How Hard Disks and Filesystems Work

Compress and Store

Anybody making full use of an off-the-shelf computer will want to gain access to the hard disk. Very few users are aware of how files are stored on a hard disk, or of the various options and with some fundamental knowledge, such access will be worry free. BY MARTIN SCHULZE

From a purely physical viewpoint hard disks are made up of rotating disks and moving read/write heads. The number of disks and read/write heads depends on the individual hard disk type.

When a computer boots, the hard disk is automatically started, and the drive motor ensures that the disks rotate at uniform speed. Older hard disks typically run at 5,400 or 7,200 revolutions per minute (rpm), newer models run at speeds of 10,000 or even 15,000 rpm.

Read and write operations require the read/write head to the correct position, in a process known as a seek operation. The noises that hard disks tend to make are caused by two things; firstly by the rotating disks and the drive motor that powers them. This is a permanent background noise. If your hard disk is too loud, you may be able to reduce its rotational speed, thus making your drive purr instead of whining.

The other source of noise is the movement of the read/write heads or more accurately their drive motors, and this noise only occurs during hard disk access.

Addressing

Two different schemes are used to address the data on a hard disk. In order to store data on the disks, the operating system needs a means of describing where to write the data, or where to read it from later. To do so, the OS tells the drive the required position; the drive then moves the head to the right position and either reads or writes data.

The smallest addressable unit is a sector, which has a capacity of 512 bytes. A track comprises of multiple sectors. Tracks form concentric circles on the surface of a disk and are organized around the hub of the disk (see Figure 1).

The number of tracks and sectors depend on the density of the of the area to which we are going to write to. The more tightly you can store bits, the smaller the gap between tracks and sectors, so there will be more of them. A hard disk's capacity is a function of these aspects. Track 0 is the outside track and numbers increase towards the center of the disk. Tracks with the same number are assigned to cylinders. The capacity of a disk is thus the result of multiplying the number of heads, by the number of tracks or cylinders, by the number of sectors per track and finally by the sector size.

When the Linux kernel is executed, it outputs details of the disks it has recognized on the console. Linux also indicates the disk geometry advertised by the disk, and displays the number of sectors and the total capacity derived from these Figures (see Figure 2).

CHS is one possible addressing scheme; the abbreviation is derived from the first letters in "Cylinder Head Sector". To access a sector on a drive, the operating system calls interrupt 0x13 (hexadecimal 13, decimal 19) to pass a

THE AUTHOR

Martin Schulze spends most of his time developing, improving and promoting free software, for example, by organizing LinuxDays, lectures, and workshops. You can contact Martin at joey@infodrom.org.

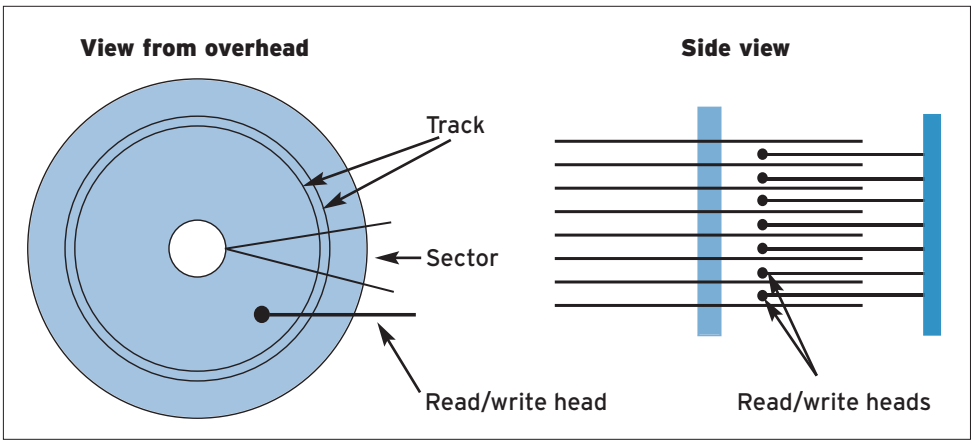

Figure 1: Hard disk internals

combination of the required head, cylinder and sector to the BIOS (see Table 1).

Size restrictions

Unfortunately, this scheme has a drawback. The number of bits available for representing the address in the computer's BIOS and in the interface between the BIOS and the hard disk is restricted. This in turn enforces restrictions on the maximum addressable disk size and geometry.

The specification as shown in Table 2 applies to the interface between the computer BIOS and the hard disk. It allows a maximum addressable size of 65536 x 16 x 256 x 512 Byte = 128 GB.

