1 Introduction

How do we construct reliable, portable, efficient, and secure computer systems? An essential component is the computer's *operating system* — the software that manages a computer's resources.

First, the bad news: operating systems concepts are among the most complex in computer science. A modern, general-purpose operating system can exceed 50 million lines of code, or in other words, more than a thousand times longer than this textbook. New operating systems are being written all the time: if you use an e-book reader, tablet, or smartphone, an operating system is managing your device. Given this inherent complexity, we limit our focus to the essential concepts that every computer scientist should know.

Now the good news: operating systems concepts are also among the most accessible in computer science. Many topics in this book will seem familiar to you — if you have ever tried to do two things at once, or picked the "wrong" line at a grocery store, or tried to keep a roommate or sibling from messing with your things, or succeeded at pulling off an April Fool's joke. Each of these activities has an analogue in operating systems. It is this familiarity that gives us hope that we can explain how operating systems work in a single textbook. All we assume of the reader is a basic understanding of the operation of a computer and the ability to read pseudo-code.

Why is studying operating systems useful?

We believe that understanding how operating systems work is essential for any student interested in building modern computer systems. Of course, everyone who uses a computer or a smartphone — or even a modern toaster — uses an operating system, so understanding the function of an operating system is useful to most computer scientists. This book aims to go much deeper than that, to explain operating system internals that we rely on every

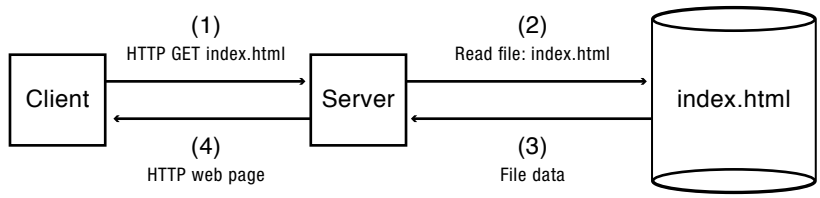

Figure 1.1: The operation of a web server. The client machine sends an HTTP GET request to the web server. The server decodes the packet, reads the file, and sends the contents back to the client.

day without realizing it.

Software engineers use many of the same technologies and design patterns as those used in operating systems to build other complex systems. Whether your goal is to work on the internals of an operating system kernel — or to build the next generation of software for cloud computing, secure web browsers, game consoles, graphical user interfaces, media players, databases, or multicore software — the concepts and abstractions needed for reliable, portable, efficient and secure software are much the same. In our experience, the best way to learn these concepts is to study how they are used in operating systems, but we hope you will apply them to a much broader range of computer systems.

To get started, consider the web server in Figure 1.1. Its behavior is amazingly simple: it receives a packet containing the name of the web page from the network, as an HTTP GET request. The web server decodes the packet, reads the file from disk, and sends the contents of the file back over the network to the user's machine.

What challenges does a web client or web server operating system face?

Part of an operating system's job is to make it easy to write applications like web servers. But digging a bit deeper, this simple story quickly raises as many questions as it answers:

- Many web requests involve both data and computation. For example, the Google home page presents a simple text box, but each search query entered in that box consults data spread over many machines. To keep their software manageable, web servers often invoke helper applications, e.g., to manage the actual search function. The main web server must be able to communicate with the helper applications for this to work. How does the operating system enable multiple applications to communicate with each other?

- What if two users (or a million) request a web page from the server at the same time? A simple approach might be to handle each request

in turn. If any individual request takes a long time, however, every other request must wait for it to complete. A faster, but more complex, solution is to *multitask*: to juggle the handling of multiple requests at once. Multitasking is especially important on modern multicore computers, where each processor can handle a different request at the same time. How does the operating system enable applications to do multiple things at once?

- For better performance, the web server might want to keep a copy, sometimes called a *cache*, of recently requested pages. In this way, if multiple users request the same page, the server can respond to subsequent requests more quickly from the cache, rather than starting each request from scratch. This requires the web server to coordinate, or *synchronize*, access to the cache's data structures by possibly thousands of web requests at the same time. How does the operating system synchronize application access to shared data?

- To customize and animate the user experience, web servers typically send clients scripting code along with the contents of the web page. But this means that clicking on a link can cause someone else's code to run on your computer. How does the client operating system protect itself from compromise by a computer virus surreptitiously embedded into the scripting code?

- Suppose the web site administrator uses an editor to update the web page. The web server must be able to read this file. How does the operating system store the bytes on disk so that the web server can find and read them?

