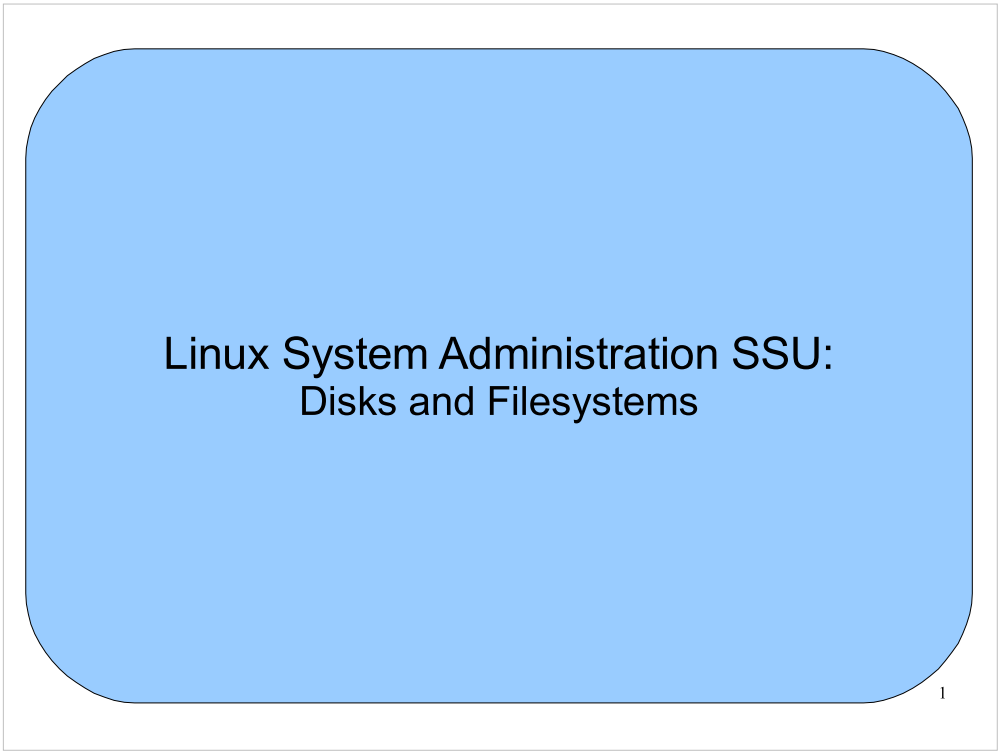
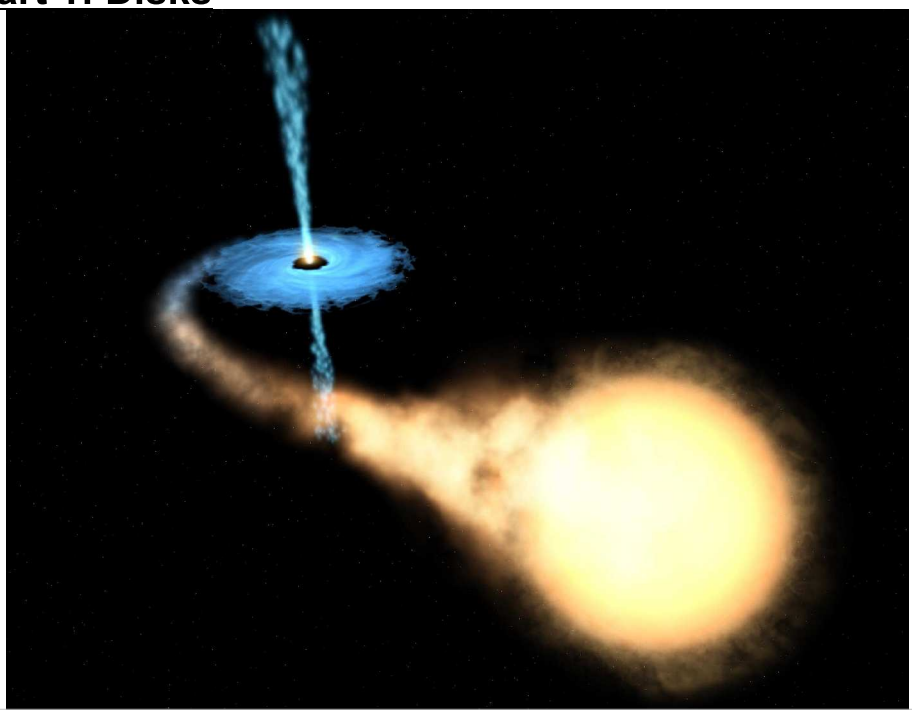
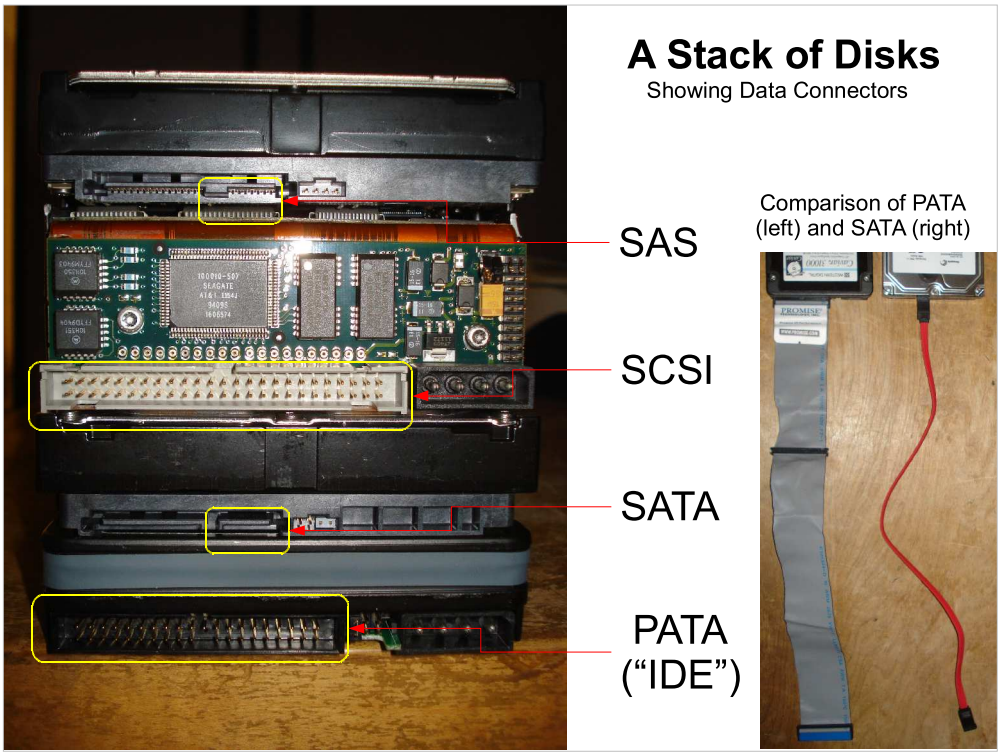


This time we'll talk about filesystems. We'll start out by looking at disk partitions, which are the traditional places to put filesystems. Then we'll take a look at “logical volumes”, which are an abstraction that moves us away from physical disk partitions. We'll also take a quick look at file permissions, attributes and ACLs.

Part 1: Disks



First, we'll take a quick look at current disk technology, and then we'll talk about some of the problems that loom on the horizon.



Here are four different types of disks, all of the same width. Each is a standard "3.5-inch" disk, and would fit into the same slot as any of the others. The two most common types of disks are Parallel ATA (PATA) disks (sometimes loosely called "IDE" disks) and Serial ATA (SATA) disks. SATA is the successor to PATA, and is found almost universally in new computers.

The move from a parallel bus to a serial bus was driven by speed. With a parallel bus, crosstalk between adjacent wires becomes more and more of a problem with increasing signal frequency. Similarly, older SCSI disks (another parallel interface, found mostly in servers) are being phased out in favor of Serial-Attached SCSI (SAS).

Disk Drive Form Factors

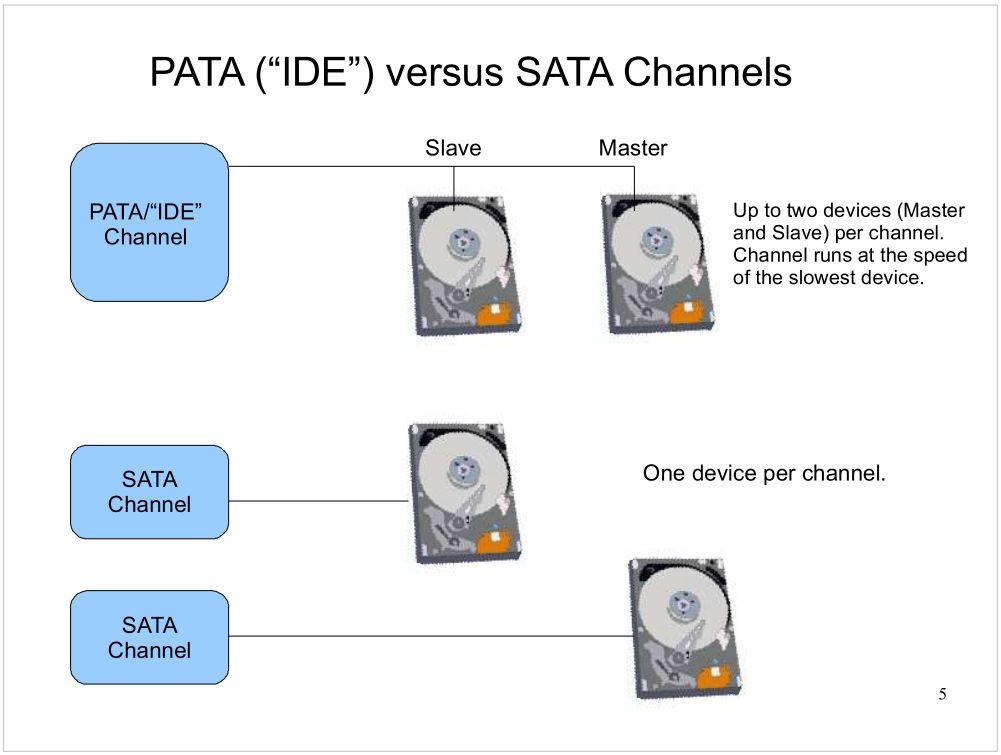
"2.5-inch" - Initially used in laptops and other restricted spaces. Increasingly used for low power consumption.

