EXCERPT FROM CHAPTER 8:
Linux GUIs: X, GNOME, and KDE

Using GNOME

This section discusses the Nautilus file manager and several GNOME utilities.

Using the Nautilus File Manager

Nautilus is a simple, powerful file manager. You can use it to create, open, view, move, and copy files and directories as well as execute programs and scripts. Nautilus gives you two ways to work with files: an innovative spatial view (*FEDORA*, Figure 8-3) and a traditional file browser view (*RHEL+FEDORA*, Figure 8-5 on page 228).

tip || GNOME Desktop and Nautilus

The GNOME Desktop is run from a backend process that runs as part of Nautilus. If that process stops running, it usually starts up again automatically. If it does not restart, start Nautilus by giving the command **nautilus** to restore your desktop. You do not have to keep the Nautilus window open to keep the desktop alive.

Spatial View

The Nautilus object window presents a spatial view (Figure 8-3); it has many simple, powerful features but may take some getting used to. The spatial (as in having the nature of space) view always provides one window per folder: Open a folder and you get a new window.

Open a spatial view of your home directory by double-clicking the home icon on the upper-left of the desktop and experiment as you read this section. Double-click

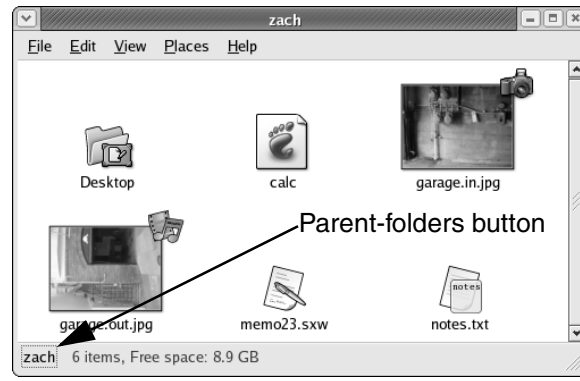

Figure 8-3 The Nautilus spatial view

the desktop icon in the spatial view and Nautilus opens a new window that displays the desktop folder.

A spatial view can display icons or a list of filenames; choose the display you prefer by choosing one of the **View as** selections from **View** on the menubar. To create files to experiment with, right click in the window (not on an icon) to display the Nautilus context menu and select **Create Folder** or **Create Document**.

tip || Use SHIFT to Close the Current Window as You Open Another

If you hold the SHIFT key down when you double-click to open a new window, Nautilus closes the current window as it opens the new one. This behavior may be more familiar and can help keep your desktop from getting cluttered. If you do not want to use the keyboard, you can get the same result by double-clicking the middle mouse button.

Window memory

Move the window by dragging the titlebar. The spatial view has *window memory*. The next time you open that folder, Nautilus opens it at the same size in the same location. Even the scrollbar will be in the same position.

Parent-folders button

The key to closing the current window and returning to the window of the parent directory is the Parent-folders button. See Figures 8-3 and 8-4. Click the Parent-folders button to display the Parent-folders pop-up menu and select the directory you want to open from this menu; Nautilus displays in a spatial view the directory you specified.

From a spatial view, you can open a folder in a traditional view by right clicking the folder and selecting **Browse Folder**.

Traditional View

Figure 8-5 shows the traditional, or file browser, window with a Side Pane (sometimes called a *sidebar*), View Pane, menubar, toolbar, and location bar. Open a browser view of your home directory by right clicking the home icon on the desktop and selecting **Browse Folder** or by selecting **Main menu: Browse Filesystem**.

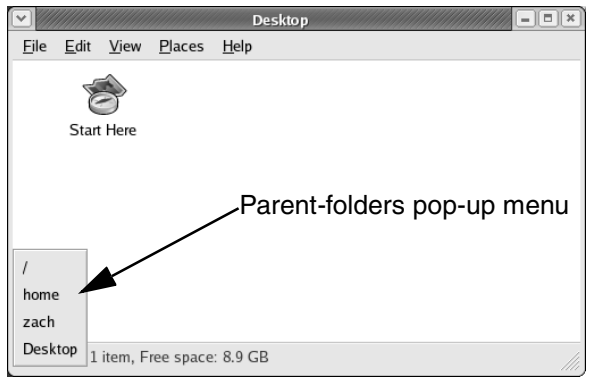

Figure 8-4 The Parent-folders pop-up menu

Side Pane

Click the box at the top of the Side Pane, just under the word **Location** on the Location bar to display the Side Pane menu. The box starts with the word **Information** in it. Select from this menu what you want Nautilus to display in the Side Pane: Information (shown), Emblems (drag emblems [page 231] to files in the View Pane), History (list of recent locations displayed by Nautilus), Notes, or Tree (displays the directory hierarchy).

View Pane

You can display icons or a list of filenames in the View Pane. Choose which you prefer by making a selection from the drop-down menu that appears on the right end of the location bar. **View as Icons** is shown in Figure 8-5 and **View as List** in Figure 8-6 (with the toolbar moved to the bottom of the window).

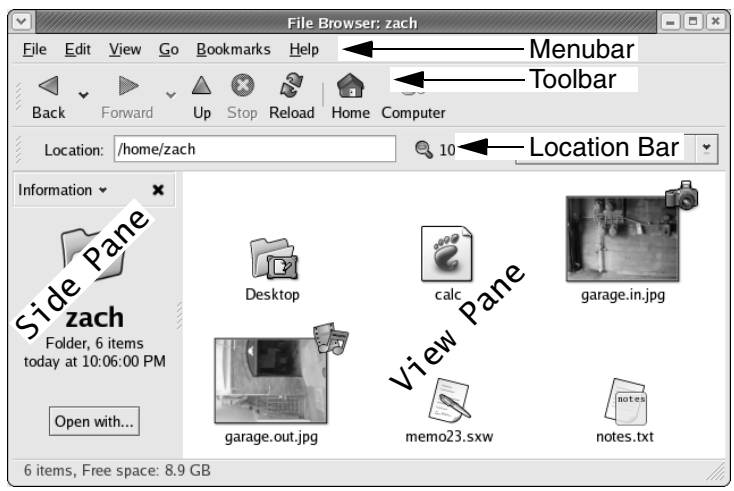

Figure 8-5 Nautilus traditional, or file browser, window

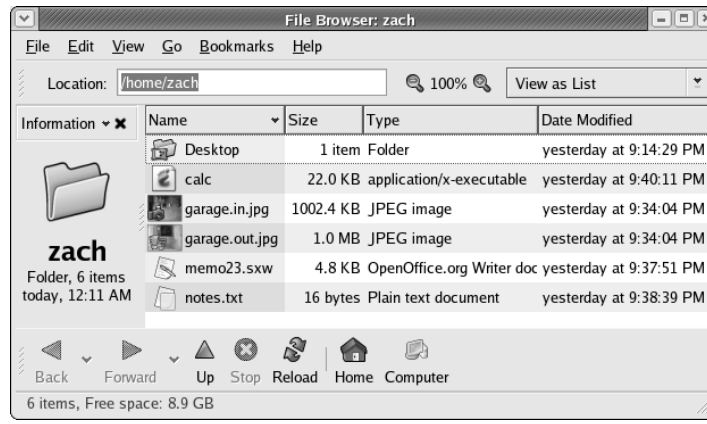

Figure 8-6 Nautilus displaying a list view with the toolbar at the bottom of the window

Control Bars

This section discusses the three control bars—menubar, toolbar, and location bar—that initially appear at the top of a Nautilus browser window (Figure 8-6).

Menubar The menubar presents a drop-down menu when you click one of its selections. Nautilus varies the menu selections, depending on what it is displaying in the View Pane.

Toolbar The Nautilus toolbar holds navigation tool icons: Back, Forward, Up, Stop, Reload, Home, and Computer.

Location bar The Location bar text box displays the pathname of the directory that is displayed in the View Pane and highlighted in the Tree tab (when it is displayed). You can also use the Location bar text box, or press CONTROL-L to display a Location dialog box, to specify a local or remote directory or URL (FTP, HTTP, and so on) of a site that you want to display in the View Pane: Enter the absolute pathname of the directory or URL (an FTP address must be in the form **ftp://*address***) and press RETURN. Nautilus displays the contents of the directory or URL.

The location bar holds two tools in addition to the location bar text box: the magnification selector and the **View as** drop-down menu. To change the magnification of the display in the View Pane, click the plus or minus sign on either side of the magnification percentage; click the magnification percentage itself to return to 100% magnification. Click anywhere on **View as** (to the right of the magnifying glass) to display the viewing choices.

Tear-away bars One handy feature of Nautilus is the ability to reposition, or tear away, the toolbar and location bar. You can move either control bar onto the root window or position it at the top or bottom of the Nautilus window. Figure 8-6 shows the toolbar at the bottom of the Nautilus window. To return a bar to the window, drag it back and it snaps into place. You can also move the toolbar so that it is vertical on either side of

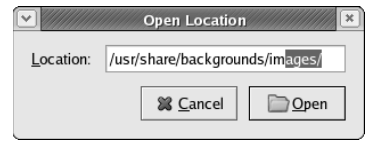

Figure 8-7 Location dialog box

the Nautilus window. Drag the toolbar handle, the striped vertical bar at the left or top end of one of the control bars, and you will drag the bar. To position the toolbar on one of the vertical sides of the window, drag the toolbar handle toward the middle of the side you want it on, and it snaps into place. Once the toolbar is vertical, you can move it to the root window and it will remain vertical.

Features Available from Both Spatial and Traditional Views

Location dialog box

You can display deeply nested directories quickly using the Location dialog box (Figure 8-7). Press CONTROL-L while the cursor is over a Nautilus window (spatial or traditional) to display the Location dialog box. Enter the absolute pathname (starting with a slash) of the directory you want to display. Nautilus helps you type by completing directory names as you go. Press TAB to accept a suggested completion or keep typing to ignore it.

Zoom images

Use the Location dialog box to display the **/usr/share/backgrounds/images** directory. Double-click an image file to display that file in a preview window. Position the mouse pointer over the image and use the mouse wheel to zoom the image.

Opening files

By default, you double-click a file or icon to open it, or you can right click the file and choose **Open** from the pop-up menu. When you open a file, Nautilus tries to figure out which tool to use to open it, using MIME (page 89) to associate the filename extension with a MIME type and a program. For example, when you open a file with a filename extension of **ps**, Nautilus calls gs (ghostscript), which displays the PostScript file in a readable format. When you open an executable file such as Mozilla, Nautilus runs the executable. When you open a text file, Nautilus opens a text editor that displays and allows you to edit the file. When you open a directory, Nautilus displays its contents. When Nautilus does not know which tool to use to open a file, it asks you.

Properties

You can view information about a file, such as ownership, permission, size, and so on, by right clicking any filename or icon and selecting **Properties** from the drop-down menu. The Properties window initially displays some basic information; click the tabs at the top of the window to see additional information. Different types of files display different sets of tabs, depending on what is appropriate to the file (context menu). You can modify settings in this window only if you have permission to do so.

