1

The Boot Process

COPYRIGHTED MATERIAL

This chapter looks at what happens during a Linux boot. It examines the processes that take place and the configuration files that are read. Booting is a critical part of the operation of a server. The boot process brings all of the network hardware online and starts all of the network daemon processes when the system is powered-up. If the server will not boot, it is unavailable to all of the users and computers that depend on it. For this reason, it is essential that the administrator of a network server understand the boot process and the configuration files involved in that process. After all, you're the person who maintains those configuration files and who is responsible for recovering the system when it won't boot.

The term *boot* comes from *bootstrap loader*, which in turn comes from the old saying "pull yourself up by your bootstraps." The meaning of this expression is that you must accomplish everything on your own without any outside help. This is an apt term for a system that must start from nothing and finish running a full operating system. When the boot process starts, there is nothing in RAM—no program to load the system. The loader that begins the process resides in non-volatile memory. On PC systems, this means that the loader is part of the ROM BIOS.

Booting a Linux PC is a multistep procedure. It involves basic PC functions as well as Linux processes. This complex process begins in the PC ROM BIOS; it starts with the ROM BIOS program that loads the boot sector from the boot device. The boot sector either contains or loads a Linux boot loader, which then loads the Linux kernel. Finally,

the kernel starts the `init` process, which loads all of the Linux services. The next few sections discuss this process in detail.

NOTE Two Linux loaders, LILO and GRUB, are covered in this chapter. LILO is given the bulk of the coverage because it is the default for most Linux distributions. GRUB is covered because it is the default loader for Red Hat Linux 7.2.

Loading the Boot Sector

The ROM BIOS is configured through the BIOS setup program. Setup programs vary among different BIOS versions, but all of them allow the administrator to define which devices are used to boot the system and the order in which those devices are checked. On some PC systems, the floppy drive and the first hard drive are the boot devices, and they are checked in that order. Systems that permit booting from the CD-ROM usually list the CD-ROM as the first boot device, followed by the first hard drive.

For an operational Linux server, set the ROM BIOS to check the floppy first and then the hard drive, even if you used a bootable CD-ROM for the initial installation. The reason for this is simple: The floppy is used to reboot an operational system when the hard drive is corrupted; the CD-ROM is only booted to install or upgrade the system software. During an installation, the system is offline, and you have plenty of time to fiddle with a BIOS setup program. But during an outage of an operational server, time is critical. You want to be able to reboot Linux and fix things as quickly as possible.

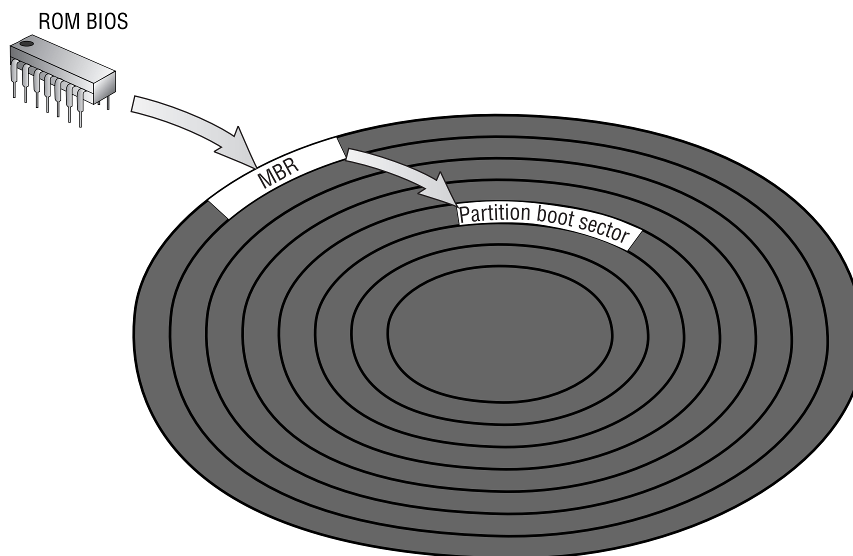
The first 512 bytes of a disk contain a boot sector. The ROM BIOS loads the boot sector from the boot device into memory, and transfers control to it. The bootstrap program from the boot sector then loads the operating system.

Floppy disks have only one boot sector, but hard disks may have more than one because each partition on a hard drive has its own boot sector. The first boot sector on the entire hard disk is called the *master boot record* (MBR). It is the only boot sector loaded from the hard drive by the ROM BIOS. The MBR contains a small loader program and a partition table. If the standard DOS MBR is used, it loads the boot sector from the active partition and then passes control to the boot sector. Thus, both the MBR and the active partition's boot sector are involved in the boot process.

Figure 1.1 shows how the boot process flows from the BIOS to the MBR and then to the partition's boot sector. This figure assumes a DOS MBR and a Linux loader in the boot sector of the active partition. Alternatively, the Linux loader can be installed in the MBR to eliminate one step in the boot process.

NOTE Appendix A, "Installing Linux," discusses the pros and cons of placing the Linux loader in the MBR.

Figure 1.1 The boot process flow



The BIOS may introduce some limitations into the Linux boot process. The Linux kernel can be installed anywhere on any of the disks available to the system, but if it is outside of those limits, the system might not be able to boot. The Linux loader depends on BIOS services. Some versions of BIOS only permit the loader to access the first two IDE hard drives: `/dev/hda` and `/dev/hdb`. Additionally, in some cases, only the first 1024 cylinders of these disks can be used when booting the system. These limitations are at their worst on old systems. New systems have two IDE disk controllers that provide access to four disk drives, and these controllers address up to 8GB of disk storage within the 1024-cylinder limit. A very old system might address only 504MB in 1024 cylinders!

For a server installation, this is not a real problem. Because servers do not dual-boot, everything can be removed from the disk, and the Linux boot files can be installed in the first partition without difficulty.

A desktop client is a different matter. Most desktops have Microsoft Windows installed in the first partition. If there is available space within the first 1024 cylinders on the first

disk drive, use `fips` to create empty space and install the Linux boot partition there. (Partitioning is discussed in detail in Appendix A.) Otherwise, a client system that dual-boots is forced to use one of the following methods:

- Install the Linux boot loader in the MBR of the first disk, and install the Linux boot partition in the first 1024 cylinders of the second disk.
- Use LOADLIN, SYSLINUX, System Commander, or a similar product to boot Linux from DOS instead of booting the system directly to Linux.
- Make a complete backup of Microsoft Windows, and repartition the disk so that both Windows and Linux are in the first 1024 cylinders. This, of course, requires a complete reinstallation of Windows.
- Create a Linux boot directory within the Windows directory structure that contains the Linux kernel and all of the files from the `/boot` directory.
- Upgrade the BIOS. This is not as difficult as it may sound. Most systems allow the BIOS to be upgraded, and many motherboard manufacturers and BIOS manufacturers have BIOS upgrades on their websites. However, don't undertake this lightly! A problem during the upgrade can leave the system unusable, and send you scurrying to the computer store to buy a replacement BIOS chip.
- Make a boot floppy or CD-ROM, and use that to start Linux. This is frequently the easiest option.

Don't be overly concerned about this potential problem. It is not a concern for servers, and even on clients it is rare. I have installed many Linux systems and have only had this problem once. In that case, it was a very old system that could directly address only 504MB per disk drive. My solution was to give the user a 250MB drive from my junk drawer as a second disk. (I never throw anything away.) I installed LILO in the MBR of his first disk and Linux on the second disk. The user was happy, Linux was installed, and I had less junk in my drawer.

