```
$ docker run alpine sh

# is really, underneath:
clone(CLONE_NEWPID|CLONE_NEWNS|...)
write cgroup.procs; pivot_root()
execve("/bin/sh")
```

NS | cgroup | caps | overlay

one Linux process

THE LINUX FEATURES BEHIND CONTAINERS

How Docker Actually Works

a from-the-kernel-up guide to what a container really is

NAMESPACES · CGROUPS · OVERLAYFS · CAPABILITIES · runc & OCI

Contents

Introduction — It's Just a Process

Part I — Groundwork

1.1 What a Process Really Is — PIDs, fork/exec, the process tree, and /proc

1.2 The Kernel Boundary — User space, kernel space, and the system call

1.3 The Filesystem and the Root — Inodes, the VFS, mounts, and what "/" means

1.4 Root, Users, and Capabilities — The privilege model containers bend

Part II — The Road to Containers

2.1 The Ancestors of the Container — chroot, jails, Zones, and the first isolation

2.2 Google, LXC, and the Birth of Docker — How the Linux primitives became a product

Part III — The Linux Primitives

3.1 Namespaces: What a Process Can See — PID, mount, UTS, and IPC isolation

3.2 The Network Namespace — veth pairs, bridges, and a network from nothing

3.3 User Namespaces and Rootless Containers — How container-root becomes an ordinary user

3.4 Control Groups: What a Process Can Use — Limiting CPU, memory, and I/O with cgroups v2

3.5 Overlay Filesystems and Why Images Are Layers — Copy-on-write, lowerdir/upperdir, and the

3.6 Locking the Door — Capabilities, seccomp, no_new_privs, and LSMs

3.7 Swapping the Root with pivot_root — chroot, pivot_root, and a container's real /

Part IV — Building a Container by Hand

4.1 Building a Container by Hand — A real container with no Docker, step by step

4.2 What an Image Actually Is — Layers, digests, manifests, and the registry

Part V — Docker Itself

5.1 Docker's Real Architecture — CLI, dockerd, containerd, shim, and runc

5.2 Docker Networking, Demystified — docker0, veth, iptables, and port publishing

5.3 Storage: Layers, Volumes, and the Writable Top — overlay2, the container layer, and persiste

5.4 The Life of a docker run — One command, traced from Enter to syscall

Epilogue — Beyond Docker

Appendix A — Glossary of Terms

INTRODUCTION

It's Just a Process

What this book argues, who it's for, and the restraints that make a container.

Introduction

It's Just a Process

Run this on any Linux machine that has Docker, then run the second command in another terminal:

```bash
docker run -d --name web nginx
ps -ef | grep nginx
```

Start nginx in a container, then look for it in the host's process list.

```
OUTPUT (on the host)
root      4123  4099  0 10:02 ?  00:00:00 nginx: master process
systemd+  4170  4123  0 10:02 ?  00:00:00 nginx: worker process
```

Look closely at what just happened. You started a "container," but the process is right there in the host's own process table, with an ordinary PID, visible to plain old `ps`. There is no second operating system. There is no lightweight virtual machine humming away underneath. There is just nginx — a normal Linux process, running directly on the host kernel, exactly like every other process on the machine.

This is the single most important idea in the book, and almost everything else follows from it:

> **The thesis:**
>
> ***A container is not a thing. It is an ordinary Linux process wearing several restraints. Docker is the tool that puts the restraints on for you.***

When people first meet containers they reach for the wrong mental model. They picture a tiny computer inside their computer — a sealed box with its own kernel, its own hardware, its own everything, like a virtual machine but smaller. That picture is not just imprecise; it is actively misleading, and it will make every later question harder than it needs to be. Why can a container see the host's CPU count? Why does a process inside it run as the host's root unless you say otherwise? Why is starting a container nearly instantaneous when booting a VM takes thirty seconds? The tiny-computer model cannot answer these. The real model answers all of them in one breath.

The four restraints

If a container is just a process wearing restraints, the obvious question is: which restraints? There are four families, and each is an independent feature of the Linux kernel that existed, and worked, before Docker was written. This book is, in effect, a guided tour of these four — what each one does, who added it to the kernel and why, and how to operate each one by hand.

What makes a process a container:

namespaces — what it can SEE (its own PIDs, mounts, network, hostname) cgroups — what it can USE (how much CPU, memory, and I/O) a pivoted root + overlay fs — what it thinks "/" IS capabilities + seccomp — what it is ALLOWED to ask the kernel

Strip all four away and you have a normal process that can see every other process, mount, and network interface on the machine; use all the RAM it likes; read the host's real root filesystem; and make any system call as root. Add them back one at a time and the same process becomes progressively more boxed-in, until — with all four in place — it looks, from the inside, like the only thing running on a private machine. Nothing was virtualized. The process was simply lied to, precisely and deliberately, by the kernel it is still sharing with everyone else.

Who this book is for

This is a book for technical readers who already use containers and now want to know what is actually happening. If you can open a shell, run `docker run`, and read a line of C without panicking, you are the intended reader. You do not need to be a kernel hacker. You do need to be willing to type the commands, because the argument of the book is one you can verify with your own hands: by the end of Part IV you will have built a working container using nothing but `unshare`, `mount`, `ip`, and a few writes to a special filesystem — no Docker anywhere in sight — and then watched `docker run` do precisely the same things.

We assume you are comfortable on the Linux command line but hazy on what happens beneath it. So Part I deliberately starts lower than you might expect — with what a process, a system call, a filesystem, and the word "root" really mean — because every later mechanism is defined in those terms, and a shaky foundation there is where most container confusion actually lives.

A note on where you run this

Everything in this book is about the Linux kernel, so the hands-on commands assume Linux. If you are on a Mac or on Windows, this is not a detour — it is the whole point made concrete. Docker Desktop on those platforms does not run containers "natively"; it runs a real Linux virtual machine and runs your containers inside it, because the features this book describes simply do not exist anywhere but the Linux kernel. To follow along, use a Linux box, a cloud VM, or WSL2 on Windows (which is, again, a genuine Linux kernel). When a command needs root we show it with `sudo`; when it needs a recent kernel feature we say so.

One last orientation before we start. The book climbs deliberately. Part I lays the groundwork. Part II tells the story of where all of this came from — the thirty-year road from a 1979 system call to a 2013 startup's last-ditch open-source release. Part III takes the kernel primitives one at a time. Part IV assembles them, by hand, into a container. Part V then opens up Docker itself and shows that it is a careful, well-engineered orchestration of everything you will have already built yourself. Let's begin where every container begins: with a single process.

PART I

Groundwork

A container is built from plain Linux pieces. Before we cage a process, we look squarely at what a process, a syscall, a filesystem, and root really are.

PART I

1.1 What a Process Really Is

PIDs, fork/exec, the process tree, and /proc

Before we can take a process and box it in, we have to be honest about what one actually is. Most of us carry a comfortable fiction: a program runs, it "is" running, and that's that. But "the program" and "the running thing" are two entirely different objects, and keeping them apart is the first foundation this whole book rests on.

A program is a file on disk. It is dead. It is a sequence of bytes in a format the kernel knows how to read — on Linux, almost always ELF — describing machine code, initial data, and instructions for how to lay it all out in memory. You can copy it, email it, checksum it. It does nothing on its own.

A process is a running instance of that program. It is alive. When you launch a program the kernel builds, in RAM, a private world for it: an address space (its own view of memory — code, heap, stack, mapped files), an execution context (register values, the program counter, the open file descriptors), and a pile of kernel bookkeeping that tracks who this process is, who its parent is, what it is allowed to do, and what it is currently waiting for. One program on disk can become a thousand processes at once; each is a separate, living instance.

Keep these apart:

PROGRAM = the file on disk (ELF bytes). Dead. PROCESS = a live instance of it in memory. Alive. One program -> many independent processes.

Inside the kernel, the entire living reality of a process is held in a single C structure called task_struct. It is a large record — pointers to the address space, the open-file table, the signal handlers, the scheduling state, the parent pointer, the credentials (which user and group the process runs as), and much more. When you eventually meet namespaces and cgroups, you will find they are nothing more exotic than extra pointers hanging off this very structure. A container, in the kernel's eyes, is just a task_struct whose pointers happen to point at private, restricted versions of things. Hold that thought; it is the punchline of the whole book, stated three hundred pages early.

The PID: a number that names a life

Every process gets an integer name: its PID, the process identifier. The kernel hands them out and uses them everywhere a process must be referred to — when you send a signal, when you wait for a child, when you read /proc. PIDs are recycled over time but are unique at any given instant. Run `ps` and the first column you usually see is the PID.

```bash
ps -ef | head -4
```

The classic full listing: UID, PID, PPID, and command.

```
UID    PID  PPID  C STIME TTY    TIME CMD
root     1     0  0 09:14 ?  00:00:03 /sbin/init
root     2     0  0 09:14 ?  00:00:00 [kthreadd]
root   842     1  0 09:14 ?  00:00:01 /usr/sbin/sshd
```

Two columns matter most here: PID, the process's own identifier, and PPID, the identifier of its parent. Every process has exactly one parent, which is how the kernel knows who to notify when a process dies — and that single fact builds the entire process tree we will meet in a moment.

PID 1 is special

Look again at that listing. PID 1 is /sbin/init — on a modern distro, usually systemd. PID 1 is the first userspace process the kernel starts at boot, and it is unlike every other process on the machine. It has two duties that no one else has.

First, it adopts orphans. When a process dies while it still has living children, those children do not die with it — they are reparented, and their new parent is PID 1. Their PPID silently changes to 1. Someone must always be responsible for every process, and PID 1 is the parent of last resort.

Second, it reaps. When a child process exits, it does not vanish immediately; it lingers as a corpse (more on this below) until its parent acknowledges the death by calling wait(). PID 1 is expected to diligently wait() on every orphan it adopts, clearing the corpses so the process table never fills with the dead.

Why you should care now:

Later, PID namespaces give a container its OWN PID 1. The first process in a container BECOMES PID 1 inside it. If it doesn't reap, zombies pile up. This is a real container footgun, and now you know why.

fork(): how a process becomes two

Here is the strangest and most beautiful idea in the Unix process model: you do not create a process from nothing. You create one by cloning an existing one. The system call is fork(), and it duplicates the calling process almost perfectly — same code, same open files, same memory contents. After fork() returns, there are two nearly identical processes where there was one.

The memory is not actually copied up front; that would be ruinously slow. Instead the kernel uses copy-on-write: parent and child share the same physical pages, marked read-only, and a page is only truly duplicated the instant one of them writes to it. fork() is therefore cheap even for a process using gigabytes of RAM.

The famous detail is that fork() returns twice — once in each process. In the parent it returns the PID of the new child; in the child it returns 0. That single difference in return value is how a program tells which copy of itself it now is.

```c
#include <stdio.h>
#include <unistd.h>

int main(void) {
    pid_t pid = fork();
    if (pid == 0)
        printf("child:  my pid is %d\n", getpid());
    else
        printf("parent: my child is %d\n", pid);
    return 0;
}
```

fork() returns 0 in the child, the child's PID in the parent.

```
OUTPUT

parent: my child is 5391
child:  my pid is 5391
```

Both lines print because both processes ran the code after fork(). The order is not guaranteed — the scheduler decides who runs first — but the child's reported PID matches the number the parent saw. Two processes, one fork.

exec(): how a process becomes a different program

fork() gives you a second copy of the same program. But you rarely want two copies of your shell — you want the shell to run ls. That is the job of the exec() family of calls (execve and its convenience wrappers). exec() does not create a process. It throws away the calling process's current program image — its code, its heap, its stack — and loads a new program in its place, inside the same task_struct, keeping the same PID.

So the universal pattern for launching a program, the pattern your shell performs every time you type a command, is fork then exec. The shell forks a copy of itself; the child immediately exec()s the program you asked for, replacing the shell's image with, say, ls; the parent shell waits for that child to finish. fork creates the new process; exec gives it a new identity.

> **The fork + exec dance:**
>
> ***1. fork() -> a child copy of the shell appears 2. exec() -> the child REPLACES itself with the new program 3. wait() -> the parent shell waits for it to exit, then reaps***

Beneath fork() lives a more general primitive: clone(). fork() is, in fact, implemented as a particular call to clone(). What clone() adds is a set of flags letting the caller choose exactly what the new process shares with its parent and what it gets a private copy of — the address space, the file table, the signal handlers. Among those flags is a family beginning CLONE_NEW: CLONE_NEWPID, CLONE_NEWNET, CLONE_NEWNS, and their siblings. Those flags are, quite literally, how a container is born. When Docker starts a container, somewhere underneath it calls clone() with those flags set. We are getting ahead of ourselves — but the entire mystery of "how does Docker make a container" reduces, at the bottom, to one call you have now met by name.

The process tree

Because every process has exactly one parent, the set of all processes forms a tree, rooted at PID 1. pstree draws it for you.

BASH

```bash
pstree -p | head -6
```

-p shows each process's PID next to its name.

OUTPUT

```
systemd(1)-+-sshd(842)---sshd(1503)---bash(1507)-+-pstree(1989)
           |-systemd-journal(311)
           |-cron(701)
           `-dbus-daemon(688)
```

You can read your own lineage straight off this tree: your shell was forked by an sshd, which was forked by the listening sshd, which was forked by systemd, PID 1. Everything traces back to 1.

Now the orphan rule from earlier becomes concrete. If a parent exits while its child still runs, the child does not fall off the tree — the kernel grafts it back on under PID 1. The tree stays connected; nobody is ever truly parentless. Inside a PID namespace, as you will see,

this same grafting happens onto the container's own PID 1 instead of the host's.

/proc: the kernel turned inside out

How did `ps` and `pstree` learn all that? They did not perform secret kernel magic. They read files. Linux exposes the live state of every process through /proc, a virtual filesystem that is not backed by any disk — each read synthesizes an answer from the kernel's in-memory structures on the spot. Under /proc there is a numbered directory for every running process, and /proc/self is a convenient alias for "whichever process is doing the reading."

BASH

```bash
ls /proc/self/
```

Each entry is a window into one running process.

OUTPUT

```
cgroup   cmdline  environ  exe   fd    maps
mounts   ns       root     stat  statm status
```

A few of these will reappear as load-bearing characters in later chapters. cmdline holds the exact argument vector; maps shows the address-space layout; fd/ is a directory of symlinks to every open file descriptor; cgroup names which control group constrains this process (Part III); and root is the symlink to what this process believes "/" is (the pivoted-root chapter lives here). But pay special attention to one entry.

BASH

```bash
ls -l /proc/self/ns
```

The handles to this process's namespaces — the heart of containers, sitting in plain sight.

OUTPUT

```
lrwxrwxrwx cgroup -> cgroup:[4026531835]
lrwxrwxrwx ipc    -> ipc:[4026531839]
lrwxrwxrwx mnt    -> mnt:[4026531840]
lrwxrwxrwx net    -> net:[4026531840]
lrwxrwxrwx pid    -> pid:[4026531836]
lrwxrwxrwx user   -> user:[4026531837]
lrwxrwxrwx uts    -> uts:[4026531838]
```

Each link names a namespace this process currently belongs to, identified by an inode number in brackets. Two processes in the same namespace show the same number; a containerized process will show different numbers from yours for some of these. We will not unpack them yet — but file away that a process's namespace membership is visible, right here, as ordinary files. That is the whole trick of Linux: even containment is just a file

you can list.

Read status for a human-readable summary that ties the chapter together — PID, PPID, the running state, and the process's credentials all in one place.

```bash
cat /proc/self/status | head -8
```

A plain-text digest of the process's task_struct.

```
Name:   cat
State:  R (running)
Pid:    2104
PPid:   1507
Uid:    1000 1000 1000 1000
Gid:    1000 1000 1000 1000
```

States, and the curious case of the zombie

A process is not always running. The State field above told you which of a small set of conditions it was in. The ones worth knowing: running (R) — on a CPU or ready to be; sleeping (S/D) — waiting for something, such as a key press or a disk read; stopped (T) — frozen by a signal, as when you press Ctrl-Z; and zombie (Z) — the strange one.

A zombie is a process that has already exited but has not yet been reaped. When a process dies, the kernel cannot fully discard it, because its exit status — did it succeed? with what code? — must be delivered to its parent. So the kernel keeps a husk: the program's memory is gone, but a tiny entry remains in the process table holding the exit status, waiting. The parent collects it by calling wait(); only then does the entry disappear. That collection is reaping, and a zombie is simply a death that nobody has acknowledged yet.

A brief zombie is normal — it exists for the microseconds between exit and the parent's wait(). The trouble is a negligent parent that never calls wait(): its dead children accumulate as permanent zombies, each holding a process-table slot, and the table is finite. This is precisely the failure mode that bites a careless container PID 1. The first process in a container is its PID 1; if that program does not reap the orphans the kernel reparents to it, zombies stack up with no host init to save them. It is the same reaping duty you met pages ago, now with real consequences — and it is why container images often ship a tiny, purpose-built init whose only job is to fork, exec your real program, and patiently wait().

Origins:

The process model here was carved out at Bell Labs in the early 1970s by Ken Thompson and Dennis Ritchie. The split between fork (make a copy) and exec (become a new program) was a deliberate, almost minimalist design choice — and fifty years on, it is still exactly how every Linux program, container or not, is launched.

Try it yourself

Three short experiments. Run them on a Linux box, a cloud VM, or WSL2. None require root.

First, watch a process find its place in the tree. The $$ variable is your shell's own PID.

```bash
echo "my shell is PID $$"
ps -o pid,ppid,state,cmd -p $$
```

Print your shell's PID, then inspect its row directly.

```
my shell is PID 1507
  PID  PPID S CMD
 1507  1503 S -bash
```

Second, manufacture a zombie and watch it appear, then disappear. The child exits at once; the parent sleeps for ten seconds without reaping, so for that window the child is a Z.

```bash
( sleep 10 & wait $! ) &   # parent that reaps late
bash -c 'sleep 0 & sleep 10' &
sleep 1; ps -o pid,ppid,state,cmd | grep -w Z
```

Look for state Z in the listing while the parent dawdles.

```
 6620  6619 Z [sleep] <defunct>
```

The <defunct> tag and the Z state are the kernel telling you: this one is dead but unreaped. Wait for the parent to finish and run `ps` again — the zombie is gone, collected at last.

Third, and most important for what follows: look at your own namespaces. These inode numbers are the identities that PID, network, and mount isolation are built from. Note them

down.

```bash
readlink /proc/self/ns/pid
readlink /proc/self/ns/net
```

The bracketed numbers identify which namespace you are in.

```
pid:[4026531836]
net:[4026531837]
```

Right now these match every other ordinary process on your machine — you all share one PID namespace, one network namespace. When we build a container by hand in Part IV, you will run this exact command inside it and watch the numbers change, and in that moment the whole abstraction will stop being a word and become an inode you can point at. For now it is enough to have seen that a process is no mystery: it is a task_struct with a PID, a parent, a memory image, a state, and a handful of files under /proc that lay it all bare. Everything we restrain from here on is one of those things.

PART I

1.2 The Kernel Boundary

User space, kernel space, and the system call

In the last chapter we said a container is an ordinary Linux process wearing four restraints. Before we can take any of those restraints apart, we have to understand the thing every single one of them is built out of: the system call. Namespaces, cgroups, the pivoted root, capabilities, seccomp — none of them is magic, and none of them is new code that Docker ships. Each is a request a process makes to the kernel, across a boundary that the kernel guards absolutely. This chapter is the map of that boundary. Learn it once here and every later mechanism becomes a special case of something you already understand.

Two worlds, one CPU

Your machine runs in two worlds. There is your code — the shell, nginx, the Python interpreter, the kernel of Docker itself — and there is the kernel: the single program that owns the hardware. These are not a polite social arrangement. The separation is enforced by the CPU, in silicon, on every instruction it executes.

On an x86-64 processor the CPU is always running at one of several privilege levels, called rings. Two of them matter. Ring 0 is the most privileged: code there can execute any instruction the chip offers — talk to the disk controller, reprogram the memory-management unit, mask interrupts, halt the processor. The kernel runs in ring 0. Ring 3 is the least privileged, and that is where everything else runs. Your process lives in ring 3, and we call ring 3 user space, ring 0 kernel space.

A process in ring 3 simply cannot reach the hardware. If it executes a privileged instruction, the CPU does not obey it; the CPU faults and traps into the kernel. The process cannot read another process's memory, cannot touch a network card, cannot open a file on disk by poking at the disk — not because it is impolite to, but because the instructions that would do those things are unavailable to it. Everything a process wants from the outside world, it must ask the kernel for. There is no other path.

The boundary, in one line:

User space proposes. Kernel space disposes. A process can compute all it likes in ring 3, but to TOUCH anything it must cross into ring 0 — and there is exactly one controlled doorway across.

The one doorway: the system call

That doorway is the system call. A system call — syscall for short — is the controlled mechanism by which a ring-3 process hands control to the kernel to perform a privileged operation on its behalf. Open a file, send a packet, create a process, map memory: each is a syscall. The kernel publishes a fixed menu of them, each identified by a small integer, the syscall number. On x86-64 Linux, `read` is number 0, `write` is 1, `openat` is 257, `clone` is 56, `execve` is 59. The numbers are an ABI: they are part of the stable contract between user space and the kernel, and they essentially never change.

Mechanically, making a syscall on x86-64 looks like this. The process puts the syscall number in the `rax` register and the arguments in `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`. Then it executes a single special instruction — `syscall` — which is the one ring-3 instruction whose whole purpose is to cross the boundary. The CPU switches to ring 0, jumps to a fixed entry point the kernel registered at boot, and starts running kernel code. The kernel reads `rax`, looks up the handler in its syscall table, validates every argument (a pointer the caller passed had better point into the caller's own memory, not the kernel's), runs the handler, puts the result back in `rax`, and executes `sysret` to drop back to ring 3 exactly where the process left off. From the program's point of view, one instruction took a while and returned a number.

You almost never write that register dance yourself, because the C library does it for you. When your C program calls `write(fd, buf, n)`, it is not calling the kernel directly — it is calling a function in glibc, a thin wrapper that loads the registers, executes `syscall`, and hands you back the result (turning a negative return into `errno` and -1 along the way). The libc wrapper is convenience, not authority: it adds no privilege. You can skip it entirely and invoke a syscall by number, which we will do in a moment to prove the point.

Watching a program talk to the kernel

Here is the most important tool in this entire book. `strace` runs a program and prints every system call it makes, with arguments and return values. Because a process can only affect the world through syscalls, strace shows you, literally and completely, everything a program does that matters. Run it on something trivial:

```bash
strace -f /bin/echo hi
```

-f also follows child processes (we'll need that later).

```
OUTPUT (trimmed)

execve("/bin/echo", ["/bin/echo", "hi"], 0x..) = 0
brk(NULL)                                 = 0x55..
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY) = 3
read(3, "\177ELF..", 832)              = 832
openat(AT_FDCWD, "/lib/libc.so.6", O_RDONLY) = 3
mmap(NULL, 2..., PROT_READ, MAP_PRIVATE, 3, 0) = 0x..
write(1, "hi\n", 3)                     = 3
exit_group(0)                             = ?
```

Read that top to bottom and you are reading the entire life of the program. `execve` replaced the shell's child with the echo binary. The cluster of `openat`/`read`/`mmap` calls is the dynamic loader pulling in the C library — the machinery behind every dynamically linked program, normally invisible. Then the one line that does the actual work: `write(1, "hi\n", 3)` — write three bytes to file descriptor 1, standard output. The return value `= 3` says the kernel accepted all three. Finally `exit_group` ends the process. Everything echo "did" is in those lines, because there is nowhere else for it to have done anything.

The syscalls that make a container

Now the punchline. There is no `create_container` system call, because there is no such kernel object as a container. What there is, is a handful of ordinary syscalls that you compose. Each of the four restraints from chapter one is one or two entries on the syscall menu, and you will meet every one of them by name in the chapters ahead:

The container syscalls (your reading list):

namespaces *clone() / unshare() / setns()* ***filesystem*** *mount() / pivot_root()* ***cgroups*** *writes to the cgroup filesystem (no syscall)* ***privilege*** *capset() / prctl() / seccomp()*

A few of those deserve a precise word now. `clone` is the syscall that creates a new process; pass it the right flags (`CLONE_NEWPID`, `CLONE_NEWNET`, and friends) and the child is born inside fresh namespaces. `unshare` moves the calling process into new namespaces without forking; `setns` lets a process join namespaces that already exist (this is how `docker exec` gets into a running container). `pivot_root` swaps what the process calls `/`. The privilege calls drop capabilities and install seccomp filters.

Cgroups are the deliberate exception, and it is worth getting right: there is no `cgroup()` system call. You control cgroups by reading and writing ordinary files under a special filesystem mounted at `/sys/fs/cgroup`. To cap a process's memory you literally `write` a number into a file there. So even the one restraint that looks like it might need special machinery is operated through the syscalls you already know — `openat`, `write`, `mkdir`. The kernel exposes the knob as a file; you turn it with file operations. We will see exactly this in the cgroups chapter.

Why there is no virtualization here

This is the moment the container model and the virtual-machine model part ways for good. When a process inside a container makes a syscall, the host kernel runs it. Directly. The `write` from a process in a container and the `write` from any other process on the box enter the very same kernel, hit the very same syscall handler, run the very same code. The container's restraints change what that handler sees and allows — which namespace it resolves a path in, whether a capability is present — but they do not interpose a second kernel. There is nothing in between.

A virtual machine is a fundamentally different animal. A guest in a VM runs its own kernel, and when that guest kernel touches hardware, the privileged instruction traps down to a hypervisor that emulates a virtual disk, a virtual network card, a virtual CPU. Two kernels, an emulation layer between them, and a hardware boundary the guest cannot see past. That layer is why a VM boots like a real machine and takes seconds to start; the absence of it is why a container starts in milliseconds — there is no second kernel to boot, just a process to fork.

The trade, stated plainly:

VM: guest syscall -> guest kernel -> hypervisor -> emulated hardware. Strong isolation, slow. Container: process syscall -> the host kernel. Period. Fast and cheap, because there is ONE kernel — the one everyone on the box is sharing.

Hold on to that last clause, because it is the security story of the rest of the book. Speed and isolation are traded against each other here. Containers are fast precisely because they share the host kernel — and sharing the host kernel means a process in a container is one kernel bug away from the host. The four restraints exist to narrow that shared surface: to limit what a container can see, use, reach, and ask for, given that it is talking to the same kernel as everything else. We will return to this with sharp edges in the security chapter.

Calling the kernel by hand

To make the boundary concrete, here is a C program that writes to standard output twice: once through the libc wrapper, and once by invoking the `write` syscall directly by number, bypassing the wrapper entirely. Both reach the same kernel handler.

```c
#include <unistd.h>
#include <sys/syscall.h>

int main(void) {
    const char *m = "via libc wrapper\n";
    write(1, m, 16);            /* glibc wrapper  */

    const char *d = "via raw syscall\n";
    syscall(SYS_write, 1, d, 16); /* no wrapper    */
    return 0;
}
```

SYS_write is the syscall number (1 on x86-64).

```bash
cc -o twowrites twowrites.c
./twowrites
strace -e trace=write ./twowrites
```

Build, run, then watch only the write calls.

```
via libc wrapper
via raw syscall
--- under strace -e trace=write: ---
write(1, "via libc wrapper\n", 16) = 16
write(1, "via raw syscall\n", 16)  = 16
```

strace cannot tell the two apart, and neither can the kernel: both lines are identical `write` syscalls. The libc wrapper was never anything more than a polite way to set up the registers. The authority was always the kernel's, on the far side of the one doorway.

Origins:

The system call is older than Unix. Time-sharing systems of the 1960s — Multics chief among them — needed to let many users' programs share one machine without trampling each other or the supervisor, so the supervisor ran privileged and user programs trapped into it for service. Unix inherited the idea and made it small and sharp. Linux took the contract further than most. Its syscall ABI is famously stable: a binary built against the syscall numbers of decades ago still runs today. "We do not break userspace" is Linus Torvalds' standing rule — the kernel may change wholesale underneath, but the doorway stays put. That stability is exactly why the container syscalls, added across many years, compose so cleanly into one tool.

Try it yourself

Find the boundary crossings with your own hands. First, watch a real program create and replace processes — the `execve` that loads a binary and the `clone` that spawns a child. `ls` of a directory is enough:

```bash
strace -f -e trace=execve,clone ls / >/dev/null
```

Trace only the process-creation syscalls.

```
OUTPUT (trimmed)
execve("/usr/bin/ls", ["ls", "/"], 0x..) = 0
+++ exited with 0 +++
```

A bare `ls` may show only `execve`, since it does no forking. Run the trace on a shell pipeline to see `clone` fire as the shell spawns each child:

```bash
strace -f -e trace=execve,clone \
       sh -c 'ls | wc -l' >/dev/null
```

The shell clones once per command in the pipe.

```
OUTPUT (trimmed)
execve("/bin/sh", ["sh", "-c", "ls | wc -l"], ..) = 0
clone(child_stack=NULL, flags=CLONE..|SIGCHLD) = 4711
clone(child_stack=NULL, flags=CLONE..|SIGCHLD) = 4712
execve("/usr/bin/ls", ..)              = 0
execve("/usr/bin/wc", ..)              = 0
```

Second, ask a process what syscall it is in right now. The kernel exposes this per-process under /proc:

```bash
cat /proc/self/syscall
```

self is the process doing the read — i.e. cat.

```
OUTPUT
0 0x3 0x7ffe.. 0x20000 0x7ffe.. 0x.. 0x.. 0x7ffe..
```

The first number is the syscall number cat was blocked in while you read this — 0, which is `read`. Of course: to tell you what it was doing, it was reading. The boundary is not an abstraction; it is a register you can dump.

Finally, get the shape of a program by counting its syscalls. `strace -c` runs the command and prints a profile: how many of each call, and how much time each cost.

```bash
strace -c ls / >/dev/null
```

-c summarizes instead of printing every line.

```
OUTPUT (trimmed)
% time  calls  syscall
------  -----  ----------
 24.1      11  mmap
 18.7       9  openat
 12.0       8  read
  9.3       1  execve
  ...
100.0      62  total
```

Sixty-odd syscalls to list a directory — almost all of them the loader and the C library getting ready, a few of them the actual work. Every one is a trip across the one boundary that this whole book is about. Keep `strace -f` in your hand from here on: when we build a container by hand in Part IV, you will run it against `docker run` and watch the `clone`, `unshare`, `mount`, and `pivot_root` calls go by — the four restraints, applied one syscall at a time, in the open.

PART I

1.3 The Filesystem and the Root

Inodes, the VFS, mounts, and what "/" means

Before a container can have its own private "/", you have to be honest about what a filesystem actually is. Most of us carry a folk model in which a file is a named thing that lives at a path, and the path is the file's true identity. That model is wrong in a way that matters enormously the moment you start assembling root filesystems by hand. So we are going to take it apart. By the end of this chapter the phrase "a container's root filesystem" will sound like exactly what it is: a perfectly ordinary directory that the kernel was instructed to treat as the top of the world for one process.

Files are not their names

On a Unix filesystem the thing that *is* a file is called the inode. The inode is a small record on disk holding everything about the file except its name and its contents-as-text: the type (regular file, directory, symlink, device), the owner and group, the permission bits, the timestamps, the size, the link count, and — crucially — the pointers to the data blocks that hold the actual bytes. Every inode has a number, unique within its filesystem. That number, not the path, is the file's real identity.

So where does the name live? In the directory. A directory is not a container of files in any physical sense; it is simply a table — a small file whose contents are a list of (name -> inode number) pairs. When you open `/etc/hostname`, the kernel walks that path one component at a time: it reads the directory `/`, looks up the name `etc` to get an inode number, reads that inode (a directory), looks up `hostname` to get another inode number, and finally reads the inode for the file itself. The path is a route; the inode is the destination.

This is why a single inode can have several names. A hard link is nothing more than a second directory entry pointing at the *same* inode number. There is no "original" and "copy" — both names are equal, peer references to one inode, and the inode carries a link count so the kernel knows how many names point at it. The data is freed only when the count reaches zero. You can watch all of this with `ls -li`, where the `-i` prints the inode number in the first column:

```bash
cd /tmp
echo 'hello' > original
ln original alias        # a hard link, not a copy
ls -li original alias
```

Two names, one inode: the link count rises to 2.

OUTPUT

```
266341 -rw-r--r-- 2 you you 6 Jun 26 10:02 alias
266341 -rw-r--r-- 2 you you 6 Jun 26 10:02 original
```

Both rows show the same inode number, `266341`, and the same link count, `2`. They are not two files that happen to match; they are one file with two names. Delete either name and the bytes survive under the other. The `stat` tool spells the inode's fields out in full:

BASH

```bash
stat /etc/hostname
```

Everything the inode knows, minus the name and the bytes.

OUTPUT

```
  File: /etc/hostname
  Size: 11        Blocks: 8     IO Block: 4096 file
Device: 259,2  Inode: 131079   Links: 1
Access: (0644/-rw-r--r--)  Uid: (0/root) Gid: (0/root)
Modify: 2026-06-26 09:58:14.000000000 +0000
```

Hold on to this separation of name from inode. It is the secret behind the two filesystem tricks that build containers. A bind mount works because the kernel can make a path point at an inode that already lives elsewhere. An overlay filesystem (chapter 11) works by combining whole trees of inodes into one apparent tree. Both are sleight of hand performed on the mapping from names to inodes — which means neither makes sense until you have stopped believing that the path is the file.

