Operating Systems:

Design and Implementation

Second Edition

Andrew S. Tanenbaum
Albert S. Woodhull

Contents

PREFACE

Most books on operating systems are strong on theory and weak on practice. This one aims to provide a better balance between the two. It covers all the fundamental principles in great detail, including processes, interprocess communication, semaphores, monitors, message passing, scheduling algorithms, input/output, deadlocks, device drivers, memory management, paging algorithms, file system design, security, and protection mechanisms. But it also discusses one particular system—MINIX, a UNIX-compatible operating system—in detail, and even provides a complete source code listing for study. This arrangement allows the reader not only to learn the principles, but also to see how they are applied in a real operating system.

When the first edition of this book appeared in 1987, it caused something of a small revolution in the way operating systems courses were taught. Until then, most courses just covered theory. With the appearance of MINIX , many schools began to have laboratory courses in which students examined a real operating system to see how it worked inside. We consider this trend highly desirable and hope this second edition strengthens it.

It its first 10 years, MINIX has undergone many changes. The original code was designed for a 256K 8088-based IBM PC with two diskette drives and no hard disk. It was also based on Version 7 of UNIX . As time went on, MINIX evolved in many ways. For example, the current version will now run on anything from the original PC (in 16-bit real mode) to large Pentiums with massive hard disks (in 32-bit protected mode). It also changed from being based on Version 7, to being based on the international POSIX standard (IEEE 1003.1 and ISO 9945-1). Finally, many features were added, perhaps too many in our view, but too few in the view of some other people, which led to the creation of LINUX. In addition, MINIX was ported to many other platforms, including the Macintosh, Amiga, Atari, and SPARC. This book covers only MINIX 2.0, which so far runs only on computers with an 80x86 CPU, on systems which can emulate such a CPU, or on the SPARC.

This second edition of the book has many changes throughout. Nearly all of the material on principles has been revised, and considerable new material has been added. However, the main change is the discussion of the new, POSIX -based MINIX , and the inclusion of the new code in this book. Also new is the inclusion of a CD-ROM in each book containing the full MINIX source code plus instructions for installing MINIX on a PC (see the file *README.TXT* in the main CD-ROM directory).

Setting up MINIX on an 80x86 PC, whether for individual use or for a laboratory is straightforward. A disk partition of at least 30 MB must be made for it, then it can be installed by just following the instructions in the *README.TXT* file on the CD-ROM. To print the *README.TXT* file on a PC, first start MS-DOS, if it is not already running (from WINDOWS, click on the MS-DOS) icon. Then type

```
copy readme.txt prn
```

to make the printout. The file can also be examined in *edit*, *wordpad*, *notepad*, or any other text editor that can handle flat ASCII text.

For schools (or individuals) that do not have PCs available, two other options are now available. Two simulators are included on the CD-ROM. One, written by Paul Ashton, runs on SPARCs. It runs MINIX as a user program on top of Solaris. As a consequence, MINIX is compiled into a SPARC binary and runs at full speed. In this mode, MINIX is no longer an operating system, but a user program, so some changes to the low-level code were necessary.

The other simulator was written by Kevin P. Lawton of Bochs Software Company. This simulator interprets the Intel 80386 instruction set and enough I/O gear that MINIX can run on the simulator. Of course running on top of an interpreter costs some performance, but it makes debugging much easier for students. This simulator has the advantage that it will run on any computer that supports the M.I.T. X Window System. For more information about both simulators, please see the CD-ROM.

The development of MINIX is an ongoing proposition. The contents of this book and its CD-ROM are merely a snapshot of the system as of the time of publication. For the current state of affairs, please see the MINIX home page on the World Wide Wide, http://www.cs.vu.nl/~ast/minix.html. In addition, MINIX has its own USENET newsgroup: *comp.os.minix*, to which readers can subscribe to find out what is going on in the MINIX world. For those with e-mail, but without newsgroup access, there is also a mailing list. Write to listserv@listserv.nodak.edu with "subscribe minix-1 <your full name>" as the first and only line in the body of the message. You will receive more information by return e-mail.

For classroom use, a problem solutions manual is available, to instructors only, from Prentice Hall. PostScript files containing all the figures in the book, suitable for making overhead sheets, can be found by following the link marked "Software and supplementary material" from http://www.cs.vu.nl/~ast/.

We have been extremely fortunate in having the help of many people during the course of this project. First and foremost, we would like to thank Kees Bot for doing the lion's share of the work in making MINIX conform to the POSIX standard and for managing the distribution. Without his enormous help, we would never have made it. He wrote large chunks of code himself (e.g. the POSIX terminal I/O), cleaned up other sections, and repaired numerous bugs that had crept in over the years. Thank you for a job well done.

Bruce Evans, Philip Homburg, Will Rose, and Michael Temari have all contributed to the development of MINIX over the years. Hundreds of other people have contributed to MINIX via the newsgroup. There were so many of them and their contributions have been so varied that we cannot even begin to list them all here, so the best we can do is a generic thank you to all of them.

Several people read parts of the manuscript and made suggestions. We would like to give our special thanks to John Casey, Dale Grit, and Frans Kaashoek.

A number of students at the Vrije Universiteit tested the beta version of the CD-ROM. These were: Ahmed Batou, Goran Dokic, Peter Gijzel, Thomer Gil, Dennis Grimbergen, Roderick Groesbeek, Wouter Haring, Guido Kollerie, Mark Lassche, Raymond Ris, Frans ter Borg, Alex van Ballegooy, Ries van der Velden, Alexander Wels, and Thomas Zeeman. We would like to thank all of them for their careful work and detailed reports.

ASW would also like to thank several of his former students, particularly Peter W. Young of Hampshire College and Maria Isabel Sanchez and William Puddy Vargas of the Universidad Nacional Autonoma de Nicaragua for the part their interest in MINIX

played in sustaining his efforts.

Finally, we would like to thank our families. Suzanne has been through this ten times now. Barbara has been through it nine times now. Marvin has been through it eight times now. Even Little Bram has been through it four times. It's kind of getting to be routine, but the love and support is still much appreciated. (ast)

As for Al's Barbara, this is the first time she has been through this. It would not have been possible without her support, patience, and good humor. It has been Gordon's good fortune to have been away at college through most of this. But it is a delight to have a son who understands and cares about the same things that fascinate me. (asw)

Andrew S. Tanenbaum
Albert S. Woodhull

Chapter 1

INTRODUCTION $\surd$

Without its software, a computer is basically a useless lump of metal. With its software, a computer can store, process, and retrieve information; display multimedia documents; search the Internet; and engage in many other valuable activities to earn its keep. Computer software can be devided roughly into two kinds: system programs, which manage the operation of the computer itself, and application programs, which perform the actual work the user wants. The most fundamental system program is the **operating system**, which controls all the computer's resources and provides the base upon which the application programs can be written.

A modern computer system consists of one of more processors, some main memory (often known as RAM—Random Access Memory), disks, printers, 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. If every programmer had to be concerned with how disk drives work, and with all the dozens of things that could go wrong when reading a disk block, it is unlikely that many programs could written at all.

Many years ago it became abundantly clear that some way had to be found to shield programmers from the complexity of the hardware. The way that has evolved gradually is to put a layer of software on top of the bare hardware, to manage all parts of the system, and present the user with an interface or **virtual machine** that is easier to understand and program. This layer of software is the operating system and forms the subject of this book.

The situation is shown in Fig. 1-1. At the bottom is the hardware, which, in many cases, is itself composed of two or more layers. The lowest layer contains physical devices, consisting of integrated chips, wires, power supplies, cathode ray tubes, and similar physical devices. How these are constructed and how they work is the province of the electrical engineer.

Next (on some machines) comes layer of primitive software that directly controls these devices and provides a cleaner interface to the next layer. This software, called the **microprogram**, is usually located in read-only memory. It is actually an interpreter, fetching the machine language instructions such as ADD, MOVE, and JUMP, and carrying them out as a series of little steps. To carry out an OUT instruction, for example, the microprogram must determine where the numbers to be added to be located, fetch them, add them, and store the result somewhere.The set of instructions that the microprogram interprets defines the **machine language**, which is not really part of the hard machine at all, but computer manufacturers always describe it in their manuals as such, so many people think of it as being the real "machine."

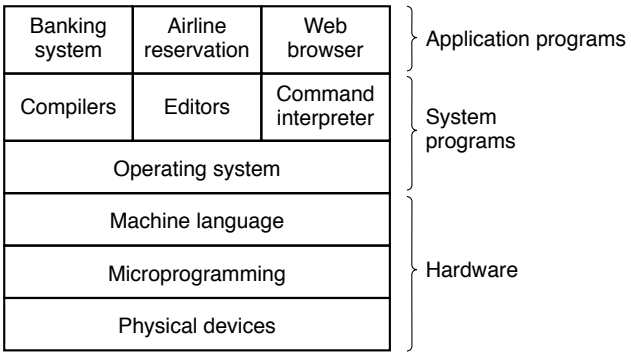


Figure 1-1. A computer system consists of hardware, system programs, and application programs.

Some computers, called **RISC (Reduced Instruction Set Computers)** machines, do not have a microprogramming level. On these machines, the hardware executes the machine language instructions directly. As examples, the Motorola 680x0 has a microprogramming level, but IBM PowerPC does not.

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

A major function of the operating system is to hide all this complexity and give 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 th 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 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 of the old microprocessors that do not have hardware protection as all). Compilers and editors run in **user mode**. If a user does not like a particular compiler, he[1] is free to write his own if he so chooses; he is not free to write his own disk interrupt handler, which is part of the operating system and is normally protected by hardware against attempts by users to modify 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 game playing.

[1] "He" should be read as "he or she" throughout the book

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, 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 PD765 (or equivalent) controller chip, which is used on many 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.

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.

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 was named after her.

1.2.1 The First Generation (1945—1955): 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 University, John von Neumann at the Institute for Advanced Study in Princeton, J. Presper Eckert and John 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 machine language, often by wiring up plugboards to control the machine's basic functions. Programming languages were unknown (even assembly language was unknown). Operating systems were unheard of. The usual mode of operation was for the programmer to sign up for a block of time on the signup sheet on the wall, then come down to the machine room, insert his or her plugboard into the computer, and spend the next few hours hoping that none of the 20,000 or so vacuum tubes would burn out during the run. Virtually all the problems were straightforward numerical calculations, such as grinding out tables of sines, cosines, and logarithms.

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

1.2.2 The Second Generation (1955—1965): Transistors and Batch Systems

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

These machines were locked away in specially airconditioned computer rooms, with staffs of professional operators to run them. Only big corporations or major government agencies or universities could afford their multimillion dollar price tags. To run a **job** (i.e., a program or set of programs), a programmer would first write the program on paper (in FORTRAN or assembler), then punch it on cards. He would then bring the card deck down to the input room and hand it to one of the operators.

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

Given the high cost of the equipment, it is not surprising that people quickly looked for ways to reduce the wasted time. The solution generally adopted was the **batch system**. The idea behind it was to collect a tray full of jobs in the input room and then read them

onto a magnetic tape using a small (relatively) inexpensive computer, such as the IBM 1401, which was very good at reading cards, copying tapes, and printing output, but not at all good at numerical calculations. Other, much more expensive machines, such as the IBM 7094, were used for the real computing. This situation is shown in Fig. 1-2.

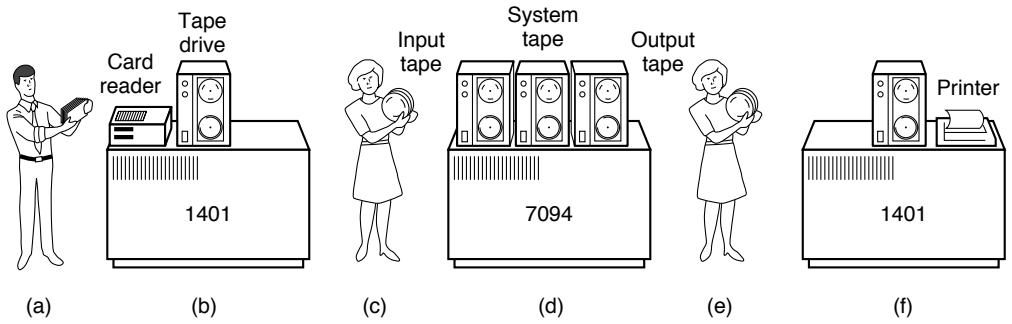


Figure 1-2. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

After about an hour of collecting a batch of jobs, the tape was rewound and brought into the machine room, where it was mounted on a tape drive. The operator then loaded a special program (the ancestor of today's operating system), which read the first job from tape and ran it. The output was written onto a second tape, instead of being printed. After each job finished, the operating system automatically read the next job from the tape and began running it. When the whole batch was done, the operator removed the input and output tapes, replaced the input tape with the next batch, and brought the output tape to a 1401 for printing **off line** (i.e., not connected to the main computer).

The structure of a typical input job is shown in Fig. 1-3. It started out with a $JOB card, specifying the maximum run time in minutes, the account number to be charged, and the programmer's name. Then came a $FORTRAN card, telling the operating system to load the FORTRAN compiler from the system tape. It was followed by the program to be compiled, and then a $LOAD card, directing the operating system to load the object program just compiled. (Compiled programs were often written on scratch tapes and had to be loaded explicitly.) Next came the $RUN card, telling the operating system to run the program with the data following it. Finally, the $END card marked the end of the job. These primitive control cards were the forerunners of modern job control languages and command interpreters.

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

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

By the early 1960s, most computer manufacturers had two distinct, and totally incompatible, product lines. On the one hand there were the word-oriented, large-scale scientific

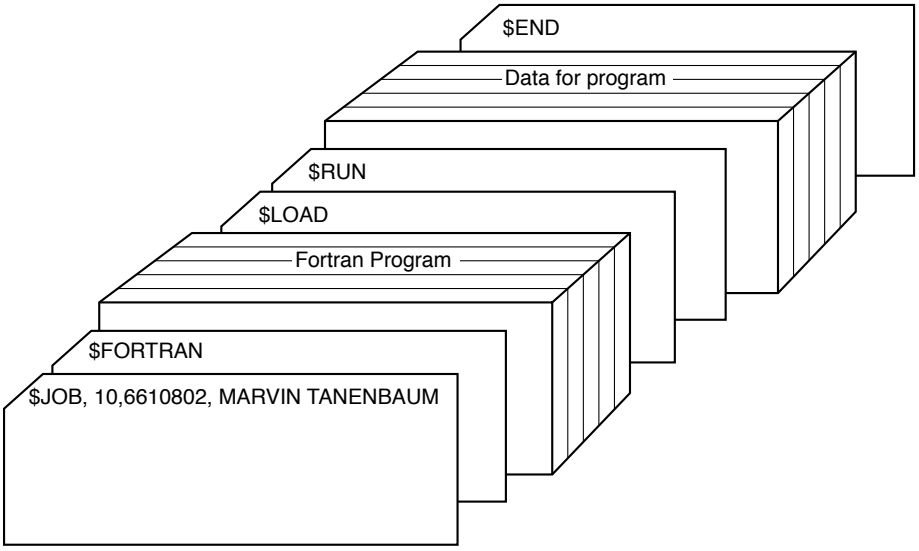


Figure 1-3. Structure of a typical FMS job.

computers, such as the 7094, which were used for numerical calculations in science and engineering. On the other hand, there were the character-oriented, commercial computers, such as the 1401, which were widely used for tape sorting and printing by banks and insurance companies.

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

BM attempted to solve both of these problems at a single stroke by introducing the System/360. The 360 was a series of software-compatible machines ranging from 1401-sized to much more powerful than the 7094. The machines differed only in price and performance (maximum memory, processor speed, number of I/O devices permitted, and so forth). Since all the machines had the same architecture and instruction set, programs written for one machine could run on all the others, at least in theory. Furthermore, the 360 was designed to handle both scientific and commercial computing. Thus a single family of machines could satisfy the needs of all customers. In subsequent years, IBM has come out with compatible successors to the 360 line, using more modern technology, known as the 370, 4300, 3080, 3090.

The 360 was the first major computer line to use (small-scale) Integrated Circuits (ICs), thus providing a major price/performance advantage over the second-generation machines, which were built up from individual transistors. It was an immediate success, and the idea of a family of compatible computers was soon adopted by all the other major manufacturers. The descendants of these machines are still in use at computer centers today, but their use is declining rapidly.

The greatest strength of the "one family" idea was simultaneously its greatest weakness. The intention was that all software, including the operating system, OS/360, had to work on all models. It had to run on small systems, which often just replaced 1401s for copying cards to tape, and on very large systems, which often replaced 7094s for doing weather forecasting and other heavy computing. It had to be good on systems with few

peripherals and on systems with many peripherals. It had to work in commercial environments and in scientific environments. Above all, it had to be efficient for all of these different uses.

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

One of the designers of OS/360, Fred Brooks, subsequently wrote a witty and incisive book describing his experiences with OS/360 (Brooks, 1975). While it would be impossible to summarize the book here, suffice it to say that the cover shows a herd of prehistoric beasts stuck in a tar pit. The cover of Silberschatz and Galvin's book (1994) makes a similar point.

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

The solution that evolved was to partition memory into several pieces, with a different job in each partition, as shown in Fig. 1-4. While one job was waiting for I/O to complete, another job could be using the CPU. If enough jobs could be held in main memory at once, the CPU could be kept busy nearly 100 percent of the time. Having multiple jobs safely in memory at once requires special hardware to protect each job against snooping and mischief by the other ones, but the 360 and other third-generation systems were equipped with this hardware.

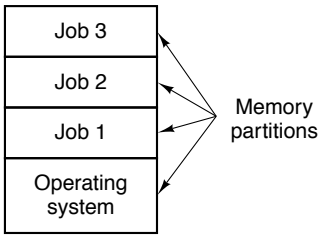


Figure 1-4. A multiprogramming system with three jobs in memory.

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

Although third-generation operating systems were well suited for big scientific cal-

culations and massive commercial data processing runs, they were still basically batch systems. Many programmers pined for the first-generation days when they had the machine all to themselves for a few hours, so they could debug their programs quickly. With third-generation systems, the time between submitting a job and getting back the output was often hours, so a single misplaced comma could cause a compilation to fail, and the programmer to waste half a day.

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

After the success of the CTSS system, MIT, Bell Labs, and General Electric (then a major computer manufacturer) decided to embark on the development of a "computer utility," a machine that would support hundreds of simultaneous timesharing users. Their model was the electricity distribution systemwhen you need electric power, you just stick a plug in the wall, and within reason, as much power as you need will be there. The designers of this system, known as **MULTICS** (MULTiplexed Information and Computing Service), envisioned one huge machine providing computing power for everyone in the Boston area. The idea that machines far more powerful than their GE-645 mainframe would be sold for under a thousand dollars by the millions only 30 years later was pure science fiction at the time.

To make a long story short, MULTICS introduced many seminal ideas into the computer literature, but turning it into a serious product and a commercial success was a lot harder than anyone had expected. Bell Labs dropped out of the project, and General Electric quit the computer business altogether. Eventually, MULTICS ran well enough to be used in a production environment at MIT and dozens of sites elsewhere, but the concept of a computer utility fizzled out as computer prices plummeted. Still, MULTICS had an enormous influence on subsequent systems. It is described in (Corbato et al., 1972; Corbato and Vyssotsky, 1965; Daley and Dennis, 1968; Organick, 1972; and Saltzer, 1974).

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

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

The history of UNIX has been told elsewhere (e.g., Salus, 1994). Because the source

code was widely available, various organizations developed their own (incompatible) versions, which led to chaos. To make it possible to write programs that could run on any UNIX system, IEEE developed a standard for UNIX, called **POSIX**, that most versions of UNIX now support. POSIX defines a minimal system call interface that conformant UNIX systems must support. In fact, some other operating systems now also support the POSIX interface.

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

With the development of LSI (Large Scale Integration) circuits, chips containing thousands of transistors on a square centimeter of silicon, the age of the microprocessor-based personal computer dawned. In terms of architecture, personal computers were not that different from minicomputers of the PDP-11 class, but in terms of price they certainly were different. Where The minicomputer made it possible for a department in a company or university to have its own computer. The microcomputer made it possible for a single individual to have his or her own computer. The most powerful personal computers used by businesses, universities, and government installations are usually called **workstations**, but they are really just large personally computers. Usually, they are connected together by a network.

The widespread availability of computing power, especially highly interactive computer power usually with excellent graphics, led to the growth of a major industry producing software for personal computers. Much of this software was user friendly meaning that it was intended for users who not only knew nothing about computers but furthermore had absolutely no intention whatsoever of learning. This was certainly a major change from OS/360, whose job control language, JCL, was so arcane that entire books were written about it (e.g., Cadow, 1970).

Two operating systems initially dominated the personal computer and workstation scene: Microsoft's MS-DOS and UNIX. MS-DOS was widely used on the IBM PC and other machines using the Intel 8088 CPU and its successors, the 80286, 80386, and 80486 (which we will refer to henceforth as the 286, 386, and 486, respectively), and later the Pentium and Pentium Pro. Although the initial version of MS-DOS was relatively primitive, subsequent versions have included more advanced features, including many taken from UNIX. Microsoft's successor to MS-DOS, WINDOWS, originally ran on top of MS-DOS (i.e., it was more like a shell than a true operating system), but starting in 1995 a freestanding version of WINDOWS, WINDOWS 95®, was released, so MS-DOS is no longer needed to support it. Another Microsoft operating system is WINDOWS NT, which is compatible with WINDOWS 95 at a certain level, but a complete rewrite from scratch internally.

The other major contender is UNIX, which is dominant on workstations and other high-end computers, such as network servers. It is especially popular on machines powered by high-performance RISC chips. These machines usually have the computing power of a minicomputer, even though they are dedicated to a single user, so it is logical that they are equipped with an operating system originally designed for minicomputers, namely UNIX.

An interesting development that began taking place during the mid-1980s is the growth of networks of personal computers running **network operating systems** and **distributed operating systems** (Tanenbaum, 1995). In a network operating system, the users are aware of the existence of multiple computers and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and

has its own local user (or users).

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

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

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

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

1.2.5 History of MINIX

When UNIX was young (Version 6), the source code was widely available, under AT&T license, and frequently studied. John Lions, of the University of New South Wales in Australia, even wrote a little booklet describing its operation, line by line (Lion, 1996). This booklet was used (with permission of AT&T) as a text in many university operating system courses.

When AT&T released Version 7, it began to realize that UNIX was valuable commercial product, so it issued Version 7 with a license that prohibited the source code from being studied in courses, in order to avoid endangering its status as a trade secret. Many universities complied by simply dropping the study of UNIX and teaching only theory.

Unfortunately, teaching only theory leaves the student with lopsided view of what an operating system is really like. The theoretical topics that are usually covered in great detail in courses and books on operating systems, such as scheduling algorithms, are in practice not really that important. Subjects that really important, such as I/O and file systems, are generally neglected because there is little theory about them.

To remedy this situation, one of the authors of this book (Tanenbaum) decided to write a new operating system from scratch that would be compatible with UNIX from the user's point of view, but completely different on the inside. By not using even one line of AT&T code, this system avoids the licensing restrictions, so it can be used for class or individual study. In this manner, readers can dissect a real operating system to see what is inside, just as biology students dissect frogs. The name MINIX stands for mini-UNIX because it is small enough that even a nonguru can understands how it works.

In addition to the advantages of eliminating the legal problems, MINIX has another advantage over UNIX. It was written a decade after UNIX and has been structured in a more modular way. The MINIX file system, for example, is not part of the operating system at all but runs as a user program. Another difference is that UNIX was designed to be efficient, MINIX was designed to be readable (inasmuch as one can speak of any

program hundreds of pages long as being readable). The MINIX code, for example, has thousands of comments in it.

UNIX. Version 7 was used as the model because of its simplicity and elegance. It is sometimes said that Version 7 was not only an improvement over all its predecessors, but also over all its successors. With the advent of POSIX, MINIX began evolving toward the new standard, while backward compatibility with existing programs. This kind of evolution is common in the computer industry, as no vendor wants to introduce a new system that none of its existing customers can use without great upheaval. The version of MINIX described in this book is based on the POSIX standard (unlike the version described in the First Edition, which was V7 based).

Like UNIX, MINIX was written in the C programming language and was intended to be easy to port to various computers. The initial implementation was for the IBM PC, because this computer is in widespread use. It was subsequently ported to the Atari, Amiga, Macintosh, and SPARC computers. In keeping with the "small is beautiful" philosophy, MINIX originally did not even require a hard disk to run, thus bringing it within range of many students' budgets (amazing as it may seem now, in the mid-1980s when MINIX first saw the light of day, hard disks were still an expensive novelty). As MINIX grew in functionality and size, it eventually got to the point that a hard disk is needed, but in keeping with the MINIX philosophy, a 30-megabyte partition is sufficient. In contrast, some commercial UNIX systems now recommend at least a 200-MB disk partition as the bare minimum.

To the average user sitting at an IBM PC, running MINIX is similar to running UNIX. Many of the basic programs, such as *cat*, *grep*, *ls*, *make*, and the shell are present and perform the same functions as their UNIX counterparts. Like the operating system itself, all these utility programs have been rewritten completely from scratch by the author, his students, and some other dedicated people.

Throughout this book MINIX will be used as an example. Most of the comments about MINIX, however, except those about the actual code, also apply to UNIX. Many of them also apply to other systems as well. This remark should be kept in mind when reading the text.

As an aside, a few words about LINUX and its relationship to MINIX may be of interest to some readers. Shortly after MINIX was released, a USENET newsgroup was formed to discuss it. Within weeks, it had 40,000 subscribers, most of whom wanted to add vast numbers of new features to MINIX to make it bigger and better (well, at least bigger). Every day, several hundred of them offered suggestions, ideas, and snippets of code. The author of MINIX successfully resisted this onslaught for several years, in order to keep MINIX small enough and clean enough for students to understand. Ever so gradually, it began to become clear that he really meant it. At that point, a Finnish student, Linus Torvalds, decided to write a MINIX clone intended to be a feature-heavy production system, rather than an educational tool. Thus was LINUX born.

1.3 Operating System Concepts

The interface between the operating system and the user programs is defined by the set of "extended instructions" that the operating system provides. These extended instructions have been traditionally known as **system calls**, although they can be implemented in several ways. To really understand what operating systems do, we must examine this interface closely. The calls available in the interface vary from operating system to oper-

ating system (although the underlying concepts tend to be similar).

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

We have chosen the latter approach. It's more work that way, but it gives more insight into what operating systems really do. In Sec. 1.4 we will look closely at the basic system calls present in both UNIX and MINIX. For simplicity's sake, we will refer only to MINI, but the corresponding UNIX system calls are based on POSIX in most cases. Before we look at the actual system calls, however, it is worth taking a bird's-eye view of MINIX, to get a general feel for what an operating system is all about. This overview applies equally well to UNIX.

The MINIX system calls fall roughly in two broad categories: those dealing with processes and those dealing with the file system. We will now examine each of these in turn.

1.3.1 Processes

A key concept in MINIX, and in all operating systems, is the **process**. A process is basically a program in execution. Associated with each process is its **address space**, a list of memory locations from some minimum (usually 0) to some maximum, which the process can read and write. The address space contains the executable program, the program's data, and its stack. Also associated with each process is some set of registers, including the program counter, stack pointer, and other hardware registers, and all the other information needed to run the program.

We will come back to the process concept in much more detail in Chap. 2, but for the time being, the easiest way to get a good intuitive feel for a process is to think about multiprogramming systems. Periodically, the operating system decides to stop running one process and start running another, for example, because the first one has had more than its share of CPU time in the past second.

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

Thus, a (suspended) process consists of its address space, usually called the **core image** (in honor of the magnetic core memories used in days of yore), and its process table entry, which contains its registers, among other things.

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

If a process can create one or more other processes (referred to as **child processes**) and these processes in turn can create child processes, we quickly arrive at the process tree structure of Fig. 1-5. Related processes that are cooperating to get some job done often need to communicate with one another and synchronize their activities. This communication is called **interprocess communication**, and will be addressed in detail in Chap. 2.

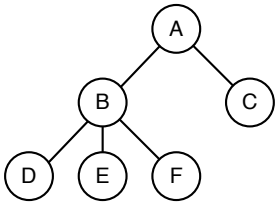


Figure 1-5. A process tree. Process A created two child processes, B and C. Process B created three child processes, D, E, and F..

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

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

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

Each person authorized to use a MINIX system is assigned a **uid** (user identification) by the system administrator. Every process started in MINIX has the uid of the person who started it. A child process has the same uid as its parent. Users .One UID, called the **super-user**, has special power and may violate many of the protection rules. In large installations, only the system administrator knows the password needed to become superuser, but many of the ordinary users (especially students) devote considerable effort to trying to find flaws in the system that allow them to become superuser without the password.

1.3.2 Files

The other broad category of system calls relates to the file system. As noted before, a major function of the operating system is to hide the peculiarities of the disks and other

I/O devices and present the programmer with a nice, clean abstract model of device-independent files. System calls are obviously needed to create files, remove files, read files, and write files. Before a file can be read, it must be opened, and after it has been read it should be closed, so calls are provided to do these things.

To provide a place to keep files, MINIX has the concept of a **directory** as a way of grouping files together. A student, for example, might have one directory for each course he was taking (for the programs needed for that course), another directory for his electronic mail, and still another directory for his World Wide Web home page. System calls are then needed to create and remove directories. Calls are also provided to put an existing file into a directory, and to remove a file from a directory. Directory entries may be either files or other directories. This model also gives rise to a hierarchy—the file systemas—shown in Fig. 1-6.

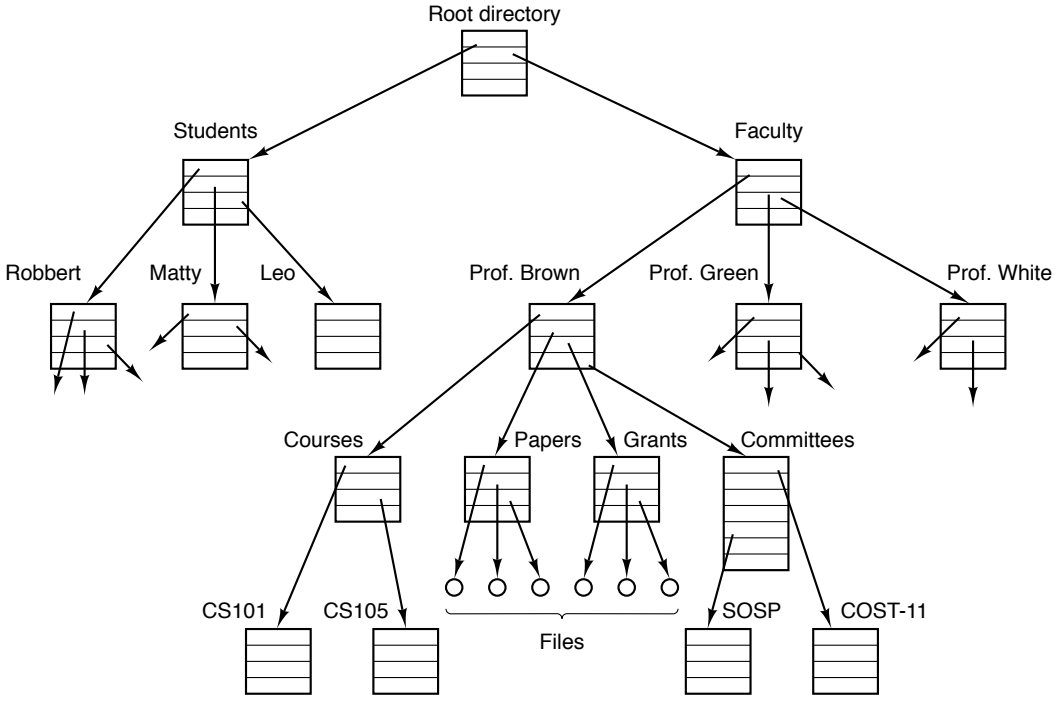


Figure 1-6. A file system for a university department.

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

Every file within the directory hierarchy can be specified by giving its **path name** from the top of the directory hierarchy, the **root directory**. Such absolute path names consist of the list of directories that must be traversed from the root directory to get to the file, with slashes separating the components. In Fig. 1-6, the path for file *CS101* is */Faculty/Prof.Brown/Courses/CS101*. The leading slash indicates that the path is absolute, that is, starting at the root directory.

At every instant, each process has a current **working directory**, in which path names not beginning with a slash are looked for. As an example, in Fig. 1-6, if */Faculty/Prof.Brown* were the working directory, then use of the path name *Courses/CS101* would yield the same file as the absolute path name given above. Processes can change their working directory by issuing a system call specifying the new working directory.

Files and directories in MINIX are protected by assigning each one an 9-bit binary protection code. The protection code consists of three 3-bit fields: one for the owner, one for other members of the owner's group (users are divided into groups by the system administrator), one for everyone else. Each field has a bit for read access, a bit for write access, and a bit for execute access. These 3 bits are known as the **rwx bits**. For example, the protection code *rwxr-x–x* means that the owner can read, write, or execute the file, other group members can read or execute (but not write) the file, and everyone else can execute (but not read or write) the file. For a directory, *x* indicates search permission. A dash means that the corresponding permission is absent.

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

Another important concept in MINIX is the mounted file system. Nearly all personal computers have one or more floppy disk drives into which floppy disks can be inserted and removed. To provide a clean way to deal with these removable media (and also CD-ROMs, which are also removable), MINIX allows the file system on a floppy disk to be attached to the main tree. Consider the situation of Fig. 1-7(a). Before the MOUNT call, the RAM disk (simulated disk in main memory) contains the primary, or **root file system**, and drive 0 contains a floppy disk containing another file system.

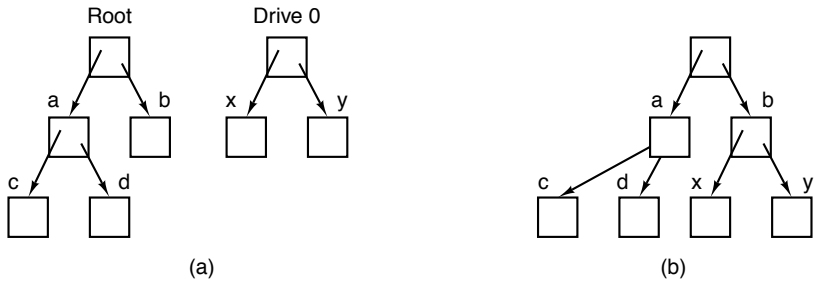


Figure 1-7. (a) Before mounting, the files on drive 0 are not accessible. (b) After mounting, they are part of the file hierarchy.

However, the file system on the drive 0 cannot be used, because there is no way to specify path names on it. MINIX does not allow path names to be prefixed by a drive name or number; that is precisely the kind of device dependence that operating systems ought to eliminate. Instead, the MOUNT system call allows the file system on the drive 0 to be attached to the root file system wherever the program wants it to be. In Fig. 1-7(b) the file system on drive 0 has been mounted on directory *b*, thus allowing access to files */b/x* and */b/y*. If directory *b* had originally contained any files they would not be accessible while the drive 0 was mounted, since */b* would refer to the root directory of drive 0. (Not being able to access these files is not as serious as it at first seems: file systems are nearly always mounted on empty directories.)

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

The last feature we will discuss in this overview is one that relates to both processes and files: pipes. A **pipe** is a sort of pseudofile that can be used to connect two processes, as shown in Fig. 1-8. When process A wants to send data to process B, it writes on the pipe as though it were an output file. Process B can read the data by reading from the pipe as though it were an input file. Thus, communication between processes in MINIX looks very much like ordinary file reads and writes. Stronger yet, the only way a process can discover that the output file it is writing on is not really a file, but a pipe, is by making a special system call.

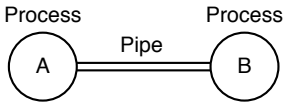


Figure 1-8. Two processes connected by a pipe.

1.3.3 The Shell

The MINIX operating system is the code that carries out the system calls. Editors, compilers, assemblers, linkers, and command interpreters definitely are not part of the operating system, even though they are important and useful. At the risk of confusing things somewhat, in this section we will look briefly at the MINIX command interpreter, called the **shell**, which although not part of the operating system, makes heavy use of many operating system features and thus serves as a good example of how the system calls can be used. It is also the primary interface between a user sitting at his terminal and the operating system.

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

```
date
```

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

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

```
date >file
```

Similarly, standard input can be redirected, as in

```
sort <file1 >file2
```

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

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

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

invokes the *cat* program to concatenate three files and send the output to sort to arrange all the lines in alphabetical order. The output of *sort* is redirected to the file */dev/lp*, which is a typical name for the special character file for the printer. (By convention, all the special files are kept in the directory */dev*.)

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

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

starts up the sort as a background job, allowing the user to continue working normally while the sort is going on. The shell has a number of other interesting features, which we do not have space to discuss here. See any of the suggested references on UNIX for more information about the shell.

1.4 System Calls

Armed with our general knowledge of how MINIX deals with processes and files, we can now begin to look at the interface between the operating system and its application programs, that is, the set of system calls. Although this discussion specifically refers to POSIX (International Standard 9945-1), hence also to MINI, most other modern operating systems have system calls that perform the same functions, even if the details differ. Since the actual mechanics of issuing a system call are highly machine dependent, and often must be expressed in assembly code, a procedure library is provided to make it possible to make system calls from C programs.

To make the system call mechanism clearer, let us take a quick look at READ. It has three parameters: the first one specifying the file, the second one specifying the buffer, and the third one specifying the number of bytes to read. A call to read from a C program might look like this:

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

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

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

MINIX has a total of 53 main system calls. These are listed in Fig. 1-9, grouped for convenience in six categories. In the following sections we will briefly examine each call to see what it does. To a large extent, the services offered by these calls determine most of what the operating system has to do, since the resource management on personal computers is minimal (at least compared to big machines with many users).

Process management	pid = fork()	Create a child process identical to the parent
	pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
	s = wait(&status)	Old version of waitpid
	s = execve(name, argv, envp)	Replace a process core image
	exit(status)	Terminate process execution and return status
	size = brk(addr)	Set the size of the data segment
	pid = getpid()	Return the caller's process id
	pid = getpgrp()	Return the id of the caller's process group
	pid = setsid()	Create a new session and return its process group id
	l = ptrace(req, pid, addr, data)	Used for debugging
Signals	s = sigaction(sig, &act, &oldact)	Define action to take on signals
	s = sigreturn(&context)	Return from a signal
	s = sigprocmask(how, &set, &old)	Examine or change the signal mask
	s = sigpending(set)	Get the set of blocked signals
	s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
	s = kill(pid, sig)	Send a signal to a process
	residual = alarm(seconds)	Set the alarm clock
	s = pause()	Suspend the caller until the next signal
File Management	fd = creat(name, mode)	Obsolete way to create a new file
	fd = mknod(name, mode, addr)	Create a regular, special, or directory i-node
	fd = open(file, how, ...)	Open a file for reading, writing or both
	s = close(fd)	Close an open file
	n = read(fd, buffer, nbytes)	Read data from a file into a buffer
	n = write(fd, buffer, nbytes)	Write data from a buffer into a file
	pos = lseek(fd, offset, whence)	Move the file pointer
	s = stat(name, &buf)	Get a file's status information
	s = fstat(fd, &buf)	Get a file's status information
	fd = dup(fd)	Allocate a new file descriptor for an open file
	s = pipe(&fd[0])	Create a pipe
	s = ioctl(fd, request, argp)	Perform special operations on a file
	s = access(name, amode)	Check a file's accessibility
	s = rename(old, new)	Give a file a new name
	s = fcntl(fd, cmd, ...)	File locking and other operations
Directory & File System Management	s = mkdir(name, mode)	Create a new directory
	s = rmdir(name)	Remove an empty directory
	s = link(name1, name2)	Create a new entry, name2, pointing to name1
	s = unlink(name)	Remove a directory entry
	s = mount(special, name, flag)	Mount a file system
	s = umount(special)	Unmount a file system
	s = sync()	Flush all cached blocks to the disk
	s = chdir(dirname)	Change the working directory
	s = chroot(dirname)	Change the root directory
Protection	s = chmod(name, mode)	Change a file's protection bits
	uid = getuid()	Get the caller's uid
	gid = getgid()	Get the caller's gid
	s = setuid(uid)	Set the caller's uid
	s = setgid(gid)	Set the caller's gid
	s = chown(name, owner, group)	Change a file's owner and group
	oldmask = umask(complmode)	Change the mode mask
Time Management	seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970
	s = stime(tp)	Set the elapsed time since Jan. 1, 1970
	s = utime(file, timep)	Set a file's "last access" time
	s = times(buffer)	Get the user and system times used so far

Figure 1-9. The MINIX system calls. The return code *s* is -1 if an error has occurred; *fd* is file descriptor; and *n* is a byte count. The other return codes are what the name suggests.

As an aside, it is worth pointing out that what constitutes a system call is open to some interpretation. The POSIX standard specifies a number of procedures that a conformant system must supply, but it does not specify whether they are system calls, library calls, or something else. In some cases, the POSIX procedures are supported as library routines in MINIX. In others, several required procedures are only minor variations of one another, and one system call handles all of them.

1.4.1 System Calls for Process Management

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

In most cases, after a FORK, the child will need to execute different code from the parent. Consider the case of the shell. It reads a command from the terminal, forks off a child process, waits for the child to execute the command, and then reads the next command when the child terminates. To wait for the child to finish, the parent executes a WAITPID system call, which just waits until the child terminates (any child if more than one exists). WAITPID can wait for a specific child, or for any old child by setting the first parameter to -1. When WAITPID completes, the address pointed to by the second parameter will be set to the child's exit status (normal or abnormal termination and exit value). Various options are also provided. The WAITPID call replaces the previous WAIT call, which is now obsolete but is provided for reasons of backward compatibility.

Now consider how fork is used by the shell. When a command is typed, the shell forks off a new process. This child process must execute the user command. It does this by using the EXEC system call, which causes its entire core image to be replaced by the file named in its first parameter. A highly simplified shell illustrating the use of FORK, WAITPID, and EXEC is shown in Fig. 1-10.

```
while (TRUE) {                  /* repeat forever */
  read_command(command, parameters);/* read input from terminal */

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

Figure 1-10. A stripped-down shell. Throughout this book, TRUE is assumed to be defined as 1.

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

Let us consider the case of a command such as

```
cp file1 file2
```

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

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

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

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

The second parameter, *argv*, is a pointer to an array. Element *i* of that array is a pointer to the *i*-th string on the command line. In our example, *argv*[0] would point to the string "cp." (As an aside, the string pointed to contains *two* characters, a "c" and a "p", although, if you look closely at the previous sentence you will also see a period inside the quotes. The period ends the sentence, but the rules of English punctuation require most punctuation marks to be *inside* the quotes, even though this is totally illogical. Hopefully, this will not cause any confusion.) Similarly, *argv*[1] would point to the 5-character string "file1" and *argv*[2] would point to the 5-character string "file2".

The third parameter of main, *envp*, is a pointer to the environment, an array of strings containing assignments of the form *name* = *value* used to pass information such as the terminal type and home directory name to a program. In Fig. 1-10, no environment is passed to the child, so the third parameter of *execve* is a zero.

If EXEC seems complicated, do not despair; it is the most complex system call. All the other ones are much simpler. As an example of a simple one, consider EXIT, which processes should use when they are finished executing. It has one parameter, the exit status (0 to 255), which is returned to the parent in the variable *status* of the WAIT or WAITPID system call. The low-order byte of *status* contains the termination status, with 0 being normal termination and the other values being various error conditions. The high-order byte contains the child's exit status (0 to 255). For example, if a parent process executes the statement

```
n = waitpid(-1, &status, options);
```

it will be suspended until some child process terminates. If the child exits with, say, 4 as the parameter to *exit*, the parent will be awakened with *n* set to the child's PID and *status* set to 0x0400 (the C convention of prefixing hexadecimal constants with 0x will be used throughout this book).

Processes in MINIX have their memory divided up into three segments: the **text segment** (i.e., the program code), the **data segment** (i.e., the variables), and the **stack segment**. The data segment grows upward and the stack grows downward, as shown in Fig. 1-11. Between them is a gap of unused address space. The stack grows into the gap automatically, as needed, but expansion of the data segment is done explicitly by using the

BRK system call. It has one parameter, which gives the address where the data segment is to end. This address may be more than the current value (data segment is growing) or less than the current value (data segment is shrinking). The parameter must, of course, be less than the stack pointer or the data and stack segments would overlap, which is forbidden.

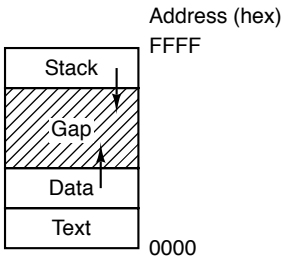


Figure 1-11. Processes have three segments: text, data, and stack. In this example, all three are in one address space, but separate instruction and data space is also supported.

As a convenience to the programmer, a library routine *sbrk* is provided that also changes the size of the data segment, only its parameter is the number of bytes to add to the data segment (negative parameters make the data segment smaller). It works by keeping track of the current size of the data segment, which is the value returned by BRK, computing the new size, and making a call asking for that number of bytes. BRK and SBRK are considered too implementation dependent and are not part of POSIX.

The next process system call is also the simplest, GETPID. It just returns the caller's pid. Remember that in FORK, only the parent was given the child's PID. If the child wants to find out its own pid, it must use GETPID. The GETPGRP call returns the pid of the caller's process group. SETSID creates a new session and sets the process group's pid to the caller's. Sessions are related to an optional feature of POSIX called **job control**, which is not supported by MINIX and which will not concern us further.

The last process management system call, PTRACE, is used by debugging programs to control the program being debugged. It allows the debugger to read and write the controlled process' memory and manage it in other ways.

1.4.2 System Calls for Signaling

Although most forms of interprocess communication are planned, situations exist in which unexpected communication is needed. For example, if a user accidently tells a text editor to list the entire contents of a very long file, and then realizes the error, some way is needed to interrupt the editor. In MINIX, the user can hit the DEL key on the keyboard, which sends a signal to the editor. The editor catches the signal and stops the print-out. Signals can also be used to report certain traps detected by the hardware, such as illegal instruction or floating point overflow. Timeouts are also implemented as signals.

When a signal is sent to a process that has not announced its willingness to accept that signal, the process is simply killed without further ado. To avoid this fate, a process can use the SIGACTION system call to announce that it is prepared to accept some signal type, and to provide the address of the signal handling procedure and a place to store the address of the current one. After a SIGACTION call, if a signal of the relevant type is generated (e.g., the DEL key), the state of the process is pushed onto its own stack, and then the signal handler is called. It may run for as long as it wants to and perform any

system calls it wants to. In practice, though, signal handlers are usually fairly short. When the signal handling procedure is done, it calls SIGRETURN to continue where it left off before the signal. The SIGACTION call replaces the older SIGNAL call, which is now provided as a library procedure, however, for backward compatibility.

Signals can be blocked in MINIX. A blocked signal is held pending until it is unblocked. It is not delivered, but also not lost. The SIGPROCMASK call allows a process to define the set of blocked signals by presenting the kernel with a bitmap. It is also possible for a process to ask for the set of signals currently pending but not allowed to be delivered due to their being blocked. The SIGPENDING call returns this set as a bitmap. Finally, the SIGSUSPEND call allows a process to atomically set the bitmap of blocked signals and suspend itself.

Instead of providing a function to catch a signal, the program may also specify the constant SIG_IGN to have all subsequent signals of the specified type ignored, or SIG_DFL to restore the default action of the signal when it occurs. The default action is either to kill the process or ignore the signal, depending upon the signal. As an example of how SIG_IGN is used, consider what happens when the shell forks off a background process as a result of

```
command &
```

It would be undesirable for a DEL signal from the keyboard to affect the background process, so after the FORK but before the EXEC, the shell does

```
sigaction(SIGINT, SIG_IGN, NULL);
```

and

```
sigaction(SIGQUIT, SIG_IGN, NULL);
```

to disable the DEL and quit signals. (the quit signal is generated by CTRL-\; it is the same as DEL except that if it is not caught or ignored it makes a core dump of the process killed.) For foreground processes (no ampersand), these signals are not ignored.

Hitting the DEL key is not the only way to send a signal. The KILL system call allows a process to signal another process (provided they have the same uid—unrelated processes cannot signal each other). Getting back to the example of background processes used above, suppose a background process is started up, but later it is decided that the process should be terminated. SIGINT and SIGQUIT have been disabled, so something else is needed. The solution is to use the *kill* program, which uses the KILL system call to send a signal to any process. By sending signal 9 (SIGKILL), to a background process, that process can be killed. SIGKILL cannot be caught or ignored.

For many real-time applications, a process needs to be interrupted after a specific time interval to do something, such as to retransmit a potentially lost packet over an unreliable communication line. To handle this situation, the ALARM system call has been provided. The parameter specifies an interval, in seconds, after which a SIGALRM signal is sent to the process. A process may only have one alarm outstanding at any instant. If an ALARM call is made with a parameter of 10 seconds, and then 3 seconds later another alarm call is made with a parameter of 20 seconds, only one signal will be generated, 20 seconds after the second call. The first signal is canceled by the second call to ALARM. if the parameter to alarm is zero, any pending alarm signal is canceled. If an alarm signal is not caught, the default action is taken and the signaled process is killed.

It sometimes occurs that a process has nothing to do until a signal arrives. For example, consider a computer-aided-instruction program that is testing reading speed and

comprehension. It displays some text on the screen and then calls ALARM to signal it after 30 seconds. While the student is reading the text, the program has nothing to do. It could sit in a tight loop doing nothing, but that would waste CPU time that another process or user might need. A better idea is to use PAUSE, which tells MINIX to suspend the process until the next signal.

1.4.3 System Calls for File Management

Many system calls relate to the file system. In this section we will look at calls that operate on individual files; in the next one we will examine those that involve directories or the file system as a whole. To create a new file, the CREAT call is used (why the call is CREAT and not CREATE has been lost in the mists of time). Its parameters provide the name of the file and the protection mode. Thus

```
fd = creat("abc", 0751);
```

creates a file called *abc* with mode 0751 octal (in C, a leading zero means that a constant is in octal). The low-order 9 bits of 0751 specify the *rwx* bits for the owner (7 means read-write-execute permission), his group (5 means read-execute), and others (1 means execute only).

CREAT not only creates a new file but also opens it for writing, regardless of the file's mode. The file descriptor returned, *fd*, can be used to write the file. If a CREAT is done on an existing file, that file is truncated to length 0, provided, of course, that the permissions are all right. The CREAT call is obsolete, as OPEN can now create new files, but it has been included for backward compatibility.

Special files are created using MKNOD rather than CREAT. A typical call is

```
fd = mknod("/dev/ttyc2", 020744, 0x0402);
```

which creates a file named */dev/ttyc2* (the usual name for console 2) and gives it mode 020744 octal (a character special file with protection bits *rwxr–r–*). The third parameter contains the major device (4) in the high-order byte and the minor device (2) in the low-order byte. The major device could have been anything, but a file named */dev/ttyc2* ought to be minor device 2. Calls to MKNOD fail unless the caller is the super-user.

To read or write an existing file, the file must first be opened using OPEN. This call specifies the file name to be opened, either as an absolute path name or relative to the working directory, and a code of *O_RDONLY*, *O_WRONLY*, or *O_RDWR*, meaning open for reading, writing, or both. The file descriptor returned can then be used for reading or writing. Afterward, the file can be closed by CLOSE, which makes the file descriptor available for reuse on a subsequent creat or open.

The most heavily used calls are undoubtedly READ and WRITE. We saw READ earlier; WRITE has the same parameters.

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

LSEEK has three parameters: the first is the file descriptor for the file, the second is a file position, and the third tells whether the file position is relative to the beginning of

the file, the current position, or the end of the file. The value returned by LSEEK is the absolute position in the file after changing the pointer.

For each file, MINIX keeps track of the file mode (regular file, special file, directory, and so on), size, time of last modification, and other information. Programs can ask to see this information via the STAT and FSTAT system calls. These differ only in that the former specifies the file by name, whereas the latter takes a file descriptor, making it useful for open files, especially standard input and standard output, whose names may not be known. Both calls provide as the second parameter a pointer to a structure where the information is to be put. The structure is shown in Fig. 1-12.

