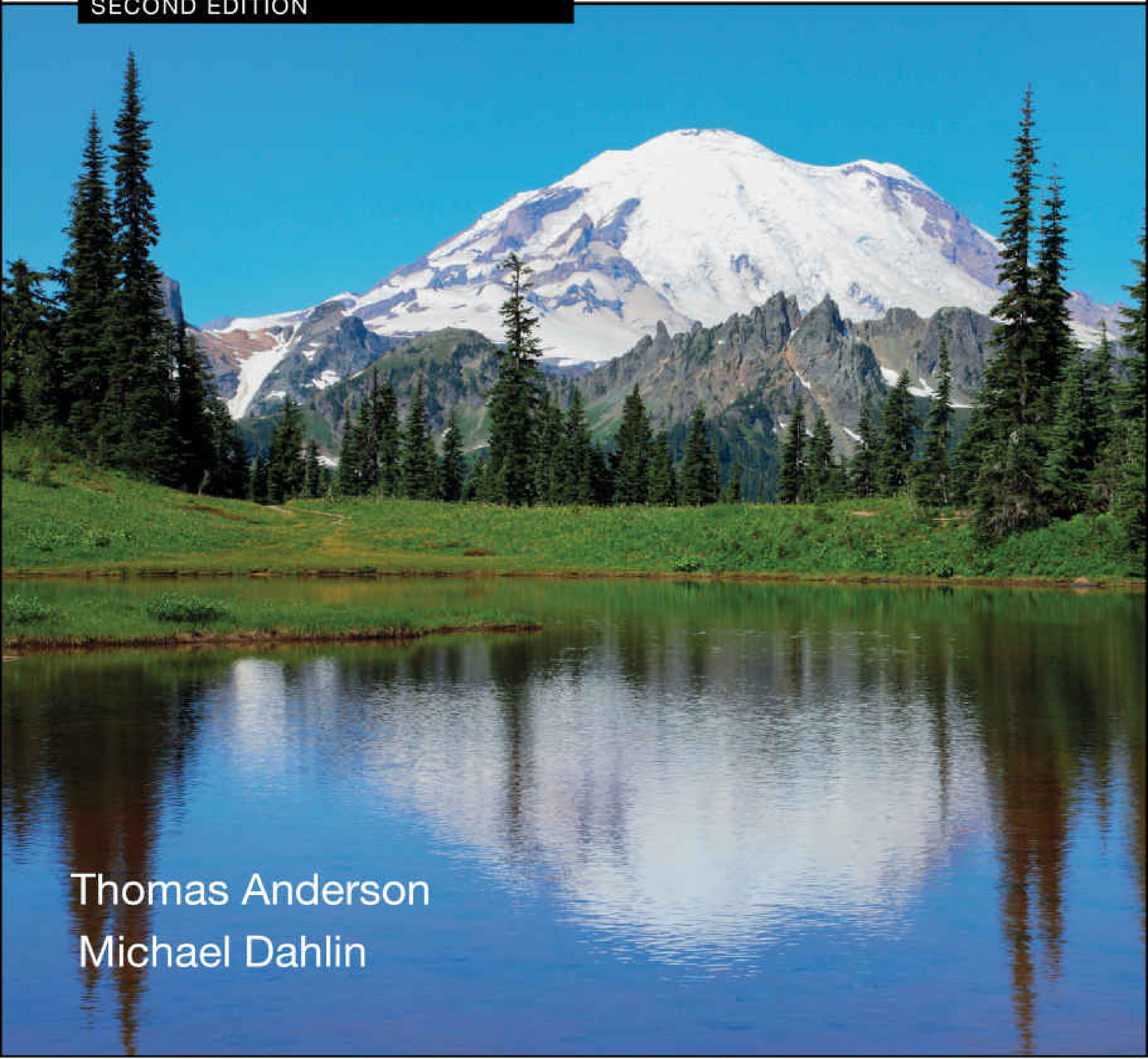
Operating Systems

Principles & Practice

Volume I: Kernels and Processes

SECOND EDITION



Thomas Anderson

Michael Dahlin

Operating Systems
Principles & Practice
Volume I: Kernels and Processes

Second Edition

Thomas Anderson
University of Washington

Mike Dahlin
University of Texas and Google

Recursive Books
recursivebooks.com

Operating Systems: Principles and Practice (Second Edition) Volume I: Kernels and Processes by Thomas Anderson and Michael Dahlin
Copyright ©Thomas Anderson and Michael Dahlin, 2011-2015.

ISBN 978-0-9856735-3-6
Publisher: Recursive Books, Ltd., http://recursivebooks.com/
Cover: Reflection Lake, Mt. Rainier
Cover design: Cameron Neat
Illustrations: Cameron Neat
Copy editors: Sandy Kaplan, Whitney Schmidt
Ebook design: Robin Briggs
Web design: Adam Anderson

SUGGESTIONS, COMMENTS, and ERRORS. We welcome suggestions, comments and error reports, by email to suggestions@recursivebooks.com

To Robin, Sandra, Katya, and Adam
Tom Anderson

To Marla, Kelly, and Keith
Mike Dahlin

Contents

Preface

Preface to the eBook Edition

Operating Systems: Principles and Practice is a textbook for a first course in undergraduate operating systems. In use at over 50 colleges and universities worldwide, this textbook provides:

- A path for students to understand high level concepts all the way down to working code.
- Extensive worked examples integrated throughout the text provide students concrete guidance for completing homework assignments.
- A focus on up-to-date industry technologies and practice

The eBook edition is split into four volumes that together contain exactly the same material as the (2nd) print edition of Operating Systems: Principles and Practice, reformatted for various screen sizes. Each volume is self-contained and can be used as a standalone text, e.g., at schools that teach operating systems topics across multiple courses.

- **Volume 1: Kernels and Processes.** This volume contains Chapters 1-3 of the print edition. We describe the essential steps needed to isolate programs to prevent buggy applications and computer viruses from crashing or taking control of your system.
- **Volume 2: Concurrency.** This volume contains Chapters 4-7 of the print edition. We provide a concrete methodology for writing correct concurrent programs that is in widespread use in industry, and we explain the mechanisms for context switching and synchronization from fundamental concepts down to assembly code.
- **Volume 3: Memory Management.** This volume contains Chapters 8-10 of the print edition. We explain both the theory and mechanisms behind 64-bit address space translation, demand paging, and virtual machines.
- **Volume 4: Persistent Storage.** This volume contains Chapters 11-14 of the print edition. We explain the technologies underlying modern extent-based, journaling, and versioning file systems.

A more detailed description of each chapter is given in the preface to the print edition.

Preface to the Print Edition

Why We Wrote This Book

Many of our students tell us that operating systems was the best course they took as an undergraduate and also the most important for their careers. We are not alone — many of our colleagues report receiving similar feedback from their students.

Part of the excitement is that the core ideas in a modern operating system — protection, concurrency, virtualization, resource allocation, and reliable storage — have become widely applied throughout computer science, not just operating system kernels. Whether you get a job at Facebook, Google, Microsoft, or any other leading-edge technology company, it is impossible to build resilient, secure, and flexible computer systems without the ability to apply operating systems concepts in a variety of settings. In a modern world, nearly everything a user does is distributed, nearly every computer is multi-core, security threats abound, and many applications such as web browsers have become mini-operating systems in their own right.

It should be no surprise that for many computer science students, an undergraduate operating systems class has become a *de facto* requirement: a ticket to an internship and eventually to a full-time position.

Unfortunately, many operating systems textbooks are still stuck in the past, failing to keep pace with rapid technological change. Several widely-used books were initially written in the mid-1980's, and they often act as if technology stopped at that point. Even when new topics are added, they are treated as an afterthought, without pruning material that has become less important. The result are textbooks that are very long, very expensive, and yet fail to provide students more than a superficial understanding of the material.

Our view is that operating systems have changed dramatically over the past twenty years, and that justifies a fresh look at both *how* the material is taught and *what* is taught. The pace of innovation in operating systems has, if anything, increased over the past few years, with the introduction of the iOS and Android operating systems for smartphones, the shift to multicore computers, and the advent of cloud computing.

To prepare students for this new world, we believe students need three things to succeed at understanding operating systems at a deep level:

- **Concepts and code.** We believe it is important to teach students both *principles* and *practice*, concepts and implementation, rather than either alone. This textbook takes concepts all the way down to the level of working code, e.g., how a context switch works in assembly code. In our experience, this is the only way students will really understand and master the material. All of the code in this book is available from the author's web site, ospp.washington.edu.

- **Extensive worked examples.** In our view, students need to be able to apply concepts in practice. To that end, we have integrated a large number of example exercises, along with solutions, throughout the text. We uses these exercises extensively in our own lectures, and we have found them essential to challenging students to go beyond a superficial understanding.

- **Industry practice.** To show students how to apply operating systems concepts in a variety of settings, we use detailed, concrete examples from Facebook, Google, Microsoft, Apple, and other leading-edge technology companies throughout the textbook. Because operating systems concepts are important in a wide range of computer systems, we take these examples not only from traditional operating systems like Linux, Windows, and OS X but also from other systems that need to solve problems of protection, concurrency, virtualization, resource allocation, and reliable storage like databases, web browsers, web servers, mobile applications, and search engines.

Taking a fresh perspective on what students need to know to apply operating systems concepts in practice has led us to innovate in every major topic covered in an undergraduate-level course:

- **Kernels and Processes.** The safe execution of untrusted code has become central to many types of computer systems, from web browsers to virtual machines to operating systems. Yet existing textbooks treat protection as a side effect of UNIX processes, as if they are synonyms. Instead, we start from first principles: what are the minimum requirements for process isolation, how can systems implement process isolation efficiently, and what do students need to know to implement functions correctly when the caller is potentially malicious?

- **Concurrency.** With the advent of multi-core architectures, most students today will spend much of their careers writing concurrent code. Existing textbooks provide a blizzard of concurrency alternatives, most of which were abandoned decades ago as impractical. Instead, we focus on providing students a *single* methodology based on Mesa monitors that will enable students to write correct concurrent programs — a methodology that is by far the dominant approach used in industry.

- **Memory Management.** Even as demand-paging has become less important, virtualization has become even more important to modern computer systems. We provide a deep treatment of address translation hardware, sparse address spaces, TLBs, and on-chip caches. We then use those concepts as a springboard for describing virtual machines and related concepts such as checkpointing and copy-on-write.

- **Persistent Storage.** Reliable storage in the presence of failures is central to the design of most computer systems. Existing textbooks survey the history of file systems, spending most of their time ad hoc approaches to failure recovery and de-fragmentation. Yet no modern file systems still use those ad hoc approaches. Instead, our focus is on how file systems use extents, journaling, copy-on-write, and RAID to achieve both high performance and high reliability.

Intended Audience

Operating Systems: Principles and Practice is a textbook for a first course in undergraduate operating systems. We believe operating systems should be taken as early as possible in an undergraduate's course of study; many students use the course as a springboard to an internship and a career. To that end, we have designed the textbook to assume minimal pre-requisites: specifically, students should have taken a data structures course and one on computer organization. The code examples are written in a combination of x86 assembly, C, and C++. In particular, we have designed the book to interface well with the Bryant and O'Halloran textbook. We review and cover in much more depth the material from the second half of that book.

We should note what this textbook is *not*: it is not intended to teach the API or internals of any specific operating system, such as Linux, Android, Windows 8, OS X, or iOS. We use many concrete examples from these systems, but our focus is on the shared problems these systems face and the technologies these systems use to solve those problems.

A Guide to Instructors

One of our goals is enable instructors to choose an appropriate level of depth for each course topic. Each chapter begins at a conceptual level, with implementation details and the more advanced material towards the end. The more advanced material can be omitted without compromising the ability of students to follow later material. No single-quarter or single-semester course is likely to be able to cover every topic we have included, but we think it is a good thing for students to come away from an operating systems course with an appreciation that there is *always* more to learn.

For each topic, we attempt to convey it at three levels:

- **How to reason about systems.** We describe core systems concepts, such as protection, concurrency, resource scheduling, virtualization, and storage, and we provide practice applying these concepts in various situations. In our view, this provides the biggest long-term payoff to students, as they are likely to need to apply these concepts in their work throughout their career, almost regardless of what project they end up working on.

- **Power tools.** We introduce students to a number of abstractions that they can apply in their work in industry immediately after graduation, and that we expect will continue to be useful for decades such as sandboxing, protected procedure calls, threads, locks, condition variables, caching, checkpointing, and transactions.

- **Details of specific operating systems.** We include numerous examples of how different operating systems work in practice. However, this material changes rapidly, and there is an order of magnitude more material than can be covered in a single semester-length course. The purpose of these examples is to illustrate how to use the operating systems principles and power tools to solve concrete problems. We do not attempt to provide a comprehensive description of Linux, OS X, or any other particular operating system.

The book is divided into five parts: an introduction (Chapter 1), kernels and processes (Chapters 2-3), concurrency, synchronization, and scheduling (Chapters 4-7), memory management (Chapters 8-10), and persistent storage (Chapters 11-14).

- **Introduction.** The goal of Chapter 1 is to introduce the recurring themes found in the later chapters. We define some common terms, and we provide a bit of the history of the development of operating systems.

- **The Kernel Abstraction.** Chapter 2 covers kernel-based process protection — the concept and implementation of executing a user program with restricted privileges. Given the increasing importance of computer security issues, we believe protected execution and safe transfer across privilege levels are worth treating in depth. We have broken the description into sections, to allow instructors to choose either a quick introduction to the concepts (up through Section 2.3), or a full treatment of the kernel implementation details down to the level of interrupt handlers. Some instructors start with concurrency, and cover kernels and kernel protection afterwards. While our textbook can be used that way, we have found that students benefit from a basic understanding of the role of operating systems in executing user programs, before introducing concurrency.

- **The Programming Interface.** Chapter 3 is intended as an impedance match for students of differing backgrounds. Depending on student background, it can be skipped or covered in depth. The chapter covers the operating system from a programmer's perspective: process creation and management, device-independent input/output, interprocess communication, and network sockets. Our goal is that students should understand at a detailed level what happens when a user clicks a link in a web browser, as the request is transferred through operating system kernels and user space processes at the client, server, and back again. This chapter also covers the organization of the operating system itself: how device drivers and the hardware abstraction layer work in a modern operating system; the difference between a monolithic and a microkernel operating system; and how policy and mechanism are separated in modern operating systems.

- **Concurrency and Threads.** Chapter 4 motivates and explains the concept of threads. Because of the increasing importance of concurrent programming, and its integration with modern programming languages like Java, many students have been introduced to multi-threaded programming in an earlier class. This is a bit dangerous, as students at this stage are prone to writing programs with race conditions, problems that may or may not be discovered with testing. Thus, the goal of this chapter is to provide a solid conceptual framework for understanding the semantics of concurrency, as well as how concurrent threads are implemented in both the operating system kernel and in user-level libraries. Instructors needing to go more quickly can omit these implementation details.

- **Synchronization.** Chapter 5 discusses the synchronization of multi-threaded programs, a central part of all operating systems and increasingly important in many other contexts. Our approach is to describe one effective method for structuring concurrent programs (based on Mesa monitors), rather than to attempt to cover several different approaches. In our view, it is more important for students to master one methodology. Monitors are a particularly robust and simple one, capable of implementing most concurrent programs efficiently. The implementation of synchronization primitives should be included if there is time, so students see that there is no magic.

- **Multi-Object Synchronization.** Chapter 6 discusses advanced topics in concurrency — specifically, the twin challenges of multiprocessor lock contention and deadlock. This material is increasingly important for students working on multicore systems, but some courses may not have time to cover it in detail.

- **Scheduling.** This chapter covers the concepts of resource allocation in the specific context of processor scheduling. With the advent of data center computing and multicore architectures, the principles and practice of resource allocation have renewed importance. After a quick tour through the tradeoffs between response time and throughput for uniprocessor scheduling, the chapter covers a set of more advanced topics in affinity and multiprocessor scheduling, power-aware and deadline scheduling, as well as basic queueing theory and overload management. We conclude these topics by walking students through a case study of server-side load management.

- **Address Translation.** Chapter 8 explains mechanisms for hardware and software address translation. The first part of the chapter covers how hardware and operating systems cooperate to provide flexible, sparse address spaces through multi-level segmentation and paging. We then describe how to make memory management efficient with translation lookaside buffers (TLBs) and virtually addressed caches. We consider how to keep TLBs consistent when the operating system makes changes to its page tables. We conclude with a discussion of modern software-based protection mechanisms such as those found in the Microsoft Common Language Runtime and Google's Native Client.

- **Caching and Virtual Memory.** Caches are central to many different types of computer systems. Most students will have seen the concept of a cache in an earlier class on machine structures. Thus, our goal is to cover the theory and implementation of caches: when they work and when they do not, as well as how they are implemented in hardware and software. We then show how these ideas are applied in the context of memory-mapped files and demand-paged virtual memory.

- **Advanced Memory Management.** Address translation is a powerful tool in system design, and we show how it can be used for zero copy I/O, virtual machines, process checkpointing, and recoverable virtual memory. As this is more advanced material, it can be skipped by those classes pressed for time.

- **File Systems: Introduction and Overview.** Chapter 11 frames the file system portion of the book, starting top down with the challenges of providing a useful file abstraction to users. We then discuss the UNIX file system interface, the major internal elements inside a file system, and how disk device drivers are structured.

- **Storage Devices.** Chapter 12 surveys block storage hardware, specifically magnetic disks and flash memory. The last two decades have seen rapid change in storage technology affecting both application programmers and operating systems designers; this chapter provides a snapshot for students, as a building block for the next two chapters. If students have previously seen this material, this chapter can be skipped.

- **Files and Directories.** Chapter 13 discusses file system layout on disk. Rather than survey all possible file layouts — something that changes rapidly over time — we use file systems as a concrete example of mapping complex data structures onto block storage devices.

- **Reliable Storage.** Chapter 14 explains the concept and implementation of reliable storage, using file systems as a concrete example. Starting with the ad hoc techniques used in early file systems, the chapter explains checkpointing and write ahead logging as alternate implementation strategies for building reliable storage, and it discusses how redundancy such as checksums and replication are used to improve reliability and availability.

We welcome and encourage suggestions for how to improve the presentation of the material; please send any comments to the publisher's website, suggestions@recursivebooks.com.

Acknowledgements

We have been incredibly fortunate to have the help of a large number of people in the conception, writing, editing, and production of this book.

We started on the journey of writing this book over dinner at the USENIX NSDI conference in 2010. At the time, we thought perhaps it would take us the summer to complete the first version and perhaps a year before we could declare ourselves done. We were very wrong! It is no exaggeration to say that it would have taken us a lot longer without the help we have received from the people we mention below.

Perhaps most important have been our early adopters, who have given us enormously useful feedback as we have put together this edition:

Carnegie-Mellon	David Eckhardt and Garth Gibson
Clarkson	Jeanna Matthews
Cornell	Gun Sirer
ETH Zurich	Mothy Roscoe
New York University	Laskshmi Subramanian
Princeton University	Kai Li
Saarland University	Peter Druschel
Stanford University	John Ousterhout
University of California Riverside	Harsha Madhyastha
University of California Santa Barbara	Ben Zhao
University of Maryland	Neil Spring
University of Michigan	Pete Chen
University of Southern California	Ramesh Govindan
University of Texas-Austin	Lorenzo Alvisi
Universtiy of Toronto	Ding Yuan
University of Washington	Gary Kimura and Ed Lazowska

In developing our approach to teaching operating systems, both before we started writing and afterwards as we tried to put our thoughts to paper, we made extensive use of lecture notes and slides developed by other faculty. Of particular help were the materials created by Pete Chen, Peter Druschel, Steve Gribble, Eddie Kohler, John Ousterhout, Mothy Roscoe, and Geoff Voelker. We thank them all.

Our illustrator for the second edition, Cameron Neat, has been a joy to work with.

We are also grateful to Lorenzo Alvisi, Adam Anderson, Pete Chen, Steve Gribble, Sam Hopkins, Ed Lazowska, Harsha Madhyastha, John Ousterhout, Mark Rich, Mothy Roscoe, Will Scott, Gun Sirer, Ion Stoica, Lakshmi Subramanian, and John Zahorjan for their helpful comments and suggestions as to how to improve the book.

We thank Josh Berlin, Marla Dahlin, Sandy Kaplan, John Ousterhout, Whitney Schmidt, and Mike Walfish for helping us identify and correct grammatical or technical bugs in the text.

We thank Jeff Dean, Garth Gibson, Mark Oskin, Simon Peter, Dave Probert, Amin Vahdat, and Mark Zbikowski for their help in explaining the internal workings of some of the commercial systems mentioned in this book.

We would like to thank Dave Wetherall, Dan Weld, Mike Walfish, Dave Patterson, Olav Kvern, Dan Halperin, Armando Fox, Robin Briggs, Katya Anderson, Sandra Anderson, Lorenzo Alvisi, and William Adams for their help and advice on textbook economics and production.

The Helen Riaboff Whiteley Center as well as Don and Jeanne Dahlin were kind enough to lend us a place to escape when we needed to get chapters written.

Finally, we thank our families, our colleagues, and our students for supporting us in this larger-than-expected effort.

I
Kernels and Processes

1. Introduction

All I really need to know I learned in kindergarten. —*Robert Fulgham*

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.

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 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.

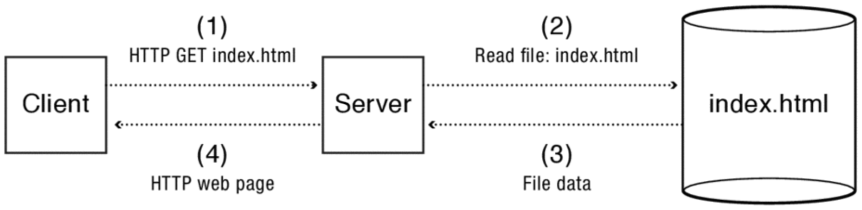


Figure 1.1: The operation of a web server. The client machine sends an HTTP GET request to the web server. The server decodes the packet, reads the file, and sends the contents back to the client.

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?

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.

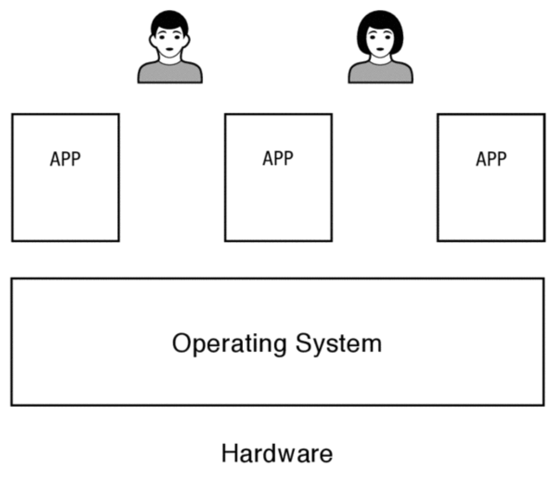


Figure 1.2: A general-purpose operating system is a layer of software that manages a computer's resources for its users and applications.

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

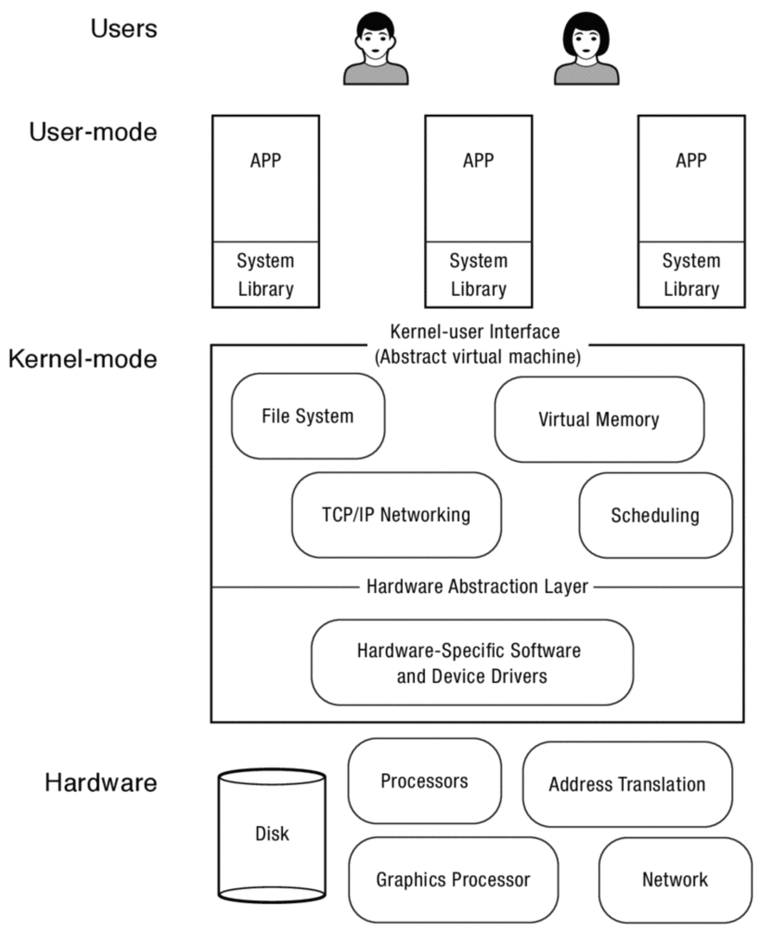


Figure 1.3: This shows the structure of a general-purpose operating system, as an expansion on the simple view presented in Figure 1.2. At the lowest level, the hardware provides processors, memory, and a set of devices for storing data and communicating with the outside world. The hardware also provides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

---

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 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

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.

- **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

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 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.

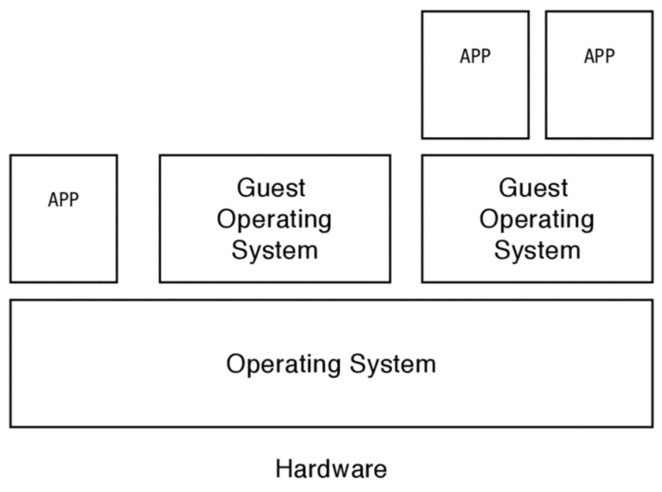


Figure 1.4: A guest operating system running inside a virtual machine.

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.

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 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

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

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:

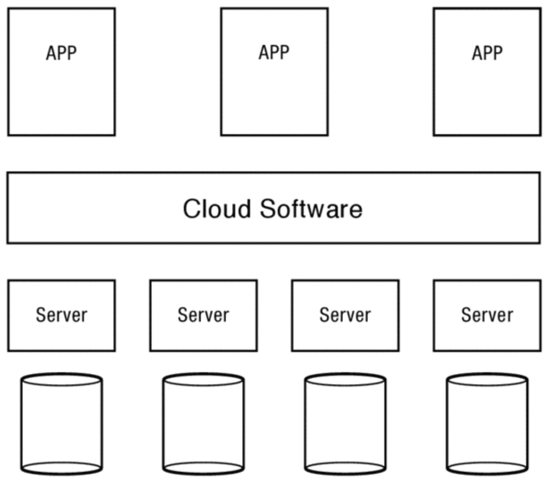


Figure 1.5: Cloud computing software provides a convenient abstraction of server resources to cloud applications.

- **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?
    - **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?

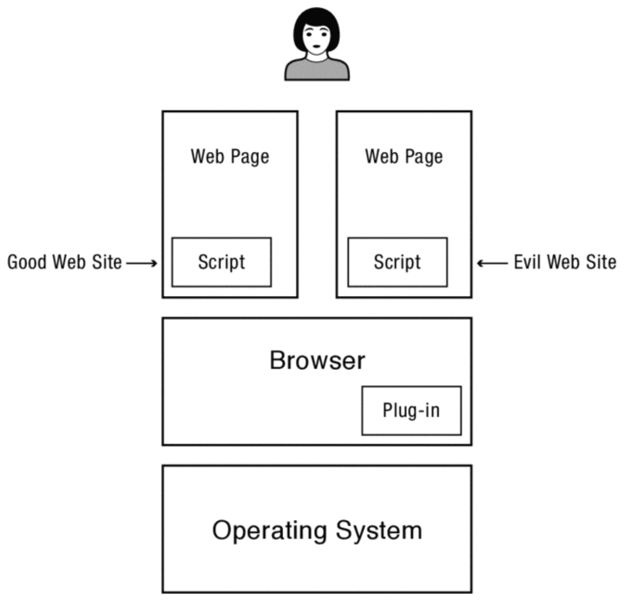


Figure 1.6: A web browser isolates scripts and plug-ins from accessing privileged resources on the host operating system.

- **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 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?
    - **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?

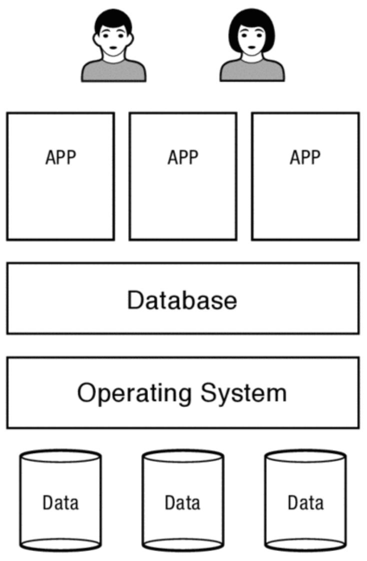


Figure 1.7: Databases perform many of the tasks of an operating system: they allocate resources among user queries to ensure responsiveness, they mask differences in the underlying operating system and hardware, and they provide a convenient programming abstraction to developers.

- **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

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

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.

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.

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

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.

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.

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

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.

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.

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.

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?

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?

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.

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.

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.

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, 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

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

	1981	**1997**	**2014**	**Factor (2014 / 1981)**
Single processor speed (MIPS)	1	200	2500	2.5 K
CPUs per computer	1	1	10+	10+
Processor $ / MIP	$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.

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

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

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.

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.

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.

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.

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 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 if 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

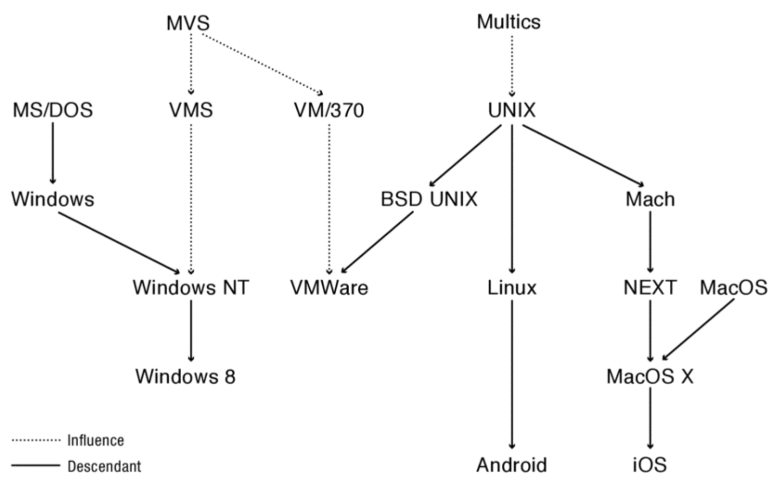


Figure 1.9: Genealogy of several modern operating systems.

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 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. Provided 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.

2. The Kernel Abstraction

Good fences make good neighbors. —*17th century proverb*

A central role of operating systems is *protection* — the isolation of potentially misbehaving applications and users so that they do not corrupt other applications or the operating system itself. Protection is essential to achieving several of the operating systems goals noted in the previous chapter:

- **Reliability.** Protection prevents bugs in one program from causing crashes in other programs or in the operating system. To the user, a system crash appears to be the operating system's fault, even if the root cause of the problem is some unexpected behavior by an application or user. Thus, for high system reliability, an operating system must bullet proof itself to operate correctly regardless of what an application or user might do.

- **Security.** Some users or applications on a system may be less than completely trustworthy; therefore, the operating system must limit the scope of what they can do. Without protection, a malicious user might surreptitiously change application files or even the operating system itself, leaving the user none the wiser. For example, if a malicious application can write directly to the disk, it could modify the file containing the operating system's code; the next time the system starts, the modified operating system would boot instead, installing spyware and disabling virus protection. For security, an operating system must prevent untrusted code from modifying system state.

- **Privacy.** On a multi-user system, each user must be limited to only the data that she is permitted to access. Without protection provided by the operating system, any user or application running on a system could access anyone's data, without the knowledge or approval of the data's owner. For example, hackers often use popular applications — such as games or screen savers — as a way to gain access to personal email, telephone numbers, and credit card data stored on a smartphone or laptop. For privacy, an operating system must prevent untrusted code from accessing unauthorized data.

- **Fair resource allocation.** Protection is also needed for effective resource allocation. Without protection, an application could gather any amount of processing time, memory, or disk space that it wants. On a single-user system, a buggy application could prevent other applications from running or make them run so slowly that they appear to stall. On a multi-user system, one user could grab all of the system's resources. Thus, for efficiency and fairness, an operating system must be able to limit the amount of resources assigned to each application or user.

