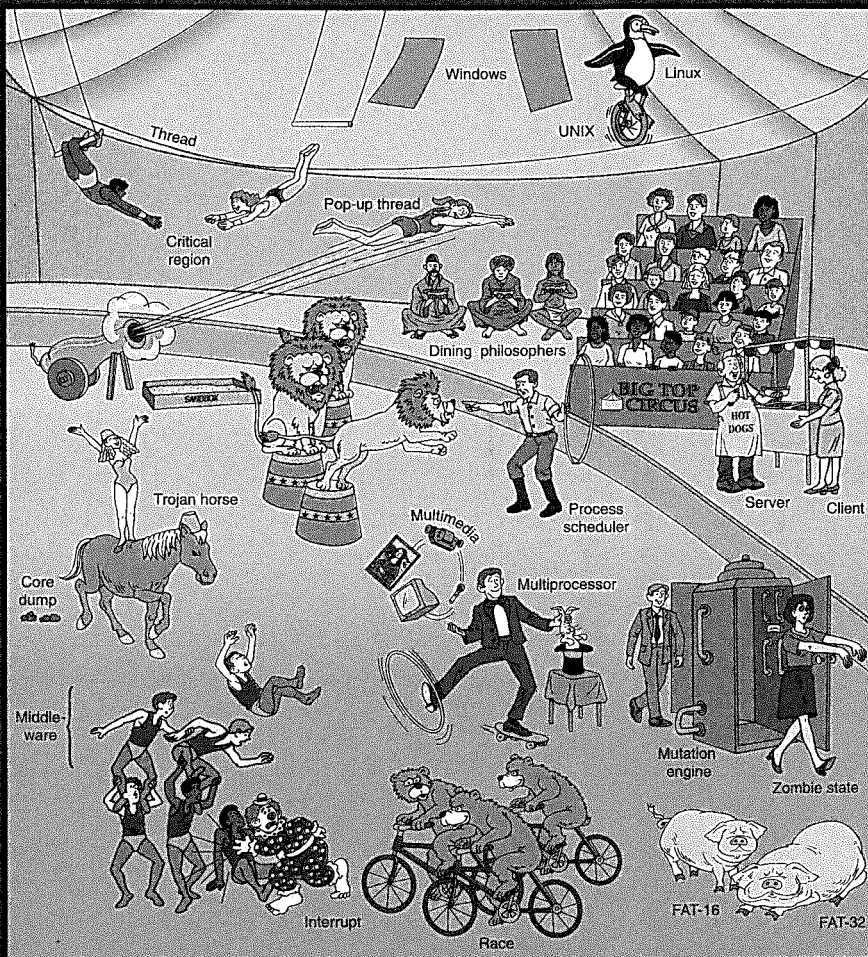


SECOND EDITION



Andrew S. Tanenbaum

MODERN OPERATING SYSTEMS

SECOND EDITION

ANDREW S. TANENBAUM

*Vrije Universiteit
Amsterdam, The Netherlands*



PRENTICE HALL

UPPER SADDLE RIVER, NEW JERSEY 07458

Library of Congress Cataloging-in-Publication Data

Tanenbaum, Andrew S.

Modern operating systems / Andrew S. Tanenbaum.--2nd ed.
p. cm.

Includes bibliographical references and index.

ISBN 0-13-031358-0

1. Operating systems (Computers) I. Title

QA76.76.O63 T359 2001

005.4'3--dc21

Vice President and Editorial Director, ECS: **Marcia Horton**

Publisher: **Alan Apt**

Associate Editor: **Toni D. Holm**

Editorial Assistant: **Amy K. Todd**

Vice President and Director of Production and Manufacturing, ESM: **David W. Riccardi**

Executive Managing Editor: **Vince O'Brien**

Managing Editor: **David A. George**

Senior Production Editor: **Camille Trentacoste**

Director of Creative Services: **Paul Belfanti**

Creative Director: **Carole Anson**

Art Director: **Heather Scott**

Cover Art: **Don Martinetti**

Cover Design: **Joseph Sengotta**

Assistant to Art Director: **John Christiana**

Interior Illustration: **Patricia Gutiérrez**

Interior Design and Typesetting; Cover Illustration Concept: **Andrew S. Tanenbaum**

Manufacturing Manager: **Trudy Piscioti**

Manufacturing Buyer: **Pat Brown**

Marketing Manager: **Jennie Burger**



© 2001 by Prentice-Hall, Inc.

Upper Saddle River, New Jersey 07458

The authors and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in this book. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks and registered trademarks. Where those designations appear in this book, and Prentice Hall and the authors were aware of a trademark claim, the designations have been printed in initial caps or all caps. All product names mentioned remain trademarks or registered trademarks of their respective owners.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5

ISBN 0-13-031358-0

Prentice-Hall International (UK) Limited, *London*

Prentice-Hall of Australia Pty. Limited, *Sydney*

Prentice-Hall Canada Inc., *Toronto*

Prentice-Hall Hispanoamericana, S.A., *Mexico*

Prentice-Hall of India Private Limited, *New Delhi*

Prentice-Hall of Japan, Inc., *Tokyo*

Pearson Education Asia Pte. Ltd., *Singapore*

Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

1

INTRODUCTION

A modern computer system consists of one or more processors, some main memory, disks, printers, a keyboard, a display, network interfaces, and other input/output devices. All in all, a complex system. Writing programs that keep track of all these components and use them correctly, let alone optimally, is an extremely difficult job. For this reason, computers are equipped with a layer of software called the **operating system**, whose job is to manage all these devices and provide user programs with a simpler interface to the hardware. These systems are the subject of this book.

The placement of the operating system is shown in Fig. 1-1. At the bottom is the hardware, which, in many cases, is itself composed of two or more levels (or layers). The lowest level contains physical devices, consisting of integrated circuit chips, wires, power supplies, cathode ray tubes, and similar physical devices. How these are constructed and how they work are the provinces of the electrical engineer.

Next comes the **microarchitecture level**, in which the physical devices are grouped together to form functional units. Typically this level contains some registers internal to the CPU (Central Processing Unit) and a data path containing an arithmetic logic unit. In each clock cycle, one or two operands are fetched from the registers and combined in the arithmetic logic unit (for example, by addition or Boolean AND). The result is stored in one or more registers. On some machines, the operation of the data path is controlled by software, called the **microprogram**. On other machines, it is controlled directly by hardware circuits.

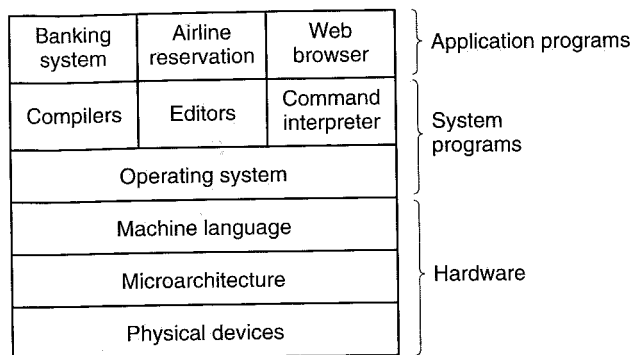


Figure 1-1. A computer system consists of hardware, system programs, and application programs.

The purpose of the data path is to execute some set of instructions. Some of these can be carried out in one data path cycle; others may require multiple data path cycles. These instructions may use registers or other hardware facilities. Together, the hardware and instructions visible to an assembly language programmer form the **ISA (Instruction Set Architecture)** level. This level is often called **machine language**.

The machine language typically has between 50 and 300 instructions, mostly for moving data around the machine, doing arithmetic, and comparing values. In this level, the input/output devices are controlled by loading values into special **device registers**. For example, a disk can be commanded to read by loading the values of the disk address, main memory address, byte count, and direction (read or write) into its registers. In practice, many more parameters are needed, and the status returned by the drive after an operation is highly complex. Furthermore, for many I/O (Input/Output) devices, timing plays an important role in the programming.

To hide this complexity, an operating system is provided. It consists of a layer of software that (partially) hides the hardware and gives the programmer a more convenient set of instructions to work with. For example, read block from file is conceptually simpler than having to worry about the details of moving disk heads, waiting for them to settle down, and so on.

On top of the operating system is the rest of the system software. Here we find the command interpreter (shell), window systems, compilers, editors, and similar application-independent programs. It is important to realize that these programs are definitely not part of the operating system, even though they are typically supplied by the computer manufacturer. This is a crucial, but subtle, point. The operating system is (usually) that portion of the software that runs in **kernel mode** or **supervisor mode**. It is protected from user tampering by the hardware (ignoring for the moment some older or low-end microprocessors that do not have

hardware protection at all). Compilers and editors run in **user mode**. If a user does not like a particular compiler, he† is free to write his own if he so chooses; he is not free to write his own clock interrupt handler, which is part of the operating system and is normally protected by hardware against attempts by users to modify it.

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

That said, in many systems there are programs that run in user mode but which help the operating system or perform privileged functions. For example, there is often a program that allows users to change their passwords. This program is not part of the operating system and does not run in kernel mode, but it clearly carries out a sensitive function and has to be protected in a special way.

In some systems, this idea is carried to an extreme form, and pieces of what is traditionally considered to be the operating system (such as the file system) run in user space. In such systems, it is difficult to draw a clear boundary. Everything running in kernel mode is clearly part of the operating system, but some programs running outside it are arguably also part of it, or at least closely associated with it.

Finally, above the system programs come the application programs. These programs are purchased or written by the users to solve their particular problems, such as word processing, spreadsheets, engineering calculations, or storing information in a database.

1.1 WHAT IS AN OPERATING SYSTEM?

Most computer users have had some experience with an operating system, but it is difficult to pin down precisely what an operating system is. Part of the problem is that operating systems perform two basically unrelated functions, extending the machine and managing resources, and depending on who is doing the talking, you hear mostly about one function or the other. Let us now look at both.

1.1.1 The Operating System as an Extended Machine

As mentioned earlier, the **architecture** (instruction set, memory organization, I/O, and bus structure) of most computers at the machine language level is primitive and awkward to program, especially for input/output. To make this point more concrete, let us briefly look at how floppy disk I/O is done using the NEC

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

PD765 compatible controller chips used on most Intel-based personal computers. (Throughout this book we will use the terms “floppy disk” and “diskette” interchangeably.) The PD765 has 16 commands, each specified by loading between 1 and 9 bytes into a device register. These commands are for reading and writing data, moving the disk arm, and formatting tracks, as well as initializing, sensing, resetting, and recalibrating the controller and the drives.

The most basic commands are read and write, each of which requires 13 parameters, packed into 9 bytes. These parameters specify such items as the address of the disk block to be read, the number of sectors per track, the recording mode used on the physical medium, the intersector gap spacing, and what to do with a deleted-data-address-mark. If you do not understand this mumbo jumbo, do not worry; that is precisely the point—it is rather esoteric. When the operation is completed, the controller chip returns 23 status and error fields packed into 7 bytes. As if this were not enough, the floppy disk programmer must also be constantly aware of whether the motor is on or off. If the motor is off, it must be turned on (with a long startup delay) before data can be read or written. The motor cannot be left on too long, however, or the floppy disk will wear out. The programmer is thus forced to deal with the trade-off between long startup delays versus wearing out floppy disks (and losing the data on them).

Without going into the *real* details, it should be clear that the average programmer probably does not want to get too intimately involved with the programming of floppy disks (or hard disks, which are just as complex and quite different). Instead, what the programmer wants is a simple, high-level abstraction to deal with. In the case of disks, a typical abstraction would be that the disk contains a collection of named files. Each file can be opened for reading or writing, then read or written, and finally closed. Details such as whether or not recording should use modified frequency modulation and what the current state of the motor is should not appear in the abstraction presented to the user.

The program that hides the truth about the hardware from the programmer and presents a nice, simple view of named files that can be read and written is, of course, the operating system. Just as the operating system shields the programmer from the disk hardware and presents a simple file-oriented interface, it also conceals a lot of unpleasant business concerning interrupts, timers, memory management, and other low-level features. In each case, the abstraction offered by the operating system is simpler and easier to use than that offered by the underlying hardware.

