Large Disk HOWTO

Table of Contents

Table of Contents

Large Disk HOWTO

Andries Brouwer, `aeb@cwi.nl`

v2.2u, 12 March 2001

All about disk geometry and the 1024 cylinder limit for disks.

For the most recent version of this text, see www.win.tue.nl.

9. Consequences

10. Details

11. The Linux IDE 8 GiB limit

12. The Linux 65535 cylinder limit

13. Extended and logical partitions

14. Problem solving

---

1. The problem

Suppose you have a disk with more than 1024 cylinders. Suppose moreover that you have an operating system that uses the old INT13 BIOS interface to disk I/O. Then you have a problem, because this interface uses a 10−bit field for the cylinder on which the I/O is done, so that cylinders 1024 and past are inaccessible.

Fortunately, Linux does not use the BIOS, so there is no problem.

Well, except for two things:

(1) When you boot your system, Linux isn't running yet and cannot save you from BIOS problems. This has some consequences for LILO and similar boot loaders.

(2) It is necessary for all operating systems that use one disk to agree on where the partitions are. In other words, if you use both Linux and, say, DOS on one disk, then both must interpret the partition table in the same way. This has some consequences for the Linux kernel and for `fdisk`.

Below a rather detailed description of all relevant details. Note that I used kernel version 2.0.8 source as a reference. Other versions may differ a bit.

2. Summary

You got a new large disk. What to do? Well, on the software side: use `fdisk` (or, better, `cfdisk`) to create partitions, and then `mke2fs` to create a filesystem, and then `mount` to attach the new filesystem to the big file hierarchy.

You need not read this HOWTO since there are *no* problems with large hard disks these days. The great majority of apparent problems is caused by people who think there might be a problem and install a disk manager, or go into `fdisk` expert mode, or specify explicit disk geometries to LILO or on the kernel command line.

However, typical problem areas are: (i) ancient hardware, (ii) several operating systems on the same disk, and sometimes (iii) booting.

Advice:

For large SCSI disks: Linux has supported them from very early on. No action required.

For large IDE disks (over 8.4 GB): get a recent stable kernel (2.0.34 or later). Usually, all will be fine now, especially if you were wise enough not to ask the BIOS for disk translations like LBA and the like.

For very large IDE disks (over 33.8 GB): see IDE problems with 34+ GB disks below.

If LILO hangs at boot time, also specify `linear` in the configuration file `/etc/lilo.conf`. (And if you did have `linear`, try without it.) If you have a recent LILO (version 21.4 or later), the keyword `lba32` will usually allow booting from anywhere on the disk, that is, the 1024 cylinder limit is gone.

There may be geometry problems that can be solved by giving an explicit geometry to kernel/LILO/fdisk.

If you have an old `fdisk` and it warns about overlapping partitions: ignore the warnings, or check using `cfdisk` that really all is well.

For HPT366, see the Linux HPT366 HOWTO.

If at boot time the kernel cannot read the partition table, consider the possibility that UDMA66 was selected while the controller or the cable or the disk drive did not support UDMA66. In such a case every attempt to read will fail, and reading the partition table is the first thing the kernel does. Make sure no UDMA66 is used.

If you think something is wrong with the size of your disk, make sure that you are not confusing binary and decimal units , and realize that the free space that `df` reports on an empty disk is a few percent smaller than the partition size, because there is administrative overhead.

If for a removable drive the kernel reports two different sizes, then one is found from the drive, and the other from the disk/floppy. This second value will be zero when the drive has no media.

Now, if you still think there are problems, or just are curious, read on.

3. Units and Sizes

A kilobyte (kB) is 1000 bytes. A megabyte (MB) is 1000 kB. A gigabyte (GB) is 1000 MB. A terabyte (TB) is 1000 GB. This is the SI norm. However, there are people that use 1 MB=1024000 bytes and talk about 1.44 MB floppies, and people who think that 1 MB=1048576 bytes. Here I follow the recent standard and write Ki, Mi, Gi, Ti for the binary units, so that these floppies are 1440 KiB (1.47 MB, 1.41 MiB), 1 MiB is 1048576 bytes (1.05 MB), 1 GiB is 1073741824 bytes (1.07 GB) and 1 TiB is 1099511627776 bytes (1.1 TB).

Quite correctly, the disk drive manufacturers follow the SI norm and use the decimal units. However, Linux kernel boot messages (for not−so−recent kernels) and some fdisk−type programs use the symbols MB and GB for binary, or mixed binary−decimal units. So, before you think your disk is smaller than was promised when you bought it, compute first the actual size in decimal units (or just in bytes).

Concerning terminology and abbreviation for binary units, Knuth has an alternative proposal, namely to use KKB, MMB, GGB, TTB, PPB, EEB, ZZB, YYB and to call these *large kilobyte*, *large megabyte*, ... *large yottabyte*. He writes: `Notice that doubling the letter connotes both binary−ness and large−ness.' This is a good proposal − `large gigabyte' sounds better than `gibibyte'. For our purposes however the only important thing is to stress that a megabyte has precisely 1000000 bytes, and that some other term and abbreviation is required if you mean something else.

3.1 Sectorsize

In the present text a sector has 512 bytes. This is almost always true, but for example certain MO disks use a sectorsize of 2048 bytes, and all capacities given below must be multiplied by four. (When using `fdisk` on such disks, make sure you have version 2.9i or later, and give the `−b 2048' option.)

3.2 Disksize

A disk with C cylinders, H heads and S sectors per track has `C*H*S` sectors in all, and can store `C*H*S*512` bytes. For example, if the disk label says C/H/S=4092/16/63 then the disk has `4092*16*63`=4124736 sectors, and can hold `4124736*512`=2111864832 bytes (2.11 GB). There is an industry convention to give C/H/S=16383/16/63 for disks larger than 8.4 GB, and the disk size can no longer be read off from the C/H/S values reported by the disk.

---

4. Disk Access

In order to read or write something from or to the disk, we have to specify a position on the disk, for example by giving a sector or block number. If the disk is a SCSI disk, then this sector number goes directly into the SCSI command and is understood by the disk. If the disk is an IDE disk using LBA, then precisely the same holds. But if the disk is old, RLL or MFM or IDE from before the LBA times, then the disk hardware expects a triple (cylinder,head,sector) to designate the desired spot on the disk.

The correspondence between the linear numbering and this 3D notation is as follows: for a disk with C cylinders, H heads and S sectors/track position (c,h,s) in 3D or CHS notation is the same as position `c*H*S` + `h*S` + (s−1) in linear or LBA notation. (The minus one is because traditionally sectors are counted from 1, not 0, in this 3D notation.)

Consequently, in order to access a very old non−SCSI disk, we need to know its *geometry*, that is, the values of C, H and S. (And if you don't know, there is a lot of good information on www.thetechpage.com.)

4.1 BIOS Disk Access and the 1024 cylinder limit

Linux does not use the BIOS, but some other systems do. The BIOS, which predates LBA times, offers with INT13 disk I/O routines that have (c,h,s) as input. (More precisely: `AH` selects the function to perform, `CH` is the low 8 bits of the cylinder number, `CL` has in bits 7−6 the high two bits of the cylinder number and in bits 5−0 the sector number, `DH` is the head number, and `DL` is the drive number (80h or 81h). This explains part of the layout of the partition table.)

