1

Introduction to the Linux Kernel

This chapter introduces the Linux kernel and Linux operating system, placing them in the historical context of Unix. Today, Unix is a family of operating systems implementing a similar application programming interface (API) and built around shared design decisions. But Unix is also a specific operating system, first built more than 40 years ago. To understand Linux, we must first discuss the first Unix system.

History of Unix

After four decades of use, computer scientists continue to regard the Unix operating system as one of the most powerful and elegant systems in existence. Since the creation of Unix in 1969, the brainchild of Dennis Ritchie and Ken Thompson has become a creature of legends, a system whose design has withstood the test of time with few bruises to its name.

Unix grew out of Multics, a failed multiuser operating system project in which Bell Laboratories was involved. With the Multics project terminated, members of Bell Laboratories' Computer Sciences Research Center found themselves without a capable interactive operating system. In the summer of 1969, Bell Lab programmers sketched out a filesystem design that ultimately evolved into Unix. Testing its design, Thompson implemented the new system on an otherwise-idle PDP-7. In 1971, Unix was ported to the PDP-11, and in 1973, the operating system was rewritten in C—an unprecedented step at the time, but one that paved the way for future portability. The first Unix widely used outside Bell Labs was Unix System, Sixth Edition, more commonly called V6.

Other companies ported Unix to new machines. Accompanying these ports were enhancements that resulted in several variants of the operating system. In 1977, Bell Labs released a combination of these variants into a single system, Unix System III; in 1982, AT&T released System V.[1]

---

[1] *What about System IV? It was an internal development version.*

The simplicity of Unix's design, coupled with the fact that it was distributed with source code, led to further development at external organizations. The most influential of these contributors was the University of California at Berkeley. Variants of Unix from Berkeley are known as Berkeley Software Distributions, or *BSD*. Berkeley's first release, 1BSD in 1977, was a collection of patches and additional software on top of Bell Labs' Unix. 2BSD in 1978 continued this trend, adding the `csh` and `vi` utilities, which persist on Unix systems to this day. The first standalone Berkeley Unix was 3BSD in 1979. It added virtual memory (VM) to an already impressive list of features. A series of 4BSD releases, 4.0BSD, 4.1BSD, 4.2BSD, 4.3BSD, followed 3BSD. These versions of Unix added job control, demand paging, and TCP/IP. In 1994, the university released the final official Berkeley Unix, featuring a rewritten VM subsystem, as 4.4BSD. Today, thanks to BSD's permissive license, development of BSD continues with the Darwin, FreeBSD, NetBSD, and OpenBSD systems.

In the 1980s and 1990s, multiple workstation and server companies introduced their own commercial versions of Unix. These systems were based on either an AT&T or a Berkeley release and supported high-end features developed for their particular hardware architecture. Among these systems were Digital's Tru64, Hewlett Packard's HP-UX, IBM's AIX, Sequent's DYNIX/ptx, SGI's IRIX, and Sun's Solaris & SunOS.

The original elegant design of the Unix system, along with the years of innovation and evolutionary improvement that followed, has resulted in a powerful, robust, and stable operating system. A handful of characteristics of Unix are at the core of its strength. First, Unix is simple: Whereas some operating systems implement thousands of system calls and have unclear design goals, Unix systems implement only hundreds of system calls and have a straightforward, even basic, design. Second, in Unix, *everything is a file*.[2] This simplifies the manipulation of data and devices into a set of core system calls: `open()`, `read()`, `write()`, `lseek()`, and `close()`. Third, the Unix kernel and related system utilities are written in C—a property that gives Unix its amazing portability to diverse hardware architectures and accessibility to a wide range of developers. Fourth, Unix has fast process creation time and the unique `fork()` system call. Finally, Unix provides simple yet robust interprocess communication (IPC) primitives that, when coupled with the fast process creation time, enable the creation of simple programs that *do one thing and do it well*. These single-purpose programs can be strung together to accomplish tasks of increasing complexity. Unix systems thus exhibit clean layering, with a strong separation between policy and mechanism.

Today, Unix is a modern operating system supporting preemptive multitasking, multithreading, virtual memory, demand paging, shared libraries with demand loading, and

[2] *Well, okay, not everything—but much is represented as a file. Sockets are a notable exception. Some recent efforts, such as Unix's successor at Bell Labs, Plan9, implement nearly all aspects of the system as a file.*

TCP/IP networking. Many Unix variants scale to hundreds of processors, whereas other Unix systems run on small, embedded devices. Although Unix is no longer a research project, Unix systems continue to benefit from advances in operating system design while remaining a practical and general-purpose operating system.

Unix owes its success to the simplicity and elegance of its design. Its strength today derives from the inaugural decisions that Dennis Ritchie, Ken Thompson, and other early developers made: choices that have endowed Unix with the capability to evolve without compromising itself.

Along Came Linus: Introduction to Linux

Linus Torvalds developed the first version of Linux in 1991 as an operating system for computers powered by the Intel 80386 microprocessor, which at the time was a new and advanced processor. Linus, then a student at the University of Helsinki, was perturbed by the lack of a powerful yet free Unix system. The reigning personal computer OS of the day, Microsoft's DOS, was useful to Torvalds for little other than playing *Prince of Persia*. Linus did use Minix, a low-cost Unix created as a teaching aid, but he was discouraged by the inability to easily make and distribute changes to the system's source code (because of Minix's license) and by design decisions made by Minix's author.

In response to his predicament, Linus did what any normal college student would do: He decided to write his own operating system. Linus began by writing a simple terminal emulator, which he used to connect to larger Unix systems at his school. Over the course of the academic year, his terminal emulator evolved and improved. Before long, Linus had an immature but full-fledged Unix on his hands. He posted an early release to the Internet in late 1991.