- Taking this a step further, the administrator may want to make a consistent set of changes to the web site so that embedded links are not left dangling, even temporarily. How can the operating system let users make a set of changes to a web site, so that requests see either the old or new pages, but not a combination of the two?

- What happens when the client browser and the web server run at different speeds? If the server tries to send a web page to the client faster than the client can render the page on the screen, where are the contents of the file stored in the meantime? Can the operating system decouple the client and server so that each can run at its own speed without slowing the other down?

- As demand on the web server grows, the administrator may need to move to more powerful hardware, with more memory, more processors, faster network devices, and faster disks. To take advantage of new hardware, must the web server be re-written each time, or can it be written in a hardware-independent fashion? What about the operating system — must it be re-written for every new piece of hardware?

We could go on, but you get the idea. This book will help you understand the answers to these and many more questions.

Chapter roadmap: The rest of this chapter discusses three topics in detail:

- **Operating System Definition.** What is an operating system, and what does it do? (Section 1.1)
- **Operating System Evaluation.** What design goals should we look for in an operating system? (Section 1.2)
- **Operating Systems: Past, Present, and Future.** How have operating systems evolved, and what new functionality are we likely to see in future operating systems? (Section 1.3)

1.1 What Is An Operating System?

operating system

An *operating system* (OS) is the layer of software that manages a computer's resources for its users and their applications. Operating systems run in a wide range of computer systems. They may be invisible to the end user, controlling embedded devices such as toasters, gaming systems, and the many computers inside modern automobiles and airplanes. They are also essential to more general-purpose systems such as smartphones, desktop computers, and servers.

Our discussion will focus on general-purpose operating systems because the technologies they need are a superset of those needed for embedded systems. Increasingly, operating systems technologies developed for general-purpose computing are migrating into the embedded sphere. For example, early mobile phones had simple operating systems to manage their hardware and to run a handful of primitive applications. Today, smartphones — phones capable of running independent third-party applications — are the fastest growing segment of the mobile phone business. These devices require much more complete operating systems, with sophisticated resource management, multi-tasking, security and failure isolation.

Likewise, automobiles are increasingly software controlled, raising a host of operating system issues. Can anyone write software for your car? What if the software fails while you are driving down the highway? Can a car's operating system be hijacked by a computer virus? Although this might seem far-fetched, researchers recently demonstrated that they could remotely turn off a car's braking system through a computer virus introduced into the car's computers via a hacked car radio. A goal of this book is to explain how to build more reliable and secure computer systems in a variety of contexts.

For general-purpose systems, users interact with applications, applications execute in an environment provided by the operating system, and the operating system mediates access to the underlying hardware, as shown in

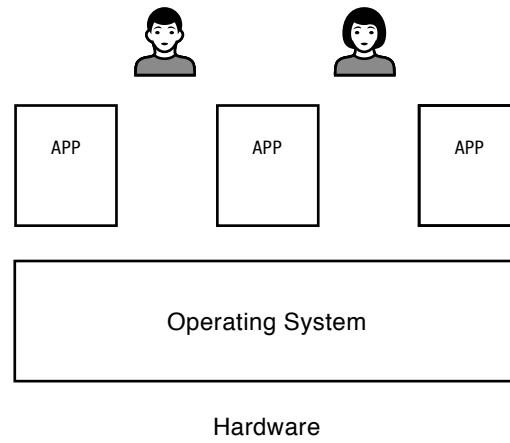

Figure 1.2: A general-purpose operating system is a layer of software that manages a computer's resources for its users and applications.

Figure 1.2 and expanded in Figure 1.3. How can an operating system run multiple applications? For this, operating systems need to play three roles:

What roles does an OS play?

1. **Referee.** Operating systems manage resources shared between different applications running on the same physical machine. For example, an operating system can stop one program and start another. Operating systems isolate applications from each other, so a bug in one application does not corrupt other applications running on the same machine. An operating system must also protect itself and other applications from malicious computer viruses. And since the applications share physical resources, the operating system needs to decide which applications get which resources and when.

2. **Illusionist.** Operating systems provide an abstraction of physical hardware to simplify application design. To write a "Hello world!" program, you do not need (or want!) to think about how much physical memory the system has, or how many other programs might be sharing the computer's resources. Instead, operating systems provide the illusion of nearly infinite memory, despite having a limited amount of physical memory. Likewise, they provide the illusion that each program has the computer's processors entirely to itself. Obviously, the reality is quite different! These illusions let you write applications independently of the amount of physical memory on the system or the physical number of processors. Because applications are written to a higher level of abstraction, the operating system can invisibly change the amount of resources assigned to each application.

