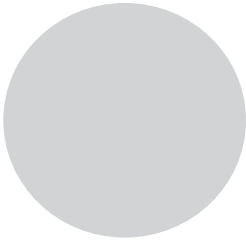


5

HOW THE LINUX KERNEL BOOTS



You now know the physical and logical structure of a Linux system, what the kernel is, and how to work with processes. This chapter will teach you how the kernel starts, or *boots*. In other words, you'll learn how the kernel moves into memory and what it does up to the point where the first user process starts.

A simplified view of the boot process looks like this:

1. The machine's BIOS or boot firmware loads and runs a boot loader.
2. The boot loader finds the kernel image on disk, loads it into memory, and starts it.
3. The kernel initializes the devices and its drivers.
4. The kernel mounts the root filesystem.
5. The kernel starts a program called *init* with a process ID of 1. This point is the *user space start*.

6. `init` sets the rest of the system processes in motion.
7. At some point, `init` starts a process allowing you to log in, usually at the end or near the end of the boot sequence.

This chapter covers the first couple of stages, focusing on the boot loaders and kernel. [Chapter 6](#) continues with the user space start by detailing *systemd*, the most widespread version of `init` on Linux systems.

Being able to identify each stage of the boot process will prove invaluable to you in fixing boot problems and understanding the system as a whole. However, the default behavior in many Linux distributions often makes it difficult, if not impossible, to identify the first few boot stages as they proceed, so you'll probably be able to get a good look only after they've completed and you log in.

5.1 Startup Messages

Traditional Unix systems produce many diagnostic messages upon boot that tell you about the boot process. The messages come first from the kernel and then from processes and initialization procedures that `init` starts. However, these messages aren't pretty or consistent, and in some cases they aren't even very informative. In addition, hardware improvements have caused the kernel to start much faster than before; the messages flash by so quickly, it can be difficult to see what's happening. As a result, most current Linux distributions do their best to hide boot diagnostics with splash screens and other forms of filler to distract you while the system starts.

The best way to view the kernel's boot and runtime diagnostic messages is to retrieve the journal for the kernel with the `journalctl` command. Running `journalctl -k` displays the messages from the current boot, but you can view previous boots with the `-b` option. We'll cover the journal in more detail in [Chapter 7](#).

If you don't have *systemd*, you can check for a logfile such as `/var/log/kern.log` or run the `dmesg` command to view the messages in the *kernel ring buffer*.

Here's a sample of what you can expect to see from the `journalctl -k` command:

```
microcode: microcode updated early to revision 0xd6, date = 2019-10-03
Linux version 4.15.0-112-generic (buildd@lcy01-amd64-027) (gcc version 7.5.0
(Ubuntu 7.5.0-3ubuntu1~18.04)) #113-Ubuntu SMP Thu Jul 9 23:41:39 UTC 2020 (Ubuntu
4.15.0-112.113-generic 4.15.18)
Command line: BOOT_IMAGE=/boot/vmlinuz-4.15.0-112-generic root=UUID=17f12d53-c3d7-4ab3-943e-
a0a72366c9fa ro quiet splash vt.handoff=1
KERNEL supported cpus:
--snip--
scsi 2:0:0:0: Direct-Access    ATA        KINGSTON SM2280S 01.R PQ: 0 ANSI: 5
sd 2:0:0:0: Attached scsi generic sgo type 0
sd 2:0:0:0: [sda] 468862128 512-byte logical blocks: (240 GB/224 GiB)
```

```
sd 2:0:0:0: [sda] Write Protect is off
sd 2:0:0:0: [sda] Mode Sense: 00 3a 00 00
sd 2:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO or FUA
sda: sda1 sda2 < sda5 >
sd 2:0:0:0: [sda] Attached SCSI disk
--snip--
```

After the kernel has started, the user-space startup procedure often generates messages. These messages will likely be more difficult to view and review because on most systems you won't find them in a single logfile. Startup scripts are designed to send messages to the console that are erased after the boot process finishes. However, this isn't a problem on Linux systems because systemd captures diagnostic messages from startup and runtime that would normally go to the console.

5.2 Kernel Initialization and Boot Options

Upon startup, the Linux kernel initializes in this general order:

1. CPU inspection
2. Memory inspection
3. Device bus discovery
4. Device discovery
5. Auxiliary kernel subsystem setup (networking and the like)
6. Root filesystem mount
7. User space start

The first two steps aren't too remarkable, but when the kernel gets to devices, the question of dependencies arises. For example, the disk device drivers may depend on bus support and SCSI subsystem support, as you saw in [Chapter 3](#). Then, later in the initialization process, the kernel must mount a root filesystem before starting init.

In general, you won't have to worry about the dependencies, except that some necessary components may be loadable kernel modules rather than part of the main kernel. Some machines may need to load these kernel modules before the true root filesystem is mounted. We'll cover this problem and its initial RAM filesystem (initrd) workaround solutions in [Section 6.8](#).

The kernel emits certain kinds of messages indicating that it's getting ready to start its first user process:

```
Freeing unused kernel memory: 2408K
Write protecting the kernel read-only data: 20480k
Freeing unused kernel memory: 2008K
Freeing unused kernel memory: 1892K
```

Here, not only is the kernel cleaning up some unused memory, but it's also protecting its own data. Then, if you're running a new enough kernel, you'll see the kernel start the first user-space process as `init`:

```
Run /init as init process
with arguments:
--snip--
```

Later on, you should be able to see the root filesystem being mounted and `systemd` starting up, sending a few messages of its own to the kernel log:

```
EXT4-fs (sda1): mounted filesystem with ordered data mode. Opts: (null)
systemd[1]: systemd 237 running in system mode. (+PAM +AUDIT +SELINUX +IMA
+APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP +GCRYPT +GNUTLS +ACL +XZ +LZ4
+SECCOMP +BLKID +ELFUTILS +KMOD -IDN2 +IDN -PCRE2 default-hierarchy=hybrid)
systemd[1]: Detected architecture x86-64.
systemd[1]: Set hostname to <duplex>.
```

At this point, you definitely know that user space has started.

5.3 Kernel Parameters

When the Linux kernel starts, it receives a set of text-based *kernel parameters* containing a few additional system details. The parameters specify many different types of behavior, such as the amount of diagnostic output the kernel should produce and device driver-specific options.

You can view the parameters passed to your system's currently running kernel by looking at the `/proc/cmdline` file:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-4.15.0-43-generic root=UUID=17f12d53-c3d7-4ab3-943e
-a0a72366c9fa ro quiet splash vt.handoff=1
```

The parameters are either simple one-word flags, such as `ro` and `quiet`, or *key=value* pairs, such as `vt.handoff=1`. Many of the parameters are unimportant, such as the `splash` flag for displaying a splash screen, but one that is critical is the `root` parameter. This is the location of the root filesystem; without it, the kernel cannot properly perform the user space start.

The root filesystem can be specified as a device file, as in this example:

```
root=/dev/sda1
```

On most contemporary systems, there are two alternatives that are more common. First, you might see a logical volume such as this:

```
root=/dev/mapper/my-system-root
```

You may also see a `UUID` (see [Section 4.2.4](#)):