```
struct stat {
    short st_dev;             /* device where i-node belongs */
    unsigned short st_ino;    /* i-node number */
    unsigned short st_mode;   /* mode word */
    short st_nlink;           /* number of links */
    short st_uid;             /* user id */
    short st_gid;             /* group id */
    short st_rdev;            /* major/minor device for special files */
    long st_size;             /* file size */
    long st_atime;            /* time of last access */
    long st_mtime;            /* time of last modification */
    long st_ctime;            /* time of last change to i-node */
};
```

Figure 1-12. The structure used to return information for the STAT and FSTAT system calls. In the actual code, symbolic names are used for some of the types.

When manipulating file descriptors, the DUP call is occasionally helpful. Consider, for example, a program that needs to close standard output (file descriptor 1), substitute another file as standard output, call a function that writes some output onto standard output, and then restore the original situation. Just closing file descriptor 1 and then opening a new file will make the new file standard output (assuming standard input, file descriptor 0, is in use), but it will be impossible to restore the original situation later.

The solution is first to execute the statement

```
fd = dup(1);
```

which uses the DUP system call to allocate a new file descriptor, *fd*, and arrange for it to correspond to the same file as standard output. Then standard output can be closed and a new file opened and used. When it is time to restore the original situation, file descriptor 1 can be closed, and then

```
n = dup(fd);
```

executed to assign the lowest file descriptor, namely, 1, to the same file as *fd*. Finally, *fd* can be closed and we are back where we started.

The DUP call has a variant that allows an arbitrary unassigned file descriptor to be made to refer to a given open file. It is called by

```
dup2(fd, fd2);
```

where *fd* refers to an open file and *fd2* is the unassigned file descriptor that is to be made to refer to the same file as *fd*. Thus if *fd* refers to standard input (file descriptor 0) and *fd2* is 4, after the call, file descriptors 0 and 4 will both refer to standard input.

Interprocess communication in MINIX uses pipes, as described earlier. When a user types

```
cat file1 file2 | sort
```

the shell creates a pipe and arranges for standard output of the first process to write to the pipe, so standard input of the second process can read from it. The PIPE system call creates a pipe and returns two file descriptors, one for writing and one for reading. The call is

```
pipe(&fd[0]);
```

where *fd* is an array of two integers and *fd*[0] is the file descriptor for reading and *fd*[1] is the one for writing. Typically, a FORK comes next, and the parent closes the file descriptor for reading and the child closes the file descriptor for writing (or vice versa), so when they are done, one process can read the pipe and the other can write on it.

Figure 1-13 depicts a skeleton procedure that creates two processes, with the output of the first one piped into the second one. (A more realistic example would do error checking and handle arguments.) First a pipe is created, and then the procedure forks, with the parent eventually becoming the first process in the pipeline and the child process becoming the second one. Since the files to be executed, *process1* and *process2*, do not know that they are part of a pipeline, it is essential that the file descriptors be manipulated so that the first process' standard output be the pipe and the second one's standard input be the pipe. The parent first closes off the file descriptor for reading from the pipe. Then it closes standard output and does a DUP call that allows file descriptor 1 to write on the pipe. It is important to realize that DUP always returns the lowest available file descriptor, in this case, 1. Then the program closes the other pipe file descriptor.

After the EXEC call, the process started will have file descriptors 0 and 2 be unchanged, and file descriptor 1 for writing on the pipe. The child code is analogous. The parameter to *execl* is repeated because the first one is the file to be executed and the second one is the first parameter, which most programs expect to be the file name.

The next system call, IOCTL, is potentially applicable to all special files. It is, for instance, used by block device drivers like the SCSI driver to control tape and CD-ROM devices. Its main use, however, is with special character files, primarily terminals. POSIX defines a number of functions which the library translates into IOCTL calls. The *tcgetattr* and *tcsetattr* library functions use IOCTL to change the characters used for correcting typing errors on the terminal, changing the terminal mode, and so forth.

Cooked mode is the normal terminal mode, in which the erase and kill characters work normally, CTRL-S and CTRL-Q can be used for stopping and starting terminal output, CTRL-D means end of file, DEL generates an interrupt signal, and CTRL-\ generates a quit signal to force a core dump.

In **raw mode**, all of these functions are disabled; every character is passed directly to the program with no special processing. Furthermore, in raw mode, a read from the terminal will give the program any characters that have been typed, even a partial line, rather than waiting for a complete line to be typed, as in cooked mode. Screen editors often use this mode.

Cbreak mode is in between. The erase and kill characters for editing are disabled, as is CTRL-D, but CTRL-S, CTRL-Q, DEL, and CTRL-\ are enabled. Like raw mode,

```
#define STD_INPUT 0     /* file descriptor for standard input */
#define STD_OUTPUT 1    /* file descriptor for standard output */

pipeline(process1, process2)
char *process1, *process2;    /* pointers to program names */
{
  int fd[2];

  pipe(&fd[0]);               /* create a pipe */
  if (fork() != 0) {
        /* The parent process executes these statements. */
        close(fd[0]);         /* process 1 does not need to read from pipe */
        close(STD_OUTPUT);    /* prepare for new standard output */
        dup(fd[1]);           /* set standard output to fd[1] */
        close(fd[1]);         /* this file descriptor not needed any more */
        execl(process1, process1, 0);
  } else {
        /* The child process executes these statements. */
        close(fd[1]);         /* process 2 does not need to write to pipe */
        close(STD_INPUT);     /* prepare for new standard input */
        dup(fd[0]);           /* set standard input to fd[0] */
        close(fd[0]);         /* this file descriptor not needed any more */
        execl(process2, process2, 0);
  }
}
```

Figure 1-13. A skeleton for setting up a two-process pipeline.

partial lines can be returned to programs (if intraline editing is turned off there is no need to wait until a whole line has been received—the user cannot change his mind and delete it, as he can in cooked mode).

POSIX does not use the terms cooked, raw, and cbreak. In POSIX terminology **canonical mode** corresponds to cooked mode. In this mode there are eleven special characters defined, and input is by lines. In **noncanonical mode** a minimum number of characters to accept and a time, specified in units of 1/10th of a second, determine how a read will be satisfied. Under POSIX there is a great deal of flexibility, and various flags can be set to make noncanonical mode behave like either cbreak or raw mode. The older terms are more descriptive, and we will continue to use them informally.

IOCTL has three parameters, for example a call to *tcsetattr* to set terminal parameters will result in

```
ioctl(fd, TCSETS, &termios);
```

The first parameter specifies a file, the second one specifies an operation, and the third one is the address of the POSIX structure that contains flags and the array of control characters. Other operation codes instruct the system to postpone the changes until all output has been sent, cause unread input to be discarded, and return the current values.

The ACCESS system call is used to determine whether a certain file access is permitted by the protection system. It is needed because some programs can run using a different user's uid. This SETUID mechanism will be described later.

The RENAME system call is used to give a file a new name. The parameters specify the old and new names.

Finally, the FCNTL call is used to control files, somewhat analogous to IOCTL (i.e., both of them are horrible hacks). It has several options, the most important of which is for advisory file locking. Using FCNTL, it is possible for a process to lock and unlock parts of files and test part of a file to see if it is locked. The call does not enforce any lock semantics. Programs must do this themselves.

1.4.4 System Calls for Directory Management

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

To see how LINK works, consider the situation of Fig. 1-14(a). Here are two users, *ast* and *jim*, each having their own directories with some files. If *ast* now executes a program containing the system call

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

the file *memo* in *jim*'s directory is now entered into *ast*'s directory under the name *note*. Thereafter, */usr/jim/memo* and */usr/ast/note* refer to the same file.

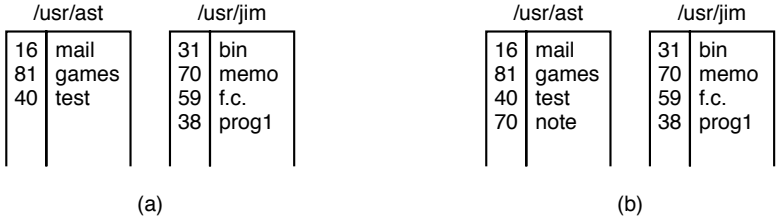


Figure 1-14. (a) Two directories before linking */usr/jim/memo* to ast's directory. (b) The same directories after linking.

Understanding how LINK works will probably make it clearer what it does. Every file in MINIX has a unique number, its i-number, that identifies it. This i-number is an index into a table of i-nodes, one per file, telling who owns the file, where its disk blocks are, and so on. A directory is simply a file containing a set of (i-number, ASCII name) pairs. In Fig. 1-14, *mail* has i-number 16, and so on. What LINK does is simply create a new directory entry with a (possibly new) name, using the i-number of an existing file. In Fig. 1-14(b), two entries have the same i-number (70) and thus refer to the same file. If either one is later removed, using the UNLINK system call, the other one remains. If both are removed, MINIX sees that no entries to the file exist (a field in the i-node keeps track of the number of directory entries pointing to the file), so the file is removed from the disk.

As we have mentioned earlier, the MOUNT system call allows two file systems to be merged into one. A common situation is to have the **root file system** containing the binary (executable) versions of the common commands and other heavily used files, on the RAM disk. The user can then insert a floppy disk, for example, containing user programs, into drive 0.

By executing the MOUNT system call, the drive 0 file system can be attached to the root file system, as shown in Fig. 1-15. A typical statement in C to perform the mount is

```
mount("/dev/fd0", "/mnt", 0);
```

where the first parameter is the name of a block special file for drive 0, the second parameter is the place in the tree where it is to be mounted.

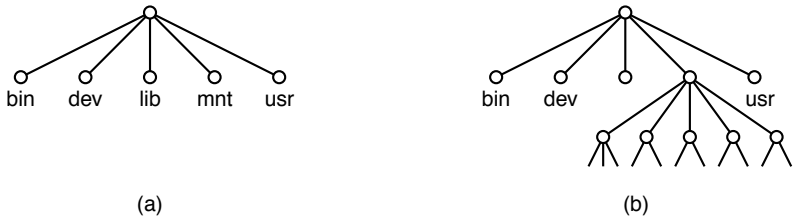


Figure 1-15. (a) File system before the mount. (b) File system after the mount.

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

MINIX maintains a block cache cache of recently used blocks in main memory to avoid having to read them from the disk if they are used again quickly. If a block in the cache is modified (by a WRITE on a file) and the system crashes before the modified block is written out to disk, the file system will be damaged. To limit the potential damage, it is important to flush the cache periodically, so that the amount of data lost by a crash will be small. The system call SYNC tells MINIX to write out all the cache blocks that have been modified since being read in. When MINIX is started up, a program called *update* is started as a background process to do a SYNC every 30 seconds, to keep flushing the cache.

Two other calls that relate to directories are CHDIR and CHROOT. The former changes the working directory and the latter changes the root directory. After the call

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

an open on the file *xyz* will open */usr/ast/test/xyz*. CHROOT works in an analogous way. Once a process has told the system to change its root directory, all absolute path names (path names beginning with a "/") will start at the new root. Only super-users may execute CHROOT, and even super-users do not do it very often.

1.4.5 System Calls for Protection

In MINIX every file has an 11-bit mode used for protection. Nine of these bits are the read-write-execute bits for the owner, group, and others. The CHMOD system call makes it possible to change the mode of a file. For example, to make a file read-only by everyone except the owner, one could execute

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

The other two protection bits, 02000 and 04000, are the SETGID (set-group-id) and SETUID (set-user-id) bits, respectively. When any user executes a program with the SETUID bit on, for the duration of that process the user's effective UID is changed to that of the file's owner. This feature is heavily used to allow users to execute programs that perform super-user only functions, such as creating directories. Creating a directory uses MKNOD, which is for the super-user only. By arranging for the *mkdir* program to be owned by the super-user and have mode 04755, ordinary users can be given the power to execute MKNOD but in a highly restricted way.

When a process executes a file that has the SETUID or SETGID bit on in its mode, it acquires an effective uid or gid different from its real uid or gid. It is sometimes important for a process to find out what its real and effective uid or gid is. The system calls GETUID and GETGID have been provided to supply this information. Each call returns both the real and effective uid or gid, so four library routines are needed to extract the proper information: *getuid*, *getgid*, *geteuid*, and *getegid*. The first two get the real uid/gid, and the last two the effective ones.

Ordinary users cannot change their uid, except by executing programs with the SETUID bit on, but the super-user has another possibility: the SETUID system call, which sets both the effective and real uids. SETGID sets both gids. The super-user can also change the owner of a file with the CHOWN system call. In short, the super-user has plenty of opportunity for violating all the protection rules, which explains why so many students devote so much of their time to trying to become super-user.

The last two system calls in this category can be executed by ordinary user processes. The first one, UMASK, sets an internal bit mask within the system, which is used to mask off mode bits when a file is created. After the call

```
umask(022);
```

the mode supplied by CREAT and MKNOD will have the 022 bits masked off before being used. Thus the call

```
creat("file", 0777);
```

will set the mode to 0755 rather than 0777. Since the bit mask is inherited by child processes, if the shell does a UMASK just after login, none of the user's processes in that session will accidently create files that other people can write on.

When a program owned by the root has the SETUID bit on, it can access any file, because its effective UID is the super-user. Frequently it is useful for the program to know if the person who called the program has permission to access a given file. If the program just tries the access, it will always succeed, and thus learn nothing.

What is needed is a way to see if the access is permitted for the real UID. The ACCESS system call provides a way to find out. The mode parameter is 4 to check for read access, 2 for write access, and 1 for execute access. Combinations of these values are also allowed. For example, with mode equal to 6, the call returns 0 if both read and write access are allowed for the real ID; otherwise -1 is returned. With mode equal to 0, a check is made to see if the file exists and the directories leading up to it can be searched.

1.4.6 System Calls for Time Management

MINIX has four system calls that involve the time-of-day clock. TIME just returns the current time in seconds, with 0 corresponding to Jan. 1, 1970 at midnight (just as the day was starting, not ending). Of course, the system clock must be set at some point in order to allow it to be read later, so STIME has been provided to let the clock be set (by the super-user). The third time call is UTIME, which allows the owner of a file (or the super-user) to change the time stored in a file's i-node. Application of this system call is fairly limited, but a few programs need it, for example, *touch*, which sets the file's time to the current time.

Finally, we have TIMES, which returns the accounting information to a process, so it can see how much CPU time it has used directly, and how much CPU time the system itself has expended on its behalf (handling its system calls). The total user and system times used by all of its children combined are also returned.

1.5 Operating System Structure

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

1.5.1 Monolithic Systems

By far the most common organization, this approach might well be subtitled "The Big Mess." The structure is that there is no structure. The operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to. When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs.

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

Even in monolithic systems, however, it is possible to have at least a little structure. The services (system calls) provided by the operating system are requested by putting the parameters in well-defined places, such as in registers or on the stack, and then executing a special trap instruction known as a **kernel call** or **supervisor call**.

This instruction switches the machine from user mode to kernel mode and transfers control to the operating system, shown as event(1) in Fig. 1-16 (Most CPUs have two modes: kernel mode, for the operating system, in which all instructions are allowed; and user mode, for user programs, in which I/O and certain other instructions are not allowed.)

The operating system then examines the parameters of the call to determine which system call is to be carried out, shown as (2) in Fig. 1-16. Next, the operating system

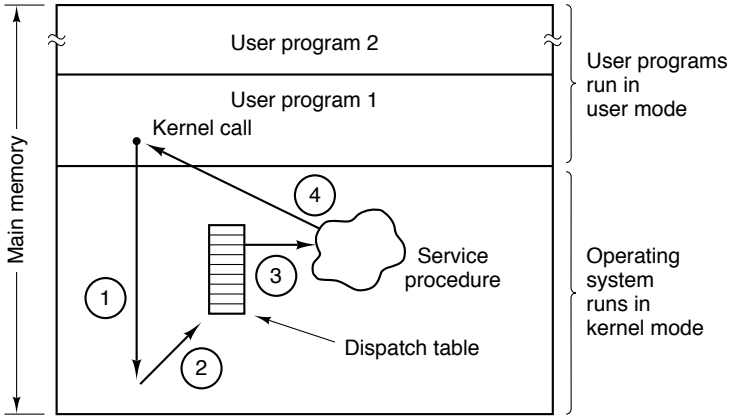


Figure 1-16. How a system call can be made: (1) User program traps to the kernel. (2) Operating system determines service number required. (3) Operating system calls service procedure. (4) Control is returned to user program.

indexes into a table that contains in slot *k* a pointer to the procedure that carries out system call *k*. This operation, shown as (3) in Fig. 1-16, identifies the service procedure, which is then called. When the work has been completed and the system call is finished, control is given back to the user program (step 4), so it can continue execution with the statement following the system call.

This organization suggests a basic structure for the operating system:

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

In this model, for each system call there is one service procedure that takes care of it. The utility procedures do things that are needed by several service procedures, such as fetching data from user programs. This division of the procedures into three layers is shown in Fig. 1-17.

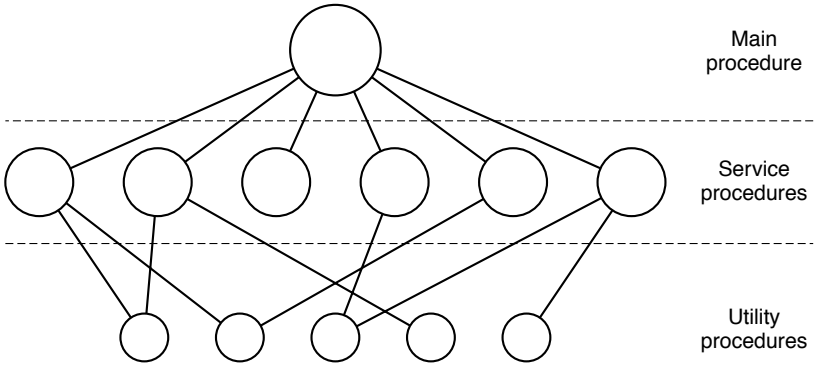


Figure 1-17. A simple structuring model for a monolithic system.

1.5.2 Layered Systems

A generalization of the approach of Fig. 1-17 is to organize the operating system as a hierarchy of layers, each one constructed upon the one below it. The first system con-

structed in this way was the THE system built at the Technische Hogeschool Eindhoven in the Netherlands by E. W. Dijkstra (1968) and his students. The THE system was a simple batch system for a Dutch computer, the Electrologica X8, which had 32K of 27-bit words (bits were expensive back then).

The system had 6 layers, as shown in Fig. 1-18. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Above layer 0, the system consisted of sequential processes, each of which could be programmed without having to worry about the fact that multiple processes were running on a single processor. In other words, layer 0 provided the basic multiprogramming of the CPU.

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

Figure 1-18. Structure of the THE operating system.

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

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

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

Whereas the THE layering scheme was really only a design aid, because all the parts of the system were ultimately linked together into a single object program, in MULTICS, the ring mechanism was very much present at run time and enforced by the hardware. The advantage of the ring mechanism is that it can easily be extended to structure user subsystems. For example, a professor could write a program to test and grade student

programs and run this program in ring n, with the student programs running in ring $n + 1$ so that they could not change their grades.

1.5.3 Virtual Machines

The initial releases of OS/360 were strictly batch systems. Nevertheless, many 360 users wanted to have timesharing, so various groups, both inside and outside IBM decided to write timesharing systems for it. The official IBM timesharing system, TSS/360, was delivered late, and when it finally arrived it was so big and slow that few sites converted over to it. It was eventually abandoned after its development had consumed some $50 million (Graham, 1970). But a group at IBM's Scientific Center in Cambridge, Massachusetts, produced a radically different system that IBM eventually accepted as a product, and which is now widely used on its mainframes.

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

The heart of the system, known as the **virtual machine monitor**, runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up, as shown in Fig. 1-19. However, unlike all other operating systems, these virtual machines are not extended machines, with files and other nice features. Instead, they are *exact* copies of the bare hardware, including kernel/user mode, I/O, interrupts, and everything else the real machine has.

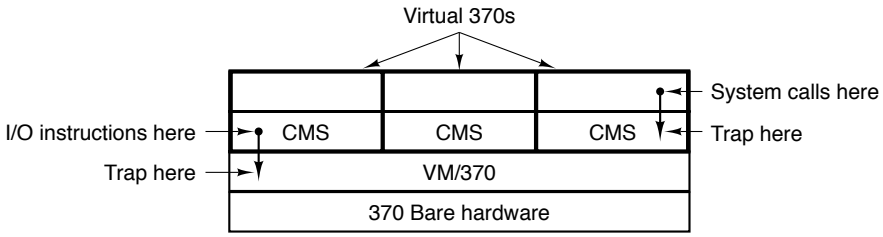


Figure 1-19. The structure of VM/370 with CMS.

Because each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the bare hardware. Different virtual machines can, and frequently do, run different operating systems. Some run one of the descendants of OS/360 for batch or transaction processing, while others run a single-user, interactive system called **CMS** (Conversational Monitor System) for timesharing users.

When a CMS program executes a system call, the call is trapped to the operating-system in its own virtual machine, not to VM/370, just as it would if it were running on a real machine instead of a virtual one. CMS then issues the normal hardware I/O instructions for reading its virtual disk or whatever is needed to carry out the call. These I/O instructions are trapped by VM/370, which then performs them as part of its simulation of the real hardware. By making a complete separation of the functions of multiprogramming and providing an extended machine, each of the pieces can be much simpler, more flexible, and easier to maintain.

The idea of a virtual machine is used nowadays in a different context: running old MS-DOS programs on a Pentium (or other 32-bit Intel CPU). When designing the Pentium

and its software, both Intel and Microsoft realized that there would be a big demand for running old software on new hardware. For this reason, Intel provided a virtual 8086 mode on the Pentium. In this mode, the machine acts like an 8086 (which is identical to an 8088 from a software point of view), including 16-bit addressing with a 1-MB limit.

This mode is used by WINDOWS, OS/2, and other operating systems for running MS-DOS programs. These programs are started up in virtual 8086 mode. As long as they execute normal instructions, they run on the bare hardware. However, when a program tries to trap to the operating system to make a system call, or tries to do protected I/O directly, a trap to the virtual machine monitor occurs.

Two variants on this design are possible. In the first one, MS-DOS itself is loaded into the virtual 8086's address space, so the virtual machine monitor just reflects the trap back to MS-DOS, just as would happen on a real 8086. When MS-DOS later tries to do the I/O itself, that operation is caught and carried out by the virtual machine monitor.

In the other variant, the virtual machine monitor just catches the first trap and does the I/O itself, since it knows what all the MS-DOS system calls are and thus knows what each trap is supposed to do. This variant is less pure than the first one, since it emulates only MS-DOS correctly, and not other operating systems, as the first one does. On the other hand, it is much faster, since it saves the trouble of starting up MS-DOS to do the I/O. A further disadvantage of actually running MS-DOS in virtual 8086 mode is that MS-DOS fiddles around with the interrupt enable/disable bit quite a lot, all of which must be emulated at considerable cost.

It is worth noting that neither of these approaches are really the same as VM/370, since the machine being emulated is not a full Pentium, but only an 8086. With the VM/370 system, it is possible to run VM/370, itself, in the virtual machine. With the pentium, it is not possible to run, say, WINDOWS in the virtual 8086 because no version of WINDOWS runs on an 8086; a 286 is the minimum for even the oldest version, and 286 emulation is not provided (let alone Pentium emulation).

With VM/370, each user process gets an exact copy of the actual computer. With virtual 8086 mode on the Pentium, each user process gets an exact copy of a different computer. Going one step further, researchers at M.I.T. built a system that gives each user a clone of the actual computer, but with a subset of the resources (Engler et al., 1995). Thus one virtual machine might get disk blocks 0 to 1023, the next one might get blocks 1024 to 2047, and so on.

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

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

1.5.4 Client-Server Model

VM/370 gains much in simplicity by moving a large part of the traditional operating system code (implementing the extended machine) into a higher layer, CMS. Nevertheless, VM/370 itself is still a complex program because simulating a number of virtual 370s is not *that* simple (especially if you want to do it reasonably efficiently).

A trend in modern operating systems is to take this idea of moving code up into higher layers even further and remove as much as possible from the operating system, leaving a minimal **kernel**. The usual approach is to implement most of the operating system functions in user processes. To request a service, such as reading a block of a file, a user process (now known as the **client process**) sends the request to a **server process**, which then does the work and sends back the answer.

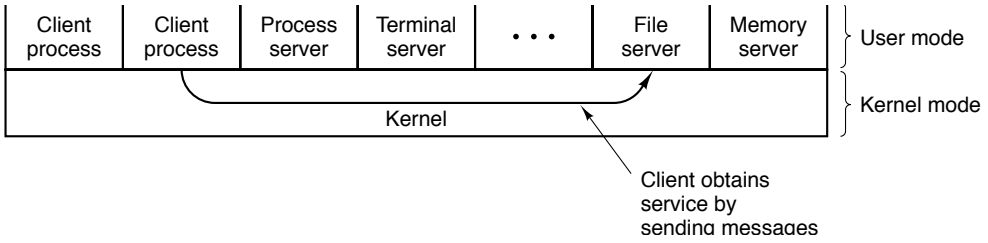


Figure 1-20. The client-server model.

In this model, shown in Fig. 1-20, all the kernel does is handle the communication between clients and servers. By splitting the operating system up into parts, each of which only handles one facet of the system, such as file service, process service, terminal service, or memory service, each part becomes small and manageable. Furthermore, because all the servers run as user-mode processes, and not in kernel mode, they do not have direct access to the hardware. As a consequence, if a bug in the file server is triggered, the file service may crash, but this will not usually bring the whole machine down.

Another advantage of the client-server model is its adaptability to use in distributed systems (see Fig. 1-21). If a client communicates with a server by sending it messages, the client need not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine. As far as the client is concerned, the same thing happens in both cases: a request was sent and a reply came back.

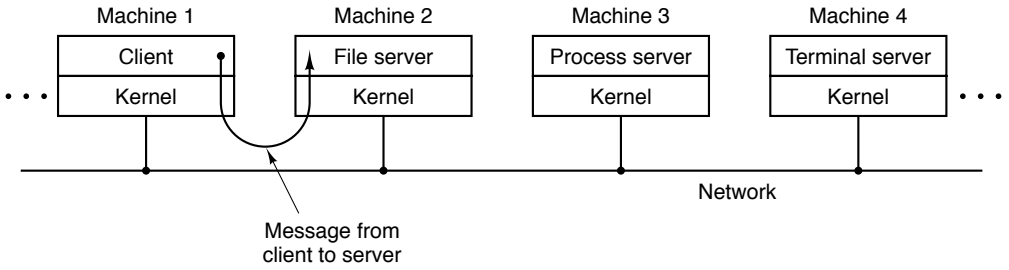


Figure 1-21. The client-server model in a distributed system.

The picture painted above of a kernel that handles only the transport of messages from clients to servers and back is not completely realistic. Some operating system functions (such as loading commands into the physical I/O device registers) are difficult, if not impossible, to do from user-space programs. There are two ways of dealing with this

problem. One way is to have some critical server processes (e.g., I/O device drivers) actually run in kernel mode, with complete access to all the hardware, but still communicate with other processes using the normal message mechanism.

The other way is to build a minimal amount of **mechanism** into the kernel but leave the **policy** decisions up to servers in user space. For example, the kernel might recognize that a message sent to a certain special address means to take the contents of that message and load it into the I/O device registers for some disk, to start a disk read. In this example, the kernel would not even inspect the bytes in the message to see if they were valid or meaningful; it would just blindly copy them into the disk's device registers. (Obviously, some scheme for limiting such messages to authorized processes only must be used.) The split between mechanism and policy is an important concept; it occurs again and again in operating systems in various contexts.

1.6 Outline Of The Rest Of This Book

Operating systems typically have four major components: process management, I/O device management, memory management, and file management. MINIX is also divided into these four parts. The next four chapters deal with these four topics, one topic per chapter. Chapter 6 is a list of suggested readings and a bibliography.

The chapters on processes, I/O, memory management, and file systems have the same general structure. First the general principles of the subject are laid out. Then comes an overview of the corresponding area of MINIX (which also applies to UNIX). Finally, the MINIX implementation is discussed in detail. The implementation section may be skimmed or skipped without loss of continuity by readers just interested in the principles of operating systems and not interested in the MINIX code. Readers who *are* interested in finding out how a real operating system (MINIX) works should read all the sections.

1.7 Summary

Operating systems can be viewed from two viewpoints: resource managers and extended machines. In the resource manager view, the operating system's job is to efficiently manage the different parts of the system. In the extended machine view, the job of the system is to provide the users with a virtual machine that is more convenient to use than the actual machine.

Operating systems have a long history, starting from the days when they replaced the operator, to modern multiprogramming systems.

The heart of any operating system is the set of system calls that it can handle. These tell what the operating system really does. For MINIX, these calls can be divided into six groups. The first group of system calls relates to process creation and termination. The second group handles signals. The third group is for reading and writing files. A fourth group is for directory management. The fifth group protects information, and the sixth group is about keeping track of time.

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

Chapter 2

PROCESSES $\surd$

We are now about no embark on a detailed study of how operating systems, in general, and MINIX, in particular, are designed and constructed. The most central concept in any operating system is the *process*: an abstraction of a running program. Everything else hinges on this concept, and it is important that the operating system designer (and the student) know what a process is as early as possible.

2.1 INTRODUCTION TO PROCESSES

All modern computers can do several things at the same time. While running a user program, a computer can also be reading from a disk and outputting test to a screen or printer. In a multiprogramming system, the CPU also switches from program to program, running each for tens or hundreds of milliseconds. While, strictly speaking, at any instant of time, the CPU is only running only one program, in the course of 1 second, it may work on several programs, thus giving the users the illusion of parallelism. Sometimes people speak of pseusoparallelism to mean this rapid switching back and forth of the CPU between programs, to contrast it with the true hardware parallelism of multiprocessor system (which have two or more CPUs sharing the same physical memory). Keeping track of multiple, parallel activities is hard for people to do. Therefore, operating system designers over the years have evolved a model (sequential processes) that makes parallelism easier to deal with. That model and its uses are the subject of this chapter.

2.1.1 The Process Model

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

In Fig. 2-1(a) we see a computer multiprogramming four programs in memory. In Fig. 2-1(b) we see four processes, each with its own flow of control (i.e., its own program counter), and each one running independently of the other ones. In Fig. 2-1(c) we see that

viewed over a long enough time interval, all the processes have made progress, but at any given instant only one process is actually running.

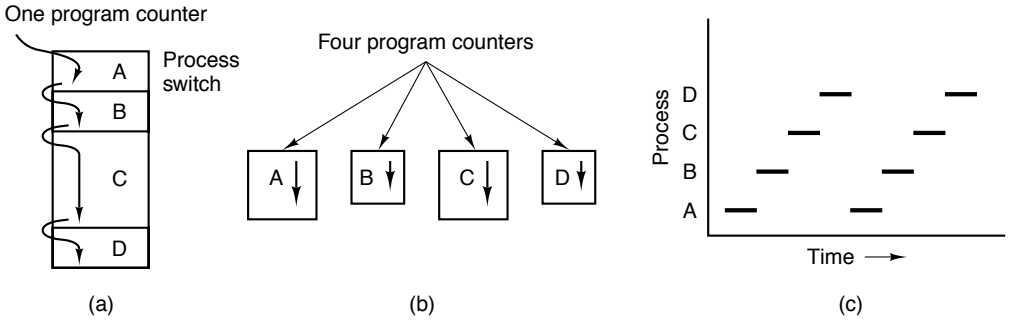


Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at any instant.

With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform, and probably not even reproducible if the same processes are run again. Thus, processes must not be programmed with built-in assumptions about timing. Consider, for example, an I/O process that starts a streamer tape to restore backed up files, executes an idle loop 10,000 times to let it get up to speed, and then issues a command to read the first record. If the CPU decides to switch to another process during the idle loop, the tape process might not run again until after the first record was already past the read head. When a process has critical real-time requirements like this, that is, particular events *must* occur within a specified number of milliseconds, special measures must be taken to ensure that they do occur. Normally, however, most processes are not affected by the underlying multiprogramming of the CPU or the relative speeds of different processes.

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

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

The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state. A single processor may be shared among several processes, with some scheduling algorithm being used to determine when to stop work on one process and service a different one.

Process Hierarchies

Operating systems that support the process concept must provide some way to create all the processes needed. In very simple systems, or in systems designed for running only a single application (e.g., controlling a device in real time), it may be possible to have all the processes that will ever be needed be present when the system comes up. In most systems, however, some way is needed to create and destroy processes as needed during operation. In MINIX, processes are created by the FORK system call, which creates an identical copy of the calling process. The child process can also execute FORK, so it is possible to get a whole tree of processes. In other operating systems, system calls exist to create a process, load its memory, and start it running. Whatever the exact nature of the system call, processes need a way to create other processes. Note that each process has one parent but zero, one, two, or more children.

As a simple example of how process trees are used, let us look at how MINIX initializes itself when it is started. A special process, called *init*, is present in the boot image. When it starts running, it reads a file telling how many terminals there are. Then it forks off one new process per terminal. These processes wait for someone to log in. If a login is successful, the login process executes a shell to accept commands. These commands may start up more processes, and so forth. Thus, all the processes in the whole system belong to a single tree, with *init* at the root. (The code for *init* is not listed in the book: neither is the shell. The line had to be drawn somewhere.)

Process States

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

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

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

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

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

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

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

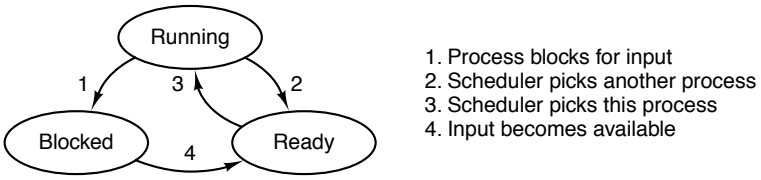


Figure 2-2. A process can be in running, blocked. or ready state. Transitions between these states are as shown.

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

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

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

This view gives rise to the model shown in Fig. 2-3. Here the lowest level of the operating system is the scheduler, with a variety of processes on top of it. All the interrupt handling and details of actually starting and stopping processes are hidden away in the scheduler, which is actually quite small. The rest of the operating system is nicely structured in process form. The model of Fig. 2-3 is used in MINIX, with the understanding that "scheduler" really means not just process scheduling, but also interrupt handling and all the interprocess communication, Nevertheless, to a first approximation, it does show

the basic structure.

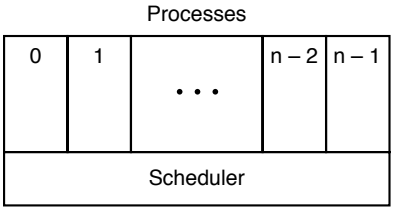


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

2.1.2 Implementation of Processes

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

In MINIX the process management, memory management, and file management are each handled by separate modules within the system, so the process table is partitioned, with each module maintaining the fields that it needs. Figure 2-4 shows some of the more important fields. The fields in the first column are the only ones relevant to this chapter. The other two columns are provided just to give an idea of what information is needed elsewhere in the system.

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	UMASK umask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	file descriptors
Process state	Signal status	Effective uid
Time when process started	Process id	Effective gid
CPU time used	Parent process	System calll parameters
Children's CPU time	Process group	Various flag
Time of the next alarm	Real uid	
Message queue pointers	Effective uid	
Pending signal bits	Real gid	
Process id	Effective gid	
Various flag bits	Bit map for signals	
	Various flag bits	

Figure 2-4. Some of the fields of the MINIX process table.

Now that we have looked at the process table, it is possible to explain a little more about how the illusion of multiple sequential processes is maintained on a machine with one CPU and many l/O devices. What follows is technically a description of how the "scheduler" of Fig. 2-3 works in MINIX but most modem operating systems work essentially the same way. Associated with each I/O device class (e.g., floppy disks, hard disks, timers, tenninals) is a location near the bottom of memory called the **interrupt vector**. It contains the address of the interrupt service procedure. Suppose that user process 3 is

running when a disk interrupt occurs. The program counter, program status word, and possibly one or more registers are pushed onto the (current) stack by the interrupt hardware. The computer then jumps to the address specified in the disk interrupt vector. That is all the hardware does. From here on, it is up to the software.

The interrupt service procedure starts out by saving all the registers in the process table entry for the current process. The current process number and a pointer to its entry are kept in global variables so they can be found quickly. Then the information deposited by the interrupt is removed from the stack. and the stack pointer is set to a temporary stack used by the process handler. Actions such as saving the registers and setting the stack pointer cannot even be expressed in C, so they are performed by a small assembly language routine. When this routine is finished, it calls a C procedure to do the rest of the work.

Interprocess communication in MINIX is via messages, so the next step is to build a message to be sent to the disk process, which will be blocked waiting for it. The message says that an interrupt occurred, to distinguish it from messages from user processes requesting disk blocks to be read and things like that. The state of the disk process is now changed from *blocked* to *ready* and the scheduler is called. In MINIX, different processes have different priorities, to give better service to I/O device handlers than to user processes. If the disk process is now the highest priority runnable process, it will be scheduled to run. If the process that was interrupted is just as important or more so, then it will be scheduled to run again, and the disk process will have to wait a little while.

Either way, the C procedure called by the assembly language interrupt code now returns, and the assembly language code loads up the registers and memory map for the now-current process and starts it running. Interrupt handling and scheduling are summarized in Fig. 2-5. It is worth noting that the details vary slightly from system to system.

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

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

2.1.3 Threads

In a traditional process, of the type we have just studied, there is a single thread of control and a single program counter in each process. However, in some modern operating systems, support is provided for multiple threads of control within a process. These threads of control are usually just called **threads**, or occasionally **lightweight processes**.

In Fig. 2-6(a) we see three traditional processes. Each process has its own address space and a single thread of control. In contrast, in Fig. 2-6(b) we see a single process with three threads of control. Although in both cases we have three threads, in Fig. 2-6(a) each of them operates in a different address space, whereas in Fig. 2-6(b) all three of them share the same address space.

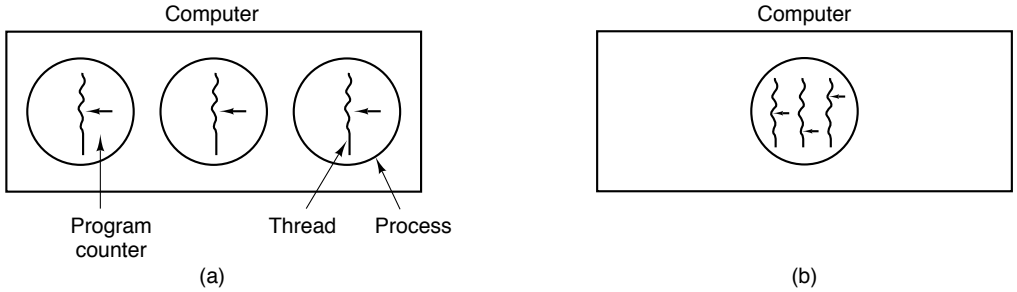


Figure 2-6. (a) Three processes each with one thread. (b) One process with three threads.

As an example of where multiple threads might be used, consider a file server process. It receives requests to read and write tiles and sends back the requested data or accepts updated data. To improve performance, the server maintains a cache of recently used files in memory, reading from the cache and writing to the cache when possible.

This situation lends itself well to the model of Fig. 2-6(b). When a request comes in, it is handed to a thread for processing. If that thread blocks part way through waiting for a disk transfer, other threads are still to able run, so the server can keep processing new requests even while disk I/O is taking place. The model of Fig. 2-6(a) is not suitable, because it is essential that all file sewer threads access the same cache, and the three threads of Fig. 2-6(a) do not share the same address space and thus cannot share the same memory cache.

Another example of where threads are useful is in browsers for the World Wide Web, such as Netscape and Mosaic. Many Web pages contain multiple small images. For each image on a Web page, the browser must set up a separate connection to the page's home site and request the image. A great deal of time is wasted establishing and releasing all these connections. By having multiple threads within the browser, many images can be requested at the same time, greatly speeding up performance in most cases, since with small images, the setup time is the limiting factor, not the speed of the transmission line.

When multiple threads are present in the same address space, a few of the fields of Fig. 2-4 are not per process, but per thread, so a separate thread table is needed, with one entry per thread. Among the per-thread items are the program counter, registers, and state. The program counter is needed because threads, like processes, can be suspended and resumed. The registers are needed because when threads are suspended, their registers must be saved. Finally., threads, like processes, can be in *running*, *ready*, or *blocked* state.

In some systems, the operating system is not aware of the threads. In other words, they are managed entirely in user space. When a thread is about to block, for example, it chooses and starts its successor before stopping. Several user-level threads packages are in common use, including the POSIX **P-threads** and Mach **C-threads** packages.

In other systems, the operating system is aware of the existence of multiple threads per process, so when a thread blocks, the operating system chooses the next one to run, either from the same process or a different one. To do scheduling, the kernel must have a thread table that lists all the threads in the system, analogous to the process table.

Although these two alternatives may seem equivalent, they differ considerably in performance. Switching threads is much faster when thread management is done in user space than when a kernel call is needed. This fact argues strongly for doing thread man-

agement in user space. On the other hand, when threads are managed entirely in user space and one thread blocks (e.g., waiting for I/O or a page fault to be handled), the kernel blocks the entire process, since it is not even aware that other threads exist. This fact argues strongly for doing thread management in the kernel. As a consequence, both systems are in use, and various hybrid schemes have been proposed as well (Anderson et al., 1992).

No matter whether threads are managed by the kernel or in user space, they introduce a raft of problems that must be solved and which change the programming model appreciably. To start with, consider the effects of the FORK system call. If the parent process has multiple threads, should the child also have them? If not, the process may not function properly, since all of them may be essential.

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

Another class of problems is related to the fact that threads share many data structures. What happens if one thread closes a file while another one is still reading from it? Suppose that one thread notices that there is too little memory and starts allocating more memory. Then, part way through, a thread switch occurs, and the new thread also notices that there is too little memory and also starts allocating more memory. Does the allocation happen once or twice? In nearly all systems that were not designed with threads in mind, the libraries (such as the memory allocation procedure) are not reentrant, and will crash if a second call is made while the first one is still active.

Another problem relates to error reporting. In UNIX, after a system call, the status of the call is put into a global variable, *errno*. What happens if a thread makes a system call, and before it is able to read *errno*, another thread makes a system call, wiping out the original value?

Next, consider signals. Some signals are logically thread specific, whereas others are not. For example, if a thread calls ALARM, it makes sense for the resulting signal to go to the thread that made the call. When the kernel is aware of threads, it can usually make sure the right thread gets the signal. When the kernel is not aware of threads, somehow the threads package must keep track of alarms. An additional complication for user-level threads exists when (as in UNIX) a process may only have one alarm at a time pending and several threads call ALARM independently.

Other signals, such as keyboard interrupt, are not thread specific. Who should catch them? One designated thread? All the threads? A newly created thread? All these solutions have problems. Furthermore, what happens if one thread changes the signal handlers without telling other threads?

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

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

and Marsh et al., 1991).

2.2 INTERPROCESS COMMUNICATION

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

Very briefly, there are three issues here. The first was alluded to above: how one process can pass information to another. The second has to do with making sure two or more processes do not get into each other's way when engaging in critical activities (suppose two processes each try to grab the last 100K of memory). The third concerns proper sequencing when dependencies are present: if process A produces data and process B prints it, B has to wait until A has produces some data before starting to print. We will examine all three of these issues starting in the next section.

2.2.1 Race Conditions

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

Imagine that our spooler directory has a large (potentially infinite) number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables, *out*, which points to the next file to be printed, and *in*, which points to the next free slot in the directory. These two variables might well be kept in a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files to be printed). More or less simultaneously, processes A and B decide they want to queue a file for printing. This situation is shown in 2-7.

In jurisdictions where Murphy's law[1] is applicable, the following might well happen. Process A reads *in* and stores the value, 7, in a local variable called *next_free_slot*. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B. Process B also reads *in*, and also gets a 7, so it stores the name of its file in slot 7 and updates *in* to be an 8. Then it goes off and does other things.

Eventually, process A runs again, starting from the place it left off last time. It looks at *next_free_slot*, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there. Then it computes *next_free_slot* + 1, which is 8, and sets *in* to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output. Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race conditions**. Debugging programs containing

[1] If something can go wrong, it will

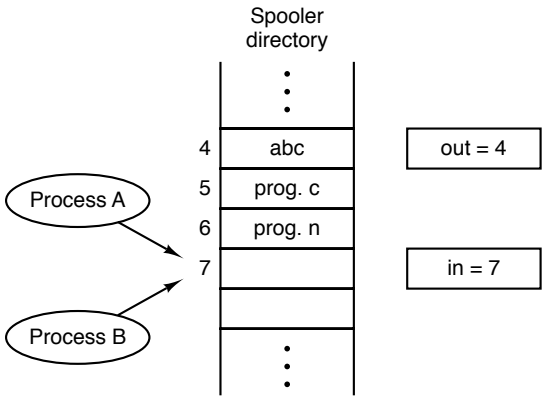


Figure 2-7. Two processes want to access shared memory at the same time .

race conditions is no fun at all. The results of most test runs are fine, but once in a weird while something weird and unexplained happens.

2.2.2 Critical Sections

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

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

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

1. No two processes may be simultaneously inside their critical regions.

2. No assumptions may be made about speeds or the number of CPUs.

3. No process running outside its critical region may block other processes.

4. No process should have to wait forever to enter its critical region.

2.2.3 Mutual Exclusion with Busy Waiting

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

Disabling Interrupts

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

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

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

Lock Variables

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

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

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

Strict Alternation

A third approach to the mutual exclusion problem is shown in Fig.**??**. This program fragment, like nearly all the others in this book, is written in C. C was chosen here because real operating systems are commonly written in C (or occasionally C++), but hardly ever in languages like Modula 2 or Pascal.

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

(a)

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

(b)

Figure 2-8. A proposed solution to the critical region problem.

In Fig.2-8, the integer variable *turn*, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects *turn*, finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing *turn* to see when it becomes 1. Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used.

When process 0 leaves the critical region, it sets *turn* to 1, to allow process 1 to enter its critical region. Suppose that process 1 finishes its critical region quickly, so both processes are in their noncritical regions, with *turn* set to 0. Now process 0 executes its whole loop quickly, coming back to its critical region and with *turn* set to 1. At this point, process 0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now, because *turn* is 1 and process 1 is busy with its noncritical region. Put differently, taking turns is not a good idea when one of the processes is much slower than the other.

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

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

Peterson's Solution

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

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

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

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

int turn;                             /* whose turn is it? */
int interested[N];                    /* all values initially 0 (FALSE)*/
void enter_region(int process)        /* process is 0 or 1 */
{
     int other;                       /* number of the other process */
     other = 1 - process;             /* the opposite of process */
     interested[process] = TRUE;      /* show that you are interested */
     turn = process;                  /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */;
}
void leave_region(int process)        /* process: who is leaving */
{
     interested[process] = FALSE;     /* indicate departure from critical region */
}
```

Figure 2-9. Peterson's solution for achieving mutual exclusion.

variables, the process calls *leave_region* to indicate that it is done and to allow the other process to enter, if it so desires.