"3.5-inch" - Was the size of drives accommodating 3.5-inch floppies. Now the most common size for desktop and server hard disks.

"5.25-inch" - Originally the size of drives accommodating 5.25-inch floppies. Still used for CD/DVD drives in desktop computers.

Not shown: "1.8-inch" - Ultra-small form factor for very small laptops and other cramped spaces.

This shows some of the form factors used for disks. Note that the names of the form factors don't reflect the actual physical dimensions of the disks.



Another difference between PATA and SATA is the number of devices per channel.

Disk Failure Rates (CMU: Schroeder and Gibson)

From a study of 100,000 disks:

* For drives less than five years old, actual failure rates were larger than manufacturer's predictions by a factor of 2–10. For five to eight year old drives, failure rates were a factor of 30 higher than manufacturer's predictions.

* Failure rates of SATA disks are not worse than the replacement rates of SCSI or Fibre Channel disks. This may indicate that disk independent factors, such as operating conditions, usage and environmental factors, affect failure rates more than inherent flaws.

* Wear-out starts early, and continues throughout the disk's lifetime.

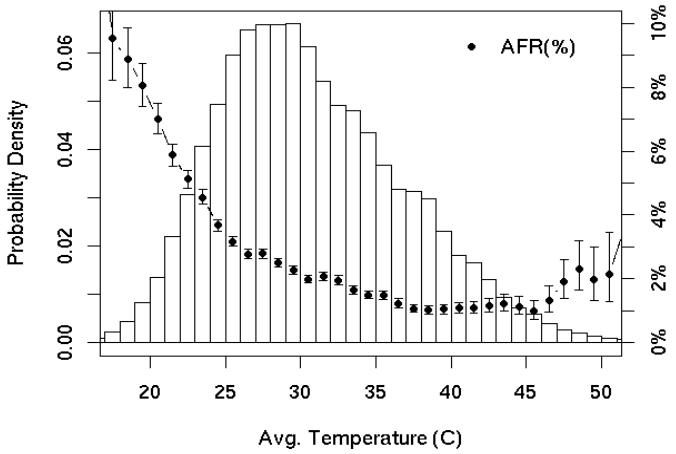
Schroeder and Gibson, in Proceedings of the 5th USENIX Conference on File and Storage Technologies
http://www.usenix.org/events/fast07/tech/schroeder/schroeder.pdf

There have been several recent studies of disk failure rates. I'll talk about a couple of particularly interesting ones, done at CMU and Google. These are some things to keep in mind when buying disks, thinking about backup strategies, or budgeting for replacement costs.

Disk Failure Rates (Google)

From a study of more than 100,000 disks:

* Disk may actually like higher temperatures



Penheiro, Weber and Barroso, in Proceedings of the 5th USENIX Conference on File and Storage Technologies
http://research.google.com/archive/disk_failures.pdf

7

The Google report confirms many of the CMU findings, and adds some interesting new finding. For example, our long-standing assumption that disks are more likely to fail at higher temperatures may not be correct. Maybe we could save money by leaving our server rooms at a higher temperature, or by eliminating some of the fans inside computers.

When manufacturers test disks, they can't run them for five years to see if they fail. Typically, they try to simulate long lifetimes by running the disks for a shorter time under extreme conditions (high temperature, for example). It may be that, because of this, manufacturers have been inadvertently selecting disk designs that prefer to run at higher temperatures.

The Problem of Error Rates (Robin Harris):

"With 12 TB of capacity in the remaining RAID 5 stripe and an URE rate of 10^14, you are highly likely to encounter a URE. Almost certain, if the drive vendors are right."

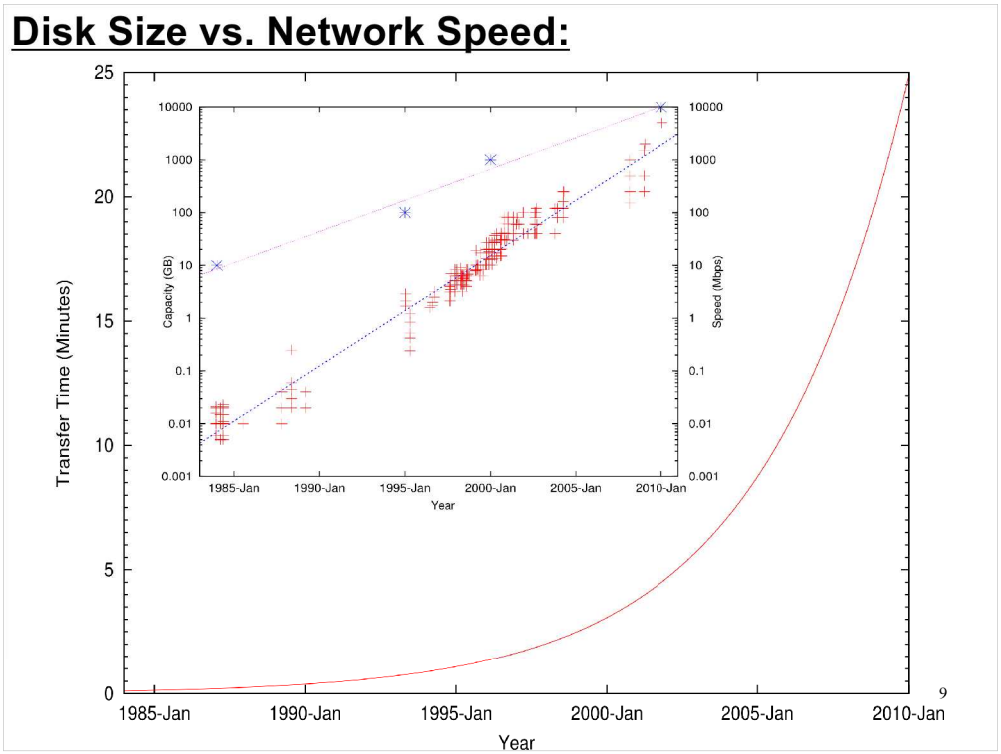
...

"The key point that seems to be missed in many of the comments is that when a disk fails in a RAID 5 array and it has to rebuild there is a significant chance of a non-recoverable read error during the rebuild (BER / UER). As there is no longer any redundancy the RAID array cannot rebuild, this is not dependent on whether you are running Windows or Linux, hardware or software RAID 5, it is simple mathematics. An honest RAID controller will log this and generally abort, allowing you to restore undamaged data from backup onto a fresh array. "

http://blogs.zdnet.com/storage/?p=162

This recent blog post by Robin Harris got a lot of attention. Manufacturers cite what's called an "Unrecoverable Read Error" (URE) rate for disks. This is a measure of the probability that a given bit of data will suddenly, and permanently, become unreadable. In the past, a URE rate of 1 in 10^14 has been acceptable, but as disks get bigger, it's becoming more and more likely that you'll encounter a URE when you read from the disk. The thing to note is that URE rates haven't kept up with disk sizes, and this is becoming a problem.

Disk Size vs. Network Speed:



Here's something I noticed recently. The smaller, inset plot shows how disk capacity (red marks) and ethernet speed (blue marks) have increased over the years. Note that disk capacity doubles approximately every two years, but ethernet speed only doubles every four years or so. To see what this means, look at the larger graph.

This graph shows the amount of time needed to transfer the entire contents of a current disk to another similar disk across the network, assuming that the only limiting factor is network speed. As you can see, this transfer time is steadily increasing, because network speeds aren't keeping pace with the increasing capacity of disks.

If this trend continues, it means that, in the future, we'll need to think more and more carefully about where our data lives.

Part 2: Partitions

A partition is just a section of a hard disk. We'll look at why we'd want to chop up a hard disk into partitions, but we'll start by looking at the structure of a hard disk.

Disk Geometry:

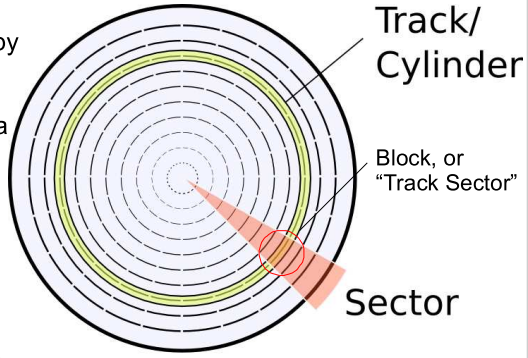
Disks are made of stacks of spinning platters, each surface of which is read by an independent "read head".