Even though there are several options for loading Linux, only a few are widely used. Most systems use the Linux loader LILO. The Red Hat Linux 7.2 system defaults to using GRUB. This chapter covers both of these commonly used loaders. We start with a close look at the default GRUB configuration generated by the Red Hat installation program.

Loading Linux with GRUB

During the installation of Red Hat Linux 7.2, you're asked to select which boot loader should be used. By default, Red Hat uses the Grand Unified Bootloader (GRUB), and creates a GRUB configuration based on the values you select during the installation. Listing 1.1

shows the GRUB configuration generated by the Red Hat installation program for a desktop client. A dual-boot client configuration is used as an example because it is slightly more complex than a server configuration (servers do not usually dual-boot).

Listing 1.1 The Default GRUB Configuration

```
[root]# cat /etc/grub.conf
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE:  You do not have a /boot partition.  This means that
#          all kernel and initrd paths are relative to /, eg.
#          root (hd0,2)
#          kernel /boot/vmlinuz-version ro root=/dev/hda3
#          initrd /boot/initrd-version.img
#boot=/dev/hda
default=0
timeout=10
splashimage=(hd0,2)/boot/grub/splash.xpm.gz
password --md5 $1$LºÒCX≤Ëª$qgeIevUEDvvQAmrm4jCd31
title Red Hat Linux (2.4.7-10)
        root (hd0,2)
        kernel /boot/vmlinuz-2.4.7-10 ro root=/dev/hda3
        initrd /boot/initrd-2.4.7-10.img
title DOS
        rootnoverify (hd0,0)
        chainloader +1
```

The GRUB configuration is stored in `grub.conf`, which is a simple text file. Lines that begin with # are comments, and the Red Hat installation program inserts several comments at the beginning of the file.

The first active command line in this configuration is `default=0`. This command identifies which operating system should be booted by default in a dual-boot configuration. The operating systems that are available to GRUB are defined at the end of the configuration. Each operating system is assigned a number, sequentially starting from 0. Thus, the first operating system defined is 0, the second is 1, the third is 2, and so on. This configuration defines two operating systems: Red Hat Linux and DOS. Red Hat Linux is listed first; therefore, it is operating system 0, and it is the operating system that will be booted by default. In this case, the command `default=0` is not really required because `default` is set to 0 whenever the `default` command is not included in the configuration. However, including the command makes a clean, self-documenting configuration.

The second active line, `timeout=10`, also relates to the default boot. The `timeout` command sets the number of seconds the operator has to interrupt the boot process before GRUB automatically loads the default operating system. In this example, the operator has 10 seconds to select the alternate operating system before Red Hat Linux is automatically booted. Even for systems that do not dual-boot, set a value for `timeout` because this allows the operator to interrupt the boot process if it is necessary to pass arguments to the kernel. Providing kernel input at the boot prompt is covered later in this chapter.

The `splashimage` command points to a file that contains the background image displayed by GRUB. During the `timeout` period, GRUB displays a boot menu. The `splashimage` file is the background displayed behind that menu.

During the initial installation of Red Hat Linux 7.2, you have an opportunity to enter a GRUB password. The password entered at that time is stored in the `grub.conf` file using the `password` command. The password "Wats?Watt?" was entered during the installation of our sample system. Note that the password is not stored as clear text. The password is encrypted, and the `--md5` option on the `password` command line lets us know that the password is encrypted with the Message Digest 5 (MD5) algorithm. The operator must enter the correct password to gain access to the full range of GRUB features. The operator can boot any of the operating systems listed in the GRUB menu without entering the password; however, optional input, such as kernel parameters, cannot be entered without the correct password. If the `password` command is not included in the `grub.conf` file, a password is not required to access any GRUB features.

The `title` command defines the exact text that will be displayed in the GRUB menu to identify an operating system. The commands that follow a `title` command and occur before the next `title` command describe an operating system to the boot loader. The sample configuration defines the following two operating systems:

```
title Red Hat Linux (2.4.7-10)
        root (hd0,2)
        kernel /boot/vmlinuz-2.4.7-10 ro root=/dev/hda3
        initrd /boot/initrd-2.4.7-10.img
title DOS
        rootnoverify (hd0,0)
        chainloader +1
```

The first `title` command defines the menu text `Red Hat Linux (2.4.7–10)`. The next three lines define the operating system that is booted when that item is selected from the GRUB menu:

`root (hd0,2)` Defines the physical location of the filesystem root for this operating system. The values defined for the `root` command are the disk device name and the partition number. Notice that GRUB device names are slightly different from normal Linux device names. GRUB calls the first hard disk `hd0`. Additionally, GRUB counts partitions differently than Linux does. GRUB counts from 0, whereas Linux counts from 1. Thus, the GRUB value `hd0,2` on a Linux system that boots from an IDE drive is the same as the Linux value `hda,3`–partition number 3 on the first IDE drive.

`kernel /boot/vmlinuz-2.4.7-10 ro root=/dev/hda3` Identifies the file that contains the operating system that is to be started, and defines any arguments passed to that operating system at run time. In this case, GRUB will load the Linux kernel stored in `vmlinuz-2.4.7-10`, and it will pass the Linux kernel the arguments `ro root=/dev/hda3`, which tell the kernel where the filesystem root is located, and that it should be mounted as read-only. The `ro` option causes Linux to mount the `root` read-only during the initial phase of the boot. (Later, the `rc.sysinit` script changes it to read-write after successfully completing the filesystem checks.)

`initrd /boot/initrd-2.4.7-10.img` Identifies a ramdisk file for Linux to use during the boot. Red Hat uses the ramdisk to provide Linux with critical modules that the kernel might need to access the disk drives.

The last `title` command defines the `DOS` menu entry. Two commands define the operating system loaded when DOS is selected from the menu:

`rootnoverify (hd0,0)` Like the `root` command, defines the physical location of the filesystem root for this operating system. But `rootnoverify` tells GRUB that the filesystem found at this location does not comply with the multiboot standards, and thus cannot be validated.

`chainloader +1` Emulates the function of the DOS MBR by simply loading the specified sector and passing boot responsibilities to the loader found there. The value `+1` is a *blocklist* value, which defines the sector address of the loader relative to the partition defined by the `rootnoverify` command. `+1` means the first sector of the partition. Taken together, the `rootnoverify` command and the `chainloader` command from our sample mean that GRUB will pass control to the loader found in the first sector of the first partition on the first IDE drive when DOS is selected from the GRUB menu. In this example, that partition contains the DOS boot loader that will be responsible for loading DOS.

The `grub.conf` file on your system will be very similar to the one in this example. The location of files may be different, and a server system's configuration usually won't define multiple operating systems, but the commands will be essentially the same.

GRUB is used with several different flavors of UNIX. It is not, however, the only boot loader used with Linux—or even the most popular Linux boot loader. Red Hat, prior to 7.2, used LILO, and most other versions of Linux still do. The next section takes a close look at LILO configuration.

Loading the Kernel with LILO

Although GRUB is a newer tool, LILO, the Linux loader, is still a versatile tool that can manage multiple boot images; and can be installed on a floppy disk, in a hard disk partition, or as the master boot record. As with GRUB, this power and flexibility comes at the price of complexity, which is illustrated by the large number of LILO configuration options.

LILO Configuration Options

Most of the time, you don't need to think about the complexity of LILO; the installation program will lead you through a simple LILO installation. It is for those times when the default installation doesn't provide the service you want that you need to understand the intricacies of LILO.

LILO is configured by the `/etc/lilo.conf` file. Listing 1.2 is the `lilo.conf` file created by a Linux installation program on a desktop client that is configured to dual-boot. Its function is very similar to the GRUB sample shown in Listing 1.1.

