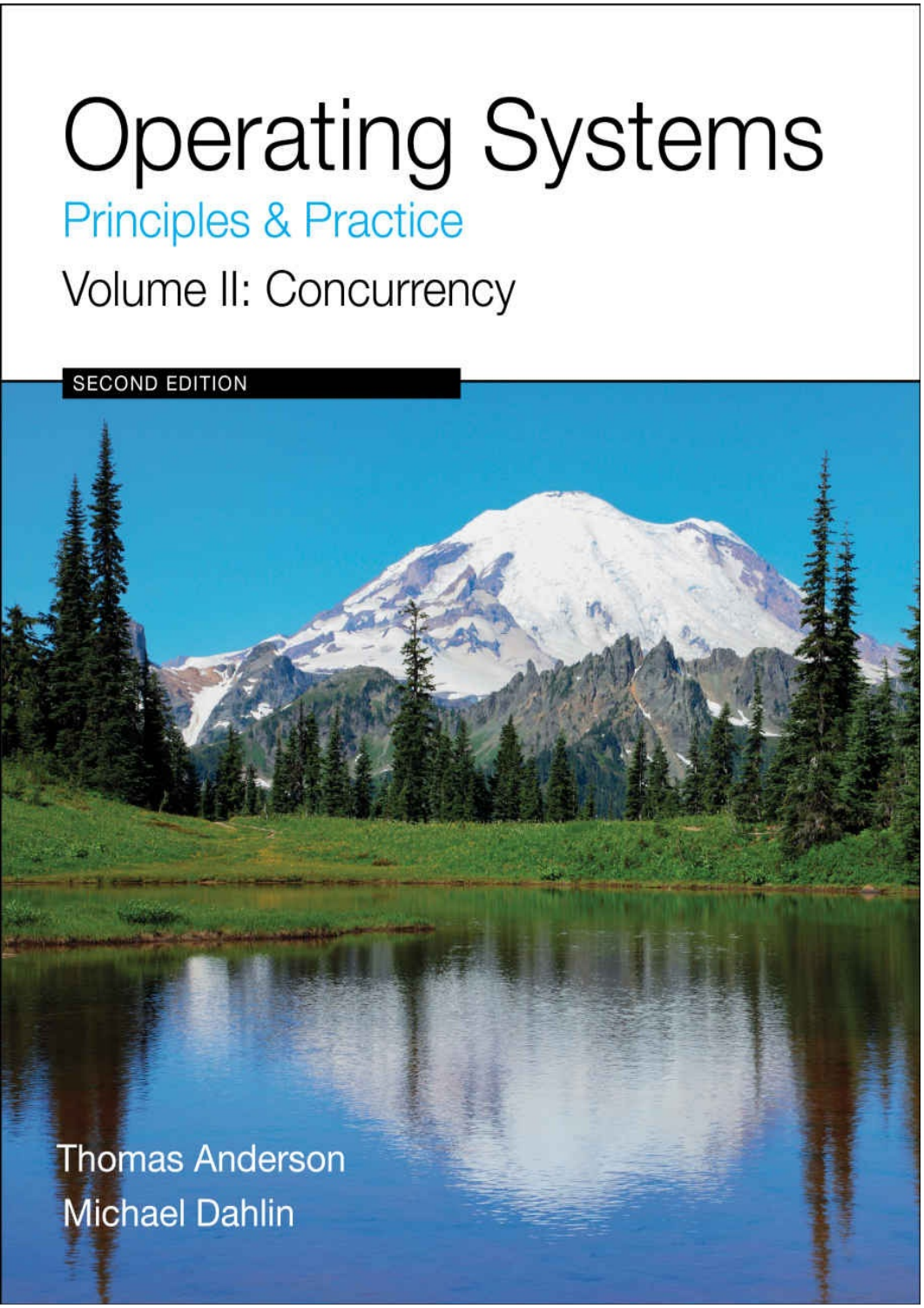

Operating Systems

Principles & Practice

Volume II: Concurrency

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 II: Concurrency by Thomas Anderson and Michael Dahlin

ISBN 978-0-9856735-4-3
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 would also like to thank Simon Peter for running the multiprocessor experiments introducing Chapter 6.

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, Rasit Eskicioglu, 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.

II
Concurrency

4. Concurrency and Threads

Many hands make light work. —*John Heywood (1546)*

In the real world — outside of computers — different activities often proceed at the same time. Five jazz musicians play their instruments while reacting to each other; one car drives north while another drives south; one part of a drug molecule is attracted to a cell's receptor, while another part is repelled; a humanoid robot walks, raises its arms, and turns its head; you fetch one article from the *New York Times* website while someone else fetches another; or millions of people make long distance phone calls on Mother's Day.

We use the word *concurrency* to refer to multiple activities that can happen at the same time. The real world is concurrent, and internally, modern computers are also concurrent. For example, a high-end server might have more than a dozen processors, 10 disks, and 4 network interfaces; a workstation might have a dozen active I/O devices including a screen, keyboard, mouse, camera, microphone, speaker, wireless network interface, wired network interface, printer, scanner, and disk drive. Today, even mobile phones often have multi-core processors.

Correctly managing concurrency is a key challenge for operating system developers. To manage hardware resources, to provide responsiveness to users, and to run multiple applications simultaneously, the operating system needs a structured way of keeping track of the various actions it needs to perform. Over the next several chapters, we will present a set of abstractions for expressing and managing concurrency. These abstractions are in widespread use in commercial operating systems because they reduce implementation complexity, improve system reliability, and improve performance.

Concurrency is also a concern for many application developers. Although the abstractions we discuss were originally developed to make it easier to write correct operating system code, they have become widely used in applications:

- Network services need to be able to handle multiple requests from their clients; a Google that could handle only one search request at a time, or an Amazon that could only allow one book to be bought at a time, would be much less useful.
- Most applications today have user interfaces; providing good responsiveness to users while simultaneously executing application logic is much easier with a structured approach to concurrency.
- Parallel programs need to be able to map work onto multiple processors to get the performance benefits of multicore architectures.
- Data management systems need concurrency to mask the latency of disk and network operations.

From the programmer's perspective, it is much easier to think sequentially than to keep track of many simultaneous activities. For example, when reading or writing the code for a

procedure, you can identify an initial state and a set of pre-conditions, think through how each successive statement changes the state, and from that determine the post-conditions. How can you write a correct program with dozens of events happening at once?

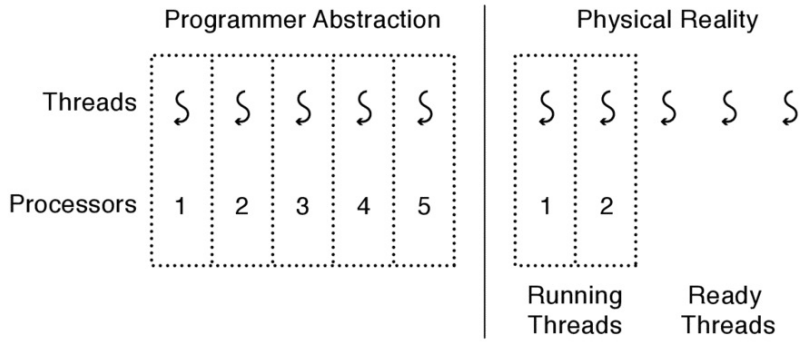

Figure 4.1: The operating system provides the illusion that programmers can create as many threads as they need, and each thread runs on its own dedicated virtual processor. In reality, of course, a machine only has a finite number of processors, and it is the operating system's job to transparently multiplex threads onto the actual processors.

The key idea is to write a concurrent program — one with many simultaneous activities — as a set of sequential streams of execution, or *threads*, that interact and share results in very precise ways. Threads let us define a set of tasks that run concurrently while the code for each task is sequential. Each thread behaves as if it has its own dedicated processor, as illustrated in Figure 4.1. As we will see later, using the thread abstraction often requires the programmer to write additional code for coordinating multiple threads accessing shared data structures; we will discuss this topic in much more detail in Chapter 5.

The thread abstraction lets the programmer create as many threads as needed without worrying about the exact number of physical processors, or exactly which processor is doing what at each instant. Of course, threads are only an abstraction: the physical hardware has a limited number of processors (and potentially only one!). The operating system's job is to provide the illusion of a nearly infinite number of virtual processors even while the physical hardware is more limited. It sustains this illusion by transparently suspending and resuming threads so that at any given time only a subset of the threads are actively running.

This chapter will define the thread abstraction, illustrate how a programmer can use the abstraction, and explain how the operating system can implement threads on top of a limited number of processors. Chapter 5 explains how to coordinate threads when they operate on shared data, and Chapter 6 covers advanced issues when programming with threads. Chapter 7 discusses the policy question: how should the operating system choose *which* thread to run next when there are more things to run than processors on which to run them.

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

- **Thread Use Cases.** What are threads useful for? (Section 4.1)
- **Thread Abstraction.** What is the thread abstraction as seen by a programmer? (Section 4.2)
- **Simple Thread API.** How can programmers use threads? (Section 4.3)
- **Thread Data Structures.** What data structures does the operating system use to manage threads? (Section 4.4)
- **Thread Life Cycle.** What states does a thread go through between initialization and completion? (Section 4.5)
- **Implementing Kernel Threads.** How do we implement the thread abstraction inside the operating system kernel? (Section 4.6)
- **Combining Kernel Threads and Single-Threaded User Processes.** How do we extend the implementation of kernel threads to support simple single-threaded processes? (Section 4.7)
- **Implementing Multi-threaded Processes.** How do we implement the thread abstraction for multi-threaded applications? (Section 4.8)
- **Alternative Abstractions.** What other abstractions can we use to express and implement concurrency? (Section 4.9)

Deja vu all over again?

Threads are widely used, and several modern programming languages directly support writing programs with multiple threads. You may have programmed with threads before or have taken classes that talk about using threads. What is new here?

The discussion in this book is designed to make sense even if you have never seen threads before. If you have seen threads before, great! But we still think you will find the discussion useful.

Beyond describing the basic thread abstraction, we emphasize two points in this chapter and the following ones.

- **Implementation.** We will describe how operating systems implement threads both for their own use and for use by user-level applications. It is important to understand how threads really work so that you can understand their costs and performance characteristics and can use them effectively.
- **Practice.** We will present a methodology for writing correct multi-threaded programs. Concurrency is increasingly important in many programming tasks, but writing correct multi-threaded programs requires much more care and discipline than writing correct single-threaded programs. That said, following a few simple rules that we will describe can greatly simplify the process of writing robust multi-threaded code.

Multithreaded programming has a well-deserved reputation for being difficult, but we

believe the ideas in this chapter and the subsequent ones can help almost anyone become better at programming with threads.

4.1 Thread Use Cases

The intuition behind the thread abstraction is simple: in a program, we can represent each concurrent task as a *thread*. Each thread provides the abstraction of sequential execution similar to the traditional programming model. In fact, we can think of a traditional program as *single-threaded* with one logical sequence of steps as each instruction follows the previous one. The program executes statements, iterates through loops, and calls/returns from procedures one after another.

A *multi-threaded program* is a generalization of the same basic programming model. Each individual thread follows a single sequence of steps as it executes statements, iterates through loops, calls/returns from procedures, etc. However, a program can now have several such threads executing at the same time.

When is it appropriate to use multiple threads within the same program? Threads have become widely used in both operating system and application code, and based on that experience, we can identify several common themes. We illustrate these themes by describing one application in some detail, to show how and why it leverages threads.

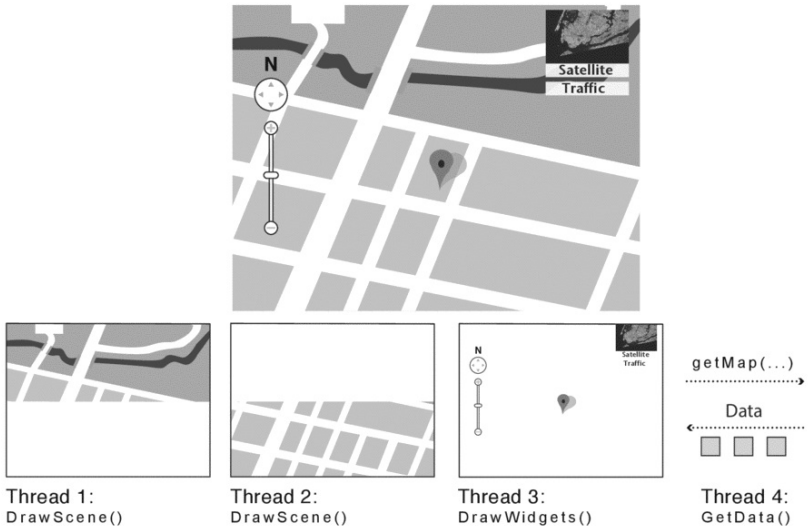

Figure 4.2: In the Earth Visualizer example, two threads each draw part of the scene, a third thread manages the user interface widgets, and a fourth thread fetches new data from a remote server. Satellite Image Credit: NASA Earth Observatory.

EXAMPLE: Consider an Earth Visualizer application similar to Google Earth (http://earth.google.com/). This application lets a user virtually fly anywhere in the world, see aerial images at different resolutions, and view other information associated with each location. A key part of the design is that the user's controls are always operable: when the user moves the mouse to a new location, the image is redrawn in the background at successively better resolutions while the program continues to let the user adjust the view, select additional information about the location for display, or enter search terms.

To implement this application, as Figure 4.2 illustrates, the programmer might write code to draw a portion of the screen, display user interface (UI) widgets, process user inputs, and fetch higher resolution images for newly visible areas. In a sequential program, these functions would run in turn. With threads, they can run concurrently so that the user interface is responsive even while new data is being fetched and the screen being redrawn.

4.1.1 Four Reasons to Use Threads

Using threads to express and manage concurrency has several advantages:

- **Program structure: expressing logically concurrent tasks.** Programs often interact with or simulate real-world applications that have concurrent activities. Threads let you express an application's natural concurrency by writing each concurrent task as a separate thread.

  In the Earth Visualizer application, threads let different activities — updating the screen, fetching additional data, and receiving new user inputs — run at the same time. For example, to get mouse input while also re-drawing the screen and sending and receiving packets off the network, the physical processors need to split their time among these tasks.

  Although one could imagine manually writing a program that interleaves these activities (e.g., draw a few pixels on the screen, then check to see if the user has moved the mouse, then check to see if new image data have arrived on the network, …), using threads greatly simplifies concurrent code.

  Another example is on the server side of the Earth Visualizer. The server needs to manage the requests of a large number of clients, each focused on a different point on the planet. Since the clients are likely behind a wide variety of access link technologies (e.g., from dialup to gigabit Ethernet), it would slow everyone down if each request needed to be completely handled before the server could start on the next one. By creating a separate thread for each client, the computation and networking needed for that client can be intermixed with other clients, without affecting the logical structure of the program. This design pattern — one server thread per client — is common; for example, the popular Apache web server assigns each client its own thread when it first connects to the server.

- **Responsiveness: shifting work to run in the background.** To improve user responsiveness and performance, a common design pattern is to create threads to perform work in the background, without the user waiting for the result. This way,

the user interface can remain responsive to further commands, regardless of the complexity of the user request. In a web browser, for example, the cancel button should continue to work even (or especially!) if the downloaded page is gigantic or a script on the page takes a long time to execute.

How does this work? Many applications have a loop: get a user command, then execute the command, then get the next command. If some commands take a long time to perform, however, an application that executes everything sequentially will not be able to check for the next operation until the previous one completes. To keep the interface responsive, we can use threads to split each command into two parts: anything that can be done instantly can be done in the main event loop, and a separate thread can perform the rest of the task in the background. In the Earth Visualizer example, we used threads to move the computationally difficult parts of the application logic — rendering the display — out of the main loop.

Operating system kernels make extensive use of threads to preserve responsiveness. Many operating systems are designed so that the common case is fast. For example, when writing a file, the operating system stores the modified data in a kernel buffer, and returns immediately to the application. In the background, the operating system kernel runs a separate thread to flush the modified data out to disk. Another example is on file reads: the kernel can have a thread which attempts to anticipate which blocks are likely to be read next (e.g., if the application is reading a large file from beginning to end), and to bring those blocks from disk before the application asks for them.

- **Performance: exploiting multiple processors.** Programs can use threads on a multiprocessor to do work in parallel; they can do the same work in less time or more work in the same elapsed time. Today, a server might have more than a dozen processors; a desktop or laptop may include eight processor cores; even most smartphones are multicore machines. Looking forward, Moore's law makes it likely that the number of processors per system will continue to increase. An advantage to using threads for parallelism is that the number of threads need not exactly match the number of processors in the hardware on which it is running. The operating system transparently switches which threads run on which processors.

  For an 8-processor machine, you could parallelize the Earth Visualizer application by splitting the demanding job of rendering different portions of the image on the screen across six threads. Then, the operating system could run those six rendering threads on six processors and run the various other threads on the two remaining processors to update the on-screen navigation widgets, construct the network messages needed to fetch additional images from the distant servers, and parse reply messages.

- **Performance: managing I/O devices.** To do useful work, computers must interact with the outside world via I/O devices. By running tasks as separate threads, when one task is waiting for I/O, the processor can make progress on a different task.

  The benefit of concurrency between the processor and the I/O is two-fold: First, processors are often much faster than the I/O systems with which they interact, so keeping the processor idle during I/O would waste much of its capacity. For example, the latency to read from disk can be tens of milliseconds, enough to execute more than 10 million instructions on a modern processor. After requesting a block from disk, the operating system can switch to another program, or another thread within the same program, until the disk completes and the original thread is ready to resume.

  Second, I/O provides a way for the computer to interact with external entities, such as users pressing keys on a keyboard or a remote computer sending network packets. The arrival of this type of I/O event is unpredictable, so the processor must be able to work on other tasks while still responding quickly to these external events.

  In the Earth Visualizer application, a snappy user interface is essential, but much of the imagery is stored on remote servers and fetched by the application only when needed. The application provides a responsive experience when a user changes location by first downloading a small, low-resolution view of the new location. While rendering those images with one thread, another thread simultaneously fetches progressively higher-resolution images, allowing the rendering thread to update the view as the higher-resolution images arrive.

Threads vs. processes

In Chapter 2, we described a process as the execution of a program with restricted rights. A thread is an independent sequence of instructions running within a program. Perhaps the best way to see how these concepts are related, is to see how different operating systems combine them in different ways:

- **One thread per process.** A simple single-threaded application has one sequence of instructions, executing from beginning to end. The operating system kernel runs those instructions in user mode to restrict access to privileged operations or system memory. The process performs system calls to ask the kernel to perform privileged operations on its behalf.
- **Many threads per process.** Alternately, a program may be structured as several concurrent threads, each executing within the restricted rights of the process. At any given time, a subset of the process's threads may be running, while the rest are suspended. Any thread running in a process can make system calls into the kernel, blocking that thread until the call returns but allowing other threads to continue to run. Likewise, when the processor gets an I/O interrupt, it preempts one of the running threads so the kernel can run the interrupt handler; when the handler finishes, the kernel resumes that thread.
- **Many single-threaded processes.** As recently as twenty years ago, many operating systems supported multiple processes but only one thread per process. To the kernel, however, each process looks like a thread: a separate sequence of instructions, executing sometimes in the kernel and sometimes at user level. For example, on a multiprocessor, if multiple processes perform system calls at the same time, the kernel, in effect, has multiple threads executing concurrently in kernel mode.
- **Many kernel threads.** To manage complexity, shift work to the background, exploit parallelism, and hide I/O latency, the operating system kernel itself can benefit from

using multiple threads. In this case, each kernel thread runs with the privileges of the kernel: it can execute privileged instructions, access system memory, and issue commands directly to I/O devices. The operating system kernel itself implements the thread abstraction for its own use.

Because of the usefulness of threads, almost all modern operating systems support both multiple threads per process and multiple kernel threads.

4.2 Thread Abstraction

Thus far, we have described what a thread is and why it is useful. Before we go farther, we must define the thread abstraction and its properties more precisely.

A *thread* is *a single execution sequence* that represents *a separately schedulable task*.

- **Single execution sequence.** Each thread executes a sequence of instructions — assignments, conditionals, loops, procedures, and so on — just as in the familiar sequential programming model.
- **Separately schedulable task.** The operating system can run, suspend, or resume a thread at any time.

4.2.1 Running, Suspending, and Resuming Threads

Threads provide the illusion of an infinite number of processors. How does the operating system implement this illusion? It must execute instructions from each thread so that each thread makes progress, but the underlying hardware has only a limited number of processors, and perhaps only one!

To map an arbitrary set of threads to a fixed set of processors, operating systems include a *thread scheduler* that can switch between threads that are running and those that are ready but not running. For example, in the previous Figure 4.1, a scheduler might suspend thread 1 from processor 1, move it to the list of ready threads, and then resume thread 5 by moving it from the ready list to run on processor 1.

Switching between threads is transparent to the code being executed within each thread. The abstraction makes each thread appear to be a single stream of execution; this means the programmer can pay attention to the sequence of instruction within a thread and not whether or when that sequence may be (temporarily) suspended to let another thread run.

Threads thus provide an execution model in which *each thread runs on a dedicated virtual processor with unpredictable and variable speed.* From the point of view of a thread's code, each instruction appears to execute immediately after the preceding one. However, the scheduler may suspend a thread between one instruction and the next and resume running it later. It is as if the thread were running on a processor that sometimes becomes very slow.

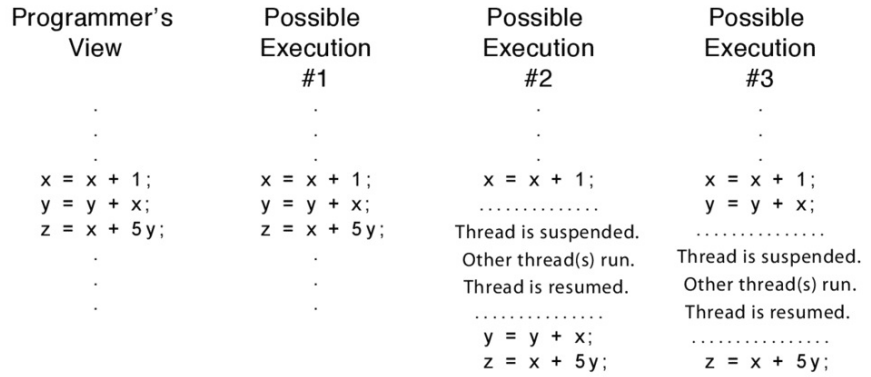

Figure 4.3: Three possible ways that a thread might execute, all of which are equivalent to the programmer.

Figure 4.3 illustrates a programmer's view of a simple program and three (of many) possible ways the program might be executed, depending on what the scheduler does. From the thread's point of view, other than the speed of execution, the alternatives are equivalent. Indeed, the thread would typically be unaware of which of these (or other) executions actually occurs.

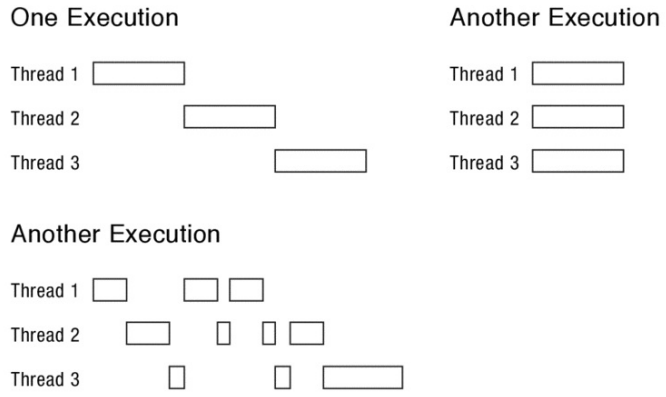

Figure 4.4: Some of the many possible ways that three threads might be interleaved at runtime.

How threads are scheduled affects a thread's interleavings with other threads. Figure 4.4 shows some of the many possible interleavings of a program with three threads. Thread programmers should therefore not make any assumptions about the relative speed with which different threads execute.

Cooperative vs. preemptive multi-threading

Although most thread systems include a scheduler that can — at least in principle — run any thread at any time, some systems provide the abstraction of *cooperative threads*. In these systems, a thread runs without interruption until it explicitly relinquishes control of the processor to another thread. An advantage of cooperative multi-threading is increased control over the interleavings among threads. For example, in most cooperative multi-threading systems, only one thread runs at a time, so while a thread is running, no other thread can run and affect the system's state.

Unfortunately, cooperative multi-threading has significant disadvantages. For example, a long-running thread can monopolize the processor, starving other threads and making the system's user interface sluggish or non-responsive. Additionally, modern multiprocessor machines run multiple threads at a time, so one would still have to reason about the possible interactions between threads even if cooperative multi-threading were used. Thus, although cooperative multi-threading was used in some significant systems in the past, including early versions of Apple's MacOS operating system, it is less often used today.

The alternative we describe in this book is sometimes called *preemptive multi-threading* since running threads can be switched at any time. Whenever the book uses the term "multi-threading," it means preemptive multi-threading unless we explicitly state otherwise.

4.2.2 Why "Unpredictable Speed"?

It may seem strange to require programmers to assume that a thread's virtual processor runs at an unpredictable speed and that any interleaving with other threads is possible. Surely, the programmer should be able to take advantage of the fact that some interleavings are more likely than others?

The thread programming model adopts this assumption as a way to guide programmers when reasoning about correctness. Rather than assuming that one thread runs at the same speed as another (or faster or slower) and trying to write programs that coordinate threads based on their relative speed of execution, multi-threaded programs should make no assumptions about the behavior of the thread scheduler. In turn, the kernel's scheduling decisions — when to assign a thread to a processor, and when to preempt it for a different thread — can be made without worrying whether they might affect program correctness.

If threads are completely independent of each other, sharing no memory or other resources, then the order of execution will not matter — any schedule will produce the same output as any other. Most multi-threaded programs share data structures, however. In this case, as Chapter 5 describes, the programmer must use explicit synchronization to ensure program correctness regardless of the possible interleaving of instructions of different threads.

Even if we could ignore the issue of scheduling — e.g., if there are more processors than threads so that each thread is assigned its own physical processor — the physical reality is that the relative execution speed of different threads can be significantly affected by factors outside their control. An extreme example is that the programmer may be debugging one thread by single-stepping it, while other threads run at full speed on other processors. If the programmer is to have any hope of understanding concurrent program behavior, the program's correctness cannot depend on which threads are being observed.

Variability in execution speed occurs during normal operation as well. Accessing memory can stall a processor for hundreds or thousands of cycles if a cache miss occurs. Other factors include how frequently the scheduler preempts the thread, how many physical processors are present on a machine, how large the caches are, how fast the memory is, how the energy-saving firmware adjusts the processors' clock speeds, what network messages arrive, or what input is received from the user. Execution speeds for the different threads of a program are hard to predict, can vary on different hardware, and can even vary from run to run on the same hardware. As a result, we must coordinate thread actions through explicit synchronization rather than by trying to reason about their relative speed.

EXAMPLE: Is a kernel interrupt handler a thread?

ANSWER: No, an interrupt handler is not a thread. A kernel interrupt handler shares some resemblance to a thread: it is a single sequence of instructions that executes from beginning to end. However, an interrupt handler is not independently schedulable: it is triggered by a hardware I/O event, rather than a decision by the thread scheduler in the kernel. Once started, the interrupt handler runs to completion, unless preempted by another (higher priority) interrupt. □

4.3 Simple Thread API

Simple Threads API

void thread_create (thread, func, arg)	Create a new thread, storing information about it in thread. Concurrently with the calling thread, thread executes the function func with the argument arg.
void thread_yield ()	The calling thread voluntarily gives up the processor to let some other thread(s) run. The scheduler can resume running the calling thread whenever it chooses to do so.
int thread_join (thread)	Wait for thread to finish if it has not already done so; then return the value passed to thread_exit by that thread. Note that thread_join may be called only once for each thread.
void	Finish the current thread. Store the value ret in the current thread's data

thread_exit (ret)	structure. If another thread is already waiting in a call to thread_join, resume it.

Figure 4.5: Simplified API for using threads.

Figure 4.5 shows a simple API for using threads. This simplified API is based on the POSIX standard pthreads API, but it omits some POSIX options and error handling for simplicity. Most other thread packages are quite similar; if you understand how to program with this API, you will find it easy to write code with most standard thread APIs.

A good way to understand the simple threads API is that it provides a way to invoke an *asynchronous procedure call*. A normal procedure call passes a set of arguments to a function, runs the function immediately on the caller's stack, and when the function is completed, returns control back to the caller with the result. An asynchronous procedure call separates the call from the return: with thread_create, the caller starts the function, but unlike a normal procedure call, the caller continues execution concurrently with the called function. Later, the caller can wait for the function completion (with thread_join).

In Chapter 3, we saw similar concepts in the UNIX process abstraction. thread_create is analogous to UNIX process fork and exec, while thread_join is analogous to UNIX process wait. UNIX fork creates a new process that runs concurrently with the process calling fork; UNIX exec causes that process to run a specific program. UNIX wait allows the calling process to suspend execution until the completion of the new process.

4.3.1 A Multi-Threaded Hello World

```
#include <stdio.h>
#include "thread.h"

static void go(int n);

#define NTHREADS 10
static thread_t threads[NTHREADS];

int main(int argc, char **argv) {
    int i;
    long exitValue;

    for (i = 0; i < NTHREADS; i++){
        thread_create(&(threads[i]), &go, i);
    }
    for (i = 0; i < NTHREADS; i++){
        exitValue = thread_join(threads[i]);
        printf("Thread %d returned with %ld\n",
                        i, exitValue);
    }
    printf("Main thread done.\n");
    return 0;
}
```

```
void go(int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
    // Not reached
}
```

```
% ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Figure 4.6: Example multi-threaded program using the simple threads API that prints "Hello" ten times. Also shown is the output of one possible run of this program.

To illustrate how to use the simple threads API, Figure 4.6 shows a very simple multi-threaded program written in 'C'. The main function uses thread_create to create 10 threads. The interesting arguments are the second and third.

- The second argument, go, is a function pointer — where the newly created thread should begin execution.
- The third argument, i, is passed to that function.

Thus, thread_create initializes the i'th thread's state so that it is prepared to call the function go with the argument i.

When the scheduler runs the i'th thread, that thread runs the function go with the value i as an argument and prints Hello from thread i. The thread then returns the value (i + 100) by calling thread_exit. This call stores the specified value in a field in the thread_t object so that thread_join can retrieve it.

The main function uses thread_join to wait for each of the threads it created. As each thread finishes, code in main reads the thread's exit value and prints it.

EXAMPLE: Why might the "Hello" message from thread 2 print *after* the "Hello"

message for thread 5, even though thread 2 was created before thread 5?

ANSWER: Creating and scheduling threads are separate operations. Although threads are usually scheduled in the order that they are created, there is no guarantee. Further, even if thread 2 started running before thread 5, it might be preempted before it reaches the printf call.

Rather, the only assumption the programmer can make is that each of the threads runs on its own virtual processor with unpredictable speed. Any interleaving is possible. □

EXAMPLE: Why must the "Thread returned" message from thread 2 print *before* the Thread returned message from thread 5?

ANSWER: Since the threads run on virtual processors with unpredictable speeds, the order in which the threads finish is indeterminate. However, **the main thread checks for thread completion in the order they were created.** It calls thread_join for thread i +1 only after thread_join for thread i has returned. □

EXAMPLE: What is the *minimum* and *maximum* number of threads that could exist when thread 5 prints "Hello?"

ANSWER: When the program starts, a main thread begins running main. That thread creates NTHREADS = 10 threads. All of those could run and complete before thread 5 prints "Hello." Thus, **the minimum is two threads** — the main thread and thread 5. On the other hand, all 10 threads could have been created, while 5 was the first to run. Thus, **the maximum is 11 threads.** □

4.3.2 Fork-Join Parallelism

Although the interface in Figure 4.5 is simple, it is remarkably powerful. Many multi-threaded applications can be designed using only these thread operations and no additional synchronization. With *fork-join parallelism*, a thread can create child threads to perform work ("fork", or thread_create), and it can wait for their results ("join"). Data may be safely shared between threads, provided it is (a) written by the parent before the child thread starts or (b) written by the child and read by the parent after the join.

If these sharing restrictions are followed, each thread executes independently and in a deterministic fashion, unaffected by the behavior of any other concurrently executing thread. The multiplexing of threads onto processors has no effect other than performance.

```
// To pass two arguments, we need a struct to hold them.
typedef struct bzeroparams {
    unsigned char *buffer;
    int length;
};

#define NTHREADS 10

void go (struct bzeroparams *p) {
    memset(p->buffer, 0, p->length);
}
```

```
// Zero a block of memory using multiple threads.
void blockzero (unsigned char *p, int length) {
    int i;
    thread_t threads[NTHREADS];
    struct bzeroparams params[NTHREADS];

// For simplicity, assumes length is divisible by NTHREADS.
    assert((length % NTHREADS) == 0);
    for (i = 0; i < NTHREADS; i++) {
        params[i].buffer = p + i * length/NTHREADS;
        params[i].length = length/NTHREADS;
        thread_create_p(&(threads[i]), &go, &params[i]);
    }
    for (i = 0; i < NTHREADS; i++) {
        thread_join(threads[i]);
    }
}
```

Figure 4.7: Routine to zero a contiguous region of memory in parallel using multiple threads. To pass two arguments (the pointer to the buffer and the length of the buffer) to the child thread, the program passes a pointer to a struct holding the two parameters.

EXAMPLE: Parallel block zero. A simple example of fork-join parallelism in operating systems is the procedure to zero a contiguous block of memory. To prevent unintentional data leakage, whenever a process exits, the operating system must zero the memory that had been allocated to the exiting process. Otherwise, a new process may be re-assigned the memory, enabling it to read potentially sensitive data. For example, an operating system's remote login program might temporarily store a user's password in memory, but the next process to use the same physical memory might be a memory-scanning program launched by a different, malicious user.

For a large process, parallelizing the zeroing function can make sense. Zeroing 1 GB of memory takes about 50 milliseconds on a modern computer; by contrast, creating and starting a new thread takes a few tens of microseconds.

Figure 4.7 illustrates the code for a parallel zero function using fork-join parallelism. The multi-threaded blockzero creates a set of threads and assigns each a disjoint portion of the memory region; the region is empty when all threads have completed their work.

In practice, the operating system will often create a thread to run blockzero in the background. The memory of an exiting process does not need to be cleared until the memory is needed — that is, when the next process is created.

To exploit this flexibility, the operating system can create a set of low priority threads to run blockzero. The kernel can then return immediately and resume running application code. Later on, when the memory is needed, the kernel can call thread_join. If the zero is complete by that point, the join will return immediately; otherwise, it will wait until the memory is safe to use.

4.4 Thread Data Structures and Life Cycle

As we have seen, each thread represents a sequential stream of execution. The operating system provides the illusion that each thread runs on its own virtual processor by transparently suspending and resuming threads. For the illusion to work, the operating system must precisely save and restore the state of a thread. However, because threads run either in a process or in the kernel, there is also *shared* state that is not saved or restored when switching the processor between threads.

Thus, to understand how the operating system implements the thread abstraction, we must define both the per-thread state and the state that is shared among threads. Then we can describe a thread's life cycle — how the operating system can create, start, stop, and delete threads to provide the abstraction.

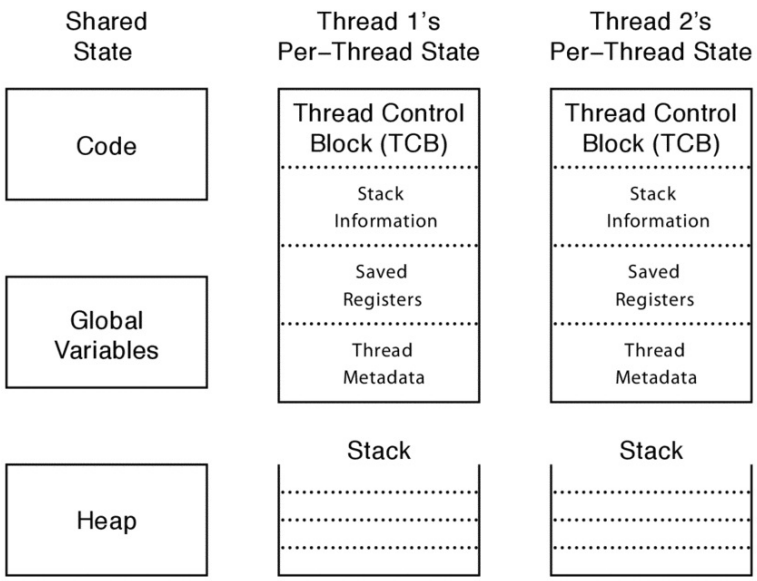

Figure 4.8: A multi-threaded process or operating system kernel has both *per-thread state* and *shared state*. The thread control block stores the per-thread state: the current state of the thread's computation (e.g., saved processor registers and a pointer to the stack) and metadata needed to manage the thread (e.g., the thread's ID, scheduling priority, owner, and resource consumption). Shared state includes the program's code, global static variables, and the heap.

4.4.1 Per-Thread State and Thread Control Block (TCB)

The operating system needs a data structure to represent a thread's state; a thread is like any other object in this regard. This data structure is called the *thread control block* (TCB). For every thread the operating system creates, it creates one TCB.

The thread control block holds two types of per-thread information:

1. The state of the computation being performed by the thread.
2. Metadata about the thread that is used to manage the thread.

Per-thread Computation State. To create multiple threads and to be able to start and stop each thread as needed, the operating system must allocate space in the TCB for the current state of each thread's computation: a pointer to the thread's stack and a copy of its processor registers.

- **Stack.** A thread's stack is the same as the stack for a single-threaded computation — it stores information needed by the nested procedures the thread is currently running. For example, if a thread calls foo(), foo() calls bar(), and bar() calls bas(), then the stack would contain a *stack frame* for each of these three procedures; each stack frame contains the local variables used by the procedure, the parameters the procedure was called with, and the return address to jump to when the procedure completes.

  Because at any given time different threads can be in different states in their sequential computations — each can be in a different place in a different procedure called with different arguments from a different nesting of enclosing procedures — each thread needs its own stack. When a new thread is created, the operating system allocates it a new stack and stores a pointer to that stack in the thread's TCB.

- **Copy of processor registers.** A processor's registers include not only its general-purpose registers for storing intermediate values for ongoing computations, but they also include special-purpose registers, such as the instruction pointer and stack pointer.

  To be able to suspend a thread, run another thread, and later resume the original thread, the operating system needs a place to store a thread's registers when that thread is not actively running. In some systems, the general-purpose registers for a stopped thread are stored on the top of the stack, and the TCB contains only a pointer to the stack. In other systems, the TCB contains space for a copy of all processor registers.

How big a stack?

An implementation question for thread systems is: how large a stack should be allocated for each thread? A stack grows and shrinks as procedure calls are made and those calls return. The size of the stack must be large enough to accommodate the deepest nesting level needed during in the thread's lifetime. With hundreds or thousands of threads, it can be wasteful to allocate more than the minimum needed.

Most modern operating systems allocate kernel stacks in physical memory, putting space at a premium. However, the maximum procedure nesting depth in the kernel is usually small. Thus, kernels typically allocate a very small fixed sized region for each thread stack, e.g., 8 KB by default in Linux on an Intel x86. The kernel stays within this bound

due to an important kernel coding convention: buffers and data structures are always allocated on the heap and never as procedure local variables. Although most programming languages allow arbitrary data structures to be defined as procedure local or "automatic" — allocated when a procedure starts and de-allocated when the procedure exits — that can cause problems when the stack is of limited size.

User-level stacks are allocated in virtual memory and so there is less need for a tight space constraint. In a single threaded process, the stack is located at the top end of the address space, where it can grow nearly without bound. To catch program errors, most operating systems will trigger an error if the user program stack grows too large too quickly, as that is usually an indication of unbounded recursion, rather than something that was the programmer's intent.

In a multi-threaded user application, it is not possible to have each stack grow without constraint. Although some programming languages, such as Google's Go, will automatically grow the stack as needed, this is still uncommon. POSIX allows the default stack size to be library dependent (e.g., larger on a desktop machine, smaller on a smartphone). As one POSIX thread tutorial put it dryly, "Exceeding the default stack limit is often very easy to do, with the usual results: program termination and/or corrupted data." [10]. Most implementations try to detect when programs exceed the default stack limit by placing a known value at the very top and bottom of the stack to serve as a guard. The guard values can be checked on every context switch; if the value changes, it is likely the thread exceeded its stack.

To support application portability, the POSIX thread standard allows the user to redefine the default stack size to whatever is needed for the correct execution of a particular program. The thread library provided with the textbook sets the default stack size to 1 MB. This is almost certainly large enough provided you adopt the kernel approach of never putting large data objects on the stack.

Per-thread Metadata. The TCB also includes *per-thread metadata* — information for managing the thread. For example, each thread might have a thread ID, scheduling priority, and status (e.g., whether the thread is waiting for an event or is ready to be placed onto a processor).

4.4.2 Shared State

As opposed to per-thread state that is allocated for each thread, some state is *shared* between threads running in the same process or within the operating system kernel (Figure 4.8). In particular, program *code* is shared by all threads in a process, although each thread may be executing at a different place within that code. Additionally, statically allocated *global variables* and dynamically allocated *heap variables* can store information that is accessible to all threads.

Other per-thread state: Thread-local variables

In addition to the per-thread state that corresponds to execution state in the single-threaded case, some systems include additional *thread-local variables*. These variables are similar to global variables in that their scope spans different procedures, but they differ in that each thread has its own copy of these variables.

Consider these examples:

- **Errno.** In UNIX, the return value of system calls is intentionally kept simple. For example, the UNIX read system call returns either the number of bytes read (if successful) or -1 (if there was a problem). Often, an application needs additional information about the cause of the error (e.g., permission error, disk offline, etc.). To provide this, the kernel sets a variable in the application memory, the errno, with a diagnostic code for the most recent system call. As UNIX originally had only one thread per process, there was no confusion: the errno referred to the most recent system call of that process.

  In a multi-threaded program, however, multiple threads can perform system calls concurrently. Rather than redefine the entire UNIX system call interface for a multi-threaded environment, errno is now a macro that maps to a thread-local variable containing the error code for that thread's most recent system call.

- **Heap internals.** Although a program's heap is logically shared — it is acceptable for one thread to allocate an object on the heap and then pass a pointer to that object to another thread — for performance reasons heaps may internally subdivide their space into per-thread regions. The advantage of subdividing the heap is that multiple threads can each allocate objects at the same time without interfering with one another. Further, by allocating objects used by the same thread from the same memory region, cache hit rates may improve. To implement these optimizations, each subdivision of the heap has thread-local variables that track what parts of the thread-local heap are in use, what parts are free, and so on. Then, the code that allocates new memory (e.g., malloc and new) is written to use these thread-local data structures and only take memory from the shared heap if the local heap is empty.

Thread-local variables are often useful, but, for simplicity, the rest of our discussion focuses only on the TCB, registers, and stack as the core pieces of per-thread state.

WARNING: Although there is an important logical division between per-thread state and shared state, the operating system typically does not enforce this division. Nothing prevents one buggy thread from accessing another thread's (conceptually private) per-thread state. Writing to a bad pointer in one thread can corrupt the stack of another. Or a careless programmer might pass a pointer to a local variable on one thread's stack to another thread, giving the second thread a pointer to a stack location whose contents may change as the first thread calls and returns from various procedures. Or the first thread can exit after handing out a pointer to a variable on its stack; the heap will reassign that memory to an unrelated purpose. Because these bugs can depend on the specific interleavings of the threads' executions, they can be extremely hard to locate and correct.

To avoid unexpected behaviors, it is therefore important when writing multi-threaded programs to know which variables are designed to be shared across threads (global

variables, objects on the heap) and which are designed to be private (local/automatic variables).

4.5 Thread Life Cycle

It is useful to consider the progression of states as a thread goes from being created, to being scheduled and de-scheduled onto and off of a processor, and then to exiting. Figure 4.9 shows the states of a thread during its lifetime.

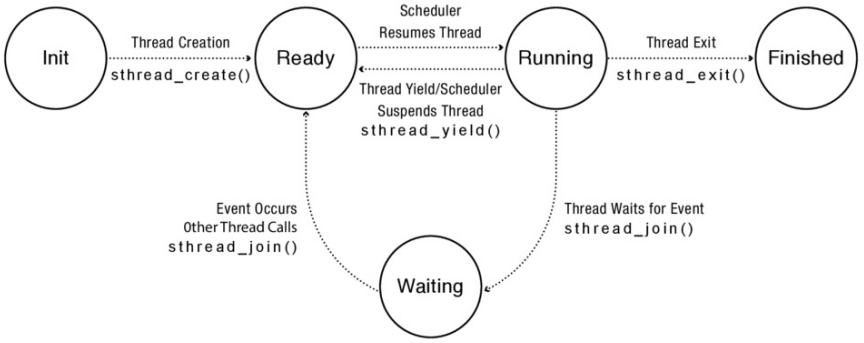

Figure 4.9: The states of a thread during its lifetime.

INIT. Thread creation puts a thread into its INIT state and allocates and initializes per-thread data structures. Once that is done, thread creation code puts the thread into the READY state by adding the thread to the *ready list*. The ready list is the set of runnable threads that are waiting their turn to use a processor. In practice, as discussed in Chapter 7, the ready list is not in fact a "list"; the operating system typically uses a more sophisticated data structure to keep track of runnable threads, such as a priority queue. Nevertheless, following convention, we will continue to refer to it as the ready list.

READY. A thread in the *READY* state is available to be run but is not currently running. Its TCB is on the ready list, and the values of its registers are stored in its TCB. At any time, the scheduler can cause a thread to transition from READY to RUNNING by copying its register values from its TCB to a processor's registers.

RUNNING. A thread in the *RUNNING* state is running on a processor. At this time, its register values are stored on the processor rather than in the TCB. A RUNNING thread can transition to the READY state in two ways:

- The scheduler can preempt a running thread and move it to the READY state by: (1) saving the thread's registers to its TCB and (2) switching the processor to run the next thread on the ready list.
- A running thread can voluntarily relinquish the processor and go from RUNNING to READY by calling yield (e.g., thread_yield in the thread library).

Notice that a thread can transition from READY to RUNNING and back many times. Since the operating system saves and restores the thread's registers exactly, only the speed of the thread's execution is affected by these transitions.

WARNING: By convention in this book, a thread that is RUNNING is not on the ready list; the ready list is for READY and not RUNNING threads. However, some operating systems, such as Linux, use a different convention, where the RUNNING thread is whichever thread is at the front of the ready list. Either convention is equivalent as long as it used consistently.

WAITING. A thread in the *WAITING* state is waiting for some event. Whereas the scheduler can move a thread in the READY state to the RUNNING state, a thread in the WAITING state cannot run until some action by another thread moves it from WAITING to READY.

The threadHello program in Figure 4.6 provides an example of a WAITING thread. After creating its children threads, the main thread must wait for them to complete, by calling thread_join once for each child. If the specific child thread is not yet done at the time of the join, the main thread goes from RUNNING to WAITING until the child thread exits.

While a thread waits for an event, it cannot make progress; therefore, it is not useful to run it. Rather than continuing to run the thread or storing the TCB on the scheduler's ready list, the TCB is stored on the *waiting list* of some *synchronization variable* associated with the event. When the required event occurs, the operating system moves the TCB from the synchronization variable's waiting list to the scheduler's ready list, transitioning the thread from WAITING to READY. We describe synchronization variables in Chapter 5.

FINISHED. A thread in the FINISHED state never runs again. The system can free some or all of its state for other uses, though it may keep some remnants of the thread in the FINISHED state for a time by putting the TCB on a *finished list*. For example, the thread_exit call lets a thread pass its exit value to its parent thread via thread_join. Eventually, when a thread's state is no longer needed (e.g., after its exit value has been read by the join call), the system can delete and reclaim the thread's state.

State of Thread	Location of Thread Control Block (TCB)	Location of Registers
INIT	Being Created	TCB
READY	Ready List	TCB
RUNNING	Running List	Processor
WAITING	Synchronization Variable's Waiting List	TCB
FINISHED	Finished List then Deleted	TCB or Deleted

Figure 4.10: Location of thread's per-thread state for different life cycle stages.

One way to understand these states is to consider where a thread's TCB and registers are stored, as shown in Figure 4.10. For example, all threads in the READY state have their TCBs on the ready list and their registers in the TCB. All threads in the RUNNING state have their TCBs on the running list and their register values in hardware registers. And all threads in the WAITING state have their TCBs on various synchronization variables' waiting lists.

The idle thread

If a system has k processors, most operating systems ensure that there are exactly k RUNNING threads, by keeping a low priority *idle thread* per processor for when there is nothing else to run.

On old machines, the idle thread would spin in a tight loop doing nothing.

Today, the idle thread still spins in a loop, but to save power, on each iteration it puts the processor into a low-power sleep mode. In sleep mode, the processor stops executing instructions until a hardware interrupt occurs. Then, the processor wakes up and handles the interrupt in the normal way — saving the state of the currently running thread (the idle thread) and running the handler. After running the handler, a thread waiting for that I/O event may now be READY. If so, the scheduler runs that thread next; otherwise, the idle thread resumes execution, putting the processor to sleep again.

Having a low-power idle thread also helps when running the operating system inside a virtual machine. Obviously, it would be inefficient for an idle operating system to consume processing cycles that could be better used by another virtual machine on the same system. Putting the processor into sleep mode is a privileged instruction, so if the operating system is running inside a virtual machine, the hardware will trap to the host kernel. The host kernel can then switch to a different virtual machine.

EXAMPLE: For the threadHello program in Figure 4.6, when thread_join returns for thread i, what is thread i's thread state?

ANSWER: When join returns, thread i has finished running and exited. The runtime system saved the exit value in the TCB and moved the TCB to the finished list (so that its exit value can be found by the parent thread). **The thread is thus in the FINISHED state.** □

EXAMPLE: For the threadHello program, what is the minimum and maximum number of times that the main thread enters the READY state on a *uniprocessor*?

ANSWER: The main thread must go into the READY state when it is first created; otherwise, it would never be scheduled. On a uniprocessor, it must also give up the processor (e.g., due to a time slice or in thread_join) in order for its children threads to run. The children threads could then completely run before the main thread is re-scheduled. Once the children have finished, the main thread can run to completion. Thus, **the minimum number of times is two.**

The maximum number of times is (near) infinite. A running thread can be preempted and re-scheduled many times, without affecting the correctness of the execution. In the limit, the thread could conceivably be preempted after each instruction! □

Where is my TCB?

A remarkably tricky implementation question is how to find the current thread's TCB. The thread library needs access to the current TCB for a number of reasons, e.g., to change its priority or to access thread-local variables.

One might think finding the TCB would be simple: just store a pointer to the TCB in a global variable. However, recall that every thread running in the same process uses exactly the same code, and therefore each thread would look in exactly the same place for the TCB. On a uniprocessor, this works: the global variable can hold the value of the current TCB, and the library can change the value whenever it switches between threads.

This does not work on a multiprocessor, however. Some systems, such as the Intel x86, have hardware support for fetching the ID of the current processor. In these systems, the thread library can maintain a global array of pointers, with the i'th entry pointing to the TCB of the thread running on the i'th processor. A running thread can then find its TCB by looking up its processor ID and then finding the corresponding entry in the array.

For systems without this feature, however, there is another approach: the stack pointer is always unique to each thread. The thread library can store a pointer to the thread TCB at the very bottom of the stack, underneath the procedure frames. (Some systems take this one step farther, and put the entire TCB at the bottom of the stack.) As long as thread stacks are aligned to start at a fixed block boundary, the low order bits of the current stack pointer can be masked to locate the pointer to the current TCB.

4.6 Implementing Kernel Threads

So far, we have described the basic data structures and operation of threads. We now describe how to implement them. The specifics of the implementation vary depending on the context:

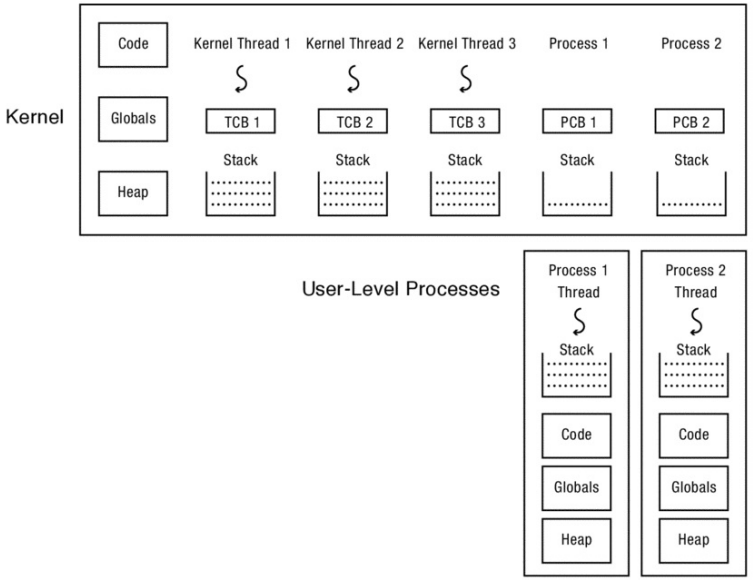

Figure 4.11: A multi-threaded kernel with three kernel threads and two single-threaded user-level processes. Each kernel thread has its own TCB and its own stack. Each user process has a stack at user-level for executing user code and a kernel interrupt stack for executing interrupts and system calls.

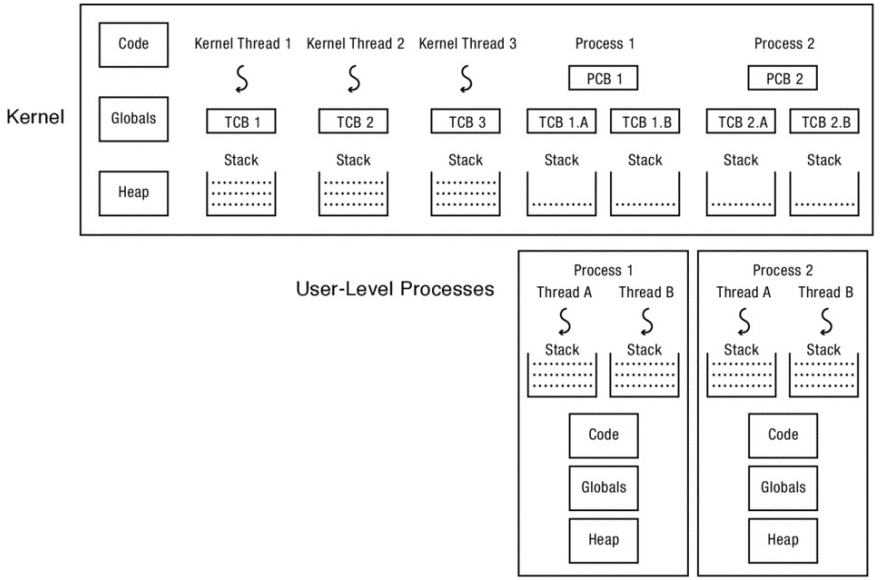

Figure 4.12: A multi-threaded kernel with three kernel threads and two user-level processes, each with two threads. Each user-level thread has a user-level stack and an interrupt stack in the kernel for executing interrupts and system calls.

- **Kernel threads.** The simplest case is implementing threads inside the operating system kernel, sharing one or more physical processors. A *kernel thread* executes kernel code and modifies kernel data structures. Almost all commercial operating systems today support kernel threads.
- **Kernel threads and single-threaded processes.** An operating system with kernel threads might also run some single-threaded user processes. As shown in Figure 4.11, these processes can invoke system calls that run concurrently with kernel threads inside the kernel.
- **Multi-threaded processes using kernel threads.** Most operating systems provide a set of library routines and system calls to allow applications to use multiple threads within a single user-level process. Figure 4.12 illustrates this case. These threads execute user code and access user-level data structures. They also make system calls into the operating system kernel. For that, they need a kernel interrupt stack just like a normal single-threaded process.
- **User-level threads.** To avoid having to make a system call for every thread operation, some systems support a model where user-level thread operations — create, yield, join, exit, and the synchronization routines described in Chapter 5 — are implemented entirely in a user-level library, without invoking the kernel.

We first describe the implementation for the baseline case of kernel threads. In Section 4.8, we explain how to extend the model to support application multi-threading implemented with kernel threads or with a user-level library.

4.6.1 Creating a Thread

```
// func is a pointer to a procedure the thread will run.
// arg is the argument to be passed to that procedure.
void
thread_create(thread_t *thread, void (*func)(int), int arg) {
    // Allocate TCB and stack
    TCB *tcb = new TCB();

    thread->tcb = tcb;
    tcb->stack_size = INITIAL_STACK_SIZE;
    tcb->stack = new Stack(INITIAL_STACK_SIZE);

    // Initialize registers so that when thread is resumed, it will start runnin
    // stub.  The stack starts at the top of the allocated region and grows down
    tcb->sp = tcb->stack + INITIAL_STACK_SIZE;
    tcb->pc = stub;

    // Create a stack frame by pushing stub's arguments and start address
    // onto the stack: func, arg
    *(tcb->sp) = arg;
    tcb->sp--;
    *(tcb->sp) = func;
    tcb->sp--;

    // Create another stack frame so that thread_switch works correctly.
    // This routine is explained later in the chapter.
    thread_dummySwitchFrame(tcb);

    tcb->state = READY;
    readyList.add(tcb);    // Put tcb on ready list
}

void
stub(void (*func)(int), int arg) {
    (*func)(arg);           // Execute the function func()
    thread_exit(0);         // If func() does not call exit,  call it here.
}
```

Figure 4.13: Pseudo-code for thread creation. The specifics of initializing the stack and the conventions for passing arguments to the initial function are machine-dependent. On the Intel x86 architecture, the stack starts at high addresses and grows down, while arguments are passed on the stack. On other systems, the stack can grow upwards and/or arguments can be passed in registers. Figure 4.14 provides pseudo-code for thread_dummySwitchFrame.

Figure 4.13 shows the pseudo-code to allocate a new thread. The goal of thread_create is to perform an asynchronous procedure call to func with arg as the argument to that procedure. When the thread runs, it will execute func(arg) concurrently with the calling thread.

There are three steps to creating a thread:

1. **Allocate per-thread state.** The first step in the thread constructor is to allocate space for the thread's per-thread state: the TCB and stack. As we have mentioned, the TCB is the data structure the thread system uses to manage the thread. The stack is an area of memory for storing data about in-progress procedures; it is allocated in memory like any other data structure.
2. **Initialize per-thread state.** To initialize the TCB, the thread constructor sets the new thread's registers to what they need to be when the thread starts RUNNING. When the thread is assigned a processor, we want it to start running func(arg). However, instead of having the thread start in func, the constructor starts the thread in a dummy function, stub, which in turn calls func.

   We need this extra step in case the func procedure returns instead of calling thread_exit. Without the stub, func would return to whatever random location is stored at the top of the stack! Instead, func returns to stub and stub calls thread_exit to finish the thread.

   To start at the beginning of stub, the thread constructor sets up the stack as if stub was just called by normal code; the specifics will depend on the calling convention of the machine. In the pseudo-code, we push stub's two arguments onto the stack: func and arg. When the thread starts running, the code in stub will access its arguments just like a normal procedure.

   In addition, we also push a dummy stack frame for thread_switch onto the stack; we defer an explanation of this detail until we discuss the implementation of thread switching.
3. **Put TCB on ready list.** The last step in creating a thread is to set its state to READY and put the new TCB on the ready list, enabling the thread to be scheduled.

4.6.2 Deleting a Thread

When a thread calls thread_exit, there are two steps to deleting the thread:

- Remove the thread from the ready list so that it will never run again.
- Free the per-thread state allocated for the thread.

Although this seems easy, there is an important subtlety: if a thread removes itself from the ready list and frees its own per-thread state, then the program may break. For example, if a thread removes itself from the ready list but an interrupt occurs before the thread finishes de-allocating its state, there is a memory leak: that thread will never resume to de-allocate its state.

Worse, suppose that a thread frees its own state? Can the thread finish running the code in

thread_exit if it does not have a stack? What happens if an interrupt occurs just after the running thread's stack has been de-allocated? If the context switch code tries to save the current thread's state, it will be writing to de-allocated memory, possibly to storage that another processor has re-allocated for some other data structure. The result could be corrupted memory, where the specific behavior depends on the precise sequence of events. Needless to say, such a bug would be very difficult to locate.

Fortunately, there is a simple fix: a thread never deletes its own state. Instead, some other thread must do it. On exit, the thread transitions to the FINISHED state, moves its TCB from the ready list to a list of *finished* threads the scheduler should never run. The thread can then safely switch to the next thread on the ready list. Once the finished thread is no longer running, it is safe for some *other* thread to free the state of the thread.

4.6.3 Thread Context Switch

To support multiple threads, we also need a mechanism to switch which threads are RUNNING and which are READY.

A *thread context switch* suspends execution of a currently running thread and resumes execution of some other thread. The switch saves the currently running thread's registers to the thread's TCB and stack, and then it restores the new thread's registers from that thread's TCB and stack into the processor.

We need to answer several questions:

- What triggers a context switch?
- How does a voluntary context switch (e.g., a call to thread_yield) work?
- How does an involuntary context switch differ from a voluntary one?
- What thread should the scheduler choose to run next?

We discuss these in turn, but we defer the last question to Chapter 7. The *mechanisms* we discuss in this Chapter work regardless of the *policy* the scheduler uses when choosing threads.

Separating mechanism from policy

Separating mechanism from policy is a useful and widely applied principle in operating system design. When mechanism and policy are cleanly separated, it is easier to introduce new policies to optimize a system for a new workload or new technology.

For example, the thread context switch abstraction cleanly separates mechanism (how to switch between threads) from policy (which thread to run) so that the mechanism works no matter what policy is used. Some systems can elect to do something simple (e.g., FIFO scheduling); other systems can optimize scheduling to meet their goals (e.g., a periodic scheduler to smoothly run real-time multimedia streams for a media device, a round-robin scheduler to balance responsiveness and throughput for a server, or a priority scheduler that devotes most resources to the visible application on a smartphone).

We will see this principle many times in this book. For example, thread synchronization mechanisms work regardless of the scheduling policy; file metadata mechanisms for locating a file's blocks work regardless of the policy for where to place the file's blocks on disk; and page translation mechanisms for mapping virtual to physical addresses work regardless of which physical pages the operating system assigns to each process.

What Triggers a Kernel Thread Context Switch? A thread context switch can be triggered by either a voluntary call into the thread library, or an involuntary interrupt or processor exception.

- **Voluntary.** The thread could call a thread library function that triggers a context switch. For example, most thread libraries provide a thread_yield call that lets the currently running thread voluntarily give up the processor to the next thread on the ready list. Similarly, the thread_join and thread_exit calls suspend execution of the current thread and start running a different one.
- **Involuntary.** An *interrupt* or *processor exception* could invoke an interrupt handler. The interrupt hardware saves the state of the running thread and executes the handler's code. The handler can decide that some other thread should run, and then switch to it. Alternatively, if the current thread should continue running, the handler restores the state of the interrupted thread and resumes execution.

  For example, many thread systems are designed to ensure that no thread can monopolize the processor. To accomplish this, they set a hardware timer to interrupt the processor periodically (e.g., every few milliseconds). The timer interrupt handler saves the state of the running thread, chooses another thread to run, and runs that thread by restoring its state to the processor.

  Other I/O hardware events (e.g., a keyboard key is pressed, a network packet arrives, or a disk operation completes) also invoke interrupt handlers. In these cases as well, the handlers save the state of the currently running thread so that it can be restored later. They then execute the handler code, and when the handler is done, they either restore the state of the current thread, or switch to a new thread. A new thread will be run if the I/O event moves a thread onto the ready list with a higher priority than the previously running thread.

Regardless, the thread system must save the current processor state, so that when the current thread resumes execution, it appears *to the thread* as if the event never occurred except for some time having elapsed. This provides the abstraction of thread execution on a virtual processor with unpredictable and variable speed.

To keep things simple, we do not want to do an involuntary context switch while we are in the middle of a voluntary one. When switching between two threads, we need to temporarily defer interrupts until the switch is complete, to avoid confusion. Processors contain privileged instructions to defer and re-enable interrupts; we make use of these in our implementation below.

Why is it necessary to turn off interrupts during thread switch?

Our implementation of thread_yield defers any interrupts that might occur during the procedure, until the yield is complete. This might seem unnecessary: after all, even if the thread context switch is interrupted, the state of the switch will be saved onto the stack. Eventually the kernel will re-schedule the thread, restore its state, and complete the thread switch.

However, a subtle inconsistency might arise. Suppose a low priority thread (e.g., the idle thread) is about to voluntarily switch to a high priority thread. It pulls the high priority thread off the ready list, and at that precise moment, an interrupt occurs. Supppose the interrupt moves a medium priority thread from WAITING to READY. Since it appears that the processor is still running the low priority thread, the interrupt handler immediately switches to the new thread. The high priority thread is in limbo! It is ready to run, but unable to do so until the low priority thread is re-scheduled. And that may not happen for a long time.

Of course, this sequence of events would not occur very often, but when it does, it would be difficult to locate or debug.

Voluntary Kernel Thread Context Switch. Because a voluntary switch is simpler to understand, we start there. Figure 4.14 shows pseudo-code for a simple implementation of thread_yield for the Intel x86 hardware architecture. A thread calls thread_yield to voluntarily relinquish the processor to another thread. The calling thread's registers are copied to its TCB and stack, and it resumes running later, when the scheduler chooses it.

