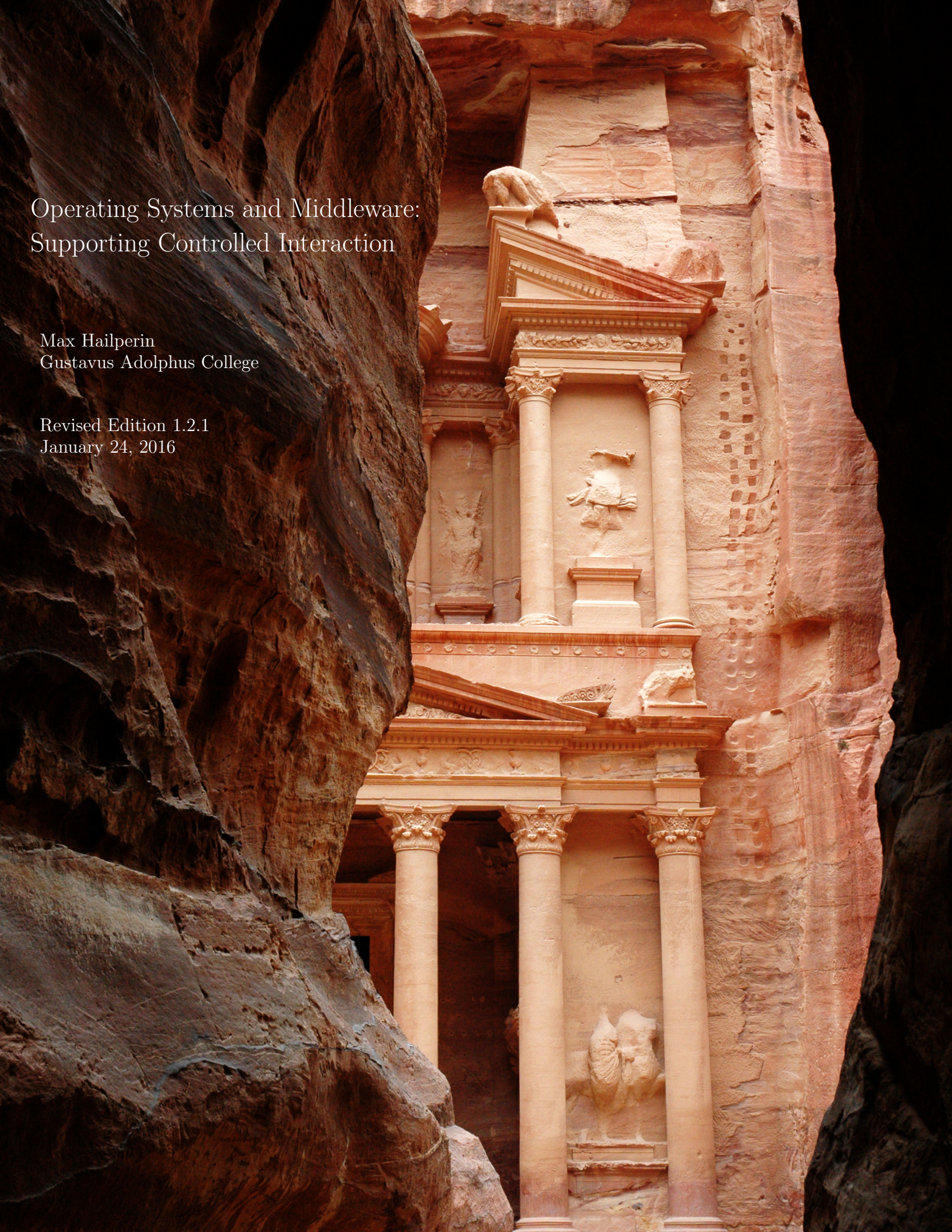
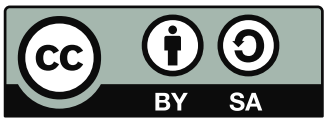


Operating Systems and Middleware: Supporting Controlled Interaction

Max Hailperin
Gustavus Adolphus College

Revised Edition 1.2.1
January 24, 2016



About the Cover

The cover photo shows the treasury coming into view at the end of the *siq*, or defile, leading to the ancient Nabatean city of Petra in present-day Jordan. The siq is a narrow, winding passage cut deep into the sandstone primarily by natural geological forces, though it was improved by the Nabateans.

Petra was a thriving spice trading city. Its prosperity can be linked to several factors, including its location on important trade routes, its access to water through sophisticated hydraulic engineering, and its easily defensible character. The siq played an important role in the latter two aspects. Water conduits were built into the walls of the siq. Meanwhile, the floor of the siq was just wide enough for a single-file merchant caravan of camels, while remaining too narrow to serve as a route for attack.

Operating systems and middleware provide a conducive environment for application programs to interact in a controlled manner, much as Petra must have served for spice merchants 2000 years ago. Access to communication and resources remain as important as then, but so does the provision of tightly controlled interfaces that ensure security.

The photo is by Rhys Davenport, who released it under a Creative Commons Attribution 2.0 Generic license. The photo is available on the web at *http://www.flickr.com/photos/33122834@N06/3437495101/*.

To my family

Contents

Preface

Suppose you sit down at your computer to check your email. One of the messages includes an attached document, which you are to edit. You click the attachment, and it opens up in another window. After you start editing the document, you realize you need to leave for a trip. You save the document in its partially edited state and shut down the computer to save energy while you are gone. Upon returning, you boot the computer back up, open the document, and continue editing.

This scenario illustrates that computations interact. In fact, it demonstrates at least three kinds of interactions between computations. In each case, one computation provides data to another. First, your email program retrieves new mail from the server, using the Internet to bridge space. Second, your email program provides the attachment to the word processor, using the operating system's services to couple the two application programs. Third, the invocation of the word processor that is running before your trip provides the partially edited document to the invocation running after your return, using disk storage to bridge time.

In this book, you will learn about all three kinds of interaction. In all three cases, interesting software techniques are needed in order to bring the computations into contact, yet keep them sufficiently at arm's length that they don't compromise each other's reliability. The exciting challenge, then, is supporting controlled interaction. This includes support for computations that share a single computer and interact with one another, as your email and word processing programs do. It also includes support for data storage and network communication. This book describes how all these kinds of support are provided both by operating systems and by additional software layered on top of operating systems, which is known as middleware.

Audience

If you are an upper-level computer science student who wants to understand how contemporary operating systems and middleware products work and why they work that way, this book is for you. In this book, you will find many forms of balance. The high-level application programmer's view, focused on the services that system software provides, is balanced with a lower-level perspective, focused on the mechanisms used to provide those services. Timeless concepts are balanced with concrete examples of how those concepts are embodied in a range of currently popular systems. Programming is balanced with other intellectual activities, such as the scientific measurement of system performance and the strategic consideration of system security in its human and business context. Even the programming languages used for examples are balanced, with some examples in Java and others in C or C++. (Only limited portions of these languages are used, however, so that the examples can serve as learning opportunities, not stumbling blocks.)

Systems Used as Examples

Most of the examples throughout the book are drawn from the two dominant families of operating systems: Microsoft Windows and the UNIX family, including especially Linux and Mac OS X. Using this range of systems promotes the students' flexibility. It also allows a more comprehensive array of concepts to be concretely illustrated, as the systems embody fundamentally different approaches to some problems, such as the scheduling of processors' time and the tracking of files' disk space.

Most of the examples are drawn from the stable core portions of the operating systems and, as such, are equally applicable to a range of specific versions. Whenever Microsoft Windows is mentioned without further specification, the material should apply to Windows NT, Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Windows 2008, Windows 7, Windows 8, Windows 2012, and Windows 10. All Linux examples are from version 2.6, though much of the material applies to other versions as well. Wherever actual Linux source code is shown (or whenever fine details matter for other reasons), the specific subversion of 2.6 is mentioned in the end-of-chapter notes. Most of the Mac OS X examples originated with version 10.4, also known as Tiger, but should be applicable to other versions.

Where the book discusses the protection of each process's memory, one additional operating system is brought into the mix of examples, in order to illustrate a more comprehensive range of alternative designs. The IBM iSeries, formerly known as the AS/400, embodies an interesting approach to protection that might see wider application within current students' lifetimes. Rather than giving each process its own address space (as Linux, Windows, and Mac OS X do), the iSeries allows all processes to share a single address space and to hold varying access permissions to individual objects within that space.

Several middleware systems are used for examples as well. The Oracle database system is used to illustrate deadlock detection and recovery as well as the use of atomic transactions. Messaging systems appear both as another application of atomic transactions and as an important form of communication middleware, supporting distributed applications. The specific messaging examples are drawn from the IBM WebSphere MQ system (formerly MQSeries) and the Java Message Service (JMS) interface, which is part of Java 2 Enterprise Edition (J2EE). The other communication middleware example is Java RMI (Remote Method Invocation).

Organization of the Text

Chapter 1 provides an overview of the text as a whole, explaining what an operating system is, what middleware is, and what sorts of support these systems provide for controlled interaction.

The next nine chapters work through the varieties of controlled interaction that are exemplified by the scenario at the beginning of the preface: interaction between concurrent computations on the same system (as between your email program and your word processor), interaction across time (as between your word processor before your trip and your word processor after your trip), and interaction across space (as between your email program and your service provider's email server).

The first of these three topics is controlled interaction between computations operating at one time on a particular computer. Before such interaction can make sense, you need to understand how it is that a single computer can be running more than one program, such as an email program in one window and a word processing program in another. Therefore, Chapter 2 explains the fundamental mechanism for dividing a computer's attention between concurrent computations, known as threads. Chapter 3 continues with the related topic of scheduling. That is, if the computer is dividing its

time between computations, it needs to decide which ones to work on at any moment.

With concurrent computations explained, Chapter 4 introduces controlled interactions between them by explaining synchronization, which is control over the threads' relative timing. For example, this chapter explains how, when your email program sends a document to your word processor, the word processor can be constrained to read the document only after the email program writes it. One particularly important form of synchronization, atomic transactions, is the topic of Chapter 5. Atomic transactions are groups of operations that take place as an indivisible unit; they are most commonly supported by middleware, though they are also playing an increasing role in operating systems.

Other than synchronization, the main way that operating systems control the interaction between computations is by controlling their access to memory. Chapter 6 explains how this is achieved using the technique known as virtual memory. That chapter also explains the many other objectives this same technique can serve. Virtual memory serves as the foundation for Chapter 7's topic, which is processes. A process is the fundamental unit of computation for protected access, just as a thread is the fundamental unit of computation for concurrency. A process is a group of threads that share a protection environment; in particular, they share the same access to virtual memory.

The next three chapters move outside the limitations of a single computer operating in a single session. First, consider the document stored before a trip and available again after it. Chapter 8 explains persistent storage mechanisms, focusing particularly on the file storage that operating systems provide. Second, consider the interaction between your email program and your service provider's email server. Chapter 9 provides an overview of networking, including the services that operating systems make available to programs such as the email client and server. Chapter 10 extends this discussion into the more sophisticated forms of support provided by communication middleware, such as messaging systems, RMI, and web services.

Finally, Chapter 11 focuses on security. Because security is a pervasive issue, the preceding ten chapters all provide some information on it as well. Specifically, the final section of each chapter points out ways in which security relates to that chapter's particular topic. However, even with that coverage distributed throughout the book, a chapter specifically on security is needed, primarily to elevate it out of technical particulars and talk about general principles and the human and organizational context surrounding

the computer technology.

The best way to use these chapters is in consecutive order. However, Chapter 5 can be omitted with only minor harm to Chapters 8 and 10, and Chapter 9 can be omitted if students are already sufficiently familiar with networking.

Relationship to Computer Science Curriculum 2008

Operating systems are traditionally the subject of a course required for all computer science majors. In recent years, however, there has been increasing interest in the idea that upper-level courses should be centered less around particular artifacts, such as operating systems, and more around cross-cutting concepts. In particular, the *Computing Curricula 2001* (CC2001) and its interim revision, *Computer Science Curriculum 2008* (CS2008), provide encouragement for this approach, at least as one option. Most colleges and universities still retain a relatively traditional operating systems course, however. Therefore, this book steers a middle course, moving in the direction of the cross-cutting concerns while retaining enough familiarity to be broadly adoptable.

The following table indicates the placement within this text of knowledge units from CS2008's computer science body of knowledge. Those knowledge units designated as core units within CS2008 are listed in italics. The book covers all core operating systems (OS) units, as well as one elective OS unit. The overall amount of coverage for each unit is always at least that recommended by CS2008, though sometimes the specific subtopics don't quite correspond exactly. Outside the OS area, this book's most substantial coverage is of Net-Centric Computing (NC); another major topic, transaction processing, comes from Information Management (IM). In each row, the listed chapters contain the bulk of the knowledge unit's coverage, though

some topics may be elsewhere.

