15

STORAGE MEDIA

In previous chapters we've looked at manipulating data at the file level. In this chapter, we will consider data at the device level. Linux has amazing capabilities for handling storage devices, whether physical storage such as hard disks, network storage, or virtual storage devices like RAID (redundant array of independent disks) and LVM (logical volume manager).

However, since this is not a book about system administration, we will not try to cover this entire topic in depth. What we will do is introduce some of the concepts and key commands that are used to manage storage devices.

To carry out the exercises in this chapter, we will use a USB flash drive, a CD-RW disc (for systems equipped with a CD-ROM burner), and a floppy disk (again, if the system is so equipped).

We will look at the following commands:

- `mount`—Mount a filesystem.
- `umount`—Unmount a filesystem.

- `fdisk`—Partition table manipulator.
- `fsck`—Check and repair a filesystem.
- `fdformat`—Format a floppy disk.
- `mkfs`—Create a filesystem.
- `dd`—Write block-oriented data directly to a device.
- `genisoimage (mkisofs)`—Create an ISO 9660 image file.
- `wodim (cdrecord)`—Write data to optical storage media.
- `md5sum`—Calculate an MD5 checksum.

Mounting and Unmounting Storage Devices

Recent advances in the Linux desktop have made storage device management extremely easy for desktop users. For the most part, we attach a device to our system and it just works. Back in the old days (say, 2004), this stuff had to be done manually. On non-desktop systems (i.e., servers) this is still a largely manual procedure, because servers often have extreme storage needs and complex configuration requirements.

The first step in managing a storage device is attaching the device to the filesystem tree. This process, called *mounting*, allows the device to participate with the operating system. As we recall from Chapter 2, Unix-like operating systems, like Linux, maintain a single filesystem tree with devices attached at various points. This contrasts with other operating systems such as MS-DOS and Windows that maintain separate trees for each device (for example *C:*, *D:*, etc.).

A file named */etc/fstab* lists the devices (typically hard disk partitions) that are to be mounted at boot time. Here is an example */etc/fstab* file from a Fedora 7 system:

```
LABEL=/12          /                       ext3    defaults        1 1
LABEL=/home        /home                   ext3    defaults        1 2
LABEL=/boot        /boot                   ext3    defaults        1 2
tmpfs              /dev/shm                tmpfs   defaults        0 0
devpts             /dev/pts                devpts  gid=5,mode=620  0 0
sysfs              /sys                    sysfs   defaults        0 0
proc               /proc                   proc    defaults        0 0
LABEL=SWAP-sda3    swap                    swap    defaults        0 0
```

Most of the filesystems listed in this example file are virtual and are not applicable to our discussion. For our purposes, the interesting ones are the first three:

```
LABEL=/12          /                       ext3    defaults        1 1
LABEL=/home        /home                   ext3    defaults        1 2
LABEL=/boot        /boot                   ext3    defaults        1 2
```

These are the hard disk partitions. Each line of the file consists of six fields, as shown in Table 15-1.

Table 15-1: *`/etc/fstab`* Fields

Field	Contents	Description
1	Device	Traditionally, this field contains the actual name of a device file associated with the physical device, such as */dev/hda1* (the first partition of the master device on the first IDE channel). But with today's computers, which have many devices that are hot pluggable (like USB drives), many modern Linux distributions associate a device with a text label instead. This label (which is added to the storage medium when it is formatted) is read by the operating system when the device is attached to the system. That way, no matter which device file is assigned to the actual physical device, it can still be correctly identified.
2	Mount point	The directory where the device is attached to the filesystem tree
3	Filesystem type	Linux allows many filesystem types to be mounted. Most native Linux filesystems are ext3, but many others are supported, such as FAT16 (`msdos`), FAT32 (`vfat`), NTFS (`ntfs`), CD-ROM (`iso9660`), etc.
4	Options	Filesystems can be mounted with various options. It is possible, for example, to mount filesystems as read only or to prevent any programs from being executed from them (a useful security feature for removable media).
5	Frequency	A single number that specifies if and when a filesystem is to be backed up with the `dump` command
6	Order	A single number that specifies in what order filesystems should be checked with the `fsck` command