Implementing protection is the job of the *operating system kernel*. The kernel, the lowest level of software running on the system, has full access to all of the machine hardware. The kernel is necessarily *trusted* to do anything with the hardware. Everything else — that is, the untrusted software running on the system — is run in a restricted environment with less than complete access to the full power of the hardware. Figure 2.1 illustrates this difference between kernel-level and user-level execution.

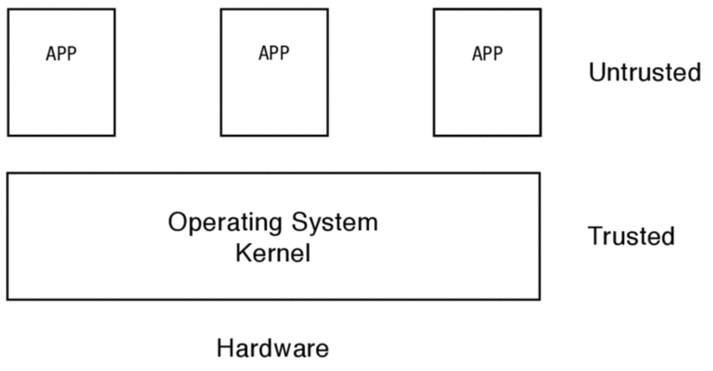


Figure 2.1: User-level and kernel-level operation. The operating system kernel is trusted to arbitrate between untrusted applications and users.

In turn, applications themselves often need to safely execute untrusted third party code. An example is a web browser executing embedded Javascript to draw a web page. Without protection, a script with an embedded virus can take control of the browser, making users think they are interacting directly with the web when in fact their web passwords are being forwarded to an attacker.

This design pattern — extensible applications running third-party scripts — occurs in many different domains. Applications become more powerful and widely used if third party developers and users can customize them, but doing so raises the issue of how to protect the application itself from rogue extensions. This chapter focuses on how the operating system protects the kernel from untrusted applications, but the principles also apply at the application level.

A *process* is the execution of an application program with restricted rights; the process is the abstraction for protected execution provided by the operating system kernel. A process needs permission from the operating system kernel before accessing the memory of any other process, before reading or writing to the disk, before changing hardware settings, and so forth. In other words, the operating system kernel mediates and checks each process's access to hardware. This chapter explains the process concept and how the kernel implements process isolation.

A key consideration is the need to provide protection while still running application code at high speed. The operating system kernel runs directly on the processor with unlimited rights. The kernel can perform any operation available on the hardware. What about applications? They need to run on the processor with all potentially dangerous operations disabled. To make this work, hardware needs to provide a bit of assistance, which we will describe shortly. Throughout the book, there are similar examples of how small amounts of carefully designed hardware can help make it much easier for the operating system to provide what users want.

Of course, both the operating system kernel and application processes running with restricted rights are in fact sharing the same machine — the same processor, the same memory, and the same disk. When reading this chapter, keep these two perspectives in mind: when we are running the operating system kernel, it can do anything; when we are running an application process on behalf of a user, the process's behavior is restricted.

Thus, a processor running an operating system is somewhat akin to someone with a split personality. When running the operating system kernel, the processor is like a warden in charge of an insane asylum with complete access to everything. At other times, the processor runs application code in a process — the processor becomes an inmate, wearing a straightjacket locked in a padded cell by the warden, protected from harming anyone else. Of course, it is the same processor in both cases, sometimes completely trustworthy and at other times completely untrusted.

Chapter roadmap: Protection raises several important questions that we will answer in the rest of the chapter:

- **The Process Abstraction.** What is a process and how does it differ from a program? (Section 2.1)
- **Dual-Mode Operation.** What hardware enables the operating system to efficiently implement the process abstraction? (Section 2.2)
- **Types of Mode Transfer.** What causes the processor to switch control from a user-level program to the kernel? (Section 2.3)
- **Implementing Safe Mode Transfer.** How do we safely switch between user level and the kernel? (Section 2.4)
- **Putting It All Together: x86 Mode Transfer.** What happens on an x86 mode switch? (Section 2.5)
- **Implementing Secure System Calls.** How do library code and the kernel work together to implement protected procedure calls from the application into the kernel? (Section 2.6)
- **Starting a New Process.** How does the operating system kernel start a new process? (Section 2.7)
- **Implementing Upcalls.** How does the operating system kernel deliver an asynchronous event to a user process? (Section 2.8)
- **Case Study: Booting an OS Kernel.** What steps are needed to start running an operating system kernel, to the point where it can create a process? (Section 2.9)

- **Case Study: Virtual Machines.** Can an operating system run inside a process? (Section 2.10)

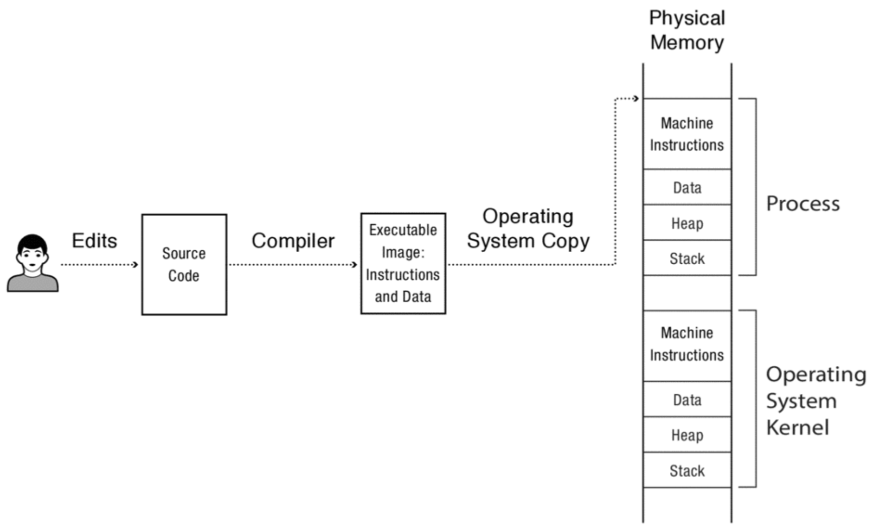


Figure 2.2: A user edits, compiles, and runs a user program. Other programs can also be stored in physical memory, including the operating system itself.

2.1 The Process Abstraction

In the model you are likely familiar with, illustrated in Figure 2.2, a programmer types code in some high-level language. A compiler converts that code into a sequence of machine instructions and stores those instructions in a file, called the program's *executable image*. The compiler also defines any static data the program needs, along with its initial values, and includes them in the executable image.

To run the program, the operating system copies the instructions and data from the executable image into physical memory. The operating system sets aside a memory region, the *execution stack*, to hold the state of local variables during procedure calls. The operating system also sets aside a memory region, called the *heap*, for any dynamically allocated data structures the program might need. Of course, to copy the program into memory, the operating system itself must already be loaded into memory, with its own stack and heap.

Ignoring protection, once a program is loaded into memory, the operating system can start it running by setting the stack pointer and jumping to the program's first instruction. The compiler itself is just another program: the operating system starts the compiler by copying its executable image into memory and jumping to its first instruction.

To run multiple copies of the same program, the operating system can make multiple copies of the program's instructions, static data, heap, and stack in memory. As we describe in Chapter 8, most operating systems reuse memory wherever possible: they store only a single copy of a program’s instructions when multiple copies of the program are executed at the same time. Even so, a separate copy of the program’s data, heap, and stack are needed. For now, we will keep things simple and assume the operating system makes a separate copy of the entire program for each process.

Thus, a process is an *instance* of a program, in much the same way that an object is an instance of a class in object-oriented programming. Each program can have zero, one or more processes executing it. For each instance of a program, there is a process with its own copy of the program in memory.

The operating system keeps track of the various processes on the computer using a data structure called the *process control block*, or PCB. The PCB stores all the information the operating system needs about a particular process: where it is stored in memory, where its executable image resides on disk, which user asked it to execute, what privileges the process has, and so forth.

Earlier, we defined a process as an instance of a program executing with restricted rights. Each of these roles — execution and protection — is important enough to merit several chapters.

This chapter focuses on protection, and so we limit our discussion to simple processes, each with one program counter, code, data, heap, and stack.

Some programs consist of multiple concurrent activities, or threads. A web browser, for example, might need to receive user input at the same time it is drawing the screen or receiving network input. Each of these separate activities has its own program counter and stack but operates on the same code and data as the other threads. The operating system runs multiple threads in a process, in much the same way that it runs multiple processes in physical memory. We generalize on the process abstraction to allow multiple activities in the same protection domain in Chapter 4.

Processes, lightweight processes, and threads

The word “process", like many terms in computer science, has evolved over time. The evolution of words can sometimes trip up the unwary — systems built at different times will use the same word in significantly different ways.

A “process" was originally coined to mean what is now called a “thread" — a logical sequence of instructions that executes either operating system or application code. The concept of a process was developed as a way of simplifying the correct construction of early operating systems that provided no protection between application programs.

Organizing the operating system as a cooperating set of processes proved immensely successful, and soon almost every new operating system was built this way, including systems that also provided protection against malicious or buggy user programs. At the time, almost all user programs were simple, single-threaded programs with only one program counter and one stack, so there was no confusion. A process was needed to run a program, that is, a single sequential execution stream with a protection boundary.

As parallel computers became more popular, though, we once again needed a word for a logical sequence of instructions. A multiprocessor program can have multiple instruction sequences running in parallel, each with its own program counter, but all cooperating within a single protection boundary. For a time, these were called “lightweight processes" (each a sequence of instructions cooperating inside a protection boundary), but eventually the word “thread” became more widely used.

This leads to the current naming convention used in almost all modern operating systems: a process executes a program, consisting of one or more threads running inside a protection boundary.

2.2 Dual-Mode Operation

Once a program is loaded into memory and the operating system starts the process, the processor fetches each instruction in turn, then decodes and executes it. Some instructions compute values, say, by multiplying two registers and putting the result into another register. Some instructions read or write locations in memory. Still other instructions, like branches or procedure calls, change the program counter and thus determine the next instruction to execute. Figure 2.3 illustrates the basic operation of a processor.

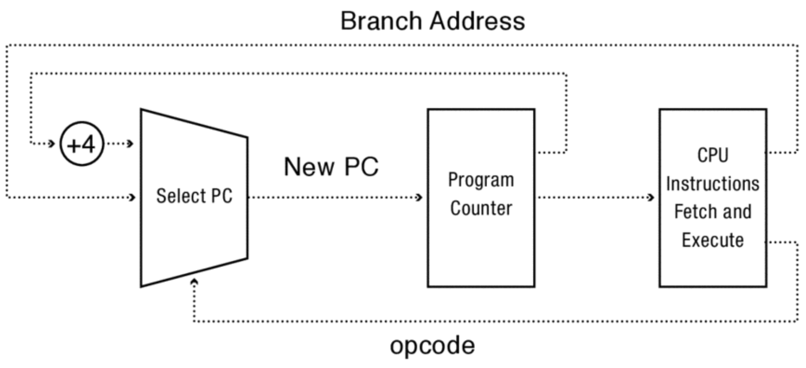


Figure 2.3: The basic operation of a CPU. Opcode, short for operation code, is the decoded instruction to be executed, e.g., branch, memory load, or arithmetic operation.

How does the operating system kernel prevent a process from harming other processes or the operating system itself? After all, when multiple programs are loaded into memory at the same time, what prevents a process from overwriting another process’s data structures, or even overwriting the operating system image stored on disk?

If we step back from any consideration of performance, a very simple, safe, and entirely hypothetical approach would be to have the operating system kernel simulate, step by step, every instruction in every user process. Instead of the processor directly executing instructions, a software interpreter would fetch, decode, and execute each user program instruction in turn. Before executing each instruction, the interpreter could check if the process had permission to do the operation in question: is it referencing part of its own memory, or someone else’s? Is it trying to branch into someone else’s code? Is it directly accessing the disk, or is it using the correct routines in the operating system to do so? The interpreter could allow all legal operations while halting any application that overstepped its bounds.

Now suppose we want to speed up our hypothetical simulator. Most instructions are perfectly safe, such as adding two registers together and storing the result in a third register. Can we modify the processor in some way to allow safe instructions to execute directly on the hardware?

To accomplish this, we implement the same checks as in our hypothetical interpreter, but in hardware rather than software. This is called *dual-mode operation*, represented by a single bit in the processor status register that signifies the current mode of the processor. In *user mode*, the processor checks each instruction before executing it to verify that it is permitted to be performed by that process. (We describe the specific checks next.) In *kernel mode*, the operating system executes with protection checks turned off.

The kernel vs. the rest of the operating system

The operating system kernel is a crucial piece of an operating system, but it is only a portion of the overall operating system. In most modern operating systems, a portion of the operating system runs in user mode as a library linked into each application. An example is library code that manages an application's menu buttons. To encourage a common user interface across applications, most operating systems provide a library of user interface widgets. Applications can write their own user interface routines, but most developers choose to reuse the routines provided by the operating system. This code could run in the kernel but does not need to do so. If the application crashes, it will not matter if that application's menu buttons stop working. The library code (but not the operating system kernel) *shares fate* with the rest of the application: a problem with one has the same effect as a problem with the other.

Likewise, parts of the operating system can run in their own user-level processes. A window manager is one example. The window manager directs mouse actions and keyboard input that occurs inside a window to the correct application, and the manager also ensures that each application modifies only that application's portion of the screen, and not the operating system's menu bar or any other application's window. Without this restriction, a malicious application could potentially take control of the machine. For example, a virus could present a login prompt that looked identical to the system login, potentially inducing users to disclose their passwords to the attacker.

Why not include the entire operating system — the library code and any user-level processes — in the kernel itself? While that might seem more logical, one reason is that it is often easier to debug user-level code than kernel code. The kernel can use low-level hardware to implement debugging support for breakpoints and for single stepping through application code; to single step the kernel requires an even lower-level debugger running underneath the kernel. The difficulty of debugging operating system kernels was the original motivation behind the development of virtual machines.

More importantly, the kernel must be trusted, as it has full control over the hardware. Any error in the kernel can corrupt the disk, the memory of some unrelated application, or simply crash the system. By separating out code that does not need to be in the kernel, the operating system can become more reliable — a bug in the window system is bad enough, but it would be even worse if it could corrupt the disk. This illustrates the *principle of least privilege*, that security and reliability are enhanced if each part of the system has exactly the privileges it needs to do its job, and no more.

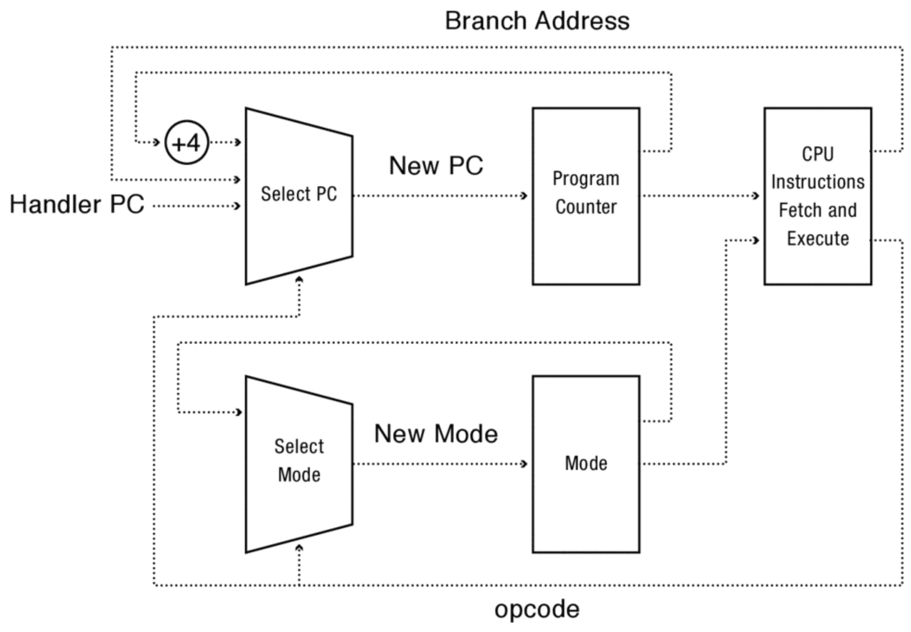


Figure 2.4: The operation of a CPU with kernel and user modes.

Figure 2.4 shows the operation of a dual-mode processor; the program counter and the mode bit together control the processor's operation. In turn, the mode bit is modified by some instructions, just as the program counter is modified by some instructions.

What hardware is needed to let the operating system kernel protect applications and users from one another, yet also let user code run directly on the processor? At a minimum, the hardware must support three things:

- **Privileged Instructions.** All potentially unsafe instructions are prohibited when executing in user mode. (Section 2.2.1)
- **Memory Protection.** All memory accesses outside of a process's valid memory region are prohibited when executing in user mode. (Section 2.2.2)
- **Timer Interrupts.** Regardless of what the process does, the kernel must have a way to periodically regain control from the current process. (Section 2.2.3)

In addition, the hardware must also provide a way to safely transfer control from user mode to kernel mode and back. As the mechanisms to do this are relatively involved, we defer the discussion of that topic to Sections 2.3 and 2.4.

The processor status register and privilege levels

Conceptually, the kernel/user mode is a one-bit register. When set to 1, the processor is in kernel mode and can do anything. When set to 0, the processor is in user mode and is restricted. On most processors, the kernel/user mode is stored in the *processor status register*. This register contains flags that control the processor's operation and is typically not directly accessible to application code. Rather, flags are set or reset as a by-product of executing instructions. For example, the hardware automatically saves the status register to memory when an interrupt occurs because otherwise the interrupt handler code would inadvertently overwrite its contents.

The kernel/user mode bit is one flag in the processor status register, set whenever the kernel is entered and reset whenever the kernel switches back to user mode. Other flags include *condition codes*, set as a side effect of arithmetic operations, to allow a more compact encoding of conditional branch instructions. Still other flags can specify whether the processor is executing with 16-bit, 32-bit, or 64-bit addresses. The specific contents of the processor status register are processor architecture dependent.

Some processor architectures, including the Intel x86, support more than two privilege levels in the processor status register (the x86 supports four privilege levels). The original reason for this was to allow the operating system kernel to be separated into two layers: (i) a core with unlimited access to the machine, and (ii) an outer layer restricted from certain operations, but with more power than completely unprivileged application code. This way, bugs in one part of the operating system kernel might not crash the entire system. However, to our knowledge, neither MacOS, Windows, nor Linux make use of this feature.

A potential future use for multiple privilege levels would be to simplify running an operating system as an application, or virtual machine, on top of another operating system. Applications running on top of the virtual machine operating system would run at user level; the virtual machine would run at some intermediate level; and the true kernel would run in kernel mode. Of course, with only four levels, this does not work for a virtual machine running on a virtual machine running on a virtual machine. For our discussion, we assume the simpler and more universal case of two levels of hardware protection.

2.2.1 Privileged Instructions

Process isolation is possible only if there is a way to limit programs running in user mode from directly changing their privilege level. We discuss in Section 2.3 that processes can indirectly change their privilege level by executing a special instruction, called a *system call*, to transfer control into the kernel at a fixed location defined by the operating system. Other than transferring control into the operating system kernel (that is, in effect, becoming the kernel) at these fixed locations, an application process cannot change its privilege level.

Other instructions are also limited to use by kernel code. The application cannot be allowed to change the set of memory locations it can access; we discuss in Section 2.2.2 how limiting an application to accessing only its own memory is essential to preventing it from either intentionally, or accidentally, corrupting or misusing the data or code from other applications or the operating system. Further, applications cannot disable processor interrupts, as we will explain in Section 2.2.3.

Instructions available in kernel mode, but not in user mode, are called *privileged instructions*. The operating system kernel must be able to execute these instructions to do its work — it needs to change privilege levels, adjust memory access, and disable and enable interrupts. If these instructions were available to applications, then a rogue application would in effect have the power of the operating system kernel.

Thus, while application programs can use only a subset of the full instruction set, the operating system executes in kernel mode with the full power of the hardware.

What happens if an application attempts to access restricted memory or attempts to change its privilege level? Such actions cause a *processor exception*. Unlike taking an exception in a programming language where the language runtime and user code handles the exception, a processor exception causes the processor to transfer control to an exception handler in the operating system kernel. Usually, the kernel simply halts the process after a privilege violation.

EXAMPLE: What could happen if applications were allowed to jump into kernel mode at any location in the kernel?

ANSWER: Although it might seem that the worst that could happen would be that the operating system would crash (bad enough!), this might also allow a malicious application to gain access to privileged data or possibly control over the machine. The operating system kernel implements a set of privileged services on behalf of applications. Typically, one of the first steps in a kernel routine is to verify whether the user has permission to perform the operation; for example, the file system checks if the user has permission to read a file before returning the data. If an application can jump past the permission check, it could potentially evade the kernel's security limits. □

2.2.2 Memory Protection

To run an application process, both the operating system and the application must be resident in memory at the same time. The application must be in memory in order to execute, while the operating system must be there to start the program and to handle any interrupts, processor exceptions, or system calls that happen while the program runs. Further, other application processes may also be stored in memory; for example, you may read email, download songs, Skype, instant message, and browse the web at the same time.

To make memory sharing safe, the operating system must be able to configure the hardware so that each application process can read and write only its own memory, not the memory of the operating system or any other application. Otherwise, an application could modify the operating system kernel's code or data to gain control over the system. For example, the application could change the login program to give the attacker full system administrator privileges. While it might seem that read-only access to memory is harmless, recall that operating systems need to provide both security and privacy. Kernel data structures — such as the file system buffer — may contain private user data. Likewise, user passwords may be stored in kernel memory while they are being verified.

MS/DOS and memory protection

As an illustration of the power of memory protection, MS/DOS was an early Microsoft operating system that did not provide it. Instead, user programs could read and modify any memory location in the system, including operating system data structures. While this was seen as acceptable for a personal computer that was only used by a single person at a time, there were a number of downsides. One obvious problem was system reliability: application bugs frequently crashed the operating system or corrupted other applications. The lack of memory protection also made the system more vulnerable to computer viruses.

Over time, some applications took advantage of the ability to change operating system data structures, for example, to change certain control parameters or to directly manipulate the frame buffer for controlling the display. As a result, changing the operating system became quite difficult; either the new version could not run the old applications, limiting its appeal, or it needed to leave these data structures in precisely the same place as they were in the old version. In other words, memory protection is not only useful for reliability and security; it also helps to enforce a well-defined interface between applications and the operating system kernel to aid future evolvability and portability.

How does the operating system prevent a user program from accessing parts of physical memory? We discuss a wide variety of different approaches in Chapter 8, but early computers pioneered a simple mechanism to provide protection. We describe it now to illustrate the general principle.

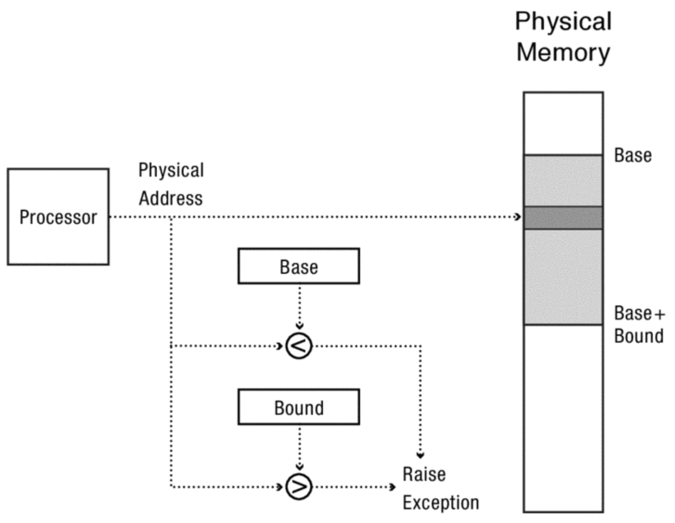


Figure 2.5: Base and bound memory protection using physical addresses. Every code and data address generated by the program is first checked to verify that its address lies within the memory region of the process.

With this approach, a processor has two extra registers, called *base and bound*. The base specifies the start of the process's memory region in physical memory, while the bound gives its endpoint (Figure 2.5). These registers can be changed only by privileged instructions, that is, by the operating system executing in kernel mode. User-level code cannot change their values.

Every time the processor fetches an instruction, it checks the address of the program counter to see if it is between the base and the bound registers. If so, the instruction fetch is allowed to proceed; otherwise, the hardware raises an exception, suspending the program and transferring control back to the operating system kernel. Although it might seem extravagant to perform two extra comparisons for each instruction, memory protection is worth the cost. In fact, we will discuss much more sophisticated and “extravagant" memory protection schemes in Chapter 8.

Likewise, for instructions that read or write data to memory, the processor checks each memory reference against the base and bound registers, generating a processor exception if the boundaries are violated. Complex instructions, such as a block copy instruction, must check every location touched by the instruction, to ensure that the application does not inadvertently or maliciously read or write to a buffer that starts in its own region but that extends into the kernel's region. Otherwise, applications could read or overwrite key parts of the operating system code or data and thereby gain control of the system.

The operating system kernel executes without the base and bound registers, allowing it to access any memory on the system — the kernel's memory or the memory of any application process running on the system. Because applications touch only their own memory, the kernel must explicitly copy any input or output into or out of the application's memory region. For example, a simple program might print "hello world". The kernel must copy the string out of the application's memory region into the screen buffer.

Memory allocation with base and bound registers is simple, analogous to heap memory allocation. When a program starts up, the kernel finds a free block of contiguous physical memory with enough room to store the entire program, its data, heap and execution stack. If the free block is larger than needed, the kernel returns the remainder to the heap for allocation to some other process.

Memory-mapped devices

On most computers, the operating system controls input/output devices — such as the disk, network, or keyboard — by reading and writing to special memory locations. Each device monitors the memory bus for the address assigned to it, and when it sees its address, the device triggers the desired I/O operation.

The operating system can use memory protection to prevent user-level processes from accessing these special memory locations. Thus, memory protection has the added advantage of limiting direct access to input/output devices by user code. By limiting each process to just its own memory locations, the kernel prevents processes from directly reading or writing to the disk controller or other devices. In this way, a buggy or malicious application cannot modify the operating system's image stored on disk, and a user cannot gain access to another user's files without first going through the operating system to check file permissions.

Using physically addressed base and bound registers can provide protection, but this does not provide some important features:

- **Expandable heap and stack.** With a single pair of base and bound registers per process, the amount of memory allocated to a program is fixed when the program starts. Although the operating system can change the bound, most programs have two (or more) memory regions that need to independently expand depending on program behavior. The execution stack holds procedure local variables and grows with the depth of the procedure call graph; the heap holds dynamically allocated objects. Most systems today grow the heap and the stack from opposite sides of program memory; this is difficult to accommodate with a pair of base and bound registers.

- **Memory sharing.** Base and bound registers do not allow memory to be shared between different processes, as would be useful for sharing code between multiple processes running the same program or using the same library.

- **Physical memory addresses.** When a program is compiled and linked, the addresses of its procedures and global variables are set relative to the beginning of the executable file, that is, starting at zero. With the mechanism we have just described using base and bound registers, each program is loaded into physical memory at runtime and must use those physical memory addresses. Since a program may be loaded at different locations depending on what other programs are running at the same time, the kernel must change every instruction and data location that refers to a global address, each time the program is loaded into memory.

- **Memory fragmentation.** Once a program starts, it is nearly impossible to relocate it. The program might store pointers in registers or on the execution stack (for example, the program counter to use when returning from a procedure), and these pointers need to be changed to move the program to a different region of physical memory. Over time, as applications start and finish at irregular times, memory will become increasingly fragmented. Potentially, memory fragmentation may reach a point where there is not enough contiguous space to start a new process, despite sufficient free memory in aggregate.

For these reasons, most modern processors introduce a level of indirection, called *virtual addresses*. With virtual addresses, every process's memory starts at the same place, e.g., zero. Each process thinks that it has the entire machine to itself, although obviously that is not the case in reality. The hardware translates these virtual addresses to physical memory locations. A simple algorithm would be to add the base register to every virtual address so that the process can use virtual addresses starting from zero.

In practice, modern systems use much more complex algorithms to translate between virtual and physical addresses. The layer of indirection provided by virtual addresses gives operating systems enormous flexibility to efficiently manage physical memory. For example, many systems with virtual addresses allocate physical memory in fixed-sized, rather than variable-sized, chunks to reduce fragmentation.

Virtual addresses can also let the heap and the stack start at separate ends of the virtual address space so they can grow according to program need (Figure 2.6). If either the stack or heap grows beyond its initially allocated region, the operating system can move it to a different larger region in physical memory but leave it at the same virtual address. The expansion is completely transparent to the user process. We discuss virtual addresses in more depth in Chapter 8.

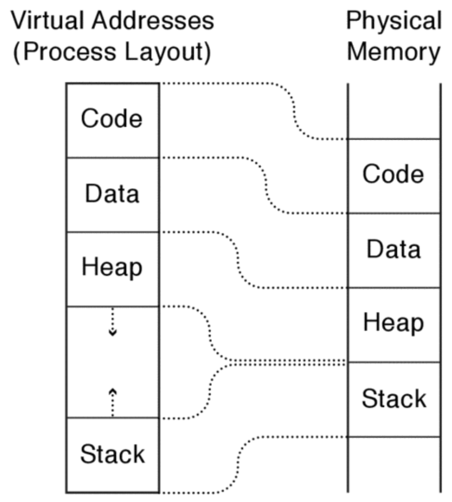


Figure 2.6: Virtual addresses allow the stack and heap regions of a process to grow independently. To grow the heap, the operating system can move the heap in physical memory without changing the heap's virtual address.

Figure 2.7 lists a simple test program to verify that a computer supports virtual addresses. The program has a single static variable; it updates the value of the variable, waits for a few seconds, and then prints the location of the variable and its value.

```
int staticVar = 0;    // a static variable
main() {
    staticVar += 1;

    // sleep causes the program to wait for x seconds
    sleep(10);
    printf ("Address: %x; Value: %d\n", &staticVar, staticVar);
}

Produces:
    Address: 5328; Value: 1
```

Figure 2.7: A simple C program whose output illustrates the difference between execution in physical memory versus virtual memory. When multiple copies of this program run simultaneously, the output does not change.

With virtual addresses, if multiple copies of this program run simultaneously, each copy of the program will print exactly the same result. This would be impossible if each copy were directly addressing physical memory locations. In other words, each instance of the program appears to run in its own complete copy of memory: when it stores a value to a memory location, it alone sees its changes to that location. Other processes change their own copies of the memory location. In this way, a process cannot alter any other process's memory, because it has no way to reference the other process's memory; only the kernel can read or write the memory of a process other than itself.

Address randomization

Computer viruses often work by attacking hidden vulnerabilities in operating system and server code. For example, if the operating system developer forgets to check the length of a user string before copying it into a buffer, the copy can overwrite the data stored immediately after the buffer. If the buffer is stored on the stack, this might allow a malicious user to overwrite the return program counter from the procedure; the attacker can then cause the server to jump to an arbitrary point (for example, into code embedded in the string). These attacks are easier to mount when a program uses the same locations for the same variables each time it runs.

Most operating systems, such as Linux, MacOS, and Windows, combat viruses by randomizing (within a small range) the virtual addresses that a program uses each time it runs. This is called *address space layout randomization*. A common technique is to pick a slightly different start address for the heap and stack for each execution. Thus, in Figure 2.7, if instead we printed the address of a procedure local variable, the address might change from run to run, even though the value of the variable would still be 1.

Some systems have begun to randomize procedure and static variable locations, as well as the offset between adjacent procedure records on the stack to make it harder to force the system to jump to the attacker's code. Nevertheless, each process appears to have its own copy of memory, disjoint from all other processes.

This is very much akin to a set of television shows, each occupying their own universe, even though they all appear on the same television. Events in one show do not (normally) affect the plot lines of other shows. Sitcom characters are blissfully unaware that Jack Bauer has just saved the world from nuclear Armageddon. Of course, just as television shows can from time to time share characters, processes can also communicate if the kernel allows it. We will discuss how this happens in Chapter 3.

EXAMPLE: Suppose we have a "perfect" object-oriented language and compiler in which only an object's methods can access the data inside the object. If the operating system runs only programs written in that language, would it still need hardware memory address protection?

ANSWER: In theory, no, but in practice, yes. The compiler would be responsible for ensuring that no application program read or modified data outside of its own objects. This requires, for example, the language runtime to do garbage collection: once an object is released back to the heap (and possibly reused by some other application), the application cannot continue to hold a pointer to the object.

In practice, this approach means that system security depends on the correct operation of the compiler in addition to the operating system kernel. Any bug in the compiler or language runtime becomes a possible way for an attacker to gain control of the machine. Many languages have extensive runtime libraries to simplify the task of writing programs in that language; often these libraries are written for performance in a language closer to the hardware, such as C. Any bug in a library routine also becomes a possible means for an attacker to gain control.

Although it may seem redundant, many systems use both language-level protection and process-level protection. For example, Google’s Chrome web browser creates a separate process (e.g., one per browser tab) to interpret the HTML, Javascript, or Java on a web page. This way, a malicious attacker must compromise both the language runtime as well as the operating system process boundary to gain control of the client machine. □

2.2.3 Timer Interrupts

Process isolation also requires hardware to provide a way for the operating system kernel to periodically regain control of the processor. When the operating system starts a user-level program, the process is free to execute any user-level (non-privileged) instructions it chooses, call any function in the process’s memory region, load or store any value to its memory, and so forth. To the user program, it appears to have complete control of the hardware within the limits of its memory region.