This specification dates back to a time when nobody could even imagine having a disk of this capacity or even using that amount of space. Of course, this predates on-line file sharing and digitized movies. Way back then nobody could imagine a computer needing 1 GB RAM and I can even remember someone saying "640 KB should be enough for anybody". Western Digital recently launched the WD2000 drive family with a capacity of 200 GB, which is no longer fully addressable using this scheme.

The problems do not stop there, as the conventional (and somewhat antiquated) PC BIOS is extremely miserly, providing only 24 bits instead of 28 and also re-arranging them (see Table 3). If you are good at maths, you should already have noticed that only 1024 x 256 x 63 x 512 bytes = 7.844 GB are addressable.

As the smallest data field is significant for all the others, this unfortunately leaves you with the values as shown in Table 4.

So that leaves you with a mere 1024 x 16 x 63 x 512 bytes = 504 MB of addressable space, and that is really not a lot by modern standards.

A BIOS upgrade or the Ontrack Disk Manager was required to use larger disks. After upgrading to a BIOS capable of translating the CHS values, the logical geometry of large hard disks no longer corresponds to the physical geometry. The logical geometry is supplied by the BIOS and automatically mapped to the physical geometry when the disk is accessed. At least this allowed users to access the total disk space on older modern drives.

The screenshot in Figure 2 shows that physical geometry of the second IDE hard disk as CHS = 119108/16/63, however, the BIOS shows different values, CHS = 7473/255/63, and this is what the kernel will report, when displaying the partition table a few lines lower down.

LBA Addressing

As you can see, this addressing scheme is stretched to its limits by today's hard disks. The solution is a completely different type of addressing which was introduced in 1995 and is known as LBA. LBA is short for "Logical Block Addressing" and numbers the sectors, or data blocks, on a hard disk sequentially, starting with 0.

Initially, only 28 bits were available for LBA; but this was a major improvement on CHS addressing and supported disks with up 128 GB. Today, 48 bits are available for addressing – assuming a block size of 512 bytes, this would permit disks with up to 131,072 terabytes. Post 2000 computers should actually have 64 bits available for LBA addressing – again assuming a block size of 512 bytes, this would allow disks with up to 8,589,934,592 terabytes to be addressed.

Kernel Support for Multiple Addressing Schemes

The kernel developers always attempt to allow Linux to support as many systems

Table 1: Calculating disk capacity

Cylinder	Head	Sector	Sector size	Capacity
5005	x 255	x 63	x 512 Bytes	= 40 GB
119108	x 16	x 63	x 512 Bytes	= 60 GB

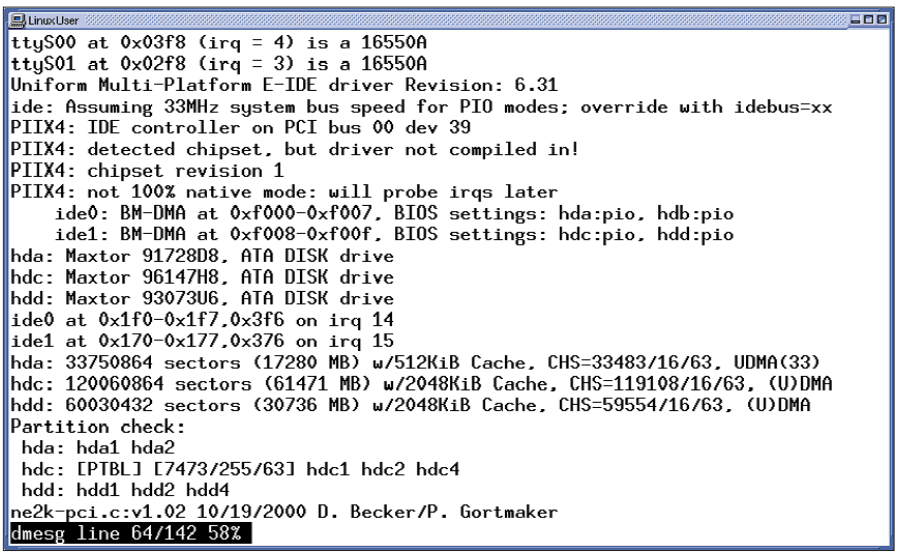

Figure 2: Linux discovers three hard disks on booting

Table 2: Bits between BIOS and IDE interface

Number	Available
16	Bits for cylinders (0..65535)
4	Bits for heads (0..15)
8	Bits for sectors (0..255)

Table 3: IDE address bits in the BIOS

Bits	Available
10	Bits for cylinders (0..1023)
8	Bits for heads (0..255)
6	Bits for sectors (0..62) (max. 63 sectors)

Table 4: Total bits for IDE addressing

Bits	Available
10	Bits for cylinders (0..1023)
4	Bits for heads (0..15)
6	Bits for sectors (0..62) (max. 63 sectors)

as possible. This is why the Linux kernel supports multiple addressing schemes. 48 bit LBA has the highest priority. If the hardware or the BIOS does not support this scheme, the IDE driver uses 28 bit LBA, defaulting to 28 bit CHS addressing only if this fails.