Viewing a List of Mounted Filesystems

The `mount` command is used to mount filesystems. Entering the command without arguments will display a list of the filesystems currently mounted:

```
[me@linuxbox ~]$ mount
/dev/sda2 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
```

```
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda5 on /home type ext3 (rw)
/dev/sda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
fusectl on /sys/fs/fuse/connections type fusectl (rw)
/dev/sdd1 on /media/disk type vfat (rw,nosuid,nodev,noatime,
uhelper=hal,uid=500,utf8,shortname=lower)
twin4:/musicbox on /misc/musicbox type nfs4 (rw,addr=192.168.1.4)
```

The format of the listing is *device* on *mount_point* type *filesystem_type* (*options*). For example, the first line shows that device */dev/sda2* is mounted as the root filesystem, is of type ext3, and is both readable and writable (the option rw). This listing also has two interesting entries at the bottom. The next-to-last entry shows a 2-gigabyte SD memory card in a card reader mounted at */media/disk*, and the last entry is a network drive mounted at */misc/musicbox*.

For our first experiment, we will work with a CD-ROM. First, let's look at a system before a CD-ROM is inserted:

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
```

This listing is from a CentOS 5 system that is using LVM to create its root filesystem. Like many modern Linux distributions, this system will attempt to automatically mount the CD-ROM after insertion. After we insert the disc, we see the following:

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
/dev/hdc on /media/live-1.0.10-8 type iso9660 (ro,noexec,nosuid,nodev,uid=500)
```

We see the same listing as before, with one additional entry. At the end of the listing, we see that the CD-ROM (which is device */dev/hdc* on this system) has been mounted on */media/live-1.0.10-8* and is type iso9660 (a CD-ROM). For the purposes of our experiment, we're interested in the name of the device. When you conduct this experiment yourself, the device name will most likely be different.

Warning: *In the examples that follow, it is vitally important that you pay close attention to the actual device names in use on your system and* ***do not use the names used in this text!***

Also, note that audio CDs are not the same as CD-ROMs. Audio CDs do not contain filesystems and thus cannot be mounted in the usual sense.

Now that we have the device name of the CD-ROM drive, let's unmount the disc and remount it at another location in the filesystem tree. To do this, we become the superuser (using the command appropriate for our system) and unmount the disc with the `umount` (notice the spelling) command:

```
[me@linuxbox ~]$ su -
Password:
[root@linuxbox ~]# umount /dev/hdc
```

The next step is to create a new mount point for the disc. A *mount point* is simply a directory somewhere on the filesystem tree. Nothing special about it. It doesn't even have to be an empty directory, though if you mount a device on a non-empty directory, you will not be able to see the directory's previous contents until you unmount the device. For our purposes, we will create a new directory:

```
[root@linuxbox ~]# mkdir /mnt/cdrom
```

Finally, we mount the CD-ROM at the new mount point. The `-t` option is used to specify the filesystem type:

```
[root@linuxbox ~]# mount -t iso9660 /dev/hdc /mnt/cdrom
```

Afterward, we can examine the contents of the CD-ROM via the new mount point:

```
[root@linuxbox ~]# cd /mnt/cdrom
[root@linuxbox cdrom]# ls
```

Notice what happens when we try to unmount the CD-ROM:

```
[root@linuxbox cdrom]# umount /dev/hdc
umount: /mnt/cdrom: device is busy
```

Why is this? We cannot unmount a device if the device is being used by someone or some process. In this case, we changed our working directory to the mount point for the CD-ROM, which causes the device to be busy. We can easily remedy the issue by changing the working directory to something other than the mount point:

```
[root@linuxbox cdrom]# cd
[root@linuxbox ~]# umount /dev/hdc
```

Now the device unmounts successfully.

