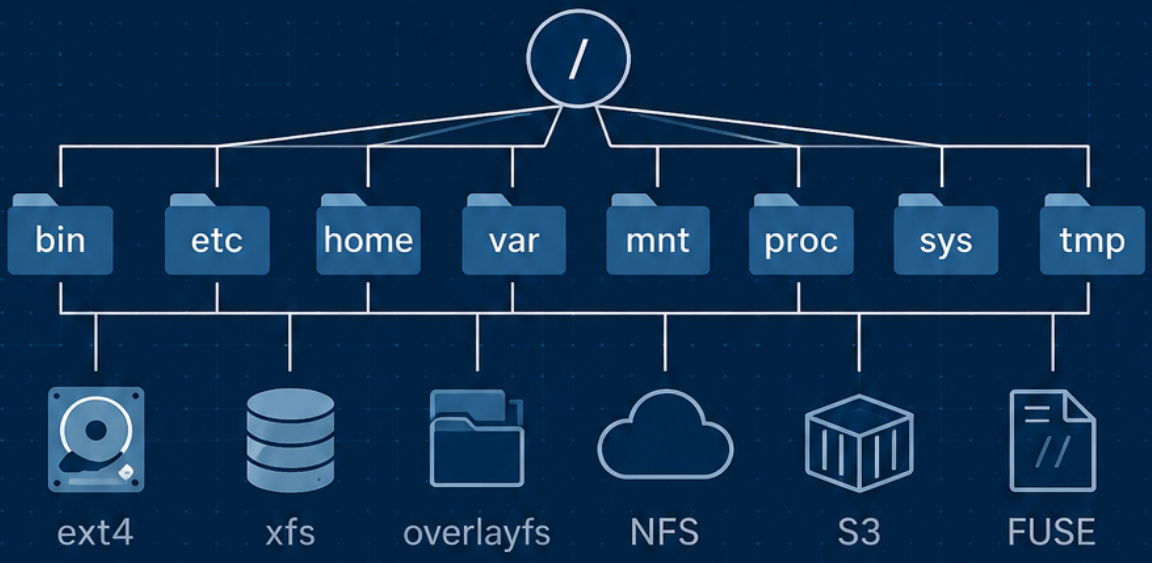
The Linux Virtual Filesystem

Architecture, Implementation, and Administration from Kernel Internals to Cloud-Native Systems



Jukka Palomaki

The Linux Virtual Filesystem

Architecture, Implementation, and Administration from Kernel Internals to Cloud-Native Systems

Steve T. Publications

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

This version was published on 2026-07-17

Contents

Architecture, Implementation, and Administration from Kernel Internals to Cloud-Native Systems

About This Book

The Linux Virtual Filesystem (VFS) is one of the most elegant abstractions in operating system design: a universal interface that allows dozens of disparate storage technologies to coexist under a single coherent API. This book takes you from foundational concepts through kernel internals to advanced topics like containerization, cloud-native storage, and modern async I/O. Whether you are a system administrator troubleshooting production filesystem issues, a developer building on FUSE or overlayfs, or a kernel engineer contributing to the VFS subsystem, this book provides the deep understanding needed to work with Linux filesystems at every level. The technical baseline assumes familiarity with the Linux 6.x kernel series.

Introduction: The Abstraction That Makes Linux Possible

Consider a simple command: `cat /etc/passwd`. You type those characters, press Enter, and the contents of a file appear on your screen. Behind that trivial interaction lies one of the most consequential engineering decisions in Unix history.

The path `/etc/passwd` is just a string. It carries no information about whether the file lives on an ext4 block device, an NFS share across the network, a tmpfs instance in RAM, or a FUSE-mounted encrypted volume. The `cat` program knows nothing of these distinctions. It opens the path with a system call, reads bytes, and writes them to stdout. Yet somewhere between the user-space library and the spinning platters (or SSD flash cells, or network packets), a sophisticated translation layer must resolve that string into actual data on some actual storage device.

That translation layer is the Virtual Filesystem, or VFS.

