10

MAINTAINING THE KERNEL

As mentioned in Section 2.2, the kernel is the core of the operating system. In some respects, the Linux kernel is no different than any other software package. You can configure, build, and install the kernel because it comes as source code. However, the procedure for doing so is substantially different than for any other software package because the kernel *runs* like no other package.

There are four basic topics pertaining to kernel management:

Configuring and compiling a new kernel Your end goal here is to build an image file, ready for the boot loader.

Manipulating loadable kernel modules You don't have to build the kernel as one massive chunk of code. Kernel modules let you load drivers and kernel services as you need them.

Configuring a boot loader such as GRUB or LILO Your kernel is useless if you can't load it into memory.

Learning miscellaneous utilities and procedures There are many facilities that tweak runtime kernel parameters and extend kernel features. You have already seen some examples of these in previous chapters, including the `/proc` filesystem and the `iptables` command.

10.1 Do You Need to Build Your Own Kernel?

Before running head-first into a kernel build, you need to ask yourself if it is worth it. Administrators who compile their own kernels have the following goals in mind:

- Installing the latest drivers
- Using the latest kernel features (especially with respect to networking and filesystems)
- Having fun

However, if you need a driver or feature, and your distribution offers a straightforward upgrade, you might opt for that instead, for several reasons:

- You can make mistakes in configuring your own kernel, and your distribution probably offers a well-rounded kernel.
- Compiling a new kernel takes a long time.
- It's far too easy to mess up the boot loader, making your system unable to boot.

Even if you choose not to build your own kernel, though, you should still learn how boot loaders and modules work.

10.2 What You Need to Build a Kernel

As with any package, the kernel source includes documentation explaining what you need to compile a functional kernel. These three aspects of your system are particularly important for building a kernel:

- A C compiler (`gcc`). Most distributions put the C compiler in a development tools package. Make sure that your C compiler agrees with the recommendations in the kernel source code, specifically the `README` file. Kernel developers have not been eager to adopt the latest versions of `gcc`. Some distributions have a separate compiler named `kgcc` for compiling the kernel.
- Disk space. A new kernel source tree can easily unpack to more than 100MB even before building any object files.
- A relatively fast computer with plenty of memory. Otherwise, the compile will take some time.

Your first step is to get the source code.

10.3 Getting the Source Code

Linux kernel versions have three components. Suppose you have kernel release 2.6.3. Here's what the numbers mean:

- 2 is the *major* release number.
- 6 is the *minor* release number.
- 3 is the *patchlevel*. Kernels with the same major and minor numbers but with different patchlevels belong to a *release series*.

There are two different kinds of kernel releases:

- *Production releases* have minor version numbers that are even (for example, 1.**2**.*x*, 2.**0**.*x*, 2.**2**.*x*, 2.**4**.*x*, 2.**6**.*x*, and so on). These versions are meant to be as stable as possible. Within a single production release series, there are no radical feature or interface changes.
- *Development releases* have minor version numbers that are odd (such as 2.**5**.). Don't use a developer release if you aren't willing to take risks. Kernel developers intentionally break code, introduce new features, and may inadvertently make the kernel unstable in a development series. Stay away from development releases unless you know exactly what you're doing.

You can get the latest Linux kernel source code at http://www.kernel.org/. The releases have names like `linux-`*`version`*`.tar.bz2` for bzip2 compression or `linux-`*`version`*`.tar.gz` for GNU Zip compression. Bzip2 archives are smaller.

In addition to the full releases available in the kernel archives, you will also find patches containing just the changes between two consecutive kernel patchlevel releases. Unless you are obsessed with upgrading your kernel with every single release, you probably won't be able to do much with a patch.

10.3.1 Unpacking the Source Archive

You can unpack kernels with `zcat`/`bzip2` and `tar`. To get started, go to `/usr/src` and make your permissions less restrictive if necessary:

```
cd /usr/src
umask 022
```

Then run one of these commands to extract the source files, depending on whether you are using bzip2 or GNU Zip:

```
bzip2 -dc linux-version.tar.bz2 | tar xvf -
zcat linux-version.tar.gz | tar xvf -
```

WARNING *If your kernel release is 2.4.18 or lower, the kernel unpacks into a directory named* `linux` *rather than* `linux-`*`version`*. *Rename any existing* `linux` *directory before unpacking one of these kernels (the new name does not matter, as long as it is different than the old one).*

10.3.2 A Look Around the Kernel Source

If you've never seen the Linux kernel source code before, it helps to take a look around first. The top-level directory includes several subdirectories and files, including these:

`README` This file is a general introduction to the Linux kernel source and what you need to get started. This document explains the version of the C compiler that you need to compile the kernel.

`Documentation` This directory contains a wealth of documents. Most of the documents here are in plain-text format, and they may describe anything from user-level programs to low-level programming interfaces. One of the most important files in this directory is `Changes`, which describes recent changes to the kernel and identifies utility programs you may need to upgrade to get full functionality.

`include` You'll find the kernel header files in this directory. If you feel comfortable with your kernel, you can use the header files in `include/linux` as your system `/usr/include/linux` header file set.

`arch` This directory contains architecture-specific kernel build files and targets. After a kernel build on a PC system, the final kernel image is in `arch/i386/boot/bzImage`.

10.3.3 Distribution Kernels

Linux distributions come with generic kernels intended to run on almost any processor type. These kernels tend to be a little larger than custom kernels, and they rely heavily on loadable modules.