Use of Linux took off, with early Linux distributions quickly gaining many users. More important to its initial success, however, is that Linux quickly attracted many developers—hackers adding, changing, improving code. Because of the terms of its license, Linux swiftly evolved into a collaborative project developed by many.

Fast forward to the present. Today, Linux is a full-fledged operating system also running on Alpha, ARM, PowerPC, SPARC, x86-64 and many other architectures. It runs on systems as small as a watch to machines as large as room-filling super-computer clusters. Linux powers the smallest consumer electronics and the largest Datacenters. Today, commercial interest in Linux is strong. Both new Linux-specific corporations, such as Red Hat, and existing powerhouses, such as IBM, are providing Linux-based solutions for embedded, mobile, desktop, and server needs.

Linux is a Unix-like system, but it is not Unix. That is, although Linux borrows many ideas from Unix and implements the Unix API (as defined by POSIX and the Single Unix Specification), it is not a direct descendant of the Unix source code like other Unix systems. Where desired, it has deviated from the path taken by other implementations, but it has not forsaken the general design goals of Unix or broken standardized application interfaces.

One of Linux's most interesting features is that it is not a commercial product; instead, it is a collaborative project developed over the Internet. Although Linus remains the creator of Linux and the *maintainer* of the kernel, progress continues through a loose-knit group of developers. Anyone can contribute to Linux. The Linux kernel, as with much of the system, is *free* or *open source* software.[3] Specifically, the Linux kernel is licensed under the GNU General Public License (GPL) version 2.0. Consequently, you are free to download the source code and make any modifications you want. The only caveat is that if you distribute your changes, you must continue to provide the recipients with the same rights you enjoyed, including the availability of the source code.[4]

Linux is many things to many people. The basics of a Linux system are the kernel, C library, toolchain, and basic system utilities, such as a login process and shell. A Linux system can also include a modern X Window System implementation including a full-featured desktop environment, such as GNOME. Thousands of free and commercial applications exist for Linux. In this book, when I say *Linux* I typically mean the *Linux kernel*. Where it is ambiguous, I try explicitly to point out whether I am referring to *Linux* as a full system or just the kernel proper. Strictly speaking, the term *Linux* refers only to the kernel.

Overview of Operating Systems and Kernels

Because of the ever-growing feature set and ill design of some modern commercial operating systems, the notion of what precisely defines an operating system is not universal. Many users consider whatever they see on the screen to be the operating system. Technically speaking, and in this book, the *operating system* is considered the parts of the system responsible for basic use and administration. This includes the kernel and device drivers, boot loader, command shell or other user interface, and basic file and system utilities. It is the stuff you *need*—not a web browser or music players. The term *system*, in turn, refers to the operating system and all the applications running on top of it.

Of course, the topic of this book is the *kernel*. Whereas the user interface is the outermost portion of the operating system, the kernel is the innermost. It is the core internals; the software that provides basic services for all other parts of the system, manages hardware, and distributes system resources. The kernel is sometimes referred to as the *supervisor*, *core*, or *internals* of the operating system. Typical components of a kernel are interrupt handlers to service interrupt requests, a scheduler to share processor time among multiple processes, a memory management system to manage process address spaces, and system services such as networking and interprocess communication. On

---

[3] *I will leave the free versus open debate to you. See http://www.fsf.org and http://www.opensource.org.*

[4] *You should read the GNU GPL version 2.0. There is a copy in the file COPYING in your kernel source tree. You can also find it online at http://www.fsf.org. Note that the latest version of the GNU GPL is version 3.0; the kernel developers have decided to remain with version 2.0.*

modern systems with protected memory management units, the kernel typically resides in an elevated system state compared to normal user applications. This includes a protected memory space and full access to the hardware. This system state and memory space is collectively referred to as *kernel-space*. Conversely, user applications execute in *user-space*. They see a subset of the machine's available resources and can perform certain system functions, directly access hardware, access memory outside of that allotted them by the kernel, or otherwise misbehave. When executing kernel code, the system is in kernel-space executing in kernel mode. When running a regular process, the system is in user-space executing in user mode.

Applications running on the system communicate with the kernel via *system calls* (see Figure 1.1). An application typically calls functions in a library—for example, the *C library*—that in turn rely on the system call interface to instruct the kernel to carry out tasks on the application's behalf. Some library calls provide many features not found in the system call, and thus, calling into the kernel is just one step in an otherwise large function. For example, consider the familiar `printf()` function. It provides formatting and buffering of the data; only one step in its work is invoking `write()` to write the data to the console. Conversely, some library calls have a one-to-one relationship with the kernel. For example, the `open()` library function does little except call the `open()` system call. Still other C library functions, such as `strcpy()`, should (one hopes) make no direct use of the kernel at all. When an application executes a system call, we say that the *kernel is executing on behalf of the application*. Furthermore, the application is said to be *executing a system call in kernel-space,* and the kernel is running in *process context*. This relationship—that applications *call into* the kernel via the system call interface—is the fundamental manner in which applications get work done.

The kernel also manages the system's hardware. Nearly all architectures, including all systems that Linux supports, provide the concept of *interrupts*. When hardware wants to communicate with the system, it issues an interrupt that literally interrupts the processor, which in turn interrupts the kernel. A number identifies interrupts and the kernel uses this number to execute a specific *interrupt handler* to process and respond to the interrupt. For example, as you type, the keyboard controller issues an interrupt to let the system know that there is new data in the keyboard buffer. The kernel notes the interrupt number of the incoming interrupt and executes the correct interrupt handler. The interrupt handler processes the keyboard data and lets the keyboard controller know it is ready for more data. To provide synchronization, the kernel can disable interrupts—either all interrupts or just one specific interrupt number. In many operating systems, including Linux, the interrupt handlers do not run in a process context. Instead, they run in a special *interrupt context* that is not associated with any process. This special context exists solely to let an interrupt handler quickly respond to an interrupt, and then exit.