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

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

The TSL Instruction

Now let us look at a proposal that requires a little help from the hardware. Many computers, especially those designed with multiple processors in mind, have an instruction TEST AND SET LOCK (TSL) that works as follows. it reads the contents of the memory word into a register and then stores a nonzero value at the memory address. The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

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

How can this instruction be used to prevent two processes from simultaneously en-

tering their critical regions? The solution is given in Fig. 2-10. There a four-instruction subroutine in a fictitious (but typical) assembly language is shown. The first instruction copies the old value of *lock* to the register and then sets *lock* to 1. Then the old value is compared with 0. If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again. Sooner or later it will become 0 (when the process currently in its critical region is done with its critical region), and the subroutine returns, with the lock set. Clearing the lock is simple. The program just stores a 0 in *lock*. No special instructions are needed.

```
enter_region:
        TSL REGISTER,LOCK       |copy LOCK to register and set LOCK to 1
        CMP REGISTER,#0         |was LOCK zero?
        JNE ENTER_REGION        |if it was non zero, LOCK was set, so loop
        RET                     |return to caller; critical region entered


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

Figure 2-10. Setting and clearing locks using TSL.

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

2.2.4 Sleep and Wakeup

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

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

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

The Producer-Consumer Problem

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

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

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

The consumer's code is similar: first test *count* to see if it is 0. If it is, go to sleep; if it is nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be sleeping, and if not, wakes it up. The code for both producer and consumer is shown in Fig. 2-11.

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

Now let us get back to the race condition. It can occur because access to *count* is unconstrained. The following situation could possibly occur. The buffer is empty and the consumer has just read *count* to see if it is 0. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. The producer enters an item in the buffer, increments *count*, and notices that it is now 1. Reasoning that *count* was just 0, and thus the consumer must be sleeping, the producer calls *wakeup* to wake the consumer up.

Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of *count* it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. If it were not lost, everything would work. A quick fix is to modify the rules to add a **wakeup waiting bit** to the picture. When a wakeup is sent to a process that is still awake, this bit is set. Later, when the process tries to go to sleep, if the wakeup waiting bit is on, it will be turned off, but the process will stay awake. The wakeup waiting bit is a piggy bank for wakeup signals.e

While the wakeup waiting bit saves the day in this simple example, it is easy to construct examples with three or more processes in which one wakeup waiting bit is insufficient. We could make another patch, and add a second wakeup waiting bit, or maybe 8 or 32 of them, but in principle the problem is still there.

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

void producer(void)
{
  int item;

  while (TRUE){                                 /* repeat forever */
    item = produce_item();                      /* generate next item */
    if (count == N) sleep();                    /* if buffer is full, go to sleep */
    insert_item(item);                          /* put item in buffer */
    count = count + 1;                          /* increment count of items in buffer */
    if (count == 1) wakeup(consumer);           /* was buffer empty? */
  }
}


void consumer(void)
{
  int item;

  while (TRUE){                                 /* repeat forever */
    if (count == 0) sleep();                    /* if buffer is empty, got to sleep */
    item = remove_item();                       /* take item out of buffer */
    count = count - 1;                          /* decrement count of items in buffer */
    if (count == N - 1) wakeup(producer);       /* was buffer full? */
    consume_item(item);                         /* print item */
  }
}
```

Figure 2-11. The producer-consumer problem with a fatal race condition.

2.2.5 Semaphores

This was the situation in 1965, when E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, called a **semaphore**, was introduced. A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.

Dijkstra proposed having two operations, DOWN and UP (generalizations of SLEEP and WAKEUP, respectively). The DOWN operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues. If the value is 0, the process is put to sleep without completing the DOWN for the moment. Checking the value, changing it, and possibly going to sleep is all done as a single, indivisible, **atomic action**. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions.

The UP operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier DOWN opera-

tion, one of them is chosen by the system (e.g., at random) and is allowed to complete its DOWN. Thus, after an UP on a semaphore with processes sleeping on it, the semaphore will still be 0, but there will be one fewer process sleeping on it. The operation of incrementing the semaphore and waking up one process is also indivisible. No process ever blocks doing an UP, just as no process ever blocks doing a WAKEUP in the earlier model.

As an aside, in Dijkstra's original paper, he used the names P and V instead of DOWN and UP, respectively, but since these have no mnemonic significance to people who do not speak Dutch (and only marginal significance to those who do), we will use the terms DOWN and UP instead. These were first introduced in Algol 68.

Solving the Producer-Consumer Problem using Semaphores

Semaphores solve the lost-wakeup problem, as shown in Fig. 2-12. It is essential that they be implemented in an indivisible way. The normal way is to implement UP and DOWN as system calls, with the operating system briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep, if necessary. As all of these actions take only a few instructions, no harm is done in disabling interrupts. If multiple CPUs are being used, each semaphore should be protected by a lock variable, with the TSL instruction used to make sure that only one CPU at a time examines the semaphore. Be sure you understand that using TSL to prevent several CPUs from accessing the semaphore at the same time is quite different from busy waiting by the producer or consumer waiting for the other to empty or till the buffer. The semaphore operation will only take a few microseconds, whereas the producer or consumer might take arbitrarily long.

This solution uses three semaphores: one called *full* for counting the number of slots that are full, one called *empty* for counting the number of slots that are empty, and one called *mutex* to make sure the producer and consumer do not access the buffer at the same time. *Full* is initially 0, *empty* is initially equal to the number of slots in the buffer, and *mutex* is initially 1. Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores**. If each process does a DOWN just before entering its critical region and an UP just after leaving it, mutual exclusion is guaranteed.

Now that we have a good interprocess communication primitive at our disposal, let us go back and look at the interrupt sequence of Fig. 2-5 again. In a system using semaphores, the natural way to hide interrupts is to have a semaphore, initially set to 0, associated with each I/O device. Just after starting an I/O device, the managing process does a DOWN on the associated semaphore, thus blocking immediately. When the interrupt comes in, the interrupt handler then does an UP on the associated semaphore, which makes the relevant process ready to run again. In this model, step 6 in Fig. 2-5 consists of doing an UP on the device's semaphore, so that in step 7 the scheduler will be able to run the device manager. Of course, if several processes are now ready, the scheduler may choose to run an even more important process next. We will look at how scheduling is done later in this chapter.

In the example of Fig. 2-12, we have actually used semaphores in two different ways. This difference is important enough to make explicit. The *mutex* semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables. This mutual exclusion is required to prevent chaos.

```
#define N 100                   /* number ot slots in the buffer */
typedef int semaphore;          /* semaphores are a special kind of int */
semaphore mutex = 1;            /* controls access to critical region */
semaphore empty = N;            /* counts empty buffer slots */
semaphore full = 0;             /* counts full buffer slots */

void producer(void)
{
    int item;
    while (TRUE) {              /* TRUE is the constant 1 */
        produce_item(&item);    /* generate something to put in buffer */
        down(&empty)2           /* decrement empty count */
        down(&mutex);           /* enter critical region */
        enter_item(item);       /* put new item in buffer */
        up(&mutex);             /* leave critical region */
        up(&full);              /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;
    while (TRUE) {              /* infinite loop */
        down(&full);            /* decrement full count */
        down(&mutex);           /* enter critical region */
        remove_item(&item);     /* take item fromybuffer */
        up(&mutex);             /* leave critical region */
        up(&empty);             /* increment count of empty slots */
        consume_item(item);     /* do something with the item */
    }
}
```

Figure 2-12. The producer-consumer problem using semaphores.

The other use of semaphores is for **synchronization**. The *full* and *empty* semaphores are needed to guarantee that certain event sequences do or do not occur. In this case, they ensure that the producer stops running when the buffer is full, and the consumer stops running when it is empty. This use is different from mutual exclusion.

Although semaphores have been around for more than a quarter of a century, people are still doing research about their use. As an example, see (Tai and Carver, 1996).

2.2.6 Monitors

With semaphores interprocess communication looks easy, right? Forget it. Look closely at the order of the DOWNs before entering or removing items from A the buffer in Fig. 2-12. Suppose that the two DOWNs in the producer's code were reversed in order, so *mutex* was decremented before *empty* instead of after it. If the buffer were completely full, the producer would block, with *mutex* set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a DOWN on *mutex*, now 0, and block too. Both processes would stay blocked forever and no more work would ever be done. This

unfortunate situation is called a **deadlock**. We will study deadlocks in detail in Chap. 3.

This problem is pointed out to show how careful you must be when using semaphores. One subtle error and everything comes to a grinding halt. It is like programming in assembly language, only worse, because the errors are race canditions, deadlocks, and other forms of unpredictable and irreproducible behavior.

To make it easier to write correct programs, Hoare (1974) and Brinch Hansen (1975) proposed a higher level synchronization primitive called a **monitor**. Their proposals differed slightly, as described below. A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor. Figure 2-13 illustrates a monitor written in an imaginary language, pidgin Pascal.

```
monitor example
  integer i;
  condition c;

  procedure producer(x);
  .
  .
  .
  end;

  procedure c:msumer(x);
  .
  .
  .
  end;
end monitor;
```

Figure 2-13. A monitor.

Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant. Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls. Typically, when a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

It is up to the compiler to implement the mutual exclusion on monitor entries, but a common way is to use a binary semaphore. Because the compiler, not the programmer, is arranging for the mutual exclusion, it is much less likely that something will go wrong. In any event, the person writing the monitor does not have to be aware of how the compiler arranges for mutual exclusion. It is sufficient to know that by turning all the critical regions into monitor procedures, no two processes will ever execute their critical regions at the same time.

Although monitors provide an easy way to achieve mutual exclusion, as we have seen above, that is not enough. We also need a way for processes to block when they cannot proceed. In the producer-consumer problem, it is easy enough to put all the tests for buffer-full and buffer-empty in monitor procedures, but how should the producer block when it finds the buffer full?

The solution lies in the introduction of **condition variables**, along with two operations on them, WAIT and SIGNAL. When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a WAIT on some condition variable, say, *full*. This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now.

This other process, for example, the consumer, can wake up its sleeping partner by doing a SIGNAL on the condition variable that its partner is waiting on. To avoid having two active processes in the monitor at the same time, we need a rule telling what happens after a SIGNAL. Hoare proposed letting the newly awakened process run, suspending the other one. Brinch Hansen proposed finessing the problem by requiring that a process doing a SIGNAL *must* exit the monitor immediately. In other words, a SIGNAL statement may appear only as the final statement in a monitor procedure. We will use Brinch Hansen's proposal because it is conceptually simpler and is also easier to implement. If a SIGNAL is done on a condition variable on which several processes are waiting, only one of them, A determined by the system scheduler, is revived.

Condition variables are not counters. They do not accumulate signals for later use the way semaphores do. Thus if a condition variable is signaled with no one waiting on it, the signal is lost. The WAIT must come before the SIGNAL. This rule makes the implementation much simpler. In practice it is not a problem because it is easy to keep track of the state of each process with variables, if need be. A process that might otherwise do a SIGNAL can see that this operation is not necessary by looking at the variables.

A skeleton of the producer-consumer problem with monitors is given in Fig. 2-14 in pidgin Pascal.

You may be thinking that the operations WAIT and SIGNAL look similar to SLEEP and WAKEUP, which we saw earlier had fatal race conditions. They *are* very similar, but with one crucial difference: SLEEP and WAKEUP failed because while one process was trying to go to sleep, the other one was trying to wake it up. With monitors, that cannot happen. The automatic mutual exclusion on monitor procedures guarantees that if, say, the producer inside a monitor procedure discovers that the buffer is full, it will be able to complete the WAIT operation without having to worry about the possibility that the scheduler may switch to the consumer just before the WAIT completes. The consumer will not even be let into the monitor at all until the WAIT is finished and the producer is marked as no longer runnable.

By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error-prone than with semaphores. Still, they too have some drawbacks. It is not for nothing that Fig. 2-14 is written in Pidgin Pascal rather than in C, as are the other examples in this book. As we said earlier, monitors are a programming language concept. The compiler must recognize them and arrange for the mutual exclusion somehow. C, Pascal, and most other languages do not have monitors, so it is unreasonable to expect their compilers to enforce any mutual exclusion rules. In fact, how could the compiler even know which procedures were in monitors and which were not?

These same languages do not have semaphores either, but adding semaphores is easy: all you need to do is add two short assembly code routines to the library to issue the UP and DOWN system calls. The compilers do not even have to know that they exist.

```
monitor ProducerConsumer
  condition full, empty;
  Integer count;

  procedure enter;
  begin
    if count = N then wait(full);
    enter_item;
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  procedure remove;
  begin
    if count = 0 then wait(empty);
    remove_item;
    count := count - 1;
    if count = N-1 then signal(full)
  end;

  count := 0;
end monitor;

procedure producer:
begin
  while true do
  begin
    produce_item;
    ProducerConsumer.enter
  end
end;

procedure consumer;
begin
  while true do
  begin
    ProducerConsurner.remove;
    consume_item
  end
end;
```

Figure 2-14. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has *N* slots.

Of course, the operating systems have to know about the semaphores, but at least if you have a semaphore-based operating system, you can still write the user programs for it in C or C++ (or even BASIC if you are masochistic enough). With monitors, you need a language that has them built in. A few languages, such as Concurrent Euclid(Hold, 1983) have them, but they are rare.

Another problem with monitors, and also with semaphores, is that they were designed for solving the mutual exclusion problem on one or more CPUs that all have access to a common memory. By putting the semaphores in the shared memory and protecting them with TSL instructions, we can avoid races. When we go to a distributed system consisting of multiple CPUs, each with its own private memory, connected by a local area network, these primitives become inapplicable. The conclusion is that semaphores are too low level and monitors are not usable except in a few programming languages. Also, none of the primitives provide for information exchange between machines. Something else is needed.

2.2.7 Message Passing

That something else is **message passing**. This method of interprocess communication uses two primitives SEND and RECEIVE, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as

```
send(destination, &message);
```

and

```
receive(source, &message);
```

The former call sends a message to a given destination and the latter once receives a message from a given source (or from ANY, if the receiver does not care). If no message is available, the receiver could block until one arrives. Alternatively, it could return immediately with an error code.

Design Issues for Message Passing Systems

Message passing systems have many challenging problems and design issues that do not arise with semaphores or monitors, especially if the communicating processes are on different machines connected by a network. For example, messages can be lost by the network. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special **acknowledgement** message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.

Now consider what happens if the message itself is received correctly, but the acknowledgement is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver can distinguish a new message from the retransmission of an old one. Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored.

Message systems also have to deal with the question of how processes are named, so that the process specified in a SEND or RECEIVE call is unambiguous. **Authentication**

is also an issue in message systems: how can the client tell that he is communicating with the real file server, and not with an imposter?

At the other end of the spectrum, there are also design issues that are important when the sender and receiver are on the same machine. One of these is performance. Copying messages from one process to another is always slower than doing a semaphore operation or entering a monitor, Much work has gone into making message passing efficient. Cheriton (1984), for example, has suggested limiting message size to what will fit in the machine's registers, and then doing message passing using the registers.

The Producer-Consumer Problem with Message Passing

Now let us see how the producer-consumer problem can be solved with message passing and no shared memory. A solution is given in Fig, 2-15. We assume that all messages are the same size and that messages sent but not yet received are buffered automatically by the operating system. In this solution, a total of N messages is used, analogous to the N slots in a shared memory buffer. The consumer starts out by sending N empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one. In this way, the total number of messages in the system remains constant in time, so they can be stored in a given amount of memory known in advance.

If the producer works faster than the consumer, all the messages will end up full, waiting for the consumer; the producer will be blocked, waiting for an empty to come back. If the consumer works faster, then the reverse happens: all the messages will be empties waiting for the producer to fill them up; the consumer will be blocked, waiting for a full message.

Many variants are possible with message passing. For starters, let us look at how messages are addressed. One way is to assign each process a unique address and have messages be addressed to processes. A different way is to invent a new data structure, called a **mailbox**. A mailbox is a place to buffer a certain number of messages, typically specified when the mailbox is created. When mailboxes are used, the address parameters in the SEND and RECEIVE calls are mailboxes, not processes. When a process tries to send to a mailbox that is full, it is suspended until a message is removed from that mailbox, making room for a new one.

For the producer-consumer problem, both the producer and consumer would create mailboxes large enough to hold N messages. The producer would send messages containing data to the consumer's mailbox, and the consumer would send empty messages to the producer's mailbox. When mailboxes are used, the buffering mechanism is clear: the destination mailbox holds messages that have been sent to the destination process but have not yet been accepted.

The other extreme from having mailboxes is to eliminate all buffering. When this approach is followed, if the SEND is done before the RECEIVE, the sending process is blocked until the RECEIVE happens, at which time the message can be copied directly from the sender to the receiver, with no intermediate buffering. Similarly, if the RECEIVE is done first, the receiver is blocked until a SEND happens. This strategy is often known as a **rendezvous**. It is easier to implement than a buffered message scheme but is less flexible since the sender and receiver are forced to run in lockstep.

The interprocess communication between user processes in MINIX (and UNIX) is via pipes, which are effectively mailboxes. The only real difference between a message system with mailboxes and the pipe mechanism is that pipes do not preserve message

```
#define N 100                                       /* number of slots in the buffer */

void producer(void)
{
  int item;
  message m;                                        /* message buffer */

  while (TRUE) {
    produce*item(&item);                            /* generate something to put in buffer */
    receive(consumer, &m);                          /* wait for an empty to arrive */
    build_message(&m, item);                        /* construct a message to send */
    send(consumer, &m);                             /* send item to consumer */
  }
}

void consumer(void)
{
  int item, i; i
  message mg

  for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
  while (TRUE) {
    receive(producer, &m);                          /* get message containing item */
    extract_item(&m, &item);                        /* extract item from message */
    send(producer, &m);                             /* send back empty reply */
    consume_item(item);                             /* do something with the item */
  }
}
```

Figure 2-15.

boundaries. In other words, if one process writes 10 messages of 100 bytes to a pipe and another process reads 1000 bytes from that pipe, the reader will get all 10 messages at once. With a true message system, each READ should return only one message. Of course, if the processes agree always to read and write fixed-size messages from the pipe, or to end each message with a special character (e.g., linefeed), no problems arise. The processes that make up the MINIX operating system itself use a true message scheme with fixed size messages for communication among themselves.

2.3 CLASSICAL IPC PROBLEMS

The operating system literature is full of interesting problems that have been widely discussed and analyzed. In the following sections we will examine three of the better-known problems.

2.3.1 The Dining Philosophers Problem

In 1965, Dijkstra posed and solved a synchronization problem he called the **dining philosophers problem**. Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new primitive is by showing how elegantly it solves the dining philosophers problem. The problem can be stated quite simply as follows. Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The layout of the table is illustrated in Fig. 2-16.

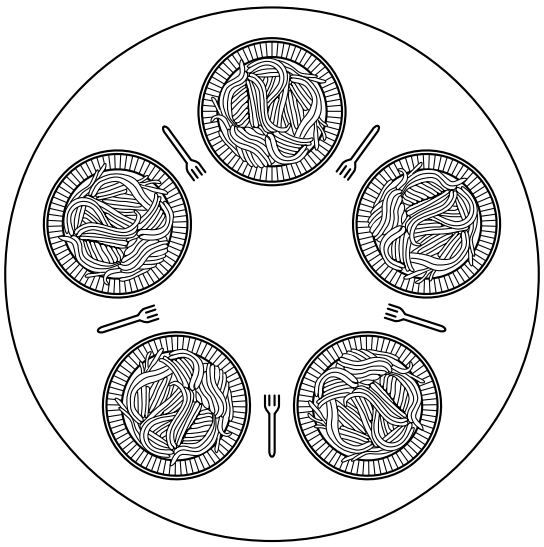


Figure 2-16. Lunch time in the Philosophy Department.

The life of a philosopher consists of alternate periods of eating and thinking. (This is something of an abstraction. even for philosophers, but the other activities are irrelevant here.) When a philosopher gets hungry. she tries to acquire her left and right fork, one at a time, in either order. If successful in acquiring two forks. she eats for a while. then puts down the forks and continues to think. The key question is: Can you write a program for each philosopher that does what it is supposed to do and never gets stuck? (It has been pointed out that the two-fork requirement is somewhat artificial; perhaps we should switch from Italian to Chinese food. substituting rice for spaghetti and chopsticks for forks.)

Figure 2-17 shows the obvious solution. The procedure *take_fork* waits until the specified fork is available and then seizes it. Unfortunately. the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks. and there will be a deadlock.

We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not. the philosopher puts down the left one. waits for some time. and then repeats the whole process. This proposal too. fails. although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously. picking up their left forks. seeing that their right forks were not available, putting down their left forks, waiting picking up their left forks again simultaneously, and so on. forever. A situation like this in which all the programs continue to run indefinitely

```
#define N 5                     /* number of philosophers */
void philosopher(int i)         /* i: philosopher number, from 0 to 4 */
{
  while (TRUE) {
    think();                    /* philosopher is thinking */
    take_fork(i);               /* take left tent */
    take_fork((i+1) % N);       /* take might fork; % is modulo operator */
    eat();                      /* yum-yum, spaghetti */
    put_fork(i);                /* put left fork back on the table */
    put_fork((i+1) % N);        /* put right fork back on the table */
  }
}
```

Figure 2-17. A nonsolution to the dining philosophers problem.

but fail to make any progress is called **starvation**. (It is called starvation even when the problem does not occur in an Italian or a Chinese restaurant.)

Now you might think, "If the philosophers would just wait a random time instead of the same time after failing to acquire the right-hand fork, the chance that everything would continue in lockstep for even an hour is very small." This observation is true, but in some applications one would prefer a solution that always works and cannot fail due to an unlikely series of random numbers. (Think about safety control in a nuclear power plant.)

One improvement to Fig. 2-17 that has no deadlock and no starvation is to protect the five statements following the call to think by a binary semaphore. Before starting to acquire forks, a philosopher would do a DOWN on *mutex*. After replacing the forks, she would do an UP on *mutex*. From a theoretical viewpoint, this solution is adequate. From a practical one, it has a performance bug: only one philosopher can be eating at any instant. With five forks available, we should be able to allow two philosophers to eat at the same time.

The solution presented in Fig. 2-18 is correct and also allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, *state*, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move only into eating state if neither neighbor is eating. Philosopher *i*'s neighbors are defined by the macros *LEFT* and *RIGHT*. In other words, if *i* is 2, *LEFT* is 1 and *RIGHT* is 3.

The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. Note that each process runs the procedure philosopher as its main code, but the other procedures, *take_forks*, *put_forks*, and test are ordinary procedures and not separate processes.

2.3.2 The Readers and Writers Problem

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem (Courtois et al., 1971), which models

```
#define N 5                     /* number of philosophers */
#define LEFT (i-1)%N            /* number of i's left neighbor */
#define RIGHT (i+1)%N           /* number of i's right neighbor */
ftdefine THINKING 0             /* philosopher is thinking */
#define HUNGRY 1                /* philosopher is trying to get forks */
#define EATING 2                /* philosopher is eating */

typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */

void philosopher(int i)         /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {              /* repeat forever */
    think();                    /* philosopher is thinking */
    take,forks(i);              /* acquire two forks or block */
    eat();                      /* yum-yum, spaghetti */
    put_forks(i);               /* put both forks back on table */
    }
}

void take_forks(int i)          /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);               /* enter critical region */
    state[i} = HUNGRY;          /* record fact that philosopher i is hungry */
    test(i);                    /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);                /* block if forks were not acquired */
}

void put_forks(i)               /* i: philosopher number, from 0 to N-1 */
{
    down(8=mutex);              /* enter critical region */
    state[i} = THINKING;        /* philosopher has finished eating */
    test(LEFT);                 /* see if left neighbor can now eat */
    test(RlGHT);                /* see if right neighbor can now eat */
    up(&mutex);                 /* exit critical region */
}

void testfi)                    /* i: philosopher number, from 0 to N-1 */
{
    if (state[i} == HUNGRY && state[LEFT] l= EATING && stale[RIGHTJ l= EATING) {
        state[i} = EATING;
        up(8=$[iI);
    }
}
```

Figure 2-18. A solution to the dinning philosopher's problem.

access to a data base. Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the data base at the same time. but if one process is updating (writing) the data base, no other processes may have i access to the data base, not even readers. The question is how do you program the readers and the writers? One solution is shown in Fig. 2-19.

```
typedef int semaphore;        /* use your imagination */
semaphore mutex = 1;          /* controls access to 'rc' */
semaphore db = 1;             /* controls access to the data base */
int rc = 0;                   /* # of processes reading or wanting to */

void reader(void)
{
  while (TRUE) {              /* repeat forever */
    down(&mutex);             /* get exclusive access to 'rc' */
    rc = rc + 1;              /* one reader more now */
    if (rc == 1) down{&db);   /* if this is the first reader */
    up(&mutex);               /* release exclusive access to 'rc' */
    read_data_base();         /* access the data */
    down(&mutex);             /* get exclusive access to 'rc' */
    rc = rc - 1;              /* one reader fewer now */
    if (rc == 0) up(&db);     /* if this is the last reader */
    up(&mutex);               /* release exclusive access to 'rc' */
    use_data_read();          /* noncritical region */
  }
}

void writer(void)
{
  while (TRUE) {              /* repeat forever */
    think_up_data();          /* noncritical region */
    down(&db);                /* get exclusive access */
    write_data_base();        /* update the data */
    up(&db};                  /* release exclusive access */
  }
}
```

Figure 2-19. A solution to the readers and writers problem.

In this solution, the first reader to get access to the data base does a DOWN on the semaphore *db*. Subsequent readers merely increment a counter, *rc*. As readers leave, they decrement the counter and the last one out does an UP on the semaphore, allowing a blocked writer, if there is one, to get in.

The solution presented here implicitly contains a subtle decision that is worth commenting on. Suppose that while a reader is using the data base, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. A third and subsequent readers can also be admitted if they come along.

Now suppose that a writer comes along. The writer cannot be admitted to the data

base, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 seconds, and each reader takes 5 seconds to do its work, the writer will never get in.

To prevent this situation, the program could be written slightly differently: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance. Courtois et al. present a solution that gives priority to writers. For details, we refer you to the paper.

2.3.3 The Sleeping Barber Problem

Another classical IPC problem takes place in a barber shop. The barber shop has one barber, one barber chair, and n chairs for waiting customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and falls asleep, as illustrated in Fig. 2-20. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full). The problem is to program the barber and the customers without getting into race conditions.

Figure 2-20. The sleeping barber.

Our solution uses three semaphores: *customers*, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), *barbers*, the number of barbers who are idle, waiting for customers (0 or 1), and *mutex*, which is used for mutual exclusion. We also need a variable, *waiting*, which also counts the waiting customers. It is essentially a copy of *customers*. The reason for having *waiting* is that there is no way to read the current value of a semaphore. in this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.

Our solution is shown in Fig. 2-21. When the barber shows up for work in the morning, he executes the procedure *barber*, causing him to block on the semaphore *customers* until somebody arrives. He then goes to sleep as shown in Fig. 2-20.

When a customer arrives, he executes *customer*, starting by acquiring *mutex* to enter a critical region. If another customer enters shortly thereafter, the second one will not be able to do anything until the first one has released *mutex*. The customer then checks to see if the number of waiting customers is less than the number of chairs. If not, he releases *mutex* and leaves without a haircut.

If there is an available chair, the customer increments the integer variable, *waiting*. Then he does an UP on the semaphore *customers*, thus waking up the barber. At this point, the customer and barber are both awake. When the customer releases *mutex*, the barber grabs it, does some housekeeping, and begins the haircut.

When the haircut is over, the customer exits the procedure and leaves the shop. Unlike our earlier examples, there is no loop for the customer because each one gets only one haircut. The barber loops, however, to try to get the next customer. If one is present, another haircut is given. If not, the barber goes to sleep.

As an aside, it is worth pointing out that although the readers and writers and sleeping barber problems do not involve data transfer, they are still belong to the area of IPC because they involve synchronization between multiple processes.

2.4 PROCESS SCHEDULING

In the examples of the previous sections, we have often had situations in which two or more processes (e.g., producer and consumer) were logically runnable. When more than one process is runnable, the operating system must decide which one to run first. The part of the operating system that makes this decision is called the **scheduler**; the algorithm it uses is called the **scheduling algorithm**.

Back in the old days of batch systems with input in the form of card images on a magnetic tape, the scheduling algorithm was simple: just run the next job on the tape. With timesharing systems, the scheduling algorithm is more complex, as there are often multiple users waiting for service, and there may be one or more batch streams as well (e.g., at an insurance company, for processing claims). Even on personal computers, there may be several user-initiated processes competing for the CPU, not to mention background jobs, such as network or electronic mail daemons sending or receiving e-mail.

Before looking at specific scheduling algorithms, we should think about what the scheduler is trying to achieve. After all, the scheduler is concerned with deciding on policy, not providing a mechanism. Various criteria come to mind as to what constitutes a good scheduling algorithm. Some of the possibilities include:

1. Fairness—make sure each process gets its fair share of the CPU.

```
#define CHAIRS 5                /* # chairs for waiting customers */
typedef int semaphore;          /* use your imagination */
semaphore customers = 0;        /* # of customers waiting for service */
semaphore barbers = 0;          /* # of barbers waiting for customers */
semaphore mutex = 1;            /* for mutual exclusion */
int waiting = 0;                /* customers are waiting (not being cut) */

void barber(void)
{
  while (TRUE) {
    down(customers);            /* go to sleep if # of customers is 0 */
    down(mutex);                /* acquire access to 'waiting' */
    waiting = waiting - 1;      /* decrement count of waiting customers */
    up(barbers);                /* one barber is now ready to cut hair */
    up(mutex);                  /* release 'waiting' */
    cut_hair();                 /* cut hair (outside critical region) */
  }
}

void customer(void)
{
  down(mutex);                  /* enter critical region */
  it (waiting < CHAIRS) {       /* if there are no tree chairs, leave */
    waiting = waiting + 1;      /* increment count of waiting customers */
    up(customers);              /* wake up barber if necessary */
    up(mutex);                  /* release access to 'waiting' */
    down(barbers);              /* go to sleep it it of free barbers is 0 */
    get_haircut();              /* be seated and be serviced */
  } else {
    up(mutex);                  /* shop is full; do not wait */
  }
}
```

Figure 2-21. A solution to the sleeping barber problem.

2. Efficiency—keep the CPU busy 100 percent of the time.
3. Response time—minimize response time for interactive users.
4. Turnaround—minimize the time batch users must wait for output.
5. Throughput—maximize the number of jobs processed per hour.

A little thought will show that some of these goals are contradictory. To minimize response time for interactive users, the scheduler should not run any batch jobs at all (except maybe between 3 AM. and 6 A.M., when all the interactive users are snug in their beds). The batch users probably will not like this algorithm, however; it violates criterion 4. It can be shown (Kleinrock, 1975) that any scheduling algorithm that favors some class of jobs hurts another class of jobs. The amount of CPU time available is finite, after all. To give one user more you have to give another user less. Such is life.

A complication that schedulers have to deal with is that every process is unique and unpredictable. Some spend a lot of time waiting for file I/O, while others would use the CPU for hours at a time if given the chance. When the scheduler starts running some process, it never knows for sure how long it will be until that process blocks, either for I/O, or on a semaphore, or for some other reason. To make sure that no process runs too long, nearly all computers have an electronic timer or clock built in, which causes an interrupt periodically. A frequency of 50 or 60 times a second (called 50 or 60 **Hertz** and abbreviated **Hz**) is common, but on many computers the operating system can set the timer frequency to anything it wants. At each clock interrupt, the operating system gets to run and decide whether the currently running process should be allowed to continue, or whether it has had enough CPU time for the moment and should be suspended to I give another process the CPU.

The strategy of allowing processes that are logically runnable to be temporarily suspended is called **preemptive scheduling**, and is in contrast to the **run to completion** method of the early batch systems. Run to completion is also called **non-preemptive scheduling**. As we have seen throughout this chapter, a process can be suspended at an arbitrary instant, without warning, so another process can be run. This leads to race conditions and necessitates semaphores, monitors, messages, or some other sophisticated method for preventing them. On the other hand, a policy of letting a process run as long as it wanted to would mean that some process computing π to a billion places could deny service to all other processes indefinitely.

Thus although non-preemptive scheduling algorithms are simple and easy to implement, they are usually not suitable for general-purpose systems with multiple competing users. On the other hand, for a dedicated system, such as a data base server, it may well be reasonable for the master process to start a child process working on a request and let it run until it completes or blocks. The difference from the general-purpose system is that all processes in the data base system are under the control of a single master, which knows what each child is going to do and about how long it will take.

2.4.1 Round Robin Scheduling

Now let us look at some specific scheduling algorithms. One of the oldest, simplest, fairest, and most widely used algorithms is **round robin**. Each process is assigned a time interval, called its **quantum**, which it is allowed to run. If the process is still running at the end of the quantum, the CPU is preempted and given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course. Round robin is easy to implement. All the scheduler needs to do is maintain a list of runnable processes, as shown in Fig. 2-22(a). When the process uses up its quantum, it is put on the end of the list, as shown in Fig. 2-22(b).

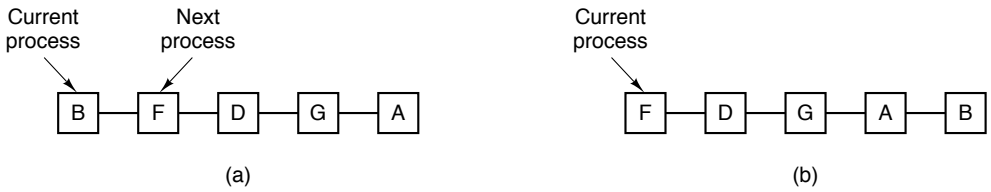


Figure 2-22. Round robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after B uses up its quantum.

The only interesting issue with round robin is the length of the quantum. Switching from one process to another requires a certain amount of time for doing the administration-saving and loading registers and memory maps, updating various tables and lists, etc. Suppose that this **process switch** or **context switch**, as it is sometimes called, takes 5 msec. Also suppose that the quantum is set at 20 msec. With these parameters, after doing 20 msec of useful work, the CPU will have to spend 5 msec on process switching. Twenty percent of the CPU time will be wasted on administrative overhead.

To improve the CPU efficiency, we could set the quantum to, say, 500 msec. Now the wasted time is less than l percent. But consider what happens on a timesharing system if ten interactive users hit the carriage return key at roughly the same time. Ten processes will be put on the list of runnable processes. If the CPU is idle, the first one will start immediately, the second one may not start until about l/2 sec later, and so on. The unlucky last one may have to wait 5 sec before getting a chance, assuming all the others use their full quanta. Most users will perceive a 5-sec response to a short command as terrible. The same problem can occur on a personal computer that supports multiprogramming.

The conclusion can be formulated as follows: setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests. A quantum around 100 msec is often a reasonable compromise.

2.4.2 Priority Scheduling

Round robin scheduling makes the implicit assumption that all processes are equally important. Frequently, the people who own and operate multiuser computers have different ideas on that subject. At a university, the pecking order may be deans first, then professors, secretaries, janitors, and finally students. The need to take external factors into account leads to **priority scheduling**. The basic idea is straightforward: each process is assigned a priority, and the runnable process with the highest priority is allowed to run.

Even on a PC with a single owner, there may be multiple processes, some more important than others. For example, a daemon process sending electronic mail in the background should be assigned a lower priority than a process displaying a video film on the screen in real time.

To prevent high-priority processes from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick (i.e., at each clock interrupt). If this action causes its priority to drop below that of the next highest process, a process switch occurs. Alternatively, each process may be assigned a maximum quantum that it is allowed to hold the CPU continuously. When this quantum is used up, the next highest priority process is given a chance to run.

Priorities can be assigned to processes statically or dynamically. On a military computer, processes started by generals might begin at priority 100, processes started by colonels at 90, majors at 80, captains at 70, lieutenants at 60, and so on, Alternatively, at a commercial computer center, high-priority jobs might cost 100 dollars an hour, medium priority 75 dollars an hour, and low priority 50 dollars an hour. The UNIX system has a command, *nice*, which allows a user to voluntarily reduce the priority of his process, in order to be nice to the other users. Nobody ever uses it.

Priorities can also be assigned dynamically by the system to achieve certain system goals. For example, some processes are highly I/O bound and spend most of their time waiting for I/O to complete. Whenever such a process wants the CPU, it should be given the CPU immediately, to let it start its next I/O request, which can then proceed in parallel

with another process actually computing. Making the I/O bound process wait a long time for the CPU will just mean having it around occupying memory for an unnecessarily long time. A simple algorithm for giving good service to I/O bound processes is to set the priority to $1/f$ where f is the fraction of the last quantum that a process used. A process that used only 2 msec of its 100 msec quantum would get priority 50, while a process that ran 50 msec before blocking would get priority 2, and a process that used the whole quantum would get priority 1.

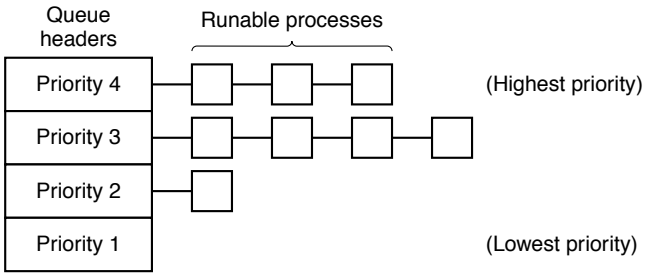


Figure 2-23. A scheduling algorithm with four priority classes.

It is often convenient to group processes into priority classes and use priority scheduling among the classes but round-robin scheduling within each class. Figure 2-23 shows a system with four priority classes. The scheduling algorithm is as follows: as long as there are runnable processes in priority class 4, just run each one for one quantum, round-robin fashion, and never bother with lower priority classes. If priority class 4 is empty, then run the class 3 processes round robin. If classes 4 and 3 are both empty, then run class 2 round robin, and so on. If priorities are not adjusted occasionally, lower priority classes may all starve to death.

2.4.3 Multiple Queues

One of the earliest priority schedulers was in CTSS (Corbato et al., 1962). CTSS had the problem that process switching was very slow because the 7094 could hold only one process in memory. Each switch meant swapping the current process to disk and reading in a new one from disk. The CTSS designers quickly realized that it was more efficient to give CPU-bound processes a large quantum once in a while, rather than giving them small quanta frequently (to reduce swapping). On the other hand, giving all processes a large quantum would mean poor response time, as we have already seen. Their solution was to set up priority classes. Processes in the highest class were run for one quantum. Processes in the next highest class were run for two quanta. Processes in the next class were run for four quanta, and so on. Whenever a process used up all the quanta allocated to it, it was moved down one class.

As an example, consider a process that needed to compute continuously for 100 quanta. It would initially be given one quantum, then swapped out. Next time it would get two quanta before being swapped out. On succeeding runs it would get 4, 8, 16, 32, and 64 quanta, although it would have used only 37 of the final 64 quanta to complete its work. Only 7 swaps would be needed (including the initial load) instead of 100 with a pure round-robin algorithm. Furthermore, as the process sank deeper and deeper into the priority queues, it would be run less and less frequently, saving the CPU for short, interactive processes.

The following policy was adopted to prevent a process that needed to run for a long time when it first started but became interactive later, from being punished forever. Whenever a carriage return was typed at a terminal, the process belonging to that terminal was moved to the highest priority class, on the assumption that it was about to become interactive. One fine day some user with a heavily CPU-bound process discovered that just sitting at the terminal and typing carriage returns at random every few seconds did wonders for his response time. He told all his friends. Moral of the story: getting it right in practice is much harder than getting it right in principle.

Many other algorithms have been used for assigning processes to priority classes. For example, the influential XDS 940 system (Lampson, 1968), built at Berkeley, had four priority classes. called terminal, I/0, short quantum, and long quantum. When a process that was waiting for terminal input was finally awakened, it went into the highest priority class (terminal). When a process waiting for a disk block became ready, it went into the second class. When a process was still running when its quantum ran out, it was initially placed in the third class. However, if a process used up its quantum too many times in a row without blocking for terminal or other I/O, it was moved down to the bottom queue. Many other systems use something similar to favor interactive users and processes over background ones.

2.4.4 Shortest Job First

Most of the above algorithms were designed for interactive systems. Now let us look at one that is especially appropriate for batch jobs for which the run times are known in advance. In an insurance company, for example, people can predict quite accurately how long it will take to run a batch of 1000 claims, since similar work is done every day. When several equally important jobs are sitting in the input queue waiting to be started, the scheduler should use **shortest job first**. Look at Fig. 2-24. Here we find four jobs *A*, *B*, *C*, and *D* with run times of 8, 4, 4, and 4 minutes, respectively. By running them in that order, the turnaround time for *A* is 8 minutes, for *B* is 12 minutes, for *C* is 16 minutes, and for *D* is 20 minutes for an average of 14 minutes.

Figure 2-24. An example of shortest job first scheduling.

Now let us consider running these four jobs using shortest job first, as shown in Fig. 2-24(b). The turnaround times are now 4, 8, 12, and 20 minutes for an average of ll minutes. Shortest job first is provably optimal. Consider the case of four jobs, with run times of a, b, c, and d, respectively. The first job finishes at time a, the second finishes at time $a+b$, and so on. The mean turnaround time is $(4a+3b+2c+d)/4$. It is clear that a contributes more to the average than the other times, so it should be the shortest job, with b next, then c and finally d as the longest as it affects only its own turnaround time. The same argument applies equally well to any number of jobs.

Because shortest job first always produces the minimum average response time, it would be nice if it could be used for interactive processes as well. To a certain extent, it

can be. Interactive processes generally follow the pattern of wait for command, execute command, wait for command, execute command, and so on. If we regard the execution of each command as a separate "job," then we could minimize overall response time by running the shortest one first. The only problem is figuring out which of the currently runnable processes is the shortest one.

One approach is to make estimates based on past behavior and run the process with the shortest estimated running time. Suppose that the estimated time per command for some terminal is T_0. Now suppose its next run is measured to be T_1. We could update our estimate by taking a weighted sum of these two numbers, that is, $aT0 + (1-a)T1$. Through the choice of a we can decide to have the estimation process forget old runs quickly, or remember them for a long time. With $a = 1/2$, we get successive estimates of

$$T_0, T_0/2 + T_1/2, T_0/4 + T_1/4 + t_2/2, T_0/8 + T_1/8 + T_2/4 + T_3/2 \tag{2.1}$$

After three new runs, the weight of T_0 in the new estimate has dropped to $1/8$.