Listing 1.2 A Sample *lilo.conf* File

```
# global section
boot=/dev/hda3
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
message=/boot/message
default=linux
# The Linux boot image
image=/boot/vmlinuz-2.4.7-10
    label=linux
```

```
    read-only
    root=/dev/hda3
# additional boot image
other=/dev/hda1
    optional
    label=dos
```


With this configuration, the user has five seconds to select either `dos` to boot Microsoft Windows or `linux` to boot Linux. If the user does not make a selection, LILO boots Linux after the five seconds have expired. The following section examines each line in this file to see how LILO is configured.

A Sample *lilo.conf* File

A `lilo.conf` file starts with a global section that contains options that apply to the entire LILO process. Some of these entries relate to the installation of LILO by `/sbin/lilo`, and are only indirectly related to the boot process.

> **NOTE** The program `/sbin/lilo` is not the boot loader. The LILO boot loader is a simple loader stored in a boot sector. `/sbin/lilo` is the program that installs and updates the LILO boot loader.

Comments in the `lilo.conf` file start with a sharp sign (#). The first active line of the global section in the sample file identifies the device that contains the boot sector. The option `boot=/dev/hda3` says that LILO is stored in the boot sector of the third partition of the first IDE disk drive. This tells us two things: where LILO is installed and where it isn't installed. LILO is not installed in the MBR of this system; it is installed in `hda3`, which must be the active partition.

The configuration option `map=/boot/map` defines the location of the map file, which contains the physical locations of the operating system kernels in a form that can be read by the LILO boot loader. (GRUB does not require a map file because it can read Linux filesystems directly.) `/boot/map` is the default value for the `map` option, so, in this case, it does not really need to be explicitly defined in the sample configuration file.

The `install=/boot/boot.b` line defines the file that `/sbin/lilo` installs in the boot sector. (`boot.b` is the LILO boot loader.) In this case, the line is not actually required because `/boot/boot.b` is the default value for `install`.

The `prompt` option causes the boot prompt to be displayed. If the `prompt` option is not included in the `lilo.conf` file, the user must press a Shift, Ctrl, or Alt key; or set the Caps Lock or Scroll Lock key to get the boot prompt. The message displayed at the boot

prompt is contained in the file identified by the `message` option. In the example, `message` points to a file named `/boot/message` that contains a full-screen display. If the `message` option is not used, the default boot prompt `boot:` is used.

The `timeout` entry defines how long the system should wait for user input before booting the default operating system. The time is defined in tenths of seconds. Therefore, `timeout=50` tells the system to wait five seconds.

> ***WARNING*** Don't use `prompt` without `timeout`. If the `timeout` option is not specified with the `prompt` option, the system will not automatically reboot. It will hang at the boot prompt, waiting for user input, and will never time out. This could be a big problem for an unattended server.

If the timeout is reached, the default kernel is booted. The `default` option identifies the default kernel. In Listing 1.2, the operating system that has the label "`linux`"is the one that will be started by default. To boot Microsoft Windows as the default operating system, simply change the `default` option to `default=dos`. The remainder of this configuration file provides the information that LILO needs to find and boot either Linux or Windows.

The `image` statement specifies the location of the Linux kernel, which is `/boot/vmlinuz-2.4.7-10` in this example. The `image` option allows you to put the Linux kernel anywhere and name it anything. The ability to change the name of the kernel comes in very handy when you want to do a kernel upgrade, which is discussed in Chapter 13, "Troubleshooting."

There are several "per-image" options used in the configuration file, some of which are specific to kernel images. The `label=linux` option defines the label that is entered at the boot prompt to load this image. Every image defined in the sample file has an associated label entry; if the operator wants to boot an image, they must enter its label.

The next option, `read-only`, is also kernel-specific. It applies to the root filesystem described previously. The `read-only` option tells LILO that the root filesystem should be mounted read-only. This protects the root filesystem during the boot and ensures that the filesystem check (`fsck`) runs reliably. Later in the startup process, the root will be remounted as read/write after `fsck` completes. See the discussion of `rc.sysinit` later in this chapter.

The `root=/dev/hda3` option is also kernel-specific. It defines the location of the root filesystem for the kernel. The `lilo.conf` file should have a root option associated with the kernel image. If it is not defined here, the root filesystem must be defined separately with the `rdev` command. However, don't do that; define the root in the LILO configuration.

The last three lines in the sample file define the other operating system that LILO is able to boot. The other OS is located in partition 1 of the first IDE drive, `other=/dev/hda1`. As the `label=dos` entry indicates, it is Microsoft Windows. The `optional` command tells `/sbin/lilo`, which is called the *mapper*, that when it builds the map file, it should consider this operating system optional. That means that `/sbin/lilo` should complete building the map file, even if this operating system is not found.

Whenever you modify the LILO configuration, invoke `/sbin/lilo` to install the new configuration. Until `/sbin/lilo` is run and maps the new configuration options, they have no effect. The `grub.conf` file, on the other hand, does not require any special processing. Changes to the GRUB configuration take effect immediately.

Only Linux and one other operating system appear in the sample file, which is the most common case for desktop clients. However, LILO can act as the boot manager for up to 16 different operating systems. It is possible to see several `other` and `image` options in a `lilo.conf` file. Multiple `image` options are used when testing different Linux kernels. The most common reason for multiple `other` options is a training system in which users boot different OSs to learn about them. In an average operational environment, only one operating system is installed on a server, and no more than two operating systems are installed on a client.

lilo.conf Hardware Options

There are many more `lilo.conf` configuration options than those described previously, but you won't need to use most of them. The sample configuration file in Listing 1.2 is almost identical to the one built by the installation program on any other system. Basically, the small subset of options just described includes the options used to build 99 percent of all LILO configuration files.

The one percent of systems that cannot be configured with the usual commands are often those systems with hardware difficulties. The `lilo.conf` file provides several options for dealing with hardware problems.

The `lba32` option is used when the boot partition is placed above the 1024-cylinder limit. This option requires a BIOS that supports 32-bit Logical Block Addresses (LBA32) for booting. The Red Hat installation program displays a "Force use of LBA32" check box in the boot loader installation screen. If this is available in your BIOS, it is the simplest way to boot from beyond the 1024-cylinder barrier.

The `linear` option forces the system to use linear sector addresses—sequential sector numbers—instead of traditional cylinder, head, and sector addresses. This is sometimes

necessary to handle large SCSI disks. It is even possible to manually define the disk geometry and linear addresses of the partitions directly in the LILO configuration file. For example:

```
disk=/dev/hda
 bios=0x80
 sectors=63
 heads=32
 cylinders=827
 partition=/dev/hda1
  start=63
 partition=/dev/hda2
  start=153216
 partition=/dev/hda3
  start=219744
```

This example defines the geometry for the first disk drive, which normally has the BIOS address of hexadecimal 80. The sectors, heads, and cylinders of the disk are defined. In the example, the linear address for the start of each partition is also given. This is an extreme example of defining the disk drive for the system; I have never had to do this.

The `append` command is another LILO option related to defining hardware. (I have used this one.) The `append` option passes a configuration parameter to the kernel. The parameter is a kernel-specific option used to identify hardware that the system failed to automatically detect. For example:

```
append = "ether=10,0x210,eth0"
```

This sample command tells the kernel the nonstandard configuration of an Ethernet card. This particular option line says that the Ethernet device `eth0` uses IRQ 10 and I/O port address 210. (The format of the parameters that can be passed to the kernel is covered in "The Linux Boot Prompt," later in this chapter.)

Linux is very good at detecting the configuration of Ethernet hardware, and software-configurable cards are good at reporting their settings. Additionally, new PCI cards do not require all of these configuration values. By and large, kernel parameters are not needed to boot the system. However, this capability exists for those times when you do need it.

LILO Boot Security

Two LILO configuration commands enhance the security of a network server. If the server is in an unsecured area, it is possible for an intruder to reboot the system and gain unauthorized access. For example, an intruder could reboot the server into single-user mode and essentially have password-free root access to part of the system. (More about single-user mode later. For now, just take my word that this can be done.)

To prevent this, add the `password` and the `restricted` options to the `lilo.conf` file. The `password` option defines a password that must be entered to reboot the system. The password is stored in the configuration file in an unencrypted format, so make sure the `lilo.conf` file can be read only by the root user. The `restricted` option softens the security a little. It says that the password is required only when passing parameters to the system during a boot. For example, if you attempt to pass the parameter `single` to the system to get it to boot into single-user mode, you must provide the password.

Always add the `restrict` option when using the `password` option in a server's `lilo.conf` file. Using `password` without `restrict` can cause the server to hang during the boot until the password is entered. If the server console is unattended, the boot can hang for an extended period of time. Using `restrict` with the `password` option ensures that the system reboots quickly after a crash, while providing adequate protection from unauthorized access through the console.

The following example includes restricted password protection for booting the Linux kernel. The example is based on the `lilo.conf` file you saw earlier, with a few lines removed that contain default values to show that you can remove those lines and still boot without a problem. Listing 1.3 uses `cat` to list the new configuration file and `lilo` to process it.

Listing 1.3 Adding Password Protection to LILO

```
[root]# cat lilo.conf
# global section
boot=/dev/hda3
prompt
timeout=50
message=/boot/message
default=linux
# the Linux boot image
image=/boot/vmlinuz-2.4.2-2
    label=linux
    read-only
    root=/dev/hda3
```

```
    password=Wats?Watt?
    restricted
# additional boot images
other=/dev/hda1
    optional
    label=dos
 [root]# lilo
Added linux *
Added dos
```

After running `/sbin/lilo`, reboot. Note that you don't have to enter the password at the boot prompt because the configuration includes the `restrict` option. However, if you attempt to boot the system and provide optional input at the boot prompt, you will be asked for the password.

The Linux Boot Prompt

The LILO and GRUB processes are modified through their configuration files. The kernel boot process is modified through input to the boot prompt. As with the LILO `append` option and the GRUB `kernel` command, the boot prompt is used to pass parameters to the kernel. The difference, however, is that the boot prompt is used to manually enter kernel parameters, whereas the `append` and `kernel` commands are used to automate the process when the same parameters must be passed to the kernel for every boot. Use the boot prompt for special situations, such as repairing a system or getting an unruly piece of equipment running; or to debug input before it is stored in the `lilo.conf` or `grub.conf` file.

You rarely need to pass parameters to the kernel through the boot prompt. When you do, it is either to change the boot process or to help the system handle a piece of unknown hardware. The `kernel` command from the `grub.conf` file shown in Listing 1.1 is an example of using boot input to change the boot process:

```
kernel /boot/vmlinuz-2.4.7-10 ro root=/dev/hda3
```

This line comes from the `grub.conf` file, but it also can be entered interactively during the boot process. When the GRUB menu is displayed at boot time, the operator is given 10 seconds to select an optional menu item, or interrupt the boot process. Interrupt the boot by pressing the Escape key. If a password is defined in the `grub.conf` file, press P, and enter the GRUB password. Then, press C for command mode, and a command line prompt appears. This is the boot prompt that allows arguments to be sent to the kernel using the `kernel` command interactively. The format of the `kernel` command is

`kernel` *`file arguments`*

where `kernel` is the command, *`file`* is the name of the file that contains the Linux kernel, and *`arguments`* are any optional arguments you wish to pass to the kernel. In the preceding `kernel` command example, `ro root=/dev/hda3` are arguments that change the default boot behavior so that the root filesystem is mounted read-only. The possible arguments depend on the kernel, not on whether GRUB or LILO is used to control the boot process. Any of the kernel arguments described in this section can be sent to the kernel in this manner on a system that uses GRUB. The LILO boot prompt is different, but the function is the same.

When the system is booted by LILO, the string `boot:` is displayed as the boot prompt. The operator can boot any operating system defined in the `lilo.conf` file by entering its name at the prompt (for example, `linux`, or `dos`). Arguments are passed to the selected operating system by placing them on the command line after the operating system name. An example of passing kernel parameters on a system booted by LILO is

```
boot: linux panic=60
```

In this example, `boot:` is the prompt, `linux` is the kernel name, and `panic=60` is the parameter passed to that kernel. The keyword `linux` is the label assigned to the Linux kernel in the LILO configuration. Use the label to tell LILO which kernel should receive the parameter. The panic argument changes the boot behavior after a system crash. It is possible for the Linux kernel to crash from an internal error, called a *kernel panic*. If the system crashes from a kernel panic, it does not automatically reboot—it stops at the boot prompt waiting for instructions.

Normally, this is a good idea. The exception is an unattended server. If you have a system that does not have an operator in attendance and that remote users rely on, it might be better to have it try an automatic reboot after it crashes. The example shown previously tells the system to wait 60 seconds and then reboot.

NOTE This might surprise Windows administrators, but I have never had a Linux system crash. In fact, I had one specialized system (collecting network measurement data, and providing Web access to that data) that ran continuously for more than a year without a single problem.

In a normal boot process, the kernel starts the `/sbin/init` program. Using the `init` argument, it is possible to tell the kernel to start another process instead of `/sbin/init`. For example, `init=/bin/sh` causes the system to run the shell program, which then can be used to repair the system if the `/sbin/init` program is corrupted.

Booting directly to the shell looks very much like booting to single-user mode with the `single` argument, but there are differences. `init=/bin/sh` does not rely on the `init`

program. `single`, on the other hand, is passed directly to `init` so that `init` can perform selected initialization procedures before placing the system into single-user mode. In both of these cases, the person who boots the computer is given password-free access to the shell unless `password` and `restrict` are defined in the `lilo.conf` file, as described in the previous section.

Handling undetected hardware is the second reason for entering data at the boot prompt, and it is the most common reason for doing so during the initial installation. Sometimes, the system has trouble detecting hardware or properly detecting the hardware's configuration. In those cases, the system needs your input at the boot prompt to properly handle the unknown hardware.

A large number of the boot input statements pass parameters to device driver modules. For example, there are about 20 different SCSI host adapter device drivers that accept boot parameters. In most cases, the system detects the SCSI adapter configuration without a problem. But if it doesn't, booting the system may be impossible. An example of passing kernel parameters to Linux to identify an undetected SCSI adapter device is

```
boot: linux aha152x=0x340,11,7
```

All hardware parameters begin with a driver name. In this case, it is the `aha152x` driver for Adaptec 1520 series adapters. The data after the equal sign is the information passed to the driver. In this case, it is the I/O port address, the IRQ, and the SCSI ID.