These contexts represent the breadth of the kernel's activities. In fact, in Linux, we can generalize that each processor is doing exactly one of three things at any given moment:

- In user-space, executing user code in a process
- In kernel-space, in process context, executing on behalf of a specific process

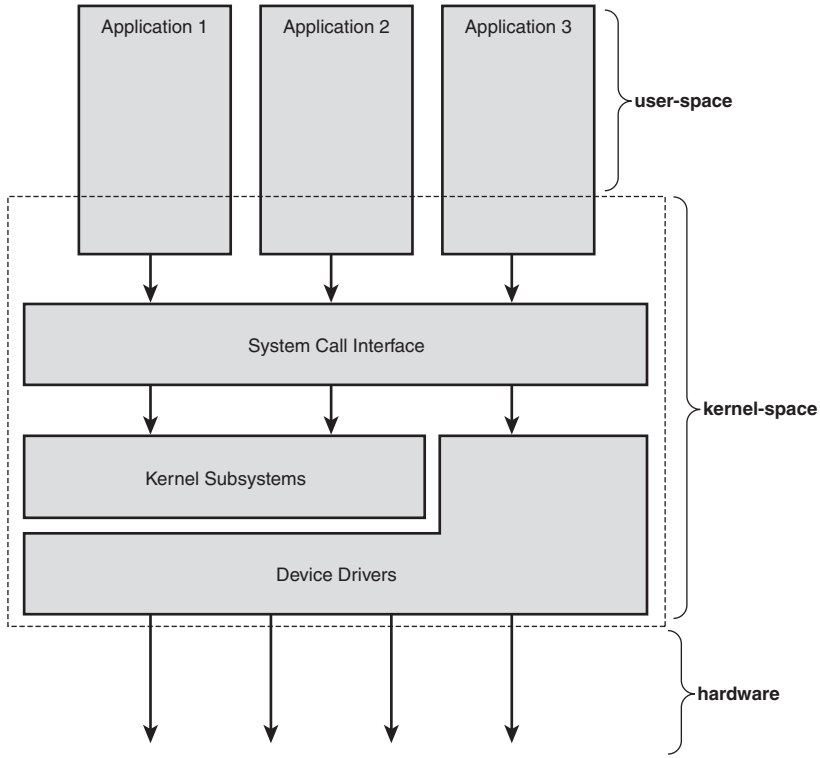

Figure 1.1 Relationship between applications, the kernel, and hardware.

- In kernel-space, in interrupt context, not associated with a process, handling an interrupt

This list is inclusive. Even corner cases fit into one of these three activities: For example, when idle, it turns out that the kernel is executing an *idle process* in process context in the kernel.

Linux Versus Classic Unix Kernels

Owing to their common ancestry and same API, modern Unix kernels share various design traits. (See the Bibliography for my favorite books on the design of the classic Unix kernels.) With few exceptions, a Unix kernel is typically a monolithic static binary. That is, it exists as a single, large, executable image that runs in a single address space. Unix systems typically require a system with a paged memory-management unit (MMU); this hardware enables the system to enforce memory protection and to provide a unique virtual address space to each process. Linux historically has required an MMU, but

special versions can actually run without one. This is a neat feature, enabling Linux to run on very small MMU-less embedded systems, but otherwise more academic than practical—even simple embedded systems nowadays tend to have advanced features such as memory-management units. In this book, we focus on MMU-based systems.

Monolithic Kernel Versus Microkernel Designs

We can divide kernels into two main schools of design: the monolithic kernel and the microkernel. (A third camp, exokernel, is found primarily in research systems.)

Monolithic kernels are the simpler design of the two, and all kernels were designed in this manner until the 1980s. Monolithic kernels are implemented entirely as a single process running in a single address space. Consequently, such kernels typically exist on disk as single static binaries. All kernel services exist and execute in the large kernel address space. Communication within the kernel is trivial because everything runs in kernel mode in the same address space: The kernel can invoke functions directly, as a user-space application might. Proponents of this model cite the simplicity and performance of the monolithic approach. Most Unix systems are monolithic in design.

Microkernels, on the other hand, are not implemented as a single large process. Instead, the functionality of the kernel is broken down into separate processes, usually called *servers*. Ideally, only the servers *absolutely* requiring such capabilities run in a privileged execution mode. The rest of the servers run in user-space. All the servers, though, are separated into different address spaces. Therefore, direct function invocation as in monolithic kernels is not possible. Instead, microkernels communicate via *message passing*: An interprocess communication (IPC) mechanism is built into the system, and the various servers communicate with and invoke "services" from each other by sending messages over the IPC mechanism. The separation of the various servers prevents a failure in one server from bringing down another. Likewise, the modularity of the system enables one server to be swapped out for another.

Because the IPC mechanism involves quite a bit more overhead than a trivial function call, however, and because a context switch from kernel-space to user-space or vice versa is often involved, message passing includes a latency and throughput hit not seen on monolithic kernels with simple function invocation. Consequently, all practical microkernel-based systems now place most or all the servers in kernel-space, to remove the overhead of frequent context switches and potentially enable direct function invocation. The Windows NT kernel (on which Windows XP, Vista, and 7 are based) and Mach (on which part of Mac OS X is based) are examples of microkernels. Neither Windows NT nor Mac OS X run any microkernel servers in user-space in their latest iteration, defeating the primary purpose of microkernel design altogether.

