. In Windows, this is done with the format command, while UNIX system typically use mkfs. Groups of blocks are usually clustered to improve performance.

1.2.4 Swap and other raw file systems

However, it is not always desirable to have a full file system For example, when we swap data to disk, speed is our primary concern. Also, we have no need to organise the data in files or directories, since we can use other addressing schemes. Therefore, some partitions can be labeled as swap space.

The same holds for large databases with their own indexing and accessing methods. A common example is Oracle.

1.2.5 Booting

Now, when a system starts, it need to know where to find an operating system to load. Typically, a small bootstrap program is stored in a ROM, which calls another program – the boot loader – which resides on a pre-determined location called the boot block. A disk with a boot partition is called a boot disk. In Windows, the MBR (Master Boot Record) is used. As a comparison, Sun Microsystems has a very elegant system which makes use of NVRAM to give a flexible way of choosing boot disks (and indeed setting many other useful boot parameters before the kernel itself is loaded).

1.3 Failure Recovery

Since a disk is a mechanical device, sooner or later, something is going to give. The failure can range from a smaller one, such a single sector, to the more severe case of a head crash (when the head falls down to the disk itself, causing unmeasurable and permanent damage). It is of course desirable to be able to cope with both types of problems.

1.3.1 Handling bad sectors

It is not uncommon at all that disk sectors fail for one reason or another. For more advanced controllers and disks (usually SCSI), this can be handled on the fly by using ECC (Error Correcting Codes). If ECC are used, the controller computes a code on every read and write, and if an error is detected, it can be deduced what the correct value is (this takes some extra bits). The controller might also remap defective sectors with spare ones (if spares were allocated at formatting time). For simpler systems, this process must be done manually, which is time intensive, and might also require that the system is taken off-line. An example for the latter is the chkdisk command in MSDOS.

1.3.2 RAID

Let's face it. It's not a matter of if a disk drive will fail or not. It will. The question is, when? And what do we do when it does? If that is the only disk, and unless we have a backup, we might be in real trouble.

One solution is to use a RAID scheme. RAID stands for ”Redundant Array of Inexpensive Disks”, and was originally developed to be able to concatenate a lot of cheaper disk drives into a larger unit. It still has this function, but more important, it also provides for redundancy. There are several ”modes” of RAID, each identified by a number:

Level	Properties
0	Striping (concatenating) two disks. No redundancy, actually the MTTF is decreased since both disks become Single Points of Failure.
1	Mirroring. Very reliable, but costly. Every physical disk is mirrored to another.
2-4	Separate parity disks
5	Data and parity is spread on all disks.
1+0	Striped mirrors
0+1	Mirrored stripes (with higher theoretical availability)

RAID 0, 1, and combinations thereof, including RAID 5 are the most common ones.

Another consequence of using RAID is that we might actually increase performance by writing to multiple drives simultaneously.

The disadvantages with RAID are higher cost (more disks for the same amount of data), potentially lower performance (unless hardware RAID controllers are used), and more complex administration. However, the advantages far outweigh the drawbacks.

1.4 Finding the Stuff We Want: Scheduling

While the disk spins, we are able to both read and write data at will. But there will be several processes which will will require disk access, so the question about how to get as good performance as possible arise. We want each request to be serviced quickly to reduce the number of processes waiting for IO. Luckily, some good algorithms has already been developed. Some of these are described below. Both FCFS and SSTF has similarities with CPU scheduling, while the SCAN and LOOK are more disk specific.

Present the different algorithm, with their purposes, properties, advantages, drawbacks.

1.4.1 First Come, First Served (FCFS)

This one is as simple as it name suggest. All requests are served in the order they arrive. While easy to implement, the performance won't be very good (as with CPU scheduling).

1.4.2 Shortest Seek Time First (SSTF)

SSTF will serve the request to the closest cylinder from the current head position first (very much like the Shortest Job First CPU scheduling). It is better than FCFS, but can lead to starvation.

