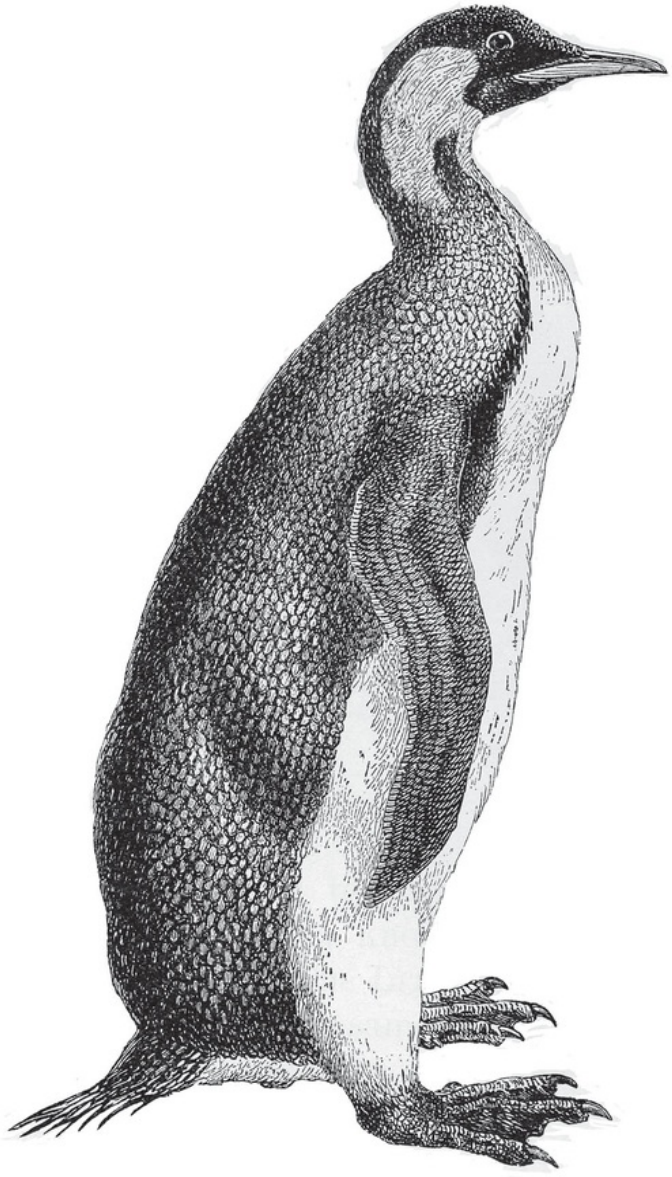
O'REILLY®

Learning Modern Linux

A Handbook for the Cloud Native Practitioner



Early Release

RAW & UNEDITED

Michael Hausenblas

Learning Modern Linux

A Handbook For The Cloud Native Practitioner

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Michael Hausenblas

Learning Modern Linux

by Michael Hausenblas

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Jeff Bleiel and John Devins

Production Editor: Beth Kelly

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

July 2022: First Edition

Revision History for the Early Release

- 2021-10-12: First Release

- 2021-12-14: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098108946 for release details.

978-1-098-10887-8

[LSI]

Chapter 1. Introduction to Linux

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at modern-linux@pm.me.

Linux is the most widely-used operating system, used in everything from mobile devices to the cloud.

You might not be familiar with the concept of an operating system. Or, you might be using an operating system such as Microsoft Windows without giving it too much thought. Or, maybe, you are new to Linux. To set the scene and get you in the right mindset, we'll take a birds-eye view of operating systems and Linux, in this chapter.

In this chapter we will first discuss what "Modern" means in the context of the book. Then we provide a high-level Linux backstory, looking at important events and phases over the past 30 years. Further, in this chapter you will learn what the role of an operating system is in general and how Linux goes about it, in particular. We have a quick look at what Linux distributions are and what resource visibility means.

If you're new to operating systems and Linux, then you want to read the whole chapter, if you're already experienced you might want to directly jump to "A 10,000ft View on Linux", providing a visual overview as well as mapping to book chapters.

But before we get into the technicalities, let's first step back a bit and focus on what we mean when we say "Modern" Linux. This is, surprisingly, a non-trivial matter.

What are Modern Environments?

The book title says "Modern" but what does that really mean? Well, in the context of this book, it can mean anything from cloud computing to a Raspberry Pi. In addition, the recent rise of Docker and related innovations in infrastructure has dramatically changed the landscape for developers and infrastructure operators alike.

Let us have a closer look at some of these modern environments, as well as the prominent role Linux plays in them.

Mobile Devices: Phones and Tablets

When I say "mobile phone" to our kids they go "in contrast to what"? In all fairness and seriousness many phones—depending on who you ask up to 80% or more—as well as tablets these days run Android, which is a Linux variant. These environments have agressive requirements around power consumption and robustness as we depend on them on a daily basis.

Cloud Computing

With the cloud we see similar pattern at scale as we see in the mobile and micro space. There are new, powerful, secure, and energy-saving CPU architecures such as the successful Arm-based AWS Graviton offerings as well as the established heavy-lifting outsourcing to cloud providers, especially in the context of open source software.

The Internet of (Smart) Things

I'm sure you've seen a lot of IoT related projects and products, from sensors to drones. Many of us have already been exposed to smart

appliances and smart cars. These environments have even more challenging requirements around power consumption that the mobile devices discusses earlier. In addition they might not even be running all the time but, for example, only wake up once a day to transmit some data. Another important aspect in these environments are real-time capabilities.

Diversity of Processor Architectures

For the past 30 years or so, Intel was the leading CPU manufacturer, dominating the micro computer and personal computer space. Intel's x86 architecture was considered the gold standard. The open approach that IBM took (publishing the specifications and enabling others to offer compatible devices) was was promising, resulting in the x86 clones, at least initially, also using Intel chips.

While Intel is still very widely used in desktop and laptop systems, with the raise of mobile devices we've seen the increasing uptake of the Arm architecture and recently also RISC-V. At the same time, multi-arch programming languaes and tooling, such as Go or Rust, are more and more widespread, creating a perfect storm.

All of these environments are examples of what I consider modern environments. And most if not all of them run on or use in one form or the other Linux.

Now that we know of all the modern (hardware) systems, you might wonder how we got here and how Linux came into being.

The Linux Story (So Far)

Linux celebrated its 30s birthday in 2021. With billions of users and thousands of developers, the Linux project is, without doubt, a world-wide (open source) success story. But how did it all this start and how did we get here?

1990s

We can consider Linus Torvalds' email on the 25th of August 1991 to the `comp.os.minix` newsgroup as the birth of the Linux project, at least the public subject record. This hobby project soon took off, both in terms of lines of code (LOC) and in terms of adoption. For example, after less than three years Linux 1.0.0 was released with over 176,000 LOCs. By then, the original goal of being able to run most Unix/GNU software was already well reached. Also, the first commercial offering appeared in the 90s, for example a distributor called Red Hat.

2000 to 2010

As a "teenager", Linux was not only maturing in terms of features and supported hardware but also growing beyond what Unix could do. In this time period we also witnessed a huge and ever-increasing buy-in of Linux by the big players, that is adoption by Google, Amazon, IBM, etc.. It was also the peak of the distro wars, resulting in business changing their directions.

2010s to now

Linux established itself as the work horse in data centers and the cloud, as well as for any types of IoT devices and phones. In a sense, one can consider the distro wars as over, and in a sense the raise of containers (from 2014/15 on) is responsible for it.

With this super quick historic review, necessary to set the context and understand the motivation for the scope of this book, we move on to a seemingly innocent question: Why does anyone need Linux, or an operating system at all?

Why an Operating System at All?

Let's say you do not have an operating system (OS) available or can not use one for whatever reason. You would then end up doing pretty much everything yourself: memory management, interrupt handling, talking with

I/O devices, managing files, configuring and managing the nework stack, the list goes on.

> **NOTE**
>
> Technically speaking an OS is not strictly needed. There are systems out there that do not have an OS. These are usually embedded systems with a tiny footprint, think of an IoT beacon. They simply do not have the resources available to keep anything else around other than one application. For example, with Rust you can use its Core and Standard Library to run any app on bare metal.

An operating system takes on all this undifferentiated heavy lifting, abstracting away the different hardware components and providing you with a (usually) clean and nicely designed Application Programming Interface (API), such as is the case with the Linux kernel that we will have a closer look at in Chapter 2. We usually call these APIs that an OS exposes *system calls* or syscalls for short. Higher level programming languages such as Go, Rust, Python, or Java, build on top of those syscalls, potentially wrapping them in libraries.

All of this allows you then to focus on the "business logic" rather than having to manage the resources yourself and also takes care of the different hardware you want to run your app on.

Let's have a look at a concrete example of a syscall. Let's say we want to identify (and print) the ID of the current user.

First, we look at the Linux syscall getuid(2):

```
...
getuid() returns the real user ID of the calling process.
...
```

OK, so this `getuid` syscall is what we could use programmatically, from a library. We will discuss Linux syscalls in greater detail in "Syscalls".

> **NOTE**
>
> You might be wondering what the (2) means in `getuid(2)`. It is a terminology that the `man` util (think: built-in help pages) uses to indicate the section of the command assgined in `man`, akin to a ZIP or country code. This is one example where the Unix legacy shows; you can find its origin in the UNIX PROGRAMMER'S MANUAL Seventh Edition, Volume 1 from 1979.

On the command line (shell) we would be using the equivalent `id` command that in turn uses the `getuid` syscall:

```
$ id --user
638114
```

Now that you have a basic idea of why an operating system, in most cases, makes sense, let's move on to the topic of Linux distributions.

Linux Distributions

When we say Linux, it might not be immediately clear what we mean. In this book we will say Linux kernel, or just a kernel, when we mean the set of syscalls and device drivers. Further when we refer to Linux distributions (or: distros, for short) we mean a concrete bundling of kernel, including package management, file system layout, init system, and a shell, pre-selected for you.

Of course, you could do all of this yourself: you could download and compile the kernel, choose a package manager, etc. and create (or: roll) your own distro. And that's what many folks did in the beginnging. Over the years, people figured out that it is maybe a better use of their time to leave this packaging (and also security patching) to experts, private or commercial alike, and simply use the resulting Linux distro.

> **TIP**
>
> If you are so inclined to build your own distribution, maybe because you are a tinkerer or you have to due to certain (business) restriction, I can recommend you to have a closer look at Arch Linux, which puts you in control and, with a little effort, allows you to create a very customized Linux distro.

To get a feeling for the vastness of the distro space, including traditional distros (Ubuntu, RHEL, Centos, etc. as discussed in XREF HERE) and modern distros (such as Bottlerocket and Flatcar, see XREF HERE), you can have a look at DistroWatch.

With the distro topic out of the way, let's move on to a totally different topic: resources, their visibility and isolation.

Resource Visibility

Linux has had, in good Unix tradition, a by-default global view on resources. This leads us to the question: what does global view mean (in contrast to what?) and what are said resources, really.

You might have heard about the saying that in Unix and by extension Linux everything is a file. In the context of this book we consider resoures to be anything which can be used to aid the execution of software. This includes hardware and its abstractions (such as CPU and RAM, files), filesystems, hard disk drives, SSDs, processes, networking-related stuff like devices or routing tables, and credentials representing users.

> **NOTE**
>
> While this is true for many things, in fact not all resources in Linux are files or represented through a file interface. However, there are systems out there, such as Plan 9, that take this much further.

Let's have a look at a concrete example of some Linux resources. First we want to query a global property (the Linux version) and then a specific hardware info about the CPUs in use (output edited to fit space):

```
$ cat /proc/version ❶
Linux version 5.4.0-81-generic (buildd@lgw01-amd64-051)
(gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04))
#91~18.04.1-Ubuntu SMP Fri Jul 23 13:36:29 UTC 2021


$ cat /proc/cpuinfo | grep "model name" ❷
model name      : Intel Core Processor (Haswell, no TSX, IBRS)
model name      : Intel Core Processor (Haswell, no TSX, IBRS)
model name      : Intel Core Processor (Haswell, no TSX, IBRS)
model name      : Intel Core Processor (Haswell, no TSX, IBRS)
```

❶ Print the Linux version.

❷ Print CPU related information, filtering for model.

With above commands we learned that this system has four Intel i7 cores at its disposal. When you log in with a different user, would you expect to see the same number of CPUs?

Let's consider a different type of resource: files. For example, if a user `troy` creates a file under `/tmp/myfile` and, permissions are allowing it ("Permissions"), and then user `worf` comes along, would they see the file or even be able to write to it?

Or, take the case of a process, that is, a program in memory that has all the necessary resources such as CPU and memory available to run. Linux identifies process using its process ID or `PID` for short ("Process Management"):

```
$ cat /proc/$$/status | head -n6 ❶
Name:   bash
Umask:  0002
State:  S (sleeping)
Tgid:   2056
```

```
Ngid:   0
Pid:    2056
```

❶ Print process status, that is, details about the current process and limit output to only show the first six lines.

WHAT IS $$?

You might have noticed the `$$` and wondered what this means. This is a special variable that is referring to the current process, see "Variables" for details on it.

Can there be multiple processes with the same `PID` in Linux? What may sound like a silly or useless question turns out to be the basis for containers (XREF HERE). The answer is yes, there can be multiple processes with the same `PID`. This happens in a containerized setup, for example, if you're running your app in Kubernetes.

Every single process might think that it is special, having the `PID 1`, which in a more traditional setup, is reserved for the root of the user space process tree, see also XREF HERE for more details.

What we can learn from these observations is that there can be a global view on a given resource (two users see a file at the exact same location) as well as a local or virtualized view, such as the process example. This begs the question: is everything in Linux by default global? Spoiler: it's not. Let's have a closer look.

Part of the illusion of having multiple users or process running in parallel is the (restricted) visibility onto resources. The way to provide a local view on (certain, supported) resources in Linux is via namespaces (XREF HERE).

A second, independent dimension is that of isolation. When I use the term isolation here I don't necessarily qualify it, that is, I make no assumptions about how well things are isolated. For example, one way to think about process isolation is to restrict the memory consumption so that one process

can not starve other processes. For example, I give your app 1 GB to RAM to use. If it uses more, it gets OOM killed. This provides a certainly level of protection. In Linux we use a kernel feature called `cgroups` to provide this kind of isolation, and in XREF HERE you will learn more about it.

On the other hand, a fully isolated environment gives the appearance that the app is entirely on its own. For example a virtual machine (VM), see also XREF HERE provides you with a full isolation.

A 10,000ft View on Linux

Woah, we went quite deep into the weeds already. Time to take a deep breath and re-focus. In the Figure 1-1 I've tried to provide you with a high-level overview on the Linux operating system, mapping it to the book chapters.

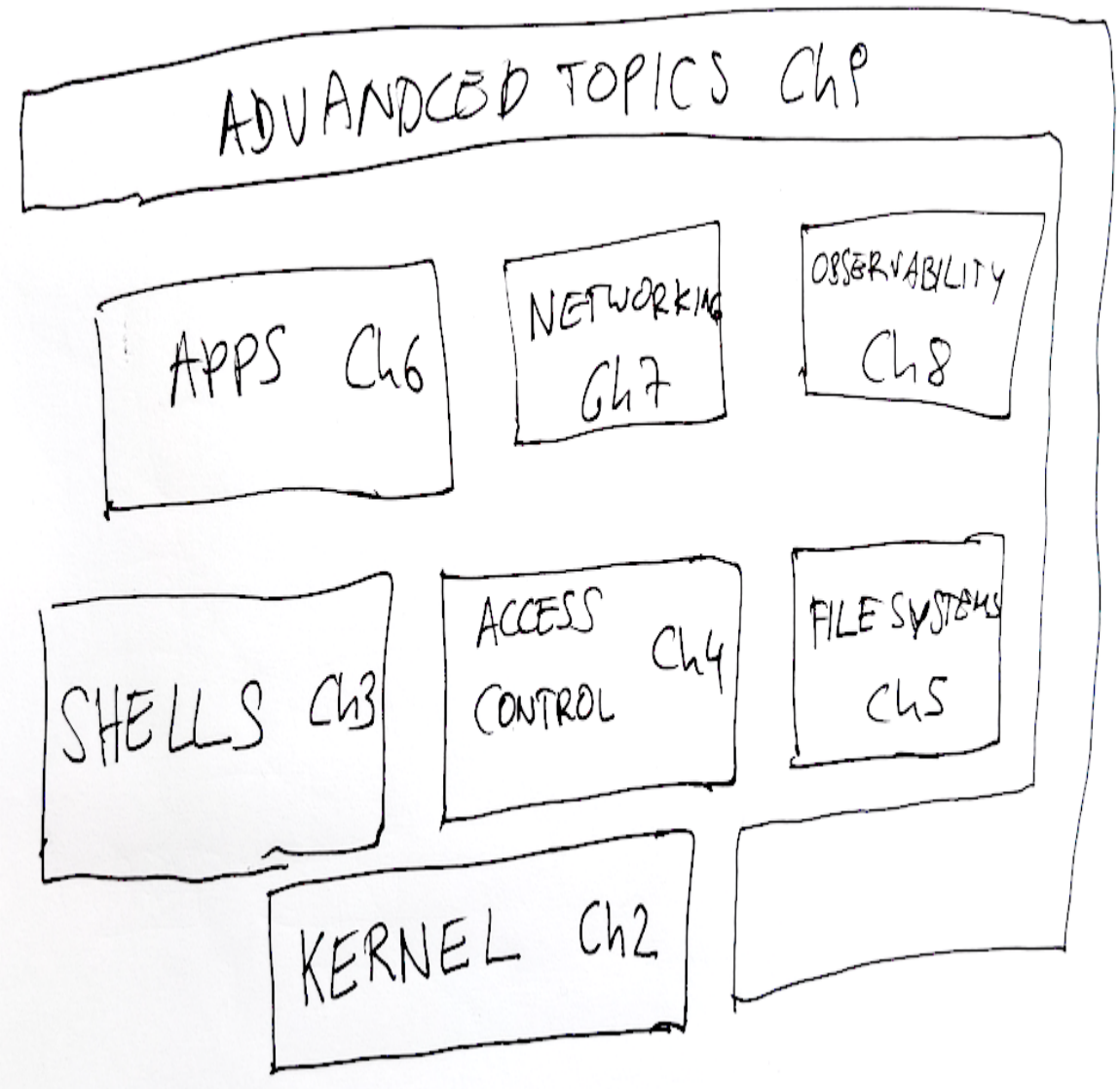


Figure 1-1. Mapping the Linux operating system to book chapters

At its core, any Linux distro has the kernel, providing the API everything else builds on. The three core topics of files, networking, and observability follow you everywhere and you can consider them as the most basic building blocks above the kernel. From a pure usage perspective, you will soon learn that the shell (where is the output file for this app?) and access control related things (why does this app crash, ah the directory is read-only, doh) are what you will be dealing with, most often.

PORTABLE OPERATING SYSTEM INTERFACE (POSIX)

We will come across the term POSIX, short for Portable Operating System Interface, every now and then in this book. Formally, POSIX is a IEEE standard to define service interfaces for UNIX operating systems. The motivation was to provide portability between different implementations. So, if you read things like POSIX-compliant think of a set of formal specifications that are especially relevant in official procurement context and less so for everyday usage.

If you want to learn more about it, check out POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing that provides a great introduction and comments on uptake and challenges around this topic.

As an aside: I've lumped up some interesting topics, from virtual machines to modern ditros in XREF HERE. I called it advanced mainly because I consider these topics as optional. That is, you could get away without learning them. Although if you really, really, really want to benefit from the full power that modern Linux can provide you, then I strongly recommend that you also this chapter. I suppose it goes without saying that, by design, the rest of the book—that is Chapter 2 to XREF HERE—are essential chapters you should most definitely study and apply the content as you go.

Conclusion

In this first chapter explained what exactly we mean when we call something "Modern". We talked about what modern environments are. This includes for example phones, the data centers of public cloud providers, as well as embedded systems such as a Raspberry Pi.

We also gave a high-level Linux backstory in this chapter, and discussed the role of an operating system in general—to abstract the underlying hardware and provide a set of basic functions such as process, memory, file, and

network management to applications—and also, how Linux specifically goes about this task, specifically about visibility of resources.

Some great books to read for you to get up to speed as well as to dive deeper concerning concepts discussed in this chapter are the listed in the following.

O'Reilly titles:

- Linux Cookbook by Carla Schroder
- Understanding the Linux Kernel by Daniel P. Bovet and Marco Cesati
- Linux Pocket Guide by Daniel J. Barrett
- Linux System Programming by Robert Love

Other resources:

- Advanced Programming in the UNIX Environment is a complete course that offers a lot introductory material incl. hands-on exercises.
- The Birth of UNIX with Brian Kernighan is a great source for Linux' legacy and provides context for a lot of the original Unix concepts.

And now, without further ado: let's start our journey into modern Linux with the core, erm, kernel of the matter!

Chapter 2. The Linux Kernel

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 2nd chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at modern-linux@pm.me.

In "Why an Operating System at All?" we said that the main function of an operating system is to abstract over different hardware and provide us with an Application Programming Interface (API). Programming against this API allows us to write applications without having to worry about where and how they are executed. In a nutshell, the kernel provides such an API to programs.

In this chapter we discuss what the Linux kernel is and how you should be thinking about it as a whole as well as about its components. You will learn about the overall Linux architecture and the essential role the Linux kernel plays. One main takeawy of this chapter should be that while the kernel provides all the core functionality, on its own it is not the operating system but only a, admittedly very central, part of it.

First, we take a birds eye view in "Linux Architecture", looking at how the kernel fits in and interacts with the underlying hardware. Then, in "CPU Architectures" we review the computational core, discussing different CPU architectures and how they relate to the kernel. Next we zoom in on the individual kernel components in "Kernel Components", and discuss the API

the kernel provides to programs you can run. Finally, we look at how to customize and extend the Linux kernel in "Kernel Extensions".

The purpose of this chapter is to equip you with the necessary terminology, make you aware of the interfacing between programs and the kernel, and give you a basic idea what the functionality is. The chapter does not aim to turn you into a kernel developer or even a sysadmin configuring and compiling kernels. If you, however, want to dive into that, I've put together some pointers at the end.

And now, let's jump into the deep end: the Linux architecture, and the central role the kernel plays in this context.

Linux Architecture

On a high level the Linux architecture looks as depicted in Figure 2-1. There are three distinct layers you can group things into:

- At the bottom you find the hardware: from CPUs to main memory to disk drives, network interfaces and peripheral devices such as keyboards and monitors.
- The kernel itself, the focus of the rest of this chapter.
- The user land: where the majority of apps are running, including operating system compnents such as shells (discsussed in Chapter 3), utilities like `ps` or `ssh`, as well as graphical user interfaces such as X Window System-based desktops.

We focus in this book is on the upper two layers of Figure 2-1, that is, the kernel and user land. The hardware layer on the other hand is something we only touch on in this and few other chapters, where relevant.

The interfaces between the different layers are well defined and part of the Linux operating system package. Between the kernel and user land the interface is called system calls (syscalls for short) and we will explore this in detail in "Syscalls".

The interface between the hardware and the kernel is, unlike the syscalls, not a single one. It consists of a collection of individual interfaces, usually grouped by hardware:

1. This interface is represented by CPU architecture specific code, see "CPU Architectures".
2. The interface with the main memory, covered in "Memory Management".
3. Network interfaces and drivers (wired and wireless), see also "Networking".
4. Filesystem and block devices driver interfaces, see "Filesystems".
5. Character devices, hardware interrupts, and device drivers, for input devices like keyboards, terminals and other I/O ("Device Drivers").

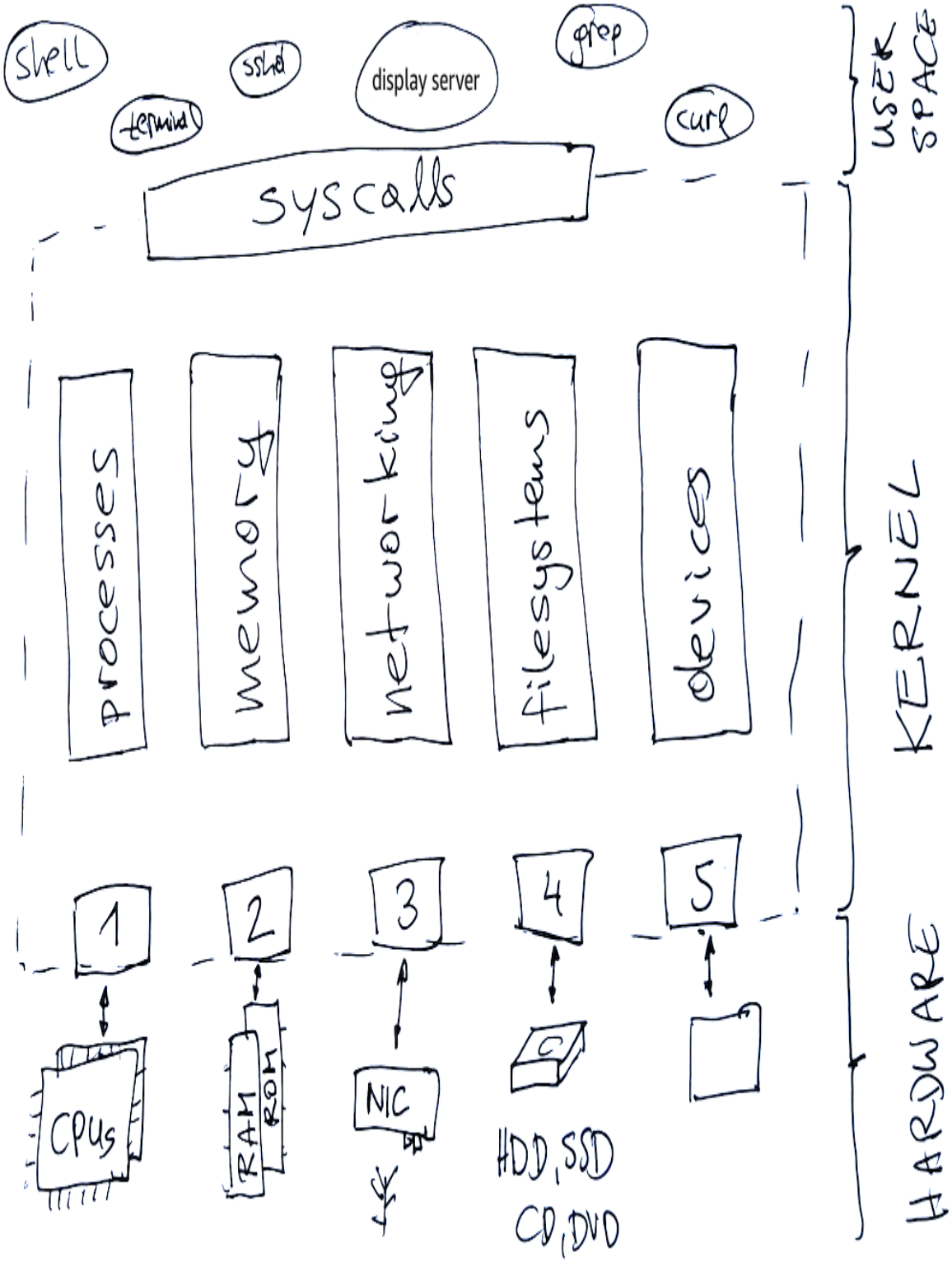


Figure 2-1. A high-level view on the Linux architecture

As you can see, many of the things we usually consider part of the Linux operating system such as shell or utilities such as `grep`, `find`, and `ping`

are, in fact, not part of the kernel but, very much like an app you download, part of user land.

On the topic of "user land": you will often read or hear about user vs. kernel mode. This effectively means how privileged the access to hardware is and how restricted the abstractions available are.

