

OPERATING SYSTEMS

A Design-Oriented Approach

Charles Crowley
University of New Mexico

J. ROBERT VAN PELT LIBRARY
MICHIGAN TECHNOLOGICAL UNIVERSITY
HOUGHTON, MICHIGAN

IRWIN

Chicago • Bogotá • Boston • Buenos Aires • Caracas
London • Madrid • Mexico City • Sydney • Toronto

**IRWIN Concerned about Our Environment**

In recognition of the fact that our company is a large end-user of fragile yet replenishable resources, we at IRWIN can assure you that every effort is made to meet or exceed Environmental Protection Agency (EPA) recommendations and requirements for a "greener" workplace.

To preserve these natural assets, a number of environmental policies, both companywide and department-specific, have been implemented. From the use of 50% recycled paper in our textbooks to the printing of promotional materials with recycled stock and soy inks to our office paper recycling program, we are committed to reducing waste and replacing environmentally unsafe products with safer alternatives.

© Richard D. Irwin, a Times Mirror Higher Education Group, Inc., company, 1997

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Irwin Book Team

Publisher: *Tom Casson*

Senior-sponsoring editor: *Elizabeth A. Jones*

Senior developmental editor: *Kelley Butcher*

Project supervisor: *Lynne Basler*

Senior Production supervisor: *Laurie Sander*

Director, Prepress Purchasing: *Kimberly Meriwether David*

Compositor: *Interactive Composition Corporation*

Typeface: *10/12 Times Roman*

Printer: *Times Mirror Higher Education Group, Inc., Print Group*

**Times Mirror****Higher Education Group****Library of Congress Cataloging-in-Publication Data**

Y1 Crowley, Charles (Charles Patrick)¹

Operating systems : a design-oriented approach / Charles Crowley.

p. cm.

Includes bibliographical references and index.

ISBN 0-256-15151-2

1. Operating systems (Computers) I. Title.

QA76.76.063C77 1997

005.4'3—dc20

96-43708

Printed in the United States of America

1 2 3 4 5 6 7 8 9 0 WCB 3 2 1 0 9 8 7 6

To Mom, Pat, Cathy, Ella, and Kendra

APR 18 2002

PREFACE

In this book I have tried to approach the traditional, junior or senior level operating systems course in a new way. There are several areas where I have done things differently:

- *Describe the external operating system interface:* I start with a description of the system call interface to an operating system.
- *Use of code:* I have tried to steer a middle course between a concepts approach and a case study approach.
- *Development of concepts:* I have tried to show how the operating system concepts developed into their present form.
- *Design orientation:* I have tried to show how ideas from the design of operating systems relate to the design of other types of programs.

EXTERNAL OPERATING SYSTEM INTERFACE

Many students come to an operating system class without a clear understanding of what an operating system really does. These students should understand how operating system services are used before learning how these services are implemented. To deal with this, the book begins with a simplified, UNIX-like set of system calls. The book includes a discussion of these calls, example programs using these calls, and a simple shell program to integrate the examples.

CONCEPTS OR CASE STUDIES

There have always been two approaches to the operating systems class. The first approach is the *concept* or *theory* approach which concentrates on the basic conceptual issues in the design of operating systems. These courses discuss each of the basic problems in operating systems design and the range of common solutions to those problems. The books are mostly text and diagrams with very little code. The second approach is the *case study* method which concentrates on an example operating system that is simple but complete. The books contain a lot of code and spend a lot of pages explaining the code in detail.

There are advantages and disadvantages to both approaches. Some people feel the ideal situation is to take both classes but this is rarely possible in an already crowded computer science curriculum so one is required to make a choice.

USE OF CODE

I have tried to find a middle course between the two approaches. This book is basically a concepts oriented book with more code than is usual. I have found that seeing actual code allows the students to understand the concepts more deeply, feel more comfortable about

the material, and ask questions they wouldn't have thought to ask in a purely concepts oriented course. The code does not comprise a complete operating system however and it as simple as possible in order to reduce the amount of pages devoted to explaining it.

DEVELOPMENT OF CONCEPTS

I have tried to show how these ideas developed. Many of the concepts in operating systems have developed over many years and the current solutions were developed slowly, in several stages. Each new solution had a problem that the next solution tried to fix. I think it helps the student to understand this development and see that these ideas were not brilliant flashes of insight that came out of nowhere but ideas that were improved by many people over many years. The development was a series of good ideas where each improvement made sense in the context in which it was developed. Seeing this development helps to understand why the solutions have their present form. In addition, it is useful to know the design constraints that caused solutions to develop into their present form because technological advances often change these constraints and old solutions that used to be inferior suddenly become practical again. Finally these developments give students examples of the design process through a series of potential but flawed solutions to a problem to a final solution that is acceptable.

DESIGN ORIENTATION

Finally there is a concentration on design. In some ways designing an operating system is a pretty specialized activity having to do mainly with resource management. But many basic design ideas run through all designs and they show up in operating systems as much as anywhere else. Throughout the book I note places where we are presented with typical design problems. I abstract the operating systems related problems and solutions from the book into the general design problems and solutions and present them in a way that they can be applied to design problems in other areas of computer science.

Clearly it is not possible to cover all design topics and issues. I am striving for two things. First, I want to give the student an awareness of design issues, where they come up, which techniques to apply, how they can be generalized, etc. I do not present an organized survey of design techniques but a series of useful ones that come up in the context of operating systems. I hope to make the student aware of design and to enable the student to start doing their own generalizing about design. Second, I present a collection of useful design techniques that the students can use in their design toolkit.

Very few computer professionals will participate in the design of an operating system during the course of their careers. While it is important that students of computer science have a good foundation in the basic concepts in specialized areas such as operating systems, it is not necessary that every computer science student

understand all the details. However, there is a thread running through all areas of software engineering: the concept of design. There are many issues which are tackled during the design of an operating system which can be generalized and applied to other areas of computer science. In this book, I attempt to focus on these design issues and their implications for other areas.

I have oriented this book to provide a solid preparation for the larger design projects the student will encounter in later software engineering courses and as preparation for their career as a software professional designing, implementing and maintaining a wide variety of systems. This orientation also enables the operating systems course follows modern developments in the field of computer science. The interrelations between the separate areas of computer science are becoming more important. For example, in the area of high speed parallel machines, it is clear that it is necessary to think of the hardware, the operating system, and the programming language as a single system to get maximum performance. Optimization in any part of the system will have consequences for the other parts of the system.

The design techniques are noted in side bars as they come up and longer explanations of each design topic are placed in separate chapters from the operating system material. The instructor can structure a course with varying degrees of concentration on design aspects. The goal is that the design sections are independent of the main flow of the text and independent from each other. This will allow the instructor to pick and choose those design sections he or she finds to be useful.

USING THIS BOOK IN A COURSE

There is more material in this book than can be comfortably covered in a one-semester course. A number of the sections of the book have been marked with an asterisk. This indicates that they can be skipped with no loss of continuity in the presentation. In addition, all of the design chapters and design sidebars can be skipped with no loss of continuity. If you skip all the design chapters you can probably just cover the book in a semester. I expect that most instructors will choose to skip some of the design sections and some of the optional sections and teach a course that is about 90 percent operating systems and 10 percent design issues. Alternatively, the design issues could be covered in a separate one-unit course, strictly on design, that is taken along with the operating systems course.

ACKNOWLEDGMENTS

I want to thank my colleague Barney Maccabe who started out as my co-author on the book but had too many other commitments to work on the book. Our discussions lead to the conception of the book and the design orientation. I have talked with Barney on many aspects of operating systems and computer science and those discussions shaped many of my ideas.

I also want to thank John Brayer who used drafts of the book in several of his operating systems courses and put up with many typos, badly written sections, and unfinished sections. His comments helped in the development of the book.

I also want to thank my other colleagues in the Computer Science Department at the University of New Mexico. Continuing “in-the-hall” discussions with them have helped me formulate and improve my ideas.

Many students in CS 481 have used drafts of the books and have provided many useful comments and suggestions. I want to particularly thank Dave Rosenbaum who provided extensive comments on the entire draft, found numerous errors and typos and taught me some things about writing good English as well.

My treatment of the design chapters was greatly influenced by the design patterns developments in recent years and particularly the fine book by Gamma et al. (1995). I used their presentation format in the design chapters.

I want to thank all the people who reviewed this book while it was being developed. I corrected many problems and got many ideas for the presentation from their careful reviews and comments. The design chapters especially profited from their comments.

- Mustaque Ahamad, Georgia Institute of Technology
- Jim Alves-Foss, University of Idaho
- Anish Arora, The Ohio State University
- Brent Auernheimer, California State University - Fresno
- Anthony Q. Baxter, University of Kentucky
- Mahesh Dodani, University of Iowa
- H. George Friedman, Jr., University of Illinois - Urbana
- Tim Gottleber
- Stephen J. Hartley, Drexel University
- Giorgio P. Ingargiola, Temple University
- Stephen J. Krebsbach, South Dakota State University
- Donald S. Miller, Arizona State University
- Matt W. Mutka, Michigan State University
- Richard Newman-Wolfe, University of Florida
- Steve Reichenbach, University of Nebraska - Lincoln
- Bernhard Weinberg, Michigan State University

Finally my wife, Ella Sitkin, put up with many bad moods and excuses that I was “too busy” during the years this book was developed. As if that wasn’t enough, she also edited several chapters.

chapter

1

Introduction

Operating systems run on computer hardware and serve as platforms for other software to run on. In this chapter, we present an overview of an operating system. We focus on four questions:

- Where does an operating system fit in a computing system?
- What does an operating system do?
- How is an operating system structured?
- Do we need an operating system?

We look at an operating system from two viewpoints: as a resource manager and as a virtual machine manager. We also look at how the operating system stands between the hardware and the user software.

1.1 WHERE DOES AN OPERATING SYSTEM FIT IN?

operating system

An *operating system* is the layer of a computer system between the hardware and the user programs (user software). (See Figure 1.1.) An operating system does what all software does: it implements some desired functionality by building on the functionality available in lower levels. Software transforms one interface into another interface. For example, a BASIC interpreter transforms a computer into a BASIC machine, a statistical package transforms a computer into a statistics machine, a computer game program transforms a computer into a game machine, etc.

An operating system is built directly on the hardware interface and provides an interface between the hardware and the user programs. The first thing we will do in this book is look at exactly what kind of interface the operating system implements. Then, in the rest of the book, we will look at how the operating systems interface is implemented.

A computer system, as it comes “out of the box,” is not easy to use—it requires an operating system and application software to run on it. An application software package, as it comes “out of the box” (or “out of the shrinkwrap”), will not run directly on the computer hardware—it requires an operating system to run it.¹ An operating system is the layer of software that nearly every application software package expects to be present. The operating system is the first program run on a computer when the computer boots up. There can be only one operating system running on a computer.²

The operating system is the layer between the hardware and the software.

1.1.1 SYSTEM LEVELS

hardware interface

operating system interface

Figure 1.1 illustrates the layered structure of the hardware, the software, and the operating system. The *hardware interface* consists of things like program counter, registers, interrupts, disks, terminals, etc. The hardware interface consists of everything you need to know about the hardware in order to write programs that will execute on the hardware. In Chapter 2, we describe the hardware interface for the example hardware used in this book. The *operating system interface* is described in Chapter 3.

¹This is not true in all cases. There are some special programs that are intended to be run on the bare hardware. One example would be a program that fixes disk problems. These are special cases, however, and are not common.

²Technically this is not always true since it is possible to have a “virtual” operating system that runs other operating systems. The correct statement is that only one operating system is in direct control of the hardware at any one time.

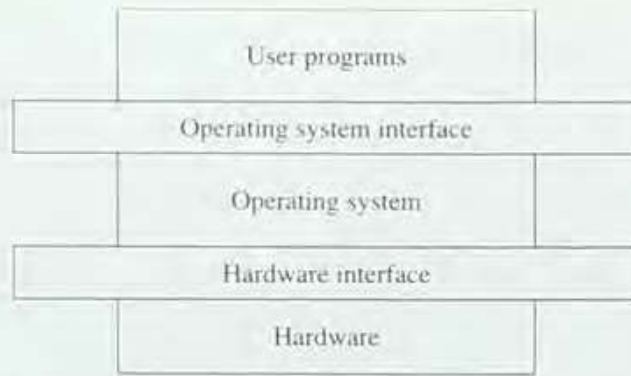


Figure 1.1 Levels in a computer system

An operating system shares characteristics with both hardware and software. An operating system is software, that is, it is a program that is compiled, linked, and run on a computer. However, it is like hardware in that only one copy of the operating system runs on a computer and it extends the capabilities of the hardware. The services of the operating system are invoked with a special “trap” or “system call” instruction that is used just like any other hardware instruction.

DESIGN TECHNIQUE: TWO-LEVEL IMPLEMENTATION

An operating system is a lower level of software that user programs run on. Together they form a two-level structure where the operating system implements functionality used by the user program. This idea is called a *two-level implementation* and it is seen in many places in computer science. We will see several more examples later in the book.

A programming language is another example. It is the lower level used to implement programs. Special-purpose languages are an even better example. The special-purpose language provides low-level primitives for solving certain kinds of problems, and the user writes programs in that language. Some examples are the 4GL languages for producing reports from database information, and spreadsheet macro languages.

In a two-level implementation, you use the “*divide and conquer*” strategy by dividing the problem into two parts and solving each part separately. A two-level implementation is a special case of a modular implementation, but the two-level case occurs so often that it is useful to discuss it as a separate design technique from modular implementation. It is common for the

lower level to implement a language that the upper level is written in.

Two-level implementations are especially useful when the lower level can be reused by more than one upper level. This is the case with an operating system and most languages. A good lower level can be reused many times.

Two-level implementations are used in operating systems to separate mechanism from policy. A **mechanism** is a set of basic facilities that can be used in many different ways. For example, a programming language is a mechanism for writing programs. A **policy** is the use of a mechanism for a particular purpose—for example, a program to solve a specific equation. An example from operating systems is the protection of files. UNIX provides a protection mechanism that determines whether other people can read, write, or execute your files. You decide on the policy, that is, which files are readable, writable, or executable. The distinction between mechanism and policy is a basic concept in operating systems and will reappear in many places in this book.

You can learn more about two-level implementations in Section 4.4.

1.2 WHAT DOES AN OPERATING SYSTEM DO?

The functions of an operating system can be viewed in two ways:

- As a resource manager.
- As an implementor of virtual computers.

By *virtual computer* we mean an implementation of the functionality of a computer in software. In this section we will introduce both viewpoints.

An operating system is a resource manager and a virtual computer manager.

1.2.1 HARDWARE RESOURCES

The operating system manages the hardware resources of the computer system. The main *hardware resources* in a computer system are:

- *Processor*—the processor is the part of the computer system capable of executing instructions.
- *Memory*—the memory contains all instructions and data used by a processor.
- *Input/output (I/O) controllers*—processors that know how to transfer data between memory and devices.
- *Disk devices*—disks provide long-term storage for data.
- *Other devices*—hardware components that accept data from or generate data for I/O controllers. Examples are terminals, networks, tape drives, etc.

Figure 1.2 shows the resources in a computer system.

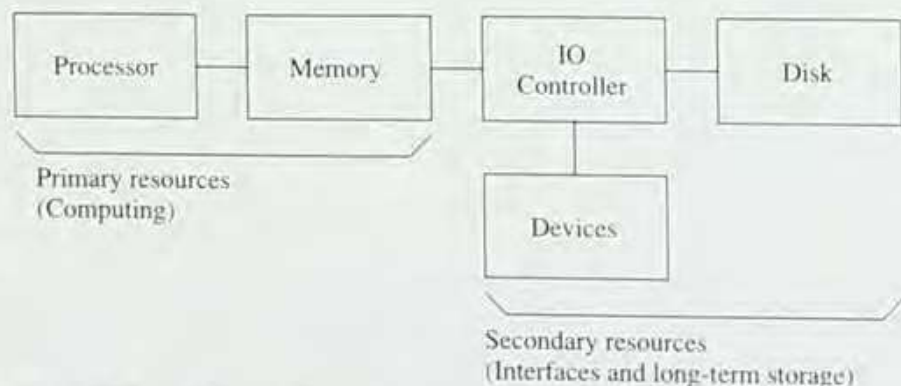


Figure 1.2 The hardware resources in a computer system

1.2.2 RESOURCE MANAGEMENT

Resource management is:

resource management

- *Transforming*—creating a new resource from an existing resource. The created resource will act as a substitute for the existing resource but will be easier to use.
- *Multiplexing*—creating the illusion of several resources from one resource.
- *Scheduling*—deciding which programs should get each resource and when they should get it.

To give a more concrete meaning for transformation and multiplexing, we consider the construction of multiple virtual printers—one for each virtual computer.

Transformation Hardware resources have moderately complex interfaces. The hardware interface to a printer might consist of a data register, a control register, and a status register. To send a character to the printer, you need to poll (repeatedly read) the status register until it indicates that the printer is ready to accept the next character. Once the printer is ready, the data value needs to be written to the data register and a “send” command written to the control register. None of this is particularly difficult, but it is tedious and error prone: you need to know the addresses of the data, control, and status registers and you need to know the structure of the bits in the control and status registers.

transformation
virtual resource

Operating systems *transform* physical resources into virtual resources to avoid the difficulties associated with using hardware resources. A *virtual resource* provides the essential functionality of the hardware resource, but is easier to use because the details of the hardware interface are hidden. For example, an operating system might provide a virtual printer that is capable of printing a character. To use this virtual resource, an application only needs to specify the character to be printed. The virtual printer provides the essential functionality of the hardware printer, while the operating system handles the details that make the hardware interface difficult to use (register addresses, register formats, and wait-until the printer is ready for each character).

Operating systems transform physical resources into virtual resources that provide functionality which is similar to the physical resources, but have a simpler interface.

Multiplexing When there are more virtual computers than physical resources, the operating system needs to make sure that the virtual computers can share the physical resources. The sharing of physical resources is called *multiplexing*. Continuing with our printer example, suppose your system only has one printer. If you run two or more applications, the operating system needs to make it appear as if each virtual computer has its own printer. That is, the operating system needs to make sure that the characters printed by one virtual computer are not mixed with the characters printed by another virtual computer.

multiplexing

To create the illusion of multiple printers, the operating system can implement a virtual printer using a disk file. In this solution, each virtual computer has its own “printer file.” Every time the application prints a character, the operating system simply appends the character to the end of the printer file which belongs to the virtual computer being used for that application. When the application finishes, the operating system queues the application’s printer file. The software that drives the physical printer only prints files.

spooling

Historically this solution was called *spooling*, and the application’s “printer file” is called a *spool file*. *SPOOL* is an acronym for *Simultaneous Peripheral Operations On Line*. Simultaneous refers to the fact that two or more virtual computers can issue printer operations at the same time. On line refers to the fact that the hardware printer is connected to the same computer as the virtual printers are running on. The alternative to on-line printing is to write the print file to a tape and print the tape on a different computer. See Section 5.16.1 for a more detailed explanation of off-line printing.

time-division multiplexing

Our printer example illustrates a simple form of *time-division multiplexing*. In time-division multiplexing, the resource is used for different virtual computers at different times. In many contexts, time-division multiplexing is called *time-sharing*.

time-sharing

This form of time-division multiplexing is relatively simple because the printer does not begin printing the output for a virtual computer until the application is finished. Moreover, the printer finishes printing all of the output for one virtual computer before it starts to print the output for another virtual computer. Other instances of time-division multiplexing may involve switching between virtual resources while they are still being used by applications. Time-division multiplexing is appropriate when the resource cannot be divided into smaller versions of itself.

space-division multiplexing

In *space-division multiplexing*, a resource is divided into smaller versions of itself and each virtual computer is given a part of the resource. Space-division multiplexing is used in multiplexing primary and secondary storage (memory and disks). Although not commonly used, the term *space-sharing* is a synonym for space-division multiplexing.

space-sharing

Multiplexing (sharing) of a single physical resource involves dividing the resource based on time or space.

An operating system multiplexes, transforms, and allocates hardware resources.

1.2.3 VIRTUAL COMPUTERS

An operating system creates software copies of the processor (the capability to execute instructions) and the memory (the capability to store information). It also

transforms the disk devices into a file system and the other input-output devices into more abstract (less detailed) and easily used devices. Thus an operating system:

- Creates multiple *processes* (simulated processors) out of the single processor and allocates them to programs. It accomplishes this by time-multiplexing the processor. process
- Creates multiple *address spaces* (memory for a process to run in) out of the memory and allocates them to processes. It accomplishes this by space-multiplexing the processor. address space
- Implements a *file system* and input/output (I/O) system so that processes can easily use and share the disks. It accomplishes this by space-multiplexing the disks and time-multiplexing the I/O channels. file system

The operating system creates what we will call *virtual computers* from the physical computer. The first thing we need to look at is how a virtual computer differs from the physical computer. virtual computer

The most significant difference is that there are many virtual computers while there is only one physical computer. The operating system creates the illusion of many virtual computers by transforming the single, physical computer. Figure 1.3 shows the structure.

Each virtual computer is similar to the physical computer in many ways but it is easier to use than the physical computer.

A computer has four basic parts: processor, primary memory, I/O, and secondary memory. Let us look at each of these in turn and see how our virtual computer differs from the physical computer.

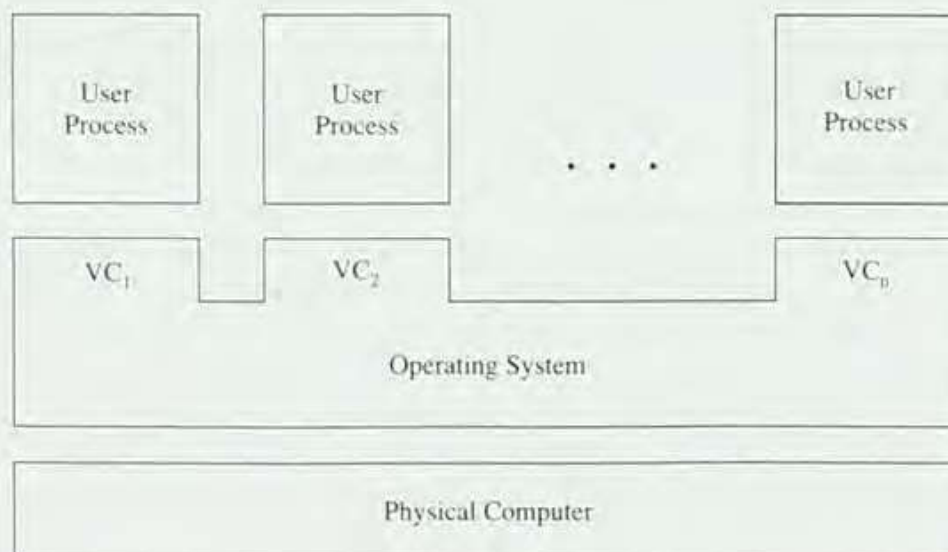


Figure 1.3 Multiple virtual computers in an operating system

¹Virtual memory systems time-multiplex memory.

Computing requires a processor and some memory.

1.3 A VIRTUAL COMPUTER

A virtual computer is a computer implemented in software using the hardware resources of the physical computer. It has the same four basic components (processor, memory, I/O, and disk) as the physical computer. Each virtual resource will be a transformed and multiplexed version of the physical resource. Figures 1.4, 1.5, 1.6, and 1.7 show the multiplexing of the four virtualized resources. Note that all of the multiplexing is done in the same way: the physical resource is broken up into discrete pieces and these pieces are put together to create the virtual resources. In some cases the pieces are spatial (space-multiplexing) and in some cases they are pieces of time (time-multiplexing).

In the next few sections we will describe the ways in which these resources are transformed into virtual resources.

Operating systems create virtual computers that can solve the same problems as a real computer, but that are easier to use.

1.3.1 VIRTUAL PROCESSOR

The virtual processor has nearly the same interface to the user as the physical processor, that is, it has nearly the same set of instructions. One reason for this is that it is implemented by using the physical processor directly. The operating system will rapidly switch the services of the physical processor between the virtual computers to implement the virtual processors.

The operating system removes some of the physical processor instructions and adds some other operations. The instructions it removes are the ones that control the physical resources of the computer (processor, memory, I/O). These are the instructions that affect access to memory, provide direct access to the devices, and change the protection state of the processor. In return for the instructions, the operating system adds instructions that allow the virtual processor to request virtual resources from the operating system. These instructions are called *system calls*. System calls allow a programmer to:

- Create new virtual computers;
- Communicate with other virtual computers;
- Allocate memory as needed;
- Do I/O; and
- Access a sophisticated file system.

system call

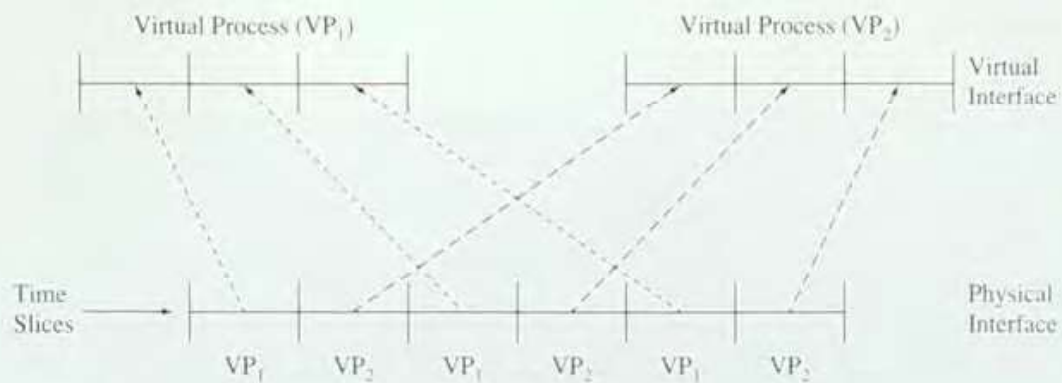


Figure 1.4 Multiplexing the processor by time-slicing

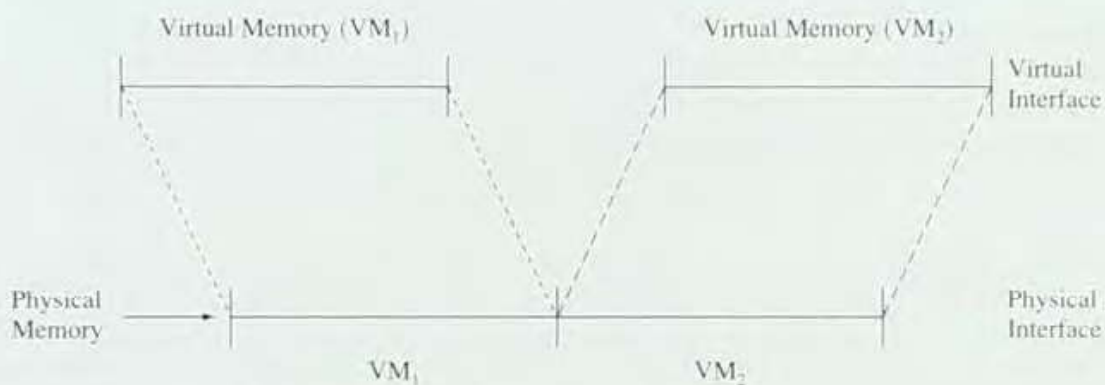


Figure 1.5 Multiplexing the physical memory by dividing it into pieces

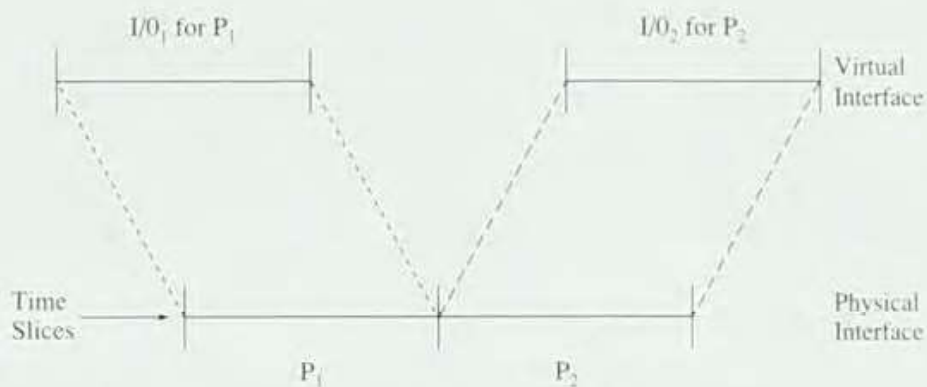


Figure 1.6 Multiplexing the I/O processor by sequential use

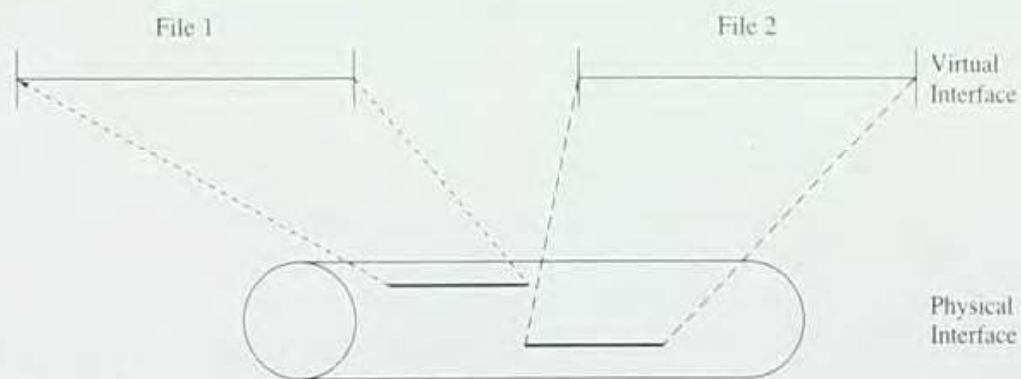


Figure 1.7 Multiplexing the disk by dividing it into pieces

Modern processors provide system call instructions precisely because operating systems need them to provide users with safe access to operating system services.

We can summarize the relationship in the equation (where P is a processor):

$$P_{\text{virtual}} = P_{\text{physical}} - \{\text{Dangerous instructions}\} + \{\text{Useful replacements}\}$$

The virtual processor does not change the instruction set of the physical processor very much since the main goal of the operating system is to create virtual copies of the processor. Creating an improved virtual instruction set is a complex job and is done by compilers and interpreters.

Modern processors make removing the resource control instructions easy by providing two processor modes: system and user. In user mode, the instructions that control the physical resources of the computer are not allowed. Processors have processor modes precisely because operating systems need them to provide a safe virtual processor. The operating system is managing the hardware resources of the computer and it must keep control of them. It allows the virtual computer to *use* these resources but not to *control* them.

1.3.2 VIRTUAL PRIMARY MEMORY

The memory of the virtual computer is very similar to the hardware memory, that is, it is a long sequence of cells with sequential, numerical names. The operating system will divide up the physical memory into parts and give a part to each virtual computer. The operating system usually creates, with help from the hardware, the illusion that the memory seen by each virtual computer is named with numerical names starting at 0. The operating system may also create the illusion that the virtual computer has more memory than the physical computer has physical memory. This is called virtual memory.

The programming language concept of an array is taken directly from this model of memory, so you are already familiar with the software emulation of physical memory, although you may not have realized it.

1.3.3 VIRTUAL SECONDARY MEMORY

Secondary storage provides long-term storage for data. This is done physically on disk blocks and virtually in disk files. The virtual computer sees a file system consisting of hierarchically named files that can be of any size and can be read and written in any size units. A considerable amount of operating system code is required to create this illusion.

1.3.4 VIRTUAL I/O

The I/O operations of the virtual computer are completely different than the I/O operations of the physical computer. The physical computer has devices with complex control and status registers. For example, the disks are basically large sequences of physical blocks with no structure. They can only be read or written in large blocks (of, say, 1024 bytes). Figure 1.8 summarizes the situation.

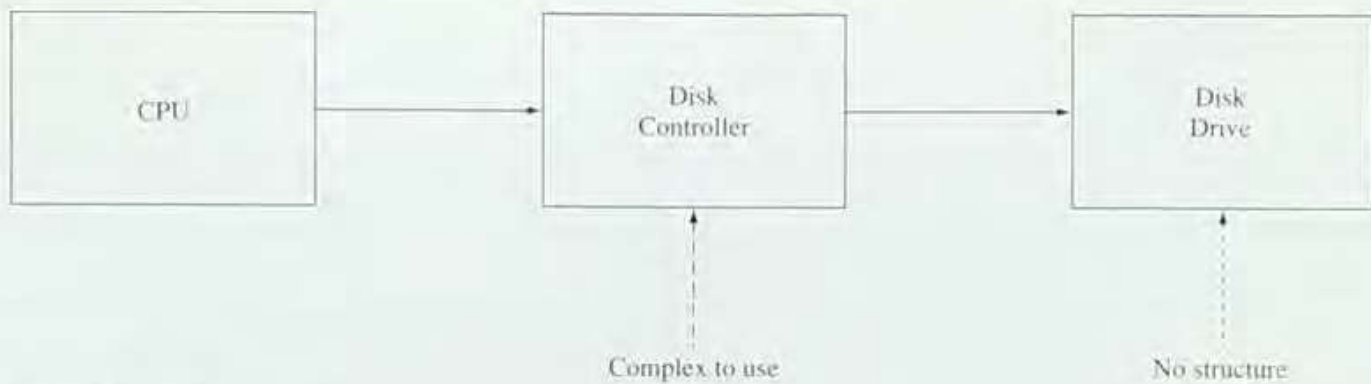


Figure 1.8 Physical I/O

By comparison, the virtual I/O provided by the virtual computer is simple and easy to use. We will see examples in Chapter 3.

You share a resource by dividing it up into pieces and distributing the pieces. These could be pieces of time or space.

1.4 DO WE NEED AN OPERATING SYSTEM?

It is not essential to have an operating system to use a computer. Some special-purpose computers do not have an operating system. For example, almost all microwave ovens have a small computer chip inside to control the oven, but few of the computer chips in microwave ovens have operating systems. Controlling a microwave oven is a very specialized task and it is the only task the computer chip is responsible for, so a full operating system would be wasteful and unnecessary. However, many of the tasks that are traditionally done in an operating system (managing devices, scheduling, pseudo-parallelism, etc.) would have to be done by the code in the microwave oven, so if you were designing such code you would want to be familiar with the techniques used in operating systems.

There are many examples of specialized computers that do not use an operating system but, in each one, the duties of the operating system are taken over by application code.

A general-purpose computer system must be able to run a wide range of programs. For such a system, an operating system is the most natural way to go. Otherwise each application would have to program its own operating system services. An operating system is useful because it contains a library of reusable services. The operating system implements many necessary features that are used by nearly every program. It is a good investment to implement them once, in the operating system, rather than in each application program.

METRIC PREFIXES

In all parts of computer science, and in the study of operating systems in particular, we often have to deal with very small numbers and very large numbers. The time of a memory access or instruction execution is getting shorter all the time and the size of disks and memories is getting larger. In this section we will show the metric prefixes that we will be using throughout the book.

First we will look at the small numbers.

Power of 10	Decimal Fraction	Prefix	Abbreviation
10^{-1}	0.1	deci	ds
10^{-2}	0.01	centi	cs
10^{-3}	0.001	milli	ms
10^{-6}	0.000001	micro	μ s
10^{-9}	0.000000001	nano	ns
10^{-12}	0.000000000001	pico	ps
10^{-15}	0.000000000000001	femto	fs
10^{-18}	0.000000000000000001	atto	as

In this book, we will only discuss small time intervals so the abbreviations are for fractions of a second. In the study of computer hardware we might be interested in nanometers and the like.

Note that the low powers of 10 have irregular names but, starting with 10^{-6} , they all end with "o".

This is because the first few prefixes have been in common use for some time but the higher ones were established by standards committees and so are regular.

The abbreviations all work out well except for the fact that millisecond and microsecond begin with the same letter and both would be candidates for being abbreviated as ms. Millisecond gets that honor and we use the greek letter mu (μ) in the abbreviation for microseconds (μ s).

Here are the large numbers.

Power of 10	Number	Prefix	Abbreviation
10^1	10	deka	Dbyte
10^2	100	hecto	Hbyte
10^3	1,000	kilo	Kbyte
10^6	1,000,000	mega	Mbyte
10^9	1,000,000,000	giga	Gbyte
10^{12}	1,000,000,000,000	tera	Tbyte
10^{15}	1,000,000,000,000,000	peta	Pbyte
10^{18}	1,000,000,000,000,000,000	exa	Ebyte

Again, they are regular starting with 10^6 . We will only use these large numbers to discuss numbers of bytes, so that is how we are showing the abbreviations.

1.5 SUMMARY

An operating system is the layer of software that runs directly on the hardware and provides support for other software running on the hardware. An operating system is implemented on the hardware interface and implements a virtual computer interface. The operating system interface is similar in basic functionality to the hardware interface but is easier to use. In addition, many virtual computers can run on a single physical computer. A user program runs as a process in a virtual computer.

We can share a resource by dividing up the times it can be used (time-division multiplexing) or we can divide up the resource itself (space-division multiplexing).

A hardware computer system has five types of hardware resources: processor(s), memory, I/O controllers, disk devices, and other input/output devices. A virtual computer provides software versions of each of these resources. A virtual processor can run programs and uses essentially the same instruction set as the hardware processor. Some hardware instructions (the ones that control the hardware itself) are not available on the virtual processor. The virtual processor has additional instructions, called system calls, which provide access to the services of the operating system. The virtual memory is used to store the user program and data. The operating system provides a separate virtual memory area for each virtual computer. Virtual input/output devices provide (indirect) access to the hardware devices. The disk devices are virtualized as a file system.

The function of an operating system can be viewed in two different ways. The operating system is a resource manager that manages (allocates and frees), multiplexes (provides multiple copies of), and transforms (makes easier to use) the hardware resources. The operating system is also a virtual computer manager that provides virtual computers for processes to run in.

An important reason for using an operating system is that it represents reusable software.

1.5.1 TERMINOLOGY

After reading this chapter, you should be familiar with the following terms:

- address space
- file system
- hardware interface
- hardware resource
- mechanism
- multiplexing
- operating system
- operating systems interface
- policy
- process
- resource management
- space-division multiplexing
- space sharing
- spooling
- system call
- time-division multiplexing
- time sharing
- transformation
- virtual computer
- virtual resource

1.5.2 REVIEW QUESTIONS

The following questions are answered in the text of this chapter:

1. What are the activities of resource management? What are the five main hardware resources in a computer system?
2. What are the advantages of virtualizing a hardware resource?
3. What do we mean by the transformation of a resource?
4. What are the four resources of a virtual computer?
5. Give a brief definition for the term *multiplexing*.
6. What is the advantage of multiplexing a resource?
7. What hardware instructions are removed from the virtual processor? Why are they removed? How are they removed? What are they replaced with?
8. Explain the difference between sharing a resource and multiplexing a resource.
9. What is a reason for not using an operating system on a computer system?
10. What do we call a millionth of a second? What do we call a billion bytes?

1.5.3 FURTHER READING

There are many good books on operating systems where you can get different viewpoints than you will find in this book. These include Silberschatz and Galvin (1994), Tanenbaum (1992), Stallings (1992), Finkel (1988), Bic and Shaw (1988), and Lister and Eager (1995).

1.6 PROBLEMS

1. A process implements a virtual computer. Consider the idea of running another operating system on the virtual computer, that is, running an operating system as a program on another operating system. There are two ways we could do this. Discuss each one in terms of practicality and efficiency.
 - a. You load the operating system code directly into the virtual computer (with no other additional software) and run it.
 - b. You first load an emulator into the virtual computer. The emulator creates a virtual version of some computer. Then you load the operating system to run on this emulator.
2. In this chapter we looked at level models. In software engineering we divide a system into modules. Compare the level model with the module model for dividing up systems.

3. A window system manages three hardware resources: the display, the mouse, and the keyboard. Discuss a window system as an operating system. What are the virtual resources? What things are multiplexed and how (space or time or both)? Are the resources transformed for easier use?
4. Imagine a program to control a microwave oven. What operating systems-like functions would it need? Resource management (what resources)? Transformation of resources? Multiplexing of resources?

chapter

2

The Hardware Interface

In this chapter we consider the hardware interface. Our goal is to introduce a simple computing system that we will use in the remainder of this text. We will use a simplified, generic RISC processor with some CISC elements. We call it the Composite RISC Architecture 1 (CRA-1). We will not describe the entire architecture, but only the parts that we will use in later chapters. We will only see a small part of the instruction set, for example.

Figure 2.1 presents an overview of the CRA-1. This system provides a *CPU* (also known as a *processor*) for executing programs, a *memory* for storing instructions and data during the execution of a program, and a *disk system* (disk controller and disk) for permanent storage of programs and data. These components are interconnected using a *system bus*.

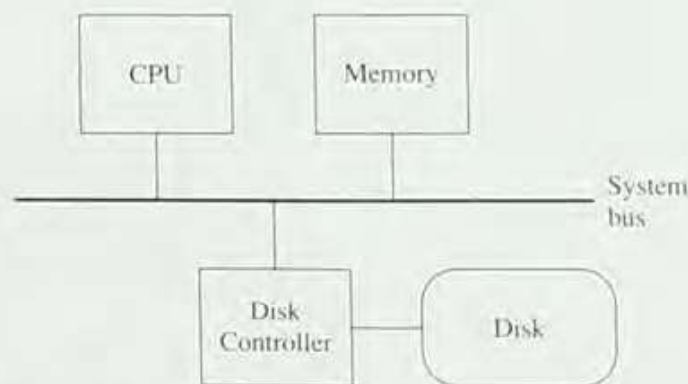


Figure 2.1 CRA-1 organization

2.1 THE CPU

2.1.1 GENERAL-PURPOSE REGISTERS

The CPU provides 32 *general-purpose registers*, each 32 bits long. In assembly language, these registers are referenced using the names *r0* through *r31*. As is common with RISC machines, *r0* always holds the value 0. The other registers are for general use, but a few have conventional uses in most programs:

cpu processor

general purpose register

- *r1*—return values from procedures.
- *r8*—first parameter to a function call (or system call).
- *r9*—second parameter to a function call (or system call).
- *r10*—third parameter to a function call (or system call).
- *r11*—fourth parameter to a function call (or system call).
- *r29*—the frame pointer.
- *r30*—the stack pointer.
- *r31*—the return address from a procedure call.

Procedure calls send their arguments in registers *r8*, *r9*, *r10*, and *r11*, and return values in *r1*.

Two registers are used to support programming language stacks. The stack register is used by the stack instructions. The stack can be used for procedure arguments beyond the four provided for in the registers. The frame pointer is used by the programming language to address items in stack frames.

2.1.2 CONTROL REGISTERS

The *control registers* hold values that control the execution of the processor. The control registers are all 32 bits long.

control register

- *ia*—The *instruction address register* contains the address of the next instruction. *pc*
- *psw*—The *program status word* has two significant bits. Bit 0 (the low-order bit) is 1 if the processor is in user mode, and 0 if the processor is in system mode. Bit 1 is 1 if interrupts are enabled, and 0 if interrupts are masked. *mode and interrupt*
- *base*—The memory *base register* is added to all addresses when the system is in user mode. *base register*
- *bound*—The memory *bound register* is the address limit. In user mode, all addresses must be less than the *bound register*; otherwise a program error interrupt (with *ip* of 2) will occur. The comparison is done before the *base register* is added. *bound register*
- *iaa*—The *interrupt instruction address register* stores the value of the *ia* register before an interrupt. When an interrupt occurs, the *ia* register is saved in the *iaa* register before the *ia* register is loaded with the address of the interrupt handler.

- *ipsw*—The *interrupt program status word* stores the value of the *psw* register before an interrupt. When an interrupt occurs, the *psw* register is saved in the *ipsw* register before it is loaded with 0.
- *ip*—The *interrupt parameter register* contains data about the last interrupt (if any is necessary).
- *iva*—The *interrupt vector address register* is the location in memory where the interrupt vector table is located.
- *timer*—The *interval timer register* is decremented once every microsecond unless it is already zero. When it is decremented from 1 to 0, a timer interrupt is generated. Storing a zero in this register clears any pending (but masked) timer interrupts.

2.1.3 PROCESSOR MODES

The processor can either be in *system mode* (if *psw* bit 0 is 0) or in *user mode* (if *psw* bit 0 is 1). In system mode, all instructions are legal and all addresses are physical addresses (see Section 2.2). In user mode, the instructions that modify the control registers are not legal and will cause a program error interrupt (with *ip* of 1). In user mode, all addresses must be less than *bound* and will have *base* added to them before memory is accessed.

Interrupts can occur only if *psw* bit 1 is 1.

2.1.4 INSTRUCTION SET

We will only specify a few instructions because that is all we will need for the purposes of this book. The instructions we will use are shown in Table 2.1.

In a load/store architecture, there are two main instructions that access memory, one to load a register from memory and another to store a register into memory. We will assume that the load and store instructions can load into and store from a control

Table 2.1 Some CRA-1 instructions

Operation	Instruction	Format
Load register	load	<i>address, rN</i>
Store register	store	<i>rN, address</i>
Load registers	loadAll	<i>address</i>
Store registers	storeAll	<i>address</i>
Move register	move	<i>rM, rN</i>
System call	syscall	
Return from interrupt	rti	

register as well as a general register.¹ For convenience we have an instruction that will store all the general registers to 32 words of memory and another instruction that will load all the general registers from 32 words of memory.² There is also an instruction to move one register to another. This instruction will work with general registers and control registers.³

The system call instruction is executed by a user process in order to request a system service. This instruction causes an interrupt which puts the processor into system mode and transfers to the system call interrupt handler in the operating system. The arguments to the system call (including the system call number) are assumed to be in the user's registers at the time of the system call.

Finally there is an instruction to return from an interrupt. We will discuss it in more detail in Section 2.3.

2.1.5 MACHINE INSTRUCTIONS IN C++ CODE

We will need to write short sequences of machine instructions at a few places in the book. Many C++ compilers allow the insertion of assembly language in an *asm block*. For example, the following code sets the timer to a new value.

```
SetTimer (int timerValue) { asm (move timerValue, timer) }
```

We will assume that we can put passed arguments in the assembly language code (as we did with `timerValue` in the above example) and the compiler will do the addressing correctly.

2.2 MEMORY AND ADDRESSING

The smallest addressable unit will be an 8-bit byte and addresses will be 32 bits long. When the processor is in system mode, all addresses are *physical addresses*, which means that the address generated by the instruction is sent directly to the memory by placing it on the system bus as a memory address. Physical addresses are also called *absolute addresses*.

When the processor is in user mode, all addresses are *logical addresses*. When an instruction generates a logical address, the processor will first check to be sure that the logical address is less than the *bound* register. If it is not, then a program error interrupt (with *ip* of 2) is generated. Otherwise, the *base* register is added to the logical address to generate a physical address. This physical address is then placed on the system bus as a memory address.

asm block

physical address

logical address

¹In an actual machine, this might be a different instruction or you might have to load into a general register and then move into a control register.

²Register *r0* will always have a value of 0 no matter what is loaded into it.

³In an actual instruction set, the instructions to move to and from control registers will probably be different than those to move between general registers. We will assume here that the assembler will work out these details.

The upshot of all this is that, in user mode, a program is restricted to a portion of the physical addresses. It cannot access any physical address less than the value of the *base* register (since it is added to every address) and it cannot access any physical address greater than (or equal to) the sum of the *base* and *bound* register (since this limit is checked for each logical address). This is a simple form of memory protection.

physical address space

memory address space

I/O address space

logical address space

The *physical address space* consists of all the physical addresses that a program can generate. It is all the addresses between 0 and 0xFFFFFFFF. The physical address space is divided up into the memory address space and the I/O address space. The *memory address space* is the part of the physical address space where we can place memory. In the CRA-1, the memory address space is from 0 to 0xEFFFFFFF. Only a part of the memory address space will have actual memory responding to those addresses. We will normally be assuming our CRA-1 has 16 megabytes (16M) of actual memory at addresses 0 to 0xEFFFFFFF. The *I/O address space* is the part of the physical address space reserved for I/O devices and consists of the physical addresses 0xF0000000 to 0xFFFFFFFF.

A *logical address space* is the range of addresses that a program running in user mode can use. This starts with 0 and goes up to the value of the *bound* register. Each logical address is mapped into a physical address when the *base* register is added to it.

2.3 INTERRUPTS

interrupt

An *interrupt* is an immediate transfer of control caused by an event in the system. There are several events that can cause an interrupt in the CRA-1.

1. *System call*—A system call instruction was executed.
2. *Timer*—The interval timer counted down to zero.
3. *Disk*—The disk device finished a transfer.
4. *Program error*—The program made an error. The *ip* register indicates exactly what happened.
 - *ip* = 0—an undefined instruction.
 - *ip* = 1—an illegal instruction in user mode. This is a move to control register instruction (which is not allowed in user mode).
 - *ip* = 2—a logical address greater than or equal to the *bound* register.

When an interrupt occurs, the CRA-1 does the following:

1. The *psw* register is saved in the *ipsw* register.
2. The *psw* register is set to 0.
3. The parameter of the interrupt (if it is a program error interrupt) is put in the *ip* register.
4. The *ia* register is saved in the *iia* register.
5. The interrupt number (system call = 0, timer = 1, disk = 2, program = 3) is multiplied by 4 and added to the contents of the *iva* register to get the interrupt vector slot. The address in that slot is loaded in the *ia* register.

First the processor saves the *psw* register in the *ipsw* register, and resets *psw* to put the processor in system mode with interrupts disabled (by storing a 0 in the *psw*). Then it stores the interrupt argument (if there is one) in the *ip* register. The processor then saves the *ia* register in the *iia* register, gets the address of the interrupt handler by looking in the appropriate slot in the *interrupt vector area* (pointed to by the *iva* register), and loads the address into the *ia* register. This completes the handling of the interrupt by the hardware, and normal instruction execution resumes. The address of the next instruction is taken from the *ia* register, which has just been set to the beginning of the interrupt handler for this interrupt.

interrupt vector area

The return from interrupt (*rti*) instruction reverses the effect of an interrupt. The *rti* instruction moves the contents of the *iia* register to the *ia* register, moves the contents of the *ipsw* register to the *psw* register, and then begins normal instruction execution. This will restart the program that was running when the interrupt occurred at the point it was interrupted. If no registers have been changed (or if they were saved and then restored just before the *rti*), then the process will continue on as if nothing had happened.

The *iva* register points to an area in memory which contains the addresses of the interrupt handlers. There is a slot in the interrupt vector area for each interrupt. The offset in the interrupt vector area depends on the interrupt number. A table of the slots is shown in Table 2.2.

Some interrupts can only occur when bit 1 of the *psw* register is 1. If it is 0, the interrupts are *masked*, meaning that the fact that they occurred is recorded but the interrupt is not actually processed until interrupts are unmasked by setting bit 1 of the *psw* register to 1. Only the timer and disk interrupts are masked; the program error and system call interrupts can occur even if bit 1 of the *psw* register is 0.

masked

If interrupts are enabled and both the timer and disk interrupts are pending, then the timer interrupt is taken first.

2.4 I/O DEVICES

The CRA-1 uses *memory-mapped I/O*, meaning that communication between the I/O devices and the processor is done through physical memory locations in the I/O address space. Each I/O device will occupy some locations in the I/O address

memory-mapped I/O

Table 2.2 Layout of the interrupt vector area

Offset	Interrupt
0-3	System call
4-7	Timer
8-11	Disk
12-15	Program error

space. That is, it will respond when those addresses are placed on the bus. The processor can write those locations to send commands and information to the I/O device, and read those locations to get information and status from the I/O device. Memory-mapped I/O makes it easy to write device drivers in a high-level language as long as the high-level language can load and store from arbitrary addresses.

2.4.1 DISK CONTROLLER

The CRA-I has one disk controller and it controls one fixed disk. Disk commands are initiated by storing them in the disk controller control register (see Figure 2.2). Disk blocks (also called sectors) contain 4096 data bytes. There are only two commands: load disk block from the disk into memory, and store disk block from memory to the disk. The disk block is specified by the disk block number, which is 20 bits long. This allows for about a million blocks on the disk, and a total disk size of 4 gigabytes.

The address to load into or store from in memory is specified by the disk memory address register, which should be loaded before the disk control register is loaded with a command.

When a disk transfer command is initiated, the disk controller goes into a busy state and remains busy until the transfer is completed. When the transfer completes, the disk controller can optionally cause a disk interrupt to inform the CPU that the disk is no longer busy. The `disk_status_reg` can be read to determine if the disk is busy.

```
const int BlockSize = 4096;

enum disk_command {LoadBlock=0, StoreBlock=1};

struct disk_control_reg {
    unsigned int command : 1;
    unsigned int interrupt_enabled : 1;
    unsigned int disk_block : 20;
    unsigned int padding : 10;
};

volatile disk_control_reg *Disk_control =
    (disk_control_reg *)0xF0000000;
void **Disk_memory_addr = (void **)0xF0000004;

enum disk_status {DiskIdle=0, DiskBusy=1};
struct disk_status_reg {
    unsigned int busy : 1;
    unsigned int padding : 31;
};

disk_status_reg *Disk_status = (disk_status_reg *)0xF0000008;
```

Figure 2.2 Declarations for the disk controller

2.5 SUMMARY

The CRA-1 hypothetical computer system has 32 general-purpose registers. Some have special purposes: always zero (*r0*), passing arguments (*r8*, *r9*, *r10*, and *r11*), stack (*r29*, *r30*), and procedure calls (*r1* and *r31*). It has nine control registers intended for use by the operating system. It has user and system processor modes. It is byte-addressed and uses 32-bit words and addresses. It has four types of interrupts: system call, timer, disk, and program error. And it has a disk and disk controller.

2.5.1 TERMINOLOGY

After reading this chapter, you should be familiar with the following terms:

- asm block
- base register
- bound register
- control register
- CPU
- general-purpose register
- I/O address space
- interrupt
- interrupt vector area
- logical address
- logical address space
- masked
- memory address space
- memory-mapped I/O
- physical address
- physical address space
- processor
- system mode
- user mode

2.5.2 REVIEW QUESTIONS

The following questions are answered in the text of this chapter:

1. What is the difference between a control register and a general-purpose register?
2. How do the base and bound registers control access to memory in user mode?

3. What happens when an interrupt occurs?
4. What happens when an *rti* is executed?
5. How do you specify machine instructions in C++?
6. What is memory-mapped I/O?

2.5.3 FURTHER READING

There are many books on computer organization and architecture that discuss in more detail the topics in the chapter. Some examples are Hamacher et al. (1990), Herzog (1996), Maccabe (1993), and Tanenbaum (1990).

The hardware reference manuals for some current RISC machines are also helpful. The major ones are the HP Precision (Hewlett Packard 1994) (<http://www.hp.com/nsa/acd.html>), the IBM RIOS (May et al. 1994) (<http://www.austin.ibm.com/tech>), and the SPARC (SPARC International 1992) (<http://www.prenhall.com/~rich/013/825000/82500-0.html>).

2.6 PROBLEMS

1. Why is there more physical address space than physical memory?
2. What is the alternative to memory-mapped I/O?
3. Why can't you mask off the system call and program error interrupts?
4. Show the code to read disk block 100 into address 0x2000.
5. Give the computation that shows that the disk controller can access a disk of up to 4 Gbytes.
6. Why is *rti* always 0? How is this useful?
7. What if a procedure has more than four arguments? What would you do?
8. Why aren't the base and bound registers used in system mode? What if they were used in system mode? Would this work? What other changes to the hardware would be necessary?
9. Why can't you change the control registers in user mode?
10. Most older processors required the interrupt vector area to be in the first part of the physical address space. Give a reason why it is better to have an *iva* register to determine the location of the interrupt vector area.
11. The interval timer only counts down to zero. Suppose we wanted to keep the time of day. How could we use the interval timer to keep the time of day?
12. Some processors save the *ia* register and the *psw* register on a stack when an interrupt occurs. What are the advantages and disadvantages of this?
13. Change the definitions of the disk controller to allow for up to four disks on a single disk controller. Then change the status register to allow it to return error codes when an operation fails.

chapter

3

The Operating System Interface

Before we look at how an operating system is implemented, we need to look at what functions will be implemented. So, in this chapter we will look at the interface the operating system provides to user programs. This interface is accessed with special machine instructions called traps or system calls. First we will look at exactly what system calls are and then we will look at the kinds of system calls that are provided by a typical operating system. We will see how files are used, how processes are created and destroyed, how processes communicate, and how shells provide interactive access to operating system functions.

3.1 WHAT ARE SYSTEM CALLS?

Almost all modern processors provide instructions that can be used as system calls. A system call instruction is an instruction that does not execute a specific function in hardware but instead generates an interrupt that causes the operating system to gain control of the processor. The operating system then determines what kind of system call it is and performs the appropriate service for the system caller. In our hardware the `syscall` instruction is used for system calls.

Figure 3.1 illustrates the flow of control during a system call. A system call goes through the following steps:

1. The program executes the system call instruction.

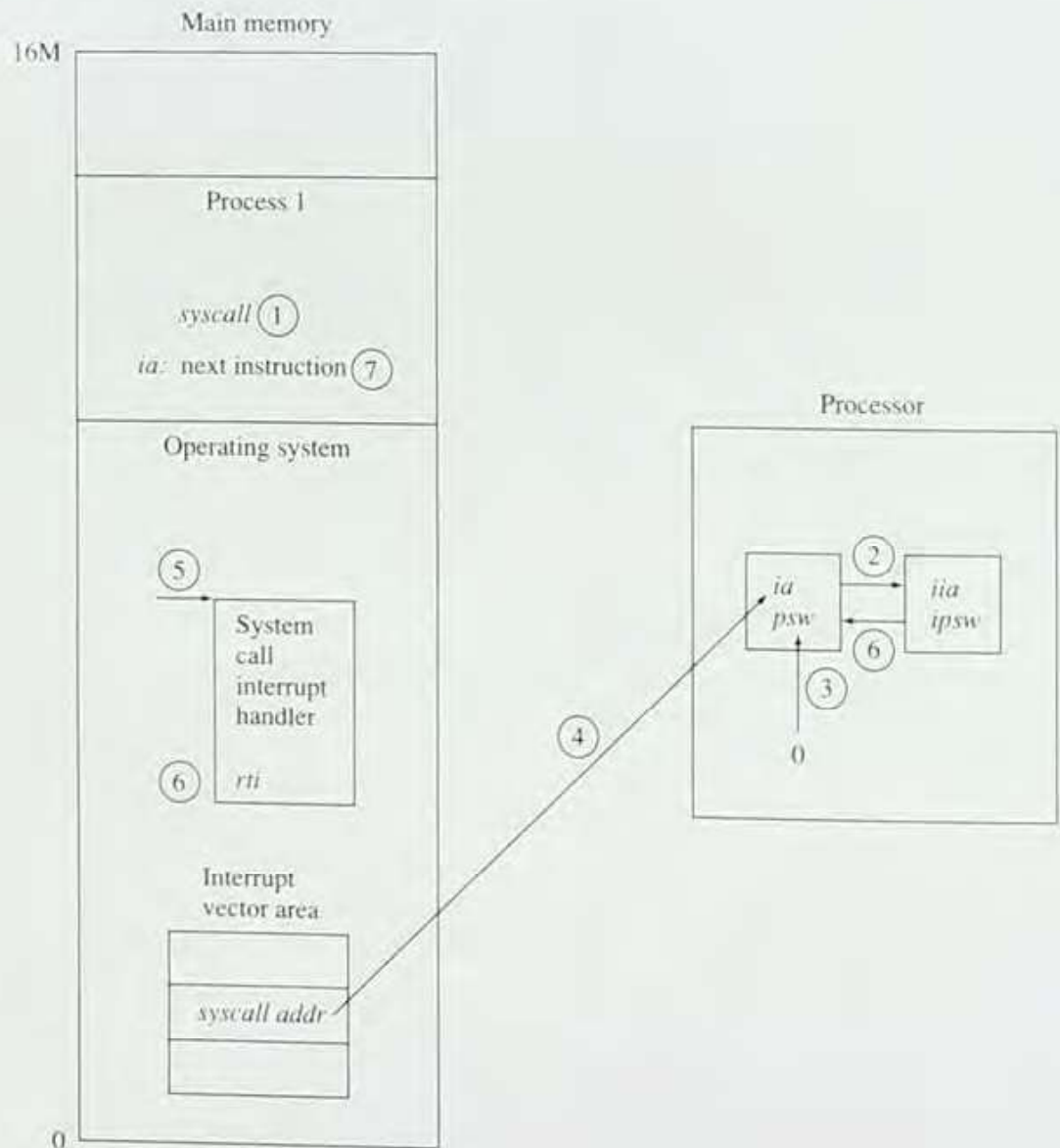


Figure 3.1 System call flow of control

2. The hardware saves the current *ia* and *psw* registers in the *iaa* and *ipsw* registers.
3. The hardware loads a value of 0 into the *psw* register, which puts the machine in system mode with interrupts disabled.
4. The hardware loads the *ia* register from the system call interrupt vector location. This completes the execution of the system call instruction by the hardware.
5. Instruction execution continues at the beginning of the system call interrupt handler (whose address was in the system call interrupt vector location).
6. The system call handler completes and executes a return from interrupt (*rti*) instruction. This restores the *ia* and *psw* from the *iaa* and *ipsw*.
7. The process that executed the system call instruction continues at the instruction after the system call.

The system call appears to be a single instruction to the caller but is implemented by many instructions in the operating system. It is similar to a procedure call instruction except that:

- The address of the procedure is not given; instead, an integer system call number is given.
- The processor mode changes from user mode to system mode.

The system caller must provide an integer argument, which is the system call number. This is usually used by the operating system as an index into a system table that determines what code is actually branched to. Since a system call is a switch to the operating system, the processor mode is changed to system mode.¹

The system calls are the instruction set of the operating system virtual processor.

3.1.1 HOW TO MAKE A SYSTEM CALL

A system call is made using the system call machine language instruction. Few compilers will allow you to make system calls directly, so they must be made from an assembly language program. In C++ you use a library of assembly language routines that make system calls for you. The library is accessed through C++ procedure calls.

Since our hypothetical C++ compiler allows in-line assembly language, we can show how to make system calls directly from C++. The following code shows how such a system call library routine would look.

¹Actually this is not necessarily so. It is possible to execute a system call while in system mode. In this case, the processor mode does not change. However, it is unusual to execute a system call in system mode.

MAKING A READ SYSTEM CALL

```
void open( char * file_name ) {
    asm (
        "load  ReadSystemCallNumber,%r8\n"
        "move  file_name,%r9\n"
        "syscall\n"
    );
}
```

The arguments are passed in registers. The return value of the system call will be passed back in register 1, which is also the register used for the return value for C++ procedures. Thus the value returned by the system call is passed back to the C++ caller.

3.1.2 WHAT IS A SYSTEM CALL INTERFACE?

The operating system provides a set of operations which are called system calls. These extend the native hardware instructions. All services that the operating system provides can be requested through system calls. From the point of view of a user, the entire functionality of the operating system is defined by the system calls. A *system call interface* is the description of the set of system calls implemented by the operating system.

system call interface

3.2 AN EXAMPLE SYSTEM CALL INTERFACE

For the purposes of this chapter we will define a simple system call interface to be used as an example. This interface is quite similar to a subset of the UNIX system call interface. Many of these calls will run unchanged on a UNIX system. We will generally *not* mention instances where the UNIX interface provides additional functions beyond what we define in this interface. You can see a complete list of the UNIX system calls by executing the UNIX command `man 2 intro` (on BSD-derived systems) or `man -s 2 intro` (on SVR4-derived systems).

3.2.1 SYSTEM CALL OVERVIEW

We will start with a brief overview of all the system calls and then describe each of them in some detail.

File and I/O System Calls:

Open—get ready to read or write a file.

Creat—create a new file and open it.

Read—read bytes from an open file.

Write—write bytes to an open file.

Lseek—change the location in the file of the next read or write.

Close—indicate that you are done reading or writing a file.

Unlink—remove a file name from a directory.

Stat—get information about a file.

Process Management System Calls:

CreateProcess—create a new process.

Exit—terminate the process making the system call.

Wait—wait for another process to Exit.

Fork—create a duplicate of the process making the system call.

Execv—run a new program in the process making the system call.

Interprocess Communication System Calls:

CreateMessageQueue—create a queue to hold messages.

SendMessage—send a message to a message queue.

ReceiveMessage—receive a message from a message queue.

DestroyMessageQueue—destroy a message queue.

3.2.2 HIERARCHICAL FILE NAMING SYSTEMS

Before we look at the file-related system calls, let us look briefly at how files are named. All operating systems need a way to name files. Most operating systems use what are called hierarchical file names, where file names are structured as a tree with the internal nodes being file directories. In this section we will briefly present the idea of hierarchical file names. We will discuss this topic again in Chapter 16.

A basic concept in a hierarchical file naming system is a directory. A *directory* is a collection of names, each referring to a file or another directory. If a name leads to another directory, then we can use that directory as a collection of names, each referring to a file or another directory. This creates a tree structure where leaf nodes are files and interior nodes are directories. There is a special directory that is the root of the tree and it is called the root directory. Figure 3.2 gives an example of a *hierarchical file naming system*.

Each directory is a collection of names, and all the names in a directory are unique. A directory forms a name space, and a directory also includes the mapping from each name to the file or directory it refers to. So, given a name and a directory, you can find the file or directory the name refers to. This is called a *name lookup*.

Files in a hierarchical file system are named with structured, multipart file names. A file name represents the path from the root directory to the file being named. A special character is used to separate the names in the path. In the example file hierarchy in Figure 3.2, the name of the directory `ch1` is `/users/faculty/crowley/book/ch1`.

directory

hierarchical file naming
system

name lookup

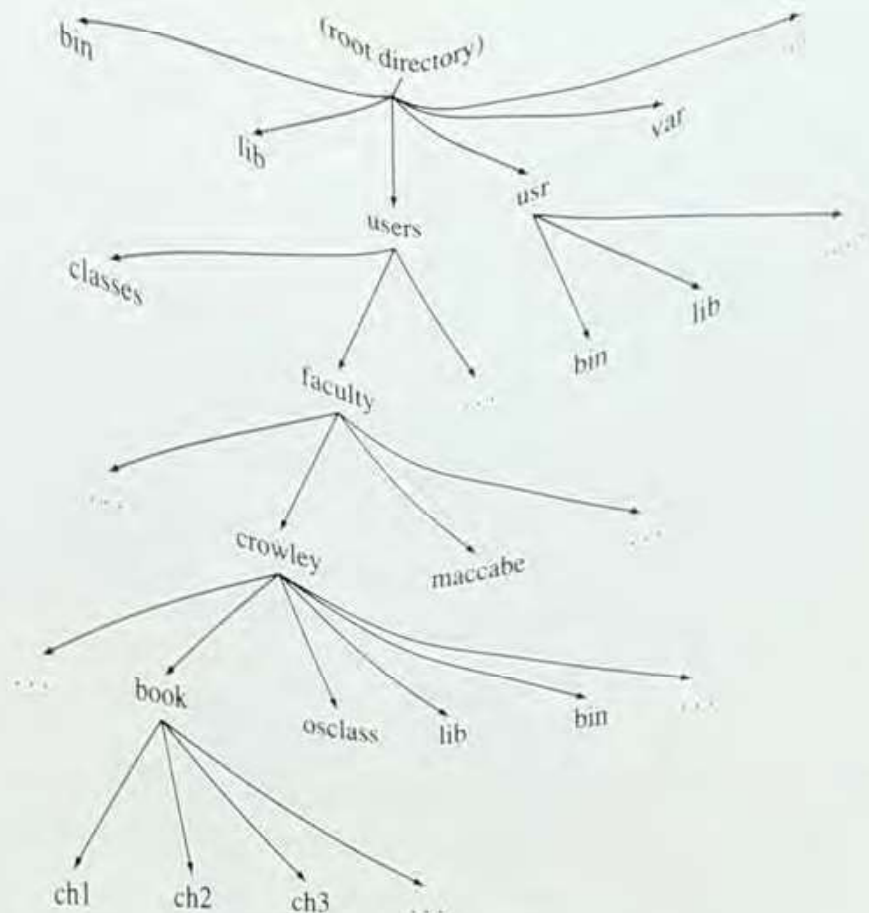


Figure 3.2 Example of a hierarchical file system

Since we see only one `ch1` in the directory tree, you know which file we mean when we say `ch1`, but if we were to refer to directory `bin` we could mean any of three directories (`/bin`, `/usr/bin`, or `/users/faculty/crowley/bin`). We will call the path names with separators (like `/usr/bin`) *full path names*. Each part in the full path name (like `usr` and `bin`) will be called a *component name*. Each component name is looked up in a directory.

Almost all modern operating systems use hierarchical file names because they have many advantages over the most common alternatives, which are a flat name space or a two-level name space. If the root directory could contain only file names and not directory names, you would have a *flat name space*. A *two-level name space* is a hierarchy that is limited to only two levels, so only the root directory can contain other directories.

full path names
component name

flat name space
two-level name space

Most operating systems provide hierarchical file names because they are easy to understand and powerful to use.

3.2.3 FILE AND I/O SYSTEM CALLS

The following table summarizes the system calls for file operations. We assume a hierarchical file system like the one used in UNIX and most other operating systems.

Call	Parameters	Returns	Notes
<i>open</i>	name, flags	fid	Create open file connected to a file
<i>creat</i>	name, mode	fid	Create file and connect to open file
<i>read</i>	fid, buffer, count	count	Read bytes from open file
<i>write</i>	fid, buffer, count	count	Write bytes to open file
<i>lseek</i>	fid, offset, mode	offset	Move position of next read or write
<i>close</i>	fid	code	Disconnect open file from file
<i>unlink</i>	name	code	Delete the named file

In order to use a file, you first need to ask for it by name. This is called *opening* the file. Opening a file creates an operating system object called an *open file*. The open file is logically connected to the file you named in the open system call. What this means inside the operating system is that the open file data structure has a pointer to the data structure that represents the file you opened.

open file

An open file has a *file location* associated with it. It is the offset in the file where the next read or write will start.

file location

After you open a file, you can read or write the open file. When you are done using the open file, you close it.

Here is a complete specification of the file-related system calls:

- `int open ( char * pathName, int openFlags );`—The named file is opened and an integer, the *open file identifier*, is returned. The file location is initialized to 0. The flags argument can be one of:

open file identifier

0—open for reading.

1—open for writing.

2—open for reading and writing.

If an error occurs in trying to open the file, a negative integer is returned. The possible error codes are:

−1—the file path name was invalid.

−2—the `openFlags` argument is out of range (not 0, 1, or 2).

−3—the file is not readable and you opened for reading.

−4—the file is not writable and you opened for writing.

- `int creat ( char * pathName, int fileMode );`²—The named file is created as an empty file and opened for writing, and an integer, the open file identifier, is returned. The file location is initialized to 0. The `fileMode` argument is the file protection mode and is not used in this operating system because we have not defined file protection modes. If some error occurs in trying to create the file, then `creat` will return a negative integer to indicate the error. The possible error codes are:
 - 1—the file path name was invalid.
 - 2—the `fileMode` argument is invalid.
- `int read( int openFileID, char * bufferAddress, int count );`—The operating system tries to read `count` bytes from the open file designated by `openFileID`. The bytes are read from the file starting at the offset provided by the file location. If fewer than `count` bytes are left before the end of the file, all the remaining bytes will be read. This is not considered to be an error. The `openFileID` must be one returned by an `open` call (with `openFlags` of 0 or 2). The bytes read are placed into the array of bytes starting at `bufferAddress`. The file location is increased by the number of bytes read. The return value is the number of bytes that were actually read. This might be less than `count` if the file location is very near the end of the file. A return value of 0 indicates that the end of the file has been reached (the file location is equal to the size of the file) and no more bytes can be read. A negative return value indicates that some error occurred in the read. The possible error codes are:
 - 1—the `openFileID` is invalid (does not represent an open file).
 - 2—the `bufferAddress` is an invalid address.
 - 3—the `count` is invalid.
 - 4—reading is not allowed on this open file.
 - 5—I/O error.
- `int write( int openFileID, char * bufferAddress, int count );`—The operating system tries to write `count` bytes to the open file designated by `openFileID`. The bytes are written to the file starting at the offset provided by the file location. The `openFileID` must be one returned by a successful `open` call (with `openFlags` equal to 1 or 2) or a `creat` call. The bytes are taken from the array of bytes starting at `bufferAddress`. The file location is increased by the number of bytes written. The return value is the number of bytes that were actually written.³ A negative return value indicates that some error occurred in the write. The possible error codes are:
 - 1—the `openFileID` is invalid (does not represent an open file).

²The final “e” is left off `create` just because UNIX does it that way. We are following the UNIX spelling so that the examples will run under UNIX. We are not sure why the “e” was left off in the name of the UNIX system call—possibly to keep all the basic system call names to five characters or less.

³For all practical purposes this will be `count`. The only way the bytes could not be written is if the write encountered the end of the medium or the disk it was writing on filled up before it could write all the bytes.

- 2—the `bufferAddress` is an invalid address.
- 3—the `count` is invalid.
- 4—writing is not allowed on this open file.
- 5—I/O error.

- `int lseek( int openFileID, int offset, int moveMode );`⁴—The file location associated with the open file is changed to a new value. The new value is computed by adding the `offset` to the base. The base is selected according to the `moveMode` as described below. The legal values for `moveMode` are:

- 0—use 0 as a base (move from the beginning of the file).
- 1—use the current file location as a base (for relative moves).
- 2—use the current file size as a base (move from the end of the file). Positive values of `offset` will cause the next write to be beyond the current end of the file. A 0 value will cause the next write to extend the file.

The returned value is the new file location. An `lseek` call with an `offset` of 0 and a `moveMode` of 1 will not change the file position but will return the current file position. This is how you query the current file position. A negative return value indicates that an error occurred in the seek. The possible error codes are:

- 1—invalid file location (negative).
- 2—invalid `openFileID` (not an open file).
- 3—the `moveMode` is invalid (not 0, 1, or 2).

- `int close( int openFileID );`—The file is closed. The `openFileID` must have been obtained from either `open` or `creat`. A return value of 0 means the close succeeded. A return value of -1 means that the `openFileID` did not refer to an open file.
- `int unlink( char * pathName );`—The file is deleted.⁵ A return value of 0 means the unlink succeeded. A return value of -1 means that the `pathName` could not be found.

The open file identifiers returned by `open` and `creat` are local to the process making the call. That is, open file identifier 3 in one process is not related to open file identifier 3 in another process.

One thing to note is that the `read` system call specifies which file to read, how much to read, and where to put the read characters, but it does not indicate which characters in the file to read (where in the file to read). The location in the file is not passed with the calls but is kept with the open file information as part of the file state.

⁴The “l” in `lseek` is again for compatibility with UNIX. When UNIX was a 16-bit system, there was a `seek` system call that was more complex than this one. Later it was changed to use 32-bit (long) integers and so added the new system call `lseek` (long seek).

⁵This is called `unlink` in UNIX because of how directories are implemented. Directories in UNIX contain only the file name and a pointer to the file descriptor. The `unlink` system call in UNIX only removes a file name from a directory and decrements the link count in the file descriptor. The last unlink of a file reduces the link count to 0 and the file itself is deleted as a side effect. We will use the UNIX name, but in this operating system this instruction deletes the file itself.

The file location starts at 0 and is moved whenever bytes are read from (or written to) the file. Thus the file is read (or written) sequentially by default. If the file was opened for update (`openFlags = 2`), then reads and writes can be intermixed and they both affect the file position.

The `lseek` system call is used to achieve random access into the file since it changes the file location that will be used for the next read or write.

Figure 3.3 shows a flow chart for using a file.

3.2.4 OPEN FILES

file

It is important to understand the difference between a file and an open file. A *file* is a passive container of data, a named sequence of bytes. The principal operation on an existing file is the `open` system call, which returns an open file. The `creat` system call creates an empty file and then opens it for writing. An *open file* is a dynamic object that can provide bytes from the file or accept bytes to be stored in the file (or both). The operations on an open file are the `read`, `write`, `lseek`, and `close` system calls. An open file is generally associated with a file and allows access to the file. Later we will generalize the notion of an open file and see how an open file can be associated with other things than a file, such as input or output devices.

In Figure 3.4 we show how the `open` system call creates the open file and connects it to the file being opened. The `read` system call gets data from the file.

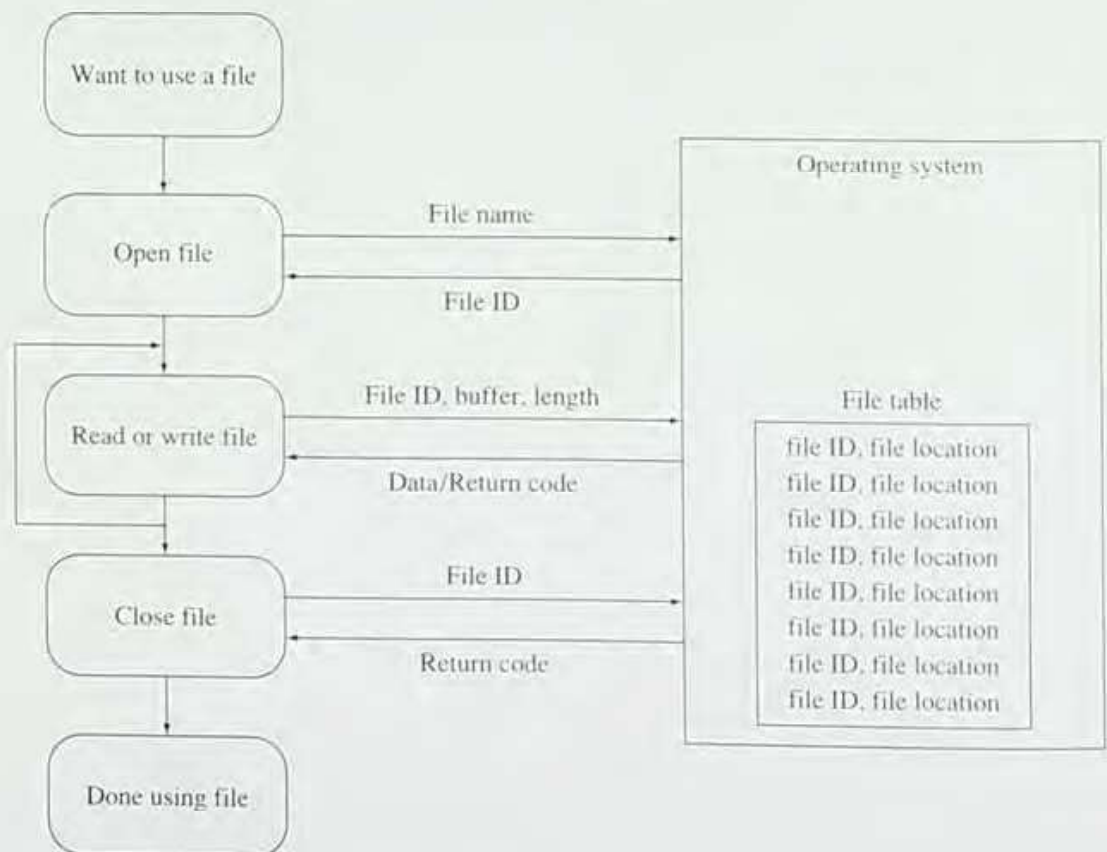


Figure 3.3 Steps in using a file

through the open file, and copies it into the I/O buffer, in the process making the read system call.

Files are passive containers of data. Open files are active sources and sinks for data.

Figure 3.5 shows the relationship between process objects (see Section 3.6), open file objects, and file objects. These three objects (processes, open files, and files) are different types of objects with different operations, but they are logically connected to each other because processes have open files and open files are connected to files.

Operating systems implement objects for users. The system calls create and operate on operating system objects.

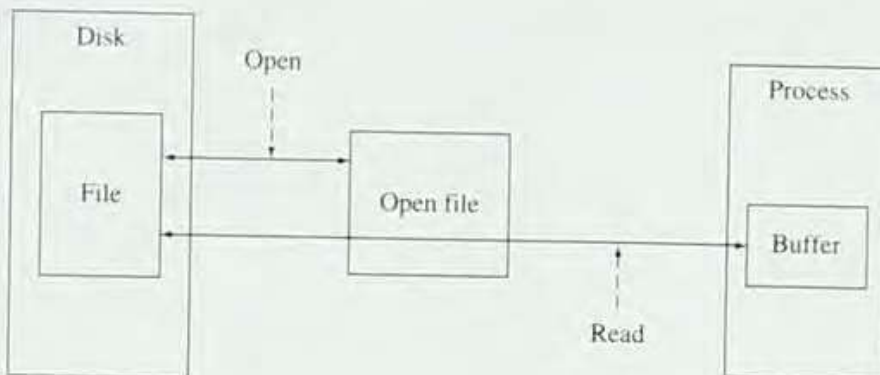


Figure 3.4 Files and open files

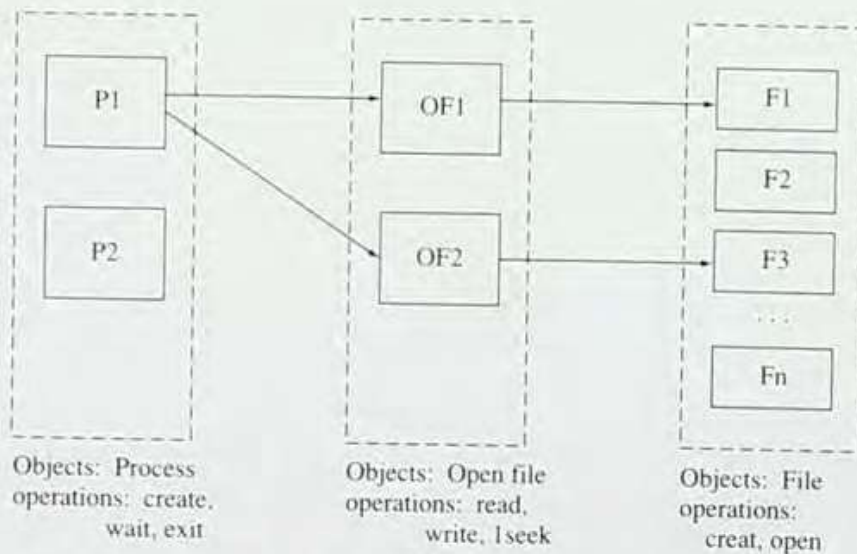


Figure 3.5 Objects and operations in an operating system

3.2.5 EXAMPLES OF FILE I/O

These system calls make it very easy to work with files. Here is the code to do a file copy.

FILE COPY

```
enum { Reading=0, Writing=1, ReadAndWrite=2, ReadWriteFile=0644 };
// This procedure takes as arguments two file path names.
// The first file is copied to the second file.
void FileCopy( char * fromFile, char * toFile ) {
    // open the source file for reading
    int fromFD = open( fromFile, Reading );
    if( fromFD < 0 ) {
        cerr << "Error opening " << fromFile << endl;
        return;
    }
    // create the destination file with protection mode read and write
    int toFD = creat( toFile, ReadWriteFile );
    if( toFD < 0 ) {
        cerr << "Error opening " << toFile << endl;
        close( fromFD );
        return;
    }
    // This loop reads one character at a time from the
    // source file and writes it to the destination file.
    while( 1 ) {
        char ch;
        int n = read( fromFD, &ch, 1 );
        // if 0 characters were read we are at the end of file
        if( n <= 0 )
            break;
        // throw away the value returned by write
        n = write( toFD, &ch, 1 );
        if( n < 0 ) {
            cerr << "Error writing " << toFile << endl;
            return;
        }
    }
    // close the two files and return
    close( fromFD );
    close( toFD );
}
```

We read and write one character at a time so the "array" of characters in the buffer is only a single character. We have to pass the address of the character because the system call expects the address of the first character of the buffer. The

variable `ch` is being used as an array of one character. We could have declared it as `char ch[1]`.

Basic I/O is simple: open, read/write/seek repeatedly, then close.

The `Reverse` program will copy a file and reverse it while copying. It demonstrates the use of `lseek`.

REVERSE

```
enum { Reading=0, Writing=1, ReadAndWrite=2 };
enum { SeekFromBeginning=0, SeekFromCurrent=1, SeekFromEnd=2 };
// This procedure reverses the bytes in a file and writes the
// reversed version in a new file.
void Reverse( char * fromFile, char * revFile ) {
    // open the source file
    int fromFD = open( fromFile, Reading );
    if( fromFD < 0 ) {
        cerr << "Error opening " << fromFile << endl;
        return;
    }
    // move the internal file pointer so the next character read will
    // be the last character of the file
    int ret=lseek( fromFD, -1, SeekFromEnd );
    if( ret < 0 ) {
        cerr << "Error seeking on " << fromFile << endl;
        close( fromFD );
        return;
    }
    int revFD = creat( revFile, 0 );
    if( revFD < 0 ) {
        cerr << "Error creating " << revFile << endl;
        close( fromFD );
        return;
    }
    // Loop while reading from the end of the source file to
    // the beginning. The loop is exited in the middle.
    while( 1 ) {
        char ch;
        int n = read( fromFD, &ch, 1 );
        if( n < 0 ) {
            cerr << "Error reading " << fromFile << endl;
            return;
        }
        n = write( revFD, &ch, 1 );
        if( n < 0 ) {
```

```

cerr << "Error writing " << revFile << endl;
return;
}
// exit the loop if lseek returns an error.
// The expected error is that the computed offset will
// be negative.
if( lseek(fromFD, -2, SeekFromCurrent) < 0 )
    break;
}
close( fromFD );
close( revFD );
}

```

We start at the end of the file by setting the file position to one less than the end-of-file file position. (Remember that the system calls read and write the *next* byte at the file position.) Each time we read a character we move the file position back two, one for the byte we just read (which moved the file position up one) and one more so that we back up one byte.

Figure 3.6 shows the process of reading a byte from the source file, writing it to the destination file, and backing up the file pointer by two bytes.

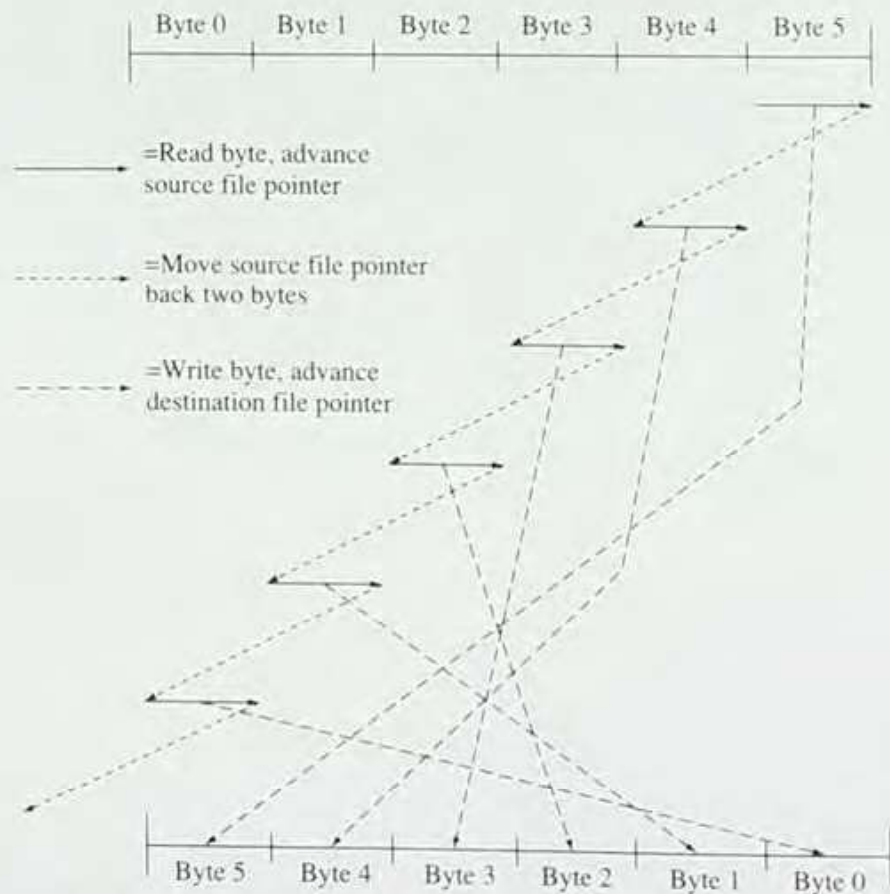


Figure 3.6 Copying and reversing a file

DESIGN TECHNIQUE: INTERFACE DESIGN

The reverse program requires an `lseek` in every iteration of the loop. For this particular program, it would be better to change the interfaces to `read` and `write` to include, as an explicit parameter to the call, the file offset at which to read or write. For example, the call might look like this: `read(fromFD, offset, &ch, 1)`. This would save one system call for each iteration of the loop and also express our intentions more clearly.

When we design an operating system (or any program, for that matter) we design two things: the interface and the implementation (for an operating system, the interface consists of the system calls and their arguments). When designing an interface it is useful to consider several design alternatives. As in any interface design, we have to consider who will be using the interface and how they will be using it. The original interface (with an implied file offset and a separate `lseek` to set it) is optimized for sequential reading and writing. For sequential file access, no `lseek` calls are required. The new interface proposed here (with explicit offsets in each read and write) is optimized for random access to the file. Experience tells us that sequential access is much

more common, and so we are led to accept the original design.

But can't we support both interfaces? We could have two sets of `read` and `write` system calls. One set would be the original set and one set would have explicit file offsets. We can do this, but this approach has other costs: more for users to learn, more decisions for users, possible confusions and mistakes by mixing up the two kinds of calls, longer manuals, and more code to implement the two sets of calls. It is better for the designers to make the decision.

You should follow this pattern in all interface designs.

- Determine who will use the interface and how they will use it.
- Think of several possible interfaces.
- Evaluate each alternative based on user and usage information.

This analysis is inexpensive, gives you a chance to examine your assumptions, and leads to improved interfaces.

See Section 4.5 for a longer discussion of interface design issues.

3.3 INFORMATION AND META-INFORMATION

Let us look at files in a more general way. Information is generally not a pure, complete, self-describing entity. Usually we want to record information about the information, that is, *meta-information*. A good example is a file. The file contains information itself, but the operating system also keeps meta-information about the file:

meta-information

- Who owns the file.
- Who can use the file and how.
- When the file was created.
- When the file was last modified.
- When the file was last read.
- How long the file is.

There is a system call that will return to you all the meta-information about a file:

- `int stat( int fileName, StatStruct * statInfo );`⁶ — The filename is looked up in the file system and the meta-information about the file is returned in the structure `statInfo`, which must be a pointer to a `StatStruct`. A return code of 0 means the `stat` succeeded and the information was written into `statInfo`. A return code of -1 means the file could not be found and no status information was returned.

We will expand on the contents of `StatStruct` in Chapter 16.

The `ls` command in UNIX (DIR in MS-DOS) can be used to display any of the information that the `stat` call returns. The `ls` command has many command line options since `stat` returns so much information. The command `man stat` will return a description of the meta-information kept in a UNIX file descriptor.

Most of the meta-information about a file is set by the operating system as a side effect of other operations. For example, when you modify a file the date of last modification is set. There is one type of meta-information that the owner of the file can set directly, and that is the protection status of the file, that is, who can use the file and how. There is usually a system call to do this. In UNIX, the system call is `chmod` (change mode).

Information is data. Meta-information is information about the data.

3.4 NAMING OPERATING SYSTEM OBJECTS

In Section 3.2.2, we discussed how the file system uses hierarchical names for naming files and directories. We want to generalize that idea a little bit so we can use the file naming system for other kinds of operating system objects.

In a file system, the names map into either files or directories. A file is an object that you then do some operation on (usually open), while a directory is another name map that you use for the next step of the file name lookup.

So far, we have used the file system to name files and directories. In practice, the names in a file system really map into file object descriptors on disk, and these file object descriptors point to the file objects themselves (see Figure 3.7). We can easily extend this file naming system to name other kinds of operating system objects as well. When a user or program needs to refer to an object, it needs a name for that object. So it is convenient to use the file name naming system to name all the objects that need names. In Figure 3.7, we have shown the file naming system naming a number of different kinds of objects, some of which we haven't yet discussed (processes and queues).

⁶The name `stat` is a shortening of "status."

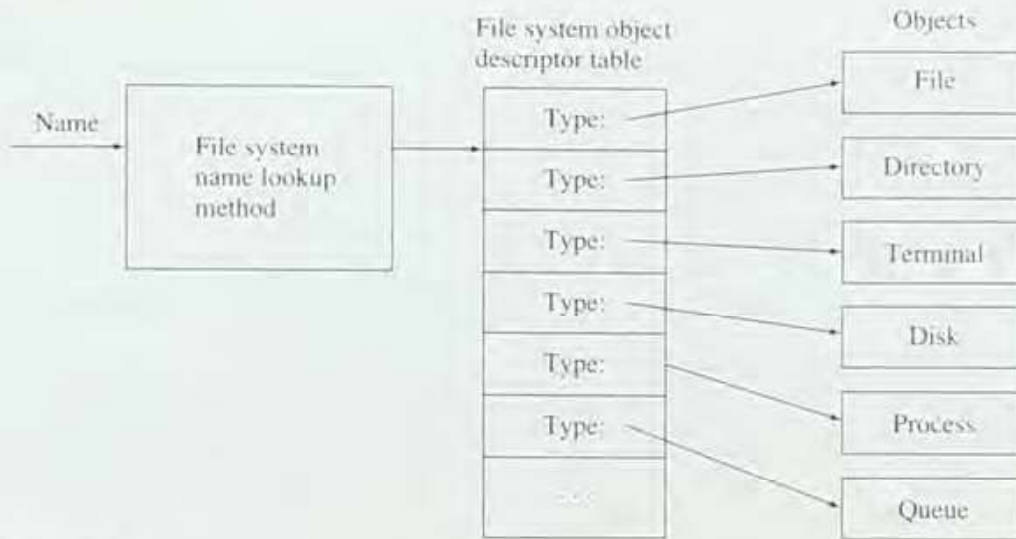


Figure 3.7 File system names mapping to objects

Name spaces map names into objects. File naming systems can map names into many different kinds of objects.

3.5 DEVICES AS FILES

We have seen how the operating system provides two related concepts for dealing with files. A *file* is a collection of information on disk, and an *open file* is an object (returned by an `open` or a `creat`) that you can read data from or write data to. So we can think of an open file as a source of bytes or a sink for bytes.

But objects other than files can also provide data and accept data. For example, the keyboard on a terminal or workstation is a source of bytes (keystrokes), and a terminal screen or a window on a workstation can accept bytes (characters) and display them on the screen. In fact, all I/O devices either receive data or send data or both. Given this similarity between devices and open files, it is useful to try to unify the concepts of file and device in an operating system. Let's see how that is done.

Every file has a name in the hierarchical file naming system. Since devices need names also, we will give them names in the file name hierarchy too. We will add special entries in the file system for the devices. In UNIX, there is a specially named directory, `/dev`, where most devices names are kept.⁷ For example, a terminal attached to the computer might have the name `/dev/tty13`.⁸ If we open

⁷This is just a convention in UNIX, not a requirement. Device files can legally occur in any directory but, in practice, it is rare to have a device file that is not in `/dev` or a subdirectory of `/dev`.

⁸Terminals are often called "ttys" for historical reasons. The first terminals were called teletypes, and that was abbreviated as tty.

the “file” name `/dev/tty13`, we get an open file object back. When we read from that open file, we are really reading from the keyboard of the terminal. When we write to that open file, we are really writing to the screen of the terminal.

For example, suppose we are sitting in front of a terminal whose name in the file system is `/dev/tty17`. The command

```
copy fileToList /dev/tty17
```

will list the file `fileToList` on the terminal `/dev/tty17` and the command

```
copy /dev/tty17 fileToRewrite
```

will take input from terminal `/dev/tty17` and write it into the file `fileToRewrite`. A program does not need to know whether it is reading from (or writing to) a file or a device. This is called *device independence*.

device independence

We can do the same thing with all devices: tape drives, floppy disks, terminals, communication lines, etc. We can even do it with windows on a workstation screen. Windows are sometimes called *pseudodevices* since they act like devices (terminals in particular) but are not real, physical devices.

Unifying devices and files makes programs device independent.

3.5.1 UNIFICATION OF THE FILE AND DEVICE CONCEPTS

The file system has two distinct parts to its services. The first part is the file naming system, which maps hierarchical names into objects (files). There is no particular reason that a name has to refer to a file since it just maps to some object. Most operating systems take advantage of that fact and use the file naming system to provide names that users can use to refer to a number of different objects. In particular, you can use the file naming system to give names to I/O devices. Later we will see the file naming system used to name other machines, processes, interprocess communication mailboxes, etc. This technique does not unify devices and files but merely uses the same system to name them.

The second part of the file system provides access to (that is, reading and writing of) files, and treats a file as a source or a sink for data. We have seen in this section how the file concept can be generalized to include devices. By unifying the concepts of file and device, we make things easier for users. A program does not need to know whether it is writing to a device or a file, so programs are simpler.

3.6 THE PROCESS CONCEPT

We previously discussed how the operating system creates a number of virtual computers. When we execute a program on one of these virtual computers, we call the executing program a *process*. The principal goal of an operating system is to implement virtual computers and the processes that run on them. A virtual computer is created

process

for each process, thereby giving the process the illusion of running on its own physical computer. The virtual computer extends the hardware instruction set with a set of system calls that request operating system services. The virtual computer is the execution environment created by the operating system, and the process is the program execution that occurs in that environment.

Many users are not aware of the process concept because they just run a program in a single process on the computer. Normally other programs create processes for you. For example, a shell (a command interpreter in UNIX) creates your first process “for free,” that is, it is automatically created for you and starts up running the program you requested. Many users use multiple processes in their work, even though they may not realize it. A shell pipeline is implemented by creating a process for each program in the pipeline. If you compile a program in the background and edit in the foreground, you are using multiple processes.

3.6.1 PROCESSES AND PROGRAMS

It is important to understand the distinction between a program and a process. A *program* is a static object that can exist in a file. It contains the instructions (the algorithm). A *process* is a dynamic object that is a program in execution. A program is a sequence of instructions. A process is a sequence of instruction executions. A program exists at a single place in space and continues to exist as time goes forward. A process exists in a limited span of time. Let’s look at an example of the difference. Consider the following C program:

program

```
main() {
    int i, prod = 1;
    for( i = 0; i < 100; ++i )
        prod = prod * i;
}
```

The program contains one multiplication ($\text{prod} = \text{prod} * i$), but the process will execute 100 multiplications, one each time through the loop.

	Program	Process
<i>Exists in</i>	Space	Time
<i>Is</i>	Static	Dynamic
<i>Consists of</i>	Instructions	Instruction executions

A program is a static object. A process is a dynamic activity.

3.6.2 PROCESS MANAGEMENT SYSTEM CALLS

We will present the process management system calls in several versions so that we can introduce one process-related concept at a time. Only the last of these versions (in Section 3.8) is compatible with the UNIX system calls.

Process Creation Without Arguments The operating system will allow a process to create new processes, to terminate itself, and to wait for a process it created to complete. There are just three system calls dealing with processes:

- `int SimpleCreateProcess( char * programName );`—A new process is created and the program `programName` is run in that process. The return value is the process identifier of the new process.
- `void SimpleExit( void );`—The process making the system call is terminated.
- `void SimpleWait( int pid );`—The calling process will wait until the process specified by the process identifier `pid` terminates.

The operating system creates the very first process when it initializes. That process then starts all other processes in the system using `SimpleCreateProcess` system calls. Each process makes a `SimpleExit` system call when it is finished. This terminates the process and releases any resources it was holding.

The operating system assigns to each process a unique number, which is used to identify the process. This is called the process identifier or PID. A process can wait for any other process to exit (as long as it knows the process identifier of the process it wants to wait for).

Processes are created, are waited for by their parent process, and exit when they are done.

Suppose you wanted to run two programs at the same time. One program is named `compiler`, the other is named `editor`. The following code creates a process for each program to run in and runs the processes at the same time. The procedure `CreateProcess1` must be called by a running process.

SIMPLE CREATE PROCESS

```
// This procedure creates two processes and waits for them
// to complete.
void CreateProcess1( void ) {
    // Create the two processes.
    int pid1 = SimpleCreateProcess( "compiler" );
    if( pid1 < 0 ) {
        cerr << "Could not create process \"compiler\" " << endl;
        return;
    }
}
```

```

int pid2 = SimpleCreateProcess( "editor" );
if( pid2 < 0 ) {
    cerr << "Could not create process \"editor\"" << endl;
    return;
}
// Wait until they are both completed.
SimpleWait( pid1 );
SimpleWait( pid2 );
// "compiler" and "editor" end by making SimpleExit system calls
SimpleExit();
}

```

Figure 3.8 shows the process hierarchy this creates.

Process Creation with Arguments Our `SimpleCreateProcess` is easy to use but does not allow us to pass any information to the process we create. Most programs require some arguments to tell them what to do. Let's extend the `Simple CreateProcess` system call to allow us to send arguments to processes we create. Different programs require different numbers of arguments, so we need a method that allows a variable number of arguments. We will use the method used in UNIX systems, that is, the arguments will be packaged as an array of strings.

In addition to passing information to a process, we want a process to return information to us when it exits. The information returned is status information about whether the child process was successful. Therefore, we allow an exiting process to send an integer return code to the process that waits for it.

The following table summarizes the process-related system calls and their arguments:

Call	Arguments	Returns	Notes
CreateProcess	Program file,args	Pid	Create the process
Exit	Return code	Does not return	Destroy the process
Wait	Pid	Return code	Wait for pid to Exit

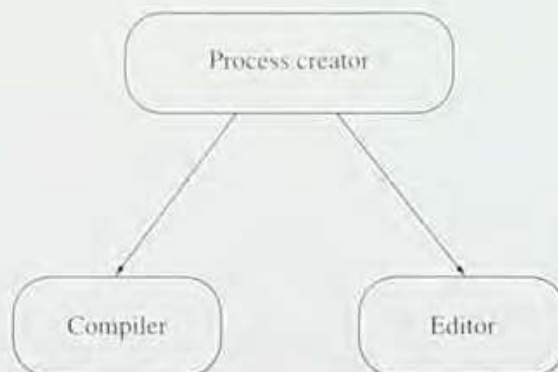


Figure 3.8 Creating two processes

The new system calls are specified as follows:

- `int CreateProcess( char * programName, int argc, char *argv [ ] );`—A new process is created and the program `programName` is run in that process. If the process creation is successful, the return value is the process identifier of the new process. The arguments are in the argument vector `argv` which is an array of strings, that is, an array of pointers to characters. `argc` gives the argument count, that is, the number of pointers to strings in `argv`. The return code is -1 if the process could not be created because process table space was not available. The return code is -2 if the `programName` could not be read for execution.
- `void Exit( int returnCode );`—This system call causes a process to exit. The return code is returned to the parent process when it executes the `Wait` system call.
- `int Wait( int pid );`—This system call will cause the calling process to wait until the process specified by `pid` exits. The return value is the return code provided by the process when it exited. Normally a return code of 0 means the process was successful.

Let's look at an example:

CREATE TWO PROCESSES

```
// This procedure creates two processes and waits for them
// to complete.
void CreateProcess2( void ) {
    // Create the argument list for the first process as an
    // array of strings created at compile time.
    static char * argv[3] = { "compiler", "fileToCompile", (char *) 0 };
    // Create the first process.
    int pid1 = CreateProcess( "compiler", 3, argv );
    if( pid1 < 0 ) {
        cerr << "Could not create process \"compiler\"" << endl;
        return;
    }
    // Create the argument list for the second process
    // at run time by assignments to the array components.
    char * argv[3];
    argv[0] = "editor";
    argv[1] = "fileToEdit";
    argv[2] = (char *) 0;
    // Create the second process.
    int pid2 = CreateProcess( "editor", 3, argv );
    if( pid2 < 0 ) {
        cerr << "Could not create process \"compiler\"" << endl;
        return;
    }
}
```

```

// Wait for both processes to complete.
(void) Wait( pld1 );
(void) Wait( pld2 );
Exit( 0 );

```

In this example we created one argument list at compile time in a static variable. The second argument list we created at run time by filling in the array values. It is more flexible to create argument lists at run time since then they can depend on information only available at run time. We ignore the return values of the processes. Figure 3.9 shows the process hierarchy this creates.

How Arguments Are Passed For `compiler` we constructed the argument vector when we declared `argb` using array initialization. We added a 0 pointer to mark the end of the array. This is not strictly necessary because we give an argument count, but it is a UNIX convention so we might as well follow it. For `editor` we constructed the argument list in the code.

Notice that the first argument is the name of the program itself. This is another UNIX convention that is widely followed, so we will follow it here. Generally, programs know what name they are called by so this information is superfluous, but some programs do make use of this information. For example, the `vi` full-screen text editor and the `ex` line editor are really the same program. The program looks at the name it was called with and goes to either full-screen mode or line mode, depending on whether that name is `vi` or `ex`. The same is true of `compress`, `uncompress`, and `zcat`, which are all the same program. The function performed is determined by the name by which the program is called.

The operating system will take these arguments and make them available to the new process by copying them from the memory of the creating process to the (newly allocated) memory of the created process (see Figure 3.10).

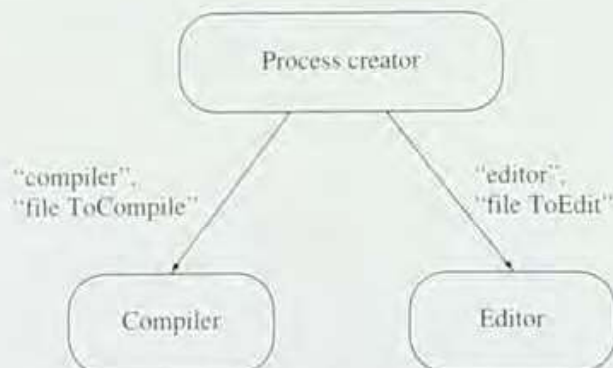


Figure 3.9 Creating two processes with arguments

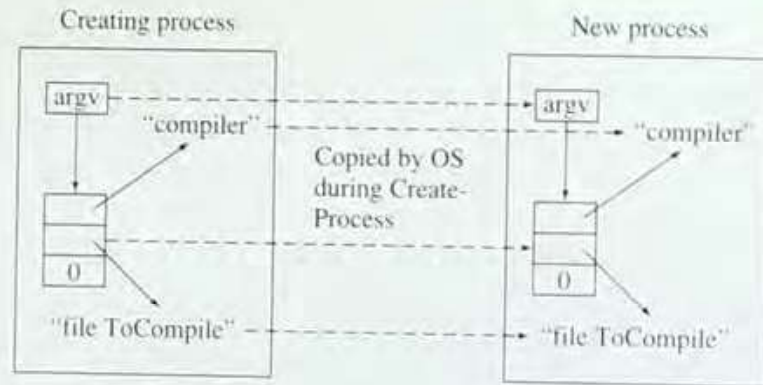


Figure 3.10 Passing arguments by copying between processes

The following code example shows how the process can get access to these arguments. It is a program that simply prints out its arguments.

PRINT ARGUMENTS

```
// This program writes out its arguments.
#include <iostream.h>
void main( int argc, char * argv[] ) {
    int i;
    for( i = 0; i < argc; ++i ) {
        cout << argv[i] << " ";
    }
    cout << "\n";
}
```

A **main** program has two arguments, the argument count and the argument vector. The operating system takes the argument vectors from the calling process and copies them to the new process. It builds these arguments on the process's stack before the process begins execution. The operating system takes care of the copying from the memory of the creating process to the memory of the created process since they do not share any memory.

3.6.3 PROCESS HIERARCHY

When one process creates another process, we say the creator is the *parent process* and the created process is the *child process*. This parent/child relationship can be shown as a tree structure of the processes in the system (see Figure 3.11), and this tree structure is called the *process hierarchy*.

Generally a parent is in control of a child process. The parent can stop the child temporarily, cause it to be terminated, send it messages, look inside its memory, etc.

parent process

child process

process hierarchy

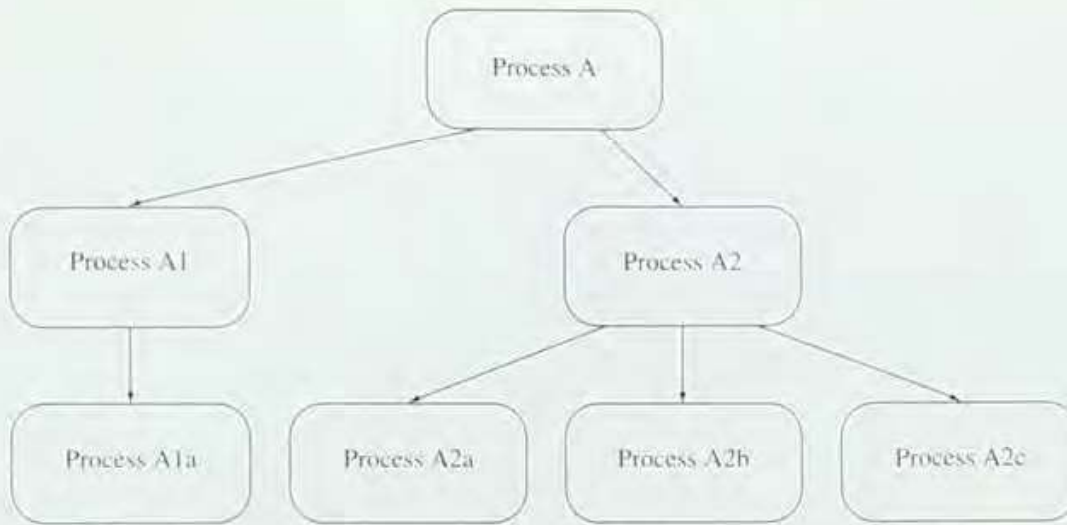


Figure 3.11 An example of process hierarchy

The operating system usually accumulates accounting information for the children in the parent. Sometimes resource control is handled through the hierarchy, that is, a parent allocates resources to the children who then allocate them to their children, etc. Usually only a parent can wait for a child process to complete. In some systems, if a parent process terminates, then the children are automatically terminated also. Other systems can “reparent” the orphan children by moving them somewhere else in the process hierarchy (UNIX does this).

Processes are organized in hierarchies. Parent processes have control over their child processes.

3.7 COMMUNICATION BETWEEN PROCESSES

Using the process system calls, we can send arguments to a new process and get a return code when that process completes. Sometimes, we will also want processes to communicate with each other while they are running. We call this *interprocess communication* or *IPC*.

interprocess communication
IPC

We will add a message system and provide system calls to send and receive messages between processes. We will allow a process to send a message even though the receiver is not yet ready to receive it. We will do this by *buffering* the messages, that is, we will save the message until the receiver asks for it. We buffer the message in a *message queue*, which holds messages that have been sent but not yet received. Message queues will be a type of object supported by the operating system. A process can create a message queue, and then any processes can send messages to it or receive messages from it.

buffering

message queue

Messages will be addressed to message queues. The alternative design would be to address a message directly to another process. In that case, we would have a message queue associated with each process. The process's message queue would hold messages that had been sent to it but that it had not received yet. We will see that addressing messages to message queues rather than specific processes has a number of advantages.

Each message will contain eight words. This is rather a small message size; therefore it will be difficult to send large quantities of data between processes. But for now we will use this small, fixed message size since we want to concentrate on the message passing and not the problems of message buffering.

Figure 3.12 shows several processes and message queues and how they are connected. Notice that a process can send to several message queues and receive from several message queues. Similarly, a message queue can be sent to by several processes and be received from by several processes.

Processes implement separate activities, but processes often need to communicate with each other.

3.7.1 COMMUNICATION-RELATED SYSTEM CALLS

We need system calls to create message queues and to send and receive messages. The following table summarizes the communication-related system calls:

Call	Parameters	Returns	Notes
CreateMessageQueue	None	qid	Create a message queue
SendMessage	qid, msg	return code	Send a message
ReceiveMessage	qid, msg	return code	Receive a message
DestroyMessageQueue	qid	return code	Destroy a message queue

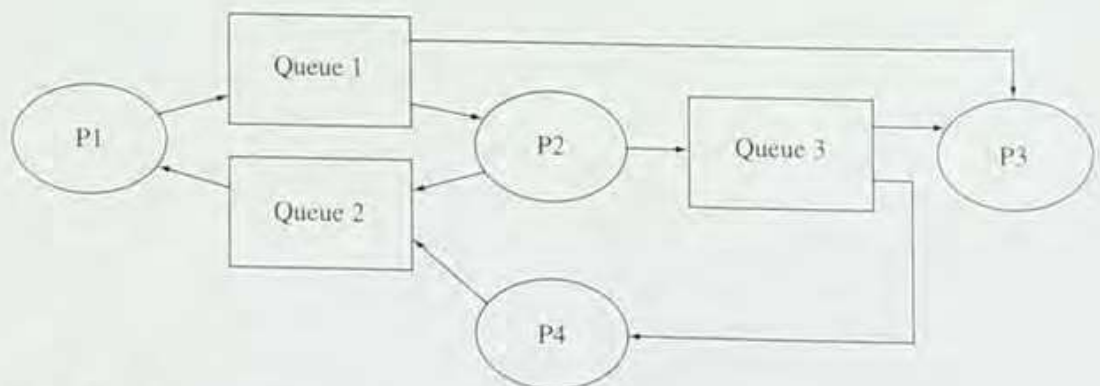


Figure 3.12 Complex message-passing paths between processes

The new system calls are described below.

- `int CreateMessageQueue( void );`—A new message queue is created and an identifier for the new message queue is returned to the caller. The message queue identifier is used in the send and receive message system calls. All message queue identifiers are global to the system; therefore all message queues can be accessed by all processes. The message queue will continue to exist until it is destroyed with a call to `DestroyMessageQueue`. A valid message queue identifier will always be a positive integer. The return value is -1 if there was a problem that prevented the queue from being created (such as no available memory).
- `int SendMessage( int msg_q_id, int * message_buffer );`—The message in `message_buffer` is sent to the message queue with identifier `msg_q_id`. The `message_buffer` is an array of 8 integers. The information in the message is not examined by the operating system but is simply transferred to the message queue and later to the receiver. The return value is 0 if the message is sent successfully. The return value is -1 if `msg_q_id` is not a valid message queue identifier, and -2 if `message_buffer` is an invalid address.
- `void ReceiveMessage( int msg_q_id, int * message_buffer );`—The oldest message in the message queue whose identifier is `msg_q_id` will be copied into `message_buffer`, an array of 8 integers. If there are no waiting messages, then the system call will not return until a message is sent to that message queue. The return value is 0 if a message is received successfully. The return value is -1 if `msg_q_id` is not a valid message queue identifier, and -2 if `message_buffer` is an invalid address.
- `int DestroyMessageQueue( int msg_q_id );`—The message queue with identifier `msg_q_id` will be destroyed. Any unreceived messages will be discarded. The identifier `msg_q_id` will no longer be valid for message sending and receiving. A return value is 0 if the message queue was destroyed. The return value is -1 if `msg_q_id` is not a valid message queue identifier.

3.7.2 EXAMPLE OF INTERPROCESS COMMUNICATION

Messages provide a basic communication capability between two processes. As an example, we will look at a program that will read a file and send the file to another process. The other process will count the number of characters in the file and return the count to the first process. This is not exactly a realistic example, but it does show the use of messages.

The following is the code for the file sender program:

MESSAGE: FILE SENDER

```
// A convenience procedure to make packaging and sending messages easier;
// Only for message of three or fewer words.
void SendMsgTo( int msg_q_id, int msg0=0, int msg1=0, int msg2=0 ) {
```

```

int msg[8];
msg[0] = msg0; msg[1] = msg1; msg[2] = msg2;
(void)SendMessage( msg_q_id, msg );
}
// This process sends the bytes in a file (one at a time)
// to another process.
enum { Reading=0, Writing=1, ReadAndWrite=2 };
enum { FileToOpen=1, SendQueue=2, ReceiveQueue=3 };
void main( int argc, char * argv[ ] ) {
    // The first argument is the file to send. Open for reading.
    int fromFD = open( argv[FileToOpen], Reading );
    if( fromFD < 0 ) {
        cerr << "Could not open file " << argv[FileToOpen] << endl;
        exit( 1 );
    }
    // The second argument is the message queue to which bytes will be sent.
    int to_q = atoi(argv[SendQueue]);
    // atoi = ASCII to integer (in C++ library)
    // Loop once for each byte in the file.
    while( 1 ) {
        char ch;
        // Read the next byte.
        int n = read( fromFD, &ch, 1 );
        // If there are no bytes left then exit the loop.
        if( n <= 0 )
            break;
        // Send each byte in a message to the message queue.
        SendMsgTo( to_q, ch );
    }
    close( fromFD );
    // Signal the end of file to the process receiving the bytes.
    SendMsgTo( to_q, 0 );
    int msg[8];
    int from_q = atoi(argv[ReceiveQueue]);
    ReceiveMessage( from_q, msg );
    // Receive the message containing the byte count and print it out.
    cout << msg[0] << " characters\n";
    exit( 0 );
}

```

We are sending the characters one per message. Obviously we could pack 32 characters per message (assuming a 32-bit word, eight bits per character, and eight-word messages), but that would complicate the code without revealing anything more about message passing in operating systems.

Here is the code for the receiver.

MESSAGE: FILE RECEIVER

```

// This process receives bytes one at a time and counts them up.
enum { SendQueue=1, ReceiveQueue=2 };

```

```

void main( int argc, char * argv[ ] ) {
    //start the count at zero.
    int count = 0;
    int msg[8];
    int from_q = atoi(argv[SendQueue]);
    while( 1 ) {
        ReceiveMessage( from_q, msg );
        if( msg[0] == 0 )
            break;
        //Any message with nonzero content is a character to count.
        ++count;
    }
    //Send the count back to the sender.
    int to_q = atoi(argv[ReceiveQueue]);
    (void) SendMsgTo( to_q, count );
    exit( 0 );
}

```

Finally, here is the code for the process that starts these two processes. Assume that the file sender has been compiled and linked, and the executable is in file **FileSender**. Similarly, the executable for the file receiver is in **FileReceiver**.

MESSAGE: START FILE SENDER AND FILE RECEIVER

```

//Convenience procedure for creating a process with up to 3 arguments.
int CreateProcessWithArgs(char * prog_name, char * arg1=0,
                          char * arg2=0, char * arg3=0) {
    char *args[5];
    //Create the argument array and create the process.
    args[0] = prog_name; args[1] = arg1;
    args[2] = arg2; args[3] = arg3; args[4] = 0;
    int argc = 4;
    if( arg3 == 0 ) --argc;
    if( arg2 == 0 ) --argc;
    if( arg1 == 0 ) --argc;
    return CreateProcess( prog_name, argc, args );
}

//Convert integers to strings
char * itoa( int n ) {
    char * result = new char[8];
    sprintf( result, "%d", n );
    return result;
}

//This process creates two message queues and then the two
//processes. No error code checking is done.
void main( int argc, char * argv[ ] ) {
    //Create the message queues the processes will use.
    int q1 = CreateMessageQueue();
    int q2 = CreateMessageQueue();
    //Create the two processes, sending each the identifier
    //for the message queues it will use.
}

```

```

int pid1 = CreateProcessWithArgs( "FileSend", "FileToSend",
                                   itoa(q1), itoa(q2) );
int pid2 = CreateProcessWithArgs( "FileReceive",
                                   itoa(q1), itoa(q2) );
//Wait for the two processes to complete.
int ret1 = wait( pid1 );
int ret2 = wait( pid2 );
//We do not use the return code ret1 and ret2 in this example.
//Destroy the message queues.
DestroyMessageQueue( q1 );
DestroyMessageQueue( q2 );
Exit( 0 );

```

The process must create the message queues first so that it can send the message queue identifiers to each process it creates. Having a common parent is an easy way to set up communication between two processes.

Figure 3.13 shows the creation and message-sending relationships between the three processes.

Message passing is a simple form of interprocess communication that is widely used in distributed operating systems. In centralized systems, more-optimized forms of interprocess communication are often used. In either case, a layer of software is placed above message passing to make interprocess communication simpler for users to use.

Message based communication is simple: sending and receiving messages.

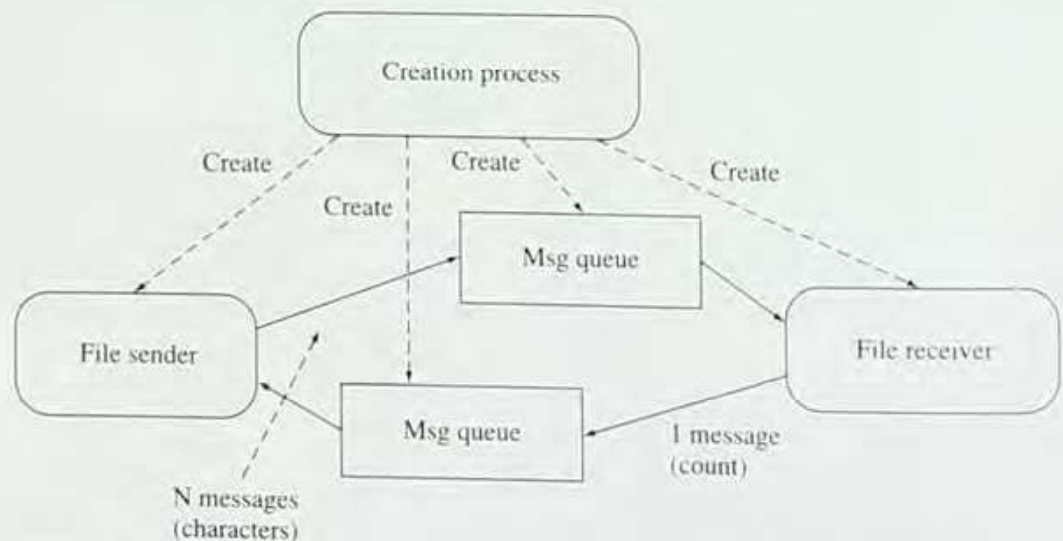


Figure 3.13 Sending a file with messages

3.8 UNIX-STYLE PROCESS CREATION

UNIX uses a very different method of process creation. In UNIX, two system calls are necessary to create a new process running a different program, one to do the actual creation of a new process and the other to run a specific program in the new process. In Section 3.9 we will discuss why UNIX does this. For now we will present the UNIX process system calls and show how they are used.

The following table summarizes the UNIX process-related system calls:

Call	Parameters	Returns	Notes
<code>fork</code>	None	pid or 0	Duplicate process
<code>execv</code>	<code>programFile, args</code>	None	Execute program in process
<code>exit</code>	return code	Does not return	Destroy process
<code>wait</code>	return code address	pid	Wait for process to exit

Here is their specification:

- `int fork( void );`—A new process is created that is an exact copy of the process making the system call. The return value to the parent is the process identifier of the new process. The return value to the child is 0. fork
- `int execv( char * programName, char *argv[ ] );`—The program `programName` is run in the calling process. The program in the calling process is overwritten and will no longer exist. The arguments are in the argument vector `argv`, which is an array of strings, that is, an array of pointers to characters. execv
- `void exit( int returnCode );`—This system call causes a process to exit. The return code is returned to whoever waits for the process.
- `int wait( int * returnCode );`—This system call will cause the calling process to wait until *any* process created by (that is, forked by) the calling process exits. The return value is the process identifier of the process that exited. The return code of that process is stored in `returnCode`.

The `fork` system call is somewhat unusual. A copy of the calling process is made and runs as a new process. The copy is of the memory of the calling process at the time of the `fork` system call, not the program the calling process was started from. The new process does not start at the beginning but at the exact point of the `fork` system call. Right after the `fork` there are two processes with identical memory images. Each one is about to return from a `fork` system call. The only difference between the two processes is the return value the operating system will give to the two returning `forks`. The `fork` of the parent returns the process identifier of the newly created process (the child process). The `fork` of the child returns a 0. Note that the `fork` system call returns twice, once to the parent process and once to the child process.

The UNIX `wait` system call is slightly different from the one we have seen before. The UNIX `wait` system call waits for *any* process to exit. You cannot control which one is waited for. The system call returns the first one it finds. As a consequence, the system call has to return the pid of the process that exited.

Sometimes it is more convenient to wait for any exited process and sometimes you want to wait for a specific process to exit. You can get the best of both worlds with a `wait` system call that waits for a specific process unless you pass in a special, unused pid (such as 0 or -1), which means to wait for any process to exit.

Figure 3.14 gives an example of a UNIX fork.

Let's redo the process creation example on page 46 with the UNIX process system calls.

UNIX: CREATE PROCESS

```
void CreateProcess3( void ) {
    int pid1, pid2;
    static char * argv[3] = { "compiler", "fileToCompile", 0 };
    //The fork will create a process that is a copy of this one.
    pid1 = fork();
    if ( pid1 == 0 ) {
        //Child process code begins here.
        //Child return pid1 == 0
        execv( "compiler", argv ); //Child executes the compiler
        //Child process code ends here.
        //execv does not return so control in the child
        //never returns here.
    }
    //Parent skips the "if" body and executes here because pid1 != 0
    argv[0] = "editor";
    argv[1] = "fileToEdit";
    argv[2] = 0;
    //Assign and test the return value from fork differently.
    if ( (pid2 = fork()) == 0 ) {
        execv( "editor", argv );
    }
    int reta, retb;
    int pida = wait( &reta );
    int pidb = wait( &retb );
}
```

In the `fork` method of process creation, the open files are inherited by the child process. This is an advantage that we will discuss in the next section.

In UNIX, pure process creation is logically separated from executing a new program in an existing process.

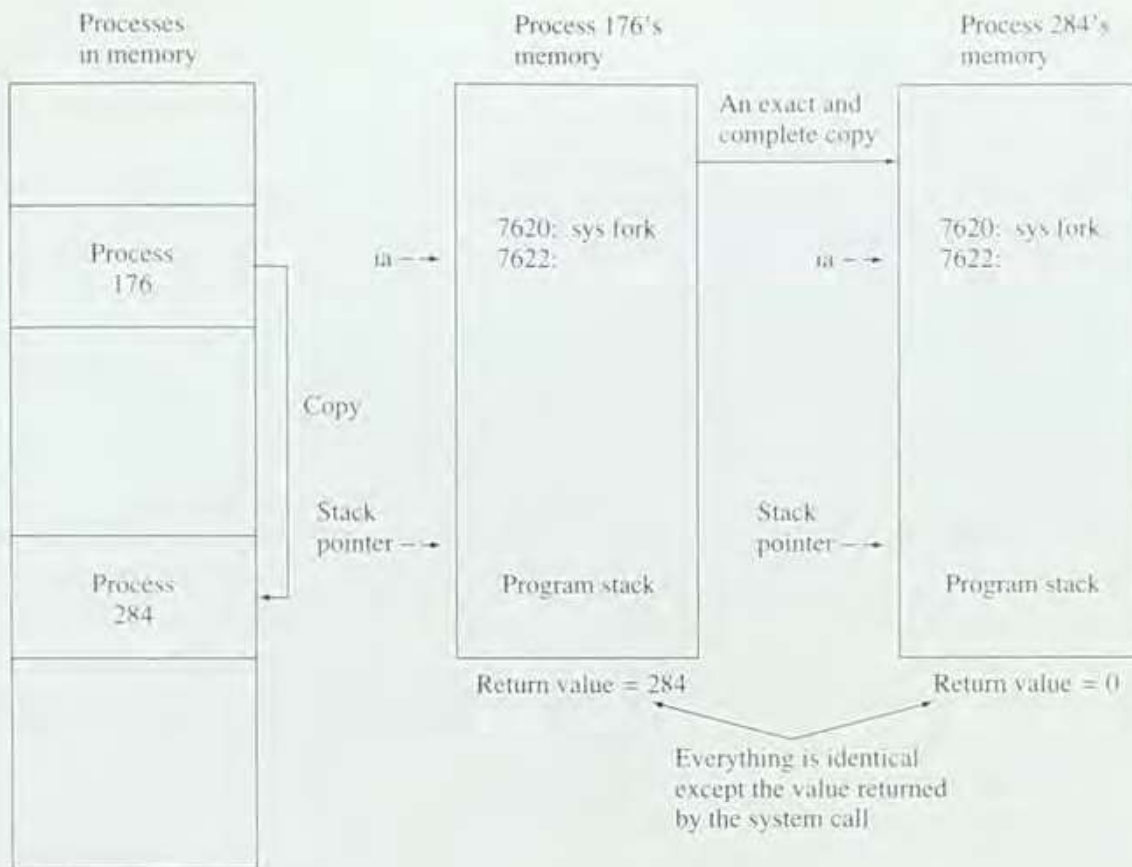


Figure 3.14 UNIX fork

3.9 STANDARD INPUT AND STANDARD OUTPUT

We have noted that programs can read from and write to open files, and that open files are named with integers that are returned by `open` and `creat`. We called these integers open file identifiers. In UNIX the open file identifiers 0 and 1 are treated specially. Open file identifier 0 is called *standard input* and open file identifier 1 is called *standard output*.

Many programs read data, transform it, and write it out again. By convention, in UNIX, these programs read their input from standard input (file identifier 0) and write their output to standard output (file identifier 1). When the shell starts a program, standard input and standard output are normally the user's terminal, but they can be changed by a method called *redirection*. If we start a shell process as follows:

```
someprog <infile >/dev/tty23
```

then the program will run with standard input reading from the open file `infile` and standard output writing to the device named `/dev/tty23`. A program whose main function is to read a sequence of bytes from its standard input, transform the stream in some way, and then write it to standard output is called a *filter*.

Figure 3.15 shows how input and output can be redirected.

standard input
standard output

redirection

filter



Figure 3.15 Standard input and output redirection

How does the shell do this? The UNIX `fork` system call makes a duplicate of the running process, and part of that duplication is that all the open files of the parent process (that called the `fork`) are copied (duplicated in UNIX jargon). When an `exec` is executed, the program running in the process is changed but all open files are preserved. So if the parent shell has its standard input reading from the user's terminal and its standard output writing to the user's terminal, then so will the child process. By default, the child inherits the standard input and output of the parent.

In UNIX, an open file is an object that exists independently of the process that created it. The process holds a pointer to the open file. When the process `forks`, the pointer is copied; thus, the child process can also use the open file.

If we want to redirect standard input from the user's terminal to a file, we can close standard input and open it again as file `infile`. UNIX has a system call that will open a file and give it a specific open file identifier. With this system call you can guarantee that the open file identifier will be 0. The open file already in 0 will be automatically closed.

We redirect standard input and output after the `fork` (so the `forking` process's standard input and output are not affected) and before the `exec` (so the new standard input and output will already be in effect when the new program begins execution). By separating process creation from execution of a new program in a process, we can create a process, set up its open files (standard input and output in particular), and then execute the program. This is the main advantage of the UNIX method of creating processes.

To achieve this effect with the `CreateProcess` form of process creation, we would need to modify the `CreateProcess` system call. For example, we might have a form of the call:

- `int CreateProcessWithStdio( char * programName, int stdin, int stdout );`—A new process is created and the program `programName` is run in that process. Open file `stdin` will be duplicated and made the standard input of the new process. Similarly, the open file `stdout` will be duplicated and made the standard output of the new process. The return value is the process identifier (pid) of the new process.

This still is not as flexible as the UNIX `fork/exec` method because it does not allow us to share other open files, only standard input and standard output.

Another approach would be to redefine `CreateProcess` as:

- `int CreateProcessAndDuplicateOpenFiles( char * programName );`—A new process is created and the program `programName` is run in that process. All of the open files in the parent process will be duplicated and be open files in the child process. The return value is the process identifier of the new process.

So to set the standard input and output for the child process, the parent process will change its standard input and output to the desired open files, create the child process, and then change the standard input and output back to their original values. This is less convenient than the UNIX `fork/exec` method since it requires the parent to change its own standard input and output before the create and change it back afterwards. It also requires the parent to share all of its open files instead of allowing it to selectively close files after the `fork` and before the `exec`.

3.10 COMMUNICATING WITH PIPES

We have seen how devices can be named using the file system hierarchical naming system and how these devices are used in the same way as files. We can use the same idea to allow interprocess communication through a technique called *named pipes* (also called a *fifo*). A *pipe* is an object that can act like a file, that is, it can be written to and read from. We can write bytes to the pipe. When we read from a pipe, we get the same bytes that were written and in the same order as they were written. A pipe is useful when we want to write from one process and read from another. This allows us to use pipes as a method of communication between processes, an alternative to messages.

fifo
pipe

Note how different this method is from messages. With messages, there is a fixed message size which both sender and receiver must adhere to. With a pipe, a process writes to the other process as if it were writing to a file. There are no fixed-size blocks. For example, the process writing the pipe can write in units of 100 bytes and the process reading the pipe can read in units of 20 bytes. (See Figure 3.16.)



Figure 3.16 Messages and pipes

In addition, we use the familiar file access interface instead of having to learn a new interface involving sending and receiving rather than writing and reading.

It is possible for a pipe to have several writers and several readers, although in that case it is hard to keep sorted out what belongs to whom.

The example we used in Section 3.7.2 to show the use of the message system calls has three processes. A sender process read a file and sent it, one character at a time, to a receiver process. The receiver process counted the characters and sent back a count when the end of file was reached. There was also a main process that created the sender and the receiver. Let's redo our message example with named pipes. First, the file sender will look like this:

PIPE: FILE SENDER

```
enum { Reading=0, Writing=1, ReadAndWrite=2};
void main( int argc, char * argv[ ] ){
    //The first argument is the file to open.
    int fromFD = open( argv[1], Reading );
    //The second argument is the pipe to write it on.
    int to_pipe = open( argv[2], Writing );
    while( 1 ) {
        char ch;
        int n = read( fromFD, &ch, 1 );
        if( n == 0 )
            break;
        write( to_pipe, &ch, 1 );
    }
    close( fromFD );
    //Closing the pipe will cause the receiver to get an
    //end of file after it reads all the characters currently
    //in the pipe.
    close( to_pipe );
    //The third argument is the pipe from which to read the count.
    int from_pipe = open( argv[3], Reading );
    int n;
    //We are assuming the int n is four bytes long.
    //so we read four bytes.
    read( from_pipe, &n, 4 );
    close( from_pipe );
    cout << n << " characters\n";
    exit( 0 );
}
```

Here is the code for the receiver.

PIPE: FILE RECEIVER

```

enum { Reading=0, Writing=1, ReadAndWrite=2 };
void main( int argc, char * argv[ ] ) {
    int count = 0;
    //The first argument is the pipe to read from.
    int from_pipe = open( argv[1], Reading );
    while( 1 ) {
        char ch;
        int n = read( from_pipe, &ch, 1 );
        if( n == 0 )
            break;
        ++count;
    }
    close( from_pipe );
    //send the count back to the sender.
    //The first argument is the pipe to write to.
    int to_pipe = open( argv[2], Writing );
    write( to_pipe, &count, 4 );
    close( to_pipe );
    exit( 0 );
}

```

In the message version, the file receiver receives a message containing a zero, which indicates the end of file. In the pipe version, it gets an actual end of file (a read count of 0) from the read on the pipe.

Finally here is the code for the process that starts these two processes:

PIPE: START FILE SENDER AND FILE RECEIVER

```

void main( int argc, char * argv[ ] ) {
    int pid1 = CreateProcessWithArgs( "FileSend", "FileToSend",
                                      "PipeToReceiver", "PipeToSender" );
    int pid2 = CreateProcessWithArgs( "FileReceive",
                                      "PipeToReceiver", "PipeToSender" );
    int ret1 = wait( pid1 );
    int ret2 = wait( pid2 );
    exit( 0 );
}

```

It is possible to connect several processes together with pipes. Figure 3.17 shows a pipeline created by connecting the standard output of one process to the standard input of another process. The two processes communicate through a pipe.

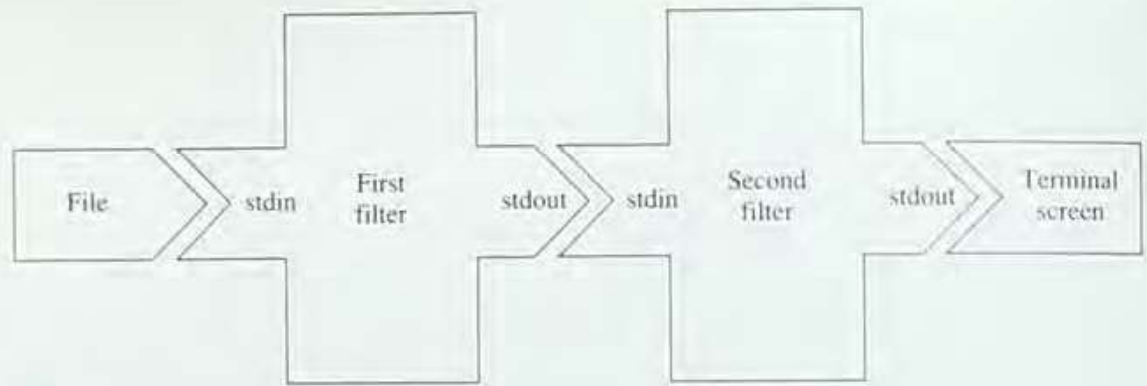


Figure 3.17 A pipeline of two processes with redirection at both ends.

Pipes connect program tools together to make larger tools.

3.10.1 NAMING OF PIPES AND MESSAGE QUEUES

One difference between named pipes and message queues (as we specified them) is how they are named. Named pipes are named by the file system naming system and they are implemented by an entry in the file system indicating that they are a named pipe. There is a system call to create a named pipe in the file system. Thus they have some kind of existence even when no process is using them. Any two processes can communicate if they both know the name of the pipe. They just both open the pipe and use it as an open file.

Message queues have no name and no existence before they are created by a process. When they are created they are given an internal (operating system defined) name for use by processes wanting to pass messages. This name can be passed to other processes, but passing the name requires some form of communication. That limits the processes that can receive the name to descendants of the process that created the message queue. So two processes need a common ancestor in order to communicate.⁹ This is much less convenient than the named pipe approach.

There is nothing inherent in message queues that requires them to be named as we have specified in this chapter. We could also allow message queues to have a place in the file system and use the file naming system. Instead of creating a message queue, we could “attach” to a message queue and get an internal name to use in message sends and receives. When we were done, we could unattach the queue.

⁹It is possible to emulate the named pipe communication in the following way. A process could create a message queue and write its name to a file. Then another process could read the name from the file and the two processes could communicate. This would work, with message queue names as we defined them, but a better solution would be to get the operating system to help you find the name.

DESIGN TECHNIQUE : CONNECTION IN PROTOCOLS

Sometimes an operating system service requires a sequence of interactions—for example, reading a sequence of blocks from a file or exchanging a sequence of message with another process. Usually there is some initial cost in setting up the service. For example, to read a file it is necessary to look up its name in a sequence of directories. This may require a number of disk reads. In this case we use a *connection* protocol where there are separate operations to connect to (open) the file and disconnect from (close) the file.

Connection protocols are more efficient because the setup is only performed once for each connection. But there are other tradeoffs. Connections sometimes tie up

resources unnecessarily and are less efficient if there is only one transaction. Connections require reliable communication partners and assume that future use can be accurately predicted.

The distinction between connection and connectionless protocols is especially important in networks of computers. Some network connection protocols use connections and some do not. Which is best depends on many factors about the network and the communication. Loose connections like the Internet are better served with connectionless protocols. For example, World Wide Web protocols are connectionless.

See Section 4.6 for more discussion of connection and connectionless protocols.

which would be like destroying the message queue. The first *attach* operation would create the message queue as an operating system data structure, and subsequent attaches would return an internal identifier to that data structure.

Similarly, we could have named pipes with an internal name only. A process would create a pipe by calling `CreatePipe` and get an internal identifier back, which could then be opened to get an open file. The original UNIX pipes were like this, except they combined the creating of the pipe with the opening of the pipe. Thus pipe creation returned two open file identifiers, one to write the pipe and one to read it. These open files were passed to forked processes in the same way as any other open files. This method of naming pipes also required processes to have a common ancestor in order to communicate with pipes. It was this limitation that led to the concept of named pipes in UNIX.

Object names can be global or local to a process and its descendants.

3.11 SUMMARY OF SYSTEM CALL INTERFACES

We have looked at a number of system calls and some alternative ways to specify some of the system calls. The system calls fall into a few categories:

- *File and I/O*—open, creat, read, write, lseek, close, unlink, stat. These gave us the ability to create and destroy files and to read and write them.

- *Processes*—CreateProcess, Wait, Exit. These gave us the ability to create and terminate processes and to wait for a child process.
- *Interprocess Communication*—CreateMessageQueue, SendMessage, ReceiveMessage, DestroyMessageQueue. These gave us the ability to create and destroy message queues and to send and receive messages using message queues.

This set represents the core functionality needed in an operating system. A real operating system will have many more system calls, but most of them will be used in specific situations to satisfy certain requirements. As an operating system evolves, its system call interface may expand to meet new needs. The biggest omission in our system call interface is the lack of facilities to deal with directories—create and destroy directories, and to create and destroy directory entries. The other missing system calls are minor but necessary functions like: getting your own process ID, getting your parent's process ID, handling networks, using shared memory, etc.

Our system call interface is based on the UNIX system call interface, and many of the calls are the same. Most UNIX systems have over a hundred system calls. We could list them here, but many of them require knowledge about operating systems that we have not covered yet.

3.12 OPERATING SYSTEM EXAMPLES

Throughout the book we will provide examples from existing operating systems. First we will discuss a topic in general terms without mentioning specific operating systems. Then we will have a section where we describe how a service is defined and implemented in several existing operating systems. In this section we will introduce several of the operating systems that will serve as running examples throughout the book.

3.12.1 UNIX

The UNIX operating system will be one of our most frequently cited examples. UNIX has a special place in the history of operating systems. The first implementations of UNIX was developed at Bell Telephone Laboratories in the early 1970s by Dennis Ritchie and Ken Thompson. It was first used inside Bell Labs, but soon it was licensed (at a very low cost) to universities for educational and research purposes. UNIX was the first widely used operating system to be written almost entirely in a high-level language (the C language). This made it feasible to port UNIX to various computer architectures. During the 1970s, UNIX proliferated at universities and was ported to several different architectures. It became the most common operating system used in university computer science departments. UNIX was widely used in computer science by students for classwork and projects and by faculty for operating systems research. This was fostered by the fact that the source code for UNIX was available for modification.

This led to many changes, improvements, and variations in UNIX. The most well-known variation was developed at the University of California at Berkeley, which distributed the Berkeley Software Distribution (BSD) version of UNIX. The BSD versions of UNIX provided many technical improvements in UNIX, including virtual memory, a faster file system, TCP/IP support, sockets, and more.

AT&T (the owner of Bell Labs) also developed a version of UNIX called System V (the “V” is the roman numeral 5) UNIX. The latest version is release 4 and is usually known as SVR4.

Many commercial workstation vendors have developed their own versions of UNIX (for example, SUN Solaris, HP UNIX, Digital UNIX, SGI IRIX, and IBM AIX). These are either derived from the BSD version or the System V version. Recently there has been some standardization on SVR4, but there are still many minor variations between UNIX systems from different vendors.

UNIX has been the basis for operating systems research for a number of years, and most of the technical developments in operating systems originated in UNIX systems. In addition, the UNIX system call interface and implementation has been widely influential, and most other operating systems developed in the last 20 years have borrowed to some degree from the UNIX interface and implementation. As a result, this book will draw heavily on UNIX and its variants.

The system call interface discussed in this chapter is nearly identical to a subset of the UNIX system call interface. Actual UNIX systems, of course, have many more system calls.

3.12.2 MACH

In the mid 1980s, researchers at Carnegie-Mellon University began working on a new operating system based on the idea of a *microkernel*, that is, a small operating system (the *kernel*) that provides only the basic services. The rest of the usual operating system functionality was provided by processes running outside the kernel. Mach provides processes that emulate the UNIX environment, so UNIX programs can be run on it. A consortium of computer vendors has developed an operating system called OSF-1, which uses Mach as its basis.

microkernel

The ideas developed in Mach have also been highly influential, and almost all operating systems developed since Mach have borrowed some ideas from Mach. As we go through the book, we will look at the basic ideas in Mach more closely.

3.12.3 MS/DOS

When IBM came out with their personal computer in the early 1980s, they also arranged for Microsoft to provide the MS-DOS operating system. MS-DOS was developed for very small personal computers and did not have many of the features expected of operating systems for larger computers. But the PC and its clones became much more successful than most people had imagined, and MS-DOS has grown to be the most widely used operating system in the world. As the power and capabilities of

the PC increased, features were added to MS / DOS to deal with the new hardware. Although MS / DOS is quite primitive by modern operating system standards, it is an open system (due to the fact that there is no protection at all) and so it could be adapted and modified by add-on programs. MS / DOS has gone through six major versions over the years since the first PC was introduced.

The MS / DOS system call interface is modeled after UNIX and has a similar set of system calls. One major difference is that it does not have the UNIX `fork` or `exec` system calls, but has a system call more like the `CreateProcess` described earlier in this chapter.

One add-on to MS / DOS is the Microsoft Windows windowing environment. Windows adds significant functionality to MS / DOS and could be considered an operating system on its own.

3.12.4 WINDOWS NT

The deficiencies in MS / DOS became apparent as it was moved to more and more powerful hardware. MS / DOS started out as an operating system for a small personal computer with a small memory, no hard disk, no protection, no memory management, and no network. A small and simple operating system was appropriate for this hardware. But as personal computers became more powerful, they got to the point where they needed and could support a full-featured operating system. Many facilities were added on to MS / DOS, including networking and a windowing system, but it had basic problems that could not be easily fixed.

As a consequence, Microsoft decided it needed a modern operating system for personal computers and decided to develop Windows NT (for New Technology). Design for Windows NT was started in late 1988 and the first version was released in 1993. The designers of Windows NT took advantage of many of the new ideas that had been developed in operating systems up to that point, including ideas from UNIX and Mach. We will describe many of the design features of Windows NT in the course of this book.

3.12.5 OS/2

OS/2 is IBM's operating system for personal computers. It was developed to remedy the known deficiencies in MS / DOS. OS/2 version 1.0 was released in 1987 and was a simple extension of MS / DOS to more capable hardware. OS/2 version 2.0 was released about 1990 and was a substantial revision and extension of OS/2. With version 2.0, OS/2 became a modern full-featured 32-bit operating system.

3.12.6 MACINTOSH OS

The other significant operating system for personal computers is the Macintosh OS (MacOS) for Apple Macintosh computers. The Macintosh provided a significant challenge to the domination of the IBM PC and PC clones in the personal computer

market. The Macintosh provided a full-fledged graphical interface and was significantly easier to use than PCs. Although the Macintosh never captured much more than 10 percent of the personal computer market, it influenced the PC market and caused the growth of MS Windows as an add-on to MS/DOS.

MacOS was a more complete operating system than MS/DOS. It did not have all the features of operating systems like UNIX but it provided substantial functionality to support the graphical user interface. It also had a sophisticated file system and support for networks and virtual memory fairly early on.

3.13 THE USER INTERFACE TO AN OPERATING SYSTEM

3.13.1 WHY YOU NEED A SHELL

All operating system services are available through system calls. But in order to issue a system call you have to be running a process. How does the first process get started? As we said before, the operating system constructs the first process. This initial process then starts all the other processes.

But suppose you want to communicate *interactively* with the operating system. If we wrote an interactive program that communicated with the user through a terminal and with the operating system through the system calls, we could make all the operating system services available interactively to a user sitting at a terminal. This is exactly what a *shell* is: a user interface to the operating system built on the programming (system call) interface. Figure 3.18 shows two viewpoints of a shell.

shell

As another example of using system calls, we will write a simple shell.

3.13.2 THE SPECIFICATION OF THE SHELL

The shell executes commands of the form:

```
commandName [argument1...]
```

The shell executes (via `fork` and `exec`) the program `CommandName` and passes the argument on the command line to it (via the `argv` array of strings). Then the shell waits for the command to exit.

3.13.3 IMPLEMENTING THE SHELL

In this section we will describe a C++ program that implements the shell interface using the system call interface. Below is the code to implement a simple shell using the system call interface. First, here are the declarations, definitions, and global data.

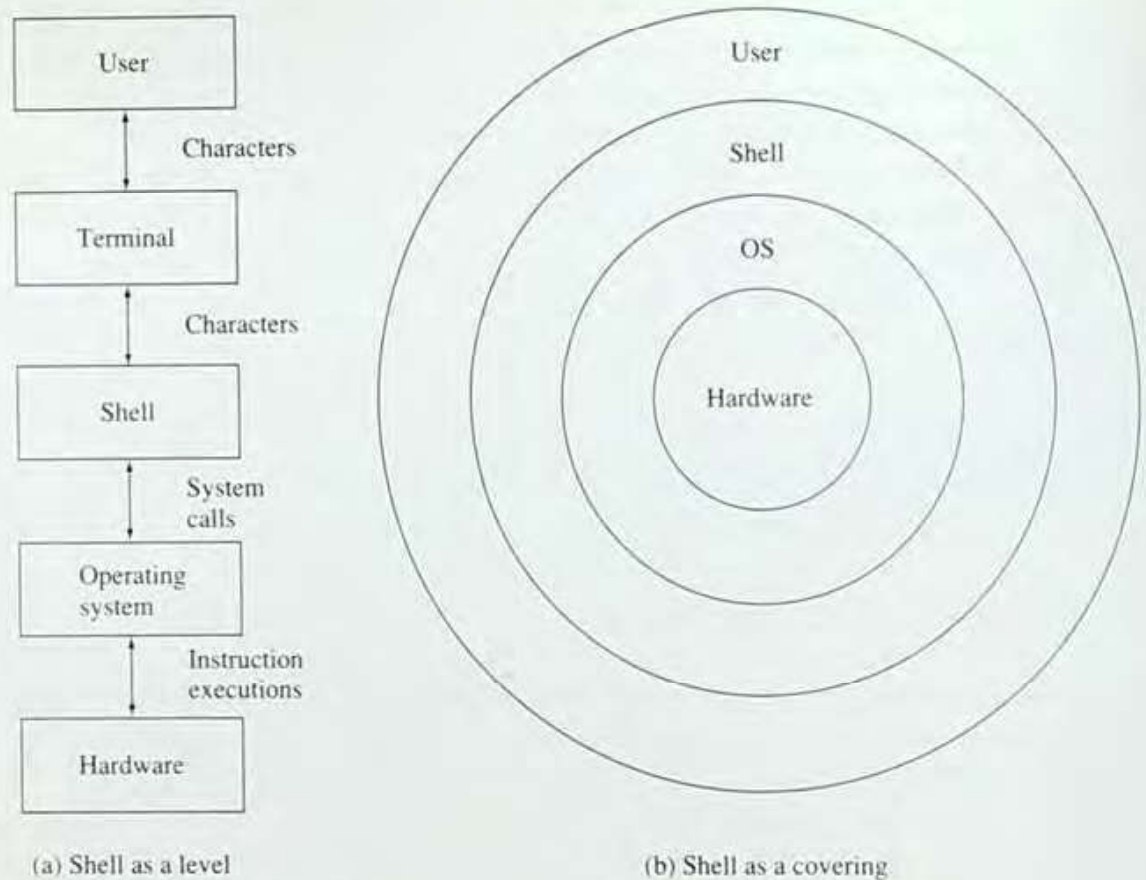


Figure 3.18 Two views of a shell

SHELL GLOBALS

```
#include <iostream.h>

//some constants
constant int ARGSize = 50; //maximum size of any one argument
constant int NARGS = 20;  //maximum number of arguments

//token types returned by getWord()
constant int STRING = 1;
constant int INREDIR = 2;
constant int OUTREDIR = 3;
constant int NEWLINE = 4;

//define the argv structure
char *argv[NARGS]; //space for argv vector
char args[NARGS][ARGSize]; //space for arguments.
```

We set limits on the number and length of the argument to shell commands. A real shell would handle this more dynamically and allocate space as needed. We are going to parse the command line with a parsing procedure. The constant definitions give the types of the tokens found by the parser.

We will assume we have a scanner that reads the input, handles quoting, detects the special symbols, and deals with tokens that are too long. We will assume the scanner has a procedure `getWord` that returns the token type and copies the token itself into the string it is passed.

Here is the code for the shell.

SHELL MAIN PROGRAM

```
//This program is a very simple shell.
void main( int argcount, char *arguments[ ] ) {
    int wasInRedir, wasOutRedir;
    char inRedir[ARGSIZE], outRedir[ARGSIZE];

    //each iteration of this loop will parse one command
    while( 1 ) {
        //display the prompt
        cout << "@";
        //So far we have not seen any redirections
        wasInRedir = 0;
        wasOutRedir = 0;
        //Set up some other variables.
        int argc = 0;
        int done = 0;
        char word[ARGSIZE];
        //Read one line from the user.
        while( !done ) {
            //getWord is the scanner, it gets one word from the line.
            int argType = getWord(word);
            //getWord also returns the type of the word it read.
            switch( argType ) {
                case INREDIR:
                    //We saw an input redirection character (a <).
                    //Record seeing it and read the input file name
                    //(which is the next word on the line).
                    wasInRedir = 1;
                    (void)getWord(inRedir);
                    break;
                case OUTREDIR:
                    //We saw an output redirection character (a >).
                    //Record seeing it and read the output file name
                    //(which is the next word on the line).
                    wasOutRedir = 1;
                    (void)getWord(outRedir);
                    break;
                case STRING:
                    //everything else is a string to put into the
                    //list of arguments.
                    strcpy(args[argc], word);
                    argv[argc] = &args[argc][0];
                    ++argc;
                    break;
            }
        }
    }
}
```

```

        case NEWLINE:
            //When we see the newline the command has ended.
            //(We do not allow commands to span lines.)
            done = 1;
            break;
    }

    //terminate the argument list
    argv[argc] = NULL;
    //The logout command is not executed, instead it exits the shell.
    if( strcmp(args[0], "logout") == 0 )
        break;
    //Fork a child process and set up stdin and stdout.
    if( fork() == 0 ) {
        //handle input redirection (if any)
        if( wasInRedir ) {
            close(0); //close standard input
            open(inRedir, 0); //reopen it as the redirect file
        }

        //handle output redirection (if any)
        if( wasOutRedir ) {
            close(1); //close standard output
            enum { UserWrite=0755 };
            creat(outRedir, UserWrite);
            //reopen it as the redirect file
        }

        //Look for the command name in two places:
        //(1) the current directory
        //(2) the /bin directory.
        char cmd[60];
        strcpy(cmd, "./"); strcat(cmd, args[0]);
        execv(cmd, &argv[0]);
        strcpy(cmd, "/bin/"); strcat(cmd, args[0]);
        execv(cmd, &argv[0]);
        //The execv only returns if the exec failed.
        cout << "Child: could not exec \"" << args[0] << "\"\n";
        exit(1);
    }

    //wait for the child to finish
    int status;
    (void) wait(&status);
}

finish;
cout << "Shell exiting. \n";
}

```

Basically the shell collects words, puts them in an argument array, and forks off a process to execute the command. The shell must handle input and output redirection.

Real shells (for example, `sh`, `osh`, `ksh`, `bash`, etc.) are much more complicated than this one (they comprise tens of thousands of lines of code), but conceptually they are the same as this shell. Their extra complexity is in providing a sophisticated user interface: history, quoting, programming facilities, etc.

DESIGN TECHNIQUE: INTERACTIVE AND PROGRAMMING INTERFACES

System calls allow a running process to use operating system services. A shell allows a user to use operating system services interactively. Many programs have only an interactive interface. For example, a drawing program allows you to create drawings by direct interaction, but you cannot create drawings from another program. This makes it tedious to create a complex but regular drawing (for example, a drawing consisting of 100 squares in a ten-by-ten array where each square is 10 percent smaller than the one above it and the one to the left of it).

Most programs could benefit from having both kinds of interfaces. Some operating systems now have *scripting architectures* that define a standard protocol for programming interfaces to applications, encourage all applications to provide a programming interface (often called a scripting interface), and provide a system-wide scripting language (and language support) that allows programmers to write programs that interact with other programs.

For more discussion of interactive and programming interfaces, see Section 4.7.

A shell is an interactive interface to an operating system. System calls are the programming interface to an operating system.

3.14 SUMMARY

The interface an operating system presents to user programs is called the system call interface. A system call is a hardware instruction that transfers control to the operating system, which then performs some service for the user. The system call interface has three main parts: the file and I/O system calls, the process management system calls, and the interprocess communication (IPC) system calls.

An operating system provides a naming system for files, devices, message queues, and other objects. This is usually a hierarchical naming system based on a tree of directories. Each directory maps names into objects. Some of the objects named in a directory are other directories and this forms the tree structure.

Files are passive containers of data on a disk. A file must be opened (a system call) before it can be used. Opening a file returns an open file. The open file can be read and written (via system calls), either sequentially or randomly. The file location keeps track of the offset at which the next read or write will take place. An open file is closed when access to it is no longer needed. Many operating systems name hardware devices in the same way as files and allow access to devices using the same interface as to files: open, read, write, close.

Operating systems keep meta-information about each file: who owns it, when it was created, who can access it, how long it is, etc. There is a system call to read the meta-information and several system calls to write it.

A process is the execution of a program. All user computation in an operating system is done through processes. The three basic system calls for processes are:

create a new process, exit the current process, and wait for another process to exit. Process creation allows the transfer of arguments to the new process. A created process is the child process of the creator, which is the parent process. This creates a process hierarchy where a parent has some control over its children.

Operating systems provide methods for processes in execution to communicate with one another. This is called interprocess communication or IPC. One basic method of interprocess communication is the transfer of messages between processes. Messages are like letters, and they are often addressed to message queues, which are like mail boxes. A process can send messages to and receive messages from message queues.

Another method of IPC (different from message passing) is to use a pipe. A pipe is a mechanism that allows IPC to look like reading and writing an open file. Instead of writing to a file, you write to another process. Each process has a standard input and a standard output.

UNIX uses a different method of process creation which separates the creation of a new process (a fork) from the execution of a new program in an existing process (an exec).

The system call interface allows a running process to request operating system services by making system calls. A shell is an interactive user interface to an operating system that allows a user to enter commands as character strings or through a graphical user interface. The shell converts them to system calls to the operating system. System call results and other information from the operating system are presented to the user through an interactive interface.

3.14.1 TERMINOLOGY

After reading this chapter, you should be familiar with the following terms:

- buffering
- child process
- component name
- device independence
- directory
- `execv`
- fifo
- file
- file location
- filter
- flat name space
- fork
- full path name
- hierarchical file naming system

- interprocess communication (IPC)
- message queue
- meta-information
- microkernel
- name lookup
- open file
- open file identifier
- parent process
- pipe
- process
- process hierarchy
- program
- redirection
- shell
- standard input
- standard output
- system call interface
- two-level name space

3.14.2 REVIEW QUESTIONS

The following questions are answered in the text of this chapter:

1. What is a system call interrupt handler?
2. Go through the flow of control during a system call.
3. What is a directory?
4. Why does a read advance the file location?
5. Why do files need to be opened?
6. What are the file modes that you can open a file in?
7. What does it mean when a `read` system call returns a 0?
8. How do you find out the current value of the file location?
9. What is the purpose of the `lseek` system call? Why is it necessary?
10. What does it mean to name a device with the file naming system?
11. What does it mean for a program to be device independent?
12. Explain the difference between a program and a process.
13. Explain the interactions between the `CreateProcess`, `Wait`, and `Exit` system calls.
14. Explain how arguments are passed to child processes.

15. In C++ argument passing there is a variable `argc`. What is `argc` used for and why can it not be 0?
16. What process is at the root of the process hierarchy tree?
17. What is message buffering? Why is it useful?
18. Explain how file identifiers are local to a process.
19. Explain how queue identifiers are global to all processes.
20. Explain the difference between the `CreateProcess` and the `fork` system calls.
21. Explain the difference between the `CreateProcess` and the `execv` system calls.
22. Explain what standard input and standard output are.
23. Explain how pipes allow you to use variable-sized messages.

3.14.3 FURTHER READING

There are many books about UNIX. An excellent book on UNIX internals is Vahalia (1996). Other books on UNIX internals are Bach (1986) and Leffler et al. (1989). Goodheart and Cox (1992) describe System V Release 4. Deitel and Kogan (1994) describe OS/2 2.0. The Open Software Foundation (1993) has a book about OSF/1 which is based on Mach. Baron et al. (1990) is one of many places to find information about Mach. Custer (1993) is the main reference on Windows NT.

3.15 PROBLEMS

1. Suppose a computer did not have a special system call instruction but it did have a typical interrupt system. How could you simulate a system call instruction on such a computer?
2. Should terminating a process also terminate all of its children? Give an example where this is a good idea and another example where it is a bad idea.
3. Should the open files of a process be closed automatically when a process exits?
4. Why do most programs require arguments? Give an example of a program that never requires any arguments.
5. Give some of the differences between opening a file and opening a device.
6. Why does the `creat` system call also open the file? What if it didn't? Can we do it ourselves and get the same effect?
7. The "Reverse" program on page 37 does not work for files of length 0. Why not?
8. Suppose we wanted to implement a linked data structure in a disk file rather than in memory. Explain how you would use the `read`, `write` and `lseek` system calls to do this. Compare this with implementing linked structures in an array using array indices as pointers.
9. Design a form of the `open` system call that subsumes the functions of the `creat` system call.

10. Write a program using the file system calls (`open`, `creat`, `read`, `write`, `lseek`, `close`, `unlink`) that determines the length of a file without using a loop in the code.
11. Modify the file copy program to read and write in blocks of 1024 bytes instead of just one byte at a time. Do *not* assume that the file will have a length that is a multiple of 1024.
12. Modify the file copy program to check for all possible errors that can occur and return an error code of 0 if there is no error and a different negative value for each type of error that you can detect.
13. Modify the reverse program to reverse the file in place, that is, reverse the file without using any other files and using a fixed amount of memory (that does not depend on the size of the file being reversed). *Hint*: You need to open the file for reading and writing. *Hint*: Exchange the bytes, in pairs, outside-in to the middle.
14. The form of `read` we defined first in this chapter specified the file descriptor, the buffer address, and the number of bytes to read. The file position was kept up to date by the system and not directly specified in the `read` system call. The `lseek` system call was used to change the file position.

Suppose we want to remove the buffer address from the `read` system call. We add a new system call `SetIOBufferAddress` with one argument that sets the buffer address for subsequent `read` (and `write`) system calls. This must be called before any `reads` or `writes` (else they will return an error code). `SetIOBufferAddress` can be called more than once and the last one is the one that is used. Rewrite the file copy program to use this form of `read`, `write`, and `SetIOBufferAddress`.

Examine the tradeoffs between this kind of I/O interface and the original interface we defined. In which cases is this better and in which cases it is worse?

15. Consider the previous problem. Suppose we copy a file with the file copy program on page 36 and that a system call takes 300 microseconds plus 20 microseconds per argument. Now consider the same program but changed as described by using the `SetIOBufferAddress` system call (once outside the loop). What is the shortest file where the `SetIOBufferAddress` version will take less time in system calls?

Compute the same answer assuming that we read and write the file 1024 bytes at a time.

16. In the chapter, we discussed standard input and standard output. UNIX also defines open file identifier 2 as *standard error*. Why would they do this? *Hint*: Think about errors in pipelines.
17. Suppose you wanted to write a program that could take input from one of two possible sources and count the length of the input. The two input are either:
 - A pipe or a file.
 - A message queue or a file.

Discuss which of the programs would be easier to write. Relate this to device independence as mentioned in Section 3.5.

18. What are the advantages of named pipes (with names in the file system) over pipes with internal names? Are there any disadvantages?

19. Suppose we want to eliminate all the arguments from the `read` system call. We define a new system call: `int CreateIOState( int fileDescriptor, char * bufferAddress, int count )`. This sets all the I/O parameters for a particular I/O state, and returns a system identifier for that I/O state. You can create as many I/O states as you want. You can create an I/O state once and use it on as many system calls as you want. The restriction is that each call will have exactly the same parameters, but fortunately this is the usual case (see the examples in this chapter). Each `read` and `write` system call has only one parameter: the I/O state identifier that was returned by `CreateIOState`. Rewrite the file copy program using this form of I/O system calls.

Examine the tradeoffs between this kind of I/O interface and the original interface we defined. In which cases is this better and in which cases is it worse?

20. Make the file copy program into an executable program and run it. Compare it against the system copy (`cp`) program for speed, size, and robustness.
21. Consider the message-sending process in Section 3.7.2. What will happen if the file being sent contains a byte with value 0?
22. Consider the message-creation process in Section 3.7.2. There is a potential bug in the program related to the fact that message queue ids are 32-bit integers. Explain the bug.
23. Consider the message creation process in Section 3.7.2. This program contains a "storage leak" in `itoa`. Explain what this is.
24. Suppose we wanted to use message passing to replace `argc/argv`. Explain how this could be done using named message queues. *Hint:* Use the process identifier to create a standard name for an argument-passing message queue.
25. Should a message queue be destroyed automatically when the process that created it exits? Give a reason for doing this and a reason against doing it.
26. Modify the file sender and receiver processes to send as many characters at a time as possible given the message format we defined. *Hint:* You will have to include a count field.
27. Consider the following suggestion: Since we can send messages to running programs, we do not really need to pass arguments to processes when they are created (in `CreateProcess` or `exec`). Instead we could just send them messages containing the arguments. Would this scheme work? Is there anything that we can do with passing argument to created processes that we cannot do with this scheme? How is one better than the other?
28. Suppose we wanted to change the message system so that you sent messages directly to another process rather than to a message queue. The message queue concept will not be visible to the programmer. Redesign the message queue system calls in Section 3.7.1 for this message system and describe your calls as we described our calls in that section. Then redo the message-passing example in Section 3.7.2 to use your new message-passing system calls.

29. Suppose we wanted to add a command separator in our simple shell in Section 3.13. When it reads a semicolon (;) it will end the command and execute it before executing the rest of the command line. Modify the shell to do this.
30. Modify the shell in Section 3.13 to allow a "&" at the end of a command. If a command ends with a "&" it will not wait for the command to finish but will return immediately with another prompt.
31. About 10 lines from the end of the shell code is a line `exit(1)`. It looks like this line exits the shell if you type in a command that it cannot find in the current directory or in the `/bin` directory. Why is this not so?
32. Modify the shell in Section 3.13 to handle pipes. Assume the scanner will return a token of type PIPE. *Assume:* A system call `pipe(fds)` that takes an array of two integers as an argument. When it returns, it will have set up a pipe where `fds[1]` is an open file identifier of the writing end of the pipe and `fds[0]` is an open file identifier of the reading end of the pipe. *Hint:* Handle each part of the pipeline in a separate iteration of the main loop. Use variables to remember the pipes that are outstanding.
33. Does it make sense to run a shell in another shell?

Glossary

A

absolute path name—An absolute path name is a path name that starts at the root directory. In UNIX, this means a path name that start with “/”.

access control list—An access control list is a list of access control items associated with a object to be protected. Each access control item specifies a user or a group of users and a set of operations that user or group of users can perform on the object protected by the access control list.

accounting—Accounting is keeping a record of the resource use by users and processes. Accounting is done by a resource manager.

address space—An address space is a set of addresses each referring to a byte in physical memory or in another address space. The addresses in an address space are usually contiguous and usually start with 0.

adjoin—To adjoin two name spaces is to combine them into a single name space. A name is looked up in the adjoined name space by looking it up in the first name space and, if it is not found, looking it up in the second name space.

aging—Aging is a technique to avoid starvation in a scheduling system. It works by adding an aging factor to the priority of each request. The aging factor must increase the request's priority as time passes and must ensure that a request will eventually be the highest priority request (after it has waited long enough).

allocation—Allocation is the assignment of a resource to a process which made a request. Allocation is done by a resource manager.

architecture—The architecture of a system is the general structure of the system. The architecture describes the main components of the system, what the general function of each component is, and the relationship between the components.

asm block—An asm block is a block of assembly language code that is recognized by a C++ compiler and placed directly in the output assembly code at the point it occurs in the source code.

asynchronous—An asynchronous system call is a system call that returns immediately and does not wait for the activity initiated by the system call to complete. For example, an asynchronous system call to read a disk block will return after the block read is requested but before the block is actually read.

atomic action—An atomic action is an action by a process that completes without interruption by any other process.

audit log—An audit log is a record of all security-related events that occur in an operating systems. A security-related event is any event that accesses a system resource. The record of each event will include the operation requested, the object the operation is requested on, the user and process requesting the operation and the time of the request.

auditing—Auditing is the activity of keeping an audit log of all requests to access system resources.

authentication—Authentication is the activity of making sure that a user is really who he or she says he or she is. This usually involves giving a password.

B

bank switching—Bank switching is a hardware technique where two physical memories occupy the same addresses but at different times. It is a technique for allowing the use of more physical memory in a limited address space.

base register—A base register contains a base address that is added to each memory address generated by a process running in user mode. A base register is a hardware register that can only be written in system mode. Base registers are used for address mapping and memory protection. Also see bound register.

batch operating system—A batch operating system is an operating system that runs user jobs in “batches”. Batch operating systems allow the use of faster I/O devices like tapes instead of card readers.

batching—Batching is a technique to avoid starvation in resource scheduling. A set of requests is collected into a batch and all the requests in the batch are serviced before any new requests are considered.

best fit—Best fit is a memory allocation algorithm where a request for a block of memory is always filled with the free block that is as small as possible and still satisfies the request.

binary semaphore—A binary semaphore is a semaphore where the count can only be zero or one.

binding time—The binding time of an attribute (of an object) is the time that the value of that attribute is determined. For example, the binding time of the address of a local variable is when the procedure it is declared in is called.

block device—A block device is an I/O device that works basically like a disk, that is, that reads and writes data in fixed size blocks, that has an address for each block, and that allows blocks to be read in any order, by address. The other type of device is the character device.

block list—A block list is a list of allocated and free blocks in a dynamic memory system. It is searched to satisfy a memory request and it is modified when a block is allocated or freed.

blocked—A process is blocked if it cannot continue executing until some event has occurred. Typical events that a process might be blocked waiting for are: the completion of an I/O operation, the arrival of a message in a message queue, the freeing of a block of memory, etc.

blocking send—A blocking send is a send that might block if a buffer is not available, the network is not available, the receiver is not ready to receive the message, or the receiver has not yet replied to the message.

booting—Bootting is the process of loading the operating system into the computer memory and initializing the operating system.

bound register—A bound register is a hardware control register that can only be written in system mode. A bound register contains a limit that is compared with each memory address generated by a process running in user mode. If the address is at or above the limit then it is deemed to be an illegal address and the instruction that generated the address causes an interrupt. This is a method of memory protection. Also see base register.

buddy system—The buddy system is a method of memory management where all blocks come in pairs and bigger blocks are created by merging a block with its paired (or "buddy") block. All blocks can also be divided up into two paired blocks. The most common form of the buddy system is the binary buddy system where all blocks have a size that is a power of two and paired blocks are created by dividing a block into two equally sized pieces.

buffering—Buffering is the process of using memory to store output of a producer process until a consumer process is ready to read it. This allows the producer and consumer to operate at different rates.

bursty—A producer is bursty if it tends to produce data at an uneven rate, that is, it might produce a lot of data in a short time and then no data for a while after that.

busy waiting—A process or processor is busy waiting if it is waiting for an event to occur by inquiring, over and over again, if the event has occurred. For example, a processor can busy wait for a memory cell to become non-zero by reading it again and again and checking if it is still zero.

C

cache coherence problem—If a cache is recording some parts of a memory and it is possible for the memory to be changed without the cache being changed then steps must be taken to ensure that the cache is kept up to date with the actual contents of memory. This is called the cache coherence problem. This problem occurs when two processors share memory and each processor can modify the memory independently of the other processor. A popular solution to the cache coherence problem is the "snoopy" cache which watches all bus transactions to see if any of the memory cells it is caching are written into.

cache hit—A cache hit occurs when an item is found in the cache.

cache hit rate—The cache hit rate is the probability that a request will be found in the cache.

cache miss—A cache miss occurs when an item is not found in the cache.

caching—Caching is an optimization technique that is used whenever a process has to compute the value of a function over and over again and it is faster to remember previous values than to compute them over again. The caching technique remembers one or more previous values of the function and does not recompute them if they are requested again. The most common example of caching is a cache memory which consists of fast memory that remembers the values of some locations in a larger, slower memory. If the same location is requested a second time then it is fetched from the faster cache memory. This makes the effective access time of the memory smaller.

capability—A capability is the right to perform a specific operation on an object.

capability list—A capability list is a set of capabilities that is associated with a process or user. As such it represents the things that the process is allowed to do. A capability list defines a protection domain.

CD—A CD is a compact disk. It is used to store digital data. A CD is accessed in the same manner as a magnetic disk, that is, it has addressable sectors that are addressed by sector number or by a cylinder, track and sector number.

channel—A channel is an I/O processor that can transfer data to and from memory and can execute an I/O program.

character device—A character device is a computer device that reads or writes a continuous stream of data. It has no fixed block size and that data do not have addresses. The other type of device is the block device.

character generator memory—A character generator memory is the memory in a terminal that determines the bit patterns that will be drawn for each ASCII character.

checkerboarding—Checkerboarding is another name for fragmentation, that is, a situation in dynamic memory management where many of the free blocks are too small to be allocated.

child process—A child process is a process that was created by, and is under the control of, another process (called the parent process).

cipher text—Cipher text is encoded text.

client process—A client process is a process that requests service from a server process.

client-server model—The client-server model is an architecture for operating systems or any systems of processes. In the client-server model, all system services are provided by a set of server processes, each of which provides a coherent set of operations. Client processes make requests for service to the server processes.

clock algorithm—A clock algorithm is one of a family of page replacement algorithms. A clock algorithm keeps a list of page frames and finds a page to replace by cycling through this list of page frames looking for a page frame that satisfies certain criteria. One possible criteria is that the referenced bit of the page is off indicating that the page has not been referenced recently.

clustering—Clustering is the grouping of a set of contiguous pages for some purpose. It is most often used for paging in where the entire cluster is paged in if any one of the pages gets a page fault. This causes new processes to load their working set with fewer page faults. The term clustering is also used for the technique of always moving a set of pages or disk blocks in and out of memory as a unit. This type of clustering is a way of increasing the effective page size or disk block size.

color map—A color map is a hardware device that maps from eight bit pixel values to 24 (or 32) bit color values. Color maps are used to reduce the size of the display memory needed in a frame buffer.

compaction—Compaction is a technique used in dynamic memory allocation for creating a large free block. In compaction all the allocated blocks are moved together and all the free blocks are collected into one large free block. Also see disk compaction.

component name—A component name is the name of an object in a directory. A path name or a compound name consists of a sequence of component names.

compound name—A compound name is a name that contains several parts. Each part is called a component name and is looked up in a directory or name space.

compressed file—A compressed file is a file that has been processed by a compression algorithm that reduces the amount of redundancy in the file and so reduces its size. Compressed files take up less space on a disk and are faster to send over communication lines.

condition variable—A condition variable is a variable used in a monitor to represent a condition or event. The two operations on a condition variable are wait and signal. A process that waits on a condition variable will be blocked until the next signal on that condition variable.

confinement problem—When a program runs and handles private data, the user might want to ensure that the program does not reveal the data to any other user. Making sure that a program does not do this is called the confinement problem.

connectionless—A connectionless protocol is one in which each message is independent of all other messages. A connectionless protocol does not require a setup or open operation.

consistency checking—A file system contains redundant data that must be consistent. For example, no block can be both free and part of a file. Consistency checking is the process of going through a file system and making sure that the redundant data is consistent. Consistency checking is done after a system has crashed and there is a possibility that the file system data are inconsistent.

context—A context of a process is the information needed to start the process running on the processor. In other words, a process context is the set of values of all processor registers that affect the execution of the process.

context switch—A context switch is the activity of saving the context of a running process and restoring the context of another process.

contiguous—A set of memory addresses is contiguous if the addresses are in sequence with no missing addresses.

contiguous file—A contiguous file is a file that is stored in contiguous disk blocks, that is, disk blocks with contiguous addresses. Contiguous files can be read faster than files organized in other ways.

control register—A control register is a hardware register in a processor that controls the execution of the processor. The processor status word is an example of a control register. Control registers can only be written in system mode.

copy-on-write—A copy-on-write page in a virtual memory system is mapped into two (or more) virtual address spaces as a read-only page. If the page is written into in any of the virtual address spaces then a private copy of the page is made for that address space and marked read-write. Copy-on-write is a technique for doing lazy copies where the copy is not done until the last possible moment.

counting semaphore—A counting semaphore is the same thing as a semaphore.

covert channel—A covert channel is a means of transmitting information that is not intended for information transfer. An example is transmitting a bit of information by making a tape drive move or not.

CPU—A CPU is a central processing unit or processor. It is the part of a computer that can execute instructions.

critical section—A critical section is a section of code in a program that accesses a set of shared variables. It is necessary to maintain mutual exclusion of the critical section for a set of

shared variables, that is, no two processes can be executing in their critical section at the same time.

current directory—The current directory (also called the working directory) is the starting point of relative path names. Usually a user is working with the files in the current directory.

curses—Curses is a library of functions that can control a wide range of terminal types by accessing a terminal control database.

cursor position—The cursor position in a terminal or text window is the place where text is inserted.

D

daemon—A daemon is an operating system process that wakes up periodically and performs some operation. For example, a paging daemon will wake up every do often and make sure there are an adequate number of free page frames.

DAT—A DAT is a digital audio tape. It is a means of storing digital data and is used in computer systems for disk backups.

database access and update—Database access and update is a pattern of interprocess communication where there is a database and readers and writers of the database. Readers can share the database but writers must have exclusive access to the database.

deadline scheduling—Deadline scheduling is a form of scheduling where each process has a deadline by which it must be completed. The scheduler tries to ensure that all processes complete by their deadline.

deadlock—Deadlock is a situation where a group of processes are all blocked and none of them can become unblocked until one of the others becomes unblocked. The simplest deadlock is two processes each of which is waiting for a message from the other.

deadlock prevention—Deadlock prevention is a technique for avoiding deadlock problems. A deadlock prevention technique makes rules so that deadlock is impossible. One technique for deadlock prevention is to require that all processes ask for all their resources in a single request, that is, a process is not allowed to request a resource while it is still holding another resource.

decryption key—A decryption key is a binary number that is the input for a decryption algorithm to decode a cipher-text message to a plain-text message.

defined external symbol—A defined external symbol is a symbol that is defined in a module and it declared external so that other modules can refer to it.

demand paging—Demand paging is the paging policy that a page is not read into memory until it is requested, that is, until there is a page fault on the page.

dense—A set of integers is dense if they are consecutive.

design problem—A design problem consists of a set of requirements (things to be accomplished) and a set of constraints (limitations on the possible solutions). The design problem is to find a solution that meets the requirements and satisfies the constraints.

design process—The design process is the set of steps to be taken in designing an artifact. The design process generally involves the stages of: stating the problem clearly, giving a description of the external appearance and behavior of the artifact, describing the internal structure of the solution, and implementing the artifact.

design space—The design space is the set of all possible solutions to a problem. Most of them will not meet all the requirements and constraints of the design problem.

device controller—A device controller is a hardware device that connects to the system bus of a computer and also connects to a device. It manages the communication between the device and the processor and memory of the computer system.

device driver—A device driver is a software component that communicates with a device controller and device. Device drivers present a uniform device interface to the operating system. Often there is a device driver for each device controller but it is possible for a device driver to communicate with several device controllers and several devices.

device independence—An operating system provides device independence if a program can read and write data from a range of devices and files with no changes to the program.

device switch—A device switch is a table in an operating system that converts device numbers into the addresses of entry points into the device driver for that device.

device virtualization—A virtualization of a device is a data structure and set of procedures (which use the data structure) that simulate the device being virtualized. The virtual device has a set of operations with the same overall functionality as the real device. Usually the virtual device is easier to use than the real device.

digital signature—A digital signature is a technique that "signs" a message so that a receiver can be certain who sent the message. It is the digital equivalent of a written signature on a document.

directory—A directory is a data structure in a file system that maps names into objects. The objects mapped are either files or other directories.

dirty bit—The dirty bit is a bit in the page table entry that indicates whether the page has been written to since the last time the dirty bit was cleared. Also see modified bit.

disk address—A disk address is the address of a disk block. It is either a sector number or a triple consisting of the cylinder, track and sector number of the disk block.

disk block—A disk block is the smallest unit which a disk can read or write. Each disk block has a disk address. Disk blocks are typically 512 bytes to 4 K bytes.

disk block pointer—A disk block pointer is an integer that contains a disk sector number. It acts as an address on the disk since it uniquely specifies a disk block.

disk cache—A disk cache is an area in memory that contains copies of the disk blocks that have been recently read or written. Disk caches save disk I/O because disk blocks that are in the cache do not need to be read from the disk.

disk compaction—Disk compaction is the process of moving the blocks on the disk around so that the blocks of a single file are contiguous and in order on the disk. After disk compaction it is faster to read files because there are no seek or latency delays.

disk driver subsystem—The disk driver subsystem is the part of the simple operating system that handles the system calls to read and write disk blocks.

disk extent—A disk extent is a sequence of contiguous disk blocks that belong to the same file. Some file systems store files in extents rather than in single disk blocks. This makes the files more contiguous and faster to read and write.

dispatch—To dispatch a process means to select it to be the next process to execute and the restore its execution context which starts it running.

dispatcher—The dispatcher is the part of an operating system that is called when the operating system has finished its work and it ready to run a process. The dispatcher picks a process to run and starts it running by restoring its execution context.

distributed operating system—A distributed operating system is an operating system that runs on a network of computers but gives the users the illusion that they are running on a single large system with one operating system. The users of a distributed system do not need to know where on the network their files or processes are.

DMA—DMA stands for direct memory access. A device controller is a DMA controller if it can read and write the memory over the system bus without the processor.

double buffering—Double buffering is a technique for speeding up the processing of a disk file by reading one buffer while processing another buffer (or writing one buffer while filling another buffer).

double index block—A double index block is a block that consists of the a series of disk block addresses of single index blocks. A single index block is a disk block that contains a series of addresses of disk blocks that contain file data.

dynamic data—The dynamic data area is the part of the memory of a running process that is used to satisfy requests by the dynamic memory allocator (the *malloc* or *new* allocator).

dynamic programming—Dynamic programming is a technique similar to caching where a problem is solved by building up a table of solutions to subprograms and use the table entries to solve larger and larger version of the problem.

dynamic relocation—Dynamic relocation is the relocation of memory addresses while the program is running.

E

eager evaluation—Eager evaluation is the computation of a function as soon as the arguments to the function are available. It is the opposite of lazy evaluation.

efficiency—Efficiency is a goal of a resource manager. An efficient use of resources is one where the least amount of the resource is used to accomplish the stated goal. Operating systems worry about efficient use of processor time and memory space.

elevator algorithm—The elevator algorithm is a disk scheduling algorithm where the disk head moves like an elevator in the building. The disk head moves in one direction (say towards in center of the disk) as long as there are requests in that direction. If there are no requests in a direction then it starts moving in the other direction.

emulator—An emulator is a program that implements (in software) the instruction set of a processor.

encryption key—An encryption key is a number that is the input to an encryption algorithm and is used to encode a plain-text message into a cipher-text message.

event model—The event model is used in graphic user interfaces (GUIs). In the event model, all events of interest to a process are combined into a single stream of events. Each event contains the type of the event, the time of the event and the parameters of the event. The main control loop of a process using the event model is a single loop which gets and then acts on events. Typical events are: keystrokes, mouse movement, mouse button clicks, input on a pipe, etc.

execv—The UNIX system call to run a new program in an existing process.

external fragmentation—External fragmentation happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cannot be effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced.

external name—An external name is a name intended for use by users of the operating system. External names are usually chosen by the user and are usually character strings.

external security—External security refers to security techniques not related the the operating system. This includes things like locks on the doors to the computer room.

external symbol—An external symbol in an object module is a symbol that is defined in this module and used in another module or defined in another module and used in this module.

F

fairness—A scheduling algorithm is fair if it treats all processes of the same priority equally.

FCFS—FCFS stand for first-come, first-served. It is another name for FIFO (first-in, first-out). In FCFS the job that has been waiting the longest is served next.

fifo—Fifo stand for first-in, first-out. In a fifo queue, the item that has been in the queue the longest is removed next. The term fifo is also used for a pipe that is named in the file naming system.

FIFO page replacement—The FIFO page replacement algorithm removes the page that has been in memory the longest.

file—A file is a collection of related information that is stored on a disk and that has a unique name.

file alias—A file alias is another name for a file. A file may have several aliases. File aliases allow a file to reside in more than one directory in the file system. This allows it to be classified in several ways. In UNIX a file alias is called a symbolic link.

file allocation table (FAT)—A file allocation table is an inverted disk block index.

file descriptor—A file descriptor is a data structure maintained by the operating system that contains all the meta-data the operating system needs to record about the file. File descriptors contain things like: the last time the file was read, the last time the file was written, the length of the file, the location of the file data on disk, the protection status of the file, etc. Another name for a file descriptor is a file control block.

file extension—A file extension is the part of the file name after a period and it indicates the type or use of the file.

file location—The file location is the offset into the file where the next read or write operation will start.

file mapping—File mapping is a technique where a disk file is mapped into the virtual address space of a process. File mapping can take the place of disk I/O.

file system—A file system is a data structure on a disk or disk partition. It contains files and directories. A file system is the largest unit of organization on a disk.

file system descriptor—A file system descriptor is a data structure on disk where the operating system keeps all the information it needs to record about the file system. Typical information kept in a file system descriptor is: the length of the file system, the lo-

cation of the root directory, the location of the file descriptors, the beginning of the chain of free blocks, etc.

file system layout—The file system layout is the way space is allocated in a file system. For example, the layout of the original UNIX file system contained: the boot block, the superblock (the file system descriptor), an area for inodes (file descriptors), and an area for data and index blocks.

filter—A filter is a program that reads a stream of ASCII characters, processes it in some way, and writes out a stream of ASCII characters.

first fit—First fit is a memory allocation algorithm where a request for a block of memory is always filled with the first free block in the block list that satisfies the request.

first-come, first-served scheduling—First-come, first-served scheduling always chooses the job that has been waiting the longest.

first-in, first-out—In a first-in, first-out queue, the item that has been in the queue the longest is removed next.

flat file—A flat file is a file that is simply a sequence of bytes with no larger structure. An example of a file that is not a flat file is a file with a record structure.

flat name space—A flat name space is a single list of names with no internal structure in the names. Each name in a flat name space must be unique.

fork—A fork is the UNIX method for creating a new process. The fork system call will create an identical copy of the forking process.

fragment—A fragment is a block of memory in a dynamic memory allocation system that is too small to satisfy any request.

fragmentation—Fragmentation occurs in a dynamic memory allocation system when many of the free blocks are too small to satisfy any request.

frame buffer—A frame buffer is a memory module that stores the image that is on a graphics display. The image on the display is refreshed from the frame buffer. The image is changed by changing the frame buffer. A frame buffer has from 1 to 32 bits per pixel on the graphics display.

framework—A framework is a shell of an application program that contains the major modules but does not contain any actual functionality.

free block—A free block is a piece of memory in a dynamic memory allocation system that is not currently in use and so is available for allocation.

full backup—A full backup of a file system is a copy (usually on tape) of every file in a file system.

full path name—A full path name is an absolute path name.

G

general purpose register—A general purpose register in a computer system is a register that can be used by the programmer for any purpose required. This is in contrast with a control register which can only be written in system mode and is used for on specific purpose.

global replacement—A global page replacement algorithm is an algorithm that looks at all the page frames in the system to decide which one to replace. This is in contrast with a local page replacement algorithm which will only replace pages that belong to the process the just had a page fault.

graphics context—A graphics context is a collection of information which gives the current default arguments to graphics operations. A graphics context contains things like: the background color, the foreground color, the fill color, the line width, the copy mode, etc.

guard page—A guard page is a page that is placed at either end of a data structure in virtual memory (for example, a stack, an array or a table). The guard page cannot be read or written and so if a process tries to write beyond the limits of the data structure it will get a protection fault when it tries to access the guard page. This can be useful in detecting array subscripts out of bound, stack overflow, table overflow, etc.

H

half context switch—A half context switch is either the saving or the restoring of the registers of a process.

handle—A handle is a pointer to a pointer (which then points to the object). It is a form of indirection and makes it easy to move objects that there are many pointers to.

hard link—A hard link is a form of file alias used in UNIX that takes advantage of the fact the directory entries contain only inode (file descriptor) numbers and not the inodes themselves. So two directory entries can refer to the same inode. Each such entry is a hard link. Also see file alias.

hardware interface—The hardware interface is the collection of hardware items that are visible to the machine language programmer.

hardware resource—Hardware resources are: processor time, memory, device time and disk space. The operating system manages the hardware resources of the system.

header section—The header section is the part of an object module or a load module that comes at the beginning of the modules and describes the module and the layout of the rest of the module.

heap—A heap is a source of managed memory that is dynamically allocated.

hierarchical file naming system—A hierarchical file naming system is a system for naming files using a tree of files and directories. Each interior node in the tree is a directory and its children are the items in the directory. Hierarchical file naming system use path names to name files.

highest response ratio next scheduling—Highest response ratio next scheduling is a scheduling algorithm that always chooses the job with the highest response ratio. The response ratio of a job is the ratio of the waiting time plus the execution time divided by the waiting time. Highest response ratio next scheduling is really shortest job first with aging.

hint—A hint is the answer to a question that is likely to be correct but is not guaranteed to be correct.

hook—A hook is a mechanism that allows a user to get control when a specific event occurs.

horizontal retrace—The horizontal retrace is the time period where the electron beam in a cathode ray tube is moving from the end of one scan line to the beginning of the next scan line.

I

I/O address space—The I/O address space is an address space used to address I/O devices. It is separate from the memory address space and used by special I/O instructions (in computers that have I/O instructions).

I/O device—An I/O (input/output) device is an electro-mechanical device that converts digital data from internal, digital form to some external representation (magnetic variations, light, sound, inks marks on paper, etc.)

I/O processor—An I/O processor is a processor that can transfer data to and from memory and can execute an I/O program.

I/O system—The I/O system is the part of an operating system that handles I/O operations and devices. The I/O system consists principally of device drivers plus some device-independent code that manages I/O requests and device driver invocation.

implementation—The implementation of an interface is the code that performs each function and that contains entry points for all the functions of the interface.

incremental backup—An incremental backup of a file system is a copy of all the files in the file system that have changed since the last backup was made.

index block—An index block is a disk block that contains the addresses of other disk blocks. It is used by file systems to record the location of the disk blocks containing file data.

indirection—Indirection is the term for the process of getting from one place to another by going through an intermediate agent. A pointer to a pointer is the simplest example.

initial process—The initial process is the first process started by an operating system. It is the only process that is not created by a create process system call made by a running process.

initialized data section—The initialized data section is the part of an object module or load module that contains the data that is explicitly given an initial value by the assembly language source code.

inode—An inode is a file descriptor in UNIX.

input device—An input device is an I/O device that takes input from an external source and converts it to digital form for internal use in a computer system.

inter-process communication—Inter-process communication (IPC) refers to any method by which one process can communicate with another. Examples are: messages, pipes, ports, shared memory and RPCs.

interface—An interface is a contract between two software entities. One entity writes code using the functions specified by the interface and the other entity writes code that implements the functions in the interface.

interleaved file—An interleaved file is a file that is stored in every other disk block or every third disk block or in some regular pattern like those. If the location of the first block of an interleaved file is known then the location of all the other blocks in the file can be computed from it.

internal fragmentation—Internal fragmentation is the space wasted inside of allocated memory blocks because of restrictions on the allowed sizes of allocated blocks (that is, on the granularity of allocation).

internal name—An internal name is a name used mainly inside an operating system. Internal names are usually integers and are usually chosen by the operating system (not the user).

internal security—Internal security is the security involving the operating system. This is opposed to external security which does not involve the operating system.

interrupt—An interrupt is a hardware action taken in response to an event. The action the hardware takes is to interrupt the execution of the currently running program and jump to an interrupt handler. Interrupts are caused by device controllers (when they complete a I/O transfer), timers (when they count down to 0), system call instructions, instruction errors, etc.

interrupt vector area—The interrupt vector area is the area in memory where the hardware looks for interrupt vectors. An interrupt vector tells the hardware what routine to jump to when an interrupt occurs.

inverted disk block index—An inverted disk block index is an array of disk block addresses as long as the number of disk blocks on the disk. The value of the array entry for a disk block is the next disk block in the file. This means one can

chain through the file allocation table to find all the disk blocks in a file.

inverted page table—An inverted page table is a page table which is indexed by page frame number rather than by page number. Inverted page tables are better for very large address spaces where ordinary page tables would be too long.

IPC—IPC stand for inter-process communication. Some common forms of IPC are: messages, ports, pipes and RPCs.

J

just-in-time—A just-in-time strategy is a lazy strategy where an operation is not performed or a resource is not allocated until the time that it is actually needed.

K

kernel—The kernel is the essential part of the an operating system that performs the basic functions that every operating system must do: process dispatching, basic memory management, and protection.

kernel thread—A kernel thread is a thread that runs in kernel mode.

kernel-mode process—A kernel-mode process is a process that is running in system mode. A user-mode process changes to a kernel-mode process when it makes a system call (or gets an external interrupt) and starts executing in operating system code.

keyed file—A keyed file is a file of records which contain a designated field called a key. Records in the file can be accessed by giving the value of the key rather than the record number.

L

language interface—A language interface is an interface in the form of a language, that is, an interface in which one can write a program. The alternative is an interface that consists of a collection of procedures that one can call from a separate programming language. A separate programming language is not required to use a language interface.

late binding—When an attribute of an object is assigned a value we call that binding the attribute. If the binding is done very close to the last possible time it could be done then we call this a late binding. For example, demand paging is an example of late binding because we do not assign physical storage to a page until the page is first used, that is, at the latest possible moment.

latency time—The latency time of a disk transfer is the time taken up waiting for the disk platter to spin around so that the required data is under the read/write heads of the disk drive.

Law of Diminishing Returns—The law of diminishing returns says that the more you invest in something the lower the return on investment will be. For example, as you begin trying to debug a program you will find lots of bugs without much effort but as time goes on and your time investment in debugging goes up, the number of bugs you will find will go down. The reason is that there are fewer bugs to find and they are the harder ones to find because the easier ones have already all been found.

lazy copy—A lazy copy is one that is done just by copying the page tables and setting the pages to copy-on-write status. Pages are copied as they are modified, that is, at the last possible moment.

lazy creation—Lazy creation is a technique where the creation of an object is delayed until the object is actually needed. For example, a graphical program will not create a window until a user action requires that the window be displayed.

lazy evaluation—Lazy evaluation is a way of implementing argument passing in a programming language. Arguments are not evaluated when they are passed but the first time they are used.

library—A library is a collection of object modules that can be included, as needed, in a program by a linker.

lightweight process—A lightweight process is a thread that is implemented by the operating system.

linker—A linker is a software tool that combines object modules and libraries into an executable load module. The principle activities of a linker are relocation and linking.

linking—Linking is the process of combining two or more object modules and fixing up the places where one module calls a procedure in another module or accesses data in another module.

little language—A little language is a special purpose language that provides access to a set of functions. For example, the output formatting language of *printf* format strings is a little language.

Little's Law—Little's Law is a formula that is true in all systems where objects enter the system, remain in the system for a while and then leave the system. Little's Law says that $N = \lambda T$ where N is the average number of objects in the system, λ is the arrival rate and T is the average time an object spends in the system.

load control—A virtual memory system must not allow too many processes to be running and competing for page frames or else the system will thrash. So a virtual memory system must have a load control policy that makes sure that some processes are swapped out if too many are competing for memory.

load module—A load module is a file that contains a program that is ready to run and that can be loaded by the run-time loader of the operating system.

load time dynamic linking—Some linkers delay the linking of library modules until the program is being loaded for execution. This is called load time dynamic linking.

local replacement—A local page replacement algorithm is one that always replaces a page from the process that caused the page fault. The opposite of this is global replacement.

locality—Locality is the tendency of accesses to memory to be for the same memory locations (temporal locality) or nearby memory locations (spatial locality) that have been accessed recently.

log structured file system—A log structured file system is a file system that records all file system changes in a change log. The entire file system is the change log. This means that all writing is done in sequential locations in the log.

logical address—A logical address is an address that must be translated by memory management hardware into a physical address before it can be presented to the hardware.

logical address space—The logical address space of a process is the set of addresses that it uses internally. These addresses are translated to physical addresses by the memory management hardware.

logical block—A logical block is a block of information in a file. The logical block is the information not the storage medium where the information is kept.

logical block number—The logical block number of a block is the block number in the file.

logical disk—A logical disk is a software object that acts like a disk, that is, it is a sequence of disk blocks. A logical disk might actually be only part of a physical disk or it might actually span several physical disks.

logical parallelism—Logical parallelism in an operating system comes from time sharing. The processor is rapidly switched between processes and it gives the appearance of the processes actually running in parallel but logical parallelism does not involve any true, physical parallelism.

logical resource—A logical resource is a software object that acts like a physical resource.

LRU page replacement—The LRU page replacement algorithm always replaces the page that has not been used for the longest time.

M

machine code section—The machine code section of an object module or a load module is where the executable machine code is stored.

magic cookie—A magic cookie is a number that appears arbitrary to the process that holds it but has a meaning to the server process that it was obtained from and is sent back to with operations. An example would be an index into a table in the logical address space of the server process. This index has no meaning in another process but has meaning in the server process.

masked—Interrupts can be masked by the hardware so they will not be taken even if they are requested.

master page number—The master page number is the part of a virtual address that gives the offset into the master page table for the process. This is used in two-level (or more) paging systems.

mechanism—A mechanism is a means of accomplishing a task that does not dictate how to accomplish the task. For example, an access control list mechanism allows you to protect your files but it does not require you to, for example, make all your files read-only.

medium-term scheduler—A medium-term scheduler decides which processes should be in memory competing for page frames and which processes should be swapped out.

memoizing—Memoizing is the technique of speeding up a recursive function by keeping a memo table of previous arguments and values of the function. Memoizing is a form of caching.

memory address space—The memory address space is that part of the physical address space of a computer system that has physical memory assigned to it.

memory as a device—A device driver can act like a disk driver but actually access the physical (or a logical) address space when a process reads or writes the simulated disk. This is a good way to give some processes access to the physical address space and protect access with the file protection system.

memory manager—A memory manager is a software module that dynamically allocates memory.

memory mapped I/O—A system uses memory mapped I/O if the device registers are accessible through the physical address space.

memory object—An object in the Mach operating system that is a source of pages. Normally a memory object gets its pages from a file but the source could be a network connection or the memory object could generate the pages on demand.

memory protection—Memory protection is a hardware mechanism that allows the user or the operating system to decide who can access memory and how they can access it. Memory protection mechanisms are most commonly combined with memory mapping mechanisms but some, like lock and key systems, for example, are not connected with a memory mapping system.

memory-mapped file—A memory-mapped file is a file that is mapped into virtual memory. See file mapping.

message—A message is the content of a communication between two processes. Message passing is a common form of inter-process communication.

message queue—A message queue is a data structure that holds messages that have been sent but not yet received. Message queues are normally maintained by the operating system and kept in operating system memory.

meta-information—Meta-information is information about information. A file contains information that can be of any kind the user wishes. The meta-information associated with the file contains things like: the length of the file, the time the file was last opened for reading, the time the file was last opened for writing, who can access the file and how, where on the disk the file information is kept, etc. File meta-information is maintained by the operating system.

micro-kernel—A micro-kernel is a minimal operating system that performs only the essential functions of an operating system. All other operating system functions are performed by system processes.

model—A model is a representation of a system. For example, an equation can be a model for the motion of a rocket, a simulation program can be a model for a bank, and the client-server model describes the general architecture of a large class of systems of processes.

modem—A modem is a modulator-demodulator. It converts digital data into tones that can be sent over a phone line (modulation) and converts tones back into digital data (demodulation). Modems allow the connection of a terminal to a computer over a phone line.

modern cryptography—Cryptography is the science of secret codes (creating codes and breaking codes). Modern cryptography is cryptography done with computers. The advent of computers allowed cryptographic techniques that would have taken too long without computers.

modified bit—The modified bit (also called the dirty bit) is a bit in the page table entry for a page that is set whenever the page is written into. This information is important to virtual memory systems since a page that has been modified must be written back out to the swap area before the page frame can be reused.

monitor—A monitor is a mechanism for process synchronization. It consists of a collection of procedures where the monitor enforces mutual exclusion between them, that is, the monitor only allows one process at a time to be executing in any of the procedures in the monitor. Monitors also provide condition variables to allow processes to wait for specific events.

mono-programming—Mono-programming is running only one program at a time in a computer system. Mono-programming was used in early mainframe computer operating systems and early personal computer operating systems.

monolithic—A monolithic operating system is one where all operating system code is in a single executable image and all operating system code runs in system mode.

mounting—Mounting is a technique whereby one file naming hierarchy is grafted on to a directory in another file naming hierarchy. Mounting is a way of combining two (or more) file naming hierarchies that combines them into a single space of unique

names. Typically a file system on a logical disk is mounted onto a directory in another file system. Mounting can be done between file systems on a single processor or between file system on different computer systems that are connected with a network.

multicomputer—A multicomputer is a computer system with two or more computers each with its own memory and with a communications network between the computers.

multiple queue scheduling—Multiple queue scheduling is a scheduling mechanism where there are several levels of queues from high to low priority. Processes move to lower priority queues as they use up their time slices on higher priority queues.

multiplexing—Multiplexing is a method of sharing a resource between processes by dividing it up into pieces or dividing the allocation up into small pieces of time. See space division multiplexing and time division multiplexing.

multiprocessing—Multiprocessing is the technique of running two or more processes in an operating system, in parallel, on two or more processors. Multiprocessing creates physical parallelism.

multiprogramming—Multiprogramming is the technique of running several programs at a time using timesharing. It allows a computer to do several things at the same time. Multiprogramming creates logical parallelism.

mutex—A mutex is another name for a binary semaphore. It is also a favored variable name for a mutual exclusion semaphore.

mutual exclusion—Mutual exclusion is a mechanism for avoiding race conditions by preventing two processes from running in their critical section at the same time.

N

N-step scan algorithm—An N-step scan algorithm is a batching variation on the scan (or elevator) algorithm where disk requests are taken in batches of requests and all the requests in a batch are satisfied before any new requests are considered. It is a way of avoiding starvation of some disk requests.

name—A name is an external identifier of an object. Names are normally chosen by the user and are normally character strings.

name assignment—Name assignment is the activity of assigning a name to an object. This might be done by the operating system or by the process creating the object.

name lookup—Name lookup is the activity of finding an object from its name. This often entails a table lookup or a series of table lookups.

name map—A name map is a function from names to objects. It is used in name lookup.

name resolution—Name resolution is another name for name lookup. It is the activity of finding an object from its name.

name space—A name space is a collection of unique names. A name space normally is associated with a name map that maps each name into an object.

name table—A name table is a lookup table for mapping names into objects.

network operating system—A network operating system is an operating system that is aware of the network it is on and can cooperate with other network operating systems on the network to pass messages and provide for a distributed file system.

next fit—Next fit is a memory allocation algorithm where a request for a block of memory is always filled with the next free block in the block list that satisfies the request.

nonblocking receive—A nonblocking receive is a message receive system call that will not block if there are no messages to receive but instead will return with an error code. Most receive system calls are blocking.

not recently used—Not recently used is another name for the clock algorithm. It is so named because it chooses pages for replacement if they have not been recently used.

O

object—An object is a resource that we are protecting in a protection system. Subjects are the active entities that request to perform operations on objects.

object module—An object module is the result of an assembly or compilation. It contains the machine code, the data, the relation information and the symbol table for the code module.

object space—An object space is a collection of objects that is the range of a name map.

off-site backup—An off-site backup is a set of backups (usually tapes) that is kept at a different location (off-site) from the disks that are being backed up. This means that a physical event, such as a fire or tornado, cannot destroy both the disks and the backups.

one-time password—A one-time password is a password that is used only once. One-time passwords are generated by an algorithm. The advantage is that a wire-tapper cannot reuse the password.

open file—An open file is an object implemented by the operating system that allows a process to read or write a file (or any other source or sink for data).

open file identifier—An open file identifier is an internal name given to an open file by the operating system. It is passed back to the operating system each time an operation on the open file is invoked. Usually it is an index into the open file pointer table.

open file pointer table—An open file pointer table is a table of all the files a process has open. It is maintained by the operating

system. It is indexed by an open file identifier and contains pointers to entries in a system-wide open file table.

open file table—An open file table is another name for an open file pointer table.

operating system—An operating system is the software that controls the main resources of a computer system. It manages those resources and creates virtual versions of them.

operating system interface—The operating system interface is the set of system calls the operating system implements. Another view of these system calls is a set of object types that operating system implements and the operations on those objects.

operation—An operation is a function that can be performed on an object.

optimal page replacement—An optimal page replacement algorithm is an algorithm that always produces the fewest page faults for any page reference sequence.

optimistic algorithm—An optimistic algorithm is one that assumes the best (or a good) case and so can perform more efficiently when the assumption is valid. The algorithm can detect when the assumption is false and recover. For example, an optimistic mutual exclusion algorithm will assume that no other process is in its critical section or will start its critical section while this process is in its critical section.

output device—An output device is an I/O device that takes data from an internal, digital form and converts it to an external form.

overlay—An overlay is a module in a program that is brought into memory on demand, while the program is running. Two or more overlays will share the same memory addresses.

P

packet—A packet is the unit of data transfer in some networks (for example, in Ethernets). It is some data along with an address of the intended receiver.

page—A page is the contents of an area of virtual memory in a running process. All pages are the same size and are aligned on addresses that are multiples of the page size. The page is the information not the medium that records it so the same page can be in a page frame in memory or in a disk block on the disk.

page fault frequency load control—Page fault frequency load control is a method of load control that monitors the page fault rate. If the rate is too low then new processes are brought into memory and if the page fault rate is too high then one or more processes is swapped out of memory.

page fault interrupt—A page fault interrupt occurs when a memory reference is made to a page that is not in memory. The present bit in the page table entry will be found to be off by the virtual memory hardware and it will signal an interrupt.

page fault rate—The page fault rate is the number of page faults divided by the number of memory references. It is the likelihood that a memory reference will cause a page fault.

page frame—A page frame is an area of physical memory that can hold a page. The page is the information (the bits) and the page frame is the medium that stores the information. A disk block that can hold a page is also called a page frame.

page reference string—When a process is running it generates a series of memory references. If we take the page number that each reference refers to then the resulting sequence is a page reference string. Page reference strings are used as input to simulators that evaluate paging algorithms.

page replacement algorithm—A page replacement algorithm decides which page to replace when a free page frame is needed.

page table—A page table is an array of page table entries. A page number is derived from a memory address and that page number is used as an index into the page table to get the page table entry for that page. The page table entry is then used to find the page frame in memory where the page resides.

page table entry—A page table entry records the information about a page. This information includes: whether the page is in memory, the page frame where the page is located in memory and the types of operations allowed on the page (read, write, and/or execute).

pager—A pager is a software object in the Mach operating system that handles paging from a memory object into and out of memory.

paging daemon—A paging daemon is an operating system process that wakes up every so often and makes sure that the free page frame pool has not gotten too low. If it has, the paging daemon tries to free some page frames.

parallel port controller—A parallel port controller is a hardware device that connects to the system bus and also connects to one or more devices through a parallel port. The parallel port controller interfaces the device to the system bus.

parent process—A parent process is a process that has created one or more children and so is their parent. A parent process can usually control many aspects of its child processes.

partition—A partition is a area covering part of a physical disk. A partition can be configured as a logical disk.

path name—A path name is a compound name consisting of one or component names. Each component name can be looked up in a directory to find the directory in which to look up the next component name. The final name in the path name refers to the object that is named by the path name.

pattern—A pattern is an outline of a design that can be reused in several design situations.

per process open file table—The per process open file table is the list of open files that is associated with a process. It generally contains pointers to entries in a system-wide open file table.

persistent object—A persistent object is an object that will continue to exist even after the program that creates it has terminated. A file is the most common example of a persistent object.

persistent storage—Persistent storage is storage that will retain its value after the process that creates and writes it has terminated. A file is the most common example of persistent storage.

phosphor—A phosphor is a chemical that will glow for a short period of time after it has been hit with a beam of electrons. Phosphors are used in cathode ray tubes (like TVs and graphic displays) where an image is created by making some parts of the phosphor coating of the tube to glow.

physical address—A physical address is an address that can be placed on the system bus and will be responded to by a hardware module on the bus (usually a memory module but possibly some other device).

physical address space—The physical address space of a computer system is the set of addresses that are valid on its system bus. Physical addresses do not require any translation before being used.

physical block—A physical block is a block on the disk. It is addressed with a physical block number.

physical block number—A file consists of a sequence of logical blocks which have logical block numbers. The disk block address of a logical block is called the physical block number of that block. In other words, the physical block number is the address of a block on the disk.

physical parallelism—Physical parallelism is parallelism caused by two (or more) processes each executing a process at the same time.

pipe—A pipe is a method of inter-process communication. It is accessed like a file. One process writes to the pipe and when another process reads from the pipe and it will read the same bytes, in the same sequence, that the first process wrote.

pixel—A pixel is a picture element. It is a dot on a graphic display screen that can be turned on or off. The patterns of pixels that are on creates the image on the graphics display.

placement—The placement problem in a dynamic memory system is the decision about where in memory to place a block. This is a problem in dynamic memory systems that use variable sized blocks but it is trivial in a paging system where all blocks are the same size and are always placed at addresses that are multiples of the page size.

plain text—A message is plain text before it is encoded. After encoding it is cipher text.

policy—A policy is a set of rules that is implemented by a mechanism. For example, a protection policy would be to only allow people in the Friends group to have read access to your files. The UNIX file protection mechanism could be used to implement this policy by setting the permission bits the right way.

policy versus mechanism—A mechanism is a means for implementing a policy. A policy is a set of rules for deciding what will be done and the mechanism is the means by which the rules are enforced.

port—A port is a message queue that only one process can read from. In the Mach operating system, a port represents an object. Operations on the object are performed by sending messages to the port. The process reading from the port is the server or manager for that object.

powers of two allocation—This is a method of dynamic memory allocation where all block sizes must be a power of two.

PPP—PPP stand for point-to-point protocol. PPP is a method of connecting to a network over a phone line. It consists of a process on one side that acts (to processes that want to use the network) like a network connection but really communicates over a phone line to another process. That other process is on a machine that is physically connected to the network and it performs the network operations for the other process.

predictive load control—Predictive load control is a form of load control where the algorithm tries to predict the load in the near future and takes steps to make sure it doesn't get too high or too low.

preemptive scheduling—Preemptive scheduling is a form of scheduling where the scheduler is allowed to allocate a resource to a process and then, later, take away the resource even though the process is not finished using it. Processor schedulers in computers are almost always preemptive schedulers whereas disk head scheduler almost never are.

prepaging—Prepaging is a paging strategy where the paging algorithm tries to predict which pages will be used in the near future and pages them into memory before they are requested, that is, before there is a page fault on them.

present bit—The present bit in a page table entry is a bit that indicates whether the page has been read into a page frame in memory or not. If a page is referenced when the present bit is off then a page fault interrupt will occur.

primary page table—The primary page table is the first page table used in a multiple level (usually two, three, or four level) paging system.

principle of minimum privilege—The principle of minimum privilege requires that a process should only have the privileges it needs to get its job done, and no more. It is used in protection systems to limit the damage when a program does not work correctly or is a Trojan horse program.

priority—The priority of a process is an indication of how important the process is. A priority scheduler will always favor higher priority processes.

priority scheduling—Priority scheduling is a scheduling method where, at all times, the highest priority process is assigned the resource.

private key cryptosystem—A private key cryptosystem is a system of encoding where the encoding key and the decoding key are the same or are easily computable from each other. In a private key cryptosystem both the encoding key and the decoding key must be kept private or the code will be compromised.

private scheduling queue—A private scheduling queue is a queue that only one process will ever wait on. The scheduler can always schedule that process by releasing any process waiting on the private scheduling queue. Usually this is the queue of a private scheduling semaphore.

privilege—A privilege is the right to perform a specific operation on a specific object. For example, a process might have the privilege of reading file *SecretData*.

privileged instruction—A privileged instruction is a hardware instruction that can only be executed when the processor is in system mode. Privileged instructions are instructions that affect the management of the hardware resources.

process—A process is an operating system concept that captures the idea of a program in execution. A process can hold resources (e.g., open files, address space, protection privileges, etc.) and it can be run by the dispatcher. In other words, it has resources to compute with and gets assigned a processor to do the computing. A process is always executing at some point in a program. A process has memory for its program and data. A process has a copy of the hardware registers that describe where in the computation it is.

process competition—Processes interact by competing when they are sharing resources that both of them need but only one can be using a time.

process cooperation—Processes interact by cooperating when they are working together on a task where each process is doing part of the task.

process descriptor—A process descriptor is a data structure in an operating system that contains all the information the operating system needs to record about the process. This information includes: the saved register state of the process, what memory is allocated to the process, where the process's page table is located, the groups the process is in, how much processor time and other resources the process has used, the files a process has open, etc.

process hierarchy—The process hierarchy is a tree where each process is a node in the tree. The child nodes of a parent node are the child processes of that parent process.

process management subsystem—The process management subsystem is the part of the simple operating system that implements processes. It does process creation, process dispatching and system call handling.

process state—While it is running, a process can be in one of several processes states. The minimal set of process states is: ready (to run), running (on a processor), and blocked (waiting for an event).

process table—The process table is a data structure in an operating system that contains the process descriptor of each process in the system. Any operation on a process will require that its process descriptor be looked up in the process table.

processor—A processor is the part of a computer system that executes instructions. It is also called a CPU.

processor mode—A processor is always running in a processor mode which determines which instructions are allowed to be executed. Most processors have two modes: user mode (where some instructions, called privileged instructions, are not allowed) and system mode (where all instructions are allowed).

processor sharing—Processor sharing is the theoretic limit of round robin where each ready process gets its share of the processor in all time periods, even very short ones.

producer-consumer—When two processes are cooperating on a task, it is common for one process to produce partially completed objects which are then passed on to the other process. We call the first process the producer, the second process a consumer and their relationship is called producer-consumer relationship.

program—A program is a set of instructions that can be used to control a process.

program phase—A program phase is a period of time in the execution of a program when the working set of pages changes slowly.

programming interface—A programming interface to a program is an interface where another process can invoke all the functions of a program. Some programs can only be controlled from the user interface.

protection—Protection is a mechanism in an operating system that checks whether a subject is allowed to perform an operation on an object.

protection database—A protection database is a database of all the protection information in a system. An important issue in the design of a protection system is how and where the protection database is stored.

protection domain—A protection domain is a set of privileges, that is, a set of rights to perform specific operations on specific objects. A process is always running in some protection domain. In some protection systems, a protection domain is a protected object and there is an operation to enter a new protection domain.

protection field—The protection field is in a page table entry and records the protection information for the page. Generally this is whether it is allowed to read, write and/or execute the page.

protection monitor—A protection monitor is a software module that checks the protection database to ensure that a requested operation is allowed.

pseudo-parallel—Two processes run in pseudo-parallel if they are timesharing a processor. Another term for pseudo-parallel is logically parallel.

pseudo-tty driver—A pseudo-tty driver is a device driver that pretends to be connected to a terminal but is really connected to a process.

public key cryptosystem—A public key cryptosystem is a system of encoding where the encoding key and the decoding key are different and the decoding key is not easily computable from the encoding key. In a public key cryptosystem only the decoding key must be kept private. The encoding key is the public key and can be widely disseminated.

Q

quantum—A quantum (or time slice) is the period of time a process is allowed to run before the operating system takes control back. This is enforced with a timer which is set before the process is started and causes an interrupt when the quantum is up.

R

race condition—A race condition is a situation where two processes either communicate or share memory while they are running and the output of one or both of the processes depends on the relative speed of the two processes. Race conditions are considered to be bad because computers should be deterministic. The reason the outputs are different depending on the relative speeds is because in some cases the output is wrong.

RAID—RAID stands for redundant array of inexpensive disks. A RAID device uses several disk drives in parallel to get higher transfer rates and/or more error correction.

RAM disk—A RAM disk is a disk driver that acts like a disk but actually stores the data on the “disk” in memory. The advantage of RAM disks is that they are much faster than regular disks. The disadvantage is that they are much more expensive than real disks and the data in them is not persistent between reboots of the operating system.

random page replacement—The random page replacement algorithm chooses a page to replace randomly when asked to select a page for replacement. It uses no information about past behavior of the program. Random page replacement is generally not a good page replacement algorithm but its worst case is the same

as its best case and so its worst case is better than most page replacement algorithms.

raster graphics—Raster graphics is a graphics technology where images on a display are produced from tiny dots on the display. The alternative to raster graphics is vector graphics which produces images by drawing them directly.

reactive system—A reactive system is a system that does not do things on its own but only to external events. User interfaces are mostly reactive in that they don't do anything unless the user initiates an action. Operating systems are basically reactive systems.

readers-writers problem—With a shared database it is generally acceptable to have several readers reading the database simultaneously but a writer must have exclusive access to the database. Maintaining these rules is called the readers-writers problem. Another name for this problem is the database access and update problem.

ready—A process is in the ready state if it is ready to run as soon as a processor is available. That is, the process is not waiting for an event such as the completion of a disk transfer.

ready list—The ready list is a list of all the processes in the system that are in the ready state. This list is a subset of the process table and is often a linked list running through the process table.

real-time operating system—A real-time process is a process that must respond to events within a certain time period. A real-time operating system is an operating system that can run real-time processes successfully.

recursive address space—A recursive address space is a virtual address space whose page tables are in another virtual address space.

recursive design problem—Often when you are trying to solve a design problem you will try a technique that leads to another instance of the same design problem. This is called a recursive design problem. Usually the recursive design problem will have different constraints than the original design problem and so different (and usually simpler) solutions are acceptable.

redirection—Redirection is the technique of changing the standard input or standard output of a process so that it gets its input or writes its output to a different place than normal. The normal place for input or output is the terminal running the program and it is redirected to or from a file.

redundancy—Redundancy is the technique of keeping more than one copy of data to protect against the data being lost.

reentrant program—A program is reentrant if it can be executed by two threads at the same time and it executes correctly for both threads. The thing that prevents a program from being reentrant is the use of global or static data. Another name for reentrant is thread-safe.

reference string—A reference string is a page reference string.

referenced bit—A referenced bit is a bit field in a page table entry that records whether the page has been reference since the last time the referenced bit was cleared. The paging hardware will set the referenced bit (for a page) every time it references a word on a page.

refreshing—Refreshing is the activity of continually rewriting the graphics image on a graphics display. Refreshing is necessary because the image on the display fades away rapidly after it is written.

region—A region is a contiguous span of pages in a sparsely populated virtual address space.

relative device—A relative device is a positioning device that returns relative movement rather than absolute locations. A mouse is a relative device whereas a touch sensitive screen is an absolute device.

relative path name—A relative path name is a path name that begins at the the current directory.

rendezvous—A rendezvous is a pattern of interprocess communication where two or more processes need to wait until they are all at a certain point in their respective programs.

replacement—Replacement is the problem of deciding which memory block to replace when making room for a new one.

reservations—Some scheduling systems allow a user to reserve a resource ahead of time. This is called a reservation. This is common in scheduling situations where the cost of waiting for a resource when it is needed is high.

resident set—The resident set of a process is the set of pages that are currently in memory.

resource management—Resource management consists of: accounting for a resource, scheduling the resource, protecting the resource, multiplexing the resource and transforming the resource.

right—A right is the ability to perform a specific operation on a specific object. Another name for a right is a permission.

round-robin scheduling—Round-robin scheduling is a scheduling method where each process gets a small quantity of time (called a quantum or time slice) to run and then it is preempted and the next process gets to run. This is called time sharing and gives the effect of all the processes running at the same time (this is called logical parallelism) albeit on much slower processors.

RPC—An RPC is a remote procedure call. RPC is a method of interprocess communication where one process communicates with another process by making a procedure call that crosses to the address space of that process and runs and the result is sent back to the address space of the caller. RPC is a method of transforming message passing so it looks like procedure calling.

run time dynamic linking—Run time dynamic linking is a technique where a library module is not linked and loaded until the first time it is called while the program is running.

running—A process is in the running state when it is executing on a processor. In a multiple processor system it is possible for two or more processes to be in the running state at the same time.

S

saturated—A queuing system is saturated if it is running near its maximum capacity and cannot absorb any more load.

scan algorithm—The scan algorithm (or elevator algorithm) is a disk scheduling algorithm where the disk head moves like an elevator in the building. The disk head moves in one direction (say towards in center of the disk) as long as there are requests in that direction. If there are no requests in a direction then it starts moving in the other direction.

scan line—A scan line in a CRT display is a horizontal line of pixels that represents one sweep of the electron beam across the screen.

scheduling—Scheduling is the activity of deciding when processes will receive the resources they request.

scripting language—A scripting language is a programming language intended mostly for interactive use and for one-shot programs that invoke the functions of other programs.

SCSI—SCSI stands for small computer system interface. SCSI is a bus standard that allows several devices of different kinds to be connected to a computer system through a single SCSI controller.

SCSI device driver—A SCSI device driver is a device driver that manages a SCSI bus. Sometimes more specialized device drivers (like disk drivers) will use the SCSI device driver to send messages on the SCSI bus.

second chance algorithm—The second chance algorithm is a form of the clock algorithm where pages that are modified but that have not been used in one clock hand rotation (hence their used bit is 0) are given one more clock hand rotation to be accessed again before they are removed. This second clock hand rotation is their second chance.

secondary page number—The secondary page number is the part of the virtual address that gives the page number of the page in the secondary page table. The secondary page table is located through the page table entry in the primary (or master) page table which is indexed by the primary (or master) page number.

sector—A disk sector is the unit of information that a disk can read and write. It is also called a disk block. A disk sector is normally from 512 bytes to 16 K bytes.

security policy—The security policy in a computer system is a set of security rules that the protection mechanisms are used to enforce.

seek—A seek is the physical movement of the read/write heads in a disk system to the cylinder where the next read or write will

take place. Most disk drives will automatically seek to the correct cylinder before a read or write operation although they also have a specific seek operation.

segment—A segment is an area in a virtual address space that contains data used for one purpose, that is, a code module, an array, a table, etc.

segment register—A segment register is a hardware register that contains the base address and length of a segment.

segmentation—Segmentation is a method of structuring virtual memory where each logical part of the program (a procedure, a table, an array, a storage pool, etc.) is placed in a separate segment. Segmentation leads to two-dimensional addressing where each byte is addressed by segment number and byte within the segment.

semaphore—A semaphore is a synchronization primitive with two operations: wait and signal (also called P and V). A semaphore is implemented with two data structures: a counter and a queue of blocked processes. The wait operation decrements the counter (if the counter is positive) or (if the counter is 0) blocks the process invoking the wait operation and puts it on the queue. The signal operation increments the counter (if the queue is empty) or (if the queue is not empty) removes a process from the queue and unblocks it.

separation of concepts—Separation of concepts is a design technique where a concept is divided into two concepts which can each be used separately. This can lead to greater flexibility and reusability.

separator character—A separator character is an ASCII character that is used to separate the parts of a compound (that is, multi-part) name. The separator character in UNIX is ':' (/user/jeff), in MS/DOS is '\ (user\jeff), in MacOS is ':' (:user:jeff), etc.

serial port—A serial port is a connection which allows bit-serial (one bit at a time) communication between a device and a computer system.

serial port controller—A serial port controller is a device controller for connecting to serial devices like modems, printers, terminals, etc.

serially reusable resource—A serially reusable resource is a resource that can only be used by one process at a time but can be used by another process once the process currently using it is finished with it.

server—A server is a software process or hardware machine that provides services to other processes or machines over a network. A typical example is a file server which provides file services over a network.

server process—A server process is a process that acts like a server, that is, that accepts requests for services and responds to them.

settling—A disk drive seeks by moving the heads in a single ballistic movement to the approximate place it expects to find the

track it is seeking to. After this movement, it figures out which track it actually got to and makes a few small adjustments to get to the correct track. These small adjustments are called settling.

shared-memory multiprocessors—A shared-memory multiprocessor is a computer system with several processes that all share some common memory. The processors may or may not have private memory of their own.

shell—A shell is a program that allows a user interactive access to the functions of an operating system. Another name for a shell is a command line processor.

short-term scheduler—A short-term scheduler decides which of the processes in memory will get to use the processor and for how long they will get to use it.

shortest job first scheduling—The shortest job first scheduling (SJF) scheduling algorithm is a non-preemptive scheduling algorithm that chooses the job that will execute the shortest amount of time.

signal—A signal is a notification that a specific event has occurred. A signal has a type but no information content beyond its type.

signaling—Signaling is a basic interprocess communication pattern where process A sends a signal to process B when process A reaches a specified point in its code. Process B will wait for that signal when it gets to a specific point in its code.

social engineering—Social engineering refers to a range of non-technical methods of gathering information that will assist in breaking into a computer system. These methods include: calling people on the phone and getting information from them (usually by lying to them), going through trash, getting a job as a janitor in a company, illegally breaking into an office, etc.

software interrupt—A software interrupt forces a process to execute an interrupt handling procedure when a specific event occurs. It is used to allow a process to respond quickly to events without having to poll for them. It mimics in software the effects of a hardware interrupt.

software page table lookup—Some newer computers do not have hardware to do page table lookups. Instead the hardware manages TLB lookups and interrupts to the operating system when a TLB lookup fails. The operating system then does the page table lookup to find the correct page table entry in software.

space division multiplexing—Space division multiplexing is a technique for sharing a resource by allocating parts of it to different processes. Memory is a good example of a resource that is allocated using space division multiplexing.

space resource—A space resource is a resource that can be divided up into pieces and can be used for space division multiplexing.

spatial locality—A process exhibits spatial locality if it is the case that if a memory cell is accessed then the memory cells

around it are likely to be accessed very soon. Processes exhibit spatial locality in code because of sequential execution and loops and they exhibit spatial locality in data because variables in a procedure are using stored together and used together and because arrays are typically accessed sequentially.

spin lock—A spin lock is a busy-waiting lock where a processor checks a memory location over and over again to see if it contains a particular value.

spooling—Spooling is a technique for virtualizing a printer. It consists of writing printed lines to a disk file and later printing the disk file when the printer is not busy.

SSTF—SSTF stands for shortest seek time first. SSTF is a disk head scheduling algorithm where the request closest to the current position of the read/write heads (that is, the request the requires the shortest seek time) is chosen next.

standard input—The standard input of a process is an input file that is already opened and ready to read when the process is started. It defaults to the terminal the process was started from but it can be redirected from a file or another device.

standard output—The standard output of a process is an output file that is already opened and ready to write when the process is started. It defaults to the terminal the process was started from but it can be redirected to a file or another device.

starvation—Starvation is a resource management problem where a process does not get the resources it needs for a long time because the resources are being allocated to other processes.

state machine—A state machine is a mathematical model of a reactive system that accepts input data (or events), moves from state to state in response to these inputs, and sometimes produces outputs.

static data—Static data is data in a program that is local to a procedure but whose storage is allocated when the program begins and remains allocated (and retains its value) throughout the execution of the program.

static relocation—Static relocation is the technique of relocating a program to a new starting address before the program begins execution.

subject—A subject is an active entity in a protection system that will try to perform operations on objects.

superblock—A superblock is the UNIX name for a file system descriptor. It contains everything that UNIX needs to know about the file system.

swap area—A swap area is a disk partition (a logical disk) where the paging system stores pages that are not in memory. A swap area is not managed by the file system, it is managed directly by the paging system using the disk device drivers.

swapping—Swapping is a technique where processes are moved into and out of memory as memory is requested and becomes

free. Swapping is handled by a medium-term scheduler. Swapping is also used in paging systems where a process that is swapped out will release all its page frames for use by other processes.

symbol table—A symbol table is a part of an object module and a load module. The symbol table contains symbols defined in the module and their offset in the module. The symbol table also contains symbols used in the module (and defined in other modules) and the places in the module they are used.

symbolic link—A symbolic link is a file that refers to another file. A symbolic link is usually implemented as a short file that contains the name of the file it links to. Another name for a symbolic link is an alias (or a file alias).

synchronization—Synchronization is the activity whereby two processes coordinate their activities so that they can work together on a task.

synchronous—A synchronous system call is one where the action of the system call is completed before the system call returns. For example, if the system call requests that data be read from a file the data are read and placed in the user's buffer before the system call returns.

system call—A system call is a request by a process for the operating system to perform an operation for the process. This is usually an operation on an operating system object that the process cannot access directly. System calls are usually signaled by a system call instruction.

system call interface—The system call interface of an operating system is the set of system calls that the operating system implements. The system call interface defines the functionality of an operating system, that is, the objects it implements and the operations that are possible of these objects.

system call interrupt handler—The system call interrupt handler is the part of an operating system that first handles system calls. All system calls go first to the system call interrupt handler which determines which system call it is and directs the request to the appropriate part of the operating system.

system mode—System mode is a processor mode where all hardware instructions (including the privileged ones) are allowed.

system process—A system process is a process that runs in user mode but performs operating that are part of the operating system's functionality. System processes are part of the operating system but they do not run in system mode.

system stack—The system stack is the stack used by the operating system when it is running.

T

tape drive—A tape drive is an I/O device that can read or write data on a magnetic tape. Almost all of these now use cartridge tapes.

task—Task is the name used in the Mach operating system for process.

temporal locality—A process exhibits temporal locality if it is the case that if a memory cell is accessed then it is likely to be accessed again very soon. Processes exhibit temporal locality in code because of loops and they exhibit temporal locality in data because variables tend to be accessed several times in a row.

terminal capability database—A terminal capability database is a database of the instruction sets of a large number of terminals. The information about each terminal indicates how to do things like: clear the screen, write characters, move the cursor, make text bold, etc.

terminal emulator—A terminal emulator is a program that imitates the interface of a terminal. That is, it will accept the same commands the terminal accepts but write in a window rather than the terminal screen as a real terminal would do.

theory of program behavior—A theory of program behavior is the set of assumptions about how typical programs reference memory. Each paging algorithm tries to predict future page accesses from information about page references in the recent past. It makes this prediction using a theory of program behavior.

thrashing—If a paging system tries to load too many processes into memory then no process will have enough memory to run and so the page fault rate will get very high. As a result the system will be spending almost all of its time handling page fault interrupts. In this case the system is said to be thrashing.

thread—A thread is a software object that can be dispatched and executes in an address space. Several threads can exist in the same address space.

thread-safe—A program is thread-safe if it can be executed by two threads at the same time and it executes correctly for both threads. The thing that prevent a program from being thread-safe is the use of global or static data. Another name for thread-safe is reentrant.

three level paging—A three level paging system has three levels of page tables. The virtual address is divided into four parts: primary page number, secondary page number, tertiary page number and offset in the page. The primary page number is used to look up a page table entry in the primary (or master) page table. That page table entry points to the secondary page table and the secondary page number is used to find the page table entry of the tertiary page table. The tertiary page number is used to find the page table entry in that table. That page table entry will point to a page and the page offset is used to find the correct byte in that page.

throughput—Throughput is rate at which processes are serviced. Throughput is a measure of how efficiently a timesharing system is running.

time division multiplexing—Time division multiplexing is a technique for sharing a resource by allocating it to a processes

for a short time and then allocating it to the next process. Time division multiplexing is good for resources that cannot be shared but can be preempted. The processor is a good example of a resource that is allocated using time division multiplexing.

time resource—A time resource is a resource that cannot be shared. Processor-time is a good example.

time sharing—Time sharing is a technique for sharing the processor by giving each process use of the processor for short periods of time.

time slice—Time slice is another term for quantum.

timer interrupt handler—A timer interrupt handler is the interrupt handler that is called when the timer interrupts. Usually this means that a process has used up its quantum and must give up the processor to another process.

TLB—TLB stands for translation look-aside buffer.

transfer time—Transfer time is the time it takes to transfer a disk block (or blocks) to memory. A disk transfer incurs three delays: the seek time, the latency time, and the transfer time. Usually the transfer time is by far the shortest of these delays.

transformation—Transformation is the technique of creating a logical resource from a physical resource in such a way that the logical resource is an improved version of the physical resource. The logical resource will have simpler operations and may be more reliable than the physical resource.

translation look-aside buffer—A translation look-aside buffer is a set of hardware registers that act as a cache for the page table of the running process. When a virtual address is used, the virtual page is first looked up in the translation look-aside buffer. If it is found then the page table is not used and the page is accessed directly. If it is not found, then the page must be looked up in the page table. After it is looked up, the page table entry is stored in the translation look-aside buffer so it will be found the next time the page is looked up.

triple index block—A triple index block is a block that consists of a series of disk block addresses of double index blocks.

Trojan horse problem—One way to penetrate security in a system is to get a person run your program with their privileges. One way to do this is to create a useful program that people want to use but hide extra code inside it to do something else, like send a copy of all their files to you or to delete all their files. Such a program is called a Trojan horse program and the Trojan horse problem is how to prevent these programs from breaking your security system. The solution to the Trojan horse problem is to run all external programs in a protection domain that only allows them to do exactly what they are supposed to do.

true parallelism—True parallelism is the parallelism obtained from running processes on two or more physical processors. It is also called physical parallelism.

two dimensional address space—Segmentation produces a two dimensional address space. That is, all memory addresses have two parts, the segment number and the byte offset in the segment.

two handed clock—The two handed clock algorithm is a variation of the clock algorithm intended to free up page frame faster than the clock algorithm. The first hand visits all the page frames and turns off the referenced bit. The second hand follows the first hand by a certain distance (say 1000 page frames back). If the second hand encounters a page frame that is still not referenced (that is, it hasn't been referenced since the first hand passed it) then it immediately frees that page frame.

two level implementation—A system uses a two level implementation if the architecture is divided into two levels with the upper level being implemented using the functions defined in the lower level.

two level name space—A two level name space has a root directory which contains files and other directories but all the other directories can contain only files. Thus there can be only two levels of directories in any path name.

two level paging—A two level paging system has two levels of page tables. The virtual address is divided into three parts: primary page number, secondary page number and offset in the page. The primary page number is used to look up a page table entry in the primary (or master) page table. That page table entry points to the secondary page table and the secondary page number is used to find the page table entry of the page containing the byte we are accessing. That page table entry will point to a page and the page offset is used to find the correct byte in that page.

two phase locking—Two phase locking is a technique used in database access to avoid deadlock in record locking. In the first phase, a process only acquires locks and makes no changes to the database. If it fails to acquire a lock in the first phase, the process releases all its locks and starts over. After it has acquired all the locks, the process enters the second stage. In the second stage, it does not acquire any locks but it can make the changes in the database. At the end of the second phase, the process releases all its locks.

two queue scheduling—A two queue scheduling system is one where new processes go into the high priority queue for one quantum. If they do not finish in the quantum period then they go into the low priority queue. When the processor is free, the first process in the high priority queue is chosen (if the high priority queue is not empty), otherwise the first process in the low priority queue is chosen.

U

undefined external symbol—An undefined external symbol is a symbol that is used in a module and is defined in another module.

uninitialized data section—The uninitialized data section is the part of an object module or load module that contains the data that is not given an initial value by the assembly language source code.

user—A user is a person who is using a computer system (by running processes on it).

user mode—When a processor is in user mode it cannot execute privileged instructions.

user threads—A user thread system is implemented by the process itself and the kernel does not know anything about it. This is faster than kernel threads but the whole process blocks if any user thread blocks.

V

vector graphics—Vector graphics is a hardware technique where the graphics are drawn directly on the phosphor by the electron beam. The alternative is raster graphics.

vertical retrace—The vertical retrace is the time period where the electron beam in a cathode ray tube is moving from the end of the last scan line to the beginning of the first scan line.

virtual computer—A virtual computer is a software abstraction implemented by an operating system. All processes run on virtual computers. A virtual computer has all the power of the physical computer but it is easier to use.

virtual memory—Virtual memory is a hardware technique where the system appears to have more memory than it actually does. This is done by time sharing the physical memory and storing parts of the memory on disk when they are not actively being used.

virtual memory object—A virtual memory object is an object in the Mach operating system that manages a section of the virtual address space. It uses a pager to transfer pages from the memory object into the physical memory.

virtual resource—A virtual resource is a software object that has the same capabilities as a physical resource does but is easier to use. It is implemented using a physical resource.

virtual swap space—Some operating systems allow some pages not to have assigned swap space. This can work because some pages will always be in memory and a page in memory does not require swap space until it needs to be replaced.

virtual terminal—A virtual terminal is a software emulation of a terminal.

virtual time—Virtual time is the time a process has been actually executing, that is, it does not include the time it spends waiting to be executed.

w

working directory—The working directory (also called the current directory) is the starting point of relative path names. Usually a user is working with the files in the working directory.

working set—The working set of a process (at a point in time) is the minimal set of pages it needs to run without excessive paging. In the working set algorithm it is the set of pages that have been accessed in the last N memory references (this range of past memory references is called the working set window).

working set page replacement—The working set page replacement algorithm is a page replacement algorithm that tries to keep the working set of each process in memory. It works by monitor-

ing the working set of each process. If there is not enough room for the working set in memory then the process is swapped out until there is enough room.

worst fit—Worst fit is a memory allocation algorithm where a request for a block of memory is always filled with the free block that is as large as possible.

WSClock algorithm—The WSClock page replacement algorithm is a page replacement algorithm that tries to approximate the working set algorithm in the framework of a clock algorithm. It does this by approximating the time interval since each page has been referenced. If that interval is larger than the working set window then the page is replaced.