

12 Drivers and the Kernel



The kernel is the part of the system that's responsible for hiding the system's hardware underneath an abstract, high-level programming interface. It provides many of the facilities that users and user-level programs take for granted. For example, the kernel assembles all of the following concepts from lower-level hardware features:

- Processes (time sharing, protected address spaces)
- Signals and semaphores
- Virtual memory (swapping, paging, mapping)
- The filesystem (files, directories, namespace)
- Interprocess communication (pipes and network connections)

The kernel contains device drivers that manage its interaction with specific pieces of hardware; the rest of the kernel is, to a large degree, device independent. The relationship between the kernel and its device drivers is similar to the relationship between user-level processes and the kernel. When a process asks the kernel to “Read the first 64 bytes of `/etc/passwd`,” the kernel might translate this request into a device driver instruction such as “Fetch block 3,348 from device 3.” The driver would further break this command down into sequences of bit patterns to be presented to the device's control registers.

The kernel is written mostly in C, with a sprinkling of assembly language to help it interface with hardware- or chip-specific functions that are not accessible through normal compiler directives.

One of the advantages of the Linux environment is that the availability of source code makes it easy, relatively speaking, to roll your own device drivers and kernel modules

from scratch. In the early days of Linux, having skills in this area was a necessity because it was difficult to effectively administer Linux systems without being able to “mold” the system to a specific environment.

Today, Linux is so pervasive that sysadmins can be perfectly effective without ever getting their hands soiled with gooey kernel code. In fact, one might argue that such activities are better left to programmers and that administrators should focus more on the overall needs of the user community. System administrators can tune the kernel or add preexisting modules as described in this chapter, but they don’t need to take a crash course in C or 80x86 assembly language programming to survive.

The bottom line is that you shouldn’t confuse the administration of modern Linux environments with the frontier husbandry of just a few years back.

12.1 KERNEL ADAPTATION

Linux systems live in a world that could potentially include any of tens of thousands of different pieces of computer hardware. The kernel must adapt to whatever hardware is present in the machine on which it’s running.

A kernel can learn about the system’s hardware in a variety of ways. The most basic is for you to provide the kernel with explicit information about the hardware it should expect to find (or pretend not to find, as the case may be). Some kernels can also prospect for devices on their own, either at boot time or dynamically once the system is running.

On the PC platform, where Linux is popular, the challenge of creating an accurate inventory of the system’s hardware is particularly difficult (and sometimes impossible). PC hardware has followed an evolutionary path not unlike our own, in which early protozoa have now given rise to everything from dingos to killer bees. This diversity is compounded by the fact that PC manufacturers usually don’t give you much technical information about the systems they sell, so you must often take your system apart and visually inspect the pieces to answer questions such as “What chipset does my Ethernet card use?”

Most modern Linux systems survive on a hybrid diet of static and dynamic kernel components, with the mix between the two being dictated primarily by the limitations of PC hardware. It’s likely that at some point during your system administration career you’ll need to provide a helping hand in the form of building a new kernel configuration.

12.2 WHY CONFIGURE THE KERNEL?

When the system is installed, it comes with a generic configuration that’s designed to run most any application on most any hardware. The generic configuration includes many different device drivers and option packages, and it has tunable parameter

values chosen for “general purpose” use. By carefully examining this configuration and adjusting it to your exact needs, you may be able to enhance your system’s performance, security, or even reliability.

Modern Linux kernels are better than their ancestors at flushing unwanted drivers from memory, but compiled-in options will always be turned on. Although reconfiguring the kernel for efficiency is less important than it used to be, a good case can still be made for doing so.

Instructions for adding a new driver start on page 222.

Another reason to reconfigure the kernel is to add support for new types of devices (i.e., to add new device drivers). The driver code can’t just be mooshed onto the kernel like a gob of Play-Doh; it has to be integrated into the kernel’s data structures and tables. On some systems, this procedure may require that you go back to the configuration files for the kernel and add in the new device, rebuilding the kernel from scratch. On other systems, you may only need to run a program designed to make these configuration changes for you.

The kernel is not difficult to configure; it’s just difficult to fix once you break it.