In general, kernel mode means fast execution with limited abstraction while user level land mode means comparatively slower but safer and mor convenient abstractions. Unless you are a kernel developer, you can almost always ignore kernel mode, since all your apps will run in user land. Knowing how to interact with the kernel ("Syscalls") on the other hand, is vital and part of our considerations.

With this Linux architecture overview out of the way, let's work our way up from the hardware.

CPU Architectures

Before we discuss the kernel components let's review a basic concept first: computer architectures or CPU families, which we will use interchangeably. The fact that Linux runs on a large number of different CPU architectures is arguably one of the reasons it is so popular.

Next to generic code and drivers, the Linux kernel contains architecture-specific code. This separation allows it to port Linux and make it available on new hardware, quickly.

There are a number of ways to figure out what CPU your Linux is running, let's have a look at a few in turn.

THE BIOS AND UEFI

Traditionally, Unix and Linux used the Basic I/O System (BIOS) for bootstrapping itself. When you power on your Linux laptop, it is entirely hardware-controlled. First off, the hardware is wired to run the Power On Self Test (POST), part of the BIOS. POST makes sure that the hardware (RAM, etc.) function as specified. We will get into the details of the mechanics in XREF HERE.

In modern environments the BIOS functions have been effectively replaced by the Unified Extensible Firmware Interface (UEFI), a public specification that defines a software interface between an operating system and platform firmware. You will still come across the term BIOS a lot in documentation and articles, so I suggest you simply replace it with UEFI in your head and move on.

One way is a dedicated tool that interacts with the BIOS called `dmidecode` and if this doesn't yield results you could try (output shortened):

```
$ lscpu
Architecture:                    x86_64 ❶
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   40 bits physical, 48 bits virtual
CPU(s):                          4 ❷
On-line CPU(s) list:             0-3
Thread(s) per core:              1
Core(s) per socket:              4
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           60
Model name:                      Intel Core Processor (Haswell, no
TSX, IBRS) ❸
Stepping:                        1
CPU MHz:                         2592.094
...
```

❶ The architecture we're looking at here is `x86_64`.

❷ It looks like there are four CPUs available.

❸ The CPU model name is Intel Core Processor (Haswell).

In the previous command we saw that the CPU architecture was reported to be `x86_64` as well as the model was reported as "Intel Core Processor (Haswell)". We will learn more about how to decode this in a moment.

Another way to glean similar architecture information is using `cat /proc/cpuinfo` or, if you're really only interested in the architecture, simply call `uname -m`.

Now that we have a handle on querying the architecture information on Linux, let's see how to decode it.

x86 Architecture

x86 is an instruction set family originally developed by Intel and later licensed to AMD. Within the kernel, when you see `x64` that refers to the Intel 64 bit processors as well as `x86` stands for Intel 32 bit. Further, when you come across `amd64` that refers to AMD 64 bit processors.

Even nowadays, you find the x86 CPU family mostly in desktops and laptops, but also widely used in servers. Specifically, `x86` forms the basis of the public cloud. It is a powerful and widely available architecture, however it is not very energy efficien. Partially due to its heavy reliance on out-of-order execution, it recently received a lot of attention around security issues such as Meltdown.

For further details, for example the Linux/x86 boot protocol or Intel and AMD specific background, see the x86 specific kernel documentation.

Arm Architecture

Arm is a more than 30 years old family of Reduced Instruction Set Computing (RISC) architectures. RISC usually consists of a large numbe of generic CPU registers along with a small set of instructions that can be executed faster.

Because the designers at Acorn—the original company behind Arm—focused from the get-go on minimal power consumption you find Arm-based chips in a number of portable devices such as iPhones. They are also in most Android-based phones, and in embedded systems found in IoT such as in the Raspberry Pi.

Given that Arm-based CPUs are fast, cheap, and produce—compared to x86 chips—less heat, you shouldn't be suprised to increasingly also find them in the datacenter, for example, AWS Graviton. While simpler than x86, Arm is not immune to vulnerabilities, for example, Spectre.

For further details, see the Arm specific kernel documentation.

RISC-V Architecture

An up-and-coming player, RISC-V (pronounce: risk five) is an open RISC standard, that was originally developed by the University of California, Berkeley (UCB). As of 2021, a number of implementations exist, ranging from Alibaba Group and Nvidia to a range of startups such as SiFive. While exciting, this is a relatively new and not widely used (yet) CPU family, and to get an idea how it look and feels, you may want to research it a little (a good start is Shae Erisson's article Linux on RISC-V).

For further details, see the RISC-V specific kernel documentation.

Now that you know the basics about CPU architectures let's move on to the kernel components.

Kernel Components

Now that you have an idea what the core compute unit, the CPU architectures, mean it's time to dive into the kernel. While the Linux kernel

is a monolithic one—that is, all the components discussed are part of a single binary—there are functional areas in the code base that we can identify and ascribe dedicated responsibilities.

As we've discussed in "Linux Architecture", the kernel sits between the hardware and the apps you want to run. The main functional blocks you find in the kernel code base are as follows:

- Process management such as starting a process based on an executable file, as discussed in "Process Management".
- Memory management, for example, allocating memory for a process, which we will review in "Memory Management".
- Networking, like managing network interfaces or providing the TCP/IP stack, as described in "Networking".
- Filesystems as you can read up in "Filesystems", supporting the creation and deleting of files, for example.
- Management of character devices and device drivers as per "Device Drivers".

These functional components oftentimes come with interdependencies and its a truly challenging task to make sure that the kernel developer motto "kernel never breaks user land" holds true.

With that, let's have a closer look at the kernel components.

Process Management

There are a number of process management related parts in the kernel. Some of them deal with CPU architecture specific things, such as interrupts and others focus on the launching and scheduling of programs.

Before we get to Linux specifics, let's note that commonly, a process is the user-facing unit, based on an executable programm (or binary). A thread, on the other hand is a unit of execution in the context of a process. You might have come across the term multi-threading, this means that a process has a

number of parallel executions going on, potentially running on different CPUs.

With the general view out of the way, let's see how Linux goes about it. From most granular to smallest unit, Linux has:

Sessions

> Contains one or more a process groups, it represents a high-level user-facing unit with optional `tty` attached. The kernel identifies a session via a number that is called session ID (SID).

Process groups

> Contains one or more processes, withat most one process group in a session being the foreground process group. The kernel identifies a process group via a number that is called process group ID (PGID).

Processes

> A process is an abstraction that groups multiple resources (address space, one or more threads, sockets, etc.), which the kernel exposes to you via `/proc/self` for the current process. The kernel identifies a process via a number that is called process ID (PID).

Threads

> The kernel implements threads as processes. That is, there are no dedicated data structures representing threads. Rather, a thread is a process that shares certain resources (such as memory or signal handlers) with other processes. The kernel identifies a thread via thread IDs (TID), thread group IDs (TGID), with the semantics that a shared TGID value means a multi-threaded process (in user land; there are also kernel threads but that's beyond our scope).

Tasks

> In the kernel there is a data structure called `task_struct` (defined in sched.h) which forms the basis of implementing processes and threads

alike. This data structure captures scheduling related information (see below), identifiers (such as PID and TGID), signal handlers, as well other information including performance and security related ones. In a nutshell, all of the above units are derived and/or anchored in tasks, however, tasks are not exposed as such outside of the kernel.

We will see sessions, process groups, and processes in action and how to manage them in XREF HERE as well as further on, in the context of containers, in XREF HERE.

Let's see some of the above terms in action:

```
$ ps -j
PID    PGID   SID   TTY      TIME CMD
6756   6756   6756  pts/0    00:00:00 bash ❶
6790   6790   6756  pts/0    00:00:00 ps ❷
```

❶ The `bash` shell process has PID, PGID, and SID of 6756. From `ls -al /proc/6756/task/6756/` we can glean the task-level information.

❷ The `ps` process has PID/PGID 6790 and the same SID as the shell.

We mentioned earlier on that in Linux the task data structure has some scheduling related information at the ready. This means that a process at any given time is in a certain state as shown in Figure 2-2.

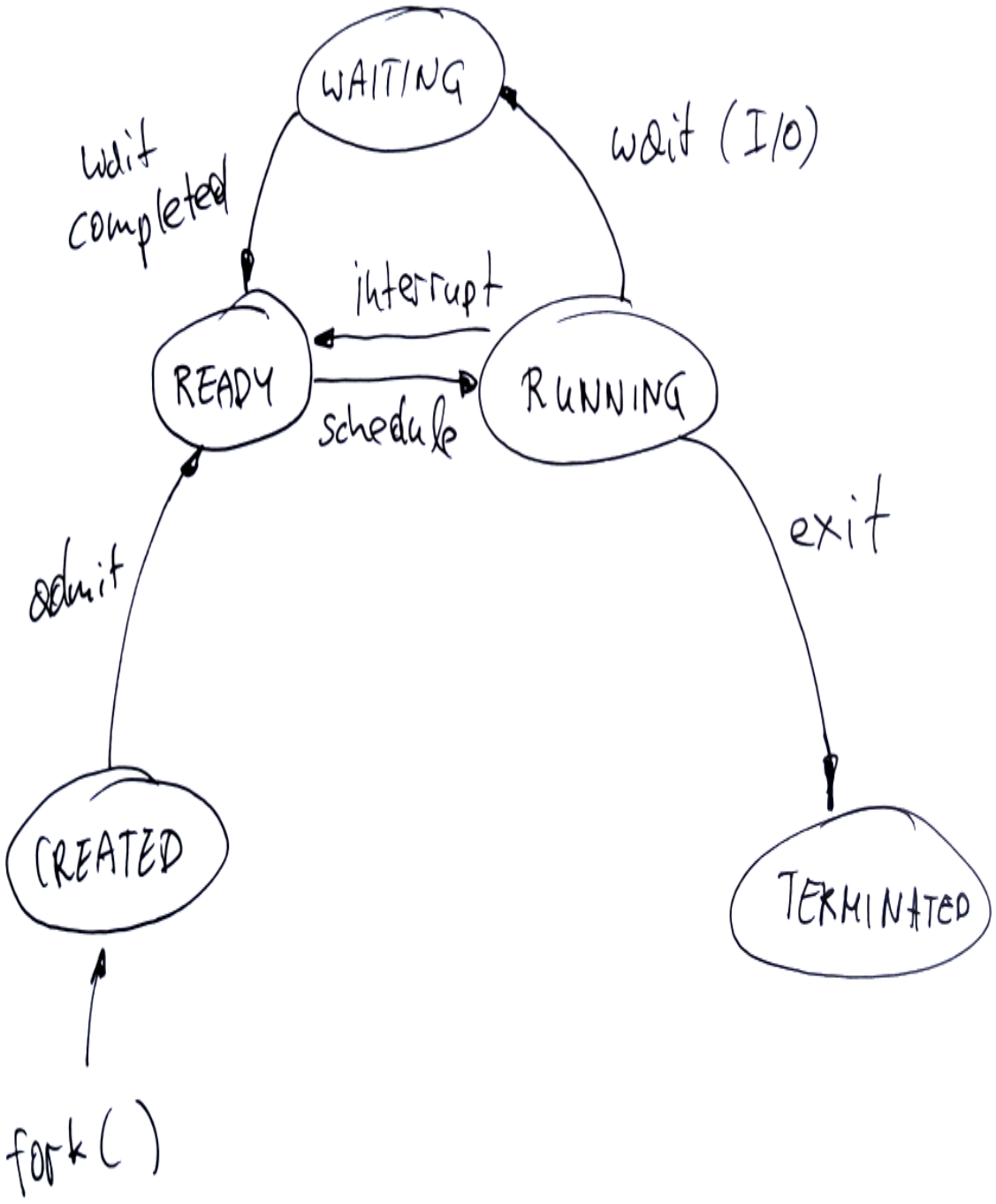


Figure 2-2. Linux process states

Different events cause state transitions. For example, a running process might transition to the waiting state when it carries out some I/O operation (such as reading from a file) and can't proceed with execution (off CPU).

After this short part on process managment, let’s have a closer look at a very related topic: memory.

Memory Management

Virtual memory makes your system appear as if it has more memory than it physically has. In fact, every process gets a lot of (virtual) memory. The way it works is as follows: both physical and virtual memory is divided into fixed length chunks we call pages.

Figure 2-3 shows the virtual address spaces of two processes, each with their own page tables. These page tables map virtual pages of the process into physical pages in main memory (aka RAM).

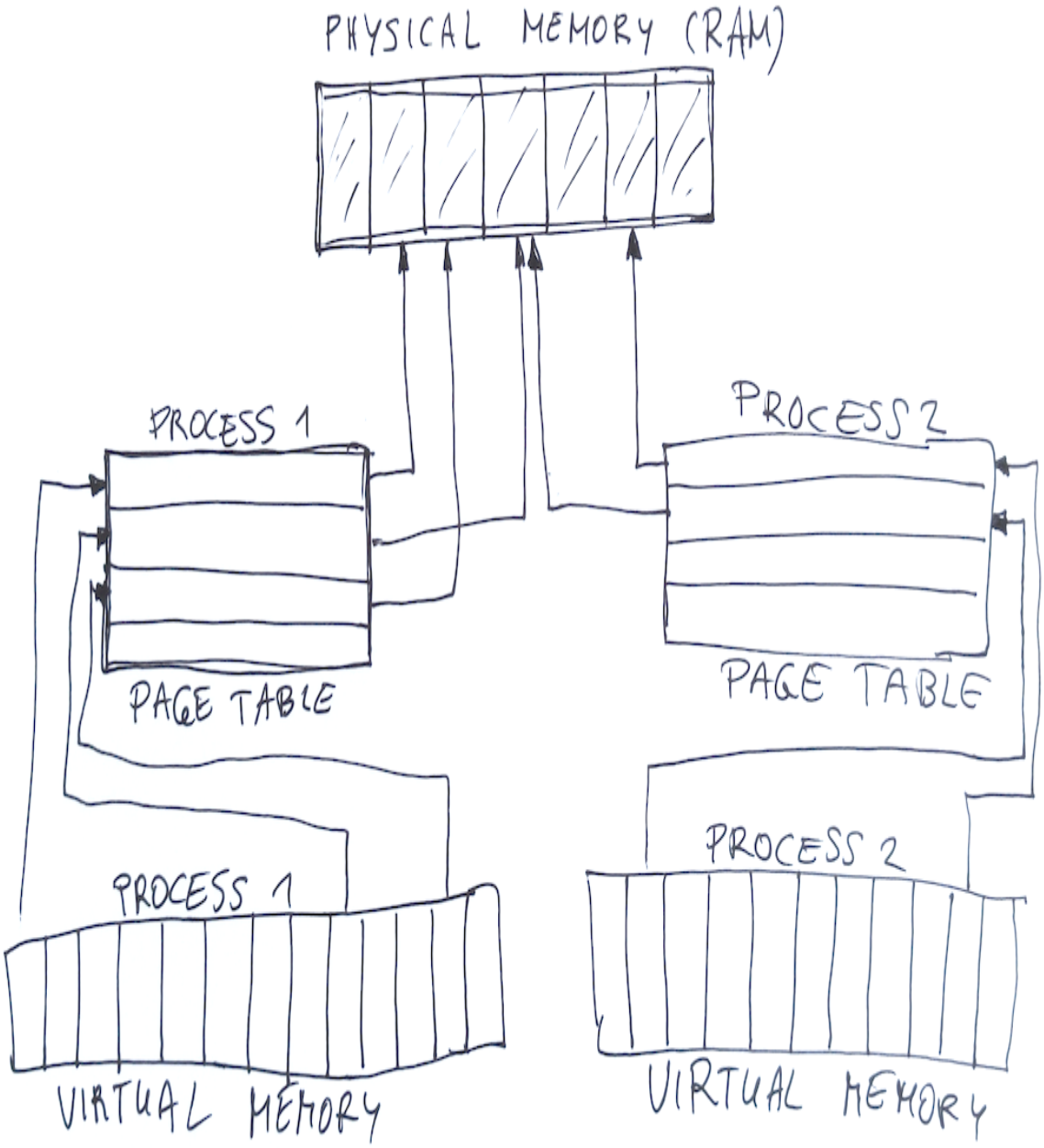


Figure 2-3. Virtual memory management overview

Multiple virtual pages can point to the same physical page via their respective process-level page tables. This is in a sense the core of memory management: how to effectively provide each process with the illusion that their page actually exists in RAM while using the existing space optimally.

Every time the CPU accesses a process' virtual page, the CPU would in principle have to translate the virtual address a process uses to the

corresponding physical address. To speed up this process—that can be multi-level and hence slow—modern CPU architectures support a lookup on-chip called translation lookaside buffer (TLB). The TLB is effectively a small cache that, in case of a miss, causes the CPU to go via the process page table(s) to calculate the physical address of a page and update the TLB with it.

Traditionally, Linux had a default page size of 4 KB but since kernel v2.6.3 it supports huge pages, to better support modern architectures and workloads. For example, 64-bit Linux allows you to use up to 128 TB of virtual address space per processes, with an approximate 64 TB of physical memory in total.

Ok, that was a lot of information and also on the theoretical side, let's have a look at it from a more practical point of view. A very useful tool to figure out memory-related information such as how much RAM is available to you is the `/proc/meminfo` interface:

```
$ grep MemTotal /proc/meminfo ❶
MemTotal:        4014636 kB

$ grep VmallocTotal /proc/meminfo ❷
VmallocTotal:   34359738367 kB

$ grep Huge /proc/meminfo ❸
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:               0 kB
```

❶ List details on physical memory (RAM); that's 4GB there.

❷ List details on virtual memory; that's a bit more than 34 TB there.

❸ List huge pages info; apparently here the page size is 2MB.

With that, we move on to the next kernel function: networking.

Networking

One important function of the kernel is to provide networking functionality. No matter if you want to browse the Web, or if you want to copy data to a remote system, you depend on the network.

The Linux network stack follows a layered architecture:

- The sockets, which abstract communication.
- The Transmission Control Protocol (TCP) for connection-oriented communication as well as User Datagram Protocol (UDP) for connection-less communication.
- The Internet Protocol (IP) for adressing machines.

These three actions are all that the kernel takes care of. The application layer protocols such as HTTP or SSH are, usually, implemented in user land.

You can get an overview of your network interfaces using (output edited):

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN mode
   DEFAULT group default qlen 1000 link/loopback
00:00:00:00:00:00
   brd 00:00:00:00:00:00
2: enp0s1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
fq_codel state
   UP mode DEFAULT group default qlen 1000 link/ether
52:54:00:12:34:56
   brd ff:ff:ff:ff:ff:ff
```

Further, `ip route` provides you with routing information. Since we have a dedicated networking chapter (XREF HERE) where we will dive deep into the networking stack, the supported protocols, and typical operations,

we keep it at this and move on to the next kernel component, block devices and filesystems.

Filesystems

Linux uses fileystems to organize files and directories on storage devices such as hard disk drives (HDDs) and solid-state drives (SSD)s or flash memory. There are many types of filesystems such as `ext4` and `btrfs` or NTFS and you can have multiple instances of the same filesystem in use.

Virtual File System (VFS) was originally introduced to support multiple filesystem types and instances. The highest layer in VFS provides a common API abstraction of functions such as open, close, read, and write. At the bottom of VFS are filesystem abstractions called plug-ins for the given filesystem.

We will go into greater detail concerning filesystems and file operations in Chapter 5.

Device Drivers

A driver is a bit of code that runs in the kernel to manage a device, which can be actual hardware—like a keyboard or disk drives—or it can be a pseudo device. Another interesting class of hardware are Graphics Processing Unit (GPU) which traditionally were used to accelerate graphics output and with it ease the load on the CPU. In the past years, GPUs have found a new use case in the context of machine learning and hence they are not exclusively relevant in desktop environments.

The driver may be built statically into the kernel or it can be built as a kernel module (“Modules”) so that it can be dynamically loaded when needed.

> **TIP**
>
> If you're interested in an interactive way to explore device drivers and how kernel components interact, check out the Linux kernel map.

The kernel driver model is complicated and out of scope for this book. However, a few hints how to interact with it in the following, just enough so that you know where to find what.

To get an overview on the devices on your Linux system, you can use:

```
$ ls -al /sys/devices/
total 0
drwxr-xr-x 15 root root 0 Aug 17 15:53 .
dr-xr-xr-x 13 root root 0 Aug 17 15:53 ..
drwxr-xr-x  6 root root 0 Aug 17 15:53 LNXSYSTM:00
drwxr-xr-x  3 root root 0 Aug 17 15:53 breakpoint
drwxr-xr-x  3 root root 0 Aug 17 17:41 isa
drwxr-xr-x  4 root root 0 Aug 17 15:53 kprobe
drwxr-xr-x  5 root root 0 Aug 17 15:53 msr
drwxr-xr-x 15 root root 0 Aug 17 15:53 pci0000:00
drwxr-xr-x 14 root root 0 Aug 17 15:53 platform
drwxr-xr-x  8 root root 0 Aug 17 15:53 pnp0
drwxr-xr-x  3 root root 0 Aug 17 15:53 software
drwxr-xr-x 10 root root 0 Aug 17 15:53 system
drwxr-xr-x  3 root root 0 Aug 17 15:53 tracepoint
drwxr-xr-x  4 root root 0 Aug 17 15:53 uprobe
drwxr-xr-x 18 root root 0 Aug 17 15:53 virtual
```

Further, you can use the following to list mounted devices:

```
$ mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
devpts on /dev/pts type devpts
(rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000)
...
tmpfs on /run/snapd/ns type tmpfs
(rw,nosuid,nodev,noexec,relatime,size=401464k,mode=755,inode64)
nsfs on /run/snapd/ns/lxd.mnt type nsfs (rw)
```

And with this we have covered the Linux kernel components and move to the interface between the kernel and user land.

Syscalls

Whether you sit in front of a terminal and type `touch test.txt` or wheter one of your apps wants to download the content of a file from a remote system, at the end of the day, you ask Linux to turn the high-level instruction such as “create a file” or “read all bytes from address so and so” into a set of concrete, architecture-dependent steps. In other words, the service interface the kernel exposes and that user land entities call is the set of system calls or syscalls for short.

Linux has hundreds of syscalls, 300+ depending on the CPU family, available. However you and your programs don’t usually invoke these syscalls directly but via what we call the C standard library. The standard library provides wrapper functions and is available in various implementations such as glibc or musl.

These wrapper libraries perform an important task. They take care of the repetitive low-level handling of the execution of a syscall. As system calls are implemented as software interrupts, causing an exception that transfers the control to an exception handler. There are a number of steps to take care of every time a syscall is invoked, as depicted in Figure 2-4:

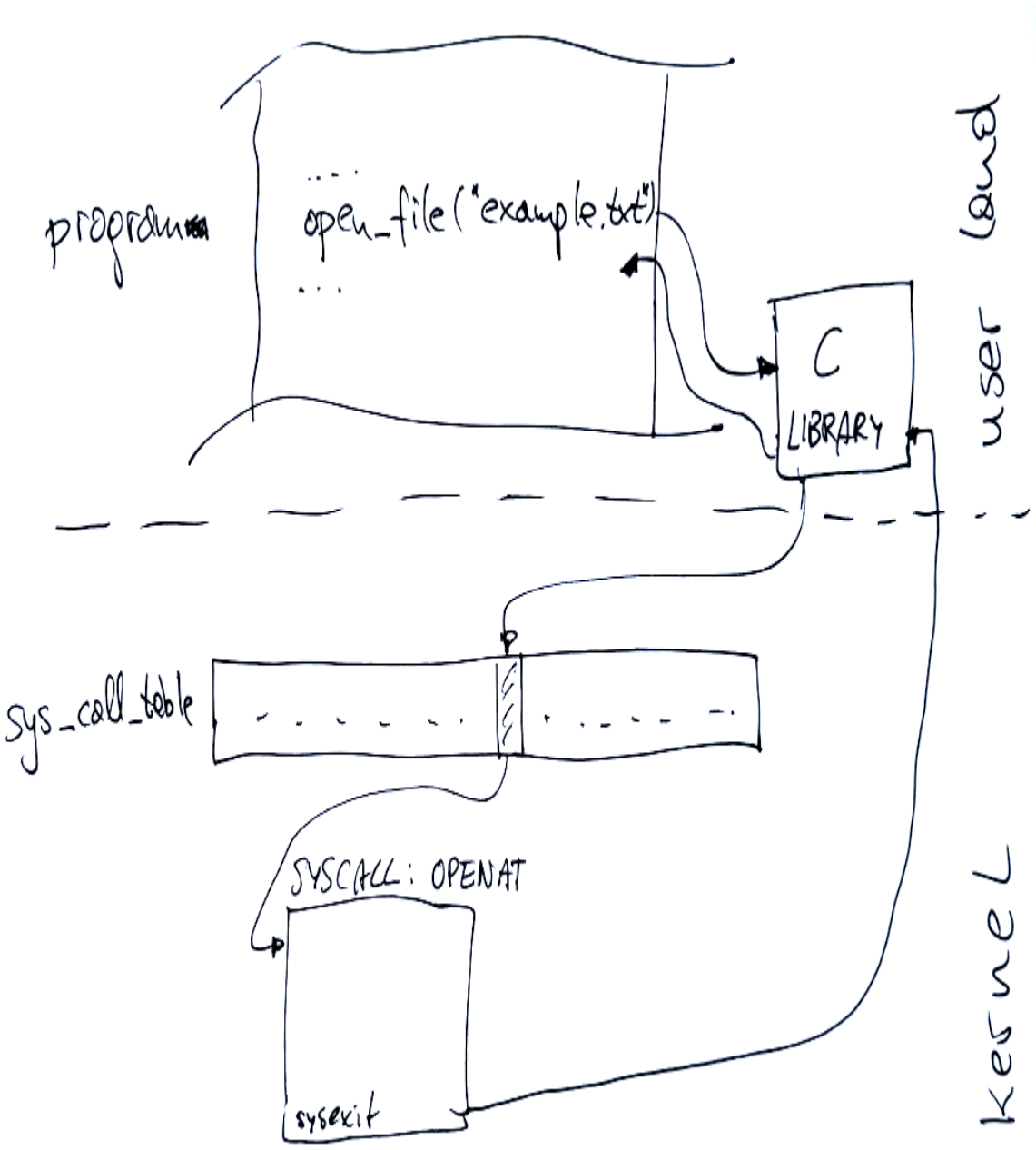


Figure 2-4. Syscall execution steps in Linux

- Defined in `syscall.h` and architecture-dependent files, the kernel uses a so called syscall table, effectively an array of function pointers in memory (storedin a variable called `sys_call_table`) to keep track of syscalls and their corresponding handlers.

- With the `system_call()` function acting like a syscall multiplexer it first saves the hardware context on the stack, then performs checks (like if tracing is performed), and then jumps to the function pointed to by the the respective syscall number index in the `sys_call_table`.
- After the syscall completed with `sysexit`, the wrapper library restores the hardware context and the programm execution resumes in user land.

Notably in the previous steps is the switching between kernel mode and user land mode, an operation that costs time.

OK, that was a little dry and theoretical, so to better appreciate how syscalls look and feel in practice let's have a look at a concrete example. We will use strace to look behind the curtain, a tool useful for troubleshooting, for example, if you don't have the source code of an app but want to learn what it does.

Let's assume you wonder what syscalls are involved when you execute the innocent looking `ls` command. Here's how you can find it out using `strace`:

```
$ strace ls ❶
execve("/usr/bin/ls", ["ls"], 0x7ffe29254910 /* 24 vars */) = 0 ❷
brk(NULL)                               = 0x5596e5a3c000 ❸
...
access("/etc/ld.so.preload", R_OK)  = -1 ENOENT (No such file or
directory) ❹
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3 ❺
...
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0
p\0\0\0\0\0\0"..., 832) = 832 ❻
...
```

❶ With `strace ls` we ask `strace` to capture syscall that `ls` uses. Note that I edited the output since `strace` generates some 162 lines on my system (this number varies between different distros, architectures, and other factors). Further, the output you see there comes via `stderr`,

so if you want to redirect it you have to use `2>` here. Learn more about this in Chapter 3.

❷ The syscall execve executes `/usr/bin/ls`, causing the shell process to be replaced.

❸ The brk syscall is an outdated way to allocate memory, it's safer and more portable to use `malloc`. Note that `malloc` is not a syscall, but a function that in turn uses `mallocopt` to decide if it needs to use the `brk` syscall or the `mmap` syscall based on the amount of memory accessed.

❹ The `access` syscall checks if the process is allowed to access a certain file.

❺ Syscall `openat` opens the file `/etc/ld.so.cache` relative to a directory file descriptor (here the 1st argument, `AT_FDCWD` which stands for the current directory) and using flags `O_RDONLY|O_CLOEXEC` (last argument).

❻ The `read` syscall reads from a file descriptor (1st argument, `3`) 832 bytes (last argument) into a buffer (2nd argument).

`strace` is useful to see exactly what syscalls have been called—in which order and with which arguments—effectively hooking into the live stream of events between user land and kernel. It's also good for performance diagnostics. Let's see where a `curl` command spends most of its time (output shortened):

```
$ strace -c \ ❶
        curl -s https://mhausenblas.info > /dev/null ❷
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
--
 26.75    0.031965         148       215           mmap
 17.52    0.020935         136       153         3 read
 10.15    0.012124         175        69           rt_sigaction
```

```
  8.00    0.009561         147        65         1 openat
  7.61    0.009098         126        72           close
  ...
  0.00    0.000000           0         1           prlimit64
------ ----------- ----------- --------- --------- ----------------
--
100.00    0.119476         141       843        11 total
```

❶ Use the `-c` option to generate overview stats of the syscalls used.

❷ Discard all output of `curl`.

Interesting: the `curl` command here spends almost half of its time with `mmap` and `read` syscalls and the `connect` syscall takes 0.3 ms, not bad.

To get a feeling for the coverage, I've put together Table 2-1 which lists examples of widely used syscalls across kernel components as well as system wide ones. You can look up details of sycalls, including their parameters and return values, via the section 2 of the man pages.

Table 2-1. Example system calls

Category	Example syscalls
Process management	`clone`, `fork`, `execve`, `wait`, `exit`, `getpid`, `setuid`, `setns`, `getrusage`, `capset`, `ptrace`

Memory management	`brk`, `mmap`, `munmap`, `mremap`, `mlock`, `mincore`
Networking	`socket`, `setsockop`, `getsockopt`, `bind`, `listen`, `accept`, `connect`, `shutdown`, `recvfrom`, `recvmsg`, `sendto`, `sethostname`, `bpf`
Filesystems	`open`, `openat`, `close`, `mknod`, `rename`, `truncate`, `mkdir`, `rmdir`, `getcwd`, `chdir`, `chroot`, `getdents`, `link`, `symlink`, `unlink`, `umask`, `stat`, `chmod`, `utime`, `access`, `ioctl`, `flock`, `read`, `write`, `lseek`, `sync`, `select`, `poll`, `mount`,
Time	`time`, `clock_settime`, `timer_create`, `alarm`, `nanosleep`
Signals	`kill`, `pause`, `signalfd`, `eventfd`,
Global	`uname`, `sysinfo`, `syslog`, `acct`, `_sysctl`, `iopl`, `reboot`

> **TIP**
>
> There is a nice, interactive page available online with source code references.

Now that you have a basic idea of the Linux kernel, its main components and interface, let's move on to the question of how to extend it.

Kernel Extensions

In this section we will focus on how to extend the kernel. In a sense, the content here is advanced and an optional one. You won't need it for your day to day work, in general.

> **NOTE**
>
> Configuring and compiling your own Linux kernel is out of scope for this book. For information on how to do it I recommend "Linux Kernel in a Nutshell" written by Greg Kroah-Hartman, one of the main Linux maintainers and project lead. He covers the entire range o tasks, starting with downloading the source code to configuration and installation steps, to kernel options at runtime.

Let's start with something easy: how do you know what kernel version you're using? You can use the following command to determine this:

```
$ uname -srm
Linux 5.11.0-25-generic x86_64 ❶
```

❶ From the `uname` output here you can tell that at time of writing, I'm using an 5.11 kernel on an `x86_64` machine, see also "x86 Architecture".

Now that we know the kernel version, address the question of how to extend the kernel out-of-tree, that is, without having to add features to the kernel source code and then have to build it. For this extention we can use modules, so let's have a look at that.

Modules

In a nutshell, a module is a program that you can load into a kernel on demand. That is, you do not necessarily have to recompile the kernel and/or reboot the machine.

To list available modules (output has been edited down as on my system there are over 1000 lines there):

```
$ find /lib/modules/$(uname -r) -type f -name '*.ko*'
/lib/modules/5.11.0-25-generic/kernel/ubuntu/ubuntu-host/ubuntu-
host.ko
/lib/modules/5.11.0-25-generic/kernel/fs/nls/nls_iso8859-1.ko
/lib/modules/5.11.0-25-generic/kernel/fs/ceph/ceph.ko
```

```
/lib/modules/5.11.0-25-generic/kernel/fs/nfsd/nfsd.ko
...
/lib/modules/5.11.0-25-generic/kernel/net/ipv6/esp6.ko
/lib/modules/5.11.0-25-generic/kernel/net/ipv6/ip6_vti.ko
/lib/modules/5.11.0-25-generic/kernel/net/sctp/sctp_diag.ko
/lib/modules/5.11.0-25-generic/kernel/net/sctp/sctp.ko
/lib/modules/5.11.0-25-generic/kernel/net/netrom/netrom.ko
```

That's great! But which modules did the kernel actually load? Let's see (output shortened):

```
$ lsmod
Module                  Size  Used by
...
linear                 20480  0
crct10dif_pclmul       16384  1
crc32_pclmul           16384  0
ghash_clmulni_intel    16384  0
virtio_net             57344  0
net_failover           20480  1 virtio_net
ahci                   40960  0
aesni_intel           372736  0
crypto_simd            16384  1 aesni_intel
cryptd                 24576  2 crypto_simd,ghash_clmulni_intel
glue_helper            16384  1 aesni_intel
```

Note that the above info is available via `/proc/modules`. This is thanks to the kernel exposing this information via a pseudo-filesystem interface, more on this topic in XREF HERE.

Want to learn more about a module or have a nice way to manipulate kernel modules? Then `modprobe` is your friend. For example, to list the dependencies:

```
$ modprobe --show-depends async_memcpy
insmod /lib/modules/5.11.0-25-
generic/kernel/crypto/async_tx/async_tx.ko
insmod /lib/modules/5.11.0-25-
generic/kernel/crypto/async_tx/async_memcpy.ko
```

Next up: an alternative, modern way to extend the kernel.

A Modern Way to Extend the Kernel: eBPF

An increasingly popular way to extend kernel functionality is eBPF. Originally, knowns as Berkeley Packet Filter (BPF), nowadays, the kernel project and technology is commonly known as eBPF (a term which does not stand for anything).

Technically, eBPF is a feature of the Linux kernel and you'll need the Linux kernel version 3.15 or above to benefit from it. It enables you to safely and efficiently extend the Linux kernel functions by using the bpf syscall. eBPF is implemented as a in-kernel virtual machine using a custom 64 bit RISC instruction set.

> **TIP**
>
> If you want to learn more about what is enabled in which kernel version for eBPF, you can be use the iovisor/bcc docs on GitHub.

In Figure 2-5 you see a high-level overview taken from Brendan Gregg's book Linux Extended BPF (eBPF) Tracing Tools:

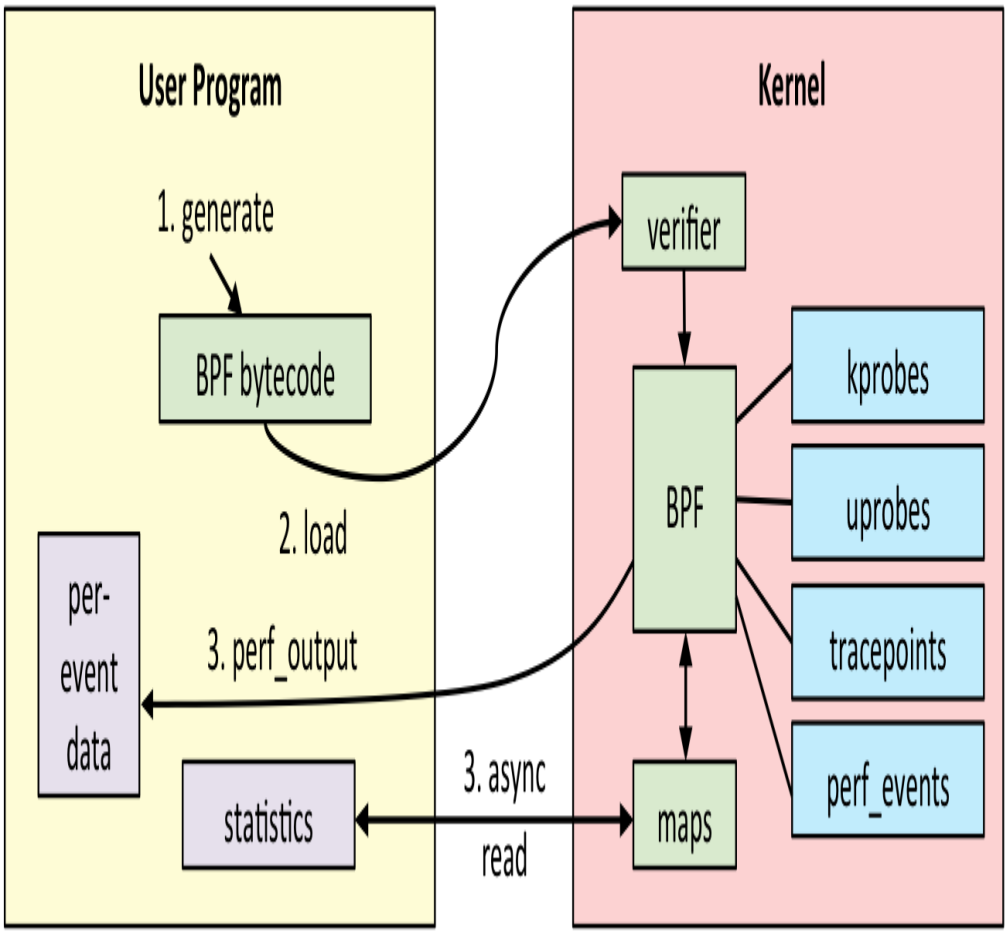


Figure 2-5. eBPF overview in the Linux kernel

eBPF is already used in a number of places and for use cases such as:

- In Kubernetes, as a CNI plugin to enable pod networking for example, in Cilium and Project Calico as well as for service scalability.
- For observability, like for Linux kernel tracing such as with iovisor/bpftrace as well as in a clustered setup with Hubble (see XREF HERE).
- As a security control, for example to perform container runtime scanning as you can use with projects such as CNCF Falco.

- Network loadbalancing like Facebook's L4 katran library.

In mid 2021 the Linux Foundation announced that Facebook, Google, Isovalent, Microsoft and Netflix joined together to create the eBPF Foundation, and with it giving the eBPF project a vendor-neutral home. Stay tuned!

To dive deeper into the eBPF topic I recommend Matt Oswalt's nice Introduction to eBPF. If you want to stay on top of things, have a look at ebpf.io.

Conclusion

The Linux kernel is the core of Linux the operating system and no matter what distribution you are using or in which ever environment you are using Linux, be it on your desktop or in the cloud, you should have a basic idea of its components and functionality.

In this chapter we reviewed the overall Linux architecture, the role the kernel has as well as its interfaces. Most importantly, the kernel abstracts away the differences of the hardware—CPU architecures and peripheral devices—and makes Linux very portable. The most important interface is the syscall interface, through which the kernel exposes its functionality, be it opening a file, allocating memory or listing network interfaces.

We have also looked a bit at inner workings of the kernel, including modules and eBPF as a way to extend the kernel functionality as well as the lower level functions offered by device drivers.

If you want to learn more about certain kernel aspects, the following resources should provide you with some starting points:

1. General:

    - The Linux Programming Interface by Michael Kerrisk (No Starch Press).

- Linux Kernel Teaching provides a nice introduction with deep-dives across the board.
- Anatomy of the Linux kernel gives a quick high-level intro.
- Operating System Kernels has a nice overview and comparison of kernel design approaches.
- KernelNewbies is a great resource if you want to dive deeper on hands-on topics.
- kernelstats shows some interesting distributions over time.
- The Linux kernel map is a visual representation of kernel components and dependencies.

2. Memory management:

- Understanding The Linux Virtual Memory Manager
- The Slab Allocator in the Linux kernel
- Kernel docs

3. Device drivers:

- Linux Device Drivers by Jonathan Corbet
- How to install a device driver on Linux
- Character device drivers
- Linux Device Drivers: Tutorial for Linux Driver Development

4. Syscalls:

- Linux Interrupts: The Basic Concepts
- The Linux Kernel: System Calls

- Linux System Call Table
- syscalls.h
- syscall lookup for x86 and x86_64

Equipped with this knowledge we are now ready to climb up the abstraction ladder a bit an move to the primary user interface we consider in this book: the shell, both in manual usage as well as automation through scripts.

Chapter 3. Shells and Scripting

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at modern-linux@pm.me.

In this chapter we will focus on interacting with Linux on the terminal, that is, via the shell which exposes a command line interface (CLI). It is vitally important to be able to use the shell effectively to accomplish everyday tasks and to that end we focus on usability, here.

First, we review some terminology and provide a gentle and concise introduction to shell basics. Then we have a look at modern, human-friendly shells, such as the Fish shell. We look at configuration and common tasks in the shell. Then, we move on to the topic of how to effectively work on the CLI using a terminal multiplexer, enabling you to work with multiple sessions, local or remote alike. In the last part of this chapter we switch gears and focus on automating tasks in the shell using scripts, including best practices how to write scripts in a safe, secure, and portable manner and also how to lint and test scripts.

There are two major ways to interact with Linux, from a CLI perspective. The first way is manual, that is, a human user sits in front of the terminal, interactively typing commands and consuming the output. This ad-hoc

interaction makes most of the things you want to do in the shell on a day-to-day basis, including:

- Listing directories, finding files, or looking for content in files.
- Copying files between directories or to remote machines.
- Reading emails, news or tweet from the terminal.

Further, we will learn how to conveniently and efficiently work with multiple shell sessions at the same time.

The other mode of operation is the automated processing of a series of commands in a special kind of file that the shell interprets for you and in turn executes. This mode is usually called shell scripting or just scripting. You typically want to use a script rather than manually repeating certain tasks. Also, scripts are the basis of many config and install systems. Scripts are indeed very convenient. However they can also pose a danger, if used without precautions. So, whenever you think about writing a script, keep the XKCD web comic shown in Figure 3-1 in mind, with kudos to Randall Munroe, made available under CC BY-NC 2.5.

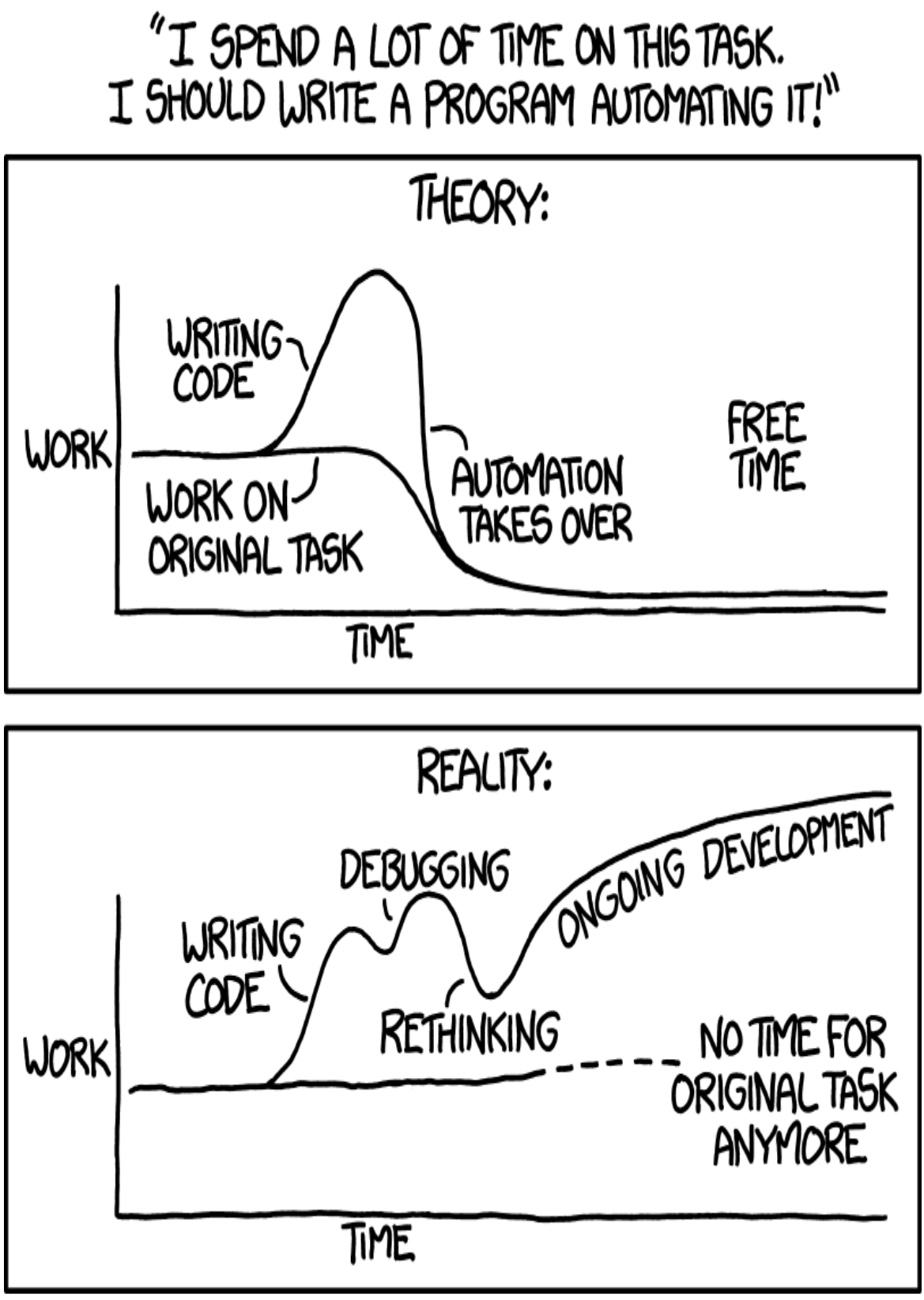


Figure 3-1. XKCD on Automation

I strongly recommend that you have a Linux environment available and try out the examples shown here right away. With that, are you ready for some (inter)action? If so, then let's start with some terminology and basic shell usage.

Basics

Before we get into different options and configurations, let's focus on some basic terms such as **terminal** and **shell**. In this section I will define the terminology and show you how to accomplish everyday tasks in the shell. We will also review modern commands and see them in action.

Terminals

We start with the terminal, or terminal emulator, or soft terminal, all of which refer to the same thing: a terminal is a program that provides a textual user interface. That is, a terminal supports reading characters from the keyboard and displaying them on the screen. Many years ago these used to be integrated devices (keyboard and screen together) but nowadays terminals are simply apps.

In addtion to the basic character-oriented input and output, terminals support so called escape sequences or escape codes, for cursor and screen handling and potentially support for colors. For example, pressing `CTRL+H` causes a backspace, that is, deletes the character to the left of the cursor.

The environment variable `TERM` has the teriminal emulater in use and its configuration is available via `infocmp` as follows (note that the output has been shortened):

```
$ infocmp
#       Reconstructed via infocmp from file:
/lib/terminfo/s/screen-256color
screen-256color|GNU Screen with 256 colors,
        am, km, mir, msgr, xenl,
        colors#0x100, cols#80, it#8, lines#24, pairs#0x10000,
        acsc=++\,\,-
```

```
-..00``aaffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
        bel=^G, blink=\E[5m, bold=\E[1m, cbt=\E[Z, civis=\E[?25l,
        clear=\E[H\E[J, cnorm=\E[34h\E[?25h, cr=\r,
        ...
```

Examples of terminals include not only `xterm`, `rxvt`, and the Gnome terminator, but also new generation ones that utilize the GPU such as Alacritty, kitty or warp.

In "Terminal multiplexer" we will come back to the topic of the terminal.

Shells

Next up is the shell, a program that runs inside the terminal and acts as a command interpreter. The shell offers input and output handling via streams, supports variables, has some built-in commands you can use, deals with command execution and status, and usually supports both interactive usage as well as scripted usage ("Scripting").

The shell is formally defined in sh and we often come across the term POSIX shell which will become more important in the context of scripts and portability.

Originally we had the Bourne shell `sh`, named after the author, but nowadays you usually find it replaced with the bash shell—a wordplay on the original version, short for "Bourne Again Shell"—which is widely used as the default.

If you are curious what you are using, use the `file -h /bin/sh` command to find out or if that fails, try `echo $0` or `echo $SHELL`.

> **NOTE**
>
> At least in this section, we assume the Bash shell (`bash`), unless we call it out explicitly.

There are many more implementations of `sh` as well as other varianats such as the Korn shell `ksh` and C Shell `csh`, nowadays not widely used. We will, however, review modern `bash` replacements in "Human-friendly Shells".

Let's start our shell basics with two fundamental features: streams and variables.

Streams

Let's start with the topic of input (streams) and output (streams) or I/O for short. How can you feed a program some input? How do you control where the output of a program lands, say, on the terminal or in a file?

First off, the shell equips every process with three default file descriptors (FD) for input and output:

- `stdin` (FD 0)
- `stdout` (FD 1)
- `stderr` (FD 2)

These file descriptors are, as depicted in Figure 3-2, by default connected to your screen and keyboard, respectively. In other words, unless you specify something else, a command you enter in the shell wil take its input (`stdin`) from your keyboard and it will deliver its output (`stdout`) to your screen.

The following shell interaction demonstrates this default behavior:

```
$ cat
This is some input I type on the keyboard and read on the
screen^C
```

Above, using `cat` as an example, you see the defaults in action and also note that I used `CTRL+c` (shown as `^C`) to termintate the command.

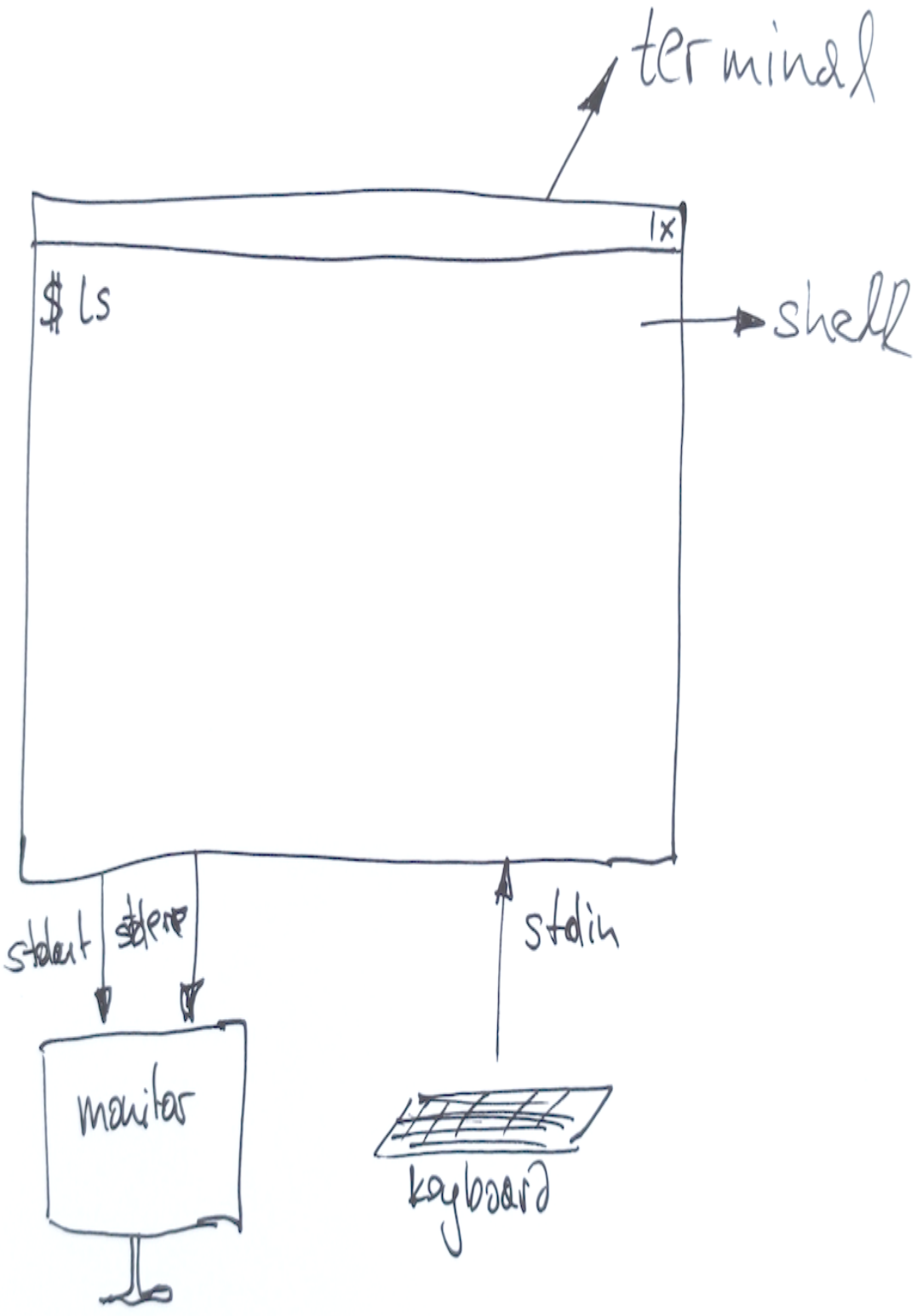


Figure 3-2. Shell I/O default streams

If you don't want to use the defaults the shell gives you, for example, you don't want `stderr` to be outputed on the screen but want to save it in a file, you can redirect the streams.

You redirect the output stream of a processes using `$FD>` and `<$FD`, with `$FD` being the file descriptor, for example `2>` means redirect the `stderr` stream. Note that `1>` and `>` is the same since `stdout` is the default, if you want to redirect both `stdout` and `stderr` use `&>` and when you want to get rid of a stream then you can use `/dev/null`.

Let's see how that works in the context of a concrete example, downloading some HTML content via `curl`:

```
$ curl https://example.com &> /dev/null ❶

$ curl https://example.com > /tmp/content.txt 2> /tmp/curl-status
❷
$ head -3 /tmp/content.txt
<!doctype html>
<html>
<head>
$ cat /tmp/curl-status
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
Left  Speed
100  1256  100  1256    0     0   3187      0 --:--:-- --:--:-- -
-:--:--  3195

$ cat > /tmp/interactive-input.txt ❸

$ tr < /tmp/curl-status [A-Z] [a-z] ❹
  % total    % received % xferd  average speed   time    time
time  current
                                 dload  upload   total   spent
left  speed
100  1256  100  1256    0     0   3187      0 --:--:-- --:--:-- -
-:--:--  3195
```

❶ Discard all output by redirecting both `stdout` and `stderr` to `/dev/null`.

❷ Redirect the output and status to different files.

❸ Interactively enter input and save to file, use `CTRL+D` to stop capturing and store the content.

❹ Lowercase all words, using the `tr` command that reads from `stdin`.

Shells usually understand a number of special characters, such as:

- `&` … at the end of a command, executes it in the background, see also "Job Control".
- `\` … continue a command on the next line, use this for better readability of long commands.
- `|` … the pipe, connects `stdout` of one process with the `stdin` of the next process, allowing you to pass data without having to store it in files as a temporary place.

PIPES AND THE UNIX PHILOSOPHY

While pipes might seem not too exciting at glance, there's much more to it. I once had a nice interaction with Doug McIlroy, the inventor of pipes. I wrote an article on Revisiting the Unix philosophy in 2018 where I drew parallels between Unix and microservices. Someone commented on the article and that comment led to Doug sending me an email (very unexpectedly and I had to verify to believe it) to clarify things.

Again, let's see some of the theoretical content in action. Let's try to figure out how many lines a HTML file contains by downloading it using `curl` and then pipe the content to the `wc` tool:

```
$ curl https://example.com 2> /dev/null | \ ❶
  wc -l ❷
46
```

❶ Use `curl` to download the content from URL and discard the status that it outputs on `stderr` (note: in practice you'd use the `-s` option of `curl` but we want to learn how to apply our hard-gained knowledge, right?).

❷ The `stdout` of `curl` is fed to `stdin` of `wc` which counts the number of lines with the `-l` option.

Now that you have a basic understanding of commands, streams and redirection, let's move on to another core shell feature, the handling of variables.

Variables

A term you will come across often in the context of shells is that of **variables**. Whenever you don't want to or can not hard code a value you can use a variable to store and change a value. Use cases include:

- Configuration items that Linux exposes, for example, the place where the shell looks for executables captured in the `$PATH` variable. This is kind of an interface where a variable might be read/write.
- You want to interactively query the user for a value, say, in the context of a script.
- When you want to shorten input by defining a long value once. For example, the URL of an HTTP API. This use case roughly corresponds to a `const` value in a program language since you don't change the value after you have declared the variable.

We distinguish between two kinds of variables:

1. Environment variables are system-wide settings; list them with `env`.

2. Shell variables are valid in the context of the current execution; list with `set` in Bash. Shell variables are not inherited by sub-processes.

You can, in Bash, use `export` to create an environment variable. When you want to access the value of a variable, then you need to put an `$` in front of it and when you want to get rid of it, use `unset`.

OK, that was a lot of information, let's see how that looks in practice (Bash):

```
$ set MY_VAR=42 ❶
$ set | grep MY_VAR ❷
_=MY_VAR=42

$ export MY_GLOBAL_VAR="fun with vars" ❸

$ set | grep 'MY_*' ❹
MY_GLOBAL_VAR='fun with vars'
_=MY_VAR=42

$ env | grep 'MY_*' ❺
MY_GLOBAL_VAR=fun with vars

$ bash ❻
$ echo $MY_GLOBAL_VAR ❼
fun with vars

$ set | grep 'MY_*' ❽
MY_GLOBAL_VAR='fun with vars'

$ exit ❾
$ unset $MY_VAR
$ set | grep 'MY_*'
MY_GLOBAL_VAR='fun with vars'
```

❶ Create a shell variable called `MY_VAR` and assign a value of 42.

❷ List shell variables and filter out `MY_VAR`, note the `_=` indicating it's not exported.

❸ Create a new environment variable called `MY_GLOBAL_VAR`.

❹ List shell variables and filter out all that start with `MY_` and we see, as expected, both the variables we created in the previous steps.

❺ List environment variables, and we see `MY_GLOBAL_VAR` as we would hope for.

❻ Create new shell session, that is, a child process of the current shell session which doesn't inherit `MY_VAR`.

❼ Access environment variable `MY_GLOBAL_VAR`.

❽ List shell variables, which gives us only `MY_GLOBAL_VAR` since we're in a child process.

❾ Exit child process, remove the `MY_VAR` shell variable and list our shell variables; as expected `MY_VAR` is gone.

In Table 3-1 I put together common shell and environment variables. You will find those variables almost everywhere and they are important to understand and to use. For any of the variables you can have a look at the respective value using `echo $XXX` with `XXX` being the variable name.

Table 3-1. Common shell and environment

v
a
r
i
a
b
l
e
s

Variable	Type	Semantics
`EDITOR`	environment	the path to program used by default to edit files
`HOME`	POSIX	the path of the home directory of the current user
`HOSTNAME`	Bash shell	the name of the current host
`IFS`	POSIX	list of characters to separate fields, used when the shell splits words on expansion
`PATH`	POSIX	contains a list of directories in which the shell looks for executable programs, binaries or scripts alike
`PS1`	environment	the primary prompt string in use
`PWD`	environment	the full path of the working directory
`RANDOM`	Bash shell	a random integer between 0 and 32767
`SHELL`	environment	contains the currently used shell
`TERM`	environment	the terminal emulator used
`UID`	environment	current user unique ID (integer value)
`USER`	environment	current user name
`_`	Bash shell	last argument to the previous command executed in the foreground

`?`	Bash shell	exit status, see "Exit Status"
`$`	Bash shell	the ID of the current process (integer value)
`0`	Bash shell	the name of the current process

Further, check out the full list of Bash specific variables and also note that the variables from Table 3-1 will come in handy again in the context of "Scripting".

Exit Status

The shell communicates the completion of a command execution to the caller using what is called the exist status. In general, it is expected that a Linux command returns a status when it terminates. This can either be a normal termination (happy path) or an abnormal termination (something went wrong). A `0` exit status means that the command was successful run, without any errors, whereas a non-zero value between 1 to 255 signals a failure. To query the exit status use `echo $?`.

Be careful with exit status handling in a pipeline, since some shells, only make the last status available. However, you can work around that limitation by using `$PIPESTATUS`.

Built-in Commands

Shells come with a number of built-in commands. Some useful examples are `yes`, `echo`, `cat`, or `read`. You can use `help` to list them, and remember that everything else is a shell-external program which you usually can find in `/usr/bin`. How do you know where to find an executable? Here are some ways:

```
$ which ls
/usr/bin/ls

$ type ls
ls is aliased to `ls --color=auto'
```

> **NOTE**
>
> One of the technical reviewers of the book rightfully pointed out that `which` is a non-POSIX, external program that may always be available. Also, they suggested that rather than `which` to use `command -v` to get the program path and or shell alias/function. See also the shellcheck docs for further details on the matter.

Job Control

A feature most shells support is called job control. By default, when you enter a command, it takes control of the screen and the keyboard, which we usually call running in the foreground. But what if you don't want to run something interactively, or, in case of a server, what if there is no input from `stdin` at all? Enter job control and background jobs: to launch a process in the background put an `&` at the end or to send a foreground process to the background press `CTRL+Z`.

The following example shows this in action, giving you a rough idea:

```
$ watch -n 5 "ls" & ❶

$ jobs ❷
Job     Group   CPU     State   Command
1       3021    0%      stopped watch -n 5 "ls" &

$ fg ❸
Every 5.0s: ls                                          Sat Aug 28
11:34:32 2021

Dockerfile
app.yaml
example.json
main.go
script.sh
test
```

❶ By putting the `&` at the end we launch the command in the background.

❷ List all jobs.

❸ With the `fg` command we can bring a process to the foreground.

If you want to keep a background process running, even after you close the shell you can prepend the `nohup` command. Further, for a process that is already running and wasn't prepended with `nohup` you can use `disown` after the fact to achieve the same effect. Finally, if you want to get rid of a running process you can use the `kill` command with various levels of forcefulness, see more details in XREF HERE.

Rather than job control, I recommend to use terminal multiplexer as discussed in "Terminal multiplexer". These programs take care of the most common use cases (shell closes, multiple processes running and need coordination, etc.) and also support working with remote systems.

Let's move on to discuss modern replacements for frequently used core commands that have been around forever.

Modern Commands

There are a handful of commands you will find yourself using over and over again, on a daily basis. This includes directory navigation (`cd`), listing the content of a directory (`ls`), finding files (`find`), or displaying the content of files (`cat`, `less`). Given that you are using these commands so often, you want to be as efficient as possible, every keystroke counts.

Now, for some of these often used commands there exist modern variations. Some of them are drop-in replacements others extend the functionality. All of them offer somewhat sane default values for common operations, rich output generally easier to comprehend, and they usually lead to you typing less to accomplish the same task. This reduces the friction when you work with the shell, making it more enjoyable and improving the flow. If you want to learn more about modern tooling check out XREF HERE.

Listing Directory Contents with `exa`

Whenever you want to know what a directory contains, you use `ls` or one of its variants with parameters. For example, in Bash I used to have `l` aliased to `ls -GAhltr`. But there's a better way: exa, a modern replacement for `ls`, written in Rust, with built-in support for Git and tree rendering. In this context, what would you guess is the most often used command after you've listed the directory content? In my experience it's to clear the screen and very often people are using `clear`. That's typing five characters and then hitting `ENTER`. You can have the same effect much faster, simply use `CTRL+L`.

Viewing File Contents with `bat`

Let's assume that you listed a directory content and found a file you want to inspect. You'd use `cat`, maybe? There's something better I recommend you to have a look at: bat. The `bat` command, shown in Figure 3-3 comes with syntax highlighting, shows non-printable characters, supports Git, and has a pager—the page-wise viewing of files longer than what can be displayed on the screen—integrated.

File: main.go

```go
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", HelloServer)
    http.ListenAndServe(":8080", nil)
}

func HelloServer(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %s!", r.URL.Path[1:])
}
```

File: app.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: something
  namespace: xample
spec:
  selector:
    matchLabels:
      app: sample
  replicas: 2
  template:
    metadata:
      labels:
        app: sample
    spec:
      containers:
        - name: example
          image: public.ecr.aws/mhausenblas/example:stable
```

Figure 3-3. Rendering of a Go file (left) and a YAML file (right) by `bat`

Finding Content in Files with `rg`

Traditionally, you would use `grep` to find something in a file. However, there's a modern command, rg, which is fast and powerful.

We're going to compare `rg` to a `find` and `grep` combination in this example, where we want to find YAML files that contain the string "sample":

```
$ find . -type f -name "*.yaml" -exec grep "sample" '{}' \; -
print ❶
      app: sample
        app: sample
./app.yaml

$ rg -t "yaml" sample ❷
app.yaml
9:      app: sample
14:        app: sample
```

❶ Use `find` and `grep` together to find a string in YAML files.

❷ Use `rg` for the same task.

If you compare the commands and the results in the previous example you see that not only is `rg` easier to use but also the results are more informative (providing context, in this case the line number).

JSON Data Processing with `jq`

And now for a bonus command. This one, `jq` is not an actual replacement but more like a specialized tool for JSON, a popular textual data format. You find JSON in HTTP APIs and configuration files, alike.

So, use jq rather than `awk` or `sed` to pick out certain values. For example, by using a JSON generator to generate some random data, I have a 2.4 kB

large JSON file `example.json` that looks something like this (only showing the first record here):

```
[
  {
    "_id": "612297a64a057a3fa3a56fcf",
    "latitude": -25.750679,
    "longitude": 130.044327,
    "friends": [
      {
        "id": 0,
        "name": "Tara Holland"
      },
      {
        "id": 1,
        "name": "Giles Glover"
      },
      {
        "id": 2,
        "name": "Pennington Shannon"
      }
    ],
    "favoriteFruit": "strawberry"
  },
...
```

Let's say we're interested in all "first" friends, that is, entry 0 in the `friends` array, of people whose favorite fruit is "strawberry". With `jq` you would do the following:

```
$ jq 'select(.[].favoriteFruit=="strawberry") | .
[].friends[0].name' example.json
"Tara Holland"
"Christy Mullins"
"Snider Thornton"
"Jana Clay"
"Wilma King"
```

That was some CLI fun, right? If you're interested in finding out more about the topic of modern commands and what other candidates there might be for you to replace, check out the mondern-unix repo, listing suggestions.

Let's now move our focus to some common tasks beyond directory navigation and file content viewing and how to go about them.

Common Tasks

There's a number of things you find yourself doing often and in addition there are certain tricks you can use to speed up your tasks in the shell. Let's review these common tasks and see how we can be more efficient.

Shorten Often-used Commands

One fundamental insight with interfaces is that commands that you are using very often should take the least effort, should be quick to enter. Now apply this idea to the shell: rather than `git diff --color-moved` I type `d` (a single character), since I'm viewing changes in my repositories many hundreds of times per day. Depending on the shell, there are different ways to achieve this: in Bash this is called an alias and in Fish ("Fish Shell") there are abbreviations you can use.

Navigating

When you enter commands on the shell prompt there are a number of things you might want to do, such as navigation (for example, move cursor to the start of the line) or manipulate the line (delete everything left to the cursor. In Table 3-2 you see common shell shortcuts listed.

Table 3-2. Shell navigation and editing

h
o
r
t
c
u
t
s

Action	Command	Note
move cursor to start of line	`CTRL+a`	-
move cursor to end of line	`CTRL+e`	-
move cursor forward one character	`CTRL+f`	-
move cursor back one character	`CTRL+b`	-
move cursor forward one word	`ALT+f`	Works only with left `ALT`
move cursor back one word	`ALT+b`	-
delete current character	`CTRL+d`	-
delete character left of cursor	`CTRL+h`	-
delete word left of cursor	`CTRL+w`	-
delete everything right of cursor	`CTRL+k`	-
delete everything left of cursor	`CTRL+u`	-
clear screen	`CTRL+l`	-
cancel command	`CTRL+c`	-
undo	`CTRL+_`	Bash only
search history	`CTRL+r`	Some shells

cancel search	`CTRL+g`	Some shells

Note that not all shortcuts may be supported in all shells and that certain actions such as history management may be implemented differently in certain shells. Take the table as a starting point and try out what your shell supports.

File Content Management

You don't always want to fire up an editor such as `vi` to add a single line of text. Also, sometimes you can't do it, for example, when you're in the context of writing a shell script ("Scripting").

So, how can you manipulate textual content? Let's have a look at a few examples:

```
$ echo "First line" > /tmp/something ❶

$ cat /tmp/something ❷
First line

$ echo "Second line" >> /tmp/something && \ ❸
  cat /tmp/something
First line
Second line

$ sed 's/line/LINE/' /tmp/something ❹
First LINE
Second LINE

$ cat << 'EOF' > /tmp/another ❺
First line
Second line
Third line
EOF

$ diff -y /tmp/something /tmp/another ❻
First line                              First line
Second line                             Second line
```

```
                                                                    >
Third line
```

❶ Create a file by redirecting the `echo` output.

❷ View content of file.

❸ Append a line to file using the >> operator and then view content.

❹ Replace content from file using `sed` and output to `stdout`.

❺ Create a file using the here document.

❻ Show differences between the files we created above.

Now that you know the basic file content manipulation techniques let's have a look at advanced viewing of file contents.

Viewing Long Files

For long files, that is, files that have more lines than the shell can display on your screen, you can use pagers like `less` or `bat` (that comes with a build-in pager). With paging, a program splits the output into pages where each page fits into what the screen can display and some commands to navigate the pages (view next page, previous page, etc.).

Another way to deal with long files is to only display a select region of the file like the first few lines. There are two handy commands for this: `head` and `tail`.

For example, to display the beginning of a file:

```
$ for i in {1..100} ; do echo $i >> /tmp/longfile ; done ❶

$ head -5 /tmp/longfile ❷
1
2
3
```

```
4
5
```

❶ Create a long file (100 lines here).

❷ Display the first five lines of the long file.

Or, to get live updates of a file that is constantly growing, we could use:

```
$ sudo tail -f /var/log/Xorg.0.log ❶
[ 36065.898] (II) event14 - ALPS01:00 0911:5288 Mouse: is tagged
by udev as: Mouse
[ 36065.898] (II) event14 - ALPS01:00 0911:5288 Mouse: device is
a pointer
[ 36065.900] (II) event15 - ALPS01:00 0911:5288 Touchpad: is
tagged by udev as: Touchpad
[ 36065.900] (II) event15 - ALPS01:00 0911:5288 Touchpad: device
is a touchpad
[ 36065.901] (II) event4  - Intel HID events: is tagged by udev
as: Keyboard
[ 36065.901] (II) event4  - Intel HID events: device is a
keyboard
...
```

❶ Display the end of a log file using `tail` with the `-f` option meaning to follow, that is, to update periodically.

Lastly in this section we look at dealing with date and time.

Date and Time Handling

The `date` command can be a useful way to generate unique file names. It allows you to generate dates in various formats including the Unix time stamp as well as to convert between different date and time formats.

```
$ date +%s ❶
1629582883

$ date -d @1629742883 '+%m/%d/%Y:%H:%M:%S' ❷
08/21/2021:21:54:43
```

❶ Create a Unix time stamp.

❷ Convert Unix time stamp to a human-readable date.

ON THE UNIX EPOCH TIME

The Unix epoch time or simply Unix time is the number of seconds elapsed since 1970-01-01T00:00:00Z. Unix time treats every day as exactly 86,400 seconds long.

If you're dealing with software that stores Unix time as a signed 32 bit integer, you might want to pay attention since this will cause issues on 2038-01-19 as then the counter will overflow, which is also known as the Year 2038 problem.

You can use online converters for more advanced operations, supporting microseconds and milliseconds resolutions.

With that we wrap up the shell basics section. By now you should have a good understanding what terminals and shells are and how to use them to do basic tasks such as navigating the filesystem, finding files and more. We move on to the topic of humand-friendly shells.

Human-friendly Shells

While the Bash shell is likely still the most widely used shell, it is not necessarily the most human-friendly one. It has been around since the late 1980s and the age sometimes shows. There are a number of modern, human-friendly shells I strongly recommend you to evaluate and use instead of Bash.

We will first do a detailed examination on one concrete example of a modern, human-friendly shell called the Fish shell, and then briefly discuss others, just to make sure you have an idea about the range of choices. We

wrap up this section with a quick recommendation and conclusion in "Which Shell Should I Use?".

Fish Shell

The Fish shell describes itself as a smart and user-friendly command line shell. Let's have a look at some basic usage first and then move on to configuration topics.

Basic Usage

For many of the daily tasks you won't notice a big difference to Bash in terms of input, most of the commands provided in Table 3-2 are valid. However, there are two areas where `fish` is different and much more convenient than `bash`:

- No explicit history management. You simply type and you get previous executions of a command shown. You can use the up and down key to select one, see Figure 3-4.
- Autosuggestions for many commands. As shown in Figure 3-5. In addition, when you press `Tab`, the Fish shell will try to complete the command, argument, or path, giving you visual hints such as coloring your input in red if it doesn't recognize the command.

Figure 3-4. Fish history handling in action

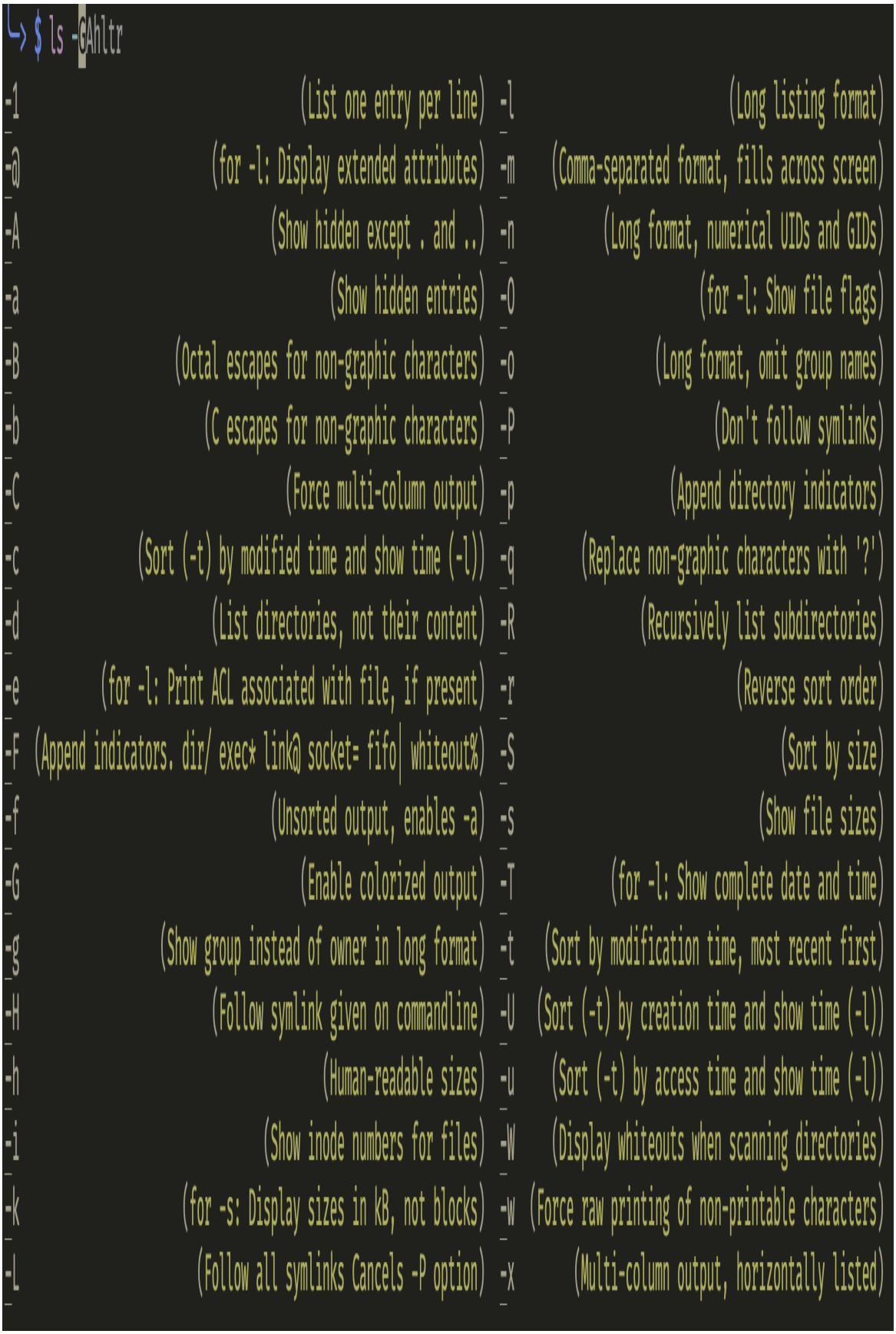

└─> $ ls -GAhltr
-1 (List one entry per line)
-@ (for -l: Display extended attributes)
-A (Show hidden except . and ..)
-a (Show hidden entries)
-B (Octal escapes for non-graphic characters)
-b (C escapes for non-graphic characters)
-C (Force multi-column output)
-c (Sort (-t) by modified time and show time (-l))
-d (List directories, not their content)
-e (for -l: Print ACL associated with file, if present)
-F (Append indicators. dir/ exec* link@ socket= fifo| whiteout%)
-f (Unsorted output, enables -a)
-G (Enable colorized output)
-g (Show group instead of owner in long format)
-H (Follow symlink given on commandline)
-h (Human-readable sizes)
-i (Show inode numbers for files)
-k (for -s: Display sizes in kB, not blocks)
-L (Follow all symlinks Cancels -P option)
-l (Long listing format)
-m (Comma-separated format, fills across screen)
-n (Long format, numerical UIDs and GIDs)
-O (for -l: Show file flags)
-o (Long format, omit group names)
-P (Don't follow symlinks)
-p (Append directory indicators)
-q (Replace non-graphic characters with '?')
-R (Recursively list subdirectories)
-r (Reverse sort order)
-S (Sort by size)
-s (Show file sizes)
-T (for -l: Show complete date and time)
-t (Sort by modification time, most recent first)
-U (Sort (-t) by creation time and show time (-l))
-u (Sort (-t) by access time and show time (-l))
-W (Display whiteouts when scanning directories)
-w (Force raw printing of non-printable characters)
-x (Multi-column output, horizontally listed)

Figure 3-5. Fish autosuggestion in action

In Table 3-3 you see some common `fish` commands listed, and in this context, note specifically the handling of environment variables.

Table 3-3. Fish shell reference

Task	Command
Export environment variable `KEY` with value `V`	`set -x KEY VAL`

AL	
Delete environment variable `KEY`	`set -e KEY`
Inline env var `KEY` for command `cmd`	`env KEY=VAL cmd`
Change path length to 1	`set -g fish_prompt_pwd_dir_length 1`
Manage abbreviations	`abbr`
Manage functions	`functions` and `funcd`

Unlike other shells, `fish` stores the exit status of the last command in a variable called `$status` instead of in `$?`.

If you're coming from Bash, you may also want to consult the Fish FAQ which addresses most of the gotchas.

Configuration

To configure the Fish shell, you simply enter the `fish_config` command and `fish` will launch a server via `http://localhost:8000` and automtically open your default browser with a fancy UI shown in Figure 3-6 which allows you to view and change settings.

colors | prompt | functions | variables | history | bindings | abbreviations

Current

Background Color:
(demo only)

```
/bright/vixens jump | dozy "fowl" > quack &
echo 'Errors are the portals to discovery
# This is a comment
Ths is an autosuggestion
```

Customize | Set Theme

Preview a theme below:

Current

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

fish default

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

ayu Light ↗

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

ayu Dark ↗

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

ayu Mirage ↗

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Solarized Light ↗

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Solarized Dark ↗

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Tomorrow ↗

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Tomorrow Night ↗

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Tomorrow Night Bright ↗

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Nord ↗

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Base16 Default Dark ↗

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Base16 Default Light ↗

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Base16 Eighties ↗

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Snow Day

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Lava

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Seaweed

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Fairground

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Bay Cruise

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Old School

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Just a Touch

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Dracula

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Mono Lace

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Mono Smoke

```
/bright/vixens jump | dozy "
echo 'Errors are the portals
# This is a comment
Ths is an autosuggestion
```

Figure 3-6. Fish shell configuration via browser

Let's now see how I have configured my environment.

My config is rather short, in `config.fish` I have the following:

```
set -x FZF_DEFAULT_OPTS "-m --bind='ctrl-o:execute(nvim
{})+abort'"
set -x FZF_DEFAULT_COMMAND 'rg --files'
set -g FZF_CTRL_T_COMMAND "command find -L \$dir -type f 2>
/dev/null | sed '1d; s#^\./##'"
set -x EDITOR nvim
set -x KUBE_EDITOR nvim
set -ga fish_user_paths /usr/local/bin
```

My prompt, defined in `fish_prompt.fish` looks as follows:

```
function fish_prompt
    set -l retc red
    test $status = 0; and set retc blue

    set -q __fish_git_prompt_showupstream
    or set -g __fish_git_prompt_showupstream auto

    function _nim_prompt_wrapper
        set retc $argv[1]
        set field_name $argv[2]
        set field_value $argv[3]

        set_color normal
        set_color $retc
        echo -n '─'
        set_color -o blue
        echo -n '['
        set_color normal
        test -n $field_name
        and echo -n $field_name:
        set_color $retc
        echo -n $field_value
        set_color -o blue
        echo -n ']'
    end

    set_color $retc
    echo -n '┬─'
```

```
    set_color -o blue
    echo -n [
    set_color normal
    set_color c07933
    echo -n (prompt_pwd)
    set_color -o blue
    echo -n ']'
     # Virtual Environment
    set -q VIRTUAL_ENV_DISABLE_PROMPT
    or set -g VIRTUAL_ENV_DISABLE_PROMPT true
    set -q VIRTUAL_ENV
    and _nim_prompt_wrapper $retc V (basename "$VIRTUAL_ENV")

    # git
    set prompt_git (fish_git_prompt | string trim -c ' ()')
    test -n "$prompt_git"
    and _nim_prompt_wrapper $retc G $prompt_git

    # New line
    echo

    # Background jobs
    set_color normal
    for job in (jobs)
        set_color $retc
        echo -n '| '
        set_color brown
        echo $job
    end
    set_color blue
    echo -n ' └─> '
        set_color -o blue
    echo -n '$ '
    set_color normal
end
```

The above prompt definition yields a prompt shown in Figure 3-7 and there note the difference between a directory that contains a Git repo and one that does not, yet another built-in visual contextual information, speeding up your flow. Also, notice the current time on the right-hand side.