Knowledge unit (italic indicates core units in CS2008)	**Chapter(s)**
OS/OverviewOfOperatingSystems	1
OS/OperatingSystemPrinciples	1, 7
OS/Concurrency	2, 4
OS/SchedulingAndDispatch	3
OS/MemoryManagement	6
OS/SecurityAndProtection	7, 11
OS/FileSystems	8
NC/Introduction	9
NC/NetworkCommunication (partial coverage)	9
NC/NetworkSecurity (partial coverage)	9
NC/WebOrganization (partial coverage)	9
NC/NetworkedApplications (partial coverage)	10
IM/TransactionProcessing	5

Your Feedback Is Welcome

Comments, suggestions, and bug reports are welcome; please send email to `max@gustavus.edu` or use the github issue tracker. Bug reports can earn you a bounty of $2.56 apiece as a token of gratitude. (The great computer scientist Donald Knuth started this tradition. Given how close to bug-free his publications have become, it seems to work.) For purposes of this reward, the definition of a bug is simple: if as a result of your comment the author chooses to make a change, then you have pointed out a bug. The change need not be the one you suggested, and the bug need not be technical in nature. Unclear writing qualifies, for example.

Features of the Text

Each chapter concludes with five standard elements. The last numbered section within the chapter is always devoted to security matters related to the chapter's topic. Next comes three different lists of opportunities for active participation by the student: exercises, programming projects, and exploration projects. Finally, the chapter ends with historical and bibliographic notes.

The distinction between exercises, programming projects, and exploration projects needs explanation. An exercise can be completed with no

outside resources beyond paper and pencil: you need just this textbook and your mind. That does not mean all the exercises are cut and dried, however. Some may call upon you to think creatively; for these, no one answer is correct. Programming projects require a nontrivial amount of programming; that is, they require more than making a small, easily identified change in an existing program. However, a programming project may involve other activities beyond programming. Several of them involve scientific measurement of performance effects, for example; these exploratory aspects may even dominate over the programming aspects. An exploration project, on the other hand, can be an experiment that can be performed with no real programming; at most you might change a designated line within an existing program. The category of exploration projects does not just include experimental work, however. It also includes projects that require you to do research on the Internet or using other library resources.

Supplemental Resources

The author of this text is making supplemental resources available on his own website. Additionally, the publisher of the earlier first edition commissioned additional resources from independent supplement authors, which may still be available through the publisher's website and would largely still apply to this revised edition. The author's website, *https://gustavus.edu/+max/os-book/*, contains at least the following materials:

- Full text of this revised edition
- Source code in Java, C, or C++ for all programs that are shown in the text
- Artwork files for all figures in the text
- A link to the book's github site, which includes an issue tracker (errata list)

About the Revised Edition

Course Technology published the first edition of this book in January of 2006 and in October of 2010 assigned the copyright back to the author, giving him the opportunity to make it freely available. This revised edition closely follows the first edition; rather than being a thorough update, it is aimed at three narrow goals:

- All errata reported in the first edition are corrected.
- A variety of other minor improvements appear throughout, such as clarified explanations and additional exercises, projects, and end-of-chapter notes.
- Two focused areas received more substantial updates:
  - The explanation of Linux's scheduler was completely replaced to correspond to the newer "Completely Fair Scheduler" (CFS), including its group scheduling feature.
  - A new section, 4.9, was added on nonblocking synchronization.

In focusing on these limited goals, a key objective was to maintain as much compatibility with the first edition as possible. Although page numbering changed, most other numbers stayed the same. All new exercises and projects were added to the end of the corresponding lists for that reason. The only newly added section, 4.9, is near the end of its chapter; thus, the only changed section number is that the old Section 4.9 ("Security and Synchronization") became 4.10. Only in Chapter 4 did any figure numbers change.

It is my hope that others will join me in making further updates and improvements to the text. I am releasing it under a Creative Commons license that allows not just free copying, but also the freedom to make modifications, so long as the modified version is released under the same terms. In order to make such modifications practical, I'm not just releasing the book in PDF form, but also as a collection of LaTeX source files that can be edited and then run through the `pdflatex` program (along with `bibtex` and `makeindex`). The source file collection also includes PDF files of all artwork figures; Course Technology has released the rights to the artwork they contracted to have redrawn. All of this is on the github site.

If you produce a modified version of this text, the Creative Commons license allows you considerable flexibility in how you make your modified version available. I would urge you to contribute it back using a "pull request" on the main github site—we will all benefit from having a central repository of progress. Separate materials to supplement the text would also be welcome. One category that occurs to me is animations or screencasts; the static figures in the text are rather limited. Another worthwhile project would be to transform the text into a more contribution-friendly form, such as a wiki.

Acknowledgments

This book was made possible by financial and logistical support from my employer, Gustavus Adolphus College, and moral support from my family. I would like to acknowledge the contributions of the publishing team, especially developmental editor Jill Batistick and Product Manager Alyssa Pratt. I am also grateful to my students for doing their own fair share of teaching. I particularly appreciate the often extensive comments I received from the following individuals, each of whom reviewed one or more chapters: Dan Cosley, University of Minnesota, Twin Cities; Allen Downey, Franklin W. Olin College of Engineering; Michael Goldweber, Xavier University; Ramesh Karne, Towson University; G. Manimaran, Iowa State University; Alexander Manov, Illinois Institute of Technology; Peter Reiher, University of California, Los Angeles; Rich Salz, DataPower Technology; Dave Schulz, Wisconsin Lutheran College; Sanjeev Setia, George Mason University; and Jon Weissman, University of Minnesota, Twin Cities. Although I did not adopt all their suggestions, I did not ignore any of them, and I appreciate them all.

In preparing the revised edition, I took advantage of suggestions from many readers. I would like to thank all of them, even those I've managed to lose track of, to whom I also apologize. Those I can thank by name are Joel Adams, Michael Brackney, Jack Briner, Justin Delegard, Ben Follis, MinChan Kim, Finn Kuusisto, Matt Lindner, Milo Martin, Gabe Schmidt, Fritz Sieker, and Alex Wauck.

Since the initial release of the revised edition, user suggestions have continued to drive most of the progress. The github issue tracker and pull requests show the history of these valuable contributions.

Chapter 1

Introduction

1.1 Chapter Overview

This book covers a lot of ground. In it, I will explain to you the basic principles that underlie a broad range of systems and also give you concrete examples of how those principles play out in several specific systems. You will see not only some of the internal workings of low-level infrastructure, but also how to build higher-level applications on top of that infrastructure to make use of its services. Moreover, this book will draw on material you may have encountered in other branches of computer science and engineering and engage you in activities ranging from mathematical proofs to the experimental measurement of real-world performance and the consideration of how systems are used and abused in social context.

Because the book as a whole covers so much ground, this chapter is designed to give you a quick view of the whole terrain, so that you know what you are getting into. This is especially important because several of the topics I cover are interrelated, so that even though I carefully designed the order of presentation, I am still going to confront you with occasional forward references. You will find, however, that this introductory chapter gives you a sufficient overview of all the topics so that you won't be mystified when a chapter on one makes some reference to another.

In Section 1.2, I will explain what an operating system is, and in Section 1.3, I will do the same for middleware. After these two sections, you will know what general topic you are studying. Section 1.4 gives you some reasons for studying that topic, by explaining several roles that I hope this book will serve for you.

After the very broad overview provided by these initial sections, the

remaining sections of this chapter are somewhat more focused. Each corresponds to one or more of the later chapters and explains one important category of service provided by operating systems and middleware. Section 1.5 explains how a single computer can run several computations concurrently, a topic addressed in more depth by Chapters 2 and 3. Section 1.6 explains how interactions between those concurrent computations can be kept under control, the topic of Chapters 4 through 7. Sections 1.7 and 1.8 extend the range of interacting computations across time and space, respectively, through mechanisms such as file systems and networking. They preview Chapter 8 and Chapters 9 and 10. Finally, Section 1.9 introduces the topic of security, a topic I revisit at the end of each chapter and then focus on in Chapter 11.

1.2 What Is an Operating System?

An *operating system* is software that uses the hardware resources of a computer system to provide support for the execution of other software. Specifically, an operating system provides the following services:

- The operating system allows multiple computations to take place concurrently on a single computer system. It divides the hardware's time between the computations and handles the shifts of focus between the computations, keeping track of where each one leaves off so that it can later correctly resume.

- The operating system controls the interactions between the concurrent computations. It can enforce rules, such as forbidding computations from modifying data structures while other computations are accessing those structures. It can also provide isolated areas of memory for private use by the different computations.

- The operating system can provide support for controlled interaction of computations even when they do not run concurrently. In particular, general-purpose operating systems provide file systems, which allow computations to read data from files written by earlier computations. This feature is optional because an embedded system, such as the computer controlling a washing machine, might in some cases run an operating system, but not provide a file system or other long-term storage.

- The operating system can provide support for controlled interaction of computations spread among different computer systems by using networking. This is another standard feature of general-purpose operating systems.

These services are illustrated in Figure 1.1.

If you have programmed only general-purpose computers, such as PCs, workstations, and servers, you have probably never encountered a computer system that was not running an operating system or that did not allow multiple computations to be ongoing. For example, when you boot up your own computer, chances are it runs Linux, Microsoft Windows, or Mac OS X and that you can run multiple application programs in individual windows on the display screen. These three operating systems will serve as my primary examples throughout the book.

To illustrate that a computer can run a single program without an operating system, consider embedded systems. A typical embedded system might have neither keyboard nor display screen. Instead, it might have temperature and pressure sensors and an output that controls the fuel injectors of your car. Alternatively, it might have a primitive keyboard and display, as on a microwave oven, but still be dedicated to running a single program.

Some of the most sophisticated embedded systems run multiple cooperating programs and use operating systems. However, more mundane embedded systems take a simpler form. A single program is directly executed by the embedded processor. That program contains instructions to read from input sensors, carry out appropriate computations, and write to the output devices. This sort of embedded system illustrates what is possible without an operating system. It will also serve as a point of reference as I contrast my definition of an operating system with an alternative definition.

One popular alternative definition of an operating system is that it provides application programmers with an abstract view of the underlying hardware resources, taking care of the low-level details so that the applications can be programmed more simply. For example, the programmer can write a simple statement to output a string without concern for the details of making each character appear on the display screen.

I would counter by remarking that abstraction can be provided without an operating system, by linking application programs with separately written libraries of supporting procedures. For example, a program could output a string using the standard mechanism of a programming language, such as C++ or Java. The application programmer would not need to know

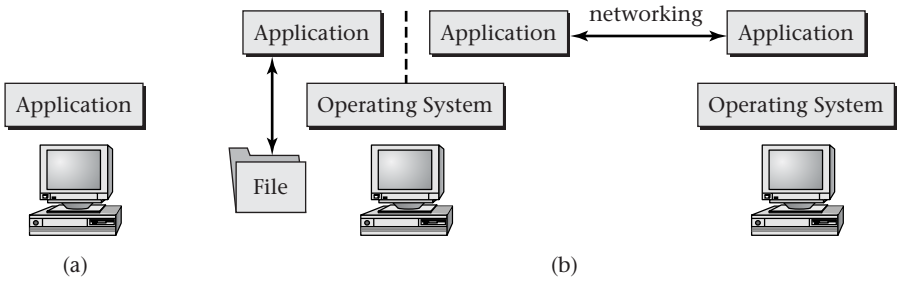


Figure 1.1: Without an operating system, a computer can directly execute a single program, as shown in part (a). Part (b) shows that with an operating system, the computer can support concurrent computations, control the interactions between them (suggested by the dashed line), and allow communication across time and space by way of files and networking.

anything about hardware. However, rather than running on an operating system, the program could be linked together with a library that performed the output by appropriately manipulating a microwave oven's display panel. Once running on the oven's embedded processor, the library and the application code would be a single program, nothing more than a sequence of instructions to directly execute. However, from the application programmer's standpoint, the low-level details would have been successfully hidden.

To summarize this argument, a library of input/output routines is not the same as an operating system, because it satisfies only the first part of my definition. It does use underlying hardware to support the execution of other software. However, it does not provide support for controlled interaction between computations. In fairness to the alternative viewpoint, it is the more historically grounded one. Originally, a piece of software could be called an operating system without supporting controlled interaction. However, the language has evolved such that my definition more closely reflects current usage.

I should also address one other alternative view of operating systems, because it is likely to be the view you have formed from your own experience using general-purpose computers. You are likely to think of an operating system as the software with which you interact in order to carry out tasks such as running application programs. Depending on the user interface to which you are accustomed, you might think the operating system is what allows you to click program icons to run them, or you might think the operating system is what interprets commands you type.

There is an element of truth to this perception. The operating system does provide the service of executing a selected application program. However, the operating system provides this service not to human users clicking icons or typing commands, but to other programs already running on the computer, including the one that handles icon clicks or command entries. The operating system allows one program that is running to start another program running. This is just one of the many services the operating system provides to running programs. Another example service is writing output into a file. The sum total of features the operating system makes available for application programmers to use in their programs is called the *Application Programming Interface* (*API*). One element of the API is the ability to run other programs.

The reason why you can click a program icon or type in a command to run a program is that general-purpose operating systems come bundled with a user-interface program, which uses the operating system API to run other programs in response to mouse or keyboard input. At a marketing level, this user-interface program may be treated as a part of the operating system; it may not be given a prominent name of its own and may not be available for separate purchase.