What Is a Virtual Filesystem?

The Linux Virtual Filesystem is an abstraction layer within the kernel. It provides two critical services: first, it presents a uniform filesystem interface to user-space programs through standard system calls like `open(2)`, `read(2)`, `write(2)`, and `stat(2)`. Second, it defines an internal kernel API that allows different filesystem implementations to coexist and interoperate. As Neil Brown wrote in his foundational 1999 paper, the VFS layer exists to ensure that "the kernel can manipulate any file system type through a common interface" [1].

The word "virtual" here does not mean fake or simulated. It means abstracted. The VFS is virtual in the same sense that a CPU's instruction set architecture is virtual: it presents a clean, consistent interface while hiding the complexity of what lies beneath. When you call `open("/var/log/syslog", O_RDONLY)`, the VFS intercepts that request, resolves the pathname through cached directory entries, locates the underlying inode, checks permissions, and dispatches the actual open operation to whatever filesystem driver manages that particular mount point. The application never needs to know whether syslog lives on XFS, Btrfs, or a network share.

This abstraction is not merely convenient. It is foundational. Without VFS, every program would need filesystem-specific code for every storage technology it might encounter. Kernel developers would face an impossible maintenance burden as new filesystems emerged. The VFS solves both problems simultaneously by defining a contract: any filesystem that implements the VFS interface can be mounted and used transparently by any application.

The Scale of the Problem

Linux supports more than forty distinct filesystem types out of the box. A typical production server might have ext4 for root, tmpfs for `/dev/shm`, procfs at `/proc`, sysfs at `/sys`, cgroup2fs at `/sys/fs/cgroup`, overlayfs for container storage, and NFS or CephFS for shared data. Each of these filesystems stores its data differently on disk (or in memory, or across a network), uses different

metadata formats, and has unique performance characteristics. Yet they all appear as a single unified directory tree to the user.

The VFS makes this possible through five core kernel objects: the superblock, which tracks filesystem-wide state; the inode, which holds file metadata; the dentry, which caches name-to-inode mappings; the file object, which represents an open file description; and the mount structure, which binds a filesystem tree to a directory in the namespace. These objects form a coherent abstraction that every filesystem driver must implement.

What makes VFS particularly powerful is its extensibility. New filesystems can be added as loadable kernel modules without recompiling the kernel. Pseudo-filesystems like `/proc` and `/sys` expose kernel data structures through the same file interface, making it possible to read process information or configure device drivers using standard shell commands. User-space filesystems built on FUSE (Filesystem in Userspace) can implement entirely custom storage backends while still appearing as regular mounted filesystems.

Why VFS Matters to You

You might be thinking: I am a system administrator, not a kernel developer. Why should I care about VFS internals?

The answer is that VFS touches everything you do with Linux systems, whether you realize it or not. When your container runtime fails to start because of a mount namespace conflict, the problem lives in the VFS layer. When a database experiences sudden write latency spikes due to dirty page writeback thresholds being exceeded, the mechanism is rooted in how the VFS interacts with the page cache and writeback subsystem. When you troubleshoot why `find` takes minutes to search a directory tree, the bottleneck may be dentry cache behavior or negative dentry accumulation. When you configure SELinux policies that block certain file accesses, the enforcement point is a VFS security hook.

Understanding VFS gives you a mental model for how Linux handles files at the deepest level. It transforms filesystem troubleshooting from guesswork to systematic diagnosis. It lets you read kernel documentation and mailing list discussions with comprehension rather than confusion. And if you ever need to contribute to the kernel or build a custom filesystem, it provides the foundation upon which everything else is built.

How This Book Is Organized

This book progresses from foundational concepts to advanced topics, with each chapter building on material from previous ones. The Introduction (this chapter) sets the stage. Chapter 1 covers the historical evolution of VFS in Linux and the design philosophy that shaped it. Chapter 2 dives deep into the five core VFS objects and how they relate to one another. Chapter 3 explains pathname resolution and the caching mechanisms that make file lookups fast.