12.3 CONFIGURATION METHODS

Four basic methods can be used to configure the Linux kernel. Chances are you’ll have the opportunity to try all of them eventually. The methods are:

- Modifying tunable (dynamic) kernel configuration parameters
- Loading new drivers and modules on the fly into an existing kernel
- Building a kernel from scratch (really, this means compiling it from source files, possibly with modifications and additions)
- Providing operational directives at boot time through the kernel loader, LILO, or GRUB. See page 23 for more information about these systems.

These methods are each applicable in slightly different situations. Modifying tunable parameters is the easiest and most common, whereas building a kernel from source files is the hardest and least often required. Fortunately, all of these approaches become second nature with a little practice.

12.4 TUNING A LINUX KERNEL

Many modules and drivers in the kernel were designed with the knowledge that one size doesn’t fit all. To increase flexibility, special hooks allow parameters such as an internal table’s size or the kernel’s behavior in a particular circumstance to be adjusted on the fly by the system administrator. These hooks are accessible through an extensive kernel-to-userland interface represented by files in the **/proc** filesystem. In many cases, a large user-level application (especially an “infrastructure” application such as a database) will require you to adjust parameters to accommodate its needs.

Special files in **/proc/sys** let you view and set kernel options at run time. These files mimic standard Linux files, but they are really back doors into the kernel. If one of these files has a value you would like to change, you can try writing to it. Unfortunately, not all of the files can be written to (regardless of their apparent permissions), and no documentation tells you which ones can and cannot be written.

For example, to change the maximum number of open files a process can have, try something like

```
# echo 32768 > /proc/sys/fs/file-max
```

Once you get used to this unorthodox interface, you'll find it quite useful, especially for changing configuration options. A word of caution, however: changes are not remembered across reboots. If you want to make permanent changes, you should use the **/etc/sysctl.conf** file documented below. Table 12.1 lists some useful options.

Table 12.1 Files in /proc/sys for some sample tunable kernel parameters

Dir ^a	File	Default	Function
F	file-max	4096	Sets max # of open files per process
F	inode-max	16384	Sets max # of open inodes per process
N	ip_forward	0	Allows IP forwarding when set to 1
N	icmp_echo_ignore_all	0	Ignores ICMP pings when set to 1
N	icmp_echo_ignore_broadcasts	0	Ignores broadcast pings when set to 1
K	shmmax	32M	Sets max shared memory size
K	ctrl-alt-del	0	Reboot on Ctrl-Alt-Delete sequence?
C	autoeject	0	Auto-eject CD-ROM on dismount?

a. F = **/proc/sys/fs**, N = **/proc/sys/net/ipv4**, K = **/proc/sys/kernel**, C = **/proc/sys/dev/cdrom**

See page 284 in the TCP chapter to learn about additional network parameters that are tunable.

A less kludgy way to modify these same parameters can be found on most systems in the form of the **sysctl** command. **sysctl** can set individual variables either from the command line or by reading a list of *variable=value* pairs from a file. By default, the file **/etc/sysctl.conf** is read at boot time and its contents used to set initial (custom) parameter values.

For example, the command

```
# sysctl net.ipv4.ip_forward=0
```

turns off IP forwarding. Note that you form the variable names used by **sysctl** by replacing the */*'s in the **/proc/sys** directory structure with dots.

12.5 ADDING DEVICE DRIVERS

A device driver is a program that manages the system's interaction with a piece of hardware. The driver translates between the hardware commands understood by

the device and the stylized programming interface used by the kernel. The existence of the driver layer helps keep Linux reasonably device independent.

Device drivers are part of the kernel; they are not user processes. However, a driver can be accessed both from within the kernel and from user space. User-level access to devices is usually provided through special device files that live in the `/dev` directory. The kernel transforms operations on these files into calls to the code of the driver.

The PC platform has been a source of chaos in the system administrator's world. A dizzying array of hardware and "standards" with varying levels of operating system support are available. Behold:

- Over 30 different SCSI chipsets are supported by Linux, and each is packaged and sold by at least twice that many vendors.
- Over 200 different network interfaces are out there, each being marketed by several different vendors under different names.
- Newer, better, cheaper types of hardware are being developed all the time. Each will require a device driver in order to work with your Linux of choice.

With the remarkable pace at which new hardware is being developed, it is practically impossible to keep the mainline OS distributions up to date with the latest hardware. It is not at all uncommon to have to add a device driver to your kernel to support a new piece of hardware.