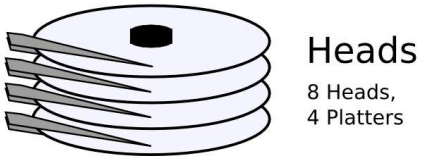
Originally, the position of a piece of data on a disk was given by the coordinates C,H and S, for "Cylinder", "Head" and "Sector".

The intersection of a cylinder with a platter surface is a "Track".

The intersection of a sector with a track is a "Block". Confusingly, the terms "Track Sector" or just "Sector" are also often used to refer to blocks.

Today, the CHS coordinates don't really refer to where the data is actually located on the disk. They're just abstractions. A more recent coordinate scheme, "Logical Block Addressing" (LBA) just numbers the blocks on the disk, starting with zero.





Each block is typically 512 bytes.

The CHS coordinate system began with floppy disks, where the (c,h,s) values really told you where to find the data. Some reasons CHS doesn't really tell you where the data is on a modern hard disk:

- As disks became smarter, they began transparently hiding bad blocks and substituting good blocks from a pool of spares.
- These disks also try to optimize I/O performance, so they want to choose where to really put the data.
- You can have arrays of disks (e.g. RAID) that appear (to the operating system) to be one disk.
- The same addressing scheme can be applied to non-disk devices, like solid-state disks.

If (c,h,s) is hard to grasp, realize that it's just equivalent to (r,z,θ). They're coordinates in a cylindrical coordinate system.

Partitions:

Sometimes, it's useful to split up a disk into smaller pieces, called “partitions”. Some motivations for this are:

• The operating system may not be able to use storage devices as large as the whole disk.

• You may want to install multiple operating systems.

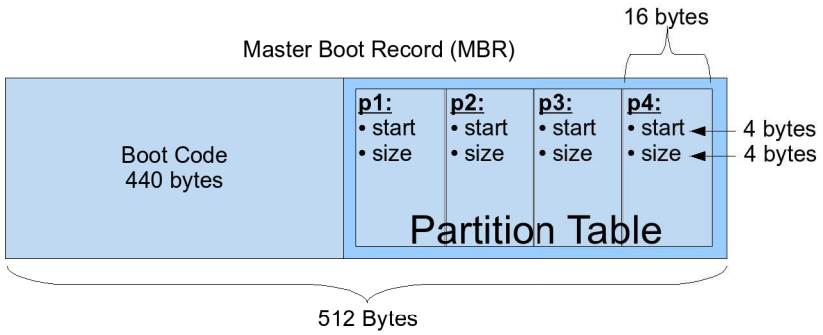
• You may want to designate one partition as swap space.

• You may want to prevent one part of your storage from filling up the whole disk.

One potential problem with having multiple partitions on a disk is that partitions are generally difficult to re-size after they are created.

Partition Table:

The first block on a disk is the "Master Boot Record" (MBR). This block contains a boot program, used to boot an operating system, and a "partition table". The partition table contains slots for up to four primary partitions. For each partition, the table specifies the starting position of the partition (in LBA coordinates) and the size of the partition (in blocks). Since the size is stored as a 4-byte value, the size of a partition is limited to about 2 Terabytes.



13

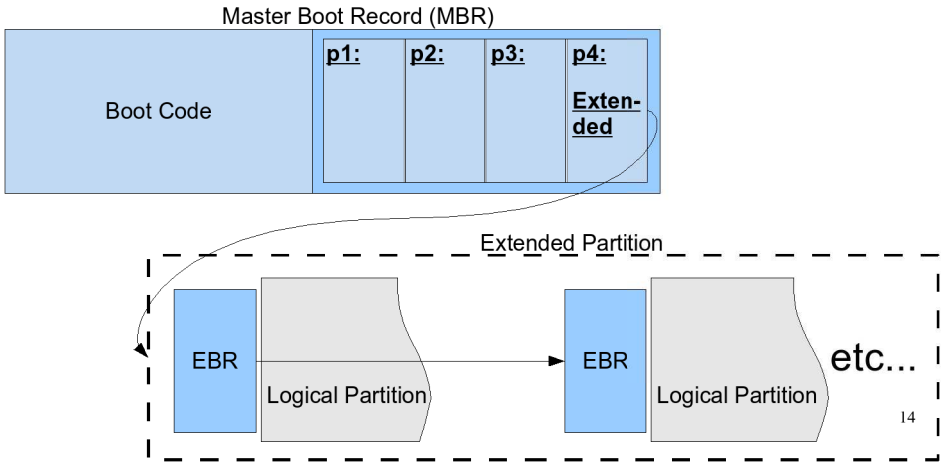
Each disk will have at least one partition. Note that you can only have up to four primary partitions. We'll talk about how to get around the 2 Terabyte size limit later.

The MBR also contains a few other values, like a disk signature, but you can see by adding up the numbers that the boot code and the parition table make up the bulk of the MBR.

In LBA coordinates, the MBR is LBA=0.

Extended Partitions:

Any one (but only one) of the primary partitions can be an "extended partition". At the start of an extended partition is an "Extended Boot Record" (EBR). This describes a "logical partition" within the extended partition. If there's more than one logical partition, the EBR will also point to the next logical partition in the chain. There's no fixed limit on the number of logical partitions.



In practice, you seldom see disks with more than half a dozen partitions. These days, the typical desktop Linux computer's disk has only two or three.

EFI and GUID Partition Tables:

The successor to the PC BIOS is called “Extensible Firmware Interface” (EFI). Currently, Intel-based Macintosh computers are the only common computers that use EFI instead of a BIOS, but it may become more common as time goes by.

Instead of an MBR-based partition table, EFI uses a different scheme, called a “GUID Partition Table” (GPT).

GPT uses 8 bytes to store addresses, so the maximum size of partitions is about 16 Exabytes (16×10^{18} bytes). That should hold us for the near future.

You can create GPT partition tables using GNU parted. See the following page for an example:
http://portal.itauth.com/2008/01/17/creating-large-2tb-linux-partitions

(Linux bootloaders like Grub can boot GPT partitions as well as the older MBR-style partitions.)

Compare this with the way we'll use fdisk, later, to create partitions.

Disk and Partition Files in /dev:

In Linux, each whole disk drive or partition is represented by a special file in the /dev directory. Programs manipulate the disks and partitions by using these special files. The files have different names, depending on the type of disk.

- **SATA, SCSI, USB or Firewire Disks:**
  These disks are represented by files named /dev/sd[a-z]. They're named in the order they're detected at boot time. Partitions have names like “sda1”, “sda2”, etc.

- **IDE/PATA Disks:**
  These disks are represented by files named /dev/hd[a-z]. The disk names will be:
  - **hda** -- Master disk on the 1st IDE channel.
  - **hdb** -- Slave disk on the 1st IDE channel.
  - **hdc** -- Master disk on the 2nd IDE channel.
  - **hdd** -- Slave disk on the 2nd IDE channel.

  ...etc.

  Partitions on each disk are numbered sequentially, starting with 1. Thus, the first partition on the master disk on the first IDE channel would be “hda1”, the second would be “hda2”, etc.

Part 3: Manipulating Partitions

17

Now we'll look at how to create and otherwise manipulate partitions on a disk. This can be dangerous work. You always need to be careful about which disk you're working on. I've tried to indicate clearly which commands require special caution.

Viewing Partitions with “fdisk”:

You can use the “fdisk -l” command to view the partition layout on a disk:

```
[root@demo ~]# fdisk -l /dev/sda

Disk /dev/sda: 160.0 GB, 160000000000 bytes
255 heads, 63 sectors/track, 19452 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          13      104391   83  Linux
/dev/sda2              14       19452   156143767+  8e  Linux LVM
```

Partition Type

Near the top, you can see the number of heads, sectors and cylinders. These may not represent physical reality, but they're the way the disk presents itself to the operating system.

Fdisk reports the size of each partition in 1024-byte “blocks”. The two partitions above are about 100 MB and about 156 GB. The “+” sign on the size of the second partition means that its size isn't an integer number of 1024-byte blocks.

The “start” and “end” values are in units of cylinders, by default. You can use the “-u” switch to cause fdisk to display start and end in terms of 512-byte “track sectors”.

To add to the confusion about terms like “block” and “sector”, fdisk uses a size of 1024 bytes (not 512) when it reports the number of “blocks” in a partition. The Linux kernel uses blocks of this size, and many Linux programs will assume that a “block” is 1024 bytes. Filesystems typically use “blocks” of 1024, 2048 or 4096 bytes.

The disk-drive industry is currently pushing new standards that would make the on-disk block size 4096 bytes.

Partition Types:

Here's the list of partition types that fdisk knows about. The most common ones are highlighted.