However, this too is only an illusion. If the application enters an infinite loop, or if the user simply becomes impatient and wants the system to stop the application, then the operating system must be able to regain control. Of course, the operating system needs to execute instructions to decide if it should stop the application, but if the application controls the processor, the operating system by definition is not running on that processor.

The operating system also needs to regain control of the processor in normal operation. Suppose you are listening to music on your computer, downloading a file, and typing at the same time. To smoothly play the music, and to respond in a timely way to user input, the operating system must be able to regain control to switch to a new task.

MacOS and preemptive scheduling

Until 2002, Apple’s MacOS lacked the ability to force a process to yield the processor back to the kernel. Instead, all application programmers were told to design their systems to periodically call into the operating system to check if there was other work to be done. The operating system would then save the state of the original process, switch control to another application, and return only when it again became the original process’s turn. This had a drawback: if a process failed to yield, e.g., because it had a bug and entered an infinite loop, the operating system kernel had no recourse. The user needed to reboot the machine to return control to the operating system. This happened frequently enough that it was given its own name: the “spinning cursor of death.”

Almost all computer systems include a device called a *hardware timer*, which can be set to interrupt the processor after a specified delay (either in time or after some number of instructions have been executed). Each timer interrupts only one processor, so a multiprocessor will usually have a separate timer for each CPU. The operating system might set each timer to expire every few milliseconds; human reaction time is a few hundred of milliseconds. Resetting the timer is a privileged operation, accessible only within the kernel, so that the user-level process cannot inadvertently or maliciously disable the timer.

When the timer interrupt occurs, the hardware transfers control from the user process to the kernel running in kernel mode. Other hardware interrupts, such as to signal the processor that an I/O device has completed its work, likewise transfer control from the user process to the kernel. A timer or other interrupt does not imply that the program has an error; in most cases, after resetting the timer, the operating system resumes execution of the process, setting the mode, program counter and registers back to the values they had immediately before the interrupt occurred. We discuss the hardware and kernel mechanisms for implementing interrupts in Section 2.4.

EXAMPLE: How does the kernel know if an application is in an infinite loop?

ANSWER: It doesn't. Typically, the operating system will terminate a process only when requested by the user or system administrator, e.g., because the application has become non-responsive to user input. The operating system needs to be able to regain control to be able to ask the user if she wants to shut down a particular process. □

2.3 Types of Mode Transfer

Once the kernel has placed a user process in a carefully constructed sandbox, the next question is how to safely transition from executing a user process to executing the kernel, and vice versa. These transitions are not rare events. A high-performance web server, for example, might switch between user mode and kernel mode thousands of times per second. Thus, the mechanism must be both fast and safe, leaving no room for malicious or buggy programs to corrupt the kernel, either intentionally or inadvertently.

2.3.1 User to Kernel Mode

We first focus on transitions from user mode to kernel mode; as we will see, transitioning in the other direction works by "undo"-ing the transition from the user process into the kernel.

There are three reasons for the kernel to take control from a user process: interrupts, processor exceptions, and system calls. Interrupts occur asynchronously — that is, they are triggered by an external event and can cause a transfer to kernel mode after any user-mode instruction.

Processor exceptions and system calls are synchronous events triggered by process execution. We use the term *trap* to refer to any synchronous transfer of control from user mode to the kernel; some systems use the term more generically for any transfer of control from a less privileged to a more privileged level.

- **Interrupts.** An *interrupt* is an asynchronous signal to the processor that some external event has occurred that may require its attention. As the processor executes instructions, it checks for whether an interrupt has arrived. If so, it completes or stalls any instructions that are in progress. Instead of fetching the next instruction, the processor hardware saves the current execution state and starts executing at a specially designated interrupt handler in the kernel. On a multiprocessor, an interrupt is taken on only one of the processors; the others continue to execute as if nothing happened.

  Each different type of interrupt requires its own handler. For timer interrupts, the handler checks if the current process is being responsive to user input to detect if the process has gone into an infinite loop. The timer handler can also switch execution to a different process to ensure that each process gets a turn. If no change is needed, the timer handler resumes execution at the interrupted instruction, transparently to the user process.

Interrupts are also used to inform the kernel of the completion of I/O requests. For example, mouse device hardware triggers an interrupt every time the user moves or clicks on the mouse. The kernel, in turn, notifies the appropriate user process — the one the user was “mousing” across. Virtually every I/O device — the Ethernet, WiFi, hard disk, thumb drive, keyboard, mouse — generates an interrupt whenever some input arrives for the processor and whenever a request completes.

An alternative to interrupts is *polling*: the kernel loops, checking each input/output device to see if an event has occurred that requires handling. Needless to say, if the kernel is polling, it is not available to run user-level code.

Interprocessor interrupts are another source of interrupts. A processor can send an interrupt to any other processor. The kernel uses these interrupts to coordinate actions across the multiprocessor; for example, when a parallel program exits, the kernel sends interrupts to stop the program from continuing to run on any other processor.

Buffer descriptors and high-performance I/O

In early computer systems, the key to good performance was to keep the processor busy; particularly for servers, the key to good performance today is keeping I/O devices, such as the network and disk device, busy. Neither Internet nor disk bandwidth has kept pace with the rapid improvement in processor performance over the past four decades, leaving them relatively more important than the CPU to system performance.

A simple, but inefficient, approach to designing the operating system software to manage an I/O device is to allow only one I/O operation to the device at any one time. In this case, interrupt handling can be a limiting factor to performance. When the device completes a request, it raises an interrupt, causing the device interrupt handler to run. The handler can then issue the next pending request to the hardware. In the meantime, while the processor is handling the interrupt, the device is idle.

For higher performance, the operating system sets up a circular queue of requests for each device to handle. (A network interface will have two queues: one for incoming packets and one for outgoing packets.) Each entry in the queue, called a *buffer descriptor*, specifies one I/O operation: the requested operation (e.g., disk read or write) and the location of the buffer to contain the data. The device hardware reads the buffer descriptor to determine what operations to perform. Provided the queue of buffer descriptors is full, the device can start working on the next operation while the operating system handles with the previous one.

Buffer descriptors are stored in memory, accessed by the device using DMA (direct memory access). An implication is that each logical I/O operation can involve several DMA requests: one to download the buffer descriptor from memory into the device, then to copy the data in or out, and then to store the success/failure of the operation back into buffer descriptor.

- **Processor exceptions.** A *processor exception* is a hardware event caused by user program behavior that causes a transfer of control to the kernel. As with an interrupt, the hardware finishes all previous instructions, saves the current execution state, and starts running at a specially designated exception handler in the kernel. For example, a processor exception occurs whenever a process attempts to perform a privileged instruction or accesses memory outside of its own memory region. Other processor exceptions occur when a process divides an integer by zero, accesses a word of memory with a non-aligned address, attempts to write to read-only memory, and so forth. In these cases, the operating system simply halts the process and returns an error code to the user. On a multiprocessor, the exception only stops execution on the processor triggering the exception; the kernel then needs to send interprocessor interrupts to stop execution of the parallel program on other processors.

Processor exceptions are also caused by more benign program events. For example, to set a breakpoint in a program, the kernel replaces the machine instruction in memory with a special instruction that invokes a trap. When the program reaches that point in its execution, the hardware switches into kernel mode. The kernel restores the old instruction and transfers control to the debugger. The debugger can then examine the program’s variables, set a new breakpoint, and resume the program at the instruction causing the exception.

Processor exceptions and virtualization

Processor exceptions are a particularly powerful tool for virtualization — the emulation of hardware that does not actually exist. As one example, it is common for different versions of a processor architecture family to support some parts of the instruction set and not others, such as when an inexpensive, low-power processor does not support floating point operations. At some cost in performance, the operating system can use processor exceptions to make the difference completely transparent to the user process. When the program issues a floating point instruction, an exception is raised, trapping into the operating system kernel. Instead of halting the process, the operating system can *emulate* the missing instruction, and, on completion, return to the user process at the instruction immediately after the one that caused the exception. In this way, the same program binary can run on different versions of the processor.

More generally, processor exceptions are used to transparently emulate a virtual machine. When a guest operating system is running as a user-level process on top of an operating system, it will attempt to execute privileged instructions as if it were running on physical hardware. These instructions will cause processor exceptions, trapping into the host operating system kernel. To maintain the illusion of physical hardware, the host kernel then performs the requested instruction of behalf of the user-level virtual machine and restarts the guest operating system at the instruction immediately following the one that caused the exception.

As a final example, processor exceptions are a key building block for memory management. With most types of virtual addressing, the processor can be set up to take an exception whenever it reads or writes inside a particular virtual address range. This allows the kernel to treat memory as *virtual* — a portion of the program memory may be stored on disk instead of in physical memory. When the program touches a missing address, the operating system exception handler fills in the data from disk before resuming the program. In this way, the operating system can execute programs that require more memory than can fit on the machine at the same time.

- **System calls.** User processes can also transition into the operating system kernel voluntarily to request that the kernel perform an operation on the user’s behalf. A *system call* is any procedure provided by the kernel that can be called from user level. Most processors implement system calls with a special `trap` or `syscall` instruction.

However, a special instruction is not strictly required; on some systems, a process triggers a system call by executing an instruction with a specific invalid opcode.

As with an interrupt or a processor exception, the trap instruction changes the processor mode from user to kernel and starts executing in the kernel at a pre-defined handler. To protect the kernel from misbehaving user programs, it is essential that the hardware transfers control on a system call to a pre-defined address — user processes *cannot* be allowed to jump to arbitrary places in the kernel.

Operating systems can provide any number of system calls. Examples include system calls to establish a connection to a web server, to send or receive packets over the network, to create or delete files, to read or write data into files, and to create a new user process. To the user program, these are called like normal procedures, with parameters and return values. The caller needs to be concerned only with the interface; it does not need to know that the routine is actually being implemented by the kernel. The kernel handles the details of checking and copying arguments, performing the operation, and copying return values back into the process's memory. When the kernel completes the system call, it resumes user-level execution at the instruction immediately after the trap.

2.3.2 Kernel to User Mode

Just as there are several different types of transitions from user to kernel mode, there are several types of transitions from kernel to user mode:

- **New process.** To start a new process, the kernel copies the program into memory, sets the program counter to the first instruction of the process, sets the stack pointer to the base of the user stack, and switches to user mode.
- **Resume after an interrupt, processor exception, or system call.** When the kernel finishes handling the request, it resumes execution of the interrupted process by restoring its program counter (in the case of a system call, the instruction after the trap), restoring its registers, and changing the mode back to user level.
- **Switch to a different process.** In some cases, such as on a timer interrupt, the kernel switches to a different process than the one that had been running before the interrupt. Since the kernel will eventually resume the old process, the kernel needs to save the process state — its program counter, registers, and so forth — in the process's control block. The kernel can then resume a different process by loading its state — its program counter, registers, and so forth — from the process's control block into the processor and then switching to user mode.
- **User-level upcall.** Many operating systems provide user programs with the ability to receive asynchronous notification of events. The mechanism, which we describe in Section 2.8, is similar to kernel interrupt handling, except at user level.

2.4 Implementing Safe Mode Transfer

Whether transitioning from user to kernel mode or in the opposite direction, care must be taken to ensure that a buggy or malicious user program cannot corrupt the kernel. Although the basic idea is simple, the low-level implementation can be a bit complex: the processor must save its state and switch what it is doing, *while* executing instructions that might alter the state that it is in the process of saving. This is akin to rebuilding a car's transmission while it barrels down the road at 60 mph.

The context switch code must be carefully crafted, and it relies on hardware support. To avoid confusion and reduce the possibility of error, most operating systems have a common sequence of instructions both for entering the kernel — whether due to interrupts, processor exceptions or system calls — and for returning to user level, again regardless of the cause.

At a minimum, this common sequence must provide:

- **Limited entry into the kernel.** To transfer control to the operating system kernel, the hardware must ensure that the entry point into the kernel is one set up by the kernel. User programs cannot be allowed to jump to arbitrary locations in the kernel. For example, the kernel code for handling the read file system call first checks whether the user program has permission to do so. If not, the kernel should return an error. Without limited entry points into the kernel, a malicious program could jump immediately after the code to perform the check, allowing the program to access to anyone's file.

- **Atomic changes to processor state.** In user mode, the program counter and stack point to memory locations in the user process; memory protection prevents the user process from accessing any memory outside of its region. In kernel mode, the program counter and stack point to memory locations in the kernel; memory protection is changed to allow the kernel to access both its own data and that of the user process. Transitioning between the two is atomic — the mode, program counter, stack, and memory protection are all changed at the same time.

- **Transparent, restartable execution.** An event may interrupt a user-level process at any point, between any instruction and the next one. For example, the processor could have calculated a memory address, loaded it into a register, and be about to store a value to that address. The operating system must be able to restore the state of the user program exactly as it was before the interrupt occurred. To the user process, an interrupt is invisible, except that the program temporarily slows down. A "hello world" program is not written to understand interrupts, but an interrupt might still occur while the program is running.

  On an interrupt, the processor saves its current state to memory, temporarily defers further events, changes to kernel mode, and then jumps to the interrupt or exception handler. When the handler finishes, the steps are reversed: the processor state is restored from its saved location, with the interrupted program none the wiser.

With that context, we now describe the hardware and software mechanism for handling an interrupt, processor exception, or system call. Later, we reuse this same basic mechanism as a building block for implementing user-level signals.

2.4.1 Interrupt Vector Table

When an interrupt, processor exception or system call trap occurs, the operating system must take different actions depending on whether the event is a divide-by-zero exception, a file read system call, or a timer interrupt. How does the processor know what code to run?

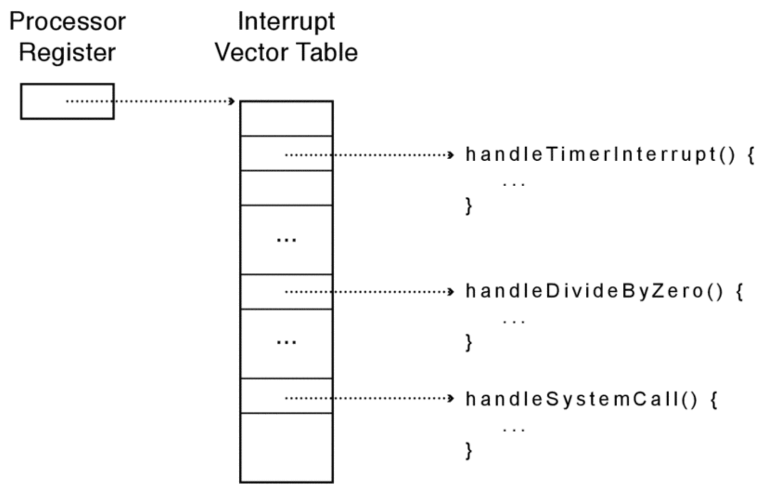


Figure 2.8: An interrupt vector table lists the kernel routines to handle various hardware interrupts, processor exceptions, and system calls.

As Figure 2.8 illustrates, the processor has a special register that points to an area of kernel memory called the *interrupt vector table*. The interrupt vector table is an array of pointers, with each entry pointing to the first instruction of a different handler procedure in the kernel. An *interrupt handler* is the term used for the procedure called by the kernel on an interrupt.

The format of the interrupt vector table is processor-specific. On the x86, for example, interrupt vector table entries 0 - 31 are for different types of processor exceptions (such as divide-by-zero); entries 32 - 255 are for different types of interrupts (timer, keyboard, and so forth); and, by convention, entry 64 points to the system call trap handler. The hardware determines which hardware device caused the interrupt, whether the trap instruction was executed, or what exception condition occurred. Thus, the hardware can select the right entry from the interrupt vector table and invoke the appropriate handler.

Some other processors have a smaller number of entry points, instead putting a code indicating the cause of the interrupt into a special hardware register. In that case, the operating system software uses the code to index into the interrupt vector table.

EXAMPLE: Why is the interrupt vector table stored in kernel rather than user memory?

ANSWER: If the interrupt vector table could be modified by application code, the application could potentially hijack the network by directing all network interrupts to its own code. Similarly, the hardware register that points to the interrupt vector table must be a protected register that can be set only when in kernel mode. □

Multiprocessors and interrupt routing

On a multiprocessor, which of the various processors should take an interrupt? Some early multiprocessors dedicated a single processor (“processor 0") to handle all external interrupts. If an event required a change to what one of the other processors was doing, processor 0 could send an interprocessor interrupt to trigger that processor to switch to a new process.

For systems needing to do a large amount of input and output, such as a web server, directing all I/O through a single processor can become a bottleneck. In modern systems, interrupt routing is increasingly programmable, under control of the kernel. Each processor usually has its own hardware timer. Likewise, disk I/O events can be sent directly to the processor that requested the I/O operation rather than to a random processor. Modern processors can run substantially faster if their data is already loaded into the processor cache, versus if their code and data are in some other processor’s cache.

Efficient delivery of network I/O packets is even more challenging. A high performance server might send and receive tens of thousands of packets per second, representing thousands of different connections. From a processing perspective, it is best to deliver incoming packets to the processor responsible for handling that connection; this requires the network interface hardware to re-direct the incoming packet based on the contents of its header (e.g., the IP address and port number of the client). Recent network controllers accomplish this by supporting multiple buffer descriptor rings for the same device, choosing which ring to use, and therefore which processor to interrupt, based on the header of the arriving packet.

2.4.2 Interrupt Stack

Where should the interrupted process’s state be saved, and what stack should the kernel’s code use?

On most processors, a special, privileged hardware register points to a region of kernel memory called the *interrupt stack*. When an interrupt, processor exception, or system call trap causes a context switch into the kernel, the hardware changes the stack pointer to point to the base of the kernel’s interrupt stack. The hardware automatically saves some of the interrupted process’s registers by pushing them onto the interrupt stack before calling the kernel’s handler.

When the kernel handler runs, it pushes any remaining registers onto the stack before performing its work. When returning from the interrupt, processor exception or system call trap, the reverse occurs: first, the handler pops the saved registers, and then, the hardware restores the registers it saved, returning to the point where the process was interrupted. When returning from a system call, the value of the saved program counter must be incremented so that the hardware returns to the instruction immediately *after* the one that caused the trap.

You might think you could use the process’s user-level stack to store its state. However, a separate, kernel-level interrupt stack is needed for two reasons.

- **Reliability.** The process’s user-level stack pointer might not be a valid memory address (e.g., if the program has a bug), but the kernel handler must continue to work properly.

- **Security.** On a multiprocessor, other threads running in the same process can modify user memory during the system call. If the kernel handler stores its local variables on the user-level stack, the user program might be able to modify the kernel's return address, potentially causing the kernel to jump to arbitrary code.

On a multiprocessor, each processor needs to have its own interrupt stack so that, for example, the kernel can handle simultaneous system calls and exceptions across multiple processors. For each processor, the kernel allocates a separate region of memory as that processor's interrupt stack.

2.4.3 Two Stacks per Process

Most operating system kernels go one step farther and allocate a kernel interrupt stack for every user-level process (and as we discuss in Chapter 4, every thread that executes user code). When a user-level process is running, the hardware interrupt stack points to that process's kernel stack. Note that when a process is running at user level, it is not running in the kernel so its kernel stack is empty.

Allocating a kernel stack per process makes it easier to switch to a new process inside an interrupt or system call handler. For example, a timer interrupt handler might decide to give the processor to a different process. Likewise, a system call might need to wait for an I/O operation to complete; in the meantime, some other process should run. With per-process stacks, to suspend a process, we store a pointer to its kernel stack in the process control block, and switch to the stack of the new process. We describe this mechanism in more detail in Chapter 4.

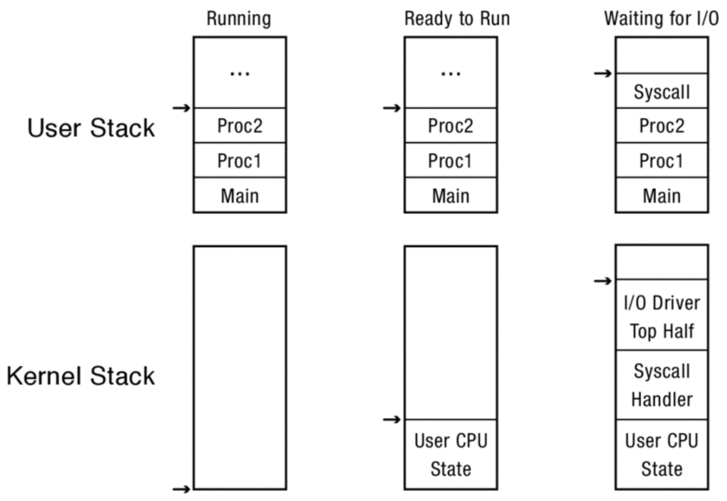


Figure 2.9: In most operating systems, a process has two stacks: one for executing user code and one for kernel code. The Figure shows the kernel and user stacks for various states of a process. When a process is running in user mode, its kernel stack is empty. When a process has been preempted (ready but not running), its kernel stack will contain the user-level processor state at the point when the user process was interrupted. When a process is inside a system call waiting for I/O, the kernel stack contains the context to

be resumed when the I/O completes, and the user stack contains the context to be resumed when the system call returns.

Figure 2.9 summarizes the various states of a process's user and kernel stacks:

- If the process is running on the processor in user mode, its kernel stack is empty, ready to be used for an interrupt, processor exception, or system call.
- If the process is running on the processor in kernel mode — due to an interrupt, processor exception or system call — its kernel stack is in use, containing the saved registers from the suspended user-level computation as well as the current state of the kernel handler.
- If the process is available to run but is waiting for its turn on the processor, its kernel stack contains the registers and state to be restored when the process is resumed.
- If the process is waiting for an I/O event to complete, its kernel stack contains the suspended computation to be resumed when the I/O finishes.

UNIX and kernel stacks

In the original implementation of UNIX, kernel memory was at a premium; main memory was roughly one million times more expensive per byte than it is today. The initial system could run with only 50KB of main memory. Instead of allocating an entire interrupt stack per process, UNIX allocated just enough memory in the process control block to store the user-level registers saved on a mode switch. In this way, UNIX could suspend a user-level process with the minimal amount of memory. UNIX still needed a few kernel stacks: one to run the interrupt handler and one for every system call waiting for an I/O event to complete, but that is much less than one for every process.

Of course, now that memory is much cheaper, most systems keep things simple and allocate a kernel stack per process or thread.

2.4.4 Interrupt Masking

Interrupts arrive asynchronously; the processor could be executing either user or kernel code when an interrupt arrives. In certain regions of the kernel — such as inside interrupt handlers themselves, or inside the CPU scheduler — taking an interrupt could cause confusion. If an interrupt handler is interrupted, we cannot set the stack pointer to point to the base of the kernel's interrupt stack — doing so would obliterate the state of the first handler.

To simplify the kernel design, the hardware provides a privileged instruction to temporarily defer delivery of an interrupt until it is safe to do so. On the x86 and several other processors, this instruction is called *disable interrupts*. However, this is a misnomer: the interrupt is only deferred (masked), and not ignored. Once a corresponding *enable interrupts* instruction is executed, any pending interrupts are delivered to the processor. The instructions to mask and unmask interrupts must be privileged; otherwise, user code could inadvertently or maliciously disable the hardware timer, allowing the machine to freeze.

If multiple interrupts arrive while interrupts are disabled, the hardware delivers them in turn when interrupts are re-enabled. However, since the hardware has limited buffering for pending interrupts, some interrupts may be lost if interrupts are disabled for too long a period of time. Generally, the hardware will buffer one interrupt of each type; the interrupt handler is responsible for checking the device hardware to see if multiple pending I/O events need to be processed.

Interrupt handlers: top and bottom halves

When a machine invokes an interrupt handler because some hardware event occurred (e.g., a timer expired, a key was pressed, a network packet arrived, or a disk I/O completed), the processor hardware typically masks interrupts while the interrupt handler executes. While interrupts are disabled, another hardware event will not trigger another invocation of the interrupt handler until the interrupt is re-enabled.

Some interrupts can trigger a large amount of processing, and it is undesirable to leave interrupts masked for too long. Hardware I/O devices have a limited amount of buffering, which can lead to dropped events if interrupts are not processed in a timely fashion. For example, keyboard hardware can drop keystrokes if the keyboard buffer is full. Interrupt handlers are therefore divided into a *top half* and a *bottom half* . Unfortunately, this terminology can differ a bit from system to system; in Linux, the sense of top and bottom are reversed. In this book, we adopt the more common (non-Linux) usage.

The interrupt handler's bottom half is invoked by the hardware and executes with interrupts masked. It is designed to complete quickly. The bottom half typically saves the state of the hardware device, resets it so that it can receive a new event, and notifies the scheduler that the top half needs to run. At this point, the bottom half is done, and it can re-enable interrupts and return to the interrupted task or (if the event is high priority) switch to the top half but with interrupts enabled. When the top half runs, it can do more general kernel tasks, such as parsing the arriving packet, delivering it to the correct user-level process, sending an acknowledgment, and so forth. The top half can also do operations that require the kernel to wait for exclusive access to shared kernel data structures, the topic of Chapter 5.

If the processor takes an interrupt in kernel mode with interrupts enabled, it is safe to use the current stack pointer rather than resetting it to the base of the interrupt stack. This approach can recursively push a series of handlers' states onto the stack; then, as each one completes, its state is popped from the stack, and the earlier handler is resumed where it left off.

2.4.5 Hardware Support for Saving and Restoring Registers

An interrupted process's registers must be saved so that the process can be restarted exactly where it left off. Because the handler might change the values in those registers as it executes, the state must be saved *before* the handler runs. Because most instructions modify the contents of registers, the hardware typically provides special instructions to make it easier to save and restore user state.

To make this concrete, consider the x86 architecture. Rather than relying on handler software to do all the work, when an interrupt or trap occurs:

- If the processor is in user mode, the x86 pushes the interrupted process's stack pointer onto the kernel's interrupt stack and switches to the kernel stack.
- The x86 pushes the interrupted process's instruction pointer.
- The x86 pushes the x86 *processor status word*. The processor status word includes control bits, such as whether the most recent arithmetic operation in the interrupted code resulted in a positive, negative, or zero value. This needs to be saved and restored for the correct behavior of any subsequent conditional branch instruction.

The hardware saves the values for the stack pointer, program counter, and processor status word *before* jumping through the interrupt vector table to the interrupt handler. Once the handler starts running, these values will be those of the handler, not those of the interrupted process.

Once the handler starts running, it can use the pushad ("push all double") instruction to save the remaining registers onto the stack. This instruction saves all 32-bit x86 integer registers. On a 16-bit x86, pusha is used instead. Because the kernel does not typically perform floating point operations, those do not need to be saved unless the kernel switches to a different process.

The x86 architecture has complementary features for restoring state: a popad instruction to pop an array of integer register values off the stack into the registers and an iret (return from interrupt) instruction that loads a stack pointer, instruction pointer, and processor status word off of the stack into the appropriate processor registers.

Architectural support for fast mode switches

Some processor architectures are able to execute user- and kernel-mode switches very efficiently, while other architectures are much slower at performing these switches.

The SPARC architecture is in the first camp. SPARC defines a set of *register windows* that operate like a hardware stack. Each register window includes a full set of the registers defined by the SPARC instruction set. When the processor performs a procedure call, it shifts to a new window, so the compiler never needs to save and restore registers across procedure calls, making them quite fast. (At a deep enough level of recursion, the SPARC will run out of its register windows; it then takes an exception that saves half the windows and resumes execution. Another exception occurs when the processor pops its last window, allowing the kernel to reload the saved windows.)

Mode switches can be quite fast on the SPARC. On a mode switch, the processor switches to a different register window. The kernel handler can then run, using the registers from the new window and not disturbing the values stored in the interrupted process's copy of its registers. Unfortunately, this comes at a cost: switching between different processes is quite expensive on the SPARC, as the kernel needs to save and restore the entire register set of every active window.

The Motorola 88000 was in the second camp. The 88000 was an early pipelined architecture; now, almost all modern computers are pipelined. For improved performance, pipelined architectures execute multiple instructions at the same time. For example, one instruction is being fetched while another is being decoded, a third is completing a floating point operation, and a fourth is finishing a store to memory. When an interrupt or processor exception occurred on the 88000, the pipeline operation was suspended, and the operating system kernel was required to save and restore the entire state of the pipeline to preserve transparency to user code.

Most modern processors with deep execution pipelines, such as the x86, instead provide *precise interrupts*: the hardware first completes all instructions that occur, in program order, before the interrupted instruction. The hardware annuls any instruction that occurs, in program order, after the interrupt or trap, even if the instruction is in progress when the processor detects the interrupt.

2.5 Putting It All Together: x86 Mode Transfer

The high level steps needed to handle an interrupt, processor exception, or system call are simple, but the details require some care.

To give a concrete example of how such "carefully crafted" code works, we now describe one way to implement an interrupt-triggered mode switch on the x86 architecture. Different operating systems on the x86 follow this basic approach, though details differ. Similarly, different architectures handle the same types of issues, but they may do so with different hardware support.

First, we provide some background on the x86 architecture. The x86 is segmented, so pointers come in two parts: (i) a segment, a region of memory such as code, data, or stack, and (ii) an offset within that segment. The current user-level instruction is a combination of the code segment (cs register) plus the instruction pointer (eip register). Likewise, the current stack position is the combination of the stack segment (ss) and the stack pointer within the stack segment (esp). The current privilege level is stored as the low-order bits of the cs register rather than in the processor status word (eflags register). The eflags register has condition codes that are modified as a by-product of executing instructions; the eflags register also has other flags that control the processor's behavior, such as whether interrupts are masked or not.

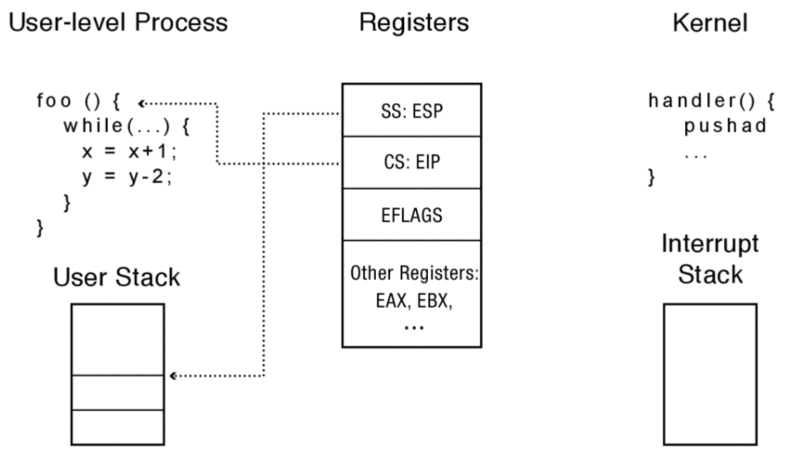


Figure 2.10: State of the system before an interrupt handler is invoked on the x86 architecture. SS is the stack segment, ESP is the stack pointer, CS is the code segment, and EIP is the program counter. The program counter and stack pointer refer to locations in the user process, and the interrupt stack is empty.

When a user-level process is running, the current state of the processor, stack, kernel interrupt vector table, and kernel stack is illustrated in Figure 2.10. When a processor exception or system call trap occurs, the hardware carefully saves a small amount of the interrupted thread state, leaving the system as shown in Figure 2.11:

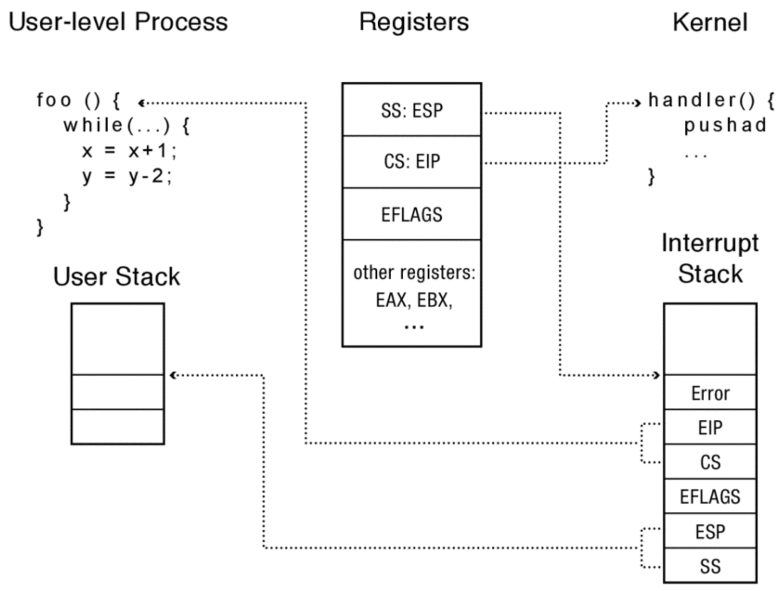


Figure 2.11: State of the system after the x86 hardware has jumped to the interrupt handler. The hardware saves the user context on the kernel interrupt stack and changes the program counter/stack to locations in kernel memory.

1. **Mask interrupts.** The hardware starts by preventing any interrupts from occurring while the processor is in the middle of switching from user mode to kernel mode.

2. **Save three key values.** The hardware saves the values of the stack pointer (the x86 esp and ss registers), the execution flags (the x86 eflags register), and the instruction pointer (the x86 eip and cs registers) to internal, temporary hardware registers.

3. **Switch onto the kernel interrupt stack.** The hardware then switches the stack segment/stack pointer to the base of the kernel interrupt stack, as specified in a special hardware register.

4. **Push the three key values onto the new stack.** Next, the hardware stores the internally saved values onto the stack.

