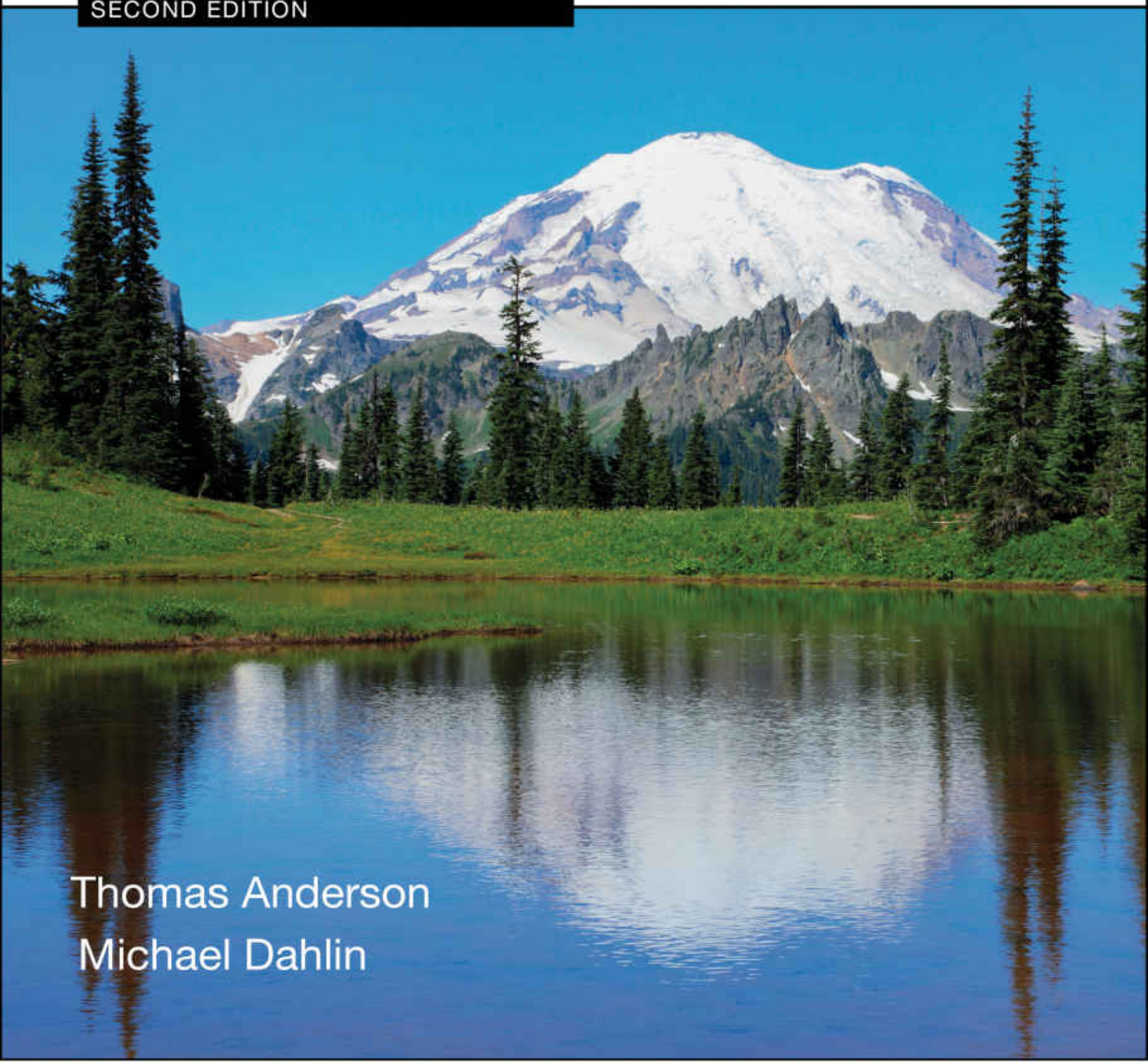
Operating Systems

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

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.

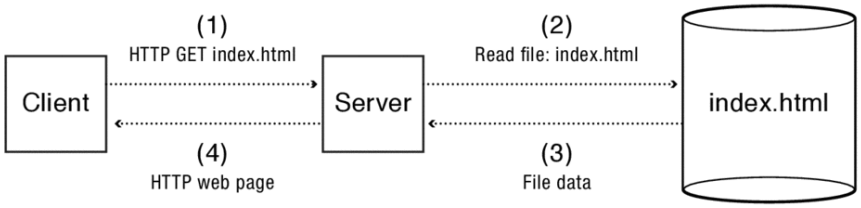


Figure 1.1: The operation of a web server. The client machine sends an HTTP GET request to the web server. The server decodes the packet, reads the file, and sends the contents back to the client.