Chapters 4 and 5 cover the system call interface and security model, explaining how user requests flow through VFS dispatch mechanisms and how access control is enforced at every level. Chapters 6 and 7 explore the pseudo-filesystems that power so much of Linux administration: `/proc`, `/sys`, tmpfs, devtmpfs, cgroupfs, debugfs, tracefs, securityfs, and configfs.

Chapters 8 and 9 focus on advanced virtual filesystems and containerization. Chapter 8 covers overlayfs and FUSE in detail, while Chapter 9 explains how mount namespaces, shared subtrees, and bind mounts enable the isolation that containers depend on. Chapters 10 and 11 address performance optimization and troubleshooting, covering the page cache, readahead, writeback, eBPF tracing, and practical diagnostic techniques.

The final chapters look outward: Chapter 12 examines how VFS powers cloud-native infrastructure, distributed filesystems, and virtualization backends. Chapter 13 surveys recent kernel developments including the new mount API, io_uring, Landlock sandboxing, and where the VFS subsystem is heading. The Conclusion ties everything together.

Throughout the book, you will find annotated C code excerpts from kernel source, shell command examples, configuration snippets, and references to specific kernel source files and documentation. The technical baseline assumes familiarity with Linux 6.x kernels, though historical context reaches back to earlier versions where relevant.

The VFS is not a static system. It evolves with every kernel release, adapting to new storage technologies, security requirements, and workload patterns. By the time you finish this book, you will understand not just how VFS works today, but why it works the way it does and where it is going next.

References

[1] Linux Kernel Documentation, “Overview of the Virtual File System,” https://docs.kernel.org/filesystems/vfs.html

[2] Neil Brown, “The Linux Virtual File-system Layer,” 1999. Referenced in kernel documentation at https://docs.kernel.org/filesystems/vfs.html

Chapter 1: Historical Evolution and Design Philosophy

The Linux Virtual Filesystem did not appear fully formed. It evolved through years of experimentation, debate, and incremental improvement in a kernel that was still finding its identity as a production-quality operating system. Understanding this history is essential because the design decisions made two decades ago continue to shape how VFS works today. Every structure, every callback, every locking convention has a story behind it.

Pre-VFS Unix Filesystem Landscape

Before the VFS existed, Unix filesystems were implemented as monolithic subsystems tightly coupled to the kernel. In early Unix systems (V7, BSD), the filesystem code was essentially one big implementation: the UFS (Unix File System). The kernel knew about inodes, blocks, and directories because there was only one format. If you wanted a different filesystem type, you had to modify the kernel source directly.

System V Release 4 (SVR4) introduced the concept of a virtual filesystem layer, inspired partly by the vnode architecture from BSD and Sun's original Unix implementations. The SVR4 VFS defined generic structures for superblocks, inodes, and directory entries, along with operation vectors that allowed different filesystem types to register their own implementations. This was the first serious attempt at filesystem abstraction in the Unix world, and it proved successful enough that many commercial Unix derivatives adopted similar architectures.

Plan 9 from Bell Labs took the abstraction even further. Its 9P network protocol essentially treated all files as remote objects, whether they lived on local storage or across a network. The Plan 9 approach influenced later Unix filesystem design, particularly in how it unified device access, process information, and regular files under a single namespace model.

Linux, in its early years (0.x through 1.2), had no such abstraction. The kernel supported minix filesystem natively, with ext added shortly after. Filesystem code was scattered across the kernel without a coherent interface layer. Adding a new filesystem type meant touching core kernel structures and hoping not to break existing ones. This worked fine when there were two or three filesystem types, but as Linux matured, the limitations became obvious.

Birth of VFS in Linux 1.3.x

The push for a proper VFS in Linux came from multiple directions simultaneously. The kernel needed to support commercial filesystems like XFS and JFS, which Sun and IBM were developing for their Unix offerings. Linux also needed to handle network filesystems (NFS client) more cleanly, and the growing number of filesystem-specific hacks in the kernel core was becoming unsustainable.