The technique of estimating the next value in a series by taking the weighted average of the current measured value and the previous estimate is sometimes called **aging**. It is applicable to many situations where a prediction must be made based on previous values. Aging is especially easy to implement when $a = 1/2$. All that is needed is to add the new value to the current estimate and divide the sum by 2 (by shifting it right 1 bit).

It is worth pointing out that the shortest job first algorithm is only optimal when all the jobs are available simultaneously. As a counterexample, consider five jobs, *A* through *E*, with run times of 2, 4, 1, 1, and 1, respectively. Their arrival times are 0, 0, 3, 3, and 3.

Initially, only *A* or *B* can be chosen, since the other three jobs have not arrived yet. Using shortest job first we will run the jobs in the order *A*, *B*, *C*, *D*, *E*, for an average wait of 4.6. However, running them in the order *B*, *C*, *D*, *E*, *A* has an average wait of 4.4.

2.4.5 Guaranteed Scheduling

A completely different approach to scheduling is to make real promises to the user about performance and then live up to them. One promise that is realistic to make and easy to live up to is this: If there are n users logged in while you are working, you will receive about $1/n$ of the CPU power. Similarly, on a single-user system with n processes running, all things being equal, each one should get $1/n$ of the CPU cycles.

To make good on this promise, the system must keep track of how much CPU each process has had since its creation. It then computes the amount of CPU each one is entitled to, namely the time since creation divided by n. Since the amount of CPU time each process has actually had is also known, it is straightforward to compute the ratio of actual CPU had to CPU time entitled. A ratio of 0.5 means that a process has only had half of what it should have had, and a ratio of 2.0 means that a process has had twice as much as it was entitled to. The algorithm is then to run the process with the lowest ratio until its ratio has moved above its closest competitor.

2.4.6 Lottery Scheduling

While making promises to the users and then living up to them is a fine idea, it is difficult to implement. However, another algorithm can be used to give similarly predictable results with a much simpler implementation. It is called **lottery scheduling** (Waldspurger and Weihl, 1994).

The basic idea is to give processes lottery tickets for various system resources, such as CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource. When applied to CPU scheduling, the system might hold a lottery 50 times a second, with each winner getting 20 msec of CPU time as a prize.

To paraphrase George Orwell: "All processes are equal, but some processes are more equal." More important processes can be given extra tickets, to increase their odds of winning. If there are 100 tickets outstanding, and one process holds 20 of them, it will have a 20 percent chance of winning each lottery. In the long run, it will get about 20 percent of the CPU. In contrast to a priority scheduler, where it is very hard to state what having a priority of 40 actually means, here the rule is clear: a process holding a fraction f of the tickets will get about a fraction f of the resource in question.

Lottery scheduling has several interesting properties. For example, if a new process shows up and is granted some tickets, at the very next lottery it will have a chance of winning in proportion to the number of tickets it holds. In other words, lottery scheduling is highly responsive.

Cooperating processes may exchange tickets if they wish. For example, when a client process sends a message to a server process and then blocks, it may give all of its tickets to the server, to increase the chance of the server running next. When the server is finished, it returns the tickets so the client can run again. In fact, in the absence of clients, servers need no tickets at all.

Lottery scheduling can be used to solve problems that are difficult to handle with other methods. One example is a video server in which several processes are feeding video streams to their clients, but at different frame rates. Suppose that the processes need frames at 10, 20, and 25 frames/sec. By allocating these processes 10, 20, and 25 tickets, respectively, they will automatically divide the CPU in the correct proportion.

2.4.7 Real-Time Scheduling

A **real-time** system is one in which time plays an essential role. Typically, one or more physical devices external to the computer generate stimuli, and the computer must react appropriately to them within a fixed amount of time. For example, the computer in a compact disc player gets the bits as they come off the drive and must convert them into music within a very tight time interval. If the calculation takes too long, the music will sound peculiar. Other real-time systems are patient monitoring in a hospital intensive-care unit, the autopilot in an aircraft, and safety control in a nuclear reactor. In all these cases, having the right answer but having it too late is often just as bad as not having it at all.

Real-time systems are generally categorized as **hard real time**, meaning there are absolute deadlines that must be met, or else, and **soft real time**, meaning that missing an occasional deadline is tolerable. In both cases, real-time behavior is achieved by dividing the program into a number of processes, each of whose behavior is predictable and known in advance. These processes are generally short lived and can run to completion in under a second. When an external event is detected, it is the job of the scheduler to schedule the processes in such a way as that all deadlines are met.

The events that a real-time system may have to respond to can be further categorized as **periodic** (occurring at regular intervals) or **aperiodic** (occurring unpredictably). A system may have to respond to multiple periodic event streams. Depending on how much time each event requires for processing, it may not even be possible to handle them all. For example, if there are m periodic events and event i occurs with period P_i and requires

C_i seconds of CPU time to handle each event, then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1 \tag{2.2}$$

A real-time system that meets this criteria is said to be **schedulable**.

As an example, consider a soft real-time system with three periodic events, with periods of l00, 200, and 500 msec, respectively. If these events require 50, 30, and 100 msec of CPU time per event, respectively, the system is schedulable because $0.5 + 0.15 + 0.2 < 1$. If a fourth event with a period of 1 sec is added, the system will remain schedulable as long as this event does not need more than 150 msec of CPU time per event. Implicit in this calculation is the assumption that the context-switching overhead is so small that it can be ignored.

Real-time scheduling algorithms can be dynamic or static. The former make their scheduling decisions at run time; the latter make them before the system starts running. Let us briefly consider a few of the dynamic real-time scheduling algorithms. The classic algorithm is the **rate monotonic algorithm** (Liu and Layland, 1973). In advance, it assigns to each process a priority proportional to the frequency of occurrence of its triggering event. For example, a process to run every 20 msec gets priority 50 and a process to run every 100 msec gets priority 10. At run time, the scheduler always runs the highest priority ready process, preempting the running process if need be. Liu and Layland proved that this algorithm is optimal.

Another popular real-time scheduling algorithm is **earliest deadline first**. Whenever an event is detected, its process is added to the list of ready processes. The list is kept sorted by deadline, which for a periodic event is the next occurrence of the event. The algorithm runs the first process on the list, the one with the closest deadline.

A third algorithm first computes for each process the amount of time it has to spare, called its **laxity**. If a process requires 200 msec and must be finished in 250 msec, its laxity is 50 msec. The algorithm, called **least laxity**, chooses the process with the smallest amount of time to spare.

While in theory it is possible to turn a general-purpose operating system into a real-time system by using one of these scheduling algorithms, in practice the context-switching overhead of general-purpose systems is so large that real-time performance can only be achieved for applications with easy time constraints. As a consequence, most real-time work uses special real-time operating systems that have certain important properties. Typically these include a small size, fast interrupt time, rapid context switch, short interval during which interrupts are disabled, and the ability to manage multiple timers in the millisecond or microsecond range.

2.4.8 Two-level Scheduling

Up until now we have more or less assumed that all runnable processes are in main memory. If insufficient main memory is available, some of the runnable processes will have to be kept on the disk, in whole or in part. This situation has major implications for scheduling, since the process switching time to bring in and run a process from disk is orders of magnitude more than switching to a process already in main memory.

A more practical way of dealing with swapped out processes is to use a two-level scheduler. Some subset of the runnable processes is first loaded into main memory, as shown in Fig. 2-25(a). The scheduler then restricts itself to only choosing processes from

this subset for a while. Periodically, a higher-level scheduler is invoked to remove processes that have been in memory long enough and to load processes that have been on disk too long. Once the change has been made, as in Fig. 2-25(b), the lower-level scheduler again restricts itself to only running processes that are actually in memory. Thus, the lower-level scheduler is concerned with making a choice among the runnable processes that are in memory at that moment, while the higher-level scheduler is concerned with shuttling processes back and forth between memory and disk.

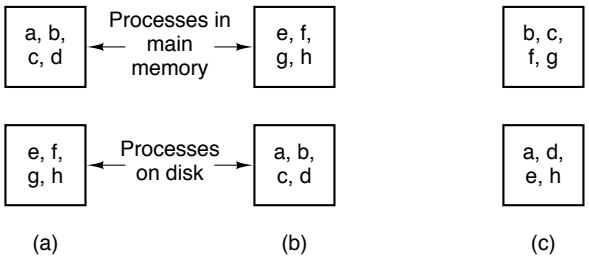


Figure 2-25. A two-level scheduler must move processes between disk and memory and also choose processes to run from among those in memory. Three different instants of time are represented by (a), (b), and (c).

Among the criteria that the higher-level scheduler could use to make its decisions are the following ones:

1. How long has it been since the process was swapped in or out?
2. How much CPU time has the process had recently?
3. How big is the process? (Small ones do not get in the way.)
4. How high is the priority of the process?

Again here we could use round-robin, priority scheduling, or any of various other methods. The two schedulers may or may not use the same algorithm.

2.4.9 Policy versus Mechanism

Up until now, we have tacitly assumed that all the processes in the system belong to different users and are thus competing for the CPU. While this is often true, sometimes it happens that one process has many children running under its control. For example, a data base management system process may have many children. Each child might be working on a different request, or each one might have some specific function to perform (query parsing, disk access, etc.). It is entirely possible that the main process has an excellent idea of which of its children are the most important (or time critical) and which the least. Unfortunately, none of the schedulers discussed above accept any input from user processes about scheduling decisions. As a result, the scheduler rarely makes the best choice.

The solution to this problem is to separate the **scheduling mechanism** from the **scheduling policy**. What this means is that the scheduling algorithm is parameterized in some way, but the parameters can be filled in by user processes. Let us consider the data base example again. Suppose that the kernel uses a priority scheduling algorithm but provides a system call by which a process can set (and change) the priorities of its

children. In this way the parent can control in detail how its children are scheduled, even though it itself does not do the scheduling. Here the mechanism is in the kernel but the policy is set by a user process.

2.5 OVERVIEW OF PROCESSES IN MINIX

Having completed our study of the principles of process management interprocess communication, and scheduling, we can now take a look at how they are applied in MINIX. Unlike UNIX, whose kernel is a monolithic program not split up into modules, MINIX itself is a collection of processes that communicate with each other and with user processes using a single interprocess communication primitive—message passing. This design gives a more modular and flexible structure, make it easy, for example, to replace the entire file system by a completely different one, without having even to recompile the kernel.

2.5.1 The Internal Structure of MINIX

Let us begin out study of MINIX by taking a bird's-eye view of the system. MINIX is structured in for layers, with each layer performing a well-defined function. The four layers are illustrated in Fig. 2-26.

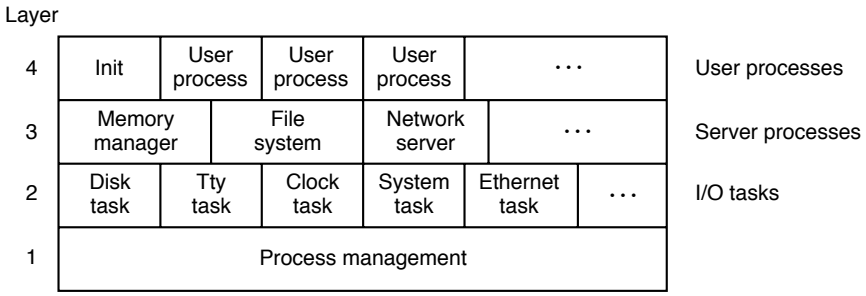


Figure 2-26. MINIX is structured in four layers.

The bottom layer catches all interrupts and traps, does scheduling, and provides higher layers with a model of independent sequential processes that communicate using messages. The code in this layer has two major functions. The first is catching the traps and interrupts, saving and restoring registers, scheduling, and the general nuts and bolts of actually making the process abstraction provided to the higher layers work. The second is handling the mechanics of messages; checking for legal destinations, locating send and receive buffers in the physical memory, and copying bytes from sender to receiver. That part of the layer dealing with the lowest level of interrupt handling is written in assembly language. The rest of the layer and all of the higher layers are written in C.

Layer 2 contains the I/O processes, one per device type. To distinguish them from ordinary processes, we will call them **tasks**, but the differences between tasks and processes are minimal. In many systems the I/O tasks are called **device drivers**; we will use the terms "tasks" and "device drivers" interchangeably. A task is needed for each device type, including disks, printers, terminals, network interfaces, and clocks. If other I/O devices are present, a task is needed for each one of these, too. One task, the system task, is

a little different, since it does not correspond to any device. We will discuss the tasks in the next chapter.

All the tasks in layer 2 and all the code in layer 1 are linked together into a single binary program called the **kernel**. Some of the tasks share common subroutines, but otherwise they are independent from one another, are scheduled independently, and communicate using messages. Intel processors starting from 286 assign one of four levels of privilege to each process. Although the tasks and the kernel are compiled together, when the kernel and the interrupt handlers are executing, they are accorded more privileges than the tasks. Thus the true kernel code can access any part of the memory and any processor register—essentially, the kernel can execute any instruction using data from anywhere in the system. Tasks cannot execute all machine level instructions, nor can they access all CPU registers or all parts of memory. They can, however, access memory regions belonging to less-privileged process, in order to perform I/O for them. One task, the system task, does not do I/O in the normal sense but exists in order to provide services, such as copying between different memory regions, for processes which are not allowed to do such things for themselves. On machines which do not provide different privilege levels, such as older Intel processors, there restrictions cannot be enforced, of course.

Layer 3 contains processes that provide useful services to the user processes. These server processes run at a less privileged level than the kernel and tasks and cannot access the I/O ports directly. They also cannot access memory outside the segments allocated to them. The **memory manager** (MM) carries out all the MINIX system calls that involve memory management, such as FORK, EXEC, and BRK. The **file system** (FS) carries out all the file system calls, such as READ, MOUNT, and CHDIR.

As we noted as the start of Chap.1, operating systems do two things: manage resources and provide an extended machine by implementing system calls. In MINIX the resource management is largely in the kernel (layers 1 and 2), and system call interpretation is in layer 3. The file system has been designed as a file "server" and can be moved to a remote machine with almost no changes. This also holds for the memory mangager, although remote memory servers are not as useful as remote file servers.

Additional servers may also exist in layer 3. Figure 2-26 shows a network server there. Although MINIX as described in this book does not include the network server, its source code is part of the standard MINIX distribution. The system can easily be recompiled to include it.

This is a good place to note that although the servers are independent processes, they differ from user processes in that they are started when the system is started, and they never terminate while the system is active. Additionally, although they run at the same privilege level as the user processes in terms of the machine instructions they are allowed to execute, they receive higher execution priority than user processes. To accommodate a new server the kernel must be recompiled. The kernel startup code installs the servers in privileged slots in the process table before any user processes are allowed to run.

Finally, layer 4 contains all the user processes—shells, editors, compilers, and user-written *a.out* programs. A running system usually has some processes that are started when the system is booted and which run forever. For example, a **daemon** is background process that executes periodically or always waits for some event, such as a packet arrival from the network. In a sense a daemon is a sever that is started independently and runs as a user process. However, unlike true severs installed in privileged slots, such programs can not get the special treatment from the kernel that the memory and file server processes receive.

2.5.2 Process Management in MINIX

Processes in MINIX follow the general process model described at some length earlier in this chapter. Processes can create subprocesses, which in turn can create more subprocesses, yielding a tree of processes. In fact, all the user processes in the whole system are part of a single tree with *init*(see Fig. 2-26) at the root.

How does this situation come about? When the computer is turned on, the hardware reads the first sector of the first track of the boot disk into memory and execute the code if finds there. The details vary depending upon whether the boot disk is a diskette or a hard disk. On a diskette this sector contains the bootstrap program. It is very small, since it has to fit in one sector. The MINIX bootstrap loads a larger program, *boot*, which then loads the operating system itself.

In contrast, hard disks require an intermediate step. A hard disk is divided into **partitions**, and the first sector of a hard disk contains a small program and the disk's **partition table**. Collectively these are called the **master boot record**. The program part is executed to read the partition table and to select the **active** partition. The active partition has a bootstrap on its first sector, which is then loaded and executed to find and start a copy of *boot* in the partition, exactly as is done when booting from a diskette.

In either case, *boot* looks for a multipart file on the diskette or partition and loads the individual parts into memory at the proper locations. The parts include the kernel, the memory manager, the file system, and *init*, the first user process. This startup process is not a trivial process. Operations that are in the realms of the disk task and the file system must be performed by boot before these parts of the system are active. In a latter section we will return to the subject of how MINIX is started. For now suffice it to say that once the loading operating is complete the kernel starts running.

During its initialization phase, the kernel starts the tasks, and then the memory manager, the file system, and any other servers that run in layer 3. when all these have run and initialized themselves, they will block, waiting for something to do. When all tasks and servers are blocked, *init*, the first user process, will be executed. It is already in memory, but it could, of course, have been loaded from the disk as a separate program since everything else is working by the time it is started. However, since *init* is started only this one time and is never reloaded from the disk, it is easiest just to include it in the system image file with the kernel, tasks, and servers.

init starts out by reading the file */etc/ttytab*, which lists all potential terminal devices. Those devices that can be used as login terminals (in the standard distribution, just the console) have an entry in *getty* fiend of */etc/ttytab*, *init* forks off a child process for each such terminal. Normally, each child executes */usr/bin/getty* which prints a message, then waits for a name to be typed. Then */usr/bin/login* is called with the name as its argument. If a particular terminal requires special treatment (e.g., a dial-up line) */etc/ttytab* can specify a command (such as */usr/bin/stty*) to be executed to initialize the line before running *getty*.

After a successful login, */bin/login* executes the user's shell (specified in the */etc/passwd* file, and normally */bin/sh* or */usr/bin/ash*). The shell waits for commands to e typed and then forks off a new process for each command. In this way, the shells are the children of *init*, the user processes are the grandchildren of *init*, and all the user processes in the system are part of a single tree.

The two principal MINIX system calls for process management are FORK and EXEC. FORK is the only way to create a process. EXEC allows a process to execute a specified program. When a program is executed, it is allocated a portion of memory whose size

is specified in the program file's header. It keeps this amount memory throughout its execution, although the distribution among data segment, stack segment, and unused can vary as the process runs.

All the information about a process in kept in the process table, which is divided up among the kernel, memory manager, and file system, with each one having the fields that it needs. When a new process comes into existence (by FORK), or an old process terminates (by EXIT or a signal), the memory manager first updates its part of the process table and then sends messages to the file system and kernel telling them to do likewise.

2.5.3 Interprocess Communication in MINIX

Three primitives are provided for sending and receiving messages. They are called by the C library procedures

```
send(dest, &message);
```

to send a message to process *dest*,

```
receive(source, &message);
```

to receive a message from process *source* (or *ANY*), and

```
send_rec(src_dst, &message);
```

to send a message and wait for a reply from the same process. The second parameter in each call is the local address of the message data. The message passing mechanism in the kernel copies the message from the sender to the receiver. The replay (for *send_rec*) overwrites the original message. In principle this kernel mechanism could be replaced by a function which copies messages over a network to a corresponding function on another machine, to implement a distributed system. In practice this would be complicated somewhat by the fact that message contents are sometimes pointers to large data structures, and a distributed system would also have to provide for copying the data itself over the network.

Each process or tast can send and receive messages from processes and tasks in its own layer, and from those in the layer directly below it. User processes may not communicate directly with the I/O tasks. The system enforces this restriction.

When a process (which also includes the tasks as a special case) sends a message to a process that is not currently waiting for a message, the sender blocks until the destination does a RECEIVE. In other works, MINIX uses the rendezvous method to avoid the problems of buffering sent, but not yet received, messages. Although less flexible than a scheme with buffering, it turns out to be adequate for this system, and much simpler because no buffering management is needed.

2.5.4 Process Scheduling in MINIX

The interrupt system is what keeps a multiprogramming operating system going. Processes block when they make requests for input, allowing other processes to execute. When input becomes available, the current running process is interrupted by the disk, keyboard, or other hardware. The clock also generates interrupts that are used to make sure a running user process that has not requested input eventually relinquishes the CPU, to give other processes their chance to run. It is the job of the lowest layer in MINIX

to hide these interrupts by turning them into messages. As far as processes (and tasks) are concerned, when an I/O device completes an operating it sends a message to some process, waking it up and making it runnable.

Each time a process is interrupted, whether from a conventional I/O device or from the clock, there is an opportunity to redetermine which process is most deserving of an opportunity to run. Of course, this must be done whenever a process terminates, as well, but in a system like MINIX interruptions due to I/O operations or the clock occur more frequently more process termination. The MINIX scheduler uses a multilevel queuing system with three levels, corresponding to layers 2, 3, and 4 in Fig. 2-26. Within the task and server levels processes run until they block, but user processes are scheduled using round robin. Tasks have the highest priority, the memory manager and file server are next, and user processes are last.

When picking a process to run, the scheduler checks to see if any tasks are ready. If one or more are ready, the one at the head of the queue is run. If no tasks are ready, a server (MM or FS) is chosen, if possible; otherwise a user is run. If no process is ready, the *IDLE* process is chosen. This is a loop that executes until the next interrupt occurs.

At each clock tick, a check is made to see if current process is a user process that has run more than 100 msec. If it is, the scheduler is called to see if another user process is waiting for the CPU. If one is found, the current process is moved to the end of its scheduling queue, and the process now at the head is run. Tasks, the memory manger, and the file system are never preempted by the clock, no matter how long they have been running.

2.6 IMPLEMENTATION OF PROCESSES IN MINIX

We are now moving closer to looking at the actual code, so a few words about the notation we will use are in order. The terms "procedure," "function," and "routine" will be used interchangeably. Names of variables, procedures and files will be written in italics, as in *rw_flag*. When a variable, procedure, or file name starts a sentence, it will be capitalized, but the actual names all begin with lowercase letters. System calls will be small caps, for example, READ.

The book and the software, both of which are continuously evolving, did not "go to press" on the same day, so there may be minor discrepancies between the references to the code, the printed listing, and the CD-ROM version. Such differences generally only affect a line or two, however. The source code printed in the book has also been simplified by eliminating code used to compile options that are not discussed in the book.

2.6.1 Organization of the MINIX Source Code

Logically, the source code is organized as two directories. The full paths to these directories on a standard MINIX system are */usr/include/* and */usr/src/* (a trailing "/" in a path name indicates that it refers to a directory). The actual location of the directories may vary from system to system, but normally the structure of the directories below the topmost level will be the same on any system. We will refer to these directories as *include/* and *src/* in this text.

The *include/* directory contains a number of POSIX standard header files. In addition, it has three subdirectories:

1. *sys/* - this subdirectory contains additional POSIX headers.

2. *minix/* - includes header files used by the operating system.
3. *ibm/* - includes header files with IBM PC-specific definitions.

To support extensions to MINIX and programs that run in the MINIX environment, other files and subdirectories are also present in *include/* as provided on the CD-ROM or over the Internet. For instance, the *include/net/* directory and its subdirectory *include/net/gen/* support network extensions.However, in this text only the files needed to compile the basic MINIX system are printed and discussed.

The *src/* directory contains three important subdirectories containing the operating system source code:

1. *kernel/* - layers 1 and 2 (processes, messages, and drivers).
2. *mm/* - the code for the memory manager.
3. *fs/* - the code for file system.

There are three other source code directories that are not printed or discussed in the text, but which are essential to producing a working system:

1. *src/lib/* - source code for library procedures (e.g., open, read).
2. *src/tools/* - source code for the init program, used to start MINIX.
3. *src/boot/* - the code for booting and installing MINIX.

The standard distribution of MINIX includes several more source directories. An operating system exists, of course, to support commands (programs) that will run on it, so there is a large *src/commands/* directory with source code for the utility programs (e.g., *cat*, *cp*, *date*, *ls*, *pwd*). Since MINIX is an educational operating system, meant to be modified, there is a *src/test/* directory with programs designed to test thoroughly a newly compiled MINIX system. Finally, the *src/inet/* directory includes source code for recompiling MINIX with network support.

For convenience we will usually refer to simple file names when it will be clear from the context what the complete path name is. It should be noted, however, that some file names appear in more than one directory. For instance, there are several file named *const.h* in which constants relevant to a particular part of the system are defined. The files in a particular directory will be discussed together, so there should not be any confusion. The files are listed in Appendix A in the order they are discussed in the text, to make it easier to follow along. Acquisition of a couple of bookmarks might be of use at this point.

Also worth noting is that Appendix B contains an alphabetical list of all files described in Appendix A, and Appendix C contains a list of where to find the definitions of macros, global variables, and procedures used in MINIX.

This code for layer 1 and 2 is contained in the directory *src/kernel/*. In this chapter, we will study the files in this directory which support process management, the lowest layer of the MINIX structure we saw in Fig. 2-26. This layer includes functions which handle system initialization, interrupts, message passing and process scheduling. In Chap. 3, we will look at the rest of the files in this directory, which support the various tasks, the second layer in Fig. 2-26. In Chap. 4, we will look at the memory manager files in *src/mm/*, and in Chap. 5, we will study the file system, whose source files are located in *src/fs/*.

When MINIX is compiled, all the source code files in *src/kernel/*, *src/mm/*, *src/fs/* are compiled to object files. All the object files in *src/kernel/* are linked to form a single executable program, *kernel*. The object files in *src/mm/* are also linked together to

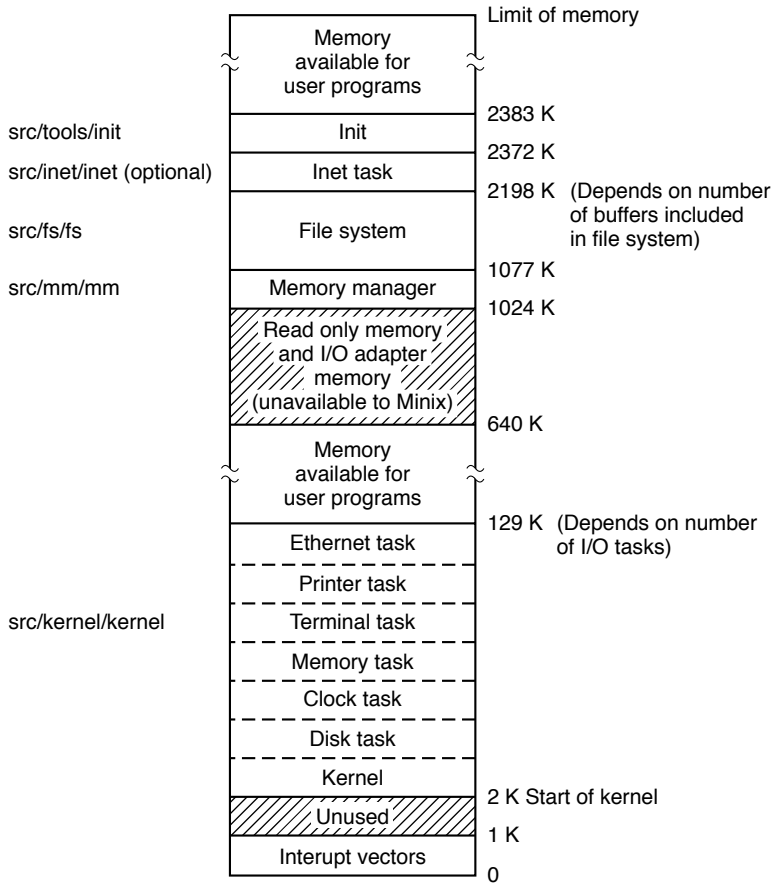


Figure 2-27. Memory layout after MINIX has been loaded from disk into memory. The four (or five, with network support) independently compiled and linked parts are clearly distinct. The sizes are approximate, depending on the configuration.

form a single executable program, *mm*. The same holds for *fs*. Extensions can be added by adding additional servers, for instance network support is added by modifying *include/minix/config.h* to enable compilation of the files in *src/inet* to form *inet*. another executable program *init* is built in *src/tools/*. The program *installboot* (whose source is in *src/tools*) adds names to each of these programs, pads each one out so that its length is a multiple of the disk sector size (to make it easier to load the parts independently), and concatenate them onto a single file. This new file is the binary of the operating system and can be copied onto the root directory of the */minix/* directory of a floppy disk or hard disk partition. Later, the boot monitor program can load and execute the operating system. Figure 2-27 shows the layout of memory after the concatenated programs are separated and loaded Details, of course,depend upon the system configuration. The example in the figure is for a MINIX system configured to take advantage of a computer equipped with several megabytes of memory. This makes it possible to allocate a large number of file system buffers, but the resulting large file system does not fit in the lower range of memory, below 640K. If the number of buffers is reduced drastically it is possible to make the entire system fit into less than 640K of memory, with room for a few user processes as well.

It is important to realize that MINIX consists of three or more totally independent programs that communicate only by passing messages. A procedure called *panic* in *src/fs/*

does not conflict with a procedure called *panic* in *src/mm/* because they ultimately are linked into different executable files. The only procedures that the three pieces of the operating system have in common are a few of the library routines in *lib/*. This modular structure makes it very easy to modify, say, the file system, without having these changes affect the memory manager. It also makes it straightforward to remove the file system altogether and put it on a different machine as a file server, communicating with user machines by sending messages over a network.

As another example of the modularity of MINIX, compiling the system with of without network support makes absolutely no difference to the memory manager or the file system and affects the kernel only because the Ethernet task is compiled there, along with support for other I/O devices. When enabled, the network server is integrated into the MINIX system as server with the same level of priority as the memory manager or the file server. Its operation can involve the transfer of large quantities of data very rapidly, and this requires higher priority than a user process would receiver. Except for the Ethernet task, however, network functions could be performed by user level processes. Network functions are not traditionally operating system functions, and detailed discussion of the network code is beyond the scope of this book. In succeeding sections and chapters the discussion will be based on a MINIX system compiled without network support.

2.6.2 The Common Header Files

The directory *include/* and its subdirectories contain a collection of files defining constants, macros, and types. The POSIX standard requires many of these definitions an specifies in which files of the main *include/* directory and its subdirectory *include/sys/* each required definitions to be found. The files in these directories are **header** or **include files**, identified by the suffix *.h* , and used by means of `#include` statements in C source files. These statements are a feature of the C language. Include files make maintenance of a large system easier.

Headers likely to be needed from compiling user programs are found in *include/* whereas *include/sys/* traditionally is used from files that are used primarily for compiling system programs and utilities. The distinction is not terribly important, and typical compilation, whether of a user program or part of the operating system, will include files from both these directories. We will discuss here the files that are needed to compile the standard MINIX system, first treating those in *include/* and then those in *include/sys*. In the next section we will discuss all the files in the *include/minix/* and emphinclude/ibm/ directories, which, as the directory names indicate, are unique to MINIX and its implementation on IBM-type computers.

The first headers to be considered are truly general purpose ones, so much so that they are not referenced by any of the C language source files for the MINIX system. Rather, they are themselves included in other header files, the master headers *src/kernel/kernel.h*, *src/mm/mm.h*, and *src/fs/fs.h* for each of the three main parts of the MINIX system, which in turn are included in every compilation. Each master header it tailored to the needs of the corresponding part of the MINIX system, but each one starts with a section like the one shown in Fig 2-28. The master headers will be discussed again in other sections of the book. This preview is to emphasize that headers from several directories are used together. In this section and the next one we will mention each of the files referenced in Fig 2-28.

Let us start with the first header in *include/*, *ansi.h* (line 0000). This is the second header that is processed whenever any part of the MINIX system is compiled; only *in-*

```
#include <minix/config.h>
#include <ansi.h>
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <limits.h>
#include <errno.h>
#include <minix/syslib.h>
```

Figure 2-28. Part of the a master header which ensures inclusion of header files needed by all C source files.

clude/minix/config.h is processed earlier. The purpose of *ansi.h* is to test whether the compiler meets the requirements of standard C, as define by the International Organization for Standards. Standard C is also called ANSI C, since the standard was originally developed by the American National Standards Institute before gaining international recognition. A Standard C compiler defines several macros that can then be tested in programs being compiled. *__STDC__* is such a macro, and it is defined by a standard compiler to have a value of 1, just as if the C preprocessor had read a line like

```
#define __STDC__
```

The compiler distributed with current versions of MINX conforms to a Standard C, but older versions MINIX were developed before the adoption of the standard, and it is still possible to compile MINIX with a classis (Kernighan & Ritchie) C compiler. It is intended that MINIX should be easy to port to new machine, and allowing older compilers is part of this. At line 0023 to 0025 the statement

```
#define _ANSI
```

is processed if a Standard C compiler is in use. *Ansi.h* defines several macros in different ways, depending upon whether the *_ANSI* macro is defined.

This most important macro in this file is *_PROTOTYPE*. This macro allows us to write function prototypes in the form

```
_PROTOTYPE (return-type function-name, (argument-type argument, ...))
```

and have this transformed by the C preprocessor int

```
return-type function-name(argument-type argument, ...)
```

if the compiler is an ANSI Standard C compiler, or

```
return-type function-name()
```

if the compiler is an old-fashioned (i.e., Kernighan & Ritchie) compiler.

Before we leave *ansi.h* let us mention one more feature. The entire file is enclosed between lines that read

```
#ifndef _ANSI_H
```

and

```
#endif
```

On the line immediately following the `#ifndef` *_ANSI_H* it itself is defined. A header file should be included only once in a compilation; this construction ensures that contents of the file will be ignored if it is included multiple times. We will see this technique used in all the header files in the *include/* directory.

The second file in *include/* that is indirectly included in every MINIX source file is the *limits.h* header (line 0100). This file defines many basic sizes, both language types such as the number of bits in an integer, as well as operating system limits such as the length of a file name. *Errno.h* (line 0200), is also included by all the master headers. It contains the error numbers that are returned to the user programs in the global variable *errno* when a system call fails. *Errno* is also used to identify some internal errors, such as trying to send a message to a non-existing task. The error numbers are negative to mark them error codes within the MINIX system, but they must be made positive before being returned to user programs. This trick that is used is that each error code is defined in a line like

```
#define EPERM (_SIGN 1)
```

(line 0236). The master header for each part of the operating system defines the macro *_SYSTEM*, but *_SYSTEM* is never defined when a user program is compiled. If *_SYSTEM* is defined, then *_SIGN* is defined as "-"; otherwise it is given a null definition.

The next group of files to be considered are not included in all the master headers, but are nevertheless used in many source files in all parts of the MINIX system. The most important is *unistd.h* (line 0400). This header defines many constants, most of which are required by POSIX. In addition, it includes prototypes for many C functions, including all those used to access MINIX system calls. Another widely used file is *string.h* (line 0600), which provides prototypes for many C functions used for string manipulation. The header *signal.h* (line 0700) defines the standard signal names. It also contains prototypes for some signal-related functions. As we will see later, signal handling involves all parts of MINIX.

Fcntl.h (line 0900) symbolically defines many parameters used in file control operations. For instance, it allows one to use the macro *O_RDONLY* instead of the numeric value 0 as a parameter to a *open* call. Although this file is referenced most by the file system, its definitions are also needed in a number of places in the kernel and the memory manager.

The remaining files in *include/* are not as widely used as the ones already mentioned. *Stdlib.h* (line 1000) defines types, macros, and function prototypes that are likely to be needed in the compilation of all but the most simple of C programs. It is one of the most frequently used headers in compiling user programs, although within the MINIX system source it is referenced by only a few files in the kernel.

As we will see when we look at the tasks layer in Chap. 3, the console and terminal interface of an operating system is complex, because many different types of hardware have to interact with operating system and user programs in a standardized way. The *termio.h* (line 1100) header defines constants, macros, and function prototypes used for control of terminal-type I/O devices. The most important structure is the *termios* structure. It contains flags to signal various modes of operation, variables to set input and output transmission speeds, and an array to hold special characters, such as the *INTR* and *KILL* characters. This structure is required by POSIX, as are many of the macros and function prototypes defined in this file.

However, as all-encompassing as the POSIX standard is meant to be, it does not provide everything one might want, and the last part of the file, from line 1241 onward,

provides extensions to POSIX. Some of these are of obvious value, such as extensions to define standard baud rates of 57,600 baud and higher, and support for terminal display screen windows. The POSIX standard does not forbid extensions, as no reasonable standard can ever be all-inclusive. But when writing a program in the MINIX environment which is intended to be portable to other environments, some caution is required to avoid the use of definitions specific to MINIX. This is easy to do. In this file and other files that define MINIX-specific extensions the use of the extensions is controlled by an

```
#ifdef _MINIX
```

statement. If *_MINIX* is not defined, the compiler will not even see the MINIX extensions.

The last file we will consider in *include/* is *a.out.h* (line 1400), a header which defines the format of the files in which executable programs are stored on disk, including the header structure used to start a file executing and the symbol table structure produced by the compiler. It is referenced only by the file system.

Now let us go on to the subdirectory *include/sys/*. As shown in Fig. 2-28, the master headers for the main parts of the MINIX system all include *sys/types.h* (line 1600) immediately after reading *ansi.h*. This header defines many data types used by MINIX. Errors that could arise from misunderstanding which fundamental data types are used in a particular situation can be avoided by using the definitions provided here. Fig. 2-29 shows the way the sizes, in bits, of a few types defined in this file differ when compiled for 16-bit or 32-bit processors. Note that all type names end with "_t". This is not just a convention; it is a requirement of the POSIX standard. This is an example of a **reserved suffix**, and it should not be used as a suffix of any name which is *not* a type name.

Type	16-Bit MINIX	32-Bit MINIX
gid_t	8	8
dev_t	16	16
pid_t	16	32
ino_t	16	32

Figure 2-29. The size, in bits, of some types on l6-bit and 32-bit systems.

Although it is not so widely used that it is included in the master headers for each section, *sys/ioctl.h* (line l800) defines many macros used for device control operations. It also contains the prototype for the IOCTL system call. This call is not directly invoked by programmers in many cases, since the POSIX-defined functions prototyped in *include/termios.h* have replaced many uses of the old *ioctl* library function for dealing with terminals, consoles, and similar devices. Nevertheless, it is still necessary. In fact, the POSIX functions for control of terminal devices are converted into IOCTL system calls by the library. Also, there are an ever-increasing number of devices, all of which need various kinds of control, which can be interfaced with a modern computer system. For instance, near the end of this file there are several operation codes defined that begin with *DSPIO*, for controlling a digital signal processor. Indeed, the main difference between MINIX as described in this book and other versions is that for purposes of the book we describe a MINIX with relatively few input/output devices. Many others, such as network interfaces, CD-ROM drives, and sound cards, can be added; control codes for all of these are defined as macros in this file.

Several other files in this directory are widely used in the MINIX system. The file *sys/sigcontex.h* (line 2000) defines structures used to preserve and restore normal system

operation before and after execution of a signal handling routine and is used both in the kernel and the memory manager. There is support in MINIX for tracing executables and analyzing core dumps with a debugger program, and *sys/ptrace.h* (line 2200) defines the various operations possible with the PTRACE system call. *Sys/stat.h* (line 2300) defines the structure which we saw in Fig. 1-12, returned by the STAT and FSTAT system calls, as well as the prototypes of the functions *stat* and *fstat* and other functions used to manipulate file properties. It is referenced in several parts of the file system and the memory manager.

The last two files we will discuss in this section are not as widely referenced as the ones discussed above. *Sys/dir.h* (line 2400) defines the structure of a MINIX directory entry. It is only referenced directly once, but this reference includes it in another header that is widely used in the file system. It is important because, among other things, it tells how many characters a file name may contain. Finally, the *sys/wait.h* (line 2500) header defines macros used by the WAIT and WAITPID system calls, which are implemented in the memory manager.

2.6.3 The MINIX Header Files

The subdirectories *include/minix/* and *include/ibm/* contain header files specific to MINIX. Files in *include/minix/* are needed for an implementation of MINIX on any platform, although there are platform-specific alternative definitions within some of them. The files in *include/ibm/* define structures and macros that are specific to MINIX as implemented on IBM-type machines.

We will start with the *minix/* directory. In the previous section, it was noted that *config.h* (line 2600) is included in the master headers for all parts of the MINIX system, and is thus the first file actually processed by the compiler. On many occasions, when differences in hardware or the way the operating system is intended to be used require changes in the configuration of MINIX, editing this file and recompiling the system is all that must be done. The user-settable parameters are all in the first part of the file. The first of these is the *MACHINE* parameter, which can take values such as *IBM_PC*, *SUN_4*, *MACINTOSH*, or other values, depending on the type of machine for which MINIX is being compiled. Most of the code for MINIX is independent of the type of machine, but an operating system always has some system-dependent code. In the few places in this book where we discuss code that is written differently for different systems we will use as our examples code for IBM PC-type machines with advanced processor chips (80386, 80486. Pentium, Pentium Pro) that use 32-bit words. We will refer to all of these as Intel 32-bit processors. MINIX can also be compiled for older IBM PCs with a 16-bit word size, and the machine-dependent parts of MINIX must be coded differently for these machines. On a PC, the compiler itself determines the machine type for which MINIX will be compiled. The standard PC MINIX compiler is the Amsterdam Compiler Kit (ACK) compiler. It identities itself by defining, in addition to the *__STDC__* macro, the *__ACK__* macro. It also defines a macro *_EM_WSIZE* which is the word size (in bytes) for its target machine. In lines 2626 to 2623 a macro *_WORD_SIZE* is assigned the value of *_EM_WSIZE*. Further along in the file and at various places in the other MINIX source files these definitions are used. For example, lines 2647 to 2650 begin with the test

```
#if (MACHINE == IBM_PC && _WORD_SIZE == 4)
```

and define a size for the file system's buffer cache on 32-bit systems.

Other definitions in *config.h* allow customization for other needs in a particular installation. For instance, there is a section that allows various types of device drivers to be included when the MINIX kernel is compiled. This is likely to be the most often edited part of the MINIX source code. This section starts out with:

```
#define ENABLE_NETWORKING 0
#define ENABLE_AT_WINI 1
#define ENABLE_BIOS_WINI 0
```

By changing the 0 in the first line to 1 we can compile a MINIX kernel for a machine that needs network support. By defining *ENABLE_AT_WINI* as 0 and *ENABLE_BIOS_WINI* as 1, we can eliminate the AT-type (i.e., IDE) hard disk driver code and use the PC BIOS for hard disk support.

The next file is *const.h* (line 2900), which illustrates another common use of header files. Here we find a variety of constant definitions that are not likely to be changed when compiling a new kernel but that are used in a number of places. Defining them here helps to prevent errors that could be hard to track down if inconsistent definitions were made in multiple places. There are other files named *const.h* in the MINIX source tree, but they are for more limited use. Definitions that are used only in the kernel are included in *src/kernel/const.h*. Definitions that are used only in the file system are included in *src/fs/const.h*. The memory manager uses *src/mm/const.h* for its local definitions. Only those definitions that are used in more than one part of the MINIX system are included in *include/minix/const.h*.

A few of the definitions in *const.h* are noteworthy. *EXTERN* is defined as a macro expanding into *extern* (line 2906). Global variables that are declared in header files and included in two or more files are declared *EXTERN*, as in

```
EXTERN int who;
```

If the variable were declared just as

```
int who;
```

and included in two or more files, some linkers would complain about a multiply defined variable. Furthermore, the C reference manual (Kernighan and Ritchie, 1988) explicitly forbids this construction.

To avoid this problem, it is necessary to have the declaration read

```
extern int who;
```

in all places but one. Using *EXTERN* prevents this problem by having it expand into extern everywhere that *const.h* is included, except following an explicit redefinition of *EXTERN* as the null string. This is done in each part of *MINIX* by putting global definitions in a special file called *glo.h*, for instance, *src/kernel/glo.h*, which is indirectly included in every compilation. Within each *glo.h* there is a sequence

```
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif
```

and in the *table.c* files of each part of MINIX there is a line

```
#define _TABLE
```

preceding the #include section. Thus when the header files are included and expanded as part of the compilation of *table.c*, *extern* is not inserted anywhere (because *EXTERN* is defined as the null string within *table.c*) and storage for the global variables is reserved only in one place, in the object file *table.o*.

If you are new to C programming and do not quite understand what is going on here, fear not; the details are really not important. Multiple inclusion of header files can cause problems for some linkers because it can lead to multiple declarations for included variables. The *EXTERN* business is simply a way to make MINIX more portable so it can be linked on machines whose linkers do not accept multiply defined variables.

PRIVATE is defined as a synonym for *static*. Procedures and data that are not referenced outside the file in which they are declared are always declared as *PRIVATE* to prevent their names from being visible outside the file in which they are declared. As a general rule, all variables and procedures should be declared with as local a scope as possible. *PUBLIC* is defined as the null string. Thus, the declaration

```
PUBLIC void free_zone(Dev_t dev, zone_t numb)
```

comes out of the C preprocessor as

```
void free_zone(Dev_t dev, zone_t numb)
```

which, according to the C scope rules, means that the name *free_zone* is exported from the file and can be used in other files. *PRIVATE* and *PUBLIC* are not necessary but are attempts to undo the damage caused by the C scope rules (the default is that names are exported outside the file; it should be just the reverse).

The rest of *const.h* defines numerical constants used throughout the system. A section of *const.h* is devoted to machine or configuration-dependent definitions. For instance, throughout the source code the basic unit of memory size is the click. The size of a click depends upon the processor architecture, and alternatives for Intel, Motorola 68000, and Sun SPARC architectures are defined on lines 2957 to 2965. This file also contains the macros *MAX* and *MIN*, so we can say

```
z = MAX(x, y);
```

to assign the larger of x and y to z.

Type.h (line 3100) is another file that is included in every compilation by means of the master headers. It contains a number of key type definitions, along with related numerical values. The most important definition in this file is *message* on lines 3135 to 3146. While we could have defined *message* to be an array of some number of bytes, it is better programming practice to have it be a structure containing a union of the various message types that are possible. Six message formats, *mess_1* through *mess_6*, are defined. A message is a structure containing a field *m_source*, telling who sent the message, a field *m_type*, telling what the message type is (e.g., GET_TIME to the clock task) and the data fields. The six message types are shown in Fig. 2-30. In the figure the first and second message types seem identical, as do the fourth and sixth types. This is true for MINIX as implemented on an Intel CPU with a 32-bit word size, but would not be the case on a machine where *int*s, *long*s and pointers were different sizes. Defining six distinct formats makes it easier to recompile for a different architecture.

When it is necessary to send a message containing, say, three integers and three pointers (or three integers and two pointers), then the first format in Fig. 2-30 is the one to use.