Another boot argument that is directly related to the configuration of device drivers is the `reserve` argument. `reserve` defines an area of I/O port address memory that is protected from *auto-probing*. To determine the configuration of their devices, most device drivers probe those regions of memory that can be legitimately used for their devices. For example, the 3COM EtherLink III Ethernet card is configured to use I/O port address 0x300 by default, but it can be configured to use any of 21 different address settings from 0x200 to 0x3e0. If the 3c509 driver did not find the adapter installed at address 0x300, it could legitimately search all 21 base address regions. Normally, this is not a problem. On occasion, however, auto-probing can return the wrong configuration values. In extreme cases, poorly designed adapters can even hang the system when they are probed. I have never personally seen an adapter hang the system, but some years ago I had an Ethernet card that returned the wrong configuration. In that case, I combined the `reserve` argument with device driver input, as in this example:

```
boot: linux reserve=0x210,16 ether=10,0x210,eth0
```

This boot input prevents device drivers from probing the 16 bytes starting at memory address 0x210. The second argument on this line passes parameters to the `ether` device driver. It tells that driver that the Ethernet adapter uses interrupt 10 and I/O port address

0x210. This specific adapter will be known as device `eth0`, which is the name of the first Ethernet device. Of course, you'll want to use the Ethernet adapter every time the system boots. Once you're sure this boot input fixes the Ethernet problem, store it as a kernel-specific option in the `lilo.conf` file. For example:

```
image = /boot/vmlinuz-2.2.5-15
    label = linux
    root = /dev/hda3
    read-only
    append = "reserve=0x210,16 ether=10,0x210,eth0"
```

The `ether` argument is also used to force the system to locate additional Ethernet adapters. Suppose that the system detects only one Ethernet adapter, and you have two Ethernet devices installed: `eth0` and `eth1`. Use this boot input to force the system to probe for the second device:

```
ether=0,0,eth1
```

Old Ethernet cards are a major reason for boot prompt input. If you have an old card and experience a problem, read the Ethernet-HOWTO for configuration advice on your specific card. New PCI Ethernet cards do not usually require boot input. Most current Ethernet cards use loadable modules for device drivers. If your Ethernet card is not recognized during the boot, it may be that its module is not loaded. The first step is to check the module's configuration.

NOTE See the "Loadable Modules" section later in this chapter for information about managing modules and for specific examples of loadable modules used for Ethernet device drivers.

This section has barely touched upon the very large number of arguments that can be entered at the boot prompt. See the "BootPrompt-HOWTO" document, by Paul Grotmaker, for the details of all of them. Most Linux systems include the HOWTO documents in `/usr/doc`.

Hardware Device Driver Initialization

When the system boots, several things happen. You have already seen the part that LILO and GRUB play in loading the operating system, but that is only the beginning. These loaders start the Linux kernel running, and then things *really* begin to happen.

The kernel is the heart of Linux. It loads into memory and initializes the various hardware device drivers. Most of the possible boot prompt arguments are intended to help the kernel initialize hardware, and the messages the kernel displays during startup help you determine what hardware is installed in the system and whether it is properly initialized.

Use the `dmesg` command to display the kernel startup messages; combine it with the `less` command or with `grep` to examine the startup messages more effectively. `less` allows you to scroll through the messages, one screenful at a time; `grep` permits you to search for something specific in the `dmesg` output. For example, combine `dmesg` and `grep` to locate kernel messages relating to the initialization of the Ethernet device `eth0`:

```
$ dmesg | grep eth0
loading device 'eth0'...
eth0: SMC Ultra at 0x340, 00 00 C0 4F 3E DD, IRQ 10 memory 0xc8000-
0xcbfff.
```

This message clearly shows the type of Ethernet adapter used (SMC Ultra) and the Ethernet MAC address assigned to the adapter (00 00 C0 4F 3E DD). Additionally, because the SMC Ultra is an ISA bus adapter, the bus interrupt (IRQ 10), the adapter memory address (0xc8000-0xcbfff), and the I/O port address (0x340) are shown. All of this information is useful for debugging a hardware configuration problem.

Most systems use Ethernet for all network communications, although some use other devices, such as serial ports for this purpose. When the kernel initializes the serial ports, it displays the device name, I/O port address, and IRQ of each serial port. It also displays the model of Universal Asynchronous Receiver Transmitter (UART) that is used for the serial interface. Old systems used 8250 UARTs, which are inadequate for use with modems and a problem for systems that need to run PPP. As this example shows, current systems use the faster 16550A UARTs:

```
ttyS00 at 0x03f8 (irq = 4) is a 16550A
ttyS02 at 0x03e8 (irq = 4) is a 16550A
```

Also of interest for a network server are the kernel components of TCP/IP. These components include the fundamental protocols, such as IP (Internet Protocol), and the network sockets interface. Sockets is an application protocol interface developed at Berkeley for BSD Unix. It provides a standard method for programs to talk to the network. The TCP/IP initialization messages from a Red Hat 7.2 system are

```
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP, IGMP
IP: routing cache hash table of 1024 buckets, 8Kbytes
```

```
TCP: Hash tables configured (established 16384 bind 16384)
Linux IP multicast router 0.06 plus PIM-SM
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
```

Reading the kernel messages helps you understand what occurs when the system starts up. Don't read these messages word for word—too many details will just bog you down. What you should do is look at the messages to gain a sense of how the system works. Of course, there are slight variations among the messages displayed on various systems, but the messages give you a very good idea of what is going on as the kernel initializes the hardware.

After the kernel concludes its portion of the boot process, the kernel starts the `init` program, which controls the rest of the startup.

Loading Linux Services—The *init* Process

The `init` process, which is process number one, is the mother of all processes. After the kernel initializes all of the devices, the `init` program runs and starts all of the software. The `init` program is configured by the `/etc/inittab` file. Listing 1.4 shows the `inittab` file that comes with Red Hat 7.2:

Listing 1.4 The *inittab* File

```
#
# inittab   This file describes how the INIT process should set up
#           the system in a certain run-level.
#
# Author:   Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>
#           Modified for RHS Linux by Marc Ewing and Donnie Barnes
#

# Default runlevel. The runlevels used by RHS are:
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS (The same as 3, if you do not have networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
#
```

```
id:5:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6

# Things to run in every runlevel.
ud::once:/sbin/update

# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now

# When our UPS tells us power has failed, schedule a shutdown for 2 minutes.
pf::powerfail:/sbin/shutdown -f -h +2 "Power Failure; System Shutting Down"

# If power was restored before the shutdown, cancel it.
pr:12345:powerokwait:/sbin/shutdown -c "Power Restored; Shutdown Cancelled"

# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6

# Run xdm in runlevel 5
# xdm is now a separate service
x:5:respawn:/etc/X11/prefdm -nodaemon
```

NOTE The comments in this sample file were edited slightly to better fit on a book page. They are a reduced version of the actual comments from the Red Hat `inittab` file.

Understanding Runlevels

To understand the `init` process and the `inittab` file, you need to understand *runlevels*, which are used to indicate the state of the system when the `init` process is complete. There is nothing inherent in the system hardware that recognizes runlevels; they are purely a software construct. `init` and `inittab` are the only reasons why the runlevels affect the state of the system. Because of this, the way runlevels are used varies from distribution to distribution. This section uses Red Hat Linux as an example.

The Linux startup process is very similar to the startup process used by System V Unix. It is more complex than the initialization on a BSD Unix system, but it is also more flexible. Like System V, Linux defines several runlevels that run the full gamut of possible system states from not-running (halted) to running multiple processes for multiple users. The comments at the beginning of the sample `inittab` file describe the runlevels:

- Runlevel 0 causes `init` to shut down all running processes and halt the system.
- Runlevel 1 is used to put the system in single-user mode. Single-user mode is used by the system administrator to perform maintenance that cannot be done when users are logged in. This runlevel may also be indicated by the letter S instead of the number 1.
- Runlevel 2 is a special multiuser mode that supports multiple users but does not support file sharing.
- Runlevel 3 is used to provide full multiuser support with the full range of services. It is the default mode used on servers that use the "text only" console logon.
- Runlevel 4 is unused by the system. You can design your own system state and implement it through runlevel 4.
- Runlevel 5 initializes the system as a dedicated X Windows terminal. This runlevel is widely used as an alternative for systems configured to launch an X desktop environment at startup. In fact, runlevel 5 is the default runlevel for most Red Hat systems because most systems are desktop clients that use an X Windows console logon.
- Runlevel 6 causes `init` to shut down all running processes and reboot the system.

