Linux Disk Management & RAID Configuration

Designing, Configuring, and Maintaining Reliable Storage Systems on Linux

Preface

Every Linux system, no matter how elegantly its applications are designed or how carefully its network is secured, ultimately depends on one foundational layer: **storage**. Data must be written, read, protected, and recovered — reliably, efficiently, and predictably. Yet for many Linux administrators and engineers, disk management and RAID configuration remain areas of uncertainty, approached with caution or, worse, with assumptions inherited from outdated practices.

This book was written to change that.

Purpose and Scope

Linux Disk Management & RAID Configuration is a comprehensive, hands-on guide to designing, configuring, and maintaining reliable storage systems on Linux. Whether you are a system administrator managing a handful of servers, a DevOps engineer architecting infrastructure at scale, or a student preparing for a career in Linux systems engineering, this book provides the knowledge and practical skills you need to make confident, informed decisions about how data is stored and protected on Linux platforms.

The scope spans the full lifecycle of Linux storage — from understanding how the kernel interacts with block devices, through partitioning and filesystem creation, to building sophisticated storage architectures that combine software RAID and Logical Volume Management (LVM). Critically, this book doesn't stop at configuration. It dedicates significant attention to **monitoring, optimization, trou-**

bleshooting, and recovery — the skills that separate capable administrators from exceptional ones.

Key Themes

Several themes run throughout these pages:

- **Architecture before action.** Every chapter emphasizes understanding *why* before learning *how*. You'll learn to think about Linux storage as a layered system, making decisions that are deliberate rather than reactive.
- **Practical, real-world application.** Commands, configurations, and examples are drawn from production Linux environments. This is not a theoretical exercise — it's a working reference.
- **Resilience and reliability.** From RAID design to backup strategies to failure recovery playbooks, the book treats data protection not as an afterthought but as a core design principle.
- **Progressive complexity.** The material builds methodically, beginning with foundational concepts in Linux storage architecture and advancing through LVM, RAID, combined configurations, and ultimately production-grade storage best practices.

How This Book Is Organized

The book is structured in a logical progression across **sixteen chapters and five appendices**. Chapters 1–4 establish the fundamentals of Linux storage: architec-

ture, disk layout planning, partitioning, and filesystem management. Chapters 5–6 dive deep into LVM, giving you full command of logical volume creation, resizing, and management. Chapters 7–10 cover RAID comprehensively — from foundational theory and design considerations to building and monitoring software RAID arrays with `mdadm` on Linux. Chapter 11 brings these two powerful subsystems together, showing how RAID and LVM can be combined for maximum flexibility and resilience. Chapters 12–15 address the operational realities of performance tuning, backup and recovery, troubleshooting failures, and implementing storage best practices in production Linux environments. Chapter 16 closes with a forward-looking perspective on evolving from disk management to broader storage architecture thinking.

The appendices serve as lasting references: command cheat sheets, RAID level comparisons, example storage layouts, a disk failure recovery playbook, and a curated Linux storage learning roadmap for continued growth.

Who This Book Is For

If you work with Linux — or aspire to — and you want to move beyond surface-level familiarity with disks and partitions toward genuine mastery of storage systems, this book is for you. Prior experience with the Linux command line is assumed; expertise in storage is not.

Acknowledgments

This book owes a debt to the broader Linux and open-source community — the kernel developers, the maintainers of tools like `mdadm`, `lvm2`, and `parted`, and the

countless contributors who have documented, debugged, and improved Linux storage over decades. I am also grateful to the technical reviewers whose sharp eyes and honest feedback strengthened every chapter, and to the readers of early drafts whose questions shaped the book's clarity and direction.

Data is the lifeblood of every system. The storage layer is where that lifeblood is kept safe. Let's build it right.

Bas van den Berg

Table of Contents

Page

Chapter 1: Understanding Linux Storage Architecture

The foundation of every reliable Linux system rests upon its storage architecture. Whether you are managing a single desktop machine or orchestrating a fleet of enterprise servers, understanding how Linux perceives, organizes, and interacts with storage devices is not merely helpful but essential. This chapter takes you on a thorough journey through the layers of Linux storage architecture, from the physical hardware spinning beneath the chassis to the abstract file systems that present data to users and applications. By the end of this chapter, you will possess a mental model of storage in Linux that will serve as the bedrock for every advanced topic that follows in this book, including partitioning, logical volume management, and RAID configuration.