1.4.3 SCAN AND C-SCAN

The SCAN algorithm works by moving the arm from the rim of the disk to the center, servicing all requests as the head pass over the relevant cylinder. When it reaches the center, it ”turns” and makes it way out again.

However, it is more likely that a new request is waiting at - or close to - the opposite end, as more time has passed since the head was in that vicinity the last time. Therefore, a variation of SCAN was developed, called C-SCAN (Circular SCAN), where the head is moved to the rim as soon as it reaches the center, and then starts moving inward again.

1.4.4 LOOK AND C-LOOK

LOOK is an improvement of SCAN, where the head ”looks ahead” to see whether there are any request in the direction of the arm, and if it is not, it reverses directly. There is also a C-LOOK algorithm analogue to the C-SCAN one.

1.4.5 BUT...WHICH ONE SHOULD WE USE?

As usual, that depends. For example, if there is only one request at a time, all algorithms will work as FCFS. Also, different algorithms may have different efficiency depending on what file allocation method is in use. The LOOK and SCAN algorithms gives more overhead, but SSTF may lead to starvation.

2 File systems – what is it, really?

OK, so we want to store a file. Fine. Just put it in directory /home/myfiles/funny.txt and we are done, right? Well, not quite. Although the file system often is visualised as a file cabinet with drawers, the real implementation differs somewhat. Lets go through two of the most important components.

2.1 The File

From the user's perspective, the file might be the actual data. However, from the file system's view, a file is more of an abstract data type (ADT) which contains all necessary information about the file, including pointers to the actual data. The below attributes are common:

- An *identifier* used by the system, which has no need of pronounceable names.
- The *name* of the file as seen by the user.
- A *location* identifying where and on what device the file resides.
- *Size* information.
- Various *timestamps* (e.g. creation, time of last modification and access).
- *Ownership*, and other protection mechanisms.

Of course, a file is pretty much useless unless we can operate on it. First, we need a set of basic operations, such as:

- Creating the file (obviously).
- Reading and writing information.
- Repositioning within the file.
- Deleting the file when we are done with it.
- Truncating it. Truncating resets the file size to zero, but keeps all other attributes (the last modification timestamp should of course be updated if such an attribute exist).

Other operations such as copying, appending or replacing a file can then be formed by combining these basic ones.

2.2 Access Methods

There are two primary access methods. The first and most simple is *sequential access*, where the data could be visualised as a head reading form a tape, moving forward or backward one step at a time. To be able to append to the file, we would therefore have to go through the whole file before reaching the end of it, and then writing the additional data. However, when we are dealing with very large files (think 'databases'), this could be very time consuming. In this case, it would be desirable to access *any* block of information directly. This method is called *direct access* and offers a set of logical records to the user. The direct access method can also be further improved with indexes (e.g. with a hash table) or even indexes to indexes to the data, if the files are very large. This offers faster access, but is more complicated to implement.

2.3 The Directory

It is good to be able to store files, but it would be cumbersome indeed if we had to know each file name by heart to be able to access it later on. So we need to be able to list existing files. Unfortunately, by having all files in the same container would not improve the situation very much, since having thousands of different files would make it quite hard to find the desired one easily (I still recall the horror of issuing the DIR command in the NT3.51 install directory...). So, we want to be able to organise our files in a better way, e.g. by having a dedicated folder for a specific project. The most common way to achieve this is with a hierarchical structure, such as a tree. Thus, a specific file could be identified with a path name, which is either absolute (from the root) or relative (from any other point in the structure).

But, it gets more complicated than that: we might also want a file or directory pointer, known as a link in order to avoid having to keep multiple copies of files or directories. Keeping multiple copies in sync when sharing work between users is usually a formidable nightmare. When the link is requested, it is resolved to the real path name. The problem is, though, that if we don't implement links carefully, we might cause loops in our directory structure. Having a file access loop forever by resolving itself recursively is obviously not a good thing. One way to do this is to ignore links when traversing the structure. Another issue to consider when using links is how to handle removal of the target file. In Solaris, there are two types of links, hard links and soft links. The soft link is just a pointer to a file. If the target is removed, the link is left pointing at same place (the filename). For hard links, the link name points to the same inode as the target. The actual data will not disappear until the last hard link to it has been deleted. It should be noted that hard links cannot span files systems, whereas soft links may do so.