The VFS: one tree, many filesystems

Linux understands dozens of on-disk formats — ext4, xfs, btrfs, f2fs — plus things that have no disk at all. It would be unbearable if every program had to know which format it was talking to. It does not, because of the Virtual File System, the VFS: a layer inside the kernel that defines one common interface — inodes, directory entries, files, superblocks — and requires each filesystem type to implement it. Above the VFS, `open`, `read`, `write`, and `stat` look identical no matter what lies below. Below it, each filesystem type plugs in its own code for "read this inode" or "list this directory."

The VFS is what lets the kernel splice radically different things into a single directory tree and have them all behave like files. An ext4 partition, a RAM-backed `tmpfs`, an overlay stack, and the entirely synthetic `proc` are all, from a program's point of view, just parts of the one tree rooted at `/`. They are pluggable backends behind a uniform front. Keep that uniformity in mind: when we later build a container's `/` out of an overlay mount, no application inside needs to know or care. The VFS makes the assembled tree indistinguishable from a plain disk.

Mounting: assembling the single tree

A freshly formatted filesystem is just a self-contained tree of inodes sitting on some device; it has its own root but no place in the world. Mounting is the act of grafting that tree onto an existing directory — the mount point — so that walking into that directory walks into the mounted filesystem instead. The unified tree you see under `/` is not one filesystem. It is many filesystems, each mounted at some point, stitched together into a seamless whole. Your `/` might be ext4, `/boot` a separate partition, `/home` an xfs volume, and `/run` a `tmpfs` — and a path that crosses from one to the next does so without you noticing.

Three tools answer "what is mounted where." The kernel exposes the live truth at `/proc/mounts`; `mount` with no arguments prints the same in a friendlier form; and `findmnt` shows it as a tree and can filter. Ask for just the ext4 mounts:

BASH
```bash
findmnt -t ext4
```

Which ext4 filesystems are mounted, and where.

OUTPUT
```
TARGET SOURCE      FSTYPE OPTIONS
/      /dev/nvme0n1p2 ext4 rw,relatime
```

`tmpfs` deserves a special mention because containers lean on it constantly. A `tmpfs` mount is a filesystem backed by RAM (and swap), not by any disk. You mount it, you write files into it, they live in memory, and they vanish when you unmount it or reboot. It is how Docker gives you fast, ephemeral scratch space and how the host serves `/run` and `/dev/shm`.

The most important kind of mount for our purposes is the bind mount. An ordinary mount attaches a whole device's filesystem at a point. A bind mount instead takes a directory that is *already* part of the tree and makes its inodes appear at a second path as well — the same trick as a hard link, but for an entire subtree, and crossing filesystem boundaries freely. After `mount --bind A B`, the directory `B` shows exactly the contents of `A`; they are two windows onto one set of inodes. This is precisely how Docker volumes graft a host directory into a container, and how, in Part IV, we will assemble a container's root by binding pieces into place. Watch one happen:

BASH
```bash
mkdir -p /tmp/src /tmp/view
echo 'I live in src' > /tmp/src/note.txt
sudo mount --bind /tmp/src /tmp/view
cat /tmp/view/note.txt
findmnt /tmp/view
```

Graft one subtree onto another path; read it from both.

```
OUTPUT
I live in src
TARGET     SOURCE                FSTYPE OPTIONS
/tmp/view  /dev/nvme0n1p2[/tmp/src] ext4 rw,relatime
```

Note the `SOURCE` column: `findmnt` is honest that `/tmp/view` is the same device, subtree `/tmp/src`. No bytes were copied. Write to `/tmp/view/note.txt` and `/tmp/src/note.txt` changes too — one inode, two paths. Unmount with `sudo umount /tmp/view` and the graft is gone, the original untouched.

The root filesystem and what "/" means

Everything above hangs from a single point: `/`, the root. At boot the kernel mounts one filesystem as the root and every other mount descends from it. It is tempting to think of `/` as an absolute, fixed fact about the machine — *the* top, the same for everyone. It is not. The root is per-process context, and it can be changed.

The old, blunt instrument is `chroot`: it tells the kernel that, for this process and its children, a given directory is now `/`. After `chroot /some/dir`, a program asking for `/etc/passwd` is steered to `/some/dir/etc/passwd`; the rest of the system still exists, but this process can no longer name it, because it has no path that climbs above its new root. The sharper instrument, `pivot_root`, goes further: it does not merely point a process at a subtree, it *replaces* the root mount of the current mount namespace with a different filesystem, then lets you unmount the old one entirely. That is what a real container uses, and chapter 13 gives it the full treatment.

Here is the punch line, and it is the whole reason this chapter exists:

The insight:

A container's "/" is not special. It is an ordinary directory on the host that the kernel was told to treat as root for one process and its children.

There is no second disk, no image magically "mounted as a drive." There is a directory — usually an overlay-assembled one — and a process whose notion of `/` was swapped to point at it. The container's filesystem isolation is the same name-to-inode machinery you just learned, aimed deliberately.

Mount propagation, briefly

One subtlety will matter when mounts meet namespaces (chapter 7) and when Docker shares volumes. A mount point carries a propagation type that decides whether mount and unmount events *under* it travel to its copies in other contexts. A shared mount propagates events both ways; a private mount propagates nothing; a slave mount receives

events from its master but sends none back. You do not need the full theory now — only the fact that it exists, because a container's mounts are typically made private or slave so that mounting something inside the container does not leak out onto the host, and a Docker bind mount can be configured to propagate or not. We will return to it the moment it becomes load-bearing.

Pseudo-filesystems: kernel interfaces dressed as files

Some of the most important entries in the tree are backed by no storage at all. They are kernel interfaces wearing the costume of files, so that you can inspect and control the kernel with the same `cat`, `echo`, and `ls` you use on real data. Name the regulars: `proc` (mounted at `/proc`) exposes one directory per process plus a heap of kernel knobs; `sysfs` (`/sys`) exposes devices and the kernel's object model; `tmpfs` is the RAM-backed store we met above; `devtmpfs` (`/dev`) presents device nodes; and `cgroup2` (`/sys/fs/cgroup`) is the control-group interface that chapter 8 will use to cap a process's CPU and memory. None of these hold files in the disk sense. Reading `/proc/self/status` does not read a stored document — it makes the kernel compute an answer and hand it back through the VFS as if it were one. This is the deepest reach of "everything is a file": even the kernel's own controls are presented as paths.

Origins:

The hierarchical filesystem, the mount model, and the idea that everything is a file all come from Unix at Bell Labs in the early 1970s. One uniform tree, assembled from many mounts, where devices and data share one naming scheme — that design is fifty years old and still load-bearing. The /proc idea was sharpened by Plan 9, Bell Labs' successor system, which pushed "everything is a file" to its limit; Linux adopted and expanded /proc and /sys from that lineage.

Try it yourself

Three short experiments make the chapter concrete. Run them on a Linux box (or WSL2 / a VM). First, prove a bind mount shares inodes; second, mount RAM as a filesystem; third, read an inode's metadata directly. Each is reversible and harmless.

BASH

```bash
# 1. Bind mount: one subtree, two paths, shared inodes.
mkdir -p /tmp/lab/src /tmp/lab/view
echo 'written via src' > /tmp/lab/src/f.txt
sudo mount --bind /tmp/lab/src /tmp/lab/view
echo 'appended via view' >> /tmp/lab/view/f.txt
cat /tmp/lab/src/f.txt    # the change shows here too
ls -i /tmp/lab/src/f.txt /tmp/lab/view/f.txt

# 2. tmpfs: a filesystem that lives in RAM.
mkdir -p /tmp/lab/ram
sudo mount -t tmpfs -o size=16m tmpfs /tmp/lab/ram
df -h /tmp/lab/ram

# 3. Inspect an inode directly.
stat /tmp/lab/src/f.txt

# Clean up.
sudo umount /tmp/lab/view /tmp/lab/ram
```

Bind a subtree, mount RAM, read an inode, then undo it all.

OUTPUT (abridged)

```
written via src
appended via view
918452 /tmp/lab/src/f.txt
918452 /tmp/lab/view/f.txt
Filesystem  Size  Used Avail Use% Mounted on
tmpfs        16M     0   16M   0% /tmp/lab/ram
  File: /tmp/lab/src/f.txt
  Inode: 918452   Links: 1
```

Read the output back against the chapter. The two paths in step 1 report the same inode number, so an append through one is visible through the other — proof that a bind mount shares inodes rather than copying. Step 2's `df` shows a 16 MB filesystem of type `tmpfs` that exists only in memory. Step 3 shows the inode's own record. You have now, by hand, performed the two filesystem operations a container relies on. In Part IV you will use the very same `mount --bind` and a swapped root to give a process a private `/` — and it will be nothing more mysterious than what you just did in `/tmp`.

PART I

1.4 Root, Users, and Capabilities

The privilege model containers bend

Every process on a Linux system runs as somebody. Not metaphorically — concretely, as a small set of integers the kernel keeps in the process's credentials. The most important of those integers is the user ID, the uid. When a process tries to open a file, send a signal, bind a port, or reboot the machine, the kernel does not ask who you are in any human sense; it compares numbers. This chapter is about those numbers, about the one number that historically meant "skip all the checks," and about the long, careful project of breaking that one number into forty smaller ones. That project — Linux capabilities — is the restraint Docker leans on most heavily, and the place where "root inside a container" turns out to mean something far more dangerous than most people assume.

Numbers that stand in for people

A running process carries a uid and a gid (group ID), and in fact carries three of each: the real uid (who launched it), the effective uid (whose authority it acts with right now), and the saved set-uid (a parked value it can switch back to). Almost every permission decision uses the effective uid. You can see your own credentials with `id`.

BASH
```bash
id
```

Print the calling process's uid, gid, and group memberships.

OUTPUT
```
uid=1000(alice) gid=1000(alice) groups=1000(alice),27(sudo)
```

The names in parentheses are a courtesy from `id`, which looks them up in /etc/passwd and /etc/group. The kernel itself never sees "alice"; it sees 1000. This matters enormously later: two systems can disagree about which name a uid maps to, or map it to nothing at all, and the kernel will not care. A uid is just a number, and a number means whatever the surrounding files say it means — or nothing.

One uid is special. uid 0 is root, and root is defined by a single brutal rule: when the effective uid is 0, most permission checks are simply skipped. Not granted, not negotiated — skipped. Root can read any file, kill any process, mount any filesystem, load kernel modules, and reconfigure the network, because for uid 0 the kernel takes the fast path past the gate entirely. Understanding what root is — an integer the kernel treats as a master key — is the foundation for understanding why a container's root is so worth

worrying about.

How the kernel checks an ordinary file

For everyone who is not root, the checks do run, and they are simpler than their reputation. Each file carries an owning uid, an owning gid, and nine permission bits: read, write, execute, in three groups — owner, group, and other. `ls -l` renders them as the familiar string.

```
BASH
ls -l /etc/shadow /etc/hostname
```

Owner, group, and the rwx bits for two files.

```
OUTPUT
-rw-r----- 1 root shadow 1234 Jun  1 09:00 /etc/shadow
-rw-r--r-- 1 root root      9 Jun  1 09:00 /etc/hostname
```

When a process opens a file the kernel checks exactly one group of bits. If the process's effective uid equals the file's owner, it uses the owner bits and stops. Otherwise, if the effective gid (or a supplementary group) matches the file's group, it uses the group bits and stops. Otherwise it uses the other bits. So /etc/shadow, with no bits at all for "other," is unreadable to alice — unless alice is root, in which case the whole dance is skipped and the file opens. That asymmetry, root versus everyone, is the entire classic model in one sentence.

The setuid trick, and why it is dangerous

Real systems need ordinary users to do a few privileged things — change their own password (which writes /etc/shadow), or ping a host (which needs a raw socket). The classic answer is the set-user-ID bit. If an executable has its setuid bit set, then when anyone runs it, the new process's effective uid becomes the file's owner, not the caller's. Look for the `s` where an `x` would normally sit.

```
BASH
ls -l /usr/bin/passwd /usr/bin/sudo
```

The s in rws is the setuid bit.

```
OUTPUT
-rwsr-xr-x 1 root root 68208 ... /usr/bin/passwd
-rwsr-xr-x 1 root root 232416 ... /usr/bin/sudo
```

Because both files are owned by root and carry the setuid bit, alice running `passwd` momentarily becomes root for the duration of that program. This is elegant and it is also

the original sin of Unix security. A setuid-root binary is a tiny door through which an unprivileged user enters the kingdom of full root. If the program has a bug — a buffer overflow, a careless `system()` call, a mishandled environment variable — that bug now runs as root. Decades of privilege-escalation exploits are, at bottom, abuses of setuid binaries that trusted their inputs too much. Hold this thought: the danger is not that the program needs *a* privilege; it is that setuid gives it *every* privilege at once.

Root was all or nothing

That last sentence names the structural flaw. In the classic model there was exactly one privileged identity, and it owned the whole toolbox. A program that needed to bind to TCP port 80 — a check that exists only to stop ordinary users from impersonating system services — had to run as root. But running as root did not grant it merely the bind-low-ports power. It granted it everything: the power to read every file, to load kernel modules, to change any process's uid, to wipe the disk. A web server wanted one key and was handed the master key, because the master key was the only key that existed.

This is precisely the problem containers must manage. A container often runs software that wants to feel like root — install packages, write to /etc, own its filesystem — without actually being entrusted with the host's master key. The kernel feature that makes the compromise possible is capabilities.

Capabilities: cutting root into forty pieces

Starting with kernel 2.2 in 1999, Linux split the monolithic power of root into a set of independent privilege units called capabilities — today around forty of them. Each names one specific thing the kernel will otherwise forbid. A few you will meet constantly:

A sampler of capabilities:

CAP_NET_BIND_SERVICE — bind to ports below 1024
CAP_NET_RAW — open raw sockets (this is how ping works)
CAP_NET_ADMIN — configure interfaces, routes, firewalls
CAP_CHOWN — change a file's owner CAP_DAC_OVERRIDE
— bypass the rwx permission checks CAP_SETUID — change
the process's uid arbitrarily CAP_SYS_ADMIN — mount,
pivot_root, and a hundred more

CAP_SYS_ADMIN deserves a warning. So many operations were filed under it over the years that holding it is very nearly equivalent to holding root itself — it is sometimes called "the new root." Granting a container CAP_SYS_ADMIN to fix one small problem usually undoes most of the security you were hoping containers would give you. When you see it in a `docker run --cap-add`, be suspicious.

A process does not hold a single flag per capability; it holds several *sets*. The permitted set is what it may use; the effective set is what is active right now (the kernel checks this one); the inheritable and ambient sets govern what survives an `execve` into a new program; and the bounding set is a ceiling — a capability dropped from the bounding set can never be regained, even by exec. You rarely manipulate these by hand, but knowing the permitted/effective distinction explains a lot of otherwise-baffling EPERM errors. Inspect any process's capabilities with `getpcaps`, and decode them in human-readable form with `capsh`.

```bash
getpcaps $$
```

Show the capability sets of the current shell ($$ is its PID).

```
OUTPUT (as an ordinary user)
Capabilities for PID 4711: =
```

An ordinary shell holds no capabilities at all — the bare `=` means every set is empty. It does not need any: as a non-root process its powers come from its uid and the file bits, not from the capability machinery. Run the same inspection inside a root shell and the lists fill up. `capsh --print` gives the fuller picture, including the all-important bounding set.

```bash
sudo capsh --print | head -3
```

Decode the current capability state in readable form.

```
OUTPUT (as root)
Current: =ep
Bounding set: cap_chown,cap_dac_override,...,cap_sys_admin,...
Ambient set:
```

Capabilities can also live on files, replacing many uses of the setuid bit with something far more surgical. Modern `ping` does not need to be setuid-root; it just needs the one power to open a raw socket, attached to the binary as a file capability. Read it with `getcap`.

```bash
getcap /usr/bin/ping
```

What capability is baked into the ping binary?

```
OUTPUT
/usr/bin/ping cap_net_raw=ep
```

That `cap_net_raw=ep` says: when this file is executed, CAP_NET_RAW lands in the new process's effective and permitted sets — and nothing else does. A bug in `ping` can now misuse exactly one privilege instead of the whole of root. This is the setuid trick made safe, and you can grant it yourself with `setcap` (which is itself a privileged operation, naturally).

> **Origins:**
>
> ***The idea of splitting root predates Linux. POSIX.1e — a draft standard for "least privilege" with capabilities and ACLs — was developed in the 1990s and ultimately withdrawn, never ratified. Linux adopted the surviving ideas anyway, landing capabilities in kernel 2.2 (1999); file capabilities and the ambient set arrived much later (2.6.24 in 2008, 4.3 in 2015). The setuid bit they soften is older still — it dates to early Unix and a 1979 patent by Dennis Ritchie.***

What "root inside a container" really is

Now the payoff. Start an ordinary container and become root inside it; then look at that same process from the host.

```bash
docker run -d --name c busybox sleep 1000
docker exec c id
ps -o pid,uid,cmd -C sleep    # run on the host
```

The container's root, seen from inside and from outside.

```
OUTPUT
uid=0(root) gid=0(root)            # inside the container
  PID  UID CMD
 5821    0 sleep 1000              # on the host
```

Read those two zeros carefully. The uid inside the container is 0. The uid on the host is *also* 0. By default these are not two different roots that merely share a name — they are the very same uid 0, the host's actual root. The container does not run as some sandboxed pseudo-root; it runs as real root, and the only things standing between it and the whole machine are the other restraints in this book: a reduced capability set, a pivoted root, a seccomp filter, and namespaces. Docker mitigates the danger by *dropping* most capabilities the moment the container starts. A default container keeps only a small, conservative subset — CAP_CHOWN, CAP_NET_BIND_SERVICE, CAP_SETUID, and a handful more — and drops the rest, CAP_SYS_ADMIN very much included. Chapter 12 lists the exact set and the seccomp filter that backs it up.

But "most capabilities dropped" is not the same as "a different user." If an attacker finds a

kernel bug that hands back a dropped capability, or escapes the namespace boundary, they land on the host as uid 0 — full root on your machine, not root in a sandbox. This is the single sharpest edge in the container model, and it is the reason user namespaces exist. A user namespace (Chapter 9) lets container-root *be* uid 0 inside while *being* some harmless unprivileged uid — say 100000 — to the host kernel. Then even a clean escape lands you as a nobody. Until you turn that on, treat root in a container as root, period.

A one-way promise: no_new_privs

There is one more small but important flag. A process can set the `no_new_privs` bit via prctl(2), and once set it is irreversible for that process and inherited by all its children. The promise it makes is precise: this process will never gain privileges through an execve again — setuid bits are ignored, file capabilities do not elevate, nothing it runs can climb higher than it already is. Container runtimes set this bit so that a compromised process cannot pivot through some stray setuid binary inside the image. It is also the prerequisite that lets an unprivileged process install a seccomp filter without that filter becoming a privilege-escalation vector — which is exactly where Chapter 12 picks up the thread.

Try it yourself

Three quick experiments, each one a sentence of this chapter made concrete. First, confirm your shell holds no capabilities, then find a binary that carries one as a file capability:

```bash
getpcaps $$
getcap /usr/bin/ping
```

Your empty capability set, and ping's single file capability.

```
Capabilities for PID 4711: =
/usr/bin/ping cap_net_raw=ep
```

Now watch a capability *matter*. `capsh --drop` removes a capability from the bounding set before running a command, so the command can never acquire it — not even from a file capability. Run `ping` normally (it works), then run it with CAP_NET_RAW dropped and watch it fail with EPERM, the kernel's way of saying "permission denied" at the syscall.

```bash
sudo capsh --drop=cap_net_raw -- \
  -c "ping -c1 127.0.0.1"
```

Strip away the one power ping depends on, then run it.

OUTPUT

```
ping: socket: Operation not permitted
```

That single line is the whole argument of the chapter in miniature. `ping` is the same binary, on the same kernel, run by the same user — but with one capability removed from its bounding set, the raw socket it needs is forbidden, and it fails. Multiply that by the dozens of capabilities Docker drops at container start, and you can feel the shape of the restraint: not a wall around the process, but a careful subtraction from what root is allowed to ask the kernel for. A container's root is the host's root with most of its powers quietly taken away — and your job, every time you reach for `--cap-add` or `--privileged`, is to know exactly which powers you are handing back.

PART II

The Road to Containers

Where containers came from, who built each piece and why — from chroot in 1979 to Google's cgroups, LXC, and the day a struggling startup open-sourced Docker.

PART II

2.1 The Ancestors of the Container

chroot, jails, Zones, and the first isolation

Docker did not invent the idea of giving a process its own private world. By the time the first `docker run` was typed in 2013, the central question — how do you make one machine behave like many, each isolated from the others — had already been asked and answered, in different ways, on at least four operating systems and across more than three decades. Linux, awkwardly, was the latecomer: it had no single built-in answer at all, only a scattering of primitives that had to be wired together by hand. To understand why containers on Linux look the way they do — assembled from namespaces and cgroups rather than handed down as one tidy feature — you have to know what came before, and what each ancestor could and could not do. This chapter is that lineage, told as a sequence of engineers facing concrete problems and inventing isolation one layer at a time.

The deep ancestor: many machines inside one

The dream is older than Unix. In the late 1960s, IBM's Cambridge Scientific Center built CP/CMS, a system whose control program (CP) gave each user the illusion of a complete, private IBM mainframe — a virtual machine — running their own copy of the lightweight CMS operating system. The lineage matured into VM/370, released in 1972, which could host dozens of independent operating-system instances on a single physical machine. This is full hardware virtualization, not containers, and the distinction matters: VM/370 simulated whole computers down to the hardware, while everything else in this chapter shares one kernel and partitions what sits above it. But the goal — many isolated environments on one expensive machine — is exactly the goal containers would inherit fifty years later.

1972 · VM/370

WHO: IBM (lineage from CP/CMS at the Cambridge Scientific Center, late 1960s). WHERE: IBM, United States. WHY: Let one mainframe present itself as many independent virtual machines, so expensive hardware could be shared and consolidated.

1979: chroot, and the first cage with a hole in it

The first piece of the container story that survives, almost unchanged, in Docker itself arrived with the Seventh Edition of Unix — Version 7 — released by the Unix developers at Bell Laboratories (Bell Telephone Laboratories, then part of AT&T) in 1979. That release introduced the `chroot` system call. Its job is narrow and precise: it changes a process's idea of where the root of the filesystem, `/`, lives. After a process calls `chroot("/srv/jail")`, every absolute path it uses is resolved as if `/srv/jail` were the top of the world. The rest of the disk simply ceases to exist for that process. It is best to credit `chroot` to the V7 Unix team of that era — the environment of Ken Thompson and Dennis Ritchie — rather than to any single named inventor.

The feature found its defining use a few years later at the University of California, Berkeley. Bill Joy and the Berkeley group used `chroot` while building the Berkeley Software Distribution — 3BSD and the early 4.x BSD releases in the early 1980s — to assemble and test software inside a clean, controlled directory tree, free of whatever clutter the real system had accumulated. That is still one of `chroot`'s most honest uses today: a known-good build environment. You can stage the same trick in two commands.

```bash
mkdir -p /tmp/jail/bin
cp /bin/busybox /tmp/jail/bin/
sudo chroot /tmp/jail /bin/busybox sh
```

Build a tiny root, then run a shell that believes it is /.

```
OUTPUT (inside the chroot)
/ # ls /
bin
/ # ls /etc
ls: /etc: No such file or directory
```

Inside that shell, `/` is really `/tmp/jail`, and `/etc` does not exist because we never put it there. It feels like isolation. It is not — or rather, it is isolation of exactly one thing, the filesystem namespace, and nothing else. This is the fatal limitation that shaped everything after it.

Why chroot was never enough

A `chroot`ed process changes its view of the filesystem and only its filesystem. It keeps the same process IDs, visible in the same global list; it shares the same network stack, the same user accounts, the same hostname, the same kernel-wide everything as every other process on the machine. Run `ps` inside the cage (if you copied it in) and you see the host's processes; the host sees yours. Two `chroot`s on one machine are not two machines — they are two filesystem views layered over one shared system.

Worse, the cage barely holds even on its one dimension, because root can walk straight out. The classic escape is almost insultingly simple: a process running as root opens a directory file descriptor, calls `chroot` into a deeper subdirectory, and then `chdir`s back up through the still-open descriptor — relative path traversal that predates the new root — until it reaches the real `/`. The kernel never promised `chroot` would contain a privileged process; it was designed as a convenience for organizing filesystem views, not a security boundary. The lesson the next generation of engineers drew was blunt: to isolate a process you must isolate everything it can see and touch — processes, network, users — not just its files, and you must do it in a way root cannot trivially undo.

1979 · chroot

WHO:* *The Unix developers at Bell Labs (the V7 team, Thompson/Ritchie era); popularized for build isolation by Bill Joy's group at UC Berkeley in 3BSD/4.x BSD, early 1980s.* *WHERE:* *Bell Telephone Laboratories (AT&T), USA; then Berkeley, California.* *WHY:* *Give a process a private filesystem root, for testing and clean software builds.* *FLAW:* *Isolates only the filesystem; PIDs, network, and users stay shared, and root can escape it.

2000: FreeBSD jails — the first real container

The first system to take the lesson seriously and ship a complete answer was FreeBSD. The work was done by Poul-Henning Kamp, a prolific FreeBSD developer, on behalf of a small hosting provider that needed to safely rent slices of a single FreeBSD server to different customers. The result, introduced in FreeBSD 4.0 in the year 2000, was the `jail`. Kamp and Robert Watson described it in the paper "Jails: Confining the omnipotent root," presented at the SANE 2000 conference — and that subtitle is the whole point. A jail set out to do precisely what `chroot` could not: confine root.

A FreeBSD jail combined a `chroot`ed filesystem with much more. Each jail had its own hostname, its own restricted set of network addresses, its own process view (a jailed process could see and signal only other processes in the same jail), and — crucially — its own root user whose powers were curtailed so that jail-root could not reach out and harm the host or other jails. To a customer it looked like a private FreeBSD machine; on the metal it was one kernel partitioned several ways. That combination — filesystem plus processes plus network plus a contained superuser — is what we now mean by a container, and FreeBSD jails are fairly described as the first real one.

2000 · FreeBSD jails

WHO: Poul-Henning Kamp, for a hosting provider that needed to partition one server among customers; paper by Kamp & Robert Watson, SANE 2000. WHERE: The FreeBSD project. WHY: Safely rent isolated slices of one FreeBSD server, each with its own contained root, network, and process space. GAIN: Confined the 'omnipotent root' chroot left free.

2004: Solaris Containers and Zones

Four years later, Sun Microsystems shipped its own take with Solaris 10 in 2004. Sun's marketing called the whole feature "Solaris Containers"; the isolation primitive at its heart was the Zone. A non-global zone was a virtualized instance of the Solaris user environment — its own process tree, users, network identity, and filesystem — all running on the single shared kernel of the global zone. The motivation was server consolidation: data centers full of lightly used machines could be collapsed onto fewer, larger Solaris servers, each carved into many zones, without the per-instance overhead of full hardware virtualization.

What set the Solaris work apart was how tightly it married isolation to resource management. Through the Solaris Resource Manager, administrators could cap and guarantee CPU, memory, and other resources per zone, so one tenant could not starve the others — a discipline Linux would later reach by a very different road called cgroups. Later Solaris releases deepened the model with ZFS integration and "branded zones" that could even emulate the environment of older Solaris releases or, in some forms, Linux. Isolation plus accounting, in one product: this is conceptually the closest pre-Linux relative of a modern container platform.

2004 · Solaris Zones

WHO: Sun Microsystems. WHERE: Solaris 10. WHY: Server consolidation — pack many isolated OS environments onto one machine, with enforced per-zone resource limits (Resource Management), and later ZFS and branded-zone integration.

Linux had nothing — so people bolted it on

Through all of this, mainstream Linux had no equivalent. There was `chroot`, inherited from Unix, and that was essentially the whole built-in toolbox. The demand, especially from the booming virtual private server hosting market, was enormous — and where the mainline kernel offered nothing, others patched it in themselves.

The first major effort was Linux-VServer, started in 2001 by Jacques Gélinas. It was a set of patches to the Linux kernel that partitioned a running system into many isolated "security contexts," each looking like an independent server, and it became a workhorse of cheap VPS hosting. A second lineage came from the company SWsoft (later Parallels), whose commercial Virtuozzo product had an open-source core released in 2005 as OpenVZ. OpenVZ delivered container-like isolation — separate process trees, network, users, and resource limits via its own "beancounters" — again as out-of-tree kernel patches, and again hugely popular with hosting providers.

The decisive fact about both is that neither was ever merged into the mainline Linux kernel. They lived as large patch sets that distributions and hosts had to apply to a modified kernel. The kernel maintainers were unwilling to absorb a single monolithic container subsystem wholesale. Instead — and this is the bridge to everything that follows — mainline Linux grew its own isolation piecemeal, as a series of small, general primitives that any program could compose: namespaces, one kind at a time, and control groups for resource accounting. The out-of-tree systems proved the demand was real; the mainline answer arrived as building blocks rather than a finished house.

2001-2005 · Linux, out of tree

WHO: Linux-VServer — Jacques Gelinas, 2001. OpenVZ — SWsoft/Virtuozzo, open-sourced 2005. WHERE: Out-of-tree patches to the Linux kernel. WHY: Give Linux the container-like isolation it lacked, chiefly for VPS hosting. FATE: Never merged into mainline; mainline instead grew namespaces + cgroups separately (ch. 6).

What every ancestor had in common

Step back and the pattern is clear. Each system solved the same problem — many isolated environments on one machine — for its own platform and on its own terms. IBM virtualized the hardware itself. `chroot` isolated a single dimension, the filesystem, and proved by its failure that one dimension is not enough. FreeBSD jails isolated all the dimensions at once and contained root, earning the title of first true container. Solaris Zones added rigorous resource control. And the out-of-tree Linux projects showed that the demand on Linux specifically was overwhelming, even as the kernel community declined to answer it with a single feature.

That refusal is the most important inheritance of all. Because mainline Linux would not adopt a turnkey container — no `jail`, no `zone` — it instead exposed a kit of orthogonal primitives that, combined just so, produce isolation indistinguishable from a jail or a zone. Docker is not a kernel feature; it is a particularly good way of assembling that kit. The next chapter opens the kit and names its parts: the namespaces that give a process its own private view of PIDs, mounts, network, users, and more — the mechanism that turns

Linux's scattered primitives into something you can finally call a container.

Try it yourself

You can feel both the power and the famous weakness of the oldest ancestor in a couple of minutes. Build a minimal root from a static BusyBox, enter it with `chroot`, and confirm that the filesystem — and only the filesystem — has changed underneath you.

```bash
mkdir -p /tmp/jail/bin
cp $(which busybox) /tmp/jail/bin/
sudo chroot /tmp/jail /bin/busybox sh
# now, inside:
ls /          # only 'bin' exists
ps            # but PIDs are the HOST's, not private
```

chroot changes /, but not the process list.

```
OUTPUT (inside the chroot)
/ # ls /
bin
/ # ps
  PID USER    COMMAND    # host processes, fully visible
```

The `ls` shows a tiny private filesystem; the `ps` shows the host's whole process list, because `chroot` never touched the PID namespace — there was no PID namespace to touch in 1979. That single gap, multiplied across processes, networks, and users, is the entire reason jails, zones, and finally namespaces had to be invented. With the ancestry in hand, we can now meet the Linux primitive that finally closes the gap: namespaces, in Chapter 6.

PART II

2.2 Google, LXC, and the Birth of Docker

How the Linux primitives became a product

By now you have a feeling for what a container is made of: isolated views of the system, fenced-off slices of its resources. But none of that arrived in a single release, from a single vendor, in a single year. It accreted. The machinery beneath Docker was built over more than a decade by kernel hackers in mailing-list threads, by a search company desperate to pack more jobs onto each server, and by a small French-founded startup in San Francisco that almost went out of business first. This chapter answers the question you actually asked: where was this developed, by whom, for what reason, and when.

There is a useful prehistory worth naming before the real story starts, because people love to point at it as 'the first container.' It wasn't, but it mattered.

1979 · chroot

Bill Joy adds the chroot system call during development of Version 7 Unix; it ships in 7th Edition / early BSD. It changes a process's idea of '/', the root of the filesystem. It isolates ONE thing — the filesystem view — and nothing else: not processes, not the network, not users. A useful trick, not a container.

Keep chroot in mind as the shape of everything that followed: each real isolation primitive does for one class of resource what chroot did for the filesystem. The Linux kernel just had to grow a chroot-like fence around every other global namespace in the system. That took eleven years.

Namespaces: eleven years of incremental kernel work

A namespace is the kernel's answer to a simple question: what if two processes could each believe they had the whole machine to themselves? Not by lying about it from userspace, but by the kernel genuinely handing each one a private view of some global resource. The catch is that Linux has many global resources — the filesystem mount table, the hostname, inter-process communication objects, the process-ID tree, the network stack, the user-ID space — and each one had to be carved up separately, carefully, by different people at different times. There was never a grand 'namespaces' patch. There was a slow procession.