```
// We enter as oldThread, but we return as newThread.
// Returns with newThread's registers and stack.
void thread_switch(oldThreadTCB, newThreadTCB) {
    pushad;                 // Push general register values onto the old stack.
    oldThreadTCB->sp = %esp; // Save the old thread's stack pointer.
    %esp = newThreadTCB->sp; // Switch to the new stack.
    popad;            // Pop register values from the new stack.
    return;
}

void thread_yield() {
    TCB *chosenTCB, *finishedTCB;

    // Prevent an interrupt from stopping us in the middle of a switch.
    disableInterrupts();

    // Choose another TCB from the ready list.
    chosenTCB = readyList.getNextThread();
    if (chosenTCB == NULL) {
        // Nothing else to run, so go back to running the original thread.
    } else {
        // Move running thread onto the ready list.
        runningThread->state = ready;
        readyList.add(runningThread);
        thread_switch(runningThread, chosenTCB); // Switch to the new thread.
        runningThread->state = running;
    }
    // Delete any threads on the finished list.
    while ((finishedTCB = finishedList->getNextThread()) != NULL) {
        delete finishedTCB->stack;
        delete finishedTCB;
    }
    enableInterrupts();
}

// thread_create must put a dummy frame at the top of its stack:
// the return PC and space for pushad to have stored a copy of the registers.
// This way, when someone switches to a newly created thread,
// the last two lines of thread_switch work correctly.
void thread_dummySwitchFrame(newThread) {
    *(tcb->sp) = stub;      // Return to the beginning of stub.
    tcb->sp--;
    tcb->sp -= SizeOfPopad;
}
```

Figure 4.14: Pseudo-code for thread_switch and thread_yield on the Intel x86 architecture. Note that thread_yield is a no-op if there are no other threads to run. Otherwise, it saves the old thread state and restores the new thread state. When the old thread is re-scheduled, it returns from thread_switch as the running thread.

The pseudo-code for thread_yield first turns off interrupts to prevent the thread system from attempting to make two context switches at the same time. The pseudo-code then pulls the next thread to run off the ready list (if any), and switches to it.

The thread_switch code may seem tricky, since it is called in the context of the old thread and finishes in the context of the new thread. To make this work, thread_switch saves the state of the registers to the stack and saves the stack pointer to the TCB. It then switches to the stack of the new thread, restores the new thread's state from the new thread's stack, and returns to whatever program counter is stored on the new stack.

A twist is that the return location may not be to thread_yield! The return is to whatever the new thread was doing beforehand. For example, the new thread might have been WAITING in thread_join and is now READY to run. The thread might have called thread_yield. Or it might be a newly created thread just starting to run.

It is essential that any routine that causes the thread to yield or block call thread_switch in the same way. Equally, to create a new thread, thread_create must set up the stack of the new thread to be as if it had suspended execution just before performing its first instruction. Then, if the newly created thread is the next thread to run, a thread can call thread_yield, switch to the newly created thread, switch to its stack pointer, pop the register values off the stack, and "return" to the new thread, even though it had never called switch in the first place.

EXAMPLE: Suppose two threads each loop, calling thread_yield on each iteration.

```
go() {
    while(1) {
```

```
            thread_yield();
        }
    }
```

What is the sequence of steps as seen by the physical processor and by each thread?

ANSWER: From the processor's point of view, one instruction follows the next, but now the instructions from different threads are interleaved (as they must be if they are multiplexed).

Figure 4.15 shows the interleaving: thread_yield is called by one thread but returns in a different thread. thread_yield deliberately violates the procedure call conventions compilers normally follow by manipulating the stack and program counter to switch between threads.

However, the threads themselves can ignore this complexity. From their point of view, they each run this loop on their own (variable-speed) virtual processor. □

Logical View

Thread 1	**Thread 2**
go(){	go(){
while(1){	while(1){
thread_yield();	thread_yield();
}	}
}	}

Physical Reality

Thread 1's instructions	**Thread 2's instructions**	**Processor's instructions**
"return" from thread_switch into stub		"return" from thread_switch into stub
call go		call go
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch
save thread 1 state to TCB		save thread 1 state to TCB
load thread 2 state		load thread 2 state
	"return" from thread_switch into stub	"return" from thread_switch into stub
	call go	call go
	call thread_yield	call thread_yield
	choose another thread	choose another thread
	call thread_switch	call thread_switch
	save thread 2 state to TCB	save thread 2 state to TCB
	load thread 1 state	load thread 1 state
return from thread_switch		return from thread_switch
return from thread_yield		return from thread_yield
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch
save thread 1 state to TCB		save thread 1 state to TCB
load thread 2 state		load thread 2 state
	return from thread_switch	return from thread_switch
	return from thread_yield	return from thread_yield
	call thread_yield	call thread_yield
	choose another thread	choose another thread
	call thread_switch	call thread_switch

	save thread 2 state to TCB	save thread 2 state to TCB
	load thread 1 state	load thread 1 state
return from thread_switch		return from thread_switch
return from thread_yield		return from thread_yield
…	…	…

Figure 4.15: Interleaving of instructions when two threads loop and call thread_yield().

A zero-thread kernel

Not only can we have a single-threaded kernel or a multi-threaded kernel, it is actually possible to have a kernel with no threads of its own — a zero-threaded kernel! In fact, this used to be quite common [107].

Consider the simple picture of the operating system described in Chapter 2. Once the system has booted, initialized its device drivers, and started some user-level processes like a login shell, everything else the kernel does is event-driven, i.e., done in response to an interrupt, processor exception, or system call.

In a simple operating system like this, there is no need for a "kernel thread" or "kernel thread control block" to keep track of an ongoing computation. Instead, when an interrupt, trap, or exception occurs, the stack pointer gets set to the base of the interrupt stack, and the instruction pointer gets set to the address of the handler. Then, the handler executes and either returns immediately to the interrupted user-level process or suspends the user-level process and "returns" to some other user-level process. In either case, the next event (interrupt, processor exception, or system call) starts this process anew.

Involuntary Kernel Thread Context Switch. Chapter 2 explained what happens when an interrupt, exception, or trap interrupts a running user-level process: hardware and software work together to save the state of the interrupted process, run the kernel's handler, and restore the state of the interrupted process.

The mechanism is almost identical when an interrupt or trap triggers a thread switch between threads in the kernel. The three steps described in Chapter 2 are slightly modified (*changes are written in italics*):

1. **Save the state.** Save the currently running *thread's* registers so that the handler can run code without disrupting the interrupted *thread*.

   Hardware saves some state when the interrupt or exception occurs, and software saves the rest of the state when the handler runs.

2. **Run the kernel's handler.** Run the kernel's handler code to handle the interrupt or exception. *Since we are already in kernel mode, we do not need to change from user to kernel mode in this step. We also do not need to change the stack pointer to the base of the kernel's interrupt stack. Instead, we can just push saved state or handler variables onto the current stack, starting from the current stack pointer.*
3. **Restore the state.** Restore the *next ready thread's* registers so that the thread can resume running where it left off.

In short, comparing a switch between kernel threads to what happens on a user-mode transfer: (1) there is no need to switch modes (and therefore no need to switch stacks) and (2) the handler can resume any thread on the ready list rather than always resuming the thread or process that was just suspended.

Implementation Details. On most processor architectures, a simple (but inefficient) way to swap to the next thread from within an interrupt handler is to call thread_switch just before the handler returns. As we have already seen, thread_switch saves the state of the current thread (that is, the state of the interrupt handler) and switches to the new kernel thread. When the original thread resumes, it will return from thread_switch, and immediately pop the interrupt context off the stack, resuming execution at the point where it was interrupted.

Most systems, such as Linux, make a small optimization to improve interrupt handling performance. The state of the interrupted thread is already saved on the stack, albeit in the format specified by the interrupt hardware. If we modify thread_switch to save and restore registers exactly as the interrupt hardware does, then returning from an interrupt and resuming a thread are the same action: they both pop the interrupt frame off the stack to resume the next thread to run.

For example, to be compatible with x86 interrupt hardware, the software implementation of thread_switch would simulate the hardware case, saving the return instruction pointer and eflags register before calling pushad to save the general-purpose registers. After switching to the new stack, it would call iret to resume the new thread, whether the new thread was suspended by a hardware event or a software call.

4.7 Combining Kernel Threads and Single-Threaded User Processes

Previously, Figure 4.11 illustrated a system with both kernel threads and single-threaded user processes. A process is a sequential execution of instructions, so each user-level process includes the process's thread. However, a process is more than just a thread because it has its own address space. Process 1 has its own view of memory, its own code, its own heap, and its own global variables that differ from those of process 2 (and from the kernel's).

Because a process contains more than just a thread, each process's process control block (PCB) needs more information than a thread control block (TCB) for a kernel thread. Like a TCB, a PCB for a single-threaded process must store the processor registers when the process's thread is not running. In addition, the PCB has information about the process's address space; when a context switch occurs from one process to another, the operating

system must change the virtual memory mappings as well as the register state.

Since the PCB and TCB each represent one thread, the kernel's ready list can contain a mix of PCBs for processes and TCBs for kernel threads. When the scheduler chooses the next thread to run, it can pick either kind. A thread switch is nearly identical whether switching between kernel threads or switching between a process's thread and a kernel thread. In both cases, the switch saves the state of the currently running thread and restores the state of the next thread to run.

As we mentioned in Chapter 2, most operating systems dedicate a kernel interrupt stack for each process. This way, when the process needs to perform a system call, or on an interrupt or processor exception, the hardware traps to the kernel, saves the user-level processor state, and starts running at a specific handler in the kernel. Once inside the kernel, the process thread behaves exactly like a kernel thread — it can create threads (or other processes), block (e.g., in UNIX process wait or on I/O), and even exit. While inside the kernel, the process can be pre-empted by a timer interrupt or I/O event, and a higher priority process or kernel thread can run in its place. The PCB and kernel stack for the preempted process stores both its current kernel state, as well as the user-level state saved when the process initiated the system call.

We can resume a process in the kernel using thread_switch. However, when we resume execution of the user-level process after the completion of a system call or interrupt, we must restore its state precisely as it was beforehand: with the correct value in its registers, executing in user mode, with the appropriate virtual memory mappings, and so forth.

An important detail is that many processor architectures have extra co-processor state, e.g., floating point registers, for user-level code. Typically, the operating system kernel does not make use of floating point operations. Therefore, the kernel does not need to save those registers when switching between kernel threads, but it does save and restore them when switching between processes.

One small difference

You may notice that a mode switch in Chapter 2 caused the x86 hardware to save not just the instruction pointer and eflags register but also the *stack pointer* of the interrupted process before starting the handler. For mode switching, the hardware changes the stack pointer to the kernel's interrupt stack, so it must save the original user-level stack pointer.

In contrast, when switching from a kernel thread to a kernel handler, the hardware does not switch stacks. Instead, the handler runs on the current stack, not on a separate interrupt stack. Therefore, the hardware does not need to save the original stack pointer; the handler just saves the stack pointer with the other registers as part of the pushad instruction.

Thus, x86 hardware works slightly differently when switching between a kernel thread and a kernel handler than when doing a mode switch:

- **Entering the handler.** When an interrupt or exception occurs, if the processor detects that it is already in kernel mode (by inspecting the eflags register), it just pushes the instruction pointer and eflags registers (but not the stack pointer) onto the existing stack. On the other hand, if the hardware detects that it is switching from user-mode to kernel-mode, then the processor also changes the stack pointer to the base of the interrupt stack and pushes the original stack pointer along with the instruction pointer and eflags registers onto the new stack.
- **Returning from the handler.** When the iret instruction is called, it inspects both the current eflags register and the value on the stack that it will use to restore the earlier eflags register. If the mode bit is identical, then iret just pops the instruction pointer and eflags register and continues to use the current stack. On the other hand, if the mode bit differs, then the iret instruction pops not only the instruction pointer and eflags register, but also the saved stack pointer, thus switching the processor's stack pointer to the saved one.

4.8 Implementing Multi-Threaded Processes

So far, we have described how to implement multiple threads that run inside the operating system kernel. Of course, we also want to be able to run user programs as well. Since many user programs are single-threaded, we start with the simple case of how to integrate kernel threads and single-threaded processes. We then turn to various ways of implementing *multi-threaded processes*, processes with multiple threads. All widely used modern operating systems support both kernel threads and multi-threaded processes. Both programming languages, such as Java, and standard library interfaces such as POSIX and simple threads, use this operating system support to provide the thread abstraction to the programmer.

4.8.1 Implementing Multi-Threaded Processes Using Kernel Threads

The simplest way to support multiple threads per process is to use the kernel thread implementation we have already described. When a kernel thread creates, deletes, suspends, or resumes a thread, it can use a simple procedure call. When a user-level thread accesses the thread library to do the same things, it uses a system call to ask the kernel to do the operation on its behalf.

As shown earlier in Figure 4.12, a thread in a process has:

- A user-level stack for executing user code.
- A kernel interrupt stack for when this thread makes system calls, causes a processor exception, or is interrupted.
- A kernel TCB for saving and restoring the per-thread state.

To create a thread, the user library allocates a user-level stack for the thread and then does a system call into the kernel. The kernel allocates a TCB and interrupt stack, and arranges the state of the thread to start execution on the user-level stack at the beginning of the requested procedure. The kernel needs to store a pointer to the TCB in the process control block; if the process exits, the kernel must terminate any other threads running in the

process.

After creating the thread, the kernel puts the new thread on the ready list, to be scheduled like any other thread, and returns unique identifier for the user program to use when referring to the newly created thread (e.g., for join).

Thread join, yield, and exit work the same way: by calling into the kernel to perform the requested function.

4.8.2 Implementing User-Level Threads Without Kernel Support

It is also possible to implement threads as a library completely at user level, without any operating system support. Early thread libraries took this pure user-level approach for the simple reason that few operating systems supported multi-threaded processes. Even once operating system support for threads became widespread, pure user-level threads were sometimes used to minimize dependencies on specific operating systems and to maximize portability; for example, the earliest implementations of Sun's Java Virtual Machine (JVM) implemented what were called *green threads*, a pure user-level implementation of threads.

The basic idea is simple. The thread library instantiates all of its data structures within the process: TCBs, the ready list, the finished list, and the waiting lists all are just data structures in the process's address space. Then, calls to the thread library are just procedure calls, akin to how the same functions are implemented within a multi-threaded kernel.

To the operating system kernel, a multi-threaded application using green threads appears to be a normal, single-threaded process. The process as a whole can make system calls, be time-sliced, etc. Unlike with kernel threads, when a process using green threads is time-sliced, the entire process, including all of its threads, is suspended.

A limitation of green threads is that the operating system kernel is unaware of the state of the user-level ready list. If the application performs a system call that blocks waiting for I/O, the kernel is unable to run a different user-level thread. Likewise, on a multiprocessor, the kernel is unable to run the different threads running within a single process on different processors.

Preemptive User-level Threads. However, it is possible on most operating systems to implement preemption among user-level threads executing within a process. As we discussed in Chapter 2, most operating systems provide an upcall mechanism to deliver asynchronous event notification to a process; on UNIX these are called signal handlers. Typical events or signals include the user hitting "Escape" or on UNIX "Control-C"; this informs the application to attempt to cleanly exit. Another common event is a timer interrupt to signal elapsed real time. To deliver an event, the kernel suspends the process execution and then resumes it running at a handler specified by the user code, typically on a separate upcall or signal stack.

To implement preemptive multi-threading for some process P :

1. The user-level thread library makes a system call to register a timer signal handler and signal stack with the kernel.
2. When a hardware timer interrupt occurs, the hardware saves P 's register state and runs the kernel's handler.
3. Instead of restoring P 's register state and resuming P where it was interrupted, the kernel's handler copies P 's saved registers onto P 's signal stack.
4. The kernel resumes execution in P at the registered signal handler on the signal stack.
5. The signal handler copies the processor state of the preempted user-level thread from the signal stack to that thread's TCB.
6. The signal handler chooses the next thread to run, re-enables the signal handler (the equivalent of re-enabling interrupts), and restores the new thread's state from its TCB into the processor. execution with the state (newly) stored on the signal stack.

This approach virtualizes interrupts and processor exceptions, providing a user-level process with a very similar picture to the one the kernel gets when these events occur.

4.8.3 Implementing User-Level Threads With Kernel Support

Today, most programs use kernel-supported threads rather than pure user-level threads. Major operating systems support threads using standard abstractions, so the issue of portability is less of an issue than it once was.

However, various systems take more of a hybrid model, attempting to combine the lightweight performance and application control over scheduling found in user-level threads, while keeping many of the advantages of kernel threads.

Hybrid Thread Join. Thread libraries can avoid transitioning to the kernel in certain cases. For example, rather than always making a system call for thread_join to wait for the target thread to finish, thread_exit can store its exit value in a data structure in the process's address space. Then, if the call to thread_join happens after the targeted thread has exited, it can immediately return the value without having to make a system call. However, if the call to thread_join precedes the call to thread_exit, then a system call is needed to transition to the WAITING state and let some other thread run. As a further optimization, on a multiprocessor it can sometimes make sense for thread_join to spin for a few microseconds before entering the kernel, in the hope that the other thread will finish in the meantime.

Per-Processor Kernel Threads. It is possible to adapt the green threads approach to work on a multiprocessor. For many parallel scientific applications, the cost of creating and synchronizing threads is paramount, and so an approach that requires a kernel call for most thread operations would be prohibitive. Instead, the library multiplexes user-level threads on top of kernel threads, in exactly the same way that the kernel multiplexes kernel threads on top of physical processors.

When the application starts up, the user-level thread library creates one kernel thread for each processor on the host machine. As long as there is no other activity on the system, the kernel will assign each of these threads a processor. Each kernel thread executes the user-

level scheduler in parallel: pull the next thread off the user-level ready list, and run it. Because thread scheduling decisions occur at user level, they can be flexible and application-specific; for example, in a parallel graph algorithm, the programmer might adjust the priority of various threads based on the results of the computation on other parts of the graph.

Of course, most of the downsides of green threads are still present in these systems:

- Any time a user-level thread calls into the kernel, its host kernel thread blocks. This prevents the thread library from running a different user-level thread on that processor in the meantime.
- Any time the kernel time-slices a kernel thread, the user-level thread it was running is also suspended. The library cannot resume that thread until the kernel thread resumes.

Scheduler Activations. To address these issues, some operating systems have added explicit support for user-level threads. One such model, implemented most recently in Windows, is called *scheduler activations*. In this approach, the user-level thread scheduler is notified (or activated) for every kernel event that might affect the user-level thread system. For example, if one thread blocks in a system call, the activation informs the user-level scheduler that it should choose another thread to run on that processor. Scheduler activations are like upcalls or signals, except that they do not return to the kernel; instead, they directly perform user-level thread suspend and resume.

Various operations trigger a scheduler activation upcall:

1. **Increasing the number of virtual processors.** When a program starts, it receives an activation to inform the program that it has been assigned a virtual processor: that activation runs the main thread and any other threads that might be created. To assign another virtual processor to the program, the kernel makes another activation upcall on the new processor; the user-level scheduler can pull a waiting thread off the ready list and run it.
2. **Decreasing the number of virtual processors.** When the kernel preempts a virtual processor (e.g., to give the processor to a different process), the kernel makes an upcall on one of the other processors assigned to the parallel program. The thread system can then move the preempted user-level thread onto the ready list, so that a different processor can run it.
3. **Transition to WAITING.** When a user-level thread blocks in the kernel waiting for I/O, the kernel similarly makes an upcall to notify the user-level scheduler that it needs to take action, e.g., to choose another thread to run while waiting for the I/O to complete.
4. **Transition from WAITING to READY.** When the I/O completes, the kernel makes an upcall to notify the scheduler that the suspended thread can be resumed.
5. **Transition from RUNNING to idle.** When a user-level activation finds an empty ready list (i.e., it has no more work to do), it can make a system call into the kernel to return the virtual processor for use by some other process.

As a result, most thread management functions — thread_create, thread_yield, thread_exit, and thread_join, as well as the synchronization functions described in Chapter 5 — are implemented as procedure calls within the process. Yet the user-level thread system always knows exactly how many virtual processors it has been assigned and is in complete control of what runs on those processors.

4.9 Alternative Abstractions

Although threads are a common way to express and manage concurrency, they are not the only way. In this section, we describe two popular alternatives, each targeted at a different application domain:

- **Asynchronous I/O and event-driven programming.** Asynchronous I/O and events allow a single-threaded program to cope with high-latency I/O devices by overlapping I/O with processing and other I/O.
- **Data parallel programming.** With data parallel programming, all processors perform the same instructions in parallel on different parts of a data set.

In each case, the goal is similar: to replace the complexities of multi-threading with a deterministic, sequential model that is easier for the programmer to understand and debug.

4.9.1 Asynchronous I/O and Event-Driven Programming

Asynchronous I/O is a way to allow a single-threaded process to issue multiple concurrent I/O requests at the same time. The process makes a system call to issue an I/O request but the call returns immediately, without waiting for the result. At a later time, the operating system provides the result to the process by either: (1) calling a signal handler, (2) placing the result in a queue in the process's memory, or (3) storing the result in kernel memory until the process makes another system call to retrieve it.

An example use of asynchronous I/O is to overlap reading from disk with other computation in the same process. Reading from disk can take tens of milliseconds. In Linux, rather than issuing a read system call that blocks until the requested data has been read from disk, a process can issue an aio_read (asynchronous I/O read) system call; this call tells the operating system to initiate the read from disk and then to immediately return. Later, the process can call aio_error to determine if the disk read has finished and aio_return to retrieve the read's results, as shown in Figure 4.16.

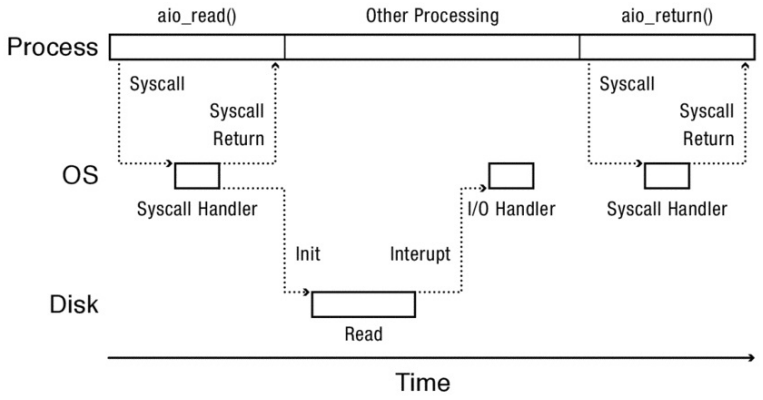

Figure 4.16: An asynchronous file read on Linux. The application calls aio_read to start the read; this system call returns immediately after the disk read is initialized. The application may then do other processing while the disk is completing the requested operation. The disk interrupts the processor when the operation is complete; this causes the kernel disk interrupt handler to run. The application at any time may ask the kernel if the results of the disk read are available, and then retrieve them with aio_return.

One common design pattern lets a single thread interleave different I/O-bound tasks by waiting for different I/O events. Consider a web server with 10 active clients. Rather than creating one thread per client and having each thread do a blocking read on the network connection, an alternative is for the server to have one thread that processes, in turn, the next message to arrive from *any* client.

For this, the server does a select call that blocks until *any* of the 10 network connections has data available to read. When the select call returns, it provides a list of connection with available data. The thread can then read from those connections, knowing that the read will always return immediately. After processing the data, the thread then calls select again to wait for the next data to arrive. Figure 4.17 illustrates this design pattern.

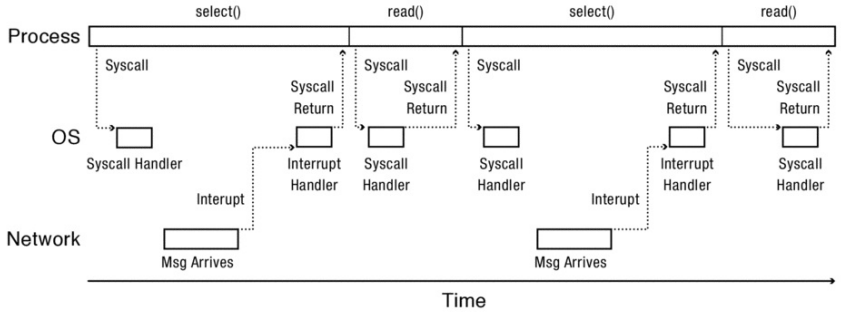

Figure 4.17: A server managing multiple concurrent connections using select. The server calls select to wait for data to arrive on any connection. The server then reads all available data, before returning to select.

Asynchronous I/O allows progress by many concurrent operating system requests. This approach gives rise to an *event-driven programming* pattern where a thread spins in a loop; each iteration gets and processes the next I/O event. To process each event, the thread typically maintains for each task a *continuation*, a data structure that keeps track of a task's current state and next step.

For example, handling a web request can involve a series of I/O steps: (a) make a network connection, (b) read a request from the network connection, (c) read the requested data from disk, and (d) write the requested data to the network connection. If a single thread is handling requests from multiple different clients at once, it must keep track of where it is in that sequence for each client.

Further, the network may divide a client's request into several packets so that the server needs to make several read calls to assemble the full packet. The server may be doing this request assembly for multiple clients at once. Therefore, it needs to keep several per-client variables (e.g., a request buffer, the number of bytes expected, and the number of bytes received so far). When a new message arrives, the thread uses the network connection's port number to identify which client sent the request and retrieves the appropriate client's variables using this port number/client ID. It can then process the data.

Event-Driven Programming vs. Threads

Although superficially different, overlapping I/O is fundamentally the same whether using asynchronous I/O and event-driven programming or synchronous I/O and threads. In either case, the program blocks until the next task can proceed, restores the state of that task, executes the next step of that task, and saves the task's state until it can take its next step. The differences are: (1) whether the state is stored in a continuation or TCB and (2) whether the state save/restore is done explicitly by the application or automatically by the thread system.

Consider a simple server that collects incoming data from several clients into a set of per-client buffers. The pseudo-code for the event-driven and thread-per-client cases is similar:

```
// Event-driven
Hashtable<Buffer*> *hash;

while(1) {
    connection = use select() to find a
                 readable connection ID
    buffer = hash.remove(connection);
    got = read(connection, tmpBuf,
                           TMP_SIZE);
    buffer->append(tmpBuf, got);
    buffer = hash.put(connection,
                      buffer);
}


// Thread-per-client
```

```
Buffer *b;

while(1) {
    got = read(connection, tmpBuf,
                           TMP_SIZE);
    buffer->append(tmpBuf, got);
}
```

When these programs execute, the system performs nearly the same work, as shown in Figure 4.18. With events, the code uses select to determine which connection's packet to retrieve next. With threads, the kernel transparently schedules each thread when data has arrived for it.

The state in both cases is also similar. In the event-driven case, the application maintains a hash table containing the buffer state for each client. The server must do a lookup to find the buffer each time a packet arrives for a particular client. In the thread-per-client case, each thread has just one buffer, and the operating system keeps track of the different threads' states.

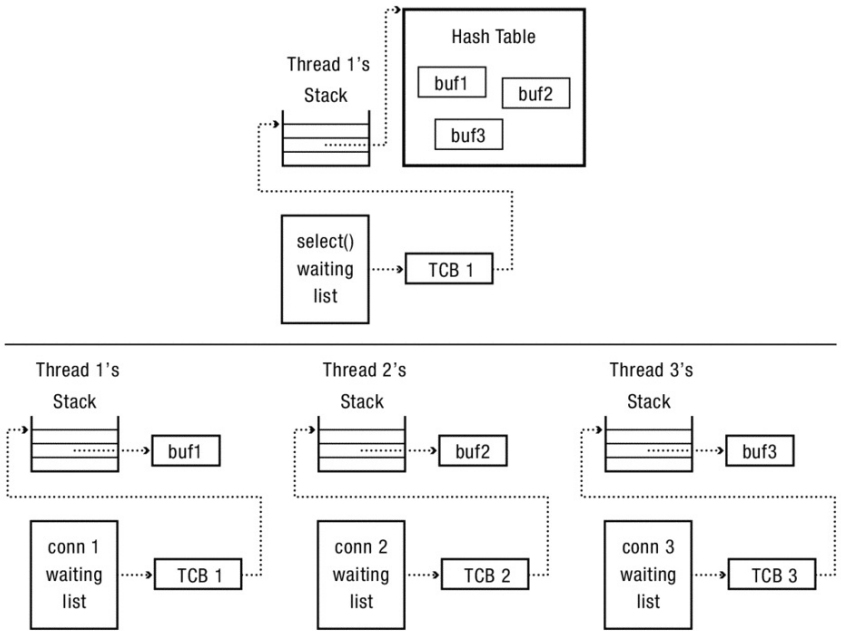

Figure 4.18: Two alternate implementations of a server. In the upper picture, a single thread uses a hash table to keep track of connection state. In the lower picture, each thread keeps a pointer to the state for one connection.

To compare the two approaches, consider again the various use cases for threads from Section 4.1:

- **Performance: Coping with high-latency I/O devices.** Either approach — event-driven or threads — can overlap I/O and processing. Which provides better performance?

  The common wisdom has been that the event-driven approach was significantly faster for two reasons. First, the space and context switch overheads of this approach could be lower because a thread system must use generic code that allocates a stack for each thread's state and that saves and restores all registers on each context switch, while the event-driven approach lets programmers allocate and save/restore just the state needed for each task. Second, some past operating systems had inefficient or unscalable implementations of their thread systems, making it important not to create too many threads for each process.

  Today, the comparison is less clear cut. Many systems now have large memories, so the cost of allocating a thread stack for each task is less critical. For example, allocating 1000 threads with an 8 KB stack per thread on a machine with 1 GB of memory would consume less than 1% of the machine's memory. Also, most operating systems now have efficient and scalable threads libraries. For example, while the Linux 2.4 kernel had poor performance when processes had many threads, Linux 2.6 revamped the thread system, improving its scalability and absolute performance.

  Anecdotal evidence suggests that the performance gap between the two approaches has greatly narrowed. For some applications, highly optimized thread management code and synchronous I/O paths can out-perform less-optimized application code and asynchronous I/O paths. In most cases, the performance difference is small enough that other factors (e.g., code simplicity and ease of maintenance) are more important than raw performance. If performance is crucial for a particular application, then, as is often the case, there is no substitute for careful benchmarking before making your decision.

- **Performance: Exploiting multiple processors.** By itself, the event-driven approach does not help a program exploit multiple processors. In practice, event-driven and thread approaches are often combined: a program that uses n processors can have n threads, each of which uses the event-driven pattern to multiplex multiple I/O-bound tasks on each processor.

- **Responsiveness: Shifting work to run in the background.** While event-driven programming can be effective when tasks are usually short-lived, threads can be more convenient when there is a mixture of foreground and background tasks. At some cost in coding complexity, the event-driven model can be adapted to this case, e.g., by cutting long tasks into smaller chunks whose state can be explicitly saved when higher priority work is pending.

- **Program structure: Expressing logically concurrent tasks.** Whenever there are two viable programming styles, there are strong advocates for each approach. The situation is no different here, with some advocates of event-driven programming

arguing that the synchronization required when threads share data makes threads more complex than events. Advocates for threads argue that they provide a more natural way to express the control flow of a program than having to explicitly store a computation's state in a continuation.

In our opinion, there remain cases where both styles are appropriate, and we use both styles in our own programs. That said, for most I/O-intensive programs, threads are preferable: they are often more natural, reasonably efficient, and simpler when running on multiple processors.

4.9.2 Data Parallel Programming

Another important application area is parallel computing, and there is an ongoing debate as to the effectiveness of threads versus other models for expressing and managing parallelism.

One popular model is *data parallel programming*, also known as SIMD (single instruction multiple data) programming or *bulk synchronous parallel programming*. In this model, the programmer describes a computation to apply in parallel across an entire data set at the same time, operating on independent data elements. The work on every data item must complete before moving onto the next step; one processor can use the results of a different processor only in some later step. As a result, the behavior of the program is deterministic. Rather than having programmers divide work among threads, the runtime system decides how to map the parallel work across the hardware's processors.

For example, taking the earlier example of zeroing a buffer in parallel in Figure 4.7, a data parallel program to zero an N item array can be as simple as:

```
forall i in 0:N-1
    array[i] = 0;
```

The runtime system would divide the array among processors to execute the computation in parallel. Of course, the runtime system itself might be implemented using threads, but this is invisible to the programmer.

Large data-analysis tasks often use data parallel programming. For example, Hadoop is an open source system that can process and analyze terabytes of data spread across hundreds or thousands of servers. It applies an arbitrary computation to each data element, such as to update the popularity of a web page based on a previous estimate of the popularity of the pages that refer to it. Hadoop applies the computation in parallel across all web pages, repeatedly, until the popularity of every page has converged. A search engine can then use the results to decide which pages should be returned in response to a search query.

Another example is SQL (Structured Query Language). SQL is a standard language for accessing databases in which programmers specify the database query to perform, and the database maps the query to lower-level thread and disk operations.

Multimedia streams (e.g., audio, video, and graphics) often have large amounts of data on which similar operations are repeatedly performed, so data parallel programming is frequently used for media processing; specialized hardware to support this type of parallel processing is common. Because they are optimized for highly structured data parallel programs, GPUs (Graphical Processing Units) can provide significantly higher rates of data processing. For example, in 2013 a mid-range Radeon 7850 GPU was capable of 1.69 TFLOPS (Trillion FLoating point Operations Per Second (double-precision)); for comparison, an Intel i7 3960 CPU (a high-end, six core general-purpose processor) was capable of 0.19 double-precision TFLOPS.

Considerable effort is currently going towards developing and using General Purpose GPUs (GPGPUs) — GPUs that better support a wider-range of programs. It is still not clear which classes of programs can work well with GPGPUs and which require more traditional CPU architectures, but for those programs that can be ported to the more restrictive GPGPU programming model, performance gains could be dramatic.

4.10 Summary and Future Directions

Concurrency is ubiquitous — not only do most smartphones, servers, desktops, laptops, and tablets have multiple cores, but users have come to expect a responsive interface at all times, I/O latencies have become gigantic compared to computer instruction cycle times, and servers must be able to process large numbers of simultaneous requests.

Although threads are not the only possible solution to these issues, they are a general-purpose technique that can be applied to a wide range of concurrency issues. In our view, multi-threaded programming is a skill that every professional programmer must master.

In this chapter, we have discussed:

- **The thread abstraction.** Threads are a set of concurrent activities, each of which executes sequentially at unpredictable speed.
- **A simple thread API.** Thread libraries, whether for use in the operating system kernel or in application code, provide the ability to perform an asynchronous procedure call.
- **Thread implementations.** The core of any implementation of preemptive multi-threading is the ability to save one thread's state and restore another's. The thread system keeps track of the saved state of all threads not currently running; it switches threads between READY and RUNNING as needed. The implementation of multi-threading can be in the kernel or at user-level, depending on the goals of the system. In our view, most systems in the future will have both a kernel-level thread system for managing concurrency in the operating system, and a lightweight thread system for expressing parallelism at the application level.
- **Alternative abstractions.** Practical alternatives to threads exist for two important domains: event-driven programming for servers as well as data parallel programming for multiprocessors.

Technology trends suggest that concurrent programming will only increase in importance

over time. After several decades in which computer architects were able to make individual processor cores run faster and faster, we have reached a point where the performance of individual cores is leveling off and where further speedups will have to come from parallel processing.

The best programming model for expressing and managing parallelism is still an active area of research, but it seems likely that threads will remain an important option for decades to come.

4.10.1 Historical Notes

The extreme engineering complexity and bugginess of commercial operating systems in the 1960's led researchers to investigate alternatives. One direct result of this experience was modern software engineering: the systematic management of complex implementation tasks through the careful control of feature lists, module testing, assertions, and so forth.

Another consequence was the use of threads for managing concurrency. One of the most influential papers in computer science history is Dijkstra's description of his THE system [48]. Dijkstra argued for constructing operating systems as a series of layered abstractions, with communicating threads implementing each layer. Within a decade, the research community was convinced. When Xerox PARC built the Alto in the late 1970's, the Alto's operating system was built from the ground up using threads. The Alto demonstrated most of the technology we now take for granted with personal computers: bit-mapped display, menus, windowing, mice, Ethernet, and email. We base much of our description of thread programming on the experiences from that project [98].

Widespread commercial adoption of threads took much longer, however. By the early 1990's, the widespread adoption of client-server computing led to several commercially important operating systems written from scratch using threads, including Microsoft's Windows NT, SUN Microsystems Solaris, and Linux. Client operating systems followed, and by the late 1990's, with Apple's introduction of OS X, all major commercial operating systems were based on threads. At about the same time, the interface to thread libraries became standardized, starting with POSIX in 1995. Likewise, modern programming languages such as Java were designed with constructs for creating and synchronizing threads.

The increasing importance of parallel processing led to the development of very lightweight user-level thread implementations, as there is little point to parallelizing an application unless it improves performance. By the early 90's, scheduler activations were developed to integrate user-level and kernel threads [2].

Even so, the topic of whether threads are a better programming model than the alternatives remains an active one [159]. Several prominent operating systems researchers have argued that normal programmers should almost never use threads because (a) it is just too hard to write multi-threaded programs that are correct and (b) most things that threads are commonly used for can be accomplished in other, safer ways [129, 160]. These are important arguments to understand — even if you disagree with them, they point out pitfalls with using threads that are important to avoid.

Exercises

For some of the following problems, you will need to download the thread library from http://ospp.cs.washington.edu/instructor.html. The comment at the top of threadHello.c explains how to compile and run a program that uses this library.

1. Download threadHello.c, compile it, and run it several times. What happens when you run it? Do you get the same result if you run it multiple times? What if you are also running some other demanding processes (e.g., compiling a big program, playing a Flash game on a website, or watching streaming video) when you run this program?
2. For the threadHello program in Figure 4.6, suppose that we delete the second for loop so that the main routine simply creates NTHREADS threads and then prints "Main thread done." What are the possible outputs of the program now. **Hint:** Fewer than NTHREADS+1 lines may be printed in some runs. Why?
3. How expensive are threads? Write a program that times how long it takes to create and then join 1000 threads, where each thread simply calls thread_exit(0) as soon as it starts running.
4. Write a program that has two threads. Make the first thread a simple loop that continuously increments a counter and prints a period (".") whenever the value of that counter is divisible by 10,000,000. Make the second thread repeatedly wait for the user to input a line of text and then print "Thank you for your input." On your system, does the first thread makes rapid progress? Does the second thread respond quickly?

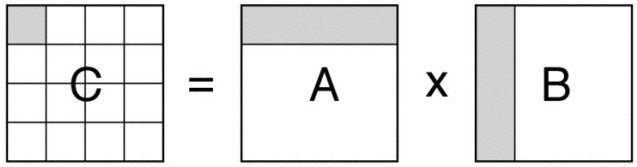

Figure 4.19: Matrix multiplication.

5. Write a program that uses threads to perform a parallel matrix multiply. To multiply two matrices, C = A * B, the result entry $C_{(i,j)}$ is computed by taking the dot product of the ith row of A and the jth column of B: $C_{i,j} = \Sigma_{k=0}^{N-1} A_{(i,k)} B_{(k,j)}$. We can divide the work by creating one thread to compute each value (or each row) in C, and then executing those threads on different processors in parallel, as shown in Figure 4.19.
6. Write a program that uses threads to perform a parallel merge sort.
7. For the threadHello program in Figure 4.6, the procedure go() has the parameter np and the local variable n. Are these variables *per-thread* or *shared* state? Where does the compiler store these variables' states?

8. For the threadHello program in Figure 4.6, the procedure main() has local variables such as i and exitValue. Are these variables *per-thread* or *shared* state? Where does the compiler store these variables?

9. In the *thread-local variables* sidebar, we described how many thread systems have this type of per-thread state.

   Describe how you would implement thread-local variables. Each thread should have an array of 1024 pointers to its thread-local variables.

   a. What would you add to the TCB?
   b. How would you change the thread creation procedure? (For simplicity, assume that when a thread is created, all 1024 entries should be initialized to NULL.)
   c. How would a running thread allocate a new thread-local variable?
   d. In your design, how would a running thread access a particular thread-local variable?

10. For the threadHello program, what is the minimum and maximum number of times that the main thread enters the WAITING state?

11. Using simple threads, write a program that creates several threads and then determines whether the threads package on your system allocates a fixed-size stack for each thread or whether each thread's stack starts at some small size and dynamically grows as needed.

    Hints: You probably want to write a recursive procedure that you can use to consume a large amount of stack memory. You may also want to examine the addresses of variables allocated to different threads' stacks. Finally, you may want to be able to determine how much memory has been allocated to your process; most operating systems have a command or utility that can show the resource consumption of currently running processes (e.g., top in Linux, Activity Monitor in OSX, or Task Manager in Windows).

5. Synchronizing Access to Shared Objects

It is not enough to be industrious. So are the ants. The question is: What are we industrious about? —*Henry David Thoreau*

Multi-threaded programs extend the traditional, single-threaded programming model so that each thread provides a single sequential stream of execution composed of familiar instructions. If a program has *independent threads* that operate on completely separate subsets of memory, we can reason about each thread separately. In this case, reasoning about independent threads differs little from reasoning about a series of independent, single-threaded programs.

However, most multi-threaded programs have both *per-thread state* (e.g., a thread's stack and registers) and *shared state* (e.g., shared variables on the heap). *Cooperating threads* read and write shared state.

Sharing state is useful because it lets threads communicate, coordinate work, and share information. For example, in the Earth Visualizer example in Chapter 4, once one thread finishes downloading a detailed image from the network, it shares that image data with a rendering thread that draws the new image on the screen.

Unfortunately, when cooperating threads share state, writing correct multi-threaded programs becomes much more difficult. Most programmers are used to thinking "sequentially" when reasoning about programs. For example, we often reason about the series of states traversed by a program as a sequence of instructions is executed. However, this sequential model of reasoning does not work in programs with cooperating threads, for three reasons:

1. **Program execution depends on the possible interleavings of threads' access to shared state.** For example, if two threads write a shared variable, one thread with the value 1 and the other with the value 2, the final value of the variable depends on which of the threads' writes finishes last.

   Although this example is simple, the problem is severe because programs need to work for *any possible interleaving*. In particular, recall that thread programmers should not make any assumptions about the relative speed at which their threads operate.

   Worse, as programs grow, there is a combinatorial explosion in the number of possible interleavings.

How can we reason about all possible interleavings of threads' actions in a multi-million line program?

2. **Program execution can be nondeterministic.** Different runs of the same program may produce different results. For example, the scheduler may make different scheduling decisions, the processor may run at a different frequency, or another concurrently running program may affect the cache hit rate. Even common debugging techniques — such as running a program under a debugger, recompiling with the -g option instead of -O, or adding a printf — can change how a program behaves.

   Jim Gray, the 1998 ACM Turing Award winner, coined the term *Heisenbugs* for bugs that disappear or change behavior when you try to examine them. Multi-threaded programming is a common source of Heisenbugs. In contrast, *Bohrbugs* are deterministic and generally much easier to diagnose.

How can we debug programs with behaviors that change across runs?

3. **Compilers and processor hardware can reorder instructions.** Modern compilers and hardware reorder instructions to improve performance. This reordering is generally invisible to single-threaded programs; compilers and processors take care to ensure that dependencies within a single sequence of instructions — that is, within a thread — are preserved. However, reordering can become visible when multiple threads interact through accessing shared variables.

   For example, consider the following code to compute q as a function of p:

```
// Thread 1

p = someComputation();
pInitialized = true;


// Thread 2

while(!pInitialized)
    ;
q = anotherComputation(p);
```

   Although it seems that p is always initialized before anotherComputation(p) is called, this is not the case. To maximize instruction level parallelism, the hardware or compiler may set pInitialized = true before the computation to compute p has completed, and anotherComputation(p) may be computed using an unexpected value.

How can we reason about thread interleavings when compilers and processor hardware may reorder a thread's operations?

Why do compilers and processor hardware reorder operations?

We often find that students are puzzled by the notion that a compiler might produce code, or a processor might execute code, in a way that is correct for a single thread but unpredictable for a multi-threaded program without synchronization.

For compilers, the issue is simple. Modern processors have deep pipelines; they execute many instructions simultaneously by overlapping the instruction fetch, instruction decode, data fetch, arithmetic operation, and conditional branch of a sequence of instructions. The processor stalls when necessary — e.g., if the result of one instruction is needed by the next. Modern compilers will reorder instructions to reduce these stalls as much as possible, provided the reordering does not change the behavior of the program.

The difficulty arises in what assumptions the compiler can make about the code. If the code is single-threaded, it is much easier to analyze possible dependencies between adjacent instructions, allowing more optimization. By contrast, variables in (unsynchronized) multi-threaded code can potentially be read or written by another thread at any point. As the example in the text demonstrated, the precise sequence of seemingly unrelated instructions can potentially affect the behavior of the program. To preserve semantics, instruction re-ordering may no longer be feasible, resulting in more processor stalls and slower code execution.

As long as the programmer uses structured synchronization for protecting shared data, the compiler can reorder instructions as needed without changing program behavior, provided that the compiler does not reorder across synchronization operations. A compiler making the more conservative assumption that all memory is shared would produce slow code even when it was not necessary.

For processor architectures, the issue is also performance. Certain optimizations are possible if the programmer is using structured synchronization but not otherwise. For example, modern processors buffer memory writes to allow instruction execution to continue while the memory is written in the background. If two adjacent instructions issue memory writes to different memory locations, they can occur in parallel and complete out of order. This optimization is safe on a single processor, but potentially unsafe if multiple processors are simultaneously reading and writing the same locations without intervening synchronization. Some processor architectures make the conservative assumption that optimizations should never change program behavior regardless of the programming style — in this case, they stall to prevent reordering. Others make a more optimistic assumption that the programmer is using structured synchronization. For your code to be portable, you should assume that the compiler and the hardware can reorder instructions except across synchronization operations.

Given these challenges, multi-threaded code can introduce subtle, non-deterministic, and non-reproducible bugs. This chapter describes a *structured synchronization* approach to sharing state in multi-threaded programs. Rather than scattering access to shared state throughout the program and attempting *ad hoc* reasoning about what happens when the threads' accesses are interleaved in various ways, a better approach is to: (1) structure the program to facilitate reasoning about concurrency, and (2) use a set of standard synchronization primitives to control access to shared state. This approach gives up some freedom, but if you consistently follow the rules we describe in this chapter, then reasoning about programs with shared state becomes much simpler.

The first part of this chapter elaborates on the challenges faced by multi-threaded programmers and on why it is dangerous to try to reason about all possible thread interleavings in the general, unstructured case. The rest of the chapter describes how to structure shared objects in multi-threaded programs so that we can reason about them. First, we structure a multi-threaded program's shared state as a set of *shared objects* that encapsulate the shared state as well as define and limit how the state can be accessed. Second, to avoid *ad hoc* reasoning about the possible interleavings of access to the state variables within a shared object, we describe how shared objects can use a small set of *synchronization primitives* — locks and condition variables — to coordinate access to their state by different threads. Third, to simplify reasoning about the code in shared objects, we describe a set of *best practices* for writing the code that implements each shared object. Finally, we dive into the details of how to implement synchronization primitives.

Multi-threaded programming has a reputation for being difficult. We agree that it takes care, but this chapter provides a set of simple rules that anyone can follow to implement objects that can be safely shared by multiple threads.

Chapter roadmap:

- **Challenges.** Why is it difficult to reason about multi-threaded programs with unstructured use of shared state? (Section 5.1)
- **Structuring Shared Objects.** How should we structure access to shared state by multiple threads? (Section 5.2)
- **Locks: Mutual Exclusion.** How can we enforce a logical sequence of operations on shared state? (Section 5.3)
- **Condition Variables: Waiting for a Change.** How does a thread wait for a change in shared state? (Section 5.4)
- **Designing and Implementing Shared Objects.** Given locks and condition variables, what is a good way to write and reason about the code for shared objects? (Section 5.5)
- **Three Case Studies.** We illustrate our methodology by using it to develop solutions to three concurrent programming challenges. (Section 5.6)
- **Implementing Synchronization Primitives.** How are locks and condition variables implemented? (Section 5.7)
- **Semaphores Considered Harmful.** What other synchronization primitives are possible, and how do they relate to locks and condition variables? (Section 5.8)

5.1 Challenges

We began this chapter with the core challenge of multi-threaded programming: a multi-threaded program's execution depends on the interleavings of different threads' access to shared memory, which can make it difficult to reason about or debug these programs. In particular, cooperating threads' execution may be affected by *race conditions.*

5.1.1 Race Conditions

A *race condition* occurs when the behavior of a program depends on the interleaving of operations of different threads. In effect, the threads run a race between their operations, and the results of the program execution depends on who wins the race.

Reasoning about even simple programs with race conditions can be difficult. To appreciate this, we now look at three extremely simple multi-threaded programs.

The world's simplest cooperating-threads program. Suppose we run a program with two threads that do the following:

Thread A	**Thread B**
x = 1;	x = 2;

EXAMPLE: What are the possible final values of x?

ANSWER: The result can be **x = 1 or x = 2** depending on which thread wins or loses the "race" to set x. □

That was easy, so let's try one that is a bit more interesting.

The world's second-simplest cooperating-threads program. Suppose that initially y = 12, and we run a program with two threads that do the following:

Thread A	**Thread B**
x = y + 1;	y = y * 2;

EXAMPLE: What are the possible final values of x?

ANSWER: The result is **x = 13 if Thread A executes first or x = 25 if Thread B executes first.** More precisely, we get x = 13 if Thread A reads y before Thread B updates y, or we get x = 25 if Thread B updates y before Thread A reads y. □

The world's third-simplest cooperating-threads program. Suppose that initially x = 0 and we run a program with two threads that do the following:

Thread A	**Thread B**
x = x + 1;	x = x + 2;

EXAMPLE: What are the possible final values of x?

ANSWER: Obviously, **one possible outcome is x = 3.** For example, Thread A runs to completion and then Thread B starts and runs to completion. However, **we can also get x = 2 or x = 1.** In particular, when we write a single statement like x = x + 1, compilers on many processors produce multiple instructions, such as: (1) load memory location x into a register, (2) add 1 to that register, and (3) store the result to memory location x. If we disassemble the above program into simple pseudo-assembly-code, we can see some of the possibilities.

One Interleaving

Thread A	**Thread B**
load r1, x	
add r2, r1, 1	
store x, r2	
	load r1, x
	add r2, r1, 2
	store x, r2

final: x == 3

Another Interleaving

Thread A	Thread B
load r1, x	
	load r1, x
add r2, r1, 1	
	add r2, r1, 2
store x, r2	
	store x, r2

final: x == 2

Yet Another Interleaving

Thread A	Thread B
load r1, x	
	load r1, x
add r2, r1, 1	
	add r2, r1, 2
	store x, r2
store x, r2	

final: x == 1

□

Even for this 2-line program, the complexity of reasoning about race conditions and interleavings is beginning to grow. Not only would one have to reason about all possible interleavings of statements, but one would also have to disassemble the program and reason about all possible interleavings of assembly instructions. (And if the compiler and hardware can reorder instructions, there are even more possibilities to consider.)

The Case of the Therac-25

The Therac-25 was a cancer therapy device, designed to deliver very high doses of radiation to a targeted region of the body in an attempt to eliminate cancer cells before they had a chance to spread. Over a several year period in the mid-1980's, a computer malfunction caused six separate patients to receive an estimated 100 times the intended dose of radiation. Three of the patients later died as a result; the others sustained serious but non-fatal injuries.

Although there were many contributing factors to the malfunction, a race condition was at the heart of both the overdose and the delay in recognizing and repairing the problem. The Therac-25 was designed to check in software that the entered dosage was medically safe before using it to configure the radiation beam. However, the software was also concurrent: the operator interface code could run at the same time that the dosage was being checked and used, with no locking or other synchronization. In rare cases, the dosage could be changed after the check and before the use, and due to a separate user interface bug, the operator could enter an overdose without either intending or realizing it.

Because the problem required a rare sequence of events, the machine appeared to work successfully for almost all patients. Years elapsed between the first incident and the final one, and during this period, the manufacturer repeatedly insisted that no overdose was possible and that the patient injuries must be due to some other factor. It took the second occurrence of the race condition at the same hospital to help reveal the system's design flaw.

5.1.2 Atomic Operations

When we disassembled the code in last example, we could reason about *atomic operations*, indivisible operations that cannot be interleaved with or split by other operations.

On most modern architectures, a load or store of a 32-bit word from or to memory is an atomic operation. So, the previous analysis reasoned about interleaving of atomic loads and stores to memory.

Conversely, a load or store is not always an atomic operation. Depending on the hardware implementation, if two threads store the value of a 64-bit floating point register to a memory address, the final result might be the first value, the second value, or a mix of the two.

5.1.3 Too Much Milk

Although one could, in principle, reason carefully about the possible interleavings of different threads' atomic loads and stores, doing so is tricky and error-prone. Later, we present a higher level abstraction for synchronizing threads, but first we illustrate the problems with using atomic loads and stores using a simple problem called, "Too Much

Milk.” The example is intentionally simple; real-world concurrent programs are often much more complex. Even so, the example shows the difficulty of reasoning about interleaved access to shared state.

The Too Much Milk problem models two roommates who share a refrigerator and who — as good roommates — make sure the refrigerator is always well stocked with milk. With such responsible roommates, the following scenario is possible:

	Roommate 1’s actions	**Roommate 2’s actions**
3:00	Look in fridge; out of milk.	
3:05	Leave for store.	
3:10	Arrive at store.	Look in fridge; out of milk.
3:15	Buy milk.	Leave for store.
3:20	Arrive home; put milk away.	Arrive at store.
3:25		Buy milk.
3:30		Arrive home; put milk away.
3:35		Oh no!

We can model each roommate as a thread and the number of bottles of milk in the fridge with a variable in memory. If the only atomic operations on shared state are atomic loads and stores to memory, is there a solution to the Too Much Milk problem that ensures both *safety* (the program never enters a bad state) and *liveness* (the program eventually enters a good state)? Here, we strive for the following properties:

- **Safety:** Never more than one person buys milk.
- **Liveness:** If milk is needed, someone eventually buys it.

WARNING: **Simplifying Assumption.** Throughout the analysis in this section, we assume that the instructions are executed in exactly the order written, i.e., neither the compiler nor the architecture reorders instructions. This assumption is crucial for reasoning about the order of atomic load and store operations, but many modern compilers and architectures violate it, so be extremely careful applying the style of analysis we present here to your own programs.

Solution 1. The basic idea is for a roommate to leave a note on the fridge before going to the store. The simplest way to leave this note — given our programming model that we have shared memory on which we can perform atomic loads and stores — is to set a flag when going to buy milk and to check this flag before going to buy milk. Each thread might run the following code:

```
if (milk==0) {        // if no milk
    if (note==0) {    // if no note
        note = 1;     // leave note
        milk++;       // buy milk
        note = 0;     // remove note
    }
}
```

Unfortunately, this implementation can violate safety. For example, the first thread could execute everything up to and including the check of the milk value and then get context switched. Then, the second thread could run through all of this code and buy milk. Finally, the first thread could be re-scheduled, see that note is zero, leave the note, buy more milk, and remove the note, leaving the system with milk == 2.