However, many of the Linux kernels that come with the various Linux distributions omit complete source code in their default installations, and furthermore, distribution kernels often differ from the official standard kernels at kernel.org. For example, Red Hat adds another component to the patchlevel to distinguish their kernels (for example, 2.4.20-**20**). If you wish to install a distribution kernel from source code, you need to use the distribution's kernel source package. In Red Hat Linux, the package name is `kernel-source-`*`version`*`.rpm`.

If your distribution has a `/usr/src/linux` directory, this does *not* mean that you have the entire kernel source. It is possible that you only have the header files. The kernel source takes up dozens of megabytes; run `du -s /usr/src/linux` for a quick check on what you have.

10.4 Configuring and Compiling the Kernel

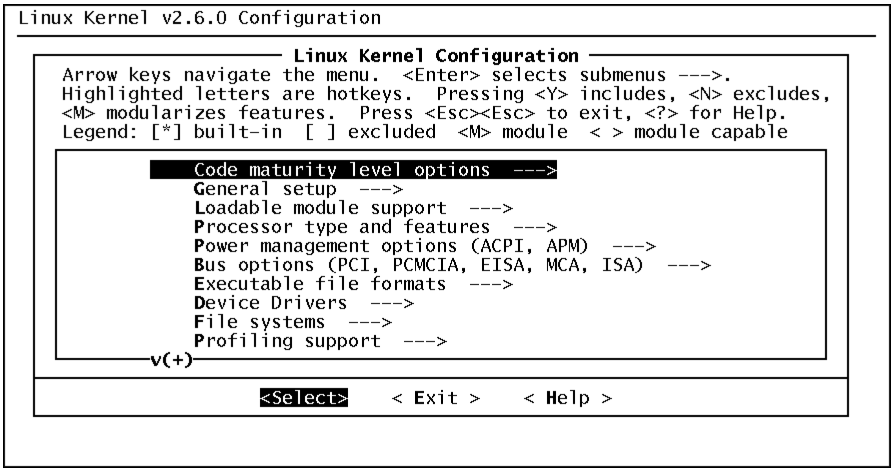
You need to configure the options that you want to have in your kernel before you build it. Your goal is to have an appropriate `.config` file in your kernel source distribution. Here's an excerpt of a typical `.config` file:

```
# CONFIG_HIGHMEM64G is not set
# CONFIG_MATH_EMULATION is not set
CONFIG_MTRR=y
CONFIG_HAVE_DEC_LOCK=y
```

This file isn't easy to write with a text editor, so there are several configuration utilities that generate the file for you. An easy, standard way to set up a kernel is to run this command:

```
make menuconfig
```

After some initial setup, `make menuconfig` runs a text-based menu interface that is shown in Figure 10-1. The box in the center of the screen contains the options at the current configuration level. You can navigate the menu with the up- and down-arrow keys. An arrow next to an item (`--->`) indicates a submenu.



```
Linux Kernel v2.6.0 Configuration

  ──────────────────── Linux Kernel Configuration ────────────────────
   Arrow keys navigate the menu.  <Enter> selects submenus --->.
   Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
   <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help.
   Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

              Code maturity level options  --->
              General setup  --->
              Loadable module support  --->
              Processor type and features  --->
              Power management options (ACPI, APM)  --->
              Bus options (PCI, PCMCIA, EISA, MCA, ISA)  --->
              Executable file formats  --->
              Device Drivers  --->
              File systems  --->
              Profiling support  --->
  ──────────v(+)

                   <Select>    < Exit >    < Help >
```

Figure 10-1: The `make menuconfig` *kernel configuration menu.*

The bottom of the menu contains three actions that you can navigate to with the left- and right-arrow keys. Pressing ENTER while one of the menu items is highlighted performs whichever of the three following actions is also highlighted:

Select If the current menu item leads to a submenu, the configuration system activates the submenu.

Exit Exits the current submenu. If you are at the top-level menu, the configuration system asks whether you would like to save the current configuration. You can activate the **Exit** option by pressing ESC.

Help Shows any online help for the current item. You can also get at the help by pressing the ? key.

To get a general feel for how configuration options work, go into the **Processor type and features** submenu. You will see options such as these:

```
[ ] Math emulation
[*] MTRR (Memory Type Range Register) support
< > /dev/cpu/microcode - Intel IA32 CPU microcode support
```

You can alter an item's configuration value by moving to the item and pressing the SPACEBAR. The square brackets (`[ ]`) provide a simple on/off toggle:

`[*]` indicates that the feature is on.

`[ ]` indicates that the feature is off.

You may not configure on/off features as kernel modules. Active features go directly into the main kernel image.

By contrast, an item with angle brackets (`< >`) denotes a feature that you may compile as a module. You can use the SPACEBAR to toggle between these values:

`<*>` indicates that the feature is on.

`<M>` indicates that the feature is configured as a module.

`< >` indicates that the feature is off.

An item that includes regular parentheses like these, `( )`, is a multiple-choice option or a number that you can customize. Press ENTER to see the option list, or enter a number.

When Should You Compile a Driver as a Module?

There aren't many absolute rules for deciding which features to compile as modules and which features to compile as part of the main kernel image, but here are a few guidelines and rules that can help you:

- Always compile your root filesystem type directly into the kernel, not as a module. Your system won't boot otherwise.
- Most administrators prefer to compile crucial disk support (such as SCSI host adapter drivers) directly into the kernel. There is a way to get around this with an initial RAM disk, but that is more an ugly hack than anything else.
- Most administrators also compile network interface card drivers directly into the kernel. This isn't as crucial as disk support, though.
- Compile obscure filesystems as modules. You may need them at some point, but there is no point in wasting memory in the meantime.