2002 · the first namespace

Mount namespaces (CLONE_NEWNS) land in kernel 2.4.19. Work led by Al Viro and other VFS contributors. Gives a process its own private set of filesystem mounts — chroot, generalized. WHY: so a process tree can mount and unmount without the rest of the system seeing it.

2006 · UTS and IPC namespaces

In kernel 2.6.19, the UTS namespace (hostname / domain name) and the IPC namespace (System V IPC, POSIX message queues) are merged. Now a process can have its own hostname and its own private IPC objects.

2007-2009 · PID and network

The PID namespace and the network namespace arrive incrementally. PID-namespace pieces and the early network namespace land around 2.6.24 (2007-08); the network namespace is rounded out by ~2.6.29 (2009): its own interfaces, routes, firewall.

2013 · the user namespace (last & hardest)

The user namespace is only completed in kernel 3.8 (early 2013), among the key contributors Eric W. Biederman. It lets UID 0 inside a container map to an unprivileged UID outside — the basis of 'rootless' containers. The trickiest of all, because getting it wrong is a security hole.

Read those dates again. The first namespace shipped in 2002; the last was finished in 2013 — the same year Docker was unveiled. The namespace machinery and the Docker product crossed the finish line in the same year, but the machinery had been under construction by a rotating cast of kernel contributors for over a decade. No company owned it. It was mainline Linux, built in the open, one global resource at a time. That is the first half of the answer to 'by whom': many people, mostly volunteers and kernel-employed engineers, across eleven years.

cgroups: Google needed to pack the fleet

Namespaces answer 'what can a process see?' They say nothing about 'how much can it use?' A process can be perfectly isolated and still eat every CPU cycle and every byte of RAM on the box. The other half of containment — accounting and limiting resource consumption — came from a very specific place and a very specific need.

2006 · cgroups born at Google

Paul Menage and Rohit Seth, engineers at Google, start work on what they first call 'process containers.' To avoid overloading the word 'container,' it is renamed 'control groups' — cgroups. WHY: Google packs enormous numbers of jobs onto each machine and must account for and cap each job's CPU and memory precisely.

Jan 2008 · cgroups in mainline

Control groups merge into the mainline Linux kernel in 2.6.24 (January 2008). The fleet- efficiency tool that Google built for itself becomes a standard kernel feature for everyone.

This is the pivotal where/who/why of the whole story, and it is easy to miss. The single most important resource-management primitive in modern containers came out of Google's internal cluster management — the lineage that led to Borg — because Google's economics depended on stuffing as many workloads as possible onto each server without letting any one of them starve the others. cgroups was not invented to make Docker. It was invented to make Google's data centers cheaper, and then donated to the commons.

LXC: the first time someone glued it together

By 2008 the kernel had most of the pieces — namespaces for isolation, cgroups for limits — but no friendly way to use them together. Wiring up clone() flags and cgroup filesystems by hand is not something application developers do for fun. Someone had to package the primitives into the thing we now call a container.

2008 · LXC (Linux Containers)

LXC, the first userspace toolset to combine namespaces + cgroups into usable containers, is created — among the key authors Daniel Lezcano and Serge Hallyn, working at IBM. LXC becomes the foundation that Docker would later build on.

LXC worked. It was real, it was open source, and it gave you containers years before Docker existed. And yet most developers never touched it. Remember that — it is the key to why Docker, and not LXC, became a household word. The raw capability was there in 2008. What was missing was an experience.

Meanwhile, inside Google: billions of containers a week

While the open-source world slowly assembled LXC, Google had long since gone all in internally. Its cluster manager, Borg, developed from the mid-2000s onward, ran essentially every Google workload inside containers — using cgroups and namespaces — at a scale no one outside the company understood at the time. Google was launching billions of containers a week before most engineers had heard the word. In 2013 Google open-sourced a slice of that internal stack as lmctfy ('Let Me Contain That For You'). And the operational knowledge from Borg would, in 2014, surface publicly as Kubernetes — but that is a later chapter's story. The point here: containers were boring, proven infrastructure inside Google for years before the wider industry caught the fever.

Docker: the main event

Now the part everyone half-remembers. Docker was not born as a grand plan to revolutionize software delivery. It was born as plumbing inside a struggling company.

The company was dotCloud, a Platform-as-a-Service startup founded by Solomon Hykes. dotCloud had French roots — Hykes is French — but the company was based in San Francisco and went through Y Combinator. Its business was running customers' web applications for them. To do that, dotCloud packed customer apps into LXC-based containers, and it built an internal engine to manage all those containers. That internal engine was Docker.

The PaaS business was not winning. So the team made the decision that changed everything: they took the internal container engine and open-sourced it.

March 2013 · Docker unveiled

Solomon Hykes gives a now-famous five-minute lightning talk, 'The future of Linux Containers,' at PyCon US in Santa Clara, California. Docker is released as open source on March 20, 2013. The company later renames itself from dotCloud to Docker, Inc., and eventually sheds the PaaS business entirely.

Here is the question that matters most: why did Docker catch fire when LXC, with the same kernel primitives underneath, never did? The answer is not a kernel feature. It is developer experience. Docker wrapped the primitives in things people actually wanted to use: a simple command-line interface; the Dockerfile, a plain recipe for building an environment; and — most decisively — an image format with layers, plus a public registry,

Docker Hub, where images could be pushed and pulled and shared the way code is shared on a git host. 'Build, ship, run.' You could build an image on your laptop and someone across the world could run the exact same thing. LXC gave you a container. Docker gave you a workflow.

libcontainer: cutting the cord

Docker originally drove LXC under the hood. That made Docker dependent on an external toolset whose behavior it did not fully control. So Docker built its own.

2014 · libcontainer

Docker replaces its LXC dependency with libcontainer, its own library written in Go that talks directly to kernel namespaces and cgroups. Docker becomes self-contained — no longer riding on top of LXC.

The standardization — and the breakup

Success brought rivals, and rivals brought a fight over standards. CoreOS, unhappy with Docker's direction, released a competing runtime called rkt in 2014 and pushed the appc image specification. A fragmented container ecosystem helped no one. The industry's response was to standardize the format so that no single company owned it.

June 2015 · the OCI is formed

The Open Container Initiative (OCI) is founded under the Linux Foundation — Docker, CoreOS, Google, Red Hat and many others — to define vendor-neutral standards: a runtime-spec and an image-spec. Docker donates libcontainer, which becomes runc, the OCI reference runtime.

2017 · containerd to the CNCF

Docker splits out its high-level runtime as containerd and donates it to the Cloud Native Computing Foundation (CNCF). The engine that started as one startup's internal tool is now broken into shared, governed components.

To ground all of this in something you can run, the version strings on a modern machine quietly recount the breakup: the Docker client, and beneath it the OCI runtime that used to be libcontainer.

```bash
$ docker --version
$ runc --version
```

The product, and the donated runtime under it.

```
EXAMPLE

Docker version 24.x.x, build ...
runc version 1.x.x
spec: 1.x.x  (the OCI runtime-spec)
```

Notice that last line: runc reports which version of the OCI runtime-spec it implements. The 2015 standardization is literally printed by the binary. The lineage is not folklore — it is in the version output.

Where this leaves us

So: where, by whom, why, when. The isolation primitives — namespaces — were built in mainline Linux by many kernel contributors from 2002 to 2013, each carving up one global resource. The resource-control primitive — cgroups — came from Google in 2006, born of fleet-packing economics, merged to mainline in 2008. LXC first glued the pieces into usable containers in 2008, out of IBM. Google had been running everything in containers internally via Borg the whole time. And Docker, the product that made all of it famous, came from dotCloud's internal plumbing, unveiled by Solomon Hykes in March 2013, succeeding not on new kernel magic but on developer experience — the CLI, the Dockerfile, the layered image, the registry.

Everything after that is decomposition: libcontainer in 2014, runc and the OCI in 2015, containerd to the CNCF in 2017. Keep these dates and names close, because the rest of this book is going to take them apart. Part III dissects the very primitives this chapter just dated — namespaces and cgroups, mechanism by mechanism. Part V dissects the very architecture this chapter just named — containerd, runc, and the OCI specs. You now know the story. Next we open the machine.

PART III

The Linux Primitives

The mechanisms themselves, one per chapter: namespaces, cgroups, overlay filesystems, capabilities and seccomp, and pivot_root. Each chapter is hands-on.

PART III

3.1 Namespaces: What a Process Can See

PID, mount, UTS, and IPC isolation

We said a container is an ordinary Linux process wearing four restraints. This chapter takes apart the first and most visible of them: namespaces, the mechanism that controls what a process can *see*. Everything you have read so far — the process, the system call boundary, the filesystem tree — has been preparation for this. A namespace is how the kernel takes a resource that is normally global to the whole machine and hands one process its own private copy, while convincing that process the copy is the whole world. Get this idea in your bones and the rest of containerization is detail.

What a namespace actually is

Pick any resource the kernel manages on your behalf: the table of process IDs, the list of mounted filesystems, the hostname, the network interfaces. Each of these is, by default, global — one table for the entire machine, shared by every process. A namespace wraps one such resource so that the processes inside the namespace see their own private instance of it, isolated from the instance everyone outside sees. Two processes in different PID namespaces can both believe they are PID 1, and neither is lying, because each is reading a different table.

There is no "container" object inside the kernel — there is no struct, no registry, nothing called a container anywhere in the source. There is only a process that happens to be a member of a set of namespaces. As of a modern kernel there are eight namespace types, and they isolate independently: a process can be in a new PID namespace but share the host's network, or have its own mounts but the host's hostname. The eight are mount (`mnt`), PID, network (`net`), IPC, UTS, user, cgroup, and time. This chapter covers four of them — `mnt`, PID, UTS, and IPC. The network namespace gets its own chapter (8) because it is large; the user namespace (chapter 9) is where rootless containers live; cgroup and time namespaces we will only name in passing.

Three syscalls drive all of them

Back at the kernel boundary you learned that nothing here is magic — every mechanism is a system call. Namespaces are no exception, and remarkably the entire feature rests on just three calls. `clone()` creates a new process and, if you pass the right `CLONE_NEW*` flags, places that new process into fresh namespaces at the moment of birth. `unshare()` takes the *calling* process and detaches it into new namespaces in place — no child required. `setns()` does the opposite of creating: it makes the calling process *join* a namespace that already exists. These map cleanly onto the verbs you need: create-with-child, create-in-place, and join.

```c
/* the flags that ask clone/unshare for new namespaces */
CLONE_NEWNS     /* mount namespace (the original; hence NS) */
CLONE_NEWUTS    /* UTS: hostname + domainname              */
CLONE_NEWIPC    /* System V IPC + POSIX message queues     */
CLONE_NEWPID    /* PID namespace                           */
CLONE_NEWNET    /* network namespace (chapter 8)           */
CLONE_NEWUSER   /* user namespace (chapter 9)              */
CLONE_NEWCGROUP/* cgroup namespace                        */
CLONE_NEWTIME   /* time namespace                          */
```

One bit per namespace type; OR them together to ask for several at once.

How does `setns()` name a namespace it wants to join? Through a file descriptor — and that is the second half of the story. Every process exposes its namespaces as a set of magic symlinks under `/proc/<pid>/ns/`. Each link names one namespace the process belongs to, and the link's target encodes the namespace type and a unique inode number. Those links are the *handles*: open one and you hold a reference to that namespace; pass the resulting fd to `setns()` and you join it. This also answers a question you may not have thought to ask — how long does a namespace live? A namespace exists as long as something references it: a member process, or an open fd to its `/proc/.../ns/` link. When the last reference goes, the kernel tears the namespace down. This is exactly how Docker keeps a container's network namespace alive even when no process is in it — by holding an open fd, or a bind mount, to the link.

```bash
ls -l /proc/$$/ns
```

The current shell's namespaces, one symlink each.

```
cgroup -> 'cgroup:[4026531835]'
ipc    -> 'ipc:[4026531839]'
mnt    -> 'mnt:[4026531840]'
net    -> 'net:[4026531992]'
pid    -> 'pid:[4026531836]'
user   -> 'user:[4026531837]'
uts    -> 'uts:[4026531838]'
```

Those numbers in brackets are inode numbers in a special `nsfs` filesystem, and they are how you *prove* two processes share — or do not share — a namespace: same inode, same namespace. Hold that fact; we use it at the end of the chapter to demonstrate isolation concretely. The numbers above all begin `40265318..` because they are the *initial* namespaces the system booted with; anything you create yourself will get a different inode.

You will rarely call these syscalls by hand. Two command-line tools wrap them: `unshare` runs a program in new namespaces (it calls `unshare()` or `clone()` for you), and `nsenter`

runs a program inside an *existing* namespace (it calls `setns()`). The spine of this chapter is doing everything by hand with `unshare`, because once you have built isolation yourself, what Docker does stops looking like sorcery.

UTS: the simplest namespace

Start with the gentlest example. The UTS namespace — the name is a fossil, "Unix Time-sharing System" — isolates exactly two things: the hostname and the NIS domain name. That is all it does, which makes it the perfect place to *see* a namespace work without distraction. Open a shell in a new UTS namespace, change its hostname, and watch the host's hostname stay put.

```bash
hostname                  # what the host is called
sudo unshare --uts bash  # new UTS ns, root shell
  hostname container1     # rename, inside the ns only
  hostname                # confirms the new name
  exit
hostname                  # back outside: unchanged
```

Rename the host inside the namespace; the real host never notices.

```
OUTPUT

thinkpad
# (now inside the unshared shell)
container1
# (after exit, back on the host)
thinkpad
```

Inside the unshared shell, `hostname container1` changed *that namespace's* copy of the hostname. The host's copy — the one the outer shell reads — was never touched, because the two shells are looking at two different copies of a single kernel field. This is the whole idea of a namespace, reduced to one string. Every other namespace is this same trick applied to a more interesting resource.

PID: the big one

The PID namespace is where isolation gets serious. Inside a new PID namespace a process sees its own private numbering of process IDs, starting fresh at 1, and — critically — it cannot see any process outside the namespace at all. The host's hundreds of processes simply do not exist as far as a contained process can tell. This is the difference between a `chroot`, which only hides the filesystem, and a real container: a container in a PID namespace cannot even enumerate, let alone signal, the processes of its host.

There is a subtlety that trips up everyone the first time. Creating a PID namespace does not, by itself, change what `ps` shows you, because `ps` reads `/proc`, and `/proc` is a *mount* — it still reflects the old PID namespace until you remount it. This is where

chapter 3 pays off: `proc` is a pseudo-filesystem the kernel renders per PID namespace, so you must give the new namespace its own freshly mounted `/proc`. There is a second subtlety: the process that calls `unshare --pid` does not itself enter the new namespace; its *next child* does. So you almost always want `--fork` (fork a child into the namespace) together with `--mount-proc` (give that child a private mount namespace with a fresh `/proc`).

BASH

```bash
sudo unshare --pid --fork --mount-proc bash
  ps -ef
  echo "I am PID $$"
  exit
```

New PID namespace, forked into it, with its own /proc so ps tells the truth.

OUTPUT

```
UID  PID  PPID  C STIME TTY   TIME     CMD
root   1     0  0 10:14 pts/0 00:00:00 bash
root   8     1  0 10:14 pts/0 00:00:00 ps -ef
I am PID 1
```

Read that output carefully, because it is startling. The shell is PID 1. Not PID 1 pretending — PID 1 as far as the kernel will report to anything inside this namespace. `ps -ef` sees exactly two processes: the bash that is PID 1 and the `ps` it spawned. The host's systemd, your editor, the hundred daemons running outside — all invisible. Yet this is the same `bash` binary, on the same kernel, and from the host you could run `ps` and see this very shell sitting at some large PID like `48211`. Two numberings of one process; each namespace reads its own.

Being PID 1 is not an honor — it is a responsibility, and getting it wrong is the source of a famous class of container bugs. On Unix, PID 1 has two special duties. First, it reaps orphans: when any process's parent dies, the orphan is re-parented to PID 1, and PID 1 must `wait()` on it to clear the zombie entry, or zombies accumulate. Second, signals to PID 1 are treated specially — the kernel will not apply default signal actions to it, so a PID 1 that installs no handler for `SIGTERM` will simply ignore `docker stop` and the kernel will eventually `SIGKILL` it after the timeout. And there is a blunt rule on top: if PID 1 in a namespace dies, the kernel kills every other process in that namespace. The namespace has no init to fall back on.

Why your container needs tini:

An app run directly as PID 1 usually reaps no zombies and handles no SIGTERM. That is why real images launch a tiny init — tini or dumb-init — as PID 1, which forwards signals and reaps orphans, and runs your app as its child.

This is the practical face of an idea from the opening chapter: a container is a process, and a process in a PID namespace inherits PID 1's burdens whether or not it was written to bear them. The minimal init processes exist for exactly this reason, and now you can see precisely what they are protecting you from.

Mount: a private list of mounts

The mount namespace — `CLONE_NEWNS`, the very first namespace Linux ever shipped — gives each member its own list of mounts. A filesystem mounted inside the namespace appears only inside it; the host's mount table is untouched. This is the namespace that, combined with `pivot_root` in chapter 13, lets a container have an entirely private filesystem tree — its own `/`, its own `/proc`, its own everything — built on the host's storage but invisible from it. Watch a mount stay private:

```bash
sudo unshare --mount bash
  mkdir -p /tmp/ns-demo
  mount -t tmpfs none /tmp/ns-demo
  mount | grep ns-demo     # the namespace sees it
  exit
mount | grep ns-demo       # the host does not
```

Mount a tmpfs inside the namespace; it never appears on the host.

```
# inside the unshared shell:
none on /tmp/ns-demo type tmpfs (rw,relatime,...)
# back on the host, grep prints nothing.
```

The `tmpfs` you mounted is real, writable, and entirely yours — and the moment you exit the shell, the namespace dies, its mount list with it, and the `tmpfs` is gone. There is a wrinkle that connects straight back to chapter 3's note on mount propagation: depending on your distribution's defaults, the new mount namespace may start with its mounts marked *shared*, in which case a mount inside could propagate back to the host. Container runtimes defend against this by making the tree private first — the equivalent of `mount --make-rprivate /` inside the namespace — so that nothing leaks outward. `unshare --mount --propagation private` does this for you. Keep that in your pocket; it is the kind of detail that turns a working demo into a working container.

IPC: isolating shared memory and queues

The IPC namespace isolates the two old families of inter-process communication that identify their objects by global keys rather than by file paths: System V IPC — shared memory segments, semaphore sets, and message queues — and POSIX message queues. Because these objects live in a global namespace by default, two unrelated programs on a host can collide on the same key. Put each container in its own IPC namespace and a shared-memory segment created inside it is simply not visible to anything outside, key collisions and all. The `ipcs` tool lists these objects:

```bash
ipcs -m                       # host shm segments
sudo unshare --ipc bash
  ipcs -m                     # empty: fresh namespace
  exit
```

A new IPC namespace starts with no shared-memory segments at all.

```
# host: several segments owned by various services
key        shmid  owner  perms  bytes
0x00000000 5      gdm    600    524288
# inside the namespace: the table is empty
key        shmid  owner  perms  bytes
```

It is a quieter isolation than PID or mount, but it matters for any workload that leans on shared memory — databases, language runtimes, scientific code — because it stops one container from reading or clobbering another's segments, and stops key collisions between containers that each assume they own the global IPC space.

Inspecting and listing namespaces

Two everyday techniques let you confirm what is going on. The first you already have: compare the inode numbers behind `/proc/<pid>/ns/<type>` for two processes — equal means shared, different means isolated. `readlink` reads one link directly. The second is `lsns`, which lists every namespace on the system, its type, its inode, and the processes inside it — the fastest way to get the lay of the land.

```bash
readlink /proc/$$/ns/pid       # this shell's PID ns
lsns -t pid                    # all PID namespaces
```

readlink names one namespace; lsns enumerates them all by type.

OUTPUT

```
pid:[4026531836]
        NS TYPE NPROCS   PID USER COMMAND
4026531836 pid     214     1 root /sbin/init
4026532192 pid       2 48211 root bash
```

There they are, side by side: the host's PID namespace with 214 processes rooted at `/sbin/init`, and the namespace you unshared with just two processes, whose `bash` the host sees as PID `48211` but which sees *itself* as PID 1. The inode numbers differ, so the namespaces are genuinely distinct. You are no longer taking the kernel's isolation on faith.

Origins:

Namespaces arrived one at a time, mostly the work of Eric Biederman and collaborators. The mount namespace came first, in 2002 (kernel 2.4.19). UTS and IPC landed in 2006; PID and network across 2007-2008; user namespaces matured by 2013 (3.8); time namespaces as late as 2020 (5.6). They were deliberately built to isolate each resource class independently, which is why you can mix and match them — and why containers are an assembly of features, not one monolithic switch.

Try it yourself

Three experiments, building from gentle to convincing. Run them on a Linux box (or WSL2 / a VM); each needs `sudo` and each is fully reversible by exiting the shell.

```bash
# 1. UTS: change the hostname in isolation.
hostname                         # e.g. thinkpad
sudo unshare --uts bash
  hostname container1
  hostname                       # -> container1
  exit
hostname                         # -> thinkpad still

# 2. PID: become PID 1, see only your own processes.
sudo unshare --pid --fork --mount-proc bash
  ps -ef                         # only bash + ps
  exit

# 3. Prove isolation by inode number.
readlink /proc/$$/ns/pid         # host PID ns inode
sudo unshare --pid --fork bash -c \
  'readlink /proc/$$/ns/pid'     # a different inode
lsns -t pid                      # list both side by side
```

Rename a host, become PID 1, then prove the two PID namespaces differ by inode.

```
OUTPUT (abridged)
thinkpad
container1
thinkpad
UID PID PPID C STIME TTY   TIME     CMD
root  1    0 0 10:22 pts/0 00:00:00 bash
root  9    1 0 10:22 pts/0 00:00:00 ps -ef
pid:[4026531836]
pid:[4026532201]
```

Step 1 shows the host's name surviving a rename that happened in another UTS namespace. Step 2 shows a shell that is PID 1 in a world of two processes. Step 3 is the proof: the two `readlink` calls report *different* inode numbers, so the two PID namespaces are genuinely separate kernel objects, and `lsns` lists them both. You have now, by hand, built four of the restraints that define a container's view of the world. What a process can *see* is settled. The next chapter turns to the fifth namespace and a harder question — what a process can *reach* on the network — and after that, in chapter 9, to who a process is allowed to *be*.

PART III

3.2 The Network Namespace

veth pairs, bridges, and a network from nothing

Of all the namespaces, the network namespace is the one that feels most like a separate computer. Where the mount namespace gives a process its own filesystem and the PID namespace its own process tree, the network namespace gives it an entire private networking stack: its own set of network interfaces, its own routing table, its own ARP cache, its own iptables and nftables rules, its own /proc/net, and — crucially — its own port number space, so two namespaces can each bind port 80 without ever colliding. Everything the kernel knows about networking is duplicated per namespace, and a process in one namespace cannot see, touch, or be touched by the interfaces of another.

Here is the part that surprises people. When the kernel creates a fresh network namespace, it does not copy the host's network into it. It creates an empty one. A brand-new network namespace contains exactly one interface — loopback, `lo` — and that interface starts administratively *down*. No Ethernet device, no IP address, no route to anywhere, not even a working `127.0.0.1`. A container, at the instant its network namespace is born, has no connectivity at all. It is an island with no bridges and no boats.

So every bit of a container's networking — its IP address, its route to the gateway, its ability to reach the host, the bridge it shares with its siblings, the NAT that lets it reach the internet — is *built*, step by careful step, after the namespace exists. Docker does this automatically and invisibly, and the result feels like magic. It is not. It is the handful of `ip` commands in this chapter, run by a daemon instead of by you. The fastest way to demystify Docker networking is to perform it yourself, by hand, once. That is what we are going to do: build a working network out of nothing, then scale it to a bridge, then reach the internet — and discover we have reinvented `docker0`.

Making and entering a namespace

There are two ways to get a network namespace. The low-level way is `unshare --net`, which spawns a process in a fresh, *anonymous* namespace — it has no name, and it lives only as long as a process holds it open. That is essentially what a container runtime does. For building things by hand, though, the friendlier tool is `ip netns`, iproute2's helper for *named* namespaces. It creates a namespace, gives it a name, and bind-mounts it under /var/run/netns/ so it persists and you can re-enter it at will.

```bash
sudo ip netns add c1
ip netns list
```

Create a named network namespace and confirm it exists.

```
OUTPUT
c1
```

To run a command *inside* that namespace, you prefix it with `ip netns exec c1`. This is the manual cousin of `docker exec`: it drops the given command into c1's network world. Let us look at what c1 starts with — its complete networking, fresh from the kernel.

```bash
sudo ip netns exec c1 ip addr
```

List every interface the new namespace has.

```
OUTPUT
1: lo: <LOOPBACK> mtu 65536 ... state DOWN
    link/loopback 00:00:00:00:00:00 brd ...
```

That is the whole of it: one loopback interface, state DOWN, with no address. Try to ping yourself from inside and it fails — even `127.0.0.1` is unreachable until you bring `lo` up. This emptiness is the blank canvas. Now we paint connectivity onto it.

The veth pair: a virtual Ethernet cable

The kernel cannot run an Ethernet wire between two namespaces, but it can do the next best thing: a *veth pair*. A veth pair is two interfaces created together and permanently linked, like the two ends of a patch cable. Whatever packet you push into one end comes out the other end, unchanged. The trick that makes them useful is that the two ends can live in *different* namespaces. Leave one end in the host and move the other end into c1, and you have a cable joining the island to the mainland.

```bash
sudo ip link add veth0 type veth \
  peer name veth1
```

Create the pair: veth0 and veth1, joined back to back.

Both ends are born in the host namespace. We leave veth0 here and push veth1 across into c1. Moving an interface into a namespace is a single command — and the moment you run it, veth1 vanishes from the host's interface list and appears inside c1.

```bash
sudo ip link set veth1 netns c1
```

Move the far end of the cable into the c1 namespace.

Now the cable is in place: veth0 in the host, veth1 in c1, the two electrically joined. But a

cable with no addresses and powered-off ports carries nothing. We assign each end an IP on the same /24 subnet — the host end gets 10.0.0.1, the container end 10.0.0.2 — and bring both ends up. Note the `ip netns exec` prefix on the c1 side: those commands run *inside* the namespace.

```bash
# host side
sudo ip addr add 10.0.0.1/24 dev veth0
sudo ip link set veth0 up
# container side
sudo ip netns exec c1 \
  ip addr add 10.0.0.2/24 dev veth1
sudo ip netns exec c1 ip link set veth1 up
sudo ip netns exec c1 ip link set lo up
```

Address both ends, power them on, raise loopback too.

Bringing `lo` up inside the namespace is easy to forget and worth the habit — plenty of software assumes 127.0.0.1 works. With both ends addressed and up, the kernel has automatically installed a connected route for 10.0.0.0/24 on each side. The island now has a boat. Let us sail it: ping the host from inside c1.

```bash
sudo ip netns exec c1 ping -c1 10.0.0.1
```

From inside the namespace, reach the host end of the cable.

```
OUTPUT
PING 10.0.0.1 ... 56 data bytes
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.05 ms

--- 10.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss
```

Zero percent packet loss. You just built a working network out of an empty namespace using nothing but a virtual cable and four addresses. There was no magic anywhere in that sequence — and there is none in Docker either. This is the irreducible core. Everything that follows is scaling it up.

Many namespaces, one bridge

A point-to-point cable connects exactly two things. To wire up a second container you could add another veth pair, but then c1 and c2 still could not talk to each other, only to the host — you would be building a star of isolated links. What you actually want is a *switch*: one shared segment that everyone plugs into. Linux has a software switch built into the kernel, the *bridge*. Create one the same way you create any virtual interface.

```bash
sudo ip link add br0 type bridge
sudo ip link set br0 up
```

A virtual Ethernet switch, br0, powered on.

The model changes slightly. Instead of giving the host end of each veth its own IP, you *enslave* it to the bridge — the host end becomes a port on the switch and carries no address of its own. The bridge itself gets one address, 10.0.0.1, which becomes the gateway every container shares. Re-doing c1 this way, and adding an identical c2, looks like this:

```bash
sudo ip addr add 10.0.0.1/24 dev br0
# attach c1's host end to the switch
sudo ip link set veth0 master br0
sudo ip link set veth0 up
# create and wire c2 the same way
sudo ip netns add c2
sudo ip link add veth-c2 type veth \
  peer name veth1
sudo ip link set veth1 netns c2
sudo ip link set veth-c2 master br0
sudo ip link set veth-c2 up
```

Each container's host-side veth becomes a bridge port.

Give c2's inside end 10.0.0.3/24 and bring it up, and now c1, c2, and the bridge all sit on one Layer-2 segment. c1 can ping c2 directly at 10.0.0.3, the frames flowing in through veth, across the bridge, and out the other veth — exactly as if all three were plugged into a physical switch. This is not an analogy for Docker's default networking. It *is* Docker's default networking. The bridge Docker creates is called `docker0`, it carries the gateway address (172.17.0.1 by default), and every container gets a veth pair with its host end enslaved to `docker0`. Run `ip link` on any Docker host and you will see `docker0` and a fistful of veth interfaces with cryptic names — the very thing you just built, automated.

Reaching the internet: routing, forwarding, NAT

Containers on the bridge can reach the host and each other, but try to ping 8.8.8.8 from inside c1 and it fails. Three things are missing, and each maps to one fix. First, c1 has a route to 10.0.0.0/24 but no idea where to send anything else; it needs a *default route* pointing at the gateway, the bridge's 10.0.0.1.

```bash
sudo ip netns exec c1 \
  ip route add default via 10.0.0.1
```

Send all non-local traffic to the bridge gateway.

Second, packets now arrive at the host destined for the wider internet, but a Linux box does not forward packets between interfaces unless you tell it to. By default it is a host, not a router. Flip the switch.

```bash
sudo sysctl -w net.ipv4.ip_forward=1
```

Turn the host into a router that forwards between interfaces.

Third — and this is the subtle one — the outside world has never heard of 10.0.0.0/24. It is a private range living only inside your host. A reply addressed to 10.0.0.2 would have nowhere to go on the real internet. The fix is *masquerading*: as packets leave the host's real interface, rewrite their source address to the host's own public address, and reverse the translation on the way back. That is source NAT, and one iptables rule installs it.

```bash
sudo iptables -t nat -A POSTROUTING \
  -s 10.0.0.0/24 -j MASQUERADE
```

NAT container traffic behind the host's own IP address.

With a default route, forwarding enabled, and masquerade in place, c1 can finally reach 8.8.8.8. The container's private address never appears on the wire; the internet sees only your host. And once more: this is precisely what Docker does. When the daemon starts it sets `net.ipv4.ip_forward=1`, and for the default bridge it adds a MASQUERADE rule for the container subnet. In Chapter 17 we will run `iptables -t nat -L -n` on a live Docker host and read the actual rules the daemon installed — and they will look exactly like the one you just wrote.

The reverse direction — letting the outside reach a container — is *port publishing*, the `-p 8080:80` you have typed a hundred times. It is the mirror image of masquerade: a *destination* NAT rule (DNAT) that rewrites traffic arriving on the host's port 8080 to the container's 10.0.0.2:80. We will leave the full rule for Chapter 17, but you can already see its shape — another line in the nat table, rewriting an address as a packet crosses the boundary.

Origins:

Network namespaces were the slowest of the namespaces to mature: the groundwork began around kernel 2.6.19 (2006) and the implementation was considered complete near 2.6.29 (2009), which is why robust container networking only really arrived at the end of that decade. The pieces they orchestrate are older. The Linux bridge dates to the late 1990s, born to make a PC act as an Ethernet switch; the veth driver arrived in 2.6.24 (2008), purpose-built to stitch namespaces together. Containers did not invent this machinery — they wired up tools the kernel already had, for exactly this kind of job.

Try it yourself

Here is the whole two-namespace experiment condensed into a script you can paste into a root shell on any Linux box (or WSL2 / a VM). It builds a namespace, runs a veth cable into it, addresses both ends, and pings across — the irreducible core of container networking in eight commands.

```bash
sudo ip netns add c1
sudo ip link add veth0 type veth \
  peer name veth1
sudo ip link set veth1 netns c1
sudo ip addr add 10.0.0.1/24 dev veth0
sudo ip link set veth0 up
sudo ip netns exec c1 \
  ip addr add 10.0.0.2/24 dev veth1
sudo ip netns exec c1 ip link set veth1 up
sudo ip netns exec c1 ip link set lo up
sudo ip netns exec c1 ping -c1 10.0.0.1
```

Build a network from nothing, then ping across the cable.

```
OUTPUT

PING 10.0.0.1 ... 56 data bytes
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.04 ms

--- 10.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss
```

When you are done, tear it down. Deleting the namespace also destroys any interface still living inside it, and removing one end of a veth pair removes its partner automatically — so cleanup is short.

```bash
sudo ip netns del c1
sudo ip link del veth0 2>/dev/null
```

Delete the namespace; veth1 dies with it, veth0 by hand.