In summary, the below directory operations are required:

- Creating a file.
- Deleting a file.
- Renaming a file.
- Listing directory contents.
- Traversing the file system.
- Searching for a file.

Additionally, we must be able to create and remove directories. How this is done differs between systems. In MSDOS, a directory must be empty (not contain any files) before it can be removed. This holds for the UNIX rmdir command as well; however, the rm (for removing a file) can be used with the quite aggressive flags -rf (recursively, force) which could remove entire file systems, unless the user is wary.

2.4 Multiple users

Most current systems today are multiuser ones. That means we need to deal with protection and security – it is usually not desirable that everyone on a system have full access. For example, it would be desirable to protect to OS files from being overwritten or erased by accident, or to avoid sensitive information being disclosed between departments. For the file system, this means that we need attributes on files and directories which tells us who may access it. In the UNIX world, each entry have a permission list, with separate permissions for

the owner, group, and everyone else. UFS and other similar systems also provides ACLs (Access Control Lists) for more fine-grained permissions. The old FAT systems used by Microsoft had no restrictions at all (except read-only, which didn't help since anyone could remove it anyways). The advantage with the traditional UNIX protection is that it is fast. The disadvantage is that it isn't very flexible. On the other hand, ACLs are quite complex to administrate, and especially so on systems lacking usable GUIs. If there is both an ACL and a permission mask is set, the ACL has precedence.

Another issue we must deal with when we have multiple users are file locking. We do not want two users to open the same file for writing since this likely would cause a corrupt file. This can either be handled as in Windows where the locking is mandatory: if one user opens the file for writing, no other can do so. The UNIX approach advisory file locks. The process itself may set the lock, and then it's up to other processes to honour that.

2.5 Organisation

If we look on a file system as a black box, what would we see if we opened the lid and peered inside, and followed, say, a write() call from the user application down to the moment it is written to disk?

Most probably, we would encounter a layered design which could appear thus:

Layer	Name	Purpose
6	Application	Issues requests using syscalls
5	Logical file system	All information about files and structure, except the actual data. This is called the *metadata*. All information about a specific file is stored in a FCB (File Control Block) which could for example contain file ownership and pointers to the actual data.
4	File organisation module	Translates between logical and physical blocks, and free-space management.
3	Basic file system	Issues read and write requests on physical block level to the device drivers.
2	I/O Control	Device drivers to support connected hardware. A link between main memory and disks.
1	Devices	Hardware such as disk drives, etc.

2.6 Using the file system(s)

2.6.1 Mounting

Before we can access any files in the file system, it must be mounted into the OS. How this is handled differs between systems, but take for example a common UNIX system, where the

mount command takes the location of the file system and a mount point as argument (the file system type is optional). The location is typically a partition or volume; the mount point is a directory within an already mounted file system (apparently, at least one file system, called the *root system* is mounted automatically at boot time). Requirements for the mount point differs between systems: in some cases, the directory must be empty, in others, the original contents is hidden until the new file system is unmounted. The file system type argument must usually be supplied only when the system to be mounted not is of the default type of that particular OS. For a UNIX system such as Sun Solaris, UFS (and since Solaris 10, ZFS) is the common system together with NFS for remote file systems Linux typically uses Ext3 (the third extended file system) or ReiserFS. Windows has for most applications dropped the FAT, and FAT32 systems, and are now using NTFS.

But wait a minute. If there are so many different file systems, and most OS:es can have different types mounted concurrently, how is that handled without undue complexity?

Well...As usual, abstraction is a good thing to have! In UNIX (and Linux) systems, we have something called VFS (Virtual File System) which sits between the file system interface (with syscalls like read(), open(), etc) and the actual file system VFS also uniquely identifies a file (called a vnode) on a network, which is useful when for example NFS is deployed. It should perhaps be noted (again) that an inode is unique only within a particular file system.

