15

Essential System Administration

In the concluding act for Part I of this book, it's time to make a visit to the world of system administration. This task is usually entrusted to a single person—the **system administrator**, also known as the **superuser** or **root user**. The administrator has vast powers, having access to practically everything. The success and stability of any UNIX installation depends, in great measure, on the effectiveness of the system administrator. Today, every user must know some of the important administrative functions that she may be called upon to perform at any time.

The job of system administration involves the management of the entire system—ranging from maintaining user accounts, security and managing disk space to performing backups. The burden is not overwhelming though because UNIX is more easily maintained and well documented than most other systems. However, UNIX gets greatly fragmented in this area, and POSIX makes no attempt to address administrative issues. We consider the routine duties in this chapter and the more involved ones in Chapter 25.

What You Will Learn

- Log in to root and become a *superuser* with **su**.
- The administrator's powers in changing the attributes of any file and in killing any process.
- Control access to the scheduling services, **at** and **cron**.
- Create, modify and delete user accounts with **useradd**, **usermod** and **userdel**.
- The concept of *run levels* and their role in startup and shutdown operations.
- Use **df** and **du** to report on both free and used disk space.
- The significance of the attributes of device files.
- Format and copy diskettes with **format**, **fdformat** and **dd**.
- Handle DOS files on diskettes with a set of "dos" commands.
- Use **cpio** to back up and restore files.
- Use **tar** to back up a directory tree and append files to an archive.

TOPICS OF SPECIAL INTEREST

- The significance of the three special file permissions bits—*SUID*, *SGID* and *sticky bit*, and how the administrator uses them in enforcing security.
- Understanding how disk read/write operations take place.

15.1 root: THE SYSTEM ADMINISTRATOR'S LOGIN

The UNIX system provides a special login name for the exclusive use of the administrator; it is called **root**. This account doesn't need to be separately created but comes with every system. Its password is generally set at the time of installation of the system and has to be used on logging in:

```
login: root
password: *******/Enter/
# _
```

The prompt of root is #, unlike the $ or % used by nonprivileged users. Once you log in as root, you are placed in root's home directory. Depending on the system, this directory could be / or /root.

On modern systems, most administrative commands are resident in /sbin and /usr/sbin, but if you are using an older system, you could find them in /etc. root's PATH list is also different from the one used by other users:

```
/sbin:/bin:/usr/sbin:/usr/bin:/usr/dt/bin
```

Since the superuser has to constantly navigate the file system, it's possible that he might inadvertently execute programs and scripts written by other users. That's why the PATH for a superuser doesn't include the current directory.

Many of the standard scripts supplied with UNIX systems for system administration work (especially the scripts related to booting) rely on the Bourne shell for execution. As administrator, you have to carefully weigh the consequences of using the Korn shell since scripts developed under this shell may not run on another host, which may not have this shell. But you must not under any circumstances use C shell scripts. Linux uses Bash for normal and system administrative activities; there's no problem there.

15.1.1 su: Acquiring Superuser Status

Any user can acquire superuser status with the **su** command if she knows the root password. For example, the user juliet (with the home directory /home/juliet) becomes a superuser in this way:

```
$ su
Password: ********                    root's password
# pwd
/home/juliet                          Prompt changes, but directory doesn't
```

Though the current directory doesn't change, the # prompt indicates that juliet now has powers of a superuser. To be in root's home directory on superuser login, use **su -l**.

Creating a User's Environment Users often rush to the administrator with the complaint that a program has stopped running. The administrator first tries running it in a simulated environment.

su, when used with a -, recreates the user's environment without taking the login-password route:

```
su - henry                                    No password required
```

This sequence executes henry's `.profile` (or the relevant startup scripts), and temporarily creates henry's environment. **su** runs a separate sub-shell, so this mode is terminated by hitting *[Ctrl-d]* or using **exit**.

15.2 THE ADMINISTRATOR'S PRIVILEGES

The superuser has enormous powers, and any command invoked by him has a greater chance of success than when issued by others. The command may also behave differently or be reserved for his exclusive use. The superuser's authority is mainly derived from the power to

- Change the contents or attributes of any file like its permissions and ownership. He can delete any file even if the directory is write-protected!
- Initiate or kill any process. The administrator can directly kill all processes except the ones essential for running the system.
- Change any user's password without knowing the existing one.
- Set the system clock with **date**.
- Address all users concurrently with **wall**.
- Limit the maximum size of files that users are permitted to create with **ulimit**.
- Control users' access to the scheduling services like **at** and **cron**.
- Control users' access to many networking services like FTP, SSH, etc.

The administrator has to use these powers with utmost caution. An apparently innocent and unplugged loophole can cause disaster if that knowledge is acquired by a mischievous person. Let's now see how the administrator exercises some of the powers listed above.

15.2.1 date: Setting the System Date

We have used **date** as a "passive" command before *(3.2)*—to display the system date. The administrator uses the same command with a numeric argument to *set* the system date. This argument is usually an eight-character string of the form *MMDDhhmm*, optionally followed by a two- or four-digit year string:

```
# date 01092124
Thu Jan  9 21:24:00 IST 2003
```

UNIX systems will continue to understand the century for some time (until the year 2038, at least). Ensure that the date is reasonably accurate as the **cron** scheduler *(9.12)* uses the clock time to run jobs.

15.2.2 wall: Communicating with Users

The **wall** command addresses all users simultaneously. Most UNIX systems don't permit users to run this command (Linux excepted), and reserve it for the sole use of the administrator:

```
# wall
The machine will be shut down today
at 14:30 hrs. The backup will be at 13:30 hrs
[Ctrl-d]
```

All users currently logged in will receive this message on their terminal. This command is routinely executed by the administrator—especially before shutdown of the system.

15.2.3 ulimit: Setting Limits on File Size

Faulty programs or negligence can eat up disk space in no time. The **ulimit** command (now a builtin in all shells) imposes a restriction on the maximum size of a file that a user is permitted to create. Though an ordinary user can only reduce this default value, the superuser can increase it:

```
ulimit 20971510                    Measured in 512-byte blocks
```

You'll often place this statement in /etc/profile so that every user has to work within these restrictions. When used by itself, **ulimit** displays the current setting. Solaris and Linux show unlimited as the default output, but it could be different on your system.

15.2.4 Controlling Use of at and cron

The access to the use of **at** and **batch** is restricted and controlled by the files at.allow and at.deny in /etc/cron.d (/etc in Linux). If at.allow is present, only users listed in the file are permitted to use **at** and **batch**. If it is not present, the system checks at.deny for users who are barred from using these commands. If neither file is present, only the system administrator is permitted to invoke **at** and **batch**.

The **cron** scheduler *(9.12)* is used by the administrator to make commands like **find** and **du** compile useful information about the system, or for automating the backup operations. Like **at** and **batch**, the authorization to use it is controlled in identical manner by two files, cron.allow and cron.deny, in /etc/cron.d (/etc in Linux).

15.3 MAINTAINING SECURITY

Because security in a computer system eventually involves files, a faulty file permission can easily be exploited by a malicious user in a destructive manner. As administrator, you have to ensure that the system directories (/bin, /usr/bin, /etc, /sbin etc.) and the files in them are not writable by others. We'll now discuss some important security-related features found on a UNIX system.

15.3.1 passwd: Changing Any Password

passwd prompts for the existing password when the command is used by a nonprivileged user *(3.9)*. However, when the superuser uses the command, the system behaves in a more lenient manner:

```
# passwd
Changing password for root
Enter the new password (minimum of 5, maximum of 8 characters)
Please use a combination of upper and lower case letters and numbers.
```

```
New password: *********
Re-enter password: *********
Password changed.                                    To be entered twice
```

Note that the system doesn't prompt for the old password this time. The administrator must closely guard the superuser password. Otherwise, the entire UNIX system may have to be reloaded! UNIX also allows the administrator the privilege of changing anybody's password without knowing it:

```
passwd henry
```

Once again, the old password is not prompted for; the new one has only to be entered twice. As users often give out their passwords to others, passwords tend to be known to others over time. What makes matters worse is that users themselves are quite averse to changing their own passwords. The **passwd** command offers features that force users to change their passwords after a specific time. For details look up Chapter 25.

Note: passwd doesn't prompt for the old password when the command is used by the superuser for changing the root password.

15.3.2 Set-User-Id (SUID): Power for a Moment

Recall the discussions on process attributes *(9.4)* where we mentioned that sometimes the *effective UID* may not be the same as the *real UID*. The time has now come to discuss a security feature which exploits this fact. Many UNIX programs have a special permissions mode that lets users update sensitive system files—like /etc/shadow—something they can't do directly with an editor. This is true of the **passwd** program:

```
-rwsr-xr-x   1 root     shadow      34808 Nov 30 17:55 /usr/bin/passwd
```

The letter s in the user category of the permissions field represents a special mode known as the **set-user-id** (SUID). This mode lets a process have the privileges of the *owner* of the file during the instance of the program. Thus, when a nonprivileged user executes **passwd**, the *effective* UID of the process is not the user's, but of root's—the owner of the program. This SUID privilege is then used by **passwd** to edit /etc/shadow.

The SUID for any file can be set by the superuser using the **chmod** command. The syntax uses the character **s** as the permission:

```
# chmod u+s a.out ; ls -l a.out
-rwsr-xr-x   1 root     staff       2113 Mar 24 11:18 a.out
```

To assign SUID in an absolute manner, simply prefix 4 to whatever octal string you would otherwise use (like 4755 instead of 755). The **set-group-id** (SGID) is similar to SUID except that a program with SGID set allows the user to have the same power as the group which owns the program. The SGID bit is 2, and some typical examples could be **chmod g+s a.out** or **chmod 2755 a.out**.

The SUID mechanism, invented by Dennis Ritchie, is a potential security hazard. It lets a user acquire hidden powers by running such a file owned by root. As administrator, you must keep

track of all SUID programs owned by root that a user may try to create or copy. The **find** command easily locates them:

```
find /home -perm -4000 -print | mailx root
```

The extra octal bit (4) signifies the SUID mode, *but* **find** *treats the - before* 4000 *as representing any other permissions*. You can use **cron** to run this program at regular intervals and mail the file list to root.

Note: The fourth permission bit is used only when a special mode of a file needs to be set. It has the value 4 for SUID, 2 for SGID and 1 for the sticky bit. The other 3 bits have their usual significance.

15.3.3 The Sticky Bit

The **sticky bit** (also called the *saved text bit*) is the last permission bit remaining to be discussed. It applies to both regular files and directories. When applied to a regular file, it ensures that the text image of a program with the bit set is permanently kept in the swap area so it can be reloaded quickly when the program's turn to use the CPU arrives. Previously, it made sense to have this bit set for programs like **vi** and **emacs**. Today, machines with ultra-fast disk drives and lots of cheap memory don't need this bit for ordinary files.

However, the sticky bit becomes a useful security feature when used with a directory. The UNIX system allows users to create files in /tmp and /var/tmp, but no one can delete files not owned by her. Strange, isn't it? That's possible because both directories have their sticky bits set:

```
# ls -ld /tmp /var/tmp
drwxrwxrwt   5 root     sys          377 Jan  9 13:28 /tmp
drwxrwxrwt   2 root     sys         7168 Jan  9 13:34 /var/tmp
```

The directories are *apparently* writable by all, but that extra t (sticky) bit ensures that kumar can't remove sharma's files in these directories. Using **chmod**, you can set the bit on a directory by using 1 as the additional bit:

```
# chmod 1775 bar                                        Or chmod +t bar
# ls -l bar
drwxrwxr-t   2 sumit    dialout      1024 Apr 13 08:25 bar
```

The sticky bit is extremely useful for implementing group projects. To let a group of users work on a set of files without infringing on security, you'll have to do this:

1. Create a common group for these users in /etc/group.
2. Create separate user accounts for them but specify the same home directory.
3. Make sure the home directory and all subdirectories are not owned by any of the users. Use **chown** to surrender ownership to root.
4. Make the directories group-writable and set their sticky bits with **chmod 1775**.

In this scenario, every user of the group has write permission on the directory and can create files and subdirectories, but can only delete those she owns. A very useful feature indeed!

15.4 USER MANAGEMENT

The term *user* in UNIX is not meant to be only a person; it can represent a project or an application as well. A group of users performing similar functions may use the same username to use the system. It's thus quite common to have usernames like marketing, accounts, and so forth. For the creation and maintenance of user accounts, UNIX provides three commands—**useradd**, **usermod** and **userdel**.

When opening a user account, you have to associate the user with a group. A group usually has more than one member with a different set of privileges. People working on a common project should be able to read one another's files, which is possible only if they belong to the same group.

- Creating a user involves defining the following parameters:
- A user identification number (UID) and username.
- A group identification number (GID) and group name.
- The home directory.
- The login shell.
- The mailbox in `/var/mail`.
- The password.

Most of these parameters are found in a single line identifying the user in `/etc/passwd`. We'll now create a group for a user and then add that user to the system.

15.4.1 groupadd: Adding a Group

If the user is to be placed in a new group, an entry for the group has to be created first in `/etc/group`. A user always has one primary group and may also have one or more **supplementary groups**. This file contains all of the named groups of the system, and a few lines of this file reveal the structure:

```
root:x:0:root
bin:x:1:root,bin,daemon
lp:x:7:
uucp:x:14:uucp,fax,root,fnet,sumit
users:x:100:henry,oracle,image,enquiry
```

Each line contains four colon-delimited fields. Let's focus our attention on the group named users shown in the first field. This is the same name you see in the group ownership column of the listing. The second field once represented the group password but is hardly used today; it is either blank or an `x`. The third field shows the user's GID (here, 100). The last field contains a list of comma-delimited usernames (henry,oracle,image,enquiry) for whom this is the supplementary group. A blank at this position doesn't mean that no one is a member of this group; it's just that it's not the supplementary group for any user. Note that primary group for a user is shown in `/etc/passwd`.

To create a new group, dba, with a GID of 241, you have to use the **groupadd** command:

```
groupadd -g 241 dba                    241 is the GID for dba
```

The command places this entry in /etc/group which you can also insert manually:

```
dba:x:241:
```

Once an entry for the group has been made, you are now ready to add a user of this group to the system.

15.4.2 useradd: Adding a User

The **useradd** command adds new users to the system. All parameters related to the user have to be provided in the command line itself:

```
# useradd -u 210 -g dba -c "THE RDBMS" -d /home/oracle -s /bin/ksh -m oracle
# _
```

This quietly creates the user oracle with a UID of 210 and group name dba. The home directory is /home/oracle, and the user will use the Korn shell. The -m option ensures that the home directory is created if it doesn't already exist and copies a sample .profile and .kshrc to the user's home directory. The line **useradd** creates in /etc/passwd is shown below:

```
oracle:x:210:241:THE RDBMS:/home/oracle:/bin/ksh
```

useradd also sets up the user's mailbox and sets the MAIL variable to point to that location (in /var/mail or /var/spool/mail). You now have to set the new user's password with the command **passwd oracle**. Once all this is done, the oracle user account is ready for use.

15.4.3 /etc/passwd and /etc/shadow: User Profiles

All user information except the password encryption is now stored in /etc/passwd. This file contained the password once, the reason why it continues to be known by that name. The encryption itself is stored in /etc/shadow. This is now the control file used by **passwd** to ascertain the legitimacy of a user's password.

Let's take the line pertaining to oracle in /etc/passwd. There are seven fields here, and their significance is noted below (in the order they appear in /etc/passwd):

- Username—The name you use to log on to a UNIX system (oracle).
- Password—No longer stores the password encryption but contains an x.
- UID—The user's numerical identification (210). No two users *should* have the same UID. **ls** prints the owner's name by matching the UID obtained from the inode with this field.
- GID—The user's numerical group identification (241). This number is also the third field in /etc/group.
- Comment or GCOS—User details, e.g., her name, address and so forth (The RDBMS). This name is used at the front of the email address for this user. Any mail sent from this user account will show the sender as *"The RDBMS" <oracle@heavens.com>*—assuming that the user belongs to the domain shown.

- Home directory—The directory where the user ends up on logging in (`/home/oracle`). The **`login`** program reads this field to set the variable `HOME`.
- Login shell—The first program executed after logging in. This is usually the shell (**`/bin/ksh`**). **`login`** sets the variable `SHELL` by reading this entry, and also fork-execs the shell process *(9.4.1)*.

For every line in `/etc/passwd`, there's a corresponding entry in `/etc/shadow`. The relevant line in this file could look something like this:

```
oracle:PR1hjiDhRM2Lg:12032::::::
```

The password encryption is shown in the second field. It's impossible to generate the password from this encryption. However, an intelligent hacker can use an encryption algorithm to generate a sequence of encrypted patterns. It's quite possible that she might just find a match, so this file must be made unreadable to all but the superuser.

Note: The last field in `/etc/passwd` is actually the command to be executed when a user logs in. This is usually the shell, but the administrator may choose a different program to restrict the user's actions.

15.4.4 usermod and userdel: Modifying and Removing Users

`usermod` is used for modifying some of the parameters set with **`useradd`**. Users sometimes need to change their login shell, and the following command line sets Bash as the login shell for the user oracle:

```
usermod -s /bin/bash oracle
```

Users are removed from the system with **`userdel`**. The following command removes the user oracle from the system:

```
userdel oracle                                    Doesn't delete user's files
```

This removes all entries pertaining to oracle from `/etc/passwd`, `/etc/group` and `/etc/shadow`. The user's home directory doesn't get deleted in the process and has to be removed separately if required.

15.5 STARTUP AND SHUTDOWN

Startup After a machine is powered on, the system looks for all peripherals and then goes through a series of steps that may take up to a few minutes to complete the boot cycle. The exact sequence is system-dependent, but the first major event is the loading of the kernel (**`/kernel/genunix`** in Solaris and **`/boot/vmlinuz`** in Linux) into memory. The kernel then spawns **`init`** (PID 1) which, in turn, spawns further processes. Some of these processes monitor all of the terminal lines, activate the network and printer. Eventually, **`init`** becomes the parent of all shells.

A UNIX system boots to a specific **state** (or mode), and this state is represented by a number or letter, called the **run level**. The default run level as well as the action to take for each run level are controlled by **`init`**. We'll consider **`init`**'s role in detail later *(25.8)*, but as of now you should know these two states:

- **Single-user** mode—This mode is important for the administrator, who uses it to perform his administrative tasks, like checking or backing up individual file systems. Other users are prevented from operating the system in single-user mode.
- **Multiuser** mode—In this mode, individual file systems are mounted *(25.6)*, and system daemons *(9.3)* are also started. Printing is possible only in multiuser mode and when the **lpsched** daemon is running.

The **who -r** command displays the run level for your system:

```
$ who -r
   .       run-level 3  Jan  9 09:39     3      0  S
```

This machine is at run level 3, a state which supports multiuser and network operations. We'll have more to discuss about run levels and the role of **init** in Part II of the text.

Shutdown The administrator also has the duty of shutting down the machine at the end of the day (if it is ever shut down). The **shutdown** command controls this sequence. **shutdown** usually performs the following activities:

- Notifies users with **wall** about the system going down with a directive to log out. Users are then expected to close their files and log out. **shutdown** itself sleeps for a minute after mailing the first message and may issue a reminder or two.
- Sends signals to all running processes so they can terminate normally.
- Logs users off and kills remaining processes.
- Unmounts *(25.6)* all secondary file systems.
- Writes information about file system status to disk *(25.9)* to preserve the integrity of the file system.
- Notifies users to reboot or switch off, or moves the system to single-user mode.

shutdown finally displays a message that could look something like these:

```
Reboot the system now or turn power off
System halted
```

The machine can now be considered to have completed the shutdown sequence successfully. You can now turn the power off unless your machine supports a power management feature that does this job automatically.

The -g option to **shutdown** overrides the default waiting time of one minute. The command can be used in these ways:

```
shutdown -g2                    Powers down machine after 2 minutes
shutdown -y -g0                 Immediate shutdown
shutdown -y -g0 -i6             Shut down and reboot
```

Some systems like Solaris have the **reboot** and **halt** commands that also shut the system down without warning the users. Unless you know what you are doing, you should stick to **shutdown** if you are administering a multiuser system.

LINUX: Linux uses the `-t` option to override the default waiting time of one minute. **`shutdown`** can also be used in these ways:

```
shutdown 17:30                    Shut down at 17:30 hours
shutdown -r now                   Shut down immediately and reboot
shutdown -h now                   Shut down immediately and halt
```

Linux also permits the use of the Windows-styled *[Ctrl][Alt][Del]* sequence to shut down the system.

15.6 MANAGING DISK SPACE

No matter how many disks are added to the system, there will always be a scramble for space. Users often forget to remove the files they no longer require. Files tend to accumulate during the day, thus slowing down the system. If this buildup is not checked, the entire disk space will eventually be eaten up. The administrator must regularly scan the disk and locate files that have outlived their utility. He needs the **`df`** and **`du`** commands for this task as well as **`find`** that has already been discussed *(11.7)*. All three commands can also be issued by any user.

15.6.1 df: Reporting Free Space

You are aware that your operating system is supported by multiple file systems *(11.1)*. The **`df`** (disk free) command reports the amount of free space available for each file system separately:

```
# df
/               (/dev/dsk/c0t0d0s0 ): 3491876 blocks   483932 files
/usr            (/dev/dsk/c0t0d0s4 ): 2434820 blocks   458466 files
/proc           (/proc             ):       0 blocks    15875 files
/dev/fd         (fd                ):       0 blocks        0 files
/etc/mnttab     (mnttab            ):       0 blocks        0 files
/var            (/dev/dsk/c0t0d0s1 ): 3881394 blocks   484212 files
/var/run        (swap              ): 2602128 blocks   109840 files
/users1         (/dev/dsk/c0t8d0s0 ): 8037576 blocks  1024196 files
/users2         (/dev/dsk/c0t9d0s0 ): 4920276 blocks   956432 files
/tmp            (swap              ): 2602128 blocks   109840 files
/export/home    (/dev/dsk/c0t0d0s7 ): 2187782 blocks   340677 files
```

There are several file systems on this Solaris machine, but you don't have control over all of them (like `/proc` and `/etc/mnttab`). The first column shows the directory where the file system is attached (mounted). The second column shows the device name of the file system. The last two columns show the number of 512-byte blocks available and the number of files that you can create.

The first line in the list refers to the root file system (/), which has 3,491,876 blocks of disk space free. It also has 483,932 inodes free, which means that up to that many additional files can be created on this file system. The system will continue to function until the free blocks or inodes are eaten away, whichever occurs earlier.

The `-t` (total) option includes the above output, as well as the total amount of disk space in the file system. We won't display its output, but we'll consider the informative `-k` option that reports in units of KB. This time, let's obtain the statistics for the / and `/usr` file systems:

```
$ df -k / /usr                                          Reports on / and /usr file systems
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c0t0d0s0    1986439  240501 1686345    13%    /
/dev/dsk/c0t0d0s4    2025076  807666 1156658    42%    /usr
```

You probably won't need to know anything more than what this output offers. It shows the percentage utilization also. Once you have identified the file system that needs to be investigated thoroughly, you need the **du** command that is considered next.

LINUX: The default output itself is quite informative; it shows both the total as well as the available space, but in a format that resembles the `-k` option used by UNIX. The `-h` option makes it even more readable by reporting in larger units (like MB, GB, etc):

```
$ df -h / /download
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda10            4.8G  2.6G  1.9G  57% /
/dev/hda13            5.6G  985M  4.3G  18% /download
```

The block size used for reporting is usually different in Linux. **find**, **df** and **du** report in 1024-byte blocks.

Note: When the space in one file system is totally consumed, the file system can't borrow space from another file system.

15.6.2 du: Disk Usage

You'll often need to find out the consumption of a specific directory tree rather than an entire file system. **du** (disk usage) is the command you need as it reports usage by a recursive examination of the directory tree. This is how **du** lists the usage of `/home/sales/tml`:

```
# du /home/sales/tml
11554    /home/sales/tml/forms
12820    /home/sales/tml/data
638      /home/sales/tml/database
.....
25170    /home/sales/tml                          Also reports a summary at end
```

By default, **du** lists the usage of each subdirectory of its argument, and finally produces a summary. The list can often be quite large, and more often than not, you may be interested only in a summary. For this, the `-s` (summary) option is quite convenient:

```
# du -s /home/sales/tml
25170   /home/sales/tml
```

Assessing Space Consumed by Users Most of the dynamic space in a system is consumed by users' home directories and data files. You should use **du -s** to report on each user's home directory. The output is brief and yet quite informative:

```
# du -s /home/*
144208    /home/henry
98290     /home/image
```

```
13834     /home/local
28346     /home/sales
```

du can also report on each file in a directory (`-a` option), but the list would be too big to be of any use. You may instead look for some of the notorious disk eaters, and exceptional reporting is what you probably need. The **find** command can do this job.

Note: You have used **find** before *(11.7)*; the command uses the `-size` option to locate large files. Multiple selection criteria can also be specified:

```
find /home -size +2048 -print                      Files above 1 MB
find /home -size +2048 -size -8192 -print          Above 1 MB and below 4 MB
```

You need to halve these figures when using Linux as **find** there uses 1024-byte blocks.

15.7 DEVICE FILES

Devices are also files, and you have seen some of these filenames in the **df** output. You open a device, read and write to it, and then close it like you do to any file. The functions for doing all this are built into the kernel for each and every device of the system. The same device can often be accessed with several different filenames. This has sometimes been done for backward compatibility and sometimes for associating a separate device with a specific function.

All device files are stored in /dev or in its subdirectories. The device names seen on one UNIX system and Linux are shown in Table 15.1. Though the device names in Linux have been known to be remarkably invariant, the same can't be said of the System V devices. If you are using Solaris or HP-UX, the names will be different. Here's a concise listing of these devices for a system running System V:

```
# ls -l /dev
total 52
brw-rw-rw-  1 root     sys      51,  0 Aug 31 07:28 cd0          CD-ROM
brw-rw-rw-  2 bin      bin       2, 64 Feb 23  1997 fd0          Default floppy drive
brw------   1 sysinfo  sysinfo   1,  0 May  7  1996 hd00         First hard disk
crw------   2 bin      bin       6,  0 Dec  5 14:12 lp0          Printer
cr--r--r--  1 root     root     50,  0 Aug 31 07:28 rcdt0 Tape drive
crw------   1 henry    terminal  0,  0 Oct 15 10:23 tty01 Terminal
crw-rw-rw-  2 bin      bin       5,  0 May  7  1996 tty1a Serial port 1
```

SVR4 also has two additional directories—/dev/dsk and /dev/rdsk containing some more files. The files in those directories sometimes have equivalents (or even links) in /dev. The lists in real life are much larger than this and includes every possible device on your system—including even the main memory of your computer. This listing reveals two vital points:

- Device files can be grouped into mainly two categories depending on the first character of the permissions field (`b` or `c`).
- The fifth field—normally representing the size for other files—consists of a pair of numbers. A device file contains no data.

Device files also have permissions with the same significance. To send output to a terminal, you need to have write permission for the device, and to read a floppy, you must have read permission for the device file. The significance of the device attributes is taken up next.

15.7.1 Block and Character Devices

First, a word about disk reading and writing. When you issue an instruction to save a file, the write operation takes place in chunks or blocks. Each block here represents an integral number of disk sectors (of 512 bytes each). The data is first transferred to a **buffer cache** (a pool of buffers) which the kernel later writes to disk. When you read from disk, the buffer cache is first accessed containing the most recently used data. If the data is found there, disk access is avoided. You may also decide to ignore this facility and access the device directly. Many devices allow you to do that, and the access method is determined by the name of the device that is called up.

Note that generally the first character in the permissions field is `c` or `b`. The floppy drive, CD-ROM and hard disk have `b` prefixed to their permissions. All data are read and written to these devices in blocks and use the buffer cache. That's why they are referred to as **block special** devices. On the other hand, the terminal, tape drive and printer are **character special** or raw devices, indicated by the letter **c**. For the latter, the read/write operations ignore the buffer cache and access the device directly.

Many devices have both a raw and a block counterpart in System V. Hard disks, floppy drives and CD-ROMs are both block and character devices. Generally, an `r` prefixed to a block device name makes it a character device. Block devices are also found separately in `/dev/dsk` and character devices in `/dev/rdsk`.

15.7.2 Major and Minor Numbers

The set of routines needed to operate a specific device is known as the **device driver**. When a particular device is accessed, the kernel calls the right device driver and passes some parameters for it to act properly. The kernel must know not only the type of device but also certain details about the device—like the density of a floppy or the partition number of the disk.

The fifth column of the previous listing doesn't show the file size in bytes, but rather a pair of two numbers separated by a comma. These numbers are called the **major** and **minor device numbers**, respectively. The major number represents the device driver; this is actually the type of device. All hard disks will have the same major number if they are attached to the same controller.

The minor number is indicative of the parameters that the kernel passes to the device driver. Often, it indicates the special characteristics of the device. For example, `fd0h1440` and `fd1h1440` represent two floppy devices attached to a particular controller. So both of them have the same major number but different minor numbers.

You'll see a lot of 0s, 1s and other digits in the device names. Often, you'll be able to identify a device from its filename. System V device files are resident in `/dev/dsk` (and `/dev/rdsk` for raw devices). The name `/dev/dsk/f0q18dt` represents a 3.5" floppy (block) device. It is bootable (0), of quad density and has 18 sectors/track—a 1.44 MB diskette. There is also a default name for many devices; the floppy drive can be accessed by `/dev/fd0`.

Note: Unlike ordinary and directory files, device files don't contain any data. They merely point to the location of the device driver routines that actually operate the device.

Table 15.1 Typical Device Names (Directory: /dev)

SVR4 Device	*Linux Device*	*Significance*
cd0 or dsk/c0t6d0s2	cdrom	CD-ROM
fd0 or diskette	fd0	Default floppy drive
dsk/f0q18dt	fd0H1440	1.44 MB floppy
rdsk/f0q18dt	fd0H1440	1.44 MB raw floppy
hd00 or dsk/c0t0d0s2	hda	First hard disk
hd10 or dsk/c1t3d0s2	hdb	Second hard disk
lp0	lp0	Printer
rcdt0 or rmt/0	st0	Tape drive
term/1	tty1	Terminal
tty1a	cua0	Serial port 1
tty2A	ttyS1	Modem port 2

15.8 HANDLING FLOPPY DISKETTES

Although the tape is the most common backup device, the floppy diskette represents the most convenient means of exchanging files between machines at work and home. For our examples, we'll be using the 3.5", 1.44 MB diskette.

15.8.1 format and fdformat: Formatting Diskettes

Before you use a floppy for backup purposes, you need to format it. This is done with the **format** or **fdformat** commands (whichever is available on your system):

```
# fdformat                                          On Solaris
Press return to start formatting floppy.
```

The `-d` option uses the DOS format.

This command formats and verifies a 1.44 MB floppy. Other systems may require you to specify the raw device name as argument:

```
format /dev/rdsk/f0q18dt          On another UNIX system running System V
```

LINUX: Linux too uses the **fdformat** command for formatting a floppy. Device names in Linux generally don't vary across the different flavors; a floppy usually has the name /dev/fd0 or /dev/fd0h1440, so you should use **fdformat /dev/fd0** (or the other device name).

15.8.2 dd: Copying Diskettes

dd (disk dump) is a versatile command that can be used to perform a variety of tasks. It is somewhat dated as some of its filtering functions have been taken over by other UNIX tools. It can be invoked by any user but is really the administrator's tool. It has a strange command line having a series of options in the form *option=value*.

dd was extensively used in copying file systems, but today its role is mainly restricted to copying media—like floppies and tapes. It is not interactive (in fact, a filter), and a pair of **dd** commands is needed to complete the operation.

We'll now use **dd** to make a copy of a 1.44 MB floppy diskette. The first step is to create the image of the floppy on disk:

```
# dd if=/dev/rdsk/f0q18dt of=$$ bs=147456
10+0 records in
10+0 records out
```

The keywords are `if=` (input filename), `of=` (output filename) and `bs=` (block size). The above command copies the raw contents of a 1.44 MB floppy to a temporary file, `$$`, using a block size of 147456—exactly one-tenth the capacity of a 1.44 MB diskette.

Next, take out the source floppy from the drive and insert a formatted target floppy. A second reversed **dd** command copies this temporary file to the diskette:

```
# dd if=$$ of=/dev/rdsk/f0q18dt bs=147456 ; rm $$
10+0 records in
10+0 records out
```

You should copy your boot floppies in this way. In the same manner, you can copy a tape, but if there are two tape drives, a single **dd** command can do the job:

```
dd if=/dev/rct0 of=/dev/rct1 bs=9k
```

dd uses only raw devices—those in `/dev/rdsk` or the ones in `/dev` beginning with an `r`—like `/dev/rdiskette` or `/dev/rct0`. Linux doesn't have separate devices for the two modes but selects the right mode automatically.

15.8.3 Handling DOS Diskettes

It is now quite common to see both Windows and UNIX systems on the desktop. UNIX today provides a family of commands (Table 15.2) that can read and write DOS floppy diskettes. These command names begin with the string `dos` in SVR4. They are modeled after UNIX commands performing similar functions.

The command required most is **doscp**, which copies files between disk and diskette:

```
doscp emp.lst /dev/dsk/f0q18dt:/per.lst
```

There are two components in the target specification—the device name (1.44 MB floppy drive) and the filename (`/per.lst`), with the `:` used as delimiter. As in **cp**, multiple file copying is also possible:

```
doscp emp[123].lst /dev/dsk/f0q18dt
```

doscat performs a simple "cat" of its arguments in the command line. When more than one filename is specified, the standard output for each is concatenated:

```
doscat /dev/dsk/f0q18dt:/CHAP01 /dev/dsk/f0q18dt:/CHAP02 > newchap
```

These commands make the newline conversions automatically *(5.13)*, but they also work with the `-r` option, in which case the files are copied or concatenated without newline conversions.

Table 15.2 shows the use of these commands with varying device names. One of them should work on your system. If `a:` and `b:` don't work, then use the appropriate filename in `/dev` or `/dev/dsk`.

LINUX: The Linux "DOS" commands begin with the string `m`, and use the corresponding DOS command as the rest of the string. Here are some examples:

```
mcopy emp.lst a:
mcopy a:* .
mdir a:
mdel a:*.txt
```

Note that Linux uses the DOS drive name. All of these commands belong to the "mtools" collection. For details, use **`man mtools`**.

Table 15.2 The Family of DOS Commands (Linux command name in parentheses)

Command	*Action*
`doscp /dev/fd0135ds18:/tags .`	Copies `tags` from DOS diskette (`mcopy`)
`doscat a:readme a:setup.txt`	Concatenates files `readme` and `setup.txt` in DOS diskette (`mtype`)
`dosdir /dev/dsk/f0q18dt`	Lists files in DOS diskette in DOS-style (`mdir`)
`dosls /dev/dsk/f0q18dt`	Lists files in UNIX **`ls`**-style
`dosmkdir a:bin`	Creates directory `bin` on DOS diskette (`mmd`)
`dosrmdir a:bin`	Removes directory `bin` on DOS diskette (`mrd`)
`dosrm /dev/dsk/f0q18dt:setup.inf`	Deletes file `setup.inf` on DOS diskette (`mdel`)
`dosformat b:`	Formats diskette in nonbootable drive for use on DOS systems (`mformat`)

15.9 cpio: A BACKUP PROGRAM

The importance of performing regular backups isn't usually appreciated until a crash has occurred and a lot of data has been lost. As administrator, you are partly responsible for the safety of the data that resides on the system. It is part of your duties to decide which files should be backed up and also to determine the periodicity of such backups. The effectiveness of the backup is determined by your ability to easily restore lost or corrupted data files. For reasons of security, backup media of sensitive data are often kept at distant locations.

We'll consider two backup programs in this chapter—**cpio** and **tar**. Both combine a group of files into an archive *(5.14)*, with suitable headers preceding the contents of each file. The backup device can be a magnetic or a cartridge tape, a floppy diskette, or even a disk file. Small systems, especially workstations, may not have the tape facility, so the floppy drive will be used here to illustrate the features of both commands.

The **cpio** command (copy input-output) copies files to and from a backup device. It uses standard input to take the list of filenames. It then copies them with their contents and headers to the standard output which can be redirected to a file or a device. This means that **cpio** can be (and is) used with redirection and piping.

cpio uses two *key* options, `-o` (output) and `-i` (input), either of which (but not both) must be there in the command line. All other options have to be used with either of these key options. The **cpio** options are shown in Table 15.3. The examples in this section and the next use System V device names. Linux users should use `/dev/fd0h1440`, and Solaris users should use `/dev/rdiskette` as the device names.

15.9.1 Backing Up Files (-o)

Since **cpio** uses only standard input, you can use **ls** to generate a list of filenames to serve as its input. The `-o` key option creates the archive on the standard output, which you need to redirect to a device file. This is how you copy files in the current directory to a 1.44 MB floppy:

```
# ls | cpio -ov > /dev/rdsk/f0q18dt                    Use /dev/fd0 in Linux
array.pl
calendar
cent2fah.pl
convert.sh
xinitrc.sam
276 blocks                                             Total size of the archive
```

The `-v` (verbose) option displays each filename on the terminal while it's being copied. **cpio** needs as input a list of files, and if this list is available in a file, redirection can be used too:

```
cpio -o >/dev/rdsk/f0q18dt < flist
```

Incremental Backups **find** can also produce a file list, so any files that satisfy its selection criteria can also be backed up. You'll frequently need to use **find** and **cpio** in combination to back up selected files—for instance, those that have been modified in the last two days:

```
find . -type f -mtime -2 -print | cpio -ovB >/dev/rdsk/f0q18dt
```

Since the path list of **find** is a dot, the files are backed up with their relative pathnames. However, if it is a /, **find** will use absolute pathnames.

The `-B` option sets the block size to 5120 bytes for input and output, which is 10 times the default size. For higher (or lower) sizes, the `-C` option has to be used:

```
ls *.pl | cpio -ovC51200 >/dev/rdsk/f0q18dt                100 times the default
```

Multivolume Backups When the created archive in the backup device is larger than the capacity of the device, **cpio** prompts for inserting a new diskette into the drive:

```
# find . -type f -print | cpio -ocB >/dev/rdsk/f0q18dt
Reached end of medium on output.
If you want to go on, type device/filename when ready
/dev/fd0                                          Device name entered
3672 blocks
```

Enter the device name when **cpio** pauses to take input. In this way, an archive can be split into several extents (volumes).

15.9.2 Restoring Files (-i)

A complete archive or selected files can be restored with the `-i` key option. To restore files, use redirection to take input from the device:

```
# cpio -iv < /dev/rdsk/f0q18dt
array.pl
calendar
cent2fah.pl
convert.sh
xinitrc.sam
276 blocks
```

When restoring subdirectories, **cpio** assumes that the subdirectory structures are also maintained on the hard disk; it can't create them in case they are not. However, the `-d` (directory) option overrides that.

cpio also accepts a quoted wild-card pattern, so multiple files fitting the pattern can be restored. Restoring only the shell scripts becomes quite easy:

```
cpio -i "*.sh" < /dev/rdsk/f0q18dt
```

Tip: A file is restored in that directory that matches its pathname. In other words, if a file has been backed up with the absolute pathname (e.g., `/home/romeo/unit13`), then it will be restored in the same directory (`/home/romeo`). However, when relative pathnames are used, files can be restored anywhere. The "relative filename" method is normally recommended because the administrator often likes to back up files from one directory and restore them in another. Make sure you use **find** with a dot, rather than a /, to specify the path list when you are using it with **cpio**. This applies to the **tar** command also.

Handling Modification Times (-m) By default, when a file is extracted, its modification time is set to the time of extraction. This could lead to problems as this file will participate in future incremental backups even though it has actually not been modified after restoration. Instead of using **touch** *(11.6.1)* to change the modification times (an impractical solution), you can use the `-m` option to tell `cpio` that the modification time has to be retained.

cpio compares the modification time of a file on the media with the one on disk (if any). If the disk file is newer than the copy, or of the same age, then it won't be restored; **cpio** then echoes this message:

```
"current <unit14> newer"
```

This is a useful built-in protection feature that safeguards the latest version of a file. (**tar** doesn't have this feature.) However, this can be overridden with the `-u` (unconditional) option.

Tip: If you are often moving files from one machine to another, use **cpio** instead of **tar**. You are then assured that a newer file on one machine is not overwritten by an older one from another.

15.9.3 Displaying the Archive (`-it`)

The `-t` option displays the contents of the device without restoring the files. This option must be combined with the `-i` key option:

```
# cpio -itv </dev/rdsk/f0q18dt
100755 henry      605  Oct 18 23:34:07 1997  cent2fah.pl
100755 henry      273  Oct 18 23:34:07 1997  check_number.pl
100755 henry      531  Oct 18 23:34:08 1997  dec2bin.pl
100755 henry      214  Oct 18 23:34:08 1997  get_home.pl
```

The files are displayed in a format resembling the listing. (Linux and Solaris output are identical to the listing.) This format shows the octal representation of the file type (10) and the permissions (0755) as well as the modification time of the file (to the nearest second!). Both file type and permissions are stored in the inode as an integral unit, and in Chapter 23 you'll learn to separate them.

Table 15.3 **cpio** Nonkey Options (used with `-i` or `-o` as relevant)

Option	*Significance*
`-d`	Creates directories as and when needed
`-c`	Writes header information in ASCII character form for portability
`-r`	Renames files in interactive manner
`-t`	Lists files in archive (only with `-i` option)
`-u`	Overwrites newer file with older version
`-v`	Verbose option; prints list of files that are being copied
`-m`	Retains original file modification time
`-f` *exp*	Copies all files except those in *exp*
`-C`*size*	Sets input-output block size to *size* bytes
`-A -O` *device*	Appends files to *device* (Solaris and Linux only)
`-H tar`	Creates or reads a tar header format (Solaris and Linux only)
`-E`*file*	Extracts only those files listed in *file* (Solaris and Linux only)

15.9.4 Other Options

There are two important options that can be used with the `-o` and `-i` options:

- The `-r` (rename) option lets you rename each file before starting the copying process. The system presents each filename and prompts you for a response. If you enter a filename, copying is done to that file; a null response leaves the file uncopied.

- The `-f` option, followed by an expression, causes **cpio** to select all files *except* those in the expression:

```
cpio -ivf "*.c" </dev/rdsk/f0q18dt          Restores all except C programs.
```

cpio relies on another command (usually **find**) or a file to provide its file list. It can't accept filename arguments in the command line. A **cpio** archive is also overwritten with every invocation of the command. This is where **tar** comes in.

15.10 tar: THE "TAPE" ARCHIVE PROGRAM

The **tar** (tape archive) command has been in existence since before the emergence of **cpio**. Today, it not only creates archives on tapes but supports floppies as well. Unlike **cpio**, **tar** doesn't normally write to the standard output (though it can be made to), but creates an archive in the media. It is a versatile command with certain exclusive features not found in **cpio**:

- It doesn't use standard input to obtain its file list. **tar** accepts file and directory names as arguments.
- It copies one or more entire directory trees; i.e., it operates recursively by default.
- It can append to an archive without overwriting the entire archive (**cpio** in Solaris and Linux also).

You have already used **tar** with its key options *(5.16)* to handle disk archives. The common key options are `-c` (copy), `-x` (extract) and `-t` (table of contents). The `-f` option additionally has to be used for specifying the device name. The **tar** options are listed in Table 15.4.

15.10.1 Backing Up Files (-c)

tar accepts directory and filenames directly on the command line. The `-c` key option is used to copy files to the backup device. The verbose option (`-v`) shows the progress of backup:

```
# tar -cvf /dev/rdsk/f0q18dt /home/sales/SQL/*.sql
a /home/sales/SQL/invoice_do_all.sql 1 tape blocks
a /home/sales/SQL/load2invoice_do_all.sql 1 tape blocks
a /home/sales/SQL/remove_duplicate.sql 1 tape blocks
a /home/sales/SQL/t_mr_alloc.sql 10 tape blocks
```

The `a` before each pathname indicates that the file is appended. The command backs up all SQL scripts with their absolute pathnames. The same restrictions apply; they can only be restored in the same directory. However, if you choose to keep the option open of installing the files in a different directory, you should first "cd" to `/home/sales/SQL` and then use a relative pathname:

```
cd /home/sales/SQL
tar -cvf /dev/rdsk/f0q18dt ./*.sql          Using the ./
```

The advantage of **tar** lies in that it can copy an entire directory tree with all its subdirectories. The current directory can be backed up with or without the hidden files:

```
tar -cvfb /dev/rdsk/f0q18dt 18 *            Doesn't back up hidden files
tar -cvfb /dev/fd0 18 .                     Backs up hidden files also
```

The files here are backed up with their relative pathnames, assuming they all fit in one diskette. If they don't, **tar** in System V may accommodate them as much as possible and then quit the program without warning.

Tip: If you have backed up your files with absolute pathnames and now want to restore them in another directory, then use the `-C` option.

The command will also execute faster if used with a block size of 18 (i.e., 18X2X512 bytes):

```
tar -cvfb /dev/rdsk/f0q18dt 18 *.sql                    ./ prefix not required
```

Since both `-f` and `-b` have to be followed by an argument, the first word (`/dev/rdsk/f0q18dt`) after the option string `-cvfb` denotes the argument `f` or `-f`, and the second word (18) will line up with `-b`.

Note: **tar** is quite liberal in its handling of options. **tar cvf** is the same as **tar -cvf**. The - symbol is not required at all! However, future versions of **tar** will not support this.

Multivolume Backup (-k) For multivolume diskette backups, **tar** in Solaris (and SCO UNIX) uses a special option (`-k`), followed by the volume size in kilobytes. This is how the file `index` is backed up in SCO UNIX:

```
# tar -cvfkb /dev/rdsk/f0q18dt 1440 18 index
Volume ends at 1439K, blocking factor = 18
tar: large file index needs 2 extents.
tar: current device seek position = OK
+++ a index 1439K [extent #1 of 2]
```

tar estimates that two 1440 KB (the argument to `-k`) diskettes will be required. After the first volume is full, **tar** prompts for a new volume:

```
tar: please insert new volume, then press RETURN.
```

At the time of restoration, the same option has to be used.

15.10.2 Restoring Files (-x)

Files are restored with the `-x` (extract) key option. When no file or directory name is specified, it restores all files from the backup device. The following command restores the files just backed up:

```
# tar -xvfb /dev/rdsk/f0q18dt 18
x /home/sales/SQL/invoice_do_all.sql, 169 bytes, 1 tape blocks
x /home/sales/SQL/load2invoice_do_all.sql, 456 bytes, 1 tape blocks
x /home/sales/SQL/remove_duplicate.sql, 237 bytes, 1 tape blocks
x /home/sales/SQL/t_mr_alloc.sql, 4855 bytes, 10 tape blocks
```

Selective extraction is also possible by providing one or more directory or filenames:

```
tar -xvf /dev/rdsk/f0q18dt /home/sales/SQL/t_mr_alloc.sql
```

Unlike **cpio**, when files are extracted, the modification times of the files also remain unchanged. This can be overridden by the `-m` option to reflect the system time at the time of extraction.

Note: Unlike **cpio**, some versions of **tar** (like in Solaris) don't read wild-card patterns. If you use **tar -xvf /dev/fd0 *.pl**, it's the shell that tries to expand the pattern, which means that the files have to reside in the current directory. However, some versions of **tar** (like in Linux) do permit the use of the wild-cards; it doesn't matter whether the files exist at all in the disk.

15.10.3 Displaying the Archive (-t)

Like in **cpio**, the `-t` key option displays the contents of the device in a long format similar to the listing:

```
# tar -tvf /dev/rdsk/f0q18dt
rwxr-xr-x203/50      472 Jun  4 09:35 1991 ./dentry1.sh
rwxr-xr-x203/50      554 Jun  4 09:52 1991 ./dentry2.sh
rwxr-xr-x203/50     2299 Jun  4 13:59 1991 ./func.sh
```

There's something here that you ought to pay attention to: The files have been backed up with relative pathnames. Each filename here is preceded by `./`. If you don't remember this but want to extract the file `func.sh` from the diskette, you'll obviously first try this:

```
# tar -xvf /dev/rdsk/f0q18dt func.sh
tar: func.sh: Not found in archive
```

tar failed to find the file because it existed there as `./func.sh` and not `func.sh`. Put the `./` before the filename, and get it this time. Remember this whenever you encounter extraction errors as above.

15.10.4 Other Options

There are a number of other options of **tar** that are worth considering:

- The `-r` key option is used to append a file to an archive. This implies that an archive can contain several versions of the same file!
- The `-u` key option also adds a file to an archive but only if the file is not already there or is being replaced with a newer version.
- The `-w` option permits interactive copying and restoration. It prints the name of the file and prompts for the action to be taken (`y` or `n`).
- Some versions of **tar** use a special option to pick up filenames from a file. You might want to use this facility when you have a list of over a hundred files, which is impractical (and sometimes, impossible) to enter in the command line. Unfortunately, this option is not standard; Solaris uses `-I` and Linux uses `-T`.

LINUX: The GNU **tar** command is more powerful than its System V counterpart and supports a host of exclusive options. Unfortunately, there is sometimes a mismatch with the options used by System V. The `-M` option is used for a multivolume backup (e.g. **tar -cvf /dev/fd0H1440 -M ***). There's one option (`-z`) related to compression that we have already discussed *(5.16.3—LINUX)*.

Table 15.4 `tar` Options

Key Options (only one to be used)	
Option	***Significance***
`-c`	Creates a new archive
`-x`	Extracts files from archive
`-t`	Lists contents of archive
`-r`	Appends files at end of archive
`-u`	Like r, but only if files are newer than those in archive

Nonkey Options	
Option	***Significance***
`-f` *device*	Uses pathname *device* as name of device instead of the default
`-v`	Verbose option—lists files in long format
`-w`	Confirms from user about action to be taken
`-b` *n*	Uses blocking factor *n*, where *n* is restricted to 20
`-m`	Changes modification time of file to time of extraction
`-I` *file*	Takes filenames from *file* (Solaris only)
`-T` *file*	Takes filenames from *file* (Linux only)
`-X` *file*	Excludes filenames in *file* (Solaris and Linux only)
`-k` *num*	Multivolume backup—sets size of volume to *num* kilobytes (Solaris only)
`-M`	Multivolume backup (Linux only)
`-z`	Compresses/uncompresses with `gzip` (Linux only)
`--bzip2`	Compresses/uncompresses with `bzip2` *(Linux only)*

15.11 CONCLUSION

We discussed the routine administrative features and tools that you need to use everyday. This chapter didn't provide the forum to discuss what to do when things go wrong. That requires from the administrator an in-depth knowledge of the different components of the system. At a deeper level, system administration involves fixing file systems, controlling the system's services and configuring the network. With people increasingly owning machines that run Linux, it's much easier today for every user to try out these advanced features than previously. For this reason, system administration is worth a second visit, which we make in Part II of this text.

WRAP UP

The system administrator or *superuser* uses the root user account, though any user can also invoke the **su** command to acquire superuser powers. Most administrative commands are resident in `/sbin` and `/usr/sbin`. The current directory doesn't feature in the superuser's PATH.

The administrator can change the attributes of any file and kill any process. The administrator also controls user access to many services like **at**, **cron**, FTP and SSH.