For example, Microsoft Windows comes with a user interface known as File Explorer, which provides features such as the Start menu and the ability to click icons. (This program was named Windows Explorer prior to Windows 8.) However, even if you are an experienced Windows user, you may never have heard of File Explorer; Microsoft has chosen to give it a very low profile, treating it as an integral part of the Microsoft Windows environment. At a technical level, however, it is distinct from the operating system proper. In order to make the distinction explicit, the true operating system is often called the *kernel*. The kernel is the fundamental portion of Microsoft Windows that provides an API supporting computations with controlled interactions.

A similar distinction between the kernel and the user interface applies to Linux. The Linux kernel provides the basic operating system services through an API, whereas *shells* are the programs (such as bash and tcsh) that interpret typed commands, and *desktop environments* are the programs, such as KDE (K Desktop Environment) and GNOME, that handle graphical interaction.

In this book, I will explain the workings of operating system kernels, the true operating systems themselves, as opposed to the user-interface programs. One reason is because user-interface programs are not constructed in any fundamentally different way than normal application programs. The

other reason is because an operating system need not have this sort of user interface at all. Consider again the case of an embedded system that controls automotive fuel injection. If the system is sufficiently sophisticated, it may include an operating system. The main control program may run other, more specialized programs. However, there is no ability for the user to start an arbitrary program running through a shell or desktop environment. In this book, I will draw my examples from general-purpose systems with which you might be familiar, but will emphasize the principles that could apply in other contexts as well.

1.3 What Is Middleware?

Now that you know what an operating system is, I can turn to the other category of software covered by this book: *middleware*. Middleware is software occupying a middle position between application programs and operating systems, as I will explain in this section.

Operating systems and middleware have much in common. Both are software used to support other software, such as the application programs you run. Both provide a similar range of services centered around controlled interaction. Like an operating system, middleware may enforce rules designed to keep the computations from interfering with one another. An example is the rule that only one computation may modify a shared data structure at a time. Like an operating system, middleware may bring computations at different times into contact through persistent storage and may support interaction between computations on different computers by providing network communication services.

Operating systems and middleware are not the same, however. They rely upon different underlying providers of lower-level services. An operating system provides the services in its API by making use of the features supported by the hardware. For example, it might provide API services of reading and writing named, variable-length files by making use of a disk drive's ability to read and write numbered, fixed-length blocks of data. Middleware, on the other hand, provides the services in its API by making use of the features supported by an underlying operating system. For example, the middleware might provide API services for updating relational database tables by making use of an operating system's ability to read and write files that contain the database.

This layering of middleware on top of an operating system, as illustrated in Figure 1.2, explains the name; middleware is in the middle of the vertical

stack, between the application programs and the operating system. Viewed horizontally rather than vertically, middleware is also in the middle of interactions between different application programs (possibly even running on different computer systems), because it provides mechanisms to support controlled interaction through coordination, persistent storage, naming, and communication.

I already mentioned relational database systems as one example of middleware. Such systems provide a more sophisticated form of persistent storage than the files supported by most operating systems. I use Oracle as my primary source of examples regarding relational database systems. Other middleware I will use for examples in the book includes the Java 2 Platform, Enterprise Edition (J2EE) and IBM's WebSphere MQ. These systems provide support for keeping computations largely isolated from undesirable interactions, while allowing them to communicate with one another even if running on different computers.

The marketing definition of middleware doesn't always correspond exactly with my technical definition. In particular, some middleware is of such fundamental importance that it is distributed as part of the operating system bundle, rather than as a separate middleware product. As an example, general-purpose operating systems all come equipped with some mechanism for translating Internet hostnames, such as *www.gustavus.edu*, into numerical addresses. These mechanisms are typically outside the operating system kernel, but provide a general supporting service to application programs. Therefore, by my definition, they are middleware, even if not normally labeled as such.

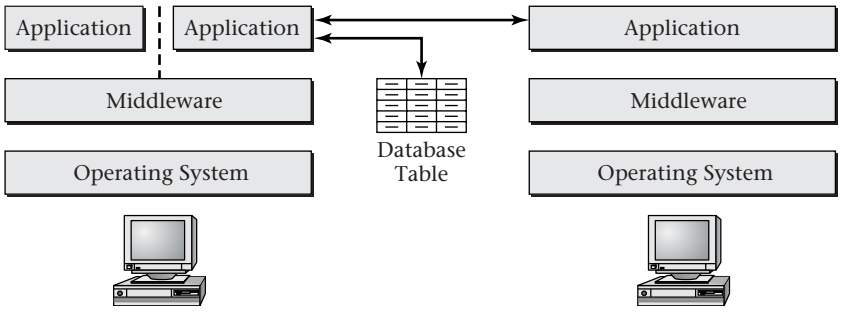


Figure 1.2: Middleware uses services from an operating system and in turn provides services to application programs to support controlled interaction.

1.4 Objectives for the Book

If you work your way through this book, you will gain both knowledge and skills. Notice that I did not say anything about *reading* the book, but rather about *working your way through* the book. Each chapter in this book concludes with exercises, programming projects, exploration projects, and some bibliographic or historical notes. To achieve the objectives of the book, you need to work exercises, carry out projects, and occasionally venture down one of the side trails pointed out by the end-of-chapter notes. Some of the exploration projects will specifically direct you to do research in outside sources, such as on the Internet or in a library. Others will call upon you to do experimental work, such as measuring the performance consequences of a particular design choice. If you are going to invest that kind of time and effort, you deserve some idea of what you stand to gain from it. Therefore, I will explain in the following paragraphs how you will be more knowledgeable and skilled after finishing the book.

First, you will gain a general knowledge of how contemporary operating systems and middleware work and some idea why they work that way. That knowledge may be interesting in its own right, but it also has practical applications. Recall that these systems provide supporting APIs for application programmers to use. Therefore, one payoff will be that if you program applications, you will be positioned to make more effective use of the supporting APIs. This is true even though you won't be an expert at any particular API; instead, you'll see the big picture of what services those APIs provide.

Another payoff will be if you are in a role where you need to alter the configuration of an operating system or middleware product in order to tune its performance or make it best serve a particular context. Again, this one book alone won't give you all the specific knowledge you need about any particular system, but it will give you the general background to make sense out of more specialized references.

Perhaps the most significant payoff for learning the details of today's systems in the context of the reasons behind their designs is that you will be in a better position to learn tomorrow's systems. You will be able to see in what ways they are different and in what ways they are fundamentally still the same. You will be able to put new features into context, often as a new solution to an old problem, or even just as a variant on an existing solution. If you really get excited by what you learn from this book, you could even use your knowledge as the foundation for more advanced study and become one of the people who develops tomorrow's systems.

Second, in addition to knowledge about systems, you will learn some skills that are applicable even outside the context of operating systems and middleware. Some of the most important skills come from the exploration projects. For example, if you take those projects seriously, you'll practice not only conducting experiments, but also writing reports describing the experiments and their results. That will serve you well in many contexts.

I have also provided you with some opportunities to develop proficiency in using the professional literature, such as documentation and the papers published in conference proceedings. Those sources go into more depth than this book can, and they will always be more up-to-date.

From the programming projects, you'll gain some skill at writing programs that have several interacting components operating concurrently with one another and that keep their interactions under control. You'll also develop some skill at writing programs that interact over the Internet. In neither case will you become a master programmer. However, in both cases, you will be laying a foundation of skills that are relevant to a range of development projects and environments.

Another example of a skill you can acquire is the ability to look at the security ramifications of design decisions. I have a security section in each chapter, rather than a security chapter only at the end of the book, because I want you to develop the habit of asking, "What are the security issues here?" That question is relevant even outside the realm of operating systems and middleware.

As I hope you can see, studying operating systems and middleware can provide a wide range of benefits, particularly if you engage yourself in it as an active participant, rather than as a spectator. With that for motivation, I will now take you on another tour of the services operating systems and middleware provide. This tour is more detailed than Sections 1.2 and 1.3, but not as detailed as Chapters 2 through 11.

1.5 Multiple Computations on One Computer

The single most fundamental service an operating system provides is to allow multiple computations to be going on at the same time, rather than forcing each to wait until the previous one has run to completion. This allows desktop computers to juggle multiple tasks for the busy humans seated in front of their screens, and it allows server computers to be responsive to requests originating from many different client computers on the Internet. Beyond these responsiveness concerns, concurrent computations can also

make more efficient use of a computer's resources. For example, while one computation is stalled waiting for input to arrive, another computation can be making productive use of the processor.

A variety of words can be used to refer to the computations underway on a computer; they may be called threads, processes, tasks, or jobs. In this book, I will use both the word "thread" and the word "process," and it is important that I explain now the difference between them.

A *thread* is the fundamental unit of concurrency. Any one sequence of programmed actions is a thread. Executing a program might create multiple threads, if the program calls for several independent sequences of actions run concurrently with one another. Even if each execution of a program creates only a single thread, which is the more normal case, a typical system will be running several threads: one for each ongoing program execution, as well as some that are internal parts of the operating system itself.

When you start a program running, you are always creating one or more threads. However, you are also creating a *process*. The process is a container that holds the thread or threads that you started running and protects them from unwanted interactions with other unrelated threads running on the same computer. For example, a thread running in one process cannot accidentally overwrite memory in use by a different process.

Because human users normally start a new process running every time they want to make a new computation happen, it is tempting to think of processes as the unit of concurrent execution. This temptation is amplified by the fact that older operating systems required each process to have exactly one thread, so that the two kinds of object were in one-to-one correspondence, and it was not important to distinguish them. However, in this book, I will consistently make the distinction. When I am referring to the ability to set an independent sequence of programmed actions in motion, I will write about creating threads. Only when I am referring to the ability to protect threads will I write about creating processes.

In order to support threads, operating system APIs include features such as the ability to create a new thread and to kill off an existing thread. Inside the operating system, there must be some mechanism for switching the computer's attention between the various threads. When the operating system suspends execution of one thread in order to give another thread a chance to make progress, the operating system must store enough information about the first thread to be able to successfully resume its execution later. Chapter 2 addresses these issues.

Some threads may not be runnable at any particular time, because they are waiting for some event, such as the arrival of input. However, in general,

an operating system will be confronted with multiple runnable threads and will have to choose which ones to run at each moment. This problem of scheduling threads' execution has many solutions, which are surveyed in Chapter 3. The scheduling problem is interesting, and has generated so many solutions, because it involves the balancing of system users' competing interests and values. No individual scheduling approach will make everyone happy all the time. My focus is on explaining how the different scheduling approaches fit different contexts of system usage and achieve differing goals. In addition I explain how APIs allow programmers to exert control over scheduling, for example, by indicating that some threads should have higher priority than others.

1.6 Controlling the Interactions Between Computations

Running multiple threads at once becomes more interesting if the threads need to interact, rather than execute completely independently of one another. For example, one thread might be producing data that another thread consumes. If one thread is writing data into memory and another is reading the data out, you don't want the reader to get ahead of the writer and start reading from locations that have yet to be written. This illustrates one broad family of control for interaction: control over the relative timing of the threads' execution. Here, a reading step must take place after the corresponding writing step. The general name for control over threads' timing is *synchronization*.

Chapter 4 explains several common synchronization patterns, including keeping a consumer from outstripping the corresponding producer. It also explains the mechanisms that are commonly used to provide synchronization, some of which are supported directly by operating systems, while others require some modest amount of middleware, such as the Java runtime environment.

That same chapter also explains a particularly important difficulty that can arise from the use of synchronization. Synchronization can force one thread to wait for another. What if the second thread happens to be waiting for the first? This sort of cyclic waiting is known as a *deadlock*. My discussion of ways to cope with deadlock also introduces some significant middleware, because database systems provide an interesting example of deadlock handling.

In Chapter 5, I expand on the themes of synchronization and middleware

by explaining transactions, which are commonly supported by middleware. A *transaction* is a unit of computational work for which no intermediate state from the middle of the computation is ever visible. Concurrent transactions are isolated from seeing each other's intermediate storage. Additionally, if a transaction should fail, the storage will be left as it was before the transaction started. Even if the computer system should catastrophically crash in the middle of a transaction's execution, the storage after rebooting will not reflect the partial transaction. This prevents results of a half-completed transaction from becoming visible. Transactions are incredibly useful in designing reliable information systems and have widespread commercial deployment. They also provide a good example of how mathematical reasoning can be used to help design practical systems; this will be the chapter where I most prominently expect you to understand a proof.

Even threads that have no reason to interact may accidentally interact, if they are running on the same computer and sharing the same memory. For example, one thread might accidentally write into memory being used by the other. This is one of several reasons why operating systems provide *virtual memory*, the topic of Chapter 6. Virtual memory refers to the technique of modifying addresses on their way from the processor to the memory, so that the addresses actually used for storing values in memory may be different from those appearing in the processor's load and store instructions. This is a general mechanism provided through a combination of hardware and operating system software. I explain several different goals this mechanism can serve, but the most simple is isolating threads in one process from those in another by directing their memory accesses to different regions of memory.

Having broached the topic of providing processes with isolated virtual memory, I devote Chapter 7 to processes. This chapter explains an API for creating processes. However, I also focus on protection mechanisms, not only by building on Chapter 6's introduction of virtual memory, but also by explaining other forms of protection that are used to protect processes from one another and to protect the operating system itself from the processes. Some of these protection mechanisms can be used to protect not just the storage of values in memory, but also longer-term data storage, such as files, and even network communication channels. Therefore, Chapter 7 lays some groundwork for the later treatment of these topics.

