Part Workbook 5. The Linux Filesystem

Table of Contents

Chapter 1. File Details

Key Concepts

- The term file refers to regular files, directories, symbolic links, device nodes, and others.
- All files have common attributes: user owner, group owner, permissions, and timing information. This information is stored in a structure called an *inode*.
- File names are contained in data structures called *dentries* (directory entries).
- A file's inode information can be examined with the **ls -l** and **stat** commands.
- Within the Linux kernel, files are generally identified by inode number. The **ls -i** command can be used to examine inode numbers.

Discussion

How Linux Save Files

Suppose elvis opens a text editor, and composes the following shopping list.

```
eggs
bacon
milk
```

When he is finished, and closes the editor, he is asked what he would like to name the file. He chooses `shopping.txt`. Once he is done, he lists the contents of the directory to make sure it is there.

```
[elvis@station elvis]$ ls -l
total 4
-rw-rw-r--    1 elvis    elvis          16 Jul 11 07:54 shopping.txt
```

This short example illustrates the three components Linux associates with a file.

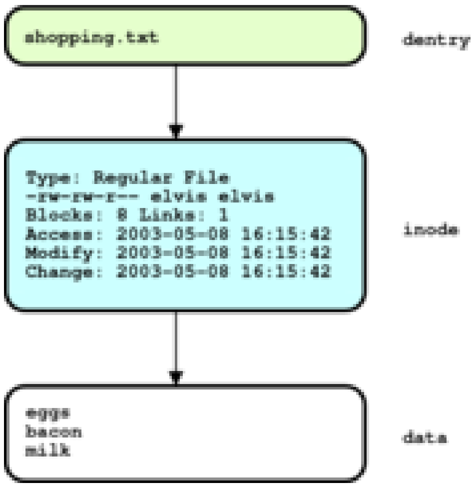
data
: The data is the content of the file, in this case, the 16 bytes that compose elvis's shopping list (13 visible characters, and 3 invisible "return" characters that indicate the end of a line). In Linux, as in Unix, every file's content is stored as a series of bytes.

metadata
: In addition to its content, every file in Linux has extra information associated with it. The entire last Workbook focused on some of this information, namely the file's user owner, group owner, and permissions. Other information, such as the last time the file was modified or read, is stored as well. Much of this *metadata* is reported when you run the **ls -l** command. In Linux (and Unix), all of the extra information associated with a file (with the important exception discussed next) is stored in a structure called an *inode*.

filename
: The filename is the exception to the rule. Although the filename could be considered metadata associated with the file, it is not stored in the inode directly. Instead, the filename is stored in a structure called a *dentry*. (The term *dentry* is a shortening of *d*irectory *entry*, and, as we will see in a later Lesson, the structure is closely associated with directories.) In essence, the filename associates a name with an inode.

In summary, there are three structures associated with every file: a *dentry* which contains a filename and refers to an *inode*, which contains the file's metadata and refers to the file's *data*. Understanding the relationships between these structures helps in understanding later concepts, such as links and directories. These structures are summarized in the figure below.

Figure 1.1. File Structures



What's in an inode?

In Linux (and Unix), every file that exists in the filesystem has an associated inode which stores all of the file's information, except for the filename. What can be found in an inode?

File Type

In Linux (and Unix), the term *file* has a very general meaning: anything that exists in the filesystem (and thus has an inode associated with it) is a file. This includes regular files and directories, which we have already encountered, symbolic links and device nodes which we will soon encounter, and a couple of more obscure creatures which are related to interprocess communication, and beyond the scope of the course. The possible file types are listed in the table below.

Table 1.1. Linux (Unix) File Types

File Type	ls abbr.	Use
Regular File	-	Storing data
Directories	d	Organizing files
Symbolic Links	l	Referring to other files
Character Device Nodes	c	Accessing devices
Block Device Nodes	b	Accessing devices

File Type	ls abbr.	Use
Named Pipes	p	Interprocess communication
Sockets	s	Interprocess communication

Each of the seven file types mentioned above uses the same inode structure, so they each have the same types of attributes: owners, permissions, modify times, etc. When listing files with **ls -l**, the file type of a file is identified by the first character, using the abbreviations found in the second column above.

Note

The term *file* is overloaded in Linux (and Unix), and has two meanings. When used in sentences such as "every file has an inode", the term refers to any of the file types listed in the table above. When used in sentences such as "The **head** command only works on files, not directories", the term file is referring only to the specific type of file that holds data. Usually, the meaning is clear from context. When a distinction has to be made, the term *regular file* is used, as in "The **ls -l** command identifies regular files with a hyphen (-)".

Ownerships and Permissions

As discussed in the previous Workbook, every (regular) file and directory has a group owner, a user owner, and a collection of three sets of read, write, and execute permissions. Because this information is stored in a file's inode, and the inode structure is the same for all files, all seven file types use the same mechanisms for controlling who has access to them, namely **chmod**, **chgrp**, and **chown**.

When listing files with **ls -l**, the first column displays the permissions (as well as file type), the third the user owner, and the fourth the group owner.

Timing Information

Each inode stores three times relevant to the file, conventionally called the *atime*, *ctime*, and *mtime*. These times record the last time a file was accessed (read), changed, or modified, respectively.

Table 1.2. File times

Abbreviation	Name	Purpose
atime	Access Time	Updates whenever the file's data is read
ctime	Change Time	Updates whenever the file's inode information changes
mtime	Modify Time	Updates whenever the file's data changes

What's the difference between *change* and *modify*? When a file's data changes, the file is said to be modified, and the mtime is updated. When a file's inode information changes, the file is said to be changed, and the file's ctime is updated. Modifying a file (and thus changing the mtime) causes the ctime to update as well, while merely reading a file (and thus changing the atime) does not cause the ctime to udpate.

What about create time?

Often, people mistake Unix's *ctime* for a "creation time". Oddly, traditional Unix (and Linux) does not record the fixed time a file was created, a fact which many consider a weakness in the design of the Unix filesystem.

File length and size

The inode records two measures of how large a file is: The file's *length* (which is the actual number of bytes of data), and the file's *size* (which is the amount of disk space the file consumes). Because of the low level details of how files are stored on a disk, the two differ. Generally, the file's size increments in chunks (usually 4 kilobytes) at a time, while the length increases byte by byte as information is added to the file. When listing files with the **ls -l** command, the file's length (in bytes) is reported in the 5th column. When listing files with the **ls -s** command, the file's size (in kilobytes) is reported instead.

Link Count

Lastly, the inode records a file's link count, or the number of dentries (filenames) that refer to the file. Usually, regular files only have one name, and the link count as one. As we will find out, however, this is not always the case. When listing files with **ls -l**, the second column gives the link count.

Viewing inode information with the stat command

Red Hat Enterprise Linux includes the **stat** command for examining a file's inode information in detail. In Unix programming, a file's collection of inode information is referred to as the *status* of the file. The **stat** command can be thought of as reporting the **stat**us of a file.

Note

The **stat** command is often not installed by default in Red Hat Enterprise Linux. If you find that your machine does not have the **stat** command, have your instructor install the stat RPM package file.

`stat` [OPTION] FILE...

Display file (or filesystem) status information.

Switch	Effect
-c, --format=FORMAT	Print only the requested information using the specified format. See the stat(1) man page for more information.
-f, --filesystem	Show information about the filesystem the file belongs to, instead of the file.
-t, --terse	Print output in terse (single line) form

In the following example, madonna examines the inode information on the file `/usr/games/fortune`.

```
[madonna@station madonna]$ stat /usr/games/fortune
  File: `/usr/games/fortune' ❶
  Size: 17795 ❷          Blocks: 40   ❸      IO Block: 4096   Regular File  ❹
Device: 303h/771d       Inode: 540564      Links: 1 ❺
Access: (0755/-rwxr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root) ❻
Access: 2003-07-09 02:36:41.000000000 -0400 ❼
Modify: 2002-08-22 04:14:02.000000000 -0400
Change: 2002-09-11 11:38:09.000000000 -0400
```

❶ The name of the file. This is information is not really stored in the inode, but in the dentry, as explained above.
❷ Inconveniently for the terminology introduced above, the **stat** command labels the length of the file "Size".
❸ The number of filesystem blocks the file consumes. Apparently, the **stat** command is using a blocksize of 2 kilobytes. [1]
❹ The file type, in this case, a regular file.
❺ The link count, or number of filenames that link to this inode. (Don't worry if you don't understand this yet.)
❻ The file's user owner, group owner, and permissions.
❼ The atime, mtime, and ctime for the file.

Viewing inode information with the ls command

While the **stat** command is convenient for listing the inode information of individual files, the **ls** command often does a better job summarizing the information for several files. We reintroduce the **ls** command, this time discussing some of its many command line switches relevant for displaying inode information.

`ls` [OPTION...] FILE...

List the files FILE..., or if a directory, list the contents of the directory.

Switch	Effect
-a, --all	Include files that start with .
-d, --directory	If FILE is a directory, list information about the directory itself, not the directory's contents.
-F, --classify	Decorate filenames with one of *, /, =, @, or | to indicate file type.
-h, --human-readable	Use "human readable" abbreviations when reporting file lengths.
-i, --inode	List index number of each file's inode.
-l	Use long listing format
-n, --numeric-uid-gid	Use numeric UIDs and GIDs, rather then usernames and groupnames.
-r, --reverse	Reverse sorting order.
-R, --recursive	List subdirectories recursively.
--time=WORD	Report (or sort by) time specified by WORD instead of mtime. WORD may be one of "atime", "access", "ctime", or "status".
-t	Sort by modification time.

In the following example, madonna takes a long listing of all of the files in the directory `/usr/games`. The different elements reported by **ls -l** are discussed in detail.

```
[madonna@station madonna]$ ls -l /usr/games/
total 28 ❶
drwxr-xr-x❷   3❸root❹   root❺        4096❻Jan 29 09:40❼chromium
-rwxr-xr-x    1 root     root        17795 Aug 22  2002 fortune
dr-xrwxr-x    3 root     games        4096 Apr  1 11:49 Maelstrom
```

❶ The total number of blocks used by files in the directory. (Note that this does *not* include subdirectories).
❷ The file type and permissions of the file.
❸ The file's link count, or the total number of dentries (filenames) that refer to this file. (Note that, for directories, this is always greater than 1!. hmmm...)
❹ The file's owner.
❺ The file's group owner.
❻ The length of the file, in bytes (Note that directories have a length as well, and the length seems to increment in blocks. hmmm....)
❼ The file's mtime, or last time the file was modified.

Identifying Files by Inode

While people tend to use filenames to identify files, the Linux kernel usually uses the inode directly. Within a filesystem, every inode is assigned a unique inode number. The inode number of a file can be listed with the **-i** command line switch to the **ls** command.

```
[madonna@station madonna]$ ls -iF /usr/games/
 540838 chromium/   540564 fortune*   312180 Maelstrom/
```

In this example, the directory `chromium` has an inode number of 540838. A file can be uniquely identified by knowing its filesystem and inode number.

Examples

Comparing file sizes with ls -s and ls -l

The user elvis is examining the sizes of executable files in the `/bin` directory. He first runs the **ls -s** command.

```
[elvis@station elvis]$ ls -s /bin
total 4860
   4 arch              0 domainname    20 login            92 sed
  96 ash              56 dumpkeys      72 ls               32 setfont
 488 ash.static       12 echo          72 mail             20 setserial
  12 aumix-minimal    44 ed            20 mkdir             0 sh
   0 awk               4 egrep         20 mknod            16 sleep
...
```

Next to each file name, the **ls** command reports the size of the file in kilobytes. For example, the file `ash` takes up 96 Kbytes of disk space. In the (abbreviated) output, that all of the sizes are divisible by four. Apparently, when storing a file on the disk, disk space gets allocated to files in chunks of 4 Kbytes. (This is referred to as the "blocksize" of the filesystem). Note that the file `awk` seems to be taking up no space.

Next, elvis examines the directory information using the **ls -l** command.

```
[elvis@station elvis]$ ls -l /bin
total 4860
-rwxr-xr-x    1 root     root         2644 Feb 24 19:11 arch
-rwxr-xr-x    1 root     root        92444 Feb  6 10:20 ash
-rwxr-xr-x    1 root     root       492968 Feb  6 10:20 ash.static
-rwxr-xr-x    1 root     root        10456 Jan 24 16:47 aumix-minimal
lrwxrwxrwx    1 root     root            4 Apr  1 11:11 awk -> gawk
...
```

This time, the lengths of the files are reported in bytes. Looking again at the file `ash`, the length is reported as 92444 bytes. This is reasonable, because rounding up to the next 4 Kilobytes, we get the 96 Kbytes reported by the **ls -s** command. Note again the file `awk`. The file is not a regular file, but a Symbolic Link, which explains why it was consuming no space. Symbolic Links will be discussed in more detail shortly.

Lastly, elvis is curious about the permissions on the `/bin` directory. When he runs **ls -l /bin**, however, he get a listing of the *contents* of the `/bin` directory, no the directory itself. He solves his problem by adding the **-d** command line switch.

```
[elvis@station elvis]$ ls -ld /bin
drwxr-xr-x    2 root     root         4096 Jul  8 09:29 /bin
```

Listing files, sorted by modify time

The user prince is exploring the system log files found in the `/var/log` directory. He is curious about recent activity on the system, so he would like to know which files have been accessed most recently. He first takes a long listing of the directory, and begins examining the reported modify times for the files.

```
[prince@station prince]$ ls -l /var/log
total 16296
-rw-------    1 root     root        20847 Jul 12  2002 boot.log
-rw-------    1 root     root        45034 Jul  6  2002 boot.log.1
```

```
-rw-------    1 root     root        29116 Jun 29  2002 boot.log.2
-rw-------    1 root     root        18785 Jun 22  2002 boot.log.3
-rw-------    1 root     root        15171 Jun 15  2002 boot.log.4
drwxr-xr-x    2 servlet  servlet      4096 Jan 20  2002 ccm-core-cms
-rw-------    1 root     root        57443 Jul 12  2002 cron
-rw-------    1 root     root        62023 Jul  6  2002 cron.1
-rw-------    1 root     root        74850 Jun 29  2002 cron.2
...
```

With 74 files to look at, prince quickly tires of skimming for recent files. Instead, he decides to let the **ls** command do the hard work for him, specifying that the output should be sorted by mtime with the **-t** command line switch.

```
[prince@station prince]$ ls -lt /var/log
total 16296
-rw-------    1 root     root        57443 Jul 12  2002 cron
-rw-------    1 root     root      2536558 Jul 12  2002 maillog
-rw-------    1 root     root       956853 Jul 12  2002 messages
-rw-rw-r--    1 root     utmp       622464 Jul 12  2002 wtmp
-rw-r--r--    1 root     root        22000 Jul 12  2002 rpmpkgs
-rw-r--r--    1 root     root        38037 Jul 12  2002 xorg.0.log
....
```

Now elvis easily reads the `cron`, `maillog`, and `messages` files as the most recently modified files. Curious which log files are *not* being used, prince repeats the command, adding the `-r` command line switch.

```
[prince@station prince]$ ls -ltr /var/log
total 16296
-rw-r--r--    1 root     root        32589 Oct 23  2001 xorg.1.log.old
drwxr-xr-x    2 servlet  servlet      4096 Jan 20  2002 ccm-core-cms
drwxr-xr-x    2 root     root         4096 Feb  3  2002 vbox
-rwx------    1 postgres postgres        0 Apr  1  2002 pgsql
drwx------    2 root     root         4096 Apr  5  2002 samba
-rw-r--r--    1 root     root        42053 May  7  2002 xorg.1.log
-rw-r--r--    1 root     root         1371 May  9  2002 xorg.setup.log
-rw-------    1 root     root            0 Jun  9  2002 vsftpd.log.4
...
```

Decorating listings with ls -F

The user blondie is exploring the `/etc/X11` directory.

```
[blondie@station blondie]$ ls /etc/X11/
applnk         prefdm        sysconfig   xorg.conf.backup          xkb
desktop-menus  proxymngr     twm         xorg.conf.wbx             Xmodmap
fs             rstart        X           xorg.conf.works           Xresources
gdm            serverconfig  xdm         XftConfig.README-OBSOLETE  xserver
lbxproxy       starthere     xorg.conf  xinit                     xsm
```

Because she does not have a color terminal, she is having a hard time distinguishing what is a regular file and what is a directory. She adds the **-F** command line switch to decorate the output.

```
[blondie@station blondie]$ ls -F /etc/X11/
applnk/         rstart/        xorg.conf                  Xmodmap
desktop-menus/  serverconfig/  xorg.conf.backup           Xresources
fs/             starthere/     xorg.conf.wbx              xserver/
gdm/            sysconfig/     xorg.conf.works            xsm/
lbxproxy/       twm/           XftConfig.README-OBSOLETE
prefdm*         X@             xinit/
proxymngr/      xdm/           xkb@
```

Now, the various files are decorated by type. Directories end in a `/`, symbolic links with a `@`, and regular files with executable permissions (implying that they are commands to be run) end in a `*`.

Online Exercises

Viewing file metadata

Lab Exercise

Objective: List files by modify time

Estimated Time: 5 mins.

Specification

1. Create a file in your home directory called `etc.bytime`. The file should contain a long listing of the `/etc` directory, sorted by modify time. The most recently modified file should be on the first line of the file.

2. Create a file in your home directory called `etc.bytime.reversed`. The file should contain a long listing of the `/etc` directory, reverse sorted by modify time. The most recently modified file should be on the last line of the file.

3. Create a file called `etc.inum`, which contains the inode number of the `/etc` directory as its only token. (Note that this is asking for the inode of the directory itself).

Deliverables

1.
    1. A file called `etc.bytime`, which contains a long listing of all files in the `/etc` directory, sorted by modify time, with the most recently modified file first.

    2. A file called `etc.bytime.reversed`, which contains a long listing of all files in the `/etc` directory, sorted by modify time, with the most recently modified file last.

    3. A file called `etc.inum`, which contains the inode number of the file `/etc` directory as its only token.

Suggestions

The first few lines of the file `etc.bytime` should look like the following, although the details (such as modify time) may differ.

```
total 2716
-rw-r--r--    1 root     root          258 May 21 09:27 mtab
-rw-r--r--    1 root     root          699 May 21 09:13 printcap
-rw-r-----    1 root     smmsp       12288 May 21 09:10 aliases.db
-rw-rw-r--    2 root     root          107 May 21 09:10 resolv.conf
-rw-r--r--    1 root     root           28 May 21 09:10 yp.conf
-rw-------    1 root     root           60 May 21 09:10 ioctl.save
-rw-r-----    1 root     root            1 May 21 09:10 lvmtab
drwxr-xr-x    2 root     root         4096 May 21 09:10 lvmtab.d
-rw-r--r--    1 root     root           46 May 21 08:55 adjtime
```

Questions

1. Which of the following is *not* a data structure associated with a file?

a. dentry

b. superblock

c. inode

d. data (blocks)

e. All of the above are data structures associated with files.