When you are through with configuration and decide to exit, the menu system will ask if you would like to save the current configuration. If you choose **Yes**, the system backs up your previous `.config` file as `.config.old` and writes a new `.config` file.

10.4.1 Configuration Options

Most of the configuration options are fairly self-explanatory. For example, in the menu shown in Figure 10-1 on page 205, the items under **Device Drivers > SCSI device support > SCSI low-level drivers** correspond to device drivers for SCSI host controllers.

Some configuration options depend on other options, and until you activate the dependency options you cannot reach configuration options that have dependencies. For example, to reach the **SCSI low-level drivers** submenu, you must first activate **SCSI device support** in the previous menu. These dependencies are not always obvious; you may need to do a little bit of experimentation to find what you're looking for. If you're really stumped on a dependency, go to the source. Look at `arch/i386/Kconfig`, the master file for all options and online help for the kernel configuration.

The following sections outline the most significant kernel options that reside within the important top-level menus. Keep in mind that these options change with time; you may see items not listed here, or the items may be in a different place in the kernel that you decide to build.

Code Maturity Level Options

Inside this menu item you will find an option named **Prompt for development and/or incomplete code/drivers**. To see the newest (but perhaps unstable) drivers and features when perusing the rest of the kernel configuration menus, select this option.

General Setup

This section has three settings that you should turn on:

Support for paging of anonymous memory Enables swap disk and file support

System V IPC Enables interprocess communication

Sysctl support Enables kernel parameter changes through `/proc`

If you do not build your kernel with the support listed here, many major packages and utilities will not work, including the X Window System.

In addition to these options, the **General Setup** menu in kernel 2.6.0 and newer versions have an option named **Kernel .config support** to make the build process save the current `.config` file in the kernel image. If you enable this option, you will also have the opportunity to enable a second option that allows you to turn on access to the embedded `.config` file via

`/proc/config.gz`. These options increase your kernel size slightly, but they can come in extremely handy later on when you need to upgrade your kernel but can't remember what configuration options you selected.

Loadable Module Support

You have some control over the kernel's module loader. In general, you should always activate the kernel module system with the **Enable loadable module support** option, as well as activating the kernel module loader option, **Automatic kernel module loading**, sometimes called the autoloader.

The only item in the kernel module support menu that you should be wary of is versioning support (at the moment, this is experimental). Kernels compiled with this option enabled may try to load modules built for a different kernel version, and this can cause trouble if you don't know exactly what you're doing.

Processor Type and Features

The Linux kernel includes a number of optimizations and other enhancements that are specific to certain processors. You can choose your processor with the **Processor family** option. Be careful — a kernel built for a "primitive" CPU supports advanced processors, but a kernel tailored to an advanced CPU will likely not work on an older (or different brand of) CPU.

Other significant options in the processor category include:

High memory support This is for machines with more than 2GB of physical memory.

Symmetric multi-processing support This is for machines with more than one processor.

MTRR support This permits optimizations that may improve certain kinds of graphics performance.

Power Management Options

There are two types of power management in PC hardware: the older APM (Advanced Power Management) and the newer ACPI (Advanced Configuration and Power Interface). You can configure a kernel with both varieties. Appropriate power management support is essential on notebooks to preserve battery life, and it is a good idea for desktops so that your machine doesn't generate too much heat and use too much electricity.

Power management features enable interesting tricks with fans, processor speeds, and more. There's a lot to explore, but you may need additional software such as `apmd` or `acpid` to take advantage of the kernel features.

Bus Options

You shouldn't need to change the bus options for most systems. Unless you have an extremely old or odd system, include PCI support (and if you'd like to list and diagnose devices on the PCI bus, install `lspci`, which is a part of the pci-utils package).

You should enable the **Support for hot-pluggable devices** option so that your system can take appropriate action when you attach and detach removable devices.

Newer kernels include base PCMCIA (PC Card/CardBus) drivers as configuration options in this menu — in previous systems, PCMCIA support was completely separate from the main kernel distribution. You still need the PCMCIA utilities if you intend to use PC cards, and at the moment, you can still leave out PCMCIA kernel support here and get it from these utilities, but this may change in the future.

Executable File Formats

To start a process from an executable file on the disk, the kernel needs to know how to load the executable file, how to initialize the process in memory, and where to start running the process. An executable's file format determines these characteristics. Most binaries on a modern Linux system are ELF (Executable and Linkable Format) files. You probably do not need to support the ancient "a.out" executable format (you may not even have the shared libraries to support these binaries), but it does not take too much memory to include support, and you can build it as a module.

Device Drivers

Configuring device drivers is a long process due to the wide variety of devices that Linux supports. Unfortunately, it's during this stage that you can get bogged down with all of the options and end up building a kernel that is far too large because you included drivers for devices that you will never have. If you're in doubt about whether you are going to use any particular driver (other than a disk driver), build it as a module so that it does not unnecessarily take up kernel memory.

The following sections describe the driver configuration options inside the **Device drivers** menu.

Plug and Play Support

You must enable **Plug and Play support** if you want any reasonably modern built-in hardware in your computer to work. You may also need plug-and-play support in order to use network cards, internal modems, and sound cards.

Block Devices

The **Block devices** section of the kernel configuration contains miscellaneous random-access storage devices, such as floppy disks and RAM disks. These devices normally hold filesystems that you directly attach to your current system with the `mount` command. The interesting drivers here include the following:

Normal floppy disk support A driver for the PC floppy disk drive.

Loopback device support A simple driver that maps a file into a block device. This driver is very useful because it allows you to mount a filesystem on a disk image.

Network block device support A driver that allows you to use a network server as a block device.

RAM disk support A driver that designates a chunk of physical memory as disk space.