Chapter 7 also provides me an opportunity to clarify one point about threads left open by Chapter 2. By showing how operating systems provide a protective boundary between themselves and the running application processes, I can explain where threads fall relative to this boundary. In particular, there are threads that are contained entirely within the operating

system kernel, others that are contained entirely within an application process, and yet others that cross the boundary, providing support from within the kernel for concurrent activities within the application process. Although it might seem natural to discuss these categories of threads in Chapter 2, the chapter on threads, I really need to wait for Chapter 7 in order to make any more sense out of the distinctions than I've managed in this introductory paragraph.

When two computations run concurrently on a single computer, the hard part of supporting controlled interaction is to keep the interaction under control. For example, in my earlier example of a pair of threads, one produces some data and the other consumes it. In such a situation, there is no great mystery to how the data can flow from one to the other, because both are using the same computer's memory. The hard part is regulating the use of that shared memory. This stands in contrast to the interactions across time and space, which I will address in Sections 1.7 and 1.8. If the producer and consumer run at different times, or on different computers, the operating system and middleware will need to take pains to convey the data from one to the other.

1.7 Supporting Interaction Across Time

General purpose operating systems all support some mechanism for computations to leave results in long-term storage, from which they can be retrieved by later computations. Because this storage persists even when the system is shut down and started back up, it is known as *persistent storage*. Normally, operating systems provide persistent storage in the form of named files, which are organized into a hierarchy of directories or folders. Other forms of persistent storage, such as relational database tables and application-defined persistent objects, are generally supported by middleware. In Chapter 8, I focus on file systems, though I also explain some of the connections with middleware. For example, I compare the storage of file directories with that of database indexes. This comparison is particularly important as these areas are converging. Already the underlying mechanisms are very similar, and file systems are starting to support indexing services like those provided by database systems.

There are two general categories of file APIs, both of which I cover in Chapter 8. The files can be made a part of the process's virtual memory space, accessible with normal load and store instructions, or they can be treated separately, as external entities to read and write with explicit

operations.

Either kind of file API provides a relatively simple interface to some quite significant mechanisms hidden within the operating system. Chapter 8 also provides a survey of some of these mechanisms.

As an example of a simple interface to a sophisticated mechanism, an application programmer can make a file larger simply by writing additional data to the end of the file. The operating system, on the other hand, has to choose the location where the new data will be stored. When disks are used, this space allocation has a strong influence on performance, because of the physical realities of how disk drives operate.

Another job for the file system is to keep track of where the data for each file is located. It also keeps track of other file-specific information, such as access permissions. Thus, the file system not only stores the files' data, but also stores *metadata*, which is data describing the data.

All these mechanisms are similar to those used by middleware for purposes such as allocating space to hold database tables. Operating systems and middleware also store information, such as file directories and database indexes, used to locate data. The data structures used for these naming and indexing purposes are designed for efficient access, just like those used to track the allocation of space to stored objects.

To make the job of operating systems and middleware even more challenging, persistent storage structures are expected to survive system crashes without significant loss of integrity. For example, it is not acceptable after a crash for specific storage space to be listed as available for allocation and also to be listed as allocated to a file. Such a confused state must not occur even if the crash happened just as the file was being created or deleted. Thus, Chapter 8 builds on Chapter 5's explanation of atomic transactions, while also outlining some other mechanisms that can be used to protect the integrity of metadata, directories, and indexes.

Persistent storage is crucially important, perhaps even more so in the Internet age than in prior times, because servers now hold huge amounts of data for use by clients all over the world. Nonetheless, persistent storage no longer plays as unique a role as it once did. Once upon a time, there were many computer systems in which the only way processes communicated was through persistent storage. Today, that is almost unthinkable, because communication often spans the Internet. Therefore, as I explain in Section 1.8, operating systems provide support for networking, and middleware provides further support for the construction of distributed systems.

1.8 Supporting Interaction Across Space

In order to build coherent software systems with components operating on differing computers, programmers need to solve lots of problems. Consider two examples: data flowing in a stream must be delivered in order, even if sent by varying routes through interconnected networks, and message delivery must be incorporated into the all-or-nothing guarantees provided by transactions. Luckily, application programmers don't need to solve most of these problems, because appropriate supporting services are provided by operating systems and middleware.

I divide my coverage of these services into two chapters. Chapter 9 provides a foundation regarding networking, so that this book will stand on its own if you have not previously studied networking. That chapter also covers services commonly provided by operating systems, or in close conjunction with operating systems, such as distributed file systems. Chapter 10, in contrast, explains the higher-level services that middleware provides for application-to-application communication, in such forms as messaging and web services. Each chapter introduces example APIs that you can use as an application programmer, as well as the more general principles behind those specific APIs.

Networking systems, as I explain in Chapter 9, are generally partitioned into layers, where each layer makes use of the services provided by the layer under it in order to provide additional services to the layer above it. At the bottom of the stack is the *physical layer*, concerned with such matters as copper, fiber optics, radio waves, voltages, and wavelengths. Above that is the *link layer*, which provides the service of transmitting a chunk of data to another computer on the same local network. This is the point where the operating system becomes involved. Building on the link-layer foundation, the operating system provides the services of the *network layer* and the *transport layer*. The network layer arranges for data to be relayed through interconnected networks so as to arrive at a computer that may be elsewhere in the world. The transport layer builds on top of this basic computer-to-computer data transmission to provide more useful application-to-application communication channels. For example, the transport layer typically uses sequence numbering and retransmission to provide applications the service of in-order, loss-free delivery of streams of data. This is the level of the most common operating system API, which provides *sockets*, that is, endpoints for these transport-layer connections.

The next layer up is the *application layer*. A few specialized application-layer services, such as distributed file systems, are integrated with operating

systems. However, most application-layer software, such as web browsers and email programs, is written by application programmers. These applications can be built directly on an operating system's socket API and exchange streams of bytes that comply with standardized protocols. In Chapter 9, I illustrate this possibility by showing how web browsers and web servers communicate.

Alternatively, programmers of distributed applications can make use of middleware to work at a higher level than sending bytes over sockets. I show two basic approaches to this in Chapter 10: messaging and Remote Procedure Calls (RPCs). Web services are a particular approach to standardizing these kinds of higher-level application communication, and have often been used with RPCs.

In a *messaging* system, an application program requests the delivery of a message. The messaging system not only delivers the message, which lower-level networking could accomplish, but also provides additional services. For example, the messaging is often integrated with transaction processing. A successful transaction may retrieve a message from an incoming message queue, update a database in response to that message, and send a response message to an outgoing queue. If the transaction fails, none of these three changes will happen; the request message will remain in the incoming queue, the database will remain unchanged, and the response message will not be queued for further delivery. Another common service provided by messaging systems is to deliver a message to any number of recipients who have subscribed to receive messages of a particular kind; the sender need not be aware of who the actual receivers are.

Middleware can also provide a mechanism for *Remote Procedure Call* (*RPC*), in which communication between a client and a server is made to look like an ordinary programming language procedure call, such as invoking a method on an object. The only difference is that the object in question is located on a different computer, and so the call and return involve network communication. The middleware hides this complexity, so that the application programmer can work largely as though all the objects were local. In Chapter 10, I explain this concept more fully and mention that it is often used in the form of web services. A *web service* is an application-layer entity that programs can communicate with using standardized protocols similar to those humans use to browse the web.

1.9 Security

Operating systems and middleware are often the targets of attacks by adversaries trying to defeat system security. Even attacks aimed at application programs often relate to operating systems and middleware. In particular, easily misused features of operating systems and middleware can be the root cause of an application-level vulnerability. On the other hand, operating systems and middleware provide many features that can be very helpful in constructing secure systems.

A system is secure if it provides an acceptably low risk that an adversary will prevent the system from achieving its owner's objectives. In Chapter 11, I explain in more detail how to think about risk and about the conflicting objectives of system owners and adversaries. In particular, I explain that some of the most common objectives for owners fall into four categories: confidentiality, integrity, availability, and accountability. A system provides *confidentiality* if it prevents inappropriate disclosure of information, *integrity* if it prevents inappropriate modification or destruction of information, and *availability* if it prevents inappropriate interference with legitimate usage. A system provides *accountability* if it provides ways to check how authorized users have exercised their authority. All of these rely on *authentication*, the ability of a system to verify the identity of a user.

Many people have a narrow view of system security. They think of those features that would not even exist, were it not for security issues. Clearly, logging in with a password (or some other, better form of authentication) is a component of system security. Equally clearly, having permission to read some files, but not others, is a component of system security, as are cryptographic protocols used to protect network communication from interception. However, this view of security is dangerously incomplete.

You need to keep in mind that the design of any component of the operating system can have security consequences. Even those parts whose design is dominated by other considerations must also reflect some proactive consideration of security consequences, or the overall system will be insecure. In fact, this is an important principle that extends beyond the operating system to include application software and the humans who operate it.

Therefore, I will make a habit of addressing security issues in every chapter, rather than only at the end of the book. Specifically, each chapter concludes with a section pointing out some of the key security issues associated with that chapter's topic. I also provide a more coherent treatment of security by concluding the book as a whole with Chapter 11, which is devoted exclusively to security. That chapter takes a holistic approach to

security, in which human factors play as important a role as technical ones.

Exercises

1.1 What is the difference between an operating system and middleware?

1.2 What do operating systems and middleware have in common?

1.3 What is the relationship between threads and processes?

1.4 What is one way an operating system might isolate threads from unwanted interactions, and what is one way that middleware might do so?

1.5 What is one way an operating system might provide persistent storage, and what is one way middleware might do so?

1.6 What is one way an operating system might support network communication, and what is one way middleware might do so?

1.7 Of all the topics previewed in this chapter, which one are you most looking forward to learning more about? Why?

Programming Project

1.1 Write, test, and debug a program in the language of your choice to carry out any task you choose. Then write a list of all the services you suspect the operating system is providing in order to support the execution of your sample program. If you think the program is also relying on any middleware services, list those as well.

Exploration Projects

1.1 Look through the titles of the papers presented at several recent conferences hosted by the USENIX Association (The Advanced Computing Systems Association); you can find the conference proceedings at *www.usenix.org*. To get a better idea what an individual paper is about, click the title to show the abstract, which is a short summary of the paper. Based on titles and abstracts, pick out a few papers that you think would make interesting supplementary reading as you work

your way through this book. Write down a list showing the bibliographic information for the papers you selected and, as near as you can estimate, where in this book's table of contents they would be appropriate to read.

1.2 Conduct a simple experiment in which you take some action on a computer system and observe what the response is. You can choose any action you wish and any computer system for which you have appropriate access. You can either observe a quantitative result, such as how long the response takes or how much output is produced, or a qualitative result, such as in what form the response arrives. Now, try replicating the experiment. Do you always get the same result? Similar ones? Are there any factors that need to be controlled in order to get results that are at least approximately repeatable? For example, to get consistent times, do you need to reboot the system between each trial and prevent other people from using the system? To get consistent output, do you need to make sure input files are kept unchanged? If your action involves a physical device, such as a printer, do you have to control variables such as whether the printer is stocked with paper? Finally, write up a careful report, in which you explain both what experiment you tried and what results you observed. You should explain how repeatable the results proved to be and what limits there were on the repeatability. You should describe the hardware and software configuration in enough detail that someone else could replicate your experiment and would be likely to get similar results.

Notes

The idea that an operating system should isolate computations from unwanted interactions, and yet support desirable interactions, has a long heritage. A 1962 paper [38] by Corbató, Daggett, and Daley points out that "different user programs if simultaneously in core memory may interfere with each other or the supervisor program so some form of memory protection mode should be available when operating user programs." However, that same paper goes on to say that although "great care went into making each user independent of the other users ... it would be a useful extension of the system if this were not always the case," so that the computer system could support group work, such as war games.

Middleware is not as well-known to the general public as operating systems are, though commercial information-system developers would be lost without it. One attempt to introduce middleware to a somewhat broader audience was Bernstein's 1996 survey article [17].

The USENIX Association, mentioned in Exploration Project 1.1, is only one of several very fine professional societies holding conferences related to the subject matter of this book. The reason why I specifically recommended looking through their proceedings is that they tend to be particularly accessible to students. In part this is because USENIX focuses on bringing practitioners and academics together; thus, the papers generally are pragmatic without being superficial. The full text is available on their website.

Chapter 2

Threads

2.1 Introduction

Computer programs consist of instructions, and computers carry out sequences of computational steps specified by those instructions. We call each sequence of computational steps that are strung together one after another a *thread*. The simplest programs to write are single-threaded, with instructions that should be executed one after another in a single sequence. However, in Section 2.2, you will learn how to write programs that produce more than one thread of execution, each an independent sequence of computational steps, with few if any ordering constraints between the steps in one thread and those in another. Multiple threads can also come into existence by running multiple programs, or by running the same program more than once.

Note the distinction between a program and a thread; the program contains instructions, whereas the thread consists of the execution of those instructions. Even for single-threaded programs, this distinction matters. If a program contains a loop, then a very short program could give rise to a very long thread of execution. Also, running the same program ten times will give rise to ten threads, all executing one program. Figure 2.1 summarizes how threads arise from programs.

