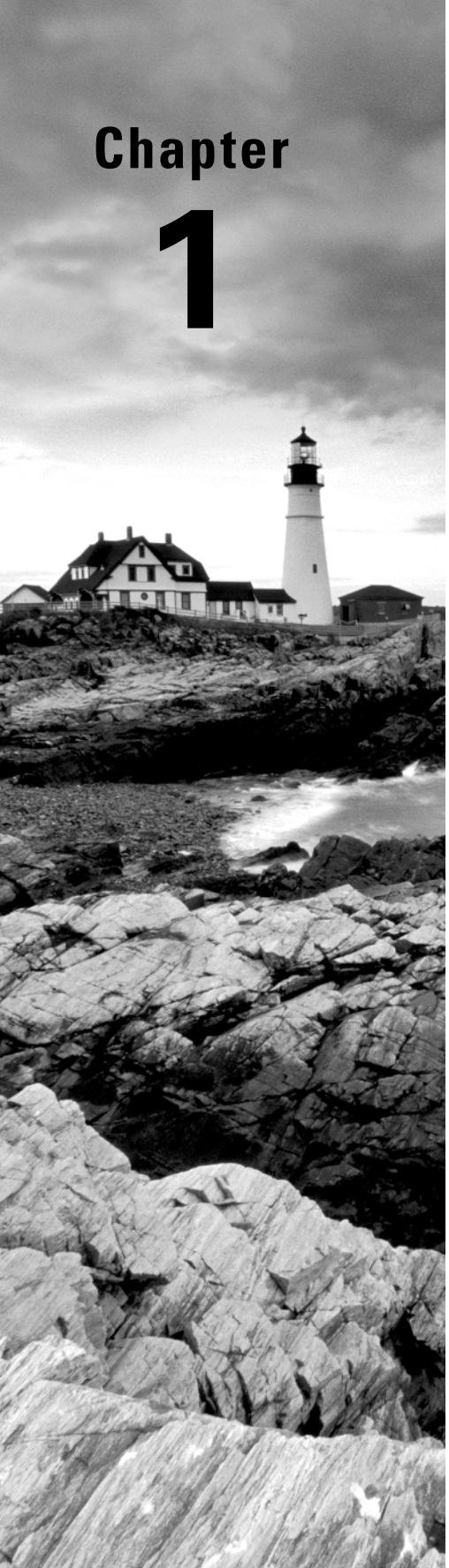
The LPI 201 Exam

PART I



Chapter 1

Starting a System

THE FOLLOWING EXAM OBJECTIVES ARE COVERED IN THIS CHAPTER:

- ✓ 202.1: Customizing SysV-init system startup
- ✓ 202.2: System Recovery
- ✓ 202.3: Alternate Bootloaders

Before you can log in and start using your Linux system, a complicated process of booting the operating system must take place. A lot happens behind the scenes in the Linux boot process. It helps to know all of what is going on just in case something goes wrong.

This chapter examines the boot and startup processes in Linux systems. First, it looks at the role the computer firmware plays in getting the process started. After that, it discusses Linux bootloaders and examines how to configure them. Next, the chapter discusses the Linux initialization process, showing how Linux decides which background applications to start at bootup. The chapter ends by looking at some system recovery options that you have available to help salvage a system that won't boot.

The Linux Boot Process

When you turn on the power to your Linux system, it triggers a series of events that eventually leads to the login prompt. Normally, you don't worry about what happens behind the scenes of those events; you just log in and start using your applications.

However, there may be times when your Linux system doesn't boot quite correctly, or perhaps an application that you expected to be running in background mode isn't running. In those cases, it helps to have a basic understanding of how Linux boots the operating system and starts programs so that you can troubleshoot the problem.

This section walks through the steps of the boot process and shows how you can watch it to see which steps failed.

Following the Boot Process

The Linux boot process can be split into three main steps:

1. The workstation firmware starts, performing a quick check of the hardware, called a *Power-On Self Test (POST)*, and then it looks for a bootloader program to run from a bootable device.
2. The bootloader runs and determines what Linux kernel program to load.
3. The kernel program loads into memory and starts the necessary background programs required for the system to operate (such as a graphical desktop manager for desktops or web and database servers for servers).

While these three steps may seem simple on the surface, a somewhat complicated ballet of operations happens behind the scenes to keep the boot process working. Each step performs several actions as it prepares your system to run Linux.

Viewing the Boot Process

You can monitor the Linux boot process by watching the system console screen as the system boots. You'll see lots of informative messages scroll by as the system detects hardware and loads the software.

Some graphical desktop Linux distributions hide the boot messages on a separate console window when they start up. Often, you can hit either the Esc key or the Ctrl+Alt+F1 key combination to view those messages.

Usually the boot messages scroll by somewhat quickly, and it's hard to see just what's happening. If you need to troubleshoot boot problems, you can review the boot-time messages using the `dmesg` command. Most Linux distributions copy the boot kernel messages into a special ring buffer in memory called the *kernel ring buffer*. The buffer is circular and set to a predetermined size. As new messages are logged into the buffer, older messages are rotated out.

The `dmesg` command displays the most recent boot messages that are currently stored in the kernel ring buffer, as shown in Listing 1.1.

Listing 1.1: The dmesg command output

```
$ dmesg
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.13.0-37-generic (buildd@kapok) (gcc version 4.8.2
 (Ubuntu 4.8.2-19ubuntu1) ) #64-Ubuntu SMP Mon Sep 22 21:28:38 UTC 2014 (Ubuntu
3.13.0-37.64-generic 3.13.11.7)
[    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-3.13.0-37-generic root=UUI
D=09007318-c158-4f3e-b519-e90bc538fb3a ro quiet splash vt.handoff=7
[    0.000000] KERNEL supported cpus:
[    0.000000]   Intel GenuineIntel
[    0.000000]   AMD AuthenticAMD
[    0.000000]   Centaur CentaurHauls
[    0.000000] e820: BIOS-provided physical RAM map:
[    0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable
[    0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff] reserved
```

```
[    0.000000] BIOS-e820: [mem 0x0000000000100000-0x000000006edeffff] usable
[    0.000000] BIOS-e820: [mem 0x000000006edf0000-0x000000006edfffff] ACPI data
[    0.000000] BIOS-e820: [mem 0x00000000fffc0000-0x00000000ffffffff] reserved
[    0.000000] NX (Execute Disable) protection: active
[    0.000000] SMBIOS 2.5 present.
[    0.000000] DMI: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox
12/01/2006
```

Most Linux distributions also store the boot messages in a log file, usually in the `/var/log` folder. For Debian-based systems, the file is usually `/var/log/boot`, while for Red Hat–based systems, the file is `/var/log/boot.log`.

Although it helps to be able to see the different messages generated during boot time, it is also helpful to know just what generates those messages. This chapter discusses each of these three boot steps, and it goes through some examples showing how they work.

The Firmware Startup

All IBM-compatible workstations and servers utilize some type of built-in firmware to control how the installed operating system starts. On older workstations and servers, this firmware was called the *Basic Input/Output System (BIOS)*. On newer workstations and servers, a new method, called the *Unified Extensible Firmware Interface (UEFI)*, is responsible for maintaining the system hardware status and launching an installed operating system.

Both methods eventually launch the main operating system program, however, and each method uses different ways of doing that. This section walks you through the basics of the BIOS and UEFI methods, showing you how they participate in the Linux boot process.

The BIOS Startup

The BIOS firmware found in older workstations and servers was somewhat limited in what it could do. The BIOS firmware had a simplistic menu interface that allowed you to change some settings to control how the system found hardware and define what device the BIOS should use to start the operating system.