Figure 3-7. Fish shell prompt

My abbreviations—think: `alias` replacement, found in other shells—look as follows:

```
$ abbr
abbr -a -U -- :q exit
abbr -a -U -- cat bat
abbr -a -U -- d 'git diff --color-moved'
abbr -a -U -- g git
abbr -a -U -- grep 'grep --color=auto'
abbr -a -U -- k kubectl
abbr -a -U -- l 'exa --long --all --git'
abbr -a -U -- ll 'ls -GAhltr'
abbr -a -U -- m make
abbr -a -U -- p 'git push'
abbr -a -U -- pu 'git pull'
abbr -a -U -- s 'git status'
abbr -a -U -- stat 'stat -x'
abbr -a -U -- vi nvim
abbr -a -U -- wget 'wget -c'
```

To add a new abbreviation use `abbr --add`. Abbreviations are handy for simple commands that take no arguments. What if you have a more complicated construct you want to shorten? Say, you want to shorten a sequence involving `git` that also takes an argument? Meet functions in Fish.

Let's now take a look at an example function, defined in `c.fish`. We can use the `functions` command to list functions, the `function` command to create a new one and, in this case, the `funced c` command to edit it:

```
function c
    git add --all
    git commit -m "$argv"
end
```

With that we have reached the end of the Fish section, providing you a usage tutorial and configuration tips and now let's have a quick look at other modern shells.

The Z-shell

Z-shell or `zsh` is a Bourne-like shell with a powerful completion system and rich theming support. With Oh My Zsh you can pretty much configure and use `zsh` in the way you've seen earlier on with `fish` while retaining wide backwards compatibility with Bash.

`zsh` uses five startup files as shown in the following (note that if `$ZDOTDIR` is not set, then `zsh` uses `$HOME` instead):

```
$ZDOTDIR/.zshenv ❶
$ZDOTDIR/.zprofile ❷
$ZDOTDIR/.zshrc ❸
$ZDOTDIR/.zlogin ❹
$ZDOTDIR/.zlogout ❺
```

❶ Sourced on all invocations of the shell, should contain commands to set the search path, plus other important environment variables. but should not contain commands that produce output or assume the shell is attached to a tty.

❷ Is meant as an alternative to `.zlogin` for `ksh` fans (these two are not intended to be used together); similar to `.zlogin`, except that it is sourced before `.zshrc`.

❸ Sourced in interactive shells, should contain commands to set up aliases, functions, options, key bindings, etc.

4. Sourced in login shells. It should contain commands that should be executed only in login shells. Note that `.zlogin` is not the place for alias definitions, options, environment variable settings, etc.

5. Sourced when login shells exit.

For more `zsh` plugins see also the awesome-zsh-plugins repo on GitHub and if you want to learn `zsh`, consider reading An Introduction to the Z Shell by Paul Falstad and Bas de Bakker.

Other Modern Shells

In addition to `fish` and `zsh` there are a number of other interesting, but not necessarily always Bash compatible shells available out there. When you have a look at those, ask yourself what the focus of the respective shell is (interactive usage vs. scripting) and how active the community around it is.

Some examples of modern shells for Linux I came across and can recommend you to have a look at include:

- Oil shell is targetting Python and JavaScript users. Put in other words: the focus is less on interactive use but more on scripting.
- murex, a POSIX shell that sports interesting features such as an integrated testing framework, typed pipelines, and event-driven programming.
- Nushell is an experimental new shell paradigm, featuring tabular output with a powerful query language. Learn more via the detailed Nu Book.
- PowerShell, a cross-platform shell that started off as a fork of the Windows PowerShell and offers a different set of semantics and interactions than POSIX shells.

There are many more out there, keep looking and try out what works best for you, try thinking beyond Bash and optimize for your use case.

Which Shell Should I Use?

At this point in time, every modern shell—other than Bash—seems like a good choice, from a human-centric perspective. Smooth auto-complete, easy config, and smart environments are no luxury in 2021 and, given the time you usually spend on the command line, you should try out different shells and pick the one you like most. I personally use the Fish shell, but many of my peers are super happy with the Z-shell.

You may have issues that make you hesitant to move away from Bash, specifically:

- Remote systems/can not install my own shell, have to use Bash.
- Compatibility, muscle memory. It can be hard to get rid of certain habits.
- Almost all instructions (implicitly) assume Bash, for example, you would see instructions like `export FOO=BAR` which is Bash specific.

It turns out that above issues are by and large not relevant to most users. While it may be the case that you have to temporarily use Bash in a remote system most of the time you will be working in an environment that you control. There is a learning curve, but the investment pays off in the long run.

With that, let's focus on another way to boost your productivity in the terminal: multiplexer.

Terminal multiplexer

We came across terminals already at the beginning of this chapter, in "Terminals". Now let's dive deeper into the topic of how to improve your

terminal usage, building on a concept that is both simple and powerful: multiplexing.

Think of it the following way: you usually work on different things that can be grouped together, for example, you may work on an open source project, authoring of a blog post or docs, some server remote access, interacting with an HTTP API to test things, and so forth. These tasks may each require one or more terminal windows and oftentimes you want to or need to do potentially interdependent tasks in two windows at the same time, for example:

- You are using the `watch` command to periodically execute a directory listing and at the same time edit a file.
- You start a server process (a Web server or application server) and want to have it running in the foreground (see also "Job Control") to keep an eye on the logs.
- You want to edit a file using `vi` and at the same time use `git` to query the status and commit changes.
- You have a VM running in the public cloud and want to `ssh` into it while having the possibility to manage files locally.

Think of all of the above examples as things that logically belong together, and in terms of time duration can range for short-term (a few minutes) to long term (days and weeks). The grouping of those tasks is usually called a session.

Now, there are a number of challenges if you want to achieve above:

- You need multiple windows, so one solution is to launch multiple terminals or if the UI supports it, multiple instances (tabs).
- You would like to have all the windows and paths around, even if you close the terminal or the remote side closes down.
- You want to expand or zoom in and out to focus on certain tasks, while keeping an overview of all your sessions, being able to

navigate between them.

To enable these tasks, people came up with the idea of overlaying a terminal with multiple windows (and sessions, to group windows). Put in other words: to multiplex the terminal I/O.

Let's have a brief look at the original implementation of terminal multiplexing, called `screen`. Then we focus in-depth on a widely used implement called `tmux` and wrap up with other options in this space.

`screen`

screen is the original terminal multiplexer and is still used. Unless you're in a remote environment where nothing else is available and/or you can't install another multiplexer you should probably not be using `screen` nowadays. One reason is that it's not actively mainted anymore, another that it is not very flexible and lacks a number of features modern terminal multiplexer have.

`tmux`

tmux is a flexible and rich terminal multiplexer that you can bend to your needs. As you can see in Figure 3-8 there are three core elements you're interacting with in `tmux`, from coarse-grained to fine-grained units:

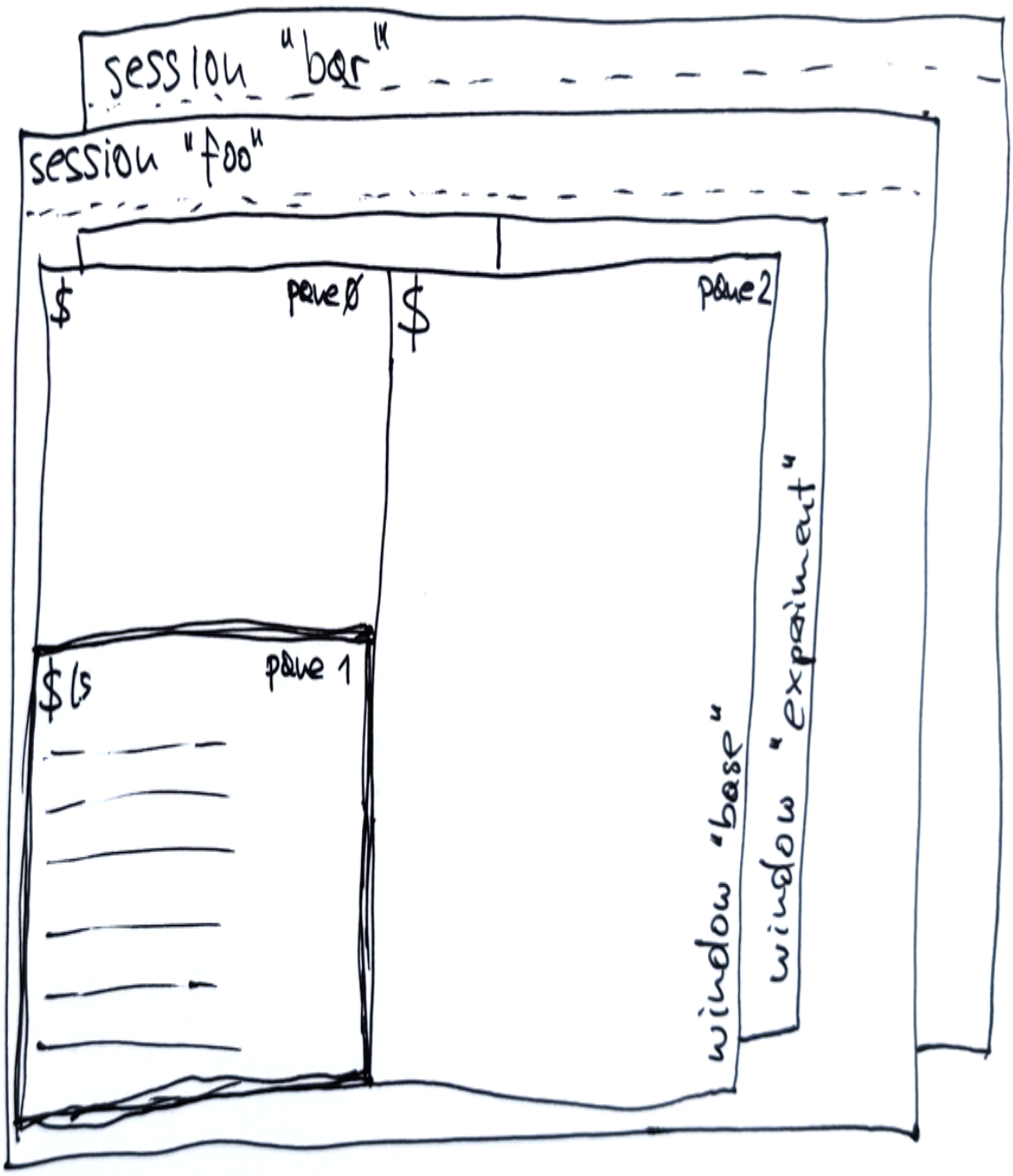


Figure 3-8. The `tmux` elements: sessions, windows, and panes.

- Sessions: a logically unit, think of it as a working environment dedicated to a specific task such as “working on project X” or “writing blog post Y”. It’s the container for all other units.
- Windows: you can think of a window as a tab in a browser, belonging to a session. It’s optional to use and oftentimes you only

have one window per session.

- Panes: those are your workhorses, effectively a single shell instance running. A pane is part of a window, and you can easily split it vertically or horizontally, as well as expand/collapse it (think: zoom), and close panes as you need them.

Just like `screen` you have the concept of attaching and detaching to a session, in `tmux`. Let's assume we start from scratch, let's launch it with a session called `test`:

```
$ tmux new -s test
```

With above command `tmux` is running as a server and you find yourself in a shell you've configured in `tmux`, running as the client. This client/server model allows you to create, enter, leave, destroy sessions and use the shells running in it without having to think of the processes running (or: failing) in it.

`tmux` uses `CTRL+b` as the default keyboard shortcut also called prefix or trigger. So for example, to list all windows you would press `CTRL+b` and then `w` or to expand the current (active) pane you would use `CTRL+b` and then `z`.

TIP

In `tmux` the default trigger is `CTRL+b`. To improve the flow, I mapped the trigger to an unused key, so a single keystroke is sufficient. The way I did it is as follows: I mapped the trigger to the `Home` key in `tmux` and further that `Home` key to the `CAPS LOCK` key by changing its mapping in `/usr/share/X11/xkb/symbols/pc` to `key <CAPS> { [ Home ] };`.

The double-mapping described here is a workaround I needed to do. So, depending on your target key or terminal you might not have to do this, but I strongly encourage you to map `CTRL+b` to an unused key you can easily reach since you will press it many times a day.

You can now use any of the commands listed in Table 3-4 to manage further sessions, windows, panes and also, when pressing `CTRL+b + d` you can detach sessions. This means effectively that you put `tmux` into the background.

When you then start a new terminal instance or, say, you `ssh` to your maching from a remote place, you can then attach to an existing session, so let's do that with the `test` session we created earlier:

```
$ tmux attach -t test ❶
```

❶ Attach to existing session called `test`. Note that if you want to detach the session from its previous terminal you would also supply the `-d` parameter.

Table 3-4 lists common `tmux` commands grouped by the units we discussed above, from widest scope (session) to narrowest one (pane).

Table 3-4. tmux reference

Target	Task	Command
Session	create new	`:new -s NAME`
Session	rename	trigger + `$`
Session	list all	trigger + `s`
Session	close	trigger
Window	create new	trigger + `c`

Window	rename	trigger + `,`
Window	switch to	trigger + `1` … `9`
Window	list all	trigger + `w`
Window	close	trigger + `&`
Pane	split horizontal	trigger + `"`
Pane	split vertical	trigger + `%`
Pane	toggle	trigger + `z`
Pane	close	trigger + `x`

Now that you have a baisc idea how to use `tmux` let's turn our attention on how to configure and customize it. My `.tmux.conf` looks as follows:

```
unbind C-b ❶
set -g prefix Home
bind Home send-prefix
bind r source-file ~/.tmux.conf \; display "tmux config reloaded
:)" ❷
bind \\ split-window -h -c "#{pane_current_path}" ❸
bind - split-window -v -c "#{pane_current_path}"
bind X confirm-before kill-session ❹
set -s escape-time 1 ❺
set-option -g mouse on ❻
set -g default-terminal "screen-256color" ❼
set-option -g status-position top ❽
set -g status-bg colour103
set -g status-fg colour215
set -g status-right-length 120
set -g status-left-length 50
set -g window-status-style fg=colour215
set -g pane-active-border-style fg=colour215
set -g @plugin 'tmux-plugins/tmux-resurrect' ❾
set -g @plugin 'tmux-plugins/tmux-continuum'
set -g @continuum-restore 'on'
run '~/.tmux/plugins/tpm/tpm'
```

❶ This line and the next two lines changes the trigger to `Home`.

❷ Reload config via `TRIGGER + r`

❸ This line and next redefines pane splitting; retain current directory of existing pane.

❹ Adds shortcuts for new and kill sessions.

❺ No delays.

❻ Enable mouse selections.

❼ Set the default terminal mode to 256color mode

❽ Theme settings (next six lines).

❾ From here to the end: plugin management.

First install tpm, the tmux plugin manager and then `TRIGGER + I` for the plugins. The plugins used here are:

- tmux-resurrect, allows to restore sessions with `Ctrl-s` (safe) and `Ctrl-r` (restore).
- tmux-continuum, automatically saves/restores session (15min interval)

Figure 3-9 shows my Alacritty terminal running `tmux`, you see the sessions with the shortcuts 0 to 9, located in the left upper corner.

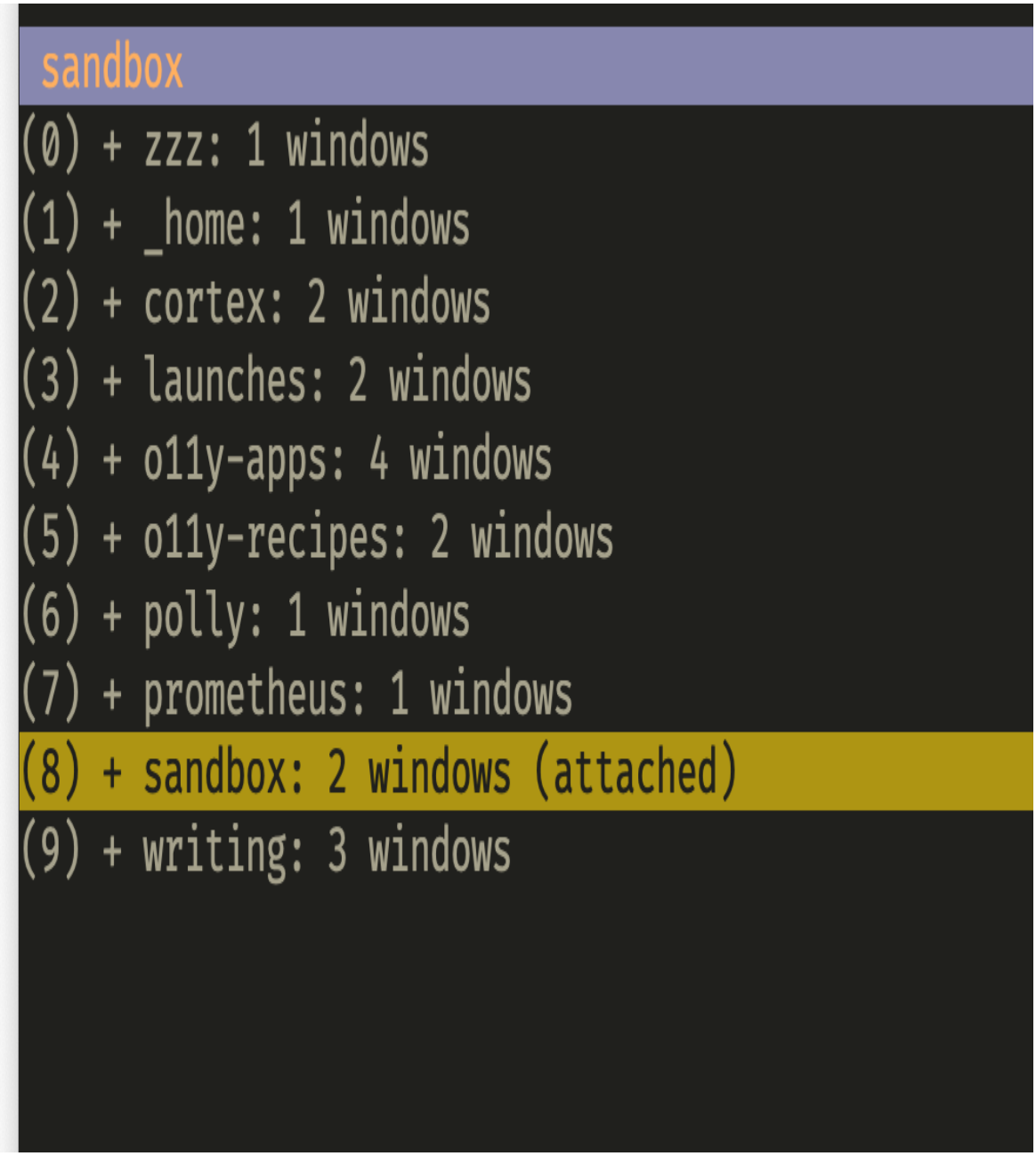


Figure 3-9. An example `tmux` instance in action, showing available sessions

While `tmux` certainly is an excellent choice, there are indeed other options than `tmux`, so let's have a peek.

Other Multiplexer

Other terminal multiplexer you can have a look at and try out include:

- tmuxinator is a meta-tool, allowing you to manage `tmux` sessions.
- Byobu is wrapper around either `screen` or `tmux`, especially interesting for you if you're using the Ubuntu or Debian-based Linux distros.
- Zellij calls itself a terminal workspace, is written in Rust and goes beyond what `tmux` offers, including a layout engine and a powerful plugin system.
- dvtm brings the concept of tiling window management, to the terminal; powerful but also a learning curve like `tmux` has.
- 3mux is a simple terminal mulitplexer written in Go, easy to use but not as powerful as `tmux`.

With this quick review of mulitplexer options out of the way, let's talk about selecting one.

Which Mulitplexer Should I Use?

Unlike with shells for human users I do have a concrete preference here in the context of terminal mulitplexer: use `tmux`. The reasons are manifold: it is mature, stable, rich (many plugins) and flexible. Many folks are using it, so there's plenty of material out there to read up on as well as help available. The others are exciting but relatively new or, as the case with `screen`, their prime time has been already some time ago.

With that, I hope I was able to convince you to consider using a terminal multiplexer to improve your terminal and shell experience, speed up your tasks and make the overall flow smoother.

BRINGING IT ALL TOGETHER: TERMINAL, MUX, SHELL

I'm using Alacritty as my terminal. It's fast and best of all: to configure it, I am using a YAML configuration file that I can version in Git, allowing me to use it on any target system in seconds. This config file called `alacritty.yml` defines all my settings for the terminal, from colors to key bindings to font sizes.

Most of the settings apply right away (hot-reload), others when I save the file. One setting, called `shell`, defines the integration between the terminal multiplexer I use (`tmux`) and the shell I use (`fish`) and looks as follows:

```
...
shell:
  program: /usr/local/bin/fish
  args:
  - -l
  - -i
  - -c
  - "tmux new-session -A -s zzz"
...
```

In the above snippet I configure Alacritty to use `fish` as the default shell but also, when I launch the terminal, it automatically attaches to a specific session. Together with the `tmux-continuum` plugin this gives me piece of mind. Even if I switch off the computer, once I restart I find my terminal with all its sessions, windows, and panes (almost) exactly in the state it was before a crash, modulo shell variables.

Now, we turn our attention to the last topic in this chapter, automating tasks with shell scripts.

Scripting

In the previous sections of this chapter we focused on the manual, interactive usage of the shell. Once you've done a certain task over and over again manually on the prompt, it's likely time to automate the task. This is where scripts come in.

We focus on writing scripts in Bash, here. This is due to two reasons:

- Most of the scripts out there are written in Bash and hence you will find a lot of examples and help available for Bash scripts.
- The likelihood of finding Bash available on a target system is high, making your potentially user base bigger than if you'd be using a (potentially more powerful but esoteric and not widely used) alternative to Bash.

Just to provide you some with some context before we start, there are shell scripts out there that clock in at several thousands of lines of code. Not that I encourage you to aim for this, quite the opposite: if you find yourself writing long scripts, ask yourself if a proper scripting language such as Python or Ruby is the better choice.

Let's step back now and develop a short but useful example, applying good practices along the way. Let's assume we want to automate the task of displaying a single statement on the screen that, given a user's GitHub handle, shows when the user joined, using their full name. Something along the line of:

```
XXXX XXXXX joined GitHub in YYYY
```

How do we go about automating this task with a script? Let's start with the basics, then review portability, and work our way up to the "business logic" of the script.

Scripting Basics

The good news is that by interactively using a shell you already know most of the relevant terms and techniques. In addition to variables, streams and

redirection, and common commands, there are a few specific things you want to be familiar with in the context of scripts, so let's review them.

Advanced Data Types

While shells usually treat everything as strings (if you want to perform some more complicated numerical tasks you should probably not use a shell script) they do support some advanced data types such as arrays.

Let's have a look at arrays in action:

```
os=('Linux' 'macOS' 'Windows') ❶
echo "${os[0]}" ❷
numberofos="${#os[@]}" ❸
```

❶ Define an array with three elements.

❷ Access the first element, would print `Linux`.

❸ Get the length of the array, resulting in `numberofos` being 3.

Flow Control

Flow control allows you to branch (`if`) or repeat (`for` and `while`) in your script, making the execution dependent on a certain condition.

Some usage examples of flow control:

```
for afile in /tmp/* ; do ❶
  echo "$afile"
done

for i in {1..10}; do ❷
    echo "$i"
done

while true; do
  ...
done ❸
```

❶ Basic loop iterating over a directory, printing each file name.

❷ Range loop.

❸ Forever loop, break out with `CTRL+C`.

Functions

Functions allow you to write more modular and reusable scripts. You have to define the function before you use it since the shell interpets the script from top to bottom.

A simple function example:

```
sayhi() { ❶
    echo "Hi $1 hope you are well!"
}

sayhi "Michael" ❷
```

❶ Function definition, parameters implicitly passed via `$n`.

❷ Function invocation, the output is “Hi Michael hope you are well!”.

Advanced I/O

With `read` you can read user input from `stdin` that you can use to elicit runtime input, for example, a menu of options. Further, rather than using `echo`, consider `printf` which allows you fine-grained control over the output, including colors. `printf` is also more portable than `echo`.

An example usage of the advanced I/O in action:

```
read name ❶
printf "Hello %s" "$name" ❷
```

❶ Read value from user input.

❷ Output value read in the previous step.

There are other, more advanced concepts available for you such as signals and traps. Given that we only want to provide an overview and introduction to the scripting topic here, I will refer you to the excellent Bash scripting cheatsheet for a comprehensive reference of all the relevant constructs. If you are serious about writing shell scripts, I can recommend you to read the bash Cookbook by Carl Albing, JP Vossen, and Cameron Newham which contains lots and lots of great snippets you can use as a starting point.

Writing Portable Bash Scripts

We now have a look at what it means to write portable scripts in Bash. But wait. What does portable mean and why should you care?

At the beginning of "Shells" we defined what POSIX means, so let's build on that. When I say portable, I mean that we are not making to many assumptions—implicitly or explicitly—about the environment a script will be executed. If a script is portable, it runs on many different systems (shells, Linux distros, etc.).

But remember that, even if you pin down the type of shell, in our case to Bash, not all features work the same way across different versions of a shell. At the end of the day it boils down to the number of different environments you can test your script.

Executing Portable Scripts

How are scripts executed? First, let's state that scripts really are simple text files, the extension doesn't matter, although often you find `.sh` as a convention used. But there are two things that turn a text file into a script that is executable, and able to be run by the shell:

- The text file needs to declare the interpreter in the first line, using what is called shebang (or hashbang) that is written as `#!`, see also the first line of the template below.

- Then, you need to make the script executable using, for example, with `chmod +x` which allows everyone to run it, or even better `chmod 750` which is more along the lines of least privileges. We will dive deep into this topic in Chapter 4.

Now that you know about the basics, let's have a look at a concrete template we can use as a starting point.

A Skeleton Template

A skeleton template for a portable Bash shell script that you can use as a seed looks as follows:

```
#!/usr/bin/env bash ❶
set -o errexit ❷
set -o nounset ❸
set -o pipefail ❹

firstargument="${1:-somedefaultvalue}" ❺

echo "$firstargument"
```

❶ The hashbang instructing the program loader that we want it to use `bash` to interpret this script.

❷ Define that we want to stop the script execution if an error happens.

❸ Define that we treat unset variables as an error (so the script is less likely to fail silently).

❹ Define that when one part of a pipe fails the whole pipe should be considered failed. This helps to avoid silent failures.

❺ An example command line parameter with a default value.

We will use this template later in this section two implement our GitHub info script.

Good Practices

I'm using "good practices" instead of "best practices" because what you should do depends on the situation and how far you want to go. There is a difference between a script you write for yourself vs. one that you ship to thousands of users, but in general, high-level good practices writing scripts are as follows:

Fail fast and loud

> Avoid silent fails and fail fast, things like `errexit` and `pipefail` do that for you. Since Bash tends to fail silently by default, failing fast is almost always a good idea.

Sensitive information

> Don't hardcode any sensitive information such as passwords into the script. Such information should be provided at runtime, via user input or calling out to an API. Also, consider that a `ps` reveals program parameters and more so that's another way how sensitive information can be leaked.

Input sanitization

> Set and provide sane defaults for variables where possible as well as sanitize the input you receive. For example, launch parameters provided or interactively ingested via `read` to avoid situations where an innocent looking `rm -rf "$PROJECTHOME/"*` wipes your drive because the variable wasn't set.

Check dependencies

> Don't assume that a certain tool or command is available, unless it's a build-in or you know your target environment. Just because your machine has `curl` installed doesn't mean the target machine has. If possible, provide fallbacks, for example, if no `curl` is available use `wget`.