3. **Glue.** Operating systems provide a set of common services that facilitate

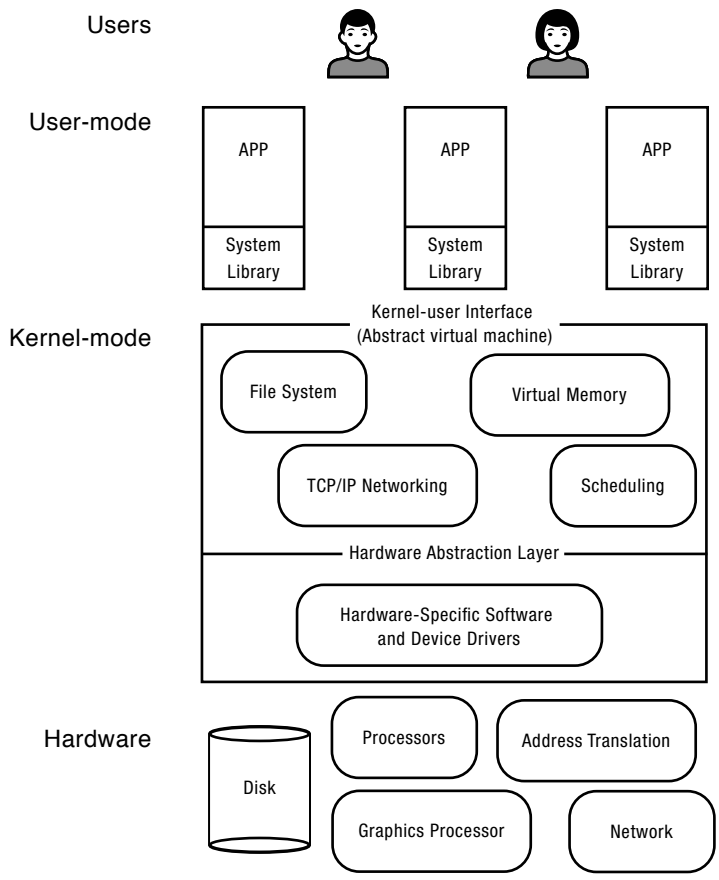

Figure 1.3: This shows the structure of a general-purpose operating system, as an expansion on the simple view presented in Figure 1.2. At the lowest level, the hardware provides processors, memory, and a set of devices for storing data and communicating with the outside world. The hardware also provides primitives that the operating system can use for fault isolation and synchronization. The operating system runs as the lowest layer of software on the computer. It contains both a device-specific layer for managing the myriad hardware devices and a set of device-independent services provided to applications. Since the operating system must isolate malicious and buggy applications from other applications or the operating system itself, much of the operating system runs in a separate execution environment protected from application code. A portion of the operating system can also run as a system library linked into each application. In turn, applications run in an execution context provided by the operating system kernel. The application context is much more than a simple abstraction on top of hardware devices: applications execute in a virtual environment that is more constrained (to prevent harm), more powerful (to mask hardware limitations), and more useful (via common services) than the underlying hardware.

sharing among applications. As a result, cut and paste works uniformly across the system; a file written by one application can be read by another. Many operating systems provide common user interface routines so applications can have the same "look and feel." Perhaps most importantly, operating systems provide a layer separating applications from hardware input and output (I/O) devices so applications can be written independently of the specific keyboard, mouse, and disk drive in use on a particular computer.

We next discuss these three roles in greater detail.

1.1.1 Resource Sharing: Operating System as Referee

What happens when multiple applications share resources?

Sharing is central to most uses of computers. Right now, my laptop is running a browser, podcast library, text editor, email program, document viewer, and newspaper. The operating system must somehow keep all of these activities separate, yet allow each the full capacity of the machine if the others are not running. At a minimum, when one program stops running, the operating system should let me run another. Better still, the operating system should let multiple applications run at the same time, so I can read email while I download a security patch to the system software.

Even individual applications can do multiple tasks at once. For instance, a web server's responsiveness improves if it handles multiple requests concurrently rather than waiting for each to complete before starting the next one. The same holds for the browser—it is more responsive if it can start rendering a page while the rest of the page is transferring. On multiprocessors, the computation inside a parallel application can be split into separate units that can be run independently for faster execution. The operating system itself is an example of software written to do multiple tasks at once. As we will illustrate throughout the book, the operating system is a customer of its own abstractions.