Thus, we have CHS encoded in three bytes, with 10 bits for the cylinder number, 8 bits for the head number, and 6 bits for the track sector number (numbered 1−63). It follows that cylinder numbers can range from 0 to 1023 and that no more than 1024 cylinders are BIOS addressable.

DOS and Windows software did not change when IDE disks with LBA support were introduced, so DOS and Windows continued needing a disk geometry, even when this was no longer needed for the actual disk I/O, but only for talking to the BIOS. This again means that Linux needs the geometry in those places where communication with the BIOS or with other operating systems is required, even on a modern disk.

This state of affairs lasted for four years or so, and then disks appeared on the market that could not be addressed with the INT13 functions (because the 10+8+6=24 bits for (c,h,s) can address not more than 8.5 GB) and a new BIOS interface was designed: the so−called Extended INT13 functions, where DS:SI points at a 16−byte Disk Address Packet that contains an 8−byte starting absolute block number.

Very slowly the Microsoft world is moving towards using these Extended INT13 functions. Probably a few years from now no modern system on modern hardware will need the concept of `disk geometry' anymore.

4.2 History of BIOS and IDE limits

ATA Specification (for IDE disks) – the 137 GB limit

> At most 65536 cylinders (numbered 0−65535), 16 heads (numbered 0−15), 255 sectors/track (numbered 1−255), for a maximum total capacity of 267386880 sectors (of 512 bytes each), that is, 136902082560 bytes (137 GB). This is not yet a problem (in 1999), but will be a few years from now.

BIOS Int 13 – the 8.5 GB limit

> At most 1024 cylinders (numbered 0−1023), 256 heads (numbered 0−255), 63 sectors/track (numbered 1−63) for a maximum total capacity of 8455716864 bytes (8.5 GB). This is a serious limitation today. It means that DOS cannot use present day large disks.

The 528 MB limit

> If the same values for c,h,s are used for the BIOS Int 13 call and for the IDE disk I/O, then both limitations combine, and one can use at most 1024 cylinders, 16 heads, 63 sectors/track, for a maximum total capacity of 528482304 bytes (528MB), the infamous 504 MiB limit for DOS with an old BIOS. This started being a problem around 1993, and people resorted to all kinds of trickery, both in hardware (LBA), in firmware (translating BIOS), and in software (disk managers). The

concept of `translation' was invented (1994): a BIOS could use one geometry while talking to the drive, and another, fake, geometry while talking to DOS, and translate between the two.

The 2.1 GB limit (April 1996)

Some older BIOSes only allocate 12 bits for the field in CMOS RAM that gives the number of cylinders. Consequently, this number can be at most 4095, and only 4095*16*63*512=2113413120 bytes are accessible. The effect of having a larger disk would be a hang at boot time. This made disks with geometry 4092/16/63 rather popular. And still today many large disk drives come with a jumper to make them appear 4092/16/63. See also over2gb.htm. Other BIOSes would not hang but just detect a much smaller disk, like 429 MB instead of 2.5 GB.

The 3.2 GB limit

There was a bug in the Phoenix 4.03 and 4.04 BIOS firmware that would cause the system to lock up in the CMOS setup for drives with a capacity over 3277 MB. See over3gb.htm.

The 4.2 GB limit (Feb 1997)

Simple BIOS translation (ECHS=Extended CHS, sometimes called `Large disk support' or just `Large') works by repeatedly doubling the number of heads and halving the number of cylinders shown to DOS, until the number of cylinders is at most 1024. Now DOS and Windows 95 cannot handle 256 heads, and in the common case that the disk reports 16 heads, this means that this simple mechanism only works up to 8192*16*63*512=4227858432 bytes (with a fake geometry with 1024 cylinders, 128 heads, 63 sectors/track). Note that ECHS does not change the number of sectors per track, so if that is not 63, the limit will be lower. See over4gb.htm.

The 7.9 GB limit

Slightly smarter BIOSes avoid the previous problem by first adjusting the number of heads to 15 (`revised ECHS'), so that a fake geometry with 240 heads can be obtained, good for 1024*240*63*512=7927234560 bytes.

The 8.4 GB limit

Finally, if the BIOS does all it can to make this translation a success, and uses 255 heads and 63 sectors/track (`assisted LBA' or just `LBA') it may reach 1024*255*63*512=8422686720 bytes, slightly less than the earlier 8.5 GB limit because the geometries with 256 heads must be avoided. (This translation will use for the number of heads the first value H in the sequence 16, 32, 64, 128, 255 for which the total disk capacity fits in 1024*H*63*512, and then computes the number of cylinders C as total capacity divided by (H*63*512).)

The 33.8 GB limit (August 1999)

The next hurdle comes with a size over 33.8 GB. The problem is that with the default 16 heads and 63 sectors/track this corresponds to a number of cylinders of more than 65535, which does not fit into a short. Most BIOSes in existence today can't handle such disks. (See, e.g., Asus upgrades for new flash images that work.) Linux kernels older than 2.2.14 / 2.3.21 need a patch. See IDE problems with 34+ GB disks below.

For another discussion of this topic, see Breaking the Barriers, and, with more details, IDE Hard Drive Capacity Barriers.

Hard drives over 8.4 GB are supposed to report their geometry as 16383/16/63. This in effect means that the `geometry' is obsolete, and the total disk size can no longer be computed from the geometry.

5. Booting

When the system is booted, the BIOS reads sector 0 (known as the MBR – the Master Boot Record) from the first disk (or from floppy or CDROM), and jumps to the code found there – usually some bootstrap loader. These small bootstrap programs found there typically have no own disk drivers and use BIOS services. This means that a Linux kernel can only be booted when it is entirely located within the first 1024 cylinders, unless you both have a modern BIOS (a BIOS that supports the Extended INT13 functions), and a modern bootloader (a bootloader that uses these functions when available).

This problem (if it is a problem) is very easily solved: make sure that the kernel (and perhaps other files used during bootup, such as LILO map files) are located on a partition that is entirely contained in the first 1024 cylinders of a disk that the BIOS can access – probably this means the first or second disk.

Thus: create a small partition, say 10 MB large, so that there is room for a handful of kernels, making sure that it is entirely contained within the first 1024 cylinders of the first or second disk. Mount it on `/boot` so that LILO will put its stuff there.

Most systems from 1998 or later will have a modern BIOS.

5.1 LILO and the `lba32' and `linear' options

Executive summary: If you use LILO as boot loader, make sure you have LILO version 21.4 or later. (It can be found at ftp://metalab.unc.edu/pub/Linux/system/boot/lilo/.) Always use the `lba32` option.

An invocation of `/sbin/lilo` (the boot map installer) stores a list of addresses in the boot map, so that LILO (the boot loader) knows from where to read the kernel image. By default these addresses are stored in (c,h,s) form, and ordinary INT13 calls are used at boot time.