Error handling

When your script fails (and it's not a matter if but only when and where) provide actionable instructions for your users. For example, rather than `Error 123` say what has failed and how your user can fix the situation, such as `Tried to write to /project/xyz/ but seems this is read-only for me`.

Documentation

Document your scripts inline (using `# Some doc here`) for main blocks and try to stick to 80 columns width for readability and diffing.

Versioning

Consider versioning your scripts using Git.

Testing

Lint and *test* the scripts, and since it's such an important practice we will discuss this in greater detail in "Linting and Testing Scripts".

Let's now move on to making scripts safe(r) by linting them while developing and testing them before you distribute them.

Linting and Testing Scripts

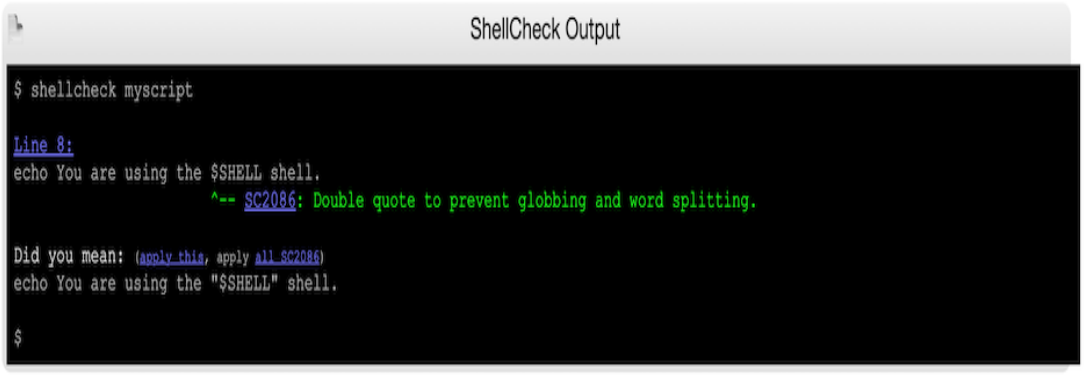
While you're developing, you want to check and lint your scripts, making sure that you're using commands and instructions right. There's a nice way to do that, depicted in Figure 3-10, a program called shellcheck; you can download and install it locally or you can use also use the online version via shellcheck.net. Also, consider formatting your script with shfmt. It automatically fixes some issues that can be reported later on by `shellcheck`.

ShellCheck

finds bugs in your shell scripts.

You can `cabal`, `apt`, `dnf`, `pkg` or `brew install` it locally right now.

Paste a script to try it out:



ShellCheck is...

- GPLv3: free as in freedom
- available on GitHub (as is this website)
- already packaged for your distro or package manager
- supported as an integrated linter in major editors
- available in CodeClimate, Codacy and CodeFactor to auto-check your GitHub repo
- written in Haskell, if you're into that sort of thing.

Figure 3-10. A screenshot of the online shellcheck tool

And further, before you check your script into a repo, consider using `bats` to test it: bats stands for "Bash Automated Testing System" and allows you to define test files as a Bash script with special syntax for test cases. Each test case is simply a Bash function with a description and you would typically invoke these scripts as part of a CI pipeline, for example as a GitHub action.

Now let's put our good practices for script writing, linting, and testing into practice. Let us implement the example script we specified in the beginning of this section.

End-to-end Example: GitHub User Info Script

In this end-to-end example we bring all of above tips and tooling together to implement our example script that is supposed to take a GitHub user handle and print out a message that contains what year the user joined, along with their full name.

This is how one implementation looks like, taking the good practices into account. Store the following in a file called `gh-user-info.sh` and make it executable:

```
#!/usr/bin/env bash

set -o errexit
set -o errtrace
set -o nounset
set -o pipefail

### Command line parameter:
targetuser="${1:-mhausenblas}" ❶

### Check if our dependencies are met:
if ! [ -x "$(command -v jq)" ]
then
  echo "jq is not installed" >&2
  exit 1
fi
```

```
### Main:
githubapi="https://api.github.com/users/"
tmpuserdump="/tmp/ghuserdump_$targetuser.json"

result=$(curl -s $githubapi$targetuser) ❷
echo $result > $tmpuserdump

name=$(jq .name $tmpuserdump -r) ❸
created_at=$(jq .created_at $tmpuserdump -r)

joinyear=$(echo $created_at | cut -f1 -d"-") ❹
echo $name joined GitHub in $joinyear ❺
```

❶ Provide a default value to use if user doesn't supply us with one.

❷ Using `curl`, access the GitHub API to download the user info as a JSON file and store it in a temporary file (next line).

❸ Using `jq` pull out the fields we need. Note that the `created_at` field has a value that looks something like `"2009-02-07T16:07:32Z"`.

❹ Using `cut` to extract the year from `created_at` field in the JSON file.

❺ Assemble the output message and print to screen.

Now let's run it with the defaults:

```
$ ./gh-user-info.sh
Michael Hausenblas joined GitHub in 2009
```

Congratulations, you now have everything at our disposal to use the shell, both interactively on the prompt and for scripting. Before we wrap up, take a moment to think about the following, concerning our `gh-user-info.sh` script:

- What if the JSON blob the GitHub API returns is not valid? What if we encounter a 500 HTTP error? Maybe adding a message along

the line “try later” is more useful if there’s nothing the user can do themselves.

- For the script to work you need network access, otherwise the `curl` call will fail. What could you do about a lack of network access? Informing the user about it and suggest what they can do to check networking may be an option.
- Think about improvements around dependency checks, for example, we implicitly assume here that `curl` is installed. Can you maybe add a check that makes the binary variable and falls back to `wget`?
- How about adding some usage help? Maybe, if the script is called with an `-h` or `--help` parameter, show a concrete usage example and the options that users can use to influenc the execution (ideally, including defining default values used).

You see now that, although this script looks good and works in most cases, there’s always something you can improve, make the user epxerience better and making the script more robust, failing with helpful and actionable user messages. In this context, consider using frameworks such as bashing, rerun, or rr to improve modularity.

Conclusion

In this chapter we focused on working with Linux in the terminal, a textual user interface. We discussed shell terminology, provided a hands-on introduction to using the shell basics, and reviewed common tasks, and how you can improve your shell productivity using modern variants of certain commands.

Then, we looked at modern, human-friendly shells, specifically at `fish`, how to configure and use it. Further, we covered terminal multiplexer by using `tmux` as the hands-on example, enabling you to work with multiple local or remote sessions, windows, and panes.

Lastly, we discussed automating tasks by writing safe and portable shell scripts, including linting and testing said scripts. Remember that shells effectively are command interpreters, and as with any kind of language you have to practice to get fluent. Having said this, now that you're equipped with the basics of using Linux from the command line, you can already work with the majority of Linux-based systems out there, be it an embedded system or a cloud VM. In any case, you will find a way to get hold of a terminal and issue commands interactively or via executing scripts.

If you want to dive deeper into the topics discussed in the chapter, here are a some further resources:

1. Terminals:
    - The TTY demystified
    - The terminal, the console and the shell - what are they?
    - What is a TTY on Linux? (and How to Use the tty Command)
    - Your terminal is not a terminal: An Introduction to Streams
2. Shells:
    - Unix Shells: Bash, Fish, Ksh, Tcsh, Zsh
    - Comparison of command shells
    - Bash vs Zsh
    - Ghost in the Shell – Part 7 – ZSH Setup
3. Terminal multiplexer:
    - A tmux Crash Course
    - A Quick and Easy Guide to tmux

- How to Use tmux on Linux (and Why It's Better Than Screen)
- The Tao of tmux
- tmux 2 Productive Mouse-Free Development

4. Shell scripts:

- Shell Style Guide
- Bash Style Guide
- Bash best practices
- Bash scripting cheatsheet
- Writing Bash Scripts that are not only Bash: Checking for Bashisms and testing with Dash

With the shell basics at our disposal we now turn our focus to access control and enforcement in Linux.

Chapter 4. Access Control

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 4th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at modern-linux@pm.me.

After the wide scope in the previous chapter on all things shell and scripting, we now focus on one specific and crucial security aspect in Linux. In this chapter we discuss the topic of users and controlling access to resources in general and files in particular.

One question that immediately comes to mind in such a multi-user setup is ownership. A user may own, for example, a file. They are allowed to read from the file, write to the file, and, say, also delete it. Given that there are other users on the system as well: what is that user are allowed to do, and how is this defined and enforced? There are also activities which you might not necessarily associate with files in the first place. For example, a user may (or may not) be allowed to change networking-related settings.

To get a handle on the topic, we will first have a look at the fundamental relationship between users, processes, and files, from an access perspective. We will also review sandboxing and access control types. Next, we focus on what is the definition of a Linux user, what users can do, and how to manage users either locally and from a central place."

Then, we move on to the topic of permissions where we look at how to control access to files and how processes are impacted by such restrictions.

We wrap up this chapter covering a range of advanced Linux features in the access control space including capabilities, seccomp profiles, and ACLs. To round things off we'll provide some security good practices around permissions and access control.

With that, let's jump right into the topic of users and resource ownership, laying the basis for the rest of the chapter.

Basics

Before we get into access control mechanism let us step back a little and take a birds eye view on the topic. This will help us to establish some terminology and clarify relationships between the main concepts.

Resources and Ownership

Linux is a multi-user operating system and as such has inherited the concept of a user ("Users") from Unix. Users can be a human user, or a process that does something in the name of a user. Then, there are resources (which we will simply refer to as files), which are any hardware or software components available to the user. In the general case, we will refer to resources as files, unless we explicitly talk about access to other kinds of resources, for example, as the case with syscalls. In Figure 4-1 you see the high-level relationships between users, processes, and files in Linux.

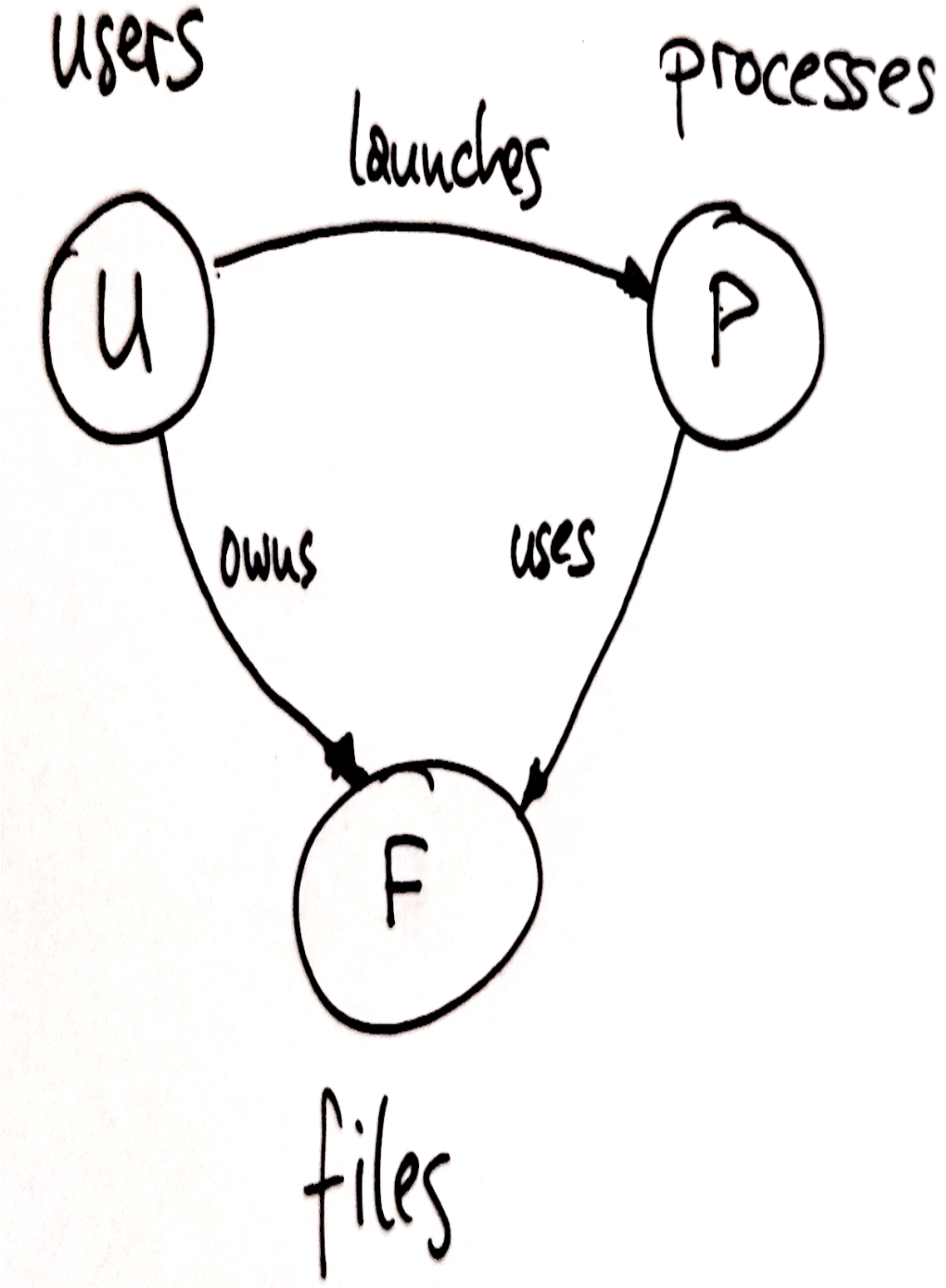

users
processes
launches
U
P
owns
uses
F
files

Figure 4-1. Users, processes, and files in Linux

- Users launch processes and own files. A process is a program (executable file) that the kernel has loaded into main memory and runs.
- Files have owners, by default, the user who creates the file owns it.
- Processes use files for communication and persistency. Of course, users indirectly also use files, but they need to do so via processes.

This depiction of the relationships between users, processes, and files is of course a very simplistic view but it allows us to understand the actors and their relationships and will come in handy later on when we discuss the interaction between different players here in greater detail.

Let's have a look at the execution context of a process, first. That is, addressing the question how restricted a process is.

Sandboxing

A term that we often come accross when talking about access to resources is sandboxing. This is a vaguely defined term and can refer to a lot of different technologies, from jails to containers to virtual machines which can be managed either in the kernel or in user land. Usually there is something that runs in the sandbox—typically some application—and the supervising mechanism enforces a certain degree of isolation between the sandboxed process and the hosting environment. If all of that sounds rather theoretical, I ask you for a little bit of patience. We will see sandboxing in action later in this chapter in "Seccomp Profiles" and then again in XREF HERE when we talk about VMs and containers.

With the basic understanding of resources, ownership, and access to said resources in your mind, let's talk briefly about some conceptual ways to go about access control.

Types of Access Control

One aspect of access control is the nature of the access itself. Does a user or process directly access a resource, maybe in an unrestricted manner? Or maybe there is a clear set of rules about what kind of resources (files or syscalls) a process can access, under what circumstances. Or maybe the access itself is even recorded.

Conceptualy, there are different access control types. The two most important and relevant to our discussion in the context of Linux are discretionary and mandatory access control:

Discretionary Access Control (DAC)

> With DAC the idea is to restrict access to resources based on the identity of the user. It's discretionary in the sense that a user having certain permissions can pass them on to other users.

Mandatory (MAC)

> MAC is based on a hierarchical model representing security levels. The users get a clearance level assigned and resources are assigned a security label. Users can only access resources corresponding to a clearance level equal to (or lower than) their own. In a MAC model, an admin strictly and exclusively controlls access, setting all permissions. In other words, users cannot set permissions themselves, even in the case that they own the resource

In addition, Linux traditionally has an all or nothing attitude, that is, you are either a superuser that has the power to change everything, or you are a normal user with limited access. There was no way to assign a user or process certain privileges such as "this process is allowed to change networking settings", you had to give it `root` access. This, naturally, has a concrete impact on a system that is breached: an attacker can misuse these wide privileges easily. We will revisit this topic in "Advanced Permission Management" and see how modern Linux features can help overcome this binary worldview, allowing for more fine-grained management of privileges.

Probably the best-known implementation of MAC for Linux is SELinux. It was developed to meet the high security requirements of government agencies and is usually exactly used in these environments since the usability suffers from the strict rules. Another option for MAC, included in the Linux kernel since version 2.6.36, and rather popular in the Ubuntu family of Linux distribution is AppArmor.

Let us now move on to the topic of users and how to manage them in Linux.

Users

In Linux we often distinguish between two types of user accounts, from a purpose or intended usage point of view:

- So called system users or system accounts. Typically, programs (sometimes called daemons) use these types of accounts to run background processes. The services provided by these programs can be part of the operating system such as networking (`sshd` for example) or on the application layer, for example `mysql` in case of a popular relational database.
- Regular users, that is, a human user that interactively uses Linux via the shell, for example.

The distinction between system and regular users is less of a technical one but more an of organizational construct. To understand that we first have to introduce the concept of a user ID (UID), a 32 Bit numerical value managed by Linux.

Linux identifies users via a UID, with a user belonging to one or more groups identified via a group ID (GID). There is a special kind of user with the UID 0 and that user is usually called `root`. This “superuser” is allowed to do anything, that is, no restriction apply. Usually you want to avoid to directly use the `root` user since that’s just too much power to have and you can easily destroy a system if you’re not careful. We will get back to this later in the chapter.

Different Linux distributions have their own ways to decide how to manage the UID range. For example, `systemd` powered distributions, see also XREF HERE, have the following convention (simplified in the following):

- UID 0 is `root`.
- UID 1 to 999 are reserved for system users.
- UID 65534 is user `nobody`, used, for example, for mapping remote users to some well-known ID, as the case with XREF HERE.
- UID 1000 to 65533 and 65536 to 4294967294 are regular users.

To figure out your own UIDs you can use the (surprise!) `id` command like so:

```
$ id -u
2016796723
```

Now that you know the basics about Linux users, let's see how you can manage users.

Managing Users Locally

The first option and traditionally the only one available is that of managing users locally. That is, only information local to the machine is used and user related information is not shared across a network of machines.

For local user management, Linux uses a simple file-based interface with a somewhat confusing naming scheme which is a historic artifact we have to live with, unfortunately. Table 4-1 lists the four files that, together, implement user management.

Table 4-1. Reference of local user ma

n
a
g
e
m
e
n
t
f
i
l
e
s

Purpose	File
user database	`/etc/passwd`
group database	`/etc/group`
user passwords	`/etc/shadow`
group passwords	`/etc/gshadow`

Think of `/etc/passwd` as a kind of mini user database keeping track of user names, UIDs, group membership as well as other data such as home directory and login shell used, for regular users. Let's have a look at a concrete example:

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash ❶
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin ❷
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
```

```
syslog:x:104:110::/home/syslog:/usr/sbin/nologin
mh9:x:1000:1001::/home/mh9:/usr/bin/fish ❸
```

❶ The root user, has UID 0.

❷ A system account (the `nologin` gives it away, see more below).

❸ My user account.

Let's have a closer look at one of the lines in `/etc/passwd` to understand the structure of a user entry in detail:

```
root:x:0:0:root:/root:/bin/bash
^    ^ ^ ^ ^    ^     ^
|    | | | |    |     └─ ❶
|    | | | |    └─ ❷
|    | | | └─ ❸
|    | | └─ ❹
|    | └─ ❺
|    └─ ❻
└─ ❼
```

❶ The login shell to use. To prevent interactive logins, use `/sbin/nologin`.

❷ The user's home directory, defaults to `/`.

❸ User information such as full name or contact data like phone number. Often also known as GECOS field, nowadays seldomly used.

❹ The user's primary group (GID), see also `/etc/group`.

❺ The UID. Note that Linux reserves UIDs below 1000 for system usage.

❻ The user's password with the `x` character meaning that the (encrypted) password is stored in `/etc/shadow` which is the default these days.

❼ The username, must be 32 characters or less long.

One thing we notice is absent in `/etc/passwd` is the one thing we would expect to find there, going by the name: the password. Passwords are, for historic reasons, stored in a file called `/etc/shadow`. While every user can read `/etc/passwd`, for `/etc/shadow` you usually need `root` privileges.

To add a user you can use the adduser command as follows:

```
$ sudo adduser mh9
Adding user `mh9' ...
Adding new group `mh9' (1001) ...
Adding new user `mh9' (1000) with group `mh9' ...
Creating home directory `/home/mh9' ... ❶
Copying files from `/etc/skel' ... ❷
New password: ❸
Retype new password:
passwd: password updated successfully
Changing the user information for mh9
Enter the new value, or press ENTER for the default ❹
        Full Name []: Michael Hausenblas
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] Y
```

❶ The `adduser` command creates a home directory.

❷ It also copies a bunch of default config files into the home dir.

❸ Need to define a password.

❹ Provide optional GECOS info.

If you want to create a system account, pass in the `-r` option. This will disable the ability to use a login shell and also avoid home directory

creation. Also, see `/etc/adduser.conf` for configuration, such as UID/GID range.

In addition to users, Linux also has the concept of groups which in a sense is just a collection of one or more users. Any regular user belongs to one default group, but can be member of additional groups. You can find out about groups and mappings via the `/etc/group` file:

```
$ cat /etc/group ❶
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:syslog
...
ssh:x:114:
landscape:x:115:
admin:x:116:
netdev:x:117:
lxd:x:118:
systemd-coredump:x:999:
mh9:x:1001: ❷
```

❶ Display the content of the group mapping file.

❷ An example group of my user with the GID 1001.

With this basic user concept and management under our belt, we move on to a potentially better way to manage users in a professional setup, allowing for scale.

Centralized User Management

If you have more than one machine or server that you wish to or have to manage users for, say, in a professional setup, then managing users locally quickly becomes old. You want a centralized way to maange users that you can apply locally, to one specific machine. There are a couple of approaches available to you, depending on your requirements and (time) budget:

- Directory based: using Lightweight Directory Access Protocol (LDAP), a decades-old suite of protocols nowadays formalized by IETF that defines how to access and maintain a distributed directory over Internet Protocol (IP). You can run an LDAP server yourself, for example, using projects like Keycloack or outsource this to a cloud provider, such as Azure Active Directory.
- With Kerberos it's possible to authenticate users a network (we will look at Kerberos in detail on XREF HERE).
- Using config management systems such as Ansible, Chef, Puppet, or SaltStack to consistently create users across machine.

The actual implementation is often dictated by the environment. That is, a company might already be using LDAP and hence the choices might be limited. The details of the different approaches and pros and cons are, however, beyond the scope of this book.

Permissions

In this section we first go into detail concerning Linux file permissions, which are central to how access control works, and then we look at permissions around processes. That is, we review runtime permissions and how they are derived from file permissions.

File Permissions

File permissions are core to Linux baseline concept of access to resources, since everything is a file in Linux, more or less. Let's first review some terminology and then discuss the representation of the metadata around file access and permissions in detail.

There are three types or scopes of permissions, from narrow to wide:

- The *user* is the owner of the file.
- A *group* has one or more members.

- The *other* category is for everyone else, other than either the owner or a group member.

Further, three types of access:

- Read (`r`): for a normal file, allows a user to view the contents of the file. For a directory, allows a user to view the names of files in the directory.
- Write (`w`): for a normal file, allows a user to modify and delete the file. For a directory, allows a user create, rename, and delete files in the directory.
- Execute (`x`): for a normal file, allows a user to execute a file if the user also has read permissions on the file. For a directory, allows a user to access, and access file infos in the directory, effectively permitting to change into it (`cd`) or list its content (`ls`).

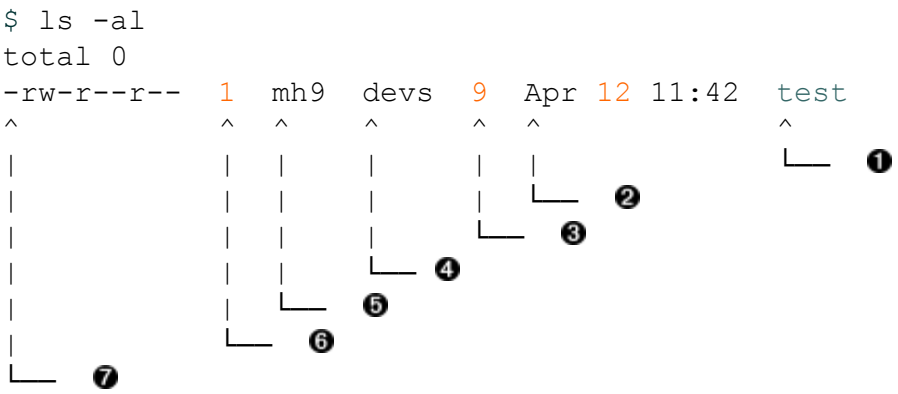
OTHER FILE ACCESS BITS

I listed `r`/`w`/`x` as the three file access types but in practices you will find others as well when you do an `ls`:

- `s` is the setuid/setgid permission applied to an executable file. That is, a user running it inherits the effective privileges of the owner or group
- `t` is the sticky bit which is only relevant for directories. If set, it prevents non-root users from deleting files in it, unless said user owns the directory/file.

There are also special settings in Linux available via the `chattr` (change attribute) command, but this is beyond the scope of this chapter.

Let's see file permissions in action (note that the spaces you see here in the output of the `ls` command have been expanded for better readability):



❶ File name

❷ Last modified time stamp

❸ File size in bytes

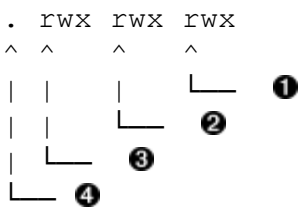
❹ Group the file belongs to

❺ File owner

❻ Number of hard links

❼ File mode

When zooming in on the *file mode*, that is, the file type and permissions referred to as <7> in above snippet, we further have fields with the following meaning:



❶ Permissions for others.

❷ Permissions for the group.

❸ Permissions for the file owner.

❹ The file type (Table 4-2) .

The first field in the file mode represents the file type, see Table 4-2 for details. The remainder of the file mode encodes the permissions set for various targets, from owner to everyone, as listed in Table 4-3.

Table 4-2. File types used in mode

Symbol	Semantics

`-`	a regular file (such as when you do `touch abc`)
`b`	block special file
`c`	character special file
`C`	high performance (contiguous data) file
`d`	a directory
`l`	a symbolic link
`p`	a named pipe (create with `mkfifo`)
`s`	a socket
`?`	some other (unknown) file type

There are some other (older or obsolete characters) such as `M` or `P` used in the position `0` which you can by and large ignore. If you're interested in what they mean, run `info ls -n "What information is listed"`.

In combination, these permissions in the file mode define what is allowed for each of the target set (user, group, everyone else) as shown in Table 4-3, checked and enforced through access.

Table 4-3. File permissions

Pattern	Effective permission	Decimal representation
`---`	none	0
`--x`	execute	1
`-w-`	write	2

`-wx`	write and execute	3
`r--`	read	4
`r-x`	read and execute	5
`rw-`	read and write	6
`rwx`	read, write, execute	7

Let's have a look at a few examples:

- `755` … full access for owner, read and execute for everyone else
- `700` … full access by its owner, none for everyone else
- `664` … read/write access for owner and group, read-only for others
- `644` … read/write for owner, read-only for everyone else
- `400` … read-only by its owner

The `664` has a special meaning, on my system. When I create a file, that's the default permission it gets assigned. You can check that with the umask command which in my case gives me `0002`.

The `setuid` permissions are used to tell the system to run an executable as the owner, with the owner's permissions. If a file is owned by `root` that can cause issues.

You can change the permissions of a file using `chmod`. Either you specify the desired permission settings explicitly (such as `644`) or using shortcuts (for example `+x` to make it executable). But how does that look in practice?

Let's make a file executable with `chmod`:

```
$ ls -al /tmp/masktest
-rw-r--r-- 1 mh9 dev 0 Aug 28 13:07 /tmp/masktest ❶
```

```
$ chmod +x /tmp/masktest ❷

$ ls -al /tmp/masktest
-rwxr-xr-x 1 mh9 dev 0 Aug 28 13:07 /tmp/masktest ❸
```

❶ Initially the file permissions are r/w for the owner and read-only for everyone else, aka `644`.

❷ Make the file executable.

❸ Now the file permissions are r/w/x for the owner and r/x for everyone else, aka `755`.

In Figure 4-2 you see what is going on under the hood. Note that you might not want to give really everyone the right to execute the file so a `chmod 744` might have been better here, giving only the owner the correct permissions while not changing it for the rest. We will further discuss this topic in "Good Practices".

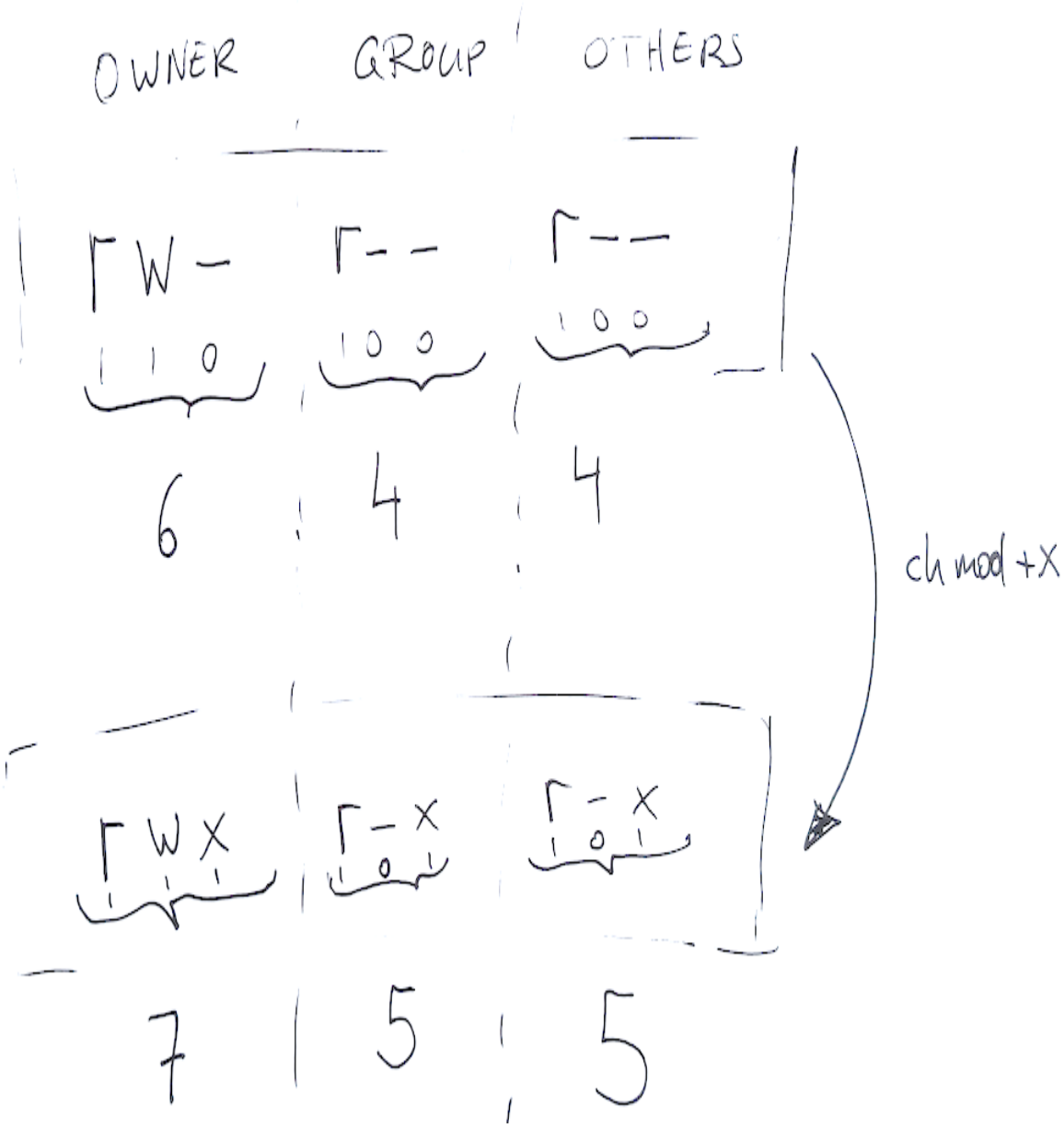


Figure 4-2. Making a file executable and how the file permissions change with it

You can also change the ownership using `chown` (and `chgrp` for the group):

```
$ touch myfile
$ ls -al myfile
-rw-rw-r-- 1 mh9 mh9 0 Sep 4 09:26 myfile ❶

$ sudo chown root myfile ❷
-rw-rw-r-- 1 root mh9 0 Sep 4 09:26 myfile
```

❶ The file `myfile` I created and own.

❷ After `chown`, now `root` owns that file.

With the basic permission management concluded let's have a look at some more advanced techniques in this space.

Process Permissions

So far we have focused on how human users access files and what the respective permissions in play are. Now we shift the focus to processes. In "Resources and Ownership" we talked about how users own files as well as how processes use files. This begs the question: what are the relevant permissions, from a process point of view?

As documented in credentials(7) there are different users IDs relevant in the context of runtime permissions:

Real UID

> The *real* UID is the UID of the user that launched the process. It represents process ownership in terms of human user. The process itself can obtain its real UID via getuid(2) and you can query it via the shell using `stat -c "%u %g" /proc/$pid/.`

Effective UID

> The Linux kernel uses the *effective* UID to determine permissions the process has when accessing shared resources such as message queues. On traditional Unix systems, they are also used for file access. Linux, however, used to use a dedicated filesystem UIDs (see below) for file access permissions, still kept around for compatibility reasons. A process can obtain its effective UID via geteuid(2).

Saved set-user UID

Saved set-user-ID are used in `suid` cases where a process can assume privileges by switching its effective UID between the real UID and saved set-user-ID. For example, for a process to be allowed to use certain network ports (XREF HERE) it needs elevated privileges, for example, run as `root`. A process can get its saved set-user-ID getresuid(2).

Filesystem UID

These Linux-specific IDs are used to determine permissions for file access. This UID was initially introduced to support use cases where a file server would act on behalf of a regular user, while isolating the process from signals by said user. Programs don't usually directly manipulate this UID. The kernel keeps track of when the effective UID is changed and automatically changes the filesystem UID with it. This means that usually the filesystem UIDs is the same as the effective UID, but can be changed via setfsuid(2). Note that technically this UID is no longer necessary since kernel v2.0 but is still supported, for compatibility.

Initially, when a child process is created via `fork(2)` it inherits copies of its parent's UIDs and during an `execve(2)` syscall, the process's real UID is preserved whereas the effective and saved set-suer IDs may change.

For example, when you run the `passwd` command, your effective UID is your UID, let's say 1000. Now, `passwd` has `suid` set enabled, which means when you run it, your effective UID is 0 (aka `root`). There are also other ways to influence the effective UID, for example using `chroot` and other sandboxing techniques.

NOTE

POSIX threads require that credentials are shared by all threads in a process. However, Linux maintains at the kernel level separate user and group credentials for each thread.

In addition to file access permissions, the kernel uses process UIDs for other things, including but not limited to:

- Establishing permissions for sending signals, for example, to determine what happens when you do a `kill -9` for a certain process ID. We will get back to this in XREF HERE,
- Permission handling for scheduling and priorities (for example, `nice`).
- Checking resource limits, which we will discuss in detail in the context of containers in XREF HERE.

While it can be straightforward to reason with effective UID in the context of `suid`, once capabilities come into play it can be more challenging.

Advanced Permission Management

While we so far focused on widely used mechanisms, the topics in this section are in a sense advanced and not necessarily something you would consider in a casual or hobby setup. For professional usage, that is production use cases where business critical workloads are deployed, you should definitely be at least aware of the following advanced permission management approaches.

Capabilities

In Linux, as traditionally the case in Unix systems, the `root` user has no restrictions when running processes. In other words, the kernel only distinguishes between two cases:

- Privileged processes, bypassing the kernel permission checks, with an effective UID of 0 (aka `root`).
- Unprivileged processes, with a non-zero effective UID, for which the kernel does permission checks as discussed in "Process Permissions".

With the introduction of the capabilities syscall in kernel v2.2, this binary worldview has changed: the privileges traditionally associated with `root` are now broken down into distinct units that can be independently assgined on a per-thread level.

In practice, the idea is that a normal processes has zero capabilities, controlled by the permissions discussed in the previous section. You can assign capabilities to executables (binaries and shell scripts) as well processes to gradually add privileges necessary to carry out a task, see also our discussion in "Least Privileges".

Now, a word of caution: capabilities are generally only relevant for system-level tasks. In other words: most of the time you won't necessarily depend on it.

In Table 4-4 you can see some of the more widely used capabilities.

Table 4-4. Examples of useful capabil

i
t
i
e
s

Capability	Semantics
`CAP_CHOWN`	Allows user to make arbitrary changes to files UIDs/GIDs
`CAP_KILL`	Allows sending of signals to processes belonging to other users
`CAP_SETUID`	Allows changing the UID
`CAP_SETPCAP`	Allows to set the capabilities of a running process
`CAP_NET_ADMIN`	Allows various network-related actions such as interface config
`CAP_NET_RAW`	Allows to use RAW and PACKET sockets
`CAP_SYS_CHROOT`	Allows to call `chroot`
`CAP_SYS_ADMIN`	Allows system admin operations including mounting filesystems
`CAP_SYS_PTRACE`	Allows to use `strace` to debug processes
`CAP_SYS_MODULE`	Allows to load kernel modules

Let's us now see capabilities in action. For starters, to view, you can use commands as shown in the following (output edited to fit):

```
$ capsh --print ❶
Current: =
Bounding set =cap_chown,cap_dac_override,cap_dac_read_search,
cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,
...

$ grep Cap /proc/$$/status ❷
```

```
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 000001ffffffffff
CapAmb: 0000000000000000
```

❶ Overview of all capabilities on the system.

❷ Capabilities for the current process (the shell).

You can manage capabilities in a fine-grained manner, that is, on a per-file basis with getcap and setcap, with the details and good practices beyond the scope of this chapter.

Capabilities help to transition from an all-or-nothing approach to finer-grained privileges on a file basis. Let's now move on to a different advanced access control topic, that of the sandboxing technique of seccomp.

Seccomp Profiles

Secure computing mode (seccomp) is a Linux kernel feature available since 2005. The basic idea behind this sandboxing technique is that, using a dedicated syscall called `seccomp(2)`, you can restrict the syscalls a process can use.

While you might find it inconvenient to manage seccomp yourself directly, there are ways to use it without too much hassle. For example, in the context of containers (XREF HERE) both Docker and Kubernetes support seccomp.

Let's now have a look at an extension of the traditional, granular file permissions.

Access Control Lists (ACL)

With access control list (ACL) we have a flexible permission mechanism in Linux that you can use on top of or in addition to the more "traditional" permissions discussed in "File Permissions". ACLs address a shortcomming of traditional permissions in that they allow you to grant permissions for a user or a group not in the group list of a user.

To check if your distribution supports ACLs, you can use `grep -i acl /boot/config*` where you'd hope to find a `POSIX_ACL=Y` somewhere in the output to confirm it. In order to use ACL for a filesystem, it must be enabled at mount time, using the `acl` option. The docs reference on acl has a lot of useful details.

We won't go into greater detail here on ACL since it's slightly outside of the scope of the book, however, being aware of it and knowing where to start can be benefical, should you come across them in the wild.

With that, let us review some good practices for access control.

Good Practices

Here are some security "good practices" in the wider context of access control. While some of them might be more applicable in professional environments, everyone should at least be aware of them.

Least Privileges

The least privileges principle says, in a nutshell, that a person or process should only have the necessary permissions to achieve a given task. For example, if an app doesn't write to a file, then it only needs read access. In the context of access control, you can practice least privileges in two ways:

- In "File Permissions" we saw what happens when using `chmod +x`. In addition to the permissions you intended, it assigns also other users some. That is, using explicit permissions via the numeral mode is better than symbolic mode. In other words: while the latter is more convenient it's less strict.

- Avoid running as root as much as you can. For example, when you need to install something, then you should be using `sudo` rather than logging in as `root`.

Avoid `setuid`

Take advantage of capabilities rather than to relying on `setuid` which is like a sledgehammer and offers attackers a great way to take over your system.

Auditing

Auditing is the idea that you record actions (along with who carried them out) in a way that the resulting log can't be tempered with. You can then use this read-only log to verify who did what, when. We will dive into this topic in XREF HERE.

Conclusion

Now that you know how Linux manages users, files and access to resources, you have everything at your disposal to carry out routine tasks safely and securely.

In this chapter we first had a look at the fundamental relationship between users, processes, and files in the context of the multi-user operating system that Linux is. We reviewed access control types, defined what users in Linux are, what they can do and how to manage them both locally and centrally. We then moved on to the topic of file permissions and how to manage these. The last part of the chapter we dedicated to advanced permissions techniques and tecnologies available in Linux, including capabilities and seccomp profiles, as well as a short compilation of good practices around access control related security.

If you want to dive deeper into the topics of this chapter, here are some resources:

1. A Survey of Access Control Policies by Amanda Crowell.

2. Capabilities:
    - Linux Capabilities In Practice
    - Linux Capabilities: making them work
3. Secomp:
    - A seccomp overview
    - Sandboxing in Linux with zero lines of code
4. Access Control Lists:
    - POSIX Access Control Lists on Linux
    - Access Control Lists via ArchLinux
    - An introduction to Linux Access Control Lists (ACLs) via Red Hat

Remember that security is an ongoing process, so you want to make sure to keep an eye on users and files, something we will go into greater detail in XREF HERE and XREF HERE, but for now let's move on to the topic of filesystems.

Chapter 5. Filesystems

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at modern-linux@pm.me.

In this chapter we focus on files and filesystems. The Unix concept of "everything is a file" lives on in Linux and while that's not true 100% of the time, most resources in Linux are indeed files. Files can be everything from the content of the letter you write to your school, to the funny GIF you download (from an obviously safe and trusted site).

There are other things that are also exposed as files in Linux. While you may not associate these resources with files, you can access them with the same methods and tools you know from regular files. For example, the kernel exposes certain runtime information (as discussed in "Process Management") about a process, such as its PID or the binary used to run the process.

What all these things have in common is a standardized, uniform interface: opening a file, gathering information about a file, writing to a file, and so forth. In Linux, filesystems provide this uniform interface. This interface together with the fact that Linux treats files as a streams of bytes, without any expecations about the structure enables us to build tools that work with a range of different file types.

In addition, the uniform interface that filesystems provide reduces your cognitive load, making it faster for you to learn how to use Linux.

In this chapter, we first define some relevant terms. Then, we look at how Linux implements the "everything is a file" abstraction. Next, we review special-purpose filesystems the kernel uses to expose information about processes or devices. We then move on to regular files and filesystems, something you would typically associate with documents, data, and programs. We compare filesystem options and discuss common operations.

Basics

Before we get into the filesystem terminology, let us first make some implicit assumptions and expectations about filesystems more explicit:

- While there are exceptions, most widely used filesystems today are hierarchical. That is they provide the user with a single filesystem tree, starting at the root (`/`).
- In the filesystem tree you find two different types of objects: directories and files. Think of directories as a organizational unit, allowing you to group files. If you like to apply the tree analogy: directories are the nodes in the tree, whereas the leafs can be either files or directories.
- You can navigate a filesystem by listing the content of a directory (`ls`), change into that directory (`cd`) and print the current working directory (`pwd`).
- Permissions are built-in: as discussed in "Permissions" one of the attributes a filesystem captures is ownership. Consequently, ownership enforces access to files and directories via the assigned permissions.
- Generally, filesystems are implemented in the kernel.

> **NOTE**
>
> While filesystems are usually, for performance reasons, implemented in kernel space, there's also an option to implement them in user land: Filesystem in Userspace (FUSE), see also the project site.

With this informal high-level explanation out of the way, we now focus on some more crisp definition of terms that you will need to understand.

Drive

A (physical) block device such as a Hard Disk Drive (HDD) or a Solid-state Drive (SSD). In the context of virtual machines a drive can be also emulated. For example, `/dev/sda` or `/dev/sdb`.

Partition

You can logically split up drives into partitions, a set of storage sectors. For example, you may decide to create two partitions on your HDD which then would show up as `/dev/sdb1` and `/dev/sdb2`.

Volume

A volume is somewhat similar to a partition, but it is more flexible and it is also formatted for a specific filesystem. We will discuss volumes in detail in "Logical Volume Manager (LVM)".

Super block

When formatted, filesystems have a special section in the beginning that captures the metadata of the filesystem. This includes things like filesystem type, blocks, state, how many inodes per block, etc.

Inodes

> In a filesystem, inodes store metadata about files, such as size, owner, location, date, and permissions. However, inodes do not store the filename and the actual data. This is kept in directories, which really are just a special kind of regular file, mapping inodes to filenames.

That was a lot of theory, so let's see these concepts in action, now. First, here's how to see what drives, partitions, and volumes are present in your system:

```
$ lsblk --exclude 7 ❶
NAME                      MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda                         8:0    0 223.6G  0 disk              ❷
├─sda1                      8:1    0   512M  0 part /boot/efi    ❸
└─sda2                      8:2    0 223.1G  0 part              ❹
  ├─elementary--vg-root   253:0    0 222.1G  0 lvm  /
  └─elementary--vg-swap_1 253:1    0   976M  0 lvm  [SWAP]
```

❶ List all block devices but exclude pseudo (loop) devices.

❷ We have a disk drive called `sda` with some 223 GB overall.

❸ There are two partitions here with `sda1` being the boot partition.

❹ The second partition called `sda2` contains two volumes, see "Logical Volume Manager (LVM)" for details.

Now that we have an overall idea of the physical and logical setup, let's have a closer look at the filesystems in use:

```
$ findmnt -D -t nosquashfs ❶
SOURCE                          FSTYPE     SIZE  USED  AVAIL USE% TARGET
udev                            devtmpfs   3.8G     0   3.8G   0% /dev
tmpfs                           tmpfs    778.9M  1.6M 777.3M   0% /run
/dev/mapper/elementary--vg-root ext4     217.6G 13.8G 192.7G   6% /
tmpfs                           tmpfs      3.8G 19.2M   3.8G   0% /dev/shm
tmpfs                           tmpfs        5M    4K     5M   0% /run/lock
tmpfs                           tmpfs      3.8G     0   3.8G   0% /sys/fs/cgroup
/dev/sda1                       vfat       511M    6M 504.9M   1% /boot/efi
tmpfs                           tmpfs    778.9M   76K 778.8M   0% /run/user/1000
```

❶ List filesystems but exclude squashfs types (specialized read-only compressed filesystem originally developed for CDs, now also for snapshots).

We can go a step further and look at individual filesystem objects such as directories or files:

```
$ stat myfile
  File: myfile
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file ❶
```

```
  Device: fc01h/64513d    Inode: 555036      Links: 1 ❷
  Access: (0664/-rw-rw-r--)  Uid: ( 1000/     mh9)   Gid: ( 1001/     mh9)
  Access: 2021-08-29 09:26:36.638447261 +0000
  Modify: 2021-08-29 09:26:36.638447261 +0000
  Change: 2021-08-29 09:26:36.638447261 +0000
   Birth: 2021-08-29 09:26:36.638447261 +0000
```

❶ File type information.

❷ Information about deivce and inode.

In the previous command, if we would have used `stat .` we would have gotten the respective directory file info including its inode, number of blocks used etc.

The Table 5-1 lists some basic filesystem commands that allow you to explore query the concepts we introduced earlier.

Table 5-1. Selection of flow-level fail-safe system and

b
l
o
c
k
d
e
v
i
c
e
c
o
m
m
a
n
d
s

Command	Use case
`lsblk`	List all block devices
`fdisk`, `parted`	Manage disk partitions
`blkid`	Show block device attributes such as UUID
`hwinfo`	Show hardware info
`file -s`	Show filesystem and partition info
`stat`, `df -i`, `ls -i`	Show and list inode related info

Another term you come across in the context of filesystems is that of links. Sometimes you want to refer to files with different names or provide shortcuts. There are two types of links in Linux:

- Hard links reference inodes and can't refer to directories and also do not work across filesystems.
- Symbolic links or symlinks are special files with their content being a string representing the path of another file.

Now let's see links in action (some outputs shortened):

```
$ ln myfile somealias ❶
$ ln -s myfile somesoftalias ❷

$ ls -al *alias ❸
-rw-rw-r-- 2 mh9 mh9 0 Sep  5 12:15 somealias
lrwxrwxrwx 1 mh9 mh9 6 Sep  5 12:45 somesoftalias -> myfile

$ stat somealias ❹
  File: somealias
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: fd00h/64768d    Inode: 6302071     Links: 2
...
$ stat somesoftalias ❺
  File: somesoftalias -> myfile
  Size: 6               Blocks: 0          IO Block: 4096   symbolic link
Device: fd00h/64768d    Inode: 6303540     Links: 1
...
```

❶ Create hard link to `myfile`.

❷ Create soft link to same file (notice the `-s` option).

❸ List files, notice the different file types and rendering of name.

❹ Show file details of hard link.

❺ Show file details of soft link.

Now that you're familiar with filesystem terminology let's explore how Linux makes it possible to treat any kind of resource as a file.

The Virtual File System (VFS)

Linux manages to provide a file-like access to many sorts of resources (in-memory, locally attached, or networked storage) through an abstraction called the Virtual File System (VFS). The basic idea is to introduce a layer of indirection between the clients (syscalls) and the individual filesystems implementing operations for a concrete device or other kind of resource. This means, VFS separates the generic operation (open, read, seek) from the actual implementation details.

VFS is an abstraction layer in the kernel that provides clients a common way to access resources, based on the file paradigm. A file, in Linux, doesn't have any prescribed structure, it is just a stream of bytes. It's up to the client to decide what the bytes mean. As shown in Figure 5-1, VFS abstracts access to different kind of filesystems:

- Local filesystems such as `ext3`, XFS, FAT or NTFS. These filesystems use drivers to access local block devices such as HDDs or SSDs.

- In-memory filesystems such as `tmpfs` that are not backed by long-term storage devices but live in main memory (RAM). We will cover this and the previous category in "Regular Files".
- Pseudo filesystems like `procfs` as discussed in "Pseudo Filesystems". These filesystems are also in-memory in nature, they are used for kernel interfacing and device abstractions.
- Networked filesystems such as NFS, Samba, Novell and others. This kind of filesystem is also using a driver, however, the storage devices where the actual data resides is not locally attached but remote. This means that the driver involves network operations and that's the reason we will cover it in XREF HERE.

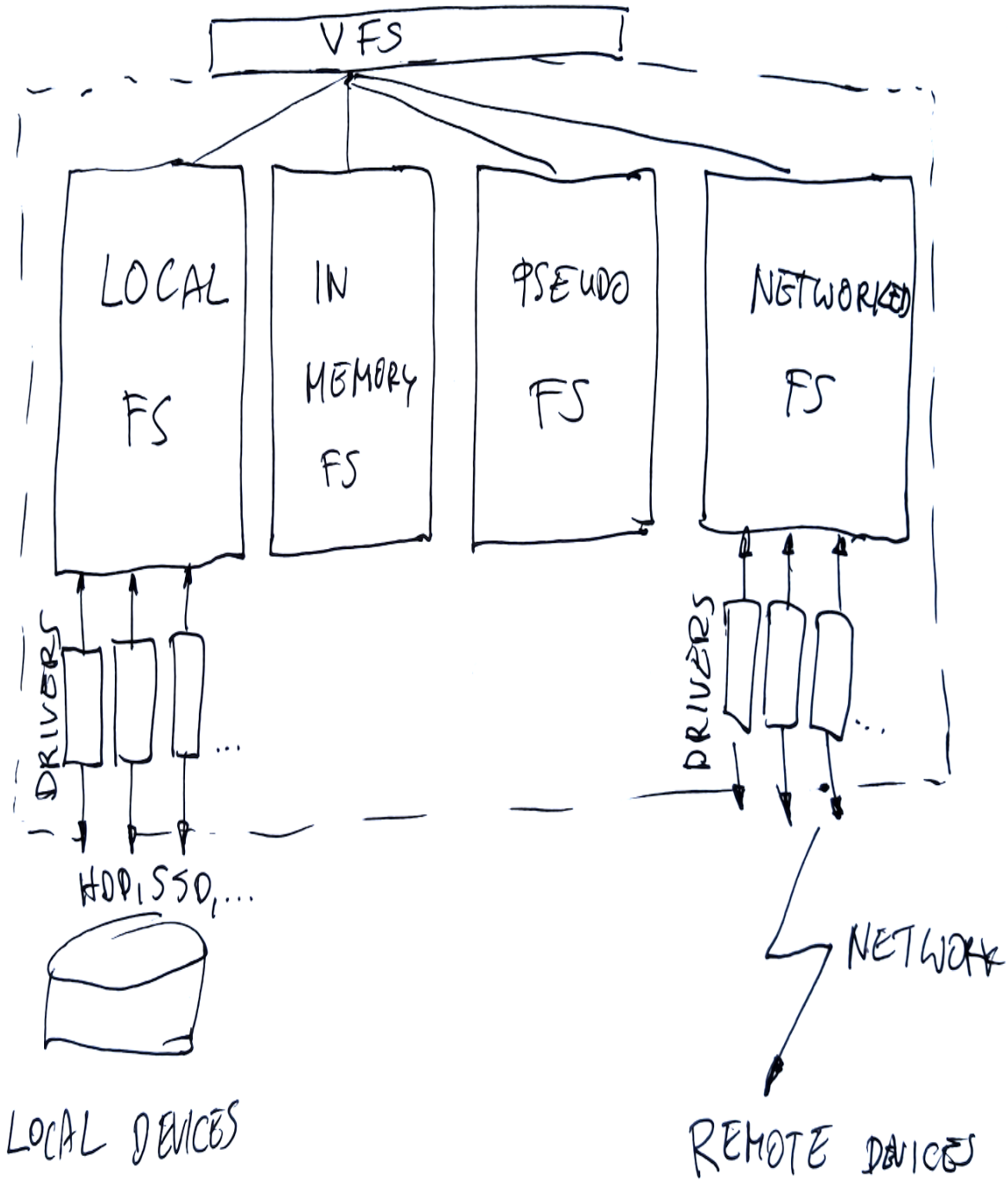


Figure 5-1. Linux VFS overview

Trying to answer the question what makes up the VFS is hard. There are over 100 syscalls related to files, however, in its core the operations can be grouped into a handful of categories as listed in Table 5-2.

Table 5-2. Selects sys calls smaking up the VFS interf

a
c
e

Category	Example syscalls
Inodes	`chmod`, `chown`, `stat`
Files	`open`, `close`, `seek`, `truncate`, `read`, `write`
Directories	`chdir`, `getcwd`, `link`, `unlink`, `rename`, `symlink`
Filesystems	`mount`, `flush`, `chroot`
Others	`mmap`, `poll`, `sync`, `flock`

Many VFS syscalls dispatch to the filesystem-specific implementation. For other syscalls, there are VFS default implementations. Further, the Linux kernel defines relevant VFS data structures (see include/linux/fs.h) such as:

- `inode`, the core filesystem object, capturing type, ownership, permissions, links, pointers to blocks containing the file data, statistics around creation and access and more.
- `file`, representing an open file (incl. path, current position, inode).
- `dentry` (directory entry) that stores its parent and children.
- `super_block`, representing a filesystem incl. mount info.
- Others such as `vfsmount` or `file_system_type`.

With the VFS overview done, let's have a closer look at the details, including volume management, filesystem operations, and common file system layouts.

Logical Volume Manager (LVM)

We previously talked about carving up drives using partitions. While doing this is possible, partitions are hard to use, especially when resizing (changing the amount of storage space) is necessary.

Logical volume manager (LVM) uses an layer of indirection between physical entities (such as drives or partitions) and the file system. This yields a setup that allows for risk-free zero-downtime expanding and automatic storage extenstion through the pooling of resources. The way LVM works is depicted in Figure 5-2:

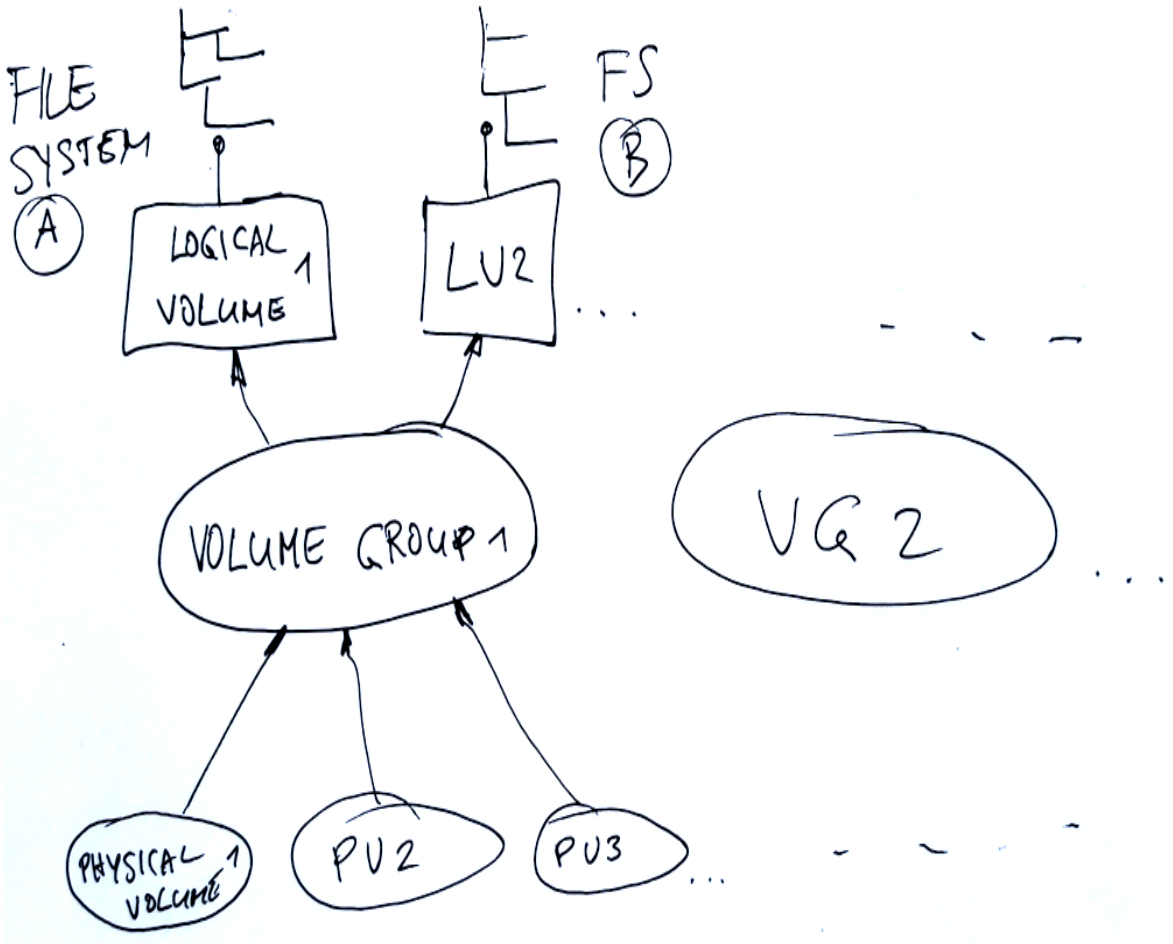


Figure 5-2. Linux LVM overview

- Physical volumes (PV) can be a disk partition, an entire disk drive, and other devices.
- Logical volumes (LV) are block devices created from VGs. These are conceptually comparable to partitions. You have to create a filesystem on an LV before you can use it. You can easily resize LVs while in-use.
- Volume groups (VG) are a go-between between a set of PVs and LVs. Think of it as a pool of PV collectively providing resources.

To manage volumes with LVM, a number of tools are required, however they are consistently named and relatively easy to use:

- PV management: `lvmdiskscan`, `pvdisplay`, `pvcreate`, `pvscan`,
- VG management: `vgs`, `vgdisplay`, `vgcreate`, `vgextend`
- LV management: `lvs`, `lvscan`, `lvcreate`,

Let's see some LVM commands in action, using a concrete setup:

```
$ sudo lvscan ❶
  ACTIVE            '/dev/elementary-vg/root' [<222.10 GiB] inherit
  ACTIVE            '/dev/elementary-vg/swap_1' [976.00 MiB] inherit

$ sudo vgs ❷
  VG             #PV #LV #SN Attr   VSize    VFree
  elementary-vg   1   2   0 wz--n- <223.07g 16.00m

$ sudo pvdisplay ❸
  --- Physical volume ---
  PV Name               /dev/sda2
  VG Name               elementary-vg
  PV Size               <223.07 GiB / not usable 3.00 MiB
  Allocatable           yes
  PE Size               4.00 MiB
  Total PE              57105
  Free PE               4
  Allocated PE          57101
  PV UUID               2OrEfB-77zU-jun3-a0XC-QiJH-erDP-1ujfAM
```

❶ List logical volumes, we have two here (`root` and `swap`) using volume group `elementary-vg`.

❷ Display volume groups, we have one here called `elementary-vg`.

❸ Display physical volumes, we have one here (`/dev/sda2`) that's assigned to the volumne group `elementary-vg`.

Whether you use a partition or an LV, two more steps, which we will cover next, are necessary to use a filesystem.

Filesystem Operations

In the following section we will discuss how to create a filesystem, given a partition or a logical volume (created using LVM). There are two steps involved: creating the filesystem—in other non-Linux operating systems this step is sometimes called formatting—and then mounting it, that is, inserting it into the filesystem tree.

Creating Filesystems

In order to use a filesystem, the first step is to create one. This means that you're setting up the management pieces that make up a filesystem, taking a partition or a volume as the input. Consult Table 5-1 if you're unsure how to gather the necessary information about the input and once you have everything together, use mkfs to create a filesystem.

`mkfs` takes two primary inputs: the type of filesystem you want to create, see one of the options we discuss in "Common Filesystems", and the device you want to create the filesystem on, for example, a logical volume:

```
mkfs -t ext4 \ ❶
    /dev/some_vg/some_lv ❷
```

❶ Create a filesystem of type `ext4`.

❷ Create the filesystem on the logical volume `/dev/some_vg/some_lv`.

As you can see from the previous command, there's not much to it to create a filesystem, so the main work for you is to figure out what filesystem type to use.

Once you have created the filesystem with `mkfs` you can then make it available in the filesystem tree.

Mounting Filesystems

Mounting a filesystem means attaching it to the filesystem tree (which starts at `/`). Use the mount to attach a filesystem. `mount` takes two main inputs: the device you want to attach and the place in the filesystem tree. In addition, other inputs you can provide are:

- Mounts options (via `-o`) such as read-only, and
- Specifing a bind mount (`--bind`) for mounting directories into the filesystem tree, something we will revisit in the context of containers.

You can use `mount` on its own as well. Here's how to list existing mounts:

```
$ mount -t ext4,tmpfs ❶
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size=797596k,mode=755)
/dev/mapper/elementary--vg-root on / type ext4 (rw,relatime,errors=remount-ro) ❷
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
tmpfs on /run/lock type tmpfs (rw,nosuid,nodev,noexec,relatime,size=5120k)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,mode=755)
```

❶ List mounts but only show certain filesystem types (`ext4` and `tmpfs` here).

❷ An example mount: the LVM VG `/dev/mapper/elementary--vg-root` of type `ext4` is mounted at the root.

You must make sure that you mount a filesystem using the type it has been created with. For example, if you're trying to mount an SD card using `mount -t vfat /dev/sdX2 /media` you have to know the SD card is formatted usinf `vfat`. You can let `mount` try all filesystems until one works using the `-a` option.

Further, the mounts are only valid for as long as the system is running, so in order to make it permanent you need to use the fstab (`/etc/fstab`). For example, here is mine (output slightly edited to fit):

```
$ cat /etc/fstab
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point> <type> <options> <dump> <pass>
/dev/mapper/elementary--vg-root / ext4 errors=remount-ro 0 1
# /boot/efi was on /dev/sda1 during installation
UUID=2A11-27C0  /boot/efi vfat umask=0077 0 1
/dev/mapper/elementary--vg-swap_1 none swap sw 0 0
```

Now you know how to manage partitions, volumes, and filesystems. Next up we review common ways to organize filesystems.

Common Filesystem Layouts

Once you have a filesystem in place, an obvious challenge is to come up with a way to organize its content. You may want to organize things like where programs are stored, configuration data, system data, and user data. We will refer to this organization of directories and their content as filesystem layout. Formally, the layout is called the Filesystem Hierarchy Standard (FHS). It defines directories, including their structure and recommended content. The Linux Foundation maintains the FHS and its a good starting point for Linux distributions to follow.

The idea behind FHS is laudable. However, in practice you will find that the filesystem layout very much depends on the Linux distribution you are using. Thus, I strongly recommend you use the `man hier` command to learn about your concrete setup.

To provide you with a high-level idea what you can expect when you see certain top-level directories, I compiled a list of common ones in Table 5-3.

Table 5-3. Common top-level directories

Directory	Semantics
`bin`, `sbin`	system programs and commands
`boot`	kernel images etc.
`dev`	devices (terminals, drives, etc.)

`etc`	system configuration files
`home`	user home directories
`lib`	shared system libraries
`mnt`, `media`	mount points for removable media (USB sticks etc)
`opt`	distro-specific, can host package manager files
`proc`, `sys`	kernel interfaces, see "Pseudo Filesystems"
`tmp`	for temporary files
`var`	logs, backups, network caches, etc.

With that, let's move on to some special kind of filesystems.

Pseudo Filesystems

Filesystems are a great way to structure and access information. By now you have likely already internalized the Linux motto that "everything is a file". We looked at how Linux provides a uniform interface via VFS in "The Virtual File System (VFS)". Now, let's take a closer look at how an interface is provided in cases where the VFS implementor is not a block device (such as an SD card or a SSD drive).

Meet pseudo filesystems: they only pretend to be filesystems so that we can interact with them in the usual manner (`ls`, `cd`, `cat`) but really they are wrapping some kernel interface. The interface can be a range of things, including:

- information about a process,
- an interaction with devices such as keyboards, or
- utilities such as special devices you can use as data sources or sinks.

Let us have a closer look at the three major pseudo filesystems Linux has, starting with the oldest.

Procfs

Linux inherited the /proc filesystem (`procfs`) from Unix. The original intention was to publish process-related information from the kernel, to make it consumable for system commands such as `ps` or `free`. It has very few rules around structure, allows read-write access, and over time many things found their way into it. In general, you find two types of information there:

- Per-process information in `/proc/PID/`. This is process-relevant information that the kernel exposes via directories with the PID as the directory name. Details concerning the info available there are listed in Table 5-4.
- Other information such as mounts, networking-related infos, TTY drivers, memory info, system version, uptime, etc.

You can glean per-process information as listed in Table 5-4 simply by using commands like `cat`. Note that most are read-only, the write semantics depend on the underlying resource.

Table 5-4. Per-process information in procfs

(mostnotable)

Entry	Type	Information
`attr`	directory	security attributes
`cgroup`	file	control groups
`cmdline`	file	command line
`cwd`	link	current working directory
`environ`	file	environment variables
`exe`	link	executable of the process
`fd`	directory	file descriptors
`io`	file	storage I/O (bytes/char read and written)
`limits`	file	resource limits
`mem`	file	memory used
`mounts`	file	mounts used
`net`	directory	network stats
`stat`	file	process status
`syscall`	file	syscall usage
`task`	directory	per task (thread) info
`timers`	file	timers info

Let's see this in action, for example, let's inspect the process status. We're using `status` here rather than `stat` which doesn't come with humand readable labels:

```
$ cat /proc/self/status | head -10 ❶
Name:   cat
Umask:  0002
State:  R (running) ❷
Tgid:   12011
Ngid:   0
Pid:    12011 ❸
PPid:   3421
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
```

❶ Get process status about the currently running command, only show the first ten lines.

❷ The current state (running, on-CPU).

❸ The PID of the current process.

Here is one more example of using `procfs` to glean information, this time from the networking space:

```
$ cat /proc/self/net/arp
IP address       HW type     Flags       HW address            Mask     Device
192.168.178.1    0x1         0x2         3c:a6:2f:8e:66:b3     *        wlp1s0
192.168.178.37   0x1         0x2         dc:54:d7:ef:90:9e     *        wlp1s0
```

As shown in the previous command we can glean ARP info about the current process from this special `/proc/self/net/arp`.

The `procfs` is very useful if you're low-level debugging or developing system tooling. It is relatively messy, so you will need the kernel docs or even better the kernel source code at hand to understand what each file respresent and how to interpret the information in it.

Let's move on to a more recent, more orderly way the kernel exposes information.

Sysfs

Where `procfs` is pretty Wild West, the /sys filesystem (`sysfs`) is a Linux-specifc, structured way for the kernel to expose select information (such as about devices) using a standardized layout.

Here are the directories in `sysfs`:

- The `block/` directory contains symbolic links to discovered block devices.

- In the `bus/` directory you find one sub-directory for eachphysical bus type supported in the kernel.
- The `class/` directory contains device classes.
- The `dev/` directory contains two sub-directories: `block/` for block devices and `char/` for character devices on the system, structured with `major-ID:minor-ID`.
- In the `devices/` directory the kernel provides a representation of the device tree.
- Via the `firmware/` directories you can manage firmware-specific attributes.
- The `fs/` directory contains subdirectories for some filesystems
- In the `module/` directories you find sub-directories for each module loaded in the kernel.

There are more subdirectories in `sysfs` but some are newish and/or poorly documented. You will find certain information duplicated in sysfs that is also available in `procfs`, but other information (such as memory info) is only available in `procfs`.

Let's see `sysfs` in action (output edited to fit):

```
$ ls -al /sys/block/sda/ | head -7 ❶
total 0
drwxr-xr-x 11 root root    0 Sep  7 11:49 .
drwxr-xr-x  3 root root    0 Sep  7 11:49 ..
-r--r--r--  1 root root 4096 Sep  8 16:22 alignment_offset
lrwxrwxrwx  1 root root    0 Sep  7 11:51 bdi ->  ../../../virtual/bdi/8:0 ❷
-r--r--r--  1 root root 4096 Sep  8 16:22 capability ❸
-r--r--r--  1 root root 4096 Sep  7 11:49 dev ❹
```

❶ List information about block device `sda` but only show the first seven lines.

❷ The `backing_dev_info` link using `MAJOR:MINOR` format.

❸ Captures device capabilities such as if it is removable.

❹ Contains the device major and minor number (`8:0`), see also the block device drivers reference for what the numbers mean.

Next up in our little pseudo filesystem review are devices.

Devfs

The /dev filesystem (`devfs`) hosts device special files, representing devices ranging from physical devices to things like random number generator or a write-only data sink.

The devices available and managed via `devfs` are:

- Block devices that handle data in blocks, for example, storage devices (drives).
- Character devices that handle things character by character, such as a terminal, a keyboard, or a mouse.
- Special devices that generate data or allow you to manipulate it, including `/dev/random` or `/dev/null`.

Let us now see `devfs` in action. For example, assume you want to get a random string. You could do something like the following:

```
tr -dc A-Za-z0-9 < /dev/urandom | head -c 42
```

The previous command generates a 42 character long random sequence containing upper and lower case as well as numerical characters. And while `/dev/urandom` looks like a file and can be used like one it indeed is a special file that, using a number of sources, generates (more or less) random output.

What do you think about the following command:

```
echo "something" > /dev/tty
```

That's right! The string "something" appeard on your display and that is by design. Because `/dev/tty` stands for the terminal and with that command we sent something (quite literally) to it.

With a good understanding of filesystems and their features, let us now turn our attention to filesystems that you want to use to manage regular files such as documents and data files.

Regular Files

In this section we focus on regular files and filesystems for such file types. Most of the day-to-day files we're dealing with when working fall in this category: office documents, YAML and JSON configuration files, images (PNG, JPEG, etc.), source code, plain text files.

Linux comes with a wealth of options. We will focus on local filesystems, both those native for Linux, as well as those in other operating systems (such as Windows/DOS) that Linux allows you to use. First, let's have a look at some common filesystems.

Common Filesystems

The term common filesystem doesn't have a formal definition. It's simply an umbrella term for filesystems that are either the defaults used in Linux distributions or widely used in storage devices such as removable devices (USB sticks, SD cards) or read-only like a CD or DVD.

In the Table 5-5 I provide a quick overview and comparison of some common filesystems that enjoy in-kernel support. Later in this section we will review some popular ones in greater

detail.

Table 5-5. Common files systems for regular files

Filesystem	Linux support since	File Size	Volume Size	Number of Files	Filename length
ext2	1993	2 TB	32 TB	10^{18}	255 characters
ext3	2001	2 TB	32 TB	variable	255 characters
ext4	2008	16 TB	1 EB	4 billion	255 characters
btrfs	2009	16 EB	16 EB	2^{18}	255 characters
XFS	2001	8 EB	8 EB	2^{64}	255 characters
ZFS	2006	16 EB	2^{128} Bytes	10^{14} files per directory	255 characters
NTFS	1997	16 TB	256 TB	2^{32}	255 characters
vfat	1995	2 GB	N/A	2^{16} per directory	255 characters

NOTE

The information provided in Table 5-5 is meant to give you a rough idea about the filesystems. Sometimes its hard to pinpoint the exact time a filesystem would be officially considered part of Linux, sometimes the numbers only make sense with the relevant context applied. For example, there are differences between theoretical limits and implementation.

Now let's take a closer look at some widely used filesystems for regular files.

Ext4

ext4 is a widely used filesystem, used by default in many distributions nowadays. It is a backwards compatible evolution of `ext3`. Like `ext3` it offers journaling, that is, changes are recorded in a log so that in the worst-case scenario (think: power outage) the recovery is fast. It's a great general purpose choice, see also the manual for usage.

XFS

XFS is a journaling filesystem that was originally designed by Silicon Graphics (SGI) for their workstations in the early 1990s. Offering support for large files and high-speed I/O, it is nowadays used, for example, in the Red Hat distributions family.

ZFS

ZFS, originally developed by Sun Microsystems in 2001, combines filesystem and volume manager functionality. While there is nowadays the OpenZFS project, offering a path forward in an open source context, there are some concerns about ZFS's integration with Linux.

FAT

This is really a family of FAT filesystems for Linux with `vfat` nowadays being mostly used. The main use case is interoperability with Windows systems as well as removable media that uses FAT. Many of the native considerations around volumes do not apply.

Drives are not the only place one can store data, so let's have a look at in-memory options.

In-memory Filesystems

There are a number of in-memory filesystems available, some general purpose, others have very specific use cases. In the following, we list some widely used in-memory filesystem (in alphabetical order):

- debugfs is a special purpose filesystem used for debugging usually mounted with `mount -t debugfs none /sys/kernel/debug"`.
- loopfs allows mapping a filesystem to blocks rather than devices (see also a mail thread on the background).
- `pipefs` is a special (pseudo) filesystem mounted on `pipe:`, enabling pipes.
- tmpfs is a general purpose filesystem that keeps file data in kernel caches. It's fast but non-persistent (power off means data is lost).
- `sockfs` is another special (pseudo) filesystem that makes network sockets look like files, sitting between the syscalls and the sockets.
- swapfs is used to realize swapping (not mountable).

Let us move on to a special category of filesystems, specifically relevant in the context of XREF HERE.

Copy-on-write Filesystems

Copy-on-write (CoW) is a nifty concept to increase I/O speed and at the same time use less space. In Figure 5-3 we depict how it works:

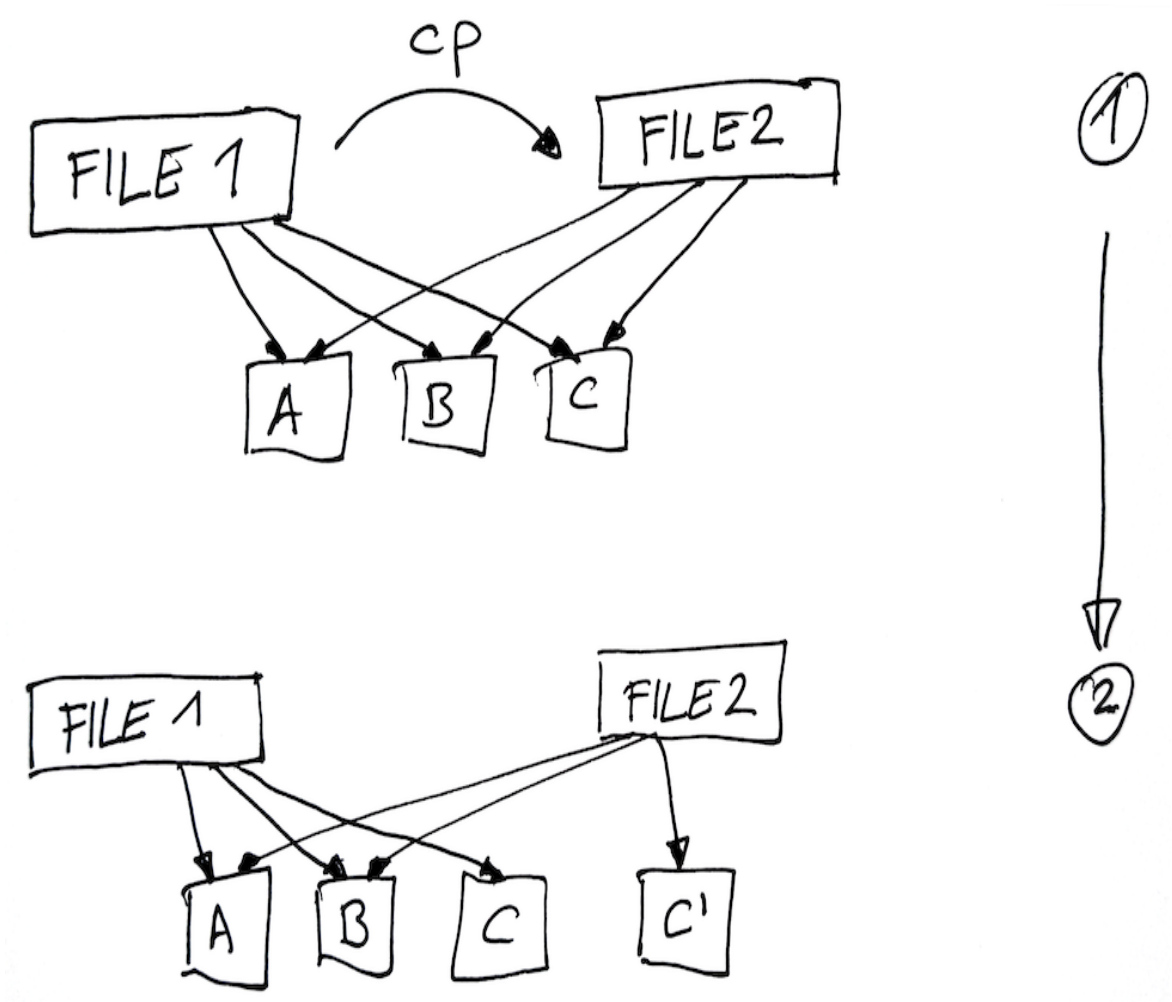


Figure 5-3. The CoW principle in action

1. The original file `FILE 1`, consisting of block `A`, `B`, and `C` is copied to a file called `FILE 2`. Rather than copying the actual blocks, only the metadata (pointers to the blocks) is copied. This is fast and doesn't use up much space since only metadata is created.
2. When `FILE 2` is modified (let's say something in block `C` is changed) only then the block `C` is copied: a new block called `C'` is created and while `FILE 2` still points to (uses) the unmodified blocks `A` and `B` it now used a new block (`C'`) capturing new data.

Before we get to implementations we need to understand a second concept relevant in this context: union mounts. This is the idea that you can combine (mount) multiple directories into one location so that to the user of the resulting directory it appears that said directory contains the combined content (or: union) of all the participating directories. With union mounts you often comes across the terms upper or lower filesystem, hinting at the layering order of the mounts, read details in Unifying filesystems with union mounts.

With union mounts, the devil is in the details. One has to come up with rules around what happens when a file exists in multiple places or what writing to or removing files means.

Let's have a quick look at implementations of CoW in the context of Linux filesystems. We will have a closer look at some of these in the context of XREF HERE, when we discuss their use as a building block for container images.

Unionfs

Unionfs, originally developed at Stony Brook University, implements a union mount for CoW filesystems. Unionfs allows you to transparently overlay files and directories from different filesystems using priorities at mount time. It was widely popular and used in the context of CD-ROMs and DVDs.

OverlayFS

OverlayFS, is a union mount filesystem implementation for Linux introduced in 2009 and added to the kernel in 2014. With OverlayFS, once a file is opened, all operations are directly handled by the underlying (lower or upper), filesystems.

AUFS

Another attempt to implement an in-kernel union mount is AUFS (advanced multi-layered unification filesystem, originally AnotherUnionFS), however it has not been merged into the kernel, yet. It is used to default in Docker (nowadays Docker defaults to OverlayFS with storage driver `overlay2`).

btrfs

btrfs, short for b-tree filesystem (and pronounced butterFS or betterFS) is a CoW initially designed by Oracle Corporation. Nowadays a number of companies contribute to the btrfs development, including Facebook, Intel, SUSE, and Red Hat.

It comes with a number of features such as snapshots (for software-based RAID), and automatic detection of silent data corruptions. This makes `btrfs` very suitable for professional environments, for example on a server.

Conclusion

In this chapter we discussed files and filesystems in Linux. Filesystems are a great and flexbile way to organize access to information in a hierachical manner. Linux has many technologies and projects around filesystems. Some are open source based, but there is also a range of commerical offerings.

We discussed the basic building blocks, from drives to partitions and volumes. Then we looked at how Linux realizes the "everything is a file" abstraction using VFS. Further, we reviewed pseudo filesystems such as `/proc` that the kernel uses to expose information about processes

or devices. We then we moved on to regular files and filesystems, where we compared common local filesystem options, as well as in-memory and CoW filesystem basics.

Some further reading resources for you to get deeper into the topics covered here are:

1. Basics:
    - UNIX File Systems: How UNIX Organizes and Accesses Files on Disk
    - KHB: A Filesystems reading list
2. VFS:
    - Overview of the Linux Virtual File System
    - Introduction to the Linux Virtual Filesystem (VFS)
    - LVM via ArchWiki
    - LVM2
    - How To Use GUI LVM Tools
    - Linux Filesystem Hierarchy guide
    - Persistent BPF objects
3. Regular files:
    - Filesystem Efficiancy - Comparision of EXT4, XFS, BTRFS, and ZFS
    - Linux Filesystem Performance Tests
    - Comparison of File Systems for an SSD
    - Kernel Korner - Unionfs: Bringing Filesystems Together
    - Getting started with btrfs for Linux

Equipped with all the knowledge around filesystems we're now ready to bring things together and focus on how to manage and launch applications.

Michael Hausenblas

Michael is a solution engineering lead in the Amazon Web Services (AWS) open source observability service team. His background is in data engineering and container orchestration, from Mesos to Kubernetes. Michael is experienced in advocacy and standardization at W3C and IETF and writes code these days mainly in Go. Before Amazon, he worked at Red Hat, Mesosphere (now: D2iQ), MapR (now part of HPE), and a decade in applied research.