The Physical Layer: How Linux Sees Hardware

When a storage device is connected to a Linux system, whether it is a traditional spinning hard disk drive, a solid state drive, or a network attached storage volume, the Linux kernel must first detect it and make it available to the rest of the operating system. This process begins at the hardware level and moves upward through a series of well defined abstractions.

At the lowest level, storage devices communicate with the system through hardware interfaces. The most common interfaces encountered in modern Linux

environments include SATA (Serial Advanced Technology Attachment), SAS (Serial Attached SCSI), NVMe (Non-Volatile Memory Express), and USB. Each of these interfaces has distinct characteristics that affect performance, reliability, and how the kernel interacts with the device.

The following table provides a comprehensive comparison of common storage interfaces found in Linux systems:

Interface	Typical Use Case	Maximum Throughput	Protocol	Kernel Driver Subsystem
SATA III	Desktop and consumer storage	6 Gbps	AHCI	libata
SAS	Enterprise servers and storage arrays	12 Gbps (SAS-3)	SCSI	SCSI midlayer
NVMe	High performance SSDs	32 Gbps (PCIe 4.0 x4)	NVMe over PCIe	nvme
USB 3.2	External and portable storage	20 Gbps (Gen 2x2)	USB Mass Storage / UAS	usb-storage / uas
Fibre Channel	SAN environments	128 Gbps (64GFC)	FCP	SCSI midlayer
iSCSI	Network attached block storage	Network dependent	SCSI over TCP/IP	SCSI midlayer + open-iscsi

When the Linux kernel boots, or when a device is hot-plugged into a running system, the kernel's device discovery mechanism springs into action. The kernel probes the hardware buses, identifies connected devices, loads the appropriate driver modules, and creates device nodes in the `/dev` directory. This entire process is facilitated by the udev subsystem, which is the userspace device manager responsible for dynamically creating and removing device nodes.

You can observe this process in real time by monitoring the kernel ring buffer. When you connect a new SATA drive, for example, you might see output like the following:

```
dmesg | tail -20
```

A typical output might include:

```
[  234.567890] ata3: SATA link up 6.0 Gbps (SStatus 133 SControl
300)
[  234.568012] ata3.00: ATA-9: Samsung SSD 870 EVO 1TB, SVT02B6Q,
max UDMA/133
[  234.568234] ata3.00: 1953525168 sectors, multi 1: LBA48 NCQ
(depth 32), AA
[  234.569456] ata3.00: configured for UDMA/133
[  234.569789] scsi 2:0:0:0: Direct-Access     ATA      Samsung
SSD 870  02B6 PQ: 0 ANSI: 5
[  234.570123] sd 2:0:0:0: [sdb] 1953525168 512-byte logical
sectors (1.00 TB/931 GiB)
[  234.570456] sd 2:0:0:0: [sdb] Write Protect is Off
[  234.570789] sd 2:0:0:0: [sdb] Write cache: enabled, read
cache: enabled, doesn't support DPO or FUA
[  234.571234] sd 2:0:0:0: [sdb] Attached SCSI disk
```

This output reveals the entire chain of discovery. The kernel detects the SATA link, identifies the device model and capabilities, assigns it to the SCSI subsystem (since Linux treats most block devices through a unified SCSI layer), and finally creates the block device node `/dev/sdb`.

Note: Even SATA and NVMe devices pass through the SCSI layer in Linux. This is a deliberate design choice that provides a unified interface for block device operations. NVMe devices, however, receive their own naming convention (`/dev/nvme0n1` instead of `/dev/sdX`) because they use a dedicated driver subsystem that bypasses the traditional SCSI midlayer for performance reasons.

Block Devices and Device Naming Conventions

In Linux, storage devices are represented as block devices. A block device is a type of device file that provides buffered access to hardware, reading and writing data in fixed-size blocks rather than as a continuous stream of bytes. This is in contrast to character devices, such as serial ports or terminals, which handle data one byte at a time.

Every block device in Linux appears as a special file under the `/dev` directory. The naming conventions for these devices follow predictable patterns that, once understood, make it straightforward to identify what type of device you are working with.