That ping is the whole thesis of the chapter in one line of output. A container does not arrive with a network; it arrives with an empty namespace and a loopback that is switched off. Connectivity is assembled afterward from primitives that predate Docker by a decade — a veth cable, a bridge that switches frames, a routing table, a forwarding flag, a NAT rule. You have now built every one of those by hand. When we open up Docker's own networking in Chapter 17 — its `docker0` bridge, its veth zoo, its iptables rules — none of it will be a black box. It will just be this chapter, run for you, at scale.

PART III

3.3 User Namespaces and Rootless Containers

How container-root becomes an ordinary user

Chapter 4 left you with a warning sharp enough to reread: by default, uid 0 inside a container is not a polite imitation of root — it is the host's actual root, the same integer 0 the kernel treats as a master key. Everything that keeps a container's root from owning your machine is subtraction layered on top of that root: the capabilities Docker drops, the seccomp filter that forbids dangerous syscalls, the namespaces that hide the rest of the system. Those restraints are good. But they are restraints on a process that, underneath, still runs as the most powerful identity the kernel knows. If any one of them fails — a kernel bug that hands back a dropped capability, a namespace boundary that leaks — the attacker does not land in a sandbox. They land on your host, as uid 0, with nothing left to stop them.

This chapter is about the feature that changes the underlying identity itself, not just the powers attached to it. A user namespace lets a process be root inside its own little world while the host kernel treats it as a harmless, unprivileged stranger. Get this right and a clean container escape lands the attacker as a nobody. Get it really right and you no longer need sudo to run containers at all.

Mapping a range of users

A user namespace does one deceptively simple thing: it establishes a translation between the uids and gids seen inside the namespace and the uids and gids the rest of the system sees. The mapping is range-based. You declare that inside-uid 0, for the next N values, corresponds to some outside-uid — say 100000 — for the next N values. From then on, a process that reads its own credentials inside the namespace sees uid 0 and believes itself root. The kernel, whenever that process touches anything outside the namespace — a host file, a shared resource, another process — silently translates that 0 into 100000 and makes its decision about an ordinary unprivileged user.

So the process is genuinely root *within* its namespace. It can create users, chown files it owns, and wield a full set of capabilities — but every one of those powers is scoped to the namespace and the objects owned by the mapped range. Touch something owned by real uid 0 on the host and the kernel sees uid 100000 reaching above its station, and says no. Two roots, then, that are not the same root: a sovereign of a small country who is a tourist everywhere else.

The killer property: the one namespace anyone can make

Chapter 7 introduced the namespaces — mount, pid, net, uts, ipc — and noted that creating most of them requires CAP_SYS_ADMIN, which an ordinary user does not have. The user namespace is the exception, and the exception is the whole story. An unprivileged user, with no capabilities and no sudo, is allowed to create a user namespace. That alone would be a curiosity. What makes it revolutionary is what happens the instant you do: inside your fresh user namespace you hold a full capability set.

Read that again. You started as a powerless user. You called one syscall and now, inside the namespace you just made, you are root with every capability lit. Those capabilities are valid only within the namespace — useless against the host — but they are exactly the capabilities the kernel demands before it will let you create the *other* namespaces. So you can now unshare a mount namespace, a pid namespace, a network namespace, all without a shred of host privilege, because the kernel checks those operations against your in-namespace capabilities. This single chain — unprivileged user makes a user namespace, becomes root inside it, uses that root to build a full set of namespaces — is the entire foundation of rootless containers. Rootless Podman and rootless Docker are, at bottom, this trick wrapped in tooling.

uid_map, gid_map, and who is allowed to write them

The mapping lives in two files per process: /proc/<pid>/uid_map and /proc/<pid>/gid_map. Each line has three whitespace-separated numbers, and the order matters:

The uid_map line format:

ID-inside-ns ID-outside-ns count 0 100000 65536 reads: inside-uid 0 maps to outside-uid 100000, and the next 65536 ids map one-for-one (1 -> 100001, and so on).

A namespace's maps are write-once and empty until set. Writing them is where the privilege question reappears, because an unrestricted mapping would be a gift to attackers — map outside-uid 0 to yourself and you have minted host root. So the kernel is strict. A process may map its own single uid freely (this is what `--map-root-user` does). To map a whole *range* of outside ids, you must either be privileged, or go through the two setuid helper binaries newuidmap and newgidmap. Those helpers are trusted: they consult two files that record exactly which ranges an administrator has delegated to each user.

Those files are /etc/subuid and /etc/subgid. A line in /etc/subuid delegates a contiguous block of subordinate uids to one user — ids that user is permitted to use *inside* a user namespace, but never as their own host identity.

```bash
cat /etc/subuid
```

Which subordinate uid ranges are delegated to whom.

```
alice:100000:65536
bob:165536:65536
```

That first line says: alice may map up to 65536 subordinate uids, starting at host uid 100000. So inside alice's user namespace, uid 0 can become host uid 100000, uid 1 becomes 100001, and so on up to 65535. These host uids belong to no real account — they are deliberately parked in unused numeric space — which is the point: files a rootless container writes end up owned by uid 100000, an identity with no login, no home, and no power. /etc/subgid does the same for groups. When you install rootless Docker or Podman, populating these two files (often via `usermod --add-subuids`) is the one setup step that genuinely needs an administrator.

Becoming root without being root

Enough theory; make it happen. `unshare` creates new namespaces and runs a command in them. With `--user` it creates a user namespace, and `--map-root-user` (short form `-r`) installs the simplest possible map: your own uid becomes 0 inside. No sudo on this command — run it as a plain user.

```bash
id         # before: an ordinary user
unshare --user --map-root-user bash
id         # now inside the new user namespace
cat /proc/self/uid_map
```

Enter a user namespace and inspect the identity it grants.

```
uid=1000(alice) gid=1000(alice) groups=1000(alice)
# (a new shell starts here)
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
         0       1000          1
```

Inside the new shell `id` reports uid 0 — you are root. But the uid_map tells the true story: inside-id 0 maps to outside-id 1000, count 1. Your single uid, and only it, was mapped. The kernel will let this root chown and chmod files it owns inside the namespace, but the moment it reaches for anything on the host it is judged as uid 1000, alice, exactly as powerless as before. To see that from the outside, leave this shell open and look at the process from another terminal.

```bash
ps -o pid,uid,cmd -C bash   # on the host
```

What uid does the host think the in-namespace root runs as?

```
OUTPUT
  PID  UID CMD
 7314 1000 bash
```

There it is, in two numbers. Inside, uid 0. Outside, uid 1000. The same process, two identities, and the host's identity is the unprivileged one. Now confirm that inside the namespace you really do hold capabilities — the powers that let you go on to build the other namespaces.

```bash
capsh --print | head -2   # inside the user namespace
```

A full capability set, valid only within this namespace.

```
OUTPUT
Current: =ep
Bounding set: cap_chown,cap_dac_override,...,cap_sys_admin
```

A complete bounding set, CAP_SYS_ADMIN included — the very capability Chapter 4 told you to fear. But here it is declawed: it authorizes operations only against resources the namespace owns. You can mount a filesystem your namespace controls; you cannot mount over the host's /etc. The capability is real, its reach is not. That is the bargain a user namespace strikes, and it is what makes the next step safe.

A rootless mini-container in one command

Stack the namespaces and you have built, with no privilege at all, the skeleton of a container. Each flag adds one namespace; `--map-root-user` supplies the in-namespace root that authorizes the rest, `--pid --fork --mount-proc` give a private process tree with a correct /proc, `--net` an empty network stack, and `--uts` an isolated hostname.

```bash
unshare --user --map-root-user --pid --fork \
        --mount-proc --net --uts bash
```

Five namespaces, one unprivileged command, zero sudo.

Inside the resulting shell you are root, you are PID-isolated, and you can change the hostname — all things an ordinary user cannot normally do.

```bash
id
hostname sandbox
hostname
ps -ef
```

Prove the isolation from inside the rootless sandbox.

```
OUTPUT

uid=0(root) gid=0(root) groups=0(root)
sandbox
UID   PID PPID CMD
root    1    0 bash
root    8    1 ps -ef
```

Bash is PID 1; the host's hundreds of processes are gone from view; the hostname is yours to set; you are root. Yet to the host this is still alice, uid 1000, holding no host capabilities whatsoever. You have assembled the load-bearing parts of a container — and a real rootless runtime does little more than this, plus an overlay root filesystem, a userspace network helper such as slirp4netns, and the subuid range so that more than one uid can be mapped.

Origins:

The user namespace was the last and hardest namespace to land, and the wait was long: the first namespace (mount) arrived in 2002, but user namespaces were not usable until kernel 3.8 in February 2013 — the culmination of years of painstaking work by Eric W. Biederman. Getting it wrong meant handing root to the world, so every credential check in the kernel had to be audited. By a tidy historical rhyme, that same month — March 2013 is when Docker was first shown in public — the piece that would make containers truly safe and the tool that made them popular arrived almost together, though it took years more for the two to meet.

An honest word on risk

User namespaces are not free of danger, and pretending otherwise would betray the spirit of this book. By letting unprivileged users reach code paths once gated behind root — mount, network configuration, and more — they enlarged the kernel's attack surface considerably. A string of local privilege-escalation CVEs over the years has had unprivileged user namespaces somewhere in the exploit chain. The defenders' response was to make the feature optional. Several distributions ship a switch to restrict or disable

unprivileged user-namespace creation, and a global ceiling exists on every system.

```bash
# Debian/Ubuntu: gate unprivileged userns creation
sysctl kernel.unprivileged_userns_clone
# Everywhere: a hard ceiling on namespaces per ns
sysctl user.max_user_namespaces
```

The two knobs that govern unprivileged user namespaces.

```
kernel.unprivileged_userns_clone = 1
user.max_user_namespaces = 63029
```

Set `kernel.unprivileged_userns_clone` to 0 and the very first `unshare --user` in this chapter fails for non-root users; set `user.max_user_namespaces` to 0 and even root cannot create one in that namespace. This is the eternal security trade: the feature that lets you run containers *without* root is the same feature that gives unprivileged code more kernel to poke at. Most modern distributions judge the trade worth it and leave it on, but a hardened or multi-tenant host may not.

Why the two roots being different is the whole point

Step back and the payoff is plain in three settings. In image builds, a Dockerfile that wants to `apt-get install` and write across the filesystem needs to feel like root — and with a user namespace it can, while the build never touches host root; this is how rootless `buildah` and `docker build` in rootless mode work. In CI, runners can hand developers container-root for their jobs without granting the build agent itself any host privilege, so a malicious build escapes into a mapped nobody rather than onto the runner. And in multi-tenant systems, two tenants can each be root in their own namespaces while mapping to *different* host uid ranges, so neither can read the other's files even if isolation elsewhere slips. In each case the same sentence does the work: root inside does not mean root outside.

Try it yourself

Three experiments, building on each other. First, enter a user namespace as an ordinary user and read the map that betrays your true identity:

```bash
unshare -U -r bash    # -U = --user, -r = map-root
id
cat /proc/self/uid_map
```

Become namespace-root and inspect the one-line uid map.

OUTPUT

```
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
         0       1000          1
```

Second, with that shell still open, look at the same process from the host and read off the uid the kernel actually enforces against:

BASH

```
ps -o pid,uid,cmd -C bash   # on the host
```

The in-namespace root's real, unprivileged host uid.

OUTPUT

```
  PID  UID CMD
 7314 1000 bash
```

Third, build the rootless multi-namespace shell and confirm the isolation from inside — a container's worth of separation, conjured without ever typing sudo:

BASH

```
unshare --user --map-root-user --pid --fork \
        --mount-proc --net --uts bash
id && hostname box && hostname && ps -ef
```

The whole chapter in one unprivileged command.

OUTPUT

```
uid=0(root) gid=0(root) groups=0(root)
box
UID   PID PPID CMD
root    1    0 bash
```

That uid 0, sovereign inside and powerless outside, is the quiet engineering marvel this chapter set out to explain. The other restraints in this book limit what root may do; the user namespace changes who root *is*. Lean on it, and the worst case stops being catastrophe and becomes merely a nuisance: an attacker who breaks all the way out arrives as a user with no name, no home, and nothing worth stealing the keys to.

PART III

3.4 Control Groups: What a Process Can Use

Limiting CPU, memory, and I/O with cgroups v2

Namespaces, which you met in the previous chapters, answer one question: what can a process *see*? They draw the walls of the container's world — its own process tree, its own network stack, its own mounts. But a process trapped behind perfect walls can still burn every core on the machine, allocate memory until the host falls over, or fork itself into oblivion. Walls control visibility. They say nothing about appetite.

That is the job of the second restraint, and it is entirely orthogonal to the first. Namespaces govern what a process can see; control groups — cgroups — govern what it can *use*. The two are independent axes. You can put a process in a fresh set of namespaces and leave it free to consume the whole machine. You can leave a process fully visible in the host's process list and cap it at half a CPU and a hundred megabytes. A real container is the intersection: isolated *and* bounded. This chapter is about the second half.

Accounting and limiting, in a hierarchy

A cgroup is a group of processes that the kernel tracks and throttles together. Cgroups form a tree: every process belongs to exactly one node, child nodes inherit the constraints of their parents, and a budget handed to a parent is shared among all its descendants. Onto each node you attach *controllers* — the kernel subsystems that do the measuring and the capping.

The controllers you will actually reach for:

The controllers:

cpu — weight (relative share) + bandwidth quota memory — hard limit, accounting, and OOM handling io — block-device throughput weight and caps pids — a ceiling on the number of processes cpuset — pin tasks to specific CPUs / NUMA nodes

Two verbs run through all of them: *accounting* and *limiting*. The kernel always counts what a group consumes — its CPU time, its resident memory, its I/O — whether or not you have set a ceiling. A limit is then optionally laid on top of the count. This is why cgroups power both `docker stats` (pure accounting) and `docker run --memory` (a hard limit): same machinery, read versus write. The `pids` controller is the quiet hero here — it is the thing standing between a fork bomb inside a container and a host that can no longer spawn a shell to kill it.

v1, v2, and the unified hierarchy

Cgroups have two generations, and the difference is structural. In v1, every controller had its *own* separate hierarchy: a tree for `cpu`, a different tree for `memory`, another for `blkio`, and a process could sit at unrelated positions in each. This was flexible and a genuine mess — controllers could disagree about which group a task belonged to, and tooling had to reconcile half a dozen mount points.

Cgroup v2 — the *unified hierarchy* — fixes this by having exactly one tree. A process lives at one place in it, and each node simply enables the subset of controllers it wants. Cleaner, coherent, and now the default on every modern systemd distribution. Docker used v1 for years and supports both, but new systems are v2, and v2 is what this chapter teaches. Check which one you are on by asking the filesystem its type — `cgroup2fs` means the unified hierarchy:

BASH

```bash
stat -fc %T /sys/fs/cgroup
```

cgroup2fs = v2 (unified); tmpfs = legacy v1 layout.

OUTPUT

```
cgroup2fs
```

The interface is a filesystem

Here the callback from chapter 3 pays off in full. The cgroup v2 interface is not a system call you invoke or a library you link. It is a pseudo-filesystem, `cgroup2`, mounted at `/sys/fs/cgroup`, and you operate it with the same `mkdir`, `echo`, and `cat` you use on any directory. The mechanics are almost suspiciously simple:

The whole API:

create a group -> mkdir a subdirectory add a process -> echo PID > cgroup.procs set a limit -> echo VALUE > memory.max read usage -> cat memory.current destroy a group -> rmdir the directory

Every node is a directory, and the kernel populates it with control files. The ones that matter:

Key v2 files:

cgroup.controllers — controllers available here cgroup.subtree_control — controllers enabled for children cgroup.procs — PIDs in this group cpu.max — "QUOTA PERIOD" bandwidth cap cpu.weight — relative CPU share (default 100) memory.max — hard memory ceiling memory.current — bytes currently charged memory.events — counters incl. oom_kill pids.max — max processes in the group

Two files deserve a closer look. `cpu.max` holds two numbers, `QUOTA PERIOD`, both in microseconds: the group may run for at most `QUOTA` microseconds out of every `PERIOD` microseconds of wall time. So `50000 100000` means "50,000 us of CPU per 100,000 us" — half of one CPU. Writing `100000 100000` grants one whole CPU; `200000 100000` grants two. The default is the literal word `max`, meaning no cap.

The other is `cgroup.subtree_control`, and it trips up everyone the first time. A controller is only usable in a child if the parent has *delegated* it downward. You do that by writing `+memory +cpu` (and so on) into the parent's `cgroup.subtree_control`. Until you do, your shiny new child directory will have no `memory.max` file at all. Enable in the parent; use in the child.

Building one by hand

Enough theory. We are going to carve out a group, cap it at half a CPU and a hundred megabytes, drop a shell into it, and then watch the kernel hold the line. This needs root (or a properly delegated cgroup); run it on a v2 Linux box, a VM, or WSL2. Start by creating the group and delegating the controllers we want into it:

BASH

```bash
cd /sys/fs/cgroup
# delegate cpu+memory+pids to our children
echo '+cpu +memory +pids' > cgroup.subtree_control
mkdir demo
cat demo/cgroup.controllers
```

mkdir makes the node; the child inherits the delegated set.

OUTPUT

```
cpu memory pids
```

The child reports exactly the controllers the parent handed down. Now set the limits — half a CPU, 100 MB of memory, and a cap of 64 processes for good measure:

```bash
echo '50000 100000' > demo/cpu.max   # 50% of 1 CPU
echo 100M           > demo/memory.max
echo 64             > demo/pids.max
cat demo/cpu.max demo/memory.max
```

Writing the files IS setting the limits — nothing else.

```
OUTPUT
50000 100000
104857600
```

Note that the kernel echoed `memory.max` back as bytes — it accepted the human `100M` and normalised it. To put a process under these limits, write its PID into `cgroup.procs`. The handy trick is to move the current shell itself, using `$$`; every command you launch from then on is its child and inherits the group:

```bash
echo $$ > demo/cgroup.procs
cat /proc/self/cgroup        # confirm where we landed
```

Move this shell in; descendants join automatically.

```
OUTPUT
0::/demo
```

Watching the CPU cap bite

The shell now lives in `/demo`, capped at half a CPU. Launch a pure CPU burner — `yes` spinning into the void — and look at it with `top`. On an idle core a single `yes` would happily eat 100%. Inside the group it cannot:

```bash
yes > /dev/null &
top -p $!        # press q to quit
```

One CPU hog, throttled by cpu.max to half a core.

```
OUTPUT
  PID USER  %CPU %MEM    TIME+ COMMAND
 4812 root  50.0  0.0  0:07.41 yes
```

Fifty percent, pinned there by the kernel's bandwidth scheduler, no matter how hard `yes` tries. The process is not aware it is being throttled; it simply gets scheduled for 50,000 us out of every 100,000 and slept the rest. Kill it with `kill %1` before moving on.

Watching the OOM killer fire

Now the memory limit. We set `memory.max` to 100 MB. Allocate past that and the kernel does not slow you down — it kills you. When a group's charged memory would exceed its hard limit and nothing can be reclaimed, the cgroup's own OOM killer reaps a process inside the group. Provoke it with a small allocator (here, a one-liner that keeps appending megabyte chunks to a list):

```bash
python3 -c '
b=[]
while True: b.append(bytearray(1024*1024))' &
wait
cat demo/memory.events
```

Allocate past memory.max; the group's OOM killer fires.

```
[1]  + killed     python3 -c ...
low 0
high 0
max 0
oom 1
oom_kill 1
```

The process is dead — `killed`, not merely failed — and `memory.events` records it: `oom_kill 1`. Run the allocator again and the counter climbs to 2. This is the same file Docker reads to tell you a container was OOM-killed, and the same limit that produces the dreaded exit code 137. The kernel held the host's memory safe by sacrificing the offender, and only the offender — processes outside `/demo` never noticed. Clean up when done with `cd /sys/fs/cgroup && rmdir demo` (after killing anything left inside it).

This is exactly what Docker does

Nothing you just typed was a toy version of containers. It was the real mechanism. When you run `docker run --cpus=0.5 --memory=100m --pids-limit=64 ...`, the runtime creates a cgroup for the container and writes the very same files: `0.5` becomes `50000 100000` in `cpu.max`, `100m` becomes `104857600` in `memory.max`, and `--pids-limit` becomes `pids.max`. The flags are a friendly skin over the directory operations you performed by hand.

You can watch it happen. Because systemd organises cgroups into slices, scopes, and services, a Docker container's group lands under `system.slice` as a scope named for its ID. Start a limited container and read its limits straight off the filesystem:

BASH

```bash
id=$(docker run -d --cpus=0.5 --memory=100m \
       --pids-limit=64 alpine sleep 600)
cg=/sys/fs/cgroup/system.slice/docker-$id.scope
cat $cg/cpu.max $cg/memory.max $cg/pids.max
cat $cg/memory.current
```

The container's cgroup, read with plain cat.

OUTPUT

```
50000 100000
104857600
64
1908736
```

There they are: the flags you passed on the command line, translated verbatim into cgroup v2 control files, plus a live `memory.current` reading of what the container is using right now — the number behind `docker stats`. The container's resource limits are not an abstraction inside the Docker daemon. They are files in `/sys/fs/cgroup`, written by the runtime, enforced by the kernel. systemd manages the surrounding tree, which is why a well-behaved runtime *delegates* through systemd rather than scribbling into the root group directly.

Origins:

Cgroups began at Google in 2006, written by Paul Menage and Rohit Seth under the name "process containers." Renamed control groups to avoid clashing with that loaded word, they merged into the Linux kernel 2.6.24 in January 2008 — built to pack and bound thousands of jobs per machine across Google's fleet, the lineage that became Borg. The v2 rewrite, the unified hierarchy, was led by Tejun Heo and stabilized around kernel 4.5 in 2016.

Try it yourself

Three experiments, the same three the chapter walked through, in one reproducible block. Run as root on a cgroup v2 box. First cap CPU and watch `yes` get throttled; then set a low memory ceiling and watch the OOM killer fire; finally, read the limits Docker set for a real running container.

```bash
# 0. Confirm v2, then build a delegated group.
stat -fc %T /sys/fs/cgroup        # expect cgroup2fs
cd /sys/fs/cgroup
echo '+cpu +memory +pids' > cgroup.subtree_control
mkdir demo

# 1. Cap CPU to 50% and throttle a burner.
echo '50000 100000' > demo/cpu.max
echo $$ > demo/cgroup.procs
yes > /dev/null &
top -bn1 -p $! | tail -1          # ~50% CPU
kill %1

# 2. Low memory ceiling -> OOM kill.
echo 50M > demo/memory.max
python3 -c 'b=[]
while True: b.append(bytearray(1<<20))' ; true
cat demo/memory.events | grep oom_kill

# 3. Read a live Docker container's limits.
id=$(docker run -d --cpus=0.5 --memory=100m \
       alpine sleep 300)
cg=/sys/fs/cgroup/system.slice/docker-$id.scope
cat $cg/cpu.max $cg/memory.max $cg/memory.current

# Clean up.
docker rm -f $id
cd /sys/fs/cgroup && rmdir demo
```

Throttle, OOM-kill, then read Docker's own cgroup.

```
OUTPUT (abridged)

cgroup2fs
 4812 root   50.0  0.0  0:03.10 yes
oom_kill 1
50000 100000
104857600
2027520
```

Read it back against the chapter. Step 1's `yes` is held at 50% by `cpu.max`. Step 2's allocator is `killed` and `memory.events` confirms `oom_kill 1`. Step 3 shows a real container's limits living as ordinary files. You have now exercised the second restraint by hand — and seen that `docker run --cpus --memory --pids-limit` is nothing more than these writes, performed for you. With namespaces controlling what a process sees and cgroups controlling what it uses, you hold both halves of a container. What remains is to assemble them into one.

PART III

3.5 Overlay Filesystems and Why Images Are Layers

Copy-on-write, lowerdir/upperdir, and the writable layer

In chapter 3 you learned that a container's "/" is just an ordinary directory the kernel was told to treat as root. That leaves a sharp question unanswered: where does that directory come from, and what is in it? A process expects a complete world — `/bin`, `/lib`, `/etc`, `/usr` — hundreds of megabytes of files it never created but cannot run without. Something has to put a full root filesystem in front of every container. This chapter is about the trick that does it, and about why that trick is the reason Docker images are built out of layers in the first place.

The problem: a full OS per container is unaffordable

Imagine the naive design. Every time you start a container, you copy a whole base image — a Debian or Alpine userland, say — into a fresh directory and chroot into it. It would work. It would also be ruinous. A modest base is tens to hundreds of megabytes; ten containers from the same image would mean ten identical copies on disk, and the copy itself would take seconds before the container even started. Run a hundred short-lived containers and you have copied the same `/usr/bin/python3` a hundred times.

The waste is obvious because almost none of those bytes ever change. A container started from an image touches a handful of files — writes a PID file, appends a log, drops a socket — and leaves the other thousands exactly as they shipped. The read-only common parts are shared by every container of that image; only the tiny set of modifications is private. So the right design is the one that shares the read-only bulk and gives each container only a thin, private, writable scratch space stacked on top. That is precisely what a union (or overlay) filesystem provides.

overlayfs: one merged tree from stacked directories

overlayfs is the mainline Linux union filesystem, in the kernel tree since version 3.18 (released December 2014). It does not store files of its own. Instead it presents a single merged directory tree synthesized, on the fly, from directories that already exist on some ordinary backing filesystem. You hand it four roles:

The four directories:

lowerdir — one or more READ-ONLY layers, stacked. upperdir — the single READ-WRITE layer; all changes land here. workdir — an empty scratch dir overlayfs needs internally. merged — the mountpoint where the union appears.

The lower layers are the interesting part. You can give several, separated by colons, and the leftmost is the top of the read-only stack: `lowerdir=top:middle:bottom`. These stacked read-only directories are exactly what an image's layers are. The upperdir is a single writable directory laid over all of them. The workdir must live on the same filesystem as the upperdir, because overlayfs uses it to stage operations and then rename them into place atomically — a rename is only atomic within one filesystem. You never look inside workdir; you just give overlayfs an empty one and leave it alone.

Copy-on-write: read down, write up

Everything overlayfs does follows from one rule: reads look downward through the stack, and writes are forced upward into the upperdir. The four cases are worth stating exactly, because container behavior is just these four cases playing out.

Reading a file. overlayfs serves it from the upperdir if a copy exists there; otherwise it walks the lower layers top to bottom and serves the first one that has the file. The leftmost lower that contains a given path wins. Nothing is copied; the read is served straight from whichever layer owns the file.

Modifying a file that exists only in a lower layer. This is the subtle one, and it is the operation people mean when they say "copy-on-write." The lower layers are read-only and must stay untouched, so before your write can proceed overlayfs performs a copy-up: it copies the entire file from the lower layer into the upperdir, then applies your change to that copy. From then on the upper copy shadows the lower original, which is never modified. Note the cost — copy-up duplicates the whole file the first time you touch it, even if you only flip one byte. A one-byte edit to a 2 GB file copies 2 GB. This is why write-heavy workloads on large files want a volume (chapter 18), not the container layer.

Creating a new file. It simply goes into the upperdir, since no lower owns that path. No copy-up is involved.

Deleting a file that exists in a lower layer. overlayfs cannot remove it from the read-only lower, so it masks it: it creates a whiteout in the upperdir. A whiteout is a character device with device numbers 0/0 sitting at the deleted file's path. When overlayfs sees a whiteout in an upper layer, it treats the path as absent in the merged view and stops looking further down. The lower file still exists on disk; the merged tree just refuses to show it.

Build one by hand

None of this is Docker-specific, and the fastest way to believe it is to mount an overlay yourself with `mount -t overlay`. Start by laying out two lower layers, an upper, a workdir, and a mountpoint. Put a file in each lower, plus one same-named file in both lowers so you can watch precedence resolve.

```bash
cd /tmp && rm -rf olab && mkdir olab && cd olab
mkdir lower1 lower2 upper work merged
echo 'only in lower1' > lower1/a.txt
echo 'only in lower2' > lower2/b.txt
echo 'from lower1'    > lower1/shared.txt
echo 'from lower2'    > lower2/shared.txt
```

Two read-only layers; shared.txt exists in both.

Now mount the overlay. Put lower2 leftmost so it sits on top of lower1 in the read-only stack, hang upper and work off it, and point the result at merged:

```bash
sudo mount -t overlay overlay \
  -o lowerdir=lower2:lower1,upperdir=upper,workdir=work \
  merged
ls merged
cat merged/shared.txt
```

lower2 is leftmost, so it wins ties over lower1.

```
a.txt  b.txt  shared.txt
from lower2
```

The merged tree is the union: `a.txt` from lower1, `b.txt` from lower2, and a single `shared.txt`. Because lower2 was leftmost, its `shared.txt` shadows lower1's — the top layer wins, exactly as a later image layer overrides an earlier one. Now trigger a copy-up by editing a file that lives only in a lower:

```bash
echo 'edited in merged' >> merged/a.txt
cat upper/a.txt          # the copied-up version
cat lower1/a.txt         # the lower is untouched
```

Writing a lower-only file copies it up into upper.

```
OUTPUT

only in lower1
edited in merged
only in lower1
```

There it is: `upper/a.txt` now holds the full file plus your appended line, while `lower1/a.txt` still reads exactly as it did before. The write never reached the lower layer; overlayfs copied the file up and edited the copy. Finally, delete a lower-only file through the merged view and inspect what landed in upper:

```bash
rm merged/b.txt
ls merged                # b.txt is gone from the union
ls -l upper/b.txt        # but a whiteout appeared in upper
```

A delete becomes a whiteout: a 0/0 char device.

```
OUTPUT

a.txt  shared.txt
c--------- 1 root root 0, 0 Jun 26 10:14 upper/b.txt
```

The `c` and the `0, 0` are the tell: `upper/b.txt` is now a character device with major/minor 0/0 — a whiteout. `lower2/b.txt` is still sitting there on disk, but overlayfs hides it because the whiteout above it says "deleted." Unmount with `sudo umount /tmp/olab/merged` and look at the four directories again: lower1 and lower2 are exactly as you created them, and `upper` holds precisely the diff — the copied-up `a.txt` and the `b.txt` whiteout. That `upper` directory, a directory of just the changes, is the shape of every image layer.

Why images are layers

Now connect it hard to Docker. Each instruction in a Dockerfile that changes the filesystem — a `RUN`, a `COPY`, an `ADD` — produces a new layer, and a layer is nothing more than the `upper`-style diff you just produced by hand: a directory containing only the files that instruction added, changed (copied up), or deleted (as whiteouts). An image is an ordered stack of these read-only diffs. To assemble the image's root filesystem, Docker mounts every layer as a lowerdir, in order, with the most recent instruction's layer leftmost.

When you `docker run`, Docker takes that read-only stack of image layers as the lowerdirs and adds exactly one fresh, empty writable directory as the upperdir. That upper is the container layer. It is why containers start instantly — no copy happens, only a mount — and it is why everything a running container writes goes into that one thin layer. It also explains the property that surprises newcomers: remove the container and its writable upper is discarded, so every change made inside vanishes. The image layers underneath were never touched. To keep data, you mount a volume past the overlay (chapter 18); the container layer is scratch space by design.

And because the lowerdirs are read-only and shared, two containers started from the same image point their overlays at the very same lower directories on disk. The base userland is stored once and mounted into a hundred containers; each gets only its own private upper. That is the deduplication that makes containers cheap. The naive copy-the-whole-OS design from the start of the chapter is replaced by share-the-lowers, write-to-a-private-upper — which is just overlayfs, aimed deliberately.

One more thread to pick up in chapter 15: layers are identified by content, not by name. Docker hashes each layer's contents into a sha256 digest, and that digest is the layer's identity. Two images that share a base layer share the identical digest, so the layer is downloaded, stored, and mounted exactly once. Content-addressing is what turns the layer model into a caching and sharing system; chapter 15 gives the image format the full treatment.

The storage drivers, briefly

overlayfs is not the only way Docker has assembled root filesystems, and the component that does the assembling is called a storage driver. Docker's original driver was aufs, an out-of-tree union filesystem that was never merged into mainline Linux — which made it a packaging headache for years. Others used devicemapper (block-level copy-on-write via LVM thin pools) or the copy-on-write snapshots built into btrfs and zfs. The modern default on essentially every current Linux host is overlay2, the storage driver built on the kernel's overlayfs, using multiple lowerdirs the way you did above. Chapter 18 returns to storage drivers and to volumes in earnest; for now, overlay2 is the one that matters.

Origins:

Union mounts have a long lineage. Plan 9 from Bell Labs let you bind several directories into one union directory in the late 1980s. UnionFS brought the idea to Linux; aufs followed and was what Docker first shipped with — yet aufs was never accepted into the mainline kernel. overlayfs, the work led by Miklos Szeredi at Red Hat, was merged into Linux 3.18 in 2014 and became the union filesystem Docker now relies on.