Linux is a monolithic kernel; that is, the Linux kernel executes in a single address space entirely in kernel mode. Linux, however, borrows much of the good from microkernels: Linux boasts a modular design, the capability to preempt itself (called *kernel preemption*), support for kernel threads, and the capability to dynamically load separate binaries (kernel modules) into the kernel image. Conversely, Linux has none of the performance-sapping features that curse microkernel design: Everything runs in kernel mode, with direct function invocation—not message passing—the modus of communication. Nonetheless, Linux is modular, threaded, and the kernel itself is schedulable. Pragmatism wins again.

As Linus and other kernel developers contribute to the Linux kernel, they decide how best to advance Linux without neglecting its Unix roots (and, more important, the Unix API). Consequently, because Linux is not based on any specific Unix variant, Linus and company can pick and choose the best solution to any given problem—or at times, invent new solutions! A handful of notable differences exist between the Linux kernel and classic Unix systems:

- Linux supports the dynamic loading of kernel modules. Although the Linux kernel is monolithic, it can dynamically load and unload kernel code on demand.
- Linux has symmetrical multiprocessor (SMP) support. Although most commercial variants of Unix now support SMP, most traditional Unix implementations did not.
- The Linux kernel is preemptive. Unlike traditional Unix variants, the Linux kernel can preempt a task even as it executes in the kernel. Of the other commercial Unix implementations, Solaris and IRIX have preemptive kernels, but most Unix kernels are not preemptive.
- Linux takes an interesting approach to thread support: It does not differentiate between threads and normal processes. To the kernel, all processes are the same—some just happen to share resources.
- Linux provides an object-oriented device model with device classes, hot-pluggable events, and a user-space device filesystem (sysfs).
- Linux ignores some common Unix features that the kernel developers consider poorly designed, such as STREAMS, or standards that are impossible to cleanly implement.
- Linux is free in every sense of the word. The feature set Linux implements is the result of the freedom of Linux's open development model. If a feature is without merit or poorly thought out, Linux developers are under no obligation to implement it. To the contrary, Linux has adopted an elitist attitude toward changes: Modifications must solve a specific real-world problem, derive from a clean design, and have a solid implementation. Consequently, features of some other modern Unix variants that are more marketing bullet or one-off requests, such as pageable kernel memory, have received no consideration.

Despite these differences, however, Linux remains an operating system with a strong Unix heritage.

Linux Kernel Versions

Linux kernels come in two flavors: stable and development. Stable kernels are production-level releases suitable for widespread deployment. New stable kernel versions are released typically only to provide bug fixes or new drivers. Development kernels, on the other hand, undergo rapid change where (almost) anything goes. As developers experiment with new solutions, the kernel code base changes in often drastic ways.

Linux kernels distinguish between stable and development kernels with a simple naming scheme (see Figure 1.2). Three or four numbers, delineated with a dot, represent Linux kernel versions. The first value is the major release, the second is the minor release, and the third is the revision. An optional fourth value is the stable version. The minor release also determines whether the kernel is a stable or development kernel; an even number is stable, whereas an odd number is development. For example, the kernel version 2.6.30.1 designates a stable kernel. This kernel has a major version of two, a minor version of six, a revision of 30, and a stable version of one. The first two values describe the "kernel series"—in this case, the 2.6 kernel series.

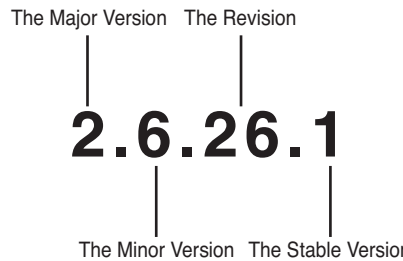

Figure 1.2 Kernel version naming convention.

Development kernels have a series of phases. Initially, the kernel developers work on new features and chaos ensues. Over time, the kernel matures and eventually a feature freeze is declared. At that point, Linus will not accept new features. Work on existing features, however, can continue. After Linus considers the kernel nearly stabilized, a code freeze is put into effect. When that occurs, only bug fixes are accepted. Shortly thereafter (hopefully), Linus releases the first version of a new stable series. For example, the development series 1.3 stabilized into 2.0 and 2.5 stabilized into 2.6.

Within a given series, Linus releases new kernels regularly, with each version earning a new revision. For example, the first version of the 2.6 kernel series was 2.6.0. The next was 2.6.1. These revisions contain bug fixes, new drivers, and new features, but the difference between two revisions—say, 2.6.3 and 2.6.4—is minor.

This is how development progressed until 2004, when at the invite-only Kernel Developers Summit, the assembled kernel developers decided to prolong the 2.6 kernel series and postpone the introduction of a 2.7 development series. The rationale was that the 2.6 kernel was well received, stable, and sufficiently mature such that new destabilizing features were unneeded. This course has proven wise, as the ensuing years have shown 2.6 is a mature and capable beast. As of this writing, a 2.7 development series is not on the table and seems unlikely. Instead, the development cycle of each 2.6 revision has grown longer, each release incorporating a mini-development series. Andrew Morton, Linus's second-in-command, has repurposed his 2.6-mm tree—once a testing ground for memory management-related changes—into a general-purpose test bed. Destabilizing

changes thus flow into 2.6-mm and, when mature, into one of the 2.6 mini-development series. Thus, over the last few years, each 2.6 release—for example, 2.6.29—has taken several months, boasting significant changes over its predecessor. This "development series in miniature" has proven rather successful, maintaining high levels of stability while still introducing new features and appears unlikely to change in the near future. Indeed, the consensus among kernel developers is that this new release process will continue indefinitely.

To compensate for the reduced frequency of releases, the kernel developers have introduced the aforementioned *stable release*. This release (the *8* in 2.6.32.8) contains crucial bug fixes, often back-ported from the under-development kernel (in this example, 2.6.33). In this manner, the previous release continues to receive attention focused on stabilization.