Each thread has a lifetime, extending from the time its first instruction execution occurs until the time of its last instruction execution. If two threads have overlapping lifetimes, as illustrated in Figure 2.2, we say they are *concurrent*. One of the most fundamental goals of an operating system is to allow multiple threads to run concurrently on the same computer. That is, rather than waiting until the first thread has completed before a

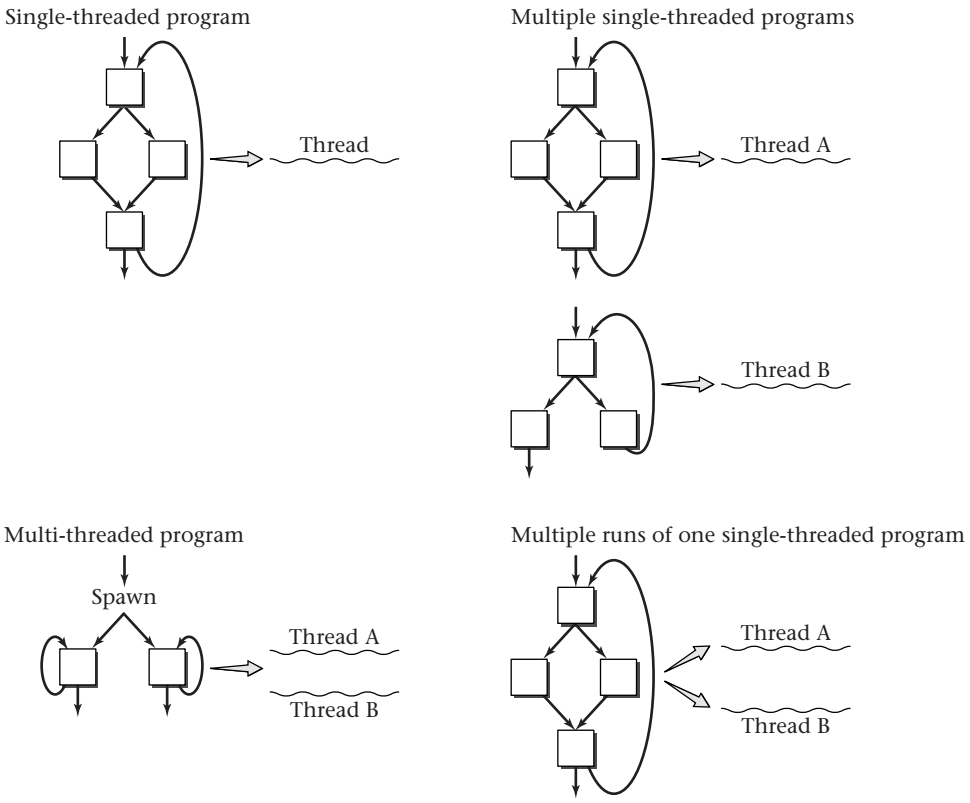


Figure 2.1: Programs give rise to threads.

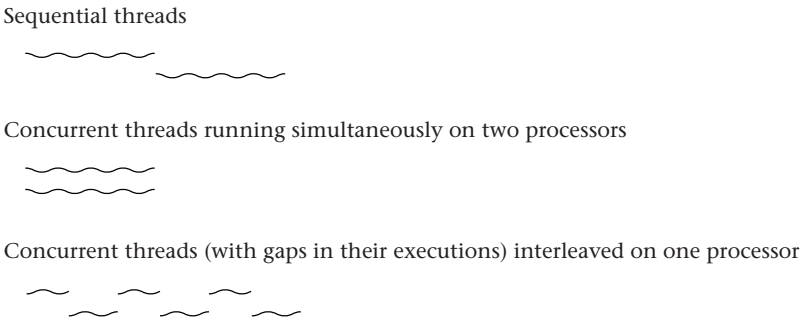


Figure 2.2: Sequential and concurrent threads

second thread can run, it should be possible to divide the computer's attention between them. If the computer hardware includes multiple processors, then it will naturally be possible to run threads concurrently, one per processor. However, the operating system's users will often want to run more concurrent threads than the hardware has processors, for reasons described in Section 2.3. Therefore, the operating system will need to divide each processor's attention between multiple threads. In this introductory textbook I will mostly limit myself to the case of all the threads needing to be run on a single processor. I will explicitly indicate those places where I do address the more general multi-processor case.

In order to make the concept of concurrent threads concrete, Section 2.2 shows how to write a program that spawns multiple threads each time the program is run. Once you know how to create threads, I will explain in Section 2.3 some of the reasons why it is desirable to run multiple threads concurrently and will offer some typical examples of the uses to which threads are put.

These first two sections explain the application programmer's view of threads: how and why the programmer would use concurrent threads. This sets us up for the next question: how does the operating system support the application programmer's desire for concurrently executing threads? In Sections 2.4 and 2.5, we will examine how the system does so. In this chapter, we will consider only the fundamentals of how the processor's attention is switched from one thread to another. Some of the related issues I address in other chapters include deciding which thread to run at each point (Chapter 3) and controlling interaction among the threads (Chapters 4, 5, 6, and 7). Also, as explained in Chapter 1, I will wait until Chapter 7 to explain the protection boundary surrounding the operating system. Thus, I will need to wait until that chapter to distinguish threads that reside entirely within that boundary, threads provided from inside the boundary for use outside of it, and threads residing entirely outside the boundary (known as *user-level threads* or, in Microsoft Windows, *fibers*).

Finally, the chapter concludes with the standard features of this book: a brief discussion of security issues, followed by exercises, programming and exploration projects, and notes.

2.2 Example of Multithreaded Programs

Whenever a program initially starts running, the computer carries out the program's instructions in a single thread. Therefore, if the program is in-

tended to run in multiple threads, the original thread needs at some point to spawn off a child thread that does some actions, while the parent thread continues to do others. (For more than two threads, the program can repeat the thread-creation step.) Most programming languages have an application programming interface (or API) for threads that includes a way to create a child thread. In this section, I will use the Java API and the API for C that is called *pthreads*, for *POSIX threads*. (As you will see throughout the book, POSIX is a comprehensive specification for UNIX-like systems, including many APIs beyond just thread creation.)

Realistic multithreaded programming requires the control of thread interactions, using techniques I show in Chapter 4. Therefore, my examples in this chapter are quite simple, just enough to show the spawning of threads.

To demonstrate the independence of the two threads, I will have both the parent and the child thread respond to a timer. One will sleep three seconds and then print out a message. The other will sleep five seconds and then print out a message. Because the threads execute concurrently, the second message will appear approximately two seconds after the first. (In Programming Projects 2.1, 2.2, and 2.3, you can write a somewhat more realistic program, where one thread responds to user input and the other to the timer.)

Figure 2.3 shows the Java version of this program. The `main` program first creates a `Thread` object called `childThread`. The `Runnable` object associated with the child thread has a `run` method that sleeps three seconds (expressed as 3000 milliseconds) and then prints a message. This `run` method starts running when the main procedure invokes `childThread.start()`. Because the `run` method is in a separate thread, the main thread can continue on to the subsequent steps, sleeping five seconds (5000 milliseconds) and printing its own message.

Figure 2.4 is the equivalent program in C, using the pthreads API. The `child` procedure sleeps three seconds and prints a message. The `main` procedure creates a `child_thread` running the `child` procedure, and then itself sleeps five seconds and prints a message. The most significant difference from the Java API is that `pthread_create` both creates the child thread and starts it running, whereas in Java those are two separate steps.

In addition to portable APIs, such as the Java and pthreads APIs, many systems provide their own non-portable APIs. For example, Microsoft Windows has the Win32 API, with procedures such as `CreateThread` and `Sleep`. In Programming Project 2.4, you can modify the program from Figure 2.4 to use this API.

```
public class Simple2Threads {
  public static void main(String args[]){
    Thread childThread = new Thread(new Runnable(){
        public void run(){
          sleep(3000);
          System.out.println("Child is done sleeping 3 seconds.");
        }
      });
    childThread.start();
    sleep(5000);
    System.out.println("Parent is done sleeping 5 seconds.");
  }

  private static void sleep(int milliseconds){
    try{
      Thread.sleep(milliseconds);
    } catch(InterruptedException e){
      // ignore this exception; it won't happen anyhow
    }
  }
}
```

Figure 2.3: A simple multithreaded program in Java

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

static void *child(void *ignored){
  sleep(3);
  printf("Child is done sleeping 3 seconds.\n");
  return NULL;
}

int main(int argc, char *argv[]){
  pthread_t child_thread;
  int code;