Incidentally, this problem only concerns IDE drives. SCSI drives, which are typically used for server systems and professional (Unix) workstations, are not affected. Server drives generally need more capacity, as they are typically required to store more data than a simple IDE drive sitting in someone's home PC. This is why SCSI drives traditionally use LBA as their block addressing scheme.

Reading from a Hard Disk

The whole hard disk is divided into data blocks of 512 bytes each. Thus each read or write operation will apply to a multiple of 512 bytes. In other words, hard disk access always occurs for blocks with a fixed length. Incidentally, the same principle also applies to CD ROMs.

Under the LBA addressing scheme when a hard disk is told to read a sector, the hard disk controller first needs to translate the sector number to the appropriated cylinder/head/sector. CHS addressing supplies these values directly as part of the request. The read/write head is then moved to the required position and reads the bits while the surface of the disk rotates past the head.

After buffering the data temporarily, the hard disk controller issues an interrupt to let the operating system know that it has successfully read some information, returning the storage position in this case, or that it has complied with a write request. The operating system can now go on to process the information.

To enhance access speeds, particularly if the same sectors are re-read, the kernel can implement a block cache. Linux uses excess memory for this purpose and reduces the cache size should normal programs require more memory.

Master Boot Record

The operating system will put some of the sectors on a hard disk to a very specific use. The first block on the hard disk is reserved for metadata. On a PC this block stores the computer's boot program, which is called by the BIOS when the computer is powered on.

Linux either stores *lilo* or *grub* in the MBR, the Master Boot Record. The boot loader can then call other programs, such as the Linux kernel, for example; the position of the Linux kernel on the hard disk is also stored in this hard disk sector.

The role of the Master Boot Record doesn't finish here though. The partition table is also stored at the end of this block and contains a description of the logical hard disk structure from the operating system's viewpoint. Operating systems tend not to use the hard disk as a whole, but to divide the disk into so-called partitions (Figures 3 and 4). The third component in the magic number 0xAA55, which informs the OS that it really is an MBR.

Storing Information

After dealing with the technical details, a further question arises: "how is information stored on a hard disk?". A simplistic approach would be to write any information that needs to be stored on the disk sequentially in contiguous blocks, as shown in Figure 4. The first file occupies three blocks and is followed by a file that occupies five. This used to be followed by a file that occupied five blocks, but was deleted later, and the space is followed by a file that occupies three blocks.

Although this approach seems quite simple, it does have a major disadvantage: the file names have not been stored. Although a program could access files one through four, it cannot map them to an intuitive name, and although a computer might be quite happy with a setup of this kind, humans tend to prefer descriptive file names with symbolic names such as */bin/bash* or */usr/bin/emacs*.

Having said that, a variant of this scheme is still in use today. If you ever had the pleasure of writing to a floppy disk on an older Unix system, you will be familiar with commands such as the following:

```
tar cf /dev/fd0 directory/
```

The files in the directory supplied as a parameter are compressed by *tar* and then written block by block to the disk. The command was simply reversed, to read the same data:

```
tar xf /dev/fd0
```

This approach may allow you to place an archive on a floppy and restrict access to the *tar* program, but floppies are slow and provide limited storage capacity,

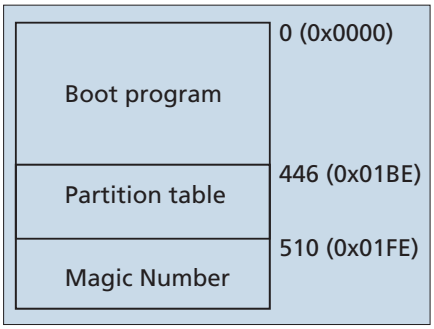

Figure 3: The Master Boot Record (MBR)