5. **Optionally save an error code.** Certain types of exceptions, such as page faults, generate an error code to provide more information about the event; for these exceptions, the hardware pushes this code, making it the top item on the stack. For other types of events, the software interrupt handler pushes a dummy value onto the stack so that the stack format is identical in both cases.

6. **Invoke the interrupt handler.** Finally, the hardware changes the code segment/program counter to the address of the interrupt handler procedure. A special register in the processor contains the location of the interrupt vector table in kernel memory. This register can only be modified by the kernel. The type of interrupt is mapped to an index in this array, and the code segment/program counter is set to the value at this index.

This starts the handler software.

The handler must first save the rest of the interrupted process's state — it needs to save the other registers before it changes them! The handler pushes the rest of the registers, including the current stack pointer, onto the stack using the x86 pushad instruction.

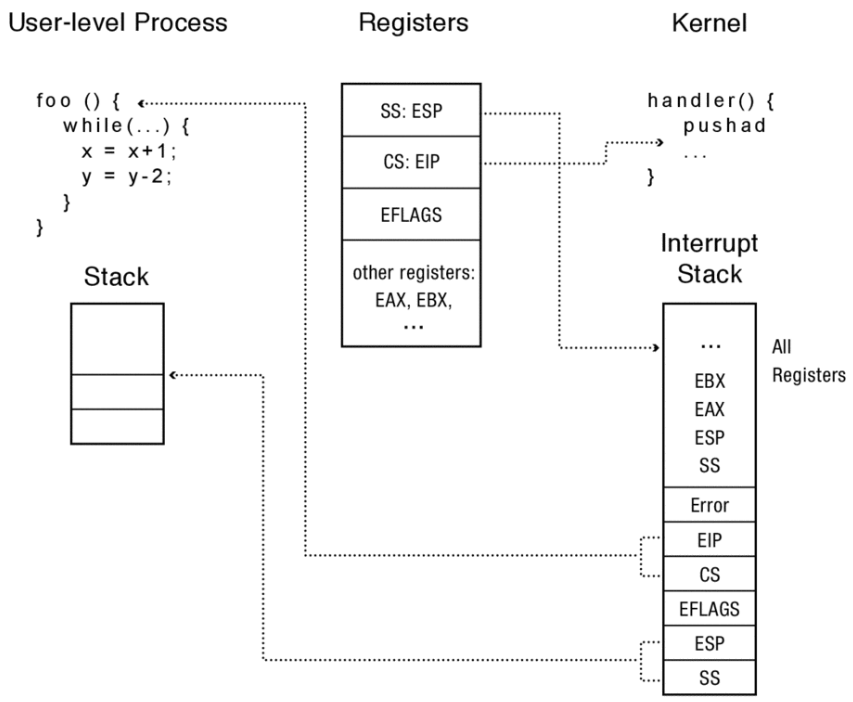


Figure 2.12: State of the system after the interrupt handler has started executing on the x86 architecture. The handler first saves the current state of the processor registers, since it may overwrite them. Note that this saves the stack pointer twice: first, the user stack pointer then the kernel stack pointer.

As Figure 2.12 shows, at this point the kernel's interrupt stack holds (1) the stack pointer, execution flags, and program counter saved by the hardware, (2) an error code or dummy value, and (3) a copy of all of the general registers (including the stack pointer but not the instruction pointer or eflags register).

Once the handler has saved the interrupted thread's state to the stack, it can use the registers as it pleases, and it can push additional items onto the stack. So, the handler can now do whatever work it needs to do.

When the handler completes, it can resume the interrupted process. To do this, the handler pops the registers it saved on the stack. This restores all registers except the execution flags, program counter, and stack pointer. For the x86 instruction set, the popad instruction is commonly used. The handler also pops the error value off the stack.

Finally, the handler executes the x86 iret instruction to restore the code segment, program counter, execution flags, stack segment, and stack pointer from the kernel's interrupt stack.

This restores the process state to exactly what it was before the interrupt. The process continues execution as if nothing happened.

A small but important detail occurs when the hardware takes an exception to emulate an instruction in the kernel, e.g., for missing floating point hardware. If the handler returns back to the instruction that caused the exception, another exception would instantly recur! To prevent an infinite loop, the exception handler modifies the program counter stored at the base on the stack to point to the instruction immediately after the one causing the mode switch. The iret instruction can then return to the user process at the correct location.

For a system call trap, the Intel x86 hardware does the increment when it saves the user-level state. The program counter for the instruction after the trap is saved on the kernel's interrupt stack.

EXAMPLE: A trapframe is the data stored by the hardware and interrupt handler at the base of the interrupt stack, describing the state of the user-level execution context. Typically, a pointer to the trapframe is passed as an argument to the handler, e.g., to allow system calls to access arguments passed in registers.

How large is the 32-bit x86 trapframe in the example given above?

ANSWER: The hardware saves six registers; the interrupt handler saves another eight general-purpose registers. In all, **56 bytes** are saved in the trapframe. □

2.6 Implementing Secure System Calls

The operating system kernel constructs a restricted environment for process execution to limit the impact of erroneous and malicious programs on system reliability. Any time a process needs to perform an action outside of its protection domain — to create a new process, read from the keyboard, or write a disk block — it must ask the operating system to perform the action on its behalf, via a system call.

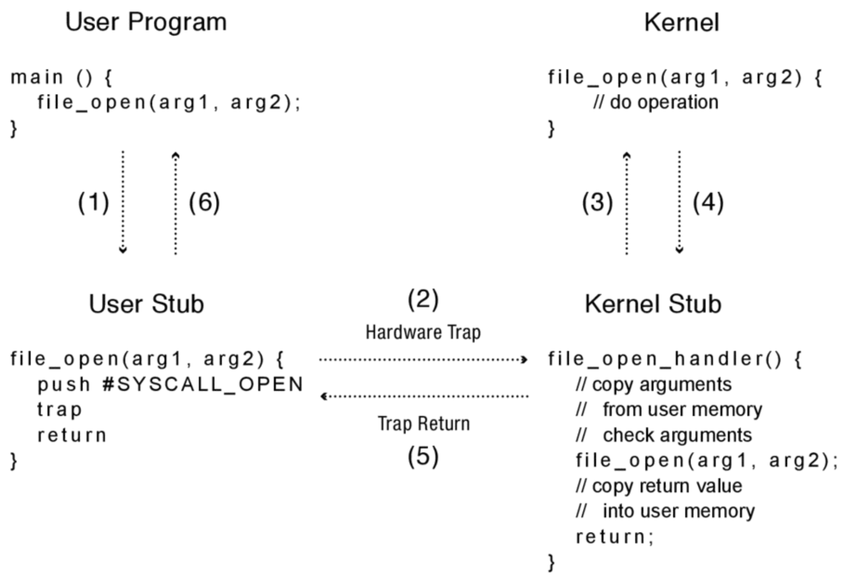
System calls provide the illusion that the operating system kernel is simply a set of library routines available to user programs. To the user program, the kernel provides a set of system call procedures, each with its own arguments and return values, that can be called like any other routine. The user program need not concern itself with how the kernel implements these calls.

Implementing system calls requires the operating system to define a *calling convention* — how to name system calls, pass arguments, and receive return values across the user/kernel boundary. Typically, the operating system uses the same convention as the compiler uses for normal procedures — some combination of passing arguments in registers and on the execution stack.

Once the arguments are in the correct format, the user-level program can issue a system call by executing the trap instruction to transfer control to the kernel. System calls, like interrupts and processor exceptions, share the same mechanism for switching between user and kernel mode. In fact, the x86 instruction to trap into the kernel on a system call is called int, for “software interrupt.”

Inside the kernel, a procedure implements each system call. This procedure behaves exactly as if the call was made from within the kernel but with one notable difference: the kernel must implement its system calls in a way that protects itself from all errors and attacks that might be launched by the misuse of the interface. Of course, most applications will use the interface correctly! But errors in an application program must not crash the kernel, and a computer virus must not be able to use the system call interface to take control of the kernel. One can think of this as an extreme version of defensive programming: the kernel should always assume that the parameters passed to a system call are intentionally designed to be as malicious as possible.

We bridge these two views — the user program calling the system call, and the kernel implementing the system call — with a pair of stubs. A *pair of stubs* is a pair of procedures that mediate between two environments, in this case between the user program and the kernel. Stubs also mediate procedure calls between computers in a distributed system.



```
main () {
    file_open(arg1, arg2);
}
```

```
file_open(arg1, arg2) {
    push #SYSCALL_OPEN
    trap
    return
}
```

```
file_open(arg1, arg2) {
      // do operation
}
```

```
file_open_handler() {
    // copy arguments
    //   from user memory
    //   check arguments
    file_open(arg1, arg2);
    // copy return value
    //   into user memory
    return;
}
```

Figure 2.13: A pair of stubs mediates between the user-level caller and the kernel’s implementation of system calls. The code is for the file_open system call; other calls have their own stubs. (1) The user process makes a normal procedure call to a stub linked with the process. (2) The stub executes the trap instruction. This transfers control to the kernel trap handler. The trap handler copies and checks its arguments and then (3) calls a routine to do the operation. Once the operation completes, (4) the code

returns to the trap handler, which copies the return value into user memory and (5) resumes the user stub immediately after the trap. (6) The user stub returns to the user-level caller.

Figure 2.13 illustrates the sequence of steps involved in a system call:

1. The user program calls the user stub in the normal way, oblivious to the fact the implementation of the procedure is in fact in the kernel.
2. The user stub fills in the code for the system call and executes the trap instruction.
3. The hardware transfers control to the kernel, vectoring to the system call handler. The handler acts as a stub on the kernel side, copying and checking arguments and then calling the kernel implementation of system call.
4. After the system call completes, it returns to the handler.
5. The handler returns to user level at the next instruction in the stub.
6. The stub returns to the caller.

```
// We assume that the caller put the filename onto the stack,
// using the standard calling convention for the x86.

open:
// Put the code for the system call we want into %eax.
    movl #SysCallOpen, %eax

// Trap into the kernel.
    int #TrapCode

// Return to the caller; the kernel puts the return value in %eax.
    ret
```

Figure 2.14: User-level library stub for the file system open system call for the x86 processor. SysCallOpen is the code for the specific system call to run. TrapCode is the index into the x86 interrupt vector table for the system call handler.

We next describe these steps in more detail. Figure 2.14 illustrates the behavior of the user-level stub for the x86. The operating system provides a library routine for each system call that takes its arguments, reformats them according to the calling convention, and executes a trap instruction. When the kernel returns, the stub returns the result provided by the kernel. Of course, the user program need not use the library routine — it is free to trap directly to the kernel; in turn, the kernel must protect itself from misbehaving programs that do not format arguments correctly.

The system call calling convention is arbitrary. In Figure 2.14, the code passes its arguments on the user stack, storing the system call code in the register %eax. The return value comes back in %eax, so there is no work to do on the return.

The int instruction saves the program counter, stack pointer, and eflags on the kernel stack before jumping to the system call handler through the interrupt vector table. The kernel handler saves any additional registers that must be preserved across function calls. It then examines the system call integer code in %eax, verifies that it is a legal opcode, and calls the correct stub for that system call.

The kernel stub has four tasks:

- **Locate system call arguments.** Unlike a regular kernel procedure, the arguments to a system call are stored in user memory, typically on the user stack. Of course, the user stack pointer may be corrupted! Even if it is valid, it is a virtual, not a physical, address. If the system call has a pointer argument (e.g., a file name or buffer), the stub must check the address to verify it is a legal address within the user domain. If so, the stub converts it to a physical address so that the kernel can safely use it. In Figure 2.14, the pointer to the string representing the file name is stored on the stack; therefore, the stub must check and translate both the stack address and the string pointer.

- **Validate parameters.** The kernel must also protect itself against malicious or accidental errors in the format or content of its arguments. A file name is typically a zero-terminated string, but the kernel cannot trust the user code to always work correctly. The file name may be corrupted; it may point to memory outside the application's region; it may start inside the application's memory region but extend beyond it; the application may not have permission to access the file; the file may not exist; and so forth. If an error is detected, the kernel returns it to the user program; otherwise, the kernel performs the operation on the application's behalf.

- **Copy before check.** In most cases, the kernel copies system call parameters into kernel memory before performing the necessary checks. The reason for this is to prevent the application from modifying the parameter *after* the stub checks the value, but *before* the parameter is used in the actual implementation of the routine. This is called a *time of check vs. time of use* (TOCTOU) attack. For example, the application could call open with a valid file name but, after the check, change the contents of the string to be a different name, such as a file containing another user's private data.

  TOCTOU is not a new attack — the first occurrence dates from the mid-1960's. While it might seem that a process necessarily stops whenever it does a system call, this is not always the case. For example, if one process shares a memory region with another process, then the two processes working together can launch a TOCTOU attack. Similarly, a parallel program running on two processors can launch a TOCTOU attack, where one processor traps into the kernel while the other modifies the string at precisely the right (or wrong) time. Note that the kernel needs to be correct in every case, while the attacker can try any number of times before succeeding.

- **Copy back any results.** For the user program to access the results of the system call, the stub must copy the result from the kernel into user memory. Again, the kernel must first check the user address and convert it to a kernel address before performing the copy.

Putting this together, Figure 2.15 shows the kernel stub for the system call open. In this case, the return value fits in a register so the stub can return directly; in other cases, such as a file read, the stub would need to copy data back into a user-level buffer.

```
int KernelStub_Open() {
    char *localCopy[MaxFileNameSize + 1];

// Check that the stack pointer is valid and that the arguments are stored at
// valid addresses.

    if (!validUserAddressRange(userStackPointer, userStackPointer + size of arguments))
        return error_code;

// Fetch pointer to file name from user stack and convert it to a kernel pointer.

    filename = VirtualToKernel(userStackPointer);

// Make a local copy of the filename.  This prevents the application
// from changing the name surreptitiously.

// The string copy needs to check each address in the string before use to make sure
// it is valid.

// The string copy terminates after it copies MaxFileNameSize to ensure we
// do not overwrite our internal buffer.

    if (!VirtualToKernelStringCopy(filename, localCopy, MaxFileNameSize))
        return error_code;

// Make sure the local copy of the file name is null terminated.

    localCopy[MaxFileNameSize] = 0;

// Check if the user is permitted to access this file.

    if (!UserFileAccessPermitted(localCopy, current_process)
        return error_code;

// Finally, call the actual routine to open the file.  This returns a file
// handle on success, or an error code on failure.

    return Kernel_Open(localCopy);
}
```

Figure 2.15: Stub routine for the open system call inside the kernel. The kernel must validate all parameters to a system call before it uses them.

After the system call finishes, the handler pops any saved registers (except %eax) and uses the iret instruction to return to the user stub immediately after the trap, allowing the user stub to return to the user program.

2.7 Starting a New Process

Thus far, we have described how to transfer control from a user-level process to the kernel on an interrupt, processor exception, or system call and how the kernel resumes execution at user level when done.

We now examine how to start running at user level in the first place. The kernel must:

- Allocate and initialize the process control block.
- Allocate memory for the process.
- Copy the program from disk into the newly allocated memory.
- Allocate a user-level stack for user-level execution.
- Allocate a kernel-level stack for handling system calls, interrupts and processor exceptions.

To start running the program, the kernel must also:

- **Copy arguments into user memory.** When starting a program, the user may give it arguments, much like calling a procedure. For example, when you click on a file icon in MacOS or Windows, the window manager asks the kernel to start the application associated with the file, passing it the file name to open. The kernel copies the file name from the memory of the window manager process to a special region of memory in the new process. By convention, arguments to a process are copied to the base of the user-level stack, and the user's stack pointer is incremented so those addresses are not overwritten when the program starts running.

- **Transfer control to user mode.** When a new process starts, there is no saved state to restore. While it would be possible to write special code for this case, most operating systems re-use the same code to exit the kernel for starting a new process and for returning from a system call. When we create the new process, we allocate a kernel stack to it, and we reserve room at the bottom of the kernel stack for the initial values of its user-space registers, program counter, stack pointer, and processor status word. To start the new program, we can then switch to the new stack and jump to the end of the interrupt handler. When the handler executes popad and iret, the processor "returns" to the start of the user program.

Finally, although you can think of a user program as starting with a call to main, in fact the compiler inserts one level of indirection. It puts a stub at the location in the process's memory where the kernel will jump when the process starts. The stub's job is to call main and then, if main returns, to call exit — the system call to terminate the process. Without the stub, a user program that returned from main would try to pop the return program counter, and since there is no such address on the stack, the processor would start executing random code.

```
start(arg1, arg2) {
    main(arg1, arg2);  // Call program main.
```

```
        exit();     // If main returns, call exit.
    }
```

2.8 Implementing Upcalls

We can use system calls for most of the communication between applications and the operating system kernel. When a program requests a protected operation, it can trap to ask the kernel to perform the operation on its behalf. Likewise, if the application needs data inside the kernel, a system call can retrieve it.

To allow applications to implement operating system-like functionality, we need something more. For many of the reasons that kernels need interrupt-based event delivery, applications can also benefit from being told when events occur that need their immediate attention. Throughout this book, we will see this pattern repeatedly: the need to *virtualize* some part of the kernel so that applications can behave more like operating systems. We call virtualized interrupts and exceptions *upcalls*. In UNIX, they are called *signals*; in Windows, they are *asynchronous events*.

There are several uses for immediate event delivery with upcalls:

- **Preemptive user-level threads.** Just as the operating system kernel runs multiple processes on a single processor, an application may run multiple tasks, or threads, in a process. A user-level thread package can use a periodic timer upcall as a trigger to switch tasks, to share the processor more evenly among user-level tasks or to stop a runaway task, e.g., if a web browser needs to terminate an embedded third party script.

- **Asynchronous I/O notification.** Most system calls wait until the requested operation completes and then return. What if the process has other work to do in the meantime? One approach is *asynchronous I/O*: a system call starts the request and returns immediately. Later, the application can poll the kernel for I/O completion, or a separate notification can be sent via an upcall to the application when the I/O completes.

- **Interprocess communication.** Most interprocess communication can be handled with system calls — one process writes data, while the other reads it sometime later. A kernel upcall is needed if a process generates an event that needs the instant attention of another process. As an example, UNIX sends an upcall to notify a process when the debugger wants to suspend or resume the process. Another use is for logout — to notify applications that they should save file data and cleanly terminate.

- **User-level exception handling.** Earlier, we described a mechanism where processor exceptions, such as divide-by-zero errors, are handled by the kernel. However, many applications have their own exception handling routines, e.g., to ensure that files are saved before the application shuts down. For this, the operating system needs to inform the application when it receives a processor exception so the application runtime, rather than the kernel, handles the event.

- **User-level resource allocation.** Operating systems allocate resources — deciding which users and processes should get how much CPU time, how much memory, and so forth. In turn, many applications are resource adaptive — able to optimize their behavior to differing amounts of CPU time or memory. An example is Java garbage collection. Within limits, a Java process can adapt to different amounts of available memory by changing the frequency with which it runs its garbage collector. The more memory, the less time Java needs to run its collector, speeding execution. For this, the operating system must inform the process when its allocation changes, e.g., because some other process needs more or less memory.

Upcalls from kernels to user processes are not always needed. Many applications are more simply structured around an event loop that polls for events and then processes each event in turn. In this model, the kernel can pass data to the process by sending it events that do not need to be handled immediately. In fact, until recently, Windows lacked support for the immediate delivery of upcalls to user-level programs.

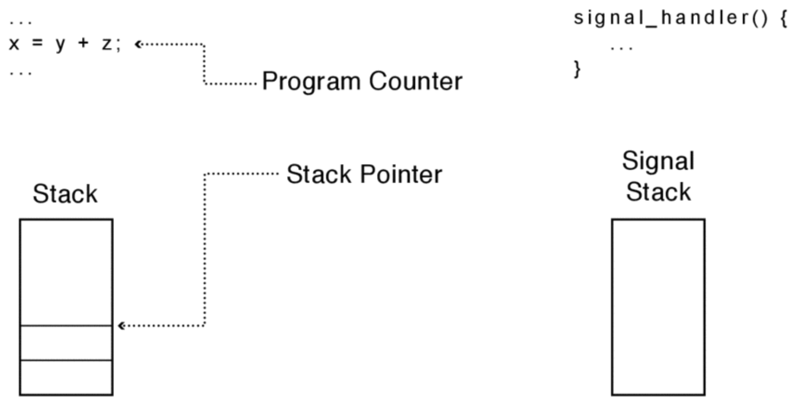


Figure 2.16: The state of the user program and signal handler before a UNIX signal. UNIX signals behave analogously to processor exceptions, but at user level.

We next describe UNIX signals as a concrete example of kernel support for upcalls. As shown in Figures 2.16 and 2.17, UNIX signals share many similarities with hardware interrupts:

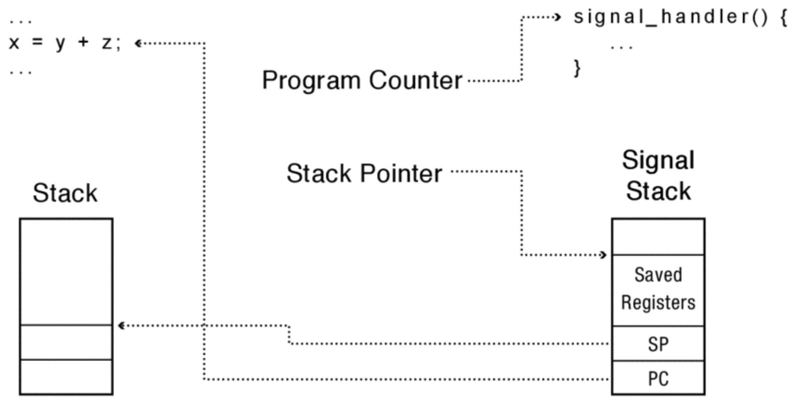


Figure 2.17: The state of the user program and signal handler during a UNIX signal. The signal stack stores the state of the hardware registers at the point where the process was interrupted, with room for the signal handler to execute on the signal stack.

- **Types of signals.** In place of hardware-defined interrupts and processor exceptions, the kernel defines a limited number of signal types that a process can receive.
- **Handlers.** Each process defines its own handlers for each signal type, much as the kernel defines its own interrupt vector table. If a process does not define a handler for a specific signal, then the kernel calls a default handler instead.
- **Signal stack.** Applications have the option to run UNIX signal handlers on the process's normal execution stack or on a special signal stack allocated by the user process in user memory. Running signal handlers on the normal stack makes it more difficult for the signal handler to manipulate the stack, e.g., if the runtime needs to raise a language-level exception.
- **Signal masking.** UNIX defers signals for events that occur while the signal handler for those types of events is in progress. Instead, the signal is delivered once the handler returns to the kernel. UNIX also provides a system call for applications to mask signals as needed.
- **Processor state.** The kernel copies onto the signal stack the saved state of the program counter, stack pointer, and general-purpose registers at the point when the program stopped. Normally, when the signal handler returns, the kernel reloads the saved state into the processor to resume program execution. The signal handler can also modify the saved state, e.g., so that the kernel resumes a different user-level task when the handler returns.

The mechanism for delivering UNIX signals to user processes requires only a small modification to the techniques already described for transferring control across the kernel-user boundary. For example, on a timer interrupt, the hardware and the kernel interrupt handler save the state of the user-level computation. To deliver the timer interrupt to user level, the kernel copies that saved state to the bottom of the signal stack, resets the saved state to point to the signal handler and signal stack, and then exits the kernel handler. The iret instruction then resumes user-level execution at the signal handler. When the signal handler returns, these steps are unwound: the processor state is copied back from the signal handler into kernel memory, and the iret returns to the original computation.

2.9 Case Study: Booting an Operating System Kernel

When a computer boots, it sets the machine's program counter to start executing at a pre-determined position in memory. Since the computer is not yet running, the initial machine instructions must be fetched and executed immediately after the power is turned on before the system has had a chance to initialize its DRAM. Instead, systems typically use a special read-only hardware memory (*Boot ROM*) to store their boot instructions. On most x86 personal computers, the boot program is called the BIOS, for "Basic Input/Output System".

There are several drawbacks to trying to store the entire kernel in ROM. The most significant problem is that the operating system would be hard to update. ROM instructions are fixed when the computer is manufactured and (except in rare cases) are never changed. If an error occurs while the BIOS is being updated, the machine can be left in a permanently unusable state — unable to boot and unable to complete the update of the BIOS.

By contrast, operating systems need frequent updates, as bugs and security vulnerabilities are discovered and fixed. This, and the fact that ROM storage is relatively slow and expensive, argues for putting only a small amount of code in the BIOS.

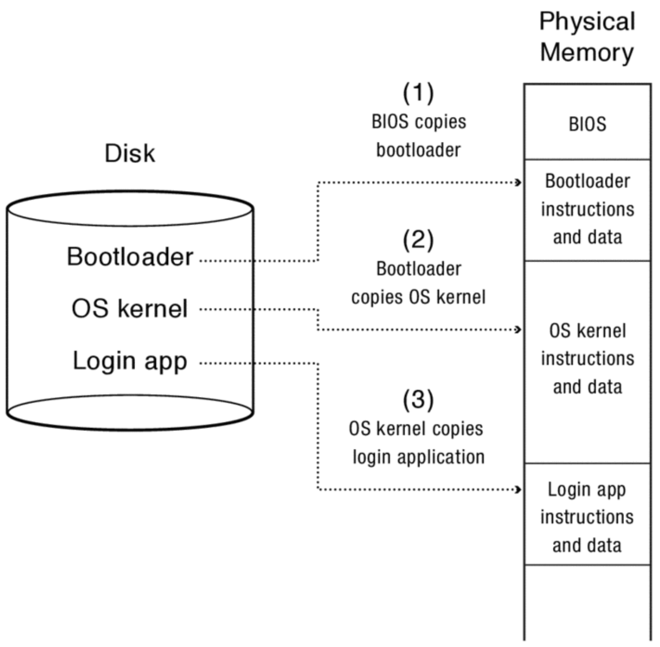

Physical Memory
Disk
Bootloader
OS kernel
Login app
(1) BIOS copies bootloader
(2) Bootloader copies OS kernel
(3) OS kernel copies login application
BIOS
Bootloader instructions and data
OS kernel instructions and data
Login app instructions and data

Figure 2.18: The boot ROM copies the bootloader image from disk into memory, and the bootloader copies

the operating system kernel image from disk into memory.

Instead, the BIOS provides a level of indirection, as illustrated in Figure 2.18. The BIOS reads a fixed-size block of bytes from a fixed position on disk (or flash RAM) into memory. This block of bytes is called the *bootloader*. Once the BIOS has copied the bootloader into memory, it jumps to the first instruction in the block. On some newer machines, the BIOS also checks that the bootloader has not been corrupted by a computer virus. (If a virus could change the bootloader and get the BIOS to jump to it, the virus would then be in control of the machine.) As a check, the bootloader is stored with a *cryptographic signature*, a specially designed function of the bytes in a file and a private cryptographic key that allows someone with the corresponding public key to verify that an authorized entity produced the file. It is computationally intractable for an attacker without the private key to create a different file with a valid signature. The BIOS checks that the bootloader code matches the signature, verifying its authenticity.

The bootloader in turn loads the kernel into memory and jumps to it. Again, the bootloader can check the cryptographic signature of the operating system to verify that it has not been corrupted by a virus. The kernel's executable image is usually stored in the file system. Thus, to find the bootloader, the BIOS needs to read a block of raw bytes from disk; the bootloader, in turn, needs to know how to read from the file system to find and read the operating system image.

When the kernel starts running, it can initialize its data structures, including setting up the interrupt vector table to point to the various interrupt, processor exception, and system call handlers. The kernel then starts the first process, typically the user login page. To run this process, the operating system reads the code for the login program from its disk location, and jumps to the first instruction in the program, using the start process procedure described above. The login process in turn can trap into the kernel using a system call whenever it needs the kernel's services, e.g., to render the login prompt on the screen. We discuss the system calls needed for processes to do useful work in Chapter 3.

2.10 Case Study: Virtual Machines

Some operating system kernels provide the abstraction of an entire virtual machine at user level. How do interrupts, processor exceptions, and system calls work in this context? To avoid confusion when discussing virtual machines, we need to recap some terminology introduced in Chapter 1. The operating system providing the virtual machine abstraction is called the *host operating system*. The operating system running inside the virtual machine is called the *guest operating system*.

The host operating system provides the illusion that the guest kernel is running on real hardware. For example, to provide a guest disk, the host kernel simulates a virtual disk as a file on the physical disk. To provide network access to the guest kernel, the host kernel simulates a virtual network using physical network packets. Likewise, the host kernel must manage memory to provide the illusion that the guest kernel is managing its own memory protection even though it is running with virtual addresses. We discuss address translation for virtual machines in more detail in Chapter 10.

How does the host kernel manage mode transfer between guest processes and the guest kernel? During boot, the host kernel initializes its interrupt vector table to point to its own interrupt handlers in host kernel memory. When the host kernel starts the virtual machine, the guest kernel starts running as if it is being booted:

1. The host loads the guest bootloader from the virtual disk and starts it running.
2. The guest bootloader loads the guest kernel from the virtual disk into memory and starts it running.
3. The guest kernel then initializes its interrupt vector table to point to the guest interrupt handlers.
4. The guest kernel loads a process from the virtual disk into guest memory.
5. To start a process, the guest kernel issues instructions to resume execution at user level, e.g., using iret on the x86. Since changing the privilege level is a privileged operation, this instruction traps into the host kernel.
6. The host kernel simulates the requested mode transfer as if the processor had directly executed it. It restores the program counter, stack pointer, and processor status word exactly as the guest operating system had intended. Note that the host kernel must protect itself from bugs in the guest operating system, and so it also must check the validity of the mode transfer — to ensure that the guest kernel is not surreptitiously attempting to get the host kernel to “switch" to an arbitrary point in the kernel code.

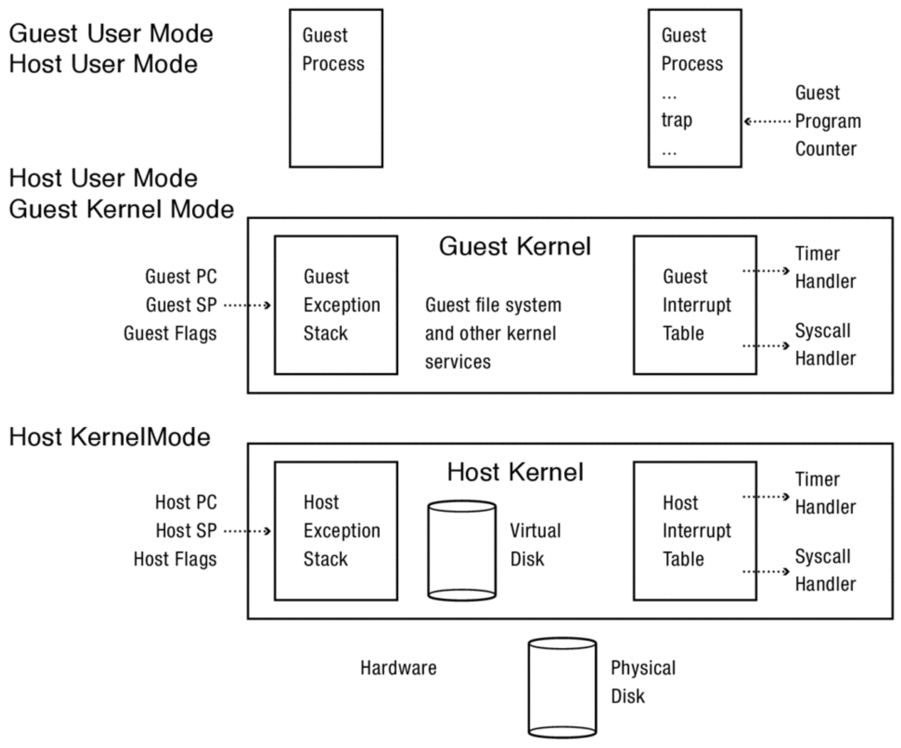


Figure 2.19: Emulation of user- and kernel-mode transfer for processes running inside a virtual machine. Both the guest kernel and the host kernel have their own copies of an interrupt vector table and interrupt stack. The guest vector table points to interrupt handlers in the guest kernel; the host vector table points to interrupt handlers in the host kernel.

Next, consider what happens when the guest user process does a system call, illustrated in Figure 2.19. To the hardware, there is only one kernel, the host operating system. Thus, the trap instruction traps into the host kernel's system call handler. Of course, the system call was not intended for the host! Rather, the host kernel simulates what would have happened had the system call instruction occurred on real hardware running the guest operating system:

1. The host kernel saves the instruction counter, processor status register, and user stack pointer on the interrupt stack of the guest operating system.
2. The host kernel transfers control to the guest kernel at the beginning of the interrupt handler, but with the guest kernel running with user-mode privilege.
3. The guest kernel performs the system call — saving user state and checking arguments.
4. When the guest kernel attempts to return from the system call back to user level, this causes a processor exception, dropping back into the host kernel.
5. The host kernel can then restore the state of the user process, running at user level, as if the guest OS had been able to return there directly.

The host kernel handles processor exceptions similarly, with one caveat. Some exceptions generated by the virtual machine are due to the user process; the host kernel forwards these to the guest kernel for handling. Other exceptions are generated by the guest kernel (e.g., when it tries to execute privileged instructions); the host kernel simulates these itself. Thus, the host kernel must track whether the virtual machine is executing in virtual user mode or virtual kernel mode.

The hardware vectors interrupts to the host kernel. Timer interrupts need special handling, as time can elapse in the host without elapsing in the guest. When a timer interrupt occurs, enough virtual time may have passed that the guest kernel is due for a timer interrupt. If so, the host kernel returns from the interrupt to the interrupt handler for the guest kernel. The guest kernel may in turn switch guest processes; its iret will cause a processor exception, returning to the host kernel, which can then resume the correct guest process.

Handling I/O interrupts is simpler: the simulation of the virtual device does not need to be anything like a real device. When the guest kernel makes a request to a virtual disk, the kernel writes instructions to the buffer descriptor ring for the disk device; the host kernel translates these instructions into operations on the virtual disk. The host kernel can simulate the disk request however it likes — e.g., through regular file reads and writes, copied into the guest kernel memory as if there was true DMA hardware. The guest kernel expects to receive an interrupt when the virtual disk completes its work; this can be triggered by the timer interrupt, but vectored to the guest disk interrupt handler instead of the guest timer interrupt handler.

Hardware support for operating systems

We have described a number of hardware mechanisms that support operating systems:

- **Privilege levels**, user and kernel.
- **Privileged instructions**: instructions available only in kernel mode.
- **Memory translation** prevents user programs from accessing kernel data structures and aids in memory management.
- **Processor exceptions** trap to the kernel on a privilege violation or other unexpected event.
- **Timer interrupts** return control to the kernel on time expiration.
- **Device interrupts** return control to the kernel to signal I/O completion.
- **Interprocessor interrupts** cause another processor to return control to the kernel.
- **Interrupt masking** prevents interrupts from being delivered at inopportune times.
- **System calls** trap to the kernel to perform a privileged action on behalf of a user program.
- **Return from interrupt**: switch from kernel mode to user mode, to a specific location in a user process.
- **Boot ROM**: code that loads startup routines from disk into memory.

To support threads, we will need one additional mechanism, described in Chapter 5:

- **Atomic read-modify-write instructions** used to implement synchronization in multi-threaded programs.

2.11 Summary and Future Directions

The process concept — the ability to execute arbitrary user programs with restricted rights — has been remarkably successful. With the exception of devices that run only a single application at a time (such as embedded systems and game consoles), every commercially successful operating system provides process isolation.

The reason for this success is obvious. Without process isolation, computer systems would be much more fragile and less secure. As recently as a decade ago, it was common for personal computers to crash on a daily basis. Today, laptops can remain working for weeks at a time without rebooting. This has occurred even though the operating system and application software on these systems have become more complex. While some of the improvement is due to better hardware reliability and automated bug tracking, process isolation has been a key technology in constructing more reliable system software.

Process isolation is also essential to building more secure computer systems. Without isolation, computer users would be forced to trust every application loaded onto the computer, not just the operating system code. In practice, however, complete process isolation remains more an aspiration than a reality. Most operating systems are vulnerable to malicious applications because the attacker can exploit any vulnerability in the kernel implementation. Even a single bug in the kernel can leave the system vulnerable. Keeping your system current with the latest patches provides some level of defense, but it is still inadvisable to download and install untrusted software off the web.

In the future, we are likely to see three complementary trends:

- **Operating system support for fine-grained protection.** Process isolation is becoming more flexible and fine-grained in order to reflect different levels of trust in different applications. Today, it is typical for a user application to have the permissions of that user. This allows a virus masquerading as a screen saver to steal or corrupt that user's data without needing to compromise the operating system first. Smartphone operating systems have started to add better controls to prevent certain applications without a "need to know" from accessing sensitive information, such as the smartphone's location or the list of frequently called telephone numbers.

- **Application-layer sandboxing.** Increasingly, many applications are becoming mini-operating systems in their own right, capable of safely executing third-party software to extend and improve the user experience. Sophisticated scripts customize web site behavior; web browsers must efficiently and completely isolate these scripts so that they cannot steal the user's data or corrupt the browser. Other applications — such as databases and desktop publishing systems — are also moving in the direction of needing application-layer sandboxing. Google's Native Client and Microsoft's Application Domains are two example systems that provide general-purpose, safe execution of third-party code at the user level.

- **Hardware support for virtualization.** Virtual machines provide an extra layer of protection beneath the operating system. Even if a malicious process run by a guest operating system on a virtual machine were able to corrupt the kernel, its impact would be limited to just that virtual machine. Below the virtual machine interface, the host operating system needs to provide isolation between different virtual machines; this is much easier in practice because the virtual machine interface is much simpler than the kernel's system call interface. For example, in a data center, virtual machines provide users with the flexibility to run any application without compromising data center operation.

  Computer manufacturers are re-designing processor architectures to reduce the cost of running a virtual machine. For example, on some new processors, guest operating systems can directly handle their own interrupts, processor exceptions, and system calls without those events needing to be mediated by the host operating system. Likewise, I/O device manufacturers are re-designing their interfaces to do direct transfers to and from the guest operating system without the need to go through the host kernel.

Exercises

1. When a user process is interrupted or causes a processor exception, the x86 hardware switches the stack pointer to a kernel stack, before saving the current process state. Explain why.

2. For the “Hello world” program, we mentioned that the kernel must copy the string from the user program to screen memory. Why must the screen’s buffer memory be protected? Explain what might happen if a malicious application could alter any pixel on the screen, not just those within its own window.

3. For each of the three mechanisms that supports dual-mode operation — privileged instructions, memory protection, and timer interrupts — explain what might go wrong without that mechanism, assuming the system still had the other two.

4. Suppose you are tasked with designing the security system for a new web browser that supports rendering web pages with embedded web page scripts. What checks would you need to implement to ensure that executing buggy or malicious scripts could not corrupt or crash the browser?

5. Define three types of user-mode to kernel-mode transfers.

6. Define four types of kernel-mode to user-mode transfers.

7. Most hardware architectures provide an instruction to return from an interrupt, such as iret. This instruction switches the mode of operation from kernel-mode to user-mode.

    a. Explain where in the operating system this instruction would be used.
    b. Explain what happens if an application program executes this instruction.

8. A hardware designer argues that there is now enough on-chip transistors to provide 1024 integer registers and 512 floating point registers. As a result, the compiler should almost never need to store anything on the stack. As an operating system guru, give your opinion of this design.

    a. What is the effect on the operating system of having a large number of registers?
    b. What hardware features would you recommend adding to the design?
    c. What happens if the hardware designer also wants to add a 16-stage pipeline into the CPU, with precise exceptions. How would that affect the user-kernel switching overhead?

9. With virtual machines, the host kernel runs in privileged mode to create a virtual machine that runs in user mode. The virtual machine provides the illusion that the guest kernel runs on its own machine in privileged mode, even though it is actually running in user mode.

    Early versions of the x86 architecture (pre-2006) were not *completely virtualizable* — these systems could not guarantee to run unmodified guest operating systems properly. One problem was the popf “pop flags” instruction that restores the processor status word. When popf was run in privileged mode, it changed both the ALU flags (e.g., the condition codes) and the systems flags (e.g., the interrupt mask). When popf was run in unprivileged mode, it changed just the ALU flags.

   a. Why do instructions like popf prevent transparent virtualization of the (old) x86 architecture?

   b. How would you change the (old) x86 hardware to fix this problem?

10. Which of the following components is responsible for loading the initial value in the program counter for an application program before it starts running: the compiler, the linker, the kernel, or the boot ROM?

11. We described how the operating system kernel mediates access to I/O devices for safety. Some newer I/O devices are *virtualizable* — they permit safe access from user-level programs, such as a guest operating system running in a virtual machine. Explain how you might design the hardware and software to get this to work. (Hint: The device needs much of the same hardware support as the operating system kernel.)

12. System calls vs. procedure calls: How much more expensive is a system call than a procedure call? Write a simple test program to compare the cost of a simple procedure call to a simple system call (getpid() is a good candidate on UNIX; see the man page). To prevent the optimizing compiler from “optimizing out" your procedure calls, do not compile with optimization on. You should use a system call such as the UNIX gettimeofday() for time measurements. Design your code so the measurement overhead is negligible. Also, be aware that timer values in some systems have limited resolution (e.g., millisecond resolution).

    Explain the difference (if any) between the time required by your simple procedure call and simple system call by discussing what work each call must do.

13. Suppose you have to implement an operating system on hardware that supports interrupts and exceptions but does not have a trap instruction. Can you devise a satisfactory substitute for traps using interrupts and/or exceptions? If so, explain how. If not, explain why.

14. Suppose you have to implement an operating system on hardware that supports exceptions and traps but does not have interrupts. Can you devise a satisfactory substitute for interrupts using exceptions and/or traps? If so, explain how. If not, explain why.

15. Explain the steps that an operating system goes through when the CPU receives an interrupt.

16. When an operating system receives a system call from a program, a switch to operating system code occurs with the help of the hardware. The hardware sets the mode of operation to kernel mode, calls the operating system trap handler at a location specified by the operating system, and lets the operating system return to user mode after it finishes its trap handling.

    Consider the stack on which the operating system must run when it receives the system call. Should this stack be different from the one the application uses, or could it use the same stack as the application program? Assume that the application program is blocked while the system call runs.

17. Write a program to verify that the operating system on your computer correctly protects itself from rogue system calls. For a single system call — such as file system open — try all possible illegal calls: e.g., an invalid system call number, an invalid stack pointer, an invalid pointer stored on the stack, etc. What happens?

3. The Programming Interface

From a programmer’s point of view, the user is a peripheral that types when you issue a read request. —*Peter Williams*

The previous chapter concerned the mechanisms needed in the operating system kernel to implement the process abstraction. A process is an instance of a program — the kernel provides an efficient sandbox for executing untrusted code at user-level, running user code directly on the processor.

This chapter concerns how we choose to use the process abstraction: what functionality does the operating system provide applications, and what should go where — what functionality should be put in the operating system kernel, what should be put into user-level libraries, and how should the operating system itself be organized?

There are as many answers to this as there are operating systems. Describing the full programming interface and internal organization for even a single operating system would take an entire book. Instead, in this chapter we explore a subset of the programming interface for UNIX, the foundation of Linux, MacOS, iOS, and Android. We also touch on how the same issues are addressed in Windows.

First, we need to answer “what” — what functions do we need an operating system to provide applications?

- **Process management.** Can a program create an instance of another program? Wait for it to complete? Stop or resume another running program? Send it an asynchronous event?

- **Input/output.** How do processes communicate with devices attached to the computer and through them to the physical world? Can processes communicate with each other?

- **Thread management.** Can we create multiple activities or threads that share memory or other resources within a process? Can we stop and start threads? How do we synchronize their use of shared data structures?

- **Memory management.** Can a process ask for more (or less) memory space? Can it share the same physical memory region with other processes?

- **File systems and storage.** How does a process store the user’s data persistently so that it can survive machine crashes and disk failures? How does the user name and organize their data?

- **Networking and distributed systems.** How do processes communicate with processes on other computers? How do processes on different computers coordinate their actions despite machine crashes and network problems?

- **Graphics and window management.** How does a process control pixels on its portion of the screen? How does a process make use of graphics accelerators?

- **Authentication and security.** What permissions does a user or a program have, and how are these permissions kept up to date? On what basis do we know the user (or program) is who they say they are?

In this chapter, we focus on just the first two of these topics: process management and input/output. We will cover thread management, memory management, and file systems in detail in later chapters in this book.

Remarkably, we can describe a functional interface for process management and input/output with just a dozen system calls, and the rest of the system call interface with another dozen. Even more remarkably, these calls are nearly unchanged from the original UNIX design. Despite being first designed and implemented in the early 1970's, most of these calls are still in wide use in systems today!

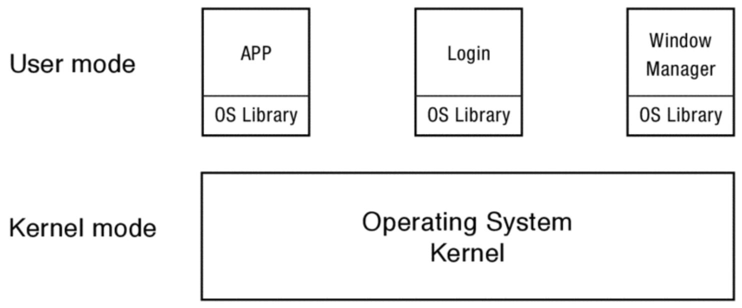


Figure 3.1: Operating system functionality can be implemented in user-level programs, in user-level libraries, in the kernel itself, or in a user-level server invoked by the kernel.

Second, we need to answer "where" — for any bit of functionality the operating system provides to user programs, we have several options for where it lives, illustrated in Figure 3.1:

- We can put the functionality in a user-level program. In both Windows and UNIX, for example, there is a user program for managing a user's login and another for managing a user's processes.

- We can put the functionality in a user-level library linked in with each application. In Windows and MacOS, user interface widgets are part of user-level libraries, included in those applications that need them.

- We can put the functionality in the operating system kernel, accessed through a system call. In Windows and UNIX, low-level process management, the file system and the network stack are all implemented in the kernel.

- We can access the function through a system call, but implement the function in a standalone server process invoked by the kernel. In many systems, the window manager is implemented as a separate server process.

How do we make this choice? It is important to realize that the choice can be (mostly) transparent to both the user and the application programmer. The user wants a system that works; the programmer wants a clean, convenient interface that does the job. As long as the operating system provides that interface, where each function is implemented is up to the operating system, based on a tradeoff between flexibility, reliability, performance, and safety.

- **Flexibility.** It is much easier to change operating system code that lives outside of the kernel, without breaking applications using the old interface. If we create a new version of a library, we can just link that library in with new applications, and over time convert old applications to use the new interface. However, if we need to change the system call interface, we must either simultaneously change both the kernel and all applications, or we must continue to support both the old and the new versions until all old applications have been converted. Many applications are written by third party developers, outside of the control of the operating system vendor. Thus, changing the system call interface is a huge step, often requiring coordination across many companies.

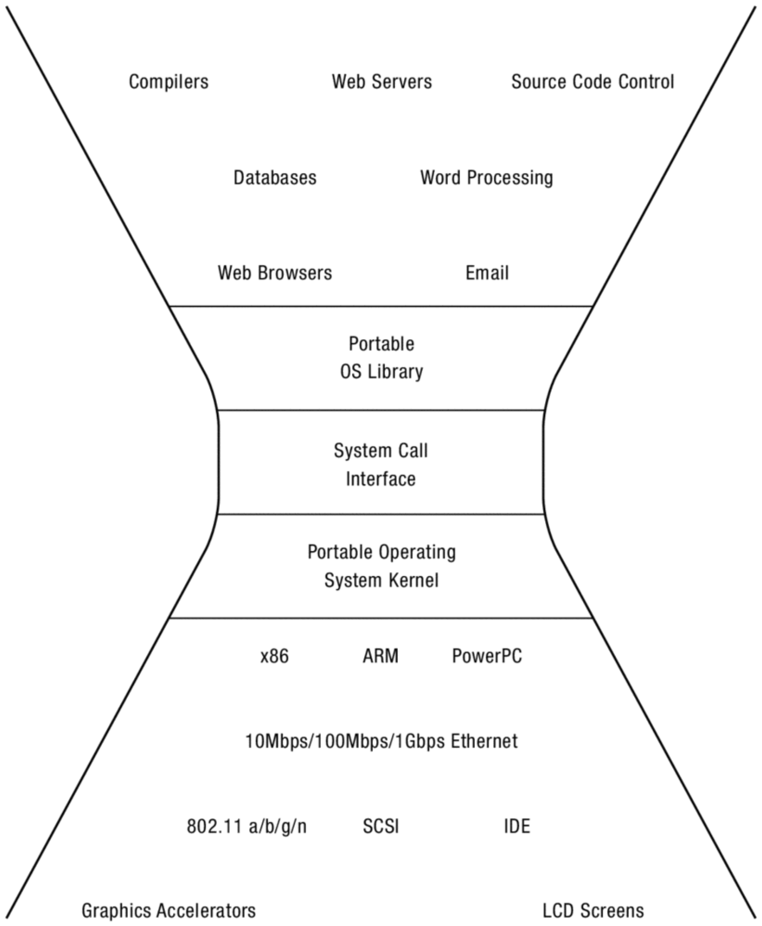


Figure 3.2: The kernel system call interface can be seen as a “thin waist,” enabling independent evolution of applications and hardware.

One of the key ideas in UNIX, responsible for much of its success, was to design its system call interface to be simple and powerful, so that almost all of the innovation in the system could happen in user code without changing the interface to the operating system. The UNIX system call interface is also highly portable — the operating system can be ported to new hardware without needing to rewrite application code. As shown in Figure 3.2, the kernel can be seen as a “thin waist,” enabling innovation at the application-level, and in the hardware, without requiring simultaneous changes in the other parts of the system.

The Internet and the “thin waist”

The Internet is another example of the benefit of designing interfaces to be simple and portable. The Internet defines a packet-level protocol that can run on top of virtually any type of network hardware and can support almost any type of network application. Creating the World Wide Web required no changes to the Internet packet delivery mechanism; likewise, the introduction of wireless networks required changes in hardware devices and in the operating system, but no changes in network applications. Although the Internet's "thin waist" can sometimes lead to inefficiencies, the upside is to foster innovation in both applications and hardware by decoupling changes in one from changes in the other.

- **Safety.** However, resource management and protection are the responsibility of the operating system kernel. As Chapter 2 explained, protection checks cannot be implemented in a user-level library because application code can skip any checks made by the library.
- **Reliability.** Improved reliability is another reason to keep the operating system kernel minimal. Kernel code needs the power to set up hardware devices, such as the disk, and to control protection boundaries between applications. However, kernel modules are typically not protected from one another, and so a bug in kernel code (whether sensitive or not) may corrupt user or kernel data. This has led some systems to use a philosophy of "what can be at user level, should be." An extreme version of approach is to isolate privileged, but less critical, parts of the operating system such as the file system or the window system, from the rest of the kernel. This is called a *microkernel* design. In a microkernel, the kernel itself is kept small, and instead most of the functionality of a traditional operating system kernel is put into a set of user-level processes, or servers, accessed from user applications via interprocess communication.
- **Performance.** Finally, transferring control into the kernel is more expensive than a procedure call to a library, and transferring control to a user-level file system server via the kernel is still even more costly. Hardware designers have attempted to reduce the cost of these boundary crossings, but their performance remains a problem. Microsoft Windows NT, a precursor to Windows 7, was initially designed as a microkernel, but over time, much of its functionality has been migrated back into the kernel for performance reasons.

Application-level sandboxing and operating system functionality

Applications that support executing third-party code or scripts in a restricted sandbox must address many of these same questions, with the sandbox playing the role of the operating system kernel. In terms of functionality: Can the scripting code start a new instance of itself? Can it do input/output? Can it perform work in the background? Can it store data persistently, and if it can, how does it name that data? Can it communicate data over the network? How does it authenticate actions?

For example, in web browsers, HTML5 not only allows scripts to draw on the screen, communicate with servers, and save and read cookies, it also has recently added programming interfaces for offline storage and cross-document communication. The Flash media player provides scripts with the ability to do asynchronous operations, file storage, network communication, memory management, and authentication.

Just as with system calls, these interfaces must be carefully designed to be bulletproof against malicious use. A decade ago, email viruses became widespread because scripts could be embedded in documents that were executed on opening; the programming interfaces for these scripts would allow them to discover the list of correspondents known to the current email user and to send them email, thereby propagating and expanding the virus with a single click. The more fully featured the interface, the more convenient it is for developers, and the more likely that some aspect of the interface will be abused by a hacker.

There are no easy answers! We will investigate the question of how to design the system call interface and where to place operating system functionality through case studies of UNIX and other systems.

Chapter roadmap:

- **Process management.** What is the system call interface for process management? (Section 3.1)
- **Input/output.** What is the system call interface for performing I/O and interprocess communication? (Section 3.2)
- **Case study: Implementing a shell.** We will illustrate these interfaces by using them to implement a user-level job control system called a *shell*. (Section 3.3)
- **Case study: Interprocess communication.** How does the communication between a client and server work? (Section 3.4)
- **Operating system structure.** Can we use the process abstraction to simplify the construction of the operating system itself and to make it more secure, more reliable, and more flexible? (Section 3.5)

3.1 Process Management

On a modern computer, when a user clicks on a file or application icon, the application starts up. How does this happen and who is called? Of course, we could implement everything that needs to happen in the kernel — draw the icon for every item in the file system, map mouse positions to the intended icon, catch the mouse click, and start the process. In early batch processing systems, the kernel was in control by necessity. Users submitted jobs, and the operating system took it from there, instantiating the process when it was time to run the job.

A different approach is to allow user programs to create and manage their own processes. This has fostered a blizzard of innovation. Today, programs that create and manage processes include window managers, web servers, web browsers, shell command line interpreters, source code control systems, databases, compilers, and document preparation systems. We could go on, but you get the idea. If creating a process is something a process can do, then anyone can build a new version of any of these applications, without recompiling the kernel or forcing anyone else to use it.

An early motivation for user-level process management was to allow developers to write their own shell command line interpreters. A *shell* is a job control system; both Windows and UNIX have a shell. Many tasks involve a sequence of steps to do something, each of which can be its own program. With a shell, you can write down the sequence of steps, as a sequence of programs to run to do each step. Thus, you can view it as a very early version of a scripting system.

For example, to compile a C program from multiple source files, you might type:

```
cc -c sourcefile1.c
cc -c sourcefile2.c
ln -o program sourcefile1.o sourcefile2.o
```

If we put those commands into a file, the shell reads the file and executes it, creating, in turn, a process to compile sourcefile1.c, a process to compile sourcefile2, and a process to link them together. Once a shell script is a program, we can create other programs by combining scripts together. In fact, on UNIX, the C compiler is itself a shell program! The compiler first invokes a process to expand header include files, then a separate process to parse the output, another process to generate (text) assembly code, and yet another to convert assembly into executable machine instructions.

There is an app for that

User-level process management is another way of saying “there is an app for that.” Instead of a single program that does everything, we can create specialized programs for each task, and mix and match what we need. The formatting system for this textbook uses over fifty separate programs.

The web is a good example of the power of composing complex applications from more specialized services. A web page does not need to do everything itself: it can mash up the results of many different web pages, and it can invoke process creation on the local server to generate part of the page. The flexibility to create processes was extremely important early on in the development of the web. HTML was initially just a way to describe the formatting for static information, but it included a way to escape to a process, e.g., to do a lookup in a database or to authenticate a user. Over time, HTML has added support for many different features that were first prototyped via execution by a separate process. And of course, HTML can still execute a process for any format not supported by the standard.

3.1.1 Windows Process Management

One approach to process management is to just add a system call to create a process, and other system calls for other process operations. This turns out to be simple in theory and complex in practice. In Windows, there is a routine called, unsurprisingly, CreateProcess, in simplified form below:

```
boolean CreateProcess(char *prog, char *args);
```

We call the process creator the *parent* and the process being created, the *child*.

What steps does CreateProcess take? As we explained in the previous chapter, the kernel needs to:

- Create and initialize the process control block (PCB) in the kernel.
- Create and initialize a new address space.
- Load the program prog into the address space.
- Copy arguments args into memory in the address space.
- Initialize the hardware context to start execution at “start”.
- Inform the scheduler that the new process is ready to run.

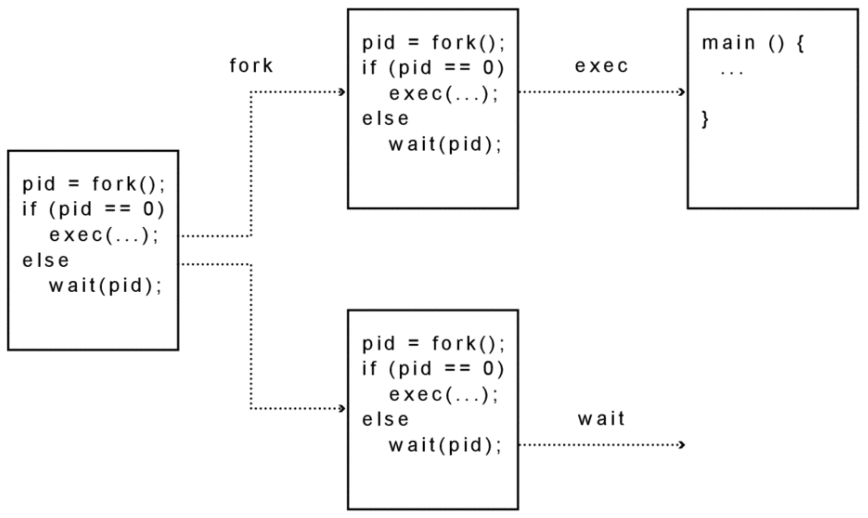
Unfortunately, there are quite a few aspects of the process that the parent might like to control, such as: its privileges, where it sends its input and output, what it should store its files, what to use as a scheduling priority, and so forth. We cannot trust the child process itself to set its own privileges and priority, and it would be inconvenient to expect every application to include code for figuring out its context. So the real interface to CreateProcess is quite a bit more complicated in practice, given in Figure 3.3.

```
// Start the child process
if (!CreateProcess(NULL,   // No module name (use command line)
    argv[1],        // Command line
    NULL,           // Process handle not inheritable
    NULL,           // Thread handle not inheritable
    FALSE,          // Set handle inheritance to FALSE
    0,              // No creation flags
    NULL,           // Use parent’s environment block
    NULL,           // Use parent’s starting directory
    &si,            // Pointer to STARTUPINFO structure
    &pi )           // Pointer to PROCESS_INFORMATION structure
)
```

Figure 3.3: Excerpt from an example of how to use the Windows CreateProcess system call. The first two arguments specify the program and its arguments; the rest concern aspects of the process runtime environment.

3.1.2 UNIX Process Management

UNIX takes a different approach to process management, one that is complex in theory and simple in practice. UNIX splits CreateProcess in two steps, called fork and exec, illustrated in Figure 3.4.



```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

Figure 3.4: The operation of the UNIX fork and exec system calls. UNIX fork makes a copy of the parent process; UNIX exec changes the child process to run the new program.

UNIX fork creates a complete copy of the parent process, with one key exception. (We need some way to distinguish between which copy is the parent and which is the child.) The child process sets up privileges, priorities, and I/O for the program that is about to be started, e.g., by closing some files, opening others, reducing its priority if it is to run in the background, etc. Because the child runs exactly the same code as the parent, it can be trusted to set up the context for the new program correctly.

Once the context is set, the child process calls *UNIX exec*. UNIX exec brings the new executable image into memory and starts it running. It may seem wasteful to make a complete copy of the parent process, just to overwrite that copy when we bring in the new executable image into memory using exec. It turns out that fork and exec can be implemented efficiently, using a set of techniques we will describe in Chapter 8.

With this design, UNIX fork takes no arguments and returns an integer. UNIX exec takes two arguments (the name of the program to run and an array of arguments to pass to the program). This is in place of the ten parameters needed for CreateProcess. In part because of the simplicity of UNIX fork and exec, this interface has remained nearly unchanged since UNIX was designed in the early 70’s. (Although the interface has not changed, the word fork is now a bit ambiguous. It is used for creating a new copy of a UNIX process, and in thread systems for creating a new thread. To disambiguate, we will always use the term “UNIX fork” to refer to UNIX’s copy process system call.)

UNIX fork

The steps for implementing UNIX fork in the kernel are:

- Create and initialize the process control block (PCB) in the kernel
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run

A strange aspect of UNIX fork is that the system call returns *twice*: once to the parent and once to the child. To the parent, UNIX returns the process ID of the child; to the child, it returns zero indicating success. Just as if you made a clone of yourself, you would need some way to tell who was the clone and who was the original, UNIX uses the return value from fork to distinguish the two copies. Some sample code to call fork is given in Figure 3.5.

```
int child_pid = fork();

if (child_pid == 0) {    // I’m the child process.
    printf("I am process #%d\n", getpid());
    return 0;
} else {                 // I’m the parent process.
    printf("I am the parent of process #%d\n", child_pid);
    return 0;
}

Possible output:
    I am the parent of process 495
    I am process 495

Another less likely but still possible output:
    I am process 456
    I am the parent of process 456
```

Figure 3.5: Example UNIX code to fork a process, and some possible outputs of running the code. getpid is a system call to get the current process’s ID.

If we run the program in Figure 3.5, what happens? If you have access to a UNIX system, you can try it and see for yourself. UNIX fork returns twice, once in the child, with a return value of zero, and once in the parent with a return value of the child’s process ID. However, we do not know whether the parent will run next or the child. The parent had been running, and so it is likely that it will reach its print statement first. However, a timer interrupt could intervene between when the parent forks the process and when it reaches the print statement, so that the processor is reassigned to the child. Or we could be running on a multicore system, where both the parent and child are running simultaneously. In either case, the child could print its output before the parent. We will talk in much more depth about the implications of different orderings of concurrent execution in the next chapter.

UNIX fork and the Chrome Web browser

Although UNIX fork is normally paired with a call to exec, in some cases UNIX fork is useful on its own. A particularly interesting example is in Google's Chrome web browser. When the user clicks on a link, Chrome forks a process to fetch and render the web page at the link, in a new tab on the browser. The parent process continues to display the original referring web page, while the child process runs the same browser, but in its own address space and protection boundary. The motivation for this design is to isolate the new link, so that if the web site is infected with a virus, it will not infect the rest of the browser. Closing the infected browser tab will then remove the link and the virus from the system.

Some security researchers take this a step further. They set up their browsers and email systems to create a new *virtual machine* for every new link, running a copy of the browser in each virtual machine; even if the web site has a virus that corrupts the guest operating system running in the virtual machine, the rest of the system will remain unaffected. In this case, closing the virtual machine cleans the system of the virus.

Interestingly, on Windows, Google Chrome does not use CreateProcess to fork new copies of the browser on demand. The difficulty is that if Chrome is updated while Chrome is running, CreateProcess will create a copy of the new version, and that may not interoperate correctly with the old version. Instead, they create a pool of helper processes that wait in the background for new links to render.

UNIX exec and wait

The UNIX system call exec completes the steps needed to start running a new program. The child process typically calls UNIX exec once it has returned from UNIX fork and configured the execution environment for the new process. We will describe more about how this works when we discuss UNIX pipes in the next section.

UNIX exec does the following steps:

- Load the program prog into the current address space.
- Copy arguments args into memory in the address space.
- Initialize the hardware context to start execution at "start."

Note that exec does not create a new process!

On the other side, often the parent process needs to pause until the child process completes, e.g., if the next step depends on the output of the previous step. In the shell example we started the chapter with, we need to wait for the two compilations to finish before it is safe to start the linker.

UNIX has a system call, naturally enough called *wait*, that pauses the parent until the child finishes, crashes, or is terminated. Since the parent could have created many child processes, wait is parameterized with the process ID of the child. With wait, a shell can create a new process to perform some step of its instructions, and then pause for that step to complete before proceeding to the next step. It would be hard to build a usable shell without wait.

However, the call to wait is optional in UNIX. For example, the Chrome browser does not need to wait for its forked clones to finish. Likewise, most UNIX shells have an option to run operations in the background, signified by appending ‘&’ to the command line. (As with fork, the word wait is now a bit ambiguous. It is used for pausing the current UNIX process to wait for another process to complete; it is also used in thread synchronization, for waiting on a condition variable. To disambiguate, we will always use the term “UNIX wait” to refer to UNIX’s wait system call. Oddly, waiting for a thread to complete is called “thread join”, even though it is most analogous to UNIX wait. Windows is simpler, with a single function called “WaitForSingleObject” that can wait for process completion, thread completion, or on a condition variable.)

Kernel handles and garbage collection

As we discussed in the previous chapter, when a UNIX process finishes, it calls the system call exit. Exit can release various resources associated with the process, such as the user stack, heap, and code segments. It must be careful, however, in how it garbage collects the process control block (PCB). Even though the child process has finished, if it deletes the PCB, then the parent process will be left with a dangling pointer if later on it calls UNIX wait. Of course, we don’t know for sure if the parent will ever call wait, so to be safe, the PCB can only be reclaimed when both the parent and the child have finished or crashed.

Generalizing, both Windows and UNIX have various system calls that return a handle to some kernel object; these handles are used in later calls as an ID. The process ID returned by UNIX fork is used in later calls to UNIX wait; we will see below that UNIX open returns a file descriptor that is used in other system calls. It is important to realize that these handles are *not* pointers to kernel data structures; otherwise, an erroneous user program could cause havoc in the kernel by making system calls with fake handles. Rather, they are specific to the process and checked for validity on each use.

Further, in both Windows and UNIX, handles are reference counted. Whenever the kernel returns a handle, it bumps a reference counter, and whenever the process releases a handle (or exits), the reference counter is decremented. UNIX fork sets the process ID reference count to two, one for the parent and one for the child. The underlying data structure, the PCB, is reclaimed only when the reference count goes to zero, that is, when both the parent and child terminate.

Finally, as we outlined in the previous chapter, UNIX provides a facility for one process to send another an instant notification, or upcall. In UNIX, the notification is sent by calling *signal*. Signals are used for terminating an application, suspending it temporarily for debugging, resuming after a suspension, timer expiration, and a host of other reasons. In the default case, where the receiving application did not specify a signal handler, the kernel implements a standard one on its behalf.

3.2 Input/Output

Computer systems have a wide diversity of input and output devices: keyboard, mouse, disk, USB port, Ethernet, WiFi, display, hardware timer, microphone, camera, accelerometer, and GPS, to name a few.

To deal with this diversity, we could specialize the application programming interface for each device, customizing it to the device’s specific characteristics. After all, a disk device is quite different from a network and both are quite different from a keyboard: a disk is addressed in fixed sized chunks, while a network sends and receives a stream of variable sized packets, and the keyboard returns individual characters as keys are pressed. While the disk only returns data when asked, the network and keyboard provide data unprompted. Early computer systems took the approach of specializing the interface to the device, but it had a significant downside: every time a new type of hardware device is invented, the system call interface has to be upgraded to handle that device.

One of the primary innovations in UNIX was to regularize all device input and output behind a single common interface. In fact, UNIX took this one giant step further: it uses this same interface for reading and writing files and for interprocess communication. This approach was so successful that it is almost universally followed in systems today. We will sketch the interface in this section, and then in the next section, show how to use it to build a shell.

The basic ideas in the UNIX I/O interface are:

- **Uniformity.** All device I/O, file operations, and interprocess communication use the same set of system calls: open, close, read and write.

- **Open before use.** Before an application does I/O, it must first call open on the device, file, or communication channel. This gives the operating system a chance to check access permissions and to set up any internal bookkeeping. Some devices, such as a printer, only allow one application access at a time — the open call can return an error if the device is in use.

  Open returns a handle to be used in later calls to read, write and close to identify the file, device or channel; this handle is somewhat misleadingly called a *file descriptor*, even when it refers to a device or channel so there is no file involved. For convenience, the UNIX shell starts applications with open file descriptors for reading and writing to the terminal.

- **Byte-oriented.** All devices, even those that transfer fixed-size blocks of data, are accessed with byte arrays. Similarly, file and communication channel access is in terms of bytes, even though we store data structures in files and send data structures across channels.

- **Kernel-buffered reads.** Stream data, such as from the network or keyboard, is stored in a kernel buffer and returned to the application on request. This allows the UNIX system call read interface to be the same for devices with streaming reads as those with block reads, such as disks and Flash memory. In both cases, if no data is available to be returned immediately, the read call blocks until it arrives, potentially giving up the processor to some other task with work to do.

- **Kernel-buffered writes.** Likewise, outgoing data is stored in a kernel buffer for transmission when the device becomes available. In the normal case, the system call write copies the data into the kernel buffer and returns immediately. This decouples the application from the device, allowing each to go at its own speed. If the application generates data faster than the device can receive it (as is common when spooling data to a printer), the write system call blocks in the kernel until there is enough room to store the new data in the buffer.

- **Explicit close.** When an application is done with the device or file, it calls close. This signals to the operating system that it can decrement the reference-count on the device, and garbage collect any unused kernel data structures.

Open vs. creat vs. stat

By default, the UNIX open system call returns an error if the application tries to open a file that does not exist; as an option (not shown above), a parameter can tell the kernel to instead create the file if it does not exist. Since UNIX also has system calls for creating a file (creat) and for testing whether a file exists (stat), it might seem as if open could be simplified to always assume that the file already exists.

However, UNIX often runs in a multi-user, multi-application environment, and in that setting the issue of system call design can become more subtle. Suppose instead of the UNIX interface, we had completely separate functions for testing if a file exists, creating a file, and opening the file. Assuming that the user has permission to test, open, or create the file, does this code work?

```
if (!exists(file)) {    // If the file doesn’t exist create it.
// Are we guaranteed the file doesn’t exist?
    create(file);
}
// Are we guaranteed the file does exist?
open(file);
```

The problem is that on a multi-user system, some other user might have created the file in between the call to test for its existence, and the call to create the file. Thus, call to create must also test the existence of the file. Likewise, some other user might have deleted the file between the call to create and the call to open. So open also needs the ability to test if the file is there, and if not to create the file (if that is the user’s intent).

UNIX addresses this with an all-purpose, atomic open: test if the file exists, optionally create it if it does not, and then open it. Because system calls are implemented in the kernel, the operating system can make open (and all other I/O systems calls) non-interruptible with respect to other system calls. If another user tries to delete a file while the kernel is executing an open system call on the same file, the delete will be delayed until the open completes. The open will return a file descriptor that will continue to work until the application closes the file. The delete will remove the file from the file system, but the file system does not actually reclaim its disk blocks until the file is closed.

For interprocess communication, we need a few more concepts:

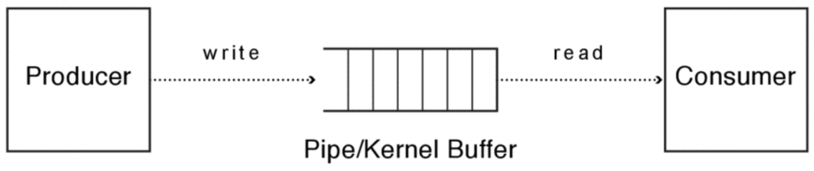


Figure 3.6: A pipe is a temporary kernel buffer connecting a process producing data with a process consuming the data.

- **Pipes.** A *UNIX pipe* is a kernel buffer with two file descriptors, one for writing (to put data into the pipe) and one for reading (to pull data out of the pipe), as illustrated in Figure 3.6. Data is read in exactly the same sequence it is written, but since the data is buffered, the execution of the producer and consumer can be decoupled, reducing waiting in the common case. The pipe terminates when either endpoint closes the pipe or exits.

  The Internet has a similar facility to UNIX pipes called TCP (Transmission Control Protocol). Where UNIX pipes connect processes on the same machine, TCP provides a bi-directional pipe between two processes running on different machines. In TCP, data is written as a sequence of bytes on one machine and read out as the same sequence on the other machine.

- **Replace file descriptor.** By manipulating the file descriptors of the child process, the shell can cause the child to read its input from, or send its output to, a file or a pipe instead of from a keyboard or to the screen. This way, the child process does not need to be aware of who is providing or consuming its I/O. The shell does this redirection using a special system call named dup2(from, to) that replaces the to file descriptor with a copy of the from file descriptor.

- **Wait for multiple reads.** For client-server computing, a server may have a pipe open to multiple client processes. Normally, read will block if there is no data to be read, and it would be inefficient for the server to poll each pipe in turn to check if there is work for it to do. The UNIX system call select(fd[], number) addresses this. Select allows the server to wait for input from any of a set of file descriptors; it returns the file descriptor that has data, but it does not read the data. Windows has an equivalent function, called WaitForMultipleObjects.

Figure 3.7 summarizes the dozen UNIX system calls discussed in this section.

	Creating and managing processes
.	
fork ()	Create a child process as a clone of the current process. The fork call returns to both the parent and child.
exec (prog, args)	Run the application prog in the current process.
exit ()	Tell the kernel the current process is complete, and its data structures should be garbage collected.
wait (processID)	Pause until the child process has exited.
signal (processID, type)	Send an interrupt of a specified type to a process.
	I/O operations
fileDesc open (name)	Open a file, channel, or hardware device, specified by name; returns a file descriptor that can be used by other calls.
pipe (fileDesc[2])	Create a one-directional pipe for communication between two processes. Pipe returns two file descriptors, one for reading and one for writing.

dup2 (fromFileDesc, toFileDesc)	Replace the toFileDesc file descriptor with a copy of fromFileDesc. Used for replacing stdin or stdout or both in a child process before calling exec.
int read (fileDesc, buffer, size)	Read up to size bytes into buffer, from the file, channel, or device. Read returns the number of bytes actually read. For streaming devices this will often be less than size. For example, a read from the keyboard device will (normally) return all of its queued bytes.
int write (fileDesc, buffer, size)	Analogous to read, write up to size bytes into kernel output buffer for a file, channel, or device. Write normally returns immediately but may stall if there is no space in the kernel buffer.
fileDesc select (fileDesc[], arraySize)	Return when any of the file descriptors in the array fileDesc[] have data available to be read. Returns the file descriptor that has data pending.
close (fileDescriptor)	Tell the kernel the process is done with this file, channel, or device.

Figure 3.7: List of UNIX system calls discussed in this section.

3.3 Case Study: Implementing a Shell

The dozen UNIX system calls listed in Figure 3.7 are enough to build a flexible and powerful command line shell, one that runs entirely at user-level with no special permissions. As we mentioned, the process that creates the shell is responsible for providing it an open file descriptor for reading commands for its input (e.g., from the keyboard), called *stdin* and for writing output (e.g., to the display), called *stdout*.

```
main() {
    char *prog = NULL;
    char **args = NULL;

    // Read the input a line at a time, and parse each line into the program
    // name and its arguments. End loop if we've reached the end of the input.
    while (readAndParseCmdLine(&prog, &args)) {

        // Create a child process to run the command.
        int child_pid = fork();

        if (child_pid == 0) {
            // I'm the child process.
            // Run program with the parent's input and output.
            exec(prog, args);
            // NOT REACHED
        } else {
            // I'm the parent; wait for the child to complete.
            wait(child_pid);
            return 0;
        }
    }
}
```

Figure 3.8: Example code for a simple UNIX shell.

Figure 3.8 illustrates the code for the basic operation of a shell. The shell reads a command line from the input, and it forks a process to execute that command. UNIX fork automatically duplicates all open file descriptors in the parent, incrementing the kernel's reference counts for those descriptors, so the input and output of the child is the same as the parent. The parent waits for the child to finish before it reads the next command to execute.

Because the commands to read and write to an open file descriptor are the same whether the file descriptor represents a keyboard, screen, file, device, or pipe, UNIX programs do not need to be aware of where their input is coming from, or where their output is going. This is helpful in a number of ways:

- **A program can be a file of commands.** Programs are normally a set of machine instructions, but on UNIX a program can be a file containing a list of commands for a shell to interpret. To disambiguate, shell programs signified in UNIX by putting "#! interpreter" as the first line of the file, where "interpreter" is the name of the shell executable.

  The UNIX C compiler works this way. When it is exec'ed, the kernel recognizes it as a shell file and starts the interpreter, passing it the file as input. The shell reads the file as a list of commands to invoke the pre-processor, parser, code generator and assembler in turn, exactly as if it was reading text input from the keyboard. When the last command completes, the shell interpreter calls exit to inform the kernel that the program is done.

- **A program can send its output to a file.** By changing the stdout file descriptor in the child, the shell can redirect the child's output to a file. In the standard UNIX shell, this is signified with a "greater than" symbol. Thus, "ls > tmp" lists the contents of the current directory into the file "tmp." After the fork and before the exec, the shell can replace the stdout file descriptor for the child using dup2. Because the parent has been cloned, changing stdout for the child has no effect on the parent.

- **A program can read its input from a file.** Likewise, by using dup2 to change the stdin file descriptor, the shell can cause the child to read its input from a file. In the standard UNIX shell, this is signified with a "less than" symbol. Thus, "zork < solution" plays the game "zork" with a list of instructions stored in the file "solution."

- **The output of one program can be the input to another program.** The shell can use a pipe to connect two programs together, so that the output of one is the input of another. This is called a *producer-consumer* relationship. For example, in the C-compiler, the output of the preprocessor is sent to the parser, and the output of the parser is sent to the code-generator and then to the assembler. In the standard UNIX shell, a pipe connecting two programs is signified by a "|" symbol, as in: "cpp file.c | cparse | cgen | as > file.o". In this case the shell creates four separate child processes, each connected by pipes to its predecessor and successor. Each of the phases can run in parallel, with the parent waiting for all of them to finish.

3.4 Case Study: Interprocess Communication

For many of the same reasons it makes sense to construct complex applications from simpler modules, it often makes sense to create applications that can specialize on a specific task, and then combine those applications into more complex structures. We gave an example above with the C compiler, but many parts of the operating system are structured this way. For example, instead of every program needing to know how to coordinate access to a printer, UNIX has a printer server, a specialized program for managing the printer queue.

For this to work, we need a way for processes to communicate with each other. Three widely used forms of interprocess communication are:

- **Producer-consumer.** In this model, programs are structured to accept as input the output of other programs. Communication is one-way: the producer only writes, and the consumer only reads. As we explained above, this allows chaining: a consumer can be, in turn, a producer for a different process. Much of the success of UNIX was due to its ability to easily compose many different programs together in this fashion.

- **Client-server.** An alternative model is to allow two-way communication between processes, as in client-server computing. The server implements some specialized task, such as managing the printer queue or managing the display. Clients send requests to the server to do some task, and when the operation is complete, the server replies back to the client.

- **File system.** Another way programs can be connected together is through reading and writing files. A text editor can import an image created by a drawing program, and the editor can in turn write an HTML file that a web server can read to know how to display a web page. A key distinction is that, unlike the first two modes, communication through the file system can be separated in *time*: the writer of the file does not need to be running at the same time as the file reader. Therefore, data needs to be stored persistently on disk or other stable storage, and the data needs to be named so that you can find the file when needed later on.

All three models are widely used both on a single system and over a network. For example, the Google MapReduce utility operates over a network in a producer-consumer fashion: the output of the map function is sent to the machines running the reduce function. The web is an example of client-server computing, and many enterprises and universities run centralized file servers to connect a text editor on one computer with a compiler running on another.

As persistent storage, file naming, and distributed computing are each complex topics in their own right, we defer the discussions of those topics to later chapters. Here we focus on interprocess communication, where both processes are running simultaneously on the same machine.

3.4.1 Producer-Consumer Communication

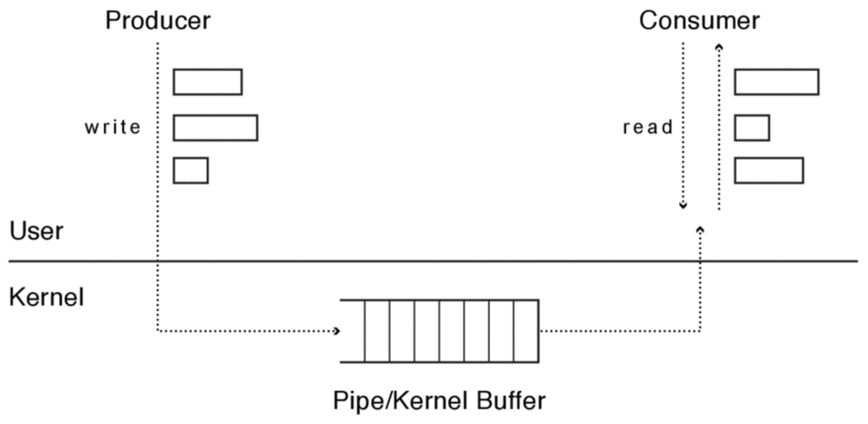


Figure 3.9: Interprocess communication between a producer application and a consumer. The producer uses the write system call to put data into the buffer; the consumer uses the read system call to take data out of the buffer.

Figure 3.9 illustrates how two processes communicate through the operating system in a producer-consumer relationship. Via the shell, we establish a pipe between the producer and the consumer. As one process computes and produces a stream of output data, it issues a sequence of write system calls on the pipe into the kernel. Each write can be of variable size. Assuming there is room in the kernel buffer, the kernel copies the data into the buffer, and returns immediately back to the producer.

At some point later, the operating system will schedule the consumer process to run. (On a multicore system, the producer and consumer could be running at the same time.) The consumer issues a sequence of read calls. Because the pipe is just a stream of bytes, the consumer can read the data out in any convenient chunking — the consumer can read in 1 KB chunks, while the producer wrote its data in 4 KB chunks, or vice versa. Each system call read made by the consumer returns the next successive chunk of data out of the kernel buffer. The consumer process can then compute on its input, sending its output to the display, a file, or onto the next consumer.

The kernel buffer allows each process to run at its own pace. There is no requirement that each process have equivalent amounts of work to do. If the producer is faster than the consumer, the kernel buffer fills up, and when the producer tries to write to a full buffer, the kernel stalls the process until there is room to store the data. Equivalently, if the consumer is faster than the producer, the buffer will empty and the next read request will stall until the producer creates more data.

In UNIX, when the producer finishes, it closes its side of the pipe, but there may still be data queued in the kernel for the consumer. Eventually, the consumer reads the last of the data, and the read system call will return an “end of file” marker. Thus, to the consumer, there is no difference between reading from a pipe and reading from a file.

Using kernel buffers to decouple the execution of the producer and consumer reduces the number and cost of context switches. Modern computers make extensive use of hardware caches to improve performance, but caches are ineffective if a program only runs for a short period of time before it must yield the processor to another task. The kernel buffer allows the operating system to run each process long enough to benefit from reuse, rather than alternating between the producer and consumer on each system call.

3.4.2 Client-Server Communication

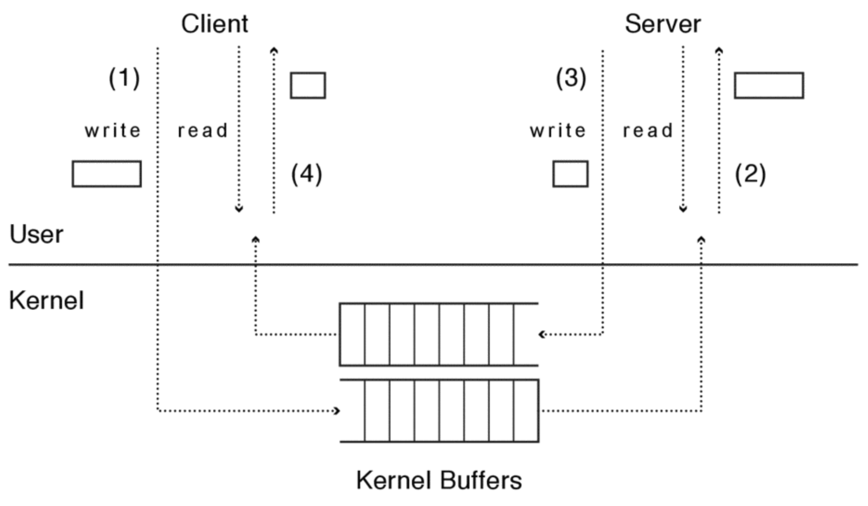


Figure 3.10: Interprocess communication between a client process and a server process. Once the client and server are connected, the client sends a request to the server by writing it into a kernel buffer. The server reads the request out of the buffer, and returns the result by writing it into a separate buffer read by the client.

We can generalize the above to illustrate client-server communication, shown in Figure 3.10. Instead of a single pipe, we create two, one for each direction. To make a request, the client writes the data into one pipe, and reads the response from the other. The server does the opposite: it reads requests from the first pipe, performs whatever is requested (provided the client has permission to make the request), and writes the response onto the second pipe.

The client and server code are shown in Figure 3.11. To simplify the code, we assume that the requests and responses are fixed-size.

```
Client:
    char request[RequestSize];
    char reply[ReplySize]

    // ..compute..

    // Put the request into the buffer.
    // Send the buffer to the server.
    write(output, request, RequestSize);

    // Wait for response.
    read(input, reply, ReplySize);

    // ..compute..

Server:
    char request[RequestSize];
    char reply[ReplySize];

    // Loop waiting for requests.
    while (1) {
        // Read incoming command.
        read(input, request, RequestSize);

        // Do operation.

        // Send result.
        write(output, reply, ReplySize);
    }
```

Figure 3.11: Example code for client-server interaction.

Streamlining client-server communication

Client-server communication is a common pattern in many systems, and so one can ask: how can we improve its performance? One step is to recognize that both the client and the server issue a write immediately followed by a read, to wait for the other side to reply; at the cost of adding a system call, these can be combined to eliminate two kernel crossings per round trip. Further, the client will always need to wait for the server, so it makes sense for it to donate its processor to run the server code, reducing delay. Microsoft added support for this optimization to Windows in the early 1990's when it converted to a microkernel design (explained a bit later in this chapter). However, as we noted earlier, modern computer architectures make extensive use of caches, so for this to work we need code and data for both the client and the server to be able to be in cache simultaneously. We will talk about mechanisms to accomplish that in a later chapter.

We can take this streamlining even further. On a multicore system, it is possible or even likely that both the client and server each have their own processor. If the kernel sets up a shared memory region accessible to both the client and the server and no other processes, then the client and server can (safely) pass requests and replies back and forth, as fast as the memory system will allow, without ever traversing into the kernel or relinquishing their processors.

Frequently, we want to allow many clients to talk to the same server. For example, there is one server to manage the print queue, although there can be many processes that want to be able to print. For this, the server uses the select system call to identify the pipe containing the request, as shown in Figure 3.12. The client code is unchanged.

```
Server:
    char request[RequestSize];
    char reply[ReplySize];
```

```
FileDescriptor clientInput[NumClients];
FileDescriptor clientOutput[NumClients];

// Loop waiting for a request from any client.
while (fd = select(clientInput, NumClients) {

    // Read incoming command from a specific client.
    read(clientInput[fd], request, RequestSize);

    // Do operation.

    // Send result.
    write(clientOutput[fd], reply, ReplySize);
}
```

Figure 3.12: Server code for communicating with multiple clients.

3.5 Operating System Structure

We started this chapter with a list of functionality that users and applications need from the operating system. We have shown that by careful design of the system call interface, we can offload some of the work of the operating system to user programs, such as to a shell or to a print server.

In the rest of this chapter, we ask how should we organize the remaining parts of the operating system. There are many dependencies among the modules inside the operating system, and there is often quite frequent interaction between these modules:

- Many parts of the operating system depend on synchronization primitives for coordinating access to shared data structures with the kernel.
- The virtual memory system depends on low-level hardware support for address translation, support that is specific to a particular processor architecture.
- Both the file system and the virtual memory system share a common pool of blocks of physical memory. They also both depend on the disk device driver.
- The file system can depend on the network protocol stack if the disk is physically located on a different machine.

This has led operating system designers to wrestle with a fundamental tradeoff: by centralizing functionality in the kernel, performance is improved and it makes it easier to arrange tight integration between kernel modules. However, the resulting systems are less flexible, less easy to change, and less adaptive to user or application needs. We discuss these tradeoffs by describing several options for the operating system architecture.

3.5.1 Monolithic Kernels

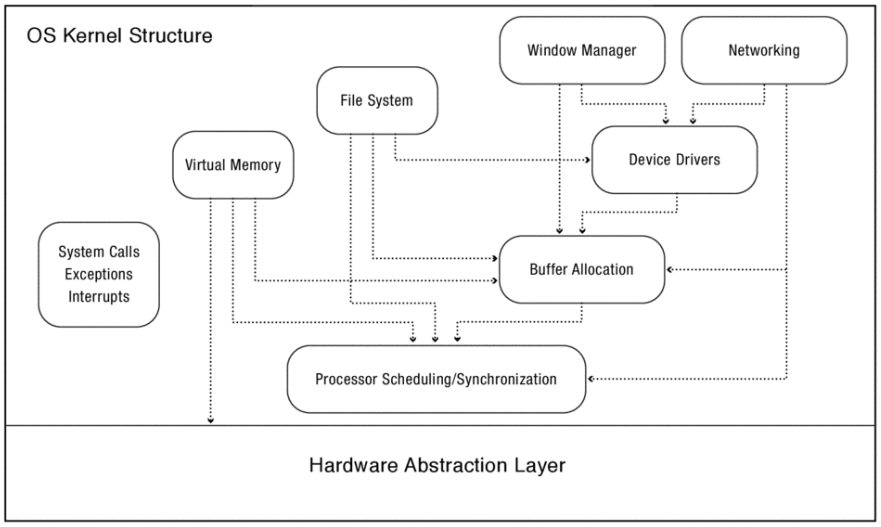


Figure 3.13: In a monolithic operating system kernel, most of the operating system functionality is linked together inside the kernel. Kernel modules directly call into other kernel modules to perform needed functions. For example, the virtual memory system uses buffer management, synchronization, and the hardware abstraction layer.

Almost all widely used commercial operating systems, such as Windows, MacOS, and Linux, take a similar approach to the architecture of the kernel — a monolithic design. As shown in Figure 3.13, with a *monolithic kernel*, most of the operating system functionality runs inside the operating system kernel. In truth, the term is a bit of a misnomer, because even in so-called monolithic systems, there are often large segments of what users consider the operating system that runs outside the kernel, either as utilities like the shell, or in system libraries, such as libraries to manage the user interface.

Internal to a monolithic kernel, the operating system designer is free to develop whatever interfaces between modules that make sense, and so there is quite a bit of variation from operating system to operating system in those internal structures. However, two common themes emerge across systems: to improve portability, almost all modern operating systems have both a hardware abstraction layer and dynamically loaded device drivers.

Hardware Abstraction Layer

A key goal of operating systems is to be portable across a wide variety of hardware platforms. To accomplish this, especially within a monolithic system, requires careful design of the *hardware abstraction layer*. The hardware abstraction layer (HAL) is a portable interface to machine configuration and processor-specific operations within the kernel. For example, within the same processor family, such as an Intel x86, different computer manufacturers will require different machine-specific code to configure and manage interrupts and hardware timers.

Operating systems that are portable across processor families, say between an ARM and an x86 or between a 32-bit and a 64-bit x86, will need processor-specific code for process and thread context switches. The interrupt, processor exception, and system call trap handling is also processor-specific; all systems have those functions, but the specific implementation will vary. As we will see in Chapter 8, machines differ quite a bit in their architecture for managing virtual address spaces; most kernels provide portable abstractions on top of the machine-dependent routines, such as to translate virtual addresses to physical addresses or to copy memory from applications to kernel memory and vice versa.

With a well-defined hardware abstraction layer in place, most of the operating system is machine- and processor-independent. Thus, porting an operating system to a new computer is just a matter of creating new implementations of these low-level HAL routines and re-linking.

The hardware abstraction layer in Windows

As a concrete example, Windows has a two-pronged strategy for portability. To allow the same Windows kernel binary to be used across personal computers manufactured by different vendors, the kernel is dynamically linked at boot time with a set of library routines specifically written for each hardware configuration. This isolates the kernel from the specifics of the motherboard hardware.

Windows also runs across a number of different processor architectures. Typically, a different kernel binary is produced for each type of processor, with any needed processor-specific code; sometimes, conditional execution is used to allow a kernel binary to be shared across closely related processor designs.

Dynamically Installed Device Drivers

A similar consideration leads to operating systems that can easily accommodate a wide variety of physical I/O devices. Although there are only a handful of different instruction set architectures in wide use today, there are a huge number of different types of physical I/O devices, manufactured by a large number of companies. There is diversity in the hardware interfaces to devices as well as in the hardware chip sets for managing the devices. A recent survey found that approximately 70% of the code in the Linux kernel was in device-specific software.

To keep the rest of the operating system kernel portable, we want to decouple the operating system source code from the specifics of each device. For instance, suppose a manufacturer creates a new printer — what steps does the operating system manufacturer need to take to accommodate that change?

The key innovation, widely adopted today, is a *dynamically loadable device driver*. A dynamically loadable device driver is software to manage a specific device, interface, or chipset, added to the operating system kernel after the kernel starts running, to handle the devices that are present on a particular machine. The device manufacturer typically provides the driver code, using a standard interface supported by the kernel. The operating system kernel calls into the driver whenever it needs to read or write data to the device.

The operating system boots with a small number of device drivers — e.g., for the disk (to read the operating system binary into memory). For the devices physically attached to the computer, the computer manufacturer bundles those drivers into a file it stores along with the bootloader. When the operating system starts up, it queries the I/O bus for which devices are attached to the computer and then loads those drivers from the file on disk. Finally, for any network-attached devices, such as a network printer, the operating system can load those drivers over the Internet.

While dynamically loadable device drivers solve one problem, they pose a different one. Errors in a device driver can corrupt the operating system kernel and application data structures; just as with a regular program, errors may not be caught immediately, so that user may be unaware that their data is being silently modified. Even worse, a malicious attacker can use device drivers to introduce a computer virus into the operating system kernel, and thereby silently gain control over the entire computer. Recent studies have found that 90% of all system crashes were due to bugs in device drivers, rather than in the operating system itself.

Operating system developers have taken five approaches to dealing with this issue:

- **Code inspection.** Operating system vendors typically require all device driver code to be submitted in advance for inspection and testing, before being allowed into the kernel.

- **Bug tracking.** After every system crash, the operating system can collect information about the system configuration and the current kernel stack, and sends this information back to a central database for analysis. Microsoft does this on a wide scale. With hundreds of millions of installed computers, even a low rate of failure can yield millions of bug reports per day. Many crashes happen inside the device driver itself, but even those that do not can sometimes be tracked down. For example, if failures are correlated with the presence of a particular device driver, or increase after the release of a new version of the driver, that can indicate the source of a problem.

- **User-level device drivers.** Both Apple and Microsoft strongly encourage new device drivers to run at user-level rather than in the kernel. Each device driver runs in a separate user-level process, using system calls to manipulate the physical device. This way, a buggy driver can only affect its own internal data structures and not the rest of the operating system kernel; if the device driver crashes, the kernel can restart it easily.

  Although user-level device drivers are becoming more common, it can be time-consuming to port existing device drivers to run at user-level. Unfortunately, there is a huge amount of existing device driver code that directly addresses internal kernel data structures; drawing a boundary around these drivers has proven difficult. Of course, supporting legacy drivers is less of a problem as completely new hardware and operating system platforms, such as smartphones and tablets, are developed.

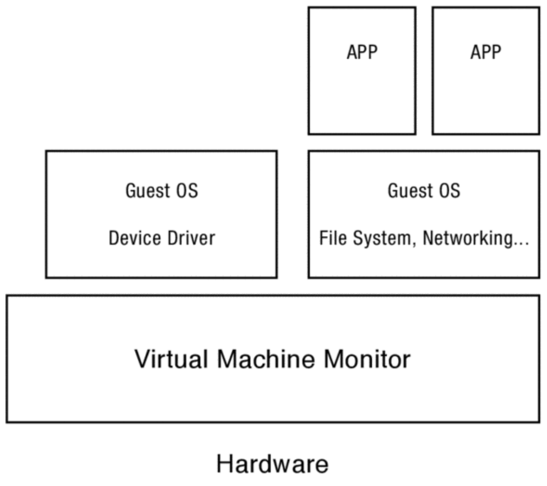


Figure 3.14: Legacy device drivers can run inside a guest operating system on top of a virtual machine in order to isolate the effect of implementation errors in driver code.

- **Virtual machine device drivers.** To handle legacy device drivers, one approach that has gained some traction is to run device driver code inside a guest operating system running on a virtual machine, as shown in Figure 3.14. The guest operating system loads the device drivers as if it was running directly on the real hardware, but when the devices attempt to access the physical hardware, the underlying virtual machine monitor regains control to ensure safety. Device drivers can still have bugs, but they can only corrupt the guest operating system and not other applications running on the underlying virtual machine monitor.

- **Driver sandboxing.** A further challenge for both user-level device drivers and virtual machine drivers is performance. Some device drivers need frequent interaction with hardware and the rest of the kernel. Some researchers have proposed running device drivers in their own restricted execution environment inside the kernel. This requires lightweight sandboxing techniques, a topic we will return to at the end of Chapter 8.

3.5.2 Microkernel

An alternative to the monolithic kernel approach is to run as much of the operating system as possible in one or more user-level servers. The window manager on most operating systems works this way: individual applications draw items on their portion of the screen by sending requests to the window manager. The window manager adjudicates which application window is in front or in back for each pixel on the screen, and then renders the result. If the system has a hardware graphics accelerator present, the window manager can use it to render items more quickly. Some systems have moved other parts of the operating system into user-level servers: the network stack, the file system, device drivers, and so forth.

The difference between a monolithic and a microkernel design is often transparent to the application programmer. The location of the service can be hidden in a user-level library — calls go to the library, which casts the requests either as system calls or as reads and writes to the server through a pipe. The location of the server can also be hidden inside the kernel — the application calls the kernel as if the kernel implements the service, but instead the kernel reformats the request into a pipe that the server can read.

A microkernel design offers considerable benefit to the operating system developer, as it easier to modularize and debug user-level services than kernel code. Aside from a potential reliability improvement, however, microkernels offer little in the way of visible benefit to end users and can slow down overall performance by inserting extra steps between the application and the services it needs. Thus, in practice, most systems adopt a hybrid model where some operating system services are run at user-level and some are in the kernel, depending on the specific tradeoff between code complexity and performance.

3.6 Summary and Future Directions

In this chapter, we have seen how system calls can be used by applications to create and manage processes, perform I/O, and communicate with other processes. Every operating system has its own unique system call interface; describing even a single interface in depth would be beyond the scope of this book. In this chapter, we focused parts of the UNIX interface because it is both compact and powerful. A key aspect of the UNIX interface are that creating a process (with fork) is separate from starting to run a program in that process (with exec); another key feature is the use of kernel buffers to decouple reading and writing data through the kernel.

Operating systems use the system call interface to provide services to applications and to aid in the internal structuring of the operating system itself. Almost all general-purpose computer systems today have a user-level shell and/or a window manager that can start and manage applications on behalf of the user. Many systems also implement parts of the operating system as user-level services accessed through kernel pipes.

As we noted, a trend is for applications to become mini-operating systems in their own right, with multiple users, resource sharing and allocation, untrusted third-party code, processor and memory management, and so forth. The system call interfaces for Windows and UNIX were not designed with this in mind, and an interesting question is how they will change to accommodate this future of powerful meta-applications.

In addition to the fine-grained sandboxing and process creation we described at the end of the last chapter, a trend is to re-structure the system call interface to make resource allocation decisions explicit and visible to applications. Traditionally, operating systems make resource allocation decisions — when to schedule a process or a thread, how much memory to give a particular application, where and when to store its disk blocks, when to send its network packets — transparently to the application, with a goal of improving end user and overall system performance. Applications are unaware of how many resources they have, appearing to run by themselves, isolated on their own (virtual) machine.

Of course, the reality is often quite different. An alternate model is for operating systems to divide resources among applications and then allow each application to decide for itself how best to use those resources. One can think of this as a type of federalism. If both the operating system and applications are governments doing their own resource allocation, they are likely to get in each other's way if they are not careful. As a simple example, consider how a garbage collector works; it assumes it has a fixed amount of memory to manage. However, as other applications start or stop, it can gain or lose memory, and if the operating system does this reallocation transparently, the garbage collector has no hope of adapting. We will see examples of this same design pattern in many different areas of operating system design.

Exercises

1. Can UNIX fork return an error? Why or why not?
   Note: You can answer this question by looking at the manual page for fork, but before you do that, think about what the fork system call does. If you were designing this call, would you need to allow fork to return an error?

2. Can UNIX exec return an error? Why or why not?
   Note: You can answer this question by looking at the manual page for exec, but before you do that, think about what the exec system call does. If you were designing this call, would you need to allow it to return an error?

3. What happens if we run the following program on UNIX?

```
main() {
    while (fork() >= 0)
        ;
}
```

4. Explain what must happen for UNIX wait to return immediately (and successfully).

5. Suppose you were the instructor of a very large introductory programming class. Explain (in English) how you would use UNIX system calls to automate testing of submitted homework assignments.

6. What happens if you run “exec csh” in a UNIX shell? Why?

7. What happens if you run “exec ls” in a UNIX shell? Why?

8. How many processes are created if the following program is run?

```
main(int argc, char ** argv) {
    forkthem(5)
}
void forkthem(int n) {
    if (n > 0) {
        fork();
        forkthem(n-1);
    }
}
```

9. Consider the following program:

```
main (int argc, char ** argv) {
    int child = fork();
    int x = 5;

    if (child == 0) {
        x += 5;
    } else {
        child = fork();
        x += 10;
        if(child) {
            x += 5;
        }
    }
}
```

   How many different copies of the variable x are there? What are their values when their process finishes?

10. What is the output of the following programs? (Please try to solve the problem without compiling and running the programs.)

```
// Program 1
main() {
    int val = 5;
    int pid;

    if (pid = fork())
        wait(pid);
    val++;
    printf("%d\n", val);
    return val;
}

// Program 2:
main() {
    int val = 5;
    int pid;
    if (pid = fork())
        wait(pid);
    else
        exit(val);
    val++;
    printf("%d\n", val);
    return val;
}
```

11. Implement a simple Linux shell in C capable of executing a sequence of programs that communicate through a pipe. For example, if the user types ls | wc, your program should fork off the two programs, which together will calculate the number of files in the directory. For this, you will need to use several of the Linux system calls described in this chapter: fork, exec, open, close, pipe, dup2, and wait. **Note:** You will to replace

stdin and stdout in the child process with the pipe file descriptors; that is the role of dup2.

12. Extend the shell implemented above to support foreground and background tasks, as well as job control: suspend, resume, and kill.

References

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In Proceedings of the 12th International conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XII, pages 2–13, 2006.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. ACM Trans. Comput. Syst., 10(1):53–79, February 1992.

[3] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In Proceedings of the fourth International conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-IV, pages 108–120, 1991.

[4] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In Proceedings of the fourth International conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-IV, pages 96–107, 1991.

[5] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation, OSDI'10, pages 1–16, 2010.

[6] Özalp Babaoglu and William Joy. Converting a swap-based system to do paging in an architecture lacking page-referenced bits. In Proceedings of the eighth ACM Symposium on Operating Systems Principles, SOSP '81, pages 78–86, 1981.

[7] David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, Jeremy Manson, John D. Mitchell, Kelvin Nilsen, Bill Pugh, and Emin Gun Sirer. The "double-checked locking is broken" declaration. http://www.cs.umd. edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

[8] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In Proceedings of the third USENIX symposium on Operating Systems Design and Implementation, OSDI '99, pages 45–58, 1999.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Proceedings of the nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pages 164–177, 2003.

[10] Blaise Barney. POSIX threads programming. http://computing.llnl.gov/tutorials/pthreads/, 2013.

[11] Joel F. Bartlett. A nonstop kernel. In Proceedings of the eighth ACM Symposium on Operating Systems Principles, SOSP '81, pages 22–29, 1981.

[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In

Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP '09, pages 29–44, 2009.

[13] A. Bensoussan, C. T. Clingen, and R. C. Daley. The multics virtual memory: concepts and design. Commun. ACM, 15(5):308–318, May 1972.

[14] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. In Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation, OSDI'10, pages 1–16, 2010.

[15] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In Proceedings of the fifteenth ACM Symposium on Operating Systems Principles, SOSP '95, pages 267–283, 1995.

[16] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. ACM Trans. Comput. Syst., 8(1):37–55, February 1990.

[17] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. ACM Trans. Comput. Syst., 9(2):175–198, May 1991.

[18] Andrew Birrell. An introduction to programming with threads. Technical Report 35, Digital Equipment Corporation Systems Research Center, 1991.

[19] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. ACM Trans. Comput. Syst., 2(1):39–59, February 1984.

[20] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation, OSDI'10, pages 1–8, 2010.

[21] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: evidence and implications. In INFOCOM, pages 126–134, 1999.

[22] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. ACM Trans. Comput. Syst., 14(1):80–107, February 1996.

[23] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In Proceedings of the seventh International conference on the World Wide Web, WWW7, pages 107–117, 1998.

[24] Max Bruning. ZFS on-disk data walk (or: Where's my data?). In OpenSolaris Developer Conference, 2008.

[25] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. ACM Trans. Comput. Syst., 15(4):412–447, November 1997.

[26] Brian Carrier. File System Forensic Analysis. Addison Wesley Professional, 2005.

[27] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP '09, pages 45–58, 2009.

[28] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: fault containment for shared-memory multiprocessors. In Proceedings of the fifteenth ACM Symposium on Operating Systems Principles, SOSP '95, pages 12–25, 1995.

[29] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. ACM Trans. Comput. Syst., 12(4):271–307, November 1994.

[30] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In Proceedings of the fourteenth ACM Symposium on Operating Systems Principles, SOSP ’93, pages 120–133, 1993.

[31] Peter M. Chen and Brian D. Noble. When virtual is better than real. In Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, HOTOS ’01, 2001.

[32] David Cheriton. The V distributed system. Commun. ACM, 31(3):314–333, March 1988.

[33] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation, OSDI ’94, 1994.

[34] David D. Clark. The structuring of systems using upcalls. In Proceedings of the tenth ACM Symposium on Operating Systems Principles, SOSP ’85, pages 171–180, 1985.

[35] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP ’09, pages 133–146, 2009.

[36] Fernando J. Corbató. On building systems that will fail. Commun. ACM, 34(9):72–81, September 1991.

[37] Fernando J. Corbató and Victor A. Vyssotsky. Introduction and overview of the Multics system. AFIPS Fall Joint Computer Conference, 27(1):185–196, 1965.

[38] R. J. Creasy. The origin of the VM/370 time-sharing system. IBM J. Res. Dev., 25(5):483–490, September 1981.

[39] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: using remote client memory to improve file system performance. In Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation, OSDI ’94, 1994.

[40] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in Multics. Commun. ACM, 11(5):306–312, May 1968.

[41] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: a new approach to improving file systems. In Proceedings of the fourteenth ACM Symposium on Operating Systems Principles, SOSP ’93, pages 15–28, 1993.

[42] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In Proceedings of the 6th USENIX Symposium on Operating Systems Design & Implementation, OSDI’04, 2004.

[43] Peter J. Denning. The working set model for program behavior. Commun. ACM, 11(5):323–333, May 1968.

[44] P.J. Denning. Working sets past and present. Software Engineering, IEEE Transactions on, SE-6(1):64 – 84, jan. 1980.

[45] Jack B. Dennis. Segmentation and the design of multiprogrammed computer systems. J. ACM, 12(4):589–602, October 1965.

[46] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. Commun. ACM, 9(3):143–155, March 1966.

[47] E. W. Dijkstra. Solution of a problem in concurrent programming control. Commun. ACM, 8(9):569–, September 1965.

[48] Edsger W. Dijkstra. The structure of the “THE”-multiprogramming system. Commun. ACM, 11(5):341–346, May 1968.

[49] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP ’09, pages 15–28, 2009.

[50] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. Portable Native Client executables. Technical report, Google, 2012.

[51] Fred Douglis and John Ousterhout. Transparent process migration: design alternatives and the Sprite implementation. Softw. Pract. Exper., 21(8):757–785, July 1991.

[52] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In Proceedings of the thirteenth ACM Symposium on Operating Systems Principles, SOSP ’91, pages 122–136, 1991.

[53] Peter Druschel and Larry L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. SIGOPS Oper. Syst. Rev., 27(5):189–202, December 1993.

[54] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. SIGOPS Oper. Syst. Rev., 36(SI):211–224, December 2002.

[55] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In Proceedings of the twentieth ACM Symposium on Operating Systems Principles, SOSP ’05, pages 17–30, 2005.

[56] D. R. Engler, M. F. Kaashoek, and J. O’Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In Proceedings of the fifteenth ACM Symposium on Operating Systems Principles, SOSP ’95, pages 251–266, 1995.

[57] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In Proceedings of the eighteenth ACM Symposium on Operating Systems Principles, SOSP ’01, pages 57–72, 2001.

[58] R. S. Fabry. Capability-based addressing. Commun. ACM, 17(7):403–412, July 1974.

[59] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In Proceedings of the seventeenth ACM Symposium on Operating Systems Principles, SOSP ’99, pages 48–63, 1999.

[60] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In Proceedings of twenty-first ACM Symposium on Operating Systems Principles, SOSP ’07, pages 307–320, 2007.

[61] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. ACM Trans.

Comput. Syst., 18(2):127–153, May 2000.

[62] Simson Garfinkel and Gene Spafford. Practical Unix and Internet security (2nd ed.). O'Reilly & Associates, Inc., 1996.

[63] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In Proceedings of the nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pages 193–206, 2003.

[64] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: ten years of implementation and experience. In Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP '09, pages 103–116, 2009.

[65] R.P. Goldberg. Survey of virtual machine research. IEEE Computer, 7(6):34–45, June 1974.

[66] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In Proceedings of the seventeenth ACM Symposium on Operating Systems Principles, SOSP '99, pages 154–169, 1999.

[67] Jim Gray. The transaction concept: virtues and limitations (invited paper). In Proceedings of the seventh International conference on Very Large Data Bases, VLDB '81, pages 144–154, 1981.

[68] Jim Gray. Why do computers stop and what can be done about it? Technical Report TR-85.7, HP Labs, 1985.

[69] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The recovery manager of the System R database manager. ACM Comput. Surv., 13(2):223–242, June 1981.

[70] Jim Gray and Andreas Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[71] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. Computer, 24(9):39–48, September 1991.

[72] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: harnessing memory redundancy in virtual machines. In Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation, OSDI'08, pages 309–322, 2008.

[73] Hadoop. http://hadoop.apache.org.

[74] Steven M. Hand. Self-paging in the Nemesis operating system. In Proceedings of the third USENIX Symposium on Operating Systems Design and Implementation, OSDI '99, pages 73–86, 1999.

[75] Per Brinch Hansen. The nucleus of a multiprogramming system. Commun. ACM, 13(4):238–241, April 1970.

[76] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. In Proceedings of the fifteenth ACM Symposium on Operating Systems Principles, SOSP '95, pages 236–, 1995.

[77] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In Proceedings of the fifth International

conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-V, pages 187–197, 1992.

[78] Rober Haskin, Yoni Malachi, and Gregory Chan. Recovery management in QuickSilver. ACM Trans. Comput. Syst., 6(1):82–108, February 1988.

[79] John L. Hennessy and David A. Patterson. Computer Architecture - A Quantitative Approach (5. ed.). Morgan Kaufmann, 2012.

[80] Maurice Herlihy. Wait-free synchronization. ACM Trans. Program. Lang. Syst., 13(1):124–149, January 1991.

[81] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann, 2008.

[82] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. Technical Report 3002, Network Appliance, 1995.

[83] C. A. R. Hoare. Monitors: An operating system structuring concept. Communications of the ACM, 17:549–557, 1974.

[84] C. A. R. Hoare. Communicating sequential processes. Commun. ACM, 21(8):666–677, August 1978.

[85] C. A. R. Hoare. The emperor’s old clothes. Commun. ACM, 24(2):75–83, February 1981.

[86] Thomas R. Horsley and William C. Lynch. Pilot: A software engineering case study. In Proceedings of the 4th International conference on Software engineering, ICSE ’79, pages 94–99, 1979.

[87] Raj Jain. The Art of Computer Systems Performance Analysis. John Wiley & Sons, 1991.

[88] Asim Kadav and Michael M. Swift. Understanding modern device drivers. In Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’12, pages 87–98, New York, NY, USA, 2012. ACM.

[89] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the VAX VMM security kernel. IEEE Trans. Softw. Eng., 17(11):1147–1165, November 1991.

[90] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in Spring. In Proceedings of the fourteenth ACM Symposium on Operating Systems Principles, SOSP ’93, pages 1–14, 1993.

[91] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP ’09, pages 207–220, 2009.

[92] L. Kleinrock and R. R. Muntz. Processor sharing queueing models of mixed scheduling disciplines for time shared system. J. ACM, 19(3):464–482, July 1972.

[93] Leonard Kleinrock. Queueing Systems, Volume II: Computer Applications. Wiley Interscience, 1976.

[94] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. ACM Trans. Database Syst., 6(2):213–226, June 1981.

[95] Leslie Lamport. A fast mutual exclusion algorithm. ACM Trans. Comput. Syst., 5(1):1–11, January 1987.

[96] B. W. Lampson. Hints for computer system design. IEEE Softw., 1(1):11–28, January 1984.

[97] Butler Lampson and Howard Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, 1979.

[98] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. Commun. ACM, 23(2):105–117, February 1980.

[99] Butler W. Lampson and Howard E. Sturgis. Reflections on an operating system design. Commun. ACM, 19(5):251–265, May 1976.

[100] James Larus and Galen Hunt. The Singularity system. Commun. ACM, 53(8):72–79, August 2010.

[101] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. In Operating Systems Review, pages 3–19, 1979.

[102] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. Quantitative system performance: computer system analysis using queueing network models. Prentice-Hall, Inc., 1984.

[103] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic (extended version). IEEE/ACM Trans. Netw., 2(1):1–15, February 1994.

[104] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. Computer, 26(7):18–41, July 1993.

[105] H. M. Levy and P. H. Lipman. Virtual memory management in the VAX/VMS operating system. Computer, 15(3):35–41, March 1982.

[106] J. Liedtke. On micro-kernel construction. In Proceedings of the fifteenth ACM Symposium on Operating Systems Principles, SOSP ’95, pages 237–250, 1995.

[107] John Lions. Lions’ Commentary on UNIX 6th Edition, with Source Code. Peer-to-Peer Communications, 1996.

[108] J. S. Liptay. Structural aspects of the System/360 model 85: ii the cache. IBM Syst. J., 7(1):15–21, March 1968.

[109] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In Proceedings of the 4th conference on Symposium on Operating Systems Design and Implementation, OSDI’00, pages 20–20, 2000.

[110] David E. Lowell and Peter M. Chen. Free transactions with Rio Vista. In Proceedings of the sixteenth ACM Symposium on Operating Systems Principles, SOSP ’97, pages 92–101, 1997.

[111] P. McKenney. Is parallel programming hard, and, if so, what can be done about it? http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.2011.05.30a.pdf.

[112] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In Ottawa Linux Symposium, pages 338–367, June 2002.

[113] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. ACM Trans. Comput. Syst., 2(3):181–197, August 1984.

[114] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. The design and implementation of the 4.4BSD operating system. Addison Wesley Longman Publishing Co., Inc., 1996.

[115] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable

synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst., 9(1):21–65, February 1991.

[116] Scott Meyers and Andrei Alexandrescu. C++ and the perils of double-checked locking. Dr. Dobbs Journal, 2004.

[117] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. ACM Trans. Comput. Syst., 15(3):217–252, August 1997.

[118] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In In the Proceedings of the eleventh ACM Symposium on Operating Systems Principles, pages 39–51, 1987.

[119] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst., 17(1):94–162, March 1992.

[120] Gordon E. Moore. Cramming more components onto integrated circuits. Electronics, 38(8):114–117, 1965.

[121] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation, OSDI'08, pages 267–280, 2008.

[122] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. J. Phys. I France, 1992.

[123] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In Proceedings of the second USENIX Symposium on Operating Systems Design and Implementation, OSDI '96, pages 229–243, 1996.

[124] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. ACM Trans. Comput. Syst., 26(3):6:1–6:26, September 2008.

[125] Elliott I. Organick. The Multics system: an examination of its structure. MIT Press, 1972.

[126] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of Zap: a system for migrating computing environments. In Proceedings of the fifth USENIX Symposium on Operating Systems Design and Implementation, OSDI '02, pages 361–376, 2002.

[127] John Ousterhout. Scheduling techniques for concurrent systems. In Proceedings of Third International Conference on Distributed Computing Systems, pages 22–30, 1982.

[128] John Ousterhout. Why aren't operating systems getting faster as fast as hardware? In Proceedings USENIX Conference, pages 247–256, 1990.

[129] John Ousterhout. Why threads are a bad idea (for most purposes). In USENIX Winter Technical Conference, 1996.

[130] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: an efficient and portable web server. In Proceedings of the annual conference on USENIX Annual Technical Conference, ATEC '99, 1999.

[131] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-lite: a unified I/O buffering and caching system. In Proceedings of the third USENIX Symposium on Operating Systems Design and Implementation, OSDI '99, pages 15–28, 1999.

[132] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays

of inexpensive disks (RAID). In Proceedings of the 1988 ACM SIGMOD International conference on Management of Data, SIGMOD '88, pages 109–116, 1988.

[133] L. Peterson, N. Hutchinson, S. O'Malley, and M. Abbott. RPC in the x-Kernel: evaluating new design techniques. In Proceedings of the twelfth ACM Symposium on Operating Systems Principles, SOSP '89, pages 91–101, 1989.

[134] Jonathan Pincus and Brandon Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. IEEE Security and Privacy, 2(4):20–27, July 2004.

[135] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In Proceedings of the 5th USENIX conference on File and Storage Technologies, FAST '07, pages 2–2, 2007.

[136] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In Proceedings of the twentieth ACM Symposium on Operating Systems Principles, SOSP '05, pages 206–220, 2005.

[137] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. Mach: A foundation for open systems. In Proceedings of the Second Workshop on Workstation Operating Systems(WWOS2), 1989.

[138] Richard F. Rashid, Avadis Tevanian, Michael Young, David B. Golub, Robert V. Baron, David L. Black, William J. Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. IEEE Trans. Computers, 37(8):896–907, 1988.

[139] E.S. Raymond. The Cathedral and the Bazaar: Musings On Linux And Open Source By An Accidental Revolutionary. O'Reilly Series. O'Reilly, 2001.

[140] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: an operating system for a personal computer. Commun. ACM, 23(2):81–92, February 1980.

[141] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. Commun. ACM, 17(7):365–375, July 1974.

[142] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. ACM Trans. Comput. Syst., 10(1):26–52, February 1992.

[143] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. Computer, 27(3):17–28, March 1994.

[144] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. ACM Trans. Comput. Syst., 2(4):277–288, November 1984.

[145] Jerome H. Saltzer. Protection and the control of information sharing in Multics. Commun. ACM, 17(7):388–402, July 1974.

[146] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. ACM Trans. Comput. Syst., 12(1):33–57, February 1994.

[147] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst., 15(4):391–411, November 1997.

[148] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you? In Proceedings of the 5th USENIX conference on File and Storage Technologies, FAST '07, 2007.

[149] Bianca Schroeder and Mor Harchol-Balter. Web servers under overload: How scheduling can help. ACM Trans. Internet Technol., 6(1):20–52, February 2006.

[150] Michael D. Schroeder, David D. Clark, and Jerome H. Saltzer. The Multics kernel design project. In Proceedings of the sixth ACM Symposium on Operating Systems Principles, SOSP ’77, pages 43–56, 1977.

[151] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. Commun. ACM, 15(3):157–170, March 1972.

[152] D. P. Siewiorek. Architecture of fault-tolerant computers. Computer, 17(8):9–18, August 1984.

[153] E. H. Spafford. Crisis and aftermath. Commun. ACM, 32(6):678–687, June 1989.

[154] Structured Query Language (SQL). http://en.wikipedia.org/wiki/SQL.

[155] Michael Stonebraker. Operating system support for database management. Commun. ACM, 24(7):412–418, July 1981.

[156] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. ACM Trans. Comput. Syst., 24(4):333–360, November 2006.

[157] K. Thompson. Unix implementation. Bell System Technical Journal, 57:1931–1946, 1978.

[158] Ken Thompson. Reflections on trusting trust. Commun. ACM, 27(8):761–763, August 1984.

[159] Paul Tyma. Thousands of threads and blocking i/o. http://www.mailinator.com/tymaPaulMultithreaded.pdf, 2008.

[160] Robbert van Renesse. Goal-oriented programming, or composition using events, or threads considered harmful. In ACM SIGOPS European Workshop on Support for Composing Distributed Applications, pages 82–87, 1998.

[161] Joost S. M. Verhofstad. Recovery techniques for database systems. ACM Comput. Surv., 10(2):167–195, June 1978.

[162] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In Proceedings of the twentieth ACM Symposium on Operating Systems Principles, SOSP ’05, pages 148–162, 2005.

[163] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In Proceedings of the fourteenth ACM Symposium on Operating Systems Principles, SOSP ’93, pages 203–216, 1993.

[164] Carl A. Waldspurger. Memory resource management in VMware ESX server. SIGOPS Oper. Syst. Rev., 36(SI):181–194, December 2002.

[165] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In Proceedings of the fifth USENIX Symposium on Operating Systems Design and Implementation, OSDI ’02, pages 195–209, 2002.

[166] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. In Proceedings of the fifteenth ACM Symposium on Operating Systems Principles, SOSP ’95, pages 96–108, 1995.

[167] Alec Wolman, M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy. On the scale and performance of cooperative web proxy caching. In Proceedings of the seventeenth ACM Symposium on Operating Systems Principles, SOSP ’99, pages 16–31, 1999.

[168] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: the kernel of a multiprocessor operating system. Commun. ACM, 17(6):337–345, June 1974.

[169] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. In Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, SP ’09, pages 79–93, 2009.

[170] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. Commun. ACM, 54(11):93–101, November 2011.

Glossary

absolute path
A file path name interpreted relative to the root directory.

abstract virtual machine
The interface provided by an operating system to its applications, including the system call interface, the memory abstraction, exceptions, and signals.

ACID properties
A mnemonic for the properties of a transaction: atomicity, consistency, isolation, and durability.

acquire-all/release-all
A design pattern to provide atomicity of a request consisting of multiple operations. A thread acquires all of the locks it might need before starting to process a request; it releases the locks once the request is done.

address translation
The conversion from the memory address the program thinks it is referencing to the physical location of the memory.

affinity scheduling
A scheduling policy where tasks are preferentially scheduled onto the same processor they had previously been assigned, to improve cache reuse.

annual disk failure rate
The fraction of disks expected to failure each year.

API
See: *application programming interface*.

application programming interface
The system call interface provided by an operating system to applications.

arm
An attachment allowing the motion of the disk head across a disk surface.

arm assembly
A motor plus the set of disk arms needed to position a disk head to read or write each surface of the disk.

arrival rate
The rate at which tasks arrive for service.

asynchronous I/O
A design pattern for system calls to allow a single-threaded process to make multiple concurrent I/O requests. When the process issues an I/O request, the system call returns immediately. The process later on receives a notification when the I/O completes.

asynchronous procedure call
A procedure call where the caller starts the function, continues execution concurrently with the called function, and later waits for the function to complete.

atomic commit
The moment when a transaction commits to apply all of its updates.

atomic memory

The value stored in memory is the last value stored by one of the processors, not a mixture of the updates of different processors.

atomic operations

Indivisible operations that cannot be interleaved with or split by other operations.

atomic read-modify-write instruction

A processor-specific instruction that lets one thread temporarily have exclusive and atomic access to a memory location while the instruction executes. Typically, the instruction (atomically) reads a memory location, does some simple arithmetic operation to the value, and stores the result.

attribute record

In NTFS, a variable-size data structure containing either file data or file metadata.

availability

The percentage of time that a system is usable.

average seek time

The average time across seeks between each possible pair of tracks on a disk.

AVM

See: *abstract virtual machine*.

backup

A logically or physically separate copy of a system’s main storage.

base and bound memory protection

An early system for memory protection where each process is limited to a specific range of physical memory.

batch operating system

An early type of operating system that efficiently ran a queue of tasks. While one program was running, another was being loaded into memory.

bathtub model

A model of disk device failure combining device infant mortality and wear out.

Belady’s anomaly

For some cache replacement policies and some reference patterns, adding space to a cache can hurt the cache hit rate.

best fit

A storage allocation policy that attempts to place a newly allocated file in the smallest free region that is large enough to hold it.

BIOS

The initial code run when an Intel x86 computer boots; acronym for Basic Input/Output System. See also: *Boot ROM*.

bit error rate

The non-recoverable read error rate.

bitmap

A data structure for block allocation where each block is represented by one bit.

block device

An I/O device that allows data to be read or written in fixed-sized blocks.

block group

A set of nearby disk tracks.

block integrity metadata

Additional data stored with a block to allow the software to validate that the block has not been corrupted.

blocking bounded queue

A bounded queue where a thread trying to remove an item from an empty queue will wait until an item is available, and a thread trying to put an item into a full queue will wait until there is room.

Bohrbugs

Bugs that are deterministic and reproducible, given the same program input. See also: *Heisenbugs*.

Boot ROM

Special read-only memory containing the initial instructions for booting a computer.

bootloader

Program stored at a fixed position on disk (or flash RAM) to load the operating system into memory and start it executing.

bounded queue

A queue with a fixed size limit on the number of items stored in the queue.

bounded resources

A necessary condition for deadlock: there are a finite number of resources that threads can simultaneously use.

buffer overflow attack

An attack that exploits a bug where input can overflow the buffer allocated to hold it, overwriting other important program data structures with data provided by the attacker. One common variation overflows a buffer allocated on the stack (e.g., a local, automatic variable) and replaces the function's return address with a return address specified by the attacker, possibly to code "pushed" onto the stack with the overflowing input.

bulk synchronous

A type of parallel application where work is split into independent tasks and where each task completes before the results of any of the tasks can be used.

bulk synchronous parallel programming

See: *data parallel programming*.

bursty distribution

A probability distribution that is less evenly distributed around the mean value than an exponential distribution. See: *exponential distribution*. Compare: *heavy-tailed distribution*.

busy-waiting

A thread spins in a loop waiting for a concurrent event to occur, consuming CPU cycles while it is waiting.

cache

A copy of data that can be accessed more quickly than the original.

cache hit

The cache contains the requested item.

cache miss

The cache does not contain the requested item.

checkpoint

A consistent snapshot of the entire state of a process, including the contents of memory and processor registers.

child process

A process created by another process. See also: *parent process*.

Circular SCAN

See: *CSCAN*.

circular waiting

A necessary condition for deadlock to occur: there is a set of threads such that each thread is waiting for a resource held by another.

client-server communication
Two-way communication between processes, where the client sends a request to the server to do some task, and when the operation is complete, the server replies back to the client.

clock algorithm
A method for identifying a not recently used page to evict. The algorithm sweeps through each page frame: if the page use bit is set, it is cleared; if the use bit is not set, the page is reclaimed.

cloud computing
A model of computing where large-scale applications run on shared computing and storage infrastructure in data centers instead of on the user’s own computer.

commit
The outcome of a transaction where all of its updates occur.

compare-and-swap
An atomic read-modify-write instruction that first tests the value of a memory location, and if the value has not been changed, sets it to a new value.

compute-bound task
A task that primarily uses the processor and does little I/O.

computer virus
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.

concurrency
Multiple activities that can happen at the same time.

condition variable
A synchronization variable that enables a thread to efficiently wait for a change to shared state protected by a lock.

continuation
A data structure used in event-driven programming that keeps track of a task’s current state and its next step.

cooperating threads
Threads that read and write shared state.

cooperative caching
Using the memory of nearby nodes over a network as a cache to avoid the latency of going to disk.

cooperative multi-threading
Each thread runs without interruption until it explicitly relinquishes control of the processor, e.g., by exiting or calling thread_yield.

copy-on-write
A method of sharing physical memory between two logically distinct copies (e.g., in different processes). Each shared page is marked as read-only so that the operating system kernel is invoked and can make a copy of the page if either process tries to write it. The process can then modify the copy and resume normal execution.

copy-on-write file system
A file system where an update to the file system is made by writing new versions of modified data and metadata blocks to free disk blocks. The new blocks can point to

unchanged blocks in the previous version of the file system. See also: *COW file system*.

core map
: A data structure used by the memory management system to keep track of the state of physical page frames, such as which processes reference the page frame.

COW file system
: See: *copy-on-write file system*.

critical path
: The minimum sequence of steps for a parallel application to compute its result, even with infinite resources.

critical section
: A sequence of code that operates on shared state.

cross-site scripting
: An attack against a client computer that works by compromising a server visited by the client. The compromised server then provides scripting code to the client that accesses and downloads the client's sensitive data.

cryptographic signature
: A specially designed function of a data block and a private cryptographic key that allows someone with the corresponding public key to verify that an authorized entity produced the data block. It is computationally intractable for an attacker without the private key to create a different data block with a valid signature.

CSCAN
: A variation of the SCAN disk scheduling policy in which the disk only services requests when the head is traveling in one direction. See also: *Circular SCAN*.

current working directory
: The current directory of the process, used for interpreting relative path names.

data breakpoint
: A request to stop the execution of a program when it references or modifies a particular memory location.

data parallel programming
: A programming model where the computation is performed in parallel across all items in a data set.

deadlock
: A cycle of waiting among a set of threads, where each thread waits for some other thread in the cycle to take some action.

deadlocked state
: The system has at least one deadlock.

declustering
: A technique for reducing the recovery time after a disk failure in a RAID system by spreading redundant disk blocks across many disks.

defense in depth
: Improving security through multiple layers of protection.

defragment
: Coalesce scattered disk blocks to improve spatial locality, by reading data from its present storage location and rewriting it to a new, more compact, location.

demand paging
: Using address translation hardware to run a process without all of its memory physically present. When the process references a missing page, the hardware traps to the kernel, which brings the page into memory from disk.

deterministic debugging
The ability to re-execute a concurrent process with the same schedule and sequence of internal and external events.
device driver
Operating system code to initialize and manage a particular I/O device.
direct mapped cache
Only one entry in the cache can hold a specific memory location, so on a lookup, the system must check the address against only that entry to determine if there is a cache hit.
direct memory access
Hardware I/O devices transfer data directly into/out of main memory at a location specified by the operating system. See also: *DMA*.
dirty bit
A status bit in a page table entry recording whether the contents of the page have been modified relative to what is stored on disk.
disk buffer memory
Memory in the disk controller to buffer data being read or written to the disk.
disk infant mortality
The device failure rate is higher than normal during the first few weeks of use.
disk wear out
The device failure rate rises after the device has been in operation for several years.
DMA
See: *direct memory access*.
dnode
In ZFS, a file is represented by variable-depth tree whose root is a dnode and whose leaves are its data blocks.
double indirect block
A storage block containing pointers to indirect blocks.
double-checked locking
A pitfall in concurrent code where a data structure is lazily initialized by first, checking without a lock if it has been set, and if not, acquiring a lock and checking again, before calling the initialization function. With instruction re-ordering, double-checked locking can fail unexpectedly.
dual redundancy array
A RAID storage algorithm using two redundant disk blocks per array to tolerate two disk failures. See also: *RAID 6*.
dual-mode operation
Hardware processor that has (at least) two privilege levels: one for executing the kernel with complete access to the capabilities of the hardware and a second for executing user code with restricted rights. See also: *kernel-mode operation*. See also: *user-mode operation*.
dynamically loadable device driver
Software to manage a specific device, interface, or chipset, added to the operating system kernel after the kernel starts running.
earliest deadline first
A scheduling policy that performs the task that needs to be completed first, but only if it can be finished in time.
EDF
See: *earliest deadline first*.

efficiency
The lack of overhead in implementing an abstraction.

erasure block
The unit of erasure in a flash memory device. Before any portion of an erasure block can be over-written, every cell in the entire erasure block must be set to a logical "1."

error correcting code
A technique for storing data redundantly to allow for the original data to be recovered even though some bits in a disk sector or flash memory page are corrupted.

event-driven programming
A coding design pattern where a thread spins in a loop; each iteration gets and processes the next I/O event.

exception
See: *processor exception*.

executable image
File containing a sequence of machine instructions and initial data values for a program.

execution stack
Space to store the state of local variables during procedure calls.

exponential distribution
A convenient probability distribution for use in queueing theory because it has the property of being memoryless. For a continuous random variable with a mean of $1/\lambda$, the probability density function is $f(x) = \lambda e\text{-}\lambda x$.

extent
A variable-sized region of a file that is stored in a contiguous region on the storage device.

external fragmentation
In a system that allocates memory in contiguous regions, the unusable memory between valid contiguous allocations. A new request for memory may find no single free region that is both contiguous and large enough, even though there is enough free memory in aggregate.

fairness
Partitioning of shared resources between users or applications either equally or balanced according to some desired priorities.

false sharing
Extra inter-processor communication required because a single cache entry contains portions of two different data structures with different sharing patterns.

fate sharing
When a crash in one module implies a crash in another. For example, a library shares fate with the application it is linked with; if either crashes, the process exits.

fault isolation
An error in one application should not disrupt other applications, or even the operating system itself.

file
A named collection of data in a file system.

file allocation table
An array of entries in the FAT file system stored in a reserved area of the volume, where each entry corresponds to one file data block, and points to the next block in the file.

file data

Contents of a file.

file descriptor
A handle to an open file, device, or channel. See also: *file handle*. See also: *file stream*.

file directory
A list of human-readable names plus a mapping from each name to a specific file or sub-directory.

file handle
See: *file descriptor*.

file index structure
A persistently stored data structure used to locate the blocks of the file.

file metadata
Information about a file that is managed by the operating system, but not including the file contents.

file stream
See: *file descriptor*.

file system
An operating system abstraction that provides persistent, named data.

file system fingerprint
A checksum across the entire file system.

fill-on-demand
A method for starting a process before all of its memory is brought in from disk. If the first access to the missing memory triggers a trap to the kernel, the kernel can fill the memory and then resume.

fine-grained locking
A way to increase concurrency by partitioning an object's state into different subsets each protected by a different lock.

finished list
The set of threads that are complete but not yet de-allocated, e.g., because a join may read the return value from the thread control block.

first-in-first-out
A scheduling policy that performs each task in the order in which it arrives.

flash page failure
A flash memory device failure where the data stored on one or more individual pages of flash are lost, but the rest of the flash continues to operate correctly.

flash translation layer
A layer that maps logical flash pages to different physical pages on the flash device. See also: *FTL*.

flash wear out
After some number of program-erase cycles, a given flash storage cell may no longer be able to reliably store information.

fork-join parallelism
A type of parallel programming where threads can be created (forked) to do work in parallel with a parent thread; a parent may asynchronously wait for a child thread to finish (join).

free space map
A file system data structure used to track which storage blocks are free and which are in use.

FTL

See: *flash translation layer*.

full disk failure
When a disk device stops being able to service reads or writes to all sectors.

full flash drive failure
When a flash device stops being able to service reads or writes to all memory pages.

fully associative cache
Any entry in the cache can hold any memory location, so on a lookup, the system must check the address against all of the entries in the cache to determine if there is a cache hit.

gang scheduling
A scheduling policy for multiprocessors that performs all of the runnable tasks for a particular process at the same time.

Global Descriptor Table
The x86 terminology for a segment table for shared segments. A Local Descriptor Table is used for segments that are private to the process.

grace period
For a shared object protected by a read-copy-update lock, the time from when a new version of a shared object is published until the last reader of the old version is guaranteed to be finished.

green threads
A thread system implemented entirely at user-level without any reliance on operating system kernel services, other than those designed for single-threaded processes.

group commit
A technique that batches multiple transaction commits into a single disk operation.

guest operating system
An operating system running in a virtual machine.

hard link
The mapping between a file name and the underlying file, typically when there are multiple path names for the same underlying file.

hardware abstraction layer
A module in the operating system that hides the specifics of different hardware implementations. Above this layer, the operating system is portable.

hardware timer
A hardware device that can cause a processor interrupt after some delay, either in time or in instructions executed.

head
The component that writes the data to or reads the data from a spinning disk surface.

head crash
An error where the disk head physically scrapes the magnetic surface of a spinning disk surface.

head switch time
The time it takes to re-position the disk arm over the corresponding track on a different surface, before a read or write can begin.

heap
Space to store dynamically allocated data structures.

heavy-tailed distribution
A probability distribution such that events far from the mean value (in aggregate) occur with significant probability. When used for the distribution of time between events, the

remaining time to the next event is positively related to the time already spent waiting — you expect to wait longer the longer you have already waited.

Heisenbugs
Bugs in concurrent programs that disappear or change behavior when you try to examine them. See also: *Bohrbugs*.

hint
A result of some computation whose results may no longer be valid, but where using an invalid hint will trigger an exception.

home directory
The sub-directory containing a user's files.

host operating system
An operating system that provides the abstraction of a virtual machine, to run another operating system as an application.

host transfer time
The time to transfer data between the host's memory and the disk's buffer.

hyperthreading
See: *simultaneous multi-threading*.

I/O-bound task
A task that primarily does I/O, and does little processing.

idempotent
An operation that has the same effect whether executed once or many times.

incremental checkpoint
A consistent snapshot of the portion of process memory that has been modified since the previous checkpoint.

independent threads
Threads that operate on completely separate subsets of process memory.

indirect block
A storage block containing pointers to file data blocks.

inode
In the Unix Fast File System (FFS) and related file systems, an inode stores a file's metadata, including an array of pointers that can be used to find all of the file's blocks. The term inode is sometimes used more generally to refer to any file system's per-file metadata data structure.

inode array
The fixed location on disk containing all of the file system's inodes. See also: *inumber*.

intentions
The set of writes that a transaction will perform if the transaction commits.

internal fragmentation
With paged allocation of memory, the unusable memory at the end of a page because a process can only be allocated memory in page-sized chunks.

interrupt
An asynchronous signal to the processor that some external event has occurred that may require its attention.

interrupt disable
A privileged hardware instruction to temporarily defer any hardware interrupts, to allow the kernel to complete a critical task.

interrupt enable
A privileged hardware instruction to resume hardware interrupts, after a non-interruptible task is completed.

interrupt handler
A kernel procedure invoked when an interrupt occurs.

interrupt stack
A region of memory for holding the stack of the kernel's interrupt handler. When an interrupt, processor exception, or system call trap causes a context switch into the kernel, the hardware changes the stack pointer to point to the base of the kernel's interrupt stack.

interrupt vector table
A table of pointers in the operating system kernel, indexed by the type of interrupt, with each entry pointing to the first instruction of a handler procedure for that interrupt.

inumber
The index into the inode array for a particular file.

inverted page table
A hash table used for translation between virtual page numbers and physical page frames.

kernel thread
A thread that is implemented inside the operating system kernel.

kernel-mode operation
The processor executes in an unrestricted mode that gives the operating system full control over the hardware. Compare: *user-mode operation*.

LBA
See: *logical block address*.

least frequently used
A cache replacement policy that evicts whichever block has been used the least often, over some period of time. See also: *LFU*.

least recently used
A cache replacement policy that evicts whichever block has not been used for the longest period of time. See also: *LRU*.

LFU
See: *least frequently used*.

Little's Law
In a stable system where the arrival rate matches the departure rate, the number of tasks in the system equals the system's throughput multiplied by the average time a task spends in the system: N = XR.

liveness property
A constraint on program behavior such that it always produces a result. Compare: *safety property*.

locality heuristic
A file system block allocation policy that places files in nearby disk sectors if they are likely to be read or written at the same time.

lock
A type of synchronization variable used for enforcing atomic, mutually exclusive access to shared data.

lock ordering
A widely used approach to prevent deadlock, where locks are acquired in a pre-determined order.

lock-free data structures
Concurrent data structure that guarantees progress for some thread: some method will finish in a finite number of steps, regardless of the state of other threads executing in

the data structure.

log
: An ordered sequence of steps saved to persistent storage.

logical block address
: A unique identifier for each disk sector or flash memory block, typically numbered from 1 to the size of the disk/flash device. The disk interface converts this identifier to the physical location of the sector/block. See also: *LBA*.

logical separation
: A backup storage policy where the backup is stored at the same location as the primary storage, but with restricted access, e.g., to prevent updates.

LRU
: See: *least recently used*.

master file table
: In NTFS, an array of records storing metadata about each file. See also: *MFT*.

maximum seek time
: The time it takes to move the disk arm from the innermost track to the outermost one or vice versa.

max-min fairness
: A scheduling objective to maximize the minimum resource allocation given to each task.

MCS lock
: An efficient spinlock implementation where each waiting thread spins on a separate memory location.

mean time to data loss
: The expected time until a RAID system suffers an unrecoverable error. See also: *MTTDL*.

mean time to failure
: The average time that a system runs without failing. See also: *MTTF*.

mean time to repair
: The average time that it takes to repair a system once it has failed. See also: *MTTR*.

memory address alias
: Two or more virtual addresses that refer to the same physical memory location.

memory barrier
: An instruction that prevents the compiler and hardware from reordering memory accesses across the barrier — no accesses before the barrier are moved after the barrier and no accesses after the barrier are moved before the barrier.

memory protection
: Hardware or software-enforced limits so that each application process can read and write only its own memory and not the memory of the operating system or any other process.

memoryless property
: For a probability distribution for the time between events, the remaining time to the next event does not depend on the amount of time already spent waiting. See also: *exponential distribution*.

memory-mapped file
: A file whose contents appear to be a memory segment in a process's virtual address space.

memory-mapped I/O

Each I/O device's control registers are mapped to a range of physical addresses on the memory bus.

memristor

A type of solid-state persistent storage using a circuit element whose resistance depends on the amounts and directions of currents that have flowed through it in the past.

MFQ

See: *multi-level feedback queue*.

MFT

See: *master file table*.

microkernel

An operating system design where the kernel itself is kept small, and instead most of the functionality of a traditional operating system kernel is put into a set of user-level processes, or servers, accessed from user applications via interprocess communication.

MIN cache replacement

See: *optimal cache replacement*.

minimum seek time

The time to move the disk arm to the next adjacent track.

MIPS

An early measure of processor performance: millions of instructions per second.

mirroring

A system for redundantly storing data on disk where each block of data is stored on two disks and can be read from either. See also: *RAID 1*.

model

A simplification that tries to capture the most important aspects of a more complex system's behavior.

monolithic kernel

An operating system design where most of the operating system functionality is linked together inside the kernel.

Moore's Law

Transistor density increases exponentially over time. Similar exponential improvements have occurred in many other component technologies; in the popular press, these often go by the same term.

mount

A mapping of a path in the existing file system to the root directory of another file system volume.

MTTDL

See: *mean time to data loss*.

MTTF

See: *mean time to failure*.

MTTR

See: *mean time to repair*.

multi-level feedback queue

A scheduling algorithm with multiple priority levels managed using round robin queues, where a task is moved between priority levels based on how much processing time it has used. See also: *MFQ*.

multi-level index

A tree data structure to keep track of the disk location of each data block in a file.

multi-level paged segmentation
A virtual memory mechanism where physical memory is allocated in page frames, virtual addresses are segmented, and each segment is translated to physical addresses through multiple levels of page tables.
multi-level paging
A virtual memory mechanism where physical memory is allocated in page frames, and virtual addresses are translated to physical addresses through multiple levels of page tables.
multiple independent requests
A necessary condition for deadlock to occur: a thread first acquires one resource and then tries to acquire another.
multiprocessor scheduling policy
A policy to determine how many processors to assign each process.
multiprogramming
See: *multitasking*.
multitasking
The ability of an operating system to run multiple applications at the same time, also called multiprogramming.
multi-threaded process
A process with multiple threads.
multi-threaded program
A generalization of a single-threaded program. Instead of only one logical sequence of steps, the program has multiple sequences, or threads, executing at the same time.
mutual exclusion
When one thread uses a lock to prevent concurrent access to a shared data structure.
mutually recursive locking
A deadlock condition where two shared objects call into each other while still holding their locks. Deadlock occurs if one thread holds the lock on the first object and calls into the second, while the other thread holds the lock on the second object and calls into the first.
named data
Data that can be accessed by a human-readable identifier, such as a file name.
native command queueing
See: *tagged command queueing*.
NCQ
See: *native command queueing*.
nested waiting
A deadlock condition where one shared object calls into another shared object while holding the first object's lock, and then waits on a condition variable. Deadlock results if the thread that can signal the condition variable needs the first lock to make progress.
network effect
The increase in value of a product or service based on the number of other people who have adopted that technology and not just its intrinsic capabilities.
no preemption
A necessary condition for deadlock to occur: once a thread acquires a resource, its ownership cannot be revoked until the thread acts to release it.
non-blocking data structure

Concurrent data structure where a thread is never required to wait for another thread to complete its operation.

non-recoverable read error
When sufficient bit errors occur within a disk sector or flash memory page, such that the original data cannot be recovered even after error correction.

non-resident attribute
In NTFS, an attribute record whose contents are addressed indirectly, through extent pointers in the master file table that point to the contents in those extents.

non-volatile storage
Unlike DRAM, memory that is durable and retains its state across crashes and power outages. See also: *persistent storage*. See also: *stable storage*.

not recently used
A cache replacement policy that evicts some block that has not been referenced recently, rather than the least recently used block.

oblivious scheduling
A scheduling policy where the operating system assigns threads to processors without knowledge of the intent of the parallel application.

open system
A system whose source code is available to the public for modification and reuse, or a system whose interfaces are defined by a public standards process.

operating system
A layer of software that manages a computer's resources for its users and their applications.

operating system kernel
The kernel is the lowest level of software running on the system, with full access to all of the capabilities of the hardware.

optimal cache replacement
Replace whichever block is used farthest in the future.

overhead
The added resource cost of implementing an abstraction versus using the underlying hardware resources directly.

ownership design pattern
A technique for managing concurrent access to shared objects in which at most one thread owns an object at any time, and therefore the thread can access the shared data without a lock.

page coloring
The assignment of physical page frames to virtual addresses by partitioning frames based on which portions of the cache they will use.

page fault
A hardware trap to the operating system kernel when a process references a virtual address with an invalid page table entry.

page frame
An aligned, fixed-size chunk of physical memory that can hold a virtual page.

paged memory
A hardware address translation mechanism where memory is allocated in aligned, fixed-sized chunks, called pages. Any virtual page can be assigned to any physical page frame.

paged segmentation

A hardware mechanism where physical memory is allocated in page frames, but virtual addresses are segmented.

pair of stubs
A pair of short procedures that mediate between two execution contexts.

paravirtualization
A virtual machine abstraction that allows the guest operating system to make system calls into the host operating system to perform hardware-specific operations, such as changing a page table entry.

parent process
A process that creates another process. See also: *child process*.

path
The string that identifies a file or directory.

PCB
See: *process control block*.

PCM
See: *phase change memory*.

performance predictability
Whether a system’s response time or other performance metric is consistent over time.

persistent data
Data that is stored until it is explicitly deleted, even if the computer storing it crashes or loses power.

persistent storage
See: *non-volatile storage*.

phase change behavior
Abrupt changes in a program’s working set, causing bursty cache miss rates: periods of low cache misses interspersed with periods of high cache misses.

phase change memory
A type of non-volatile memory that uses the phase of a material to represent a data bit. See also: *PCM*.

physical address
An address in physical memory.

physical separation
A backup storage policy where the backup is stored at a different location than the primary storage.

physically addressed cache
A processor cache that is accessed using physical memory addresses.

pin
To bind a virtual resource to a physical resource, such as a thread to a processor or a virtual page to a physical page.

platter
A single thin round plate that stores information in a magnetic disk, often on both surfaces.

policy-mechanism separation
A system design principle where the implementation of an abstraction is independent of the resource allocation policy of how the abstraction is used.

polling
An alternative to hardware interrupts, where the processor waits for an asynchronous event to occur, by looping, or busy-waiting, until the event occurs.

portability
The ability of software to work across multiple hardware platforms.
precise interrupts
All instructions that occur before the interrupt or exception, according to the program execution, are completed by the hardware before the interrupt handler is invoked.
preemption
When a scheduler takes the processor away from one task and gives it to another.
preemptive multi-threading
The operating system scheduler may switch out a running thread, e.g., on a timer interrupt, without any explicit action by the thread to relinquish control at that point.
prefetch
To bring data into a cache before it is needed.
principle of least privilege
System security and reliability are enhanced if each part of the system has exactly the privileges it needs to do its job and no more.
priority donation
A solution to priority inversion: when a thread waits for a lock held by a lower priority thread, the lock holder is temporarily increased to the waiter's priority until the lock is released.
priority inversion
A scheduling anomaly that occurs when a high priority task waits indefinitely for a resource (such as a lock) held by a low priority task, because the low priority task is waiting in turn for a resource (such as the processor) held by a medium priority task.
privacy
Data stored on a computer is only accessible to authorized users.
privileged instruction
Instruction available in kernel mode but not in user mode.
process
The execution of an application program with restricted rights — the abstraction for protection provided by the operating system kernel.
process control block
A data structure that stores all the information the operating system needs about a particular process: e.g., where it is stored in memory, where its executable image is on disk, which user asked it to start executing, and what privileges the process has. See also: *PCB*.
process migration
The ability to take a running program on one system, stop its execution, and resume it on a different machine.
processor exception
A hardware event caused by user program behavior that causes a transfer of control to a kernel handler. For example, attempting to divide by zero causes a processor exception in many architectures.
processor scheduling policy
When there are more runnable threads than processors, the policy that determines which threads to run first.
processor status register
A hardware register containing flags that control the operation of the processor, including the privilege level.
producer-consumer communication

Interprocess communication where the output of one process is the input of another.

proprietary system
A system that is under the control of a single company; it can be changed at any time by its provider to meet the needs of its customers.

protection
The isolation of potentially misbehaving applications and users so that they do not corrupt other applications or the operating system itself.

publish
For a read-copy-update lock, a single, atomic memory write that updates a shared object protected by the lock. The write allows new reader threads to observe the new version of the object.

queueing delay
The time a task waits in line without receiving service.

quiescent
For a read-copy-update lock, no reader thread that was active at the time of the last modification is still active.

race condition
When the behavior of a program relies on the interleaving of operations of different threads.

RAID
A Redundant Array of Inexpensive Disks (RAID) is a system that spreads data redundantly across multiple disks in order to tolerate individual disk failures.

RAID 1
See: *mirroring*.

RAID 5
See: *rotating parity*.

RAID 6
See: *dual redundancy array*.

RAID strip
A set of several sequential blocks placed on one disk by a RAID block placement algorithm.

RAID stripe
A set of RAID strips and their parity strip.

R-CSCAN
A variation of the CSCAN disk scheduling policy in which the disk takes into account rotation time.

RCU
See: *read-copy-update*.

read disturb error
Reading a flash memory cell a large number of times can cause the data in surrounding cells to become corrupted.

read-copy-update
A synchronization abstraction that allows concurrent access to a data structure by multiple readers and a single writer at a time. See also: *RCU*.

readers/writers lock
A lock which allows multiple “reader” threads to access shared data concurrently provided they never modify the shared data, but still provides mutual exclusion whenever a “writer” thread is reading or modifying the shared data.

ready list

The set of threads that are ready to be run but which are not currently running.

real-time constraint
The computation must be completed by a deadline if it is to have value.

recoverable virtual memory
The abstraction of persistent memory, so that the contents of a memory segment can be restored after a failure.

redo logging
A way of implementing a transaction by recording in a log the set of writes to be executed when the transaction commits.

relative path
A file path name interpreted as beginning with the process's current working directory.

reliability
A property of a system that does exactly what it is designed to do.

request parallelism
Parallel execution on a server that arises from multiple concurrent requests.

resident attribute
In NTFS, an attribute record whose contents are stored directly in the master file table.

response time
The time for a task to complete, from when it starts until it is done.

restart
The resumption of a process from a checkpoint, e.g., after a failure or for debugging.

roll back
The outcome of a transaction where none of its updates occur.

root directory
The top-level directory in a file system.

root inode
In a copy-on-write file system, the inode table's inode: the disk block containing the metadata needed to find the inode table.

rotating parity
A system for redundantly storing data on disk where the system writes several blocks of data across several disks, protecting those blocks with one redundant block stored on yet another disk. See also: *RAID 5*.

rotational latency
Once the disk head has settled on the right track, it must wait for the target sector to rotate under it.

round robin
A scheduling policy that takes turns running each ready task for a limited period before switching to the next task.

R-SCAN
A variation of the SCAN disk scheduling policy in which the disk takes into account rotation time.

safe state
In the context of deadlock, a state of an execution such that regardless of the sequence of future resource requests, there is at least one safe sequence of decisions as to when to satisfy requests such that all pending and future requests are met.

safety property
A constraint on program behavior such that it never computes the wrong result. Compare: *liveness property*.

sample bias

A measurement error that occurs when some members of a group are less likely to be included than others, and where those members differ in the property being measured.

sandbox
A context for executing untrusted code, where protection for the rest of the system is provided in software.

SCAN
A disk scheduling policy where the disk arm repeatedly sweeps from the inner to the outer tracks and back again, servicing each pending request whenever the disk head passes that track.

scheduler activations
A multiprocessor scheduling policy where each application is informed of how many processors it has been assigned and whenever the assignment changes.

scrubbing
A technique for reducing non-recoverable RAID errors by periodically scanning for corrupted disk blocks and reconstructing them from the parity block.

secondary bottleneck
A resource with relatively low contention, due to a large amount of queueing at the primary bottleneck. If the primary bottleneck is improved, the secondary bottleneck will have much higher queueing delay.

sector
The minimum amount of a disk that can be independently read or written.

sector failure
A magnetic disk error where data on one or more individual sectors of a disk are lost, but the rest of the disk continues to operate correctly.

sector sparing
Transparently hiding a faulty disk sector by remapping it to a nearby spare sector.

security
A computer’s operation cannot be compromised by a malicious attacker.

security enforcement
The mechanism the operating system uses to ensure that only permitted actions are allowed.

security policy
What operations are permitted — who is allowed to access what data, and who can perform what operations.

seek
The movement of the disk arm to re-position it over a specific track to prepare for a read or write.

segmentation
A virtual memory mechanism where addresses are translated by table lookup, where each entry in the table is to a variable-size memory region.

segmentation fault
An error caused when a process attempts to access memory outside of one of its valid memory regions.

segment-local address
An address that is relative to the current memory segment.

self-paging
A resource allocation policy for allocating page frames among processes; each page replacement is taken from a page frame already assigned to the process causing the page fault.

semaphore
A type of synchronization variable with only two atomic operations, P() and V(). P waits for the value of the semaphore to be positive, and then atomically decrements it. V atomically increments the value, and if any threads are waiting in P, triggers the completion of the P operation.

serializability
The result of any program execution is equivalent to an execution in which requests are processed one at a time in some sequential order.

service time
The time it takes to complete a task at a resource, assuming no waiting.

set associative cache
The cache is partitioned into sets of entries. Each memory location can only be stored in its assigned set, by it can be stored in any cache entry in that set. On a lookup, the system needs to check the address against all the entries in its set to determine if there is a cache hit.

settle
The fine-grained re-positioning of a disk head after moving to a new track before the disk head is ready to read or write a sector of the new track.

shadow page table
A page table for a process inside a virtual machine, formed by constructing the composition of the page table maintained by the guest operating system and the page table maintained by the host operating system.

shared object
An object (a data structure and its associated code) that can be accessed safely by multiple concurrent threads.

shell
A job control system implemented as a user-level process. When a user types a command to the shell, it creates a process to run the command.

shortest job first
A scheduling policy that performs the task with the least remaining time left to finish.

shortest positioning time first
A disk scheduling policy that services whichever pending request can be handled in the minimum amount of time. See also: *SPTF*.

shortest seek time first
A disk scheduling policy that services whichever pending request is on the nearest track. Equivalent to shortest positioning time first if rotational positioning is not considered. See also: *SSTF*.

SIMD (single instruction multiple data) programming
See data parallel programming

simultaneous multi-threading
A hardware technique where each processor simulates two (or more) virtual processors, alternating between them on a cycle-by-cycle basis. See also: *hyperthreading*.

single-threaded program
A program written in a traditional way, with one logical sequence of steps as each instruction follows the previous one. Compare: *multi-threaded program*.

slip sparing
When remapping a faulty disk sector, remapping the entire sequence of disk sectors between the faulty sector and the spare sector by one slot to preserve sequential

access performance.

soft link
A directory entry that maps one file or directory name to another. See also: *symbolic link*.

software transactional memory (STM)
A system for general-purpose transactions for in-memory data structures.

software-loaded TLB
A hardware TLB whose entries are installed by software, rather than hardware, on a TLB miss.

solid state storage
A persistent storage device with no moving parts; it stores data using electrical circuits.

space sharing
A multiprocessor allocation policy that assigns different processors to different tasks.

spatial locality
Programs tend to reference instructions and data near those that have been recently accessed.

spindle
The axle of rotation of the spinning disk platters making up a disk.

spinlock
A lock where a thread waiting for a BUSY lock “spins” in a tight loop until some other thread makes it FREE.

SPTF
See: *shortest positioning time first*.

SSTF
See: *shortest seek time first*.

stable property
A property of a program, such that once the property becomes true in some execution of the program, it will stay true for the remainder of the execution.

stable storage
See: *non-volatile storage*.

stable system
A queueing system where the arrival rate matches the departure rate.

stack frame
A data structure stored on the stack with storage for one invocation of a procedure: the local variables used by the procedure, the parameters the procedure was called with, and the return address to jump to when the procedure completes.

staged architecture
A staged architecture divides a system into multiple subsystems or stages, where each stage includes some state private to the stage and a set of one or more worker threads that operate on that state.

starvation
The lack of progress for one task, due to resources given to higher priority tasks.

state variable
Member variable of a shared object.

STM
See: *software transactional memory (STM)*.

structured synchronization

A design pattern for writing correct concurrent programs, where concurrent code uses a set of standard synchronization primitives to control access to shared state, and where all routines to access the same shared state are localized to the same logical module.

superpage
: A set of contiguous pages in physical memory that map a contiguous region of virtual memory, where the pages are aligned so that they share the same high-order (superpage) address.

surface
: One side of a disk platter.

surface transfer time
: The time to transfer one or more sequential sectors from (or to) a surface once the disk head begins reading (or writing) the first sector.

swapping
: Evicting an entire process from physical memory.

symbolic link
: See: *soft link*.

synchronization barrier
: A synchronization primitive where n threads operating in parallel check in to the barrier when their work is completed. No thread returns from the barrier until all n check in.

synchronization variable
: A data structure used for coordinating concurrent access to shared state.

system availability
: The probability that a system will be available at any given time.

system call
: A procedure provided by the kernel that can be called from user level.

system reliability
: The probability that a system will continue to be reliable for some specified period of time.

tagged command queueing
: A disk interface that allows the operating system to issue multiple concurrent requests to the disk. Requests are processed and acknowledged out of order. See also: *native command queueing*. See also: *NCQ*.

tagged TLB
: A translation lookaside buffer whose entries contain a process ID; only entries for the currently running process are used during translation. This allows TLB entries for a process to remain in the TLB when the process is switched out.

task
: A user request.

TCB
: See: *thread control block*.

TCQ
: See: *tagged command queueing*.

temporal locality
: Programs tend to reference the same instructions and data that they had recently accessed.

test and test-and-set
: An implementation of a spinlock where the waiting processor waits until the lock is FREE before attempting to acquire it.

thrashing
When a cache is too small to hold its working set. In this case, most references are cache misses, yet those misses evict data that will be used in the near future.
thread
A single execution sequence that represents a separately schedulable task.
thread context switch
Suspend execution of a currently running thread and resume execution of some other thread.
thread control block
The operating system data structure containing the current state of a thread. See also: *TCB*.
thread scheduler
Software that maps threads to processors by switching between running threads and threads that are ready but not running.
thread-safe bounded queue
A bounded queue that is safe to call from multiple concurrent threads.
throughput
The rate at which a group of tasks are completed.
time of check vs. time of use attack
A security vulnerability arising when an application can modify the user memory holding a system call parameter (such as a file name), *after* the kernel checks the validity of the parameter, but *before* the parameter is used in the actual implementation of the routine. Often abbreviated TOCTOU.
time quantum
The length of time that a task is scheduled before being preempted.
timer interrupt
A hardware processor interrupt that signifies a period of elapsed real time.
time-sharing operating system
An operating system designed to support interactive use of the computer.
TLB
See: *translation lookaside buffer*.
TLB flush
An operation to remove invalid entries from a TLB, e.g., after a process context switch.
TLB hit
A TLB lookup that succeeds at finding a valid address translation.
TLB miss
A TLB lookup that fails because the TLB does not contain a valid translation for that virtual address.
TLB shootdown
A request to another processor to remove a newly invalid TLB entry.
TOCTOU
See: *time of check vs. time of use attack*.
track
A circle of sectors on a disk surface.
track buffer
Memory in the disk controller to buffer the contents of the current track even though those sectors have not yet been requested by the operating system.
track skewing

A staggered alignment of disk sectors to allow sequential reading of sectors on adjacent tracks.

transaction
A group of operations that are applied persistently, atomically as a group or not at all, and independently of other transactions.

translation lookaside buffer
A small hardware table containing the results of recent address translations. See also: *TLB*.

trap
A synchronous transfer of control from a user-level process to a kernel-mode handler. Traps can be caused by processor exceptions, memory protection errors, or system calls.

triple indirect block
A storage block containing pointers to double indirect blocks.

two-phase locking
A strategy for acquiring locks needed by a multi-operation request, where no lock can be released before all required locks have been acquired.

uberblock
In ZFS, the root of the ZFS storage system.

UNIX exec
A system call on UNIX that causes the current process to bring a new executable image into memory and start it running.

UNIX fork
A system call on UNIX that creates a new process as a complete copy of the parent process.

UNIX pipe
A two-way byte stream communication channel between UNIX processes.

UNIX signal
An asynchronous notification to a running process.

UNIX stdin
A file descriptor set up automatically for a new process to use as its input.

UNIX stdout
A file descriptor set up automatically for a new process to use as its output.

UNIX wait
A system call that pauses until a child process finishes.

unsafe state
In the context of deadlock, a state of an execution such that there is at least one sequence of future resource requests that leads to deadlock no matter what processing order is tried.

upcall
An event, interrupt, or exception delivered by the kernel to a user-level process.

use bit
A status bit in a page table entry recording whether the page has been recently referenced.

user-level memory management
The kernel assigns each process a set of page frames, but how the process uses its assigned memory is left up to the application.

user-level page handler
An application-specific upcall routine invoked by the kernel on a page fault.

user-level thread
A type of application thread where the thread is created, runs, and finishes without calls into the operating system kernel.

user-mode operation
The processor operates in a restricted mode that limits the capabilities of the executing process. Compare: *kernel-mode operation*.

utilization
The fraction of time a resource is busy.

virtual address
An address that must be translated to produce an address in physical memory.

virtual machine
An execution context provided by an operating system that mimics a physical machine, e.g., to run an operating system as an application on top of another operating system.

virtual machine honeypot
A virtual machine constructed for the purpose of executing suspect code in a safe environment.

virtual machine monitor
See: *host operating system*.

virtual memory
The illusion of a nearly infinite amount of physical memory, provided by demand paging of virtual addresses.

virtualization
Provide an application with the illusion of resources that are not physically present.

virtually addressed cache
A processor cache which is accessed using virtual, rather than physical, memory addresses.

volume
A collection of physical storage blocks that form a logical storage device (e.g., a logical disk).

wait while holding
A necessary condition for deadlock to occur: a thread holds one resource while waiting for another.

wait-free data structures
Concurrent data structure that guarantees progress for every thread: every method finishes in a finite number of steps, regardless of the state of other threads executing in the data structure.

waiting list
The set of threads that are waiting for a synchronization event or timer expiration to occur before becoming eligible to be run.

wear leveling
A flash memory management policy that moves logical pages around the device to ensure that each physical page is written/erased approximately the same number of times.

web proxy cache
A cache of frequently accessed web pages to speed web access and reduce network traffic.

work-conserving scheduling policy
A policy that never leaves the processor idle if there is work to do.

working set
The set of memory locations that a program has referenced in the recent past.

workload
A set of tasks for some system to perform, along with when each task arrives and how long each task takes to complete.

wound wait
An approach to deadlock recovery that ensures progress by aborting the most recent transaction in any deadlock.

write acceleration
Data to be stored on disk is first written to the disk’s buffer memory. The write is then acknowledged and completed in the background.

write-back cache
A cache where updates can be stored in the cache and only sent to memory when the cache runs out of space.

write-through cache
A cache where updates are sent immediately to memory.

zero-copy I/O
A technique for transferring data across the kernel-user boundary without a memory-to-memory copy, e.g., by manipulating page table entries.

zero-on-reference
A method for clearing memory only if the memory is used, rather than in advance. If the first access to memory triggers a trap to the kernel, the kernel can zero the memory and then resume.

Zipf distribution
The relative frequency of an event is inversely proportional to its position in a rank order of popularity.

About the Authors

Thomas Anderson holds the Warren Francis and Wilma Kolm Bradley Chair of Computer Science and Engineering at the University of Washington, where he has been teaching computer science since 1997.

Professor Anderson has been widely recognized for his work, receiving the Diane S. McEntyre Award for Excellence in Teaching, the USENIX Lifetime Achievement Award, the IEEE Koji Kobayashi Computers and Communications Award, the ACM SIGOPS Mark Weiser Award, the USENIX Software Tools User Group Award, the IEEE Communications Society William R. Bennett Prize, the NSF Presidential Faculty Fellowship, and the Alfred P. Sloan Research Fellowship. He is an ACM Fellow. He has served as program co-chair of the ACM SIGCOMM Conference and program chair of the ACM Symposium on Operating Systems Principles (SOSP). In 2003, he helped co-found the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI).

Professor Anderson's research interests span all aspects of building practical, robust, and efficient computer systems, including operating systems, distributed systems, computer networks, multiprocessors, and computer security. Over his career, he has authored or co-authored over one hundred peer-reviewed papers; nineteen of his papers have won best paper awards.

Michael Dahlin is a Principal Engineer at Google. Prior to that, from 1996 to 2014, he was a Professor of Computer Science at the University of Texas in Austin, where he taught operating systems and other subjects and where he was awarded the College of Natural Sciences Teaching Excellence Award.

Professor Dahlin's research interests include Internet- and large-scale services, fault tolerance, security, operating systems, distributed systems, and storage systems.

Professor Dahlin's work has been widely recognized. Over his career, he has authored over seventy peer reviewed papers; ten of which have won best paper awards. He is both an ACM Fellow and an IEEE Fellow, and he has received an Alfred P. Sloan Research Fellowship and an NSF CAREER award. He has served as the program chair of the ACM Symposium on Operating Systems Principles (SOSP), co-chair of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI), and co-chair of the International World Wide Web conference (WWW).