When the configuration file specifies `lba32` or `linear`, linear addresses are stored. With `lba32` also linear addresses are used at boot time, when the BIOS supports extended INT13. With `linear`, or with an old BIOS, these linear addresses are converted back to (c,h,s) form, and ordinary INT13 calls are used.

Thus, with `lba32` there are no geometry problems and there is no 1024 cylinder limit. Without it there is a 1024 cylinder limit. What about the geometry?

The boot loader and the BIOS must agree as to the disk geometry. `/sbin/lilo` asks the kernel for the geometry, but there is no guarantee that the Linux kernel geometry coincides with what the BIOS will use. Thus, often the geometry supplied by the kernel is worthless. In such cases it helps to give LILO the `` `linear' `` option. The advantage is that the Linux kernel idea of the geometry no longer plays a role. The disadvantage is that `lilo` cannot warn you when part of the kernel was stored above the 1024 cylinder limit, and you may end up with a system that does not boot.

5.2 A LILO bug

With LILO versions below v21 there is another disadvantage: the address conversion done at boot time has a bug: when c*H is 65536 or more, overflow occurs in the computation. For H larger than 64 this causes a stricter limit on c than the well−known c < 1024; for example, with H=255 and an old LILO one must have c < 258. (c=cylinder where kernel image lives, H=number of heads of disk)

5.3 1024 cylinders is not 1024 cylinders

Tim Williams writes: `I had my Linux partition within the first 1024 cylinders and still it wouldnt boot. First when I moved it below 1 GB did things work.' How can that be? Well, this was a SCSI disk with AHA2940UW controller which uses either H=64, S=32 (that is, cylinders of 1 MiB = 1.05 MB), or H=255, S=63 (that is, cylinders of 8.2 MB), depending on setup options in firmware and BIOS. No doubt the BIOS assumed the former, so that the 1024 cylinder limit was found at 1 GiB, while Linux used the latter and LILO thought that this limit was at 8.4 GB.

5.4 No 1024 cylinder limit on old machines with IDE

The `nuni` boot loader does not use BIOS services but accesses IDE drives directly. So, one can put it on a floppy or in the MBR and boot from anywhere on any IDE drive (not only from the first two). Find it at //metalab.unc.edu/pub/Linux/system/boot/loaders/.

---

6. Disk geometry, partitions and `overlap'

If you have several operating systems on your disks, then each uses one or more disk partitions. A disagreement on where these partitions are may have catastrophic consequences.

The MBR contains a *partition table* describing where the (primary) partitions are. There are 4 table entries, for 4 primary partitions, and each looks like

```
struct partition {
        char active;    /* 0x80: bootable, 0: not bootable */
        char begin[3];  /* CHS for first sector */
        char type;
        char end[3];    /* CHS for last sector */
        int start;      /* 32 bit sector number (counting from 0) */
        int length;     /* 32 bit number of sectors */
};
```

(where CHS stands for Cylinder/Head/Sector).

This information is redundant: the location of a partition is given both by the 24−bit `begin` and `end` fields, and by the 32−bit `start` and `length` fields.

Linux only uses the `start` and `length` fields, and can therefore handle partitions of not more than 2^32 sectors, that is, partitions of at most 2 TiB. That is thirty times larger than the disks available today, so maybe it will be enough for the next six years or so. (So, partitions can be very large, but there is a serious restriction in that a file in an ext2 filesystem on hardware with 32−bit integers cannot be larger than 2 GiB.)

DOS uses the `begin` and `end` fields, and uses the BIOS INT13 call to access the disk, and can therefore only handle disks of not more than 8.4 GB, even with a translating BIOS. (Partitions cannot be larger than 2.1 GB because of restrictions of the FAT16 file system.) The same holds for Windows 3.11 and WfWG and Windows NT 3.*.

Windows 95 has support for the Extended INT13 interface, and uses special partition types (c, e, f instead of b, 6, 5) to indicate that a partition should be accessed in this way. When these partition types are used, the `begin` and `end` fields contain dummy information (1023/255/63). Windows 95 OSR2 introduces the FAT32 file system (partition type b or c), that allows partitions of size at most 2 TiB.

What is this nonsense you get from `fdisk` about `overlapping' partitions, when in fact nothing is wrong? Well – there is something `wrong': if you look at the `begin` and `end` fields of such partitions, as DOS does, they overlap. (And that cannot be corrected, because these fields cannot store cylinder numbers above 1024 – there will always be `overlap' as soon as you have more than 1024 cylinders.) However, if you look at the `start` and `length` fields, as Linux does, and as Windows 95 does in the case of partitions with partition type c, e or f, then all is well. So, ignore these warnings when `cfdisk` is satisfied and you have a Linux–only disk. Be careful when the disk is shared with DOS. Use the commands `cfdisk -Ps /dev/hdx` and `cfdisk -Pt /dev/hdx` to look at the partition table of `/dev/hdx`.

7. Translation and Disk Managers

Disk geometry (with heads, cylinders and tracks) is something from the age of MFM and RLL. In those days it corresponded to a physical reality. Nowadays, with IDE or SCSI, nobody is interested in what the `real' geometry of a disk is. Indeed, the number of sectors per track is variable – there are more sectors per track close to the outer rim of the disk – so there is no `real' number of sectors per track. Quite the contrary: the IDE command INITIALIZE DRIVE PARAMETERS (91h) serves to tell the disk how many heads and sectors per track it is supposed to have today. It is quite normal to see a large modern disk that has 2 heads report 15 or 16 heads to the BIOS, while the BIOS may again report 255 heads to user software.

For the user it is best to regard a disk as just a linear array of sectors numbered 0, 1, ..., and leave it to the firmware to find out where a given sector lives on the disk. This linear numbering is called LBA.

So now the conceptual picture is the following. DOS, or some boot loader, talks to the BIOS, using (c,h,s) notation. The BIOS converts (c,h,s) to LBA notation using the fake geometry that the user is using. If the disk accepts LBA then this value is used for disk I/O. Otherwise, it is converted back to (c',h',s') using the geometry that the disk uses today, and that is used for disk I/O.

Note that there is a bit of confusion in the use of the expression `LBA': As a term describing disk capabilities it means `Linear Block Addressing' (as opposed to CHS Addressing). As a term in the BIOS Setup, it describes a translation scheme sometimes called `assisted LBA' – see above under ` The 8.4 GB limit '.

Something similar works when the firmware doesn't speak LBA but the BIOS knows about translation. (In the setup this is often indicated as `Large'.) Now the BIOS will present a geometry (C,H,S) to the operating system, and use (C',H',S') while talking to the disk controller. Usually S = S', C = C'/N and H = H'*N, where N is the smallest power of two that will ensure C' <= 1024 (so that least capacity is wasted by the rounding down in C' = C/N). Again, this allows access of up to 8.4 GB (7.8 GiB).

(The third setup option usually is `Normal', where no translation is involved.)