One of the limitations of the original BIOS firmware was that it could read only one sector's worth of data from a hard drive into memory in order to run. As you can probably guess, that's not enough space to load an entire operating system. To get around that limitation, most operating systems (including Linux and Microsoft Windows) split the boot process into two parts.

First, the BIOS runs a bootloader program. The *bootloader* is a small program that initializes the necessary hardware to find and run the full operating system, which is usually found at another location on the same hard drive but sometimes situated on a separate internal or external storage device.

The bootloader program usually has a configuration file, so you can tell it where to find the actual operating system file to run or even to produce a small menu allowing the user to boot between multiple operating systems.

To get things started, the BIOS must know where to find the bootloader program on an installed storage device. Most BIOS setups allow you to load the bootloader program from several locations:

- An internal hard drive
- An external hard drive
- A CD/DVD drive
- A USB memory stick
- A network server

When booting from a hard drive, you must designate which hard drive, and which partition on the hard drive, the BIOS should load the bootloader program from. This is done by defining a *Master Boot Record (MBR)*.

The MBR is the first sector on the first hard drive partition on the system. There is only one MBR for the computer system. The BIOS looks for the MBR and reads the program stored there into memory. Since the bootloader program must fit in one sector, it must be very small—so it can't do much. The bootloader program mainly points to the location of the actual operating system kernel file, which is stored in a boot sector of a separate partition on the system. There are no size limitations on the kernel boot file.

The bootloader program isn't required to point directly to an operating system kernel file—it can point to any type of program, including another bootloader program. You can create a primary bootloader program that points to a secondary bootloader program, which provides options to load multiple operating systems. This process is called *chainloading*.

The UEFI Startup

While there were plenty of limitations with BIOS, computer manufacturers learned to live with them, and BIOS was the default standard for IBM-compatible systems for many years. However, as operating systems became more complicated, it eventually became clear that a new boot method needed to be developed.

Intel created the *Extensible Firmware Interface (EFI)* in 1998 to address some of the limitations of BIOS. The adoption of EFI was somewhat of a slow process, but by 2005 the idea caught on with other vendors, and the *Universal EFI (UEFI)* specification was adopted as a standard. These days, just about all IBM-compatible desktop and server systems utilize the UEFI firmware standard.

Instead of relying on a single boot sector on a hard drive to hold the bootloader program, UEFI specifies a special disk partition, called the *EFI System Partition (ESP)* to store bootloader programs. This allows for any size of bootloader program, plus the ability to store multiple bootloader programs for multiple operating systems.

The ESP setup utilizes the old Microsoft File Allocation Table (FAT) filesystem to store the bootloader programs. On Linux systems, the ESP is typically mounted in the `/boot/efi` folder, and the bootloader files are typically stored using the `.efi` filename extension.

The UEFI firmware utilizes a built-in mini bootloader (sometimes referred to as a *boot manager*), which allows you to configure just which bootloader program file to launch.

Not all Linux distributions support the UEFI firmware. If you're using a UEFI system, make sure that the Linux distribution you select supports it.

With UEFI, you need to register each individual bootloader file that you want to appear at boot time in the boot manager interface menu. The `efibootmgr` Linux application allows you to create and remove boot entries or change the boot order. The UEFI interface includes a shell environment, allowing you to enter commands to alter boot settings, or select the bootloader to run each time you boot the system.

Once the firmware finds and runs the bootloader, its job is done. The bootloader step in the boot process can be somewhat complicated. The next section dives into covering this step.

Another popular new technology commonly found on UEFI systems is the use of solid state drives (SSDs) to replace legacy hard drives. SSD hardware doesn't use legacy hard drive controllers, but instead, utilizes the *Non-Volatile Memory Express (NVMe)* standard. The Linux kernel has supported NVMe systems since kernel version 3.3, and the UEFI boot process works just fine when using SSD hardware in Linux systems.

Linux Bootloaders

The bootloader program helps bridge the gap between the system firmware and the full Linux operating system kernel. In Linux, you have several choices of bootloaders. The most popular ones that you'll run across are these:

- Linux Loader (LILO)
- Grand Unified Bootloader (GRUB) Legacy
- GRUB2

In the original versions of Linux, the *Linux Loader (LILO)* bootloader was the only one available. It was extremely limited in what it could do, but it accomplished its purpose, that is, loading the Linux kernel from the BIOS startup. LILO became the default bootloader used by Linux distributions in the 1990s. The LILO configuration file is stored in a single file, `/etc/lilo.conf`, which defines the systems to boot. Unfortunately, LILO doesn't work with UEFI systems, so it has limited use on modern systems and is quickly fading into history.

The first version of the GRUB bootloader (now called *GRUB Legacy*) was created in 1999 to provide a more robust and configurable bootloader to replace LILO. GRUB quickly became the default bootloader for all Linux distributions, whether they were run on BIOS or UEFI systems.

GRUB2 was created in 2005 as a total rewrite of the GRUB Legacy system. It supports advanced features, such as the ability to load hardware driver modules and use logic statements to alter the boot menu options dynamically, depending on conditions detected on the system (such as if an external hard drive is connected).

Since UEFI can load any size of bootloader program, it's now possible to load a Linux operating system kernel directly without a special bootloader program. This feature was incorporated in the Linux kernel starting with version 3.3.0. However, this method isn't common, because bootloader programs can provide more versatility in booting, especially when working with multiple operating systems.

This section walks through the basics of both the GRUB Legacy and GRUB2 bootloaders, which should cover just about every Linux distribution that you'll run into these days.

GRUB Legacy

The GRUB Legacy bootloader was designed to simplify the process of creating boot menus and passing options to kernels. GRUB Legacy allows you to select multiple kernels and/or operating systems using a menu interface as well as an interactive shell. You configure the menu interface to provide options for each kernel or operating system you wish to boot. The interactive shell provides a way for you to customize boot commands on the fly.

Both the menu and the interactive shell utilize a set of commands that control features of the bootloader. This section walks you through how to configure the GRUB Legacy bootloader, how to install it, and how to interact with it at boot time.

Configuring GRUB Legacy

When you use the GRUB Legacy interactive menu, you need to tell it what options to show. You do that using special GRUB *menu commands*.

The GRUB Legacy system stores the menu commands in a standard text configuration file. The configuration file used by GRUB Legacy is `menu.lst`, and it is stored in the `/boot/grub` folder. (While not a requirement, some Linux distributions create a separate `/boot` partition on the hard drive.) Red Hat–derived Linux distributions (such as CentOS and Fedora) use `grub.conf` instead of `menu.lst` for the configuration file.

The GRUB Legacy configuration file consists of two sections:

- Global definitions
- Operating system boot definitions

The global definitions section defines commands that control the overall operation of the GRUB Legacy boot menu. The global definitions must appear first in the configuration file. There are only a handful of global settings. Table 1.1 shows these settings.

TABLE 1.1 GRUB Legacy global commands

Setting	Description
`color`	Specifies the foreground and background colors to use in the boot menu
`default`	Defines the default menu option to select
`fallback`	A secondary menu selection to use if the default menu option fails
`hiddenmenu`	Don't display the menu selection options
`splashimage`	Points to an image file to use as the background for the boot menu
`timeout`	Specifies the amount of time to wait for a menu selection before using the default

For GRUB Legacy, to define a value for a command you just list the value as a command-line parameter:

```
default 0
timeout 10
color white/blue yellow/blue
```

The `color` command defines the color scheme for the menu. The first pair of values defines the foreground/background colors for normal menu entries, while the second pair defines the foreground/background colors for the selected menu entry.

