

# 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.<sup>1</sup>

---

<sup>1</sup> 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 `esh` 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*.<sup>2</sup> 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, multi-threading, virtual memory, demand paging, shared libraries with demand loading, and

---

<sup>2</sup> 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 Linux: 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.<sup>3</sup> 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.<sup>4</sup>

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

---

<sup>3</sup> I will leave the free versus open debate to you. See <http://www.fsf.org> and <http://www.opensource.org>.

<sup>4</sup> 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

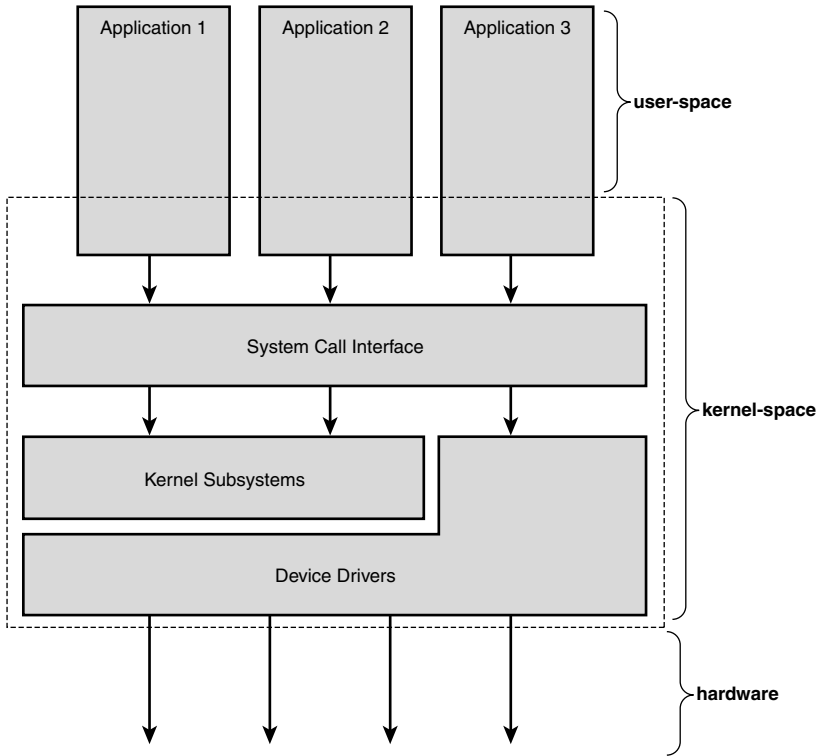


Figure 1.1 Relationship between applications, the kernel, and hardware.

- 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 inter-process 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.

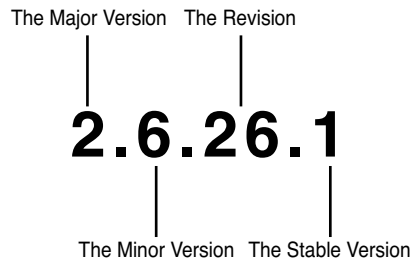


Figure 1.2 Kernel version naming convention.

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!*