# Linux Kernel Programming

*Developing kernel architecture and device drivers for character, block, USB, and network interfaces*

**THIERRY GAYET**

www.bpbonline.com

First Edition 2025

Copyright © BPB Publications, India

ISBN: 978-93-65897-364

**Dedicated to**

*My wife,* ***Yuniati Gayet***

*My children,* ***Angela Gayet*** *and* ***Luca Gayet***

# About the Author

**THIERRY GAYET** currently works as a senior developer for TIXEO, a company that has developed an ultra-secure French video conferencing system validated by the country's security authority.

He holds a DEST (postgraduate diploma) in computer science, specializing in artificial intelligence, which he obtained in the late 1990s from CNAM (University of Le Mans).

With over 30 years of experience in Linux development, he has extensively explored software development. His expertise has also extended to fields such as artificial intelligence, cybersecurity, and quantum physics.

Dynamic and dedicated to Linux and system hacking, he participates in repair cafés and continues to design systems based on embedded Linux or IoT.

In the 1990s, when the internet had not yet taken off as much as it does today, he created a Linux user association or group with the aim of helping to better understand and popularize the use of Linux.

Author of numerous articles in Linux Magazine and Misc, he has also presented numerous professional training sessions, conferences, and lectures in various topical areas. In this book, he shares his knowledge and insights to help you better understand the Linux kernel and develop your own drivers.

# About the Reviewers

- **Austin Kim** has more than 14 years of experience in embedded Linux **Board Support Package** (**BSP**) development. He has worked on many tasks such as board bring-up, crash and performance troubleshooting, and bootloader development for Arm-based devices. He has strong skills in reverse engineering and debugging binaries using tools like TRACE32, Crash-Utility, and ftrace.

  Currently, Austin works as a Linux kernel BSP engineer at LG Electronics. He enjoys sharing practical skills in reverse engineering and debugging through courses about Armv8-A, RISC-V architecture, and kernel crash debugging.

- **Martin Yanev** is a highly accomplished software engineer with nearly a decade of experience across diverse industries, including aerospace and medical technology. Over his illustrious career, Martin has carved a niche for himself in developing and integrating cutting-edge software solutions for critical domains such as air traffic control and chromatography systems. Renowned as an esteemed instructor and computer science professor at Fitchburg State University, he possesses a deep understanding of the full spectrum of OpenAI APIs and exhibits mastery in constructing, training, and fine-tuning AI systems. As a widely recognized author, Martin has shared his expertise to help others navigate the complexities of AI development. With his exceptional track record and multifaceted skill set, Martin continues to propel innovation and drive transformative advancements in the field of software engineering.

# Acknowledgement

I would like to express my sincere gratitude to everyone who contributed to the completion of this book, especially my wife, Yuniati, who encouraged me, and finally, my children, who helped and supported me throughout the writing process.

I am extremely grateful to BPB Publications for their guidance and expertise in the completion of this book. Their support and assistance were invaluable in navigating the complexities of the publication process.

I would also like to thank the reviewers, technical experts, and editors who provided valuable comments and helped improve this manuscript. Their ideas and suggestions significantly improved the quality of the book. Finally, I would like to express my gratitude to the readers who expressed interest in our book. Your support and encouragement were greatly appreciated.

Thank you to everyone who contributed to the completion of this book.

# Preface

Understanding the fundamentals of an operating system can be relatively complex. This book has taken on the challenge of popularizing and exploring the various concepts implemented within the GNU/Linux kernel.

Composed of 13 in-depth chapters, this book covers a wide range of topics essential to understanding this kernel.

We will begin with a brief introduction to the history and evolution of this kernel. Then, we will take a block-by-block look at the kernel structure itself, which will form the basis for the subsequent chapters.

Chapter 4 will describe in detail the device model used to structure files in sysfs and used within the kernel itself, among other things, by udev.

Chapter 5 will introduce us to the world of character drivers to develop drivers for keyboards, mice, and many other devices. Chapter 6 will cover another category of drivers: block drivers used for local or remote network mass storage. At this point, you will have the tools to develop your own file system if you wish.

Chapter 7 will then follow, describing how to develop drivers for your USB devices.

Chapter 8 will detail the intricacies of the network stack with the various hooks related to Netfilter. This will be followed by a detailed look at the LSM used to secure a Linux system through numerous system hooks.

The remaining chapters will finally detail memory, communications systems often linked to hardware, process management, and debugging.

Through practical examples, comprehensive explanations, and a structured approach, this book aims to provide the reader with a solid understanding of the GNU/Linux kernel. Whether you are a novice or an experienced user, I hope this book will be useful for exploring the fundamentals of this operating system and will help you develop it further by developing your own extensions or drivers.