In this view, the function of the operating system is to present the user with the equivalent of an **extended machine** or **virtual machine** that is easier to program than the underlying hardware. How the operating system achieves this goal is a long story, which we will study in detail throughout this book. To summarize it in a nutshell, the operating system provides a variety of services that programs can obtain using special instructions called system calls. We will examine some of the more common system calls later in this chapter.

1.1.2 The Operating System as a Resource Manager

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

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

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

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

The other kind of multiplexing is space multiplexing. Instead of the customers taking turns, each one gets part of the resource. For example, main memory is normally divided up among several running programs, so each one can be resident at the same time (for example, in order to take turns using the CPU). Assuming there is enough memory to hold multiple programs, it is more efficient to hold several programs in memory at once rather than give one of them all of it,

especially if it only needs a small fraction of the total. Of course, this raises issues of fairness, protection, and so on, and it is up to the operating system to solve them. Another resource that is space multiplexed is the (hard) disk. In many systems a single disk can hold files from many users at the same time. Allocating disk space and keeping track of who is using which disk blocks is a typical operating system resource management task.

1.2 HISTORY OF OPERATING SYSTEMS

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

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

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

1.2.1 The First Generation (1945–55) Vacuum Tubes and Plugboards

After Babbage’s unsuccessful efforts, little progress was made in constructing digital computers until World War II. Around the mid-1940s, Howard Aiken at Harvard, John von Neumann at the Institute for Advanced Study in Princeton, J. Presper Eckert and William Mauchley at the University of Pennsylvania, and Konrad Zuse in Germany, among others, all succeeded in building calculating engines. The first ones used mechanical relays but were very slow, with cycle times measured in seconds. Relays were later replaced by vacuum tubes. These machines were enormous, filling up entire rooms with tens of thousands of vacuum tubes, but they were still millions of times slower than even the cheapest personal computers available today.

In these early days, a single group of people designed, built, programmed, operated, and maintained each machine. All programming was done in absolute

5

INPUT/OUTPUT

One of the main functions of an operating system is to control all the computer's I/O (Input/Output) devices. It must issue commands to the devices, catch interrupts, and handle errors. It should also provide an interface between the devices and the rest of the system that is simple and easy to use. To the extent possible, the interface should be the same for all devices (device independence). The I/O code represents a significant fraction of the total operating system. How the operating system manages I/O is the subject of this chapter.

This chapter is organized as follows. First we will look at some of the principles of I/O hardware, and then we will look at I/O software in general. I/O software can be structured in layers, with each layer having a well-defined task to perform. We will look at these layers to see what they do and how they fit together.

Following that introduction, we will look at several I/O devices in detail: disks, clocks, keyboards, and displays. For each device we will look at its hardware and software. Finally, we will consider power management.

5.1 PRINCIPLES OF I/O HARDWARE

Different people look at I/O hardware in different ways. Electrical engineers look at it in terms of chips, wires, power supplies, motors, and all the other physical components that make up the hardware. Programmers look at the interface

presented to the software—the commands the hardware accepts, the functions it carries out, and the errors that can be reported back. In this book we are concerned with programming I/O devices, not designing, building, or maintaining them, so our interest will be restricted to how the hardware is programmed, not how it works inside. Nevertheless, the programming of many I/O devices is often intimately connected with their internal operation. In the next three sections we will provide a little general background on I/O hardware as it relates to programming. It may be regarded as a review and expansion of the introductory material in Sec. 1.4.

5.1.1 I/O Devices

I/O devices can be roughly divided into two categories: **block devices** and **character devices**. A block device is one that stores information in fixed-size blocks, each one with its own address. Common block sizes range from 512 bytes to 32,768 bytes. The essential property of a block device is that it is possible to read or write each block independently of all the other ones. Disks are the most common block devices.

If you look closely, the boundary between devices that are block addressable and those that are not is not well defined. Everyone agrees that a disk is a block addressable device because no matter where the arm currently is, it is always possible to seek to another cylinder and then wait for the required block to rotate under the head. Now consider a tape drive used for making disk backups. Tapes contain a sequence of blocks. If the tape drive is given a command to read block *N*, it can always rewind the tape and go forward until it comes to block *N*. This operation is analogous to a disk doing a seek, except that it takes much longer. Also, it may or may not be possible to rewrite one block in the middle of a tape. Even if it were possible to use tapes as random access block devices, that is stretching the point somewhat: they are normally not used that way.

The other type of I/O device is the character device. A character device delivers or accepts a stream of characters, without regard to any block structure. It is not addressable and does not have any seek operation. Printers, network interfaces, mice (for pointing), rats (for psychology lab experiments), and most other devices that are not disk-like can be seen as character devices.

This classification scheme is not perfect. Some devices just do not fit in. Clocks, for example, are not block addressable. Nor do they generate or accept character streams. All they do is cause interrupts at well-defined intervals. Memory-mapped screens do not fit the model well either. Still, the model of block and character devices is general enough that it can be used as a basis for making some of the operating system software dealing with I/O device independent. The file system, for example, deals just with abstract block devices and leaves the device-dependent part to lower-level software.

I/O devices cover a huge range in speeds, which puts considerable pressure on the software to perform well over many orders of magnitude in data rates. Fig. 5-1 shows the data rates of some common devices. Most of these devices tend to get faster as time goes on.

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

Figure 5-1. Some typical device, network, and bus data rates.

5.1.2 Device Controllers

I/O units typically consist of a mechanical component and an electronic component. It is often possible to separate the two portions to provide a more modular and general design. The electronic component is called the **device controller** or **adapter**. On personal computers, it often takes the form of a printed circuit card

that can be inserted into an expansion slot. The mechanical component is the device itself. This arrangement is shown in Fig. 1-5.

The controller card usually has a connector on it, into which a cable leading to the device itself can be plugged. Many controllers can handle two, four, or even eight identical devices. If the interface between the controller and device is a standard interface, either an official ANSI, IEEE, or ISO standard or a de facto one, then companies can make controllers or devices that fit that interface. Many companies, for example, make disk drives that match the IDE or SCSI interface.

The interface between the controller and the device is often a very low-level interface. A disk, for example, might be formatted with 256 sectors of 512 bytes per track. What actually comes off the drive, however, is a serial bit stream, starting with a **preamble**, then the 4096 bits in a sector, and finally a checksum, also called an **Error-Correcting Code (ECC)**. The preamble is written when the disk is formatted and contains the cylinder and sector number, the sector size, and similar data, as well as synchronization information.

The controller's job is to convert the serial bit stream into a block of bytes and perform any error correction necessary. The block of bytes is typically first assembled, bit by bit, in a buffer inside the controller. After its checksum has been verified and the block declared to be error free, it can then be copied to main memory.

The controller for a monitor also works as a bit serial device at an equally low level. It reads bytes containing the characters to be displayed from memory and generates the signals used to modulate the CRT beam to cause it to write on the screen. The controller also generates the signals for making the CRT beam do a horizontal retrace after it has finished a scan line, as well as the signals for making it do a vertical retrace after the entire screen has been scanned. If it were not for the CRT controller, the operating system programmer would have to explicitly program the analog scanning of the tube. With the controller, the operating system initializes the controller with a few parameters, such as the number of characters or pixels per line and number of lines per screen, and lets the controller take care of actually driving the beam.

5.1.3 Memory-Mapped I/O

Each controller has a few registers that are used for communicating with the CPU. By writing into these registers, the operating system can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action. By reading from these registers, the operating system can learn what the device's state is, whether it is prepared to accept a new command, and so on.

In addition to the control registers, many devices have a data buffer that the operating system can read and write. For example, a common way for computers to display pixels on the screen is to have a video RAM, which is basically just a data buffer, available for programs or the operating system to write into.

The issue thus arises of how the CPU communicates with the control registers and the device data buffers. Two alternatives exist. In the first approach, each control register is assigned an **I/O port** number, an 8- or 16-bit integer. Using a special I/O instruction such as

```
IN REG,PORT,
```

the CPU can read in control register PORT and store the result in CPU register REG. Similarly, using

```
OUT PORT,REG
```

the CPU can write the contents of REG to a control register. Most early computers, including nearly all mainframes, such as the IBM 360 and all of its successors, worked this way.

In this scheme, the address spaces for memory and I/O are different, as shown in Fig. 5-2(a). The instructions

```
IN R0,4
```

and

```
MOV R0,4
```

are completely different in this design. The former reads the contents of I/O port 4 and puts it in R0 whereas the latter reads the contents of memory word 4 and puts it in R0. The 4s in these examples thus refer to different and unrelated address spaces.

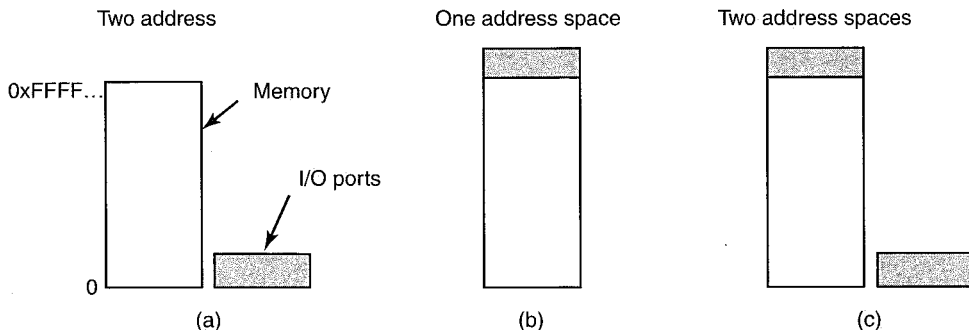


Figure 5-2. (a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid.

The second approach, introduced with the PDP-11, is to map all the control registers into the memory space, as shown in Fig. 5-2(b). Each control register is assigned a unique memory address to which no memory is assigned. This system is called **memory-mapped I/O**. Usually, the assigned addresses are at the top of the address space. A hybrid scheme, with memory-mapped I/O data buffers and

separate I/O ports for the control registers is shown in Fig. 5-2(c). The Pentium uses this architecture, with addresses 640K to 1M being reserved for device data buffers in IBM PC compatibles, in addition to I/O ports 0 through 64K.

How do these schemes work? In all cases, when the CPU wants to read a word, either from memory or from an I/O port, it puts the address it needs on the bus' address lines and then asserts a READ signal on a bus' control line. A second signal line is used to tell whether I/O space or memory space is needed. If it is memory space, the memory responds to the request. If it is I/O space, the I/O device responds to the request. If there is only memory space [as in Fig. 5-2(b)], every memory module and every I/O device compares the address lines to the range of addresses that it services. If the address falls in its range, it responds to the request. Since no address is ever assigned to both memory and an I/O device, there is no ambiguity and no conflict.

The two schemes for addressing the controllers have different strengths and weaknesses. Let us start with the advantages of memory-mapped I/O. First, if special I/O instructions are needed to read and write the device control registers, access to them requires the use of assembly code since there is no way to execute an IN or OUT instruction in C or C++. Calling such a procedure adds overhead to controlling I/O. In contrast, with memory-mapped I/O, device control registers are just variables in memory and can be addressed in C the same way as any other variables. Thus with memory-mapped I/O, a I/O device driver can be written entirely in C. Without memory-mapped I/O, some assembly code is needed.

Second, with memory-mapped I/O, no special protection mechanism is needed to keep user processes from performing I/O. All the operating system has to do is refrain from putting that portion of the address space containing the control registers in any user's virtual address space. Better yet, if each device has its control registers on a different page of the address space, the operating system can give a user control over specific devices but not others by simply including the desired pages in its page table. Such a scheme can allow different device drivers to be placed in different address spaces, not only reducing kernel size but also keeping one driver from interfering with others.