The following table explains the naming conventions used by the Linux kernel for different types of storage devices:

Device Pattern	Description	Example	Partition Notation
`/dev/sd[a-z]`	SCSI, SATA, SAS, and USB disks	`/dev/sda`	`/dev/sda1`, `/dev/sda2`
`/dev/nvme[0-9]n[0-9]`	NVMe solid state drives	`/dev/nvme0n1`	`/dev/nvme0n1p1`, `/dev/nvme0n1p2`
`/dev/vd[a-z]`	Virtio disks in virtual machines	`/dev/vda`	`/dev/vda1`, `/dev/vda2`
`/dev/xvd[a-z]`	Xen virtual block devices	`/dev/xvda`	`/dev/xvda1`, `/dev/xvda2`
`/dev/hd[a-d]`	Legacy IDE/PATA disks (rare today)	`/dev/hda`	`/dev/hda1`, `/dev/hda2`
`/dev/md[0-9]`	Software RAID arrays	`/dev/md0`	`/dev/md0p1` (if partitioned)

`/dev/dm-[0-9]`	Device mapper devices (LVM, LUKS)	`/dev/dm-0`	Accessed via mapper names
`/dev/loop[0-9]`	Loop devices (files as block devices)	`/dev/loop0`	`/dev/loop0p1`

To list all block devices currently recognized by the system, the `lsblk` command is invaluable:

```
lsblk
```

A sample output might look like this:

```
NAME         MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda            8:0    0 465.8G  0 disk
├─sda1         8:1    0   512M  0 part /boot/efi
├─sda2         8:2    0     1G  0 part /boot
└─sda3         8:3    0 464.3G  0 part
  ├─vg0-root 253:0    0    50G  0 lvm  /
  ├─vg0-swap 253:1    0     8G  0 lvm  [SWAP]
  └─vg0-home 253:2    0 406.3G  0 lvm  /home
sdb            8:16   0 931.5G  0 disk
nvme0n1      259:0    0 476.9G  0 disk
├─nvme0n1p1 259:1    0   512M  0 part
└─nvme0n1p2 259:2    0 476.4G  0 part
```

This output demonstrates several important concepts simultaneously. You can see a SATA disk (`sda`) that has been partitioned and further divided using LVM (Logical Volume Manager). A second SATA disk (`sdb`) appears with no partitions, meaning it is either new or has been wiped. An NVMe drive (`nvme0n1`) is also present with two partitions. The hierarchical relationship between disks, partitions, and logical volumes is immediately visible.

For more detailed information about block devices, including their UUIDs, file system types, and labels, you can use the `blkid` command:

```
sudo blkid
```

```
/dev/sda1: UUID="A1B2-C3D4" TYPE="vfat" PARTLABEL="EFI System
Partition" PARTUUID="12345678-abcd-efgh-ijkl-123456789abc"
/dev/sda2: UUID="abcdef01-2345-6789-abcd-ef0123456789"
TYPE="ext4" PARTUUID="23456789-bcde-fghi-jklm-234567890bcd"
/dev/sda3: UUID="bcdef012-3456-789a-bcde-f01234567890"
TYPE="LVM2_member" PARTUUID="34567890-cdef-ghij-
klmn-345678901cde"
/dev/mapper/vg0-root: UUID="cdef0123-4567-89ab-cdef-012345678901"
TYPE="ext4"
/dev/mapper/vg0-swap: UUID="def01234-5678-9abc-def0-123456789012"
TYPE="swap"
/dev/mapper/vg0-home: UUID="ef012345-6789-abcd-ef01-234567890123"
TYPE="xfs"
```

Note: The UUID (Universally Unique Identifier) is critically important in Linux storage management. Unlike device names such as `/dev/sda` which can change between boots depending on detection order, UUIDs remain constant. This is why modern Linux distributions use UUIDs in `/etc/fstab` for mounting file systems rather than device names.

The Linux Storage Stack: From Hardware to File System

Understanding the Linux storage stack requires appreciating that data passes through multiple layers between the application and the physical disk. Each layer adds functionality, abstraction, or both. Grasping this layered architecture is crucial because problems at any layer can affect the layers above it, and performance tuning often requires adjustments at specific layers.

