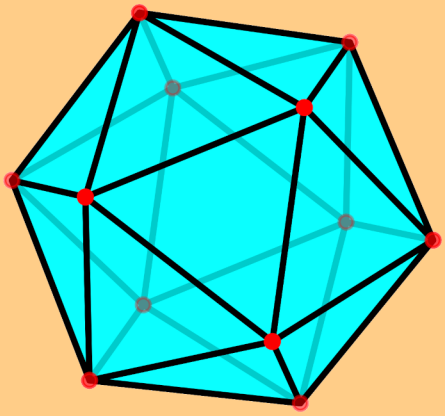
Updated for Linux 6.x Updated for Linux 6.x Updated for Linux 6.x Updated for Linux 6.x Updated for Linux 6.x Updated for Linux 6.x Updated for Linux 6.x Updated for Linux 6.x Updated for Linux 6.x Updated Linux

# Understanding Linux: The Kernel Perspective

Vladimir Likić, PhD



Leanpub

# Understanding Linux: The Kernel Perspective

## Covers Linux Kernel 6.x

Vladimir Likic

This book is available at https://leanpub.com/linuxkernel

This version was published on 2025-12-20


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016 - 2025 Vladimir Likic

*This book is dedicated to my parents.*

# Contents

# Prologue

The Unix operating system is based on a few elegant ideas conceived in the late 1960s. The first version appeared in 1969, originally called UNICS—a pun on the unsuccessful MULTICS project. It was later renamed UNIX, with the first edition of the UNIX Programmer's Manual released in 1971. Over the next five decades, Unix was refined and extended by generations of elite computer scientists. This organic growth led to the incredible richness and depth, all inherited by today's Linux.

Modern Linux is a continuation of Unix, shaped by the Open Source development model and profoundly influenced by the rise of the Internet and relentless advances in hardware. It runs on faster, cheaper, and more ubiquitous machines than its creators could have imagined.

At the core of Linux is the kernel—the lowest layer of the system software stack and the component that interfaces directly with the hardware. The kernel is both highly efficient and largely invisible to the untrained eye. While most technically inclined users know that the kernel keeps programs running, this is only part of the story: the kernel creates processes (programs in execution), and those processes exist only within the kernel's context. Regular programs do not run directly on hardware but on the combination of the kernel and the hardware.

This book explores Linux from the kernel perspective: our goal is to peek under the hood of the Linux kernel to understand its principles. To that end, we sometimes skip over less essential details. For example, while device drivers are important to the kernel, they are only of marginal relevance to users seeking a deeper understanding of how the kernel works. Accordingly, we treat them briefly. A key challenge of writing such a book is that the Linux kernel cannot be learned from theory alone. Practical experience is essential. That's why a defining feature are exercises—hands-on examples that demonstrate working kernel modules.

Kernel modules allow additional C code to be compiled and inserted into a running kernel, where it becomes part of the kernel itself. This makes them an ideal tool for experimentation—the most exciting way to learn about the kernel. Each exercise is designed to illustrate a specific concept and to give the reader a taste of the fascinating world of kernel programming.

# Introduction

## What is the Linux kernel?

Under the hood of every Linux system runs the Linux kernel—the software layer that interfaces directly with the hardware and facilitates running of all other programs. At any moment, the kernel is juggling the processes[1] on the system, keeping most frozen while briefly awakening others, all while enforcing what each is allowed to do. The combination of the kernel and the hardware forms a single, tightly integrated platform on which all processes run. When we say "a program is running", what we really mean is that the kernel is presenting a process to us. In fact, the kernel created the process, and the process exists only within the kernel's context. Without the kernel, there would be no such thing as a "process."

To put this in a practical perspective, as I write these words, 294 processes are running on my machine:

```
$ ps aux | wc -l
294
```

In this snippet, the shell launched two programs: *ps* (actually */bin/ps*) and *wc* (*/usr/bin/wc*). These programs ran, sent their output to the standard output, and then exited. The shell managed the redirection—sending the output of *ps* as input to *wc*, and then printing the result.

Where is the kernel in all this? *bash* was running on it. More precisely, the kernel was running bash. It was the kernel that transformed the *bash* executable (a file of carefully structured data) into a running process. When I typed *ps* at the *bash* prompt, this requested from the kernel to execute the binary */bin/ps*. The kernel checked whether *bash* had permission to do so, then created a new process from the *ps* executable */bin/ps*.

This simple example illustrates how, on a Linux system, no meaningful action occurs without the silent involvement of the Linux kernel. Starting a program, accessing a file on disk, or using any peripheral device—all these actions involve the kernel.

[1] A process is a program in execution

The kernel also manages the memory for every running process. When a process allocates memory, it is actually manipulating an artificial structure presented to it by the kernel, called virtual memory (we discuss this later in some detail). If the process is entitled to the requested memory, the kernel maps actual physical memory and links the process to it. (Of course, the kernel may also deny the request.)