The Linux Kernel Development Community

When you begin developing code for the Linux kernel, you become a part of the global kernel development community. The main forum for this community is the *Linux Kernel Mailing List* (oft-shortened to *lkml*). Subscription information is available at http://vger.kernel.org. Note that this is a high-traffic list with hundreds of messages a day and that the other readers—who include all the core kernel developers, including Linus—are not open to dealing with nonsense. The list is, however, a priceless aid during development because it is where you can find testers, receive peer review, and ask questions.

Later chapters provide an overview of the kernel development process and a more complete description of participating successfully in the kernel development community. In the meantime, however, lurking on (silently reading) the Linux Kernel Mailing List is as good a supplement to this book as you can find.

Before We Begin

This book is about the Linux kernel: its goals, the design that fulfills those goals, and the implementation that realizes that design. The approach is practical, taking a middle road between theory and practice when explaining how everything works. My objective is to give you an insider's appreciation and understanding for the design and implementation of the Linux kernel. This approach, coupled with some personal anecdotes and tips on kernel hacking, should ensure that this book gets you off the ground running, whether you are looking to develop core kernel code, a new device driver, or simply better understand the Linux operating system.

While reading this book, you should have access to a Linux system and the kernel source. Ideally, by this point, you are a Linux user and have poked and prodded at the source, but require some help making it all come together. Conversely, you might never have used Linux but just want to learn the design of the kernel out of curiosity. However, if your desire is to write some code of your own, there is no substitute for the source. The source code is *freely* available; use it!

Oh, and above all else, *have fun*!

2

Getting Started with the Kernel

In this chapter, we introduce some of the basics of the Linux kernel: where to get its source, how to compile it, and how to install the new kernel. We then go over the differences between the kernel and user-space programs and common programming constructs used in the kernel. Although the kernel certainly is unique in many ways, at the end of the day it is little different from any other large software project.

Obtaining the Kernel Source

The current Linux source code is always available in both a complete *tarball* (an archive created with the *tar* command) and an incremental patch from the official home of the Linux kernel, http://www.kernel.org.

Unless you have a specific reason to work with an older version of the Linux source, you *always* want the latest code. The repository at kernel.org is the place to get it, along with additional patches from a number of leading kernel developers.

Using Git

Over the last couple of years, the kernel hackers, led by Linus himself, have begun using a new version control system to manage the Linux kernel source. Linus created this system, called *Git*, with speed in mind. Unlike traditional systems such as *CVS*, Git is distributed, and its usage and workflow is consequently unfamiliar to many developers. I strongly recommend using Git to download and manage the Linux kernel source.

You can use Git to obtain a copy of the latest "pushed" version of Linus's tree:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

When checked out, you can update your tree to Linus's latest:

```
$ git pull
```

With these two commands, you can obtain and subsequently keep up to date with the official kernel tree. To commit and manage your own changes, see Chapter 20, "Patches,

Hacking, and the Community." A complete discussion of Git is outside the scope of this book; many online resources provide excellent guides.

Installing the Kernel Source

The kernel tarball is distributed in both GNU zip (gzip) and bzip2 format. Bzip2 is the default and preferred format because it generally compresses quite a bit better than gzip. The Linux kernel tarball in bzip2 format is named `linux-x.y.z.tar.bz2`, where *x.y.z* is the version of that particular release of the kernel source. After downloading the source, uncompressing and untarring it is simple. If your tarball is compressed with bzip2, run

```
$ tar xvjf linux-x.y.z.tar.bz2
```

If it is compressed with GNU zip, run

```
$ tar xvzf linux-x.y.z.tar.gz
```

This uncompresses and untars the source to the directory `linux-x.y.z`. If you use git to obtain and manage the kernel source, you do not need to download the tarball. Just run the *git clone* command as described and git downloads and unpacks the latest source.

> **Where to Install and Hack on the Source**
>
> The kernel source is typically installed in `/usr/src/linux`. You should not use this source tree for development because the kernel version against which your C library is compiled is often linked to this tree. Moreover, you should not require root in order to make changes to the kernel—instead, work out of your home directory and use root only to install new kernels. Even when installing a new kernel, `/usr/src/linux` should remain untouched.

Using Patches

Throughout the Linux kernel community, patches are the *lingua franca* of communication. You will distribute your code changes in patches and receive code from others as patches. *Incremental patches* provide an easy way to move from one kernel tree to the next. Instead of downloading each large tarball of the kernel source, you can simply apply an incremental patch to go from one version to the next. This saves everyone bandwidth and you time. To apply an incremental patch, from *inside* your kernel source tree, simply run

```
$ patch -p1 < ../patch-x.y.z
```

Generally, a patch to a given version of the kernel is applied against the previous version. Generating and applying patches is discussed in much more depth in later chapters.

The Kernel Source Tree

The kernel source tree is divided into a number of directories, most of which contain many more subdirectories. The directories in the root of the source tree, along with their descriptions, are listed in Table 2.1.

Table 2.1 **Directories in the Root of the Kernel Source Tree**

Directory	Description
`arch`	Architecture-specific source
`block`	Block I/O layer
`crypto`	Crypto API
`Documentation`	Kernel source documentation
`drivers`	Device drivers
`firmware`	Device firmware needed to use certain drivers
`fs`	The VFS and the individual filesystems
`include`	Kernel headers
`init`	Kernel boot and initialization
`ipc`	Interprocess communication code
`kernel`	Core subsystems, such as the scheduler
`lib`	Helper routines
`mm`	Memory management subsystem and the VM
`net`	Networking subsystem
`samples`	Sample, demonstrative code
`scripts`	Scripts used to build the kernel
`security`	Linux Security Module
`sound`	Sound subsystem
`usr`	Early user-space code (called initramfs)
`tools`	Tools helpful for developing Linux
`virt`	Virtualization infrastructure

A number of files in the root of the source tree deserve mention. The file `COPYING` is the kernel license (the GNU GPL v2). `CREDITS` is a listing of developers with more than a trivial amount of code in the kernel. `MAINTAINERS` lists the names of the individuals who maintain subsystems and drivers in the kernel. `Makefile` is the base kernel Makefile.