```
root=UUID=17f12d53-c3d7-4ab3-943e-a0a72366c9fa
```

Both of these are preferable because they do not depend on a specific kernel device mapping.

The `ro` parameter instructs the kernel to mount the root filesystem in read-only mode upon user space start. This is normal; read-only mode ensures that `fsck` can check the root filesystem safely before trying to do anything serious. After the check, the bootup process remounts the root filesystem in read-write mode.

Upon encountering a parameter that it doesn't understand, the Linux kernel saves that parameter. The kernel later passes the parameter to `init` when performing the user space start. For example, if you add `-s` to the kernel parameters, the kernel passes the `-s` to the `init` program to indicate that it should start in single-user mode.

If you're interested in the basic boot parameters, the `bootparam(7)` manual page gives an overview. If you're looking for something very specific, you can check out *kernel-params.txt*, a reference file that comes with the Linux kernel.

With these basics covered, you should feel free to skip ahead to [Chapter 6](#) to learn the specifics of user space start, the initial RAM disk, and the `init` program that the kernel runs as its first process. The remainder of this chapter details how the kernel loads into memory and starts, including how it gets its parameters.

5.4 Boot Loaders

At the start of the boot process, before the kernel and `init` start, a *boot loader* program starts the kernel. The boot loader's job sounds simple: it loads the kernel into memory from somewhere on a disk and then starts the kernel with a set of kernel parameters. However, this job is more complicated than it appears. To understand why, consider the questions that the boot loader must answer:

- Where is the kernel?
- What kernel parameters should be passed to the kernel when it starts?

The answers are (typically) that the kernel and its parameters are usually somewhere on the root filesystem. It may sound like the kernel parameters should be easy to find, but remember that the kernel itself is not yet running, and it's the kernel that usually traverses a filesystem to find the necessary files. Worse, the kernel device drivers normally used to access the disk are also unavailable. Think of this as a kind of "chicken or egg" problem. It can get even more complicated than this, but for now, let's see how a boot loader overcomes the obstacles of the drivers and the filesystem.

A boot loader does need a driver to access the disk, but it's not the same one that the kernel uses. On PCs, boot loaders use the traditional *Basic Input/Output System (BIOS)* or the newer *Unified Extensible Firmware Interface (UEFI)* to access disks. (*Extensible Firmware Interface*, or *EFI*, and UEFI will be discussed in more detail in [Section 5.8.2](#).) Contemporary disk hardware includes

firmware allowing the BIOS or UEFI to access attached storage hardware via *Linear Block Addressing (LBA)*. LBA is a universal, simple way to access data from any disk, but its performance is poor. This isn't a problem, though, because boot loaders are often the only programs that must use this mode for disk access; after starting, the kernel has access to its own high-performance drivers.

NOTE

*To determine if your system uses a BIOS or UEFI, run **efibootmgr**. If you get a list of boot targets, your system has UEFI. If instead you're told that EFI variables aren't supported, your system uses a BIOS. Alternatively, you can check to see that `/sys/firmware/efi` exists; if so, your system uses UEFI.*

Once access to the disk's raw data has been resolved, the boot loader must do the work of locating the desired data on the filesystem. Most common boot loaders can read partition tables and have built-in support for read-only access to filesystems. Thus, they can find and read the files that they need to get the kernel into memory. This capability makes it far easier to dynamically configure and enhance the boot loader. Linux boot loaders have not always had this capability; without it, configuring the boot loader was more difficult.

In general, there's been a pattern of the kernel adding new features (especially in storage technology), followed by boot loaders adding separate, simplified versions of those features to compensate.

5.4.1 Boot Loader Tasks

A Linux boot loader's core functionality includes the ability to do the following:

- Select from multiple kernels.
- Switch between sets of kernel parameters.
- Allow the user to manually override and edit kernel image names and parameters (for example, to enter single-user mode).
- Provide support for booting other operating systems.

Boot loaders have become considerably more advanced since the inception of the Linux kernel, with features such as command-line history and menu systems, but a basic need has always been flexibility in kernel image and parameter selection. (One surprising phenomenon is that some needs have actually diminished. For example, because you can perform an emergency or recovery boot from a USB storage device, you rarely have to worry about manually entering kernel parameters or going into single-user mode.) Current boot loaders offer more power than ever, which can be particularly handy if you're building custom kernels or just want to tweak parameters.

5.4.2 Boot Loader Overview

Here are the main boot loaders that you may encounter:

GRUB A near-universal standard on Linux systems, with BIOS/MBR and UEFI versions.

LILO One of the first Linux boot loaders. ELILO is a UEFI version.

SYSLINUX Can be configured to run from many different kinds of filesystems.

LOADLIN Boots a kernel from MS-DOS.

systemd-boot A simple UEFI boot manager.