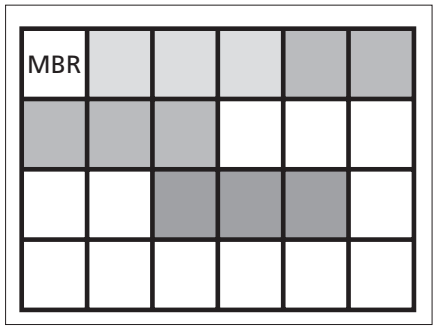

Figure 4: Contiguous data storage

```
Linux User
finlandia!joey(pts/32):~> ssh luonnotar df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda3             10080520    904156   8664296  10% /
/dev/hda2                15554      1903     12848  13% /boot
/dev/hda5             29214120   6636444  21093676  24% /var
finlandia!joey(pts/32):~> df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1                64221     42119     18786  70% /
/dev/sda3               896286    437307    412677  52% /usr
/dev/sdc4              8686312   7438404    806668  91% /var
/dev/sdd4              8620716   7663772    519032  94% /home
/dev/sde2              8751784   1479192   6828024  18% /src
kuolema:/home/ftp/debian
                      27358543  10119276  17239267  37% /pub/debian
finlandia!joey(pts/32):~>
```

Figure 5: Configuring the filesystem on two computers

so you can probably live with the restrictions. Random access to arbitrary files, allowing you to manipulate, delete and create files at any time is simply not possible using this approach.

Today the DOS filesystem and the M tools are typically used for floppies, although tape drives will still tend to write raw data, just like floppies used to. In this case, the filenames are kept, as *tar* places them in the code it stores, and they are available after reading the archive.

The Filesystem

Obviously, simplistic contiguous storage is not sufficient to permit names and attributes to be stored along with the file data, particularly if random access to arbitrary data or files is required.

Instead, a uniform structure is required to accommodate the files (and directories) in the storage space (normally a partition). This structure is typically referred to as a filesystem.

The *mount* command is used to attach filesystems to a running system. This is normally taken care of automatically on booting a computer. Some manual editing is required after adding a new drive or partition. The system configuration file for filesystems to be mounted at start-up is called */etc/ fstab*.

This is a major asset of Linux and similar systems. Instead of using drive letters that change when a partition is added or removed, filesystems are simply added to the tree below / (root).

Even if the hard disk label changes at a later date, or if you add an additional hard disk, this will not affect the logical position in the filesystem tree, as the IDs reflect the IDE or SCSI channel. There is no need to reconfigured applications or paths, instead you simply edit the configuration file, */etc/fstab*.

Smaller systems commonly provide a single root filesytem, /. Larger systems might additionally place */var*, */usr*, and */home* on partitions of their own, to provide more space and independence.

Inodes

The Linux kernel automatically places files on the correct file system, or reads them from it, by reference to the path name and the mounted file system. To do so, a kernel mechanism parses the path name and discovers the filesystem that stores the file. On Unix oriented filesystems every file (and every directory, being a special type of file) is mapped to an inode. Inodes map directory entries to data blocks on a partition. Inodes themselves are special data blocks that store metadata for the files they point to. You can envisage them as a kind of data structure that stores additional information for a file, such as its length, access permissions, ownership, access times, and pointers to the blocks in use. Inodes are normally used by the internal filesystem only.

Users can also find out which inode belongs to which file. The *ls -i* command (see Figure 6) shows both the filename and the inode it uses. Thus, the number of files and directories on a file system is not only restricted by the size of the files and the total capacity of the partition, but also by the number of inodes – this is a point to watch out for, when setting up a system.

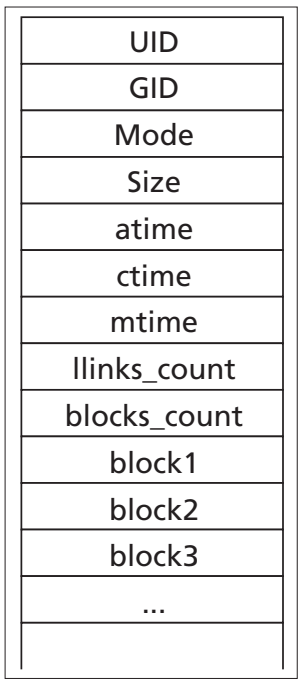

Figure 6: General structure of an inode

The general structure of an inode is shown in Figure 6. The actual structure will depend on the basic filesystem. The structure used by the Second Extended Filesystem, the Linux standard, is described in the *ext2_fs.h* kernel file.

The Linux filesystem provides access to a variety of information that users will be confronted with in various forms. Whether on the command line or in the kernel – the filesystem is literally ubiquitous.

SELLING OUT FAST!

More information at:
www.linux-magazine.com/Backissues

LINUX MAGAZINE