Building the Kernel

Building the kernel is easy. It is surprisingly easier than compiling and installing other system-level components, such as glibc. The 2.6 kernel series introduced a new configuration and build system, which made the job even easier and is a welcome improvement over earlier releases.

Configuring the Kernel

Because the Linux source code is available, it follows that you can configure and custom tailor it before compiling. Indeed, it is possible to compile support into your kernel for only the specific features and drivers you want. Configuring the kernel is a required step before building it. Because the kernel offers myriad features and supports a varied basket of hardware, there is a *lot* to configure. Kernel configuration is controlled by configuration options, which are prefixed by `CONFIG` in the form `CONFIG_FEATURE`. For example, symmetrical multiprocessing (SMP) is controlled by the configuration option `CONFIG_SMP`. If this option is set, SMP is enabled; if unset, SMP is disabled. The configure options are used both to decide which files to build and to manipulate code via preprocessor directives.

Configuration options that control the build process are either *Booleans* or *tristates*. A Boolean option is either *yes* or *no*. Kernel features, such as `CONFIG_PREEMPT`, are usually Booleans. A tristate option is one of *yes*, *no*, or *module*. The *module* setting represents a configuration option that is set but is to be compiled as a module (that is, a separate dynamically loadable object). In the case of tristates, a *yes* option explicitly means to compile the code into the main kernel image and not as a module. Drivers are usually represented by tristates.

Configuration options can also be strings or integers. These options do not control the build process but instead specify values that kernel source can access as a preprocessor macro. For example, a configuration option can specify the size of a statically allocated array.

Vendor kernels, such as those provided by Canonical for Ubuntu or Red Hat for Fedora, are precompiled as part of the distribution. Such kernels typically enable a good cross section of the needed kernel features and compile nearly all the drivers as modules. This provides for a great base kernel with support for a wide range of hardware as separate modules. For better or worse, as a kernel hacker, you need to compile your own kernels and learn what modules to include on your own.

Thankfully, the kernel provides multiple tools to facilitate configuration. The simplest tool is a text-based command-line utility:

```
$ make config
```

This utility goes through each option, one by one, and asks the user to interactively select *yes*, *no*, or (for tristates) *module*. Because this takes a *long* time, unless you are paid by the hour, you should use an ncurses-based graphical utility:

```
$ make menuconfig
```

Or a gtk+-based graphical utility:

```
$ make gconfig
```

These three utilities divide the various configuration options into categories, such as "Processor Type and Features." You can move through the categories, view the kernel options, and of course change their values.

This command creates a configuration based on the defaults for your architecture:

```
$ make defconfig
```

Although these defaults are somewhat arbitrary (on i386, they are rumored to be Linus's configuration!), they provide a good start if you have never configured the kernel. To get off and running quickly, run this command and then go back and ensure that configuration options for your hardware are enabled.

The configuration options are stored in the root of the kernel source tree in a file named `.config`. You may find it easier (as most of the kernel developers do) to just edit this file directly. It is quite easy to search for and change the value of the configuration options. After making changes to your configuration file, or when using an existing configuration file on a new kernel tree, you can validate and update the configuration:

```
$ make oldconfig
```

You should always run this before building a kernel.

The configuration option `CONFIG_IKCONFIG_PROC` places the complete kernel configuration file, compressed, at `/proc/config.gz`. This makes it easy to clone your current configuration when building a new kernel. If your current kernel has this option enabled, you can copy the configuration out of `/proc` and use it to build a new kernel:

```
$ zcat /proc/config.gz > .config
$ make oldconfig
```

After the kernel configuration is set—however you do it—you can build it with a single command:

```
$ make
```

Unlike kernels before 2.6, you no longer need to run `make dep` before building the kernel—the dependency tree is maintained automatically. You also do not need to specify a specific build type, such as bzImage, or build modules separately, as you did in old versions. The default Makefile rule will handle everything.

Minimizing Build Noise

A trick to minimize build noise, but still see warnings and errors, is to redirect the output from `make`:

```
$ make > ../detritus
```

If you need to see the build output, you can read the file. Because the warnings and errors are output to standard error, however, you normally do not need to. In fact, I just do

```
$ make > /dev/null
```

This redirects all the worthless output to that big, ominous sink of no return, `/dev/null`.

Spawning Multiple Build Jobs

The `make` program provides a feature to split the build process into a number of parallel *jobs*. Each of these jobs then runs separately and concurrently, significantly speeding up the build process on multiprocessing systems. It also improves processor utilization because the time to build a large source tree includes significant time in I/O wait (time in which the process is idle waiting for an I/O request to complete).

By default, `make` spawns only a single job because Makefiles all too often have incorrect dependency information. With incorrect dependencies, multiple jobs can step on each other's toes, resulting in errors in the build process. The kernel's Makefiles have correct dependency information, so spawning multiple jobs does not result in failures. To build the kernel with multiple make jobs, use

```
$ make -jn
```

Here, *n* is the number of jobs to spawn. Usual practice is to spawn one or two jobs per processor. For example, on a 16-core machine, you might do

```
$ make -j32 > /dev/null
```

Using utilities such as the excellent `distcc` or `ccache` can also dramatically improve kernel build time.

Installing the New Kernel

After the kernel is built, you need to install it. How it is installed is architecture- and boot loader-dependent—consult the directions for your boot loader on where to copy the kernel image and how to set it up to boot. Always keep a known-safe kernel or two around in case your new kernel has problems!