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.

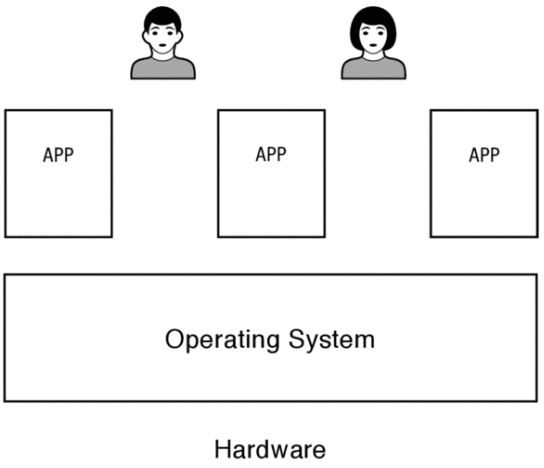


Figure 1.2: A general-purpose operating system is a layer of software that manages a computer's resources for its users and applications.

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:

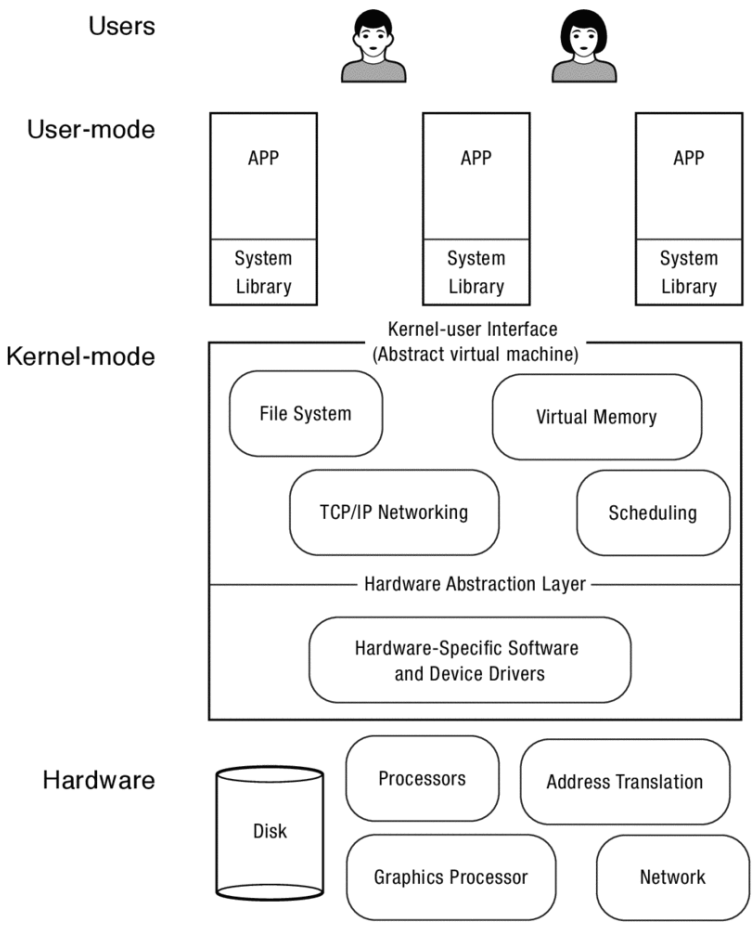


Figure 1.3: This shows the structure of a general-purpose operating system, as an expansion on the simple view presented in Figure 1.2. At the lowest level, the hardware provides processors, memory, and a set of devices for storing data and communicating with the outside world. The hardware also provides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

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.

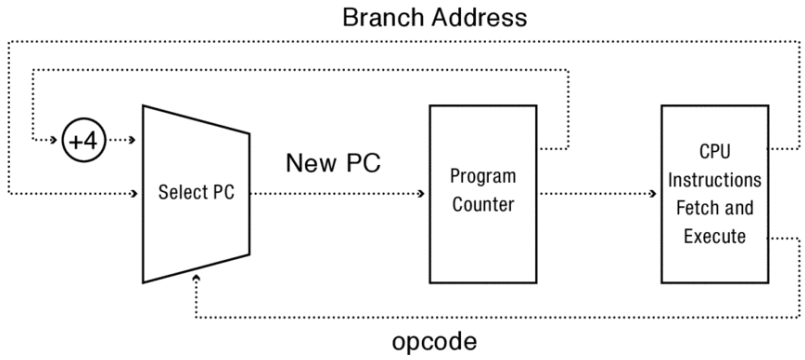


Figure 2.3: The basic operation of a CPU. Opcode, short for operation code, is the decoded instruction to be executed, e.g., branch, memory load, or arithmetic operation.

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.

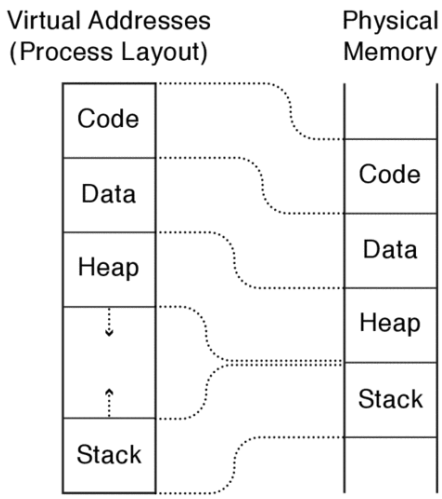


Figure 2.6: Virtual addresses allow the stack and heap regions of a process to grow independently. To grow the heap, the operating system can move the heap in physical memory without changing the heap's virtual address.

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

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

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.

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.

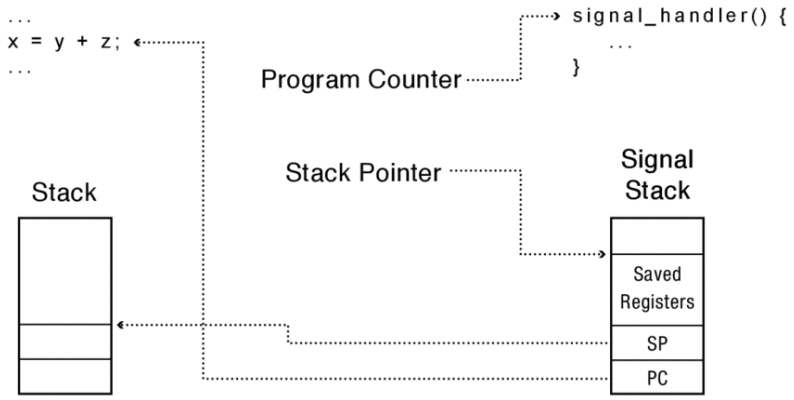


Figure 2.17: The state of the user program and signal handler during a UNIX signal. The signal stack stores the state of the hardware registers at the point where the process was interrupted, with room for the signal handler to execute on the signal stack.

- **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.

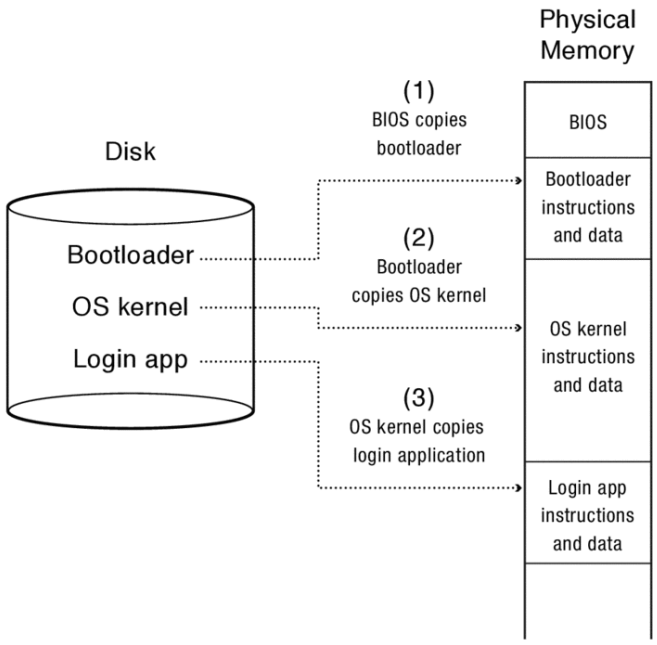


Figure 2.18: The boot ROM copies the bootloader image from disk into memory, and the bootloader copies

- **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!

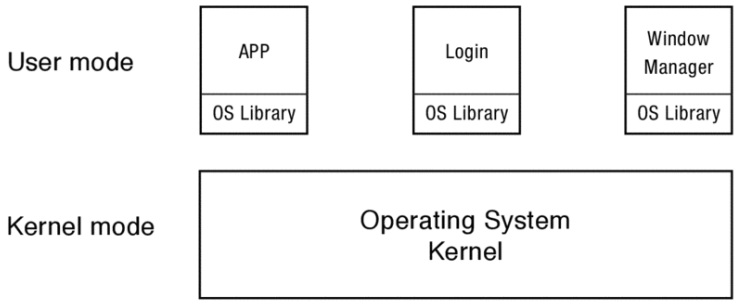


Figure 3.1: Operating system functionality can be implemented in user-level programs, in user-level libraries, in the kernel itself, or in a user-level server invoked by the kernel.

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.

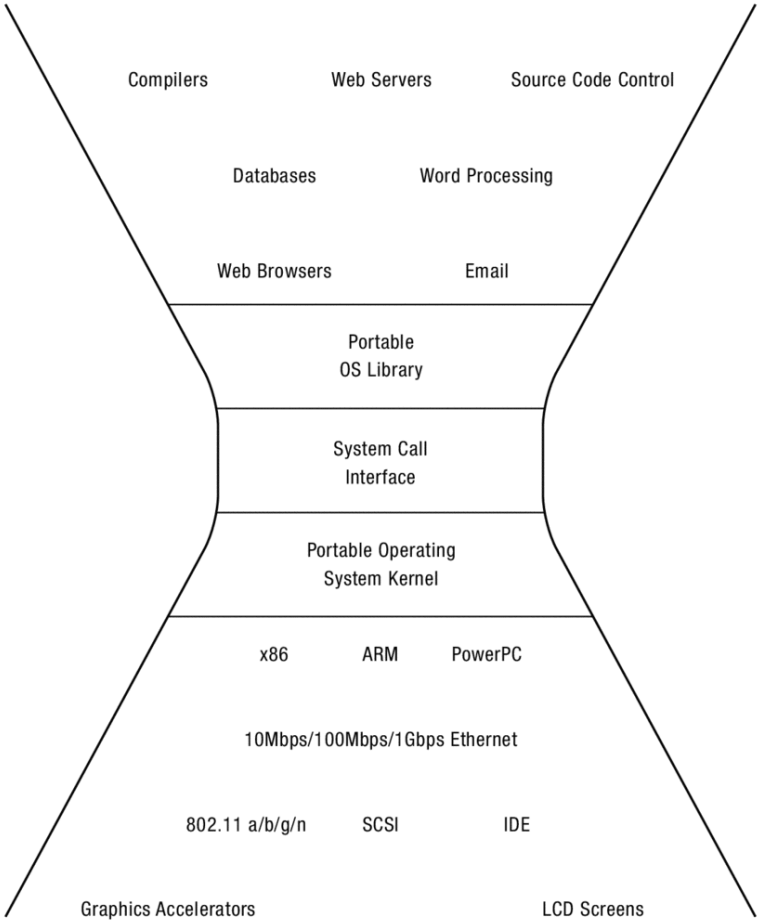


Figure 3.2: The kernel system call interface can be seen as a “thin waist,” enabling independent evolution of applications and hardware.

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

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*.

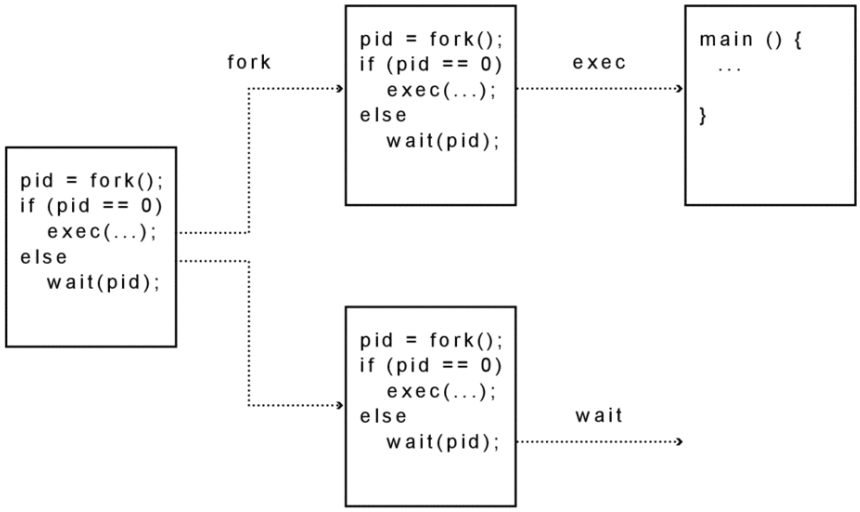


Figure 3.4: The operation of the UNIX fork and exec system calls. UNIX fork makes a copy of the parent process; UNIX exec changes the child process to run the new program.

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:

- **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:

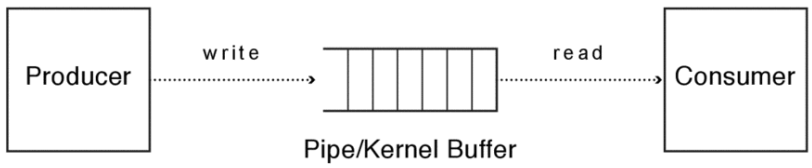


Figure 3.6: A pipe is a temporary kernel buffer connecting a process producing data with a process consuming the data.

- **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.

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.