2. Which of the following file types does not use a data structure called an inode?

a. regular file

b. directory

c. symbolic link

d. character device node

e. All of the above file types use the inode data structure.

3. Which of the following information is *not* stored in a file's inode?

a. The file's modify time

b. The file's permissions

c. The file's user owner

d. The file's name

e. All of the above information is stored in the inode.

Use the output from the following commands to answer the next 2 questions.

```
[student@station student]$ stat /bin
  File: "/bin"
  Size: 2048       Blocks: 4          IO Block: 4096   Directory
Device: 309h/777d Inode: 44177       Links: 2
Access: (0755/drwxr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
Access: Wed Mar 19 09:38:51 2003
Modify: Wed Jan 22 16:36:06 2003
Change: Wed Jan 22 16:36:06 2003
[student@station student]$ ls -l /usr/bin/tree
-rwxr-xr-x    1 root     root        18546 Jun 23  2002 /usr/bin/tree
```

4. How many blocks are in use by the directory `/bin` as shown above?

a. 2

b. 4

c. 2048

d. 4096

5. What are the permissions for the file `/usr/bin/tree` as shown above?

a. 640

b. 644

c. 755

d. 775

6. Which command(s) will show the size and permissions of the file `/etc/passwd`?

a. stat /etc/passwd

b. df -h

c. cat /etc/passwd

d. ls -l /etc/passwd

7. Which command syntax will show the owner and group of the directory `/etc`?

a. ls /etc

b. ls -l /etc

c. ls -d /etc

d. ls -ld /etc

8. Which time for a file is shown by the **ls -l** command?

a. The file's modify time

b. The file's change time

c. The file's access time

d. The file's creation time

e. None of the above.

9. Which time is updated when a file is read?

a. The file's modify time

b. The file's change time

c. The file's access time

d. A and C

e. All of the above.

10. Which time is updated when data is appended to a file?

a. The file's modify time

b. The file's change time

c. The file's access time

d. A and B

e. All of the above.

Chapter 2. Hard and Soft Links

Key Concepts

- The **ln** command creates two distinct types of links.
- Hard links assign multiple dentries (filenames) to a single inode.
- Soft links are distinct inodes that reference other filenames.

Discussion

The Case for Hard Links

Occasionally, Linux users want the same file to exist two separate places, or have two different names. One approach is by creating a hard link.

Suppose elvis and blondie are collaborating on a duet. They would both like to be able to work on the lyrics as time allows, and be able to benefit from one another's work. Rather than copying an updated file to one another every time they make a change, and keeping their individual copies synchronized, they decide to create a hard link.

Blondie has set up a collaboration directory called `~/music`, which is group owned and writable by members of the group music. She has elvis do the same. She then creates the file `~/music/duet.txt`, **chgrp**s it to the group music, and uses the **ln** command to link the file into elvis's directory.

```
[blondie@station blondie]$ ls -ld music/
drwxrwxr-x    2 blondie  music        4096 Jul 13 05:45 music/
[blondie@station blondie]$ echo "Knock knock" > music/duet.txt
[blondie@station blondie]$ chgrp music music/duet.txt
[blondie@station blondie]$ ln music/duet.txt /home/elvis/music/duet.txt
```

Because the file was *linked*, and not copied, it is the same file under two names. When elvis edits `/home/elvis/music/duet.txt`, he is editing `/home/blondie/music/duet.txt` as well.

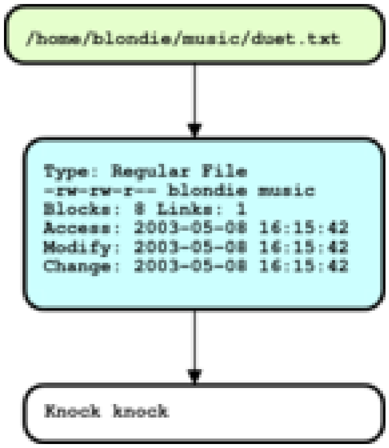
```
[elvis@station elvis]$ echo "who's there?" >> music/duet.txt
```

```
[blondie@station blondie]$ cat music/duet.txt
Knock knock
who's there?
```

Hard Link Details

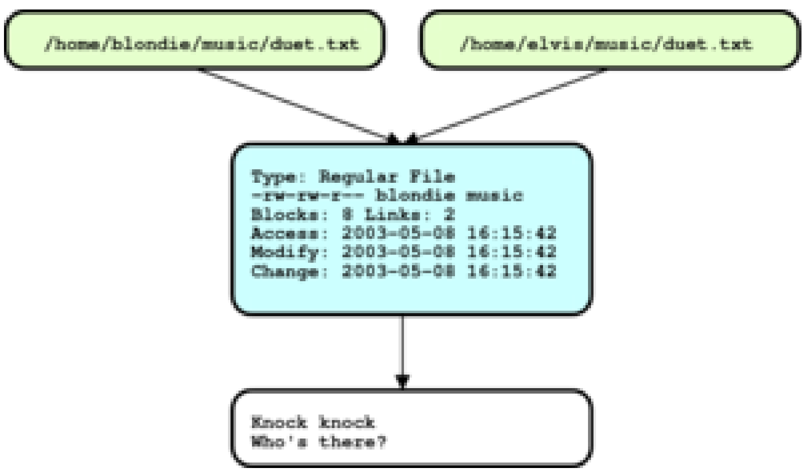
How are hard links implemented? When created, the file `/home/blondie/music/duet.txt` consists of a dentry, an inode, and data, as illustrated below.

Figure 2.1. Regular File



After using the **ln**command to create the link, the file is still a single inode, but there are now two dentries referring to the file.

Figure 2.2. Hard Link



When blondie runs the **ls -l** command, look closely at the second column, which reports the link count for the file.

```
[blondie@station blondie]$ ls -l music/duet.txt
-rw-rw-r--    2 blondie  music          25 Jul 13 06:08 music/duet.txt
```

Until now, we have not been paying much attention to the link count, and it has almost always been 1 for regular files (implying one dentry referencing one inode). Now, however, two dentries are referencing the inode, and the file has a link count of 2. If blondie changes permissions on the file `/home/blondie/music/duet.txt`, what happens to the file `/home/elvis/music/duet.txt`?

```
[blondie@station blondie]$ chmod o-r music/duet.txt
```

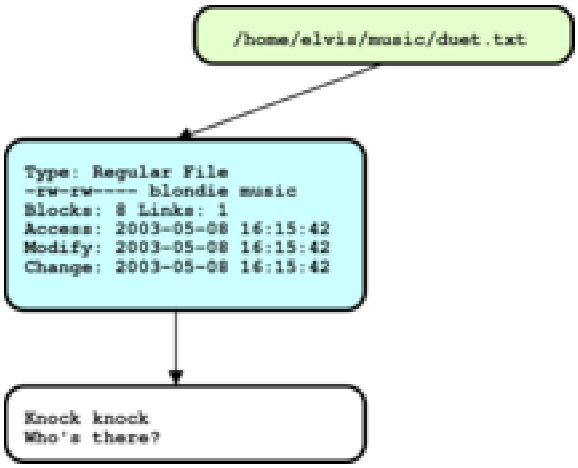
```
[elvis@station elvis]$ ls -l music/duet.txt
-rw-rw----    2 blondie  music          25 Jul 13 06:08 music/duet.txt
```

Because both halves of the link share the same inode, elvis sees the changed permissions as well.

What happens if blondie removes `/home/blondie/music/duet.txt`? The inode `/home/elvis/music/duet.txt` still exists, but with one less dentry referencing it.

```
[blondie@station blondie]$ rm music/duet.txt
```

Figure 2.3. Hard link after half is removed



What would you expect the link count of the file `/home/elvis/music/duet.txt` to be now?

```
[elvis@station elvis]$ ls -l music/duet.txt
-rw-rw----    1 blondie  music          25 Jul 13 06:08 music/duet.txt
```

A little awkwardly, elvis is left with a file owned by blondie, but it is still a valid file, nonetheless. At a low level, the **rm** command is not said to delete a file, but "unlink" it. A file (meaning the file's inode and data) are automatically deleted from the system when its link count goes to 0 (implying that there are no longer any dentries (filenames) referencing the file).

The Case for Soft Links

The other approach to assigning a single file two names is called a soft link. While superficially similar, soft links are implemented very differently from hard links.

The user madonna is obsessively organized, and has collected her regular errands into seven todo lists, one for every day of the week.

```
[madonna@station madonna]$ ls todo/
friday.todo  saturday.todo  thursday.todo  wednesday.todo
monday.todo  sunday.todo    tuesday.todo
```

She consults her todo lists multiple times a day, but finds that she has trouble remembering what day of week it is. She would rather have a single file called `today.todo`, which she updates each morning. She decides to use a soft link instead. Because today is Tuesday, she uses the same **ln** command that is used for creating hard links, but adds the **-s** command line switch to specify a soft link.

```
[madonna@station todo]$ ls
friday.todo  saturday.todo  thursday.todo  wednesday.todo
monday.todo  sunday.todo    tuesday.todo
[madonna@station todo]$ ln -s tuesday.todo today.todo
[madonna@station todo]$ ls -l
total 32
-rw-rw-r--    1 madonna  madonna       138 Jul 14 09:54 friday.todo
-rw-rw-r--    1 madonna  madonna        29 Jul 14 09:54 monday.todo
-rw-rw-r--    1 madonna  madonna       579 Jul 14 09:54 saturday.todo
-rw-rw-r--    1 madonna  madonna       252 Jul 14 09:54 sunday.todo
-rw-rw-r--    1 madonna  madonna       519 Jul 14 09:54 thursday.todo
```

```
lrwxrwxrwx    1 madonna  madonna        12 Jul 14 09:55 today.todo -> tuesday.todo
-rw-rw-r--    1 madonna  madonna        37 Jul 14 09:54 tuesday.todo
-rw-rw-r--    1 madonna  madonna      6587 Jul 14 09:55 wednesday.todo
```

Examine closely file type (the first character of each line in the **ls -l** command) of the newly created file `today.todo`. It is not a regular file ("-"), or a directory ("d"), but a "l", indicating a *symbolic link*. A symbolic link, also referred to as a "soft" link, is a file which references another file by filename. Soft links are similar to *aliases* found in other operating systems. Helpfully, the **ls -l** command also displays what file the soft link refers to, where `today.todo -> tuesday.todo` implies the soft link titled *today.todo* references the file `tuesday.todo`.

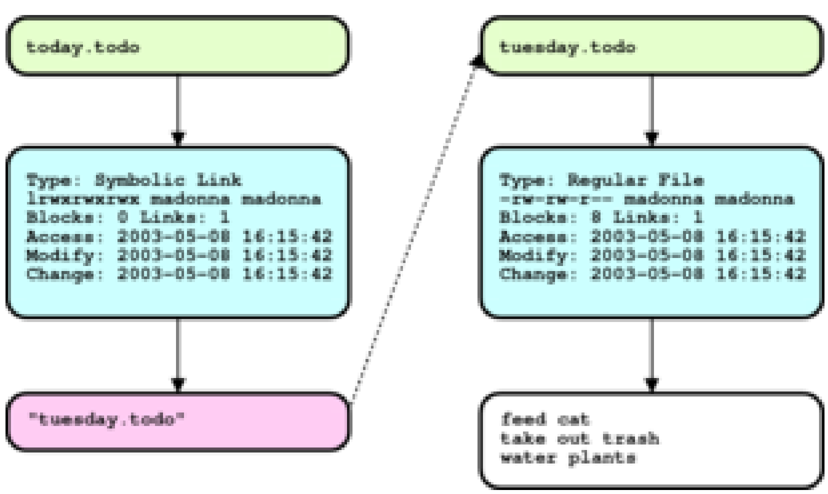
Now, whenever madonna references the file `today.todo`, she is really examining the file `tuesday.todo`.

```
[madonna@station todo]$ cat today.todo
feed cat
take out trash
water plants
[madonna@station todo]$ cat tuesday.todo
feed cat
take out trash
water plants
```

Soft Link Details

How are soft links implemented? When created, the file `tuesday.txt`, like most files, consists of a dentry, an inode, and data, as illustrated below. When the soft link `today.txt` was created, the soft link (unlike a hard link) really is a new file, with a newly created inode. The link is not a regular file, however, but a symbolic link. Symbolic links, rather than storing actual data, store the name of another file. When the Linux kernel is asked to refer to the symbolic link, the kernel automatically *resolves* the link by looking up the new filename. The user (or really, the process on behalf of the user) that referred to the symbolic link doesn't know the difference.

Figure 2.4. Soft Links



Creating Links with the ln Command

As illustrated above, both hard links and soft links are created with the **ln** command.

`ln` [OPTION...] TARGET [LINK]

Create the link LINK referencing the file TARGET.

`ln` [OPTION...] TARGET... [DIRECTORY]

Create link(s) to the file(s) TARGET in the directory DIRECTORY.

Switch	Effect
-f, --force	clobber existing destination files
-s, --symbolic	make symbolic (soft) link instead of hard link

The **ln** command behaves very similarly to the **cp** command: if the last argument is a directory, the command creates links in the specified directory which refer to (and are identically named) to the preceding arguments. Unlike the **cp** command, if only one argument is given, the **ln** command will effectively assume a last argument of ".". When specifying links, the **ln** command expects the name of the original file(s) first, and the name of the link last. Reversing the order doesn't produce the desired results. Again, when in doubt, recall the behavior of the **cp** command.

In the following short example, madonna creates the file `orig.txt`. She then tries to make a hard link to it called `newlnk.txt`, but gets the order of the arguments wrong. Realizing her mistake, she then corrects the problem.

```
[madonna@station madonna]$ date > orig.txt
[madonna@station madonna]$ ln newlnk.txt orig.txt
ln: accessing `newlnk.txt': No such file or directory
[madonna@station madonna]$ ln orig.txt newlnk.txt
```

Issues with Soft Links

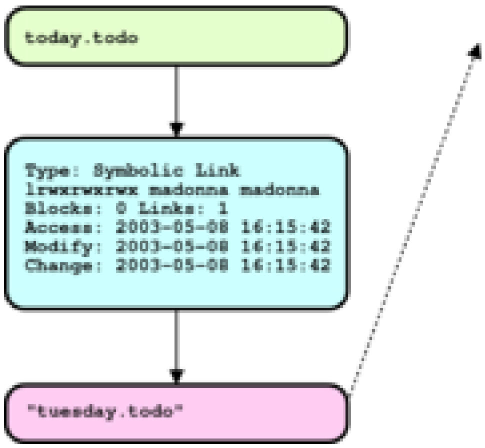
Dangling Links

Soft links are susceptible to a couple of problems that hard links are not. The first is called *dangling* links. What happens if madonna renames or removes the file `tuesday.todo`?

```
[madonna@station todo]$ mv tuesday.todo tuesday.hide
[madonna@station todo]$ ls -l
total 32
-rw-rw-r--    1 madonna  madonna       138 May 14 09:54 friday.todo
-rw-rw-r--    1 madonna  madonna        29 May 14 09:54 monday.todo
-rw-rw-r--    1 madonna  madonna       579 May 14 09:54 saturday.todo
-rw-rw-r--    1 madonna  madonna       252 May 14 09:54 sunday.todo
-rw-rw-r--    1 madonna  madonna       519 May 14 09:54 thursday.todo
lrwxrwxrwx    1 madonna  madonna        12 May 14 09:55 today.todo -> tuesday.todo
-rw-rw-r--    1 madonna  madonna        37 May 14 10:22 tuesday.hide
-rw-rw-r--    1 madonna  madonna      6587 May 14 09:55 wednesday.todo
[madonna@station todo]$ cat today.todo
cat: today.todo: No such file or directory
```

The symbolic link `today.todo` still references the file `tuesday.todo`, but the file `tuesday.todo` no longer exists! When madonna tries to read the contents of `today.todo`, she is met with an error.

Figure 2.5. Dangling Links

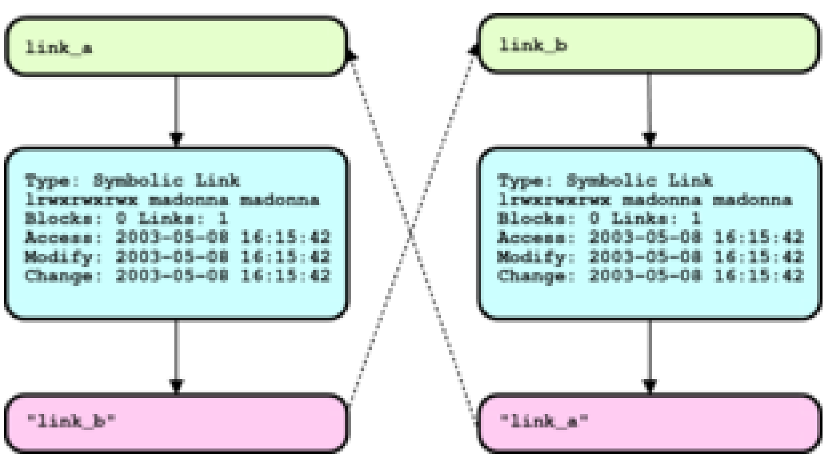


Recursive links

The second problem that symbolic links are susceptible to is recursive links. In day to day use, recursive links are not nearly as common as dangling links, and someone almost has to be looking for them to create them. What if madonna created two symbolic links, `link_a`, which referenced a file named `link_b`, and `link_b`, which references the file `link_a`, as illustrated below?

```
[madonna@station madonna]$ ln -s link_a link_b
[madonna@station madonna]$ ln -s link_b link_a
[madonna@station madonna]$ ls -l
total 0
lrwxrwxrwx    1 madonna  madonna         6 Jul 14 10:41 link_a -> link_b
lrwxrwxrwx    1 madonna  madonna         6 Jul 14 10:41 link_b -> link_a
```

Figure 2.6. Recursive Links



When madonna tries to read `link_a`, the kernel resolves link_a to link_b, which it then resolves back to link_a, and so on. Fortunately, the kernel will only resolve a link so many times before it suspects that it is caught in a recursive link, and gives up.

```
[madonna@station madonna]$ cat link_a
cat: link_a: Too many levels of symbolic links
```

Absolute vs. Relative Soft Links

When creating soft links, users can choose between specifying the link's target using a relative or absolute reference. If the soft link, and its target, are never going to be relocated, the choice doesn't matter. Often, however, users can't anticipate how the files they create will be used in the future. Usually, relative links are more resilient to unexpected changes.

Comparing Hard and Soft Links

When should a soft link be used, and when should a hard link be used? Generally, hard links are more appropriate if both instances of the link have a reasonable use, even if the other instance didn't exist. In the example above, even if blondie decided not to work on the duet and removed her file, elvis could reasonably continue to work. Soft links are generally more appropriate when one file cannot reasonably exist without the other file. In the example above, madonna could not have tasks for "today" if she did not have tasks for "tuesday". These are general guidelines, however, not hard and fast rules.