Only device drivers designed for use with Linux (and usually, a specific version of the Linux kernel) can be successfully installed on a Linux system. Drivers for other operating systems (like Windows) will not work, so you will need to purchase new hardware with this in mind. In addition, devices vary in their degree of compatibility and functionality when used with Linux, so it's wise to pay some attention to the results other sites have obtained with any hardware you are considering.

Vendors are becoming more aware of the Linux market, and they even provide Linux drivers on occasion. You may be lucky and find that your vendor will furnish you with both drivers and installation instructions. More likely, you will only find the driver you need on some uncommented web page. In either case, this section shows you what is really going on when you add a device driver.

Device numbers

Many devices have a corresponding file in `/dev`, the notable exceptions on modern operating systems being network devices. By virtue of being device files, the files in `/dev` each have a major and minor device number associated with them. The kernel uses these numbers to map references to a device file to the corresponding driver.

The major device number identifies the driver with which the file is associated (in other words, the type of device). The minor device number usually identifies which particular instance of a given device type is to be addressed. The minor device number is sometimes called the unit number.

You can see the major and minor number of a device file with **ls -l**:

```
% ls -l /dev/sda
brw-rw---- 1 root  disk   8,  0 Mar  3 1999 /dev/sda
```

This example shows the first SCSI disk on a Linux system. It has a major number of 8 and a minor number of 0.

The minor device number is sometimes used by the driver to select the particular characteristic of a device. For example, a single tape drive can have several files in **/dev** representing it in various configurations of recording density and rewind characteristics. In essence, the driver is free to interpret the minor device number in whatever way it wants. Look up the man page for the driver to determine what convention it's using.

There are actually two types of device files: block device files and character device files. A block device is read or written one block (a group of bytes, usually a multiple of 512) at a time; a character device can be read or written one byte at a time. Some devices support access through both block and character device files, although this is extremely rare under Linux.

It is sometimes convenient to implement an abstraction as a device driver even when it controls no actual device. Such phantom devices are known as pseudo-devices. For example, a user who logs in over the network is assigned a PTY (pseudo-TTY) that looks, feels, and smells like a serial port from the perspective of high-level software. This trick allows programs written in the days when everyone used a TTY to continue to function in the world of windows and networks.

When a program performs an operation on a device file, the kernel automatically catches the reference, looks up the appropriate function name in a table, and transfers control to it. To perform an unusual operation that doesn't have a direct analog in the filesystem model (for example, ejecting a floppy disk), a program can use the **ioctl** system call to pass a message directly from user space into the driver.

12.6 ADDING A LINUX DEVICE DRIVER

On Linux systems, device drivers are typically distributed in one of three forms:

- A patch against a specific kernel version
- A loadable module
- An installation script that applies appropriate patches

The most common of all these is the patch against a specific kernel version. These patches can in most cases be applied with the following procedure:¹

```
# cd /usr/src/linux ; patch -p1 < patch_file
```

Diffs made against a different minor version of the kernel may fail, but the driver should still work. Here, we cover how to manually add a network “snarf” driver to the

1. Of course, the kernel source package must be installed before you can modify the kernel tree.

kernel. It's a very complicated and tedious process, especially when compared to other operating systems we've seen.

By convention, Linux kernel source resides in `/usr/src/linux`. Within the **drivers** subdirectory, you'll need to find the subdirectory that corresponds to the type of device you have. A directory listing of **drivers** looks like this:

```
% ls -F /usr/src/linux/drivers
Makefile  cdrom/  i2o/      nubus/    sbus/     telephony/
acorn/    char/   isdn/     parport/  scsi/     usb/
ap1000/   dio/    macintosh/ pci/       sgi/      video/
atm/      fc4/    misc/     pcmcia/   sound/    zorro/
block/    i2c/    net/      pnp/      tc/
```

The most common directories to which drivers are added are **block**, **char**, **net**, **usb**, **sound**, and **scsi**. These directories contain drivers for block devices (such as IDE disk drives), character devices (such as serial ports), network devices, USB devices, sound cards, and SCSI cards, respectively. Some of the other directories contain drivers for the buses themselves (e.g., **pci**, **nubus**, and **zorro**); it's unlikely that you will need to add drivers to these directories. Some directories contain platform-specific drivers, such as **macintosh**, **acorn**, and **ap1000**. Some directories contain specialty devices such as **atm**, **isdn**, and **telephony**.

