CHAPTER 1

Introduction to Linux

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 bird's-eye view of operating systems and Linux in this chapter.

We'll first discuss what *modern* means in the context of the book. Then we'll review a high-level Linux backstory, looking at important events and phases over the past 30 years. Further, in this chapter you'll learn what the role of an operating system is in general and how Linux fills this role. We also take a quick look at what Linux distributions are and what resource visibility means.

If you're new to operating systems and Linux, you'll want to read the entire chapter. If you're already experienced with Linux, you might want to jump to "A Ten-Thousand-Foot View of Linux" on page 8, which provides a visual overview as well as mapping to the book's 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 nontrivial matter.

What Are Modern Environments?

The book title specifies *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's take a closer look at some of these modern environments and the prominent role Linux plays in them:

Mobile devices
: When I say "mobile phone" to our kids, they say, "In contrast to what?" In all fairness and seriousness, these days many phones—depending on who you ask, up to 80% or more—as well as tablets run Android, which is a Linux variant. These environments have aggressive requirements around power consumption and robustness, as we depend on them on a daily basis. If you're interested in developing Android apps, consider visiting the Android developer site for more information.

Cloud computing
: With the cloud, we see at scale a similar pattern as in the mobile and micro space. There are new, powerful, secure, and energy-saving CPU architectures 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.

Internet of (Smart) Things
: I'm sure you've seen a lot of Internet of Things (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 than mobile devices. 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 of these environments is real-time capabilities. If you're interested in getting started with Linux in the IoT context, consider the AWS IoT EduKit.

Diversity of processor architectures
: For the past 30 years or so, Intel has been the leading CPU manufacturer, dominating the microcomputer 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 promising, resulting in x86 clones that also used Intel chips, at least initially.

: While Intel is still widely used in desktop and laptop systems, with the rise of mobile devices we've seen the increasing uptake of the ARM architecture and recently RISC-V. At the same time, multi-arch programming languages and tooling, such as Go or Rust, are becoming 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 Linux in one form or another.

Now that we know about 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 30th birthday in 2021. With billions of users and thousands of developers, the Linux project is, without doubt, a worldwide (open source) success story. But how did it all this start, and how did we get here?

1990s
: We can consider Linus Torvalds's email on August 25, 1991, to the `comp.os.minix` newsgroup as the birth of the Linux project, at least in terms of the public 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 that time, the original goal of being able to run most Unix/GNU software was already well reached. Also, the first commercial offering appeared in the 1990s: Red Hat Linux.

2000 to 2010
: As a "teenager," Linux was not only maturing in terms of features and supported hardware but was 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, and so on. It was also the peak of the distro wars, resulting in businesses changing their directions.

2010s to now
: Linux established itself as the workhorse 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 being over (nowadays, most commercial systems are either Red Hat or Debian based), and in a sense, the rise of containers (from 2014/15 on) is responsible for this development.

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 cannot 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 network stack—the list goes on.

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 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" on page 22.

You might be wondering what the `(2)` means in `getuid(2)`. It's a terminology that the `man` utility (think built-in help pages) uses to indicate the section of the command assigned in `man`, akin to a postal or country code. This is one example where the Unix legacy is apparent; 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 using 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 "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 and related components, including package management, file system layout, init system, and a shell, preselected for you.

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

If you are inclined to build your own distribution, maybe because you are a tinkerer or because you have to due to certain business restrictions, I recommend you take 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, Red Hat Enterprise Linux [RHEL], CentOS, etc., as discussed in Chapter 6) and modern distros (such as Bottlerocket and Flatcar; see Chapter 9), take a look at DistroWatch.

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

Why are we talking about resource visibility here in the first place? The main reason is to raise awareness about this topic and to get you in the right state of mind for one of the important themes in the context of modern Linux: containers. Don't worry if you don't get all of the details now; we will come back to this topic throughout the book and specifically in Chapter 6, in which we discuss containers and their building blocks in greater detail.

You might have heard the saying that in Unix, and by extension Linux, everything is a file. In the context of this book, we consider resources to be anything that 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, solid-state drives (SSDs), processes, networking-related stuff like devices or routing tables, and credentials representing users.