Sometimes, more practical restrictions make the choice between hard links and soft links easier. The following outlines some of the differences between hard and soft links. Do not be concerned if you do not understand the last two points, they are included for completeness.

Table 2.1. Comparing Hard and Soft Links

Hard Links	Soft Links
Directories may not be hard linked.	Soft links may refer to directories.
Hard links have no concept of "original" and "copy". Once a hard link has been created, all instances are treated equally.	Soft links have a concept of "referrer" and "referred". Removing the "referred" file results in a dangling referrer.
Hard links must refer to files in the same filesystem.	Soft links may span filesystems (partitions).
Hard links may be shared between "chroot"ed directories.	Soft links may not refer to files outside of a "chroot"ed directory.

Examples

Working with hard links

In her home directory, blondie has a file called `rhyme` and a directory called `stuff`. She takes a long listing with **ls -li**, where the **-i** command line switch causes the **ls** command to print the inode number of each file as the first column of output.

Because each inode in a filesystem has a unique inode number, the inode number can be used to identify a file. In fact, when keeping track of files internally, the kernel usually refers to a file by inode number rather than filename.

```
[blondie@station blondie]$ ls -il
total 8
 246085 -rw-rw-r--    1 blondie  blondie        51 Jul 18 15:29 rhyme
 542526 drwxrwxr-x    2 blondie  blondie      4096 Jul 18 15:34 stuff
```

She creates a hard link to the `rhyme` file, and views the directory contents again.

```
[blondie@station blondie]$ ln rhyme hard_link
[blondie@station blondie]$ ls -li
 246085 -rw-rw-r--    2 blondie  blondie        51 Jul 18 15:29 hard_link
```

```
 246085 -rw-rw-r--    2 blondie  blondie        51 Jul 18 15:29 rhyme
 542526 drwxrwxr-x    2 blondie  blondie      4096 Jul 18 15:34 stuff
```

The link count for `rhyme` is now 2. Additionally notice that the inode number for both `rhyme` and `hard_link` is 246085, implying that although there are two names for the file (two dentries), there is only one inode.

If we change the permissions on `rhyme`, the permissions on `hard_link` will change as well. Why? the two filenames refer to the same inode. Because the inode references a file's content, they also share the same data.

```
[blondie@station blondie]$ chmod 660 rhyme
[blondie@station blondie]$ ls -li
 246085 -rw-rw----    2 blondie  blondie        51 Jul 18 15:29 hard_link
 246085 -rw-rw----    2 blondie  blondie        51 Jul 18 15:29 rhyme
 542526 drwxrwxr-x    2 blondie  blondie      4096 Jul 18 15:34 stuff
[blondie@station blondie]$ echo "Hickory, Dickory, Dock," > rhyme
[blondie@station blondie]$ echo "Three mice ran up a clock." >> hard_link
[blondie@station blondie]$ cat rhyme
Hickory, Dickory, Dock,
Three mice ran up a clock.
```

Moving or even removing the original file has no effect on the link file.

```
[blondie@station blondie]$ mv rhyme stuff
[blondie@station blondie]$ ls -Rli
.:
total 8
 246085 -rw-rw----    2 blondie  blondie        51 Jul 18 15:29 hard_link
 542526 drwxrwxr-x    2 blondie  blondie      4096 Jul 18 15:38 stuff

./stuff:
total 4
 246085 -rw-rw----    2 blondie  blondie        51 Jul 18 15:29 rhyme
```

Working with soft links

The user blondie now repeats the exact same exercise, but uses a soft link instead of a hard link. She starts with an identical setup as the example above.

```
[blondie@station blondie]$ ls -li
total 8
 246085 -rw-rw-r--    1 blondie  blondie        29 Jul 18 15:25 rhyme
 542526 drwxrwxr-x    2 blondie  blondie      4096 Jul 18 15:25 stuff
```

She now creates a soft link to the `rhyme` file, and views the directory contents again.

```
[blondie@station blondie]$ ln -s rhyme soft_link
[blondie@station blondie]$ ls -li
total 8
 246085 -rw-rw-r--    1 blondie  blondie        29 Jul 18 15:25 rhyme
 250186 lrwxrwxrwx    1 blondie  blondie         5 Jul 18 15:26 soft_link -> rhyme
 542526 drwxrwxr-x    2 blondie  blondie      4096 Jul 18 15:25 stuff
```

In contrast to the hard link above, the soft link exists as a distinct inode (with a distinct inode number), and the link counts of each of the files remains 1. This implies that there are now two dentries and two inodes. When referenced, however, the files behave identically as in the case of hard links.

```
[blondie@station blondie]$ echo "Hickory, Dickory, Dock," > rhyme
[blondie@station blondie]$ echo "Three mice ran up a clock." >> soft_link
[blondie@station blondie]$ cat rhyme
Hickory, Dickory, Dock,
Three mice ran up a clock.
```

Unlike the hardlink, the softlink cannot survive if the original file is moved or removed. Instead, blondie is left with a dangling link.

```
[blondie@station blondie]$ ls -liR
.:
total 4
 250186 lrwxrwxrwx    1 blondie  blondie         5 Jul 18 15:26 soft_link -> rhyme
 542526 drwxrwxr-x    2 blondie  blondie      4096 Jul 18 15:31 stuff

./stuff:
total 4
 246085 -rw-rw-r--    1 blondie  blondie        51 Jul 18 15:29 rhyme
[blondie@station blondie]$ cat soft_link
cat: soft_link: No such file or directory
```

Working with soft links and directories

Soft links are also useful as pointers to directories. Hard links can only be used with ordinary files.

```
[einstein@station einstein]$ ln -s /usr/share/doc docs
[einstein@station einstein]$ ls -il
  10513 lrwxrwxrwx    1 einstein einstein       14 Mar 18 20:31 docs -> /usr/share/doc
  10512 -rw-rw----    2 einstein einstein      949 Mar 18 20:10 hard_link
  55326 drwxrwxr-x    2 einstein einstein     1024 Mar 18 20:28 stuff
```

The user einstein can now easily change to the docs directory without having to remember or type the long absolute path.

Online Exercises

Creating and Managing Links

Lab Exercise

Objective: Create and Manage hard and soft links

Estimated Time: 10 mins.

Specification

All files should be created in your home directory.

1. Create a file called `cal.orig` in your home directory, which contains a text calendar of the current month (as produced by the **cal** command).

2. Create a hard link to the file `cal.orig`, called `cal.harda`

3. Create a hard link to the file `cal.orig`, called `cal.hardb`

4. Create a soft link to the file `cal.orig`, called `cal.softa`

5. Remove the file `cal.orig`, so that the soft link you just created is now a dangling link.

6. Create a soft link to the `/usr/share/doc` directory, called `docabs`, using an absolute reference.

7. Create a soft link to the `../../usr/share/doc` directory, called `docrel`, using a relative reference. (Note: depending on the location of your home directory, you may need to add or remove

some `..` references from the proceeding filename. Include enough so that the the soft link is a true relative reference to the `/usr/share/doc` directory.

If you have finished the exercise correctly, you should be able to reproduce output similar to the following.

```
[student@station student]$ ls -l
total 12
-rw-rw-r--    2 student  student       138 Jul 21 10:03 cal.harda
-rw-rw-r--    2 student  student       138 Jul 21 10:03 cal.hardb
lrwxrwxrwx    1 student  student         8 Jul 21 10:03 cal.softa -> cal.orig
lrwxrwxrwx    1 student  student        14 Jul 21 10:03 docabs -> /usr/share/doc
lrwxrwxrwx    1 student  student        19 Jul 21 10:03 docrel -> ../../usr/share/doc
```

Deliverables

1.
    1. A file called `cal.harda`.
    2. A file called `cal.hardb`, which is a hard link to the proceeding file.
    3. A file called `cal.softa`, which is a dangling soft link to the nonexistent file `cal.orig`.
    4. A file called `docabs`, which is a soft link to the `/usr/share/doc` directory, using an absolute reference.
    5. A file called `docrel`, which is a soft link to the `/usr/share/doc` directory, using a relative reference.

Sharing a Hard Linked File

Lab Exercise

Objective: Share a hard linked file between two users.

Estimated Time: 15 mins.

Specification

You would like to create a hard linked file that you will share with another user.

1. As your primary user, create a subdirectory of `/tmp` named after your account name, such as `/tmp/`*`student`*, where *`student`* is replaced with your username.
2. Still as your primary user, create a file called `/tmp/`*`student`*`/novel.txt`, which contains the text "Once upon a time."

```
[student@station student]$ mkdir /tmp/student
[student@station student]$ echo "Once Upon a Time," > /tmp/student/novel.txt
[student@station student]$ ls -al /tmp/student/
total 12
drwxrwxr-x    2 student  student      4096 Jul 21 10:13 .
drwxrwxrwt   28 root     root         4096 Jul 21 10:12 ..
-rw-rw-r--    1 student  student        18 Jul 21 10:13 novel.txt
```

3. Now log in as (or **su -** to) your first alternate account. Create a directory in `/tmp` which is named after your alternate account, such as `/tmp/`*`student_a`*.
4. As your first alternate user, in your newly created directory, create a hard link to the file `/tmp/`*`student`*`/novel.txt`, called `/tmp/`*`student_a`*`/novel.lnk`. Try to edit the file, changing the

line from "Once upon a time,", to "It was a dark and stormy night.". Why did you have difficulties? Are you able to modify the ownerships or permissions of the file `novel.lnk`? Why or Why not?

```
[student@station student]$ su - student_a
Password:
[student_a@station student_a]$ mkdir /tmp/student_a
[student_a@station student_a]$ ln /tmp/student/novel.txt /tmp/student_a/novel.ln
k
[student_a@station student_a]$ echo "It was a dark and stormy night." >> /tmp/st
udent_a/novel.lnk
-bash: /tmp/student_a/novel.lnk: Permission denied
```

5. As your primary user, adjust the permissions and/or ownerships on the file `/tmp/`*`student`*`/novel.txt`, so that your first alternate user is able to modify the file.

6. As your first alternate user, apply the edit mentioned above. When you are finished, the file `/tmp/`*`student_a`*`/novel.lnk` should contain only the text "It was a dark and stormy night.".

Deliverables

1.
   1. A file called `/tmp/`*`student`*`/novel.txt`, where *`student`* is replaced with the name of your primary user, owned by your primary user. The file should have appropriate ownerships and permissions so that it can be modified by your first alternate account. The file should contain only the text "It was a dark and stormy night.".

   2. A file called `/tmp/`*`student_a`*`/novel.lnk`, where *`student_a`* is replaced with the name of your first alternate account. The file should be a hard link to the file `/tmp/`*`student`*`/novel.txt`.

Questions

Use the output from the following command to help answer the next 5 questions.

```
[student@station student]$ ls -li /usr/bin/
 342997 lrwxrwxrwx    1 root     root            5 Apr  1 11:18 ./bunzip2 -> bzip2
 342998 lrwxrwxrwx    1 root     root            5 Apr  1 11:18 ./bzcat -> bzip2
 342999 lrwxrwxrwx    1 root     root            6 Apr  1 11:18 ./bzcmp -> bzdiff
 343004 lrwxrwxrwx    1 root     root            6 Apr  1 11:18 ./bzless -> bzmore
 343066 lrwxrwxrwx    1 root     root           16 Apr  1 11:12 ./gunzip -> ../../bin/gunzip
 343112 lrwxrwxrwx    1 root     root           14 Apr  1 11:12 ./gzip -> ../../bin/gzip
 343136 lrwxrwxrwx    1 root     root            2 Apr  1 11:21 ./lz -> uz
 343123 -rwxr-xr-x    3 root     root        57468 Jan 24 23:42 ./rx
 343123 -rwxr-xr-x    3 root     root        57468 Jan 24 23:42 ./rz
 343065 -rwxr-xr-x    3 root     root        61372 Jan 24 23:42 ./sb
 343065 -rwxr-xr-x    3 root     root        61372 Jan 24 23:42 ./sx
 343065 -rwxr-xr-x    3 root     root        61372 Jan 24 23:42 ./sz
 347486 lrwxrwxrwx    1 root     root            8 Jul 21 16:43 ./uncompress -> compress
 343117 -rwxr-xr-x    3 root     root         3029 Jan 31 11:08 ./zegrep
 343117 -rwxr-xr-x    3 root     root         3029 Jan 31 11:08 ./zfgrep
 343117 -rwxr-xr-x    3 root     root         3029 Jan 31 11:08 ./zgrep
```

Note that many lines have been omitted from the previous command's output, leaving only a few interesting one.

1. Which of the following files share the same inode?

   a. `lz`

   b. `uz`

c. `rx`

d. `sb`

e. `sx`

f. `sz`

2. Removing which of the following files would create a dangling link?

   a. `bzip2`

   b. `lz`

   c. `uz`

   d. `sb`

   e. `compress`

   f. `zgrep`

3. How many files (listed or not) share inode number 343123?

   a. 1

   b. 2

   c. 3

   d. None of the above.

   e. It cannot be determined from the information provided.

4. Examine the lengths of the symbolic links such as `bzcat`, `lz`, and `uncompress`, as reported in the 6th column of the output above. Which of the following best explains what the length of a symbolic link represents?

   a. The length represents the length of the filename that the symbolic link resolves to.

   b. The length represents the number of files which share the soft link.

   c. The length is the length of the file that the symbolic link resolves to.

   d. The length is arbitrary, and serves no purpose.

   e. None of the above.

Suppose the system administrator moved the `/usr/bin` directory, as shown.

```
[root@station root]# mv /usr/bin /usr/lib/bin
```

5. Which files in the new `/usr/lib/bin` directory would be dangling symbolic links?

   a. `bzcat`

   b. `gunzip`

   c. `gzip`

   d. `lz`

   e. `uncompress`

   f. `zgrep`

6. What is the correct command for creating a shortcut from your home directory that points to a `/data/project` directory?

   a. **`ln /data/project /home/student/project`**

   b. **`ln /home/student/project /data/project`**

   c. **`ln -s /data/project /home/student/project`**

   d. **`ln -s /home/student/project /data/project`**

7. Projects A, B, and C all use the file `/data/script`. All teams want to have a copy in their own project directory but they also want to be sure that any changes to the original file are reflected in their copies. Using project_A as an example, which commands would accomplish this goal?

   a. **`ln /data/script /data/project_A/script`**

   b. **`cp /data/script /data/project_A/script`**

   c. **`ln -s /data/script /data/project_A/script`**

   d. **`ln -s /data/project_A/script /data/script`**

   e. A and C

8. The team leader of project_D wants to use the script as a starting point, but intends to modify it in a way that the other teams will not want to use. What is the best way for to get the original script?

   a. **`ln /data/script /data/project_D/script`**

   b. **`cp /data/script /data/project_D/script`**

   c. **`ln -s /data/script /data/project_D/script`**

   d. **`ln -s /data/project_D/script /data/script`**

Chapter 3. Directories and Device Nodes

Key Concepts

- The term file refers to regular files, directories, symbolic links, device nodes, and others.
- All files have common attributes: user owner, group owner, permissions, and timing information.
- File meta-information is contained in a data structure called inodes.
- File names are contained in data structures called directory entries (dentries).
- File meta-information can be examined with the **ls -l** and **stat** commands.

Discussion

Directories

Directory Structure

Earlier, we introduced two structures associated with files, namely *dentries*, which associate filenames with inodes, and *inodes*, which associate all of a file's attributes with its content. We now see how these structures relate to directories.

The user prince is using a directory called `report` to manage files for a report he is writing. He recursively lists the `report` directory, including **-a** (which specifies to list "all" entries, including those beginning with a ".") and **-i** (which specifies to list the inode number of a file as well as filename). What results is the following listing of the directories and files, along with their inode number.

```
[prince@station prince]$ ls -iaR report
report:
 592253 .   249482 ..   592255 html   592254 text

report/html:
 592255 .   592253 ..   592261 chap1.html   592262 chap2.html   592263 figures

report/html/figures:
 592263 .   592255 ..   592264 image1.png

report/text:
 592254 .   592253 ..   592257 chap1.txt   592258 chap2.txt
```

Notice the files (directories) "." and ".." are included in the output. As mentioned in a previous Workbook, the directory "." refers to a directory itself, and the directory ".." refers to the directory's parent. Every directory actually contains entries called `.` and `..`, though they are treated as hidden files (because they "begin with ."), and not displayed unless **-a** is specified.

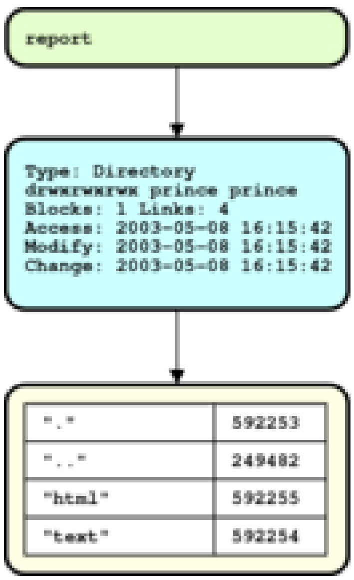
The same directories, files, and inode numbers are reproduced below, in an easier format.

```
path                    | inode
--------------------------------------------
```

```
report/                 | 592253
|-- html                | 592255
|   |-- chap1.html      | 592261
|   |-- chap2.html      | 592262
|   `-- figures         | 592263
|       `-- image1.png  | 592264
`-- text                | 592254
    |-- chap1.txt       | 592257
    `-- chap2.txt       | 592258
```

As seen in the following figure of the `report` directory, directories have the same internal structure as regular files: a dentry, an inode, and data. The data that directories store, however, are the dentries associated with the files the directory is said to contain. A directory is little more than a table of dentries, mapping filenames to the underlying inodes that represent files. When introduced, the name *dentry* was said to be derived from *d*irectory *entry*. We now see that directories are no more complicated than that: a directory is a collection of dentries.

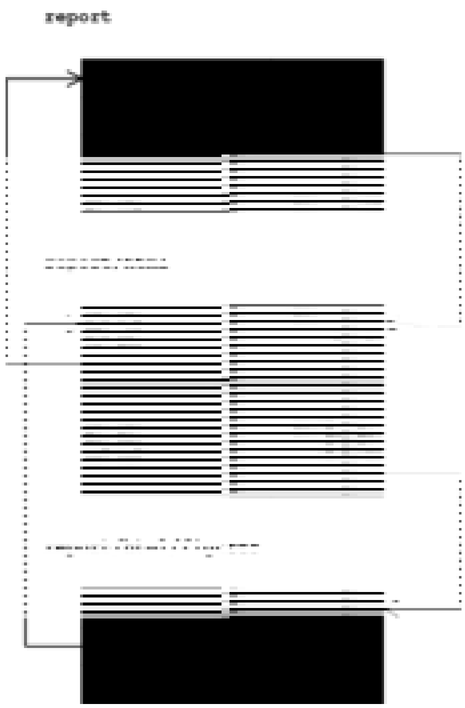
Figure 3.1. The Internal Structure of Directories



Directory Links

Earlier, we observed that the link counts of directories, as reflected in the second column of the **ls -l** command, was always two or greater. This follows naturally from the fact that every directory is referenced at least twice, once by itself (as the directory "."), and once by its parent (with an actual directory name, such as `report`). The following diagram of the dentries contained by the `report` directory, its subdirectory `html`, and its subdirectory `figures`, helps to illustrate.