Initial RAM disk (initrd) support A driver that provides a special kind of RAM disk that a boot loader gives to the kernel as the initial `/` partition during the very first stages of `init`. Many distributions use an initial RAM disk for SCSI drivers and other modules that may not be in their stock kernels.

Parallel port IDE device support A driver for certain older portable disk devices.

ATA (IDE) Support

You should always compile ATA (IDE) support directly into your kernel unless you know exactly what you're doing. There are several kinds of IDE drivers that you can enable here:

Disk support A driver required if you want hard drives to work. Needless to say, you should compile this driver directly into the kernel.

CD-ROM support A driver for ATAPI CD-ROM and DVD-ROM drives.

Floppy support A driver for various removable-media drives with an ATAPI interface.

SCSI emulation A driver for ATAPI CD-R and CD-RW drives. Older Linux CD-burning software worked exclusively with SCSI drivers; if your CD-burning software is up to date, you do not need this driver.

Various chipsets Drivers for various specific chipsets. If your motherboard's IDE chipset is listed, you might be able to squeak out a little more performance with one of these drivers.

SCSI Support

Even if you have no SCSI devices, you may still need SCSI support because many Linux device drivers work through emulated SCSI devices. For example, if you want to use USB mass storage, you need SCSI disk support.

These are the media drivers:

SCSI disk support Covers all fixed and removable-media disk-like storage devices, except CD-ROM drives.

SCSI tape support

SCSI CD-ROM support

SCSI generic support Allows applications talk to a SCSI device at a low level. You need SCSI generic support for a wide range of devices, including CD burners, changer devices on tape drives, SCSI scanners, and more.

A look inside the **SCSI low-level drivers** submenu reveals drivers for many kinds of SCSI host controllers. Some of the drivers are named after the SCSI chipset on the host controller, so you may have to look carefully at your hardware to find a match. The low-level driver list also includes support for some parallel port devices, such as Zip drives.

NOTE *If your kernel and root partition are on a SCSI disk, compile SCSI support (including the disk and host controller drivers) directly into the kernel rather than as modules. This makes booting much easier, because you do not need to worry about loading SCSI modules from a disk that the kernel does not yet know how to access.*

Networking Support

You need networking support for almost any system, even if you do not have an external network connection. Many applications use local network interface features.

Your first order of business is to pick the networking devices that you need. For most Ethernet devices, look under **Ethernet (10 or 100Mbit)**. There are many devices to choose from. If you're not too sure what you need, don't be afraid to configure several drivers as modules. In addition, if you plan to dial up to the Internet or use certain DSL connections, you need **PPP support**, along with asynchronous (serial port) or PPP over Ethernet (PPPoE) support. If you're not sure about PPP options, configure them as modules.

The configuration options inside the **Networking options** submenu represent the trickiest part of the kernel network configuration. You need to select all of the options and protocols that you intend to use. The essential options for many packages include the following:

TCP/IP networking

Unix domain sockets

Packet socket (for features such as promiscuous mode)

Not so obvious is **Network packet filtering**, which is required for any kind of firewall or NAT (Network Address Translation, or "IP masquerading") support. Just selecting this option is not enough because, in general, your kernel needs all of the connection tracking, NAT, and masquerade options if you want to use NAT. Therefore, you must enter the **Netfilter Configuration** submenu and choose more options there, such as these:

Connection tracking for NAT (be sure to include any protocols you need to track).

IP tables support for firewalling and NAT. Once you enable IP tables, you get several more options, including various **match** options for filtering. You don't have to worry about most of these, except for **Connection state** and **Connection tracking**, both of which you need for NAT. That's not the end of the things you need for NAT — look out for and enable **Full NAT** and **MASQUERADE target support**.

Packet filtering and **REJECT target support** for firewalls.

Input Device Support

You need several configuration options to support your keyboard and mouse. There are two levels of support. The basic drivers support the PS/2 keyboard. Also, look for **PS/2 Mouse**.

Character Devices and Parallel Port Support

Among the most important character devices are **Virtual terminals** (`/dev/tty0`, `/dev/tty1`, and so on). Unless you have a special type of server, you also need to put the console on the virtual terminal.

Here are some other things to check up on:

Serial drivers Standard PC serial ports use or emulate 8250/16550 UART chips. You almost certainly need this support if you plan to use a modem.

Unix98 PTY support Some programs now expect to see the dynamic Unix98 `/dev/pts/*` pseudo-terminal devices rather than old static names like `/dev/ttyp*`.

Enhanced real time clock support This makes `/dev/rtc` available to programs like `hwclock`. You should enable this option for any modern machine.

`/dev/agpart` This is for direct-rendered graphics (GLX and DRI) on AGP graphics cards.

Parallel printer support This gives you `/dev/lp*` devices. To get this option, you must go back to the main kernel menu and enable **Parallel port support**. You also need **PC-style hardware** for any standard PC.

NOTE *There is a section for* ***Mice*** *in the character device configuration, but you can probably safely ignore it, because the drivers there are for very old bus mice.*

Sound

There are two parts to sound configuration: API support and drivers. The current sound system is called Advanced Linux Sound Architecture (ALSA), but you should also include the emulation support for the old OSS (Open Sound System), because many programs still use the OSS interface.

The driver configuration for sound devices is very similar to that for network devices — many of the sound drivers carry the names of the sound chipset.

USB Support

When configuring USB, choose an interface driver first:

UHCI and **OHCI** are USB 1.1 interfaces. Even if you have a USB 2.0 motherboard, you need one of these. If you don't know which one your motherboard supports, pick both (modules are okay); the kernel will sort it out.