WHY UNMOUNTING IS IMPORTANT

If you look at the output of the `free` command, which displays statistics about memory usage, you will see a statistic called *buffers*. Computer systems are designed to go as fast as possible. One of the impediments to system speed is slow devices. Printers are a good example. Even the fastest printer is extremely slow by computer standards. A computer would be very slow indeed if it had to stop and wait for a printer to finish printing a page. In the early days of PCs (before multitasking), this was a real problem. If you were working on a spreadsheet or text document, the computer would stop and become unavailable every time you printed. The computer would send the data to the printer as fast as the printer could accept it, but it was very slow because printers don't print very fast. This problem was solved by the advent of the *printer buffer*, a device containing some RAM memory, that would sit between the computer and the printer. With the printer buffer in place, the computer would send the printer output to the buffer, and it would quickly be stored in the fast RAM so the computer could go back to work without waiting. Meanwhile, the printer buffer would slowly *spool* the data to the printer from the buffer's memory at the speed at which the printer could accept it.

This idea of buffering is used extensively in computers to make them faster. Don't let the need to occasionally read or write data to or from slow devices impede the speed of the system. Operating systems store data that has been read from, and is to be written to, storage devices in memory for as long as possible before actually having to interact with the slower device. On a Linux system, for example, you will notice that the system seems to fill up memory the longer it is used. This does not mean Linux is "using" all the memory, it means that Linux is taking advantage of all the available memory to do as much buffering as it can.

This buffering allows writing to storage devices to be done very quickly, because the writing to the physical device is being deferred to a future time. In the meantime, the data destined for the device is piling up in memory. From time to time, the operating system will write this data to the physical device.

Unmounting a device entails writing all the remaining data to the device so that it can be safely removed. If the device is removed without first being unmounted, the possibility exists that not all the data destined for the device has been transferred. In some cases, this data may include vital directory updates, which will lead to *filesystem corruption*, one of the worst things that can happen on a computer.

Determining Device Names

It's sometimes difficult to determine the ameof a device. Back in the old days, it wasn't very hard. A device was always in the same place and didn't change. Unix-like systems like it that way. Back when Unix was developed, "changing a disk drive" involved using a forklift to remove a washing

machine–sized device from the computer room. In recent years, the typical desktop hardware configuration has become quite dynamic, and Linux has evolved to become more flexible than its ancestors.

In the examples above, we took advantage of the modern Linux desktop's ability to "automagically" mount the device and then determine the name after the fact. But what if we are managing a server or some other environment where this does not occur? How can we figure it out?

First, let's look at how the system names devices. If we list the contents of the */dev* directory (where all devices live), we can see that there are lots and lots of devices:

```
[me@linuxbox ~]$ ls /dev
```

The contents of this listing reveal some patterns of device naming. Table 15-2 lists a few.

Table 15-2: Linux Storage Device Names

Pattern	Device
*/dev/fd**	Floppy disk drives
*/dev/hd**	IDE (PATA) disks on older systems. Typical motherboards contain two IDE connectors, or *channels*, each with a cable with two attachment points for drives. The first drive on the cable is called the *master* device and the second is called the *slave* device. The device names are ordered such that */dev/hda* refers to the master device on the first channel, */dev/hdb* is the slave device on the first channel; */dev/hdc*, the master device on the second channel, and so on. A trailing digit indicates the partition number on the device. For example, */dev/hda1* refers to the first partition on the first hard drive on the system while */dev/hda* refers to the entire drive.
*/dev/lp**	Printers
*/dev/sd**	SCSI disks. On recent Linux systems, the kernel treats all disk-like devices (including PATA/SATA hard disks, flash drives, and USB mass storage devices such as portable music players and digital cameras) as SCSI disks. The rest of the naming system is similar to the older */dev/hd** naming scheme described above.
/dev/sr*	Optical drives (CD/DVD readers and burners)

In addition, we often see symbolic links such as */dev/cdrom*, */dev/dvd*, and */dev/floppy*, which point to the actual device files, provided as a convenience.