All of the lines in the `inittab` file that begin with a sharp sign (#) are comments. A liberal dose of comments is needed to interpret the file because the syntax of actual `inittab` configuration lines is terse and somewhat arcane. An `inittab` entry has this general format:

`label:runlevel:action:process`

The *`label`* is a one- to four-character tag that identifies the entry. Some systems support only two-character labels. For this reason, most people limit all labels to two characters. The labels can be any arbitrary character string, but in practice, certain labels are commonly used. The label for a `getty` or other login process is usually the numeric suffix of the `tty` to which the process is attached. Other labels used in the Red Hat Linux distribution are

- `id` for the line that defines the default runlevel used by `init`
- `si` for the system initialization process
- `l`*`n`* where *n* is a number from 1 to 6 that indicates the runlevel being initialized by this process
- `ud` for the update process
- `ca` for the process run when Ctrl+Alt+Del is pressed
- `pf` for the process run when the UPS indicates a power failure
- `pr` for the process run when power is restored by the UPS before the system is fully shut down
- `x` for the process that turns the system into an X terminal

The *`runlevel`* field indicates the runlevels to which the entry applies. For example, if the field contains a 3, the process identified by the entry must be run for the system to initialize runlevel 3. More than one runlevel can be specified, as illustrated in the sample file by the `pr` entry. Entries that have an empty *`runlevel`* field are not involved in initializing specific runlevels. For example, an entry that is invoked by a special event, such as the three-finger salute (Ctrl+Alt+Del), does not have a value in the *`runlevel`* field.

The *`action`* field defines the conditions under which the process is run. Table 1.1 lists all of the valid action values and the meaning of each one.

Table 1.1 Valid Action Values

Action	Meaning
Boot	Runs when the system boots. Ignores runlevel.
Bootwait	Runs when the system boots, and `init` waits for the process to complete. Runlevels are ignored.
Ctrlaltdel	Runs when Ctrl+Alt+Del is pressed, which passes the SIGINT signal to `init`. Runlevels are ignored.
Initdefault	Doesn't execute a process. It sets the default runlevel.
Kbrequest	Runs when `init` receives a signal from the keyboard. This requires that a key combination be mapped to `KeyBoardSignal`.
Off	Disables the entry so the process is not run.
Once	Runs one time for every runlevel.
Ondemand	Runs when the system enters one of the special runlevels A, B, or C.
Powerfail	Runs when `init` receives the SIGPWR signal.
Powerokwait	Runs when `init` receives the SIGPWR signal and the file `/etc/powerstatus` contains the word OK.
Powerwait	Runs when `init` receives the SIGPWR signal, and `init` waits for the process to complete.
Respawn	Restarts the process whenever it terminates.
sysinit	Runs before any `boot` or `bootwait` processes.
wait	Runs the process upon entering the run mode, and `init` waits for the process to complete.

The last field in an `inittab` entry is the process field. It contains the process that `init` executes. The process appears in the exact format that is used to execute the process from the

command line. Therefore, the process field starts with the name of the process that is to be executed, and follows it with the arguments that will be passed to that process. For example, `/sbin/shutdown -t3 -r now`, which is the process executed when Ctrl+Alt+Del is pressed, is the same command that could be typed at the shell prompt to reboot the system.

Special-Purpose Entries

Using what you have just learned about the syntax of the `inittab` file, take a closer look at the sample in Listing 1.4. You can ignore most of the file; more than half of it consists of comments. Many of the other lines are entries that are used only for special functions:

- The `id` entry defines the default runlevel, which is usually 3 for a text console or 5 for an X console.
- The `ud` entry calls the `/sbin/update` process, which cleans up the I/O buffers before disk I/O starts in order to protect the integrity of the disks.
- The `pf`, `pr`, and `ca` entries are invoked only by special interrupts.

WARNING Some administrators are tempted to change the ca entry to eliminate the ability to reboot the system with the three-finger salute. This is not a bad idea for server systems, but don't do it for desktop systems. Users need to have a method to force a graceful shutdown when things go wrong. If it is disabled, the user might resort to the power switch, which can result in lost data and other disk troubles.

Six of the lines in the `inittab` file start—and when necessary, restart—the `getty` processes that provide virtual terminal services. One example from Listing 1.4 explains them all:

```
3:2345:respawn:/sbin/mingetty tty3
```

The label field contains a 3, which is the numeric suffix of the device, `tty3`, to which the process is attached. This `getty` is started for runlevels 2, 3, 4, and 5. When the process terminates (for example, when a user terminates the connection to the device), the process is immediately restarted by `init`.

The pathname of the process that is to be started is `/sbin/mingetty`. Red Hat uses `mingetty`, which is a minimal version of `getty` that is specifically designed for virtual terminal support. On a Caldera 2.2 system, the pathname would be `/sbin/getty` with the `VC` command-line option, which tells `getty` that it is servicing a virtual terminal. The result, however, would be the same: to start a virtual terminal service process for `tty3`.

Every runlevel that accepts terminal input uses getty. Runlevel 5 has one additional entry in the inittab file to start an X terminal:

```
x:5:respawn:/etc/X11/prefdm -nodaemon
```

This line starts—and when necessary, restarts—the X application used for the X-based console logon required by runlevel 5.

Every line in the inittab file handles some important task. However, the real heart of the inittab file consists of the seven lines that follow the comment “System initialization” near the beginning of the inittab file (refer to Listing 1.4.) They are the lines that invoke the startup scripts. The first of these is the si entry:

```
si::sysinit:/etc/rc.d/rc.sysinit
```

This entry tells init to initialize the system by running the boot script located at /etc/rc.d/rc.sysinit. This script, like all startup scripts, is an executable file that contains Linux shell commands. Notice that the entry shows the full path to the startup script. One of the most common complaints about different Linux distributions is that the key files are stored in different locations in the filesystem. Don’t worry about memorizing these differences—just look in the /etc/inittab file. It tells you exactly where the startup scripts are located.

The six lines that follow the si entry in inittab are used to invoke the startup scripts for each runlevel. Except for the runlevel involved, each line is identical:

```
l5:5:wait:/etc/rc.d/rc 5
```

This line starts all of the processes and services needed to provide the full multiuser support defined by runlevel 5. The label is l5, which is symbolic of level 5. The runlevel is, of course, 5. init is directed to wait until the startup script terminates before going on to any other entries in the inittab file that relate to runlevel 5. init executes the script /etc/rc.d/rc, and passes that script the command-line argument 5.

Startup Scripts

Anything that can be run from a shell prompt can be stored in a file and run as a shell script. System administrators use this capability to automate all kinds of processes; Linux uses this capability to automate the startup of system services. Two main types of scripts are used: the *system initialization* script and the *runlevel initialization* script.

System Initialization

The system initialization script runs first. On a Red Hat system, this is a single script named `/etc/rc.d/rc.sysinit`. Other Linux distributions might use a different filename, but all versions of Linux use script files to initialize the system. The `rc.sysinit` script performs many essential system initialization tasks, such as preparing the network and the filesystems for use.

The `rc.sysinit` script begins the network initialization by reading the `/etc/sysconfig/network` file, which contains several network configuration values set during the initial installation. If the file is not found, networking is disabled. If it is found, the script assigns the system the hostname stored there.