EHCI is a USB 2.0 interface.

You also want to enable the **USB device filesystem** option that can maintain USB device information in `/proc/bus/usb`.

Configuring the kernel to support USB devices is fairly straightforward, but there are two gotchas:

- The **Mass storage support** option requires that you also enable SCSI and SCSI disk support.
- The **Interface device support (HID)** option requires that you enable the input drivers described earlier. To get USB mouse and keyboard support, you need to enable **HID input layer support** in the USB support menu. Don't use the boot protocol drivers unless you know exactly what you're doing.

Filesystems

You can add support for many different filesystems to your Linux kernel configuration. However, *make sure* that you compile your primary filesystem directly into the kernel. For example, if your root partition uses ext2, don't modularize ext2 support. Otherwise, your system will not be able to mount the root filesystem and therefore will not boot.

These are the most important filesystems:

Second extended (ext2) The Linux standard for many years.

Third extended (ext3) A journaled version of ext2; now the standard for many distributions.

Reiserfs A high-performance journaled filesystem.

DOS/FAT/NT MS-DOS- and Windows-compatible filesystems. To get MS-DOS filesystem support, you need to enable FAT. VFAT is the extended filesystem introduced with Windows 95. You need VFAT if you plan to read images from the flash memory cards in digital cameras and similar devices.

ISO9660 A standard CD-ROM filesystem. The Linux driver includes the Rock Ridge extensions. You can also add the Microsoft Joliet extensions.

UDF A newer CD-ROM and DVD filesystem.

Minix A filesystem that was the Linux standard a long time ago (it even predates Linux). It never hurts to include Minix filesystem support as a module.

Pseudo filesystems These are system interfaces, not storage mechanisms. The `/proc` filesystem is one of the best-known pseudo-filesystems, and one that you should *always* configure. You should also include the `/dev/pts` filesystem and the virtual memory filesystem.

Network filesystems These are used for sharing files with machines over the network.

There are also a couple important filesystem-related options:

Kernel automounter support Enables automatic filesystem mounting and unmounting. This is a popular solution for managing network filesystem access.

Partition types Allows Linux to read partitioning schemes other than the regular PC partition tables, including BSD disklabels and Solaris x86 partitions.

This wraps up the important kernel configuration options. You shouldn't really have to worry about the other options (such as the Profiling support used for kernel development); let's turn our focus to compiling the kernel.

10.4.2 Compiling the Kernel and Modules

After you save your kernel configuration, you're ready to build the kernel. To get a list of `make` targets, you can run this command:

```
make help
```

Your main goals are to compile the `bzImage` compressed kernel image and the kernel modules. Because these two are the default targets, you only need to run the following to build everything:

```
make
```

The compile takes some time. As it runs along, you see messages like this:

```
  CC      init/version.o
```

If you see a message containing `[M]`, it means that `make` is compiling a module:

```
  CC [M]  net/sctp/protocol.o
```

The following output appears when `make` builds `bzImage`:

```
Kernel: arch/i386/boot/bzImage is ready
```

As the preceding message indicates, the build process creates your kernel image as `arch/i386/boot/bzImage`. The process also creates a file named `System.map` that contains kernel symbols and locations. You may have to wait a little while after the "Kernel ready" message appears, because `make` may still need to compile some kernel modules. Don't interrupt the build; make sure that you wait until you get your prompt back before installing the kernel.

NOTE *In kernel versions prior to 2.6, you had to run the following commands for a complete kernel build:*

```
make dep
make bzImage
```

Failed Compiles

Recent kernels are very self-contained; there are only two primary things that can go wrong during the build process:

- If you get a parse error or some other sort of coherent compiler error, the compiler on your machine probably doesn't match the recommended compiler in the `README` file. Kernel code is very particular about the compiler, especially due to all of the assembly code involved.
- If the compiler dies unexpectedly midway through the build process, your hardware may be at fault. Compiling a kernel stresses a machine much more than almost any other task. Bad memory and overclocked or overheated processors are the main culprits.

If you need to see each command in the build process to track down a problem, use this command:

```
make V=1
```

10.4.3 Installing Modules

Before you install and boot from your new kernel, you should put the new kernel modules in `/lib/modules` with this command:

```
make modules_install
```

Your new modules should appear in `/lib/modules/`*`version`*, where *`version`* is your kernel version. If you fail to install the modules before booting your kernel, the kernel module utilities will not automatically recognize the new modules, and you may end up missing a few drivers.

The module installation builds a module dependency list in `/lib/modules/version/modules.dep`. If this stage fails, it's likely that your module utilities are out of date. As of this writing, the module utilities are in a package named `module-init-tools`.

10.4.4 Kernel Parameters

Sometimes you need to send extra parameters to the kernel at boot time in order to get devices or services working. For example, one of the most elementary kernel parameters is `root=`*`partition`*, which sets the initial root partition to *`partition`*.

You can enter a kernel parameter after the kernel image name. For example, at a LILO prompt where your kernel label is `Linux`, you can type this:

```
Linux root=/dev/hda3
```

`Documentation/kernel-parameters.txt` contains a list of all kernel parameters. Most of them are for hardware. In addition to `root=`*`partition`*, here are the most important parameters:

`init=`*`path`* This starts *`path`* as the first process on the system in place of `/sbin/init`. For example, you can use `init=/bin/sh` to get out of a tight spot if your `init` does not work properly.

`mem=`*`size`* This specifies that the machine has *`size`* memory; the kernel should not autodetect the memory size. For example, to specify 512MB of memory, you would use `mem=512M`.

`rootfstype=`*`type`* This specifies that the root filesystem's type is *`type`*.