All processes must obey the rules imposed by the kernel. Any violation triggers drastic consequences: typically, the kernel terminates the offending process, and the user sees the mysterious message: *segmentation fault (core dumped)*.

Because the kernel is at the heart of every Linux system, the benefit of understanding how it works is obvious. This is especially true for system administrators—who, in fact, already possess a great deal of operational knowledge about the kernel.

And what about developers? While it's possible to partially isolate the programming environment from the underlying OS, complete isolation is never achievable. Every component of the development environment itself runs on the kernel and is subject to its control.

Modern paradigms like DevOps and cloud native further blur the lines between development, system administration, testing, and deployment. If you work with Linux systems in any capacity, having a solid grasp of the kernel is almost always relevant— and highly desirable.

Consider the following example.

First, a warning: running the example below may crash your computer. If you choose to proceed, open a *bash* terminal and run the following command:

```
$ c(){ c|c& };c
```

This one-liner can send the kernel into a tailspin, usually forcing a reboot to recover the system. This is known as a "fork bomb", and we will explain what is happening here in the chapter Processes. For now note that the above example is run as a regular user—not as root—and this behaviour is expected, not a bug.

But how is this even possible? To understand what's happening here, one needs a non-trivial grasp of Unix, including:

- What exactly does the command do? (This requires understanding *bash* more than the kernel.)
- What does the kernel do when this command is run? (This also explains why it goes into a tailspin.)
- How can a regular user crash the system with such a simple command, and why is this not a bug? (This reflects the Unix philosophy.)

- What can be done to prevent a malicious user from doing this? (This calls for practical sysadmin knowledge.)

Answering these questions requires a well-rounded understanding of Linux, *bash*, and the kernel—this is the kind of knowledge that this book aims to provide.

## The kernel source: a double-edged sword

The source code of the entire Linux kernel is freely available on the Internet for anyone to download. So one possible approach to studying the kernel is to study the kernel source code. This however is a daunting proposition. The kernel source code is very, very complex. Today not even Linus Torvalds understands all of it[2]. A great deal of prior knowledge is required to know even where to start with the kernel source code, and going straight for the kernel code is therefore very much a double-edged sword.

The Linux kernel is most interesting when it's *alive*—running on hardware. Interacting with the running kernel is far more instructive (and fun) than merely reading its source code, though a bit of the latter is necessary too. If we treat the running system as a black box, we can probe it with inputs, observe the outputs, and by correlating the two gain deep insights into what the kernel is doing. This book makes extensive use of that approach.

## Kernel and applications: a first approximation

In the first approximation, one might think of a Linux system as consisting of two parts: the kernel and the applications that run atop it. The kernel is the software layer that interfaces directly with the hardware; the applications are programs that run under the kernel's supervision and make the computer usable to the user.

Consider, for example, the utility *ps* used earlier. *ps* is a small standalone program, */bin/ps*; when run from the command line, it executes under the kernel's supervision, typically prints some output, and then exits. *ps* is an example of an application, and more specifically, a utility—a special kind of application.

Both the Linux kernel and Linux utilities trace their lineage to Unix. Unix was conceived in 1969 and has been in continuous development for over 50 years; Linux represents a major development in the most recent phase of Unix, so to speak[3].

[2] At the Open Source Summit held in Vancouver in 2018, Linus said: "*No one knows the whole kernel. But, having looked at patches for many many years, I know the big picture.*"

[3] Dennis Ritchie, one of the original creators of Unix, said in 1999: "*I think the Linux phenomenon is quite delightful, because it draws so strongly on the basis that Unix provided*"

Today, Linux powers both servers and embedded devices. Android, the most popular mobile OS, is Linux-based. Linux spans an incredible range, from handheld gadgets to the world's most powerful computers. Android dominates the global mobile market; every one of the world's 500 fastest supercomputers runs Linux; and the Internet runs on Linux—millions of servers at Google, Facebook, Amazon, X/Twitter, and eBay operate atop it.

Never before has a single operating system dominated the computing landscape as thoroughly as Linux does today. The only segment it hasn't conquered is the desktop which remains the domain of Microsoft Windows. Tellingly, that segment is also in long-term decline.

The Linux kernel is Open Source[4], and anyone can download the entire kernel code. This means you can study it, modify it, and even sell it under certain conditions. For anyone curious about how the kernel works, this is excellent news.

Rewind to the era before 1992, and only a privileged class of engineers at elite tech companies like Sun Microsystems or Silicon Graphics had access to kernel source code. Today, anyone with an Internet connection can download, compile, and even modify the Linux kernel (in fact, the most important and the most widely used kernel in the world). We truly live in exceptional times.

## Exercise 1.1: Getting the kernel source code

At the time of writing, several long-term maintenance kernel releases are listed on the Linux kernel website[5]. One of these is version 6.1, which we will use throughout this book when examining kernel code.

The goal of Exercise 1.1 is to download the kernel source, unpack it, and begin developing a basic orientation to its structure.