The initialization script performs many small but important tasks, such as setting the system clock, applying any keyboard maps, and starting USB and PnP support. The bulk of the script, however, is used to prepare the filesystem for use. The script activates the swap file, which is necessary before the swap space is used. The `rc.sysinit` script also runs the filesystem check, using the `fsck` command to check the structure and integrity of the Linux filesystems. If a filesystem error is encountered that `fsck` cannot simply repair, the boot process stops, and the system reboots in single-user mode. You then must run `fsck` manually, and repair the disk problems yourself. When you finish the repairs, exit the single-user shell. The system will then attempt to restart the interrupted boot process from where it left off.

The initialization script mounts the `/proc` filesystem and, after the `fsck` completes, mounts the root filesystem as read-write. Recall that the root filesystem was initially mounted as read-only. The root must be remounted as read-write before the system can be used. The script also mounts other local filesystems listed in the `/etc/fstab` file. (The `fstab` file is described in Chapter 9, "File Sharing.") The `rc.sysinit` script finishes up by loading the loadable kernel modules.

Other initialization scripts may look different from Red Hat's, but they perform very similar functions. The order may be different, but the major functions are the same: initialize the swap file, and check and mount the local filesystems.

Runlevel Initialization

After the system initialization script is run, `init` runs a script for the specific runlevel. On Red Hat, Mandrake, and Caldera systems, this is done by running a control script and passing it the runlevel number. The control script, `/etc/rc.d/rc`, then runs all of the scripts that are appropriate for the runlevel. It does this by running the scripts that are stored in the directory `/etc/rcn.d`, where *n* is the specified runlevel. For example, if the

rc script is passed a 5, it runs the scripts found in the directory /etc/rc.d/rc5.d. A listing of that directory from a Red Hat system shows that there are lots of scripts:

Listing 1.5 Runlevel Initialization Scripts

```
$ ls /etc/rc.d
init.d  rc0.d  rc2.d  rc4.d  rc6.d     rc.sysinit
rc      rc1.d  rc3.d  rc5.d  rc.local
$ ls /etc/rc.d/rc3.d
K03rhnsd      K35smb       K74ntpd    S05kudzu     S25netfs        S85httpd
K16rarpd      K45arpwatch  K74ypserv  S06reconfig  S26apmd         S90crond
K20nfs        K45named     K74ypxfrd  S08ipchains  S28autofs       S90xfs
K20rstatd     K50snmpd     K75gated   S09isdn      S40atd          S95anacron
K20rusersd    K50tux       K84bgpd    S10network   S55sshd         S99linuxconf
K20rwalld     K55routed    K84ospf6d  S12syslog    S56rawdevices   S99local
K20rwhod      K61ldap      K84ospfd   S13portmap   S56xinetd
K28amd        K65identd    K84ripd    S14nfslock   S60lpd
K34yppasswdd  K73ypbind    K84ripngd  S17keytable  S80sendmail
K35dhcpd      K74nscd      K85zebra   S20random    S85gpm
```

The scripts that begin with a K are used to kill processes when exiting a specific runlevel. In Listing 1.5, the K scripts are used when terminating runlevel 5. The scripts that start with an S are used when starting runlevel 5. None of the items in rc5.d, however, is really a startup script. They are logical links to the real scripts, which are located in the /etc/rc.d/init.d directory. For example, S80sendmail is linked to /etc/init.d/sendmail. This raises the question of why the scripts are executed from the directory rc5.d instead of directly from init.d, where they actual reside. The reasons are simple. The same scripts are needed for several different runlevels. Using logical links, the scripts can be stored in one place and still be accessed by every runlevel from the directory used by that runlevel. Additionally, the order in which the scripts are executed is controlled by the script name.

The scripts are executed in alphabetical order, based on name. Thus, S10network is executed before S80sendmail. This allows the system to control, through a simple naming convention, the order in which scripts are executed. Different runlevels can execute the scripts in different orders while still allowing the real scripts in init.d to have simple, descriptive names. Listing 1.6 shows the real script names in the init.d directory:

Listing 1.6 The init.d Script Files

```
$ ls init.d
amd       functions  kdcrotate  network  rarpd       rwalld   xfs
anacron   gated      keytable   nfs      rawdevices  rwhod    xinetd
```

```
apmd      gpm       killall   nfslock  reconfig  sendmail  ypbind
arpwatch  halt      kudzu     nscd     rhnsd     single    yppasswdd
atd       httpd     ldap      ntpd     ripd      smb       ypserv
autofs    identd    linuxconf ospf6d   ripngd    snmpd     ypxfrd
bgpd      ipchains  lpd       ospfd    routed    sshd      zebra
crond     iptables  named     portmap  rstatd    syslog
dhcpd     isdn      netfs     random   rusersd   tux
```

Several of these scripts are clearly of interest to administrators of network servers:

- The `httpd` script starts the Web server.
- The `xinet` script starts the Extended Internet daemon (`xinetd`).
- The `named` script starts the DNS name server.
- The `nfs` script starts the NFS file server.
- `sendmail` starts the e-mail server.

It is useful to know where these services really start in case something goes wrong. All of these scripts may be important when troubleshooting a network problem.

Controlling Scripts

You can control which scripts are executed and the order in which they are executed by directly changing the logical links in the runlevel directory, but that's not the best way. It's easier to control startup scripts using a tool specifically designed for this purpose. Red Hat systems use the `chkconfig` command, which is a command-line tool based on the `chkconfig` program from the Silicon Graphics IRIX version of Unix. The Linux version has some enhancements, such as the capability to control which runlevels the scripts run under. The `--list` option of the `chkconfig` command displays the current settings:

```
[root]# chkconfig --list named
named 0:off 1:off 2:off 3:on 4:on 5:on 6:off
```

This example shows the structure of a `chkconfig` command line. `chkconfig` is the command, `--list` is the option, and `named` is the name of a script file found in the `init.d` directory. It is the script file that the command affects.

To enable or disable a script for a specific runlevel, specify the runlevel with the `--level` option, followed by the name of the script you wish to control and the action you wish to take, either `on` to enable the script or `off` to disable it. For example, to disable `named` for runlevel 5, enter the following:

```
[root]# chkconfig --level 5 named off
[root]# chkconfig --list named
named 0:off 1:off 2:off 3:on 4:on 5:off 6:off
```

`chkconfig` reads the comments in the `init.d` script file to determine the runlevels in which the script is run by default, and to obtain information needed to create the correct logical links in the runlevel directory. This information must be found in the script file in a comment that contains the keyword `chkconfig`. Here is an example from the `ipchains` script:

```
[root]# grep chkconfig ipchains
# chkconfig: 2345 08 92
```

In this comment, the keyword `chkconfig` is followed by three values:

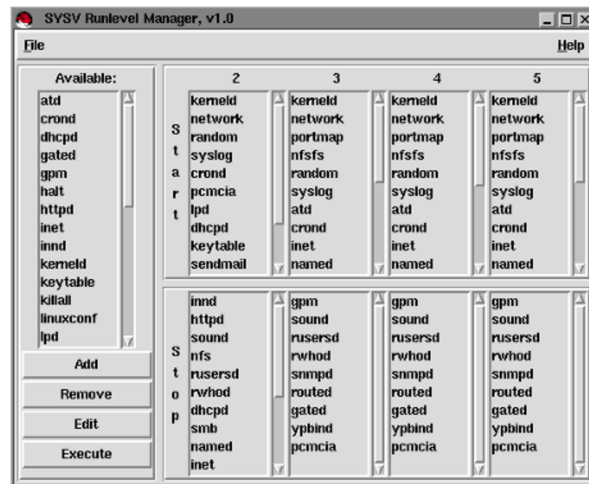
- First, the list of runlevels in which this script is run by default. Here, the list contains four runlevels (2, 3, 4, and 5). If the script is not run by default at any runlevel, this field contains a dash (-).
- Next, the numeric prefix used to name the logical link to the script file used during startup. Here, the numeric prefix used for startup is 08. Therefore, the link placed in the runlevel directory will be named `S08ipchains`.
- Finally, the numeric prefix used to name the logical link to the script file used during shutdown. Here, the numeric prefix used for shutdown is 92. Therefore, the link placed in the runlevel directory will be named `K92ipchains`.

Editing the `chkconfig` comment in the script in the `init.d` directory changes the values that `chkconfig` uses to create the links. However, this is not necessary. The values selected by Red Hat were chosen to ensure that services start in the proper order. The only time you may need to set these values is when you write your own startup script for a custom service.

`chkconfig` is used on Red Hat and several other Linux systems. It is not, however, the only widely used tool for controlling scripts. `tksysv`, the SYSV Runlevel Manager, is available on several distributions; and it runs under X Windows. Figure 1.2 shows the SYSV Runlevel Manager window.

The SYSV Runlevel Manager lists all of the available startup scripts, as well as the scripts that are currently being used by each runlevel. Each runlevel has a column of the display that is divided into Start and Stop scripts. These categories correspond to the `S` and `K` scripts in the directories. Using `tksysv`'s simple visual interface, you can add scripts to a runlevel from the list of available scripts, or delete scripts from a runlevel. You can even select a script from the list of available scripts and execute it in real time to start a service without rebooting.

Figure 1.2 The SYSV Runlevel Manager Window



The *rc.local* Script

In general, you do not directly edit boot scripts. The exception to this rule is the `rc.local` script located in the `/etc/rc.d` directory. It is the one customizable startup file, and it is reserved for your use; you can put anything you want in there. After the system initialization script and the runlevel scripts execute, the system executes `rc.local`. Since it is executed last, the values you set in the `rc.local` script are not overridden by another script.

If you add third-party software that needs to be started at boot time, put the code to start it in the `rc.local` script. Additionally, if something is not installed or configured correctly by the installation process, it can be manually configured in `rc.local`.

Loadable Modules

Loadable modules are pieces of object code that can be loaded into a running kernel. This is a very powerful feature. It allows Linux to add device drivers to a running Linux system in real time. This means that the system can boot a generic Linux kernel and then add the drivers needed for the hardware on a specific system. The hardware is immediately available without rebooting the system.

Usually, you have very little involvement with loadable modules. In general, the system detects your hardware and determines the correct modules during the initial installation.

But not always. Sometimes hardware is not detected during the installation, and other times new hardware is added to a running system. To handle these things, you need to know how to work with loadable modules.

Listing the Loaded Modules

Use the `lsmod` command to check which modules are loaded in your system. Listing 1.7 shows an example:

Listing 1.7 Listing Loaded Modules

```
$ lsmod
Module             Size  Used by
ide-cd            27072   0  (autoclean)
cdrom             28512   0  (autoclean) [ide-cd]
soundcore          4464   0  (autoclean)
parport_pc        14768   1  (autoclean)
lp                 6416   0  (autoclean)
parport           25600   1  (autoclean) [parport_pc lp]
autofs            11520   0  (autoclean) (unused)
smc-ultra          5792   1
8390               6752   0  [smc-ultra]
nls_iso8859-1      2832   1  (autoclean)
nls_cp437          4352   1  (autoclean)
vfat               9584   1  (autoclean)
fat               32384   0  (autoclean) [vfat]
ext3              64624   3
jbd               40992   3  [ext3]
```

Loadable modules perform a variety of tasks. Some modules are hardware device drivers, such as the `smc-ultra` module for the SMC Ultra Ethernet card. Other modules provide support for the wide array of filesystems available in Linux, such as the ISO8859 filesystem used on CD-ROMs or the DOS FAT filesystem with long filename support (`vfat`).

Each entry in the listing produced by the `lsmod` command begins with the name of the module followed by the size of the module. As the size field indicates, modules are small. Often, they work together to get the job done. The interrelationships of modules are called *module dependencies*, which are an important part of properly managing modules. The listing tells you which modules depend on other modules. In our sample, the `smc-ultra` driver depends on the `8390` module. You can tell that from the `8390` entry, but not from the `smc-ultra` entry. The `8390` entry lists the modules that depend on it under the heading `Used by`.

Most of the lines in Listing 1.7 contain the word `autoclean`. This means that a module can be removed from memory automatically if it is unused. `autoclean` is only one of the module options. You can select different options when manually loading modules.

Manually Maintaining Modules

Modules can be manually loaded using the `insmod` command. This command is very straightforward—it's just the command and the module name. For example, to load the 3c509 device driver, enter **`insmod 3c509`**. This does not install the module with the `autoclean` option. If you want this driver removed from memory when it is not in use, add the `–k` option to the `insmod` command, and enter **`insmod -k 3c509`**.

One limitation with the `insmod` command is that it does not understand module dependencies. If you used it to load the `smc-ultra` module, it would not automatically load the required `8390` module. For this reason, `modprobe` is a better command for manually loading modules. As with the `insmod` command, the syntax is simple. To load the `smc-ultra` drive, simply enter **`modprobe smc-ultra`**.

`modprobe` reads the module dependencies file that is produced by the `depmod` command. Whenever the kernel or the module libraries are updated, run `depmod` to produce a new file containing the module dependencies. The command `depmod -a` searches all of the standard modules libraries and creates the necessary file. After it is run, you can use `modprobe` to install any module and have the other modules it depends on automatically installed.

Use the `rmmod` command to remove unneeded modules. Again, the syntax is simple; `rmmod appletalk` removes the `appletalk` driver from your system.

These manual maintenance commands have limited utility on a running system, because the correct things are usually done by Linux without any prodding from you. For example, I booted a small system on my home network, and immediately ran `lsmod`. I saw from this listing that I had `appletalk` and `ipx` installed, and I knew I didn't need either one. I typed in **`rmmode appletalk`**, but the message returned was `rmmod: module appletalk not loaded` because the system had already removed this unneeded module faster than I could type the command. Additionally, attempting to remove a command that is currently active returns the message `Device or resource busy`. For these reasons, I have rarely needed to use the `rmmod` command on an operational system.

In Sum

This chapter has taken a network server from power up to full operation. We have gone from the ROM BIOS to the Linux boot loader to the kernel initialization to the `init` process and, finally, to the boot scripts. All of these things play an important role in starting the system, and all of them can be configured by you.

Many operating systems hide the boot details, assuming that the administrator will be confused by the messages. Linux hides nothing. It accepts the fact that ultimately you're in control of this process, and you can exercise as much or as little of that control as you want. You can modify kernel behavior with boot prompt input, and control the behavior of the Linux loader through the `lilo.conf` file or the `grub.conf` file. You configure the `init` process through the `inittab` file and control system services through the startup scripts. All of these configuration files are text files that are completely under your control.

Other than the `rc.local` file, you will rarely change the files discussed in this chapter. But when you do need to fix or debug something, it is good to know where and when things happen in the boot process. Knowledge is a good thing, even if you only use it to ensure that your support contractors know what they are talking about.

An important piece of knowledge gained from this chapter is the understanding of how startup really works. Underneath all of the different tools provided by all of the different Linux distributions there is a boot process that has many similarities. Knowing where the files are stored that start and configure critical network services is very valuable information for any network administrator, particularly when things go wrong. In the next chapter, "The Network Interface," we look even deeper into the process that configures the server's network interface.