Sharing raises several challenges for an operating system:

- **Resource allocation.** The operating system must keep all simultaneous activities separate, allocating resources to each as appropriate. A computer usually has only a few processors and a finite amount of memory, network bandwidth, and disk space. When there are multiple tasks to do at the same time, how should the operating system decide how many resources to give to each? Seemingly trivial differences in how resources are allocated can impact user-perceived performance. As we will see in Chapter 9, an operating system that allocates too little memory to a program slows down not only that particular program, but often other applications as well.

  To illustrate the difference between execution on a physical machine versus on the abstract machine provided by the operating system, what should happen if an application executes an infinite loop?

```
while (true) {
    ;
}
```

If programs ran directly on raw hardware, this code fragment would lock up the computer, making it completely non-responsive to user input. If the operating system ensures that each program gets its own slice of the computer's resources, a specific application might lock up, but other programs could proceed unimpeded. Additionally, the user could ask the operating system to force the looping program to exit.

fault isolation

- **Isolation.** An error in one application should not disrupt other applications, or even the operating system itself. This is called *fault isolation*. Anyone who has taken an introductory computer science class knows the value of an operating system that can protect itself and other applications from programmer bugs. Debugging would be vastly harder if an error in one program could corrupt data structures in other applications. Likewise, downloading and installing a screen saver or other application should not crash unrelated programs, provide a way for a malicious attacker to surreptitiously install a computer virus, or let one user access or change another's data without permission.

  Fault isolation requires restricting the behavior of applications to less than the full power of the underlying hardware. Otherwise, any application downloaded off the web, or any script embedded in a web page, could completely control the machine. Any application could install spyware into the operating system to log every keystroke you type, or record the password to every web site you visit. Without fault isolation provided by the operating system, any bug in any program might irretrievably corrupt the disk. Error-prone or malignant applications could cause all sorts of havoc.

- **Communication.** The flip side of isolation is the need for communication between different applications and different users. For example, a web site may be implemented by a cooperating set of applications: one to select advertisements, another to cache recent results, yet another to fetch and merge data from disk, and several more to cooperatively scan the web for new content to index. For this to work, the various programs must communicate with one another. If the operating system prevents bugs and malicious users and applications from affecting other users and their applications, how does it also support communication to share results? In setting up boundaries, an operating system must also allow those boundaries to be crossed in carefully controlled ways when the need arises.

In its role as referee, an operating system is somewhat akin to that of a particularly patient kindergarten teacher. It balances needs, separates conflicts,

and facilitates sharing. One user should not be allowed to monopolize system resources or to access or corrupt another user's files without permission; a buggy application should not be able to crash the operating system or other unrelated applications; and yet, applications must also work together. Enforcing and balancing these concerns is a central role of the operating system.

1.1.2 Masking Limitations: Operating System as Illusionist

What happens when applications need more resources than the hardware provides?

A second important role of an operating system is to mask the restrictions inherent in computer hardware. Physical constraints limit hardware resources — a computer has only a limited number of processors and a limited amount of physical memory, network bandwidth, and disk. Further, since the operating system must decide how to divide its fixed resources among the various applications running at each moment, a particular application can have differing amounts of resources from time to time, even when running on the same hardware. While some applications are designed to take advantage of a computer's specific hardware configuration and resource assignment, most programmers prefer to use a higher level of abstraction.

virtualization

Virtualization provides an application with the illusion of resources that are not physically present. For example, the operating system can provide the abstraction that each application has a dedicated processor, even though at a physical level there may be only a single processor shared among all the applications running on the computer.

With the right hardware and operating system support, most physical resources can be virtualized. For example, hardware provides only a small, finite amount of memory, while the operating system provides applications the illusion of a nearly infinite amount of virtual memory. Wireless networks drop or corrupt packets; the operating system masks these failures to provide the illusion of a reliable service. At a physical level, magnetic disk and flash RAM support block reads and writes, where the size of the block depends on the physical device characteristics, addressed by a device-specific block number. Most programmers prefer to work with byte-addressable files organized by name into hierarchical directories. Even the type of processor can be virtualized to allow the same, unmodified application to run on a smartphone, tablet, and laptop computer.

Pushing this one step further, some operating systems virtualize the entire computer, running the operating system as an application on top of another operating system (see Figure 1.4). This is called creating a *virtual machine*. The operating system running in the virtual machine, called the *guest operating system*, thinks it is running on a real, physical machine, but this is an illusion presented by the true operating system running underneath.

virtual machine

guest operating system

What is the purpose of a virtual machine?