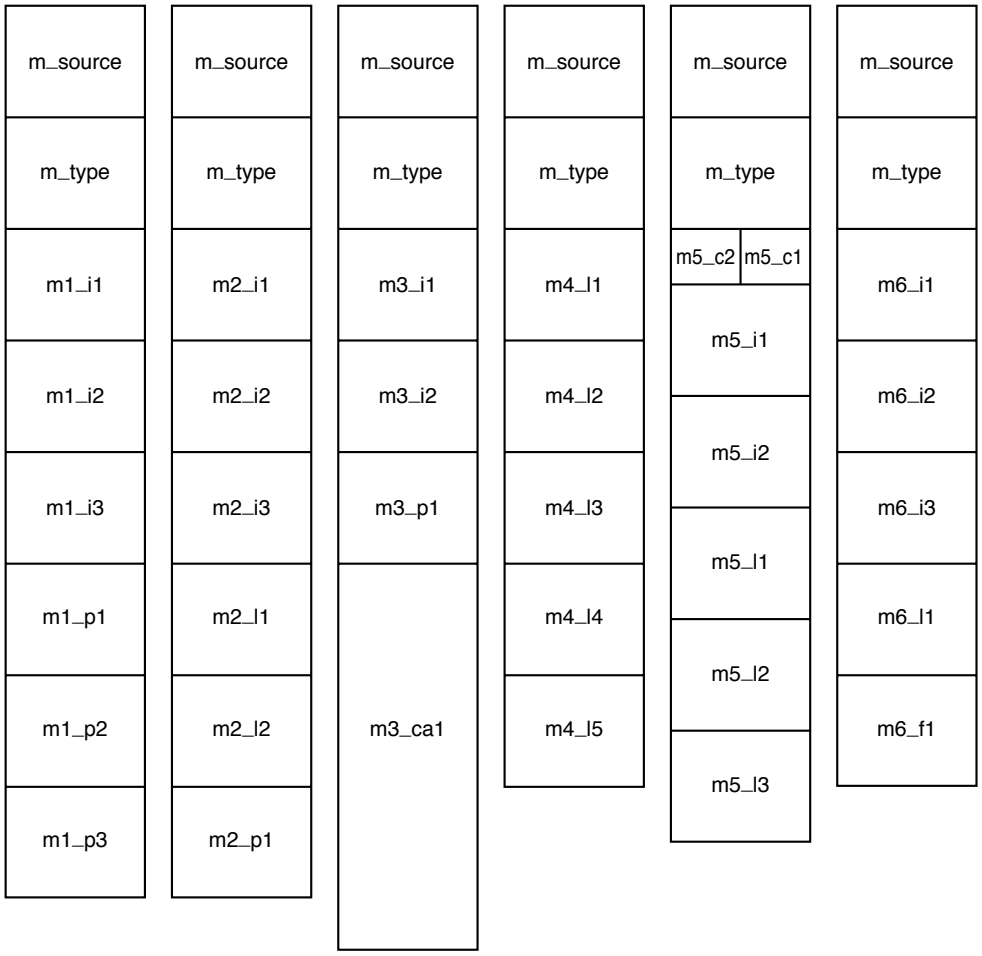


Figure 2-30. The six messages types used in MINIX. The sizes of message elements will vary, depending upon the architecture of the machine; this diagram illustrates sizes on a machine with 32-bit pointers, such as the Pentium (Pro).

The same applies to the other formats. How does one assign a value to the first integer in the first format? Suppose that the message is called *x*. Then *x.m_u* refers to the union portion of the message struct. To refer to the first of the six alternatives in the union, we use *x.m_u.m_m1*. Finally, to get at the first integer in this struct we say *x.m_u.m_m1.m1i1*. This is quite a mouthful, so somewhat shorter field names are defined as macros after the definition of message itself. Thus *x.m1_i1* can be used instead of *x.m_u.m_m1.m1i1*. The short names all have the form of the letter m, the format number, an underscore, one or two letters indicating whether the field is an integer, pointer, long, character, character array, or function, and a sequence number to distinguish multiple instances of the same type within a message.

As an aside, while discussing message formats, this is a good place to note that an operating system and its compiler often have an "understanding" about things like the layout of structures and this can make the implementor's life easier. In MINIX the *int* fields in messages are sometimes used to hold *unsigned* data types. In some cases this could cause overflow, but the code was written using the knowledge that the MINIX compiler copies *unsigned* types to *int*s and *vice versa* without changing the data or generating code to detect overflow. A more compulsive approach would be to replace each int field with a union of an *int* and an *unsigned*. The same applies to the *long* fields in the messages;

some of them may be used to pass *unsigned long* data. Are we cheating here? Perhaps, one might say, but if you wish to port MINIX to a new platform, quite clearly the exact format of the messages is something to which you must pay a great deal of attention, and now you have been alerted that the behavior of the compiler is another factor that needs attention.

There is one other file in *include/minix* that is universally used, by means of inclusion in the master headers. This is *syslib.h* (line 3300), which contains prototypes for C library functions called from within the operating system to access other operating system services. The C libraries are not discussed in detail in this text, but many of them are standard and will be available for any C compiler. However, the C functions referenced by *syslib.h* are of course quite specific to MINIX and a port of MINIX to a new system with a different compiler requires porting these library functions. Fortunately this is not difficult, since these functions simply extract the parameters of the function call and insert them into a message structure, then send the message and extract the results from the reply message. Many of these library functions are defined in a dozen or fewer lines of C code.

When a process needs to execute a MINIX system call, it sends a message to the memory manager (MM for short) or the file system (FS for short). Each message contains the number of the system call desired. These numbers are defined in the next file, *callnr.h* (line 3400).

The file *com.h* (line 3500) mostly contains common definitions used in messages from MM and FS to the I/O tasks. The task numbers are also defined. To distinguish them from process numbers, task numbers are negative. This header also defines the message types (function codes) that can be sent to each task. For example, the clock task accepts codes *SET_ALARM* (which is used to set a timer), *CLOCK_TICK* (when a clock interrupt has occurred), *GET_TIME* (request for the real time), and *SET_TIME* (to set the current time of day). The value *REAL_TIME* is the message type for the reply to the *GET_TIME* request.

Finally, *include/minix/* contains several more specialized headers. Among these are *boot.h* (line 3700), which is used by both the kernel and file system to define devices and to access parameters passed to the system by the boot program. Another example is *keymap.h* (line 3800), which defines the structures used to implement specialized keyboard layouts for the character sets needed for different languages. It is also needed by programs which generate and load these tables. Some files here, like *partition.h* (line 4000), are used only by the kernel, and not by the file system or the memory manager. In an implementation with support for additional I/O devices there are more header files like this, supporting other devices. Their placement in this directory needs explanation. Ideally all user programs would access devices only through the operating system, and files like this would be placed in *src/kernel/*. However, the realities of system management require that there be some user commands that access system-level structures, such as commands to make disk partitions. It is to support such utility programs that such specialized header files are placed in the *include/* directory tree.

The last specialized header directory we will consider, *include/ibm/*, contains two files which provide definitions related to the IBM PC family of computers. One of these is *diskparm.h*, which is needed by the floppy disk task. Although this task is included in the standard version of MINIX, its source code is not discussed in detail in this text, since it is so similar to the hard disk task. The other file in this directory is *partition.h* (line 4100), which defines disk partition tables and related constants as used on IBM compatible systems. These are placed here to facilitate porting MINIX to another hardware

platform. For different hardware *include/ibm/partition.h* have to be replaced, presumably with a *partition.h* in another appropriately named directory, but the structure defined in the file *include/minix/partition.h* is internal to MINIX and should remain unchanged in a MINIX hosted on a different hardware platform.

2.6.4 Process Data Structures and Header Files

Now let us dive in and see what the code in *src/kernel/* looks like. In the previous two sections we structured our discussion around an excerpt from a typical master header, we will look first at the real master header for the kernel, *kernel.h* (line 4200). It begins by defining three macros. The first, *_POSIX_SOURCE* is a *feature test macro* defined by the POSIX standard itself. All such macros are required to begin with the underscore character, The effect of defining the *_POSIX_SOURCE* macro is to ensure that all symbols required by the standard and any that are explicitly permitted, but not required, will be visible, while hiding any additional symbols that are unofficial extensions to POSIX. We have already mentioned the next two definitions: the *_MINIX* macro overrides the effect of *_POSIX_SOURCE* for extensions defined by MINIX, and *_SYSTEM* can be tested wherever it is important to do something differently when compiling system code, as opposed to user code, such as changing the sign of error codes.

Kernel.h then includes other header files from *include/* and its subdirectories *include/sys/* and *include/minix/*, including all those referred to in Fig. 2-28. We have discussed all of these files in the previous two sections. Finally, four more headers from the local directory, *src/kernel/*, are included.

This is a good place to point out for newcomers to the C language how file names are quoted in a #include statement. Every C compiler has a default directory in which it looks for include files. Usually, this is */usr/include/*, as it is in a standard MINIX system. When the name of a file to be included is quoted between less-than and greater-than symbols (< >) the compiler searches for the file in the default include directory or in a specified subdirectory of the default directory. When the name is quoted between ordinary quote characters (`"..."`) the file is searched for first in the current directory (or a specified subdirectory) and then, if not found there, in the default directory.

Kernel.h makes it possible to guarantee that all source files share a large number of important definitions by writing the single line

```
#include "kernel.h"
```

in each of the other kernel source files. Since the order of inclusion of header files is sometimes important, *kernel.h* also ensures that this ordering is done correctly, once and forever. This carries to a higher level the "get it right once, then forget the details" technique embodied in the header file concept. There are similar master headers in the source directories for the file system and the memory manager.

Now let us proceed to look at the four local header files included in *kernel.h* Just as we have files *const.h* and *type.h* in the common header directory *include/minix/*, we also have files *const.h*. and *type.h* in the kernel source directory, *src/kernel/*. The files in *include/minix/* are placed there because they are needed by many parts of the system, including programs that run under the control of the system. The files in *src/kernel/* provide definitions needed only for compilation of the kernel. The FS and MM source directories also contain *consn.h* and *type.h* files to define constants and types needed only for those parts of the system. The other two files included in the master header, *proto.h*

and *glo.h*, have no counterparts in the main *include/* directories, but we will find that they, too, have counterparts used in compiling the file system and the memory manager.

Const.h (line 4300) contains a number of machine-dependent values, that is, values that apply to the Intel CPU chips, but that are likely to be different when MINIX is compiled for different hardware. These values are enclosed between

```
#if (CHIP == INTEL)
```

and

```
#endif
```

statements (lines 4302 to 4396) to bracket them.

When compiling MINIX for one of the Intel chips the macros *CHIP* and *INTEL* are defined and set equal in *include/minix/config.h* (line 2768), and thus the machine-dependent code will be compiled. When MINIX was ported to a system based on the Motorola 68000, the people doing the port added sections of code bracketed by

```
#if (CHIP == M68000)
```

and

```
#endif
```

and made appropriate changes in *include/minix/config.h* so a line reading

```
#define CHIP M68000
```

would be effective. In this way, MINIX can deal with constants and code that are specific to one system. This construction does not especially enhance readability, so it should be used as little as possible. In fact, in the interest of readability, we have removed many sections of machine-dependent code for 68000 and other processors from the version of the code printed in this text. The code distributed on the CD-ROM and via the Internet retains the code for other platforms.

A few of the definitions in *const.h* deserve special mention. Some of these are machine dependent, such as important interrupt vectors and field values used for resetting the interrupt controller chip after each interrupt. Each task within the kernel has its own stack, but while handling interrupts a special stack of size *K_STACK_BYTES*, defined here on line 4304, is used. This is also defined within the machine dependent section, since a different architecture could require more or less stack space.

Other definitions are machine-independent, but needed by many parts of the kernel code. For instance, the MINIX scheduler has *NQ*(3) priority queues, named *TASK_Q* (highest priority), *SERVER_Q* (middle priority), and *USER_Q* (lowest priority). The names are used to make the source code understandable, but the numeric values defined by these macros are actually compiled into the executable program. Finally, the last line of *const.h* defines *pringf* as a macro which will evaluate as *printk*. This allows the kernel to print messages, such as error messages, on the console using a procedure defined within the kernel. This bypasses the usual mechanism, which requires passing messages from the kernel to the file system, and then from the file system to the printer task. During a system failure this might not work. We will see calls to *prinrf* alias *printk*, in a kernel procedure called *panic*, which, as you might expect, is invoked when fatal errors are detected.

The file *type.h* (line 4500) defines several prototypes and structures used in any implementation of MINIX. The *tasktab* structure defines the structure of an element of the *tasktab* array and the *memory* structure (lines 4513 to 4516) defines the two quantities that uniquely specify an area of memory. This is a good place to mention some concepts used in referring to memory. A *click* is the basic unit of measurement of memory; in MINIX for Intel processors a click is 256 bytes. Memory is measured as *phys_clicks*, which can be used by the kernel to access any memory element anywhere in the system, or as *vir_clicks*, used by processes other than the kernel. A *vir_clicks* memory reference is always with respect to the base of a segment of memory assigned to a particular process, and the kernel often has to make translations between the two. The inconvenience of this is offset by the fact that a process can do all its own memory references in *vir_clicks*. One might suppose that the same unit could be used to specify the size of either type of memory, but there is an advantage to using *vir_clicks* to specify the size of a unit of memory allocated to a process, since when this unit is used a check is done to be sure that no memory is accessed outside of what has been specifically assigned to the current process. This is a major feature of the **protected mode** of modern Intel processors, such as the Pentium and Pentium Pro. Its absence in the early 8086 and 8088 processors caused some headaches in the design of earlier versions of MINIX.

Type.h also contains several machine-dependent type definitions, such as the *port_t*, *segm_t*, and *reg_t* types (lines 4525 to 4527) used on Intel processors, used, respectively, to address I/O ports, memory segments, and CPU registers.

Structures, too, may be machine-dependent. On lines 4537 to 4558 the *stackframe_s* structure, which defines how the machine registers are saved on the stack, is defined for Intel processors. This structure is extremely important—it is used to save and restore the internal state of the CPU whenever a process is put into or taken out of the "running" state of Fig. 2-2. Defining it in a form that can be efficiently read or written by assembly language code reduces the time required for a context switch. *Segdesc_s* is another structure related to the architecture of Intel processors. It is part of the protection mechanism that keeps processes from accessing memory regions outside those assigned to them.

To illustrate differences between platforms a few definitions for the Motorola 68000 family of processors were retained in this file. The Intel processor family includes some models with 16-bit registers and others with 32-bit registers, so the basic *reg_t* type is unsigned, for the Intel architecture. For Motorola processors *reg_t* is defined as the *u32_t* type. These processors also need a *stackframe_s* structure (lines 4583 to 4603), but the layout is different, to make the assembly code operations that use it as fast as possible. The Motorola architecture has no need at all for the *port_t* and *segm_t* types, or for the *segdesc_s* structure. There are also several structures defined for the Motorola architecture that have no Intel counterparts.

The next file, *proto.h* (line 4700), is the longest header file we will see. Prototypes of all functions that must be known outside of the file in which they are defined are in this file. All are written using the *_PROTOTYPE* macro discussed in the previous section, and thus the MINIX kernel can be compiled either with a classic C (Kernighan and Ritchie) compiler, such as the original MINIX C compiler, or a modern ANSI Standard C compiler, such as the one which is part of the MINIX Version 2 distribution. A number of these prototypes are system-dependent, including interrupt and exception handlers and functions that are written in assembly language. Prototypes of functions needed by drivers not discussed in this text are not shown. Conditional code for Motorola processors has also been deleted from this and the remaining files we will discuss.

The last of the kernel headers included in the master header is *glo.h* (line 5000) Here

we find the kernel's global variables. The purpose of the macro *EXTERN* was described in the discussion of *include/minix/const.h*. It normally expands into *extern*. Note that many definitions in *glo.h* are preceded by this macro. *EXTERN* is forced to be undefined when this file is included in *table.c*, where the macro *_TABLE* is defined. Including *glo.h* in other C source files makes the variables in *table.c* known to the other modules in the kernel. *Held_head* and *held_rail* (lines 5013 and 5014) are pointers to a queue of pending interrupts. *Proc_ptr* (line 5018) points to the process table entry for the current process. When a system call or interrupt occurs, it tells where to store the registers and processor state. *Sig_procs* (line 5021) counts the number of processes that have signals pending that have not yet been sent to the memory manager for processing. A few items in *glo.h* are defined with *extern* instead of *EXTERN*. These include *sizes*, an array filled in by the boot monitor, the task table, *tasktab*, and the task stack, *t_stack*. The last two are **initialized variables**, a feature of the C language. The use of the *EXTERN* macro is not compatible with C-style initialization, since a variable can only be initialized once.

Each task has its own stack within *t_stack*. During interrupt handling, the kernel uses a separate stack, but it is not declared here, since it is only accessed by the assembly language level routine that handles interrupt processing, and does not need to be known globally.

There are two more kernel header files that are widely used, although not so much that they are included in *kernel.h*. The first of these is *proc.h* (line 5100), which defines a process table entry as a struct *proc* (lines 5110 to 5148). Later on in the same file, it defines the process table itself as an array of such structs, *proc*[*NR_TASKS* + *NR_PROCS*] (line 5186). In the C language this reuse of a name is permitted. The macro *NR_TASKS* is defined in *include/minix/const.h* (line 2953) and *NR_PROCS* is defined in *include/minix/config.h* (line 2639). Together these set the size of the process table. *NR_PROCS* can be changed to create a system capable of handling a larger number of users. Because the process table is accessed frequently, and calculating an address in an array requires slow multiplication operations, an array of pointers to the process table elements, *pproc_addr* (line 5187), is used to allow speedy access.

Each table entry contains storage for the process' registers, stack pointer. state, memory map, stack limit, process id, accounting, alarm time, and message information. The first part of each process table entry is a *stackframe_s* structure. A process is put into execution by loading its stack pointer with the address of its process table entry and popping all the CPU registers from this structure. When a process cannot complete a SEND because the destination is not waiting, the sender is put onto a queue pointed to by the destination's *p_callerq* field (line 5137). That way, when the destination finally does a RECEIVE, it is easy to find all the processes wanting to send to it. The *p_sendlink* field (line 5138) is used to link the members of the queue together.

When a process does a RECEIVE and there is no message waiting for it, it blocks and the number of the process it wants to RECEIVE from is stored in *p_getfrom*. The address of the message buffer is stored in *p_messbuf*. The last three fields in each process table slot are *p_nextready*, *p_pending*, and *p_pendcount* (lines 5143 to 5145). The first is of these used to link processes together on the scheduler queues, and the second is a bit map used to keep track of signals that have not yet been passed to the memory manager (because the memory manager is not waiting for a message). The last field is a count of these signals.

The flag bits in *p_flags* define the state of each table entry. If any of the bits is set, the process cannot be run. The various flags are defined and described on lines 5154 to 5160. If the slot is not in use, *P_SLOT_FREE* is set. After a *FORK*, *NO_MAP* is set to

prevent the child process from running until its memory map has been set up. *SENDING* and *RECEIVING* indicate that the process is blocked trying to send or receive a message. *PENDING* and *SIG_PENDING* indicate that signals have been received, and *P_STOP* provides support for tracing, during debugging.

The macro *proc_addr* (line 5179) is provided because it is not possible to have negative subscripts in C. Logically, the array *proc* should go from -*NR_TASKS* to +*NR_PROCS*. Unfortunately, in C it must start at 0, so *proc*[0] refers to the most negative task, and so forth. To make it easier to keep track of which slot goes with which process, we can write

```
rp = procf_addr(n);
```

to assign to *rp* the address of the process slot for process n, either positive or negative.

Bill_ptr (line 5191) points to the process being charged for the CPU. When a user process calls the file system, and the file system is running, *proc_ptr* (in glo.h) points to the file system process. However, *bill_ptr* will point to the user making the call, since CPU time used by the file system is charged as system time to the caller.

The two arrays *rdy_head* and *rdy_tail* are used to maintain the scheduling queues. The first process on, for example, the task queue is pointed to by *rdy_head*[*TASK_Q*].

Another header that is included in a number of different source files is *protect.h* (line 5200). Almost everything in this file deals with architecture details of the lntel processors that support protected mode (the 80286, 80386, 80486, Pentium, and Pentium Pro). A detailed description of these chips is beyond the scope of this book. Suffice it to say that they contain internal registers that point to **descriptor tables** in memory. Descriptor tables define how system resources are used and prevent processes from accessing memory assigned to other processes. In addition the processor architecture provides for four **privilege levels**, of which MINIX takes advantage of three. These are defined symbolically on lines 5243 to 5245. The most central parts of the kernel, the parts that run during interrupts and that switch processes, run with *INTR_PRIVILEGE*. There is no part of memory or register in the CPU that cannot be accessed by a process with this privilege level. The tasks run at *TASK_PRIVILEGE* level, which allows them to access I/O but not to use instructions that modify special registers, like those that point to descriptor tables. Servers and user processes run at *USER_PRIVILEGE* level. Processes executing at this level are unable to execute certain instructions, for instance those that access I/O ports, change memory assignments, or change privilege levels themselves. The concept of privilege levels will be familiar to those who are familiar with the architecture of modern CPUs, but those who have learned computer architecture through study of the assembly language of low-end microprocessors may not have encountered such restrictions.

There are several other header files in the kernel directory, but we will mention only two more here. First, there is *sconst.h* (line 5400), which contains constants, used by assembler code. These are all offsets into the *stackframe_s* structure portion of a process table entry, expressed in a form usable by the assembler. Since assembler code is not processed by the C compiler, it is simpler to have such definitions in a separate file. Also, since these definitions are all machine dependent, isolating them here simplifies the process of porting MINIX to another processor which will need a different version of *sconst.h*. Note that many offsets are expressed as the previous value plus *W*, which is set equal to the word size at line 5401. This allows the same file to serve for compiling a 16-bit or 32-bit version of MINIX.

There is a potential problem here. Header files are supposed to allow one to provide a single correct set of definitions and then proceed to use them in many places without devoting a lot of further attention to the details. Obviously, duplicate definitions, like

those in *sconst.h*, violate that principle. This is a special case, of course, but as such, special attention is required if changes are made either to this file or to *proc.h*, to ensure the two files are consistent.

The final header we will mention here is *assert.h* (line 5500). The POSIX standard requires the availability of an *assert* function, which can be used to make a run-time test and abort a program, printing a message. In fact, POSIX requires that an *assert.h* header be provided in the *include/* directory, and one is provided there. So why is there another version here? The answer is that when something goes wrong in a user process, the operating system can be counted upon to provide services such as printing a message to the console. But if something goes wrong in the kernel itself, the normal system resources cannot be counted upon. The kernel thus provides its own routines to handle *assert* and print messages, independently of the versions in the normal system library.

There are a few header files in *kernel/* we have not discussed yet. They support the I/O tasks and will be described in the next chapter where they are relevant. Before passing on to the executable code, however, let us look at *table.c* (line 5600), whose compiled object file will contain all the kernel data structures. We have already seen many of these data structures defined, in *glo.h* and *proc.h*. On line 5625 the macro *_TABLE* is defined, immediately before the `#include` statements. As explained earlier, this definition causes *EXTERN* to become defined as the null string, and storage space to be allocated for all the data declarations preceded by *EXTERN*. In addition to the structures in *glo.h* and *proc.h*, storage for a few global variables used by the terminal task, defined in *tty.h*, is also allocated here.

In addition to the variables declared in header files there are two other places where global data storage is allocated. Some definitions are made directly in *table.c*. On lines 5639 to 5674 stack space is allocated for each task. For each optional task the corresponding *ENABLE_XXX* macro (defined in the file *include/minix/config.h*) is used to calculate the stack size. Thus no space is allocated for a task that is not enabled. Following this, the various *ENABLE_XXX* macros are used to determine whether each optional task will be represented in the *tasktab* array, composed of *tasktab* structures, as declared earlier in *src/kernel/type.h* (lines 5699 to 5731). There is an element for each process that is started during system initialization, whether task, server, or user process (i.e., init). The array index implicitly maps between task numbers and the associated startup procedures. *Tasktab* also specifies the stack space needed for each process and provides an identification string for each process. it has been put here rather than in a header file because the trick with *EXTERN* used to prevent multiple declarations does not work with initialized variables; that is, you may not say

```
extern int x = 3;
```

anywhere. The previous definitions of stack size also permit allocation of stack space for all of the tasks on line 5734.

Despite trying to isolate all user-settable configuration information in *include/minix/config.h*, an error is possible in matching the size of the *tasktab* array to *NR_TASKS*. At the end of *table.c* a test is made for this error, using a little trick. The array *dummy_tasktab* is declared here in such a way that its size will be impossible and will trigger a compiler error if a mistake has been made. Since the dummy array is declared as *extern*, no space is allocated for it here (or anywhere). Since it is not referenced anywhere else in the code, this will not bother the compiler.

The other place where global storage is allocated is at the end of the assembly language file *mpx386.s* (line 6483). This allocation, at the label *_sizes*, puts a magic number

(to identify a valid MINIX kernel) at the very beginning of the kernel's data segment. Additional space is allocated here by the `.space` pseudo-instruction. Reservation of storage in this way by the assembly language program makes it possible to force the *_sizes* array to be physically located at the beginning of the kernel's data segment, making it easy to program boot to put the data in the right place. The boot monitor reads the magic number and, if it is correct, overwrites it to initialize the *_sizes* array with the sizes of different parts of the MINIX system. The kernel uses these data during initialization. At startup time, as far as the kernel is concerned, this is an initialized data area. However, the data the kernel eventually finds there are not available at compilation time. They are patched in by the boot monitor just before the kernel is started. This is all somewhat unusual normally one does not need to write programs that know about the internal structure of other programs. But the period of time after power is applied, but before the operating system is running, is nothing if not unusual and requires unusual techniques.

2.6.5 Bootstrapping MINIX

It is almost time to start looking at the executable code. But before we do that let us take a few moments to understand how MINIX is loaded into memory. It is of course, loaded from a disk. Figure 2-31 shows how diskettes and partitioned disks are laid out.

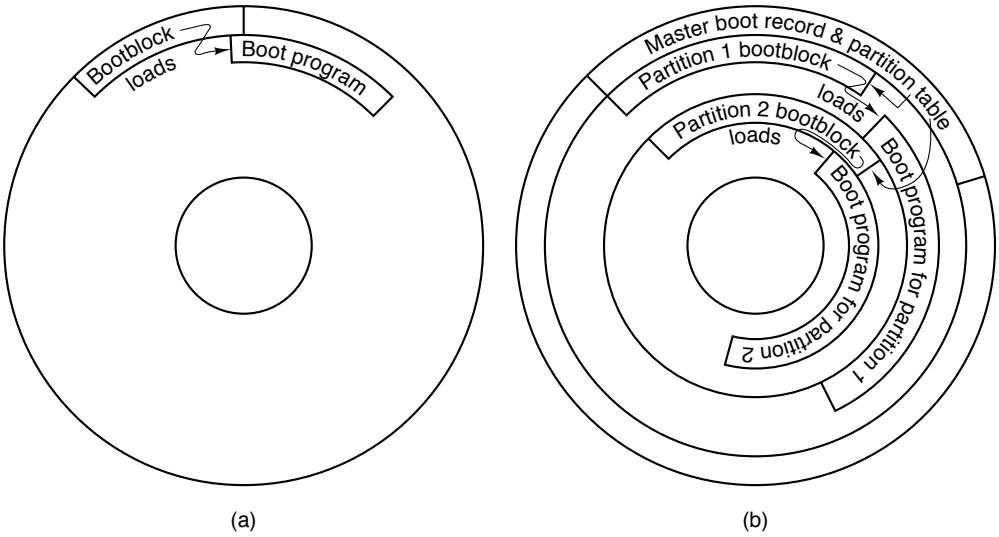


Figure 2-31. Disk structures used for bootstrapping. (a) Unpartitioned disk. The first sector is the boot block. (b) Partitioned disk. The first sector is the master boot record.

When the system is started, the hardware (actually, a program in ROM) reads the first sector of the boot disk and executes the code found there. On an unpartitioned MINIX diskette the first sector is a bootblock which loads the boot program, as in Fig. 2-31(a). Hard disks are partitioned, and the program on the first sector reads the partition table, which is also in the first sector, and loads and executes the first sector of the active partition, as shown in Fig. 2-31(b). (Normally one and only one partition in marked active). A MINIX partition has the same structure as an unpartitioned MINIX diskette, with a bootblock that loads the boot program.

The actual situation can be a little more complicated than the figure shows, because a partition may contain subpartitions. In this case the first sector of the partition will be

another master boot record containing the partition table for the subpartitions. Eventually, however, control will be passed to a boot sector, the first sector on a device that is not further subdivided. On a diskette the first sector is always a boot sector. MINIX does allow a form of partitioning of a diskette, but only the first partition may be booted; there is no separate master boot record, and subpartitions are not possible. This makes it possible for partitioned and non-partitioned to be mounted in exactly the same way. The main use for a partitioned floppy disk is that it provides a convenient say to divide an installation disk into a root image to be copied to a RAM disk and mounted portion that can be dismounted when no longer needed, in order to free the diskette drive for continuing the installation process.

The MINIX boot sector is modified at the time it is written to the disk by patching in the sector numbers needed to find a program called *boot* on its partition or subpartition. This patching is necessary because previous to loading the operating system there is no way to use the directory and file names to find a file. A special program called *installboot* is used to do the patching and writing of the boot record. *Boot* is the secondary loader for MINIX. It can do more than just load the operating system however, as it is a **monitor program** that allows the user to change, set and save various parameters. *Boot* looks in the second sector of its partition to find a set of parameters to use. MINIX, like standard UNIX, reserves the first 1K block of every disk device as a **bootblock**, but only one 512-byte sector is loaded by the ROM boot loader or the master boot sector, so 512 bytes are available for saving settings. These control the boot operation, and are also passed to operating system itself. The default settings present a menu with one choice, to start MINIX, but the settings can be modified to present a more complex menu allowing other operating systems to be started (by loading and executing boot sectors from other partitions), or to start MINIX with various options. The default settings can also be modified to bypass the menu and start MINIX immediately.

Boot is not a part of the operating system, but it is smart enough to use the file system data structures to find the actual operating system image. By default, *boot* looks for a file called */minix*, or if there is a */minix/* directory, for the newest file within it, but the boot parameters can be changed to look for a file with any name. This degree of flexibility is unusual, and most operating systems have predefined file name for the system image. But, MINIX is an unusual operating system that encourages users to modify it and create experimental new versions. Prudence demands that users who do this should have a way to select multiple versions, in order to be able to return to the last version that worked correctly when an experiment fails.

The MINIX image loaded by *boot* is nothing more than a concatenation of the individual files produced by the compiler when the kernel, memory management, file system, and *init* programs are compiled. Each of these includes a short header of the type defined in *include/a.out.h*, and from the information in the header of each part, *boot* determines how much space to reserve for uninitialized data after loading the executable code and the initialized data for each part, so the next part can be loaded at the proper address. The *_sizes* array mentioned in the previous section also receives a copy of this information so the kernel itself can have access to the locations and sizes of all the modules loaded by *boot*. The regions of memory available for loading the bootsector, *boot* itself, and MINIX will depend upon the hardware. Also, some machine architectures may require adjustment of internal addresses within the executable code to correct them for the actual address where a program is loaded. The segmented architecture of Intel processors makes this unnecessary. Since details of the loading process differ with machine type, and *boot* it not itself part of the operating system, we will not discuss it further here. The important

thing is that by one means or another the operating system is loaded into memory. Once the loading is complete, control passes to the executable code of the kernel.

As an aside, we should mention that operating systems are not universally loaded from local disks. **Diskless workstations** may load their operating systems from a remote disk, over a network connection. This requires network software in ROM, of course. Although details vary from what we have described here, the elements of the process are likely to be similar. The ROM code must be just smart enough to get an executable file over the net that can then obtain the complete operating system. If MINIX were loaded this way, very little would need to be changed in the initialization process that occurs once the operating system code is loaded into memory. It would, of course, need a network sever and a modified file system that could access files via the network.

2.6.6 System Initialization

MINIX for IBM PC-type machines can be compiled in l6-bit mode if compatibility with older processor chips is required, or in 32-bit mode for better performance on 80386+ processors. The same C source code is used and the compiler generates the appropriate output depending upon whether the compiler itself is the 16-bit or 32-bit version of the compiler. A macro defined by the compiler itself determines the definition of the *_WORD_SIZE* macro in *include/minix/config.h*. The first part of MINIX to execute is written in assembly language, and different source code files must be used for the 16-bit or 32-bit compiler. The 32-bit version of the initialization code is in *mpx386.s*. The alternative, for 16-bit systems, is in *mpx88.s*. Both of these also include assembly language support for other low-level kernel operations. The selection is made automatically in *mpx.s*. This file is so short that the entire file can be presented in Fig. 2-32.

```
#include <minix/config.h>
#if _WORD_SIZE == 2
#include "mpx88.s"
#else
#include "mpx386.s"
#endif
```

Figure 2-32. How alternative assembly language source tiles are selected.

Mpx.s shows an unusual use of the C preprocessor `#include` statement. Customarily `#include` is used to include header files, but it can also be used to select an alternate section of source code. Using `#if` statements to do this would require putting all the code in both of the large files *mpx88.s* and *mpx386.s* into a single file. Not only would this be unwieldy; it would also be wasteful of disk space, since in a particular installation it is likely that one or the other of these two files will not be used at all and can be archived or deleted. In the following discussion we will use the 32-bit *mpx386.s* as our example.

Since this is our first look at executable code, let us start with a few words about how we will do this throughout the book. The multiple source files used in compiling a large C program can be hard to follow. In general, we will keep discussions confined to a single file at a time, and we will go in order through the files. We will start with the entry point for each part of the MINIX system, and we will follow the main line of execution. When a call to a supporting function is encountered, we will say a few words about the purpose of the call, but normally we will not go into a detailed description of

the internals of the function at that point, leaving that until we arrive at the definition of the called function. Important subordinate functions are usually defined in the same file in which they are called, following the higher-level calling functions, but small or general-purpose functions are sometimes collected in separate files. Also, an attempt has been made to put machine-dependent code in separate files from machine-independent code to facilitate portability to other platforms. A substantial amount of effort has been made to organize the code, and, in fact, many files were rewritten in the course of writing this text in order to organize them better for the reader. But a large program has many branches, and sometimes understanding a main function requires reading the functions it calls, so having a few slips of paper to use as bookmarks and deviating from our order of discussion to look at things in a different order may be helpful at times.

Having laid out our intended way of organizing the discussion of the code, we must start off by immediately justifying a major exception. The startup of MINIX involves several transfers of control between the assembly language routines in *mpx386.s* and routines written in C and found in the files *start.c* and *main.c*. We will describe these routines in the order that they are executed, even though that involves jumping from one file to another.

Once the bootstrap process has loaded the operating system into memory, control is transferred to the label *MINIX* (in *mpx386.s*, line 6051). The first instruction is a jump over a few bytes of data; this includes the boot monitor flags (line 6054), used by the boot monitor to identify various characteristics of the kernel, most importantly, whether it is a 16-bit or 32-bit system. The boot monitor always starts in 16-bit mode, but switches the CPU to 32-bit mode if necessary. This happens before control passes to *MINIX*. The monitor also sets up a stack. There is a substantial amount of work to be done by the assembly language code, setting up a stack frame to provide the proper environment for code compiled by the C compiler, copying tables used by the processor to define memory segments, and setting various processor registers. As soon as this work is complete, the initialization process continues by calling (at line 6109) the C function *cstart*. Note that it is referred to as *_cstart* in the assembly language code. This is because all functions compiled by the C compiler have an underscore prepended to their names in the symbol tables, and the linker looks for such names when separately compiled modules are linked. Since the assembler does not add underscores, the writer of an assembly language program must explicitly add one in order for the linker to be able to find a corresponding name in the object file compiled by the C compiler. *Cstart* calls another routine to initialize the **Global Descriptor Table**, the central data structure used by Intel 32-bit processors to oversee memory protection, and the **Interrupt Descriptor Table**, used to select the code to be executed for each possible interrupt type. Upon returning from *cstart* the `lgdt` and `lidt` instructions (lines 6115 and 6116) make these tables effective by loading the dedicated registers by which they are addressed. The following instruction,

```
jmpf CS_SELECTOR:csinit
```

looks at first glance like a no-operation, since it transfers control to exactly where control would be if there were a series of `nop` instructions in its place. But this is an important pan of the initialization process. This jump forces use of the structures just initialized. After some more manipulation of the processor registers, *MINIX* terminates with a jump (not a call) at line 613 to the kernel's *main* entry point (in *main.c*). At this point the initialization code in *mpx386.s* is complete. The rest of the file contains code to start or restart a task or process, interrupt handlers, and other support routines that had to be written in assembly language for efficiency. We will return to these in the next section.

We will now look at the top-level C initialization functions. The general strategy is to do as much as possible using high-level C code. There are already two versions of the *mpx* code, as we have seen, and anything that can be off-loaded to C code eliminates two chunks of assembler code. Almost the first thing done by *cstart* (in *start.c*, line 6524) is to set up the CPU's protection mechanisms and the interrupt tables, by calling *prot_init*. Then it does such things as copying the boot parameters to the kernel's part of memory and converting them into numeric values. It also determines the type of video display, size of memory, machine , type, processor operating mode (real or protected), and whether a return to the boot monitor is possible. All information is stored in appropriate global variables, for access when needed by any part of the kernel code.

Main (in *main.c*, line 6721), completes initialization and then starts normal execution of the system. It configures the interrupt control hardware by calling *intr_init*. This is done here because it can not be done until the machine type is known, and the procedure is in a separate file because it is so dependent upon the hardware. The parameter (1) in the call tells *intr_init* that it is initializing for MINIX. With a parameter (0) it can be called to reinitialize the hardware to the original state. The call to *intr_init* also takes two steps to insure that any interrupts that occur before initialization is complete have no effect. First a byte is written to each interrupt controller chip that inhibits response to external input. Then all entries in the table used to access device-specific interrupt handlers are filled in with the address of a routine that will harmlessly print a message if a spurious interrupt is received. Later these table entries will be replaced, one by one, with pointers to the handler routines, as each of the I/O tasks runs its own initialization routine. Each task then will reset a bit in the interrupt controller chip to enable its own interrupt input.

Mem_init is called next. It initializes an array that defines the location and size of each chunk of memory available in the system. As with the initialization of the interrupt hardware, the details are hardware-dependent and isolation of *mem_init* as a function in a separate file keeps main itself free of code that is not portable to different hardware.

The largest part of main's code is devoted to setup of the process table, so that when the first tasks and processes are scheduled, their memory maps and registers will be set correctly. All slots in the process table are marked as free, and the *pproc_addr* array that speeds access to the process table is initialized by the loop on lines 6745 to 6749. The code on line 6748,

```
(pproc_addr + NR_TASKS)[t] = rp;
```

could just as well have been defined as

```
pproc_addr[t + NR_TASKS] = rp;
```

because in the C language $a[i]$ is just another way of writing $*(a+i)$. So it does not make much difference if you add a constant to a or to i. Some C compilers generate slightly better code if you add a constant to the array instead of the index.

The largest part of *main*, the long loop on lines 6762 to 6815, initializes the process table with the necessary information to run the tasks, servers, and *init*. All of these processes must be present at startup time and none of them will terminate during normal operation. At the start of the loop, *rp* is assigned the address of a process table entry (line 6763). Since *rp* is a pointer to a structure, the elements of the structure can be accessed using notation like *rp-¿p_name*, as is done on line 6765. This notation is used extensively in the MINIX source code.

The tasks, of course, are all compiled into the same file as the kernel, and the information about their stack requirements is in the *tasktab* array defined in *table.c*. Since tasks

are compiled into the kernel and can call code and access data located anywhere in the kernel's space, the size of an individual task is not meaningful, and the size field for each of them is filled with the sizes for the kernel itself. The array *sizes* contains the text and data sizes in clicks of the kernel, memory manager, file system, and *init*. This information is patched into the kernel's data area by boot before the kernel starts executing and appears to the kernel as if the compiler had provided it. The first two elements of *sizes* are the kernel's text and data sizes; the next two are the memory manager's, and so on. If any of the four programs does not use separate I and D space, the text size is 0 and the text and data are lumped together as data. Assigning *sizeindex* a value of zero (line 6775) for each of the tasks assures that the zeroth element of *sizes* at lines 6783 and 6784 will be accessed for all of the tasks. The assignment to *sizeindex* at line 6778 gives each of the servers and *init* its own index into *sizes*.

The design of the original IBM PC placed read-only memory at the top of the usable range of memory, which is limited to 1 MB on an 8088 CPU. Modern PC-compatible machines always have more memory than the original PC, but for compatibility they still have read-only memory at the same addresses as the older machines. Thus, the read-write memory is discontinuous, with a block of ROM between the lower 640 KB and the upper range above 1 MB. The boot monitor loads the servers and *init* into the memory range above the ROM if possible. This is primarily for the benefit of the file system, so a very large block cache can be used without bumping into the read-only memory. The conditional code at lines 6804 to 6810 ensures that this use of the high memory area is recorded in the process table.

Two entries in the process table correspond to processes that do not need to be scheduled in the ordinary way. These are the *IDLE* and *HARDWARE* processes. *IDLE* is a do-nothing loop that is executed when there is nothing else ready to run, and the *HARDWARE* process exists for bookkeeping purposes—it is credited with the time used while servicing an interrupt. All other processes are put on the appropriate queues by the code in line 6811. The function called, *lock_ready*, sets a lock variable, *switching*, before modifying the queues and then removes the lock, when the queue has been modified. The locking and unlocking are not required at this point, when nothing is running yet, but this is the standard method, and there is no point in creating extra code to be used just once.

The last step in initializing each slot in the process table is to call *alloc_segments*. This procedure is part of the system task, but of course no tasks are running yet, and it is called as an ordinary procedure at line 6814. It is a machine-dependent routine that sets into the proper fields the locations, sizes, and permission levels for the memory segments used by each process. For older Intel processors that do not support protected mode, it defines only the segment locations. It would have to be rewritten to handle a processor type with a different method of allocating memory.

Once the process table is initialized for all the tasks, the servers, and *init*, the system is almost ready to roll. The variable *bill_ptr* tells which process gets billed for processor time; it needs to have an initial value set at line 6818, and *IDLE* is an appropriate choice. Later on it may be changed by the next function called, *lock_pick_proc*. All of the tasks are now ready to run and *bill_ptr* will be changed when a user process runs. *Lock_pick_proc*'s other job is to make the variable *proc_ptr* point to the entry in the process table for the next process to be run. This selection is made by examining the task, server, and user process queues, in that order. In this case, the result is to point *proc_ptr* to the entry point for the console task, which is always the first one to be started.

Finally, *main* has run its course. In many C programs *main* is a loop, but in the MINIX kernel its job is done once the initialization is complete. The call to *restart* on line 6822

starts the first task. Control will never return to *main*.

_Restart is an assembly language routine in *mpx386.s*. In fact, *_restart* is not a complete function; it is an intermediate entry point in a larger procedure. We will discuss it in detail in the next section; for now we will just say that *_restart* causes a context switch, so the process pointed to by *proc_ptr* will run. When *_restart* has executed for the first time we can say that MINIX is running—it is executing a process. *_Restart* is executed again and again as tasks, servers, and user processes are given their opportunities to run and then are suspended, either to wait for input or to give other processes their turns.

The task queued first (the one using slot 0 of the process table, that is, the one with the most negative number) is always the console task, so other tasks can use it to report progress or problems as they start. It runs until it blocks trying to receive a message. Then the next task will run until it, too, blocks trying to receive a message. Eventually, all the tasks will be blocked, so the memory manager and file system can run. Upon running for the first time, each of these will do some initialization, but both of them will eventually block, also. Finally *init* will fork off a *getty* process for each terminal. These processes will block until input is typed at some terminal, at which point the first user can log in.

We have now traced the startup of MINIX through three files, two written in C and one in assembly language. The assembly language file, *mpx386.s*, contains additional code used in handling interrupts, which we will look at in the next section. However, before we go on let us wrap up with a brief description of the remaining routines in the two C files. The other procedures in *start.c* are *k_atoi* (line 6594), which converts a string to an integer, and *k_getenv* (line 6606), which is used to find entries in the kernel's environment, which is a copy of the boot parameters. These are both simplified versions of standard library functions which are rewritten here in order to keep the kernel simple. The only remaining procedure in *main.c* is *panic* (line 6829). It is called when the system has discovered a condition that makes it impossible to continue. Typical panic conditions are a critical disk block being unreadable, an inconsistent internal state being detected, or one part of the system calling another part with invalid parameters. The calls to *printf* here are actually calls to the kernel routine *printk*, so the kernel can print on the console even if normal interprocess communication is disrupted.

2.6.7 Interrupt Handling in MINIX

The details of interrupt hardware are system dependent, but any system must have elements functionally equivalent to those to be described for systems with 32-bit Intel CPUs. Interrupts generated by hardware devices are electrical signals and are handled in the first place by an interrupt controller, an integrated circuit that can sense a number of such signals and for each one generate a unique data pattern on the processor's data bus. This is necessary because the processor itself has only one input for sensing all these devices, and thus cannot differentiate which device needs service. PCs using Intel 32-bit processors are normally equipped with two such controller chips. Each can handle eight inputs, but one is a slave which feeds its output to one of the inputs of the master, so fifteen distinct external devices can be sensed by the combination, as shown in Fig. 2-33.

In the figure, interrupt signals arrive on the various *IRQ n* lines shown at the right. The connection to the CPU's INT pin tells the processor that an interrupt has occurred. The INTA (interrupt acknowledge) signal from the CPU causes the controller responsible for the interrupt to put data on the system data bus telling the processor which service routine to execute. The interrupt controller chips are programmed during system initialization, when *main* calls *intr_init*. The programming determines the output sent to the CPU for

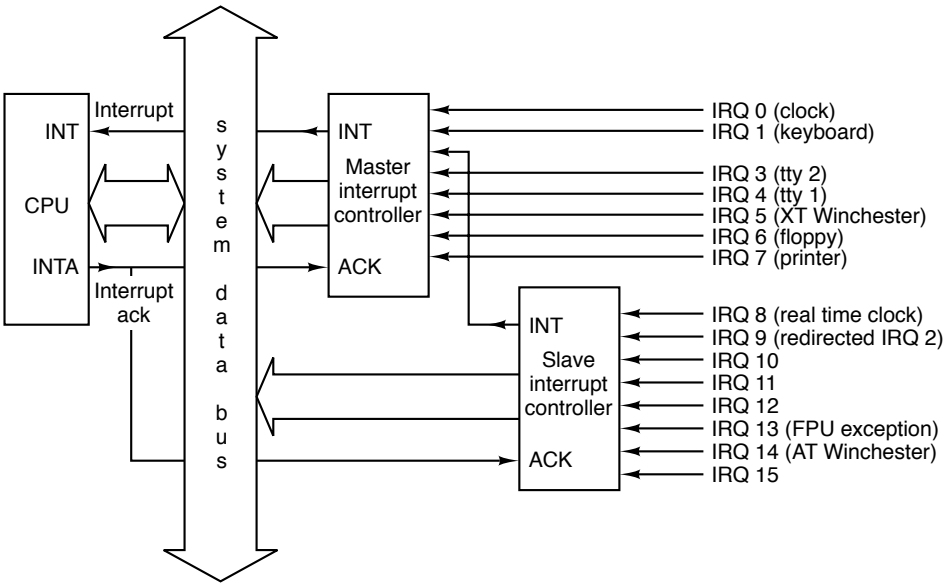


Figure 2-33. Interrupt processing hardware on a 32-bit lntel PC.

a signal received on each of the input lines, as well as various other parameters of the controller's operation. The data put on the bus is an 8-bit number, used to index into a table of up to 256 elements. The MINIX table has 56 elements. Of these, 35 are actually used; the others are reserved for use with future Intel processors or for future enhancements to MINIX. On 32-bit Intel processors this table contains interrupt gate descriptors, each of which is an 8-byte structure with several fields.

There are several possible modes of response to interrupts; in the one used by MINIX the fields of most concern to us in each of the interrupt gate descriptors point to the service routine's executable code segment and the starting address within it. The CPU executes the code pointed to by the selected descriptor. The result is exactly the same as execution of an

```
int nnn
```

assembly language instruction. The only difference is that in the case of a hardware interrupt the *nnn* originates from a register in the interrupt controller chip, rather than from an instruction in program memory.