```
// Thread A                        // Thread B
if (milk==0) {
                                   if (milk==0) {
                                       if (note==0) {
                                           note = 1;
                                           milk++;
                                           note = 0;
                                       }
                                   }
    if (note==0) {
        note = 1;
        milk++;
        note = 0;
    }
}
```

```
Oh no!
```

This “solution” makes the problem worse! The preceding code usually works, but it may fail occasionally when the scheduler does just the right (or wrong) thing. We have created a Heisenbug that causes the program to occasionally fail in ways that may be very difficult to reproduce (e.g., probably only when the grader is looking at it or when the CEO is demonstrating a new product at a trade show).

Solution 2. In solution 1, the roommate checks the note before setting it. This opens up the possibility that one roommate has already made a decision to buy milk before notifying the other roommate of that decision. If we use two variables for the notes, a

roommate can create a note before checking the other note and the milk and making a decision to buy. For example, we can do the following:

Path A

```
noteA = 1;          // leave note
if (noteB==0) {     // if no note  A1
    if (milk==0) { // if no milk  A2
        milk++;     // buy milk    A3
    }
}
noteA = 0;          // remove note A
```

Path B

```
noteB = 1;          // leave note
if (noteA==0) {     // if no note  B1
    if (milk==0) { // if no milk  B2
        milk++;     // buy milk    B3
    }               //             B4
}                   //             B5
noteB = 0;          // remove note
```

If the first thread executes the Path A code and the second thread executes the Path B code, this protocol is safe; by having each thread write a note (“I might buy milk”) before deciding to buy milk, we ensure the safety property: at most one thread buys milk.

Although this intuition is solid, proving the safety property without enumerating all possible interleavings requires care.

Safety Proof. Assume for the sake of contradiction that the algorithm is *not* safe — both A and B buy milk. Consider the state of the two variables (noteB, milk) when thread A is at the line marked **A1**, at the precise moment when the atomic load of noteB from shared memory to A’s register occurs. There are three cases to consider:

- **Case 1:** (noteB = 1, milk = any value). This state contradicts the assumption that thread A buys milk and reaches **A3**.
- **Case 2:** (noteB = 0, milk > 0). In this simple program, the property milk > 0 is a *stable property* — once it becomes true, it remains true forever. Thus, if milk > 0 is true when A is at **A1**, A’s test at line **A2** will fail, and A will not buy milk, contradicting our assumption.
- **Case 3:** (noteB = 0, milk = 0). We know that thread B must not currently be executing any of the lines marked **B1-B5**. We also know that either noteA == 1 or milk > 0 will be true from this time forward (noteA OR milk is also a stable property). This means that B cannot buy milk in the future (either the test at B1 or B2 must fail), which contradicts our assumption that both A and B buy milk.

Since every case contradicts the assumption, the algorithm is safe. □

Liveness. Unfortunately, Solution 2 does not ensure liveness. In particular, it is possible for both threads to set their respective notes, for each thread to check the other thread’s note, and for both threads to decide not to buy milk.

This brings us to Solution 3.

Solution 3. Solution 2 was safe because a thread would avoid buying milk if there were any chance that the other thread *might* buy milk. For Solution 3, we ensure that at least one of the threads determines whether the other thread has bought milk or not before deciding whether or not to buy. In particular, we do the following:

Path A

```
noteA = 1;          // leave note A
while (noteB==1) { // wait for no note B
    ;               // spin
}
if (milk==0) {      // if no milk M
    milk++;         // buy milk
}
noteA = 0;          // remove note A
```

Path B

```
noteB = 1;           // leave note B
if (noteA==0) {      // if no note A
    if (milk==0) {  // if no milk
        milk++;      // buy milk
    }                //
}                    //
noteB = 0;           // remove note B
```

We can show that Solution 3 is safe using an argument similar to the one we used for Solution 2.

To show that Solution 3 is live, observe that code path B has no loops, so eventually thread B must finish executing the listed code. Eventually, noteB == 0 becomes true and remains true. Therefore, thread A must eventually reach line **M** and decide whether to buy milk. If it finds M == 1, then milk has been bought. If it finds M == 0, then it will buy milk. Either way, the liveness property — that if needed, some milk is bought — is met.

5.1.4 Discussion

Assuming that the compiler and processor execute instructions in program order, the preceding proof shows that it is possible to devise a solution to Too Much Milk that is both safe and live using nothing but atomic load and store operations on shared memory.

Although the solution we presented only works for two roommates, there is a generalization, called Peterson's algorithm, which works with any fixed number of n threads. More details on Peterson's algorithm can be found elsewhere (e.g., http://en.wikipedia.org/wiki/Peterson's_algorithm).

However, our solution for Too Much Milk (and likewise Peterson's algorithm) is not terribly satisfying:

- The solution is *complex* and requires careful reasoning to be convinced that it works.
- The solution is *inefficient*. In Too Much Milk, while thread A is waiting, it is *busy-waiting* and consuming CPU resources. In Peterson's generalized solution, *all* n threads can busy-wait. Busy-waiting is particularly problematic on modern systems with preemptive multi-threading, as the spinning thread may be holding the processor waiting for an event that cannot occur until some preempted thread is re-scheduled to run.
- The solution *may fail* if the compiler or hardware reorders instructions. This limitation can be addressed by using *memory barriers* (see sidebar). Adding memory barriers would further increase the implementation complexity of the algorithm; barriers do not address the other limitations just mentioned.

Memory barriers

Suppose you are writing low-level code that must reason about the ordering of memory operations. How can this be done on modern hardware and with modern compilers?

A *memory barrier* instruction prevents the compiler and hardware from reordering memory accesses across the barrier — no accesses before the barrier are moved after the barrier and no accesses after the barrier are moved before the barrier. One can add memory barriers to the Too Much Milk solution or to Peterson's algorithm to get code that works on modern machines with modern compilers. Of course, this makes the code even more complex.

Details of how to issue a memory barrier instruction depend on hardware and compiler details. However, a good example is gcc's __sync_synchronize() builtin, which tells the compiler not to reorder memory accesses across the barrier and to issue processor-specific instructions that the underlying hardware treats as a memory barrier.

5.1.5 A Better Solution

The next section describes a better approach to writing programs in which multiple threads access shared state. We write *shared objects* that use *synchronization objects* to coordinate different threads' access to shared state.

Suppose, for example, we had a primitive called a *lock* that only one thread at a time can own. Then, we can solve the Too Much Milk problem by defining the class for a Kitchen object with the following method:

```
Kitchen::buyIfNeeded() {
    lock.acquire();
    if (milk == 0) {      // if no milk
        milk++;           // buy milk
    }
    lock.release();
}
```

After outlining a strategy for managing synchronization in the next section, we define locks and condition variables (another type of synchronization object) in Sections 5.3 and 5.4.

5.2 Structuring Shared Objects

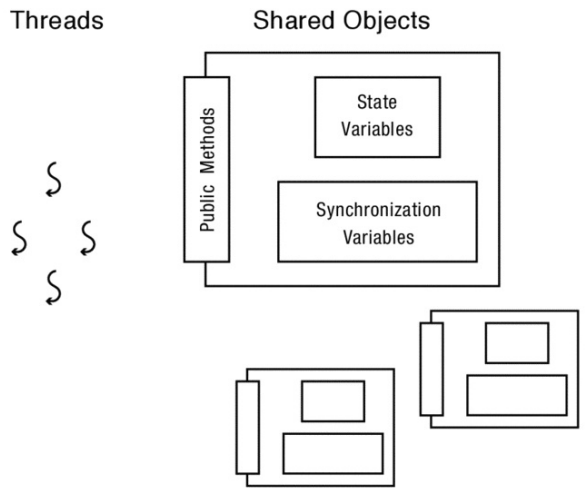

Figure 5.1: In a multi-threaded program, threads are separate from and operate concurrently on shared objects. Shared objects contain both shared state and synchronization variables, used for controlling concurrent access to shared state.

Decades of work have developed a much simpler approach to writing multi-threaded programs than using just atomic loads and stores. This approach extends the modularity of object-oriented programming to multi-threaded programs. As Figure 5.1 illustrates, a multi-threaded program is built using *shared objects* and a set of threads that operate on them.

Shared objects are objects that can be accessed safely by multiple threads. All shared state in a program — including variables allocated on the heap (e.g., objects allocated with malloc or new) and static, global variables — should be encapsulated in one or more

shared objects.

Programming with shared objects extends traditional object-oriented programming, in which objects hide their implementation details behind a clean interface. In the same way, shared objects hide the details of synchronizing the actions of multiple threads behind a clean interface. The threads using shared objects need only understand the interface; they do not need to know how the shared object internally handles synchronization.

Like regular objects, programmers can design shared objects for whatever modules, interfaces, and semantics an application needs. Each shared object's class defines a set of public methods on which threads operate. To assemble the overall program from these shared objects, each thread executes a "main loop" written in terms of actions on public methods of shared objects.

Since shared objects encapsulate the program's shared state, the main loop code that defines a thread's high-level actions need not concern itself with synchronization details. The programming model thus looks very similar to that for single-threaded code.

Shared objects, monitors, and syntactic sugar

We focus on *shared objects* because object-oriented programming provides a good way to think about shared state: hide shared state behind public methods that provide a clean interface to threads and that handle the details of synchronization.

Although we use object-oriented terminology in our discussion, the ideas are equally applicable to non-object-oriented languages. For example, where a C++ program might define a class of shared objects with public methods, a C program might define a struct with synchronization variables and state variables as fields. Rather than scattering the code that accesses the struct's fields, a well-designed C program will have a fixed set of functions that operate on the struct's fields.

Conversely, some programming languages build in even more support for shared objects than we describe here. When a programming language includes support for shared objects, a shared object is often called a *monitor*. Early languages with monitors include Brinch Hansen's Concurrent Pascal and Xerox PARC's Mesa; today, Java supports monitors via the synchronized keyword.

We regard the distinctions between procedural languages, object-oriented languages, and languages with built-in support for monitors as relatively unimportant syntactic sugar — they are just a different way of writing the same thing. We use the terms "shared objects" or "monitors" broadly to refer to a conceptual approach that can and should be used to manage concurrency regardless of the particular programming language.

In this book, our code and pseudo-code are based on C++'s syntax. We believe provides the right level of detail for teaching the shared objects or monitors approach. We prefer teaching with C++ to Java because we want to explicitly show where locks and condition variables are allocated and accessed rather than relying on operations hidden by a language's built in monitor syntax. Conversely, we prefer C++ to C because we think C++'s support for object-oriented programming may help you internalize the underlying philosophy of the shared object approach.

5.2.1 Implementing Shared Objects

Of course, internally the shared objects must handle the details of synchronization. As Figure 5.2 shows, shared objects are implemented in layers.

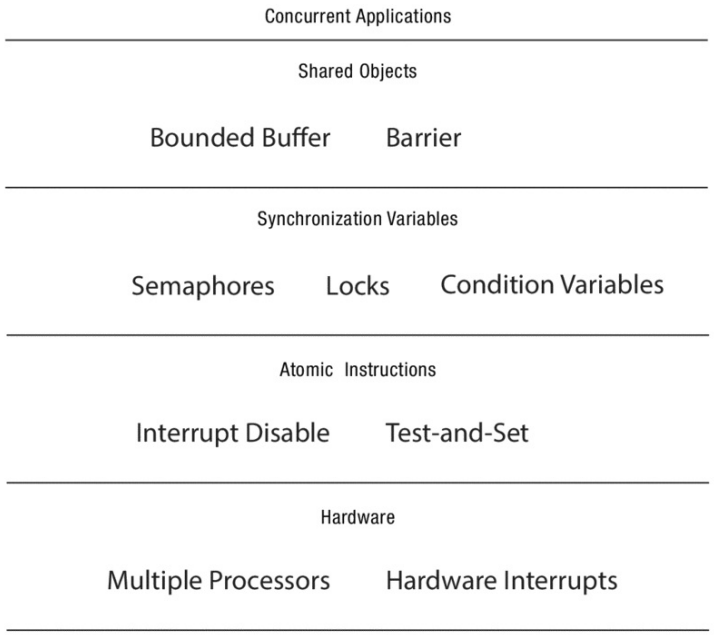

Figure 5.2: Multi-threaded programs are built with shared objects. Shared objects are built using synchronization variables and state variables. Synchronization variables are implemented using specialized processor instructions to manage interrupt delivery and to atomically read-modify-write memory locations.

- **Shared object layer.** As in standard object-oriented programming, shared objects define application-specific logic and hide internal implementation details. Externally, they appear to have the same interface as you would define for a single-threaded program.
- **Synchronization variable layer.** Rather than implementing shared objects directly with carefully interleaved atomic loads and stores, shared objects include *synchronization variables* as member variables. Synchronization variables, stored in memory just like any other object, can be included in any data structure.

  A *synchronization variable* is a data structure used for coordinating concurrent access to shared state. Both the interface and the implementation of synchronization

variables must be carefully designed. In particular, we build shared objects using two types of synchronization variables: *locks* and *condition variables*. We define these and describe how to construct them in Sections 5.3 and 5.4.

Synchronization variables coordinate access to *state variables*, which are just the normal member variables of an object that you are familiar with from single-threaded programming (e.g., integers, strings, arrays, and pointers).

Using synchronization variables simplifies implementing shared objects. In fact, not only do shared objects externally resemble traditional single-threaded objects, but, by implementing them with synchronization variables, their internal implementations are quite similar to those of single-threaded programs.

- **Atomic instruction layer.** Although the layers above benefit from a simpler programming model, it is not turtles all the way down. Internally, synchronization variables must manage the interleavings of different threads' actions.

  Rather than implementing synchronization variables, such as locks and condition variables, using atomic loads and stores as we tried to do for the Too Much Milk problem, modern implementations build synchronization variables using *atomic read-modify-write instructions*. These processor-specific instructions let one thread have temporarily exclusive and atomic access to a memory location while the instruction executes. Typically, the instruction atomically reads a memory location, does some simple arithmetic operation to the value, and stores the result. The hardware guarantees that any other thread's instructions accessing the same memory location will occur either entirely before, or entirely after, the atomic read-modify-write instruction.

5.2.2 Scope and Roadmap

As Figure 5.2 indicates, concurrent programs are built on top of shared objects. The rest of this chapter focuses on the middle layers of the figure — how to build shared objects using synchronization objects and how to build synchronization objects out of atomic read-modify-write instructions. Chapter 6 discusses issues that arise when composing multiple shared objects into a larger program.

5.3 Locks: Mutual Exclusion

A *lock* is a synchronization variable that provides *mutual exclusion* — when one thread holds a lock, no other thread can hold it (i.e., other threads are *excluded*). A program associates each lock with some subset of shared state and requires a thread to hold the lock when accessing that state. Then, only one thread can access the shared state at a time.

Mutual exclusion greatly simplifies reasoning about programs because a thread can perform an arbitrary set of operations while holding a lock, and those operations *appear to be atomic* to other threads. In particular, because a lock enforces mutual exclusion and threads must hold the lock to access shared state, no other thread can observe an intermediate state. Other threads can only observe the state left after the lock release.

EXAMPLE: Locking to group multiple operations. Consider, for example, a bank account object that includes a list of transactions and a total balance. To add a new transaction, we acquire the account's lock, append the new transaction to the list, read the old balance, modify it, write the new balance, and release the lock. To query the balance and list of recent transactions, we acquire the account's lock, read the recent transactions from the list, read the balance, and release the lock. Using locks in this way guarantees that one update or query completes before the next one starts. Every query always reflects the complete set of recent transactions.

Another example of grouping is when printing output. Without locking, if two threads called printf at the same time, the individual characters of the two messages could be interleaved, garbling their meaning. Instead, on modern multi-threaded operating systems, printf uses a lock to ensure that the group of characters in each message prints as a unit.

It is much easier to reason about interleavings of atomic groups of operations rather than interleavings of individual operations for two reasons. First, there are (obviously) fewer interleavings to consider. Reasoning about interleavings on a coarser-grained basis reduces the sheer number of cases to consider. Second, and more important, we can make each atomic group of operations correspond to the logical structure of the program, which allows us to reason about *invariants* not specific *interleavings*.

In particular, shared objects usually have one lock guarding all of an object's state. Each public method acquires the lock on entry and releases the lock on exit. Thus, reasoning about a shared class's code is similar to reasoning about a traditional class's code: we assume a set of invariants when a public method is called and re-establish those invariants before a public method returns. If we define our invariants well, we can then reason about each method independently.

5.3.1 Locks: API and Properties

A lock enables mutual exclusion by providing two methods: Lock::acquire() and Lock::release(). These methods are defined as follows:

- A lock can be in one of two states: BUSY or FREE.
- A lock is initially in the FREE state.
- Lock::acquire waits until the lock is FREE and then atomically makes the lock BUSY.

  Checking the state to see if it is FREE and setting the state to BUSY are together an *atomic operation*. Even if multiple threads try to acquire the lock, at most one thread will succeed. One thread observes that the lock is FREE and sets it to BUSY; the other threads just see that the lock is BUSY and wait.
- Lock::release makes the lock FREE. If there are pending `acquire` operations, this state change causes one of them to proceed.

We describe how to implement locks with these properties in Section 5.7. Using locks makes solving the Too Much Milk problem trivial. Both threads run the following code:

```
lock.acquire();
if (milk == 0) {     // if no milk
    milk++;          // buy milk
}
lock.release();
```

EXAMPLE: Many routines in an operating system kernel need to allocate and de-allocate memory blocks. Assuming you are given the code for a single-threaded kernel memory allocator, explain how to implement a thread-safe memory allocator.

ANSWER: Using C malloc and free as an example, we can convert them to be thread-safe by acquiring a lock before accessing the heap, and releasing it after the block has been allocated or freed. Since malloc and free read and modify the same data structures, it is essential to use the *same* lock in both procedures, heaplock.

```
char *malloc (int n) {
    char *p;

    heaplock.acquire();
 // Code for single-threaded malloc()
 // p = allocate block of memory
 //     of size n.
    heaplock.release();
    return p;
}


void free (char *p) {


    heaplock.acquire();
 // Code for single-threaded free()
 // Put p back on free list.

    heaplock.release();

}
```

□

Formal properties. A lock can be defined more precisely as follows. A thread *holds a lock* if it has returned from a lock's `acquire` method more often than it has returned from a lock's `release` method. A thread *is attempting to acquire* a lock if it has called but not yet returned from a call to `acquire` on the lock.

A lock should ensure the following three properties:

1. **Mutual Exclusion.** At most one thread holds the lock.
2. **Progress.** If no thread holds the lock and any thread attempts to acquire the lock, then eventually some thread succeeds in acquiring the lock.
3. **Bounded waiting.** If thread T attempts to acquire a lock, then there exists a bound on the number of times other threads can successfully acquire the lock before T does.

Mutual exclusion is a safety property because locks prevent more than one thread from accessing shared state.

Progress and bounded waiting are liveness properties. If a lock is FREE, *some* thread must be able to acquire it. Further, any *particular* thread that wants to acquire the lock must eventually succeed in doing so.

If these definitions sound stilted, it is because we have carefully crafted them to avoid introducing subtle corner cases. For example, if a thread holding a lock never releases it, other threads cannot make progress, so we define the *bounded waiting* condition in terms of successful `acquire` operations.

WARNING: **Non-property: Thread ordering.** The *bounded waiting* property defined above guarantees that a thread will eventually get a chance to acquire the lock. However, it does not promise that waiting threads acquire the lock in FIFO order. Most implementations of locks that you will encounter — for example with POSIX threads — do not provide FIFO ordering.

5.3.2 Case Study: Thread-Safe Bounded Queue

As in standard object-oriented programming, each shared object is an instance of a class that defines the class's state and the methods that operate on that state.

The class's state includes both state variables (e.g., ints, floats, strings, arrays, and pointers) and synchronization variables (e.g., locks). Every time a class constructor produces another instance of a shared object, it allocates both a new lock and new instances of the state protected by that lock.

A *bounded queue* is a queue with a fixed size limit on the number of items stored in the queue. Operating system kernels use bounded queues for managing interprocess communication, TCP and UDP sockets, and I/O requests. Because the kernel runs in a finite physical memory, the kernel must be designed to work properly with finite resources. For example, instead of a simple, infinite buffer between a producer and a consumer thread, the kernel will instead use a limited size buffer, or bounded queue.

A *thread-safe bounded queue* is a type of a bounded queue that is safe to call from multiple concurrent threads. Figure 5.3 gives an implementation; it lets any number of threads safely insert and remove items from the queue. As Figure 5.4 illustrates, a program can allocate multiple such queues (e.g., queue1, queue2, and queue3), each of which includes its own lock and state variables.

```
// Thread-safe queue interface

const int MAX = 10;
```

```
class TSQueue {
  // Synchronization variables
    Lock lock;

  // State variables
    int items[MAX];
    int front;
    int nextEmpty;

  public:
    TSQueue();
    ~TSQueue(){};
    bool tryInsert(int item);
    bool tryRemove(int *item);
};

// Initialize the queue to empty
// and the lock to free.
TSQueue::TSQueue() {
    front = nextEmpty = 0;
}

// Try to insert an item. If the queue is
// full, return false; otherwise return true.
bool
TSQueue::tryInsert(int item) {
    bool success = false;

    lock.acquire();
    if ((nextEmpty - front) < MAX) {
        items[nextEmpty % MAX] = item;
        nextEmpty++;
        success = true;
    }
    lock.release();
    return success;
}

// Try to remove an item. If the queue is
// empty, return false; otherwise return true.
bool
TSQueue::tryRemove(int *item) {
    bool success = false;

    lock.acquire();
    if (front < nextEmpty) {
        *item = items[front % MAX];
        front++;
        success = true;
    }
    lock.release();
    return success;
}
```

Figure 5.3: A thread-safe bounded queue. For implementation simplicity, we assume the queue stores integers (rather than arbitrary objects) and the total number of items stored is modest.

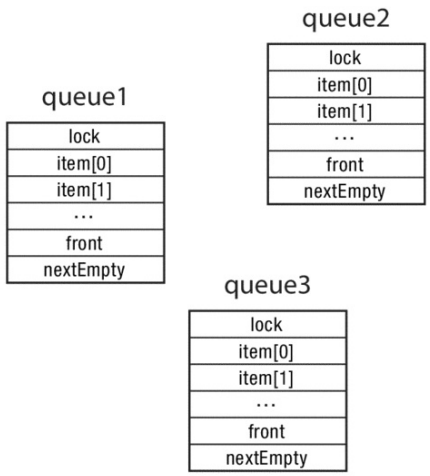

Figure 5.4: Three shared objects, each an instance of class TSQueue.

The queue stores only a fixed number, MAX, of items. When the queue is full, an insert request returns an error. Similarly, when the queue is empty, a remove request returns an error. Section 5.4 shows how *condition variables* let the calling thread *wait* instead of returning an error. On insert, the thread waits until the queue has space to store the item and, on remove, it waits until the queue has at least one item queued before returning it.

The TSQueue implementation defines a circular queue that stores data in a fixed size array, items[MAX]. The state variable, front is the next item in the queue to be removed, if any; nextEmpty is the next location for a new item, if any. To keep the example as simple as possible, only items of type int can be stored in and removed from the queue, and we assume the total number of items stored fits within a 64 bit integer.

All of these variables are as they would be for a single-threaded version of this object. The lock allows tryInsert and tryRemove to atomically read and write multiple variables just as a single-threaded version would.

EXAMPLE: What constraints are true of TSQueue at the moment immediately after the lock is acquired? What constraints hold immediately before the lock is released?

ANSWER: Because the lock enforces mutual exclusion and is always held whenever a thread modifies a state variable, when the lock is acquired the object's state variables must be either: (i) in the initial state or (ii) in the state left by a previous thread when it released the lock. These constraints are the same as for single-threaded code using a bounded queue:

- The total number of items ever inserted in the queue is nextEmpty.
- The total number of items ever removed from the queue is front.
- front <= nextEmpty
- The current number of items in the queue is nextEmpty - front.
- nextEmpty - front <= MAX

The lock holder always re-establishes these constraints before releasing the lock. □

EXAMPLE: Are these constraints also true if the lock is not held?

ANSWER: No. It seems intuitive that if the constraints hold immediately before the lock is released, then they must also hold immediately after the lock is released. However, this is not the case. In the meantime, some other thread may have acquired the lock and may be in the process of modifying the state variables. In general, if the lock is not held, one cannot say *anything* about the object's state variables. □

Critical Sections

A *critical section* is a sequence of code that atomically accesses shared state. By ensuring that a thread holds the object's lock while executing any of its critical sections, we ensure that each critical section appears to execute atomically on its shared state. There is a critical section in each of the methods tryInsert and tryRemove.

Notice two things:

- Each class can define multiple methods that operate on the shared state defined by the class, so there may be *multiple critical sections per class*. However, for each instance of the class (i.e., for each object), only one thread holds the object's lock at a time, so *only one thread actively executes any of the critical sections per shared object instance.* For the TSQueue class, if one thread calls queue1.tryInsert and another calls queue1.tryRemove, the insert occurs either before the remove or vice versa.
- A program can create *multiple instances of a class*. Each instance is a shared object, and each shared object has its own lock. Thus, different threads may be active in the critical sections for different shared object instances. For the TSQueue class, if one thread calls queue1.tryInsert, another thread calls queue2.tryRemove, and a third thread calls queue3.tryInsert, all three threads may be simultaneously executing critical section code operating on *different instances* of the TSQueue class.

Using Shared Objects

Shared objects are allocated in the same way as other objects. They can be dynamically allocated from the heap using malloc and new, or they can be statically allocated in global memory by declaring static variables in the program.

Multiple threads must be able to access shared objects. If shared objects are global variables, then a thread's code can refer to an object's global name to reference it; the compiler computes the corresponding address. If shared objects are dynamically allocated, then each thread that uses an object needs a pointer or reference to it.

Two common ways to provide a thread a pointer to a shared object are: (1) provide a pointer to the shared object when the thread is created, and (2) store references to shared objects in other shared objects (e.g., containers). For example, a program might have a global, shared (and synchronized!) hash table that threads can use to store and retrieve references to other shared objects.

Figure 5.5 shows a simple program that creates three queues and then creates some threads that insert into these queues. It then removes 20 items from each queue and prints the values it removes. The initial main thread allocates the shared queues on the heap using new, and provides each worker thread a pointer to one of the shared queues.

```
// TSQueueMain.cc
//   Test code for TSQueue.

int main(int argc, char **argv) {
    TSQueue *queues[3];
    sthread_t workers[3];
    int i, j;

    // Start worker threads to insert.
    for (i = 0; i < 3; i++) {
        queues[i] = new TSQueue();
        thread_create_p(&workers[i],
                    putSome, queues[i]);
    }

    // Wait for some items to be put.
    thread_join(workers[0]);

    // Remove 20 items from each queue.
    for (i = 0; i < 3; i++) {
        printf("Queue %d:\n", i);
        testRemoval(&queues[i]);
    }
}

// Insert 50 items into a queue.
void *putSome(void *p) {
    TSQueue *queue = (TSQueue *)p;
    int i;

    for (i = 0; i < 50; i++) {
        queue->tryInsert(i);
    }
    return NULL;
}

// Remove 20 items from a queue.
void testRemoval(TSQueue *queue) {
    int i, item;

    for (i = 0; i < 20; j++) {
```

```
        if (queue->tryRemove(&item))
          printf("Removed %d\n", item);
        else
          printf("Nothing there.\n");
        }
    }
}
```

Figure 5.5: This code creates three TSQueue objects and then adds and removes some items from these queues. We use thread_create_p instead of thread_create so that we can pass to the newly created thread a pointer to the queue it should use.

WARNING: **Put shared objects on the heap, not the stack.** While nothing prevents you from writing a program that allocates a shared object as an automatic variable in a procedure or method, you should not write programs that do this. The compiler allocates automatic variables (sometimes called "local variables", with good reason) on the stack during procedure invocation. If one thread passes a pointer or reference to one of its automatic variables to another thread and later returns from the procedure where the automatic variable was allocated, then that second thread now has a pointer into a region of the first thread's stack that may be used for other purposes. To prevent this error, a few garbage-collected languages, such as Google's Go, automatically convert all automatic data to being heap-allocated if the data can be referenced outside of the procedure.

You might be tempted to argue that, for a particular program, you know that the procedure will never return until all of the threads with which it is sharing an object are done using that object, and that therefore sharing one of the procedure's local variables is safe. The problem with this argument is that the code may change over time, introducing a dangerous and subtle bug. When sharing dynamically allocated variables, it is best to stay in the habit of sharing variables only from the heap — and never sharing variables from the stack — across threads.

5.4 Condition Variables: Waiting for a Change

Condition variables provide a way for one thread to wait for another thread to take some action. For example, in the thread-safe queue example in Figure 5.3, rather than returning an error when we try to remove an item from an empty queue, we might wait until the queue is non-empty, and then always return an item.

Similarly, a web server might wait until a new request arrives; a word processor might wait for a key to be pressed; a weather simulator's coordinator thread might wait for the worker threads calculating temperatures in each region to finish; or, in our Earth Visualizer example, a thread in charge of rendering part of the screen might wait for a user command or for new data to update the view.

In all of these cases, we want a thread to wait for some action to change the system state so that the thread can make progress.

```
int
TSQueue::remove() {
    int item;
    bool success;

    do {
        success = tryRemove(&item);
    } until(success);
    return item;
}
```

Figure 5.6: A polling-based implementation of TSQueue::remove. The code retries in a loop until it succeeds in removing an item.

One way for a thread to wait would be to poll — to repeatedly check the shared state to see if it has changed. As shown in Figure 5.6, a polling implementation of remove would have a simple wrapper that repeatedly calls tryRemove until it returns success. Unfortunately, this approach is inefficient: the waiting thread continually loops, or busy-waits, consuming processor cycles without making useful progress. Worse, busy-waiting can delay the scheduling of other threads — perhaps exactly the thread for which the looping thread is waiting.

The sleep fix?

We often find that students want to "fix" the polling-based approach by adding a delay. For example, in Figure 5.6, we could add a call to sleep to yield the processor for (say) 100 ms after each unsuccessful tryRemove call. This would allow some other thread to run while the waiting thread is waiting.

This approach has two problems. First, although it reduces the inefficiency of polling, it does not eliminate it. Suspending and scheduling a thread imposes non-trivial overheads, and a program with many polling threads would still waste significant resources. Second, periodic polling adds latency. In our Earth Visualizer example, if the thread waiting for keyboard input waited 100 ms between each check, the application might become noticeably more sluggish.

As an extreme example, one of the authors once had an employee implement a network server that provided several layers of processing, where each layer had a thread that received work from the layer above and sent the work to the layer below. Measurements of the server showed surprisingly bad performance; we expected each request to take a few milliseconds, but instead each took just over half a second. Fortunately, the performance was so poor that it was easy to track down the problem: layers passed work to each other through bounded queues much like TSQueue, but the queue remove method was implemented as a polling loop with a 100 ms delay. With five such layers of processing, the server became unusable. Fortunately, the fix was simple: use condition variables instead.

5.4.1 Condition Variable Definition

A *condition variable* is a synchronization object that lets a thread efficiently wait for a change to shared state that is protected by a lock. A condition variable has three methods:

- **CV::wait(Lock *lock).** This call atomically *releases the lock* and *suspends execution of the calling thread*, placing the calling thread on the condition variable's waiting list. Later, when the calling thread is re-enabled, it *re-acquires the lock* before returning from the `wait` call.
- **CV::signal().** This call takes one thread off the condition variable's waiting list and marks it as eligible to run (i.e., it puts the thread on the scheduler's ready list). If no threads are on the waiting list, `signal` has no effect.
- **CV::broadcast().** This call takes all threads off the condition variable's waiting list and marks them as eligible to run. If no threads are on the waiting list, `broadcast` has no effect.

WARNING: Note that condition variable `wait` and `signal` are different from the UNIX system calls wait and signal. The nomenclature is unfortunate but longstanding. In this book, we always use the terms, UNIX wait and UNIX signal, to refer to the UNIX variants, and simple `wait` and `signal` to refer to condition variable operations.

A condition variable is used to wait for a change to shared state, and a lock must always protect updates to shared state. Thus, the condition variable API is designed to work in concert with locks. All three methods (`wait`, `signal`, and `broadcast`) should only be called while the associated lock is held.

```
SharedObject::someMethodThatWaits() {
    lock.acquire();

// Read and/or write shared state here.

    while (!testOnSharedState()) {
        cv.wait(&lock);
    }
    assert(testOnSharedState());

// Read and/or write shared state here.

    lock.release();
}

SharedObject::someMethodThatSignals() {
    lock.acquire();

// Read and/or write shared state here.

// If state has changed in a way that
// could allow another thread to make
// progress, signal (or broadcast).

    cv.signal();

    lock.release();
}
```

Figure 5.7: Design patterns for waiting using a condition variable (top) and for waking up a waiter (bottom). Since many critical sections need to both `wait` and `signal`, these two design patterns are often combined in one method.

The standard design pattern for a shared object is a lock and zero or more condition variables. A method that waits using a condition variable works as shown on the top in Figure 5.7. In this code, the calling thread first acquires the lock and can then read and write the shared object's state variables. To wait until testOnSharedState succeeds, the thread calls `wait` on the shared object's condition variable cv. This atomically puts the thread on the waiting list and releases the lock, allowing other threads to enter the critical section. Once the waiting thread is signaled, it re-acquires the lock and returns from `wait`. The monitor can then safely test the state variables to see if testOnSharedState succeeds. If so, the monitor performs its tasks, releases the lock, and returns.

The bottom of Figure 5.7 shows the complementary code that causes a waiting thread to wake up. Whenever a thread changes the shared object's state in a way that enables a waiting thread to make progress, the thread must signal the waiting thread using the condition variable.

A thread waiting on a condition variable must inspect the object's state in a loop. The condition variable's `wait` method releases the lock (to let other threads change the state of interest) and then re-acquires the lock (to check that state again).

Similarly, the only reason for a thread to `signal` (or `broadcast`) is that it has just changed the shared state in a way that may be of interest to a waiting thread. To make a change to shared state, the thread must hold the lock on the state variables, so `signal` and `broadcast` are also always called while holding a lock.

Discussion. Condition variables have been carefully designed to work in tandem with locks and shared state. The precise definition of condition variables includes three properties worth additional comment:

- A condition variable is *memoryless.*

  The condition variable, itself, has no internal state other than a queue of waiting threads. Condition variables do not need their own state because they are always used inside shared objects that have their own state.

  If no threads are currently on the condition variable's waiting list, a `signal` or `broadcast` has no effect. No thread calls `wait` unless it holds the lock, checks the state variables, and finds that it needs to wait. Thus, the condition variable has no "memory" of earlier calls to `signal` or `broadcast`. After `signal` is called, if sometime later another thread calls `wait`, it will block until the *next* `signal` (or `broadcast`) is called, regardless of how many times `signal` has been called in the past.

- *CV::wait atomically releases the lock.*

  A thread always calls `wait` while holding a lock. The call to `wait` *atomically* releases the lock and puts the thread on the condition variable's waiting list. Atomicity

ensures that there is no separation between checking the shared object's state, deciding to wait, adding the waiting thread to the condition variable's queue, and releasing the lock so that some other thread can access the shared object.

If threads released the lock before calling `wait`, they could miss a signal or broadcast and wait forever. Consider the case where thread T_1 checks an object's state and decides to wait, so it releases the lock in anticipation of putting itself on the condition variable's waiting list. At that precise moment, T_2 preempts T_1. T_2 acquires the lock, changes the object's state to what T_1 wants, and calls `signal`, but the waiting list is empty so the call to `signal` has no effect. Finally, T_1 runs again, puts itself on the waiting list, and suspends execution. The lack of atomicity means that T_1 missed the signal and is now waiting, potentially forever.

Once `wait` releases the lock, any number of threads might run before `wait` re-acquires the lock after a `signal`. In the meantime, the state variables might have changed — in fact, they are almost *certain* to have changed. Code must not assume just because something was true before `wait` was called, it remains true when `wait` returns. The only assumption you should make on return from `wait` is that the lock is held, and the normal invariants that hold at the start of the critical section are true.

- When a waiting thread is re-enabled via `signal` or `broadcast`, *it may not run immediately*.

  When a waiting thread is re-enabled, it is moved to the scheduler's ready queue with no special priority, and the scheduler may run it at some later time. Furthermore, when the thread finally does run, it must re-acquire the lock, which means that other threads may have acquired and released the lock in the meantime, between when the signal occurs and when the waiter re-acquires the lock. Therefore, even if the desired predicate were true when `signal` or `broadcast` was called, it may no longer be true when `wait` returns.

  This may seem like a small window of vulnerability, but concurrent programs must work with all possible schedules. Otherwise, programs may fail sometimes, but not always, making debugging very difficult. See the sidebar on Mesa vs. Hoare semantics for a discussion of the history behind this property.

WARNING: The points above have an important implication for programmers: `wait` *must always be called from within a loop*.

Because `wait` releases the lock, and because there is no guarantee of atomicity between `signal` or `broadcast` and the return of a call to `wait`, there is no guarantee that the checked-for state still holds. Therefore, a waiting thread must always wait in a loop, rechecking the state until the desired predicate holds. Thus, the design pattern is:

```
...
while (predicateOnStateVariables(...)) {
  wait(&lock);
}
...
```

and not:

```
...
if (predicateOnStateVariables(...)) {
    wait(&lock);
}
...
```

There are two fundamental reasons why condition variables impose this requirement: to simplify the implementation and to improve modularity.

- **Simplifying the implementation.** When a waiting thread is re-enabled, it may not run immediately. Other threads may access the shared state before it runs, and the desired predicate on the shared state may no longer hold when `wait` finally does return.

  This behavior simplifies the implementation of condition variables without increasing the complexity of the code that uses them. No special code is needed for scheduing; `signal` puts the signaled thread onto the ready list and lets the scheduler choose when to run it. Similarly, no special code is needed to re-acquire the lock at the end of `wait`. The woken thread calls `acquire` when it is re-scheduled. As with any attempt to acquire a lock, it may succeed immediately, or it may wait if some other thread acquired the lock first.

  Some implementations go even further and warn that a call to `wait` may return even if no thread has called `signal` or `broadcast`. So, not only is it possible that the desired predicate on the state *is no longer true*, it is possible that the desired predicate on the state *was never true*. For example, the Java definition of condition variables allows for "spurious wakeups":

  > When waiting upon a Condition, a "spurious wakeup" is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for. An implementation is free to remove the possibility of spurious wakeups but it is recommended that applications programmers always assume that they can occur and so always wait in a loop. (From https://docs.oracle.com/javase/8/docs/api/)

- **Improving modularity.** Waiting in a loop that checks the shared state makes shared objects' code more modular because we can reason about when the thread will continue by looking only at the `wait` loop. In particular, we do not need to examine the rest of the shared object's code to understand where and why calls to `signal` and `broadcast` are made to know the post-condition for the `wait` loop. For example, in

Figure 5.7, we know the assert call will never fail without having to look at any other code.

Not only does waiting in a loop simplify writing and reasoning about the code that waits, it simplifies writing and reasoning about the code that signals or broadcasts. Signaling at the wrong time will never cause a waiting thread to proceed when it should not. Signal and `broadcast` can be regarded as *hints* that it *might* be a good time to proceed; if the hints prove to be wrong, no damage is done. You can always convert a `signal` to a `broadcast`, or add any number of `signal` or `broadcast` calls, without changing the semantics of a shared object. Avoiding extra `signal` and `broadcast` calls may matter for performance, but not for correctness.

Bottom line: Given the range of possible implementations and the modularity benefits, `wait` must always be done from within a loop that tests the desired predicate.

Mesa vs. Hoare semantics

In modern condition variables, `signal` or `broadcast` calls take waiting threads from a condition variable's waiting list and put them on the ready list. Later, when these threads are scheduled, they may block for some time while they try to re-acquire the lock. Thus, modern condition variables implement what are often called *Mesa Semantics* (for Mesa, an early programming language at Xerox PARC that implemented these semantics). Despite the name, Mesa was not the first system to use "Mesa" semantics; Brinch Hansen had proposed their use five years earlier. However, PARC was the first to use Mesa semantics extensively in a very large operating system, and the name stuck.

C.A.R. "Tony" Hoare proposed a different definition for condition variables. Under *Hoare semantics*, when a thread calls `signal`, execution of the signaling thread is suspended, the ownership of the lock is immediately transferred to one of the waiting threads, and execution of that thread is immediately resumed. Later, when the resumed thread releases the lock, ownership of the lock reverts to the signaling thread, whose execution continues.

Under Hoare semantics, signaling is atomic with the resumption of a waiting thread, and a signaled thread may assume that the state has not changed since the signal that woke it up was issued. Under Mesa semantics, waiting is always done in a loop: while (predicate()) {cv.wait(&lock);}. Under Hoare semantics, waiting can be done with a simple conditional: if (predicate()) {cv.wait(&lock);}.

Mesa semantics are much more widely used, but some argue that the atomicity of signaling and resuming a waiting process makes it easier to prove liveness properties of programs under Hoare semantics. If we know that one thread is waiting on a condition, and we do a signal, we know that the waiting thread (and not some other late-arriving thread) will resume and make progress.

The authors of this book come down strongly on the side of Mesa semantics. The modularity advantages of Mesa greatly simplify reasoning about an object's core safety properties. For the properties we care most about (i.e., the safety properties that threads proceed only when they are supposed to) and for large programs where modularity matters, Mesa semantics seem vastly preferable. Later in this chapter, we will explain how to implement FIFO queueing with Mesa semantics, for where liveness concerns are paramount.

As a practical matter the debate has been settled: essentially all systems, including both Java and POSIX, use Mesa semantics. We know of no widely used system that implements Hoare semantics. Programmers that assume the weaker Mesa semantics — always writing while (predicate()) — will write programs that work under either definition. The overhead of the "extra" check of the predicate upon return from wait in a while loop is unlikely to be significant compared to the signaling and scheduling overheads. As a programmer, you will not go wrong if you write your code assuming Mesa semantics.

5.4.2 Thread Life Cycle Revisited

Chapter 4 discussed how a thread can switch between the READY, WAITING, and RUNNING states. We now explain the WAITING state in more detail.

A RUNNING thread that calls `wait` is put in the WAITING state. This is typically implemented by moving the thread control block (TCB) from the ready list to the condition variable's list of waiting threads. Later, when some RUNNING thread calls `signal` or `broadcast` on that condition variable, one (if `signal`) or all (if `broadcast`) of the TCBs on that condition variable's waiting list are moved to the ready list. This changes those threads from the WAITING state to the READY state. At some later time, the scheduler selects a READY thread and runs it by moving it to the RUNNING state. Eventually, the signaled thread runs.

Locks are similar. A lock `acquire` on a busy lock puts the caller into the WAITING state, with the caller's TCB on a list of waiting TCBs associated with the lock. Later, when the lock owner calls `release`, one waiting TCB is moved to the ready list, and that thread transitions to the READY state.

Notice that threads that are RUNNING or READY have their state located at a pre-defined, "global" location: the CPU (for a RUNNING thread) or the scheduler's list of ready threads (for a READY thread). However, threads that are WAITING typically have their state located on some per-lock or per-condition-variable queue of waiting threads. Then, a `signal`, `broadcast`, or `release` call can easily find and re-enable a waiting thread for that particular condition variable or lock.

```
// Thread-safe blocking queue.

const int MAX = 10;

class BBQ{
  // Synchronization variables
    Lock lock;
    CV itemAdded;
    CV itemRemoved;
```

```
  // State variables
    int items[MAX];
    int front;
    int nextEmpty;

  public:
    BBQ();
    ~BBQ() {};
    void insert(int item);
    int remove();
};

// Initialize the queue to empty,
// the lock to free, and the
// condition variables to empty.
BBQ::BBQ() {
    front = nextEmpty = 0;
}

// Wait until there is room and
// then insert an item.
void
BBQ::insert(int item) {
    lock.acquire();
    while ((nextEmpty - front) == MAX) {
        itemRemoved.wait(&lock);
    }
    items[nextEmpty % MAX] = item;
    nextEmpty++;
    itemAdded.signal();
    lock.release();
}

// Wait until there is an item and
// then remove an item.
int
BBQ::remove() {
    int item;

    lock.acquire();
    while (front == nextEmpty) {
        itemAdded.wait(&lock);
    }
    item = items[front % MAX];
    front++;
    itemRemoved.signal();
    lock.release();
    return item;
}
```

Figure 5.8: A thread-safe blocking bounded queue using Mesa-style condition variables.

5.4.3 Case Study: Blocking Bounded Queue

We can use condition variables to implement a *blocking bounded queue*, one where a thread trying to remove an item from an empty queue will wait until an item is available, and a thread trying to put an item into a full queue will wait until there is room. Figure 5.8 defines the blocking bounded queue's interface and implementation.

As in TSQueue, we acquire and release the lock at the beginning and end of the public methods (e.g., insert and remove). Now, however, we can atomically release the lock and wait if there is no room in insert or no item in remove. Before returning, insert signals on itemAdded since a thread waiting in remove may now be able to proceed; similarly, remove signals on itemRemoved before it returns.

We `signal` rather than `broadcast` because each insert allows at most one remove to proceed, and vice versa.

EXAMPLE: What invariants hold when `wait` returns in BBQ:remove? Is an item guaranteed to be in the queue? Why or why not?

ANSWER: Exactly the same invariants hold when `wait` returns as when the thread first acquired the lock. These are the same constraints as listed earlier for the thread-safe (non-blocking) bounded queue TSQueue.

In particular, although there is always an item in the queue when insert calls `signal`, there is *no* guarantee that the item is still in the queue when `wait` returns. Even if the language runtime avoids spurious wakeups, some other thread may have run between the `signal` and the return from `wait`. That thread may perform a remove, acquire the BBQ::lock, find the item, and empty the queue, all before `wait` returns. □

5.5 Designing and Implementing Shared Objects

Although multi-threaded programming has a reputation for being difficult, shared objects provide a basis for writing simple, safe code for multi-threaded programs. In this section, we provide a methodology for writing correct multi-threaded code using shared objects.

- We first define a high-level approach to designing shared objects. Given a concurrent problem, where do you start? (Section 5.5.1)
- We provide six specific rules, or best practices, that you should always follow when writing multi-threaded shared objects. (Section 5.5.2)
- We describe three common pitfalls to multi-threading in C, C++, and Java code. (Section 5.5.3)

Our experience is that following this approach and these rules makes it much more likely that you will write code that is not only correct but also easy for others to read, understand, and maintain.

On simplicity

One of the themes running through this textbook is the importance of simple abstractions in building robust, reliable operating systems. Operating systems place a premium on

reliability; if the operating system breaks, the computer becomes temporarily unusable, or worse. And yet, it is nearly impossible to fully test whether some piece of multi-threaded operating system code works under all possible conditions and all possible schedule interleavings. This places a premium on designing solutions that work the first time they are run, by keeping code simple.

Particularly with concurrent code, it is not enough for the code to work. It also needs to be simple enough to understand. We often find students write intricate concurrent code in solutions to our homework assignments and exams. Perhaps the difficulty of the topic suggests to students that their solutions must also be difficult to understand! Sometimes these solutions work; more often the complexity hides a design flaw.

Even if your code is literally correct, we would like to encourage you to not stop there. Is it easy to understand *why* your code works? If not, try again. Even if you can get the code to work this time, someone else may need to come along later and change it. For concurrent code to be maintainable over time, it is essential that the next developer to work on the code be able to understand it.

Yet, often in technology circles, simplicity is considered an insult. Someone might say, "Anyone could have done that!", meaning it as a put down. We take the other side: a simple design should be seen as a complement. Complexity should be introduced only where it is absolutely necessary. Consider three possible states for one of your designs (hat tip to John Ousterhout for this list):

- The code is simple enough that anyone can understand it. If someone says this to you, the appropriate response is to take it as a complement and reply, "Thank you."
- The code is so complicated that only the author can understand it. While this might be useful in the short-term as a strategy to keep the author employed (after all, no one else can fix or improve code without understanding it first), it is not such a good idea over the long term. Eventually, you will want to work on something new!
- The code is so complicated not even the author can understand it. Concurrent code often lands in this category, unnecessarily in our view. Using the rules we introduce in this section will help put your code in the first and not the last bucket.

Of course, writing individual shared objects is not enough. Most programs have multiple shared objects, and new issues arise when combining them. But, before trying to compose multiple shared objects, we must make sure that each individual object works. Chapter 6 discusses the issues that arise when programs use multiple shared objects.

5.5.1 High Level Methodology

A shared object has public methods, private methods, state variables, and synchronization variables; its synchronization variables include a lock and one or more condition variables. At this level, shared object programming resembles standard object-oriented programming, except that we have added synchronization variables to each shared object. This similarity is deliberate: the interfaces to locks and condition variables have been carefully defined so that we can continue to apply familiar techniques for programming and reasoning about objects.

Therefore, most high-level design challenges for a shared object's class are the same as for class design in single-threaded programming:

- Decompose the problem into objects.
- For each object:
  - Define a clean interface.
  - Identify the right internal state and invariants to support that interface.
  - Implement methods with appropriate algorithms to manipulate that state.

These steps require creativity and sound engineering judgment and intuition. Going from single-threaded to multi-threaded programming does not make these steps much more difficult.

Compared to how you implement a class in a single-threaded program, the new steps needed for the multi-threaded case for shared objects are straightforward:

1. Add a lock.
2. Add code to `acquire` and `release` the lock.
3. Identify and add condition variables.
4. Add loops to wait using the condition variables.
5. Add `signal` and `broadcast` calls.

We discuss each of these steps in turn.

Other than these fairly mechanical changes, writing the rest of your code proceeds as in the single-threaded case.

1. **Add a lock.** Each shared object needs a lock as a member variable to enforce mutually exclusive access to the object's shared state.

   This chapter focuses on the simple case where each shared object includes exactly one lock. In Chapter 6, we will talk about more advanced variations, such as an ownership design pattern where higher-level program structure enforces mutual exclusion by ensuring that at most one thread at a time owns and can access an object.

2. **Add code to acquire and release the lock.** All code accessing the object's shared state — any state shared across more than one thread — must hold the object's lock. Typically, all of an object's member variables are shared state.

   The simplest and most common approach is to acquire the lock at the start of each public method and release it at the end of each public method. Doing so makes it easy to inspect your code to verify that a lock is always held when needed. It also means that the lock is already held when each private method is called, and you do not need

to re-acquire it.

WARNING: You may be tempted to try to avoid acquiring the lock in some methods or parts of some methods. Do not be tempted by this "optimization" until you are an experienced programmer and have done sufficient profiling of the code to verify that the optimization will significantly speed up your program, *and* you fully understand the hazards posed by compiler and architecture instruction re-ordering.

Acquiring an uncontended lock is a relatively inexpensive operation. By contrast, reasoning about memory interleavings can be quite difficult — and the instruction reordering done by modern compilers and processors makes it even harder. Later in this section, we discuss one commonly used (and abused) "optimization," double-checked locking, that is outright dangerous to use.

3. **Identify and add condition variables.** How do you decide what condition variables a shared object needs?

   A systematic way to approach this problem is to consider each method and ask, *"When can this method wait?"* Then, you can map each situation in which a method can wait to a condition variable.

   You have considerable freedom in deciding how many condition variables a class should have and what each should represent. A good option is to add a condition variable for each situation in which the method must wait.

   EXAMPLE: Blocking bounded queue with two condition variables. In our blocking bounded queue example, if the queue is full, `insert` must wait until another thread removes an item, so we created a condition variable itemRemoved. Similarly, if the queue is empty, `remove` must wait until another thread inserts an item, so we created a condition variable itemAdded. It is natural in this case to create two condition variables, itemAdded to wait until the queue has items, and itemRemoved to wait until the queue has space.

   Alternatively, a single condition variable can often suffice. In fact, early versions of Java defined a single condition variable per object and did not let programmers allocate additional ones. Using this approach, any thread that waits for any reason uses that condition variable; if the condition variable is used by different threads waiting for different reasons, then any thread that wakes up a thread must `broadcast` on the condition variable.

   EXAMPLE: Blocking bounded queue with one condition variable. It is also possible to implement the blocking bounded queue with a single condition variable, i.e., somethingChanged, on which threads in both `insert` or threads in `remove` can wait. With this approach, both `insert` and `remove` need to `broadcast` rather than `signal` to ensure that the right threads get a chance to run.

   Programs that are more complex make these trade-offs more interesting. For example, imagine a ResourceManager class that allows a calling thread to request exclusive access to any subset of n distinct resources. One could imagine creating 2^n condition variables; this would let a requesting thread wait on a condition variable representing exactly its desired combination. However, it would be simpler to have a single condition variable on which requesting threads wait and to broadcast on that condition whenever a resource is freed. Depending on the number of resources and the expected number of waiting threads, this simpler approach may even be more efficient.

   The bottom line is that there is no hard and fast rule for how many condition variables to use in a shared object. Selecting condition variables requires thought, and different designers may use different numbers of condition variables for a given class. Like many other design decisions, this is a matter of programmer taste, judgment, and experience. Asking "When can this method wait?" will help you identify what is for you a natural way of thinking about a shared object's condition variables.

4. **Add loops to wait using the condition variables.** Add a while(…) {cv.wait()} loop into each method that you identified as potentially needing to wait before returning.

   Remember that every call to `wait` must be enclosed in a while loop that tests an appropriate predicate. Modern implementations almost invariably provide Mesa semantics and often allow for spurious wakeups (i.e., a thread can return from `wait` even if no thread called `signal` or `broadcast`). Therefore, a thread must always check the condition before proceeding. Even if the condition was true when the `signal` or `broadcast` call occurred, it may no longer be true when the waiting thread resumes execution.

   Modularity benefits. If you always wait in a while loop, your code becomes highly modular. You can look at the code that waits, and when it proceeds, know *without* examining any other code that the condition holds. Even erroneous calls to `signal` or `broadcast` will not change how the waiting code behaves.

   For example, consider the assertion in the following code:

```
...
while (!workAvailable()) {
    cond.wait(&lock);
}
assert(workAvailable());
...
```

   We know that the assertion holds by local inspection *without knowing anything about the code that calls `signal` or `broadcast`*.

   Waiting in a while loop also makes the signal and broadcast code more robust. Adding an extra `signal`, or changing a `signal` to a `broadcast`, will not introduce bugs.

   HINT: **Top-down design.** As you start writing your code, you may know that a method needs to include a wait loop, but you may not know exactly what the predicate should be. In this situation, it is often useful to name a private method function that will perform the test (e.g., workAvailable in the preceding example) and

write the code that defines the function later.

5. **Add `signal` and `broadcast` calls.** Just as you must decide when methods can wait, you must decide when methods can let other waiting threads proceed. It is usually easy to ask, "Can a call to this method allow another thread to proceed?" and then add a `signal` or `broadcast` call if the answer is yes. But which call should you use?

   CV::signal is appropriate when: (1) at most one waiting thread can make progress, and (2) any thread waiting on the condition variable can make progress. In contrast, `broadcast` is needed when: (1) multiple waiting threads may all be able to make progress, or (2) different threads are using the same condition variable to wait for different predicates, so some of the waiting threads can make progress but others cannot.

   EXAMPLE: Consider the n-resource ResourceManager problem described earlier. For the solution with a single condition variable, we must `broadcast` on the condition variable whenever a resource is freed. We do not know which thread(s) can make progress, so we tell them all to check. If, instead, we used `signal`, then the "wrong" thread might receive the `signal`, and a thread that could make progress might remain blocked.

   It is always safe to use `broadcast`. Even in cases where `signal` would suffice, at worst, all of the waiting threads would run and check the condition in the while loop, but only one would continue out of the loop. Compared to `signal`, this would consume some additional resources, but it would not introduce any bugs.

5.5.2 Implementation Best Practices

Above, we described the basic thought process you should follow when designing a shared object. To make things more concrete, we next give a set of six simple rules that we strongly advocate you follow; these are a set of "best practices" for writing code for shared objects.

Coding standards, soapboxes, and preaching

Some programmers rebel against coding standards. We do not understand their logic. For concurrent programming in particular, a few good design patterns have stood the test of time (and many unhappy people who have departed from those patterns). For concurrent programming, *debugging does not work.* You must rely on: (a) writing correct code, and (b) writing code that you and others can read and understand — not just for now, but also over time as the code changes. Following the rules we provide will help you write correct, readable code.

When we teach multi-threaded programming, we treat the six rules described in this section as *required coding standards* for all multi-threaded code that students write in our course. We say, "We cannot control what you do when you leave this class, but while you are in this class, any solution that violates these standards is, by definition, *wrong*."

In fact, we feel so strongly about these rules that one of us actually presents them in class by standing on a table and pronouncing them as the Six Commandments of multi-threaded programming:

1. Thou shalt always do things the same way.

and so on.

The particular formulation (and presentation) of these rules evolved from our experience teaching multi-threaded programming dozens of times to hundreds of students and identifying common mistakes. We have found that when we insist that students follow these rules, the vast majority find it easy to write clear and correct code for shared objects. Conversely, in earlier versions of the course, when we phrased these items as "strong suggestions," many students found themselves adrift, unable to write code for even the simplest shared objects.

Our advice to those learning multi-threaded programming is to treat these rules as a given and follow them strictly for a semester or so, until writing shared objects is easy. At that point, you most likely will understand concurrent programming well enough to decide whether to continue to follow the rules.

We also believe that experienced programmers benefit from adhering closely to these rules. Since we began teaching them, we have also disciplined ourselves to follow them unless there is a very good reason not to. We have found exceptions to be rare. Conversely, when we catch ourselves being tempted to deviate from the rules, the vast majority of the time our code improves if we force ourselves to rewrite the code to follow the rules.

Although the rules may come across as opinionated (and they are), they are far from novel. Over three decades ago, Lampson and Redell's paper, "Experience with Processes and Monitors in Mesa," provided similar advice (in a more measured tone).

---

1. **Consistent structure.** The first rule is a meta-rule that underlies the other five rules: *follow a consistent structure.* Although programming with a clean, consistent structure is always useful, it is particularly important to strictly follow tried-and-true design patterns for shared objects.

   At a minimum, even if one way is not inherently better than another, following the same strategy every time: (1) frees you to focus on the core problem because the details of the standard approach become a habit, and (2) makes it easier for those who follow to review, maintain, and debug your code. (And it will make it easier for *you* to maintain and debug your code.)

   As an analogy, electricians follow standards for the colors of wire they use for different tasks. White is neutral. Black or red is hot. Copper is ground. An electrician does not have to decide "Hm. I have a bit more white on my belt today than black, should I use white or black for my grounds?" When an electrician walks into a room she wired last month, she does not have to spend time trying to remember which color is which. If an electrician walks into a room she has never seen before, she can immediately determine what the wiring is doing, without having to trace it back into the switchboard. Similar advantages apply to coding standards.

However, for concurrent programs, the evidence is that the abstractions we describe *are* better than almost all others. Until you become a *very* experienced concurrent programmer, take advantage of the hard-won experience of those that have come before you. Once you are a concurrency guru, you are welcome to invent a better mousetrap.

Sure, you can cut corners and occasionally save a line or two of typing by departing from the standards. However, you will have to spend a few minutes thinking to convince yourself that you are right on a case-by-case basis (and another few minutes typing comments to convince the next person to look at the code that you are right), and a few hours or weeks tracking down bugs when you are wrong. It is just not worth it.

2. **Always synchronize with locks and condition variables.**

   Many operating systems, such as Linux, Windows, and MacOS, provide a diversity of synchronization primitives. At the end of this chapter, we will describe one such primitive, semaphores, which is particularly widely used in operating system kernel implementations. Compared to locks and condition variables, semaphores are equally powerful: you can build condition variables using semaphores and vice versa. If so, why pick one over the other?

   We recommend that you be able to read and understand semaphores so you can understand legacy code, but that you only write new code using locks and condition variables. Almost always, code using locks and condition variables is clearer than the equivalent code using semaphores because it is more "self-documenting." If the code is well structured, what each synchronization action is doing should be obvious. Admittedly, semaphores sometimes seem to fit what you are doing perfectly because you can map the object's invariants exactly onto the internal state of the semaphore; for example, you can write an extremely concise version of our blocking bounded queue using semaphores. But what happens when the code changes next month? Will the fit remain as good? For consistency and simplicity, choose one of the two styles and stick with it. In our opinion, the right one is to use locks and condition variables.

3. **Always acquire the lock at the beginning of a method and release it right before the return.**

   This extends the principle of consistent structure: pick one way to do things and always follow it. The benefit here is that it is easy to read code and see where the lock is or is not held because synchronization is structured on a method-by-method basis. Conversely, if `acquire` and `release` calls are buried in the middle of a method, it is harder to quickly inspect and understand the code.

   Taking a step back, if there is a logical chunk of code that you can identify as a set of actions that require a lock, then that section should probably be its own procedure: it is a set of logically related actions. If you find yourself wanting to acquire a lock in the middle of a procedure, that is usually a red flag that you should break the piece you are considering into a separate procedure. We are all sometimes lazy about creating new procedures when we should. Take advantage of this signal, and the result will be clearer code.

   There are two corollaries to this rule. First, if your code is well structured, all shared data will be encapsulated in an object, and therefore all accesses to shared data will be protected by a lock. Since compilers and processors *never* re-order instructions across lock operations, this rule guarantees instruction re-ordering is not a concern for your code.

   Second, from time to time, we see students attempting to acquire a lock in one procedure, and release it in another procedure, or worse, in a completely different thread. (One popular idea is to acquire a lock in a parent thread, pass it in thread_fork to a child, and have the child release the lock after it has started.) *Do not do this.* For one, it can make it very difficult for someone reading your code to determine which shared variables are protected by which lock; by acquiring at the beginning of the procedure and releasing at the end, which variables go with which locks is obvious.

   While some early thread systems allowed lock passing, most recently designed systems prohibit it. For example, in POSIX, lock release is "undefined" when called by a different thread than the thread that acquired the lock. In other words, it might work on some systems, but it is not portable. In Java, it is completely prohibited.

4. **Always hold the lock when operating on a condition variable.**

   The reason you signal on a condition variable — after manipulating shared state — is that another thread is waiting in a loop for some test on shared state to become true. Condition variables are useless without shared state, and shared state should only be accessed while holding a lock.

   Many libraries enforce this rule — that you cannot call condition variable methods unless you hold the corresponding lock. However, some run-time systems and libraries allow sloppiness, so take care.

5. **Always wait in a while() loop**

   The pattern should always be:

```
while (predicateOnStateVariables(...)) {
    condition->wait(&lock);
}
```

   and never:

```
...
if (predicateOnStateVariables(...)) {
    wait(&lock);
}
...
```

   Here, predicateOnStateVariables(…) is code that looks at the state variables of the

current object to decide if the thread should proceed.

You may be tempted to guard a `wait` call with an if conditional rather than a while loop when you can deduce from the global structure of the program that, despite Mesa semantics, any time a thread returns from wait, it can proceed. Avoid this temptation.

While works any time if does, and it works in situations when if does not. By the principle of consistent structure, do things the same way every time. But there are three additional issues.

- Using if breaks modularity. In the preceding example, to know whether using if will work, you must consider the global structure of the program: what threads there are, where `signal` is called, etc. The problem is that a change in code in one method (say, adding a `signal`) can then cause a bug in another method (where the `wait` is). Using while is self-documenting; anyone can look at the `wait` and see exactly when a thread may proceed.
- Always using while gives you incredible freedom about where to put a `signal`. In fact, `signal` becomes a hint — you can add a signal to an arbitrary place in a correct program and have it remain correct.
- Using if breaks portability. Some implementations of condition variables allow spurious wakeups, while others do not. For example, implementations of condition variables in both Java and the POSIX pthreads library are allowed to return from `wait` even though no thread called `signal` or `broadcast`.

6. **(Almost) never use `thread_sleep`.**

   Many thread libraries have a `thread_sleep` function that suspends execution of the calling thread for some period of wall clock time. Once that time passes, the thread is returned to the scheduler's ready queue and can run again.

   Never use `thread_sleep` to have one thread wait for another thread to perform a task. The correct way to wait for a condition to become true is to `wait` on a condition variable.

   In general, `thread_sleep` is appropriate only when there is a particular real-time moment when you want to perform some action, such as a timeout for when to declare a remote server non-responsive. If you catch yourself writing while(testOnObjectState()) {thread_sleep();}, treat this as a red flag that you are probably making a mistake.

   Similarly, if a thread must wait for an object's state to change, it should `wait` on a condition variable, and not just call thread_yield. Use thread_yield only when a low-priority thread *that can still make progress* wants to let a higher-priority thread to run.

5.5.3 Three Pitfalls

We next describe three common pitfalls. The first, double-checked locking, is a problem in many different programming languages, including C, C++ and Java. The second and third pitfalls are specific to Java. Java is a modern type-safe language that included support for threads from its inception. This built-in support makes multi-threaded programming in Java convenient. However, some aspects of the language are *too* flexible and can encourage bad practices. We highlight those pitfalls here.

1. **Double-Checked Locking.**

   We strongly advise holding a shared object's lock across any method that accesses the object's member variables. Programmers are often tempted to avoid some of these lock acquire and release operations. Unfortunately, such efforts often result in code that is complex, wrong, or both.

   To illustrate the challenges, consider the *double-checked locking* design pattern. The canonical example is an object that is allocated and initialized lazily the first time it is needed by any thread. (This example and analysis is taken from Meyers and Alexandrescu, "C++ and the Perils of Double-Checked Locking." http://www.aristeia.com/Papers/DDJ_Jul_ Aug_2004_revised.pdf) Being good programmers, we can hide the lazy allocation inside an object, Singleton, which returns a pointer to the object, creating it if needed.

   The "optimization" is to acquire the lock if the object has not already been allocated, but to avoid acquiring the lock if the object already exists. Because there can be a race condition between the first check and acquiring the lock, the check must be made again inside the lock.

```
class Singleton {
  public:
    static Singleton* instance();
    Lock lock;
    ...
  private:
    static Singleton* pInstance;
};

Singleton* Singleton::pInstance = NULL;


// BUG!  DON'T DO THIS!
Singleton*
Singleton::instance() {
   if (pInstance == NULL) {
       lock.acquire();
       if (pInstance == NULL){
           pInstance = new Instance();
       }
       lock.release();
   }
   return pInstance;
}
```

Although the intuition is appealing, **this code does not work.** The problem is that the statement pInstance = new Instance() is not an atomic operation; in fact, it comprises at least three steps:

1. Allocate memory for a Singleton object.
2. Initialize the Singleton object's memory by running the constructor.
3. Make pInstance point to this newly constructed object.

The problem is that modern compilers and hardware architectures can reorder these events. Thus, it is possible for thread 1 to execute the first step and then the third step; then thread 2 can call instance, see that pInstance is non-null, return it, and begin using this object before thread 1 finishes initializing it.

Discussion. This is just an example of dangers that lurk when you try to elide locks; the lesson applies more broadly. This example is extremely simple — fewer than 10 lines of code with very simple logic — yet a number of published solutions have been wrong. As Meyers and Alexandrescu note, some tempting solutions using temporary variables and the volatile keyword do not work. Bacon et al.'s "The 'Double-Checked Locking is Broken' Declaration" discusses a range of non-solutions in Java.
http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

This type of optimization is risky and often does not provide significant performance gains in practice. Most programmers should not consider them. Even expert programmers should habitually stick to simpler programming patterns, like the ones we have discussed, and only consider optimizations like double-checked locking when performance measurements and profiling indicate that the optimizations would significantly improve overall performance.

2. **Avoid defining a synchronized block in the middle of a method.**

   Java provides built in language support for shared objects. The base Object class, from which all classes inherit, includes a lock and a condition variable as members. Any method declaration can include the keyword synchronized to indicate that the object's lock is to be automatically acquired on entry to the method and automatically released on any return from the method. For example:

```
public synchronized foo() {
    // Do something; lock is automatically acquired/released.
}
```

   This syntax is useful — it follows rule #2 above, and it frees the programmer from having to worry about details, like making sure the lock is released before every possible return point including exceptions. The pitfall is that Java also allows a *synchronized block* in the middle of a method. For example:

```
public bar() {
    // Do something without holding the lock
    synchronized{
        // Do something while holding the lock
    }
    // Do something without holding the lock
}
```

   This construct violates rule #3 from Section 5.5.2 and suffers from the disadvantages listed there. The solution is the same as discussed above: when you find yourself tempted to write a synchronized block in the middle of a Java method, treat that as a strong hint that you should define a separate method to more clearly encapsulate the logical chunk you have identified.

3. **Keep shared state classes separate from thread classes.**

   Java defines a class called Thread that implements an interface called Runnable that other classes can implement in order to be treated as threads by the runtime system. To write the code that represents a thread's "main loop," you typically extend the Thread class or implement a class that implements Runnable.

   The pitfall is that, when extending the Thread class (or writing a new class that implements Runnable), you may be tempted to include not only the thread's main loop but also state to be shared across multiple threads, blurring the lines between the threads and the shared objects. This is almost always confusing.

   For example, for a blocking bounded queue, rather than defining two classes, BBQ for the shared queue and WorkerThread for the threads, you may be tempted to combine the two into a single class — for example, a queue with an associated worker thread. If this sounds confusing, it is, but it is a pitfall that we frequently see in student code.

   The solution is simple. Always make sure threads and shared objects are defined in separate classes. State that can be accessed by multiple threads, locks, and condition variables should never appear in any Java class that extends Thread or implements Runnable.

5.6 Three Case Studies

The best way to learn how to program concurrently is to practice. Multithreaded programming is an important skill, and we anticipate that almost everyone reading this book will over time need to write many multi-threaded programs. To help get you started, this section walks through several examples.

5.6.1 Readers/Writers Lock

First, we implement a *readers/writers lock*. Like a normal mutual exclusion lock, a readers/writers lock (RWLock) protects shared data. However, it makes the following optimization. To maximize performance, an RWLock allows multiple "reader" threads to

simultaneously access the shared data. Any number of threads can safely read shared data at the same time, as long as no thread is modifying the data. However, only one "writer" thread may hold the RWLock at any one time. (While a "reader" thread is restricted to only read access, a "writer" thread may read *and* write the data structure.) When a writer thread holds the RWLock, it may safely modify the data, as the lock guarantees that no other thread (whether reader or writer) may simultaneously hold the lock. The mutual exclusion is thus between any writer and any other writer, and between any writer and *the set* of readers.

Optimizing for the common case

Reader/writer locks are an example of an important principle in the design of computer systems: optimizing for the common case. Performance optimizations often have the side effect of making the code more complex to understand and reason about. Code that is more complex is more likely to be buggy, and more likely to have new bugs introduced as features are added. How do we decide when an optimization is worth the cost?

One approach is to profile your code. Then, *and only then*, optimize the code paths that are frequently used.

In the case of locks, it is obviously simpler to use a regular mutual exclusion lock. Replacing a mutual exclusion lock with a reader-writer lock is appropriate when both of the following are true: (i) there is substantial contention for the mutual exclusion lock and (ii) a substantial majority of the accesses are read-only. In other words, it is only appropriate to use if it would make a significant difference.

Reader-writer locks are very commonly used in databases, where they are used to support faster search queries over the database, while also supporting less frequent database updates. Another common use is inside the operating system kernel, where core data structures are often read by many threads and only infrequently updated.

To generalize our mutual exclusion lock into a readers/writers lock, we implement a new kind of shared object, RWLock, to guard access to the shared data and to enforce these rules. The RWLock is implemented using our standard synchronization building blocks: mutual exclusion locks and condition variables.

A thread that wants to (atomically) read the shared data proceeds as follows:

```
rwLock->startRead();
// Read shared data
rwLock->doneRead();
```

Similarly, a thread that wants to (atomically) write the shared data does the following:

```
rwLock->startWrite();
// Read and write shared data
rwLock->doneWrite();
```

To design the RWLock class, we begin by defining its interface (already done in this case) and its shared state. For the state, it is useful to keep enough data to allow a precise characterization of the object; especially when debugging, having too much state is better than having too little. Here, the object's behavior is fully characterized by the number of threads reading or writing and the number of threads waiting to read or write, so we have chosen to keep four integers to track these values. Figure 5.9 shows the members of and interface to the RWLock class.

```
class RWLock{
  private:
    // Synchronization variables
    Lock lock;
    CV readGo;
    CV writeGo;

    // State variables
    int activeReaders;
    int activeWriters;
    int waitingReaders;
    int waitingWriters;

  public:
    RWLock();
    ~RWLock() {};
    void startRead();
    void doneRead();
    void startWrite();
    void doneWrite();

  private:
    bool readShouldWait();
    bool writeShouldWait();
};
```

Figure 5.9: The interface and member variables for our readers/writers lock.

Next, we add synchronization variables by asking, "When can methods wait?" First, we add a mutual exclusion lock: the RWLock methods must wait whenever another thread is accessing the RWLock state variables. Next, we observe that startRead or startWrite may have to wait, so we add a condition variable for each case: readGo and writeGo.

RWLock::doneRead and doneWrite do not wait (other than to acquire the mutual exclusion lock). Therefore, these methods do not need any additional condition variables.

We can now implement RWLock. Figure 5.10 shows the complete solution, which we develop in a few simple steps. Much of what we need to do is almost automatic.

- Since we always acquire/release mutual exclusion locks at the beginning/end of a method (and never in the middle), we can write calls to acquire and release the

mutual exclusion lock at the start and end of each public method before even thinking in detail about what these methods do.

At this point, startRead and doneRead look like this:

```
void RWLock::startRead() {
    lock.acquire();


    lock.release();
}

void RWLock::doneRead() {
    lock.acquire();


    lock.release();
}
```

RWLock::startWrite and RWLock::doneWrite are similar.

- Since we know startRead and startWrite may have to wait, we can write a while(…) {wait(…);} loop in the middle of each. In fact, we can defer thinking about the details by inserting a private method to be defined later, as the predicate for the while loop (e.g., readShouldWait and writeShouldWait).

  At this point, startRead looks like this:

```
void RWLock::startRead() {
    lock.acquire();

    while (readShouldWait()) {
        readGo.Wait(&lock);
    }

    lock.release();
}
```

RWLock::StartWrite() looks similar.

Now things get a bit more complex. We can add code to track activeReaders, activeWriters, waitingReaders, and waitingWriters. Since we hold mutual exclusion locks in all of the public methods, this is easy to do. For example, a call to startRead initially increments the number of waiting readers; when the thread gets past the while loop, the number of waiting readers is decremented, but the number of active readers is incremented.

When reads or writes finish, it may become possible for waiting threads to proceed. We therefore need to add `signal` or `broadcast` calls to doneRead and doneWrite. The simplest solution would be to `broadcast` on both readGo and writeGo in each method, but that would be both inefficient and (to our taste) less clear about how the class actually works.

Instead, we observe that in doneRead, when a read completes, there are two interesting cases: (a) no writes are pending, and nothing needs to be done since this read cannot prevent other reads from proceeding, or (b) a write is pending, and this is the last active read, so one write can proceed. In case (b), we use `signal` since at most one write can proceed, and any write waiting on the condition variable can proceed.

Our code for startRead and doneRead is now done:

```
// Wait until no active or waiting
// writes, then proceed.
void RWLock::startRead() {
    lock.acquire();
    waitingReaders++;
    while (readShouldWait()) {
        readGo.Wait(&lock);
    }
    waitingReaders--;
    activeReaders++;
    lock.release();
}

// Done reading. If no other active
// reads, a write may proceed.
void RWLock::doneRead() {
    lock.acquire();
    activeReaders--;
    if (activeReaders == 0
         && waitingWriters > 0) {
        writeGo.signal();
    }
    lock.release();
}
```

Code for startWrite and doneWrite is similar. For doneWrite, if there are any pending writes, we signal on writeGo. Otherwise, we broadcast on readGo.

Finally, we need to define the readShouldWait and writeShouldWait predicates. Here, we implement a *writers preferred* solution: reads should wait if there are any active or pending writers, while writes wait only while there are active readers or active writers. Otherwise, a continuous stream of new readers could starve a write request and prevent if from ever being serviced.

```
bool
RWLock::readShouldWait() {
  return (activeWriters > 0 || waitingWriters > 0);
```

```
    }
```

The code for writeShouldWait is similar.

Since readShouldWait and writeShouldWait are private methods that are always called from public methods that hold the mutual exclusion lock, they do not need to acquire the lock.

Figure 5.10 gives the full code. This solution may not be to your taste. You may decide to use more or fewer condition variables, use different state variables to implement different invariants, or change when to call `signal` or `broadcast`. The shared object approach allows designers freedom in these dimensions.

```
// Wait until no active or waiting
// writes, then proceed.
void RWLock::startRead() {
    lock.acquire();
    waitingReaders++;
    while (readShouldWait()) {
        readGo.Wait(&lock);
    }
    waitingReaders--;
    activeReaders++;
    lock.release();
}

// Done reading. If no other active
// reads, a write may proceed.
void RWLock::doneRead() {
    lock.acquire();
    activeReaders--;
    if (activeReaders == 0
         && waitingWriters > 0) {
        writeGo.signal();
    }
    lock.release();
}

// Read waits if any active or waiting
// write ("writers preferred").
bool
RWLock::readShouldWait() {
    return (activeWriters > 0
             || waitingWriters > 0);
}


// Wait until no active read or
// write then proceed.
void RWLock::startWrite() {
    lock.acquire();
    waitingWriters++;
    while (writeShouldWait()) {
        writeGo.Wait(&lock);
    }
    waitingWriters--;
    activeWriters++;
    lock.release();
}

// Done writing. A waiting write or
// read may proceed.
void
RWLock::doneWrite() {
    lock.acquire();
    activeWriters--;
    assert(activeWriters == 0);
    if (waitingWriters > 0) {
        writeGo.signal();
    }
    else {
        readGo.broadcast();
    }
    lock.release();
}

// Write waits for active read or write.
bool
RWLock::writeShouldWait() {
    return (activeWriters > 0
             || activeReaders > 0);
}
```

Figure 5.10: An implementation of a readers/writers lock.

Single stepping and model checking your code

Suppose you have written some concurrent code, and you would like to verify that the solution behaves as you expect. One thing you should *always* do — whether for sequential or concurrent code — is to use a debugger to single step through the code on various inputs, to verify that the program logic is doing what you expect it to do, and do the variables have the values you expect.

This is especially useful for concurrent programs. Since the program must work for any possible thread schedule, you can use the debugger to consider what happens when threads are interleaved in different ways. Does your program logic still do what you expect?

For example, for the RWLock class, you can:

- Start a single reader. Does it go all the way through? Obviously, it should not wait, since no one has the lock and there are no writers. When it finishes readDone, are the state variables back to their initial state?
- Start a writer, and after it acquires the mutual exclusion lock, start a reader. Does it wait for the lock? When the writer finishes startWrite, does the reader proceed and then wait for the writer to call doneWrite? Does the reader proceed after that?

- Start a reader, followed by a writer, followed by another reader. And so forth.

We encourage you to do this for the examples in this section. The examples are short enough that you can execute them by hand, but we also provide code if you want to try this in a debugger.

A more systematic approach is called model checking. To fully verify that a concurrent program does what it was designed to do, a model checker enumerates all possible sequences of operations, and tries each one in turn. Since this could result in a nearly infinite number of possible tests even for a fairly simple program, to be practical model checking needs to reduce the search space. For code that follows our guidelines — with locks to protect shared data — the exact ordering of instructions is no longer important. For example, preempting a thread that holds a lock is immaterial to the behavior of the program.

Rather, the behavior of the program depends on the sequence of synchronization instructions: which thread is first to acquire the lock, which thread waits on a condition variable, and so forth. Thus, a model checker can proceed in two steps: first verify that there are no unlocked accesses to shared data, and then enumerate various sequences of synchronization operations. Even with this, the number of possibilities can be prohibitively large, and so typically the model checker will verify however many different interleavings it can within some time limit.

5.6.2 Synchronization Barriers

With data parallel programming, as we explained in Chapter 4, the computation executes in parallel across a data set, with each thread operating on a different partition of the data. Once all threads have completed their work, they can safely use each other's results in the next (data parallel) step in the algorithm. MapReduce is an example of data parallel programming, but there are many other systems with the same structure.

For this to work, we need an efficient way to check whether all n threads have finished their work. This is called a *synchronization barrier*. It has one operation, `checkin`. A thread calls `checkin` when it has completed its work; no thread may return from `checkin` until *all* n threads have checked in. Once all threads have checked in, it is safe to use the results of the previous step.

Note that a synchronization barrier is different from a memory barrier, defined earlier in the chapter. A synchronization barrier is called concurrently by many threads; the barrier prevents any thread from proceeding until all threads reach the barrier. A memory barrier is called by one thread, to prevent the thread from proceeding until all memory operations that occur before the barrier have completed and are visible to other threads.

An implementation of MapReduce using a synchronization barrier might look like the code in Figure 5.11.

```
Create n threads.
Create barrier.

Each thread executes map operation in parallel.
barrier.checkin();

Each thread sends data in parallel to reducers.
barrier.checkin();

Each thread executes reduce operation in parallel.
barrier.checkin();
```

Figure 5.11: An implementation of MapReduce using synchronization barriers.

An alternative to using a synchronization barrier would be to create n threads at each step; the main thread could then call thread_join on each thread to ensure its completion. While this would be correct, it might be inefficient. Not only would n new threads need to be started at each step, the partitioning of work among threads would also need to be redone each time. Frequently, each thread in a data parallel computation can work on the same data repeatedly over many steps, maximizing the efficiency of the hardware processor cache.

We can derive an implementation for a synchronization barrier in the same way as we described above for the readers/writers lock.

- We create a Barrier class, with a lock to protect its internal state variables: how many have checked in so far (count), and how many we are expecting (numThreads).
- We `acquire` the lock at the beginning of `checkin`, and we `release` it at the end.
- Since threads may have to wait in `checkin`, we need a condition variable, allCheckedIn.
- We put the `wait` in a while loop, checking if all n threads have checked in yet.
- The last thread to `checkin` does a `broadcast` to wake up all of the waiters.

Figure 5.12 gives the full implementation. Note that we still use a while loop, even though the signal means that the thread can safely exit `checkin`. There is no harm in using a while statement, and it protects against the possibility of the runtime library issuing spurious wakeups.

```
// A single use synch barrier.
class Barrier{
  private:
    // Synchronization variables
    Lock lock;
    CV allCheckedIn;

    // State variables
    int numEntered;
    int numThreads;

  public:
    Barrier(int n);
```

```
    ~Barrier();
    void checkin();
};

Barrier::Barrier(int n) {
    numEntered = 0;
    numThreads = n;
}

// No one returns until all threads
// have called checkin.
void
checkin() {
    lock.acquire();
    numEntered++;
    if (numEntered < numThreads) {
        while (numEntered < numThreads)
        allCheckedIn.wait(&lock);
    } else { // last thread to checkin
        allCheckedIn.broadcast();
    }
    lock.release();
}
```

Figure 5.12: Candidate implementation of a synchronization barrier. With this implementation, each instance of a barrier can be safely used only one time.

The design is straightforward, but a problem is that the barrier can only be used once. One way to see this is that the state of the barrier does not revert to the same state it had when it was created. Implementing a reusable barrier is a bit more subtle.

- The *first* thread to leave (the one that wakes up the other threads) cannot reset the state, because until the other threads have woken up, the state is needed so that they know to exit the while loop.
- The *last* thread to leave the barrier cannot reset the state for the next iteration, because there is a possible race condition. Suppose a thread finishes `checkin` and calls `checkin` on the next barrier *before* the last thread wakes up and leaves the previous barrier. In that case, the thread would find that n threads have already checked in (because the state hasn't been reset), and so it would think it is "ok to proceed!"

A simple way to implement a re-usable barrier is to use two single-use barriers. The first barrier ensures that all threads are checked in, and the second ensures that all threads have woken up from allCheckedIn.wait. The nth thread to leave can safely reset numCheckedIn; the nth thread to call `checkin` can safely reset numLeaving. Figure 5.13 gives the result.

```
// A re-usable synch barrier.
class Barrier{
  private:
    // Synchronization variables
    Lock lock;
    CV allCheckedIn;
    CV allLeaving;

    // State variables
    int numEntered;
    int numLeaving;
    int numThreads;

  public:
    Barrier(int n);
    ~Barrier();
    void checkin();
};

Barrier::Barrier(int n) {
    numEntered = 0;
    numLeaving = 0;
    numThreads = n;
}

// No one returns until all threads
// have called checkin.
void
checkin() {
    lock.acquire();
    numEntered++;
    if (numEntered < numThreads) {
        while (numEntered < numThreads)
        allCheckedIn.wait(&lock);
    } else {
    // no threads in allLeaving.wait
        numLeaving = 0;
        allCheckedIn.broadcast();
    }
    numLeaving++;
    if (numLeaving < numThreads) {
        while (numLeaving < numThreads)
        allLeaving.wait(&lock);
    } else {
    // no threads in allCheckedIn.wait
        numEntered = 0;
        allLeaving.broadcast();
    }
    lock.release();
}
```

Figure 5.13: Implementation of a re-usable synchronization barrier.

5.6.3 FIFO Blocking Bounded Queue

Assuming Mesa semantics for condition variables, our implementation of the thread-safe blocking bounded queue in Figure 5.8 does not guarantee freedom from starvation. For example, a thread may call remove and wait in the while loop because the queue is empty.

Starvation would occur if every time another thread inserts an item into the queue, a *different* thread calls remove, acquires the lock, sees that the queue is full, and removes the item before the waiting thread resumes.

Often, starvation is not a concern. For example, if we have one thread putting items into the queue, and n equivalent worker threads removing items from the queue, it may not matter which of the worker threads goes first. Even if starvation is a concern, as long as calls to insert and remove are infrequent, or the buffer is rarely empty or full, every thread is highly likely to make progress.

Suppose, however, we do need a thread-safe bounded buffer that does guarantee progress to all threads. We can more formally define the liveness constraint as:

- **Starvation-freedom.** If a thread waits in insert, then it is guaranteed to proceed after a bounded number of remove calls complete, and vice versa.
- **First-in-first-out (FIFO).** A stronger constraint is that the queue is first-in-first-out, or FIFO. The nth thread to acquire the lock in remove retrieves the item inserted by the nth thread to acquire the lock in insert.

Under Hoare semantics, the implementation in Figure 5.8 is FIFO, and therefore also starvation-free, provided that `signal` wakes up the thread waiting the longest.

Here we consider a related question: can we implement a starvation-free or FIFO bounded buffer using Mesa semantics? We need to ensure that when one thread signals a waiter, the waiting thread (and not any other) removes the item.

```
ConditionQueue insertQueue;
ConditionQueue removeQueue;
int numRemoveCalled = 0; // # of times remove has been called
int numInsertCalled = 0; // # of times insert has been called

int
FIFOBBQ::remove() {
    int item;
    int myPosition;
    CV *myCV, *nextWaiter;

    lock.acquire();

    myPosition = numRemoveCalled++;
    mycv = new CV;  // Create a new condition variable to wait on.
    removeQueue.append(myCV);

// Even if I am woken up, wait until it is my turn.
    while (front < myPosition || front == nextEmpty) {
        mycv->Wait(&lock);
    }

    delete self;    // The condition variable is no longer needed.
    item = items[front % size];
    front++;

// Wake up the next thread waiting in insert, if any.
    nextWaiter = insertQueue.removeFromFront();
    if (nextWaiter != NULL)
        nextWaiter->Signal(&lock);
    lock.release();
    return item;
}
```

Figure 5.14: An implementation of FIFO Blocking Bounded Buffer using Mesa semantics. ConditionQueue is a linked list of condition variables.

The easiest way to do this is to create a condition variable for each separate waiting thread. Then, you can be precise as to which thread to wake up! Although you might be worried that this would be space inefficient, on modern computer systems a condition variable (or lock) takes up just a few words of DRAM; it is small compared to the rest of the storage needed per thread.

The outline of the solution is as follows:

- Create a condition variable for every waiter.
- Put condition variables on a queue in FIFO order.
- Signal wakes up the thread at the front of the queue.
- Be CAREFUL about spurious wakeups!

We give an implementation of FIFOBBQ::remove in Figure 5.14; insert is similar.

The implementation easily extends to the case where we want the queue to be last in first out (LIFO) rather than FIFO, or if want it to wake up threads in some priority order. With Hoare semantics, this is not as easy; we would need to have a different implementation of CV for each different queueing discipline, rather than leaving it to those few applications where the specific order matters.

5.7 Implementing Synchronization Objects

Now that we have described locks and condition variables and shown how to use them in shared objects, we turn to how to implement these important building blocks.

Recall from Chapter 4 that threads can be implemented in the kernel or at user level. We start by describing how to implement synchronization for kernel threads; at the end of this section we discuss the changes needed to support these abstractions for user-level threads.

Both locks and condition variables have state. For locks, this is the state of the lock (FREE or BUSY) and a queue of zero or more threads waiting for the lock to become FREE. For condition variables, the state is the queue of threads waiting to be signaled. Either way, the challenge is to atomically modify those data structures.

The Too Much Milk discussion showed that it is both complex and costly to implement atomic actions with just memory reads and writes. Therefore, modern implementations use more powerful hardware primitives that let us atomically read, modify, and write pieces of

state. We use two hardware primitives:

- **Disabling interrupts.** On a single processor, we can make a sequence of instructions atomic by disabling interrupts on that single processor.
- **Atomic read-modify-write instructions.** On a multiprocessor, disabling interrupts is insufficient to provide atomicity. Instead, architectures provide special instructions to atomically read and update a word of memory. These instructions are globally atomic with respect to the instructions on every processor.

Each of these primitives also serves as a memory barrier; they inform the compiler and hardware that all prior instructions must complete before the atomic instruction is executed.

5.7.1 Implementing Uniprocessor Locks by Disabling Interrupts

On a uniprocessor, any sequence of instructions by one thread appears atomic to other threads if no context switch occurs in the middle of the sequence. So, on a uniprocessor, a thread can make a sequence of actions atomic by disabling interrupts (and refraining from calling thread library functions that can trigger a context switch) during the sequence.

This observation suggests a trivial — but seriously limited — approach to implementing locks on a uniprocessor:

```
Lock::acquire() { disableInterrupts(); }

Lock::release() { enableInterrupts(); }
```

This implementation does provide the mutual exclusion property we need from locks. Some uniprocessor kernels use this simple approach, but it does not suffice as a general implementation for locks. If the code sequence the lock protects runs for a long time, interrupts will be disabled for that long. This will prevent other threads from running, and it will make the system unresponsive to handling user inputs or other real-time tasks. Furthermore, although this approach can work in the kernel where all code is (presumably) carefully crafted and trusted to release the lock quickly, we cannot let untrusted user-level code run with interrupts turned off since a malicious or buggy program could then monopolize the processor.

5.7.2 Implementing Uniprocessor Queueing Locks

A more general solution is based on the observation that if the lock is BUSY, there is no point in running the acquiring thread until the lock is free. Instead, we should context switch to the next ready thread.

The implementation briefly disables interrupts to protect the lock's data structures, but re-enables them once a thread has acquired the lock or determined that the lock is BUSY. The Lock implementation shown in Figure 5.15 illustrates this approach. If a lock is BUSY when a thread tries to acquire it, the thread moves its TCB onto the lock's waiting list. The thread then suspends itself and switches to the next runnable thread. The call to suspend does not return until the thread is put back on the ready list, e.g., until some thread calls Lock::release.

```
class Lock {
  private:
    int value = FREE;
    Queue waiting;
  public:
    void acquire();
    void release();
}

Lock::acquire() {
    TCB *chosenTCB;

    disableInterrupts();
    if (value == BUSY) {
        waiting.add(runningThread);
        runningThread->state = WAITING;
        chosenTCB = readyList.remove();
        thread_switch(runningThread,
                      chosenTCB);
        runningThread->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}

Lock::release() {
// next thread to hold lock
    TCB *next;

    disableInterrupts();
    if (waiting.notEmpty()) {
    // move one TCB from waiting
    // to ready
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
```

Figure 5.15: Pseudo-code for a uniprocessor queueing lock. Temporarily disabling interrupts provides atomic access to the data structures implementing the lock. suspend(oldTCB, newTCB) switches from the current thread to the next to be run. It returns only after some other thread calls release and moves it to the ready list.

In our implementation, if a thread is waiting for the lock, a call to release does not set value to FREE. Instead, it leaves value as BUSY. The woken thread is guaranteed to be the next that executes the critical section. This arrangement ensures freedom from starvation.

WARNING: This optimization is specific to this implementation. Users of locks should not make assumptions about the order in which waiting threads acquire a lock.

EXAMPLE: In Lock::acquire, thread_switch is called with interrupts turned off. Who turns them back on?

ANSWER: The next thread to run re-enables interrupts. In particular, most implementations of thread systems enforce the invariant that a thread always disables interrupts before performing a context switch. As a result, interrupts are always disabled when the thread runs again after a context switch. Thus, whenever a thread returns from a context switch, it must re-enable interrupts. For example, the Lock::acquire code in Figure 5.15 re-enables interrupts before returning; the yield implementation in Chapter 4 disables interrupts before the context switch and then re-enables them afterwards. □

5.7.3 Implementing Multiprocessor Spinlocks

On a multiprocessor, however, disabling interrupts is insufficient. Even when interrupts are turned off on one processor, other threads are running concurrently. Operations by a thread on one processor are interleaved with operations by other threads on other processors.

Since turning off interrupts is insufficient, most processor architectures provide *atomic read-modify-write instructions* to support synchronization. These instructions can read a value from a memory location to a register, modify the value, and write the modified value to memory atomically with respect to all instructions on other processors.

Implementing read-modify-write instructions

Students often ask at this point how the processor hardware implements atomic instructions such as test-and-set. If each processor has its own cache, what is to keep two processors from reading and updating the same location at the same time? Although a complete explanation is beyond the scope of this textbook, the hardware uses the same mechanism as it uses for cache coherence.

Every entry in a processor cache has a state, either *exclusive* or *read-only*. If any other processors have a cached copy of the data, it must be *read-only* everywhere. To modify a shared memory location, the processor must have an *exclusive* copy of the data; no other cache is allowed to have a copy. Otherwise, one processor could read an out-of-date value for some location that another processor has already updated. To read or write a location that is stored *exclusive* in some other cache, the processor needs to fetch the latest value from that cache.

Read-modify-write instructions piggyback on this mechanism. To execute one of these instructions, the hardware acquires an *exclusive* copy of the memory, removing copies from all other caches. Then the instruction executes on the local copy; after the instruction completes, other processors are allowed to read the result by fetching the latest value.

As an example, some architectures provide a *test-and-set* instruction, which atomically reads a value from memory to a register and writes the value 1 to that memory location.

```
class SpinLock {
  private:
    int value = 0; // 0 = FREE; 1 = BUSY

  public:
    void acquire() {
        while (test_and_set(&value)) // while BUSY
            ; // spin
    }

    void release() {
        value = 0;
        memory_barrier();
    }
}
```

Figure 5.16: A multiprocessor spinlock implementation using test-and-set.

Figure 5.16 implements a lock using test_and_set. This lock is called a *spinlock* because a thread waiting for a BUSY lock “spins” (busy-waits) in a tight loop until some other lock releases the lock. This approach is inefficient if locks are held for long periods. However, for locks that are only held for short periods (i.e., less time than a context switch would take), spinlocks make sense.

Interrupt handlers and spinlocks

Whenever an interrupt handler accesses shared data, that data must be protected by a spinlock instead of a queueing lock. As we explained in Chapter 2 and Chapter 4, interrupt handlers are not threads: they must run to completion without blocking so that the hardware can deliver the next interrupt. With a queueing lock, the lock might be held when the interrupt handler starts, making it impossible for the interrupt handler to work correctly.

Whenever any thread acquires a spinlock used within an interrupt handler, the thread *must* disable interrupts first. Otherwise, deadlock can result if the interrupt arrives at an inopportune moment. The handler could spin forever waiting for a lock held by the thread it interrupted. Most likely, the system would need to be rebooted to clear the problem.

To avoid these types of errors, most operating systems keep interrupt handlers extremely simple. For example, many interrupt handlers simply wake up a thread to do the heavy lifting of managing the I/O device. Waking up a thread requires mutually exclusive access to the ready list, protected by a spinlock that is never used without first disabling interrupts.

5.7.4 Implementing Multiprocessor Queueing Locks

Often, we need to support critical sections of varying length. For example, we may want a general solution that does not make assumptions about the running time of methods that hold locks.

```
class Lock {
   private:
     int value = FREE;
     SpinLock spinLock;
     Queue waiting;
   public:
     void acquire();
     void release();
}

Lock::acquire() {
    spinLock.acquire();
    if (value != FREE) {
        waiting.add(runningThread);
        scheduler.suspend(&spinLock);
       // scheduler releases spinLock
    } else {
        value = BUSY;
        spinLock.release();
    }
}

void Lock::release() {
    TCB *next;

    spinLock.acquire();
    if (waiting.notEmpty()) {
        next = waiting.remove();
        scheduler.makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
}

class Scheduler {
  private:
    Queue readyList;
    SpinLock schedulerSpinLock;
  public:
    void suspend(SpinLock *lock);
    void makeReady(Thread *thread);
}

void
Scheduler::suspend(SpinLock *lock) {
    TCB *chosenTCB;

    disableInterrupts();
    schedulerSpinLock.acquire();
    lock->release();
    runningThread->state = WAITING;
    chosenTCB = readyList.getNextThread();
    thread_switch(runningThread,
                  chosenTCB);
    runningThread->state = RUNNING;
    schedulerSpinLock.release();
    enableInterrupts();
}

void
Scheduler::makeReady(TCB *thread) {
    disableInterrupts();
    schedulerSpinLock.acquire();
    readyList.add(thread);
    thread->state = READY;
    schedulerSpinLock.release();
    enableInterrupts();
}
```

Figure 5.17: Pseudo-code for a multiprocessor queueing lock. Both the scheduler and the lock use spinlocks to protect their internal data structures. Any thread that tries to acquire the lock when it is BUSY is put on a queue for later wakeup. Care is needed to prevent the waiting thread from being put back on the ready list before it has completed the thread_switch.

We cannot completely eliminate busy-waiting on a multiprocessor, but we can minimize it. As we mentioned, the scheduler ready list needs a spinlock. The scheduler holds this spinlock for only a few instructions; further, if the ready list spinlock is BUSY, there is no point in trying to switch to a different thread, as that would require access to the ready list.

To reduce contention on the ready list spinlock, we use a *separate* spinlock to guard access to each lock's internal state. Once a thread holds the lock's spinlock, the thread can inspect and update the lock's state. If the lock is FREE, the thread sets the value and releases its spinlock. If the lock is BUSY, more work is needed: we need to put the current thread on the waiting list for the lock, suspend the current thread, and switch to a new thread.

Careful sequencing is needed, however, as shown in Figure 5.17. To suspend a thread on a multiprocessor, we need to first disable interrupts to ensure the thread is not preempted while holding the ready list spinlock. We then acquire the ready list spinlock, and *only then* is it safe to release the lock's spinlock and switch to a new thread. The ready list spinlock is released by the next thread to run. Otherwise, a different thread on another processor might put the waiting thread back on the ready list (and start it running) before the waiting thread has completed its context switch.

Later, when the lock is released, if any threads are waiting for the lock, one of them is moved off the lock's waiting list to the scheduler's ready list.

EXAMPLE: What might happen if we released the Lock's spinlock before the call to

suspend?

ANSWER: The basic issue is that we want to make sure the acquiring thread finishes suspending itself before a thread releasing the lock tries to reschedule it. If we allowed makeReady to run before suspend, makeReady would mark the acquring thread READY, but suspend would then change the thread's state to WAITING. The acquiring thread would then be stuck in the WAITING state forever. Since this sequence would happen very rarely, it would be extremely difficult to locate the problem. □

NOTE: In the implementation in Figure 5.17, the single scheduler spinlock can become a bottleneck as the number of processors increases. Instead, as we explain in Chapter 6, most systems have one ready list per processor, each protected by a different spinlock. Different processors can then simultaneously add and remove threads to different lists. Typically, the WAITING thread is placed on the ready list of the same processor where it had previously been RUNNING; this improves cache performance as that processor's cache may still contain code and data from the last time the thread ran. Putting the thread back on the same ready list also prevents the thread from being run by any other processor before the thread has completed its context switch. Once it is READY, any idle processor can run the thread by acquiring the spinlock of the ready list where it is enqueued, removing the thread, and releasing the spinlock.

5.7.5 Case Study: Linux 2.6 Kernel Mutex Lock

We illustrate how locks are implemented in practice by examining the Linux 2.6 kernel. The Linux code closely follows the approach we described above, except that it is *optimized for the common case*.

In Linux, most locks are FREE most of the time. Further, even if a lock is BUSY, it is likely that no other thread is waiting for it. The alternative, that locks are often BUSY, or have long queues of threads waiting for them, means that any thread that needs the lock will usually need to wait, slowing the system down.

The Linux implementation of locks takes advantage of this by providing an extremely fast path for the case when the thread does not need to wait for the lock in acquire, and when there is no thread not need to wake up a thread in release. A slow path, similar to Figure 5.17, is used for all other cases.

Further, having a fast path for acquiring a FREE lock, and releasing a lock with no waiting thread, is also a concern for user-level thread libraries, discussed below.

To optimize the common case path, Linux takes advantage of hardware-specific features of the x86. The x86 supports a large number of different read-modify-write instructions, including atomic decrement (subtract one from the memory location, returning the previous value), atomic increment, atomic exchange (swap the value of the memory location with the value stored in a register), and atomic test-and-set.

The key idea is to design the lock data structures to allow the lock to be acquired and released on the fast path *without* first acquiring the spinlock or disabling interrupts. The slowpath does require acquiring the spinlock. Instead of being binary, the lock value is an integer count with three states:

```
struct mutex {
    /* 1: unlocked, 0: locked, negative: locked, possible waiters */
    atomic_t          count;
    spinlock_t        wait_lock;
    struct list_head  wait_list;
};
```

The Linux lock acquire code is a macro (to avoid making a procedure call on the fast path) that translates to a short sequence of instructions. The x86 lock prefix before the decl instruction signifies to the processor that the instruction should be executed atomically.

```
lock decl (%eax)  // atomic decrement of a memory location
                  // address in %eax is pointer to lock->count
jns 1f            // jump if not signed (if value is now 0)      call slowpath_acquire  1:
```

If the lock was FREE, the lock is acquired with only two instructions; if the lock was BUSY, the code leaves count < 0 and invokes a separate routine to handle the slow path. The slow path disables preemption, acquires the spinlock, puts the thread on the lock wait queue, and then re-checks whether the lock has been released in the meantime. For this, it uses the atomic exchange instruction:

```
for (;;) {
    /*
     * Lets try to take the lock again - this is needed even if
     * we get here for the first time (shortly after failing to
     * acquire the lock), to make sure that we get a wakeup once
     * it's unlocked. Later on, if we sleep, this is the
     * operation that gives us the lock. We xchg it to -1, so
     * that when we release the lock, we properly wake up the
     * other waiters:
     */
    if (atomic_xchg(&lock->count, -1) == 1)
        break;

    /* didn't get the lock, go to sleep: */
    ...
}
```

If successful, the lock is acquired. If unsuccessful, the thread releases the spinlock and switches to the next ready thread. When the thread returns from suspend, unlike in Figure 5.17, the lock may not be FREE, and so the thread must try again.

Eventually, the thread breaks out of the loop, which means that it found a moment when the lock was FREE (lock->count = 1), and at that moment it set the lock to the "busy, possible waiters" state (by setting count = -1). The thread now has the lock, and it cleans up by resetting count = 0 if there are no other waiters.

```
/* set it to 0 if there are no waiters left: */
if (list_empty(&lock->wait_list))
    atomic_set(&lock->count, 0);
```

It then releases the spinlock and re-enables preemptions.

On release, the fast path is two inlined instructions if the lock value was 0 (the lock has no waiters).

```
lock incl (%eax)     // atomic increment      jg 1f        // jump if new value is 1
call slowpath_release  1:
```

On the slow path, count was negative. The increment instruction leaves the lock BUSY. Then, the thread acquires the spinlock, sets the count to be FREE, and wakes up one of the waiting threads.

```
spin_lock_mutex(&lock->wait_lock, flags);
    /*
     * Unlock lock here
     */
atomic_set(&lock->count, 1);
if (!list_empty(&lock->wait_list)) {
    struct mutex_waiter *waiter =
        list_entry(lock->wait_list.next,
            struct mutex_waiter, list);
    wake_up_process(waiter->task);
}
spin_unlock_mutex(&lock->wait_lock, flags);
```

Notice that this function always sets count to 1, even if there are waiting threads. As a result, a new thread may swoop in and acquire the lock on its fast path, setting count = 0. In this case, the waiting thread is still woken up, and when it eventually runs, the main loop above will set count = -1.

This example demonstrates that acquiring and releasing a lock can be inexpensive. Programmers sometimes go to great lengths to avoid acquiring a lock in a particular situation. However, the reasoning in such cases can be subtle, and omitting needed locks is dangerous. In cases where there is little contention, avoiding locks is unlikely to significantly improve performance, so it is usually better just to keep things simple and rely on locks to ensure mutual exclusion when accessing shared state.

5.7.6 Implementing Condition Variables

We can implement condition variables using a similar approach to the one used to implement locks, with one simplification: since the lock is held whenever the `wait`, `signal`, or `broadcast` is called, we already have mutually exclusive access to the condition wait queue. As with locks, care is needed to prevent a waiting thread from being put back on the ready list until it has completed its context switch; we can accomplish this by acquiring the scheduler spinlock *before* we release the monitor lock. Another thread may acquire the monitor lock and start to signal the waiting thread, but it will not be able to complete the signal until the scheduler lock is released immediately after the context switch.

```
class CV {
  private:
    Queue waiting;
  public:
    void wait(Lock *lock);
    void signal();
    void broadcast();
}

// Monitor lock is held by current thread.
void CV::wait(Lock *lock) {
    assert(lock.isHeld());
    waiting.add(myTCB);
    // Switch to new thread and release lock.
    scheduler.suspend(&lock);
    lock->acquire();
}

// Monitor lock is held by current thread.
void CV::signal() {
    if (waiting.notEmpty()) {
        thread = waiting.remove();
        scheduler.makeReady(thread);
    }
}

void CV::broadcast() {
    while (waiting.notEmpty()) {
        thread = waiting.remove();
        scheduler.makeReady(thread);
    }
}
```

Figure 5.18: Pseudo-code for implementing a condition variable. suspend and makeReady are defined in Figure 5.17.

Figure 5.18 shows an implementation with Mesa semantics — when we signal a waiting thread, that thread becomes READY, but it may not run immediately and must still re-acquire the monitor lock. It is possible for another thread to acquire the monitor lock first and to change the state guarded by the lock before the waiting thread returns from CV::wait.

5.7.7 Implementing Application-level Synchronization

The preceding discussion focused on implementing locks and condition variables for kernel threads. In that case, everything (code, shared state, lock data structures, thread control blocks, and the ready list) is in kernel memory, and all threads run in kernel mode. Fortunately, although some details change, the same basic approach works when we implement locks and condition variables for use by threads that run at user level.

Recall from Chapter 4 that there are two ways of supporting application-level concurrency: via system calls to access kernel thread operations or via a user-level thread scheduler.

Kernel-Managed Threads. With kernel-managed threads, the kernel provides threads to a process and manages the thread ready list. The kernel scheduler needs to know when a thread is waiting for a lock or condition variable so that it can suspend the thread and switch to the next ready thread.

In the simplest case, we can place the lock and condition variable data structures, including the waiting lists, in the kernel's address space. Each method call on the synchronization object translates to a system call. Then, the implementations described above for kernel-level locks and condition variables can be used without significant change.

A more sophisticated approach splits the lock's state and implementation into a fast path and slow path, similar to the Linux lock described above. For example, each lock has two data structures: (i) the process's address space holds something similar to the count field and (ii) the kernel holds the spinlock and wait_list queue.

Then, acquiring a FREE lock or releasing a lock with no waiting threads takes a few instructions at user level, with no system call. The slow path still needs a system call (e.g., when a waiting thread needs to suspend execution). We leave the details of the implementation as an exercise for the reader.

User-Managed Threads. In a thread library that operates completely at user level, the library creates multiple kernel threads to serve as virtual processors, and then multiplexes user-level threads over those virtual processors. This situation is similar to kernel threads, except operating inside the process's address space rather than in the kernel's address space. In particular, the code, shared state, lock and condition variable data structures, thread control blocks, and the ready list are in the process's address space.

The only significant change has to do with disabling interrupts. Obviously, a user-level thread package cannot disable system-level interrupts; the kernel cannot allow an untrusted process to disable interrupts and potentially run forever.

Fortunately, the thread library only needs to disable upcalls from the operating system; these are used to trigger thread preemption and other operations in the user-level scheduler, and they could cause inconsistency if they occur while the library is modifying scheduler data structures. Most modern operating systems have a way to temporarily disable upcalls, and then to deliver those upcalls once it is safe to do so. By ensuring the user-level scheduler and upcall handler cannot run at the same time, the fast path mutex implementation described above can be used here as well.

5.8 Semaphores Considered Harmful

> "During system conception it transpired that we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores."
> (From Dijkstra "The structure of the 'THE'-Multiprogramming System" *Communications of the ACM* v. 11 n. 5 May 1968.)

This book focuses on constructing shared objects using locks and condition variables for synchronization. However, over the years, many different synchronization primitives have been proposed, including communicating sequential processes, event delivery, message passing, and so forth. It is important to realize that none of these are more powerful than using locks and condition variables; a program using any of these paradigms can be mapped to monitors using straightforward transformations.

One type of synchronization, a *semaphore*, is worth discussing in detail since it is still widely used. Semaphores were introduced by Dijkstra to provide synchronization in the THE operating system, which (among other advances) explored structured ways of using concurrency in operating system design.

Semaphores are defined as follows:

- A semaphore has a non-negative value.
- When a semaphore is created, its value can be initialized to any non-negative integer.
- Semaphore::P() waits until the value is positive. Then, it atomically decrements value by 1 and returns.
- Semaphore::V() atomically increments the value by 1. If any threads are waiting in P, one is enabled, so that its call to P succeeds at decrementing the value and returns.
- No other operations are allowed on a semaphore; in particular, no thread can directly read the current value of the semaphore.

Note that Semaphore::P is an atomic operation: the read that observes the positive value is atomic with the update that decrements it. As a result, semaphores can never have a negative value, even when multiple threads call P concurrently.

Likewise, if V occurs when there is a waiting thread in P, then P's increment and V's decrement of value are atomic: no other thread can observe the incremented value, and the thread in P is guaranteed to decrement the value and return.

Given this definition, semaphores can be used for either mutual exclusion (like locks) or general waiting for another thread to do something (a bit like condition variables).

To use a semaphore as a mutual exclusion lock, initialize it to 1. Then, Semaphore::P is equivalent to Lock::acquire, and Semaphore::V is equivalent to Lock::release.

Using a semaphore for more general waiting is trickier. A useful analogy for semaphores is thread_join. With thread_join, the precise order of events does not matter: if the forked thread finishes before the parent thread calls thread_join, then the call returns right away. On the other hand, if the parent calls thread_join first, then it waits until the thread finishes, and then returns.

Semaphore P and V can be set up to behave similarly. Typically (but not always), you initialize the semaphore to 0. Then, each call to Semaphore::P waits for the corresponding thread to call V. If the V is called first, then P returns immediately.

The difficulty comes when trying to coordinate shared state (needing mutual exclusion) with general waiting. From a distance, Semaphore::P is *similar to* CV::wait(&lock) and Semaphore::V is *similar to* CV::signal. However, there are important differences. First, CV::wait(&lock) atomically releases the monitor lock, so that you can safely check the shared object's state and then atomically suspend execution.

By contrast, Semaphore::P does *not* release an associated mutual exclusion lock. Typically, the lock is released before the call to P; otherwise, no other thread can access the shared state until the thread resumes. The programmer must carefully construct the program to work properly in this case. Second, whereas a condition variable is stateless, a semaphore has a value. If no threads are waiting, a call to CV::signal has no effect, while a call to Semaphore::V increments the value. This causes the next call to Semaphore::P to proceed without blocking.

Semaphores considered harmful. Our view is that programming with locks and condition variables is superior to programming with semaphores. We advise you to always write your code using those synchronization variables for two reasons.

First, using separate lock and condition variable classes makes code more self-documenting and easier to read. As the quote from Dijkstra notes, two different abstractions are needed, and code is clearer when the role of each synchronization variable is made clear through explicit typing. For example, it is much easier to verify that every lock acquire is paired with a lock release, if they are not mixed with other calls to P and V for general waiting.

Second, a stateless condition variable bound to a lock is a better abstraction for generalized waiting than a semaphore. By binding a condition variable to a lock, we can conveniently wait on any arbitrary predicate on an object's state. In contrast, semaphores rely on the programmer to carefully map the object's state to the semaphore's value so that a decision to wait or proceed in P can be made entirely based on the value, without holding a lock or examining the rest of the shared object's state.

Although we do not recommend writing new code with semaphores, code based on semaphores is not uncommon, especially in operating systems. So, it is important to understand the semantics of semaphores and be able to read and understand semaphore-based code written by others.

NOTE: **Semaphores in interrupt handlers.** In one situation, semaphores are superior to condition variables and locks: synchronizing communication between an I/O device and threads waiting for I/O completion. Typically, the hardware communicates with the device driver via a shared in-memory data structure. This data structure is read and written concurrently by both hardware and the kernel, but the shared access cannot be coordinated with a software lock. Instead, the hardware and device drivers use carefully designed atomic memory operations.

If a hardware device needs attention, e.g., because a network packet has arrived that needs handling, or a disk request has completed, the hardware updates the shared data structure and starts an interrupt handler. The interrupt handler is often simple: it just wakes up a waiting thread and returns. For this, one might consider using a condition variable and calling `signal` without holding the lock (this is sometimes called a *naked notify*). Unfortunately, there is a corner case: suppose that the operating system thread first checks the data structure, sees that no work is currently needed, and is just about to call `wait` on the condition variable. At that moment, the hardware updates the data structure with the new work and triggers the interrupt handler to call `signal`. Because the thread has not called `wait` yet, the `signal` has no effect. Thus, when the thread calls `wait`, the signal has already occurred, and the thread waits — possibly for a long time.

A common solution is for device interrupts to use semaphores instead. Because semaphores are stateful, it does not matter whether the thread calls P or the interrupt handler calls V first: the result is the same, the V cannot be lost.

To help illustrate the difference between semaphores and condition variables, we consider four candidate implementations of condition variables using semaphores.

EXAMPLE: Suppose you are writing concurrent application software on an operating system that only provides semaphores. Does the following code correctly implement condition variables?

