

3

HOW LINUX BOOTS

You now know the physical structure of a Linux system, what the kernel is, and how to work with processes. This chapter teaches you how the system starts (*boots*) — that is, how the kernel gets into memory and how the regular system processes get started.

As it turns out, there isn't much to the boot process:

1. A boot loader finds the kernel image on the disk, loads it into memory, and starts it.
2. The kernel initializes the devices and its drivers.
3. The kernel mounts the root filesystem.
4. The kernel starts a program called `init`.
5. `init` sets the rest of the processes in motion.
6. The last processes that `init` starts as part of the boot sequence allow you to log in.

Identifying each stage of the boot process is invaluable in fixing boot problems and understanding the system as a whole. To start, zero in on the boot loader, which is the initial screen or prompt you get after the computer does its power-on self-test, asking which operating system to run. After you make a choice, the boot loader runs the Linux kernel, handing control of the system to the kernel.

There is a detailed discussion of the kernel elsewhere in this book (Section 2.2 explains the role of the kernel, and Chapter 10 tells you how to build one yourself), but this chapter covers the kernel initialization stage, the stage when the kernel prints a bunch of messages about the hardware present on the system. The kernel starts `init` just after it displays a message proclaiming that the kernel has mounted the root filesystem:

```
VFS: Mounted root (ext2 filesystem) readonly.
```

Soon after, you will see a message about `init` starting, followed by system service startup messages, and finally you get a login prompt of some sort.

NOTE *On Red Hat Linux, the `init` note is especially obvious, because it “welcomes” you to “Red Hat Linux.” All messages thereafter show success or failure in brackets at the right-hand side of the screen.*

Most of this chapter deals with `init`, because it is the part of the boot sequence where you have the most control. Section 3.2 deals with the boot loaders.

3.1 `init`

There is nothing special about `init`. It is a program just like any other on the Linux system, and you’ll find it in `/sbin` along with other system binaries. The main purpose of `init` is to start and stop other programs in a particular sequence. All you have to know is how this sequence works.

There are a few different variations, but most Linux distributions use the System V style discussed here. Some distributions use a simpler version that resembles the BSD `init`, but you are unlikely to encounter this.

3.1.1 Runlevels

At any given time on a Linux system, a certain base set of processes are running. This state of the machine is called its *runlevel*, and it is denoted with a number from 0 through 6. The system spends most of its time in a single runlevel. However, when you shut the machine down, `init` switches to a different runlevel in order to terminate the system services in an orderly fashion and to tell the kernel to stop. Yet another runlevel is for *single-user mode*, discussed in Section 3.2.4.

The easiest way to get a handle on runlevels is to examine the `init` configuration file, `/etc/inittab`. Look for a line like the following:

```
id:5:initdefault:
```

This line means that the default runlevel on the system is 5. All lines in the `inittab` file take this form, with four fields separated by colons occurring in the following order:

- A unique identifier (a short string, such as `id` in the preceding example)
- The applicable runlevel number(s)
- The action that `init` should take (in the preceding example, the action is to set the default runlevel to 5)
- A command to execute (optional)

There is no command to execute in the preceding `initdefault` example because a command doesn't make sense in the context of setting the default runlevel. Look a little further down in `inittab`, until you see a line like this:

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

This particular line is important because it triggers most of the system configuration and services through the `rc*.d` and `init.d` directories (see Section 3.1.2). You can see that `init` is set to execute a command called `/etc/rc.d/rc 5` when in runlevel 5. The `wait` action tells when and how `init` runs the command: run `rc 5` once when entering runlevel 5, and then wait for this command to finish before doing anything else.

There are several different actions in addition to `initdefault` and `wait`, especially pertaining to power management, and the `inittab(5)` manual page tells you all about them. The ones that you're most likely to encounter are explained in the following sections.

respawn

The `respawn` action causes `init` to run the command that follows, and if the command finishes executing, to run it again. You're likely to see something similar to this line in your `inittab` file:

```
1:2345:respawn:/sbin/mingetty tty1
```

The `getty` programs provide login prompts. The preceding line is for the first virtual console (`/dev/tty1`), the one you see when you press `ALT-F1` or `CONTROL-ALT-F1` (see Section 3.3). The `respawn` action brings the login prompt back after you log out.

ctrlaltdel

The `ctrlaltdel` action controls what the system does when you press `CONTROL-ALT-DELETE` on a virtual console. On most systems, this is some sort of reboot command, using the `shutdown` command, which is covered in Section 3.1.5.

sysinit

The `sysinit` action is the very first thing that `init` should run when it starts up, before entering any runlevels.

3.1.2 How Processes in Runlevels Start

You are now ready to learn how `init` starts the system services, just before it lets you log in. Recall this `inittab` line from earlier:

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

This small line triggers many other programs. `rc` stands for *run commands*, and you will hear people refer to the commands as scripts, programs, or services. So, where are these commands, anyway?

For runlevel 5, in this example, the commands are probably either in `/etc/rc.d/rc5.d` or `/etc/rc5.d`. Runlevel 1 uses `rc1.d`, runlevel 2 uses `rc2.d`, and so on. You might find the following items in the `rc5.d` directory:

| | | |
|----------------|----------------|--------------|
| S10syslogd | S20ppp | S99gpm |
| S12kerneld | S25netstd_nfs | S99httpd |
| S15netstd_init | S30netstd_misc | S99rnmologin |
| S18netbase | S45pcmcia | S99sshd |
| S20acct | S89atd | |
| S20logoutd | S89cron | |

The `rc 5` command starts programs in this runlevel directory by running the following commands:

```
S10syslogd start
S12kerneld start
S15netstd_init start
S18netbase start
...
...
S99sshd start
```

Notice the `start` argument in each command. The `S` in a command name means that the command should run in start mode, and the number (00 through 99) determines where in the sequence `rc` starts the command.

The `rc*.d` commands are usually shell scripts that start programs in `/sbin` or `/usr/sbin`. Normally, you can figure out what one of the commands actually does by looking at the script with `less` or another pager program.

You can start one of these services by hand. For example, if you want to start the `httpd` Web server program manually, run `S99httpd start`. Similarly, if you ever need to kill one of the services when the machine is on, you can run the command in the `rc*.d` directory with the `stop` argument (`S99httpd stop`, for instance).

Some `rc*.d` directories contain commands that start with `K` (for “kill,” or stop mode). In this case, `rc` runs the command with the `stop` argument instead of `start`. You are most likely to encounter `K` commands in runlevels that shut the system down.

3.1.3 Adding and Removing Services

If you want to add, delete, or modify services in the `rc*.d` directories, you need to take a closer look at the files inside. A long listing reveals a structure like this:

```
lrwxrwxrwx . . . S10sysklogd -> ../init.d/sysklogd
lrwxrwxrwx . . . S12kerneld -> ../init.d/kerneld
lrwxrwxrwx . . . S15netstd_init -> ../init.d/netstd_init
lrwxrwxrwx . . . S18netbase -> ../init.d/netbase
...
```

The commands in an `rc*.d` directory are actually symbolic links to files in an `init.d` directory, usually in `/etc` or `/etc/rc.d`. Linux distributions contain these links so that they can use the same startup scripts for all runlevels. This convention is by no means a requirement, but it often makes organization a little easier.

To prevent one of the commands in the `init.d` directory from running in a particular runlevel, you might think of removing the symbolic link in the appropriate `rc*.d` directory. This does work, but if you make a mistake and ever need to put the link back in place, you might have trouble remembering the exact name of the link. Therefore, you shouldn't remove links in the `rc*.d` directories, but rather, add an underscore (`_`) to the beginning of the link name like this:

```
mv S99httpd _S99httpd
```

At boot time, `rc` ignores `_S99httpd` because it doesn't start with `S` or `K`. Furthermore, the original name is still obvious, and you have quick access to the command if you're in a pinch and need to start it by hand.

To add a service, you must create a script like the others in the `init.d` directory and then make a symbolic link in the correct `rc*.d` directory. You may need to read Chapter 7 first if you aren't familiar with shell scripts. However, the easiest way to write a script is to examine the scripts already in `init.d`, make a copy of one that you understand, and modify the copy.

When adding a service, make sure that you choose an appropriate place in the boot sequence to start the service. If the service starts too soon, it may not work, due to a dependency on some other service. For non-essential services, most systems administrators prefer numbers in the 90s, after most of the services that came with the system.

Linux distributions usually come with a command to enable and disable services in the `rc*.d` directories. For example, in Debian, the command is `update-rc.d`, and in Red Hat Linux, the command is `chkconfig`. Graphical user interfaces are also available. Using these programs helps keep the startup directories consistent and helps with upgrades.

HINT *One of the most common Linux installation problems is an improperly configured XFree86 server that flicks on and off, making the system unusable on console. To stop this behavior, boot into single-user mode and alter your runlevel or runlevel services. Look for something containing `xdm`, `gdm` or `kdm` in your `rc*.d` directories, or your `/etc/inittab`.*

3.1.4 Controlling `init`

Occasionally, you need to give `init` a little kick to tell it to switch runlevels, to re-read the `inittab` file, or just to shut down the system. Because `init` is always the first process on a system, its process ID is always 1.

You can control `init` with `telinit`. For example, if you want to switch to runlevel 3, use this command:

```
telinit 3
```

When switching runlevels, `init` tries to kill off any processes that aren't in the `inittab` file for the new runlevel. Therefore, you should be careful about changing runlevels.

When you need to add or remove respawning jobs or make any other change to the `inittab` file, you must tell `init` about the change and cause it to re-read the file. Some people use `kill -HUP 1` to tell `init` to do this. This traditional method works on most versions of Unix, as long as you type it correctly. However, you can also run this `telinit` command:

```
telinit q
```

You can also use `telinit s` to switch to single-user mode (see Section 3.2.4).

3.1.5 Shutting Down

`init` also controls how the system shuts down and reboots. The proper way to shut down a Linux machine is to use the `shutdown` command.

There are two basic ways to use `shutdown`. If you *halt* the system, it shuts the machine down and keeps it down. To make the machine halt immediately, use this command:

```
shutdown -h now
```

On most modern machines with reasonably recent versions of Linux, a `halt` cuts the power to the machine. You can also *reboot* the machine. For a reboot, use `-r` instead of `-h`.

The shutdown process takes several seconds. You should never reset or power off a machine during this stage. In the preceding example, now is the time to shut down. This argument is mandatory, but there are many ways of specifying it. If you want the machine to go down sometime in the future, one way is to use `+n`, where `n` is the number of minutes shutdown should wait before doing its work. For other options, look at the `shutdown(8)` manual page.

To make the system reboot in ten minutes, run this command:

```
shutdown -r +10
```

On Linux, `shutdown` notifies anyone logged on that the machine is going down, but it does little real work. If you specify a time other than `now`, `shutdown` creates a file called `/etc/nologin`. When this file is present, the system prohibits logins by anyone except the superuser.

When system shutdown time finally arrives, `shutdown` tells `init` to switch to runlevel 0 for a halt and runlevel 6 for a reboot. When `init` enters runlevel 0 or 6, all of the following takes place, which you can verify by looking at the scripts inside `rc0.d` and `rc6.d`:

1. `init` kills every process that it can (as it would when switching to any other runlevel).
2. The initial `rc0.d/rc6.d` commands run, locking system files into place and making other preparations for shutdown.
3. The next `rc0.d/rc6.d` commands unmount all filesystems other than the root.
4. Further `rc0.d/rc6.d` commands remount the root filesystem read-only.
5. Still more `rc0.d/rc6.d` commands write all buffered data out to the filesystem with the `sync` program (see Section 2.4.4).
6. The final `rc0.d/rc6.d` commands tell the kernel to reboot or stop with the `reboot`, `halt`, or `poweroff` program.

The `reboot` and `halt` programs behave differently for each runlevel, potentially causing confusion. By default, these programs call `shutdown` with the `-r` or `-h` options. but if the system is already at the halt or reboot runlevel, the programs tell the kernel to shut itself off immediately. If you really want to shut your machine down in a hurry (disregarding any possible damage from a disorderly shutdown), use the `-f` option.

3.2 Boot Loaders

Before the kernel runs `init`, a boot loader starts the kernel. On occasion, you need to tell the boot loader to load different kernels or operating systems, and to start in different modes. This is especially important when trying to fix a system that has a problem that prevents a full boot. To fix such a problem, you may need single-user mode or an alternate kernel.

The boot loader loads a kernel image into memory and hands control of the CPU to the new image, possibly supplying it with some parameters. These parameters are simple text strings like `-s` for booting in single-user mode and `root=partition` for using *partition* as the root filesystem instead of the default. You can specify a runlevel number as a parameter to make the system boot into a runlevel other than the default.

To type in a kernel name and parameters, however, you first need to know how to get to a boot prompt. Unfortunately, there are several different boot loaders out there, and because you can control a boot loader's behavior, Linux distributions customize to their hearts' content.

The next sections tell you how to get to a boot prompt in order to enter a kernel name and parameters. If you need to know how to install a boot loader or change its configuration, see Section 10.5.

3.2.1 LILO

LILO (Linux Loader) has been around for almost as long as the Linux kernel. The LILO boot prompt usually ends with `boot:.` If your system boots with LILO, it's likely that you get a fancy screen of graphics at boot time, because this is the default for many distributions, including Red Hat Linux. If you see a screen like this, look for a part that reads "Press Control-x for text mode." If you see that message, type `CONTROL-X` to get to the boot prompt.

If the system defaults to text mode, look at the prompt as soon as it appears. If the prompt says `LILO` with nothing else, press the `SHIFT` key to get the rest of the boot prompt to appear. On the other hand, if you get a boot prompt immediately, you need to watch out, because the system likely will boot if you don't type anything in a certain amount of time.

Once you're at a LILO boot prompt, press the `TAB` key to show a list of kernel and operating system options. The default kernel name is probably something like `linux`. To boot this kernel with no options, enter `linux`. To use options, specify them after the kernel name:

```
linux option1 option2 ...
```

For example, to boot the system in single-user mode, type this:

```
linux -s
```

Or, to boot `linux` in single-user mode with the root filesystem as `/dev/hda3` instead of your normal default, type this:

```
linux root=/dev/hda3 -s
```

3.2.2 GRUB

GRUB stands for Grand Unified Bootloader, a system that is slowly replacing LILO. GRUB has plenty of whiz-bang features, but most important is its ability to navigate filesystems, so you can read files without loading a kernel.

Wiseguy Solaris and BSD administrators like to say that they have enjoyed this capability for some time.

GRUB has a menu interface that's easy enough to navigate, but if you need to boot from a different kernel, change the root partition, or supply extra kernel parameters, you should get into the mini-shell. Press `c` at the menu to get this prompt:

```
grub>
```

Let's say that you want to boot the kernel at `/boot/vmlinuz` with a root of `/dev/hda3`. Furthermore, your system is messed up, so you need single-user mode with the `-s` kernel option. Type the following at the GRUB prompt:

```
root (hd0,2)
kernel /boot/vmlinuz root=/dev/hda3 -s
boot
```

The `root (hd0,2)` line sets GRUB's idea of the current root partition — that is, the filesystem where GRUB expects to find the kernel. `hd0` is the first hard drive (that is, the first disk that GRUB finds; for example, the Linux device `/dev/hda` if this is your first hard disk). However, `2` specifies the *third* partition (`/dev/hda3`) because GRUB partition numbers start at 0.

The word `kernel` sets a kernel image and its parameters. `/boot/vmlinuz` refers to a kernel image file on `(hd0,2)`. Unfortunately, GRUB does not normally pass the information from the preceding `root()` line on to the kernel, so you should always specify `root=partition` as a kernel parameter.

NOTE *You can combine the `root` and `kernel` lines by preceding the kernel image with the GRUB root partition. Therefore, the preceding two lines could be written as the single line `kernel (hd0,2)/boot/vmlinuz root=/dev/hda3 -s`.*

The last line, `boot`, tells GRUB to load and execute the kernel image.

NOTE *On certain systems (especially those with SCSI disks and stock kernels), you may need an initial RAM disk:*

```
initrd /boot/initrd
```

See Section 10.5.4 for information on why you may need an initial RAM disk.

In case you're trying to boot a partition with another boot loader (such as a Windows partition) by hand with GRUB, try the following, where *partition* uses the GRUB device syntax explained earlier (e.g., `hd(0,1)`):

```
rootnoverify partition
makeactive
chainloader +1
boot
```

3.2.3 Other Boot Loaders

There are many other ways to boot a kernel, including from DOS via LOADLIN or SYSLINUX, over the network, or even directly from the PC BIOS with LinuxBIOS! Most other boot loaders work like LILO, although some do not support the TAB key to list options. You may need to pay attention to boot diagnostics for this information. However, the way you enter parameters is usually identical to LILO and GRUB.

3.2.4 Single-User Mode and Emergency Booting

When something goes wrong with the system, an administrator's first recourse for booting quickly to a usable state is *single-user mode*. The idea is that the system quickly boots to a root shell instead of going through the whole mess of services. On Linux, single-user mode is usually runlevel 1. You may need to type the root password to enter single-user mode.

Common tasks in single-user mode include the following:

- Checking filesystems after a system crash
- Fixing problems in critical files, such as `/etc/fstab`, `/etc/passwd`, and `/etc/inittab`
- Restoring from backups after a system crash

Don't expect too many amenities in single-user mode. You may need to set the terminal type (enter `TERM=linux`) to get full-screen editors to work, and the network may not be available. You can configure the network and other systems by hand if necessary, but it's a pain.

When you finish with single-user mode, you can exit the shell to see if the system starts normally. However, it's usually a good idea to reboot the system, because the transition from single-user mode to regular multi-user mode is not always perfect.

If you have a real mess on your hands, and even single-user mode doesn't work, you can try the `-b` kernel parameter for an emergency boot shell instead of any kind of orderly startup. This does not mount the root filesystem as read-write or give you much of anything else, so you'll probably need to do some remounting, and possibly mount the `/proc` filesystem before getting anything useful done. However, if things are really this broken, you might consider using a rescue CD-ROM instead of this rudimentary emergency boot shell mode. You may also be able to get an emergency boot shell by using the `init=/bin/sh` kernel parameter.

Finally, if you break your kernel or boot loader, you won't be able to get to single-user mode without extra help. You can often boot your system with a kernel from a bootable CD-ROM by passing the root parameter to the CD-ROM's kernel boot loader. Your system might look somewhat strange without your regular kernel, but you should still be able to move files around or perhaps even compile a new kernel to get yourself out of the jam.

3.3 Virtual Consoles

The final stage of the boot process starts one or more programs that allow you to log in to the system console. Linux has two primary display modes: console (text) mode, and an X Window System server (graphics mode, usually via a display manager). The kernel boots in console mode, but on most distributions the system switches over to graphics mode near when the `rc*.d` commands complete.

Linux has several *virtual consoles*. Each virtual console may run in graphics or text mode. When in text mode, you can switch between consoles with an ALT-Function key combination — for example, ALT-F1 takes you to `/dev/tty1`, ALT-F2 goes to `/dev/tty2`, and so on.

A virtual console used by XFree86 in graphics mode is slightly different. Rather than getting a virtual console assignment directly from `/etc/inittab`, an XFree86 server takes over a free virtual console. For example, if you have `getty` processes running on `tty1` and `tty2`, a new XFree86 server takes over `tty3`. In addition, after XFree86 puts a virtual console into graphics mode, you must normally press a CONTROL-ALT-Function key combination to switch to another virtual console instead of the simpler ALT-Function key combination.

The upshot of all of this is that if you want to see your text console after your system boots, press CONTROL-ALT-F1. To get back to the X11 session, press ALT-F2, ALT-F3, and so on, until you get to the X session.