The storage stack in Linux, from top to bottom, consists of the following layers:

Layer	Component	Purpose	Key Tools
Application Layer	User applications and services	Read and write files	Any application
VFS Layer	Virtual File System	Provides unified interface for all file systems	N/A (kernel internal)
File System Layer	ext4, XFS, Btrfs, ZFS, etc.	Organizes data into files and directories	mkfs, tune2fs, xfs_info
Page Cache	Kernel memory management	Caches frequently accessed data in RAM	vmstat, free, /proc/meminfo
Block Layer	I/O scheduler, block device interface	Manages and optimizes I/O requests	iostat, blktrace, /sys/block
Device Mapper	LVM, dm-crypt, dm-raid, multipath	Provides virtual block devices	dmsetup, lvm, cryptsetup
SCSI / NVMe Layer	SCSI midlayer, NVMe driver	Communicates with storage controllers	sg_inq, nvme-cli
Hardware Layer	Physical disks, controllers, interfaces	Stores data persistently	smartctl, hdparm

Let us walk through what happens when an application writes data to a file. Suppose a database process issues a write call. The request first passes through the Virtual File System (VFS), which is the kernel's abstraction layer that allows Linux to support dozens of different file systems through a single, consistent API. The VFS determines which file system the target file resides on and dispatches the request to the appropriate file system driver.

The file system driver, whether it is ext4, XFS, or Btrfs, translates the file-level operation into block-level operations. It determines which blocks on the disk need to be written, updates metadata structures such as inodes and allocation bitmaps, and submits the block I/O requests to the block layer.

Before reaching the disk, the block layer's I/O scheduler may reorder, merge, or prioritize the requests to optimize throughput and latency. For traditional spinning disks, schedulers like `mq-deadline` attempt to minimize seek time by grouping nearby requests. For SSDs and NVMe devices, the `none` (noop) scheduler is often preferred since these devices have no mechanical seek penalty.

You can check and change the I/O scheduler for a device using the sysfs interface:

```
# Check the current scheduler for sda
cat /sys/block/sda/queue/scheduler

[mq-deadline] kyber bfq none

# Change the scheduler to bfq
echo bfq > /sys/block/sda/queue/scheduler
```

If Device Mapper is involved, such as when LVM or disk encryption is in use, there is an additional layer of translation. The Device Mapper takes virtual block addresses and maps them to physical block addresses on one or more underlying devices. This is what allows LVM to span a logical volume across multiple physical disks, or LUKS encryption to transparently encrypt all data before it reaches the disk.

The sysfs and procfs Interfaces for Storage

Linux exposes an extraordinary amount of information about storage devices through its virtual file systems, particularly `/sys` (sysfs) and `/proc` (procfs). These interfaces are not mere diagnostic tools; they are the primary mechanism through

which administrators and automation scripts query and configure storage behavior at runtime.

The `/sys/block/` directory contains a subdirectory for each block device recognized by the kernel. Within each device directory, you will find a wealth of information:

```
# List all recognized block devices
ls /sys/block/

sda  sdb  nvme0n1  dm-0  dm-1  dm-2  loop0

# View the size of sda in 512-byte sectors
cat /sys/block/sda/size

976773168

# View the device model
cat /sys/block/sda/device/model

Samsung SSD 870

# View the rotation flag (0 = SSD, 1 = HDD)
cat /sys/block/sda/queue/rotational

0
```

This last command is particularly useful in scripts that need to apply different configurations based on whether a device is a spinning disk or a solid state drive. Many system tuning tools, including the `tuned` daemon, use this flag to automatically select appropriate I/O schedulers and read-ahead values.

The `/proc/partitions` file provides a quick summary of all partitions known to the kernel:

```
cat /proc/partitions

major minor  #blocks  name

   8        0  488386584 sda
   8        1     524288 sda1
   8        2    1048576 sda2
   8        3  486813720 sda3
   8       16  976762584 sdb
 259        0  500107608 nvme0n1
 259        1     524288 nvme0n1p1
 259        2  499583320 nvme0n1p2
 253        0   52428800 dm-0
 253        1    8388608 dm-1
 253        2  425996312 dm-2
```

The major and minor numbers shown here are the kernel's internal identifiers for block devices. The major number identifies the driver responsible for the device (8 for SCSI/SATA devices, 259 for NVMe, 253 for device mapper), while the minor number identifies the specific device or partition within that driver's domain.

Practical Exploration: Mapping Your System's Storage

Now that you understand the theoretical foundations, it is time to put this knowledge into practice. The following exercise walks you through a comprehensive exploration of your own Linux system's storage architecture.