```
void CV::wait(Lock *lock) {
    lock->release();
    semaphore.P();
    lock->acquire();
}

void CV::signal() {
    semaphore.V();
}
```

ANSWER: No. Condition variables are stateless, while semaphores have state. We can illustrate this difference with a counterexample.

What happens if a thread calls `signal` and no one is waiting? Nothing. What happens if another thread later calls wait? The thread waits.

By contrast, consider what happens with a semaphore. What happens if a thread calls V and no one is waiting? The value of the semaphore is incremented. What happens if a thread later calls P? The value of the semaphore is decremented, and the thread continues.

In other words, P and V are commutative. The result is the same no matter what order they occur. Condition variables are not commutative: `wait` does not return until the next `signal`. This is why condition variables must be accessed while holding a lock — code

using a condition variable needs to access shared state variables to do its job.

With condition variables, if a thread calls `signal` a thousand times, when no one is waiting, the next `wait` will still go to sleep. With the above code, the next thousand threads that `wait` will return immediately. □

EXAMPLE: What about the following code?

```
void CV::wait(Lock *lock) {
    lock->release();
    semaphore.P();
    lock->acquire();
}

void CV::signal() {
  if (!semaphore.queueEmpty())
      semaphore.V();
}
```

ANSWER: Closer, but still no. For one, the definition of a semaphore does not allow users of the semaphore to look at the contents of the semaphore queue. But more importantly, there is a race condition. Once the lock is released, some other thread can slip in, `acquire` the lock and call `signal` before the waiting thread gets to call P. In that case, the queue is empty, so the waiter never exits the while loop.

Instead, the definition of CV::wait is that the lock is released and the thread goes to sleep atomically. □

EXAMPLE: What about the following code?

```
void CV::wait(Lock *lock) {
    waitQueue.append(myTCB);
    lock->release();
    semaphore.P();
    lock->acquire();
}

void CV::signal() {
    if (!waitQueue.isEmpty())
        semaphore.V();
}
```

ANSWER: Very close but still no. There is still a race condition. Suppose a thread calls `wait`, and releases the lock. Then another thread acquires the lock and calls `signal`. With condition variables, the waiter should wake up, but with the implementation above, a third thread could swoop in, acquire the lock, call `wait`, and decrement the semaphore before the first waiter has a chance to run.

For some programs, this difference would not be noticeable, but for others, it could cause problems. □

EXAMPLE: Is it possible to implement condition variables using semaphores?

ANSWER: Yes, using the technique we outlined for implementing the FIFO bounded buffer: create a semaphore for each waiter and then wake up exactly the right waiter. This solution was developed by Andrew Birrell in order to implement condition variables on top of Microsoft Windows before it supported them natively.

```
// Put thread on queue of waiting
// threads.
void CV::wait(Lock *lock) {
    semaphore = new Semaphore(0);
    waitQueue.Append(semaphore);
    lock.release();
    semaphore.P();
    lock.acquire();
}

// Wake up one waiter if any.
void CV::signal() {
    if (!waitQueue.isEmpty()) {
        semaphore = queue.Remove();
        semaphore.V();
    }
}
```

□

5.9 Summary and Future Directions

This chapter advocates using a systematic, structured approach to writing multi-threaded code that shares state across threads. The approach, shared objects with concurrent access managed with locks and condition variables, has stood the test of time. Using shared objects makes reasoning about multi-threaded programs vastly simpler than it would be if we tried to reason about the possible interleavings of individual loads and stores. Further, by following a systematic approach, we make it possible for others to read, understand, maintain, and change the multi-threaded code we write.

In this chapter, we have discussed:

- **Race conditions.** The fundamental challenge to writing multi-threaded code that uses shared data is that the behavior of the program may depend on the precise ordering of operations executed by each thread. This non-deterministic behavior is difficult to reason about, reproduce, and debug.
- **Locks and condition variables.** Two useful sychronization abstractions are locks, providing mutual exclusion, and condition variables, for waiting for shared state to change.
- **A methodology for writing shared objects.** Using locks and condition variables, we outlined a sequence of steps to writing correct synchronization code for coordinating access to shared objects. Following this methodology has proven enormously helpful

for students in our classes by reducing the likelihood of design errors.

- **Implementations of synchronization.** Locks and condition variables can be efficiently implemented using hardware support for atomic read-modify-write instructions and, where necessary, the ability to temporarily defer hardware interrupts. In particular, we showed that the overhead of acquiring and releasing a non-contested lock can be as low as four instructions.
- **Semaphores.** Semaphores are a widely implemented alternative to locks and condition variables, with a constructive role in managing hardware I/O interrupts.

In short, this chapter defines a set of core skills that almost any programmer will use over and over again during the coming decade or longer.

That is not the whole story. As the next chapter will discuss, complex systems often include many shared objects and threads. This poses new challenges: synchronizing operations that span multiple shared objects, avoiding deadlocks in which a set of threads are all waiting for each other to do something, and maximizing performance when large numbers of threads are contending for a single object.

5.9.1 Historical Notes

Once researchers accepted the need to explicitly manage concurrency using threads, the challenge became how best to coordinate multi-threaded access to shared data. A large number of different abstractions were proposed, and it took some time to work out the different strengths and weaknesses of the various approaches.

Monitors — that is, managing shared data structures with locks and condition variables — were proposed in the early 1970's in separate papers by Tony Hoare [83] and Per Brinch Hansen [75]. One early advantage of monitors was the ability to formally prove properties about multi-threaded code; for example with Hoare-style semantics for condition variables, any statement which is true of the shared object immediately before a `signal` is also true of the object immediately after the return from `wait`. As we saw with the Too Much Milk example, without explicit synchronization, it can be quite difficult to reason about concurrent execution.

By the early 1980's, Xerox PARC had built the first personal computer, the Alto, with all of its system software written using threads (called lightweight processes at the time) and monitors. The methodology we present in this chapter originated with that project [98]. It is hard to overstate how radical an approach this was; almost all widely used operating systems of the time, including UNIX, were built using semaphores.

An alternative line of work advocated completely prohibiting access by threads to shared data, as a way of eliminating race conditions. Instead of shared data, all data was private to a single thread; as a result, locks were never needed. An early example of this approach was Communicating Sequential Processes (CSP), also developed by Tony Hoare [84]. Google's Go language for concurrent web programming is a modern language that supports both monitors and the CSP style of programming. With CSP and Go, a thread that needs to perform an operation on some other thread's data sends it a message; the receiving thread can either reply with the result, or in data-flow style, forward the result onto some other thread.

While there was considerable and vigorous debate at the time as to whether message-passing or shared-memory were better models for programming concurrency, the debate was largely resolved by a simple observation made by Lauer and Needham [101]. Any program using monitors can be recast into CSP using a simple transformation, and vice versa. The execution of a procedure with a monitor lock is equivalent to processing a message in CSP; a monitor is, in effect, single-threaded while it is holding the lock. Thus, the choice of which style to use is largely a matter of taste and convention, and most programmers have chosen to use threads and monitors.

Exercises

1. True or False: If a multi-threaded program runs correctly in all cases on a single time-sliced processor, then it will run correctly if each thread is run on a separate processor of a shared-memory multiprocessor. Justify your answer.
2. Show that solution 3 to the Too Much Milk problem is safe — that it guarantees that at most one roommate buys milk.
3. Precisely describe the set of possible outputs that could occur when the program shown in Figure 5.5 is run.
4. Suppose that you mistakenly create an automatic (local) variable v in one thread t1 and pass a pointer to v to another thread t2. Is it possible that a write by t1 to some variable other than v will change the state of v as observed by t2? If so, explain how this can happen and give an example. If not, explain why not.
5. Suppose that you mistakenly create an automatic (local) variable v in one thread t1 and pass a pointer to v to another thread t2. Is it possible that a write by t2 to v will cause t1 to execute the wrong code? If so, explain how. If not, explain why not.
6. Assuming Mesa semantics for condition variables, our implementation of the blocking bounded queue in Figure 5.8 does not guarantee freedom from starvation: if a continuous stream of threads makes insert (or remove) calls, a waiting thread could wait forever. For example, a thread may call insert and wait in the while loop because the queue is full. Starvation would occur if every time another thread removes an item from the queue and signals the waiting thread, a *different* thread calls insert, sees that the queue is not full, and inserts an item before the waiting thread resumes.

   Prove that under Hoare semantics and assuming that signal wakes the longest-waiting thread, our implementation of BBQ ensures freedom from starvation. More precisely, prove that if a thread waits in insert, then it is guaranteed to proceed after a bounded number of remove calls complete, and vice versa.
7. As noted in the previous problem, our implementation of the blocking bounded queue in Figure 5.8 does not guarantee freedom from starvation. Modify the code to ensure freedom from starvation so that if a thread waits in insert, it is guaranteed to proceed after a bounded number of remove() calls complete, and vice versa. **Note:** Your

implementation must work under Mesa semantics for condition variables.

8. Wikipedia provides an implementation of Peterson's algorithm to provide mutual exclusion using loads and stores at http://en.wikipedia.org/wiki/Peterson's_algorithm. Unfortunately, this code is not guaranteed to work with modern compilers or hardware. Update the code to include memory barriers where necessary. (Of course, you could add a memory barrier before and after each instruction; your solution should instead add memory barriers only where necessary for correctness.)

9. Linux provides a sys_futex() system call to assist in implementing hybrid user-level/kernel-level locks and condition variables.

   A call to long sys_futex(void *addr1, FUTEX_WAIT, int val1, NULL, NULL, 0) checks to see if the memory at address addr1 has the same value as val1. If so, the calling thread is suspended. If not, the calling thread returns immediately with the error return value EWOULDBLOCK. In addition, the system call returns with the value EINTR if the thread receives a signal.

   A call to long sys_futex(void *addr1, FUTEX_WAKE, 1, NULL, NULL, 0) causes one thread waiting on addr1 to return.

   Consider the following simple implementation of a hybrid user-level/kernel-level lock.

```
class TooSimpleFutexLock {
  private:
    int val;

  public:
    TooSimpleMutex() : val (0) { }  // Constructor

    void acquire () {
        int c;
        // atomic_inc returns *old* value
        while ((c = atomic_inc (val)) != 0) {
            futex_wait (&val, c + 1);
        }
    }

    void release () {
        val = 0;
        futex_wake (&val, 1);
    }
};
```

   There are three problems with this code.

   a. **Performance.** The goal of this code is to avoid making expensive system calls in the uncontested case of an `acquire` on a FREE lock or a `release` of a lock with no other waiting threads. This code fails to meet this goal. Why?

   b. **Performance.** A subtle corner case occurs when multiple threads try to acquire the lock at the same time. It can show up as occasional slowdowns and bursts of CPU usage. What is the problem?

   c. **Correctness**. A corner case can cause the mutual exclusion correctness condition to be violated, allowing two threads to both believe they hold the lock. What is the problem?

10. In the readers/writers lock example for the function RWLock::doneRead, why do we use writeGo.Signal rather than writeGo.Broadcast?

11. Show how to implement a semaphore by generalizing the multi-processor lock implementation shown in Figure 5.17.

12. In Section 5.1.3, we presented a solution to the Too Much Milk problem. To make the problem more interesting, we will also allow roommates to drink milk.

    Implement in C++ or Java a Kitchen class with a drinkMilkAndBuyIfNeeded(). This method should randomly (with a 20% probability) change the value of milk from 1 to 0. Then, if the value just became 0, it should buy milk (incrementing milk back to 1). The method should return 1 if the roommate bought milk and 0 otherwise.

    Your solution should use locks for synchronization and work for any number of roommates. Test your implementation by writing a program that repeatedly creates a Kitchen object and varying numbers of roommate threads; each roommate thread should call drinkMilkAndBuyIfNeeded() multiple times in a loop.

    Hint: You will probably write a main() thread that creates a Kitchen object, creates multiple roommate threads, and then waits for all of the roommates to finish their loops. If you are writing in C++ with the POSIX threads library, you can use pthread_join() to have one thread wait for another thread to finish. If you are writing in Java with the java.lang.Thread class, you can use the join() method.

13. For the solution to Too Much Milk suggested in the previous problem, each call to drinkMilkAndBuyIfNeeded() is atomic and holds the lock from the start to the end even if one roommate goes to the store. This solution is analogous to the roommate padlocking the kitchen while going to the store, which seems a bit unrealistic.

    Implement a better solution to drinkMilkAndBuyIfNeeded() using both locks and condition variables. Since a roommate now needs to release the lock to the kitchen while going to the store, you will no longer acquire the lock at the start of this function and release it at the end. Instead, this function will call two helper-functions, each of which acquires/releases the lock. For example:

```
int Kitchen::drinkMilkAndBuyIfNeeded() {
    int iShouldBuy = waitThenDrink();

    if (iShoudBuy) {
        buyMilk();
    }
}
```

In this function, waitThenDrink() should wait if there is no milk (using a condition variable) until there is milk, drink the milk, and if the milk is now gone, return a nonzero value to flag that the caller should buy milk. BuyMilk() should buy milk and then broadcast to let the waiting threads know that they can proceed.

Again, test your code with varying numbers of threads.

14. Before entering a *priority critical section*, a thread calls PriorityLock::enter(priority). When the thread exits the critical section, it calls PriorityLock::exit(). If several threads are waiting to enter a priority critical section, the one with the numerically highest priority should be the next one allowed in. Implement PriorityLock using monitors (locks and condition variables) and following the programming standards defined in this chapter.
    a. Define the state and synchronization variables and describe the purpose of each.
    b. Implement PriorityLock::enter(int priority).
    c. Implement PriorityLock::exit().
15. Implement a *priority condition variable.* A priority condition variable (PCV) has three public methods:

```
void PCV::wait(Lock *enclosingLock, int priority);

void PCV::signal(Lock *enclosingLock);

void PCV::broadcast(Lock *enclosingLock, int priority);
```

These methods are similar to those of a standard condition variable. The one difference is that a PCV enforces both *priority* and *ordering*.

In particular, signal(Lock *lock) causes the currently waiting thread with the highest priority to return from wait; if multiple threads with the same priority are waiting, then the one that is waiting the longest should return before any that have been waiting a shorter amount of time.

Similarly, broadcast(Lock *lock, int priority) causes all currently waiting threads whose priority equals or exceeds priority to return from wait.

For full credit, you must follow the *thread coding standards* described in this chapter.

16. A synchronous buffer is one where the thread placing an item into the buffer waits until the thread retrieving the item has gotten it, and only then returns.

    Implement a synchronous buffer using Mesa-style locks and condition variables, with the following routines:

```
// Put item into the buffer and return only once the item
// has been retrieved by some thread.
SyncBuf::put(item);

// Wait until there is an item in the buffer, and then return it.
SyncBuf::get();
```

Any number of threads can concurrently call SyncBuf::get and SyncBuf::put; the module pairs off puts and gets. Each item should be returned exactly once, and there should be no unnecessary waiting. Once the item is retrieved, the thread that called put with the item should return.

17. You have been hired by a company to do climate modelling of oceans. The inner loop of the program matches atoms of different types as they form molecules. In an excessive reliance on threads, each atom is represented by a thread.
    a. Your task is to write code to form water out of two hydrogen threads and one oxygen thread (H2O). You are to write the two procedures: HArrives() and OArrives(). A water molecule forms when two H threads are present and one O thread; otherwise, the atoms must wait. Once all three are present, one of the threads calls MakeWater(), and only then, all three depart.
    b. The company wants to extend its work to handle cloud modelling. Your task is to write code to form ozone out of three oxygen threads. Each of the threads calls OArrives(), and when three are present, one calls MakeOzone(), and only then, all three depart.
    c. Extending the product line into beer production, your task is to write code to form alcohol (C2H6O) out of two carbon atoms, six hydrogens, and one oxygen.

    You must use locks and Mesa-style condition variables to implement your solutions, using the best practices as defined in this chapter. Obviously, an atom that arrives *after* the molecule is made must wait for a different group of atoms to be present. There should be no busy-waiting and you should correctly handle spurious wakeups. There must also be no useless waiting: atoms should not wait if there is a sufficient number of each type.

6. Multi-Object Synchronization

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone. —*Kansas state law, early 1900s*

In the previous chapter, we described a key building block for writing concurrent programs: how to design an object that can be shared between multiple threads. In this chapter, we need to go one step further: what happens as programs become more complex, with multiple shared objects and multiple locks? To answer this, we need to reason about the interactions between shared objects.

Several considerations arise in this context:

- **Multiprocessor performance.** Modern computers have increasing numbers of processors because of the difficulty of improving single CPU performance. The design of shared objects can have a large impact on multiprocessor performance. For example, a lock protecting a frequently accessed shared object can become a bottleneck, since only one thread can hold the lock at a time.
- **Correctness.** Performance considerations often cause designers to re-engineer their data structures for increased concurrency. Splitting a single shared object into a set of related objects each with their own lock can improve performance. However, it also raises issues of correctness. For programs with multiple shared objects, we face a problem similar to the one faced when reasoning about atomic loads and stores: even if each individual operation on a shared object is atomic, we must reason about interactions of sequences of operations across objects.
- **Deadlock.** One way to help reason about the behavior of operations across multiple objects is to hold multiple locks. This approach raises the possibility of deadlock, where threads are permanently stuck waiting for each other in a cycle.

No cookbook recipe always works for addressing these challenges. In particular, current techniques have two basic limitations. First, they pose engineering trade-offs. Some solutions are general but complex or expensive; others are simple but slow; still others are simple and cheap but not general. Second, many solutions are inherently *non-modular*: they require reasoning about the global structure of the system and internal implementation details of modules to understand or restrict how different modules can interact.

Chapter roadmap:

- **Multiprocessor Lock Performance.** Can we predict when a lock will become a bottleneck on a multiprocessor? (Section 6.1)
- **Lock Design Patterns.** If a lock is a bottleneck, can we restructure the program to reduce the problem? (Section 6.2)

- **Lock Contention.** If a lock is still a bottleneck after re-structuring, what then? (Section 6.3)
- **Multi-Object Atomicity.** How can we make a sequence of operations across multiple objects appear atomic to other threads? (Section 6.4)
- **Deadlock.** What causes deadlock in multi-threaded programs, and what solutions exist to prevent or break deadlocks? (Section 6.5)
- **Non-Blocking Synchronization.** Are there ways to eliminate locks in complex multi-object programs? (Section 6.6)

6.1 Multiprocessor Lock Performance

Client-server applications often have ample parallelism for modern multicore architectures with dozens of processors. Each separate client request can be handled by a different thread running on a different processor; this is called *request parallelism*. Likewise, server operating systems often have ample parallelism – applications with large numbers of threads can make a large number of concurrent system calls into the kernel.

Even with ample request parallelism, however, performance can often be disappointing. Once locks and condition variables are added to a server application to allow it to process requests concurrently, throughput may be only slightly faster on a fifty-way multiprocessor than on a uniprocessor. Most often, this can be due to three causes:

1. **Locking.** A lock implies mutual exclusion — only one thread at a time can hold the lock. As a result, access to a shared object can limit parallelism.
2. **Communication of shared data.** The performance of a modern processor can vary by a factor of ten (or more) depending on whether the data needed by the processor is already in its cache or not. Modern processors are designed with large caches, so that almost all of the data needed by the processor will already be stored in the cache. On a uniprocessor, it is rare that the processor needs to wait.

   However, on a multiprocessor, the situation is different. Shared data protected by a lock will often need to be copied from one cache to another. Shared data is often in the cache of the processor that last held the lock, and it is needed in the cache of the processor that is next to acquire the lock. Moving data can slow critical section performance significantly compared to a uniprocessor.
3. **False sharing.** A further complication is that the hardware keeps track of shared data at a fixed granularity, often in units of a cache entry of 32 or 64 bytes. This reduces hardware management overhead, but it can cause performance problems if multiple data structures with different sharing behavior fit in the same cache entry. This is called *false sharing*.

Fortunately, these effects can be reduced through careful design of shared objects. We caution, however, that you should keep your shared object design simple until you have proven, through detailed measurement, that a more complex design is necessary to achieve your performance target.

The evolution of Linux kernel locking

The first versions of Linux ran only on uniprocessor machines. To allow Linux to run on multiprocessors, version 2.0 introduced the Big Kernel Lock (BKL) — a single lock that protected all of the kernel's shared data structures. The BKL allowed the kernel to function on multiprocessor machines, but scalability and performance were limited. So, over time, different subsystems and different data structures got their own locks, allowing them to be accessed without holding the BKL.

By version 2.6, Linux has been highly optimized to run well on multiprocessor machines. It now has thousands of different locks, and researchers have demonstrated scalability for a range of benchmarks on a 48 processor machine. Still, the BKL remains in use in a few — mostly less performance-critical — parts of the Linux kernel, like the reboot system call, some older file systems, and some device drivers.

To illustrate these concepts, consider a web server with an in-memory cache of recently fetched pages. It is often faster to simply return a page from memory rather than regenerating it from scratch. For example, Google might receive a large number of searches for election results on election night, and there is little reason to do all of the work of a general search in that case.

To implement caching of web pages, the server might have a shared data structure, such as a hash table on the search terms, to point to the cached page if it exists. The hash table is shared among the threads handling client requests, and therefore needs a lock. The hash table is updated whenever a thread generates a new page that is not in the cache. The code might also mark pages that have been recently fetched, to keep them in memory in preference to other requests that do not occur as frequently.

An important question in this design is whether the single lock on the hash table will significantly limit server parallelism. How can we tell if the lock on a shared object is going to be a problem?

A convenient approach is to derive a bound on the performance of a parallel program by assuming that the rest of the program is perfectly parallelizable — in other words, that the only limiting factor is that only one thread at a time can hold the shared lock.

EXAMPLE: Suppose that, on average, a web server spends 5% of each request accessing and manipulating its hash table of recently used web pages. If the hash table is protected by a single lock, what is the maximum possible throughput gain?

ANSWER: The time spent in the critical section is inherently sequential. If we assume all other parts of the server are perfectly parallelizable, then the maximum speedup is a **factor of 20** regardless of how many processors are used. □

As we mentioned earlier, a further complication is that it can take much longer to fetch shared data on a multiprocessor because the data is unlikely to be in the processor cache. If the portion of the program holding the lock is slower on a multiprocessor than on a uniprocessor, the potential gain in throughput can be severely limited.

EXAMPLE: In the example above, what is the maximum throughput improvement if the

hash table code runs four times slower on a multiprocessor due to the need to move shared data between processor caches?

ANSWER: The potential throughput improvement would be small even if a large number of processors are used.

$$\text{Throughput gain} \leq \frac{1/4 \times}{0.05} = \mathbf{5}$$

□

We can study the effect of cache behavior on multiprocessor performance with a simple experiment. The experiment is a intended only as an illustration; it is not meant a reflection of normal program behavior, but rather as a way of isolating the effect of hardware on the performance of code using shared objects.

Suppose we set up an array of a thousand shared objects, where each object is a simple integer counter protected by a spinlock. (We use a spinlock rather than a lock to avoid measuring context switch time.) The program iterates through the array. For each item, it acquires the lock, increments the counter, and releases the lock. We repeat the loop a thousand times to improve measurement precision.

Consider the following scenarios:

- **One thread, one array.** When one thread iterates through the array, incrementing each counter in turn, the test gives the time it takes to acquire and release an array of uncontended locks.
- **Two threads, two arrays.** When two threads iterate through disjoint arrays, this gives the slowdown when doing work in parallel. On most architectures, there is little to no slowdown to parallel execution.
- **Two threads, one array.** When two threads iterate through the *same* array, each lock is acquired by a thread running on one processor, and then, shortly afterwards, acquired by a different thread running on a different processor. Thus, the performance illustrates the added cost of moving the shared object data from one processor to another.
- **Two threads, alternate elements of one array.** To measure the impact of false sharing, one thread can iterate through the array acquiring the odd entries, and the other thread can iterate through the array acquiring the even entries. If there was no effect to false sharing, this would be identical to the two array case — the threads never use the same data.

One thread, one array	51.2
Two threads, two arrays	52.5
Two threads, one array	197.4
Two threads, alternating	127.3

Figure 6.1: Number of CPU cycles to execute a simple critical section to increment a counter. Measurements taken on a 64-core AMD Opteron 6262, with threads assigned to processor cores that do not share a cache. The performance difference between these cases largely disappears when threads are assigned to cores that share an L2 cache.

Table 6.1 shows example results for a single multiprocessor, a 64-core AMD Opteron; the performance on different machines will vary. The threads were assigned to cores that do not share a cache.

On this machine, there is very little slowdown in critical section performance when threads access disjoint locks. However, critical section execution time slows down by a factor of four when multiple processors access the same data. The slowdown is also significant when false sharing occurs.

6.2 Lock Design Patterns

We next discuss a set of approaches that can reduce the impact of locking on multiprocessor performance. Often, the best practice is to start simple, with a single lock per shared object. If an object's interface is well designed, then refactoring its implementation to increase concurrency and performance can be done once the system is built and performance measurements can identify any bottlenecks. An adage to follow is: "It is easier to go from a working system to a working, fast system than to go from a fast system to a fast, working system."

We discuss four design patterns to increase concurrency when it is necessary:

- **Fine-Grained Locking.** Partition an object's state into different subsets each protected by a different lock.
- **Per-Processor Data Structures.** Partition an object's state so that all or most accesses are performed by threads running on the same processor.
- **Ownership Design Pattern.** Remove a shared object from a shared container so that only a single thread can read or modify the object.
- **Staged Architecture.** Divide system into multiple stages, so that only those threads within each stage can access that stage's shared data.

6.2.1 Fine-Grained Locking

A simple and widely used approach to decrease contention for a shared lock is to partition

the shared object's state into different subsets, each protected by its own lock. This is called *fine-grained locking*.

The web server cache discussed above provides an example. The cache can use a shared hash table to store and locate recently used web pages; because the hash table is shared, it needs a lock to provide mutual exclusion. The lock is acquired and released at the start and end of each of the hash table methods: put(key, value), value = get(key), and value = remove(key).

If the single lock limits performance, an alternative is to have one lock per hash bucket. The methods acquire the lock for bucket b before accessing any record that hashes to that bucket. Provided that the number of buckets is large enough, and no single bucket receives a large fraction of requests, then different threads can use and update the hash table in parallel.

However, there is no free lunch. Dividing an object's state into different pieces protected by different locks can significantly increase the object's complexity. Suppose we want to implement a hash table whose number of hash buckets grows as the number of objects it stores increases. If we have a single lock, this is easy to do. But, what if we use fine-grained locking? Then, the design becomes more complex because we have some methods, like put and get, that operate on one bucket and other methods, like resize, that operate across multiple buckets.

Several solutions are possible:

1. **Introduce a readers/writers lock.** Suppose we have a readers/writers lock on the overall structure of the hash table (e.g., the number of buckets and the array of buckets) and a mutual exclusion lock on each bucket. Methods that work on a single bucket at a time, such as put and get, acquire the table's readers/writers lock in read mode and also acquire the relevant bucket's mutual exclusion lock. Methods that change the table's structure, such as resize, must acquire the readers/writers lock in write mode; the readers/writers lock prevents any other threads from using the hash table while it is being resized.
2. **Acquire every lock.** Methods that change the structure of the hash table, such as resize, must first iterate through every bucket, acquiring its lock, before proceeding. Once resize has a lock on every bucket, it is guaranteed that no other thread is concurrently accessing or modifying the hash table.
3. **Divide the hash key space.** Another solution is to divide the hash key space into r regions, to have a mutual exclusion lock for each region, and to allow each region to be resized independently when it becomes heavily loaded. Then, get, put, and resizeRegion each acquire the relevant region's mutual exclusion lock.

Which solution is best? It is not obvious. The first solution is simple and appears to allow high concurrency, but acquiring the readers/writers lock even in read mode may have high overhead. For example, we gave an implementation of a readers/writers lock in Chapter 5 where acquiring a read-only lock involves acquiring a mutual exclusion lock on both entrance and exit. Access to the underlying mutual exclusion lock may become a bottleneck.

The second solution makes resize expensive, but if resize is a rare operation, that may be acceptable. The third solution balances concurrency for get/put against the cost of resize, but it is more complex and may require tuning the number of groups to get good performance.

Further, these trade-offs may change as the implementation becomes more complex. For example, to trigger resize at appropriate times, we probably need to maintain an additional nObjects count of the number of objects currently stored in the hash table, so whatever locking approach we use would need to be extended to cover this information.

EXAMPLE: How might you use fine-grained locking to reduce contention for the lock protecting the shared memory heap in malloc/free or new/delete?

ANSWER: One approach would be to partition the heap into separate memory regions, each with its own lock. For example, a fast implementation of a heap on a uniprocessor uses n buckets, where the ith bucket contains blocks of size 2^i, and serves requests of size $2^{i-1} + 1$ to 2^i. If there are no free blocks in the ith bucket, an item from the next larger bucket i + 1 is split in two. Using fine-grained locking, each bucket can be given its own lock. □

6.2.2 Per-Processor Data Structures

A related technique to fine-grained locking is to partition the shared data structure based on the number of processors on the machine. For example, instead of one shared hash table of cached pages, an alternative design would have N separate hash tables, where N is the number of processors. Each thread uses the hash table based on the processor where it is currently running. Each hash table still needs its own lock in case a thread is context switched in the middle of an operation, but in the common case, only threads running on the same processor contend for the same lock.

Often, this is combined with a per-processor ready list, ensuring that each thread preferentially runs on the same processor each time it is context switched, further improving execution speed.

An advantage of this approach is better hardware cache behavior; as we saw in the previous section, shared data that must be communicated between processors can slow down the execution of critical sections. Of course, the disadvantage is that the hash tables are now partitioned, so that a web page may be cached in one processor's hash table, and needed in another. Whether this is a performance benefit depends on the relative impact of reducing communication of shared data versus the decreased effectiveness of the cache.

EXAMPLE: How might you use per-processor data structures to reduce contention for the memory heap? Under what conditions would this work well?

ANSWER: The heap can be partitioned into N separate memory regions, one for each processor. Calls to malloc/new would use the local heap; free/delete would return the data to the heap where it was allocated. **This would perform well provided that (i) rebalancing the heaps was rare and (ii) most allocated data is freed by the thread that acquires it.** □

6.2.3 Ownership Design Pattern

A common synchronization technique in large, multi-threaded programs is an *ownership design pattern*. In this pattern, a thread removes an object from a container and can then access the object without holding a lock: the program structure guarantees that at most one thread owns an object at a time.

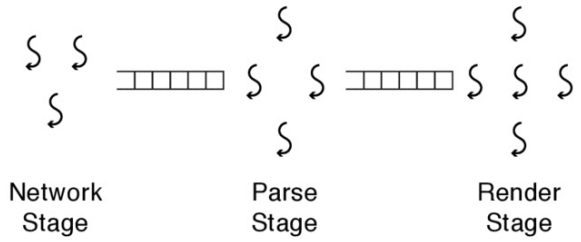

Figure 6.2: A multi-stage server based on the ownership pattern. In the first stage, one thread exclusively owns each network connection. In later stages, one thread parses and renders a given object at a time.

As an example, a single web page can contain multiple objects, including HTML frames, style sheets, and images. Consider a multi-threaded web browser whose processing is divided into three stages: receiving an object via the network, parsing the object, and rendering the object (see Figure 6.2). The first stage has one thread per network connection; the other stages have several worker threads, each of which processes one object at a time.

The work queues between stages coordinate object ownership. Objects in the queues are not being accessed by any thread. When a worker thread in the *parse* stage removes an object from the stage's work queue, it owns the object and has exclusive access to it. When the thread is done parsing the object, it puts it into the second queue and stops accessing it. A worker thread from the *render* stage then removes it from the second queue, gaining exclusive access to it to render it to the screen.

EXAMPLE: How might you use the ownership design pattern to reduce contention for the memory heap?

ANSWER: Ownership can be seen as an extension of per-processor data structures; instead of one heap per processor, **we can have one heap per thread.** Provided that the same thread that allocates memory also frees it, the thread can safely use its own heap without a lock and only return to the global heap when the local heap is out of space. □

Commutative interface design

Class and interface design can often constrain implementations in ways that require locking. An example is the UNIX API. Like most operating systems, the UNIX open system call returns a file handle that is used for further operations on the file; the same system call is also used to initialize a network socket. The open call gives the operating system the ability to allocate internal data structures to track the current state of the file or network socket, and more broadly, which files and sockets are in use.

UNIX also specifies that each successive call to open returns the next integer file handle; as we saw in Chapter 3, the UNIX shell uses this feature when redirecting stdin and stdout to a file or pipe.

A consequence of the design of the UNIX API is that the implementation of open requires a lock. For early UNIX systems, this was not an issue, but modern multi-threaded web servers open extremely large numbers of network sockets and files. Because of the semantics of the API, the implementation of open cannot use fine-grained locking or a per-processor data structure.

A better choice, where possible, is to design the API to be *commutative*: the result of two calls is the same regardless of which call was made first. For example, if the implementation can return any unique integer as a file handle, rather than the next successive one, then the implementation could allocate out of a per-processor bucket of open file handles. The implementation would then need a lock only for the special case of allocating specific handles such as stdin and stdout.

6.2.4 Staged Architecture

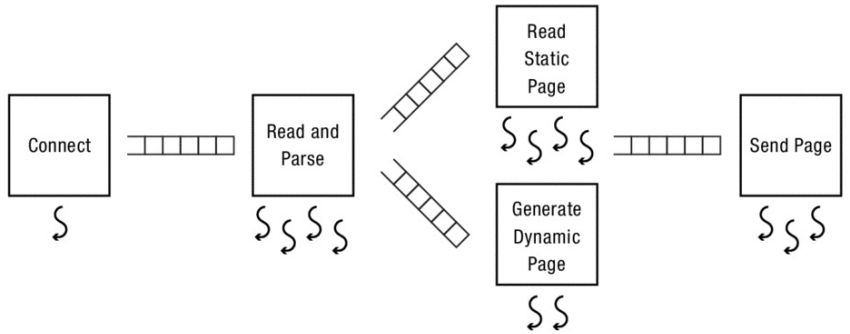

Figure 6.3: A staged architecture for a simple web server.

The *staged architecture* pattern, illustrated in Figure 6.3, divides a system into multiple subsystems, called stages. Each stage includes state private to the stage and a set of one or more worker threads that operate on that state. Different stages communicate by sending messages to each other via shared producer-consumer queues. Each worker thread repeatedly pulls the next message from a stage's incoming queue and then processes it, possibly producing one or more messages for other stages' queues.

Figure 6.3 shows a staged architecture for a simple web server that has a first *connect* stage that uses one thread to set up network connections and that passes each connection to a second *read and parse* stage.

The *read and parse* stage has several threads, each of which repeatedly gets a connection from the incoming queue, reads a request from the connection, parses the request to determine what web page is being requested, and checks to see if the page is already cached.

Assuming the page is not already cached, if the request is for a static web page (e.g., an HTML file), the *read and parse* stage passes the request and connection to the *read static page* stage, where one of the stage's threads reads the specified page from disk. Otherwise, the *read and parse* stage passes the request and connection to the *generate dynamic page* stage, where one of the stage's threads runs a program that dynamically generates a page in response to the request.

Once the page has been fetched or generated, the page and connection are passed to the *send page* stage, where one of the threads transmits the page over the connection.

The key property of a staged architecture is that the state of each stage is private to that stage. This improves modularity, making it easier to reason about each stage individually and about interactions across stages.

As an example of the modularity benefits, consider a system where different stages are produced by different teams or even different companies. Each stage can be designed and tested almost independently, and the system is likely to work as expected when the stages are brought together. For example, it is common practice for a web site to use a web server from one company and a database from another company and for the two to communicate via messages.

Another benefit is improved cache locality. A thread operating on a subset of the system's state may have better cache behavior than a thread that accesses state from all stages. On the other hand, for some workloads, passing a request from stage to stage could hurt cache behavior compared to doing all of the processing for a request on one processor.

Also note that for good performance, the processing in each stage must be large enough to amortize the cost of sending and receiving messages.

The special case of exactly one thread per stage is *event-driven programming*, described in Chapter 4. With event-driven programming, there is no concurrency within a stage, so no locking is required. Each message is processed atomically with respect to that stage's state.

One challenge with staged architectures is dealing with overload. System throughput is limited by the throughput of the slowest stage. If the system is overloaded, the slowest stage will fall behind, and its work queue will grow. Depending on the system's implementation, two bad things could happen. First, the queue could grow indefinitely, consuming more and more memory until the system memory heap is exhausted. Second, if the queue is limited to a finite size, once that size is reached, earlier stages must either discard work for the overloaded stage or block until the queue has room. Notice that if they block, then the backpressure will limit the throughput of earlier stages to that of the bottleneck stage, and their queues in turn may begin to grow.

One solution is to dynamically vary the number of threads per stage. If a stage's incoming queue is growing, the program can shift processing resources to it by reducing the number of threads for a lightly-loaded stage in favor of more threads for the stage that is falling behind.

6.3 Lock Contention

Sometimes, even after applying the techniques described in the previous section, locking may remain a bottleneck to good performance on a multiprocessor. For example, with fine-grained locking of a hash table, if a bucket contains a particularly popular item, say the cached page for Justin Bieber, then the lock on that bucket can be a source of contention.

In this section, we discuss two alternate implementations of the lock abstraction that work better for locks that are bottlenecks:

- **MCS Locks.** MCS is an implementation of a spinlock optimized for the case when there are a significant number of waiting threads.
- **RCU Locks.** RCU is an implementation of a reader/writer lock, optimized for the case when there are many more readers than writers. RCU reduces the overhead for readers at a cost of increased overhead for writers. More importantly, RCU has somewhat different semantics than a normal reader/writer lock, placing a burden on the user of the lock to understand its dangers.

Although both approaches are used in modern operating system kernels, we caution that neither is a panacea. They should only be used once profiling has shown that the lock is a source of contention and no other options are available.

6.3.1 MCS Locks

Recall that the lock implementation described in Chapter 5 was tuned for the common case where the lock was usually FREE. Is there an efficient implementation of locks when the lock is usually BUSY?

Unfortunately, the overhead of acquiring and releasing a lock can *increase* dramatically with the number of threads contending for the lock. For a contended lock, this can further increase the number of threads waiting for the lock. Consider again the example we used earlier, of a spinlock protecting a shared counter:

```
void Counter::Increment() {
    while (test_and_set(&lock)) // while BUSY
        ; // spin
    value++;
    lock = FREE;
    memory_barrier();
}
```

Even if many threads try to increment the same counter, only one thread at a time can execute the critical section; the other threads must wait their turn. As we observed earlier, because the counter value must be communicated from one lock holder to the next, the critical section will take significantly longer on a multiprocessor than on a single processor.

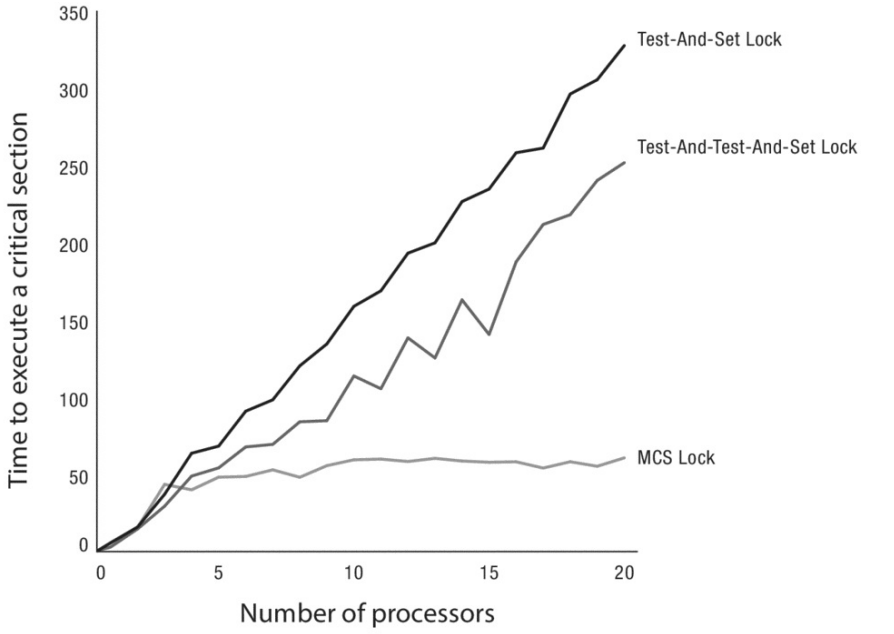

Figure 6.4: The overhead of three alternative lock implementations as a function of the number of processors contending for the lock: (a) test-and-set, (b) test and test-and-set, and (c) MCS. Measurements taken on a 64-core AMD Opteron 6262. The non-smooth curves are typical of measurements of real systems.

However, the situation with multiple waiting threads is even worse. The time to execute a critical section protected by a spinlock increases linearly with the number of spinning processors. Figure 6.4 illustrates this effect. The problem is that before a processor can execute an atomic read-modify-write instruction, the hardware must obtain exclusive access to that memory location. Any other read-modify-write instruction must occur either before or afterwards.

Thus, if a number of processors are executing a spin loop, they will all be trying to gain exclusive access to the memory location of the lock. The store instruction to clear the lock also needs exclusive access, and the hardware has no way to know that it should prioritize the lock release ahead of the competing requests to see if the lock is free.

One might think that it would help to check that the lock is free before trying to acquire it with a test-and-set; this is called *test and test-and-set*:

```
void Counter::Increment() {
    while (lock == BUSY || test_and_set(&lock)) // while BUSY
        ; // spin
    value++;
    lock = FREE;
    memory_barrier();
}
```

However, it turns out this does not help. When the lock is released, the new value of the lock, FREE, must be communicated to the other waiting processors. On modern systems, each processor separately fetches the data into its cache. Eventually one of them gets the new value and acquires the lock. If the critical section is not very long, the other processors will still be busy fetching the new value and trying to acquire the lock, preventing the lock release from completing.

One approach is to adjust the frequency of polling to the length of time that the thread has been waiting. A more scalable solution is to assign each waiting thread a separate memory location where it can spin. To release a lock, the bit is set for *one* thread, telling it that it is the next to acquire the lock.

The most widely used implementation of this idea is known as the MCS lock, after the initials of its authors, Mellor-Crummey and Scott. The MCS lock takes advantage of an atomic read-modify-write instruction called *compare-and-swap* that is supported on most modern multiprocessor architectures. Compare-and-swap tests the value of a memory location and swaps in a new value if the old value has not changed.

```
class MCSLock {
  private:
    Queue *tail = NULL;
}

MCSLock::release() {

    if (compare_and_swap(&tail,
                    myTCB, NULL)) {

        // If tail == myTCB, no one is
        // waiting. MCSLock is now free.

    } else {
        // Someone is waiting.
        while (myTCB->next == NULL)
            ; // spin until next is set

        // Tell next thread to proceed.
        myTCB->next->needToWait = FALSE;
    }
}
```

```
MCSLock::acquire() {
    Queue *oldTail = tail;

    myTCB->next = NULL;
    while (!compare_and_swap(&tail,
                  oldTail, &myTCB)) {

        // Try again if someone else
        // changed tail in the meantime.

        oldTail = tail;
    }

    // If oldTail == NULL, lock acquired.
    if (oldTail != NULL) {
        // Need to wait.
        myTCB->needToWait = TRUE;
        memory_barrier();
        oldTail->next = myTCB;
        while (myTCB->needToWait)
            ; //spin
    }
}
```

Figure 6.5: Pseudo-code for an MCS queueing lock, where each waiting thread spins on a separate memory location in its thread control block (myTCB). The operation, compare-and-swap, atomically inserts the TCB at the tail of the queue.

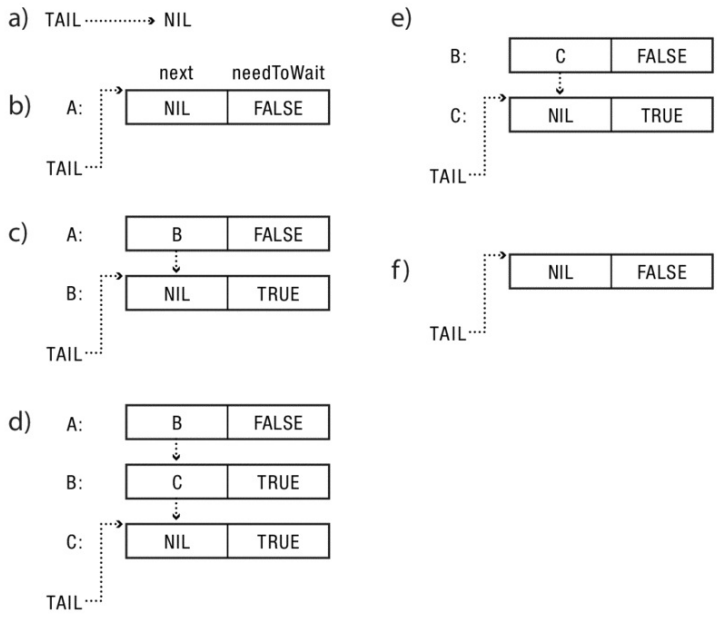

Figure 6.6: The behavior of the MCS queueing lock. Initially (a), tail is NULL indicating that the lock is FREE. To acquire the lock (b), thread A atomically sets tail to point to A's TCB. Additional threads B and C queue by adding themselves (atomically) to the tail (c) and (d); they then spin on their respective TCB's needToWait flag. Thread A hands the lock to B by clearing B's needToWait flag (e); B hands the lock to C by clearing C's needToWait fla (f). C releases the lock by setting tail back to NULL (a) iff no one else is waiting — that is, iff tail still points to C's TCB.

Compare-and-swap can be used to build a queue of waiting threads, without a separate spinlock. A waiting thread atomically adds itself to the *tail* of the queue, and then spins on a flag in its queue entry. When a thread releases the lock, it sets the flag in the next queue entry, signaling to the thread that its turn is next. Figure 6.5 provides an implementation, and Figure 6.6 illustrates the algorithm in action.

Because each thread in the queue spins on its own queue entry, the lock can be passed efficiently from one thread to another along the queue. Of course, the overhead of setting up the queue means that an MCS lock is less efficient than a normal spinlock unless there are a large number of waiting threads.

6.3.2 Read-Copy-Update (RCU)

Read-copy-update (RCU) provides high-performance synchronization for data structures that are frequently read and occasionally updated. In particular, RCU optimizes the read path to have extremely low synchronization costs even with a large number of concurrent readers. However, writes can be delayed for a long time — tens of milliseconds in some

implementations.

Why Not Use a Readers/Writers Lock?

Standard readers/writers locks are a poor fit for certain types of read-dominated workloads. Recall that these locks allow an arbitrary number of concurrent active readers, but when there is an active writer, no other writer or reader can be active.

The problem occurs when there are many concurrent reads with short critical sections. Before reading, each reader must acquire a readers/writers lock in read mode and release it afterwards. On both entrance and exit, the reader must update some state in the readers/writers synchronization object. Even when there are only readers, the readers/writers synchronization object can become a bottleneck. This limits the rate at which readers can enter the critical section, because they can only acquire the lock one at a time. For critical sections of less than a few thousand cycles, and for programs with dozens of threads simultaneously reading a shared object, the standard readers/writers lock can limit performance.

While the readers/writers synchronization object could be implemented with an MCS lock and thereby reduce some of the effects of lock contention, it does not change the inherent serial access of the readers/writers control structure.

The RCU Approach

How can concurrent reads access a data structure — one that can also be written — without having to update the state of a synchronization variable on each read?

To meet this challenge, an RCU lock retains the basic structure of a reader/writers lock: readers (and writers) surround each critical section with calls to acquire and release the RCU lock in read-mode (or write-mode). An RCU lock makes three important changes to the standard interface:

1. **Restricted update.** With RCU, the writer thread must *publish* its changes to the shared data structure with a single, atomic memory write. Typically, this is done by updating a single pointer, as we illustrate below by using RCU to update a shared list.

   Although restricted updates might seem to severely limit the types of data structure operations that are possible under RCU, this is not the case. A common pattern is for the writer thread to make a *copy* of a complex data structure (or a portion of it), update the copy, and then publish a pointer to the copy into a shared location where it can be accessed by new readers.

2. **Multiple concurrent versions.** RCU allows any number of read-only critical sections to be in progress at the same time as the update. These read-only critical sections may see the old or new version of the data structure.

3. **Integration with the thread scheduler.** Because there may be readers still in progress when an update is made, the shared object must maintain multiple versions of its state, to guarantee that an old version is not freed until all readers have finished accessing it. The time from when an update is published until the last reader is done with the previous version is called the *grace period*. The RCU lock uses information provided by the thread scheduler to determine when a grace period ends.

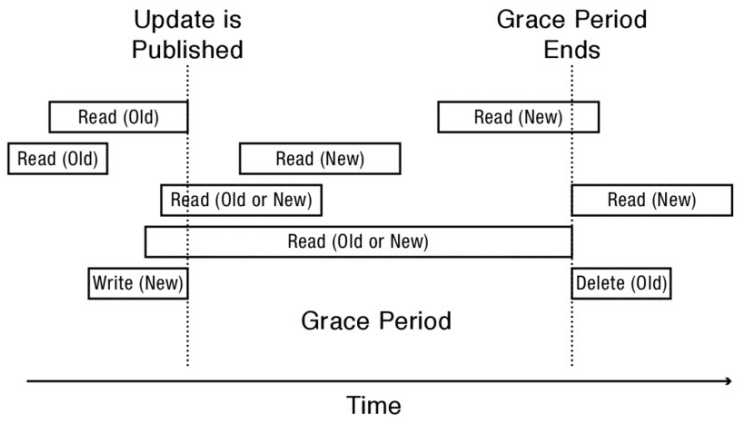

Figure 6.7: Timeline for an update concurrent with several reads for a data structure accessed with read-copy-update (RCU) synchronization.

Figure 6.7 shows the timeline for the critical sections of a writer and several reader threads under RCU. If a function that reads the data structure completes before a write is published, it sees the old version of the data structure; if a reader begins after a write is published, it sees the new version. But, if a reader begins *before* and ends *after* a write is published, it may see either the old version or the new one. If it reads the updated pointer more than once, it may see the old one and then the new one. Which version it sees depends on which version of the single, atomically-updated memory location it observes. However, the system guarantees that the old version is not deleted until the grace period expires. The deletion of the old version must be delayed until all reads that might observe the old version have completed.

RCU API and Use

RCU is a synchronization abstraction that allows concurrent access to a data structure by multiple readers and a single writer at a time. Figure 6.8 shows a typical API.

Reader API

readLock()	Enter read-only critical section.
readUnlock()	Exit read-only critical section.

Writer API

writeLock()	Enter read-write critical section.
publish()	Atomically update shared data structure.
writeUnlock()	Exit read-write critical section.
synchronize()	Wait for all currently active readers to exit critical section, to allow for garbage collection of old versions of the object.

Scheduler API

quiescentState()	Of the read-only threads on this processor who were active during the most recent RCU::publish, all have exited the critical section.

Figure 6.8: Sample programming interface for read-copy-update (RCU) synchronization. In Java's implementation of RCU locks, synchronize and quiescentState are not needed because the language-level garbage collector automatically detects when old versions can no longer be accessed. In the implementation of RCU in the Linux kernel, synchronize is split into two calls: one to start the grace period, and one to wait until the grace period completes.

A reader calls RCU::readLock and RCU::readUnlock before and after accessing the shared data structure. A writer calls: RCU::writeLock to exclude other writers; RCU::publish to issue the write that atomically updates the data structure so that reads can see the updates; RCU::writeUnlock to let other writers proceed; and RCU::synchronize to wait for the grace period to expire so that the old version of the object can be freed.

As Figure 6.9 illustrates, writes are serialized — only one write can proceed at a time. However, a write can be concurrent with any number of reads. A write can also be concurrent with another write's grace period: there may be any number of versions of the object until multiple overlapping grace periods expire.

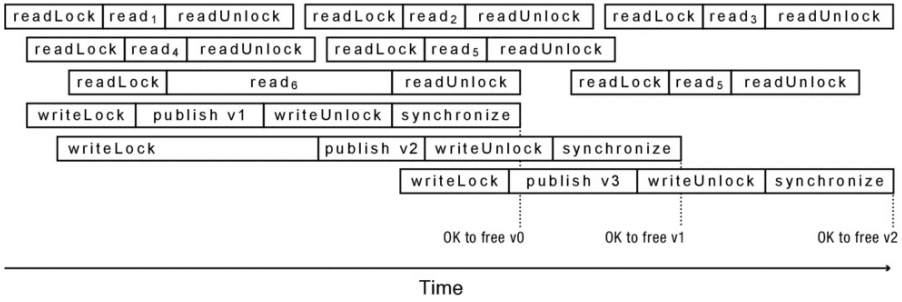

Figure 6.9: RCU allows one write at a time, and it allows reads to overlap each other and writes. The initial version is v0, and overlapping writes update the version to v1, v2, and then v3.

EXAMPLE: For each read in Figure 6.9, which version(s) of the shared state can the read observe?

ANSWER: If a read overlaps a publish, it can return the published value or the previous value:

Read	Value Returned	Reason
$read_1$	v0 or v1	Overlaps publish v1.
$read_2$	v2	After publish v2, before publish v3.
$read_3$	v3	After publish v3.
$read_4$	v0 or v1	Overlaps publish v1.
$read_5$	v1 or v2	Overlaps publish v2.
$read_6$	v0, v1, or v2	Overlaps publish v1 and v2.
$read_7$	v3	After publish v2.

□

```
typedef struct ElementS{
  int key;
```

```
  int value;
  struct ElementS *next;
} Element;

class RCUList {
  private:
    RCULock rcuLock;
    Element *head;
  public:
    bool search(int key, int *value);
    void insert(Element *item, value);
    bool remove(int key);
};

bool
RCUList::search(int key, int *valuep) {
    bool result = FALSE;
    Element *current;

    rcuLock.readLock();
    current = head;
    for (current = head; current != NULL;
                current = current->next) {
        if (current->key == key) {
            *valuep = current->value;
            result = TRUE;
            break;
        }
    }
    rcuLock.readUnlock();
    return result;
}
```

Figure 6.10: Declaration of data structures and API for a linked list that uses RCU for synchronization, and the implementation of a read-only method for searching the linked list using RCU.

EXAMPLE: RCU linked list. Figures 6.10 and 6.11 show how to use RCU locks to implement a linked list that can be accessed concurrently by many readers, while also being updated by one writer.

The list data structure comprises an RCU lock and a pointer to the head of the list. Each entry in the list has two data fields — key and value — as well as a pointer to the next record on the list.

The search method is read-only: after registering with readLock, it scans down the list until it finds an element with a matching key. If the element is found, the method uses the parameter to return the value field and then returns TRUE. Otherwise, the method returns FALSE to indicate that no matching record was found.

The methods to update the list are more subtle. Each of them is arranged so that a single pointer update is sufficient to publish the new version of the list to the readers. In particular, it is important that insert initialize the data structure *before* updating the head pointer to make the new element visible to readers.

```
void
RCUList::insert(int key,
                    int value) {
    Element *item;

    // One write at a time.
    rcuLock.writeLock();

    // Initialize item.
    item = (Element*)
         malloc(sizeof(Element));
    item->key = key;
    item->value = value;
    item->next = head;

    // Atomically update list.
    rcuLock.publish(&head, item);

    // Allow other writes
    // to proceed.
    rcuLock.writeUnlock();

    // Wait until no reader
    // has old version.
    rcuLock.synchronize();
}

bool
RCUList::remove(int key) {
    bool found = FALSE;
    Element *prev, *current;

    // One write at a time.
    rcuLock.WriteLock();
    for (prev = NULL, current = head;
            current != NULL; prev = current,
            current = current->next) {
        if (current->key == key) {
            found = TRUE;

            // Publish update to readers
            if (prev == NULL) {
                rcuLock.publish(&head,
                            current->next);
            } else {
                rcuLock.publish(&(prev->next),
                            current->next);
            }
            break;
        }
    }

    // Allow other writes to proceed.
    rcuLock.writeUnlock();

    // Wait until no reader has old version.
    if (found) {
        rcuLock.synchronize();
```

```
        free(current);
    }
    return found;
}
```

Figure 6.11: Implementation of a linked list using RCU for synchronization.

Implementing RCU

When implementing RCU, the central goal is to minimize the cost of read critical sections: the system must allow an arbitrary number of concurrent readers. Conversely, writes can have high *latency*. In particular, grace periods can be long, with tens of milliseconds from when an update is published until the system can guarantee that no readers are still using the old version. Even so, write *overhead* — the CPU time needed per write — should be modest.

A common technique for achieving these goals is to integrate the RCU implementation with that of the thread scheduler. This is in contrast with the readers/writers lock described in the previous chapter, which makes no assumptions about the thread scheduler, but which must track exactly how many readers are active at any given time.

In particular, the implementation we present requires two things from the scheduler: (1) read-only critical sections complete without being interrupted and (2) whenever a thread on a processor is interrupted, the scheduler updates some per-processor RCU state. Then, once a write completes, RCULock::Synchronize simply waits for all processors to be interrupted at least once. At that point, the old version of the object is known to be *quiescent* — no thread has access to the old version (other than the writer who changed it).

```
class RCULock{
  private:
  // Global state
    Spinlock globalSpin;
    long globalCounter;
  // One per processor
    DEFINE_PER_PROCESSOR(
       static long, quiescentCount);

  // Per-lock state
    Spinlock writerSpin;

  // Public API omitted
}

void RCULock::ReadLock() {
    disableInterrupts();
}

void RCULock::ReadUnlock() {
    enableInterrupts();
}

// Called by scheduler
void RCULock::QuiescentState(){
    memory_barrier();
    PER_PROC_VAR(quiescentCount) =
                    globalCounter;
    memory_barrier();
}

void RCULock::writeLock() {
    writerSpin.acquire();
}

void RCULock::writeUnlock() {
    writerSpin.release();
}

void RCULock::publish (void **pp1,
                       void *p2){
    memory_barrier();
    *pp1 = p2;
    memory_barrier();
}

void
RCULock::synchronize() {
    int p, c;

    globalSpin.acquire();
    c = ++globalCounter;
    globalSpin.release();

    FOREACH_PROCESSOR(p) {
        while((PER_PROC_VAR(
          quiescentCount, p) - c) < 0) {
            // release CPU for 10ms
            sleep(10);
        }
    }
}
```

Figure 6.12: A quiescence-based RCU implementation. The code assumes that spinlock acquire/release and interrupt enable/disable trigger a memory barrier. *Credit:* This pseudo-code is based on an implementation by Paul McKenney in “Is Parallel Programming Hard, And, If So, What Can be Done About It?”

Figure 6.12 shows an implementation of RCU based on quiescent states. Notice first that readLock and readUnlock are inexpensive: they update no state and merely ensure that the read is not interrupted. RCU::writeLock and writeUnlock are also inexpensive. They acquire and release a spinlock to ensure that at most one write per RCULock can proceed at a time.

RCU::publish is also simple. It executes a memory barrier so that all modifications to the shared object are completed before the pointer is updated. It then updates the pointer, and then executes another memory barrier so that other processors observe the update.

RCU::synchronize and quiescentState work together to ensure that when synchronize

returns, all threads are guaranteed to be done with the old version of the object. RCU::synchronize increments a global counter and then waits until all processors' match the new value of that counter. RCU::quiescentState is called by the scheduler whenever that processor is interrupted. It updates that processor's quiescentCount to match the current globalCounter. Thus, once quiescentCount is at least as large as c, on every processor, synchronize knows that no remaining readers can observe the old version.

6.4 Multi-Object Atomicity

Once a program has multiple shared objects, it becomes both necessary and challenging to reason about interactions across objects. For example, consider a system storing a bank's accounts. A reasonable design choice might be for each customer's account to be a shared object with a lock (either a mutual exclusion lock or a readers/writers lock, as described in Chapter 5). Consider, however, transferring $100 from account A to account B, as follows:

```
A->subtract(100);
B->add(100);
```

Although each individual action is atomic, the sequence of actions is not. As a result, there may be a time where, say, A tells B that the money has been sent, but the money is not yet in B's account.

Similarly, consider a bank manager who wants to answer a question: "How much money does the bank have?" If the manager's program simply reads from each account, the calculation may exclude or double-count money "in flight" between accounts, such as in the transfer from A to B.

These examples illustrate a general problem that arises whenever a program contains multiple shared objects. Even if the object guarantees that each method operates atomically, *sequences* of operations by different threads can be interleaved. The same issues of managing multiple locks also apply to fine-grained locking within an object.

6.4.1 Careful Class Design

Sometimes it is possible to address this issue through careful class and interface design. This includes the design of individual objects (e.g., specifying clean interfaces that expose the right abstractions). It also includes the architecture of how those objects interact (e.g., structuring a system architecture in well-defined layers).

For example, you would face the same issues if you tried to solve Too Much Milk problem with a Note object that has two methods, readNote and writeNote, and a Fridge object with two methods, checkForMilk and addMilk. Atomicity of these individual operations is not sufficient to provide the desired behavior without considerable programming effort.

On the other hand, if we refactor the objects so that we have:

```
Fridge::checkforMilkAndSetNoteIfNeeded();
Fridge::addMilk();
```

Then, the problem becomes straightforward.

This advice may seem obvious: of course, you should strive for elegant designs for both single- and multi-threaded code. Nonetheless, we emphasize that the choices you make for your interfaces, abstractions, and software architecture can dramatically affect the complexity or feasibility of your designs.

6.4.2 Acquire-All/Release-All

Better interface design has limits, however. Sometimes, multiple locks are needed for program structure or for greater concurrency. Is there a general technique to perform a set of operations that require multiple locks, so that the group of operations appears atomic? For clarity, we will refer to a group of operations as a *request*.

One approach, called *acquire-all/release-all* is to acquire *every* lock that may be needed at any point while processing the entire set of operations in the request. Then, once the thread has all of the locks it might need, the thread can execute the request, and finally, release the locks.

EXAMPLE: Consider a hash table with one lock per hash bucket. To move an item from one bucket to another, the hash table supports a changeKey(item, k1, k2) operation. With acquire-all/release-all, this function could be implemented to first acquire both the locks for k1 and k2, then remove the item under k1 and insert it under k2, and finally release both locks.

Acquire-all/release-all allows significant concurrency. When individual requests touch non-overlapping subsets of state protected by different locks, they can proceed in parallel.

A key property of this approach is *serializability* across requests: the result of any program execution is equivalent to an execution in which requests are processed one at a time in some sequential order. Serializability allows one to reason about multi-step tasks *as if* each task executed alone.

As Figure 6.13 illustrates, requests that access non-overlapping data can proceed in parallel. The result is the same as if the system first executed one request and then the other (or equivalently, the reverse). On the other hand, if two requests touch the same data, then the fact that all locks are acquired at the beginning and released at the end implies that one request is completed before the other one begins.

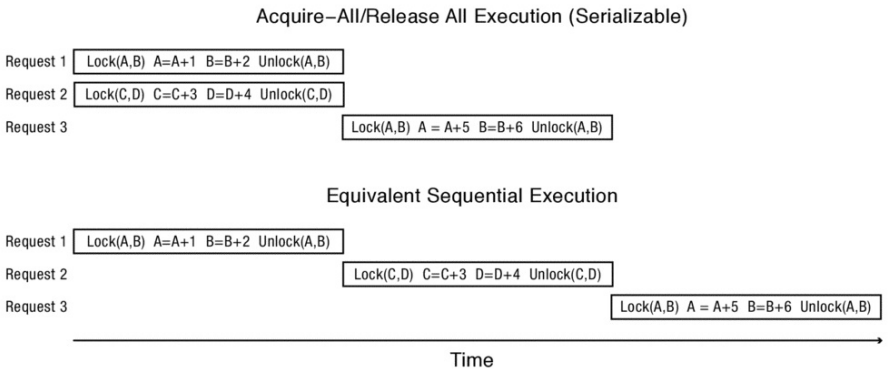

Figure 6.13: Locking multiple objects using an acquire-all/release-all pattern results in a serializable execution that is equivalent to an execution where requests are executed sequentially in some order.

One challenge to using this approach is knowing exactly what locks will be needed by a request before beginning to process it. A potential solution is to conservatively acquire more locks than needed (e.g., acquire any locks that *may* be needed by a particular request), but this may be difficult to determine. Without first executing the request, how can we know which locks will be needed?

6.4.3 Two-Phase Locking

Two phase locking refines the acquire-all/release-all pattern to address this concern. Instead of acquiring all locks before processing the request, locks can be acquired as needed for each operation. However, locks are not *released* until all locks needed by the request have been acquired. Most implementations simply release all locks at the end of the request.

Two-phase locking avoids needing to know what locks to grab *a priori*. Therefore, programs can avoid acquiring locks they do not need, and they may not need to hold locks as long.

EXAMPLE: The changeKey(item, k1, k2) function for a hash table with per-bucket locks could be implemented to acquire k1's lock, remove the item using key k1, acquire k2's lock, insert the item using key k2, and release both locks.

Like acquire-all/release-all, two-phase locking is serializable. If two requests have non-overlapping data, they are commutative and therefore serializable. Otherwise, there is some overlapping data between the two requests, protected by one or more locks. Provided a request completes, it must have acquired all of those locks, and made its changes to the overlapping data, before releasing any of them. Thus, any overlapping request must have read or modified the data in the overlap either entirely before or after the other request.

Unlike acquire-all/release-all, however, two-phase locking can in some cases lead to deadlock, the topic of the next section.

EXAMPLE: Suppose one thread starts executing changeKey(item, k1, k2) and another thread simultaneously tries to move a different item in the other direction from k2 to k1. If the first thread acquires k1's lock and the second thread acquires k2's lock, neither will be able to make progress.

6.5 Deadlock

A challenge to constructing complex multi-threaded programs is the possibility of deadlock. A *deadlock* is a cycle of waiting among a set of threads, where each thread waits for some other thread in the cycle to take some action.

Deadlock can occur in many different situations, but one of the simplest is *mutually recursive locking*, shown in the code fragment below:

```
// Thread A

lock1.acquire();
lock2.acquire();
lock2.release();
lock1.release();

// Thread B

lock2.acquire();
lock1.acquire();
lock1.release();
lock2.release();
```

Suppose two shared objects with mutual exclusion locks can call into each other while holding their locks. Deadlock can occur when one thread holds the lock on the first object, and another thread holds the lock on the second object. If the first thread calls into the second object while still holding onto its lock, it will need to wait for the second object's lock. If the other thread does the same thing in reverse, neither will be able to make progress.

We can also get into deadlock with two locks and a condition variable, shown below:

```
// Thread A

lock1.acquire();
...
lock2.acquire();
while (need to wait) {
    cv.wait(&lock2);
}
...
lock2.release();
```

```
...
lock1.release();

// Thread B

lock1.acquire();
...
lock2.acquire();
...
cv.signal();
lock2.release();
...
lock1.release();
```

In *nested waiting*, one shared object calls into another shared object while holding the first object's lock, and then waits on a condition variable. CV::wait releases the lock of the second object, but not the first. Deadlock results if the thread that can signal the condition variable needs the first lock to make progress.

The problem of deadlock is much broader than just locks and condition variables. Deadlock can occur anytime a thread waits for an event that cannot happen because of a cycle of waiting for a resource held by the first thread. As in the examples above, resources can be locks, but they can also be any other scarce quantity: memory, processing time, disk blocks, or space in a buffer.

Suppose we have two bounded buffers, where one thread puts a request into one buffer, and gets a response out of the other. Deadlock can result if another thread does the reverse.

```
// Thread A

buffer1.put();
buffer1.put();
...
buffer2.get();
buffer2.get();

// Thread B

buffer2.put();
buffer2.put();
...
buffer1.get();
buffer1.get();
```

If the buffers are almost full, both threads will need to wait for there to be room, and so neither will be able to reach the point where they can pull data out of the other buffer to allow the other thread to make progress.

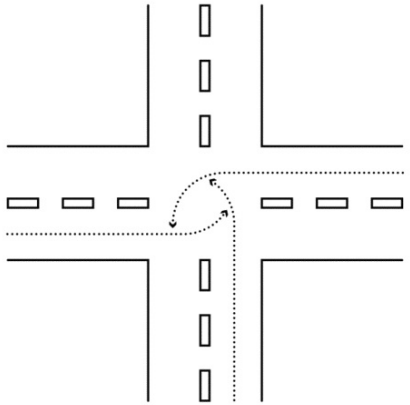

Figure 6.14: An example of deadlock where three tractor-trailer trucks enter an intersection without first checking whether they can clear the intersection.

Deadlocks also occur in real life. We encourage you to develop your intuition about deadlocks by considering why deadlocks occur and how we might prevent them. For example, if we lived in a world without stop signs, we might see the deadlock in Figure 6.14 more often.

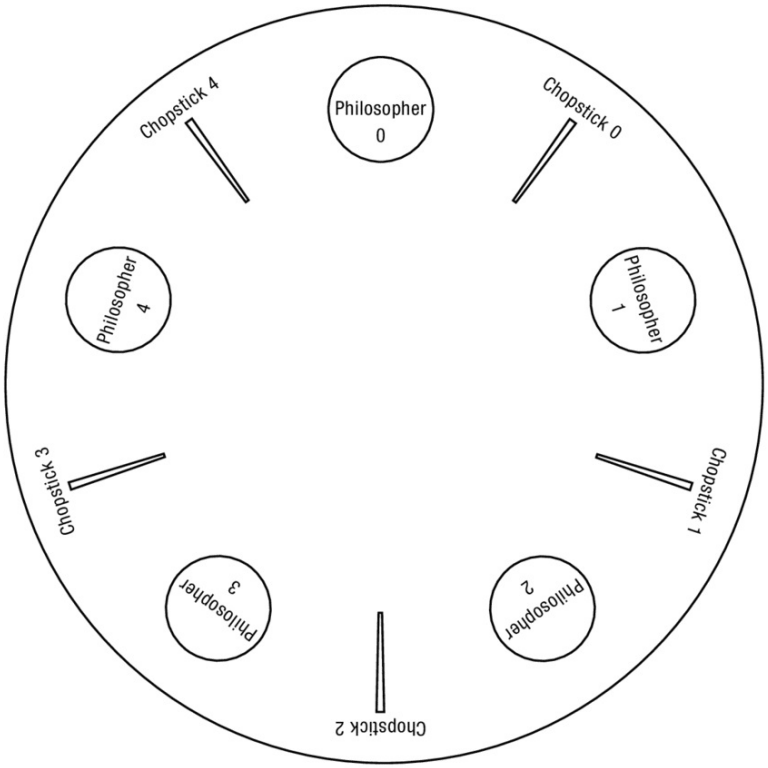

Figure 6.15: In this example of the dining philosophers problem, there are 5 philosophers, 5 plates, and 5 chopsticks.

The scarce resource leading to deadlock can even be a chopstick. The Dining Philosophers problem is a classic illustration of both the challenges and solutions to deadlock; an example is shown in Figure 6.15. There is a round table with n plates alternating with n chopsticks around the circle. A philosopher sitting at a plate requires two chopsticks to eat. Suppose that each philosopher proceeds by picking up the chopstick on the left, picking up the chopstick on the right, eating, and then putting down both chopsticks. If every philosopher follows this approach, there can be a deadlock: each philosopher takes the chopstick on the left but can be stuck waiting for the philosopher on the right to release the chopstick.

Note that mutually recursive locking is equivalent to Dining Philosophers with n = 2.

The rest of this section addresses the following questions:

- **Deadlock vs. Starvation.** How does deadlock relate to the concepts of liveness and starvation?
- **Necessary Conditions for Deadlock.** What conditions are required for deadlock to be possible?
- **Preventing Deadlock.** What techniques can be used to prevent deadlock?
- **The Banker's Algorithm for Avoiding Deadlock.** The Banker's Algorithm is a general-purpose mechanism for preventing deadlock by exploiting knowledge of what resources may be needed in the future.
- **Detecting and Recovering From Deadlock.** In some systems, deadlock is not prevented but repaired when it occurs. How can we detect deadlock and then recover?

6.5.1 Deadlock vs. Starvation

Deadlock and starvation are both liveness concerns. In *starvation*, a thread fails to make progress for an indefinite period of time. Deadlock is a form of starvation but with the stronger condition: a group of threads forms a cycle where none of the threads make progress because each thread is waiting for some other thread in the cycle to take action. Thus, deadlock implies starvation (literally, for the dining philosophers), but starvation does not imply deadlock.

For example, recall the readers/writers example discussed in Section 5.6.1. A writer only waits if a reader or writer is active. In the writers-preferred solution we gave, waiting readers can starve if new writers arrive sufficiently frequently; likewise, waiting writers can starve if there is an active reader, and new readers arrive and become active before the last one completes. Note that such starvation would not be deadlock because there is no cycle. The waiting readers are waiting on the active writers to finish, and the waiting writers are waiting on the active readers to finish, but no active thread is waiting on a waiting reader or writer.

Just because a system can suffer deadlock or starvation does not mean that it always will. A system is *subject to starvation* if a thread could starve in some circumstances. A system is *subject to deadlock* if a group of threads could deadlock in some circumstances. Here, the circumstances that affect whether deadlock or starvation occurs could include a broad range of factors, such as: the choices made by the scheduler, the number of threads running, the workload or sequence of requests processed by the system, which threads win races to acquire locks, and which threads are enabled in what order when signals or broadcasts occur.

A system that is subject to starvation or deadlock may be live in many or most runs and starve or deadlock only for particular workloads or "unlucky" interleavings. For example, in mutually recursive locking, the deadlock only occurs if both threads obtain the outer locks at about the same time. For the Dining Philosophers problem, philosophers may succeed in eating for a long time before hitting the unlucky sequence of events that causes them to deadlock. Similarly, in the readers/writers example, the writers-preferred solution will allow some reads to complete as long as the rate of writes stays below some threshold.

Since testing may not discover deadlock problems, it is important to construct systems that

are deadlock-free by design.

6.5.2 Necessary Conditions for Deadlock

There are four necessary conditions for deadlock to occur. Knowing these conditions is useful for designing solutions: if you can prevent any one of these conditions, then you can eliminate the possibility of deadlock.

1. **Bounded resources.** There are a finite number of threads that can simultaneously use a resource.
2. **No preemption.** Once a thread acquires a resource, its ownership cannot be revoked until the thread acts to release it.
3. **Wait while holding.** A thread holds one resource while waiting for another. This condition is sometimes called *multiple independent requests* because it occurs when a thread first acquires one resource and then tries to acquire another.
4. **Circular waiting.** There is a set of waiting threads such that each thread is waiting for a resource held by another.

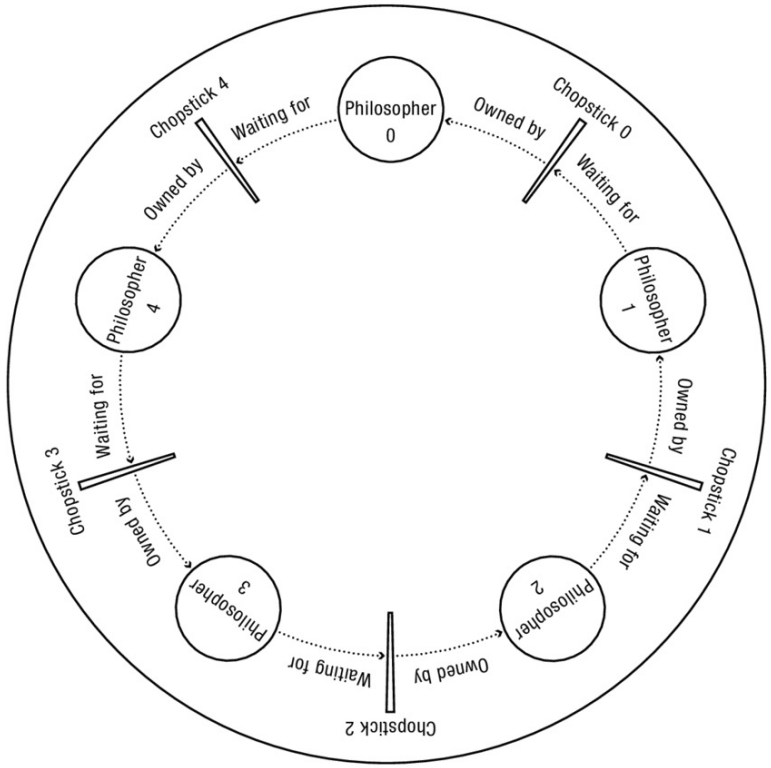

Figure 6.16: Graph representation of the state of a deadlocked Dining Philosophers system. Circles represent threads, boxes represent resources, an arrow from a box/resource to a circle/thread represents an *owned by* relationship, and an arrow from a circle/thread to a box/resource represents a *waiting for* relationship.

EXAMPLE: Show that the Dining Philosophers meet all four conditions for deadlock.

ANSWER: To see that all four conditions are met, observe that

1. **Bounded resources.** Each chopstick can be held by a single philosopher at a time.
2. **No preemption.** Once a philosopher picks up a chopstick, she does not release it until she is done eating, even if that means no one will ever eat.
3. **Wait while holding.** When a philosopher needs to wait for a chopstick, she continues to hold onto any chopsticks she has already picked up.
4. **Circular waiting.** Figure 6.16 maps the state of a deadlocked Dining Philosophers implementation to an abstract graph that shows which resources are *owned by* which threads and which threads *wait for* which resources. In this type of graph, if there is one instance of each type of resource (e.g., a particular chopstick), then a cycle

implies deadlock assuming the system does not allow preemption.

□

The four conditions are necessary *but not sufficient* for deadlock. When there are multiple instances of a type of resource, there can be a cycle of waiting without deadlock because a thread not in the cycle may return resources that enable a waiting thread to proceed.

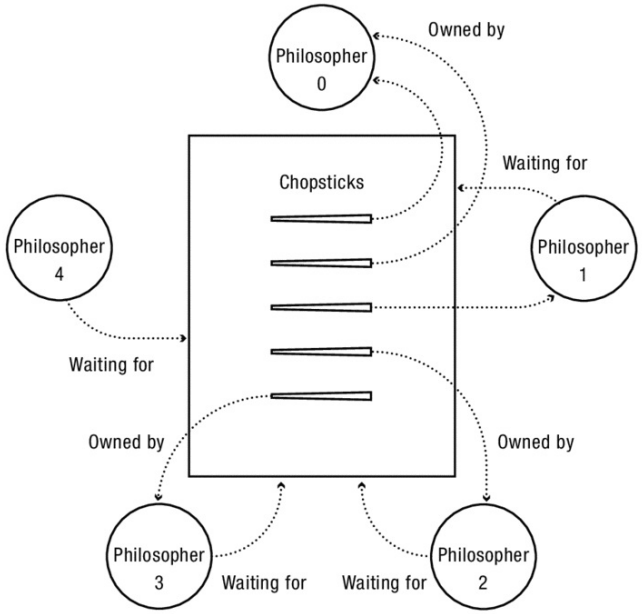

Figure 6.17: Graph representation of the state of a Dining Philosophers system that includes a cycle among waiting threads and resources but that is not deadlocked. Circles represent threads, boxes represent resources, dots within a box represent multiple instances of a resource, an arrow from a dot/resource instance to a circle/thread represents an *owned by* relationship and an arrow from a circle/thread to a box/resource represents a *waiting for* relationship.

Suppose we have 5 philosophers at a table with 5 chopsticks, but the chopsticks are placed in a tray at the center of the table when not in use. We could be in the state illustrated in Figure 6.17, where philosopher 1 has two chopsticks, philosophers 2, 3, and 4 each have one chopstick and are waiting for another chopstick, while philosopher 5 has no chopsticks. In this state, we have bounded resources (five chopsticks), no preemption (we cannot forcibly remove a chopstick from a hungry philosopher's hand), wait while holding (philosophers 2, 3 and 4 are holding a chopstick while waiting for another), and circular waiting (each of philosophers 2, 3, and 4 are waiting for a resource held by another of them). However, we do not have deadlock. Eventually, philosopher 1 will release its two chopsticks, which may, for example, allow philosophers 2 and 3 to eat and release their chopsticks. In turn, this would allow philosophers 4 and 5 to eat.

Although the system shown in Figure 6.17 is not currently deadlocked, it is still *subject to deadlock*. For example, if philosopher 1 returns two chopsticks, philosopher 5 picks up one, and philosopher 1 picks up the other, then the system would deadlock.

6.5.3 Preventing Deadlock

Preventing deadlock can be challenging. For example, consider a system with three resources — A, B, and C — and two threads that access them. Thread 1 acquires A then C then B, and thread 2 acquires B then C then A. The following sequence can lead to deadlock:

	Thread 1	Thread 2
1	Acquire A	
2		Acquire B
3	Acquire C	
4		Wait for C
5	Wait for B	

How could we avoid this deadlock? The deadlock's circular waiting occurs when we reach step 5, but our fate was sealed much earlier. In particular, once we complete step 2 and thread 2 acquires B, deadlock is inevitable. To prevent the deadlock, we have to realize at step 2 that it will occur at step 5. Once step 1 completes and thread 1 acquires A, we cannot let thread 2 complete step 2 and acquire B or deadlock will follow.

This example illustrates that for an arbitrary program, preventing deadlock can take one of three approaches:

1. **Exploit or limit the behavior of the program.** Often, we can change the behavior of a program to prevent one of the four necessary conditions for deadlock, and thereby eliminate the possibility of deadlock. In the above example, we can eliminate deadlock by changing the program to never wait for B while holding C.
2. **Predict the future.** If we can know what threads may or will do, then we can avoid deadlock by having threads wait (e.g., thread 2 can wait at step 2 above) *before* they would head into a possible deadlock.
3. **Detect and recover.** Another alternative is to allow threads to recover or "undo"

actions that take a system into a deadlock; in the above example, when thread 2 finds itself in deadlock, it can recover by reverting to an earlier state.

We discuss these three options in this and the following two sub-sections.

Section 6.5.2 listed four necessary conditions for deadlock. These conditions are useful because they suggest approaches for preventing deadlock: if a system is structured to prevent at least one of the conditions, then the system cannot deadlock. Considering these conditions in the context of a given system often points to a viable deadlock prevention strategy. Below, we discuss some commonly used approaches.

Bounded resources: Provide sufficient resources. One way to ensure deadlock freedom is to arrange for sufficient resources to satisfy all threads' demands. A simple example would be to add a single chopstick to the middle of the table in Dining Philosophers; that is enough to eliminate the possibility of deadlock. As another example, thread implementations often reserve space in the TCB for the thread to be inserted into a waiting list or the ready list. While it would be theoretically possible to dynamically allocate space for the list entry only when it is needed, that could open up the chance that the system would run out of memory at exactly the wrong time, leading to deadlock.

No preemption: Preempt resources. Another technique is to allow the runtime system to forcibly reclaim resources held by a thread. For example, an operating system can preempt a page of memory from a running process by copying it to disk in order to prevent applications from deadlocking as they acquire memory pages.

Wait while holding: Release lock when calling out of module. For nested modules, each of which has its own lock, waiting on a condition variable in an inner module can lead to a nested waiting deadlock. One solution is to restructure a module's code so that no locks are held when calling other modules. For example, we can change the code on the left to the code on the right, provided that the program does not depend on the three steps occurring atomically:

```
Module::foo() {
    lock.acquire();
    doSomeStuff();
    otherModule->bar();
    doOtherStuff();
    lock.release();
}

Module::doSomeStuff() {
    x = x + 1;
}

Module::doOtherStuff() {
    y = y - 2;
}
```

```
Module::foo() {
    doSomeStuff();
    otherModule->bar();
    doOtherStuff();
}

Module::doSomeStuff() {
    lock.acquire();
    x = x + 1;
    lock.release();
}

Module::doOtherStuff() {
    lock.acquire();
    y = y - 2;
    lock.release();
}
```

Deadlock and kernel paging

Early operating systems were often run on machines with very limited amounts of main memory. In response, going back at least as far as Multics, portions of the kernel (both code and data) could be swapped to disk in order to save space. Then, when the code and data was needed, they could be brought into main memory, swapping with some other portion of the kernel that was not currently in use.

A challenge to making this work was deadlock. The code to swap in or out portions of the kernel needed to be kept in memory, along with any code or data it might touch along any possible execution path. Without very strict module layering, it would be easy to miss a dependency that would, in rare cases, trigger a latent deadlock. Often, the only possible repair would be to reboot.

Because of the inherent complexity of this approach, most modern operating systems keep all kernel code and almost all data structures memory resident; the one exception is that some kernels still swap the page tables for application virtual memory, a topic we will discuss in Chapter 9.

In theory, one could eliminate the risk of deadlocks due to nested monitors by always releasing locks when calling code outside of a module. In practice, doing so is likely to be cumbersome, not only from the extra code needed to acquire and release locks, but also because of the extra thought needed to transform a single atomic method that holds a lock across a series of actions to a sequence of atomic methods that each acquire and release the lock. As a result, programmers often take the decidedly non-modular and admittedly unsatisfying approach of considering whether the outside module being called is likely to wait on something that depends on enclosing monitor lock. If such waiting is unlikely, the call can made with the enclosing lock held.

Circular waiting: Lock ordering. An approach used in many systems is to identify an ordering among locks and only acquire locks in that order.

For example, C printf acquires a lock to ensure printed messages appear atomic rather than mixed up with those of other threads. Because waiting for that lock does not lead to circular waiting, printf can be safely called while holding most kernel locks.

For a hash table with per-bucket locks and an operation changeKeys(item, k1, k2) to move an item from one bucket to another, we can avoid deadlock by always acquiring the lock for the lower-numbered bucket before the one for the higher-numbered bucket. This prevents circular waiting since a thread only waits for threads holding higher-numbered locks. Those threads can be waiting as well, but only for threads with even higher-numbered locks, and so forth.

Likewise, we can eliminate deadlock among the dining philosophers if — instead of always picking up the chopstick on the left and then the one on the right — the philosophers number the chopsticks from 1 to n and always pick up the lower-numbered chopstick before the higher-numbered one.

6.5.4 The Banker's Algorithm for Avoiding Deadlock

A general technique to eliminate wait-while-holding is to wait until all needed resources are available and then to acquire them atomically at the beginning of an operation, rather than incrementally as the operation proceeds. We saw this earlier with acquire-all/release-all; it cannot deadlock as long as the implementation acquires all of the locks atomically rather than one at a time. As another example, a dining philosopher might wait until the two neighboring chopsticks are available and then simultaneously pick them both up.

Of course, a thread may not know exactly which resources it will need to complete its work, but it can still acquire all resources that it *might* need. Consider an operating system for mobile phones where memory is constrained and cannot be preempted by copying it to disk. Rather than having applications request additional memory as needed, we might instead have each application state its maximum memory needs and allocate that much memory when it starts.

Disadvantages of this approach include: the effect on program modularity, the challenge of having applications accurately estimate their worst-case needs, and the cost of allocating significantly more resources than may be necessary in the common case.

Dijkstra developed the Banker's Algorithm as a way to improve on the performance of acquire-all. Although few systems use it in its full generality, we include the discussion because simplified versions of the algorithm are common. The Banker's Algorithm also sheds light on the distinction between *safe* and *unsafe* states and how the occurrence of deadlocks often depends on a system's workload and sequence of operations.

In the Banker's Algorithm, a thread states its maximum resource requirements when it begins a task, but it then acquires and releases those resources incrementally as the task runs. The runtime system delays granting some requests to ensure that the system never deadlocks.

The insight behind the algorithm is that a system that may deadlock will not necessarily do so: for some interleavings of requests it will deadlock, but for others it will not. By delaying when some resource requests are processed, a system can avoid interleavings that could lead to deadlock.

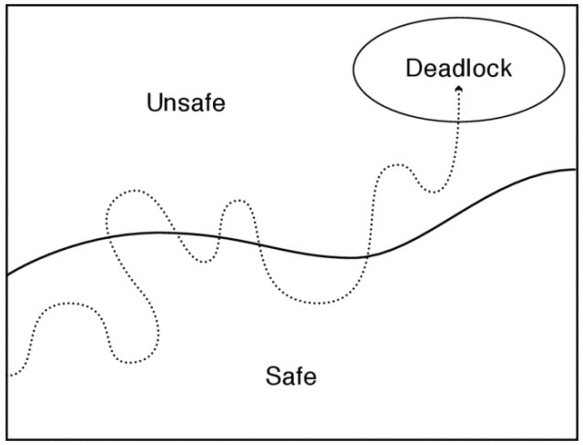

Figure 6.18: A process can be in a *safe*, *unsafe*, or *deadlocked* state. The dashed line illustrates a sequence of states visited by a thread — some are safe, some are unsafe, and the final state is a deadlock.

A deadlock-prone system can be in one of three states: a *safe state*, an *unsafe state*, and a *deadlocked state* (see Figure 6.18.)

- In a *safe state*, for any possible sequence of resource requests, there is at least one *safe sequence* of processing the requests that eventually succeeds in granting all pending and future requests.
- In an *unsafe state*, there is at least one sequence of future resource requests that leads to deadlock no matter what processing order is tried.
- In a *deadlocked state*, the system has at least one deadlock.

A system in a safe state controls its own destiny: for any workload, it can avoid deadlock by delaying the processing of some requests. In particular, the Banker's Algorithm delays any request that takes it from a safe to an unsafe state. Once the system enters an unsafe state, it may not be able to avoid deadlock.

Notice that an unsafe state does not always lead to deadlock. A system in an unsafe state may remain that way or return to a safe state, depending on the specific interleaving of resource requests and completions. However, as long as the system remains in an unsafe state, a bad workload or unlucky scheduling of requests can force it to deadlock.

The Banker's Algorithm keeps a system in a safe state. The algorithm is based on a loose analogy with a small-town banker who has a maximum amount, total, that can be loaned at one time and a set of businesses that each have a credit line, max[i], for business i. A business borrows and pays back amounts of money as various projects start and end, so that business i always has an outstanding loan amount between 0 and max[i]. If all of a business's requests within the credit line are granted, the business eventually reaches a

state where all current projects are finished, and the loan balance returns to zero.

A conservative banker might issue credit lines only until the sum is at most the total funds that the banker has available. This approach is analogous to *acquire-all* or *provide sufficient resources*. It guarantees that the system remains in a safe state. All businesses with credit lines eventually complete their projects.

However, a more aggressive banker can issue more credit as long as the bank can cover its commitment to each business — i.e., to provide a loan of max[i] if business i requests it. The algorithm assumes the bank is permitted to *delay* requests to increase a loan amount. For example, the bank might lose the paperwork for a few hours, days, or weeks.

By delaying loan requests, the bank remains in a safe state — a state for which there exists at least one series of loan fulfillments by which every business i can eventually receive its maximal loan max[i], complete its projects, and pay back all of its loan. The bank can then use that repaid money to grant pending loans to other businesses.

```
class ResourceMgr{
  private:
    Lock lock;
    CV cv;
    int r;          // Number of resources
    int t;          // Number of threads
    int avail[];    // avail[i]: instances of resource i available
    int max[][];    // max[i][j]: max of resource i needed by thread j
    int alloc[][];  // alloc[i][j]: current allocation of resource i to thread j
  ...
}
```

Figure 6.19: State maintained by the Banker Algorithm's resource manager. Resource manager code is in Figures 6.20 and 6.21.

```
// Invariant: the system is in a safe state.
ResourceMgr::Request(int resourceID, int threadID) {
    lock.Acquire();
    assert(isSafe());
    while (!wouldBeSafe(resourceID, threadID)) {
        cv.Wait(&lock);
    }
    alloc[resourceID][threadID]++;
    avail[resourceID]--;
    assert(isSafe());
    lock.Release();
}
```

Figure 6.20: High-level pseudo-code for the Banker's Algorithm. The state maintained by the algorithm is defined in Figure 6.19. The methods isSafe and wouldBeSafe are defined in Figure 6.21.

```
// A state is safe iff there exists a safe sequence of grants that are sufficien
// to allow all threads to eventually receive their maximum resource needs.
```

```
bool
ResourceMgr::isSafe() {
    int j;
    int toBeAvail[] = copy avail[];
    int need[][] = max[][] - alloc[][];  // need[i][j] is initialized to
                                         // max[i][j] - alloc[i][j]
    bool finish[] = [false, false, false, ...]; // finish[j] is true
                                         // if thread j is guaranteed to finish

    while (true) {
        j = any threadID such that:
              (finish[j] == false) && forall i: need[i][j] <= toBeAvail[i];
        if (no such j exists) {
            if (forall j: finish[j] == true) {
                return true;
            } else {
                return false;
            }
        } else {  // Thread j will eventually finish and return its
                  // current allocation to the pool.
            finish[j] = true;
            forall i:  toBeAvail[i] = toBeAvail[i] + alloc[i][j];
        }
    }
}

// Hypothetically grant request and see if resulting state is safe.

bool
ResourceMgr::wouldBeSafe(int resourceID, int threadID) {
    bool result = false;

    avail[resourceID]--;
    alloc[resourceID][threadID]++;
    if (isSafe()) {
        result = true;
    }
    avail[resourceID]++;
    alloc[resourceID][threadID]--;
    return result;
}
```

Figure 6.21: Pseudo-code for the Banker's Algorithm test whether the next state would be safe to enter. If not, the system delays until it would be safe.

Figure 6.20 shows pseudo-code for a version of the Banker's Algorithm that manages a set of r resources for a set of t threads. To simplify the discussion, threads request each unit of resource separately, but the algorithm can be extended to allow multiple resources to be requested at the same time.

The high-level idea is simple: when a request arrives, wait to grant the request until it is safe to do so. As Figure 6.19 shows, we can realize this high-level approach by tracking: (i) the current allocation of each resource to each thread, (ii) the maximum allocation possible for each thread, and (iii) the current set of available, unallocated resources.

Figure 6.21 shows how to test whether a state is safe. Recall that a state is safe if some sequence of thread executions allows each thread to obtain its maximum resource need, finish its work, and release its resources. We first see if the currently free resources suffice to allow any thread to finish. If so, then the resources held by that thread will eventually be released back to the system. Next, we see if the currently free resources plus any resources held by the thread identified in the first step suffice to allow any other thread to finish; if so, the second thread's resources will also eventually be released back to the system. We continue this process until we have identified all threads guaranteed to finish, provided we serve requests in a particular order. If that set includes all of the threads, the state is safe.

EXAMPLE: Page allocation with the Banker's Algorithm. Suppose we have a system with 8 pages of memory and three processes: A, B, and C, which need 4, 5, and 5 pages to complete, respectively.

If they take turns requesting one page each, and the system grants requests in order, the system deadlocks, reaching a state where each process is stuck until some other process releases memory:

Process	**Allocation**											
A	0	**1**	1	1	**2**	2	2	**3**	3	3	**wait**	**wait**
B	0	0	**1**	1	1	**2**	2	2	**3**	3	3	**wait**
C	0	0	0	**1**	1	1	**2**	2	2	**wait**	**wait**	**wait**
Total	0	1	2	3	4	5	6	7	8	8	8	8

On the other hand, if the system follows the Banker's Algorithm, then it can delay some processes and guarantee that all processes eventually complete:

Process	**Allocation**																		
A	0	**1**	1	1	**2**	2	2	**3**	3	3	**4**	**0**	0	0	0	0	0	0	0
B	0	0	**1**	1	1	**2**	2	2	**wait**	**wait**	**wait**	**wait**	**3**	**4**	4	**5**	**0**	0	0
C	0	0	0	**1**	1	1	**2**	2	2	**wait**	**wait**	**wait**	**3**	3	**wait**	**wait**	**4**	**5**	**0**
Total	0	1	2	3	4	5	6	7	7	7	8	4	6	7	7	8	4	5	0

By delaying B and C in the ninth through twelfth steps, A can complete and release its resources. Then, by delaying C in the fifteenth and sixteenth steps, B can complete and release its resources.

The Banker's Algorithm is noticeably more involved than other approaches we discuss. Although it is rarely used in its entirety, understanding the distinction between *safe*, *unsafe*, and *deadlocked* states and how deadlock events depend on request ordering are key to preventing deadlock.

Additionally, understanding the Banker's Algorithm can help to design simple solutions for specific problems. For example, if we apply the Banker's Algorithm to the Dining Philosopher's problem, then it is safe for a philosopher to pick up a chopstick provided that afterwards (a) some philosopher will have two chopsticks or (b) a chopstick will remain on the table. In case (a), eventually that philosopher will finish eating and the other philosophers will be able to proceed. In case (b), the philosopher can pick up the chopstick because deadlock can still be avoided in the future.

EXAMPLE: Use the Banker's Algorithm to devise a rule for when it is safe for a thread to acquire a pair of locks, A and B, with mutually recursive locking.

ANSWER: Suppose a thread needs to acquire locks A and B, in that order, while another thread needs to acquire lock B first, then A. **A thread is always allowed to acquire its second lock. It may acquire its first lock provided the other thread does not already hold its first lock.** □

6.5.5 Detecting and Recovering From Deadlocks

Rather than preventing deadlocks, some systems allow deadlocks to occur and recover from them when they arise.

Why allow deadlocks to occur at all? Sometimes, it is difficult or expensive to enforce sufficient structure on the system's data and workloads to guarantee that deadlock will never occur. If deadlocks are rare, why pay the overhead in the common case to prevent them?

For this approach to work, we need: (i) a way to recover from deadlock when it occurs, ideally with minimal harm to the goals of the user, and (ii) a way to detect deadlock so that we know when to invoke the recovery mechanism. We discuss recovery first because it provides context for understanding the tradeoffs in implementing detection.

Recovering From Deadlocks

Recovering from a deadlock once it has occurred is challenging. A deadlock implies that some threads hold resources while waiting for others, and that progress is impossible.

Because the resources are by definition not revocable, forcibly taking resources away from some or all of the deadlocked threads is not an ideal solution. As a simple example, if a

process is part of a deadlock, some operating systems give the user the option to kill the process and release the process's resources. Although this sounds drastic, if a deadlocked process cannot make any progress, killing it does not make the user much worse off.

However, under the lock-based shared object programming abstractions we have discussed, killing all of the threads in a given process can be dangerous. If a deadlocked thread holds a lock on a shared kernel object, killing the thread and marking the lock as free could leave the kernel object in an inconsistent state.

Instead, we need some systematic way to recover when some required resource is unavailable. Two widely used approaches have been developed to deal with this issue:

- **Proceed without the resource.** Web services are often designed to be resilient to resource unavailability. A rule of thumb for the web is that a significant fraction of a web site's customers will give up and go elsewhere if the site's latency becomes too long, for whatever reason. Whether the problem is a hardware failure, software failure, or deadlock, does not really matter. The web site needs to be designed to quickly respond back to the user, regardless of the type of problem.

  Amazon's web site is a good example of this design paradigm. It is designed as an interlocking set of modules, where any individual module can be offline because of a failure. Thus, all other parts of the web site must be designed to be able to cope when some needed resource is unavailable. For example, under normal operation, Amazon's software will check the inventory to ensure that an item is in stock before completing an order. However, if a deadlock or failure causes the inventory server to delay responding beyond some threshold, the front-end web server will give up, complete the order, and then queue a background check to make sure the item was in fact in the inventory. If the item was in fact not available (e.g., because some other customer purchased it in the meantime), an apology is sent to the customer. As long as that does not happen often, it can be better than making the customer wait, especially in the case of deadlock, where the wait could be indefinite.

  Because deadlocks are rare and hard to test for, this requires coding discipline to handle error conditions systematically throughout the program.

Optimistic concurrency control

Transactions can also be used to avoid deadlocks. Optimistic concurrency control lets transactions execute in parallel without locking any data, but it only lets a transaction commit if none of the objects accessed by the transaction have been modified since the transaction began. Otherwise, the transaction must abort and retry.

To implement transactions with optimistic concurrency control, Each transaction keeps track of which versions of which objects it reads and updates. All updates are applied to a local copy. Then, before a transaction commits, the system verifies that no object the transaction accessed has been modified in the meantime; if there is a conflict, the transaction must abort. Of course, committing a transaction may invalidate other transactions that are in progress (ones that use data modified by this transaction). Those conflicts will be detected when the later transactions try to commit.

Optimistic concurrency control works well when different transactions most commonly use different subsets of data. In these cases, the approach not only eliminates deadlock, but it also maximizes concurrency since threads do not wait for locks. On the other hand, many conflicting, concurrent transactions increase overhead by repeatedly rolling back and re-executing transactions.

- **Transactions: rollback and retry.** A more general technique is used by *transactions*; transactions provide a safe mechanism for revoking resources assigned to a thread. We discuss transactions in detail in Chapter 14; they are widely used in both databases and file systems. For deadlock recovery, transactions provide two important services:

  1. **Thread rollback.** Transactions ensure that revoking locks from a thread does not leave the system's objects in an inconsistent state. Instead, we rollback, or undo, the deadlocked thread's actions to a clean state. To fix the deadlock, we can choose one or more victim threads, stop them, undo their actions, and let other threads proceed.
  2. **Thread restarting.** Once the deadlock is broken and other threads have completed some or all of their work, the victim thread is restarted. When these threads complete, the system behaves as if the victim threads never caused a deadlock but, instead, just had their executions delayed.

  A transaction defines a safe point for rollback and restart. Each transaction has a beginTransaction and endTransaction statement; rollback undoes all changes back to beginTransaction. After a rollback, the thread can be safely restarted at the beginTransaction.

  A key feature of transactions is that no other thread is allowed to see the results of a transaction until the transaction completes. That way, if the changes a transaction makes need to be rolled back due to a deadlock, only that one thread is affected. This can be accomplished with two-phase locking, provided locks are not released until after the transaction is complete. If the transaction is successful, it *commits*, the transaction's locks are released, and the transaction's changes to shared state become visible to other threads.

  If, however, a transaction fails to reach its endTransaction statement (e.g., because of a deadlock or because some other exception occurred), the transaction *aborts*. The system can reset all of the state modified by the transaction to what it was when the transaction began. One way to support this is to maintain a copy of the initial values of all state modified by each transaction; this copy can be discarded when the transaction commits.

  If a transactional system becomes deadlocked, the system can abort one or more of the deadlocked transactions. Aborting these transactions rolls back the system's state to what it would have been if these transactions had never started and releases the aborted transactions' locks and other resources. If aborting the chosen transactions

releases sufficient resources, the deadlock is broken, and the remaining transactions can proceed. If not, the system can abort additional transactions.

A related question that arises in transactional systems is *which* thread to abort and which threads to allow to proceed. An important consideration is liveness. Progress can be ensured, and starvation avoided, by prioritizing the oldest transactions. Then, when the system needs to abort some transaction, it can abort the *youngest*. This ensures that *some* transaction, e.g., the oldest, will eventually complete. The aborted transaction eventually becomes the oldest, and so it also will complete.

An example of this approach is *wound wait*. With wound wait, a younger transaction may wait for a resource held by an older transaction. Eventually, the older transaction will complete and release the resource, so deadlock cannot result. However, if an older transaction needs to wait on a resource held by a younger transaction, the resource is preempted and the younger transaction is aborted and restarted.

Detecting Deadlock

Once we have a general way to recover from a deadlock, we need a way to tell if a deadlock has occurred, so we know when to trigger the recovery. An important consideration is that the detection mechanism can be conservative: it can trigger the repair if we *might* be in a deadlock state. This approach risks a false positive where a non-deadlocked thread is incorrectly classified as deadlocked. Depending on the overhead of the repair operation, it can sometimes be more efficient to use a simpler mechanism for detection even if that leads to the occasional false positive.

For example, a program can choose to wait only briefly (or not to wait at all) before declaring that recovery is needed. We saw an example earlier with how Amazon's web site is designed. As another example, in old-style, circuit-switched telephone networks, a call reserved a circuit at a series of switches along its path. If the connection setup failed to find a free circuit at any hop, rather than wait for a circuit at the next hop to become free, it cancelled the connection attempt and gave the user an error message, "All circuits are busy. Please try again later."

A modern analogue is the Internet. When a router is overloaded and runs out of packet buffers, it simply drops incoming packets. An alternative would be for each router to wait to send a packet until it knows the next router has room — an approach that could lead to deadlock. Precisely identifying whether deadlock has occurred would incur more overhead than simply dropping and resending some packets.

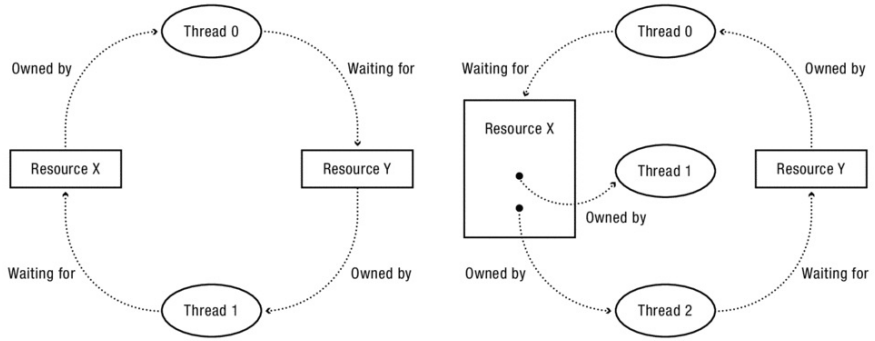

Figure 6.22: Example graphs used for deadlock detection. Left: single instance of each resource. Right: multiple instances of one resource. Threads and resources are nodes; directed edges represent the *owned by* and *waiting for* relationships among them.

There are various ways to identify deadlocks more precisely.

If there are several resources and only one thread can hold each resource at a time (e.g., one printer, one keyboard, and one audio speaker or several mutual exclusion locks), then we can detect a deadlock by analyzing a simple graph. In the graph, shown on the left in Figure 6.22, each thread and each resource is represented by a node. There is a directed edge (i) from a resource to a thread if the resource is *owned by* the thread and (ii) from a thread to a resource if the thread is *waiting for* the resource. There is a deadlock if and only if there is a cycle in such a graph.

If there are multiple instances of some resources, then we represent a resource with k interchangeable instances (e.g., k equivalent printers) as a node with k connection points. This is illustrated by the right graph in Figure 6.22. Now, a cycle is a necessary but not sufficient condition for deadlock.

Another solution, described by Coffman, Elphick, and Shoshani in 1971 is a variation of Dijkstra's Banker's Algorithm. In this algorithm, we assume we no longer know max[][], so we cannot assess whether the current state is safe or whether some future sequence of requests can force deadlock. However, we can look at the current set of resources, granted requests, and pending requests and ask whether it is possible for the current set of requests to eventually be satisfied assuming no more requests come and all threads eventually complete. If so, there is no deadlock (although we may be in an unsafe state); otherwise, there is a deadlock.

```
// A state is safe iff there exists a safe sequence of grants that would allow
// all threads to eventually receive their maximum resource needs.
//
// avail[] holds free resource count
// alloc[][] holds current allocation
// request[][] holds currently-blocked requests
```

```
bool
ResourceMgr::isDeadlocked() {
    int j;
    int toBeAvail[] = copy avail[];
    bool finish[] = [false, false, false, ...]; // finish[j] is true if thread
                                                // j is guaranteed to finish

    while(true) {
        j = any threadID such that (finish[j] == false) &&
                                (forall i: request[i][j] <= toBeAvail[i]);
        if (no such j exists) {
            if (forall j: finish[j] == true) {
                return false;
            } else {
                return true;
            }
        } else {
            // Thread j *may* eventually finish and
            // return its current allocation to the pool.
            finish[j] = true;
            forall i: toBeAvail[i] = toBeAvail[i] + alloc[i][j];
        }
    }
}
```

Figure 6.23: Coffman et al.'s test for deadlock. This algorithm is similar to the isSafe() test of the Banker's Algorithm shown in Figure 6.21.

Figure 6.23 shows the pseudo-code for the isDeadlocked method, a variation of the isSafe method shown in Figure 6.21 for the Banker's Algorithm.

One might hope that we could avoid deadlock by asking, "Will satisfying the current request put us in a deadlocked state?" and then blocking any request that does. The Coffman et al. algorithm highlights that deadlock is determined not just by what requests are granted but also by what requests are waiting. The request that triggers deadlock ("circular wait") will be a request that waits, not one that is granted.

6.6 Non-Blocking Synchronization

Chapter 5 described a core abstraction for synchronization — shared objects, with one lock per object. This abstraction works well for building multi-threaded programs the vast majority of the time. As concurrent programs become more complicated, however, issues of lock contention, the semantics of operations that span multiple objects, and deadlock can arise. Worse, the solutions to these issues often require us to compromise modularity; for example, whether a particular program can deadlock requires understanding in detail how the implementations of various shared objects interact.

Some researchers have posed a radical question: would it be better to write complex concurrent programs without locks? By eliminating locking, we would remove lock contention and deadlock as design considerations, fostering a more modular program structure. However, these techniques can be *much* more complex to use. To date, concurrent implementations without locks have only been used for a few carefully designed runtime library modules written by expert programmers. We sketch the ideas because there is a chance that they will become more important as the number of processors per computer continues to increase.

Today, the cases where these approaches are warranted are rare. These advanced techniques should only be considered by experienced programmers who have mastered the basic lock-based approaches. Many of you will probably never need to use these techniques. If you are tempted to do so, take extra care. Measure the performance of your system to ensure that these techniques yield significant gains, and seek out extra peer review from trusted colleagues to help ensure that the code works as intended.

Programmers often assume that acquiring a lock is an expensive operation, and therefore try to reduce locking throughout their programs. The most likely result from premature optimization is a program that is buggy, hard to maintain, no faster than a clean implementation, and, ironically, harder to tune than a cleanly architected program. On most platforms, acquiring or releasing a lock is a highly tuned primitive — acquiring an uncontended lock is often nearly free. If there is contention, you probably needed the lock!

In Section 6.3, we saw an example of synchronization without locks. RCU lets reads proceed without acquiring a lock or updating shared synchronization state, but it still requires updates to acquire locks. If the thread that holds the lock is interrupted, has a bug that causes it to stop making progress, or becomes deadlocked, other threads can be delayed for a long — perhaps unlimited — period of time.

It is possible to build data structures that are completely *non-blocking* for both read and write operations. A non-blocking method is one where one thread is never required to wait for another thread to complete its operation. Acquiring a lock is a blocking operation: if the thread holding the lock stops, is delayed, or deadlocks, all other threads must wait for it to finish the critical section.

More formally, a *wait-free data structure* is one that guarantees progress for every thread: every method finishes in a finite number of steps, regardless of the state of other threads executing in the data structure or their rate of execution. A *lock-free data structure* is one that guarantees progress for some thread: some method will finish in a finite number of steps.

A common building block for wait-free and lock-free data structures is the atomic compare-and-swap instruction available on most modern processors. We saw a taste of this in the implementation of the MCS lock in Section 6.3. There, we used compare-and-swap to atomically append to a linked list of waiting threads *without first acquiring a lock.*

Wait-free and lock-free data structures apply this idea more generally to completely eliminate the use of locks. For example, a lock-free hash table could be built as an array of pointers to each bucket:

- **Lookup.** A lookup de-references the pointer and checks the bucket.
- **Update.** To update a bucket, the thread allocates a new copy of the bucket, and then uses compare-and-swap to atomically replace the pointer if and only if it has not been changed in the meantime. If two threads simultaneously attempt to update the bucket

(for example, to add a new entry), one succeeds and the other must retry.

The logic can be much more complex for more intricate data structures, and as a result, designing efficient wait-free and lock-free data structures remains the domain of experts. Nonetheless, non-blocking algorithms exist for a wide range of data structures, including FIFO queues, double-ended queues, LIFO stacks, sets, and balanced trees. Several of these can be found in the Java Virtual Machine runtime library.

In addition, considerable effort has also gone into studying ways to automate the construction of wait-free and lock-free data structures. For example, transactions with optimistic concurrency control provide a very flexible approach to implementing lock-free applications. Recall that optimistic concurrency control lets transactions proceed without locking the data they access. Transactions abort if, at commit-time, any of their accessed data has changed in the meantime. Most modern databases use a form of optimistic concurrency control to provide atomic and fault-tolerant updates of on-disk data structures.

EXAMPLE: Is optimistic concurrency control lock-free, wait-free, or both?

ANSWER: To see that **optimistic concurrency control is lock-free,** consider two conflicting transactions executing at the same time. The first one to commit succeeds, and the second must abort and retry. **An implementation is wait-free if it uses wound wait or some other mechanism to bound the number of retries for a transaction to successfully commit.** □

Extending this idea, *software transactional memory (STM)* is a promising approach to support general-purpose transactions for in-memory data structures. Unfortunately, the cost of an STM transaction is often significantly higher than that of a traditional critical section; this is because of the need to maintain the state required to check dependencies and the state required either to update the object if there is no conflict or to roll back its state if a conflict is detected. It is an open question whether the overhead of STM can be reduced to where it can be used more widely. In situations where STM can be used, it provides a way to compose different modules without having to lock contention or deadlock concerns.

6.7 Summary and Future Directions

Advanced synchronization techniques should be approached with caution. Your first goal should be to construct a program that works, even if doing so means putting “one big lock” around everything in a data structure or even in an entire program.

Resist the temptation to do anything more complicated unless you **know** that doing so is necessary. How do you know? Do not guess. Measure your system’s performance. Measuring the “before” and “after” performance of a program and its subsystems not only helps you make good decisions about the program on which you are working, but it also helps you develop good intuition for the programs you write in the future.

Spend time early in the design process developing a clean structure for your program. Given that issues with multi-object synchronization often blur module boundaries, it is vital to have an overall structure that lets you reason about how the different pieces of your program will interact. Strive for a strict layering or hierarchy of modules. It is easier to make such programs deadlock-free, and it is easier to test them as well.

Although performance is important, it is usually easier to start with a clean, simple, and correct design, measure it to identify its bottlenecks, and then optimize the bottlenecks than to start with a complex design and try to tune its performance, let alone fix its bugs.

In this chapter, we have presented a set of conceptual tools and techniques for managing complex, multi-object concurrent programs. We have addressed: estimating the impact of locks on multiprocessor performance, design patterns to reduce contention for locks, implementation techniques such as MCS and RCU for high-contention locks, strategies for achieving atomicity across multiple operations on the same object or across objects, and algorithms for deadlock prevention and recovery.

Yet, writing concurrent programs remains frustratingly complex. We believe that an important area for future work will be to develop better tools for managing and reducing that complexity. The last decade has seen the development of a new generation of tools for helping programmers improve software reliability, by automatically identifying test coverage, memory leaks, reuse of de-allocated data, buffer overflows, and bad pointer arithmetic.

Extending this approach to concurrent programs is a grand challenge. A promising avenue is to use automated tools for detecting memory races; a well-written program should have no reads or writes to shared memory without holding the lock that protects that data structure. Once a program has been shown to be without races, model checking can be used to systematically test that shared objects work for all possible thread interleavings.

Exercises

1. Figure 6.13 shows the parallel execution of some requests and an equivalent sequential execution — request 1 then request 2 then request 3. Two other sequential executions are also equivalent to the parallel execution shown in the figure. What are these other equivalent sequential executions?
2. Generalize the rules for two-phase locking to include both mutual exclusion locks and readers/writers locks. What can be done in the expanding phase? What can be done in the contracting phase?
3. Consider the variation of the Dining Philosophers problem shown in Figure 6.17, where all unused chopsticks are placed in the center of the table and any philosopher can eat with any two chopsticks.

   One way to prevent deadlock in this system is to provide sufficient resources. For a system with n philosophers, what is the minimum number of chopsticks that ensures deadlock freedom? Why?
4. If the queues between stages are finite, is it possible for a staged architecture to deadlock even if each individual stage is internally deadlock free? If so, give an example. If not, prove it.

5. Suppose you build a system using a staged architecture with some fixed number of threads operating in each stage. Assuming each stage is individually deadlock free, describe two ways to guarantee that your system as a whole cannot deadlock. Each way should eliminate a different one of the 4 necessary conditions for deadlock.
6. Consider a system with four mutual exclusion locks (A, B, C, and D) and a readers/writers lock (E). Suppose the programmer follows these rules:
    a. Processing for each request is divided into two parts.
    b. During the first part, no lock may be released, and, if E is held in writing mode, it cannot be downgraded to reading mode. Furthermore, lock A may not be acquired if any of locks B, C, D, or E are held in any mode. Lock B may not be acquired if any of locks C, D, or E are held in any mode. Lock C may not be acquired if any of locks D or E are held in any mode. Lock D may not be acquired if lock E is held in any mode. Lock E may always be acquired in read mode or write mode, and it can be upgraded from read to write mode but not downgraded from write to read mode.
    c. During the second part, any lock may be released, and lock E may be downgraded from write mode to read mode; releases and downgrades can happen in any order; by the end of part 2, all locks must be released; and no locks may be acquired or upgraded.

    Do these rules ensure serializability? Do they ensure freedom from deadlock? Why?
7. In RCUList::remove, a possible strategy to increase concurrency would be to hold a read lock while searching for the target item, and to grab the write lock once it is found. Specifically: (i) replace the writeLock and writeUnlock calls with readLock and readUnlock calls, and (ii) insert new writeLock and writeUnlock calls at the beginning and end of the code that is executed when the if conditional test succeeds. Will this work?
8. Implement a highly concurrent, multi-threaded file buffer cache. A buffer cache stores recently used disk blocks in memory for improved latency and throughput. Disk blocks have unique numbers and are fixed size. The cache provides two routines:

```
void blockread(char *x, int blocknum);

void blockwrite(char *x, int blocknum);
```

    These routines read/write complete, block-aligned, fixed-size blocks. blockread reads a block of data into x; blockwrite (eventually) writes the data in x to disk. On a read, if the requested data is in the cache, the buffer will return it. Otherwise, the buffer must fetch the data from disk, making room in the cache by evicting a block as necessary. If the evicted block is modified, the cache must first write the modified data back to disk. On a write, if the block is not already in the buffer, it must make room for the new block. Modified data is stored in the cache and written back later to disk when the block is evicted.

    Multiple threads can call blockread and blockwrite concurrently, and to the maximum degree possible, those operations should be allowed to complete in parallel. You should assume the disk driver has been implemented; it provides the same interface as the file buffer cache: diskblockread and diskblockwrite. The disk driver routines are synchronous (the calling thread blocks until the disk operation completes) and re-entrant (while one thread is blocked, other threads can call into the driver to queue requests).
9. Suppose we have a version of the Dining Philosopher's problem where the chopsticks are placed in the middle of the table, each Philosopher needs three chopsticks before she will start to eat, and every Philosopher will return all of their chopsticks to the shared pool when done eating. (For example, the Philosopher needs two chopsticks to eat with and one to point at the white board.)
    a. Using the Banker's Algorithm, devise a rule for when is it safe for a Philosopher to pick up a chopstick. Explain why.
    b. Now suppose each Philosopher needs k chopsticks, for $k > 3$. Generalize the rule you developed above to work for any k.

7. Scheduling

Time is money —*Ben Franklin*

The best performance improvement is the transition from the non-working state to the working state. That's infinite speedup. —*John Ousterhout*

When there are multiple things to do, how do you choose which one to do first? In the last few chapters, we have described how to create threads, switch between them, and synchronize their access to shared data. At any point in time, some threads are running on the system's processor. Others are waiting their turn for a processor. Still other threads are blocked waiting for I/O to complete, a condition variable to be signaled, or for a lock to be released. When there are more runnable threads than processors, the *processor scheduling policy* determines which threads to run first.

You might think the answer to this question is easy: just do the work in the order in which it arrives. After all, that seems to be the only fair thing to do. Because it is obviously fair, almost all government services work this way. When you go to your local Department of Motor Vehicles (DMV) to get a driver's license, you take a number and wait your turn. Although fair, the DMV often feels slow. There's a reason why: as we'll see later in this chapter, doing things in order of arrival is sometimes the worst thing you can do in terms of improving user-perceived response time. Advertising that your operating system uses the same scheduling algorithm as the DMV is probably not going to increase your sales!

You might think that the answer to this question is unimportant. With the million-fold improvement in processor performance over the past thirty years, it might seem that we are a million times less likely to have anything waiting for its turn on a processor. We disagree! Server operating systems in particular are often overloaded. Parallel applications can create more work than processors, and if care is not taken in the design of the scheduling policy, performance can badly degrade. There are subtle relationships between scheduling policy and energy management on battery-powered devices such as smartphones and laptops. Further, scheduling issues apply to any scarce resource, whether the source of contention is the processor, memory, disk, or network. We will revisit the issues covered in this chapter throughout the rest of the book.

Scheduling policy is not a panacea. Without enough capacity, performance may be poor regardless of which thread we run first. In this chapter, we will also discuss how to predict overload conditions and how to adapt to them.

Fortunately, you probably have quite a bit of intuition as to impact of different scheduling policies and capacity on issues like response time, fairness, and throughput. Anyone who waits in line probably wonders how we could get the line to go faster. That's true whether we're waiting in line at the supermarket, a bank, the DMV, or at a popular restaurant. Remarkably, in each of these settings, there is a different approach to how they deal with waiting. We will try to answer why.

There is no one right answer; rather, any scheduling policy poses a complex set of

tradeoffs between various desirable properties. The goal of this chapter is not to enumerate all of the interesting possibilities, explore the full design space, or even to identify specific useful policies. Instead, we describe some of the trade-offs and try to illustrate how a designer can approach the problem of selecting a scheduling policy.

Consider what happens if you are running the web site for a company trying to become the next Facebook. Based on history, you'll be able to guess how much server capacity you need to be able to keep up with demand and still have reasonable response time. What happens if your site appears on Slashdot, and suddenly you have twice as many users as you had an hour ago? If you are not careful, everyone will think your site is terribly slow, and permanently go elsewhere. Google, Amazon, and Yahoo have each estimated that they lose approximately 5-10% of their customers if their response time increases by as little as 100 milliseconds. If faced with overload:

- Would quickly implementing a different scheduling policy help, or hurt?
- How much worse will your performance be if the number of users doubles again?
- Should you turn away some users so that others will get acceptable performance?
- Does it matter which users you turn away?
- If you run out to the local electronics store and buy a server, how much better will performance get?
- Do the answers change if you are under a denial-of-service attack by a competitor?

In this chapter, we will try to give you the conceptual and analytic tools to help you answer these questions.

Performance terminology

In Chapter 1 we defined some performance-related terms we will use throughout this chapter and the rest of the book; we summarize those terms here.

- **Task.** A user request. A task is also often called a *job*. A task can be any size, from simply redrawing the screen to show the movement of the mouse cursor to computing the shape of a newly discovered protein. When discussing scheduling, we use the term task, rather than thread or process, because a single thread or process may be responsible for multiple user requests or tasks. For example, in a word processor, each character typed is an individual user request to add that character to the file and display the result on the screen.
- **Response time (or delay).** The user-perceived time to do some task.
- **Predictability.** Low variance in response times for repeated requests.
- **Throughput.** The rate at which tasks are completed.
- **Scheduling overhead.** The time to switch from one task to another.
- **Fairness.** Equality in the number and timeliness of resources given to each task.
- **Starvation.** The lack of progress for one task, due to resources given to a higher priority task.

Chapter roadmap:

- **Uniprocessor Scheduling.** How do uniprocessor scheduling policies affect fairness, response time, and throughput? (Section 7.1)
- **Multiprocessor Scheduling.** How do scheduling policies change when we have multiple processor cores per computer? (Section 7.2)
- **Energy-Aware Scheduling.** Many new computer systems can save energy by turning off portions of the computer, slowing the execution speed. How do we make this tradeoff while minimizing the impact on user perceived response time? (Section 7.3)
- **Real-Time Scheduling.** More generally, how do we make sure tasks finish in time? (Section 7.4)
- **Queueing Theory.** In a server environment, how are response time and throughput affected by the rate at which requests arrive for processing and by the scheduling policy? (Section 7.5)
- **Overload Management.** How do we keep response time reasonable when a system becomes overloaded? (Section 7.6)
- **Case Study: Servers in a Data Center.** How do we combine these technologies to manage servers a data center? (Section 7.7)

7.1 Uniprocessor Scheduling

We start by considering one processor, generalizing to multiprocessor scheduling policies in the next section. We begin with three simple policies — first-in-first-out, shortest-job-first, and round robin — as a way of illustrating scheduling concepts. Each approach has its own the strengths and weaknesses, and most resource allocation systems (whether for processors, memory, network or disk) combine aspects of all three. At the end of the discussion, we will show how the different approaches can be synthesized into a more practical and complete processor scheduler.

Before proceeding, we need to define a few terms. A *workload* is a set of tasks for some system to perform, along with when each task arrives and how long each task takes to complete. In other words, the workload defines the input to a scheduling algorithm. Given a workload, a processor scheduler decides when each task is to be assigned the processor.

We are interested in scheduling algorithms that work well across a wide variety of environments, because workloads will vary quite a bit from system to system and user to user. Some tasks are *compute-bound* and only use the processor. Others, such as a compiler or a web browser, mix I/O and computation. Still others, such as a BitTorrent download, are *I/O-bound*, spending most of their time waiting for I/O and only brief periods computing. In the discussion, we start with very simple compute-bound workloads and then generalize to include mixtures of different types of tasks as we proceed.

Some of the policies we outline are the best possible policy on a particular metric and workload, and some are the worst possible policy. When discussing optimality and pessimality, we are only comparing to policies that are *work-conserving*. A scheduler is work-conserving if it never leaves the processor idle if there is work to do. Obviously, a trivially poor policy has the processor sit idle for long periods when there are tasks in the ready list.

Our discussion also assumes the scheduler has the ability to *preempt* the processor and give it to some other task. Preemption can happen either because of a timer interrupt, or because some task arrives on the ready list with a higher priority than the current task, at least according to some scheduling policy. We explained how to switch the processor between tasks in Chapter 2 and Chapter 4. While much of the discussion is also relevant to non-preemptive schedulers, there are few such systems left, so we leave that issue aside for simplicity.

7.1.1 First-In-First-Out (FIFO)

Perhaps the simplest scheduling algorithm possible is first-in-first-out (FIFO): do each task in the order in which it arrives. (FIFO is sometimes also called first-come-first-served, or FCFS.) When we start working on a task, we keep running it until it finishes. FIFO minimizes overhead, switching between tasks only when each one completes. Because it minimizes overhead, if we have a fixed number of tasks, and those tasks only need the processor, FIFO will have the best throughput: it will complete the most tasks the most quickly. And as we mentioned, FIFO appears to be the definition of fairness — every task patiently waits its turn.

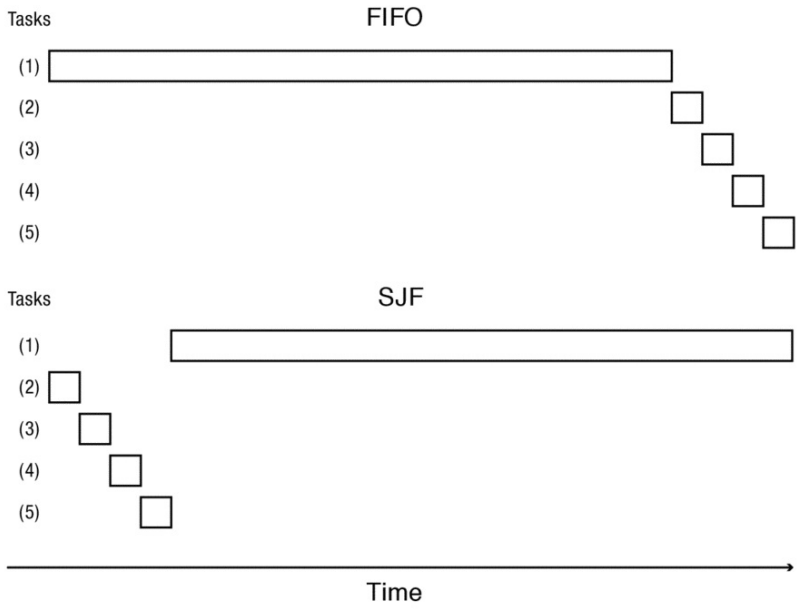

Figure 7.1: Completion times with FIFO (top) and SJF (bottom) scheduling when several short tasks (2-5) arrive immediately after a long task (1).

Unfortunately, FIFO has a weakness. If a task with very little work to do happens to land in line behind a task that takes a very long time, then the system will seem very inefficient. Figure 7.1 illustrates a particularly bad workload for FIFO; it also shows SJF, which we will discuss in a bit. If the first task in the queue takes one second, and the next four arrive an instant later, but each only needs a millisecond of the processor, then they will all need to wait until the first one finishes. The average response time will be over a second, but the optimal average response time is much less than that. In fact, if we ignore switching overhead, there are some workloads where FIFO is literally the worst possible policy for average response time.

FIFO and memcached

Although you may think that FIFO is too simple to be useful, there are some important cases where it is exactly the right choice for the workload. One such example is memcached. Many web services, such as Facebook, store their user data in a database. The database provides flexible and consistent lookups, such as, which friends need to be notified of a particular update to a user's Facebook wall. In order to improve performance, Facebook and other systems put a cache called memcached in front of the database, so that if a user posts two items to her Facebook wall, the system only needs to lookup the friend list once. The system first checks whether the information is cached, and if so uses that copy.

Because almost all requests are for small amounts of data, memcached replies to requests in FIFO order. This minimizes overhead, as there is no need to time slice between requests. For this workload where tasks are roughly equal in size, FIFO is simple, minimizes average response time, and even maximizes throughput. Win-win!

7.1.2 Shortest Job First (SJF)

If FIFO can be a poor choice for average response time, is there an optimal policy for minimizing average response time? The answer is yes: schedule the shortest job first (SJF).

Suppose we could know how much time each task needed at the processor. (In general, we will not know, so this is not meant as a practical policy! Rather, we use it as a thought experiment; later on, we will see how to approximate SJF in practice.) If we always schedule the task that has the least remaining work to do, that will minimize average response time. (For this reason, some call SJF shortest-remaining-time-first or SRTF.)

To see that SJF is optimal, consider a hypothetical alternative policy that is not SJF, but that we think might be optimal. Because the alternative is not SJF, at some point it will choose to run a task that is longer than something else in the queue. If we now switch the order of tasks, keeping everything the same, but doing the shorter task first, we will reduce the average response time. Thus, any alternative to SJF cannot be optimal.

Figure 7.1 illustrates SJF on the same example we used for FIFO. If a long task is the first to arrive, it will be scheduled (if we are work-conserving). When a short task arrives a bit later, the scheduler will preempt the current task, and start the shorter one. The remaining short tasks will be processed in order of arrival, followed by finishing the long task.

What counts as "shortest" is the remaining time left on the task, not its original length. If we are one nanosecond away from finishing an hour-long task, we will minimize average response time by staying with that task, rather than preempting it for a minute long task that just arrived on the ready list. Of course, if they both arrive at about the same time, doing the minute long task first will dramatically improve average response time.

Starvation and sample bias

Systems that might suffer from starvation require extra care when being measured. Suppose you want to compare FIFO and SJF experimentally. You set up two computers, one running each scheduler, and send them the same sequence of tasks. After some period, you stop and report the average response time of completed tasks. If some tasks starve, however, the set of completed tasks will be different for the two policies. We will have excluded the longest tasks from the results for SJF, skewing the average response time even further. Put another way, if you want to manipulate statistics to "prove" a point, this is a good trick to use!

How might you redesign the experiment to provide a valid comparison between FIFO and SJF?

Does SJF have any other downsides (other than being impossible to implement because it requires knowledge of the future)? It turns out that SJF is pessimal for variance in response time. By doing the shortest tasks as quickly as possible, SJF necessarily does longer tasks as slowly as possible (among policies that are work-conserving). In other words, there is a fundamental tradeoff between reducing average response time and reducing the variance in average response time.

Worse, SJF can suffer from starvation and frequent context switches. If enough short tasks arrive, long tasks may never complete. Whenever a new task on the ready list is shorter than the remaining time left on the currently scheduled task, the scheduler will switch to the new task. If this keeps happening indefinitely, a long task may never finish.

Suppose a supermarket manager reads a portion of this textbook and decides to implement shortest job first to reduce average waiting times. The manager tells herself: who cares about variance! A benefit is that there would no longer be any need for express lanes — if someone has only a few items, she can be immediately whisked to the front of the line, interrupting the parent shopping for eighteen kids. Of course, the wait times of the customers with full baskets skyrockets; if the supermarket is open twenty-four hours a day, customers with the largest purchases might have to wait until 3am to finally get through the line. This would probably lead their best customers to go to the supermarket down the street, not exactly what the manager had in mind!

Customers could also try to game the system: if you have a lot of items to purchase, simply go through the line with one item at a time — you will always be whisked to the front, at least until everyone else figures out the same dodge.

Shortest Job First and bandwidth-constrained web service

Although SJF may seem completely impractical, there are circumstances where it is exactly the right policy. One example is in a web server for static content. Many small-scale web servers are limited by their bandwidth to the Internet, because it is often more expensive to pay for more capacity. Web pages at most sites vary in size, with most pages being relatively short, while some pages are quite large. The average response time for accessing web pages is dominated by the more frequent requests to short pages, while the bandwidth costs are dominated by the less frequent requests to large pages.

This combination is almost ideal for using SJF for managing the allocation of network bandwidth by the server. With static pages, it is possible to predict from the name of the page how much bandwidth each request will consume. By transferring short pages first, the web server can ensure that its average response time is very low. Even if most requests are to small pages, the aggregate bandwidth for small pages is low, so requests to large pages are not significantly slowed down. The only difficulty comes when the web server is overloaded, because then the large page requests can be starved. As we will see later, overload situations need their own set of solutions.

7.1.3 Round Robin

A policy that addresses starvation is to schedule tasks in a round robin fashion. With Round Robin, tasks take turns running on the processor for a limited period of time. The scheduler assigns the processor to the first task in the ready list, setting a timer interrupt for some delay, called the *time quantum*. At the end of the quantum, if the task has not completed, the task is preempted and the processor is given to the next task in the ready list. The preempted task is put back on the ready list where it can wait its next turn. With Round Robin, there is no possibility that a task will starve — it will eventually reach the front of the queue and get its time quantum.

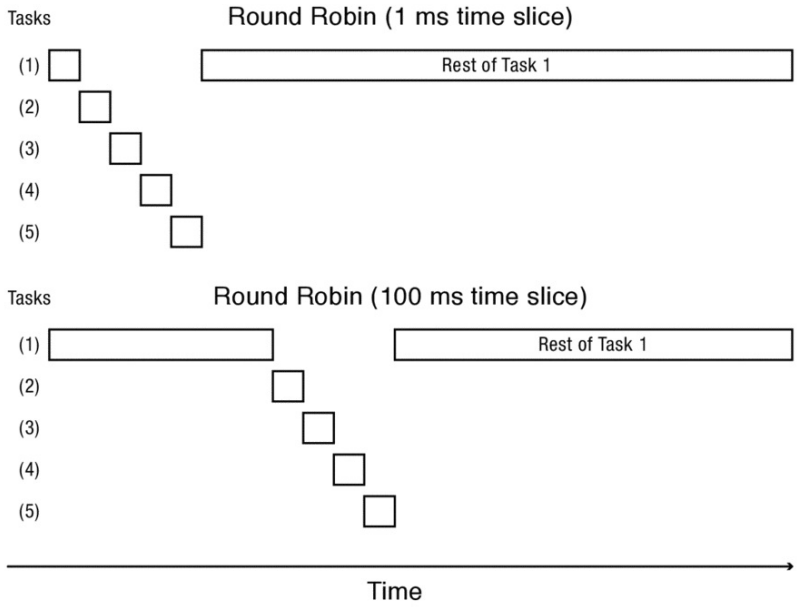

Figure 7.2: Completion times with Round Robin scheduling when short tasks arrive just after a long task, with a time quantum of 1 ms (top) and 100 ms (bottom).

Of course, we need to pick the time quantum carefully. One consideration is overhead: if we have too short a time quantum, the processor will spend all of its time switching and getting very little useful work done. If we pick too long a time quantum, tasks will have to wait a long time until they get a turn. Figure 7.2 shows the behavior of Round Robin, on the same workload as in Figure 7.1, for two different values for the time quantum.

A good analogy for Round Robin is a particularly hyperkinetic student, studying for multiple finals simultaneously. You won't get much done if you read a paragraph from one textbook, then switch to reading a paragraph from the next textbook, and then switch to yet a third textbook. However, if you never switch, you may never get around to studying for some of your courses.

What is the overhead of a Round Robin time slice?

One might think that the cost of switching tasks after a time slice is modest: the cost of interrupting the processor, saving its registers, dispatching the timer interrupt handler, and restoring the registers of the new task. On a modern processor, all these steps can be completed in a few tens of microseconds.

However, we must also include the impact of time slices on the efficiency of the processor cache. Each newly scheduled task will need to fetch its data from memory into cache, evicting some of the data that had been stored by the previous task. Exactly how long this takes will depend on the memory hierarchy, the reference pattern of the new task, and whether any of its state is still in the cache from its previous time slice. Modern processors often have multiple levels of cache to improve performance. Reloading just the first level on-chip cache from scratch can take several milliseconds; reloading the second and third level caches takes even longer. Thus, it is typical for operating systems to set their time slice interval to be somewhere between 10 and 100 milliseconds, depending on the goals of the system: better responsiveness or reduced overhead.

One way of viewing Round Robin is as a compromise between FIFO and SJF. At one extreme, if the time quantum is infinite (or at least, longer than the longest task), Round Robin behaves exactly the same as FIFO. Each task runs to completion and then yields the processor to the next in line. At the other extreme, suppose it was possible to switch between tasks with zero overhead, so we could choose a time quantum of a single instruction. With fine-grained time slicing, tasks would finish in the order of length, as with SJF, but slower: a task A will complete within a factor of n of when it would have under SJF, where n is the maximum number of other runnable tasks.

Simultaneous multi-threading

Although zero overhead switching may seem far-fetched, most modern processors do a form of it called *simultaneous multi-threading (SMT)* or *hyperthreading*. With SMT, each processor simulates two (or more) virtual processors, alternating between them on a cycle-by-cycle basis. Since most threads need to wait for memory from time to time, another thread can use the processor during those gaps, or vice versa. In normal operation, neither thread is significantly slowed when running on an SMT.

You can test whether your computer implements SMT by testing how fast the processor operates when it has one or more tasks, each running a tight loop of arithmetic operations. (Note that on a multicore system, you will need to create enough tasks to fill up each of the cores, or physical processors, before the system will begin to use SMT.) With one task per physical processor, each task will run at the maximum rate of the processor. With a two-way SMT and two tasks per processor, each task will run at somewhat less than the maximum rate, but each task will run at approximately the same uniform speed. As you increase the number of tasks beyond the SMT level, however, the operating system will begin to use coarse-grained time slicing, so tasks will progress in spurts — alternating

time on and off the processor.

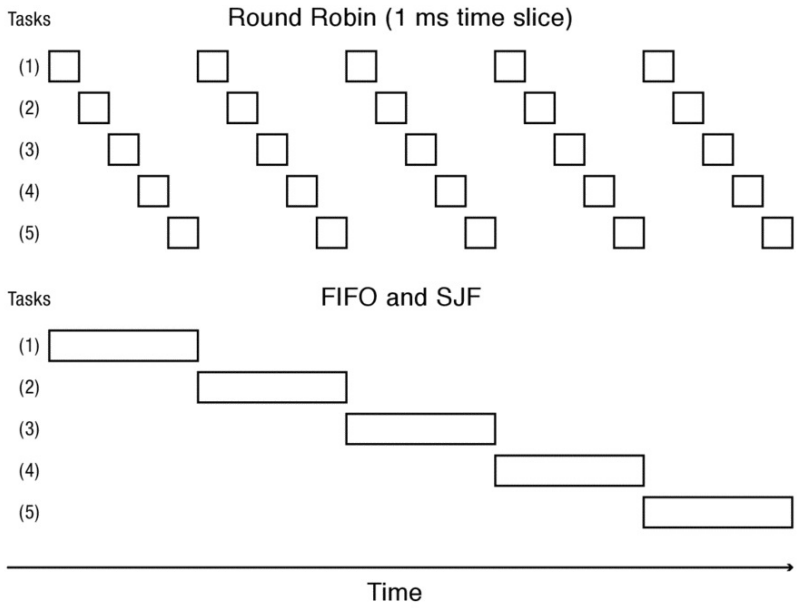

Figure 7.3: Completion times with Round Robin (top) versus FIFO and SJF (bottom) when scheduling equal length tasks.

Unfortunately, Round Robin has some weaknesses. Figure 7.3 illustrates what happens for FIFO, SJF, and Round Robin when several tasks start at roughly same time and are of the same length. Round Robin will rotate through the tasks, doing a bit of each, finishing them all at roughly the same time. This is nearly the worst possible scheduling policy for this workload! FIFO does much better, picking a task and sticking with it until it finishes. Not only does FIFO reduce average response time for this workload relative to Round Robin, no task is worse off under FIFO — every task finishes at least as early as it would have under Round Robin. Time slicing added overhead without any benefit. Finally, consider what SJF does on this workload. SJF schedules tasks in exactly the same order as FIFO. The first task that arrives will be assigned the processor, and as soon as it executes a single instruction, it will have less time remaining than all of the other tasks, and so it will run to completion. Since we know SJF is optimal for average response time, this means that both FIFO and Round Robin are optimal for some workloads and pessimal for others, just different ones in each case.

Round Robin and streaming video

Round Robin is sometimes the best policy even when all tasks are roughly the same size. An example is managing the server bandwidth for streaming video. When streaming, response time is much less of a concern than achieving a predictable, stable rate of progress. For this, Round Robin is nearly ideal: all streams progress at the same rate. As long as Round Robin serves the data as fast or faster than the viewer consumes the video stream, the time to completely download the stream is unimportant.

Depending on the time quantum, Round Robin can also be quite poor when running a mixture of I/O-bound and compute-bound tasks. I/O-bound tasks often need very short periods on the processor in order to compute the next I/O operation to issue. Any delay to be scheduled onto the processor can lead to system-wide slowdowns. For example, in a text editor, it often takes only a few milliseconds to echo a keystroke to the screen, a delay much faster than human perception. However, if we are sharing the processor between a text editor and several other tasks using Round Robin, the editor must wait several time quanta to be scheduled for each keystroke — with a 100 ms time quantum, this can become annoyingly apparent to the user.

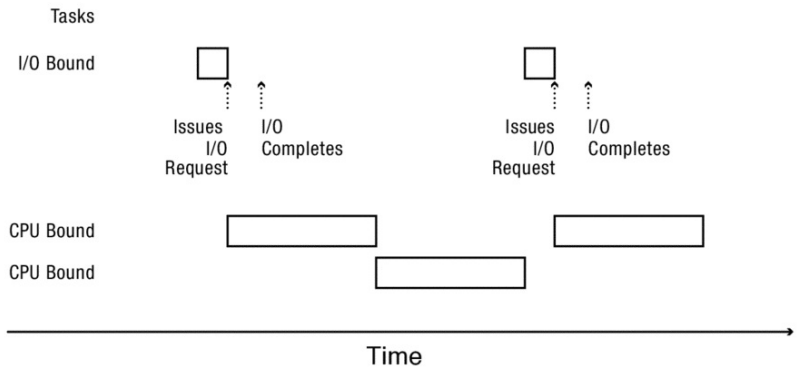

Figure 7.4: Scheduling behavior with Round Robin when running a mixture of I/O-bound and compute-bound tasks. The I/O-bound task yields the processor when it does I/O. Even though the I/O completes quickly, the I/O-bound task must wait to be reassigned the processor until the compute-bound tasks both complete their time quanta.

Figure 7.4 illustrates similar behavior with a disk-bound task. Suppose we have a task that computes for 1 ms and then uses the disk for 10 ms, in a loop. Running alone, the task can keep the disk almost completely busy. Suppose we also have two compute bound tasks; again, running by themselves, they can keep the processor busy. What happens when we run the disk-bound and compute-bound tasks at the same time? With Round Robin and a time quantum of 100 ms, the disk-bound task slows down by nearly a factor of twenty — each time it needs the processor, it must wait nearly 200 ms for its turn. SJF on this workload would perform well — prioritizing short tasks at the processor keeps the disk-bound task busy, while modestly slowing down the compute-bound tasks.

If you have ever tried to surf the web while doing a large BitTorrent download over a slow link, you can see that network operations visibly slow during the download. This is even though your browser may need to transfer only a very small amount of data to provide good responsiveness. The reason is quite similar. Browser packets get their turn, but only after being queued behind a much larger number of packets for the bulk download. Prioritizing the browser's packets would have only a minimal impact on the download speed and a large impact on the perceived responsiveness of the system.

7.1.4 Max-Min Fairness

In many settings, a fair allocation of resources is as important to the design of a scheduler as responsiveness and low overhead. On a multi-user machine or on a server, we do not want to allow a single user to be able to monopolize the resources of the machine, degrading service for other users. While it might seem that fairness has little value in single-user machines, individual applications are often written by different companies, each with an interest in making their application performance look good even if that comes at a cost of degrading responsiveness for other applications.

Another complication arises with whether we should allocate resources fairly among users, applications, processes, or threads. Some applications may run inside a single process, while others may create many processes, and each process may involve multiple threads. Round robin among threads can lead to starvation if applications with only a single thread are competing with applications with hundreds of threads. We can be concerned with fair allocation at any of these levels of granularity: threads within a process, processes for a particular user, users sharing a physical machine. For example, we could be concerned with making sure that every thread within a process makes progress. For simplicity, however, our discussion will assume we are interested in providing fairness among processes — the same principles apply if the unit receiving resources is the user, application, or thread.

Fairness is easy if all processes are compute-bound: Round Robin will give each process an equal portion of the processor. In practice, however, different processes consume resources at different rates. An I/O-bound process may need only a small portion of the processor, while a compute-bound process is willing to consume all available processor time. What is a fair allocation when there is a diversity of needs?

One possible answer is to say that whatever Round Robin does is fair — after all, each process gets an equal chance at the processor. As we saw above, however, Round Robin can result in I/O-bound processes running at a much slower rate than they would if they had the processor to themselves, while compute-bound processes are barely affected at all. That hardly seems fair!

While there are many possible definitions of fairness, a particularly useful one is called *max-min fairness*. Max-min fairness iteratively maximizes the minimum allocation given to a particular process (user, application or thread) until all resources are assigned.

If all processes are compute-bound, the behavior of max-min is simple: we maximize the minimum by giving each process exactly the same share of the processor — that is, by using Round Robin.

The behavior of max-min fairness is more interesting if some processes cannot use their entire share, for example, because they are short-running or I/O-bound. If so, we give those processes their entire request and redistribute the unused portion to the remaining processes. Some of the processes receiving the extra portion may not be able to use their entire revised share, and so we must iterate, redistributing any unused portion. When no remaining requests can be fully satisfied, we divide the remainder equally among all remaining processes.

Consider the example in the previous section. The disk-bound process needed only 10% of the processor to keep busy, but Round Robin only gave it 0.5% of the processor, while each of the two compute-bound processes received nearly 50%. Max-min fairness would assign 10% of the processor to the I/O-bound process, and it would split the remainder equally between the two compute-bound processes, with 45% each.

A hypothetical but completely impractical implementation of max-min would be to give the processor at each instant to whichever process has received the least portion of the processor. In the example above, the disk-bound task would always be scheduled instantly, preempting the compute-bound processes. However, we have already seen why this would not work well. With two equally long tasks, as soon as we execute one instruction in one task, it would have received more resources than the other one, so to preserve "fairness" we would need to instantly switch to the next task.

We can approximate a max-min fair allocation by relaxing this constraint — to allow a process to get ahead of its fair allocation by one time quantum. Every time the scheduler needs to make a choice, it chooses the task for the process with the least accumulated time on the processor. If a new process arrives on the queue with much less accumulated time, such as the disk-bound task, it will preempt the process, but otherwise the current process will complete its quantum. Tasks may get up to one time quantum more than their fair share, but over the long term the allocation will even out.

The algorithm we just described was originally defined for network, and not processor, scheduling. If we share a link between a browser request and a long download, we will get reasonable responsiveness for the browser if we have approximately fair allocation — the browser needs few network packets, and so under max-min its packets will always be scheduled ahead of the packets from the download.

Even this approximation, though, can be computationally expensive, since it requires tasks to be maintained on a priority queue. For some server environments, there can be tens or even hundreds of thousands of scheduling decisions to be made every second. To reduce the computational overhead of the scheduler, most commercial operating systems use a somewhat different algorithm, to the same goal, which we describe next.

7.1.5 Case Study: Multi-Level Feedback

Most commercial operating systems, including Windows, MacOS, and Linux, use a scheduling algorithm called *multi-level feedback queue (MFQ)*. MFQ is designed to achieve several simultaneous goals:

- **Responsiveness.** Run short tasks quickly, as in SJF.

- **Low Overhead.** Minimize the number of preemptions, as in FIFO, and minimize the time spent making scheduling decisions.
- **Starvation-Freedom.** All tasks should make progress, as in Round Robin.
- **Background Tasks.** Defer system maintenance tasks, such as disk defragmentation, so they do not interfere with user work.
- **Fairness.** Assign (non-background) processes approximately their max-min fair share of the processor.

As with any real system that must balance several conflicting goals, MFQ does not perfectly achieve any of these goals. Rather, it is intended to be a reasonable compromise in most real-world cases.

MFQ is an extension of Round Robin. Instead of only a single queue, MFQ has multiple Round Robin queues, each with a different priority level and time quantum. Tasks at a higher priority level preempt lower priority tasks, while tasks at the same level are scheduled in Round Robin fashion. Further, higher priority levels have *shorter* time quanta than lower levels.

Tasks are moved between priority levels to favor short tasks over long ones. A new task enters at the top priority level. Every time the task uses up its time quantum, it drops a level; every time the task yields the processor because it is waiting on I/O, it stays at the same level (or is bumped up a level); and if the task completes it leaves the system.

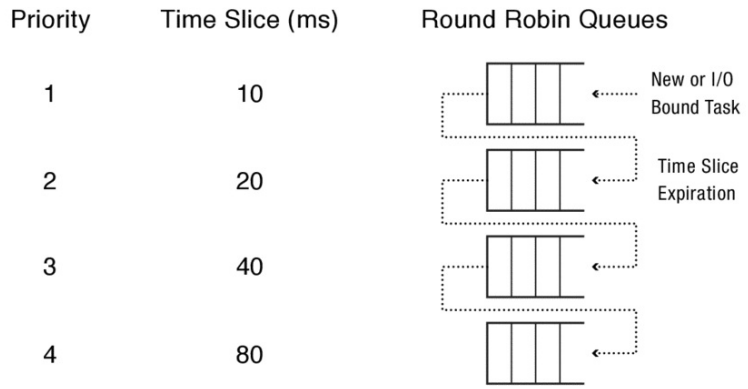

Figure 7.5: Multi-level Feedback Queue when running a mixture of I/O-bound and compute-bound tasks. New tasks enter at high priority with a short quantum; tasks that use their quantum are reduced in priority.

Figure 7.5 illustrates the operation of an MFQ with four levels. A new compute-bound task will start as high priority, but it will quickly exhaust its time quantum and fall to the next lower priority, and then the next. Thus, an I/O-bound task needing only a modest amount of computing will almost always be scheduled quickly, keeping the disk busy. Compute-bound tasks run with a long time quantum to minimize switching overhead while still sharing the processor.

So far, the algorithm we have described does not achieve starvation freedom or max-min fairness. If there are too many I/O-bound tasks, the compute-bound tasks may receive no time on the processor. To combat this, the MFQ scheduler monitors every process to ensure it is receiving its fair share of the resources. At each level, Linux actually maintains two queues — tasks whose processes have already reached their fair share are only scheduled if all other processes at that level have also received their fair share. Periodically, any process receiving less than its fair share will have its tasks increased in priority; equally, tasks that receive more than their fair share can be reduced in priority.

Adjusting priority also addresses strategic behavior. From a purely selfish point of view, a task can attempt to keep its priority high by doing a short I/O request immediately before its time quantum expires. Eventually the system will detect this and reduce its priority to its fair-share level.

Our previously hapless supermarket manager reads a bit farther into the textbook and realizes that supermarket express lanes are a form of multi-level queue. By limiting express lanes to customers with a few items, the manager can ensure short tasks complete quickly, reducing average response time. The manager can also monitor wait times, adding extra lanes to ensure that everyone is served reasonably quickly.

7.1.6 Summary

We summarize the lessons from this section:

- FIFO is simple and minimizes overhead.
- If tasks are variable in size, then FIFO can have very poor average response time.
- If tasks are equal in size, FIFO is optimal in terms of average response time.
- Considering only the processor, SJF is optimal in terms of average response time.
- SJF is pessimal in terms of variance in response time.
- If tasks are variable in size, Round Robin approximates SJF.
- If tasks are equal in size, Round Robin will have very poor average response time.
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin.
- Max-min fairness can improve response time for I/O-bound tasks.
- Round Robin and Max-min fairness both avoid starvation.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.

In the rest of this chapter, we extend these ideas to multiprocessors, energy-constrained environments, real-time settings, and overloaded conditions.

7.2 Multiprocessor Scheduling

Today, most general-purpose computers are multiprocessors. Physical constraints in circuit design make it easier to add computational power by adding processors, or cores, onto a single chip, rather than making individual processors faster. Many high-end desktops and servers have multiple processing chips, each with multiple cores, and each core with hyperthreading. Even smartphones have 2-4 processors. This trend is likely to accelerate, with systems of the future having dozens or perhaps hundreds of processors per computer.

This poses two questions for operating system scheduling:

- How do we make effective use of multiple cores for running sequential tasks?
- How do we adapt scheduling algorithms for parallel applications?

7.2.1 Scheduling Sequential Applications on Multiprocessors

Consider a server handling a very large number of web requests. A common software architecture for servers is to allocate a separate thread for each user connection. Each thread consults a shared data structure to see which portions of the requested data are cached, and fetches any missing elements from disk. The thread then spools the result out across the network.

How should the operating system schedule these server threads? Each thread is I/O-bound, repeatedly reading or writing data to disk and the network, and therefore makes many small trips through the processor. Some requests may require more computation; to keep average response time low, we will want to favor short tasks.

A simple approach would be to use a centralized multi-level feedback queue, with a lock to ensure only one processor at a time is reading or modifying the data structure. Each idle processor takes the next task off the MFQ and runs it. As the disk or network finishes requests, threads waiting on I/O are put back on the MFQ and executed by the network processor that becomes idle.

There are several potential performance problems with this approach:

- **Contention for the MFQ lock.** Depending on how much computation each thread does before blocking on I/O, the centralized lock may become a bottleneck, particularly as the number of processors increases.
- **Cache Coherence Overhead.** Although only a modest number of instructions are needed for each visit to the MFQ, each processor will need to fetch the current state of the MFQ from the cache of the previous processor to hold the lock. On a single processor, the scheduling data structure is likely to be already loaded into the cache. On a multiprocessor, the data structure will be accessed and modified by different processors in turn, so the most recent version of the data is likely to be cached only by the processor that made the most recent update. Fetching data from a remote cache can take two to three orders of magnitude longer than accessing locally cached data. Since the cache miss delay occurs while holding the MFQ lock, the MFQ lock is held for longer periods and so can become even more of a bottleneck.
- **Limited Cache Reuse.** If threads run on the first available processor, they are likely to be assigned to a different processor each time they are scheduled. This means that any data needed by the thread is unlikely to be cached on that processor. Of course, some of the thread's data will have been displaced from the cache during the time it was blocked, but on-chip caches are so large today that much of the thread's data will remain cached. Worse, the most recent version of the thread's data is likely to be in a remote cache, requiring even more of a slowdown as the remote data is fetched into the local cache.

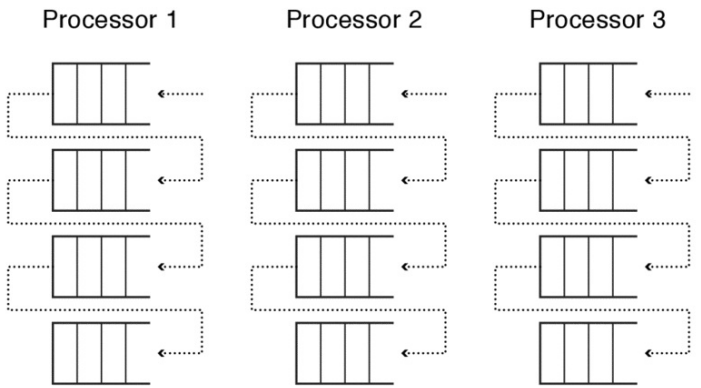

Figure 7.6: Per-processor scheduling data structures. Each processor has its own (multi-level) queue of ready threads.

For these reasons, commercial operating systems such as Linux use a *per-processor* data structure: a separate copy of the multi-level feedback queue for each processor. Figure 7.6 illustrates this approach.

Each processor uses *affinity scheduling*: once a thread is scheduled on a processor, it is returned to the same processor when it is re-scheduled, maximizing cache reuse. Each processor looks at its own copy of the queue for new work to do; this can mean that some processors can idle while others have work waiting to be done. Rebalancing occurs only if the queue lengths are persistent enough to compensate for the time to reload the cache for the migrated threads. Because rebalancing is possible, the per-processor data structures must still be protected by locks, but in the common case the next processor to use the data will be the last one to have written it, minimizing cache coherence overhead and lock contention.

7.2.2 Scheduling Parallel Applications

A different set of challenges occurs when scheduling parallel applications onto a multiprocessor. There is often a natural decomposition of a parallel application onto a set

of processors. For example, an image processing application may divide the image up into equal size chunks, assigning one to each processor. While the application could divide the image into many more chunks than processors, this comes at a cost in efficiency: less cache reuse and more communication to coordinate work at the boundary between each chunk.

If there are multiple applications running at the same time, the application may receive fewer or more processors than it expected or started with. New applications can start up, acquiring processing resources. Other applications may complete, releasing resources. Even without multiple applications, the operating system itself will have system tasks to run from time to time, disrupting the mapping of parallel work onto a fixed number of processors.

Oblivious Scheduling

One might imagine that the scheduling algorithms we have already discussed can take care of these cases. Each thread is time sliced onto the available processors; if two or more applications create more threads in aggregate than processors, multi-level feedback will ensure that each thread makes progress and receives a fair share of the processor. This is often called *oblivious scheduling*, as the operating system scheduler operates without knowledge of the intent of the parallel application — each thread is scheduled as a completely independent entity. Figure 7.7 illustrates oblivious scheduling.

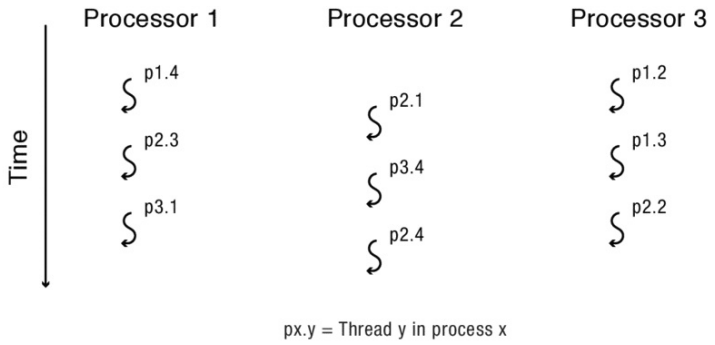

Figure 7.7: With oblivious scheduling, threads are time sliced by the multiprocessor operating system, with no attempt to ensure threads from the same process run at the same time.

Unfortunately, several problems can occur with oblivious scheduling on multiprocessors:

- **Bulk synchronous delay.** A common design pattern in parallel programs is to split work into roughly equal sized chunks; once all the chunks finish, the processors synchronize at a barrier before communicating their results to the next stage of the computation. This *bulk synchronous* parallelism is easy to manage — each processor works independently, sharing its results only with the next stage in the computation. Google MapReduce is a widely used bulk synchronous application.

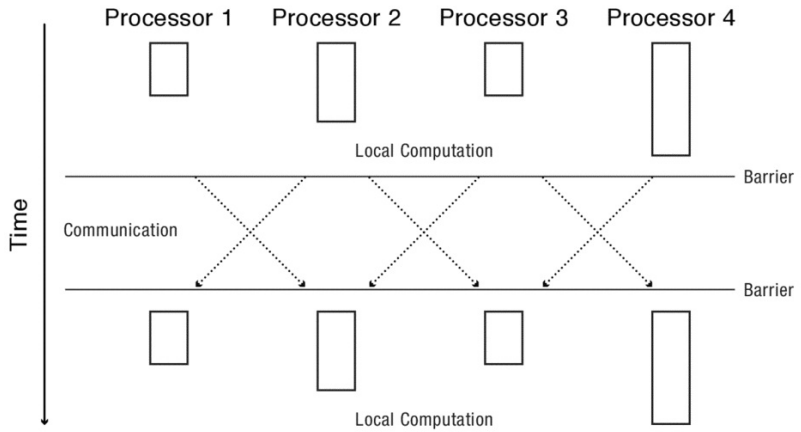

Figure 7.8: Bulk synchronous design pattern for a parallel program; each processor computes on local data and waits for every other processor to complete before proceeding to the next step. Preempting one processor can stall all processors until the preempted process is resumed.

Figure 7.8 illustrates the problem with bulk synchronous computation under oblivious scheduling. At each step, the computation is limited by the slowest processor to complete that step. If a processor is preempted, its work will be delayed, stalling the remaining processors until the last one is scheduled. Even if one of the waiting processors picks up the preempted task, a single preemption can delay the entire computation by a factor of two, and possibly even more with cache effects. Since the application does not know that a processor was preempted, it cannot adapt its decomposition for the available number of processors, so each step is similarly delayed until the processor is returned.

- **Producer-consumer delay.** Some parallel applications use a producer-consumer design pattern, where the results of one thread are fed to the next thread, and the output of that thread is fed onward, as in Figure 7.9. Preempting a thread in the middle of a producer-consumer chain can stall all of the processors in the chain.

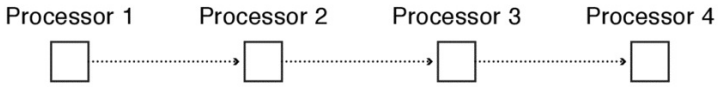

Figure 7.9: Producer-consumer design pattern for a parallel program. Preempting one stage can stall the remainder.

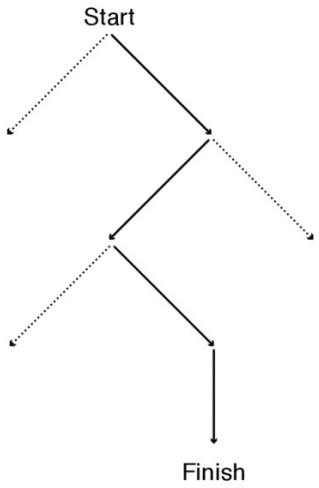

Figure 7.10: Critical path of a parallel program; delays on the critical path increase execution time.

- **Critical path delay.** More generally, parallel programs have a *critical path* — the minimum sequence of steps for the application to compute its result. Figure 7.10 illustrates the critical path for a fork-join parallel program. Work off the critical path can occur in parallel, but its precise scheduling is less important. Preempting a thread on the critical path, however, will slow down the end result. Although the application programmer may know which parts of the computation are on the critical path, with oblivious scheduling, the operating system will not; it will be equally likely to preempt a thread on the critical path as off.
- **Preemption of lock holder.** Many parallel programs use locks and condition variables for synchronizing their parallel execution. Often, to reduce the cost of acquiring locks, parallel programs will use a "spin-then-wait" strategy — if a lock is busy, the waiting thread spin-waits briefly for it to be released, and if the lock is still busy, it blocks and looks for other work to do. This can reduce overhead in the common case that the lock is held for only short periods of time. With oblivious scheduling, however, the lock holder can be preempted — other tasks will spin-then-wait until the lock holder is re-scheduled, increasing overhead.
- **I/O.** Many parallel applications do I/O, and this can cause problems if the operating system scheduler is oblivious to the application decomposition into parallel work. If a read or write request blocks in the kernel, the thread blocks as well. To reuse the processor while the thread is waiting, the application program must have created more threads than processors, so that the scheduler can have an extra one to run in place of the blocked thread. However, if the thread does not block (e.g., on a file read when the file is cached in memory), that means that the scheduler has more threads than processors, and so needs to do time slicing to multiplex threads onto processors — causing all of the problems we have listed above.

Gang Scheduling

One possible approach to some of these issues is to schedule all of the tasks of a program together. This is called *gang scheduling*. The application picks some decomposition of work into some number of threads, and those threads run either together or not at all. If the operating system needs to schedule a different application, if there are insufficient idle resources, it preempts all of the processors of an application to make room. Figure 7.11 illustrates an example of gang scheduling.

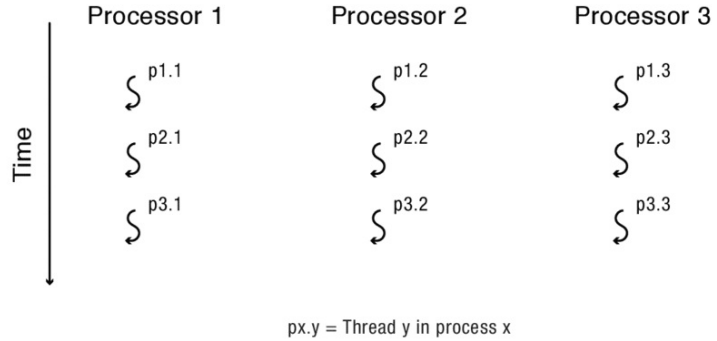

Figure 7.11: With gang scheduling, threads from the same process are scheduled at exactly the same time, and they are time sliced together to provide a chance for other processes to run.

Because of the value of gang scheduling, commercial operating systems, such as Linux, Windows, and MacOS, have mechanisms for dedicating a set of processors to a single application. This is often appropriate on a server dedicated to a single primary use, such as a database needing precise control over thread assignment. The application can *pin* each thread to a specific processor and (with the appropriate permissions) mark it to run with high priority. The system reserves a small subset of the processors to run other applications, multiplexed in the normal way but without interfering with the primary application.

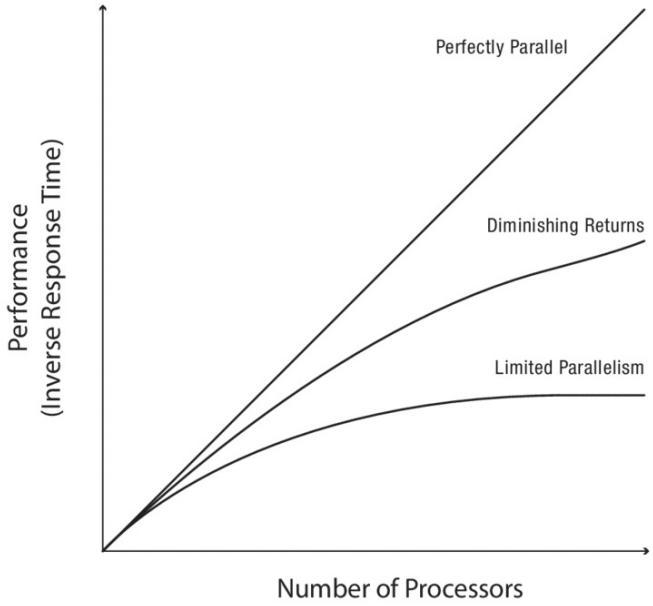

Figure 7.12: Performance as a function of the number of processors, for some typical parallel applications. Some applications scale linearly with the number of processors; others achieve diminishing returns.

For multiplexing multiple parallel applications, however, gang scheduling can be inefficient. Figure 7.12 illustrates why. It shows the performance of three example parallel programs as a function of the number of processors assigned to the application. While some applications have perfect speedup and can make efficient use of many processors, other applications reach a point of diminishing returns, and still others have a maximum parallelism. For example, if adding processors does not decrease the time spent on the program's critical path, there is no benefit to adding those resources.

An implication of Figure 7.12 is that it is usually more efficient to run two parallel programs each with half the number of processors, than to time slice the two programs, each gang scheduled onto all of the processors. Allocating different processors to different tasks is called *space sharing*, to differentiate it from time sharing, or time slicing — allocating a single processor among multiple tasks by alternating in time when each is scheduled onto the processor. Space sharing on a multiprocessor is also more efficient in that it minimizes processor context switches: as long as the operating system has not changed the allocation, the processors do not even need to be time sliced. Figure 7.13 illustrates an example of space sharing.

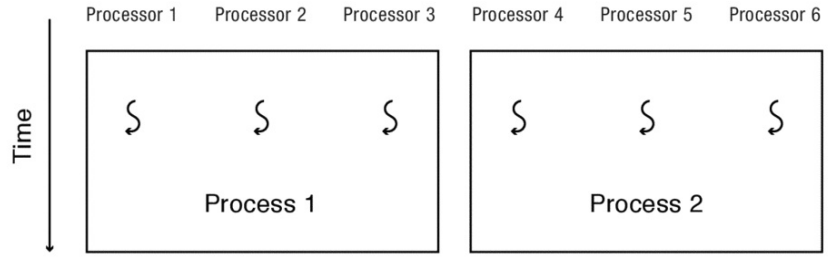

Figure 7.13: With space sharing, each process is assigned a subset of the processors.

Space sharing is straightforward if all tasks start and stop at the same time; in that case, we can just allocate evenly. However, the number of available processors is often a dynamic property in a multiprogrammed setting, because tasks start and stop at irregular intervals. How does the application know how many processors to use if the number changes over time?

Scheduler Activations

A solution, recently added to Windows, is to make the assignment and re-assignment of processors to applications visible to applications. Applications are given an execution context, or *scheduler activation*, on each processor assigned to the application; the application is informed explicitly, via an upcall, whenever a processor is added to its allocation or taken away. Blocking on an I/O request also causes an upcall to allow the application to repurpose the processor while the thread is waiting for I/O.

As we noted in Chapter 4, user-level thread management is possible with scheduler activations. The operating system kernel assigns processors to applications, either evenly or according to some priority weighting. Each application then schedules its user-level threads onto the processors assigned to it, changing its allocation as the number of processors varies due to external events such as other processes starting or stopping. If no other application is running, an application can use all of the processors of the machine; with more contention, the application must remap its work onto a smaller number of processors.

Scheduler activations defines a *mechanism* for informing an application of its processor allocation, but it leaves open the question of the *multiprocessor scheduling policy*: how many processors should we assign each process? This is an open research question. As we explained in our discussion of uniprocessor scheduling policies, there is a fundamental tradeoff between policies (such as Shortest Job First) that improve average response time and those (such as max-min fairness) that attempt to achieve fair allocation of resources among different applications. In the multiprocessor setting, average response time may be improved by giving extra resources to parallel interactive tasks provided this did not cause long-running compute intensive parallel tasks to starve for resources.

7.3 Energy-Aware Scheduling

Another important consideration for processor scheduling is its impact on battery life and energy use. Laptops and smartphones compete on the basis of battery life, and even for servers, energy usage is a large fraction of the overall system cost. Choices that the operating system makes can have a large effect on these issues.

One might think that processor scheduling has little role to play with respect to system energy usage. After all, each application has a certain amount of computing that needs to be done, computing that requires energy whether we are running on a direct power line or off of a battery. Of course, the operating system should delay background or system maintenance tasks (such as software upgrades) for when the system is connected to power, but this is likely to be a relatively minor effect on the overall power budget.

In part because of the importance of battery life to computer users, modern architectures have developed a number of ways of trading reduced computation speed for lower energy use. In other words, the mental model of each computation taking a fixed amount of energy is no longer accurate. There is quite a bit of flux in the types of hardware support available on different systems, and systems five years from now are likely to make very different tradeoffs than those in place today. Thus, our goal in this section is not to provide a set of widely used algorithms for managing power, but rather to outline the design issues energy management poses for the operating system.

Several power optimizations are possible, provided hardware support:

- **Processor design.** There can be several orders of magnitude difference between one processor design and another with respect to power consumption. Often, making a processor faster requires extra circuitry, such as out of order execution, that itself consumes power; low power processors are slower and simpler. Likewise, processors designed for lower clock speeds can tolerate lower voltage swings at the circuit level, reducing power consumption dramatically. Some systems have begun to put this tradeoff under the control of the operating system, by including both a high power, high performance multiprocessor and a low power, lower performance uniprocessor on the same chip. High power is appropriate when response time is at a premium and low power when power consumption is more important.
- **Processor usage.** For systems with multiple processor chips, or multiple cores on a single chip, lightly used processors can be disabled to save power. Processors will typically draw much less power when they are completely idle, but as we mentioned above, many parallel programs achieve some benefit from using extra processors, yet also reach a point of diminishing returns. Thus, there is a tradeoff between somewhat faster execution (e.g., by using all available resources) and lower energy use (e.g., by turning off some processors even when using them would slightly decrease response time).
- **I/O device power.** Devices not in use can be powered off. Although this is most obvious in terms of the display, devices such as the WiFi or cellphone network interface also consume large amounts of power. Power-constrained embedded systems such as sensors will turn on their network interface hardware periodically to send or receive data, and then go back to quiescence. For this to work, the senders and receivers need to synchronize their periods of transmission, or the hardware needs to have a low power listening mode.

Heat dissipation

A closely related topic to energy use is heat dissipation. In laptop computers, you can save weight by not including a fan to cool the processor. However, a modern multicore chip will consume up to 150 Watts, or more than a very bright incandescent light bulb. Just as with a light bulb, the heat generated has to go somewhere. Making things significantly more complicated, the processor will also break permanently if it runs at too high a temperature. Thus, the operating system increasingly must monitor and manage the temperature of the processor to ensure it stays within its operating region. Much like a cheetah, portable computers are now capable of running at very fast speeds for short periods of time, before they need to take a break to cool down. Or they can amble at much slower speeds for a longer period of time.

The laptop one of us used to write this book illustrates this. Formatting this textbook takes about a minute when the computer is cold, but the same formatting request will stall in the middle of the build for several minutes if run immediately after a previous build request.

At times, different power optimizations interact in subtle ways. For example, running application code quickly can sometimes improve power efficiency, by enabling the network interface hardware to be turned off more quickly once the application finishes. Because context switching consumes both time and energy to reload processor caches, affinity scheduling improves both performance and energy efficiency.

In most cases, however, there is a tradeoff: how should the operating system balance between competing demands for timeliness and energy efficiency? If the user has requested maximum responsiveness or maximum battery life, the choice is easy, but often the user wants a reasonable tradeoff between the two.

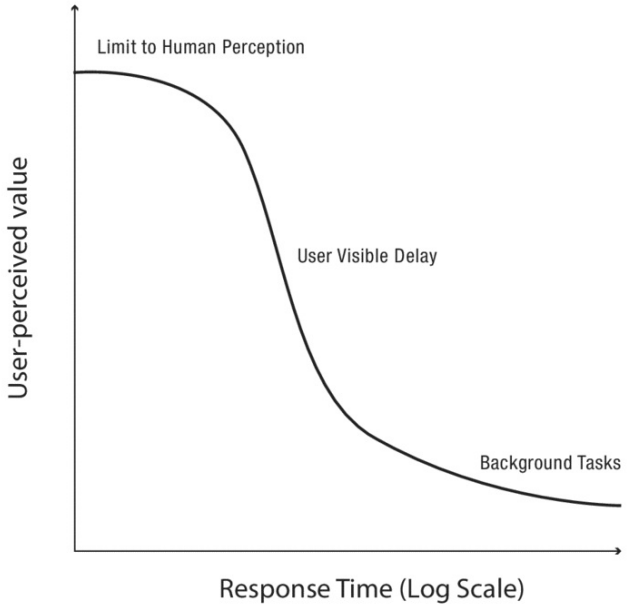

Figure 7.14: Example relationship between response time and user-perceived value. For most applications, faster response time is valuable within a range. Below some threshold, users will not be able to perceive the difference. Above some threshold, users will perform other activities while waiting for the result.

One approach would be to consider the value that the user places on fast response time for a particular application: quickly updating the display after a user interface command is probably more important than transferring files quickly in the background. We can capture the relationship between response time and value in Figure 7.14. Although the precise shape and magnitude will vary from user to user and application to application, the curve will head down and to the right — the longer something takes, the less useful it is. Often, the curve is S-shaped. Human perception is unable to tell the difference between a few tens of milliseconds, so adding a short delay will not matter that much for most tasks; likewise, if a protein folding computation has already taken a few minutes, it won't matter much if it takes a few more seconds. Not everything will be S-shaped: in high frequency stock trading, value starts high and plummets to zero within a few milliseconds.

Response time predictability affects this relationship as well. An online video that cuts out for a few seconds every minute is much less watchable than one that is lower quality on average but more predictable.

If we combine Figure 7.14 with the fact that increased energy use often provides diminishing returns in terms of improved performance, this suggests a three prong strategy to spend the system's energy budget where it will make the most difference:

- **Below the threshold of human perception.** Optimize for energy use, to the extent that tasks can be executed with greater energy efficiency without the user noticing.
- **Above the threshold of human perception.** Optimize for response time if the user will notice any slowdown.
- **Long-running or background tasks.** Balance energy use and responsiveness depending on the available battery resources.

Battery life and the kernel-user boundary

An emerging issue on smartphones is that application behavior can have a significant impact on battery life, e.g., by more intensive use of the network or other power-hungry features of the architecture. If a user runs a mix of applications, how can she know which was most responsible for their smartphone running out of power? Among the resources we will discuss in this book, energy is almost unique in being a *non-virtualizable* resource. When an application drains the battery, the energy lost is no longer available to any other applications.

How can we prevent a misbehaving or greedy application from using more than its share of the battery? One model is to let the user decide: for the kernel to measure and record how much energy was used by each application, so the user can determine if each application is worth it. Apple has taken a different approach with the iPhone. Because Apple controls which applications can run on the system, it can and has barred applications that (in its view) unnecessarily drain the battery. It will be interesting to see which of these models wins out over time.

7.4 Real-Time Scheduling

On some systems, the operating system scheduler must account for process deadlines. For example, the sensor and control software to manage an airplane's flight path must be executed in a timely fashion, if it is to be useful at all. Similarly, the software to control anti-lock brakes or anti-skid traction control on an automobile must occur at a precise time if it is to be effective. In a less life critical domain, when playing a movie on a computer, the next frame must be rendered in time or the user will perceive the video quality as poor.

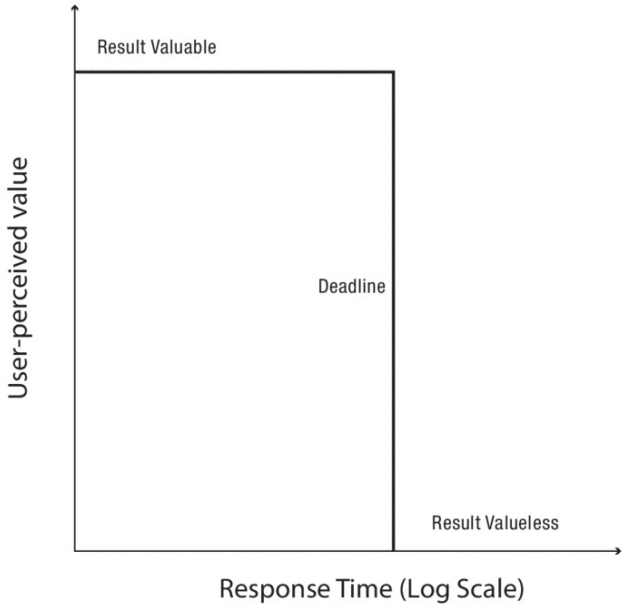

Figure 7.15: With real-time constraints, the value of completing some task drops to zero if the deadline is not met.

These systems have *real-time constraints*: computation that must be completed by a deadline if it is to have value. Real-time constraints are a special case of Figure 7.14, shown in Figure 7.15, where the value of completing a task is uniform up to the deadline, and then drops to zero.

How do we design a scheduler to ensure deadlines are met?

We might start by assigning real-time tasks a higher priority than any less time critical tasks. We could then run the system under a variety of different of different workloads, and see if the system continues to comfortably meet its deadlines in all cases. If not, the system may need a faster processor or other hardware resources to speed up the real-time tasks.

Unfortunately, testing alone is insufficient for guaranteeing real-time constraints. Recall that the specific ordering of execution events can sometimes lead to different execution sequences — e.g., sometimes a thread will need to wait for a lock held another thread, and other times the lock will be FREE.

One option is that, instead of threads, we should use a completely deterministic and repeatable schedule that ensures that the deadlines are met. This can work if the real-time tasks are periodic and fixed in advance. However, in dynamic systems, it is difficult to account for all possible variations affecting how long different parts of the computation will take.

There are three widely used techniques for increasing the likelihood that threads meet their deadlines. These approaches are also useful whenever timeliness matters without a strict deadline, e.g., to ensure responsiveness of a user interface.

- **Over-provisioning.** A simple step is to ensure that the real-time tasks, in aggregate, use only a fraction of the system's processing power. This way, the real-time tasks will be scheduled quickly, without having to wait for higher-priority, compute-intensive tasks. The equivalent step in college is to avoid signing up for too many hard courses in the same semester!
- **Earliest deadline first.** Careful choice of the scheduling policy can also help meet deadlines. If you have a pile of homework to do, neither shortest job first nor round robin will ensure that the assignment due tomorrow gets done in time. Instead, real-time schedulers, mimicking real life, use a policy called *earliest deadline first* (EDF). EDF sorts tasks by their deadline and performs them in that order. If it is possible to schedule the required work to meet their deadlines, and the tasks only need the processor (and not I/O, locks or other resources), EDF will ensure that all tasks are done in time.

  For complex tasks, however, EDF can produce anomalous behavior. Consider two tasks. Task A is I/O-bound with a deadline at 12 ms, needing 1 ms of computation followed by 10 ms of I/O. Task B is compute-bound with a deadline at 10 ms, but needing 5 ms of computation. Although there is a schedule that will meet both deadlines (run task A first), EDF will run the compute-bound task first, causing the I/O-bound task to miss its deadline.

  This limitation can be addressed by breaking tasks into shorter units, each with its own deadline. In the example, the true deadline for the compute portion of the I/O-bound task is at 2 ms, because if it is not completed by then, the overall task deadline will be missed. If your homework next week needs a book from the library, you need to put that on hold first, even if that slightly delays the homework you have due tomorrow.
- **Priority donation.** Another problem can occur through the interaction of shared data structures, priorities, and deadlines. Suppose we have three tasks, each with a different priority level. The real-time task runs at the highest priority, and it has sufficient processing resources to meet its deadline, with some time to spare. However, the three tasks also access a shared data structure, protected by a lock.

  Suppose the low priority acquires the lock to modify the data structure, but it is then preempted by the medium priority task. The relative priorities imply that we should run the medium priority task first, even though the low priority task is in the middle of a critical section. Next, suppose the real-time task preempts the medium task and proceeds to access the shared data structure. It will find the lock busy and wait. Normally, the wait would be short, and the real-time task would be able to meet its deadline despite the delay. However, in this case, when the high priority task waits for the lock, the scheduler will pick the medium priority task to run next, causing an indefinite delay. This is called *priority inversion*; it can occur whenever a high

priority task must wait for a lower priority task to complete its work.

A commonly used solution, implemented in most commercial operating systems, is called *priority donation*: when a high priority task waits on a shared lock, it temporarily donates its priority to the task holding the lock. This allows the low priority task to be scheduled to complete the critical section, at which point its priority reverts to its original state, and the processor is re-assigned to the high priority, waiting, task.

7.5 Queueing Theory

Suppose you build a new web service, and the week before you are to take it live, you test it to see whether it will have reasonable response time. If your tests show that the performance is terrible, what then? Is it because the implementation is too slow? Perhaps you have the wrong scheduler? Quick, let's re-implement that linked list with a hash table! And add more levels to the multi-level feedback queue! Our advice: don't panic. In this section, we consider a third possibility, an effect that often trumps all of the others: response time depends non-linearly on the rate that tasks arrive at a system. Understanding this relationship is the topic of *queueing theory*.

Fortunately, if you have ever waited in line (and who hasn't?), you have an intuitive understanding of queueing theory. Its concepts apply whenever there is a queue waiting for a turn, whether it is tasks waiting for a processor, web requests waiting for a turn at a web server, restaurant patrons waiting for a table, cars waiting at a busy intersection, or people waiting in line at the supermarket.

While queueing theory is capable of providing precise predictions for complex systems, our interest is providing you the tools to be able to do back of the envelope calculations for where the time goes in a real system. For performance debugging, coarse estimates are often enough. For this reason, we make two simplifying assumptions for this discussion. First, we assume the system is work-conserving, so that all tasks that arrive are eventually serviced; this will normally be the case except in extreme overload conditions, a topic we will discuss in the next section of this chapter. Second, although the scheduling policy can affect a system's queueing behavior, we will keep things simple and assume FIFO scheduling.

7.5.1 Definitions

Because queueing theory is concerned with the root causes of system performance, and not just its observable effects, we need to introduce a bit more terminology. A simple abstract queueing system is illustrated by Figure 7.16. In any queueing system, tasks arrive, wait their turn, get service, and leave. If tasks arrive faster than they can be serviced, the queue grows. The queue shrinks when the service rate exceeds the arrival rate.

To begin, we will consider single-queue, single-server, work-conserving systems. Later, we will introduce more complexity such as multiple queues, multiple servers, and finite queues that can discard some requests.

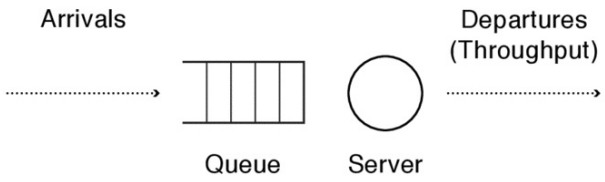

Figure 7.16: An abstract queueing system. Tasks arrive, wait their turn in the queue, get service, and leave.

- **Server.** A server is anything that performs tasks. A web server is obviously a server, performing web requests, but so is the processor on a client machine, since it executes application tasks. The cashier at a supermarket and a waiter in a restaurant are also servers.
- **Queueing delay (W) and number of tasks queued (Q).** The queueing delay, or wait time, is the total time a task must wait to be scheduled. In a time slicing system, a task might need to wait multiple times for the same server to complete its task; in this case the queueing delay includes all of the time a task spends waiting until it is completed.
- **Service time (S).** The service time S, or execution time, is the time to complete a task assuming no waiting.
- **Response time (R).** The response time is the queueing delay (how long you wait in line) plus the service time (how long it takes once you get to the front of the line).

$$R = W + S$$

In the web server example we started with, the poor performance can be due to either factor — the system could be too slow even when no one is waiting, or the system could be too slow because each request spends most of its time waiting for service.

We can improve the response time by improving either factor. We can reduce the queueing delay by buying more servers (for example, by having more processors than ready threads or more cashiers than customers), and we can reduce service time by buying a faster server or by engineering a faster implementation.

- **Arrival rate (λ) and arrival process.** The arrival rate λ is the average rate at which new tasks arrive.

  More generally, the arrival process describes when tasks arrive including both the average arrival rate and the pattern of those arrivals such as whether arrivals are bursty or spread evenly over time. As we will see, burstiness can have a large impact

on queueing behavior.

- **Service rate (μ).** The service rate μ is the number of tasks the server can complete per unit of time when there is work to do. Notice that the service rate μ is the inverse of the service time S.
- **Utilization (U).** The utilization is the fraction of time the server is busy ($0 \le U \le 1$). In a work-conserving system, utilization is determined by the ratio of the average arrival rate to the service rate:

$$U = \lambda / \mu \text{ if } \lambda < \mu$$
$$= 1 \quad \text{if } \lambda \ge \mu)$$

Notice that if $\lambda > \mu$, tasks arrive more quickly than they can be serviced. Such an overload condition is unstable; in a work-conserving system, the queue length and queueing delay grow without bound.

- **Throughput (X).** Throughput is the number of tasks processed by the system per unit of time. When the system is busy, the server processes tasks at the rate of μ, so we have:

$$X = U \mu$$

Combining this equation with the previous one, we can see that when the average arrival rate λ is less than the service rate μ, the system throughput matches the arrival rate. We can also see that the throughput can never exceed μ no matter how quickly tasks arrive.

$$X = \lambda \text{ if } U < 1$$
$$= \mu \text{ if } U = 1$$

- **Number of tasks in the system (N).** The average number of tasks in the system is just the number queued plus the number receiving service:

$$N = Q + U$$

7.5.2 Little's Law

Little's Law is a theorem proved by John Little in 1961 that applies to any *stable system* where the arrival rate matches the departure rate. It defines a very general relationship between the average throughput, response time, and the number of tasks in the system:

$$N = X R$$

Although this relationship is simple and intuitive, it is powerful because the "system" can be anything with arriving and departing tasks, provided the system is stable — regardless of the arrival process, number of servers, or queueing order.

EXAMPLE: Suppose we have a queueing system like the one shown in Figure 7.16 and we observe over the course of an hour that an average of 100 requests arrive and depart each second and that the average request is completed 50 ms after it arrives. On average, how many requests are being handled by the system?

ANSWER: Since the arrival rate matches the departure rate, the system is stable and we can use Little's Law. We have a throughput X = 100 requests/second and a response time R = 50 ms = 0.05 seconds:

$$N = X R$$
$$= 100 \text{ requests/second} \times 0.05 \text{ seconds}$$
$$= \mathbf{5 \text{ requests}}$$

In this system there are, on average, 5 requests waiting in the queue or being served. □

We can also zoom in to see what is happening at the server, ignoring the queue. The server itself is a system, and Little's Law applies there, too.

EXAMPLE: Suppose we have a server that processes one request at a time and we observe that an average of 100 requests arrive and depart each second and that the average request completes 5 ms after it arrives. What is the average utilization of the server?

ANSWER: The utilization of the server is the fraction of time the server is busy processing a request. Because the server handles one request at a time, its utilization equals the average number of requests in the server-only system. Using Little's Law:

$$U = N = X R$$

$$= 100 \text{ requests/second} \times 0.005 \text{ seconds}$$

$$= \mathbf{0.5 \text{ requests}}$$

The average utilization is 0.5 or 50%. □

We can also look at the subsystem comprising just the queue.

EXAMPLE: For the system described in the previous two examples, how long does an average request spend in the queue, and on average how many requests are in the queue?

ANSWER: We know that an average task takes 50 ms to get through the queue and server and that it spends 5 ms at the server, so it must spend 45 ms in the queue. Similarly, we know that on average the system holds 5 tasks with 0.5 of them in the server, so the average queue length is 4.5 tasks.

We can get the same result with Little's Law. One hundred tasks pass through the queue per second and spend an average of 45 ms in the queue, so the average number of tasks in the queue is:

$$N = X R$$

$$= 100 \text{ requests/second} \times 0.045 \text{ seconds}$$

$$= \mathbf{4.5 \text{ requests}}$$

□

Although Little's Law is useful, remember that it only provides information about the system's averages over time.

EXAMPLE: One thing might puzzle you. In the previous example, if the average number of tasks in the queue is 4.5 and processing a request takes 5 ms, how can the average queueing delay for a request be 45 ms rather than 4.5 × 5 ms = 22.5 ms?

ANSWER: The *average* number of requests in the queue is 4.5. Sometimes there are more; sometimes there are fewer. Queues will grow during bursts of arrivals, and they will shrink when tasks are arriving slowly.

In fact, from the 0.5 server utilization rate calculated above, we know that the queue is empty half the time. To make up for the empty periods, there *must* be periods with longer-than-average queue lengths.

Unfortunately, the queues tend to be full during busy periods and they tend to be empty during idle periods, so relatively few requests enjoy short or empty queues and relatively many suffer long queues. So, the average request sees a longer queue than the average queue length over time might suggest, and the (per-request) average queueing delay exceeds the (time) average queue length times the (per-request) average service time. □

Not only can we apply Little's Law to a simple queueing system or its subcomponents, we can apply it to more complex systems, even those whose internal structure we do not fully understand.

EXAMPLE: Suppose there is a complex web service like Google, Facebook, or Amazon, and we know that the average request takes 100 milliseconds and that the service handles an average of 10,000 queries per second. How many requests are pending in the system on average?

ANSWER: Applying Little's Law:

$$N = X R$$

$$= 10000 \text{ requests/second} \times 0.1 \text{ seconds}$$

$$= \mathbf{1000 \text{ requests}}$$

Note that this is true regardless of the internal structure of the web service. It may have many load balancers, processors, network switches, and databases, each with separate queues, and each with different queueing policies, but in aggregate in steady state the number of requests being handled must be equal to the product of the response time and the throughput. □

7.5.3 Response Time Versus Utilization

Because having more servers (whether processors on chip or cashiers in a supermarket) or faster servers is costly, you might think that the goal of the system designer is to maximize utilization. However, in most cases, there is no free lunch: as we will see, higher utilization normally implies higher queueing delay and higher response times.

Operating a system at high utilization also increases the risk of overload. Suppose you plan to minimize costs by operating a web site at 95% utilization, but your service turns out to be a little more popular than you expected. You can quickly find yourself operating

in the unstable regime where requests are arriving faster than you can service them ($\lambda > \mu$) and where your queues and waiting times are growing without bound.

As a designer, you need to find an appropriate tradeoff between higher utilization and better response time. Fifty years ago, computer designers made the tradeoff in favor of higher utilization: when computers are wildly expensive, it is annoying but understandable to make people wait for the computer. Now that computers are much cheaper, our lives are better! We now usually make the computer do the waiting.

We can predict a queueing system's average response time from its arrival process and service time, but the relationship is more complex than the relationships discussed so far.

To provide intuition, we start with some extreme scenarios that bound the behavior of a queueing system; we will introduce more realistic scenarios as we proceed.

Broadly speaking, higher arrival rates and burstier arrival patterns tend to yield longer queue lengths and response times than lower arrival rates and smoother arrival patterns.

Best case: Evenly spaced arrivals. Suppose we have a set of fixed-sized tasks that arrive equally spaced from one another. For As long as the rate at which tasks arrive is less than the rate at which the server completes the tasks, there will be no queueing at all. Perfection! Each server finishes the previous customer in time for the next arrival.

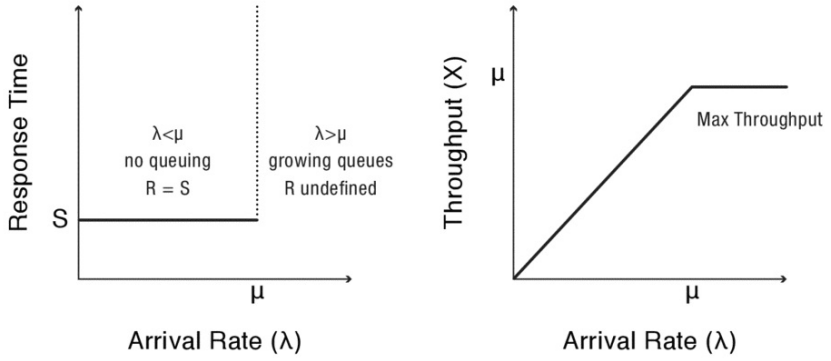

Figure 7.17: Best case response time and throughput as a function of the task arrival rate relative to the service rate. These graphs assume arrivals are evenly spaced and service times are fixed-size.

Figure 7.17 illustrates the relationship between arrival rate and response time for this best case scenario of evenly spaced arrivals. There are three regimes:

- **$\lambda < \mu$.** If the arrival rate is below the service rate, there is no queueing and the response time equals the service time.

For example, suppose we have a server that can handle 1000 requests per second, and one request arrives every 1000, 100, or 10 milliseconds. The server finishes processing request i- 1 before request i arrives, and request i completes 1 ms after it arrives, clearing the way for request i + 1.

The situation remains the same if arrivals are more closely spaced at 1.1, 1.01, 1.001, and so on down to 1.0 ms, where each request arrives at the moment the previous request completes.

- **$\lambda = \mu$.** If the arrival rate matches the service rate, the system is in a precarious equilibrium. If the queues are initially empty, they will stay empty, but if the queues are initially full, they will remain full.

Suppose arrivals are coming every 1.0 ms, and at some point during the day a single extra request arrives; that request must wait until the previous one completes, but the server will then be busy when the next request arrives. That single extra request produces queueing delay for every subsequent request.

- **$\lambda > \mu$.** If the arrival rate exceeds the service rate, queues will grow without bound. In this case, the system is not in equilibrium, and the steady state response time is undefined.

Suppose the task arrival rate is one per 0.999 ms so that tasks arrive slightly faster than they can be processed? If a system's arrival rate exceeds its service rate, then under our simple model its queues will grow without bound, and its queueing delay is undefined. In practice, memory is finite; once the queue's capacity is reached, the system must discard some of the arriving requests.

Figure 7.17 also shows the relationship between arrival rate and throughput. When the arrival rate is less than the service rate, increasing the arrival rate increases throughput. Once the arrival rate matches or exceeds the service rate, faster arrivals just grow the queues more quickly, they do not increase useful throughput.

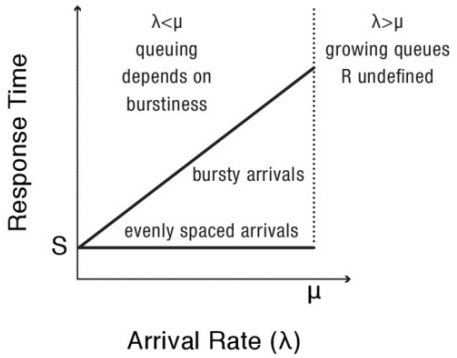

Figure 7.18: Response time for a server that can handle 10 requests per second as we vary arrival rate of fixed-size tasks in two scenarios: evenly spaced arrivals and bursty arrivals where all of a second's requests arrive in a group at the start of the second.

Worst case: Bursty arrivals. Now consider the opposite case. Suppose a group of tasks arrive at exactly the same time. The average wait time increases linearly as more tasks arrive together — one task in a group can be serviced right away, but others must wait.

Figure 7.18 considers a hypothetical server with a maximum throughput of 10 tasks per second as we vary the number of tasks that arrive per second. The graph shows two cases: one where requests are evenly spaced as in Figure 7.17 and the other where requests arrive in a burst at the start of each second.

Even when the request rate is below the server's service rate, bursty arrivals suffer queueing delays. For example, when five requests arrive as a group at the start of each second, the first request is served immediately and finishes 0.1 seconds later. The server can then start processing the second request, finishing it 0.2 seconds after the start of the interval. The third, fourth, and fifth requests finish at 0.3, 0.4, and 0.5 seconds after the start of the second, giving an average response time of $(0.1 + 0.2 + 0.3 + 0.4 + 0.5)/5 = 0.3$ seconds. By the same logic, if ten requests arrive as a group, the average response time is $(0.1 + 0.2 + 0.3 + 0.4 + 0.5 + 0.6 + 0.7 + 0.8 + 0.9 + 1.0)/10 = 0.55$ seconds. If the same requests had arrived evenly spaced, their average response time would have been over five times better!

Exponential arrivals. Most systems are somewhere in between this best case and worst case. Rather than being perfectly synchronized or perfectly desynchronized, task arrivals in many systems are random. For example, different customers in a supermarket do not coordinate with each other as to when they arrive.

Likewise, service times are not perfectly equal — there is randomness there as well. At a doctor's office, everyone has an appointment, so it may seem like that should be the best case scenario, and no one should ever have to wait. Even so, there is often queueing! Why? If the amount of time the doctor takes with each patient is sometimes shorter and sometimes longer than the appointment length, then random chance will cause queueing.

A particularly useful model for understanding queueing behavior is to use an exponential distribution to describe the time between tasks arriving and the time it takes to service each task. Once you get past a bit of math, the exponential provides a stunningly simple *approximate* description of most real-life queueing systems. We do not claim that all real systems always obey the exponential model in detail; in fact, most do not. However, the model is often accurate enough to provide insight on system behaviors, and as we will discuss, it is easy to understand the circumstances under which it is inaccurate.

Model vs. reality

When trying to understand a complex system, it is often useful to construct a model of its behavior. A model is a simplification that tries to capture the most important aspects of a more complex system's behavior. Models are neither true nor false, but they can be useful or not for a particular purpose. It is often the case that a more complex model will yield a closer approximation; whether the added complexity is useful or gets in the way depends on how the model is being used.

We often find it useful to use simple workload models when debugging early system implementations. Using the types of analysis described in this chapter and an understanding of the system being built, it is usually possible to predict how the system should behave under simple workloads. If measured behavior deviates from these predictions, there is a bug in our understanding or implementation of the system. Simple workloads can help us improve our understanding if it is the former and track down the bug if it is the latter.

We could, instead, evaluate early implementations by feeding them more realistic workloads. For example, if we are building a new web server, we could feed it a workload trace captured at some other server. However, this approach is often more complex. For example, to test our system under a range of conditions, we need to gather a range of traces — some with low load, some with high; some with bursty loads, some with smooth; etc.

Worst, even though this approach is more complex, it may yield less insight because it is harder to predict the expected system behavior. If we run a simulation with a trace and get worse performance than we expected, is it because we do not understand our system or because we do not understand the trace?

This is not to suggest that simple models are always superior to more complex, more realistic ones. Once we are satisfied with our new system's behavior for workloads we understand, we should test it for workloads we do not understand or control. There may be (and probably are) important behaviors not captured in our simple models. We might find, for example, that bursts of interest in particular topics create "hot spots" of load that we did not anticipate. Evaluation under more realistic models might make us realize that we need to implement more aggressive caching of recently popular pages.

Selecting the right model for system evaluation is a delicate balance between complexity and accuracy. If after abstracting away detail, we can still provide approximately correct predictions of system behavior under a variety of scenarios, then it is likely the model captures the most important aspects of the system. If the model is inaccurate in some important respect, then it means our explanation for how the system behaves is too coarse, and to improve the prediction we need to revise the model.

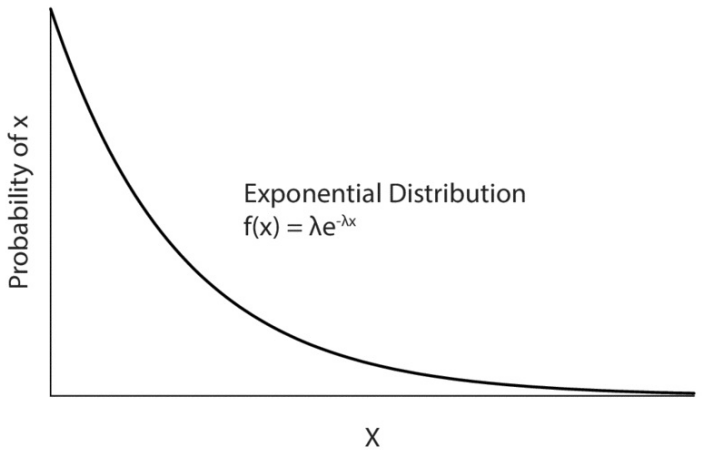

Figure 7.19: Exponential probability distribution.

First, the math. An *exponential distribution* of a continuous random variable with a mean of 1 / λ has the probability density function, shown in Figure 7.19:

$$f(x) = \lambda e^{-\lambda x}$$

Fortunately, you need not understand that equation in any detail, except for the following. A useful property of an exponential distribution is that it is *memoryless*. A memoryless distribution for the time between two events means that the likelihood of an event occurring remains the same, *no matter how long we have already waited* for the event, or what other events may have already happened. For example, on a web server, web requests from different users (usually) arrive independently. Sometimes, two requests will arrive close together in time; sometimes there will be more of a delay. For example, suppose a web server receives a request from a new user on average every 10 ms. If you want to predict how long until the next request arrives, it probably does not matter when the *last* request arrived: 0, 1, 5, or 50 ms ago. The expected time to the next request is still probably about 10 ms.

Not every distribution is memoryless. A Gaussian, or normal, distribution for the time between events is closer to the best case scenario described above — arrivals occur randomly, but they tend to occur at regular intervals, give or take a bit.

Some probability distributions work the other way. With a *heavy-tailed distribution*, the longer you have waited for some event, the longer you are likely to still need to wait. This is closer to the worst case behavior above, as it means that most events are clustered together.

For example, a ticket seller's web site might see bursty workloads. For long periods of time the site might see little traffic, but when tickets for a popular concert of sporting event go on sale, the traffic may be overwhelming. Here, external factors introduce synchronization across different users' activities so that requests from different users do not arrive independently. Such a workload is unlikely to be memoryless; if you look at a ticket seller's web site at a random moment and see that it has been a long time since the last request arrived, you probably arrived during a lull, and you can predict that it will likely be a long time until the next request arrives. On the other hand, if the last request just arrived, you probably arrived during a burst, and the next request will arrive soon.

With a memoryless distribution, the behavior of queueing systems becomes simple to understand. One can think of the queue as a finite state machine: with some probability, a new task arrives, increasing the queue by one. If the queue length is non-zero, with some other probability, a task completes, decreasing the queue by one. With a memoryless distribution of arrivals and departures, the probability of each transition is constant and independent of the other transitions, as illustrated in Figure 7.20.

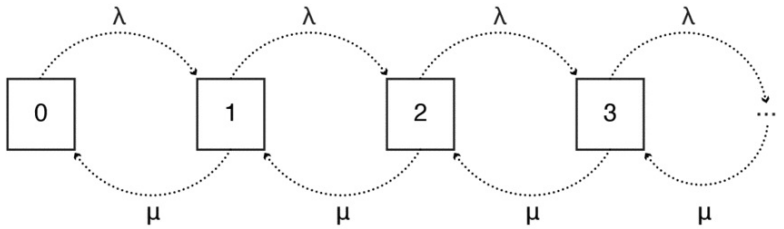

Figure 7.20: State machine representing a queue with exponentially distributed arrivals and departures. λ is the rate of arrivals; μ is the rate at which the server completes each task. With an exponential distribution, the probability of a state transition is independent of how long the system has been in any given state.

Assuming that λ < μ, the system is stable Assuming stability and exponential distributions for the arrival and departure processes, we can solve the model to determine the average response time R as a function of the utilization U and service time S:

$$R = S / (1 - U)$$

Recall that the utilization, the fraction of time that the server is busy, is simply the ratio between λ and μ.

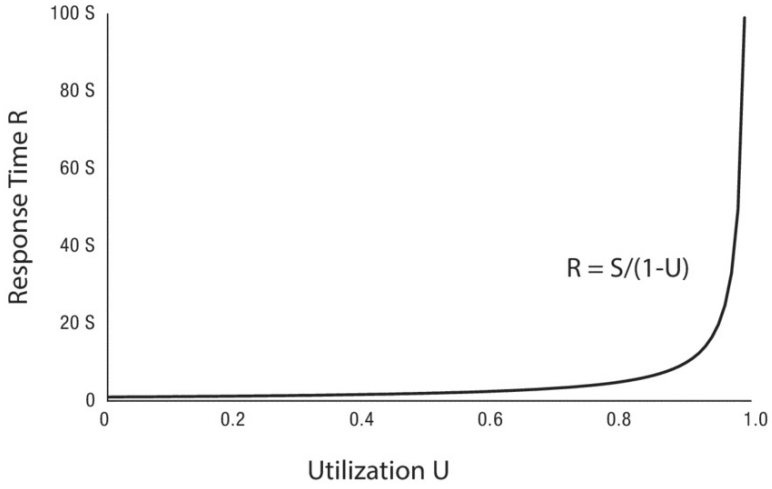

Figure 7.21: Relationship between response time and utilization, assuming exponentially distributed arrivals and departures. Average response time goes to infinity as the system approaches full utilization.

This equation is graphed in Figure 7.21. When utilization is low, there is little queueing delay and response time is close to the service time. Furthermore, when utilization is low, small increases in the arrival rate result in small increases in queueing delay and response time.

As utilization increases, queueing and response time also increase, and the relationship is non-linear. At high utilizations, the queueing delay is high, and small increases in the arrival rate can drastically increase queueing delay and response time.

EXAMPLE: Suppose a queueing system with exponentially distributed arrivals and task sizes is 20% utilized and the load increases by 5%, by how much does the response time increase? How does that increase compare to the case when utilization goes from 90% to 95%?

ANSWER: At 20% utilization, the response time is:

$$R = S / (1 - U)$$
$$= S / (1 - 0.2)$$
$$= 1.25\ S$$

At 25% utilization, the response time is:

$$R = S / (1 - U)$$
$$= S / (1 - 0.25)$$
$$= 1.33\ S$$

The 5% increase in load increases response time by about 8%.

Using the same equation, at 90% utilization we have R = 10S and at 95% we have R = 20S, **the 5% increase in load increases response time by a factor of two.** □

The response time of a system becomes unbounded as the system approaches full utilization. Although it might seem that full utilization is an achievable goal, if there is any randomness in arrivals *or* any randomness in service times, full utilization cannot be achieved in steady state without making some tasks wait unbounded amounts of time.

In most systems, well before a system reaches full utilization, average response time will become unbearably long. In the next section, we discuss some of the steps system designers can take in response to overload.

Variance in the response time increases even faster as the system approaches full utilization, proportional to $1 / (1 - U)^2$. Even with 99% utilization, 1% of the time there is no queue at all; random chance means that while sometimes a large number of customers arrive at nearly the same time, at other times the server will be able to work through all of the backlog. If you are lucky enough to arrive at just that moment, you can receive service without waiting. If you are unlucky enough to arrive immediately after a burst of other customers, your wait will be quite long.

Exponential arrivals are burstier than the evenly spaced ones we considered in Figure 7.17 and less bursty than the ones we considered in Figure 7.18. The response time line for the exponential arrivals is higher than the one for evenly spaced arrivals, which was flat across the entire stable range form U = 0 to U = 1, and the line is lower than the one for more bursty arrivals, which rose rapidly even when utilization was low. In general burstier arrivals will produce worse response time for a given level of load.

7.5.4 "What if?" Questions

Queueing theory is particularly useful for answering "what if?" questions: what happens if we change some design parameter of the system. In this section, we consider a selection of these questions, as a way of providing you a bit more intuition.

Scheduling Policy

What happens to the response time curve for other scheduling policies? It depends on the burstiness and predictability of the workload.

If the distribution of arrivals or service times is less bursty than an exponential (e.g., evenly spaced or Gaussian), FIFO will deliver nearly optimal response times, while Round Robin will perform worse than FIFO.

If task service times are exponentially distributed but individual task times are unpredictable, the average response time is the exactly the same for Round Robin as for FIFO. With a memoryless distribution, every queued task has the same expected remaining service time, so switching among tasks has no impact other than to increase overhead.

On the other hand, if task lengths can be predicted and there is variability of service times, Shortest Job First can improve average response time, particularly if arrivals are bursty.

Many real-world systems exhibit more bursty arrivals or service times than an exponential distribution. A bursty distribution is sometimes called *heavy-tailed* because it has more very long tasks; since the mean rate is the same, this also implies that the distribution has even more very short tasks. For example, web page size is heavy-tailed; so is the processing time per web page. Process execution times on desktop computers are also heavy-tailed. For these types of systems, burstiness results in worse average response time than would be predicted by an exponential distribution. That said, for these types of systems, there is an even greater benefit to approximating SJF to avoid stalling small requests behind long ones, and Round Robin will outperform FIFO.

Using SJF (or an approximation) to improve average response time comes at a cost of an increase in response time for long tasks. At low utilization, this increase is small, but at high utilization SJF can result in a massive increase in average response time for long tasks.

To see this, note that any server alternates between periods of being idle (when the queue is empty) and periods of being busy (when the queue is non-empty). If we ignore switching overhead, the scheduling discipline has no impact on these periods — they are only affected by when tasks arrive. Scheduling can only affect which tasks the server handles first.

With SJF, a long task will only complete immediately before an idle period; it is always the last thing in the queue to complete. As utilization increases, these idle periods become increasingly rare. For example, if the server is 99% busy, the server will be idle only 1% of the time. Further, idle periods are *not* evenly distributed — a server is much more likely to be idle if it was idle a second ago. This means that the long jobs are likely to wait for a long time under SJF under high load.

Workloads That Vary With the Queueing Delay

So far, we have assumed that arrival rates and service times are independent of queueing delay. This is not always the case.

For example, suppose a system has 10 users. Each repeatedly issues one request, waits for the result, thinks about the results, and issues the next request. In such a system, the arrival rate will generally be lower during periods when many tasks are queued than during periods when few are. In the limit, during periods when 10 tasks are queued, no new tasks can arrive and the arrival rate is zero.

Or, consider an online store that becomes overloaded and sluggish during a holiday shopping season. Rather than continuing to browse, some customers may get fed up and leave, reducing the number of active browsing sessions and thereby reducing the arrival rate of requests for individual web pages.

Another example is a system with a finite queue. If there is a burst of load that fills the queue, subsequent requests will be turned away until there is space. This heavy-load behavior can be modeled as either a reduced arrival rate or a reduced average service time (some tasks are “processed” by being discarded).

Multiple Servers

Many real systems have not just one but multiple servers. Does it matter whether there is a single queue for everyone or a separate queue per server? Real systems take both approaches: supermarkets tend to have a separate queue per cashier; banks tend to have a single shared queue for bank tellers. Some systems do both: airports often have a single queue at security but have separate queues for the parking garage. Which is better for response time?

Clearly, there are often efficiency gains from having separate queues. Multiprocessor schedulers use separate queues for affinity scheduling and to reduce switching costs; in a supermarket, it may not be practical to have a single queue. On the other hand, users often consider a single (FIFO) queue to be fairer than separate queues. It often seems that we always end up in the slowest line at the supermarket, even if that cannot possibly be true for everyone.

If we focus on average response time, however, a single queue is always better than separate queues, provided that users are not allowed to jump lanes. The reason is simple: because of variations in how long each task takes to service, one server can be idle while another server has multiple queued tasks. Likewise, a single fast server is always better for response time than a large number of slower servers of equal aggregate capacity to the fast server. There is no difference when all servers are busy, but the single fast server will process requests faster when there are fewer active tasks than servers.

Secondary Bottlenecks

If a processor is 90% busy serving web requests, and we add another processor to reduce its load, how much will that improve average response time? Unfortunately, there is not enough information to say. You might like to believe that it will reduce response time by a considerable amount, from $R = S / (1 - 0.9) = 10S$ to $R = S / (1 - 0.45) = 1.8S$.

However, suppose each web request needs not only processing time, but also disk I/O and network bandwidth. If the disk was 80% busy beforehand, it will appear that the processor utilization was the primary problem. Once you add an extra processor, however, the disk becomes the new limiting factor to good performance.

In some cases, queueing theory can make a specific prediction as to the impact of improving one part of a system in isolation. For example, if arrival times are exponentially distributed and independent of the system response time, and if the service times at the processor, disk, and network are also exponentially distributed and independent of one another, then the overall response time for the system is just the sum of the response times of the components:

$$R = \sum_i S_i / (1 - U_i)$$

In this case, improving one part of the system will affect just its contribution to the aggregate system response time. Even though these conditions may not always hold, this is often useful as an approximation to what will occur in real life.

7.5.5 Lessons

To summarize, almost all real-world systems exhibit some randomness in their arrival process or their service times, or both. For these systems:

- Response time increases with increased load.
- System performance is predictable across a range of load factors if we can estimate the average service time per request.
- Burstiness increases average response time. It is mathematically convenient to assume an exponential distribution, but many real-world systems exhibit more burstiness and therefore worse user performance.

7.6 Overload Management

Many systems operate without any direct control over their workload. In the previous section, we explained that good response time and low variance in the response time are both predicated on operating well below peak utilization. If your web service generates interest on Slashdot, however, you can suddenly receive a ton of traffic from new users. Success! Except that the new users discover your service has horrible performance. Disaster!

More sophisticated scheduling can help at low to moderate load, but if the load is more than system can handle, response time will spike, even for short tasks.

The key idea in overload management is to design your system to do less work when overloaded. This will seem strange! After all, you want your system to work a particular way; how can you cripple the user's experience just when your system becomes popular? Under overload conditions, however, your system is incapable of serving all of the requests in the normal way. The only question is: do you choose what to disable, or do you let events choose for you?

An obvious step is to simply reject some requests in order to preserve reasonable response time for the remaining ones. While this can seem harsh, it is also pragmatic. Under overload, the only way to give anyone good service is to reduce or eliminate service for others.

The approach of turning away requests under overload conditions is common in streaming video applications. An overloaded movie service will reject requests to start new streams so that it can continue to provide good streaming service to users that have already started. Likewise, during the NCAA basketball tournament or during the Olympics, the broadcaster will turn requests away, rather than giving everyone poor service.

An apt analogy, perhaps, is that of a popular restaurant. Why not set out acres of tables so that everyone who shows up at the restaurant can be seated? If the waiters Round Robin among the various tables, you can be seated, but wait an hour to get a menu, then wait another hour to make an order, and so forth. That is one way of dealing with a persistent overload situation — by making the user experience so unpleasant that none of your customers will return! As absurd as this scenario is, however, it is close to how we allocate scarce space on congested highways — by making everyone wait.

A less obvious step is to somehow reduce the service time per request under overload conditions. A good example of this happened on September 11, 2001 when CNN's web page was overwhelmed with people trying to get updates about the terrorist attacks. To make the site usable, CNN shifted to a static page that was less personalized and sophisticated but that was faster to serve. As another example, when experiencing unexpected load, EBay will update its auction listings less frequently, saving work that can be used for processing other requests. Finally, an overloaded movie service can reduce the bit rate for everyone in order to serve more simultaneous requests at slightly lower quality.

Amazon has designed its web site to always return a result quickly, even when the requested data is unavailable due to overload conditions. Every backend service has both a normal interface and a fallback to use if its results are not ready in time. For example, this means a user can be told that their purchase will be shipped shortly, even when the book is actually out of stock. This is a strategic decision that it is better to give a wrong answer quickly, and apologize later, rather than to wait to give the right answer more slowly.

Unfortunately, many systems have the opposite problem: they do more work per request as load increases. A simple example of this would be using a linked list to manage a queue of requests: as more requests are queued, more processing time is used maintaining the queue and not getting useful work done. If amount of work per task increases as the load increases, then response times will soar even faster with increased utilization, and throughput can decrease as we add load. This makes overload management even more important.

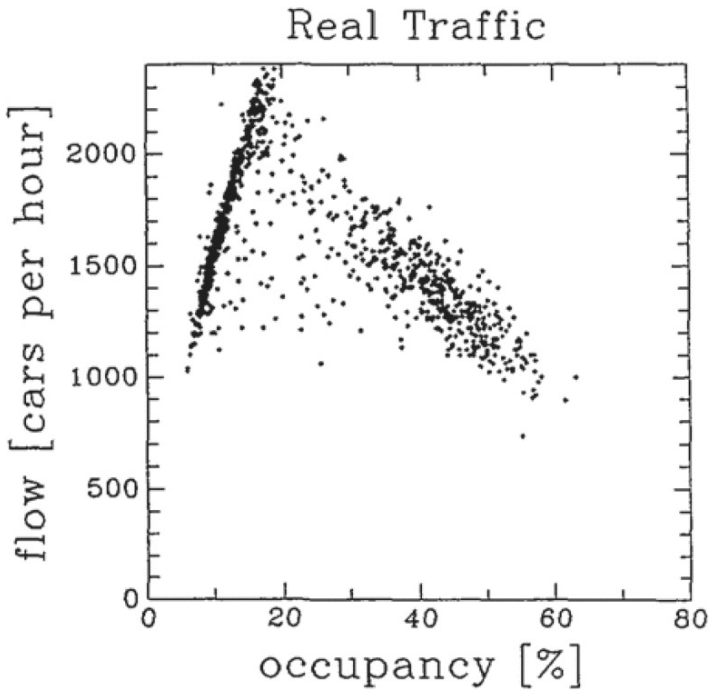

Figure 7.22: Measured throughput (cars per hour) versus occupancy (percentage of the road covered with vehicles). Each data point represents a separate observation. At low load, throughput increases linearly; once load passes a critical point, adding vehicles decreases average throughput. As each vehicle moves more slowly, it takes more time on the highway to complete its journey, increasing load. Data reprinted from Nagel and Schreckenberg [122].

A real-life example of this phenomenon is with highway traffic. Figure 7.22 provides measured data of throughput versus load for one stretch of highway. As you add cars to an empty highway, it increases the rate that cars traverse a given point on the highway. However, at very high loads, the density of cars causes a transition to stop and go traffic, where the rate of progress is much slower than when there were fewer cars. A common solution for highways is to use onramp limiters — to limit the rate that new cars can enter the highway if the system is close to overload.

Time-slicing in the presence of caches has similar behavior. When load is low, there are few time slices, and every task uses its cache efficiently. As more tasks are added to the system, there are more time slices and fewer cache hits, slowing down the processor just when we need it to be running at peak efficiency. In networks, packets are dropped when the network is overloaded. Without careful protocol design, this can cause the sender to retransmit packets, further overloading the network. TCP congestion control, now a common part of almost every Internet connection, was developed precisely to deal with this effect.

You may have even experienced this issue. Some students, as homework piles up, become less, rather than more, efficient. After all, it is hard to concentrate on one project if you know that you really ought to be also working on a different one. But if you decide to take the lessons of this textbook to heart and decide to blow off some of your homework to get the rest of your assignments done, let us suggest that you choose some class other than operating systems!

7.7 Case Study: Servers in a Data Center

We can illustrate the application of the ideas discussed in this chapter, by considering how we should manage a collection of servers in a data center to provide responsive web service. Many web services, such as Google, Facebook, and Amazon, are organized as a set of front-end machines that redirect incoming requests to a larger set of back-end machines. We illustrate this in Figure 7.23. This architecture isolates clients from the architecture of the back-end systems, so that more capacity can be added to the back-end simply by changing the configuration of the front-end systems. Back-end servers can also be taken off-line, have their software upgraded, and so forth, completely transparently to clients.

To provide good response time to the clients of the web service:

- When clients first connect to the service, the front-end node assigns each customer to a back-end server to balance load. Customers can be spread evenly across the back-end servers or they can be assigned to a node with low current load, much as customers at a supermarket select the shortest line for a cashier.
- Additional requests from the same client can be assigned to the same back-end server, as a form of affinity scheduling. Once a server has fetched client data, it will be faster for it to handle additional requests.
- We need to prevent individual users from hogging resources, because that can disrupt performance for other users. A back-end server can favor short tasks over long ones; they can also keep track of the total resources used by each client, reducing the scheduling priority of any client consuming more than their fair share of resources.
- It is often crucial to the usability of a web service to keep response time small. This requires monitoring both the rate of arrivals and the average amount of computing, disk, and network resources consumed by each request. Back-end servers should be added before average server utilization gets too high.
- Since it often takes considerable time to bring new servers online, we need to predict future load and have a backup plan for overload conditions.

7.8 Summary and Future Directions

Resource scheduling is an ancient topic in computer science. Almost from the moment that computers were first multiprogrammed, operating system designers have had to decide which tasks to do first and which to leave for later. This decision — the system's scheduling policy — can have a significant impact on system responsiveness and usability.

Fortunately, the cumulative effect of Moore's Law has shifted the balance towards a focus on improving response time for users, rather than on efficient utilization of resources for the computer. At the same time, the massive scale of the Internet means that many services need to be designed to provide good response time across a wide range of load conditions. Our goal in this chapter is to give you the conceptual basis for making those design choices.

Several ongoing trends pose new and interesting challenges to effective resource scheduling.

- **Multicore systems.** Although almost all new servers, desktops, laptops and smartphones are multicore systems, relatively few widely used applications have been redesigned to take full advantage of multiple processors. This is likely to change over the next few years as multicore systems become ubiquitous and as they scale to larger numbers of processors per chip. Although we have the concepts in place to manage resource sharing among multiple parallel applications, commercial systems are only just now starting to deploy these ideas. It will be interesting to see how the theory works out in practice.
- **Cache affinity.** Over the past twenty years, processor architects have radically increased both the size and number of levels of on-chip caches. There is little reason to believe that this trend will reverse. Although processor clock rates are improving slowly, transistor density is still increasing at a rapid rate. This will make it both possible and desirable to have even larger, multi-level on-chip caches to achieve good performance. Thus, it is likely that scheduling for cache affinity will be an even larger factor in the future than it is today. Balancing when to respect affinity and when to migrate is still somewhat of an open question, as is deciding how to spread or coalesce application threads across caches.
- **Energy-aware scheduling.** The number of energy-constrained computers such as smartphones, tablets, and laptops, now far outstrips powered computers such as desktops and servers. As a result, we are likely to see the development of hardware to monitor and manage energy use by applications, and the operating system will need to make use of that hardware support. We are likely to see operating systems sandbox application energy use to prevent faulty or malicious applications from running down the battery. Likewise, just as applications can adapt to changing numbers of processors, we are likely to see applications that adapt their behavior to energy availability.

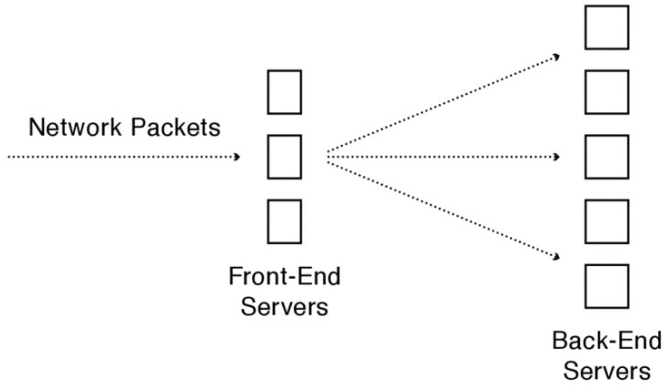

Figure 7.23: A web service often consists of a number of front-end servers who redirect incoming client requests to a larger set of back-end servers.

Exercises

1. For shortest job first, if the scheduler assigns a task to the processor, and no other task becomes schedulable in the meantime, will the scheduler ever preempt the current task? Why or why not?
2. Devise a workload where FIFO is pessimal — it does the worst possible choices — for average response time.
3. Suppose you do your homework assignments in SJF-order. After all, you feel like you are making a lot of progress! What might go wrong?
4. Given the following mix of tasks, task lengths, and arrival times, compute the completion and response time for each task, along with the average response time for the FIFO, RR, and SJF algorithms. Assume a time slice of 10 milliseconds and that all times are in milliseconds.

Task	Length	Arrival Time	Completion Time	Response Time
0	85	0		
1	30	10		
2	35	15		
3	20	80		

4	50	85

Average:

5. Is it possible for an application to run slower when assigned 10 processors than when assigned 8? Why or why not?
6. Suppose your company is considering using one of two candidate scheduling algorithms. One is Round Robin, with an overhead of 1% of the processing power of the system. The second is a wizzy new system that predicts the future and so it can closely approximate SJF, but it takes an overhead of 10% of the processing power of the system.

   Assume randomized arrivals and random task lengths. Under what conditions will the simpler algorithm outperform the more complex, and vice versa?
7. Are there non-trivial workloads for which Multi-level Feedback Queue is an optimal policy? Why or why not? (A trivial workload is one with only one or a few tasks or tasks that last a single instruction.)
8. If a queueing system with one server has a workload of 1000 tasks arriving per second, and the average number of tasks waiting or getting service is 5, what is the average response time per task?
9. Is it possible for a system in equilibrium to have both bounded average response time and 100% utilization? Why or why not?
10. For a queueing system with random arrivals and service times, how does the variance in the service time affect the system response time? Briefly explain.
11. Most round-robin schedulers use a fixed size quantum. Give an argument in favor of and against a small quantum.
12. Which provides the best average response time when there are multiple servers (e.g., bank tellers, supermarket cash registers, airline ticket takers): a single FIFO queue or a FIFO queue per server? Why? Assume that you cannot predict how long any customer is going to take at the server, and that once you have picked a queue to wait in, you are stuck and cannot change queues.
13. Three tasks, A, B, and C are run concurrently on a computer system.
    - Task A arrives first at time 0, and uses the CPU for 100 ms before finishing.
    - Task B arrives shortly after A, still at time 0. Task B loops ten times; for each iteration of the loop, B uses the CPU for 2 ms and then it does I/O for 8 ms.
    - Task C is identical to B, but arrives shortly after B, still at time 0.

    Assuming there is no overhead to doing a context switch, identify when A, B and C will finish for each of the following CPU scheduling disciplines:
    a. FIFO
    b. Round robin with a 1 ms time slice
    c. Round robin with a 100 ms time slice
    d. Multilevel feedback with four levels, and a time slice for the highest priority level is 1 ms.
    e. Shortest job first
14. For each of the following processor scheduling policies, describe the set of workloads under which that policy is optimal in terms of minimizing average response time (does the same thing as shortest job first) and the set of workloads under which the policy is pessimal (does the same thing as longest job first). If there are no workloads under which a policy is optimal or pessimal, indicate that.
    a. FIFO
    b. Round robin
    c. Multilevel feedback queues
15. Explain how you would set up a valid experimental comparison between two scheduling policies, one of which can starve some jobs.
16. As system administrator of a popular social networking website, you notice that usage peaks during working hours (10am – 5pm) and the evening (7 – 10pm) on the US east coast. The CEO asks you to design a system where during these peak hours there will be three levels of users. Users in level 1 are the center of the social network, and so they are to enjoy better response time than users in level 2, who in turn will enjoy better response time than users in level 3. You are to design such a system so that all users will still get some progress, but with the indicated preferences in place.
    a. Will a fixed priority scheme with pre-emption and three fixed priorities work? Why, or why not?
    b. Will a UNIX-style multi-feedback queue work? Why, or why not?
17. Consider the following preemptive priority-scheduling algorithm based on dynamically changing priorities. Larger numbers imply higher priority. Tasks are preempted whenever there is a higher priority task. When a task is waiting for CPU (in the ready queue, but not running), its priority changes at a rate of a:

    $P(t) = P_0 + a \times (t - t_0)$

    where t_0 is the time at which the task joins the ready queue and P_0 is its initial priority, assigned when the task enters the ready queue or is preempted. Similarly, when it is running, the task's priority changes at a rate b. The parameters a, b and P_0 can be used to obtain many different scheduling algorithms.
    a. What is the algorithm that results from $P_0 = 0$ and $b > a > 0$?
    b. What is the algorithm that results from $P_0 = 0$ and $a < b < 0$?
    c. Suppose tasks are assigned a priority 0 when they arrive, but they retain their priority when they are preempted. What happens if two tasks arrive at nearly the same time and $a > 0 > b$?

d. How should we adjust the algorithm to eliminate this pathology?

18. For a computer with two cores and a hyperthreading level of two, draw a graph of the rate of progress of a compute-intensive task as a function of time, depending on whether it is running alone, or with 1, 2, 3, or 4 other tasks.

19. Implement a test on your computer to see if your answer to the previous problem is correct.

20. A countermeasure is a strategy by which a user (or an application) exploits the characteristics of the processor scheduling policy to get as much of the processing time as possible. For example, if the scheduler trusts users to give accurate estimates of how long each task will take, it can give higher priority to short tasks. However, a countermeasure would be for the user to tell the system that the user's tasks are short even when they are not.

    Devise a countermeasure strategy for each of the following processor scheduling policies; your strategy should minimize an individual application's response time (even if it hurts overall system performance). You may assume perfect knowledge — for example, your strategy can be based on which jobs will arrive in the future, where your application is in the queue, and how long the tasks ahead of you will run before blocking. Your strategy should also be robust — it should work properly even if there are no other tasks in the system, there are only short tasks, or there are only long running tasks. If no strategy will improve your application's response time, then explain why.

    a. Last in first out

    b. Round robin, assuming tasks are put at the end of the ready list when they become ready to run

    c. Multilevel feedback queues, where tasks are put on the highest priority queue when they become ready to run

21. Consider a computer system running a general-purpose workload. Measured utilizations (in terms of time, not space) are given in Figure 7.24.

Processor utilization	20.0%
Disk	99.7%
Network	5.0%

Figure 7.24: Measured utilizations of a computer system.

For each of the following changes, say what its likely impact will be on processor utilization, and explain why. Is it likely to significantly increase, marginally increase, significantly decrease, marginally decrease, or have no effect on the processor utilization?

a. Get a faster CPU

b. Get a faster disk

c. Increase the degree of multiprogramming

d. Get a faster network