A number as a boot parameter indicates the `init` runlevel. You can use `-s` or `S` for single-user mode.

10.5 Installing Your Kernel with a Boot Loader

Boot loaders load kernel images into memory and hand control of the CPU over to the newly loaded kernel. To make your new kernel work, you must tell the boot loader about the kernel.

If you have never worked with a boot loader before, you need to know how a PC boots from the hard disk. On a normal Microsoft-based system, the boot process works like this:

1. After performing the power-on self test (POST), the BIOS loads sector 1 of the hard disk into memory and runs whatever landed in that memory. On most Windows-based systems, this sector is called the Master Boot Record (MBR).
2. The MBR extracts the partition table from the disk and identifies the "active" partition.
3. The MBR loads and runs yet another boot sector from the active partition, setting the operating system in motion.

On a Linux system, you can install the kernel boot loader on the active partition *or* you can replace the MBR with the boot loader. If you decide to replace the MBR with a boot loader such as GRUB or LILO, it is important to remember that the active partition has little meaning — these boot loaders address partitions based on their own configuration systems.

From a practical point of view, your decision to overwrite the MBR (and thus circumvent the active partition) makes little difference. If your machine runs Linux, you probably want GRUB or LILO as your default boot loader. Even if you have a dual-boot machine, you still want to use GRUB or LILO, because both boot loaders are capable of loading boot sectors from other operating systems.

The important point is that you know exactly where your boot loader resides. Let's say that your disk is `/dev/hda` and you have a Linux root partition at `/dev/hda3`. If you replace the MBR by writing GRUB to `/dev/hda`, you need to remember that the active partition is now irrelevant; GRUB uses its own configuration system to access partitions. However, if you decide to write the boot loader to `/dev/hda3` instead, keeping the old MBR, your system will get to that boot loader only if `/dev/hda3` is the active partition.

NOTE *If you need to replace the MBR on a hard disk, run the DOS command* `FDISK /MBR`.

When configuring a boot loader, be sure that you know the location of the root partition and any kernel parameters.

10.5.1 Which Boot Loader?

Because there are two popular Linux boot loaders, you may wonder which one you should choose:

LILO Linux Loader. This was one of the very first Linux boot loaders. Its disadvantages are that it is not terribly flexible and that you must run `lilo` to install a new boot block every time you install a new kernel. However, LILO is fairly self-contained.

GRUB Grand Unified Boot Loader. This is a newer boot system gaining in popularity, and it does not need a reconfiguration for every new kernel because it can read many kinds of filesystems. This feature is especially handy for situations where you might need to boot from an old kernel. Initially, GRUB is slightly trickier to configure than LILO, but it is much easier to deal with once installed, because you do not need to replace the boot sector for every new kernel. I recommend GRUB.

10.5.2 GRUB

With GRUB, you need only install the boot loader code on your machine once; after that, you can modify a GRUB menu file for any new kernel that you want to boot on a permanent basis.

The GRUB boot files are in `/boot/grub`. GRUB loads the various `stage` files into memory during the boot process. If you already have GRUB on your machine, you just need to modify the `/boot/grub/menu.lst` file when you add a new kernel image under a new name. (If you install a new kernel image with the same name as the old kernel, you do not need to modify the menu file.)

Some distributions preinstall GRUB but have a different name for `menu.lst`. You may need to dig around in `/boot` to find the correct filename. In any case, the menu file looks like this:

```
default 0
timeout 10

title Linux
  kernel (hd0,0)/boot/vmlinuz root=/dev/hda1

title backup
  kernel (hd0,0)/boot/vmlinuz.0 root=/dev/hda1
```

The parameters in the menu file are as follows:

`default` The `title` entry that GRUB should boot by default. 0 is the first entry, 1 is the second, and so on.

`timeout` How long GRUB should wait before loading the default image.

`title` A label for a kernel.

`kernel` The kernel image and parameters, possibly including the root device.

In GRUB, `(hd0)` is the first hard disk on your system, usually `/dev/hda`. GRUB assigns disk mappings in the order that they appear on your system, so if you have `/dev/hda` and `/dev/hdc` devices but no `/dev/hdb`, GRUB would assign `(hd0)` to `/dev/hda` and `(hd1)` to `/dev/hdc`. The first number in the GRUB name is for the disk, and the second is for the partition (if there is a partition). Check the `/boot/grub/device.map` file for your system's hard drive mapping.

The kernel images in the preceding example are on the root partition of the primary master ATA hard disk. There are two kernel images: a regular kernel at /boot/vmlinuz and a backup at /boot/vmlinuz.0.

Root Device Confusion

You may find it odd that the root partition is actually specified twice in the kernel line in the preceding example (you can see (hd0,0) and /dev/hda1). This is easy to explain: the (hd0,0) is where GRUB expects to find the kernel image, and root=/dev/hda1 is a Linux kernel parameter that tells the kernel what it should mount as the root partition. These two locations are usually, but not always, the same. However, the GRUB and Linux device names are completely different, and so you need them in two places in the configuration file.

Unfortunately, you may see this alternate syntax in menu.lst:

```
root (hd0,0)
kernel /boot/vmlinuz.0 root=/dev/hda1
```

This is confusing, because GRUB does not pass its root parameter (hd0,0) to the Linux kernel. Omitting the GRUB root parameter can prevent some head scratching.

Booting Other Operating Systems

There are all sorts of other things you can do with GRUB, like load splash screens and change the title (use info grub to see all of the options). However, the only other essential thing you should know is how to make a dual-boot system.