To download the kernel 6.1 source code, use:

[4]For the history of Linux see Appendix 3

[5]https://www.kernel.org/category/releases.html

```
$ wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.1.tar.xz
--2025-05-17 22:28:27--
    https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.1.tar.xz
Resolving cdn.kernel.org (cdn.kernel.org)... 2a04:4e42:7::432,
    151.101.29.176
Connecting to cdn.kernel.org (cdn.kernel.org)|2a04:4e42:7::432|
    :443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 134728520 (128M) [application/x-xz]
Saving to: ‘linux-6.1.tar.xz’

linux-6.1.tar.xz  100%[===================================>] 128.49M
    6.09MB/s    in 21s
2025-05-17 22:28:48 (6.07 MB/s) - ‘linux-6.1.tar.xz’
    saved [134728520/134728520]
```

As the output shows, the compressed kernel 6.1 source is just over 120 MB. To unpack the source code:

```
$ tar xfv linux-6.1.tar.xz
linux-6.1/
linux-6.1/.clang-format
linux-6.1/.cocciconfig
linux-6.1/.get_maintainer.ignore
linux-6.1/.gitattributes
linux-6.1/.gitignore
linux-6.1/.mailmap
linux-6.1/.rustfmt.toml
linux-6.1/COPYING
linux-6.1/CREDITS
linux-6.1/Documentation/
linux-6.1/Documentation/.gitignore
linux-6.1/Documentation/ABI/
...
```

Then inspect the files and directories that came with the kernel:

```
$ cd linux-6.1
$ ls -F
arch/            io_uring/    README
block/           ipc/         rust/
certs/           Kbuild       samples/
COPYING          Kconfig      scripts/
CREDITS          kernel/      security/
crypto/          lib/         sound/
Documentation/   LICENSES/    tools/
drivers/         MAINTAINERS  usr/
fs/              Makefile     virt/
include/         mm/
init/            net/
```

Unpacking *linux-6.1.tar.xz* produced a directory named *linux-6.1/*, containing the kernel source components organised into various subdirectories. Henceforth, we will refer to this directory as the root of the kernel source tree. Since this root directory's name changes with each new kernel version, we will always express file paths relative to it. For example, to refer to *linux-6.1/mm*, we will simply write *./mm*.

Spend some time exploring the top-level directories to get a feel for how the kernel is organised. For example, *./arch/* contains architecture-specific code (e.g., x86, arm, riscv), *./drivers/* holds device drivers (USB, network cards, etc.), and *./fs/* implements various filesystems. The *./mm/* directory contains the memory management subsystem, and *./kernel/* includes core kernel functionality like process scheduling and timekeeping.

You don't need to understand any of the code yet, just notice the scope and modularity. The kernel is a large but well-structured system. In later chapters, we will zoom into some of these subsystems, but for now, it's useful to build a mental map of where things live in the kernel source tree.

## Kernel space, user space, and system calls

The terms kernel space and user space appear frequently in Linux discussions but are often misunderstood. A common misconception is that running a command as *root* means operating in kernel space. From the kernel's perspective, all user activities—including those by root—occur in user space. The root user simply has elevated privileges within the user space, but does not run code in kernel space.

Consider the bigger picture: the kernel has full control over the system. It can view any process's actions, access all data, and control all hardware with the highest privileges. In fact, the kernel is the core of the system. Because of this power, the

kernel must strictly separate its own actions from user actions and prevent user processes from gaining kernel privileges. This strict separation between kernel space and user space—regardless of user identity, even for *root*—is to ensure system security and stability.

Even reading a few bytes from a file requires accessing hardware, something only the kernel can do. In other words, even simple actions like reading a file must cross into kernel space. So how does a regular user perform such tasks? A user process must request the kernel to act on its behalf. The kernel first verifies whether the process has permission, and only if authorised does it carry out the request. This interaction happens through special procedures called *system calls*.

The kernel provides system calls for all common operations a user process might need. For example, when a process wants to write to a file, it invokes the appropriate system call to request the kernel to perform the write. The kernel checks the user's permission, and only if allowed does it execute the write on the process's behalf. This happens quickly and transparently; if permission is denied, the kernel rejects the request, and the user sees a "permission denied" error.

What, then, is *kernel space*? Simply put, this refers to the set of operations that only the kernel can perform. When a user process needs to carry out such an operation, it must invoke a system call to request the kernel's service. The modern Linux kernel provides about 330 system calls, covering a wide range of functions. As a result, a regular user's access to kernel space is restricted to what these system calls permit. Chapter four of this book is dedicated to system calls.

## Summary

This chapter introduces the Linux kernel—the core of every Linux system. It explains how the kernel manages processes, interfaces with hardware, and enables programs to run. We discuss what it means for a process to "run" and how the kernel ultimately handles every command issued by a user. We explore the utility and complexity of reading the kernel source code. We also examine the distinction between kernel and user space, a separation that is theoretically sound and practically necessary. The chapter concludes with guidance on downloading the kernel source code, which will be used throughout the book.