Operating Systems

Principles & Practice

SECOND EDITION

Thomas Anderson
Michael Dahlin

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

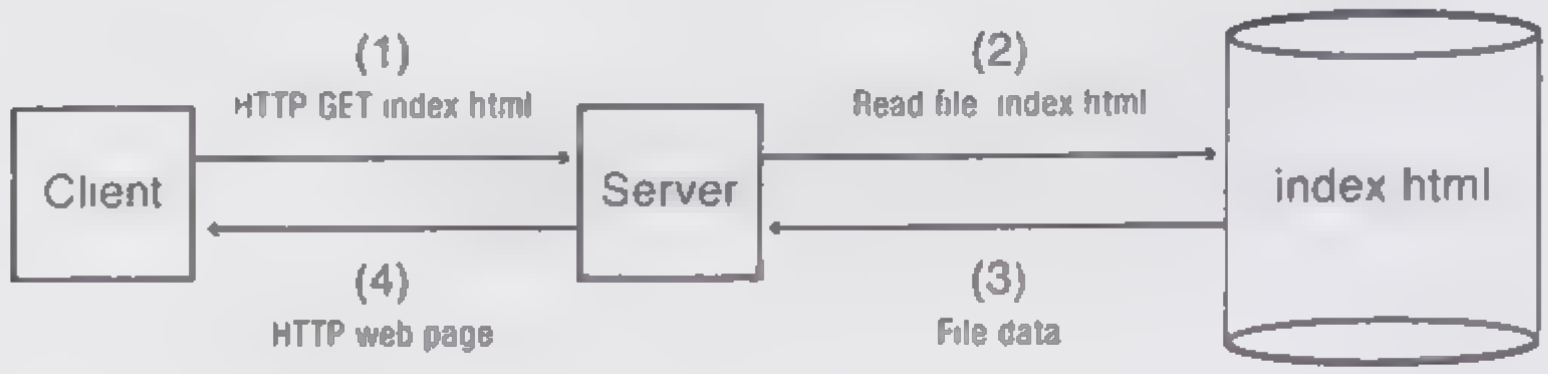

Figure 1.1: The operation of a web server. The client machine sends an HTTP GET request to the web server. The server decodes the packet, reads the file, and sends the contents back to the client

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

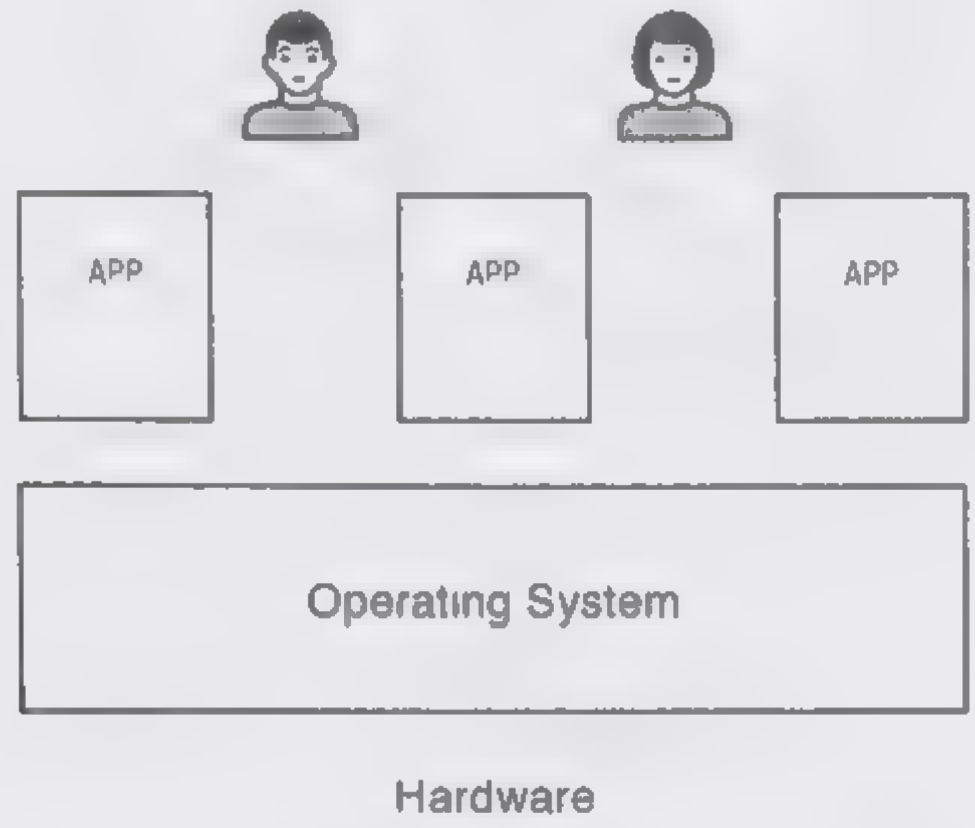

Figure 1.2: A general purpose operating system is a layer of software that manages a computer's resources for its users and applications.

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

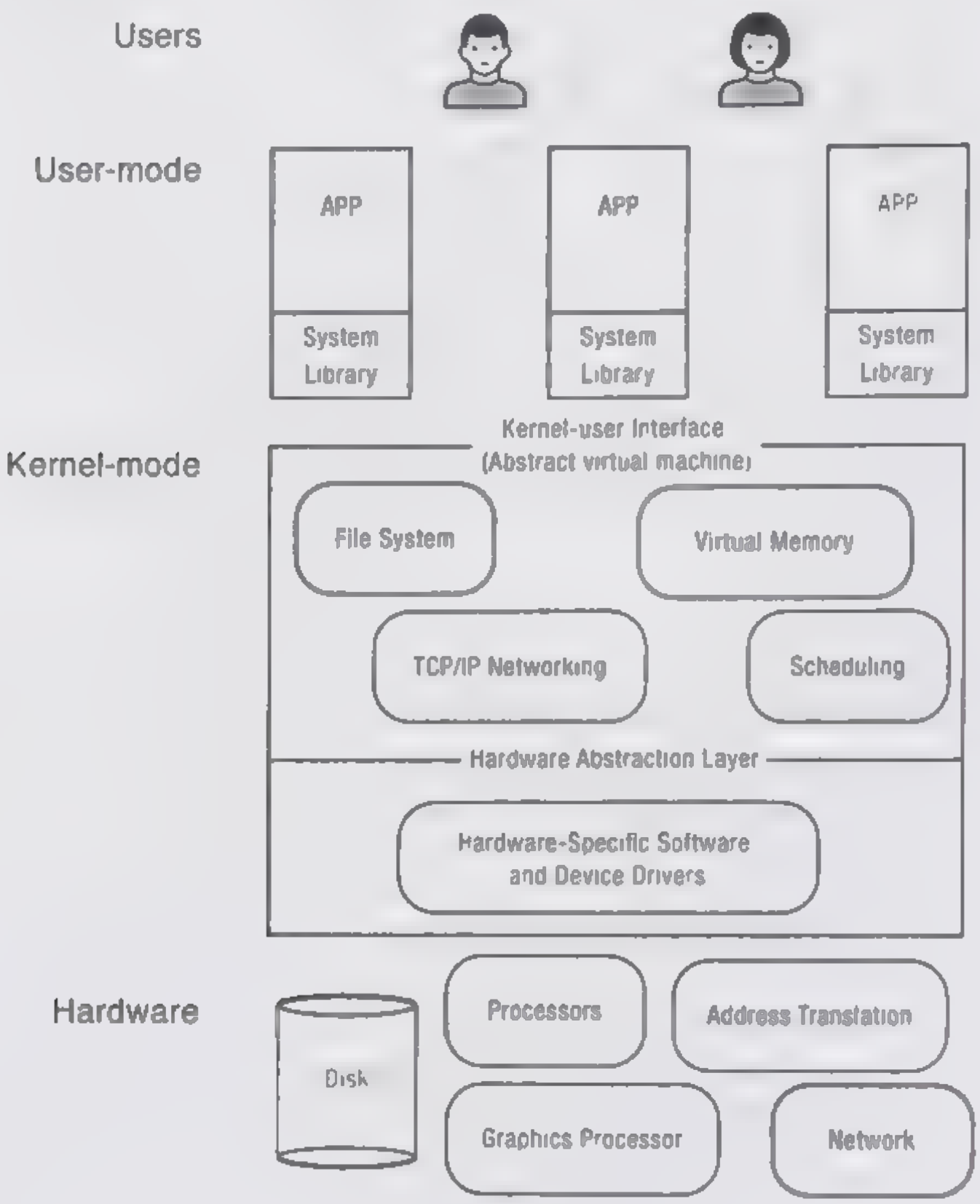

Figure 1.3: This shows the structure of a general-purpose operating system, as an expansion on the simple view presented in Figure 1.2. At the lowest level, the hardware provides processors, memory, and a set of devices for storing data and communicating with the outside world. The hardware also provides 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.

virtual machine

guest operating system

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.