Here is a definition for a DOS or Windows system on /dev/hda3:

```
title dos
  rootnoverify (hd0,2)
  makeactive
  chainloader +1
```

Remember how the PC boot loaders usually work, by first loading whatever is on the first sector of the disk, then loading the first sector of the active partition.

Installing GRUB

To put GRUB on your system for the first time, you must make sure that you have a menu.lst file. The GRUB installation process does not create menu.lst; you must come up with this file on your own. If you don't, you can still boot your system, but you must type a series of commands resembling entries in menu.lst to do so, as follows:

```
kernel (hd0,0)/boot/vmlinuz root=/dev/hda1
boot
```

To install the GRUB software, run this command to put the boot sector on your disk:

```
grub-install device
```

Here, *device* is your boot device, such as /dev/hda. However, if you have a special /boot partition, you need to do something like this:

```
grub-install --root-directory=/boot device
```

After running grub-install, review your /boot/grub/device.map file to make sure that the devices relevant to booting the kernel are in the map file and agree with your menu.lst file.

10.5.3 LILO

To load your kernel with LILO, you must settle on a permanent location for your kernel image and install a new boot block for every change you make to the kernel configuration.

Let's say that you want to boot /vmlinuz as your kernel with a root partition of /dev/hda1. Furthermore, you want to install the boot loader on /dev/hda (replacing the MBR). Do the following:

1. Move the new image into place at /boot/vmlinuz.
2. Create the LILO configuration file, /etc/lilo.conf. An elementary configuration might look like this:

   ```
   boot=/dev/hda
   root=/dev/hda1
   install=/boot/boot.b
   map=/boot/map
   vga=normal
   delay=20

   image=/boot/vmlinuz
       label=Linux
       read-only
   ```

3. Run lilo -t -v to test the configuration without changing the system.
4. Run lilo to install the boot loader code at /dev/hda.

You can add more images and boot sectors to the LILO configuration. For example, if you saved your previous kernel as /boot/vmlinuz.0, you could add this to your lilo.conf to make the old image available as backup at the LILO prompt:

```
image=/boot/vmlinuz.0
    label=backup
    read-only
```

You can use the `other` keyword for foreign operating systems. The following addition to `lilo.conf` offers a Windows partition on `/dev/hda3` as `dos` at the LILO prompt:

```
other=/dev/hda3
    label=dos
```

LILO Parameters

Some of the most important `lilo.conf` parameters are listed here:

`boot=`*`bootdev`* Writes the new boot sector to the *`bootdev`* device.

`root=`*`rootdev`* Uses *`rootdev`* as the kernel's default root partition.

`read-only` Mounts the root partition initially as read-only. You should include this for normal boot situations (`init` remounts the root partition in read-write mode later).

`append="`*`options`*`"` Includes *`options`* as kernel boot parameters.

`delay=`*`num`* Displays the LILO prompt for *`num`* tenths of a second at boot time before booting the default kernel image.

`map=`*`map_file`* Stores the kernel's location in *`map_file`*. Don't delete this file.

`install=`*`file`* Specifies that *`file`* is the actual boot loader code that `lilo` writes to the boot sector.

`image=`*`file`* Defines a bootable kernel image with *`file`*. You should always use the `label` parameter directly following this definition to name the image.

`label=`*`name`* Uses *`name`* to label the current boot entry.

`other=`*`partition`* Defines another partition that contains a boot sector; analogous to `image` for other operating systems. Use `label` to define a label for the boot sector.

`linear` Remaps the boot sector load references when the BIOS disk geometry is different than the geometry that the Linux kernel sees. This is usually not necessary, because most modern BIOS code can recognize very large disks.

10.5.4 Initial RAM Disks

An *initial RAM disk* is a temporary root partition that the kernel should mount before doing anything else. Red Hat Linux uses initial RAM disks to support SCSI host controller drivers and other drivers that are compiled as modules.

NOTE *You do* not *need an initial RAM disk if you compile all the drivers necessary to mount your root filesystem directly into your kernel. The overwhelming majority of systems do not need an initial RAM disk.*

To install an initial RAM disk on a Red Hat Linux system, follow these steps:

1. Build your kernel and install the modules. Do not run any boot loader configuration or reboot just yet.
2. Run this command to create a RAM disk image (where *version* is your kernel version):

   ```
   mkinitrd /boot/initrd-version   version
   ```

3. If your boot loader is LILO, add this line to your new kernel's section in `lilo.conf` and run `lilo`:

   ```
   initrd=/boot/initrd-version
   ```

4. For GRUB, add the following to the appropriate kernel section:

   ```
   initrd /boot/initrd-version
   ```

10.6 Testing the Kernel

When you boot your new kernel for the first time, you should take a careful look at the kernel diagnostic messages to make sure that all of your hardware shows up and that the drivers are doing what they are supposed to do. Unfortunately, kernel messages tend to fly by so quickly that you can't see them. You can run `dmesg` to see the most recent messages, but to see everything, you need to look at your log files.

Most `/etc/syslog.conf` files send kernel messages to a file such as `/var/log/kernel.log`. If you don't see it anywhere, add a line like this to your `/etc/syslog.conf`:

```
kern.*                          /var/log/kernel.log
```

Then run this command:

```
kill -HUP `/var/run/syslogd.pid`
```

You may wish to make a checklist for your new kernel to make sure that your system still operates as it should. Here's an example:

- Do the network interface and network firewalls work?
- Are all of your disk partitions still visible?
- Does the kernel see your serial, parallel, and USB ports?
- Does all of your external hardware work?
- Does the X Window System work?

10.7 Boot Floppies

You can also boot the kernel from a floppy disk. Creating a boot floppy can be useful when recovering a Linux system that has no active kernel.

To create a boot floppy, put a freshly formatted floppy disk in the drive, go to the kernel source directory, and run this command:

```
make bzdisk
```

Of course, this only works if the size of your `bzImage` is smaller than the floppy disk capacity.

You may also need to run `rdev` to set the floppy's default root device. For example, if your root partition is `/dev/hda1`, use this command:

```
rdev /dev/fd0 /dev/hda1
```

Again, boot floppies are primarily useful during a system recovery when there is no active kernel. For testing new kernels, it's far better just to use an advanced boot loader such as GRUB.

10.8 Working with Loadable Kernel Modules

Loadable modules are little pieces of kernel code that you can load and unload into the kernel memory space while the kernel is running.

The `make modules_install` command (discussed in the "Installing Modules" section) installs the kernel module object files in `/lib/modules/`*`version`*, where *`version`* is your kernel version number. Module object filenames end with `.ko` in kernel versions 2.6.0 and later, and `.o` in older releases.

All distributions use modules in some capacity. If you would like to see the modules currently loaded on your system, run this command:

```
lsmod
```

The output should look something like this:

```
Module                  Size  Used by
es1370                 24768  0 (autoclean)
appletalk              19696  13 (autoclean)
```

This output includes `es1370` (a sound card driver) and `appletalk` (a network protocol driver). `autoclean` means that the kernel may try to automatically unload the module if it is not used for some time.

To load a module, use the `modprobe` command:

```
modprobe module
```

To remove a single module, use the -r option:

```
modprobe -r module
```

As mentioned earlier, you can find module dependencies in /lib/modules/*version*/modules.dep. Dependencies don't arrive on your system by magic; you must build an explicit list (the kernel module install process usually does this for you). You may need to create module dependencies by hand for all installed kernel versions after installing a module that doesn't come with the kernel. You can do this by running this command:

```
depmod -a
```

However, this doesn't always work, because depmod may try to read the symbol function memory address locations in the currently running kernel. This won't work if you're trying to build dependencies for kernels other than the one you're running. To zero in on a particular kernel version, find a System.map file that corresponds to that kernel, and then run this command:

```
depmod -a -F System.map version
```

As mentioned earlier, though, you do not need to run depmod under normal circumstances, because make modules_install runs it for you.

HINT *If you can't find the module that corresponds to a particular feature, go to the feature in the kernel configuration menu and press ? to get the help screen. This usually displays the module name.*

10.8.1 Kernel Module Loader

It's inconvenient to manually load a module every time you need to use a particular feature or driver. For example, if you compiled a certain filesystem as a module, it would be too much work to run a modprobe before a mount command that referenced the filesystem.

The Linux kernel provides an automatic module loader that can load most modules without additional modprobe or insmod commands. To use the loader, enable the **Kernel module loader** option in the kernel configuration. A kernel with the module loader runs modprobe to load modules that it wants.

There are limits to what the module loader can do without additional configuration. In general, it can load most modules that do not involve specific devices. For example, the module loader can load IP tables modules, and it can load filesystem modules as long as you specify the filesystem type with a mount command.

The module loader cannot guess your hardware. For instance, the module loader will not try to figure out what kind of Ethernet card is in your system. Therefore, you need to provide extra hints with the modprobe.conf file.

10.8.2 modprobe Configuration

The modprobe program reads /etc/modprobe.conf for important device information. Most entries are aliases such as this:

```
alias eth0 tulip
```

Here, the kernel loads the tulip module if you configure the eth0 network interface. In other cases, you may need to specify drivers by their major device numbers, such as this entry for an Adaptec host controller:

```
alias block-major-8 aic7xxx
```

Wildcards are also possible in modprobe.conf aliases. For example, if all of your Ethernet interface cards use the tulip driver, you can use this line to catch all interfaces:

```
alias eth* tulip
```

NOTE *The module utilities discussed here are the ones that go with kernel version 2.6.0. These programs are part of the* module-init-utils *package. Earlier kernel versions used the* modutils *package. The most significant difference between the two sets of utilities is that the new package reads* modprobe.conf *instead of* modules.conf. *The syntax in both files is very similar.*

Chaining Modules and the install Keyword

You can chain modules together with the install keyword. For example, if SCSI disk support isn't compiled directly into the kernel, you can force it, as in this example for /dev/sda on the Adaptec host controller from the preceding section:

```
alias block-major-8 my_scsi_modules
install my_scsi_modules /sbin/modprobe sd_mod; /sbin/modprobe aic7xxx
```

This works as follows:

1. A process (or the kernel) tries to access /dev/sda. Assume that this device is not yet mapped to an actual device.
2. The kernel sees that /dev/sda isn't mapped to its device, which has a block major number of 8. Therefore, the kernel runs this command:

   ```
   modprobe block-major-8
   ```

3. modprobe searches though /etc/modprobe.conf for block-major-8 and finds the alias line.

4. The alias line says to look for my_scsi_modules, so modprobe runs itself, this time as follows:

```
modprobe my_scsi_modules
```

5. The new modprobe sees install my_scsi_modules in modprobe.conf, and then runs the command(s) that follow in the file. In this case, these commands are two additional modprobe commands.

You can include any command that you like in an install line. If you need to debug something or just want to experiment, try adding an echo command.

NOTE *There is a* remove *keyword that works like* install *but runs its command when you remove a module.*

Module Options

Kernel modules can take various parameters with the options keyword, as shown in this example for a SoundBlaster 16:

```
alias snd-card-0 snd-sb16
options snd-sb16 port=0x220 irq=5 dma8=1 dma16=5 mpu_port=0x330
```