If a BIOS does not know about `Large' or `LBA', then there are software solutions around. Disk Managers like OnTrack or EZ–Drive replace the BIOS disk handling routines by their own. Often this is accomplished by having the disk manager code live in the MBR and subsequent sectors (OnTrack calls this code DDO: Dynamic Drive Overlay), so that it is booted before any other operating system. That is why one may have problems when booting from a floppy when a Disk Manager has been installed.

The effect is more or less the same as with a translating BIOS – but especially when running several different operating systems on the same disk, disk managers can cause a lot of trouble.

Linux does support OnTrack Disk Manager since version 1.3.14, and EZ–Drive since version 1.3.29. Some more details are given below.

---

8. Kernel disk translation for IDE disks

If the Linux kernel detects the presence of some disk manager on an IDE disk, it will try to remap the disk in the same way this disk manager would have done, so that Linux sees the same disk partitioning as for example DOS with OnTrack or EZ–Drive. However, NO remapping is done when a geometry was specified on the command line – so a `` `hd=``*cyls*`,`*heads*`,`*secs*' command line option might well kill compatibility with a disk manager.

If you are hit by this, and know someone who can compile a new kernel for you, find the file `linux/drivers/block/ide.c` and remove in the routine `ide_xlate_1024()` the test `if (drive->forced_geom) { ...; return 0; }`.

The remapping is done by trying 4, 8, 16, 32, 64, 128, 255 heads (keeping H*C constant) until either C <= 1024 or H = 255.

The details are as follows – subsection headers are the strings appearing in the corresponding boot messages. Here and everywhere else in this text partition types are given in hexadecimal.

8.1 EZD

EZ–Drive is detected by the fact that the first primary partition has type 55. The geometry is remapped as described above, and the partition table from sector 0 is discarded – instead the partition table is read from sector 1. Disk block numbers are not changed, but writes to sector 0 are redirected to sector 1. This behaviour can be changed by recompiling the kernel with `#define FAKE_FDISK_FOR_EZDRIVE 0` in `ide.c`.

8.2 DM6:DDO

OnTrack DiskManager (on the first disk) is detected by the fact that the first primary partition has type 54. The geometry is remapped as described above and the entire disk is shifted by 63 sectors (so that the old sector 63 becomes sector 0). Afterwards a new MBR (with partition table) is read from the new sector 0. Of course this shift is to make room for the DDO – that is why there is no shift on other disks.

8.3 DM6:AUX

OnTrack DiskManager (on other disks) is detected by the fact that the first primary partition has type 51 or 53. The geometry is remapped as described above.

8.4 DM6:MBR

An older version of OnTrack DiskManager is detected not by partition type, but by signature. (Test whether the offset found in bytes 2 and 3 of the MBR is not more than 430, and the short found at this offset equals 0x55AA, and is followed by an odd byte.) Again the geometry is remapped as above.

8.5 PTBL

Finally, there is a test that tries to deduce a translation from the `start` and `end` values of the primary partitions: If some partition has start and end sector number 1 and 63, respectively, and end heads 31, 63, 127 or 254, then, since it is customary to end partitions on a cylinder boundary, and since moreover the IDE interface uses at most 16 heads, it is conjectured that a BIOS translation is active, and the geometry is remapped to use 32, 64, 128 or 255 heads, respectively. However, no remapping is done when the current idea of the geometry already has 63 sectors per track and at least as many heads (since this probably means that a remapping was done already).

8.6 Getting rid of a disk manager

When Linux detects OnTrack Disk Manager, it will shift all disk accesses by 63 sectors. Similarly, when Linux detects EZ–Drive, it shifts all accesses of sector 0 to sector 1. This means that it may be difficult to get rid of these disk managers. Most disk managers have an uninstall option, but if you need to remove some disk manager an approach that often works is to give an explicit disk geometry on the command line. Now Linux skips the `ide_xlate_1024()` routine, and one can wipe out the partition table with disk manager (and probably lose access to all disk data) with the command

```
dd if=/dev/zero of=/dev/hdx bs=512 count=1
```

The details depend a little on kernel version. Recent kernels (since 2.3.21) recognize boot parameters like "hda=remap" and "hdb=noremap", so that it is possible to get or avoid the EZD shift regardless of the contents of the partition table. The "hdX=noremap" boot parameter also avoids the OnTrack Disk Manager shift.

---

9. Consequences

What does all of this mean? For Linux users only one thing: that they must make sure that LILO and `fdisk` use the right geometry where `right' is defined for `fdisk` as the geometry used by the other operating systems on the same disk, and for LILO as the geometry that will enable successful interaction with the BIOS at boot time. (Usually these two coincide.)

How does `fdisk` know about the geometry? It asks the kernel, using the `HDIO_GETGEO` ioctl. But the user can override the geometry interactively or on the command line.

How does LILO know about the geometry? It asks the kernel, using the `HDIO_GETGEO` ioctl. But the user can override the geometry using the `` `disk=' `` option in `/etc/lilo.conf` (see lilo.conf(5)). One may also give the `linear` option to LILO, and it will store LBA addresses instead of CHS addresses in its map file, and find out of the geometry to use at boot time (by using INT 13 Function 8 to ask for the drive geometry).

How does the kernel know what to answer? Well, first of all, the user may have specified an explicit geometry with a `` `hda=*cyls*,*heads*,*secs*' `` kernel command line option (see bootparam(7)), perhaps by hand, or by asking the boot loader to supply such an option to the kernel. For example, one can tell LILO to supply such an option by adding an `` `append = "hda=*cyls*,*heads*,*secs*"' `` line in `/etc/lilo.conf` (see lilo.conf(5)). And otherwise the kernel will guess, possibly using values obtained from the BIOS or the hardware.

It is possible (since Linux 2.1.79) to change the kernel's ideas about the geometry by using the `/proc` filesystem. For example

```
# sfdisk -g /dev/hdc
/dev/hdc: 4441 cylinders, 255 heads, 63 sectors/track
# cd /proc/ide/ide1/hdc
# echo bios_cyl:17418 bios_head:128 bios_sect:32 > settings
# sfdisk -g /dev/hdc
/dev/hdc: 17418 cylinders, 128 heads, 32 sectors/track
#
```

This is especially useful if you need so many boot parameters that you overflow LILO's (very limited) command line length.

How does the BIOS know about the geometry? The user may have specified it in the CMOS setup. Or the geometry is read from the disk, and possibly translated as specified in the setup. In the case of SCSI disks, where no geometry exists, the geometry that the BIOS has to invent can often be specified by jumpers or setup options. (For example, Adaptec controllers have the possibility to choose between the usual H=64, S=32 and the `extended translation' H=255, S=63.) Sometimes the BIOS reads the partition table to see with what geometry the disk was last partitioned – it will assume that a valid partition table is present when the 55aa signature is present. This is good, in that it allows moving disks to a different machine. But having the BIOS behaviour depend on the disk contents also causes strange problems. (For example, it has been reported that a 2.5 GB disk was seen as having 528 MB because the BIOS read the partition table and concluded that it should use untranslated CHS. Another effect is found in the report that unpartitioned disks were slower than partitioned ones, because the BIOS tested 32–bit mode by reading the MBR and seeing whether it correctly got the 55aa signature.)

How does the disk know about the geometry? Well, the manufacturer invents a geometry that multiplies out to approximately the right capacity. Many disks have jumpers that change the reported geometry, in order to avoid BIOS bugs. For example, all IBM disks allow the user to choose between 15 and 16 heads, and many manufacturers add jumpers to make the disk seem smaller than 2.1 GB or 33.8 GB. See also below. Sometimes there are utilities that change the disk firmware.

9.1 Computing LILO parameters

Sometimes it is useful to force a certain geometry by adding `` `hda=*cyls*,*heads*,*secs*' `` on the kernel command line. Almost always one wants *secs*=63, and the purpose of adding this is to specify *heads*. (Reasonable values today are *heads*=16 and *heads*=255.) What should one specify for *cyls*? Precisely that number that will give the right total capacity of C*H*S sectors. For example, for a drive with 71346240 sectors (36529274880 bytes) one would compute C as 71346240/(255*63)=4441 (for example using the program `bc`), and give boot parameter `hdc=4441,255,63`. How does one know the right total capacity? For example,

```
# hdparm -g /dev/hdc | grep sectors
 geometry     = 4441/255/63, sectors = 71346240, start = 0
# hdparm -i /dev/hdc | grep LBAsects
```

```
 CurCHS=16383/16/63, CurSects=16514064, LBA=yes, LBAsects=71346240
```

gives two ways of finding the total number of sectors 71346240. The kernel output

```
# dmesg | grep hdc
...
hdc: Maxtor 93652U8, 34837MB w/2048kB Cache, CHS=70780/16/63
 hdc: [PTBL] [4441/255/63] hdc1 hdc2 hdc3! hdc4 < hdc5 > ...
```

tells us about (at least) 34837*2048=71346176 and about (at least) 70780*16*63=71346240 sectors. In this case the second value happens to be precisely correct, but in general both may be rounded down. This is a good way to approximate the disk size when `hdparm` is unavailable. Never give a too large value for *cyls*! In the case of SCSI disks the precise number of sectors is given in the kernel boot messages:

```
SCSI device sda: hdwr sector= 512 bytes. Sectors= 17755792 [8669 MB] [8.7 GB]
```

(and MB, GB are rounded, not rounded down, and `binary').

---

10. Details

10.1 IDE details − the seven geometries

The IDE driver has five sources of information about the geometry. The first (G_user) is the one specified by the user on the command line. The second (G_bios) is the BIOS Fixed Disk Parameter Table (for first and second disk only) that is read on system startup, before the switch to 32−bit mode. The third (G_phys) and fourth (G_log) are returned by the IDE controller as a response to the IDENTIFY command − they are the `physical' and `current logical' geometries.

On the other hand, the driver needs two values for the geometry: on the one hand G_fdisk, returned by a `HDIO_GETGEO` ioctl, and on the other hand G_used, which is actually used for doing I/O. Both G_fdisk and G_used are initialized to G_user if given, to G_bios when this information is present according to CMOS, and to G_phys otherwise. If G_log looks reasonable then G_used is set to that. Otherwise, if G_used is unreasonable and G_phys looks reasonable then G_used is set to G_phys. Here `reasonable' means that the number of heads is in the range 1−16.

To say this in other words: the command line overrides the BIOS, and will determine what `fdisk` sees, but if it specifies a translated geometry (with more than 16 heads), then for kernel I/O it will be overridden by output of the IDENTIFY command.

Note that G_bios is rather unreliable: for systems booting from SCSI the first and second disk may well be SCSI disks, and the geometry that the BIOS reported for sda is used by the kernel for hda. Moreover, disks that are not mentioned in the BIOS Setup are not seen by the BIOS. This means that, e.g., in an IDE−only system where hdb is not given in the Setup, the geometries reported by the BIOS for the first and second disk will apply to hda and hdc.

The IDENTIFY DRIVE command

When an IDE drive is sent the IDENTIFY DRIVE (0xec) command, it will return 256 words (512 bytes) of information. This contains lots of technical stuff. Let us only describe here what plays a role in geometry matters. The words are numbered 0−255.

We find three pieces of information here: DefaultCHS (words 1,3,6), CurrentCHS (words 54−58) and LBAcapacity (words 60−61).

	Description	Example
0	bit field: bit 6: fixed disk, bit 7: removable medium	0x0040
1	Default number of cylinders	16383
3	Default number of heads	16
6	Default number of sectors per track	63
10−19	Serial number (in ASCII)	K8033FEC
23−26	Firmware revision (in ASCII)	DA620CQ0
27−46	Model name (in ASCII)	Maxtor 54098U8
49	bit field: bit 9: LBA supported	0x2f00
53	bit field: bit 0: words 54−58 are valid	0x0007
54	Current number of cylinders	16383
55	Current number of heads	16
56	Current number of sectors per track	63
57−58	Current total number of sectors	16514064
60−61	Default total number of sectors	80041248
255	Checksum and signature (0xa5)	0xf9a5

In the ASCII strings each word contains two characters, the high order byte the first, the low order byte the second. The 32−bit values are given with low order word first. Words 54−58 are set by the command INITIALIZE DRIVE PARAMETERS (0x91). They are significant only when CHS addressing is used, but may help to find the actual disk size in case the disk sets DefaultCHS to 4092/16/63 in order to avoid BIOS problems.

Sometimes, when a jumper causes a big drive to misreport LBAcapacity (often to 66055248 sectors, in order to stay below the 33.8 GB limit), one needs a fourth piece of information to find the actual disk size, namely the result of the READ NATIVE MAX ADDRESS (0xf8) command.

10.2 SCSI details

The situation for SCSI is slightly different, as the SCSI commands already use logical block numbers, so a \`geometry' is entirely irrelevant for actual I/O. However, the format of the partition table is still the same, so `fdisk` has to invent some geometry, and also uses `HDIO_GETGEO` here − indeed, `fdisk` does not distinguish between IDE and SCSI disks. As one can see from the detailed description below, the various drivers each invent a somewhat different geometry. Indeed, one big mess.

If you are not using DOS or so, then avoid all extended translation settings, and just use 64 heads, 32 sectors per track (for a nice, convenient 1 MiB per cylinder), if possible, so that no problems arise when you move the disk from one controller to another. Some SCSI disk drivers (aha152x, pas16, ppa, qlogicfas, qlogicisp) are so nervous about DOS compatibility that they will not allow a Linux−only system to use more than about 8 GiB. This is a bug.

What is the real geometry? The easiest answer is that there is no such thing. And if there were, you wouldn't want to know, and certainly NEVER, EVER tell `fdisk` or LILO or the kernel about it. It is strictly a business between the SCSI controller and the disk. Let me repeat that: only silly people tell `fdisk`/LILO/kernel about the true SCSI disk geometry.

But if you are curious and insist, you might ask the disk itself. There is the important command READ CAPACITY that will give the total size of the disk, and there is the MODE SENSE command, that in the Rigid Disk Drive Geometry Page (page 04) gives the number of cylinders and heads (this is information that cannot be changed), and in the Format Page (page 03) gives the number of bytes per sector, and sectors per track. This latter number is typically dependent upon the notch, and the number of sectors per track varies − the outer tracks have more sectors than the inner tracks. The Linux program `scsiinfo` will give this information. There are many details and complications, and it is clear that nobody (probably not even the operating system) wants to use this information. Moreover, as long as we are only concerned about `fdisk` and LILO, one typically gets answers like C/H/S=4476/27/171 − values that cannot be used by `fdisk` because the partition table reserves only 10 resp. 8 resp. 6 bits for C/H/S.

Then where does the kernel `HDIO_GETGEO` get its information from? Well, either from the SCSI controller, or by making an educated guess. Some drivers seem to think that we want to know `reality', but of course we only want to know what the DOS or OS/2 FDISK (or Adaptec AFDISK, etc) will use.

Note that Linux `fdisk` needs the numbers H and S of heads and sectors per track to convert LBA sector numbers into c/h/s addresses, but the number C of cylinders does not play a role in this conversion. Some drivers use (C,H,S) = (1023,255,63) to signal that the drive capacity is at least 1023*255*63 sectors. This is unfortunate, since it does not reveal the actual size, and will limit the users of most `fdisk` versions to about 8 GiB of their disks − a real limitation in these days.

In the description below, M denotes the total disk capacity, and C, H, S the number of cylinders, heads and sectors per track. It suffices to give H, S if we regard C as defined by M / (H*S).

By default, H=64, S=32.

aha1740, dtc, g_NCR5380, t128, wd7000:

> H=64, S=32.

aha152x, pas16, ppa, qlogicfas, qlogicisp:

> H=64, S=32 unless C > 1024, in which case H=255, S=63, C = min(1023, M/(H*S)). (Thus C is truncated, and H*S*C is not an approximation to the disk capacity M. This will confuse most versions of `fdisk`.) The `ppa.c` code uses M+1 instead of M and says that due to a bug in `sd.c` M is off by 1.

advansys:

H=64, S=32 unless C > 1024 and moreover the `> 1 GB' option in the BIOS is enabled, in which case H=255, S=63.

aha1542:

Ask the controller which of two possible translation schemes is in use, and use either H=255, S=63 or H=64, S=32. In the former case there is a boot message "aha1542.c: Using extended bios translation".

aic7xxx:

H=64, S=32 unless C > 1024, and moreover either the "extended" boot parameter was given, or the `extended' bit was set in the SEEPROM or BIOS, in which case H=255, S=63. In Linux 2.0.36 this extended translation would always be set in case no SEEPROM was found, but in Linux 2.2.6 if no SEEPROM is found extended translation is set only when the user asked for it using this boot parameter (while when a SEEPROM is found, the boot parameter is ignored). This means that a setup that works under 2.0.36 may fail to boot with 2.2.6 (and require the `linear` keyword for LILO, or the `aic7xxx=extended` kernel boot parameter).

buslogic:

H=64, S=32 unless C >= 1024, and moreover extended translation was enabled on the controller, in which case if M < 2^22 then H=128, S=32; otherwise H=255, S=63. However, after making this choice for (C,H,S), the partition table is read, and if for one of the three possibilities (H,S) = (64,32), (128,32), (255,63) the value endH=H−1 is seen somewhere then that pair (H,S) is used, and a boot message is printed "Adopting Geometry from Partition Table".

fdomain:

Find the geometry information in the BIOS Drive Parameter Table, or read the partition table and use H=endH+1, S=endS for the first partition, provided it is nonempty, or use H=64, S=32 for M < 2^21 (1 GiB), H=128, S=63 for M < 63*2^17 (3.9 GiB) and H=255, S=63 otherwise.

in2000:

Use the first of (H,S) = (64,32), (64,63), (128,63), (255,63) that will make C <= 1024. In the last case, truncate C at 1023.

seagate:

Read C,H,S from the disk. (Horrors!) If C or S is too large, then put S=17, H=2 and double H until C <= 1024. This means that H will be set to 0 if M > 128*1024*17 (1.1 GiB). This is a bug.

ultrastor and u14_34f:

One of three mappings ((H,S) = (16,63), (64,32), (64,63)) is used depending on the controller mapping mode.

If the driver does not specify the geometry, we fall back on an educated guess using the partition table, or using the total disk capacity.

Look at the partition table. Since by convention partitions end on a cylinder boundary, we can, given `end = (endC,endH,endS)` for any partition, just put H = `endH+1` and S = `endS`. (Recall that sectors are counted from 1.) More precisely, the following is done. If there is a nonempty partition, pick the partition with the largest `beginC`. For that partition, look at `end+1`, computed both by adding `start` and `length` and by assuming that this partition ends on a cylinder boundary. If both values agree, or if `endC` = 1023 and `start+length` is an integral multiple of `(endH+1)*endS`, then assume that this partition really was aligned on a cylinder boundary, and put H = `endH+1` and S = `endS`. If this fails, either because there are no partitions, or because they have strange sizes, then look only at the disk capacity M. Algorithm: put H = M/(62*1024) (rounded up), S = M/(1024*H) (rounded up), C = M/(H*S) (rounded down). This has the effect of producing a (C,H,S) with C at most 1024 and S at most 62.

11. The Linux IDE 8 GiB limit

The Linux IDE driver gets the geometry and capacity of a disk (and lots of other stuff) by using an ATA IDENTIFY request. Until recently the driver would not believe the returned value of lba_capacity if it was more than 10% larger than the capacity computed by C*H*S. However, by industry agreement large IDE disks (with more than 16514064 sectors) return C=16383, H=16, S=63, for a total of 16514064 sectors (7.8 GB) independent of their actual size, but give their actual size in lba_capacity.

Recent Linux kernels (2.0.34, 2.1.90) know about this and do the right thing. If you have an older Linux kernel and do not want to upgrade, and this kernel only sees 8 GiB of a much larger disk, then try changing the routine `lba_capacity_is_ok` in `/usr/src/linux/drivers/block/ide.c` into something like

```
static int lba_capacity_is_ok (struct hd_driveid *id) {
        id->cyls = id->lba_capacity / (id->heads * id->sectors);
        return 1;
}
```

For a more cautious patch, see 2.1.90.

11.1 BIOS complications

As just mentioned, large disks return the geometry C=16383, H=16, S=63 independent of the actual size, while the actual size is returned in the value of LBAcapacity. Some BIOSes do not recognize this, and translate this 16383/16/63 into something with fewer cylinders and more heads, for example 1024/255/63 or 1027/255/63. So, the kernel must not only recognize the single geometry 16383/16/63, but also all BIOS−mangled versions of it. Since 2.2.2 this is done correctly (by taking the BIOS idea of H and S, and computing C = capacity/(H*S)). Usually this problem is solved by setting the disk to Normal in the BIOS setup (or, even better, to None, not mentioning it at all to the BIOS). If that is impossible because you have to boot from it or use it also with DOS/Windows, and upgrading to 2.2.2 or later is not an option, use kernel boot parameters.

If a BIOS reports 16320/16/63, then this is usually done in order to get 1024/255/63 after translation.

There is an additional problem here. If the disk was partitioned using a geometry translation, then the kernel may at boot time see this geometry used in the partition table, and report `hda: [PTBL] [1027/255/63]`. This is bad, because now the disk is only 8.4 GB. This was fixed in 2.3.21. Again, kernel boot parameters will help.

11.2 Jumpers that select the number of heads

Many disks have jumpers that allow you to choose between a 15−head an a 16−head geometry. The default settings will give you a 16−head disk. Sometimes both geometries address the same number of sectors, sometimes the 15−head version is smaller. There may be a good reason for this setup: Petri Kaukasoina writes: `A 10.1 Gig IBM Deskstar 16 GP (model IBM−DTTA−351010) was jumpered for 16 heads as default but this old PC (with AMI BIOS) didn't boot and I had to jumper it for 15 heads. hdparm −i tells RawCHS=16383/15/63 and LBAsects=19807200. I use 20960/15/63 to get the full capacity.' For the jumper settings, see http://www.storage.ibm.com/techsup/hddtech/hddtech.htm.

11.3 Jumpers that clip total capacity

Many disks have jumpers that allow you to make the disk appear smaller than it is. A silly thing to do, and probably no Linux user ever wants to use this, but some BIOSes crash on big disks. The usual solution is to keep the disk entirely out of the BIOS setup. But this may be feasible only if the disk is not your boot disk.

The first serious limit was the 4096 cylinder limit (that is, with 16 heads and 63 sectors/track, 2.11 GB). For example, a Fujitsu MPB3032ATU 3.24 GB disk has default geometry 6704/15/63, but can be jumpered to appear as 4092/16/63, and then reports LBAcapacity 4124736 sectors, so that the operating system cannot guess that it is larger in reality. In such a case (with a BIOS that crashes if it hears how big the disk is in reality, so that the jumper is required) one needs boot parameters to tell Linux about the size of the disk.

That is unfortunate. Most disks can be jumpered so as to appear as a 2 GB disk and then report a clipped geometry like 4092/16/63 or 4096/16/63, but still report full LBAcapacity. Such disks will work well, and use full capacity under Linux, regardless of jumper settings.

A more recent limit is the 33.8 GB limit. Linux kernels older than 2.2.14 / 2.3.21 need a patch to be able to cope with IDE disks larger than this.

With an old BIOS and a disk larger than 33.8 GB, the BIOS may hang, and in such cases booting may be impossible, even when the disk is removed from the CMOS settings. See also the BIOS 33.8 GB limit.

Therefore, large IBM and Maxtor disks come with a jumper that make the disk appear as a 33.8 GB disk. For example, the IBM Deskstar 37.5 GB (DPTA−353750) with 73261440 sectors (corresponding to 72680/16/63, or 4560/255/63) can be jumpered to appear as a 33.8 GB disk, and then reports geometry 16383/16/63 like any big disk, but LBAcapacity 66055248 (corresponding to 65531/16/63, or 4111/255/63). Similar things hold for recent large Maxtor disks.

With the jumper present, both the geometry (16383/16/63) and the size (66055248) are conventional and give no information about the actual size. Moreover, attempts to access sector 66055248 and above yield I/O errors. However, on Maxtor drives the actual size can be found and made accessible using the READ NATIVE MAX ADDRESS and SET MAX ADDRESS commands. Presumably this is what MaxBlast/EZ−Drive does. Now there is also a small Linux utility setmax.c for this, and also a kernel patch has been published.

Early large Maxtor disks have an additional detail: the J46 jumper for these 34−40 GB disks changes the geometry from 16383/16/63 to 4092/16/63 and does not change the reported LBAcapacity. This means that also with jumper present the BIOS (old Award 4.5*) will hang at boot time. For this case Maxtor provides a utility JUMPON.EXE that upgrades the firmware to make J46 behave as described above.

On recent Maxtor drives the call `setmax -d 0 /dev/hdX` will give you max capacity again. However, on slightly older drives a firmware bug does not allow you to use `-d 0`, and `setmax -d 255 /dev/hdX` returns you to almost full capacity.

For IBM things are worse: the jumper really clips capacity and there is no software way to get it back. The solution is not to use the jumper but use `setmax -m 66055248 /dev/hdX` to software−clip the disk. ("How?" you say − "I cannot boot!". IBM gives the tip: *If a system with Award BIOS hangs during drive detection: Reboot the system and hold the F4 key to bypass autodetection of the drive(s).* If this doesn't help, find a different computer, connect the drive to it, and run `setmax` there. After doing this you go back to the first machine and are in the same situation as with jumpered Maxtor disks: booting works, and after getting past the BIOS either a patched kernel or a `setmax -d 0` gets you full capacity.

12. The Linux 65535 cylinder limit

The `HDIO_GETGEO` ioctl returns the number of cylinders in a short. This means that if you have more than 65535 cylinders, the number is truncated, and (for a typical SCSI setup with 1 MiB cylinders) a 80 GiB disk may appear as a 16 GiB one. Once one recognizes what the problem is, it is easily avoided.

12.1 IDE problems with 34+ GB disks

Drives larger than 33.8 GB will not work with kernels older than 2.2.14 / 2.3.21. The details are as follows. Suppose you bought a new IBM−DPTA−373420 disk with a capacity of 66835440 sectors (34.2 GB). Pre−2.3.21 kernels will tell you that the size is 769*16*63 = 775152 sectors (0.4 GB), which is a bit disappointing. And giving command line parameters hdc=4160,255,63 doesn't help at all − these are just ignored. What happens? The routine idedisk_setup() retrieves the geometry reported by the disk (which is 16383/16/63) and overwrites what the user specified on the command line, so that the user data is used only for the BIOS geometry. The routine current_capacity() or idedisk_capacity() recomputes the cylinder number as 66835440/(16*63)=66305, but since this is stored in a short, it becomes 769. Since lba_capacity_is_ok() destroyed id−>cyls, every following call to it will return false, so that the disk capacity becomes 769*16*63. For several kernels a patch is available. A patch for 2.0.38 can be found at ftp.kernel.org. A patch for 2.2.12 can be found at www.uwsg.indiana.edu (some editing may be required to get rid of the html markup). The 2.2.14 kernels do support these disks. In the 2.3.* kernel series, there is support for these disks since 2.3.21. One can also `solve' the problem in hardware by using a jumper to clip the size to 33.8 GB. In many cases a BIOS upgrade will be required if one wants to boot from the disk.

13. Extended and logical partitions

Above, we saw the structure of the MBR (sector 0): boot loader code followed by 4 partition table entries of 16 bytes each, followed by an AA55 signature. Partition table entries of type 5 or F or 85 (hex) have a special significance: they describe *extended* partitions: blobs of space that are further partitioned into *logical* partitions. (So, an extended partition is only a box, it cannot be used itself, one uses the logical partitions inside.) Only the location of the first sector of an extended partition is important. This first sector contains a partition table with four entries: one a logical partition, one an extended partition, and two unused. In this way one gets a chain of partition table sectors, scattered over the disk, where the first one describes three primary partitions and the extended partition, and each following partition table sector describes one logical partition and the location of the next partition table sector.

It is important to understand this: When people do something stupid while partitioning a disk, they want to know: Is my data still there? And the answer is usually: Yes. But if logical partitions were created then the partition table sectors describing them are written at the beginning of these logical partitions, and data that was there before is lost.

The program sfdisk will show the full chain. E.g.,

```
# sfdisk -l -x /dev/hda

Disk /dev/hda: 16 heads, 63 sectors, 33483 cylinders
Units = cylinders of 516096 bytes, blocks of 1024 bytes, counting from 0

   Device Boot Start     End   #cyls   #blocks   Id  System
/dev/hda1          0+    101     102-    51376+  83  Linux
/dev/hda2        102    2133    2032   1024128   83  Linux
/dev/hda3       2134   33482   31349  15799896    5  Extended
/dev/hda4          0       -       0         0    0  Empty

/dev/hda5       2134+   6197    4064-  2048224+  83  Linux
    -           6198   10261    4064   2048256    5  Extended
    -           2134    2133       0         0    0  Empty
    -           2134    2133       0         0    0  Empty

/dev/hda6       6198+  10261    4064-  2048224+  83  Linux
    -          10262   16357    6096   3072384    5  Extended
    -           6198    6197       0         0    0  Empty
    -           6198    6197       0         0    0  Empty
...
/dev/hda10     30581+  33482    2902-  1462576+  83  Linux
    -          30581   30580       0         0    0  Empty
    -          30581   30580       0         0    0  Empty
    -          30581   30580       0         0    0  Empty

#
```

It is possible to construct bad partition tables. Many kernels get into a loop if some extended partition points back to itself or to an earlier partition in the chain. It is possible to have two extended partitions in one of these partition table sectors so that the partition table chain forks. (This can happen for example with an fdisk that does not recognize each of 5, F, 85 as an extended partition, and creates a 5 next to an F.) No standard fdisk type program can handle such situations, and some handwork is required to repair them. The Linux kernel will accept a fork at the outermost level. That is, you can have two chains of logical partitions. Sometimes this is useful – for example, one can use type 5 and be seen by DOS, and the other type 85, invisible for DOS, so that DOS FDISK will not crash because of logical partitions past cylinder 1024. Usually one needs `sfdisk` to create such a setup.

---

14. Problem solving

Many people think they have problems, while in fact nothing is wrong. Or, they think that the problems they have are due to disk geometry, while in fact disk geometry has nothing to do with the matter. All of the above may have sounded complicated, but disk geometry handling is extremely easy: do nothing at all, and all is fine; or perhaps give LILO the keyword `lba32` if it doesn't get past `LI' when booting. Watch the kernel boot messages, and remember: the more you fiddle with geometries (specifying heads and cylinders to LILO and fdisk and on the kernel command line) the less likely it is that things will work. Roughly speaking, all is fine by default.

And remember: nowhere in Linux is disk geometry used, so no problem you have while running Linux can be caused by disk geometry. Indeed, disk geometry is used only by LILO and by fdisk. So, if LILO fails to boot the kernel, that may be a geometry problem. If different operating systems do not understand the partition table, that may be a geometry problem. Nothing else. In particular, if mount doesnt seem to work, never worry about disk geometry -- the problem is elsewhere.

14.1 Problem: My IDE disk gets a bad geometry when I boot from SCSI.

It is quite possible that a disk gets the wrong geometry. The Linux kernel asks the BIOS about hd0 and hd1 (the BIOS drives numbered 80H and 81H) and assumes that this data is for hda and hdb. But on a system that boots from SCSI, the first two disks may well be SCSI disks, and thus it may happen that the fifth disk, which is the first IDE disk hda, gets assigned a geometry belonging to sda. Such things are easily solved by giving boot parameters `hda=C,H,S' for the appropriate numbers C, H and S, either at boot time or in /etc/lilo.conf.

14.2 Nonproblem: Identical disks have different geometry?

`I have two identical 10 GB IBM disks. However, fdisk gives different sizes for them. Look:

```
# fdisk -l /dev/hdb
Disk /dev/hdb: 255 heads, 63 sectors, 1232 cylinders
Units = cylinders of 16065 * 512 bytes

   Device Boot  Start      End   Blocks   Id  System
/dev/hdb1           1     1232  9896008+  83  Linux native
# fdisk -l /dev/hdd
Disk /dev/hdd: 16 heads, 63 sectors, 19650 cylinders
Units = cylinders of 1008 * 512 bytes

   Device Boot  Start      End   Blocks   Id  System
/dev/hdd1           1    19650  9903568+  83  Linux native
```

How come?'

What is happening here? Well, first of all these drives really are 10gig: hdb has size 255*63*1232*512 = 10133544960, and hdd has size 16*63*19650*512 = 10141286400, so, nothing is wrong and the kernel sees both as 10.1 GB. Why the difference in size? That is because the kernel gets data for the first two IDE disks from the BIOS, and the BIOS has remapped hdb to have 255 heads (and 16*19650/255=1232 cylinders). The rounding down here costs almost 8 MB.

If you would like to remap hdd in the same way, give the kernel boot parameters `hdd=1232,255,63'.

14.3 Nonproblem: fdisk sees much more room than df?

fdisk will tell you how many blocks there are on the disk. If you make a filesystem on the disk, say with mke2fs, then this filesystem needs some space for bookkeeping -- typically something like 4% of the filesystem size, more if you ask for a lot of inodes during mke2fs. For example:

```
# sfdisk -s /dev/hda9
4095976
# mke2fs -i 1024 /dev/hda9
```

```
mke2fs 1.12, 9-Jul-98 for EXT2 FS 0.5b, 95/08/09
...
204798 blocks (5.00%) reserved for the super user
...
# mount /dev/hda9 /somewhere
# df /somewhere
Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hda9            3574475      13  3369664      0%   /mnt
# df -i /somewhere
Filesystem           Inodes   IUsed   IFree  %IUsed Mounted on
/dev/hda9            4096000      11 4095989     0%  /mnt
#
```

We have a partition with 4095976 blocks, make an ext2 filesystem on it, mount it somewhere and find that it only has 3574475 blocks – 521501 blocks (12%) was lost to inodes and other bookkeeping. Note that the difference between the total 3574475 and the 3369664 available to the user are the 13 blocks in use plus the 204798 blocks reserved for root. This latter number can be changed by tune2fs. This `–i 1024' is only reasonable for news spools and the like, with lots and lots of small files. The default would be:

```
# mke2fs /dev/hda9
# mount /dev/hda9 /somewhere
# df /somewhere
Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hda9            3958475      13  3753664      0%   /mnt
# df -i /somewhere
Filesystem           Inodes   IUsed   IFree  %IUsed Mounted on
/dev/hda9            1024000      11 1023989     0%  /mnt
#
```

Now only 137501 blocks (3.3%) are used for inodes, so that we have 384 MB more than before. (Apparently, each inode takes 128 bytes.) On the other hand, this filesystem can have at most 1024000 files (more than enough), against 4096000 (too much) earlier.