coreboot (formerly LinuxBIOS) A high-performance replacement for the PC BIOS that can include a kernel.

Linux Kernel EFISTUB A kernel plug-in for loading the kernel directly from a EFI/UEFI System Partition (ESP).

efilinux A UEFI boot loader intended to serve as a model and reference for other UEFI boot loaders.

This book deals almost exclusively with GRUB. The rationale behind using other boot loaders is that they're simpler to configure than GRUB, they're faster, or they provide some other special-purpose functionality.

You can learn a lot about a boot loader by getting to a boot prompt where you can enter a kernel name and parameters. To do this, you need to know how to get to a boot prompt or menu. Unfortunately, this can sometimes be difficult to figure out because Linux distributions heavily customize boot loader behavior and appearance. It's usually impossible to tell just by watching the boot process which boot loader the distribution uses.

The next sections tell you how to get to a boot prompt in order to enter a kernel name and parameters. Once you're comfortable with that, you'll see how to configure and install a boot loader.

5.5 GRUB Introduction

GRUB stands for *Grand Unified Boot Loader*. We'll cover GRUB 2, but there's also an older version called GRUB Legacy that's no longer in active use.

One of GRUB's most important capabilities is filesystem navigation that allows for easy kernel image and configuration selection. One of the best ways to see this in action and to learn about GRUB in general is to look at its menu. The interface is easy to navigate, but there's a good chance that you've never seen it.

To access the GRUB menu, press and hold SHIFT when your BIOS startup screen first appears, or ESC if your system has UEFI. Otherwise, the boot loader configuration may not pause before loading the kernel. Figure 5-1 shows the GRUB menu.

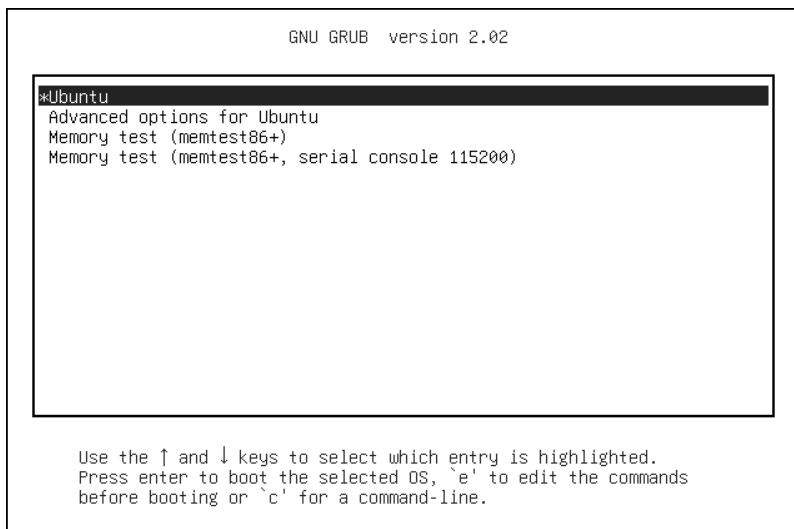


Figure 5-1: GRUB menu

Try the following to explore the boot loader:

1. Reboot or power on your Linux system.
2. Hold down SHIFT during the BIOS self-test or ESC at the firmware splash screen to get the GRUB menu. (Sometimes these screens are not visible, so you have to guess when to press the button.)
3. Press e to view the boot loader configuration commands for the default boot option. You should see something like Figure 5-2 (you might have to scroll down to see all of the details).

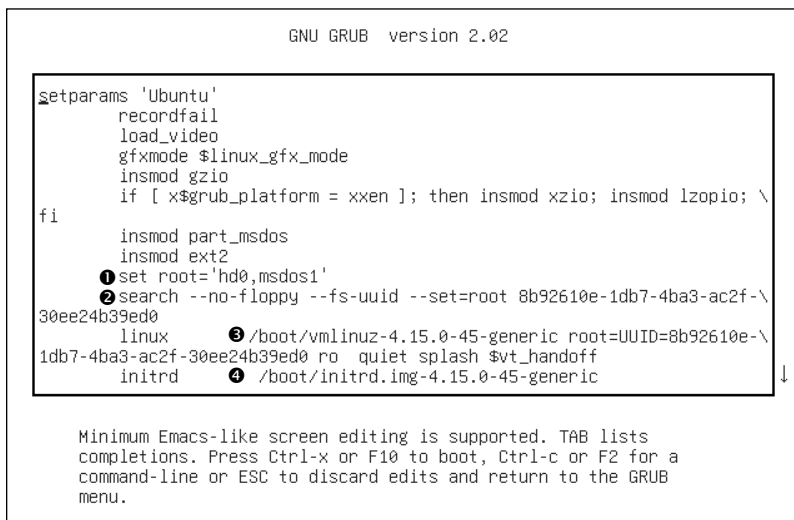


Figure 5-2: GRUB configuration editor

This screen tells us that for this configuration, the root is set with a UUID, the kernel image is `/boot/vmlinuz-4.15.0-45-generic`, and the kernel parameters include `ro`, `quiet`, and `splash`. The initial RAM filesystem is `/boot/initrd.img-4.15.0-45-generic`. But if you've never seen this sort of configuration before, you might find it somewhat confusing. Why are there multiple references to `root`, and why are they different? Why is `insmod` here? If you've seen this before, you might remember that it's a Linux kernel feature normally run by `udev`.

The double takes are warranted, because GRUB doesn't *use* the Linux kernel (remember, its job is to *start* the kernel). The configuration you see consists wholly of features and commands internal to GRUB, which exists in its own separate world.

The confusion stems partly from the fact that GRUB borrows terminology from many sources. GRUB has its own "kernel" and its own `insmod` command to dynamically load GRUB modules, completely independent of the Linux kernel. Many GRUB commands are similar to Unix shell commands; there's even an `ls` command to list files.

NOTE

There's a GRUB module for LVM that is required to boot systems where the kernel resides on a logical volume. You might see this on your system.

By far, the most confusion results from GRUB's use of the word *root*. Normally, you think of `root` as your system's root filesystem. In a GRUB configuration, this is a kernel parameter, located somewhere after the image name of the `linux` command.

Every other reference to `root` in the configuration is to the GRUB root, which exists only inside of GRUB. The GRUB "root" is the filesystem where GRUB searches for kernel and RAM filesystem image files.

In Figure 5-2, the GRUB root is first set to a GRUB-specific device (`hdo,msdos1`), a default value for this configuration ❶. In the next command, GRUB then searches for a particular UUID on a partition ❷. If it finds that UUID, it sets the GRUB root to that partition.

To wrap it up, the `linux` command's first argument (`/boot/vmlinuz- . . .`) is the location of the Linux kernel image file ❸. GRUB loads this file from the GRUB root. The `initrd` command is similar, specifying the file for the initial RAM filesystem covered in Chapter 6 ❹.

You can edit this configuration inside GRUB; doing so is usually the easiest way to temporarily fix an erroneous boot. To permanently fix a boot problem, you'll need to change the configuration (see Section 5.5.2), but for now, let's go one step deeper and examine some GRUB internals with the command-line interface.

5.5.1 Exploring Devices and Partitions with the GRUB Command Line

As you can see in Figure 5-2, GRUB has its own device-addressing scheme. For example, the first hard disk found is named `hd0`, followed by `hd1`, and so on. Device name assignments are subject to change, but fortunately GRUB can search all partitions for UUIDs to find the one where the kernel resides, as you just saw in Figure 5-2 with the search command.

Listing Devices

To get a feel for how GRUB refers to the devices on your system, access the GRUB command line by pressing C at the boot menu or configuration editor. You should get the GRUB prompt:

```
grub>
```

You can enter any command here that you see in a configuration, but to get started, try a diagnostic command instead: `ls`. With no arguments, the output is a list of devices known to GRUB:

```
grub> ls
(hd0) (hd0,msdos1)
```

In this case, there is one main disk device denoted by `(hd0)` and a single partition `(hd0,msdos1)`. If there were a swap partition on the disk, it would show up as well, such as `(hd0,msdos5)`. The `msdos` prefix on the partitions tells you that the disk contains an MBR partition table; it would begin with `gpt` for GPT, found on UEFI systems. (There are even deeper combinations with a third identifier, where a BSD disklabel map resides inside a partition, but you won't normally have to worry about this unless you're running multiple operating systems on one machine.)

To get more detailed information, use `ls -l`. This command can be particularly useful because it displays any UUIDs of the partition filesystems. For example:

```
grub> ls -l
Device hd0: No known filesystem detected - Sector size 512B - Total size
32009856KiB
    Partition hd0,msdos1: Filesystem type ext* - Last modification time
    2019-02-14 19:11:28 Thursday, UUID 8b92610e-1db7-4ba3-ac2f-
    30ee24b39ed0 - Partition start at 1024Kib - Total size 32008192KiB
```

This particular disk has a Linux `ext2/3/4` filesystem on the first MBR partition. Systems using a swap partition will show another partition, but you won't be able to tell its type from the output.

File Navigation

Now let's look at GRUB's filesystem navigation capabilities. Determine the GRUB root with the `echo` command (recall that this is where GRUB expects to find the kernel):

```
grub> echo $root
hd0,msdos1
```

To use GRUB's `ls` command to list the files and directories in that root, you can append a forward slash to the end of the partition:

```
grub> ls (hd0,msdos1)/
```

Because it's inconvenient to type the actual root partition, you can substitute the root variable to save yourself some time:

```
grub> ls ($root)/
```

The output is a short list of file and directory names on that partition's filesystem, such as *etc/*, *bin/*, and *dev/*. This is now a completely different function of the GRUB `ls` command. Before, you were listing devices, partition tables, and perhaps some filesystem header information. Now you're actually looking at the contents of filesystems.

You can take a deeper look into the files and directories on a partition in a similar manner. For example, to inspect the */boot* directory, start with the following:

```
grub> ls ($root)/boot
```

NOTE

Use the up and down arrow keys to flip through the GRUB command history and the left and right arrows to edit the current command line. The standard readline keys (CTRL-N, CTRL-P, and so on) also work.

You can also view all currently set GRUB variables with the `set` command:

```
grub> set
?=0
color_highlight=black/white
color_normal=white/black
--snip--
prefix=(hd0,msdos1)/boot/grub
root=hd0,msdos1
```

One of the most important of these variables is `$prefix`, the filesystem and directory where GRUB expects to find its configuration and auxiliary support. We'll discuss GRUB configuration next.

Once you've finished with the GRUB command-line interface, you can press `ESC` to return to the GRUB menu. Alternatively, if you've set all of the necessary configuration for boot (including the `linux` and possibly `initrd` variables), you can enter the `boot` command to boot that configuration. In any case, boot your system. We're going to explore the GRUB configuration, and that's best done when you have your full system available.

5.5.2 GRUB Configuration

The GRUB configuration directory is usually */boot/grub* or */boot/grub2*. It contains the central configuration file, *grub.cfg*, an architecture-specific directory such as *i386-pc* containing loadable modules with a *.mod* suffix, and a few other items such as fonts and localization information. We won't modify *grub.cfg* directly; instead, we'll use the `grub-mkconfig` command (or `grub2-mkconfig` on Fedora).

Reviewing `grub.cfg`

First, take a quick look at `grub.cfg` to see how GRUB initializes its menu and kernel options. You'll see that the file consists of GRUB commands, which usually begin with a number of initialization steps followed by a series of menu entries for different kernel and boot configurations. The initialization isn't complicated, but there are a lot of conditionals at the beginning that might lead you to believe otherwise. This first part just consists of a bunch of function definitions, default values, and video setup commands such as this:

```
if loadfont $font ; then
    set gfxmode=auto
    load_video
    insmod gfxterm
    --snip--
```

NOTE

Many variables such as `$font` originate from a `load_env` call near the beginning of `grub.cfg`.

Later in the configuration file, you'll find the available boot configurations, each beginning with the `menuentry` command. You should be able to read and understand this example based on what you learned in the preceding section:

```
menuentry 'Ubuntu' --class ubuntu --class gnu-linux --class gnu --class os $menuentry_id_option
'gnulinux-simple-8b92610e-1db7-4ba3-ac2f-30ee24b39ed0' {
    recordfail
    load_video
    gfxmode $linux_gfx_mode
    insmod gzio
    if [ x$grub_platform = xxen ]; then insmod xzio; insmod lzopio; fi
    insmod part_msdos
    insmod ext2
    set root='hd0,msdos1'
    search --no-floppy --fs-uuid --set=root 8b92610e-1db7-4ba3-ac2f-30ee24b39ed0
    linux /boot/vmlinuz-4.15.0-45-generic root=UUID=8b92610e-1db7-4ba3-ac2f-30ee24b39ed0
    ro quiet splash $vt_handoff
    initrd /boot/initrd.img-4.15.0-45-generic
}
```

Examine your `grub.cfg` file for submenu commands containing multiple `menuentry` commands. Many distributions use the `submenu` command for older versions of the kernel so that they don't crowd the GRUB menu.

Generating a New Configuration File

If you want to make changes to your GRUB configuration, don't edit your `grub.cfg` file directly, because it's automatically generated and the system occasionally overwrites it. You'll set up your new configuration elsewhere and then run `grub-mkconfig` to generate the new configuration.

To see how the configuration generation works, look at the very beginning of *grub.cfg*. There should be comment lines such as this:

```
### BEGIN /etc/grub.d/00_header ###
```

Upon further inspection, you'll find that nearly every file in */etc/grub.d* is a shell script that produces a piece of the *grub.cfg* file. The `grub-mkconfig` command itself is a shell script that runs everything in */etc/grub.d*. Keep in mind that GRUB itself does not run these scripts at boot time; we run the scripts in user space to generate the *grub.cfg* file that GRUB runs.

Try it yourself as root. Don't worry about overwriting your current configuration. This command by itself simply prints the configuration to the standard output.

```
# grub-mkconfig
```

What if you want to add menu entries and other commands to the GRUB configuration? The short answer is that you should put your customizations into a new *custom.cfg* file in your GRUB configuration directory (usually */boot/grub/custom.cfg*).

The long answer is a little more complicated. The */etc/grub.d* configuration directory gives you two options: *40_custom* and *41_custom*. The first, *40_custom*, is a script that you can edit yourself, but it's the least stable; a package upgrade is likely to destroy any changes you make. The *41_custom* script is simpler; it's just a series of commands that load *custom.cfg* when GRUB starts. If you choose this second option, your changes won't appear when you generate your configuration file because GRUB does all of the work at boot time.

NOTE

The numbers in front of the filenames affect the processing order; lower numbers come first in the configuration file.

The two options for custom configuration files aren't particularly extensive, and there's nothing stopping you from adding your own scripts to generate configuration data. You might see some additions specific to your particular distribution in the */etc/grub.d* directory. For example, Ubuntu adds memory tester boot options (`memtest86+`) to the configuration.

To write and install a newly generated GRUB configuration file, you can write the configuration to your GRUB directory with the `-o` option to `grub-mkconfig`, like this:

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

As usual, back up your old configuration and make sure that you're installing to the correct directory.

Now we're going to get into some of the more technical details of GRUB and boot loaders. If you're tired of hearing about boot loaders and the kernel, skip to [Chapter 6](#).

5.5.3 GRUB Installation

Installing GRUB is more involved than configuring it. Fortunately, you won't normally have to worry about installation because your distribution should handle it for you. However, if you're trying to duplicate or restore a bootable disk, or preparing your own boot sequence, you might need to install it on your own.

Before proceeding, read [Section 5.8.3](#) to get an idea of how PCs boot and determine whether you're using MBR or UEFI boot. Next, build the GRUB software set and determine where your GRUB directory will be; the default is `/boot/grub`. You may not need to build GRUB if your distribution does it for you, but if you do, see [Chapter 16](#) for how to build software from source code. Make sure that you build the correct target: it's different for MBR or UEFI boot (and there are even differences between 32-bit and 64-bit EFI).

Installing GRUB on Your System

Installing the boot loader requires that you or an installer program determine the following:

- The target GRUB directory as seen by your currently running system. As just mentioned, that's usually `/boot/grub`, but it might be different if you're installing GRUB on another disk for use on another system.
- The current device of the GRUB target disk.
- For UEFI booting, the current mount point of the EFI system partition (usually `/boot/efi`).

Remember that GRUB is a modular system, but in order to load modules, it must read the filesystem that contains the GRUB directory. Your task is to construct a version of GRUB capable of reading that filesystem so that it can load the rest of its configuration (`grub.cfg`) and any required modules. On Linux, this usually means building a version of GRUB with its `ext2.mod` module (and possibly `lvm.mod`) preloaded. Once you have this version, all you need to do is place it on the bootable part of the disk and place the rest of the required files into `/boot/grub`.

Fortunately, GRUB comes with a utility called `grub-install` (not to be confused with `install-grub`, which you might find on some older systems), which performs most of the work of installing the GRUB files and configuration for you. For example, if your current disk is at `/dev/sda` and you want to install GRUB on that disk's MBR with your current `/boot/grub` directory, use this command:

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

WARNING

Incorrectly installing GRUB may break the bootup sequence on your system, so don't take this command lightly. If you're concerned, research how to back up your MBR with `dd`, back up any other currently installed GRUB directory, and make sure that you have an emergency bootup plan.

Installing GRUB Using MBR on an External Storage Device

To install GRUB on a storage device outside the current system, you must manually specify the GRUB directory on that device as your current system now sees it. For example, say you have a target device of `/dev/sdc` and that device's root filesystem containing `/boot` (for example, `/dev/sdc1`) is mounted on `/mnt` of your current system. This implies that when you install GRUB, your current system will see the GRUB files in `/mnt/boot/grub`. When running `grub-install`, tell it where those files should go as follows:

```
# grub-install --boot-directory=/mnt/boot /dev/sdc
```

On most MBR systems, `/boot` is a part of the root filesystem, but some installations put `/boot` into its own separate filesystem. Make sure that you know where your target `/boot` resides.

Installing GRUB with UEFI

UEFI installation is supposed to be easier, because all you have to do is copy the boot loader into place. But you also need to “announce” the boot loader to the firmware—that is, save the loader configuration to the NVRAM—with the `efibootmgr` command. The `grub-install` command runs this if it's available, so normally you can install GRUB on a UEFI system like this:

```
# grub-install --efi-directory=efi_dir --bootloader-id=name
```

Here, `efi_dir` is where the UEFI directory appears on your current system (usually `/boot/efi/EFI`, because the UEFI partition is typically mounted at `/boot/efi`) and `name` is an identifier for the boot loader.

Unfortunately, many problems can crop up when you're installing a UEFI boot loader. For example, if you're installing to a disk that will eventually end up in another system, you have to figure out how to announce that boot loader to the new system's firmware. And there are differences in the install procedure for removable media.

But one of the biggest problems is UEFI secure boot.

5.6 UEFI Secure Boot Problems

One newer problem affecting Linux installations is dealing with the *secure boot* feature found on recent PCs. When active, this UEFI mechanism requires any boot loader to be digitally signed by a trusted authority in order to run. Microsoft has required hardware vendors shipping Windows 8 and later with their systems to use secure boot. The result is that if you try to install an unsigned boot loader on these systems, the firmware will reject the loader and the operating system won't load.

Major Linux distributions have no problem with secure boot because they include signed boot loaders, usually based on a UEFI version of GRUB. Often there's a small signed shim that goes between UEFI and GRUB; UEFI runs the shim, which in turn executes GRUB. Protecting against booting

unauthorized software is an important feature if your machine is not in a trustworthy environment or needs to meet certain security requirements, so some distributions go a step further and require that the entire boot sequence (including the kernel) be signed.

There are some disadvantages to secure boot systems, especially for someone experimenting with building their own boot loaders. You can get around the secure boot requirement by disabling it in the UEFI settings. However, this won't work cleanly for dual-boot systems since Windows won't run without secure boot enabled.

5.7 Chainloading Other Operating Systems

UEFI makes it relatively easy to support loading other operating systems because you can install multiple boot loaders in the EFI partition. However, the older MBR style doesn't support this functionality, and even if you do have UEFI, you may still have an individual partition with an MBR-style boot loader that you want to use. Instead of configuring and running a Linux kernel, GRUB can load and run a different boot loader on a specific partition on your disk; this is called *chainloading*.

To chainload, create a new menu entry in your GRUB configuration (using one of the methods described in the section "[Generating a New Configuration File](#)"). Here's an example for a Windows installation on the third partition of a disk:

```
menuentry "Windows" {
    insmod chain
    insmod ntfs
    set root=(hd0,3)
    chainloader +1
}
```

The `+1` option tells `chainloader` to load whatever is at the first sector of a partition. You can also get it to directly load a file, by using a line like this to load the `io.sys` MS-DOS loader:

```
menuentry "DOS" {
    insmod chain
    insmod fat
    set root=(hd0,3)
    chainloader /io.sys
}
```

5.8 Boot Loader Details

Now we'll look quickly at some boot loader internals. To understand how boot loaders like GRUB work, first we'll survey how a PC boots when you turn it on. Because they must address the many inadequacies of traditional

PC boot mechanisms, boot loading schemes have several variations, but there are two main ones: MBR and UEFI.

5.8.1 MBR Boot

In addition to the partition information described in [Section 4.1](#), the MBR includes a small area of 441 bytes that the PC BIOS loads and executes after its Power-On Self-Test (POST). Unfortunately, this space is inadequate to house almost any boot loader, so additional space is necessary, resulting in what is sometimes called a *multistage boot loader*. In this case the initial piece of code in the MBR does nothing other than load the rest of the boot loader code. The remaining pieces of the boot loader are usually stuffed into the space between the MBR and the first partition on the disk. This isn't terribly secure because anything can overwrite the code there, but most boot loaders do it, including most GRUB installations.

This scheme of shoving the boot loader code after the MBR doesn't work with a GPT-partitioned disk using the BIOS to boot because the GPT information resides in the area after the MBR. (GPT leaves the traditional MBR alone for backward compatibility.) The workaround for GPT is to create a small partition called a *BIOS boot partition* with a special UUID (21686148-6449-6E6F-744E-656564454649) to give the full boot loader code a place to reside. However, this isn't a common configuration, because GPT is normally used with UEFI, not the traditional BIOS. It's usually found only in older systems that have very large disks (greater than 2TB); these are too large for MBR.

5.8.2 UEFI Boot

PC manufacturers and software companies realized that the traditional PC BIOS is severely limited, so they decided to develop a replacement called Extensible Firmware Interface (EFI), which we've already discussed a bit in a few places in this chapter. EFI took a while to catch on for most PCs, but today it's the most common, especially now that Microsoft requires secure boot for Windows. The current standard is Unified EFI (UEFI), which includes features such as a built-in shell and the ability to read partition tables and navigate filesystems. The GPT partitioning scheme is part of the UEFI standard.

Bootting is radically different on UEFI systems compared to MBR. For the most part, it's much easier to understand. Rather than executable boot code residing outside of a filesystem, there's always a special VFAT filesystem called the EFI System Partition (ESP), which contains a directory named *EFI*. The ESP is usually mounted on your Linux system at `/boot/efi`, so you'll probably find most of the EFI directory structure starting at `/boot/efi/EFI`. Each boot loader has its own identifier and a corresponding subdirectory, such as `efi/microsoft`, `efi/apple`, `efi/ubuntu`, or `efi/grub`. A boot loader file has a `.efi` extension and resides in one of these subdirectories, along with other supporting files. If you go exploring, you might find files such as `grubx64.efi` (the EFI version of GRUB) and `shimx64.efi`.

NOTE

The ESP differs from a BIOS boot partition, described in [Section 5.8.1](#), and has a different UUID. You shouldn't encounter a system with both.

There's a wrinkle, though: you can't just put old boot loader code into the ESP, because the old code was written for the BIOS interface. Instead, you must provide a boot loader written for UEFI. For example, when using GRUB, you must install the UEFI version of GRUB rather than the BIOS version. And, as explained earlier in "Installing GRUB with UEFI," you must announce new boot loaders to the firmware.

Finally, as [Section 5.6](#) noted, we have to contend with the "secure boot" issue.

5.8.3 How GRUB Works

Let's wrap up our discussion of GRUB by looking at how it does its work:

1. The PC BIOS or firmware initializes the hardware and searches its boot-order storage devices for boot code.
2. Upon finding the boot code, the BIOS/firmware loads and executes it. This is where GRUB begins.
3. The GRUB core loads.
4. The core initializes. At this point, GRUB can now access disks and filesystems.
5. GRUB identifies its boot partition and loads a configuration there.
6. GRUB gives the user a chance to change the configuration.
7. After a timeout or user action, GRUB executes the configuration (the sequence of commands in the *grub.cfg* file, as outlined in [Section 5.5.2](#)).
8. In the course of executing the configuration, GRUB may load additional code (modules) in the boot partition. Some of these modules may be preloaded.
9. GRUB executes a boot command to load and execute the kernel as specified by the configuration's *linux* command.

Steps 3 and 4 of this sequence, where the GRUB core loads, can be complicated due to the inadequacies of traditional PC boot mechanisms. The biggest question is "Where *is* the GRUB core?" There are three basic possibilities:

- Partially stuffed between the MBR and the beginning of the first partition
- In a regular partition
- In a special boot partition: a GPT boot partition, ESP, or elsewhere

In all cases except where you have an UEFI/ESP, the PC BIOS loads 512 bytes from the MBR, and that's where GRUB starts. This little piece (derived from *boot.img* in the GRUB directory) isn't yet the core, but it contains the start location of the core and loads the core from this point.

However, if you have an ESP, the GRUB core goes there as a file. The firmware can navigate the ESP and directly execute all of GRUB or any other operating system loader located there. (You might have a shim in the ESP that goes just before GRUB to handle secure boot, but the idea is the same.)

Still, on most systems, this isn't the complete picture. The boot loader might also need to load an initial RAM filesystem image into memory before loading and executing the kernel. That's what the `initrd` configuration parameter specifies, and we'll cover it in [Section 6.8](#). But before you learn about the initial RAM filesystem, you should learn about the user space start—that's where the next chapter begins.