The first serious VFS patches appeared in the Linux 2.3 development series around 1999. Neil Brown authored a comprehensive paper titled "The Linux Virtual File-system Layer" that laid out the design principles [2]. Around the same time, Michael K. Johnson published "A tour of the Linux VFS," which documented what was already partially implemented [3]. Andrew Morton, who maintained the -mm (memory management) tree and served as a gatekeeper for patches entering the mainline kernel, played a crucial role in shepherding VFS development through review and integration. The VFS work was part of a broader modernization effort in the 2.3 series that also introduced preemptive scheduling, devfs, and ext3 journaling, all changes that required the kernel to handle filesystem operations more flexibly than before.

The early VFS implementation borrowed heavily from SVR4's vnode model but adapted it to Linux conventions. The key insight was that filesystem operations could be organized around a small set of generic objects (superblock, inode, dentry, file) with callback vectors pointing to filesystem-specific implementations. This design allowed the kernel to dispatch any file operation through a uniform path: the VFS layer receives the request, looks up the appropriate object, follows the operation vector to the filesystem driver, and returns the result.

By the time Linux 2.4 shipped in January 2001, VFS was mature enough to support ext2, ext3, minix, NFS, ISO9660, FAT, and several other filesystem

types through a common interface. The foundation was solid, though many refinements would follow over the next two decades.

Design Principles: Abstraction, Extensibility, Minimalism

Three principles guided VFS design from the beginning and continue to govern its evolution:

Abstraction. The VFS presents a uniform interface that hides filesystem-specific details. User-space programs never see the difference between ext4 and XFS. Kernel subsystems that interact with the filesystem (memory management, security, namespaces) work through VFS abstractions without knowing the underlying storage technology. This abstraction is enforced by strict type discipline: filesystem drivers implement well-defined structures (`struct super_operations`, `struct inode_operations`, `struct file_operations`) and the VFS layer dispatches to them through function pointers.

Extensibility. New filesystem types can be added as loadable kernel modules without modifying the VFS core or recompiling the kernel. A filesystem driver registers itself with `register_filesystem()`, providing a `struct file_system_type` that describes its capabilities and callback functions. The VFS maintains a registry of known filesystem types and uses it during mount operations to find the appropriate driver. This extensibility is what enables the rich ecosystem of Linux filesystems, from mainstream options like Btrfs and ZFS (via FUSE) to specialized filesystems for embedded systems, databases, and security applications.

Minimalism. The VFS does as little work as possible in the common path. It provides the glue between user requests and filesystem drivers but avoids adding unnecessary overhead. Operations that are filesystem-agnostic (like dentry cache lookups) are handled by the VFS directly. Operations that require filesystem-specific knowledge (like reading an inode from disk) are delegated through callback vectors. This minimalism keeps the common path fast while preserving flexibility for edge cases.

These principles are not always easy to reconcile. Abstraction can introduce overhead. Extensibility requires careful interface design. Minimalism sometimes means pushing complexity into filesystem drivers rather than centralizing it in the VFS core. The resulting architecture is a pragmatic compromise,

and understanding where the compromises were made helps explain many of VFS's quirks and conventions.

The VFS as a Kernel Subsystem

The VFS does not exist in isolation. It interacts with nearly every other kernel subsystem, and these interactions shape its behavior in ways that matter for both performance and correctness. Understanding these interdependencies is essential for diagnosing complex issues where the root cause lives outside the filesystem layer itself.

The page cache, part of the memory management subsystem, stores cached file data. The VFS uses the page cache through `struct address_space_operations`, which defines how a filesystem reads pages from disk and writes them back. Writeback of dirty pages is coordinated between the VFS, the page cache, and the block layer through a complex dance of thresholds, timers, and per-CPU writer throttling. When a database administrator sees write latency spikes, the mechanism is often rooted in how the VFS interacts with the page cache and writeback subsystem.