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

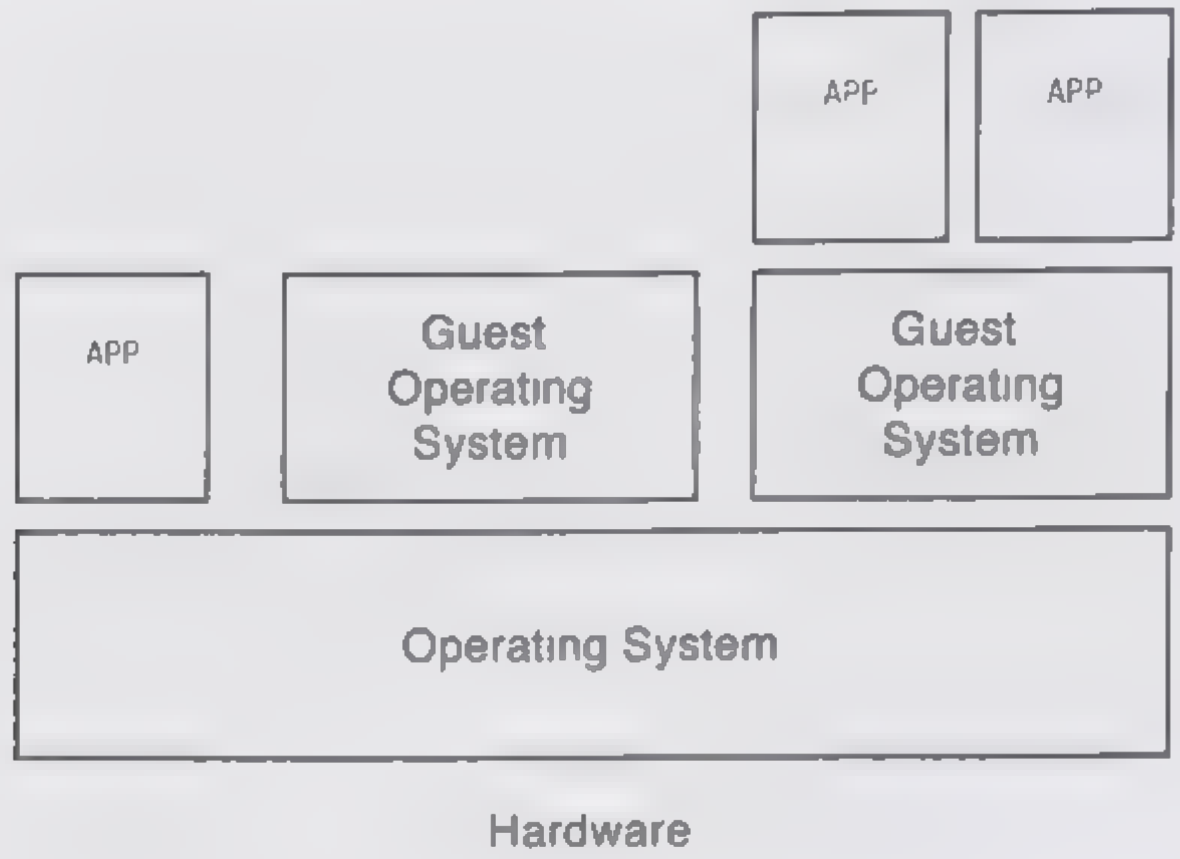

Figure 1.4: A guest operating system running inside a virtual machine.

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 a 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?

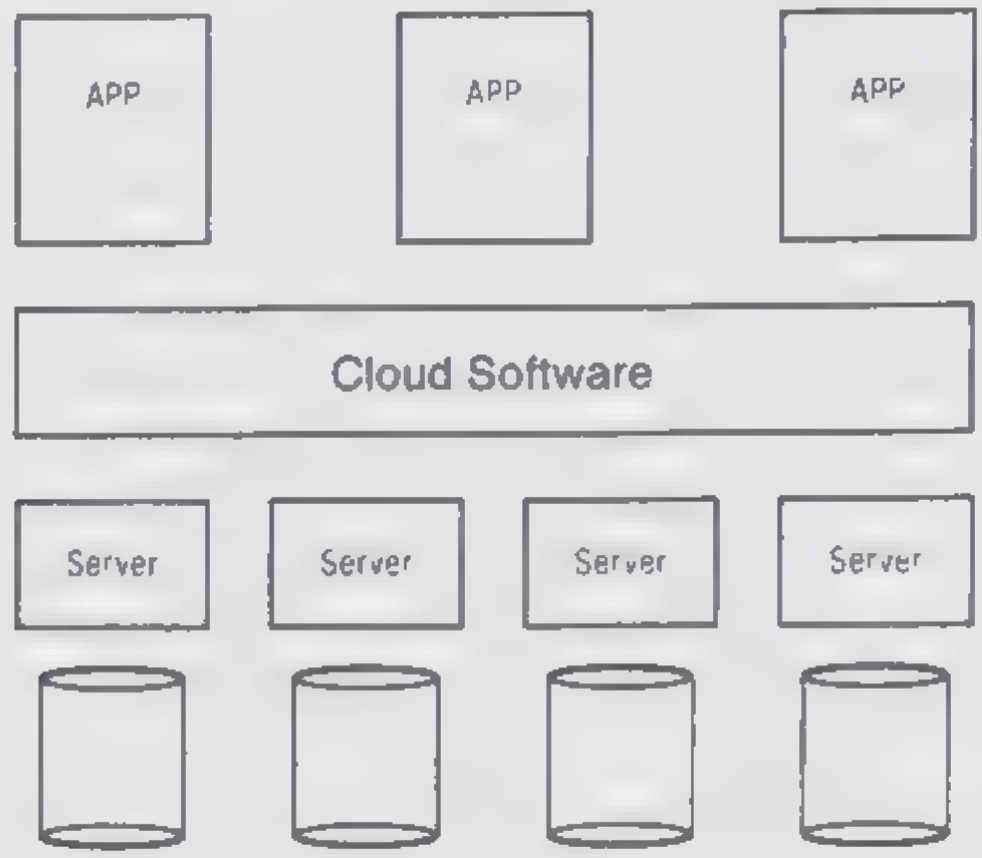

Figure 1.5: Cloud computing software provides a convenient abstraction of server resources to cloud applications.

  - **Illusionist.** The computing resources in the cloud are continually evolving; what abstractions are provided to isolate application developers from changes in the underlying hardware?
  - **Glue.** Cloud services often distribute their work across different machines. What abstractions should cloud software provide to help services coordinate and share data between their various activities?