After the global definitions, you place definitions for the individual operating systems that are installed on the system. Each operating system should have its own definition section. You can use many boot definition settings to customize how the bootloader finds the operating system kernel file. Fortunately, just a few commands are required to define the operating system. The ones to remember are as follows:

- `title`—The first line for each boot definition section, this is what appears in the boot menu.
- `root`—Defines the disk and partition where the GRUB `/boot` folder partition is located on the system.
- `kernel`—Defines the kernel image file stored in the `/boot` folder to load.
- `initrd`—Defines the initial RAM disk file, which contains drivers necessary for the kernel to interact with the system hardware.
- `rootnoverify`—Defines non-Linux boot partitions, such as Windows.

The `root` command defines the hard drive and partition that contain the `/boot` folder for GRUB Legacy. Unfortunately, GRUB Legacy uses a somewhat odd way of referencing those values:

```
(hddrive, partition)
```

Also unfortunately, GRUB Legacy doesn't refer to hard drives the way Linux does: it uses a numbering system to reference both disks and partitions, starting at 0 instead of at 1. For example, to reference the first partition on the first hard drive of the system, you'd use `(hd0,0)`. To reference the second partition on the first hard drive, you'd use `(hd0,1)`.

The `initrd` command is another important feature in GRUB Legacy. It helps solve a problem that arises when using specialized hardware or filesystems as the root drive. The `initrd` command defines a file that's mounted by the kernel at boot time as a RAM disk. The kernel can then load modules from the RAM disk, which then allows it to access hardware or filesystems not compiled into the kernel itself. (Chapter 3, "Mastering the Kernel," discusses how to create the `initrd` image, as well as the `initramfs` image, used by Debian systems).

Listing 1.2 shows a sample GRUB Legacy configuration file that defines both a Windows partition and a Linux partition for booting.

Listing 1.2: Sample GRUB Legacy configuration file

```
default 0
timeout 10
color white/blue yellow/blue

title CentOS Linux
root (hd1,0)
kernel (hd1,0)/boot/vmlinuz
initrd /boot/initrd

title Windows
rootnoverify (hd0,0)
```

This example shows two boot options: one for a CentOS Linux system and one for a Windows system. The CentOS system is installed on the first partition of the second hard drive, while the Windows system is installed on the first partition of the first hard drive. The Linux boot selection specifies the kernel file to load, as well as the `initrd` image file to load into memory.

Installing GRUB Legacy

Once you build the GRUB Legacy configuration file, you must install the GRUB Legacy program in the MBR. The command to do this is `grub-install`.

The `grub-install` command uses a single parameter that indicates the partition on which to install GRUB. You can specify the partition using either the Linux or GRUB Legacy format. For example, to use the Linux format, you'd write

```
# grub-install /dev/sda
```

to install GRUB on the MBR of the first hard drive. To use the GRUB Legacy format, you must enclose the hard drive format in quotes:

```
# grub-install '(hd0)'
```

If you're using the chainloading method and prefer to install a copy of GRUB Legacy on the boot sector of a partition instead of to the MBR of a hard drive, you must specify the partition, again using either the Linux or GRUB format:

```
# grub-install /dev/sda1
# grub-install 'hd(0,0)'
```

You don't need to reinstall GRUB Legacy in the MBR after making changes to the configuration file, because GRUB Legacy reads the configuration file each time it runs.

Interacting with GRUB Legacy

When you boot a system that uses the GRUB Legacy bootloader, you'll see a menu that shows the boot options that you defined in the configuration file. If you wait for the timeout to expire, the default boot option will process. Alternatively, you can use the arrow keys to select one of the boot options and then press the Enter key to select it.

You can also edit boot options on the fly from the GRUB menu. First, arrow to the boot option that you want to modify, and then press the `E` key. Use the arrow key to move the cursor to the line that you need to modify, and then press the `E` key to edit it. Press the `B` key to boot the system using the new values. You can also press the `C` key at any time to enter an interactive shell mode, allowing you to submit commands on the fly.

GRUB 2

Since the GRUB2 system was intended as an improvement over GRUB Legacy, many of the features are the same, with just a few twists. For example, the GRUB2 system changes the configuration filename to `grub.cfg`, and it stores it in the `/boot/grub/` folder. (This allows you to have both GRUB Legacy and GRUB2 installed at the same time.)

Configuring GRUB2

There are also a few changes to the commands used in GRUB2. For example, instead of the `title` command, GRUB uses the `menuentry` command, and you must also enclose each individual boot section within braces immediately following the `menuentry` command. Here's an example of a GRUB2 configuration file:

```
menuentry "CentOS Linux" {
   set root=(hd1,1)
   linux /boot/vmlinuz
   initrd /initrd
}
menuentry "Windows" {
   set root=(hd0,1)
}
```

Notice that GRUB2 uses the `set` command to assign values to the `root` keyword, and it uses an equal sign to assign the device. GRUB2 utilizes environment variables to configure settings instead of commands.

To make things more confusing, GRUB2 changes the numbering system for partitions. While it still uses `0` for the first hard drive, the first partition is set to `1`. So to define the `/boot` folder on the first partition of the first hard drive, you now need to use

```
set root=hd(0,1)
```

Also, notice that the `rootnoverify` and `kernel` commands are not used in GRUB2. Non-Linux boot options are now defined the same as Linux boot options using the `root` environment variable, and you define the kernel location using the `linux` command.

The configuration process for GRUB2 is also somewhat different. While GRUB2 uses the `/boot/grub/grub.cfg` file as the configuration file, you should never modify that file. Instead, there are separate configuration files stored in the `/etc/grub.d` folder. This allows you (or the system) to create individual configuration files for each boot option installed on your system (for example, one configuration file for booting Linux and another for booting Windows).

For global commands, use the `/etc/default/grub` configuration file. The format for some of the global commands has changed from the GRUB Legacy commands so it is `GRUB_TIMEOUT` instead of just `timeout`.

Most Linux distributions generate the new `grub.cfg` configuration file automatically after certain events, such as when upgrading the kernel. Usually, the distribution will keep a boot option pointing to the old kernel file just in case the new one fails.

Installing GRUB2

Unlike GRUB Legacy, you don't need to install GRUB2. All you need to do is to rebuild the main installation file by running the `grub-mkconfig` program.

The `grub-mkconfig` program reads configuration files stored in the `/etc/grub.d` folder and assembles the commands into the single `grub.cfg` configuration file.

You can update the configuration file manually by running the `grub-mkconfig` command:

```
# grub-mkconfig > /boot/grub/grub.cfg
```

Notice that you must either redirect the output of the `grub-mkconfig` program to the `grub.cfg` configuration file or use the `-o` option to specify the output file. By default, the `grub-mkconfig` program just outputs the new configuration file commands to standard output.

Interacting with GRUB2

The GRUB2 bootloader produces a boot menu similar to the GRUB Legacy method. You can use arrow keys to switch between boot options, the `E` key to edit a boot entry, or the `C` key to bring up the GRUB2 command line to submit interactive boot commands. Figure 1.1 illustrates the editing of an entry in the GRUB2 boot menu on an Ubuntu system.

FIGURE 1.1 Editing an Ubuntu GRUB2 menu entry

Some graphical desktops (such as Ubuntu) hide the GRUB boot menu behind a graphical interface. Usually, if you hold down the Shift key when the system first boots, this will display the GRUB boot menu.