The task-switching mechanism of a 32-bit Intel processor that is called into play in response to an interrupt is complex, and changing the program counter to execute another function is only a part of it. When the CPU receives an interrupt while running a process it sets up a new stack for use during the interrupt service. The location of this stack is determined by an entry in the **Task State Segment** (TSS). There is one such structure for the entire system, initialized by *cstart*'s call to *prot_init*, and modified as each process is started. The effect is that the new stack created by an interrupt always starts at the end of the *stackframe_s* structure within the process table entry of the interrupted process. The CPU automatically pushes several key registers onto this new stack, including those necessary to reinstate the interrupted process' own stack and restore its program counter. When the interrupt handler code starts running, it uses this area in the process table as its stack, and much of the information needed to return to the interrupted process will have already been stored. The interrupt handler pushes the contents of additional registers, filling the stackframe, and then switches to a stack provided by the kernel while it does whatever must be done to service the interrupt.

Termination of an interrupt service routine is done by switching the stack from the kernel stack back to a stackframe in the process table (but not necessarily the same one that was created by the last interrupt), explicitly popping the additional registers. and executing an `iretd` (return from interrupt) instruction. `Iretd` restores the state that existed before an interrupt, restoring the registers that were pushed by the hardware and switching back to a stack that was in use before an interrupt. Thus an interrupt stops a process, and completion of the interrupt service restarts a process, possibly a different one from the one that was most recently stopped. Unlike the simpler interrupt mechanisms that are the usual subject of assembly language programming texts, nothing is stored on the interrupted process' working stack during an interrupt. Furthermore, because the stack is created anew in a known location (determined by the TSS) after an interrupt, control of multiple processes is simplified. To start a different process all that is necessary is to point the stack pointer to another process' stackframe, pop the registers that were explicitly pushed, and execute an `iretd` instruction.

The CPU disables all interrupts when it receives an interrupt. This guarantees that nothing can occur to cause the stackframe within a process table entry to overflow. This is automatic, but assembly-level instructions exist to disable and enable interrupts, as well. The interrupt handler reenables interrupts after switching to the kernel stack, located outside the process table. It must disable all interrupts again before it switches back to a stack within the process table, of course, but while it is handling an interrupt other interrupts can occur and be processed. The CPU keeps track of nested interrupts, and employs a simpler method of switching to an interrupt service routine and returning from one when an interrupt handler is interrupted. When a new interrupt is received while a handler (or other kernel code) is executing, a new stack is not created. Instead, the CPU pushes the essential registers needed for resumption of the interrupted code onto the existing stack. When an `iretd` is encountered while executing kernel code, a simpler return mechanism is used, too. The processor can determine how to handle the `iretd` by examining the code segment selector that is popped from the stack as part of the `iretd`'s action.

The privilege levels mentioned earlier control the different responses to interrupts received while a process is running and while kernel code (including interrupt service routines) is executing. The simpler mechanism is used when the privilege level of the interrupted code is the same as the privilege level of the code to be executed in response to the interrupt. It is only when the interrupted code is less privileged than the interrupt service code that the more elaborate mechanism, using the TSS and a new stack, is employed. The privilege level of a code segment is recorded in the code segment selector, and as this is one of the items stacked during an interrupt, it can be examined upon return from the interrupt to determine what the `iretd` instruction must do. Another service is provided by the hardware when a new stack is created to use while servicing an interrupt. The hardware checks to make sure the new stack is big enough for at least the minimum quantity of information that must be placed on it. This protects the more privileged kernel code from being accidentally (or maliciously) crashed by a user process making a system call with an inadequate stack. These mechanisms are built into the processor specifically for use in the implementation of operating systems that support multiple processes.

This behavior may be confusing if you are unfamiliar with the internal working of 32-bit Intel CPUs. Ordinarily we try to avoid describing such details, but understanding what happens when an interrupt occurs and when an `iretd` instruction is executed is essential to understanding how the kernel controls the transitions to and from the "running" state of Fig. 2-2. The fact that the hardware handles much of the work makes life much easier

for the programmer, and presumably makes the resulting system more efficient. All this help from the hardware does, however, make it hard to understand what is happening just by reading the software.

Only a tiny part of the MINIX kernel actually sees hardware interrupts. This code is in *mpx386.s*. There is an entry point for each interrupt. The source code at each entry point, *_hwint00* to *_hwint07*, (lines 6164 to 6193) looks like a call to *hwint_master* (line 6143), and the entry points *_hwint08* to *_hwint15* (lines 6222 to 6251) look like calls to *hwint_slave* (line 6199). Each entry point appears to pass a parameter in the call, indicating which device needs service. In fact, these are really not calls, but macros, and eight separate copies of the code defined by the macro definition of *hwint_master* are assembled, with only the *irq* parameter different. Similarly, eight copies of the *hwint_slave* macro are assembled. This may seem extravagant, but assembled code is very compact. The object code for each expanded macro occupies less than 40 bytes. In servicing an interrupt, speed is important, and doing it this way eliminates the overhead of executing code to load a parameter, call a subroutine, and retrieve the parameter.

We will continue the discussion of *hwint_master* as if it really were a single function, rather than a macro that is expanded in eight different places. Recall that before *hwint_master* begins to execute, the CPU has created a new stack in the interrupted process' *stackframe_s*, within its process table slot, and that several key registers have already been saved there. The first action of *hwint_master* is to call save (line 6144). This subroutine pushes all the other registers necessary to restart the interrupted process. *Save* could have been written inline as part of the macro to increase speed, but this would have more than doubled the size of the macro, and in any case *save* is needed for calls by other functions. As we shall see, *save* plays tricks with the stack. Upon returning to *hwint_master*, the kernel stack, not a stackframe in the process table, is in use. The next step is to manipulate the interrupt controller, to prevent it from receiving another interrupt from the source that generated the current interrupt (lines 6145 to 6147). This operation masks the ability of the controller chip to respond to a particular input; the CPU's ability to respond to all interrupts is inhibited internally when it first receives the interrupt signal and has not yet been restored at this point.

The code on lines 6148 to 6150 resets the interrupt controller and then enables the CPU to again receive interrupts from other sources. Next, the number of the interrupt being serviced is used by the indirect `call` instruction on line 6152 to index into a table of addresses of the device-specific lowe-level routines. We call these low-level routines, but they are written in C, and they typically perform operations like servicing an input device and transferring the data to a buffer where it can be accessed when the corresponding task has its next chance to run. A substantial amount of processing may happen before the return from this call.

We will see examples of low-level driver code in the next chapter. However, in order to understand what is happening here in *hwint_master*, we now mention that the low-level code may call *interrupt* (in *proc.c*, which we will discuss in the next section), and that *interrupt* transforms the interrupt into a message to the task that services the device that caused the interrupt. Furthermore, a call to *interrupt* invokes the scheduler and may select this task to run next. Upon returning from the call to the device-specific code, the processor's ability to respond to all interrupts is again disabled, by the `cli` instruction on line 6154, and the interrupt controller is prepared to be able to respond to the particular device that caused the current interrupt when all interrupts are next reenabled (lines 6157 to 6159). Then *hwint_master* terminates with a `ret` instruction (line 6160). It is not obvious that something tricky happens here. If a process was interrupted, the stack in use

at this point is the kernel stack, and not the stack within a process table that was set up by the hardware before *hwint_master* was started. In this case, manipulation of the stack by *save* will have left the address of *_resrart* on the kernel stack. This results in a task, server, or user process once again executing. It may not be, and in fact is unlikely to be, the same process as was executing originally. This depends upon whether the processing of the message created by the device-specific interrupt service routine caused a change in the process scheduling queues. This, then, is the heart of the mechanism which creates the illusion of multiple processes executing simultaneously.

To be complete, let us mention that when an interrupt occurs while kernel code is executing, the kernel stack is already in use, and save leaves the address of *restart1* on the kernel stack. In this case, whatever the kernel was doing previously continues after the `ret` at the end of *hwint_master*. Thus interrupts may be nested, but when all the low-level service routines are complete *_restart* will finally execute, and a process different from the one that was interrupted may be put into execution.

Hwint_slave (line 6199) is very similar to *hwint_master*, except that it must reenable both the master and slave controllers, since both of them are disabled by receipt of an interrupt by the slave. There are a few subtle aspects of assembly language to be seen here. First, on line 6206 there is a line

```
jmp .+2
```

which specifies a jump whose target address is the immediately following instruction. This instruction is placed here solely to add a small delay. The authors of the original IBM PC BIOS considered a delay necessary between consecutive I/O instructions, and we are following their example, although it may not be necessary on all current IBM PC-compatible computers. This kind of fine tuning is one reason why programming hardware devices is considered an esoteric craft by some. On line 6214 there is a conditional jump to an instruction with a numeric label,

```
0: ret
```

to be found on line 6218. Note that the line

```
jz 0f
```

does not specify a number of bytes to jump over, as in the previous example. The *0f* here is not a hexadecimal number. This is the way the assembler used by the MINIX compiler specifies a **local label**; the *0f* means a jump **forward** to the next numeric label 0. Ordinary label names are not permitted to begin with numeric characters. Another interesting and possibly confusing point is that the same label occurs elsewhere in the same file, on line 6160 in *hwint_master*. The situation is even more complicated than it looks at first glance since these labels are within macros and the macros are expanded before the assembler sees this code. Thus there are actually sixteen 0: labels in the code seen by the assembler. The possible proliferation of labels declared within macros is, indeed, the reason why the assembly language provides local labels; when resolving a local label the assembler uses the nearest one that matches in the specified direction, and additional occurrences of a local label are ignored.

Now let us move on to look at *save* (line 6261), which we have already mentioned several times. Its name describes one of its functions, which is to save the context of the interrupted process on the stack provided by the CPU, which is a stackframe within the process table. *Save* uses the variable *_k_reenter* to count and determine the level of nesting of interrupts. If a process was executing when the current interrupt occurred, the

```
mov esp, k_stktop
```

instruction on line 6274 switches to the kernel stack, and the following instruction pushes the address of *_restart* (line 6275). Otherwise, the kernel stack is already in use, and the address of *restart1* is pushed instead (line 6281). In either case, with a possibly different stack in use from the one that was in effect upon entry, and with the return address in the routine that called it buried beneath the registers that have just been pushed, an ordinary `return` instruction is not adequate for returning to the caller. The

```
jmp RETADR-P_STACKBASE(eax)
```

instructions that terminate the two exit points of *save*, at line 6277 and line 6282, use the address that was pushed when *save* was called.

The next procedure in *mpx386.s* is *_s_call*, which begins on line 6288. Before looking at its internal details, look at how it ends. There is no `ret` or `jmp` at its end. After disabling interrupts with the `cli` on line 6315, execution continues at *_restart*. *_S_call* is the system call counterpart of the interrupt handling mechanism. Control arrives at *_s_call* following a software interrupt, that is, execution of an `int` *nnn* instruction. Software interrupts are treated like hardware interrupts, except of course the index into the Interrupt Descriptor Table is encoded into the *nnn* part of an `int` *nnn* instruction, rather than being supplied by an interrupt controller chip. Thus, when *_s_call* is entered, the CPU has already switched to a stack inside the process table (supplied by the Task State Segment), and several registers have already been pushed onto this stack. By falling through to *_restart*, the call to *_s_call* ultimately terminates with an `iretd` instruction, and, just as with a hardware interrupt, this instruction will start whatever process is pointed to by *proc_ptr* at that point. Figure 2-34 compares the handling of a hardware interrupt and a system call using the software interrupt mechanism.

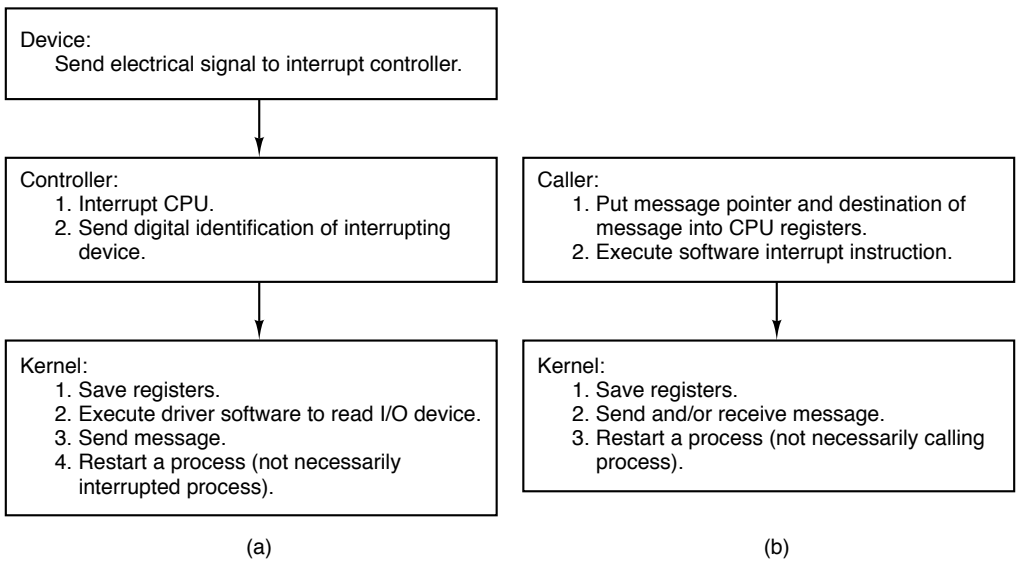


Figure 2-34. (a) How a hardware interrupt is processed. (b) How a system call is made.

Let us now look at some details of *_s_call*. The alternate label, *_p_s_call*, is a vestige of the 16-bit version of MINIX, which has separate routines for protected mode and real mode operation. In the 32-bit version all calls to either label end up here. A programmer invoking a MINIX system call writes a function call in C that looks like any other

function call, whether to a locally defined function or to a routine in the C library. The library code supporting a system call sets up a message, loads the address of the message and the process id of the destination into CPU registers, and then invokes an `int` *SYS386_VECTOR* instruction. As described above, the result is that control passes to the start of *_s_call*, and several registers have already been pushed onto a stack inside the process table.

The first part of the *_s_call* code resembles an inline expansion of *save* and saves the additional registers that must be preserved. Just as in *save*, a

```
mov esp, k_stktop
```

instruction then switches to the kernel stack, and interrupts are reenabled. (The similarity of a software interrupt to a hardware interrupt extends to both disabling all interrupts). Following this comes a call to *_sys_call*, which we will discuss in the next section. For now we just say that it causes a message to be delivered, and that this in turn causes the scheduler to run. Thus, when *_sys_call* returns, it is probable that *proc_ptr* will be pointing to a different process from the one that initiated the system call. Before execution falls through to *restart*, a `cli` instruction disables interrupts to protect the stackframe of the process that is about to be restarted.

We have seen that *_restart* (line 6322) is reached in several ways:

1. By a call from *main* when the system starts.

2. By a jump from *hwint_master* or *hwint_slave* after a hardware interrupt.

3. By falling through from *_s_call* after a system call.

In every case interrupts are disabled at this point. *_Restart* calls *unhold* if it detects that any unserviced interrupts have been held up because they arrived while other interrupts were being processed. This allows the other interrupts to be converted into messages before any process is restarted. This temporarily reenables interrupts, but they are disabled again before *unhold* returns. By line 6333 the next process to run has been definitively chosen, and with interrupts disabled it cannot be changed. The process table was carefully constructed so it begins with a stack frame, and the instruction on this line,

```
mov esp, (_proc_ptr)
```

points the CPU's stack pointer register at the stack frame. The

```
lldt P_LDT_SEL(esp)
```

instruction then loads the processor's local descriptor table register from the stack frame. This prepares the processor to use the memory segments belonging to the next process to be run. The following instruction loads the address in the next process' process table entry that where the stack for the next interrupt will be set up, and the following instruction stores this address into the TSS. The first part of *_restart* is not necessary after an interrupt that occurs when kernel code, (including interrupt service code) is executing, since the kernel stack will be in use and termination of the interrupt service should allow the kernel code to continue. The label *restart1* (line 6337) marks the point where execution resumes in this case. At this point *k_reenter* is decremented to record that one level of possibly nested interrupts has been disposed of, and the remaining instructions restore the

processor to the state it was in when the next process executed last. The penultimate instruction modifies the stack pointer so the return address that was pushed when save was called is ignored. If the last interrupt occurred when a process was executing, the final instruction, `iretd`, completes the return to execution of whatever process is being allowed to run next, restoring its remaining registers, including its stack segment and stack pointer. If, however, this encounter with the `iretd` came via *restart1*, the kernel stack in use is not a stackframe, but the kernel stack, and this is not a return to an interrupted process, but the completion of an interrupt that occurred while kernel code was executing. The CPU detects this when the code segment descriptor is popped from the stack during execution of the `iretd`, and the complete action of the `ireld` in this case is to retain the kernel stack in use.

There are a few more things to discuss in *mpx386.s*. In addition to hardware and software interrupts, various error conditions internal to the CPU can cause the initiation of an **exception**. Exceptions are not always bad. They can be used to stimulate the operating system to provide a service, such as providing more memory for a process to use, or swapping in a currently swapped-out memory page, although such services are not implemented in standard MINIX. But, when an exception occurs, it should not be ignored. Exceptions are handled by the same mechanism as interrupts, using descriptors in the interrupt descriptor table. These entries in the table point to the sixteen exception handler entry points, beginning with *_divide_error* and ending with *_copr_error*, found near the end of *mpx386.s*, on lines 6350 to 6412. These all jump to *exception* (line 6420) or *errexception* (line 6431) depending upon whether the condition pushes an error code onto the stack or not. The handling here in the assembly code is similar to that we have already seen, registers are pushed and the C routine *_exception* (note the underscore) is called to handle the event. The consequences of exceptions vary. Some are ignored, some cause panics, and some result in sending signals to processes. We will examine *_exception* in a later section.

There is one other entry point that is handled like an interrupt, *_level0_call* (line 6458). Its function will be discussed in the next section, when we discuss the code to which it jumps, *_level0_func*. The entry point is here in *mpx386.s* with the interrupt and exception entry points because it too is invoked by execution of an int instruction. Like the exception routines, it calls *save*, and thus eventually the code that is jumped to here will terminate by a `ret` that leads to *_restart*. The last executable function in *mpx386.s* is *idle_rask* (line 6465). This is a do-nothing loop that is executed whenever there is no other process ready to run.

Finally, some data storage space is reserved at the end of the assembly language file. There are two different data segments defined here. The

```
.sect .rom
```

declaration at line 6478 ensures that this storage space is allocated at the very beginning of the kernel's data segment. The compiler puts a magic number here so *boot* can verify that the file it loads is a valid kernel image. *Boot* then overwrites the magic number and subsequent space with the *_sizes* array data, as described in the discussion of kernel data structures. Enough space is reserved here for a *_sizes* array with a total of sixteen entries, in case additional servers are added to MINIX. The other data storage area defined at the

```
.sect .bss
```

(line 6483) declaration reserves space in the kernel's normal uninitialized variable area for the kernel stack and for variables used by the exception handlers. Servers and ordinary

processes have stack space reserved when an executable file is linked and depend upon the kernel to properly set the stack segment descriptor and the stack pointer when they are executed. The kernel has to do this for itself.

2.6.8 Interprocess Communication in MINIX

Processes in MINIX communicate by messages, using the rendezvous principle. When a process does a SEND, the lowest layer of the kernel checks to see if the destination is waiting for a message from the sender (or from ANY sender). If so, the message is copied from the sender's buffer to the receiver's buffer, and both processes are marked as runnable. If the destination is not waiting for a message from the sender, the sender is marked as blocked and put onto a queue of processes waiting to send to the receiver.

When a process does a RECEIVE, the kernel checks to see if any process is queued trying to send to it. If so, the message is copied from the blocked sender to the receiver, and both are marked as runnable. If no process is queued trying to send to it, the receiver blocks until a message arrives.

The high-level code for interprocess communication is found in *proc.c*. The kernel's job is to translate either a hardware interrupt or a software interrupt into a message. The former are generated by hardware and the latter are the way a request for system services, that is, a system call, is communicated to the kernel. These cases are similar enough that they could have been handled by a single function, but it was more efficient to create two specialized functions.

First we will look at *interrupt* (line 6938). It is called by the low-level interrupt service routine for a device after receipt of a hardware interrupt. *Interrupt*'s function is to convert the interrupt into a message for the task that handles the interrupting device, and typically very little processing is done before calling *interrupt*. For example, the entire low-level interrupt-handler for the hard disk driver consists of just these three lines:

```
w_status = in_byte(w_wn->base + REG_STATUS); /* acknowledge interrupt */
interrupt(WINCHESTER);
return 1;
```

If it were not necessary to read an I/O port on the hard disk controller to obtain the status, the call to *interrupt* could have been in *mpx386.s* instead of *at_wini.c*. The first thing *interrupt* does is check if an interrupt was already being serviced when the current interrupt was received, by looking at the variable *k_reenter* (line 6962). In this case the current interrupt is queued and *interrupt* returns. The current interrupt will be serviced later when *unhold* is called. The next action is to check whether the task is waiting for an interrupt (lines 6978 to 6981). If the task is not ready to receive, its *p_int_blocked* flag is set—we will see later that this makes it possible to recover the lost interrupt—and no message is sent. If this test is passed, the message is sent. Sending a message from *HARDWARE* to a task is simple, because the tasks and the kernel are compiled into the same file and can access the same data areas. The code on lines 6989 to 6992 sends the message, by filling in the destination task's message buffer source and type fields, resetting the destination's *RECEIVING* flag, and unblocking the task. Once the message is ready the destination task is scheduled to run. We will discuss scheduling in more detail in the next section, but the code in *interrupt* on lines 6997 to 7003 provides a preview of what we will see—this is an inline substitute for the *ready* procedure that is called to queue a process. It is simple here, since messages originating from interrupts go only to

tasks, and thus there is no need to determine which of the three process queues needs to be changed.

The next function in *proc.c* is *sys_call*. It has a similar function to *interrupt*: it converts a software interrupt (the `int` *SYS386_VECTOR* instruction by which a system call is initiated) into a message. But since there are a wider range of possible sources and destinations in this case, and since the call may require either sending or receiving or both sending and receiving a message, *sys_call* has more work to do. As is often the case, this means the code for *sys_call* is short and simple, since it does most of its work by calling other procedures. The first such call is to *isoksrc_dest*, a macro defined in *proc.h* (line.5172), which incorporates yet another macro, *isokprocn*, also defined in *proc.h* (line 5171). The effect is to check to make sure the process specified as the source or destination of the message is valid. At line 7026 a similar test, *isuserp* (also a macro defined in *proc.h*), is performed to make sure that if the call is from a user process it is asking to send a message and then receive a reply, the only kind of call permitted to user processes. Such errors are unlikely, but the tests are easily done, as ultimately they compile into code to perform comparisons of small integers. At this most basic level of the operating system testing for even the most unlikely errors is advisable. This code is likely to be executed many times each second during every second that the computer system on which it runs is active.

Finally, if the call requires sending a message, *mini_send* is called (line 7031), and if receiving a message is required, *mini_rec* is called (line 7039). These functions are the heart of the normal message passing mechanism of MINIX and deserve careful study.

Mini_send (line 7045) has three parameters: the caller, the process to be sent to, and a pointer to the buffer where the message is. It performs a number of tests. First, it makes sure that user processes try to send messages only to FS or MM. In line 7060 the parameter *caller_ptr* is tested with the macro *isuserp* to determine if the caller is a user process, and the parameter *dest* is tested with a similar function, *issysentn*, to determine if it is FS or MM. If the combination is not permitted *mini_send* terminates with an error.

Next a check is made to be sure the destination of the message is an active process, not an empty slot in the process table (line 7062). On lines 7068 to 7073 *mini_send* checks to see if the message falls entirely within the user's data segment, code segment, or the gap between them. If not, an error code is returned.

The next test is to check for a possible deadlock. On line 7079 is a test to make sure the destination of the message is not trying to send a message back to the caller.

The key test in *mini_send* is on lines 7088 to 7090. Here a check is made to see if the destination is blocked on a RECEIVE, as shown by the *RECEIVING* bit in the *p_flags* field of its process table entry. If it is waiting, then the next question is: "Who is it waiting for?" If it is waiting for the sender, or for ANY, *CopyMess* is executed to copy the message and the receiver is unblocked by resetting its *RECEIVING* bit. *CopyMess* is defined as a macro on line 6932. It calls the assembly language routine *cp_mess* in *klib386.s*.

If, on the other hand, the receiver is not blocked, or is blocked but waiting for a message from someone else, the code on lines 7098 to 7111 is executed to block and queue the sender. All processes wanting to send to a given destination are strung together on a linked list, with the destination's *p_callerq* field pointing to the process table entry of the process at the head of the queue. The example of Fig. 2-35(a) shows what happens when process 3 is unable to send to process 0. If process 4 is subsequently also unable to send to process 0, we get the situation of Fig. 2-35(b).

Mini_rec (line 6119) is called by *sys_call* when its function parameter is *RECEIVE* or *BOTH*. The loop on lines 7137 to 7151 searches through all the processes queued waiting

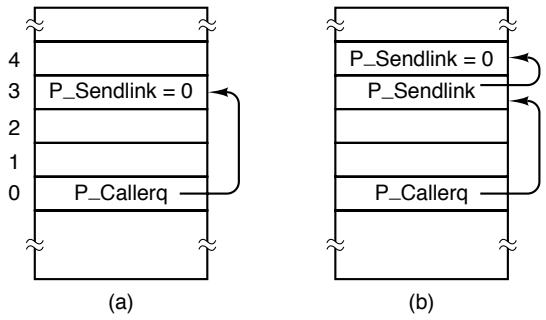


Figure 2-35. Queueing of processes trying to send to process 0.

to send to the receiver to see if any are acceptable. If one is found, the message is copied from sender to receiver; then the sender is unblocked, made ready to run, and removed from the queue of processes trying to send to the receiver.

If no suitable sender is found, a check is made to see if the receiving process' *p_int_blocked* flag indicates that an interrupt for this destination was previously blocked (line 7154). If so a message is constructed at this point—since messages from *HARDWARE* have no content other than *HARDWARE* in the source field and *HARD_INT* in the type field there is no need to call *CopyMess* in this case.

If a blocked interrupt is not found the process' source and buffer address are saved in its process table entry, and it is marked as blocked with its *RECEIVING* bit set. The call to *unready* on line 7165 removes the receiver from the scheduler's queue of runnable processes. The call is conditional to avoid blocking the process just yet if there is another bit set in its *p_flags*; a signal may be pending. and the process should have another chance to run soon to deal with the signal.

The penultimate statement in *mini_rec* (lines 7171 and 7172) has to do with how the kernel-generated signals *SIGINT*, *SIGQUIT*, and *SIGALRM* are handled. When one of these occurs, a message is sent to the memory manager, if it is waiting for a message from ANY. If not, the signal is remembered in the kernel until the memory manager finally tries to receive from ANY. That is tested here, and, if necessary, *inform* is called to informed it of the pending signals.

2.6.9 Scheduling in MINIX

MINIX uses a multilevel scheduling algorithm that closely follows the structure shown in Fig. 2-26. In that figure we see I/O tasks in layer 2, server processes in layer 3, and user processes in layer 4. The scheduler maintains three queues of runnable processes, one for each layer, as shown in Fig. 2-36. The array *rdy_head* has one entry for each queue, with that entry pointing to the process at the head of the queue. Similarly, *rdy_tail* is an array whose entries point to the last process on each queue. Both of these arrays are defined with the *EXTERN* macro in *proc.h* (lines 5192 and 5193).

Whenever a blocked process is awakened, it is appended to the end of its queue. The existence of the array *rdy_tail* makes adding a process to the end of a queue efficient. Whenever a running process becomes blocked, or a runnable process is killed by a signal, that process is removed from the scheduler's queues. Only runnable processes are queued.

Given the queue structures just described, the scheduling algorithm is simple: find the highest priority queue that is not empty and pick the process at the head of that queue. If all the queues are empty, the idle routine is run. In Fig. 2-36 *TASK_Q* has the highest

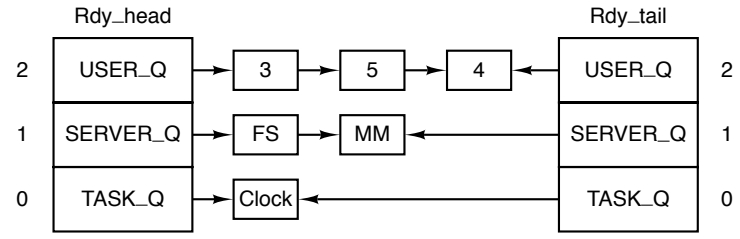


Figure 2-36. The scheduler maintains three queues, one per priority level.

priority. The scheduling code is in *proc.c*. The queue is chosen in *pick_proc* (line 7179). This function's major job is to set *proc_ptr*. Any change to the queues that might affect the choice of which process to run next requires *pick_proc* to be called again. Whenever the current process blocks, *pick_proc* is called to reschedule the CPU.

Pick_proc is simple. There is a test for each queue. *TASK_Q* is tested first. and if a process on this queue is ready, *pick_proc* sets *proc_ptr* and returns immediately. Next, *SERVER_Q* is tested, and, again, if a process is ready *pick_proc* sets *proc_ptr* and returns. If there is a ready process on the *USER_Q* queue, *bill_ptr* is changed to charge the user process for the CPU time it is about to be given (line 7198). This assures that the last user process to run is charged for work done on its behalf by the system. If none of the queues have a ready task line 7204 transfers billing to the *IDLE* process and schedules it. The process chosen to run is not removed from its queue merely because it has been selected.

The procedures *ready* (line 7210) and *unready* (line 7258) are called to enter a runnable process on its queue and remove a no-longer runnable process from its queue, respectively. *Ready* is called from both *mini_send* and *mini_rec*, as we have seen. It could also have been called from *interrupt*, but in the interest of speeding up interrupt processing its functional equivalent was written into *interrupt* as inline code. *Ready* manipulates one of the three process queues. It straightforwardly adds the process to the tail of the appropriate queue.

Unready also manipulates the queues. Normally, the process it removes is at the head of its queue, since a process must be running in order to block. In such a case *unready* calls *pick_proc* before returning, as, for example, in line 7293. A user process that is not running can also become *unready* if it is sent a signal, and if the process is not found at the head of one of the queues, a search is made through the *USER_Q* for it, and it is removed if found.

Although most scheduling decisions are made when a process blocks or unblocks, scheduling must also be done when the clock task notices that the current user process has exceeded its quantum. In this case the clock task calls *sched* (line 7311) to move the process at the head of *USER_Q* to the end of that queue. This algorithm results in running user processes in a straight round-robin fashion. The file system, memory manager, and I/O tasks are never put on the end of their queues because they have been running too long. They are trusted to work properly, and to block after having finished their work.

There are a few more routines in *proc.c* that support process scheduling. Five of these, *lock_mini_send*, *lock_pick_proc*, *lock_ready*, *lock_unreaaly*, and *lock_sched*, set a lock, using the variable *switching* before calling the corresponding function and then release the lock upon completion. The last function in this file, *unhold* (line 7400), was mentioned in our discussion of *_restart* in *mpx386.s*. It loops through the queue of held-up interrupts, calling *interrupt* for each one, in order to get all pending interrupts converted to messages before another process is allowed to run.

In summary, the scheduling algorithm maintains three priority queues, one for the I/O tasks, one for the server processes, and one for the user processes. The first process on the highest priority queue is always run next. Tasks and servers are always allowed to run until they block, but the clock task monitors the time used by user processes. If a user process uses up its quantum, it is put at the end of its queue, thus achieving a simple round-robin scheduling among the competing user processes.

2.6.10 Hardware-Dependent Kernel Support

There are several C functions that are very dependent upon the hardware. To facilitate porting MINIX to other systems these functions are segregated in the files to be discussed in this section, *exceptions.c*, *i8259.c*, and *protect.c*, rather than being included in the same files with the higher-level code they support.

Exception.c contains the exception handler, *exception* (line 7512), which is called (as *_exception*) by the assembly language part of the exception handling code in *mpx386.s*. Exceptions originating from user processes are converted to signals. Users are expected to make mistakes in their own programs, but an exception originating in the operating system indicates something is seriously wrong and causes a panic. The array *ex_data* (lines 7522 to 7540) determines the error message to be printed in case of panic, or the signal to be sent to a user process for each exception; Earlier Intel processors do not generate all the exceptions, and the third field in each entry indicates the minimum processor model that is capable of generating each one. This array provides an interesting summary of the evolution of the Intel family of processors upon which MINIX has been implemented. On line 7563 an alternate message is printed if a panic results from an interrupt that would not be expected from the processor in use.

The three functions in *i8259.c* are used during system initialization to initialize the Intel 8259 interrupt controller chips. *Intr_init* (line 7621) initializes the controllers. It writes data to several port locations. On a few lines a variable derived from the boot parameters is tested, for instance, the first port writes on line 7637, to accommodate different computer models. On line 7638, and again on line 7644, the parameter *mine* is tested, and a value appropriate either for MINIX or for the BIOS ROM is written to the port. When leaving MINIX *intr_init* can be called to restore the BIOS vectors, allowing a graceful exit back to the boot monitor. *Mine* selects the mode to use. Fully understanding what is going on here would require study of the documentation for the 8259 integrated circuit, and thus we will not dwell on the details. We will point out that the *out_byte* call on line 7642 makes the master controller unresponsive to any input except from the slave, and the similar operation on line 7648 inhibits the response of the slave to all of its inputs. Also. the final line of the function preloads the address of *spurious_irq*, the next function in the file (line 7657), into each slot in *irq_table*. This ensures that any interrupt generated before the real handlers are installed will do no harm.

The last function in *i8259.c* is *put_irq_handler* (line 7673). At initialization each task that must respond to an interrupt calls this to put its own handler address into the interrupt table, overwriting the address of *spurious_irq*.

Protect.c contains routines related to protected mode operation of Intel processors. The **Global Descriptor Table** (GDT), **Local Descriptor Tables** (LDTs), and the **Interrupt Descriptor Table**, all located in memory, provide protected access to system resources. The GDT and IDT are pointed to by special registers within the CPU, and GDT entries point to LDTs. The GDT is available to all processes and holds segment descriptors for memory regions used by the operating system. There is normally one LDT for

each process, holding segment descriptors for the memory regions used by the process. Descriptors are 8-byte structures with a number of components, but the most important parts of a segment descriptor are the fields that describe the base address and the limit of a memory region. The IDT is also composed of 8-byte descriptors, with the most important part being the address of the code to be executed when the corresponding interrupt is activated.

Prot_init (line 7767) is called by *start.c* to set up the GDT on lines 7828 to 7845. The IBM PC BIOS requires that it be ordered in a certain way, and all the indices into it are defined in *protect.h*. Space for an LDT for each process is allocated in the process table. Each contains two descriptors, for a code segment and a data segment—recall we are discussing here segments as defined by the hardware; these are not the same as the segments managed by the operating system, which considers the hardware-defined data segment to be further divided into data and stack segments. On lines 7851 to 7857 descriptors for each LDT are built in the GDT. The functions *init_dataseg* and *init_codeseg* actually build these descriptors. The entries in the LDTs themselves are initialized when a process' memory map is changed (i.e., when an EXEC system call is made).

Another processor data structure that needs initialization is the **Task State Segment** (TSS). The structure is defined at the start of this file (lines 7725 to 7753) and provides space for storage of processor registers and other information that must be saved when a task switch is made. MINIX uses only the fields that define where a new stack is to be built when an interrupt occurs. The call to *init_dataseg* on line 7867 ensures that it can be located using the GDT.

To understand how MINIX works at the lowest level, perhaps the most important thing is to understand how exceptions, hardware interrupts, or `int` *nnn* instructions lead to the execution of the various pieces of code that has been written to service them. This is accomplished by means of the interrupt gate descriptor table. The array *gate_table* (lines 7786 to 7818), is initialized by the compiler with the addresses of the routines that handle exceptions and hardware interrupts and then is used in the loop at lines 7873 to 7877 to initialize a large part of this table, using calls to the *int_gate* function. The remaining vectors, *SYS_VECTOR*, *SYS386_VECTOR*, and *LEVEL0_VECTOR*, require different privilege levels and are initialized following the loop.

There are good reasons for the way the data are structured in the descriptors, based on details of the hardware and the need to maintain compatibility between advanced processors and the 16-bit 286 processor. Fortunately, we can normally leave these details to Intel's processor designers. For the most part the C language allows us to avoid the details. However, in implementing a real operating system the details must be faced at some point. Figure 2-37 shows the internal structure of one kind of segment descriptor. Note that the base address, which C programs can refer to as a simple 32-bit unsigned integer, is split into three parts, two of which are separated by a number of 1-, 2-, and 4-bit quantities. The limit is a 20-bit quantity stored as separate 16-bit and 4-bit chunks. The limit is interpreted as either a number of bytes or a number of 4096-byte pages, based on the value of the G (granularity) bit. Other descriptors, such as those used to specify how interrupts are handled, have different, but equally complex structures. We discuss these structures in more detail in Chapter 4.

Most of the other functions defined in *protect.c* are devoted to conventing between variables used in C programs and the rather ugly forms these data take in the machine readable descriptors such as the one in Fig. 2-37. *Init_codeseg* (line 7889) and *init_dataseg* (line 7906) are similar in operation and are used to convert the parameters passed to them into segment descriptors. They each, in turn, call the next function, *sdesc* (line 7922), to

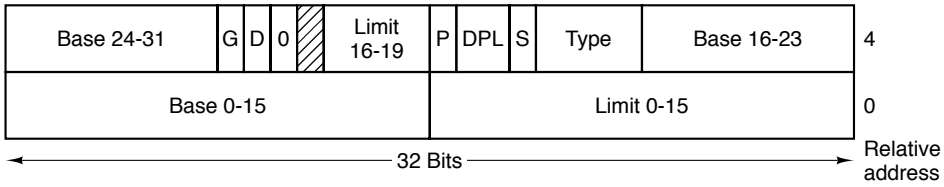


Figure 2-37. The format of an Intel segment descriptor.

complete the job. This is where the messy details of the structure shown in Fig. 2-37 are dealt with. *Init_codeseg* and *init_dataseg* are not used just at system initialization. In addition, they are also called by the system task whenever a new process is started up, in order to allocate the proper memory segments for the process to use. *Seg2phys* (line 7947), called only from *start.c*, performs an operation which is the inverse of that of *sdesc*, extracting the base address of a segment from a segment descriptor. *Int_gate* (line 7969) performs a similar function to *init_codeseg* and *init_dataseg* in building entries for the interrupt descriptor table.

The final function in *protect.c*, *enable_iop* (line 7988) performs a dirty trick. We have pointed out in several places that one function of an operating system is to protect system resources, and one way MINIX does so is by using privilege levels to make certain kinds of instructions off limits to user programs. However, MINIX is also intended to be run on small systems, which are likely to have only one user or perhaps just a few trusted users. On such a system a user could very well want to write an application program that accesses I/O ports, for instance, for use in scientific data acquisition. The file system has a little secret built into it—when the files */dev/mem* or */dev/kmem* are opened, the memory task calls *enable_iop*, which changes the privilege level for I/O operations, allowing the current process to execute instructions which read and write I/O ports. The description of the purpose of the function is more complicated than the function itself, which just sets two bits in the word in the stack frame entry of the calling process that will be loaded into the CPU status register when the process is next executed. There is no need for another function to undo this, as it will apply only to the calling process.

2.6.11 Utilities and the Kernel Library

Finally, the kernel has a library of support functions written in assembly language that are included by compiling *klib.s* and a few utility programs, written in C, in the file *misc.c*. Let us first look at the assembly language files. *Klib.s* (line 8000), is a short file similar to *mpx.s*, which selects the appropriate machine-specific version based upon the definition of *WORD_SIZE*. The code we will discuss is in *klib386.s* (line 8100). This contains about two dozen utility routines that are in assembly code, either for efficiency or because they cannot be written in C at all.

_Monitor (line 8166) makes it possible to return to the boot monitor. From the point of view of the boot monitor all of MINIX is just a subroutine, and when MINIX is started, a return address to the monitor is left on the monitor's stack. *_Monitor* just has to restore the various segment selectors and the stack pointer that was saved when MINIX was started, and then return as from any other subroutine.

The next function, *_check_mem* (line 8198), is used at startup time to determine the size of a block of memory. It performs a simple test on every sixteenth byte, using two patterns which test every bit with both "0" and "1" values.

Although *_phys_copy* (see below) could have been used for copying messages, *_cp_mess*

(line 8243), a faster specialized procedure, has been provided for that purpose. It is called by

```
cp_mess(source, src_clicks, src_offset, dest_cIicks, dest_offset);
```

where *source* is the sender's process number, which is copied into the *m_source* field of the receiver's buffer. Both the source and destination addresses are specified by giving a click number, typically the base of the segment containing the buffer. and an offset from that click. This form of specifying the source and destination is more efficient than the 32-bit addresses used by *_phys_copy*.

_Exit, *__exit*, and *___exit* (lines 8283 to 8285) are defined because some library routines that might be used in compiling MINIX make calls to the standard C function *exit*. An exit from the kernel is not a meaningful concept; there is nowhere to go. The solution here is to enable interrupts and enter an endless loop. Eventually, an I/O operation or the clock will cause an interrupt and normal system operation will resume. The entry point for *___main* (line 8289) is another attempt to deal with a compiler action which, while it might make sense while compiling a user program, does not have any purpose in the kernel. It points to an assembly language ret (return from subroutine) instruction.

_In_byte (line 8300), *_in_word* (line 8314), *_out_byte* (line 8328), and *_out_word* (line 8342) provide access to I/O ports, which on Intel hardware occupy a separate address space from memory and use different instructions from memory reads and writes. *_Port_read* (line 8359), *_port_read_byte* (line 8386), *_port_write* (line 8412), and *_port_write_byte* (line 8439) handle transfers of blocks of data between I/O ports and memory; they are used primarily for transfers to and from the disk which must be done more rapidly than is possible with the other I/O calls. The byte versions read 8 bits rather than l6 bits in each operation to accommodate older 8-bit peripheral devices.

Occasionally, it is necessary for a task to disable all CPU interrupts temporarily. It does this by calling *_lack* (line 8462). When interrupts can be reenabled, the task can call *_unlock* (line 8474) to enable interrupts. A single machine instruction performs each one of these operations. In contrast, the code for *_enable_irq* (line 8488) and *_disable_irq* (line 8521) is more complicated. They work at the level of the interrupt controller chips to enable and disable individual hardware interrupts.

Phys_copy (line 8564) is called in C by

```
phys_copy(source_address, destination_address, bytes);
```

and copies a block of data from anywhere in physical memory to anywhere else. Both addresses are absolute, that is, address 0 really means the first byte in the entire address space, and all three parameters are unsigned longs.

The next two short functions are very specific to Intel processors. *_Mem_rdw* (line 8608) returns a 16-bit word from anywhere in memory. The result is zero-extended into the 32-bit eax register. The *_reset* function (line 8623) resets the processor. It does this by loading the processor's interrupt descriptor table register with a null pointer and then executing a software interrupt. This has the same effect as a hardware reset.

The next two routines support the video display and are used by the console task. *_Mem_vid_copy* (line 8643) copies a string of words containing alternate character and attribute bytes from the kernel's memory region to the video display memory. *_Vid_vid_copy* (line 8696) copies a block within the video memory itself. This is somewhat more complicated, since the destination block may overlap the source block, and the direction of the move is important.

The last function in this file is *_level0* (line 8773) It allows tasks to have the most privileged permission level, level zero, when necessary. It is used for such things as resetting the CPU or accessing the PC's ROM BIOS routines.

The C language utilities in *misc.c* are specialized. *Mem_init* (line 8820) is called only by *main*, when MINIX is first started. There can be two or three disjoint regions of memory on an IBM-PC compatible computer. The size of the lowest range, known to PC users as "ordinary" memory, and of the memory range that starts above the PC ROM area ("extended" memory) are reported by the BIOS to the boot monitor, which in turn passes the values as boot parameters, which are interpreted by *cstart* and written to *low_memsize* and *ext_memsize* at boot time. The third region is "shadow" memory, into which the BIOS ROM may be copied to provide an improvement in performance, since ROM memory is usually slower than writable memory. Since MINIX does not normally use the BIOS, *mem_init* attempts to locate this memory and add it to the pool of memory available for its use. It does this by calling *check_mem* to test the memory region where this memory may sometimes be found.

The next routine, *env_parse* (line 8865) is also used at startup time. The boot monitor can pass arbitrary strings like "DPETH0=300:l0" to MINIX in the boot parameters. *Env_parse* tries to find a string whose first field matches its first argument, env, and then to extract the requested field. The comments in the code explain the use of the function. It is provided primarily to aid the user who wants to add new drivers which may need to be provided with parameters. The example "DPETH0" is used to pass configuration information to an Ethernet adapter when networking support is compiled into MINIX.

The last two routines we will discuss in this chapter are *bad_assertion* (line 8935) and *bad_compare* (line 8947). They are compiled only if the macro *DEBUG* is defined as *TRUE*. They support the macros in *assert.h*. Although they are not referenced in any of the code discussed in this text, they may be useful for debugging to the reader who wants to create a modified version of MINIX.

2.7 Summary

To hide the effects of interrupts, operating systems provide a conceptual model consisting of sequential processes running in parallel. Processes can communicate with each other using interprocess communication primitives, such as semaphores, monitors, or messages. These primitives are used to ensure that no two processes are ever in their critical sections at the same time. A process can be running, runnable, or blocked and can change state when it or another process executes one of the interprocess communication primitives.

Interprocess communication primitives can be used to solve such problems as the producer-consumer, dining philosophers, reader-writer, and sleeping barber. Even with these primitives, care has to be taken to avoid errors and deadlocks. Many scheduling algorithms are known, including round-robin, priority scheduling, multilevel queues, and policy-driven schedulers.

MINIX supports the process concept and provides messages for interprocess communication. Messages are not buffered, so a SEND succeeds only when the receiver is waiting for it. Similarly. a RECEIVE succeeds only when a message is already available. If either operation does not succeed, the caller is blocked.

When an interrupt occurs, the lowest level of the kernel creates and sends a message to the task associated with the interrupting device. For example, the disk task calls *receive*

and is blocked after writing a command to the disk controller hardware requesting it to read a block of data. The controller hardware causes an interrupt to occur when the data are ready. The low-level software then builds a message for the disk task and marks it as runnable. When the scheduler chooses the disk task to run, it gets and processes the message. It is also possible for the interrupt handler to do some work directly, such as a clock interrupt updating the time.

Task switching may follow an interrupt. When a process is interrupted, a stack is created within the process table entry of the process, and all the information needed to restart it is put on the new stack. Any process can be restarted by setting the stack pointer to point to its process table entry and initiating a sequence of instructions to restore the CPU registers, culminating with an `iretd` instruction. The scheduler decides which process table entry to put into the stack pointer.

Interrupts also occur when the kernel itself is running. The CPU detects this, and the kernel stack, rather a stack within the process table, is used. Thus nested interrupts can occur, and when a later interrupt service routine terminates, the one below it can complete. When all interrupts have been serviced, a process is restarted.

The MINIX scheduling algorithm uses three priority queues, the highest one for tasks, the next one for the file system, memory manager, and other servers, if any, and the lowest one for user processes. User processes are run round robin for one quantum at a time. All the others are run until they block or are preempted.

Chapter 3

INPUT/OUTPUT $\surd$

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 device 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.