Figure 3.2. Dentry Tables for the `report`, `report/html`, and `report/html/figures` directories.



When prince takes a long listing of the `report` directory, he sees the four files in the first table.

```
[prince@station prince]$ ls -ial report
total 16
 592253 drwxrwxr-x    4 prince   prince       4096 Jul 14 13:27 .
 249482 drwx-----x    6 prince   prince       4096 Jul 14 13:27 ..
 592255 drwxrwxr-x    3 prince   prince       4096 Jul 14 13:49 html
 592254 drwxrwxr-x    2 prince   prince       4096 Jul 14 13:49 text
```

Every file in the listing is a directory, and the link count (here the third column, since the inode number has been prepended as the first column) of each is greater than or equal to two. Can we account for each of the links? Let's start by listing the references to inode number 592253 (the `report` directory, or above, simply "`.`").

1. The entry `.`, found in the directory itself.
2. The parent directory (not pictured) contains an entry called `report`, which references the same inode.
3. The subdirectory `html` contains an entry called `..`, which references inode 592253 as its parent directory.
4. Likewise, the subdirectory `text` (not pictured) contains an entry called `..`, which references the same inode.

Accounting for itself (which calls it "."), it's parent (which calls it "report"), and its two subdirectories (which call it ".."), we have found all four links to the inode 592253 reported by the **ls -l** command.

In the following listing, the `report/html` directory has a link count of 3. Can you find all three references to inode number 592255 in the figure above?

```
[prince@station prince]$ ls -ial report/html
```

```
total 20
 592255 drwxrwxr-x    3 prince   prince       4096 Jul 14 13:49 .
 592253 drwxrwxr-x    4 prince   prince       4096 Jul 14 13:27 ..
 592261 -rw-rw-r--    1 prince   prince       2012 Jul 14 13:28 chap1.html
 592262 -rw-rw-r--    1 prince   prince       2012 Jul 14 13:28 chap2.html
 592263 drwxrwxr-x    2 prince   prince       4096 Jul 14 13:28 figures
```

"html" in the directory `report`*, "." in the directory* `report/html`*, and ".." in the directory* `report/html/figures`*.*

In summary, directories are simply collections of dentries for the files the directory is said to contain, which map filenames to inodes. Every directory contains at least two links, one from its own directory entry ".", and one from its parent's entry with the directory's conventional name. Directories are referenced by an additional link for every subdirectory, which refer to the directory as "..".

Device Nodes

We have now discussed three types of "creatures" which can exist in the Linux filesystem, namely regular files, directories, and symbolic links. In this section, we shift gears, and discuss the last two types of filesystem entries (that will be covered in this course), block and character device nodes.

Block and Character Device Nodes

Device nodes exist in the filesystem, but do not contain data in the same way that regular files, or even directories and symbolic links, contain data. Instead, the job of a device node is to act as a conduit to a particular device driver within the kernel. When a user writes to a device node, the device node transfers the information to the appropriate device driver in the kernel. When a user would like to collect information from a particular device, they read from that device's associated device node, just as reading from a file.

By convention, device nodes live within a dedicated directory called `/dev`. In the following, the user elvis takes a long listing of files in the `/dev` directory.

```
[elvis@station elvis]$ ls -l /dev
total 228
crw-------    1 root     root      10,  10 Jan 30 05:24 adbmouse
crw-r--r--    1 root     root      10, 175 Jan 30 05:24 agpgart
crw-------    1 root     root      10,   4 Jan 30 05:24 amigamouse
...
crw-------    1 elvis    root      14,   7 Jan 30 05:24 audioctl
brw-rw----    1 root     disk      29,   0 Jan 30 05:24 aztcd
crw-------    1 elvis    root      10, 128 Jan 30 05:24 beep
brw-rw----    1 root     disk      41,   0 Jan 30 05:24 bpcd
crw-------    1 root     root      68,   0 Jan 30 05:24 capi20
...
```

As there are over 7000 entries in the `/dev` directory, the output has been truncated to only the first several files. Focusing on the first character of each line, most of the files within `/dev` are not regular files or directories, but instead either *character device nodes* ("c"), or *block device nodes* ("b"). The two types of device nodes reflect the fact that device drivers in Linux fall into one of two major classes, character devices and block devices.

Block Devices
: Block devices are devices that read and write information a chunk ("block") at a time. Block devices customarily allow random access, meaning that a block of data could be read from anywhere on the device, in any order. Examples of block devices include hard drives, floppy drives, and CD/ROM drives.

Character Devices

Character devices are often devices that read and write information as a stream of bytes ("characters"), and there is a natural concept of what it means to read or write the "next" character. Examples of character devices include keyboards, mice, sound cards, and printers. Some character device drivers support memory buffers as well.

Under the Hood

The real distinction between character and block device drivers relates to how the device driver interacts with the Linux kernel. block devices ("disks") interact with the the unified I/O cache, while character devices bypass the cache and interact with processes directly.

Terminals as Devices

In the following, elvis has logged onto a Linux machine on both the first and second virtual consoles. In the first workbook, we learned how to identify terminals by name, and found that the name of the first virtual console was `tty1`, and the second virtual console was `tty2`. Now, we can see that the "name" of a terminal is really the name of the device node which maps to that terminal. In the following listing, the device nodes `/dev/tty1` through `/dev/tty6` are the device nodes for the first 6 virtual consoles, respectively.

```
[elvis@station elvis]$ ls -l /dev/tty[1-6]
crw--w----    1 elvis    tty        4,   1 May 14 16:06 /dev/tty1
crw--w----    1 elvis    tty        4,   2 May 14 16:06 /dev/tty2
crw-------    1 root     root       4,   3 May 14 08:50 /dev/tty3
crw-------    1 root     root       4,   4 May 14 08:50 /dev/tty4
crw-------    1 root     root       4,   5 May 14 08:50 /dev/tty5
crw-------    1 root     root       4,   6 May 14 08:50 /dev/tty6
```

In the following, elvis, working from virtual console number 1, will redirect the output of the **cal** command three times; first, to a file called `/tmp/cal`, secondly, to the `/dev/tty1` device node, and lastly, to the `/dev/tty2` device node.

```
[elvis@station elvis]$ cal > /tmp/cal   ❶
[elvis@station elvis]$ cal > /dev/tty1 ❷
      May 2003
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
[elvis@station elvis]$ cal > /dev/tty2 ❸
```

❶ This case should be familiar; the output was merely redirected to a newly created file called `cal`.

❷ From appearances, the redirection didn't happen, but it did. The output of the command was redirected to the device node for the first virtual console, which did what it was "supposed to do", namely, display all information written to it on the screen.

❸ Where did the output of the **cal** command go this time? The information was redirected to the device node for the second virtual console, which did what it was "supposed to do", namely displayed it on the second virtual console.

The second redirection merits a little more attention. When elvis redirected the output to the device node controlling his current virtual console, `/dev/tty1`, the effect was as if he had performed no redirection at all. Why? When elvis runs interactive commands without redirection, they write to the controlling terminal's device node *by default*. Redirecting the command's output to `/dev/tty1` is akin to saying "but instead of writing your output to my terminal, write your output to my terminal."

Upon switching to the second virtual console, using the **CTRL+ALT+F2** key sequence, elvis finds the following characters on the screen.

```
Red Hat Enterprise Linux Server release 5 (Tikanga)
Kernel 2.6.18-8.el5 on an i686

station login: elvis
Password:
Last login: Mon May 14 16:55:22 on tty1
You have new mail.
[elvis@station elvis]$ ❶       ❷May 2003
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31 ❸
```

❶ This is where elvis's cursor was sitting after logging in on the second virtual console, and switching to virtual console number 1.

❷ Here is where the output of the **cal** command was written to the terminal. Note the lack of a linefeed separating the output. This is not a natural, well formatted occurrence, but something odd that elvis asked the device driver to do.

❸ Lastly, the output of the **cal** command tailed off, but notice that the **bash** shell *did not* offer a fresh prompt. In fact, the **bash** shell didn't even realize that the characters were written to the terminal. It's still waiting for elvis to enter a command.

Device Permissions, Security, and the Console User

Continuing the train of thought from above, the user elvis (who has logged onto the first two virtual consoles, `tty1` and `tty2`) next tries to redirect the output of the **cal** command to virtual console number 3, but runs into problems.

```
[elvis@station elvis]$ cal > /dev/tty3
-bash: /dev/tty3: Permission denied
```

Why was elvis not able to perform the same trick on the third virtual console? because elvis has not logged in on the third virtual console, and therefore does not *own* the device. Examine again the long listing of the **ls -l** virtual console device nodes:

```
[elvis@station elvis]$ ls -l /dev/tty[1-6]
crw--w----    1 elvis    tty        4,   1 May 16 13:38 /dev/tty1
crw--w----    1 elvis    tty        4,   2 May 16 13:38 /dev/tty2
crw-------    1 root     root       4,   3 May 16 10:02 /dev/tty3
crw-------    1 root     root       4,   4 May 16 10:02 /dev/tty4
crw-------    1 root     root       4,   5 May 16 10:02 /dev/tty5
crw-------    1 root     root       4,   5 May 16 10:02 /dev/tty6
```

Because device nodes are considered files, they also have user owners, group owners, and a collection of permissions. When reading or writing from device nodes, permissions apply just as if reading or writing to a regular file. This allows a system administrator (or the software on the system) to control who has access to particular devices using a familiar technique; namely, by managing file owners and permissions.

What happens when prince logs in on the third virtual console?

```
[elvis@station elvis]$ ls -l /dev/tty[1-6]
crw--w----    1 elvis    tty        4,   1 May 16 13:38 /dev/tty1
crw--w----    1 elvis    tty        4,   2 May 16 13:38 /dev/tty2
crw--w----    1 prince   tty        4,   3 May 16 13:46 /dev/tty3
crw-------    1 root     root       4,   4 May 16 10:02 /dev/tty4
```

```
crw-------    1 root     root       4,   5 May 16 10:02 /dev/tty5
crw-------    1 root     root       4,   6 May 16 10:02 /dev/tty6
```

When a user logs in, they take ownership of the device node that controls their terminal. Processes that they run are then able to read input or write output to the terminal. In general, the permissions on device nodes do not allow standard users to access devices directly. Two categories of exceptions occur.

Terminals
: Because users need to be able to communicate with the system, they (or, more exactly, the processes that they run) must be able to read from and write to the terminal they are using. Usually, part of the process of logging into a system involves transferring ownership of the terminal's device node to the user.

"Console Users"
: In Red Hat Enterprise Linux, users can be granted special permissions not because of who they are, but because of where they logged in from. Namely, when a user logs into a virtual console, or the graphical X server, they are considered a "console user". Console users are granted access to hardware devices associated with the console, such as the floppy drive and sound card. When the console user logs out of the system, ownerships for these devices are restored to system defaults. None of this happens if a user logs in over the network, for example. (If the user is not sitting at the machine, would it be reasonable for them to use the floppy drive?)

In summary, Linux (and Unix) uses device nodes to allow users access to devices on the system. The managing of devices on a Linux system can be a large and complicated topic. As an introduction, we have examined enough about how terminal device nodes are used to introduce the concept, and identify a couple of advantages to the device node approach.

- When writing programs, programmers do not need to deal with device details. They can treat all input and output as if it they were simply reading or writing to a file.
- Access to devices can be controlled through the same techniques of file ownerships and permissions that are used for regular files.

Examples

Interpreting Directory Links Counts

The user elvis takes a long listing of the `/var/spool` directory. He is interested in interpreting the subdirectory's link counts, as listed in the second column.

```
[elvis@station elvis]$ ls -l /var/spool
total 64
drwxr-xr-x    2 root     root         4096 Jan 24 16:26 anacron
drwx------    3 daemon   daemon       4096 Jun 18 02:00 at
drwxrwx---    2 smmsp    smmsp        4096 Jul 21 10:42 clientmqueue
drwx------    2 root     root         4096 Jun 18 16:12 cron
drwx------    3 lp       sys          8192 Jul 18 17:38 cups
drwxr-xr-x   23 root     root         4096 Jan 24 18:52 lpd
drwxrwxr-x    2 root     mail         4096 Jul 21 10:11 mail
drwx------    2 root     mail         8192 Jul 21 10:43 mqueue
drwxr-xr-x   17 root     root         4096 Feb 24 19:41 postfix
drwxr-xr-x    2 rpm      rpm          4096 Apr 11 06:18 repackage
drwxrwxrwt    2 root     root         4096 Apr  5 23:46 samba
drwxr-xr-x    2 root     root         8192 Jul 16 17:53 up2date
drwxrwxrwt    2 root     root         4096 Feb  3 19:13 vbox
```

Noticing that it has a link count of 17, elvis concludes that the `postfix` directory contains 15 subdirectories. (1 link (shown above) for the `postfix` entry, 1 link for the entry `.` found within `postfix` (not shown), and 15 for the entries `..` within each of 15 subdirectories.)

Examining a long listing of the `/var/spool/postfix` directory, he concludes that he was right (there are 15 subdirectories).

```
[elvis@station elvis]$ ls -l /var/spool/postfix/
total 60
drwx------    2 postfix  root         4096 Feb 24 19:41 active
drwx------    2 postfix  root         4096 Feb 24 19:41 bounce
drwx------    2 postfix  root         4096 Feb 24 19:41 corrupt
drwx------    2 postfix  root         4096 Feb 24 19:41 defer
drwx------    2 postfix  root         4096 Feb 24 19:41 deferred
drwxr-xr-x    2 root     root         4096 Apr  1 12:22 etc
drwx------    2 postfix  root         4096 Feb 24 19:41 flush
drwx------    2 postfix  root         4096 Feb 24 19:41 incoming
drwxr-xr-x    2 root     root         4096 Apr 11 05:54 lib
drwx-wx---    2 postfix  postdrop     4096 Feb 24 19:41 maildrop
drwxr-xr-x    2 root     root         4096 Feb 24 19:41 pid
drwx------    2 postfix  root         4096 Feb 24 19:41 private
drwx--x---    2 postfix  postdrop     4096 Feb 24 19:41 public
drwx------    2 postfix  root         4096 Feb 24 19:41 saved
drwxr-xr-x    3 root     root         4096 Feb 24 19:41 usr
```

Questions

Use the output from the following two commands to answer the next 3 questions.

```
[student@station student]$ tree /etc/sysconfig/networking/
/etc/sysconfig/networking/
|-- devices
|   `-- ifcfg-eth0
|-- ifcfg-lo
`-- profiles
    |-- default
    |   |-- hosts
    |   |-- ifcfg-eth0
    |   |-- network
    |   `-- resolv.conf
    `-- netup
        |-- hosts
        |-- ifcfg-eth0
        |-- network
        `-- resolv.conf

4 directories, 10 files
[student@station student]$ ls -iaR /etc/sysconfig/networking/
/etc/sysconfig/networking/:
  49180 .   244801 ..    65497 devices    49019 ifcfg-lo    65498 profiles

/etc/sysconfig/networking/devices:
  65497 .    49180 ..    73383 ifcfg-eth0

/etc/sysconfig/networking/profiles:
  65498 .    49180 ..    65499 default   558071 netup

/etc/sysconfig/networking/profiles/default:
  65499 .     73386 hosts         73384 network
  65498 ..    73383 ifcfg-eth0    73385 resolv.conf

/etc/sysconfig/networking/profiles/netup:
 558071 .    558076 hosts        558072 network
  65498 ..   558077 ifcfg-eth0   558075 resolv.conf
```

1. What would you expect to be the link count of inode number 65498?

    a. 2

    b. 3

    c. 4

    d. 5

    e. None of the above

2. What would you expect to be the link count of inode number 49180?

    a. 2

    b. 3

    c. 4

    d. 5

    e. None of the above

3. What would you expect to be the link count of inode number 65499?

    a. 2

    b. 3

    c. 4

    d. 5

    e. None of the above

Use the output from the following command to answer the next 4 questions.

```
[elvis@station 030_section_questions]$ ls -l /dev/tty[1-6] /dev/fd0 /dev/audio
crw--w----    1 elvis    tty        4,   1 Jul 22 15:30 /dev/tty1
crw--w----    1 prince   tty        4,   2 Jul 22 15:30 /dev/tty2
crw--w----    1 elvis    tty        4,   3 Jul 22 15:30 /dev/tty3
crw--w----    1 blondie  tty        4,   4 Jul 22 15:30 /dev/tty4
crw-------    1 root     root       4,   5 Jul 22 09:29 /dev/tty5
crw-------    1 root     root       4,   6 Jul 22 09:29 /dev/tty6
brw-rw----    1 prince   floppy     2,   0 Jan 30 05:24 /dev/fd0
crw-------    1 prince   root      14,   4 Jan 30 05:24 /dev/audio
```

4. The user elvis is logged in to which virtual console(s)?

    a. Virtual Console Number 1

    b. Virtual Console Number 2

    c. Virtual Console Number 3

    d. Virtual Console Number 4

    e. Virtual Console Number 5

    f. Virtual Console Number 6

5. Which of the following are block device nodes?

    a. `/dev/tty1`

    b. `/dev/tty2`

    c. `/dev/tty3`

    d. `/dev/tty6`

    e. `/dev/fd0`

    f. `/dev/audio`

6. Which user is currently considered the "Console User"?

    a. elvis

    b. prince

    c. blondie

    d. None of the above

    e. It cannot be determined from the information provided.

7. The user elvis has also logged on using the X graphical environment. He tries to play an audio CD using the **gnome-cd** player. Which of the following best explains why this will not work?

    a. Only users who have logged on using a virtual console may access the audio devices.

    b. The user elvis is not considered the "Console User", and does not have write permissions to the device nodes that connect to the audio device drivers.

    c. The user elvis is not a member of the group "audio", and does not have write permissions to the device nodes that connect to the audio device drivers.

    d. None of the above

Chapter 4. Disks, Filesystems, and Mounting

Key Concepts

- Linux allows low level access to disk drives through device nodes in the `/dev` directory.
- Usually, disks are *formatted* with a filesystem, and *mounted* to a directory instead.
- Filesystems are created with some variant of the **mkfs** command.
- The default filesystem of Red Hat Enterprise Linux is the ext3 filesystem.
- The **mount** command is used to map the root directory of a disk's (or a disk partition's) filesystem to an already existing directory. That directory is then referred to as a *mount point*.
- The **umount** command is used to unmount a filesystem from a mount point.
- The **df** command is used to report filesystem usage, and tables currently mounted devices.