- **Web browsers** (Figure 1.6), such as Chrome, Internet Explorer, Firefox, and Safari, play a role similar to an operating system. Browsers load and display web pages, but, as we mentioned earlier, many pages embed scripting programs that the browser must execute. These scripts can be buggy or malicious, hackers have used them to take over vast numbers of home computers. Like an operating system, the browser must isolate the user, other web sites, and even the browser itself from errors or malicious activity by these scripts. Similarly, most browsers have a plug-in architecture for supporting extensions, and these extensions must also be isolated to prevent them from causing harm.
  - **Referee.** How can a browser ensure responsiveness when a user has multiple tabs open with each tab running a script from a different web site? How can we limit web scripts and plug-ins to prevent bugs from crashing the browser and malicious scripts from accessing sensitive user data?
  - **Illusionist.** Many web services are geographically distributed to improve the user experience. Not only does this put servers closer to users, but if one server crashes or its network connection has

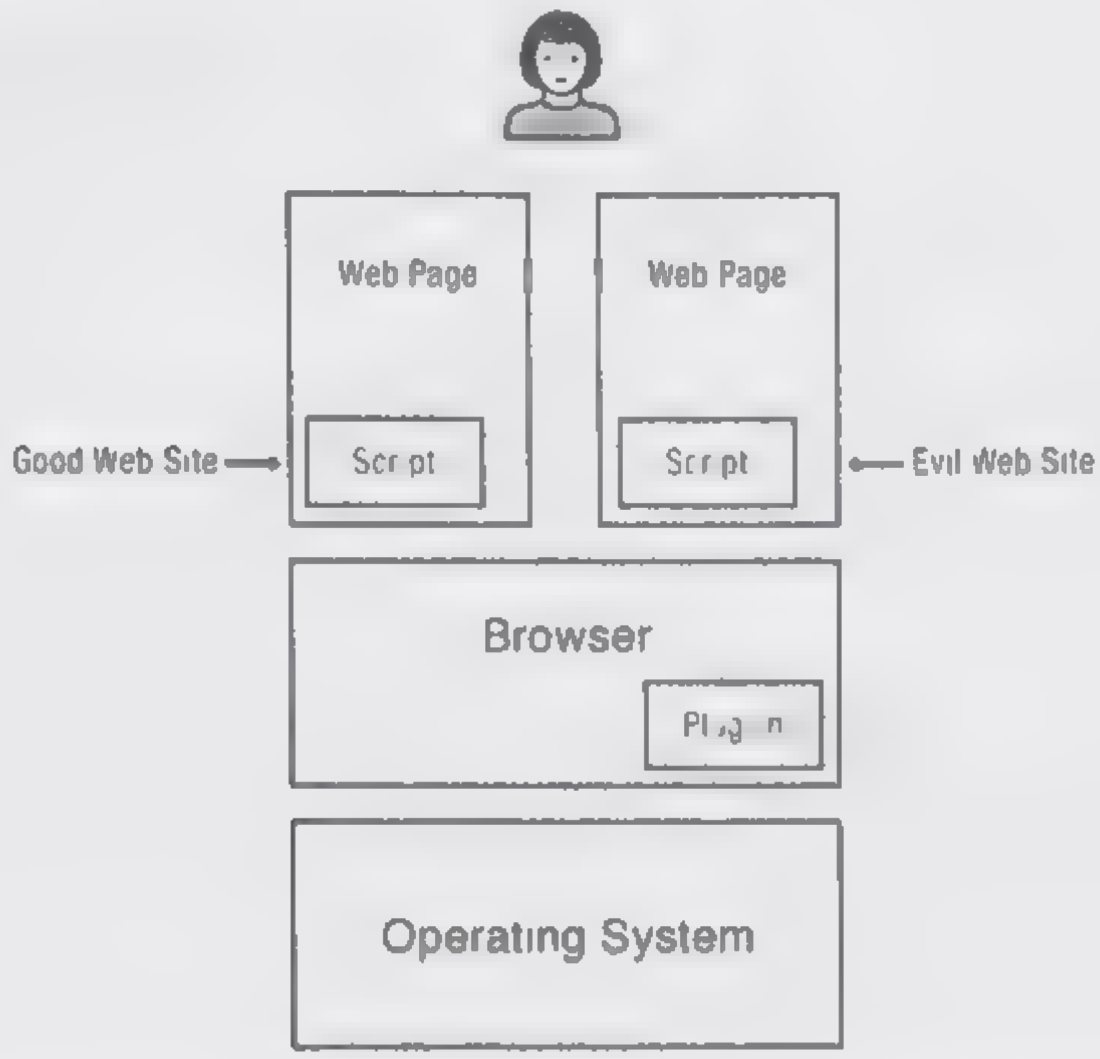

Figure 1.6: A web browser isolates scripts and plug-ins from accessing privileged resources on the host operating system.

problems, a browser can connect to a different site. The user in most cases does not notice the difference, even when updating a shopping cart or web form. How does the browser make server changes transparent to the user?

- **Glue.** How does the browser achieve a portable execution environment for scripts that works consistently across operating systems and hardware platforms?

- **Media players**, such as Flash and Silverlight, are often packaged as browser plug-ins, but they themselves provide an execution environment for scripting programs. Thus, these systems face many of the same issues as both browsers and operating systems on which they run: isolation of buggy or malicious code, concurrent background and foreground tasks, and plug-in architectures.

  - **Referee.** Media players are often in the news for being vulnerable to some new, malicious attack. How should media players sandbox malicious or buggy scripts to prevent them from corrupting the host machine?

  - **Illusionist.** Media applications are often both computationally intensive and highly interactive. How do they coordinate foreground and background activities to maintain responsiveness?

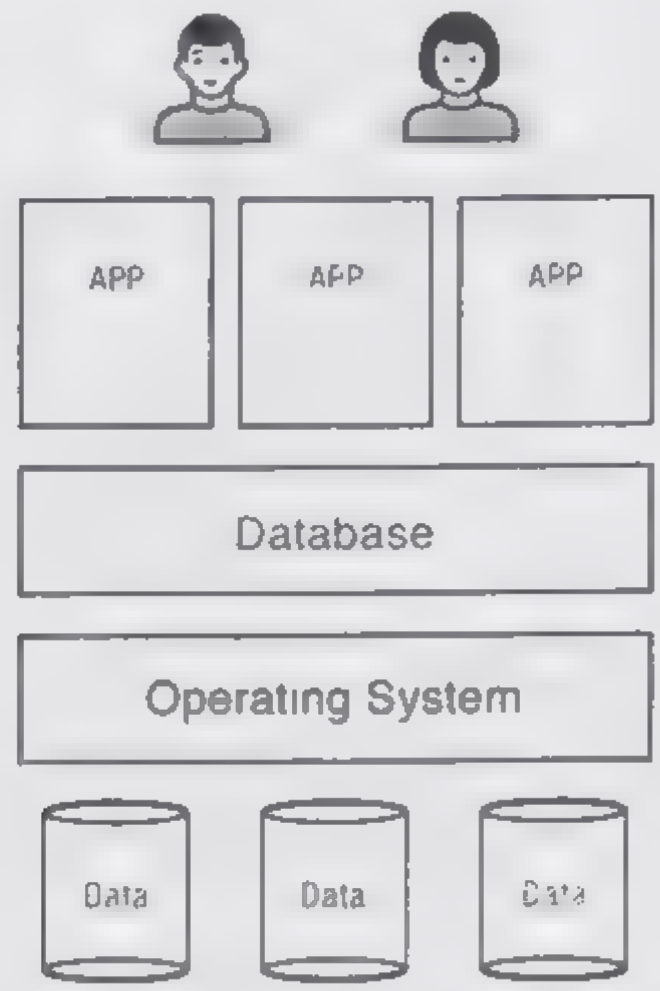

Figure 1.7: Databases perform many of the tasks of an operating system: they allocate resources among user queries to ensure responsiveness; they mask differences in the underlying operating system and hardware, and they provide a convenient programming abstraction to developers.

  - **Glue.** High performance graphics hardware rapidly evolves in response to the demands of the video game market. How do media players provide a set of standard API's for scripts to work across a diversity of graphics accelerators?

- **Multiplayer games** often have extensibility API's to allow third party software vendors to extend the game in significant ways. Often these extensions are miniature games in their own right. Yet game extensions must also be prevented from breaking the overall rules of the game.

  - **Referee.** Many games try to offload work to client machines to reduce server load and improve responsiveness, but this opens up games to the threat of users installing specialized extensions to gain an unfair advantage. How do game designers set limits for extensions and game players to ensure a level playing field?
  - **Illusionist.** If objects in the game are spread across client and server machines, is that distinction visible to extension code or is the interface at a higher level?
  - **Glue.** Most successful games have a large number of extensions; how should a game designer set up their API's to make it easier to foster a community of developers?

- **Multi-user database systems** (Figure 1.7), such as Oracle and Microsoft's SQL Server, allow large organizations to store, query, and update large data sets, such as detailed records of every purchase ever made at Amazon or Walmart. Large scale data analysis greatly optimizes business operations, but, as a consequence, databases face many of the same challenges as operating systems. They are simultaneously accessed by many different users in many different locations. They therefore must allocate resources among different user requests, isolate concurrent updates to shared data, and ensure that data is stored consistently on disk. In fact, several of the techniques we discuss in Chapter 14 were originally developed for database systems.
  - **Referee.** How should resources be allocated among the various users of a database? How does the database enforce data privacy so that only authorized users access relevant data?
  - **Illusionist.** How does the database mask machine failures so that data is always stored consistently regardless of when the failure occurs?
  - **Glue.** What common services make it easier to develop database applications?

- **Parallel applications** are programs designed to take advantage of multiple processors on a single computer. Each application divides its work onto a fixed number of processors and must ensure that accesses to shared data structures are coordinated to preserve consistency. While some parallel programs directly use the services provided by the underlying operating system, others need careful control of the assignment of work to processors to achieve good performance. These systems interpose a runtime system on top of the operating system to manage user level parallelism, essentially building a mini-operating system on top of the underlying one.
  - **Referee.** When there are more tasks to perform than processors, how does the runtime system decide which tasks to perform first?
  - **Illusionist.** How does the runtime system hide physical details of the hardware from the programmer, such as the number of processors or the interprocessor communication latency?
  - **Glue.** Highly concurrent data structures can make it easier to write efficient parallel programs; how do we program trees, hash tables, and lists so that they can be used by multiple processors at the same time?

- **The Internet** is used everyday by a huge number of people, but at the physical layer, those users share the same underlying resources. How should the Internet handle resource contention? Because of its diverse

user base, the Internet is rife with malicious behavior, such as denial-of-service attacks that flood traffic on certain links to prevent legitimate users from communicating. Various attempts are underway to design solutions that will let the Internet continue to function despite such attacks.

- **Referee.** Should the Internet treat all users identically (e.g., network neutrality) or should ISPs be able to favor some uses over others? Can the Internet be re-designed to prevent denial-of-service, spam, phishing, and other malicious behaviors?
- **Illusionist.** The Internet provides the illusion of a single worldwide network that can deliver a packet from any machine on the Internet to any other machine. However, network hardware is composed of many discrete network elements with: (i) the ability to transmit limited size packets over a limited distance, and (ii) some chance that packets will be garbled in the process. The Internet transforms the network into something more useful for applications like the web — a facility to reliably transmit data of arbitrary length, anywhere in the world.
- **Glue.** The Internet protocol suite was explicitly designed to act as an interoperability layer that lets network applications evolve independently of changes in network hardware, and vice versa. Does the success of the Internet hold any lessons for operating system design?

Many of these systems use the same techniques and design patterns as operating systems. Studying operating systems is a great way to understand how these others systems work. In a few cases, different mechanisms are used to achieve the same goals, but, even here, the boundaries are fuzzy. For example, browsers often use compile-time checks to prevent scripts from gaining control over them, while most operating systems use hardware-based protection to limit application programs from taking over the machine. More recently, however, some smartphone operating systems have begun to use the same compile-time techniques as browsers to protect against malicious mobile applications. In turn, some browsers have begun to use operating system hardware-based protection to improve the isolation they provide.

To avoid spreading our discussion too thinly, this book focuses on how operating systems work. Just as it is easier to learn a second computer programming language after you become fluent in the first, it is better to see how operating systems principles apply in one context before learning how they can be applied in other settings. We hope and expect, however, that you will be able to apply the concepts in this book more widely than just operating system design.

1.2 Operating System Evaluation

What design goals should we look for in an operating system?

Having defined what an operating system does, how should we choose among alternative designs? We discuss several desirable criteria for operating systems:

- **Reliability and Availability.** Does the operating system do what you want?
- **Security.** Can the operating system be corrupted by an attacker?
- **Portability.** Is the operating system easy to move to new hardware platforms?
- **Performance.** Is the user interface responsive, or does the operating system impose too much overhead?
- **Adoption.** How many other users are there for this operating system?

In many cases, tradeoffs between these criteria are inevitable — improving a system along one dimension may hurt it along another. We conclude this section with some concrete examples of design tradeoffs.

1.2.1 Reliability and Availability

reliability

Perhaps the most important characteristic of an operating system is its reliability. *Reliability* means that a system does exactly what it is designed to do. As the lowest level of software running on the system, operating system errors can have devastating and hidden effects. If the operating system breaks, you may not be able to get work done, and in some cases, you may even lose previous work, e.g., if the failure corrupts files on disk. By contrast, application failures can be much more benign, precisely because operating systems provide fault isolation and a rapid and clean restart after an error.

Making an operating system reliable is challenging. Operating systems often operate in a hostile environment, one where computer viruses and other malicious code try to take control of the system by exploiting design or implementation errors in the operating system's defenses.

Unfortunately, the most common ways to improve software reliability, such as running test cases for common code paths, are less effective when applied to operating systems. Since malicious attacks can target a specific vulnerability precisely to cause execution to follow a rare code path, everything must work correctly for the operating system to be reliable. Even without intentionally malicious attacks, extremely rare corner cases can occur regularly: for an operating system with a million users, a once in a billion event will eventually occur to someone.

availability

A related concept is *availability*, the percentage of time that the system is usable. A buggy operating system that crashes frequently, losing the user's work, is both unreliable and unavailable. A buggy operating system that

crashes frequently but never loses the user's work and cannot be subverted by a malicious attack is reliable but unavailable. An operating system that has been subverted but continues to appear to run normally while logging the user's keystrokes is unreliable but available.

Thus, both reliability *and* availability are desirable. Availability is affected by two factors: the frequency of failures, measured as the *mean time to failure* (*MTTF*), and the time it takes to restore a system to a working state after a failure (for example, to reboot), called the *mean time to repair* (*MTTR*). Availability can be improved by increasing the MTTF or reducing the MTTR.

mean time to failure

mean time to repair

Throughout this book, we will present various approaches to improving operating system reliability and availability. In many cases, the abstractions may seem at first glance overly rigid and formulaic. It is important to realize this is done on purpose! Only precise abstractions provide a basis for constructing reliable and available systems.

1.2.2 Security

security

privacy

Two concepts closely related to reliability are security and privacy. *Security* means the computer's operation cannot be compromised by a malicious attacker. *Privacy* is an aspect of security: data stored on the computer is only accessible to authorized users.

Alas, no useful computer is perfectly secure! Any complex piece of software has bugs, and seemingly innocuous bugs can be exploited by an attacker to gain control of the system. Or the computer hardware might be tampered with, to provide access to the attacker. Or the computer's administrator might be untrustworthy, using his or her credentials to steal user data. Or an OS software developer might be untrustworthy, inserting a backdoor for the attacker to gain access to the system.

Nevertheless, an operating system can be, and should be, designed to minimize its vulnerability to attack. For example, strong fault isolation can prevent third party applications from taking over the system. Downloading and installing a screen saver or other application should not provide a way for an attacker to surreptitiously install a *computer virus* on the system. A computer program that modifies an operating system or application to copy itself from computer to computer without the computer owner's permission or knowledge. Once installed on a computer, a virus often provides the attacker control over the system's resources or data. An example computer virus is a keylogger: a program that modifies the operating system to record every keystroke entered by the user and send them back to the attacker's machine. In this way, the attacker could gain access to the user's passwords, bank account numbers, and other private information. Likewise, a malicious screen saver might surreptitiously scan the disk for files containing personal information or turn the system into an email spam server.

computer virus

Even with strong fault isolation, a system can be insecure if its applications are not designed for security. For example, the Internet email standard provides no strong assurance of the sender's identity, it is possible to form

an email message with anyone's email address in the "from" field, not necessarily the actual sender's. Thus, an email message can appear to be from someone (perhaps someone you trust), when in reality it is from the attacker and contains, as an attachment, a malicious virus that takes over the computer when the attachment is opened. By now, you are hopefully suspicious of clicking on any email attachment. Stepping back, the issue could be seen as a limitation of the interaction between the email system and the operating system. If the operating system provided a cheap and easy way to process an attachment in an isolated execution environment with limited capabilities, then even attachments containing viruses would do no harm.

Complicating matters is that the operating system must not only prevent unwanted access to shared data, it must also *allow* access in many cases. Users and programs must be able to interact with each other, so that it is possible to cut and paste text between different applications, and to share data written to disk or over the network. If each program were completely standalone and never needed to interact with any other program, then fault isolation by itself would be sufficient. However, we not only want to isolate programs from one another, but to easily share data between programs and between users.

enforcement

security policy

Thus, an operating system needs both an enforcement mechanism and a security policy. *Enforcement* is how the operating system ensures that only permitted actions are allowed. The *security policy* defines what is permitted who is allowed to access what data, and who can perform what operations.

Malicious attackers can target vulnerabilities in either enforcement mechanisms or security policies. An error in enforcement can allow an attacker to evade the policy, an error in the policy can allow the attacker access when it should have been prohibited.

1.2.3 Portability

portability

All operating systems provide applications with an abstraction of the underlying computer hardware, a *portable* abstraction is one that does not change as the hardware changes. A program written for Microsoft's Windows 8 should run correctly regardless of whether a specific graphics card is being used, whether persistent storage is provided via flash memory or rotating magnetic disk, or whether the network is Bluetooth, WiFi, or gigabit Ethernet.

Portability also applies to the operating system itself. As we have noted, operating systems are among the most complex software systems ever invented, making it impractical to re-write them from scratch every time new hardware is produced or a new application is developed. Instead, new operating systems are often derived, at least in part, from old ones. As one example, iOS, the operating system for the iPhone and iPad, was derived from the MacOS X code base.

As a result, most successful operating systems have a lifetime measured in decades. Microsoft Windows 8 originally began with the development of Windows NT starting in 1988. At that time, the typical computer was 10000 times less powerful, and with 10000 times less memory and disk storage,

than is the case today. Operating systems that last decades are no anomaly. Microsoft's prior operating system, MS/DOS, was introduced in 1981. It later evolved into the early versions of Microsoft Windows before finally being phased out around 2000.

This means that operating systems must be designed to support applications that have not yet been written and to run on hardware that has not yet been developed. Likewise, developers do not want to re-write applications when the operating system is ported from machine to machine. Sometimes, the importance of "future proofing" an operating system is discovered only in retrospect. Microsoft's first operating system, MS/DOS, was designed in 1981 assuming that personal computers would never have more than 640 KB of memory. This limitation was acceptable at the time, but today, even cellphones have orders of magnitude more memory than that.

abstract virtual machine

application programming interface

How might we design an operating system to achieve portability? As we illustrated earlier in Figure 1.3, it helps to have a simple, standard way for applications to interact with the operating system, the *abstract virtual machine (AVM)*. This is the interface provided by operating systems to applications, including: (i) the *application programming interface (API)*, the list of function calls the operating system provides to applications, (ii) the memory access model, and (iii) which instructions can be legally executed. For example, an instruction to change whether the hardware is executing trusted operating system code, or untrusted application code, must be available to the operating system but not to applications.

A well-designed operating system AVM provides a fixed point across which both application code and hardware can evolve independently. This is similar to the role of the Internet Protocol (IP) standard in networking. Distributed applications such as email and the web, written using IP, are insulated from changes in the underlying network technology (Ethernet, WiFi, optical). Equally important is that changes in applications, from email to instant messaging to file sharing, do not require simultaneous changes in the underlying hardware.

hardware abstraction layer

This notion of a portable hardware abstraction is so powerful that operating systems use the same idea internally: the operating system itself can largely be implemented independently of the hardware specifics. The interface that makes this possible is called the *hardware abstraction layer (HAL)*. It might seem that the operating system AVM and the operating system HAL should be identical, or nearly so — after all, both are portable layers designed to hide hardware details. The AVM must do more, however. As we noted, applications execute in a restricted, virtualized context and with access to high-level common services, while the operating system itself uses a procedural abstraction much closer to the actual hardware.

Today, Linux is an example of a highly portable operating system. It has been used as the operating system for web servers, personal computers, tablets, netbooks, e-book readers, smartphones, set top boxes, routers, WiFi access points, and game consoles. Linux is based on an operating system called UNIX, which was originally developed in the early 1970's. UNIX was written by a

small team of developers. It was designed to be compact, simple to program, and highly portable, but at some cost in performance. Over the years, UNIX's and Linux's portability and convenient programming abstractions have been keys to their success.

1.2.4 Performance

While the portability of an operating system becomes apparent over time, the performance of an operating system is often immediately visible to its users. Although we often associate performance with each individual application, the operating system's design can greatly affect the application's perceived performance. The operating system decides when an application can run, how much memory it can use, and whether its files are cached in memory or clustered efficiently on disk. The operating system also mediates application access to memory, the network, and the disk. It must avoid slowing down the critical path while still providing needed fault isolation and resource sharing between applications.

overhead

efficiency

Performance is not a single quantity. Rather, it can be measured in several different ways. One performance metric is the *overhead*, the added resource cost of implementing an abstraction presented to applications. A related concept is *efficiency*, the lack of overhead in an abstraction. One way to measure overhead (or inversely, efficiency) is the degree to which the abstraction impedes application performance. Suppose you could run the application directly on the underlying hardware without the overhead of the operating system abstraction, how much would that improve the application's performance?

fairness

Operating systems also need to allocate resources among applications, and this can affect the performance of the system as perceived by the end user. One issue is *fairness* between different users or applications running on the same machine. Should resources be divided equally between different users or applications, or should some get preferential treatment? If so, how does the operating system decide what tasks get priority?

response time

throughput

Two related concepts are response time and throughput. *Response time*, sometimes called *delay*, is how long it takes for a single task to run, from the time it starts to the time it completes. For example, a highly visible response time for desktop computers is the time from when the user moves the hardware mouse until the pointer on the screen reflects the user's action. An operating system that provides poor response time can be unusable. *Throughput* is the rate at which the system completes tasks. Throughput is a measure of efficiency for a group of tasks rather than a single one. While it might seem that designs that improve response time would also necessarily improve throughput, this is not the case, as we discuss in Chapter 7.

predictability

A related consideration is performance *predictability*: whether the system's response time or other metric is consistent over time. Predictability can often be more important than average performance. If a user operation sometimes takes an instant but sometimes much longer, the user may find it difficult to adapt. Consider, for example, two systems. In one, each keystroke is usually

instantaneous, but 1% of the time, it takes 10 seconds to take effect. In the other system, a keystroke always takes exactly 0.1 seconds to appear on the screen. Average response time is the same in both systems, but the second is more predictable. Which do you think would be more user friendly?

EXAMPLE To illustrate the concepts of efficiency, overhead, fairness, response time, throughput, and predictability, consider a car driving to its destination. If no other cars or pedestrians were ever on the road, the car could go quite quickly, never needing to slow down for stoplights. Stop signs and stoplights enable multiple cars to share the road, at some cost in overhead and response time for each individual driver. As the system becomes more congested, predictability suffers. Throughput of the system improves with carpooling. With dedicated carpool lanes, carpooling can even reduce delay, despite carpoolers needing to coordinate their pickups. Scrapping the car and building mass transit can improve predictability, throughput, and fairness.

1.2.5 Adoption

In addition to reliability, portability and performance, the success of an operating system depends on two factors outside its immediate control: the wide availability of applications ported to that operating system, and the wide availability of hardware that the operating system can support. An iPhone runs iOS, but without the pre-installed applications and the contents of the App Store, the iPhone would be just another cellphone.

network effect

The *network effect* occurs when the value of some technology depends not only on its intrinsic capabilities, but also on the number of other people who have adopted it. Application and hardware designers spend their efforts on those operating system platforms with the most users, while users favor those operating systems with the best applications or the cheapest hardware. If this sounds circular, it is! More users imply more applications and cheaper hardware; more applications and cheaper hardware imply more users, in a virtuous cycle.

Consider how you might design an operating system to take advantage of the network effect, or at least to avoid being crushed by it. An obvious step would be to design the system to make it easy to accommodate new hardware and for applications to be ported across different versions of the same operating system.

proprietary system

open system

A more subtle issue is the choice of whether the operating system programming interface (API), or the operating system source code itself, is open or proprietary. A *proprietary* system is one under the control of a single company; it can be changed at any time by its provider to meet the needs of its customers. An *open system* is one where the system's source code is public, giving anyone the ability to inspect and change the code. Often, an open system has an API that can be changed only with the agreement of a public standards body. Adherence to standards provides assurance to application developers that

the API will not be changed except by general agreement; on the other hand, standards bodies can make it difficult to quickly add new, desired features.

Neither open nor proprietary systems are intrinsically better for adoption. Windows 8 and MacOS are proprietary operating systems; Linux is an open operating system. All three are widely used. Open systems are easier to adapt to a wide variety of hardware platforms, but they risk devolving into multiple versions, impairing the network effect. Purveyors of proprietary operating systems argue that their systems are more reliable and better adapted to the needs of their customers. Interoperability problems can be reduced if the same company controls both the hardware and the software, but limiting an operating system to one hardware platform impairs the network effect and risks alienating consumers.

Making it easy to port applications from existing systems to a new operating system can help a new system become established; conversely, designing an operating system API that makes it difficult to port applications away from the operating system can help prevent competition from becoming established. Thus, there are often commercial pressures for operating system interfaces to become idiosyncratic. Throughout this book, we discuss operating systems issues at a conceptual level, but remember that the details may vary considerably for any specific operating system due to important, but sometimes chaotic, commercial interests.

1.2.6 Design Tradeoffs

Most practical operating system designs strike a balance between the goals of reliability, security, portability, performance, and adoption. Design choices that improve portability — for example, preserving legacy interfaces — often make the system as a whole less reliable and less secure. Similarly, it is often possible to increase system performance by breaking an abstraction. However, such performance optimizations may add complexity and therefore potentially hurt reliability. The operating system designer must carefully weigh these competing goals.

EXAMPLE To illustrate the tradeoff between performance and complexity, consider the following true story. A research operating system developed in the late 1980's used a type safe language to reduce the incidence of programmer errors. For speed, the most frequently used routines at the core of the operating system were implemented in assembly code. In one of these routines, the implementation team decided to use a sequence of instructions that shaved a single instruction off a very frequently used code path, but that would sometimes break if the operating system exceeded a particular size. At the time, the operating system was nowhere near this limit. After a few years of production use, however, the system started mysteriously crashing, apparently at random, and only after many days of execution. Many weeks of painstaking investigation revealed the problem: the operating system had grown beyond the limit assumed in the assembly code implementation. The fix was easy,

	1981	1997	2014	Factor (2014/1981)
Uniprocessor speed (MIPS)	1	200	2500	2.5 K
CPUs per computer	1	1	10+	10+
Processor MIPS/$	$100K	$25	$0.20	500 K
DRAM Capacity (MiB)/$	0.002	2	1K	500 K
Disk Capacity (GiB)/$	0.003	7	25K	10 M
Home Internet	300 bps	256 Kbps	20 Mbps	100 K
Machine room network	10 Mbps (shared)	100 Mbps (switched)	10 Gbps (switched)	1000+
Ratio of users to computers	100:1	1:1	1:several	100+

Figure 1.8: Approximate computer server performance over time, reflecting widely used servers of each era in 1981, a minicomputer, in 1997, a high-end workstation, in 2014, a rack-mounted multicore server. MIPS stands for "millions of instructions per second," a rough measure of processor performance. The VAX 11/782 was introduced in 1982, it achieved 1 MIP. DRAM prices are from Hennessey and Patterson *Computer Architecture: A Quantitative Approach*. Disk drive prices are from John McCallum. The Hayes smartmodem, introduced in 1981, ran at 300 bps. The 10 Mbps shared Ethernet standard was also introduced in 1981. One of the authors built his first operating system in 1982, used a VAX at his first job, and owned a Hayes to work from home.

once the problem was found, but the question is: do you think the original optimization was worth the risk?

1.3 Operating Systems: Past, Present, and Future

How have operating systems evolved, and what new functionality are we likely to see in future operating systems?

We conclude this chapter by discussing the origins of operating systems, in order to illustrate where these systems are heading in the future. As the lowest layer of software running on top of computer hardware, operating systems date back to the first computers, evolving nearly as rapidly as computer hardware.

1.3.1 Impact of Technology Trends

How has hardware changed?

Moore's Law

The most striking aspect of the last fifty years in computing technology has been the cumulative effect of Moore's Law and the comparable advances in related technologies, such as memory and disk storage. *Moore's Law* states that transistor density increases exponentially over time, similar exponential improvements have occurred in many other component technologies. Figure 1.8 provides an overview of the past three decades of technology improvements in computer hardware. The cost of processing and memory has decreased by almost six orders of magnitude over this period; the cost of disk capacity has

decreased by seven orders of magnitude. Not all technologies have improved at the same rate; disk latency (not shown in the table) has improved, but at a much slower rate than disk capacity. These relative changes have radically altered both the use of computers and the tradeoffs faced by operating system designers.

It is hard to imagine how things used to be. Today, you probably carry a smartphone in your pocket, with an incredibly powerful computer inside. Thousands of server computers wait patiently for you to type in a search query; when the query arrives, they can synthesize a response in a fraction of a second. In the early years of computing, however, the computers were more expensive than the salaries of the people who used them. Users would queue up, often for days, for their turn to run a program. A similar progression from expensive to cheap devices occurred with telephones over the past hundred years. Initially, telephone lines were very expensive, with a single shared line among everyone in a neighborhood. Over time, of course, both computers and telephones have become cheap enough to sit idle until we need them.

Despite these changes, operating systems still face the same conceptual challenges as they did fifty years ago. To manage computer resources for applications and users, they must allocate resources among applications, provide fault isolation and communication services, abstract hardware limitations, and so forth. We have made tremendous progress towards improving the reliability, security, efficiency, and portability of operating systems, but much more is needed. Although we do not know precisely how computing technology or application demand will evolve over the next 10-20 years, it is highly likely that these fundamental operating system challenges will persist.

1.3.2 Early Operating Systems

Computers were expensive, users would wait

The first operating systems were runtime libraries intended to simplify the programming of early computer systems. Rather than the tiny, inexpensive yet massively complex hardware and software systems of today, the first computers often took up an entire floor of a warehouse, cost millions of dollars, and yet were capable of being used only by a single person at a time. The user would first reset the computer, load the program by toggling it into the system one bit at a time, and hit go, producing output to be pored over during the next user's turn. If the program had a bug, the user would need to wait to try the run over again, often the next day.

It might seem like there was no need for an operating system in this setting. However, since computers were enormously expensive, reducing the likelihood of programmer error was paramount. The first operating systems were developed as a way to reduce errors by providing a standard set of common services. For example, early operating systems provided standard input/output (I/O) routines that each user could link into their programs. These services made it more likely that a user's program would produce useful output.

Although these initial operating systems were a huge step forward, the result was still extremely inefficient. It was around this time that the CEO of IBM famously predicted that we would only ever need five computers in the world. If computers today cost millions of dollars and could only run tiny applications by one person at a time, he might have been right.

1.3.3 Multi-User Operating Systems

How do we run multiple programs at the same time?

The next step forward was sharing, introducing many of the advantages, and challenges, that we see in today's operating systems. When processor time is valuable, restricting the system to one user at a time is wasteful. For example, in early systems the processor remained idle while the user loaded the program, even if there was a long line of people waiting their turn.

batch operating system

A *batch operating system* works on a queue of tasks. It runs a simple loop: load, run, and unload each job in turn. While one job was running, the operating system sets up the I/O devices to do background transfers for the next/previous job using a process called *direct memory access* (*DMA*). With DMA, the I/O device transfers its data directly into memory at a location specified by the operating system. When the I/O transfer completes, the hardware interrupts the processor, transferring control to the operating system interrupt handler. The operating system starts the next DMA transfer and then resumes execution of the application. The interrupt appears to the application as if nothing had happened, except for some delay between one instruction and the next.

direct memory access

multitasking

multiprogramming

Batch operating systems were soon extended to run multiple applications at once, called *multitasking* or sometimes *multiprogramming*. Multiple programs are loaded into memory at the same time, each ready to use the processor if for any reason the previous task needed to pause, for example, to read additional input or produce output. Multitasking increases processor efficiency to nearly 100%, if the queue of tasks is long enough, and a sufficient number of I/O devices can keep feeding the processor, there is no need for the processor to wait.

However, processor sharing raises the need for program isolation, to limit a bug in one program from crashing or corrupting another. During this period, computer designers added hardware memory protection, to reduce the overhead of fault isolation.

How do we debug an operating system?

A practical challenge with batch computing, however, is how to debug the operating system itself. Unlike an application program, a batch operating system assumes it is in direct control of the hardware. New versions can only be tested by stopping every application and rebooting the system, essentially turning the computer back into a single-user system. Needless to say, this was an expensive operation, often scheduled for the dead of the night.

host operating system

Virtual machines address this limitation (see Figure 1.4). Instead of running a test operating system directly on the hardware, virtual machines run an *operating system* as an application. The *host operating system*, also called a *virtual machine monitor*, exports an abstract virtual machine (AVM) that is

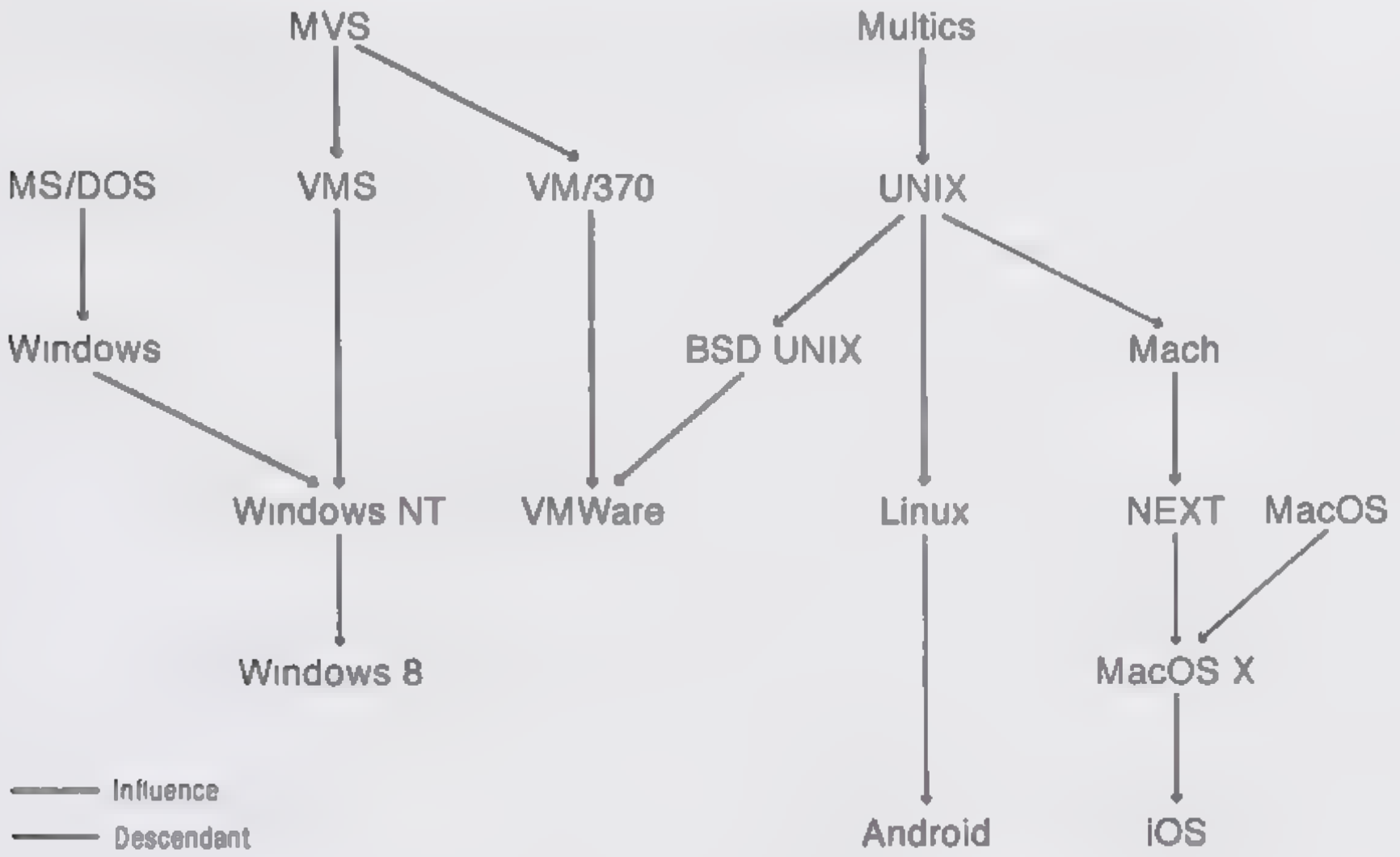

Figure 1.9: Genealogy of several modern operating systems.

identical to the underlying hardware. The test operating system running on top of the virtual machine does not need to know that it is running in a virtual environment — it executes instructions, accesses hardware devices, and restores application state after an interrupt just as it it were running on real hardware.

Virtual machines are now widely used for operating system development, backward compatibility, and cross-platform support. Application software that runs only on an old version of an operating system can share hardware with entirely new applications. The virtual machine monitor runs two virtual machines — one for the new operating system for current applications and a separate one for legacy applications. As another example, MacOS users who need to run Windows or Linux applications can do so by running them inside a virtual machine.

1.3.4 Time-Sharing Operating Systems

How do we make operating systems interactive?

Eventually, the cumulative effect of Moore's Law meant that the cost of computing dropped to where systems could be optimized for users rather than for efficient use of the processor. UNIX, for example, was developed in the early 70's on a spare computer that no one was using at the time. UNIX became the basis for Apple's MacOS X, Linux, VMware (a widely used virtual machine monitor), and Google Android. Figure 1.9 traces the lineage of these operating systems.

time-sharing operating system

Time-sharing operating systems — such as Windows, MacOS, or Linux — are

designed to support interactive use of the computer rather than the batch mode processing of earlier systems. With time-sharing, the user types input on a keyboard or other input device directly connected to the computer. Each keystroke or mouse action causes an interrupt to the processor signaling the event, the interrupt handler reads the event from the device and queues it inside the operating system. When the user's word processor, game, or other application resumes, it fetches the event from the operating system, processes it, and alters the display appropriately before fetching the next event. Hundreds or even thousands of such events can be processed per second, requiring both the operating system and the application to be designed for frequent, very short bursts of activity rather than the sustained execution model of batch processing.

The basic operation of a web server is similar to a time-sharing system. The web server waits for a packet to arrive, to request a web page, web search, or book purchase. The network hardware copies the arriving packet into memory using DMA. Once the transfer is complete, the hardware signals the packet's arrival by interrupting the processor. This triggers the server to perform the requested task. Likewise, the processor is interrupted as each block of a web page is read from disk into memory. Like a time-sharing system, server operating systems must be designed to handle very large numbers of short actions per second.

The earliest time-sharing systems supported many simultaneous users, but even this was just a phase. Eventually, computers became cheap enough that people could afford their own dedicated "personal" computers, which would sit patiently unused for much of the day. Access to shared data became paramount, cementing the shift to client-server computing.

1.3.5 Modern Operating Systems

Today, we have a vast diversity of computing devices, with many different operating systems running on them. The tradeoffs faced by an operating system designer depend on the physical capabilities of the hardware as well as application and user needs. Here are some examples of operating systems that you may have used recently:

- **Desktop, laptop, and netbook operating systems.** Examples include Windows 8, MacOS X, and Linux. These systems are single user, run many applications, and have various I/O devices. One might think that with only one user, there would be no need to design the system to support sharing, and indeed the initial personal computer operating systems took this approach. They had a very limited ability to isolate different parts of the system from each other. Over time, however, it became clear that stricter fault isolation was needed to improve system reliability and resilience against computer viruses. Other key design goals for these systems include adoption (to support a rich set of applications) and interactive performance.

- **Smartphone operating systems.** A smartphone is a cellphone with an embedded computer capable of running third party applications. Examples of smartphone operating systems include iOS, Android, Symbian, WebOS, Blackberry OS and Windows Phone. While smartphones have only one user, they must support many applications. Key design goals include responsiveness, support for a wide variety of applications, and efficient use of the battery. Another design goal is user privacy. Because third party applications might surreptitiously gather private data such as the user's contact list for marketing purposes, the operating system must be designed to limit access to protected user data.

- **Server operating systems.** Search engines, web media, e-commerce sites, and email systems are hosted on computers in data centers, each of these computers runs an operating system, often an industrial strength version of one of the desktop systems described above. Usually, only a single application, such as a web server, runs per machine, but the operating system must coordinate thousands of simultaneous incoming network connections. Throughput in handling a large number of requests per second is a key design goal. At the same time, there is a premium on responsiveness: Amazon and Google both report that adding even 100 milliseconds of delay to each web request can significantly affect revenue. Servers also operate in a hostile environment, where malicious attackers may attempt to subvert or block the service; resistance to attack is an essential requirement.

- **Virtual machines.** As we noted, a virtual machine monitor is an operating system that can run another operating system as if it were an application. Examples include VMWare, Xen, and Windows Virtual PC. Virtual machine monitors face many of the same challenges as other operating systems, with the added challenge posed by coordinating a set of coordinators. A guest operating system running inside a virtual machine makes resource allocation and fault isolation decisions as if it were in complete control of its resources, even though it is sharing the system with other operating systems and applications.

  A commercially important use of virtual machines is to to allow a single server machine to run a set of independent services. Each virtual machine can be configured as needed by that particular service. For example, this allows multiple unrelated web servers to share the same physical hardware. The primary design goal for virtual machines is thus efficiency and low overhead.

- **Embedded systems.** Over time, computers have become cheap enough to integrate into any number of consumer devices, from cable TV set-top boxes, to microwave ovens, the control systems for automobiles and airplanes, LEGO robots, and medical devices, such as MRI machines and WiFi-based intravenous titration systems. Embedded devices typically run a customized operating system bundled with the task-specific

software that controls the device. Although you might think these systems as too simple to merit much attention, software errors in them can have devastating effects. One example is the Therac-25, an early computer controlled radiology device. Programming errors in the operating system code caused the system to malfunction, leading to several patient deaths.

- **Server clusters.** For fault tolerance, scale, and responsiveness, web sites are increasingly implemented on distributed clusters of computers housed in one or more geographically distributed data centers located close to users. If one computer fails due to a hardware fault, software crash, or power failure, another computer can take over its role. If demand for the web site exceeds what a single computer can accommodate, web requests can be partitioned among multiple machines. As with normal operating systems, server cluster applications run on top of an abstract cluster interface to isolate the application from hardware changes and to isolate faults in one application from affecting other applications in the same data center. Likewise, resources can be shared between: (1) various applications on the same web site (such as Google Search, Google Earth, and Gmail), and (2) multiple web sites hosted on the same cluster hardware (such as with Amazon's Elastic Compute Cloud or Google's Compute Engine).

1.3.6 Future Operating Systems

Where are operating systems heading from here over the next decade? Operating systems have become dramatically better at resisting malicious attacks, but they still have quite a ways to go. If security and reliability challenges can be met, huge potential benefits would result from having computers tightly control and coordinate physical infrastructure (such as the power grid, the telephone network, and a hospital's medical devices and medical record systems). Thousands of lives are lost annually through traffic accidents that could potentially be prevented through computer control of automobiles. If we are to rely on computers for these critical systems, we need greater assurance that operating systems are up to the task.

Second, underlying hardware changes will often trigger new work in operating system design. The future of operating systems is also the future of hardware:

- **Very large scale data centers.** Operating systems will need to coordinate the hundreds of thousands or even millions of computers in data centers to support essential online services.
- **Very large scale multicore systems.** Computer architectures already contain several processors per chip, this trend will continue, yielding systems with hundreds or possibly even thousands of processors per machine.

- **Ubiquitous portable computing devices.** With the advent of smartphones, tablets, and e-book readers, computers and their operating systems will become untethered from the keyboard and the screen, responding to voice, gestures, and perhaps even brain waves.

- **Very heterogeneous systems.** As every device becomes programmable, operating systems will be needed for a huge variety of devices, from supercomputers to refrigerators to individual light switches.

- **Very large scale storage.** All data that can be stored, will be; the operating system will need to store enormous amounts of data reliably, so that it can be retrieved at any point, even decades later.

Managing all this is the job of the operating system.

Exercises

1. What is an example of an operating system as:
   a) Referee?
   b) Illusionist?
   c) Glue?

2. What is the difference, if any, between the following terms:
   a) Reliability vs. availability?
   b) Security vs. privacy?
   c) Security enforcement vs. security policy?
   d) Throughput vs. response time?
   e) Efficiency vs. overhead?
   f) Application programming interface (API) vs. abstract virtual machine (AVM)?
   g) Abstract virtual machine (AVM) vs. hardware abstraction layer (HAL)?
   h) Proprietary vs. open operating system?
   i) Batch vs. interactive operating system?
   j) Host vs. guest operating system?

3. Define the term, direct memory access (DMA).

For the following questions, take a moment to speculate. We provide answers to these questions throughout the book, but, given what you know now, how would you answer them? Before there were operating systems, someone needed to develop solutions without being able to look them up! How would you have designed the first operating system?

4. Suppose a computer system and all of its applications were completely bug free. Suppose further that everyone in the world were completely honest and trustworthy. In other words, we need not consider fault isolation.

   a) How should an operating system allocate time on the processor? Should it give the entire processor to each application until it no longer needs it? If there were multiple tasks ready to go at the same time, should it schedule first the task with the least amount of work to do or the one with the most? Justify your answer.

   b) How should the operating system allocate physical memory to applications? What should happen if the set of applications does not fit in memory at the same time?

   c) How should the operating system allocate its disk space? Should the first user to ask acquire all of the free space? What would the likely outcome be for that policy?

5 Now suppose the computer system needs to support fault isolation What hardware and/or operating support do you think would be needed to do the following?

   a) Protect an application's data structures in memory from being corrupted by other applications.

   b) Protecting one user's disk files from being accessed or corrupted by another user.

   c) Protecting the network from a virus trying to use your computer to send spam.

6 How should an operating system support communication between applications? Explain your reasoning.

   a) Through the file system?

   b) Through messages passed between applications?

   c) Through regions of memory shared between the applications?

   d) All of the above?

   e) None of the above?

7. How would you design combined hardware and software support to provide the illusion of a nearly infinite virtual memory on a limited amount of physical memory?

8. How would you design a system to run an entire operating system as an application on top of another operating system?

9. How would you design a system to update complex data structures on disk in a consistent fashion despite machine crashes?

10. Society itself must grapple with managing resources. What ways do governments use to allocate resources, isolate misuse, and foster sharing in real life?

11. Suppose you were tasked with designing and implementing an ultra-reliable and ultra-available operating system. What techniques would you use? What tests, if any, might be sufficient to convince you of the system's reliability, short of handing your operating system to millions of users to serve as beta testers?

12. MTTR, and therefore availability, can be improved by reducing the time to reboot a system after a failure. What techniques might you use to speed up booting? Would your techniques always work after a failure?

13. For the computer you are currently using, how should the operating system designers prioritize among reliability, security, portability, performance, and adoption? Explain why.