One benefit of a virtual machine is application portability. If a program runs only on an old version of an operating system, it can still work on a new system running a virtual machine. The virtual machine hosts the application on the old operating system, running atop the new one. Virtual machines also

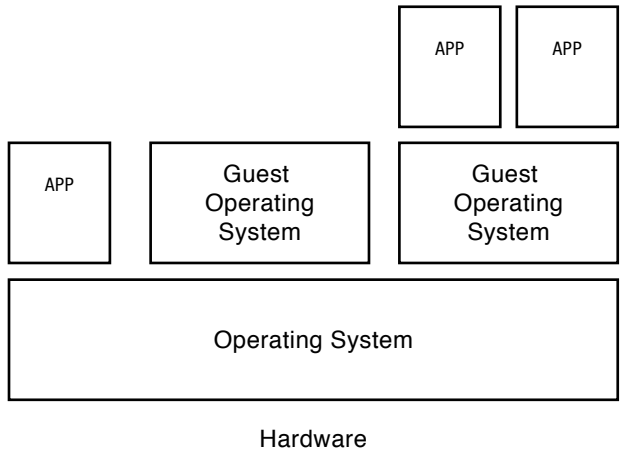

Figure 1.4: A guest operating system running inside a virtual machine.

aid debugging. If an operating system can be run as an application, then its developers can set breakpoints, stop the kernel, and single step their code just as they would when debugging an application.

Throughout the book, we discuss techniques that the operating system uses to accomplish these and other illusions. In each case, the operating system provides a more convenient and flexible programming abstraction than that provided by the underlying hardware.

1.1.3 Providing Common Services: Operating System as Glue

Can we raise the level of abstraction above bare hardware?

Operating systems play a third key role: providing a set of common, standard services to applications to simplify and standardize their design. An example is the web server described earlier in this chapter. The operating system hides the specifics of how the network and disk devices work, providing a simpler abstraction based on receiving/sending reliable streams of bytes and reading/writing named files. This lets the web server focus on its core task — decoding incoming requests and filling them — rather than on formatting data into individual network packets and disk blocks.

An important reason for the operating system to provide common services, rather than letting each application provide its own, is to facilitate sharing among applications. The web server must be able to read the file that the text editor wrote. For applications to share files, they must be stored in a standard format, with a standard system for managing file directories. Most operating systems also provide a standard way for applications to pass messages and to share memory.

The choice of which services an operating system should provide is often judgment call. For example, computers can come configured with a blizzard of

different devices: different graphics co-processors and pixel formats, different network interfaces (WiFi, Ethernet, and Bluetooth), different disk drives (SCSI, IDE), different device interfaces (USB, Firewire), and different sensors (GPS, accelerometers), not to mention different versions of each. Most applications can ignore these differences, by using only a generic interface provided by the operating system. For other applications, such as a database, the specific disk drive may matter quite a bit. For applications that can operate at a higher level of abstraction, the operating system serves as an interoperability layer so that both applications and devices can evolve independently.

Another standard service in most modern operating systems is the graphical user interface library. Both Microsoft's and Apple's operating systems provide a set of standard user interface widgets. This facilitates a common "look and feel" to users so that frequent operations — such as pull down menus and "cut" and "paste" commands — are handled consistently across applications.

Most of the code in an operating system implements these common services. However, much of the complexity of operating systems is due to resource sharing and the masking of hardware limits. Because common service code uses the abstractions provided by the other two operating system roles, this book will focus primarily on the operating system as a referee and as an illusionist.

1.1.4 Operating System Design Patterns

Are the roles of referee, illusionist, and glue unique to operating systems?

The challenges that operating systems address are not unique — they apply to many different computer domains. Many complex software systems have multiple users, run programs written by third-party developers, and/or need to coordinate many simultaneous activities. These pose questions of resource allocation, fault isolation, communication, abstractions of physical hardware, and how to provide a useful set of common services for software developers. Not only are the challenges the same, but often the solutions are, as well: these systems use many of the design patterns and techniques described in this book.

We next describe some of the systems with design challenges similar to those found in operating systems:

- **Cloud computing** (Figure 1.5) is a model of computing where applications run on shared computing and storage infrastructure in large-scale data centers instead of on the user's own computers. Cloud computing must address many of the same issues as in operating systems in terms of sharing, abstraction, and common services.

  - **Referee.** How are resources allocated between competing applications running in the cloud? How are buggy or malicious applications prevented from disrupting other applications?