A file system can be mounted automatically at boot time (in Solaris, such file systems are specified in /etc/vfstab), or manually with the mount command.

2.6.2 Operating on a file

When a file is created, a new FCB is either created or a free one is allocated, depending on the implementation. The actual directory is updated with the file name and the FCB.

When working (reading or writing) on a file, it must first be opened with the open syscall. If the call is successful, the FCB is copied into a system-wide open-file table (usually residing in memory). If multiple processes opens the same file, the system-wide table will be updated to reflect that (multiple write requests may be accepted or not, depending on the system). Then, an entry in the per-process open-file table is created and a pointer to the system-wide FCB is set. Finally, a pointer to the to the per-process entry is returned from the open() call. This pointer is called a *file descriptor* in the UNIX/Linux world, and a *file handle* in the Windows world. The per-process entry will also contain information about which flags the file was opened with (e.g. read, write, or both). This flag would then be used to ensure no illegal operation would be performed on the file. The advantage with doing this at open() time is that access won't have to be checked (and more importantly, fetched from disk) on every read or write operation.

2.6.3 File Types

As most people are aware, there are different file types. Some files are executable, whilst others are text files, and yet others seem to be good for nothing at all. How do the OS differentiate between them, how does it know what to do with them?

One way, such as Mac OS, stores the type in the file ADT. The advantage with such systems are ease of use, while the disadvantage, if this is mandatory, is that no suitable type for a new application may exist. Other OS:es, such as Windows, differentiate with the file extension (the part of the file name following the dot). For example, an executable file might end with .exe (there are a several other executable formats). UNIX and Linux use the extension as a clue for

the user what to do with it – however, an executable file must have the executable bit set for the particular user or group.

2.7 Space allocation

The challenge of space allocation is to ensure that the available disk space is utilised as effectively as possible. There are three primary methods, which are covered below.

2.7.1 Contiguous

When contiguous allocation is used, each file must use a set of contiguous blocks. This is very similar to the sequential file access method. As with memory management, the problem with external fragmentation arises. Manual defragmentation is very cumbersome. Another problem is that the system need to know how much space to allocate beforehand, which may be complicated at times (especially for log files, which just tends to grow and grow).

2.7.2 Linked

With this scheme, the disk blocks of a given file are connected by a linked list. This means the blocks may be scattered all over the disk, but the danger is what happens if a bad block would cause the chain of links to fail. Another thing to consider is that linked allocation only is efficient with sequential access (a direct access request for a block in the middle will not work, since we must start at the beginning and traverse the list until we find the desired block). On the other hand, if a file becomes larger, it is easy to expand it with another free block.

2.7.3 Indexed

The indexed scheme is like the linked one, but with an index with all pointers. This supports direct access, but without external fragmentation. Depending on file size, different types and levels of index can be used to achieve performance for small and large files alike (such as the UFS inode).

2.7.4 Free space management

Another point to consider is how to keep track of free space. As mentioned before, the free-space manager is part of the file organisation module. The manager maintain a free-space list containing all free disk blocks. When a file or directory (it is said that ”In UNIX, everything is a file” which is true enough – the directory is just a file with a special bit set) is created, a number of blocks are requested from the manager, which (if available) are removed from the free-space list. Similarly, when a file is deleted, the blocks are returned to the list.

So how does the list work? One way is to use a bit vector, where each block takes one bit. Free blocks are set to ones, allocated ones zero. This is both simple and efficient as long as the bit vector can be kept in memory. However, this can become troublesome for large disks.

As an alternative, a linked list may be used, where a pointer to the first block is kept and and all subsequent blocks are linked to the next one. To improve performance (i.e. to avoid extensive list traversals), the pointers of a number of free blocks can be stored in the first one, and the last pointer contain a pointer to another block with a group of blocks. Also, since it is common that several contiguous blocks are allocated (and freed) at the same time, the address of the first block together with a count of available blocks following that one could be kept.