Discussion

Disk Devices

Linux (and Unix) allows users direct, low level access to disk drives through device nodes in the `/dev` directory. The following table lists the filenames of common disk device nodes, and the disks with which they are associated.

Table 4.1. Linux Disk Device Nodes

Device Node	Disk
`/dev/fd0`	Floppy Disk
`/dev/hda`	IDE Primary Master
`/dev/hdb`	IDE Primary Slave
`/dev/hdc`	IDE Secondary Master
`/dev/hdd`	IDE Secondary Slave
`/dev/sd[a-z]`	SCSI Disks
`/dev/cdrom`	Symbolic Link to CD/ROM

Although device nodes exist for disk drives, usually standard users do not have permissions to access them directly. In the following, elvis has performed a long listing of the various device nodes tabled above.

```
[elvis@station elvis]$ ls -l /dev/fd0 /dev/hd[abcd] /dev/sda /dev/cdrom
lrwxrwxrwx    1 root     root            8 Oct  1  2002 /dev/cdrom -> /dev/hdc
brw-rw----    1 elvis    floppy     2,   0 Jan 30 05:24 /dev/fd0
brw-rw----    1 root     disk       3,   0 Jan 30 05:24 /dev/hda
brw-rw----    1 root     disk       3,  64 Jan 30 05:24 /dev/hdb
brw-------    1 elvis    disk      22,   0 Jan 30 05:24 /dev/hdc
brw-rw----    1 root     disk      22,  64 Jan 30 05:24 /dev/hdd
```

```
brw-rw----    1 root     disk       8,   0 Jan 30 05:24 /dev/sda
```

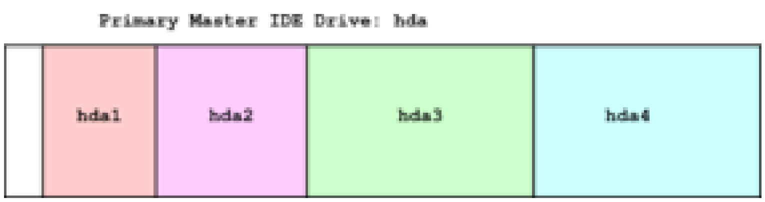
By default, elvis does not have permissions to access the machine's fixed drives. Because he is (apparently) logged on at the console, he is considered the "console user", and has gained permissions to access the floppy and CD/ROM drives. We will take advantage of this fact during this lesson.

Interestingly, the file `/dev/cdrom` is not a device node, but a symbolic link, which resolves to the block device node `/dev/hdc`. Most modern CD/ROM drives physically attach to the machine using either an IDE or SCSI interface, and so appear to the kernel as just another SCSI or IDE drive. Some applications, however, such as the **gnome-cd** audio CD player, want to access "the CD/ROM". For them, `/dev/cdrom` provides access to the CD/ROM, no matter how its attached to the system.

More often than not, hard disks are further divided into *partitions*. Partitions are regions of the hard disk that can each be used as if it were a separate disk. Just as there are device nodes for every disk, there are also device nodes for every disk partition. The name of a partition's device node is simply the partition number appended to the name of the disk's device node. For example, the device node for the third partition of the primary slave IDE drive is called `/dev/hdb3`.

The following diagram illustrates a hard disk that has been divided into four partitions, and the device nodes which address each of the partitions.

Figure 4.1. Hard Disk Partitions



Low Level Access to Drives

In the following, elvis is exploring low level access to his floppy drive. He starts by ensuring he has read and write permissions to the floppy's device node, **cat**ting the `/etc/resolv.conf` file, and then redirecting the output of the command to the floppy drive (`/dev/fd0`).

```
[elvis@station elvis]$ ls -l /dev/fd0
brw-rw----    1 elvis    floppy     2,   0 Jan 30 05:24 /dev/fd0
[elvis@station elvis]$ cat /etc/resolv.conf
search example.com
nameserver 192.168.0.254
[elvis@station elvis]$ cat /etc/resolv.conf > /dev/fd0
-bash: /dev/fd0: No such device or address
```

At first perplexed by the error message, elvis realizes that there is no floppy disk in his floppy disk drive. He places an old, unused floppy (that he doesn't care about the contents of) into the drive.

```
[elvis@station elvis]$ cat /etc/resolv.conf > /dev/fd0
-bash: /dev/fd0: Read-only file system
```

Perplexed again, elvis removes the floppy from the drive, examines it, slides the write protection tab on the floppy disk to the writable position, and reinserts the floppy.

```
[elvis@station elvis]$ cat /etc/resolv.conf > /dev/fd0
```

Finally, the floppy drive's light comes on, and elvis hears the disk spin as information is written to the floppy. Curious to see what he has written to the floppy disk, elvis next tries to read the floppy with the **less** pager.

```
[elvis@station elvis]$ less /dev/fd0
/dev/fd0 is not a regular file (use -f to see it)
```

The **less** pager seems to be telling elvis, "Sane people don't read from device nodes directly, but if you really want to do it, I'll let you." Because elvis really wants to do it, he adds the **-f** command line switch.

```
[elvis@station elvis]$ less -f /dev/fd0
```

On the first page of the pager, elvis recognizes the first few characters as the contents of `/etc/resolv.conf`. After that, however, the pager shows unintelligible data, with occasional human readable text interspersed.

```
search example.com
nameserver 192.168.0.254
B^RA^^@^@V^^|F^@^@^@AdDBP^A^^|^C^F^B|^@DVP|Q^^^R ABAoot failed^@^@^@LDLINUX SYS
...
```

The user elvis continues to page through the "file" using the **less** pager, seeing apparently random snippets of text and binary characters. When he feels like he's gotten the point, he quits the pager.

What is the point? By accessing disk drives through their device nodes, users may see (and write) the contents of the drive *byte for byte*. To the user, the drive looks like a (very big) file. When elvis **cat**s the contents of a file to the drive's device node, the information is transferred, byte for byte, to the drive.

On the first few bytes of the floppy, elvis sees a copy of the contents of the `/etc/resolv.conf` file. On the floppy, what is the filename associated with the information? Trick question. *It doesn't have one.* Who is the user owner? What are the permissions? *There are none.* It's *just data*. When elvis goes back to read the contents of the drive, he sees the data he wrote there, namely the contents of the `/etc/resolv.conf` file. After that, he sees whatever happened to be on the floppy before he started.

Filesystems

The previous section demonstrated how to access drives at a low level. Obviously, people do not like to store their information on drives as one stream of data. They like to store their information in files, and give the files filenames. They like to organize their files into directories, and say who can access the directory and who cannot. All of this structuring of information is the responsibility of what is called a *filesystem*.

A *filesystem* provides order to disk drives by organizing the drive into fixed sized chunks called blocks. The filesystem then organizes these blocks, effectively saying "this block is going to contain only inodes", "this block is going to contain only dentries", "these 3 block over here, and that one over there, are going to contain the contents of the file `/etc/services`", or "this first block is going to store information which keeps track of what all the other blocks are being used for". Filesystems provide all of this structure that is usually taken for granted.

Before a disk can be used to store files in a conventional sense, it must be initialized with this type of low level structure. In Linux, this is usually referred to as "creating a filesystem". In other operating systems, it is usually referred to as "formatting the disk". Linux supports a large number of different types of filesystems (the fs(5) man page lists just a few). While Linux's native filesystem is the ext2 (or in Red Hat Enterprise Linux, the ext3) filesystem, it also supports the native filesystems of many other operating systems, such as the DOS FAT filesystem, or the OS/2 High Performance File System.

In Linux, filesystems are created with some variant of the **mkfs** command. Because these commands are usually used only by the administrative user, they do not live in the standard `/bin` or `/usr/bin` directories, and therefore cannot be invoked as simple commands. Instead, they live in the `/sbin` directory, which is reserved for administrative commands. In the following, elvis lists all commands that begin `mkfs` in the `/sbin` directory.

```
[elvis@station elvis]$ ls /sbin/mkfs*
/sbin/mkfs         /sbin/mkfs.ext2  /sbin/mkfs.msdos
/sbin/mkfs.cramfs  /sbin/mkfs.ext3  /sbin/mkfs.vfat
```

Apparently, there is one copy of the **mkfs** command for each type of filesystem that can be constructed, including the ext2 and msdos filesystems. The user elvis next formats the same floppy he used above with the ext2 filesystem. Because the **mkfs.ext2** command did not live in one of the "standard" directories, elvis needs to refer to the command using an absolute reference, `/sbin/mkfs.ext2`.

```
[elvis@station elvis]$ /sbin/mkfs.ext2 /dev/fd0
mke2fs 1.39 (29-May-2006)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
4096 inodes, 4096 blocks
204 blocks (4.98%) reserved for the super user
First data block=0
1 block group
32768 blocks per group, 32768 fragments per group
4096 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 20 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

The `mkfs.ext2` command displays information about the filesystem as it creates it on the device `/dev/fd0`. When the command completes, the filesystem has been initialized, and the floppy is ready to be used.

The `mkfs` command, and its variants, can be configured with a large collection of command line switches which specify low level details about the filesystem. These details are beyond the scope of this course, however. Fortunately, the various options default to very reasonable general purpose defaults. For the curious, more information can be found in the mkfs.ext2(8) and similar man pages.

Mounting Filesystems

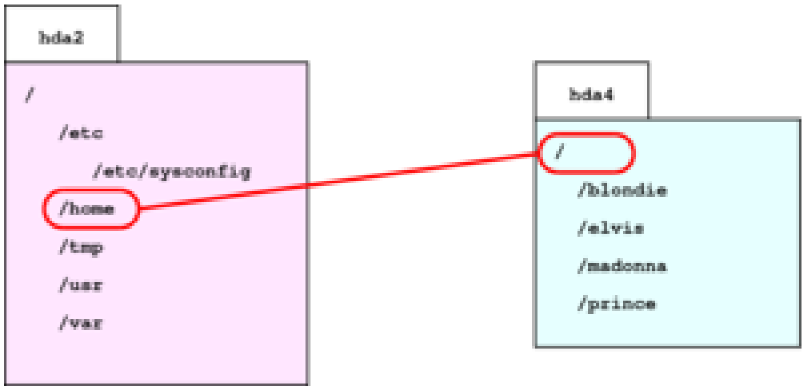
Once a disk or a disk partition has been formatted with a filesystem, users need some way to access the directories and files that the filesystem provides. In other operating systems, users are usually very aware of disk partitions, because they have to refer to them using labels such as *C:* or *D:*. In Unix, users are often unaware of partitions, because different disk partitions are organized into a single directory structure.

How is this done? Every filesystem provides a root directory that serves as the base of that filesystem. Upon booting the system, one of your disk's partitions acts as the *root partition*, and its root directory becomes the system's root directory `/`. Sometimes, that's the end of the story. The `/` directory has subdirectories, and those subdirectories have subdirectories, all of which reside in the root partition's filesystem.

If a system has multiple disks, however, or if a disk has multiple partitions, the story gets more complicated. In order to access the filesystems on the other partitions, the root directories of those filesystems are mapped to an already existing directory through a standard Unix technique called *mounting*.

In the example diagrammed below, the filesystem on the partition `/dev/hda2` is being used as the root partition, and contains the `/etc`, `/tmp`, `/home`, and other expected directories. The `/dev/hda4` partition has also been formatted with a filesystem, and its root directory contains the directories `/blondie`, `/prince`, and others. In order to make use of this filesystem, it is mounted to the `/home` directory, which already existed in the root partition's filesystem. This mount usually happens as a normal part of the system's boot up process.

Figure 4.2. Mounting a Filesystem



Now, all references to the `/home` directory are transparently mapped to the root directory of the filesystem on `/dev/hda4`, giving the appearance that the `/home` directory contains the subdirectories `blondie`, `elvis`, etc., as seen below. When a filesystem is mounted over a directory in this manner, that directory is referred to as a *mount point*.

```
[elvis@station elvis]$ ls /home
blondie   elvis     madonna   prince
```

How can elvis tell which partition contains a given file? Just using the **ls** command, he can't! To the **ls** command, and usually in a user's mind, all of the different partitions are gracefully combined into a single, seamless directory structure.

Viewing Mount Points

How can a user determine which directories are being used as mount points? One approach is to run the **mount** command without arguments.

```
[elvis@station elvis]$ mount
/dev/hda3 on / type ext3 (rw)
none on /proc type proc (rw)
usbdevfs on /proc/bus/usb type usbdevfs (rw)
/dev/hda1 on /boot type ext3 (rw)
none on /dev/pts type devpts (rw,gid=5,mode=620)
none on /dev/shm type tmpfs (rw)
```

Without arguments, the **mount** command returns a list of current mount points, the device that is mounted to it, the type of filesystem that device has been formatted with, and any mount options associated with the mount. In the above example, the `/dev/hda3` partition is being used as the root partition, and the ext3 filesystem on partition `/dev/hda1` has been mounted to the directory `/boot`. Note that several of the filesystems listed above are said to be on the device "none". These are *virtual filesystems*, which are implemented by the kernel directly, and do not exist on any physical device.

Why Bother?

If you seldom know which directories are being used as mount points, and which files exist in which partitions, why bother even talking about it? For now, we will address two reasons. The first reason is that there can be subtle issues that creep up which are related to the underlying filesystem. Partitions can run out of space. If a filesystem mounted on `/home` runs out of space, no more files can be created underneath the `/home` directory. This has no effect on the `/tmp` directory, however, because it belongs to another filesystem. In Unix, when a partition fills up, it only effects the part of the directory structure underneath its mount point, not the entire directory tree.

Users can determine how much space is available on a partition with the **df** command, which stands for "**d**isk **f**ree".

`df` [OPTION...] [FILE...]

Show information about all partitions, or partition on which FILE resides.

Switch	Effect
-a, --all	Show all filesystems, including those of size 0
-h, --human-readable	Print sizes in human readable format
-i, --inodes	List inode usage instead of block usage
-T, --print-type	Include filesystem type

Not only will the **df** command show how much space is left on particular partitions, but it also gives a very readable table of which devices are mounted to which directories. In the following, the filesystem on `/dev/hda2` is being used as the root partition, the filesystem on `/dev/hda1` is mounted to the `/boot` directory, and `/dev/hda4` is mounted to `/home`. A partition on a second disk drive, namely the `/dev/hdb2` partition, is mounted to a non standard directory named `/data`.

```
[elvis@station elvis]$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/hda2              8259708   6708536   1131592  86% /
/dev/hda1               102454     24227     72937  25% /boot
/dev/hda4              5491668    348768   4863936   7% /home
/dev/hdb2              4226564   1417112   2594748  36% /data
none                    127592         0    127592   0% /dev/shm
```

Mounting Temporary Media: The /media directory.

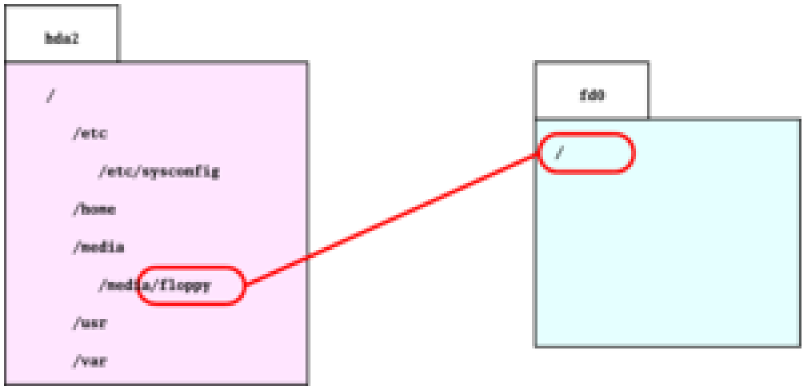
The second reason users need to be aware of filesystems and mount points involves temporary media such as floppies and CD/ROM drives. Like any block device, floppy disks and CD/ROM disks are formatted with filesystems. In order to access these filesystems, they must be mounted into the directory structure, using a (already existing) directory as a mount point. Which directory should be used?

The `/media` directory contains subdirectories such as `/media/floppy` and `/media/cdrom`, or even `/media/camera`, which are intended for just this purpose; they serve as mount points for temporary media. (Thus the name of the `/media` directory.) If you would like to make use of a temporary disk, such as a floppy disk, you must first mount the filesystem into your directory structure, using the **mount** command.

```
[elvis@station elvis]$ mount /media/floppy
[elvis@station elvis]$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/hda2              8259708   6708536   1131592  86% /
/dev/hda1               102454     24227     72937  25% /boot
/dev/hda4              5491668    348768   4863936   7% /home
/dev/hdb2              4226564   1417112   2594748  36% /data
none                    127592         0    127592   0% /dev/shm
/dev/fd0                  1412        13      1327   1% /media/floppy
```

On the last line, the **df** command now reports the newly mounted floppy drive, and elvis can copy files onto the floppy.

Figure 4.3. Mounting a Formatted Floppy




```
[elvis@station elvis]$ cp /etc/services /media/floppy/
[elvis@station elvis]$ ls /media/floppy/
lost+found  services
```

Where did the directory `lost+found` come from? This directory was created when the filesystem was created, and always exists in the root directory of an ext2 or ext3 filesystem. It is used occasionally when repairing damaged filesystems.

When elvis has finished using the floppy, he detaches it from the filesystem using the **umount** command.

```
[elvis@station elvis]$ umount /media/floppy/
[elvis@station elvis]$ ls /media/floppy/
```

Once the floppy disk's filesystem has been detached from the `/media/floppy` directory, the directory is just an empty directory.

Mounting Issues

Mounting devices is one of the more awkward and problematic issues for new Linux (and Unix) users. The following issues can also occur, which serve to complicate the matter.

Permissions
: By default, only the root user can mount and unmount devices. Temporary media are handled differently, however. The "Console User" (someone who has logged in from a virtual console or the X login screen) gains ownership of devices associated with the physical machine, such as a floppy drive, and special permissions to mount these devices to predefined mount points, such as `/media/floppy`. If a user has logged in by some other technique, such as over the network, or via the **su** command, they will not be considered the "Console User", and will not have permissions to mount these devices.

Busy Filesystems
: A filesystem can only be unmounted if it is considered "non-busy". What can keep a filesystem "busy"? Any open file, or any process that has a current working directory in the filesystem, "busy"'s the filesystem. The only way for the filesystem to be unmounted is to track down any processes that might be keeping the filesystem "busy", and kill them.

Automounters
: The GNOME graphical environment runs an *automounter*, which keeps an eye on the CD/ROM drive, and will automatically mount

the filesystem of any newly inserted disk. The automounter is part of the graphical environment, and does not exist if a user logged in through a virtual console. Also, the automounter only works for the CD/ROM drive. The floppy drive, or other devices, must be mounted "manually".

Kernel Buffering

In order to improve performance, the kernel buffers all block device (harddrive) interactions. For example, when you copy a file to a floppy, the file might seem to have been copied almost immediately. Later, when you unmount the floppy with the **umount** command, the command takes a while to return while writes are being committed to the floppy. When unmounting the device, the kernel is forced to commit all pending transactions to the disk.

What would happen if you removed the floppy from the drive before buffered writes were committed to disk? At best, the files that you thought you had copied to the floppy would not be there. At worst, you may have a corrupted floppy, and a confused Linux kernel the next time someone tries to mount a floppy.

The upshot: Not only must you mount temporary media (such as floppies) before you can use them, you must also unmount the media when you are done.

Examples

Using an Unformatted Floppy

The user madonna has a collection of songs which she would like to copy to an unformatted floppy to share with friends. Because she knows that all of her friends use Red Hat Enterprise Linux, she decides to format the floppy with the ext2 filesystem.

```
[madonna@station madonna]$ /sbin/mkfs.ext2 /dev/fd0
mke2fs 1.32 (09-Nov-2002)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
184 inodes, 1440 blocks
72 blocks (5.00%) reserved for the super user
First data block=1
1 block group
8192 blocks per group, 8192 fragments per group
184 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 21 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

Next, she mounts the floppy, and copies her files over to it.

```
[madonna@station madonna]$ mount /media/floppy/
[madonna@station madonna]$ cp song* /media/floppy/
[madonna@station madonna]$ cd /media/floppy/
[madonna@station floppy]$ ls
```

```
lost+found  song02.ogg  song04.ogg  song06.ogg
song01.ogg  song03.ogg  song05.ogg  song07.ogg
```

She then unmounts her floppy.

```
[madonna@station madonna]$ umount /media/floppy/
umount: /media/floppy: device is busy
```

Why would the floppy not unmount? Some process either has an open file, or a current working directory within the floppy's filesystem. The offending process is madonna's **bash** shell, whose current working directory is `/media/floppy`. In order to unmount the floppy, madonna must **cd** to somewhere else, such as her home directory.

```
[madonna@station floppy]$ cd
[madonna@station madonna]$ umount /media/floppy/
```

She can now remove the floppy from the drive.

Using a DOS Formatted Floppy

Eventually, madonna comes across a friend that insists he can only use DOS formatted floppies. Madonna formats another floppy, this time with the MS-DOS filesystem, mounts the floppy, and copies her files over to it.

```
[madonna@station madonna]$ /sbin/mkfs.msdos /dev/fd0
mkfs.msdos 2.8 (28 Feb 2001)
[madonna@station madonna]$ mount /media/floppy/
[madonna@station madonna]$ cp song0* /media/floppy/
```

After she has copied the files to the floppy, she decides that she would like to create a soft link to identify her favorite song for her friend.

```
[madonna@station madonna]$ cd /media/floppy/
[madonna@station floppy]$ ls
song01.ogg  song03.ogg  song05.ogg  song07.ogg
song02.ogg  song04.ogg  song06.ogg
[madonna@station floppy]$ ln -s  song06.ogg my_favorite_song.ogg
ln: creating symbolic link `my_favorite_song.ogg' to `song06.ogg': Operation not
 permitted
```

Why could madonna not create the link? Although Linux supports the MS-DOS filesystem, the MS-DOS filesystem is much simpler than traditional Linux filesystems, and Linux needs to make some compromises. One of these compromises is that the MS-DOS filesystem does not support soft (or hard) links. Neither does the MS-DOS filesystem support file owners and file permissions. How does Linux handle this? It treats all files as owned by the same user, and all permissions as 755. What happens when madonna tries to change permissions on one of the files?

```
[madonna@station floppy]$ chmod 664 song05.ogg
chmod: changing permissions of `song05.ogg' (requested: 0664, actual: 0644): Ope
ration not permitted
```

Again, the operation not permitted. Different filesystems provide different capabilities, and the MS-DOS filesystem is not as featured as the ext2 filesystem.

Now that she is finished, madonna **cd**'s to her home directory, and unmounts the floppy.

```
[madonna@station floppy]$ cd
[madonna@station madonna]$ umount /media/floppy/
```

Why did she **cd** to her home directory first?

*Her **bash** shell's current working directory was inside the floppy's filesystem, which would have prevented her from unmounting the floppy.*

Floppy Images

Several friends have asked madonna for copies of the same floppy. After a while, she grows tired of formatting floppies, mounting them, copying the same 8 songs over to it, and unmounting. She decides to speed up the process by creating an image file for her floppy.

She first prepares a floppy with the files that she wants to distribute.

```
[madonna@station madonna]$ /sbin/mkfs.ext2 /dev/fd0 > /dev/null
mke2fs 1.32 (09-Nov-2002)
[madonna@station madonna]$ mount /media/floppy/
[madonna@station madonna]$ cp song* /media/floppy/
[madonna@station madonna]$ ls /media/floppy/
lost+found  song02.ogg  song04.ogg  song06.ogg
song01.ogg  song03.ogg  song05.ogg  song07.ogg
[madonna@station madonna]$ umount /media/floppy/
```

Next, she copies the contents of the *unmounted* floppy, byte for byte, into a file called `songs.img`.

```
[madonna@station madonna]$ cat /dev/fd0 > songs.img
```

The **cat** command, that we have been using on to view simple text files, works just as well on binary files, or, in this case, binary disk data. After a few seconds of the floppy drive spinning, she has a new file. How big is the file?

```
[madonna@station madonna]$ ls -s songs.img
1444 songs.img
```

1.4 MBytes, exactly what you would expect from a floppy. She stores this image file, and whenever someone new wants a copy of her floppy, she reverses the process by **cat**ing the image back down to an unformatted floppy.

```
[madonna@station madonna]$ cat songs.img > /dev/fd0
```

Because the image file is transferred, byte for byte, to the new floppy, this command has the effect of formatting the floppy with an ext2 filesystem, and copying the files to it, all in one step. This is a powerful technique known as *imaging* drives, and works on any type of disk, not just floppies.

When accessing devices at a low level, it is important that the device is unmounted. If madonna had performed to last command on a mounted floppy, the kernel would have almost certainly been confused, and corrupted the floppy.

Online Exercises

Using Floppies

Lab Exercise

Objective: Format, mount, and umount a floppy drive

Estimated Time: 15 mins.

Setup

You will need a floppy for this exercise. The contents of the floppy will be destroyed.

Specification

In this lab exercise, you will format a floppy disk, mount it, copy files to it, and then unmount the floppy.

1. Make sure that the floppy disk is not write protected, and place it in the floppy drive.
2. Format the floppy with the ext2 filesystem, using the **/sbin/mkfs.ext2** command.
3. Mount the floppy onto the `/media/floppy` directory, using the **mount** command.
4. Recursively copy the contents of the `/etc/sysconfig` directory onto your floppy. (Ignore any errors that you do not have permissions to read some files.)
5. Unmount your floppy. Swap floppies with a neighbor at this point, if possible.
6. Mount your neighbor's floppy (or remount your own), again to the `/media/floppy` directory.
7. With the floppy still mounted, capture the output of the **df** command into the file `df.floppy` in your home directory.

Deliverables

1.
    1. A floppy formatted with the ext2 filesystem mounted to the `/media/floppy` directory.
    2. A directory `/media/floppy/sysconfig`, which is a recursive copy of the `/etc/sysconfig` directory (perhaps with a few inaccessible files omitted).
    3. A file in your home directory called `df.floppy`, which contains the output of the `df` command.

Possible Solution

The following sequence of commands provides one possible solution to this exercise.

```
[student@station student]$ /sbin/mkfs.ext2 /dev/fd0
mke2fs 1.32 (09-Nov-2002)
Filesystem label=
OS type: Linux
...
[student@station student]$ mount /media/floppy/
[student@station student]$ cp -r /etc/sysconfig /media/floppy/
cp: cannot open `/etc/sysconfig/rhn/up2date' for reading: Permission denied
cp: cannot open `/etc/sysconfig/rhn/systemid' for reading: Permission denied
...
[student@station student]$ df > df.floppy
[student@station student]$ umount /media/floppy/
```

Cleaning Up

You may unmount your floppy after you have been graded. If you are proceeding to the next exercise, save the contents of your floppy.

Imaging a Floppy

Lab Exercise

Objective: Create an image of a floppy disk drive

Estimated Time: 15 mins.

Setup

You will need the floppy you created in the previous exercise, namely, an ext2 formatted floppy containing a recursive copy of the `/etc/sysconfig` directory.

Specification

In this lab exercise, you will create an image file of a floppy, and then restore the floppy using the image file.

1. Make sure that the floppy disk is not write protected, and place it in the floppy drive. Make sure that the floppy is *not* mounted. Unmount the floppy with the **umount** command, if necessary.
2. Using the **cat** command, make an image file of the floppy in your home directory, called `floppy.img`.
3. Using the **ls -s** command, confirm that the size of your image file is 1444 Kbytes.
4. Using the **file** command, make sure the file is being identified as an ext2 filesystem.
5. With your floppy still in the drive, and still unmounted, reformat the floppy with the msdos filesystem (using the **/sbin/mkfs.msdos** command).
6. Mount your floppy to the `/media/floppy` directory, and, using the **mount** command without arguments, confirm that the floppy has been formatted with the msdos (vfat) filesystem.
7. Unmount the floppy. Using the **cat** command, restore your floppy from the image file.
8. Mount your floppy, again to the `/media/floppy` directory. Using the **mount** command, without arguments, confirm that the original ext2 filesystem has been restored.

Deliverables

1.
    1. A floppy formatted with the ext2 filesystem mounted to the `/media/floppy` directory.
    2. An image of the same floppy, stored in the file `floppy.img` in your home directory.

Possible Solution

The following commands provide one solution to the first four steps of this exercise.

```
[student@station student]$ cat /dev/fd0 > floppy.img
[student@station student]$ file floppy.img
floppy.img: Linux rev 1.0 ext2 filesystem data
[student@station student]$ ls -s
total 1448
   4 df.floppy  1444 floppy.img
```

Cleaning Up

Because the image file you created from this exercise is fairly large, you may want to remove it after your exercise has been graded.

Questions

Some questions may use the following information:

```
[student@station student]$ mount
/dev/hda9 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/hda2 on /boot type ext3 (rw)
/dev/hda7 on /home type ext3 (rw)
none on /dev/shm type tmpfs (rw)
/dev/hda8 on /tmp type ext3 (rw)
/dev/hda5 on /usr type ext3 (rw)
/dev/hda6 on /var 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)
```

1. Which command or commands will show what filesystems are currently mounted?

    a. **chown**

    b. **df**

    c. **mount**

    d. **ls**

    e. **mkdir**

2. According to the screen output above, which device contains the `/home` filesystem?

    a. /dev/pts

    b. /dev/hda1

    c. /dev/hda5

    d. /dev/hda7

    e. None of the above

3. According to the screen output above, where is the device `/dev/hda6` mounted?

    a. /dev/pts

    b. /home

    c. /var

    d. /usr

    e. None of the above

4. When unmounting a device, the error message "umount: /media/floppy: device is busy" can mean what?

    a. An application was started from the directory `/media/floppy` and is still running.

    b. The current working directory for some shell is `/media/floppy`

    c. The floppy is read only.

    d. You do not have permission to use the **umount** command.

e. An application has the file `/media/floppy/make.log` open for writing.

5. Why is it important to unmount removable media before physically removing it?

   a. If you do not, the system will shut down.

   b. So that the next disk inserted can be read and mounted without error.

   c. So that the disk can be read and mounted on another system without error.

   d. Most systems cache data written to a device. The **umount** ensures that this data gets written from the cache to the device so that it is not lost.

6. Which of the following commands can be used to format an unformatted floppy?

   a. `/sbin/mkfs.ext2`

   b. `/sbin/mkfsys.msdos`

   c. `format`

   d. `floppyfd`

   e. `mount`

The user elvis tries to unmount the floppy, and receives the following error message.

```
[elvis@station elvis]$ umount /media/floppy/
umount: only student can unmount /dev/fd0 from /media/floppy
```

7. What is the most reasonable explanation for the message?

   a. The user student write protected the floppy.

   b. The user student mounted the floppy, and therefore only that user may umount the floppy.

   c. The user elvis does not have permissions to run the **umount** command.

   d. The user student formatted the floppy, so only the user student may mount and unmount it.

8. Which of the following commands would create an ext2 filesystem on the third partition of the primary master IDE drive?

   a. **`/sbin/mkfs.ext2 /dev/hda`**

   b. **`/sbin/mkfs.ext2 /dev/fd0`**

   c. **`/sbin/mkfs.msdos /dev/hda`**

   d. **`/sbin/mkfs.msdos /dev/hda3`**

   e. None of the above

9. Which of the following commands would create an ext2 filesystem on the second partition of a SCSI disk?

   a. **`/sbin/mkfs.ext2 /dev/sda2`**

   b. **`/sbin/mkfs.ext2 /dev/hda2`**

c. **`/sbin/mkfs.msdos /dev/sda2`**

d. **`/sbin/mkfs.msdos /dev/hda2`**

e. None of the above

When logged into the X graphical environment, using GNOME, elvis tries to mount a newly inserted CD/ROM.

```
[elvis@station elvis]$ mount /media/cdrom
mount: according to mtab, /dev/cdrom is already mounted on /media/cdrom
mount failed
```

10. What is the most reasonable explanation for the message?

    a. Another user, logged in over the network, mounted the CD/ROM without elvis realizing it.

    b. The GNOME automounter automatically mounted the CD/ROM.

    c. The `/etc/mtab` file is out of sync, and the CD/ROM is really not currently mounted.

    d. None of the above.

Chapter 5. Locating Files with locate and find

Key Concepts

- The **locate** command uses a database to quickly locate files on the system by filename.
- The **find** command performs a real time, recursive search of the filesystem.
- The **find** command can search for files based on inode information.
- The **find** command can perform arbitrary commands on files.

Discussion

Locating Files

It is common to find config files in /etc or executables in a bin directory, however, sometimes it is necessary to search the system for a specific file. Two of the common tools for this are **locate** and **find**.

The command **locate** prints the names of files and directories that match a supplied pattern. It is the faster of the two commands because it relies on a database (updated daily by default) instead of searching real time. The downside of this is that it will not find files created today or it will find files that have been deleted since the last update of the database.

The command **find** can find files by name but can also search for files by owner, group, type, modification date, and many other criteria. With its real time search through the directory tree, it is slower than **locate** but it is also more flexible.

Using Locate

The **locate** command quickly reports all files on the disk whose filename contains the specified text. The search relies on a database, which is updated nightly, so recently created files will probably not be reported. The database does remember file permissions, however, so you will only see files which you would normally have permissions to see.

Earlier we used the command **umount** to unlink a filesystem from the directory tree. Lets see what files on the system include the string "umount" in their names.

```
[blondie@station blondie]$ locate umount
/bin/umount
/sbin/umount.cifs
/sbin/umount.nfs
/sbin/umount.nfs4
/usr/bin/gnome-umount
/usr/share/doc/samba-3.0.23c/htmldocs/manpages/smbumount.8.html
/usr/share/doc/samba-3.0.23c/htmldocs/manpages/umount.cifs.8.html
/usr/share/man/man2/umount.2.gz
/usr/share/man/man2/umount2.2.gz
/usr/share/man/man8/umount.8.gz
/usr/share/man/man8/umount.cifs.8.gz
/usr/share/man/man8/umount.nfs.8.gz
```

Notice that in addition to `/bin/umount` we also locate variants for special network related filesystems, several man page files.

The **locate** command also supports "file globs", or, more formally, pathname expansion, using the same ***, *?*, and *[...]* expressions as the **bash** shell. For example, if you knew that there was a PNG image of a fish somewhere on the system, you might try the following locate command.

For reasons discussed in a later workbook, it's a better idea to wrap any "globs" in quotes, though you can often get away without them.

```
[blondie@station ~]$ locate "*fish*.png"
/usr/share/backgrounds/tiles/fish.png
/usr/share/gnome/help/fish/C/figures/fish_applet.png
/usr/share/gnome/help/fish/es/figures/fish_applet.png
...
/usr/share/gnome/help/fish-applet-2/de/figures/fish_applet.png
/usr/share/gnome/help/fish-applet-2/de/figures/fish_settings.png
/usr/share/gnome/help/fish-applet-2/ja/figures/fish_applet.png
...
/usr/share/gnome/panel/pixmaps/fishanim.png
/usr/share/icons/hicolor/16x16/apps/gnome-panel-fish.png
...
```

Using find

The **find** command is used to search the filesystem for files that meet a specified criteria. Almost any aspect of a file can be specified, such as its name, its size, the last time it was modified, even its link count. (The only exception is the file's content. For that, we need to wait for a command called **grep**, which complements **find** nicely.)

The **find** command's syntax takes a little getting accustomed to, but once learned, is very usable. A **find** command essentially consists of three parts: a root directory (or directories), a search criteria, and an action.

find (root directory) (criteria) (action)

The default directory is `"."`, the default criteria is "every file", and the default action is "print" (the filename), so running the **find** command without arguments will simply descend the current directory, printing every filename. If given a directory name as a single argument, the same would be done for that directory.

```
[madonna@station madonna]$ find /etc/sysconfig/networking/
/etc/sysconfig/networking/
/etc/sysconfig/networking/devices
/etc/sysconfig/networking/devices/ifcfg-eth0
/etc/sysconfig/networking/profiles
/etc/sysconfig/networking/profiles/default
/etc/sysconfig/networking/profiles/default/network
/etc/sysconfig/networking/profiles/default/resolv.conf
/etc/sysconfig/networking/profiles/default/hosts
/etc/sysconfig/networking/profiles/default/ifcfg-eth0
/etc/sysconfig/networking/profiles/netup
/etc/sysconfig/networking/profiles/netup/network
/etc/sysconfig/networking/profiles/netup/resolv.conf
/etc/sysconfig/networking/profiles/netup/hosts
/etc/sysconfig/networking/profiles/netup/ifcfg-eth0
/etc/sysconfig/networking/ifcfg-lo
```

Usually, however, the **find** command is given criteria to refine its search, in the form of (non standard) command line switches. For example, the **-name** command line switch is used to find files with a given name. As with **locate**, globs are supported, but should be quoted.

```
[madonna@station madonna]$ find /etc -name "*.conf"
```

```
/etc/gdm/securitytokens.conf
/etc/gdm/custom.conf
/etc/gssapi_mech.conf
...
/etc/updatedb.conf
/etc/reader.conf
/etc/selinux/restorecond.conf
find: /etc/selinux/targeted/modules/active: Permission denied
find: /etc/selinux/targeted/modules/previous: Permission denied
/etc/selinux/targeted/setrans.conf
/etc/selinux/semanage.conf
...
```

While superficially similar to the **locate** command, **find** functions by performing a search *in real time*. This can take a lot longer, but avoids the issue of an "out of sync" database. Note that if the proper ordering is not followed, **find** becomes quickly confused.

```
[madonna@station madonna]$ find -name "*.conf" /etc
find: paths must precede expression
Usage: find [path...] [expression]
```

Find Criteria

If you browse the find(1) man page, you will discover that an overwhelming selection of criteria can be specified for your search. Almost any aspect of the file that can be reported by the **stat** command or the **ls** command is fair game. The following table summarizes some of the more common search criteria.

Table 5.1. Search Criteria for the find Command

switch	specification
-empty	The file is a directory or regular file, and is empty.
-group *gname*	The file is group owned by *gname*.
-inum *n*	The file has an inode number *n*.
-links *n*	The file has *n* links.
-mmin *n*	The file was last modified *n* minutes ago.
-mtime *n*	The file was last modified *n* days ago.
-name *pattern*	The file's name matches the file glob *pattern*.
-newer *filename*	The file was modified more recently than *filename*.
-perm *mode*	The file's permissions are exactly *mode*.
-perm -*mode*	All of the permission bits *mode* are set for the file.
-perm +*mode*	Any of the permission bits *mode* are set for the file.
-size *n*	The file has a size of *n*.
-type *c*	The file is of type *c*, where *c* is "f" (regular file), "d" (directory), or "l" (symbolic link). See the man page for more details.
-user *uname*	File is owned by the user *uname*.

More options are available, but these should give you an idea of the flexibility of the **find** command. Any criteria that takes a numeric argument, such as **-size** or **-mtime**, recognizes arguments of the form +3 (meaning more than 3), -3 (meaning less than 3), or 3 (meaning exactly 3).

If multiple criteria are specified, by default, all criteria must be met. If multiple criteria are separated by **-or**, however, either condition may be met. Criteria can be inverted by preceding the criteria with **-not**.

As an example, the following command finds all files under `/var` which are not group writable.

```
[elvis@station elvis]$ find /var -not -perm +20
/var
/var/lib
/var/lib/rpm
/var/lib/rpm/Packages
/var/lib/rpm/Basenames
/var/lib/rpm/Name
/var/lib/rpm/Group
...
```

Find Actions

You can also specify what you would like done to files that meet the specified criteria. By default, if no criteria is specified, the file name is printed to standard out, one file per line. Other options are summarized in the following table.

Table 5.2. Action Specifications for the find Command

Switch	Action
-exec *command* ;	Execute *command* on matching files. Use *{}* to indicate where filename should be substituted.
-ok *command* ;	Like -exec, but prompt for each file
-ls	Print file in **ls -dils** format.

Again, others exist. Consult the find(1) man page.

Perhaps the most useful, and definitely the most awkward, of these is **-exec**, and its close cousin **-ok**. The **-exec** mechanism is powerful: rather than printing the names of matching files, arbitrary commands can be run. The **-exec** mechanism is awkward, because the syntax for specifying the command to run is tricky. The command should be written after the **-exec** switch, using a literal *{}* as a placeholder for the file name. The command should be terminated with a literal *;*, but as will be seen a later Workbook, the *;* has special significance to the shell, and so must be "escaped" by prepending a \. An example will help clarify the syntax.

Suppose madonna wanted to make a copy of every file greater than 200 Kbytes in size out of the `/etc` directory. First, she finds which files meet the criteria.

```
[madonna@station madonna]$ find /etc -size +200k 2>/dev/null
/etc/selinux/targeted/policy/policy.21
/etc/firmware/microcode.dat
/etc/prelink.cache
/etc/termcap
/etc/pki/tls/certs/ca-bundle.crt
/etc/gconf/gconf.xml.defaults/%gconf-tree-sr@Latn.xml
/etc/gconf/gconf.xml.defaults/%gconf-tree-cs.xml
...
```

(The **2>/dev/null** serves to "throw away" complaints about directories madonna does not have permissions to access.)

To confirm the sizes of the files, she reruns the command, specifying the "action" of **-ls**.

```
[madonna@station madonna]$ find /etc -size +200k -ls 2>/dev/null
132462 1112 -rw-r--r--   1 root     root      1130305 Aug 22 15:41 /etc/selinux/targeted/policy/policy.
132045  788 -rw-r--r--   1 root     root       798555 Dec  4  2006 /etc/firmware/microcode.dat
132705  320 -rw-r--r--   1 root     root       319141 Aug 24 13:20 /etc/prelink.cache
130862  800 -rw-r--r--   1 root     root       807103 Jul 12  2006 /etc/termcap
131090  440 -rw-r--r--   1 root     root       441017 Nov 30  2006 /etc/pki/tls/certs/ca-bundle.crt
```

```
132279  460 -rw-r--r--   1 root     root       459789 Aug 22 15:43 /etc/gconf/gconf.xml.defaults/%gconf
132673  468 -rw-r--r--   1 root     root       467812 Aug 22 15:43 /etc/gconf/gconf.xml.defaults/%gconf
...
```

Now, she makes a directory called `/tmp/big`, and composes a **cp** command on the **find** command line, remembering the following.

- Place a *{}* as a placeholder for matching file names
- Terminate the command with a *\;*.

```
[madonna@station madonna]$ mkdir /tmp/big
[madonna@station madonna]$ find /etc -size +200k -exec cp {} /tmp/big \;  2>/dev
/null
[madonna@station madonna]$ ls /tmp/big/
apps_gnome_settings_daemon_keybindings.schemas  %gconf-tree-nn.xml
apps_nautilus_preferences.schemas               %gconf-tree-or.xml
ca-bundle.crt                                   %gconf-tree-pa.xml
clock.schemas                                   %gconf-tree-pl.xml
...
```

Rather than printing the file name, the find command copied the files to the `/tmp/big` directory.

Examples

Using locate

There are several ways to find a specific file.

```
[blondie@station blondie]$ locate rmdir
/bin/rmdir
/usr/lib/perl5/5.8.8/i386-linux-thread-multi/auto/POSIX/rmdir.al
/usr/share/doc/bash-3.1/loadables/rmdir.c
/usr/share/man/man1/rmdir.1.gz
/usr/share/man/man1p/rmdir.1p.gz
/usr/share/man/man2/rmdir.2.gz
/usr/share/man/man3p/rmdir.3p.gz
[blondie@station blondie]$ find /bin -name "*dir*"
/bin/mkdir
/bin/rmdir
[blondie@station blondie]$ which rmdir
/bin/rmdir
```

In the above examples, **locate** shows everything in the database with the string "rmdir" including the command and man pages. **find** shows all files under `/bin` that include "dir" in the name. Finally, **which** shows the absolute path for a known command.

You can also include filename expansion characters in your search:

```
[blondie@station blondie]$ locate "*theme*png"
...
/home/elvis/gdm/themes/RHEL/background.png
/home/elvis/gdm/themes/RHEL/distribution.png
/home/elvis/gdm/themes/RHEL/icon-language.png
/home/elvis/gdm/themes/RHEL/icon-reboot.png
/home/elvis/gdm/themes/RHEL/icon-session.png
/home/elvis/gdm/themes/RHEL/icon-shutdown.png
/home/elvis/gdm/themes/RHEL/logo.png
...
```

Recall that **locate** uses a database and will not locate files that have been created since the database was last updated. The example below should not show any output from the locate command.

```
[blondie@station blondie]$ touch ~/locate_example_file
[blondie@station blondie]$ locate locate_example_file
```

Because the locate database does not yet know about the `locate_example_file` file, no files are reported.

Using find

The **find** searches the actual directory tree from the specified beginning point.

```
[elvis@station elvis]$ find /home/elvis
/home/elvis
/home/elvis/.metacity
/home/elvis/.metacity/sessions
/home/elvis/.metacity/sessions/1187895446-1983-3849805419.ms
/home/elvis/.metacity/sessions/1187972526-2245-2949308818.ms
...
```

Multiple starting directories can be specified for the **find** command.

```
[elvis@station elvis]$ find /bin /usr/bin -name "*dir*"
/bin/rmdir
/bin/mkdir
/usr/bin/dir
/usr/bin/lndir
/usr/bin/dirname
...
```

The **find** command can be used to discover symbolic links.

```
[elvis@station ~]$ find /usr/bin -type l
/usr/bin/lastb
/usr/bin/spam
/usr/bin/gnome-character-map
/usr/bin/rdistd
/usr/bin/ipmish
/usr/bin/python2
...
[elvis@station ~]$ ls -l /usr/bin/lastb
lrwxrwxrwx 1 root root 4 Aug 22 15:36 /usr/bin/lastb -> last
```

As a complicated example, **find** can produce a "**ls -l** style" listing of everything on the system not owned by the users *root*, *bin* or *elvis*. As there may be directories where search access is denied, redirecting errors to `/dev/null` reduces screen clutter.

```
[elvis@station elvis]$ find / -not -user root -not -user bin -not -user elvis
-ls 2> /dev/null
198506   96 -rwxr-xr-x   1 rpm      rpm         89344 Jan  4  2007 /bin/rpm
1013889    8 drwx------   3 madonna  madonna      4096 Aug 25 08:43 /home/madonna
1013833    8 drwx------   3 pataki   pataki       4096 Aug 22 15:43 /home/pataki
1013986    8 drwx------   4 rhauser_a rhauser_a     4096 Aug 23 20:33 /home/rhauser_a
1013903    8 drwx------   3 prince   prince       4096 Aug 22 15:43 /home/prince
1014000    8 drwx------   3 rhauser_c rhauser_c     4096 Aug 23 11:02 /home/rhauser_c
1013896    8 drwx------   3 bob      bob          4096 Aug 22 15:43 /home/bob
1013854    8 drwx------   3 blondie  blondie      4096 Aug 25 08:59 /home/blondie
1013875    8 drwx------   3 nero     nero         4096 Aug 22 15:43 /home/nero
1013882    8 drwx------   3 einstein einstein     4096 Aug 22 15:43 /home/einstein
...
```

Using find to Execute Commands on Files

Find all the files under `/tmp` with a link count greater than 1 and make a copy of each in a directory called `/tmp/links`.

```
[blondie@station blondie]$ ls -l /tmp/*file
-rw-rw-r--    2 blondie    blondie           0 Mar 17 22:33 /tmp/linkfile
-rw-rw-r--    2 blondie    blondie           0 Mar 17 22:33 /tmp/newfile
[blondie@station blondie]$ mkdir /tmp/links
[blondie@station blondie]$ find /tmp -type f -links +1 -exec cp {} /tmp/links \;
[blondie@station blondie]$ ls /tmp/links
linkfile
newfile
```

Online Exercises

Locating files

Lab Exercise

Objective: Devise and execute a **find** command that produces the result described in each of the following.

Estimated Time: 20 mins.

Specification

Use the **find** command to find files which match the following criteria, and redirect the output to the specified files in your home directory. When listing filenames, make sure every filename is an absolute reference.

You will encounter a number of "Permission denied" messages when **find** tries to recurse into directories for which you do not have permissions to access. Do not be concerned with these errors. You can suppress these error messages by appending **2> /dev/null**to your **find** command.

You may need to consult the **find(1)** man page to find the answer for some of the problems.

Deliverables

The following command lines will produce the appropriate answers.

find /var/lib -user webalizer

find /var -user root -group mail

find /usr/bin -size +1000000c -ls

find /etc/sysconfig -exec file {} \;

find /usr/bin -type f -links +3

1.
    1. The file `varlib.webalizer`, which contains a list of all files under the `/var/lib` directory which are owned by the user "webalizer".

    2. The file `var.rootmail`, which contains a list of all files under the `/var` directory which are owned by the user "root" and group owned by the group "mail".

    3. The file `bin.big` which contains a **ls -dils** style listing of all files under the `/usr/bin` directory that are greater than 1000000 characters in size.

4. Execute the **file** command on every file under `/etc/sysconfig`, and record the output in the file `sysconfig.find`.
5. The file `big.links`, which contains a list of the filenames of regular files underneath the `/usr/bin` directory which have a link count of greater than 3.

Questions

1. Which **find** option will locate files of exactly 100 blocks?
   a. -size +100
   b. -size 100
   c. -inum 100
   d. -inum +100
2. Which **find** option will locate files with inode number 100?
   a. -type f
   b. -size 100
   c. -inum 100
   d. -perm 100
3. Which **find** option or options will locate only ordinary files which have a link count of 2 or more?
   a. -type f -links +2
   b. -links +1
   c. -type o -links +1
   d. -type f -links +1
4. Which **find** option will print the output in a **ls -l** style format?
   a. -type
   b. -size
   c. -inum
   d. -ls
5. Which **find** option or options will locate files owned by user root and group sys?
   a. -user root -and -group sys
   b. -user root -group sys
   c. -user root
   d. -group sys

6. Which command or commands will list files that were recently created and include the string "coffee" in their names?

   a. slocate coffee

   b. find . -name coffee

   c. find . -name "*coffee*"

   d. ls -R *coffee*

7. Which command or commands will list files that include the string "coffee" in their names?

   a. slocate coffee

   b. find . -name coffee

   c. find . -name "*coffee*"

   d. ls -R *coffee*

8. Which **find** command will locate ordinary files under `/home` or `/tmp` with world writable permissions?

   a. find /home /tmp -type f -perm -2

   b. find /home -or /tmp -type f -perm 002

   c. find /home /tmp -type o -perm 2

   d. find /home /tmp -perm -2

9. Which **find** option can be added to the previous answer so that each file found will have the "other" write permission removed?

   a. -exec chmod o-w \;

   b. -exec chmod o-w

   c. -exec chmod o-w {} \;

   d. -exec chmod -2

10. Which **find** option can be used such that each file found will have permissions removed and have **find** prompt interactively whether or not to change the permissions?

    a. -ok

    b. -ask

    c. -exec

    d. -exec -ok

Chapter 6. Compressing Files: gzip and bzip2

Key Concepts

- Compressing seldom used files saves disk space.
- The most commonly used compression command is **gzip**.
- The **bzip2** command is newer, and provides the most efficient compression.

Discussion

Why Compress Files?

Files that are not used very often are often compressed. Large files are also compressed before transferring to other systems or users. The advantages of saved space and bandwidth usually outweighs the added time it takes to compress and uncompress files.

Text files often have patterns that can be compressed up to 75% but binary files rarely compress more than 25%. In fact, it is even possible for a compressed binary file to be larger than the original file!

Standard Linux Compression Utilities

As better and better compression techniques have been developed, new compression utilities have gained favor. For backwards compatibility, however, older compression utilities are still retained. Often, there is a trade off between compression efficiency and CPU activity. Sometimes, older compression utilities do "good enough" in a much shorter time.

The following list discusses the two most common compression utilities used in Linux and Unix.

gzip (.gz)
: The **gzip** command is the most versatile and most commonly used decompression utility. Files compressed with **gzip** are uncompressed with **gunzip**. Additionally, the **gzip** command supports the following command line switches.

Switch	Effect
-c	Redirect Output to stdout
-d	Decompress instead of compress file
-r	Recurse through subdirectories, compressing individual files.
-1 ... -9	Specify trade off between CPU intensity and compression efficiency.

bzip2 (.bz)
: The **bzip2** command is a relative newcomer, which tends to produce the most compact compressed files, but is the most CPU intensive. Files compressed with **bzip2** are uncompressed with **bunzip2**. The **bzip2** command supports the following command line switches.

Switch	Effect
-c	Redirect Output to stdout
-d	Decompress instead of compress file

The following examples illustrate the use and relative efficiency of the compression commands.

```
[elvis@station elvis]$ ls -sh termcap
725K termcap
[elvis@station elvis]$ gzip termcap
[elvis@station elvis]$ ls -sh termcap*
234K termcap.gz
[elvis@station elvis]$ gzip -d termcap
[elvis@station elvis]$ ls -sh termcap*
725K termcap
```

```
[elvis@station elvis]$ ls -sh termcap
725K termcap
[elvis@station elvis]$ bzip2 termcap
[elvis@station elvis]$ ls -sh termcap*
185K termcap.bz2
[elvis@station elvis]$ bunzip2 termcap.bz2
[elvis@station elvis]$ ls -sh termcap*
725K termcap
```

Other Compression Utilities

Another compression utility available in Red Hat Enterprise Linux is **zip**. This utility is compatible with the DOS/Windows **PKzip/Winzip** utilities and can compress more than one file into a single file, something that **gzip** and **bzip2** cannot do.

Linux and Unix users often prefer instead to use **tar** and **gzip** together in preference to **zip**. The command **tar** is discussed in the next lesson.

Examples

Working with gzip

Madonna also has a copy of the same bigfile but prefers to use **gzip** compression.

```
[madonna@station madonna]$ gzip bigfile
[madonna@station madonna]$ ls -l bigfile*
-rw-r--r--    1 madonna madonna   131069 Mar 18 15:29 bigfile.gz
[madonna@station madonna]$ gunzip bigfile.gz
[madonna@station madonna]$ ls -l bigfile*
-rw-r--r--    1 madonna madonna   409305 Mar 18 15:29 bigfile
```

Notice the better compression algorithm from this utility.

Using gzip Recursively

The **gzip** command includes a **-r** command line switch, which will recurse through subdirectories, compressing individual files. In the following example, madonna will create a local copy of the `/etc/sysconfig/networking` directory, and then recursively compress the copy.

```
[madonna@station madonna]$ cp -r /etc/sysconfig/networking .
[madonna@station madonna]$ gzip -r networking
```

```
[madonna@station madonna]$ tree networking/
networking/
|-- devices
|   `-- ifcfg-eth0.gz
|-- ifcfg-lo.gz
`-- profiles
    |-- default
    |   |-- hosts.gz
    |   |-- ifcfg-eth0.gz
    |   |-- network.gz
    |   `-- resolv.conf.gz
    `-- netup
        |-- hosts.gz
        |-- ifcfg-eth0.gz
        |-- network.gz
        `-- resolv.conf.gz

4 directories, 10 files
```

Working with bzip2

Elvis realizes that the **compress** utility that he first used is old and decides to try a much newer compression utility.

```
[elvis@station elvis]$ bzip2 bigfile
[elvis@station elvis]$ ls -l bigfile*
-rw-r--r--    1 elvis elvis   154563 Mar 18 15:29 bigfile.bz2
[elvis@station elvis]$ bunzip2 bigfile.bz2
[elvis@station elvis]$ ls -l bigfile*
-rw-r--r--    1 elvis elvis   409305 Mar 18 15:29 bigfile
```

Notice that to uncompress this archive, Elvis must give the filename with the `bz2` extension. In the other examples, the utility used could find a file of the given base name with a known extension.

Online Exercises

Working with compression Utilities

Lab Exercise

Objective: Compress large files

Estimated Time: 10 mins.

Specification

1. Copy the files `/etc/gconf/schemas/gnome-terminal.schemas` and `/usr/bin/gimp` into your home directory, preserving their original filenames. (The first is an example of a large text file, the second is an example of a large binary file.) Use the **gzip** command to compress each of the newly created files.

2. Again, copy the files `/etc/gconf/schemas/gnome-terminal.schemas` and `/usr/bin/gimp` into your home directory. This time, use the **bzip2** command to compress the two files.

3. One last time, copy the `/etc/gconf/schemas/gnome-terminal.schemas` and `/usr/bin/gimp` files into your home directory. Use the **ls -s** command to compare the sizes of the various compression techniques.

Deliverables

1.
    1. The file `gnome-terminal.schemas` in your home directory, which is a copy of `/etc/gconf/schemas/gnome-terminal.schemas`.
    2. The file `gnome-terminal.schemas.gz`, the **gzip**ed version of `gnome-terminal.schemas`.
    3. The file `gnome-terminal.schemas.bz2`, the **bzip2**ed version of `gnome-terminal.schemas`.
    4. The file `gimp` in your home directory, which is a copy of `/usr/bin/gimp`.
    5. The file `gimp.gz`, the **gzip**ed version of `gimp`.
    6. The file `gimp.bz2`, the **bzip2**ed version of `gimp`.

Questions

1. What filename extension is generally associated with files compressed using the **bzip2** utility?

    a. .Z

    b. .gz

    c. .bz2

    d. .tar

2. What filename extension is generally associated with files compressed using the **gzip** utility?

    a. .Z

    b. .gz

    c. .bz2

    d. .tar

3. Which commands can uncompress a `.gz` file?

    a. uncompress

    b. gunzip

    c. gzip -d

    d. bunzip2

4. Why is compression most useful for text files?

    a. Binary files may get corrupted when compressed.

    b. Binary files are always larger after being compressed

    c. Utilities cannot compress a binary file

d. Binary files are often already efficiently using space and little is gained by compressing them.

5. Assuming that the text files exist in the current directory, what will the command **gzip report.txt draft.txt schedule.txt** produce?

a. A single file with three text files in it.

b. A single compressed file with three text files in it.

c. Three compressed text files.

d. An error message.

Chapter 7. Archiving Files with tar

Key Concepts

- Archiving files allows an entire directory structure to be stored as a single file.
- Archives are created, listed, and extracted with the **tar** command.
- Archive files are often compressed as well.
- The **fileroller** application provides a GUI interface to archiving files.

Discussion

Archive Files

Often, if a directory and its underlying files are not going to be used for a while, or if the entire directory tree is going to be transferred from one place or another, people convert the directory tree into an *archive* file. The archive contains the directory and its underlying files and subdirectories, packaged as a single file. In Linux (and Unix), the most common command for creating and extracting archives is the **tar** command.

Originally, archive files provided a solution to backing up disks to tape. When backing up a filesystem, the entire directory structure would be converted into a single file, which was written directly to a tape drive. The **tar** command derived its name from "t"ape "ar"chive.

Today, the **tar** is seldom used to write to tapes directly, but instead creates archive files which are often referred to as "tar files", "tar archives", or sometimes informally as "tarballs". These archive files are conventionally given the `.tar` filename extension.

Tar Command Basics

When running the **tar** command, the first command line must be selected from the following choices.

Switch	Effect
-c, --create	Create an archive file
-x, --extract	Extract an archive file
-t, --list	List the contents of an archive file

There are others, but almost always one of these three will suffice. See the tar(1) man page for more details.

Next, almost every invocation of the **tar** command must include the **-f** command line switch and its argument, which specifies which archive file is being created, extracted, or listed.

As an example, the user prince has been working on a report, which involves several subdirectories and files.

```
report/
|-- html/
|   |-- chap1.html
|   |-- chap2.html
|   `-- figures/
|       `-- image1.png
```

```
`-- text/
    |-- chap1.txt
    `-- chap2.txt

3 directories, 5 files
```

He would like to email a copy of the report to a friend. Rather than attach each individual file to an email message, he decides to create an archive of the report directory. He uses the **tar** command, specifying **-c** to "c"reate an archive, and using the **-f** command line switch to specify the archive file to create.

```
[prince@station prince]$ tar -c -f report.tar report
[prince@station prince]$ ls -s
total 24
   4 report    20 report.tar
```

The newly created archive file `report.tar` now contains the entire contents of the `report` directory, and its subdirectories. In order to confirm that the archive was created correctly, prince lists the contents of the archive file with the **tar -t** command (again using **-f** to specify which archive file).

```
[prince@station prince]$ tar -t -f report.tar
report/
report/text/
report/text/chap1.txt
report/text/chap2.txt
report/html/
report/html/figures/
report/html/figures/image1.png
report/html/chap1.html
report/html/chap2.html
```

As further confirmation, prince extracts the archive file in the `/tmp` directory, using **tar -x**.

```
[prince@station prince]$ cd /tmp
[prince@station tmp]$ tar -x -f /home/prince/report.tar
[prince@station tmp]$ ls -R report/
report/:
html  text

report/html:
chap1.html  chap2.html  figures

report/html/figures:
image1.png

report/text:
chap1.txt  chap2.txt
```

Now convinced that the archive file contains the report, and that his friend should be able to extract it, he cleans up the test copy, and uses the **mutt** command to email the archive as an attachment.

```
[prince@station tmp]$ rm -fr report/
[prince@station tmp]$ cd
[prince@station prince]$ mutt -a report.tar -s "My Report" elvis@example.com
```

Don't be concerned if you aren't familiar with **mutt**. This just serves as an example of why someone might want to create a **tar** archive.

More About tar

The first command line switch to the **tar** command must be one of the special switches discussed above. Because the first switch is always one of a few choices, the **tar** command allows a shortcut; you do not need to include the leading hyphen. Often, experienced users of **tar** will use shortened command lines like the following.

```
[prince@station prince]$ tar cf report.tar report
[prince@station prince]$ tar tf report.tar
report/
report/text/
report/text/chap1.txt
report/text/chap2.txt
report/html/
report/html/figures/
report/html/figures/image1.png
report/html/chap1.html
report/html/chap2.html
```

Creating archives introduces a lot of complicated questions, such as some of the following.

- When creating archives, how should links be handled? Do I archive the link, or what the link refers to?
- When extracting archives as root, do I want all of the files to be owned by root, or by the original owner? What if the original owner doesn't exist on the system I'm unpacking the tar on?
- What happens if the tape drive I'm archiving to runs out of room in the middle of the archive?

The answers to these, and many other questions as well, can be decided with an overwhelming number of command line switches to the **tar** command, as **tar --help** or a quick look at the tar(1) man page will demonstrate. The following table lists some of the more commonly used switches, and there use will be discussed below.

Switch	Effect
-C, --directory=*DIR*	Change to directory *DIR*
-P, --absolute-reference	don't strip leading / from filenames
-v, --verbose	list files processed
-z, --gzip	internally **gzip** archive
-j, --bzip2	internally **bzip2** archive

Absolute References

Suppose prince wanted to archive a snapshot of the current networking configuration of his machine. He might run a command like the following. (Note the inclusion of the **-v** command line switch, which lists each file as it is processed.)

```
[prince@station prince]$ tar cvf net.tar /etc/sysconfig/networking
tar: Removing leading `/' from member names
etc/sysconfig/networking/
etc/sysconfig/networking/devices/
etc/sysconfig/networking/devices/ifcfg-eth0
etc/sysconfig/networking/profiles/
etc/sysconfig/networking/profiles/default/
etc/sysconfig/networking/profiles/default/network
...
```

As the leading message implies, what was an absolute reference to `/etc/sysconfig/networking` is converted to relative references inside the archive: None of the entries have leading slashes. Why is this done? What happens if prince turns right around and extracts the archive?

```
[prince@station prince]$ tar xvf net.tar
etc/sysconfig/networking/
etc/sysconfig/networking/devices/
etc/sysconfig/networking/devices/ifcfg-eth0
```

```
etc/sysconfig/networking/profiles/
etc/sysconfig/networking/profiles/default/
etc/sysconfig/networking/profiles/default/network
etc/sysconfig/networking/profiles/default/resolv.conf
...
[prince@station prince]$ ls -R etc/
etc/:
sysconfig

etc/sysconfig:
networking

etc/sysconfig/networking:
devices  ifcfg-lo  profiles

etc/sysconfig/networking/devices:
ifcfg-eth0

etc/sysconfig/networking/profiles:
default  netup

etc/sysconfig/networking/profiles/default:
hosts  ifcfg-eth0  network  resolv.conf

etc/sysconfig/networking/profiles/netup:
hosts  ifcfg-eth0  network  resolv.conf
```

Because the file entries were relative, the archive unpacked into the *local* directory. As a rule, archive files will always unpack locally, reducing the chance that you will unintentionally clobber files in your filesystem by unpacking an archive on top of them. When constructing the archive, this behavior can be overridden with the **-P** command line switch.

Establishing Context

When extracting the archive above, the first "interesting" directory is the `networking` directory, because it contains the relevant subdirectories and files. When extracting the archive, however, and "extra" `etc` and `etc/sysconfig` are created. In order to get to the interesting directory, someone has to work his way down to it.

When constructing an archive, the **-C** command line switch can be used to help establish context by changing directory before the archive is constructed. Compare the following two **tar** commands.

```
[prince@station prince]$ tar cvf net.tar /etc/sysconfig/networking
tar: Removing leading `/' from member names
etc/sysconfig/networking/
etc/sysconfig/networking/devices/
etc/sysconfig/networking/devices/ifcfg-eth0
etc/sysconfig/networking/profiles/
etc/sysconfig/networking/profiles/default/
etc/sysconfig/networking/profiles/default/network
...
[prince@station prince]$ tar cvf net.tar -C /etc/sysconfig networking
networking/
networking/devices/
networking/devices/ifcfg-eth0
networking/profiles/
networking/profiles/default/
networking/profiles/default/network
...
```

In the second case, the **tar** command first changes to the `/etc/sysconfig` directory, and then creates a copy of the `networking` directory found there. When the resulting archive file is extracted, the "interesting" directory is immediately available.

Of course, prince could have used the **cd** command before running the **tar** command to the same effect, but the **-C** command line switch is often more efficient.

Compressing archives

Often, the **tar** command is used to archive files that will not be used anytime soon. Because the resulting archive files will not be used soon, they are compressed as well. In the following, prince is able to save a significant amount of disk space by **gzip**ing his archive of his home directory.

```
[prince@station prince]$ tar cf /tmp/prince.tar -C /home/prince .
[prince@station prince]$ ls -s /tmp/prince.tar
 224 /tmp/prince.tar
[prince@station prince]$ gzip /tmp/prince.tar
[prince@station prince]$ ls -s /tmp/prince.tar.gz
  28 /tmp/prince.tar.gz
```

Because users are often creating and then compressing archives, or dealing with archives that have been compressed, the **tar** command provides three command line switches for internally compressing (or decompressing) archive files. Above, prince could have obtained the same result by adding a **-z** command line switch.

```
[prince@station prince]$ tar czf /tmp/prince.tar.gz -C /home/prince .
[prince@station prince]$ ls -s /tmp/prince.tar.gz
  28 /tmp/prince.tar.gz
```

The combination of **tar** and **gzip** is found so often, that often the `.tar.gz` filename extension will be abbreviated `.tgz`.

```
[prince@station prince]$ tar czf /tmp/prince.tgz -C /home/prince .
[prince@station prince]$ tar tzf /tmp/prince.tgz
./
./.bash_profile
./Desktop/
./Desktop/redhat_academy.desktop
./.bashrc
./.bash_logout
./.zshrc
```

With older versions of the **tar** command, when expanding or listing a **tar** archive, you would have to respecify the appropriate compression (with the **-z** or **-j** command line switches). In recent version of Red Hat Enterprise Linux, however, the **tar** command will now automatically recognize a compressed archive, and decompress it appropriately.

```
[prince@station prince]$ tar tf /tmp/prince.tgz
./
./.bash_profile
./Desktop/
./Desktop/redhat_academy.desktop
./.bashrc
./.bash_logout
./.zshrc
```

Examples

Creating a tar Archive

The user einstein wants to make a copy of the **bash** documentation that he can take along with him. He quickly tars up the `/usr/share/doc/bash-2.05b` directory.

```
[einstein@station einstein]$ tar cvzf bashdoc.tgz -C /usr/share/doc bash-3.1
bash-3.1/
```

```
bash-3.1/bashref.ps
bash-3.1/bash.0
bash-3.1/bash.html
bash-3.1/article.ps
bash-3.1/complete/
bash-3.1/complete/complete2.ianmac
...
[einstein@station einstein]$ ls -s bashdoc.tgz
 1240 bashdoc.tgz
```

Once he gets the file to its new location, he extracts the archive.

```
[einstein@station einstein]$ tar xvf bashdoc.tgz
bash-3.1/
bash-3.1/bashref.ps
bash-3.1/bash.0
bash-3.1/bash.html
bash-3.1/article.ps
bash-3.1/complete/
bash-3.1/complete/complete2.ianmac
...
```

Tarring Directly to a Floppy

The user maxwell wants to quickly compare the LDAP configuration on two different machines. The machines are not connected to a network, but both have a floppy drive. Rather than creating an archive, formatting a floppy, mounting the floppy, copying the archive, and unmounting the floppy, maxwell decides to save a few steps. With an *unmounted* floppy in the drive, maxwell runs the following command.

```
[maxwell@station maxwell]$ tar cvzf /dev/fd0 -C /etc openldap
openldap/
openldap/ldapfilter.conf
openldap/ldap.conf
openldap/ldapsearchprefs.conf
openldap/ldaptemplates.conf
```

He then ejects the floppy and carries it to the second machine. The following command extracts the archive into his local directory.

```
[maxwell@station maxwell]$ tar xvzf /dev/fd0
openldap/
openldap/ldapfilter.conf
openldap/ldap.conf
openldap/ldapsearchprefs.conf
openldap/ldaptemplates.conf
openldap/ldaptemplates.conf

gzip: stdin: decompression OK, trailing garbage ignored
tar: Child died with signal 13
tar: Error exit delayed from previous errors
```

Although the **tar** command (or, more accurately, the **gzip** command) complained about "trailing garbage", the archive was successfully extracted.

What happened here? The tar command wrote directly to the floppy's device node, so the archive file was written byte for byte onto the floppy as raw data. Upon extracting the archive, the file was read byte for byte, until the file was entirely read. The **gzip** command kept going, however, trying to decompress whatever was sitting on the floppy before the archive was written. This is the "trailing garbage" that the **gzip** command complained about. What was the filename of the archive as it was sitting on the floppy? (Trick question!)

The file doesn't have a name, because the floppy doesn't have a filesystem. (What ever filesystem might have existed on the floppy was destroyed by the archive).

Oops.

The user einstein wants to create an archive of his home directory. He tries the following command.

```
[einstein@station einstein]$ tar cvzf ~/einstein.tgz ~
tar: Removing leading `/' from member names
home/einstein/
home/einstein/.kde/
home/einstein/.kde/Autostart/
...
home/einstein/.bash_history
home/einstein/einstein.tgz
tar: /home/einstein/einstein.tgz: file changed as we read it
tar: Error exit delayed from previous errors
```

Why did the **tar** command error out? The archive was being written to the file `/home/einstein/einstein.tgz`. The archive included every file in the `/home/einstein` directory. Eventually, the **tar** command tried to append the file `/home/einstein/einstein.tgz` to the archive `/home/einstein/einstein.tgz`. This obviously causes problems.

Fortunately, the **tar** command is now smart enough to detect circular references. In the (not too distant) "old days", the first clue that something was wrong in situations like this was the long time it took the **tar** command to run, and the second clue was the error message saying that the disk was out of space.

What's the solution? Make sure that the archive file you're creating does not exist in the directory your archiving. The `/tmp` directory comes in handy.

```
[einstein@station einstein]$ tar czf /tmp/einstein.tgz ~
tar: Removing leading `/' from member names
[einstein@station einstein]$ mv /tmp/einstein.tgz .
```

Online Exercises

Archiving Directories

Lab Exercise

Objective: Create an archive using the **tar** command.

Estimated Time: 15 mins.

Specification

1. In your home directory, create the file `zip_docs.tar` which is an archive of the documentation for the **zip** package located in the `/usr/share/doc/zip*` directory.

2. Create the file `/tmp/`*`student`*`.tgz`, which is a **gzip**ed archive of your home directory. Replace *student* with your username.

Deliverables

Solutions

*tar cvf ~/zip_docs.tar /usr/share/doc/zip**

tar cvzf /tmp/student.tgz ~

1.
    1. The file `zip_docs.tar` in your home directory, which is an archive of the `/usr/share/doc/zip*` directory.
    2. The file `/tmp/`*`student`*`.tgz`, with *`student`* replaced with your username, which is a **gzip**ed archive of your home directory.

Questions

1. Which of the following commands would create an archive called `archive.tar`?

    a. **`tar -c -f archive.tar`**

    b. **`tar -x -f archive.tar /usr/games`**

    c. **`tar -t -f archive.tar /usr/games`**

    d. **`tar -c -f archive.tar /usr/games`**

    e. None of the above.

2. Which of the following commands would list the contents of an archive file called `archive.tar`?

    a. **`tar tf archive.tar`**

    b. **`tar -xf archive.tar`**

    c. **`tar -c -f archive.tar`**

    d. **`tar --list archive.tar`**

    e. None of the above.

3. Which of the following commands would extract the contents of an archive file called `archive.tar`?

    a. **`tar tf archive.tar`**

    b. **`tar -xf archive.tar`**

    c. **`tar -c -f archive.tar`**

    d. **`tar --list archive.tar`**

    e. None of the above.

4. You have downloaded a file titled `linux-2.5.34.tar.gz`. Which of the following commands can you run to extract the contents of the file?

    a. **`tar xvzf linux-2.5.34.tar.gz`**

    b. **`tar -x -f linux-2.5.34.tar.gz`**

    c. **`tar -x -z -f linux-2.5.34.tar.gz`**

    d. **`tar -xZf linux-2.5.34.tar.gz`**

e. **`tar fz linux-2.5.34.tar.gz`**

f. **`tar -x -f linux-2.5.34.tar.gz -z`**

5. You would like to make a **bzip2** compressed archive of the `/usr/share/sounds` directory, so that when someone extracts the archive, it extracts starting with the directory `sounds`. Which of the following commands will create the archive?

   a. **`tar -c -f /tmp/sounds.tar.bz2 /usr/share/sounds`**

   b. **`tar cvjf /tmp/sounds.tar.bz2 -C /usr/share sounds`**

   c. **`tar -c -f /tmp/sounds.tar.bz2 -C /usr/share/sounds -j`**

   d. **`tar -cj /tmp/sounds.tar.bz2 -f /usr/share/sounds`**

   e. None of the above

6. What filename extension does the **tar** command add automatically when creating an archive?

   a. .tar

   b. .tgz

   c. .tar.gz

   d. .zip

   e. No extension is added by tar.

7. Usually, when a tar archive is extracted with the command **`tar xzf archive.tgz`**, where are the files placed?

   a. Relative to the root of the current filesystem

   b. Relative to the current working directory

   c. Relative to the directory specified on the command line

   d. Relative to the root of the root filesystem

   e. Relative to the `/tmp` directory

8. A file has been downloaded called `archive.tgz`. How can you view the contents of this archive?

   a. tar tzvf archive.tgz

   b. tar jtvf archive.tgz

   c. tar tvf archive.tgz

   d. tar cvzf archive.tgz

   e. tar tf archive.tgz