If you are working on a system that does not automatically mount removable devices, you can use the following technique to determine how

the removable device is named when it is attached. First, start a real-time view of the */var/log/messages* file (you may require superuser privileges for this):

```
[me@linuxbox ~]$ sudo tail -f /var/log/messages
```

The last few lines of the file will be displayed and then pause. Next, plug in the removable device. In this example, we will use a 16MB flash drive. Almost immediately, the kernel will notice the device and probe it:

```
Jul 23 10:07:53 linuxbox kernel: usb 3-2: new full speed USB device using uhci_h
cd and address 2
Jul 23 10:07:53 linuxbox kernel: usb 3-2: configuration #1 chosen from 1 choice
Jul 23 10:07:53 linuxbox kernel: scsi3 : SCSI emulation for USB Mass Storage dev
ices
Jul 23 10:07:58 linuxbox kernel: scsi scan: INQUIRY result too short (5), using
36
Jul 23 10:07:58 linuxbox kernel: scsi 3:0:0:0: Direct-Access Easy Disk 1.00 PQ:
0 ANSI: 2
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte hardware secto
rs (16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect is off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive cache: write t
hrough
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte hardware secto
rs (16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect is off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive cache: write t
hrough
Jul 23 10:07:59 linuxbox kernel:  sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI removable disk
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: Attached scsi generic sg3 type 0
```

After the display pauses again, press CTRL-C to get the prompt back. The interesting parts of the output are the repeated references to `[sdb]`, which matches our expectation of a SCSI disk device name. Knowing this, two lines become particularly illuminating:

```
Jul 23 10:07:59 linuxbox kernel:  sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI removable disk
```

This tells us the device name is */dev/sdb* for the entire device and */dev/sdb1* for the first partition on the device. As we have seen, working with Linux means lots of interesting detective work!

Note: *Using the `tail -f /var/log/messages` technique is a great way to watch what the system is doing in near realtime.*

With our device name in hand, we can now mount the flash drive:

```
[me@linuxbox ~]$ sudo mkdir /mnt/flash
[me@linuxbox ~]$ sudo mount /dev/sdb1 /mnt/flash
[me@linuxbox ~]$ df
```

```
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2             15115452   5186944   9775164  35% /
/dev/sda5             59631908  31777376  24776480  57% /home
/dev/sda1               147764     17277    122858  13% /boot
tmpfs                   776808         0    776808   0% /dev/shm
/dev/sdb1                15560         0     15560   0% /mnt/flash
```

The device name will remain the same as long as it remains physically attached to the computer and the computer is not rebooted.

Creating New Filesystems

Let's say that we want to reformat the flash drive with a Linux native filesystem, rather than the FAT32 system it has now. This involves two steps: first, (optionally) creating a new partition layout if the existing one is not to our liking, and second, creating a new, empty filesystem on the drive.

Warning: *In the following exercise, we are going to format a flash drive. Use a drive that contains nothing you care about because it will be erased! Again,* ***make absolutely sure you are specifying the correct device name for your system, not the one shown in the text. Failure to heed this warning could result in formatting (i.e., erasing) the wrong drive!***

Manipulating Partitions with fdisk

The `fdisk` program allows us to interact directly with disk-like devices (such as hard disk drives and flash drives) at a very low level. With this tool we can edit, delete, and create partitions on the device. To work with our flash drive, we must first unmount it (if needed) and then invoke the `fdisk` program as follows:

```
[me@linuxbox ~]$ sudo umount /dev/sdb1
[me@linuxbox ~]$ sudo fdisk /dev/sdb
```

Notice that we must specify the device in terms of the entire device, not by partition number. After the program starts up, we will see the following prompt:

```
Command (m for help):
```

Entering an `m` will display the program menu:

```
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition
   l   list known partition types
   m   print this menu
   n   add a new partition
```

```
   o   create a new empty DOS partition table
   p   print the partition table
   q   quit without saving changes
   s   create a new empty Sun disklabel
   t   change a partition's system id
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit
   x   extra functionality (experts only)

Command (m for help):
```