Physical filesystems read and write data through the block I/O subsystem. The VFS translates file offsets into block numbers, then submits I/O requests to the block layer through bio (block I/O) structures. Network filesystems bypass the block layer entirely, communicating directly through socket interfaces, which means their performance characteristics and failure modes differ fundamentally from local storage.

The Linux Security Module (LSM) framework inserts hooks throughout the VFS call path to enforce mandatory access control policies. Every file open, permission check, and attribute change passes through LSM hooks before reaching the filesystem driver. A permission denial that puzzles an administrator might originate in an LSM hook rather than traditional Unix permissions, and diagnosing the difference requires understanding where security checks sit in the VFS dispatch chain.

Mount namespaces isolate the view of the filesystem hierarchy per process group. The VFS maintains separate mount trees for each namespace, and mount propagation controls how changes in one namespace affect others. This is the foundation of container filesystem isolation, and mount-related bugs are among the most common container issues encountered in production.

File descriptors, open file descriptions, and current working directories are all per-process or per-thread resources managed through VFS structures. The `struct files_struct` in the task structure holds the file descriptor table, which maps to `struct file` objects in the VFS. When an application exhausts its file descriptor limit, the failure manifests as `EMFILE` errors at the VFS layer, even though the root cause may be a resource leak in application code.

Understanding these interactions is crucial for diagnosing complex issues. A filesystem performance problem might actually be a page cache sizing issue. A mount namespace conflict in a container might stem from shared subtree propagation settings. The VFS sits at the center of all of this, and its abstractions provide the common vocabulary for reasoning about these interactions.

Lessons from Other Operating Systems

Linux VFS did not develop in a vacuum. Several other operating systems contributed ideas:

SVR4 vnodes: System V Release 4's vnode architecture provided the initial blueprint. The concept of operation vectors (vnodeops in SVR4, mapped to `file_operations`, `inode_operations`, etc. in Linux) is directly descended from this design. However, Linux simplified the approach by using fewer vnode types and relying more on callback dispatch than type-specific code paths.

Plan 9: The Plan 9 filesystem model, where everything is a file accessible through the 9P protocol, influenced how Linux treats pseudo-filesystems. The idea that process information (`/proc`), device nodes (`/dev`), and kernel configuration (`/sys`) can all be exposed through the same file interface owes much to Plan 9's philosophy.

Solaris dnodes: Solaris's directory node (dnode) concept, which separates naming from metadata, influenced Linux's dentry/inode split. The dentry cache in Linux serves a similar purpose: caching name-to-inode mappings to avoid expensive filesystem lookups on every path resolution.

FreeBSD VFS: FreeBSD's virtual filesystem implementation, which evolved from BSD and SVR4 foundations, shares many design similarities with Linux VFS. The two implementations diverged in specific areas (locking, namespace handling, mount propagation), but the core architecture remains comparable.

Cross-pollination between the communities has been significant, particularly around shared subtrees and mount namespace concepts.

Linux VFS ultimately became its own thing: not a direct copy of any predecessor, but an original implementation that absorbed useful ideas while staying true to Linux conventions and development practices. The result is an abstraction layer that has proven remarkably durable, flexible, and performant over more than twenty-five years of continuous evolution.

References

[1] Neil Brown, “The Linux Virtual File-system Layer,” 1999. Referenced in kernel documentation at https://docs.kernel.org/filesystems/vfs.html [2] Michael K. Johnson, “A tour of the Linux VFS,” 1996. Available at https://tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html

Chapter 2: Core VFS Objects: The Building Blocks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

The Superblock: Filesystem-Wide State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

The Inode: Metadata Without a Name

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

The Dentry: Naming and Caching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

The File Object: Open File Description

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

The Mount Structure: Binding Filesystems to Directories

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

How the Objects Fit Together

Chapter 3: Pathname Resolution and Caching Mechanisms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

The Lookup Algorithm: Walking a Path Component by Component

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

The Dentry Cache (dcache): Structure and Hash Table Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Negative Dentries and Caching Nonexistence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Inode Caching and the Icache