As an example, on an x86 system using grub, you would copy `arch/i386/boot/bzImage` to `/boot`, name it something like `vmlinuz-`*`version`*, and edit `/boot/grub/grub.conf`, adding a new entry for the new kernel. Systems using LILO to boot would instead edit `/etc/lilo.conf` and then rerun `lilo`.

Installing modules, thankfully, is automated and architecture-independent. As root, simply run

```
% make modules_install
```

This installs all the compiled modules to their correct home under `/lib/modules`.

The build process also creates the file `System.map` in the root of the kernel source tree. It contains a symbol lookup table, mapping kernel symbols to their start addresses. This is used during debugging to translate memory addresses to function and variable names.

A Beast of a Different Nature

The Linux kernel has several unique attributes as compared to a normal user-space application. Although these differences do not necessarily make developing kernel code *harder* than developing user-space code, they certainly make doing so *different*.

These characteristics make the kernel a beast of a different nature. Some of the usual rules are bent; other rules are entirely new. Although some of the differences are obvious (we all know the kernel can do anything it wants), others are not so obvious. The most important of these differences are

- The kernel has access to neither the C library nor the standard C headers.
- The kernel is coded in GNU C.
- The kernel lacks the memory protection afforded to user-space.
- The kernel cannot easily execute floating-point operations.
- The kernel has a small per-process fixed-size stack.
- Because the kernel has asynchronous interrupts, is preemptive, and supports SMP, synchronization and concurrency are major concerns within the kernel.
- Portability is important.

Let's briefly look at each of these issues because all kernel developers must keep them in mind.

No libc or Standard Headers

Unlike a user-space application, the kernel is not linked against the standard C library—or any other library, for that matter. There are multiple reasons for this, including a chicken-and-the-egg situation, but the primary reason is speed and size. The full C library—or even a decent subset of it—is too large and too inefficient for the kernel.

Do not fret: Many of the usual libc functions are implemented inside the kernel. For example, the common string manipulation functions are in `lib/string.c`. Just include the header file `<linux/string.h>` and have at them.

> **Header Files**
>
> When I talk about header files in this book, I am referring to the kernel header files that are part of the kernel source tree. Kernel source files cannot include outside headers, just as they cannot use outside libraries.
>
> The base files are located in the `include/` directory in the root of the kernel source tree. For example, the header file `<linux/inotify.h>` is located at `include/linux/inotify.h` in the kernel source tree.
>
> A set of architecture-specific header files are located in `arch/<architecture>/include/asm` in the kernel source tree. For example, if compiling for the x86 architecture, your architecture-specific headers are in `arch/x86/include/asm`. Source code includes these headers via just the `asm/` prefix, for example `<asm/ioctl.h>`.

Of the missing functions, the most familiar is `printf()`. The kernel does not have access to `printf()`, but it does provide `printk()`, which works pretty much the same as its more familiar cousin. The `printk()` function copies the formatted string into the kernel log buffer, which is normally read by the syslog program. Usage is similar to `printf()`:

```
printk("Hello world! A string '%s' and an integer '%d'\n", str, i);
```

One notable difference between `printf()` and `printk()` is that `printk()` enables you to specify a priority flag. This flag is used by `syslogd` to decide where to display kernel messages. Here is an example of these priorities:

```
printk(KERN_ERR "this is an error!\n");
```

Note there is no comma between `KERN_ERR` and the printed message. This is intentional; the priority flag is a preprocessor-define representing a string literal, which is concatenated onto the printed message during compilation. We use `printk()` throughout this book.

GNU C

Like any self-respecting Unix kernel, the Linux kernel is programmed in C. Perhaps surprisingly, the kernel is not programmed in strict ANSI C. Instead, where applicable, the kernel developers make use of various language extensions available in *gcc* (the GNU Compiler Collection, which contains the C compiler used to compile the kernel and most everything else written in C on a Linux system).

The kernel developers use both ISO C99[1] and GNU C extensions to the C language. These changes wed the Linux kernel to gcc, although recently one other compiler, the Intel C compiler, has sufficiently supported enough gcc features that it, too, can compile the Linux kernel. The earliest supported gcc version is 3.2; gcc version 4.4 or later is recommended. The ISO C99 extensions that the kernel uses are nothing special and, because C99 is an official revision of the C language, are slowly cropping up in a lot of other code. The more unfamiliar deviations from standard ANSI C are those provided by GNU C. Let's look at some of the more interesting extensions that you will see in the kernel; these changes differentiate kernel code from other projects with which you might be familiar.

Inline Functions

Both C99 and GNU C support *inline functions*. An inline function is, as its name suggests, inserted inline into each function call site. This eliminates the overhead of function invocation and return (register saving and restore) and allows for potentially greater optimization as the compiler can optimize both the caller and the called function as one. As a downside (nothing in life is free), code size increases because the contents of the function are copied into all the callers, which increases memory consumption and instruction cache footprint. Kernel developers use inline functions for small time-critical functions.

---

[1] *ISO C99 is the latest major revision to the ISO C standard. C99 adds numerous enhancements to the previous major revision, ISO C90, including designated initializers, variable length arrays, C++-style comments, and the* `long long` *and* `complex` *types. The Linux kernel, however, employs only a subset of C99 features.*

Making large functions inline, especially those used more than once or that are not exceedingly time critical, is frowned upon.

An inline function is declared when the keywords `static` and `inline` are used as part of the function definition. For example

```
static inline void wolf(unsigned long tail_size)
```

The function declaration must precede any usage, or else the compiler cannot make the function inline. Common practice is to place inline functions in header files. Because they are marked `static`, an exported function is not created. If an inline function is used by only one file, it can instead be placed toward the top of just that file.

In the kernel, using inline functions is preferred over complicated macros for reasons of type safety and readability.