0	Empty	1e	Hidden W95 FAT1	80	Old Minix	be	Solaris boot
1	FAT12	24	NEC DOS	81	Minix / old Lin	bf	Solaris
2	XENIX root	39	Plan 9	82	Linux swap / So	c1	DRDOS/sec (FAT-
3	XENIX usr	3c	PartitionMagic	83	Linux	c4	DRDOS/sec (FAT-
4	FAT16 <32M	40	Venix 80286	84	OS/2 hidden C:	c6	DRDOS/sec (FAT-
5	Extended	41	PPC PReP Boot	85	Linux extended	c7	Syrinx
6	FAT16	42	SFS	86	NTFS volume set	da	Non-FS data
7	HPFS/NTFS	4d	QNX4.x	87	NTFS volume set	db	CP/M / CTOS / .
8	AIX	4e	QNX4.x 2nd part	88	Linux plaintext	de	Dell Utility
9	AIX bootable	4f	QNX4.x 3rd part	8e	Linux LVM	df	BootIt
a	OS/2 Boot Manag	50	OnTrack DM	93	Amoeba	e1	DOS access
b	W95 FAT32	51	OnTrack DM6 Aux	94	Amoeba BBT	e3	DOS R/O
c	W95 FAT32 (LBA)	52	CP/M	9f	BSD/OS	e4	SpeedStor
e	W95 FAT16 (LBA)	53	OnTrack DM6 Aux	a0	IBM Thinkpad hi	eb	BeOS fs
f	W95 Ext'd (LBA)	54	OnTrackDM6	a5	FreeBSD	ee	EFI GPT
10	OPUS	55	EZ-Drive	a6	OpenBSD	ef	EFI (FAT-12/16/
11	Hidden FAT12	56	Golden Bow	a7	NeXTSTEP	f0	Linux/PA-RISC b
12	Compaq diagnost	5c	Priam Edisk	a8	Darwin UFS	f1	SpeedStor
14	Hidden FAT16 <3	61	SpeedStor	a9	NetBSD	f4	SpeedStor
16	Hidden FAT16	63	GNU HURD or Sys	ab	Darwin boot	f2	DOS secondary
17	Hidden HPFS/NTF	64	Novell Netware	b7	BSDI fs	fd	Linux raid auto
18	AST SmartSleep	65	Novell Netware	b8	BSDI swap	fe	LANstep
1b	Hidden W95 FAT3	70	DiskSecure Mult	bb	Boot Wizard hid	ff	BBT
1c	Hidden W95 FAT3	75	PC/IX				

Creating Partitions with “fdisk”:

You can use fdisk to create or delete partitions on a disk. If you type “fdisk /dev/sda”, for example, you'll be dropped into fdisk's command-line environment, where several simple one-character commands allow you to manipulate partitions on the disk.

Some fdisk commands:

- **p** Print the partition table
- **n** Create a new partition
- **d** Delete a partition
- **t** Change a partition's type
- **q** Quit without saving changes
- **w** Write the new partition table and exit

Note: In fdisk, the term “primary” partition means one that's not an “extended” partition.

```
[root@demo ~]# fdisk /dev/sdb

Command (m for help): n

Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-9726, default 1): +
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-9726,
default 9726): +40G

Command (m for help): p
Disk /dev/sdb: 80.0 GB, 80000000000 bytes
255 heads, 63 sectors/track, 9726 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start  End      Blocks   Id  System
/dev/sdb1               1 4864    39070048+  83  Linux
```


Notice that nothing you do in fdisk is actually written to the disk until you type “w”. If you decide you've made a mistake, you can always quit without saving anything by typing “q”.

Changing a Partition's Type:

Here's how to change a partition's type, using fdisk. In this example, we change the partition from the default type ("Linux") to mark it as a swap partition.

```
Command (m for help): p

Disk /dev/sdb: 80.0 GB, 80000000000 bytes
255 heads, 63 sectors/track, 9726 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               1        4864    39070048+  83  Linux
/dev/sdb2            4865        9726    39054015   83  Linux

Command (m for help): t
Partition number (1-4): 2
Hex code (type L to list codes): 82
Changed system type of partition 2 to 82 (Linux swap / Solaris)

Command (m for help): p

Disk /dev/sdb: 80.0 GB, 80000000000 bytes
255 heads, 63 sectors/track, 9726 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               1        4864    39070048+  83  Linux
/dev/sdb2            4865        9726    39054015   82  Linux swap / Solaris
```

Formatting a Swap Partition:

Before a swap partition can be used, it needs to be formatted. You can do this with the “mkswap” command:

WARNING!

```
[root@demo ~]#  mkswap /dev/sdb2
Setting up swapspace version 1, size=39054015 kB
```

WARNING!

Note that this will re-format the designated partition immediately, without asking for confirmation, so be careful!

To start using the new swap space immediately, use the “swapon” command:

```
[root@demo ~]# swapon /dev/sdb2
```

As we'll see later, you can also cause this swap partition to be used automatically, at boot time.

Here's one of those very dangerous commands. Please make sure you point mkswap at the right disk partition.

Saving Partition Layout with “sfdisk”:

You can save a partition layout into a file, so that it can later be restored. One way to do this is the “sfdisk” command. For example, this command will save the disk partitioning information into the file hda.out:

```
[root@demo ~]#  sfdisk -d /dev/hda > hda.out
```

If the disk is replaced later, or if you have another identical disk that you want to partition in the same way, you can use this command:

WARNING!

```
[root@demo ~]#  sfdisk /dev/hda < hda.out
```

WARNING!

Note that this command should be used very carefully, since it will (without asking for confirmation) wipe out any existing partition table on the disk. The content of hda.out looks like this:

```
# partition table of /dev/hda
unit: sectors

/dev/hda1 : start=       63, size=   208782, Id=83, bootable
/dev/hda2 : start=   208845, size=312287535, Id=8e
/dev/hda3 : start=        0, size=        0, Id= 0
/dev/hda4 : start=        0, size=        0, Id= 0
```


Another dangerous command.

Part 4: Filesystem Structure

In order to understand filesystems, it's important to have a little knowledge about how they're laid out on disk. Terms like “superblock” and “block group” show up in error messages sometimes, and knowing what they mean can save you a lot of grief. Here's a primer on filesystem structure.

What is a Filesystem?

A filesystem is a way of organizing data on a block device. The filesystem organizes data into “files”, each of which has a name and other metadata attributes. These files are grouped into hierarchical “directories”, making it possible to locate a particular file by specifying its name and directory path. Some of the metadata typically associated with each file are:

- **Timestamps**, recording file creation or modification times.
- **Ownership**, specifying a user or group to whom the file belongs.
- **Permissions**, specifying who has access to the file.

Linux originally used the “minix” filesystem, from the operating system of the same name, but quickly switched to what was called the “Extended Filesystem” (in 1992) followed by an improved “Second Extended Filesystem” (in 1993). The two latter filesystems were developed by French software developer Remy Card.

The Second Extended Filesystem (ext2) remained the standard Linux filesystem until the early years of the next century, when it was supplanted by the “Third Extended Filesystem” (ext3), written by Scottish software developer Stephen Tweedie.

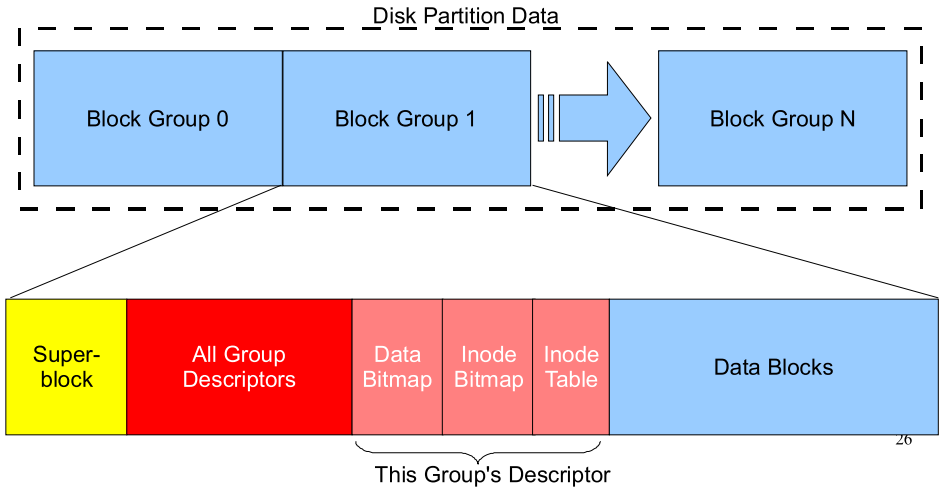
25

Linux also supports many other filesystems, including Microsoft's VFAT and NTFS, and the ISO9660 filesystem used on CDs and DVDs.

A block device is a device like a disk where you can directly address individual “blocks” of data. Linux separates devices into “character” devices, which just read and write streams of bytes, and “block” devices, in which parts of the device's storage can be directly addressed.

How ext2 and ext3 Work:

The ext2 and ext3 filesystems are very similar. Both divide a disk partition into “block groups” of a fixed size. At the beginning of each block group is metadata about the filesystem in general, and that block group in particular. There is much redundancy in this metadata, making it possible to detect and correct damage to the filesystem.

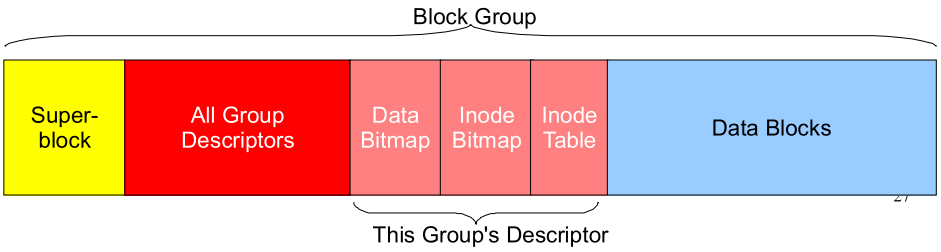


Superblocks:

The ext2/ext3 filesystem as a whole is described in a chunk of data called the "Superblock". The superblock contains:

- a name for the filesystem (a "label"),
- the size of the filesystem's block groups,
- timestamps showing when the filesystem was last mounted,
- a flag saying whether it was unmounted cleanly,
- a number showing the amount of unused space in the filesystem,

and much other information. The superblock is duplicated at the beginning of many block groups. Normally, the operating system only uses the copy at the beginning of block group 0, but if this is lost or damaged, the data can be recovered from one of the other copies. During normal operation, the operating system keeps all copies of the superblock synchronized.



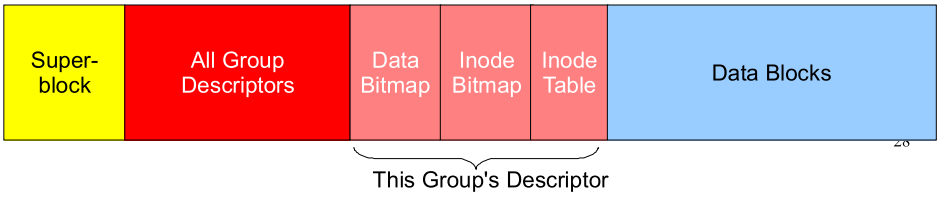
The superblock is actually duplicated at the beginning of each block group for ext2 filesytems. For ext3, there's the option of only duplicating it in some block groups. If this option is turned on (as it is by default), the superblock is only duplicated in block groups 0, 1 and powers of 3, 5 and 7.

Inodes and Group Descriptors:

Each file's data is stored in the "data blocks" section of a block group. Files are described by records stored in blocks called "index nodes" (inodes). The inodes are stored in a part of the block group called the "group descriptor". Data in each inode includes:

- the file's name,
- the file's owner,
- the group to which the file belongs,
- several timestamps,
- permission settings for the file,
- pointers to the data blocks that contain the file's data,

and other information. The group descriptors are so important that copies of the block descriptors for every block group are stored in each block group. Normally, the operating system only uses the descriptors stored in block group 0 for all block groups, but if a filesystem is damaged or has been uncleanly unmounted it's possible to verify the filesystem's integrity and repair damage by using other copies.



As with the superblock, the operating system normally keeps all of the copies of a given group's group descriptors in sync.

Directories are also described by inodes. Each inode has a "type" that identifies it a a file, a directory, or some other special type of thing.

The inodes are numbered sequentially, and files can be identified by their inode number as well as their name.

The "data bitmap" is a set of ones and zeroes, each corresponding to one of the blocks in the block group's data section. If a one is set in the bitmap, that means that this block is used. A zero means that it's free. The data bitmap lets us know which blocks we can use.

Similarly, the "inode bitmap" tells us which entries in the inode table are free.

The Journal:

Although ext2 and ext3 are very similar, ext3 has one important feature that ext2 lacks: journaling. We say that ext3 is a "journaled" filesystem because, instead of writing data directly into data blocks, the filesystem drivers first write a list of tasks into a journal. These tasks describe any changes that need to be made to the data blocks.

The operating system then periodically looks at the journal to see if there are any tasks that need doing. These tasks are then done, in order, and each completed task is marked as "done" in the journal.

If the computer crashes, the journal is examined at the next reboot to see if there were any outstanding tasks that needed to be done. If so, they're done. Any garbled information left at the end of the journal is ignored and cleared.

Journaling makes it much quicker to check the integrity of a filesytem after a crash, since only a few items in the journal need to be looked at. In contrast, when an ext2 filesystem crashes, the operating system needs to scan the entire filesystem looking for problems.

Other than journaling, ext2 and ext3 are largely the forward- and backward-compatible. An ext2 filesystem can easily be converted to ext3 by adding a journal. Going the other way may be possible, too, if an ext3 filesystem doesn't use any features that aren't present in an ext2 filesystem.

The journal is described by a special inode, usually inode number 8.

Filesystem Limits:

Some size limits for filesystems:

Size Limits	ext2	ext3	ext4 (future)
Max. File Size:	2 TB	2 TB	16 TB
Max. Filesystem Size:	16 TB	16 TB	1 EB

Part 5: Filesystem Tools:

Now lets look at some tools for creating and manipulating filesystems.

Making a Filesystem:

To make an ext2 or ext3 filesystem, use the “mke2fs” command. There shouldn't be any reason to create an ext2 filesystem these days, so from here on out I'll assume that we're working with ext3 filesystems.

WARNING!

```
[root@demo ~]# mke2fs -j -Lmydata /dev/sdb1
```

Make an ext2 filesystem... but add a journal, making it ext3. Give it this label. Create it on this partition.

Note that the command above will format (or re-format) the designated partition without asking for any confirmation. Please make sure you point it at the partition you really want to format.

The filesystem label can be any text you choose, but usually the label is chosen to be the same as the name of the location at which you expect to mount the filesystem. For example, a filesystem intended to be mounted at “/boot”, would probably probably be created with “-L/boot”. For the “/” and “/boot” filesystems, this should always be done, but it's good practice for other filesystems, too.

Another dangerous command.

Example mke2fs Output:

```
[root@demo ~]# mke2fs -j -Lmydata /dev/sdb1
mke2fs 1.38 (30-Jun-2005)
Filesystem label=mydata
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
122109952 inodes, 244190000 blocks
12209500 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=247463936
7453 block groups
32768 blocks per group, 32768 fragments per group
16384 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912,
        819200, 884736, 1605632, 2654208,
        4096000, 7962624, 11239424, 20480000,
        23887872, 71663616, 78675968,
        102400000, 214990848
Writing inode tables: done
creating root dir
```


Note that mke2fs divides the disk up into 7,453 block groups, but only (!) 18 copies of the superblock are created. If this were an ext2 filesystem, there would be 7,453 copies. The total number of inodes available (including all inodes in all block groups) is 122,109,952. This is the maximum number of files that this filesystem will hold.

Changing the Attributes of a Filesystem:

The tune2fs command can be used to change the attributes of an ext2/ext3 filesystem after it has been created. For example, to change the filesystem's label:

```
[root@demo ~]# tune2fs -L/data /dev/sdb1
```

Some other useful things that tune2fs can do:

-l List superblock information.

-j Add a journal to an ext2 filesystem, making it ext3.

-c Set the maximum mount count for the filesystem, after which a filesystem check will occur (0 = never check).

-i Set the interval between filesystem checks (0 = never check).

Changing a filesystem's label is perfectly safe. It won't cause you to lose any data. (But it might cause confusion if you're already referring to the old label somewhere.) The same is true for the other flags listed above.

Looking at Filesystem Metadata:

"tune2fs -l" will show you a filesystem's superblock information:

```
[root@demo ~]# tune2fs -l /dev/sda1
tune2fs 1.39 (29-May-2006)
Filesystem volume name:   /boot
Filesystem state:         clean
Inode count:              26104
Block count:              104388
Reserved block count:     5219
Free blocks:              55562
Free inodes:              26037
First block:              1
Block size:               1024
Blocks per group:         8192
Inodes per group:         2008
Inode blocks per group:   251
Filesystem created:       Mon Sep 10 10:58:16 2007
Last mount time:          Fri Dec 26 10:23:03 2008
Last write time:          Fri Dec 26 10:23:03 2008
Mount count:              60
Maximum mount count:      -1
Last checked:             Mon Sep 10 10:58:16 2007
Check interval:           0 (<none>)
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:               128
Journal inode:            8
etc...
```


You can see this plus block group information by using the "dumpe2fs" command.

Note the "mount count", "maximum mount count", "last checked" and "check interval" entries. We'll see later that the "fsck" command uses these.

Checking a Filesystem:

```
[root@demo ~]# fsck /dev/sdb1
```

If a computer loses power unexpectedly, the filesystems on its disks may be left in an untidy state. The "filesystem check" (fsck) command looks at ext2/ext3 filesystems and tries to find and repair damage. Fsck can only be run on unmounted filesystems.

Each filesystem's superblock contains a flag saying whether the filesystem was cleanly unmounted. If it was, fsck just exits without doing anything further.

If the filesystem wasn't cleanly unmounted, fsck checks it. Under ext3, fsck first just looks at the journal and completes any outstanding operations, if possible. If this works, then fsck exits.

If the ext3 journal is damaged, or if this is an ext2 filesystem, fsck scans the filesystem for damage. It does this primarily by looking for inconsistencies between the various copies of the superblock and block group descriptors. If inconsistencies are found, fsck tries to resolve them, using various strategies.

The filesystem's superblock also contains a "mount count", "maximum mount count", "last check date" and "check interval". If the mount count exceeds the maximum, a scan of the filesystem is forced even if it was cleanly unmounted. If the time since the last check date exceeds the check interval, a scan is also forced. Both of these forced checks can be disabled, by using tune2fs.

Modifying fsck's Behavior:

Some useful fsck options:

-f Force a scan, even if the filesystem appears to have been cleanly unmounted.

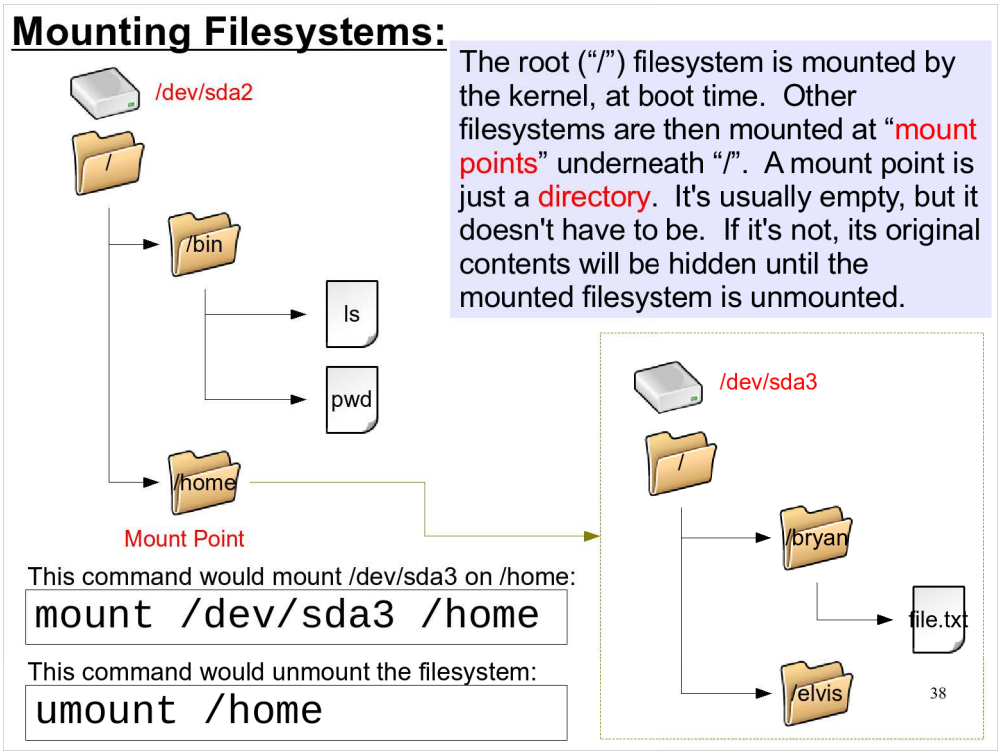
-b Specify an alternative superblock, in case the primary superblock has been damaged.

-y Answer "yes" to any questions fsck asks.

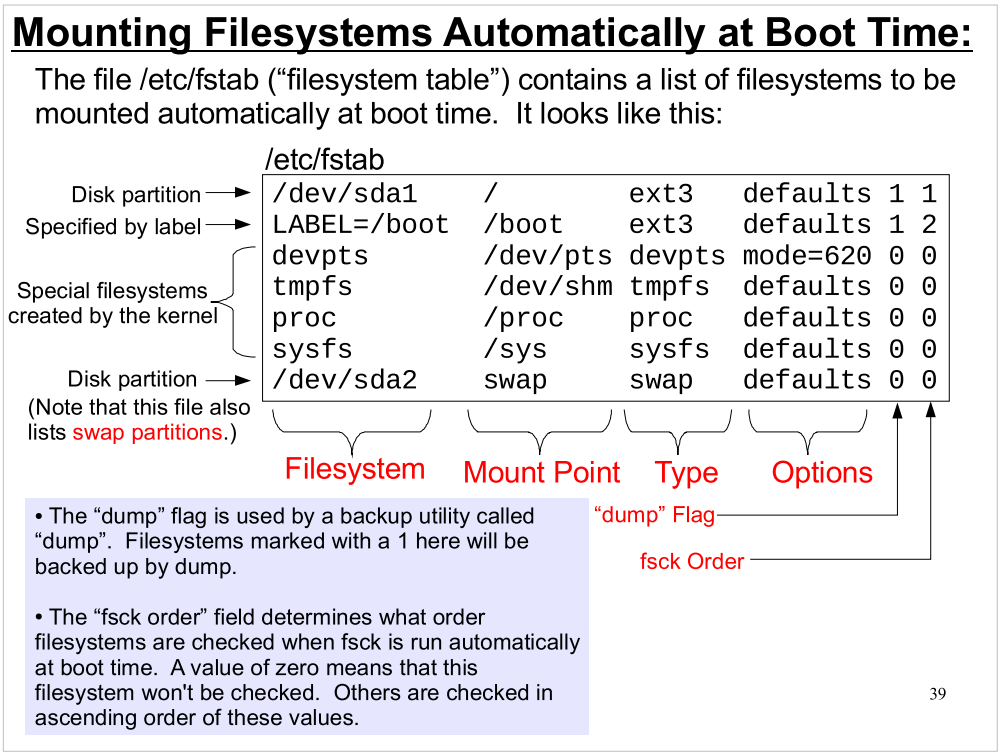
-A Check all filesystems.

-C Show a progress bar as fsck works. (It can sometimes take a very long time.)

Fsck is actually a wrapper that calls a different type-specific filesystem checker for each different type of filesystem that it knows about.



The directory tree of each physical device is grafted onto the same tree, with the root directory ("/") at the top. There are no "C:" or "D:" drives under Linux. Every file you have access to lives in the same tree, and you don't need to care what device the file lives on.



Among the “options” settings you can use “noauto” to cause the given filesystem not to be automatically mounted at boot time. In that case, you'd need to manually mount it later, using the “mount” command.

If you have a filesystem listed in /etc/fstab, you can mount it either like this, with two arguments:

```
mount /dev/sda1 /
```

or like this, with one argument:

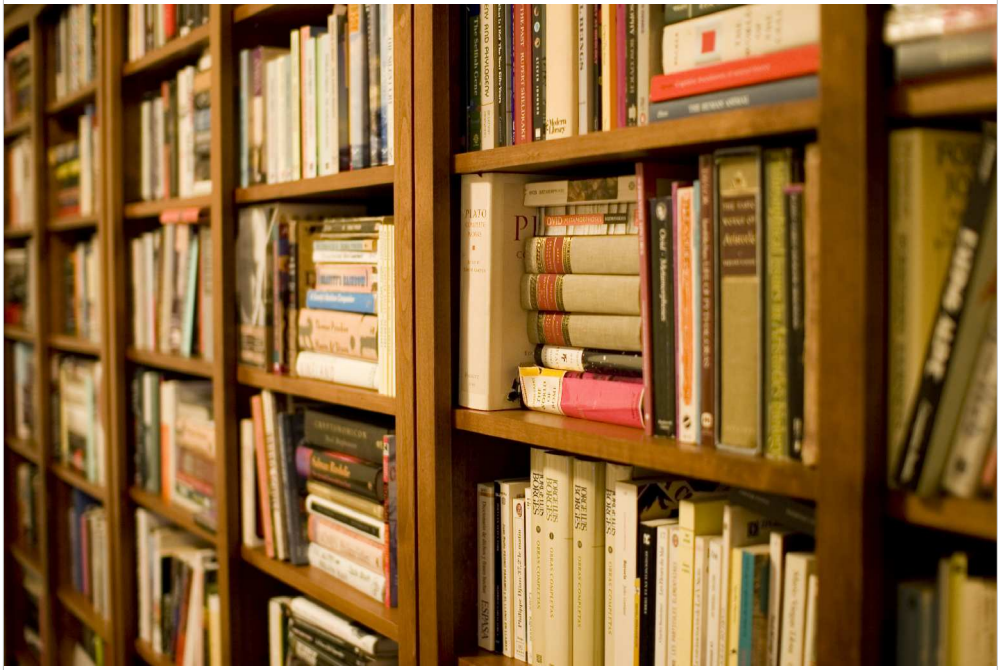
```
mount /dev/sda1
```

or

```
mount /
```

since /etc/fstab lets “mount” know what you mean by these.

Part 6: Logical Volume Management



In past years I wouldn't have included a section on LVM, but these days Linux distributions use it by default. You'll need to know a little about LVM to understand how any current Linux computer's filesystems are laid out.

The LVM System:

The ext2 and ext3 filesystems are limited by the size of the partitions they occupy. Partitions are difficult to resize, and they can't grow beyond the whole size of the disk. What can we do if we need more space than that for our filesystem?

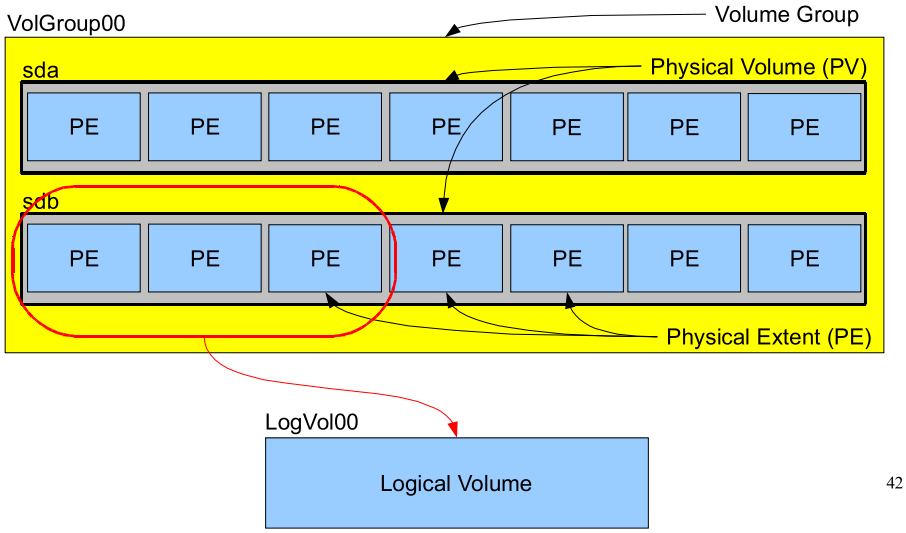
One solution is the Logical Volume Management (LVM) system. LVM lets you define “logical volumes” that can be used like disk partitions. Unlike partitions, logical volumes can span multiple disks, and they can easily grow or shrink.

These days, when you install a Linux distribution on a computer, some of the filesystems that are created will (by default) be on logical volumes, not physical disk partitions. This makes it important to understand how LVM works.

As we'll see, LVM also provides us with another, software-based, way to avoid the 2 TB partition limit imposed by MBR-style partition tables.

Logical Volume Structure:

LVM divides each disk into chunks called “physical extents” (PEs). Disks are added to “volume groups” (VGs). Each VG is a pool of physical extents from which “logical volumes” (LVs) can be formed. An LV can be expanded by adding more PEs from the pool. If an LV needs to grow even larger, more PEs can be added to the pool by adding disks to the volume group.



Creating Logical Volumes:

First, let's make a new disk available to the LVM system by initializing it as an LVM “physical volume” using “pvcreate”:

```
[root@demo ~]# pvcreate /dev/sdb
```

Then, let's create a new volume group and add the newly-initialized disk to it:

```
[root@demo ~]# vgcreate VolGroup01 /dev/sdb
```

Now, let's create a 500 GB logical volume from the pool of space in our new volume group:

```
[root@demo ~]# lvcreate -L500G -nLogVol00 VolGroup01
```

Now we can create a filesystem on the logical volume, just as we'd use a partition:

```
[root@demo ~]# mke2fs -j -L/data /dev/VolGroup01/LogVol00
```

Finally, we can mount the logical volume just as we'd mount a partition:

```
[root@demo ~]# mount /dev/VolGroup01/LogVol00 /data
```

Note that you can point pvcreate at either a whole disk, as above, or a disk partition (like “/dev/sdb1”). If you use a whole disk, the disk's partition table is wiped out, since LVM doesn't need it. Thus, LVM can be used to completely avoid the 2 TB limit imposed by MBR-style partition tables.

This may be one of the reasons current distributions are using LVM by default. Disks are rapidly approaching 2 TB in size, and it looks like most non-Mac computers are going to be using the BIOS/MBR architecture for a while, rather than moving to EFI/GPT. LVM provides a way to support large disks without any hardware changes.

Examining Volume Groups:

You can find out about a volume group by using the “vgdisplay” command:

```
[root@demo ~]# vgdisplay VolGroup00
  --- Volume group ---
  VG Name               VolGroup00
  System ID
  Format                lvm2
  Metadata Areas        1
  Metadata Sequence No  3
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                2
  Open LV               2
  Max PV                0
  Cur PV                1
  Act PV                1
  VG Size               148.91 GB
  PE Size               32.00 MB
  Total PE              4765
  Alloc PE / Size       4765 / 148.91 GB
  Free  PE / Size       0 / 0
  VG UUID               blHfoy-z03Z-DzTQ-PH4p-uYfJ-jkHS-29Hxob
```

Notice these. They tell you how many physical extents are in the volume group, and how many are still available for making new logical volumes.

If you move a disk to a different computer that already has a volume group with the same name, you may need to use the UUID of the volume groups to rename one of them. Use “vgrename” for this.

Growing a Logical Volume:

If we don't have any free PEs in our volume group, we can add another disk:

```
[root@demo ~]# vgextend VolGroup01 /dev/sdc
```

Now that we have more PEs, we can assign some of them to one of our existing logical volumes, to make it bigger:

```
[root@demo ~]# lvextend -L+100G /dev/VolGroup01/LogVol00
```

Extending the logical volume doesn't extend the filesystem on top of it. We have to do that by hand. For ext2/ext3 filesystems, you can use the resize2fs command to do this. The command below will just resize the filesystem so that it occupies all of the available space in the logical volume:

```
[root@demo ~]# resize2fs /dev/VolGroup01/LogVol00
```

For many more “stupid LVM tricks” see: http://www.howtoforge.com/linux_lvm

Part 7: Managing File Ownerships and Permissions:

46

Now lets take a quick look at how some of the metadata stored in a file's inode is used. In particular, we'll look at file ownerships, and three mechanisms for controlling access to files.

The “chown” and “chgrp” Commands:

A file's user ownership and group ownership can be changed with “chown” (change ownership) command:

```
[root@demo ~]# ls -l junk.dat
-rw-r--r-- 1 bkw1a bkw1a 0 Jan 25 00:07 junk.dat

[root@demo ~]# chown elvis junk.dat
[root@demo ~]# ls -l junk.dat
-rw-r--r-- 1 elvis bkw1a 0 Jan 25 00:07 junk.dat

[root@demo ~]# chown elvis.demo junk.dat
[root@demo ~]# ls -l junk.dat
-rw-r--r-- 1 elvis demo 0 Jan 25 00:07 junk.dat
```

Group ownership can also be changed with the “chgrp” command:

```
[root@demo ~]# chgrp demo junk.dat
```

47

Under Linux, each files or directory has both a “user” ownership and a “group” ownership. A file's ownership and permissions (which we'll look at next) control who has access to the file. Note that, in the Unix world, directories are just another kind of file, and have the same kind of ownerhips and permissions.

Ownership of all files in an entire directory tree can be changed by using the “-R” (for “recursive”) flag on chown and chgrp. For example:

chown -R elvis.demo phase1

where “phase1” is a directory.

The “stat” Command:

The set of permissions pertaining to a file is called the file's “mode”. The mode is displayed symbolically by commands like “ls”:

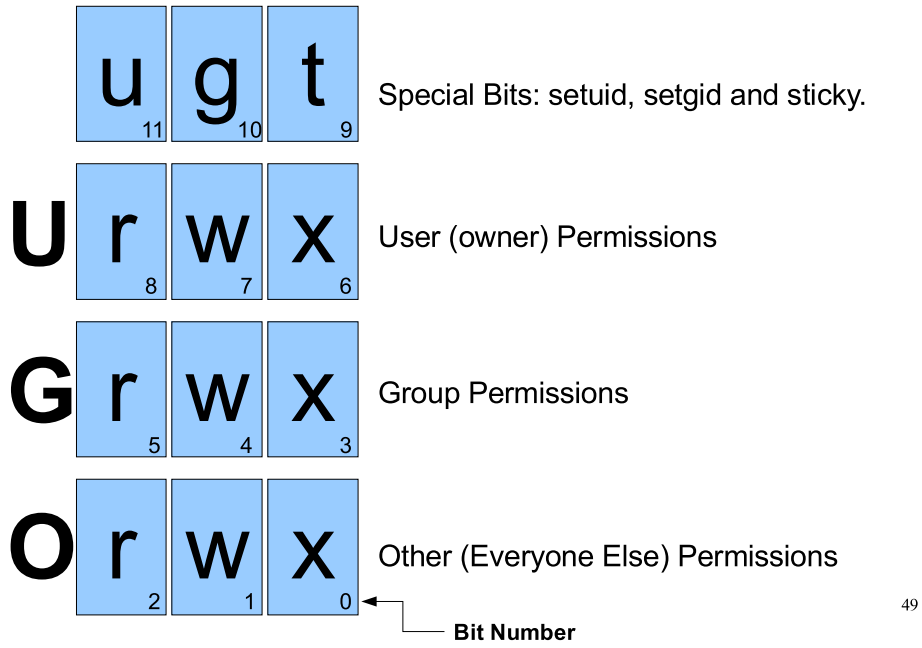
```
-rw-r-----  1 bkw1a demo     72 Jan 18 10:52 readme.txt
```

mode

Internally, though, the file's mode is represented by four sets of three bits (12 bits in all), which can collectively be written as a four-digit octal number. The “stat” command shows the mode in both formats:

```
~/demo> stat readme.txt
  File: `readme.txt'
  Size: 72              Blocks: 8          IO Block: 4096
regular file
Device: fd00h/64768d    Inode: 17008595    Links: 1
Access: (0640/-rw-r-----)  Uid: (500/bkw1a)   Gid: (505/demo)
Access: 2009-01-19 10:58:02.000000000 -0500
Modify: 2009-01-18 10:52:29.000000000 -0500
Change: 2009-01-18 11:38:30.000000000 -0500
```

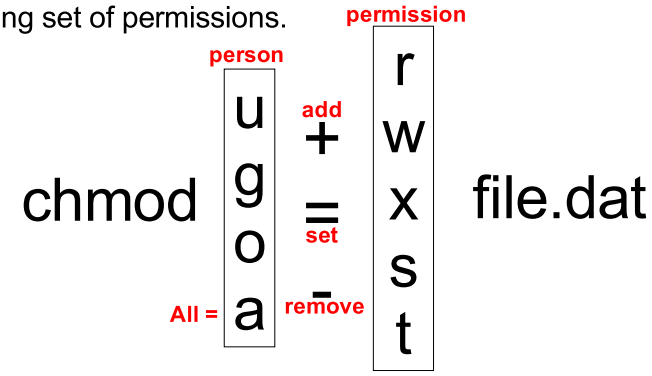
Internal Representation of File Mode Bits:



File permissions are actually stored as a set of 12 bits, shown above.

The “chmod” Command:

Permissions on files can be changed with the “chmod” (“change mode”) command. Permissions can either be specified symbolically or as an octal number. The symbolic form is most useful when modifying an existing set of permissions.



Give all users read permission:

```
~/demo> chmod a+r readme.dat
```

Give user and group read permission:

```
~/demo> chmod ug+r readme.dat
```

Alternatively, modes can be set directly as octal numbers:

Set the file's mode to rw-r-r-:

```
~/demo> chmod 0644 readme.dat
```


You can use “chmod g+s directory” to set the “setgid” bit on a directory. We'll see later that you can use “chmod u+s” to turn on a “setuid” bit.

Permissions on Directories:

- If you have write permission on a directory, you can delete any file within the directory, regardless of whether you have ownership or write permission on the particular file.

- You need execute permission on a directory in order to traverse it. For example, to "cd" into a directory, you need execute permission.

- You need read permission on a directory in order to list its contents, even if all of the individual files within the directory are readable by you.

The meaning of permissions on files if fairly clear, but the meaning of read, write and execute on a directory may need some explanation.

The “setgid” Bit:

It is possible to set the permissions and ownership on a directory so that files created within the directory will inherit the group ownership of the directory. This is accomplished by setting the “setgid” bit in the directory's permissions:

```
drwxrwsr-x   2 bkw1a demo   4096 Jan 27 13:12 shared
```

Setgid bit

Files subsequently created in the “shared” directory will have their group ownership set to “demo”, making it easier to share them with other members of this group.

52

If the setgid bit is set on a parent directory, child directories inherit the bit.

It's important to note, though, that setting this bit on a parent directory doesn't change the ownership or permissions of files or subdirectories that already existed before you set the setgid bit on the parent directory. It only affects files and directories created thereafter.

The “setuid” bit, which we'll see next, has no effect on directories.

The "setuid" Bit: (CAUTION!)

It is possible to set the permissions and ownership on an executable file so that the program will always automatically run as though it had been invoked by a specified user. This is most often used to allow unprivileged users to do something that only the root user could normally do. Changing a password, for example:

```
~/demo> ls -al /usr/bin/passwd
-rwsr-xr-x 1 root root 22984 Jan  6  2007 /usr/bin/passwd
```

Setuid bit

When the setuid bit is set, the file will run as though it had been invoked by the file's owner (root, in this case).

```
[root@ayesha ~]# chmod u+s /usr/bin/passwd
```

Setting the setuid bit.

When invoked by a non-root user, passwd runs as root and allows the user to change his or her own password, but no others. But if it's running as though it was invoked as root, how does passwd know that it was really invoked by a non-root user? The answer is that each process actually maintains two UID records, called an "effective" UID and a "real" UID. Because of this, passwd can look at the real UID to decide how it wants to behave.

It's important to note that it's up to setuid programs to decide on their own what they will and won't do. Mistakes often lead to security problems.

What happens if the setuid bit is set, but the file isn't executable? In that case, the output of "ls" would look like:
"-rwSr-xr-x" (with a capitol S), indicating that the file would execute as root if it were exectuable.

What effect does the "setgid" bit have on files? it causes the file to run as though it had been invoked by a member of the owning group.

The “Sticky Bit”:

One of the bits in a file's mode is called the “sticky” bit. If this bit is set on a directory, only a file's owner (or root) is allowed to delete or rename files in this directory, no matter what would otherwise be allowed. The sticky bit is most often used on temporary directories, like /tmp, where everyone needs to have write access, but it's desirable to prevent users from deleting one another's files.

```
drwxrwxrwt  34 root root 36864 Jan 27 15:49 tmp
```

The sticky bit shows up in the symbolic representation of the permissions as a “t” in the last position if the “x” bit is set for “others”, and as a “T” in this position otherwise.

Attributes, and Immutable Files:

In addition to the file permissions available on all Unix filesystems, the common filesystems under Linux also support a set of extended file attributes. Some of these are quite esoteric, but one, at least, is widely useful. This is the "immutable" attribute.

Files marked as immutable cannot be changed or deleted, even by the root user (although the root user has the power to remove the immutable attribute). This is useful for preventing accidental or malicious modification of files that are normally unchanging.

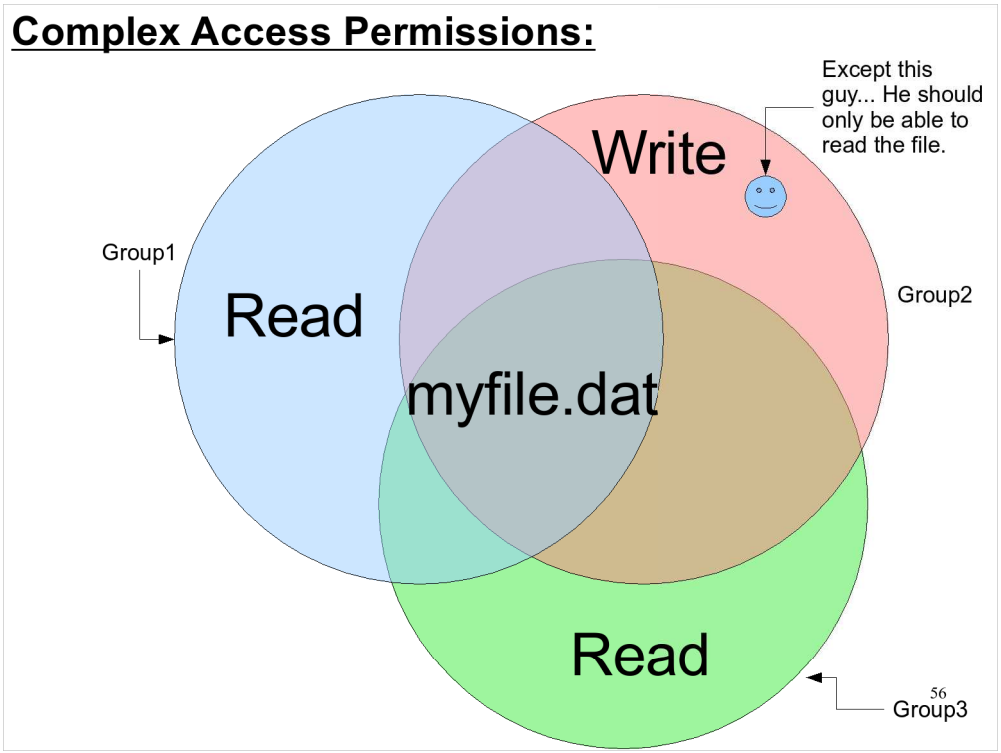
Attributes can be listed with "lsattr" and changed with "chattr":

```
[root@demo ~]# lsattr junk.dat
------------- junk.dat

[root@demo ~]# chattr +i junk.dat

[root@demo ~]# lsattr junk.dat
----i-------- junk.dat
```

The attribute can be removed with the "-i" flag.



But what if we want to have a really complex system of access permissions for a file? We can't do this with just user, group and other.

Access Control Lists (ACLs):

In addition to the read/write/execute permissions for user/group/other, the most common Linux filesystems also offer a mechanism to deal with more complex access restrictions. This mechanism is called Access Control Lists (ACLs).

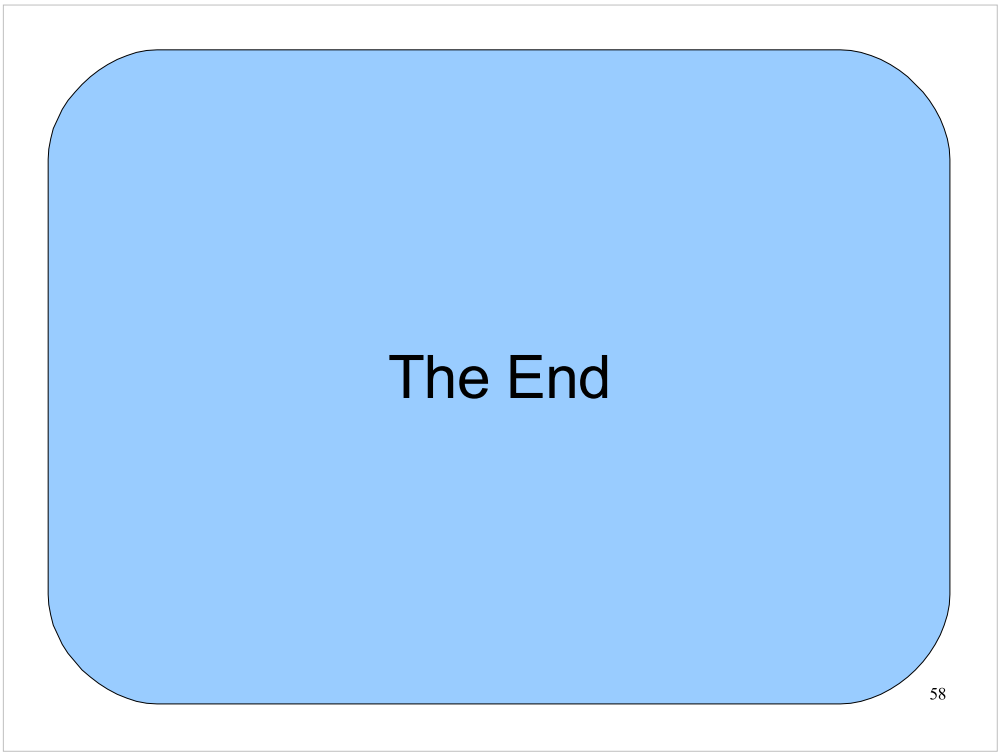
When ACLs are available, each file or directory can have a complex set of access permissions associated with it. These permissions consist of an arbitrarily long list of access control rules. A rule can be created, for example, to give a particular user read-only access to a file, or to allow read-write access to a particular group.

ACLs can be modified with the "setfacl" command, and viewed with the "getfacl" command.

```
[root@demo ~]# getfacl myfile.dat
# file: myfile.dat
# owner: elvis
# group: demo
user::rw-
group::r--
other::---

[root@demo ~]# setfacl -m user:priscilla:rw myfile.dat

[root@demo ~]# getfacl myfile.dat
# file: myfile.dat
# owner: elvis
# group: demo
user::rw-
user:priscilla:rw
group::r--
other::---
```



Thank You!