Third, with memory-mapped I/O, every instruction that can reference memory can also reference control registers. For example, if there is an instruction, TEST, that tests a memory word for 0, it can also be used to test a control register for 0, which might be the signal that the device is idle and can accept a new command. The assembly language code might look like this:

```

LOOP:  TEST PORT_4      // check if port 4 is 0
        BEQ READY      // if it is 0, go to ready
        BRANCH LOOP    // otherwise, continue testing
READY:
```

If memory-mapped I/O is not present, the control register must first be read into the CPU, then tested, requiring two instructions instead of one. In the case of the

loop given above, a fourth instruction has to be added, slightly slowing down the responsiveness of detecting an idle device.

In computer design, practically everything involves trade-offs, and that is the case here too. Memory-mapped I/O also has its disadvantages. First, most computers nowadays have some form of caching of memory words. Caching a device control register would be disastrous. Consider the assembly code loop given above in the presence of caching. The first reference to `PORT_4` would cause it to be cached. Subsequent references would just take the value from the cache and not even ask the device. Then when the device finally became ready, the software would have no way of finding it out. Instead, the loop would go on forever.

To prevent this situation with memory-mapped I/O, the hardware has to be equipped with the ability to selectively disable caching, for example, on a per page basis. This feature adds extra complexity to both the hardware and the operating system, which has to manage the selective caching.

Second, if there is only one address space, then all memory modules and all I/O devices must examine all memory references to see which ones to respond to. If the computer has a single bus, as in Fig. 5-3(a), having everyone look at every address is straightforward.

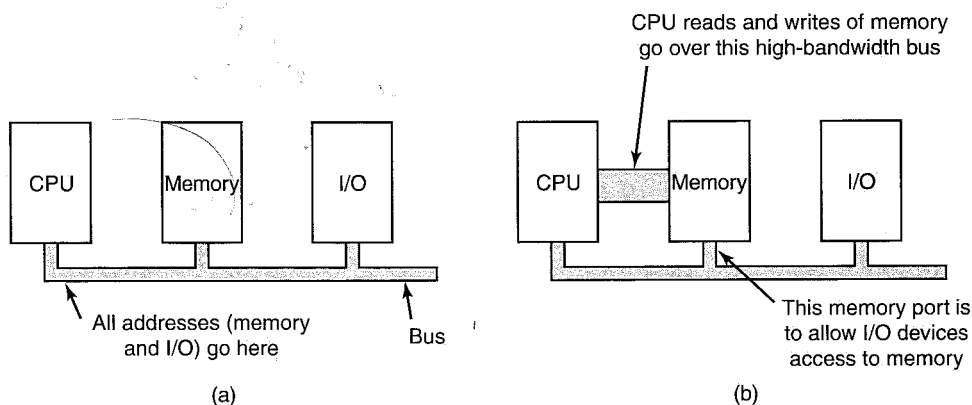


Figure 5-3. (a) A single-bus architecture. (b) A dual-bus memory architecture.

However, the trend in modern personal computers is to have a dedicated high-speed memory bus as shown in Fig. 5-3(b), a property also found in mainframes, incidentally. This bus is tailored for optimize memory performance, with no compromises for the sake of slow I/O devices. Pentium systems even have three external buses (memory, PCI, ISA), as shown in Fig. 1-11.

The trouble with having a separate memory bus on memory-mapped machines is that the I/O devices have no way of seeing memory addresses as they go by on the memory bus, so they have no way of responding. Again, special measures have to be taken to make memory-mapped I/O work on a system with multiple buses. One possibility is to first send all memory references to the memory. If

the memory fails to respond, then the CPU tries the other buses. This design can be made to work but requires additional hardware complexity.

A second possible design is to put a snooping device on the memory bus to pass all addresses presented to potentially interested I/O devices. The problem here is that I/O devices may not be able to process requests at the speed the memory can.

A third possible design, which is the one used on the Pentium configuration of Fig. 1-11, is to filter addresses in the PCI bridge chip. This chip contains range registers that are preloaded at boot time. For example, 640K to 1M could be marked as a nonmemory range. Addresses that fall within one of the ranges marked as nonmemory are forwarded onto the PCI bus instead of to memory. The disadvantage of this scheme is the need for figuring out at boot time which memory addresses are not really memory addresses. Thus each scheme has arguments for and against it, so compromises and trade-offs are inevitable.

5.1.4 Direct Memory Access (DMA)

No matter whether a CPU does or does not have memory-mapped I/O, it needs to address the device controllers to exchange data with them. The CPU can request data from an I/O controller one byte at a time but doing so wastes the CPU's time, so a different scheme, called **DMA (Direct Memory Access)** is often used. The operating system can only use DMA if the hardware has a DMA controller, which most systems do. Sometimes this controller is integrated into disk controllers and other controllers, but such a design requires a separate DMA controller for each device. More commonly, a single DMA controller is available (e.g., on the parentboard) for regulating transfers to multiple devices, often concurrently.

No matter where it is physically located, the DMA controller has access to the system bus independent of the CPU, as shown in Fig. 5-4. It contains several registers that can be written and read by the CPU. These include a memory address register, a byte count register, and one or more control registers. The control registers specify the I/O port to use, the direction of the transfer (reading from the I/O device or writing to the I/O device), the transfer unit (byte at a time or word at a time), and the number of bytes to transfer in one burst.

To explain how DMA works, let us first look at how disk reads occur when DMA is not used. First the controller reads the block (one or more sectors) from the drive serially, bit by bit, until the entire block is in the controller's internal buffer. Next, it computes the checksum to verify that no read errors have occurred. Then the controller causes an interrupt. When the operating system starts running, it can read the disk block from the controller's buffer a byte or a word at a time by executing a loop, with each iteration reading one byte or word from a controller device register and storing it in main memory.

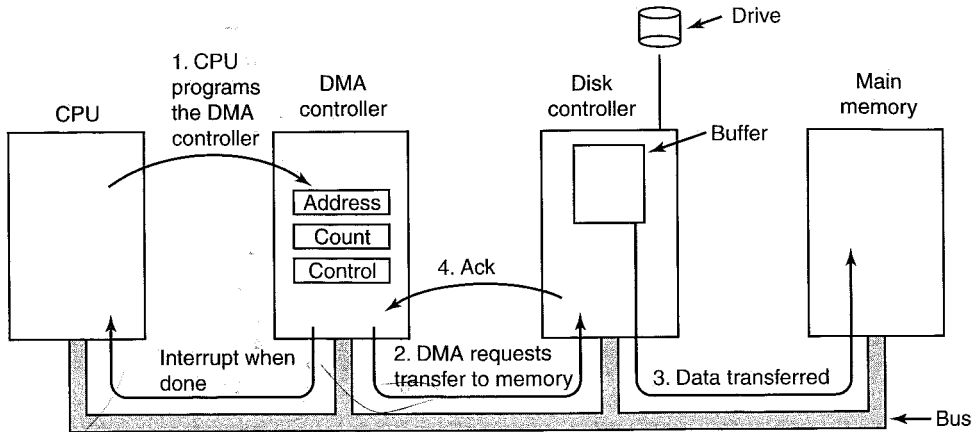


Figure 5-4. Operation of a DMA transfer.

When DMA is used, the procedure is different. First the CPU programs the DMA controller by setting its registers so it knows what to transfer where (step 1 in Fig. 5-4). It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum. When valid data are in the disk controller's buffer, DMA can begin.

The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller (step 2). This read request looks like any other read request, and the disk controller does not know or care whether it came from the CPU or from a DMA controller. Typically, the memory address to write to is on the bus' address lines so when the disk controller fetches the next word from its internal buffer, it knows where to write it. The write to memory is another standard bus cycle (step 3). When the write is complete, the disk controller sends an acknowledgement signal to the disk controller, also over the bus (step 4). The DMA controller then increments the memory address to use and decrements the byte count. If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0. At that time, the DMA controller interrupts the CPU to let it know that the transfer is now complete. When the operating system starts up, it does not have to copy the disk block to memory; it is already there.

DMA controllers vary considerably in their sophistication. The simplest ones handle one transfer at a time, as described above. More complex ones can be programmed to handle multiple transfers at once. Such controllers have multiple sets of registers internally, one for each channel. The CPU starts by loading each set of registers with the relevant parameters for its transfer. Each transfer must use a different device controller. After each word is transferred (steps 2 through 4) in Fig. 5-4, the DMA controller decides which device to service next. It may be set up to use a round-robin algorithm, or it may have a priority scheme design to favor some devices over others. Multiple requests to different device controllers

may be pending at the same time, provided that there is an unambiguous way to tell the acknowledgements apart. Often a different acknowledgement line on the bus is used for each DMA channel for this reason.

Many buses can operate in two modes: word-at-a-time mode and block mode. Some DMA controllers can also operate in either mode. In the former mode, the operation is as described above: the DMA controller requests for the transfer of one word and gets it. If the CPU also wants the bus, it has to wait. The mechanism is called **cycle stealing** because the device controller sneaks in and steals an occasional bus cycle from the CPU once in a while, delaying it slightly. In block mode, the DMA controller tells the device to acquire the bus, issue a series of transfers, then release the bus. This form of operation is called **burst mode**. It is more efficient than cycle stealing because acquiring the bus takes time and multiple words can be transferred for the price of one bus acquisition. The down side to burst mode is that it can block the CPU and other devices for a substantial period of time if a long burst is being transferred.

In the model we have been discussing, sometimes called **fly-by mode**, the DMA controller tells the device controller to transfer the data directly to main memory. An alternative mode that some DMA controllers use is to have the device controller send the word to the DMA controller, which then issues a second bus request to write the word to wherever it is supposed to go. This scheme requires an extra bus cycle per word transferred, but is more flexible in that it can also perform device-to-device copies and even memory-to-memory copies (by first issuing a read to memory and then issuing a write to memory at a different address).

Most DMA controllers use physical memory addresses for their transfers. Using physical addresses requires the operating system to convert the virtual address of the intended memory buffer into a physical address and write this physical address into the DMA controller's address register. An alternative scheme used in a few DMA controllers is to write virtual addresses into the DMA controller instead. Then the DMA controller must use the MMU to have the virtual-to-physical translation done. Only in the case that the MMU is part of the memory (possible, but rare) rather than part of the CPU, can virtual addresses be put on the bus.

We mentioned earlier that the disk first reads data into its internal buffer before DMA can start. You may be wondering why the controller does not just store the bytes in main memory as soon as it gets them from the disk. In other words, why does it need an internal buffer? There are two reasons. First, by doing internal buffering, the disk controller can verify the checksum before starting a transfer. If the checksum is incorrect, an error is signaled and no transfer is done.

The second reason is that once a disk transfer has started, the bits keep arriving from the disk at a constant rate, whether the controller is ready for them or not. If the controller tried to write data directly to memory, it would have to go

over the system bus for each word transferred. If the bus were busy due to some other device using it (e.g., in burst mode), the controller would have to wait. If the next disk word arrived before the previous one had been stored, the controller would have to store it somewhere. If the bus were very busy, the controller might end up storing quite a few words and having a lot of administration to do as well. When the block is buffered internally, the bus is not needed until the DMA begins, so the design of the controller is much simpler because the DMA transfer to memory is not time critical. (Some older controllers did, in fact, go directly to memory with only a small amount of internal buffering, but when the bus was very busy, a transfer might have had to be terminated with an overrun error.)

Not all computers use DMA. The argument against it is that the main CPU is often far faster than the DMA controller and can do the job much faster (when the limiting factor is not the speed of the I/O device). If there is no other work for it to do, having the (fast) CPU wait for the (slow) DMA controller to finish is pointless. Also, getting rid of the DMA controller and having the CPU do all the work in software saves money, important on low-end (embedded) computers.

5.1.5 Interrupts Revisited

We briefly introduced interrupts in Sec. 1.4.3, but there is more to be said. In a typical personal computer system, the interrupt structure is as shown in Fig. 5-5. At the hardware level, interrupts work as follows. When an I/O device has finished the work given to it, it causes an interrupt (assuming that interrupts have been enabled by the operating system). It does this by asserting a signal on a bus line that it has been assigned. This signal is detected by the interrupt controller chip on the parentboard, which then decides what to do.

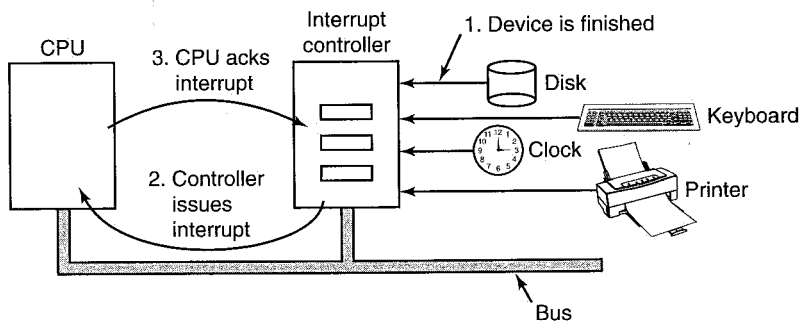


Figure 5-5. How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

If no other interrupts are pending, the interrupt controller processes the interrupt immediately. If another one is in progress, or another device has made a simultaneous request on a higher-priority interrupt request line on the bus, the

device is just ignored for the moment. In this case it continues to assert an interrupt signal on the bus until it is serviced by the CPU.

To handle the interrupt, the controller puts a number on the address lines specifying which device wants attention and asserts a signal that interrupts the CPU.

The interrupt signal causes the CPU to stop what it is doing and start doing something else. The number on the address lines is used as an index into a table called the **interrupt vector** to fetch a new program counter. This program counter points to the start of the corresponding interrupt service procedure. Typically traps and interrupts use the same mechanism from this point on, and frequently share the same interrupt vector. The location of the interrupt vector can be hardwired into the machine or it can be anywhere in memory, with a CPU register (loaded by the operating system) pointing to its origin.

Shortly after it starts running, the interrupt service procedure acknowledges the interrupt by writing a certain value to one of the interrupt controller's I/O ports. This acknowledgement tells the controller that it is free to issue another interrupt. By having the CPU delay this acknowledgement until it is ready to handle the next interrupt, race conditions involving multiple almost simultaneous interrupts can be avoided. As an aside, some (older) computers do not have a centralized interrupt controller chip, so each device controller requests its own interrupts.

The hardware always saves certain information before starting the service procedure. Which information is saved and where it is saved varies greatly from CPU to CPU. As a bare minimum, the program counter must be saved, so the interrupted process can be restarted. At the other extreme, all the visible registers and a large number of internal registers may be saved as well.

One issue is where to save this information. One option is to put it in internal registers that the operating system can read out as needed. A problem with this approach is that then the interrupt controller cannot be acknowledged until all potentially relevant information has been read out, lest a second interrupt overwrite the internal registers saving the state. This strategy leads to long dead times when interrupts are disabled and possibly lost interrupts and lost data.

Consequently, most CPUs save the information on the stack. However, this approach, too, has problems. To start with: whose stack? If the current stack is used, it may well be a user process stack. The stack pointer may not even be legal, which would cause a fatal error when the hardware tried to write some words at it. Also, it might point to the end of a page. After several memory writes, the page boundary might be exceeded and a page fault generated. Having a page fault occur during the hardware interrupt processing creates a bigger problem: where to save the state to handle the page fault?

If the kernel stack is used, there is a much better chance of the stack pointer being legal and pointing to a pinned page. However, switching into kernel mode may require changing MMU contexts and will probably invalidate most or all of

the cache and TLB. Reloading all of these, statically or dynamically will increase the time to process an interrupt and thus waste CPU time.

Another problem is caused by the fact that most modern CPUs are heavily pipelined and often superscalar (internally parallel). In older systems, after each instruction was finished executing, the microprogram or hardware checked to see if there was an interrupt pending. If so, the program counter and PSW were pushed onto the stack and the interrupt sequence begun. After the interrupt handler ran, the reverse process took place, with the old PSW and program counter popped from the stack and the previous process continued.

This model makes the implicit assumption that if an interrupt occurs just after some instruction, all the instructions up to and including that instruction have been executed completely, and no instructions after it have executed at all. On older machines, this assumption was always valid. On modern ones it may not be.

For starters, consider the pipeline model of Fig. 1-6(a). What happens if an interrupt occurs while the pipeline is full (the usual case)? Many instructions are in various stages of execution. When the interrupt occurs, the value of the program counter may not reflect the correct boundary between executed instructions and nonexecuted instructions. More likely, it reflects the address of the next instruction to be fetched and pushed into the pipeline rather than the address of the instruction that just was processed by the execution unit.

As a consequence, there may be a well-defined boundary between instructions that have actually executed and those that have not, but the hardware may not know what it is. Consequently, when the operating system must return from an interrupt, it cannot just start filling the pipeline from the address contained in the program counter. It must figure out what the last executed instruction was, often a complex task that may require analyzing the state of the machine.

Although this situation is bad, interrupts on a superscalar machine, such as that of Fig. 1-6(b) are far worse. Because instructions may execute out of order, there may be no well-defined boundary between the executed and nonexecuted instructions. It may well be that instructions 1, 2, 3, 5, and 8 have executed, but instructions 4, 6, 7, 9, 10, and beyond have not. Furthermore, the program counter may now be pointing to instruction 9, 10, or 11.

An interrupt that leaves the machine in a well-defined state is called a **precise interrupt** (Walker and Cragon, 1995). Such an interrupt has four properties:

1. The PC (Program Counter) is saved in a known place.
2. All instructions before the one pointed to by the PC have fully executed.
3. No instruction beyond the one pointed to by the PC has been executed.
4. The execution state of the instruction pointed to by the PC is known.

Note that there is no prohibition on instructions beyond the one pointed to by the PC from starting. It is just that any changes they make to registers or memory

must be undone before the interrupt happens. It is permitted that the instruction pointed to has been executed. It is also permitted that it has not been executed. However, it must be clear which case applies. Often, if the interrupt is an I/O interrupt, the instruction will not yet have started. However, if the interrupt is really a trap or page fault, then the PC generally points to the instruction that caused the fault so it can be restarted later.

An interrupt that does not meet these requirements is called an **imprecise interrupt** and makes life extremely unpleasant for the operating system writer, who now has to figure out what has happened and what still has to happen. Machines with imprecise interrupts usually vomit a large amount of internal state onto the stack to give the operating system the possibility of figuring out what was going on. Saving a large amount of information to memory on every interrupt makes interrupts slow and recovery even worse. This leads to the ironic situation of having very fast superscalar CPUs sometimes being unsuitable for real-time work due to slow interrupts.

Some computers are designed so that some kinds of interrupts and traps are precise and others are not. For example, having I/O interrupts be precise but traps due to fatal programming errors be imprecise is not so bad since no attempt need be made to restart the running process. Some machines have a bit that can be set to force all interrupts to be precise. The downside of setting this bit is that it forces the CPU to carefully log everything it is doing and maintain shadow copies of registers so it can generate a precise interrupt at any instant. All this overhead has a major impact on performance.

Some superscalar machines, such as the Pentium Pro and all of its successors, have precise interrupts to allow old 386, 486, and Pentium I programs to work correctly (superscalar was introduced in the Pentium Pro; the Pentium I just had two pipelines). The price paid for precise interrupts is extremely complex interrupt logic within the CPU to make sure that when the interrupt controller signals that it wants to cause an interrupt, all instructions up to some point are allowed to finish and none beyond that point are allowed to have any noticeable effect on the machine state. Here the price is paid not in time, but in chip area and in complexity of the design. If precise interrupts were not required for backward compatibility purposes, this chip area would be available for larger on-chip caches, making the CPU faster. On the other hand, imprecise interrupts make the operating system far more complicated and slower, so it is hard to tell which approach is really better.

5.2 PRINCIPLES OF I/O SOFTWARE

Let us now turn away from the I/O hardware and look at the I/O software. First we will look at the goals of the I/O software and then at the different ways I/O can be done from the point of view of the operating system.

5.2.1 Goals of the I/O Software

A key concept in the design of I/O software is known as **device independence**. What it means is that it should be possible to write programs that can access any I/O device without having to specify the device in advance. For example, a program that reads a file as input should be able to read a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device. Similarly, one should be able to type a command such as

```
sort <input >output
```

and have it work with input coming from a floppy disk, an IDE disk, a SCSI disk, or the keyboard, and the output going to any kind of disk or the screen. It is up to the operating system to take care of the problems caused by the fact that these devices really are different and require very different command sequences to read or write.

Closely related to device independence is the goal of **uniform naming**. The name of a file or a device should simply be a string or an integer and not depend on the device in any way. In UNIX, all disks can be integrated in the file system hierarchy in arbitrary ways so the user need not be aware of which name corresponds to which device. For example, a floppy disk can be **mounted** on top of the directory */usr/ast/backup* so that copying a file to */usr/ast/backup/monday* copies the file to the floppy disk. In this way, all files and devices are addressed the same way: by a path name.

Another important issue for I/O software is **error handling**. In general, errors should be handled as close to the hardware as possible. If the controller discovers a read error, it should try to correct the error itself if it can. If it cannot, then the device driver should handle it, perhaps by just trying to read the block again. Many errors are transient, such as read errors caused by specks of dust on the read head, and will go away if the operation is repeated. Only if the lower layers are not able to deal with the problem should the upper layers be told about it. In many cases, error recovery can be done transparently at a low level without the upper levels even knowing about the error.

Still another key issue is **synchronous** (blocking) versus **asynchronous** (interrupt-driven) transfers. Most physical I/O is asynchronous—the CPU starts the transfer and goes off to do something else until the interrupt arrives. User programs are much easier to write if the I/O operations are blocking—after a read system call the program is automatically suspended until the data are available in the buffer. It is up to the operating system to make operations that are actually interrupt-driven look blocking to the user programs.

Another issue for the I/O software is **buffering**. Often data that come off a device cannot be stored directly in its final destination. For example, when a packet comes in off the network, the operating system does not know where to put it until it has stored the packet somewhere and examined it. Also, some devices

have severe real-time constraints (for example, digital audio devices), so the data must be put into an output buffer in advance to decouple the rate at which the buffer is filled from the rate at which it is emptied, in order to avoid buffer under-runs. Buffering involves considerable copying and often has a major impact on I/O performance.

The final concept that we will mention here is sharable versus dedicated devices. Some I/O devices, such as disks, can be used by many users at the same time. No problems are caused by multiple users having open files on the same disk at the same time. Other devices, such as tape drives, have to be dedicated to a single user until that user is finished. Then another user can have the tape drive. Having two or more users writing blocks intermixed at random to the same tape will definitely not work. Introducing dedicated (unshared) devices also introduces a variety of problems, such as deadlocks. Again, the operating system must be able to handle both shared and dedicated devices in a way that avoids problems.

5.2.2 Programmed I/O

There are three fundamentally different ways that I/O can be performed. In this section we will look at the first one (programmed I/O). In the next two sections we will examine the others (interrupt-driven I/O and I/O using DMA). The simplest form of I/O is to have the CPU do all the work. This method is called **programmed I/O**.

It is simplest to illustrate programmed I/O by means of an example. Consider a user process that wants to print the eight-character string "ABCDEFGH" on the printer. It first assembles the string in a buffer in user space, as shown in Fig. 5-6(a).

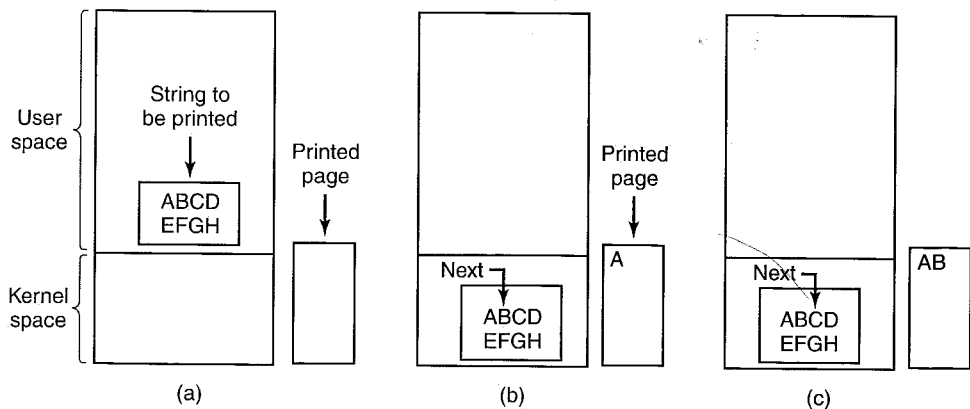


Figure 5-6. Steps in printing a string.

The user process then acquires the printer for writing by making a system call to open it. If the printer is currently in use by another process, this call will fail

and return an error code or will block until the printer is available, depending on the operating system and the parameters of the call. Once it has the printer, the user process makes a system call telling the operating system to print the string on the printer.

The operating system then (usually) copies the buffer with the string to an array, say, *p*, in kernel space, where it is more easily accessed (because the kernel may have to change the memory map to get at user space). It then checks to see if the printer is currently available. If not, it waits until it is available. As soon as the printer is available, the operating system copies the first character to the printer's data register, in this example using memory-mapped I/O. This action activates the printer. The character may not appear yet because some printers buffer a line or a page before printing anything. In Fig. 5-6(b), however, we see that the first character has been printed and that the system has marked the "B" as the next character to be printed.

As soon as it has copied the first character to the printer, the operating system checks to see if the printer is ready to accept another one. Generally, the printer has a second register, which gives its status. The act of writing to the data register causes the status to become not ready. When the printer controller has processed the current character, it indicates its availability by setting some bit in its status register or putting some value in it.

At this point the operating system waits for the printer to become ready again. When that happens, it prints the next character, as shown in Fig. 5-6(c). This loop continues until the entire string has been printed. Then control returns to the user process.

The actions followed by the operating system are summarized in Fig. 5-7. First the data are copied to the kernel. Then the operating system enters a tight loop outputting the characters one at a time. The essential aspect of programmed I/O, clearly illustrated in this figure, is that after outputting a character, the CPU continuously polls the device to see if it is ready to accept another one. This behavior is often called **polling** or **busy waiting**.

```

copy_from_user(buffer, p, count);
for (i = 0; i < count; i++) {
    while (*printer_status_reg != READY);
    *printer_data_register = p[i];
}
return_to_user();
/* p is the kernel bufer */
/* loop on every character */
/* loop until ready */
/* output one character */

```

Figure 5-7. Writing a string to the printer using programmed I/O.

Programmed I/O is simple but has the disadvantage of tying up the CPU full time until all the I/O is done. If the time to "print" a character is very short (because all the printer is doing is copying the new character to an internal buffer), then

busy waiting is fine. Also, in an embedded system, where the CPU has nothing else to do, busy waiting is reasonable. However, in more complex systems, where the CPU has other work to do, busy waiting is inefficient. A better I/O method is needed.

5.2.3 Interrupt-Driven I/O

Now let us consider the case of printing on a printer that does not buffer characters but prints each one as it arrives. If the printer can print, say 100 characters/sec, each character takes 10 msec to print. This means that after every character is written to the printer's data register, the CPU will sit in an idle loop for 10 msec waiting to be allowed to output the next character. This is more than enough time to do a context switch and run some other process for the 10 msec that would otherwise be wasted.

The way to allow the CPU to do something else while waiting for the printer to become ready is to use interrupts. When the system call to print the string is made, the buffer is copied to kernel space, as we showed earlier, and the first character is copied to the printer as soon as it is willing to accept a character. At that point the CPU calls the scheduler and some other process is run. The process that asked for the string to be printed is blocked until the entire string has printed. The work done on the system call is shown in Fig. 5-8(a).

```
copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler();
```

(a)

```
if (count == 0) {
    unblock_user();
} else {
    *printer_data_register = p[i];
    count = count - 1;
    i = i + 1;
}
acknowledge_interrupt();
return_from_interrupt();
```

(b)

Figure 5-8. Writing a string to the printer using interrupt-driven I/O. (a) Code executed when the print system call is made. (b) Interrupt service procedure.

When the printer has printed the character and is prepared to accept the next one, it generates an interrupt. This interrupt stops the current process and saves its state. Then the printer interrupt service procedure is run. A crude version of this code is shown in Fig. 5-8(b). If there are no more characters to print, the interrupt handler takes some action to unblock the user. Otherwise, it outputs the next character, acknowledges the interrupt, and returns to the process that was running just before the interrupt, which continues from where it left off.

5.2.4 I/O Using DMA

An obvious disadvantage of interrupt-driven I/O is that an interrupt occurs on every character. Interrupts take time, so this scheme wastes a certain amount of CPU time. A solution is to use DMA. Here the idea is to let the DMA controller feed the characters to the printer one at a time, without the CPU being bothered. In essence, DMA is programmed I/O, only with the DMA controller doing all the work, instead of the main CPU. An outline of the code is given in Fig. 5-9.

<code>copy_from_user(buffer, p, count);</code>	<code>acknowledge_interrupt();</code>
<code>set_up_DMA_controller();</code>	<code>unblock_user();</code>
<code>scheduler();</code>	<code>return_from_interrupt();</code>
(a)	(b)

Figure 5-9. Printing a string using DMA. (a) Code executed when the print system call is made. (b) Interrupt service procedure.

The big win with DMA is reducing the number of interrupts from one per character to one per buffer printed. If there are many characters and interrupts are slow, this can be a major improvement. On the other hand, the DMA controller is usually much slower than the main CPU. If the DMA controller is not capable of driving the device at full speed, or the CPU usually has nothing to do anyway while waiting for the DMA interrupt, then interrupt-driven I/O or even programmed I/O may be better.

5.3 I/O SOFTWARE LAYERS

I/O software is typically organized in four layers, as shown in Fig. 5-10. Each layer has a well-defined function to perform and a well-defined interface to the adjacent layers. The functionality and interfaces differ from system to system, so the discussion that follows, which examines all the layers starting at the bottom, is not specific to one machine.

5.3.1 Interrupt Handlers

While programmed I/O is occasionally useful, for most I/O, interrupts are an unpleasant fact of life and cannot be avoided. They should be hidden away, deep in the bowels of the operating system, so that as little of the operating system as possible knows about them. The best way to hide them is to have the driver starting an I/O operation block until the I/O has completed and the interrupt occurs. The driver can block itself by doing a down on a semaphore, a wait on a condition variable, a receive on a message, or something similar, for example.

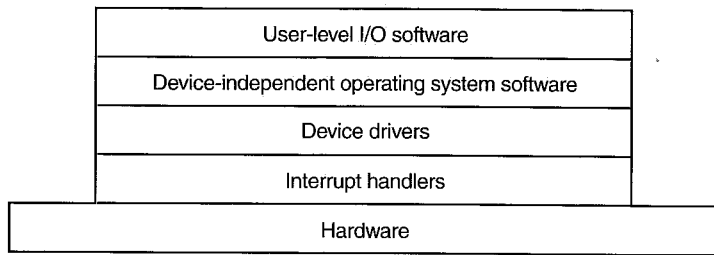


Figure 5-10. Layers of the I/O software system.

When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt. Then it can unblock the driver that started it. In some cases it will just complete up on a semaphore. In others it will do a signal on a condition variable in a monitor. In still others, it will send a message to the blocked driver. In all cases the net effect of the interrupt will be that a driver that was previously blocked will now be able to run. This model works best if drivers are structured as kernel processes, with their own states, stacks, and program counters.

Of course, reality is not quite so simple. Processing an interrupt is not just a matter of taking the interrupt, doing an up on some semaphore, and then executing an IRET instruction to return from the interrupt to the previous process. There is a great deal more work involved for the operating system. We will now give an outline of this work as a series of steps that must be performed in software after the hardware interrupt has completed. It should be noted that the details are very system dependent, so some of the steps listed below may not be needed on a particular machine and steps not listed may be required. Also, the steps that do occur may be in a different order on some machines.

1. Save any registers (including the PSW) that have not already been saved by the interrupt hardware.
2. Set up a context for the interrupt service procedure. Doing this may involve setting up the TLB, MMU and a page table.
3. Set up a stack for the interrupt service procedure.
4. Acknowledge the interrupt controller. If there is no centralized interrupt controller, reenale interrupts.
5. Copy the registers from where they were saved (possibly some stack) to the process table.
6. Run the interrupt service procedure. It will extract information from the interrupting device controller's registers.

7. Choose which process to run next. If the interrupt has caused some high-priority process that was blocked to become ready, it may be chosen to run now.
8. Set up the MMU context for the process to run next. Some TLB set up may also be needed.
9. Load the new process' registers, including its PSW.
10. Start running the new process.

As can be seen, interrupt processing is far from trivial. It also takes a considerable number of CPU instructions, especially on machines in which virtual memory is present and page tables have to be set up or the state of the MMU stored (e.g., the *R* and *M* bits). On some machines the TLB and CPU cache may also have to be managed when switching between user and kernel modes, which takes additional machine cycles.

5.3.2 Device Drivers

Earlier in this chapter we looked at what device controllers do. We saw that each controller has some device registers used to give it commands or some device registers used to read out its status or both. The number of device registers and the nature of the commands vary radically from device to device. For example, a mouse driver has to accept information from the mouse telling how far it has moved and which buttons are currently depressed. In contrast, a disk driver has to know about sectors, tracks, cylinders, heads, arm motion, motor drives, head settling times, and all the other mechanics of making the disk work properly. Obviously, these drivers will be very different.

As a consequence, each I/O device attached to a computer needs some device-specific code for controlling it. This code, called the **device driver**, is generally written by the device's manufacturer and delivered along with the device. Since each operating system needs its own drivers, device manufacturers commonly supply drivers for several popular operating systems.

Each device driver normally handles one device type, or at most, one class of closely related devices. For example, a SCSI disk driver can usually handle multiple SCSI disks of different sizes and different speeds, and perhaps a SCSI CD-ROM as well. On the other hand, a mouse and joystick are so different that different drivers are usually required. However, there is no technical restriction on having one device driver control multiple unrelated devices. It is just not a good idea.

In order to access the device's hardware, meaning the controller's registers, the device driver normally has to be part of the operating system kernel, at least with current architectures. Actually, it would be possible to construct drivers that

ran in user space, with system calls for reading and writing the device registers. In fact, this design would be a good idea, since it would isolate the kernel from the drivers and the drivers from each other. Doing this would eliminate a major source of system crashes—buggy drivers that interfere with the kernel in one way or another. However, since current operating systems expect drivers to run in the kernel, that is the model we will consider here.

Since the designers of every operating system know that pieces of code (drivers) written by outsiders will be installed in it, it needs to have an architecture that allows such installation. This means having a well-defined model of what a driver does and how it interacts with the rest of the operating system. Device drivers are normally positioned below the rest of the operating system, as illustrated in Fig. 5-11.

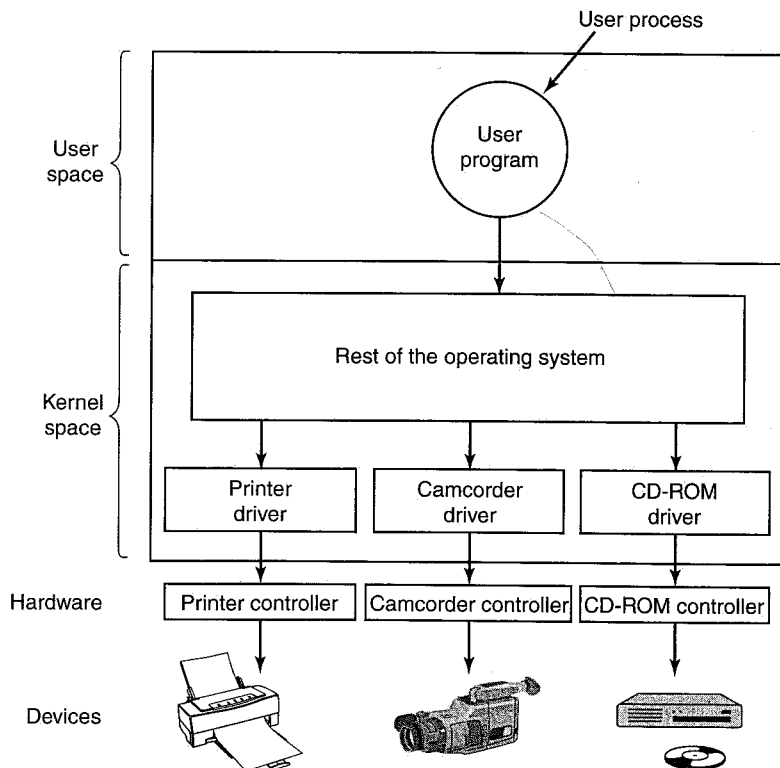


Figure 5-11. Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

Operating systems usually classify drivers into one of a small number of categories. The most common categories are the **block devices**, such as disks, which

contain multiple data blocks that can be addressed independently, and the **character devices**, such as keyboards and printers, which generate or accept a stream of characters.

Most operating systems define a standard interface that all block drivers must support and a second standard interface that all character drivers must support. These interfaces consist of a number of procedures that the rest of the operating system can call to get the driver to do work for it. Typical procedures are those to read a block (block device) or write a character string (character device).

In some systems, the operating system is a single binary program that contains all of the drivers that it will need compiled into it. This scheme was the norm for years with UNIX systems because they were run by computer centers and I/O devices rarely changed. If a new device was added, the system administrator simply recompiled the kernel with the new driver to build a new binary.

With the advent of personal computers, with their myriad of I/O devices, this model no longer worked. Few users are capable of recompiling or relinking the kernel, even if they have the source code or object modules, which is not always the case. Instead, operating systems, starting with MS-DOS, went over to a model in which drivers were dynamically loaded into the system during execution. Different systems handle loading drivers in different ways.

A device driver has several functions. The most obvious one is to accept abstract read and write requests from the device-independent software above it and see that they are carried out. But there are also a few other functions they must perform. For example, the driver must initialize the device, if needed. It may also need to manage its power requirements and log events.

Many device drivers have a similar general structure. A typical driver starts out by checking the input parameters to see if they are valid. If not, an error is returned. If they are valid, a translation from abstract to concrete terms may be needed. For a disk driver, this may mean converting a linear block number into the head, track, sector, and cylinder numbers for the disk's geometry.

Next the driver may check if the device is currently in use. If it is, the request will be queued for later processing. If the device is idle, the hardware status will be examined to see if the request can be handled now. It may be necessary to switch the device on or start a motor before transfers can be begun. Once the device is on and ready to go, the actual control can begin.

Controlling the device means issuing a sequence of commands to it. The driver is the place where the command sequence is determined, depending on what has to be done. After the driver knows which commands it is going to issue, it starts writing them into the controller's device registers. After writing each command to the controller, it may be necessary to check to see if the controller accepted the command and is prepared to accept the next one. This sequence continues until all the commands have been issued. Some controllers can be given a linked list of commands (in memory) and told to read and process them all by itself without further help from the operating system.

After the commands have been issued, one of two situations will apply. In many cases the device driver must wait until the controller does some work for it, so it blocks itself until the interrupt comes in to unblock it. In other cases, however, the operation finishes without delay, so the driver need not block. As an example of the latter situation, scrolling the screen in character mode requires just writing a few bytes into the controller's registers. No mechanical motion is needed, so the entire operation can be completed in nanoseconds.

In the former case, the blocked driver will be awakened by the interrupt. In the latter case, it will never go to sleep. Either way, after the operation has been completed, the driver must check for errors. If everything is all right, the driver may have data to pass to the device-independent software (e.g., a block just read). Finally, it returns some status information for error reporting back to its caller. If any other requests are queued, one of them can now be selected and started. If nothing is queued, the driver blocks waiting for the next request.

This simple model is only a rough approximation to reality. Many factors make the code much more complicated. For one thing, an I/O device may complete while a driver is running, interrupting the driver. The interrupt may cause a device driver to run. In fact, it may cause the current driver to run. For example, while the network driver is processing an incoming packet, another packet may arrive. Consequently, drivers have to be **reentrant**, meaning that a running driver has to expect that it will be called a second time before the first call has completed.

In a hot pluggable system, devices can be added or removed while the computer is running. As a result, while a driver is busy reading from some device, the system may inform it that the user has suddenly removed that device from the system. Not only must the current I/O transfer be aborted without damaging any kernel data structures, but any pending requests for the now-vanished device must also be gracefully removed from the system and their callers given the bad news. Furthermore, the unexpected addition of new devices may cause the kernel to juggle resources (e.g., interrupt request lines), taking old ones away from the driver and giving it new ones in their place.

Drivers are not allowed to make system calls, but they often need to interact with the rest of the kernel. Usually, calls to certain kernel procedures are permitted. For example, there are usually calls to allocate and deallocate hardwired pages of memory for use as buffers. Other useful calls are needed to manage the MMU, timers, the DMA controller, the interrupt controller, and so on.

5.3.3 Device-Independent I/O Software

Although some of the I/O software is device specific, other parts of it are device independent. The exact boundary between the drivers and the device-independent software is system (and device) dependent, because some functions that could be done in a device-independent way may actually be done in the

drivers, for efficiency or other reasons. The functions shown in Fig. 5-12 are typically done in the device-independent software.

Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

Figure 5-12. Functions of the device-independent I/O software.

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software. Below we will look at the above issues in more detail.

Uniform Interfacing for Device Drivers

A major issue in an operating system is how to make all I/O devices and drivers look more-or-less the same. If disks, printers, keyboards, etc., are all interfaced in different ways, every time a new device comes along, the operating system must be modified for the new device. Having to hack on the operating system for each new device is not a good idea.

One aspect of this issue is the interface between the device drivers and the rest of the operating system. In Fig. 5-13(a) we illustrate a situation in which each device driver has a different interface to the operating system. What this means is that the driver functions available for the system to call differ from driver to driver. It might also mean that the kernel functions that the driver needs also differ from driver to driver. Taken together, it means that interfacing each new driver requires a lot of new programming effort.

In contrast, in Fig. 5-13(b), we show a different design in which all drivers have the same interface. Now it becomes much easier to plug in a new driver, providing it conforms to the driver interface. It also means that driver writers know what is expected of them (e.g., what functions they must provide and what kernel functions they may call). In practice, not all devices are absolutely identical, but usually there are only a small number of device types and even these are generally almost the same. For example, even block and character devices have many functions in common.

Another aspect of having a uniform interface is how I/O devices are named. The device-independent software takes care of mapping symbolic device names onto the proper driver. For example, in UNIX a device name, such as `/dev/disk0`, uniquely specifies the i-node for a special file, and this i-node contains the **major device number**, which is used to locate the appropriate driver. The i-node also

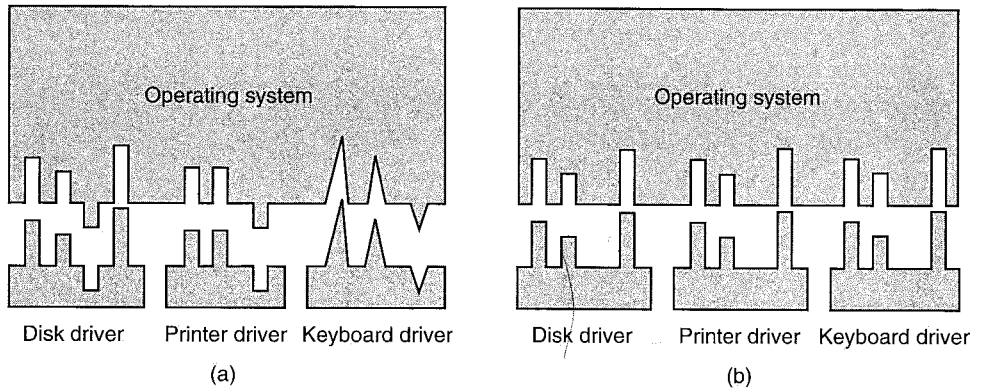


Figure 5-13. (a) Without a standard driver interface. (b) With a standard driver interface.

contains the **minor device number**, which is passed as a parameter to the driver in order to specify the unit to be read or written. All devices have major and minor numbers, and all drivers are accessed by using the major device number to select the driver.

Closely related to naming is protection. How does the system prevent users from accessing devices that they are not entitled to access? In both UNIX and Windows 2000 devices appears in the file system as named objects, which means that the usual protection rules for files also apply to I/O devices. The system administrator can then set the proper permissions for each device.

Buffering

Buffering is also an issue, both for block and character devices for a variety of reasons. To see one of them, consider a process that wants to read data from a modem. One possible strategy for dealing with the incoming characters is to have the user process do a read system call and block waiting for one character. Each arriving character causes an interrupt. The interrupt service procedure hands the character to the user process and unblocks it. After putting the character somewhere, the process reads another character and blocks again. This model is indicated in Fig. 5-14(a).

The trouble with this way of doing business is that the user process has to be started up for every incoming character. Allowing a process to run many times for short runs is inefficient, so this design is not a good one.

An improvement is shown in Fig. 5-14(b). Here the user process provides an n -character buffer in user space and does a read of n characters. The interrupt service procedure puts incoming characters in this buffer until it fills up. Then it

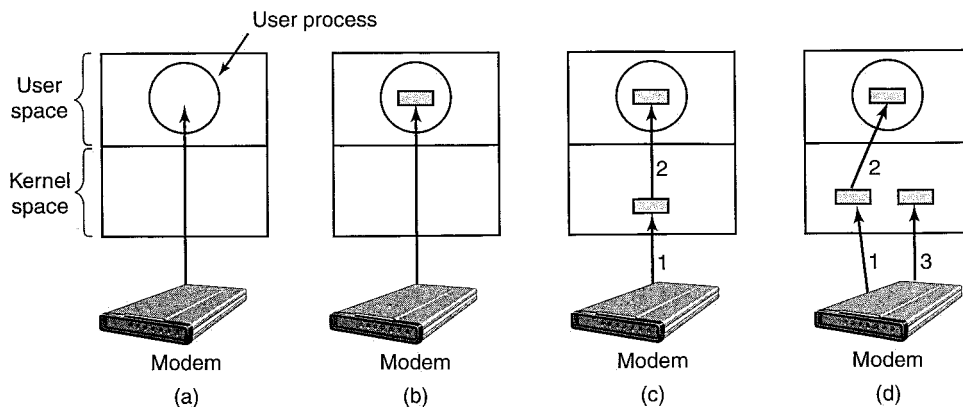


Figure 5-14. (a) Unbuffered input. (b) Buffering in user space. (c) Buffering in the kernel followed by copying to user space. (d) Double buffering in the kernel.

wakes up the user process. This scheme is far more efficient than the previous one, but it, too, has a drawback: what happens if the buffer is paged out when a character arrives? The buffer could be locked in memory, but if many processes start locking pages in memory, the pool of available pages will shrink and performance will degrade.

Yet another approach is to create a buffer inside the kernel and have the interrupt handler put the characters there, as shown in Fig. 5-14(c). When this buffer is full, the page with the user buffer is brought in, if needed, and the buffer copied there in one operation. This scheme is far more efficient.

However, even this scheme suffers from a problem: What happens to characters that arrive while the page with the user buffer is being brought in from the disk? Since the buffer is full, there is no place to put them. A way out is to have a second kernel buffer. After the first buffer fills up, but before it has been emptied, the second one is used, as shown in Fig. 5-14(d). When the second buffer fills up, it is available to be copied to the user (assuming the user has asked for it). While the second buffer is being copied to user space, the first one can be used for new characters. In this way, the two buffers take turns: while one is being copied to user space, the other is accumulating new input. A buffering scheme like this is called **double buffering**.

Buffering is also important on output. Consider, for example, how output is done to the modem without buffering using the model of Fig. 5-14(b). The user process executes a write system call to output n characters. The system has two choices at this point. It can block the user until all the characters have been written, but this could take a very long time over a slow telephone line. It could also release the user immediately and do the I/O while the user computes some more, but this leads to an even worse problem: how does the user process know that the output has been completed and it can reuse the buffer? The system could generate

a signal or software interrupt, but that style of programming is difficult and prone to race conditions. A much better solution is for the kernel to copy the data to a kernel buffer, analogous in Fig. 5-14(c) (but the other way), and unblock the caller immediately. Now it does not matter when the actual I/O has been completed. The user is free to reuse the buffer the instant it is unblocked.

Buffering is a widely-used technique, but it has a downside as well. If data get buffered too many times, performance suffers. Consider, for example, the network of Fig. 5-15. Here a user does a system call to write to the network. The kernel copies the packet to a kernel buffer to allow the user to proceed immediately (step 1).

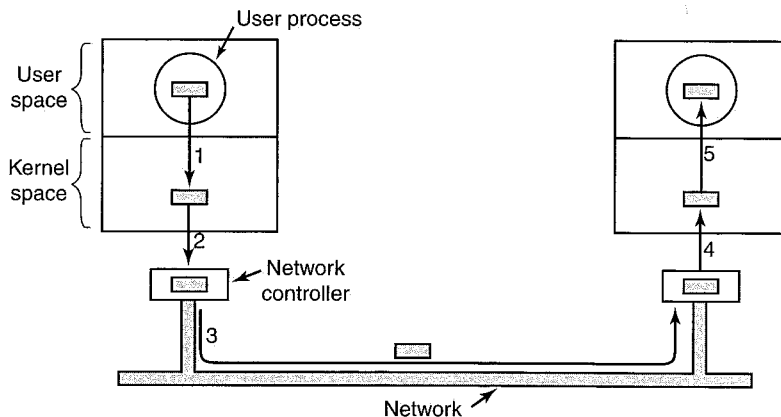


Figure 5-15. Networking may involve many copies of a packet.

When the driver is called, it copies the packet to the controller for output (step 2). The reason it does not output to the wire directly from kernel memory is that once a packet transmission has been started, it must continue at a uniform speed. The driver cannot guarantee that it can get to memory at a uniform speed because DMA channels and other I/O devices may be stealing many cycles. Failing to get a word on time would ruin the packet. By buffering the packet inside the controller, this problem is avoided.

After the packet has been copied to the controller's internal buffer, it is copied out onto the network (step 3). Bits arrive at the receiver shortly after being sent, so just after the last bit has been sent, that bit arrives at the receiver, where the packet has been buffered in the controller. Next the packet is copied to the receiver's kernel buffer (step 4). Finally, it is copied to the receiving process' buffer (step 5). Usually, the receiver then sends back an acknowledgement. When the sender gets the acknowledgement, it is free to send the next packet. However, it should be clear that all this copying is going to slow down the transmission rate considerably because all the steps must happen sequentially.

Error Reporting

Errors are far more common in the context of I/O than in other contexts. When they occur, the operating system must handle them as best it can. Many errors are device-specific and must be handled by the appropriate driver, but the framework for error handling is device independent.

One class of I/O errors are programming errors. These occur when a process asks for something impossible, such as writing to an input device (keyboard, mouse, scanner, etc.) or reading from an output device (printer, plotter, etc.). Other errors include providing an invalid buffer address or other parameter, and specifying an invalid device (e.g., disk 3 when the system has only two disks). The action to take on these errors is straightforward: just report back an error code to the caller.

Another class of errors is the class of actual I/O errors, for example, trying to write a disk block that has been damaged or trying to read from a camcorder that has been switched off. In these circumstances, it is up to the driver to determine what to do. If the driver does not know what to do, it may pass the problem back up to device-independent software.

What this software does depends on the environment and the nature of the error. If it is a simple read error and there is an interactive user available, it may display a dialog box asking the user what to do. The options may include retrying a certain number of times, ignoring the error, or killing the calling process. If there is no user available, probably the only real option is to have the system call fail with an error code.

However, some errors cannot be handled this way. For example, a critical data structure, such as the root directory or free block list, may have been destroyed. In this case, the system may have to display an error message and terminate.

Allocating and Releasing Dedicated Devices

Some devices, such as CD-ROM recorders, can be used only by a single process at any given moment. It is up to the operating system to examine requests for device usage and accept or reject them, depending on whether the requested device is available or not. A simple way to handle these requests is to require processes to perform opens on the special files for devices directly. If the device is unavailable, the open fails. Closing such a dedicated device then releases it.

An alternative approach is to have special mechanisms for requesting and releasing dedicated devices. An attempt to acquire a device that is not available blocks the caller instead of failing. Blocked processes are put on a queue. Sooner or later, the requested device becomes available and the first process on the queue is allowed to acquire it and continue execution.

Device-Independent Block Size

Different disks may have different sector sizes. It is up to the device-independent software to hide this fact and provide a uniform block size to higher layers, for example, by treating several sectors as a single logical block. In this way, the higher layers only deal with abstract devices that all use the same logical block size, independent of the physical sector size. Similarly, some character devices deliver their data one byte at a time (e.g., modems), while others deliver theirs in larger units (e.g., network interfaces). These differences may also be hidden.

5.3.4 User-Space I/O Software

Although most of the I/O software is within the operating system, a small portion of it consists of libraries linked together with user programs, and even whole programs running outside the kernel. System calls, including the I/O system calls, are normally made by library procedures. When a C program contains the call

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

the library procedure *write* will be linked with the program and contained in the binary program present in memory at run time. The collection of all these library procedures is clearly part of the I/O system.

While these procedures do little more than put their parameters in the appropriate place for the system call, there are other I/O procedures that actually do real work. In particular, formatting of input and output is done by library procedures. One example from C is *printf*, which takes a format string and possibly some variables as input, builds an ASCII string, and then calls *write* to output the string. As an example of *printf*, consider the statement

```
printf("The square of %3d is %6d\n", i, i*i);
```

It formats a string consisting of the 14-character string "The square of " followed by the value *i* as a 3-character string, then the 4-character string " is ", then *i*² as six characters, and finally a line feed.

An example of a similar procedure for input is *scanf* which reads input and stores it into variables described in a format string using the same syntax as *printf*. The standard I/O library contains a number of procedures that involve I/O and all run as part of user programs.

Not all user-level I/O software consists of library procedures. Another important category is the spooling system. **Spooling** is a way of dealing with dedicated I/O devices in a multiprogramming system. Consider a typical spooled device: a printer. Although it would be technically easy to let any user process open the character special file for the printer, suppose a process opened it and then did nothing for hours. No other process could print anything.

Instead what is done is to create a special process, called a **daemon**, and a special directory, called a **spooling directory**. To print a file, a process first generates the entire file to be printed and puts it in the spooling directory. It is up to the daemon, which is the only process having permission to use the printer's special file, to print the files in the directory. By protecting the special file against direct use by users, the problem of having someone keeping it open unnecessarily long is eliminated.

Spooling is not only used for printers. It is also used in other situations. For example, file transfer over a network often uses a network daemon. To send a file somewhere, a user puts it in a network spooling directory. Later on, the network daemon takes it out and transmits it. One particular use of spooled file transmission is the USENET News system. This network consists of millions of machines around the world communicating using the Internet. Thousands of news groups exist on many topics. To post a news message, the user invokes a news program, which accepts the message to be posted and then deposits it in a spooling directory for transmission to other machines later. The entire news system runs outside the operating system.

Figure 5-16 summarizes the I/O system, showing all the layers and the principal functions of each layer. Starting at the bottom, the layers are the hardware, interrupt handlers, device drivers, device-independent software, and finally the user processes.

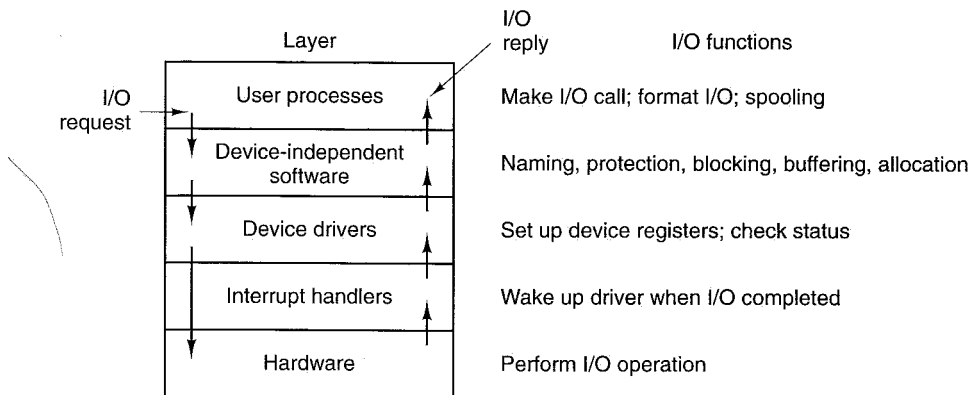


Figure 5-16. Layers of the I/O system and the main functions of each layer.

The arrows in Fig. 5-16 show the flow of control. When a user program tries to read a block from a file, for example, the operating system is invoked to carry out the call. The device-independent software looks for it in the buffer cache, for example. If the needed block is not there, it calls the device driver to issue the request to the hardware to go get it from the disk. The process is then blocked until the disk operation has been completed.

When the disk is finished, the hardware generates an interrupt. The interrupt handler is run to discover what has happened, that is, which device wants attention right now. It then extracts the status from the device and wakes up the sleeping process to finish off the I/O request and let the user process continue.

5.4 DISKS

Now we will begin studying some real I/O devices. We will begin with disks. After that we will examine clocks, keyboards, and displays.

5.4.1 Disk Hardware

Disks come in a variety of types. The most common ones are the magnetic disks (hard disks and floppy disks). They are characterized by the fact that reads and writes are equally fast, which makes them ideal as secondary memory (paging, file systems, etc.). Arrays of these disks are sometimes used to provide highly-reliable storage. For distribution of programs, data, and movies, various kinds of optical disks (CD-ROMs, CD-Recordables, and DVDs) are also important. In the following sections we will first describe the hardware and then the software for these devices.

Magnetic Disks

Magnetic disks are organized into cylinders, each one containing as many tracks as there are heads stacked vertically. The tracks are divided into sectors, with the number of sectors around the circumference typically being 8 to 32 on floppy disks, and up to several hundred on hard disks. The number of heads varies from 1 to about 16.

Some magnetic disks have little electronics and just deliver a simple serial bit stream. On these disks, the controller does most of the work. On other disks, in particular, **IDE (Integrated Drive Electronics)** disks, the drive itself contains a microcontroller that does some work and allows the real controller to issue a set of higher-level commands.

A device feature that has important implications for the disk driver is the possibility of a controller doing seeks on two or more drives at the same time. These are known as **overlapped seeks**. While the controller and software are waiting for a seek to complete on one drive, the controller can initiate a seek on another drive. Many controllers can also read or write on one drive while seeking on one or more other drives, but a floppy disk controller cannot read or write on two drives at the same time. (Reading or writing requires the controller to move bits on a microsecond time scale, so one transfer uses up most of its computing power.) The situation is different for hard disks with integrated controllers, and in

of points around its perimeter. All the points are relative to the (0, 0) origin. Using this system, it is easy to scale the characters up or down. All that has to be done is to multiply each coordinate by the same scale factor. In this way, a True-Type character can be scaled up or down to any point size, even fractional point sizes. Once at the proper size, the points can be connected using the well-known follow-the-dots algorithm taught in kindergarten (note that modern kindergartens use splines for smoother results). After the outline has been completed, the character can be filled in. An example of some characters scaled to three different point sizes is given in Fig. 5-44.