**Chapter 1: History of the GNU/Linux Kernel -** This chapter provides a historical overview of the kernel's evolution, its inspirations, and the key players who brought this famous kernel to life.

**Chapter 2: Introduction to the Linux Kernel** - After providing a historical overview, the first part will provide a detailed presentation of the kernel architecture. This will then serve as a key chapter for the rest of the book.

**Chapter 3: Introduction to Device Drivers** - In the second part, we will explore the development of a Hello World driver and discuss various compilation techniques, inter-driver dependency management, and module loading and unloading.

**Chapter 4: Linux Device Model** - This chapter lays the foundation for the various registration and declaration processes between drivers, which will be used throughout the book.

**Chapter 5: Character Device Drivers** - This chapter covers the development of the first type of character-based driver.

**Chapter 6: Block Drivers and Virtual Filesystem** - This second type of block-based driver is used to manage SATA or network drives.

**Chapter 7: USB drivers and libusb** - This chapter details the drivers used to control devices via the USB protocol.

**Chapter 8: Network Drivers** - The network, at the heart of today's communications, will be discussed through the implementation of a network card driver. Details of the Netfilter architecture will also be presented for filtering or your own firewall.

**Chapter 9: Linux Security Modules** - Like all operating systems, maximum security is essential. This is what the NSA initiated with SELinux in late 2000 by proposing system-level hooks allowing precise access control. This chapter will provide a behind-the-scenes understanding and, perhaps, the opportunity to develop your own LSM.

**Chapter 10: Kernel Memory and DMA** - This chapter details the physical and virtual memory management modes. Direct or DMA access without going through the CPU will also be covered, including NUMA management.

**Chapter 11: Navigating Linux Communication Interfaces** - Whether it is an embedded device or a standard motherboard, different hardware communication protocols are used, such as I2C, to manage peripherals.

**Chapter 12: Process Management** - Since the GNU/Linux kernel is a multi-tasking, multi-user system, a scheduler is used to properly schedule the various tasks in the kernel or user space. Without it, the kernel would be single-tasking.

**Chapter 13: Debugging GNU/Linux Kernel and Drivers** - Developing is one thing, but being able to fine-tune a driver or a specific part of the kernel is another. Indeed, due to their relative complexity, the Linux kernel has seen the development of specific tools that complement each other. The reader will be given a brief overview of the various tools.

## Code Bundle and Coloured Images

Please follow the link to download the
***Code Bundle*** and the ***Coloured Images*** of the book:

**https://rebrand.ly/27joie2**

The code bundle for the book is also hosted on GitHub at
**https://github.com/bpbpublications/Linux-Kernel-Programming**.
In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

## Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

**errata@bpbonline.com**

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

# Table of Contents

# CHAPTER 1
# History of the GNU/Linux Kernel

## Introduction

This first chapter describes the origin and the evolution of the functionalities of the Linux kernel to better understand not only what the kernel is but also the way in which it has evolved structurally.

## Structure

The chapter covers the following topics:

- Inspirations
- Linux kernel history
- Evolutions of the several features by release/milestones
- License of the GNU/Linux

## Objectives

This chapter is an important preamble to understanding where Linux comes from, its history, and why we need to study this operating system, which owes its fame in part to the openness of its code, its robustness, and its performance. At the end of this chapter, you will have an overview of what the Linux kernel is. *Linux Torvald,* the inventor of

Linux, did not invent everything from scratch but was inspired by what existed at the time, whether he owned it or not. It is, therefore, useful to know these inspirations for a good understanding.

## Inspirations

In the 1960s and 1970s, Unix was an operating system developed at AT&T's Bell Labs. It was known for its modular design, stability, and efficiency. Unix (often written in capital letters, **UNIX**, particularly when it is the official trademark) is an operating system created by Bell Labs in 1969 as an interactive time-sharing system.

The name **Unix** is a play on words, combining **uniplexed** (meaning single or one at a time) and **multics** (referring to an earlier operating system project). Unix was developed as a successor to the Multics operating system.

*Ken Thompson* and *Dennis Ritchie* are considered the inventors of Unix.