Inline Assembly

The gcc C compiler enables the embedding of assembly instructions in otherwise normal C functions. This feature, of course, is used in only those parts of the kernel that are unique to a given system architecture.

The `asm()` compiler directive is used to inline assembly code. For example, this inline assembly directive executes the x86 processor's `rdtsc` instruction, which returns the value of the timestamp (`tsc`) register:

```
unsigned int low, high;
asm volatile("rdtsc" : "=a" (low), "=d" (high));
/* low and high now contain the lower and upper 32-bits of the 64-bit tsc */
```

The Linux kernel is written in a mixture of C and assembly, with assembly relegated to low-level architecture and fast path code. The vast majority of kernel code is programmed in straight C.

Branch Annotation

The gcc C compiler has a built-in directive that optimizes conditional branches as either very likely taken or very unlikely taken. The compiler uses the directive to appropriately optimize the branch. The kernel wraps the directive in easy-to-use macros, `likely()` and `unlikely()`.

For example, consider an `if` statement such as the following:

```
if (error) {
        /* ... */
}
```

To mark this branch as very unlikely taken (that is, likely not taken):

```
/* we predict 'error' is nearly always zero ... */
if (unlikely(error)) {
        /* ... */
}
```

Conversely, to mark a branch as very likely taken:

```
/* we predict 'success' is nearly always nonzero ... */
if (likely(success)) {
        /* ... */
}
```

You should only use these directives when the branch direction is overwhelmingly known *a priori* or when you want to optimize a specific case at the cost of the other case. This is an important point: These directives result in a performance boost when the branch is correctly marked, but a performance *loss* when the branch is mismarked. A common usage, as shown in these examples, for `unlikely()` and `likely()` is error conditions. As you might expect, `unlikely()` finds much more use in the kernel because `if` statements tend to indicate a special case.

No Memory Protection

When a user-space application attempts an illegal memory access, the kernel can trap the error, send the `SIGSEGV` signal, and kill the process. If the kernel attempts an illegal memory access, however, the results are less controlled. (After all, who is going to look after the kernel?) Memory violations in the kernel result in an *oops*, which is a major kernel error. It should go without saying that you must not illegally access memory, such as dereferencing a `NULL` pointer—but within the kernel, the stakes are much higher!

Additionally, kernel memory is not pageable. Therefore, every byte of memory you consume is one less byte of available physical memory. Keep that in mind the next time you need to add *one more feature* to the kernel!

No (Easy) Use of Floating Point

When a user-space process uses floating-point instructions, the kernel manages the transition from integer to floating point mode. What the kernel has to do when using floating-point instructions varies by architecture, but the kernel normally catches a trap and then initiates the transition from integer to floating point mode.

Unlike user-space, the kernel does not have the luxury of seamless support for floating point because it cannot easily trap itself. Using a floating point inside the kernel requires manually saving and restoring the floating point registers, among other possible chores. The short answer is: *Don't do it*! Except in the rare cases, no floating-point operations are in the kernel.

Small, Fixed-Size Stack

User-space can get away with statically allocating many variables on the stack, including huge structures and thousand-element arrays. This behavior is legal because user-space has a large stack that can dynamically grow. (Developers on older, less advanced operating systems—say, DOS—might recall a time when even user-space had a fixed-sized stack.)

The kernel stack is neither large nor dynamic; it is small and fixed in size. The exact size of the kernel's stack varies by architecture. On x86, the stack size is configurable at compile-time and can be either 4KB or 8KB. Historically, the kernel stack is two pages, which generally implies that it is 8KB on 32-bit architectures and 16KB on 64-bit architectures—this size is fixed and absolute. Each process receives its own stack.

The kernel stack is discussed in much greater detail in later chapters.

Synchronization and Concurrency

The kernel is susceptible to race conditions. Unlike a single-threaded user-space application, a number of properties of the kernel allow for concurrent access of shared resources and thus require synchronization to prevent races. Specifically

- Linux is a preemptive multitasking operating system. Processes are scheduled and rescheduled at the whim of the kernel's process scheduler. The kernel must synchronize between these tasks.
- Linux supports symmetrical multiprocessing (SMP). Therefore, without proper protection, kernel code executing simultaneously on two or more processors can concurrently access the same resource.
- Interrupts occur asynchronously with respect to the currently executing code. Therefore, without proper protection, an interrupt can occur in the midst of accessing a resource, and the interrupt handler can then access the same resource.
- The Linux kernel is preemptive. Therefore, without protection, kernel code can be preempted in favor of different code that then accesses the same resource.

Typical solutions to race conditions include spinlocks and semaphores. Later chapters provide a thorough discussion of synchronization and concurrency.

Importance of Portability

Although user-space applications do not *have* to aim for portability, Linux is a portable operating system and should remain one. This means that architecture-independent C code must correctly compile and run on a wide range of systems, and that architecture-dependent code must be properly segregated in system-specific directories in the kernel source tree.

A handful of rules—such as remain endian neutral, be 64-bit clean, do not assume the word or page size, and so on—go a long way. Portability is discussed in depth in a later chapter.

Conclusion

To be sure, the kernel has unique qualities. It enforces its own rules and the stakes, managing the entire system as the kernel does, are certainly higher. That said, the Linux kernel's complexity and barrier-to-entry is not qualitatively different from any other large soft-

ware project. The most important step on the road to Linux development is the realization that the kernel is not something to fear. Unfamiliar, sure. Insurmountable? Not at all.

This and the previous chapter lay the foundation for the topics we cover through this book's remaining chapters. In each subsequent chapter, we cover a specific kernel concept or subsystem. Along the way, it is imperative that you read and modify the kernel source. Only through actually reading and experimenting with the code can you ever understand it. The source is freely available—use it!