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Follow Symlinks, Bind Mounts, and Special Cases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Performance Characteristics: When Path Resolution Becomes a Bottleneck

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Chapter 4: System Calls and the VFS Interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

The Syscall Entry Path: From User Space to VFS Dispatch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

File Operations Vector (`struct file_operations`)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Inode Operations Vector (`struct inode_operations`)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Superblock Operations Vector (`struct super_operations`)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Dentry Operations Vector (`struct dentry_operations`)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Key System Calls Walked Through

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Chapter 5: Security Model and Permissions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Discretionary Access Control: Traditional Unix Permissions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

The Permission Checking Path in VFS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Linux Security Modules (LSM) Hooks in VFS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Capabilities and Fine-Grained Privilege Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Extended Attributes, ACLs, and Security Labels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

SELinux, AppArmor, and Smack Integration with VFS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Chapter 6: Pseudo-Filesystems I: /proc, /sys, tmpfs, and devtmpfs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

What Makes a Filesystem "Pseudo"?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

/proc: The Process Information Filesystem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

/sys: The Sysfs Hierarchy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

tmpfs: Memory-Backed RAM Filesystem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

devtmpfs: Automatic Device Node Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Chapter 7: Pseudo-Filesystems II: cgroupfs, debugfs, tracefs, securityfs, configfs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

cgroupfs: Control Groups Filesystem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

debugfs: Kernel Debugging Interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

tracefs: Tracing Infrastructure Filesystem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

securityfs: LSM Configuration Interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

configfs: Dynamic Object Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Chapter 8: Advanced Virtual Filesystems: overlayfs and FUSE

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

overlayfs: Union Mount Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

overlayfs Implementation Details

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

FUSE: User-Space Filesystem Framework

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

FUSE Performance and Limitations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

overlayfs Versus btrfs Subvolumes: A Structured Comparison

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Real-World Case Studies

Chapter 9: Namespaces, Mounts, and Containerization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Mount Namespaces: Isolating the View of the Filesystem Hierarchy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Private, Shared, Slave, and Unbindable Mount Propagation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Bind Mounts and Their Role in Container Storage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

The Mount Tree Data Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Containers and VFS: Docker, Podman, and Kubernetes Perspectives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Container Storage Driver Comparison

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Overlayfs as the Default Container Storage Driver

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Chapter 10: Performance, Scalability, and Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

The Page Cache: Where Most File I/O Actually Happens

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Read-Ahead Algorithms and Prefetching Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Writeback and the Dirty Page Management Subsystem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Direct I/O and Bypassing VFS Caching (O_DIRECT, io_uring)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Scalability Challenges: Large Directories, Namespace Storms, Lock Contention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Tuning Parameters and Performance Diagnostics

Chapter 11: Troubleshooting, Debugging, Tracing, and Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Syscall Tracing with strace and Performance Analysis with perf

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Kernel Tracing: ftrace, tracefs, and Built-In VFS Traces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

eBPF for VFS Diagnostics (kprobes, tracepoints, BCC tools)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Common Failure Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Mount Options for Debugging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Monitoring Filesystem Health and Performance in Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Chapter 12: Cloud-Native Use Cases and Virtualization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Container Storage Orchestration: CSI, Persistent Volumes, and VFS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Network Filesystems Through the VFS Lens

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Virtualization Filesystem Backends (virtiofs, 9p, VMware vmhgfs)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Ephemeral Storage Patterns in Kubernetes and Serverless

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Cloud Provider Filesystems: EBS, GCE Persistent Disks, Azure Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Distributed Filesystem Challenges at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Chapter 13: Recent Kernel Developments and Future Directions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

The New Mount API: fs_context, fs_parameter, open_tree, move_mount

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

io_uring and Asynchronous File I/O Through VFS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Landlock: Unprivileged Process Sandboxing via Pathnames

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

statx and Extended File Attributes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

BPF Filesystem Hooks and Runtime Security

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Where VFS Is Heading: Trends, Open Questions, and Community Directions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

Conclusion: The Living Abstraction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/thelinuxvirtualfilesystem.