Try it yourself

Two experiments. First, the by-hand overlay from above, condensed, so you can watch precedence, copy-up, and a whiteout in one run. Run it on a Linux box (or WSL2 / a VM); it is reversible and harmless.

```bash
cd /tmp && rm -rf olab && mkdir olab && cd olab
mkdir lower1 lower2 upper work merged
echo from-l1 > lower1/shared.txt
echo from-l2 > lower2/shared.txt
echo only-l1 > lower1/a.txt
sudo mount -t overlay overlay \
  -o lowerdir=lower2:lower1,upperdir=upper,workdir=work \
  merged
cat merged/shared.txt      # precedence: top layer wins
echo more >> merged/a.txt  # forces a copy-up
ls upper                   # a.txt now present in upper
rm merged/shared.txt       # creates a whiteout
ls -l upper/shared.txt     # 0/0 char device = whiteout
sudo umount /tmp/olab/merged
```

Precedence, copy-up, and a whiteout in one sitting.

```
OUTPUT (abridged)
from-l2
a.txt
c--------- 1 root root 0, 0 Jun 26 10:14 upper/shared.txt
```

Second, look at what Docker actually mounted. Start any container in the background and grep the host's mount table for its overlay. You will see the same options you just typed by hand — a lowerdir of stacked image layers, a single upperdir (the container layer), and a workdir — proving Docker is doing nothing more exotic than the mount above, at scale.

```bash
docker run -d --name ovl alpine sleep 300 >/dev/null
mount | grep -m1 overlay | tr ',' '\n' | head
docker rm -f ovl >/dev/null
```

Docker's real overlay mount: lowerdir/upperdir/workdir.

```
OUTPUT (abridged)
overlay on /var/lib/docker/overlay2/<id>/merged
  type overlay (rw,relatime
lowerdir=/var/lib/docker/overlay2/l/AAA:.../l/BBB
upperdir=/var/lib/docker/overlay2/<id>/diff
workdir=/var/lib/docker/overlay2/<id>/work)
```

Read it against the chapter. The `lowerdir` is a colon-separated stack of image layers, read-only and shared with every other container of that image. The `upperdir` ends in `/diff` — that is the container's private writable layer, the one discarded when you `docker rm` it. The `workdir` is overlayfs's atomic scratch space. The container thinks its disk is a clean, complete `/`; it is in fact a union you could have built yourself in `/tmp`, which is exactly what you just did.

PART III

3.6 Locking the Door

Capabilities, seccomp, no_new_privs, and LSMs

We have spent this book taking the contained process apart restraint by restraint. Namespaces (Chapters 7 through 10) change what the process can *see* — its own process tree, its own network, its own mount table. Cgroups (Chapter 11) change how much it can *consume* — CPU, memory, I/O. Both are real and both matter. But neither touches a stubborn fact we established back in Chapter 1: the process is still running directly on the host kernel, and it can still attempt any system call that kernel implements. A namespace hides the host's other processes; it does not stop the contained process from calling `init_module` to load a rootkit, or `reboot` to power-cycle the machine, or `mount` to graft new filesystems into its world. The restraints so far are walls and meters. This chapter is about the last one, which is different in kind: it shrinks *what the process is allowed to ask the kernel for in the first place*.

Recall the boundary from Chapter 2. Everything a process wants from the outside world is a syscall, and the kernel guards that boundary absolutely. The final restraint works right at the gate. It does not hide the dangerous operation or rate-limit it — it makes the request itself fail, before the kernel ever acts on it. This is defense in depth in its purest form. Even if an attacker finds a gap in the namespace isolation, a process that physically cannot issue the dangerous syscall cannot exploit it. You are not trusting the wall to be perfect; you are also removing the tools.

Capabilities, revisited as a default policy

Chapter 4 introduced capabilities as the project of cutting root's all-or-nothing power into about forty independent pieces. There we cared about the mechanism. Here we care about the *policy* Docker applies with it, because that policy is the first thing standing between a compromised container and your host. When you `docker run` an ordinary container, the runtime does not hand the container's root the full set. It starts from nothing and grants back a small, deliberately conservative bounding set — historically about fourteen capabilities. Everything else is gone, and gone from the bounding set, which (Chapter 4) means it can never be regained, not even by executing a setuid binary.

What a default Docker container KEEPS:

CHOWN change a file's owner DAC_OVERRIDE bypass the rwx permission checks FOWNER / FSETID act as a file's owner; keep setuid bits KILL send signals regardless of ownership SETGID / SETUID change the process's gid / uid SETPCAP move capabilities between its own sets NET_BIND_SERVICE bind to ports below 1024 NET_RAW open raw sockets (this is how ping works) SYS_CHROOT call chroot() MKNOD create the special device nodes it needs AUDIT_WRITE write records to the kernel audit log SETFCAP set file capabilities

That list is chosen so a typical image can install packages, drop privileges to a service account, and bind port 80 — the everyday work of a server — without anything more. Now read what is conspicuously *absent*, because the omissions are the whole point.

What a default Docker container DROPS:

SYS_ADMIN mount, pivot_root, and a hundred more SYS_MODULE load and unload kernel modules SYS_PTRACE trace and inspect other processes SYS_BOOT reboot or kexec the machine SYS_TIME set the system clock NET_ADMIN reconfigure interfaces, routes, firewalls SYS_RAWIO raw I/O port and /dev/mem access ... and roughly two dozen more

SYS_ADMIN alone is why this matters. As Chapter 4 warned, so many privileged operations were filed under it that holding it is very nearly holding root. Dropping it — together with SYS_MODULE, SYS_PTRACE, and SYS_BOOT — is what makes a container's "root" tolerable despite being, by default, the host's actual uid 0. The capability set is the difference between a root that can chown files and a root that can load a kernel module.

The operational rule that falls out of this is simple. The default set is already small, but it is still a one-size compromise. The best practice for a service you control is to drop everything and add back only what that one service actually needs:

```bash
docker run --cap-drop ALL \
  --cap-add NET_BIND_SERVICE nginx
```

Empty the bounding set, then grant exactly one power back.

At the opposite, dangerous extreme sits `--privileged`. It is tempting to read it as "give the container root," but the container already has root. What `--privileged` actually does is *remove the restraints*: it grants the full capability set, exposes all host devices under /dev, and runs the container unconfined by seccomp and the LSM profile. It is not one switch but the simultaneous defeat of nearly everything this chapter describes. Reach for it only when you genuinely mean "this container is part of the host," and know that an escape from it is an escape onto the machine.

Reading a container's capabilities

You do not have to take the list on faith. The kernel publishes each process's capability sets as hexadecimal bitmasks in /proc/<pid>/status, and a container can read its own.

```bash
docker run --rm alpine \
  grep Cap /proc/self/status
```

The five capability masks of a default container's process.

```
CapInh: 0000000000000000
CapPrm: 00000000a80425fb
CapEff: 00000000a80425fb
CapBnd: 00000000a80425fb
CapAmb: 0000000000000000
```

CapEff is the effective set — the one the kernel checks at the moment of a syscall. The hex is unreadable on its own, but `capsh` will decode it into names, and now the policy above appears in concrete form.

```bash
capsh --decode=00000000a80425fb
```

Turn the effective-set bitmask into a list of capabilities.

```
0x00000000a80425fb=cap_chown,cap_dac_override,
cap_fowner,cap_fsetid,cap_kill,cap_setgid,
cap_setuid,cap_setpcap,cap_net_bind_service,
cap_net_raw,cap_sys_chroot,cap_mknod,
cap_audit_write,cap_setfcap
```

Fourteen names, exactly the kept set — and no cap_sys_admin anywhere in it. Run the same container with `--cap-drop ALL` and CapEff reads all zeros; run it `--privileged` and the mask fills with every capability the kernel offers. The bitmask is the policy, made auditable in one line.

Seccomp: filtering the syscalls themselves

Capabilities gate *privileged* operations, but they are coarse. Many syscalls need no capability at all and are still dangerous as attack surface — an obscure, buggy syscall is a buggy syscall whether or not it is privileged. Seccomp, secure computing mode, attacks the problem from the other direction: it filters syscalls by *number*, regardless of who is asking.

Modern seccomp is mode 2, "seccomp-bpf." A process installs a small BPF program — a classic, in-kernel bytecode filter — and from then on the kernel runs that filter on entry to every syscall the process makes. The filter sees the syscall number and the scalar (non-pointer) argument values, and it returns one action. The important actions are:

Seccomp filter actions:

ALLOW let the syscall proceed normally ERRNO block it, returning a chosen error (e.g. EPERM) KILL terminate the process (SIGSYS) immediately TRAP raise SIGSYS so a handler can intercept it LOG allow it but record it (useful for building rules)

The ERRNO action is subtler than KILL and is what Docker leans on: the blocked syscall does not crash the program, it simply *fails* as if the kernel had refused it, returning a normal error the program already knows how to handle. A filter is installed through prctl(PR_SET_SECCOMP) or the dedicated seccomp() syscall, and it has two properties that make it trustworthy. Once installed it cannot be removed or loosened — a process can only ever *narrow* its own filter. And it is inherited across fork and execve, so a child cannot shed it by running a new program. That second property is also why an unprivileged process may only install a filter if it has first set no_new_privs, which we come to next: without that, a filter could be used to confuse a setuid binary into misbehaving, turning the sandbox into an escalation.

Docker ships a default seccomp profile — a JSON document that is essentially an allowlist. It permits the great majority of syscalls (something over 300 of the roughly 400 the kernel exposes) and blocks the dangerous remainder: `keyctl` and `add_key` (kernel keyring tampering), `mount` and `umount2`, `reboot`, `swapon`, `kexec_load`, `init_module`, `finit_module`, `ptrace` under most configurations, and more. You can replace it or remove it with --security-opt:

```bash
# Disable the filter entirely (dangerous):
docker run --security-opt seccomp=unconfined alpine
# Or supply your own allowlist:
docker run --security-opt seccomp=profile.json app
```

Turning the seccomp profile off, or pointing at a custom one.

Seccomp in miniature

The mechanism is small enough to hold in your head, and a few lines of C make it concrete. This program installs a filter that kills the process the moment it calls `mkdir`, then calls it. Using libseccomp keeps the BPF generation out of the way; the shape is the whole lesson.

```c
#include <seccomp.h>
#include <sys/prctl.h>
#include <sys/stat.h>

int main(void) {
    /* default action: allow everything... */
    scmp_filter_ctx ctx =
        seccomp_init(SCMP_ACT_ALLOW);
    /* ...except mkdir, which is killed */
    seccomp_rule_add(ctx, SCMP_ACT_KILL,
                     SCMP_SYS(mkdir), 0);
    seccomp_load(ctx);   /* installs the BPF */
    mkdir("/tmp/x", 0755);  /* never returns */
    return 0;
}
```

Allow all syscalls but mkdir; mkdir triggers SIGSYS.

Compile it with `-lseccomp` and run it, and the kernel does not return an error from `mkdir` — it kills the process outright with SIGSYS, the signal that means "bad system call."

```
OUTPUT
$ ./block_mkdir
Bad system call (core dumped)
```

That is the entire engine behind Docker's seccomp profile, scaled up: a default verdict, a table of per-syscall exceptions, loaded once and inherited forever. Docker's profile inverts this toy's defaults — it allows the listed syscalls and returns ERRNO for the rest — but the machinery is exactly what you see here.

no_new_privs: a one-way promise

We met no_new_privs at the close of Chapter 4 as the flag that makes seccomp safe for unprivileged use; here is its full shape. Set with prctl(PR_SET_NO_NEW_PRIVS) and available since kernel 3.5, it is a single bit with a precise meaning: once on, this process and every child it ever spawns can *never gain privileges* through an execve. Setuid and setgid bits are ignored; file capabilities no longer elevate; nothing the process runs can climb higher than the process already sits. And it is strictly one-way — there is no syscall to turn it back off.

Docker sets it whenever a container runs without the full privilege of the host — that is,

whenever you drop capabilities or rely on the seccomp profile without `--privileged`. It closes a specific hole: a compromised process inside the image cannot pivot through some stray setuid-root binary to win back the capabilities Docker carefully dropped. You can confirm it the same way you read capabilities, from /proc.

```bash
docker run --rm alpine \
  grep NoNewPrivs /proc/self/status
```

Check whether the no-new-privileges bit is set.

```
NoNewPrivs:    1
```

A 1 there means the door is not only locked but the lock has been welded: no setuid binary in the image, however privileged on disk, can hand this process anything it does not already have.

LSMs: a second kernel, judging everything

Everything so far — capabilities, seccomp, no_new_privs — lives in the kernel's ordinary permission path. Linux Security Modules are a separate, *mandatory* access-control layer bolted on top. After the classic checks pass and the capability check passes, the kernel consults the active LSM, and the LSM can still say no. It is a second judge, and its verdict is not something a process can negotiate away with ownership or root, which is what "mandatory" means.

Two LSMs dominate, and which one you meet depends on your distribution. AppArmor (Ubuntu, Debian, SUSE) is *path-based*: a profile is a list of file paths a confined program may read, write, or execute, plus the capabilities and network operations it may use. Docker ships and loads a default profile called `docker-default` that, for instance, denies writes to sensitive parts of /proc and /sys even though the container's root would otherwise be allowed. SELinux (Fedora, RHEL, CentOS) is *label-based*: every process and every file carries a security label, and policy is written in terms of which label may act on which. Docker uses SELinux to label container processes and their files so that one container's label cannot touch another's, or the host's. The familiar `:z` and `:Z` suffixes on `-v` volume mounts are SELinux relabeling requests — `:z` shares a volume's label between containers, `:Z` makes it private to one.

You do not usually configure these by hand for an ordinary container; the runtime applies a sane default and gets out of the way. What matters for understanding Docker is the shape: an LSM is yet another independent layer the dangerous operation must pass, enforced by the kernel, that does not care whether the caller is root.

Defense in depth, and the caveat under it all

Step back and the four restraints of this book click together. A well-run container is a process with its view reduced by namespaces (Chapters 7 through 10), its appetite capped by cgroups (Chapter 11), and — this chapter — its requests to the kernel constrained on four fronts at once: a minimal capability set, a seccomp allowlist, no_new_privs welded on, and an LSM profile judging every operation. Layer a user namespace (Chapter 9) underneath so that container-root maps to an unprivileged host uid, and an attacker who defeats one layer still faces the next, and lands — if they land at all — as a nobody.

But honesty requires the caveat we have carried since Chapter 2: all of this runs on the *one shared kernel*. Seccomp shrinks the set of syscalls an attacker may reach, yet whatever remains on the allowlist is real kernel code executing on your behalf. A zero-day in an *allowed* syscall is a path through every layer at once — the capability check never fires, the filter waved the call through, the LSM saw nothing unusual. That irreducible risk is exactly why projects like gVisor and Kata Containers exist, interposing a second kernel or a real VM between the container and the host. We pick up that thread in the epilogue. For now, the lesson is that container security is not a wall but a stack of independent subtractions — and the more of them you keep, the smaller the surface that one kernel bug can reach.

Try it yourself

First, read a default container's effective capabilities and decode them, confirming for yourself that the catch-all SYS_ADMIN is absent. Substitute the CapEff value your own system prints into the decode call.

```bash
docker run --rm alpine \
  grep -E 'CapEff|NoNewPrivs' /proc/self/status
capsh --decode=00000000a80425fb | tr ',' '\n' \
  | grep sys_admin || echo 'sys_admin NOT held'
```

Inspect the effective set and the no-new-privs bit.

```
CapEff:    00000000a80425fb
NoNewPrivs:    1
sys_admin NOT held
```

Now watch the seccomp profile bite. The default profile blocks `mount`, so a container that tries to mount a filesystem fails — even though its root, with the dropped capability aside, might otherwise have stood a chance. Run it under the default profile, then again `unconfined`, and compare. (Mounting also needs SYS_ADMIN, so add it back to isolate seccomp as the thing being tested.)

BASH

```bash
# Default profile: the mount syscall is filtered.
docker run --rm --cap-add SYS_ADMIN alpine \
  mount -t tmpfs none /mnt
# No profile: now only the capability gates it.
docker run --rm --cap-add SYS_ADMIN \
  --security-opt seccomp=unconfined alpine \
  mount -t tmpfs none /mnt && echo mounted
```

Same mount, with the filter on, then off.

OUTPUT

```
mount: permission denied (are you root?)
mounted
```

Read the contrast carefully, because it is the chapter in two lines. The first command has the capability `mount` needs — and still fails, because seccomp refused the syscall before the capability check ever mattered. The second, with the filter removed, succeeds. Two independent layers stood between the container and a mount, and you just watched one of them hold the line on its own. That is defense in depth: not a single perfect barrier, but several imperfect ones, each narrowing what the process on your shared kernel is allowed to ask for.

PART III

3.7 Swapping the Root with pivot_root

chroot, pivot_root, and a container's real /

By now you have built almost everything a container needs out of spare kernel parts. Chapter 11 stacked an image's read-only layers under a thin writable layer and handed you a single merged directory — the container's root filesystem, sitting somewhere on the host like /var/lib/docker/.../merged. Chapter 7 gave you a private mount namespace, a mount table the container can rearrange without the host noticing. What remains is the act that joins them: convincing the process that that merged directory is not some folder buried under /var/lib — that it *is* the root of the world, the slash at the head of every absolute path.

We met the ancestor of this trick back in Chapter 5. chroot, born in 1979, can make a directory look like /. But chroot is not what containers actually use, and the reason why is the heart of this chapter. The real tool is a syscall called pivot_root, and once you understand the difference you understand the last filesystem primitive standing between you and a container built entirely by hand.

Why chroot is not enough

chroot(path) changes one thing: the directory the kernel treats as the starting point for resolving absolute pathnames in the calling process. After chroot("/new"), a process that opens /etc/passwd really opens /new/etc/passwd. It is genuinely useful, and for decades it was how people built minimal sandboxes. But it has three weaknesses that make it unfit to be a container's real boundary.

First, it is escapable by anyone holding CAP_SYS_CHROOT — which a default container root has. The classic escape is a few lines long. chroot only moves the root-for-pathname-lookup; it does not check that your current working directory is inside the new root, and an open directory file descriptor survives the call untouched. So:

```bash
# pseudo-code of the classic chroot escape
fd = open("/", O_RDONLY)   # grab a handle to old /
mkdir("escape"); chroot("escape")  # new, deeper root
fchdir(fd)                 # cwd now ABOVE the new root
for (i=0;i<256;i++) chdir("..")  # walk up to real /
chroot(".")                # adopt the host root again
```

chroot moves the root but leaves a door propped open.

Because the leaked descriptor still points at the old root, fchdir puts the working directory *outside* the chroot, and repeated chdir("..") climbs to the true filesystem root, which a

final chroot then adopts. The jail dissolves. chroot alone is not a security boundary, and the kernel developers have always said so.

Second, and more fundamental, chroot does not move any mounts. Every filesystem that was mounted before the call is still mounted, still in the process's mount table, still reachable by a process that can find a path to it. The host's disks have not gone anywhere; chroot merely changed where pathname resolution begins. Third, the pseudo filesystems a real system needs — /proc, /sys, /dev — are simply not there inside the new root unless you put them there yourself. chroot relocates the starting point and does nothing else.

What pivot_root does instead

pivot_root(new_root, put_old) operates a level deeper. It does not fiddle with a per-process pathname hint; it changes the root *mount* of the current mount namespace. After the call, new_root is the namespace's / — and the mount that used to be the root is moved, wholesale, to the put_old directory. put_old must itself sit under new_root, so the old world ends up hanging off a subdirectory of the new one, where you can see it just long enough to detach it.

And detach it you do. Once the old root has been relocated to, say, /oldroot, you unmount /oldroot. With that single umount the host's entire filesystem tree — every mount that came before — vanishes from the namespace. There is no leaked descriptor to climb back through, because in a private mount namespace there is no longer any mount that leads to the host at all. That is the difference in one sentence: chroot hides the host behind a new starting point that a clever process can step around; pivot_root removes the host from the mount table entirely, so there is nothing left to step around.

chroot vs pivot_root, in one line each:

chroot: new start for path lookup; old mounts remain, reachable, escapable. pivot_root: new root mount; old root moved aside and then unmounted out of existence.

The exact recipe

This is the part everyone gets subtly wrong, so here is the precise sequence, with the reason for each step. Assume rootfs is the merged overlay directory from Chapter 11 and that you are already inside a fresh mount namespace (unshare --mount).

```bash
# 0. inside a new mount namespace already
# 1. stop our mount changes leaking back to the host
mount --make-rprivate /

# 2. new_root MUST be a mount point: bind it on itself
mount --bind /path/rootfs /path/rootfs

# 3. a place to park the old root, under new_root
mkdir -p /path/rootfs/oldroot

# 4. pivot
cd /path/rootfs
pivot_root . oldroot

# 5. make the new root our cwd's frame of reference
cd /

# 6. supply the pseudo-filesystems inside the new world
mount -t proc proc /proc
mount -t sysfs sys /sys
mount -t tmpfs tmpfs /dev    # or a devtmpfs

# 7. detach the old root, then remove its mountpoint
umount -l /oldroot
rmdir /oldroot
```

The canonical pivot_root dance, step by step.

Step 1, mount --make-rprivate /, is the one people omit and then wonder why their host's mount table fills with junk. A mount namespace starts as a copy of the host's, and by default mounts propagate between them along shared subtrees. Making the whole tree rprivate severs that propagation, so the bind mount, the pivot, and the later umounts stay strictly inside the namespace. Without it, your tidy container surgery leaks out onto the machine you are running on.

Step 2, the bind-mount-on-itself, looks like a no-op and is anything but. pivot_root flatly refuses to operate unless new_root is itself a mount point — not merely a directory. An overlay merged directory usually *is* a mount (overlayfs mounts onto it), but a plain directory of files is not, and the kernel returns EINVAL. Bind mounting the directory onto itself turns it into a mount point without changing a single file inside it, which is exactly what the syscall demands. Memorize this trick; it is the single most common reason a hand-rolled pivot_root fails.

Steps 4 and 5 are a pair. pivot_root . oldroot uses the current directory as new_root and rootfs/oldroot as put_old, so after the call . refers to the new root and oldroot holds the old one. But your shell's idea of "." and the kernel's root can briefly disagree, so cd / immediately afterward re-anchors you cleanly at the new root. Step 6 then mounts fresh /proc, /sys, and /dev inside the new world — recall from Chapter 7 that /proc in particular must be mounted *after* you are in the new PID namespace (unshare's --mount-proc does exactly this) so that ps and /proc/self reflect the container's process tree, not the host's.

Step 7 is the payoff. umount -l /oldroot lazily detaches the old root — lazy because processes may still hold references to it for an instant — and with that the host tree is gone. rmdir tidies the empty mountpoint. Run ls / now and you see only the container's own rootfs; cat /proc/mounts shows a mount table describing a world that contains no host.

How the pieces lock together

Four chapters, one filesystem. overlayfs (Chapter 11) assembles the image's layers into one merged directory — the container's rootfs. The mount namespace (Chapter 7) gives that container a private mount table, so the rearrangement you are about to do is invisible to the host and the host's mounts can be discarded without consequence. pivot_root makes the merged directory the namespace's actual /. And unmounting the old root completes the enclosure, leaving the process with no path, no mount, and no descriptor that leads back to the machine it is running on. That is the full filesystem restraint, and you have now seen every part of it.

This is not a toy approximation of what real runtimes do — it is what they do. runc, the low-level runtime under Docker, uses pivot_root to enter a container's root filesystem, falling back to chroot or an MS_MOVE-based maneuver only in narrow edge cases, such as running inside an initramfs where pivot_root is not permitted. So "what Docker does to the filesystem" is, almost exactly, the recipe above with the overlay merged directory passed as new_root.

Origins:

pivot_root entered Linux around 2.3.41 in early 2000, and it was not built for containers at all — containers did not exist yet. It was built for boot: the kernel mounts a small temporary root (an initrd, later an initramfs) to find drivers, then must switch to the real disk-backed root filesystem without rebooting. pivot_root was the clean way to swap one mounted root for another in a running system. Container runtimes simply discovered, years later, that the boot-time tool for changing / was exactly the tool they needed too. chroot, by contrast, dates to 1979 (Ch. 5) — twenty-one years older, and a layer shallower.

Try it yourself

Let us build a real one. The cleanest rootfs to grab is Alpine's minirootfs tarball — a few megabytes that unpack into a working userland. Download and extract it into a directory:

BASH

```
mkdir -p rootfs
URL=https://dl-cdn.alpinelinux.org/alpine/v3.20
wget -qO alpine.tar.gz \
  $URL/releases/x86_64/alpine-minirootfs-3.20.0-x86_64.tar.gz
tar -xzf alpine.tar.gz -C rootfs
ls rootfs
```

A complete tiny userland in a directory.

OUTPUT

```
bin  dev  etc  home  lib  proc  root  sbin
sys  tmp  usr  var
```

Now enter a fresh mount and PID namespace and perform the pivot. The --pid --fork makes the shell PID 1 of a new process tree; running the dance by hand lets you see each step (use sudo unless you have set up a user namespace as in Chapter 9):

BASH

```
sudo unshare --mount --pid --fork bash
# --- now inside the new namespaces ---
R=$PWD/rootfs
mount --make-rprivate /
mount --bind $R $R
mkdir -p $R/oldroot
cd $R
pivot_root . oldroot
cd /
mount -t proc proc /proc
umount -l /oldroot && rmdir /oldroot
```

The whole pivot, performed inside two new namespaces.

You are now standing in the container's root. Look around and count the processes — the host has disappeared from both the filesystem and the process table:

BASH

```
ls /
cat /etc/os-release | head -1
ps -ef
```

Confirm the new world: only the rootfs, only our processes.

OUTPUT

```
bin  dev  etc  home  lib  proc  root  sbin
sys  tmp  usr  var
NAME="Alpine Linux"
PID   USER     TIME  COMMAND
    1 root      0:00 bash
    7 root      0:00 ps -ef
```

ls / shows the Alpine rootfs and nothing of the host. /etc/os-release says Alpine, even though your machine may be Ubuntu. ps reports bash as PID 1 with only itself and your ps for company — proof that the fresh /proc mount reflects the container's PID namespace, not the host's. Try cd /oldroot: there is no such directory, because you unmounted the host out of existence. You have, with a handful of commands, assembled the exact filesystem isolation a real container runs inside.

Everything is now in place. The overlay builds the root, the mount namespace isolates it, pivot_root installs it as /, and the host is gone. In the next chapter you will gather this and every other primitive in the book and assemble a working container by hand, start to finish, with no Docker anywhere in sight.

PART IV

Building a Container by Hand

The payoff: assemble a working container with nothing but shell commands, then see what an image actually is — layers, a manifest, and a registry protocol.

PART IV

4.1 Building a Container by Hand

A real container with no Docker, step by step

You have met every restraint separately. The PID namespace gave a process its own view of the process table (Chapters 7-9). cgroups capped what it could consume (Chapter 10). overlayfs assembled an image into a single root directory (Chapter 11). Capabilities and seccomp narrowed the kernel interface it could touch (Chapter 12). pivot_root swapped that directory in as the real / and unmounted the host out of existence (Chapter 13). Six restraints, six chapters, each one demonstrated in isolation.

Now you combine them. In this chapter you build a working container with nothing but shell commands and the running kernel — no Docker, no runc, no container library of any kind. By the end you will have a process that sees only its own PIDs, owns its own hostname, network, and mount table, believes an Alpine filesystem is /, is capped at half a CPU and 100 MB of RAM, can reach the internet, and runs with a trimmed set of capabilities. That is a container. Everything `docker run` does, you are about to do by hand.

Open two terminals. One is the container's shell; the other is the host, where you reach in to wire up networking and limits from outside. Run the host-side commands with sudo (or as root). We will mark which terminal each block belongs to.

Step 0 - get a root filesystem

A container needs a userland to be /. The smallest convenient one is Alpine's minirootfs: a few megabytes that unpack into a complete filesystem. Download and extract it on the host:

```bash
# [host]
mkdir -p ~/cbh && cd ~/cbh
V=3.20.0
U=https://dl-cdn.alpinelinux.org/alpine/v3.20
F=alpine-minirootfs-$V-x86_64.tar.gz
wget -qO alpine.tar.gz $U/releases/x86_64/$F
mkdir -p rootfs
tar -xzf alpine.tar.gz -C rootfs
ls rootfs
```

A complete tiny userland, unpacked into rootfs/.

```
bin  dev  etc  home  lib  proc  root  sbin
sys  tmp  usr  var
```

That plain directory is enough to run a container, and we use it as the main path to keep the moving parts visible. But it is not quite what Docker does. Docker never mutates the image itself; it stacks a thin writable layer over the read-only image with overlayfs (Chapter 11) so the container's changes are copy-on-write. To do the same, treat rootfs as the lower layer and build a merged view:

```bash
# [host] optional: the real Docker way, copy-on-write
cd ~/cbh
mkdir -p upper work merged
sudo mount -t overlay overlay \
  -o lowerdir=rootfs,upperdir=upper,workdir=work \
  merged
# now use ~/cbh/merged as the rootfs below
```

lowerdir is the image; writes land in upper/, COW.

Either path works. The walkthrough below uses the plain rootfs directory; if you mounted the overlay, substitute merged for rootfs everywhere and you have the genuine Docker layering.

Step 1 - new namespaces

One command creates the isolation. unshare detaches the new shell from each of the host's namespaces in turn:

```bash
# [container terminal]
cd ~/cbh
sudo unshare \
  --pid --mount --uts --ipc --net \
  --fork --mount-proc \
  bash
# --- you are now inside the container ---
```

Each flag is one chapter's namespace, made fresh.

Each flag, mapped to its chapter:

--pid new PID namespace; this bash becomes PID 1 (ch.7) --mount private mount table for the pivot ahead (ch.7) --uts own hostname / domainname (ch.7) --ipc isolated SysV IPC and POSIX message queues (ch.7) --net empty network namespace; wired up in step 4 (ch.8) --fork fork so the child, not unshare, is PID 1 (ch.7) --mount-proc mount a fresh /proc for the new PID ns (ch.7)

--net is the one to watch: it hands you an isolated network namespace with only a down

loopback interface and no route off the machine. The container is born offline. We connect it deliberately in step 4. For a rootless container, add --user --map-root-user, which maps your namespace-root to your unprivileged host user (Chapter 9); the steps below then need no sudo inside, though networking gets fiddlier and is left as a challenge at the end.

Step 2 - hostname

You are in a fresh UTS namespace, so setting the hostname here changes only the container's view (Chapter 7):

```bash
# [container]
hostname container
hostname
```

A name change the host never sees.

```
OUTPUT
container
```

Step 3 - pivot into the rootfs

This is the moment the host disappears. You are already in a private mount namespace, so the pivot_root dance from Chapter 13 stays sealed inside the container. Run it exactly:

```bash
# [container]
R=$PWD/rootfs              # or merged, if overlay
mount --make-rprivate /    # stop leaks to the host
mount --bind $R $R         # new_root must be a mount
mkdir -p $R/oldroot
cd $R
pivot_root . oldroot       # swap / for the rootfs
cd /
```

new_root becomes /; old root parked at /oldroot.

Now supply the pseudo-filesystems the userland expects, then cut the host loose. A minimal /dev is enough — bind a few device nodes or lay down a tmpfs and create them:

```bash
# [container]
mount -t proc proc /proc
mount -t sysfs sys /sys
mount -t tmpfs tmpfs /dev
mknod -m 666 /dev/null    c 1 3
mknod -m 666 /dev/zero    c 1 5
mknod -m 666 /dev/random  c 1 8
mknod -m 666 /dev/urandom c 1 9
umount -l /oldroot && rmdir /oldroot
```

Fresh /proc, /sys, /dev; then the host is gone.

Look around. The host has vanished from both the filesystem and the process table, and the userland identifies as Alpine:

```bash
# [container]
ls /
cat /etc/os-release | head -1
ps -ef
```

Only the rootfs, only our processes, Alpine as /.

```
OUTPUT
bin  dev  etc  home  lib  oldroot  proc  root
sbin  sys  tmp  usr  var
NAME="Alpine Linux"
PID   USER     TIME  COMMAND
    1 root      0:00 bash
    9 root      0:00 ps -ef
```

There it is — the wow moment. ps shows your bash as PID 1 with nothing but itself and ps for company, because the fresh /proc reflects the container's PID namespace. /etc/os-release says Alpine even if your machine runs Ubuntu. cd /oldroot fails: you unmounted the host out of existence. This is genuine container isolation, built from three commands and a syscall.

Step 4 - networking

The container is sealed but offline. To give it the internet you reach in from the host with a veth pair: a virtual cable with two ends, one left on the host and one pushed into the container's network namespace. First, from the host, find the container's PID — the unshared bash is the entry point of that namespace:

BASH
```
# [host]
CPID=$(pgrep -f 'unshare .*bash' | head -1)
# the namespace's init is PID 1 inside; CPID is its
# host-side PID, which addresses the net namespace
echo $CPID
```

CPID names the container's network namespace by PID.

Create the pair, move one end inside, and address both ends. The container side gets 10.10.0.2; the host side 10.10.0.1 acts as its gateway:

BASH
```
# [host]
ip link add vethH type veth peer name vethC
ip link set vethC netns $CPID
ip addr add 10.10.0.1/24 dev vethH
ip link set vethH up
# bring up the container end inside its namespace
nsenter -t $CPID -n ip addr add 10.10.0.2/24 dev vethC
nsenter -t $CPID -n ip link set vethC up
nsenter -t $CPID -n ip link set lo up
nsenter -t $CPID -n ip route add default via 10.10.0.1
```

veth pair, addresses, and a default route inside.