In 1974, Unix became the first operating system developed in the C language. *Linus Torvalds*, the creator of the Linux kernel, was a Unix user during his university days and was inspired by Unix's concepts of multi-user, multitasking, and resource management (see *Figure 1.1*). The following figure shows us a concentric view or onion model of a Unix system:

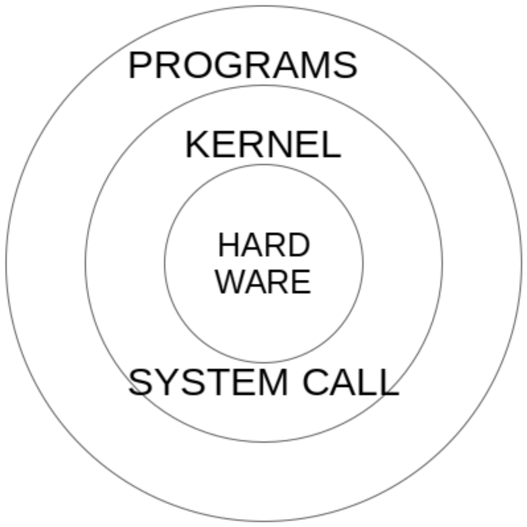

***Figure 1.1**: Onion view of the UNIX system*

Unix is often considered to be based on an onion model.

The model proposed by the Unix systems is layered, going from the hardware to the user via the kernel, which controls it and the software that provides utilities via several kinds of software. The following figure shows us a vertical view of the onion model:

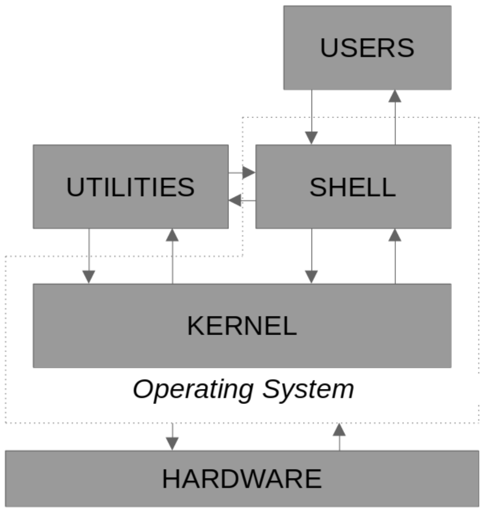

***Figure 1.2**: Another view of a UNIX system layered view*

View of the architecture of a Unix System is similar to an actual Linux in outline.

As we will explore, Linux distributions are numerous, and they inspire each other. With Unix systems, it was the same because when we talk about Unix systems (for those who knew them), we talk about IBM AIX, HP UX, Irix, Solaris, UnixWare, Ultrix, Microsoft Xenix, SCO Unix, Net/OpenBSD, and many others.

All these systems have evolved and contributed to developing the modern systems we know today in their own way. The following figure shows us the countless versions of Unix systems that were common at the time and that for the most part, have been replaced today by the Linux system:

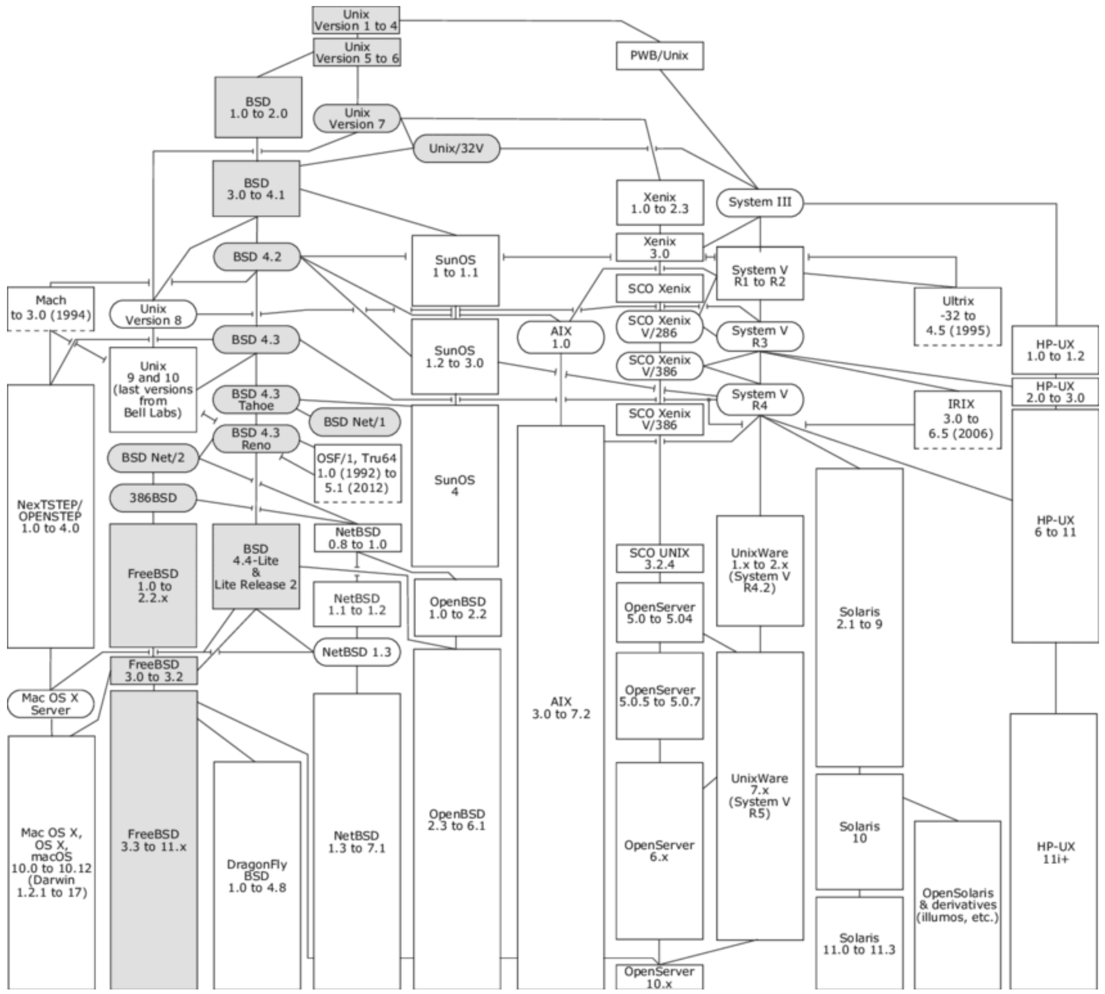

***Figure 1.3**: History and dependencies of Unix systems*

Many of the commands in 1972's Unix 2nd edition are still used in today's Linux.

Another inspiration was Minix (for mini-UNIX), an educational and experimental operating system, created by *Andrew S. Tanenbaum* (also known as AST) in 1984 for students to dissect a real OS. Indeed, AT&T had banned the use of Unix for educational purposes and forbids its use for teaching.

MINIX was a microkernel written in the C programming language to reimplement V7. It was designed for the IBM PC (with 256KB RAM, 360KB 5.25-inch floppy disk). It was structured in a more modular way than UNIX and is compatible with it from a user point of view, but completely different from the kernel side.

Its goals were to deliver an open-source, clean design that students can understand, a small microkernel, the rest of the OS is user processes, and communicate using synchronous message passing. Many of the basic programs, such as **cat, grep, ls, make, ...,** and the shell are present and perform the same functions as UNIX MINIX requires 20 MB of hard disk partition. MINIX is not as efficient as UNIX because it is designed to be readable. The following figure clearly shows the positioning of controllers in user space and no kernel, which will be seen by Linux as an inefficient approach in the following figure:

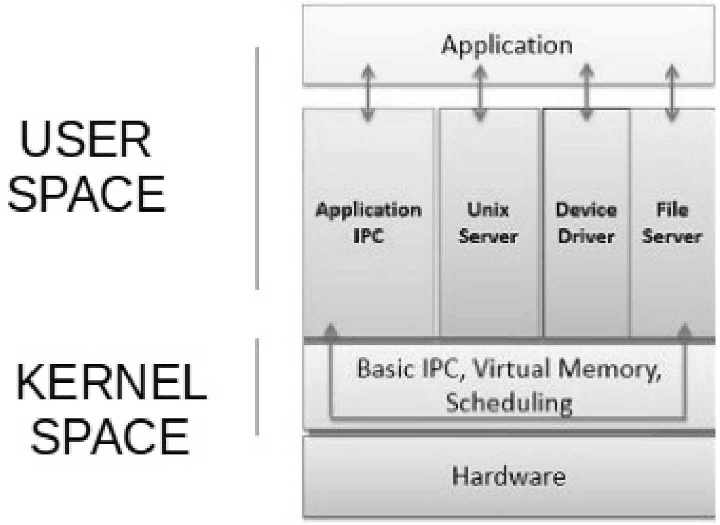

***Figure 1.4****: Architecture of a microkernel kernel*

The kernel is the core part of an operating system; it manages the system resources. It is like a bridge between the application and the hardware of the computer. In a monolithic kernel, user services and kernel services are both kept in the same address space.

*Linus Torvalds* worked on Minix for a period, gaining practical experience in operating system design. However, he found that Minix had limitations in terms of performance and features, which motivated him to create his own kernel, which gave birth to the Linux kernel a few years later. The initial version was crashing after an hour following a heat problem.

We should not forget one of the latest and important inspirations that made the Linux kernel what it has become today. This owes its current name to it, namely GNU/Linux, which derives from the rights of the GPL to which it is attached.

GNU is a free operating system created in 1983 by *Richard Stallman* (as shown in *Figure 1.5*), maintained by the GNU Project. GNU covers the concepts and operation of UNIX. This project already has a kernel called HURD, which is a play on words, as it is intended to sound like **herd** (as in a herd of animals) and represents a desire to move away from the Unix monolithic kernel model. It entered competition with Linux, which overshadowed it.