An outline of the chapter is as follows. First, we will look briefly 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-designed task to perform. We will look at these layers to see what they do and how they fit together.

After that comes a section on deadlocks. We will define deadlocks precisely, show how they are caused, give two models for analyzing them, and discuss some algorithms fro preventing their occurrence.

Then we will take a bird's-eye view of I/O in MINIX. Following that introduction, we will look at four I/O devices in detail—the RAM disk, the hard disk, the clock, and the terminal. For each device we will look at its hardware, software, and implementation in MINIX. Finally, the chapter closes with a short discussion of a little piece of MINIX that is located in the same layer as the I/O tasks but is itself not an I/O task. It provides some services to the memory manager and file system, such as fetching blocks of data from a user process.

3.1 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.

3.1.1 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 an 8mm or DAT tape drive used for making disk backups. Its 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 not normally 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 only with abstract block devices and leaves the device-dependent part to lower-level software called **device drivers**.

3.1.2 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 a slot on the computer's **parentboard** (previously incorrectly called a motherboard). The mechanical component is the device itself.

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 standard ANSI, IEEE, or ISO, 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 (Integrated Drive Electronics) and SCSI (Small Computer System Interface) interfaces.

We mention this distinction between controller and device because the operating system nearly always deals with the controller, not the device. Most small computers use the single bus model of 3-1 for communication between the CPU and the controllers.

Large mainframes often use a different model, with multiple buses and specialized I/O computers called **I/O channels** taking some of the load off the main CPU.

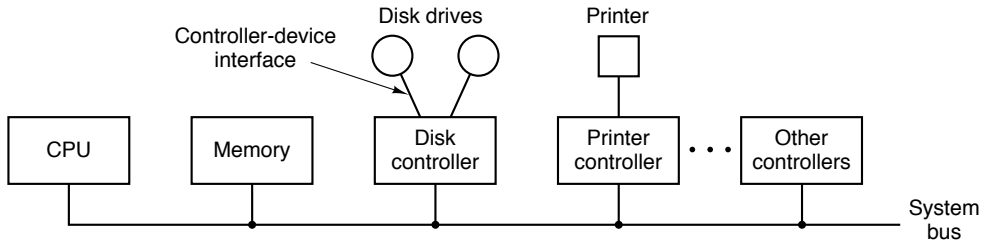


Figure 3-1. A model for connecting the CPU, memory, controllers, and I/O devices.

The interface between the controller and the device is often low-level. A disk, for example, might be formatted with 16 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.

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 CRT terminal 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 per line and number of lines per screen, and lets the controller take care of actually driving the beam.

Each controller has a few registers that are used for communicating with the CPU. On some computers, these registers are part of the regular memory address space. This scheme is called **memory-mapped I/O**. The 680x0, for example, uses this method. Other computers use a special address space for I/O, with each controller allocated a certain portion of it. The assignment of I/O addresses to devices is made by bus decoding logic associated with the controller. Some manufacturers of so-called IBM PC compatibles use different I/O addresses from those IBM uses. In addition to I/O ports, many controllers use interrupts to tell the CPU when they are ready to have their registers read or written. An interrupt is, in the first place, an electrical event. A hardware Interrupt ReQuest line (IRQ) is a physical input to the interrupt controller chip. The number of such inputs is limited; Pentium-class PCs have only 15 available for I/O devices. Some controllers are hard-wired onto the system parentboard, as is, for instance, the keyboard controller of an IBM PC. In the case of a controller that plugs into the backplane, switches or wire jumpers on the device controller sometimes can be used to select which IRQ the device will use, in order to avoid conflicts (although with some boards, such as Plug 'n Play, the IRQs can be set in software). The interrupt controller chip maps each IRQ input to an

interrupt vector, which locates the corresponding interrupt service software. Figure 3-2 shows the I/O addresses, hardware interrupts, and interrupt vectors allocated to some of the controllers on an IBM PC, as an example. MINIX uses the same hardware interrupts, but the MINIX interrupt vectors are different from those shown here for MS-DOS.

I/O controller	I/O address	Hardware IRQ	Interrupt vector
Clock	040 – 043	0	8
Keyboard	060 – 063	1	9
Hard disk	1F0 – 1F7	14	118
Secondary RS232	2F8 – 2FF	3	11
Printer	378 – 37F	7	15
Floppy disk	3F0 – 3F7	6	14
Primary RS232	3F8 – 3FF	4	12

Figure 3-2. Some examples of controllers, their I/O addresses, their hardware interrupt lines, and their interrupt vectors on a typical PC running MS-DOS.

The operating system performs I/O by writing commands into the controller's registers. The IBM PC floppy disk controller, for example, accepts 15 different commands, such as READ, WRITE, SEEK, FORMAT, and RECALIBRATE. Many of the commands have parameters, which are also loaded into the controller's registers. When a command has been accepted, the CPU can leave the controller alone and go off to do other work. When the command has been completed, the controller causes an interrupt in order to allow the operating system to gain control of the CPU and test the results of the operation. The CPU gets the results and device status by reading one or more bytes of information from the controller's registers.

3.1.3 Direct Memory Access (DMA)

Many controllers, especially those for block devices, support **Direct Memory Access** or DMA. 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 memory.

Naturally, a programmed CPU loop to read the bytes one at a time from the controller wastes CPU time. DMA was invented to free the CPU from this low-level work. When it is used, the CPU gives the controller two items of information, in addition to the disk address of the block: the memory address where the block is to go, and the number of bytes to transfer, as shown in Fig. 3-3.

After the controller has read the entire block from the device into its buffer and verified the checksum, it copies the first byte or word into the main memory at the address specified by the DMA memory address. Then it increments the DMA address and decrements the DMA count by the number of bytes just transferred. This process is repeated until

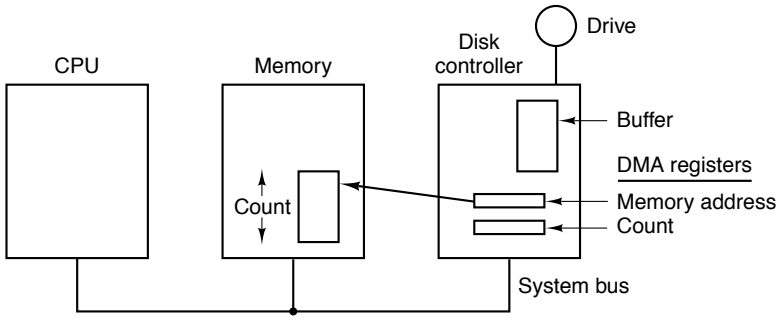


Figure 3-3. A DMA transfer is done entirely by the controller.

the DMA count becomes zero, at which time the controller causes an interrupt. When the operating system starts up, it does not have to copy the block to memory; it is already there.

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? The 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, 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.)

The two-step buffering process described above has important implications for I/O performance. While the data are being transferred from the controller to the memory, either by the CPU or by the controller, the next sector will be passing under the disk head and the bits arriving in the controller. Simple controllers just cannot cope with doing input and output at the same time, so while a memory transfer is taking place, the sector passing under the disk head is lost.

As a result, the controller will be able to read only every other block. Reading a complete track will then require two full rotations, one for the even blocks and one for the odd blocks. If the time to transfer a block from the controller to memory over the bus is longer than the time to read a block from the disk, it may be necessary to read one block and then skip two (or more) blocks.

Skipping blocks to give the controller time to transfer data to memory is called **interleaving**. When the disk is formatted, the blocks are numbered to take account of the interleave factor. In Fig. 3-4(a) we see a disk with 8 blocks per track and no interleaving. In Fig. 3-4(b) we see the same disk with single interleaving. In Fig. 3-4(c) double interleaving is shown.

The idea of numbering the blocks this way is to allow the operating system to read consecutively numbered blocks and still achieve the maximum speed of which the hardware is capable. If the blocks were numbered as in Fig. 3-4(a) but the controller could read only alternate blocks, an operating system that allocated an 8-block file in consecu-

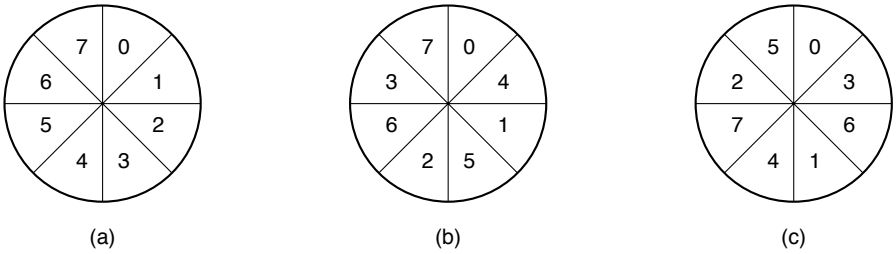


Figure 3-4. (a) No interleaving. (b) Single interleaving. (c) Double interleaving.

tive disk blocks would require eight disk rotations to read blocks 0 through 7 in order. (Of course, if the operating system knew about the problem and allocated its blocks differently, it could solve the problem in software, but it is better to have the controller worry about the interleaving.)

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 some money.

3.2 Principles of I/O Software

Let us turn away from the hardware and now look at how the I/O software is structured. The general goals of the I/O software are easy to state. The basic idea is to organize the software as a series of layers, with the lower ones concerned with hiding the peculiarities of the hardware from the upper ones, and the upper ones concerned with presenting a nice, clean, regular interface to the users. In the following sections we will look at these goals and how they are achieved.

3.2.1 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 read files on a floppy disk, on a hard disk. or on a CD ROM. without having to modify the programs for each different device type. One should be able to type a command such as

```
sort <input >output
```

and have it work with input coming from a floppy disk, a hard disk, or the keyboard, and the output going to the floppy disk, the hard disk, or even 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 device drivers to actually write the data to the output device.

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 together 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 flles 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 command 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.

The final concept that we will deal with 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. Again, the operating system must be able to handle both shared and dedicated devices in a way that avoids problems.

These goals can be achieved in a comprehensible and efficient way by structuring the I/O software in four layers:

1. Interrupt handlers (bottom).
2. Device drivers.
3. Device-independent operating system software.
4. User-level software (top).

These four layers are (not accidently) the same four layers that we saw in Fig. 2-26. In the following sections we will look at each one in turn, starting at the bottom. The emphasis in this chapter is on the device drivers (layer 2), but we will summarize the rest of the I/0 software to show how the various pieces of the l/O system fit together.

3.2.2 Interrupt Handlers

Interrupts are an unpleasant fact of life. They should be hidden away, deep in the bowels of the operating system, so that as little of the system as possible knows about them. The best way to hide them is to have every process starting an I/O operation block until the I/O has completed and the interrupt occurs. The process can block itself by doing a DOWN on a semaphore, a WAIT on a condition variable, or a RECEIVE on a message, for example.

When the interrupt happens, the interrupt procedure does whatever it has to in order to unblock the process that started it. In some systems it will do an 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 process. In all cases the net effect of the interrupt will be that a process that was previously blocked will now be able to run.

3.2.3 Device Drivers

All the device-dependent code goes in the device drivers. Each device driver handles one device type, or at most, one class of closely related devices. For example, it would probably be a good idea to have a single terminal driver, even if the system supported several different brands of terminals, all slightly different. On the other hand, a dumb, mechanical hardcopy terminal and an intelligent bitmap graphics terminal with a mouse are so different that different drivers should be used.

Earlier in this chapter we looked at what device controllers do. We saw that each controller has one or more device registers used to give it commands. The device drivers issue these commands and check that they are carried out properly. Thus, the disk driver is the only part of the operating system that knows how many registers that disk controller has and what they are used for. It alone knows about sectors, tracks, cylinders, heads, arm motion, interleave factors, motor drives, head settling times, and all the other mechanics of making the disk work properly.

In general terms, the job of a device driver is to accept abstract requests from the device-independent software above it and see to it that the request is executed. A typical request is to read block *n*. If the driver is idle at the time a request comes in, it starts carrying out the request immediately. If, however, it is already busy with a request, it will normally enter the new request into a queue of pending requests to be dealt with as soon as possible.

The first step in actually carrying out an I/O request, say, for a disk, is to translate it from abstract to concrete terms. For a disk driver, this means figuring out where on the disk the requested block actually is, checking to see if the drive's motor is running. determining if the arm is positioned on the proper cylinder, and so on. In short, it must decide which controller operations are required and in what sequence.

Once it has determined which commands to issue to the controller, it starts issuing them by writing into the controller's device registers. Some controllers can handle only one command at a time. Other controllers are willing to accept a linked list of commands, which they then carry out by themselves without further help from the operating system.

After the command or 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 on some terminals requires just writing a few bytes into the controller's registers. No mechanical motion is needed, so the entire operation can be completed in a few microseconds.

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, it 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.

3.2.4 Device-Independent I/O Software

device-independent software. In MINIX, most of the device-independent software is part of the file system, in layer 3 (Fig. 2-26). Although we will study the tile system in Chap. 5, we will take a quick look at the device-independent software here, to provide some perspective on I/O and show better where the drivers fit in.

Uniform interfacing for device drivers
Device naming
Device protection
Providing a device-independent block size
Buffering
Storage allocation on block devices
Allocating and releasing dedicated devices
Error reporting

Figure 3-5. 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.

A major issue in an operating system is how objects such as files and I/O devices are named. The device-independent software takes care of mapping symbolic device names onto the proper driver. In UNIX a device name, such as */dev/tty00*, 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 contains the **minor device number**, which is passed as a parameter to the driver to specify the unit to be read or written.

Closely related to naming is protection. How does the system prevent users from accessing devices that they are not entitled to access? In most personal computer systems, there is no protection at all. Any process can do anything it wants to. In most mainframe systems, access to I/O devices by user processes is completely forbidden. In UNIX, a more flexible scheme is used. The special files corresponding to I/O devices are protected by the usual rwx bits. The system administrator can then set the proper permissions for each device.

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 must also be hidden.

Buffering is also an issue, both for block and character devices. For block devices, the hardware generally insists upon reading and writing entire blocks at once, but user processes are free to read and write in arbitrary units. If a user process writes half a block, the operating system will normally keep the data around internally until the rest of the data are written, at which time the block can go out to the disk. For character devices,

users can write data to the system faster than it can be output, necessitating buffering. Keyboard input that arrives before it is needed also requires buffering.

When a file is created and filled with data, new disk blocks have to be allocated to the file. To perform this allocation, the operating system needs a list or bit map of free blocks per disk, but the algorithm for locating a free block is device independent and can be done above the level of the driver.

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 tiles for devices directly. If the device is unavailable, the OPEN will fail. Closing such a dedicated device would then release it.

Error handling, by and large, is done by the drivers. Most errors are highly device dependent, so only the driver knows what to do (e.g., retry, ignore it, panic). A typical error is caused by a disk block that has been damaged and cannot be read any more. After the driver has tried to read the block a certain number of times, it gives up and informs the device-independent software. How the error is treated from here on is device independent. If the terror occurred while reading a user file, it may be sufficient to report the error back to the caller. However, if it occurred while reading a critical system data structure, such as the block containing the bit map showing which blocks are free, the operating system may have no choice but to print an error message and terminate.

3.2.5 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. 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 Internet electronic mail system. This network consists of millions of machines around the world communicating using many computer networks. To send mail to someone, you call a program such as *send*, which accepts the letter to be sent and then deposits it in a spooling directory for transmission later. The entire mail system runs outside the operating system.

Figure 3-6 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.

The arrows in Fig. 3-6 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 in the block cache, for example. If the needed block is not there, it calls the device driver to issue the request to the hardware. 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.

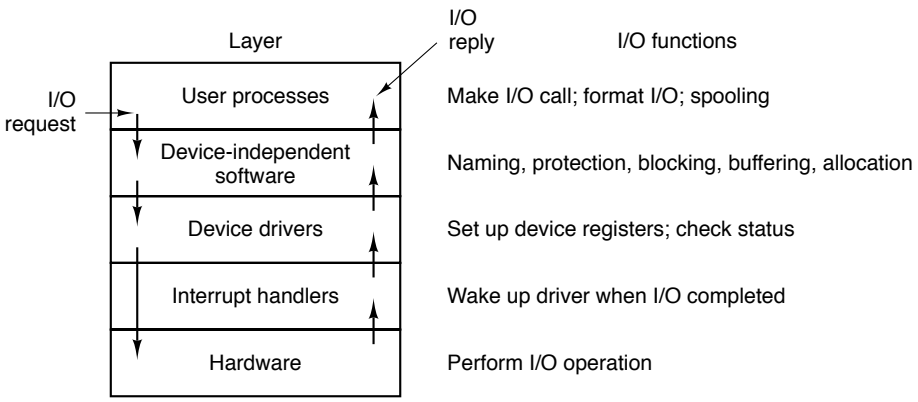


Figure 3-6. Layers of the I/O system and the main functions of each layer.

3.3 Deadlocks

Computer systems are full of resources that can only be used by one process at a time. Common examples include flatbed plotters, CD-ROM readers, CD-ROM recorders, 8mm DAT tape drive backup systems, imagesetters, and slots in the system's process table. Having two processes simultaneously writing to the printer leads to gibberish. Having two processes using the same slot in the process table will probably lead to a system crash. Consequently, all operating systems have the ability to (temporarily) grant a process exclusive access to certain resources.

For many applications, a process needs exclusive access to not one resource, but several. Consider, for example, a marketing company that specializes in making large, detailed demographic maps of the United States on a 1-meter wide flatbed plotter. The demographic information comes from CD-ROMs containing census and other data. Suppose that process *A* asks for the CD-ROM drive and gets it. A moment later, process *B* asks for the flatbed plotter and gets it, too. Now process *A* asks for the plotter and blocks waiting for it. Finally, process *B* asks for the CD-ROM drive and also blocks. At this point both processes are blocked and will remain so forever. This situation is called a **deadlock**. Deadlocks are not a good thing to have in your system.

Deadlocks can occur in many situations besides requesting dedicated I/O devices. In a data base system, for example, a program may have to lock several records it is using, to avoid race conditions. If process *A* locks record *R1* and process *B* locks record *R2*, and then each process tries to lock the other one's record, we also have a deadlock. Thus deadlocks can occur on hardware resources or on software resources.

In this section we will examine deadlocks more closely to see how they arise and how they cam he prevented or avoided. As examples, we will talk about acquiring physical devices such as tape drives, CD-ROM drives, and plotters, because these are easy to visualize, but the principles and algorithms hold equally well for other kinds of deadlocks.

3.3.1 Resources

Deadlocks can occur when processes have been granted exclusive access to devices, files, and so forth. To make the discussion of deadlocks as general as possible, we will refer to the objects granted as **resources**. A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a data base). A computer will normally have many different resources that can be acquired. For some resources, several identical instances may be available, such as three tape drives. When several copies of a resource are available, any one of them can be used to satisfy any request for the resource. In short, a resource is anything that can only be used by a single process at any instant.

Resources come in two types: preemptable and nonpreemptable. A **preemptable resource** is one that can be taken away from the process owning it with no ill effects. Memory is an example of a preemptable resource. Consider, for example, a system with 512K of user memory, one printer, and two 512K processes that each want to print something. Process *A* requests and gets the printer, then starts to compute the values to print. Before it has finished with the computation, it exceeds its time quantum and is swapped out.

Process *B* now runs and tries, unsuccessfully, to acquire the printer. Potentially, we now have a deadlock situation, because *A* has the printer and *B* has the memory, and neither can proceed without the resource held by the other. Fortunately, it is possible to preempt (take away) the memory from *B* by swapping it out and swapping *A* in. Now *A* can run, do its printing, and then release the printer. No deadlock occurs.

A **nonpreemptable resource**, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail. If a process has begun to print output, taking the printer away from it and giving it to another process will result in garbled output. Printers are not preemptable.

In general, deadlocks involve nonpreemptable resources. Potential deadlocks that involve preemptable ones can usually be resolved by reallocating resources from one process to another. Thus our treatment will focus on nonpreemptable resources.

The sequence of events required to use a resource is:

1. Request the resource.

2. Use the resource.

3. Release the resource.

If the resource is not available when it is requested, the requesting process is forced to wait. In some operating systems, the process is automatically blocked when a resource request fails and awakened when it becomes available. In other systems, the request fails with an error code, and it is up to the calling process to wait a little while and try again.

3.3.2 Principles of Deadlock

Deadlock can be defined formally as follows:

> *A set of processes is deadlocked If each process in the set is waiting for an event that only another process in the set can cause.*

Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever.

In most cases, the event that each process is waiting for is the release of some resource currently possessed by another member of the set. In other words, each member of the set of deadlocked processes is waiting for a resource that is owned by a deadlocked process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The number of processes and the number and kind of resources possessed and requested are unimportant.

Conditions for Deadlock

Coffman et al. (l971) showed that four conditions must hold for there to be a deadlock:

1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.

2. Hold and wait condition. Processes currently holding resources granted earlier can request new resources.

3. No preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.

4. Circular wait condition. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

All four of these conditions must be present for a deadlock to occur. If one or more of these conditions is absent, no deadlock is possible.

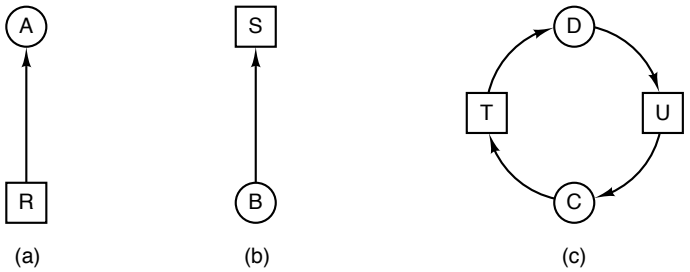


Figure 3-7. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

Deadlock Modeling

Holt (1972) showed how these four conditions can be modeled using directed graphs. The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares. An arc from a resource node (square) to a process node (circle) means that the resource previously has been requested by, granted to, and is currently held by that process. In Fig. 3-7(a), resource *R* is currently assigned to process *A*.

An arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig. 3-7(b) process *B* is waiting for resource *S*. In Fig. 3-7(c) we see a deadlock: process *C* is waiting for resource *T*, which is currently held by process *D*. Process *D* is not about to release resource *T* because it is waiting for resource *U*, held by *C*. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle. In this example, the cycle is *C-T-D-U-C*.

Now let us look at an example of how resource graphs can be used. Imagine that we have three processes, *A*, *B*, and *C*, and three resources, *R*, *S*, and *T*. The requests and releases of the three processes are given in Fig. 3-8(a)-(c). The operating system is free to run any unblocked process at any instant, so it could decide to run *A* until *A* finished all its work, then run *B* to completion, and finally run *C*.

This ordering does not lead to any deadlocks (because there is no competition for resources) but it also has no parallelism at all. In addition to requesting and releasing resources, processes compute and do I/O. When the processes are run sequentially, there is no possibility that while one process is waiting for I/O, another can use the CPU. Thus running the processes strictly sequentially may not be optimal. On the other hand, if none of the processes do any I/O at all, shortest job first is better than round robin, so under some circumstances running all processes sequentially may be the best way.

Let us now suppose that the processes do both I/O and computing, so that round robin is a reasonable scheduling algorithm. The resource requests might occur in the order of Fig. 3-8(d). If these six requests are carried out in that order, the six resulting resource graphs are shown in Fig. 3-8(e)-(j). After request 4 has been made, A blocks waiting for *S*, as shown in Fig. 3-8(h). In the next two steps *B* and *C* also block, ultimately leading to a cycle and the deadlock of Fig. 3-8(j).

However, as we have already mentioned, the operating system is not required to run the processes in any special order. In particular, if granting a particular request might lead to deadlock, the operating system can simply suspend the process without granting the request (i.e., just not schedule the process) until it is safe. In Fig. 3-8, if the operating system knew about the impending deadlock, it could suspend *B* instead of granting it *S*. By running only *A* and *C*, we would get the requests and releases of Fig. 3-8(k) instead

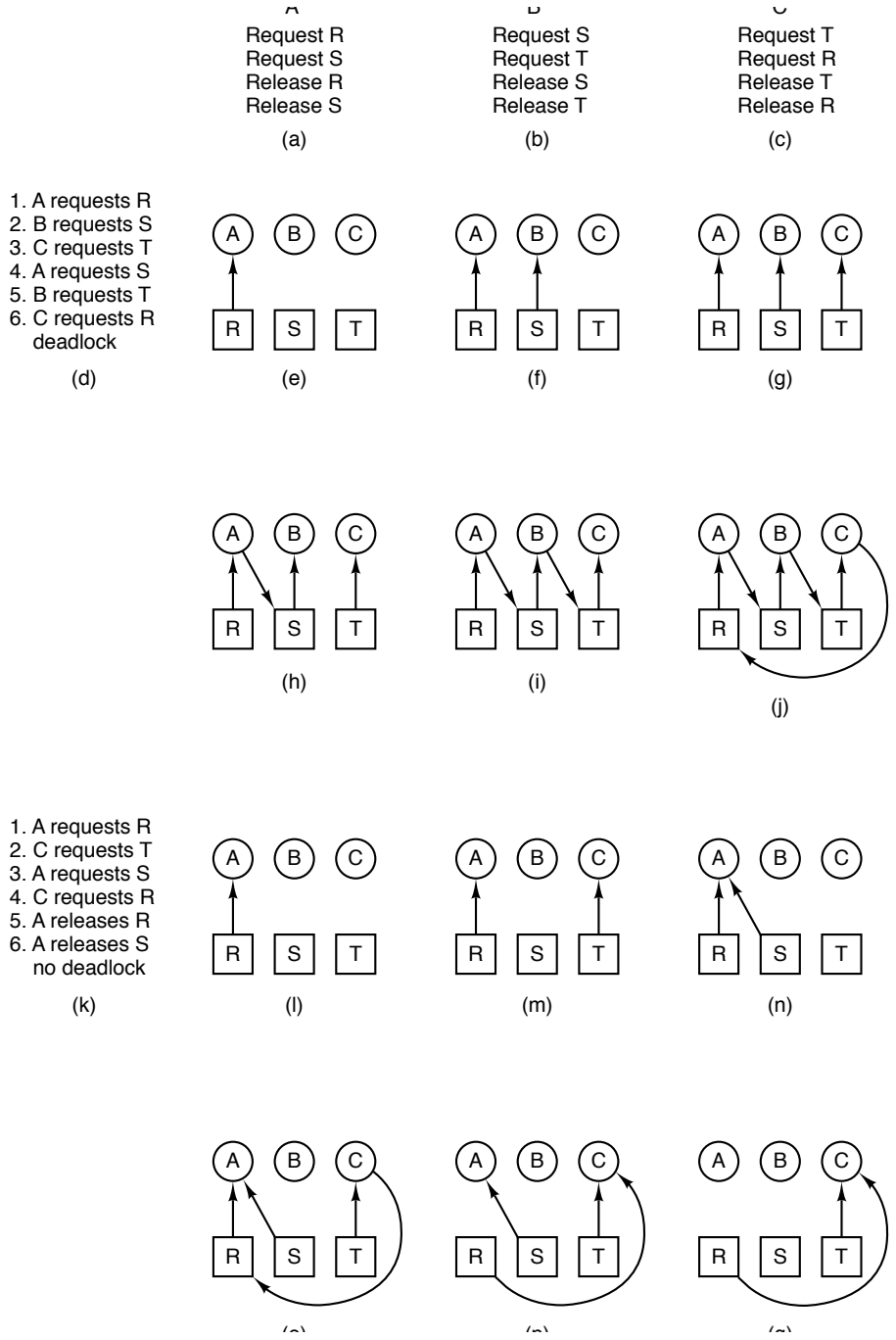


Figure 3-8. An example of how deadlock occurs and how it can be avoided.

of Fig. 3-8(d). This sequence leads to the resource graphs of Fig. 3-8(l)-(q), which do not lead to deadlock.

After step (q), process *B* can be granted *S* because *A* is finished and *C* has everything it needs. Even if *B* should eventually block when requesting *T*, no deadlock can occur. *B* will just wait until *C* is finished.

Later in this chapter we will study a detailed algorithm for making allocation decisions that do not lead to deadlock. The point to understand now is that resource graphs are a tool that let us see if a given request/release sequence leads to deadlock. We just carry out the requests and releases step by step, and after every step check the graph to see if it contains any cycles. If so, we have a deadlock; if not, there is no deadlock. Although our treatment of resource graphs has been for the case of a single resource of each type, resource graphs can also be generalized to handle multiple resources of the same type (Holt, 1972).

In general, four strategies are used for dealing with deadlocks.

1. Just ignore the problem altogether.
2. Detection and recovery.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four required conditions.

We will examine each of these methods in tum in the next four sections.

3.3.3 The Ostrich Algorithm

The simplest approach is the ostrich algorithm: stick your head in the sand and pretend there is no problem at all. Different people react to this strategy in different ways. Mathematicians find it totally unacceptable and say that deadlocks must be prevented at all costs. Engineers ask how often the problem is expected, how often the system crashes for other reasons, and how serious a deadlock is. If deadlocks occur on the average once every 50 years, but system crashes due to hardware failures, compiler errors, and operating system bugs occur once a month, most engineers would not be willing to pay a large penalty in performance or convenience to eliminate deadlocks.

To make this contrast more specific, UNIX (and MINIX) potentially suffer from deadlocks that are not even detected, let alone automatically broken. The total number of processes in the system is determined by the number of entries in the process table. Thus process table slots are finite resources. If a FORK fails because the table is full, a reasonable approach for the program doing the FORK is to wait a random time and try again.

Now suppose that a UNIX system has 100 process slots. Ten programs are running, each of which needs to create 12 (sub)processes. After each process has created 9 processes, the 10 original processes and the 90 new processes have exhausted the table. Each of the 10 original processes now sits in an endless loop forking and failing—a deadlock. The probability of this happening is minuscule, but it could happen. Should we abandon processes and the FORK call to eliminate the problem?

The maximum number of open files is similarly restricted by the size of the i-node table, so a similar problem occurs when it fills up. Swap space on the disk is another limited resource. In fact, almost every table in the operating system represents a finite

resource. Should we abolish all of these because it might happen that a collection of n processes might each claim $1/n$ of the total, and then each try to claim another one?

The UNIX approach is just to ignore the problem on the assumption that most users would prefer an occasional deadlock to a rule restricting all users to one process, one open file, and one of everything. If deadlocks could be eliminated for free, there would not be much discussion. The problem is that the price is high, mostly in terms of putting inconvenient restrictions on processes, as we will see shortly. Thus we are faced with an unpleasant trade-off between convenience and correctness, and a great deal of discussion about which is more important.

3.3.4 Detection and Recovery

A second technique is detection and recovery. When this technique is used, the system does not do anything except monitor the requests and releases of resources. Every time a resource is requested or released, the resource graph is updated, and a check is made to see if any cycles exist. If a cycle exists, one of the processes in the cycle is killed. If this does not break the deadlock, another process is killed, and so on until the cycle is broken.

A somewhat cruder method is to not even maintain the resource graph but instead periodically check to see if there are any processes that have been continuously blocked for more than say, 1 hour. Such processes are then killed.

Detection and recovery is the strategy often used on large mainframe computers, especially batch systems in which killing a process and then restarting it is usually acceptable. Care must be taken to restore any modified files to their original state, however, and undo any other side effects that may have occurred.

3.3.5 Deadlock Prevention

The third deadlock strategy is to impose suitable restrictions on processes so that deadlocks are structurally impossible. The four conditions stated by Coffman et al. (1971) provide a clue to some possible solutions. If we can ensure that at least one of these conditions is never satisfied, then deadlocks will be impossible (Havender, 1968).

First let us attack the mutual exclusion condition. If no resource were ever assigned exclusively to a single process, we would never have deadlocks. However, it is equally clear that allowing two processes to write on the printer at the same time will lead to chaos. By spooling printer output, several processes can generate output at the same time. In this model, the only process that actually requests the physical printer is the printer daemon. Since the daemon never requests any other resources, we can eliminate deadlock for the printer.

Unfortunately, not all devices can be spooled (the process table does not lend itself well to being spooled). Furthermore, competition for disk space for spooling can itself lead to deadlock. What would happen if two processes each filled up half of the available spooling space with output and neither was finished? If the daemon were programmed to begin printing even before all the output were spooled, the printer might lie idle if an output process decided to wait several hours after the first burst of output. For this reason, daemons are normally programmed to print only after the complete output file is available. Neither process will ever finish, so we have a deadlock on the disk.

The second of the conditions stated by Coffman et al. looks more promising. If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way to achieve this goal is to require all processes to request all their

resources before starting execution. If everything were available, the process would be allocated whatever it needed and could run to completion. If one or more resources were busy. nothing would be allocated and the process would just wait.

An immediate problem with this approach is that many processes do not know how many resources they will need until they have started running. Another problem is that resources will not be used optimally with this approach. Take, as an example, a process that reads data from an input tape, analyzes it for an hour, and then writes an output tape as well as plots the results. If all resources must be requested in advance, the process will tie up the output tape drive and the plotter for an hour.

A slightly different way to break the hold-and-wait condition is to require a process requesting a resource to first temporarily release all the resources it currently holds. Only if the request is successful can it get the original resources back.

Attacking the third condition (no preemption) is even less promising than attacking the second one. If a process has been assigned the printer and is in the middle of printing its output, forcibly taking away the printer because a needed plotter is not available will lead to a mess.

Only one condition is left. The circular wait can be eliminated in several ways. One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one. For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.

Another way to avoid the circular wait is to provide a global numbering of all the resources, as shown in Fig. 3-9(a). Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer.

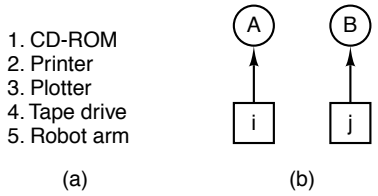


Figure 3-9. (a) Numerically ordered resources. (b) A resource graph.

With this rule, the resource allocation graph can never have cycles. Let us see why this is true for the case of two processes, in Fig. 3-9(b). We can get a deadlock only if A requests resource j and B requests resource i. Assuming i and j are distinct resources, they will have different numbers. If $i > j$, then A is not allowed to request j. If $i < j$, then B is not allowed to request i. Either way, deadlock is impossible.

With multiple processes the same logic holds. At every instant, one of the assigned resources will be highest. The process holding that resource will never ask for a resource already assigned. It will either finish, or at worst, request even higher numbered resources, all of which are available. Eventually, it will finish and free its resources. At this point, some other process will hold the highest resource and can also finish. In short, there exists a scenario in which all processes finish, so no deadlock is present.

A minor variation of this algorithm is to drop the requirement that resources be acquired in strictly increasing sequence and merely insist that no process request a resource lower than what it is already holding. If a process initially requests 9 and 10, and then

releases both of them, it is effectively starting all over, so there is no reason to prohibit it from now requesting resource 1.

Although numerically ordering the resources eliminates the problem of deadlocks, it may be impossible to find an ordering that satisfies everyone. When the resources include process table slots, disk spooler space, locked data base records. and other abstract resources, the number of potential resources and different uses may be so large that no ordering could possibly work.

The various approaches to deadlock prevention are summarized in Fig. 3-10.

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Figure 3-10. Summary of approaches to deadlock prevention.

3.3.6 Deadlock Avoidance

In Fig. 3-8 we saw that deadlock was avoided not by imposing arbitrary rules on processes but by carefully analyzing each resource request to see if it could be safely granted. The question arises: is there an algorithm that can always avoid deadlock by making the right choice all the time? The answer is a qualified yes—we can avoid deadlocks, but only if certain information is available in advance. In this section we examine ways to avoid deadlock by careful resource allocation.

The Banker's Algorithm for a Single Resource

A scheduling algorithm that can avoid deadlocks is due to Dijkstra (1965) and is known as the **banker's algorithm**. It is modeled on the way a small-town banker might deal with a group of customers to whom he has granted lines of credit. In Fig. 3-11(a) we see four customers, each of whom has been granted a certain number of credit units (e.g., 1 unit is 1K dollars). The banker knows that not all customers will need their maximum credit immediately, so he has only reserved 10 units rather than 22 to service them. (In this analogy, customers are processes, units are, say, tape drives, and the banker is the operating system.)

The customers go about their respective businesses, making loan requests from time to time. At a certain moment, the situation is as shown in Fig. 3-11(b). A list of customers showing the money already loaned (tape drives already assigned) and the maximum credit available (maximum number of tape drives needed at once later) is called the **state** of the system with respect to resource allocation.

A state is said to be a **safe** state if there exists a sequence of other states that leads to all the customers getting loans up to their credit limits (all the processes getting all their resources and terminating). The state of Fig. 3-11(b) is safe because with two units left, the banker can delay any requests except Marvin's, thus letting Marvin finish and release all four of his resources. With four units in hand, the banker can let either Suzanne or Barbara have the necessary units, etc.

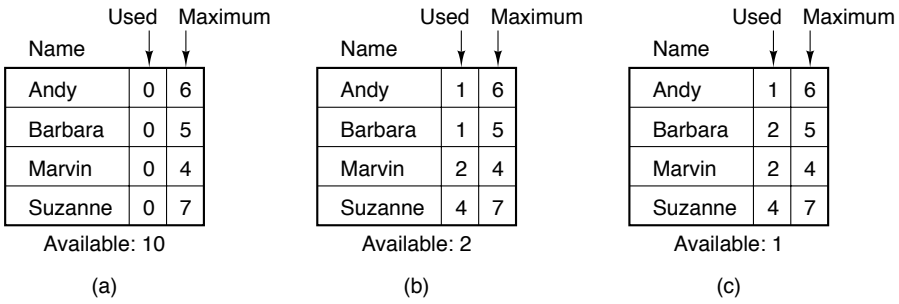


Figure 3-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

Consider what would happen if a request from Barbara for one more unit were granted in Fig. 3-11(b). We would have the situation of Fig. 3-11(c), which is unsafe. If all the customers suddenly asked for their maximum loans, the banker could not satisfy any of them, and we would have a deadlock. An unsafe state does not have to lead to deadlock, since a customer might not need the entire credit line available, but the banker cannot count on this behavior.

The banker's algorithm is thus to consider each request as it occurs and see if granting it leads to a safe state. If it does, the request is granted; otherwise, it is postponed until later. To see if a state is safe, the banker checks to see if he has enough resources to satisfy the customer closest to his or her maximum. If so, those loans are assumed to be repaid, and the customer now closest to his or her limit is checked, and so on. If all loans can eventually be repaid, the state is safe and the initial request can be granted.

Resource Trajectories

The above algorithm was described in terms of a single resource class (e.g., only tape drives or only printers, but not some of each). In Fig. 3-12 we see a model for dealing with two processes and two resources, for example, a printer and a plotter. The horizontal axis represents the number of instructions executed by process A. The vertical axis represents the number of instructions executed by process B. At I_1 A requests a printer; at I_2 it needs a plotter. The printer and plotter are released at I_3 and I_4, respectively. Process B needs the plotter from I_5 to I_7 and the printer from I_6 to I_8.

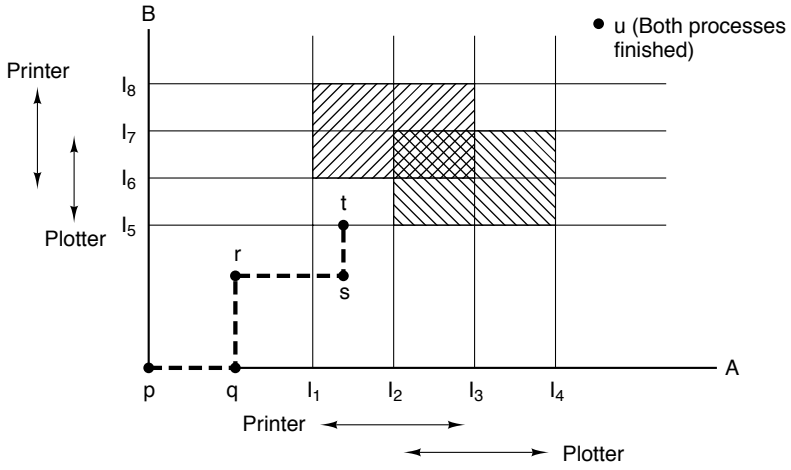


Figure 3-12. Two process resource trajectories.

Every point in the diagram represents a joint state of the two processes. Initially, the state is at p, with neither process having executed any instructions. If the scheduler chooses to run A first, we get to the point q, in which A has executed some number of instructions, but B has executed none. At point q the trajectory becomes vertical, indicating that the scheduler has chosen to run B. With a single processor, all paths must be horizontal or vertical, never diagonal. Furthermore, motion is always to the north or east, never to the south or west (processes cannot run backward).

When A crosses the I_1 line on the path from r to s, it requests and is granted the printer. When B reaches point t, it requests the plotter.

The regions that are shaded are especially interesting. The region with lines slanting from southwest to northeast represents both processes having the printer. The mutual exclusion rule makes it impossible to enter this region. Similarly, the region shaded the other way represents both processes having the plotter and is equally impossible.

If the system ever enters the box bounded by I_1 and I_2 on the sides and I_5 and I_6, top and bottom, it will eventually deadlock when it gets to the intersection of I_2 and I_6. At this point, A is requesting the plotter and B is requesting the printer, and both are already assigned. The entire box is unsafe and must not be entered. At point r the only safe thing to do is run process A until it gets to I_4. Beyond that, any trajectory to u will do.

The Banker's Algorithm for Multiple Resources

This graphical model is difficult to apply to the general case of an arbitrary number of processes and an arbitrary number of resource classes, each with multiple instances (e.g., two plotters, three tape drives). However, the banker's algorithm can be generalized to do the job. Figure 3-13 shows how it works. In Fig. 3-13 we see two matrices. The one on the left shows how many of each resource is currently assigned to each of the live processes. The matrix on the right shows how many resources each process still needs in order to complete. As in the single resource case. processes must state their total resource needs before executing, so that the system can compute the right-hand matrix at each step.

Process	Tape drives	Plotters	Printers	CD-ROMS
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	CD-ROMS
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

Figure 3-13. The banker's algorithm with multiple resources.

The three vectors at the right of the figure show the existing resources, E, the possessed resources, P, and the available resources. A, respectively. From E we see that the system has six tape drives, three plotters, four printers, and two CD-ROMs. Of these, five tape drives, three plotters, two printers, and two CD-ROMs are currently assigned. This fact can be seen by adding up the four resource columns in the left-hand matrix. The available resource vector is simply the difference between what the system has and what is currently in use.

The algorithm for checking to see if a state is safe can now be stated.

1. Look for a row, *R*, whose unmet resource needs are all smaller than or equal to *A*. If no such row exists, the system will eventually deadlock since no process can run to completion.

2. Assume the process of the row chosen requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to the *A* vector.

3. Repeat steps 1 and 2 until either all processes are marked terminated, in which case the initial state was safe, or until a deadlock occurs, in which case it was not.

If several processes are eligible to be chosen in step 1, it does not matter which one is selected: the resource pool either gets larger, or at worst, stays the same.

Now let us get back to the example of Fig. 3-13. The current state is safe. Suppose that process *B* now requests a printer. This request can be granted because the resulting state is still safe (process *D* can finish, and then processes *A* or *E*, followed by the rest).

Now imagine that after giving *B* one of the two remaining printers, *E* wants to have the last printer. Granting that request would reduce the vector of available resources to (1 0 0 0), which leads to deadlock. Clearly *E*'s request may not be satisfied immediately and must be deferred for a while.

This algorithm was first published by Dijkstra in 1965. Since that time, nearly every book on operating systems has described it in detail. Innumerable papers have been written about various aspects of it. Unfortunately, few authors have had the audacity to point out that although in theory the algorithm is wonderful, in practice it is essentially useless because processes rarely know what their maximum resource needs will be in advance. In addition, the number of processes is not fixed, but dynamically varying as new users log in and out. Furthermore, resources that were thought to be available can suddenly vanish (tape drives can break).

In summary, the schemes described earlier under the name "prevention" are overly restrictive, and the algorithm described here as "avoidance" requires information that is usually not available. If you can think of a general-purpose algorithm that does the job in practice as well as in theory, write it up and send it to your local computer science journal.

For specific applications, many excellent special-purpose algorithms are known. As an example, in many data base systems, an operation that occurs frequently is requesting locks on several records and then updating all the locked records. When multiple processes are running at the same time, there is a real danger of deadlock.

The approach often used is called **two-phase locking**. In the first phase, the process tries to lock all the records it needs, one at a time. If it succeeds, it performs its updates and releases the locks. If some record is already locked, it releases the locks it already has and just starts all over. In a certain sense. this approach is similar to requesting all the resources needed in advance, or at least before anything irreversible is done.

However, this strategy is not applicable in general. In real-time systems and process control systems, for example, it is not acceptable to just terminate a process partway through because a resource is not available and start all over again. Neither is it acceptable to start over if the process has read or written messages to the network, updated files, or anything else that cannot be safely repeated. The algorithm works only in those situations where the programmer has very carefully arranged things so that the program can be stopped at any point during the first phase and restarted. Unfortunately, not all applications can be structured in this way.

3.4 Overview of I/O in MINIX

MINIX I/O is structured as shown in Fig. 3-6. The top four layers of that figure correspond to the four-layered structure of MINIX shown in Fig. 2-26. In the following sections we will look briefly at each of the layers, with the emphasis on the device drivers. Interrupt handling was covered in the previous chapter, and the device-independent I/O will be discussed when we come to the file system, in Chap. 5.

3.4.1 Interrupt Handlers in MINIX

Many of the device drivers start some I/O device and then block, waiting for a message to arrive. That message is usually generated by the interrupt handler for the device. Other device drivers do not start any physical l/O (e.g., reading from RAM disk and writing to a memory-mapped display), do not use interrupts, and do not wait for a message from an I/O device. In the previous chapter the mechanism by which interrupts generate messages and cause task switches has been presented in great detail, and we will say no more about it here. But interrupt handlers may do more than just generate a message. Frequently they also do some work in processing input and output at the lowest level. We will discuss this in a general way here and then return to the details when we look at the code for various devices.

For disk devices, input and output is generally a matter of commanding a device to perform its operation, and then waiting until the operation is complete. The disk controller does most of the work, and very little is required of the interrupt handler. We saw that the entire interrupt handler for the hard disk task consists of just three lines of code, with the only I/O operation being the reading of a single byte to determine the status of the controller. Our lives would be simple indeed if all interrupts could be handled so easily.

However, there is sometimes more for the low-level handler to do. The message passing mechanism has a cost. When an interrupt may occur frequently but the amount of l/O handled per interrupt is small, it may pay to make the handler itself do somewhat more work and to postpone sending a message to the task until a subsequent interrupt, when there is more for the task to do. MINIX handles interrupts from the clock this way. On many clock ticks there is very little to be done, except for maintaining the time. This can be done without sending a message to the clock task itself. The clock handler increments a variable, appropriately named *pending_ticks*. The current time is the sum of the time recorded when the clock task itself last ran plus the value of *pending_ricks*. When the clock task receives a message and wakes up, it adds *pending_ticks* to its main timekeeping variable and then zeroes *pending_ricks*. The clock interrupt handler examines some other variables and sends a message to the clock task only when it detects the task has actual work to do, such as delivering an alarm or scheduling a new process to execute. It may also send a message to the terminal task.