Figure 5-44. Some examples of character outlines at different point sizes.

Once the filled character is available in mathematical form, it can be rasterized, that is, converted to a bitmap at whatever resolution is desired. By first scaling and then rasterizing, we can be sure that the characters displayed on the screen and those that appear on the printer will be as close as possible, differing only in quantization error. To improve the quality still more, it is possible to embed hints in each character telling how to do the rasterization. For example, both serifs on the top of the letter T should be identical, something that might not otherwise be the case due to roundoff error.

5.8 NETWORK TERMINALS

Network terminals are used to connect a remote user to computer over a network, either a local area network or a wide area network. There are two different philosophies of how network terminals should work. In one view, the terminal

should have a large amount of computing power and memory in order to run complex protocols to compress the amount of data sent over the network. (A **protocol** is a set of requests and responses that a sender and receiver agree upon in order to communicate over a network or other interface.) In the other view, the terminal should be extremely simple, basically displaying pixels and not doing much thinking in order to make it very cheap. In the following two sections we will discuss an example of each philosophy. First we will examine the sophisticated X Window System. Then we will look at the minimal SLIM terminal.

5.8.1 The X Window System

The ultimate in intelligent terminals is a terminal that contains a CPU as powerful as the main computer, along with megabytes of memory, a keyboard, and a mouse. One terminal of this type is the **X terminal**, which runs the X Window System (often just called X), developed at M.I.T. as part of project Athena. An X terminal is a computer that runs the X software and which interacts with programs running on a remote computer.

The program inside the X terminal that collects input from the keyboard or mouse and accepts commands from a remote computer is called the **X server**. It has to keep track of which window is currently selected (where the mouse pointer is), so it knows which client to send any new keyboard input to. It communicates over the network with **X clients** running on some remote host. It sends them keyboard and mouse input and accepts display commands from them.

It may seem strange to have the X server inside the terminal and the clients on the remote host, but the X server's job is to display bits, so it makes sense to be near the user. From the program's point of view, it is a client telling the server to do things, like display text and geometric figures. The server (in the terminal) just does what it is told, as do all servers. The arrangement of client and server is shown in Fig. 5-45.

It is also possible to run the X Window System on top of UNIX or another operating system. In fact, many UNIX systems run X as their standard windowing system, even on standalone machines or for accessing remote computers over the Internet. What the X Window System really defines is the protocol between the X client and the X server, as shown in Fig. 5-45. It does not matter whether the client and server are on the same machine, separated by 100 meters over a local area network, or are thousands of kilometers apart and connected by the Internet. The protocol and operation of the system is identical in all cases.

X is just a windowing system. It is not a complete GUI. To get a complete GUI, others layer of software are run on top of it. One layer is **Xlib**, which is a set of library procedures for accessing the X functionality. These procedures form the basis of the X Window System and are what we will examine below, but they are too primitive for most user programs to access directly. For example, each mouse click is reported separately, so that determining that two clicks really form a double click has to be handled above Xlib.

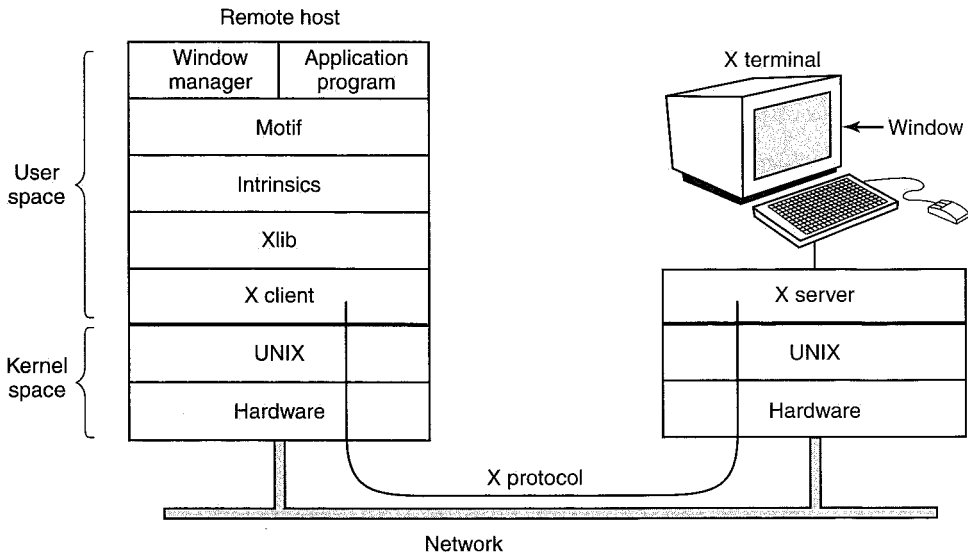


Figure 5-45. Clients and servers in the M.I.T. X Window System.

To make programming with X easier, a toolkit consisting of the **Intrinsics** is supplied as part of X. This layer manages buttons, scroll bars, and other GUI elements, called **widgets**. To make a true GUI interface, with a uniform look and feel, yet another layer is needed. The most popular one is called **Motif**. Most applications make use of calls to Motif rather than Xlib.

Also worth noting is that window management is not part of X itself. The decision to leave it out was fully intentional. Instead, a separate X client process, called a **window manager**, controls the creation, deletion, and movement of windows on the screen. To manage windows, it sends commands to the X server telling what to do. It often runs on the same machine as the X client, but in theory can run anywhere.

This modular design, consisting of several layers and multiple programs, makes X highly portable and flexible. It has been ported to most versions of UNIX, including Solaris, BSD, AIX, Linux, and so on, making it possible for application developers to have a standard user interface for multiple platforms. It has also been ported to other operating systems. In contrast, in Windows, the windowing and GUI systems are mixed together in the GDI and located in the kernel, which makes them harder to maintain. For example, the Windows 98 GUI is still fundamentally 16 bits, more than a decade after the Intel CPUs were 32 bits.

Now let us take a brief look at X as viewed from the Xlib level. When an X program starts, it opens a connection to one or more X servers— let us call them workstations even though they might be colocated on the same machine as the X

program itself. X considers this connection to be reliable in the sense that lost and duplicate messages are handled by the networking software and it does not have to worry about communication errors. Usually, TCP/IP is used between the client and server.

Four kinds of messages go over the connection:

1. Drawing commands from the program to the workstation.
2. Replies by the workstation to program queries.
3. Keyboard, mouse, and other event announcements.
4. Error messages.

Most drawing commands are sent from the program to the workstation as one-way messages. No reply is expected. The reason for this design is that when the client and server processes are on different machines, it may take a substantial period of time for the command to reach the server and be carried out. Blocking the application program during this time would slow it down unnecessarily. On the other hand, when the program needs information from the workstation, it simply has to wait until the reply comes back.

Like Windows, X is highly event driven. Events flow from the workstation to the program, usually in response to some human action such as keyboard strokes, mouse movements, or a window being uncovered. Each event message is 32 bytes, with the first byte giving the event type and the next 31 bytes providing additional information. Several dozen kinds of events exist, but a program is sent only those events that it has said it is willing to handle. For example, if a program does not want to hear about key releases, it is not sent any key release events. As in Windows, events are queued, and programs read events from the queue. However, unlike Windows, the operating system never calls procedures within the application program on its own. It does not even know which procedure handles which event.

A key concept in X is the **resource**. A resource is a data structure that holds certain information. Application programs create resources on workstations. Resources can be shared among multiple processes on the workstation. Resources tend to be short lived and do not survive workstation reboots. Typical resources include windows, fonts, colormaps (color palettes), pixmaps (bitmaps), cursors, and graphic contexts. The latter are used to associate properties with windows and are similar in concept to device contexts in Windows.

A rough, incomplete skeleton of an X program is shown in Fig. 5-46. It begins by including some required headers and then declaring some variables. It then connects to the X server specified as the parameter to *XOpenDisplay*. Then it allocates a window resource and stores a handle to it in *win*. In practice, some initialization would happen here. After that it tells the window manager that the new window exists so the window manager can manage it.

```

#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
    Display disp;                /* server identifier */
    Window win;                  /* window identifier */
    GC gc;                       /* graphic context identifier */
    XEvent event;                /* storage for one event */
    int running = 1;

    disp = XOpenDisplay("display_name"); /* connect to the X server */
    win = XCreateSimpleWindow(disp, ... ); /* allocate memory for new window */
    XSetStandardProperties(disp, ...);    /* announces window to window mgr */
    gc = XCreateGC(disp, win, 0, 0);      /* create graphic context */
    XSelectInput(disp, win, ButtonPressMask | KeyPressMask | ExposureMask);
    XMapRaised(disp, win);              /* display window; send Expose event */

    while (running) {
        XNextEvent(disp, &event); /* get next event */
        switch (event.type) {
            case Expose:      ...; break; /* repaint window */
            case ButtonPress: ...; break; /* process mouse click */
            case Keypress:    ...; break; /* process keyboard input */
        }
    }

    XFreeGC(disp, gc);            /* release graphic context */
    XDestroyWindow(disp, win);    /* deallocate window's memory space */
    XCloseDisplay(disp);          /* tear down network connection */
}

```

Figure 5-46. A skeleton of an X Window application program.

The call to *XCreateGC* creates a graphic context in which properties of the window are stored. In a more complete program, they might be initialized here. The next statement, the call to *XSelectInput*, tells the X server which events the program is prepared to handle. In this case it is interested in mouse clicks, keystrokes, and windows being uncovered. In practice, a real program would be interested in other events as well. Finally, the call to *XMapRaised* maps the new window onto the screen as the uppermost window. At this point the window becomes visible on the screen.

The main loop consists of two statements and is logically much simpler than the corresponding loop in Windows. The first statement here gets an event and the second one dispatches on the event type for processing. When some event

indicates that the program has finished, *running* is set to 0 and the loop terminates. Before exiting, the program releases the graphic context, window, and connection.

It is worth mentioning that not everyone likes a GUI. Many programmers prefer a traditional command-line oriented interface of the type discussed in Sec. 5.6.2 above. X handles this via a client program called *xterm*. This program emulates an old VT102 intelligent terminal, complete with all the escape sequences. Thus editors such as *vi* and *emacs* and other software that uses termcap work in these windows without modification.

5.8.2 The SLIM Network Terminal

Over the years, the main computing paradigm has oscillated between centralized and decentralized computing. The first computers, such as the ENIAC, were, in fact, personal computers, albeit large ones, because only one person could use one at once. Then came timesharing systems, in which many remote users at simple terminals shared a big central computer. Next came the PC era, in which the users had their own personal computers again.

While the decentralized PC model has advantages, it also has some severe disadvantages that are only beginning to be taken seriously. Probably the biggest problem is that each PC has a large hard disk and complex software that must be maintained. For example, when a new release of the operating system comes out, a great deal of work has to be done to perform the upgrade on each machine separately. At most corporations, the labor costs of doing this kind of software maintenance dwarf the actual hardware and software costs. For home users, the labor is technically free, but few people are capable of doing it correctly and fewer still enjoy doing it. With a centralized system, only one or a few machines have to be updated and those machines have a staff of experts to do the work.

A related issue is that users should make regular backups of their gigabyte file systems, but few of them do. When disaster strikes, a great deal of moaning and wringing of hands tends to follow. With a centralized system, backups can be made every night by automated tape robots.

Another advantage is that resource sharing is easier with centralized systems. A system with 64 remote users, each with 64 MB of RAM will have most of that RAM idle most of the time. With a centralized system with 4 GB of RAM, it never happens that some user temporarily needs a lot of RAM but cannot get it because it is on someone else's PC. The same argument holds for disk space and other resources.

It is probably a fair conclusion to say that most users want high-performance interactive computing, but do not really want to administer a computer. This has led researchers to reexamine timesharing using dumb terminals (now politely called **thin clients**) that meet modern terminal expectations. X was a step in this direction, but an X server is still a complex system with megabytes of software