Exercise 1: Complete Storage Inventory

Begin by creating a complete inventory of all storage devices on your system. Execute each command and record the results.

Step 1: List all block devices with full detail.

```
lsblk -o NAME,TYPE,SIZE,FSTYPE,MOUNTPOINT,MODEL,SERIAL,ROTA,DISC-
MAX
```

This command displays the device name, type, size, file system type, mount point, hardware model, serial number, whether it is rotational, and the maximum discard (TRIM) size. The ROTA column is particularly important: a value of 1 indicates a spinning hard drive, while 0 indicates a solid state drive. The DISC-MAX column shows whether the device supports TRIM operations, which is essential for SSD health and performance.

Step 2: Examine the kernel's view of storage controllers.

```
lspci | grep -i -E "storage|sata|nvme|scsi|raid"
```

This command queries the PCI bus for all storage-related controllers. You might see output like:

```
00:17.0 SATA controller: Intel Corporation Cannon Lake PCH SATA
AHCI Controller (rev 10)
01:00.0 Non-Volatile memory controller: Samsung Electronics Co
Ltd NVMe SSD Controller SM981/PM981/PM983
```

Step 3: Verify that the correct kernel modules are loaded for your storage devices.

```
lsmod | grep -i -E "ahci|nvme|scsi|sd_mod|dm_mod"
```

```
nvme                   45056  2
nvme_core              98304  5 nvme
ahci                   40960  1
libahci                32768  1 ahci
sd_mod                 57344  5
scsi_mod              253952  5 sd_mod,libata,sg,sr_mod,ahci
dm_mod                155648  9
```

Step 4: Examine the device mapper table if LVM or encryption is in use.

```
sudo dmsetup ls --tree
```

This command displays the device mapper hierarchy in a tree format, showing how virtual devices map to physical devices.

Step 5: Check the health of your storage devices using SMART data.

```
sudo smartctl -a /dev/sda
```

The SMART (Self-Monitoring, Analysis, and Reporting Technology) data provides insight into the physical health of the drive, including temperature, error counts, power-on hours, and predictive failure indicators. If `smartctl` is not installed, you can install it from the `smartmontools` package:

```
# On Debian/Ubuntu
sudo apt install smartmontools

# On RHEL/CentOS/Fedora
sudo dnf install smartmontools
```

Exercise 2: Understanding Device Relationships

This exercise helps you understand how devices, partitions, and logical volumes relate to each other.

Step 1: Create a visual map of your storage hierarchy.

```
lsblk -f
```

Step 2: For each mounted file system, identify the complete chain from mount point to physical device.

```
# Find the device behind a mount point
df -h /home

# If it is an LVM volume, find the physical device
sudo lvs -o +devices

# If it is a RAID array, find the member devices
cat /proc/mdstat
```

Step 3: Document the UUIDs of all your storage devices and verify they match your `/etc/fstab` entries.

```
sudo blkid
cat /etc/fstab
```

Compare the UUIDs from `blkid` with those referenced in `/etc/fstab`. Any mismatch could prevent your system from booting correctly or mounting file systems at startup.

Note: Always keep a backup of your `/etc/fstab` file before making changes. An incorrect fstab entry can render your system unbootable. If this happens, you can boot from a live USB and correct the file.

Key Concepts to Carry Forward

Before proceeding to the next chapter, ensure that you have internalized the following concepts, as they will be referenced repeatedly throughout this book:

Concept	Why It Matters
Block devices are accessed through `/dev`	All disk operations in Linux target device files
Device names can change between boots	Always use UUIDs or labels for persistent identification
The storage stack has multiple layers	Performance issues and failures can originate at any layer
The kernel provides rich introspection via sysfs	You can query and tune storage behavior at runtime without rebooting
Device Mapper provides virtualization	LVM, encryption, and software RAID all depend on this subsystem

I/O schedulers affect performance	Different workloads and device types benefit from different schedulers
SMART monitoring detects failing drives	Proactive monitoring prevents data loss

This chapter has established the vocabulary and mental model you need to work effectively with Linux storage. You now understand how the kernel discovers and represents storage hardware, how data flows through the storage stack, and how to interrogate your system to understand its current storage configuration. In the next chapter, we will build upon this foundation by exploring disk partitioning in depth, covering both the legacy MBR partitioning scheme and the modern GPT standard, along with the tools Linux provides for creating and managing partition layouts.