The container can now reach the host. For the internet beyond, turn the host into a router: enable forwarding and NAT the container's subnet out through the host's uplink (eth0 here):

BASH
```
# [host]
sysctl -w net.ipv4.ip_forward=1
iptables -t nat -A POSTROUTING \
  -s 10.10.0.0/24 -o eth0 -j MASQUERADE
iptables -A FORWARD -i eth0 -o vethH -j ACCEPT
iptables -A FORWARD -i vethH -o eth0 -j ACCEPT
```

Forwarding plus MASQUERADE = NAT to the internet.

Back inside the container, prove it. Set a resolver, then ping and fetch:

BASH
```
# [container]
echo 'nameserver 1.1.1.1' > /etc/resolv.conf
ping -c 2 1.1.1.1
wget -qO- http://example.com | head -1
```

A container on the network, reaching the world.

```
OUTPUT
PING 1.1.1.1 (1.1.1.1): 56 data bytes
64 bytes from 1.1.1.1: seq=0 ttl=57 time=11.4 ms
64 bytes from 1.1.1.1: seq=1 ttl=57 time=11.2 ms
<!doctype html>
```

Step 5 - resource limits

An unbounded container can starve the host, so cap it with a cgroup v2 group (Chapter 10), set from the host. Half a CPU and 100 MB of memory, then move the container's bash into the group so it and every child it forks are accounted together:

```bash
# [host]
CG=/sys/fs/cgroup/cbh
mkdir -p $CG
echo '50000 100000' > $CG/cpu.max    # 50ms per 100ms
echo 104857600      > $CG/memory.max # 100 MB
echo $CPID          > $CG/cgroup.procs
```

cpu.max caps to 50%, memory.max to 100 MB.

Test the ceilings from inside. A busy loop never exceeds 50% of a core; a process that grabs more than 100 MB is killed by the OOM killer rather than swallowing host RAM:

```bash
# [container]
yes > /dev/null &         # spin one core
top -bn1 | grep yes       # watch it pinned near 50%
# now provoke the memory cap:
tail /dev/zero            # allocate without bound
```

CPU stays at 50%; the mem hog gets OOM-killed.

```
OUTPUT
  PID USER  %CPU  COMMAND
   42 root  49.7  yes
Killed
```

The yes loop tops out near 50% no matter how hard it spins, and tail /dev/zero is killed the instant its allocations cross 100 MB. The container can no longer hurt the machine.

Step 6 - drop capabilities and syscalls

The final restraint shrinks the kernel interface (Chapter 12). Even as namespace-root, the container should not keep dangerous powers like loading kernel modules or rebooting the host. Use capsh to drop capabilities and set no_new_privs before exec'ing the real workload, so nothing it launches can regain them:

```bash
# [container] run the workload with trimmed powers
capsh --drop=cap_sys_module,cap_sys_admin,cap_sys_boot \
  --secbits=0x2f \
  -- -c 'id; cat /proc/self/status | grep CapEff'
```

no_new_privs + dropped caps for the final process.

Full syscall filtering goes one level deeper. seccomp installs a BPF program that the kernel checks on every syscall, blocking the ones a container has no business making — Docker ships a default profile of roughly 40 denied calls. Installing it by hand needs a few lines of C calling prctl(PR_SET_SECCOMP) (or libseccomp), so we keep this step light and point back to Chapter 12 for the mechanics. The principle stands: capabilities remove privileges; seccomp removes whole syscalls. Docker applies both for you.

What you just built

A container, with nothing left out:

container = unshare (namespaces, ch.7-9) + cgroup write (limits, ch.10) + pivot_root over an overlay (rootfs, ch.11+13) + veth/NAT (network, ch.8) + cap-drop / seccomp (syscalls, ch.12) Nothing more. No magic, no daemon, no proprietary kernel.

Read that box again and notice what is absent: there is no container engine in it. Every line is a stock Linux feature, most of them older than Docker by a decade. You have now built, by hand, what `docker run` builds. Part V shows Docker doing exactly this — the same syscalls, the same cgroup files, the same pivot_root — wrapped in a daemon, an image format, and a command-line that makes it feel like one atomic act. The trick is fully demystified.

Below is the whole core walkthrough as one runnable script, kept to the parts that live inside a single namespace (steps 0-3, 6); the cross-terminal wiring of steps 4 and 5 stays manual because it is driven from the host by PID. Run host setup first, then this from the container shell after unshare:

BASH

```
#!/bin/sh
# run inside: sudo unshare --pid --mount --uts \
#   --ipc --net --fork --mount-proc bash
set -e
hostname container
R=$PWD/rootfs
mount --make-rprivate /
mount --bind "$R" "$R"
mkdir -p "$R/oldroot"
cd "$R"
pivot_root . oldroot
cd /
mount -t proc proc /proc
mount -t sysfs sys /sys
mount -t tmpfs tmpfs /dev
mknod -m 666 /dev/null c 1 3
mknod -m 666 /dev/urandom c 1 9
umount -l /oldroot && rmdir /oldroot
echo 'nameserver 1.1.1.1' > /etc/resolv.conf
echo 'container ready:'; cat /etc/os-release | head -1
exec capsh --drop=cap_sys_module,cap_sys_boot \
  --secbits=0x2f -- -c 'exec /bin/sh'
```

The single-namespace half of the build, end to end.

Try it yourself

You have the working core. Now push it three ways. First, swap the plain rootfs for the overlay from step 0 so the container is copy-on-write: mount the overlay on the host, pass merged as R, and confirm that files you create inside appear in upper/ while rootfs/ stays pristine — that is a Docker image layer in miniature.

Second, make it rootless. Add --user --map-root-user to unshare and run the whole build with no sudo, mapping your namespace-root to your unprivileged host UID (Chapter 9). You will hit walls — mknod is restricted, networking needs slirp4netns or a setuid helper — and working around them teaches you exactly which privileges Docker's rootless mode papers over.

Third, measure it. Time the whole build (date before unshare, date after the rootfs is ready) and compare it to `time docker run --rm alpine true`. You will find your hand-rolled container starts in milliseconds, because it is the same handful of syscalls — Docker's extra time is image pulls, the daemon round-trip, and bookkeeping, not the isolation itself. The isolation, you now know cold, is free.

PART IV

4.2 What an Image Actually Is

Layers, digests, manifests, and the registry

You have been saying "image" for fourteen chapters. It is time to open one and see that the word names something far more modest than it sounds. A Docker image is not a disk image. It is not a virtual machine snapshot, not a single packed binary, not a frozen RAM dump. It does not boot. There is no kernel inside it. An image is two boring things glued together: a set of layers — each a gzipped tar of filesystem changes, the very diffs that became your overlay lowerdirs in chapter 11 — and a couple of small JSON documents that say which layers, in what order, with what defaults. That is the whole format. You can crack one open with `tar` and `jq` and read every byte of it yourself, which is exactly what this chapter does.

Everything is identified by its content hash

Before opening anything, learn the one idea the format is built on: content addressing. Every piece of an image — each layer blob, the config document, the manifest itself — is identified not by a filename or a version number but by the sha256 hash of its own bytes. The hash is the name. Change one byte of a layer and its digest changes completely, which means a digest is an immutable, verifiable claim about exact contents. When a registry hands you a blob it claims is `sha256:9b2a...`, you recompute the hash and check; if it matches, the bytes are provably the ones that were named.

Two consequences fall straight out. First, deduplication: if two images share a base layer, that layer hashes to the same digest in both, so it is stored once on disk and in the registry, downloaded once, and mounted into both. The layer sharing you saw at the overlay level in chapter 11 is enforced by content addressing here. Second, caching: a build step whose inputs are unchanged produces a layer with an unchanged digest, so Docker reuses the cached blob instead of rebuilding it.

This is why a digest is not the same thing as a tag. A tag — `alpine:3.19`, `myapp:latest` — is a mutable human-friendly name that points at a digest, the way a Git branch points at a commit. The maintainer can repoint `alpine:3.19` at a freshly rebuilt image tomorrow; the tag is the same, the digest underneath is different. A digest never moves. So for anything that must be reproducible — a production deploy, a base image in a Dockerfile, a supply-chain audit — you pin by digest, not by tag:

```bash
# mutable: may point somewhere new tomorrow
FROM alpine:3.19
# immutable: provably these exact bytes, forever
FROM alpine@sha256:c5b1261d...
```

A tag is a movable name; a digest is the content itself.

Crack one open with docker save

Enough theory. `docker save` writes an image out as a plain tar archive in the OCI layout, and from there it is just files. Pull a small image, save it, and unpack it into a directory you can poke at:

```bash
docker pull alpine:3.19
docker save alpine:3.19 -o alpine.tar
mkdir alpine && tar xf alpine.tar -C alpine
ls alpine
```

docker save emits a tar; unpack it like any other.

```
OUTPUT
blobs  index.json  manifest.json  oci-layout
repositories
```

Everything addressable lives under `blobs/sha256/`, each file named for its own digest. `index.json` is the top-level entry point; `oci-layout` just declares the layout version. The whole image is a content-addressed store plus one index that says where to start. Walk it from the top: the index points at a manifest, the manifest points at a config and the layers.

The manifest: a table of contents

The image manifest, media type `application/vnd.oci.image.manifest.v1+json`, is the table of contents for one image on one platform. It is small. It lists exactly two kinds of thing: the single config blob, and the ordered list of layer blobs — each entry giving a media type, a size in bytes, and the all-important digest. Pull the manifest's digest out of the index and read it:

```bash
MAN=$(jq -r '.manifests[0].digest' \
  alpine/index.json | cut -d: -f2)
jq . alpine/blobs/sha256/$MAN
```

index.json -> manifest digest -> the manifest JSON.

```
OUTPUT (abridged)
{
  "schemaVersion": 2,
  "mediaType":
   "application/vnd.oci.image.manifest.v1+json",
  "config": {
    "mediaType":
     "application/vnd.oci.image.config.v1+json",
    "digest": "sha256:c5b1261d...",
    "size": 1471 },
  "layers": [ {
    "mediaType":
     "application/vnd.oci.image.layer.v1.tar+gzip",
    "digest": "sha256:4abcec68...",
    "size": 3623807 } ]
}
```

Read it as the contract it is. The manifest names one config blob and, for Alpine, exactly one layer — Alpine is a single-layer image. A fatter image lists its layers here in order, bottom to top. The manifest contains no files itself; it is pure references. To materialize the image you fetch the blobs it names by their digests and stack the layers in listed order. Note the layer media type ends in `tar+gzip`: the blob is a gzipped tarball, which matters in a moment.

The config: where docker inspect gets its answers

The config blob, media type `application/vnd.oci.image.config.v1+json`, is the metadata document. It holds three things worth knowing cold. First, the target platform — `architecture` and `os`, e.g. amd64 / linux. Second, an ordered list of `rootfs.diff_ids`, one per layer. Third, the runtime defaults under a `config` key: the `Env`, `Cmd`, `Entrypoint`, `WorkingDir`, `User`, `ExposedPorts`, and `Volumes` a container inherits unless you override them. And a `history` array describing how each layer was built. This document is where `docker inspect` reads its answers from.

```bash
BASH
CFG=$(jq -r '.config.digest' \
  alpine/blobs/sha256/$MAN | cut -d: -f2)
jq '{architecture, os, config, rootfs}' \
  alpine/blobs/sha256/$CFG
```

The config holds platform, runtime defaults, and diff_ids.

```
OUTPUT (abridged)
{
  "architecture": "amd64",
  "os": "linux",
  "config": {
    "Env": ["PATH=/usr/local/sbin:/usr/local/bin:..."],
    "Cmd": ["/bin/sh"],
    "WorkingDir": "" },
  "rootfs": {
    "type": "layers",
    "diff_ids": ["sha256:f6f1937c..."] }
}
```

This is the missing half of the Dockerfile picture. Chapter 11 showed that `RUN`, `COPY`, and `ADD` each produce a filesystem layer. The instructions that do not change the filesystem — `ENV`, `CMD`, `ENTRYPOINT`, `LABEL`, `EXPOSE`, `WORKDIR`, `USER`, `VOLUME` — produce no layer at all. They write into this config JSON instead. `CMD ["/bin/sh"]` is just the `Cmd` field you see above; `EXPOSE 8080` is an entry in `ExposedPorts`; `ENV` appends to that `Env` array. That is the entire mechanism: a Dockerfile produces a stack of layer tarballs plus this one settings document.

The layers: tarballs of changes, whiteouts and all

Now the layers themselves. A layer blob is a tar of the filesystem changes that one build step made — the `upper`-style diff from chapter 11, serialized — and, per the `tar+gzip` media type, gzip-compressed. List the contents of Alpine's single layer and you are looking at a minimal root filesystem:

```bash
LAY=$(jq -r '.layers[0].digest' \
  alpine/blobs/sha256/$MAN | cut -d: -f2)
tar tzf alpine/blobs/sha256/$LAY | head
```

A layer is a (gzipped) tar; list it like any tarball.

```
OUTPUT (abridged)
bin/
bin/busybox
etc/
etc/alpine-release
lib/
usr/
...
```

Deletions get encoded too. Recall from chapter 11 that overlayfs marks a deleted file with a whiteout. The image format carries the same idea in the tar: a file deleted by a build step appears in the layer as a zero-length entry named `.wh.<name>` — for example deleting `etc/foo` writes `etc/.wh.foo` into the layer. An unpacker that sees a `.wh.` entry removes

the corresponding name from the layers below it. So a layer's tar records additions and modifications as ordinary files and deletions as these whiteout markers; nothing more is needed to express an arbitrary diff.

diff_id versus digest: why a layer has two hashes

You may have noticed the same layer is named two different ways. The manifest called it `sha256:4abcec68...`; the config's `diff_ids` called it `sha256:f6f1937c...`. Both are correct, and the difference is exact and deliberate:

Two hashes, two jobs:

diff_id = sha256 of the UNcompressed layer tar. Stable identity of the content itself. digest = sha256 of the COMPRESSED (gzipped) blob. Identity of the bytes stored and transferred.

Two hashes exist because a layer has two representations. On the wire and in the registry it is a gzipped blob, and the manifest must name the exact bytes you will download and verify — that is the digest, the hash of the compressed blob. But gzip is not deterministic: compress the same tar with a different gzip version or level and you get different bytes, hence a different digest, even though the actual files are identical. So the content identity used for the overlay rootfs is the diff_id, the hash of the uncompressed tar, which depends only on the files. The runtime stacks layers and compares them by diff_id; the registry transfers and verifies them by digest. Same layer, two hats.

Multi-arch: one tag, many images

How does a single `alpine:3.19` tag run on your amd64 laptop and an arm64 server alike? With an image index, media type `application/vnd.oci.image.index.v1+json` (Docker's older name is a "manifest list"). An index is a manifest of manifests: it lists, per platform, the digest of the per-architecture manifest for that platform. There is no filesystem in an index — just a routing table from `{os, architecture}` to a real manifest digest.

```bash
docker manifest inspect alpine:3.19 | \
  jq '.manifests[] | {platform, digest}'
```

The index maps each platform to its own manifest.

```
OUTPUT (abridged)
{ "platform": { "architecture": "amd64",
    "os": "linux" },
  "digest": "sha256:c5b1261d..." }
{ "platform": { "architecture": "arm64",
    "os": "linux" },
  "digest": "sha256:8a1b27f0..." }
```

When you pull, the client reads the index, finds the entry matching its own platform, and fetches that manifest — and only that one. The amd64 and arm64 layers never touch your machine. One tag, one index, many concrete images underneath.

The registry is just HTTPS GETs

A registry — Docker Hub, GHCR, Amazon ECR, your private Harbor — sounds like a special service, but pulling an image is a handful of HTTP GET requests against the OCI Distribution API (standardized from the Docker Registry HTTP API v2). There are two endpoints that matter. `GET /v2/<name>/manifests/<reference>` returns a manifest or an index, where reference is a tag or a digest. `GET /v2/<name>/blobs/<digest>` returns one blob — the config or a layer — by its digest. That is essentially the entire pull protocol.

The one wrinkle is auth. Most registries first send you to a token endpoint for a short-lived bearer token, even for public images. The dance against Docker Hub looks like this — fetch a token, then ask for the manifest with it:

```bash
REPO=library/alpine
TOK=$(curl -s \
  "https://auth.docker.io/token?service=\
registry.docker.io&scope=repository:$REPO:pull" \
  | jq -r .token)
curl -s \
  -H "Authorization: Bearer $TOK" \
  -H "Accept: application/vnd.oci.image.index.v1+json" \
  "https://registry-1.docker.io/v2/$REPO/\
manifests/3.19" | jq '.mediaType, .manifests[0]'
```

Token dance, then GET the manifest by tag.

```
OUTPUT (abridged)
"application/vnd.oci.image.index.v1+json"
{
  "mediaType":
   "application/vnd.oci.image.manifest.v1+json",
  "digest": "sha256:c5b1261d...",
  "platform": { "architecture": "amd64",
    "os": "linux" }
}
```

The `Accept` header is how you ask for what you want; the server returns an index here because `alpine` is multi-arch. From a manifest digest you would GET the config blob and each layer blob the same way, by digest, verifying each against its name as it arrives. `docker pull` is exactly this loop with a progress bar. Because the protocol is standardized, the same requests work against GHCR and ECR with only the hostname and token endpoint changed.

Image spec versus runtime spec

One last distinction to set up the next chapter. Everything here is the OCI image specification: how an image is packaged, addressed, and distributed. It is not how a container runs. To launch a container, a runtime first unpacks the layers in order into a single directory — a plain root filesystem, the merged overlay of chapter 11 — and then translates the image config into a separate document called `config.json` that obeys the OCI runtime specification. That runtime config describes namespaces, cgroups, capabilities, the command to exec. A rootfs directory plus a runtime `config.json` is called an OCI bundle, and a runtime like runc consumes the bundle and does the `clone`/`pivot_root`/`execve` you have studied since chapter 7. The image config you read above (`Cmd`, `Env`, `WorkingDir`) is the seed for that runtime config. Chapter 16 follows the bundle into runc; the split to remember is image spec = how it's packaged, runtime spec = how it's run.

Origins:

Docker's first image format (v1) chained layers by random IDs and was not content-addressed — layers could not be safely shared or verified by hash. The Registry v2 protocol and the schema 2 manifest (2016) made everything digest- addressed: the model in this chapter. Docker donated that format to the Open Container Initiative, which published the OCI image-spec 1.0 in 2017, and later standardized the registry API as the OCI Distribution Spec (1.0 in 2021).

Try it yourself

First, walk a real image end to end with nothing but `docker save`, `tar`, and `jq` — index to manifest to config and layers, then list a layer's files. Every digest is truncated in the output below; yours will be full 64-hex strings.

```bash
docker pull alpine:3.19
docker save alpine:3.19 -o a.tar
mkdir a && tar xf a.tar -C a
MAN=$(jq -r '.manifests[0].digest' \
  a/index.json | cut -d: -f2)
CFG=$(jq -r '.config.digest' \
  a/blobs/sha256/$MAN | cut -d: -f2)
jq -r '.config.Cmd, .architecture' \
  a/blobs/sha256/$CFG
LAY=$(jq -r '.layers[0].digest' \
  a/blobs/sha256/$MAN | cut -d: -f2)
tar tzf a/blobs/sha256/$LAY | head -3
```

Manifest -> config -> Cmd, arch; then the layer files.

```
OUTPUT (abridged)
["/bin/sh"]
amd64
bin/
bin/busybox
etc/
```

Second, pull a blob straight from the registry by digest with curl: fetch a token, GET the amd64 manifest, then GET its config blob and verify it hashes to the digest you asked for. This is the pull protocol stripped to its bones.

```bash
REPO=library/alpine
TOK=$(curl -s \
  "https://auth.docker.io/token?service=\
registry.docker.io&scope=repository:$REPO:pull" \
  | jq -r .token)
AUTH="Authorization: Bearer $TOK"
AC="Accept: application/vnd.oci.image.manifest.v1+json"
MD=$(curl -s -H "$AUTH" \
  -H "Accept: \
application/vnd.oci.image.index.v1+json" \
  "https://registry-1.docker.io/v2/$REPO/\
manifests/3.19" | jq -r \
  '.manifests[]|select(.platform.architecture==\
"amd64").digest')
CD=$(curl -s -H "$AUTH" -H "$AC" \
  "https://registry-1.docker.io/v2/$REPO/\
manifests/$MD" | jq -r .config.digest)
curl -sL -H "$AUTH" \
  "https://registry-1.docker.io/v2/$REPO/\
blobs/$CD" | sha256sum
echo "$CD"   # compare: should match the sum above
```

Token -> index -> manifest -> blob, verified by hash.

```
OUTPUT (abridged)
c5b1261d... -     (sha256sum of the downloaded blob)
sha256:c5b1261d...    (the digest we requested)
```

The two hashes match, which is the whole point of content addressing: the name you asked for is provably the bytes you received. Third, if you have `skopeo` or `crane` installed, `skopeo inspect --raw docker://alpine:3.19 | jq .mediaType` or `crane manifest alpine:3.19` shows you the index directly, and `crane manifest alpine@<arch-digest>` drills into one platform — the same documents you unpacked from `docker save`, fetched live over the protocol above. An image, all the way down, is layers plus a little JSON.

PART V

Docker Itself

Now that the magic is gone, we follow Docker's real architecture and trace one `docker run` from the CLI down through the kernel calls of the earlier chapters.

PART V

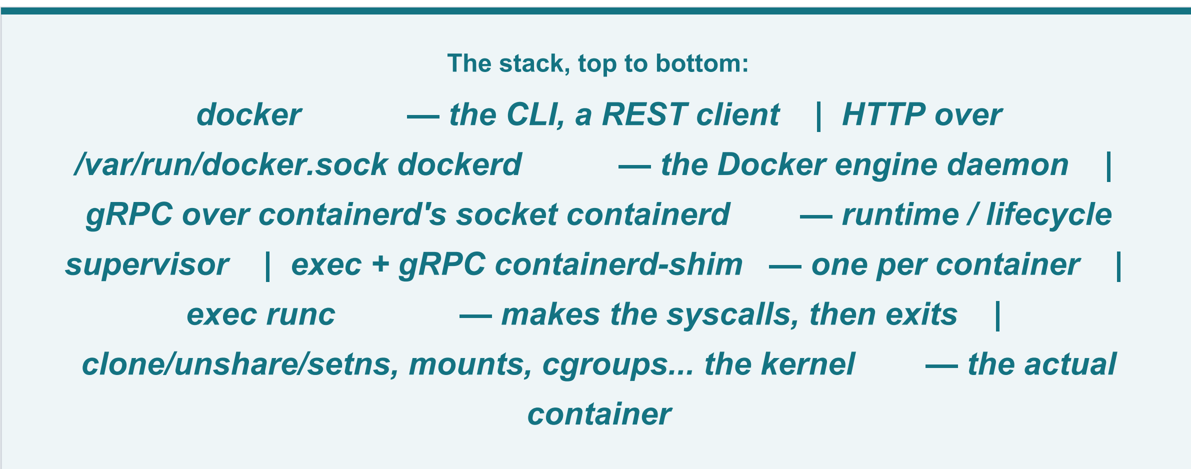
5.1 Docker's Real Architecture

CLI, dockerd, containerd, shim, and runc

Part III took the machine apart restraint by restraint: namespaces for what a process can see, cgroups for how much it can use, capabilities and seccomp for what it may ask the kernel to do, pivot_root and overlayfs for the root it lives in. Each of those was a syscall, or a small family of them, that you could fire by hand. And yet you do not type clone() and setns() and capset() when you start a container. You type one word: docker. There is a tall stack of software between that word and those syscalls, and this chapter is the cross-section. We are going to slice straight down through it.

Start with the reveal, because almost everyone gets it wrong: 'Docker' is not a program. It is not even a program with plugins. It is a STACK of separate processes — a thin client, a high-level daemon, a runtime supervisor, a per-container babysitter, and a tiny one-shot tool that does the actual kernel work — each doing exactly one job and handing off to the next. When you understand which process does what, the whole system stops being magic and becomes a chain you can trace, interrupt, and replace one link at a time.

Here is the chain, top to bottom. The convenience you type lives at the very top, furthest from the kernel; the syscalls of Part III happen at the very bottom. We will walk down it one process at a time.



docker: a thin REST client

The docker command you type is astonishingly dumb, and that is by design. It is a REST client. It does not create containers, pull images, or touch a single namespace. It takes your arguments, turns them into an HTTP request against the Docker Engine API, sends that request to the daemon over a UNIX domain socket — by default /var/run/docker.sock — reads the HTTP response, and prints it. That is the entire job. The socket is the boundary between the client and everything else.

Because it is just HTTP over a socket, you do not even need the docker binary to talk to the daemon. Any tool that can speak HTTP to a UNIX socket will do. curl can:

```bash
$ curl --unix-socket /var/run/docker.sock \
       http://localhost/version
```

Talk to the daemon directly — no docker CLI.

EXAMPLE

```
{
  "Version": "24.0.7",
  "ApiVersion": "1.43",
  "Os": "linux",
  "Arch": "amd64",
  "Components": [
    {"Name": "Engine", ...},
    {"Name": "containerd", ...},
    {"Name": "runc", ...}
  ]
}
```

Notice what the daemon reports about itself: an Engine, a containerd, and a runc. The version endpoint is already telling you the stack exists. The CLI is one HTTP hop away from the work; everything real happens on the far side of that socket.

dockerd: the high-level engine

On the other side of the socket sits dockerd, the Docker daemon — the part of the Moby project that most people mean when they say 'Docker.' It is the high-level engine, and it owns everything that is about convenience and policy rather than about the kernel: it serves the Engine API, authenticates registry pulls, builds images from Dockerfiles, manages networks and volumes, wires up bridges and port forwards, and keeps the catalog of what images and containers exist.

But here is the crucial modern fact: dockerd does NOT create containers anymore. It used to, back when libcontainer lived inside it (ch. 6). Today it delegates. When you ask for a container, dockerd prepares the image and the configuration, then hands the actual lifecycle down to containerd over a separate gRPC socket. dockerd is the manager that decides what should run; it is not the process that runs it.

containerd: the runtime supervisor

containerd is the next layer down — a CNCF project that Docker split out of its daemon and donated in 2017. Think of it as the container supervisor. It owns the container lifecycle (create, start, stop, delete, exec), and it owns the image plumbing: the content store, where image blobs are kept addressed by their digest (ch. 15), and the snapshotters, which assemble those layers into a ready-to-use root filesystem using overlayfs (ch. 11).

containerd is where an image stops being a stack of tarballs and becomes a mounted rootfs.

containerd has a life independent of Docker. It ships its own command-line client, ctr, for poking at it directly, and it exposes a CRI plugin — the Container Runtime Interface — which is the door Kubernetes walks through. dockerd talks to containerd over gRPC; so does Kubernetes; so does ctr. containerd does not itself make the namespace and cgroup syscalls either. It goes one layer lower still — but not directly. It goes through a shim.

containerd-shim: one babysitter per container

For every running container there is exactly one shim process — on a modern system, containerd-shim-runc-v2. This is the single most misunderstood piece of the stack, and the most clever. Ask the obvious question: why is it there at all? Why doesn't containerd just run the container as its own child?

Because then your containers would die every time you upgraded Docker. If containerd were the direct parent of every container, restarting containerd — for an upgrade, a crash, a config reload — would orphan or kill every container on the box. The shim breaks that coupling. The shim, not containerd, is the parent of the container process. It is the one holding the container's standard input, output, and error streams open. It is the one that reaps the container and remembers its exit code. So containerd can stop and restart freely; the shims keep running, the containers keep running, and when containerd comes back up it simply reconnects to the shims that were patiently waiting.

You can see the consequence in the process tree. The shim and its container are NOT children of containerd or dockerd. They have been reparented to PID 1 — init — precisely so that the daemons above them are detachable. Run ps while a container is alive and look at the parent PIDs:

```bash
$ docker run -d --name web nginx
$ ps -eo pid,ppid,comm | grep -E \
    'dockerd|containerd|nginx'
```

PPID is the giveaway: who parents whom.

```
  PID  PPID COMMAND
  812     1 dockerd
  840     1 containerd
 2310     1 containerd-shim
 2331  2310 nginx
 2380  2331 nginx
```

Read the PPID column. dockerd's parent is 1. containerd's parent is 1. The shim's parent is 1 — it is not under containerd. And nginx's parent is the shim (2310), not containerd and

certainly not dockerd. The daemons sit off to the side as managers; the live process hangs off its shim, which hangs off init. That is the restart-survival design made visible.

runc: the tool that actually makes the container

At the very bottom is runc — small, short-lived, and the only piece in the whole stack that touches the kernel features of Part III. It was libcontainer; it became the OCI reference runtime in 2015 (ch. 6). The shim execs runc to bring a container into existence. runc reads its instructions, makes the exact clone()/unshare()/setns() calls to build the namespaces, sets up the mounts and pivot_root, writes the cgroup limits, drops capabilities and installs the seccomp filter, execs the container's entrypoint — and then it EXITS.

That last word matters. runc is not a daemon. It does not stay running alongside the container. It is a one-shot tool: it performs the setup, hands the now-running process back to the shim (which inherited the stdio and remains the parent), and gets out of the way. This is why you never see runc in ps next to a long-running container — by then runc is long gone. The shim stays; runc fires once and quits.

The bundle: rootfs + config.json

How does runc know what namespaces to create, which mounts to set up, what cgroup limits to apply, which capabilities to keep, what process to exec? It is told, by a single directory called an OCI runtime BUNDLE. A bundle is exactly two things and nothing more:

An OCI runtime bundle is:

rootfs/ — the unpacked image layers, the filesystem the container will see config.json — the OCI runtime-spec: namespaces, mounts, cgroup limits, caps, seccomp, and the process to run

This is the precise place where the two OCI specs meet, and the distinction is worth stating plainly. The image you pulled (ch. 15) follows the OCI IMAGE spec: layers, a config, a manifest, all content-addressed. But runc does not consume images. containerd converts the image into a runtime bundle: it unpacks the layers into rootfs/ via the snapshotter, and it translates the image into a config.json that follows the OCI RUNTIME spec. Image-spec in, runtime-spec out. The config.json is the runtime-spec made concrete for this one container.

And config.json is, quite literally, the four restraints of Part III written down in machine-readable form. Here is a trimmed but real-shaped excerpt:

```bash
$ cat config.json   # trimmed
{
  "process": {
    "args": ["/bin/sh", "-c", "nginx -g..."],
    "capabilities": {
      "bounding": ["CAP_CHOWN",
                   "CAP_NET_BIND_SERVICE"]
    }
  },
  "linux": {
    "namespaces": [
      {"type": "pid"},
      {"type": "mount"},
      {"type": "network"},
      {"type": "uts"},
      {"type": "ipc"}
    ],
    "resources": {
      "memory": {"limit": 536870912},
      "cpu": {"quota": 50000,
              "period": 100000}
    },
    "seccomp": {"defaultAction":
                "SCMP_ACT_ERRNO"}
  }
}
```

The four restraints, declared as data.

Map it line by line to Part III. process.args is the entrypoint runc will exec. linux.namespaces is the list of private views to create — pid, mount, network, uts, ipc — each one a clone() flag (ch. 7-9). linux.resources is the cgroup limits: half a gigabyte of memory, fifty percent of one CPU (ch. 10). process.capabilities is what survives the capability drop (ch. 12). seccomp is the syscall filter (ch. 12). 'Limit what it sees, limit what it uses, limit what it may do, give it its own root' — that sentence is this JSON file. runc's whole job is to make config.json true.

Seeing the layers, live

You do not have to take the stack on faith. The running system reports every layer. docker info names the runtime and the containerd version underneath it:

```bash
$ docker info | grep -iE \
    'runtime|containerd|runc'
```

The engine confesses its own internals.

EXAMPLE

```
 Runtimes: io.containerd.runc.v2 runc
 Default Runtime: runc
 containerd version: 3dce8eb...
 runc version: v1.1.9-0-gccaecfc
```

Go one layer down and ask containerd directly, with its own client ctr — bypassing dockerd entirely. The container dockerd created is visible to containerd because dockerd created it THROUGH containerd:

BASH

```bash
$ ctr --namespace moby containers list
```

containerd's own view, no Docker involved.

EXAMPLE

```
CONTAINER        IMAGE    RUNTIME
9f2a...c1        -        io.containerd.runc.v2
```

And one layer below that, runc itself keeps state. Point it at containerd's state directory and it will list and describe the very same container — proof that the bottom of the stack is a real, independent tool, not an internal function of containerd:

BASH

```bash
$ runc --root \
  /run/containerd/runc/moby list
$ runc --root \
  /run/containerd/runc/moby state 9f2a...c1
```

runc knows the container by name.