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 specific hardware information 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 the preceding 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 the user `troy` creates a file under `/tmp/myfile` with permission to do so ("Permissions" on page 80), would another user, `worf`, 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 available to run, such as CPU and memory. Linux identifies a process using its *process ID*, or `PID` for short ("Process Management" on page 17):

```
$ 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 show only the first six lines.

What Is $$?

You might have noticed the `$$` and wondered what this means. This is a special variable that is referring to the current process (see "Variables" on page 37 for details). Note that in the context of a shell, `$$` is the process ID of the shell (such as bash) in which you typed the command.

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 (see "Containers" on page 131). The answer is yes, there can be multiple processes with the same `PID`, in different contexts called *namespaces* (see "Linux Namespaces" on page 133). This happens, for example, in a containerized setup, such as when you're running your app in Docker or Kubernetes.

Every single process might think that it is special, having `PID 1`, which in a more traditional setup is reserved for the root of the user space process tree (see "The Linux Startup Process" on page 117 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 raises 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 processes 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 (see "Linux Namespaces" on page 133).

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 cannot starve other processes. For example, I give your app 1 GB of RAM to use. If it uses more, it gets out-of-memory killed. This provides a certain level of protection. In Linux we use a kernel feature called `cgroups` to provide this kind of isolation, and in "Linux cgroups" on page 135 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 "Virtual Machines" on page 217) can be used to provide you with full isolation.

A Ten-Thousand-Foot View of Linux

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

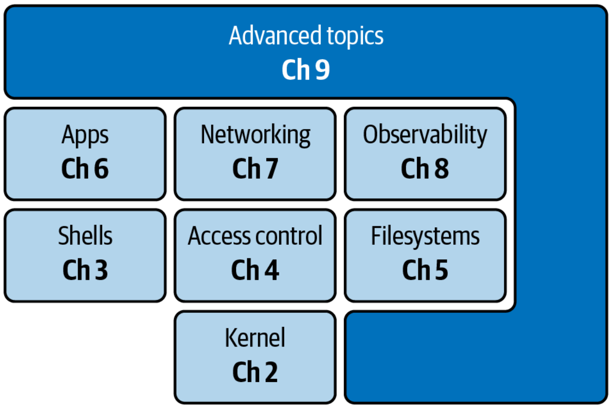

Figure 1-1. Mapping the Linux operating system to book chapters

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

As an aside: I've collected some interesting topics, from virtual machines to modern distros, in Chapter 9. I call these topics "advanced" mainly because I consider them optional. That is, you could get away without learning them. But if you really, really, really want to benefit from the full power that modern Linux can provide you, I strongly recommend that you read Chapter 9. I suppose it goes without saying that, by design, the rest of the book—that is Chapter 2 to Chapter 8—are essential chapters you should most definitely study and apply the content as you go.

Portable Operating System Interface

We will come across the term *POSIX*, short for *Portable Operating System Interface*, every now and then in this book. Formally, POSIX is an 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 in everyday usage.

Linux was built to be POSIX-compliant as well as to be compliant with the UNIX System V Interface Definition (SVID), which gave it the flavor of old-time AT&T UNIX systems, as opposed to Berkeley Software Distribution (BSD)-style systems.

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

Conclusion

When we call something "modern" in the context of this book, we mean using Linux in modern environments, including phones, data centers (of public cloud providers), and embedded systems such as a Raspberry Pi.

In this chapter, I shared a high-level version of the Linux backstory. We 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 how Linux goes about this task, specifically regarding visibility of resources.

The following resources will help you continue getting up to speed as well as dive deeper into concepts discussed in this chapter:

O'Reilly titles

- *Linux Cookbook* by Carla Schroder
- *Understanding the Linux Kernel* by Daniel P. Bovet and Marco Cesati
- *Efficient Linux at the Command Line* by Daniel J. Barrett
- *Linux System Programming* by Robert Love

Other resources

- Advanced Programming in the UNIX Environment is a complete course that offers introductory material and hands-on exercises.
- "The Birth of UNIX" with Brian Kernighan is a great resource for learning about Linux's 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

In "Why an Operating System at All?" on page 3, we learned that the main function of an operating system is to abstract over different hardware and provide us with an 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 takeaway of this chapter is that while the kernel provides all the core functionality, on its own it is not the operating system but only a very central part of it.