  code = pthread_create(&child_thread, NULL, child, NULL);
  if(code){
    fprintf(stderr, "pthread_create failed with code %d\n", code);
  }
  sleep(5);
  printf("Parent is done sleeping 5 seconds.\n");
  return 0;
}
```

Figure 2.4: A simple multithreaded program in C

2.3 Reasons for Using Concurrent Threads

You have now seen how a single execution of one program can result in more than one thread. Presumably, you were already at least somewhat familiar with generating multiple threads by running multiple programs, or by running the same program multiple times. Regardless of how the threads come into being, we are faced with a question. Why is it desirable for the computer to execute multiple threads concurrently, rather than waiting for one to finish before starting another? Fundamentally, most uses for concurrent threads serve one of two goals:

Responsiveness: allowing the computer system to respond quickly to something external to the system, such as a human user or another computer system. Even if one thread is in the midst of a long computation, another thread can respond to the external agent. Our example programs in Section 2.2 illustrated responsiveness: both the parent and the child thread responded to a timer.

Resource utilization: keeping most of the hardware resources busy most of the time. If one thread has no need for a particular piece of hardware, another may be able to make productive use of it.

Each of these two general themes has many variations, some of which we explore in the remainder of this section. A third reason why programmers sometimes use concurrent threads is as a tool for modularization. With this, a complex system may be decomposed into a group of interacting threads.

Let's start by considering the responsiveness of a web server, which provides many client computers with the specific web pages they request over the Internet. Whenever a client computer makes a network connection to the server, it sends a sequence of bytes that contain the name of the desired web page. Therefore, before the server program can respond, it needs to read in those bytes, typically using a loop that continues reading in bytes from the network connection until it sees the end of the request. Suppose one of the clients is connecting using a very slow network connection, perhaps via a dial-up modem. The server may read the first part of the request and then have to wait a considerable length of time before the rest of the request arrives over the network. What happens to other clients in the meantime? It would be unacceptable for a whole website to grind to a halt, unable to serve any clients, just waiting for one slow client to finish issuing its request. One way some web servers avoid this unacceptable situation is by using multiple threads, one for each client connection, so that even if

one thread is waiting for data from one client, other threads can continue interacting with the other clients. Figure 2.5 illustrates the unacceptable single-threaded web server and the more realistic multithreaded one.

On the client side, a web browser may also illustrate the need for responsiveness. Suppose you start loading in a very large web page, which takes considerable time to download. Would you be happy if the computer froze up until the download finished? Probably not. You expect to be able to work on a spreadsheet in a different window, or scroll through the first part of the web page to read as much as has already downloaded, or at least click on the Stop button to give up on the time-consuming download. Each of these can be handled by having one thread tied up loading the web page over the network, while another thread is responsive to your actions at the keyboard and mouse.

This web browser scenario also lets me foreshadow later portions of the textbook concerning the controlled interaction between threads. Note that I sketched several different things you might want to do while the web page downloaded. In the first case, when you work on a spreadsheet, the two concurrent threads have almost nothing to do with one another, and the operating system's job, beyond allowing them to run concurrently, will mostly consist of isolating each from the other, so that a bug in the web browser doesn't overwrite part of your spreadsheet, for example. This is generally done by encapsulating the threads in separate protection environments known as *processes*, as we will discuss in Chapters 6 and 7. (Some systems call processes *tasks*, while others use *task* as a synonym for *thread*.) If, on the other hand, you continue using the browser's user interface while the download continues, the concurrent threads are closely related parts of a

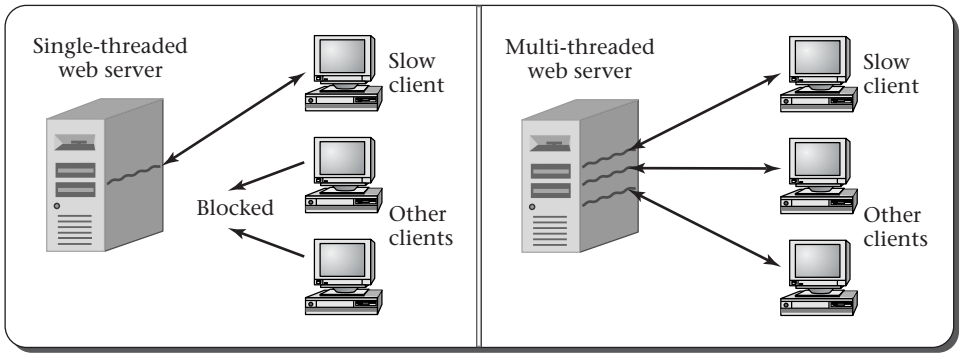


Figure 2.5: Single-threaded and multithreaded web servers

single application, and the operating system need not isolate the threads from one another. However, it may still need to provide mechanisms for regulating their interaction. For example, some coordination between the downloading thread and the user-interface thread is needed to ensure that you can scroll through as much of the page as has been downloaded, but no further. This coordination between threads is known as *synchronization* and is the topic of Chapters 4 and 5.

Turning to the utilization of hardware resources, the most obvious scenario is when you have a dual-processor computer. In this case, if the system ran only one thread at a time, only half the processing capacity would ever be used. Even if the human user of the computer system doesn't have more than one task to carry out, there may be useful housekeeping work to keep the second processor busy. For example, most operating systems, if asked to allocate memory for an application program's use, will store all zeros into the memory first. Rather than holding up each memory allocation while the zeroing is done, the operating system can have a thread that proactively zeros out unused memory, so that when needed, it will be all ready. If this housekeeping work (zeroing of memory) were done on demand, it would slow down the system's real work; by using a concurrent thread to utilize the available hardware more fully, the performance is improved. This example also illustrates that not all threads need to come from user programs. A thread can be part of the operating system itself, as in the example of the thread zeroing out unused memory.

Even in a single-processor system, resource utilization considerations may justify using concurrent threads. Remember that a computer system contains hardware resources, such as disk drives, other than the processor. Suppose you have two tasks to complete on your PC: you want to scan all the files on disk for viruses, and you want to do a complicated photo-realistic rendering of a three-dimensional scene including not only solid objects, but also shadows cast on partially transparent smoke clouds. From experience, you know that each of these will take about an hour. If you do one and then the other, it will take two hours. If instead you do the two concurrently—running the virus scanner in one window while you run the graphics rendering program in another window—you may be pleasantly surprised to find both jobs done in only an hour and a half.

The explanation for the half-hour savings in elapsed time is that the virus scanning program spends most of its time using the disk drive to read files, with only modest bursts of processor activity each time the disk completes a read request, whereas the rendering program spends most of its time doing processing, with very little disk activity. As illustrated in Figure 2.6, running

them in sequence leaves one part of the computer's hardware idle much of the time, whereas running the two concurrently keeps the processor and disk drive both busy, improving the overall system efficiency. Of course, this assumes the operating system's scheduler is smart enough to let the virus scanner have the processor's attention (briefly) whenever a disk request completes, rather than making it wait for the rendering program. I will address this issue in Chapter 3.

As you have now seen, threads can come from multiple sources and serve multiple roles. They can be internal portions of the operating system, as in the example of zeroing out memory, or part of the user's application software. In the latter case, they can either be dividing up the work within a multithreaded process, such as the web server and web browser examples, or can come from multiple independent processes, as when a web browser runs in one window and a spreadsheet in another. Regardless of these variations, the typical reasons for running the threads concurrently remain unchanged: either to provide increased responsiveness or to improve system efficiency by more fully utilizing the hardware. Moreover, the basic mechanism used to divide the processor's attention among multiple threads remains the same in these different cases as well; I describe that mechanism in Sections 2.4 and 2.5. Of course, some cases require the additional protection mechanisms provided by processes, which we discuss in Chapters 6 and 7. However, even then, it is still necessary to leave off work on one thread and pick up work on another.

2.4 Switching Between Threads

In order for the operating system to have more than one thread underway on a processor, the system needs to have some mechanism for switching attention between threads. In particular, there needs to be some way to leave off from in the middle of a thread's sequence of instructions, work for a while on other threads, and then pick back up in the original thread right where it left off. In order to explain thread switching as simply as possible, I will initially assume that each thread is executing code that contains, every once in a while, explicit instructions to temporarily switch to another thread. Once you understand this mechanism, I can then build on it for the more realistic case where the thread contains no explicit thread-switching points, but rather is automatically interrupted for thread switches.

Suppose we have two threads, A and B, and we use A1, A2, A3, and so forth as names for the instruction execution steps that constitute A, and

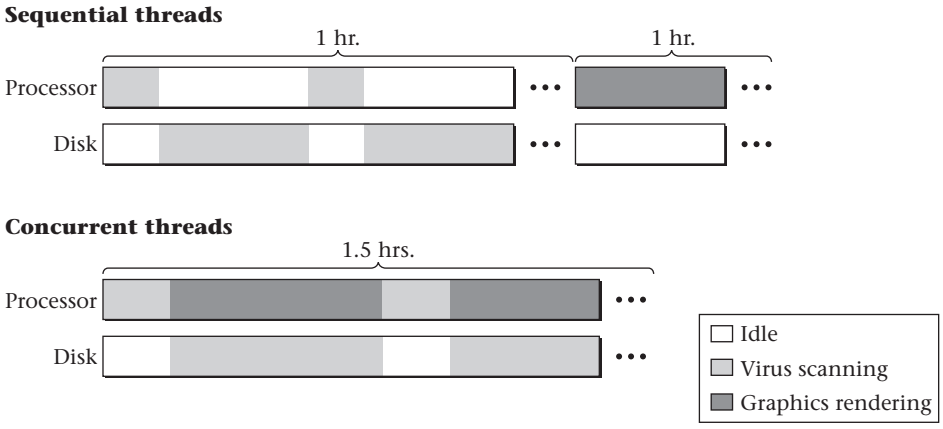


Figure 2.6: Overlapping processor-intensive and disk-intensive activities

similarly for B. In this case, one possible execution sequence might be as shown in Figure 2.7. As I will explain subsequently, when thread A executes `switchFromTo(A,B)` the computer starts executing instructions from thread B. In a more realistic example, there might be more than two threads, and each might run for many more steps (both between switches and overall), with only occasionally a new thread starting or an existing thread exiting.

Our goal is that the steps of each thread form a coherent execution sequence. That is, from the perspective of thread A, its execution should not be much different from one in which A1 through A8 occurred consecutively, without interruption, and similarly for thread B's steps B1 through B9. Suppose, for example, steps A1 and A2 load two values from memory into registers, A3 adds them, placing the sum in a register, and A4 doubles that register's contents, so as to get twice the sum. In this case, we want to make sure that A4 really does double the sum computed by A1 through A3, rather than doubling some other value that thread B's steps B1 through B3 happen to store in the same register. Thus, we can see that switching threads cannot simply be a matter of a jump instruction transferring control to the appropriate instruction in the other thread. At a minimum, we will also have to save registers into memory and restore them from there, so that when a thread resumes execution, its own values will be back in the registers.

In order to focus on the essentials, let's put aside the issue of how threads start and exit. Instead, let's focus just on the normal case where one thread in progress puts itself on hold and switches to another thread where that

Thread A	**Thread B**
A1	
A2	
A3	
`switchFromTo(A,B)`	
	B1
	B2
	B3
	`switchFromTo(B,A)`
A4	
A5	
`switchFromTo(A,B)`	
	B4
	B5
	B6
	B7
	`switchFromTo(B,A)`
A6	
A7	
A8	
`switchFromTo(A,B)`	
	B8
	B9

Figure 2.7: Switching between threads

other thread last left off, such as the switch from A5 to B4 in the preceding example. To support switching threads, the operating system will need to keep information about each thread, such as at what point that thread should resume execution. If this information is stored in a block of memory for each thread, then we can use the addresses of those memory areas to refer to the threads. The block of memory containing information about a thread is called a *thread control block* or *task control block* (*TCB*). Thus, another way of saying that we use the addresses of these blocks is to say that we use pointers to thread control blocks to refer to threads.

Our fundamental thread-switching mechanism will be the `switchFromTo` procedure, which takes two of these thread control block pointers as parameters: one specifying the thread that is being switched out of, and one specifying the next thread, which is being switched into. In our running example, `A` and `B` are pointer variables pointing to the two threads' control blocks, which we use alternately in the roles of outgoing thread and next thread. For example, the program for thread A contains code after instruction A5 to switch from `A` to `B`, and the program for thread B contains code after instruction B3 to switch from `B` to `A`. Of course, this assumes that each thread knows both its own identity and the identity of the thread to switch to. Later, we will see how this unrealistic assumption can be eliminated. For now, though, let's see how we could write the `switchFromTo` procedure so that `switchFromTo(A, B)` would save the current execution status information into the structure pointed to by `A`, read back previously saved information from the structure pointed to by `B`, and resume where thread B left off.

We already saw that the execution status information to save includes not only a position in the program, often called the *program counter* (*PC*) or *instruction pointer* (*IP*), but also the contents of registers. Another critical part of the execution status for programs compiled with most higher level language compilers is a portion of the memory used to store a stack, along with a stack pointer register that indicates the position in memory of the current top of the stack. You likely have encountered this form of storage in some prior course—computer organization, programming language principles, or even introduction to computer science. If not, Appendix A provides the information you will need before proceeding with the remainder of this chapter.

When a thread resumes execution, it must find the stack the way it left it. For example, suppose thread A pushes two items on the stack and then is put on hold for a while, during which thread B executes. When thread A resumes execution, it should find the two items it pushed at the top of the

stack—even if thread B did some pushing of its own and has not yet gotten around to popping. We can arrange for this by giving each thread its own stack, setting aside a separate portion of memory for each of them. When thread A is executing, the *stack pointer* (or SP register) will be pointing somewhere within thread A's stack area, indicating how much of that area is occupied at that time. Upon switching to thread B, we need to save away A's stack pointer, just like other registers, and load in thread B's stack pointer. That way, while thread B is executing, the stack pointer will move up and down within B's stack area, in accordance with B's own pushes and pops.

Having discovered this need to have separate stacks and switch stack pointers, we can simplify the saving of all other registers by pushing them onto the stack before switching and popping them off the stack after switching, as shown in Figure 2.8. We can use this approach to outline the code for switching from the outgoing thread to the next thread, using `outgoing` and `next` as the two pointers to thread control blocks. (When switching from `A` to `B`, `outgoing` will be `A` and `next` will be `B`. Later, when switching back from `B` to `A`, `outgoing` will be `B` and `next` will be `A`.) We will use `outgoing->SP` and `outgoing->IP` to refer to two slots within the structure pointed to by `outgoing`, the slot used to save the stack pointer and the one used to save the instruction pointer. With these assumptions, our code has the following general form:

```
  push each register on the (outgoing thread's) stack
  store the stack pointer into outgoing->SP
  load the stack pointer from next->SP
  store label L's address into outgoing->IP
  load in next->IP and jump to that address
L:
  pop each register from the (resumed outgoing thread's) stack
```

Note that the code before the label (`L`) is done at the time of switching away from the outgoing thread, whereas the code after that label is done later, upon resuming execution when some other thread switches back to the original one.

This code not only stores the outgoing thread's stack pointer away, but also restores the next thread's stack pointer. Later, the same code will be used to switch back. Therefore, we can count on the original thread's stack pointer to have been restored when control jumps to label `L`. Thus, when the registers are popped, they will be popped from the original thread's stack, matching the pushes at the beginning of the code.

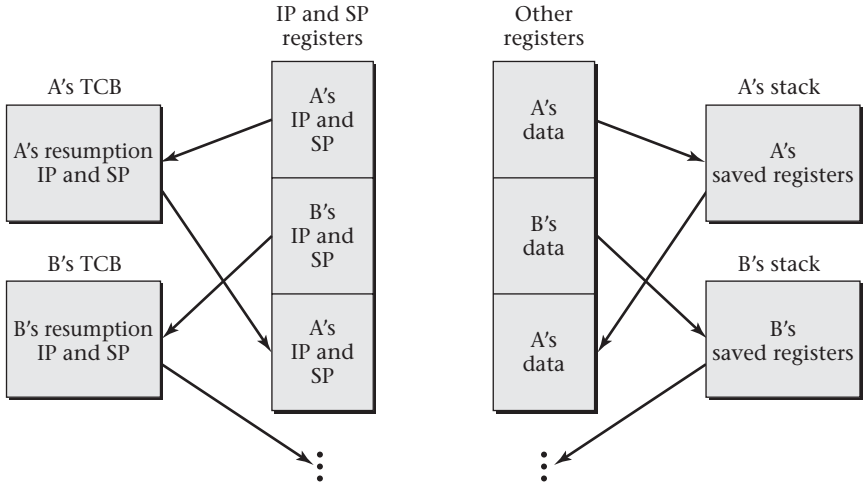


Figure 2.8: Saving registers in thread control blocks and per-thread stacks

We can see how this general pattern plays out in a real system, by looking at the thread-switching code from the Linux operating system for the i386 architecture. (The i386 architecture is also known as the x86 or IA-32; it is a popular processor architecture used in standard personal computer processors such as Intel's Core, Xeon, and Atom families and AMD's FX and Opteron families.) If you don't want to see real code, you can skip ahead to the paragraph after the block of assembly code. However, even if you aren't familiar with i386 assembly language, you ought to be able to see how this code matches the preceding pattern.

This is real code extracted from the Linux kernel, though with some peripheral complications left out. The stack pointer register is named `%esp`, and when this code starts running, the registers known as `%ebx` and `%esi` contain the `outgoing` and `next` pointers, respectively. Each of those pointers is the address of a thread control block. The location at offset 812 within the TCB contains the thread's instruction pointer, and the location at offset 816 contains the thread's stack pointer. (That is, these memory locations contain the instruction pointer and stack pointer to use when resuming that thread's execution.) The code surrounding the thread switch does not keep any important values in most of the other registers; only the special flags register and the register named `%ebp` need to be saved and restored. With that as background, here is the code, with explanatory comments:

```
pushfl                  # pushes the flags on outgoing's stack
pushl %ebp              # pushes %ebp on outgoing's stack
```

```
   movl %esp,816(%ebx)    # stores outgoing's stack pointer
   movl 816(%esi),%esp    # loads next's stack pointer
   movl $1f,812(%ebx)     # stores label 1's address,
                          #    where outgoing will resume
   pushl 812(%esi)        # pushes the instruction address
                          #    where next resumes
   ret                    # pops and jumps to that address
1: popl %ebp              # upon later resuming outgoing,
                          #    restores %ebp
   popfl                  # and restores the flags
```

Having seen the core idea of how a processor is switched from running one thread to running another, we can now eliminate the assumption that each thread switch contains the explicit names of the outgoing and next threads. That is, we want to get away from having to name threads `A` and `B` in `switchFromTo(A, B)`. It is easy enough to know which thread is being switched away from, if we just keep track at all times of the currently running thread, for example, by storing a pointer to its control block in a global variable called `current`. That leaves the question of which thread is being selected to run next. What we will do is have the operating system keep track of all the threads in some sort of data structure, such as a list. There will be a procedure, `chooseNextThread()`, which consults that data structure and, using some scheduling policy, decides which thread to run next. In Chapter 3, I will explain how this scheduling is done; for now, take it as a black box. Using this tool, one can write a procedure, `yield()`, which performs the following four steps:

```
outgoing = current;
next = chooseNextThread();
current = next;   // so the global variable will be right
switchFromTo(outgoing, next);
```

Now, every time a thread decides it wants to take a break and let other threads run for a while, it can just invoke `yield()`. This is essentially the approach taken by real systems, such as Linux. One complication in a multiprocessor system is that the `current` thread needs to be recorded on a per-processor basis.

Thread switching is often called *context switching*, because it switches from the execution context of one thread to that of another thread. Many authors, however, use the phrase *context switching* differently, to refer to switching processes with their protection contexts—a topic we will discuss

in Chapter 7. If the distinction matters, the clearest choice is to avoid the ambiguous term *context switching* and use the more specific *thread switching* or *process switching*.

Thread switching is the most common form of *dispatching* a thread, that is, of causing a processor to execute it. The only way a thread can be dispatched without a thread switch is if a processor is idle.

2.5 Preemptive Multitasking

At this point, I have explained thread switching well enough for systems that employ *cooperative multitasking*, that is, where each thread's program contains explicit code at each point where a thread switch should occur. However, more realistic operating systems use what is called *preemptive multitasking*, in which the program's code need not contain any thread switches, yet thread switches will none the less automatically be performed from time to time.

One reason to prefer preemptive multitasking is because it means that buggy code in one thread cannot hold all others up. Consider, for example, a loop that is expected to iterate only a few times; it would seem safe, in a cooperative multitasking system, to put thread switches only before and after it, rather than also in the loop body. However, a bug could easily turn the loop into an infinite one, which would hog the processor forever. With preemptive multitasking, the thread may still run forever, but at least from time to time it will be put on hold and other threads allowed to progress.

Another reason to prefer preemptive multitasking is that it allows thread switches to be performed when they best achieve the goals of responsiveness and resource utilization. For example, the operating system can preempt a thread when input becomes available for a waiting thread or when a hardware device falls idle.

Even with preemptive multitasking, it may occasionally be useful for a thread to voluntarily give way to the other threads, rather than to run as long as it is allowed. Therefore, even preemptive systems normally provide `yield()`. The name varies depending on the API, but often has `yield` in it; for example, the pthreads API uses the name `sched_yield()`. One exception to this naming pattern is the Win32 API of Microsoft Windows, which uses the name `SwitchToThread()` for the equivalent of `yield()`.

Preemptive multitasking does not need any fundamentally different thread switching mechanism; it simply needs the addition of a hardware interrupt mechanism. In case you are not familiar with how interrupts work, I will

first take a moment to review this aspect of hardware organization.

Normally a processor will execute consecutive instructions one after another, deviating from sequential flow only when directed by an explicit jump instruction or by some variant such as the `ret` instruction used in the Linux code for thread switching. However, there is always some mechanism by which external hardware (such as a disk drive or a network interface) can signal that it needs attention. A hardware timer can also be set to demand attention periodically, such as every millisecond. When an I/O device or timer needs attention, an *interrupt* occurs, which is almost as though a procedure call instruction were forcibly inserted between the currently executing instruction and the next one. Thus, rather than moving on to the program's next instruction, the processor jumps off to the special procedure called the *interrupt handler*.

The interrupt handler, which is part of the operating system, deals with the hardware device and then executes a *return from interrupt* instruction, which jumps back to the instruction that had been about to execute when the interrupt occurred. Of course, in order for the program's execution to continue as expected, the interrupt handler needs to be careful to save all the registers at the start and restore them before returning.

Using this interrupt mechanism, an operating system can provide preemptive multitasking. When an interrupt occurs, the interrupt handler first saves the registers to the current thread's stack and takes care of the immediate needs, such as accepting data from a network interface controller or updating the system's idea of the current time by one millisecond. Then, rather than simply restoring the registers and executing a return from interrupt instruction, the interrupt handler checks whether it would be a good time to preempt the current thread and switch to another.

For example, if the interrupt signaled the arrival of data for which a thread had long been waiting, it might make sense to switch to that thread. Or, if the interrupt was from the timer and the current thread had been executing for a long time, it may make sense to give another thread a chance. These policy decisions are related to scheduling, the topic of Chapter 3.

In any case, if the operating system decides to preempt the current thread, the interrupt handler switches threads using a mechanism such as the `switchFromTo` procedure. This switching of threads includes switching to the new thread's stack, so when the interrupt handler restores registers before returning, it will be restoring the new thread's registers. The previously running thread's register values will remain safely on its own stack until that thread is resumed.

2.6 Security and Threads

One premise of this book is that every topic raises its own security issues. Multithreading is no exception. However, this section will be quite brief, because with the material covered in this chapter, I can present only the security problems connected with multithreading, not the solutions. So that I do not divide problems from their solutions, this section provides only a thumbnail sketch, leaving serious consideration of the problems and their solutions to the chapters that introduce the necessary tools.

Security issues arise when some threads are unable to execute because others are hogging the computer's attention. Security issues also arise because of unwanted interactions between threads. Unwanted interactions include a thread writing into storage that another thread is trying to use or reading from storage another thread considers confidential. These problems are most likely to arise if the programmer has a difficult time understanding how the threads may interact with one another.

The security section in Chapter 3 addresses the problem of some threads monopolizing the computer. The security sections in Chapters 4, 5, and 7 address the problem of controlling threads' interaction. Each of these chapters also has a strong emphasis on design approaches that make interactions easy to understand, thereby minimizing the risks that arise from incomplete understanding.

Exercises

2.1 Based on the examples in Section 2.2, name at least one difference between the `sleep` procedure in the POSIX API and the `Thread.sleep` method in the Java API.

2.2 Give at least three more examples, beyond those given in the text, where it would be useful to run more concurrent threads on a computer than that computer's number of processors. Indicate how your examples fit the general reasons to use concurrency listed in the text.

2.3 Suppose thread A goes through a loop 100 times, each time performing one disk I/O operation, taking 10 milliseconds, and then some computation, taking 1 millisecond. While each 10-millisecond disk operation is in progress, thread A cannot make any use of the processor. Thread B runs for 1 second, purely in the processor, with no I/O. One millisecond of processor time is spent each time the processor switches

threads; other than this switching cost, there is no problem with the processor working on thread B during one of thread A's I/O operations. (The processor and disk drive do not contend for memory access bandwidth, for example.)

(a) Suppose the processor and disk work purely on thread A until its completion, and then the processor switches to thread B and runs all of that thread. What will the total elapsed time be?

(b) Suppose the processor starts out working on thread A, but every time thread A performs a disk operation, the processor switches to B during the operation and then back to A upon the disk operation's completion. What will the total elapsed time be?

2.4 Consider a uniprocessor system where each arrival of input from an external source triggers the creation and execution of a new thread, which at its completion produces some output. We are interested in the response time from triggering input to resulting output.

(a) Input arrives at time 0 and again after 1 second, 2 seconds, and so forth. Each arrival triggers a thread that takes 600 milliseconds to run. Before the thread can run, it must be created and dispatched, which takes 10 milliseconds. What is the average response time for these inputs?

(b) Now a second source of input is added, with input arriving at times 0.1 seconds, 1.1 seconds, 2.1 seconds, and so forth. These inputs trigger threads that only take 100 milliseconds to run, but they still need 10 milliseconds to create and dispatch. When an input arrives, the resulting new thread is not created or dispatched until the processor is idle. What is the average response time for this second class of inputs? What is the combined average response time for the two classes?

(c) Suppose we change the way the second class of input is handled. When the input arrives, the new thread is immediately created and dispatched, even if that preempts an already running thread. When the new thread completes, the preempted thread resumes execution after a 1 millisecond thread switching delay. What is the average response time for each class of inputs? What is the combined average for the two together?

2.5 When control switches away from a thread and later switches back to that thread, the thread resumes execution where it left off. Simi-

larly, when a procedure calls a subroutine and later the subroutine returns, execution picks back up where it left off in the calling procedure. Given this similarity, what is the essential difference between thread switching and subroutine call/return? You saw that each thread has a separate stack, each in its own area of memory. Why is this not necessary for subroutine invocations?

Programming Projects

2.1 If you program in C, read the documentation for `pthread_cancel`. Using this information and the model provided in Figure 2.4 on page 26, write a program where the initial (main) thread creates a second thread. The main thread should read input from the keyboard, waiting until the user presses the Enter key. At that point, it should kill off the second thread and print out a message reporting that it has done so. Meanwhile, the second thread should be in an infinite loop, each time around sleeping five seconds and then printing out a message. Try running your program. Can the sleeping thread print its periodic messages while the main thread is waiting for keyboard input? Can the main thread read input, kill the sleeping thread, and print a message while the sleeping thread is in the early part of one of its five-second sleeps?

2.2 If you program in Java, read the documentation for the `stop` method in the `Thread` class. (Ignore the information about it being deprecated. That will make sense only after you read Chapter 4 of this book.) Write the program described in Programming Project 2.1, except do so in Java. You can use the program shown in Figure 2.3 on page 25 as a model.

2.3 Read the API documentation for some programming language other than C, C++, or Java to find out how to spawn off a thread and how to sleep. Write a program in this language equivalent to the Java and C example programs in Figures 2.3 and 2.4 on pages 25 and 26. Then do the equivalent of Programming Projects 2.1 and 2.2 using the language you have chosen.

2.4 If you program in C under Microsoft Windows, you can use the native Win32 API instead of the portable pthreads API. Read the documentation of `CreateThread` and `Sleep` and modify the program of Figure 2.4 on page 26 to use these procedures.

Exploration Projects

2.1 Try the experiment of running a disk-intensive process and a processor-intensive process concurrently. Write a report carefully explaining what you did and in which hardware and software system context you did it, so that someone else could replicate your results. Your report should show how the elapsed time for the concurrent execution compared with the times from sequential execution. Be sure to do multiple trials and to reboot the system before each run so as to eliminate effects that come from keeping disk data in memory for re-use. If you can find documentation for any performance-monitoring tools on your system, which would provide information such as the percentage of CPU time used or the number of disk I/O operations per second, you can include this information in your report as well.

2.2 Early versions of Microsoft Windows and Mac OS used cooperative multitasking. Use the web, or other sources of information, to find out when each switched to preemptive multitasking. Can you find and summarize any examples of what was written about this change at the time?

2.3 How frequently does a system switch threads? You can find this out on a Linux system by using the `vmstat` program. Read the man page for `vmstat`, and then run it to find the number of context switches per second. Write a report in which you carefully explain what you did and the hardware and software system context in which you did it, so that someone else could replicate your results.

Notes

The idea of executing multiple threads concurrently seems to have occurred to several people (more or less concurrently) in the late 1950s. They did not use the word *thread*, however. For example, a 1959 article by E. F. Codd et al. [34] stated that "the second form of parallelism, which we shall call *nonlocal*, provides for concurrent execution of instructions which need not be neighbors in an instruction stream, but which may belong, if you please, to entirely separate and unrelated programs." From the beginning, authors were aware of both reasons for using concurrency that I have emphasized (resource utilization and responsiveness). The same article by Codd et al., for example, reports that "one object of concurrently running tasks which

belong to different (perhaps totally unrelated) programs is to achieve a more balanced loading of the facilities than would be possible if all the tasks belonged to a single program. Another object is to achieve a specified real-time response in a situation in which messages, transactions, etc., are to be processed on-line."

I mentioned that an operating system may dedicate a thread to preemptively zeroing out memory. One example of this is the *zero page thread* in Microsoft Windows. See Russinovich and Solomon's book [126] for details.

I extracted the Linux thread switching code from version 2.6.0-test1 of the kernel. Details (such as the offsets 812 and 816) may differ in other versions. The kernel source code is written in a combination of assembly language and C, contained in `include/asm-i386/system.h` as included into `kernel/sched.c`. To obtain pure assembly code, I fed the source through the `gcc` compiler. Also, the `ret` instruction is a simplification; the actual kernel at that point jumps to a block of code that ends with the `ret` instruction.

My brief descriptions of the POSIX and Java APIs are intended only as concrete illustrations of broader concepts, not as a replacement for documentation of those APIs. You can find the official documentation on the web at *http://www.unix.org* and *http://java.sun.com*, respectively.

Chapter 3

Scheduling

3.1 Introduction

In Chapter 2 you saw that operating systems support the concurrent execution of multiple threads by repeatedly switching each processor's attention from one thread to another. This switching implies that some mechanism, known as a *scheduler*, is needed to choose which thread to run at each time. Other system resources may need scheduling as well; for example, if several threads read from the same disk drive, a disk scheduler may place them in order. For simplicity, I will consider only processor scheduling. Normally, when people speak of *scheduling*, they mean processor scheduling; similarly, the *scheduler* is understood to mean the processor scheduler.

A scheduler should make decisions in a way that keeps the computer system's users happy. For example, picking the same thread all the time and completely ignoring the others would generally not be a good scheduling policy. Unfortunately, there is no one policy that will make all users happy all the time. Sometimes the reason is as simple as different users having conflicting desires: for example, user A wants task A completed quickly, while user B wants task B completed quickly. Other times, though, the relative merits of different scheduling policies will depend not on whom you ask, but rather on the context in which you ask. As a simple example, a student enrolled in several courses is unlikely to decide which assignment to work on without considering when the assignments are due.

Because scheduling policies need to respond to context, operating systems provide scheduling mechanisms that leave the user in charge of more subtle policy choices. For example, an operating system may provide a mechanism for running whichever thread has the highest numerical priority,

while leaving the user the job of assigning priorities to the threads. Even so, no one mechanism (or general family of policies) will suit all goals. Therefore, I spend much of this chapter describing the different goals that users have for schedulers and the mechanisms that can be used to achieve those goals, at least approximately. Particularly since users may wish to achieve several conflicting goals, they will generally have to be satisfied with "good enough."

Before I get into the heavily values-laden scheduling issues, though, I will present one goal everyone can agree upon: A thread that can make productive use of a processor should always be preferred over one that is waiting for something, such as the completion of a time delay or the arrival of input. In Section 3.2, you will see how schedulers arrange for this by keeping track of each thread's state and scheduling only those that can run usefully.

Following the section on thread states, I devote Section 3.3 entirely to the question of users' goals, independent of how they are realized. Then I spend one section apiece on three broad families of schedulers, examining for each not only how it works but also how it can serve users' goals. These three families of schedulers are those based on fixed thread priorities (Section 3.4), those based on dynamically adjusted thread priorities (Section 3.5), and those based less on priorities than on controlling each thread's proportional share of processing time (Section 3.6). This three-way division is not the only possible taxonomy of schedulers, but it will serve to help me introduce several operating systems' schedulers and explain the principles behind them while keeping in mind the context of users' goals. After presenting the three families of schedulers, I will briefly remark in Section 3.7 on the role scheduling plays in system security. The chapter concludes with exercises, programming and exploration projects, and notes.

3.2 Thread States

A typical thread will have times when it is waiting for some event, unable to execute any useful instructions until the event occurs. Consider a web server that reads a client's request from the network, reads the requested web page from disk, and then sends the page over the network to the client. Initially the server thread is waiting for the network interface to have some data available. If the server thread were scheduled on a processor while it was waiting, the best it could do would be to execute a loop that checked over and over whether any data has arrived—hardly a productive use of the

processor's time. Once data is available from the network, the server thread can execute some useful instructions to read the bytes in and check whether the request is complete. If not, the server needs to go back to waiting for more data to arrive. Once the request is complete, the server will know what page to load from disk and can issue the appropriate request to the disk drive. At that point, the thread once again needs to wait until such time as the disk has completed the requisite physical movements to locate the page. To take a different example, a video display program may display one frame of video and then wait some fraction of a second before displaying the next so that the movie doesn't play too fast. All the thread could do between frames would be to keep checking the computer's real-time clock to see whether enough time had elapsed—again, not a productive use of the processor.

In a single-thread system, it is plausible to wait by executing a loop that continually checks for the event in question. This approach is known as *busy waiting*. However, a modern general-purpose operating system will have multiple threads competing for the processor. In this case, busy waiting is a bad idea because any time that the scheduler allocates to the busy-waiting thread is lost to the other threads without achieving any added value for the thread that is waiting.

Therefore, operating systems provide an alternative way for threads to wait. The operating system keeps track of which threads can usefully run and which are waiting. The system does this by storing runnable threads in a data structure called the *run queue* and waiting threads in *wait queues*, one per reason for waiting. Although these structures are conventionally called queues, they may not be used in the first-in, first-out style of true queues. For example, there may be a list of threads waiting for time to elapse, kept in order of the desired time. Another example of a wait queue would be a set of threads waiting for the availability of data on a particular network communication channel.

Rather than executing a busy-waiting loop, a thread that wants to wait for some event notifies the operating system of this intention. The operating system removes the thread from the run queue and inserts the thread into the appropriate wait queue, as shown in Figure 3.1. Because the scheduler considers only threads in the run queue for execution, it will never select the waiting thread to run. The scheduler will be choosing only from those threads that can make progress if given a processor on which to run.

In Chapter 2, I mentioned that the arrival of a hardware interrupt can cause the processor to temporarily stop executing instructions from the current thread and to start executing instructions from the operating system's

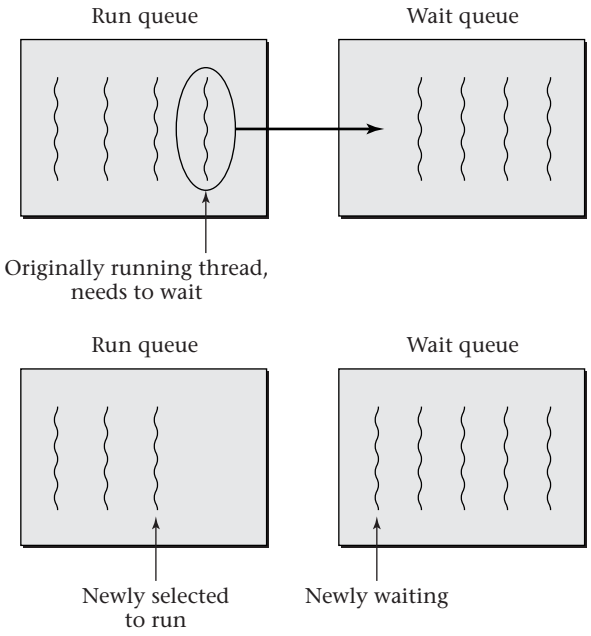


Figure 3.1: When a thread needs to wait, the operating system moves it from the run queue to a wait queue. The scheduler selects one of the threads remaining in the run queue to dispatch, so it starts running. An animated GIF version of this figure is also available on this book's web site.

interrupt handler. One of the services this interrupt handler can perform is determining that a waiting thread doesn't need to wait any longer. For example, the computer's real-time clock may be configured to interrupt the processor every one hundredth of a second. The interrupt handler could check the first thread in the wait queue of threads that are waiting for specific times to elapse. If the time this thread was waiting for has not yet arrived, no further threads need to be checked because the threads are kept in time order. If, on the other hand, the thread has slept as long as it requested, then the operating system can move it out of the list of sleeping threads and into the run queue, where the thread is available for scheduling. In this case, the operating system should check the next thread similarly, as illustrated in Figure 3.2.

Putting together the preceding information, there are at least three distinct states a thread can be in:

- *Runnable* (but not running), awaiting dispatch by the scheduler
- *Running* on a processor
- *Waiting* for some event

Some operating systems may add a few more states in order to make finer distinctions (waiting for one kind of event versus waiting for another kind) or to handle special circumstances (for example, a thread that has finished running, but needs to be kept around until another thread is notified). For simplicity, I will stick to the three basic states in the foregoing list. At critical moments in the thread's lifetime, the operating system will change the thread's state. These thread state changes are indicated in Figure 3.3. Again, a real operating system may add a few additional transitions; for example, it may be possible to forcibly terminate a thread, even while it is in a waiting state, rather than having it terminate only of its own accord while running.

3.3 Scheduling Goals

Users expect a scheduler to maximize the computer system's performance and to allow them to exert control. Each of these goals can be refined into several more precise goals, which I explain in the following subsections. High performance may mean high throughput (Section 3.3.1) or fast response time (Section 3.3.2), and user control may be expressed in terms of urgency, importance, or resource allocation (Section 3.3.3).

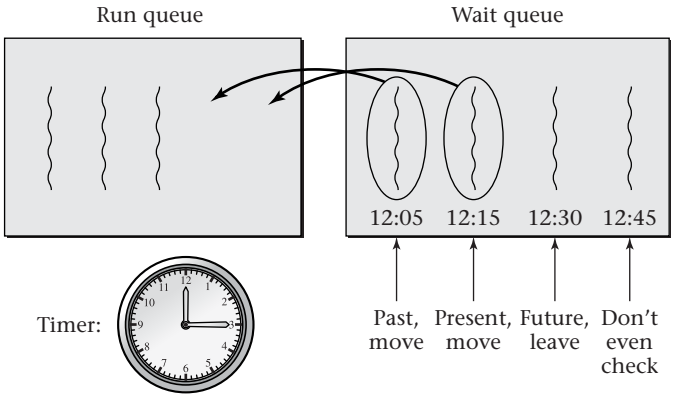


Figure 3.2: When the operating system handles a timer interrupt, all threads waiting for times that have now past are moved to the run queue. Because the wait queue is kept in time order, the scheduler need only check threads until it finds one waiting for a time still in the future. In this figure, times are shown on a human scale for ease of understanding. An animated GIF version of this figure is also available on this book's web site.

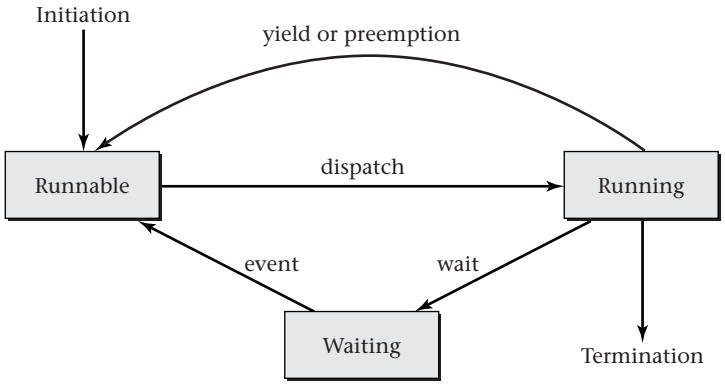


Figure 3.3: Threads change states as shown here. When a thread is initially created, it is runnable, but not actually running on a processor until dispatched by the scheduler. A running thread can voluntarily yield the processor or can be preempted by the scheduler in order to run another thread. In either case, the formerly running thread returns to the runnable state. Alternatively, a running thread may wait for an external event before becoming runnable again. A running thread may also terminate.

3.3.1 Throughput

Many personal computers have far more processing capability available than work to do, and they largely sit idle, patiently waiting for the next keystroke from a user. However, if you look behind the scenes at a large Internet service, such as Google, you'll see a very different situation. Large rooms filled with rack after rack of computers are necessary in order to keep up with the pace of incoming requests; any one computer can cope only with a small fraction of the traffic. For economic reasons, the service provider wants to keep the cluster of servers as small as possible. Therefore, the throughput of each server must be as high as possible. The *throughput* is the rate at which useful work, such as search transactions, is accomplished. An example measure of throughput would be the number of search transactions completed per second.

Maximizing throughput certainly implies that the scheduler should give each processor a runnable thread on which to work, if at all possible. However, there are some other, slightly less obvious, implications as well. Remember that a computer system has more components than just processors. It also has I/O devices (such as disk drives and network interfaces) and a memory hierarchy, including cache memories. Only by using all these resources efficiently can a scheduler maximize throughput.

I already mentioned I/O devices in Chapter 2, with the example of a computationally intensive graphics rendering program running concurrently with a disk-intensive virus scanner. I will return to this example later in the current chapter to see one way in which the two threads can be efficiently interleaved. In a nutshell, the goal is to keep both the processor and the disk drive busy all the time. If you have ever had an assistant for a project, you may have some appreciation for what this entails: whenever your assistant was in danger of falling idle, you had to set your own work aside long enough to explain the next assignment. Similarly, the processor must switch threads when necessary to give the disk more work to do.

Cache memories impact throughput-oriented scheduling in two ways, though one arises only in multiprocessor systems. In any system, switching between different threads more often than necessary will reduce throughput because processor time will be wasted on the overhead of context switching, rather than be available for useful work. The main source of this context-switching overhead is not the direct cost of the switch itself, which entails saving a few registers out and loading them with the other thread's values. Instead, the big cost is in reduced cache memory performance, for reasons I will explain in a moment. On multiprocessor systems a second issue arises: