ANDREW S. TANENBAUM
HERBERT BOS

MODERN OPERATING SYSTEMS

Fourth Edition

MODERN OPERATING SYSTEMS

FOURTH EDITION

Trademarks

MODERN OPERATING SYSTEMS

FOURTH EDITION

ANDREW S. TANENBAUM
HERBERT BOS

Vrije Universiteit
Amsterdam, The Netherlands

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montréal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Vice President and Editorial Director, ECS: *Marcia Horton*
Executive Editor: *Tracy Johnson*
Program Management Team Lead: *Scott Disanno*
Program Manager: *Carole Snyder*
Project Manager: *Camille Trentacoste*
Operations Specialist: *Linda Sager*
Cover Design: *Black Horse Designs*
Cover art: *Jason Consalvo*
Media Project Manager: *Renata Butera*

Library of Congress Cataloging-in-Publication Data
On file

PEARSON

ISBN-10: 0-13-359162-X
ISBN-13: 978-0-13-359162-0

To Suzanne, Barbara, Daniel, Aron, Nathan, Marvin, Matilde, and Olivia. The list keeps growing. (AST)

To Marieke, Duko, Jip, and Spot. Fearsome Jedi, all. (HB)

CONTENTS

11 CASE STUDY 2: WINDOWS 8 857

PREFACE

The fourth edition of this book differs from the third edition in numerous ways. There are large numbers of small changes everywhere to bring the material up to date as operating systems are not standing still. The chapter on Multimedia Operating Systems has been moved to the Web, primarily to make room for new material and keep the book from growing to a completely unmanageable size. The chapter on Windows Vista has been removed completely as Vista has not been the success Microsoft hoped for. The chapter on Symbian has also been removed, as Symbian no longer is widely available. However, the Vista material has been replaced by Windows 8 and Symbian has been replaced by Android. Also, a completely new chapter, on virtualization and the cloud has been added. Here is a chapter-by-chapter rundown of the changes.

- Chapter 1 has been heavily modified and updated in many places but with the exception of a new section on mobile computers, no major sections have been added or deleted.
- Chapter 2 has been updated, with older material removed and some new material added. For example, we added the futex synchronization primitive, and a section about how to avoid locking altogether with Read-Copy-Update.
- Chapter 3 now has more focus on modern hardware and less emphasis on segmentation and MULTICS.
- In Chapter 4 we removed CD-Roms, as they are no longer very common, and replaced them with more modern solutions (like flash drives). Also, we added RAID level 6 to the section on RAID.

- Chapter 5 has seen a lot of changes. Older devices like CRTs and CD-ROMs have been removed, while new technology, such as touch screens have been added.
- Chapter 6 is pretty much unchanged. The topic of deadlocks is fairly stable, with few new results.
- Chapter 7 is completely new. It covers the important topics of virtualization and the cloud. As a case study, a section on VMware has been added.
- Chapter 8 is an updated version of the previous material on multiprocessor systems. There is more emphasis on multicore and manycore systems now, which have become increasingly important in the past few years. Cache consistency has become a bigger issue recently and is covered here, now.
- Chapter 9 has been heavily revised and reorganized, with considerable new material on exploiting code bugs, malware, and defenses against them. Attacks such as null pointer dereferences and buffer overflows are treated in more detail. Defense mechanisms, including canaries, the NX bit, and address-space randomization are covered in detail now, as are the ways attackers try to defeat them.
- Chapter 10 has undergone a major change. The material on UNIX and Linux has been updated but the major addtion here is a new and lengthy section on the Android operating system, which is very common on smartphones and tablets.
- Chapter 11 in the third edition was on Windows Vista. That has been replaced by a chapter on Windows 8, specifically Windows 8.1. It brings the treatment of Windows completely up to date.
- Chapter 12 is a revised version of Chap. 13 from the previous edition.
- Chapter 13 is a thoroughly updated list of suggested readings. In addition, the list of references has been updated, with entries to 223 new works published after the third edition of this book came out.
- Chapter 7 from the previous edition has been moved to the book's Website to keep the size somewhat manageable).
- In addition, the sections on research throughout the book have all been redone from scratch to reflect the latest research in operating systems. Furthermore, new problems have been added to all the chapters.

Numerous teaching aids for this book are available. Instructor supplements can be found at *www.pearsonhighered.com/tanenbaum*. They include PowerPoint sheets, software tools for studying operating systems, lab experiments for students, simulators, and more material for use in operating systems courses. Instructors using this book in a course should definitely take a look. The Companion Website for this book is also located at *www.pearsonhighered.com/tanenbaum*. The specific site for this book is password protected. To use the site, click on the picture of the cover and then follow the instructions on the student access card that came with your text to create a user account and log in. Student resources include:

- An online chapter on Multimedia Operating Systems
- Lab Experiments
- Online Exercises
- Simulation Exercises

A number of people have been involved in the fourth edition. First and foremost, Prof. Herbert Bos of the Vrije Universiteit in Amsterdam has been added as a coauthor. He is a security, UNIX, and all-around systems expert and it is great to have him on board. He wrote much of the new material except as noted below.

Our editor, Tracy Johnson, has done a wonderful job, as usual, of herding all the cats, putting all the pieces together, putting out fires, and keeping the project on schedule. We were also fortunate to get our long-time production editor, Camille Trentacoste, back. Her skills in so many areas have saved the day on more than a few occasions. We are glad to have her again after an absence of several years. Carole Snyder did a fine job coordinating the various people involved in the book.

The material in Chap. 7 on VMware (in Sec. 7.12) was written by Edouard Bugnion of EPFL in Lausanne, Switzerland. Ed was one of the founders of the VMware company and knows this material as well as anyone in the world. We thank him greatly for supplying it to us.

Ada Gavrilovska of Georgia Tech, who is an expert on Linux internals, updated Chap. 10 from the Third Edition, which she also wrote. The Android material in Chap. 10 was written by Dianne Hackborn of Google, one of the key developers of the Android system. Android is the leading operating system on smartphones, so we are very grateful to have Dianne help us. Chap. 10 is now quite long and detailed, but UNIX, Linux, and Android fans can learn a lot from it. It is perhaps worth noting that the longest and most technical chapter in the book was written by two women. We just did the easy stuff.

We haven't neglected Windows, however. Dave Probert of Microsoft updated Chap. 11 from the previous edition of the book. This time the chapter covers Windows 8.1 in detail. Dave has a great deal of knowledge of Windows and enough vision to tell the difference between places where Microsoft got it right and where it got it wrong. Windows fans are certain to enjoy this chapter.

The book is much better as a result of the work of all these expert contributors. Again, we would like to thank them for their invaluable help.

We were also fortunate to have several reviewers who read the manuscript and also suggested new end-of-chapter problems. These were Trudy Levine, Shivakant Mishra, Krishna Sivalingam, and Ken Wong. Steve Armstrong did the PowerPoint sheets for instructors teaching a course using the book.

Normally copyeditors and proofreaders don't get acknowledgements, but Bob Lentz (copyeditor) and Joe Ruddick (proofreader) did exceptionally thorough jobs. Joe in particular, can spot the difference between a roman period and an italics period from 20 meters. Nevertheless, the authors take full responsibility for any residual errors in the book. Readers noticing any errors are requested to contact one of the authors.

Finally, last but not least, Barbara and Marvin are still wonderful, as usual, each in a unique and special way. Daniel and Matilde are great additions to our family. Aron and Nathan are wonderful little guys and Olivia is a treasure. And of course, I would like to thank Suzanne for her love and patience, not to mention all the *druiven*, *kersen*, and *sinaasappelsap*, as well as other agricultural products. (AST)

Most importantly, I would like to thank Marieke, Duko, and Jip. Marieke for her love and for bearing with me all the nights I was working on this book, and Duko and Jip for tearing me away from it and showing me there are more important things in life. Like Minecraft. (HB)

Andrew S. Tanenbaum
Herbert Bos

ABOUT THE AUTHORS

Andrew S. Tanenbaum has an S.B. degree from M.I.T. and a Ph.D. from the University of California at Berkeley. He is currently a Professor of Computer Science at the Vrije Universiteit in Amsterdam, The Netherlands. He was formerly Dean of the Advanced School for Computing and Imaging, an interuniversity graduate school doing research on advanced parallel, distributed, and imaging systems. He was also an Academy Professor of the Royal Netherlands Academy of Arts and Sciences, which has saved him from turning into a bureaucrat. He also won a prestigious European Research Council Advanced Grant.

In the past, he has done research on compilers, operating systems, networking, and distributed systems. His main research focus now is reliable and secure operating systems. These research projects have led to over 175 refereed papers in journals and conferences. Prof. Tanenbaum has also authored or co-authored five books, which have been translated into 20 languages, ranging from Basque to Thai. They are used at universities all over the world. In all, there are 163 versions (language + edition combinations) of his books.

Prof. Tanenbaum has also produced a considerable volume of software, notably MINIX, a small UNIX clone. It was the direct inspiration for Linux and the platform on which Linux was initially developed. The current version of MINIX, called MINIX 3, is now focused on being an extremely reliable and secure operating system. Prof. Tanenbaum will consider his work done when no user has any idea what an operating system crash is. MINIX 3 is an ongoing open-source project to which you are invited to contribute. Go to *www.minix3.org* to download a free copy of MINIX 3 and give it a try. Both x86 and ARM versions are available.

Prof. Tanenbaum's Ph.D. students have gone on to greater glory after graduating. He is very proud of them. In this respect, he resembles a mother hen.

Prof. Tanenbaum is a Fellow of the ACM, a Fellow of the IEEE, and a member of the Royal Netherlands Academy of Arts and Sciences. He has also won numerous scientific prizes from ACM, IEEE, and USENIX. If you are unbearably curious about them, see his page on Wikipedia. He also has two honorary doctorates.

Herbert Bos obtained his Masters degree from Twente University and his Ph.D. from Cambridge University Computer Laboratory in the U.K.. Since then, he has worked extensively on dependable and efficient I/O architectures for operating systems like Linux, but also research systems based on MINIX 3. He is currently a professor in Systems and Network Security in the Dept. of Computer Science at the Vrije Universiteit in Amsterdam, The Netherlands. His main research field is system security. With his students, he works on novel ways to detect and stop attacks, to analyze and reverse engineer malware, and to take down botnets (malicious infrastructures that may span millions of computers). In 2011, he obtained an ERC Starting Grant for his research on reverse engineering. Three of his students have won the Roger Needham Award for best European Ph.D. thesis in systems.

This page intentionally left blank

MODERN OPERATING SYSTEMS

This page intentionally left blank

1

INTRODUCTION

A modern computer consists of one or more processors, some main memory, disks, printers, a keyboard, a mouse, a display, network interfaces, and various other input/output devices. All in all, a complex system.oo If every application programmer had to understand how all these things work in detail, no code would ever get written. Furthermore, managing all these components and using them optimally is an exceedingly challenging job. For this reason, computers are equipped with a layer of software called the **operating system**, whose job is to provide user programs with a better, simpler, cleaner, model of the computer and to handle managing all the resources just mentioned. Operating systems are the subject of this book.

Most readers will have had some experience with an operating system such as Windows, Linux, FreeBSD, or OS X, but appearances can be deceiving. The program that users interact with, usually called the **shell** when it is text based and the **GUI** (**Graphical User Interface**)—which is pronounced "gooey"—when it uses icons, is actually not part of the operating system, although it uses the operating system to get its work done.

A simple overview of the main components under discussion here is given in Fig. 1-1. Here we see the hardware at the bottom. The hardware consists of chips, boards, disks, a keyboard, a monitor, and similar physical objects. On top of the hardware is the software. Most computers have two modes of operation: kernel mode and user mode. The operating system, the most fundamental piece of software, runs in **kernel mode** (also called **supervisor mode**). In this mode it has complete access to all the hardware and can execute any instruction the machine is capable of executing. The rest of the software runs in **user mode**, in which only a subset of the machine instructions is available. In particular, those instructions that affect control of the machine or do **I/O**)**Input**/Output" are forbidden to user-mode programs. We will come back to the difference between kernel mode and user mode repeatedly throughout this book. It plays a crucial role in how operating systems work.

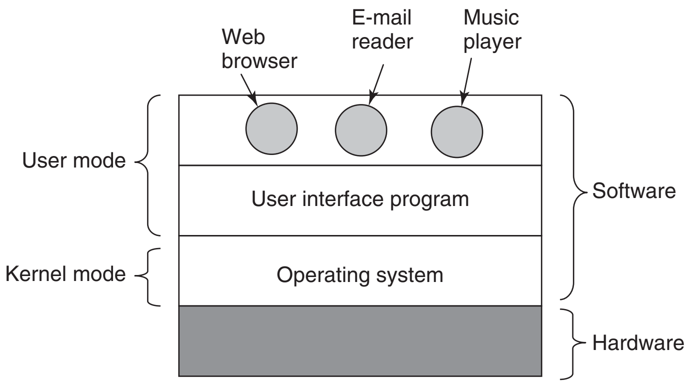


Figure 1-1. Where the operating system fits in.

The user interface program, shell or GUI, is the lowest level of user-mode software, and allows the user to start other programs, such as a Web browser, email reader, or music player. These programs, too, make heavy use of the operating system.

The placement of the operating system is shown in Fig. 1-1. It runs on the bare hardware and provides the base for all the other software.

An important distinction between the operating system and normal (user-mode) software is that if a user does not like a particular email reader, he† is free to get a different one or write his own if he so chooses; he is not free to write his own clock interrupt handler, which is part of the operating system and is protected by hardware against attempts by users to modify it.

This distinction, however, is sometimes blurred in embedded systems (which may not have kernel mode) or interpreted systems (such as Java-based systems that use interpretation, not hardware, to separate the components).

Also, in many systems there are programs that run in user mode but help the operating system or perform privileged functions. For example, there is often a program that allows users to change their passwords. It is not part of the operating system and does not run in kernel mode, but it clearly carries out a sensitive function and has to be protected in a special way. In some systems, this idea is carried to an extreme, and pieces of what is traditionally considered to be the operating

† "He" should be read as "he or she" throughout the book.

system (such as the file system) run in user space. In such systems, it is difficult to draw a clear boundary. Everything running in kernel mode is clearly part of the operating system, but some programs running outside it are arguably also part of it, or at least closely associated with it.

Operating systems differ from user (i.e., application) programs in ways other than where they reside. In particular, they are huge, complex, and long-lived. The source code of the heart of an operating system like Linux or Windows is on the order of five million lines of code or more. To conceive of what this means, think of printing out five million lines in book form, with 50 lines per page and 1000 pages per volume (larger than this book). It would take 100 volumes to list an operating system of this size—essentially an entire bookcase. Can you imagine getting a job maintaining an operating system and on the first day having your boss bring you to a bookcase with the code and say: “Go learn that.” And this is only for the part that runs in the kernel. When essential shared libraries are included, Windows is well over 70 million lines of code or 10 to 20 bookcases. And this excludes basic application software (things like Windows Explorer, Windows Media Player, and so on).

It should be clear now why operating systems live a long time—they are very hard to write, and having written one, the owner is loath to throw it out and start again. Instead, such systems evolve over long periods of time. Windows 95/98/Me was basically one operating system and Windows NT/2000/XP/Vista/Windows 7 is a different one. They look similar to the users because Microsoft made very sure that the user interface of Windows 2000/XP/Vista/Windows 7 was quite similar to that of the system it was replacing, mostly Windows 98. Nevertheless, there were very good reasons why Microsoft got rid of Windows 98. We will come to these when we study Windows in detail in Chap. 11.

Besides Windows, the other main example we will use throughout this book is UNIX and its variants and clones. It, too, has evolved over the years, with versions like System V, Solaris, and FreeBSD being derived from the original system, whereas Linux is a fresh code base, although very closely modeled on UNIX and highly compatible with it. We will use examples from UNIX throughout this book and look at Linux in detail in Chap. 10.

In this chapter we will briefly touch on a number of key aspects of operating systems, including what they are, their history, what kinds are around, some of the basic concepts, and their structure. We will come back to many of these important topics in later chapters in more detail.

1.1 WHAT IS AN OPERATING SYSTEM?

It is hard to pin down what an operating system is other than saying it is the software that runs in kernel mode—and even that is not always true. Part of the problem is that operating systems perform two essentially unrelated functions:

providing application programmers (and application programs, naturally) a clean abstract set of resources instead of the messy hardware ones and managing these hardware resources. Depending on who is doing the talking, you might hear mostly about one function or the other. Let us now look at both.

1.1.1 The Operating System as an Extended Machine

The **architecture** (instruction set, memory organization, I/O, and bus structure) of most computers at the machine-language level is primitive and awkward to program, especially for input/output. To make this point more concrete, consider modern **SATA** (**Serial ATA**) hard disks used on most computers. A book (Anderson, 2007) describing an early version of the interface to the disk—what a programmer would have to know to use the disk—ran over 450 pages. Since then, the interface has been revised multiple times and is more complicated than it was in 2007. Clearly, no sane programmer would want to deal with this disk at the hardware level. Instead, a piece of software, called a **disk driver**, deals with the hardware and provides an interface to read and write disk blocks, without getting into the details. Operating systems contain many drivers for controlling I/O devices.

But even this level is much too low for most applications. For this reason, all operating systems provide yet another layer of abstraction for using disks: files. Using this abstraction, programs can create, write, and read files, without having to deal with the messy details of how the hardware actually works.

This abstraction is the key to managing all this complexity. Good abstractions turn a nearly impossible task into two manageable ones. The first is defining and implementing the abstractions. The second is using these abstractions to solve the problem at hand. One abstraction that almost every computer user understands is the file, as mentioned above. It is a useful piece of information, such as a digital photo, saved email message, song, or Web page. It is much easier to deal with photos, emails, songs, and Web pages than with the details of SATA (or other) disks. The job of the operating system is to create good abstractions and then implement and manage the abstract objects thus created. In this book, we will talk a lot about abstractions. They are one of the keys to understanding operating systems.

This point is so important that it is worth repeating in different words. With all due respect to the industrial engineers who so carefully designed the Macintosh, hardware is ugly. Real processors, memories, disks, and other devices are very complicated and present difficult, awkward, idiosyncratic, and inconsistent interfaces to the people who have to write software to use them. Sometimes this is due to the need for backward compatibility with older hardware. Other times it is an attempt to save money. Often, however, the hardware designers do not realize (or care) how much trouble they are causing for the software. One of the major tasks of the operating system is to hide the hardware and present programs (and their programmers) with nice, clean, elegant, consistent, abstractions to work with instead. Operating systems turn the ugly into the beautiful, as shown in Fig. 1-2.

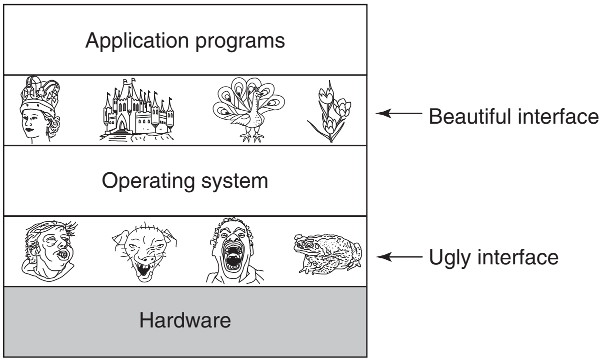


Figure 1-2. Operating systems turn ugly hardware into beautiful abstractions.

It should be noted that the operating system's real customers are the application programs (via the application programmers, of course). They are the ones who deal directly with the operating system and its abstractions. In contrast, end users deal with the abstractions provided by the user interface, either a command-line shell or a graphical interface. While the abstractions at the user interface may be similar to the ones provided by the operating system, this is not always the case. To make this point clearer, consider the normal Windows desktop and the line-oriented command prompt. Both are programs running on the Windows operating system and use the abstractions Windows provides, but they offer very different user interfaces. Similarly, a Linux user running Gnome or KDE sees a very different interface than a Linux user working directly on top of the underlying X Window System, but the underlying operating system abstractions are the same in both cases.

In this book, we will study the abstractions provided to application programs in great detail, but say rather little about user interfaces. That is a large and important subject, but one only peripherally related to operating systems.

1.1.2 The Operating System as a Resource Manager

The concept of an operating system as primarily providing abstractions to application programs is a top-down view. An alternative, bottom-up, view holds that the operating system is there to manage all the pieces of a complex system. Modern computers consist of processors, memories, timers, disks, mice, network interfaces, printers, and a wide variety of other devices. In the bottom-up view, the job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs wanting them.

Modern operating systems allow multiple programs to be in memory and run at the same time. Imagine what would happen if three programs running on some computer all tried to print their output simultaneously on the same printer. The first few lines of printout might be from program 1, the next few from program 2, then some from program 3, and so forth. The result would be utter chaos. The operating system can bring order to the potential chaos by buffering all the output destined for the printer on the disk. When one program is finished, the operating system can then copy its output from the disk file where it has been stored for the printer, while at the same time the other program can continue generating more output, oblivious to the fact that the output is not really going to the printer (yet).

When a computer (or network) has more than one user, the need for managing and protecting the memory, I/O devices, and other resources is even more since the users might otherwise interfere with one another. In addition, users often need to share not only hardware, but information (files, databases, etc.) as well. In short, this view of the operating system holds that its primary task is to keep track of which programs are using which resource, to grant resource requests, to account for usage, and to mediate conflicting requests from different programs and users.

Resource management includes **multiplexing** (sharing) resources in two different ways: in time and in space. When a resource is time multiplexed, different programs or users take turns using it. First one of them gets to use the resource, then another, and so on. For example, with only one CPU and multiple programs that want to run on it, the operating system first allocates the CPU to one program, then, after it has run long enough, another program gets to use the CPU, then another, and then eventually the first one again. Determining how the resource is time multiplexed—who goes next and for how long—is the task of the operating system. Another example of time multiplexing is sharing the printer. When multiple print jobs are queued up for printing on a single printer, a decision has to be made about which one is to be printed next.

The other kind of multiplexing is space multiplexing. Instead of the customers taking turns, each one gets part of the resource. For example, main memory is normally divided up among several running programs, so each one can be resident at the same time (for example, in order to take turns using the CPU). Assuming there is enough memory to hold multiple programs, it is more efficient to hold several programs in memory at once rather than give one of them all of it, especially if it only needs a small fraction of the total. Of course, this raises issues of fairness, protection, and so on, and it is up to the operating system to solve them. Another resource that is space multiplexed is the disk. In many systems a single disk can hold files from many users at the same time. Allocating disk space and keeping track of who is using which disk blocks is a typical operating system task.

1.2 HISTORY OF OPERATING SYSTEMS

Operating systems have been evolving through the years. In the following sections we will briefly look at a few of the highlights. Since operating systems have historically been closely tied to the architecture of the computers on which they run, we will look at successive generations of computers to see what their operating systems were like. This mapping of operating system generations to computer generations is crude, but it does provide some structure where there would otherwise be none.

The progression given below is largely chronological, but it has been a bumpy ride. Each development did not wait until the previous one nicely finished before getting started. There was a lot of overlap, not to mention many false starts and dead ends. Take this as a guide, not as the last word.

The first true digital computer was designed by the English mathematician Charles Babbage (1792–1871). Although Babbage spent most of his life and fortune trying to build his "analytical engine," he never got it working properly because it was purely mechanical, and the technology of his day could not produce the required wheels, gears, and cogs to the high precision that he needed. Needless to say, the analytical engine did not have an operating system.

As an interesting historical aside, Babbage realized that he would need software for his analytical engine, so he hired a young woman named Ada Lovelace, who was the daughter of the famed British poet Lord Byron, as the world's first programmer. The programming language Ada® is named after her.

1.2.1 The First Generation (1945–55): Vacuum Tubes

After Babbage's unsuccessful efforts, little progress was made in constructing digital computers until the World War II period, which stimulated an explosion of activity. Professor John Atanasoff and his graduate student Clifford Berry built what is now regarded as the first functioning digital computer at Iowa State University. It used 300 vacuum tubes. At roughly the same time, Konrad Zuse in Berlin built the Z3 computer out of electromechanical relays. In 1944, the Colossus was built and programmed by a group of scientists (including Alan Turing) at Bletchley Park, England, the Mark I was built by Howard Aiken at Harvard, and the ENIAC was built by William Mauchley and his graduate student J. Presper Eckert at the University of Pennsylvania. Some were binary, some used vacuum tubes, some were programmable, but all were very primitive and took seconds to perform even the simplest calculation.

In these early days, a single group of people (usually engineers) designed, built, programmed, operated, and maintained each machine. All programming was done in absolute machine language, or even worse yet, by wiring up electrical circuits by connecting thousands of cables to plugboards to control the machine's basic functions. Programming languages were unknown (even assembly language was unknown). Operating systems were unheard of. The usual mode of operation was for the programmer to sign up for a block of time using the signup sheet on the wall, then come down to the machine room, insert his or her plugboard into the computer, and spend the next few hours hoping that none of the 20,000 or so vacuum tubes would burn out during the run. Virtually all the problems were simple straightforward mathematical and numerical calculations, such as grinding out tables of sines, cosines, and logarithms, or computing artillery trajectories.

By the early 1950s, the routine had improved somewhat with the introduction of punched cards. It was now possible to write programs on cards and read them in instead of using plugboards; otherwise, the procedure was the same.

1.2.2 The Second Generation (1955–65): Transistors and Batch Systems

The introduction of the transistor in the mid-1950s changed the picture radically. Computers became reliable enough that they could be manufactured and sold to paying customers with the expectation that they would continue to function long enough to get some useful work done. For the first time, there was a clear separation between designers, builders, operators, programmers, and maintenance personnel.

These machines, now called **mainframes**, were locked away in large, specially air-conditioned computer rooms, with staffs of professional operators to run them. Only large corporations or major government agencies or universities could afford the multimillion-dollar price tag. To run a **job** (i.e., a program or set of programs), a programmer would first write the program on paper (in FORTRAN or assembler), then punch it on cards. He would then bring the card deck down to the input room and hand it to one of the operators and go drink coffee until the output was ready.

When the computer finished whatever job it was currently running, an operator would go over to the printer and tear off the output and carry it over to the output room, so that the programmer could collect it later. Then he would take one of the card decks that had been brought from the input room and read it in. If the FORTRAN compiler was needed, the operator would have to get it from a file cabinet and read it in. Much computer time was wasted while operators were walking around the machine room.

Given the high cost of the equipment, it is not surprising that people quickly looked for ways to reduce the wasted time. The solution generally adopted was the **batch system**. The idea behind it was to collect a tray full of jobs in the input room and then read them onto a magnetic tape using a small (relatively) inexpensive computer, such as the IBM 1401, which was quite good at reading cards, copying tapes, and printing output, but not at all good at numerical calculations. Other, much more expensive machines, such as the IBM 7094, were used for the real computing. This situation is shown in Fig. 1-3.

After about an hour of collecting a batch of jobs, the cards were read onto a magnetic tape, which was carried into the machine room, where it was mounted on a tape drive. The operator then loaded a special program (the ancestor of today's operating system), which read the first job from tape and ran it. The output was written onto a second tape, instead of being printed. After each job finished, the operating system automatically read the next job from the tape and began running

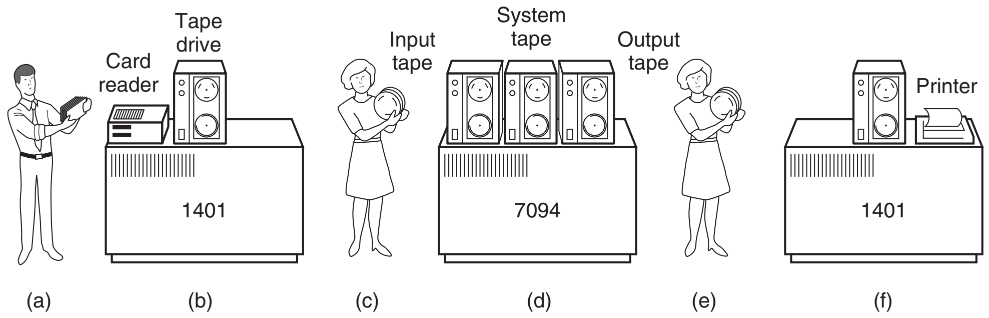


Figure 1-3. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

it. When the whole batch was done, the operator removed the input and output tapes, replaced the input tape with the next batch, and brought the output tape to a 1401 for printing **off line** (i.e., not connected to the main computer).

The structure of a typical input job is shown in Fig. 1-4. It started out with a $JOB card, specifying the maximum run time in minutes, the account number to be charged, and the programmer's name. Then came a $FORTRAN card, telling the operating system to load the FORTRAN compiler from the system tape. It was directly followed by the program to be compiled, and then a $LOAD card, directing the operating system to load the object program just compiled. (Compiled programs were often written on scratch tapes and had to be loaded explicitly.) Next came the $RUN card, telling the operating system to run the program with the data following it. Finally, the $END card marked the end of the job. These primitive control cards were the forerunners of modern shells and command-line interpreters.

Large second-generation computers were used mostly for scientific and engineering calculations, such as solving the partial differential equations that often occur in physics and engineering. They were largely programmed in FORTRAN and assembly language. Typical operating systems were FMS (the Fortran Monitor System) and IBSYS, IBM's operating system for the 7094.

1.2.3 The Third Generation (1965–1980): ICs and Multiprogramming

By the early 1960s, most computer manufacturers had two distinct, incompatible, product lines. On the one hand, there were the word-oriented, large-scale scientific computers, such as the 7094, which were used for industrial-strength numerical calculations in science and engineering. On the other hand, there were the

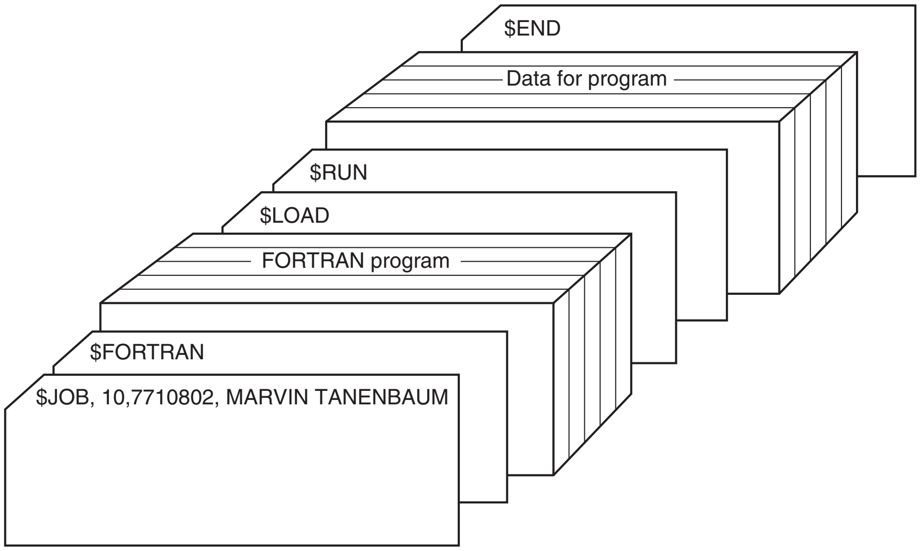


Figure 1-4. Structure of a typical FMS job.

character-oriented, commercial computers, such as the 1401, which were widely used for tape sorting and printing by banks and insurance companies.

Developing and maintaining two completely different product lines was an expensive proposition for the manufacturers. In addition, many new computer customers initially needed a small machine but later outgrew it and wanted a bigger machine that would run all their old programs, but faster.

IBM attempted to solve both of these problems at a single stroke by introducing the System/360. The 360 was a series of software-compatible machines ranging from 1401-sized models to much larger ones, more powerful than the mighty 7094. The machines differed only in price and performance (maximum memory, processor speed, number of I/O devices permitted, and so forth). Since they all had the same architecture and instruction set, programs written for one machine could run on all the others—at least in theory. (But as Yogi Berra reputedly said: “In theory, theory and practice are the same; in practice, they are not.”) Since the 360 was designed to handle both scientific (i.e., numerical) and commercial computing, a single family of machines could satisfy the needs of all customers. In subsequent years, IBM came out with backward compatible successors to the 360 line, using more modern technology, known as the 370, 4300, 3080, and 3090. The zSeries is the most recent descendant of this line, although it has diverged considerably from the original.

The IBM 360 was the first major computer line to use (small-scale) **ICs** (**Integrated Circuits**), thus providing a major price/performance advantage over the second-generation machines, which were built up from individual transistors. It was an immediate success, and the idea of a family of compatible computers was soon adopted by all the other major manufacturers. The descendants of these machines are still in use at computer centers today. Nowadays they are often used for managing huge databases (e.g., for airline reservation systems) or as servers for World Wide Web sites that must process thousands of requests per second.

The greatest strength of the "single-family" idea was simultaneously its greatest weakness. The original intention was that all software, including the operating system, **OS/360**, had to work on all models. It had to run on small systems, which often just replaced 1401s for copying cards to tape, and on very large systems, which often replaced 7094s for doing weather forecasting and other heavy computing. It had to be good on systems with few peripherals and on systems with many peripherals. It had to work in commercial environments and in scientific environments. Above all, it had to be efficient for all of these different uses.

There was no way that IBM (or anybody else for that matter) could write a piece of software to meet all those conflicting requirements. The result was an enormous and extraordinarily complex operating system, probably two to three orders of magnitude larger than FMS. It consisted of millions of lines of assembly language written by thousands of programmers, and contained thousands upon thousands of bugs, which necessitated a continuous stream of new releases in an attempt to correct them. Each new release fixed some bugs and introduced new ones, so the number of bugs probably remained constant over time.

One of the designers of OS/360, Fred Brooks, subsequently wrote a witty and incisive book (Brooks, 1995) describing his experiences with OS/360. While it would be impossible to summarize the book here, suffice it to say that the cover shows a herd of prehistoric beasts stuck in a tar pit. The cover of Silberschatz et al. (2012) makes a similar point about operating systems being dinosaurs.

Despite its enormous size and problems, OS/360 and the similar third-generation operating systems produced by other computer manufacturers actually satisfied most of their customers reasonably well. They also popularized several key techniques absent in second-generation operating systems. Probably the most important of these was **multiprogramming**. On the 7094, when the current job paused to wait for a tape or other I/O operation to complete, the CPU simply sat idle until the I/O finished. With heavily CPU-bound scientific calculations, I/O is infrequent, so this wasted time is not significant. With commercial data processing, the I/O wait time can often be 80 or 90% of the total time, so something had to be done to avoid having the (expensive) CPU be idle so much.

The solution that evolved was to partition memory into several pieces, with a different job in each partition, as shown in Fig. 1-5. While one job was waiting for I/O to complete, another job could be using the CPU. If enough jobs could be held in main memory at once, the CPU could be kept busy nearly 100% of the time. Having multiple jobs safely in memory at once requires special hardware to protect each job against snooping and mischief by the other ones, but the 360 and other third-generation systems were equipped with this hardware.

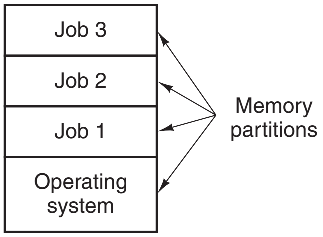


Figure 1-5. A multiprogramming system with three jobs in memory.

Another major feature present in third-generation operating systems was the ability to read jobs from cards onto the disk as soon as they were brought to the computer room. Then, whenever a running job finished, the operating system could load a new job from the disk into the now-empty partition and run it. This technique is called **spooling** (from **Simultaneous Peripheral Operation On Line**) and was also used for output. With spooling, the 1401s were no longer needed, and much carrying of tapes disappeared.

Although third-generation operating systems were well suited for big scientific calculations and massive commercial data-processing runs, they were still basically batch systems. Many programmers pined for the first-generation days when they had the machine all to themselves for a few hours, so they could debug their programs quickly. With third-generation systems, the time between submitting a job and getting back the output was often several hours, so a single misplaced comma could cause a compilation to fail, and the programmer to waste half a day. Programmers did not like that very much.

This desire for quick response time paved the way for **timesharing**, a variant of multiprogramming, in which each user has an online terminal. In a timesharing system, if 20 users are logged in and 17 of them are thinking or talking or drinking coffee, the CPU can be allocated in turn to the three jobs that want service. Since people debugging programs usually issue short commands (e.g., compile a five-page procedure†) rather than long ones (e.g., sort a million-record file), the computer can provide fast, interactive service to a number of users and perhaps also work on big batch jobs in the background when the CPU is otherwise idle. The first general-purpose timesharing system, **CTSS** (**Compatible Time Sharing System**), was developed at M.I.T. on a specially modified 7094 (Corbató et al., 1962). However, timesharing did not really become popular until the necessary protection hardware became widespread during the third generation.

After the success of the CTSS system, M.I.T., Bell Labs, and General Electric (at that time a major computer manufacturer) decided to embark on the development of a "computer utility," that is, a machine that would support some hundreds

†We will use the terms "procedure," "subroutine," and "function" interchangeably in this book.

of simultaneous timesharing users. Their model was the electricity system—when you need electric power, you just stick a plug in the wall, and within reason, as much power as you need will be there. The designers of this system, known as **MULTICS** (**MULTiplexed Information and Computing Service**), envisioned one huge machine providing computing power for everyone in the Boston area. The idea that machines 10,000 times faster than their GE-645 mainframe would be sold (for well under $1000) by the millions only 40 years later was pure science fiction. Sort of like the idea of supersonic trans-Atlantic undersea trains now.

MULTICS was a mixed success. It was designed to support hundreds of users on a machine only slightly more powerful than an Intel 386-based PC, although it had much more I/O capacity. This is not quite as crazy as it sounds, since in those days people knew how to write small, efficient programs, a skill that has subsequently been completely lost. There were many reasons that MULTICS did not take over the world, not the least of which is that it was written in the PL/I programming language, and the PL/I compiler was years late and barely worked at all when it finally arrived. In addition, MULTICS was enormously ambitious for its time, much like Charles Babbage's analytical engine in the nineteenth century.

To make a long story short, MULTICS introduced many seminal ideas into the computer literature, but turning it into a serious product and a major commercial success was a lot harder than anyone had expected. Bell Labs dropped out of the project, and General Electric quit the computer business altogether. However, M.I.T. persisted and eventually got MULTICS working. It was ultimately sold as a commercial product by the company (Honeywell) that bought GE's computer business and was installed by about 80 major companies and universities worldwide. While their numbers were small, MULTICS users were fiercely loyal. General Motors, Ford, and the U.S. National Security Agency, for example, shut down their MULTICS systems only in the late 1990s, 30 years after MULTICS was released, after years of trying to get Honeywell to update the hardware.

By the end of the 20th century, the concept of a computer utility had fizzled out, but it may well come back in the form of **cloud computing**, in which relatively small computers (including smartphones, tablets, and the like) are connected to servers in vast and distant data centers where all the computing is done, with the local computer just handling the user interface. The motivation here is that most people do not want to administrate an increasingly complex and finicky computer system and would prefer to have that work done by a team of professionals, for example, people working for the company running the data center. E-commerce is already evolving in this direction, with various companies running emails on multiprocessor servers to which simple client machines connect, very much in the spirit of the MULTICS design.

Despite its lack of commercial success, MULTICS had a huge influence on subsequent operating systems (especially UNIX and its derivatives, FreeBSD, Linux, iOS, and Android). It is described in several papers and a book (Corbató et al., 1972; Corbató and Vyssotsky, 1965; Daley and Dennis, 1968; Organick, 1972;

and Saltzer, 1974). It also has an active Website, located at *www.multicians.org*, with much information about the system, its designers, and its users.

Another major development during the third generation was the phenomenal growth of minicomputers, starting with the DEC PDP-1 in 1961. The PDP-1 had only 4K of 18-bit words, but at $120,000 per machine (less than 5% of the price of a 7094), it sold like hotcakes. For certain kinds of nonnumerical work, it was almost as fast as the 7094 and gave birth to a whole new industry. It was quickly followed by a series of other PDPs (unlike IBM's family, all incompatible) culminating in the PDP-11.

One of the computer scientists at Bell Labs who had worked on the MULTICS project, Ken Thompson, subsequently found a small PDP-7 minicomputer that no one was using and set out to write a stripped-down, one-user version of MULTICS. This work later developed into the **UNIX** operating system, which became popular in the academic world, with government agencies, and with many companies.

The history of UNIX has been told elsewhere (e.g., Salus, 1994). Part of that story will be given in Chap. 10. For now, suffice it to say that because the source code was widely available, various organizations developed their own (incompatible) versions, which led to chaos. Two major versions developed, **System V**, from AT&T, and **BSD** (**Berkeley Software Distribution**) from the University of California at Berkeley. These had minor variants as well. To make it possible to write programs that could run on any UNIX system, IEEE developed a standard for UNIX, called **POSIX**, that most versions of UNIX now support. POSIX defines a minimal system-call interface that conformant UNIX systems must support. In fact, some other operating systems now also support the POSIX interface.

As an aside, it is worth mentioning that in 1987, the author released a small clone of UNIX, called **MINIX,** for educational purposes. Functionally, MINIX is very similar to UNIX, including POSIX support. Since that time, the original version has evolved into MINIX 3, which is highly modular and focused on very high reliability. It has the ability to detect and replace faulty or even crashed modules (such as I/O device drivers) on the fly without a reboot and without disturbing running programs. Its focus is on providing very high dependability and availability. A book describing its internal operation and listing the source code in an appendix is also available (Tanenbaum and Woodhull, 2006). The MINIX 3 system is available for free (including all the source code) over the Internet at *www.minix3.org*.

The desire for a free production (as opposed to educational) version of MINIX led a Finnish student, Linus Torvalds, to write **Linux**. This system was directly inspired by and developed on MINIX and originally supported various MINIX features (e.g., the MINIX file system). It has since been extended in many ways by many people but still retains some underlying structure common to MINIX and to UNIX. Readers interested in a detailed history of Linux and the open source movement might want to read Glyn Moody's (2001) book. Most of what will be said about UNIX in this book thus applies to System V, MINIX, Linux, and other versions and clones of UNIX as well.

1.2.4 The Fourth Generation (1980–Present): Personal Computers

With the development of **LSI** (**Large Scale Integration**) circuits—chips containing thousands of transistors on a square centimeter of silicon—the age of the personal computer dawned. In terms of architecture, personal computers (initially called **microcomputers**) were not all that different from minicomputers of the PDP-11 class, but in terms of price they certainly were different. Where the minicomputer made it possible for a department in a company or university to have its own computer, the microprocessor chip made it possible for a single individual to have his or her own personal computer.

In 1974, when Intel came out with the 8080, the first general-purpose 8-bit CPU, it wanted an operating system for the 8080, in part to be able to test it. Intel asked one of its consultants, Gary Kildall, to write one. Kildall and a friend first built a controller for the newly released Shugart Associates 8-inch floppy disk and hooked the floppy disk up to the 8080, thus producing the first microcomputer with a disk. Kildall then wrote a disk-based operating system called **CP/M** (**Control Program for Microcomputers**) for it. Since Intel did not think that disk-based microcomputers had much of a future, when Kildall asked for the rights to CP/M, Intel granted his request. Kildall then formed a company, Digital Research, to further develop and sell CP/M.

In 1977, Digital Research rewrote CP/M to make it suitable for running on the many microcomputers using the 8080, Zilog Z80, and other CPU chips. Many application programs were written to run on CP/M, allowing it to completely dominate the world of microcomputing for about 5 years.

In the early 1980s, IBM designed the IBM PC and looked around for software to run on it. People from IBM contacted Bill Gates to license his BASIC interpreter. They also asked him if he knew of an operating system to run on the PC. Gates suggested that IBM contact Digital Research, then the world's dominant operating systems company. Making what was surely the worst business decision in recorded history, Kildall refused to meet with IBM, sending a subordinate instead. To make matters even worse, his lawyer even refused to sign IBM's nondisclosure agreement covering the not-yet-announced PC. Consequently, IBM went back to Gates asking if he could provide them with an operating system.

When IBM came back, Gates realized that a local computer manufacturer, Seattle Computer Products, had a suitable operating system, **DOS** (**Disk Operating System**). He approached them and asked to buy it (allegedly for $75,000), which they readily accepted. Gates then offered IBM a DOS/BASIC package, which IBM accepted. IBM wanted certain modifications, so Gates hired the person who wrote DOS, Tim Paterson, as an employee of Gates' fledgling company, Microsoft, to make them. The revised system was renamed **MS-DOS** (**MicroSoft Disk Operating System**) and quickly came to dominate the IBM PC market. A key factor here was Gates' (in retrospect, extremely wise) decision to sell MS-DOS to computer companies for bundling with their hardware, compared to Kildall's attempt to sell CP/M to end users one at a time (at least initially). After all this transpired, Kildall died suddenly and unexpectedly from causes that have not been fully disclosed.

By the time the successor to the IBM PC, the IBM PC/AT, came out in 1983 with the Intel 80286 CPU, MS-DOS was firmly entrenched and CP/M was on its last legs. MS-DOS was later widely used on the 80386 and 80486. Although the initial version of MS-DOS was fairly primitive, subsequent versions included more advanced features, including many taken from UNIX. (Microsoft was well aware of UNIX, even selling a microcomputer version of it called XENIX during the company's early years.)

CP/M, MS-DOS, and other operating systems for early microcomputers were all based on users typing in commands from the keyboard. That eventually changed due to research done by Doug Engelbart at Stanford Research Institute in the 1960s. Engelbart invented the Graphical User Interface, complete with windows, icons, menus, and mouse. These ideas were adopted by researchers at Xerox PARC and incorporated into machines they built.

One day, Steve Jobs, who co-invented the Apple computer in his garage, visited PARC, saw a GUI, and instantly realized its potential value, something Xerox management famously did not. This strategic blunder of gargantuan proportions led to a book entitled *Fumbling the Future* (Smith and Alexander, 1988). Jobs then embarked on building an Apple with a GUI. This project led to the Lisa, which was too expensive and failed commercially. Jobs' second attempt, the Apple Macintosh, was a huge success, not only because it was much cheaper than the Lisa, but also because it was **user friendly**, meaning that it was intended for users who not only knew nothing about computers but furthermore had absolutely no intention whatsoever of learning. In the creative world of graphic design, professional digital photography, and professional digital video production, Macintoshes are very widely used and their users are very enthusiastic about them. In 1999, Apple adopted a kernel derived from Carnegie Mellon University's Mach microkernel which was originally developed to replace the kernel of BSD UNIX. Thus, **Mac OS X** is a UNIX-based operating system, albeit with a very distinctive interface.

When Microsoft decided to build a successor to MS-DOS, it was strongly influenced by the success of the Macintosh. It produced a GUI-based system called Windows, which originally ran on top of MS-DOS (i.e., it was more like a shell than a true operating system). For about 10 years, from 1985 to 1995, Windows was just a graphical environment on top of MS-DOS. However, starting in 1995 a freestanding version, Windows 95, was released that incorporated many operating system features into it, using the underlying MS-DOS system only for booting and running old MS-DOS programs. In 1998, a slightly modified version of this system, called Windows 98 was released. Nevertheless, both Windows 95 and Windows 98 still contained a large amount of 16-bit Intel assembly language.

Another Microsoft operating system, **Windows NT** (where the NT stands for **New Technology**), which was compatible with Windows 95 at a certain level, but a complete rewrite from scratch internally. It was a full 32-bit system. The lead designer for Windows NT was David Cutler, who was also one of the designers of the VAX VMS operating system, so some ideas from VMS are present in NT. In fact, so many ideas from VMS were present in it that the owner of VMS, DEC, sued Microsoft. The case was settled out of court for an amount of money requiring many digits to express. Microsoft expected that the first version of NT would kill off MS-DOS and all other versions of Windows since it was a vastly superior system, but it fizzled. Only with Windows NT 4.0 did it finally catch on in a big way, especially on corporate networks. Version 5 of Windows NT was renamed Windows 2000 in early 1999. It was intended to be the successor to both Windows 98 and Windows NT 4.0.

That did not quite work out either, so Microsoft came out with yet another version of Windows 98 called **Windows Me** (**Millennium Edition**). In 2001, a slightly upgraded version of Windows 2000, called Windows XP was released. That version had a much longer run (6 years), basically replacing all previous versions of Windows.

Still the spawning of versions continued unabated. After Windows 2000, Microsoft broke up the Windows family into a client and a server line. The client line was based on XP and its successors, while the server line included Windows Server 2003 and Windows 2008. A third line, for the embedded world, appeared a little later. All of these versions of Windows forked off their variations in the form of **service packs**. It was enough to drive some administrators (and writers of operating systems textbooks) balmy.

Then in January 2007, Microsoft finally released the successor to Windows XP, called Vista. It came with a new graphical interface, improved security, and many new or upgraded user programs. Microsoft hoped it would replace Windows XP completely, but it never did. Instead, it received much criticism and a bad press, mostly due to the high system requirements, restrictive licensing terms, and support for **Digital Rights Management**, techniques that made it harder for users to copy protected material.

With the arrival of Windows 7, a new and much less resource hungry version of the operating system, many people decided to skip Vista altogether. Windows 7 did not introduce too many new features, but it was relatively small and quite stable. In less than three weeks, Windows 7 had obtained more market share than Vista in seven months. In 2012, Microsoft launched its successor, Windows 8, an operating system with a completely new look and feel, geared for touch screens. The company hopes that the new design will become the dominant operating system on a much wider variety of devices: desktops, laptops, notebooks, tablets, phones, and home theater PCs. So far, however, the market penetration is slow compared to Windows 7.

The other major contender in the personal computer world is UNIX (and its various derivatives). UNIX is strongest on network and enterprise servers but is also often present on desktop computers, notebooks, tablets, and smartphones. On x86-based computers, Linux is becoming a popular alternative to Windows for students and increasingly many corporate users.

As an aside, throughout this book we will use the term **x86** to refer to all modern processors based on the family of instruction-set architectures that started with the 8086 in the 1970s. There are many such processors, manufactured by companies like AMD and Intel, and under the hood they often differ considerably: processors may be 32 bits or 64 bits with few or many cores and pipelines that may be deep or shallow, and so on. Nevertheless, to the programmer, they all look quite similar and they can all still run 8086 code that was written 35 years ago. Where the difference is important, we will refer to explicit models instead—and use **x86-32** and **x86-64** to indicate 32-bit and 64-bit variants.

FreeBSD is also a popular UNIX derivative, originating from the BSD project at Berkeley. All modern Macintosh computers run a modified version of FreeBSD (OS X). UNIX is also standard on workstations powered by high-performance RISC chips. Its derivatives are widely used on mobile devices, such as those running iOS 7 or Android.

Many UNIX users, especially experienced programmers, prefer a command-based interface to a GUI, so nearly all UNIX systems support a windowing system called the **X Window System** (also known as **X11**) produced at M.I.T. This system handles the basic window management, allowing users to create, delete, move, and resize windows using a mouse. Often a complete GUI, such as **Gnome** or **KDE**, is available to run on top of X11, giving UNIX a look and feel something like the Macintosh or Microsoft Windows, for those UNIX users who want such a thing.

An interesting development that began taking place during the mid-1980s is the growth of networks of personal computers running **network operating systems** and **distributed operating systems** (Tanenbaum and Van Steen, 2007). In a network operating system, the users are aware of the existence of multiple computers and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and has its own local user (or users).

Network operating systems are not fundamentally different from single-processor operating systems. They obviously need a network interface controller and some low-level software to drive it, as well as programs to achieve remote login and remote file access, but these additions do not change the essential structure of the operating system.

A distributed operating system, in contrast, is one that appears to its users as a traditional uniprocessor system, even though it is actually composed of multiple processors. The users should not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system.

True distributed operating systems require more than just adding a little code to a uniprocessor operating system, because distributed and centralized systems differ in certain critical ways. Distributed systems, for example, often allow applications to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimize the amount of parallelism.

Communication delays within the network often mean that these (and other) algorithms must run with incomplete, outdated, or even incorrect information. This situation differs radically from that in a single-processor system in which the operating system has complete information about the system state.

1.2.5 The Fifth Generation (1990–Present): Mobile Computers

Ever since detective Dick Tracy started talking to his "two-way radio wrist watch" in the 1940s comic strip, people have craved a communication device they could carry around wherever they went. The first real mobile phone appeared in 1946 and weighed some 40 kilos. You could take it wherever you went as long as you had a car in which to carry it.

The first true handheld phone appeared in the 1970s and, at roughly one kilogram, was positively featherweight. It was affectionately known as "the brick." Pretty soon everybody wanted one. Today, mobile phone penetration is close to 90% of the global population. We can make calls not just with our portable phones and wrist watches, but soon with eyeglasses and other wearable items. Moreover, the phone part is no longer that interesting. We receive email, surf the Web, text our friends, play games, navigate around heavy traffic—and do not even think twice about it.

While the idea of combining telephony and computing in a phone-like device has been around since the 1970s also, the first real smartphone did not appear until the mid-1990s when Nokia released the N9000, which literally combined two, mostly separate devices: a phone and a **PDA** (Personal Digital Assistant). In 1997, Ericsson coined the term *smartphone* for its GS88 "Penelope."

Now that smartphones have become ubiquitous, the competition between the various operating systems is fierce and the outcome is even less clear than in the PC world. At the time of writing, Google's Android is the dominant operating system with Apple's iOS a clear second, but this was not always the case and all may be different again in just a few years. If anything is clear in the world of smartphones, it is that it is not easy to stay king of the mountain for long.

After all, most smartphones in the first decade after their inception were running **Symbian** OS. It was the operating system of choice for popular brands like Samsung, Sony Ericsson, Motorola, and especially Nokia. However, other operating systems like **RIM's** Blackberry OS (introduced for smartphones in 2002) and Apple's iOS (released for the first **iPhone** in 2007) started eating into Symbian's market share. Many expected that RIM would dominate the business market, while iOS would be the king of the consumer devices. Symbian's market share plummeted. In 2011, Nokia ditched Symbian and announced it would focus on Windows Phone as its primary platform. For some time, Apple and RIM were the toast of the town (although not nearly as dominant as Symbian had been), but it did not take very long for Android, a Linux-based operating system released by Google in 2008, to overtake all its rivals.

For phone manufacturers, Android had the advantage that it was open source and available under a permissive license. As a result, they could tinker with it and adapt it to their own hardware with ease. Also, it has a huge community of developers writing apps, mostly in the familiar Java programming language. Even so, the past years have shown that the dominance may not last, and Android's competitors are eager to claw back some of its market share. We will look at Android in detail in Sec. 10.8.

1.3 COMPUTER HARDWARE REVIEW

An operating system is intimately tied to the hardware of the computer it runs on. It extends the computer's instruction set and manages its resources. To work, it must know a great deal about the hardware, at least about how the hardware appears to the programmer. For this reason, let us briefly review computer hardware as found in modern personal computers. After that, we can start getting into the details of what operating systems do and how they work.

Conceptually, a simple personal computer can be abstracted to a model resembling that of Fig. 1-6. The CPU, memory, and I/O devices are all connected by a system bus and communicate with one another over it. Modern personal computers have a more complicated structure, involving multiple buses, which we will look at later. For the time being, this model will be sufficient. In the following sections, we will briefly review these components and examine some of the hardware issues that are of concern to operating system designers. Needless to say, this will be a very compact summary. Many books have been written on the subject of computer hardware and computer organization. Two well-known ones are by Tanenbaum and Austin (2012) and Patterson and Hennessy (2013).

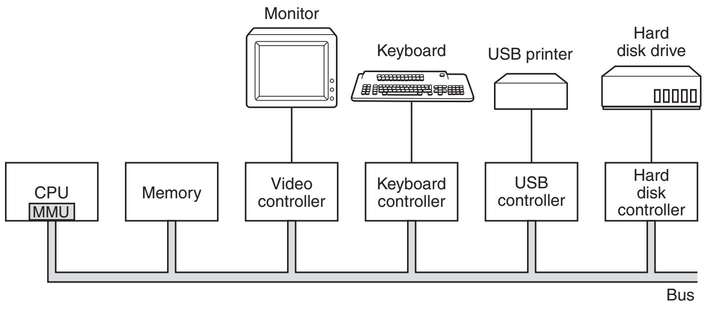


Figure 1-6. Some of the components of a simple personal computer.

1.3.1 Processors

The "brain" of the computer is the CPU. It fetches instructions from memory and executes them. The basic cycle of every CPU is to fetch the first instruction from memory, decode it to determine its type and operands, execute it, and then fetch, decode, and execute subsequent instructions. The cycle is repeated until the program finishes. In this way, programs are carried out.

Each CPU has a specific set of instructions that it can execute. Thus an x86 processor cannot execute ARM programs and an ARM processor cannot execute x86 programs. Because accessing memory to get an instruction or data word takes much longer than executing an instruction, all CPUs contain some registers inside to hold key variables and temporary results. Thus the instruction set generally contains instructions to load a word from memory into a register, and store a word from a register into memory. Other instructions combine two operands from registers, memory, or both into a result, such as adding two words and storing the result in a register or in memory.

In addition to the general registers used to hold variables and temporary results, most computers have several special registers that are visible to the programmer. One of these is the **program counter**, which contains the memory address of the next instruction to be fetched. After that instruction has been fetched, the program counter is updated to point to its successor.

Another register is the **stack pointer**, which points to the top of the current stack in memory. The stack contains one frame for each procedure that has been entered but not yet exited. A procedure's stack frame holds those input parameters, local variables, and temporary variables that are not kept in registers.

Yet another register is the **PSW** (**Program Status Word**). This register contains the condition code bits, which are set by comparison instructions, the CPU priority, the mode (user or kernel), and various other control bits. User programs may normally read the entire PSW but typically may write only some of its fields. The PSW plays an important role in system calls and I/O.

The operating system must be fully aware of all the registers. When time multiplexing the CPU, the operating system will often stop the running program to (re)start another one. Every time it stops a running program, the operating system must save all the registers so they can be restored when the program runs later.

To improve performance, CPU designers have long abandoned the simple model of fetching, decoding, and executing one instruction at a time. Many modern CPUs have facilities for executing more than one instruction at the same time. For example, a CPU might have separate fetch, decode, and execute units, so that while it is executing instruction n, it could also be decoding instruction $n + 1$ and fetching instruction $n + 2$. Such an organization is called a **pipeline** and is illustrated in Fig. 1-7(a) for a pipeline with three stages. Longer pipelines are common. In most pipeline designs, once an instruction has been fetched into the pipeline, it must be executed, even if the preceding instruction was a conditional branch that was taken.

Pipelines cause compiler writers and operating system writers great headaches because they expose the complexities of the underlying machine to them and they have to deal with them.

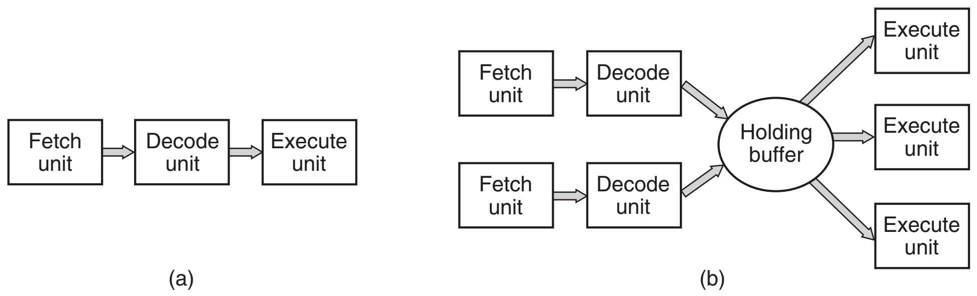


Figure 1-7. (a) A three-stage pipeline. (b) A superscalar CPU.

Even more advanced than a pipeline design is a **superscalar** CPU, shown in Fig. 1-7(b). In this design, multiple execution units are present, for example, one for integer arithmetic, one for floating-point arithmetic, and one for Boolean operations. Two or more instructions are fetched at once, decoded, and dumped into a holding buffer until they can be executed. As soon as an execution unit becomes available, it looks in the holding buffer to see if there is an instruction it can handle, and if so, it removes the instruction from the buffer and executes it. An implication of this design is that program instructions are often executed out of order. For the most part, it is up to the hardware to make sure the result produced is the same one a sequential implementation would have produced, but an annoying amount of the complexity is foisted onto the operating system, as we shall see.

Most CPUs, except very simple ones used in embedded systems, have two modes, kernel mode and user mode, as mentioned earlier. Usually, a bit in the PSW controls the mode. When running in kernel mode, the CPU can execute every instruction in its instruction set and use every feature of the hardware. On desktop and server machines, the operating system normally runs in kernel mode, giving it access to the complete hardware. On most embedded systems, a small piece runs in kernel mode, with the rest of the operating system running in user mode.

User programs always run in user mode, which permits only a subset of the instructions to be executed and a subset of the features to be accessed. Generally, all instructions involving I/O and memory protection are disallowed in user mode. Setting the PSW mode bit to enter kernel mode is also forbidden, of course.

To obtain services from the operating system, a user program must make a **system call**, which traps into the kernel and invokes the operating system. The TRAP instruction switches from user mode to kernel mode and starts the operating system. When the work has been completed, control is returned to the user program at the instruction following the system call. We will explain the details of the system call mechanism later in this chapter. For the time being, think of it as a special kind of procedure call that has the additional property of switching from user mode to kernel mode. As a note on typography, we will use the lower-case Helvetica font to indicate system calls in running text, like this: read.

It is worth noting that computers have traps other than the instruction for executing a system call. Most of the other traps are caused by the hardware to warn of an exceptional situation such as an attempt to divide by 0 or a floating-point underflow. In all cases the operating system gets control and must decide what to do. Sometimes the program must be terminated with an error. Other times the error can be ignored (an underflowed number can be set to 0). Finally, when the program has announced in advance that it wants to handle certain kinds of conditions, control can be passed back to the program to let it deal with the problem.

Multithreaded and Multicore Chips

Moore's law states that the number of transistors on a chip doubles every 18 months. This "law" is not some kind of law of physics, like conservation of momentum, but is an observation by Intel cofounder Gordon Moore of how fast process engineers at the semiconductor companies are able to shrink their transistors. Moore's law has held for over three decades now and is expected to hold for at least one more. After that, the number of atoms per transistor will become too small and quantum mechanics will start to play a big role, preventing further shrinkage of transistor sizes.

The abundance of transistors is leading to a problem: what to do with all of them? We saw one approach above: superscalar architectures, with multiple functional units. But as the number of transistors increases, even more is possible. One obvious thing to do is put bigger caches on the CPU chip. That is definitely happening, but eventually the point of diminishing returns will be reached.

The obvious next step is to replicate not only the functional units, but also some of the control logic. The Intel Pentium 4 introduced this property, called **multithreading** or **hyperthreading** (Intel's name for it), to the x86 processor, and several other CPU chips also have it—including the SPARC, the Power5, the Intel Xeon, and the Intel Core family. To a first approximation, what it does is allow the CPU to hold the state of two different threads and then switch back and forth on a nanosecond time scale. (A thread is a kind of lightweight process, which, in turn, is a running program; we will get into the details in Chap. 2.) For example, if one of the processes needs to read a word from memory (which takes many clock cycles), a multithreaded CPU can just switch to another thread. Multithreading does not offer true parallelism. Only one process at a time is running, but thread-switching time is reduced to the order of a nanosecond.

Multithreading has implications for the operating system because each thread appears to the operating system as a separate CPU. Consider a system with two actual CPUs, each with two threads. The operating system will see this as four CPUs. If there is only enough work to keep two CPUs busy at a certain point in time, it may inadvertently schedule two threads on the same CPU, with the other CPU completely idle. This choice is far less efficient than using one thread on each CPU.

Beyond multithreading, many CPU chips now have four, eight, or more complete processors or **cores** on them. The multicore chips of Fig. 1-8 effectively carry four minichips on them, each with its own independent CPU. (The caches will be explained below.) Some processors, like Intel Xeon Phi and the Tilera TilePro, already sport more than 60 cores on a single chip. Making use of such a multicore chip will definitely require a multiprocessor operating system.

Incidentally, in terms of sheer numbers, nothing beats a modern **GPU** (**Graphics Processing Unit**). A GPU is a processor with, literally, thousands of tiny cores. They are very good for many small computations done in parallel, like rendering polygons in graphics applications. They are not so good at serial tasks. They are also hard to program. While GPUs can be useful for operating systems (e.g., encryption or processing of network traffic), it is not likely that much of the operating system itself will run on the GPUs.

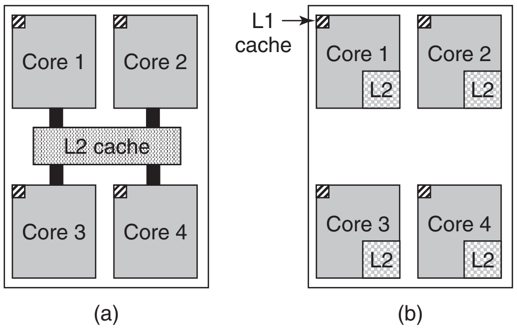


Figure 1-8. (a) A quad-core chip with a shared L2 cache. (b) A quad-core chip with separate L2 caches.

1.3.2 Memory

The second major component in any computer is the memory. Ideally, a memory should be extremely fast (faster than executing an instruction so that the CPU is not held up by the memory), abundantly large, and dirt cheap. No current technology satisfies all of these goals, so a different approach is taken. The memory system is constructed as a hierarchy of layers, as shown in Fig. 1-9. The top layers have higher speed, smaller capacity, and greater cost per bit than the lower ones, often by factors of a billion or more.

The top layer consists of the registers internal to the CPU. They are made of the same material as the CPU and are thus just as fast as the CPU. Consequently, there is no delay in accessing them. The storage capacity available in them is

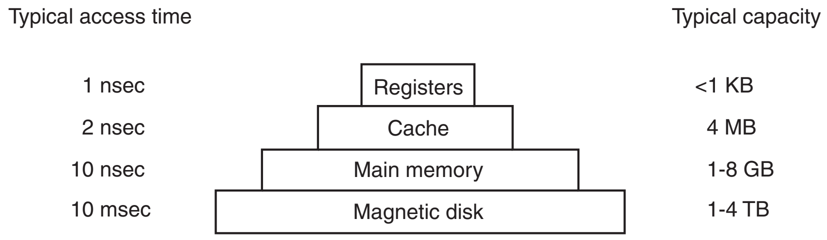


Figure 1-9. A typical memory hierarchy. The numbers are very rough approximations.

typically 32 × 32 bits on a 32-bit CPU and 64 × 64 bits on a 64-bit CPU. Less than 1 KB in both cases. Programs must manage the registers (i.e., decide what to keep in them) themselves, in software.

Next comes the cache memory, which is mostly controlled by the hardware. Main memory is divided up into **cache lines**, typically 64 bytes, with addresses 0 to 63 in cache line 0, 64 to 127 in cache line 1, and so on. The most heavily used cache lines are kept in a high-speed cache located inside or very close to the CPU. When the program needs to read a memory word, the cache hardware checks to see if the line needed is in the cache. If it is, called a **cache hit**, the request is satisfied from the cache and no memory request is sent over the bus to the main memory. Cache hits normally take about two clock cycles. Cache misses have to go to memory, with a substantial time penalty. Cache memory is limited in size due to its high cost. Some machines have two or even three levels of cache, each one slower and bigger than the one before it.

Caching plays a major role in many areas of computer science, not just caching lines of RAM. Whenever a resource can be divided into pieces, some of which are used much more heavily than others, caching is often used to improve performance. Operating systems use it all the time. For example, most operating systems keep (pieces of) heavily used files in main memory to avoid having to fetch them from the disk repeatedly. Similarly, the results of converting long path names like

/home/ast/projects/minix3/src/kernel/clock.c

into the disk address where the file is located can be cached to avoid repeated lookups. Finally, when the address of a Web page (URL) is converted to a network address (IP address), the result can be cached for future use. Many other uses exist.

In any caching system, several questions come up fairly soon, including:

1. When to put a new item into the cache.
2. Which cache line to put the new item in.
3. Which item to remove from the cache when a slot is needed.
4. Where to put a newly evicted item in the larger memory.

Not every question is relevant to every caching situation. For caching lines of main memory in the CPU cache, a new item will generally be entered on every cache miss. The cache line to use is generally computed by using some of the high-order bits of the memory address referenced. For example, with 4096 cache lines of 64 bytes and 32 bit addresses, bits 6 through 17 might be used to specify the cache line, with bits 0 to 5 the byte within the cache line. In this case, the item to remove is the same one as the new data goes into, but in other systems it might not be. Finally, when a cache line is rewritten to main memory (if it has been modified since it was cached), the place in memory to rewrite it to is uniquely determined by the address in question.

Caches are such a good idea that modern CPUs have two of them. The first level or **L1 cache** is always inside the CPU and usually feeds decoded instructions into the CPU's execution engine. Most chips have a second L1 cache for very heavily used data words. The L1 caches are typically 16 KB each. In addition, there is often a second cache, called the **L2 cache**, that holds several megabytes of recently used memory words. The difference between the L1 and L2 caches lies in the timing. Access to the L1 cache is done without any delay, whereas access to the L2 cache involves a delay of one or two clock cycles.

On multicore chips, the designers have to decide where to place the caches. In Fig. 1-8(a), a single L2 cache is shared by all the cores. This approach is used in Intel multicore chips. In contrast, in Fig. 1-8(b), each core has its own L2 cache. This approach is used by AMD. Each strategy has its pros and cons. For example, the Intel shared L2 cache requires a more complicated cache controller but the AMD way makes keeping the L2 caches consistent more difficult.

Main memory comes next in the hierarchy of Fig. 1-9. This is the workhorse of the memory system. Main memory is usually called **RAM** (**Random Access Memory**). Old-timers sometimes call it **core memory**, because computers in the 1950s and 1960s used tiny magnetizable ferrite cores for main memory. They have been gone for decades but the name persists. Currently, memories are hundreds of megabytes to several gigabytes and growing rapidly. All CPU requests that cannot be satisfied out of the cache go to main memory.

In addition to the main memory, many computers have a small amount of nonvolatile random-access memory. Unlike RAM, nonvolatile memory does not lose its contents when the power is switched off. **ROM** (**Read Only Memory**) is programmed at the factory and cannot be changed afterward. It is fast and inexpensive. On some computers, the bootstrap loader used to start the computer is contained in ROM. Also, some I/O cards come with ROM for handling low-level device control.

EEPROM (**Electrically Erasable PROM**) and **flash memory** are also nonvolatile, but in contrast to ROM can be erased and rewritten. However, writing them takes orders of magnitude more time than writing RAM, so they are used in the same way ROM is, only with the additional feature that it is now possible to correct bugs in programs they hold by rewriting them in the field.

Flash memory is also commonly used as the storage medium in portable electronic devices. It serves as film in digital cameras and as the disk in portable music players, to name just two uses. Flash memory is intermediate in speed between RAM and disk. Also, unlike disk memory, if it is erased too many times, it wears out.

Yet another kind of memory is CMOS, which is volatile. Many computers use CMOS memory to hold the current time and date. The CMOS memory and the clock circuit that increments the time in it are powered by a small battery, so the time is correctly updated, even when the computer is unplugged. The CMOS memory can also hold the configuration parameters, such as which disk to boot from. CMOS is used because it draws so little power that the original factory-installed battery often lasts for several years. However, when it begins to fail, the computer can appear to have Alzheimer's disease, forgetting things that it has known for years, like which hard disk to boot from.

1.3.3 Disks

Next in the hierarchy is magnetic disk (hard disk). Disk storage is two orders of magnitude cheaper than RAM per bit and often two orders of magnitude larger as well. The only problem is that the time to randomly access data on it is close to three orders of magnitude slower. The reason is that a disk is a mechanical device, as shown in Fig. 1-10.

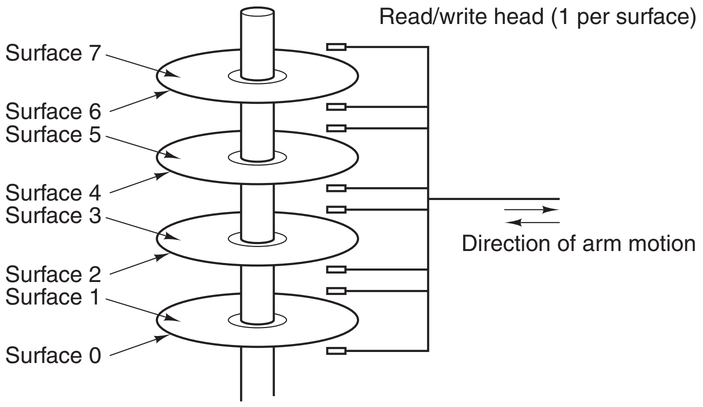


Figure 1-10. Structure of a disk drive.

A disk consists of one or more metal platters that rotate at 5400, 7200, 10,800 RPM or more. A mechanical arm pivots over the platters from the corner, similar to the pickup arm on an old 33-RPM phonograph for playing vinyl records.

Information is written onto the disk in a series of concentric circles. At any given arm position, each of the heads can read an annular region called a **track**. Together, all the tracks for a given arm position form a **cylinder**.

Each track is divided into some number of sectors, typically 512 bytes per sector. On modern disks, the outer cylinders contain more sectors than the inner ones. Moving the arm from one cylinder to the next takes about 1 msec. Moving it to a random cylinder typically takes 5 to 10 msec, depending on the drive. Once the arm is on the correct track, the drive must wait for the needed sector to rotate under the head, an additional delay of 5 msec to 10 msec, depending on the drive's RPM. Once the sector is under the head, reading or writing occurs at a rate of 50 MB/sec on low-end disks to 160 MB/sec on faster ones.

Sometimes you will hear people talk about disks that are really not disks at all, like **SSDs**, (**Solid State Disks**). SSDs do not have moving parts, do not contain platters in the shape of disks, and store data in (Flash) memory. The only ways in which they resemble disks is that they also store a lot of data which is not lost when the power is off.

Many computers support a scheme known as **virtual memory**, which we will discuss at some length in Chap. 3. This scheme makes it possible to run programs larger than physical memory by placing them on the disk and using main memory as a kind of cache for the most heavily executed parts. This scheme requires remapping memory addresses on the fly to convert the address the program generated to the physical address in RAM where the word is located. This mapping is done by a part of the CPU called the **MMU** (**Memory Management Unit**), as shown in Fig. 1-6.

The presence of caching and the MMU can have a major impact on performance. In a multiprogramming system, when switching from one program to another, sometimes called a **context switch**, it may be necessary to flush all modified blocks from the cache and change the mapping registers in the MMU. Both of these are expensive operations, and programmers try hard to avoid them. We will see some of the implications of their tactics later.

1.3.4 I/O Devices

The CPU and memory are not the only resources that the operating system must manage. I/O devices also interact heavily with the operating system. As we saw in Fig. 1-6, I/O devices generally consist of two parts: a controller and the device itself. The controller is a chip or a set of chips that physically controls the device. It accepts commands from the operating system, for example, to read data from the device, and carries them out.

In many cases, the actual control of the device is complicated and detailed, so it is the job of the controller to present a simpler (but still very complex) interface to the operating system. For example, a disk controller might accept a command to read sector 11,206 from disk 2. The controller then has to convert this linear sector number to a cylinder, sector, and head. This conversion may be complicated by the fact that outer cylinders have more sectors than inner ones and that some bad sectors have been remapped onto other ones. Then the controller has to determine which cylinder the disk arm is on and give it a command to move in or out the requisite number of cylinders. It has to wait until the proper sector has rotated under the head and then start reading and storing the bits as they come off the drive, removing the preamble and computing the checksum. Finally, it has to assemble the incoming bits into words and store them in memory. To do all this work, controllers often contain small embedded computers that are programmed to do their work.

The other piece is the actual device itself. Devices have fairly simple interfaces, both because they cannot do much and to make them standard. The latter is needed so that any SATA disk controller can handle any SATA disk, for example. **SATA** stands for **Serial ATA** and **ATA** in turn stands for **AT Attachment**. In case you are curious what AT stands for, this was IBM's second generation "Personal Computer Advanced Technology" built around the then-extremely-potent 6-MHz 80286 processor that the company introduced in 1984. What we learn from this is that the computer industry has a habit of continuously enhancing existing acronyms with new prefixes and suffixes. We also learned that an adjective like "advanced" should be used with great care, or you will look silly thirty years down the line.

SATA is currently the standard type of disk on many computers. Since the actual device interface is hidden behind the controller, all that the operating system sees is the interface to the controller, which may be quite different from the interface to the device.

Because each type of controller is different, different software is needed to control each one. The software that talks to a controller, giving it commands and accepting responses, is called a **device driver**. Each controller manufacturer has to supply a driver for each operating system it supports. Thus a scanner may come with drivers for OS X, Windows 7, Windows 8, and Linux, for example.

To be used, the driver has to be put into the operating system so it can run in kernel mode. Drivers can actually run outside the kernel, and operating systems like Linux and Windows nowadays do offer some support for doing so. The vast majority of the drivers still run below the kernel boundary. Only very few current systems, such as MINIX 3, run all drivers in user space. Drivers in user space must be allowed to access the device in a controlled way, which is not straightforward.

There are three ways the driver can be put into the kernel. The first way is to relink the kernel with the new driver and then reboot the system. Many older UNIX systems work like this. The second way is to make an entry in an operating system file telling it that it needs the driver and then reboot the system. At boot time, the operating system goes and finds the drivers it needs and loads them. Windows works this way. The third way is for the operating system to be able to accept new drivers while running and install them on the fly without the need to reboot. This way used to be rare but is becoming much more common now. Hot-pluggable devices, such as USB and IEEE 1394 devices (discussed below), always need dynamically loaded drivers.

Every controller has a small number of registers that are used to communicate with it. For example, a minimal disk controller might have registers for specifying the disk address, memory address, sector count, and direction (read or write). To activate the controller, the driver gets a command from the operating system, then translates it into the appropriate values to write into the device registers. The collection of all the device registers forms the **I/O port space**, a subject we will come back to in Chap. 5.

On some computers, the device registers are mapped into the operating system's address space (the addresses it can use), so they can be read and written like ordinary memory words. On such computers, no special I/O instructions are required and user programs can be kept away from the hardware by not putting these memory addresses within their reach (e.g., by using base and limit registers). On other computers, the device registers are put in a special I/O port space, with each register having a port address. On these machines, special IN and OUT instructions are available in kernel mode to allow drivers to read and write the registers. The former scheme eliminates the need for special I/O instructions but uses up some of the address space. The latter uses no address space but requires special instructions. Both systems are widely used.

Input and output can be done in three different ways. In the simplest method, a user program issues a system call, which the kernel then translates into a procedure call to the appropriate driver. The driver then starts the I/O and sits in a tight loop continuously polling the device to see if it is done (usually there is some bit that indicates that the device is still busy). When the I/O has completed, the driver puts the data (if any) where they are needed and returns. The operating system then returns control to the caller. This method is called **busy waiting** and has the disadvantage of tying up the CPU polling the device until it is finished.

The second method is for the driver to start the device and ask it to give an interrupt when it is finished. At that point the driver returns. The operating system then blocks the caller if need be and looks for other work to do. When the controller detects the end of the transfer, it generates an **interrupt** to signal completion.

Interrupts are very important in operating systems, so let us examine the idea more closely. In Fig. 1-11(a) we see a three-step process for I/O. In step 1, the driver tells the controller what to do by writing into its device registers. The controller then starts the device. When the controller has finished reading or writing the number of bytes it has been told to transfer, it signals the interrupt controller chip using certain bus lines in step 2. If the interrupt controller is ready to accept the interrupt (which it may not be if it is busy handling a higher-priority one), it asserts a pin on the CPU chip telling it, in step 3. In step 4, the interrupt controller puts the number of the device on the bus so the CPU can read it and know which device has just finished (many devices may be running at the same time).

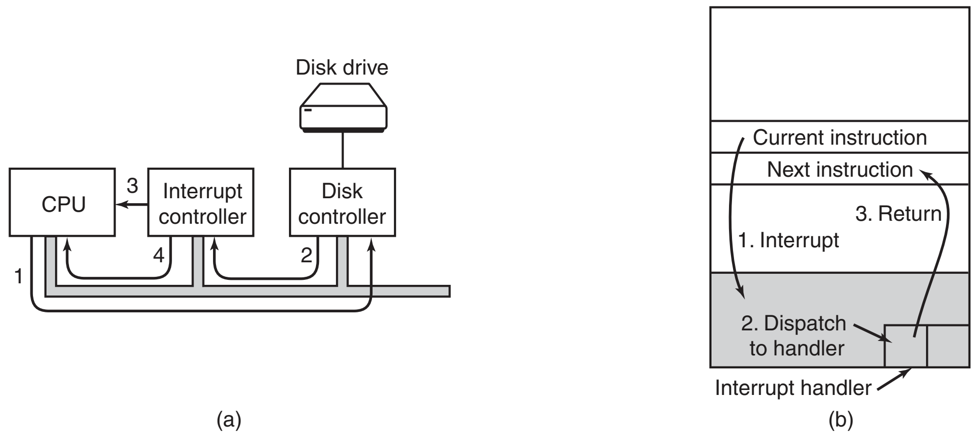


Figure 1-11. (a) The steps in starting an I/O device and getting an interrupt. (b) Interrupt processing involves taking the interrupt, running the interrupt handler, and returning to the user program.

Once the CPU has decided to take the interrupt, the program counter and PSW are typically then pushed onto the current stack and the CPU switched into kernel mode. The device number may be used as an index into part of memory to find the address of the interrupt handler for this device. This part of memory is called the **interrupt vector**. Once the interrupt handler (part of the driver for the interrupting device) has started, it removes the stacked program counter and PSW and saves them, then queries the device to learn its status. When the handler is all finished, it returns to the previously running user program to the first instruction that was not yet executed. These steps are shown in Fig. 1-11(b).

The third method for doing I/O makes use of special hardware: a **DMA** (**Direct Memory Access**) chip that can control the flow of bits between memory and some controller without constant CPU intervention. The CPU sets up the DMA chip, telling it how many bytes to transfer, the device and memory addresses involved, and the direction, and lets it go. When the DMA chip is done, it causes an interrupt, which is handled as described above. DMA and I/O hardware in general will be discussed in more detail in Chap. 5.

Interrupts can (and often do) happen at highly inconvenient moments, for example, while another interrupt handler is running. For this reason, the CPU has a way to disable interrupts and then reenable them later. While interrupts are disabled, any devices that finish continue to assert their interrupt signals, but the CPU is not interrupted until interrupts are enabled again. If multiple devices finish while interrupts are disabled, the interrupt controller decides which one to let through first, usually based on static priorities assigned to each device. The highest-priority device wins and gets to be serviced first. The others must wait.

1.3.5 Buses

The organization of Fig. 1-6 was used on minicomputers for years and also on the original IBM PC. However, as processors and memories got faster, the ability of a single bus (and certainly the IBM PC bus) to handle all the traffic was strained to the breaking point. Something had to give. As a result, additional buses were added, both for faster I/O devices and for CPU-to-memory traffic. As a consequence of this evolution, a large x86 system currently looks something like Fig. 1-12.

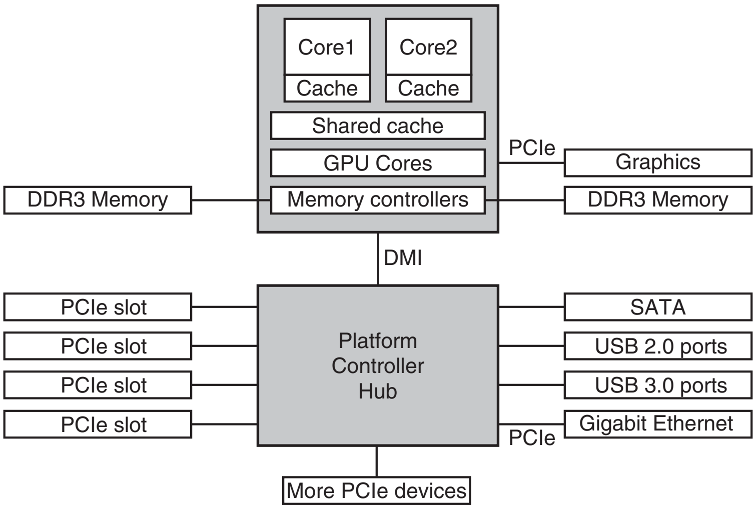


Figure 1-12. The structure of a large x86 system.

This system has many buses (e.g., cache, memory, PCIe, PCI, USB, SATA, and DMI), each with a different transfer rate and function. The operating system must be aware of all of them for configuration and management. The main bus is the **PCIe** (**Peripheral Component Interconnect Express**) bus.

The PCIe bus was invented by Intel as a successor to the older **PCI** bus, which in turn was a replacement for the original **ISA** (**Industry Standard Architecture**) bus. Capable of transferring tens of gigabits per second, PCIe is much faster than its predecessors. It is also very different in nature. Up to its creation in 2004, most buses were parallel and shared. A **shared bus architecture** means that multiple devices use the same wires to transfer data. Thus, when multiple devices have data to send, you need an arbiter to determine who can use the bus. In contrast, PCIe makes use of dedicated, point-to-point connections. A **parallel bus architecture** as used in traditional PCI means that you send each word of data over multiple wires. For instance, in regular PCI buses, a single 32-bit number is sent over 32 parallel wires. In contrast to this, PCIe uses a **serial bus architecture** and sends all bits in a message through a single connection, known as a lane, much like a network packet. This is much simpler, because you do not have to ensure that all 32 bits arrive at the destination at exactly the same time. Parallelism is still used, because you can have multiple lanes in parallel. For instance, we may use 32 lanes to carry 32 messages in parallel. As the speed of peripheral devices like network cards and graphics adapters increases rapidly, the PCIe standard is upgraded every 3–5 years. For instance, 16 lanes of PCIe 2.0 offer 64 gigabits per second. Upgrading to PCIe 3.0 will give you twice that speed and PCIe 4.0 will double that again.

Meanwhile, we still have many legacy devices for the older PCI standard. As we see in Fig. 1-12, these devices are hooked up to a separate hub processor. In the future, when we consider PCI no longer merely *old*, but *ancient*, it is possible that all PCI devices will attach to yet another hub that in turn connects them to the main hub, creating a tree of buses.

In this configuration, the CPU talks to memory over a fast DDR3 bus, to an external graphics device over PCIe and to all other devices via a hub over a **DMI** (**Direct Media Interface**) bus. The hub in turn connects all the other devices, using the Universal Serial Bus to talk to USB devices, the SATA bus to interact with hard disks and DVD drives, and PCIe to transfer Ethernet frames. We have already mentioned the older PCI devices that use a traditional PCI bus.

Moreover, each of the cores has a dedicated cache and a much larger cache that is shared between them. Each of these caches introduces another bus.

The **USB** (**Universal Serial Bus**) was invented to attach all the slow I/O devices, such as the keyboard and mouse, to the computer. However, calling a modern USB 3.0 device humming along at 5 Gbps "slow" may not come naturally for the generation that grew up with 8-Mbps ISA as the main bus in the first IBM PCs. USB uses a small connector with four to eleven wires (depending on the version), some of which supply electrical power to the USB devices or connect to ground. USB is a centralized bus in which a root device polls all the I/O devices every 1 msec to see if they have any traffic. USB 1.0 could handle an aggregate load of 12 Mbps, USB 2.0 increased the speed to 480 Mbps, and USB 3.0 tops at no less than 5 Gbps. Any USB device can be connected to a computer and it will function immediately, without requiring a reboot, something pre-USB devices required, much to the consternation of a generation of frustrated users.

The **SCSI** (**Small Computer System Interface**) bus is a high-performance bus intended for fast disks, scanners, and other devices needing considerable bandwidth. Nowadays, we find them mostly in servers and workstations. They can run at up to 640 MB/sec.

To work in an environment such as that of Fig. 1-12, the operating system has to know what peripheral devices are connected to the computer and configure them. This requirement led Intel and Microsoft to design a PC system called **plug and play**, based on a similar concept first implemented in the Apple Macintosh. Before plug and play, each I/O card had a fixed interrupt request level and fixed addresses for its I/O registers. For example, the keyboard was interrupt 1 and used

I/O addresses 0x60 to 0x64, the floppy disk controller was interrupt 6 and used I/O addresses 0x3F0 to 0x3F7, and the printer was interrupt 7 and used I/O addresses 0x378 to 0x37A, and so on.

So far, so good. The trouble came in when the user bought a sound card and a modem card and both happened to use, say, interrupt 4. They would conflict and would not work together. The solution was to include DIP switches or jumpers on every I/O card and instruct the user to please set them to select an interrupt level and I/O device addresses that did not conflict with any others in the user's system. Teenagers who devoted their lives to the intricacies of the PC hardware could sometimes do this without making errors. Unfortunately, nobody else could, leading to chaos.

What plug and play does is have the system automatically collect information about the I/O devices, centrally assign interrupt levels and I/O addresses, and then tell each card what its numbers are. This work is closely related to booting the computer, so let us look at that. It is not completely trivial.

1.3.6 Booting the Computer

Very briefly, the boot process is as follows. Every PC contains a parentboard (formerly called a motherboard before political correctness hit the computer industry). On the parentboard is a program called the system **BIOS** (**Basic Input Output System**). The BIOS contains low-level I/O software, including procedures to read the keyboard, write to the screen, and do disk I/O, among other things. Nowadays, it is held in a flash RAM, which is nonvolatile but which can be updated by the operating system when bugs are found in the BIOS.

When the computer is booted, the BIOS is started. It first checks to see how much RAM is installed and whether the keyboard and other basic devices are installed and responding correctly. It starts out by scanning the PCIe and PCI buses to detect all the devices attached to them. If the devices present are different from when the system was last booted, the new devices are configured.

The BIOS then determines the boot device by trying a list of devices stored in the CMOS memory. The user can change this list by entering a BIOS configuration program just after booting. Typically, an attempt is made to boot from a CD-ROM (or sometimes USB) drive, if one is present. If that fails, the system boots from the hard disk. The first sector from the boot device is read into memory and executed. This sector contains a program that normally examines the partition table at the end of the boot sector to determine which partition is active. Then a secondary boot loader is read in from that partition. This loader reads in the operating system from the active partition and starts it.

The operating system then queries the BIOS to get the configuration information. For each device, it checks to see if it has the device driver. If not, it asks the user to insert a CD-ROM containing the driver (supplied by the device's manufacturer) or to download it from the Internet. Once it has all the device drivers, the operating system loads them into the kernel. Then it initializes its tables, creates whatever background processes are needed, and starts up a login program or GUI.

1.4 THE OPERATING SYSTEM ZOO

Operating systems have been around now for over half a century. During this time, quite a variety of them have been developed, not all of them widely known. In this section we will briefly touch upon nine of them. We will come back to some of these different kinds of systems later in the book.

1.4.1 Mainframe Operating Systems

At the high end are the operating systems for mainframes, those room-sized computers still found in major corporate data centers. These computers differ from personal computers in terms of their I/O capacity. A mainframe with 1000 disks and millions of gigabytes of data is not unusual; a personal computer with these specifications would be the envy of its friends. Mainframes are also making something of a comeback as high-end Web servers, servers for large-scale electronic commerce sites, and servers for business-to-business transactions.

The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amounts of I/O. They typically offer three kinds of services: batch, transaction processing, and timesharing. A batch system is one that processes routine jobs without any interactive user present. Claims processing in an insurance company or sales reporting for a chain of stores is typically done in batch mode. Transaction-processing systems handle large numbers of small requests, for example, check processing at a bank or airline reservations. Each unit of work is small, but the system must handle hundreds or thousands per second. Timesharing systems allow multiple remote users to run jobs on the computer at once, such as querying a big database. These functions are closely related; mainframe operating systems often perform all of them. An example mainframe operating system is OS/390, a descendant of OS/360. However, mainframe operating systems are gradually being replaced by UNIX variants such as Linux.

1.4.2 Server Operating Systems

One level down are the server operating systems. They run on servers, which are either very large personal computers, workstations, or even mainframes. They serve multiple users at once over a network and allow the users to share hardware and software resources. Servers can provide print service, file service, or Web service. Internet providers run many server machines to support their customers and Websites use servers to store the Web pages and handle the incoming requests. Typical server operating systems are Solaris, FreeBSD, Linux and Windows Server 201x.

1.4.3 Multiprocessor Operating Systems

An increasingly common way to get major-league computing power is to connect multiple CPUs into a single system. Depending on precisely how they are connected and what is shared, these systems are called parallel computers, multicomputers, or multiprocessors. They need special operating systems, but often these are variations on the server operating systems, with special features for communication, connectivity, and consistency.

With the recent advent of multicore chips for personal computers, even conventional desktop and notebook operating systems are starting to deal with at least small-scale multiprocessors and the number of cores is likely to grow over time. Luckily, quite a bit is known about multiprocessor operating systems from years of previous research, so using this knowledge in multicore systems should not be hard. The hard part will be having applications make use of all this computing power. Many popular operating systems, including Windows and Linux, run on multiprocessors.

1.4.4 Personal Computer Operating Systems

The next category is the personal computer operating system. Modern ones all support multiprogramming, often with dozens of programs started up at boot time. Their job is to provide good support to a single user. They are widely used for word processing, spreadsheets, games, and Internet access. Common examples are Linux, FreeBSD, Windows 7, Windows 8, and Apple's OS X. Personal computer operating systems are so widely known that probably little introduction is needed. In fact, many people are not even aware that other kinds exist.

1.4.5 Handheld Computer Operating Systems

Continuing on down to smaller and smaller systems, we come to tablets, smartphones and other handheld computers. A handheld computer, originally known as a **PDA** (**Personal Digital Assistant**), is a small computer that can be held in your hand during operation. Smartphones and tablets are the best-known examples. As we have already seen, this market is currently dominated by Google's Android and Apple's iOS, but they have many competitors. Most of these devices boast multicore CPUs, GPS, cameras and other sensors, copious amounts of memory, and sophisticated operating systems. Moreover, all of them have more third-party applications (**"apps"**) than you can shake a (USB) stick at.

1.4.6 Embedded Operating Systems

Embedded systems run on the computers that control devices that are not generally thought of as computers and which do not accept user-installed software. Typical examples are microwave ovens, TV sets, cars, DVD recorders, traditional phones, and MP3 players. The main property which distinguishes embedded systems from handhelds is the certainty that no untrusted software will ever run on it. You cannot download new applications to your microwave oven—all the software is in ROM. This means that there is no need for protection between applications, leading to design simplification. Systems such as Embedded Linux, QNX and VxWorks are popular in this domain.

1.4.7 Sensor-Node Operating Systems

Networks of tiny sensor nodes are being deployed for numerous purposes. These nodes are tiny computers that communicate with each other and with a base station using wireless communication. Sensor networks are used to protect the perimeters of buildings, guard national borders, detect fires in forests, measure temperature and precipitation for weather forecasting, glean information about enemy movements on battlefields, and much more.

The sensors are small battery-powered computers with built-in radios. They have limited power and must work for long periods of time unattended outdoors, frequently in environmentally harsh conditions. The network must be robust enough to tolerate failures of individual nodes, which happen with ever-increasing frequency as the batteries begin to run down.

Each sensor node is a real computer, with a CPU, RAM, ROM, and one or more environmental sensors. It runs a small, but real operating system, usually one that is event driven, responding to external events or making measurements periodically based on an internal clock. The operating system has to be small and simple because the nodes have little RAM and battery lifetime is a major issue. Also, as with embedded systems, all the programs are loaded in advance; users do not suddenly start programs they downloaded from the Internet, which makes the design much simpler. TinyOS is a well-known operating system for a sensor node.

1.4.8 Real-Time Operating Systems

Another type of operating system is the real-time system. These systems are characterized by having time as a key parameter. For example, in industrial process-control systems, real-time computers have to collect data about the production process and use it to control machines in the factory. Often there are hard deadlines that must be met. For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time. If, for example, a welding robot welds too early or too late, the car will be ruined. If the action absolutely *must* occur at a certain moment (or within a certain range), we have a **hard real-time system**. Many of these are found in industrial process control, avionics, military, and similar application areas. These systems must provide absolute guarantees that a certain action will occur by a certain time.

A **soft real-time system**, is one where missing an occasional deadline, while not desirable, is acceptable and does not cause any permanent damage. Digital audio or multimedia systems fall in this category. Smartphones are also soft real-time systems.

Since meeting deadlines is crucial in (hard) real-time systems, sometimes the operating system is simply a library linked in with the application programs, with everything tightly coupled and no protection between parts of the system. An example of this type of real-time system is eCos.

The categories of handhelds, embedded systems, and real-time systems overlap considerably. Nearly all of them have at least some soft real-time aspects. The embedded and real-time systems run only software put in by the system designers; users cannot add their own software, which makes protection easier. The handhelds and embedded systems are intended for consumers, whereas real-time systems are more for industrial usage. Nevertheless, they have a certain amount in common.

1.4.9 Smart Card Operating Systems

The smallest operating systems run on smart cards, which are credit-card-sized devices containing a CPU chip. They have very severe processing power and memory constraints. Some are powered by contacts in the reader into which they are inserted, but contactless smart cards are inductively powered, which greatly limits what they can do. Some of them can handle only a single function, such as electronic payments, but others can handle multiple functions. Often these are proprietary systems.

Some smart cards are Java oriented. This means that the ROM on the smart card holds an interpreter for the Java Virtual Machine (JVM). Java applets (small programs) are downloaded to the card and are interpreted by the JVM interpreter. Some of these cards can handle multiple Java applets at the same time, leading to multiprogramming and the need to schedule them. Resource management and protection also become an issue when two or more applets are present at the same time. These issues must be handled by the (usually extremely primitive) operating system present on the card.

1.5 OPERATING SYSTEM CONCEPTS

Most operating systems provide certain basic concepts and abstractions such as processes, address spaces, and files that are central to understanding them. In the following sections, we will look at some of these basic concepts ever so briefly, as an introduction. We will come back to each of them in great detail later in this book. To illustrate these concepts we will, from time to time, use examples, generally drawn from UNIX. Similar examples typically exist in other systems as well, however, and we will study some of them later.

1.5.1 Processes

A key concept in all operating systems is the **process**. A process is basically a program in execution. Associated with each process is its **address space**, a list of memory locations from 0 to some maximum, which the process can read and write. The address space contains the executable program, the program's data, and its stack. Also associated with each process is a set of resources, commonly including registers (including the program counter and stack pointer), a list of open files, outstanding alarms, lists of related processes, and all the other information needed to run the program. A process is fundamentally a container that holds all the information needed to run a program.

We will come back to the process concept in much more detail in Chap. 2. For the time being, the easiest way to get a good intuitive feel for a process is to think about a multiprogramming system. The user may have started a video editing program and instructed it to convert a one-hour video to a certain format (something that can take hours) and then gone off to surf the Web. Meanwhile, a background process that wakes up periodically to check for incoming email may have started running. Thus we have (at least) three active processes: the video editor, the Web browser, and the email receiver. Periodically, the operating system decides to stop running one process and start running another, perhaps because the first one has used up more than its share of CPU time in the past second or two.

When a process is suspended temporarily like this, it must later be restarted in exactly the same state it had when it was stopped. This means that all information about the process must be explicitly saved somewhere during the suspension. For example, the process may have several files open for reading at once. Associated with each of these files is a pointer giving the current position (i.e., the number of the byte or record to be read next). When a process is temporarily suspended, all these pointers must be saved so that a read call executed after the process is restarted will read the proper data. In many operating systems, all the information about each process, other than the contents of its own address space, is stored in an operating system table called the **process table**, which is an array of structures, one for each process currently in existence.

Thus, a (suspended) process consists of its address space, usually called the **core image** (in honor of the magnetic core memories used in days of yore), and its process table entry, which contains the contents of its registers and many other items needed to restart the process later.

The key process-management system calls are those dealing with the creation and termination of processes. Consider a typical example. A process called the **command interpreter** or shell reads commands from a terminal. The user has just typed a command requesting that a program be compiled. The shell must now create a new process that will run the compiler. When that process has finished the compilation, it executes a system call to terminate itself.

If a process can create one or more other processes (referred to as **child processes**) and these processes in turn can create child processes, we quickly arrive at the process tree structure of Fig. 1-13. Related processes that are cooperating to get some job done often need to communicate with one another and synchronize their activities. This communication is called **interprocess communication**, and will be addressed in detail in Chap. 2.

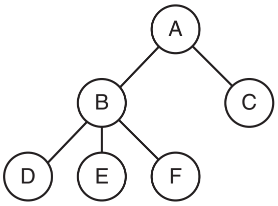


Figure 1-13. A process tree. Process *A* created two child processes, *B* and *C*. Process *B* created three child processes, *D*, *E*, and *F*.

Other process system calls are available to request more memory (or release unused memory), wait for a child process to terminate, and overlay its program with a different one.

Occasionally, there is a need to convey information to a running process that is not sitting around waiting for this information. For example, a process that is communicating with another process on a different computer does so by sending messages to the remote process over a computer network. To guard against the possibility that a message or its reply is lost, the sender may request that its own operating system notify it after a specified number of seconds, so that it can retransmit the message if no acknowledgement has been received yet. After setting this timer, the program may continue doing other work.

When the specified number of seconds has elapsed, the operating system sends an **alarm signal** to the process. The signal causes the process to temporarily suspend whatever it was doing, save its registers on the stack, and start running a special signal-handling procedure, for example, to retransmit a presumably lost message. When the signal handler is done, the running process is restarted in the state it was in just before the signal. Signals are the software analog of hardware interrupts and can be generated by a variety of causes in addition to timers expiring. Many traps detected by hardware, such as executing an illegal instruction or using an invalid address, are also converted into signals to the guilty process.

Each person authorized to use a system is assigned a **UID** (**User IDentification**) by the system administrator. Every process started has the UID of the person who started it. A child process has the same UID as its parent. Users can be members of groups, each of which has a **GID** (**Group IDentification**).

One UID, called the **superuser** (in UNIX), or **Administrator** (in Windows), has special power and may override many of the protection rules. In large installations, only the system administrator knows the password needed to become superuser, but many of the ordinary users (especially students) devote considerable effort seeking flaws in the system that allow them to become superuser without the password.

We will study processes and interprocess communication in Chap. 2.

1.5.2 Address Spaces

Every computer has some main memory that it uses to hold executing programs. In a very simple operating system, only one program at a time is in memory. To run a second program, the first one has to be removed and the second one placed in memory.

More sophisticated operating systems allow multiple programs to be in memory at the same time. To keep them from interfering with one another (and with the operating system), some kind of protection mechanism is needed. While this mechanism has to be in the hardware, it is controlled by the operating system.

The above viewpoint is concerned with managing and protecting the computer's main memory. A different, but equally important, memory-related issue is managing the address space of the processes. Normally, each process has some set of addresses it can use, typically running from 0 up to some maximum. In the simplest case, the maximum amount of address space a process has is less than the main memory. In this way, a process can fill up its address space and there will be enough room in main memory to hold it all.

However, on many computers addresses are 32 or 64 bits, giving an address space of 2^{32} or 2^{64} bytes, respectively. What happens if a process has more address space than the computer has main memory and the process wants to use it all? In the first computers, such a process was just out of luck. Nowadays, a technique called virtual memory exists, as mentioned earlier, in which the operating system keeps part of the address space in main memory and part on disk and shuttles pieces back and forth between them as needed. In essence, the operating system creates the abstraction of an address space as the set of addresses a process may reference. The address space is decoupled from the machine's physical memory and may be either larger or smaller than the physical memory. Management of address spaces and physical memory form an important part of what an operating system does, so all of Chap. 3 is devoted to this topic.

1.5.3 Files

Another key concept supported by virtually all operating systems is the file system. As noted before, a major function of the operating system is to hide the peculiarities of the disks and other I/O devices and present the programmer with a nice, clean abstract model of device-independent files. System calls are obviously needed to create files, remove files, read files, and write files. Before a file can be read, it must be located on the disk and opened, and after being read it should be closed, so calls are provided to do these things.

To provide a place to keep files, most PC operating systems have the concept of a **directory** as a way of grouping files together. A student, for example, might have one directory for each course he is taking (for the programs needed for that course), another directory for his electronic mail, and still another directory for his World Wide Web home page. System calls are then needed to create and remove directories. Calls are also provided to put an existing file in a directory and to remove a file from a directory. Directory entries may be either files or other directories. This model also gives rise to a hierarchy—the file system—as shown in Fig. 1-14.

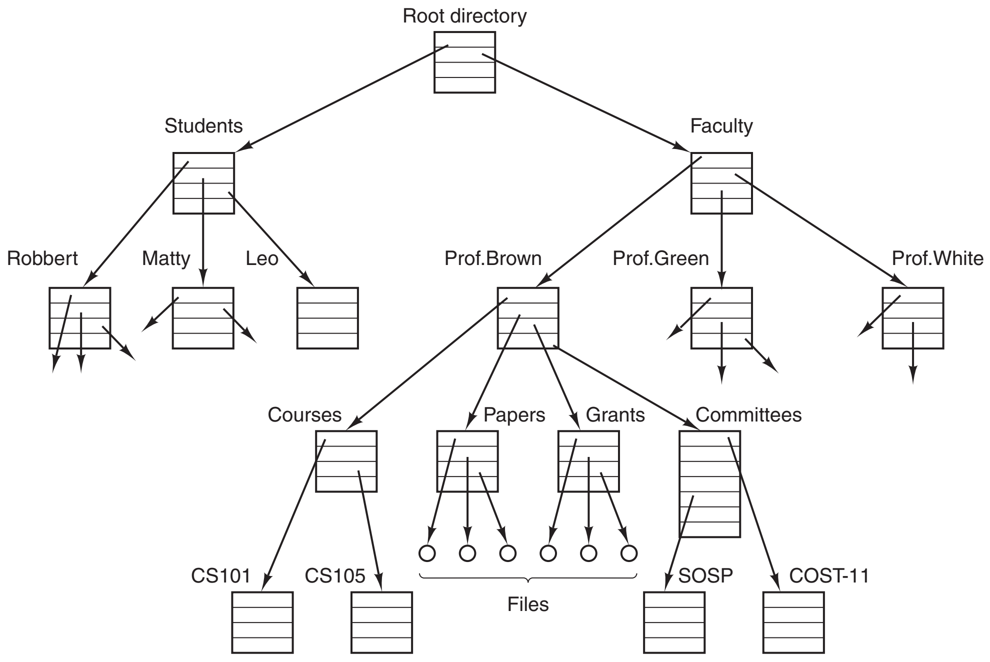


Figure 1-14. A file system for a university department.

The process and file hierarchies both are organized as trees, but the similarity stops there. Process hierarchies usually are not very deep (more than three levels is unusual), whereas file hierarchies are commonly four, five, or even more levels deep. Process hierarchies are typically short-lived, generally minutes at most, whereas the directory hierarchy may exist for years. Ownership and protection also differ for processes and files. Typically, only a parent process may control or even access a child process, but mechanisms nearly always exist to allow files and directories to be read by a wider group than just the owner.

Every file within the directory hierarchy can be specified by giving its **path name** from the top of the directory hierarchy, the **root directory**. Such absolute path names consist of the list of directories that must be traversed from the root directory to get to the file, with slashes separating the components. In Fig. 1-14, the path for file *CS101* is */Faculty/Prof.Brown/Courses/CS101*. The leading slash indicates that the path is absolute, that is, starting at the root directory. As an aside, in Windows, the backslash (\) character is used as the separator instead of the slash (/) character (for historical reasons), so the file path given above would be written as *\Faculty\Prof.Brown\Courses\CS101*. Throughout this book we will generally use the UNIX convention for paths.

At every instant, each process has a current **working directory**, in which path names not beginning with a slash are looked for. For example, in Fig. 1-14, if */Faculty/Prof.Brown* were the working directory, use of the path *Courses/CS101* would yield the same file as the absolute path name given above. Processes can change their working directory by issuing a system call specifying the new working directory.

Before a file can be read or written, it must be opened, at which time the permissions are checked. If the access is permitted, the system returns a small integer called a **file descriptor** to use in subsequent operations. If the access is prohibited, an error code is returned.

Another important concept in UNIX is the mounted file system. Most desktop computers have one or more optical drives into which CD-ROMs, DVDs, and Bluray discs can be inserted. They almost always have USB ports, into which USB memory sticks (really, solid state disk drives) can be plugged, and some computers have floppy disks or external hard disks. To provide an elegant way to deal with these removable media UNIX allows the file system on the optical disc to be attached to the main tree. Consider the situation of Fig. 1-15(a). Before the mount call, the **root file system**, on the hard disk, and a second file system, on a CD-ROM, are separate and unrelated.

However, the file system on the CD-ROM cannot be used, because there is no way to specify path names on it. UNIX does not allow path names to be prefixed by a drive name or number; that would be precisely the kind of device dependence that operating systems ought to eliminate. Instead, the mount system call allows the file system on the CD-ROM to be attached to the root file system wherever the program wants it to be. In Fig. 1-15(b) the file system on the CD-ROM has been mounted on directory *b*, thus allowing access to files */b/x* and */b/y*. If directory *b* had contained any files they would not be accessible while the CD-ROM was mounted, since */b* would refer to the root directory of the CD-ROM. (Not being able to access these files is not as serious as it at first seems: file systems are nearly always mounted on empty directories.) If a system contains multiple hard disks, they can all be mounted into a single tree as well.

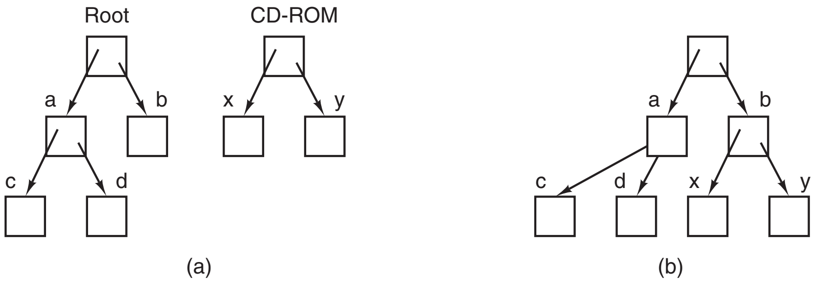


Figure 1-15. (a) Before mounting, the files on the CD-ROM are not accessible. (b) After mounting, they are part of the file hierarchy.

Another important concept in UNIX is the **special file**. Special files are provided in order to make I/O devices look like files. That way, they can be read and written using the same system calls as are used for reading and writing files. Two kinds of special files exist: **block special files** and **character special files**. Block special files are used to model devices that consist of a collection of randomly addressable blocks, such as disks. By opening a block special file and reading, say, block 4, a program can directly access the fourth block on the device, without regard to the structure of the file system contained on it. Similarly, character special files are used to model printers, modems, and other devices that accept or output a character stream. By convention, the special files are kept in the */dev* directory. For example, */dev/lp* might be the printer (once called the line printer).

The last feature we will discuss in this overview relates to both processes and files: pipes. A **pipe** is a sort of pseudofile that can be used to connect two processes, as shown in Fig. 1-16. If processes *A* and *B* wish to talk using a pipe, they must set it up in advance. When process *A* wants to send data to process *B*, it writes on the pipe as though it were an output file. In fact, the implementation of a pipe is very much like that of a file. Process *B* can read the data by reading from the pipe as though it were an input file. Thus, communication between processes in UNIX looks very much like ordinary file reads and writes. Stronger yet, the only way a process can discover that the output file it is writing on is not really a file, but a pipe, is by making a special system call. File systems are very important. We will have much more to say about them in Chap. 4 and also in Chaps. 10 and 11.

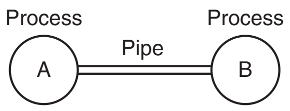


Figure 1-16. Two processes connected by a pipe.

1.5.4 Input/Output

All computers have physical devices for acquiring input and producing output. After all, what good would a computer be if the users could not tell it what to do and could not get the results after it did the work requested? Many kinds of input and output devices exist, including keyboards, monitors, printers, and so on. It is up to the operating system to manage these devices.

Consequently, every operating system has an I/O subsystem for managing its I/O devices. Some of the I/O software is device independent, that is, applies to many or all I/O devices equally well. Other parts of it, such as device drivers, are specific to particular I/O devices. In Chap. 5 we will have a look at I/O software.

1.5.5 Protection

Computers contain large amounts of information that users often want to protect and keep confidential. This information may include email, business plans, tax returns, and much more. It is up to the operating system to manage the system security so that files, for example, are accessible only to authorized users.

As a simple example, just to get an idea of how security can work, consider UNIX. Files in UNIX are protected by assigning each one a 9-bit binary protection code. The protection code consists of three 3-bit fields, one for the owner, one for other members of the owner's group (users are divided into groups by the system administrator), and one for everyone else. Each field has a bit for read access, a bit for write access, and a bit for execute access. These 3 bits are known as the **rwx bits**. For example, the protection code *rwxr-x--x* means that the owner can **r**ead, **w**rite, or e**x**ecute the file, other group members can read or execute (but not write) the file, and everyone else can execute (but not read or write) the file. For a directory, *x* indicates search permission. A dash means that the corresponding permission is absent.

In addition to file protection, there are many other security issues. Protecting the system from unwanted intruders, both human and nonhuman (e.g., viruses) is one of them. We will look at various security issues in Chap. 9.

1.5.6 The Shell

The operating system is the code that carries out the system calls. Editors, compilers, assemblers, linkers, utility programs, and command interpreters definitely are not part of the operating system, even though they are important and useful. At the risk of confusing things somewhat, in this section we will look briefly at the UNIX command interpreter, the shell. Although it is not part of the operating system, it makes heavy use of many operating system features and thus serves as a good example of how the system calls are used. It is also the main interface between a user sitting at his terminal and the operating system, unless the user is using a graphical user interface. Many shells exist, including *sh*, *csh*, *ksh*, and *bash*. All of them support the functionality described below, which derives from the original shell (*sh*).

When any user logs in, a shell is started up. The shell has the terminal as standard input and standard output. It starts out by typing the **prompt**, a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user now types

```
date
```

for example, the shell creates a child process and runs the *date* program as the child. While the child process is running, the shell waits for it to terminate. When the child finishes, the shell types the prompt again and tries to read the next input line.

The user can specify that standard output be redirected to a file, for example,

```
date >file
```

Similarly, standard input can be redirected, as in

```
sort <file1 >file2
```

which invokes the sort program with input taken from *file1* and output sent to *file2*.

The output of one program can be used as the input for another program by connecting them with a pipe. Thus

```
cat file1 file2 file3 | sort >/dev/lp
```

invokes the *cat* program to con*cat*enate three files and send the output to *sort* to arrange all the lines in alphabetical order. The output of *sort* is redirected to the file */dev/lp*, typically the printer.

If a user puts an ampersand after a command, the shell does not wait for it to complete. Instead it just gives a prompt immediately. Consequently,

```
cat file1 file2 file3 | sort >/dev/lp &
```

starts up the sort as a background job, allowing the user to continue working normally while the sort is going on. The shell has a number of other interesting features, which we do not have space to discuss here. Most books on UNIX discuss the shell at some length (e.g., Kernighan and Pike, 1984; Quigley, 2004; Robbins, 2005).

Most personal computers these days use a GUI. In fact, the GUI is just a program running on top of the operating system, like a shell. In Linux systems, this fact is made obvious because the user has a choice of (at least) two GUIs: Gnome and KDE or none at all (using a terminal window on X11). In Windows, it is also possible to replace the standard GUI desktop (*Windows Explorer*) with a different program by changing some values in the registry, although few people do this.

1.5.7 Ontogeny Recapitulates Phylogeny

After Charles Darwin's book *On the Origin of the Species* was published, the German zoologist Ernst Haeckel stated that "ontogeny recapitulates phylogeny." By this he meant that the development of an embryo (ontogeny) repeats (i.e., recapitulates) the evolution of the species (phylogeny). In other words, after fertilization, a human egg goes through stages of being a fish, a pig, and so on before turning into a human baby. Modern biologists regard this as a gross simplification, but it still has a kernel of truth in it.

Something vaguely analogous has happened in the computer industry. Each new species (mainframe, minicomputer, personal computer, handheld, embedded computer, smart card, etc.) seems to go through the development that its ancestors did, both in hardware and in software. We often forget that much of what happens in the computer business and a lot of other fields is technology driven. The reason the ancient Romans lacked cars is not that they liked walking so much. It is because they did not know how to build cars. Personal computers exist *not* because millions of people have a centuries-old pent-up desire to own a computer, but because it is now possible to manufacture them cheaply. We often forget how much technology affects our view of systems and it is worth reflecting on this point from time to time.

In particular, it frequently happens that a change in technology renders some idea obsolete and it quickly vanishes. However, another change in technology could revive it again. This is especially true when the change has to do with the relative performance of different parts of the system. For instance, when CPUs became much faster than memories, caches became important to speed up the "slow" memory. If new memory technology someday makes memories much faster than CPUs, caches will vanish. And if a new CPU technology makes them faster than memories again, caches will reappear. In biology, extinction is forever, but in computer science, it is sometimes only for a few years.

As a consequence of this impermanence, in this book we will from time to time look at "obsolete" concepts, that is, ideas that are not optimal with current technology. However, changes in the technology may bring back some of the so-called "obsolete concepts." For this reason, it is important to understand why a concept is obsolete and what changes in the environment might bring it back again.

To make this point clearer, let us consider a simple example. Early computers had hardwired instruction sets. The instructions were executed directly by hardware and could not be changed. Then came microprogramming (first introduced on a large scale with the IBM 360), in which an underlying interpreter carried out the "hardware instructions" in software. Hardwired execution became obsolete. It was not flexible enough. Then RISC computers were invented, and microprogramming (i.e., interpreted execution) became obsolete because direct execution was faster. Now we are seeing the resurgence of interpretation in the form of Java applets that are sent over the Internet and interpreted upon arrival. Execution speed is not always crucial because network delays are so great that they tend to dominate. Thus the pendulum has already swung several cycles between direct execution and interpretation and may yet swing again in the future.

Large Memories

Let us now examine some historical developments in hardware and how they have affected software repeatedly. The first mainframes had limited memory. A fully loaded IBM 7090 or 7094, which played king of the mountain from late 1959 until 1964, had just over 128 KB of memory. It was mostly programmed in assembly language and its operating system was written in assembly language to save precious memory.

As time went on, compilers for languages like FORTRAN and COBOL got good enough that assembly language was pronounced dead. But when the first commercial minicomputer (the PDP-1) was released, it had only 4096 18-bit words of memory, and assembly language made a surprise comeback. Eventually, minicomputers acquired more memory and high-level languages became prevalent on them.

When microcomputers hit in the early 1980s, the first ones had 4-KB memories and assembly-language programming rose from the dead. Embedded computers often used the same CPU chips as the microcomputers (8080s, Z80s, and later 8086s) and were also programmed in assembler initially. Now their descendants, the personal computers, have lots of memory and are programmed in C, C++, Java, and other high-level languages. Smart cards are undergoing a similar development, although beyond a certain size, the smart cards often have a Java interpreter and execute Java programs interpretively, rather than having Java being compiled to the smart card's machine language.

Protection Hardware

Early mainframes, like the IBM 7090/7094, had no protection hardware, so they just ran one program at a time. A buggy program could wipe out the operating system and easily crash the machine. With the introduction of the IBM 360, a primitive form of hardware protection became available. These machines could then hold several programs in memory at the same time and let them take turns running (multiprogramming). Monoprogramming was declared obsolete.

At least until the first minicomputer showed up—without protection hardware—so multiprogramming was not possible. Although the PDP-1 and PDP-8 had no protection hardware, eventually the PDP-11 did, and this feature led to multiprogramming and eventually to UNIX.

When the first microcomputers were built, they used the Intel 8080 CPU chip, which had no hardware protection, so we were back to monoprogramming—one program in memory at a time. It was not until the Intel 80286 chip that protection hardware was added and multiprogramming became possible. Until this day, many embedded systems have no protection hardware and run just a single program.

Now let us look at operating systems. The first mainframes initially had no protection hardware and no support for multiprogramming, so they ran simple operating systems that handled one manually loaded program at a time. Later they acquired the hardware and operating system support to handle multiple programs at once, and then full timesharing capabilities.

When minicomputers first appeared, they also had no protection hardware and ran one manually loaded program at a time, even though multiprogramming was well established in the mainframe world by then. Gradually, they acquired protection hardware and the ability to run two or more programs at once. The first microcomputers were also capable of running only one program at a time, but later acquired the ability to multiprogram. Handheld computers and smart cards went the same route.

In all cases, the software development was dictated by technology. The first microcomputers, for example, had something like 4 KB of memory and no protection hardware. High-level languages and multiprogramming were simply too much for such a tiny system to handle. As the microcomputers evolved into modern personal computers, they acquired the necessary hardware and then the necessary software to handle more advanced features. It is likely that this development will continue for years to come. Other fields may also have this wheel of reincarnation, but in the computer industry it seems to spin faster.

Disks

Early mainframes were largely magnetic-tape based. They would read in a program from tape, compile it, run it, and write the results back to another tape. There were no disks and no concept of a file system. That began to change when IBM introduced the first hard disk—the RAMAC (RAndoM ACcess) in 1956. It occupied about 4 square meters of floor space and could store 5 million 7-bit characters, enough for one medium-resolution digital photo. But with an annual rental fee of $35,000, assembling enough of them to store the equivalent of a roll of film got pricey quite fast. But eventually prices came down and primitive file systems were developed.

Typical of these new developments was the CDC 6600, introduced in 1964 and for years by far the fastest computer in the world. Users could create so-called "permanent files" by giving them names and hoping that no other user had also decided that, say, "data" was a suitable name for a file. This was a single-level directory. Eventually, mainframes developed complex hierarchical file systems, perhaps culminating in the MULTICS file system.

As minicomputers came into use, they eventually also had hard disks. The standard disk on the PDP-11 when it was introduced in 1970 was the RK05 disk, with a capacity of 2.5 MB, about half of the IBM RAMAC, but it was only about

40 cm in diameter and 5 cm high. But it, too, had a single-level directory initially. When microcomputers came out, CP/M was initially the dominant operating system, and it, too, supported just one directory on the (floppy) disk.

Virtual Memory

Virtual memory (discussed in Chap. 3) gives the ability to run programs larger than the machine's physical memory by rapidly moving pieces back and forth between RAM and disk. It underwent a similar development, first appearing on mainframes, then moving to the minis and the micros. Virtual memory also allowed having a program dynamically link in a library at run time instead of having it compiled in. MULTICS was the first system to allow this. Eventually, the idea propagated down the line and is now widely used on most UNIX and Windows systems.

In all these developments, we see ideas invented in one context and later thrown out when the context changes (assembly-language programming, monoprogramming, single-level directories, etc.) only to reappear in a different context often a decade later. For this reason in this book we will sometimes look at ideas and algorithms that may seem dated on today's gigabyte PCs, but which may soon come back on embedded computers and smart cards.

1.6 SYSTEM CALLS

We have seen that operating systems have two main functions: providing abstractions to user programs and managing the computer's resources. For the most part, the interaction between user programs and the operating system deals with the former; for example, creating, writing, reading, and deleting files. The resource-management part is largely transparent to the users and done automatically. Thus, the interface between user programs and the operating system is primarily about dealing with the abstractions. To really understand what operating systems do, we must examine this interface closely. The system calls available in the interface vary from one operating system to another (although the underlying concepts tend to be similar).

We are thus forced to make a choice between (1) vague generalities ("operating systems have system calls for reading files") and (2) some specific system ("UNIX has a read system call with three parameters: one to specify the file, one to tell where the data are to be put, and one to tell how many bytes to read").

We have chosen the latter approach. It's more work that way, but it gives more insight into what operating systems really do. Although this discussion specifically refers to POSIX (International Standard 9945-1), hence also to UNIX, System V, BSD, Linux, MINIX 3, and so on, most other modern operating systems have system calls that perform the same functions, even if the details differ. Since the actual mechanics of issuing a system call are highly machine dependent and often must be expressed in assembly code, a procedure library is provided to make it possible to make system calls from C programs and often from other languages as well.

It is useful to keep the following in mind. Any single-CPU computer can execute only one instruction at a time. If a process is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a trap instruction to transfer control to the operating system. The operating system then figures out what the calling process wants by inspecting the parameters. Then it carries out the system call and returns control to the instruction following the system call. In a sense, making a system call is like making a special kind of procedure call, only system calls enter the kernel and procedure calls do not.

To make the system-call mechanism clearer, let us take a quick look at the read system call. As mentioned above, it has three parameters: the first one specifying the file, the second one pointing to the buffer, and the third one giving the number of bytes to read. Like nearly all system calls, it is invoked from C programs by calling a library procedure with the same name as the system call: *read*. A call from a C program might look like this:

```
count = read(fd, buffer, nbytes);
```

The system call (and the library procedure) return the number of bytes actually read in *count*. This value is normally the same as *nbytes*, but may be smaller, if, for example, end-of-file is encountered while reading.

If the system call cannot be carried out owing to an invalid parameter or a disk error, *count* is set to −1, and the error number is put in a global variable, *errno*. Programs should always check the results of a system call to see if an error occurred.

System calls are performed in a series of steps. To make this concept clearer, let us examine the read call discussed above. In preparation for calling the *read* library procedure, which actually makes the read system call, the calling program first pushes the parameters onto the stack, as shown in steps 1–3 in Fig. 1-17.

C and C++ compilers push the parameters onto the stack in reverse order for historical reasons (having to do with making the first parameter to *printf*, the format string, appear on top of the stack). The first and third parameters are called by value, but the second parameter is passed by reference, meaning that the address of the buffer (indicated by &) is passed, not the contents of the buffer. Then comes the actual call to the library procedure (step 4). This instruction is the normal procedure-call instruction used to call all procedures.

The library procedure, possibly written in assembly language, typically puts the system-call number in a place where the operating system expects it, such as a register (step 5). Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel (step 6). The TRAP instruction is actually fairly similar to the procedure-call instruction in the

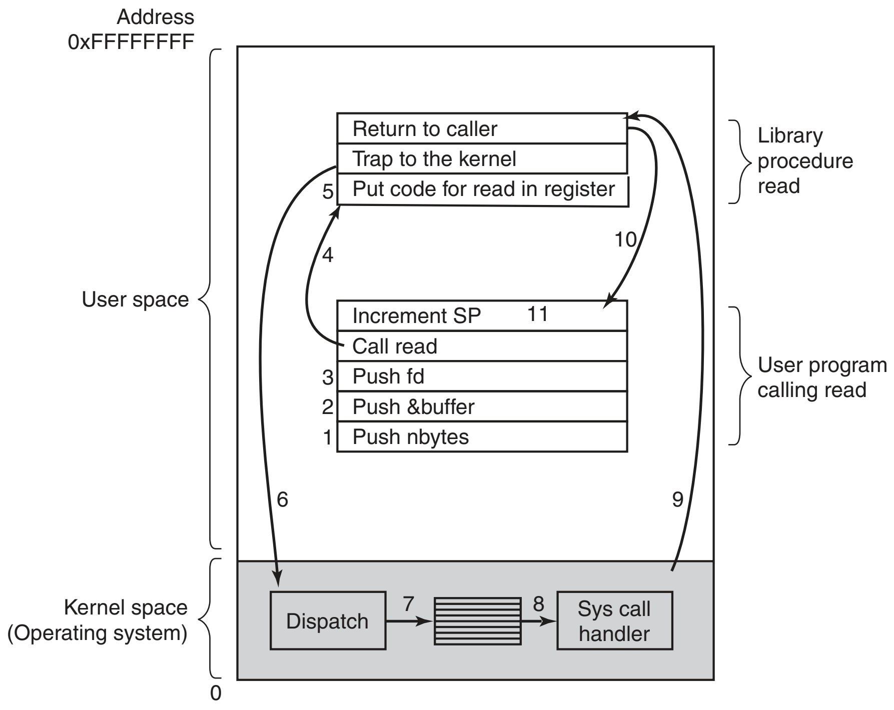


Figure 1-17. The 11 steps in making the system call read(fd, buffer, nbytes).

sense that the instruction following it is taken from a distant location and the return address is saved on the stack for use later.

Nevertheless, the TRAP instruction also differs from the procedure-call instruction in two fundamental ways. First, as a side effect, it switches into kernel mode. The procedure call instruction does not change the mode. Second, rather than giving a relative or absolute address where the procedure is located, the TRAP instruction cannot jump to an arbitrary address. Depending on the architecture, either it jumps to a single fixed location or there is an 8-bit field in the instruction giving the index into a table in memory containing jump addresses, or equivalent.

The kernel code that starts following the TRAP examines the system-call number and then dispatches to the correct system-call handler, usually via a table of pointers to system-call handlers indexed on system-call number (step 7). At that point the system-call handler runs (step 8). Once it has completed its work, control may be returned to the user-space library procedure at the instruction following the TRAP instruction (step 9). This procedure then returns to the user program in the usual way procedure calls return (step 10).

To finish the job, the user program has to clean up the stack, as it does after any procedure call (step 11). Assuming the stack grows downward, as it often does, the compiled code increments the stack pointer exactly enough to remove the parameters pushed before the call to *read*. The program is now free to do whatever it wants to do next.

In step 9 above, we said "may be returned to the user-space library procedure" for good reason. The system call may block the caller, preventing it from continuing. For example, if it is trying to read from the keyboard and nothing has been typed yet, the caller has to be blocked. In this case, the operating system will look around to see if some other process can be run next. Later, when the desired input is available, this process will get the attention of the system and run steps 9–11.

In the following sections, we will examine some of the most heavily used POSIX system calls, or more specifically, the library procedures that make those system calls. POSIX has about 100 procedure calls. Some of the most important ones are listed in Fig. 1-18, grouped for convenience in four categories. In the text we will briefly examine each call to see what it does.

To a large extent, the services offered by these calls determine most of what the operating system has to do, since the resource management on personal computers is minimal (at least compared to big machines with multiple users). The services include things like creating and terminating processes, creating, deleting, reading, and writing files, managing directories, and performing input and output.

As an aside, it is worth pointing out that the mapping of POSIX procedure calls onto system calls is not one-to-one. The POSIX standard specifies a number of procedures that a conformant system must supply, but it does not specify whether they are system calls, library calls, or something else. If a procedure can be carried out without invoking a system call (i.e., without trapping to the kernel), it will usually be done in user space for reasons of performance. However, most of the POSIX procedures do invoke system calls, usually with one procedure mapping directly onto one system call. In a few cases, especially where several required procedures are only minor variations of one another, one system call handles more than one library call.

1.6.1 System Calls for Process Management

The first group of calls in Fig. 1-18 deals with process management. Fork is a good place to start the discussion. Fork is the only way to create a new process in POSIX. It creates an exact duplicate of the original process, including all the file descriptors, registers—everything. After the fork, the original process and the copy (the parent and child) go their separate ways. All the variables have identical values at the time of the fork, but since the parent's data are copied to create the child, subsequent changes in one of them do not affect the other one. (The program text, which is unchangeable, is shared between parent and child.) The fork call returns a value, which is zero in the child and equal to the child's **PID** (**Process IDentifier**) in the parent. Using the returned PID, the two processes can see which one is the parent process and which one is the child process.

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory- and file-system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Figure 1-18. Some of the major POSIX system calls. The return code *s* is −1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time. The parameters are explained in the text.

In most cases, after a fork, the child will need to execute different code from the parent. Consider the case of the shell. It reads a command from the terminal, forks off a child process, waits for the child to execute the command, and then reads the next command when the child terminates. To wait for the child to finish, the parent executes a waitpid system call, which just waits until the child terminates (any child if more than one exists). Waitpid can wait for a specific child, or for any old child by setting the first parameter to −1. When waitpid completes, the address pointed to by the second parameter, *statloc*, will be set to the child process' exit status (normal or abnormal termination and exit value). Various options are also provided, specified by the third parameter. For example, returning immediately if no child has already exited.

Now consider how fork is used by the shell. When a command is typed, the shell forks off a new process. This child process must execute the user command. It does this by using the execve system call, which causes its entire core image to be replaced by the file named in its first parameter. (Actually, the system call itself is exec, but several library procedures call it with different parameters and slightly different names. We will treat these as system calls here.) A highly simplified shell illustrating the use of fork, waitpid, and execve is shown in Fig. 1-19.

```
#define TRUE 1

while (TRUE) {                                      /* repeat forever */
      type_prompt( );                               /* display prompt on the screen */
      read_command(command, parameters);            /* read input from terminal */

      if (fork( ) != 0) {                           /* fork off child process */
          /* Parent code. */
          waitpid(−1, &status, 0);                  /* wait for child to exit */
      } else {
          /* Child code. */
          execve(command, parameters, 0);           /* execute command */
      }
}
```

Figure 1-19. A stripped-down shell. Throughout this book, *TRUE* is assumed to be defined as 1.

In the most general case, execve has three parameters: the name of the file to be executed, a pointer to the argument array, and a pointer to the environment array. These will be described shortly. Various library routines, including *execl*, *execv*, *execle*, and *execve*, are provided to allow the parameters to be omitted or specified in various ways. Throughout this book we will use the name exec to represent the system call invoked by all of these.

Let us consider the case of a command such as

```
cp file1 file2
```

used to copy *file1* to *file2*. After the shell has forked, the child process locates and executes the file *cp* and passes to it the names of the source and target files.

The main program of *cp* (and main program of most other C programs) contains the declaration

```
main(argc, argv, envp)
```

where *argc* is a count of the number of items on the command line, including the program name. For the example above, *argc* is 3.

The second parameter, *argv*, is a pointer to an array. Element *i* of that array is a pointer to the *i*th string on the command line. In our example, *argv*[0] would point to the string "cp", *argv*[1] would point to the string "file1", and *argv*[2] would point to the string "file2".

The third parameter of *main*, *envp*, is a pointer to the environment, an array of strings containing assignments of the form *name = value* used to pass information such as the terminal type and home directory name to programs. There are library procedures that programs can call to get the environment variables, which are often used to customize how a user wants to perform certain tasks (e.g., the default printer to use). In Fig. 1-19, no environment is passed to the child, so the third parameter of *execve* is a zero.

If exec seems complicated, do not despair; it is (semantically) the most complex of all the POSIX system calls. All the other ones are much simpler. As an example of a simple one, consider exit, which processes should use when they are finished executing. It has one parameter, the exit status (0 to 255), which is returned to the parent via *statloc* in the waitpid system call.

Processes in UNIX have their memory divided up into three segments: the **text segment** (i.e., the program code), the **data segment** (i.e., the variables), and the **stack segment**. The data segment grows upward and the stack grows downward, as shown in Fig. 1-20. Between them is a gap of unused address space. The stack grows into the gap automatically, as needed, but expansion of the data segment is done explicitly by using a system call, brk, which specifies the new address where the data segment is to end. This call, however, is not defined by the POSIX standard, since programmers are encouraged to use the *malloc* library procedure for dynamically allocating storage, and the underlying implementation of *malloc* was not thought to be a suitable subject for standardization since few programmers use it directly and it is doubtful that anyone even notices that brk is not in POSIX.

1.6.2 System Calls for File Management

Many system calls relate to the file system. In this section we will look at calls that operate on individual files; in the next one we will examine those that involve directories or the file system as a whole.

To read or write a file, it must first be opened. This call specifies the file name to be opened, either as an absolute path name or relative to the working directory, as well as a code of *O_RDONLY*, *O_WRONLY*, or *O_RDWR*, meaning open for reading, writing, or both. To create a new file, the *O_CREAT* parameter is used.

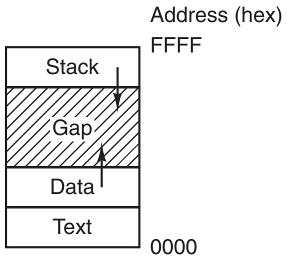


Figure 1-20. Processes have three segments: text, data, and stack.

The file descriptor returned can then be used for reading or writing. Afterward, the file can be closed by close, which makes the file descriptor available for reuse on a subsequent open.

The most heavily used calls are undoubtedly read and write. We saw read earlier. Write has the same parameters.

Although most programs read and write files sequentially, for some applications programs need to be able to access any part of a file at random. Associated with each file is a pointer that indicates the current position in the file. When reading (writing) sequentially, it normally points to the next byte to be read (written). The lseek call changes the value of the position pointer, so that subsequent calls to read or write can begin anywhere in the file.

Lseek has three parameters: the first is the file descriptor for the file, the second is a file position, and the third tells whether the file position is relative to the beginning of the file, the current position, or the end of the file. The value returned by lseek is the absolute position in the file (in bytes) after changing the pointer.

For each file, UNIX keeps track of the file mode (regular file, special file, directory, and so on), size, time of last modification, and other information. Programs can ask to see this information via the stat system call. The first parameter specifies the file to be inspected; the second one is a pointer to a structure where the information is to be put. The fstat calls does the same thing for an open file.

1.6.3 System Calls for Directory Management

In this section we will look at some system calls that relate more to directories or the file system as a whole, rather than just to one specific file as in the previous section. The first two calls, mkdir and rmdir, create and remove empty directories, respectively. The next call is link. Its purpose is to allow the same file to appear under two or more names, often in different directories. A typical use is to allow several members of the same programming team to share a common file, with each of them having the file appear in his own directory, possibly under different names. Sharing a file is not the same as giving every team member a private copy; having a shared file means that changes that any member of the team makes are instantly visible to the other members—there is only one file. When copies are made of a file, subsequent changes made to one copy do not affect the others.

To see how link works, consider the situation of Fig. 1-21(a). Here are two users, *ast* and *jim*, each having his own directory with some files. If *ast* now executes a program containing the system call

```
link("/usr/jim/memo", "/usr/ast/note");
```

the file *memo* in *jim*'s directory is now entered into *ast*'s directory under the name *note*. Thereafter, */usr/jim/memo* and */usr/ast/note* refer to the same file. As an aside, whether user directories are kept in */usr*, */user*, */home*, or somewhere else is simply a decision made by the local system administrator.

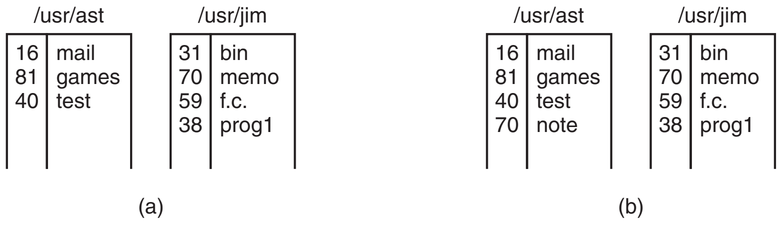


Figure 1-21. (a) Two directories before linking */usr/jim/memo* to *ast*'s directory. (b) The same directories after linking.

Understanding how link works will probably make it clearer what it does. Every file in UNIX has a unique number, its i-number, that identifies it. This i-number is an index into a table of **i-nodes**, one per file, telling who owns the file, where its disk blocks are, and so on. A directory is simply a file containing a set of (i-number, ASCII name) pairs. In the first versions of UNIX, each directory entry was 16 bytes—2 bytes for the i-number and 14 bytes for the name. Now a more complicated structure is needed to support long file names, but conceptually a directory is still a set of (i-number, ASCII name) pairs. In Fig. 1-21, *mail* has i-number 16, and so on. What link does is simply create a brand new directory entry with a (possibly new) name, using the i-number of an existing file. In Fig. 1-21(b), two entries have the same i-number (70) and thus refer to the same file. If either one is later removed, using the unlink system call, the other one remains. If both are removed, UNIX sees that no entries to the file exist (a field in the i-node keeps track of the number of directory entries pointing to the file), so the file is removed from the disk.

As we have mentioned earlier, the mount system call allows two file systems to be merged into one. A common situation is to have the root file system, containing the binary (executable) versions of the common commands and other heavily used files, on a hard disk (sub)partition and user files on another (sub)partition. Further, the user can then insert a USB disk with files to be read.

By executing the mount system call, the USB file system can be attached to the root file system, as shown in Fig. 1-22. A typical statement in C to mount is

```
mount("/dev/sdb0", "/mnt", 0);
```

where the first parameter is the name of a block special file for USB drive 0, the second parameter is the place in the tree where it is to be mounted, and the third parameter tells whether the file system is to be mounted read-write or read-only.

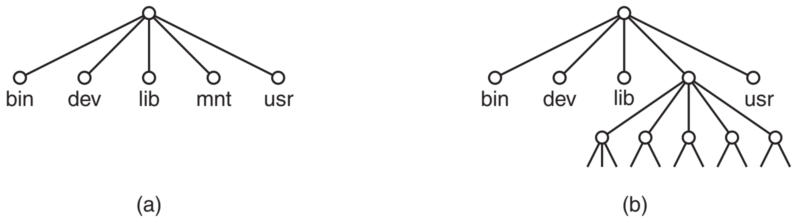


Figure 1-22. (a) File system before the mount. (b) File system after the mount.

After the mount call, a file on drive 0 can be accessed by just using its path from the root directory or the working directory, without regard to which drive it is on. In fact, second, third, and fourth drives can also be mounted anywhere in the tree. The mount call makes it possible to integrate removable media into a single integrated file hierarchy, without having to worry about which device a file is on. Although this example involves CD-ROMs, portions of hard disks (often called **partitions** or **minor devices**) can also be mounted this way, as well as external hard disks and USB sticks. When a file system is no longer needed, it can be unmounted with the umount system call.

1.6.4 Miscellaneous System Calls

A variety of other system calls exist as well. We will look at just four of them here. The chdir call changes the current working directory. After the call

```
chdir("/usr/ast/test");
```

an open on the file *xyz* will open */usr/ast/test/xyz*. The concept of a working directory eliminates the need for typing (long) absolute path names all the time.

In UNIX every file has a mode used for protection. The mode includes the read-write-execute bits for the owner, group, and others. The chmod system call makes it possible to change the mode of a file. For example, to make a file read-only by everyone except the owner, one could execute

```
chmod("file", 0644);
```

The kill system call is the way users and user processes send signals. If a process is prepared to catch a particular signal, then when it arrives, a signal handler is run. If the process is not prepared to handle a signal, then its arrival kills the process (hence the name of the call).

POSIX defines a number of procedures for dealing with time. For example, time just returns the current time in seconds, with 0 corresponding to Jan. 1, 1970 at midnight (just as the day was starting, not ending). On computers using 32-bit words, the maximum value time can return is $2^{32} - 1$ seconds (assuming an unsigned integer is used). This value corresponds to a little over 136 years. Thus in the year 2106, 32-bit UNIX systems will go berserk, not unlike the famous Y2K problem that would have wreaked havoc with the world's computers in 2000, were it not for the massive effort the IT industry put into fixing the problem. If you currently have a 32-bit UNIX system, you are advised to trade it in for a 64-bit one sometime before the year 2106.

1.6.5 The Windows Win32 API

So far we have focused primarily on UNIX. Now it is time to look briefly at Windows. Windows and UNIX differ in a fundamental way in their respective programming models. A UNIX program consists of code that does something or other, making system calls to have certain services performed. In contrast, a Windows program is normally event driven. The main program waits for some event to happen, then calls a procedure to handle it. Typical events are keys being struck, the mouse being moved, a mouse button being pushed, or a USB drive inserted. Handlers are then called to process the event, update the screen and update the internal program state. All in all, this leads to a somewhat different style of programming than with UNIX, but since the focus of this book is on operating system function and structure, these different programming models will not concern us much more.

Of course, Windows also has system calls. With UNIX, there is almost a one-to-one relationship between the system calls (e.g., read) and the library procedures (e.g., *read*) used to invoke the system calls. In other words, for each system call, there is roughly one library procedure that is called to invoke it, as indicated in Fig. 1-17. Furthermore, POSIX has only about 100 procedure calls.

With Windows, the situation is radically different. To start with, the library calls and the actual system calls are highly decoupled. Microsoft has defined a set of procedures called the **Win32 API** (**Application Programming Interface**) that programmers are expected to use to get operating system services. This interface is (partially) supported on all versions of Windows since Windows 95. By decoupling the API interface from the actual system calls, Microsoft retains the ability to change the actual system calls in time (even from release to release) without invalidating existing programs. What actually constitutes Win32 is also slightly ambiguous because recent versions of Windows have many new calls that were not previously available. In this section, Win32 means the interface supported by all versions of Windows. Win32 provides compatibility among versions of Windows.

The number of Win32 API calls is extremely large, numbering in the thousands. Furthermore, while many of them do invoke system calls, a substantial number are carried out entirely in user space. As a consequence, with Windows it is impossible to see what is a system call (i.e., performed by the kernel) and what is simply a user-space library call. In fact, what is a system call in one version of Windows may be done in user space in a different version, and vice versa. When we discuss the Windows system calls in this book, we will use the Win32 procedures (where appropriate) since Microsoft guarantees that these will be stable over time. But it is worth remembering that not all of them are true system calls (i.e., traps to the kernel).

The Win32 API has a huge number of calls for managing windows, geometric figures, text, fonts, scrollbars, dialog boxes, menus, and other features of the GUI. To the extent that the graphics subsystem runs in the kernel (true on some versions of Windows but not on all), these are system calls; otherwise they are just library calls. Should we discuss these calls in this book or not? Since they are not really related to the function of an operating system, we have decided not to, even though they may be carried out by the kernel. Readers interested in the Win32 API should consult one of the many books on the subject (e.g., Hart, 1997; Rector and Newcomer, 1997; and Simon, 1997).

Even introducing all the Win32 API calls here is out of the question, so we will restrict ourselves to those calls that roughly correspond to the functionality of the UNIX calls listed in Fig. 1-18. These are listed in Fig. 1-23.

Let us now briefly go through the list of Fig. 1-23. CreateProcess creates a new process. It does the combined work of fork and execve in UNIX. It has many parameters specifying the properties of the newly created process. Windows does not have a process hierarchy as UNIX does so there is no concept of a parent process and a child process. After a process is created, the creator and createe are equals. WaitForSingleObject is used to wait for an event. Many possible events can be waited for. If the parameter specifies a process, then the caller waits for the specified process to exit, which is done using ExitProcess.

The next six calls operate on files and are functionally similar to their UNIX counterparts although they differ in the parameters and details. Still, files can be opened, closed, read, and written pretty much as in UNIX. The SetFilePointer and GetFileAttributesEx calls set the file position and get some of the file attributes.

Windows has directories and they are created with CreateDirectory and RemoveDirectory API calls, respectively. There is also a notion of a current directory, set by SetCurrentDirectory. The current time of day is acquired using GetLocalTime.

The Win32 interface does not have links to files, mounted file systems, security, or signals, so the calls corresponding to the UNIX ones do not exist. Of course, Win32 has a huge number of other calls that UNIX does not have, especially for managing the GUI. Windows Vista has an elaborate security system and also supports file links. Windows 7 and 8 add yet more features and system calls.

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount, so no umount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Figure 1-23. The Win32 API calls that roughly correspond to the UNIX calls of Fig. 1-18. It is worth emphasizing that Windows has a very large number of other system calls, most of which do not correspond to anything in UNIX.

One last note about Win32 is perhaps worth making. Win32 is not a terribly uniform or consistent interface. The main culprit here was the need to be backward compatible with the previous 16-bit interface used in Windows 3.x.

1.7 OPERATING SYSTEM STRUCTURE

Now that we have seen what operating systems look like on the outside (i.e., the programmer's interface), it is time to take a look inside. In the following sections, we will examine six different structures that have been tried, in order to get some idea of the spectrum of possibilities. These are by no means exhaustive, but they give an idea of some designs that have been tried in practice. The six designs we will discuss here are monolithic systems, layered systems, microkernels, client-server systems, virtual machines, and exokernels.

1.7.1 Monolithic Systems

By far the most common organization, in the monolithic approach the entire operating system runs as a single program in kernel mode. The operating system is written as a collection of procedures, linked together into a single large executable binary program. When this technique is used, each procedure in the system is free to call any other one, if the latter provides some useful computation that the former needs. Being able to call any procedure you want is very efficient, but having thousands of procedures that can call each other without restriction may also lead to a system that is unwieldy and difficult to understand. Also, a crash in any of these procedures will take down the entire operating system.

To construct the actual object program of the operating system when this approach is used, one first compiles all the individual procedures (or the files containing the procedures) and then binds them all together into a single executable file using the system linker. In terms of information hiding, there is essentially none—every procedure is visible to every other procedure (as opposed to a structure containing modules or packages, in which much of the information is hidden away inside modules, and only the officially designated entry points can be called from outside the module).

Even in monolithic systems, however, it is possible to have some structure. The services (system calls) provided by the operating system are requested by putting the parameters in a well-defined place (e.g., on the stack) and then executing a trap instruction. This instruction switches the machine from user mode to kernel mode and transfers control to the operating system, shown as step 6 in Fig. 1-17. The operating system then fetches the parameters and determines which system call is to be carried out. After that, it indexes into a table that contains in slot *k* a pointer to the procedure that carries out system call *k* (step 7 in Fig. 1-17).

This organization suggests a basic structure for the operating system:

1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

In this model, for each system call there is one service procedure that takes care of it and executes it. The utility procedures do things that are needed by several service procedures, such as fetching data from user programs. This division of the procedures into three layers is shown in Fig. 1-24.

In addition to the core operating system that is loaded when the computer is booted, many operating systems support loadable extensions, such as I/O device drivers and file systems. These components are loaded on demand. In UNIX they are called **shared libraries**. In Windows they are called **DLLs** (**Dynamic-Link Libraries**). They have file extension *.dll* and the *C:\Windows\system32* directory on Windows systems has well over 1000 of them.

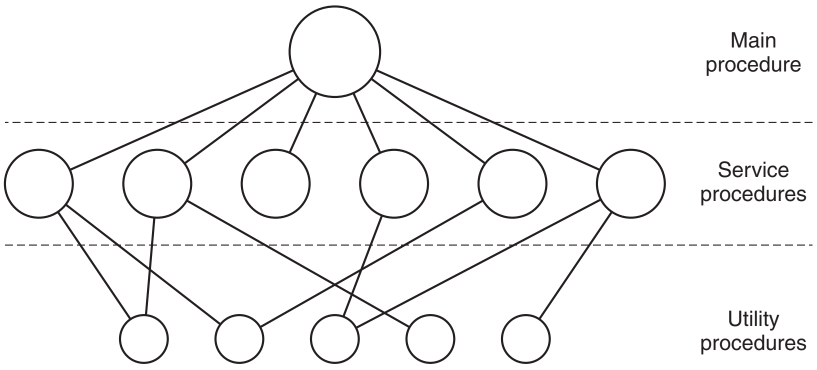


Figure 1-24. A simple structuring model for a monolithic system.

1.7.2 Layered Systems

A generalization of the approach of Fig. 1-24 is to organize the operating system as a hierarchy of layers, each one constructed upon the one below it. The first system constructed in this way was the THE system built at the Technische Hogeschool Eindhoven in the Netherlands by E. W. Dijkstra (1968) and his students. The THE system was a simple batch system for a Dutch computer, the Electrologica X8, which had 32K of 27-bit words (bits were expensive back then).

The system had six layers, as shown in Fig. 1-25. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Above layer 0, the system consisted of sequential processes, each of which could be programmed without having to worry about the fact that multiple processes were running on a single processor. In other words, layer 0 provided the basic multiprogramming of the CPU.

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure 1-25. Structure of the THE operating system.

Layer 1 did the memory management. It allocated space for processes in main memory and on a 512K word drum used for holding parts of processes (pages) for which there was no room in main memory. Above layer 1, processes did not have to worry about whether they were in memory or on the drum; the layer 1 software took care of making sure pages were brought into memory at the moment they were needed and removed when they were not needed.

Layer 2 handled communication between each process and the operator console (that is, the user). On top of this layer each process effectively had its own operator console. Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities. Layer 4 was where the user programs were found. They did not have to worry about process, memory, console, or I/O management. The system operator process was located in layer 5.

A further generalization of the layering concept was present in the MULTICS system. Instead of layers, MULTICS was described as having a series of concentric rings, with the inner ones being more privileged than the outer ones (which is effectively the same thing). When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to make the equivalent of a system call, that is, a TRAP instruction whose parameters were carefully checked for validity before the call was allowed to proceed. Although the entire operating system was part of the address space of each user process in MULTICS, the hardware made it possible to designate individual procedures (memory segments, actually) as protected against reading, writing, or executing.

Whereas the THE layering scheme was really only a design aid, because all the parts of the system were ultimately linked together into a single executable program, in MULTICS, the ring mechanism was very much present at run time and enforced by the hardware. The advantage of the ring mechanism is that it can easily be extended to structure user subsystems. For example, a professor could write a program to test and grade student programs and run this program in ring n, with the student programs running in ring $n + 1$ so that they could not change their grades.

1.7.3 Microkernels

With the layered approach, the designers have a choice where to draw the kernel-user boundary. Traditionally, all the layers went in the kernel, but that is not necessary. In fact, a strong case can be made for putting as little as possible in kernel mode because bugs in the kernel can bring down the system instantly. In contrast, user processes can be set up to have less power so that a bug there may not be fatal.

Various researchers have repeatedly studied the number of bugs per 1000 lines of code (e.g., Basilli and Perricone, 1984; and Ostrand and Weyuker, 2002). Bug density depends on module size, module age, and more, but a ballpark figure for serious industrial systems is between two and ten bugs per thousand lines of code. This means that a monolithic operating system of five million lines of code is likely to contain between 10,000 and 50,000 kernel bugs. Not all of these are fatal, of course, since some bugs may be things like issuing an incorrect error message in a situation that rarely occurs. Nevertheless, operating systems are sufficiently buggy that computer manufacturers put reset buttons on them (often on the front panel), something the manufacturers of TV sets, stereos, and cars do not do, despite the large amount of software in these devices.

The basic idea behind the microkernel design is to achieve high reliability by splitting the operating system up into small, well-defined modules, only one of which—the microkernel—runs in kernel mode and the rest run as relatively powerless ordinary user processes. In particular, by running each device driver and file system as a separate user process, a bug in one of these can crash that component, but cannot crash the entire system. Thus a bug in the audio driver will cause the sound to be garbled or stop, but will not crash the computer. In contrast, in a monolithic system with all the drivers in the kernel, a buggy audio driver can easily reference an invalid memory address and bring the system to a grinding halt instantly.

Many microkernels have been implemented and deployed for decades (Haertig et al., 1997; Heiser et al., 2006; Herder et al., 2006; Hildebrand, 1992; Kirsch et al., 2005; Liedtke, 1993, 1995, 1996; Pike et al., 1992; and Zuberi et al., 1999). With the exception of OS X, which is based on the Mach microkernel (Accetta et al., 1986), common desktop operating systems do not use microkernels. However, they are dominant in real-time, industrial, avionics, and military applications that are mission critical and have very high reliability requirements. A few of the better-known microkernels include Integrity, K42, L4, PikeOS, QNX, Symbian, and MINIX 3. We now give a brief overview of MINIX 3, which has taken the idea of modularity to the limit, breaking most of the operating system up into a number of independent user-mode processes. MINIX 3 is a POSIX-conformant, open source system freely available at *www.minix3.org* (Giuffrida et al., 2012; Giuffrida et al., 2013; Herder et al., 2006; Herder et al., 2009; and Hruby et al., 2013).

The MINIX 3 microkernel is only about 12,000 lines of C and some 1400 lines of assembler for very low-level functions such as catching interrupts and switching processes. The C code manages and schedules processes, handles interprocess communication (by passing messages between processes), and offers a set of about 40 kernel calls to allow the rest of the operating system to do its work. These calls perform functions like hooking handlers to interrupts, moving data between address spaces, and installing memory maps for new processes. The process structure of MINIX 3 is shown in Fig. 1-26, with the kernel call handlers labeled *Sys*. The device driver for the clock is also in the kernel because the scheduler interacts closely with it. The other device drivers run as separate user processes.

Outside the kernel, the system is structured as three layers of processes all running in user mode. The lowest layer contains the device drivers. Since they run in user mode, they do not have physical access to the I/O port space and cannot issue I/O commands directly. Instead, to program an I/O device, the driver builds a structure telling which values to write to which I/O ports and makes a kernel call telling

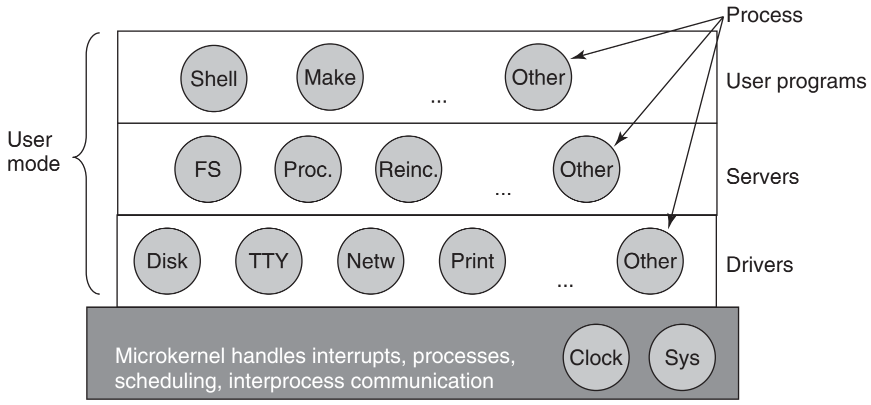


Figure 1-26. Simplified structure of the MINIX system.

the kernel to do the write. This approach means that the kernel can check to see that the driver is writing (or reading) from I/O it is authorized to use. Consequently (and unlike a monolithic design), a buggy audio driver cannot accidentally write on the disk.

Above the drivers is another user-mode layer containing the servers, which do most of the work of the operating system. One or more file servers manage the file system(s), the process manager creates, destroys, and manages processes, and so on. User programs obtain operating system services by sending short messages to the servers asking for the POSIX system calls. For example, a process needing to do a read sends a message to one of the file servers telling it what to read.

One interesting server is the **reincarnation server**, whose job is to check if the other servers and drivers are functioning correctly. In the event that a faulty one is detected, it is automatically replaced without any user intervention. In this way, the system is self healing and can achieve high reliability.

The system has many restrictions limiting the power of each process. As mentioned, drivers can touch only authorized I/O ports, but access to kernel calls is also controlled on a per-process basis, as is the ability to send messages to other processes. Processes can also grant limited permission for other processes to have the kernel access their address spaces. As an example, a file system can grant permission for the disk driver to let the kernel put a newly read-in disk block at a specific address within the file system's address space. The sum total of all these restrictions is that each driver and server has exactly the power to do its work and nothing more, thus greatly limiting the damage a buggy component can do.

An idea somewhat related to having a minimal kernel is to put the **mechanism** for doing something in the kernel but not the **policy**. To make this point better, consider the scheduling of processes. A relatively simple scheduling algorithm is to assign a numerical priority to every process and then have the kernel run the highest-priority process that is runnable. The mechanism—in the kernel—is to look for the highest-priority process and run it. The policy—assigning priorities to processes—can be done by user-mode processes. In this way, policy and mechanism can be decoupled and the kernel can be made smaller.

1.7.4 Client-Server Model

A slight variation of the microkernel idea is to distinguish two classes of processes, the **servers**, each of which provides some service, and the **clients**, which use these services. This model is known as the **client-server** model. Often the lowest layer is a microkernel, but that is not required. The essence is the presence of client processes and server processes.

Communication between clients and servers is often by message passing. To obtain a service, a client process constructs a message saying what it wants and sends it to the appropriate service. The service then does the work and sends back the answer. If the client and server happen to run on the same machine, certain optimizations are possible, but conceptually, we are still talking about message passing here.

An obvious generalization of this idea is to have the clients and servers run on different computers, connected by a local or wide-area network, as depicted in Fig. 1-27. Since clients communicate with servers by sending messages, the clients need not know whether the messages are handled locally on their own machines, or whether they are sent across a network to servers on a remote machine. As far as the client is concerned, the same thing happens in both cases: requests are sent and replies come back. Thus the client-server model is an abstraction that can be used for a single machine or for a network of machines.

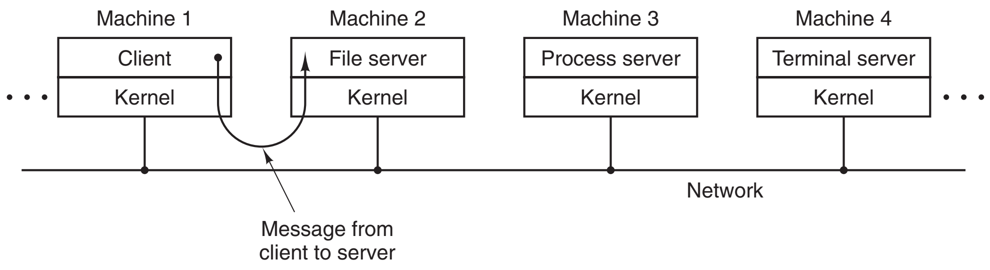


Figure 1-27. The client-server model over a network.

Increasingly many systems involve users at their home PCs as clients and large machines elsewhere running as servers. In fact, much of the Web operates this way. A PC sends a request for a Web page to the server and the Web page comes back. This is a typical use of the client-server model in a network.

1.7.5 Virtual Machines

The initial releases of OS/360 were strictly batch systems. Nevertheless, many 360 users wanted to be able to work interactively at a terminal, so various groups, both inside and outside IBM, decided to write timesharing systems for it. The official IBM timesharing system, TSS/360, was delivered late, and when it finally arrived it was so big and slow that few sites converted to it. It was eventually abandoned after its development had consumed some $50 million (Graham, 1970). But a group at IBM's Scientific Center in Cambridge, Massachusetts, produced a radically different system that IBM eventually accepted as a product. A linear descendant of it, called **z/VM**, is now widely used on IBM's current mainframes, the zSeries, which are heavily used in large corporate data centers, for example, as e-commerce servers that handle hundreds or thousands of transactions per second and use databases whose sizes run to millions of gigabytes.

VM/370

This system, originally called CP/CMS and later renamed VM/370 (Seawright and MacKinnon, 1979), was based on an astute observation: a timesharing system provides (1) multiprogramming and (2) an extended machine with a more convenient interface than the bare hardware. The essence of VM/370 is to completely separate these two functions.

The heart of the system, known as the **virtual machine monitor**, runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up, as shown in Fig. 1-28. However, unlike all other operating systems, these virtual machines are not extended machines, with files and other nice features. Instead, they are *exact* copies of the bare hardware, including kernel/user mode, I/O, interrupts, and everything else the real machine has.

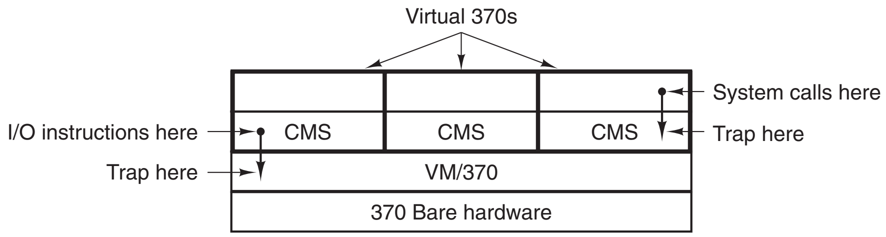


Figure 1-28. The structure of VM/370 with CMS.

Because each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the bare hardware. Different virtual machines can, and frequently do, run different operating systems. On the original IBM VM/370 system, some ran OS/360 or one of the other large batch or transaction-processing operating systems, while others ran a single-user, interactive system called **CMS** (**Conversational Monitor System**) for interactive timesharing users. The latter was popular with programmers.

When a CMS program executed a system call, the call was trapped to the operating system in its own virtual machine, not to VM/370, just as it would be were it running on a real machine instead of a virtual one. CMS then issued the normal hardware I/O instructions for reading its virtual disk or whatever was needed to carry out the call. These I/O instructions were trapped by VM/370, which then performed them as part of its simulation of the real hardware. By completely separating the functions of multiprogramming and providing an extended machine, each of the pieces could be much simpler, more flexible, and much easier to maintain.

In its modern incarnation, z/VM is usually used to run multiple complete operating systems rather than stripped-down single-user systems like CMS. For example, the zSeries is capable of running one or more Linux virtual machines along with traditional IBM operating systems.

Virtual Machines Rediscovered

While IBM has had a virtual-machine product available for four decades, and a few other companies, including Oracle and Hewlett-Packard, have recently added virtual-machine support to their high-end enterprise servers, the idea of virtualization has largely been ignored in the PC world until recently. But in the past few years, a combination of new needs, new software, and new technologies have combined to make it a hot topic.

First the needs. Many companies have traditionally run their mail servers, Web servers, FTP servers, and other servers on separate computers, sometimes with different operating systems. They see virtualization as a way to run them all on the same machine without having a crash of one server bring down the rest.

Virtualization is also popular in the Web hosting world. Without virtualization, Web hosting customers are forced to choose between **shared hosting** (which just gives them a login account on a Web server, but no control over the server software) and dedicated hosting (which gives them their own machine, which is very flexible but not cost effective for small to medium Websites). When a Web hosting company offers virtual machines for rent, a single physical machine can run many virtual machines, each of which appears to be a complete machine. Customers who rent a virtual machine can run whatever operating system and software they want to, but at a fraction of the cost of a dedicated server (because the same physical machine supports many virtual machines at the same time).

Another use of virtualization is for end users who want to be able to run two or more operating systems at the same time, say Windows and Linux, because some of their favorite application packages run on one and some run on the other. This situation is illustrated in Fig. 1-29(a), where the term "virtual machine monitor" has been renamed **type 1 hypervisor**, which is commonly used nowadays because "virtual machine monitor" requires more keystrokes than people are prepared to put up with now. Note that many authors use the terms interchangeably though.

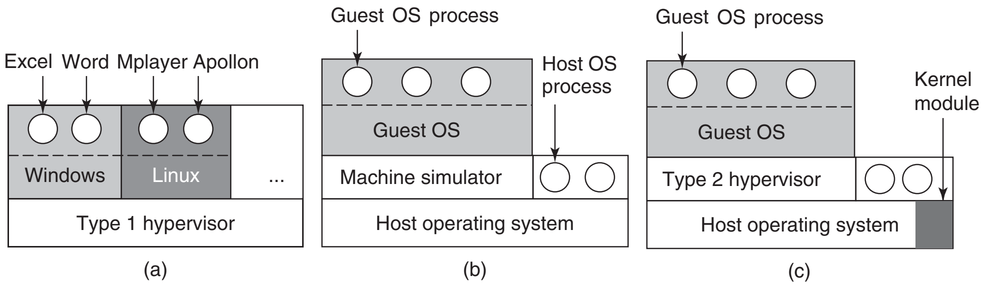


Figure 1-29. (a) A type 1 hypervisor. (b) A pure type 2 hypervisor. (c) A practical type 2 hypervisor.

While no one disputes the attractiveness of virtual machines today, the problem then was implementation. In order to run virtual machine software on a computer, its CPU must be virtualizable (Popek and Goldberg, 1974). In a nutshell, here is the problem. When an operating system running on a virtual machine (in user mode) executes a privileged instruction, such as modifying the PSW or doing I/O, it is essential that the hardware trap to the virtual-machine monitor so the instruction can be emulated in software. On some CPUs—notably the Pentium, its predecessors, and its clones—attempts to execute privileged instructions in user mode are just ignored. This property made it impossible to have virtual machines on this hardware, which explains the lack of interest in the x86 world. Of course, there were interpreters for the Pentium, such as *Bochs*, that ran on the Pentium, but with a performance loss of one to two orders of magnitude, they were not useful for serious work.

This situation changed as a result of several academic research projects in the 1990s and early years of this millennium, notably Disco at Stanford (Bugnion et al., 1997) and Xen at Cambridge University (Barham et al., 2003). These research papers led to several commercial products (e.g., VMware Workstation and Xen) and a revival of interest in virtual machines. Besides VMware and Xen, popular hypervisors today include KVM (for the Linux kernel), VirtualBox (by Oracle), and Hyper-V (by Microsoft).

Some of these early research projects improved the performance over interpreters like *Bochs* by translating blocks of code on the fly, storing them in an internal cache, and then reusing them if they were executed again. This improved the performance considerably, and led to what we will call **machine simulators**, as shown in Fig. 1-29(b). However, although this technique, known as **binary translation**, helped improve matters, the resulting systems, while good enough to publish papers about in academic conferences, were still not fast enough to use in commercial environments where performance matters a lot.

The next step in improving performance was to add a kernel module to do some of the heavy lifting, as shown in Fig. 1-29(c). In practice now, all commercially available hypervisors, such as VMware Workstation, use this hybrid strategy (and have many other improvements as well). They are called **type 2 hypervisors** by everyone, so we will (somewhat grudgingly) go along and use this name in the rest of this book, even though we would prefer to called them type 1.7 hypervisors to reflect the fact that they are not entirely user-mode programs. In Chap. 7, we will describe in detail how VMware Workstation works and what the various pieces do.

In practice, the real distinction between a type 1 hypervisor and a type 2 hypervisor is that a type 2 makes uses of a **host operating system** and its file system to create processes, store files, and so on. A type 1 hypervisor has no underlying support and must perform all these functions itself.

After a type 2 hypervisor is started, it reads the installation CD-ROM (or CD-ROM image file) for the chosen **guest operating system** and installs the guest OS on a virtual disk, which is just a big file in the host operating system's file system. Type 1 hypervisors cannot do this because there is no host operating system to store files on. They must manage their own storage on a raw disk partition.

When the guest operating system is booted, it does the same thing it does on the actual hardware, typically starting up some background processes and then a GUI. To the user, the guest operating system behaves the same way it does when running on the bare metal even though that is not the case here.

A different approach to handling control instructions is to modify the operating system to remove them. This approach is not true virtualization, but **paravirtualization**. We will discuss virtualization in more detail in Chap. 7.

The Java Virtual Machine

Another area where virtual machines are used, but in a somewhat different way, is for running Java programs. When Sun Microsystems invented the Java programming language, it also invented a virtual machine (i.e., a computer architecture) called the **JVM** (**Java Virtual Machine**). The Java compiler produces code for JVM, which then typically is executed by a software JVM interpreter. The advantage of this approach is that the JVM code can be shipped over the Internet to any computer that has a JVM interpreter and run there. If the compiler had produced SPARC or x86 binary programs, for example, they could not have been shipped and run anywhere as easily. (Of course, Sun could have produced a compiler that produced SPARC binaries and then distributed a SPARC interpreter, but JVM is a much simpler architecture to interpret.) Another advantage of using JVM is that if the interpreter is implemented properly, which is not completely trivial, incoming JVM programs can be checked for safety and then executed in a protected environment so they cannot steal data or do any damage.

1.7.6 Exokernels

Rather than cloning the actual machine, as is done with virtual machines, another strategy is partitioning it, in other words, giving each user a subset of the resources. Thus one virtual machine might get disk blocks 0 to 1023, the next one might get blocks 1024 to 2047, and so on.

At the bottom layer, running in kernel mode, is a program called the **exokernel** (Engler et al., 1995). Its job is to allocate resources to virtual machines and then check attempts to use them to make sure no machine is trying to use somebody else's resources. Each user-level virtual machine can run its own operating system, as on VM/370 and the Pentium virtual 8086s, except that each one is restricted to using only the resources it has asked for and been allocated.

The advantage of the exokernel scheme is that it saves a layer of mapping. In the other designs, each virtual machine thinks it has its own disk, with blocks running from 0 to some maximum, so the virtual machine monitor must maintain tables to remap disk addresses (and all other resources). With the exokernel, this remapping is not needed. The exokernel need only keep track of which virtual machine has been assigned which resource. This method still has the advantage of separating the multiprogramming (in the exokernel) from the user operating system code (in user space), but with less overhead, since all the exokernel has to do is keep the virtual machines out of each other's hair.

1.8 THE WORLD ACCORDING TO C

Operating systems are normally large C (or sometimes C++) programs consisting of many pieces written by many programmers. The environment used for developing operating systems is very different from what individuals (such as students) are used to when writing small Java programs. This section is an attempt to give a very brief introduction to the world of writing an operating system for small-time Java or Python programmers.

1.8.1 The C Language

This is not a guide to C, but a short summary of some of the key differences between C and languages like **Python** and especially Java. Java is based on C, so there are many similarities between the two. Python is somewhat different, but still fairly similar. For convenience, we focus on Java. Java, Python, and C are all imperative languages with data types, variables, and control statements, for example. The primitive data types in C are integers (including short and long ones), characters, and floating-point numbers. Composite data types can be constructed using arrays, structures, and unions. The control statements in C are similar to those in Java, including if, switch, for, and while statements. Functions and parameters are roughly the same in both languages.

One feature C has that Java and Python do not is explicit pointers. A **pointer** is a variable that points to (i.e., contains the address of) a variable or data structure. Consider the statements

```
char c1, c2, *p;
c1 = 'c';
p = &c1;
c2 = *p;
```

which declare *c1* and *c2* to be character variables and *p* to be a variable that points to (i.e., contains the address of) a character. The first assignment stores the ASCII code for the character "c" in the variable *c1*. The second one assigns the address of *c1* to the pointer variable *p*. The third one assigns the contents of the variable pointed to by *p* to the variable *c2*, so after these statements are executed, *c2* also contains the ASCII code for "c". In theory, pointers are typed, so you are not supposed to assign the address of a floating-point number to a character pointer, but in practice compilers accept such assignments, albeit sometimes with a warning. Pointers are a very powerful construct, but also a great source of errors when used carelessly.

Some things that C does not have include built-in strings, threads, packages, classes, objects, type safety, and garbage collection. The last one is a show stopper for operating systems. All storage in C is either static or explicitly allocated and released by the programmer, usually with the library functions *malloc* and *free*. It is the latter property—total programmer control over memory—along with explicit pointers that makes C attractive for writing operating systems. Operating systems are basically real-time systems to some extent, even general-purpose ones. When an interrupt occurs, the operating system may have only a few microseconds to perform some action or lose critical information. Having the garbage collector kick in at an arbitrary moment is intolerable.

1.8.2 Header Files

An operating system project generally consists of some number of directories, each containing many *.c* files containing the code for some part of the system, along with some *.h* header files that contain declarations and definitions used by one or more code files. Header files can also include simple **macros**, such as

```
#define BUFFER_SIZE 4096
```

which allows the programmer to name constants, so that when *BUFFER_SIZE* is used in the code, it is replaced during compilation by the number 4096. Good C programming practice is to name every constant except 0, 1, and −1, and sometimes even them. Macros can have parameters, such as

```
#define max(a, b) (a > b ? a : b)
```

which allows the programmer to write

```
i = max(j, k+1)
```

and get

```
i = (j > k+1 ? j : k+1)
```

to store the larger of *j* and *k+1* in *i*. Headers can also contain conditional compilation, for example

```
#ifdef X86
intel_int_ack();
#endif
```

which compiles into a call to the function *intel_int_ack* if the macro *X86* is defined and nothing otherwise. Conditional compilation is heavily used to isolate architecture-dependent code so that certain code is inserted only when the system is compiled on the X86, other code is inserted only when the system is compiled on a SPARC, and so on. A *.c* file can bodily include zero or more header files using the *#include* directive. There are also many header files that are common to nearly every *.c* and are stored in a central directory.

1.8.3 Large Programming Projects

To build the operating system, each *.c* is compiled into an **object file** by the C compiler. Object files, which have the suffix *.o*, contain binary instructions for the target machine. They will later be directly executed by the CPU. There is nothing like Java byte code or Python byte code in the C world.

The first pass of the C compiler is called the **C preprocessor**. As it reads each *.c* file, every time it hits a *#include* directive, it goes and gets the header file named in it and processes it, expanding macros, handling conditional compilation (and certain other things) and passing the results to the next pass of the compiler as if they were physically included.

Since operating systems are very large (five million lines of code is not unusual), having to recompile the entire thing every time one file is changed would be unbearable. On the other hand, changing a key header file that is included in thousands of other files does require recompiling those files. Keeping track of which object files depend on which header files is completely unmanageable without help.

Fortunately, computers are very good at precisely this sort of thing. On UNIX systems, there is a program called *make* (with numerous variants such as *gmake*, *pmake*, etc.) that reads the *Makefile*, which tells it which files are dependent on which other files. What *make* does is see which object files are needed to build the operating system binary and for each one, check to see if any of the files it depends on (the code and headers) have been modified subsequent to the last time the object file was created. If so, that object file has to be recompiled. When *make* has determined which *.c* files have to recompiled, it then invokes the C compiler to recompile them, thus reducing the number of compilations to the bare minimum. In large projects, creating the *Makefile* is error prone, so there are tools that do it automatically.

Once all the *.o* files are ready, they are passed to a program called the **linker** to combine all of them into a single executable binary file. Any library functions called are also included at this point, interfunction references are resolved, and machine addresses are relocated as need be. When the linker is finished, the result is an executable program, traditionally called *a.out* on UNIX systems. The various components of this process are illustrated in Fig. 1-30 for a program with three C files and two header files. Although we have been discussing operating system development here, all of this applies to developing any large program.

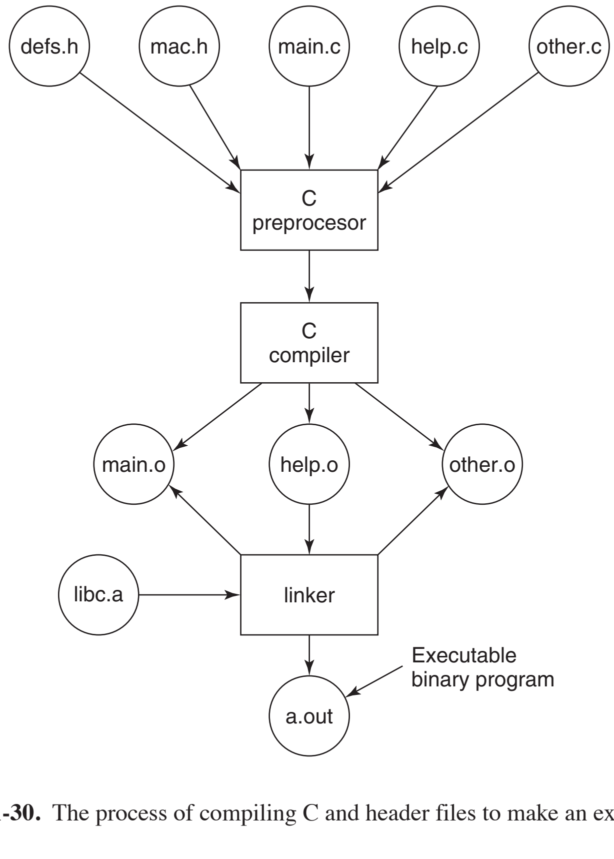


Figure 1-30. The process of compiling C and header files to make an executable.

1.8.4 The Model of Run Time

Once the operating system binary has been linked, the computer can be rebooted and the new operating system started. Once running, it may dynamically load pieces that were not statically included in the binary such as device drivers and file systems. At run time the operating system may consist of multiple segments, for the text (the program code), the data, and the stack. The text segment is normally immutable, not changing during execution. The data segment starts out at a certain size and initialized with certain values, but it can change and grow as need be. The stack is initially empty but grows and shrinks as functions are called and returned from. Often the text segment is placed near the bottom of memory, the data segment just above it, with the ability to grow upward, and the stack segment at a high virtual address, with the ability to grow downward, but different systems work differently.

In all cases, the operating system code is directly executed by the hardware, with no interpreter and no just-in-time compilation, as is normal with Java.

1.9 RESEARCH ON OPERATING SYSTEMS

Computer science is a rapidly advancing field and it is hard to predict where it is going. Researchers at universities and industrial research labs are constantly thinking up new ideas, some of which go nowhere but some of which become the cornerstone of future products and have massive impact on the industry and users. Telling which is which turns out to be easier to do in hindsight than in real time. Separating the wheat from the chaff is especially difficult because it often takes 20 to 30 years from idea to impact.

For example, when President Eisenhower set up the Dept. of Defense's Advanced Research Projects Agency (ARPA) in 1958, he was trying to keep the Army from killing the Navy and the Air Force over the Pentagon's research budget. He was not trying to invent the Internet. But one of the things ARPA did was fund some university research on the then-obscure concept of packet switching, which led to the first experimental packet-switched network, the ARPANET. It went live in 1969. Before long, other ARPA-funded research networks were connected to the ARPANET, and the Internet was born. The Internet was then happily used by academic researchers for sending email to each other for 20 years. In the early 1990s, Tim Berners-Lee invented the World Wide Web at the CERN research lab in Geneva and Marc Andreesen wrote a graphical browser for it at the University of Illinois. All of a sudden the Internet was full of twittering teenagers. President Eisenhower is probably rolling over in his grave.

Research in operating systems has also led to dramatic changes in practical systems. As we discussed earlier, the first commercial computer systems were all batch systems, until M.I.T. invented general-purpose timesharing in the early 1960s. Computers were all text-based until Doug Engelbart invented the mouse and the graphical user interface at Stanford Research Institute in the late 1960s. Who knows what will come next?

In this section and in comparable sections throughout the book, we will take a brief look at some of the research in operating systems that has taken place during the past 5 to 10 years, just to give a flavor of what might be on the horizon. This introduction is certainly not comprehensive. It is based largely on papers that have been published in the top research conferences because these ideas have at least survived a rigorous peer review process in order to get published. Note that in computer science—in contrast to other scientific fields—most research is published in conferences, not in journals. Most of the papers cited in the research sections were published by either ACM, the IEEE Computer Society, or USENIX and are available over the Internet to (student) members of these organizations. For more information about these organizations and their digital libraries, see

ACM	http://www.acm.org
IEEE Computer Society	http://www.computer.org
USENIX	http://www.usenix.org

Virtually all operating systems researchers realize that current operating systems are massive, inflexible, unreliable, insecure, and loaded with bugs, certain ones more than others (*names withheld here to protect the guilty*). Consequently, there is a lot of research on how to build better operating systems. Work has recently been published about bugs and debugging (Renzelmann et al., 2012; and Zhou et al., 2012), crash recovery (Correia et al., 2012; Ma et al., 2013; Ongaro et al., 2011; and Yeh and Cheng, 2012), energy management (Pathak et al., 2012; Petrucci and Loques, 2012; and Shen et al., 2013), file and storage systems (Elnably and Wang, 2012; Nightingale et al., 2012; and Zhang et al., 2013a), high-performance I/O (De Bruijn et al., 2011; Li et al., 2013a; and Rizzo, 2012), hyperthreading and multithreading (Liu et al., 2011), live update (Giuffrida et al., 2013), managing GPUs (Rossbach et al., 2011), memory management (Jantz et al., 2013; and Jeong et al., 2013), multicore operating systems (Baumann et al., 2009; Kapritsos, 2012; Lachaize et al., 2012; and Wentzlaff et al., 2012), operating system correctness (Elphinstone et al., 2007; Yang et al., 2006; and Klein et al., 2009), operating system reliability (Hruby et al., 2012; Ryzhyk et al., 2009, 2011 and Zheng et al., 2012), privacy and security (Dunn et al., 2012; Giuffrida et al., 2012; Li et al., 2013b; Lorch et al., 2013; Ortolani and Crispo, 2012; Slowinska et al., 2012; and Ur et al., 2012), usage and performance monitoring (Harter et. al, 2012; and Ravindranath et al., 2012), and virtualization (Agesen et al., 2012; Ben-Yehuda et al., 2010; Colp et al., 2011; Dai et al., 2013; Tarasov et al., 2013; and Williams et al., 2012) among many other topics.

1.10 OUTLINE OF THE REST OF THIS BOOK

We have now completed our introduction and bird's-eye view of the operating system. It is time to get down to the details. As mentioned already, from the programmer's point of view, the primary purpose of an operating system is to provide some key abstractions, the most important of which are processes and threads, address spaces, and files. Accordingly the next three chapters are devoted to these critical topics.

Chapter 2 is about processes and threads. It discusses their properties and how they communicate with one another. It also gives a number of detailed examples of how interprocess communication works and how to avoid some of the pitfalls.

In Chap. 3 we will study address spaces and their adjunct, memory management, in detail. The important topic of virtual memory will be examined, along with closely related concepts such as paging and segmentation.

Then, in Chap. 4, we come to the all-important topic of file systems. To a considerable extent, what the user sees is largely the file system. We will look at both the file-system interface and the file-system implementation.

Input/Output is covered in Chap. 5. The concepts of device independence and device dependence will be looked at. Several important devices, including disks, keyboards, and displays, will be used as examples.

Chapter 6 is about deadlocks. We briefly showed what deadlocks are in this chapter, but there is much more to say. Ways to prevent or avoid them are discussed.

At this point we will have completed our study of the basic principles of single-CPU operating systems. However, there is more to say, especially about advanced topics. In Chap. 7, we examine virtualization. We discuss both the principles, and some of the existing virtualization solutions in detail. Since virtualization is heavily used in cloud computing, we will also gaze at existing clouds. Another advanced topic is multiprocessor systems, including multicores, parallel computers, and distributed systems. These subjects are covered in Chap. 8.

A hugely important subject is operating system security, which is covered in Chap 9. Among the topics discussed in this chapter are threats (e.g., viruses and worms), protection mechanisms, and security models.

Next we have some case studies of real operating systems. These are UNIX, Linux, and Android (Chap. 10), and Windows 8 (Chap. 11). The text concludes with some wisdom and thoughts about operating system design in Chap. 12.

1.11 METRIC UNITS

To avoid any confusion, it is worth stating explicitly that in this book, as in computer science in general, metric units are used instead of traditional English units (the furlong-stone-fortnight system). The principal metric prefixes are listed in Fig. 1-31. The prefixes are typically abbreviated by their first letters, with the units greater than 1 capitalized. Thus a 1-TB database occupies 10^{12} bytes of storage and a 100-psec (or 100-ps) clock ticks every 10^{-10} seconds. Since milli and micro both begin with the letter ‘‘m,’’ a choice had to be made. Normally, ‘‘m’’ is for milli and ‘‘μ’’ (the Greek letter mu) is for micro.

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta

Figure 1-31. The principal metric prefixes.

It is also worth pointing out that, in common industry practice, the units for measuring memory sizes have slightly different meanings. There kilo means 2^{10} (1024) rather than 10^3 (1000) because memories are always a power of two. Thus a 1-KB memory contains 1024 bytes, not 1000 bytes. Similarly, a 1-MB memory contains 2^{20} (1,048,576) bytes and a 1-GB memory contains 2^{30} (1,073,741,824) bytes. However, a 1-Kbps communication line transmits 1000 bits per second and a 10-Mbps LAN runs at 10,000,000 bits/sec because these speeds are not powers of two. Unfortunately, many people tend to mix up these two systems, especially for disk sizes. To avoid ambiguity, in this book, we will use the symbols KB, MB, and GB for 2^{10}, 2^{20}, and 2^{30} bytes respectively, and the symbols Kbps, Mbps, and Gbps for 10^3, 10^6, and 10^9 bits/sec, respectively.

1.12 SUMMARY

Operating systems can be viewed from two viewpoints: resource managers and extended machines. In the resource-manager view, the operating system's job is to manage the different parts of the system efficiently. In the extended-machine view, the job of the system is to provide the users with abstractions that are more convenient to use than the actual machine. These include processes, address spaces, and files.

Operating systems have a long history, starting from the days when they replaced the operator, to modern multiprogramming systems. Highlights include early batch systems, multiprogramming systems, and personal computer systems.

Since operating systems interact closely with the hardware, some knowledge of computer hardware is useful to understanding them. Computers are built up of processors, memories, and I/O devices. These parts are connected by buses.

The basic concepts on which all operating systems are built are processes, memory management, I/O management, the file system, and security. Each of these will be treated in a subsequent chapter.

The heart of any operating system is the set of system calls that it can handle. These tell what the operating system really does. For UNIX, we have looked at four groups of system calls. The first group of system calls relates to process creation and termination. The second group is for reading and writing files. The third group is for directory management. The fourth group contains miscellaneous calls.

Operating systems can be structured in several ways. The most common ones are as a monolithic system, a hierarchy of layers, microkernel, client-server, virtual machine, or exokernel.

PROBLEMS

1. What are the two main functions of an operating system?

2. In Section 1.4, nine different types of operating systems are described. Give a list of applications for each of these systems (one per operating systems type).

3. What is the difference between timesharing and multiprogramming systems?

4. To use cache memory, main memory is divided into cache lines, typically 32 or 64 bytes long. An entire cache line is cached at once. What is the advantage of caching an entire line instead of a single byte or word at a time?

5. On early computers, every byte of data read or written was handled by the CPU (i.e., there was no DMA). What implications does this have for multiprogramming?

6. Instructions related to accessing I/O devices are typically privileged instructions, that is, they can be executed in kernel mode but not in user mode. Give a reason why these instructions are privileged.

7. The family-of-computers idea was introduced in the 1960s with the IBM System/360 mainframes. Is this idea now dead as a doornail or does it live on?

8. One reason GUIs were initially slow to be adopted was the cost of the hardware needed to support them. How much video RAM is needed to support a 25-line $\times$ 80-row character monochrome text screen? How much for a 1200 $\times$ 900-pixel 24-bit color bitmap? What was the cost of this RAM at 1980 prices ($5/KB)? How much is it now?

9. There are several design goals in building an operating system, for example, resource utilization, timeliness, robustness, and so on. Give an example of two design goals that may contradict one another.

10. What is the difference between kernel and user mode? Explain how having two distinct modes aids in designing an operating system.

11. A 255-GB disk has 65,536 cylinders with 255 sectors per track and 512 bytes per sector. How many platters and heads does this disk have? Assuming an average cylinder seek time of 11 ms, average rotational delay of 7 msec and reading rate of 100 MB/sec, calculate the average time it will take to read 400 KB from one sector.

12. Which of the following instructions should be allowed only in kernel mode?

(a) Disable all interrupts.
(b) Read the time-of-day clock.
(c) Set the time-of-day clock.
(d) Change the memory map.

13. Consider a system that has two CPUs, each CPU having two threads (hyperthreading). Suppose three programs, *P0*, *P1*, and *P2*, are started with run times of 5, 10 and 20 msec, respectively. How long will it take to complete the execution of these programs? Assume that all three programs are 100% CPU bound, do not block during execution, and do not change CPUs once assigned.

14. A computer has a pipeline with four stages. Each stage takes the same time to do its work, namely, 1 nsec. How many instructions per second can this machine execute?

15. Consider a computer system that has cache memory, main memory (RAM) and disk, and an operating system that uses virtual memory. It takes 1 nsec to access a word from the cache, 10 nsec to access a word from the RAM, and 10 ms to access a word from the disk. If the cache hit rate is 95% and main memory hit rate (after a cache miss) is 99%, what is the average time to access a word?

16. When a user program makes a system call to read or write a disk file, it provides an indication of which file it wants, a pointer to the data buffer, and the count. Control is then transferred to the operating system, which calls the appropriate driver. Suppose that the driver starts the disk and terminates until an interrupt occurs. In the case of reading from the disk, obviously the caller will have to be blocked (because there are no data for it). What about the case of writing to the disk? Need the caller be blocked awaiting completion of the disk transfer?

17. What is a trap instruction? Explain its use in operating systems.

18. Why is the process table needed in a timesharing system? Is it also needed in personal computer systems running UNIX or Windows with a single user?

19. Is there any reason why you might want to mount a file system on a nonempty directory? If so, what is it?

20. For each of the following system calls, give a condition that causes it to fail: `fork`, `exec`, and `unlink`.

21. What type of multiplexing (time, space, or both) can be used for sharing the following resources: CPU, memory, disk, network card, printer, keyboard, and display?

22. Can the

```
count = write(fd, buffer, nbytes);
```

call return any value in *count* other than *nbytes*? If so, why?

23. A file whose file descriptor is *fd* contains the following sequence of bytes: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. The following system calls are made:

```
lseek(fd, 3, SEEK_SET);
read(fd, &buffer, 4);
```

where the lseek call makes a seek to byte 3 of the file. What does *buffer* contain after the read has completed?

24. Suppose that a 10-MB file is stored on a disk on the same track (track 50) in consecutive sectors. The disk arm is currently situated over track number 100. How long will it take to retrieve this file from the disk? Assume that it takes about 1 ms to move the arm from one cylinder to the next and about 5 ms for the sector where the beginning of the file is stored to rotate under the head. Also, assume that reading occurs at a rate of 200 MB/s.

25. What is the essential difference between a block special file and a character special file?

26. In the example given in Fig. 1-17, the library procedure is called *read* and the system call itself is called read. Is it essential that both of these have the same name? If not, which one is more important?

27. Modern operating systems decouple a process address space from the machine's physical memory. List two advantages of this design.

28. To a programmer, a system call looks like any other call to a library procedure. Is it important that a programmer know which library procedures result in system calls? Under what circumstances and why?

29. Figure 1-23 shows that a number of UNIX system calls have no Win32 API equivalents. For each of the calls listed as having no Win32 equivalent, what are the consequences for a programmer of converting a UNIX program to run under Windows?

30. A portable operating system is one that can be ported from one system architecture to another without any modification. Explain why it is infeasible to build an operating system that is completely portable. Describe two high-level layers that you will have in designing an operating system that is highly portable.

31. Explain how separation of policy and mechanism aids in building microkernel-based operating systems.

32. Virtual machines have become very popular for a variety of reasons. Nevertheless, they have some downsides. Name one.

33. Here are some questions for practicing unit conversions:

(a) How long is a nanoyear in seconds?
(b) Micrometers are often called microns. How long is a megamicron?
(c) How many bytes are there in a 1-PB memory?
(d) The mass of the earth is 6000 yottagrams. What is that in kilograms?

34. Write a shell that is similar to Fig. 1-19 but contains enough code that it actually works so you can test it. You might also add some features such as redirection of input and output, pipes, and background jobs.

35. If you have a personal UNIX-like system (Linux, MINIX 3, FreeBSD, etc.) available that you can safely crash and reboot, write a shell script that attempts to create an unlimited number of child processes and observe what happens. Before running the experiment, type sync to the shell to flush the file system buffers to disk to avoid ruining the file system. You can also do the experiment safely in a virtual machine. **Note**: Do not try this on a shared system without first getting permission from the system administrator. The consequences will be instantly obvious so you are likely to be caught and sanctions may follow.

36. Examine and try to interpret the contents of a UNIX-like or Windows directory with a tool like the UNIX *od* program. (*Hint*: How you do this will depend upon what the OS allows. One trick that may work is to create a directory on a USB stick with one operating system and then read the raw device data using a different operating system that allows such access.)

2

PROCESSES AND THREADS

We are now about to embark on a detailed study of how operating systems are designed and constructed. The most central concept in any operating system is the *process*: an abstraction of a running program. Everything else hinges on this concept, and the operating system designer (and student) should have a thorough understanding of what a process is as early as possible.

Processes are one of the oldest and most important abstractions that operating systems provide. They support the ability to have (pseudo) concurrent operation even when there is only one CPU available. They turn a single CPU into multiple virtual CPUs. Without the process abstraction, modern computing could not exist. In this chapter we will go into considerable detail about processes and their first cousins, threads.

2.1 PROCESSES

All modern computers often do several things at the same time. People used to working with computers may not be fully aware of this fact, so a few examples may make the point clearer. First consider a Web server. Requests come in from all over asking for Web pages. When a request comes in, the server checks to see if the page needed is in the cache. If it is, it is sent back; if it is not, a disk request is started to fetch it. However, from the CPU's perspective, disk requests take eternity. While waiting for a disk request to complete, many more requests may come in. If there are multiple disks present, some or all of the newer ones may be fired off to other disks long before the first request is satisfied. Clearly some way is needed to model and control this concurrency. Processes (and especially threads) can help here.

Now consider a user PC. When the system is booted, many processes are secretly started, often unknown to the user. For example, a process may be started up to wait for incoming email. Another process may run on behalf of the antivirus program to check periodically if any new virus definitions are available. In addition, explicit user processes may be running, printing files and backing up the user's photos on a USB stick, all while the user is surfing the Web. All this activity has to be managed, and a multiprogramming system supporting multiple processes comes in very handy here.

In any multiprogramming system, the CPU switches from process to process quickly, running each for tens or hundreds of milliseconds. While, strictly speaking, at any one instant the CPU is running only one process, in the course of 1 second it may work on several of them, giving the illusion of parallelism. Sometimes people speak of **pseudoparallelism** in this context, to contrast it with the true hardware parallelism of **multiprocessor** systems (which have two or more CPUs sharing the same physical memory). Keeping track of multiple, parallel activities is hard for people to do. Therefore, operating system designers over the years have evolved a conceptual model (sequential processes) that makes parallelism easier to deal with. That model, its uses, and some of its consequences form the subject of this chapter.

2.1.1 The Process Model

In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of **sequential processes**, or just **processes** for short. A process is just an instance of an executing program, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel than to try to keep track of how the CPU switches from program to program. This rapid switching back and forth is called **multiprogramming**, as we saw in Chap. 1.

In Fig. 2-1(a) we see a computer multiprogramming four programs in memory. In Fig. 2-1(b) we see four processes, each with its own flow of control (i.e., its own logical program counter), and each one running independently of the other ones. Of course, there is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter. When it is finished (for the time being), the physical program counter is saved in the process' stored logical program counter in memory. In Fig. 2-1(c) we see that, viewed over a long enough time interval, all the processes have made progress, but at any given instant only one process is actually running.

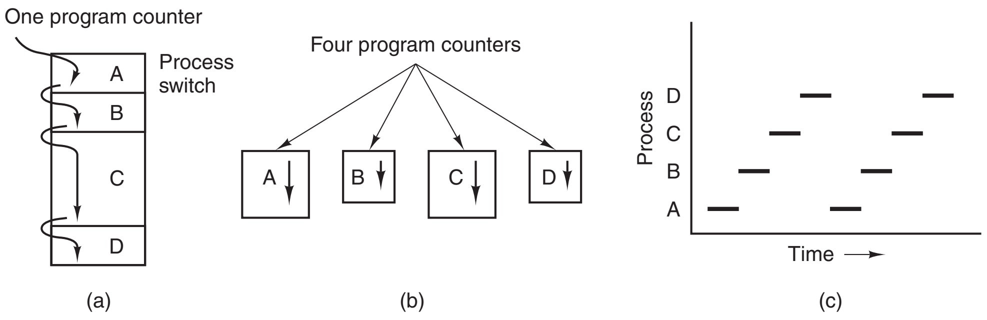


Figure 2-1. (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

In this chapter, we will assume there is only one CPU. Increasingly, however, that assumption is not true, since new chips are often multicore, with two, four, or more cores. We will look at multicore chips and multiprocessors in general in Chap. 8, but for the time being, it is simpler just to think of one CPU at a time. So when we say that a CPU can really run only one process at a time, if there are two cores (or CPUs) each of them can run only one process at a time.

With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform and probably not even reproducible if the same processes are run again. Thus, processes must not be programmed with built-in assumptions about timing. Consider, for example, an audio process that plays music to accompany a high-quality video run by another device. Because the audio should start a little later than the video, it signals the video server to start playing, and then runs an idle loop 10,000 times before playing back the audio. All goes well, if the loop is a reliable timer, but if the CPU decides to switch to another process during the idle loop, the audio process may not run again until the corresponding video frames have already come and gone, and the video and audio will be annoyingly out of sync. When a process has critical real-time requirements like this, that is, particular events *must* occur within a specified number of milliseconds, special measures must be taken to ensure that they do occur. Normally, however, most processes are not affected by the underlying multiprogramming of the CPU or the relative speeds of different processes.

The difference between a process and a program is subtle, but absolutely crucial. An analogy may help you here. Consider a culinary-minded computer scientist who is baking a birthday cake for his young daughter. He has a birthday cake recipe and a kitchen well stocked with all the input: flour, eggs, sugar, extract of vanilla, and so on. In this analogy, the recipe is the program, that is, an algorithm expressed in some suitable notation, the computer scientist is the processor (CPU), and the cake ingredients are the input data. The process is the activity consisting of our baker reading the recipe, fetching the ingredients, and baking the cake.

Now imagine that the computer scientist's son comes running in screaming his head off, saying that he has been stung by a bee. The computer scientist records where he was in the recipe (the state of the current process is saved), gets out a first aid book, and begins following the directions in it. Here we see the processor being switched from one process (baking) to a higher-priority process (administering medical care), each having a different program (recipe versus first aid book). When the bee sting has been taken care of, the computer scientist goes back to his cake, continuing at the point where he left off.

The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state. A single processor may be shared among several processes, with some scheduling algorithm being accustomed to determine when to stop work on one process and service a different one. In contrast, a program is something that may be stored on disk, not doing anything.

It is worth noting that if a program is running twice, it counts as two processes. For example, it is often possible to start a word processor twice or print two files at the same time if two printers are available. The fact that two processes happen to be running the same program does not matter; they are distinct processes. The operating system may be able to share the code between them so only one copy is in memory, but that is a technical detail that does not change the conceptual situation of two processes running.

2.1.2 Process Creation

Operating systems need some way to create processes. In very simple systems, or in systems designed for running only a single application (e.g., the controller in a microwave oven), it may be possible to have all the processes that will ever be needed be present when the system comes up. In general-purpose systems, however, some way is needed to create and terminate processes as needed during operation. We will now look at some of the issues.

Four principal events cause processes to be created:

1. System initialization.
2. Execution of a process-creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

When an operating system is booted, typically numerous processes are created. Some of these are foreground processes, that is, processes that interact with (human) users and perform work for them. Others run in the background and are not associated with particular users, but instead have some specific function. For example, one background process may be designed to accept incoming email, sleeping most of the day but suddenly springing to life when email arrives. Another background process may be designed to accept incoming requests for Web pages hosted on that machine, waking up when a request arrives to service the request. Processes that stay in the background to handle some activity such as email, Web pages, news, printing, and so on are called **daemons**. Large systems commonly have dozens of them. In UNIX†, the *ps* program can be used to list the running processes. In Windows, the task manager can be used.

In addition to the processes created at boot time, new processes can be created afterward as well. Often a running process will issue system calls to create one or more new processes to help it do its job. Creating new processes is particularly useful when the work to be done can easily be formulated in terms of several related, but otherwise independent interacting processes. For example, if a large amount of data is being fetched over a network for subsequent processing, it may be convenient to create one process to fetch the data and put them in a shared buffer while a second process removes the data items and processes them. On a multiprocessor, allowing each process to run on a different CPU may also make the job go faster.

In interactive systems, users can start a program by typing a command or (double) clicking on anicon. Taking either of these actions starts a new process and runs the selected program in it. In command-based UNIX systems running X, the new process takes over the window in which it was started. In Windows, when a process is started it does not have a window, but it can create one (or more) and most do. In both systems, users may have multiple windows open at once, each running some process. Using the mouse, the user can select a window and interact with the process, for example, providing input when needed.

The last situation in which processes are created applies only to the batch systems found on large mainframes. Think of inventory management at the end of a day at a chain of stores. Here users can submit batch jobs to the system (possibly remotely). When the operating system decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

Technically, in all these cases, a new process is created by having an existing process execute a process creation system call. That process may be a running user process, a system process invoked from the keyboard or mouse, or a batch-manager process. What that process does is execute a system call to create the new process. This system call tells the operating system to create a new process and indicates, directly or indirectly, which program to run in it.

In UNIX, there is only one system call to create a new process: fork. This call creates an exact clone of the calling process. After the fork, the two processes, the parent and the child, have the same memory image, the same environment strings, and the same open files. That is all there is. Usually, the child process then executes execve or a similar system call to change its memory image and run a new

† In this chapter, UNIX should be interpreted as including almost all POSIX-based systems, including Linux, FreeBSD, OS X, Solaris, etc., and to some extent, Android and iOS as well.

program. For example, when a user types a command, say, *sort*, to the shell, the shell forks off a child process and the child executes *sort*. The reason for this two-step process is to allow the child to manipulate its file descriptors after the fork but before the execve in order to accomplish redirection of standard input, standard output, and standard error.

In Windows, in contrast, a single Win32 function call, CreateProcess, handles both process creation and loading the correct program into the new process. This call has 10 parameters, which include the program to be executed, the command-line parameters to feed that program, various security attributes, bits that control whether open files are inherited, priority information, a specification of the window to be created for the process (if any), and a pointer to a structure in which information about the newly created process is returned to the caller. In addition to CreateProcess, Win32 has about 100 other functions for managing and synchronizing processes and related topics.

In both UNIX and Windows systems, after a process is created, the parent and child have their own distinct address spaces. If either process changes a word in its address space, the change is not visible to the other process. In UNIX, the child's initial address space is a *copy* of the parent's, but there are definitely two distinct address spaces involved; no writable memory is shared. Some UNIX implementations share the program text between the two since that cannot be modified. Alternatively, the child may share all of the parent's memory, but in that case the memory is shared **copy-on-write**, which means that whenever either of the two wants to modify part of the memory, that chunk of memory is explicitly copied first to make sure the modification occurs in a private memory area. Again, no writable memory is shared. It is, however, possible for a newly created process to share some of its creator's other resources, such as open files. In Windows, the parent's and child's address spaces are different from the start.

2.1.3 Process Termination

After a process has been created, it starts running and does whatever its job is. However, nothing lasts forever, not even processes. Sooner or later the new process will terminate, usually due to one of the following conditions:

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Most processes terminate because they have done their work. When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished. This call is exit in UNIX and ExitProcess in

Windows. Screen-oriented programs also support voluntary termination. Word processors, Internet browsers, and similar programs always have an icon or menu item that the user can click to tell the process to remove any temporary files it has open and then terminate.

The second reason for termination is that the process discovers a fatal error. For example, if a user types the command

```
cc foo.c
```

to compile the program *foo.c* and no such file exists, the compiler simply announces this fact and exits. Screen-oriented interactive processes generally do not exit when given bad parameters. Instead they pop up a dialog box and ask the user to try again.

The third reason for termination is an error caused by the process, often due to a program bug. Examples include executing an illegal instruction, referencing nonexistent memory, or dividing by zero. In some systems (e.g., UNIX), a process can tell the operating system that it wishes to handle certain errors itself, in which case the process is signaled (interrupted) instead of terminated when one of the errors occurs.

The fourth reason a process might terminate is that the process executes a system call telling the operating system to kill some other process. In UNIX this call is kill. The corresponding Win32 function is TerminateProcess. In both cases, the killer must have the necessary authorization to do in the killee. In some systems, when a process terminates, either voluntarily or otherwise, all processes it created are immediately killed as well. Neither UNIX nor Windows works this way, however.

2.1.4 Process Hierarchies

In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a process hierarchy. Note that unlike plants and animals that use sexual reproduction, a process has only one parent (but zero, one, two, or more children). So a process is more like a hydra than like, say, a cow.

In UNIX, a process and all of its children and further descendants together form a process group. When a user sends a signal from the keyboard, the signal is delivered to all members of the process group currently associated with the keyboard (usually all active processes that were created in the current window). Individually, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

As another example of where the process hierarchy plays a key role, let us look at how UNIX initializes itself when it is started, just after the computer is booted. A special process, called *init*, is present in the boot image. When it starts running, it reads a file telling how many terminals there are. Then it forks off a new process per terminal. These processes wait for someone to log in. If a login is successful, the login process executes a shell to accept commands. These commands may start up more processes, and so forth. Thus, all the processes in the whole system belong to a single tree, with *init* at the root.

In contrast, Windows has no concept of a process hierarchy. All processes are equal. The only hint of a process hierarchy is that when a process is created, the parent is given a special token (called a **handle**) that it can use to control the child. However, it is free to pass this token to some other process, thus invalidating the hierarchy. Processes in UNIX cannot disinherit their children.

2.1.5 Process States

Although each process is an independent entity, with its own program counter and internal state, processes often need to interact with other processes. One process may generate some output that another process uses as input. In the shell command

```
cat chapter1 chapter2 chapter3 | grep tree
```

the first process, running *cat*, concatenates and outputs three files. The second process, running *grep*, selects all lines containing the word "tree." Depending on the relative speeds of the two processes (which depends on both the relative complexity of the programs and how much CPU time each one has had), it may happen that *grep* is ready to run, but there is no input waiting for it. It must then block until some input is available.

When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. It is also possible for a process that is conceptually ready and able to run to be stopped because the operating system has decided to allocate the CPU to another process for a while. These two conditions are completely different. In the first case, the suspension is inherent in the problem (you cannot process the user's command line until it has been typed). In the second case, it is a technicality of the system (not enough CPUs to give each process its own private processor). In Fig. 2-2 we see a state diagram showing the three states a process may be in:

1. Running (actually using the CPU at that instant).
2. Ready (runnable; temporarily stopped to let another process run).
3. Blocked (unable to run until some external event happens).

Logically, the first two states are similar. In both cases the process is willing to run, only in the second one, there is temporarily no CPU available for it. The third state is fundamentally different from the first two in that the process cannot run, even if the CPU is idle and has nothing else to do.

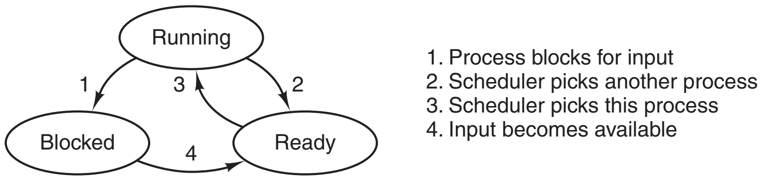

Running
Blocked
Ready
1
2
3
4
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

Four transitions are possible among these three states, as shown. Transition 1 occurs when the operating system discovers that a process cannot continue right now. In some systems the process can execute a system call, such as pause, to get into blocked state. In other systems, including UNIX, when a process reads from a pipe or special file (e.g., a terminal) and there is no input available, the process is automatically blocked.

Transitions 2 and 3 are caused by the process scheduler, a part of the operating system, without the process even knowing about them. Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. Transition 3 occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again. The subject of scheduling, that is, deciding which process should run when and for how long, is an important one; we will look at it later in this chapter. Many algorithms have been devised to try to balance the competing demands of efficiency for the system as a whole and fairness to individual processes. We will study some of them later in this chapter.

Transition 4 occurs when the external event for which a process was waiting (such as the arrival of some input) happens. If no other process is running at that instant, transition 3 will be triggered and the process will start running. Otherwise it may have to wait in *ready* state for a little while until the CPU is available and its turn comes.

Using the process model, it becomes much easier to think about what is going on inside the system. Some of the processes run programs that carry out commands typed in by a user. Other processes are part of the system and handle tasks such as carrying out requests for file services or managing the details of running a disk or a tape drive. When a disk interrupt occurs, the system makes a decision to stop running the current process and run the disk process, which was blocked waiting for that interrupt. Thus, instead of thinking about interrupts, we can think about user processes, disk processes, terminal processes, and so on, which block when they are waiting for something to happen. When the disk has been read or the character typed, the process waiting for it is unblocked and is eligible to run again.

This view gives rise to the model shown in Fig. 2-3. Here the lowest level of the operating system is the scheduler, with a variety of processes on top of it. All the interrupt handling and details of actually starting and stopping processes are hidden away in what is here called the scheduler, which is actually not much code. The rest of the operating system is nicely structured in process form. Few real systems are as nicely structured as this, however.

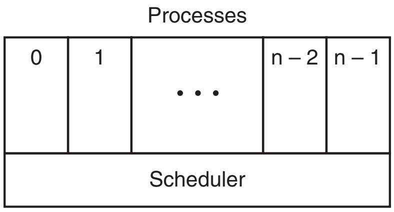


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

2.1.6 Implementation of Processes

To implement the process model, the operating system maintains a table (an array of structures), called the **process table**, with one entry per process. (Some authors call these entries **process control blocks**.) This entry contains important information about the process' state, including its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from *running* to *ready* or *blocked* state so that it can be restarted later as if it had never been stopped.

Figure 2-4 shows some of the key fields in a typical system. The fields in the first column relate to process management. The other two relate to memory management and file management, respectively. It should be noted that precisely which fields the process table has is highly system dependent, but this figure gives a general idea of the kinds of information needed.

Now that we have looked at the process table, it is possible to explain a little more about how the illusion of multiple sequential processes is maintained on one (or each) CPU. Associated with each I/O class is a location (typically at a fixed location near the bottom of memory) called the **interrupt vector**. It contains the address of the interrupt service procedure. Suppose that user process 3 is running when a disk interrupt happens. User process 3's program counter, program status word, and sometimes one or more registers are pushed onto the (current) stack by the interrupt hardware. The computer then jumps to the address specified in the interrupt vector. That is all the hardware does. From here on, it is up to the software, in particular, the interrupt service procedure.

All interrupts start by saving the registers, often in the process table entry for the current process. Then the information pushed onto the stack by the interrupt is

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 2-4. Some of the fields of a typical process-table entry.

removed and the stack pointer is set to point to a temporary stack used by the process handler. Actions such as saving the registers and setting the stack pointer cannot even be expressed in high-level languages such as C, so they are performed by a small assembly-language routine, usually the same one for all interrupts since the work of saving the registers is identical, no matter what the cause of the interrupt is.

When this routine is finished, it calls a C procedure to do the rest of the work for this specific interrupt type. (We assume the operating system is written in C, the usual choice for all real operating systems.) When it has done its job, possibly making some process now ready, the scheduler is called to see who to run next. After that, control is passed back to the assembly-language code to load up the registers and memory map for the now-current process and start it running. Interrupt handling and scheduling are summarized in Fig. 2-5. It is worth noting that the details vary somewhat from system to system.

A process may be interrupted thousands of times during its execution, but the key idea is that after each interrupt the interrupted process returns to precisely the same state it was in before the interrupt occurred.

2.1.7 Modeling Multiprogramming

When multiprogramming is used, the CPU utilization can be improved. Crudely put, if the average process computes only 20% of the time it is sitting in memory, then with five processes in memory at once the CPU should be busy all the time. This model is unrealistically optimistic, however, since it tacitly assumes that all five processes will never be waiting for I/O at the same time.

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly-language procedure saves registers.
4. Assembly-language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly-language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

A better model is to look at CPU usage from a probabilistic viewpoint. Suppose that a process spends a fraction p of its time waiting for I/O to complete. With n processes in memory at once, the probability that all n processes are waiting for I/O (in which case the CPU will be idle) is p^n. The CPU utilization is then given by the formula

$$\text{CPU utilization} = 1 - p^n$$

Figure 2-6 shows the CPU utilization as a function of n, which is called the **degree of multiprogramming**.

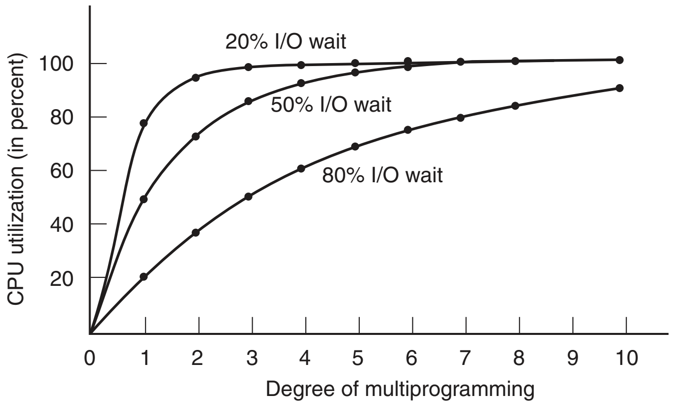


Figure 2-6. CPU utilization as a function of the number of processes in memory.

From the figure it is clear that if processes spend 80% of their time waiting for I/O, at least 10 processes must be in memory at once to get the CPU waste below 10%. When you realize that an interactive process waiting for a user to type something at a terminal (or click on an icon) is in I/O wait state, it should be clear that I/O wait times of 80% and more are not unusual. But even on servers, processes doing a lot of disk I/O will often have this percentage or more.

For the sake of accuracy, it should be pointed out that the probabilistic model just described is only an approximation. It implicitly assumes that all *n* processes are independent, meaning that it is quite acceptable for a system with five processes in memory to have three running and two waiting. But with a single CPU, we cannot have three processes running at once, so a process becoming ready while the CPU is busy will have to wait. Thus the processes are not independent. A more accurate model can be constructed using queueing theory, but the point we are making—multiprogramming lets processes use the CPU when it would otherwise become idle—is, of course, still valid, even if the true curves of Fig. 2-6 are slightly different from those shown in the figure.

Even though the model of Fig. 2-6 is simple-minded, it can nevertheless be used to make specific, although approximate, predictions about CPU performance. Suppose, for example, that a computer has 8 GB of memory, with the operating system and its tables taking up 2 GB and each user program also taking up 2 GB. These sizes allow three user programs to be in memory at once. With an 80% average I/O wait, we have a CPU utilization (ignoring operating system overhead) of $1 - 0.8^3$ or about 49%. Adding another 8 GB of memory allows the system to go from three-way multiprogramming to seven-way multiprogramming, thus raising the CPU utilization to 79%. In other words, the additional 8 GB will raise the throughput by 30%.

Adding yet another 8 GB would increase CPU utilization only from 79% to 91%, thus raising the throughput by only another 12%. Using this model, the computer's owner might decide that the first addition was a good investment but that the second was not.

2.2 THREADS

In traditional operating systems, each process has an address space and a single thread of control. In fact, that is almost the definition of a process. Nevertheless, in many situations, it is desirable to have multiple threads of control in the same address space running in quasi-parallel, as though they were (almost) separate processes (except for the shared address space). In the following sections we will discuss these situations and their implications.

2.2.1 Thread Usage

Why would anyone want to have a kind of process within a process? It turns out there are several reasons for having these miniprocesses, called **threads**. Let us now examine some of them. The main reason for having threads is that in many applications, multiple activities are going on at once. Some of these may block from time to time. By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

We have seen this argument once before. It is precisely the argument for having processes. Instead, of thinking about interrupts, timers, and context switches, we can think about parallel processes. Only now with threads we add a new element: the ability for the parallel entities to share an address space and all of its data among themselves. This ability is essential for certain applications, which is why having multiple processes (with their separate address spaces) will not work.

A second argument for having threads is that since they are lighter weight than processes, they are easier (i.e., faster) to create and destroy than processes. In many systems, creating a thread goes 10–100 times faster than creating a process. When the number of threads needed changes dynamically and rapidly, this property is useful to have.

A third reason for having threads is also a performance argument. Threads yield no performance gain when all of them are CPU bound, but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application.

Finally, threads are useful on systems with multiple CPUs, where real parallelism is possible. We will come back to this issue in Chap. 8.

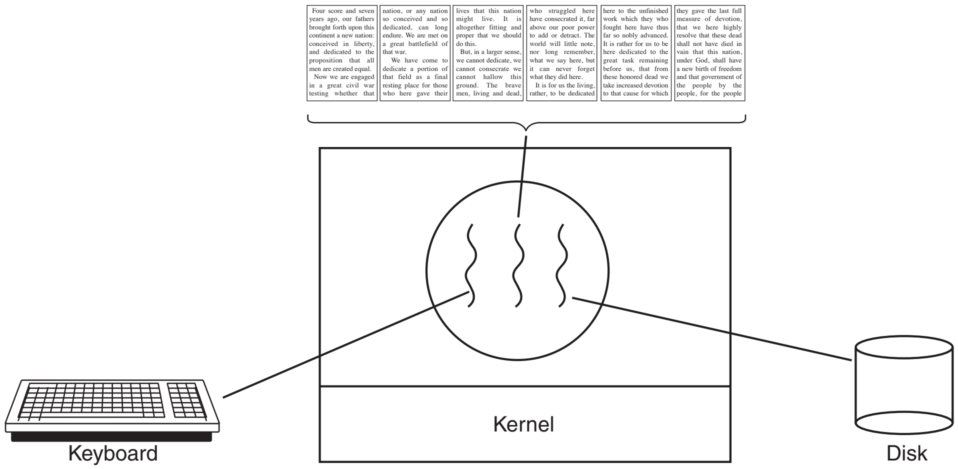
It is easiest to see why threads are useful by looking at some concrete examples. As a first example, consider a word processor. Word processors usually display the document being created on the screen formatted exactly as it will appear on the printed page. In particular, all the line breaks and page breaks are in their correct and final positions, so that the user can inspect them and change the document if need be (e.g., to eliminate widows and orphans—incomplete top and bottom lines on a page, which are considered esthetically unpleasing).

Suppose that the user is writing a book. From the author's point of view, it is easiest to keep the entire book as a single file to make it easier to search for topics, perform global substitutions, and so on. Alternatively, each chapter might be a separate file. However, having every section and subsection as a separate file is a real nuisance when global changes have to be made to the entire book, since then hundreds of files have to be individually edited, one at a time. For example, if proposed standard xxxx is approved just before the book goes to press, all occurrences of "Draft Standard xxxx" have to be changed to "Standard xxxx" at the last minute. If the entire book is one file, typically a single command can do all the substitutions. In contrast, if the book is spread over 300 files, each one must be edited separately.

Now consider what happens when the user suddenly deletes one sentence from page 1 of an 800-page book. After checking the changed page for correctness, he now wants to make another change on page 600 and types in a command telling the word processor to go to that page (possibly by searching for a phrase occurring only there). The word processor is now forced to reformat the entire book up to page 600 on the spot because it does not know what the first line of page 600 will be until it has processed all the previous pages. There may be a substantial delay before page 600 can be displayed, leading to an unhappy user.

Threads can help here. Suppose that the word processor is written as a two-threaded program. One thread interacts with the user and the other handles reformatting in the background. As soon as the sentence is deleted from page 1, the interactive thread tells the reformatting thread to reformat the whole book. Meanwhile, the interactive thread continues to listen to the keyboard and mouse and responds to simple commands like scrolling page 1 while the other thread is computing madly in the background. With a little luck, the reformatting will be completed before the user asks to see page 600, so it can be displayed instantly.

While we are at it, why not add a third thread? Many word processors have a feature of automatically saving the entire file to disk every few minutes to protect the user against losing a day's work in the event of a program crash, system crash, or power failure. The third thread can handle the disk backups without interfering with the other two. The situation with three threads is shown in Fig. 2-7.



Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.

We have come to dedicate a portion of that field as a final resting place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here.

It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people, for the people

Figure 2-7. A word processor with three threads.

If the program were single-threaded, then whenever a disk backup started, commands from the keyboard and mouse would be ignored until the backup was finished. The user would surely perceive this as sluggish performance. Alternatively, keyboard and mouse events could interrupt the disk backup, allowing good performance but leading to a complex interrupt-driven programming model. With three threads, the programming model is much simpler. The first thread just interacts with the user. The second thread reformats the document when told to. The third thread writes the contents of RAM to disk periodically.

It should be clear that having three separate processes would not work here because all three threads need to operate on the document. By having three threads instead of three processes, they share a common memory and thus all have access to the document being edited. With three processes this would be impossible.

An analogous situation exists with many other interactive programs. For example, an electronic spreadsheet is a program that allows a user to maintain a matrix, some of whose elements are data provided by the user. Other elements are computed based on the input data using potentially complex formulas. When a user changes one element, many other elements may have to be recomputed. By having a background thread do the recomputation, the interactive thread can allow the user to make additional changes while the computation is going on. Similarly, a third thread can handle periodic backups to disk on its own.

Now consider yet another example of where threads are useful: a server for a Website. Requests for pages come in and the requested page is sent back to the client. At most Websites, some pages are more commonly accessed than other pages. For example, Sony's home page is accessed far more than a page deep in the tree containing the technical specifications of any particular camera. Web servers use this fact to improve performance by maintaining a collection of heavily used pages in main memory to eliminate the need to go to disk to get them. Such a collection is called a **cache** and is used in many other contexts as well. We saw CPU caches in Chap. 1, for example.

One way to organize the Web server is shown in Fig. 2-8(a). Here one thread, the **dispatcher**, reads incoming requests for work from the network. After examining the request, it chooses an idle (i.e., blocked) **worker thread** and hands it the request, possibly by writing a pointer to the message into a special word associated with each thread. The dispatcher then wakes up the sleeping worker, moving it from blocked state to ready state.

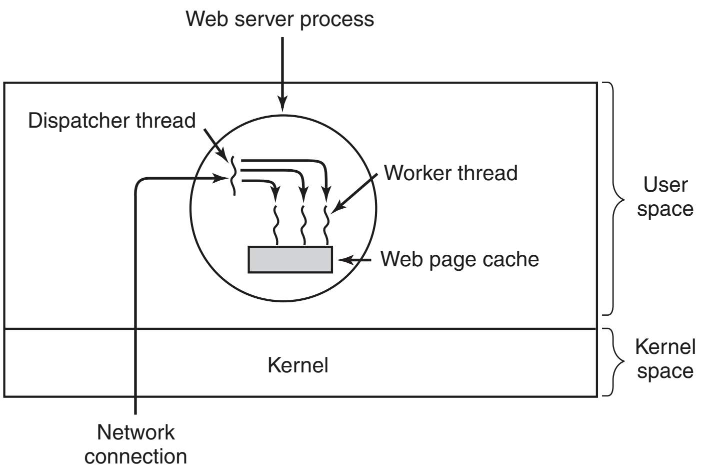


Figure 2-8. A multithreaded Web server.

When the worker wakes up, it checks to see if the request can be satisfied from the Web page cache, to which all threads have access. If not, it starts a read operation to get the page from the disk and blocks until the disk operation completes.

When the thread blocks on the disk operation, another thread is chosen to run, possibly the dispatcher, in order to acquire more work, or possibly another worker that is now ready to run.

This model allows the server to be written as a collection of sequential threads. The dispatcher's program consists of an infinite loop for getting a work request and handing it off to a worker. Each worker's code consists of an infinite loop consisting of accepting a request from the dispatcher and checking the Web cache to see if the page is present. If so, it is returned to the client, and the worker blocks waiting for a new request. If not, it gets the page from the disk, returns it to the client, and blocks waiting for a new request.

A rough outline of the code is given in Fig. 2-9. Here, as in the rest of this book, *TRUE* is assumed to be the constant 1. Also, *buf* and *page* are structures appropriate for holding a work request and a Web page, respectively.

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Figure 2-9. A rough outline of the code for Fig. 2-8. (a) Dispatcher thread. (b) Worker thread.

Consider how the Web server could be written in the absence of threads. One possibility is to have it operate as a single thread. The main loop of the Web server gets a request, examines it, and carries it out to completion before getting the next one. While waiting for the disk, the server is idle and does not process any other incoming requests. If the Web server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the Web server is waiting for the disk. The net result is that many fewer requests/sec can be processed. Thus, threads gain considerable performance, but each thread is programmed sequentially, in the usual way.

So far we have seen two possible designs: a multithreaded Web server and a single-threaded Web server. Suppose that threads are not available but the system designers find the performance loss due to single threading unacceptable. If a nonblocking version of the read system call is available, a third approach is possible. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a nonblocking disk operation is started.

The server records the state of the current request in a table and then goes and gets the next event. The next event may either be a request for new work or a reply from the disk about a previous operation. If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed. With nonblocking disk I/O, a reply probably will have to take the form of a signal or interrupt.

In this design, the "sequential process" model that we had in the first two cases is lost. The state of the computation must be explicitly saved and restored in the table every time the server switches from working on one request to another. In effect, we are simulating the threads and their stacks the hard way. A design like this, in which each computation has a saved state, and there exists some set of events that can occur to change the state, is called a **finite-state machine**. This concept is widely used throughout computer science.

It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking calls (e.g., for disk I/O) and still achieve parallelism. Blocking system calls make programming easier, and parallelism improves performance. The single-threaded server retains the simplicity of blocking system calls but gives up performance. The third approach achieves high performance through parallelism but uses nonblocking calls and interrupts and thus is hard to program. These models are summarized in Fig. 2-10.

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Figure 2-10. Three ways to construct a server.

A third example where threads are useful is in applications that must process very large amounts of data. The normal approach is to read in a block of data, process it, and then write it out again. The problem here is that if only blocking system calls are available, the process blocks while data are coming in and data are going out. Having the CPU go idle when there is lots of computing to do is clearly wasteful and should be avoided if possible.

Threads offer a solution. The process could be structured with an input thread, a processing thread, and an output thread. The input thread reads data into an input buffer. The processing thread takes data out of the input buffer, processes them, and puts the results in an output buffer. The output buffer writes these results back to disk. In this way, input, output, and processing can all be going on at the same time. Of course, this model works only if a system call blocks only the calling thread, not the entire process.

2.2.2 The Classical Thread Model

Now that we have seen why threads might be useful and how they can be used, let us investigate the idea a bit more closely. The process model is based on two independent concepts: resource grouping and execution. Sometimes it is useful to separate them; this is where threads come in. First we will look at the classical thread model; after that we will examine the Linux thread model, which blurs the line between processes and threads.

One way of looking at a process is that it is a way to group related resources together. A process has an address space containing program text and data, as well as other resources. These resources may include open files, child processes, pending alarms, signal handlers, accounting information, and more. By putting them together in the form of a process, they can be managed more easily.

The other concept a process has is a thread of execution, usually shortened to just **thread**. The thread has a program counter that keeps track of which instruction to execute next. It has registers, which hold its current working variables. It has a stack, which contains the execution history, with one frame for each procedure called but not yet returned from. Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately. Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.

What threads add to the process model is to allow multiple executions to take place in the same process environment, to a large degree independent of one another. Having multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer. In the former case, the threads share an address space and other resources. In the latter case, processes share physical memory, disks, printers, and other resources. Because threads have some of the properties of processes, they are sometimes called **lightweight processes**. The term **multithreading** is also used to describe the situation of allowing multiple threads in the same process. As we saw in Chap. 1, some CPUs have direct hardware support for multithreading and allow thread switches to happen on a nanosecond time scale.

In Fig. 2-11(a) we see three traditional processes. Each process has its own address space and a single thread of control. In contrast, in Fig. 2-11(b) we see a single process with three threads of control. Although in both cases we have three threads, in Fig. 2-11(a) each of them operates in a different address space, whereas in Fig. 2-11(b) all three of them share the same address space.

When a multithreaded process is run on a single-CPU system, the threads take turns running. In Fig. 2-1, we saw how multiprogramming of processes works. By switching back and forth among multiple processes, the system gives the illusion of separate sequential processes running in parallel. Multithreading works the same way. The CPU switches rapidly back and forth among the threads, providing the illusion that the threads are running in parallel, albeit on a slower CPU than the real one. With three compute-bound threads in a process, the threads would appear to be running in parallel, each one on a CPU with one-third the speed of the real CPU.

Different threads in a process are not as independent as different processes. All threads have exactly the same address space, which means that they also share the

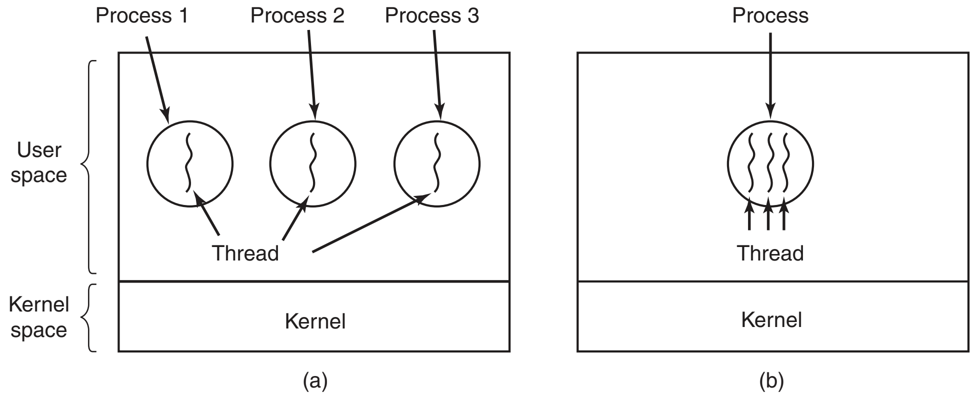


Figure 2-11. (a) Three processes each with one thread. (b) One process with three threads.

same global variables. Since every thread can access every memory address within the process' address space, one thread can read, write, or even wipe out another thread's stack. There is no protection between threads because (1) it is impossible, and (2) it should not be necessary. Unlike different processes, which may be from different users and which may be hostile to one another, a process is always owned by a single user, who has presumably created multiple threads so that they can cooperate, not fight. In addition to sharing an address space, all the threads can share the same set of open files, child processes, alarms, and signals, an so on, as shown in Fig. 2-12. Thus, the organization of Fig. 2-11(a) would be used when the three processes are essentially unrelated, whereas Fig. 2-11(b) would be appropriate when the three threads are actually part of the same job and are actively and closely cooperating with each other.

Per-process items	**Per-thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

The items in the first column are process properties, not thread properties. For example, if one thread opens a file, that file is visible to the other threads in the process and they can read and write it. This is logical, since the process is the unit of resource management, not the thread. If each thread had its own address space, open files, pending alarms, and so on, it would be a separate process. What we are trying to achieve with the thread concept is the ability for multiple threads of execution to share a set of resources so that they can work together closely to perform some task.

Like a traditional process (i.e., a process with only one thread), a thread can be in any one of several states: running, blocked, ready, or terminated. A running thread currently has the CPU and is active. In contrast, a blocked thread is waiting for some event to unblock it. For example, when a thread performs a system call to read from the keyboard, it is blocked until input is typed. A thread can block waiting for some external event to happen or for some other thread to unblock it. A ready thread is scheduled to run and will as soon as its turn comes up. The transitions between thread states are the same as those between process states and are illustrated in Fig. 2-2.

It is important to realize that each thread has its own stack, as illustrated in Fig. 2-13. Each thread's stack contains one frame for each procedure called but not yet returned from. This frame contains the procedure's local variables and the return address to use when the procedure call has finished. For example, if procedure *X* calls procedure *Y* and *Y* calls procedure *Z*, then while *Z* is executing, the frames for *X*, *Y*, and *Z* will all be on the stack. Each thread will generally call different procedures and thus have a different execution history. This is why each thread needs its own stack.

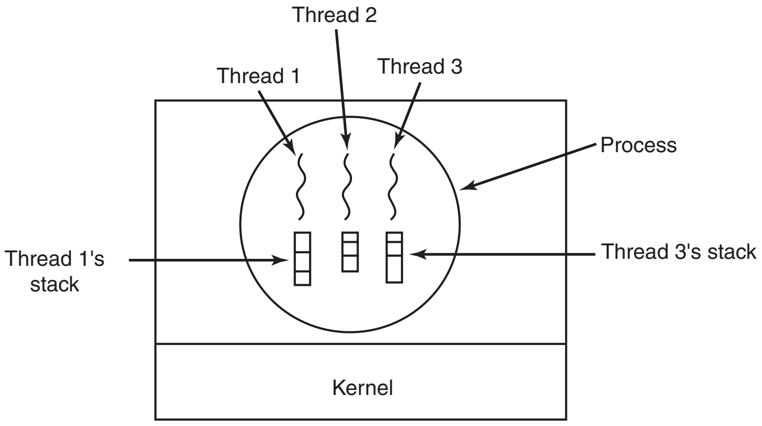


Figure 2-13. Each thread has its own stack.

When multithreading is present, processes usually start with a single thread present. This thread has the ability to create new threads by calling a library procedure such as *thread_create*. A parameter to *thread_create* specifies the name of a procedure for the new thread to run. It is not necessary (or even possible) to specify anything about the new thread's address space, since it automatically runs in the address space of the creating thread. Sometimes threads are hierarchical, with a parent-child relationship, but often no such relationship exists, with all threads being equal. With or without a hierarchical relationship, the creating thread is usually returned a thread identifier that names the new thread.

When a thread has finished its work, it can exit by calling a library procedure, say, *thread_exit*. It then vanishes and is no longer schedulable. In some thread systems, one thread can wait for a (specific) thread to exit by calling a procedure, for example, *thread_join*. This procedure blocks the calling thread until a (specific) thread has exited. In this regard, thread creation and termination is very much like process creation and termination, with approximately the same options as well.

Another common thread call is *thread_yield*, which allows a thread to voluntarily give up the CPU to let another thread run. Such a call is important because there is no clock interrupt to actually enforce multiprogramming as there is with processes. Thus it is important for threads to be polite and voluntarily surrender the CPU from time to time to give other threads a chance to run. Other calls allow one thread to wait for another thread to finish some work, for a thread to announce that it has finished some work, and so on.

While threads are often useful, they also introduce a number of complications into the programming model. To start with, consider the effects of the UNIX fork system call. If the parent process has multiple threads, should the child also have them? If not, the process may not function properly, since all of them may be essential.

However, if the child process gets as many threads as the parent, what happens if a thread in the parent was blocked on a read call, say, from the keyboard? Are two threads now blocked on the keyboard, one in the parent and one in the child? When a line is typed, do both threads get a copy of it? Only the parent? Only the child? The same problem exists with open network connections.

Another class of problems is related to the fact that threads share many data structures. What happens if one thread closes a file while another one is still reading from it? Suppose one thread notices that there is too little memory and starts allocating more memory. Partway through, a thread switch occurs, and the new thread also notices that there is too little memory and also starts allocating more memory. Memory will probably be allocated twice. These problems can be solved with some effort, but careful thought and design are needed to make multithreaded programs work correctly.

2.2.3 POSIX Threads

To make it possible to write portable threaded programs, IEEE has defined a standard for threads in IEEE standard 1003.1c. The threads package it defines is called **Pthreads**. Most UNIX systems support it. The standard defines over 60 function calls, which is far too many to go over here. Instead, we will just describe a few of the major ones to give an idea of how it works. The calls we will describe below are listed in Fig. 2-14.

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figure 2-14. Some of the Pthreads function calls.

All Pthreads threads have certain properties. Each one has an identifier, a set of registers (including the program counter), and a set of attributes, which are stored in a structure. The attributes include the stack size, scheduling parameters, and other items needed to use the thread.

A new thread is created using the *pthread_create* call. The thread identifier of the newly created thread is returned as the function value. This call is intentionally very much like the fork system call (except with parameters), with the thread identifier playing the role of the PID, mostly for identifying threads referenced in other calls.

When a thread has finished the work it has been assigned, it can terminate by calling *pthread_exit*. This call stops the thread and releases its stack.

Often a thread needs to wait for another thread to finish its work and exit before continuing. The thread that is waiting calls *pthread_join* to wait for a specific other thread to terminate. The thread identifier of the thread to wait for is given as a parameter.

Sometimes it happens that a thread is not logically blocked, but feels that it has run long enough and wants to give another thread a chance to run. It can accomplish this goal by calling *pthread_yield*. There is no such call for processes because the assumption there is that processes are fiercely competitive and each wants all the CPU time it can get. However, since the threads of a process are working together and their code is invariably written by the same programmer, sometimes the programmer wants them to give each other another chance.

The next two thread calls deal with attributes. *Pthread_attr_init* creates the attribute structure associated with a thread and initializes it to the default values. These values (such as the priority) can be changed by manipulating fields in the attribute structure.

Finally, *pthread_attr_destroy* removes a thread's attribute structure, freeing up its memory. It does not affect threads using it; they continue to exist.

To get a better feel for how Pthreads works, consider the simple example of Fig. 2-15. Here the main program loops *NUMBER_OF_THREADS* times, creating a new thread on each iteration, after announcing its intention. If the thread creation fails, it prints an error message and then exits. After creating all the threads, the main program exits.

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

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d\n", tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d\n", i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d\n", status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

Figure 2-15. An example program using threads.

When a thread is created, it prints a one-line message announcing itself, then it exits. The order in which the various messages are interleaved is nondeterminate and may vary on consecutive runs of the program.

The Pthreads calls described above are not the only ones. We will examine some of the others after we have discussed process and thread synchronization.

2.2.4 Implementing Threads in User Space

There are two main places to implement threads: user space and the kernel. The choice is a bit controversial, and a hybrid implementation is also possible. We will now describe these methods, along with their advantages and disadvantages.

The first method is to put the threads package entirely in user space. The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. The first, and most obvious, advantage is that a user-level threads package can be implemented on an operating system that does not support threads. All operating systems used to fall into this category, and even now some still do. With this approach, threads are implemented by a library.

All of these implementations have the same general structure, illustrated in Fig. 2-16(a). The threads run on top of a run-time system, which is a collection of procedures that manage threads. We have seen four of these already: *pthread_create*, *pthread_exit*, *pthread_join*, and *pthread_yield*, but usually there are more.

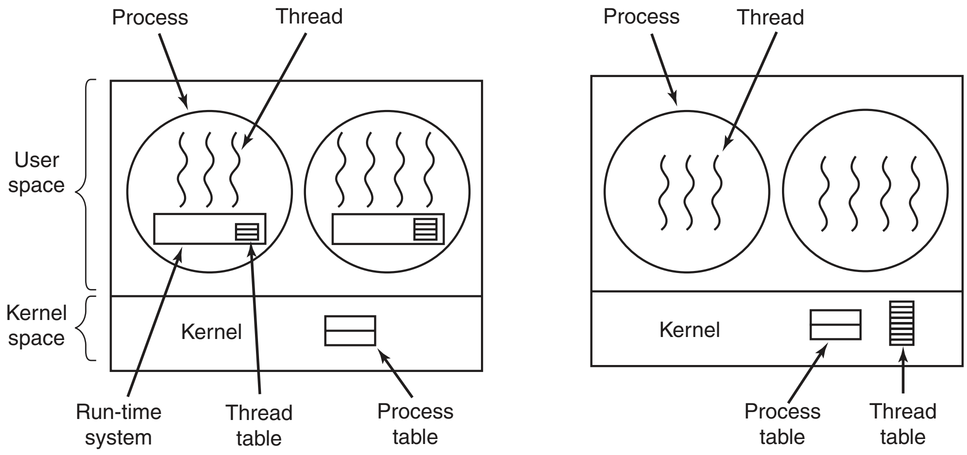


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

When threads are managed in user space, each process needs its own private **thread table** to keep track of the threads in that process. This table is analogous to the kernel's process table, except that it keeps track only of the per-thread properties, such as each thread's program counter, stack pointer, registers, state, and so forth. The thread table is managed by the run-time system. When a thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table, exactly the same way as the kernel stores information about processes in the process table.

When a thread does something that may cause it to become blocked locally, for example, waiting for another thread in its process to complete some work, it calls a run-time system procedure. This procedure checks to see if the thread must be put into blocked state. If so, it stores the thread's registers (i.e., its own) in the thread table, looks in the table for a ready thread to run, and reloads the machine registers with the new thread's saved values. As soon as the stack pointer and program counter have been switched, the new thread comes to life again automatically. If the machine happens to have an instruction to store all the registers and another one to load them all, the entire thread switch can be done in just a handful of instructions. Doing thread switching like this is at least an order of magnitude—maybe more—faster than trapping to the kernel and is a strong argument in favor of user-level threads packages.

However, there is one key difference with processes. When a thread is finished running for the moment, for example, when it calls *thread_yield*, the code of *thread_yield* can save the thread's information in the thread table itself. Furthermore, it can then call the thread scheduler to pick another thread to run. The procedure that saves the thread's state and the scheduler are just local procedures, so invoking them is much more efficient than making a kernel call. Among other issues, no trap is needed, no context switch is needed, the memory cache need not be flushed, and so on. This makes thread scheduling very fast.

User-level threads also have other advantages. They allow each process to have its own customized scheduling algorithm. For some applications, for example, those with a garbage-collector thread, not having to worry about a thread being stopped at an inconvenient moment is a plus. They also scale better, since kernel threads invariably require some table space and stack space in the kernel, which can be a problem if there are a very large number of threads.

Despite their better performance, user-level threads packages have some major problems. First among these is the problem of how blocking system calls are implemented. Suppose that a thread reads from the keyboard before any keys have been hit. Letting the thread actually make the system call is unacceptable, since this will stop all the threads. One of the main goals of having threads in the first place was to allow each one to use blocking calls, but to prevent one blocked thread from affecting the others. With blocking system calls, it is hard to see how this goal can be achieved readily.

The system calls could all be changed to be nonblocking (e.g., a read on the keyboard would just return 0 bytes if no characters were already buffered), but requiring changes to the operating system is unattractive. Besides, one argument for user-level threads was precisely that they could run with *existing* operating systems. In addition, changing the semantics of read will require changes to many user programs.

Another alternative is available in the event that it is possible to tell in advance if a call will block. In most versions of UNIX, a system call, select, exists, which allows the caller to tell whether a prospective read will block. When this call is present, the library procedure *read* can be replaced with a new one that first does a select call and then does the read call only if it is safe (i.e., will not block). If the read call will block, the call is not made. Instead, another thread is run. The next time the run-time system gets control, it can check again to see if the read is now safe. This approach requires rewriting parts of the system call library, and is inefficient and inelegant, but there is little choice. The code placed around the system call to do the checking is called a **jacket** or **wrapper**.

Somewhat analogous to the problem of blocking system calls is the problem of page faults. We will study these in Chap. 3. For the moment, suffice it to say that computers can be set up in such a way that not all of the program is in main memory at once. If the program calls or jumps to an instruction that is not in memory, a page fault occurs and the operating system will go and get the missing instruction (and its neighbors) from disk. This is called a page fault. The process is blocked while the necessary instruction is being located and read in. If a thread causes a page fault, the kernel, unaware of even the existence of threads, naturally blocks the entire process until the disk I/O is complete, even though other threads might be runnable.01

Another problem with user-level thread packages is that if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU. Within a single process, there are no clock interrupts, making it impossible to schedule processes round-robin fashion (taking turns). Unless a thread enters the run-time system of its own free will, the scheduler will never get a chance.

One possible solution to the problem of threads running forever is to have the run-time system request a clock signal (interrupt) once a second to give it control, but this, too, is crude and messy to program. Periodic clock interrupts at a higher frequency are not always possible, and even if they are, the total overhead may be substantial. Furthermore, a thread might also need a clock interrupt, interfering with the run-time system's use of the clock.

Another, and really the most devastating, argument against user-level threads is that programmers generally want threads precisely in applications where the threads block often, as, for example, in a multithreaded Web server. These threads are constantly making system calls. Once a trap has occurred to the kernel to carry out the system call, it is hardly any more work for the kernel to switch threads if the old one has blocked, and having the kernel do this eliminates the need for constantly making select system calls that check to see if read system calls are safe. For applications that are essentially entirely CPU bound and rarely block, what is the point of having threads at all? No one would seriously propose computing the first *n* prime numbers or playing chess using threads because there is nothing to be gained by doing it that way.

2.2.5 Implementing Threads in the Kernel

Now let us consider having the kernel know about and manage the threads. No run-time system is needed in each, as shown in Fig. 2-16(b). Also, there is no thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system. When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table.

The kernel's thread table holds each thread's registers, state, and other information. The information is the same as with user-level threads, but now kept in the kernel instead of in user space (inside the run-time system). This information is a subset of the information that traditional kernels maintain about their single-threaded processes, that is, the process state. In addition, the kernel also maintains the traditional process table to keep track of processes.

All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure. When a thread blocks, the kernel, at its option, can run either another thread from the same process (if one is ready) or a thread from a different process. With user-level threads, the run-time system keeps running threads from its own process until the kernel takes the CPU away from it (or there are no ready threads left to run).

Due to the relatively greater cost of creating and destroying threads in the kernel, some systems take an environmentally correct approach and recycle their threads. When a thread is destroyed, it is marked as not runnable, but its kernel data structures are not otherwise affected. Later, when a new thread must be created, an old thread is reactivated, saving some overhead. Thread recycling is also possible for user-level threads, but since the thread-management overhead is much smaller, there is less incentive to do this.

Kernel threads do not require any new, nonblocking system calls. In addition, if one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads, and if so, run one of them while waiting for the required page to be brought in from the disk. Their main disadvantage is that the cost of a system call is substantial, so if thread operations (creation, termination, etc.) a common, much more overhead will be incurred.

While kernel threads solve some problems, they do not solve all problems. For example, what happens when a multithreaded process forks? Does the new process have as many threads as the old one did, or does it have just one? In many cases, the best choice depends on what the process is planning to do next. If it is going to call exec to start a new program, probably one thread is the correct choice, but if it continues to execute, reproducing all the threads is probably best.

Another issue is signals. Remember that signals are sent to processes, not to threads, at least in the classical model. When a signal comes in, which thread should handle it? Possibly threads could register their interest in certain signals, so when a signal came in it would be given to the thread that said it wants it. But what happens if two or more threads register for the same signal? These are only two of the problems threads introduce, and there are more.

2.2.6 Hybrid Implementations

Various ways have been investigated to try to combine the advantages of user-level threads with kernel-level threads. One way is use kernel-level threads and then multiplex user-level threads onto some or all of them, as shown in Fig. 2-17.

When this approach is used, the programmer can determine how many kernel threads to use and how many user-level threads to multiplex on each one. This model gives the ultimate in flexibility.

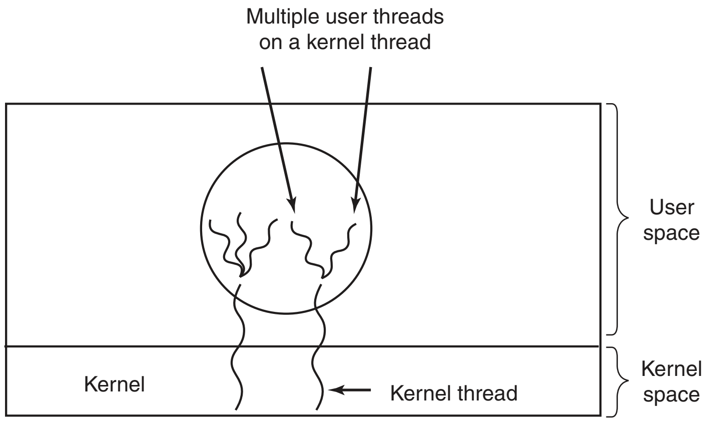


Figure 2-17. Multiplexing user-level threads onto kernel-level threads.

With this approach, the kernel is aware of *only* the kernel-level threads and schedules those. Some of those threads may have multiple user-level threads multiplexed on top of them. These user-level threads are created, destroyed, and scheduled just like user-level threads in a process that runs on an operating system without multithreading capability. In this model, each kernel-level thread has some set of user-level threads that take turns using it.

2.2.7 Scheduler Activations

While kernel threads are better than user-level threads in some key ways, they are also indisputably slower. As a consequence, researchers have looked for ways to improve the situation without giving up their good properties. Below we will describe an approach devised by Anderson et al. (1992), called **scheduler activations**. Related work is discussed by Edler et al. (1988) and Scott et al. (1990).

The goals of the scheduler activation work are to mimic the functionality of kernel threads, but with the better performance and greater flexibility usually associated with threads packages implemented in user space. In particular, user threads should not have to make special nonblocking system calls or check in advance if it is safe to make certain system calls. Nevertheless, when a thread blocks on a system call or on a page fault, it should be possible to run other threads within the same process, if any are ready.

Efficiency is achieved by avoiding unnecessary transitions between user and kernel space. If a thread blocks waiting for another thread to do something, for example, there is no reason to involve the kernel, thus saving the overhead of the kernel-user transition. The user-space run-time system can block the synchronizing thread and schedule a new one by itself.

When scheduler activations are used, the kernel assigns a certain number of virtual processors to each process and lets the (user-space) run-time system allocate threads to processors. This mechanism can also be used on a multiprocessor where the virtual processors may be real CPUs. The number of virtual processors allocated to a process is initially one, but the process can ask for more and can also return processors it no longer needs. The kernel can also take back virtual processors already allocated in order to assign them to more needy processes.

The basic idea that makes this scheme work is that when the kernel knows that a thread has blocked (e.g., by its having executed a blocking system call or caused a page fault), the kernel notifies the process' run-time system, passing as parameters on the stack the number of the thread in question and a description of the event that occurred. The notification happens by having the kernel activate the run-time system at a known starting address, roughly analogous to a signal in UNIX. This mechanism is called an **upcall**.

Once activated, the run-time system can reschedule its threads, typically by marking the current thread as blocked and taking another thread from the ready list, setting up its registers, and restarting it. Later, when the kernel learns that the original thread can run again (e.g., the pipe it was trying to read from now contains data, or the page it faulted over has been brought in from disk), it makes another upcall to the run-time system to inform it. The run-time system can either restart the blocked thread immediately or put it on the ready list to be run later.

When a hardware interrupt occurs while a user thread is running, the interrupted CPU switches into kernel mode. If the interrupt is caused by an event not of interest to the interrupted process, such as completion of another process' I/O, when the interrupt handler has finished, it puts the interrupted thread back in the state it was in before the interrupt. If, however, the process is interested in the interrupt, such as the arrival of a page needed by one of the process' threads, the interrupted thread is not restarted. Instead, it is suspended, and the run-time system is started on that virtual CPU, with the state of the interrupted thread on the stack. It is then up to the run-time system to decide which thread to schedule on that CPU: the interrupted one, the newly ready one, or some third choice.

An objection to scheduler activations is the fundamental reliance on upcalls, a concept that violates the structure inherent in any layered system. Normally, layer n offers certain services that layer $n + 1$ can call on, but layer n may not call procedures in layer $n + 1$. Upcalls do not follow this fundamental principle.

2.2.8 Pop-Up Threads

Threads are frequently useful in distributed systems. An important example is how incoming messages, for example requests for service, are handled. The traditional approach is to have a process or thread that is blocked on a receive system call waiting for an incoming message. When a message arrives, it accepts the message, unpacks it, examines the contents, and processes it.

However, a completely different approach is also possible, in which the arrival of a message causes the system to create a new thread to handle the message. Such a thread is called a **pop-up thread** and is illustrated in Fig. 2-18. A key advantage of pop-up threads is that since they are brand new, they do not have any history—registers, stack, whatever—that must be restored. Each one starts out fresh and each one is identical to all the others. This makes it possible to create such a thread quickly. The new thread is given the incoming message to process. The result of using pop-up threads is that the latency between message arrival and the start of processing can be made very short.

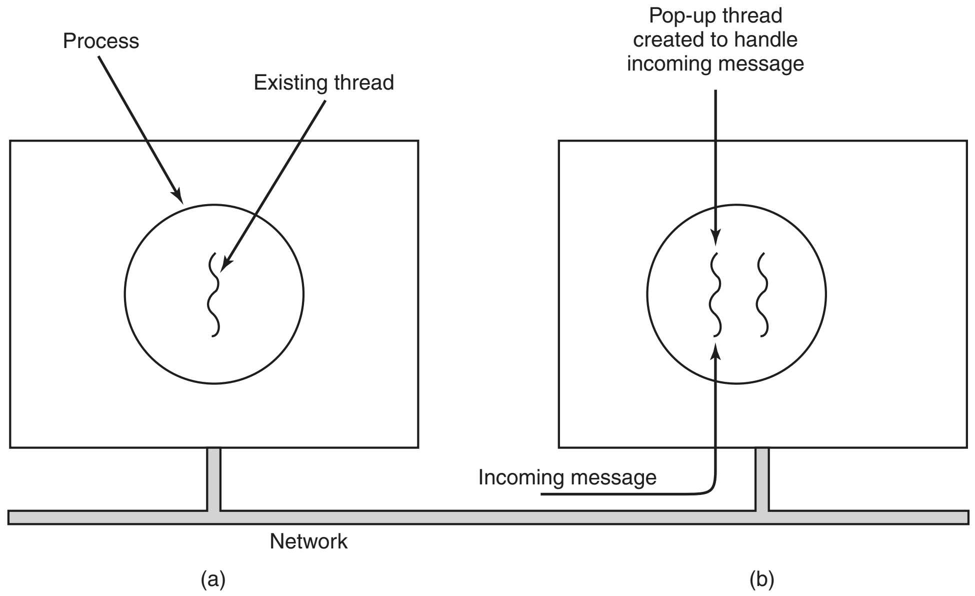


Figure 2-18. Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

Some advance planning is needed when pop-up threads are used. For example, in which process does the thread run? If the system supports threads running in the kernel's context, the thread may run there (which is why we have not shown the kernel in Fig. 2-18). Having the pop-up thread run in kernel space is usually easier and faster than putting it in user space. Also, a pop-up thread in kernel space can easily access all the kernel's tables and the I/O devices, which may be needed for interrupt processing. On the other hand, a buggy kernel thread can do more damage than a buggy user thread. For example, if it runs too long and there is no way to preempt it, incoming data may be permanently lost.

2.2.9 Making Single-Threaded Code Multithreaded

Many existing programs were written for single-threaded processes. Converting these to multithreading is much trickier than it may at first appear. Below we will examine just a few of the pitfalls.

As a start, the code of a thread normally consists of multiple procedures, just like a process. These may have local variables, global variables, and parameters. Local variables and parameters do not cause any trouble, but variables that are global to a thread but not global to the entire program are a problem. These are variables that are global in the sense that many procedures within the thread use them (as they might use any global variable), but other threads should logically leave them alone.

As an example, consider the *errno* variable maintained by UNIX. When a process (or a thread) makes a system call that fails, the error code is put into *errno*. In Fig. 2-19, thread 1 executes the system call access to find out if it has permission to access a certain file. The operating system returns the answer in the global variable *errno*. After control has returned to thread 1, but before it has a chance to read *errno*, the scheduler decides that thread 1 has had enough CPU time for the moment and decides to switch to thread 2. Thread 2 executes an open call that fails, which causes *errno* to be overwritten and thread 1's access code to be lost forever. When thread 1 starts up later, it will read the wrong value and behave incorrectly.

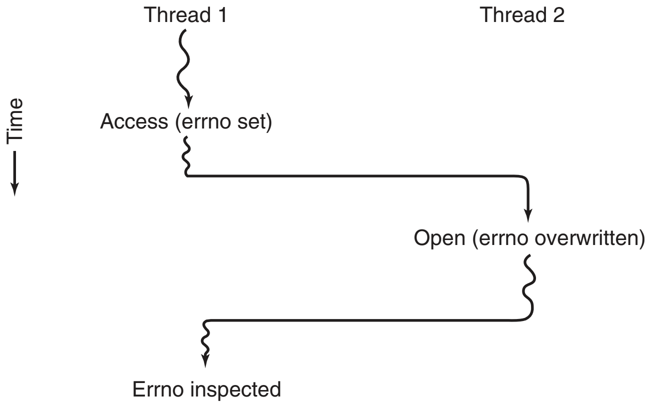


Figure 2-19. Conflicts between threads over the use of a global variable.

Various solutions to this problem are possible. One is to prohibit global variables altogether. However worthy this ideal may be, it conflicts with much existing software. Another is to assign each thread its own private global variables, as shown in Fig. 2-20. In this way, each thread has its own private copy of *errno* and other global variables, so conflicts are avoided. In effect, this decision creates a new scoping level, variables visible to all the procedures of a thread (but not to other threads), in addition to the existing scoping levels of variables visible only to one procedure and variables visible everywhere in the program.

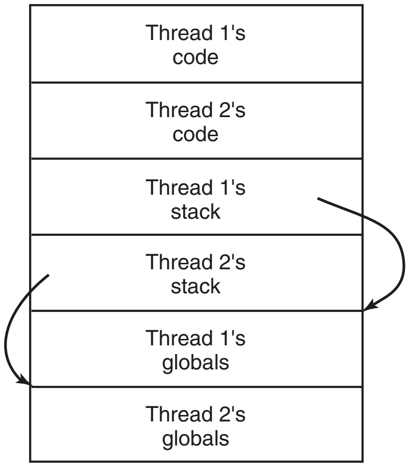


Figure 2-20. Threads can have private global variables.

Accessing the private global variables is a bit tricky, however, since most programming languages have a way of expressing local variables and global variables, but not intermediate forms. It is possible to allocate a chunk of memory for the globals and pass it to each procedure in the thread as an extra parameter. While hardly an elegant solution, it works.

Alternatively, new library procedures can be introduced to create, set, and read these threadwide global variables. The first call might look like this:

```
create_global("bufptr");
```

It allocates storage for a pointer called *bufptr* on the heap or in a special storage area reserved for the calling thread. No matter where the storage is allocated, only the calling thread has access to the global variable. If another thread creates a global variable with the same name, it gets a different storage location that does not conflict with the existing one.

Two calls are needed to access global variables: one for writing them and the other for reading them. For writing, something like

```
set_global("bufptr", &buf);
```

will do. It stores the value of a pointer in the storage location previously created by the call to *create_global*. To read a global variable, the call might look like

```
bufptr = read_global("bufptr");
```

It returns the address stored in the global variable, so its data can be accessed.

The next problem in turning a single-threaded program into a multithreaded one is that many library procedures are not reentrant. That is, they were not designed to have a second call made to any given procedure while a previous call has not yet finished. For example, sending a message over the network may well be programmed to assemble the message in a fixed buffer within the library, then to trap to the kernel to send it. What happens if one thread has assembled its message in the buffer, then a clock interrupt forces a switch to a second thread that immediately overwrites the buffer with its own message?

Similarly, memory-allocation procedures such as *malloc* in UNIX, maintain crucial tables about memory usage, for example, a linked list of available chunks of memory. While *malloc* is busy updating these lists, they may temporarily be in an inconsistent state, with pointers that point nowhere. If a thread switch occurs while the tables are inconsistent and a new call comes in from a different thread, an invalid pointer may be used, leading to a program crash. Fixing all these problems effectively means rewriting the entire library. Doing so is a nontrivial activity with a real possibility of introducing subtle errors.

A different solution is to provide each procedure with a jacket that sets a bit to mark the library as in use. Any attempt for another thread to use a library procedure while a previous call has not yet completed is blocked. Although this approach can be made to work, it greatly eliminates potential parallelism.

Next, consider signals. Some signals are logically thread specific, whereas others are not. For example, if a thread calls alarm, it makes sense for the resulting signal to go to the thread that made the call. However, when threads are implemented entirely in user space, the kernel does not even know about threads and can hardly direct the signal to the right one. An additional complication occurs if a process may only have one alarm pending at a time and several threads call alarm independently.

Other signals, such as keyboard interrupt, are not thread specific. Who should catch them? One designated thread? All the threads? A newly created pop-up thread? Furthermore, what happens if one thread changes the signal handlers without telling other threads? And what happens if one thread wants to catch a particular signal (say, the user hitting CTRL-C), and another thread wants this signal to terminate the process? This situation can arise if one or more threads run standard library procedures and others are user-written. Clearly, these wishes are incompatible. In general, signals are difficult enough to manage in a single-threaded environment. Going to a multithreaded environment does not make them any easier to handle.

One last problem introduced by threads is stack management. In many systems, when a process' stack overflows, the kernel just provides that process with more stack automatically. When a process has multiple threads, it must also have multiple stacks. If the kernel is not aware of all these stacks, it cannot grow them automatically upon stack fault. In fact, it may not even realize that a memory fault is related to the growth of some thread's stack.

These problems are certainly not insurmountable, but they do show that just introducing threads into an existing system without a fairly substantial system redesign is not going to work at all. The semantics of system calls may have to be redefined and libraries rewritten, at the very least. And all of these things must be done in such a way as to remain backward compatible with existing programs for the limiting case of a process with only one thread. For additional information about threads, see Hauser et al. (1993), Marsh et al. (1991), and Rodrigues et al. (2010).

2.3 INTERPROCESS COMMUNICATION

Processes frequently need to communicate with other processes. For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line. Thus there is a need for communication between processes, preferably in a well-structured way not using interrupts. In the following sections we will look at some of the issues related to this **InterProcess Communication**, or **IPC**.

Very briefly, there are three issues here. The first was alluded to above: how one process can pass information to another. The second has to do with making sure two or more processes do not get in each other's way, for example, two processes in an airline reservation system each trying to grab the last seat on a plane for a different customer. The third concerns proper sequencing when dependencies are present: if process *A* produces data and process *B* prints them, *B* has to wait until *A* has produced some data before starting to print. We will examine all three of these issues starting in the next section.

It is also important to mention that two of these issues apply equally well to threads. The first one—passing information—is easy for threads since they share a common address space (threads in different address spaces that need to communicate fall under the heading of communicating processes). However, the other two—keeping out of each other's hair and proper sequencing—apply equally well to threads. The same problems exist and the same solutions apply. Below we will discuss the problem in the context of processes, but please keep in mind that the same problems and solutions also apply to threads.

2.3.1 Race Conditions

In some operating systems, processes that are working together may share some common storage that each one can read and write. The shared storage may be in main memory (possibly in a kernel data structure) or it may be a shared file; the location of the shared memory does not change the nature of the communication or the problems that arise. To see how interprocess communication works in practice, let us now consider a simple but common example: a print spooler. When a process wants to print a file, it enters the file name in a special **spooler directory**. Another process, the **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.

Imagine that our spooler directory has a very large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables, *out*, which points to the next file to be printed, and *in*, which points to the next free slot in the directory. These two variables might well be kept in a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes *A* and *B* decide they want to queue a file for printing. This situation is shown in Fig. 2-21.

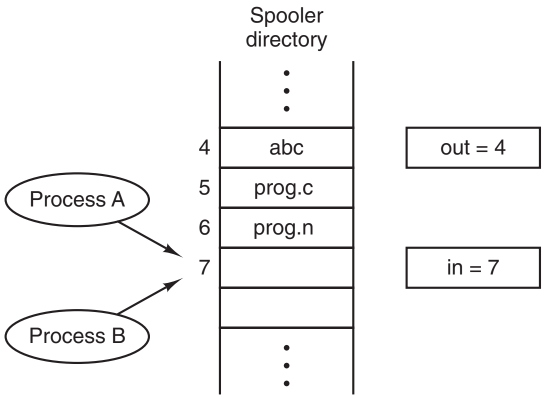


Figure 2-21. Two processes want to access shared memory at the same time.

In jurisdictions where Murphy's law† is applicable, the following could happen. Process *A* reads *in* and stores the value, 7, in a local variable called *next_free_slot*. Just then a clock interrupt occurs and the CPU decides that process *A* has run long enough, so it switches to process *B*. Process *B* also reads *in* and also gets a 7. It, too, stores it in *its* local variable *next_free_slot*. At this instant both processes think that the next available slot is 7.

Process *B* now continues to run. It stores the name of its file in slot 7 and updates *in* to be an 8. Then it goes off and does other things.

Eventually, process *A* runs again, starting from the place it left off. It looks at *next_free_slot*, finds a 7 there, and writes its file name in slot 7, erasing the name that process *B* just put there. Then it computes *next_free_slot* + 1, which is 8, and sets *in* to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process *B* will never receive any output. User *B* will hang around the printer for years, wistfully hoping for output that

† If something can go wrong, it will.

never comes. Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race conditions**. Debugging programs containing race conditions is no fun at all. The results of most test runs are fine, but once in a blue moon something weird and unexplained happens. Unfortunately, with increasing parallelism due to increasing numbers of cores, race condition are becoming more common.

2.3.2 Critical Regions

How do we avoid race conditions? The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time. Put in other words, what we need is **mutual exclusion**, that is, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing. The difficulty above occurred because process *B* started using one of the shared variables before process *A* was finished with it. The choice of appropriate primitive operations for achieving mutual exclusion is a major design issue in any operating system, and a subject that we will examine in great detail in the following sections.

The problem of avoiding race conditions can also be formulated in an abstract way. Part of the time, a process is busy doing internal computations and other things that do not lead to race conditions. However, sometimes a process has to access shared memory or files, or do other critical things that can lead to races. That part of the program where the shared memory is accessed is called the **critical region** or **critical section**. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races.

Although this requirement avoids race conditions, it is not sufficient for having parallel processes cooperate correctly and efficiently using shared data. We need four conditions to hold to have a good solution:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block any process.
4. No process should have to wait forever to enter its critical region.

In an abstract sense, the behavior that we want is shown in Fig. 2-22. Here process *A* enters its critical region at time T_1. A little later, at time T_2 process *B* attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, *B* is temporarily suspended until time T_3 when *A* leaves its critical region, allowing *B* to enter immediately. Eventually *B* leaves (at T_4) and we are back to the original situation with no processes in their critical regions.

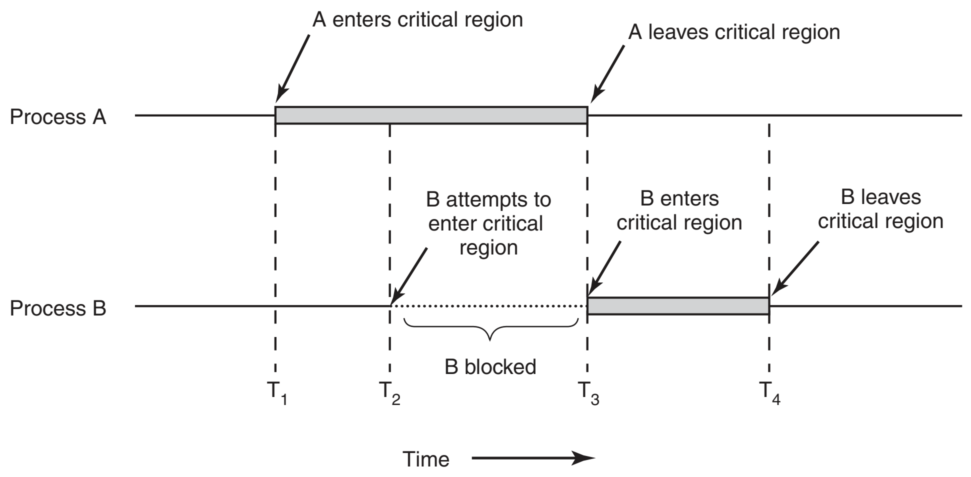


Figure 2-22. Mutual exclusion using critical regions.

2.3.3 Mutual Exclusion with Busy Waiting

In this section we will examine various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter *its* critical region and cause trouble.

Disabling Interrupts

On a single-processor system, the simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. What if one of them did it, and never turned them on again? That could be the end of the system. Furthermore, if the system is a multiprocessor (with two or more CPUs) disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

On the other hand, it is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or especially lists. If an interrupt occurrs while the list of ready processes, for example, is in an inconsistent state, race conditions could occur. The conclusion is: disabling interrupts is often a useful technique within the operating system itself but is not appropriate as a general mutual exclusion mechanism for user processes.

The possibility of achieving mutual exclusion by disabling interrupts—even within the kernel—is becoming less every day due to the increasing number of multicore chips even in low-end PCs. Two cores are already common, four are present in many machines, and eight, 16, or 32 are not far behind. In a multicore (i.e., multiprocessor system) disabling the interrupts of one CPU does not prevent other CPUs from interfering with operations the first CPU is performing. Consequently, more sophisticated schemes are needed.

Lock Variables

As a second attempt, let us look for a software solution. Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

Now you might think that we could get around this problem by first reading out the lock value, then checking it again just before storing into it, but that really does not help. The race now occurs if the second process modifies the lock just after the first process has finished its second check.

Strict Alternation

A third approach to the mutual exclusion problem is shown in Fig. 2-23. This program fragment, like nearly all the others in this book, is written in C. C was chosen here because real operating systems are virtually always written in C (or occasionally C++), but hardly ever in languages like Java, Python, or Haskell. C is powerful, efficient, and predictable, characteristics critical for writing operating systems. Java, for example, is not predictable because it might run out of storage at a critical moment and need to invoke the garbage collector to reclaim memory at a most inopportune time. This cannot happen in C because there is no garbage collection in C. A quantitative comparison of C, C++, Java, and four other languages is given by Prechelt (2000).

In Fig. 2-23, the integer variable *turn*, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects *turn*, finds it to be 0, and enters its critical region. Process 1 also

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}

                (a)
```

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}

                (b)
```

Figure 2-23. A proposed solution to the critical-region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

finds it to be 0 and therefore sits in a tight loop continually testing *turn* to see when it becomes 1. Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a **spin lock**.

When process 0 leaves the critical region, it sets *turn* to 1, to allow process 1 to enter its critical region. Suppose that process 1 finishes its critical region quickly, so that both processes are in their noncritical regions, with *turn* set to 0. Now process 0 executes its whole loop quickly, exiting its critical region and setting *turn* to 1. At this point *turn* is 1 and both processes are executing in their noncritical regions.

Suddenly, process 0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now, because *turn* is 1 and process 1 is busy with its noncritical region. It hangs in its while loop until process 1 sets *turn* to 0. Put differently, taking turns is not a good idea when one of the processes is much slower than the other.

This situation violates condition 3 set out above: process 0 is being blocked by a process not in its critical region. Going back to the spooler directory discussed above, if we now associate the critical region with reading and writing the spooler directory, process 0 would not be allowed to print another file because process 1 was doing something else.

In fact, this solution requires that the two processes strictly alternate in entering their critical regions, for example, in spooling files. Neither one would be permitted to spool two in a row. While this algorithm does avoid all races, it is not really a serious candidate as a solution because it violates condition 3.

Peterson's Solution

By combining the idea of taking turns with the idea of lock variables and warning variables, a Dutch mathematician, T. Dekker, was the first one to devise a software solution to the mutual exclusion problem that does not require strict alternation. For a discussion of Dekker's algorithm, see Dijkstra (1965).

In 1981, G. L. Peterson discovered a much simpler way to achieve mutual exclusion, thus rendering Dekker's solution obsolete. Peterson's algorithm is shown in Fig. 2-24. This algorithm consists of two procedures written in ANSI C, which means that function prototypes should be supplied for all the functions defined and used. However, to save space, we will not show prototypes here or later.

```
#define FALSE  0
#define TRUE   1
#define N      2                         /* number of processes */

int turn;                                /* whose turn is it? */
int interested[N];                       /* all values initially 0 (FALSE) */

void enter_region(int process);          /* process is 0 or 1 */
{
     int other;                          /* number of the other process */

     other = 1 − process;                /* the opposite of process */
     interested[process] = TRUE;         /* show that you are interested */
     turn = process;                     /* set flag */
     while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)           /* process: who is leaving */
{
     interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Before using the shared variables (i.e., before entering its critical region), each process calls *enter_region* with its own process number, 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *leave_region* to indicate that it is done and to allow the other process to enter, if it so desires.

Let us see how this solution works. Initially neither process is in its critical region. Now process 0 calls *enter_region*. It indicates its interest by setting its array element and sets *turn* to 0. Since process 1 is not interested, *enter_region* returns immediately. If process 1 now makes a call to *enter_region*, it will hang there until *interested*[0] goes to *FALSE*, an event that happens only when process 0 calls *leave_region* to exit the critical region.

Now consider the case that both processes call *enter_region* almost simultaneously. Both will store their process number in *turn*. Whichever store is done last is the one that counts; the first one is overwritten and lost. Suppose that process 1 stores last, so *turn* is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region until process 0 exits its critical region.

The TSL Instruction

Now let us look at a proposal that requires a little help from the hardware. Some computers, especially those designed with multiple processors in mind, have an instruction like

```
TSL RX,LOCK
```

(Test and Set Lock) that works as follows. It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock*. The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

It is important to note that locking the memory bus is very different from disabling interrupts. Disabling interrupts then performing a read on a memory word followed by a write does not prevent a second processor on the bus from accessing the word between the read and the write. In fact, disabling interrupts on processor 1 has no effect at all on processor 2. The only way to keep processor 2 out of the memory until processor 1 is finished is to lock the bus, which requires a special hardware facility (basically, a bus line asserting that the bus is locked and not available to processors other than the one that locked it).

To use the TSL instruction, we will use a shared variable, *lock*, to coordinate access to shared memory. When *lock* is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets *lock* back to 0 using an ordinary move instruction.

How can this instruction be used to prevent two processes from simultaneously entering their critical regions? The solution is given in Fig. 2-25. There a four-instruction subroutine in a fictitious (but typical) assembly language is shown. The first instruction copies the old value of *lock* to the register and then sets *lock* to 1. Then the old value is compared with 0. If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again. Sooner or later it will become 0 (when the process currently in its critical region is done with its critical region), and the subroutine returns, with the lock set. Clearing the lock is very simple. The program just stores a 0 in *lock*. No special synchronization instructions are needed.

One solution to the critical-region problem is now easy. Before entering its critical region, a process calls *enter_region*, which does busy waiting until the lock is free; then it acquires the lock and returns. After leaving the critical region the process calls *leave_region*, which stores a 0 in *lock*. As with all solutions based on critical regions, the processes must call *enter_region* and *leave_region* at the correct times for the method to work. If one process cheats, the mutual exclusion will fail. In other words, critical regions work only if the processes cooperate.

```
enter_region:
      TSL REGISTER,LOCK                        | copy lock to register and set lock to 1
      CMP REGISTER,#0                          | was lock zero?
      JNE enter_region                         | if it was not zero, lock was set, so loop
      RET                                      | return to caller; critical region entered


leave_region:
      MOVE LOCK,#0                             | store a 0 in lock
      RET                                      | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

An alternative instruction to TSL is XCHG, which exchanges the contents of two locations atomically, for example, a register and a memory word. The code is shown in Fig. 2-26, and, as can be seen, is essentially the same as the solution with TSL. All Intel x86 CPUs use XCHG instruction for low-level synchronization.

```
enter_region:
      MOVE REGISTER,#1                         | put a 1 in the register
      XCHG REGISTER,LOCK                       | swap the contents of the register and lock variable
      CMP REGISTER,#0                          | was lock zero?
      JNE enter_region                         | if it was non zero, lock was set, so loop
      RET                                      | return to caller; critical region entered


leave_region:
      MOVE LOCK,#0                             | store a 0 in lock
      RET                                      | return to caller
```

Figure 2-26. Entering and leaving a critical region using the XCHG instruction.

2.3.4 Sleep and Wakeup

Both Peterson's solution and the solutions using TSL or XCHG are correct, but both have the defect of requiring busy waiting. In essence, what these solutions do is this: when a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is.

Not only does this approach waste CPU time, but it can also have unexpected effects. Consider a computer with two processes, *H*, with high priority, and *L*, with low priority. The scheduling rules are such that *H* is run whenever it is in ready state. At a certain moment, with *L* in its critical region, *H* becomes ready to run (e.g., an I/O operation completes). *H* now begins busy waiting, but since *L* is never scheduled while *H* is running, *L* never gets the chance to leave its critical region, so *H* loops forever. This situation is sometimes referred to as the **priority inversion problem**.

Now let us look at some interprocess communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions. One of the simplest is the pair sleep and wakeup. Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened. Alternatively, both sleep and wakeup each have one parameter, a memory address used to match up sleeps with wakeups.

The Producer-Consumer Problem

As an example of how these primitives can be used, let us consider the **producer-consumer** problem (also known as the **bounded-buffer** problem). Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out. (It is also possible to generalize the problem to have *m* producers and *n* consumers, but we will consider only the case of one producer and one consumer because this assumption simplifies the solutions.)

Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

This approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier with the spooler directory. To keep track of the number of items in the buffer, we will need a variable, *count*. If the maximum number of items the buffer can hold is *N*, the producer's code will first test to see if *count* is *N*. If it is, the producer will go to sleep; if it is not, the producer will add an item and increment *count*.

The consumer's code is similar: first test *count* to see if it is 0. If it is, go to sleep; if it is nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be awakened, and if so, wakes it up. The code for both producer and consumer is shown in Fig. 2-27.

To express system calls such as sleep and wakeup in C, we will show them as calls to library routines. They are not part of the standard C library but presumably would be made available on any system that actually had these system calls. The procedures *insert_item* and *remove_item*, which are not shown, handle the bookkeeping of putting items into the buffer and taking items out of the buffer.

Now let us get back to the race condition. It can occur because access to *count* is unconstrained. As a consequence, the following situation could possibly occur. The buffer is empty and the consumer has just read *count* to see if it is 0. At that

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
      int item;

      while (TRUE) {                            /* repeat forever */
            item = produce_item( );             /* generate next item */
            if (count == N) sleep( );           /* if buffer is full, go to sleep */
            insert_item(item);                  /* put item in buffer */
            count = count + 1;                  /* increment count of items in buffer */
            if (count == 1) wakeup(consumer);   /* was buffer empty? */
      }
}


void consumer(void)
{
      int item;

      while (TRUE) {                            /* repeat forever */
            if (count == 0) sleep( );           /* if buffer is empty, got to sleep */
            item = remove_item( );              /* take item out of buffer */
            count = count − 1;                  /* decrement count of items in buffer */
            if (count == N − 1) wakeup(producer); /* was buffer full? */
            consume_item(item);                 /* print item */
      }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

instant, the scheduler decides to stop running the consumer temporarily and start running the producer. The producer inserts an item in the buffer, increments *count*, and notices that it is now 1. Reasoning that *count* was just 0, and thus the consumer must be sleeping, the producer calls *wakeup* to wake the consumer up.

Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of *count* it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. If it were not lost, everything would work. A quick fix is to modify the rules to add a **wakeup waiting bit** to the picture. When a wakeup is sent to a process that is still awake, this bit is set. Later, when the process tries to go to sleep, if the wakeup waiting bit is on, it will be turned off, but the process will stay awake. The wakeup waiting bit is a piggy bank for storing wakeup signals. The consumer clears the wakeup waiting bit in every iteration of the loop.

While the wakeup waiting bit saves the day in this simple example, it is easy to construct examples with three or more processes in which one wakeup waiting bit is insufficient. We could make another patch and add a second wakeup waiting bit, or maybe 8 or 32 of them, but in principle the problem is still there.

2.3.5 Semaphores

This was the situation in 1965, when E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, which he called a **semaphore**, was introduced. A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.

Dijkstra proposed having two operations on semaphores, now usually called down and up (generalizations of sleep and wakeup, respectively). The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues. If the value is 0, the process is put to sleep without completing the down for the moment. Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible **atomic action**. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions. Atomic actions, in which a group of related operations are either all performed without interruption or not performed at all, are extremely important in many other areas of computer science as well.

The up operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down. Thus, after an up on a semaphore with processes sleeping on it, the semaphore will still be 0, but there will be one fewer process sleeping on it. The operation of incrementing the semaphore and waking up one process is also indivisible. No process ever blocks doing an up, just as no process ever blocks doing a wakeup in the earlier model.

As an aside, in Dijkstra's original paper, he used the names P and V instead of down and up, respectively. Since these have no mnemonic significance to people who do not speak Dutch and only marginal significance to those who do—*Proberen* (try) and *Verhogen* (raise, make higher)—we will use the terms down and up instead. These were first introduced in the Algol 68 programming language.

Solving the Producer-Consumer Problem Using Semaphores

Semaphores solve the lost-wakeup problem, as shown in Fig. 2-28. To make them work correctly, it is essential that they be implemented in an indivisible way. The normal way is to implement up and down as system calls, with the operating system briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep, if necessary. As all of these actions take only a few instructions, no harm is done in disabling interrupts. If multiple CPUs are being used, each semaphore should be protected by a lock variable, with the TSL or XCHG instructions used to make sure that only one CPU at a time examines the semaphore.

Be sure you understand that using TSL or XCHG to prevent several CPUs from accessing the semaphore at the same time is quite different from the producer or consumer busy waiting for the other to empty or fill the buffer. The semaphore operation will take only a few microseconds, whereas the producer or consumer might take arbitrarily long.

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

This solution uses three semaphores: one called *full* for counting the number of slots that are full, one called *empty* for counting the number of slots that are empty, and one called *mutex* to make sure the producer and consumer do not access the buffer at the same time. *Full* is initially 0, *empty* is initially equal to the number of slots in the buffer, and *mutex* is initially 1. Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores**. If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed.

Now that we have a good interprocess communication primitive at our disposal, let us go back and look at the interrupt sequence of Fig. 2-5 again. In a system using semaphores, the natural way to hide interrupts is to have a semaphore, initially set to 0, associated with each I/O device. Just after starting an I/O device, the managing process does a down on the associated semaphore, thus blocking immediately. When the interrupt comes in, the interrupt handler then does an up on the associated semaphore, which makes the relevant process ready to run again. In this model, step 5 in Fig. 2-5 consists of doing an up on the device's semaphore, so that in step 6 the scheduler will be able to run the device manager. Of course, if several processes are now ready, the scheduler may choose to run an even more important process next. We will look at some of the algorithms used for scheduling later on in this chapter.

In the example of Fig. 2-28, we have actually used semaphores in two different ways. This difference is important enough to make explicit. The *mutex* semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables. This mutual exclusion is required to prevent chaos. We will study mutual exclusion and how to achieve it in the next section.

The other use of semaphores is for **synchronization**. The *full* and *empty* semaphores are needed to guarantee that certain event sequences do or do not occur. In this case, they ensure that the producer stops running when the buffer is full, and that the consumer stops running when it is empty. This use is different from mutual exclusion.

2.3.6 Mutexes

When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used. Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space.

A **mutex** is a shared variable that can be in one of two states: unlocked or locked. Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked.

Two procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls *mutex_lock*. If the mutex is currently unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.

On the other hand, if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls *mutex_unlock*. If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

Because mutexes are so simple, they can easily be implemented in user space provided that a TSL or XCHG instruction is available. The code for *mutex_lock* and *mutex_unlock* for use with a user-level threads package are shown in Fig. 2-29. The solution with XCHG is essentially the same.

```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again
ok:     RET                         | return to caller; critical region entered


mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET                         | return to caller
```

Figure 2-29. Implementation of *mutex_lock* and *mutex_unlock*.

The code of *mutex_lock* is similar to the code of *enter_region* of Fig. 2-25 but with a crucial difference. When *enter_region* fails to enter the critical region, it keeps testing the lock repeatedly (busy waiting). Eventually, the clock runs out and some other process is scheduled to run. Sooner or later the process holding the lock gets to run and releases it.

With (user) threads, the situation is different because there is no clock that stops threads that have run too long. Consequently, a thread that tries to acquire a lock by busy waiting will loop forever and never acquire the lock because it never allows any other thread to run and release the lock.

That is where the difference between *enter_region* and *mutex_lock* comes in. When the later fails to acquire a lock, it calls *thread_yield* to give up the CPU to another thread. Consequently there is no busy waiting. When the thread runs the next time, it tests the lock again.

Since *thread_yield* is just a call to the thread scheduler in user space, it is very fast. As a consequence, neither *mutex_lock* nor *mutex_unlock* requires any kernel calls. Using them, user-level threads can synchronize entirely in user space using procedures that require only a handful of instructions.

The mutex system that we have described above is a bare-bones set of calls. With all software, there is always a demand for more features, and synchronization primitives are no exception. For example, sometimes a thread package offers a call *mutex_trylock* that either acquires the lock or returns a code for failure, but does not block. This call gives the thread the flexibility to decide what to do next if there are alternatives to just waiting.

There is a subtle issue that up until now we have glossed over but which is worth at least making explicit. With a user-space threads package there is no problem with multiple threads having access to the same mutex, since all the threads operate in a common address space. However, with most of the earlier solutions, such as Peterson's algorithm and semaphores, there is an unspoken assumption that multiple processes have access to at least some shared memory, perhaps only one word, but something. If processes have disjoint address spaces, as we have consistently said, how can they share the *turn* variable in Peterson's algorithm, or semaphores or a common buffer?

There are two answers. First, some of the shared data structures, such as the semaphores, can be stored in the kernel and accessed only by means of system calls. This approach eliminates the problem. Second, most modern operating systems (including UNIX and Windows) offer a way for processes to share some portion of their address space with other processes. In this way, buffers and other data structures can be shared. In the worst case, that nothing else is possible, a shared file can be used.

If two or more processes share most or all of their address spaces, the distinction between processes and threads becomes somewhat blurred but is nevertheless present. Two processes that share a common address space still have different open files, alarm timers, and other per-process properties, whereas the threads within a single process share them. And it is always true that multiple processes sharing a common address space never have the efficiency of user-level threads since the kernel is deeply involved in their management.

Futexes

With increasing parallelism, efficient synchronization and locking is very important for performance. Spin locks are fast if the wait is short, but waste CPU cycles if not. If there is much contention, it is therefore more efficient to block the process and let the kernel unblock it only when the lock is free. Unfortunately, this has the inverse problem: it works well under heavy contention, but continuously switching to the kernel is expensive if there is very little contention to begin with. To make matters worse, it may not be easy to predict the amount of lock contention.

One interesting solution that tries to combine the best of both worlds is known as **futex**, or "fast user space mutex." A futex is a feature of Linux that implements basic locking (much like a mutex) but avoids dropping into the kernel unless it really has to. Since switching to the kernel and back is quite expensive, doing so improves performance considerably. A futex consists of two parts: a kernel service and a user library. The kernel service provides a "wait queue" that allows multiple processes to wait on a lock. They will not run, unless the kernel explicitly unblocks them. For a process to be put on the wait queue requires an (expensive) system call and should be avoided. In the absence of contention, therefore, the futex works completely in user space. Specifically, the processes share a common lock variable—a fancy name for an aligned 32-bit integer that serves as the lock. Suppose the lock is initially 1—which we assume to mean that the lock is free. A thread grabs the lock by performing an atomic "decrement and test" (atomic functions in Linux consist of inline assembly wrapped in C functions and are defined in header files). Next, the thread inspects the result to see whether or not the lock was free. If it was not in the locked state, all is well and our thread has successfully grabbed the lock. However, if the lock is held by another thread, our thread has to wait. In that case, the futex library does not spin, but uses a system call to put the thread on the wait queue in the kernel. Hopefully, the cost of the switch to the kernel is now justified, because the thread was blocked anyway. When a thread is done with the lock, it releases the lock with an atomic "increment and test" and checks the result to see if any processes are still blocked on the kernel wait queue. If so, it will let the kernel know that it may unblock one or more of these processes. If there is no contention, the kernel is not involved at all.

Mutexes in Pthreads

Pthreads provides a number of functions that can be used to synchronize threads. The basic mechanism uses a mutex variable, which can be locked or unlocked, to guard each critical region. A thread wishing to enter a critical region first tries to lock the associated mutex. If the mutex is unlocked, the thread can enter immediately and the lock is atomically set, preventing other threads from entering. If the mutex is already locked, the calling thread is blocked until it is unlocked. If multiple threads are waiting on the same mutex, when it is unlocked, only one of them is allowed to continue and relock it. These locks are not mandatory. It is up to the programmer to make sure threads use them correctly.

The major calls relating to mutexes are shown in Fig. 2-30. As expected, mutexes can be created and destroyed. The calls for performing these operations are *pthread_mutex_init* and *pthread_mutex_destroy*, respectively. They can also be locked—by *pthread_mutex_lock*—which tries to acquire the lock and blocks if is already locked. There is also an option for trying to lock a mutex and failing with an error code instead of blocking if it is already blocked. This call is *pthread_mutex_trylock*. This call allows a thread to effectively do busy waiting if that is ever needed. Finally, *pthread_mutex_unlock* unlocks a mutex and releases exactly one thread if one or more are waiting on it. Mutexes can also have attributes, but these are used only for specialized purposes.

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure 2-30. Some of the Pthreads calls relating to mutexes.

In addition to mutexes, Pthreads offers a second synchronization mechanism: **condition variables**. Mutexes are good for allowing or blocking access to a critical region. Condition variables allow threads to block due to some condition not being met. Almost always the two methods are used together. Let us now look at the interaction of threads, mutexes, and condition variables in a bit more detail.

As a simple example, consider the producer-consumer scenario again: one thread puts things in a buffer and another one takes them out. If the producer discovers that there are no more free slots available in the buffer, it has to block until one becomes available. Mutexes make it possible to do the check atomically without interference from other threads, but having discovered that the buffer is full, the producer needs a way to block and be awakened later. This is what condition variables allow.

The most important calls related to condition variables are shown in Fig. 2-31. As you would probably expect, there are calls to create and destroy condition variables. They can have attributes and there are various calls for managing them (not shown). The primary operations on condition variables are *pthread_cond_wait* and *pthread_cond_signal*. The former blocks the calling thread until some other thread signals it (using the latter call). The reasons for blocking and waiting are not part of the waiting and signaling protocol, of course. The blocking thread often is waiting for the signaling thread to do some work, release some resource, or perform some other activity. Only then can the blocking thread continue. The condition variables allow this waiting and blocking to be done atomically. The *pthread_cond_broadcast* call is used when there are multiple threads potentially all blocked and waiting for the same signal.

Condition variables and mutexes are always used together. The pattern is for one thread to lock a mutex, then wait on a conditional variable when it cannot get what it needs. Eventually another thread will signal it and it can continue. The *pthread_cond_wait* call atomically unlocks the mutex it is holding. For this reason, the mutex is one of the parameters.

It is also worth noting that condition variables (unlike semaphores) have no memory. If a signal is sent to a condition variable on which no thread is waiting, the signal is lost. Programmers have to be careful not to lose signals.

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Figure 2-31. Some of the Pthreads calls relating to condition variables.

As an example of how mutexes and condition variables are used, Fig. 2-32 shows a very simple producer-consumer problem with a single buffer. When the producer has filled the buffer, it must wait until the consumer empties it before producing the next item. Similarly, when the consumer has removed an item, it must wait until the producer has produced another one. While very simple, this example illustrates the basic mechanisms. The statement that puts a thread to sleep should always check the condition to make sure it is satisfied before continuing, as the thread might have been awakened due to a UNIX signal or some other reason.

2.3.7 Monitors

With semaphores and mutexes interprocess communication looks easy, right? Forget it. Look closely at the order of the downs before inserting or removing items from the buffer in Fig. 2-28. Suppose that the two downs in the producer's code were reversed in order, so *mutex* was decremented before *empty* instead of after it. If the buffer were completely full, the producer would block, with *mutex* set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a down on *mutex*, now 0, and block too. Both processes would stay blocked forever and no more work would ever be done. This unfortunate situation is called a deadlock. We will study deadlocks in detail in Chap. 6.

This problem is pointed out to show how careful you must be when using semaphores. One subtle error and everything comes to a grinding halt. It is like programming in assembly language, only worse, because the errors are race conditions, deadlocks, and other forms of unpredictable and irreproducible behavior.

To make it easier to write correct programs, Brinch Hansen (1973) and Hoare (1974) proposed a higher-level synchronization primitive called a **monitor**. Their proposals differed slightly, as described below. A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor. Figure 2-33 illustrates a monitor written in an imaginary language, Pidgin Pascal. C cannot be used here because monitors are a *language* concept and C does not have them.

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

#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* used for signaling */
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                        /* put item in buffer */
                pthread_cond_signal(&condc);       /* wake up consumer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                        /* take item out of buffer */
                pthread_cond_signal(&condp);       /* wake up producer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}

int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Figure 2-32. Using threads to solve the producer-consumer problem.

Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant. Monitors are a programming-language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls. Typically, when a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

It is up to the compiler to implement mutual exclusion on monitor entries, but a common way is to use a mutex or a binary semaphore. Because the compiler, not the programmer, is arranging for the mutual exclusion, it is much less likely that something will go wrong. In any event, the person writing the monitor does not have to be aware of how the compiler arranges for mutual exclusion. It is sufficient to know that by turning all the critical regions into monitor procedures, no two processes will ever execute their critical regions at the same time.

Although monitors provide an easy way to achieve mutual exclusion, as we have seen above, that is not enough. We also need a way for processes to block when they cannot proceed. In the producer-consumer problem, it is easy enough to put all the tests for buffer-full and buffer-empty in monitor procedures, but how should the producer block when it finds the buffer full?

The solution lies in the introduction of **condition variables**, along with two operations on them, wait and signal. When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a wait on some condition variable, say, *full*. This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now. We saw condition variables and these operations in the context of Pthreads earlier.

This other process, for example, the consumer, can wake up its sleeping partner by doing a signal on the condition variable that its partner is waiting on. To avoid having two active processes in the monitor at the same time, we need a rule telling what happens after a signal. Hoare proposed letting the newly awakened process run, suspending the other one. Brinch Hansen proposed finessing the problem by requiring that a process doing a signal *must* exit the monitor immediately. In other words, a signal statement may appear only as the final statement in a monitor procedure. We will use Brinch Hansen's proposal because it is conceptually simpler and is also easier to implement. If a signal is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived.

As an aside, there is also a third solution, not proposed by either Hoare or Brinch Hansen. This is to let the signaler continue to run and allow the waiting process to start running only after the signaler has exited the monitor.

Condition variables are not counters. They do not accumulate signals for later use the way semaphores do. Thus, if a condition variable is signaled with no one

```
monitor example
          integer i;
          condition c;

          procedure producer( );
          .
          .
          .
          end;

          procedure consumer( );
          .          .          .
          end;
end monitor;
```

Figure 2-33. A monitor.

waiting on it, the signal is lost forever. In other words, the wait must come before the signal. This rule makes the implementation much simpler. In practice, it is not a problem because it is easy to keep track of the state of each process with variables, if need be. A process that might otherwise do a signal can see that this operation is not necessary by looking at the variables.

A skeleton of the producer-consumer problem with monitors is given in Fig. 2-34 in an imaginary language, Pidgin Pascal. The advantage of using Pidgin Pascal here is that it is pure and simple and follows the Hoare/Brinch Hansen model exactly.

You may be thinking that the operations wait and signal look similar to sleep and wakeup, which we saw earlier had fatal race conditions. Well, they *are* very similar, but with one crucial difference: sleep and wakeup failed because while one process was trying to go to sleep, the other one was trying to wake it up. With monitors, that cannot happen. The automatic mutual exclusion on monitor procedures guarantees that if, say, the producer inside a monitor procedure discovers that the buffer is full, it will be able to complete the wait operation without having to worry about the possibility that the scheduler may switch to the consumer just before the wait completes. The consumer will not even be let into the monitor at all until the wait is finished and the producer has been marked as no longer runnable.

Although Pidgin Pascal is an imaginary language, some real programming languages also support monitors, although not always in the form designed by Hoare and Brinch Hansen. One such language is Java. Java is an object-oriented language that supports user-level threads and also allows methods (procedures) to be grouped together into classes. By adding the keyword synchronized to a method declaration, Java guarantees that once any thread has started executing that method, no other thread will be allowed to start executing any other synchronized method of that object. Without synchronized, there are no guarantees about interleaving.

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count − 1;
                if count = N − 1 then signal(full)
        end;

        count := 0;
end monitor;

procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has *N* slots.

A solution to the producer-consumer problem using monitors in Java is given in Fig. 2-35. Our solution has four classes. The outer class, *ProducerConsumer*, creates and starts two threads, *p* and *c*. The second and third classes, *producer* and *consumer*, respectively, contain the code for the producer and consumer. Finally, the class *our_monitor*, is the monitor. It contains two synchronized threads that are used for actually inserting items into the shared buffer and taking them out. Unlike the previous examples, here we have the full code of *insert* and *remove*.

```
public class ProducerConsumer {
      static final int N = 100;       // constant giving the buffer size
      static producer p = new producer( );    // instantiate a new producer thread
      static consumer c = new consumer( ); // instantiate a new consumer thread
      static our_monitor mon = new our_monitor( );      // instantiate a new monitor

      public static void main(String args[ ]) {
        p.start( );      // start the producer thread
        c.start( );      // start the consumer thread
      }

      static class producer extends Thread {
        public void run( )  {// run method contains the thread code
          int item;
          while (true) {      // producer loop
            item = produce_item( );
            mon.insert(item);
          }
        }
        private int produce_item( ) { ... }      // actually produce
      }

      static class consumer extends Thread {
        public void run( )  {run method contains the thread code
          int item;
          while (true) {      // consumer loop
            item = mon.remove( );
            consume_item (item);
          }
        }
        private void consume_item(int item) { ... }// actually consume
      }

      static class our_monitor {  // this is a monitor
        private int buffer[ ] = new int[N];
        private int count = 0, lo = 0, hi = 0;  // counters and indices

        public synchronized void insert(int val) {
          if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
          buffer [hi] = val; // insert an item into the buffer
          hi = (hi + 1) % N;        // slot to place next item in
          count = count + 1;      // one more item in the buffer now
          if (count == 1) notify( );        // if consumer was sleeping, wake it up
        }

        public synchronized int remove( ) {
          int val;
          if (count == 0) go_to_sleep( );     // if the buffer is empty, go to sleep
          val = buffer [lo]; // fetch an item from the buffer
          lo = (lo + 1) % N;        // slot to fetch next item from
          count = count – 1;      // one few items in the buffer
          if (count == N – 1) notify( ); // if producer was sleeping, wake it up
          return val;
       }
       private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
      }
}
```

Figure 2-35. A solution to the producer-consumer problem in Java.

The producer and consumer threads are functionally identical to their counterparts in all our previous examples. The producer has an infinite loop generating data and putting it into the common buffer. The consumer has an equally infinite loop taking data out of the common buffer and doing some fun thing with it.

The interesting part of this program is the class *our_monitor*, which holds the buffer, the administration variables, and two synchronized methods. When the producer is active inside *insert*, it knows for sure that the consumer cannot be active inside *remove*, making it safe to update the variables and the buffer without fear of race conditions. The variable *count* keeps track of how many items are in the buffer. It can take on any value from 0 through and including $N - 1$. The variable *lo* is the index of the buffer slot where the next item is to be fetched. Similarly, *hi* is the index of the buffer slot where the next item is to be placed. It is permitted that $lo = hi$, which means that either 0 items or N items are in the buffer. The value of *count* tells which case holds.

Synchronized methods in Java differ from classical monitors in an essential way: Java does not have condition variables built in. Instead, it offers two procedures, *wait* and *notify*, which are the equivalent of *sleep* and *wakeup* except that when they are used inside synchronized methods, they are not subject to race conditions. In theory, the method *wait* can be interrupted, which is what the code surrounding it is all about. Java requires that the exception handling be made explicit. For our purposes, just imagine that *go_to_sleep* is the way to go to sleep.

By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error prone than using semaphores. Nevertheless, they too have some drawbacks. It is not for nothing that our two examples of monitors were in Pidgin Pascal instead of C, as are the other examples in this book. As we said earlier, monitors are a programming-language concept. The compiler must recognize them and arrange for the mutual exclusion somehow or other. C, Pascal, and most other languages do not have monitors, so it is unreasonable to expect their compilers to enforce any mutual exclusion rules. In fact, how could the compiler even know which procedures were in monitors and which were not?

These same languages do not have semaphores either, but adding semaphores is easy: all you need to do is add two short assembly-code routines to the library to issue the up and down system calls. The compilers do not even have to know that they exist. Of course, the operating systems have to know about the semaphores, but at least if you have a semaphore-based operating system, you can still write the user programs for it in C or C++ (or even assembly language if you are masochistic enough). With monitors, you need a language that has them built in.

Another problem with monitors, and also with semaphores, is that they were designed for solving the mutual exclusion problem on one or more CPUs that all have access to a common memory. By putting the semaphores in the shared memory and protecting them with TSL or XCHG instructions, we can avoid races. When we move to a distributed system consisting of multiple CPUs, each with its own private memory and connected by a local area network, these primitives become inapplicable. The conclusion is that semaphores are too low level and monitors are not usable except in a few programming languages. Also, none of the primitives allow information exchange between machines. Something else is needed.

2.3.8 Message Passing

That something else is **message passing**. This method of interprocess communication uses two primitives, send and receive, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as

```
send(destination, &message);
```

and

```
receive(source, &message);
```

The former call sends a message to a given destination and the latter one receives a message from a given source (or from *ANY*, if the receiver does not care). If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

Design Issues for Message-Passing Systems

Message-passing systems have many problems and design issues that do not arise with semaphores or with monitors, especially if the communicating processes are on different machines connected by a network. For example, messages can be lost by the network. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special **acknowledgement** message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.

Now consider what happens if the message is received correctly, but the acknowledgement back to the sender is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver be able to distinguish a new message from the retransmission of an old one. Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored. Successfully communicating in the face of unreliable message passing is a major part of the study of computer networks. For more information, see Tanenbaum and Wetherall (2010).

Message systems also have to deal with the question of how processes are named, so that the process specified in a send or receive call is unambiguous. **Authentication** is also an issue in message systems: how can the client tell that it is communicating with the real file server, and not with an imposter?

At the other end of the spectrum, there are also design issues that are important when the sender and receiver are on the same machine. One of these is performance. Copying messages from one process to another is always slower than doing a semaphore operation or entering a monitor. Much work has gone into making message passing efficient.

The Producer-Consumer Problem with Message Passing

Now let us see how the producer-consumer problem can be solved with message passing and no shared memory. A solution is given in Fig. 2-36. We assume that all messages are the same size and that messages sent but not yet received are buffered automatically by the operating system. In this solution, a total of *N* messages is used, analogous to the *N* slots in a shared-memory buffer. The consumer starts out by sending *N* empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one. In this way, the total number of messages in the system remains constant in time, so they can be stored in a given amount of memory known in advance.

If the producer works faster than the consumer, all the messages will end up full, waiting for the consumer; the producer will be blocked, waiting for an empty to come back. If the consumer works faster, then the reverse happens: all the messages will be empties waiting for the producer to fill them up; the consumer will be blocked, waiting for a full message.

Many variants are possible with message passing. For starters, let us look at how messages are addressed. One way is to assign each process a unique address and have messages be addressed to processes. A different way is to invent a new data structure, called a **mailbox**. A mailbox is a place to buffer a certain number of messages, typically specified when the mailbox is created. When mailboxes are used, the address parameters in the send and receive calls are mailboxes, not processes. When a process tries to send to a mailbox that is full, it is suspended until a message is removed from that mailbox, making room for a new one.

For the producer-consumer problem, both the producer and consumer would create mailboxes large enough to hold *N* messages. The producer would send messages containing actual data to the consumer's mailbox, and the consumer would send empty messages to the producer's mailbox. When mailboxes are used, the buffering mechanism is clear: the destination mailbox holds messages that have been sent to the destination process but have not yet been accepted.

The other extreme from having mailboxes is to eliminate all buffering. When this approach is taken, if the send is done before the receive, the sending process is blocked until the receive happens, at which time the message can be copied directly from the sender to the receiver, with no buffering. Similarly, if the receive is done first, the receiver is blocked until a send happens. This strategy is often known as a **rendezvous**. It is easier to implement than a buffered message scheme but is less flexible since the sender and receiver are forced to run in lockstep.

```
#define N 100                                  /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                /* message buffer */

     while (TRUE) {
          item = produce_item( );              /* generate something to put in buffer */
          receive(consumer, &m);               /* wait for an empty to arrive */
          build_message(&m, item);             /* construct a message to send */
          send(consumer, &m);                  /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);               /* get message containing item */
          item = extract_item(&m);             /* extract item from message */
          send(producer, &m);                  /* send back empty reply */
          consume_item(item);                  /* do something with the item */
     }
}
```

Figure 2-36. The producer-consumer problem with *N* messages.

Message passing is commonly used in parallel programming systems. One well-known message-passing system, for example, is **MPI** (**Message-Passing Interface**). It is widely used for scientific computing. For more information about it, see for example Gropp et al. (1994), and Snir et al. (1996).

2.3.9 Barriers

Our last synchronization mechanism is intended for groups of processes rather than two-process producer-consumer type situations. Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase. This behavior may be achieved by placing a **barrier** at the end of each phase. When a process reaches the barrier, it is blocked until all processes have reached the barrier. This allows groups of processes to synchronize. Barrier operation is illustrated in Fig. 2-37.

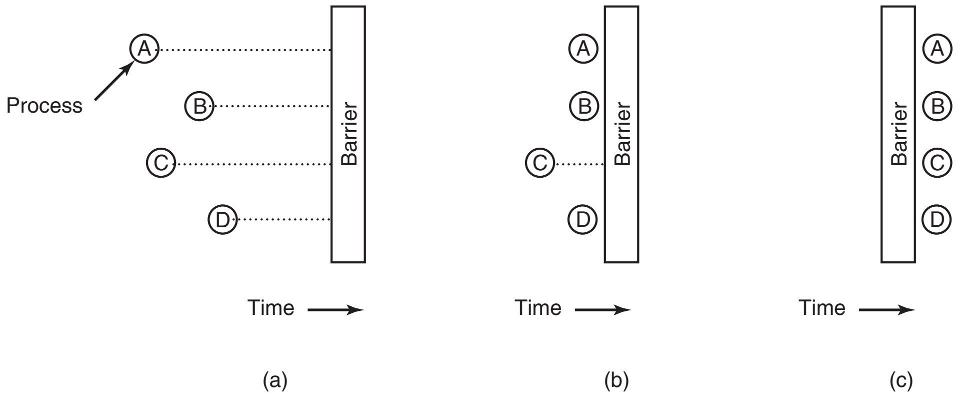


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

In Fig. 2-37(a) we see four processes approaching a barrier. What this means is that they are just computing and have not reached the end of the current phase yet. After a while, the first process finishes all the computing required of it during the first phase. It then executes the barrier primitive, generally by calling a library procedure. The process is then suspended. A little later, a second and then a third process finish the first phase and also execute the barrier primitive. This situation is illustrated in Fig. 2-37(b). Finally, when the last process, *C*, hits the barrier, all the processes are released, as shown in Fig. 2-37(c).

As an example of a problem requiring barriers, consider a common relaxation problem in physics or engineering. There is typically a matrix that contains some initial values. The values might represent temperatures at various points on a sheet of metal. The idea might be to calculate how long it takes for the effect of a flame placed at one corner to propagate throughout the sheet.

Starting with the current values, a transformation is applied to the matrix to get the second version of the matrix, for example, by applying the laws of thermodynamics to see what all the temperatures are ΔT later. Then the process is repeated over and over, giving the temperatures at the sample points as a function of time as the sheet heats up. The algorithm produces a sequence of matrices over time, each one for a given point in time.

Now imagine that the matrix is very large (for example, 1 million by 1 million), so that parallel processes are needed (possibly on a multiprocessor) to speed up the calculation. Different processes work on different parts of the matrix, calculating the new matrix elements from the old ones according to the laws of physics. However, no process may start on iteration $n + 1$ until iteration n is complete, that is, until all processes have finished their current work. The way to achieve this goal is to program each process to execute a barrier operation after it has finished its part of the current iteration. When all of them are done, the new matrix (the input to the next iteration) will be finished, and all processes will be simultaneously released to start the next iteration.

2.3.10 Avoiding Locks: Read-Copy-Update

The fastest locks are no locks at all. The question is whether we can allow for concurrent read and write accesses to shared data structures without locking. In the general case, the answer is clearly no. Imagine process A sorting an array of numbers, while process B is calculating the average. Because A moves the values back and forth across the array, B may encounter some values multiple times and others not at all. The result could be anything, but it would almost certainly be wrong.

In some cases, however, we can allow a writer to update a data structure even though other processes are still using it. The trick is to ensure that each reader either reads the old version of the data, or the new one, but not some weird combination of old and new. As an illustration, consider the tree shown in Fig. 2-38. Readers traverse the tree from the root to its leaves. In the top half of the figure, a new node X is added. To do so, we make the node "just right" before making it visible in the tree: we initialize all values in node X, including its child pointers. Then, with one atomic write, we make X a child of A. No reader will ever read an inconsistent version. In the bottom half of the figure, we subsequently remove B and D. First, we make A's left child pointer point to C. All readers that were in A will continue with node C and never see B or D. In other words, they will see only the new version. Likewise, all readers currently in B or D will continue following the original data structure pointers and see the old version. All is well, and we never need to lock anything. The main reason that the removal of B and D works without locking the data structure, is that **RCU** (**Read-Copy-Update**), decouples the *removal* and *reclamation* phases of the update.

Of course, there is a problem. As long as we are not sure that there are no more readers of B or D, we cannot really free them. But how long should we wait? One minute? Ten? We have to wait until the last reader has left these nodes. RCU carefully determines the maximum time a reader may hold a reference to the data structure. After that period, it can safely reclaim the memory. Specifically, readers access the data structure in what is known as a **read-side critical section** which may contain any code, as long as it does not block or sleep. In that case, we know the maximum time we need to wait. Specifically, we define a **grace period** as any time period in which we know that each thread to be outside the read-side critical section at least once. All will be well if we wait for a duration that is at least equal to the grace period before reclaiming. As the code in a read-side critical section is not allowed to block or sleep, a simple criterion is to wait until all the threads have executed a context switch.

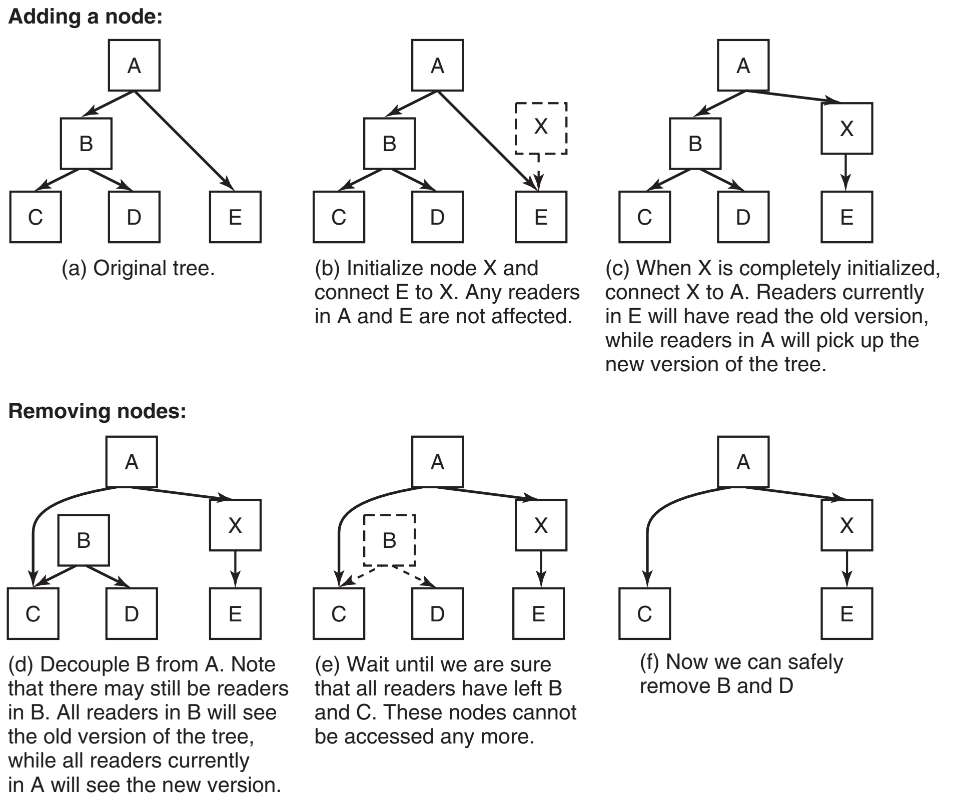

Adding a node:
A
B
C
D
E
X
(a) Original tree.
(b) Initialize node X and connect E to X. Any readers in A and E are not affected.
(c) When X is completely initialized, connect X to A. Readers currently in E will have read the old version, while readers in A will pick up the new version of the tree.
Removing nodes:
(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version.
(e) Wait until we are sure that all readers have left B and C. These nodes cannot be accessed any more.
(f) Now we can safely remove B and D

Figure 2-38. Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks.

2.4 SCHEDULING

When a computer is multiprogrammed, it frequently has multiple processes or threads competing for the CPU at the same time. This situation occurs whenever two or more of them are simultaneously in the ready state. If only one CPU is available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the **scheduler**, and the algorithm it uses is called the **scheduling algorithm**. These topics form the subject matter of the following sections.

Many of the same issues that apply to process scheduling also apply to thread scheduling, although some are different. When the kernel manages threads, scheduling is usually done per thread, with little or no regard to which process the thread belongs. Initially we will focus on scheduling issues that apply to both processes and threads. Later on we will explicitly look at thread scheduling and some of the unique issues it raises. We will deal with multicore chips in Chap. 8.

2.4.1 Introduction to Scheduling

Back in the old days of batch systems with input in the form of card images on a magnetic tape, the scheduling algorithm was simple: just run the next job on the tape. With multiprogramming systems, the scheduling algorithm became more complex because there were generally multiple users waiting for service. Some mainframes still combine batch and timesharing service, requiring the scheduler to decide whether a batch job or an interactive user at a terminal should go next. (As an aside, a batch job may be a request to run multiple programs in succession, but for this section, we will just assume it is a request to run a single program.) Because CPU time is a scarce resource on these machines, a good scheduler can make a big difference in perceived performance and user satisfaction. Consequently, a great deal of work has gone into devising clever and efficient scheduling algorithms.

With the advent of personal computers, the situation changed in two ways. First, most of the time there is only one active process. A user entering a document on a word processor is unlikely to be simultaneously compiling a program in the background. When the user types a command to the word processor, the scheduler does not have to do much work to figure out which process to run—the word processor is the only candidate.

Second, computers have gotten so much faster over the years that the CPU is rarely a scarce resource any more. Most programs for personal computers are limited by the rate at which the user can present input (by typing or clicking), not by the rate the CPU can process it. Even compilations, a major sink of CPU cycles in the past, take just a few seconds in most cases nowadays. Even when two programs are actually running at once, such as a word processor and a spreadsheet, it hardly matters which goes first since the user is probably waiting for both of them to finish. As a consequence, scheduling does not matter much on simple PCs. Of course, there are applications that practically eat the CPU alive. For instance rendering one hour of high-resolution video while tweaking the colors in each of the 107,892 frames (in NTSC) or 90,000 frames (in PAL) requires industrial-strength computing power. However, similar applications are the exception rather than the rule.

When we turn to networked servers, the situation changes appreciably. Here multiple processes often do compete for the CPU, so scheduling matters again. For example, when the CPU has to choose between running a process that gathers the daily statistics and one that serves user requests, the users will be a lot happier if the latter gets first crack at the CPU.

The "abundance of resources" argument also does not hold on many mobile devices, such as smartphones (except perhaps the most powerful models) and nodes in sensor networks. Here, the CPU may still be weak and the memory small. Moreover, since battery lifetime is one of the most important constraints on these devices, some schedulers try to optimize the power consumption.

In addition to picking the right process to run, the scheduler also has to worry about making efficient use of the CPU because process switching is expensive. To start with, a switch from user mode to kernel mode must occur. Then the state of the current process must be saved, including storing its registers in the process table so they can be reloaded later. In some systems, the memory map (e.g., memory reference bits in the page table) must be saved as well. Next a new process must be selected by running the scheduling algorithm. After that, the memory management unit (MMU) must be reloaded with the memory map of the new process. Finally, the new process must be started. In addition to all that, the process switch may invalidate the memory cache and related tables, forcing it to be dynamically reloaded from the main memory twice (upon entering the kernel and upon leaving it). All in all, doing too many process switches per second can chew up a substantial amount of CPU time, so caution is advised.

Process Behavior

Nearly all processes alternate bursts of computing with (disk or network) I/O requests, as shown in Fig. 2-39. Often, the CPU runs for a while without stopping, then a system call is made to read from a file or write to a file. When the system call completes, the CPU computes again until it needs more data or has to write more data, and so on. Note that some I/O activities count as computing. For example, when the CPU copies bits to a video RAM to update the screen, it is computing, not doing I/O, because the CPU is in use. I/O in this sense is when a process enters the blocked state waiting for an external device to complete its work.

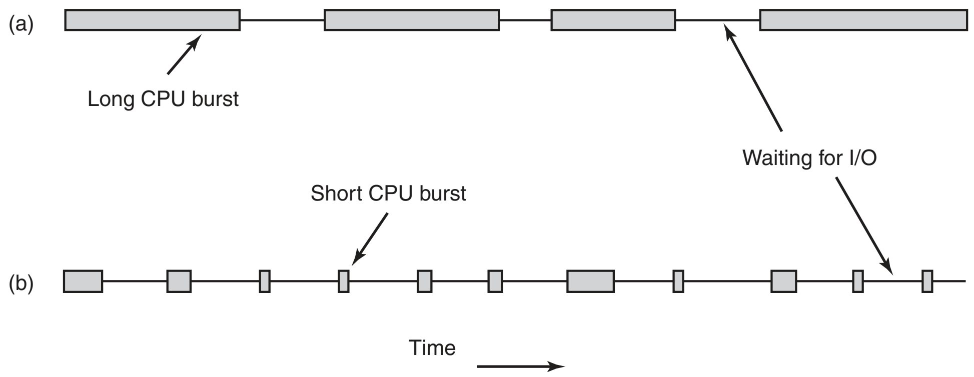


Figure 2-39. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

The important thing to notice about Fig. 2-39 is that some processes, such as the one in Fig. 2-39(a), spend most of their time computing, while other processes, such as the one shown in Fig. 2-39(b), spend most of their time waiting for I/O.

The former are called **compute-bound** or **CPU-bound**; the latter are called **I/O-bound**. Compute-bound processes typically have long CPU bursts and thus infrequent I/O waits, whereas I/O-bound processes have short CPU bursts and thus frequent I/O waits. Note that the key factor is the length of the CPU burst, not the length of the I/O burst. I/O-bound processes are I/O bound because they do not compute much between I/O requests, not because they have especially long I/O requests. It takes the same time to issue the hardware request to read a disk block no matter how much or how little time it takes to process the data after they arrive.

It is worth noting that as CPUs get faster, processes tend to get more I/O-bound. This effect occurs because CPUs are improving much faster than disks. As a consequence, the scheduling of I/O-bound processes is likely to become a more important subject in the future. The basic idea here is that if an I/O-bound process wants to run, it should get a chance quickly so that it can issue its disk request and keep the disk busy. As we saw in Fig. 2-6, when processes are I/O bound, it takes quite a few of them to keep the CPU fully occupied.

When to Schedule

A key issue related to scheduling is when to make scheduling decisions. It turns out that there are a variety of situations in which scheduling is needed. First, when a new process is created, a decision needs to be made whether to run the parent process or the child process. Since both processes are in ready state, it is a normal scheduling decision and can go either way, that is, the scheduler can legitimately choose to run either the parent or the child next.

Second, a scheduling decision must be made when a process exits. That process can no longer run (since it no longer exists), so some other process must be chosen from the set of ready processes. If no process is ready, a system-supplied idle process is normally run.

Third, when a process blocks on I/O, on a semaphore, or for some other reason, another process has to be selected to run. Sometimes the reason for blocking may play a role in the choice. For example, if *A* is an important process and it is waiting for *B* to exit its critical region, letting *B* run next will allow it to exit its critical region and thus let *A* continue. The trouble, however, is that the scheduler generally does not have the necessary information to take this dependency into account.

Fourth, when an I/O interrupt occurs, a scheduling decision may be made. If the interrupt came from an I/O device that has now completed its work, some process that was blocked waiting for the I/O may now be ready to run. It is up to the scheduler to decide whether to run the newly ready process, the process that was running at the time of the interrupt, or some third process.

If a hardware clock provides periodic interrupts at 50 or 60 Hz or some other frequency, a scheduling decision can be made at each clock interrupt or at every *k*th clock interrupt. Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts. A **nonpreemptive** scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or voluntarily releases the CPU. Even if it runs for many hours, it will not be forcibly suspended. In effect, no scheduling decisions are made during clock interrupts. After clock-interrupt processing has been finished, the process that was running before the interrupt is resumed, unless a higher-priority process was waiting for a now-satisfied timeout.

In contrast, a **preemptive** scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available). Doing preemptive scheduling requires having a clock interrupt occur at the end of the time interval to give control of the CPU back to the scheduler. If no clock is available, nonpreemptive scheduling is the only option.

Categories of Scheduling Algorithms

Not surprisingly, in different environments different scheduling algorithms are needed. This situation arises because different application areas (and different kinds of operating systems) have different goals. In other words, what the scheduler should optimize for is not the same in all systems. Three environments worth distinguishing are

1. Batch.
2. Interactive.
3. Real time.

Batch systems are still in widespread use in the business world for doing payroll, inventory, accounts receivable, accounts payable, interest calculation (at banks), claims processing (at insurance companies), and other periodic tasks. In batch systems, there are no users impatiently waiting at their terminals for a quick response to a short request. Consequently, nonpreemptive algorithms, or preemptive algorithms with long time periods for each process, are often acceptable. This approach reduces process switches and thus improves performance. The batch algorithms are actually fairly general and often applicable to other situations as well, which makes them worth studying, even for people not involved in corporate mainframe computing.

In an environment with interactive users, preemption is essential to keep one process from hogging the CPU and denying service to the others. Even if no process intentionally ran forever, one process might shut out all the others indefinitely due to a program bug. Preemption is needed to prevent this behavior. Servers also fall into this category, since they normally serve multiple (remote) users, all of whom are in a big hurry. Computer users are always in a big hurry.

In systems with real-time constraints, preemption is, oddly enough, sometimes not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly. The difference with interactive systems is that real-time systems run only programs that are intended to further the application at hand. Interactive systems are general purpose and may run arbitrary programs that are not cooperative and even possibly malicious.

Scheduling Algorithm Goals

In order to design a scheduling algorithm, it is necessary to have some idea of what a good algorithm should do. Some goals depend on the environment (batch, interactive, or real time), but some are desirable in all cases. Some goals are listed in Fig. 2-40. We will discuss these in turn below.

All systems
- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems
- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems
- Response time - respond to requests quickly
- Proportionality - meet users’ expectations

Real-time systems
- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

Figure 2-40. Some goals of the scheduling algorithm under different circumstances.

Under all circumstances, fairness is important. Comparable processes should get comparable service. Giving one process much more CPU time than an equivalent one is not fair. Of course, different categories of processes may be treated differently. Think of safety control and doing the payroll at a nuclear reactor’s computer center.

Somewhat related to fairness is enforcing the system’s policies. If the local policy is that safety control processes get to run whenever they want to, even if it means the payroll is 30 sec late, the scheduler has to make sure this policy is enforced.

Another general goal is keeping all parts of the system busy when possible. If the CPU and all the I/O devices can be kept running all the time, more work gets done per second than if some of the components are idle. In a batch system, for example, the scheduler has control of which jobs are brought into memory to run. Having some CPU-bound processes and some I/O-bound processes in memory together is a better idea than first loading and running all the CPU-bound jobs and then, when they are finished, loading and running all the I/O-bound jobs. If the latter strategy is used, when the CPU-bound processes are running, they will fight for the CPU and the disk will be idle. Later, when the I/O-bound jobs come in, they will fight for the disk and the CPU will be idle. Better to keep the whole system running at once by a careful mix of processes.

The managers of large computer centers that run many batch jobs typically look at three metrics to see how well their systems are performing: throughput, turnaround time, and CPU utilization. **Throughput** is the number of jobs per hour that the system completes. All things considered, finishing 50 jobs per hour is better than finishing 40 jobs per hour. **Turnaround time** is the statistically average time from the moment that a batch job is submitted until the moment it is completed. It measures how long the average user has to wait for the output. Here the rule is: Small is Beautiful.

A scheduling algorithm that tries to maximize throughput may not necessarily minimize turnaround time. For example, given a mix of short jobs and long jobs, a scheduler that always ran short jobs and never ran long jobs might achieve an excellent throughput (many short jobs per hour) but at the expense of a terrible turnaround time for the long jobs. If short jobs kept arriving at a fairly steady rate, the long jobs might never run, making the mean turnaround time infinite while achieving a high throughput.

CPU utilization is often used as a metric on batch systems. Actually though, it is not a good metric. What really matters is how many jobs per hour come out of the system (throughput) and how long it takes to get a job back (turnaround time). Using CPU utilization as a metric is like rating cars based on how many times per hour the engine turns over. However, knowing when the CPU utilization is almost 100% is useful for knowing when it is time to get more computing power.

For interactive systems, different goals apply. The most important one is to minimize **response time**, that is, the time between issuing a command and getting the result. On a personal computer where a background process is running (for example, reading and storing email from the network), a user request to start a program or open a file should take precedence over the background work. Having all interactive requests go first will be perceived as good service.

A somewhat related issue is what might be called **proportionality**. Users have an inherent (but often incorrect) idea of how long things should take. When a request that the user perceives as complex takes a long time, users accept that, but when a request that is perceived as simple takes a long time, users get irritated. For example, if clicking on an icon that starts uploading a 500-MB video to a cloud server takes 60 sec, the user will probably accept that as a fact of life because he does not expect the upload to take 5 sec. He knows it will take time.

On the other hand, when a user clicks on the icon that breaks the connection to the cloud server after the video has been uploaded, he has different expectations. If it has not completed after 30 sec, the user will probably be swearing a blue streak, and after 60 sec he will be foaming at the mouth. This behavior is due to the common user perception that sending a lot of data is *supposed* to take a lot longer than just breaking the connection. In some cases (such as this one), the scheduler cannot do anything about the response time, but in other cases it can, especially when the delay is due to a poor choice of process order.

Real-time systems have different properties than interactive systems, and thus different scheduling goals. They are characterized by having deadlines that must or at least should be met. For example, if a computer is controlling a device that produces data at a regular rate, failure to run the data-collection process on time may result in lost data. Thus the foremost need in a real-time system is meeting all (or most) deadlines.

In some real-time systems, especially those involving multimedia, predictability is important. Missing an occasional deadline is not fatal, but if the audio process runs too erratically, the sound quality will deteriorate rapidly. Video is also an issue, but the ear is much more sensitive to jitter than the eye. To avoid this problem, process scheduling must be highly predictable and regular. We will study batch and interactive scheduling algorithms in this chapter. Real-time scheduling is not covered in the book but in the extra material on multimedia operating systems on the book's Website.

2.4.2 Scheduling in Batch Systems

It is now time to turn from general scheduling issues to specific scheduling algorithms. In this section we will look at algorithms used in batch systems. In the following ones we will examine interactive and real-time systems. It is worth pointing out that some algorithms are used in both batch and interactive systems. We will study these later.

First-Come, First-Served

Probably the simplest of all scheduling algorithms ever devised is nonpreemptive **first-come, first-served**. With this algorithm, processes are assigned the CPU in the order they request it. Basically, there is a single queue of ready processes. When the first job enters the system from the outside in the morning, it is started immediately and allowed to run as long as it wants to. It is not interrupted because it has run too long. As other jobs come in, they are put onto the end of the queue. When the running process blocks, the first process on the queue is run next. When a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue, behind all waiting processes.

The great strength of this algorithm is that it is easy to understand and equally easy to program. It is also fair in the same sense that allocating scarce concert tickets or brand-new iPhones to people who are willing to stand on line starting at 2 A.M. is fair. With this algorithm, a single linked list keeps track of all ready processes. Picking a process to run just requires removing one from the front of the queue. Adding a new job or unblocked process just requires attaching it to the end of the queue. What could be simpler to understand and implement?

Unfortunately, first-come, first-served also has a powerful disadvantage. Suppose there is one compute-bound process that runs for 1 sec at a time and many I/O-bound processes that use little CPU time but each have to perform 1000 disk reads to complete. The compute-bound process runs for 1 sec, then it reads a disk block. All the I/O processes now run and start disk reads. When the compute-bound process gets its disk block, it runs for another 1 sec, followed by all the I/O-bound processes in quick succession.

The net result is that each I/O-bound process gets to read 1 block per second and will take 1000 sec to finish. With a scheduling algorithm that preempted the compute-bound process every 10 msec, the I/O-bound processes would finish in 10 sec instead of 1000 sec, and without slowing down the compute-bound process very much.

Shortest Job First

Now let us look at another nonpreemptive batch algorithm that assumes the run times are known in advance. In an insurance company, for example, people can predict quite accurately how long it will take to run a batch of 1000 claims, since similar work is done every day. When several equally important jobs are sitting in the input queue waiting to be started, the scheduler picks the **shortest job first**. Look at Fig. 2-41. Here we find four jobs *A*, *B*, *C*, and *D* with run times of 8, 4, 4, and 4 minutes, respectively. By running them in that order, the turnaround time for *A* is 8 minutes, for *B* is 12 minutes, for *C* is 16 minutes, and for *D* is 20 minutes for an average of 14 minutes.

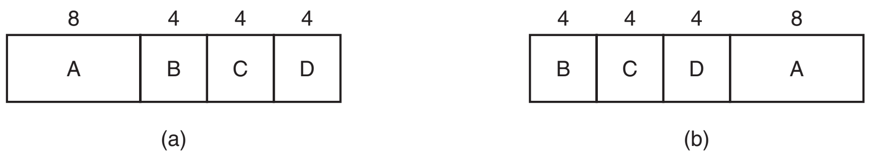


Figure 2-41. An example of shortest-job-first scheduling. (a) Running four jobs in the original order. (b) Running them in shortest job first order.

Now let us consider running these four jobs using shortest job first, as shown in Fig. 2-41(b). The turnaround times are now 4, 8, 12, and 20 minutes for an average of 11 minutes. Shortest job first is provably optimal. Consider the case of four jobs, with execution times of a, b, c, and d, respectively. The first job finishes at time a, the second at time $a + b$, and so on. The mean turnaround time is $(4a + 3b + 2c + d)/4$. It is clear that a contributes more to the average than the other times, so it should be the shortest job, with b next, then c, and finally d as the longest since it affects only its own turnaround time. The same argument applies equally well to any number of jobs.

It is worth pointing out that shortest job first is optimal only when all the jobs are available simultaneously. As a counterexample, consider five jobs, A through E, with run times of 2, 4, 1, 1, and 1, respectively. Their arrival times are 0, 0, 3, 3, and 3. Initially, only A or B can be chosen, since the other three jobs have not arrived yet. Using shortest job first, we will run the jobs in the order A, B, C, D, E, for an average wait of 4.6. However, running them in the order B, C, D, E, A has an average wait of 4.4.

Shortest Remaining Time Next

A preemptive version of shortest job first is **shortest remaining time next**. With this algorithm, the scheduler always chooses the process whose remaining run time is the shortest. Again here, the run time has to be known in advance. When a new job arrives, its total time is compared to the current process' remaining time. If the new job needs less time to finish than the current process, the current process is suspended and the new job started. This scheme allows new short jobs to get good service.

2.4.3 Scheduling in Interactive Systems

We will now look at some algorithms that can be used in interactive systems. These are common on personal computers, servers, and other kinds of systems as well.

Round-Robin Scheduling

One of the oldest, simplest, fairest, and most widely used algorithms is **round robin**. Each process is assigned a time interval, called its **quantum**, during which it is allowed to run. If the process is still running at the end of the quantum, the CPU is preempted and given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course. Round robin is easy to implement. All the scheduler needs to do is maintain a list of runnable processes, as shown in Fig. 2-42(a). When the process uses up its quantum, it is put on the end of the list, as shown in Fig. 2-42(b).

The only really interesting issue with round robin is the length of the quantum. Switching from one process to another requires a certain amount of time for doing all the administration—saving and loading registers and memory maps, updating

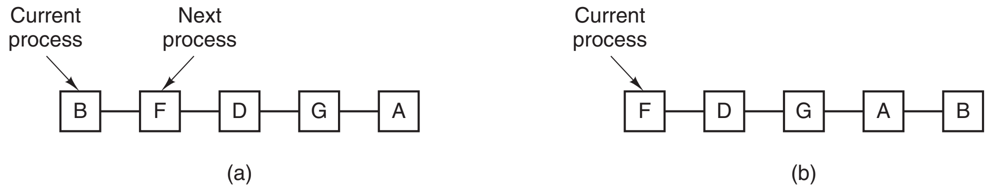


Figure 2-42. Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

various tables and lists, flushing and reloading the memory cache, and so on. Suppose that this **process switch** or **context switch**, as it is sometimes called, takes 1 msec, including switching memory maps, flushing and reloading the cache, etc. Also suppose that the quantum is set at 4 msec. With these parameters, after doing 4 msec of useful work, the CPU will have to spend (i.e., waste) 1 msec on process switching. Thus 20% of the CPU time will be thrown away on administrative overhead. Clearly, this is too much.

To improve the CPU efficiency, we could set the quantum to, say, 100 msec. Now the wasted time is only 1%. But consider what happens on a server system if 50 requests come in within a very short time interval and with widely varying CPU requirements. Fifty processes will be put on the list of runnable processes. If the CPU is idle, the first one will start immediately, the second one may not start until 100 msec later, and so on. The unlucky last one may have to wait 5 sec before getting a chance, assuming all the others use their full quanta. Most users will perceive a 5-sec response to a short command as sluggish. This situation is especially bad if some of the requests near the end of the queue required only a few milliseconds of CPU time. With a short quantum they would have gotten better service.

Another factor is that if the quantum is set longer than the mean CPU burst, preemption will not happen very often. Instead, most processes will perform a blocking operation before the quantum runs out, causing a process switch. Eliminating preemption improves performance because process switches then happen only when they are logically necessary, that is, when a process blocks and cannot continue.

The conclusion can be formulated as follows: setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests. A quantum around 20–50 msec is often a reasonable compromise.

Priority Scheduling

Round-robin scheduling makes the implicit assumption that all processes are equally important. Frequently, the people who own and operate multiuser computers have quite different ideas on that subject. At a university, for example, the pecking order may be the president first, the faculty deans next, then professors, secretaries, janitors, and finally students. The need to take external factors into account leads to **priority scheduling**. The basic idea is straightforward: each process is assigned a priority, and the runnable process with the highest priority is allowed to run.

Even on a PC with a single owner, there may be multiple processes, some of them more important than others. For example, a daemon process sending electronic mail in the background should be assigned a lower priority than a process displaying a video film on the screen in real time.

To prevent high-priority processes from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick (i.e., at each clock interrupt). If this action causes its priority to drop below that of the next highest process, a process switch occurs. Alternatively, each process may be assigned a maximum time quantum that it is allowed to run. When this quantum is used up, the next-highest-priority process is given a chance to run.

Priorities can be assigned to processes statically or dynamically. On a military computer, processes started by generals might begin at priority 100, processes started by colonels at 90, majors at 80, captains at 70, lieutenants at 60, and so on down the totem pole. Alternatively, at a commercial computer center, high-priority jobs might cost $100 an hour, medium priority $75 an hour, and low priority $50 an hour. The UNIX system has a command, *nice*, which allows a user to voluntarily reduce the priority of his process, in order to be nice to the other users. Nobody ever uses it.

Priorities can also be assigned dynamically by the system to achieve certain system goals. For example, some processes are highly I/O bound and spend most of their time waiting for I/O to complete. Whenever such a process wants the CPU, it should be given the CPU immediately, to let it start its next I/O request, which can then proceed in parallel with another process actually computing. Making the I/O-bound process wait a long time for the CPU will just mean having it around occupying memory for an unnecessarily long time. A simple algorithm for giving good service to I/O-bound processes is to set the priority to $1/f$, where f is the fraction of the last quantum that a process used. A process that used only 1 msec of its 50-msec quantum would get priority 50, while a process that ran 25 msec before blocking would get priority 2, and a process that used the whole quantum would get priority 1.

It is often convenient to group processes into priority classes and use priority scheduling among the classes but round-robin scheduling within each class. Figure 2-43 shows a system with four priority classes. The scheduling algorithm is as follows: as long as there are runnable processes in priority class 4, just run each one for one quantum, round-robin fashion, and never bother with lower-priority classes. If priority class 4 is empty, then run the class 3 processes round robin. If classes 4 and 3 are both empty, then run class 2 round robin, and so on. If priorities are not adjusted occasionally, lower-priority classes may all starve to death.

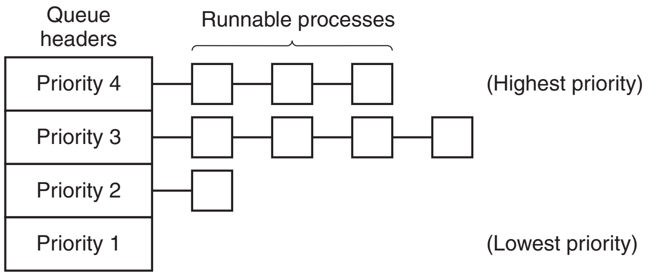


Figure 2-43. A scheduling algorithm with four priority classes.

Multiple Queues

One of the earliest priority schedulers was in CTSS, the M.I.T. Compatible TimeSharing System that ran on the IBM 7094 (Corbató et al., 1962). CTSS had the problem that process switching was slow because the 7094 could hold only one process in memory. Each switch meant swapping the current process to disk and reading in a new one from disk. The CTSS designers quickly realized that it was more efficient to give CPU-bound processes a large quantum once in a while, rather than giving them small quanta frequently (to reduce swapping). On the other hand, giving all processes a large quantum would mean poor response time, as we have already seen. Their solution was to set up priority classes. Processes in the highest class were run for one quantum. Processes in the next-highest class were run for two quanta. Processes in the next one were run for four quanta, etc. Whenever a process used up all the quanta allocated to it, it was moved down one class.

As an example, consider a process that needed to compute continuously for 100 quanta. It would initially be given one quantum, then swapped out. Next time it would get two quanta before being swapped out. On succeeding runs it would get 4, 8, 16, 32, and 64 quanta, although it would have used only 37 of the final 64 quanta to complete its work. Only 7 swaps would be needed (including the initial load) instead of 100 with a pure round-robin algorithm. Furthermore, as the process sank deeper and deeper into the priority queues, it would be run less and less frequently, saving the CPU for short, interactive processes.

The following policy was adopted to avoid punishing forever a process that needed to run for a long time when it first started but became interactive later. Whenever a carriage return (*Enter* key) was typed at a terminal, the process belonging to that terminal was moved to the highest-priority class, on the assumption that it was about to become interactive. One fine day, some user with a heavily CPU-bound process discovered that just sitting at the terminal and typing carriage returns at random every few seconds did wonders for his response time. He told all his friends. They told all their friends. Moral of the story: getting it right in practice is much harder than getting it right in principle.

Shortest Process Next

Because shortest job first always produces the minimum average response time for batch systems, it would be nice if it could be used for interactive processes as well. To a certain extent, it can be. Interactive processes generally follow the pattern of wait for command, execute command, wait for command, execute command, etc. If we regard the execution of each command as a separate "job," then we can minimize overall response time by running the shortest one first. The problem is figuring out which of the currently runnable processes is the shortest one.

One approach is to make estimates based on past behavior and run the process with the shortest estimated running time. Suppose that the estimated time per command for some process is T_0. Now suppose its next run is measured to be T_1. We could update our estimate by taking a weighted sum of these two numbers, that is, $aT_0 + (1 - a)T_1$. Through the choice of a we can decide to have the estimation process forget old runs quickly, or remember them for a long time. With $a = 1/2$, we get successive estimates of

$$T_0,\quad T_0/2 + T_1/2,\quad T_0/4 + T_1/4 + T_2/2,\quad T_0/8 + T_1/8 + T_2/4 + T_3/2$$

After three new runs, the weight of T_0 in the new estimate has dropped to 1/8.

The technique of estimating the next value in a series by taking the weighted average of the current measured value and the previous estimate is sometimes called **aging**. It is applicable to many situations where a prediction must be made based on previous values. Aging is especially easy to implement when $a = 1/2$. All that is needed is to add the new value to the current estimate and divide the sum by 2 (by shifting it right 1 bit).

Guaranteed Scheduling

A completely different approach to scheduling is to make real promises to the users about performance and then live up to those promises. One promise that is realistic to make and easy to live up to is this: If n users are logged in while you are working, you will receive about $1/n$ of the CPU power. Similarly, on a single-user system with n processes running, all things being equal, each one should get $1/n$ of the CPU cycles. That seems fair enough.

To make good on this promise, the system must keep track of how much CPU each process has had since its creation. It then computes the amount of CPU each one is entitled to, namely the time since creation divided by n. Since the amount of CPU time each process has actually had is also known, it is fairly straightforward to compute the ratio of actual CPU time consumed to CPU time entitled. A ratio of 0.5 means that a process has only had half of what it should have had, and a ratio of 2.0 means that a process has had twice as much as it was entitled to. The algorithm is then to run the process with the lowest ratio until its ratio has moved above that of its closest competitor. Then that one is chosen to run next.

Lottery Scheduling

While making promises to the users and then living up to them is a fine idea, it is difficult to implement. However, another algorithm can be used to give similarly predictable results with a much simpler implementation. It is called **lottery scheduling** (Waldspurger and Weihl, 1994).

The basic idea is to give processes lottery tickets for various system resources, such as CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource. When applied to CPU scheduling, the system might hold a lottery 50 times a second, with each winner getting 20 msec of CPU time as a prize.

To paraphrase George Orwell: "All processes are equal, but some processes are more equal." More important processes can be given extra tickets, to increase their odds of winning. If there are 100 tickets outstanding, and one process holds 20 of them, it will have a 20% chance of winning each lottery. In the long run, it will get about 20% of the CPU. In contrast to a priority scheduler, where it is very hard to state what having a priority of 40 actually means, here the rule is clear: a process holding a fraction *f* of the tickets will get about a fraction *f* of the resource in question.

Lottery scheduling has several interesting properties. For example, if a new process shows up and is granted some tickets, at the very next lottery it will have a chance of winning in proportion to the number of tickets it holds. In other words, lottery scheduling is highly responsive.

Cooperating processes may exchange tickets if they wish. For example, when a client process sends a message to a server process and then blocks, it may give all of its tickets to the server, to increase the chance of the server running next. When the server is finished, it returns the tickets so that the client can run again. In fact, in the absence of clients, servers need no tickets at all.

Lottery scheduling can be used to solve problems that are difficult to handle with other methods. One example is a video server in which several processes are feeding video streams to their clients, but at different frame rates. Suppose that the processes need frames at 10, 20, and 25 frames/sec. By allocating these processes 10, 20, and 25 tickets, respectively, they will automatically divide the CPU in approximately the correct proportion, that is, 10 : 20 : 25.

Fair-Share Scheduling

So far we have assumed that each process is scheduled on its own, without regard to who its owner is. As a result, if user 1 starts up nine processes and user 2 starts up one process, with round robin or equal priorities, user 1 will get 90% of the CPU and user 2 only 10% of it.

To prevent this situation, some systems take into account which user owns a process before scheduling it. In this model, each user is allocated some fraction of the CPU and the scheduler picks processes in such a way as to enforce it. Thus if two users have each been promised 50% of the CPU, they will each get that, no matter how many processes they have in existence.

As an example, consider a system with two users, each of which has been promised 50% of the CPU. User 1 has four processes, *A*, *B*, *C*, and *D*, and user 2 has only one process, *E*. If round-robin scheduling is used, a possible scheduling sequence that meets all the constraints is this one:

A E B E C E D E A E B E C E D E ...

On the other hand, if user 1 is entitled to twice as much CPU time as user 2, we might get

A B E C D E A B E C D E ...

Numerous other possibilities exist, of course, and can be exploited, depending on what the notion of fairness is.

2.4.4 Scheduling in Real-Time Systems

A **real-time** system is one in which time plays an essential role. Typically, one or more physical devices external to the computer generate stimuli, and the computer must react appropriately to them within a fixed amount of time. For example, the computer in a compact disc player gets the bits as they come off the drive and must convert them into music within a very tight time interval. If the calculation takes too long, the music will sound peculiar. Other real-time systems are patient monitoring in a hospital intensive-care unit, the autopilot in an aircraft, and robot control in an automated factory. In all these cases, having the right answer but having it too late is often just as bad as not having it at all.

Real-time systems are generally categorized as **hard real time**, meaning there are absolute deadlines that must be met—or else!— and **soft real time**, meaning that missing an occasional deadline is undesirable, but nevertheless tolerable. In both cases, real-time behavior is achieved by dividing the program into a number of processes, each of whose behavior is predictable and known in advance. These processes are generally short lived and can run to completion in well under a second. When an external event is detected, it is the job of the scheduler to schedule the processes in such a way that all deadlines are met.

The events that a real-time system may have to respond to can be further categorized as **periodic** (meaning they occur at regular intervals) or **aperiodic** (meaning they occur unpredictably). A system may have to respond to multiple periodic-event streams. Depending on how much time each event requires for processing, handling all of them may not even be possible. For example, if there are m periodic events and event i occurs with period P_i and requires C_i sec of CPU time to handle each event, then the load can be handled only if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le 1$$

A real-time system that meets this criterion is said to be **schedulable**. This means it can actually be implemented. A process that fails to meet this test cannot be scheduled because the total amount of CPU time the processes want collectively is more than the CPU can deliver.

As an example, consider a soft real-time system with three periodic events, with periods of 100, 200, and 500 msec, respectively. If these events require 50, 30, and 100 msec of CPU time per event, respectively, the system is schedulable because $0.5 + 0.15 + 0.2 < 1$. If a fourth event with a period of 1 sec is added, the system will remain schedulable as long as this event does not need more than 150 msec of CPU time per event. Implicit in this calculation is the assumption that the context-switching overhead is so small that it can be ignored.

Real-time scheduling algorithms can be static or dynamic. The former make their scheduling decisions before the system starts running. The latter make their scheduling decisions at run time, after execution has started. Static scheduling works only when there is perfect information available in advance about the work to be done and the deadlines that have to be met. Dynamic scheduling algorithms do not have these restrictions.

2.4.5 Policy Versus Mechanism

Up until now, we have tacitly assumed that all the processes in the system belong to different users and are thus competing for the CPU. While this is often true, sometimes it happens that one process has many children running under its control. For example, a database-management-system process may have many children. Each child might be working on a different request, or each might have some specific function to perform (query parsing, disk access, etc.). It is entirely possible that the main process has an excellent idea of which of its children are the most important (or time critical) and which the least. Unfortunately, none of the schedulers discussed above accept any input from user processes about scheduling decisions. As a result, the scheduler rarely makes the best choice.

The solution to this problem is to separate the **scheduling mechanism** from the **scheduling policy**, a long-established principle (Levin et al., 1975). What this means is that the scheduling algorithm is parameterized in some way, but the parameters can be filled in by user processes. Let us consider the database example once again. Suppose that the kernel uses a priority-scheduling algorithm but provides a system call by which a process can set (and change) the priorities of its children. In this way, the parent can control how its children are scheduled, even though it itself does not do the scheduling. Here the mechanism is in the kernel but policy is set by a user process. Policy-mechanism separation is a key idea.

2.4.6 Thread Scheduling

When several processes each have multiple threads, we have two levels of parallelism present: processes and threads. Scheduling in such systems differs substantially depending on whether user-level threads or kernel-level threads (or both) are supported.

Let us consider user-level threads first. Since the kernel is not aware of the existence of threads, it operates as it always does, picking a process, say, *A*, and giving *A* control for its quantum. The thread scheduler inside *A* decides which thread to run, say *A1*. Since there are no clock interrupts to multiprogram threads, this thread may continue running as long as it wants to. If it uses up the process' entire quantum, the kernel will select another process to run.

When the process *A* finally runs again, thread *A1* will resume running. It will continue to consume all of *A*'s time until it is finished. However, its antisocial behavior will not affect other processes. They will get whatever the scheduler considers their appropriate share, no matter what is going on inside process *A*.

Now consider the case that *A*'s threads have relatively little work to do per CPU burst, for example, 5 msec of work within a 50-msec quantum. Consequently, each one runs for a little while, then yields the CPU back to the thread scheduler. This might lead to the sequence *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, before the kernel switches to process *B*. This situation is illustrated in Fig. 2-44(a).

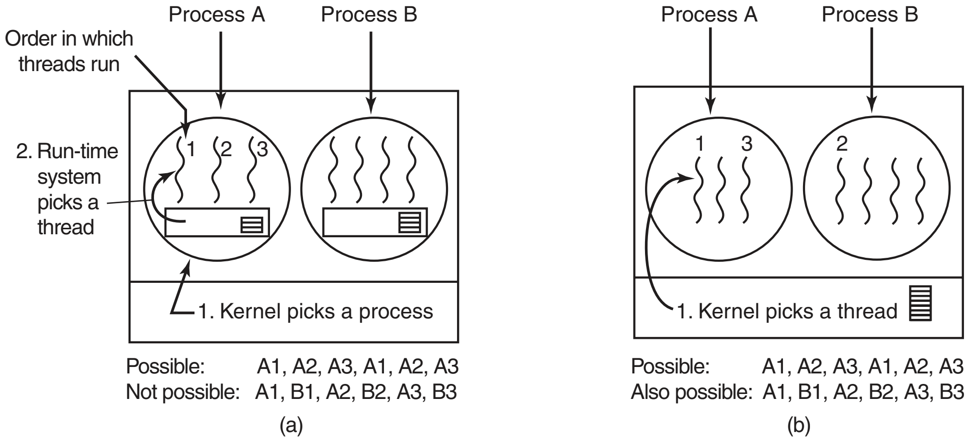

Process A
Process B
Order in which threads run
1
2
3
2. Run-time system picks a thread
1. Kernel picks a process
Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3
(a)
Process A
Process B
1
3
2
1. Kernel picks a thread
Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3
(b)

Figure 2-44. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

The scheduling algorithm used by the run-time system can be any of the ones described above. In practice, round-robin scheduling and priority scheduling are most common. The only constraint is the absence of a clock to interrupt a thread that has run too long. Since threads cooperate, this is usually not an issue.

Now consider the situation with kernel-level threads. Here the kernel picks a particular thread to run. It does not have to take into account which process the thread belongs to, but it can if it wants to. The thread is given a quantum and is forcibly suspended if it exceeds the quantum. With a 50-msec quantum but threads that block after 5 msec, the thread order for some period of 30 msec might be *A1*, *B1*, *A2*, *B2*, *A3*, *B3*, something not possible with these parameters and user-level threads. This situation is partially depicted in Fig. 2-44(b).

A major difference between user-level threads and kernel-level threads is the performance. Doing a thread switch with user-level threads takes a handful of machine instructions. With kernel-level threads it requires a full context switch, changing the memory map and invalidating the cache, which is several orders of magnitude slower. On the other hand, with kernel-level threads, having a thread block on I/O does not suspend the entire process as it does with user-level threads.

Since the kernel knows that switching from a thread in process *A* to a thread in process *B* is more expensive than running a second thread in process *A* (due to having to change the memory map and having the memory cache spoiled), it can take this information into account when making a decision. For example, given two threads that are otherwise equally important, with one of them belonging to the same process as a thread that just blocked and one belonging to a different process, preference could be given to the former.

Another important factor is that user-level threads can employ an application-specific thread scheduler. Consider, for example, the Web server of Fig. 2-8. Suppose that a worker thread has just blocked and the dispatcher thread and two worker threads are ready. Who should run next? The run-time system, knowing what all the threads do, can easily pick the dispatcher to run next, so that it can start another worker running. This strategy maximizes the amount of parallelism in an environment where workers frequently block on disk I/O. With kernel-level threads, the kernel would never know what each thread did (although they could be assigned different priorities). In general, however, application-specific thread schedulers can tune an application better than the kernel can.

2.5 CLASSICAL IPC PROBLEMS

The operating systems literature is full of interesting problems that have been widely discussed and analyzed using a variety of synchronization methods. In the following sections we will examine three of the better-known problems.

2.5.1 The Dining Philosophers Problem

In 1965, Dijkstra posed and then solved a synchronization problem he called the **dining philosophers problem**. Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new primitive is by showing how elegantly it solves the dining philosophers problem. The problem can be stated quite simply as follows. Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The layout of the table is illustrated in Fig. 2-45.

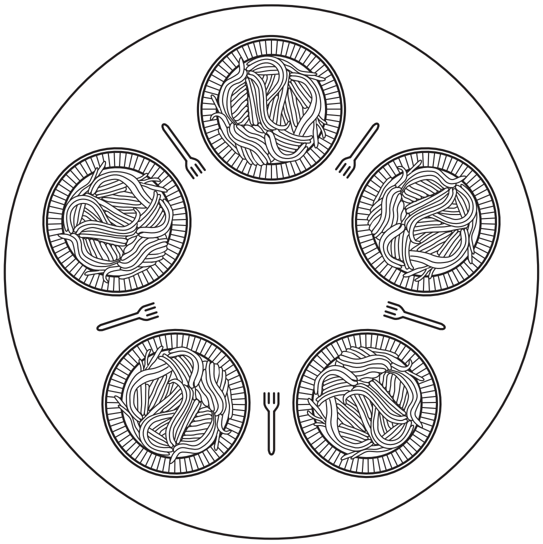


Figure 2-45. Lunch time in the Philosophy Department.

The life of a philosopher consists of alternating periods of eating and thinking. (This is something of an abstraction, even for philosophers, but the other activities are irrelevant here.) When a philosopher gets sufficiently hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think. The key question is: Can you write a program for each philosopher that does what it is supposed to do and never gets stuck? (It has been pointed out that the two-fork requirement is somewhat artificial; perhaps we should switch from Italian food to Chinese food, substituting rice for spaghetti and chopsticks for forks.)

Figure 2-46 shows the obvious solution. The procedure *take_fork* waits until the specified fork is available and then seizes it. Unfortunately, the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

We could easily modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too, fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks,

```
#define N 5                                 /* number of philosophers */

void philosopher(int i)                     /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                         /* philosopher is thinking */
          take_fork(i);                     /* take left fork */
          take_fork((i+1) % N);             /* take right fork; % is modulo operator */
          eat( );                           /* yum-yum, spaghetti */
          put_fork(i);                      /* put left fork back on the table */
          put_fork((i+1) % N);              /* put right fork back on the table */
     }
}
```

Figure 2-46. A nonsolution to the dining philosophers problem.

waiting, picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress, is called **starvation**. (It is called starvation even when the problem does not occur in an Italian or a Chinese restaurant.)

Now you might think that if the philosophers would just wait a random time instead of the same time after failing to acquire the right-hand fork, the chance that everything would continue in lockstep for even an hour is very small. This observation is true, and in nearly all applications trying again later is not a problem. For example, in the popular Ethernet local area network, if two computers send a packet at the same time, each one waits a random time and tries again; in practice this solution works fine. However, in a few applications one would prefer a solution that always works and cannot fail due to an unlikely series of random numbers. Think about safety control in a nuclear power plant.

One improvement to Fig. 2-46 that has no deadlock and no starvation is to protect the five statements following the call to *think* by a binary semaphore. Before starting to acquire forks, a philosopher would do a down on *mutex*. After replacing the forks, she would do an up on *mutex*. From a theoretical viewpoint, this solution is adequate. From a practical one, it has a performance bug: only one philosopher can be eating at any instant. With five forks available, we should be able to allow two philosophers to eat at the same time.

The solution presented in Fig. 2-47 is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, *state*, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move into eating state only if neither neighbor is eating. Philosopher *i*'s neighbors are defined by the macros *LEFT* and *RIGHT*. In other words, if *i* is 2, *LEFT* is 1 and *RIGHT* is 3.

The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. Note that each process runs the procedure *philosopher* as its main code, but the other procedures, *take_forks*, *put_forks*, and *test*, are ordinary procedures and not separate processes.

```
#define N              5                  /* number of philosophers */
#define LEFT           (i+N−1)%N          /* number of i's left neighbor */
#define RIGHT          (i+1)%N            /* number of i's right neighbor */
#define THINKING       0                  /* philosopher is thinking */
#define HUNGRY         1                  /* philosopher is trying to get forks */
#define EATING         2                  /* philosopher is eating */

typedef int semaphore;                    /* semaphores are a special kind of int */
int state[N];                             /* array to keep track of everyone's state */
semaphore mutex = 1;                      /* mutual exclusion for critical regions */
semaphore s[N];                           /* one semaphore per philosopher */


void philosopher(int i)                   /* i: philosopher number, from 0 to N−1 */
{
      while (TRUE) {                      /* repeat forever */
            think( );                     /* philosopher is thinking */
            take_forks(i);                /* acquire two forks or block */
            eat( );                       /* yum-yum, spaghetti */
            put_forks(i);                 /* put both forks back on table */
      }
}

void take_forks(int i)                    /* i: philosopher number, from 0 to N−1 */
{
      down(&mutex);                       /* enter critical region */
      state[i] = HUNGRY;                  /* record fact that philosopher i is hungry */
      test(i);                            /* try to acquire 2 forks */
      up(&mutex);                         /* exit critical region */
      down(&s[i]);                        /* block if forks were not acquired */
}

void put_forks(i)                         /* i: philosopher number, from 0 to N−1 */
{
      down(&mutex);                       /* enter critical region */
      state[i] = THINKING;                /* philosopher has finished eating */
      test(LEFT);                         /* see if left neighbor can now eat */
      test(RIGHT);                        /* see if right neighbor can now eat */
      up(&mutex);                         /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N−1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

Figure 2-47. A solution to the dining philosophers problem.

2.5.2 The Readers and Writers Problem

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem (Courtois et al., 1971), which models access to a database. Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers. The question is how do you program the readers and the writers? One solution is shown in Fig. 2-48.

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
      while (TRUE) {                    /* repeat forever */
            down(&mutex);               /* get exclusive access to rc */
            rc = rc + 1;                /* one reader more now */
            if (rc == 1) down(&db);     /* if this is the first reader ... */
            up(&mutex);                 /* release exclusive access to rc */
            read_data_base( );          /* access the data */
            down(&mutex);               /* get exclusive access to rc */
            rc = rc − 1;                /* one reader fewer now */
            if (rc == 0) up(&db);       /* if this is the last reader ... */
            up(&mutex);                 /* release exclusive access to rc */
            use_data_read( );           /* noncritical region */
      }
}


void writer(void)
{
      while (TRUE) {                    /* repeat forever */
            think_up_data( );           /* noncritical region */
            down(&db);                  /* get exclusive access */
            write_data_base( );         /* update the data */
            up(&db);                    /* release exclusive access */
      }
}
```

Figure 2-48. A solution to the readers and writers problem.

In this solution, the first reader to get access to the database does a down on the semaphore *db*. Subsequent readers merely increment a counter, *rc*. As readers leave, they decrement the counter, and the last to leave does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

The solution presented here implicitly contains a subtle decision worth noting. Suppose that while a reader is using the database, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. Additional readers can also be admitted if they come along.

Now suppose a writer shows up. The writer may not be admitted to the database, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 sec, and each reader takes 5 sec to do its work, the writer will never get in.

To avoid this situation, the program could be written slightly differently: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance. Courtois et al. present a solution that gives priority to writers. For details, we refer you to the paper.

2.6 RESEARCH ON PROCESSES AND THREADS

In Chap. 1, we looked at some of the current research in operating system structure. In this and subsequent chapters we will look at more narrowly focused research, starting with processes. As will become clear in time, some subjects are much more settled than others. Most of the research tends to be on the new topics, rather than ones that have been around for decades.

The concept of a process is an example of something that is fairly well settled. Almost every system has some notion of a process as a container for grouping together related resources such as an address space, threads, open files, protection permissions, and so on. Different systems do the grouping slightly differently, but these are just engineering differences. The basic idea is not very controversial any more, and there is little new research on the subject of processes.

Threads are a newer idea than processes, but they, too, have been chewed over quite a bit. Still, the occasional paper about threads appears from time to time, for example, about thread clustering on multiprocessors (Tam et al., 2007), or on how well modern operating systems like Linux scale with many threads and many cores (Boyd-Wickizer, 2010).

One particularly active research area deals with recording and replaying a process' execution (Viennot et al., 2013). Replaying helps developers track down hard-to-find bugs and security experts to investigate incidents.

Similarly, much research in the operating systems community these days focuses on security issues. Numerous incidents have demonstrated that users need better protection from attackers (and, occasionally, from themselves). One approach is to track and restrict carefully the information flows in an operating system (Giffin et al., 2012).

Scheduling (both uniprocessor and multiprocessor) is still a topic near and dear to the heart of some researchers. Some topics being researched include energy-efficient scheduling on mobile devices (Yuan and Nahrstedt, 2006), hyperthreading-aware scheduling (Bulpin and Pratt, 2005), and bias-aware scheduling (Koufaty, 2010). With increasing computation on underpowered, battery-constrained smartphones, some researchers propose to migrate the process to a more powerful server in the cloud, as and when useful (Gordon et al., 2012). However, few actual system designers are walking around all day wringing their hands for lack of a decent thread-scheduling algorithm, so it appears that this type of research is more researcher-push than demand-pull. All in all, processes, threads, and scheduling are not hot topics for research as they once were. The research has moved on to topics like power management, virtualization, clouds, and security.

2.7 SUMMARY

To hide the effects of interrupts, operating systems provide a conceptual model consisting of sequential processes running in parallel. Processes can be created and terminated dynamically. Each process has its own address space.

For some applications it is useful to have multiple threads of control within a single process. These threads are scheduled independently and each one has its own stack, but all the threads in a process share a common address space. Threads can be implemented in user space or in the kernel.

Processes can communicate with one another using interprocess communication primitives, for example, semaphores, monitors, or messages. These primitives are used to ensure that no two processes are ever in their critical regions at the same time, a situation that leads to chaos. A process can be running, runnable, or blocked and can change state when it or another process executes one of the interprocess communication primitives. Interthread communication is similar.

Interprocess communication primitives can be used to solve such problems as the producer-consumer, dining philosophers, and reader-writer. Even with these primitives, care has to be taken to avoid errors and deadlocks.

A great many scheduling algorithms have been studied. Some of these are primarily used for batch systems, such as shortest-job-first scheduling. Others are common in both batch systems and interactive systems. These algorithms include round robin, priority scheduling, multilevel queues, guaranteed scheduling, lottery scheduling, and fair-share scheduling. Some systems make a clean separation between the scheduling mechanism and the scheduling policy, which allows users to have control of the scheduling algorithm.

PROBLEMS

1. In Fig. 2-2, three process states are shown. In theory, with three states, there could be six transitions, two out of each state. However, only four transitions are shown. Are there any circumstances in which either or both of the missing transitions might occur?
2. Suppose that you were to design an advanced computer architecture that did process switching in hardware, instead of having interrupts. What information would the CPU need? Describe how the hardware process switching might work.
3. On all current computers, at least part of the interrupt handlers are written in assembly language. Why?
4. When an interrupt or a system call transfers control to the operating system, a kernel stack area separate from the stack of the interrupted process is generally used. Why?
5. A computer system has enough room to hold five programs in its main memory. These programs are idle waiting for I/O half the time. What fraction of the CPU time is wasted?
6. A computer has 4 GB of RAM of which the operating system occupies 512 MB. The processes are all 256 MB (for simplicity) and have the same characteristics. If the goal is 99% CPU utilization, what is the maximum I/O wait that can be tolerated?
7. Multiple jobs can run in parallel and finish faster than if they had run sequentially. Suppose that two jobs, each needing 20 minutes of CPU time, start simultaneously. How long will the last one take to complete if they run sequentially? How long if they run in parallel? Assume 50% I/O wait.
8. Consider a multiprogrammed system with degree of 6 (i.e., six programs in memory at the same time). Assume that each process spends 40% of its time waiting for I/O. What will be the CPU utilization?
9. Assume that you are trying to download a large 2-GB file from the Internet. The file is available from a set of mirror servers, each of which can deliver a subset of the file's bytes; assume that a given request specifies the starting and ending bytes of the file. Explain how you might use threads to improve the download time.
10. In the text it was stated that the model of Fig. 2-11(a) was not suited to a file server using a cache in memory. Why not? Could each process have its own cache?
11. If a multithreaded process forks, a problem occurs if the child gets copies of all the parent's threads. Suppose that one of the original threads was waiting for keyboard input. Now two threads are waiting for keyboard input, one in each process. Does this problem ever occur in single-threaded processes?
12. In Fig. 2-8, a multithreaded Web server is shown. If the only way to read from a file is the normal blocking read system call, do you think user-level threads or kernel-level threads are being used for the Web server? Why?
13. In the text, we described a multithreaded Web server, showing why it is better than a single-threaded server and a finite-state machine server. Are there any circumstances in which a single-threaded server might be better? Give an example.

14. In Fig. 2-12 the register set is listed as a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.

15. Why would a thread ever voluntarily give up the CPU by calling *thread_yield*? After all, since there is no periodic clock interrupt, it may never get the CPU back.

16. Can a thread ever be preempted by a clock interrupt? If so, under what circumstances? If not, why not?

17. In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server. It takes 12 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in the block cache. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. How many requests/sec can the server handle if it is single threaded? If it is multithreaded?

18. What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?

19. In Fig. 2-15 the thread creations and messages printed by the threads are interleaved at random. Is there a way to force the order to be strictly thread 1 created, thread 1 prints message, thread 1 exits, thread 2 created, thread 2 prints message, thread 2 exists, and so on? If so, how? If not, why not?

20. In the discussion on global variables in threads, we used a procedure *create_global* to allocate storage for a pointer to the variable, rather than the variable itself. Is this essential, or could the procedures work with the values themselves just as well?

21. Consider a system in which threads are implemented entirely in user space, with the run-time system getting a clock interrupt once a second. Suppose that a clock interrupt occurs while some thread is executing in the run-time system. What problem might occur? Can you suggest a way to solve it?

22. Suppose that an operating system does not have anything like the select system call to see in advance if it is safe to read from a file, pipe, or device, but it does allow alarm clocks to be set that interrupt blocked system calls. Is it possible to implement a threads package in user space under these conditions? Discuss.

23. Does the busy waiting solution using the *turn* variable (Fig. 2-23) work when the two processes are running on a shared-memory multiprocessor, that is, two CPUs sharing a common memory?

24. Does Peterson's solution to the mutual-exclusion problem shown in Fig. 2-24 work when process scheduling is preemptive? How about when it is nonpreemptive?

25. Can the priority inversion problem discussed in Sec. 2.3.4 happen with user-level threads? Why or why not?

26. In Sec. 2.3.4, a situation with a high-priority process, *H*, and a low-priority process, *L*, was described, which led to *H* looping forever. Does the same problem occur if round-robin scheduling is used instead of priority scheduling? Discuss.

27. In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used? Explain.

28. When a computer is being developed, it is usually first simulated by a program that runs one instruction at a time. Even multiprocessors are simulated strictly sequentially like this. Is it possible for a race condition to occur when there are no simultaneous events like this?

29. The producer-consumer problem can be extended to a system with multiple producers and consumers that write (or read) to (from) one shared buffer. Assume that each producer and consumer runs in its own thread. Will the solution presented in Fig. 2-28, using semaphores, work for this system?

30. Consider the following solution to the mutual-exclusion problem involving two processes *P0* and *P1*. Assume that the variable turn is initialized to 0. Process *P0*'s code is presented below.

```
/* Other code */

while (turn != 0) { } /* Do nothing and wait. */
Critical Section /* . . . */
turn = 0;

/* Other code */
```

For process *P1*, replace 0 by 1 in above code. Determine if the solution meets *all* the required conditions for a correct mutual-exclusion solution.

31. How could an operating system that can disable interrupts implement semaphores?

32. Show how counting semaphores (i.e., semaphores that can hold an arbitrary value) can be implemented using only binary semaphores and ordinary machine instructions.

33. If a system has only two processes, does it make sense to use a barrier to synchronize them? Why or why not?

34. Can two threads in the same process synchronize using a kernel semaphore if the threads are implemented by the kernel? What if they are implemented in user space? Assume that no threads in any other processes have access to the semaphore. Discuss your answers.

35. Synchronization within monitors uses condition variables and two special operations, wait and signal. A more general form of synchronization would be to have a single primitive, waituntil, that had an arbitrary Boolean predicate as parameter. Thus, one could say, for example,

waituntil $x < 0$ or $y + z < n$

The signal primitive would no longer be needed. This scheme is clearly more general than that of Hoare or Brinch Hansen, but it is not used. Why not? (*Hint*: Think about the implementation.)

36. A fast-food restaurant has four kinds of employees: (1) order takers, who take customers' orders; (2) cooks, who prepare the food; (3) packaging specialists, who stuff the food into bags; and (4) cashiers, who give the bags to customers and take their money. Each employee can be regarded as a communicating sequential process. What form of interprocess communication do they use? Relate this model to processes in UNIX.

37. Suppose that we have a message-passing system using mailboxes. When sending to a full mailbox or trying to receive from an empty one, a process does not block. Instead, it gets an error code back. The process responds to the error code by just trying again, over and over, until it succeeds. Does this scheme lead to race conditions?

38. The CDC 6600 computers could handle up to 10 I/O processes simultaneously using an interesting form of round-robin scheduling called processor sharing. A process switch occurred after each instruction, so instruction 1 came from process 1, instruction 2 came from process 2, etc. The process switching was done by special hardware, and the overhead was zero. If a process needed *T* sec to complete in the absence of competition, how much time would it need if processor sharing was used with *n* processes?

39. Consider the following piece of C code:

```
void main( ) {
    fork( );
    fork( );
    exit( );
}
```

How many child processes are created upon execution of this program?

40. Round-robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen if a process occurred twice in the list? Can you think of any reason for allowing this?

41. Can a measure of whether a process is likely to be CPU bound or I/O bound be determined by analyzing source code? How can this be determined at run time?

42. Explain how time quantum value and context switching time affect each other, in a round-robin scheduling algorithm.

43. Measurements of a certain system have shown that the average process runs for a time *T* before blocking on I/O. A process switch requires a time *S*, which is effectively wasted (overhead). For round-robin scheduling with quantum *Q*, give a formula for the CPU efficiency for each of the following:

(a) $Q = \infty$
(b) $Q > T$
(c) $S < Q < T$
(d) $Q = S$
(e) Q nearly 0

44. Five jobs are waiting to be run. Their expected run times are 9, 6, 3, 5, and *X*. In what order should they be run to minimize average response time? (Your answer will depend on *X*.)

45. Five batch jobs. *A* through *E*, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead.

(a) Round robin.
(b) Priority scheduling.
(c) First-come, first-served (run in order 10, 6, 2, 4, 8).
(d) Shortest job first.

For (a), assume that the system is multiprogrammed, and that each job gets its fair share of the CPU. For (b) through (d), assume that only one job at a time runs, until it finishes. All jobs are completely CPU bound.

46. A process running on CTSS needs 30 quanta to complete. How many times must it be swapped in, including the very first time (before it has run at all)?

47. Consider a real-time system with two voice calls of periodicity 5 msec each with CPU time per call of 1 msec, and one video stream of periodicity 33 ms with CPU time per call of 11 msec. Is this system schedulable?

48. For the above problem, can another video stream be added and have the system still be schedulable?

49. The aging algorithm with $a = 1/2$ is being used to predict run times. The previous four runs, from oldest to most recent, are 40, 20, 40, and 15 msec. What is the prediction of the next time?

50. A soft real-time system has four periodic events with periods of 50, 100, 200, and 250 msec each. Suppose that the four events require 35, 20, 10, and x msec of CPU time, respectively. What is the largest value of x for which the system is schedulable?

51. In the dining philosophers problem, let the following protocol be used: An even-numbered philosopher always picks up his left fork before picking up his right fork; an odd-numbered philosopher always picks up his right fork before picking up his left fork. Will this protocol guarantee deadlock-free operation?

52. A real-time system needs to handle two voice calls that each run every 6 msec and consume 1 msec of CPU time per burst, plus one video at 25 frames/sec, with each frame requiring 20 msec of CPU time. Is this system schedulable?

53. Consider a system in which it is desired to separate policy and mechanism for the scheduling of kernel threads. Propose a means of achieving this goal.

54. In the solution to the dining philosophers problem (Fig. 2-47), why is the state variable set to *HUNGRY* in the procedure *take_forks*?

55. Consider the procedure *put_forks* in Fig. 2-47. Suppose that the variable *state*[*i*] was set to *THINKING* *after* the two calls to *test*, rather than *before*. How would this change affect the solution?

56. The readers and writers problem can be formulated in several ways with regard to which category of processes can be started when. Carefully describe three different variations of the problem, each one favoring (or not favoring) some category of processes. For each variation, specify what happens when a reader or a writer becomes ready to access the database, and what happens when a process is finished.

57. Write a shell script that produces a file of sequential numbers by reading the last number in the file, adding 1 to it, and then appending it to the file. Run one instance of the script in the background and one in the foreground, each accessing the same file. How long does it take before a race condition manifests itself? What is the critical region? Modify the script to prevent the race. (*Hint*: use

```
ln file file.lock
```

to lock the data file.)

58. Assume that you have an operating system that provides semaphores. Implement a message system. Write the procedures for sending and receiving messages.

59. Solve the dining philosophers problem using monitors instead of semaphores.

60. Suppose that a university wants to show off how politically correct it is by applying the U.S. Supreme Court's "Separate but equal is inherently unequal" doctrine to gender as well as race, ending its long-standing practice of gender-segregated bathrooms on campus. However, as a concession to tradition, it decrees that when a woman is in a bathroom, other women may enter, but no men, and vice versa. A sign with a sliding marker on the door of each bathroom indicates which of three possible states it is currently in:

- Empty
- Women present
- Men present

In some programming language you like, write the following procedures: *woman_wants_to_enter*, *man_wants_to_enter*, *woman_leaves*, *man_leaves*. You may use whatever counters and synchronization techniques you like.

61. Rewrite the program of Fig. 2-23 to handle more than two processes.

62. Write a producer-consumer problem that uses threads and shares a common buffer. However, do not use semaphores or any other synchronization primitives to guard the shared data structures. Just let each thread access them when it wants to. Use sleep and wakeup to handle the full and empty conditions. See how long it takes for a fatal race condition to occur. For example, you might have the producer print a number once in a while. Do not print more than one number every minute because the I/O could affect the race conditions.

63. A process can be put into a round-robin queue more than once to give it a higher priority. Running multiple instances of a program each working on a different part of a data pool can have the same effect. First write a program that tests a list of numbers for primality. Then devise a method to allow multiple instances of the program to run at once in such a way that no two instances of the program will work on the same number. Can you in fact get through the list faster by running multiple copies of the program? Note that your results will depend upon what else your computer is doing; on a personal computer running only instances of this program you would not expect an improvement, but on a system with other processes, you should be able to grab a bigger share of the CPU this way.

64. The objective of this exercise is to implement a multithreaded solution to find if a given number is a perfect number. N is a perfect number if the sum of all its factors, excluding itself, is N; examples are 6 and 28. The input is an integer, N. The output is true if the number is a perfect number and false otherwise. The main program will read the numbers *N* and *P* from the command line. The main process will spawn a set of *P* threads. The numbers from 1 to *N* will be partitioned among these threads so that two threads do not work on the name number. For each number in this set, the thread will determine if the number is a factor of *N*. If it is, it adds the number to a shared buffer that stores factors of *N*. The parent process waits till all the threads complete. Use the appropriate synchronization primitive here. The parent will then determine if the input number is perfect, that is, if *N* is a sum of all its factors and then report accordingly. (**Note**: You can make the computation faster by restricting the numbers searched from 1 to the square root of *N*.)

65. Implement a program to count the frequency of words in a text file. The text file is partitioned into *N* segments. Each segment is processed by a separate thread that outputs the intermediate frequency count for its segment. The main process waits until all the threads complete; then it computes the consolidated word-frequency data based on the individual threads' output.

3

MEMORY MANAGEMENT

Main memory (RAM) is an important resource that must be very carefully managed. While the average home computer nowadays has 10,000 times more memory than the IBM 7094, the largest computer in the world in the early 1960s, programs are getting bigger faster than memories. To paraphrase Parkinson's Law, "Programs expand to fill the memory available to hold them." In this chapter we will study how operating systems create abstractions from memory and how they manage them.

What every programmer would like is a private, infinitely large, infinitely fast memory that is also nonvolatile, that is, does not lose its contents when the electric power is switched off. While we are at it, why not make it inexpensive, too? Unfortunately, technology does not provide such memories at present. Maybe you will discover how to do it.

What is the second choice? Over the years, people discovered the concept of a **memory hierarchy**, in which computers have a few megabytes of very fast, expensive, volatile cache memory, a few gigabytes of medium-speed, medium-priced, volatile main memory, and a few terabytes of slow, cheap, nonvolatile magnetic or solid-state disk storage, not to mention removable storage, such as DVDs and USB sticks. It is the job of the operating system to abstract this hierarchy into a useful model and then manage the abstraction.

The part of the operating system that manages (part of) the memory hierarchy is called the **memory manager**. Its job is to efficiently manage memory: keep track of which parts of memory are in use, allocate memory to processes when they need it, and deallocate it when they are done.

In this chapter we will investigate several different memory management models, ranging from very simple to highly sophisticated. Since managing the lowest level of cache memory is normally done by the hardware, the focus of this chapter will be on the programmer's model of main memory and how it can be managed. The abstractions for, and the management of, permanent storage—the disk—are the subject of the next chapter. We will first look at the simplest possible schemes and then gradually progress to more and more elaborate ones.

3.1 NO MEMORY ABSTRACTION

The simplest memory abstraction is to have no abstraction at all. Early mainframe computers (before 1960), early minicomputers (before 1970), and early personal computers (before 1980) had no memory abstraction. Every program simply saw the physical memory. When a program executed an instruction like

```
MOV REGISTER1,1000
```

the computer just moved the contents of physical memory location 1000 to *REGISTER1*. Thus, the model of memory presented to the programmer was simply physical memory, a set of addresses from 0 to some maximum, each address corresponding to a cell containing some number of bits, commonly eight.

Under these conditions, it was not possible to have two running programs in memory at the same time. If the first program wrote a new value to, say, location 2000, this would erase whatever value the second program was storing there. Nothing would work and both programs would crash almost immediately.

Even with the model of memory being just physical memory, several options are possible. Three variations are shown in Fig. 3-1. The operating system may be at the bottom of memory in RAM (Random Access Memory), as shown in Fig. 3-1(a), or it may be in ROM (Read-Only Memory) at the top of memory, as shown in Fig. 3-1(b), or the device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below, as shown in Fig. 3-1(c). The first model was formerly used on mainframes and minicomputers but is rarely used any more. The second model is used on some handheld computers and embedded systems. The third model was used by early personal computers (e.g., running MS-DOS), where the portion of the system in the ROM is called the **BIOS** (Basic Input Output System). Models (a) and (c) have the disadvantage that a bug in the user program can wipe out the operating system, possibly with disastrous results.

When the system is organized in this way, generally only one process at a time can be running. As soon as the user types a command, the operating system copies the requested program from disk to memory and executes it. When the process finishes, the operating system displays a prompt character and waits for a user new command. When the operating system receives the command, it loads a new program into memory, overwriting the first one.

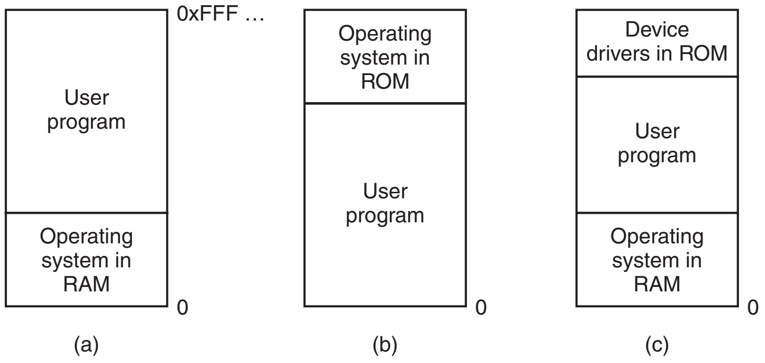


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

One way to get some parallelism in a system with no memory abstraction is to program with multiple threads. Since all threads in a process are supposed to see the same memory image, the fact that they are forced to is not a problem. While this idea works, it is of limited use since what people often want is *unrelated* programs to be running at the same time, something the threads abstraction does not provide. Furthermore, any system that is so primitive as to provide no memory abstraction is unlikely to provide a threads abstraction.

Running Multiple Programs Without a Memory Abstraction

However, even with no memory abstraction, it is possible to run multiple programs at the same time. What the operating system has to do is save the entire contents of memory to a disk file, then bring in and run the next program. As long as there is only one program at a time in memory, there are no conflicts. This concept (swapping) will be discussed below.

With the addition of some special hardware, it is possible to run multiple programs concurrently, even without swapping. The early models of the IBM 360 solved the problem as follows. Memory was divided into 2-KB blocks and each was assigned a 4-bit protection key held in special registers inside the CPU. A machine with a 1-MB memory needed only 512 of these 4-bit registers for a total of 256 bytes of key storage. The PSW (Program Status Word) also contained a 4-bit key. The 360 hardware trapped any attempt by a running process to access memory with a protection code different from the PSW key. Since only the operating system could change the protection keys, user processes were prevented from interfering with one another and with the operating system itself.

Nevertheless, this solution had a major drawback, depicted in Fig. 3-2. Here we have two programs, each 16 KB in size, as shown in Fig. 3-2(a) and (b). The former is shaded to indicate that it has a different memory key than the latter. The first program starts out by jumping to address 24, which contains a MOV instruction. The second program starts out by jumping to address 28, which contains a CMP instruction. The instructions that are not relevant to this discussion are not shown. When the two programs are loaded consecutively in memory starting at address 0, we have the situation of Fig. 3-2(c). For this example, we assume the operating system is in high memory and thus not shown.

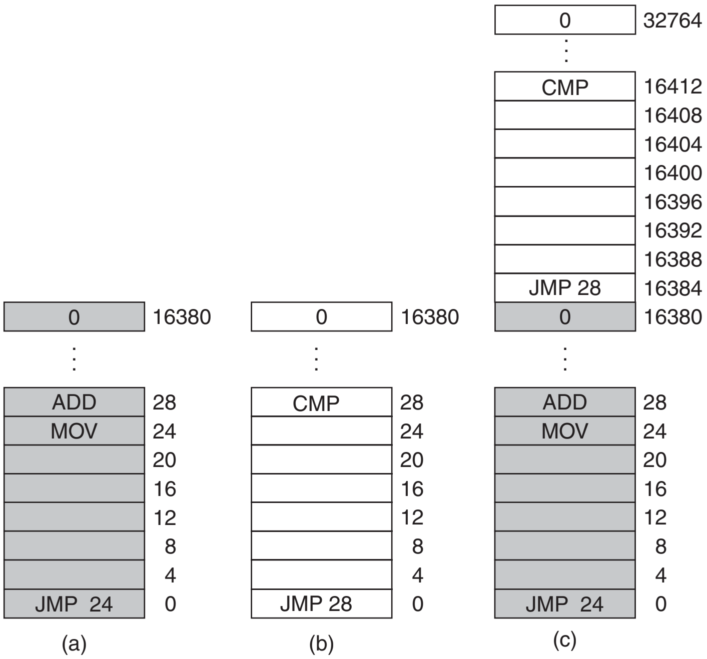


Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

After the programs are loaded, they can be run. Since they have different memory keys, neither one can damage the other. But the problem is of a different nature. When the first program starts, it executes the JMP 24 instruction, which jumps to the instruction, as expected. This program functions normally.

However, after the first program has run long enough, the operating system may decide to run the second program, which has been loaded above the first one, at address 16,384. The first instruction executed is JMP 28, which jumps to the ADD instruction in the first program, instead of the CMP instruction it is supposed to jump to. The program will most likely crash in well under 1 sec.

The core problem here is that the two programs both reference absolute physical memory. That is not what we want at all. What we want is that each program can reference a private set of addresses local to it. We will show how this can be acomplished shortly. What the IBM 360 did as a stop-gap solution was modify the second program on the fly as it loaded it into memory using a technique known as **static relocation**. It worked like this. When a program was loaded at address 16,384, the constant 16,384 was added to every program address during the load process (so “JMP 28” became “JMP 16,412”, etc.).While this mechanism works if done right, it is not a very general solution and slows down loading. Furthermore, it requires extra information in all executable programs to indicate which words contain (relocatable) addresses and which do not. After all, the “28” in Fig. 3-2(b) has to be relocated but an instruction like

```
MOV REGISTER1,28
```

which moves the number 28 to *REGISTER1* must not be relocated. The loader needs some way to tell what is an address and what is a constant.

Finally, as we pointed out in Chap. 1, history tends to repeat itself in the computer world. While direct addressing of physical memory is but a distant memory on mainframes, minicomputers, desktop computers, notebooks, and smartphones, the lack of a memory abstraction is still common in embedded and smart card systems. Devices such as radios, washing machines, and microwave ovens are all full of software (in ROM) these days, and in most cases the software addresses absolute memory. This works because all the programs are known in advance and users are not free to run their own software on their toaster.

While high-end embedded systems (such as smartphones) have elaborate operating systems, simpler ones do not. In some cases, there is an operating system, but it is just a library that is linked with the application program and provides system calls for performing I/O and other common tasks. The **e-Cos** operating system is a common example of an operating system as library.

3.2 A MEMORY ABSTRACTION: ADDRESS SPACES

All in all, exposing physical memory to processes has several major drawbacks. First, if user programs can address every byte of memory, they can easily trash the operating system, intentionally or by accident, bringing the system to a grinding halt (unless there is special hardware like the IBM 360’s lock-and-key scheme). This problem exists even if only one user program (application) is running. Second, with this model, it is difficult to have multiple programs running at once (taking turns, if there is only one CPU). On personal computers, it is common to have several programs open at once (a word processor, an email program, a Web browser), one of them having the current focus, but the others being reactivated at the click of a mouse. Since this situation is difficult to achieve when there is no abstraction from physical memory, something had to be done.

3.2.1 The Notion of an Address Space

Two problems have to be solved to allow multiple applications to be in memory at the same time without interfering with each other: protection and relocation. We looked at a primitive solution to the former used on the IBM 360: label chunks of memory with a protection key and compare the key of the executing process to that of every memory word fetched. However, this approach by itself does not solve the latter problem, although it can be solved by relocating programs as they are loaded, but this is a slow and complicated solution.

A better solution is to invent a new abstraction for memory: the address space. Just as the process concept creates a kind of abstract CPU to run programs, the address space creates a kind of abstract memory for programs to live in. An **address space** is the set of addresses that a process can use to address memory. Each process has its own address space, independent of those belonging to other processes (except in some special circumstances where processes want to share their address spaces).

The concept of an address space is very general and occurs in many contexts. Consider telephone numbers. In the United States and many other countries, a local telephone number is usually a 7-digit number. The address space for telephone numbers thus runs from 0,000,000 to 9,999,999, although some numbers, such as those beginning with 000 are not used. With the growth of smartphones, modems, and fax machines, this space is becoming too small, in which case more digits have to be used. The address space for I/O ports on the x86 runs from 0 to 16383. IPv4 addresses are 32-bit numbers, so their address space runs from 0 to $2^{32} - 1$ (again, with some reserved numbers).

Address spaces do not have to be numeric. The set of *.com* Internet domains is also an address space. This address space consists of all the strings of length 2 to 63 characters that can be made using letters, numbers, and hyphens, followed by *.com*. By now you should get the idea. It is fairly simple.

Somewhat harder is how to give each program its own address space, so address 28 in one program means a different physical location than address 28 in another program. Below we will discuss a simple way that used to be common but has fallen into disuse due to the ability to put much more complicated (and better) schemes on modern CPU chips.

Base and Limit Registers

This simple solution uses a particularly simple version of **dynamic relocation**. What it does is map each process' address space onto a different part of physical memory in a simple way. The classical solution, which was used on machines ranging from the CDC 6600 (the world's first supercomputer) to the Intel 8088 (the heart of the original IBM PC), is to equip each CPU with two special hardware registers, usually called the **base** and **limit** registers. When these registers are used, programs are loaded into consecutive memory locations wherever there is room and without relocation during loading, as shown in Fig. 3-2(c). When a process is run, the base register is loaded with the physical address where its program begins in memory and the limit register is loaded with the length of the program. In Fig. 3-2(c), the base and limit values that would be loaded into these hardware registers when the first program is run are 0 and 16,384, respectively. The values used when the second program is run are 16,384 and 32,768, respectively. If a third 16-KB program were loaded directly above the second one and run, the base and limit registers would be 32,768 and 16,384.

Every time a process references memory, either to fetch an instruction or read or write a data word, the CPU hardware automatically adds the base value to the address generated by the process before sending the address out on the memory bus. Simultaneously, it checks whether the address offered is equal to or greater than the value in the limit register, in which case a fault is generated and the access is aborted. Thus, in the case of the first instruction of the second program in Fig. 3-2(c), the process executes a

```
JMP 28
```

instruction, but the hardware treats it as though it were

```
JMP 16412
```

so it lands on the CMP instruction as expected. The settings of the base and limit registers during the execution of the second program of Fig. 3-2(c) are shown in Fig. 3-3.

Using base and limit registers is an easy way to give each process its own private address space because every memory address generated automatically has the base-register contents added to it before being sent to memory. In many implementations, the base and limit registers are protected in such a way that only the operating system can modify them. This was the case on the CDC 6600, but not on the Intel 8088, which did not even have the limit register. It did have multiple base registers, allowing program text and data, for example, to be independently relocated, but offered no protection from out-of-range memory references.

A disadvantage of relocation using base and limit registers is the need to perform an addition and a comparison on every memory reference. Comparisons can be done fast, but additions are slow due to carry-propagation time unless special addition circuits are used.

3.2.2 Swapping

If the physical memory of the computer is large enough to hold all the processes, the schemes described so far will more or less do. But in practice, the total amount of RAM needed by all the processes is often much more than can fit in memory. On a typical Windows, OS X, or Linux system, something like 50–100

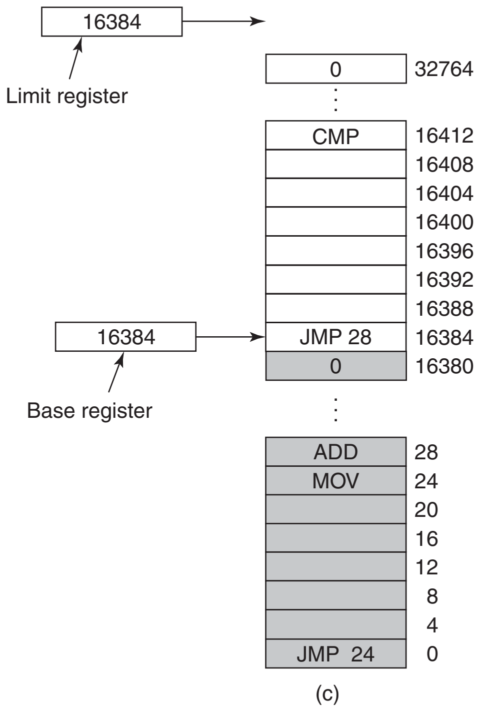


Figure 3-3. Base and limit registers can be used to give each process a separate address space.

processes or more may be started up as soon as the computer is booted. For example, when a Windows application is installed, it often issues commands so that on subsequent system boots, a process will be started that does nothing except check for updates to the application. Such a process can easily occupy 5–10 MB of memory. Other background processes check for incoming mail, incoming network connections, and many other things. And all this is before the first user program is started. Serious user application programs nowadays, like Photoshop, can easily require 500 MB just to boot and many gigabytes once they start processing data. Consequently, keeping all processes in memory all the time requires a huge amount of memory and cannot be done if there is insufficient memory.

Two general approaches to dealing with memory overload have been developed over the years. The simplest strategy, called **swapping**, consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk. Idle processes are mostly stored on disk, so they do not take up any memory when they are not running (although some of them wake up periodically to do their work, then go to sleep again). The other strategy, called **virtual memory**, allows programs to run even when they are only partially in main memory. Below we will study swapping; in Sec. 3.3 we will examine virtual memory.

The operation of a swapping system is illustrated in Fig. 3-4. Initially, only process *A* is in memory. Then processes *B* and *C* are created or swapped in from disk. In Fig. 3-4(d) *A* is swapped out to disk. Then *D* comes in and *B* goes out. Finally *A* comes in again. Since *A* is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution. For example, base and limit registers would work fine here.

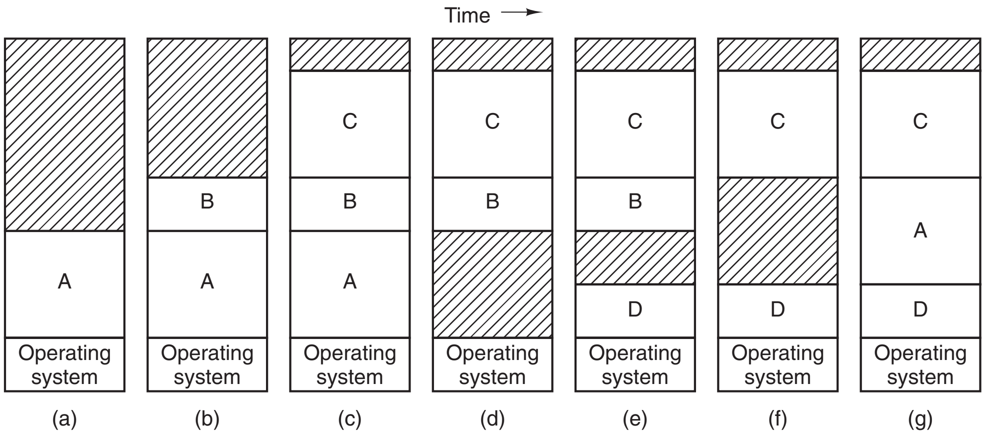


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible. This technique is known as **memory compaction**. It is usually not done because it requires a lot of CPU time. For example, on a 16-GB machine that can copy 8 bytes in 8 nsec, it would take about 16 sec to compact all of memory.

A point that is worth making concerns how much memory should be allocated for a process when it is created or swapped in. If processes are created with a fixed size that never changes, then the allocation is simple: the operating system allocates exactly what is needed, no more and no less.

If, however, processes' data segments can grow, for example, by dynamically allocating memory from a heap, as in many programming languages, a problem occurs whenever a process tries to grow. If a hole is adjacent to the process, it can be allocated and the process allowed to grow into the hole. On the other hand, if the process is adjacent to another process, the growing process will either have to be moved to a hole in memory large enough for it, or one or more processes will have to be swapped out to create a large enough hole. If a process cannot grow in memory and the swap area on the disk is full, the process will have to suspended until some space is freed up (or it can be killed).

If it is expected that most processes will grow as they run, it is probably a good idea to allocate a little extra memory whenever a process is swapped in or moved, to reduce the overhead associated with moving or swapping processes that no longer fit in their allocated memory. However, when swapping processes to disk, only the memory actually in use should be swapped; it is wasteful to swap the extra memory as well. In Fig. 3-5(a) we see a memory configuration in which space for growth has been allocated to two processes.

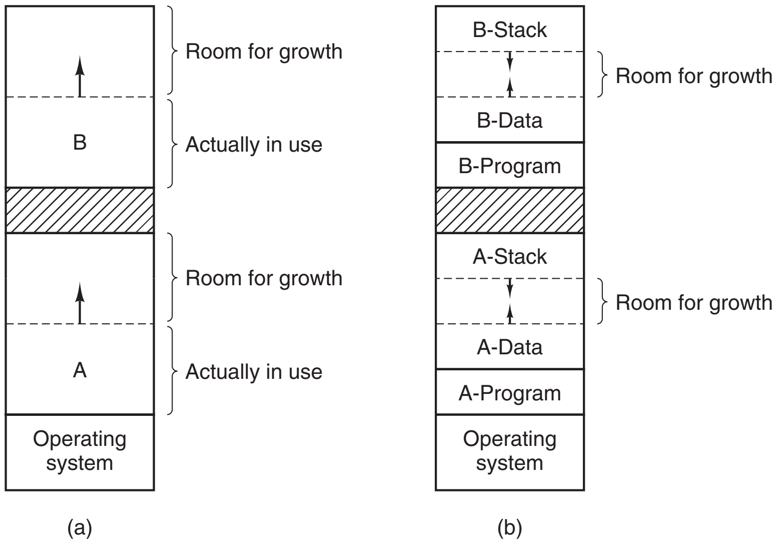


Figure 3-5. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

If processes can have two growing segments—for example, the data segment being used as a heap for variables that are dynamically allocated and released and a stack segment for the normal local variables and return addresses—an alternative arrangement suggests itself, namely that of Fig. 3-5(b). In this figure we see that each process illustrated has a stack at the top of its allocated memory that is growing downward, and a data segment just beyond the program text that is growing upward. The memory between them can be used for either segment. If it runs out, the process will either have to be moved to a hole with sufficient space, swapped out of memory until a large enough hole can be created, or killed.

3.2.3 Managing Free Memory

When memory is assigned dynamically, the operating system must manage it. In general terms, there are two ways to keep track of memory usage: bitmaps and free lists. In this section and the next one we will look at these two methods. In

Chapter 10, we will look at some specific memory allocators used in Linux (like buddy and slab allocators) in more detail.

Memory Management with Bitmaps

With a bitmap, memory is divided into allocation units as small as a few words and as large as several kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure 3-6 shows part of memory and the corresponding bitmap.

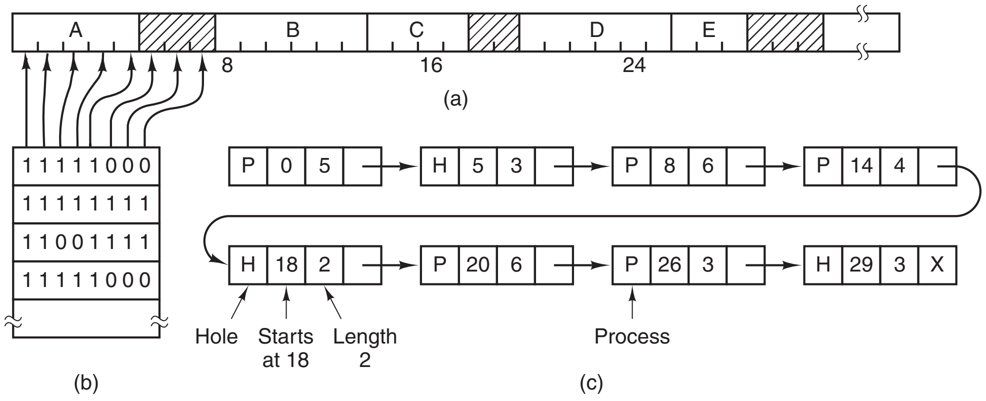


Figure 3-6. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bitmap. However, even with an allocation unit as small as 4 bytes, 32 bits of memory will require only 1 bit of the map. A memory of $32n$ bits will use n map bits, so the bitmap will take up only 1/32 of memory. If the allocation unit is chosen large, the bitmap will be smaller, but appreciable memory may be wasted in the last unit of the process if the process size is not an exact multiple of the allocation unit.

A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit. The main problem is that when it has been decided to bring a k-unit process into memory, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the map. Searching a bitmap for a run of a given length is a slow operation (because the run may straddle word boundaries in the map); this is an argument against bitmaps.

Memory Management with Linked Lists

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment either contains a process or is an empty hole between two processes. The memory of Fig. 3-6(a) is represented in Fig. 3-6(c) as a linked list of segments. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next item.

In this example, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward. A terminating process normally has two neighbors (except when it is at the very top or bottom of memory). These may be either processes or holes, leading to the four combinations shown in Fig. 3-7. In Fig. 3-7(a) updating the list requires replacing a P by an H. In Fig. 3-7(b) and Fig. 3-7(c), two entries are coalesced into one, and the list becomes one entry shorter. In Fig. 3-7(d), three entries are merged and two items are removed from the list.

Since the process table slot for the terminating process will normally point to the list entry for the process itself, it may be more convenient to have the list as a double-linked list, rather than the single-linked list of Fig. 3-6(c). This structure makes it easier to find the previous entry and to see if a merge is possible.

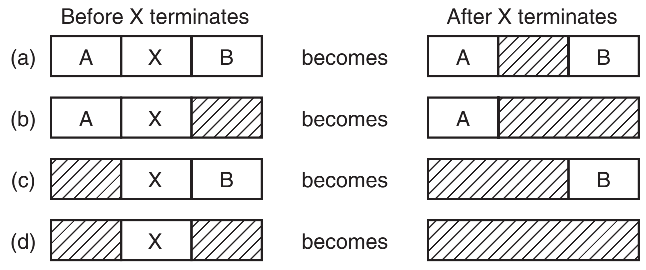


Figure 3-7. Four neighbor combinations for the terminating process, *X*.

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a created process (or an existing process being swapped in from disk). We assume that the memory manager knows how much memory to allocate. The simplest algorithm is **first fit**. The memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

A minor variation of first fit is **next fit**. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does. Simulations by Bays (1977) show that next fit gives slightly worse performance than first fit.

Another well-known and widely used algorithm is **best fit**. Best fit searches the entire list, from beginning to end, and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed, to best match the request and the available holes.

As an example of first fit and best fit, consider Fig. 3-6 again. If a block of size 2 is needed, first fit will allocate the hole at 5, but best fit will allocate the hole at 18.

Best fit is slower than first fit because it must search the entire list every time it is called. Somewhat surprisingly, it also results in more wasted memory than first fit or next fit because it tends to fill up memory with tiny, useless holes. First fit generates larger holes on the average.

To get around the problem of breaking up nearly exact matches into a process and a tiny hole, one could think about **worst fit**, that is, always take the largest available hole, so that the new hole will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

All four algorithms can be speeded up by maintaining separate lists for processes and holes. In this way, all of them devote their full energy to inspecting holes, not processes. The inevitable price that is paid for this speedup on allocation is the additional complexity and slowdown when deallocating memory, since a freed segment has to be removed from the process list and inserted into the hole list.

If distinct lists are maintained for processes and holes, the hole list may be kept sorted on size, to make best fit faster. When best fit searches a list of holes from smallest to largest, as soon as it finds a hole that fits, it knows that the hole is the smallest one that will do the job, hence the best fit. No further searching is needed, as it is with the single-list scheme. With a hole list sorted by size, first fit and best fit are equally fast, and next fit is pointless.

When the holes are kept on separate lists from the processes, a small optimization is possible. Instead of having a separate set of data structures for maintaining the hole list, as is done in Fig. 3-6(c), the information can be stored in the holes. The first word of each hole could be the hole size, and the second word a pointer to the following entry. The nodes of the list of Fig. 3-6(c), which require three words and one bit (P/H), are no longer needed.

Yet another allocation algorithm is **quick fit**, which maintains separate lists for some of the more common sizes requested. For example, it might have a table with *n* entries, in which the first entry is a pointer to the head of a list of 4-KB holes, the second entry is a pointer to a list of 8-KB holes, the third entry a pointer to 12-KB holes, and so on. Holes of, say, 21 KB, could be put either on the 20-KB list or on a special list of odd-sized holes.

With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge with them is possible is quite expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

3.3 VIRTUAL MEMORY

While base and limit registers can be used to create the abstraction of address spaces, there is another problem that has to be solved: managing bloatware. While memory sizes are increasing rapidly, software sizes are increasing much faster. In the 1980s, many universities ran a timesharing system with dozens of (more-or-less satisfied) users running simultaneously on a 4-MB VAX. Now Microsoft recommends having at least 2 GB for 64-bit Windows 8. The trend toward multimedia puts even more demands on memory.

As a consequence of these developments, there is a need to run programs that are too large to fit in memory, and there is certainly a need to have systems that can support multiple programs running simultaneously, each of which fits in memory but all of which collectively exceed memory. Swapping is not an attractive option, since a typical SATA disk has a peak transfer rate of several hundreds of MB/sec, which means it takes seconds to swap out a 1-GB program and the same to swap in a 1-GB program.

The problem of programs larger than memory has been around since the beginning of computing, albeit in limited areas, such as science and engineering (simulating the creation of the universe or even simulating a new aircraft takes a lot of memory). A solution adopted in the 1960s was to split programs into little pieces, called **overlays**. When a program started, all that was loaded into memory was the overlay manager, which immediately loaded and ran overlay 0. When it was done, it would tell the overlay manager to load overlay 1, either above overlay 0 in memory (if there was space for it) or on top of overlay 0 (if there was no space). Some overlay systems were highly complex, allowing many overlays in memory at once. The overlays were kept on the disk and swapped in and out of memory by the overlay manager.

Although the actual work of swapping overlays in and out was done by the operating system, the work of splitting the program into pieces had to be done manually by the programmer. Splitting large programs up into small, modular pieces was time consuming, boring, and error prone. Few programmers were good at this. It did not take long before someone thought of a way to turn the whole job over to the computer.

The method that was devised (Fotheringham, 1961) has come to be known as **virtual memory**. The basic idea behind virtual memory is that each program has its own address space, which is broken up into chunks called **pages**. Each page is a contiguous range of addresses. These pages are mapped onto physical memory, but not all pages have to be in physical memory at the same time to run the program. When the program references a part of its address space that is in physical memory, the hardware performs the necessary mapping on the fly. When the program references a part of its address space that is *not* in physical memory, the operating system is alerted to go get the missing piece and re-execute the instruction that failed.

In a sense, virtual memory is a generalization of the base-and-limit-register idea. The 8088 had separate base registers (but no limit registers) for text and data. With virtual memory, instead of having separate relocation for just the text and data segments, the entire address space can be mapped onto physical memory in fairly small units. We will show how virtual memory is implemented below.

Virtual memory works just fine in a multiprogramming system, with bits and pieces of many programs in memory at once. While a program is waiting for pieces of itself to be read in, the CPU can be given to another process.

3.3.1 Paging

Most virtual memory systems use a technique called **paging**, which we will now describe. On any computer, programs reference a set of memory addresses. When a program executes an instruction like

```
MOV REG,1000
```

it does so to copy the contents of memory address 1000 to REG (or vice versa, depending on the computer). Addresses can be generated using indexing, base registers, segment registers, and other ways.

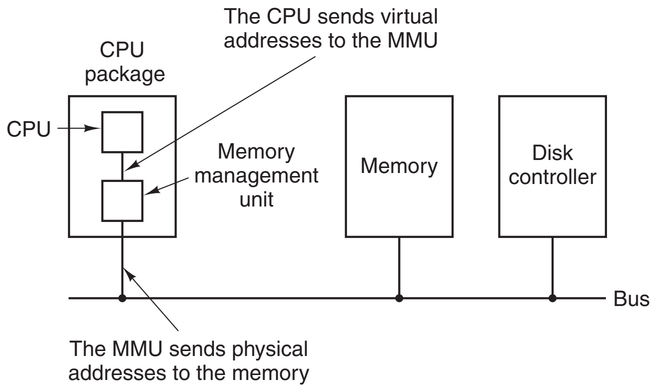

The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

Figure 3-8. The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was years ago.

These program-generated addresses are called **virtual addresses** and form the **virtual address space**. On computers without virtual memory, the virtual address is put directly onto the memory bus and causes the physical memory word with the same address to be read or written. When virtual memory is used, the virtual addresses do not go directly to the memory bus. Instead, they go to an **MMU** (**Memory Management Unit**) that maps the virtual addresses onto the physical memory addresses, as illustrated in Fig. 3-8.

A very simple example of how this mapping works is shown in Fig. 3-9. In this example, we have a computer that generates 16-bit addresses, from 0 up to 64K − 1. These are the virtual addresses. This computer, however, has only 32 KB of physical memory. So although 64-KB programs can be written, they cannot be loaded into memory in their entirety and run. A complete copy of a program's core image, up to 64 KB, must be present on the disk, however, so that pieces can be brought in as needed.

The virtual address space consists of fixed-size units called pages. The corresponding units in the physical memory are called **page frames**. The pages and page frames are generally the same size. In this example they are 4 KB, but page sizes from 512 bytes to a gigabyte have been used in real systems. With 64 KB of virtual address space and 32 KB of physical memory, we get 16 virtual pages and 8 page frames. Transfers between RAM and disk are always in whole pages. Many processors support multiple page sizes that can be mixed and matched as the operating system sees fit. For instance, the x86-64 architecture supports 4-KB, 2-MB, and 1-GB pages, so we could use 4-KB pages for user applications and a single 1-GB page for the kernel. We will see later why it is sometimes better to use a single large page, rather than a large number of small ones.

The notation in Fig. 3-9 is as follows. The range marked 0K–4K means that the virtual or physical addresses in that page are 0 to 4095. The range 4K–8K refers to addresses 4096 to 8191, and so on. Each page contains exactly 4096 addresses starting at a multiple of 4096 and ending one shy of a multiple of 4096.

When the program tries to access address 0, for example, using the instruction

```
MOV REG,0
```

virtual address 0 is sent to the MMU. The MMU sees that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287). It thus transforms the address to 8192 and outputs address 8192 onto the bus. The memory knows nothing at all about the MMU and just sees a request for reading or writing address 8192, which it honors. Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.

Similarly, the instruction

```
MOV REG,8192
```

is effectively transformed into

```
MOV REG,24576
```

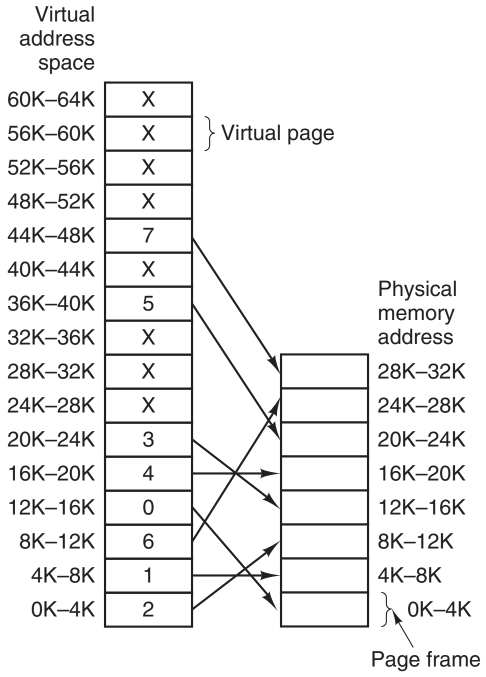


Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

because virtual address 8192 (in virtual page 2) is mapped onto 24576 (in physical page frame 6). As a third example, virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address 12288 + 20 = 12308.

By itself, this ability to map the 16 virtual pages onto any of the eight page frames by setting the MMU's map appropriately does not solve the problem that the virtual address space is larger than the physical memory. Since we have only eight physical page frames, only eight of the virtual pages in Fig. 3-9 are mapped onto physical memory. The others, shown as a cross in the figure, are not mapped. In the actual hardware, a **Present/absent bit** keeps track of which pages are physically present in memory.

What happens if the program references an unmapped address, for example, by using the instruction

```
MOV REG,32780
```

which is byte 12 within virtual page 8 (starting at 32768)? The MMU notices that the page is unmapped (indicated by a cross in the figure) and causes the CPU to trap to the operating system. This trap is called a **page fault**. The operating system picks a little-used page frame and writes its contents back to the disk (if it is not already there). It then fetches (also from the disk) the page that was just referenced into the page frame just freed, changes the map, and restarts the trapped instruction.

For example, if the operating system decided to evict page frame 1, it would load virtual page 8 at physical address 4096 and make two changes to the MMU map. First, it would mark virtual page 1's entry as unmapped, to trap any future accesses to virtual addresses between 4096 and 8191. Then it would replace the cross in virtual page 8's entry with a 1, so that when the trapped instruction is reexecuted, it will map virtual address 32780 to physical address 4108 (4096 + 12).

Now let us look inside the MMU to see how it works and why we have chosen to use a page size that is a power of 2. In Fig. 3-10 we see an example of a virtual address, 8196 (0010000000000100 in binary), being mapped using the MMU map of Fig. 3-9. The incoming 16-bit virtual address is split into a 4-bit page number and a 12-bit offset. With 4 bits for the page number, we can have 16 pages, and with 12 bits for the offset, we can address all 4096 bytes within a page.

The page number is used as an index into the **page table**, yielding the number of the page frame corresponding to that virtual page. If the *Present/absent* bit is 0, a trap to the operating system is caused. If the bit is 1, the page frame number found in the page table is copied to the high-order 3 bits of the output register, along with the 12-bit offset, which is copied unmodified from the incoming virtual address. Together they form a 15-bit physical address. The output register is then put onto the memory bus as the physical memory address.

3.3.2 Page Tables

In a simple implementation, the mapping of virtual addresses onto physical addresses can be summarized as follows: the virtual address is split into a virtual page number (high-order bits) and an offset (low-order bits). For example, with a 16-bit address and a 4-KB page size, the upper 4 bits could specify one of the 16 virtual pages and the lower 12 bits would then specify the byte offset (0 to 4095) within the selected page. However a split with 3 or 5 or some other number of bits for the page is also possible. Different splits imply different page sizes.

The virtual page number is used as an index into the page table to find the entry for that virtual page. From the page table entry, the page frame number (if any) is found. The page frame number is attached to the high-order end of the offset, replacing the virtual page number, to form a physical address that can be sent to the memory.

Thus, the purpose of the page table is to map virtual pages onto page frames. Mathematically speaking, the page table is a function, with the virtual page number as argument and the physical frame number as result. Using the result of this

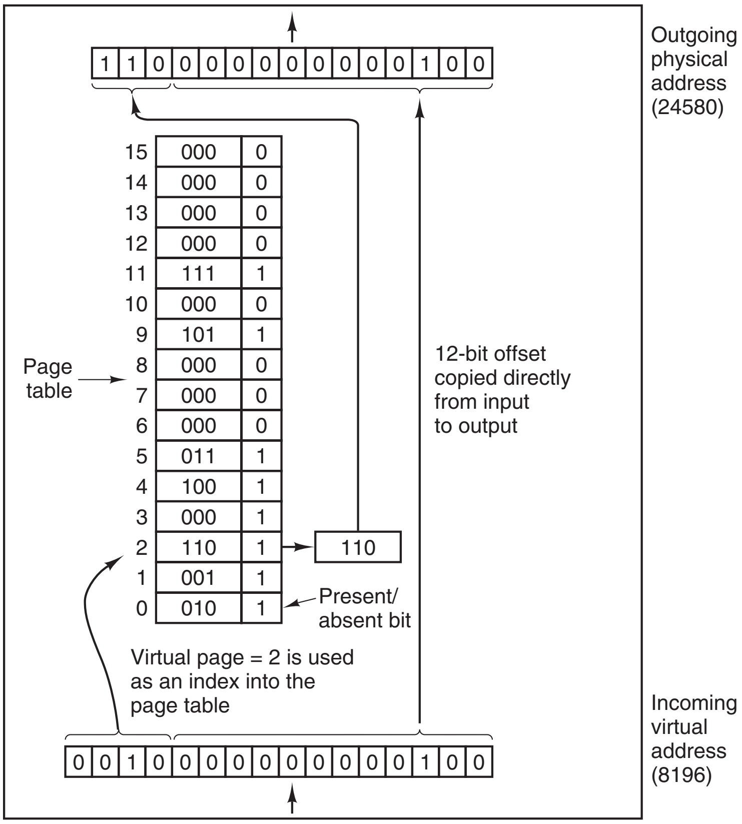


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

function, the virtual page field in a virtual address can be replaced by a page frame field, thus forming a physical memory address.

In this chapter, we worry only about virtual memory and not full virtualization. In other words: no virtual machines yet. We will see in Chap. 7 that each virtual machine requires its own virtual memory and as a result the page table organization becomes much more complicated—involving shadow or nested page tables and more. Even without such arcane configurations, paging and virtual memory are fairly sophisticated, as we shall see.

Structure of a Page Table Entry

Let us now turn from the structure of the page tables in the large, to the details of a single page table entry. The exact layout of an entry in the page table is highly machine dependent, but the kind of information present is roughly the same from machine to machine. In Fig. 3-11 we present a sample page table entry. The size varies from computer to computer, but 32 bits is a common size. The most important field is the *Page frame number*. After all, the goal of the page mapping is to output this value. Next to it we have the *Present/absent* bit. If this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault.

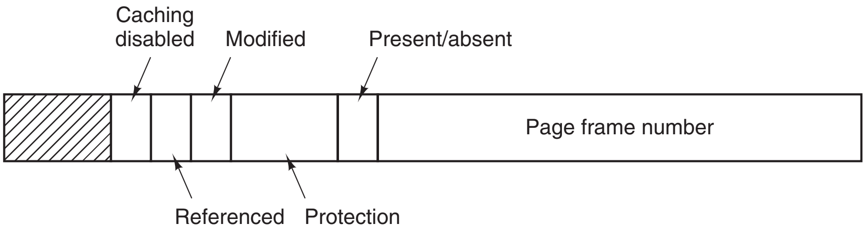


Figure 3-11. A typical page table entry.

The *Protection* bits tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only. A more sophisticated arrangement is having 3 bits, one bit each for enabling reading, writing, and executing the page.

The *Modified* and *Referenced* bits keep track of page usage. When a page is written to, the hardware automatically sets the *Modified* bit. This bit is of value when the operating system decides to reclaim a page frame. If the page in it has been modified (i.e., is "dirty"), it must be written back to the disk. If it has not been modified (i.e., is "clean"), it can just be abandoned, since the disk copy is still valid. The bit is sometimes called the **dirty bit**, since it reflects the page's state.

The *Referenced* bit is set whenever a page is referenced, either for reading or for writing. Its value is used to help the operating system choose a page to evict when a page fault occurs. Pages that are not being used are far better candidates than pages that are, and this bit plays an important role in several of the page replacement algorithms that we will study later in this chapter.

Finally, the last bit allows caching to be disabled for the page. This feature is important for pages that map onto device registers rather than memory. If the operating system is sitting in a tight loop waiting for some I/O device to respond to a command it was just given, it is essential that the hardware keep fetching the word from the device, and not use an old cached copy. With this bit, caching can be turned off. Machines that have a separate I/O space and do not use memory-mapped I/O do not need this bit.

Note that the disk address used to hold the page when it is not in memory is not part of the page table. The reason is simple. The page table holds only that information the hardware needs to translate a virtual address to a physical address.

Information the operating system needs to handle page faults is kept in software tables inside the operating system. The hardware does not need it.

Before getting into more implementation issues, it is worth pointing out again that what virtual memory fundamentally does is create a new abstraction—the address space—which is an abstraction of physical memory, just as a process is an abstraction of the physical processor (CPU). Virtual memory can be implemented by breaking the virtual address space up into pages, and mapping each one onto some page frame of physical memory or having it (temporarily) unmapped. Thus this section is bassically about an abstraction created by the operating system and how that abstraction is managed.

3.3.3 Speeding Up Paging

We have just seen the basics of virtual memory and paging. It is now time to go into more detail about possible implementations. In any paging system, two major issues must be faced:

1. The mapping from virtual address to physical address must be fast.
2. If the virtual address space is large, the page table will be large.

The first point is a consequence of the fact that the virtual-to-physical mapping must be done on every memory reference. All instructions must ultimately come from memory and many of them reference operands in memory as well. Consequently, it is necessary to make one, two, or sometimes more page table references per instruction. If an instruction execution takes, say, 1 nsec, the page table lookup must be done in under 0.2 nsec to avoid having the mapping become a major bottleneck.

The second point follows from the fact that all modern computers use virtual addresses of at least 32 bits, with 64 bits becoming the norm for desktops and laptops. With, say, a 4-KB page size, a 32-bit address space has 1 million pages, and a 64-bit address space has more than you want to contemplate. With 1 million pages in the virtual address space, the page table must have 1 million entries. And remember that each process needs its own page table (because it has its own virtual address space).

The need for large, fast page mapping is a very significant constraint on the way computers are built. The simplest design (at least conceptually) is to have a single page table consisting of an array of fast hardware registers, with one entry for each virtual page, indexed by virtual page number, as shown in Fig. 3-10. When a process is started up, the operating system loads the registers with the process' page table, taken from a copy kept in main memory. During process execution, no more memory references are needed for the page table. The advantages of this method are that it is straightforward and requires no memory references during mapping. A disadvantage is that it is unbearably expensive if the page table is large; it is just not practical most of the time. Another one is that having to load the full page table at every context switch would completely kill performance.

At the other extreme, the page table can be entirely in main memory. All the hardware needs then is a single register that points to the start of the page table. This design allows the virtual-to-physical map to be changed at a context switch by reloading one register. Of course, it has the disadvantage of requiring one or more memory references to read page table entries during the execution of each instruction, making it very slow.

Translation Lookaside Buffers

Let us now look at widely implemented schemes for speeding up paging and for handling large virtual address spaces, starting with the former. The starting point of most optimization techniques is that the page table is in memory. Potentially, this design has an enormous impact on performance. Consider, for example, a 1-byte instruction that copies one register to another. In the absence of paging, this instruction makes only one memory reference, to fetch the instruction. With paging, at least one additional memory reference will be needed, to access the page table. Since execution speed is generally limited by the rate at which the CPU can get instructions and data out of the memory, having to make two memory references per memory reference reduces performance by half. Under these conditions, no one would use paging.

Computer designers have known about this problem for years and have come up with a solution. Their solution is based on the observation that most programs tend to make a large number of references to a small number of pages, and not the other way around. Thus only a small fraction of the page table entries are heavily read; the rest are barely used at all.

The solution that has been devised is to equip computers with a small hardware device for mapping virtual addresses to physical addresses without going through the page table. The device, called a **TLB** (**Translation Lookaside Buffer**) or sometimes an **associative memory**, is illustrated in Fig. 3-12. It is usually inside the MMU and consists of a small number of entries, eight in this example, but rarely more than 256. Each entry contains information about one page, including the virtual page number, a bit that is set when the page is modified, the protection code (read/write/execute permissions), and the physical page frame in which the page is located. These fields have a one-to-one correspondence with the fields in the page table, except for the virtual page number, which is not needed in the page table. Another bit indicates whether the entry is valid (i.e., in use) or not.

An example that might generate the TLB of Fig. 3-12 is a process in a loop that spans virtual pages 19, 20, and 21, so that these TLB entries have protection codes for reading and executing. The main data currently being used (say, an array being processed) are on pages 129 and 130. Page 140 contains the indices used in the array calculations. Finally, the stack is on pages 860 and 861.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 3-12. A TLB to speed up paging.

Let us now see how the TLB functions. When a virtual address is presented to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries simultaneously (i.e., in parallel). Doing so requires special hardware, which all MMUs with TLBs have. If a valid match is found and the access does not violate the protection bits, the page frame is taken directly from the TLB, without going to the page table. If the virtual page number is present in the TLB but the instruction is trying to write on a read-only page, a protection fault is generated.

The interesting case is what happens when the virtual page number is not in the TLB. The MMU detects the miss and does an ordinary page table lookup. It then evicts one of the entries from the TLB and replaces it with the page table entry just looked up. Thus if that page is used again soon, the second time it will result in a TLB hit rather than a miss. When an entry is purged from the TLB, the modified bit is copied back into the page table entry in memory. The other values are already there, except the reference bit. When the TLB is loaded from the page table, all the fields are taken from memory.

Software TLB Management

Up until now, we have assumed that every machine with paged virtual memory has page tables recognized by the hardware, plus a TLB. In this design, TLB management and handling TLB faults are done entirely by the MMU hardware. Traps to the operating system occur only when a page is not in memory.

In the past, this assumption was true. However, many RISC machines, including the SPARC, MIPS, and (the now dead) HP PA, do nearly all of this page management in software. On these machines, the TLB entries are explicitly loaded by the operating system. When a TLB miss occurs, instead of the MMU going to the page tables to find and fetch the needed page reference, it just generates a TLB fault and tosses the problem into the lap of the operating system. The system must find the page, remove an entry from the TLB, enter the new one, and restart the instruction that faulted. And, of course, all of this must be done in a handful of instructions because TLB misses occur much more frequently than page faults.

Surprisingly enough, if the TLB is moderately large (say, 64 entries) to reduce the miss rate, software management of the TLB turns out to be acceptably efficient. The main gain here is a much simpler MMU, which frees up a considerable amount of area on the CPU chip for caches and other features that can improve performance. Software TLB management is discussed by Uhlig et al. (1994).

Various strategies were developed long ago to improve performance on machines that do TLB management in software. One approach attacks both reducing TLB misses and reducing the cost of a TLB miss when it does occur (Bala et al., 1994). To reduce TLB misses, sometimes the operating system can use its intuition to figure out which pages are likely to be used next and to preload entries for them in the TLB. For example, when a client process sends a message to a server process on the same machine, it is very likely that the server will have to run soon. Knowing this, while processing the trap to do the send, the system can also check to see where the server's code, data, and stack pages are and map them in before they get a chance to cause TLB faults.

The normal way to process a TLB miss, whether in hardware or in software, is to go to the page table and perform the indexing operations to locate the page referenced. The problem with doing this search in software is that the pages holding the page table may not be in the TLB, which will cause additional TLB faults during the processing. These faults can be reduced by maintaining a large (e.g., 4-KB) software cache of TLB entries in a fixed location whose page is always kept in the TLB. By first checking the software cache, the operating system can substantially reduce TLB misses.

When software TLB management is used, it is essential to understand the difference between different kinds of misses. A **soft miss** occurs when the page referenced is not in the TLB, but is in memory. All that is needed here is for the TLB to be updated. No disk I/O is needed. Typically a soft miss takes 10–20 machine instructions to handle and can be completed in a couple of nanoseconds. In contrast, a **hard miss** occurs when the page itself is not in memory (and of course, also not in the TLB). A disk access is required to bring in the page, which can take several milliseconds, depending on the disk being used. A hard miss is easily a million times slower than a soft miss. Looking up the mapping in the page table hierarchy is known as a **page table walk**.

Actually, it is worse that that. A miss is not just soft or hard. Some misses are slightly softer (or slightly harder) than other misses. For instance, suppose the page walk does not find the page in the process' page table and the program thus incurs a page fault. There are three possibilities. First, the page may actually be in memory, but not in this process' page table. For instance, the page may have been brought in from disk by another process. In that case, we do not need to access the disk again, but merely map the page appropriately in the page tables. This is a pretty soft miss that is known as a **minor page fault**. Second, a **major page fault** occurs if the page needs to be brought in from disk. Third, it is possible that the program simply accessed an invalid address and no mapping needs to be added in the TLB at all. In that case, the operating system typically kills the program with a **segmentation fault**. Only in this case did the program do something wrong. All other cases are automatically fixed by the hardware and/or the operating system—at the cost of some performance.

3.3.4 Page Tables for Large Memories

TLBs can be used to speed up virtual-to-physical address translation over the original page-table-in-memory scheme. But that is not the only problem we have to tackle. Another problem is how to deal with very large virtual address spaces. Below we will discuss two ways of dealing with them.

Multilevel Page Tables

As a first approach, consider the use of a **multilevel page table**. A simple example is shown in Fig. 3-13. In Fig. 3-13(a) we have a 32-bit virtual address that is partitioned into a 10-bit *PT1* field, a 10-bit *PT2* field, and a 12-bit *Offset* field. Since offsets are 12 bits, pages are 4 KB, and there are a total of 2^{20} of them.

The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time. In particular, those that are not needed should not be kept around. Suppose, for example, that a process needs 12 megabytes: the bottom 4 megabytes of memory for program text, the next 4 megabytes for data, and the top 4 megabytes for the stack. In between the top of the data and the bottom of the stack is a gigantic hole that is not used.

In Fig. 3-13(b) we see how the two-level page table works. On the left we see the top-level page table, with 1024 entries, corresponding to the 10-bit *PT1* field. When a virtual address is presented to the MMU, it first extracts the *PT1* field and uses this value as an index into the top-level page table. Each of these 1024 entries in the top-level page table represents 4M because the entire 4-gigabyte (i.e., 32-bit) virtual address space has been chopped into chunks of 4096 bytes.

The entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table. Entry 0 of the top-level page table points to the page table for the program text, entry 1 points to the page table for the data, and entry 1023 points to the page table for the stack. The other (shaded) entries are not used. The *PT2* field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

As an example, consider the 32-bit virtual address 0x00403004 (4,206,596 decimal), which is 12,292 bytes into the data. This virtual address corresponds to *PT1* = 1, *PT2* = 3, and *Offset* = 4. The MMU first uses *PT1* to index into the top-

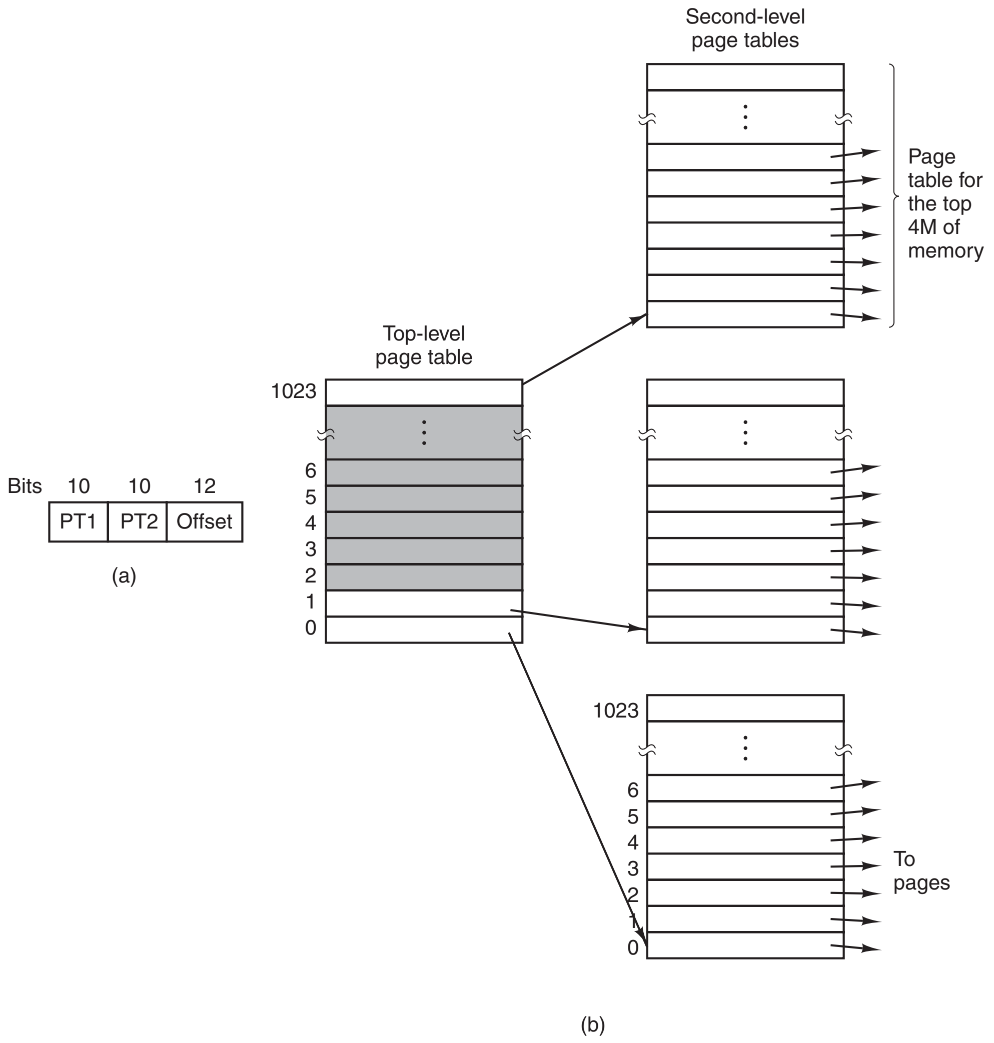


Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

level page table and obtain entry 1, which corresponds to addresses 4M to 8M − 1. It then uses *PT2* to index into the second-level page table just found and extract entry 3, which corresponds to addresses 12288 to 16383 within its 4M chunk (i.e., absolute addresses 4,206,592 to 4,210,687). This entry contains the page frame number of the page containing virtual address 0x00403004. If that page is not in memory, the *Present/absent* bit in the page table entry will have the value zero, causing a page fault. If the page is present in memory, the page frame number taken from the second-level page table is combined with the offset (4) to construct the physical address. This address is put on the bus and sent to memory.

The interesting thing to note about Fig. 3-13 is that although the address space contains over a million pages, only four page tables are needed: the top-level table, and the second-level tables for 0 to 4M (for the program text), 4M to 8M (for the data), and the top 4M (for the stack). The *Present/absent* bits in the remaining 1021 entries of the top-level page table are set to 0, forcing a page fault if they are ever accessed. Should this occur, the operating system will notice that the process is trying to reference memory that it is not supposed to and will take appropriate action, such as sending it a signal or killing it. In this example we have chosen round numbers for the various sizes and have picked *PT1* equal to *PT2*, but in actual practice other values are also possible, of course.

The two-level page table system of Fig. 3-13 can be expanded to three, four, or more levels. Additional levels give more flexibility. For instance, Intel's 32 bit 80386 processor (launched in 1985) was able to address up to 4-GB of memory, using a two-level page table that consisted of a **page directory** whose entries pointed to page tables, which, in turn, pointed to the actual 4-KB page frames. Both the page directory and the page tables each contained 1024 entries, giving a total of $2^{10} \times 2^{10} \times 2^{12} = 2^{32}$ addressable bytes, as desired.

Ten years later, the Pentium Pro introduced another level: the **page directory pointer table**. In addition, it extended each entry in each level of the page table hierarchy from 32 bits to 64 bits, so that it could address memory above the 4-GB boundary. As it had only 4 entries in the page directory pointer table, 512 in each page directory, and 512 in each page table, the total amount of memory it could address was still limited to a maximum of 4 GB. When proper 64-bit support was added to the x86 family (originally by AMD), the additional level *could* have been called the "page directory pointer table pointer" or something equally horri. That would have been perfectly in line with how chip makers tend to name things. Mercifully, they did not do this. The alternative they cooked up, "**page map level 4**," may not be a terribly catchy name either, but at least it is short and a bit clearer. At any rate, these processors now use all 512 entries in all tables, yielding an amount of addressable memory of $2^9 \times 2^9 \times 2^9 \times 2^9 \times 2^{12} = 2^{48}$ bytes. They could have added another level, but they probably thought that 256 TB would be sufficient for a while.

Inverted Page Tables

An alternative to ever-increasing levels in a paging hierarchy is known as **inverted page tables**. They were first used by such processors as the PowerPC, the UltraSPARC, and the Itanium (sometimes referred to as "Itanic," as it was not nearly the success Intel had hoped for). In this design, there is one entry per page frame in real memory, rather than one entry per page of virtual address space. For example, with 64-bit virtual addresses, a 4-KB page size, and 4 GB of RAM, an inverted page table requires only 1,048,576 entries. The entry keeps track of which (process, virtual page) is located in the page frame.

Although inverted page tables save lots of space, at least when the virtual address space is much larger than the physical memory, they have a serious downside: virtual-to-physical translation becomes much harder. When process *n* references virtual page *p*, the hardware can no longer find the physical page by using *p* as an index into the page table. Instead, it must search the entire inverted page table for an entry (*n*, *p*). Furthermore, this search must be done on every memory reference, not just on page faults. Searching a 256K table on every memory reference is not the way to make your machine blindingly fast.

The way out of this dilemma is to make use of the TLB. If the TLB can hold all of the heavily used pages, translation can happen just as fast as with regular page tables. On a TLB miss, however, the inverted page table has to be searched in software. One feasible way to accomplish this search is to have a hash table hashed on the virtual address. All the virtual pages currently in memory that have the same hash value are chained together, as shown in Fig. 3-14. If the hash table has as many slots as the machine has physical pages, the average chain will be only one entry long, greatly speeding up the mapping. Once the page frame number has been found, the new (virtual, physical) pair is entered into the TLB.

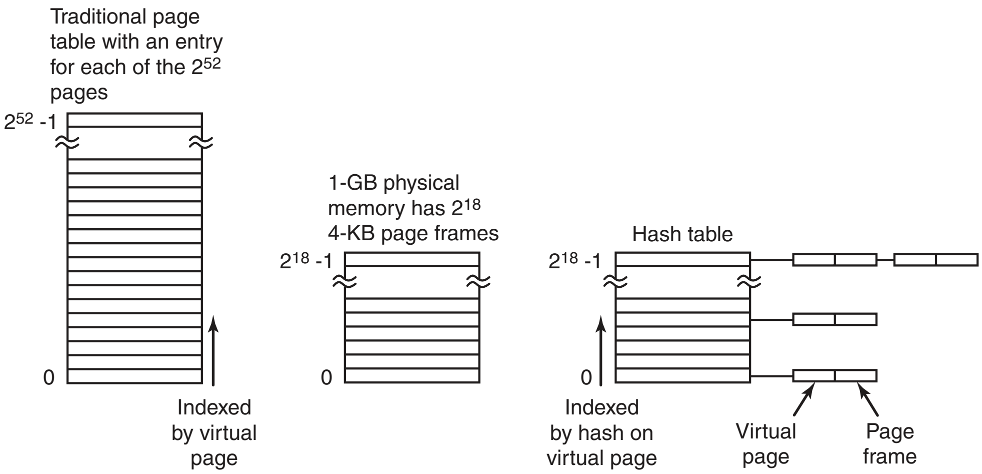

Traditional page table with an entry for each of the 2^52 pages
2^52 -1
0
Indexed by virtual page
1-GB physical memory has 2^18 4-KB page frames
2^18 -1
0
Hash table
2^18 -1
0
Indexed by hash on virtual page
Virtual page
Page frame

Figure 3-14. Comparison of a traditional page table with an inverted page table.

Inverted page tables are common on 64-bit machines because even with a very large page size, the number of page table entries is gigantic. For example, with 4-MB pages and 64-bit virtual addresses, 2^{42} page table entries are needed. Other approaches to handling large virtual memories can be found in Talluri et al. (1995).

3.4 PAGE REPLACEMENT ALGORITHMS

When a page fault occurs, the operating system has to choose a page to evict (remove from memory) to make room for the incoming page. If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted.

While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. Below we will describe some of the most important ones.

It is worth noting that the problem of "page replacement" occurs in other areas of computer design as well. For example, most computers have one or more memory caches consisting of recently used 32-byte or 64-byte memory blocks. When the cache is full, some block has to be chosen for removal. This problem is precisely the same as page replacement except on a shorter time scale (it has to be done in a few nanoseconds, not milliseconds as with page replacement). The reason for the shorter time scale is that cache block misses are satisfied from main memory, which has no seek time and no rotational latency.

A second example is in a Web server. The server can keep a certain number of heavily used Web pages in its memory cache. However, when the memory cache is full and a new page is referenced, a decision has to be made which Web page to evict. The considerations are similar to pages of virtual memory, except that the Web pages are never modified in the cache, so there is always a fresh copy "on disk." In a virtual memory system, pages in main memory may be either clean or dirty.

In all the page replacement algorithms to be studied below, a certain issue arises: when a page is to be evicted from memory, does it have to be one of the faulting process' own pages, or can it be a page belonging to another process? In the former case, we are effectively limiting each process to a fixed number of pages; in the latter case we are not. Both are possibilities. We will come back to this point in Sec. 3.5.1.

3.4.1 The Optimal Page Replacement Algorithm

The best possible page replacement algorithm is easy to describe but impossible to actually implement. It goes like this. At the moment that a page fault occurs, some set of pages is in memory. One of these pages will be referenced on the very next instruction (the page containing that instruction). Other pages may not be referenced until 10, 100, or perhaps 1000 instructions later. Each page can be labeled with the number of instructions that will be executed before that page is first referenced.

The optimal page replacement algorithm says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, removing the former pushes the page fault that will fetch it back as far into the future as possible. Computers, like people, try to put off unpleasant events for as long as they can.

The only problem with this algorithm is that it is unrealizable. At the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next. (We saw a similar situation earlier with the shortest-job-first scheduling algorithm—how can the system tell which job is shortest?) Still, by running a program on a simulator and keeping track of all page references, it is possible to implement optimal page replacement on the *second* run by using the page-reference information collected during the *first* run.

In this way, it is possible to compare the performance of realizable algorithms with the best possible one. If an operating system achieves a performance of, say, only 1% worse than the optimal algorithm, effort spent in looking for a better algorithm will yield at most a 1% improvement.

To avoid any possible confusion, it should be made clear that this log of page references refers only to the one program just measured and then with only one specific input. The page replacement algorithm derived from it is thus specific to that one program and input data. Although this method is useful for evaluating page replacement algorithms, it is of no use in practical systems. Below we will study algorithms that *are* useful on real systems.

3.4.2 The Not Recently Used Page Replacement Algorithm

In order to allow the operating system to collect useful page usage statistics, most computers with virtual memory have two status bits, *R* and *M*, associated with each page. *R* is set whenever the page is referenced (read or written). *M* is set when the page is written to (i.e., modified). The bits are contained in each page table entry, as shown in Fig. 3-11. It is important to realize that these bits must be updated on every memory reference, so it is essential that they be set by the hardware. Once a bit has been set to 1, it stays 1 until the operating system resets it.

If the hardware does not have these bits, they can be simulated using the operating system's page fault and clock interrupt mechanisms. When a process is started up, all of its page table entries are marked as not in memory. As soon as any page is referenced, a page fault will occur. The operating system then sets the *R* bit (in its internal tables), changes the page table entry to point to the correct page, with mode READ ONLY, and restarts the instruction. If the page is subsequently modified, another page fault will occur, allowing the operating system to set the *M* bit and change the page's mode to READ/WRITE.

The *R* and *M* bits can be used to build a simple paging algorithm as follows. When a process is started up, both page bits for all its pages are set to 0 by the operating system. Periodically (e.g., on each clock interrupt), the *R* bit is cleared, to distinguish pages that have not been referenced recently from those that have been.

When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their *R* and *M* bits:

Class 0: not referenced, not modified.
Class 1: not referenced, modified.
Class 2: referenced, not modified.
Class 3: referenced, modified.

Although class 1 pages seem, at first glance, impossible, they occur when a class 3 page has its *R* bit cleared by a clock interrupt. Clock interrupts do not clear the *M* bit because this information is needed to know whether the page has to be rewritten to disk or not. Clearing *R* but not *M* leads to a class 1 page.

The **NRU** (**Not Recently Used**) algorithm removes a page at random from the lowest-numbered nonempty class. Implicit in this algorithm is the idea that it is better to remove a modified page that has not been referenced in at least one clock tick (typically about 20 msec) than a clean page that is in heavy use. The main attraction of NRU is that it is easy to understand, moderately efficient to implement, and gives a performance that, while certainly not optimal, may be adequate.

3.4.3 The First-In, First-Out (FIFO) Page Replacement Algorithm

Another low-overhead paging algorithm is the **FIFO** (**First-In, First-Out**) algorithm. To illustrate how this works, consider a supermarket that has enough shelves to display exactly *k* different products. One day, some company introduces a new convenience food—instant, freeze-dried, organic yogurt that can be reconstituted in a microwave oven. It is an immediate success, so our finite supermarket has to get rid of one old product in order to stock it.

One possibility is to find the product that the supermarket has been stocking the longest (i.e., something it began selling 120 years ago) and get rid of it on the grounds that no one is interested any more. In effect, the supermarket maintains a linked list of all the products it currently sells in the order they were introduced. The new one goes on the back of the list; the one at the front of the list is dropped.

As a page replacement algorithm, the same idea is applicable. The operating system maintains a list of all pages currently in memory, with the most recent arrival at the tail and the least recent arrival at the head. On a page fault, the page at the head is removed and the new page added to the tail of the list. When applied to stores, FIFO might remove mustache wax, but it might also remove flour, salt, or butter. When applied to computers the same problem arises: the oldest page may still be useful. For this reason, FIFO in its pure form is rarely used.

3.4.4 The Second-Chance Page Replacement Algorithm

A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the *R* bit of the oldest page. If it is 0, the page is both old and unused, so it is replaced immediately. If the *R* bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.

The operation of this algorithm, called **second chance**, is shown in Fig. 3-15. In Fig. 3-15(a) we see pages *A* through *H* kept on a linked list and sorted by the time they arrived in memory.

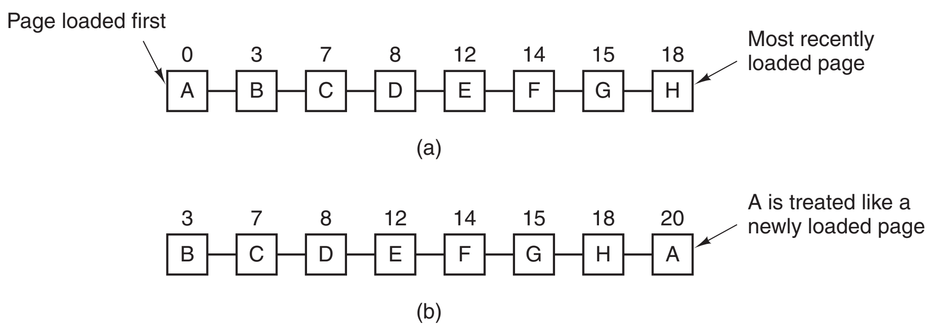


Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their load times.

Suppose that a page fault occurs at time 20. The oldest page is *A*, which arrived at time 0, when the process started. If *A* has the *R* bit cleared, it is evicted from memory, either by being written to the disk (if it is dirty), or just abandoned (if it is clean). On the other hand, if the *R* bit is set, *A* is put onto the end of the list and its "load time" is reset to the current time (20). The *R* bit is also cleared. The search for a suitable page continues with *B*.

What second chance is looking for is an old page that has not been referenced in the most recent clock interval. If all the pages have been referenced, second chance degenerates into pure FIFO. Specifically, imagine that all the pages in Fig. 3-15(a) have their *R* bits set. One by one, the operating system moves the pages to the end of the list, clearing the *R* bit each time it appends a page to the end of the list. Eventually, it comes back to page *A*, which now has its *R* bit cleared. At this point *A* is evicted. Thus the algorithm always terminates.

3.4.5 The Clock Page Replacement Algorithm

Although second chance is a reasonable algorithm, it is unnecessarily inefficient because it is constantly moving pages around on its list. A better approach is to keep all the page frames on a circular list in the form of a clock, as shown in Fig. 3-16. The hand points to the oldest page.

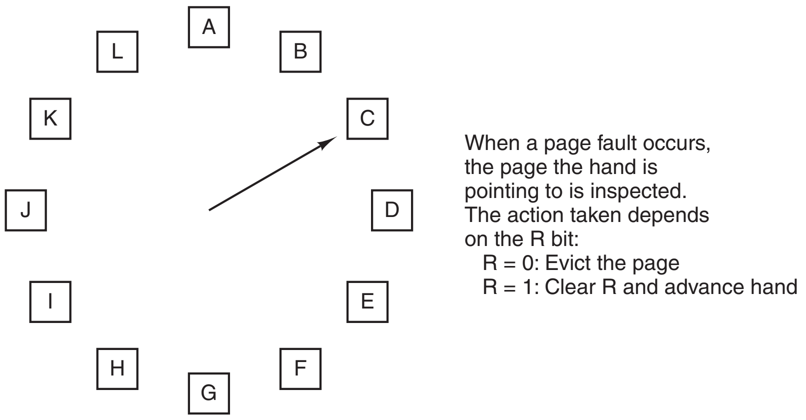

A
B
C
D
E
F
G
H
I
J
K
L
When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
R = 0: Evict the page
R = 1: Clear R and advance hand

Figure 3-16. The clock page replacement algorithm.

When a page fault occurs, the page being pointed to by the hand is inspected. If its *R* bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position. If *R* is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with $R = 0$. Not surprisingly, this algorithm is called **clock**.

3.4.6 The Least Recently Used (LRU) Page Replacement Algorithm

A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again soon. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called **LRU** (**Least Recently Used**) paging.

Although LRU is theoretically realizable, it is not cheap by a long shot. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation, even in hardware (assuming that such hardware could be built).

However, there are other ways to implement LRU with special hardware. Let us consider the simplest way first. This method requires equipping the hardware with a 64-bit counter, *C*, that is automatically incremented after each instruction. Furthermore, each page table entry must also have a field large enough to contain the counter. After each memory reference, the current value of *C* is stored in the page table entry for the page just referenced. When a page fault occurs, the operating system examines all the counters in the page table to find the lowest one. That page is the least recently used.

3.4.7 Simulating LRU in Software

Although the previous LRU algorithm is (in principle) realizable, few, if any, machines have the required hardware. Instead, a solution that can be implemented in software is needed. One possibility is called the **NFU** (**Not Frequently Used**) algorithm. It requires a software counter associated with each page, initially zero. At each clock interrupt, the operating system scans all the pages in memory. For each page, the *R* bit, which is 0 or 1, is added to the counter. The counters roughly keep track of how often each page has been referenced. When a page fault occurs, the page with the lowest counter is chosen for replacement.

The main problem with NFU is that it is like an elephant: it never forgets anything. For example, in a multipass compiler, pages that were heavily used during pass 1 may still have a high count well into later passes. In fact, if pass 1 happens to have the longest execution time of all the passes, the pages containing the code for subsequent passes may always have lower counts than the pass-1 pages. Consequently, the operating system will remove useful pages instead of pages no longer in use.

Fortunately, a small modification to NFU makes it able to simulate LRU quite well. The modification has two parts. First, the counters are each shifted right 1 bit before the *R* bit is added in. Second, the *R* bit is added to the leftmost rather than the rightmost bit.

Figure 3-17 illustrates how the modified algorithm, known as **aging**, works. Suppose that after the first clock tick the *R* bits for pages 0 to 5 have the values 1, 0, 1, 0, 1, and 1, respectively (page 0 is 1, page 1 is 0, page 2 is 1, etc.). In other words, between tick 0 and tick 1, pages 0, 2, 4, and 5 were referenced, setting their *R* bits to 1, while the other ones remained 0. After the six corresponding counters have been shifted and the *R* bit inserted at the left, they have the values shown in Fig. 3-17(a). The four remaining columns show the six counters after the next four clock ticks.

When a page fault occurs, the page whose counter is the lowest is removed. It is clear that a page that has not been referenced for, say, four clock ticks will have four leading zeros in its counter and thus will have a lower value than a counter that has not been referenced for three clock ticks.

This algorithm differs from LRU in two important ways. Consider pages 3 and 5 in Fig. 3-17(e). Neither has been referenced for two clock ticks; both were referenced in the tick prior to that. According to LRU, if a page must be replaced, we should choose one of these two. The trouble is, we do not know which of them was referenced last in the interval between tick 1 and tick 2. By recording only 1 bit per time interval, we have now lost the ability to distinguish references early in the

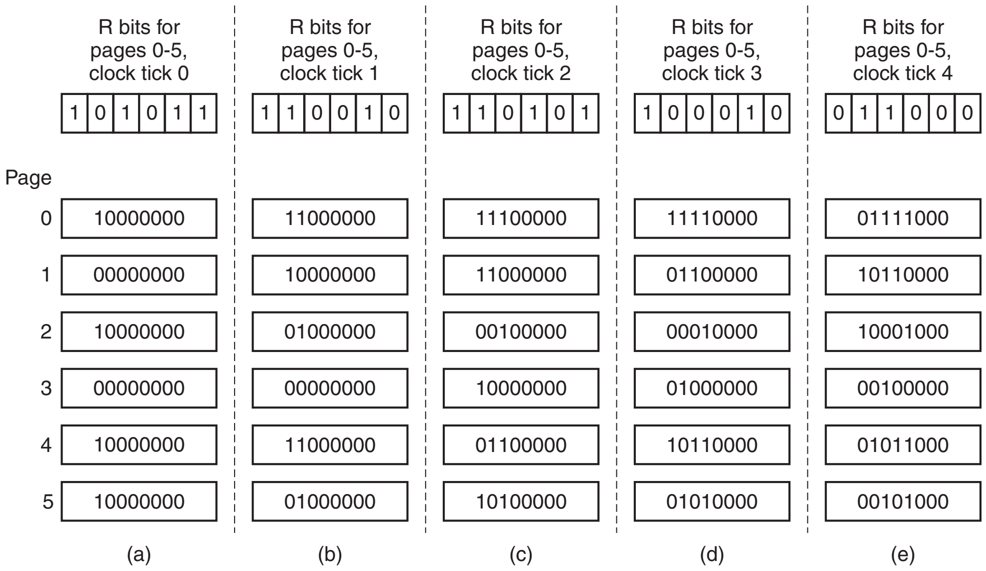


Figure 3-17. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

clock interval from those occurring later. All we can do is remove page 3, because page 5 was also referenced two ticks earlier and page 3 was not.

The second difference between LRU and aging is that in aging the counters have a finite number of bits (8 bits in this example), which limits its past horizon. Suppose that two pages each have a counter value of 0. All we can do is pick one of them at random. In reality, it may well be that one of the pages was last referenced nine ticks ago and the other was last referenced 1000 ticks ago. We have no way of seeing that. In practice, however, 8 bits is generally enough if a clock tick is around 20 msec. If a page has not been referenced in 160 msec, it probably is not that important.

3.4.8 The Working Set Page Replacement Algorithm

In the purest form of paging, processes are started up with none of their pages in memory. As soon as the CPU tries to fetch the first instruction, it gets a page fault, causing the operating system to bring in the page containing the first instruction. Other page faults for global variables and the stack usually follow quickly. After a while, the process has most of the pages it needs and settles down to run with relatively few page faults. This strategy is called **demand paging** because pages are loaded only on demand, not in advance.

Of course, it is easy enough to write a test program that systematically reads all the pages in a large address space, causing so many page faults that there is not enough memory to hold them all. Fortunately, most processes do not work this way. They exhibit a **locality of reference**, meaning that during any phase of execution, the process references only a relatively small fraction of its pages. Each pass of a multipass compiler, for example, references only a fraction of all the pages, and a different fraction at that.

The set of pages that a process is currently using is its **working set** (Denning, 1968a; Denning, 1980). If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase (e.g., the next pass of the compiler). If the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly, since executing an instruction takes a few nanoseconds and reading in a page from the disk typically takes 10 msec At a rate of one or two instructions per 10 msec, it will take ages to finish. A program causing page faults every few instructions is said to be **thrashing** (Denning, 1968b).

In a multiprogramming system, processes are often moved to disk (i.e., all their pages are removed from memory) to let others have a turn at the CPU. The question arises of what to do when a process is brought back in again. Technically, nothing need be done. The process will just cause page faults until its working set has been loaded. The problem is that having numerous page faults every time a process is loaded is slow, and it also wastes considerable CPU time, since it takes the operating system a few milliseconds of CPU time to process a page fault.

Therefore, many paging systems try to keep track of each process' working set and make sure that it is in memory before letting the process run. This approach is called the **working set model** (Denning, 1970). It is designed to greatly reduce the page fault rate. Loading the pages *before* letting processes run is also called **prepaging.** Note that the working set changes over time.

It has long been known that programs rarely reference their address space uniformly, but that the references tend to cluster on a small number of pages. A memory reference may fetch an instruction or data, or it may store data. At any instant of time, t, there exists a set consisting of all the pages used by the k most recent memory references. This set, $w(k, t)$, is the working set. Because the $k = 1$ most recent references must have used all the pages used by the $k > 1$ most recent references, and possibly others, $w(k, t)$ is a monotonically nondecreasing function of k. The limit of $w(k, t)$ as k becomes large is finite because a program cannot reference more pages than its address space contains, and few programs will use every single page. Figure 3-18 depicts the size of the working set as a function of k.

The fact that most programs randomly access a small number of pages, but that this set changes slowly in time explains the initial rapid rise of the curve and then the much slower rise for large k. For example, a program that is executing a loop occupying two pages using data on four pages may reference all six pages every 1000 instructions, but the most recent reference to some other page may be a million instructions earlier, during the initialization phase. Due to this asymptotic behavior, the contents of the working set is not sensitive to the value of k chosen. To

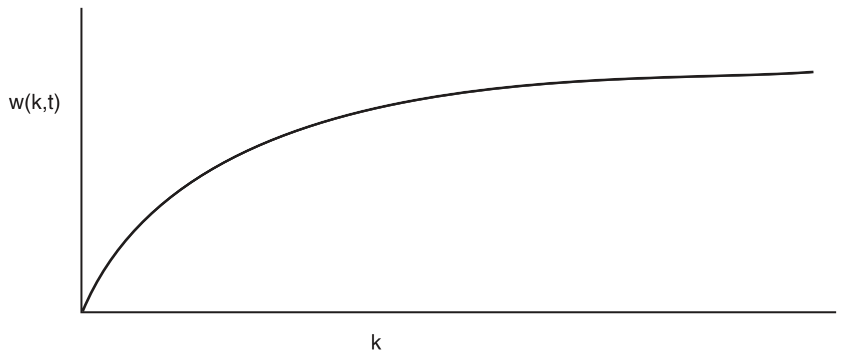


Figure 3-18. The working set is the set of pages used by the *k* most recent memory references. The function $w(k, t)$ is the size of the working set at time *t*.

put it differently, there exists a wide range of *k* values for which the working set is unchanged. Because the working set varies slowly with time, it is possible to make a reasonable guess as to which pages will be needed when the program is restarted on the basis of its working set when it was last stopped. Prepaging consists of loading these pages before resuming the process.

To implement the working set model, it is necessary for the operating system to keep track of which pages are in the working set. Having this information also immediately leads to a possible page replacement algorithm: when a page fault occurs, find a page not in the working set and evict it. To implement such an algorithm, we need a precise way of determining which pages are in the working set. By definition, the working set is the set of pages used in the *k* most recent memory references (some authors use the *k* most recent page references, but the choice is arbitrary). To implement any working set algorithm, some value of *k* must be chosen in advance. Then, after every memory reference, the set of pages used by the most recent *k* memory references is uniquely determined.

Of course, having an operational definition of the working set does not mean that there is an efficient way to compute it during program execution. One could imagine a shift register of length *k*, with every memory reference shifting the register left one position and inserting the most recently referenced page number on the right. The set of all *k* page numbers in the shift register would be the working set. In theory, at a page fault, the contents of the shift register could be read out and sorted. Duplicate pages could then be removed. The result would be the working set. However, maintaining the shift register and processing it at a page fault would both be prohibitively expensive, so this technique is never used.

Instead, various approximations are used. One commonly used approximation is to drop the idea of counting back *k* memory references and use execution time instead. For example, instead of defining the working set as those pages used during the previous 10 million memory references, we can define it as the set of pages used during the past 100 msec of execution time. In practice, such a definition is just as good and much easier to work with. Note that for each process, only its own execution time counts. Thus if a process starts running at time T and has had 40 msec of CPU time at real time $T + 100$ msec, for working set purposes its time is 40 msec. The amount of CPU time a process has actually used since it started is often called its **current virtual time**. With this approximation, the working set of a process is the set of pages it has referenced during the past τ seconds of virtual time.

Now let us look at a page replacement algorithm based on the working set. The basic idea is to find a page that is not in the working set and evict it. In Fig. 3-19 we see a portion of a page table for some machine. Because only pages located in memory are considered as candidates for eviction, pages that are absent from memory are ignored by this algorithm. Each entry contains (at least) two key items of information: the (approximate) time the page was last used and the R (Referenced) bit. An empty white rectangle symbolizes the other fields not needed for this algorithm, such as the page frame number, the protection bits, and the M (Modified) bit.

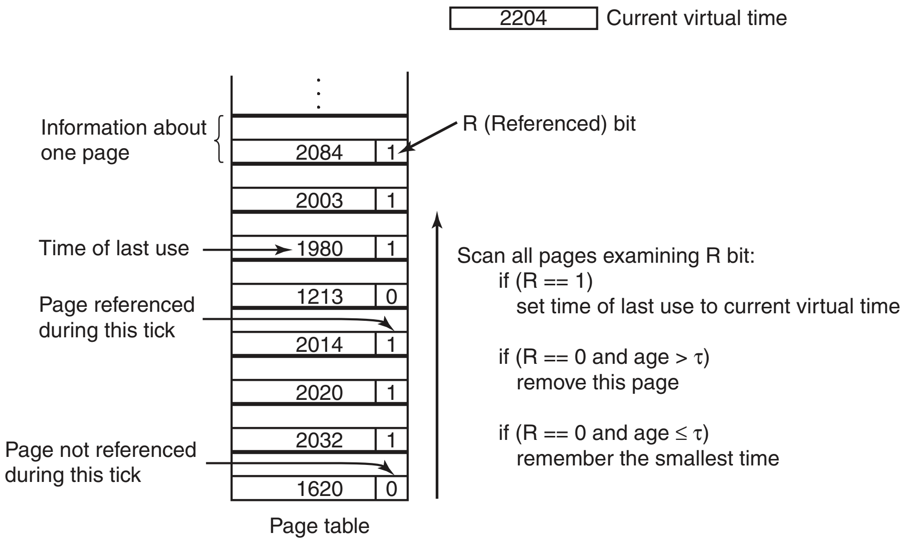

2204
Current virtual time
Information about one page
R (Referenced) bit
2084 1
2003 1
Time of last use
1980 1
1213 0
Page referenced during this tick
2014 1
2020 1
2032 1
Page not referenced during this tick
1620 0
Page table
Scan all pages examining R bit:
if (R == 1)
set time of last use to current virtual time
if (R == 0 and age > τ)
remove this page
if (R == 0 and age ≤ τ)
remember the smallest time

Figure 3-19. The working set algorithm.

The algorithm works as follows. The hardware is assumed to set the R and M bits, as discussed earlier. Similarly, a periodic clock interrupt is assumed to cause software to run that clears the *Referenced* bit on every clock tick. On every page fault, the page table is scanned to look for a suitable page to evict.

As each entry is processed, the R bit is examined. If it is 1, the current virtual time is written into the *Time of last use* field in the page table, indicating that the page was in use at the time the fault occurred. Since the page has been referenced during the current clock tick, it is clearly in the working set and is not a candidate for removal (τ is assumed to span multiple clock ticks).

If R is 0, the page has not been referenced during the current clock tick and may be a candidate for removal. To see whether or not it should be removed, its age (the current virtual time minus its *Time of last use*) is computed and compared to τ. If the age is greater than τ, the page is no longer in the working set and the new page replaces it. The scan continues updating the remaining entries.

However, if R is 0 but the age is less than or equal to τ, the page is still in the working set. The page is temporarily spared, but the page with the greatest age (smallest value of *Time of last use*) is noted. If the entire table is scanned without finding a candidate to evict, that means that all pages are in the working set. In that case, if one or more pages with $R = 0$ were found, the one with the greatest age is evicted. In the worst case, all pages have been referenced during the current clock tick (and thus all have $R = 1$), so one is chosen at random for removal, preferably a clean page, if one exists.

3.4.9 The WSClock Page Replacement Algorithm

The basic working set algorithm is cumbersome, since the entire page table has to be scanned at each page fault until a suitable candidate is located. An improved algorithm, which is based on the clock algorithm but also uses the working set information, is called **WSClock** (Carr and Hennessey, 1981). Due to its simplicity of implementation and good performance, it is widely used in practice.

The data structure needed is a circular list of page frames, as in the clock algorithm, and as shown in Fig. 3-20(a). Initially, this list is empty. When the first page is loaded, it is added to the list. As more pages are added, they go into the list to form a ring. Each entry contains the *Time of last use* field from the basic working set algorithm, as well as the R bit (shown) and the M bit (not shown).

As with the clock algorithm, at each page fault the page pointed to by the hand is examined first. If the R bit is set to 1, the page has been used during the current tick so it is not an ideal candidate to remove. The R bit is then set to 0, the hand advanced to the next page, and the algorithm repeated for that page. The state after this sequence of events is shown in Fig. 3-20(b).

Now consider what happens if the page pointed to has $R = 0$, as shown in Fig. 3-20(c). If the age is greater than τ and the page is clean, it is not in the working set and a valid copy exists on the disk. The page frame is simply claimed and the new page put there, as shown in Fig. 3-20(d). On the other hand, if the page is dirty, it cannot be claimed immediately since no valid copy is present on disk. To avoid a process switch, the write to disk is scheduled, but the hand is advanced and the algorithm continues with the next page. After all, there might be an old, clean page further down the line that can be used immediately.

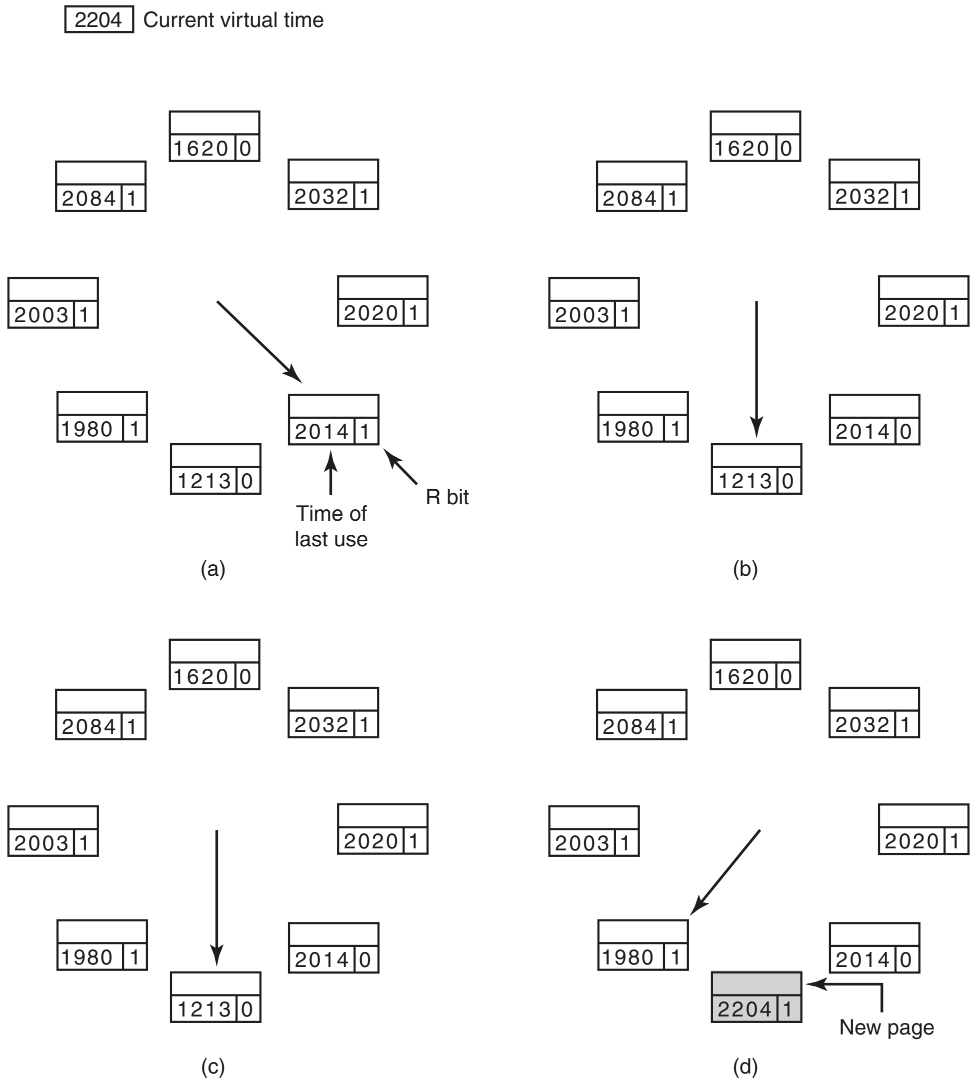


Figure 3-20. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when $R = 1$. (c) and (d) give an example of $R = 0$.

In principle, all pages might be scheduled for disk I/O on one cycle around the clock. To reduce disk traffic, a limit might be set, allowing a maximum of *n* pages to be written back. Once this limit has been reached, no new writes would be scheduled.

What happens if the hand comes all the way around and back to its starting point? There are two cases we have to consider:

1. At least one write has been scheduled.
2. No writes have been scheduled.

In the first case, the hand just keeps moving, looking for a clean page. Since one or more writes have been scheduled, eventually some write will complete and its page will be marked as clean. The first clean page encountered is evicted. This page is not necessarily the first write scheduled because the disk driver may reorder writes in order to optimize disk performance.

In the second case, all pages are in the working set, otherwise at least one write would have been scheduled. Lacking additional information, the simplest thing to do is claim any clean page and use it. The location of a clean page could be kept track of during the sweep. If no clean pages exist, then the current page is chosen as the victim and written back to disk.

3.4.10 Summary of Page Replacement Algorithms

We have now looked at a variety of page replacement algorithms. Now we will briefly summarize them. The list of algorithms discussed is given in Fig. 3-21.

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Figure 3-21. Page replacement algorithms discussed in the text.

The optimal algorithm evicts the page that will be referenced furthest in the future. Unfortunately, there is no way to determine which page this is, so in practice this algorithm cannot be used. It is useful as a benchmark against which other algorithms can be measured, however.

The NRU algorithm divides pages into four classes depending on the state of the *R* and *M* bits. A random page from the lowest-numbered class is chosen. This algorithm is easy to implement, but it is very crude. Better ones exist.

FIFO keeps track of the order in which pages were loaded into memory by keeping them in a linked list. Removing the oldest page then becomes trivial, but that page might still be in use, so FIFO is a bad choice.

Second chance is a modification to FIFO that checks if a page is in use before removing it. If it is, the page is spared. This modification greatly improves the performance. Clock is simply a different implementation of second chance. It has the same performance properties, but takes a little less time to execute the algorithm.

LRU is an excellent algorithm, but it cannot be implemented without special hardware. If this hardware is not available, it cannot be used. NFU is a crude attempt to approximate LRU. It is not very good. However, aging is a much better approximation to LRU and can be implemented efficiently. It is a good choice.

The last two algorithms use the working set. The working set algorithm gives reasonable performance, but it is somewhat expensive to implement. WSClock is a variant that not only gives good performance but is also efficient to implement.

All in all, the two best algorithms are aging and WSClock. They are based on LRU and the working set, respectively. Both give good paging performance and can be implemented efficiently. A few other good algorithms exist, but these two are probably the most important in practice.

3.5 DESIGN ISSUES FOR PAGING SYSTEMS

In the previous sections we have explained how paging works and have given a few of the basic page replacement algorithms. But knowing the bare mechanics is not enough. To design a system and make it work well you have to know a lot more. It is like the difference between knowing how to move the rook, knight, bishop, and other pieces in chess, and being a good player. In the following sections, we will look at other issues that operating system designers must consider carefully in order to get good performance from a paging system.

3.5.1 Local versus Global Allocation Policies

In the preceding sections we have discussed several algorithms for choosing a page to replace when a fault occurs. A major issue associated with this choice (which we have carefully swept under the rug until now) is how memory should be allocated among the competing runnable processes.

Take a look at Fig. 3-22(a). In this figure, three processes, *A*, *B*, and *C*, make up the set of runnable processes. Suppose *A* gets a page fault. Should the page replacement algorithm try to find the least recently used page considering only the six pages currently allocated to *A*, or should it consider all the pages in memory? If it looks only at *A*'s pages, the page with the lowest age value is *A5*, so we get the situation of Fig. 3-22(b).

On the other hand, if the page with the lowest age value is removed without regard to whose page it is, page *B3* will be chosen and we will get the situation of Fig. 3-22(c). The algorithm of Fig. 3-22(b) is said to be a **local** page replacement

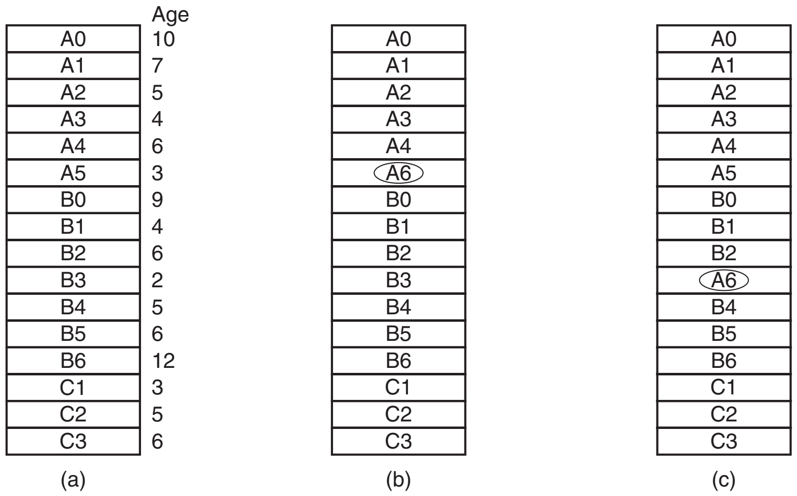


Figure 3-22. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

algorithm, whereas that of Fig. 3-22(c) is said to be a **global** algorithm. Local algorithms effectively correspond to allocating every process a fixed fraction of the memory. Global algorithms dynamically allocate page frames among the runnable processes. Thus the number of page frames assigned to each process varies in time.

In general, global algorithms work better, especially when the working set size can vary a lot over the lifetime of a process. If a local algorithm is used and the working set grows, thrashing will result, even if there are a sufficient number of free page frames. If the working set shrinks, local algorithms waste memory. If a global algorithm is used, the system must continually decide how many page frames to assign to each process. One way is to monitor the working set size as indicated by the aging bits, but this approach does not necessarily prevent thrashing. The working set may change size in milliseconds, whereas the aging bits are a very crude measure spread over a number of clock ticks.

Another approach is to have an algorithm for allocating page frames to processes. One way is to periodically determine the number of running processes and allocate each process an equal share. Thus with 12,416 available (i.e., nonoperating system) page frames and 10 processes, each process gets 1241 frames. The remaining six go into a pool to be used when page faults occur.

Although this method may seem fair, it makes little sense to give equal shares of the memory to a 10-KB process and a 300-KB process. Instead, pages can be allocated in proportion to each process' total size, with a 300-KB process getting 30 times the allotment of a 10-KB process. It is probably wise to give each process some minimum number, so that it can run no matter how small it is. On some machines, for example, a single two-operand instruction may need as many as six pages because the instruction itself, the source operand, and the destination operand may all straddle page boundaries. With an allocation of only five pages, programs containing such instructions cannot execute at all.

If a global algorithm is used, it may be possible to start each process up with some number of pages proportional to the process' size, but the allocation has to be updated dynamically as the processes run. One way to manage the allocation is to use the **PFF** (**Page Fault Frequency**) algorithm. It tells when to increase or decrease a process' page allocation but says nothing about which page to replace on a fault. It just controls the size of the allocation set.

For a large class of page replacement algorithms, including LRU, it is known that the fault rate decreases as more pages are assigned, as we discussed above. This is the assumption behind PFF. This property is illustrated in Fig. 3-23.

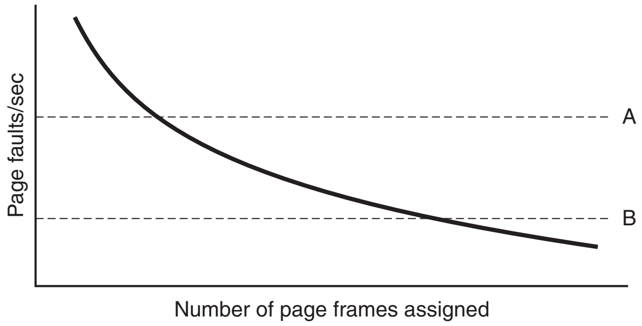


Figure 3-23. Page fault rate as a function of the number of page frames assigned.

Measuring the page fault rate is straightforward: just count the number of faults per second, possibly taking a running mean over past seconds as well. One easy way to do this is to add the number of page faults during the immediately preceding second to the current running mean and divide by two. The dashed line marked *A* corresponds to a page fault rate that is unacceptably high, so the faulting process is given more page frames to reduce the fault rate. The dashed line marked *B* corresponds to a page fault rate so low that we can assume the process has too much memory. In this case, page frames may be taken away from it. Thus, PFF tries to keep the paging rate for each process within acceptable bounds.

It is important to note that some page replacement algorithms can work with either a local replacement policy or a global one. For example, FIFO can replace the oldest page in all of memory (global algorithm) or the oldest page owned by the current process (local algorithm). Similarly, LRU or some approximation to it can replace the least recently used page in all of memory (global algorithm) or the least recently used page owned by the current process (local algorithm). The choice of local versus global is independent of the algorithm in some cases.

On the other hand, for other page replacement algorithms, only a local strategy makes sense. In particular, the working set and WSClock algorithms refer to some specific process and must be applied in that context. There really is no working set for the machine as a whole, and trying to use the union of all the working sets would lose the locality property and not work well.

3.5.2 Load Control

Even with the best page replacement algorithm and optimal global allocation of page frames to processes, it can happen that the system thrashes. In fact, whenever the combined working sets of all processes exceed the capacity of memory, thrashing can be expected. One symptom of this situation is that the PFF algorithm indicates that some processes need more memory but no processes need less memory. In this case, there is no way to give more memory to those processes needing it without hurting some other processes. The only real solution is to temporarily get rid of some processes.

A good way to reduce the number of processes competing for memory is to swap some of them to the disk and free up all the pages they are holding. For example, one process can be swapped to disk and its page frames divided up among other processes that are thrashing. If the thrashing stops, the system can run for a while this way. If it does not stop, another process has to be swapped out, and so on, until the thrashing stops. Thus even with paging, swapping may still be needed, only now swapping is used to reduce potential demand for memory, rather than to reclaim pages.

Swapping processes out to relieve the load on memory is reminiscent of two-level scheduling, in which some processes are put on disk and a short-term scheduler is used to schedule the remaining processes. Clearly, the two ideas can be combined, with just enough processes swapped out to make the page-fault rate acceptable. Periodically, some processes are brought in from disk and other ones are swapped out.

However, another factor to consider is the degree of multiprogramming. When the number of processes in main memory is too low, the CPU may be idle for substantial periods of time. This consideration argues for considering not only process size and paging rate when deciding which process to swap out, but also its characteristics, such as whether it is CPU bound or I/O bound, and what characteristics the remaining processes have.

3.5.3 Page Size

The page size is a parameter that can be chosen by the operating system. Even if the hardware has been designed with, for example, 4096-byte pages, the operating system can easily regard page pairs 0 and 1, 2 and 3, 4 and 5, and so on, as 8-KB pages by always allocating two consecutive 8192-byte page frames for them.

Determining the best page size requires balancing several competing factors. As a result, there is no overall optimum. To start with, two factors argue for a small page size. A randomly chosen text, data, or stack segment will not fill an integral number of pages. On the average, half of the final page will be empty. The extra space in that page is wasted. This wastage is called **internal fragmentation**. With n segments in memory and a page size of p bytes, $np/2$ bytes will be wasted on internal fragmentation. This reasoning argues for a small page size.

Another argument for a small page size becomes apparent if we think about a program consisting of eight sequential phases of 4 KB each. With a 32-KB page size, the program must be allocated 32 KB all the time. With a 16-KB page size, it needs only 16 KB. With a page size of 4 KB or smaller, it requires only 4 KB at any instant. In general, a large page size will cause more wasted space to be in memory than a small page size.

On the other hand, small pages mean that programs will need many pages, and thus a large page table. A 32-KB program needs only four 8-KB pages, but 64 512-byte pages. Transfers to and from the disk are generally a page at a time, with most of the time being for the seek and rotational delay, so that transferring a small page takes almost as much time as transferring a large page. It might take 64×10 msec to load 64 512-byte pages, but only 4×12 msec to load four 8-KB pages.

Also, small pages use up much valuable space in the **TLB**. Say your program uses 1 MB of memory with a working set of 64 KB. With 4-KB pages, the program would occupy at least 16 entries in the TLB. With 2-MB pages, a single TLB entry would be sufficient (in theory, it may be that you want to separate data and instructions). As TLB entries are scarce, and critical for performance, it pays to use large pages wherever possible. To balance all these trade-offs, operating systems sometimes use different page sizes for different parts of the system. For instance, large pages for the kernel and smaller ones for user processes.

On some machines, the page table must be loaded (by the operating system) into hardware registers every time the CPU switches from one process to another. On these machines, having a small page size means that the time required to load the page registers gets longer as the page size gets smaller. Furthermore, the space occupied by the page table increases as the page size decreases.

This last point can be analyzed mathematically. Let the average process size be s bytes and the page size be p bytes. Furthermore, assume that each page entry requires e bytes. The approximate number of pages needed per process is then s/p, occupying se/p bytes of page table space. The wasted memory in the last page of the process due to internal fragmentation is $p/2$. Thus, the total overhead due to the page table and the internal fragmentation loss is given by the sum of these two terms:

$$\text{overhead} = se/p + p/2$$

The first term (page table size) is large when the page size is small. The second term (internal fragmentation) is large when the page size is large. The optimum must lie somewhere in between. By taking the first derivative with respect to p and equating it to zero, we get the equation

$$-se/p^2 + 1/2 = 0$$

From this equation we can derive a formula that gives the optimum page size (considering only memory wasted in fragmentation and page table size). The result is:

$$p = \sqrt{2se}$$

For s = 1MB and e = 8 bytes per page table entry, the optimum page size is 4 KB. Commercially available computers have used page sizes ranging from 512 bytes to 64 KB. A typical value used to be 1 KB, but nowadays 4 KB is more common.

3.5.4 Separate Instruction and Data Spaces

Most computers have a single address space that holds both programs and data, as shown in Fig. 3-24(a). If this address space is large enough, everything works fine. However, if it's too small, it forces programmers to stand on their heads to fit everything into the address space.

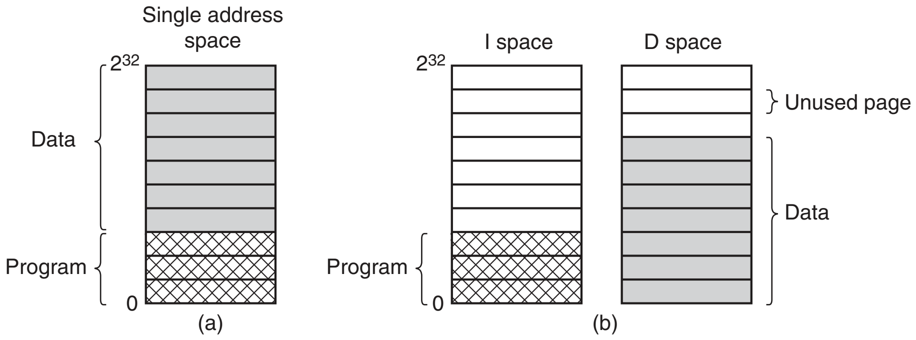


Figure 3-24. (a) One address space. (b) Separate I and D spaces.

One solution, pioneered on the (16-bit) PDP-11, is to have separate address spaces for instructions (program text) and data, called **I-space** and **D-space**, respectively, as illustrated in Fig. 3-24(b). Each address space runs from 0 to some maximum, typically $2^{16} - 1$ or $2^{32} - 1$. The linker must know when separate I- and D-spaces are being used, because when they are, the data are relocated to virtual address 0 instead of starting after the program.

In a computer with this kind of design, both address spaces can be paged, independently from one another. Each one has its own page table, with its own mapping of virtual pages to physical page frames. When the hardware wants to fetch an instruction, it knows that it must use I-space and the I-space page table. Similarly, data must go through the D-space page table. Other than this distinction, having separate I- and D-spaces does not introduce any special complications for the operating system and it does double the available address space.

While address spaces these days are large, their sizes used to be a serious problem. Even today, though, separate I- and D-spaces are still common. However, rather than for the normal address spaces, they are now used to divide the L1 cache. After all, in the L1 cache, memory is still plenty scarce.

3.5.5 Shared Pages

Another design issue is sharing. In a large multiprogramming system, it is common for several users to be running the same program at the same time. Even a single user may be running several programs that use the same library. It is clearly more efficient to share the pages, to avoid having two copies of the same page in memory at the same time. One problem is that not all pages are sharable. In particular, pages that are read-only, such as program text, can be shared, but for data pages sharing is more complicated.

If separate I- and D-spaces are supported, it is relatively straightforward to share programs by having two or more processes use the same page table for their I-space but different page tables for their D-spaces. Typically in an implementation that supports sharing in this way, page tables are data structures independent of the process table. Each process then has two pointers in its process table: one to the I-space page table and one to the D-space page table, as shown in Fig. 3-25. When the scheduler chooses a process to run, it uses these pointers to locate the appropriate page tables and sets up the MMU using them. Even without separate I- and D-spaces, processes can share programs (or sometimes, libraries), but the mechanism is more complicated.

When two or more processes share some code, a problem occurs with the shared pages. Suppose that processes *A* and *B* are both running the editor and sharing its pages. If the scheduler decides to remove *A* from memory, evicting all its pages and filling the empty page frames with some other program will cause *B* to generate a large number of page faults to bring them back in again.

Similarly, when *A* terminates, it is essential to be able to discover that the pages are still in use so that their disk space will not be freed by accident. Searching all the page tables to see if a page is shared is usually too expensive, so special data structures are needed to keep track of shared pages, especially if the unit of sharing is the individual page (or run of pages), rather than an entire page table.

Sharing data is trickier than sharing code, but it is not impossible. In particular, in UNIX, after a fork system call, the parent and child are required to share both program text and data. In a paged system, what is often done is to give each of these processes its own page table and have both of them point to the same set of pages. Thus no copying of pages is done at fork time. However, all the data pages are mapped into both processes as READ ONLY.

As long as both processes just read their data, without modifying it, this situation can continue. As soon as either process updates a memory word, the violation of the read-only protection causes a trap to the operating system. A copy is then

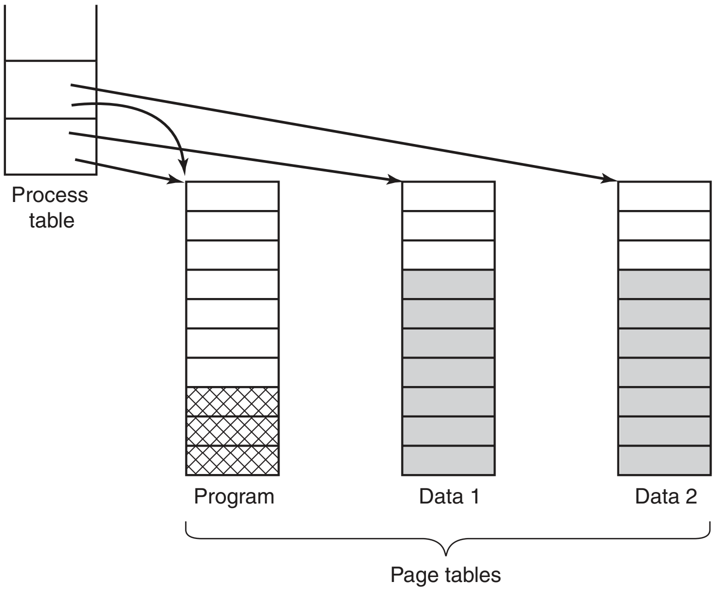


Figure 3-25. Two processes sharing the same program sharing its page tables.

made of the offending page so that each process now has its own private copy. Both copies are now set to READ/WRITE, so subsequent writes to either copy proceed without trapping. This strategy means that those pages that are never modified (including all the program pages) need not be copied. Only the data pages that are actually modified need to be copied. This approach, called **copy on write**, improves performance by reducing copying.

3.5.6 Shared Libraries

Sharing can be done at other granularities than individual pages. If a program is started up twice, most operating systems will automatically share all the text pages so that only one copy is in memory. Text pages are always read only, so there is no problem here. Depending on the operating system, each process may get its own private copy of the data pages, or they may be shared and marked read only. If any process modifies a data page, a private copy will be made for it, that is, copy on write will be applied.

In modern systems, there are many large libraries used by many processes, for example, multiple I/O and graphics libraries. Statically binding all these libraries to every executable program on the disk would make them even more bloated than they already are.

Instead, a common technique is to use **shared libraries** (which are called **DLLs** or **Dynamic Link Libraries** on Windows). To make the idea of a shared library clear, first consider traditional linking. When a program is linked, one or more object files and possibly some libraries are named in the command to the linker, such as the UNIX command

```
ld *.o –lc –lm
```

which links all the *.o* (object) files in the current directory and then scans two libraries, */usr/lib/libc.a* and */usr/lib/libm.a*. Any functions called in the object files but not present there (e.g., *printf*) are called **undefined externals** and are sought in the libraries. If they are found, they are included in the executable binary. Any functions that they call but are not yet present also become undefined externals. For example, *printf* needs *write*, so if *write* is not already included, the linker will look for it and include it when found. When the linker is done, an executable binary file is written to the disk containing all the functions needed. Functions present in the libraries but not called are not included. When the program is loaded into memory and executed, all the functions it needs are there.

Now suppose common programs use 20–50 MB worth of graphics and user interface functions. Statically linking hundreds of programs with all these libraries would waste a tremendous amount of space on the disk as well as wasting space in RAM when they were loaded since the system would have no way of knowing it could share them. This is where shared libraries come in. When a program is linked with shared libraries (which are slightly different than static ones), instead of including the actual function called, the linker includes a small stub routine that binds to the called function at run time. Depending on the system and the configuration details, shared libraries are loaded either when the program is loaded or when functions in them are called for the first time. Of course, if another program has already loaded the shared library, there is no need to load it again—that is the whole point of it. Note that when a shared library is loaded or used, the entire library is not read into memory in a single blow. It is paged in, page by page, as needed, so functions that are not called will not be brought into RAM.

In addition to making executable files smaller and also saving space in memory, shared libraries have another important advantage: if a function in a shared library is updated to remove a bug, it is not necessary to recompile the programs that call it. The old binaries continue to work. This feature is especially important for commercial software, where the source code is not distributed to the customer. For example, if Microsoft finds and fixes a security error in some standard DLL, *Windows Update* will download the new DLL and replace the old one, and all programs that use the DLL will automatically use the new version the next time they are launched.

Shared libraries come with one little problem, however, that has to be solved, however. The problem is illustrated in Fig. 3-26. Here we see two processes sharing a library of size 20 KB (assuming each box is 4 KB). However, the library is located at a different address in each process, presumably because the programs themselves are not the same size. In process 1, the library starts at address 36K; in process 2 it starts at 12K. Suppose that the first thing the first function in the library has to do is jump to address 16 in the library. If the library were not shared, it could be relocated on the fly as it was loaded so that the jump (in process 1) could be to virtual address 36K + 16. Note that the physical address in the RAM where the library is located does not matter since all the pages are mapped from virtual to physical addresses by the MMU hardware.

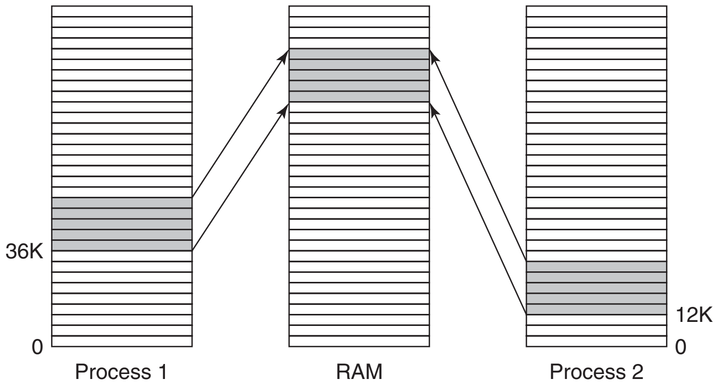


Figure 3-26. A shared library being used by two processes.

However, since the library is shared, relocation on the fly will not work. After all, when the first function is called by process 2 (at address 12K), the jump instruction has to go to 12K + 16, not 36K + 16. This is the little problem. One way to solve it is to use copy on write and create new pages for each process sharing the library, relocating them on the fly as they are created, but this scheme defeats the purpose of sharing the library, of course.

A better solution is to compile shared libraries with a special compiler flag telling the compiler not to produce any instructions that use absolute addresses. Instead only instructions using relative addresses are used. For example, there is almost always an instruction that says jump forward (or backward) by *n* bytes (as opposed to an instruction that gives a specific address to jump to). This instruction works correctly no matter where the shared library is placed in the virtual address space. By avoiding absolute addresses, the problem can be solved. Code that uses only relative offsets is called **position-independent code**.

3.5.7 Mapped Files

Shared libraries are really a special case of a more general facility called **memory-mapped files**. The idea here is that a process can issue a system call to map a file onto a portion of its virtual address space. In most implementations, no pages are brought in at the time of the mapping, but as pages are touched, they are demand paged in one page at a time, using the disk file as the backing store. When the process exits, or explicitly unmaps the file, all the modified pages are written back to the file on disk.

Mapped files provide an alternative model for I/O. Instead, of doing reads and writes, the file can be accessed as a big character array in memory. In some situations, programmers find this model more convenient.

If two or more processes map onto the same file at the same time, they can communicate over shared memory. Writes done by one process to the shared memory are immediately visible when the other one reads from the part of its virtual address spaced mapped onto the file. This mechanism thus provides a high-bandwidth channel between processes and is often used as such (even to the extent of mapping a scratch file). Now it should be clear that if memory-mapped files are available, shared libraries can use this mechanism.

3.5.8 Cleaning Policy

Paging works best when there is an abundant supply of free page frames that can be claimed as page faults occur. If every page frame is full, and furthermore modified, before a new page can be brought in, an old page must first be written to disk. To ensure a plentiful supply of free page frames, paging systems generally have a background process, called the **paging daemon**, that sleeps most of the time but is awakened periodically to inspect the state of memory. If too few page frames are free, it begins selecting pages to evict using some page replacement algorithm. If these pages have been modified since being loaded, they are written to disk.

In any event, the previous contents of the page are remembered. In the event one of the evicted pages is needed again before its frame has been overwritten, it can be reclaimed by removing it from the pool of free page frames. Keeping a supply of page frames around yields better performance than using all of memory and then trying to find a frame at the moment it is needed. At the very least, the paging daemon ensures that all the free frames are clean, so they need not be written to disk in a big hurry when they are required.

One way to implement this cleaning policy is with a two-handed clock. The front hand is controlled by the paging daemon. When it points to a dirty page, that page is written back to disk and the front hand is advanced. When it points to a clean page, it is just advanced. The back hand is used for page replacement, as in the standard clock algorithm. Only now, the probability of the back hand hitting a clean page is increased due to the work of the paging daemon.

3.5.9 Virtual Memory Interface

Up until now, our whole discussion has assumed that virtual memory is transparent to processes and programmers, that is, all they see is a large virtual address space on a computer with a small(er) physical memory. With many systems, that is true, but in some advanced systems, programmers have some control over the memory map and can use it in nontraditional ways to enhance program behavior. In this section, we will briefly look at a few of these.

One reason for giving programmers control over their memory map is to allow two or more processes to share the same memory. sometimes in sophisticated ways. If programmers can name regions of their memory, it may be possible for one process to give another process the name of a memory region so that process can also map it in. With two (or more) processes sharing the same pages, high bandwidth sharing becomes possible—one process writes into the shared memory and another one reads from it. A sophisticated example of such a communication channel is described by De Bruijn (2011).

Sharing of pages can also be used to implement a high-performance message-passing system. Normally, when messages are passed, the data are copied from one address space to another, at considerable cost. If processes can control their page map, a message can be passed by having the sending process unmap the page(s) containing the message, and the receiving process mapping them in. Here only the page names have to be copied, instead of all the data.

Yet another advanced memory management technique is **distributed shared memory** (Feeley et al., 1995; Li, 1986; Li and Hudak, 1989; and Zekauskas et al., 1994). The idea here is to allow multiple processes over a network to share a set of pages, possibly, but not necessarily, as a single shared linear address space. When a process references a page that is not currently mapped in, it gets a page fault. The page fault handler, which may be in the kernel or in user space, then locates the machine holding the page and sends it a message asking it to unmap the page and send it over the network. When the page arrives, it is mapped in and the faulting instruction is restarted. We will examine distributed shared memory in Chap. 8.

3.6 IMPLEMENTATION ISSUES

Implementers of virtual memory systems have to make choices among the major theoretical algorithms, such as second chance versus aging, local versus global page allocation, and demand paging versus prepaging. But they also have to be aware of a number of practical implementation issues as well. In this section we will take a look at a few of the common problems and some solutions.

3.6.1 Operating System Involvement with Paging

There are four times when the operating system has paging-related work to do: process creation time, process execution time, page fault time, and process termination time. We will now briefly examine each of these to see what has to be done.

When a new process is created in a paging system, the operating system has to determine how large the program and data will be (initially) and create a page table for them. Space has to be allocated in memory for the page table and it has to be initialized. The page table need not be resident when the process is swapped out but has to be in memory when the process is running. In addition, space has to be allocated in the swap area on disk so that when a page is swapped out, it has somewhere to go. The swap area also has to be initialized with program text and data so that when the new process starts getting page faults, the pages can be brought in. Some systems page the program text directly from the executable file, thus saving disk space and initialization time. Finally, information about the page table and swap area on disk must be recorded in the process table.

When a process is scheduled for execution, the MMU has to be reset for the new process and the TLB flushed, to get rid of traces of the previously executing process. The new process' page table has to be made current, usually by copying it or a pointer to it to some hardware register(s). Optionally, some or all of the process' pages can be brought into memory to reduce the number of page faults initially (e.g., it is certain that the page pointed to by the program counter will be needed).

When a page fault occurs, the operating system has to read out hardware registers to determine which virtual address caused the fault. From this information, it must compute which page is needed and locate that page on disk. It must then find an available page frame in which to put the new page, evicting some old page if need be. Then it must read the needed page into the page frame. Finally, it must back up the program counter to have it point to the faulting instruction and let that instruction execute again.

When a process exits, the operating system must release its page table, its pages, and the disk space that the pages occupy when they are on disk. If some of the pages are shared with other processes, the pages in memory and on disk can be released only when the last process using them has terminated.

3.6.2 Page Fault Handling

We are finally in a position to describe in detail what happens on a page fault. The sequence of events is as follows:

1. The hardware traps to the kernel, saving the program counter on the stack. On most machines, some information about the state of the current instruction is saved in special CPU registers.

2. An assembly-code routine is started to save the general registers and other volatile information, to keep the operating system from destroying it. This routine calls the operating system as a procedure.

3. The operating system discovers that a page fault has occurred, and tries to discover which virtual page is needed. Often one of the hardware registers contains this information. If not, the operating system

must retrieve the program counter, fetch the instruction, and parse it in software to figure out what it was doing when the fault hit.

4. Once the virtual address that caused the fault is known, the system checks to see if this address is valid and the protection is consistent with the access. If not, the process is sent a signal or killed. If the address is valid and no protection fault has occurred, the system checks to see if a page frame is free. If no frames are free, the page replacement algorithm is run to select a victim.
5. If the page frame selected is dirty, the page is scheduled for transfer to the disk, and a context switch takes place, suspending the faulting process and letting another one run until the disk transfer has completed. In any event, the frame is marked as busy to prevent it from being used for another purpose.
6. As soon as the page frame is clean (either immediately or after it is written to disk), the operating system looks up the disk address where the needed page is, and schedules a disk operation to bring it in. While the page is being loaded, the faulting process is still suspended and another user process is run, if one is available.
7. When the disk interrupt indicates that the page has arrived, the page tables are updated to reflect its position, and the frame is marked as being in the normal state.
8. The faulting instruction is backed up to the state it had when it began and the program counter is reset to point to that instruction.
9. The faulting process is scheduled, and the operating system returns to the (assembly-language) routine that called it.
10. This routine reloads the registers and other state information and returns to user space to continue execution, as if no fault had occurred.

3.6.3 Instruction Backup

When a program references a page that is not in memory, the instruction causing the fault is stopped partway through and a trap to the operating system occurs. After the operating system has fetched the page needed, it must restart the instruction causing the trap. This is easier said than done.

To see the nature of this problem at its worst, consider a CPU that has instructions with two addresses, such as the Motorola 680x0, widely used in embedded systems. The instruction

```
MOV.L #6(A1),2(A0)
```

is 6 bytes, for example (see Fig. 3-27). In order to restart the instruction, the operating system must determine where the first byte of the instruction is. The value of the program counter at the time of the trap depends on which operand faulted and how the CPU's microcode has been implemented.

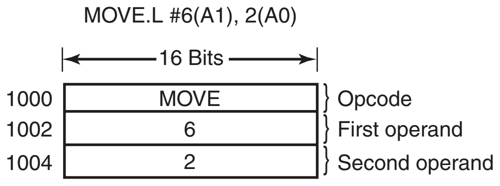


Figure 3-27. An instruction causing a page fault.

In Fig. 3-27, we have an instruction starting at address 1000 that makes three memory references: the instruction word and two offsets for the operands. Depending on which of these three memory references caused the page fault, the program counter might be 1000, 1002, or 1004 at the time of the fault. It is frequently impossible for the operating system to determine unambiguously where the instruction began. If the program counter is 1002 at the time of the fault, the operating system has no way of telling whether the word in 1002 is a memory address associated with an instruction at 1000 (e.g., the address of an operand) or an opcode.

Bad as this problem may be, it could have been worse. Some 680x0 addressing modes use autoincrementing, which means that a side effect of executing the instruction is to increment one (or more) registers. Instructions that use autoincrement mode can also fault. Depending on the details of the microcode, the increment may be done before the memory reference, in which case the operating system must decrement the register in software before restarting the instruction. Or, the autoincrement may be done after the memory reference, in which case it will not have been done at the time of the trap and must not be undone by the operating system. Autodecrement mode also exists and causes a similar problem. The precise details of whether autoincrements and autodecrements have or have not been done before the corresponding memory references may differ from instruction to instruction and from CPU model to CPU model.

Fortunately, on some machines the CPU designers provide a solution, usually in the form of a hidden internal register into which the program counter is copied just before each instruction is executed. These machines may also have a second register telling which registers have already been autoincremented or autodecremented, and by how much. Given this information, the operating system can unambiguously undo all the effects of the faulting instruction so that it can be restarted. If this information is not available, the operating system has to jump through hoops to figure out what happened and how to repair it. It is as though the hardware designers were unable to solve the problem, so they threw up their hands and told the operating system writers to deal with it. Nice guys.

3.6.4 Locking Pages in Memory

Although we have not discussed I/O much in this chapter, the fact that a computer has virtual memory does not mean that I/O is absent. Virtual memory and I/O interact in subtle ways. Consider a process that has just issued a system call to read from some file or device into a buffer within its address space. While waiting for the I/O to complete, the process is suspended and another process is allowed to run. This other process gets a page fault.

If the paging algorithm is global, there is a small, but nonzero, chance that the page containing the I/O buffer will be chosen to be removed from memory. If an I/O device is currently in the process of doing a DMA transfer to that page, removing it will cause part of the data to be written in the buffer where they belong, and part of the data to be written over the just-loaded page. One solution to this problem is to lock pages engaged in I/O in memory so that they will not be removed. Locking a page is often called **pinning** it in memory. Another solution is to do all I/O to kernel buffers and then copy the data to user pages later.

3.6.5 Backing Store

In our discussion of page replacement algorithms, we saw how a page is selected for removal. We have not said much about where on the disk it is put when it is paged out. Let us now describe some of the issues related to disk management.

The simplest algorithm for allocating page space on the disk is to have a special swap partition on the disk or, even better, on a separate disk from the file system (to balance the I/O load). Most UNIX systems work like this. This partition does not have a normal file system on it, which eliminates all the overhead of converting offsets in files to block addresses. Instead, block numbers relative to the start of the partition are used throughout.

When the system is booted, this swap partition is empty and is represented in memory as a single entry giving its origin and size. In the simplest scheme, when the first process is started, a chunk of the partition area the size of the first process is reserved and the remaining area reduced by that amount. As new processes are started, they are assigned chunks of the swap partition equal in size to their core images. As they finish, their disk space is freed. The swap partition is managed as a list of free chunks. Better algorithms will be discussed in Chap. 10.

Associated with each process is the disk address of its swap area, that is, where on the swap partition its image is kept. This information is kept in the process table. Calculating the address to write a page to becomes simple: just add the offset of the page within the virtual address space to the start of the swap area. However, before a process can start, the swap area must be initialized. One way is to copy the entire process image to the swap area, so that it can be brought *in* as needed. The other is to load the entire process in memory and let it be paged *out* as needed.

However, this simple model has a problem: processes can increase in size after starting. Although the program text is usually fixed, the data area can sometimes grow, and the stack can always grow. Consequently, it may be better to reserve separate swap areas for the text, data, and stack and allow each of these areas to consist of more than one chunk on the disk.

The other extreme is to allocate nothing in advance and allocate disk space for each page when it is swapped out and deallocate it when it is swapped back in. In this way, processes in memory do not tie up any swap space. The disadvantage is that a disk address is needed in memory to keep track of each page on disk. In other words, there must be a table per process telling for each page on disk where it is. The two alternatives are shown in Fig. 3-28.

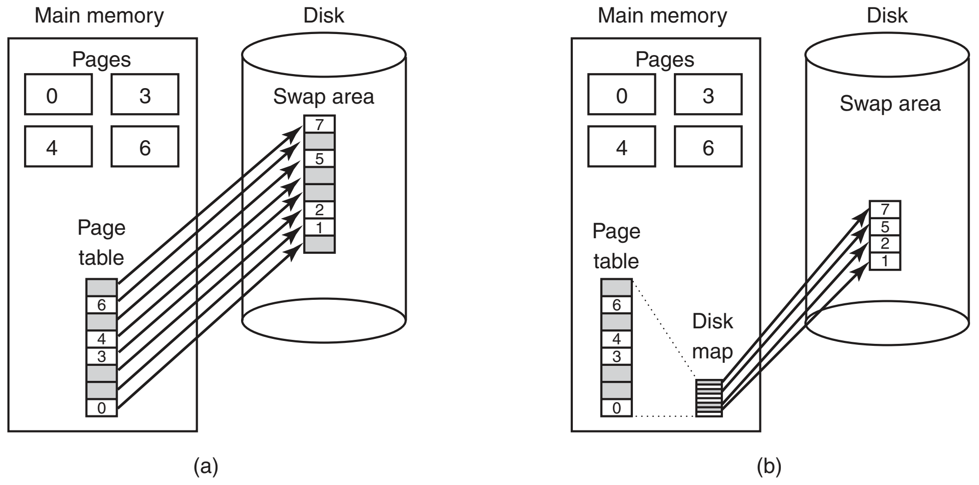


Figure 3-28. (a) Paging to a static swap area. (b) Backing up pages dynamically.

In Fig. 3-28(a), a page table with eight pages is shown. Pages 0, 3, 4, and 6 are in main memory. Pages 1, 2, 5, and 7 are on disk. The swap area on disk is as large as the process virtual address space (eight pages), with each page having a fixed location to which it is written when it is evicted from main memory. Calculating this address requires knowing only where the process' paging area begins, since pages are stored in it contiguously in order of their virtual page number. A page that is in memory always has a shadow copy on disk, but this copy may be out of date if the page has been modified since being loaded. The shaded pages in memory indicate pages not present in memory. The shaded pages on the disk are (in principle) superseded by the copies in memory, although if a memory page has to be swapped back to disk and it has not been modified since it was loaded, the (shaded) disk copy will be used.

In Fig. 3-28(b), pages do not have fixed addresses on disk. When a page is swapped out, an empty disk page is chosen on the fly and the disk map (which has room for one disk address per virtual page) is updated accordingly. A page in memory has no copy on disk. The pages' entries in the disk map contain an invalid disk address or a bit marking them as not in use.

Having a fixed swap partition is not always possible. For example, no disk partitions may be available. In this case, one or more large, preallocated files within the normal file system can be used. Windows uses this approach. However, an optimization can be used here to reduce the amount of disk space needed. Since the program text of every process came from some (executable) file in the file system, the executable file can be used as the swap area. Better yet, since the program text is generally read only, when memory is tight and program pages have to be evicted from memory, they are just discarded and read in again from the executable file when needed. Shared libraries can also work this way.

3.6.6 Separation of Policy and Mechanism

An important tool for managing the complexity of any system is to split policy from mechanism. This principle can be applied to memory management by having most of the memory manager run as a user-level process. Such a separation was first done in Mach (Young et al., 1987) on which the discussion below is based.

A simple example of how policy and mechanism can be separated is shown in Fig. 3-29. Here the memory management system is divided into three parts:

1. A low-level MMU handler.
2. A page fault handler that is part of the kernel.
3. An external pager running in user space.

All the details of how the MMU works are encapsulated in the MMU handler, which is machine-dependent code and has to be rewritten for each new platform the operating system is ported to. The page-fault handler is machine-independent code and contains most of the mechanism for paging. The policy is largely determined by the external pager, which runs as a user process.

When a process starts up, the external pager is notified in order to set up the process' page map and allocate the necessary backing store on the disk if need be. As the process runs, it may map new objects into its address space, so the external pager is once again notified.

Once the process starts running, it may get a page fault. The fault handler figures out which virtual page is needed and sends a message to the external pager, telling it the problem. The external pager then reads the needed page in from the disk and copies it to a portion of its own address space. Then it tells the fault handler where the page is. The fault handler then unmaps the page from the external pager's address space and asks the MMU handler to put it into the user's address space at the right place. Then the user process can be restarted.

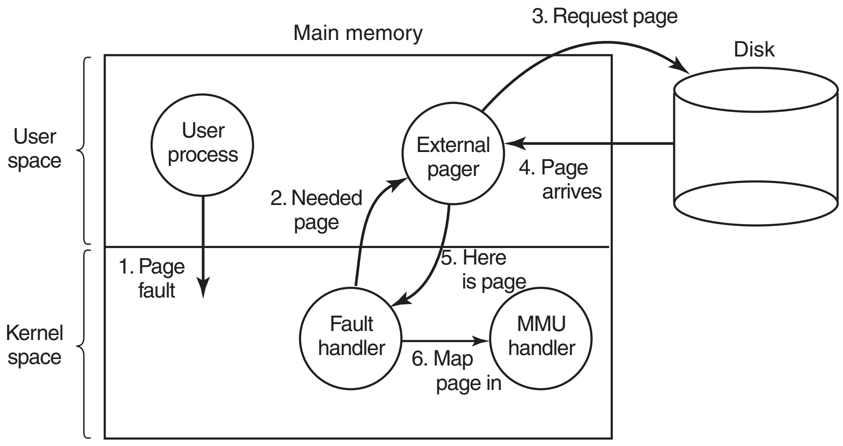


Figure 3-29. Page fault handling with an external pager.

This implementation leaves open where the page replacement algorithm is put. It would be cleanest to have it in the external pager, but there are some problems with this approach. Principal among these is that the external pager does not have access to the *R* and *M* bits of all the pages. These bits play a role in many of the paging algorithms. Thus, either some mechanism is needed to pass this information up to the external pager, or the page replacement algorithm must go in the kernel. In the latter case, the fault handler tells the external pager which page it has selected for eviction and provides the data, either by mapping it into the external pager's address space or including it in a message. Either way, the external pager writes the data to disk.

The main advantage of this implementation is more modular code and greater flexibility. The main disadvantage is the extra overhead of crossing the user-kernel boundary several times and the overhead of the various messages being sent between the pieces of the system. At the moment, the subject is highly controversial, but as computers get faster and faster, and the software gets more and more complex, in the long run sacrificing some performance for more reliable software will probably be acceptable to most implementers.

3.7 SEGMENTATION

The virtual memory discussed so far is one-dimensional because the virtual addresses go from 0 to some maximum address, one address after another. For many problems, having two or more separate virtual address spaces may be much better than having only one. For example, a compiler has many tables that are built up as compilation proceeds, possibly including

1. The source text being saved for the printed listing (on batch systems).
2. The symbol table, containing the names and attributes of variables.
3. The table containing all the integer and floating-point constants used.
4. The parse tree, containing the syntactic analysis of the program.
5. The stack used for procedure calls within the compiler.

Each of the first four tables grows continuously as compilation proceeds. The last one grows and shrinks in unpredictable ways during compilation. In a one-dimensional memory, these five tables would have to be allocated contiguous chunks of virtual address space, as in Fig. 3-30.

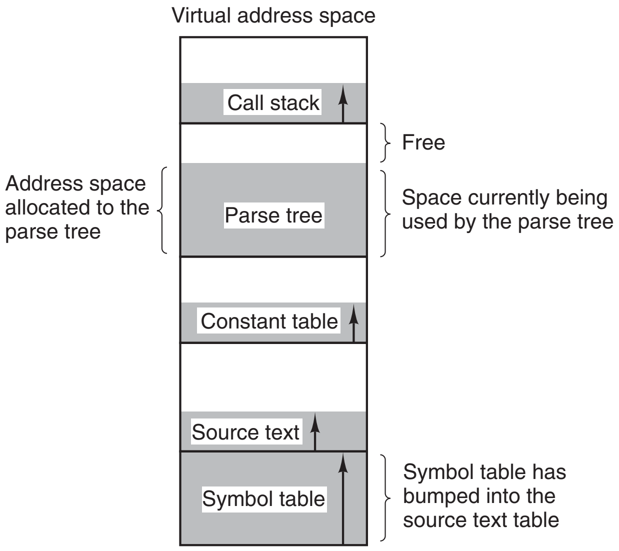

Virtual address space
Call stack
Free
Address space allocated to the parse tree
Parse tree
Space currently being used by the parse tree
Constant table
Source text
Symbol table
Symbol table has bumped into the source text table

Figure 3-30. In a one-dimensional address space with growing tables, one table may bump into another.

Consider what happens if a program has a much larger than usual number of variables but a normal amount of everything else. The chunk of address space allocated for the symbol table may fill up, but there may be lots of room in the other tables. What is needed is a way of freeing the programmer from having to manage the expanding and contracting tables, in the same way that virtual memory eliminates the worry of organizing the program into overlays.

A straightforward and quite general solution is to provide the machine with many completely independent address spaces, which are called **segments**. Each segment consists of a linear sequence of addresses, starting at 0 and going up to some maximum value. The length of each segment may be anything from 0 to the maximum address allowed. Different segments may, and usually do, have different lengths. Moreover, segment lengths may change during execution. The length of a stack segment may be increased whenever something is pushed onto the stack and decreased whenever something is popped off the stack.

Because each segment constitutes a separate address space, different segments can grow or shrink independently without affecting each other. If a stack in a certain segment needs more address space to grow, it can have it, because there is nothing else in its address space to bump into. Of course, a segment can fill up, but segments are usually very large, so this occurrence is rare. To specify an address in this segmented or two-dimensional memory, the program must supply a two-part address, a segment number, and an address within the segment. Figure 3-31 illustrates a segmented memory being used for the compiler tables discussed earlier. Five independent segments are shown here.

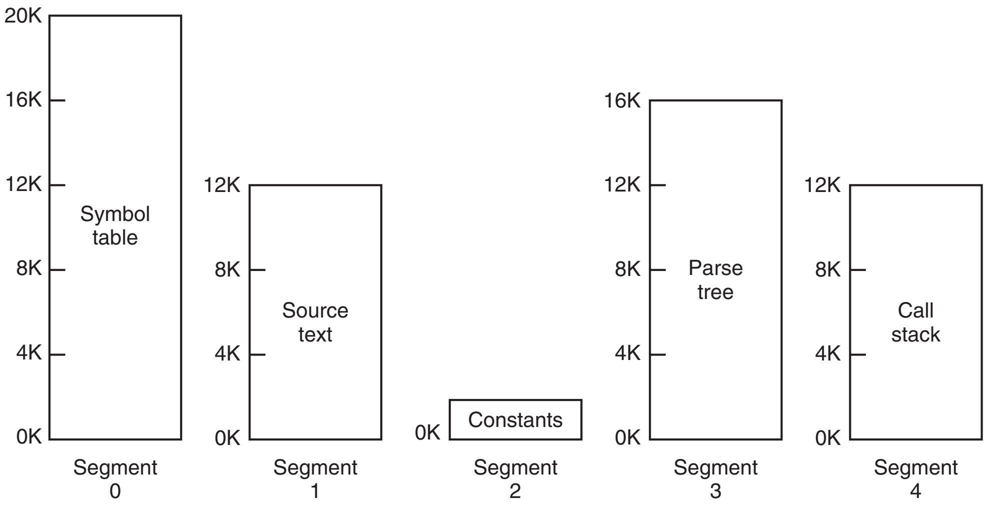


Figure 3-31. A segmented memory allows each table to grow or shrink independently of the other tables.

We emphasize here that a segment is a logical entity, which the programmer is aware of and uses as a logical entity. A segment might contain a procedure, or an array, or a stack, or a collection of scalar variables, but usually it does not contain a mixture of different types.

A segmented memory has other advantages besides simplifying the handling of data structures that are growing or shrinking. If each procedure occupies a separate segment, with address 0 as its starting address, the linking of procedures compiled separately is greatly simplified. After all the procedures that constitute a program have been compiled and linked up, a procedure call to the procedure in segment n will use the two-part address $(n, 0)$ to address word 0 (the entry point).

If the procedure in segment *n* is subsequently modified and recompiled, no other procedures need be changed (because no starting addresses have been modified), even if the new version is larger than the old one. With a one-dimensional memory, the procedures are packed tightly right up next to each other, with no address space between them. Consequently, changing one procedure's size can affect the starting address of all the other (unrelated) procedures in the segment. This, in turn, requires modifying all procedures that call any of the moved procedures, in order to incorporate their new starting addresses. If a program contains hundreds of procedures, this process can be costly.

Segmentation also facilitates sharing procedures or data between several processes. A common example is the shared library. Modern workstations that run advanced window systems often have extremely large graphical libraries compiled into nearly every program. In a segmented system, the graphical library can be put in a segment and shared by multiple processes, eliminating the need for having it in every process' address space. While it is also possible to have shared libraries in pure paging systems, it is more complicated. In effect, these systems do it by simulating segmentation.

Since each segment forms a logical entity that programmers know about, such as a procedure, or an array, different segments can have different kinds of protection. A procedure segment can be specified as execute only, prohibiting attempts to read from or store into it. A floating-point array can be specified as read/write but not execute, and attempts to jump to it will be caught. Such protection is helpful in catching bugs. Paging and segmentation are compared in Fig. 3-32.

3.7.1 Implementation of Pure Segmentation

The implementation of segmentation differs from paging in an essential way: pages are of fixed size and segments are not. Figure 3-33(a) shows an example of physical memory initially containing five segments. Now consider what happens if segment 1 is evicted and segment 7, which is smaller, is put in its place. We arrive at the memory configuration of Fig. 3-33(b). Between segment 7 and segment 2 is an unused area—that is, a hole. Then segment 4 is replaced by segment 5, as in Fig. 3-33(c), and segment 3 is replaced by segment 6, as in Fig. 3-33(d). After the system has been running for a while, memory will be divided up into a number of chunks, some containing segments and some containing holes. This phenomenon, called **checkerboarding** or **external fragmentation**, wastes memory in the holes. It can be dealt with by compaction, as shown in Fig. 3-33(e).

3.7.2 Segmentation with Paging: MULTICS

If the segments are large, it may be inconvenient, or even impossible, to keep them in main memory in their entirety. This leads to the idea of paging them, so that only those pages of a segment that are actually needed have to be around.

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Figure 3-32. Comparison of paging and segmentation.

Several significant systems have supported paged segments. In this section we will describe the first one: MULTICS. In the next one we will discuss a more recent one: the Intel x86 up until the x86-64.

The MULTICS operating system was one of the most influential operating systems ever, having had a major influence on topics as disparate as UNIX, the x86 memory architecture, TLBs, and cloud computing. It was started as a research project at M.I.T. and went live in 1969. The last MULTICS system was shut down in 2000, a run of 31 years. Few other operating systems have lasted more-or-less unmodified anywhere near that long. While operating systems called Windows have also have be around that long, Windows 8 has absolutely nothing in common with Windows 1.0 except the name and the fact that it was written by Microsoft. Even more to the point, the ideas developed in MULTICS are as valid and useful now as they were in 1965, when the first paper was published (Corbató and Vyssotsky, 1965). For this reason, we will now spend a little bit of time looking at the most innovative aspect of MULTICS, the virtual memory architecture. More information about MULTICS can be found at *www.multicians.org*.

MULTICS ran on the Honeywell 6000 machines and their descendants and provided each program with a virtual memory of up to 2^{18} segments, each of which

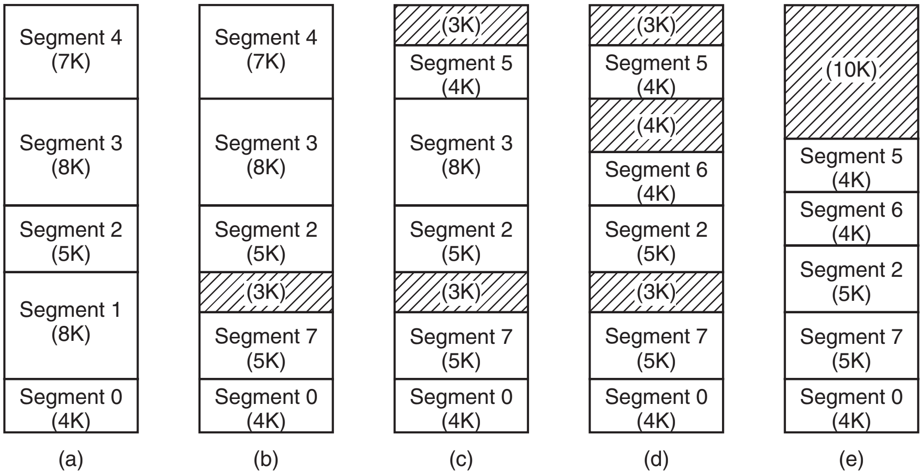


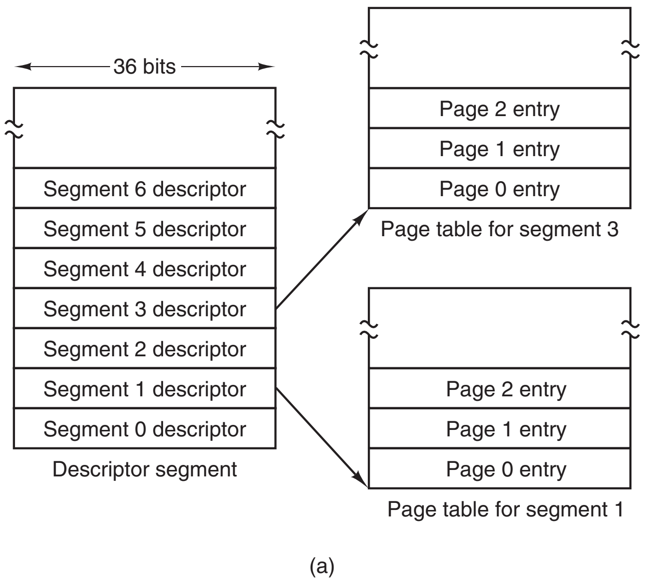
Figure 3-33. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

was up to 65,536 (36-bit) words long. To implement this, the MULTICS designers chose to treat each segment as a virtual memory and to page it, combining the advantages of paging (uniform page size and not having to keep the whole segment in memory if only part of it was being used) with the advantages of segmentation (ease of programming, modularity, protection, sharing).

Each MULTICS program had a segment table, with one descriptor per segment. Since there were potentially more than a quarter of a million entries in the table, the segment table was itself a segment and was paged. A segment descriptor contained an indication of whether the segment was in main memory or not. If any part of the segment was in memory, the segment was considered to be in memory, and its page table was in memory. If the segment was in memory, its descriptor contained an 18-bit pointer to its page table, as in Fig. 3-34(a). Because physical addresses were 24 bits and pages were aligned on 64-byte boundaries (implying that the low-order 6 bits of page addresses were 000000), only 18 bits were needed in the descriptor to store a page table address. The descriptor also contained the segment size, the protection bits, and other items. Figure 3-34(b) illustrates a segment descriptor. The address of the segment in secondary memory was not in the segment descriptor but in another table used by the segment fault handler.

Each segment was an ordinary virtual address space and was paged in the same way as the nonsegmented paged memory described earlier in this chapter. The normal page size was 1024 words (although a few small segments used by MULTICS itself were not paged or were paged in units of 64 words to save physical memory).

An address in MULTICS consisted of two parts: the segment and the address within the segment. The address within the segment was further divided into a page



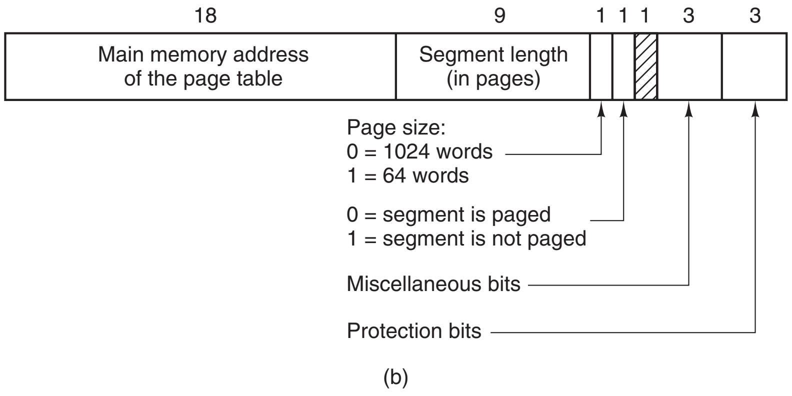


Figure 3-34. The MULTICS virtual memory. (a) The descriptor segment pointed to the page tables. (b) A segment descriptor. The numbers are the field lengths.

number and a word within the page, as shown in Fig. 3-35. When a memory reference occurred, the following algorithm was carried out.

1. The segment number was used to find the segment descriptor.
2. A check was made to see if the segment's page table was in memory. If it was, it was located. If it was not, a segment fault occurred. If there was a protection violation, a fault (trap) occurred.

3. The page table entry for the requested virtual page was examined. If the page itself was not in memory, a page fault was triggered. If it was in memory, the main-memory address of the start of the page was extracted from the page table entry.
4. The offset was added to the page origin to give the main memory address where the word was located.
5. The read or store finally took place.

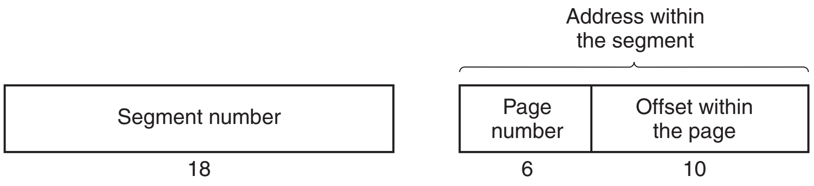


Figure 3-35. A 34-bit MULTICS virtual address.

This process is illustrated in Fig. 3-36. For simplicity, the fact that the descriptor segment was itself paged has been omitted. What really happened was that a register (the descriptor base register) was used to locate the descriptor segment's page table, which, in turn, pointed to the pages of the descriptor segment. Once the descriptor for the needed segment was been found, the addressing proceeded as shown in Fig. 3-36.

As you have no doubt guessed by now, if the preceding algorithm were actually carried out by the operating system on every instruction, programs would not run very fast. In reality, the MULTICS hardware contained a 16-word high-speed TLB that could search all its entries in parallel for a given key. This was the first system to have a TLB, something used in all modern architectures. It is illustrated in Fig. 3-37. When an address was presented to the computer, the addressing hardware first checked to see if the virtual address was in the TLB. If so, it got the page frame number directly from the TLB and formed the actual address of the referenced word without having to look in the descriptor segment or page table.

The addresses of the 16 most recently referenced pages were kept in the TLB. Programs whose working set was smaller than the TLB size came to equilibrium with the addresses of the entire working set in the TLB and therefore ran efficiently; otherwise, there were TLB faults.

3.7.3 Segmentation with Paging: The Intel x86

Up until the x86-64, the virtual memory system of the x86 resembled that of MULTICS in many ways, including the presence of both segmentation and paging. Whereas MULTICS had 256K independent segments, each up to 64K 36-bit words, the x86 has 16K independent segments, each holding up to 1 billion 32-bit

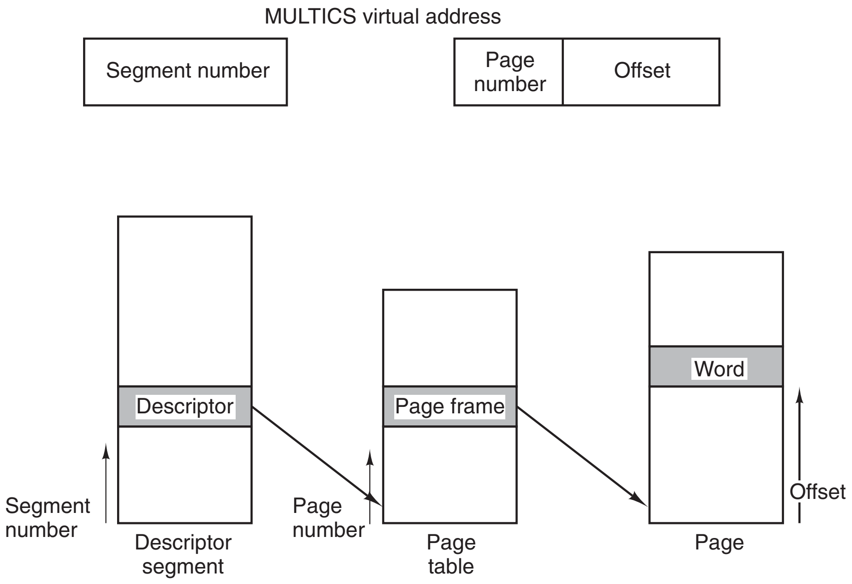
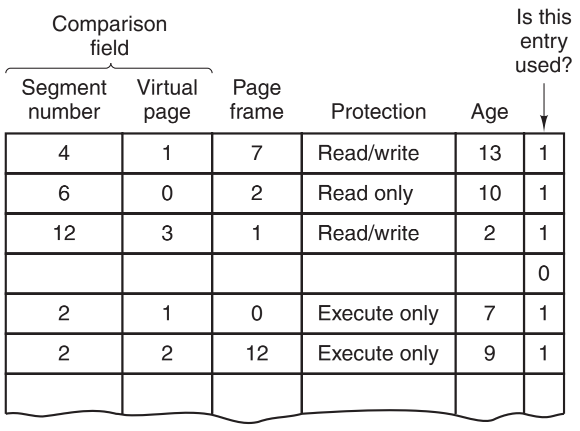


Figure 3-36. Conversion of a two-part MULTICS address into a main memory address.



Comparison field					Is this entry used?
Segment number	Virtual page	Page frame	Protection	Age	
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Figure 3-37. A simplified version of the MULTICS TLB. The existence of two page sizes made the actual TLB more complicated.

words. Although there are fewer segments, the larger segment size is far more important, as few programs need more than 1000 segments, but many programs need large segments. As of x86-64, segmentation is considered obsolete and is no longer supported, except in legacy mode. Although some vestiges of the old segmentation mechanisms are still available in x86-64's native mode, mostly for compatibility, they no longer serve the same role and no longer offer true segmentation. The x86-32, however, still comes equipped with the whole shebang and it is the CPU we will discuss in this section.

The heart of the x86 virtual memory consists of two tables, called the **LDT** (**Local Descriptor Table**) and the **GDT** (**Global Descriptor Table**). Each program has its own LDT, but there is a single GDT, shared by all the programs on the computer. The LDT describes segments local to each program, including its code, data, stack, and so on, whereas the GDT describes system segments, including the operating system itself.

To access a segment, an x86 program first loads a selector for that segment into one of the machine's six segment registers. During execution, the CS register holds the selector for the code segment and the DS register holds the selector for the data segment. The other segment registers are less important. Each selector is a 16-bit number, as shown in Fig. 3-38.

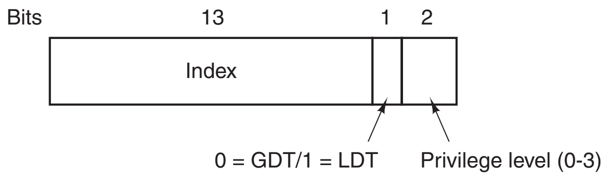


Figure 3-38. An x86 selector.

One of the selector bits tells whether the segment is local or global (i.e., whether it is in the LDT or GDT). Thirteen other bits specify the LDT or GDT entry number, so these tables are each restricted to holding 8K segment descriptors. The other 2 bits relate to protection, and will be described later. Descriptor 0 is forbidden. It may be safely loaded into a segment register to indicate that the segment register is not currently available. It causes a trap if used.

At the time a selector is loaded into a segment register, the corresponding descriptor is fetched from the LDT or GDT and stored in microprogram registers, so it can be accessed quickly. As depicted in Fig. 3-39, a descriptor consists of 8 bytes, including the segment's base address, size, and other information.

The format of the selector has been cleverly chosen to make locating the descriptor easy. First either the LDT or GDT is selected, based on selector bit 2. Then the selector is copied to an internal scratch register, and the 3 low-order bits set to 0. Finally, the address of either the LDT or GDT table is added to it, to give a direct pointer to the descriptor. For example, selector 72 refers to entry 9 in the GDT, which is located at address GDT + 72.

Let us now trace the steps by which a (selector, offset) pair is converted to a physical address. As soon as the microprogram knows which segment register is

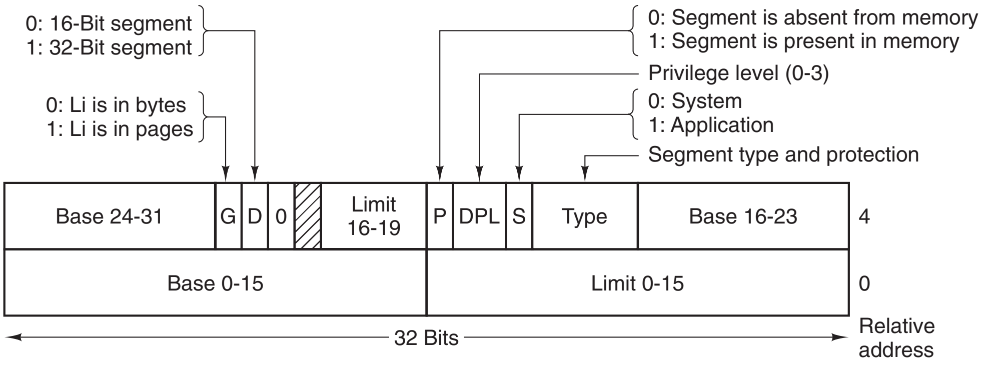


Figure 3-39. x86 code segment descriptor. Data segments differ slightly.

being used, it can find the complete descriptor corresponding to that selector in its internal registers. If the segment does not exist (selector 0), or is currently paged out, a trap occurs.

The hardware then uses the *Limit* field to check if the offset is beyond the end of the segment, in which case a trap also occurs. Logically, there should be a 32-bit field in the descriptor giving the size of the segment, but only 20 bits are available, so a different scheme is used. If the *Gbit* (Granularity) field is 0, the *Limit* field is the exact segment size, up to 1 MB. If it is 1, the *Limit* field gives the segment size in pages instead of bytes. With a page size of 4 KB, 20 bits are enough for segments up to 2^{32} bytes.

Assuming that the segment is in memory and the offset is in range, the x86 then adds the 32-bit *Base* field in the descriptor to the offset to form what is called a **linear address**, as shown in Fig. 3-40. The *Base* field is broken up into three pieces and spread all over the descriptor for compatibility with the 286, in which the *Base* is only 24 bits. In effect, the *Base* field allows each segment to start at an arbitrary place within the 32-bit linear address space.

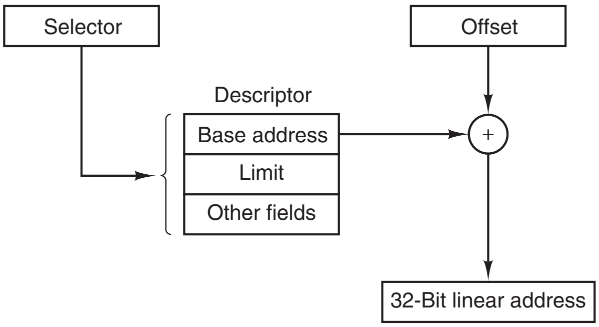
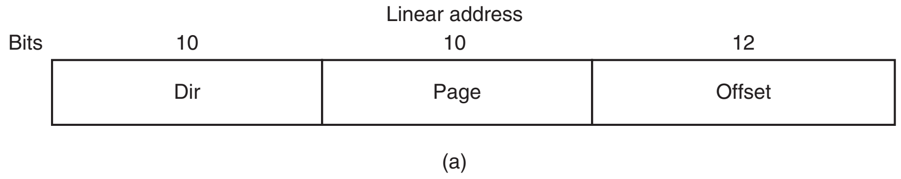


Figure 3-40. Conversion of a (selector, offset) pair to a linear address.

If paging is disabled (by a bit in a global control register), the linear address is interpreted as the physical address and sent to the memory for the read or write. Thus with paging disabled, we have a pure segmentation scheme, with each segment's base address given in its descriptor. Segments are not prevented from overlapping, probably because it would be too much trouble and take too much time to verify that they were all disjoint.

On the other hand, if paging is enabled, the linear address is interpreted as a virtual address and mapped onto the physical address using page tables, pretty much as in our earlier examples. The only real complication is that with a 32-bit virtual address and a 4-KB page, a segment might contain 1 million pages, so a two-level mapping is used to reduce the page table size for small segments.

Each running program has a page directory consisting of 1024 32-bit entries. It is located at an address pointed to by a global register. Each entry in this directory points to a page table also containing 1024 32-bit entries. The page table entries point to page frames. The scheme is shown in Fig. 3-41.



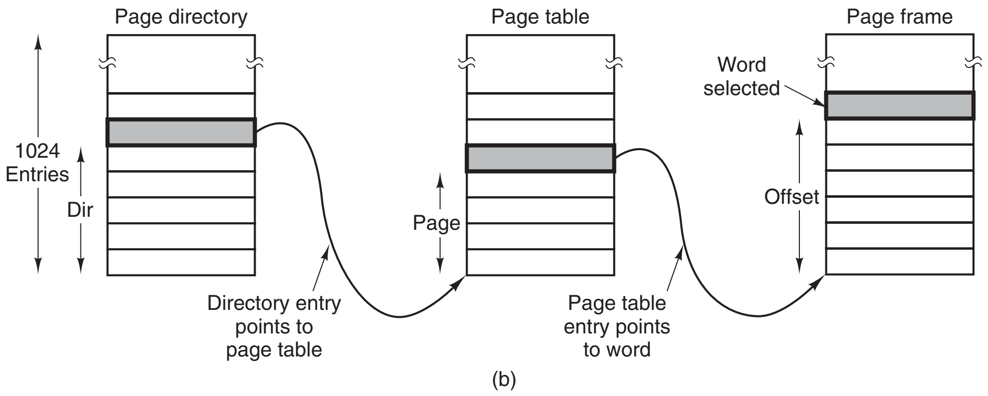


Figure 3-41. Mapping of a linear address onto a physical address.

In Fig. 3-41(a) we see a linear address divided into three fields, *Dir*, *Page*, and *Offset*. The *Dir* field is used to index into the page directory to locate a pointer to the proper page table. Then the *Page* field is used as an index into the page table to find the physical address of the page frame. Finally, *Offset* is added to the address of the page frame to get the physical address of the byte or word needed.

The page table entries are 32 bits each, 20 of which contain a page frame number. The remaining bits contain access and dirty bits, set by the hardware for the benefit of the operating system, protection bits, and other utility bits.

Each page table has entries for 1024 4-KB page frames, so a single page table handles 4 megabytes of memory. A segment shorter than 4M will have a page directory with a single entry, a pointer to its one and only page table. In this way, the overhead for short segments is only two pages, instead of the million pages that would be needed in a one-level page table.

To avoid making repeated references to memory, the x86, like MULTICS, has a small TLB that directly maps the most recently used *Dir-Page* combinations onto the physical address of the page frame. Only when the current combination is not present in the TLB is the mechanism of Fig. 3-41 actually carried out and the TLB updated. As long as TLB misses are rare, performance is good.

It is also worth noting that if some application does not need segmentation but is simply content with a single, paged, 32-bit address space, that model is possible. All the segment registers can be set up with the same selector, whose descriptor has *Base* = 0 and *Limit* set to the maximum. The instruction offset will then be the linear address, with only a single address space used—in effect, normal paging. In fact, all current operating systems for the x86 work this way. OS/2 was the only one that used the full power of the Intel MMU architecture.

So why did Intel kill what was a variant of the perfectly good MULTICS memory model that it supported for close to three decades? Probably the main reason is that neither UNIX nor Windows ever used it, even though it was quite efficient because it eliminated system calls, turning them into lightning-fast procedure calls to the relevant address within a protected operating system segment. None of the developers of any UNIX or Windows system wanted to change their memory model to something that was x86 specific because it would break portability to other platforms. Since the software was not using the feature, Intel got tired of wasting chip area to support it and removed it from the 64-bit CPUs.

All in all, one has to give credit to the x86 designers. Given the conflicting goals of implementing pure paging, pure segmentation, and paged segments, while at the same time being compatible with the 286, and doing all of this efficiently, the resulting design is surprisingly simple and clean.

3.8 RESEARCH ON MEMORY MANAGEMENT

Traditional memory management, especially paging algorithms for uniprocessor CPUs, was once a fruitful area for research, but most of that seems to have largely died off, at least for general-purpose systems, although there are some people who never say die (Moruz et al., 2012) or are focused on some application, such as online transaction processing, that has specialized requirements (Stoica and Ailamaki, 2013). Even on uniprocessors, paging to SSDs rather than to hard disks brings up new issues and requires new algorithms (Chen et al., 2012). Paging to the up-and-coming nonvolatile phase-change memories also requires rethinking paging for performance (Lee et al., 2013), and latency reasons (Saito and Oikawa, 2012), and because they wear out if used too much (Bheda et al., 2011, 2012).

More generally, research on paging is still ongoing, but it focuses on newer kinds of systems. For example, virtual machines have rekindled interest in memory management (Bugnion et al., 2012). In the same area, the work by Jantz et al. (2013) lets applications provide guidance to the system with respect to deciding on the physical page to back a virtual page. An aspect of server consolidation in the cloud that affects paging is that the amount of physical memory available to a virtual machine can vary over time, requiring new algorithms (Peserico, 2013).

Paging in multicore systems has become a hot new area of research (Boyd-Wickizer et al., 2008, Baumann et al., 2009). One contributing factor is that multicore systems tend to have a lot of caches shared in complex ways (Lopez-Ortiz and Salinger, 2012). Closely related to this multicore work is research on paging in NUMA systems, where different pieces of memory may have different access times (Dashti et al., 2013; and Lankes et al., 2012).

Also, smartphones and tablets have become small PCs and many of them page RAM to "disk," only disk on a smartphone is flash memory. Some recent work is reported by Joo et al. (2012).

Finally, interest is memory management for real-time systems continues to be present (Kato et al., 2011).

3.9 SUMMARY

In this chapter we have examined memory management. We saw that the simplest systems do not swap or page at all. Once a program is loaded into memory, it remains there in place until it finishes. Some operating systems allow only one process at a time in memory, while others support multiprogramming. This model is still common in small, embedded real-time systems.

The next step up is swapping. When swapping is used, the system can handle more processes than it has room for in memory. Processes for which there is no room are swapped out to the disk. Free space in memory and on disk can be kept track of with a bitmap or a hole list.

Modern computers often have some form of virtual memory. In the simplest form, each process' address space is divided up into uniform-sized blocks called pages, which can be placed into any available page frame in memory. There are many page replacement algorithms; two of the better algorithms are aging and WSClock.

To make paging systems work well, choosing an algorithm is not enough; attention to such issues as determining the working set, memory allocation policy, and page size is required.

Segmentation helps in handling data structures that can change size during execution and simplifies linking and sharing. It also facilitates providing different protection for different segments. Sometimes segmentation and paging are combined to provide a two-dimensional virtual memory. The MULTICS system and the 32-bit Intel x86 support segmentation and paging. Still, it is clear that few operating system developers care deeply about segmentation (because they are married to a different memory model). Consequently, it seems to be going out of fashion fast. Today, even the 64-bit version of the x86 no longer supports real segmentation.

PROBLEMS

1. The IBM 360 had a scheme of locking 2-KB blocks by assigning each one a 4-bit key and having the CPU compare the key on every memory reference to the 4-bit key in the PSW. Name two drawbacks of this scheme not mentioned in the text.

2. In Fig. 3-3 the base and limit registers contain the same value, 16,384. Is this just an accident, or are they always the same? It is just an accident, why are they the same in this example?

3. A swapping system eliminates holes by compaction. Assuming a random distribution of many holes and many data segments and a time to read or write a 32-bit memory word of 4 nsec, about how long does it take to compact 4 GB? For simplicity, assume that word 0 is part of a hole and that the highest word in memory contains valid data.

4. Consider a swapping system in which memory consists of the following hole sizes in memory order: 10 MB, 4 MB, 20 MB, 18 MB, 7 MB, 9 MB, 12 MB, and 15 MB. Which hole is taken for successive segment requests of

(a) 12 MB
(b) 10 MB
(c) 9 MB

for first fit? Now repeat the question for best fit, worst fit, and next fit.

5. What is the difference between a physical address and a virtual address?

6. For each of the following decimal virtual addresses, compute the virtual page number and offset for a 4-KB page and for an 8 KB page: 20000, 32768, 60000.

7. Using the page table of Fig. 3-9, give the physical address corresponding to each of the following virtual addresses:

(a) 20
(b) 4100
(c) 8300

8. The Intel 8086 processor did not have an MMU or support virtual memory. Nevertheless, some companies sold systems that contained an unmodified 8086 CPU and did paging. Make an educated guess as to how they did it. (*Hint*: Think about the logical location of the MMU.)

9. What kind of hardware support is needed for a paged virtual memory to work?

10. Copy on write is an interesting idea used on server systems. Does it make any sense on a smartphone?

11. Consider the following C program:

```
int    X[N];
int step = M;      /* M is some predefined constant */
for (int i = 0; i < N; i += step) X[i] = X[i] + 1;
```

(a) If this program is run on a machine with a 4-KB page size and 64-entry TLB, what values of M and N will cause a TLB miss for every execution of the inner loop?
(b) Would your answer in part (a) be different if the loop were repeated many times? Explain.

12. The amount of disk space that must be available for page storage is related to the maximum number of processes, n, the number of bytes in the virtual address space, v, and the number of bytes of RAM, r. Give an expression for the worst-case disk-space requirements. How realistic is this amount?

13. If an instruction takes 1 nsec and a page fault takes an additional n nsec, give a formula for the effective instruction time if page faults occur every k instructions.

14. A machine has a 32-bit address space and an 8-KB page. The page table is entirely in hardware, with one 32-bit word per entry. When a process starts, the page table is copied to the hardware from memory, at one word every 100 nsec. If each process runs for 100 msec (including the time to load the page table), what fraction of the CPU time is devoted to loading the page tables?

15. Suppose that a machine has 48-bit virtual addresses and 32-bit physical addresses.

(a) If pages are 4 KB, how many entries are in the page table if it has only a single level? Explain.
(b) Suppose this same system has a TLB (Translation Lookaside Buffer) with 32 entries. Furthermore, suppose that a program contains instructions that fit into one page and it sequentially reads long integer elements from an array that spans thousands of pages. How effective will the TLB be for this case?

16. You are given the following data about a virtual memory system:

(a)The TLB can hold 1024 entries and can be accessed in 1 clock cycle (1 nsec).
(b) A page table entry can be found in 100 clock cycles or 100 nsec.
(c) The average page replacement time is 6 msec.

If page references are handled by the TLB 99% of the time, and only 0.01% lead to a page fault, what is the effective address-translation time?

17. Suppose that a machine has 38-bit virtual addresses and 32-bit physical addresses.

(a) What is the main advantage of a multilevel page table over a single-level one?
(b) With a two-level page table, 16-KB pages, and 4-byte entries, how many bits should be allocated for the top-level page table field and how many for the next-level page table field? Explain.

18. Section 3.3.4 states that the Pentium Pro extended each entry in the page table hierarchy to 64 bits but still could only address only 4 GB of memory. Explain how this statement can be true when page table entries have 64 bits.

19. A computer with a 32-bit address uses a two-level page table. Virtual addresses are split into a 9-bit top-level page table field, an 11-bit second-level page table field, and an offset. How large are the pages and how many are there in the address space?

20. A computer has 32-bit virtual addresses and 4-KB pages. The program and data together fit in the lowest page (0–4095) The stack fits in the highest page. How many entries are needed in the page table if traditional (one-level) paging is used? How many page table entries are needed for two-level paging, with 10 bits in each part?

21. Below is an execution trace of a program fragment for a computer with 512-byte pages. The program is located at address 1020, and its stack pointer is at 8192 (the stack grows toward 0). Give the page reference string generated by this program. Each instruction occupies 4 bytes (1 word) including immediate constants. Both instruction and data references count in the reference string.

Load word 6144 into register 0
Push register 0 onto the stack
Call a procedure at 5120, stacking the return address
Subtract the immediate constant 16 from the stack pointer
Compare the actual parameter to the immediate constant 4
Jump if equal to 5152

22. A computer whose processes have 1024 pages in their address spaces keeps its page tables in memory. The overhead required for reading a word from the page table is 5 nsec. To reduce this overhead, the computer has a TLB, which holds 32 (virtual page, physical page frame) pairs, and can do a lookup in 1 nsec. What hit rate is needed to reduce the mean overhead to 2 nsec?

23. How can the associative memory device needed for a TLB be implemented in hardware, and what are the implications of such a design for expandability?

24. A machine has 48-bit virtual addresses and 32-bit physical addresses. Pages are 8 KB. How many entries are needed for a single-level linear page table?

25. A computer with an 8-KB page, a 256-KB main memory, and a 64-GB virtual address space uses an inverted page table to implement its virtual memory. How big should the hash table be to ensure a mean hash chain length of less than 1? Assume that the hash-table size is a power of two.

26. A student in a compiler design course proposes to the professor a project of writing a compiler that will produce a list of page references that can be used to implement the optimal page replacement algorithm. Is this possible? Why or why not? Is there anything that could be done to improve paging efficiency at run time?

27. Suppose that the virtual page reference stream contains repetitions of long sequences of page references followed occasionally by a random page reference. For example, the sequence: 0, 1, ... , 511, 431, 0, 1, ... , 511, 332, 0, 1, ... consists of repetitions of the sequence 0, 1, ... , 511 followed by a random reference to pages 431 and 332.

(a) Why will the standard replacement algorithms (LRU, FIFO, clock) not be effective in handling this workload for a page allocation that is less than the sequence length?
(b) If this program were allocated 500 page frames, describe a page replacement approach that would perform much better than the LRU, FIFO, or clock algorithms.

28. If FIFO page replacement is used with four page frames and eight pages, how many page faults will occur with the reference string 0172327103 if the four frames are initially empty? Now repeat this problem for LRU.

29. Consider the page sequence of Fig. 3-15(b). Suppose that the *R* bits for the pages *B* through *A* are 11011011, respectively. Which page will second chance remove?

30. A small computer on a smart card has four page frames. At the first clock tick, the *R* bits are 0111 (page 0 is 0, the rest are 1). At subsequent clock ticks, the values are 1011, 1010, 1101, 0010, 1010, 1100, and 0001. If the aging algorithm is used with an 8-bit counter, give the values of the four counters after the last tick.

31. Give a simple example of a page reference sequence where the first page selected for replacement will be different for the clock and LRU page replacement algorithms. Assume that a process is allocated 3=three frames, and the reference string contains page numbers from the set 0, 1, 2, 3.

32. In the WSClock algorithm of Fig. 3-20(c), the hand points to a page with $R = 0$. If $\tau = 400$, will this page be removed? What about if $\tau = 1000$?

33. Suppose that the WSClock page replacement algorithm uses a τ of two ticks, and the system state is the following:

Page	Time stamp	V	R	M
0	6	1	0	1
1	9	1	1	0
2	9	1	1	1
3	7	1	0	0
4	4	0	0	0

where the three flag bits *V*, R, and *M* stand for Valid, Referenced, and Modified, respectively.

(a) If a clock interrupt occurs at tick 10, show the contents of the new table entries. Explain. (You can omit entries that are unchanged.)
(b) Suppose that instead of a clock interrupt, a page fault occurs at tick 10 due to a read request to page 4. Show the contents of the new table entries. Explain. (You can omit entries that are unchanged.)

34. A student has claimed that "in the abstract, the basic page replacement algorithms (FIFO, LRU, optimal) are identical except for the attribute used for selecting the page to be replaced."

(a) What is that attribute for the FIFO algorithm? LRU algorithm? Optimal algorithm?
(b) Give the generic algorithm for these page replacement algorithms.

35. How long does it take to load a 64-KB program from a disk whose average seek time is 5 msec, whose rotation time is 5 msec, and whose tracks hold 1 MB

(a) for a 2-KB page size?
(b) for a 4-KB page size?

The pages are spread randomly around the disk and the number of cylinders is so large that the chance of two pages being on the same cylinder is negligible.

36. A computer has four page frames. The time of loading, time of last access, and the *R* and *M* bits for each page are as shown below (the times are in clock ticks):

Page	Loaded	Last ref.	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

(a) Which page will NRU replace?
(b) Which page will FIFO replace?
(c) Which page will LRU replace?
(d) Which page will second chance replace?

37. Suppose that two processes *A* and *B* share a page that is not in memory. If process *A* faults on the shared page, the page table entry for process *A* must be updated once the page is read into memory.

(a) Under what conditions should the page table update for process *B* be delayed even though the handling of process *A*'s page fault will bring the shared page into memory? Explain.
(b) What is the potential cost of delaying the page table update?

38. Consider the following two-dimensional array:

```
int   X[64][64];
```

Suppose that a system has four page frames and each frame is 128 words (an integer occupies one word). Programs that manipulate the *X* array fit into exactly one page and always occupy page 0. The data are swapped in and out of the other three frames. The *X* array is stored in row-major order (i.e., *X*[0][1] follows *X*[0][0] in memory). Which of the two code fragments shown below will generate the lowest number of page faults? Explain and compute the total number of page faults.

Fragment A
```
for (int j = 0; j < 64; j++)
   for (int i = 0; i < 64; i++) X[i][j] = 0;
```

Fragment B
```
for (int i = 0; i < 64; i++)
   for (int j = 0; j < 64; j++) X[i][j] = 0;
```

39. You have been hired by a cloud computing company that deploys thousands of servers at each of its data centers. They have recently heard that it would be worthwhile to handle a page fault at server A by reading the page from the RAM memory of some other server rather than its local disk drive.

(a) How could that be done?
(b) Under what conditions would the approach be worthwhile? Be feasible?

40. One of the first timesharing machines, the DEC PDP-1, had a (core) memory of 4K 18-bit words. It held one process at a time in its memory. When the scheduler decided to run another process, the process in memory was written to a paging drum, with 4K 18-bit words around the circumference of the drum. The drum could start writing (or reading) at any word, rather than only at word 0. Why do you suppose this drum was chosen?

41. A computer provides each process with 65,536 bytes of address space divided into pages of 4096 bytes each. A particular program has a text size of 32,768 bytes, a data size of 16,386 bytes, and a stack size of 15,870 bytes. Will this program fit in the machine's address space? Suppose that instead of 4096 bytes, the page size were 512 bytes, would it then fit? Each page must contain either text, data, or stack, not a mixture of two or three of them.

42. It has been observed that the number of instructions executed between page faults is directly proportional to the number of page frames allocated to a program. If the available memory is doubled, the mean interval between page faults is also doubled. Suppose that a normal instruction takes 1 microsec, but if a page fault occurs, it takes 2001 μsec (i.e., 2 msec) to handle the fault. If a program takes 60 sec to run, during which time it gets 15,000 page faults, how long would it take to run if twice as much memory were available?

43. A group of operating system designers for the Frugal Computer Company are thinking about ways to reduce the amount of backing store needed in their new operating system. The head guru has just suggested not bothering to save the program text in the swap area at all, but just page it in directly from the binary file whenever it is needed. Under what conditions, if any, does this idea work for the program text? Under what conditions, if any, does it work for the data?

44. A machine-language instruction to load a 32-bit word into a register contains the 32-bit address of the word to be loaded. What is the maximum number of page faults this instruction can cause?

45. Explain the difference between internal fragmentation and external fragmentation. Which one occurs in paging systems? Which one occurs in systems using pure segmentation?

46. When segmentation and paging are both being used, as in MULTICS, first the segment descriptor must be looked up, then the page descriptor. Does the TLB also work this way, with two levels of lookup?

47. We consider a program which has the two segments shown below consisting of instructions in segment 0, and read/write data in segment 1. Segment 0 has read/execute protection, and segment 1 has just read/write protection. The memory system is a demand-paged virtual memory system with virtual addresses that have a 4-bit page number, and a 10-bit offset. The page tables and protection are as follows (all numbers in the table are in decimal):

Segment 0		Segment 1	
Read/Execute		Read/Write	
Virtual Page #	Page frame #	Virtual Page #	Page frame #
0	2	0	On Disk
1	On Disk	1	14
2	11	2	9
3	5	3	6
4	On Disk	4	On Disk
5	On Disk	5	13
6	4	6	8
7	3	7	12

For each of the following cases, either give the real (actual) memory address which results from dynamic address translation or identify the type of fault which occurs (either page or protection fault).

(a) Fetch from segment 1, page 1, offset 3
(b) Store into segment 0, page 0, offset 16
(c) Fetch from segment 1, page 4, offset 28
(d) Jump to location in segment 1, page 3, offset 32

48. Can you think of any situations where supporting virtual memory would be a bad idea, and what would be gained by not having to support virtual memory? Explain.

49. Virtual memory provides a mechanism for isolating one process from another. What memory management difficulties would be involved in allowing two operating systems to run concurrently? How might these difficulties be addressed?

50. Plot a histogram and calculate the mean and median of the sizes of executable binary files on a computer to which you have access. On a Windows system, look at all .exe and .dll files; on a UNIX system look at all executable files in */bin*, */usr/bin*, and */local/bin* that are not scripts (or use the *file* utility to find all executables). Determine the optimal page size for this computer just considering the code (not data). Consider internal fragmentation and page table size, making some reasonable assumption about the size of a page table entry. Assume that all programs are equally likely to be run and thus should be weighted equally.

51. Write a program that simulates a paging system using the aging algorithm. The number of page frames is a parameter. The sequence of page references should be read from a file. For a given input file, plot the number of page faults per 1000 memory references as a function of the number of page frames available.

52. Write a program that simulates a toy paging system that uses the WSClock algorithm. The system is a toy in that we will assume there are no write references (not very realistic), and process termination and creation are ignored (eternal life). The inputs will be:

- The reclamation age threshhold
- The clock interrupt interval expressed as number of memory references
- A file containing the sequence of page references

(a) Describe the basic data structures and algorithms in your implementation.
(b) Show that your simulation behaves as expected for a simple (but nontrivial) input example.
(c) Plot the number of page faults and working set size per 1000 memory references.
(d) Explain what is needed to extend the program to handle a page reference stream that also includes writes.

53. Write a program that demonstrates the effect of TLB misses on the effective memory access time by measuring the per-access time it takes to stride through a large array.

(a) Explain the main concepts behind the program, and describe what you expect the output to show for some practical virtual memory architecture.
(b) Run the program on some computer and explain how well the data fit your expectations.
(c) Repeat part (b) but for an older computer with a different architecture and explain any major differences in the output.

54. Write a program that will demonstrate the difference between using a local page replacement policy and a global one for the simple case of two processes. You will need a routine that can generate a page reference string based on a statistical model. This model has N states numbered from 0 to $N - 1$ representing each of the possible page references and a probability p_i associated with each state i representing the chance that the next reference is to the same page. Otherwise, the next page reference will be one of the other pages with equal probability.

(a) Demonstrate that the page reference string-generation routine behaves properly for some small N.
(b) Compute the page fault rate for a small example in which there is one process and a fixed number of page frames. Explain why the behavior is correct.
(c) Repeat part (b) with two processes with independent page reference sequences and twice as many page frames as in part (b).
(d) Repeat part (c) but using a global policy instead of a local one. Also, contrast the per-process page fault rate with that of the local policy approach.

55. Write a program that can be used to compare the effectiveness of adding a tag field to TLB entries when control is toggled between two programs. The tag field is used to effectively label each entry with the process id. Note that a nontagged TLB can be simulated by requiring that all TLB entries have the same tag at any one time. The inputs will be:

- The number of TLB entries available
- The clock interrupt interval expressed as number of memory references
- A file containing a sequence of (process, page references) entries
- The cost to update one TLB entry

(a) Describe the basic data structures and algorithms in your implementation.
b) Show that your simulation behaves as expected for a simple (but nontrivial) input example.
(c) Plot the number of TLB updates per 1000 references.

4

FILE SYSTEMS

All computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information within its own address space. However, the storage capacity is restricted to the size of the virtual address space. For some applications this size is adequate, but for others, such as airline reservations, banking, or corporate record keeping, it is far too small.

A second problem with keeping information within a process' address space is that when the process terminates, the information is lost. For many applications (e.g., for databases), the information must be retained for weeks, months, or even forever. Having it vanish when the process using it terminates is unacceptable. Furthermore, it must not go away when a computer crash kills the process.

A third problem is that it is frequently necessary for multiple processes to access (parts of) the information at the same time. If we have an online telephone directory stored inside the address space of a single process, only that process can access it. The way to solve this problem is to make the information itself independent of any one process.

Thus, we have three essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. The information must survive the termination of the process using it.
3. Multiple processes must be able to access the information at once.

Magnetic disks have been used for years for this long-term storage. In recent years, solid-state drives have become increasingly popular, as they do not have any moving parts that may break. Also, they offer fast random access. Tapes and optical disks have also been used extensively, but they have much lower performance and are typically used for backups. We will study disks more in Chap. 5, but for the moment, it is sufficient to think of a disk as a linear sequence of fixed-size blocks and supporting two operations:

1. Read block *k*.
2. Write block *k*

In reality there are more, but with these two operations one could, in principle, solve the long-term storage problem.

However, these are very inconvenient operations, especially on large systems used by many applications and possibly multiple users (e.g., on a server). Just a few of the questions that quickly arise are:

1. How do you find information?
2. How do you keep one user from reading another user's data?
3. How do you know which blocks are free?

and there are many more.

Just as we saw how the operating system abstracted away the concept of the processor to create the abstraction of a process and how it abstracted away the concept of physical memory to offer processes (virtual) address spaces, we can solve this problem with a new abstraction: the file. Together, the abstractions of processes (and threads), address spaces, and files are the most important concepts relating to operating systems. If you really understand these three concepts from beginning to end, you are well on your way to becoming an operating systems expert.

Files are logical units of information created by processes. A disk will usually contain thousands or even millions of them, each one independent of the others. In fact, if you think of each file as a kind of address space, you are not that far off, except that they are used to model the disk instead of modeling the RAM.

Processes can read existing files and create new ones if need be. Information stored in files must be **persistent**, that is, not be affected by process creation and termination. A file should disappear only when its owner explicitly removes it. Although operations for reading and writing files are the most common ones, there exist many others, some of which we will examine below.

Files are managed by the operating system. How they are structured, named, accessed, used, protected, implemented, and managed are major topics in operating system design. As a whole, that part of the operating system dealing with files is known as the **file system** and is the subject of this chapter.

From the user's standpoint, the most important aspect of a file system is how it appears, in other words, what constitutes a file, how files are named and protected, what operations are allowed on files, and so on. The details of whether linked lists or bitmaps are used to keep track of free storage and how many sectors there are in a logical disk block are of no interest, although they are of great importance to the designers of the file system. For this reason, we have structured the chapter as several sections. The first two are concerned with the user interface to files and directories, respectively. Then comes a detailed discussion of how the file system is implemented and managed. Finally, we give some examples of real file systems.

4.1 FILES

In the following pages we will look at files from the user's point of view, that is, how they are used and what properties they have.

4.1.1 File Naming

A file is an abstraction mechanism. It provides a way to store information on the disk and read it back later. This must be done in such a way as to shield the user from the details of how and where the information is stored, and how the disks actually work.

Probably the most important characteristic of any abstraction mechanism is the way the objects being managed are named, so we will start our examination of file systems with the subject of file naming. When a process creates a file, it gives the file a name. When the process terminates, the file continues to exist and can be accessed by other processes using its name.

The exact rules for file naming vary somewhat from system to system, but all current operating systems allow strings of one to eight letters as legal file names. Thus *andrea*, *bruce*, and *cathy* are possible file names. Frequently digits and special characters are also permitted, so names like *2*, *urgent!*, and *Fig.2-14* are often valid as well. Many file systems support names as long as 255 characters.

Some file systems distinguish between upper- and lowercase letters, whereas others do not. UNIX falls in the first category; the old MS-DOS falls in the second. (As an aside, while ancient, MS-DOS is still very widely used in embedded systems, so it is by no means obsolete.) Thus, a UNIX system can have all of the following as three distinct files: *maria*, *Maria*, and *MARIA*. In MS-DOS, all these names refer to the same file.

An aside on file systems is probably in order here. Windows 95 and Windows 98 both used the MS-DOS file system, called **FAT-16**, and thus inherit many of its properties, such as how file names are constructed. Windows 98 introduced some extensions to FAT-16, leading to **FAT-32**, but these two are quite similar. In addition, Windows NT, Windows 2000, Windows XP, Windows Vista, Windows 7, and Windows 8 all still support both FAT file systems, which are really obsolete now. However, these newer operating systems also have a much more advanced native file system (**NTFS**) that has different properties (such as file names in Unicode). In fact, there is second file system for Windows 8, known as **ReFS** (or **Resilient File System**), but it is targeted at the server version of Windows 8. In this chapter, when we refer to the MS-DOS or FAT file systems, we mean FAT-16 and FAT-32 as used on Windows unless specified otherwise. We will discuss the FAT file systems later in this chapter and NTFS in Chap. 12, where we will examine Windows 8 in detail. Incidentally, there is also a new FAT-like file system, known as **exFAT** file system, a Microsoft extension to FAT-32 that is optimized for flash drives and large file systems. Exfat is the only modern Microsoft file system that OS X can both read and write.

Many operating systems support two-part file names, with the two parts separated by a period, as in *prog.c*. The part following the period is called the **file extension** and usually indicates something about the file. In MS-DOS, for example, file names are 1 to 8 characters, plus an optional extension of 1 to 3 characters. In UNIX, the size of the extension, if any, is up to the user, and a file may even have two or more extensions, as in *homepage.html.zip*, where *.html* indicates a Web page in HTML and *.zip* indicates that the file (*homepage.html*) has been compressed using the *zip* program. Some of the more common file extensions and their meanings are shown in Fig. 4-1.

Extension	Meaning
.bak	Backup file
.c	C source program
.gif	Compuserve Graphical Interchange Format image
.hlp	Help file
.html	World Wide Web HyperText Markup Language document
.jpg	Still picture encoded with the JPEG standard
.mp3	Music encoded in MPEG layer 3 audio format
.mpg	Movie encoded with the MPEG standard
.o	Object file (compiler output, not yet linked)
.pdf	Portable Document Format file
.ps	PostScript file
.tex	Input for the TEX formatting program
.txt	General text file
.zip	Compressed archive

Figure 4-1. Some typical file extensions.

In some systems (e.g., all flavors of UNIX) file extensions are just conventions and are not enforced by the operating system. A file named *file.txt* might be some kind of text file, but that name is more to remind the owner than to convey any actual information to the computer. On the other hand, a C compiler may actually insist that files it is to compile end in *.c*, and it may refuse to compile them if they do not. However, the operating system does not care.

Conventions like this are especially useful when the same program can handle several different kinds of files. The C compiler, for example, can be given a list of several files to compile and link together, some of them C files and some of them assembly-language files. The extension then becomes essential for the compiler to tell which are C files, which are assembly files, and which are other files.

In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system and specify for each one which program "owns" that extension. When a user double clicks on a file name, the program assigned to its file extension is launched with the file as parameter. For example, double clicking on *file.docx* starts Microsoft *Word* with *file.docx* as the initial file to edit.

4.1.2 File Structure

Files can be structured in any of several ways. Three common possibilities are depicted in Fig. 4-2. The file in Fig. 4-2(a) is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows use this approach.

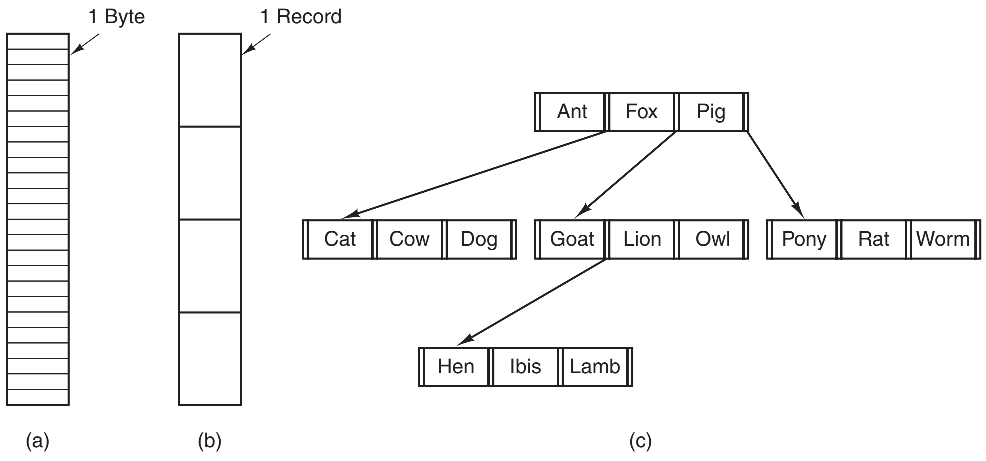


Figure 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Having the operating system regard files as nothing more than byte sequences provides the maximum amount of flexibility. User programs can put anything they want in their files and name them any way that they find convenient. The operating system does not help, but it also does not get in the way. For users who want to do unusual things, the latter can be very important. All versions of UNIX (including Linux and OS X) and Windows use this file model.

The first step up in structure isillustrated in Fig. 4-2(b). In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record. As a historical note, in decades gone by, when the 80-column punched card was king of the mountain, many (mainframe) operating systems based their file systems on files consisting of 80-character records, in effect, card images. These systems also supported files of 132-character records, which were intended for the line printer (which in those days were big chain printers having 132 columns). Programs read input in units of 80 characters and wrote it in units of 132 characters, although the final 52 could be spaces, of course. No current general-purpose system uses this model as its primary file system any more, but back in the days of 80-column punched cards and 132-character line printer paper this was a common model on mainframe computers.

The third kind of file structure is shown in Fig. 4-2(c). In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a **key** field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

The basic operation here is not to get the "next" record, although that is also possible, but to get the record with a specific key. For the zoo file of Fig. 4-2(c), one could ask the system to get the record whose key is *pony*, for example, without worrying about its exact position in the file. Furthermore, new records can be added to the file, with the operating system, and not the user, deciding where to place them. This type of file is clearly quite different from the unstructured byte streams used in UNIX and Windows and is used on some large mainframe computers for commercial data processing.

4.1.3 File Types

Many operating systems support several types of files. UNIX (again, including OS X) and Windows, for example, have regular files and directories. UNIX also has character and block special files. **Regular files** are the ones that contain user information. All the files of Fig. 4-2 are regular files. **Directories** are system files for maintaining the structure of the file system. We will study directories below. **Character special files** are related to input/output and used to model serial I/O devices, such as terminals, printers, and networks. **Block special files** are used to model disks. In this chapter we will be primarily interested in regular files.

Regular files are generally either ASCII files or binary files. ASCII files consist of lines of text. In some systems each line is terminated by a carriage return character. In others, the line feed character is used. Some systems (e.g., Windows) use both. Lines need not all be of the same length.

The great advantage of ASCII files is that they can be displayed and printed as is, and they can be edited with any text editor. Furthermore, if large numbers of programs use ASCII files for input and output, it is easy to connect the output of one program to the input of another, as in shell pipelines. (The interprocess plumbing is not any easier, but interpreting the information certainly is if a standard convention, such as ASCII, is used for expressing it.)

Other files are binary, which just means that they are not ASCII files. Listing them on the printer gives an incomprehensible listing full of random junk. Usually, they have some internal structure known to programs that use them.

For example, in Fig. 4-3(a) we see a simple executable binary file taken from an early version of UNIX. Although technically the file is just a sequence of bytes, the operating system will execute a file only if it has the proper format. It has five sections: header, text, data, relocation bits, and symbol table. The header starts with a so-called **magic number**, identifying the file as an executable file (to prevent the accidental execution of a file not in this format). Then come the sizes of the various pieces of the file, the address at which execution starts, and some flag bits. Following the header are the text and data of the program itself. These are loaded into memory and relocated using the relocation bits. The symbol table is used for debugging.

Our second example of a binary file is an archive, also from UNIX. It consists of a collection of library procedures (modules) compiled but not linked. Each one is prefaced by a header telling its name, creation date, owner, protection code, and size. Just as with the executable file, the module headers are full of binary numbers. Copying them to the printer would produce complete gibberish.

Every operating system must recognize at least one file type: its own executable file; some recognize more. The old TOPS-20 system (for the DECsystem 20) went so far as to examine the creation time of any file to be executed. Then it located the source file and saw whether the source had been modified since the binary was made. If it had been, it automatically recompiled the source. In UNIX terms, the *make* program had been built into the shell. The file extensions were mandatory, so it could tell which binary program was derived from which source.

Having strongly typed files like this causes problems whenever the user does anything that the system designers did not expect. Consider, as an example, a system in which program output files have extension *.dat* (data files). If a user writes a program formatter that reads a *.c* file (C program), transforms it (e.g., by converting it to a standard indentation layout), and then writes the transformed file as output, the output file will be of type *.dat*. If the user tries to offer this to the C compiler to compile it, the system will refuse because it has the wrong extension. Attempts to copy *file.dat* to *file.c* will be rejected by the system as invalid (to protect the user against mistakes).

While this kind of "user friendliness" may help novices, it drives experienced users up the wall since they have to devote considerable effort to circumventing the operating system's idea of what is reasonable and what is not.

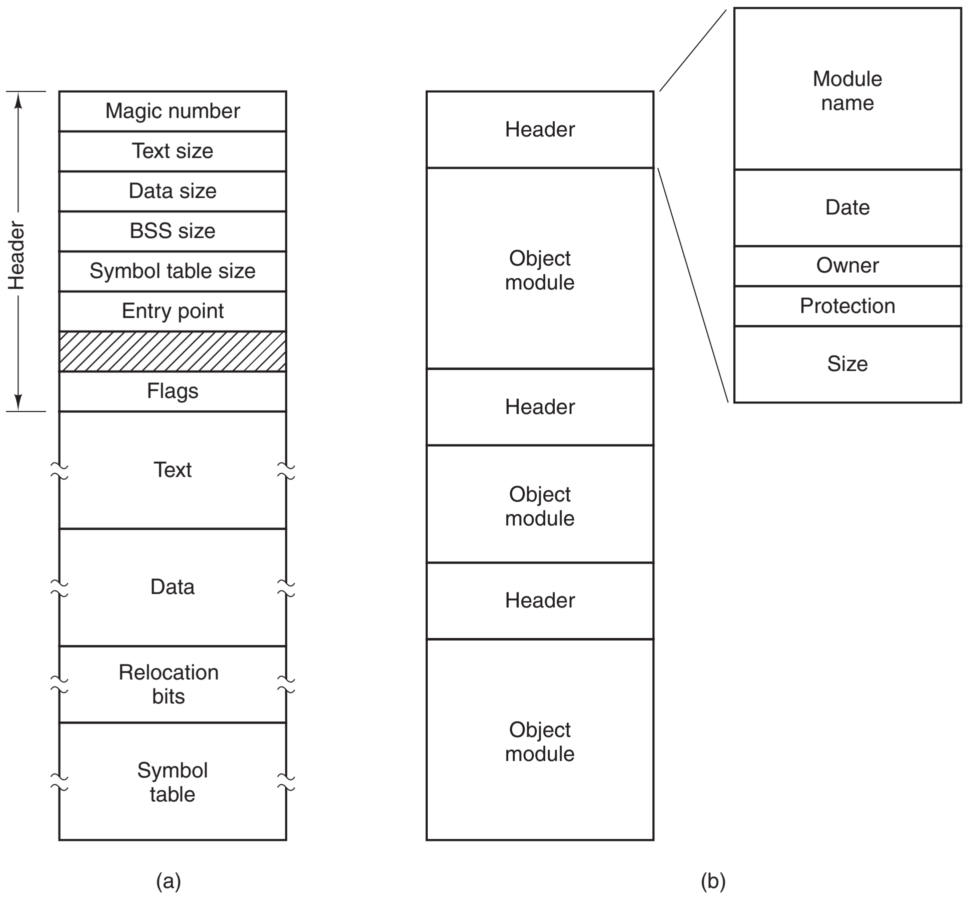


Figure 4-3. (a) An executable file. (b) An archive.

4.1.4 File Access

Early operating systems provided only one kind of file access: **sequential access**. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order. Sequential files could be rewound, however, so they could be read as often as needed. Sequential files were convenient when the storage medium was magnetic tape rather than disk.

When disks came into use for storing files, it became possible to read the bytes or records of a file out of order, or to access records by key rather than by position. Files whose bytes or records can be read in any order are called **random-access files**. They are required by many applications.

Random access files are essential for many applications, for example, database systems. If an airline customer calls up and wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight without having to read the records for thousands of other flights first.

Two methods can be used for specifying where to start reading. In the first one, every read operation gives the position in the file to start reading at. In the second one, a special operation, seek, is provided to set the current position. After a seek, the file can be read sequentially from the now-current position. The latter method is used in UNIX and Windows.

4.1.5 File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was last modified and the file's size. We will call these extra items the file's **attributes**. Some people call them **metadata**. The list of attributes varies considerably from system to system. The table of Fig. 4-4 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.

The first four attributes relate to the file's protection and tell who may access it and who may not. All kinds of schemes are possible, some of which we will study later. In some systems the user must present a password to access a file, in which case the password must be one of the attributes.

The flags are bits or short fields that control or enable some specific property. Hidden files, for example, do not appear in listings of all the files. The archive flag is a bit that keeps track of whether the file has been backed up recently. The backup program clears it, and the operating system sets it whenever a file is changed. In this way, the backup program can tell which files need backing up. The temporary flag allows a file to be marked for automatic deletion when the process that created it terminates.

The record-length, key-position, and key-length fields are only present in files whose records can be looked up using a key. They provide the information required to find the keys.

The various times keep track of when the file was created, most recently accessed, and most recently modified. These are useful for a variety of purposes. For example, a source file that has been modified after the creation of the corresponding object file needs to be recompiled. These fields provide the necessary information.

The current size tells how big the file is at present. Some old mainframe operating systems required the maximum size to be specified when the file was created, in order to let the operating system reserve the maximum amount of storage in advance. Workstation and personal-computer operating systems are thankfully clever enough to do without this feature nowadays.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 4-4. Some possible file attributes.

4.1.6 File Operations

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Below is a discussion of the most common system calls relating to files.

1. Create. The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.

2. Delete. When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.

3. Open. Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.

4. Close. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a

maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.

5. Read. Data are read from file. Usually, the bytes come from the current position. The caller must specify how many data are needed and must also provide a buffer to put them in.

6. Write. Data are written to the file again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.

7. Append. This call is a restricted form of write. It can add data only to the end of the file. Systems that provide a minimal set of system calls rarely have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.

8. Seek. For random-access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.

9. Get attributes. Processes often need to read file attributes to do their work. For example, the UNIX *make* program is commonly used to manage software development projects consisting of many source files. When *make* is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

10. Set attributes. Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection-mode information is an obvious example. Most of the flags also fall in this category.

11. Rename. It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

4.1.7 An Example Program Using File-System Calls

In this section we will examine a simple UNIX program that copies one file from its source file to a destination file. It is listed in Fig. 4-5. The program has minimal functionality and even worse error reporting, but it gives a reasonable idea of how some of the system calls related to files work.

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                          /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);               /* ANSI prototype */

#define BUF_SIZE 4096                           /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                        /* protection bits for output file */

int main(int argc, char *argv[])
{
     int in_fd, out_fd, rd_count, wt_count;
     char buffer[BUF_SIZE];

     if (argc != 3) exit(1);                    /* syntax error if argc is not 3 */

     /* Open the input file and create the output file */
     in_fd = open(argv[1], O_RDONLY);           /* open the source file */
     if (in_fd < 0) exit(2);                    /* if it cannot be opened, exit */
     out_fd = creat(argv[2], OUTPUT_MODE);      /* create the destination file */
     if (out_fd < 0) exit(3);                   /* if it cannot be created, exit */

     /* Copy loop */
     while (TRUE) {
          rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
          if (rd_count <= 0) break;             /* if end of file or error, exit loop */
          wt_count = write(out_fd, buffer, rd_count); /* write data */
          if (wt_count <= 0) exit(4);           /* wt_count <= 0 is an error */
     }

     /* Close the files */
     close(in_fd);
     close(out_fd);
     if (rd_count == 0)                         /* no error on last read */
          exit(0);
     else
          exit(5);                              /* error on last read */
}
```

Figure 4-5. A simple program to copy a file.

The program, *copyfile*, can be called, for example, by the command line

```
copyfile abc xyz
```

to copy the file *abc* to *xyz*. If *xyz* already exists, it will be overwritten. Otherwise, it will be created. The program must be called with exactly two arguments, both legal file names. The first is the source; the second is the output file.

The four *#include* statements near the top of the program cause a large number of definitions and function prototypes to be included in the program. These are needed to make the program conformant to the relevant international standards, but will not concern us further. The next line is a function prototype for *main*, something required by ANSI C, but also not important for our purposes.

The first *#define* statement is a macro definition that defines the character string *BUF_SIZE* as a macro that expands into the number 4096. The program will read and write in chunks of 4096 bytes. It is considered good programming practice to give names to constants like this and to use the names instead of the constants. Not only does this convention make programs easier to read, but it also makes them easier to maintain. The second *#define* statement determines who can access the output file.

The main program is called *main*, and it has two arguments, *argc*, and *argv*. These are supplied by the operating system when the program is called. The first one tells how many strings were present on the command line that invoked the program, including the program name. It should be 3. The second one is an array of pointers to the arguments. In the example call given above, the elements of this array would contain pointers to the following values:

```
argv[0] = "copyfile"
argv[1] = "abc"
argv[2] = "xyz"
```

It is via this array that the program accesses its arguments.

Five variables are declared. The first two, *in_fd* and *out_fd*, will hold the **file descriptors**, small integers returned when a file is opened. The next two, *rd_count* and *wt_count*, are the byte counts returned by the read and write system calls, respectively. The last one, *buffer*, is the buffer used to hold the data read and supply the data to be written.

The first actual statement checks *argc* to see if it is 3. If not, it exits with status code 1. Any status code other than 0 means that an error has occurred. The status code is the only error reporting present in this program. A production version would normally print error messages as well.

Then we try to open the source file and create the destination file. If the source file is successfully opened, the system assigns a small integer to *in_fd*, to identify the file. Subsequent calls must include this integer so that the system knows which file it wants. Similarly, if the destination is successfully created, *out_fd* is given a value to identify it. The second argument to *creat* sets the protection mode. If either the open or the create fails, the corresponding file descriptor is set to −1, and the program exits with an error code.

Now comes the copy loop. It starts by trying to read in 4 KB of data to *buffer*. It does this by calling the library procedure *read*, which actually invokes the read system call. The first parameter identifies the file, the second gives the buffer, and the third tells how many bytes to read. The value assigned to *rd_count* gives the number of bytes actually read. Normally, this will be 4096, except if fewer bytes are remaining in the file. When the end of the file has been reached, it will be 0. If *rd_count* is ever zero or negative, the copying cannot continue, so the *break* statement is executed to terminate the (otherwise endless) loop.

The call to *write* outputs the buffer to the destination file. The first parameter identifies the file, the second gives the buffer, and the third tells how many bytes to write, analogous to *read*. Note that the byte count is the number of bytes actually read, not *BUF_SIZE*. This point is important because the last *read* will not return 4096 unless the file just happens to be a multiple of 4 KB.

When the entire file has been processed, the first call beyond the end of file will return 0 to *rd_count*, which will make it exit the loop. At this point the two files are closed and the program exits with a status indicating normal termination.

Although the Windows system calls are different from those of UNIX, the general structure of a command-line Windows program to copy a file is moderately similar to that of Fig. 4-5. We will examine the Windows 8 calls in Chap. 11.

4.2 DIRECTORIES

To keep track of files, file systems normally have **directories** or **folders**, which are themselves files. In this section we will discuss directories, their organization, their properties, and the operations that can be performed on them.

4.2.1 Single-Level Directory Systems

The simplest form of directory system is having one directory containing all the files. Sometimes it is called the **root directory**, but since it is the only one, the name does not matter much. On early personal computers, this system was common, in part because there was only one user. Interestingly enough, the world's first supercomputer, the CDC 6600, also had only a single directory for all files, even though it was used by many users at once. This decision was no doubt made to keep the software design simple.

An example of a system with one directory is given in Fig. 4-6. Here the directory contains four files. The advantages of this scheme are its simplicity and the ability to locate files quickly—there is only one place to look, after all. It is sometimes still used on simple embedded devices such as digital cameras and some portable music players.

4.2.2 Hierarchical Directory Systems

The single level is adequate for very simple dedicated applications (and was even used on the first personal computers), but for modern users with thousands of files, it would be impossible to find anything if all files were in a single directory.

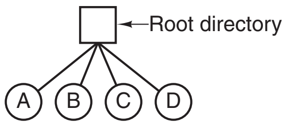


Figure 4-6. A single-level directory system containing four files.

Consequently, a way is needed to group related files together. A professor, for example, might have a collection of files that together form a book that he is writing, a second collection containing student programs submitted for another course, a third group containing the code of an advanced compiler-writing system he is building, a fourth group containing grant proposals, as well as other files for electronic mail, minutes of meetings, papers he is writing, games, and so on.

What is needed is a hierarchy (i.e., a tree of directories). With this approach, there can be as many directories as are needed to group the files in natural ways. Furthermore, if multiple users share a common file server, as is the case on many company networks, each user can have a private root directory for his or her own hierarchy. This approach is shown in Fig. 4-7. Here, the directories *A*, *B*, and *C* contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

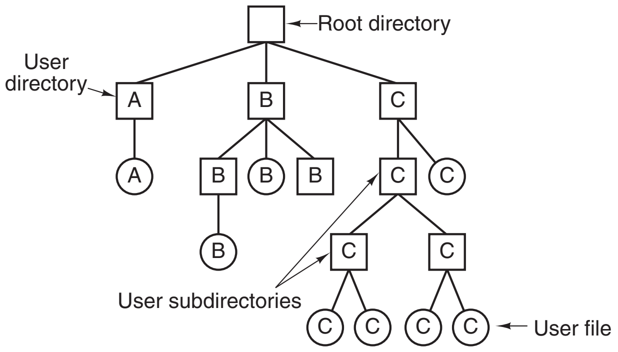


Figure 4-7. A hierarchical directory system.

The ability for users to create an arbitrary number of subdirectories provides a powerful structuring tool for users to organize their work. For this reason, nearly all modern file systems are organized in this manner.

4.2.3 Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file is given an **absolute path name** consisting of the path from the root directory to the file. As an example, the path */usr/ast/mailbox* means that the root directory contains a subdirectory *usr*, which in turn contains a subdirectory *ast*, which contains the file *mailbox*. Absolute path names always start at the root directory and are unique. In UNIX the components of the path are separated by /. In Windows the separator is \ . In MULTICS it was >. Thus, the same path name would be written as follows in these three systems:

```
Windows     \usr\ast\mailbox
UNIX        /usr/ast/mailbox
MULTICS     >usr>ast>mailbox
```

No matter which character is used, if the first character of the path name is the separator, then the path is absolute.

The other kind of name is the **relative path name**. This is used in conjunction with the concept of the **working directory** (also called the **current directory**). A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For example, if the current working directory is */usr/ast*, then the file whose absolute path is */usr/ast/mailbox* can be referenced simply as *mailbox*. In other words, the UNIX command

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

and the command

```
cp mailbox mailbox.bak
```

do exactly the same thing if the working directory is */usr/ast*. The relative form is often more convenient, but it does the same thing as the absolute form.

Some programs need to access a specific file without regard to what the working directory is. In that case, they should always use absolute path names. For example, a spelling checker might need to read */usr/lib/dictionary* to do its work. It should use the full, absolute path name in this case because it does not know what the working directory will be when it is called. The absolute path name will always work, no matter what the working directory is.

Of course, if the spelling checker needs a large number of files from */usr/lib*, an alternative approach is for it to issue a system call to change its working directory to */usr/lib*, and then use just *dictionary* as the first parameter to open. By explicitly changing the working directory, it knows for sure where it is in the directory tree, so it can then use relative paths.

Each process has its own working directory, so when it changes its working directory and later exits, no other processes are affected and no traces of the change are left behind in the file system. In this way, it is always perfectly safe for a process to change its working directory whenever it finds that to be convenient. On the other hand, if a *library procedure* changes the working directory and does not change back to where it was when it is finished, the rest of the program may not work since its assumption about where it is may now suddenly be invalid. For this reason, library procedures rarely change the working directory, and when they must, they always change it back again before returning.

Most operating systems that support a hierarchical directory system have two special entries in every directory, "." and "..", generally pronounced "dot" and "dotdot." Dot refers to the current directory; dotdot refers to its parent (except in the root directory, where it refers to itself). To see how these are used, consider the UNIX file tree of Fig. 4-8. A certain process has */usr/ast* as its working directory. It can use .. to go higher up the tree. For example, it can copy the file */usr/lib/dictionary* to its own directory using the command

```
cp ../lib/dictionary .
```

The first path instructs the system to go upward (to the *usr* directory), then to go down to the directory *lib* to find the file *dictionary*.

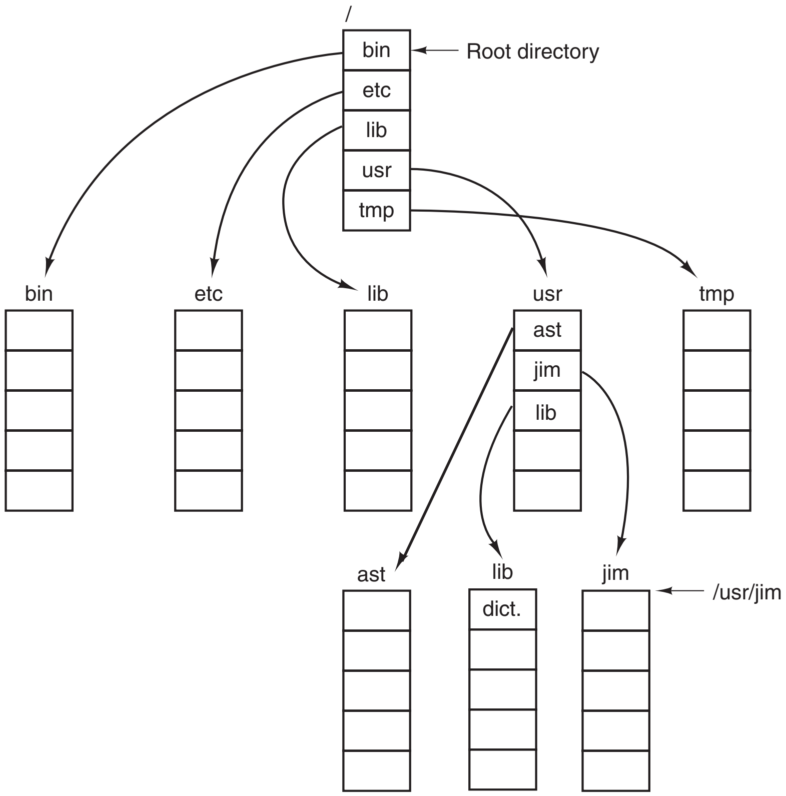


Figure 4-8. A UNIX directory tree.

The second argument (dot) names the current directory. When the *cp* command gets a directory name (including dot) as its last argument, it copies all the files to that directory. Of course, a more normal way to do the copy would be to use the full absolute path name of the source file:

```
cp /usr/lib/dictionary .
```

Here the use of dot saves the user the trouble of typing *dictionary* a second time. Nevertheless, typing

```
cp /usr/lib/dictionary dictionary
```

also works fine, as does

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

All of these do exactly the same thing.

4.2.4 Directory Operations

The allowed system calls for managing directories exhibit more variation from system to system than system calls for files. To give an impression of what they are and how they work, we will give a sample (taken from UNIX).

1. Create. A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system (or in a few cases, by the *mkdir* program).
2. Delete. A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot be deleted.
3. Opendir. Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.
4. Closedir. When a directory has been read, it should be closed to free up internal table space.
5. Readdir. This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.
6. Rename. In many respects, directories are just like files and can be renamed the same way files can be.
7. Link. Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path

name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. A link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a **hard link**.

8. Unlink. A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, unlink.

The above list gives the most important calls, but there are a few others as well, for example, for managing the protection information associated with a directory.

A variant on the idea of linking files is the **symbolic link**. Instead, of having two names point to the same internal data structure representing a file, a name can be created that points to a tiny file naming another file. When the first file is used, for example, opened, the file system follows the path and finds the name at the end. Then it starts the lookup process all over using the new name. Symbolic links have the advantage that they can cross disk boundaries and even name files on remote computers. Their implementation is somewhat less efficient than hard links though.

4.3 FILE-SYSTEM IMPLEMENTATION

Now it is time to turn from the user's view of the file system to the implementor's view. Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like, and similar interface issues. Implementors are interested in how files and directories are stored, how disk space is managed, and how to make everything work efficiently and reliably. In the following sections we will examine a number of these areas to see what the issues and trade-offs are.

4.3.1 File-System Layout

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. Sector 0 of the disk is called the **MBR** (**Master Boot Record**) and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, which is called the **boot block**, and execute it. The program in the boot block loads the operating system contained in that partition. For uniformity, every partition starts with a boot block, even if it does not contain a bootable operating system. Besides, it might contain one in the future.

Other than starting with a boot block, the layout of a disk partition varies a lot from file system to file system. Often the file system will contain some of the items shown in Fig. 4-9. The first one is the **superblock**. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched. Typical information in the superblock includes a magic number to identify the file-system type, the number of blocks in the file system, and other key administrative information.

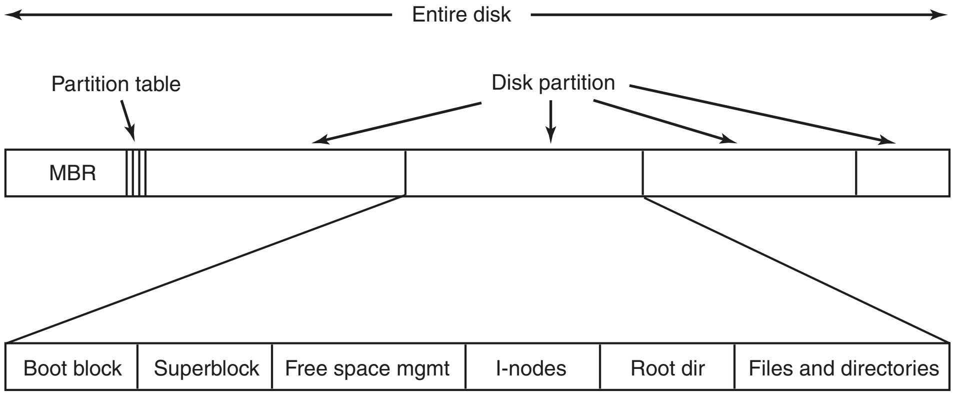


Figure 4-9. A possible file-system layout.

Next might come information about free blocks in the file system, for example in the form of a bitmap or a list of pointers. This might be followed by the i-nodes, an array of data structures, one per file, telling all about the file. After that might come the root directory, which contains the top of the file-system tree. Finally, the remainder of the disk contains all the other directories and files.

4.3.2 Implementing Files

Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems. In this section, we will examine a few of them.

Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous run of disk blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. With 2-KB blocks, it would be allocated 25 consecutive blocks.

We see an example of contiguous storage allocation in Fig. 4-10(a). Here the first 40 disk blocks are shown, starting with block 0 on the left. Initially, the disk was empty. Then a file *A*, of length four blocks, was written to disk starting at the beginning (block 0). After that a six-block file, *B*, was written starting right after the end of file *A*.

Note that each file begins at the start of a new block, so that if file *A* was really 3½ blocks, some space is wasted at the end of the last block. In the figure, a total of seven files are shown, each one starting at the block following the end of the previous one. Shading is used just to make it easier to tell the files apart. It has no actual significance in terms of storage.

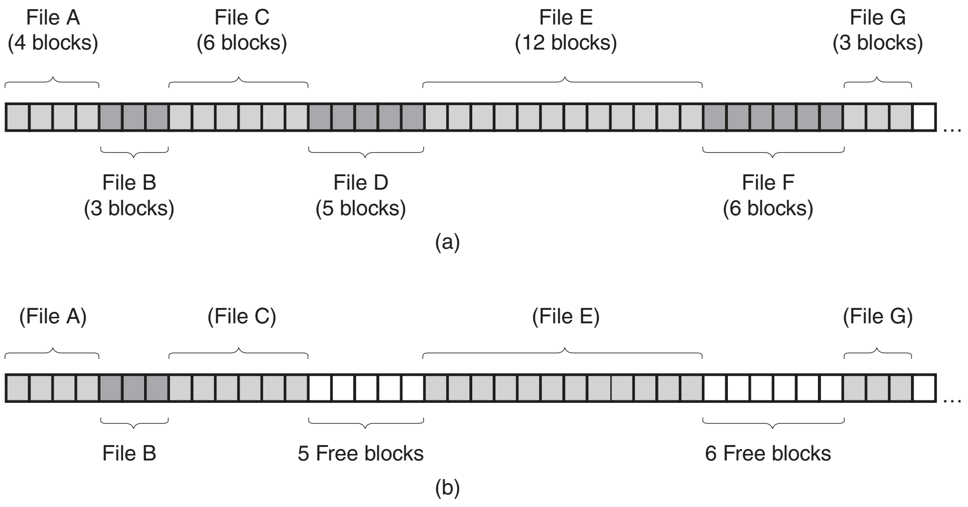


Figure 4-10. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

Contiguous disk-space allocation has two significant advantages. First, it is simple to implement because keeping track of where a file's blocks are is reduced to remembering two numbers: the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.

Second, the read performance is excellent because the entire file can be read from the disk in a single operation. Only one seek is needed (to the first block). After that, no more seeks or rotational delays are needed, so data come in at the full bandwidth of the disk. Thus contiguous allocation is simple to implement and has high performance.

Unfortunately, contiguous allocation also has a very serious drawback: over the course of time, the disk becomes fragmented. To see how this comes about, examine Fig. 4-10(b). Here two files, *D* and F, have been removed. When a file is removed, its blocks are naturally freed, leaving a run of free blocks on the disk. The disk is not compacted on the spot to squeeze out the hole, since that would involve copying all the blocks following the hole, potentially millions of blocks, which would take hours or even days with large disks. As a result, the disk ultimately consists of files and holes, as illustrated in the figure.

Initially, this fragmentation is not a problem, since each new file can be written at the end of disk, following the previous one. However, eventually the disk will fill up and it will become necessary to either compact the disk, which is prohibitively expensive, or to reuse the free space in the holes. Reusing the space requires maintaining a list of holes, which is doable. However, when a new file is to be created, it is necessary to know its final size in order to choose a hole of the correct size to place it in.

Imagine the consequences of such a design. The user starts a word processor in order to create a document. The first thing the program asks is how many bytes the final document will be. The question must be answered or the program will not continue. If the number given ultimately proves too small, the program has to terminate prematurely because the disk hole is full and there is no place to put the rest of the file. If the user tries to avoid this problem by giving an unrealistically large number as the final size, say, 1 GB, the editor may be unable to find such a large hole and announce that the file cannot be created. Of course, the user would be free to start the program again and say 500 MB this time, and so on until a suitable hole was located. Still, this scheme is not likely to lead to happy users.

However, there is one situation in which contiguous allocation is feasible and, in fact, still used: on CD-ROMs. Here all the file sizes are known in advance and will never change during subsequent use of the CD-ROM file system.

The situation with DVDs is a bit more complicated. In principle, a 90-min movie could be encoded as a single file of length about 4.5 GB, but the file system used, **UDF** (**Universal Disk Format**), uses a 30-bit number to represent file length, which limits files to 1 GB. As a consequence, DVD movies are generally stored as three or four 1-GB files, each of which is contiguous. These physical pieces of the single logical file (the movie) are called **extents**.

As we mentioned in Chap. 1, history often repeats itself in computer science as new generations of technology occur. Contiguous allocation was actually used on magnetic-disk file systems years ago due to its simplicity and high performance (user friendliness did not count for much then). Then the idea was dropped due to the nuisance of having to specify final file size at file-creation time. But with the advent of CD-ROMs, DVDs, Blu-rays, and other write-once optical media, suddenly contiguous files were a good idea again. It is thus important to study old systems and ideas that were conceptually clean and simple because they may be applicable to future systems in surprising ways.

Linked-List Allocation

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig. 4-11. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

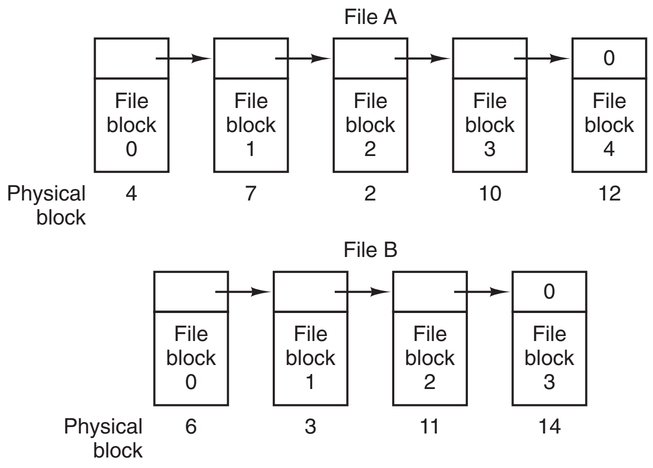


Figure 4-11. Storing a file as a linked list of disk blocks.

Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block). Also, it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there.

On the other hand, although reading a file sequentially is straightforward, random access is extremely slow. To get to block n, the operating system has to start at the beginning and read the $n - 1$ blocks prior to it, one at a time. Clearly, doing so many reads will be painfully slow.

Also, the amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two. With the first few bytes of each block occupied by a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to the copying.

Linked-List Allocation Using a Table in Memory

Both disadvantages of the linked-list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory. Figure 4-12 shows what the table looks like for the example of Fig. 4-11. In both figures, we have two files. File *A* uses disk blocks 4, 7, 2, 10, and 12, in that order, and file *B* uses disk blocks 6, 3, 11, and 14, in that order. Using the table of Fig. 4-12, we can start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6. Both chains are terminated with a special marker (e.g., −1) that is not a valid block number. Such a table in main memory is called a **FAT** (**File Allocation Table**).

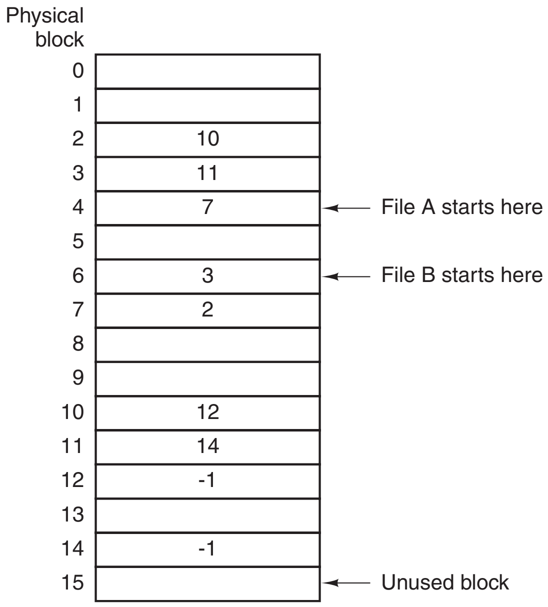


Figure 4-12. Linked-list allocation using a file-allocation table in main memory.

Using this organization, the entire block is available for data. Furthermore, random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is.

The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 1-TB disk and a 1-KB block size, the table needs 1 billion entries, one for each of the 1 billion disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 3 GB or 2.4 GB of main memory all the time, depending on whether the system is optimized for space or time. Not wildly practical. Clearly the FAT idea does not scale well to large disks. It was the original MS-DOS file system and is still fully supported by all versions of Windows though.

I-nodes

Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an **i-node** (**index-node**), which lists the attributes and disk addresses of the file's blocks. A simple example is depicted in Fig. 4-13. Given the i-node, it is then possible to find all the blocks of the file.

The big advantage of this scheme over linked files using an in-memory table is that the i-node need be in memory only when the corresponding file is open. If each i-node occupies *n* bytes and a maximum of *k* files may be open at once, the total memory occupied by the array holding the i-nodes for the open files is only *kn* bytes. Only this much space need be reserved in advance.

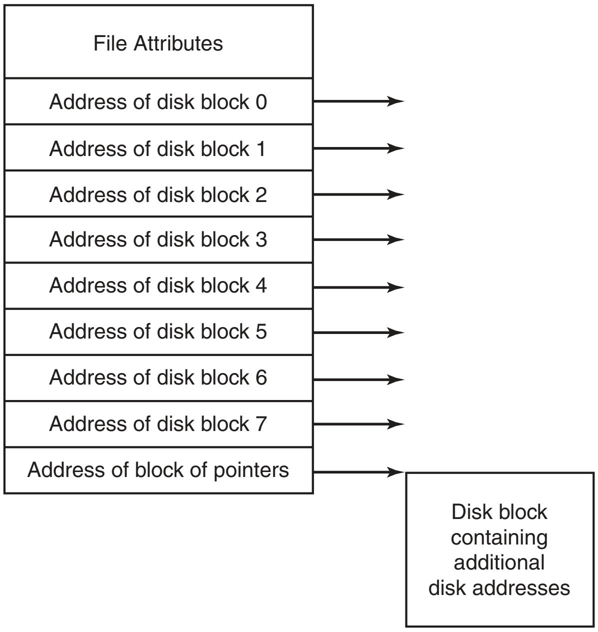


Figure 4-13. An example i-node.

This array is usually far smaller than the space occupied by the file table described in the previous section. The reason is simple. The table for holding the linked list of all disk blocks is proportional in size to the disk itself. If the disk has *n* blocks, the table needs *n* entries. As disks grow larger, this table grows linearly with them. In contrast, the i-node scheme requires an array in memory whose size is proportional to the maximum number of files that may be open at once. It does not matter if the disk is 100 GB, 1000 GB, or 10,000 GB.

One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address of a block containing more disk-block addresses, as shown in Fig. 4-13. Even more advanced would be two or more such blocks containing disk addresses or even disk blocks pointing to other disk blocks full of addresses. We will come back to i-nodes when studying UNIX in Chap. 10. Similarly, the Windows NTFS file system uses a similar idea, only with bigger i-nodes that can also contain small files.

4.3.3 Implementing Directories

Before a file can be read, it must be opened. When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry on the disk. The directory entry provides the information needed to find the disk blocks. Depending on the system, this information may be the disk address of the entire file (with contiguous allocation), the number of the first block (both linked-list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

A closely related issue is where the attributes should be stored. Every file system maintains various file attributes, such as each file's owner and creation time, and they must be stored somewhere. One obvious possibility is to store them directly in the directory entry. Some systems do precisely that. This option is shown in Fig. 4-14(a). In this simple design, a directory consists of a list of fixed-size entries, one per file, containing a (fixed-length) file name, a structure of the file attributes, and one or more disk addresses (up to some maximum) telling where the disk blocks are.

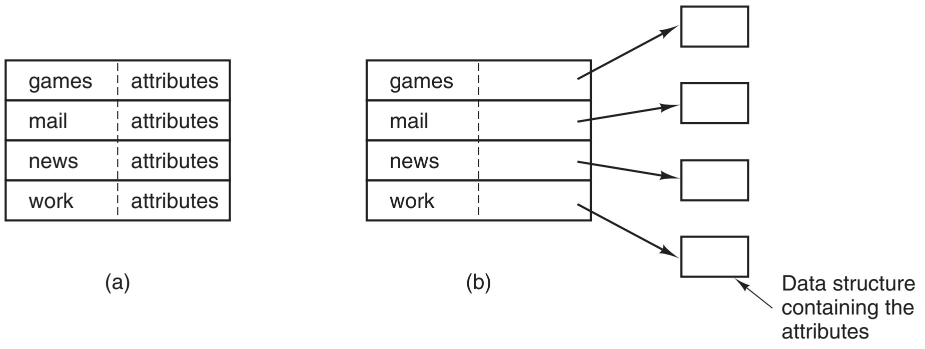


Figure 4-14. (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries. In that case, the directory entry can be shorter: just a file name and an i-node number. This approach is illustrated in Fig. 4-14(b). As we shall see later, this method has some advantages over putting them in the directory entry.

So far we have made the assumption that files have short, fixed-length names. In MS-DOS files have a 1–8 character base name and an optional extension of 1–3 characters. In UNIX Version 7, file names were 1–14 characters, including any extensions. However, nearly all modern operating systems support longer, variable-length file names. How can these be implemented?

The simplest approach is to set a limit on file-name length, typically 255 characters, and then use one of the designs of Fig. 4-14 with 255 characters reserved for each file name. This approach is simple, but wastes a great deal of directory space, since few files have such long names. For efficiency reasons, a different structure is desirable.

One alternative is to give up the idea that all directory entries are the same size. With this method, each directory entry contains a fixed portion, typically starting with the length of the entry, and then followed by data with a fixed format, usually including the owner, creation time, protection information, and other attributes. This fixed-length header is followed by the actual file name, however long it may be, as shown in Fig. 4-15(a) in big-endian format (e.g., SPARC). In this example we have three files, *project-budget*, *personnel*, and *foo*. Each file name is terminated by a special character (usually 0), which is represented in the figure by a box with a cross in it. To allow each directory entry to begin on a word boundary, each file name is filled out to an integral number of words, shown by shaded boxes in the figure.

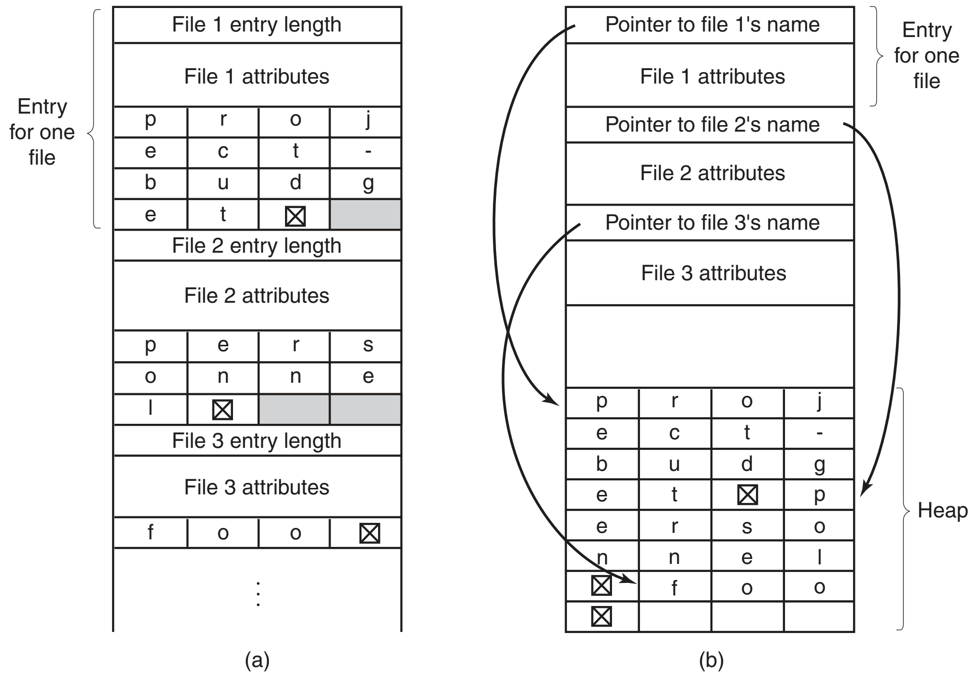


Figure 4-15. Two ways of handling long file names in a directory. (a) In-line. (b) In a heap.

A disadvantage of this method is that when a file is removed, a variable-sized gap is introduced into the directory into which the next file to be entered may not fit. This problem is essentially the same one we saw with contiguous disk files, only now compacting the directory is feasible because it is entirely in memory. Another problem is that a single directory entry may span multiple pages, so a page fault may occur while reading a file name.

Another way to handle variable-length names is to make the directory entries themselves all fixed length and keep the file names together in a heap at the end of the directory, as shown in Fig. 4-15(b). This method has the advantage that when an entry is removed, the next file entered will always fit there. Of course, the heap must be managed and page faults can still occur while processing file names. One minor win here is that there is no longer any real need for file names to begin at word boundaries, so no filler characters are needed after file names in Fig. 4-15(b) as they are in Fig. 4-15(a).

In all of the designs so far, directories are searched linearly from beginning to end when a file name has to be looked up. For extremely long directories, linear searching can be slow. One way to speed up the search is to use a hash table in each directory. Call the size of the table n. To enter a file name, the name is hashed onto a value between 0 and $n - 1$, for example, by dividing it by n and taking the remainder. Alternatively, the words comprising the file name can be added up and this quantity divided by n, or something similar.

Either way, the table entry corresponding to the hash code is inspected. If it is unused, a pointer is placed there to the file entry. File entries follow the hash table. If that slot is already in use, a linked list is constructed, headed at the table entry and threading through all entries with the same hash value.

Looking up a file follows the same procedure. The file name is hashed to select a hash-table entry. All the entries on the chain headed at that slot are checked to see if the file name is present. If the name is not on the chain, the file is not present in the directory.

Using a hash table has the advantage of much faster lookup, but the disadvantage of more complex administration. It is only really a serious candidate in systems where it is expected that directories will routinely contain hundreds or thousands of files.

A different way to speed up searching large directories is to cache the results of searches. Before starting a search, a check is first made to see if the file name is in the cache. If so, it can be located immediately. Of course, caching only works if a relatively small number of files comprise the majority of the lookups.

4.3.4 Shared Files

When several users are working together on a project, they often need to share files. As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users. Figure 4-16 shows the file system of Fig. 4-7 again, only with one of C's files now present in one of B's directories as well. The connection between B's directory and the shared file is called a

link. The file system itself is now a **Directed Acyclic Graph**, or **DAG**, rather than a tree. Having the file system be a DAG complicates maintenance, but such is life.

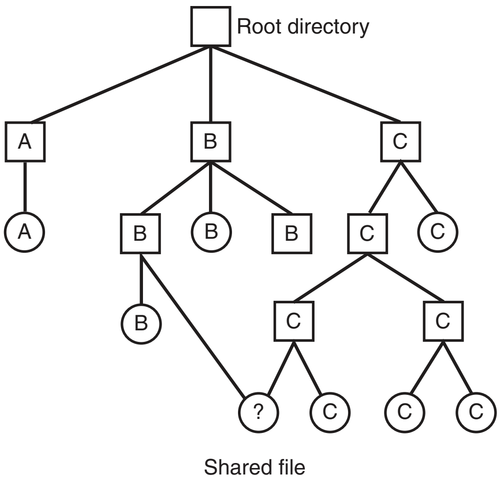


Figure 4-16. File system containing a shared file.

Sharing files is convenient, but it also introduces some problems. To start with, if directories really do contain disk addresses, then a copy of the disk addresses will have to be made in *B*'s directory when the file is linked. If either *B* or *C* subsequently appends to the file, the new blocks will be listed only in the directory of the user doing the append. The changes will not be visible to the other user, thus defeating the purpose of sharing.

This problem can be solved in two ways. In the first solution, disk blocks are not listed in directories, but in a little data structure associated with the file itself. The directories would then point just to the little data structure. This is the approach used in UNIX (where the little data structure is the i-node).

In the second solution, *B* links to one of *C*'s files by having the system create a new file, of type LINK, and entering that file in *B*'s directory. The new file contains just the path name of the file to which it is linked. When *B* reads from the linked file, the operating system sees that the file being read from is of type LINK, looks up the name of the file, and reads that file. This approach is called **symbolic linking**, to contrast it with traditional (hard) linking.

Each of these methods has its drawbacks. In the first method, at the moment that *B* links to the shared file, the i-node records the file's owner as *C*. Creating a link does not change the ownership (see Fig. 4-17), but it does increase the link count in the i-node, so the system knows how many directory entries currently point to the file.

If *C* subsequently tries to remove the file, the system is faced with a problem. If it removes the file and clears the i-node, *B* will have a directory entry pointing to

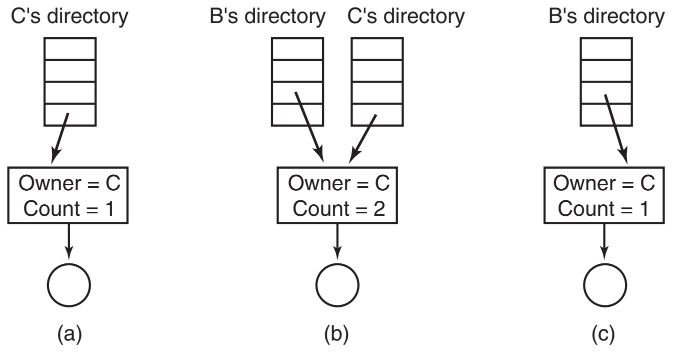


Figure 4-17. (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

an invalid i-node. If the i-node is later reassigned to another file, *B*'s link will point to the wrong file. The system can see from the count in the i-node that the file is still in use, but there is no easy way for it to find all the directory entries for the file, in order to erase them. Pointers to the directories cannot be stored in the i-node because there can be an unlimited number of directories.

The only thing to do is remove *C*'s directory entry, but leave the i-node intact, with count set to 1, as shown in Fig. 4-17(c). We now have a situation in which *B* is the only user having a directory entry for a file owned by *C*. If the system does accounting or has quotas, *C* will continue to be billed for the file until *B* decides to remove it, if ever, at which time the count goes to 0 and the file is deleted.

With symbolic links this problem does not arise because only the true owner has a pointer to the i-node. Users who have linked to the file just have path names, not i-node pointers. When the *owner* removes the file, it is destroyed. Subsequent attempts to use the file via a symbolic link will fail when the system is unable to locate the file. Removing a symbolic link does not affect the file at all.

The problem with symbolic links is the extra overhead required. The file containing the path must be read, then the path must be parsed and followed, component by component, until the i-node is reached. All of this activity may require a considerable number of extra disk accesses. Furthermore, an extra i-node is needed for each symbolic link, as is an extra disk block to store the path, although if the path name is short, the system could store it in the i-node itself, as a kind of optimization. Symbolic links have the advantage that they can be used to link to files on machines anywhere in the world, by simply providing the network address of the machine where the file resides in addition to its path on that machine.

There is also another problem introduced by links, symbolic or otherwise. When links are allowed, files can have two or more paths. Programs that start at a given directory and find all the files in that directory and its subdirectories will locate a linked file multiple times. For example, a program that dumps all the files in a directory and its subdirectories onto a tape may make multiple copies of a linked file. Furthermore, if the tape is then read into another machine, unless the dump program is clever, the linked file will be copied twice onto the disk, instead of being linked.

4.3.5 Log-Structured File Systems

Changes in technology are putting pressure on current file systems. In particular, CPUs keep getting faster, disks are becoming much bigger and cheaper (but not much faster), and memories are growing exponentially in size. The one parameter that is not improving by leaps and bounds is disk seek time (except for solid-state disks, which have no seek time).

The combination of these factors means that a performance bottleneck is arising in many file systems. Research done at Berkeley attempted to alleviate this problem by designing a completely new kind of file system, LFS (the **Log-structured File System**). In this section we will briefly describe how LFS works. For a more complete treatment, see the original paper on LFS (Rosenblum and Ousterhout, 1991).

The idea that drove the LFS design is that as CPUs get faster and RAM memories get larger, disk caches are also increasing rapidly. Consequently, it is now possible to satisfy a very substantial fraction of all read requests directly from the file-system cache, with no disk access needed. It follows from this observation that in the future, most disk accesses will be writes, so the read-ahead mechanism used in some file systems to fetch blocks before they are needed no longer gains much performance.

To make matters worse, in most file systems, writes are done in very small chunks. Small writes are highly inefficient, since a 50-μsec disk write is often preceded by a 10-msec seek and a 4-msec rotational delay. With these parameters, disk efficiency drops to a fraction of 1%.

To see where all the small writes come from, consider creating a new file on a UNIX system. To write this file, the i-node for the directory, the directory block, the i-node for the file, and the file itself must all be written. While these writes can be delayed, doing so exposes the file system to serious consistency problems if a crash occurs before the writes are done. For this reason, the i-node writes are generally done immediately.

From this reasoning, the LFS designers decided to reimplement the UNIX file system in such a way as to achieve the full bandwidth of the disk, even in the face of a workload consisting in large part of small random writes. The basic idea is to structure the entire disk as a great big log.

Periodically, and when there is a special need for it, all the pending writes being buffered in memory are collected into a single segment and written to the disk as a single contiguous segment at the end of the log. A single segment may thus contain i-nodes, directory blocks, and data blocks, all mixed together. At the start of each segment is a segment summary, telling what can be found in the segment. If the average segment can be made to be about 1 MB, almost the full bandwidth of the disk can be utilized.

In this design, i-nodes still exist and even have the same structure as in UNIX, but they are now scattered all over the log, instead of being at a fixed position on the disk. Nevertheless, when an i-node is located, locating the blocks is done in the usual way. Of course, finding an i-node is now much harder, since its address cannot simply be calculated from its i-number, as in UNIX. To make it possible to find i-nodes, an i-node map, indexed by i-number, is maintained. Entry *i* in this map points to i-node *i* on the disk. The map is kept on disk, but it is also cached, so the most heavily used parts will be in memory most of the time.

To summarize what we have said so far, all writes are initially buffered in memory, and periodically all the buffered writes are written to the disk in a single segment, at the end of the log. Opening a file now consists of using the map to locate the i-node for the file. Once the i-node has been located, the addresses of the blocks can be found from it. All of the blocks will themselves be in segments, somewhere in the log.

If disks were infinitely large, the above description would be the entire story. However, real disks are finite, so eventually the log will occupy the entire disk, at which time no new segments can be written to the log. Fortunately, many existing segments may have blocks that are no longer needed. For example, if a file is overwritten, its i-node will now point to the new blocks, but the old ones will still be occupying space in previously written segments.

To deal with this problem, LFS has a **cleaner** thread that spends its time scanning the log circularly to compact it. It starts out by reading the summary of the first segment in the log to see which i-nodes and files are there. It then checks the current i-node map to see if the i-nodes are still current and file blocks are still in use. If not, that information is discarded. The i-nodes and blocks that are still in use go into memory to be written out in the next segment. The original segment is then marked as free, so that the log can use it for new data. In this manner, the cleaner moves along the log, removing old segments from the back and putting any live data into memory for rewriting in the next segment. Consequently, the disk is a big circular buffer, with the writer thread adding new segments to the front and the cleaner thread removing old ones from the back.

The bookkeeping here is nontrivial, since when a file block is written back to a new segment, the i-node of the file (somewhere in the log) must be located, updated, and put into memory to be written out in the next segment. The i-node map must then be updated to point to the new copy. Nevertheless, it is possible to do the administration, and the performance results show that all this complexity is worthwhile. Measurements given in the papers cited above show that LFS outperforms UNIX by an order of magnitude on small writes, while having a performance that is as good as or better than UNIX for reads and large writes.

4.3.6 Journaling File Systems

While log-structured file systems are an interesting idea, they are not widely used, in part due to their being highly incompatible with existing file systems. Nevertheless, one of the ideas inherent in them, robustness in the face of failure, can be easily applied to more conventional file systems. The basic idea here is to keep a log of what the file system is going to do before it does it, so that if the system crashes before it can do its planned work, upon rebooting the system can look in the log to see what was going on at the time of the crash and finish the job. Such file systems, called **journaling file systems**, are actually in use. Microsoft's NTFS file system and the Linux ext3 and ReiserFS file systems all use journaling. OS X offers journaling file systems as an option. Below we will give a brief introduction to this topic.

To see the nature of the problem, consider a simple garden-variety operation that happens all the time: removing a file. This operation (in UNIX) requires three steps:

1. Remove the file from its directory.
2. Release the i-node to the pool of free i-nodes.
3. Return all the disk blocks to the pool of free disk blocks.

In Windows analogous steps are required. In the absence of system crashes, the order in which these steps are taken does not matter; in the presence of crashes, it does. Suppose that the first step is completed and then the system crashes. The i-node and file blocks will not be accessible from any file, but will also not be available for reassignment; they are just off in limbo somewhere, decreasing the available resources. If the crash occurs after the second step, only the blocks are lost.

If the order of operations is changed and the i-node is released first, then after rebooting, the i-node may be reassigned, but the old directory entry will continue to point to it, hence to the wrong file. If the blocks are released first, then a crash before the i-node is cleared will mean that a valid directory entry points to an i-node listing blocks now in the free storage pool and which are likely to be reused shortly, leading to two or more files randomly sharing the same blocks. None of these outcomes are good.

What the journaling file system does is first write a log entry listing the three actions to be completed. The log entry is then written to disk (and for good measure, possibly read back from the disk to verify that it was, in fact, written correctly). Only after the log entry has been written, do the various operations begin. After the operations complete successfully, the log entry is erased. If the system now crashes, upon recovery the file system can check the log to see if any operations were pending. If so, all of them can be rerun (multiple times in the event of repeated crashes) until the file is correctly removed.

To make journaling work, the logged operations must be **idempotent**, which means they can be repeated as often as necessary without harm. Operations such as "Update the bitmap to mark i-node *k* or block *n* as free" can be repeated until the cows come home with no danger. Similarly, searching a directory and removing any entry called *foobar* is also idempotent. On the other hand, adding the newly freed blocks from i-node *K* to the end of the free list is not idempotent since they may already be there. The more-expensive operation "Search the list of free blocks and add block *n* to it if it is not already present" is idempotent. Journaling file systems have to arrange their data structures and loggable operations so they all are idempotent. Under these conditions, crash recovery can be made fast and secure.

For added reliability, a file system can introduce the database concept of an **atomic transaction**. When this concept is used, a group of actions can be bracketed by the begin transaction and end transaction operations. The file system then knows it must complete either all the bracketed operations or none of them, but not any other combinations.

NTFS has an extensive journaling system and its structure is rarely corrupted by system crashes. It has been in development since its first release with Windows NT in 1993. The first Linux file system to do journaling was ReiserFS, but its popularity was impeded by the fact that it was incompatible with the then-standard ext2 file system. In contrast, ext3, which is a less ambitious project than ReiserFS, also does journaling while maintaining compatibility with the previous ext2 system.

4.3.7 Virtual File Systems

Many different file systems are in use—often on the same computer—even for the same operating system. A Windows system may have a main NTFS file system, but also a legacy FAT-32 or FAT-16 drive or partition that contains old, but still needed, data, and from time to time a flash drive, an old CD-ROM or a DVD (each with its own unique file system) may be required as well. Windows handles these disparate file systems by identifying each one with a different drive letter, as in *C:*, *D:*, etc. When a process opens a file, the drive letter is explicitly or implicitly present so Windows knows which file system to pass the request to. There is no attempt to integrate heterogeneous file systems into a unified whole.

In contrast, all modern UNIX systems make a very serious attempt to integrate multiple file systems into a single structure. A Linux system could have ext2 as the root file system, with an ext3 partition mounted on */usr* and a second hard disk with a ReiserFS file system mounted on */home* as well as an ISO 9660 CD-ROM temporarily mounted on */mnt*. From the user's point of view, there is a single file-system hierarchy. That it happens to encompass multiple (incompatible) file systems is not visible to users or processes.

However, the presence of multiple file systems is very definitely visible to the implementation, and since the pioneering work of Sun Microsystems (Kleiman, 1986), most UNIX systems have used the concept of a **VFS** (**virtual file system**) to try to integrate multiple file systems into an orderly structure. The key idea is to abstract out that part of the file system that is common to all file systems and put that code in a separate layer that calls the underlying concrete file systems to actually manage the data. The overall structure is illustrated in Fig. 4-18. The discussion below is not specific to Linux or FreeBSD or any other version of UNIX, but gives the general flavor of how virtual file systems work in UNIX systems.

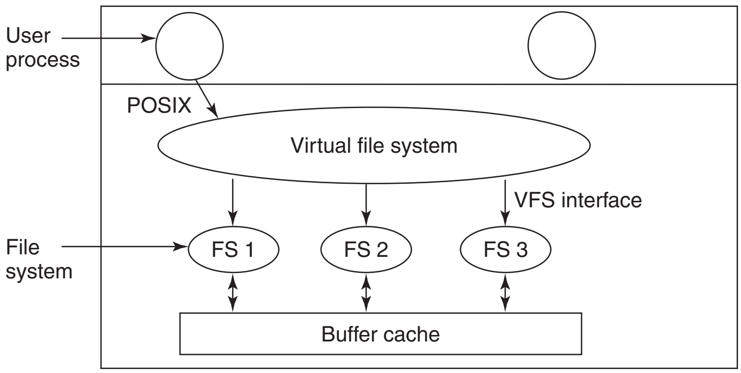


Figure 4-18. Position of the virtual file system.

All system calls relating to files are directed to the virtual file system for initial processing. These calls, coming from user processes, are the standard POSIX calls, such as open, read, write, lseek, and so on. Thus the VFS has an "upper" interface to user processes and it is the well-known POSIX interface.

The VFS also has a "lower" interface to the concrete file systems, which is labeled **VFS interface** in Fig. 4-18. This interface consists of several dozen function calls that the VFS can make to each file system to get work done. Thus to create a new file system that works with the VFS, the designers of the new file system must make sure that it supplies the function calls the VFS requires. An obvious example of such a function is one that reads a specific block from disk, puts it in the file system's buffer cache, and returns a pointer to it. Thus the VFS has two distinct interfaces: the upper one to the user processes and the lower one to the concrete file systems.

While most of the file systems under the VFS represent partitions on a local disk, this is not always the case. In fact, the original motivation for Sun to build the VFS was to support remote file systems using the **NFS** (**Network File System**) protocol. The VFS design is such that as long as the concrete file system supplies the functions the VFS requires, the VFS does not know or care where the data are stored or what the underlying file system is like.

Internally, most VFS implementations are essentially object oriented, even if they are written in C rather than C++. There are several key object types that are normally supported. These include the superblock (which describes a file system), the v-node (which describes a file), and the directory (which describes a file system directory). Each of these has associated operations (methods) that the concrete file systems must support. In addition, the VFS has some internal data structures for its own use, including the mount table and an array of file descriptors to keep track of all the open files in the user processes.

To understand how the VFS works, let us run through an example chronologically. When the system is booted, the root file system is registered with the VFS. In addition, when other file systems are mounted, either at boot time or during operation, they, too must register with the VFS. When a file system registers, what it basically does is provide a list of the addresses of the functions the VFS requires, either as one long call vector (table) or as several of them, one per VFS object, as the VFS demands. Thus once a file system has registered with the VFS, the VFS knows how to, say, read a block from it—it simply calls the fourth (or whatever) function in the vector supplied by the file system. Similarly, the VFS then also knows how to carry out every other function the concrete file system must supply: it just calls the function whose address was supplied when the file system registered.

After a file system has been mounted, it can be used. For example, if a file system has been mounted on */usr* and a process makes the call

```
open("/usr/include/unistd.h", O_RDONLY)
```

while parsing the path, the VFS sees that a new file system has been mounted on */usr* and locates its superblock by searching the list of superblocks of mounted file systems. Having done this, it can find the root directory of the mounted file system and look up the path *include/unistd.h* there. The VFS then creates a v-node and makes a call to the concrete file system to return all the information in the file's i-node. This information is copied into the v-node (in RAM), along with other information, most importantly the pointer to the table of functions to call for operations on v-nodes, such as read, write, close, and so on.

After the v-node has been created, the VFS makes an entry in the file-descriptor table for the calling process and sets it to point to the new v-node. (For the purists, the file descriptor actually points to another data structure that contains the current file position and a pointer to the v-node, but this detail is not important for our purposes here.) Finally, the VFS returns the file descriptor to the caller so it can use it to read, write, and close the file.

Later when the process does a read using the file descriptor, the VFS locates the v-node from the process and file descriptor tables and follows the pointer to the table of functions, all of which are addresses within the concrete file system on which the requested file resides. The function that handles read is now called and code within the concrete file system goes and gets the requested block. The VFS has no idea whether the data are coming from the local disk, a remote file system over the network, a USB stick, or something different. The data structures involved are shown in Fig. 4-19. Starting with the caller's process number and the file descriptor, successively the v-node, read function pointer, and access function within the concrete file system are located.

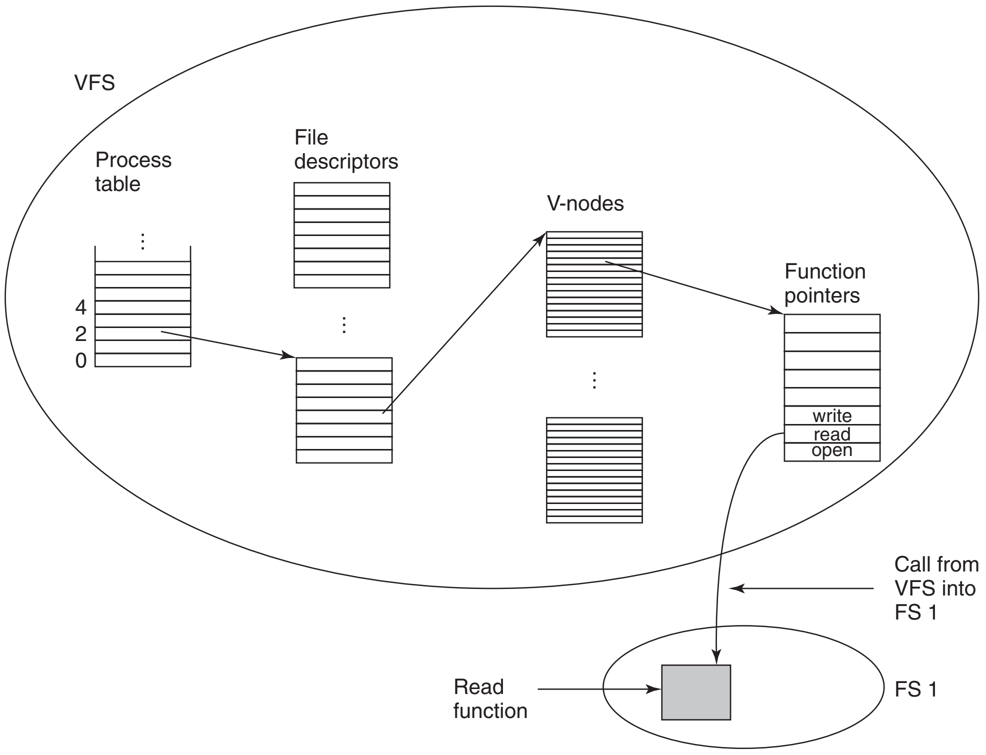


Figure 4-19. A simplified view of the data structures and code used by the VFS and concrete file system to do a read.

In this manner, it becomes relatively straightforward to add new file systems. To make one, the designers first get a list of function calls the VFS expects and then write their file system to provide all of them. Alternatively, if the file system already exists, then they have to provide wrapper functions that do what the VFS needs, usually by making one or more native calls to the concrete file system.

4.4 FILE-SYSTEM MANAGEMENT AND OPTIMIZATION

Making the file system work is one thing; making it work efficiently and robustly in real life is something quite different. In the following sections we will look at some of the issues involved in managing disks.

4.4.1 Disk-Space Management

Files are normally stored on disk, so management of disk space is a major concern to file-system designers. Two general strategies are possible for storing an *n* byte file: *n* consecutive bytes of disk space are allocated, or the file is split up into a number of (not necessarily) contiguous blocks. The same trade-off is present in memory-management systems between pure segmentation and paging.

As we have seen, storing a file as a contiguous sequence of bytes has the obvious problem that if a file grows, it may have to be moved on the disk. The same problem holds for segments in memory, except that moving a segment in memory is a relatively fast operation compared to moving a file from one disk position to another. For this reason, nearly all file systems chop files up into fixed-size blocks that need not be adjacent.

Block Size

Once it has been decided to store files in fixed-size blocks, the question arises how big the block should be. Given the way disks are organized, the sector, the track, and the cylinder are obvious candidates for the unit of allocation (although these are all device dependent, which is a minus). In a paging system, the page size is also a major contender.

Having a large block size means that every file, even a 1-byte file, ties up an entire cylinder. It also means that small files waste a large amount of disk space. On the other hand, a small block size means that most files will span multiple blocks and thus need multiple seeks and rotational delays to read them, reducing performance. Thus if the allocation unit is too large, we waste space; if it is too small, we waste time.

Making a good choice requires having some information about the file-size distribution. Tanenbaum et al. (2006) studied the file-size distribution in the Computer Science Department of a large research university (the VU) in 1984 and then again in 2005, as well as on a commercial Web server hosting a political Website (*www.electoral-vote.com*). The results are shown in Fig. 4-20, where for each power-of-two file size, the percentage of all files smaller or equal to it is listed for each of the three data sets. For example, in 2005, 59.13% of all files at the VU were 4 KB or smaller and 90.84% of all files were 64 KB or smaller. The median file size was 2475 bytes. Some people may find this small size surprising.

What conclusions can we draw from these data? For one thing, with a block size of 1 KB, only about 30–50% of all files fit in a single block, whereas with a 4-KB block, the percentage of files that fit in one block goes up to the 60–70% range. Other data in the paper show that with a 4-KB block, 93% of the disk blocks are used by the 10% largest files. This means that wasting some space at the end of each small file hardly matters because the disk is filled up by a small number of

Length	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

Figure 4-20. Percentage of files smaller than a given size (in bytes).

large files (videos) and the total amount of space taken up by the small files hardly matters at all. Even doubling the space the smallest 90% of the files take up would be barely noticeable.

On the other hand, using a small block means that each file will consist of many blocks. Reading each block normally requires a seek and a rotational delay (except on a solid-state disk), so reading a file consisting of many small blocks will be slow.

As an example, consider a disk with 1 MB per track, a rotation time of 8.33 msec, and an average seek time of 5 msec. The time in milliseconds to read a block of *k* bytes is then the sum of the seek, rotational delay, and transfer times:

$$5 + 4.165 + (k/1000000) \times 8.33$$

The dashed curve of Fig. 4-21 shows the data rate for such a disk as a function of block size. To compute the space efficiency, we need to make an assumption about the mean file size. For simplicity, let us assume that all files are 4 KB. Although this number is slightly larger than the data measured at the VU, students probably have more small files than would be present in a corporate data center, so it might be a better guess on the whole. The solid curve of Fig. 4-21 shows the space efficiency as a function of block size.

The two curves can be understood as follows. The access time for a block is completely dominated by the seek time and rotational delay, so given that it is going to cost 9 msec to access a block, the more data that are fetched, the better.

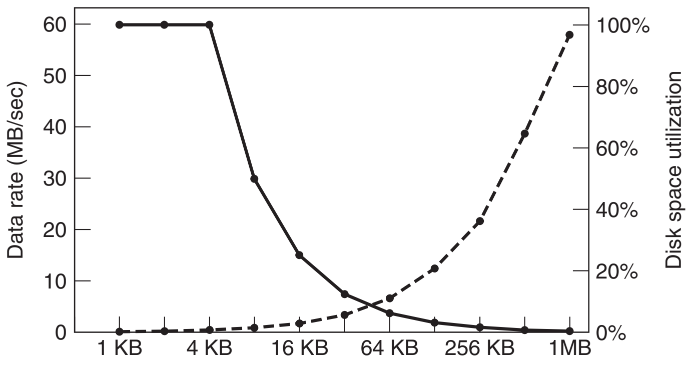


Figure 4-21. The dashed curve (left-hand scale) gives the data rate of a disk. The solid curve (right-hand scale) gives the disk-space efficiency. All files are 4 KB.

Hence the data rate goes up almost linearly with block size (until the transfers take so long that the transfer time begins to matter).

Now consider space efficiency. With 4-KB files and 1-KB, 2-KB, or 4-KB blocks, files use 4, 2, and 1 block, respectively, with no wastage. With an 8-KB block and 4-KB files, the space efficiency drops to 50%, and with a 16-KB block it is down to 25%. In reality, few files are an exact multiple of the disk block size, so some space is always wasted in the last block of a file.

What the curves show, however, is that performance and space utilization are inherently in conflict. Small blocks are bad for performance but good for disk-space utilization. For these data, no reasonable compromise is available. The size closest to where the two curves cross is 64 KB, but the data rate is only 6.6 MB/sec and the space efficiency is about 7%, neither of which is very good. Historically, file systems have chosen sizes in the 1-KB to 4-KB range, but with disks now exceeding 1 TB, it might be better to increase the block size to 64 KB and accept the wasted disk space. Disk space is hardly in short supply any more.

In an experiment to see if Windows NT file usage was appreciably different from UNIX file usage, Vogels made measurements on files at Cornell University (Vogels, 1999). He observed that NT file usage is more complicated than on UNIX. He wrote:

> *When we type a few characters in the* Notepad *text editor, saving this to a file will trigger 26 system calls, including 3 failed open attempts, 1 file overwrite and 4 additional open and close sequences.*

Nevertheless, Vogels observed a median size (weighted by usage) of files just read as 1 KB, files just written as 2.3 KB, and files read and written as 4.2 KB. Given the different data sets measurement techniques, and the year, these results are certainly compatible with the VU results.

Keeping Track of Free Blocks

Once a block size has been chosen, the next issue is how to keep track of free blocks. Two methods are widely used, as shown in Fig. 4-22. The first one consists of using a linked list of disk blocks, with each block holding as many free disk block numbers as will fit. With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is required for the pointer to the next block.) Consider a 1-TB disk, which has about 1 billion disk blocks. To store all these addresses at 255 per block requires about 4 million blocks. Generally, free blocks are used to hold the free list, so the storage is essentially free.

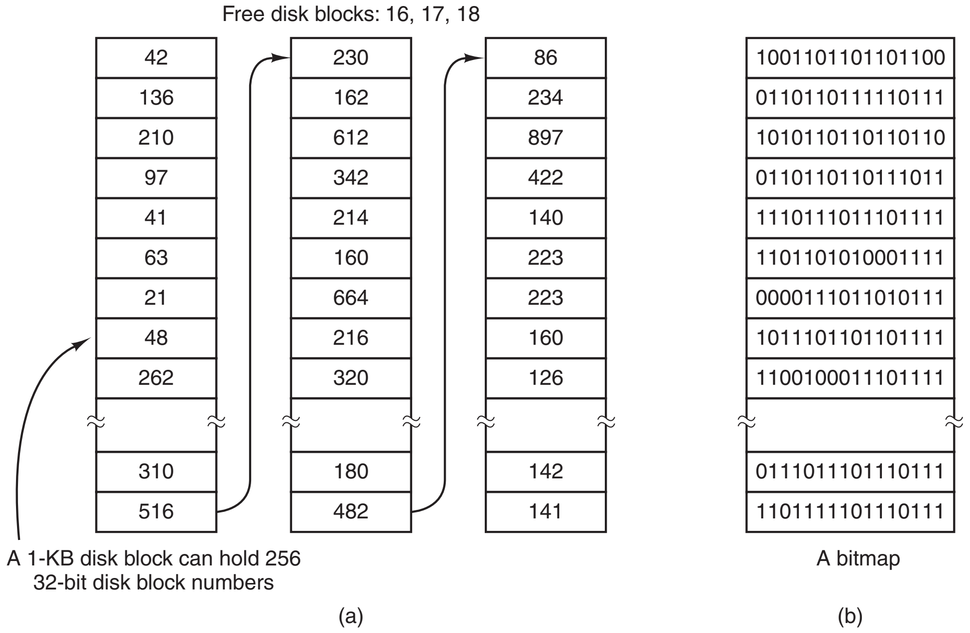


Figure 4-22. (a) Storing the free list on a linked list. (b) A bitmap.

The other free-space management technique is the bitmap. A disk with *n* blocks requires a bitmap with *n* bits. Free blocks are represented by 1s in the map, allocated blocks by 0s (or vice versa). For our example 1-TB disk, we need 1 billion bits for the map, which requires around 130,000 1-KB blocks to store. It is not surprising that the bitmap requires less space, since it uses 1 bit per block, vs. 32 bits in the linked-list model. Only if the disk is nearly full (i.e., has few free blocks) will the linked-list scheme require fewer blocks than the bitmap.

If free blocks tend to come in long runs of consecutive blocks, the free-list system can be modified to keep track of runs of blocks rather than single blocks. An 8-, 16-, or 32-bit count could be associated with each block giving the number of consecutive free blocks. In the best case, a basically empty disk could be represented by two numbers: the address of the first free block followed by the count of free blocks. On the other hand, if the disk becomes severely fragmented, keeping track of runs is less efficient than keeping track of individual blocks because not only must the address be stored, but also the count.

This issue illustrates a problem operating system designers often have. There are multiple data structures and algorithms that can be used to solve a problem, but choosing the best one requires data that the designers do not have and will not have until the system is deployed and heavily used. And even then, the data may not be available. For example, our own measurements of file sizes at the VU in 1984 and 1995, the Website data, and the Cornell data are only four samples. While a lot better than nothing, we have little idea if they are also representative of home computers, corporate computers, government computers, and others. With some effort we might have been able to get a couple of samples from other kinds of computers, but even then it would be foolish to extrapolate to all computers of the kind measured.

Getting back to the free list method for a moment, only one block of pointers need be kept in main memory. When a file is created, the needed blocks are taken from the block of pointers. When it runs out, a new block of pointers is read in from the disk. Similarly, when a file is deleted, its blocks are freed and added to the block of pointers in main memory. When this block fills up, it is written to disk.

Under certain circumstances, this method leads to unnecessary disk I/O. Consider the situation of Fig. 4-23(a), in which the block of pointers in memory has room for only two more entries. If a three-block file is freed, the pointer block overflows and has to be written to disk, leading to the situation of Fig. 4-23(b). If a three-block file is now written, the full block of pointers has to be read in again, taking us back to Fig. 4-23(a). If the three-block file just written was a temporary file, when it is freed, another disk write is needed to write the full block of pointers back to the disk. In short, when the block of pointers is almost empty, a series of short-lived temporary files can cause a lot of disk I/O.

An alternative approach that avoids most of this disk I/O is to split the full block of pointers. Thus instead of going from Fig. 4-23(a) to Fig. 4-23(b), we go from Fig. 4-23(a) to Fig. 4-23(c) when three blocks are freed. Now the system can handle a series of temporary files without doing any disk I/O. If the block in memory fills up, it is written to the disk, and the half-full block from the disk is read in. The idea here is to keep most of the pointer blocks on disk full (to minimize disk usage), but keep the one in memory about half full, so it can handle both file creation and file removal without disk I/O on the free list.

With a bitmap, it is also possible to keep just one block in memory, going to disk for another only when it becomes completely full or empty. An additional benefit of this approach is that by doing all the allocation from a single block of the bitmap, the disk blocks will be close together, thus minimizing disk-arm motion.

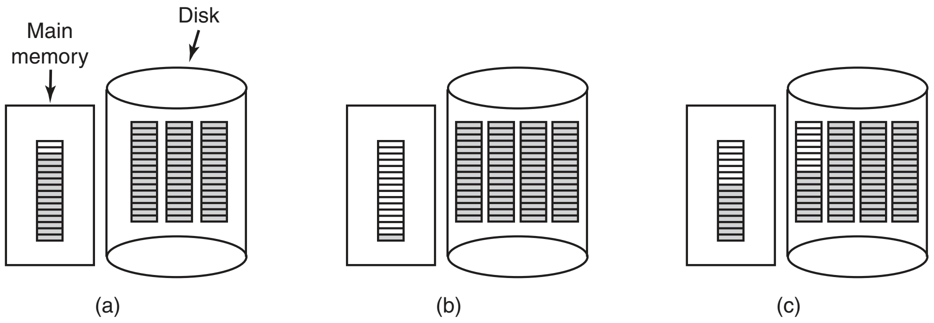


Figure 4-23. (a) An almost-full block of pointers to free disk blocks in memory and three blocks of pointers on disk. (b) Result of freeing a three-block file. (c) An alternative strategy for handling the three free blocks. The shaded entries represent pointers to free disk blocks.

Since the bitmap is a fixed-size data structure, if the kernel is (partially) paged, the bitmap can be put in virtual memory and have pages of it paged in as needed.

Disk Quotas

To prevent people from hogging too much disk space, multiuser operating systems often provide a mechanism for enforcing disk quotas. The idea is that the system administrator assigns each user a maximum allotment of files and blocks, and the operating system makes sure that the users do not exceed their quotas. A typical mechanism is described below.

When a user opens a file, the attributes and disk addresses are located and put into an open-file table in main memory. Among the attributes is an entry telling who the owner is. Any increases in the file's size will be charged to the owner's quota.

A second table contains the quota record for every user with a currently open file, even if the file was opened by someone else. This table is shown in Fig. 4-24. It is an extract from a quota file on disk for the users whose files are currently open. When all the files are closed, the record is written back to the quota file.

When a new entry is made in the open-file table, a pointer to the owner's quota record is entered into it, to make it easy to find the various limits. Every time a block is added to a file, the total number of blocks charged to the owner is incremented, and a check is made against both the hard and soft limits. The soft limit may be exceeded, but the hard limit may not. An attempt to append to a file when the hard block limit has been reached will result in an error. Analogous checks also exist for the number of files to prevent a user from hogging all the i-nodes.

When a user attempts to log in, the system examines the quota file to see if the user has exceeded the soft limit for either number of files or number of disk blocks.