The first thing we want to do is examine the existing partition layout. We do this by entering `p` to print the partition table for the device:

```
Command (m for help): p

Disk /dev/sdb: 16 MB, 16006656 bytes
1 heads, 31 sectors/track, 1008 cylinders
Units = cylinders of 31 * 512 = 15872 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               2        1008       15608+   b  W95 FAT32
```

In this example, we see a 16MB device with a single partition (1) that uses 1006 of the available 1008 cylinders on the device. The partition is identified as a Windows 95 FAT32 partition. Some programs will use this identifier to limit the kinds of operation that can be done to the disk, but most of the time changing the identifier is not critical. However, in the interest of demonstration, we will change it to indicate a Linux partition. To do this, we must first find out what ID is used to identify a Linux partition. In the listing above, we see that the ID `b` is used to specify the existing partition. To see a list of the available partition types, we refer back to the program menu. There we can see the following choice:

```
   l   list known partition types
```

If we enter `l` at the prompt, a large list of possible types is displayed. Among them we see `b` for our existing partition type and `83` for Linux.

Going back to the menu, we see this choice to change a partition ID:

```
   t   change a partition's system id
```

We enter `t` at the prompt and enter the new ID:

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): 83
Changed system type of partition 1 to 83 (Linux)
```

This completes all the changes that we need to make. Up to this point, the device has been untouched (all the changes have been stored in memory, not on the physical device), so we will write the modified partition table to the device and exit.

To do this, we enter `w` at the prompt:

```
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
[me@linuxbox ~]$
```

If we had decided to leave the device unaltered, we could have entered `q` at the prompt, which would have exited the program without writing the changes. We can safely ignore the ominous-sounding warning message.

Creating a New Filesystem with mkfs

With our partition editing done (lightweight though it might have been), it's time to create a new filesystem on our flash drive. To do this, we will use `mkfs` (short for *make filesystem*), which can create filesystems in a variety of formats. To create an ext3 filesystem on the device, we use the `-t` option to specify the ext3 system type, followed by the name of the device containing the partition we wish to format:

```
[me@linuxbox ~]$ sudo mkfs -t ext3 /dev/sdb1
mke2fs 1.40.2 (12-Jul-2012)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
3904 inodes, 15608 blocks
780 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=15990784
2 block groups
8192 blocks per group, 8192 fragments per group
1952 inodes per group
Superblock backups stored on blocks:
        8193

Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 34 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
[me@linuxbox ~]$
```

The program will display a lot of information when ext3 is the chosen filesystem type. To reformat the device to its original FAT32 filesystem, specify vfat as the filesystem type:

```
[me@linuxbox ~]$ sudo mkfs -t vfat /dev/sdb1
```

This process of partitioning and formatting can be used anytime additional storage devices are added to the system. While we worked with a tiny flash drive, the same process can be applied to internal hard disks and other removable storage devices like USB hard drives.

Testing and Repairing Filesystems

In our earlier discussion of the */etc/fstab* file, we saw some mysterious digits at the end of each line. Each time the system boots, it routinely checks the integrity of the filesystems before mounting them. This is done by the `fsck` program (short for *filesystem check*). The last number in each *fstab* entry specifies the order in which the devices are to be checked. In our example above, we see that the root filesystem is checked first, followed by the *home* and *boot* filesystems. Devices with a zero as the last digit are not routinely checked.

In addition to checking the integrity of filesystems, `fsck` can also repair corrupt filesystems with varying degrees of success, depending on the amount of damage. On Unix-like filesystems, recovered portions of files are placed in the *lost+found* directory, located in the root of each filesystem.

To check our flash drive (which should be unmounted first), we could do the following:

```
[me@linuxbox ~]$ sudo fsck /dev/sdb1
fsck 1.40.8 (13-Mar-2012)
e2fsck 1.40.8 (13-Mar-2012)
/dev/sdb1: clean, 11/3904 files, 1661/15608 blocks
```

In my experience, filesystem corruption is quite rare unless there is a hardware problem, such as a failing disk drive. On most systems, filesystem corruption detected at boot time will cause the system to stop and direct you to run `fsck` before continuing.

WHAT THE FSCK?

In Unix culture, fsck is often used in place of a popular word with which it shares three letters. This is especially appropriate, given that you will probably be uttering the aforementioned word if you find yourself in a situation where you are forced to run `fsck`.

Formatting Floppy Disks

For those of us still using computers old enough to be equipped with floppy-disk drives, we can manage those devices, too. Preparing a blank floppy for use is a two-step process. First, we perform a low-level format on the disk, and then we create a filesystem. To accomplish the formatting, we use the `dformat` program specifying the name of the floppy device (usually */dev/fd0*):

```
[me@linuxbox ~]$ sudo fdformat /dev/fd0
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
Verifying ... done
```

Next, we apply a FAT filesystem to the disk with `mkfs`:

```
[me@linuxbox ~]$ sudo mkfs -t msdos /dev/fd0
```

Notice that we use the msdos filesystem type to get the older (and smaller) style file allocation tables. After a disk is prepared, it may be mounted like other devices.

Moving Data Directly to and from Devices

While we usually think of data on our computers as being organized into files, it is also possible to think of the data in "raw" form. If we look at a disk drive, for example, we see that it consists of a large number of "blocks" of data that the operating system sees as directories and files. If we could treat a disk drive as simply a large collection of data blocks, we could perform useful tasks, such as cloning devices.

The `dd` program performs this task. It copies blocks of data from one place to another. It uses a unique syntax (for historical reasons) and is usually used this way:

```
dd if=input_file of=output_file [bs=block_size [count=blocks]]
```

Let's say we had two USB flash drives of the same size and we wanted to exactly copy the first drive to the second. If we attached both drives to the computer and they were assigned to devices */dev/sdb* and */dev/sdc* respectively, we could copy everything on the first drive to the second drive with the following:

```
dd if=/dev/sdb of=/dev/sdc
```

Alternatively, if only the first device were attached to the computer, we could copy its contents to an ordinary file for later restoration or copying:

```
dd if=/dev/sdb of=flash_drive.img
```

Warning: *The* `dd` *command is very powerful. Though its name derives from* data definition, *it is sometimes called* destroy disk *because users often mistype either the* `if` *or* `of` *specifications.* ***Always double-check your input and output specifications before pressing ENTER!***

Creating CD-ROM Images

Writing a recordable CD-ROM (either a CD-R or CD-RW) consists of two steps: first, constructing an *ISO image file* that is the exact filesystem image of the CD-ROM, and second, writing the image file onto the CD-ROM medium.

Creating an Image Copy of a CD-ROM

If we want to make an ISO image of an existing CD-ROM, we can use `dd` to read all the data blocks off the CD-ROM and copy them to a local file. Say we had an Ubuntu CD and we wanted to make an ISO file that we could later use to make more copies. After inserting the CD and determining its device name (we'll assume */dev/cdrom*), we can make the ISO file like so:

```
dd if=/dev/cdrom of=ubuntu.iso
```

This technique works for data DVDs as well, but it will not work for audio CDs as they do not use a filesystem for storage. For audio CDs, look at the `cdrdao` command.

A PROGRAM BY ANY OTHER NAME...

If you look at online tutorials for creating and burning optical media like CD-ROMs and DVDs, you will frequently encounter two programs called `mkisofs` and `cdrecord`. These programs were part of a popular package called `cdrtools` authored by Jörg Schilling. In the summer of 2006, Mr. Schilling made a license change to a portion of the `cdrtools` package that, in the opinion of many in the Linux community, created a license incompatibility with the GNU GPL. As a result, a *fork* of the cdrtools project was started, which now includes replacement programs for `cdrecord` and `mkisofs` named `wodim` and `genisoimage`, respectively.

Creating an Image from a Collection of Files

To create an ISO image file containing the contents of a directory, we use the `enisoimage` program. To do this, we first create a directory containing all the files we wish to include in the image and then execute the `genisoimage` command to create the image file. For example, if we had created a directory called *~/cd-rom-files* and filled it with files for our CD-ROM, we could create an image file named *cd-rom.iso* with the following command:

```
genisoimage -o cd-rom.iso -R -J ~/cd-rom-files
```

The `-R` option adds metadata for the *Rock Ridge extensions*, which allow the use of long filenames and POSIX-style file permissions. Likewise, the `-J` option enables the *Joliet extensions*, which permit long filenames in Windows.

Writing CD-ROM Images

After we have an image file, we can burn it onto our optical media. Most of the commands we discuss below can be applied to both recordable CD-ROM and DVD media.

Mounting an ISO Image Directly

There is a trick that we can use to mount an ISO image while it is still on our hard disk and treat it as though it were already on optical media. By adding the `-o loop` option to `mount` (along with the required `-t iso9660` filesystem type), we can mount the image file as though it were a device and attach it to the filesystem tree:

```
mkdir /mnt/iso_image
mount -t iso9660 -o loop image.iso /mnt/iso_image
```

In the example above, we created a mount point named */mnt/iso_image* and then mounted the image file *image.iso* at that mount point. After the image is mounted, it can be treated just as though it were a real CD-ROM or DVD. *Remember to unmount the image when it is no longer needed.*

Blanking a Rewritable CD-ROM

Rewritable CD-RW media need to be erased or *blanked* before being reused. To do this, we can use `wodim`, specifying the device name for the CD writer and the type of blanking to be performed. The `wodim` program offers several types. The most minimal (and fastest) is the `fast` type:

```
wodim dev=/dev/cdrw blank=fast
```

Writing an Image

To write an image, we again use `wodim`, specifying the name of the optical media writer device and the name of the image file:

```
wodim dev=/dev/cdrw image.iso
```

In addition to the device name and image file, `wodim` supports a very large set of options. Two common ones are `-v` for verbose output and `-dao`, which writes the disc in *disc-at-once* mode. This mode should be used if you are preparing a disc for commercial reproduction. The default mode for `wodim` is *track-at-once*, which is useful for recording music tracks.

Extra Credit

It's often useful to verify the integrity of an ISO image that we have downloaded. In most cases, a distributor of an ISO image will also supply a *checksum file*. A checksum is the result of an exotic mathematical calculation resulting in a number that represents the content of the target file. If the contents of the file change by even one bit, the resulting checksum will be much different. The most common method of checksum generation uses the `md5sum` program. When you use `md5sum`, it produces a unique hexadecimal number:

```
md5sum image.iso
34e354760f9bb7fbf85c96f6a3f94ece  image.iso
```

After you download an image, you should run `md5sum` against it and compare the results with the `md5sum` value supplied by the publisher.

In addition to checking the integrity of a downloaded file, we can use `md5sum` to verify newly written optical media. To do this, we first calculate the checksum of the image file and then calculate a checksum for the medium. The trick to verifying the medium is to limit the calculation to only the portion of the optical medium that contains the image. We do this by determining the number of 2048-byte blocks the image contains (optical media is always written in 2048-byte blocks) and reading that many blocks from the medium. On some types of media, this is not required. A CD-R written in disc-at-once mode can be checked this way:

```
md5sum /dev/cdrom
34e354760f9bb7fbf85c96f6a3f94ece  /dev/cdrom
```

Many types of media, such as DVDs, require a precise calculation of the number of blocks. In the example below, we check the integrity of the image file *dvd-image.iso* and the disc in the DVD reader */dev/dvd*. Can you figure out how this works?

```
md5sum dvd-image.iso; dd if=/dev/dvd bs=2048 count=$(( $(stat -c "%s" dvd-image
.iso) / 2048 )) | md5sum
```