EXAMPLE

```
ID          PID    STATUS   BUNDLE
9f2a...c1   2331   running  /run/.../9f2a...c1

{ "id": "9f2a...c1", "status": "running",
  "pid": 2331, "bundle": "/run/.../9f2a..." }
```

Origins:

libcontainer (2014) — Docker's native Go library, cut free of LXC. runc (2015) — libcontainer donated to the OCI as the reference runtime. containerd (2017) — split out of dockerd, donated to the CNCF. Moby (2017) — the upstream open-source project that Docker CE is assembled from.

Why split it into so many pieces?

Five processes to run one nginx looks like over-engineering until you see what the seams buy you. Separation of concerns is the obvious win: the CLI knows HTTP, dockerd knows builds and networks, containerd knows lifecycle and images, runc knows syscalls. But the real prize is the OCI standard at the bottom. Because runc consumes a standard bundle, you can swap it out. Want stronger isolation? Drop in gVisor's runsc, which intercepts syscalls in userspace, or Kata Containers, which wraps each container in a lightweight VM. Want a smaller, C-based runtime? Use crun. containerd does not care; it speaks runtime-spec to whatever sits below.

And because containerd exposes the CRI, Kubernetes can talk to it directly and skip dockerd entirely — it never needed the build-and-network engine. That is why Kubernetes removed its Docker shim ('dockershim') in version 1.24 in 2022: it had been routing through dockerd to reach containerd, and once everyone was on containerd anyway, the detour was pure overhead. The layering you just traced is exactly what made that removal painless.

Try it yourself

First, see the whole stack as one process tree while a container runs — dockerd and containerd off to the side, the shim parented to init, the app under the shim:

```bash
$ docker run -d --name demo nginx
$ pstree -ps $(pgrep -n nginx)
$ docker info | grep -i 'runtime'
```

Walk the tree from nginx back up to init.

```
systemd(1)---containerd-shim(2310)---
  nginx(2331)---nginx(2380)
 Default Runtime: runc
```

Note that pstree walks UP from nginx and hits the shim, then init — it never passes

through containerd or dockerd, because they are not in this process's ancestry. Now prove the bottom of the stack works alone. Export an image's filesystem into a rootfs/, let runc generate a default config.json with runc spec, and run a container with PURE runc — no dockerd, no containerd, no shim:

```bash
$ mkdir -p bundle/rootfs && cd bundle
$ docker export $(docker create alpine) \
    | tar -C rootfs -xf -
$ runc spec
$ sudo runc run demo
```

A bundle = rootfs + config.json. runc runs it.

```
/ # cat /etc/os-release | head -1
NAME="Alpine Linux"
/ # ps -ef
PID   USER   COMMAND
1     root   sh
/ # exit
```

Sit with that last result. You exported the image with Docker, but you ran the container with nothing but runc and a bundle — and inside it, your shell is PID 1, in its own pid namespace, on its own rootfs. No daemon was involved. That is the floor of the entire stack, working in isolation. Everything above it — containerd, dockerd, the CLI — is convenience, lifecycle, and policy layered on top of this one small tool that turns a config.json into the syscalls of Part III. In ch. 19 we will start at the top with a single docker run and trace the call all the way down this chain until those syscalls fire.

PART V

5.2 Docker Networking, Demystified

docker0, veth, iptables, and port publishing

In Chapter 8 you built a container network from nothing. You made an empty network namespace, ran a veth cable into it, enslaved the host end to a Linux bridge, addressed the bridge as a gateway, flipped on IP forwarding, and added a single MASQUERADE rule so the namespace could reach the internet. Ten or so `ip` and `iptables` commands, and you had a working, internet-connected container network. This chapter has one job: to walk up to a *real* running Docker host, point at each piece of its networking, and watch you recognize every one. There is nothing here you have not already done by hand.

Docker's networking is pluggable, organized around *drivers*. The default is `bridge` — the docker0 model you already built. `host` puts the container in the host's own network namespace, no isolation at all. `none` gives it an empty namespace and nothing else. `overlay` stitches containers across multiple hosts with VXLAN tunnels, for Swarm and orchestrators. `macvlan` and `ipvlan` hand a container a presence directly on the physical LAN, each with its own MAC or IP. We will spend almost all our time on `bridge`, because it is the default, it is what most people mean by "Docker networking," and it is the one you have already reinvented.

Meet docker0, the bridge you already built

When the Docker daemon starts for the first time, it creates a Linux bridge named `docker0` and gives it a private subnet — 172.17.0.0/16 by default — with the bridge itself holding the first address, 172.17.0.1, which becomes the gateway every container on that network shares. This is the exact `br0` you made in Chapter 8, only named docker0 and created by a daemon instead of by you. Look at it directly:

```bash
ip addr show docker0
```

The default Docker bridge, as a plain kernel interface.

```
3: docker0: <BROADCAST,MULTICAST,UP> mtu 1500 ...
    link/ether 02:42:9a:1b:2c:3d brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 \
          scope global docker0
```

There it is: a bridge, up, carrying 172.17.0.1/16. Docker's own view of the same thing is the network list. The three default networks — `bridge`, `host`, and `none` — correspond

one-to-one to the docker0 bridge, the host namespace, and the empty namespace.

```bash
docker network ls
```

Docker's three built-in networks, one per default driver.

```
NETWORK ID     NAME     DRIVER   SCOPE
a1b2c3d4e5f6   bridge   bridge   local
f6e5d4c3b2a1   host     host     local
0011223344ff   none     null     local
```

And `docker network inspect bridge` shows the subnet and gateway Docker assigned — the same numbers the kernel reported, now from Docker's bookkeeping:

```bash
docker network inspect bridge \
  -f '{{range .IPAM.Config}}{{.Subnet}} {{.Gateway}}{{end}}'
```

Pull just the subnet and gateway out of the inspect JSON.

```
172.17.0.0/16 172.17.0.1
```

One container, one veth, one bridge port

Start a container and Docker performs your Chapter 8 ritual. It creates a veth pair; it moves one end into the container's network namespace and renames it `eth0` — the same `ip link set ... netns` move you made by hand, plus a rename; and it enslaves the other end to docker0 with `ip link set <veth> master docker0`. The container gets the next free address from the subnet and a default route via 172.17.0.1. Run one and look:

```bash
docker run -d --name web nginx
ip link
```

Start a container, then list the host's interfaces.

```
3: docker0: <...,UP> mtu 1500 ...
5: vetha1b2c3@if4: <...,UP> mtu 1500 \
     master docker0 state UP
```

The `vetha1b2c3` interface is the host end of the cable — the cryptic name is just Docker

avoiding collisions. The `@if4` suffix is the kernel telling you its peer is interface index 4, which lives in the container's namespace. `master docker0` is the enslavement: this veth is a port on the switch. The bridge's own view confirms the membership:

```bash
bridge link
```

List the interfaces enslaved to bridges on this host.

```
OUTPUT
5: vetha1b2c3@if4: <...,UP> ... master docker0 \
     state forwarding
```

Notice what is *missing* from `ip netns list`: the container. Docker creates anonymous network namespaces and, unlike `ip netns`, does not bind-mount them under /var/run/netns/, so the friendly helper cannot see them. They are still real — you just reach them through the container's process instead of a name. Find the PID, then use `nsenter -t <pid> -n` to run a command in that namespace:

```bash
pid=$(docker inspect -f '{{.State.Pid}}' web)
sudo nsenter -t "$pid" -n ip addr show eth0
```

Enter the container's netns by PID and read its eth0.

```
OUTPUT
4: eth0@if5: <BROADCAST,MULTICAST,UP> mtu 1500 ...
    inet 172.17.0.2/16 brd 172.17.255.255 \
          scope global eth0
```

Interface index 4, named eth0, addressed 172.17.0.2/16 — and its `@if5` points right back at the host's veth index 5. The two ends of one cable, exactly as in Chapter 8. Inside that namespace, `ip route` would show a default route via 172.17.0.1, the docker0 gateway. Nothing new under the sun.

Out to the internet: forwarding and MASQUERADE

The container can reach the host and its siblings on docker0, but 172.17.0.0/16 is private — the internet has never heard of it. For outbound traffic to work, the daemon does the same two things you did: it sets `net.ipv4.ip_forward=1` so the host routes between interfaces, and it installs a source-NAT (MASQUERADE) rule for the bridge subnet. Read the NAT table's POSTROUTING chain and there is your rule:

```bash
sudo iptables -t nat -L POSTROUTING -n
```

The source-NAT rules applied as packets leave the host.

```
OUTPUT
Chain POSTROUTING (policy ACCEPT)
target      prot source        destination
MASQUERADE  all  172.17.0.0/16 0.0.0.0/0
```

That single line is the rule you wrote by hand in Chapter 8, verbatim except for the subnet. As packets from 172.17.0.0/16 leave for the wider world, their source address is rewritten to the host's, and replies are translated back. Docker also adds a set of custom chains — `DOCKER`, `DOCKER-USER`, `DOCKER-ISOLATION-STAGE-1` and friends — that organize forwarding and per-network isolation. `DOCKER-USER` is the one to know: it runs before Docker's own rules, so it is where you put your own filtering without the daemon clobbering it.

Port publishing is just DNAT

The reverse direction — letting the outside *reach* a container — is `-p`, and it is destination NAT, the mirror image of MASQUERADE. When you run a container with `-p 8080:80`, the daemon does two things. First, it inserts a DNAT rule into the nat table's DOCKER chain that rewrites traffic arriving on the host's port 8080 to the container's port 80. Second, it starts a small userland helper, `docker-proxy`, that listens on host port 8080 as a fallback — it handles cases the iptables fast path misses, such as a container reaching its own published port (hairpin). The DNAT rule is the one that carries real traffic. Publish a port and read the DOCKER chain:

```bash
docker run -d -p 8080:80 --name web2 nginx
sudo iptables -t nat -L DOCKER -n
```

Publish host:8080 to container:80, then read the rule.

```
OUTPUT
Chain DOCKER (2 references)
target  prot source     destination
RETURN  all  0.0.0.0/0  0.0.0.0/0
DNAT    tcp  0.0.0.0/0  0.0.0.0/0 \
        tcp dpt:8080 to:172.17.0.3:80
```

Read that last line plainly: any TCP packet whose destination port (`dpt`) is 8080 has its destination rewritten (`to:`) to 172.17.0.3:80 — the container's own address and port. Trace an inbound connection step by step. A client opens a TCP connection to

your_host:8080. The packet hits the host's NAT PREROUTING, jumps to the DOCKER chain, matches the DNAT rule, and has its destination rewritten to 172.17.0.3:80. With forwarding enabled, the kernel routes the now-rewritten packet out through docker0, across the veth, into the container's eth0, where nginx is listening on port 80. The reply retraces the path; conntrack reverses the translation so the client sees a reply from your_host:8080, never suspecting a rewrite happened. That is the whole of port publishing.

Names between containers: user-defined networks

The default `bridge` network has one notable gap: no name resolution. Two containers on docker0 can reach each other by IP, but `ping web` will not work — there is no DNS. Create a *user-defined* bridge network and that changes. Docker runs an embedded DNS resolver, reachable from inside every container on the network at the special address 127.0.0.11, and registers each container's name in it. Containers then resolve one another by name automatically:

```bash
docker network create mynet
docker run -d --name api --network mynet nginx
docker run --rm --network mynet busybox \
  ping -c1 api
```

A user-defined bridge gives containers DNS by name.

```
PING api (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 \
     time=0.071 ms
```

Inside the container, /etc/resolv.conf points at 127.0.0.11; the resolver intercepts queries for container names and answers them, forwarding everything else to the host's real DNS. This is the practical reason the docs steer you toward user-defined networks over the default bridge: automatic service discovery by name, plus better isolation between unrelated networks. Note the subnet jumped to 172.18.0.0/16 — each user-defined network gets its own subnet, its own bridge interface (`br-<id>`), and its own slice of the iptables machinery.

The other drivers, briefly

`--network host` skips all of this. The container shares the host's network namespace outright: no veth, no separate IP, no NAT, no docker0. A server in the container binds the host's ports directly, so `-p` is meaningless and there is zero network isolation. It is fast and occasionally necessary, and it throws away the very thing namespaces give you. `--network none` is the opposite extreme — the container gets an empty namespace with only loopback, exactly the blank canvas from the start of Chapter 8, and nothing else. `overlay` is the multi-host story: it builds a virtual Layer-2 segment spanning several

Docker hosts by wrapping container frames in VXLAN tunnels, so containers on different machines behave as if on one bridge. It is the foundation of Swarm networking and conceptually close to what Kubernetes CNI plugins arrange.

Every Docker concept, mapped to its ch.8 primitive:

docker0 default bridge -> a Linux bridge (br0) container's eth0 -> the far end of a veth pair vethXXXX on the host -> veth end enslaved to the bridge 172.17.0.1 gateway -> IP address on the bridge outbound internet -> ip_forward + MASQUERADE (SNAT) -p 8080:80 publishing -> a DNAT rule in the nat table 127.0.0.11 name lookup -> an embedded DNS resolver There is no Docker networking. There is the Linux network stack, configured for you.

One honest caveat about the future. Everything above is shown through `iptables`, which has been Docker's tool of choice since the beginning. Newer distributions are migrating from iptables to `nftables` as the kernel's packet-filtering frontend, and recent Docker releases are learning to drive nftables directly. The user-facing model — bridge, veth, NAT for outbound, DNAT for publishing — does not change; only the syntax of the rules you read does. On an nftables host you would inspect `nft list ruleset` instead, and find the same MASQUERADE and DNAT logic wearing different clothes.

Origins:

Docker shipped with the docker0 bridge model from its first public release in 2013 — the design barely changed. In 2015 Docker extracted its networking into libnetwork and defined the Container Network Model (CNM): networks, endpoints, and sandboxes, with pluggable drivers. The wider container world took a different fork. Kubernetes and the CNCF standardized on CNI, the Container Network Interface — a thinner contract that just asks a plugin to wire up a namespace and hand back an IP. CNM and CNI solve the same Chapter 8 problem; they only disagree on where the seams between the pieces should fall.

Try it yourself

Put it all together on any Linux Docker host (or WSL2 / a VM). Start a published nginx, then go find every piece by hand: the veth, its docker0 membership, the DNAT and MASQUERADE rules, the container's eth0, and finally a real request through the whole chain.

```bash
docker run -d -p 8080:80 --name web nginx
# the host end of the cable, enslaved to docker0
ip link | grep -A1 veth
# the publish (DNAT) and outbound (SNAT) rules
sudo iptables -t nat -L DOCKER -n | grep dpt:8080
sudo iptables -t nat -L POSTROUTING -n \
  | grep MASQUERADE
# enter the container's netns and see eth0
pid=$(docker inspect -f '{{.State.Pid}}' web)
sudo nsenter -t "$pid" -n ip addr show eth0
# drive a request through DNAT -> veth -> nginx
curl -s -o /dev/null -w '%{http_code}\n' \
  localhost:8080
```

Find the veth, the rules, the eth0 — then curl through it.

```
5: vetha1b2c3@if4: <...,UP> ... master docker0
DNAT  tcp ... tcp dpt:8080 to:172.17.0.2:80
MASQUERADE  all  172.17.0.0/16  0.0.0.0/0
4: eth0@if5: <...,UP> mtu 1500 ...
    inet 172.17.0.2/16 ... scope global eth0
200
```

Walk that output backward and you have re-derived the chapter. The `200` came back because curl hit localhost:8080, the DNAT rule rewrote it to 172.17.0.2:80, the kernel forwarded it across docker0 and through the veth named vetha1b2c3 into the container's eth0, where nginx answered. Every hop is a primitive you built by hand in Chapter 8: a bridge, a veth pair, a routing table, IP forwarding, and two NAT rules. Docker did not invent a networking stack. It learned to run the one Linux already had — quickly, repeatably, and so smoothly that it looked like magic. Now you can see straight through it.

PART V

5.3 Storage: Layers, Volumes, and the Writable Top

overlay2, the container layer, and persistent data

In chapter 11 you built an overlay by hand and watched an image's shape emerge: a stack of read-only lower layers, a single writable upper, and a merged tree that looks like an ordinary root filesystem. Chapter 15 then showed how those layers are content-addressed and shared. This chapter takes that mechanism and follows it onto disk. Where exactly does the overlay2 driver put things? What happens to your data when a container is removed? And when the answer is "it's gone," what are the three ways to keep data alive past a container's lifetime? By the end you will be able to point at a directory under `/var/lib/docker` and say what each path is, and you will never again confuse a volume with a bind mount.

The writable layer, made concrete

Recall the deal from chapter 11. An image is a stack of read-only layers. When you `docker run`, Docker mounts an overlay whose lowerdirs are those image layers and whose upperdir is a fresh, empty directory. That empty upper is the container's writable layer. Every byte the container writes at runtime — a log line, a PID file, a temp file, a package installed after start — lands in that upper. Nothing it writes touches the image layers, which stay read-only and shared with every other container of the same image.

The overlay2 storage driver keeps all of this under one directory: `/var/lib/docker/overlay2/`. Each layer is a subdirectory there. Inside a layer directory you will find a `diff/` subdirectory — the actual files that layer contributes — and a `link` file holding a short symbolic name, plus a `lower` file listing the layers beneath it. The short names live under `/var/lib/docker/overlay2/l/` as symlinks, purely so the mount command's `lowerdir=` string stays under the kernel's page-size limit; a stack of long layer IDs would overflow it.

A running container's overlay mount reads exactly like the one you typed by hand in chapter 11, just with longer paths. Its `lowerdir` is the colon-separated stack of image layers. Its `upperdir` is the container's own `diff/` — the writable layer. Its `workdir` is overlayfs's atomic scratch space. And its `merged` directory is the rootfs the container actually sees as `/`. You can pull these four paths straight out of Docker:

```bash
docker run -d --name web nginx >/dev/null
docker inspect web \
  -f '{{json .GraphDriver.Data}}' | tr ',' '\n'
```

The four overlay paths Docker handed the kernel.

OUTPUT (abridged)

```
{"LowerDir":"/var/lib/docker/overlay2/a1.../diff:
  /var/lib/docker/overlay2/b2.../diff"
"MergedDir":"/var/lib/docker/overlay2/c3.../merged"
"UpperDir":"/var/lib/docker/overlay2/c3.../diff"
"WorkDir":"/var/lib/docker/overlay2/c3.../work"}
```

Now look at the live kernel mount and confirm Docker is doing nothing exotic — the same `lowerdir`/`upperdir`/`workdir` options from chapter 11's hands-on, at scale:

BASH

```bash
mount | grep -m1 overlay | tr ',' '\n' | head -5
```

The real overlay mount on the host.

OUTPUT (abridged)

```
overlay on /var/lib/docker/overlay2/c3.../merged
  type overlay (rw,relatime
lowerdir=/var/lib/docker/overlay2/l/AAA:.../l/BBB
upperdir=/var/lib/docker/overlay2/c3.../diff
workdir=/var/lib/docker/overlay2/c3.../work)
```

The `UpperDir` ending in `/diff` is the one to remember: it is the physical directory holding everything this container has written. If you create a file inside the container, it appears there on the host. That is not a metaphor — it is the same inode.

The ephemerality problem

The writable layer has three properties that make it the wrong place for data you care about. First, it is ephemeral: `docker rm` deletes the container's `diff/`, `merged/`, and `work/` directories outright. Every change made inside a removed container is gone. This is by design — the writable layer is scratch space, and disposability is the point of a container. But it surprises anyone who treated a container like a small server.

Second, writing through an overlay is slower than writing to a native filesystem. The culprit is copy-up, from chapter 11: the first write to any file that lives in a lower layer copies the entire file up into the writable layer before the write proceeds. Flip one byte of a 2 GB file and you copy 2 GB. Third, the writable layer is private and unshareable — no other container can see it, and it cannot be backed by network storage. So data you want to keep, share, or write fast must live outside the layer system entirely. The mechanism for that is mounts.

Three kinds of mount

Docker offers three ways to put storage into a container that is not part of the layer stack: volumes, bind mounts, and tmpfs mounts. They differ in where the data lives and how long it survives, but mechanically they are all the same act — a `mount` the runtime performs inside the container's mount namespace (chapters 7 and 13), listed in the OCI `config.json` `mounts` array (chapter 16). Volumes are not magic; they are bind mounts of a Docker-managed directory.

1. Volumes: Docker-managed storage

A volume is a directory Docker owns and tracks for you. Create one with `docker volume create`, or let `docker run -v name:/path` create it on first use; the modern, explicit spelling is `--mount type=volume,src=name,dst=/path`. The data lives under `/var/lib/docker/volumes/<name>/_data`, and Docker bind-mounts that host directory onto the container path — the very `mount --bind` you met in chapter 3. Because the storage sits outside the container's writable layer, it survives `docker rm`, and a fresh container can mount the same volume and find the data intact. Volumes are the preferred mechanism: portable across hosts, managed by Docker, and able to use volume drivers that back them with NFS or cloud storage instead of local disk.

```bash
docker volume create app-data
docker volume inspect app-data \
  -f '{{.Mountpoint}}'
```

A volume is just a managed host directory.

```
OUTPUT
app-data
/var/lib/docker/volumes/app-data/_data
```

Now prove persistence. Write to the volume from one container, destroy that container, then mount the same volume in a new one and read the data back:

```bash
docker run --rm -v app-data:/d alpine \
  sh -c 'echo hello > /d/note.txt'
docker run --rm -v app-data:/d alpine \
  cat /d/note.txt
```

First container is already gone; data is not.

```
OUTPUT
hello
```

The first container was removed by `--rm` the instant its `sh -c` finished, taking its writable layer with it. The note survived because it never lived in that layer — it lives in `/var/lib/docker/volumes/app-data/_data/note.txt` on the host, which you can `cat` directly.

2. Bind mounts: an arbitrary host directory

A bind mount maps a path you choose on the host straight into the container: `-v /host/path:/ctr/path`, or `--mount type=bind,src=/host/path,dst=/ctr/path`. Unlike a volume, Docker does not manage or even create the storage — it is whatever is already at that host path, mounted in with the same `mount --bind` from chapter 3. This is the developer's favorite: bind-mount your source tree into the container and edits on either side are instantly visible on the other, because both sides are the same files. The cost is coupling — the container now depends on the host's directory layout — and a real security consideration, since the container can read and write host files outside its image.

Two option suffixes matter. Append `:ro` to mount read-only, so the container cannot modify the host files at all. On SELinux-enforcing hosts (chapter 12) append `:z` or `:Z` to relabel the host directory for container access: lowercase `:z` applies a shared label so several containers may use it, while uppercase `:Z` applies a private label for one container only. Without the relabel, SELinux will deny the container access to your bind-mounted directory and you will see permission errors that have nothing to do with file modes.

```bash
mkdir -p /tmp/src && echo v1 > /tmp/src/f.txt
docker run --rm \
  -v /tmp/src:/work:ro alpine \
  cat /work/f.txt
```

Read-only bind mount of a host directory.

```
OUTPUT
v1
```

3. tmpfs mounts: RAM-backed scratch

A tmpfs mount, `--tmpfs /path` or `--mount type=tmpfs,dst=/path`, mounts a RAM-backed filesystem (chapter 3) into the container. It never touches disk and vanishes when the container stops — there is no host directory to find afterward. That makes it the right place for secrets you do not want persisted and for high-churn scratch data you never need to keep. Confirm it is genuinely tmpfs and genuinely not on disk:

```bash
BASH

docker run --rm --tmpfs /scratch alpine \
  sh -c 'echo s > /scratch/x; df -T /scratch'
```

tmpfs lives in RAM, not under /var/lib/docker.

```
OUTPUT (abridged)

Filesystem  Type  1K-blocks Used ... Mounted on
tmpfs       tmpfs    ...        0 ... /scratch
```

The three mount types:

VOLUME — /var/lib/docker/volumes/<n>/_data ; survives rm ; managed, portable, drivers (NFS/cloud) ; default pick. BIND — any host path you choose ; lives as long as host dir ; dev source trees ; couples to host, security-sensitive. TMPFS — RAM only, never on disk ; gone on stop ; secrets, scratch.

Where it all lives, and what it costs

Two directories hold almost everything: image and container layers sit under `/var/lib/docker/overlay2/`, and volumes under `/var/lib/docker/volumes/`. Docker can total it up for you. `docker system df` breaks down space by images, containers, local volumes, and the build cache, and the RECLAIMABLE column tells you how much you would get back by pruning unused objects.

```bash
BASH

docker system df
du -sh /var/lib/docker/overlay2
```

Account for layer, container, and volume space.

```
OUTPUT (abridged)

TYPE          TOTAL  ACTIVE  SIZE   RECLAIMABLE
Images        12     4       3.1GB  1.8GB (58%)
Containers    7      2       142MB  98MB (69%)
Local Volumes 3      1       512MB  340MB
Build Cache   88     0       2.2GB  2.2GB
3.4G  /var/lib/docker/overlay2
```

Layer sharing is what keeps these numbers sane. Because the image layers are read-only lowerdirs shared by every container of that image, ten containers started from one image cost roughly one copy of the image plus ten tiny upper layers — not ten copies of the image. The expensive bytes are stored once; only each container's private diff is

duplicated, and a freshly started container's diff is usually a few kilobytes.

Why databases want a volume

Copy-up turns into write amplification for any workload that rewrites files in place, and a database is the worst case. Its data files start life in image or lower layers, so the first write to each copies the whole file up; then heavy random writes pour into the overlay's writable layer, which was never built for high-throughput I/O. The fix is one flag: put the data directory on a volume, so the database writes straight to a native filesystem outside the overlay, with no copy-up and no layer overhead. It is also the only way the data survives a container rebuild. As a rule, any container whose job is to write a lot of data writes it to a volume, not to its layer.

Origins:

Docker's first storage driver was aufs (2013), an out-of-tree union filesystem that dogged packaging for years. The RHEL world leaned on devicemapper over LVM thin pools. overlay2, built on the in-kernel overlayfs, became the default around Docker 17.06 (mid-2017) and won because it is in-tree, fast, and simple. Volumes and the -v flag date to early Docker; the longer --mount syntax was added later for clarity and to make the type (volume / bind / tmpfs) explicit.

Try it yourself

Three experiments on a Linux box (or WSL2 / a VM). First, find a container's UpperDir and watch a file you create inside the container appear there on the host — proving the writable layer is a real directory under overlay2.

```bash
docker run -d --name u alpine sleep 300 >/dev/null
UP=$(docker inspect u \
  -f '{{.GraphDriver.Data.UpperDir}}')
docker exec u sh -c 'echo hi > /made-inside.txt'
sudo cat "$UP/made-inside.txt"
docker rm -f u >/dev/null
```

The file written in the container is in UpperDir.

```
hi
```

Second, prove a named volume outlives the container that wrote to it. Write in one container, remove it, read from a brand-new one.

```bash
docker volume create keep >/dev/null
docker run --rm -v keep:/d alpine \
  sh -c 'date > /d/when.txt'
docker run --rm -v keep:/d alpine cat /d/when.txt
docker volume rm keep >/dev/null
```

The timestamp survives the first container's removal.

```
OUTPUT
Thu Jun 26 10:14:02 UTC 2026
```

Third, bind-mount a host directory and edit it live from both sides. Change it on the host, read the change in the container, then change it in the container and read it back on the host — they are the same files.

```bash
mkdir -p /tmp/live && echo host-1 > /tmp/live/f
docker run -d --name b \
  -v /tmp/live:/work alpine sleep 300 >/dev/null
docker exec b cat /work/f
docker exec b sh -c 'echo ctr-2 > /work/f'
cat /tmp/live/f
docker rm -f b >/dev/null
```

Both sides see the same inode in real time.

```
OUTPUT
host-1
ctr-2
```

Read the three together. The writable layer is a directory you can open on the host but that dies with the container. A volume is a Docker-managed directory under `/var/lib/docker/volumes` that outlives it. A bind mount is a host directory of your choosing spliced in. All three are the same kernel operation — a bind into the container's mount namespace — differing only in who owns the directory on the other end and how long it is meant to last.

PART V

5.4 The Life of a docker run

One command, traced from Enter to syscall

This is the chapter the whole book was building toward. You have met every part in isolation: the process and its syscalls (Chapters 1-2), the filesystem and privilege model (Chapters 3-4), the namespaces (Chapters 7-9), cgroups (Chapter 10), overlayfs (Chapter 11), capabilities and seccomp (Chapter 12), pivot_root (Chapter 13). You even assembled them by hand into a real container with nothing but shell commands (Chapter 14). Then you saw the engineering on top: the image format (Chapter 15), the daemon stack (Chapter 16), the networking plumbing (Chapter 17), and the layer lifecycle (Chapter 18).

Now we run one command and watch every one of those pieces fire, in order, naming each component and each kernel primitive as it does its job. The command is deliberately loaded with flags so that all four restraints show up:

```bash
docker run -d -p 8080:80 \
  --memory=256m --name web nginx
```

Detached, port-mapped, memory-capped, named.

By the time this returns a container ID, the kernel has created namespaces, written a cgroup, pivoted a root filesystem, dropped capabilities, installed a seccomp filter, rewritten the firewall, and exec'd nginx as PID 1 of a brand new world. Let us follow it down.

Stage 1 - the CLI talks to the daemon (ch.16)

Pressing Enter runs the `docker` binary, and that binary does almost nothing interesting. It parses your flags, builds a JSON request, and POSTs it to the dockerd daemon over a Unix socket — the Engine API from Chapter 16. The client never touches the kernel; it is a thin HTTP front-end.

`docker run` is not one operation. It is sugar for `create` + `start` (and a `pull` first if the image is missing). You can drive the same API yourself with curl over the socket and watch the daemon answer:

```bash
curl --unix-socket /var/run/docker.sock \
  -H 'Content-Type: application/json' \
  -X POST http:/v1.45/containers/create?name=web \
  -d '{"Image":"nginx",
       "HostConfig":{
         "Memory":268435456,
         "PortBindings":{"80/tcp":[
           {"HostPort":"8080"}]}}}'
```

The exact request the CLI builds for you.

```
OUTPUT
{"Id":"d9b1a7e4c2f0...","Warnings":[]}
```

That returns a container ID but starts nothing. A second call, POST /containers/web/start, is what actually brings the container to life. The CLI fires both for you and hides the seam — but the seam is real, and it is why `docker create` and `docker start` exist as separate commands.

Stage 2 - image resolution and pull (ch.15)

Before anything can start, dockerd needs the nginx image. It checks the local image store under /var/lib/docker. If the image is absent, it pulls — and the pull is exactly the registry protocol from Chapter 15.

The pull, step by step (ch.15):

1. resolve the tag 'nginx:latest' to a digest 2. GET the manifest for that digest 3. GET the config blob (the image's metadata) 4. GET each layer blob the manifest lists 5. unpack each layer into overlay2 as a snapshot

Each layer is a tar of filesystem changes. dockerd unpacks them into /var/lib/docker/overlay2 as stacked snapshots — the read-only lower layers you met in Chapters 11 and 18. The image config also carries the Entrypoint and Cmd, which decide what process the container will run; hold that thought for the execve at the end.

Stage 3 - dockerd hands off to containerd (ch.16)

dockerd does not run containers itself. It records a container in its own database, then calls containerd over gRPC (Chapter 16). containerd's first job is to prepare the snapshot: it mounts the overlay, stacking the image's read-only layers as lowerdir and adding a fresh writable upperdir on top. The merged result is the container's root filesystem (Chapters 11 and 18).

```bash
mount -t overlay overlay \
  -o lowerdir=<img L3>:<L2>:<L1>,\
  upperdir=<diff>,workdir=<work> \
  <merged>
```

Image layers below, the container's writes above.

This is the same mount you ran by hand in Chapter 14. The writable upperdir is where everything the container changes at runtime lands; the image stays pristine, which is why a thousand containers can share one nginx image.

Stage 4 - building the OCI bundle (ch.15, ch.16)

containerd now assembles a runtime bundle: a directory holding the merged rootfs and a single file, config.json. That file is the entire container expressed as data. Read it and you are reading the four restraints, declared rather than executed:

```bash
{
 "process": {
  "args": ["nginx","-g","daemon off;"],
  "capabilities": { "effective": [
     "CAP_CHOWN","CAP_NET_BIND_SERVICE"] },
  "noNewPrivileges": true },
 "linux": {
  "namespaces": [
    {"type":"pid"},  {"type":"network"},
    {"type":"mount"},{"type":"uts"},
    {"type":"ipc"} ],
  "resources": { "memory": {
     "limit": 268435456 } },
  "seccomp": { "defaultAction":
     "SCMP_ACT_ERRNO" } },
 "mounts": [
   {"destination":"/proc","type":"proc"},
   {"destination":"/sys","type":"sysfs"},
   {"destination":"/dev","type":"tmpfs"} ]
}
```

A trimmed config.json: every restraint as data.

config.json IS the four restraints (declared):

***namespaces[]   -> the new worlds   (ch.7-9) resources.memory ->
cgroup, 256m cap   (ch.10) capabilities   -> the default set   (ch.12)
seccomp   -> the syscall filter (ch.12) mounts /proc /sys /dev +
volumes (ch.3, ch.18) process.args   -> image Entrypoint   (ch.15)***

Notice that nothing in this file has happened yet. It is a blueprint. Turning it into a running,

restrained process is the job of the last two stages — and it is where the kernel finally takes over.

Stage 5 - the shim and runc (ch.16)

containerd does not call runc directly either. It starts a containerd-shim-runc-v2 process — the durable parent from Chapter 16 — and it is the shim that invokes `runc create` then `runc start` against the bundle. The shim matters because it outlives runc: runc does the setup and exits, but the shim stays as PID 1's parent so the container survives even if dockerd or containerd restart. This is why you can upgrade Docker without killing your containers.

Who calls whom (ch.16):

docker --HTTP/socket--> dockerd dockerd --gRPC---------> containerd containerd --spawns----> containerd-shim-runc-v2 shim --exec---------> runc create / runc start

Stage 6 - runc does the kernel work (Part III)

This is the climax. Everything above was bookkeeping and message-passing in user space. runc is where config.json stops being data and becomes syscalls — the exact syscalls you spent Part III learning. Here is the order, each step tagged with the chapter that taught it:

runc's syscalls, in order:

1. clone()/unshare() with the CLONE_NEW* flags NEWNS|NEWPID|NEWNET|NEWUTS|NEWIPC (+NEWUSER) (ch.2, ch.7-9) 2. write PID into the cgroup -> memory.max=256m (ch.10) 3. set up the mount namespace: mount --make-rprivate; bind the rootfs; pivot_root onto the overlay merged dir; mount /proc, /sys, /dev (ch.3, ch.11, ch.13) 4. sethostname() in the UTS namespace (ch.7) 5. drop capabilities to the default set, set no_new_privs, install seccomp BPF (ch.4, ch.12) 6. execve() the entrypoint (nginx) as PID 1 (ch.1, ch.2)

Read that list against the container you built by hand in Chapter 14. It is the same sequence: unshare the namespaces, join the cgroup, pivot_root over an overlay, drop caps and filter syscalls, then exec the workload. runc is a compiled, spec-driven, race-free version of your shell script. Nothing more.

The clone() flags decide which of the host's worlds the new process keeps and which it

gets fresh. The cgroup write is what makes --memory=256m real: from this instant nginx and every child it forks are accounted against memory.max, and crossing it summons the OOM killer (Chapter 10). The pivot_root is the moment the host filesystem vanishes and the nginx image becomes / (Chapter 13). The capability drop and seccomp filter shrink the kernel interface so a compromised nginx cannot reach the syscalls it has no business calling (Chapter 12).

And then the final act: execve(). runc replaces itself with nginx. Because it is the first process in the new PID namespace, nginx becomes PID 1 inside (Chapters 1-2). runc exits, having done its work; the shim remains as the parent. A restrained process is now running.

Stage 7 - networking is wired up (ch.8, ch.17)

A container in a fresh network namespace is born offline — only a down loopback, no route off the machine, exactly as in Chapter 8. dockerd's networking layer (libnetwork) fixes that from the host side, with the veth dance from Chapters 8 and 17:

Connecting the container (ch.8, ch.17):

1. create a veth pair (a virtual cable) 2. attach one end to the docker0 bridge 3. move the other end into the net namespace and rename it eth0 4. assign 172.17.x.x and a default route 5. for -p 8080:80, insert an iptables DNAT rule (plus MASQUERADE for outbound)

That last rule is what makes -p 8080:80 work. A DNAT entry in the nat table rewrites packets arriving on the host's port 8080 to the container's 172.17.x.x:80, and a MASQUERADE rule lets the container's replies and outbound traffic find their way back out (Chapter 17). The kernel's own firewall is doing the port publishing; Docker just writes the rules.

Stage 8 - it is just a process now (ch.1)

The container is running, and the punchline of the whole book is this: nginx is an ordinary host process. It wears four restraints, but it is scheduled by the same scheduler, shows up in the same process table, and can be signalled like any other process. From the host's god's-eye view there is no isolation of visibility at all:

```bash
ps -ef | grep nginx
docker inspect web --format '{{.State.Pid}}'
curl -s localhost:8080 | head -1
```

The container, seen three ways from the host.

OUTPUT

```
root  31847  31820  nginx: master process
syst  31889  31847  nginx: worker process
31847
<!DOCTYPE html>
```

PID 31847 on the host is PID 1 inside the container — one process, two numbers, because of the PID namespace (Chapter 7). curl to localhost:8080 lands on the host's port, gets DNATed into the container, and nginx answers. Everything works, and everything is explainable.

Stage 9 - stop and remove (ch.18)

Tearing down runs the lifecycle in reverse. `docker stop` sends SIGTERM to PID 1 and, if it does not exit within the grace period, SIGKILL — plain signals to a plain process (Chapter 1). `docker rm` then discards the writable upper layer (Chapter 18), so every change the container made evaporates, and tears down the veth and the iptables rules from Stage 7. The image's read-only layers are untouched, ready for the next container.

The whole pipeline, in one column

docker run, end to end:

docker parse flags, POST to the socket (16) v dockerd pull image, create record (15,16) v containerd prepare overlay snapshot (11,18) v containerd build OCI bundle = rootfs+config (15) v shim start, invoke runc (16) v runc clone + cgroup + pivot_root + caps + seccomp + execve (Part III) v = a restrained host process running nginx (1)

See it with your own eyes (ch.2)

None of this is a metaphor. The promise of Chapter 2 was that you can watch a program's syscalls go by, and you can cash it in here. strace runc (or attach to the shim) and the entire climax scrolls past in real syscalls:

BASH

```bash
strace -f -e trace=clone,unshare,mount,\
pivot_root,setns,prctl,execve \
  runc create --bundle <bundle> mycid
```

The four restraints, live, as kernel calls.

```
OUTPUT
clone(... CLONE_NEWNS|CLONE_NEWPID|
  CLONE_NEWNET|CLONE_NEWUTS|CLONE_NEWIPC) = 31847
mount("proc","/proc","proc",...) = 0
pivot_root(".","oldroot")           = 0
prctl(PR_SET_NO_NEW_PRIVS, 1)         = 0
prctl(PR_SET_SECCOMP, ...)            = 0
execve("/usr/sbin/nginx", ...)        = 0
```

There it is, the book's whole thesis on one screen: clone for the namespaces, mount and pivot_root for the rootfs, prctl for no_new_privs and seccomp, execve for the workload. Every line is a syscall you can make yourself.

That is the closing message. Nothing in `docker run` is magic. Every single step is a feature of the Linux kernel that you can drive with your own hands — and in Chapter 14 you did. Docker is superb engineering: an image format, a daemon, a registry protocol, a network layer, and a one-line command, all layered over mechanisms the kernel already provided. It did not invent isolation. It made the kernel's own isolation easy enough that we forgot it was there. Now you can never unsee it.

Try it yourself

Run the exact command, then verify each restraint from the outside — from the host, where there is no isolation and you can inspect everything. This battery proves, one restraint at a time, that the container is exactly the sum of the kernel features this book described.

```bash
BASH
docker run -d -p 8080:80 \
  --memory=256m --name web nginx
PID=$(docker inspect web \
  --format '{{.State.Pid}}')
echo $PID
```

Start it; grab the host PID of container PID 1.

First, the process restraint (Chapter 1): nginx is a normal host process. The host sees it; the container's own view is narrower, but the host's god view is total.

```bash
BASH
ps -o pid,comm -p $PID
```

A container is a process on the host.

```
OUTPUT
  PID COMMAND
31847 nginx
```

Second, the cgroup (Chapter 10). Read the process's cgroup and then the memory ceiling — it must be 256 MB, the 268435456 bytes you asked for:

```bash
cat /proc/$PID/cgroup
CG=/sys/fs/cgroup/$(cat /proc/$PID/cgroup \
  | cut -d: -f3)
cat $CG/memory.max
```

The cgroup, and the cap you set, on disk.

```
0::/system.slice/docker-d9b1a7e4c2f0.scope
268435456
```

Third, the namespaces (Chapters 7-9). Each is an inode under /proc/<pid>/ns; the container's differ from the host's, which is the whole point — different inode, different world:

```bash
ls -l /proc/$PID/ns
```

Distinct inodes = distinct namespaces.

```
net  -> net:[4026532567]
pid  -> pid:[4026532570]
mnt  -> mnt:[4026532565]
uts  -> uts:[4026532563]
ipc  -> ipc:[4026532566]
```

Fourth, the network namespace (Chapters 8, 17). Enter just the net namespace with nsenter and look: eth0 with a 172.17 address that does not exist anywhere on the host:

```bash
nsenter -t $PID -n ip addr show eth0
```

The container's private eth0 and IP.

```
eth0: <BROADCAST,MULTICAST,UP,LOWER_UP>
  inet 172.17.0.2/16 brd 172.17.255.255
    scope global eth0
```

Fifth, capabilities and seccomp (Chapter 12). The status file shows the trimmed capability bounding set and that a seccomp filter is installed (Seccomp: 2 means filter mode):

```bash
grep -E 'Cap|Seccomp' /proc/$PID/status
```

Dropped caps and an active syscall filter.

```
CapEff:  00000000a80425fb
CapBnd:  00000000a80425fb
Seccomp: 2
Seccomp_filters:        1
```

Five commands, five restraints, all confirmed from the outside against a live container. Run them once and the book stops being a sequence of chapters and becomes a single picture: a process, in its own namespaces, capped by a cgroup, rooted in an overlay, narrowed by caps and seccomp, wired to the network by a veth and a firewall rule. That is a container. You can build it, you can break it down, and now you can see exactly how Docker does both.

EPILOGUE

Beyond Docker

Podman, Kubernetes and the CRI, sandboxed runtimes, and where this is heading.

Epilogue

Beyond Docker

What you can now see through

When you started this book, a container may have looked like a small, sealed machine. You can no longer see it that way, and that is the point. You now know that the machine was never there — that a container is one ordinary process the kernel has been told to show a private view of the world (namespaces), to live within a budget (cgroups), to treat a stacked, copy-on-write directory as its root (overlayfs and pivot_root), and to ask the kernel only for a vetted set of operations (capabilities and seccomp). You built one of these by hand, and then you watched Docker — daemon, containerd, runc — assemble the very same pieces, faster and more carefully than you could by typing.

> **The whole book in one line:**
>
> ***There is no container. There is a process, and there is everything the kernel agreed to hide from it.***

Docker is not the only one

Because the mechanisms live in the kernel and the formats live in the OCI specifications, Docker is just one program that drives them — and increasingly not the one running in production. Knowing the primitives means every alternative is now legible to you rather than being a fresh thing to learn.

Podman builds containers from the same OCI bundles and the same runc, but with no long-running daemon: each container is a direct child of the command you ran, which makes rootless operation (Chapter 9's user namespaces, used in earnest) the natural default rather than a bolt-on. containerd, which you met inside Docker, is perfectly happy without Docker on top of it; it is the runtime Kubernetes talks to directly.

Kubernetes and the CRI

At scale, no one types `docker run`. Kubernetes schedules containers across a fleet, and it does not speak to Docker at all anymore. It speaks the Container Runtime Interface (CRI) to a runtime like containerd or CRI-O, which in turn calls runc, which makes the same clone, setns, mount, and pivot_root calls you now know by name. The famous 2020 "Kubernetes is deprecating Docker" headlines panicked a lot of people who pictured containers as Docker-shaped; readers of this book would have shrugged, because the layer being removed was only the part that translated to Docker's API — the container underneath was always the kernel's, not Docker's.

When namespaces aren't enough

A container shares the host kernel. That is its great efficiency and its great limitation: a kernel vulnerability is a shared wall, and a syscall the seccomp filter forgot is a way through it. Two families of runtime push back. gVisor (Google) puts a user-space kernel between the container and the host, intercepting syscalls so the real kernel sees far fewer of them. Kata Containers goes the other way and wraps each container in a genuine, lightweight virtual machine — turning the tradeoff dial back toward VMs, but keeping the OCI interface so the tools above don't notice. Both plug in where runc does, which is exactly why they could exist at all.

Where this is heading

Watch a few currents. Rootless and unprivileged containers are becoming the default rather than the brave choice, on the back of maturing user namespaces. cgroups v2 has won, and with it finer, more honest resource control. WebAssembly is emerging as a second kind of sandbox — not a Linux process at all, but increasingly carried by the same OCI registries and orchestrators, which is only possible because the industry spent a decade separating the format from the mechanism. The mechanisms themselves keep advancing too: eBPF now lets you observe and enforce policy in the kernel with a precision that makes the old iptables rules of Chapter 17 look quaint.

But the foundation under all of it is the one you now hold. New runtimes, new sandboxes, new orchestrators will keep arriving, and the marketing around them will keep implying that something fundamentally new has been invented. Usually it hasn't. Usually it is a new arrangement of namespaces, cgroups, a union filesystem, and a syscall filter — the same four restraints, dressed differently. You can see through the dress now. That is a durable thing to know, and it will outlast every tool named in this book.

Go on, then:

Open a shell. unshare a namespace. Watch a process believe it is alone in the world. You know exactly how the trick is done.

APPENDIX A

Glossary of Terms

Every kernel feature, daemon, and standard named in the book, defined in one place.

Appendix A — Glossary of Terms

`namespace` *kernel isolation primitive*

A kernel feature that gives a process its own private view of one class of global resource (PIDs, mounts, network, users, ...).

Appears in: ch. 7-9

`cgroup` *control group*

A kernel mechanism that limits and accounts a process group's use of CPU, memory, and I/O.

Appears in: ch. 10

`capability` *split-up root privilege*

One slice of the powers historically bundled into root (e.g. CAP_NET_ADMIN), grantable independently.

Appears in: ch. 4, 12

`overlayfs` *union filesystem*

A copy-on-write filesystem that stacks read-only image layers under a writable layer; the basis of container images.

Appears in: ch. 11

`runc` *OCI runtime*

The small reference tool that actually creates a container from an OCI bundle by calling clone/unshare, setns, and friends.

Appears in: ch. 16

`OCI` *Open Container Initiative*

The standards body and specs (image + runtime) that decoupled the container format from Docker.

Appears in: ch. 6, 15

`process` *running program*

A live instance of a program in memory: an address space, execution context, and kernel bookkeeping (a task_struct).

Appears in: ch. 1

`PID` *process identifier*

The integer the kernel assigns to identify a process; PID 1 is the special init process that adopts orphans and reaps them.

Appears in: ch. 1

`fork` *copy a process*

The syscall that creates a new process by duplicating the caller with copy-on-write memory; it returns twice.

Appears in: ch. 1

`exec` *replace the image*

The syscall family that discards the current program image and loads a new program in the same process, keeping the PID.

Appears in: ch. 1

`clone` *general fork*

The flexible primitive beneath fork; with CLONE_NEW* flags it creates a process in new namespaces — how containers are made.

Appears in: ch. 1

`zombie` *unreaped dead process*

A process that has exited but whose exit status has not yet been collected by its parent via wait().

Appears in: ch. 1

`syscall` *system call*

The single controlled entry point a ring-3 process uses to ask the kernel to perform a privileged operation on its behalf.

Appears in: ch. 2

`user space` *user space (ring 3)*

The unprivileged CPU mode where ordinary processes run, unable to touch hardware except by making system calls.

Appears in: ch. 2

`kernel space` ***kernel space (ring 0)***

The privileged CPU mode where the kernel runs, with direct access to all hardware and the right to run any instruction.

Appears in: ch. 2

`ABI` ***application binary interface***

The stable low-level contract — including syscall numbers and register conventions — between user programs and the kernel.

Appears in: ch. 2

`strace` ***syscall tracer***

A tool that runs a program and prints every system call it makes, revealing everything the program does that affects the outside world.

Appears in: ch. 2

`hypervisor` ***virtual machine monitor***

The layer a virtual machine traps into to emulate hardware for a guest kernel — the component a container has no equivalent of.

Appears in: ch. 2

`inode` ***index node***

The on-disk record that *is* a file — type, owner, permissions, timestamps, link count, and pointers to the data blocks — identified by a number rather than by any name.

Appears in: ch. 3

`hard link` ***hard link***

A directory entry that points at an already-existing inode, giving one file a second equal name and raising its link count.

Appears in: ch. 3

`VFS` ***Virtual File System***

The kernel layer that defines one common interface so every filesystem type — ext4, tmpfs, overlay, proc — appears as part of a single tree.

Appears in: ch. 3

`mount` *mount*

Attaching a filesystem's tree at a directory (the mount point) so the unified "/" is assembled from many separate filesystems.

Appears in: ch. 3

`bind mount` *bind mount*

A mount that makes an existing subtree appear at a second path, sharing its inodes rather than copying data — the basis of Docker volumes.

Appears in: ch. 3

`pseudo-filesystem` *pseudo-filesystem*

A filesystem with no backing storage (proc, sysfs, cgroup2, tmpfs) that presents kernel interfaces as files readable and writable through the VFS.

Appears in: ch. 3

`uid` *user ID*

The integer identity a process runs as; the kernel checks this number, not a username, for almost every permission decision.

Appears in: ch. 4

`setuid bit` *set-user-ID bit*

A file permission bit that makes an executable run with the effective uid of its owner rather than its caller, the classic and risky way to grant privilege.

Appears in: ch. 4

`CAP_SYS_ADMIN` *the catch-all capability*

A capability so broad — covering mount, pivot_root, and much more — that holding it is nearly equivalent to full root.

Appears in: ch. 4

`bounding set` *capability bounding set*

A per-process ceiling on capabilities; a capability dropped from it can never be regained, even across an execve.

Appears in: ch. 4

`no_new_privs` *no-new-privileges flag*

An irreversible, inherited process flag that forbids gaining privilege through execve, neutralizing setuid binaries and file capabilities.

Appears in: ch. 4

`chroot` *change root*

A Unix system call (Version 7, 1979) that changes a process's apparent root directory; it isolates the filesystem view only, and root can escape it, which is why it is not a security boundary on its own.

Appears in: ch. 5

`FreeBSD jail` *jail*

A FreeBSD isolation feature (2000, Poul-Henning Kamp) that confines a process group with its own filesystem, network, process view, and a contained root user — arguably the first true container.

Appears in: ch. 5

`Solaris Zone` *zone*

Sun's OS-level virtualization primitive in Solaris 10 (2004), an isolated user environment on a shared kernel, paired with per-zone resource management.

Appears in: ch. 5

`OpenVZ` *Open Virtuozzo*

Out-of-tree Linux kernel patches (SWsoft, 2005) providing container-like isolation and resource limits for VPS hosting; never merged into mainline Linux.

Appears in: ch. 5

`Linux-VServer` *VServer*

An out-of-tree Linux patch set (Jacques Gelinas, 2001) that partitioned a system into isolated security contexts; widely used by hosts, but never merged into the mainline kernel.

Appears in: ch. 5

`cgroups` *Control Groups*

Kernel mechanism, born at Google in 2006 (Paul Menage, Rohit Seth) and merged in 2.6.24 (2008), that accounts for and limits a process group's resource use (CPU, memory).

Appears in: ch. 6

LXC — *Linux Containers*

First userspace toolset (2008, key authors Daniel Lezcano and Serge Hallyn at IBM) to combine namespaces and cgroups into usable containers; Docker's original foundation.

Appears in: ch. 6

Borg — *Google cluster manager*

Google's internal cluster manager from the mid-2000s that ran workloads in containers at massive scale; its lineage led to lmctfy (2013) and informed Kubernetes (2014).

Appears in: ch. 6

---

libcontainer — *Docker's native runtime library*

Go library Docker introduced in 2014 to talk directly to kernel namespaces and cgroups, replacing its LXC dependency; later donated to the OCI as runc.

Appears in: ch. 6

---

containerd — *High-level container runtime*

Docker's high-level runtime, split out and donated to the CNCF in 2017; manages images and container lifecycle above the low-level runc.

Appears in: ch. 6

---

unshare — *unshare*

The syscall (and CLI tool) that detaches the *calling* process into new namespaces in place, without creating a child.

Appears in: ch. 7

---

setns — *set namespace*

The syscall that makes the calling process *join* an existing namespace, named by an fd opened on /proc/<pid>/ns/<type>; the nsenter tool wraps it.

Appears in: ch. 7

PID namespace — *process-ID namespace*

An isolated numbering of process IDs starting at 1; its PID 1 reaps orphans and handles signals specially, and its death kills every process in the namespace.

Appears in: ch. 7

`mount namespace` ***mount namespace***

CLONE_NEWNS: a private list of mounts, so a filesystem mounted inside is invisible to the host — the basis of a container's own filesystem tree.

Appears in: ch. 7

`UTS namespace` ***UTS namespace***

Isolates the hostname and NIS domain name; the simplest namespace and the clearest demonstration of per-process resource copies.

Appears in: ch. 7

`network namespace` ***network namespace***

An isolated copy of the kernel's networking stack — its own interfaces, routing table, firewall rules, ARP cache, and port space; a fresh one contains only a down loopback interface.

Appears in: ch. 8

`veth` ***virtual ethernet pair***

Two linked virtual interfaces created together; a packet entering one end emerges from the other, and the ends may sit in different namespaces, making it the cable that connects a container to the host.

Appears in: ch. 8

`ip netns` ***named network namespace helper***

The iproute2 command that creates, lists, and runs commands inside named network namespaces, persisting them under /var/run/netns/.

Appears in: ch. 8

`bridge` ***Linux bridge***

A software Ethernet switch in the kernel; host-side veth ends are enslaved to it so multiple namespaces share one Layer-2 segment — the model behind Docker's docker0.

Appears in: ch. 8

`masquerade` ***source NAT / masquerade***

An iptables rule that rewrites the source address of outbound packets to the host's own address, letting containers on a private subnet reach the internet behind the host.

Appears in: ch. 8

`ip_forward` *IP forwarding flag*

The net.ipv4.ip_forward sysctl that lets the host route packets between interfaces instead of acting as a single endpoint; required for containers to reach networks beyond the host.

Appears in: ch. 8

`user namespace` *user namespace*

The namespace that maps a range of inside uids/gids to a different range outside; the only namespace an unprivileged user may create, and the basis of rootless containers.

Appears in: ch. 9

`uid_map` *UID mapping*

The write-once /proc/<pid>/uid_map file whose lines read "inside-id outside-id count", defining how a user namespace's uids translate to host uids.

Appears in: ch. 9

`subuid` *subordinate uid range*

A block of host uids recorded in /etc/subuid that an administrator delegates to a user for use inside user namespaces — never as that user's own host identity.

Appears in: ch. 9

`newuidmap` *setuid mapping helper*

A trusted setuid binary (with newgidmap) that writes a user namespace's uid_map for a whole delegated range, enforcing the limits in /etc/subuid.

Appears in: ch. 9

`rootless container` *rootless container*

A container run with no host privilege, built by creating a user namespace, becoming root inside it, and using that in-namespace root to create the other namespaces.

Appears in: ch. 9

`unprivileged_userns_clone` *userns restriction knob*

A sysctl (with user.max_user_namespaces) some distributions use to restrict or disable creation of user namespaces by unprivileged users, trading capability for reduced kernel attack surface.

Appears in: ch. 9

`controller` *cgroup controller*

A kernel subsystem (cpu, memory, io, pids, cpuset) attached to a cgroup node that measures and caps one class of resource.

Appears in: ch. 10

`unified hierarchy` *cgroup v2 unified hierarchy*

The single cgroup v2 tree where each process sits at one place and each node enables a subset of controllers, replacing v1's separate per-controller hierarchies.

Appears in: ch. 10

`cpu.max` *CPU bandwidth limit*

The v2 file holding "QUOTA PERIOD" in microseconds; "50000 100000" caps a group at 50,000 us of CPU per 100,000 us — half a core.

Appears in: ch. 10

`memory.max` *memory hard limit*

The v2 file setting a group's hard memory ceiling; exceeding it with nothing to reclaim triggers the cgroup OOM killer, logged in memory.events.

Appears in: ch. 10

`subtree_control` *cgroup.subtree_control*

The file in which a parent delegates controllers to its children by writing e.g. "+cpu +memory"; without it, children have no limit files.

Appears in: ch. 10

`lowerdir` *lower directory*

One or more read-only layers in an overlay mount, stacked with the leftmost winning; these are an image's layers.

Appears in: ch. 11

`upperdir` *upper directory*

The single read-write layer of an overlay mount where all changes land; in Docker this is the container layer, discarded when the container is removed.

Appears in: ch. 11

---

copy-up *copy-up*

Overlayfs copying a file from a read-only lower into the upper before a write, so the lower stays untouched — the heart of copy-on-write, and costly for large files.

Appears in: ch. 11

whiteout *whiteout*

A character device with device numbers 0/0 placed in the upper layer to mask a file that exists in a lower, making it appear deleted in the merged view.

Appears in: ch. 11

storage driver *storage driver*

The Docker component that assembles a container's root filesystem; overlay2 (built on overlayfs) is the modern default, succeeding aufs, devicemapper, btrfs and zfs.

Appears in: ch. 11

seccomp *secure computing mode*

A kernel facility that filters a process's syscalls; in mode 2 a BPF program inspects each syscall's number and scalar arguments and returns an action such as ALLOW, ERRNO, or KILL.

Appears in: ch. 12

seccomp action *seccomp filter verdict*

The result a seccomp filter returns per syscall: ALLOW (proceed), ERRNO (fail with a chosen error), KILL (terminate the process), TRAP, or LOG.

Appears in: ch. 12

default seccomp profile *Docker's seccomp allowlist*

A JSON allowlist Docker installs that permits most syscalls and blocks dangerous ones such as mount, reboot, keyctl, and init_module, usually via the ERRNO action.

Appears in: ch. 12

--privileged *the privileged flag*

A docker run option that removes nearly all restraints at once — full capabilities, all host devices, and an unconfined seccomp and LSM profile — making a container effectively part of the host.

Appears in: ch. 12

`LSM` *Linux Security Module*

A mandatory access-control layer enforced by the kernel on top of the classic checks; AppArmor (path-based) and SELinux (label-based) are the two Docker integrates with.

Appears in: ch. 12

`pivot_root` *pivot the root mount*

The syscall that changes the root mount of the current mount namespace, making new_root the new / and moving the old root to a put_old directory under it so it can then be unmounted away.

Appears in: ch. 13

`put_old` *old-root parking directory*

The second argument to pivot_root: a directory beneath new_root where the kernel relocates the previous root mount, so the caller can detach it with umount and erase the host tree from view.

Appears in: ch. 13

`chroot escape` *chroot break-out*

The classic technique by which a process holding CAP_SYS_CHROOT leaves a chroot using a leaked directory file descriptor and fchdir, proving chroot alone is not a security boundary.

Appears in: ch. 13

`bind-on-itself` *self bind mount*

Bind mounting a directory onto its own path so it becomes a mount point, satisfying pivot_root's requirement that new_root be a mount point without altering the files inside it.

Appears in: ch. 13

`make-rprivate` *private mount propagation*

Setting a mount subtree's propagation to private (recursively) so mount and umount operations inside a namespace do not leak back to the host's mount table.

Appears in: ch. 13

`rootfs` *root filesystem*

The directory tree a container treats as /, supplied here by Alpine's minirootfs and installed as the real root with pivot_root.

Appears in: ch. 14

`veth pair` *virtual ethernet pair*

Two linked virtual interfaces acting as a cable: one end stays on the host, the other is moved into the container's network namespace to connect it to the outside.

Appears in: ch. 14

`MASQUERADE` *source NAT to internet*

An iptables target that rewrites the container subnet's source address to the host's, letting the container's traffic exit through the host's uplink and replies find their way back.

Appears in: ch. 14

`nsenter` *enter a namespace*

A tool that runs a command inside an existing process's namespaces, used here to configure the container's network namespace from the host by addressing it with the container's PID.

Appears in: ch. 14

`manifest` *image manifest*

A small JSON document (media type application/vnd.oci.image.manifest.v1+json) that lists one image's config blob and its ordered layer blobs, each by media type, size, and digest.

Appears in: ch. 15

`config` *image config*

The JSON blob (application/vnd.oci.image.config.v1+json) holding architecture/os, the ordered rootfs diff_ids, the build history, and runtime defaults (Env, Cmd, Entrypoint, WorkingDir, User, ExposedPorts, Volumes) — the source of docker inspect.

Appears in: ch. 15

`digest` *content digest*

The sha256 hash of an object's exact bytes, used as its immutable name; for a layer it is the hash of the compressed (gzipped) blob that is stored and transferred.

Appears in: ch. 15

`diff_id` *diff_id*

The sha256 of a layer's UNcompressed tar — the stable identity of its file contents, independent of gzip output — used to stack layers into the rootfs, as opposed to the digest of the compressed blob.

Appears in: ch. 15

---

image index *image index / manifest list*

A JSON document (application/vnd.oci.image.index.v1+json) mapping each platform (os/architecture) to a per-architecture manifest digest, so one tag can serve amd64, arm64, and more.

Appears in: ch. 15

OCI Distribution *OCI Distribution Spec*

The standardized registry HTTP API (from Docker Registry v2) for pulling images: GET /v2/<name>/manifests/<ref> for manifests and GET /v2/<name>/blobs/<digest> for blobs, authorized by a bearer token.

Appears in: ch. 15

---

dockerd *Docker daemon*

The high-level Docker engine (part of Moby) that serves the Engine API and manages images, builds, networks and volumes; it delegates actual container creation to containerd rather than doing it itself.

Appears in: ch. 16

---

containerd-shim *Per-container shim process*

One process per container (containerd-shim-runc-v2) that becomes the container's parent, holds its stdio and exit status, and is reparented to PID 1 so dockerd and containerd can be restarted without killing containers.

Appears in: ch. 16

OCI runtime bundle *rootfs + config.json*

A directory containing an unpacked root filesystem (rootfs/) and a config.json describing the OCI runtime spec; the input runc consumes. containerd converts an OCI image into such a bundle.

Appears in: ch. 16

---

config.json *OCI runtime-spec document*

The machine-readable declaration of a container's namespaces, mounts, cgroup limits, capabilities, seccomp profile and process args — the 'four restraints' of Part III written as data for runc to enact.

Appears in: ch. 16

CRI *Container Runtime Interface*

containerd's Kubernetes-facing gRPC API; it lets Kubernetes drive containerd directly, which is why the Docker shim (dockershim) was removed in Kubernetes 1.24 (2022).

Appears in: ch. 16

docker0 *default Docker bridge*

The Linux bridge the daemon creates on first start, carrying the 172.17.0.0/16 subnet and the 172.17.0.1 gateway; every container on the default bridge network gets a veth enslaved to it — the br0 of Chapter 8, automated.

Appears in: ch. 17

veth (Docker) *container veth pair*

For each container Docker makes a veth pair, renames one end eth0 inside the container's namespace, and enslaves the other (vethXXXX) to docker0; the host end shows up in `ip link` with an @ifN peer index.

Appears in: ch. 17

DNAT *destination NAT / port publishing*

The iptables rule behind `-p 8080:80`: traffic arriving on a host port has its destination rewritten to the container's IP and port, installed by the daemon in the nat table's DOCKER chain.

Appears in: ch. 17

docker-proxy *userland port proxy*

A small helper the daemon starts per published port; it listens on the host port as a fallback to the iptables DNAT fast path, notably handling hairpin connections where a container reaches its own published port.

Appears in: ch. 17

embedded DNS *127.0.0.11 resolver*

A DNS resolver Docker exposes inside containers on user-defined networks at 127.0.0.11, registering each container's name so peers resolve one another by name; absent on the default bridge network.

Appears in: ch. 17

`network driver` *Docker network driver*

The pluggable backend for a network: bridge (default docker0), host (shares the host namespace), none (loopback only), overlay (VXLAN across hosts), and macvlan/ipvlan (direct presence on the LAN).

Appears in: ch. 17

`container layer` *writable layer*

The single empty upperdir Docker adds atop an image's read-only layers at `docker run`; holds all runtime writes and is deleted with the container.

Appears in: ch. 18

`overlay2` *overlay2 storage driver*

Docker's default storage driver, built on the kernel's overlayfs, that stores image and container layers under /var/lib/docker/overlay2/, each as a directory with a diff/ holding its files.

Appears in: ch. 18

`volume` *docker volume*

Docker-managed storage at /var/lib/docker/volumes/<name>/_data, bind-mounted into the container; survives container removal and may use volume drivers (NFS, cloud). The preferred way to persist data.

Appears in: ch. 18

`tmpfs mount` *tmpfs mount*

A RAM-backed mount (--tmpfs) that never hits disk and disappears when the container stops; used for secrets and scratch data.

Appears in: ch. 18

`write amplification` *copy-up write amplification*

The cost of writing through an overlay: the first write to a lower-layer file copies the whole file up, making databases and other rewrite-heavy workloads belong on a volume.

Appears in: ch. 18

`docker run` *create + start*

The CLI command that, in one step, creates a container from an image and starts it (pulling the image first if needed); it POSTs to dockerd over the Engine API socket.

Appears in: ch. 19

Engine API *dockerd's HTTP API*

The REST interface the docker CLI uses to talk to dockerd over /var/run/docker.sock; you can drive it directly with curl --unix-socket.

Appears in: ch. 19

OCI bundle *rootfs + config.json*

The runtime bundle containerd assembles for runc: the merged root filesystem plus a config.json that declares the namespaces, cgroup limits, capabilities, seccomp profile, and process arguments.

Appears in: ch. 19

DNAT rule *port-publish translation*

The iptables nat-table entry Docker inserts for -p 8080:80, rewriting traffic on the host port to the container's address and port so the published port reaches the container.

Appears in: ch. 19