Since our example device is a network-related device, we will add the driver to the directory **drivers/net**. We'll need to modify the following files:

- **drivers/net/Makefile**, so that our driver will be compiled
- **drivers/net/Config.in**, so that our device will appear in the config options
- **drivers/net/Space.c**, so that the device will be probed on startup

After putting the `.c` and `.h` files for the driver in **drivers/net**, we'll add the driver to **drivers/net/Makefile**. The lines we'd add (near the end of the file) follow.

```
ifeq ($(CONFIG_SNARF),y)
    L_OBJS += snarf.o
else
    ifeq ($(CONFIG_SNARF),m)
        M_OBJS += snarf.o
    endif
endif
```

This configuration adds the `snarf` driver so that it can be either configured as a module or built into the kernel.

After adding the device to the **Makefile**, we have to make sure we can configure the device when we configure the kernel. All network devices need to be listed in the file **drivers/net/Config.in**. To add the device so that it can be built either as a module or as part of the kernel (consistent with what we claimed in the **Makefile**), we add the following line:

```
tristate 'Snarf device support' CONFIG_SNARF
```

The `tristate` keyword means you can build the device as a module. If the device cannot be built as a module, use the keyword `bool` instead of `tristate`. The next token is the string to display on the configuration screen. It can be any arbitrary text, but it should identify the device that is being configured. The final token is the configuration macro. This token needs to be the same as that tested for with the `ifeq` clause in the **Makefile**.

The last file we need to edit to add our device to the system is **drivers/net/Space.c**. **Space.c** contains references to the probe routines for the device driver, and it also controls the device probe order. Here, we'll have to edit the file in two different places. First we'll add a reference to the probe function, then we'll add the device to the list of devices to probe for.

At the top of the **Space.c** file are a bunch of references to other probe functions. We'll add the following line to that list:

```
extern int snarf_probe(struct device *);
```

Next, to add the device to the actual probe list, we need to determine which list to add it to. A separate probe list is kept for each type of bus (PCI, EISA, SBus, MCA, ISA, parallel port, etc.). The `snarf` device is a PCI device, so we'll add it to the list called `pci_probes`. The line that says

```
struct devprobe pci_probes[] __initdata = {
```

is followed by an ordered list of devices. The devices higher up in the list are probed first. Probe order does not usually matter for PCI devices, but some devices are sensitive. Just to be sure the `snarf` device is detected, we'll add it to the top of the list:

```
struct devprobe pci_probes[] __initdata = {
#ifdef CONFIG_SNARF
    snarf_probe, 0},
#endif
```

The device has now been added to the Linux kernel. When we next configure the kernel, the device should appear as a configuration option under "network devices."

12.7 DEVICE FILES

By convention, device files are kept in the `/dev` directory. Large systems, especially those with networking and pseudo-terminals, may support hundreds of devices.

Device files are created with the **mknod** command, which has the syntax

```
mknod filename type major minor
```

where *filename* is the device file to be created, *type* is **c** for a character device or **b** for a block device, and *major* and *minor* are the major and minor device numbers. If you are creating a device file that refers to a driver that's already present in your kernel, check the man page for the driver to find the appropriate major and minor device numbers.

Official Linux device number assignments can be found at www.lanana.org.



Red Hat and Debian provide a script called **MAKEDEV** (in **/dev**) to automatically supply default values to **mknod**. Study the script to find the arguments needed for your device. For example, to make PTY entries, you'd use the following commands:

```
# cd /dev
# ./MAKEDEV pty
```

Naming conventions for devices

Naming conventions for devices are somewhat random. They are often holdovers from the way things were done under UNIX on a DEC PDP-11, as archaic as that may sound in this day and age.

See Chapter 7 for more information about serial ports.

Serial device files are named **ttyS** followed by a number that identifies the specific interface to which the port is attached. TTYs are sometimes represented by more than one device file; the extra files usually provide access to alternative flow control methods or locking protocols.

The names of tape devices often include not only a reference to the drive itself but also an indication of whether the device rewinds after each tape operation and the density at which it reads and writes.

IDE hard disk devices are named **/dev/hdLP**, where **L** is a letter that identifies the unit (with **a** being the master on the first IDE interface, **b** being the slave on that interface, **c** being the master on the second IDE interface, etc.) and **P** is the partition number (starting with 1). For example, the first partition on the first IDE disk is typically **/dev/hda1**. SCSI disks are named similarly, but with the prefix **/dev/sd** instead of **/dev/hd**. You can drop the partition number on both types of devices to access the entire disk (e.g., **/dev/hda**).

SCSI CD-ROM drives are referred to by the files **/dev/scdN**, where **N** is a number that distinguishes multiple CD-ROM drives. Modern IDE (ATAPI) CD-ROM drives are referred to just like IDE hard disks (e.g., **/dev/hdc**).

12.8 LOADABLE KERNEL MODULES

Loadable kernel module (LKM) support allows a device driver—or any other kernel service—to be linked into and removed from the kernel while it is running. This facility makes the installation of drivers much easier, since the kernel binary does not need to be changed. It also allows the kernel to be smaller because drivers are not loaded unless they are needed.

Loadable modules are implemented by one or more documented “hooks” into the kernel that additional device drivers can grab onto. A user-level command communicates with the kernel and tells it to load new modules into memory. There is usually a command that unloads drivers as well.

Although loadable drivers are convenient, they are not entirely safe. Any time you load or unload a module, you risk causing a kernel panic. We don't recommend loading or unloading an untested module when you are not willing to crash the machine.

Under Linux, almost anything can be built as a loadable kernel module. The exceptions are the root filesystem type, the device on which the root filesystem resides, and the PS/2 mouse driver.

Loadable kernel modules are conventionally stored under `/lib/modules/version`, where *version* is the version of your Linux kernel as returned by `uname -r`. You can inspect the currently loaded modules with the `lsmod` command:

```
# lsmod
Module          Size  Used by
ppp             21452  0
slhc            4236  0 [ppp]
ds              6344  1
i82365         26648  1
pcmcia_core    37024  0 [ds i82365]
```

This machine has the PCMCIA controller modules, the PPP driver, and the PPP header compression modules loaded.

Linux LKMs can be manually loaded into the kernel with `insmod`. For example, we could manually insert our example `snarf` module with the command

```
# insmod /path/to/snarf.o
```

Parameters can also be passed to loadable kernel modules; for example,

```
# insmod /path/to/snarf.o io=0xXXX irq=X
```

Once a loadable kernel module has been manually inserted into the kernel, it will only be removed if you explicitly request its removal or if the system is rebooted. We could use `rmmod snarf` to remove our `snarf` module.

You can use `rmmod` at any time, but it works only if the number of current references to the module (listed in the `Used by` column of `lsmod`'s output) is 0.

You can also load Linux LKMs semiautomatically with `modprobe`, a wrapper for `insmod` that understands dependencies, options, and installation and removal procedures. `modprobe` uses the `/etc/modules.conf` file to figure out how to handle each module.

You can dynamically generate an `/etc/modules.conf` file that corresponds to all your currently installed modules by running `modprobe -c`. This command generates a long file that looks like this:

```
#This file was generated by: modprobe -c (2.1.121)
path[pcmcia]=/lib/modules/preferred
path[pcmcia]=/lib/modules/default
path[pcmcia]=/lib/modules/2.3.39
path[misc]=/lib/modules/2.3.39
...
# Aliases
alias block-major-1 rd
alias block-major-2 floppy
```

```

...
alias char-major-4 serial
alias char-major-5 serial
alias char-major-6 lp
...
alias dos msdos
alias plip0 plip
alias ppp0 ppp
options ne io=x0340 irq=9

```

The path statements tell where a particular module can be found. You can modify or add entries of this type if you want to keep your modules in a nonstandard location.

The alias statement provides a mapping between block major device numbers, character major device numbers, filesystems, network devices, and network protocols and their corresponding module names.

The options lines are not dynamically generated. They specify options that should be passed to a module when it is loaded. For example, we could use the following line to tell the `snarf` module its proper I/O address and interrupt vector:²

```
options snarf io=0xXXX irq=X
```

modprobe also understands the statements `pre-install`, `post-install`, `pre-remove`, `post-remove`, `install`, and `remove`. These statements allow commands to be executed when a specific module is inserted into or removed from the running kernel. They take the following form

```

pre-install module command ...
install module command ...
post-install module command ...
pre-remove module command ...
remove module command ...
post-remove module command ...

```

and are run before insertion, simultaneously with insertion (if possible), after insertion, before removal, during removal (if possible), and after removal.



Debian systems execute `/etc/cron.daily/modutils` once a day to shoo away any modules that have overstayed their welcome in the kernel.

12.9 BUILDING A LINUX KERNEL

Because Linux is evolving so rapidly, it is much more likely that you'll be faced with the need to build a Linux kernel than you would if you were running a big-iron operating system. Kernel patches, device drivers, and new functionality are constantly becoming available. This is really something of a mixed blessing. On one hand, it's convenient to always have support for the "latest and greatest," but on the other hand

2. If you're using PC hardware, it can be a challenge to create a configuration in which device interrupt request vectors (IRQs) and I/O ports do not overlap. You can view the current assignments on your system by examining the contents of `/proc/interrupts` and `/proc/ioports`, respectively.

it can become quite time consuming to keep up with the constant flow of new material. But after you successfully build a kernel once, you'll feel empowered and eager to do it again.

It's less likely that you'll need to build a kernel on your own if you're running a "stable" version. Linux has adopted a versioning scheme in which the second part of the version number indicates whether the kernel is stable (even numbers) or in development (odd numbers). For example, kernel version 2.4.3 would be a "stable" kernel, whereas 2.5.3 would be a "development" kernel. Now you know.

Linux kernel configuration has come a long way, but it still feels very primitive compared to the procedures used on some other systems. The process revolves around the **.config** file in the root of the kernel source directory (usually **/usr/src/linux**).³ All of the kernel configuration information is specified in this file, but its format is somewhat cryptic. Use the decoding guide in **Documentation/Configure.help** to find out what the various options mean.

To save folks from having to edit the **.config** file directly, Linux has several **make** targets that let you configure the kernel with different interfaces. If you are running X Windows, the prettiest configuration interface is provided by **make xconfig**. This command brings up a graphical configuration screen on which you can pick the devices to add to your kernel (or compile as loadable modules).

If you are not running X, you can use a **curses**-based⁴ alternative invoked with **make menuconfig**. Finally, there is the older style **make config**, which prompts you to respond to every single configuration option available without letting you later go back and change your mind.

We recommend **make xconfig** if you are running X and **make menuconfig** if you aren't. Avoid **make config**.

These tools are straightforward as far as the options you can turn on, but unfortunately they are painful to use if you want to maintain several versions of the kernel for multiple architectures or hardware configurations.

The various configuration interfaces described above all generate a **.config** file that looks something like this:

```
# Automatically generated make config: don't edit
#
# Code maturity level options

CONFIG_EXPERIMENTAL=y
#
# Processor type and features
#
```

3. Again, you will need to install the kernel source package before you can build a kernel on your system; see page 733 for tips on package installation.
4. **curses** is a library from the days of yore used to create text-based GUIs that run in a terminal window.

```
# CONFIG_M386 is not set
# CONFIG_M486 is not set
# CONFIG_M586 is not set
# CONFIG_M586TSC is not set
CONFIG_M686=y
CONFIG_X86_WP_WORKS_OK=y
CONFIG_X86_INVLPG=y
CONFIG_X86_BSWAP=y
CONFIG_X86_POPAD_OK=y
CONFIG_X86_TSC=y
CONFIG_X86_GOOD_APIC=y
...
```

As you can see, the contents are rather cryptic and provide no descriptions of what the CONFIG tags mean. Sometimes you can figure out the meaning. Basically, each CONFIG line refers to a specific kernel configuration option. The value `y` compiles the option into the kernel; `m` enables it, but as a loadable module.

Some things can be configured as modules and some can't. You just have to know which is which; it's not clear from the `.config` file. There is also no easy mapping of the CONFIG tags to meaningful information. However, you can usually extract this information from the `Config.in` file located in each driver directory. The `Config.in` files are difficult and inconvenient to track down, so it's best to just use `make xconfig` or `make menuconfig`.

Once you have a working, running kernel, you may need to pass special configuration options to it at boot time, such as the root device it should use or an instruction to probe for multiple Ethernet cards. The boot loader (typically LILO or GRUB) passes in these options. You add static configuration options to the `/etc/lilo.conf` or `/boot/grub/grub.conf` file, depending on which boot loader you use. See page 23 for more information.

If it's not possible to edit the boot loader configuration file (perhaps you broke something and the machine can't boot), you can pass the options in by hand. For example, at a LILO boot prompt you could type

```
LILO: linux root=/dev/hda1 ether=0,0,eth0 ether=0,0,eth1
```

to tell LILO to load the kernel specified by the "linux" tag, to use the root device `/dev/hda1`, and to probe for two Ethernet cards.

A similar example using GRUB would look like this:

```
grub> kernel /vmlinuz root=/dev/hda1 ether=0,0,eth0 ether=0,0,eth1
grub> boot
```

Building the Linux kernel binary

Setting up an appropriate `.config` file is the most important part of the Linux kernel configuration process, but you must jump through several more hoops to turn that file into a finished kernel.

Here's an outline of the entire process:

- **cd** to **/usr/src/linux**.
- Run **make xconfig** or **make menuconfig**.
- Run **make dep**.
- Run **make clean**.
- Run **make bzImage**.
- Run **make modules**.
- Run **make modules_install**.
- Copy **/usr/src/linux/arch/i386/boot/bzImage** to **/boot/vmlinuz**.
- Copy **/usr/src/linux/arch/i386/boot/System.map** to **/boot/System.map**.
- Edit **/etc/lilo.conf** (LILO) or **/boot/grub/grub.conf** (GRUB) to add a configuration line for the new kernel.
- If you're using LILO, run **/sbin/lilo** to install the reconfigured boot loader.

The **make clean** step is not always strictly necessary, but it is generally a good idea to start with a clean build environment. In practice, many problems can be traced back to skipping this step.

12.10 DON'T FIX IT IF IT AIN'T BROKEN

With new Linux kernel versions arriving on the scene every few months and new drivers and patches being released every day, it's easy to become addicted to patching and upgrades. After all, what's more exciting than telling your user community that you just found a new kernel patch and that you'll be taking the mail server down for the afternoon to install it? Some administrators justify their existence this way; everybody likes to be the hero.

A good system administrator carefully weighs needs and risks when planning kernel upgrades and patches. Sure, the new release may be the latest and greatest, but is it as stable as the current version? Could the upgrade or patch be delayed and installed with another group of patches at the end of the month? It's important to resist the temptation to let "keeping up with the joneses" (in this case, the kernel hacking community) dominate the best interests of your user community.

A good rule of thumb is to upgrade or apply patches only when the productivity gains you expect to obtain (usually measured in terms of reliability and performance) will exceed the effort and lost time required to perform the installation. If you're having trouble quantifying the specific gain, that's a good sign that the patch can wait for another day.

12.11 RECOMMENDED READING

BECK, MICHAEL, ET AL. *Linux Kernel Internals, 2nd Edition*. Reading, MA: Addison-Wesley. 1998.

This is an older Linux kernel book, but it's still our favorite. Although it's a bit outdated, it gives a good explanation of the kernel's inner workings.

12.12 EXERCISES

- E12.1** Describe what the kernel does. Explain the difference between a kernel that uses modules and one that doesn't.
- E12.2** For each configuration method listed on page 221, give a practical example of a parameter that might need to be changed.
- ★ **E12.3** Examine the values of several parameters from Table 12.1 on page 222 using both the `/proc` method and the `sysctl` method. Change two of the values with one method and then read them back with the other method. Verify that the system's behavior has actually changed in response to your tuning. Turn in a typescript of your experiment. (Requires root access.)
- ★ **E12.4** At a local flea market, you get a great deal on a laptop card that gives you Ethernet connectivity through the parallel port. What steps would you need to perform to make Linux recognize this new card? Should you compile support directly into the kernel or add it as a module? Why? (Bonus question: if your hourly consulting fee is \$80, estimate the value of the labor needed to get this cheapie Ethernet interface working.)
- ★ **E12.5** A new release of the Linux kernel just came out, and you want to upgrade all the machines in the student lab (about 50 machines, not all identical). What issues should you consider? What procedure should you follow? What problems might occur, and how would you deal with them?
- ★★ **E12.6** In the lab, configure a kernel with `xconfig` or `menuconfig` and build a kernel binary. Install and run the new system. Turn in `dmesg` output from the old and new kernels and highlight the differences. (Requires root access.)