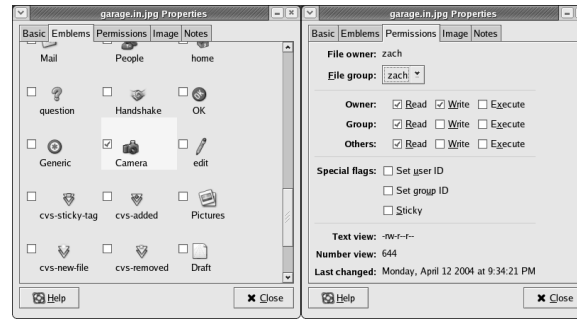

Figure 8-8 Properties window: Emblems tab left; Permissions tab right

Basic The **Basic** tab displays information about the file and enables you to select a custom icon for the file or change its name. To change the name of the file, make your changes in the text box.

Emblems The **Emblems** tab (Figure 8-8, left) allows you to add or remove emblems associated with the file by placing/removing a check mark next to an emblem. Figure 8-9 shows what the Important, New, and Special emblems look like on a file icon. Nautilus displays emblems in both its icon and list views. You can also place an emblem on an icon by dragging it from the Side Pane Emblems tab to an icon in the View Pane.

Permissions The **Permissions** tab (Figure 8-8, right) allows you to change file permissions (page 173). When the **Read** button in the Owner row (*User* elsewhere, see the tip "chmod: **o** for Other, **u** for Owner" on page 175) has a check mark in it, the owner has permission to read from the file. When you click all the buttons in the Owner row so they all contain check marks, the owner has read, write, and execute permission. The owner of a file can change the group that the file is associated with to any other group the owner is associated with. When you run as Superuser, you can change the name of the user who owns the file and the group associated with the file. Directory permissions work as explained on page 176.

Start Here

You can display the Start Here window by double-clicking the Start Here icon on the root window or entering **start-here:** in the Nautilus location bar. Nautilus dis-

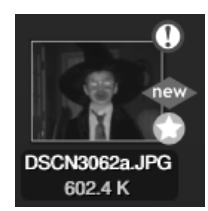

Figure 8-9 Emblems

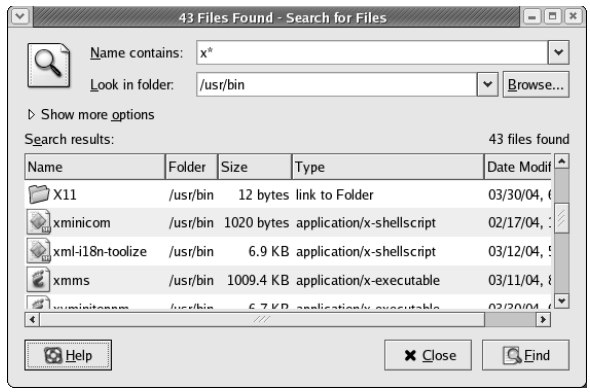

Figure 8-10 GNOME search tool: simple search

plays Start Here as a series of directories. Although Start Here is not strictly a menu, it is the root of a virtual directory structure that can help you in your day-to-day work on the system. In many areas, Start Here functions overlap those of the Main menu. The Preferences folder, a subdirectory of Start Here, duplicates exactly the **Preferences** item on the Main menu. Look around and experiment; you will not be allowed to do anything harmful to the system unless you are logged in as **root**.

EXCERPT FROM CHAPTER 11:
System Administration: Core Concepts

The xinetd Super Server

The **xinetd** daemon is a more secure replacement for the **inetd** super server that originally shipped with 4.3BSD. The Internet super server listens for network connections and, when one is made, launches a specified server daemon and forwards the data from the socket (page 439) to the daemon's standard input. The **xinetd** super server performs these tasks and provides additional features to control access.

The version of **xinetd** distributed with Red Hat Linux is linked against **libwrap.a**, so it is capable of using the **/etc/hosts.allow** and **/etc/hosts.deny** files for access control (see "TCP Wrappers," page 404 for more information). Using TCP Wrappers can simplify configuration but hides some of the more advanced features of **xinetd**.

The base configuration for **xinetd** is stored in the **/etc/xinetd.conf** file. The file supplied by Red Hat is shown following:

```
$ cat /etc/xinetd.conf
# Simple configuration file for xinetd
defaults
{
        instances               = 60
        log_type                = SYSLOG authpriv
        log_on_success          = HOST PID
        log_on_failure          = HOST
        cps                     = 25 30
}
includedir /etc/xinetd.d
```

The **defaults** section specifies the default configuration of **xinetd**; the files in the included directory, **/etc/xinetd.d**, specify server-specific configurations. Defaults can be overridden by server-specific configuration files.

In the preceding file, the **instances** directive specifies that no daemon may run more than 60 copies of itself at one time. The **log_type** directive specifies that **xinetd** send messages to the system log daemon (**syslogd**, page 546) using the **authpriv** facility. The next two lines specify what to log on success and on failure. The **cps** (connections per second) directive specifies that no more than 25 connections to a specific service should be made per second and that the service should be disabled for 30 seconds if this limit is exceeded.

The following **xinetd** configuration file allows telnet connections from the local system and any system with an IP address that starts with **192.168.**. This configuration file does not rely on TCP wrappers and so does not rely on the **hosts.allow** and **hosts.deny** files.

```
$ cat /etc/xinetd.d/telnet
service telnet
{
        socket_type     = stream
        wait            = no
        user            = root
        server          = /usr/sbin/in.telnetd
        only_from       = 192.168.0.0/16 127.0.0.1
        disable         = no
}
```

The **socket_type** indicates whether the socket uses TCP or UDP. TCP-based protocols establish a connection between the client and the server and are identified by the type **stream**. UDP-based protocols rely on the transmission of individual datagrams and are identified by the type **dgram**.

When **wait** is set to **no**, **xinetd** handles multiple, concurrent connections to this service. Setting **wait** to **yes** causes **xinetd** to wait for the server process to complete before handling the next request for that service. In general, UDP services should be set to **yes** and TCP services to **no**. If you were to set **wait** to **yes** for a service such as telnet, only one person would be able to use the service at a time.

The **user** specifies the user that the server runs as. If the user is a member of multiple groups, you can also specify the group on a separate line using the keyword **group**. The **user** directive is ignored if **xinetd** is run as other than **root**. The **server** provides the pathname of the server program that **xinetd** runs for this service.

The **only_from** specifies which systems **xinetd** allows to use the service. Use IP addresses only because using hostnames can make the service unavailable if DNS fails. Zeros at the right of an IP address are treated as wildcards: **192.168.0.0** allows access from any system in the **192.168** subnet.

The **disable** line can disable a service without removing the configuration file. As shipped by Red Hat, a number of services include an **xinetd** configuration file with **disable** set to **yes**. To run one of these services, you must change **disable** to **no** in the appropriate file in **xinetd.d** and restart **xinetd**:

```
# /sbin/service xinetd restart
Stopping xinetd:                                           [  OK  ]
Starting xinetd:                                           [  OK  ]
```

Securing a Server

This section discusses how to secure a server using TCP wrappers or by setting up a chroot jail.

TCP Wrappers: Client/Server Security (hosts.allow and hosts.deny)

When you open a local system to access from remote systems, you must ensure that you

- Open the local system only to systems you want to allow to access it.
- Allow each remote system to access only the data you want it to access.
- Allow each remote system to access data only in the manner you want it to (read only, read/write, write only).

As part of the client/server model, TCP Wrappers, which can be used for any daemon that is linked against **libwrap.a**, uses the **/etc/hosts.allow** and **/etc/hosts.deny**

files to form the basis of a simple access control language. This access control language defines rules that selectively allow clients to access server daemons on a local system based on the client's address and the daemon the client tries to access.

Each line in the **hosts.allow** and **hosts.deny** files has the following format:

daemon_list : client_list [: command]

where ***daemon_list*** is a comma-separated list of one or more server daemons (such as **portmap**, **vsftpd**, **sshd**), ***client_list*** is a comma-separated list of one or more clients (see Table 11-3, "Specifying a Client," on page 399), and the optional ***command*** is the command that is executed when a client from ***client_list*** tries to access a server daemon from ***daemon_list***.

When a client requests a connection with a local server, the **hosts.allow** and **hosts.deny** files are consulted as follows until a match is found. The first match determines whether the client is allowed to access the server.

1. If the daemon/client pair matches a line in **hosts.allow**, access is granted.
2. If the daemon/client pair matches a line in **hosts.deny**, access is denied.
3. If there is no match in either the **hosts.allow** or the **hosts.deny** files, access is granted.

When either the **hosts.allow** or the **hosts.deny** file does not exist, it is as though that file were empty. Although it is not recommended, you can allow access to all daemons for all clients by removing both files.

Examples For a more secure system, put the following line in **hosts.deny** to block all access:

```
$ cat /etc/hosts.deny
...
ALL : ALL : echo '%c tried to connect to %d and was blocked' >> /var/log/tcpwrappers.log
```

This line blocks any client attempting to connect to any service, unless specifically permitted in **hosts.allow**. When this rule is matched, it adds a line to the **/var/log/tcpwrappers.log** file. The **%c** expands to client information and the **%d** expands to the name of the daemon the client attempted to connect to.

With the preceding **hosts.deny** file in place, you can put lines in **hosts.allow** explicitly to allow access to certain services and systems. For example, the following **hosts.allow** file allows anyone to connect to the OpenSSH daemon (ssh, scp, sftp) but allows telnet connections only from the same network as the local system and users on the 192.168. subnet:

```
$ cat /etc/hosts.allow
sshd : ALL
in.telnet : LOCAL
in.telnet : 192.168.? 127.0.0.1
...
```

The first line in the preceding file allows connection from any system (ALL) to **sshd**. The second line allows connection from any system in the same domain as the server (LOCAL). The third line matches any system whose IP address starts **192.168.** and the local system.

Setting Up a chroot Jail

On early UNIX systems, the root directory was a fixed point in the file system. On modern UNIX variants, including Linux, you can define the root directory on a per-process basis. The chroot utility allows you to run a process with a root directory other than **/**.

The root directory is the top of the directory hierarchy and has no parents: A process cannot access any files above the root directory (because they do not exist). If, for example, you run a program (process) and specify its root directory as **/home/sam/jail**, the program would have no concept of any files in **/home/sam** or above: **jail** is the program's root directory and is labeled **/** (not **jail**).

By creating an artificial root directory, frequently called a (chroot) jail, you prevent a program from being able to access and modify, possibly maliciously, files outside the directory hierarchy starting at its root. You must set up a chroot jail properly in order to increase security: If you do not set up a chroot jail correctly, you can make it easier for a malicious user to gain access to a system than if there were no chroot jail.

Using chroot

Creating a chroot jail is simple: Working as **root**, give the command **/usr/sbin/chroot** ***directory***. The ***directory*** becomes the root directory and the process attempts to run the default shell. Working as **root** from the **/home/sam** directory, the following example attempts to set up a chroot jail in the (existing) **/home/sam/jail** directory:

```
# /usr/sbin/chroot jail
/usr/sbin/chroot: /bin/bash: No such file or directory
```

In the preceding example, chroot sets up the chroot jail, but when it attempts to run a the bash shell, it fails. Once the jail is set up, the directory that was named **jail** takes on the name of the root directory: **/** and chroot cannot find the file identified by the pathname **/bin/bash**. The chroot jail is working perfectly but is not very useful.

Getting a chroot jail to work the way you want is more complicated. To get the preceding example to run bash in a chroot jail, you need to create a **bin** directory in **jail** (**/home/sam/jail/bin**) and copy **/bin/bash** to this directory. Because the bash binary is dynamically linked to shared libraries (page 813), you need to copy these libraries into **jail** too. The libraries go in **/lib**. The following example creates the necessary directories, copies **bash**, uses ldd to display the shared library dependencies of bash, and copies the necessary libraries into **lib**:

```
$ pwd
/home/sam/jail
$ mkdir bin lib
$ cp /bin/bash bin
$ ldd bin/bash
        libtermcap.so.2 => /lib/libtermcap.so.2 (0x4002a000)
        libdl.so.2 => /lib/libdl.so.2 (0x4002f000)
        libc.so.6 => /lib/tls/libc.so.6 (0x42000000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$ cp /lib/libtermcap.so.2 lib
$ cp /lib/libdl.so.2 lib
$ cp /lib/tls/libc.so.6 lib
$ cp /lib/ld-linux.so.2 lib
```

Now that everything is set up, you can try starting the chroot jail again. All the setup can be done by an ordinary user; you have to run chroot as Superuser:

```
$ su
Password:
# /usr/sbin/chroot .
bash-2.05b# pwd
/
bash-2.05b# ls
bash: ls: command not found
bash-2.05b#
```

This time the chroot finds and starts **bash**, which displays its default prompt (**bash-2.05b#**). The pwd command works because it is a shell builtin (page 211). However, bash cannot find the ls utility (it is not in the chroot jail). You can copy **/bin/ls** and its libraries into the jail if you want users in the jail to be able to use ls.

To set up a useful chroot jail, you need to determine which utilities the users of the chroot jail are going to need. Then, you can copy the binaries and their libraries into the jail, or you can build static copies of the binaries and put them in the jail without the need for separate libraries. (The statically linked binaries are considerably larger than their dynamic counterparts. The base system with bash and the core utilities is over 50 megabytes.) You can find the source code for most of the common utilities you need in the **bash** and **coreutils** SRPMS (source rpm) packages.

Whichever technique you choose, you must put a copy of su in the jail. The su command is required to run programs as a user other than **root**. Because **root** can break out of a chroot jail, it is imperative that you run a program in the chroot jail as a user other than **root**.

The dynamic version of su that Red Hat distributes requires PAM and will not work within a jail. You need to build a copy of su from the source to use in a jail. By default, a copy of su you build does not require PAM. Refer to "GNU Configure and Build System" on page 459 for instructions on how to build packages such as **coreutils** (which includes su).

In order to be able to use su, you need to copy the relevant lines from the **/etc/passwd** and **/etc/shadow** files into files with the same names in the **etc** directory inside the jail.

tip || Keeping Multiple chroot Jails

If you are going to deploy multiple chroot jails, it is a good idea to keep a clean copy of the **bin** and **lib** files somewhere other than in one of the active jails.

Running a Service in a chroot Jail

Running a shell inside a jail has limited use. You are more likely to need to run a specific service inside the jail. To run a service inside a jail, you need make sure all the files needed by that service are inside the jail. The format of a command to start a service in a chroot jail is

```
# /usr/sbin/chroot jailpath /bin/su user daemonname &
```

Where *jailpath* is the pathname of the jail directory, *user* is the username that runs the daemon, and *daemonname* is the path (inside the jail) of the daemon that provides the service.

Several servers are set up to take advantage of chroot jails. The system-config-bind utility (page 712) automatically sets up DNS so **named** runs in a jail and the **vsftpd** FTP server can automatically start chroot jails for clients (page 598).

Security Considerations

Some services need to be run as **root**, but they release their **root** privilege once started (Procmail and **vsftpd** are examples). If you are running such a service, you do not need to put su inside the jail.

It is possible for a process run as **root** to escape from a chroot jail. For this reason, you should always su to another user before starting a program running inside the jail. Also, be careful about what setuid (page 175) binaries you allow inside a jail because a security hole in one of these can compromise the security of the jail. Also, make sure the user cannot access executable files that she uploads.

DHCP

Instead of having network configuration information stored in local files on each system, DHCP (Dynamic Host Configuration Protocol) enables client systems to retrieve network configuration information each time they connect to the network. A DHCP server assigns temporary IP addresses from a pool of addresses to clients as needed.

This technique has several advantages over storing network configuration information in local files:

- DHCP makes it possible for a new user to set up an Internet connection without having to work with IP addresses, netmasks, DNS addresses, and other technical details. DHCP also makes it faster for an experienced user to set up a connection.
- DHCP facilitates assignment and management of IP addresses and related network information by centralizing the process on a server. A system administrator can configure new systems, including laptops that connect to the network from different locations, to use DHCP, which assigns IP addresses only when each system connects to the network. The pool of IP addresses is managed as a group on the DHCP server.
- DHCP enables IP addresses to be used by more than one system, reducing the number of IP addresses needed overall. This conservation of addresses is important as the Internet is quickly running out of IPv4 addresses. Although one IP address can be used by only one system at a time, many end-user systems require addresses only occasionally, when they connect to the Internet. By reusing IP addresses, DHCP lengthens the life of the IPv4 protocol. DHCP applies to IPv4 only, as IPv6 forces systems to configure their IP addresses automatically (called autoconfiguration) when they connect to a network (page 340).

DHCP is particularly useful for administrators responsible for a large number of systems because it removes the requirement for individual systems to store unique configuration information. DHCP allows an administrator to set up a master system and deploy new systems with a copy of the master's hard disk. In educational establishments and other open access facilities, it is common to store the hard disk image on a shared drive and have each workstation automatically restore itself to pristine condition at the end of each day.

More Information

Web www.dhcp.org

FAQ www.dhcp-handbook.com/dhcp_faq.html

HOWTO *DHCP Mini HOWTO*

How DHCP Works

The client daemon, **dhclient**, contacts the server daemon, **dhcpd**, to obtain the IP address, netmask, broadcast address, nameserver address, and other networking parameters. The server provides a *lease* on the IP address to the client. The client can request specific terms of the lease, including its duration, and the server can limit these terms. While connected to the network, a client typically requests extensions of its lease as necessary so its IP address remains the same. The lease can expire once the client is disconnected from the network; the server can give the client a new IP

address when it requests a new lease. You can also set up a DHCP server to provide static IP addresses for specific clients (refer to "Static versus Dynamic IP Addresses" on page 334).

DHCP is broadcast based, so the client and server must be on the same subnet (page 337).

DHCP Client

A DHCP client requests network configuration parameters from the DHCP server and uses those parameters to configure its network interface.

Prerequisites

Install the following package:

- **dhclient**

dhclient: The DHCP Client

When a DHCP client system connects to the network, dhclient requests a lease from the DHCP server and configures the client's network interface(s). Once a DHCP client has requested and established a lease, it stores information about the lease in **/etc/dhclient.leases**. This information is used to reestablish a lease when either the server or client needs to reboot. The DHCP client configuration file, **/etc/dhclient.conf**, is required only for custom configurations. The following **dhclient.conf** file specifies a single interface, **eth0**:

```
$ cat /etc/dhclient.conf
interface "eth0"
{
send dhcp-client-identifier 1:xx:xx:xx:xx:xx:xx;
send dhcp-lease-time 86400;
}
```

In the preceding file, the 1 in the **dhcp-client-identifier** specifies an Ethernet network and **xx:xx:xx:xx:xx:xx** is the *MAC address* (page 981) of the device controlling that interface. See page 412 for instructions on how to display a MAC address. The **dhcp-lease-time** is the duration, in seconds, of the lease on the IP address. While the client is connected to the network, dhclient automatically renews the lease each time half of the lease is up. The choice of 86400 seconds, or one day, is a reasonable choice for a workstation.

DHCP Server

The DHCP server maintains a list of IP addresses and other configuration parameters. When requested, the DHCP server provides configuration parameters to a client.

Prerequisites

Install the following package:

- **dhcpd**

Run chkconfig to cause **dhcpd** to start when the system goes multiuser:

```
# /sbin/chkconfig dhcpd on
```

Start **dhcpd**:

```
# /sbin/service dhcpd start
```

dhcpd: The DHCP Daemon

A simple DCHP server allows you to add clients to a network without maintaining a list of assigned IP addresses. A simple network, such as a home LAN sharing an Internet connection, can use DHCP to assign a dynamic IP address to almost all nodes. The exceptions are servers and routers, which must be at known network locations in order to be able to receive connections. If servers and routers are configured without DHCP, you can specify a simple DHCP server configuration in **/etc/dhcpd.conf**:

```
$ cat /etc/dhcpd.conf
default-lease-time 600;
max-lease-time 86400;

option subnet-mask 255.255.255.0;
option broadcast-address 192.168.1.255;
option routers 192.168.1.1;
option domain-name-servers 192.168.1.1;

subnet 192.168.1.0 netmask 255.255.255.0 {
   range 192.168.1.2 192.168.1.200;
}
```

The preceding configuration file specifies a LAN where the router and DNS are both located on **192.168.1.1**. The **default-lease-time** specifies the number of seconds a dynamic IP lease will be valid if the client does not specify a duration. The **max-lease-time** is the maximum time allowed for a lease.

The information in the **option** lines is sent to each client when it connects. The names following the word **option** specify what the following argument represents. For example, the **option broadcast-address** line specifies the broadcast address of the network. The **routers** and **domain-name-servers** options allow multiple values separated by commas.

The **subnet** section includes a **range** line that specifies the range of IP addresses that the DHCP server can assign. If you define multiple subnets, you can define options, such as **subnet-mask**, inside the **subnet** section. Options defined outside all **subnet** sections are global and apply to all subnets.

The preceding configuration file assigns addresses in the range between 192.168.1.2 and 192.168.1.200. The DHCP server starts at the bottom of the range and attempts to assign a new IP address to each new client. Once the DHCP server reaches the top of the range, it starts reassigning IP addresses that have been used, but are not currently in use. If you have fewer systems than IP addresses, the IP address of each system should remain fairly constant. You cannot use the same IP address for more than one system at a time.

Once you have configured a DHCP server, you can start or restart it using the **dhcpd** init script:

```
# /sbin/service dhcpd restart
```

Once the server is running, clients configured to obtain an IP address from the server using DHCP should be able to do so.

Static IP Addresses

As mentioned earlier, routers and servers typically require static IP addresses. While you can manually configure IP addresses for these systems, it may be more convenient to have the DHCP server provide them with static IP addresses.

When a system that requires a specific static IP address connects to the network and contacts the DHCP server, the server needs a way to identify the system so it can assign the proper IP address. The DHCP server uses the *MAC address* (page 981) of the system's Ethernet card (NIC) to identify the system. When you set up the server, you must know the MAC addresses of each of the systems that requires a static IP address.

Displaying a MAC address

You can use ifconfig to display the MAC addresses of the Ethernet boards (NICs) in a system. The MAC address is the colon-separated series of hexadecimal number pairs following **HWaddr**:

```
$ /sbin/ifconfig | grep -i hwaddr
eth0      Link encap:Ethernet  HWaddr BA:DF:00:DF:C0:FF
eth1      Link encap:Ethernet  HWaddr 00:02:B3:41:35:98
```

Run ifconfig on each system that requires a static IP address. Once you have determined the MAC address of each system that requires a static IP address, you can add a **host** section to the **/etc/dhcpd.conf** file for each of these systems, instructing the DHCP server to assign a specific address to each system. The following **host** section assigns the address **192.168.1.1** to the system with the MAC address of **BA:DF:00:DF:C0:FF**:

```
$ cat /etc/dhcpd.conf
...
host router {
   hardware ethernet BA:DF:00:DF:C0:FF;
   fixed-address 192.168.1.1;
   option host-name router;
}
```

The name following **host** is used internally by **dhcpd**, while the name specified after **option host-name** is passed to the client and can be a hostname or a FQDN.

After making changes to **dhcpd.conf**, restart **dhcpd** using service and the **dhcpd** init script (page 411).

nsswitch.conf: Which Service to Look at First

With the advent of NIS and DNS, it was no longer a simple matter of searching a local file for user and system information. Where you used to look in **/etc/passwd** to get user information and look in **/etc/hosts** to find system address information, you can now use several methods to find this type of information. The **/etc/nsswitch.conf** (name service switch configuration) file specifies which methods to use and the order to use them in when looking for a certain type of information. You can also specify what action the system takes based on whether a method works or fails.

Format Each line in **nsswitch.conf** specifies how to search for a piece of information, such as a user's password. The format of a line in **nsswitch.conf** is

info: ***method [[action]] [method [[action]]...]***

where ***info*** specifies the type of information that the line describes (discussed following), ***method*** is the method used to find the information (described following), and ***action*** is the response to the return status of the preceding ***method***. The action is enclosed within square brackets.

How nsswitch.conf Works

When called upon to supply information that **nisswitch.conf** describes, the system examines the line with the appropriate ***info*** field. The system uses the methods specified on the line starting with the method on the left. By default, when the information is found, the system stops searching. Without an ***action*** specification, when a method fails to return a result, the next action is tried. It is possible for the search to end without finding the requested information.

Information

The **nisswitch.conf** file commonly controls searches for users (in **passwd**), passwords (in **shadow**), host IP address, and group information. The following list describes most of the types of information (***info*** in the format discussed earlier) that **nsswitch.conf** controls searches for.

automount Automount (**/etc/auto.master** and **/etc/auto.misc**, page 671)
bootparams Diskless and other booting options. See the **bootparam** man page.
ethers *MAC address* (page 981)

group	Groups of users (**/etc/group**, page 428)
hosts	System information (**/etc/hosts**, page 428)
netgroup	Netgroup information (**/etc/netgroup**, page 430)
networks	Network information (**/etc/networks**)
passwd	User information (**/etc/passwd**, page 430)
protocols	Protocol information (**/etc/protocols**, page 432)
publickey	Used for NFS running in secure mode
rpc	RPC names and numbers (**/etc/rpc**, page 433)
services	Services information (**/etc/services**, page 433)
shadow	Shadow password information (**/etc/shadow**, page 433)

Methods

Following is a list of the types of information that **nsswitch.conf** controls searches for (***method*** in the format on page 413). For each type of information, you can specify one or more of the following methods:[2]

files	Searches local files such as **/etc/passwd** and **/etc/hosts**.
nis	Searches the NIS database; **yp** is an alias for **nis**.
dns	Queries the DNS (**hosts** queries only).
compat	**±** syntax in **passwd, group, and shadow** files (page 416).

Search Order

The information provided by two or more methods may overlap: For example, **files** and **nis** can each provide password information for the same user. With overlapping information, you need to consider which method you want to be authoritative (take precedence), and put that method at the left of the list of methods.

The default **nsswitch.conf** file lists methods without actions, assuming no overlap (which is normal). In this case, the order is not critical: When one method fails, the system goes to the next one; all that is lost is a bit of time for a failure. Order becomes critical when you use actions between methods, or when you have overlapping entries that differ.

The first of the following lines from **nisswitch.conf** causes the system to search for password information in **/etc/passwd** and, if that fails, to use NIS to find the information. If the user you are looking for is listed in both places, the information in the local file would be used and therefore would be authoritative. The second line uses NIS and, if that fails, searches **/etc/hosts** and, if that fails, checks with DNS to find host information.

2. There are other, less commonly used methods. See the default **/etc/nisswitch.conf** file and the **nisswitch.conf** man page for more information. Although NIS+ belongs in this list, it is not implemented for Linux and is not discussed in this book.

```
passwd          files nis
hosts           nis files dns
```

Action Items

Each method can optionally be followed by an action item that specifies what to do if the method succeeds or fails for one of a number of reasons. The format of an action item is

[[!]STATUS=action]

where the opening and closing square brackets are part of the format and do not indicate that the contents are optional; ***STATUS*** (by convention uppercase although it is not case sensitive) is the status being tested for; and ***action*** is the action to be taken if ***STATUS*** matches the status returned by the preceding method. The leading exclamation point (**!**) is optional and negates the status.

Values for ***STATUS*** are

NOTFOUND The method worked, but the value being searched for was not found. Default action is **continue**.

SUCCESS The method worked, and the value being searched for was found; no error was returned. Default action is **return**.

UNAVAIL The method failed because it is permanently unavailable. For example, the required file may not be accessible or the required server may be down. Default action is **continue**.

TRYAGAIN The method failed because it is temporarily unavailable. For example, a file may be locked or a server overloaded. Default action is **continue**.

Values for ***action*** are

return Returns to the calling routine with or without a value.

continue Continues with the next method. Any returned value is overwritten by a value found by the next method.

For example, the following line from **nsswitch.conf** causes the system first to use DNS to search for the IP address of a given host. The action item following the DNS method tests to see if the status returned by the method is not (**!**) UNAVAIL.

```
hosts           dns [!UNAVAIL=return] files
```

The system takes the action associated with the ***STATUS*** (**return**) if the DNS method did not return UNAVAIL (!UNAVAIL), that is, if DNS returned SUCCESS, NOTFOUND, or TRYAGAIN. The result is that the following method (**files**) is used only when the DNS server is unavailable: If the DNS server is *not un*available (read the two negatives as "is available,"), the search returns the domain name or reports the domain name was not found. The search uses the **files** method (check the local **/etc/hosts** file) only if the server is not available.

compat Method: ± in passwd, group, and shadow files

You can put special codes in the **/etc/passwd**, **/etc/group**, and **/etc/shadow** files that cause the system, when you specify the **compat** method in **nisswitch.conf**, to combine and modify entries in the local files and the NIS maps.

A plus sign (+) at the beginning of a line in one of these files adds NIS information; a minus sign (–) removes information. For example, to use these codes in the **passwd** file, specify **passwd: compat** in **nisswitch.conf**. The system then goes through the **passwd** file in order, adding or removing the appropriate NIS entries when it gets to each line that starts with a + or –.

Although you can put a plus sign at the end of the **passwd** file and specify **passwd: compat** in **nsswitch.conf** to search the local **passwd** file and then go through the NIS map, it is more efficient to put **passwd: file nis** in **nsswitch.conf** and not modify the **passwd** file.

PAM

PAM (actually Linux-PAM, or Linux Pluggable Authentication Modules) allows a system administrator to determine how various applications use *authentication* (page 957) to verify the identity of a user. PAM provides shared libraries (page 813) of modules (located in **/lib/security**) that, when called by an application, authenticate a user. Pluggable refers to the ease with which you can add and remove modules from the authentication stack. The configuration files kept in the **/etc/pam.d** directory determine the method of authentication and contain a list, or stack, of calls to the modules. PAM may also use other files, such as **/etc/passwd**, when necessary.

Instead of building the authentication code into each application, PAM provides shared libraries to keep the authentication code separate from the application code. The techniques of authenticating users stay the same from application to application. PAM enables a system administrator to change the authentication mechanism for a given application without touching the application.

PAM provides authentication for a variety of system-entry services (login, ftp, and so on). You can take advantage of PAM's ability to stack authentication modules to integrate system-entry services with different authentication mechanisms, such as RSA, DCE, Kerberos, and smart cards.

From login through using su to shutting the system down, whenever you are asked for a password (or not asked for a password because the system trusts that you are who you say you are), PAM makes it possible for systems administrators to configure the authentication process and makes the configuration process essentially the same for all applications that use PAM to do their authentication.

EXCERPT FROM CHAPTER 20:
sendmail: Setting Up Mail Clients, Servers, and More

Sending and receiving email require three pieces of software. At each end, there is a client, called an MUA (Mail User Agent), which is a bridge between a user and the mail system. Common MUAs are mutt, Kmail, Mozilla Mail, and Outlook. When you send an email, the MUA hands it to an MTA (a Mail Transfer Agent such as **sendmail**), which transfers it to the destination server. At the destination, an MDA (a Mail Delivery Agent such as **procmail**) puts the mail in the recipient's mailbox file. On Linux systems, the MUA on the receiving system either reads the mailbox file or retrieves mail from a remote MUA or MTA, such as an ISP's SMTP (mail) server, using POP (Post Office Protocol) or IMAP (Internet Message Access Protocol).

Most Linux MUAs expect a local copy of **sendmail** will deliver outgoing email. On some systems, including those with a dialup connection to the Internet, **sendmail** relays email to an ISP's mail server. Because **sendmail** uses SMTP (Simple Mail Transfer Protocol) to deliver email, **sendmail** is often referred to as an SMTP server.

In the default Red Hat setup, the **sendmail** MTA uses **procmail** as the local MDA. By default, **procmail** writes email to the end of the recipient's mailbox file. You can also use **procmail** to sort email according to a set of rules, either on a per-user basis or globally. The global filtering function is useful for systemwide filtering to detect spam and for other tasks, but the per-user feature is largely superfluous on a modern system. Traditional UNIX MUAs were simple programs that could not filter mail and delegated this function to MDAs such as **procmail**. Modern MUAs incorporate this functionality.

caution || You Do Not Need to Set Up sendmail to Send and Receive Email

Most MUAs can use POP or IMAP for receiving email. These protocols do not require an MTA such as **sendmail**. Thus you do not need to install or configure **sendmail** (or other MTA) to receive email. You still need SMTP to send email. However, the SMTP server can be at a remote location, such as your ISP, so you do not need to concern yourself with it.

Introduction

When the network that was to evolve into the Internet was first set up, it connected a few computers, each serving a large number of users and running several services. Each computer was capable of sending and receiving email and had a unique host name, which was used as a destination for email.

Today, the Internet has a large number of transient clients. Because these clients do not have fixed IP addresses or hostnames, they cannot receive email directly. Users on these systems usually have an account on an email server run by their employer or an ISP, and they collect email from this account using POP or IMAP. Unless you own a domain that you want to receive email at, you will not need to set up **sendmail** as an incoming SMTP server.

You can set up **sendmail** on a client system so all it does is relay outbound mail to an SMTP server. This configuration is required by organizations that use firewalls to prevent email from being sent out on the Internet from any system other than the company's official mail servers. As a partial defense against spreading viruses, some ISPs block outbound port 25 to prevent their customers from sending email directly to a remote computer. This configuration is required by these ISPs.

You can also set up **sendmail** as an outbound server that does not use an ISP as a relay. In this configuration, **sendmail** connects directly to the SMTP servers for the domains receiving the email. An ISP set up as a relay is configured this way.

You can set up **sendmail** to accept email for a registered domain name as specified in the domain's DNS MX record (page 706). However, most mail clients (MUAs) do not interact directly with **sendmail** to receive email. Instead, they use POP or IMAP, protocols that include features for managing mail folders, leaving messages on the server, and reading only the subject of an email without downloading the entire message. If you want to collect your email from a system other than the one running the incoming mail server, you may need to set up a POP or IMAP server, as discussed on page 628.

Prerequisites

Install the following packages:

- **sendmail** (required)
- **sendmail-cf** (required to configure **sendmail**)
- **squirrelmail** (optional, provides Webmail, page 625)
- **spamassassin** (optional, provides spam filtering, page 622)
- **mailman** (optional, provides mailing list support, page 627)
- **imap** (optional, provides IMAP and POP incoming mail server daemons)

Run chkconfig to cause **sendmail** to start when the system goes multiuser (by default, **sendmail** does not run in single user mode):

```
# /sbin/chkconfig sendmail on
```

Start **sendmail**. Because **sendmail** is normally running, you need to restart it to cause **sendmail** to reread its configuration files. The following restart command works even when **sendmail** is not running; it just fails to shut down **sendmail**:

```
# /sbin/service sendmail restart
Shutting down sendmail:                                    [  OK  ]
Shutting down sm-client:                                   [  OK  ]
Starting sendmail:                                         [  OK  ]
Starting sm-client:                                        [  OK  ]
```

Run chkconfig to cause the spamassassin daemon, **spamd**, to start when the system goes multiuser (spamassassin is normally installed in this configuration):

```
# /sbin/chkconfig spamassassin on
```

As with **sendmail**, spamassassin is normally running; restart it to cause **spamd** to reread its configuration files:

```
# /sbin/service spamassassin restart
Starting spamd:                                            [  OK  ]
```

The IMAP and POP protocols are implemented as several different daemons that are controlled by **xinetd**. See page 628 for information on these daemons and how to start them.

More Information

Web **sendmail** www.sendmail.org
IMAP www.imap.org
SquirrelMail www.squirrelmail.org
Postfix www.postfix.org/docs.html
Qmail www.qmail.org
Mailman www.list.org
procmail www.procmail.org
SpamAssassin spamassassin.org
Spam database razor.sourceforge.net

JumpStart I: Configuring sendmail on a Client

This JumpStart configures an outbound **sendmail** server. This server

- Uses a remote SMTP server, typically an ISP, to relay email to its destination
- Sends to the SMTP server email originating from the local system only; it does not forward email originating from other systems
- Does not handle inbound email; as is frequently the case, you need to use POP or IMAP to receive email

To set up this server, you must edit **/etc/mail/sendmail.mc** and restart **sendmail**.

Change sendmail.mc

The **dnl** at the start of the following line in **sendmail.mc** says that the line is a comment:

```
dnl define(`SMART_HOST',`smtp.your.provider')
```

To specify a remote SMTP server, you must open **sendmail.mc** in an editor and change the preceding line, deleting **dnl** from the beginning of the line and replacing **smtp.your.provider** with the FQDN of your ISP's SMTP server (obtain this name from your ISP). Be careful not to alter the back tick (`) preceding or the single quotation mark (') following the FQDN. If your ISP's SMTP server is at **smtp.myisp.com**, you would change the line:

```
dnl define(`SMART_HOST',`smtp.your.provider')
```

Restart sendmail

When you restart it, **sendmail** regenerates the **sendmail.cf** file from the **sendmail.mc** file you edited:

```
# /sbin/service sendmail restart
```

Test

Test **sendmail** with the following command:

```
$ echo "my sendmail test" | /usr/sbin/sendmail user@remote.host
```

Replace *user@remote.host* with an email address on another system where you receive email. You need to send email to a remote system to make sure that **sendmail** is relaying your email.

JumpStart II: Configuring sendmail on a Server

If you want to receive inbound email sent to a registered domain that you own, you need to set up **sendmail** as an incoming mail server; this JumpStart describes how to set up such a server. This server

- Accepts outbound email from the local system only
- Delivers outbound email directly to the recipient's system, without using a relay

- Accepts inbound email from any system

This server does not relay outbound email originating on other systems. Refer to "access: Setting Up a Relay Host" on page 620 if you want the local system to act as a relay. For this configuration to work, you must be able to make outbound connections from, and receive inbound connections to, port 25.

The line in **sendmail.mc** that limits **sendmail** to accept inbound email from the local system only is

```
DAEMON_OPTIONS(`Port=smtp,Addr=127.0.0.1, Name=MTA')dnl
```

To allow **sendmail** to accept inbound email from other systems, remove the parameter **Addr=127.0.0.1,** from the preceding line, leaving the following line:

```
DAEMON_OPTIONS(`Port=smtp, Name=MTA')dnl
```

By default sendmail does not use a remote SMTP server to relay email, so there is nothing to change to cause sendmail to send email directly to recipients' systems. (JumpStart I set up a SMART_HOST to relay email.)

Once you have restarted **sendmail**, it will accept mail addressed to the local system, as long as there is a DNS MX record (page 706) pointing at the local system. If you are not running a DNS server, you must ask your ISP to set up an MX record.

How sendmail Works

Outbound email

When you send email, the MUA passes the email to **sendmail**, which creates in the **/var/spool/mqueue** directory two files that hold the message while **sendmail** processes it. In order to generate unique filenames for a particular piece of email, **sendmail** generates a random string for each piece of email and uses the string in filenames pertaining to the email. The **sendmail** daemon stores the body of the message in a file named **df** (data file) followed by the generated string. It stores the headers and other information in a file named **qf** (queue file) followed by the generated string.

If a delivery error occurs, sendmail creates a temporary copy of the message that it stores in a file whose name starts with **tf** and logs errors in a file whose name starts **xf**. Once an email has been sent successfully, **sendmail** removes all files pertaining to that email from **/var/spool/mqueue**.

Incoming email

By default, the MDA stores incoming messages in users' files in the mail spool directory, **/var/spool/mail**, in **mbox** format (next paragraph). Within this directory, each user has a mail file named with the user's username. Mail remains in these files until it is collected, typically by an MUA. Once an MUA collects the mail from the mail spool, the MUA stores the mail as directed by the user, usually in the user's home directory hierarchy.

mbox versus maildir

The **mbox** format, which **sendmail** uses, stores all messages for a user in a single file. To prevent corruption, the file must be locked while a process is adding messages to or deleting messages from the file; you cannot delete a message at the same time the MTA is adding messages. A competing format, **maildir**, stores each message in a separate file. This format does not use locks, allowing an MUA to read and delete messages at the same time as new mail is delivered. In addition, the **maildir** format is better able to handle larger mailboxes. The downside is that the **maildir** format adds overhead when using a protocol such as IMAP to check messages. Qmail (page 633) uses **maildir** format mailboxes.

Mail logs

The **sendmail** daemon stores log messages in **/var/log/maillog**. Other mail servers, such as the **imapd** and **ipop3d** daemons may log information to this file. Following is a sample log entry:

/var/log/maillog

```
# cat /var/log/maillog
...
Mar  3 16:25:33 MACHINENAME sendmail[7225]: i23GPXvm007224:
to=<user@localhost.localdomain>, ctladdr=<root@localhost.localdomain>
(0/0), delay=00:00:00, xdelay=00:00:00, mailer=local, pri=30514,
dsn=2.0.0, stat=Sent
```

Each log entry starts with a timestamp, the name of the system sending the email, the name of the mail server (**sendmail**), and a unique identification number. The address of the recipient follows the **to=** label and the address of the sender follows **ctladdr=**. Additional fields provide the name of the mailer and the time it took to send the message. If a message is sent correctly, the **stat=** label is followed by **Sent**.

A message is marked **Sent** when **sendmail** sends it; **Sent** does not indicate that the message has been delivered. If a message is not delivered due to an error down the line, the sender usually receives an email saying that it was not delivered, giving a reason why.

If you get a lot of email, the **maillog** file can grow quite large; the **syslog** logrotate (page 543) entry is set up to archive and rotate the **maillog** files regularly.

Aliases and Forwarding

There are three files that can forward email: **.forward** (page 615), **aliases** (next), and **virtusertable** (page 621). See page 621 for a table comparing the three files.

/etc/aliases

Most of the time when you send email, it goes to a specific person; the recipient, **user@system**, maps to a specific, real user on the specified system. Sometimes you may want email to go to a class of user and not to a specific recipient. Examples of classes of users are **postmaster**, **webmaster**, **root**, **tech_support**, and so on. Different users may receive this email at different times or the email may be answered by a group of users. You can use the **/etc/aliases** file to map inbound addresses to local users, files, commands, and remote addresses.

Each line in **/etc/aliases** contains the name of a local pseudouser, followed by a colon, whitespace, and a comma-separated list of destinations. The default installation includes a number of aliases that redirect messages for certain pseudousers to **root**. These have the form

```
system:         root
```

Sending messages to the **root** account is a good way of making them easy to review, but, because it is rare that anyone checks **root**'s email, you may want to send copies to a real user. The following line forwards mail sent to **abuse** on the local system to **root** and **alex**:

```
abuse:          root, alex
```

You can create simple mailing lists with this type of alias. For example, the following alias sends copies of all email sent to **admin** on the local system to several users, including Zach, who is on a different system:

```
admin:          sam, helen, mark, zach@tcorp.com
```

You can direct email to a file by specifying an absolute pathname in place of a destination address. The following alias, which is quite popular among less conscientious system administrators, redirects email sent to **complaints** to **/dev/null** (page 426) where they disappear:

```
complaints:     /dev/null
```

You can also send email to standard input of a command by preceding the command with a pipe character (|). This technique is commonly used with mailing list software such as Mailman (page 627). For each list it maintains, Mailman has entries, such as the following entry for **mylist**, in the **aliases** file:

```
mylist:         "|/var/mailman/mail/mailman post mylist"
```

newaliases After you edit **/etc/aliases**, you must either run newaliases as **root** or restart **sendmail** to recreate the **aliases.db** file that **sendmail** reads.

praliases You can use praliases to list aliases currently loaded by **sendmail**:

```
# /usr/sbin/praliases | head -5
postmaster:root
daemon:root
adm:root
lp:root
shutdown:root
```

~/.forward Systemwide aliases are useful in many cases, but non**root** users cannot make or change them. Sometimes you may want to forward your own mail: Maybe you want to have mail from several systems go to one address or perhaps you just want to forward your mail while you are at another office for a week. The **~/.forward** file allows ordinary users to forward their email.

Lines in a **.forward** file are the same as the right column of the **aliases** file explained previously: Destinations are listed one per line and can be a local user, a remote email address, a filename, or a command preceded by a pipe character (|).

Mail that you forward does not go to your local mailbox. If you want to forward mail and keep a copy in your local mailbox, you must specify your local username preceded by a backslash to prevent an infinite loop. The following example sends Sam's email to himself on the local system and on the system at **tcorp.com**:

```
$cat ~sam/.forward
sams@tcorp.com
\sam
```

Related Programs

sendmail

The **sendmail** distribution includes several programs. The primary program, **sendmail**, reads from standard input and sends an email to the recipient specified by its argument. You can use **sendmail** from the command line to check that the mail delivery system is working and to email the output of scripts:

```
$ echo "sample email" | /usr/sbin/sendmail sams@tcrop.net
```

mailq

The mailq utility displays the status of the outgoing mail queue and normally reports there are no messages in the queue. Messages in the queue usually indicate a problem with the local or remote **sendmail** configuration or a network problem.

```
# /usr/bin/mailq
/var/spool/mqueue is empty
                     Total requests: 0
```

mailstats

The mailstats utility reports on the number and sizes of messages **sendmail** has sent and received since the date it displays on the first line:

```
# /usr/sbin/mailstats
Statistics from Sat Dec 22 16:02:34 2001
M   msgsfr  bytes_from   msgsto    bytes_to  msgsrej msgsdis  Mailer
0        0          0K    17181     103904K        0       0  prog
4   368386    4216614K   136456    1568314K    20616       0  esmtp
9   226151   26101362K   479025   12776528K     4590       0  local
=============================================================
T   594537   30317976K   632662   14448746K    25206       0
C   694638               499700               146185
```

In the preceding output, each mailer is identified by the first column, which displays the mailer number, and by the last column, which displays the name of the mailer. The second through fifth columns display the number and total sizes of messages sent and received by the mailer. The sixth and seventh columns display the number of messages rejected and discarded respectively. The row that starts with **T** lists the column totals and the row that starts with **C** lists the number of TCP connections.

makemap

The makemap utility processes text configuration files in **/etc/mail** into the database format that sendmail reads (***.db** files). You do not need to run makemap manually; it is invoked by the **sendmail** init script when you start or restart **sendmail**.

Configuring sendmail

The **sendmail** configuration files reside in **/etc/mail** where the primary configuration file is **sendmail.cf**. This directory contains other text configuration files, such as **access**, **mailertable**, and **virtusertable**. The **sendmail** daemon does not read these files but reads the corresponding ***.db** files in the same directory.

You can use makemap or give the command **make** from the **/etc/mail** directory to generate all the ***.db** files, although this step is not usually necessary. The **sendmail** init script automatically generates these files when you restart **sendmail**:

```
# /sbin/service sendmail restart
```

The sendmail.mc and sendmail.cf Files

This **sendmail.cf** file is not intended to be edited by hand and contains a large warning to this effect:

```
$ cat /etc/mail/sendmail.cf
...
##########################################################
#####
#####   DO NOT EDIT THIS FILE!  Only edit the source .mc file.
#####
##########################################################
...
```

Editing sendmail.mc and Generating sendmail.cf

The **sendmail.cf** file is generated from **sendmail.mc** using the m4 macro processor. It can be helpful to use a text editor that supports syntax highlighting, such as vim, to edit **sendmail.mc**.

dnl Many of the lines in **sendmail.mc** start with **dnl**, which stands for **delete to new line**, and instructs m4 to delete from the **dnl** to the end of the line (the next NEWLINE character). Because m4 ignores anything on a line after a **dnl** instruction, you can use **dnl** to introduce comments; it works the same way as a # does in a shell script.

Many of the lines in **sendmail.mc** *end* with **dnl**. Because NEWLINEs immediately follow these **dnl**s, these **dnl**s are superfluous; you can remove them if you like.

After editing **sendmail.mc**, you need to regenerate **sendmail.cf** and restart **sendmail** to make your changes take effect. You can give the command **make** from the **/etc/mail** directory to regenerate **sendmail.cf**, although this step is not usually necessary. The **sendmail** init script automatically regenerates **sendmail.cf** when you restart **sendmail**.

About sendmail.mc

Lines near the beginning of **sendmail.mc** provide basic configuration information.

```
divert(-1)dnl
include(`/usr/share/sendmail-cf/m4/cf.m4')dnl
VERSIONID(`setup for Red Hat Linux')dnl
OSTYPE(`linux')dnl
```

The line that starts with **divert** tells m4 to discard extraneous output it may generate when processing this file.

The **include** statement, which tells m4 where to find the macro definition file that it will use to process the rest of this file, points to the file named **cf.m4**. The **cf.m4** file contains other **include** statements that include parts of the **sendmail** configuration rule sets.

The **VERSIONID** statement defines a string that indicates the version of this configuration. You can change this string to include a brief comment about the changes you make to this file or other information. The value of this string is not significant to **sendmail**.

Do not change the **OSTYPE** statement unless you are migrating a **sendmail.mc** file from another operating system.

Other statements you may want to change are explained in the following sections and in the **sendmail** documentation.

tip || Quoting m4 Strings

The m4 macro processor, which converts **sendmail.mc** to **sendmail.cf**, requires strings to be preceded by a back tick (`) and closed with a single quotation mark (').

Masquerading

Typically, you want your email to appear to come from the user and the domain where you receive email; sometimes the outbound server is in a different domain than the inbound server. You can cause **sendmail** to alter outbound messages so that they appear to come from a user and/or domain other than the one they are sent from: You *masquerade* (page 982) the message.

There are several lines in **sendmail.mc** that pertain to this type of masquerading; each is commented out in the file that Red Hat distributes:

```
dnl MASQUERADE_AS(`mydomain.com')dnl
dnl MASQUERADE_DOMAIN(localhost)dnl
dnl FEATURE(masquerade_entire_domain)dnl
```

The MASQUERADE_AS statement causes email that you send from the local system to appear to come from the domain specified within the single quotation marks (**mydomain.com** in the commented out line in the distributed file). Remove the leading **dnl** and change **mydomain.com** to the domain name that you want mail to appear to come from.

The MASQUERADE_DOMAIN statement causes email from the specified system or domain to be masqueraded, just as local email is. That is, email from the system spec-

ified in this statement is treated as though it came from the local system: It is changed to appear to come from the domain specified in the MASQUERADE_AS statement. Remove the leading **dnl** and change **localhost** to the name of the system or domain that sends the email that you want to masquerade. If the name you specify has a leading period, it specifies a domain; if there is no leading period, it specifies a system or host. You can have as many MASQUERADE_DOMAIN statements as necessary.

The **masquerade_entire_domain** feature statement causes sendmail also to masquerade subdomains of the domain specified in the MASQUERADE_DOMAIN statement. Remove the leading **dnl** to masquerade entire domains.

Accepting Email from Unknown Hosts

As shipped by Red Hat, **sendmail** is configured to accept email from domains that it cannot resolve (and that may not exist). To turn this feature off and cut down the amount of spam you receive, add **dnl** to the beginning of the following line:

```
FEATURE(`accept_unresolvable_domains')dnl
```

When this feature is off, **sendmail** uses DNS to look up the domains of all email it receives; if it cannot resolve the domain, it rejects the email.

Setting Up a Backup Server

You can set up a backup mail server to hold email when the primary mail server experiences problems. For maximum coverage, the backup server should be on a different connection to the Internet from the primary server.

Setting up a backup server is easy: Remove the leading **dnl** from the following line in the *backup* mail server's **sendmail.mc** file:

```
dnl FEATURE(`relay_based_on_MX')dnl
```

DNS MX records (page 706) specify where email for a domain should be sent. You can have multiple MX records for a domain, each pointing to a different mail server. When a domain has multiple MX records, each record usually has a different priority; priority is specified by a two-digit number with lower numbers specifying higher priorities.

When attempting to deliver email, an MTA first tries to deliver email to the highest priority server. Failing that delivery, it tries to deliver to a lower-priority server. If you activate the **relay_based_on_MX** feature and point a low-priority MX record at a secondary mail server, the mail server will accept email for the domain. The mail server will then forward email to the server identified by the highest priority MX record for the domain when that server becomes available.

Other Files in /etc/mail

The **/etc/mail** directory holds most of the files that control **sendmail**. This section discusses three of those files: **mailertable**, **access**, and **virtusertable**.

EXCERPT FROM CHAPTER 28:
Programming the Bourne Again Shell

Chapters 7 and 9 introduced the Bourne Again Shell. This chapter describes additional commands, builtins, and concepts that carry shell programming to a point where it can be useful. The first programming constructs covered are control structures, or control flow constructs. These structures allow you to write scripts that can loop over command line arguments, make decisions based on the value of a variable, set up menus, and more. The Bourne Again Shell uses the same constructs found in such high-level programming languages as C.

This chapter goes on to explain how the Here document makes it possible for you to redirect input to a script to come from the script itself, rather than from the terminal or other file. "Expanding Null or Unset Variables" (page 884) shows you various ways to set default values for a variable. The section on the exec builtin demonstrates how it provides an efficient way to execute a command by replacing a process and how you can use it to redirect input and output from within a script. The next section covers the trap builtin, which provides a way to detect and respond to operating system signals (or interrupts, such as the one that is generated when you press CONTROL-C). Finally, the section on functions gives you a clean way to execute code similar to scripts much more quickly and efficiently.

This chapter contains many examples of shell programs. Although they illustrate certain concepts, most use information from earlier examples as well. This overlap not only reinforces your overall knowledge of shell programming but also demonstrates how commands can be combined to solve complex tasks. Running, modifying, and experimenting with the examples are good ways to become comfortable with the underlying concepts.

tip || **Do Not Name a Shell Script test**

You can create a problem for yourself if you give a shell script the name **test**. A Linux utility has the same name. Depending on how you have your **PATH** variable set up and how you call the program, you may run your script or the utility, leading to confusing results.

This chapter illustrates concepts with simple and more complicated examples. The more complex scripts illustrate traditional shell programming practices and introduce some Linux utilities often used in scripts. The first time you read the chapter, you can skip these sections without loss of continuity. Return to them later when you feel comfortable with the basic concepts.

Control Structures

The *control flow* commands alter the order of execution of commands within a shell script. Control structures include the **if...then**, **for...in**, **while**, **until**, and **case** statements. In addition, the **break** and **continue** statements work in conjunction with the control flow structures to alter the order of execution of commands within a script.

if...then

The syntax of the **if...then** control structure is

if ***test-command***
then
commands
fi

The ***bold*** words in the syntax description are the items you supply to cause the structure to have the desired effect. The *nonbold* words are the keywords the shell uses to identify the control structure.

Figure 28-1 shows that the **if** statement tests the status returned by the ***test-command*** and transfers control based on this status. The end of the **if** structure is marked by a **fi** statement, which is *if* spelled backwards. The following script prompts you for two words, reads them in, and then uses an **if** structure to evaluate the result returned by the test builtin when it compares the two words. The test builtin returns a status of *true* if the two words are the same and *false* if they are not. Double quotation marks around **$word1** and **$word2** make sure that test works properly if you enter a string that contains a SPACE or other special character:

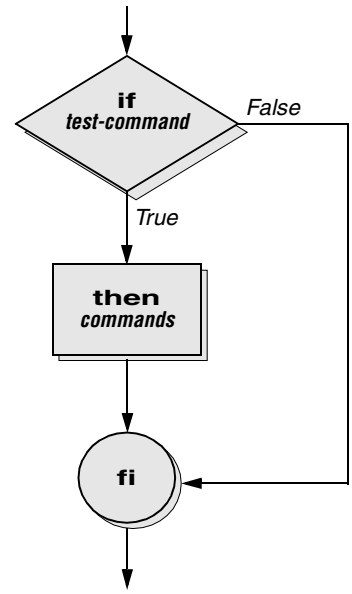

Figure 28-1 An **if...then** flowchart

```
$ cat if1
echo -n "word 1: "
read word1
echo -n "word 2: "
read word2

if test "$word1" = "$word2"
    then
        echo "Match"
fi
echo "End of program."

$ if1
word 1: peach
word 2: peach
Match
End of program.
```

test In the preceding example, the *test-command* is **test "$word1" = "$word2"**. The test builtin returns a *true* status if its first and third arguments have the relationship specified by its second argument. If this command returns a *true* status (= 0), the shell executes the commands between the **then** and **fi** statements. If the command returns a *false* status (not = 0), the shell passes control to the statement after **fi** without executing the statements between **then** and **fi**. The effect of this **if** statement is to display **Match** if the two words match. The script always displays **End of program**.

In the Bourne Again Shell, test is a builtin—part of the shell. It is also a stand-alone utility kept in **/usr/bin/test**. This chapter discusses and demonstrates many Bourne Again Shell builtins. You usually use the builtin version if it is available and the utility if it is not. Each version of a command may vary slightly from one shell to the next and from the utility to any of the shell builtins. To locate documentation, first determine whether you are using a builtin or a stand-alone utility. Use the type builtin for this purpose:

```
$ type test cat echo who if
test is a shell builtin
cat is hashed (/bin/cat)
echo is a shell builtin
who is /usr/bin/who
if is a shell keyword
```

To get more information on a stand-alone utility, use the man or info command followed by the name of the utility. Refer to "Builtins" on page 211 for instructions on how to find information on a builtin command.

The next program uses an **if** structure at the beginning of a script to check that you supplied at least one argument on the command line. The **–eq** test operator compares two integers. This structure displays a message and exits from the script if you do not supply an argument:

```
$ cat chkargs
if test $# -eq 0
    then
        echo "You must supply at least one argument."
        exit 1
fi
echo "Program running."
$ chkargs
You must supply at least one argument.
$ chkargs abc
Program running.
```

A test like the one shown in **chkargs** is a key component of any script that requires arguments. To prevent the user from receiving meaningless or confusing information from the script, the script needs to check whether the user has supplied the appropriate arguments. Sometimes the script simply tests whether arguments exist (as in **chkargs**). Other scripts test for a specific number or specific kinds of arguments.

You can use test to ask a question about the status of a file argument or the relationship between two file arguments. After verifying that at least one argument has been given on the command line, the following script tests whether the argument is the name of a regular file (not a directory or other type of file) in the working directory. The test builtin with the **–f** option and the first command line argument (**$1**) check the file:

```
$ cat is_regfile
if test $# -eq 0
    then
        echo "You must supply at least one argument."
        exit 1
fi
if test -f "$1"
    then
        echo "$1 is a regular file in the working directory"
    else
        echo "$1 is NOT a regular file in the working directory"
fi
```

With test and various options, you can test many other characteristics of a file. Some of the options are listed in Table 28-1

table 28-1 || Options to the test Utility

Option	Test Performed on File
–d	Exists and is a directory file
–e	Exists
–f	Exists and is a regular file
–r	Exists and is readable
–s	Exists and has a length greater than 0
–w	Exists and is writable
–x	Exists and is executable

Other test options provide a way to test for a relationship between two files, such as whether one file is newer than another. Refer to later examples in this chapter, as well as the test man page, for more detailed information. (Although test is a builtin, the utility described on the test man page functions similarly.)

tip || Always Test the Arguments

To keep the examples in this and subsequent chapters short and focused on specific concepts, the code to verify arguments is often omitted or abbreviated. It is a good practice to include tests for argument verification in your own shell programs. Doing so will result in scripts that are easier to run and debug.

The following example, another version of **chkargs**, checks for arguments in a way that is more traditional for Linux shell scripts. The example uses the bracket ([]) synonym for test. Rather than using the word test in scripts, you can surround the arguments to test with brackets, as shown. The brackets must be surrounded by whitespace (SPACEs or TABs).

```
$ cat chkargs2
if [ $# -eq 0 ]
    then
        echo "Usage: chkargs2 argument..." 1>&2
        exit 1
fi
echo "Program running."
exit 0
$ chkargs2
Usage: chkargs2 arguments
$ chkargs2 abc
Program running.
```

The error message that **chkargs2** displays is called a *usage message* and uses the 1>&2 notation to redirect its output to standard error (page 260). After issuing the usage message, **chkargs2** exits with an exit status of 1, indicating that an error has occurred. The **exit 0** command at the end of the script causes **chkargs2** to exit with a 0 status after the program runs without an error.

The usage message is a common notation to specify the type and number of arguments the script takes. Many Linux utilities provide usage messages similar to the one in **chkargs2**. When you call a utility or other program with the wrong number or kind of arguments, you often see a usage message. Following is the usage message that cp displays when you call it without any arguments:

```
$ cp
cp: missing file arguments
Try `cp --help' for more information.
```

if...then...else

The introduction of the **else** statement turns the **if** structure into the two-way branch shown in Figure 28-2. The syntax of the **if...then...else** control structure is

if ***test-command***
then
commands
else
commands
fi

Because a semicolon (;) ends a command just as a NEWLINE does, you can place **then** on the same line as **if** by preceding it with a semicolon. (Because **if** and **then** are separate builtins, they require a command separator between them; a semicolon and NEWLINE work equally well.) Some people prefer this notation for aesthetic reasons; others, because it saves space:

if ***test-command****; then*
commands
else
commands
fi

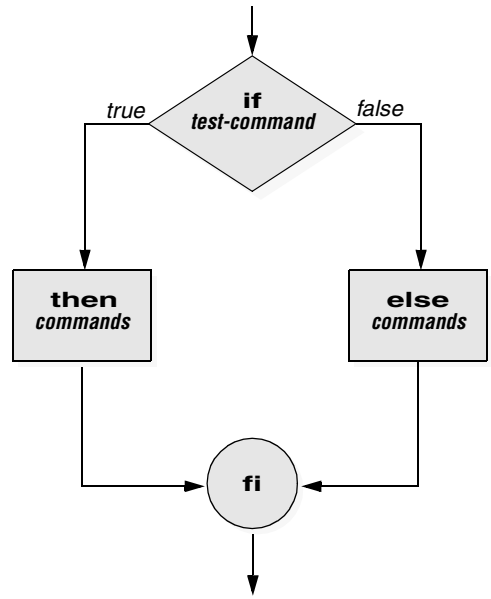

Figure 28-2 An **if...then...else** flowchart

If the *test-command* returns a *true* status, the **if** structure executes the commands between the **then** and **else** statements and then diverts control to the statement following **fi**. If the *test-command* returns a *false* status, the **if** structure executes the commands following the **else** statement.

The next script builds on **chkargs2**. When you run **out** with arguments that are filenames, it displays the files on the terminal. If the first argument is a **–v** (called an option in this case), **out** uses less (page 120) to display the files one page at a time. After determining that it was called with at least one argument, **out** tests its first argument to see whether it is **–v**. If the result of the test is *true* (if the first argument is **–v**), **out** shifts the arguments to get rid of the **–v** and displays the files using less. If the result of the test is *false* (if the first argument is *not* **–v**), the script uses cat to display the files:

```
$ cat out
if [ $# -eq 0 ]
    then
        echo "Usage: out [-v] filenames..." 1>&2
        exit 1
fi
if [ "$1" = "-v" ]
    then
        shift
        less -- "$@"
    else
        cat -- "$@"
fi
```

optional ||

In **out**, the **––** argument to cat and less tells the utility that no more options follow on the command line and not to consider leading hyphens (**–**) in the following list as indicating options. Thus **––** allows you to view a file with a name that starts with a hyphen. Although not common, filenames beginning with a hyphen do occasionally occur. (One way to create such a file is to use the command **cat > –fname**.) The **––** argument works with all Linux utilities that use the getopts function, a bash builtin, to parse their options. It does not work with all Linux utilities (for example, more). It is particularly useful with rm to remove a file whose name starts with a hyphen (**rm –– –fname**), including any that you create while experimenting with the **––** argument.

if...then...elif

The format of the **if...then...elif** control structure, shown in Figure 28-3, is as follows:

```
if test-command
        then
            commands
        elif test-command
            then
                commands
.
.
        else
            commands
fi
```

The **elif** statement combines the **else** statement and the **if** statement and allows you to construct a nested set of **if...then...else** structures (Figure 28-3). The difference between the **else** statement and the **elif** statement is that each **else** statement must be paired with a **fi** statement, whereas multiple nested **elif** statements require only a single closing **fi** statement.

The following example shows an **if...then...elif** control structure. This shell script compares three words that the user enters. The first **if** statement uses the AND operator (**–a**) as an argument to test. The test builtin returns a *true* status only if the first and the second logical comparisons are true (that is, if **word1** matches **word2** and **word2** matches **word3**). If **test** returns a *true* status, the program executes the command following the next **then** statement and passes control to the **fi** statement, and the script terminates:

```
$ cat if3
echo –n "word 1: "
read word1
echo –n "word 2: "
read word2
echo –n "word 3: "
```

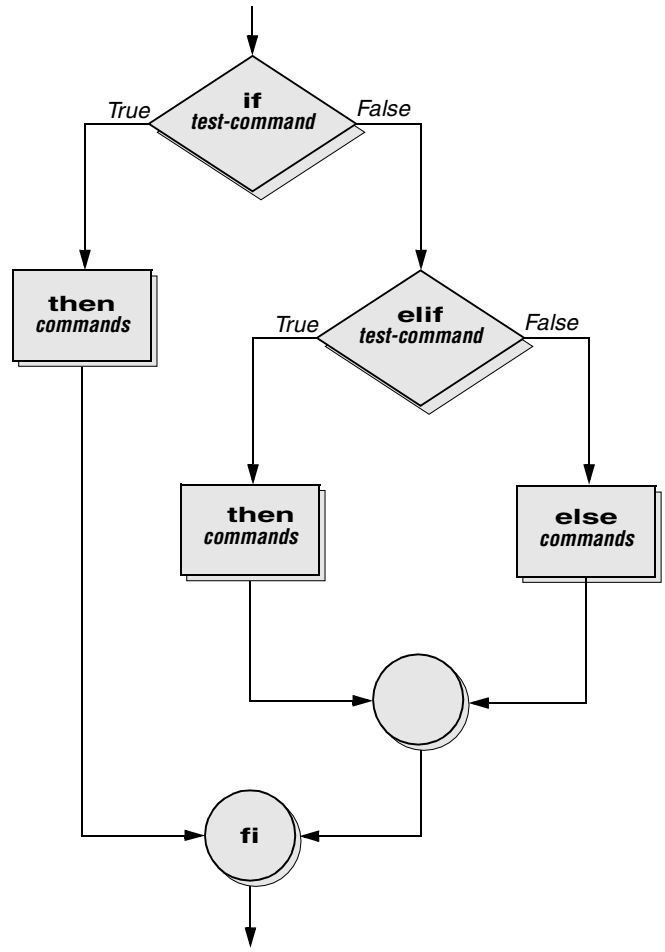

Figure 28-3 An **if...then...elif** flowchart

```
read word3
if [ "$word1" = "$word2" -a "$word2" = "$word3" ]
    then
        echo "Match: words 1, 2, & 3"
    elif [ "$word1" = "$word2" ]
    then
        echo "Match: words 1 & 2"
    elif [ "$word1" = "$word3" ]
    then
        echo "Match: words 1 & 3"
    elif [ "$word2" = "$word3" ]
    then
        echo "Match: words 2 & 3"
    else
        echo "No match"
fi
```

```
$ if3
word 1: apple
word 2: orange
word 3: pear
No match
$ if3
word 1: apple
word 2: orange
word 3: apple
Match: words 1 & 3
$ if3
word 1: apple
word 2: apple
word 3: apple
Match: words 1, 2, & 3
```

If the three words are not the same, the structure passes control to the first **elif**, which begins a series of tests to see if any pair of words is the same. As the nesting continues, if any one of the **if** statements is satisfied, the structure passes control to the next **then** statement and subsequently to the statement after **fi**. Each time an **elif** statement is not satisfied, the structure passes control to the next **elif** statement. In the **if3** script, the double quotation marks around the arguments to echo that contain ampersands (&) prevent the shell from interpreting the ampersands as special characters.

The lnks Script

The following script, **lnks**, demonstrates the **if...then** and **if...then...elif** control structures. This script finds hard links to its first argument, a filename. If you provide a name of a directory as the second argument, **lnks** searches for links in that directory and all subdirectories. If you do not specify a directory, **lnks** searches the working directory and its subdirectories.

```
$ cat lnks
#!/bin/bash
# Identify links to a file
# Usage: lnks file [directory]

if [ $# -eq 0 -o $# -gt 2 ]; then
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
fi
if [ -d "$1" ]; then
    echo "First argument cannot be a directory." 1>&2
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
else
    file="$1"
fi
```

```
if [ $# -eq 1 ]; then
    directory="."
elif [ -d "$2" ]; then
    directory="$2"
else
    echo "Optional second argument must be a directory." 1>&2
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
fi

# Check to make sure file exists and is a regular file:
if [ ! -f "$file" ]; then
    echo "lnks: $file not found or special file" 1>&2
    exit 1
fi
# Check link count on file
set -- $(ls -l "$file")

linkcnt=$2
if [ "$linkcnt" -eq 1 ]; then
    echo "lnks: no other link to $file" 1>&2
    exit 0
fi

# Get the inode of the given file
set $(ls -i "$file")

inode=$1

# Find and print the files with that inode number
echo "lnks: using find to search for links..." 1>&2
find "$directory" -xdev -inum $inode -print
```

In the following example, Alex uses **lnks** while he is in his home directory to search for links to a file named **letter** in the working directory. The **lnks** script reports that **/home/alex/letter** and **/home/jenny/draft** are links to the same file:

```
$ lnks letter /home
lnks: using find to search for links...
/home/alex/letter
/home/jenny/draft
```

In addition to the **if...then...elif** control structure, **lnks** introduces other features that are commonly used in shell programs. The following discussion describes **lnks** section by section.

The first line of the **lnks** script specifies the shell to execute the script (refer to "**#!** Specifies a Shell" on page 271):

```
#!/bin/bash
```

In this chapter the **#!** notation appears only in more complex examples. It ensures that the proper shell executes the script, even if the user is currently running a different shell. It also works correctly if invoked within another shell script.

The second and third lines of **lnks** are comments; the shell ignores the text that follows pound signs up to the next NEWLINE character. These comments in **lnks** briefly identify what the file does and how to use it.

```
# Identify links to a file
# Usage: lnks file [directory]
```

The first **if** statement in **lnks** tests whether **lnks** was called with zero arguments or more than two arguments:

```
if [ $# -eq 0 -o $# -gt 2 ]; then
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
fi
```

If either of these conditions is true, **lnks** sends a usage message to standard error and exits with a status of 1. The double quotation marks around the usage message prevent the shell from interpreting the brackets as special characters. The brackets in the usage message indicate to the user that the **directory** argument is optional.

The second **if** statement tests to see whether **$1** is a directory (the **–d** argument to test returns a *true* value if the file exists and is a directory):

```
if [ -d "$1" ]; then
    echo "First argument cannot be a directory." 1>&2
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
else
    file="$1"
fi
```

If it is a directory, **lnks** presents a usage message and exits. If it is not a directory, **lnks** saves the value of **$1** in the **file** variable because later in the script set resets the command line arguments. If the value of **$1** is not saved before the **set** command is issued, its value is lost.

The next section of **lnks** is an **if...then...elif** statement:

```
if [ $# -eq 1 ]; then
    directory="."
elif [ -d "$2" ]; then
    directory="$2"
else
    echo "Optional second argument must be a directory." 1>&2
    echo "Usage: lnks file [directory]" 1>&2
    exit 1
fi
```

The first ***test-command*** determines whether the user specified a single argument on the command line. If the ***test-command*** returns 0 (*true*), the user-created variable named **directory** is assigned the value of the working directory (**.**). If the ***test-command*** returns a *false* value, the **elif** statement tests whether the second argument is a directory. If it is a directory, the **directory** variable is set equal to the second command line argument, **$2**. If **$2** is not a directory, **lnks** sends a usage message to standard error and exits with a status of 1.

The next **if** statement in **lnks** tests whether **$file** does not exist. This is an important inquiry because it would be pointless for **lnks** to spend time looking for links to a nonexistent file.

The test builtin with the three arguments, **!**, **–f**, and **$file**, evaluates to *true* if the file **$file** does *not* exist:

```
[ ! -f "$file" ]
```

The **!** operator preceding the **–f** argument to test negates its result, yielding *false* if the file **$file** *does* exist and is a regular file.

Next, **lnks** uses set and **ls –l** to check the number of links **$file** has:

```
# Check link count on file
set -- $(ls -l "$file")
linkcnt=$2
if [ "$linkcnt" -eq 1 ]; then
    echo "lnks: no other links to $file" 1>&2
    exit 0
fi
```

The set builtin uses command substitution (page 281) to set the positional parameters to the output of **ls –l**. In the output of **ls –l**, the second field is the link count, so the user-created variable **linkcnt** is set equal to **$2**. The **––** used with set prevents set from interpreting as an option the first argument that **ls –l** produces (the first argument is the access permissions for the file, and it is likely to begin with **–**). The **if** statement checks whether **$linkcnt** is equal to 1; if it is, **lnks** displays a message and exits. Although this message is not truly an error message, it is redirected to standard error. The way **lnks** has been written, all informational messages are sent to standard error. Only the final product of lnks—the pathnames of links to the specified file—is sent to standard output, so you can redirect the output as you please.

If the link count is greater than one, **lnks** goes on to identify the inode (page 436) for **$file**. As explained in Chapter 6 (page 180), comparing the inodes associated with filenames is a good way to determine whether the filenames are links to the same file. The **lnks** script uses set again to set the positional parameters to the output of **ls –i**. The first argument to set is the inode number for the file, so the user-created variable named **inode** is set to the value of **$1**:

```
# Get the inode of the given file
set $(ls -i "$file")
inode=$1
```

Finally, **lnks** uses the find utility to search for filenames having inodes that match **$inode**.

```
# Find and print the files with that inode number
echo "lnks: using find to search for links..." 1>&2
find "$directory" -xdev -inum $inode -print
```

The find utility searches for files that meet the criteria specified by its arguments, beginning its search with the directory specified by its first argument (**$directory** in this case) and searching all subdirectories. The last three arguments to find specify

that the filenames of files having inodes matching **$inode** should be sent to standard output. Because files in different filesystems can have the same inode number and not be linked, find must search only directories in the same filesystem as **$file** for accurate results. The **–xdev** argument to find prevents the search of subdirectories on other filesystems. Refer to page 177 and page 436 for more information about filesystems and links. Refer to the find man page for more information on find.

The echo above the find command in **lnks**, which tells the user that find is running, is included because find frequently takes a long time to run. Because **lnks** does not include a final exit statement, the exit status of **lnks** is that of the last command it runs, find.

Debugging Shell Scripts

When you are writing a script such as **lnks**, it is easy to make mistakes. While you are debugging a script, you can use the shell's **–x** option, which causes the shell to display each command before it runs the command. This trace of a script's execution can give you a lot of information about where the problem is.

Suppose that Alex wants to run **lnks** as in the previous example, while displaying each command before it is executed. He can either set the **–x** option for the current shell (**set –x**) so that all scripts display commands as they are run or use the **–x** option to affect only the script he is currently executing:

```
$ bash -x lnks letter /home
```

Each command that the script executes is preceded by a plus sign (**+**) so that you can distinguish the output of the trace from any output that your script produces. You can also set the **–x** option of the shell running the script by putting the following set command at the top of the script:

```
set -x
```

Turn off the debug option with a plus sign:

```
set +x
```

for...in

The **for...in** structure has the following format:

for ***loop-index*** *in* ***argument-list***
do
        commands
done

This structure (Figure 28-4) assigns the value of the first argument in the ***argument-list*** to the ***loop-index*** and executes the ***commands*** between the **do** and **done** statements. The **do** and **done** statements mark the beginning and end of the **for** loop.

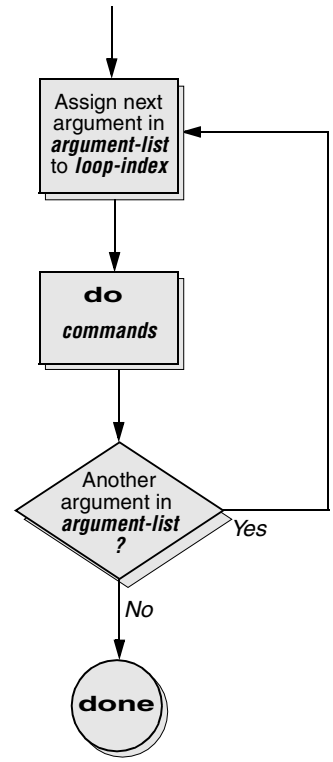

Figure 28-4 A **for...in** flowchart

After it passes control to the **done** statement, the structure assigns the value of the second argument in the *argument-list* to the *loop-index* and repeats the *commands*. The structure repeats the *commands* between the **do** and **done** statements: once for each argument in the *argument-list*. When the structure exhausts the *argument-list*, it passes control to the statement following **done**.

The following **for...in** structure assigns **apples** to the user-created variable **fruit** and then displays the value of **fruit**, which is **apples**. Next the structure assigns **oranges** to **fruit** and repeats the process. When it exhausts the argument list, the structure transfers control to the statement following **done**, which displays a message:

```
$ cat fruit
for fruit in apples oranges pears bananas
do
    echo "$fruit"
done
echo "Task complete."
```

```
$ fruit
apples
oranges
pears
bananas
Task complete.
```

The next script lists the names of the directory files in the working directory by looping over all the files, using test to determine which are directory files:

```
$ cat dirfiles
for i in *
do
    if [ -d "$i" ]
        then
            echo "$i"
    fi
done
```

The ambiguous file reference character * stands for all files (except invisible files) in the working directory. Prior to executing the **for** loop, the shell expands the * and uses the resulting list to assign successive values to the index variable **i**.

for

The **for** control structure has the following format:

for ***loop-index***
do
        commands
done

In the **for** structure the ***loop-index*** automatically takes on the value of each of the command line arguments, one at a time. It performs a sequence of commands, usually involving each argument in turn.

The following shell script shows a **for** structure displaying each of the command line arguments. The first line of the shell script, **for arg**, implies **for arg in "$@"**, where the shell expands **"$@"** into a list of quoted command line arguments **"$1" "$2" "$3"**.... The balance of the script corresponds to the **for...in** structure:

```
$ cat for_test
for arg
do
    echo "$arg"
done

$ for_test candy gum chocolate
candy
gum
chocolate
```