First, we take a bird's-eye view, looking at how the kernel fits in and interacts with the underlying hardware. Then, 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 and discuss the API the kernel provides to programs you can run. Finally, we look at how to customize and extend the Linux kernel.

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, however, you want to dive into that, I've put together some pointers at the end of the chapter.

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

Linux Architecture

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

Hardware
: From CPUs and main memory to disk drives, network interfaces, and peripheral devices such as keyboards and monitors.

The kernel
: The focus of the rest of this chapter. Note that there are a number of components that sit between the kernel and user land, such as the init system and system services (networking, etc.), but that are, strictly speaking, not part of the kernel.

User land
: Where the majority of apps are running, including operating system components such as shells (discussed in Chapter 3), utilities like `ps` or `ssh`, and graphical user interfaces such as X Window System–based desktops.

We focus in this book on the upper two layers of Figure 2-1, that is, the kernel and user land. We only touch on the hardware layer in this and a 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 is the interface called *system calls* (*syscalls* for short). We will explore this in detail in "syscalls" on page 22.

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. The CPU interface (see "CPU Architectures" on page 14)
2. The interface with the main memory, covered in "Memory Management" on page 19
3. Network interfaces and drivers (wired and wireless; see "Networking" on page 20)
4. Filesystem and block devices driver interfaces (see "Filesystems" on page 21)
5. Character devices, hardware interrupts, and device drivers, for input devices like keyboards, terminals, and other I/O (see "Device Drivers" on page 21)

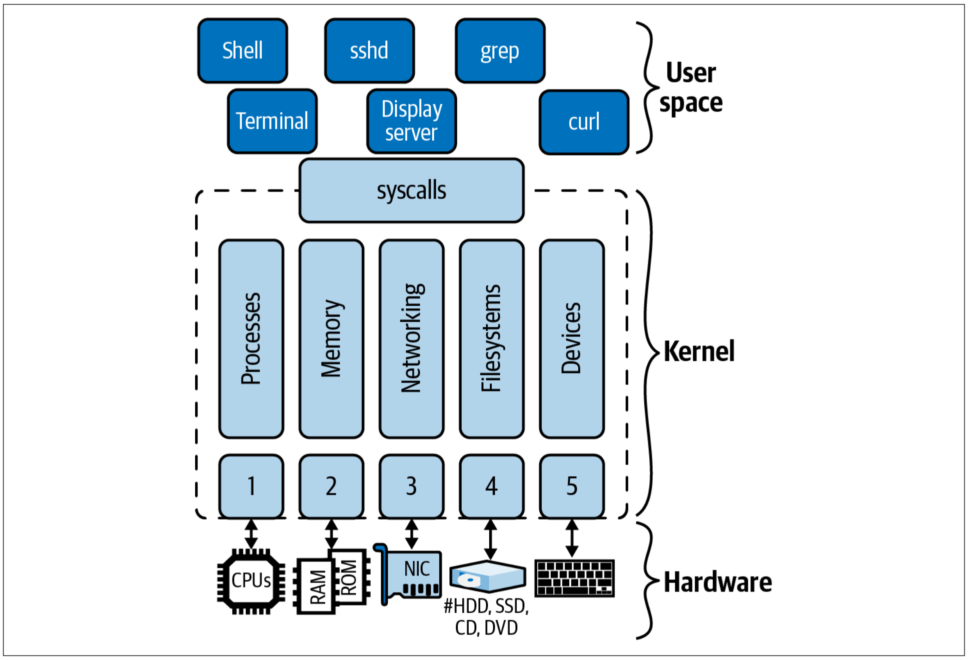

Figure 2-1. A high-level view of the Linux architecture

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 versus kernel mode. This effectively refers to how privileged the access to hardware is and how restricted the abstractions available are.

In general, *kernel mode* means fast execution with limited abstraction, whereas *user mode* means comparatively slower but safer and more 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 page 22), 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: 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 "The Linux Startup Process" on page 117.

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* 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 called `dmidecode` that interacts with the BIOS. If this doesn't yield results, you could try the following (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`, and 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 by using `cat /proc/cpuinfo`, or, if you're only interested in the architecture, by simply calling `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 Advanced Micro Devices (AMD). Within the kernel, `x64` refers to the Intel 64-bit processors, and `x86` stands for Intel 32-bit. Further, `amd64` refers to AMD 64-bit processors.

Today, you'll mostly find the x86 CPU family in desktops and laptops, but it's also widely used in servers. Specifically, `x86` forms the basis of the public cloud. It is a powerful and widely available architecture but isn't very energy efficient. 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

More than 30 years old, ARM is a family of Reduced Instruction Set Computing (RISC) architectures. RISC usually consists of many 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 they are fast, cheap, and produce less heat than x86 chips, you shouldn't be surprised to increasingly find ARM-based CPUs—such as AWS Graviton—in the data center. While simpler than x86, ARM is not immune to vulnerabilities, such as Spectre. For further details, see the ARM-specific kernel documentation.

RISC-V Architecture

An up-and-coming player, RISC-V (pronounced *risk five*) is an open RISC standard that was originally developed by the University of California, Berkeley. As of 2021, a number of implementations exist, ranging from Alibaba Group and Nvidia to 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 kernel documentation.

Kernel Components

Now that you know the basics of CPU architectures, 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" on page 12, 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
- Memory management, such as allocating memory for a process or map a file into memory
- Networking, like managing network interfaces or providing the network stack
- Filesystems providing file management and supporting the creation and deletion of files
- Management of character devices and device drivers

These functional components often come with interdependencies, and it's 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 program (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 *multithreading*, which means that a process has a number of parallel executions going on, potentially running on different CPUs.

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

Sessions
: Contain one or more process groups and represent a high-level user-facing unit with optional `tty` attached. The kernel identifies a session via a number called *session ID* (SID).

Process groups
: Contain one or more processes, with at most one process group in a session as the foreground process group. The kernel identifies a process group via a number called *process group ID* (PGID).

Processes
: Abstractions that group 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 called *process ID* (PID).

Threads
: Implemented by the kernel 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) and *thread group IDs* (TGID), with the semantics that a shared TGID value means a multithreaded 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*—that forms the basis of implementing processes and threads alike. This data structure captures scheduling-related information, identifiers (such as PID and TGID), and signal handlers, as well as other information, such as that related to performance and security. In a nutshell, all of the aforementioned 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 learn how to manage them in Chapter 6, and they'll appear again in the context of containers in Chapter 9.

Let's see some of these concepts 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 at any given time a process is in a certain state, as shown in Figure 2-2.

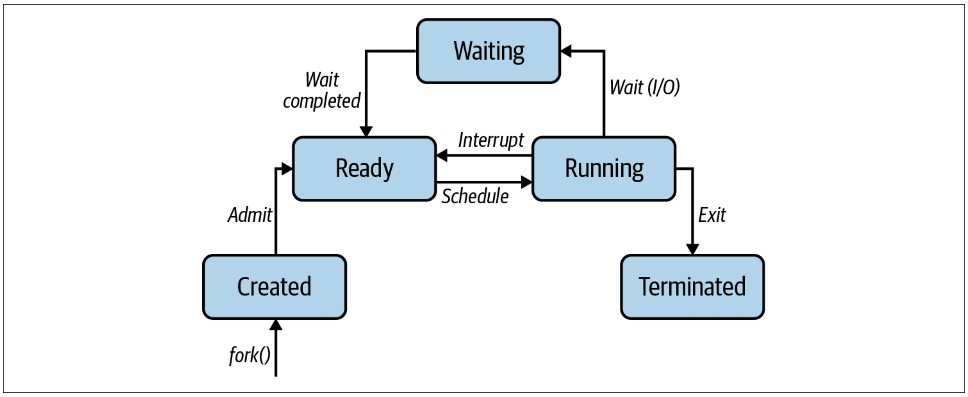

Figure 2-2. Linux process states

Strictly speaking, the process states are a little more complicated; for example, Linux distinguishes between interruptible and uninterruptible sleep, and there is also the zombie state (in which it has lost its parent process). If you're interested in the details, check out the article "Process States in Linux".

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).

Having taken a quick look at process management, let's examine a 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. This is how it works: both physical memory and virtual memory are divided into fixed-length chunks we call *pages*.

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

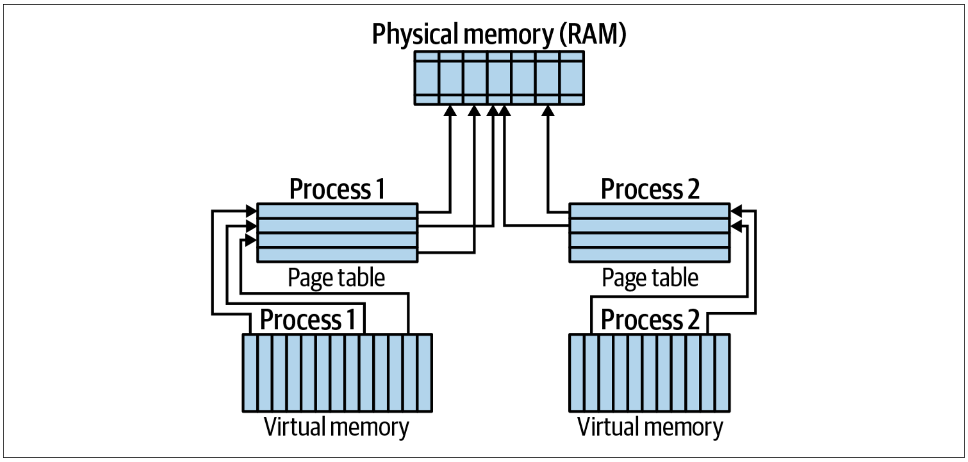

Figure 2-3. Virtual memory management overview

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 its page actually exists in RAM while using the existing space optimally.

Every time the CPU accesses a process's 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—which can be multilevel 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 (with virtual being the theoretical addressable number of memory addresses) per process, with an approximate 64 TB of physical memory (with physical being the amount of RAM you have in your machine) in total.

OK, that was a lot of theoretical information. 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 4 GB there.

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

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

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

Networking

One important function of the kernel is to provide networking functionality. Whether you want to browse the web or copy data to a remote system, you depend on the network.

The Linux network stack follows a layered architecture:

Sockets
: For abstracting communication

Transmission Control Protocol (TCP) and User Datagram Protocol (UDP)
: For connection-oriented communication and connectionless communication, respectively

Internet Protocol (IP)
: For addressing 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 (Chapter 7) 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 filesystems to organize files and directories on storage devices such as hard disk drives (HDDs) and solid-state drives (SSDs) 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 on filesystems and file operations in Chapter 5.

Device Drivers

A *driver* is a bit of code that runs in the kernel. Its job is to manage a device, which can be actual hardware—like a keyboard, a mouse, or hard disk drives—or it can be a pseudo-device such as a pseudo-terminal under */dev/pts/* (which is not a physical device but can be treated like one).

Another interesting class of hardware are *graphics processing units* (GPUs), which traditionally were used to accelerate graphics output and ease the load on the CPU. In recent 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 (see "Modules" on page 26) so that it can be dynamically loaded when needed.

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, following are a few hints for interacting with it, just enough so that you know where to find what.

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

```
$ 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)
```

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 whether 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: around three hundred or more, depending on the CPU family. 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. 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:

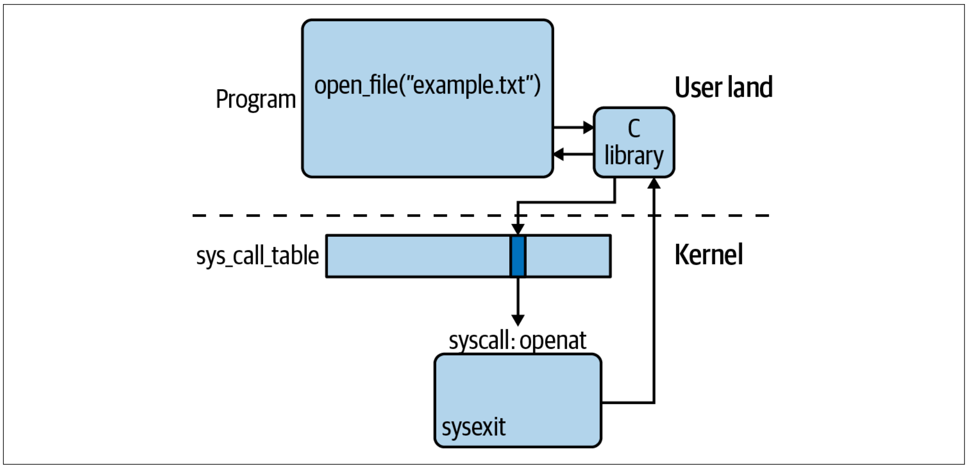

Figure 2-4. syscall execution steps in Linux

1. Defined in *syscall.h* and architecture-dependent files, the kernel uses a so-called *syscall table*, effectively an array of function pointers in memory (stored in a variable called `sys_call_table`) to keep track of syscalls and their corresponding handlers.
2. 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 respective syscall number index in the `sys_call_table`.
3. After the syscall is completed with `sysexit`, the wrapper library restores the hardware context, and the program execution resumes in user land.

Notable 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\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 the 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. You'll 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 first 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 (first argument, `3`) 832 bytes (last argument) into a buffer (second 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`.

Interestingly, 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 help you 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 syscalls, including their parameters and return values, via section 2 of the man pages.

Table 2-1. Example syscalls

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`, `setsockopt`, `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`

There is a nice interactive syscall table 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 optional. You won't need it for your day-to-day work, in general.

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* (O'Reilly) by Greg Kroah-Hartman, one of the main Linux maintainers and project lead. He covers the entire range of tasks, from 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 the time of writing, I'm using a 5.11 kernel on an `x86_64` machine (see also "x86 Architecture" on page 15).

Now that we know the kernel version, we can 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 build it. For this extension 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. Nowadays, Linux detects most of the hardware automatically, and with it Linux loads its modules automatically. But there are cases where you want to manually load a module. Consider the following case: the kernel detects a video card and loads a generic module. However, the video card manufacturer offers a better third-party module (not available in the Linux kernel) that you may choose to use instead.

To list available modules, run the following command (output has been edited down, as there are over one thousand lines on my system):

```
$ 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 take a look (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 preceding information is available via */proc/modules*. This is thanks to the kernel exposing this information via a pseudo-filesystem interface; more on this topic is presented in Chapter 6.

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 known as *Berkeley Packet Filter* (BPF), nowadays the kernel project and technology is commonly known as *eBPF* (a term that 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 an in-kernel virtual machine using a custom 64-bit RISC instruction set.

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

In Figure 2-5 you see a high-level overview taken from Brendan Gregg's book *BPF Performance Tools: Linux System and Application Observability* (Addison Wesley).

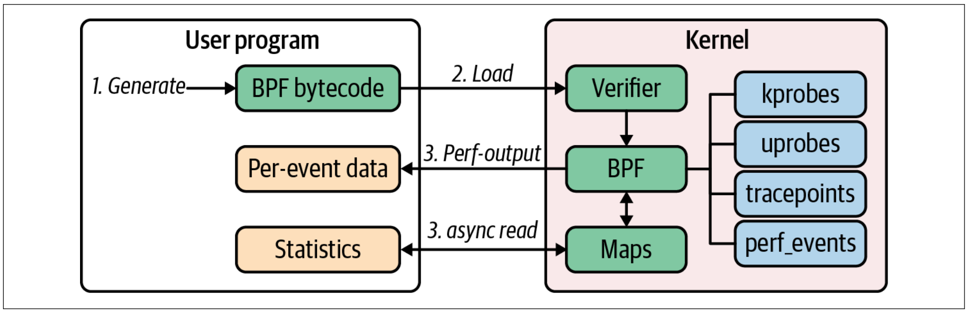

Figure 2-5. eBPF overview in the Linux kernel

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

As a CNI plug-in to enable pod networking in Kubernetes
: For example, in Cilium and Project Calico. Also, for service scalability.

For observability
: For Linux kernel tracing, such as with iovisor/bpftrace, as well as in a clustered setup with Hubble (see Chapter 8).

As a security control
: For example, to perform container runtime scanning as you can use with projects such as CNCF Falco.

For network load balancing
: Such as in 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!

If you want to stay on top of things, have a look at *ebpf.io*.

Conclusion

The Linux kernel is the core of the Linux operating system, and no matter what distribution or environment you are using Linux in—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 of the kernel, and its interfaces. Most importantly, the kernel abstracts away the differences of the hardware—CPU architectures 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 the inner workings of the kernel, including modules and eBPF. If you want to extend the kernel functionality or implement performant tasks in the kernel (controlled from the user space), then eBPF is definitely worth taking a closer look at.

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

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 into hands-on topics.
- kernelstats shows some interesting distributions over time.
- The Linux Kernel Map is a visual representation of kernel components and dependencies.

Memory management

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

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*

syscalls

- "Linux Interrupts: The Basic Concepts"
- The Linux Kernel: System Calls
- Linux System Call Table
- *syscalls.h* source code
- syscall lookup for x86 and x86_64

eBPF

- "Introduction to eBPF" by Matt Oswalt
- eBPF maps documentation

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