Alternative Bootloaders

While GRUB Legacy and GRUB2 are the most popular Linux bootloader programs, you may run into a few others, depending on which Linux distributions you are using.

The *Systemd-boot* bootloader program is starting to gain popularity in Linux distributions that use the systemd init method (see the next section). The systemd-boot bootloader generates a menu of boot image options, and can load any EFI boot image.

The *U-Boot* bootloader program can boot from any type of disk, and load any type of boot image.

The *Syslinux project* includes five separate bootloader programs that have special uses in Linux:

- SYSLINUX—A bootloader for systems that use the Microsoft FAT filesystem (popular for booting from USB memory sticks)
- EXTLINUX—A mini bootloader for booting from an ext2, ext3, ext4, or btrfs filesystem
- ISOLINUX—A bootloader for booting from a LiveCD or LiveDVD
- PXELINUX—A bootloader for booting from a network server
- MEMDISK—A utility to boot older DOS operating systems from the other Syslinux bootloaders

The ISOLINUX bootloader is popular for use in distributions that release a LiveDVD version. The bootloader requires two files: `isolinux.bin`, which contains the bootloader program image, and `isolinux.cfg`, which contains the configuration settings.

Newer systems also allow booting an iso image from a USB flash drive. To do this you'll need an additional file, `isodhpfx.bin`, which is a hybrid bootloader program image that can be loaded onto a hard drive or USB flash drive. This file is generated by the `xorriso` program.

The PXELINUX bootloader is somewhat complicated. It uses the *Pre-boot Execution Environment (PXE)* standard, which defines how a network workstation can boot and load an operating system from a central network server. PXE uses DHCP to assign a network address to the workstation and BOOTP to load the bootloader image from the server. The network server must support the Trivial File Transfer Protocol (TFTP) to transfer the boot image file to the workstation.

To utilize PXELINUX, the TFTP server needs to have the PXELINUX bootloader program stored as `/tftpboot/pxelinux.0` available for the workstations to download. Each workstation must also have a configuration file available in the `/tftpboot/pxelinux.cfg` directory. The files are named based on the MAC address of the workstation, and they contain specific configuration settings required for that workstation.

Secure Bootloaders

There's one other feature new to the UEFI boot method that can cause heartburn for Linux administrators. UEFI supports a feature called *secure boot*. In secure boot, the UEFI boot manager only loads bootloader images that are digitally signed to ensure their safety. This is a great feature to prevent a virus or malware program from taking over your system, but does add an additional layer of complexity to the Linux boot process.

There are generally three ways to run Linux in a secure boot environment:

- Disable secure booting in the UEFI boot manager
- Purchase your own digital signature key to sign your bootloader images
- Use a bootloader image signed by someone else

The first method is most often the easiest, as long as your system UEFI boot manager allows you to disable the secure boot feature. However, that can leave your Linux system vulnerable to attack, and not all systems allow you to disable secure boot.

Purchasing your own digital signature key can be expensive, and somewhat cumbersome if you change bootloader images frequently. Each time you change the bootloader image you need to re-sign the file, which means getting an external signing agent involved.

There is, however, a relatively simple solution. A few Linux organizations produce signed mini-bootloader images for public use. The mini-bootloader acts as a middleman in the boot process. The UEFI boot manager boots the mini-bootloader, then, in turn, it boots the standard Linux bootloader image.

Currently the two most popular mini-bootloader image methods are from the Linux Foundation (called preloader), and Fedora (called shim). The shim mini-bootloader file is named `shim.efi`, and is stored in the uefi folder on the system. When it boots, it automatically looks for a GRUB 2 bootloader image file named `grubx64.efi`, also in the uefi folder on the system. That way you can still change the GRUB 2 bootloader image without having to worry about the digitally signed `shim.efi` file.

Process Initialization

A Linux system comprises many programs running in background to provide services for the system. The `init` program starts all of those programs when the Linux system starts up. This is called the *initialization process*.

When the kernel finishes loading, it looks for the `init` program in one of three locations:

- `/sbin/init`
- `/etc/init`
- `/bin/init`

If none of these files exist, the kernel attempts to start a generic shell session using the `/bin/sh` program. If that fails as well, the kernel enters *panic mode* and stops processing.

The main job of the `init` program is to start other programs. The programs that start are based on the features that you want running in your Linux system. For example, a Linux server doesn't necessarily need to start a graphical desktop environment, or a Linux desktop doesn't necessarily need to start the Apache web server service.

Currently three popular initialization process methods are used in Linux distributions:

- Unix System V (also called SysV)
- systemd
- Upstart

The original Linux `init` program was based on the Unix System V `init` program, and it became commonly called *SysV* (or sometimes *SysV-init*). The SysV `init` program uses a series of shell scripts divided into separate *runlevels* in order to determine which programs run at what times. A runlevel groups common applications that must all start or stop together into a common group. Each program uses a separate shell script to start and stop the individual program, but the system can run all of the scripts at the same time.

The system administrator sets the runlevel in which the Linux system starts. This in turn determines which set of programs is running. The system administrator can also change the runlevel at any time while the system is running.

The SysV `init` program had served the Linux community well for many years, but as Linux systems became more complicated and required more services, the runlevel shell scripts became more complicated. This caused Linux developers to look for other solutions.

The *systemd* program was developed by the Red Hat Linux group to handle starting and stopping programs in dynamic Linux environments. Instead of runlevels, it uses targets and units to control what applications run at any time on the system. It uses separate configuration files that determine this behavior.

The *Upstart* version of the `init` program was developed as part of the Ubuntu Linux distribution. Its main goal was to handle the dynamic environment that hot-pluggable devices cause in Linux. The Upstart method uses separate configuration files for each service, and each service configuration file sets in which runlevel the service should start. That way, you have just one service file that's used for multiple runlevels.

The following sections take a closer look at each of these initialization process methods to help you get comfortable in any Linux environment.

The SysV Method

The key to the SysV initialization process is runlevels. The `init` program determines which programs to start based on the runlevel of the system.

Runlevels are numbered from 0 to 6, and each one is assigned a set of programs that should be running for that runlevel. When the Linux kernel starts, it determines which runlevel to start by a configuration file. It's important to know how to manage runlevels and how to determine when each runlevel is used by the kernel. The following sections show you how to do that.

Runlevels

While each Linux distribution defines applications that should be running at specific runlevels, there are some general guidelines that you can use. Table 1.2 shows the general usage of the Linux runlevels.

TABLE 1.2 Linux runlevels

Runlevel	Description
0	Shut down the system.
1	Single-user mode, used for system maintenance.
2	On Debian-based systems, multiuser graphical mode.
3	On Red Hat–based systems, multiuser text mode.
4	Undefined.
5	On Red Hat–based systems, multiuser graphical mode.
6	Reboot the system.

Most Linux distributions use the Red Hat runlevel method of using runlevel 3 for multiuser text mode and runlevel 5 for multiuser graphical mode. However, Debian-based systems use runlevel 2 for all multiuser modes.

Starting Applications in a Runlevel

There are two ways to start applications in runlevels:

- Using the `/etc/inittab` file
- Using startup scripts

The `/etc/inittab` file defines what applications start at which runlevel. Each line in the `/etc/inittab` file defines an application and uses the following format:

```
id:runlevels:action:process
```

The `id` field contains one to four characters that uniquely define the process. The `runlevels` field contains a list of runlevels in which the application should be running. The list is not comma separated, so the value 345 indicates that the application should be started in runlevels 3, 4, and 5.

The `action` field contains a keyword that tells the kernel what to do with the application for that runlevel. Possible values are shown in Table 1.3.

TABLE 1.3 The SysV inittab action values

Action	Description
`boot`	The process is started at boot time.
`bootwait`	The process is started at boot time, and the system will wait for it to finish.
`initdefault`	Specifies the runlevel to enter after the system boots.
`kbrequest`	The process is started after a special key combination is pressed.
`once`	The process is started once when the runlevel is entered.
`powerfail`	The process is started when the system is powered down.
`powerwait`	The process is started when the system is powered down, and the system will wait for it to finish.
`respawn`	The process is started when the runlevel is entered and restarted whenever it terminates.
`sysinit`	The process is started at boot time before any `boot` or `bootwait` items.
`wait`	The process is started once, and the system will wait for it to finish.

The initdefault line specifies the runlevel in which the system normally runs after boot:

```
id:3:initdefault:
```

Besides the runlevels, the SysV init method also specifies startup scripts to control how applications start and stop. The /etc/init.d/rc or /etc/rc.d/rc script runs all scripts with a specified runlevel. The scripts themselves are stored in the /etc/init.d/rcx.d or /etc/rcx.d folder, where *x* is the runlevel number.

Scripts are stored with a specific filename that indicates whether they start or stop at the runlevel. Scripts that start with an S start the programs, and scripts that start with a K stop the programs. The script filenames also contain a number, which indicates the order in which the rc program runs the scripts. This allows you to specify which scripts get started before others in order to control any dependency issues.

Modifying Program Runlevels

Working through all of the script files stored to start and stop individual programs can be somewhat of a hassle. To make life easier, Linux distributions include a couple of utilities to assign a runlevel easily for any program that you need to start or stop:

- chkconfig
- update-rc.d

The chkconfig command is used in most Red Hat–based Linux distributions. It's a very versatile command that allows you to list at what runlevels the application starts and also to change the runlevels in which a specific application starts.

When used with the --list parameter, the chkconfig command displays all of the applications defined, along with the runlevels in which they start. Alternatively, you can list a specific application to see how it is configured:

```
$ chkconfig --list network
network    0:off  1:off  2:on  3:on  4:on  5:on  6:off
$
```

In this example, the network application is configured to start in runlevels 2, 3, 4, and 5. You can then use the --levels parameter to modify the runlevels:

```
# chkconfig --levels 12345 network on
```

This sets the network program to start on runlevels 1, 2, 3, 4, and 5.

Table 1.4 shows the different formats that you can use with the chkconfig command.

TABLE 1.4 The chkconfig formats

Format	Description
chkconfig *program*	Check if the program is set to start at the current runlevel.

TABLE 1.4 The chkconfig formats *(continued)*

Format	Description
`chkconfig` *program* `on`	Start the program at the default runlevel.
`chkconfig` *program* `off`	Don't start the program at the default runlevel.
`chkconfig --add` *program*	Add the program to start at boot.
`chkconfig --del` *program*	Remove the program from starting at boot.
`chkconfig --levels [`*levels*`]` *program* `on`	Set the program to start at the specified runlevels.
`chkconfig --list` *program*	Display the current runlevel settings for the program.

For Debian-based Linux distributions, you'll need to use the `update-rc.d` command to control application runlevels. To start a program at the default runlevel, just use the following format:

```
update-rc.d program defaults
```

To remove the program from starting at the default runlevel, use the following format:

```
update-rc.d program remove
```

If you want to specify what runlevels the program starts and stops at, you'll need to use the following format:

```
update-rc.d -f program start 40 2 3 4 5 . stop 80 0 1 6 .
```

The 40 and 80 specify the relative order within the runlevel when the program should start or stop (from 0 to 99). This allows you to customize exactly when specific programs are started or stopped during the boot sequence.

Checking the System Runlevel

You've seen that the `/etc/inittab` file indicates the default runlevel with the `initdefault` action, but there's no guarantee that's the runlevel at which your Linux system is currently running. The `runlevel` command displays both the current runlevel and the previous runlevel for the system:

```
$ runlevel
N 2
$
```

The first character is the previous runlevel. The N character means the system is in the original boot runlevel. The second character is the current runlevel.

Changing Runlevels

You can change the current runlevel of your Linux system using either the init or telinit command. Just specify the runlevel number as the command-line parameter. For example, to reboot your system you can enter this command:

```
# init 6
```

The downside to using the init command is that it immediately changes the system to the specified runlevel. That may not be an issue if you're the only person on your Linux system, but in a multiuser Linux environment, this can have adverse effects for the other users.

A kinder way to change the runlevel on multiuser systems is to use one of a handful of special commands designed for that purpose:

- shutdown—Gracefully changes the runlevel to 1, or single-user mode
- halt—Gracefully changes the runlevel to 0 to stop the system
- poweroff—Gracefully changes the runlevel to 0 to stop the system
- reboot—Gracefully changes the runlevel to 6 to restart the system

Each of these commands also allows you to specify a message to send to any other users on the system before it changes the runlevel. You can also specify a time for the change, such as +15 for 15 minutes.

The systemd Method

The systemd initialization process method is quickly gaining in popularity in the Linux world. It's currently the default initialization process used in the Fedora, CentOS, and Red Hat Linux distributions.

The systemd initialization process introduced a major paradigm shift in how Linux systems handle services. This has also caused some controversy in the Linux world. Instead of lots of small initialization shell scripts, the systemd method employs one monolithic program that uses individual configuration files for each service. This is somewhat of a departure from the earlier Linux philosophy.

This section walks you through the basics of how the systemd initialization process works.

Units and Targets

Instead of using shell scripts and runlevels, the systemd method uses units and targets. A *unit* defines a service or action on the system. It consists of a name, a type, and a configuration file. There are currently eight different types of systemd units:

- automount
- device

- mount
- path
- service
- snapshot
- socket
- target

The systemd program identifies units by their name and type using the format name.type. You use the systemctl command to list the units currently loaded in your Linux system:

```
# systemctl list-units
UNIT                        LOAD   ACTIVE SUB       DESCRIPTION
...
crond.service               loaded active running   Command Scheduler
cups.service                loaded active running   CUPS Printing Service
dbus.service                loaded active running   D-Bus System Message
...
multi-user.target           loaded active active    Multi-User System
network.target              loaded active active    Network
paths.target                loaded active active    Paths
remote-fs.target            loaded active active    Remote File Systems
slices.target               loaded active active    Slices
sockets.target              loaded active active    Sockets
...
#
```

Linux distributions can have hundreds of different units loaded and active. We just selected a few from the output to show you what they look like. The systemd method uses service type units to manage the daemons on the Linux system. The target type units are important in that they group multiple units together so that they can be started at the same time. For example, the network.target unit groups all of the units required to start the network interfaces for the system.

The systemd initialization process uses targets similar to the way SysV uses runlevels. A *target* represents a different group of services that should be running on the system. Instead of changing runlevels to alter what's running on the system, you just change targets.

To make the transition from SysV to systemd smoother, there are targets that mimic the standard 0 through 6 SysV runlevels, called runlevel0.target through runlevel6.target.

Configuring Units

Each unit requires a configuration file that defines what program it starts and how it should start the program. The systemd system stores unit configuration files in the /lib/

systemd/system folder. Here's an example of the sshd.service unit configuration file used in CentOS:

```
# cat sshd.service
[Unit]
Description=OpenSSH server daemon
After=syslog.target network.target auditd.service

[Service]
EnvironmentFile=/etc/sysconfig/sshd
ExecStartPre=/usr/sbin/sshd-keygen
ExecStart=/usr/sbin/sshd -D $OPTIONS
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
RestartSec=42s

[Install]
WantedBy=multi-user.target
#
```

The sshd.service configuration file defines the program to start (/usr/sbin/sshd), along with some other features, such as what services should run before the sshd service starts (the After line), what target level the system should be in (the WantedBy line), and how to reload the program (the Restart line).

Target units also use configuration files. They don't define programs, but instead they define which service units to start. Here's an example of the graphical.target unit configuration file used in CentOS:

```
# cat graphical.target
#  This file is part of systemd.
#
# systemd is free software; you can redistribute it and/or modify it
# under the terms of the GNU Lesser General Public License as published by
#  the Free Software Foundation; either version 2.1 of the License, or
#  (at your option) any later version.
[Unit]
Description=Graphical Interface
Documentation=man:systemd.special(7)
Requires=multi-user.target
After=multi-user.target
Conflicts=rescue.target
Wants=display-manager.service
AllowIsolate=yes
```

```
[Install]
Alias=default.target
#
```

The target configuration defines what targets should be loaded first (the `After` line), what targets are required for this target to start (the `Requires` line), what targets conflict with this target (the `Conflicts` line), and what targets or services the target requires to be running (the `Wants` line).

Setting the Default Target

The default target used when the Linux system boots is defined in the `/etc/systemd/system` folder as the file `default.target`. This is the file the `systemd` program looks for when it starts up. This file is normally set as a link to a standard target file in the `/lib/systemd/system` folder:

```
# ls -al default.target
lrwxrwxrwx. 1 root root 36 Oct  1 09:14 default.target ->
 /lib/systemd/system/graphical.target
#
```

On this CentOS system, the default target is set to the `graphical.target` unit.

The systemctl Program

In the systemd method, you use the `systemctl` program to control services and targets. The `systemctl` program uses options to define what action to take, as shown in Table 1.5.

TABLE 1.5 The systemctl commands

Command name	Explanation
`list-units`	Displays the current status of all configured units
`default`	Changes to the default target unit
`isolate`	Starts the named unit and stops all others
`start name`	Starts the named unit
`stop name`	Stops the named unit
`reload name`	Causes the named unit to reload its configuration file
`restart name`	Causes the named unit to shut down and restart
`status name`	Displays the status of the named unit (You can pass a PID value rather than a name, if you like.)

Command name	Explanation
`enable name`	Configures the unit to start when the computer next boots
`disable name`	Configures the unit not to start when the computer next boots

Instead of using shell scripts to start and stop services, you use the `start` and `stop` commands:

```
# systemctl stop sshd.service
# systemctl status sshd.service
sshd.service - OpenSSH server daemon
Loaded: loaded (/usr/lib/systemd/system/sshd.service; disabled)
Active: inactive (dead)

Oct 04 10:33:33 localhost.localdomain systemd[1]: Stopped OpenSSH server
 daemon.

# systemctl start sshd.service
# systemctl status sshd.service
sshd.service - OpenSSH server daemon
Loaded: loaded (/usr/lib/systemd/system/sshd.service; disabled)
Active: active (running) since Thu 2014-10-04 10:34:08 EDT; 4s ago
Process: 3882
ExecStartPre=/usr/sbin/sshd-keygen (code=exited, status=0/SUCCESS)
Main PID: 3889 (sshd)
CGroup: /system.slice/sshd.service
            3889 /usr/sbin/sshd -D

Oct 04 10:34:08 localhost.localdomain sshd-keygen[3882]: Generating SSH2 RSA
host key: [  OK  ]
Oct 04 10:34:08 localhost.localdomain systemd[1]: Started OpenSSH server daemon.
Oct 04 10:34:08 localhost.localdomain sshd[3889]: Server listening on
 0.0.0.0 port 22.
Oct 04 10:34:08 localhost.localdomain sshd[3889]: Server listening on ::
 port 22.
#
```

To change the target that is currently running, you must use the `isolate` command. For example, to enter single-user mode, you'd use the following:

```
# systemctl isolate rescue.target
```

To go back to the default target for the system, you just use the default command.

There's also a `systemd-delta` command that allows you to identify when multiple configuration files exist and overwrite each other.

One of the more controversial features of the systemd initialization process is that it doesn't use the standard Linux syslogd log file system. Instead, it has its own log files, and those log files are not stored in text format. To view the systemd log files, you need to use the `journalctl` program.

The Upstart Method

The Ubuntu Linux distribution created the Upstart initialization process as a replacement for the SysV initialization process. The goal was to create an initialization process that could better handle hot-pluggable devices that are commonly found on systems today.

The Upstart method replaces the `/etc/inittab` file and all of the `/etc/init.d` startup scripts with a single `/etc/init` folder. The `/etc/init` folder contains configuration files for each system service and program that needs to start, usually in the format `name.conf`, where `name` is the program name.

An example of an Upstart configuration file looks like this:

```
# tty1 - getty
#
# This service maintains a getty on tty1 from the point the system is
# started until it is shut down again.

start on stopped rc RUNLEVEL=[2345] and (
            not-container or
            container CONTAINER=lxc or
            container CONTAINER=lxc-libvirt)

stop on runlevel [!2345]

respawn
exec /sbin/getty -8 38400 tty1
$
```

The `tty1.conf` configuration file specifies when the tty1 console port should be enabled. Notice that the configuration file specifies the runlevels when it should start (2, 3, 4, and 5).

One of the great features of Upstart is that it can trigger a script to run not only at a specific runlevel but also when a device is connected to the system.

To change the runlevel or event that starts or stops a program, you just modify the program's configuration file in the `/etc/init` folder.

To stop a program or service using Upstart, you just use the `stop` command-line command, along with the program or service name:

```
$ sudo stop bluetooth
bluetooth stop/waiting
$
```

Likewise, to start up a program or service, you use the `start` command:

```
$ sudo start bluetooth
bluetooth start/running, process 2635
$
```

The Upstart initialization method provides a simpler way of managing programs and services running on the Linux system.

The *Linux Standard Base (LSB)* is a specification supported by some Linux distributions in order to attempt to create a standard experience between Linux systems. Part of the LSB defines how systems use commands in init scripts. Standard init commands such as `start`, `stop`, and `restart` are defined by the LSB, and they should be supported by all Linux distributions that follow the LSB. Unfortunately, some major Linux distributions (such as Debian) have recently dropped out of the LSB group, making it seem less likely that there will be a standard Linux initialization method anytime soon.

System Recovery

There's nothing worse than starting up your Linux system and not getting a login prompt. Plenty of things can go wrong in the Linux startup process, but most of the issues come down to two categories:

- Kernel failures
- Drive failures

Fortunately, there are ways to help recover your Linux system from many of these errors. The following sections walk you through some standard troubleshooting practices that you can follow to attempt to recover a Linux system that fails to boot.

Kernel Failures

Kernel failures occur when the Linux kernel stops running in memory, causing the Linux system to crash. They are often a result of a software change, such as installing a new kernel without the appropriate module or library changes or starting (or stopping) a program at a new runlevel. Often these types of boot errors can be fixed by starting the system using an alternative method and editing the necessary files to change the system. This section describes the techniques that you can use to alter the system setup after a failed boot.

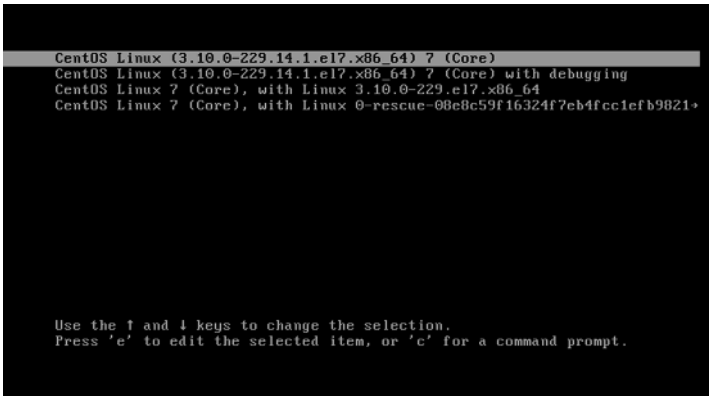
Selecting Previous Kernels at Boot

One of the biggest culprits for a failed boot is upgrading the Linux kernel, either on your own or from a packaged distribution upgrade. When you install a new kernel file, it's always a good idea to leave the old kernel file in place and create an additional entry in the GRUB boot menu to point to the new kernel.

By creating multiple kernel entries in the GRUB boot menu, you can select which kernel version to boot. If the new kernel fails to boot properly, you can reboot and select the older kernel version.

Most Linux distributions do this automatically when adding a new kernel, keeping the most recent older kernel available in the boot menu, as shown in Figure 1.2.

FIGURE 1.2 The CentOS Grub boot menu with multiple kernel options



Single-User Mode

At times, you may need to perform some type of system maintenance, such as adding a new hardware module or library file to get the system to boot properly. In these situations, you will want the system to boot up without allowing multiple users to connect, especially in a server environment. This is called *single-user mode*.

The GRUB menu allows you to start the system in single-user mode by adding the `single` command to the `linux` line in the boot menu commands. To get there, press the `e` key on the boot option in the GRUB boot menu.

When you add the `single` command, the system will boot into runlevel 1, which creates a single login for the root user account. Once you log in as the root user account, you can modify the appropriate modules, `init` scripts, or GRUB boot menu options necessary to get your system started correctly.

Passing Kernel Parameters

Besides the single-user mode trick, you can add other kernel parameters to the `linux` command in the GRUB boot menu. The kernel accepts lots of parameters that alter the

hardware modules that it activates or the hardware settings that it looks for with specific devices. (This is especially true for sound and network cards.) You can specify the different hardware settings as additional parameters to the kernel in the `linux` command and then boot from that entry in the GRUB menu.

Root Drive Failure

Perhaps the worst feeling a Linux system administrator can experience is seeing that the bootloader can't read the root drive device. This type of error may not be fatal, however, because it is sometimes possible to recover from a corrupt root drive. This section walks through the steps that you can take to attempt to recover a corrupt root drive and possibly save your Linux system.

Using a Rescue Disk

Many Linux distributions provide what's called a *rescue disk* to be used when fatal disk errors occur. The rescue disk usually boots either from the CD/DVD drive or as a USB stick, and it loads a small Linux system into memory. Since the Linux system runs entirely in memory, it can leave all of the workstation hard drives free for examination and repair. From the system command-line prompt, you can perform some diagnostic and repair tasks on your system hard drives.

The tool of choice for checking and fixing hard drive errors is the `fsck` command. The `fsck` command isn't a program. Rather, it is an alias for a family of commands that are specific to different types of filesystems (such as ext2, ext3, and ext4). You need to run the `fsck` command against the device name of the partition that contains the root directory of your Linux system. For example, if the root directory is on the `/dev/sda1` partition, you'd run this command:

```
# fsck /dev/sda1
```

The `fsck` command will examine the inode table, along with the file blocks stored on the hard drive, and attempt to reconcile them. If any errors occur, you will be prompted whether to repair them or not. If there are many errors on the partition, you can add the `-y` parameter to answer yes automatically to all of the repair questions. After a successful repair, it's a good idea to run the `fsck` command one more time to ensure that all of the errors have been found and corrected. Continue running the `fsck` command until you get a clean run with no errors.

Mounting a Root Drive

When the `fsck` repair is complete, you can test the repaired partition by mounting it into the virtual directory created in memory. Just use the `mount` command to mount it to an available mount directory:

```
# mount /dev/sda1 /media
```

You can examine the filesystem stored in the partition to ensure that it's not corrupt. Before rebooting, you should unmount the partition using the `umount` command:

```
# umount /dev/sda1
```

After successfully unmounting the partition, you can reboot your Linux system using the standard bootloader and attempt to boot using the standard kernel and runlevels.

EXERCISE 1.1

Using Rescue Mode

This exercise demonstrates how to start your Linux distribution in single-user mode to examine filesystems and configurations without performing a complete bootup. To use single-user mode, follow these steps:

1. First, start your Linux distribution as normal, and log in at the standard login prompt (either the graphical desktop or the command-line login) as your normal user account.
2. Type **`runlevel`** to determine the default runlevel for your system. The first character returned refers to the previous runlevel (`N` denotes no previous runlevel since the system booted). The second character is the current runlevel. This is likely to be `2` on Debian-based systems, `3` on command-line Red Hat–based systems, or `5` on graphical desktop Red Hat–based systems.
3. Now reboot your system, and hit an arrow key when the GRUB menu appears to stop the countdown timer. If you're using a Linux distribution that hides the GRUB menu (such as Ubuntu), hold down the Shift key when the system boots to display the GRUB menu.
4. At the GRUB menu, use the arrow keys to go to the default menu entry (usually the first entry in the list), then press the `e` key. This takes GRUB into edit mode.
5. Look for either the `linux` or `linux16` menu command lines. These define the kernel used to start the session.
6. Go to the end of the `linux` or `linux16` line, and add the word **`single`**. Press **`Ctrl-x`** to save the change temporarily and start your system using that menu entry.
7. The Linux system will boot into single-user mode. Depending on your Linux distribution, it may prompt you to enter the root user account or to press Ctrl+D to continue on with the normal boot. Enter the root user account password to enter single-user mode.
8. Now you are at the root user command prompt. Enter the command **`runlevel`** to view the current runlevel. It should show runlevel 1. From here you can modify configuration files, check filesystems, and change user accounts.
9. Reboot the system by typing **`reboot`**.

10. You should return to the standard boot process and GRUB menu options as before. Select the standard GRUB menu option to boot your system and then log in.
11. At a command line prompt, type **`runlevel`** to ensure that you are back to the normal default runlevel for your Linux system.

Summary

Although Linux distributions are designed to boot without any user intervention, it helps to know the Linux boot process in case anything does go wrong. Most Linux systems use either the GRUB Legacy or GRUB2 bootloader program. These programs both reside in the BIOS Master Boot Record or in the ESP partition on UEFI systems. The bootloader loads the Linux kernel program, which then runs the `init` program to start individual background programs required for the Linux system.

There are several different types of initialization scripts used by Linux distributions. The SysV method is the oldest, derived from the original Unix system. The systemd method is a more modern replacement, and it is becoming more popular in many Linux distributions. The Ubuntu Linux distribution has also created the Upstart method, which is an advanced initialization method to the SysV method, allowing more dynamic control over which programs start based on what hardware devices are detected at boot time.

Finally, no discussion on Linux startup is complete without examining system recovery methods. If your Linux system fails to boot, the most likely cause is either a kernel issue or a root device issue. For kernel issues, you can often modify the GRUB menu to add additional kernel parameters or even to boot from an older version of the kernel. For root drive issues, you can try to boot from a rescue mode into a version of Linux running in memory and then use the `fsck` command to repair a damaged root drive.

Exam Essentials

Describe the Linux boot process. The BIOS or UEFI starts a bootloader program from the Master Boot Record, which is usually the Linux GRUB Legacy or GRUB2 program. The bootloader program loads the Linux kernel into memory, which in turn looks for the `init` program to run. The `init` program starts individual application programs and starts either the command-line terminals or the graphical desktop manager.

Describe the Linux GRUB Legacy and GRUB2 bootloaders. The GRUB Legacy bootloader stores files in the `/boot/grub` folder, and it uses the `menu.lst` or `grub.conf` configuration file to define commands used at boot time. The commands can create a boot menu, allowing you to select between multiple boot locations, options, or features. You must use the `grub-install` program to install the GRUB Legacy bootloader program into the

Master Boot Record. The GRUB2 bootloader also stores files in the /boot/grub folder, but it uses the grub.cfg configuration file to define the menu commands. You don't edit the grub.cfg file directly, but instead you store files in the /etc/default/grub file or individual configuration files in the /etc/grub.d folder. Run the update-grub program to generate the grub.cfg file from the configuration files.

Describe alternative Linux bootloaders. The LILO bootloader is used on older Linux systems. It uses the /etc/lilo.conf configuration file to define the boot options. The Syslinux project has created the most popular alternative Linux bootloaders. The SYSLINUX bootloader runs on FAT filesystems, such as USB memory sticks. The ISOLINUX bootloader is popular on LiveCD distributions, because it can boot from a CD or DVD. It stores the bootloader program in the isolinux.bin file and configuration settings in the isolinux.cfg file. The PXELINUX bootloader program allows a network workstation to boot from a network server. The server must contain the pxelinux.0 image file along with the pxelinux.cfg directory, which contains separate configuration files for each workstation. EXTLINUX is a small bootloader program that can be used on smaller embedded Linux systems.

Describe Linux runlevels and the init structure. The init program starts applications based on a default runlevel. Runlevels from 0 to 6 are defined, with each runlevel containing scripts that start and stop specific programs. The /etc/inittab configuration file defines the default runlevel for the Linux system, along with which programs start at what runlevel. Init scripts are stored in the /etc/init.d and /etc/rc.d folders and run as needed.

Explain the SysV initialization system. The SysV init system uses a default runlevel specified by the line id:2:initdefault: in the /etc/inittab file. You use either the chkconfig, update-rc.d, or systemctl command to change which services are started when switching to specific runlevels. Runlevels 0, 1, and 6 are reserved for shutdown, single-user mode, and rebooting, respectively. Runlevels 3, 4, and 5 are the common user runlevels on Red Hat and most other distributions, and runlevel 2 is the normal user runlevel on Debian systems.

Describe how to change SysV init runlevels. The programs init and telinit can be used to change to other runlevels. shutdown, halt, poweroff, and reboot are also useful when shutting down, rebooting, or switching to single-user mode.

Explain the systemd initialization system. The systemd system uses units and targets to control services. The default target is specified by the file /etc/systemd/system/default.target, and it is a link to a target file in the /lib/systemd/system folder.

Describe how to change systemd init targets. You use the systemctl program to start and stop services, as well as to change the target level of the system.

Describe how to recover from a failed boot. The GRUB bootloaders provide you with options that can help if your Linux system fails to boot. You can press the e key at the GRUB boot menu to edit any boot menu entry and then add any additional kernel parameters, such as placing the system in single-user mode. You can also use a rescue disk to boot Linux into memory, then use the fsck command to repair any corrupt hard drives, and then use the mount command to mount them to examine the files.

Review Questions

You can find the answers in the Appendix.

1. What program does the workstation firmware start at boot time?
   - **A.** A bootloader
   - **B.** The `init` program
   - **C.** The Windows OS
   - **D.** The `mount` command
   - **E.** The `telinit` program
2. Where does the firmware first look for a Linux bootloader program?
   - **A.** The `/boot/grub` folder
   - **B.** The Master Boot Record (MBR)
   - **C.** The `/var/log` folder
   - **D.** A boot partition
   - **E.** The `/etc` folder
3. What Linux command lets you examine the most recent boot messages?
   - **A.** `fsck`
   - **B.** `init`
   - **C.** `mount`
   - **D.** `dmesg`
   - **E.** `chkconfig`
4. What folder do most Linux distributions use to store boot logs?
   - **A.** `/etc`
   - **B.** `/var/messages`
   - **C.** `/var/log`
   - **D.** `/boot`
   - **E.** `/proc`
5. Where does the workstation BIOS attempt to find a bootloader program? (Select all that apply.)
   - **A.** An internal hard drive
   - **B.** An external hard drive
   - **C.** A DVD drive
   - **D.** A USB memory stick
   - **E.** A network server

6. Where is the Master Boot Record located?
   - **A.** The first sector of the first hard drive on the system
   - **B.** The boot partition of any hard drive on the system
   - **C.** The last sector of the first hard drive on the system
   - **D.** Any sector on any hard drive on the system
   - **E.** The first sector of the second hard drive on the system
7. Where is the EFI System Partition (ESP) stored on Linux systems?
   - **A.** `/boot`
   - **B.** `/etc`
   - **C.** `/var`
   - **D.** `/boot/efi`
   - **E.** `/boot/grub`
8. What file extension do UEFI bootloader files use?
   - **A.** `.cfg`
   - **B.** `.uefi`
   - **C.** `.lst`
   - **D.** `.conf`
   - **E.** `.efi`
9. Which was the first bootloader program used in Linux?
   - **A.** GRUB Legacy
   - **B.** LILO
   - **C.** GRUB2
   - **D.** SYSLINUX
   - **E.** ISOLINUX
10. Where are the GRUB Legacy configuration files stored?
    - **A.** `/boot/grub`
    - **B.** `/boot/efi`
    - **C.** `/etc`
    - **D.** `/var`
    - **E.** `/proc`
11. Where are GRUB2 configuration files stored? (Select all that apply.)
    - **A.** `/proc`
    - **B.** `/etc/grub.d`
    - **C.** `/boot/grub`
    - **D.** `/boot/efi`
    - **E.** `/var`

12. What command must you run to generate the GRUB2 `grub.cfg` configuration file?
    A. `chkconfig`
    B. `update-rc.d`
    C. `grub-mkconfig`
    D. `grub-install`
    E. `init`

13. What program does the kernel use to start other programs?
    A. GRUB2
    B. `systemctl`
    C. `telinit`
    D. `init`
    E. BIOS

14. Which configuration file contains the SysV default runlevel?
    A. `/etc/init.d`
    B. `/etc/inittab`
    C. `/etc/grub.d`
    D. `/etc/rc.d`
    E. `/boot/grub.cfg`

15. What runlevel is the default for Debian-based systems?
    A. 0
    B. 1
    C. 6
    D. 5
    E. 2

16. What command would you use to change the current runlevel? (Select all that apply.)
    A. `telinit`
    B. `chkconfig`
    C. `update-rc.d`
    D. `init`
    E. `dmesg`

17. What command displays the runlevels in which a program will be started?
    A. `chkconfig`
    B. `init`
    C. `dmesg`
    D. `update-rc.d`
    E. `telinit`

18. What command do Debian systems use to set the runlevels for programs?

A. `chkconfig`

B. `update-rc.d`

C. `init`

D. `telinit`

E. `dmesg`

19. What program allows you to fix corrupt hard drive partitions?

A. `mount`

B. `umount`

C. `fsck`

D. `init`

E. `telinit`

20. Which command allows you to append a partition to the virtual directory on a running Linux system?

A. `mount`

B. `umount`

C. `fsck`

D. `dmesg`

E. `init`