In the terminal task we see another variation on the theme of interrupt handlers. This task handles several different kinds of hardware, including the keyboard and the RS-232 lines. These each have their own interrupt handler. The keyboard exactly fits the description of a device where there may be relatively little I/O to do in response to each interrupt. On a PC an interrupt occurs each time a key is pressed or released. This includes special keys like the SHIFT and CTRL keys, but if we ignore them for the moment, we can say that on the average half a character is received per interrupt. Since there is not much the terminal task can do with half a character, it makes sense to send it a message only when something worthwhile can be accomplished. We will examine the details later;

for now we will just say that the keyboard interrupt handler does the low-level reading of data from the keyboard and then filters out events it can ignore, such as the release of an ordinary key. (The release of a special key, for instance, the SHIFT key, cannot be ignored.) Then codes representing all nonignored events are placed in a queue for later processing by the terminal task itself.

The keyboard interrupt handler differs from the simple paradigm we have presented of the interrupt handler that sends a message to its associated task, because the interrupt handler sends no message at all. Instead, when it adds a code to the queue, it modifies a variable, *tty_timeout*, that is read by the clock interrupt handler. When an interrupt does not change the queue, *tty_timeout* is not changed either. On the next clock tick the clock handler sends a message to the terminal task if there have been changes to the queue. Other terminal-type interrupt handlers, for instance those for the RS-232 lines, work the same way. A message to the terminal task will arrive soon after a character is received, but a message is not necessarily generated for each character when characters are arriving rapidly. Several characters may accumulate and then be processed in response to a single message. Moreover, all terminal devices are checked each time a message is received by the terminal task.

3.4.2 Device Drivers in MINIX

For each class of I/O device present in a MINIX system, a separate I/O task (device driver) is present. These drivers are full-fledged processes, each with its own state, registers, stack, and so on. Device drivers communicate with each other (where necessary) and with the file system using the standard message passing mechanism used by all MINIX processes. Simple device drivers are written as single source files, such as *clock.c*. For other drivers, such as the drivers for the RAM disk, the hard disk, and the floppy disk, there is a source file to support each type of device, as well as a set of common routines in *driver.c* to support all of the different hardware types. In a sense this divides the device driver level of Fig. 3-6 into two sublevels. This separation of the hardware-dependent and hardware-independent parts of the software makes for easy adaptation to a variety of different hardware configurations. Although some common source code is used, the driver for each disk type runs as a separate process, in order to support rapid data transfers.

The terminal driver source code is organized in a similar way, with hardware-independent code in *tty.c* and source code to support different devices, such as memory-mapped consoles, the keyboard, serial lines, and pseudo terminals in separate files. In this case, however, a single process supports all of the different device types.

For groups of devices such as disk devices and terminals, for which there are several source files, there are also header files. *Driver.h* supports all the block device drivers. *Try.h* provides common definitions for all the terminal devices.

The main difference between device drivers and other processes is that the device drivers are linked together in the kernel, and thus all share a common address space. As a result, if several device drivers use a common procedure, only one copy will be linked into the MINIX binary.

This design is highly modular and moderately efficient. It is also one of the few places where MINIX differs from UNIX in an essential way. In MINIX a process reads a file by sending a message to the file system process. The file system, in turn, may send a message to the disk driver asking it to read the needed block. This sequence (slightly simplified from reality) is shown in Fig. 3-14(a). By making these interactions via the message mechanism, we force various parts of the system to interface in standard ways with other

parts. Nevertheless, by putting all the device drivers in the kernel address space, they have easy access to the process table and other key data structures when needed.

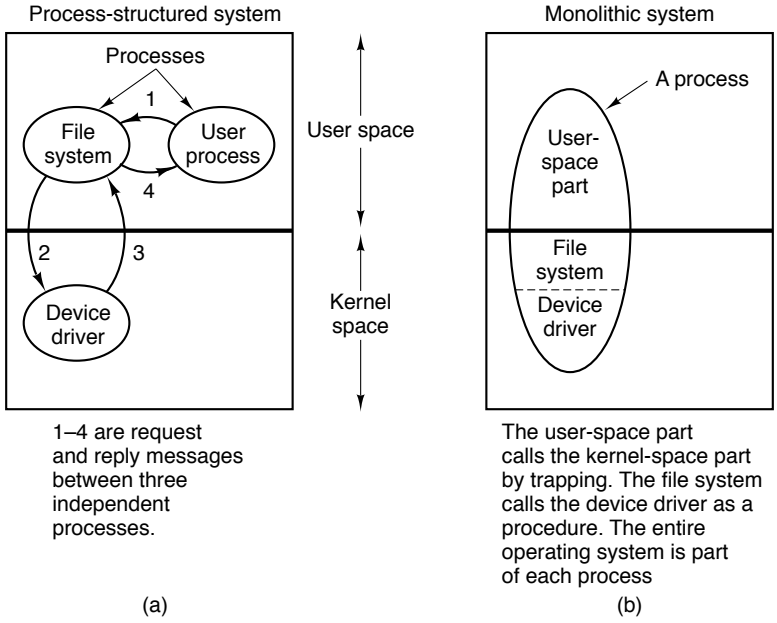


Figure 3-14. Two ways of structuring user-system communication.

In UNIX all processes have two parts: a user-space part and a kernel-space part, as shown in Fig. 3-14(b). When a system call is made, the operating system switches from the user-space part to the kernel-space part in a somewhat magical way. This structure is a remnant of the MULTICS design, in which the switch was just an ordinary procedure call, rather than a trap followed by saving the state of the user-part, as it is in UNIX.

Device drivers in UNIX are simply kernel procedures that are called by the kernel-space part of the process. When a driver needs to wait for an interrupt, it calls a kernel procedure that puts it to sleep until some interrupt handler wakes it up. Note that it is the user process itself that is being put to sleep here, because the kernel and user parts are really different parts of the same process.

Among operating system designers, arguments about the merits of monolithic systems, as in UNIX, versus process-structured systems, as in MINIX, are endless. The MINIX approach is better structured (more modular), has cleaner interfaces between the pieces, and extends easily to distributed systems in which the various processes run on different computers. The UNIX approach is more efficient, because procedure calls are much faster than sending messages. MINIX was split into many processes because we believe that with increasingly powerful personal computers available, cleaner software structure was worth making the system slightly slower. Be warned that many operating system designers do not share this belief.

In this chapter, drivers for RAM disk, hard disk, clock, and terminal are discussed. The standard MINIX configuration also includes drivers for floppy disk and printer, which are not discussed in detail. The MINIX software distribution contains source code for additional drivers for RS-232 serial lines, a SCSI interface, CD-ROM, Ethernet adapter, and sound card. These may be included by recompiling MINIX.

All of these tasks interface with other parts of the MINIX system in the same way: request messages are sent to the tasks. The messages contain a variety of fields used to hold the operation code (e.g., READ or WRITE) and its parameters. A task attempts to

fulfill a request and returns a reply message.

For block devices, the fields of the request and reply messages are shown in Fig. 3-15. The request message includes the address of a buffer area containing data to be transmitted or in which received data are expected. The reply includes status information so the requesting process can verify that its request was properly carried out. The fields for the character devices are basically similar but can vary slightly from task to task. Messages to the clock task, for example, contain times, and messages to the terminal task can contain the address of a data structure which specifies all of the many configurable aspects of a terminal, such as the characters to use for the intraline editing functions erase-character and kill-line.

Requests		
Field	**Type**	**Meaning**
m.m_type	int	Operation requested
m.DEVICE	int	Minor device to use
m.PROC_NR	int	Process requesting the I/O
m.COUNT	int	Byte count or ioctl code
m.POSITION	long	Position on device
m.ADDRESS	char*	Address within requesting process

Replies		
Field	**Type**	**Meaning**
m.m_type	int	Always TASK_REPLY
m.REP_PROC_NR	int	Same as PROC_NR in request
m.REP_STATUS	int	Bytes transferred or error number

Figure 3-15. Fields of the messages sent by the file system to the block device drivers and fields of the replies sent back.

The function of each task is to accept requests from other processes, normally the file system, and carry them out. All the block device tasks have been written to get a message, carry it out, and send a reply. Among other things, this decision means that these tasks are strictly sequential and do not contain any internal multiprogramming, to keep them simple. When a hardware request has been issued, the task does a RECEIVE operation specifying that it is interested only in accepting interrupt messages, not new requests for work. Any new request messages are just kept waiting until the current work has been done (rendezvous principle). The terminal task is slightly different, since a single task services several devices. Thus, it is possible to accept a new request for input from the keyboard while a request to read from a serial line is still being fulfilled. Nevertheless, for each device a request must be completed before beginning a new one.

The main program for each block device driver is structurally the same and is outlined in Fig. 3-16. When the system first comes up, each of the drivers is started up in turn to give each a chance to initialize internal tables and similar things. Then each driver task blocks by trying to get a message. When a message comes in, the identity of the caller is

saved, and a procedure is called to carry out the work, with a different procedure invoked for each operation available. After the work has been finished, a reply is sent back to the caller, and the task then goes back to the top of the loop to wait for the next request.

```
void io_task() {
  initialize();                  /* only done once, during system init. */
  while (TRUE) {
        receive(ANY, &mess);/* wait for a request for work */
        caller = mess.source;/* process from whom message came */
        switch(mess.type) {
            case READ:    rcode = dev_read(&mess); break;
            case WRITE:   rcode = dev_write(&mess); break;
            /* Other cases go here, including OPEN, CLOSE, and IOCTL */
             default:          rcode = ERROR;
        }
        mess.type = TASK_REPLY;
        mess.status = rcode;/* result code */
        send(caller, &mess);/* send reply message back to caller */
  }
}
```

Figure 3-16. Outline of the main procedure of an I/O task.

Each of the *dev_xxx* procedures handles one of the operations of which the driver is capable. It returns a status code telling what happened. The status code, which is included in the reply message as the field *REP_STATUS*, is the count of bytes transferred (zero or positive) if all went well, or the error number (negative) if something went wrong. This count may differ from the number of bytes requested. When the end of a file is reached, the number of bytes available may be less than number requested. On terminals at most one line is returned, even if the count requested is larger.

3.4.3 Device-Independent I/O Software in MINIX

In MINIX the file system process contains all the device-independent I/O code. The I/O system is so closely related to the file system that they were merged into one process. The functions performed by the file system are those shown in Fig. 3-5, except for requesting and releasing dedicated devices, which do not exist in MINIX as it is presently configured. They could, however, easily be added to the relevant device drivers should the need arise in the future.

In addition to handling the interface with the drivers, buffering, and block allocation, the file system also handles protection and the management of i-nodes, directories, and mounted file systems. It will be covered in detail in Chap. 5.

3.4.4 User-level I/O Software in MINIX

The general model outlined earlier in this chapter also applies here. Library procedures are available for making system calls and for all the C functions required by the POSIX standard, such as the formatted input and output functions *printf* and *scanf* The standard MINIX configuration contains one spooler daemon, *lpd*, which spools and prints files passed to it by the *lp* command. The standard MINIX software distribution contains

a number of daemons that support various network functions. Network operations require some operating system support that is not part of MINIX in the configuration described in this book, but MINIX can easily be recompiled to add the network server. It runs at the same priority as the memory manager and the file system, and like them, it runs as a user process.

3.4.5 Deadlock Handling in MINIX

True to its heritage, MINIX follows the same path as UNIX with respect to deadlocks: it just ignores the problem altogether. MINIX contains no dedicated I/O devices, although if someone wanted to hang an industry standard DAT tape drive on a PC, making the software for it would not pose any special problems. In short, the only place deadlocks can occur are with the implicit shared resources, such as process table slots, i-node table slots, and so on. None of the known deadlock algorithms can deal with resources like these that are not requested explicitly.

Actually, the above is not strictly true. Accepting the risk that user processes could deadlock is one thing, but within the operating system itself a few places do exist where considerable care has been taken to avoid problems. The main one is the interaction between the file system and the memory manager. The memory manager sends messages to the file system to read the binary file (executable program) during an EXEC system call, as well as in other contexts. If the file system is not idle when the memory manager is trying to send to it, the memory manager will be blocked. If the file system should then try to send a message to the memory manager, it too would discover that the rendezvous fails and would block, leading to a deadlock.

This problem has been avoided by constructing the system in such a way that the file system never sends *request* messages to the memory manager, just *replies*, with one minor exception. The exception is that upon starting up, the file system reports the size of the RAM disk to the memory manager, which is guaranteed to be waiting for the message.

It is possible to lock devices and files even without operating system support. A file name can serve as a truly global variable, whose presence or absence can be noted by all other processes. A special directory, */usr/spool/locks/* is usually present on MINIX systems, as on most UNIX systems, where processes can create **lock tiles**, to mark any resources they are using. The MINIX file system also supports POSIX-style advisory file locking. But neither of these mechanisms is enforceable. They depend upon the good behavior of processes, and there is nothing to prevent a program from using a resource that is locked by another process. This is not exactly the same thing as preemption of the resource, because it does not prevent the first process from attempting to continue its use of the resource. In other words, there is no mutual exclusion. The result of such an action by an ill-behaved process is likely to be a mess, but no deadlock results.

3.5 Block Devices in MINIX

In the following sections we will return to the device drivers, the main topic of this chapter, and study several of them in detail. MINIX supports several different block devices, so we will begin by discussing common aspects of all block devices. Then we will discuss the RAM disk, the hard disk, and the floppy disk. Each of these is interesting for a different reason. The RAM disk is a good example to study because it has all the properties of block devices in general except the actual I/O—because the “disk” is actually

just a portion of memory. This simplicity makes it a good place to start. The hard disk shows what a real disk driver looks like. One might expect the floppy disk to be easier to support than the hard disk, but, in fact, it is not. We will not discuss all the details of the floppy disk, but we will point out several of the complications to be found in the floppy disk driver.

Following the discussion of block drivers, we will discuss other driver classes. The clock is important because every system has one, and because it is completely different from all the other drivers. It is also of interest as an exception to the rule that all devices are either block or character devices, because it does not fit into either category. Finally, we will discuss the terminal driver, which is important on all systems, and furthermore, is a good example of a character device driver.

Each of these sections describes the relevant hardware, the software principles behind the driver, an overview of the implementation, and the code itself. This structure may make the sections useful reading even for readers who are not interested in the details of the code itself.

3.5.1 Overview of Block Device Drivers in MINIX

We mentioned earlier that the main procedures of all I/O drivers have a similar structure. MINIX always has at least three block device tasks (the RAM disk driver, the floppy driver, and one of several possible hard disk drivers) compiled into the system. In addition, a CD-ROM task and a SCSI (Small Computer Standard Interface) driver maybe compiled in, if support for such device is needed. Although the driver for each of these executes as an independent process, the fact that they are all compiled as part of the kernel executable makes it possible to share a considerable amount of the code, especially the utility procedures.

Each block device driver has to do some initialization, of course. The RAM disk driver has to reserve some memory, the hard disk driver has to determine the parameters of the hard disk hardware, and so on. All of the disk drivers are called individually for hardware-specific initialization. After doing whatever may be necessary, each driver then calls the function containing its main loop. This loop is executed forever; there is no return to the caller. Within the main loop a message is received, a function to perform the operation needed by each message is called, and then a reply message is generated.

The common main loop called by each disk driver task is not just a copy of a library function compiled into each driver. There is only one copy of the main loop code in the MINIX binary. The technique used is to have each of the individual drivers to pass to the main loop a parameter consisting of a pointer to a table of the addresses of the functions that driver will use for each operation and then call these functions indirectly. This technique also makes is possible for drivers to share functions. Figure 3-17 shows an outline of the main loop, in a form similar to that of Fig. 3-16. Statements

```
code = (*entry_points->dev_read)(&mess);
```

are indirect function calls. A different *dev_read* function is called by each driver, even though each driver is executing the same main loop. But some other operations, for example CLOSE, are simple enough that more than one device can call the same function.

The use of single copy of the loop is a good illustration of the process concept that we introduced in Chap 1 and discussed in Chap 2. There is only one copy of the executable code in memory for the main loop of the block device drivers, but it is executed as the main loop of three or more distinct processes. Each of these processes is probably at a

```
message mess;                /* message buffer */

void shared_io_task(struct driver_table *entry_points) {
/* initialization is done by each task before calling this */
  while (TRUE) {
        receive(ANY, &mess);
        caller = mess.source;
        switch(mess.type) {
            case READ:    rcode = (*entry_points->dev_read)(&mess); break;
            case WRITE:   rcode = (*entry_points->dev_write)(&mess); break;
            /* Other cases go here, including OPEN, CLOSE, and IOCTL */
            default:rcode = ERROR;
        }
        mess.type = TASK_REPLY;
        mess.status = rcode;/* result code */
        send(caller, &mess);
  }
}
```

Figure 3-17. An I/O driver main procedure using indirect calls.

different point in the code at a given instant, and each is operating upon its own set of data and has its own stack.

There are six possible operations that can be requested of any device driver. These correspond to the possible values that can be found in the *m.m_type* field of the message of Fig. 3-15. They are:

1. OPEN
2. CLOSE
3. READ
4. WRITE
5. IOCTL
6. SCATTERED_IO

Many of these operations are most likely familiar to readers with programming experience. At the device driver level most operations are related to system calls with the same name. For instance, the meanings of *READ* and *WRITE* should be fairly clear. For each of these operations, a block of data is transferred from the device to the memory of the process that initiated the call, or vice versa. A *READ* operation normally does not result in a return to the caller until the data transfer is complete, but an operating system may buffer data transferred during a *WRITE* for actual transfer to the destination at a later time, and return to the caller immediately. That is fine as far as the caller is concerned; it is then free to reuse the buffer from which the operating system has copied the data to write. *OPEN* and *CLOSE* for a device have similar meanings to the way the open and close system calls apply to operations on files: an *OPEN* operation should verify that the device is accessible, or return an error message if not, and a *CLOSE* should guarantee that

any buffered data that were written by the caller are completely transferred to their final destination on the device.

The *IOCTL* operation may not be so familiar. Many I/O devices have operational parameters which occasionally must be examined and perhaps changed. *IOCTL* operations do this. A familiar example is changing the speed of transmission or the parity of a communications line. For block devices, *IOCTL* operations are less common. Examining or changing the way a disk device is partitioned is done using an *IOCTL* operation in MINIX (although it could just as well have been done by reading and writing a block of data).

The *SCATTERED_IO* operation is no doubt the least familiar of these. Except with exceedingly fast disk devices (for example, the RAM disk), satisfactory disk I/O performance is difficult to obtain if all disk requests are for individual blocks, one at a time. A *SCATTERED_IO* request allows the file system to make a request to read or write multiple blocks. In the case of a *READ* operation, the additional blocks may not have been requested by the process on whose behalf the call is made; the operating system attempts to anticipate future requests for data. In such a request not all the transfers requested are necessarily honored by the device driver. The request for each block may be modified by a flag bit that tells the device driver that the request is optional. In effect the file system can say: "It would be nice to have all these data, but I do not really need them all right now." The device can do what is best for it. The floppy disk driver, for instance, will return all the data blocks it can read from a single track, effectively saying, "I will give you these, but it takes too long to move to another track; ask me again later for the rest."

When data must be written, there is no question of its being optional; every write is mandatory. Nevertheless, the operating system may buffer a number of write requests in the hope that writing multiple blocks can be done more efficiently than handling each request as it comes in. In a *SCATTERED_IO* request, whether for reading or writing, the list of blocks requested is sorted, and this makes the operation more efficient than handling the requests randomly. In addition, making only one call to the driver to transfer multiple blocks reduces the number of messages sent within MINIX.

3.5.2 Common Block Device Driver Software

Definitions that are needed by all of the block device drivers are located in *driver.h*. The most important thing in this file is the *driver* structure, on lines 9010 to 9020, which is used by each driver to pass a list of the addresses of the functions it will use to perform each part of its job. Also defined here is the *device* structure (lines 9031 to 9034) which holds the most important information about partitions, the base address, and the size, in byte units. This format was chosen so no conversions are necessary when working with memory-based devices, maximizing speed of response. With real disks there are so many other factors delaying access that converting to sectors is not a significant inconvenience.

The main loop and common functions of all the block device drivers are in *driver.c*. After doing whatever hardware-specific initialization may be necessary, each driver calls *driver_task*, passing a *driver* structure as the argument to the call. After obtaining the address of a buffer to use for DMA operations the main loop (lines 9158 to 9199) is entered.

The file system is the only process that is supposed to send a message to a driver task. The **switch** on lines 9165 to 9175 checks for this. A leftover interrupt from the hardware is ignored, and other misdirected message results only in printing a warning on the screen. This seems innocuous enough, but of course the process that sent the erroneous message is probably permanently blocked waiting for a replay.

In the **switch** statement in the main loop, the first three message types, *DEV_OPEN*, *DEV_CLOSE*, and *DEV_IOCTL*, result in indirect calls using addresses passed in the *driver* structure. The *DEV_OPEN*, *DEV_CLOSE*, and *SCATTERED_IO* messages result in direct calls to *do_rdwt* or *do_vrdwt*. However, the driver structure is passed as an argument by all the calls from within the **switch**, whether direct or indirect, so all called functions can make further use of it as needed.

After doing whatever is requested in the message, some sort of cleanup may be necessary, depending upon the nature of the device. For a floppy disk, for instance, this might involve starting a timer to turn off the disk drive motor if another request does not arrive soon. An indirect call is used for this as well. Following the cleanup, a reply message is constructed and sent to the caller (lines 9194 to 9198).

The first thing each driver does after entering the main loop is to call *init_buffer* (line 9205), which assigns a buffer for use in DMA operations. The same buffer is used by all the driver tasks, if they use it all—some drivers do not use DMA. The initialization for each entry after the first are redundant but do no harm. It would be more cumbersome to code a test to see whether the initialization should be skipped.

That this initialization is even necessary at all is due to a quirk of the hardware of the original IBM PC, which requires that the DMA buffer not cross a 64K boundary. That is, a 1K DMA buffer may begin at 64510, but not at 64514, because a buffer starting at the latter address extends just beyond the 64K boundary at 65536.

This annoying rule occurs because the IBM PC used an old DMA chip, the Intel 8237A, which contains a 16-bit counter. A bigger counter is needed because DMA uses absolute addresses, not addresses relative to a segment register. On older machines that can address only 1M of memory, the low-order 16 bits of the DMA address are loaded into the 8237A, and the high-order 4 bits are loaded into a 4-bit latch. Newer machines use an 8-bit latch and can address 16M. When the 8237A goes from 0xFFFF to 0x0000, it does not generate a carry into the latch, so the DMA address suddenly jumps down by 64K in memory.

A portable C program cannot specify an absolute memory location for a data structure, so there is no way to prevent the compiler from placing the buffer in an unusable location. The solution is to allocate an array of bytes twice as large as necessary at *buffer* (line 9135) and to reserve a pointer *tmp_buf* (line 9135) to use for actually accessing this array. *Init_buffer* makes a trial setting of *tmp_buf* pointing to the beginning of *buffer*, then tests to see if that allows enough space before a 64K boundary is hit. If the trial setting does not provide enough space, *tmp_buf* is incremented by the number of bytes actually required. Thus some space is always wasted at one end or the other of the space allocated in buffer, but there is never a failure due to the buffer falling on a 64K boundary.

Newer computers of the IBM PC family have better DMA controllers, and this code could be simplified, and a small amount of memory reclaimed, if one could be sure that one's machine were immune to this problem. If you are considering this, however, consider how the bug will manifest itself if you are wrong. If a 1K DMA buffer is desired, the chance is 1 in 64 that there will be a problem on a machine with the old DMA chip. Every time the kernel source code is modified in a way that changes the size of the compiled kernel, there is the same probability that the problem will manifest itself. Most likely, when the failure occurs next month or next year, it will be attributed to the code that was last modified. Unexpected hardware "features" like this can cause weeks of time spent looking for exceedingly obscure bugs (all the more so when, like this one, the technical reference manual says nary a word about them).

Do_rdwt is the next function in *driver.c*. It, in turn, may call three device-dependent

functions pointed to by the textitdr_prepare ,*dr_schedule*, and *dr_finish* fields in the *driver* structure. In what follows we will use the C language-like notation **function_pointer* to indicate we are talking about the function pointed to by *function_pointer*.

After checking to see that the byte count in the request is positive, *do_rdwt* calls **dr_prepare*. This should succeed, since **dr_prepare* can fail only if an invalid device is specified in an OPEN operation. Next, a standard *iorequest_s* structure (defined on line 3194 in *include/minix/type.h* is filled in. Then comes another indirect call, this time to **dr_schedule*. As we will see in the discussion of disk hardware in the next section, responding to disk requests in the order they are received can be inefficient, and this routine allows a particular device to handle requests in the way that is best for the device. The indirection here masks much possible variation in the way individual devices perform. For the RAM disk, *dr_schedule* points to a routine that actually performs the I/O, and the next indirect call, to *dr_finish*, is a do-nothing operation. For a real disk, *dr_finish* points to a routine that carries out all of the pending data transfers requested in all previous calls to **dr_schedule* since the last call to **dr_finish*. As we will see, however, in some circumstances the call to **dr_finish* may not result in a transfer of all the data requested.

In whatever call does an actual data transfer, the *io_nbytes* count in the *iorequest_s* structure is modified, returning an a negative number if there was an error or a positive number indicating the difference between the number of bytes in the original request and the number successfully transferred. It is not necessarily an error if no bytes are transferred; this indicates that the end of the device has been reached. Upon returning to the main loop, the negative error code or the byte count is returned in the reply message *REP_STATUS* field if there was an error. Otherwise the bytes remaining to be transferred are subtracted from the original request in the *COUNT* field of the message (line 9249), and the result (the number actually transferred) is returned in *REP_STATUS* in the reply message from *driver_task*.

The next function, *do_vrdwt*, handles all scattered I/O requests. A message that requests a scattered I/O request uses the *ADDRESS* field to point to an array of *iorequest_s* type structures, each of which specifies the information needed for one request: the address of the buffer, the offset on the device, the number of bytes, and whether the operation s a read or a write. All the operations in one request will be for either reading or writing, and they will be sorted into block order on the device. There is more work to do than for the simple read or write performed by *do_rdwt*, since the array of requests must be copied into the kernel's space, but once this has been done, the same three indirect calls to the device-dependent **dr_prepare*, **dr_schedule*, and **dr_finish* routines are made. The difference is that the middle call, to **dr_schedule*, is done in a loop, once for each request, or until an error occurs (line 9288 to 9290). After termination of the loop, **dr_finish* is called once, and then the array of requests is copied back where it came from. The *io_nbytes* field of each element in the array will have been changed to reflect the number of bytes transferred, and although the total is not passed back directly in the reply message that *driver_task* constructs, the caller can extract the total from this array.

In a scattered I/O read request, not all the transfers requested in the call to **dr_schedule* are necessarily honored when the final call to **dr_finish* is made, as we discussed in the previous section. The *io_request* field in the *iorequest_s* strcuture contains a flag bit that tells the device driver if a request for the block is optional.

The next few routines in *driver.c* are for general support of the above operations. A **dr_name* call can be used to return the name of a device. For a device with no specific name the *no_name* function retrieves the device's name from the table of tasks. Some devices may not require a particular service, for instance, a RAM disk does not require

that anything special be done upon a *DEV_CLOSE* request. The *do_nop* function fills in here, returning various codes depending upon the kind of request made. The following functions, *nop_finish*, and *nop_cleanup*, are similar dummy routines for devices that need no **dr_finish* or **dr_cleanup* services.

Some disk device funtions require delays, for example, to wait for a floppy disk motor to come up to speed. Thus *driver.c* is a good place for the next function, *clock_mess*, used to send messages to the clock task. It is called with the number of clock ticks to wait and the address of a function to call when the timeout period is complete.

Finally, *do_diocntl* (line 9364) carries out *DEV_IOCTL* requests for a block device. It is an error if any *DEV_IOCTL* operation other than reading (*DIOCGETP*) or writing (*DIOCSETP*) partition information is requested. *Do_diocntl* calls the device's **dr_prepare* function to verify the device is valid and to get a pointer to the device structure that describes the partition base and size in byte units. On a request to read, it calls the device's **dr_geometry* function to get the last cylinder, head, and sector information about the partition.

3.5.3 The Driver Library

The files *drvlib.h* and *drvlib.c* contain system-dependent code that supports disk partitions on IBM PC compatible computers.

Partitioning allows a single storage device to be divided up into subdevices. It is most commonly used with hard disks, but MINIX provides support for partitioning floppy disks, as well. Some reasons to partition a disk device are:

1. Disk capacity is cheaper per unit in large disks. If two or more operating systems with different file systems are used, it is more economical to partition a single large disk than to install multiple smaller disks for each operating system.

2. Operating systems may have limits to the device size they can handle. The version of MINIX discussed here can handle a 1-GB file system, but older versions are limited to 256 MB. Any disk space beyond that is wasted.

3. Two or more different file systems may be used by an operating system. For example, a standard file system may be used for ordinary files and a differently structured file system may be used for virtual memory swap space.

4. It may be convenient to put a portion of a system's files on a separate logical device. Putting the MINIX root file system on a small device makes it easy to back up and facilitates copying it to a RAM disk at boot time.

Support for disk partitions is platform specific. This specificity is not related to the hardware. Partition support is device independent. But if more than one operating system is to run on a particular set of hardware, all must agree on a format for the partition table. On IBM PCs the standard is set by the MS-DOS *fdisk* command, and other OSs, such as MINIX, OS/2, and Linux, use this format so they can coexist with MS-DOS. When MINIX is ported to another machine type, it makes sense to use a partition table format compatible with other operating systems used on the new hardware. Thus the MINIX source code to support partitions on IBM computers is put in *drvlib.c*, rather than being included in *driver.c*, to make it easier to port MINIX 3 to different hardware.

The basic data structure inherited from the firmware designers is defined in *include/ibm/partition.h*, which is included by a **#include** statement in *drvlib.h*. This includes information on the

cylinder-head-sector geometry of each partition, as well as codes identifying the type of file system on the partition and an active flag indicating if it is bootable. Most of this information is not needed by MINIX once the file system is verified.

The partition function (in *drvlib.c*, line 9521) is called the first time a block device is opened. Its arguments include a *driver* structure, so it can call device-specific functions, an initial minor device number, and a parameter indicating whether the partitioning style is floppy disk, primary partition, or subpartition. It calls the device-specific **dr_prepare* function to verify the device is valid and to get the base address and the size into a *device* structure of the type mentioned in the previous section. Then it calls *get_part_table* to determine if a partition table is present and, if so, to read it. If there is no partition table, the work is done. Otherwise the minor device number of the first partition is computed, using the rules for numbering minor devices that apply to the style of partitioning specified in the original call. In the case of primary partitions the partition table is sorted so the order of the partitions is consistent with that used by other operating systems.

At this point another call is made to **dr_prepare*, this time using the newly calculated device number of the first partition. If the subdevice is valid, then a loop is made over all the entries in the table, checking that the values read from the table on the device are not out of the range obtained earlier for the base and size of the entire device. If there is a discrepancy, the table in memory is adjusted to conform. This may seem paranoid, but since partition tables may be written by different operating systems, a programmer using another system may have cleverly tried to use the partition table for something unexpected or there could be garbage in the table on disk for some other reason. We put the most trust in the numbers we calculate using MINIX. Better safe than sorry.

Still within the loop, for all partitions on the device, if the partition is identified as a MINIX partition, *partition* is called recursively to gather subpartition information. If a partition is identified as an extended partition, the next function, *extpartition*, is called instead.

Extpartition (line 9593) has nothing to do with MINIX operating system itself, so we will not discuss details. MS-DOS uses extended partitions, which are just another mechanism for creating subpartitions. In order to support MINIX commands that can read and write MS-DOS files, we need to know about these subpartitions.

Get_part_table (line 9642) calls *do_rdwt* to get the sector on a device (or subdevice) where a partition table is located. The offset argument is zero if it is called to get a primary partition or nonzero for a subpartition. It checks for the magic number (0xAA55) and returns true or false status to indicate whether a valid partition table was found. If a table is found, it copies it to the table address that was passed as an argument.

Finally, *sort* (line 9676) sorts the entries in a partition table by lowest sector. Entries that are marked as having no partition are excluded from the sort, so they come at the end, even though they may have a zero value in their low sector field. The sort is a simple bubble sort; there is no need to use a fancy algorithm to sort a list of four items.

3.6 Ram Disks

Now we will get back to the individual block device drivers and study several of them in detail. The first one we will look at is the RAM disk driver. It can be used to provide access to any part of memory. Its primary use is to allow a part of memory to be reserved for use like an ordinary disk. This does not provide permanent storage, but once files have been copied to this area they can be accessed extremely quickly.

In a system such as MINIX, which was designed to work even on computers with only one floppy disk, the RAM disk has another advantage. By putting the root device on the RAM disk, the one floppy disk can be mounted and unmounted at will, allowing for removable media. Putting the root device on the floppy disk would make it impossible to save files on floppies, since the root device (the only floppy) cannot be unmounted. In addition, having the root device on the RAM disk makes the system highly flexible: any combination of floppy disks or hard disks can be mounted on it. Although most computers now have hard disks, except computers used in embedded systems, the RAM disk is useful during installation, before the hard disk is ready for use by MINIX, or when it is desired to use MINIX temporarily without doing a full installation.

3.6.1 RAM Disk Hardware and Software

The idea behind a RAM disk is simple. A block device is a storage medium with two commands: write a block and read a block. Normally, these blocks are stored on rotating memories, such as floppy disks or hard disks. A RAM disk is simpler. It just uses a preallocated portion of main memory for storing the blocks. A RAM disk has the advantage of having instant access (no seek or rotational delay), making it suitable for storing programs or data that are frequently accessed.

As an aside, it is worth briefly pointing out a difference between systems that support mounted file systems and those that do not (e.g., MS-DOS and WINDOWS). With mounted file systems, the root device is always present and in a fixed location, and removable file systems (i.e., disks) can be mounted in the file tree to form an integrated file system. Once everything has been mounted, the user need not worry at all about which device a file is on.

In contrast, with systems like MS-DOS, the user must specify the location of each file, either explicitly as in *B:\DIR\FILE* or by using certain defaults (current device, current directory, and so on). With only one or two floppy disks, this burden is manageable, but on a large computer system, with dozens of disks, having to keep track of devices all the time would be unbearable. Remember that UNIX runs on systems ranging from an IBM-PC, through workstations and supercomputers up to the Cray-2; MS-DOS runs only on small systems.

Figure 3-18 shows the idea behind a RAM disk. The RAM disk is split up into n blocks, depending on how much memory has been allocated for it. Each block is the same size as the block size used on the real disks. When the driver receives a message to read or write a block, it just computes where in the RAM disk memory the requested block lies and reads from it or writes to it, instead of from or to a floppy or hard disk. The transfer is done by calling an assembly language procedure that copies to or from the user program at the maximum speed of which the hardware is capable.

A RAM disk driver may support several areas of memory used as RAM disk, each distinguished by a different minor device number. Usually, these areas are distinct, but in some situations it may be convenient to have them overlap, as we shall see in the next section.

3.6.2 Overview of the RAM Disk Driver in MINIX

The RAM disk driver is actually four closely related drivers in one. Each message to it specifies a minor device as follows:

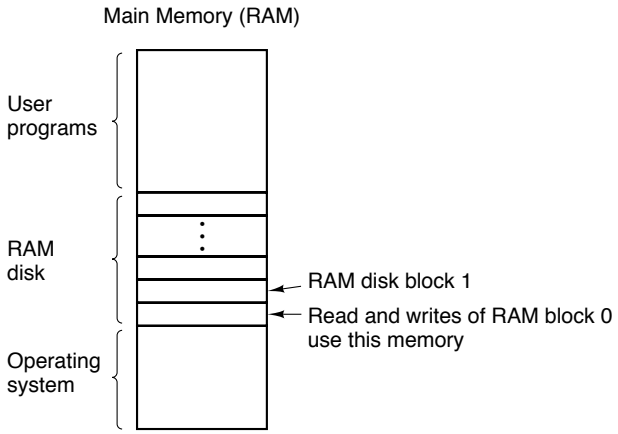


Figure 3-18. A RAM disk.

```
0: /dev/ram    1: /dev/mem    2: /dev/kmem    3: /dev/null
```

The first special file listed above, */dev/ram*, is a true RAM disk. Neither its size nor its origin is built into the driver. They are determined by the file system when MINIX is booted. By default a RAM disk of the same size as the root file system image device is created, so the root tile system can be copied to it. A boot parameter can be used to specify a RAM disk larger than the root file system, or if the root is not to be copied to the RAM, the specified size may be any value that fits in memory and leaves enough memory for system operation. Once the size is known a block of memory big enough is found and removed from the memory pool, even before the memory manager begins its work. This strategy makes it possible to increase or reduce the amount of RAM disk present without having to recompile the operating system.

The next two minor devices are used to read and write physical memory and kernel memory, respectively. When */dev/mem* is opened and read, it yields the contents of physical memory locations starting at absolute address zero (the real-mode interrupt vectors). Ordinary user programs never do this, but a system program concerned with debugging the system might need this facility. Opening */dev/mem* and writing on it will change the interrupt vectors. Needless to say, this should only be done with the greatest of caution by an experienced user who knows exactly what he is doing.

The special file */dev/kmem* is like */dev/mem*, except that byte 0 of this file is byte 0 of the kernel's data memory, a location whose absolute address varies, depending on the size of the MINIX kernel code. It too is used mostly for debugging and very special programs. Note that the RAM disk areas covered by these two minor devices overlap. If you know exactly how the kernel is placed in memory, you can open */dev/mem*, seek to the beginning of the kernel's data area, and see exactly the same thing as reading from the beginning of */dev/kmem*. But, if you recompile the kernel, changing its size, or if in a subsequent version of MINIX the kernel is moved somewhere else in memory, you will have to seek a different amount in */dev/mem* to see the same thing you see at the start of */dev/kmem*. Both of these special files should be protected to prevent everyone except the superuser from using them.

The last file in this group, */dev/null*, is a special file that accepts data and throws them away. It is commonly used in shell commands when the program being called generates output that is not needed. For example,

```
a.out >/dev/null
```

runs the program *a.out* but discards its output. The RAM disk driver effectively treats this minor device as having zero size, so no data are, ever copied to or from it.

The code for handling */dev/ram*, */dev/mem*, and */dev/kmem* is identical. The only difference among them is that each one corresponds to a different portion of memory, indicated by the arrays *ram_origin* and *ram_limit*, each indexed by minor device number.

3.6.3 Implementation of the RAM Disk Driver in MINIX

As with other disk drivers the main loop of the RAM disk is in the file *driver.c*. The device-specific support for memory devices is in *memory.c*. The array *m_geom* (line 9721) holds the base and size of each of the four memory devices. The driver structure *m_dtab* on lines 9733 to 9743 defines the memory device calls that will be made from the main loop. Four of the entries in this table are do-little or do-nothing routines in *driver.c*, a sure clue that the operation of a RAM disk is not terribly complicated. The main procedure *mem_task* (line 9749) calls one function to do some local initialization. After that, it calls the main loop, which gets messages, dispatches to the appropriate procedure, and sends the replies. There is no return to *mem_task* upon completion.

On a read or write operation the main loop makes three calls: one to prepare a device, one to schedule the I/O operations, and one to finish the operation. For a memory device a call to *m_prepare* is the first of these. It checks that a valid minor device has been requested and then returns the address of the structure that holds the base address and size of the requested RAM area. The second call is for *m_schedule* (line 9774). This does all the work. For memory devices the name of this function is a misnomer; by definition, any location is as accessible as any other in random access memory, and thus there is no need to do any scheduling, as there would be with a disk having a moving arm.

The RAM disk's operation is so simple and fast there is never any reason to postpone a request, and the first thing done by this function is to clear the bit that may be set by a scattered I/0 call to indicate completion of an operation is optional. The destination address passed in the message points to a location in the caller's memory space, and the code at lines 9792 to 9794 converts this into an absolute address in the system memory and then checks that it is a valid address. The actual data transfer takes place on line 9818 or line 9820 and is a straightforward copying of data from one place to another.

A memory device does not need a third step to finish a read or write operation, and the corresponding slot in *m_dtab* is a call to *nop_finish*.

Opening a memory device is done by *m_do_open* (line 9829). The main job is done by calling *m_prepare* to check that a valid device is being referenced. In the case of a reference to */dev/mem* or */dev/kmem*, a call to *enable_iop* (in the file *protect.c*) is made to change the CPU's current privilege level. This is not necessary to access memory. It is a trick to deal with another problem. Recall that Pentium-class CPUs implement four privilege levels. User programs are at the least privileged level. Intel processors also have an architectural feature that is not present in many other systems, a separate set of instructions to address I/O ports. On these processors I/O ports are treated separately from memory. Normally, an attempt by a user process to execute an instruction that addresses an I/O port causes a general protection exception. However, there are valid reasons for MINIX to allow users to write programs that can access ports, especially on small systems. Thus *enable_iop* changes the CPU's I/O Protection Level (IOPL) bits to permit this. The effect is to allow a process permitted to open */dev/mem* or */dev/kmem* the additional privilege of access to I/O ports. On an architecture where I/O devices are addressed as memory locations, the *rwx* bits for these devices automatically cover access

to I/O. If this feature were hidden, it might be considered a security flaw, but now you know about it. If you plan to use MINIX to control a bank security system, you might want to recompile the kernel without this function.

The next function, *m_init* (line 9849), is called only once, when *mem_task* is called for the first time. It sets up the base address and size of */dev/kmem* and it also sets the size of */dev/mem* to 1 MB, l6 MB, or 4 GB-1, depending upon whether MINIX is running in 8088, 80286, or 80386 mode. These sizes are the maximum sizes supported by MINIX and do not have anything to do with how much RAM is installed in the machine.

The RAM disk supports several IOCTL operations in *m_ioctl* (line 9874). The *MIOCRAMSIZE* is a convenient way for the file system to set the RAM disk size. The *MIOCSPSINFO* operation is used by both the file system and the memory manager to set the addresses of their parts of the process table into the *psinfo* table, where the utility program *ps* can retrieve it using a *MIOCSPSINFO* operation. *Ps* is a standard UNIX program whose implementation is complicated by MINIX's microkernel structure, which puts the process table information needed by the program in several different places. The IOCTL system call is a convenient way to handle this problem. Otherwise a new version of *ps* would have to be compiled each time a new version of MINIX were compiled.

The last function in *memory.c* is *m_geometry* (line 9934). Memory devices do not have a geometry of cylinders, tracks, and sectors per track like mechanical disk drives, but in case the RAM disk is asked it will oblige by pretending it does.

3.7 Disks

The RAM disk is a good introduction to disk drivers (Because it is so simple), but real disks present a number of issues that we have not yet touched upon. In the following sections we will first say a few words about disk hardware and then take a look at disk drivers in general and the MINIX hard disk driver in particular. We will not examine the floppy disk driver in detail, but we will go over some of the ways a floppy disk driver differs from a hard disk driver.

3.7.1 Disk Hardware

All real 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 some hard disks. The simplest designs have the same number of sectors on each track. All sectors contain the same number of bytes, although a little thought will make it clear that sectors close to the outer rim of the disk will be physically longer than those close to the hub. The time to read or write each sector will be same, however. The data density is obviously higher on the innermost cylinders, and some disk designs require a change in the drive current to the read-write heads for the inner tracks. This is handled by the disk controller hardware and is not visible to the user (or the implementor of an operating system).

The difference in data density between inner and outer tracks means a sacrifice in capacity. and more sophisticated systems exist. Floppy disk designs that rotate at higher speeds when the heads are over the outer tracks have been tried. This allows more sectors on those tracks, increasing disk capacity. Such disks are not supported by any system for which MINIX is currently available, however. Modern large hard drives also have

more sectors per track on outer tracks than on inner tracks. These are **IDE (Integrated Drive Electronics)** drives, and the sophisticated processing done by the drive's built-in electronics masks the details. To the operating system they appear to have a simple geometry with the same number of sectors on each track.

The drive and controller electronics are as important as the mechanical hardware. The main element of the controller card that is plugged into the computer's backplane is a specialized integrated circuit, really a small microcomputer. For a hard disk the controller card circuitry may be simpler than for a floppy disk, but this is because the hard drive itself has a powerful electronic controller built in. 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 a system with more than one of these hard drives they can operate simultaneously, at least to the extent of transferring between the disk and the controller's buffer memory. Only one transfer between the controller and the system memory is possible at once, however. The ability to perform two or more operations at the same time can reduce the average access time considerably.

Figure 3-19 compares parameters of double-sided, double-density diskettes, the standard storage medium for the original IBM PC, with parameters of a typical medium-capacity hard drive such as might be found on a Pentium-based computer. MINIX uses 1K blocks, so with either of these disk formats the blocks used by the software consist of two consecutive sectors, which are always read or written together as a unit.

Parameter	IBM 360-KB floppy disk	WD 540-MB hard disk
Number of cylinders	40	1048
Tracks per cylinder	2	4
Sectors per track	9	252
Sectors per disk	720	1056384
Bytes per sector	512	512
Bytes per disk	368640	540868608
Seek time (adjacent cylinders)	6 msec	4 msec
Seek time (average case)	77 msec	11 msec
Rotation time	200 msec	13 msec
Motor stop/start time	250 msec	9 sec
Time to transfer 1 sector	22 msec	53 μsec

Figure 3-19. Disk parameters for the original IBM PC 360-KB floppy disk and a Western Digital WD AC2540 540-MB hard disk.

One thing to be aware of in looking at the specifications of modern hard disks is

that the geometry specified, and used by the driver software, may be different than the physical format. The hard disk described in Fig. 3-19, for instance, is specified with "recommended setup parameters" of lO48 cylinders, 16 heads, and 63 sectors per track. The controller electronics mounted on the disk converts the logical head and sector parameters supplied by the operating system to the physical ones used by the disk. This is another example of a compromise designed to maintain compatibility with older systems, in this case old firmware. The designers of the original IBM PC only allotted a 6-bit field for the BIOS ROM's sector count, and a disk that has more than 63 physical sectors per track must work with an artificial set of logical disk parameters. In this case the vendor's specifications state that there are really four heads, and thus it would appear that there are really 252 sectors per track, as indicated in the figure. This is an over-simplification, because disks like these have more sectors on the outermost tracks than on the inner tracks. The disk described in the figure does have four physical heads, but there are actually slightly over 3000 cylinders. The cylinders are grouped in a dozen zones which have from 57 sectors per track in the innermost zones to 105 cylinders per track on the outermost cylinders. These numbers are not to be found in the disk's specifications, and the translations done by the drive's electronics make it unnecessary for us to know such details.

3.7.2 Disk Software

In this section we will look at some issues related to disk drivers in general. First, consider how long it takes to read or write a disk block. The time required is determined by three factors:

1. The seek time (the time to move the arm to the proper cylinder).
2. The rotational delay (the time for the proper sector to rotate under the head).
3. The actual data transfer time.

For most disks, the seek time dominates the other two times, so reducing the mean seek time can improve system performance substantially.

Disk devices are prone to errors. Some kind of error check, a checksum or a cyclic redundancy check, is always recorded along with the data in each sector on a disk. Even the sector addresses recorded when the disk is formatted have check data. Floppy disk controller hardware can report when an error is detected, but the software must then decide what to do about it. Hard disk controllers often take on much of this burden.

Particularly with hard disks, the transfer time for consecutive sectors within a track can be very fast. Thus reading more data than is requested and caching them in memory can be very effective in speeding disk access.

Disk Arm Scheduling Algorithms

If the disk driver accepts requests one at a time and carries them out in that order, that is, **First-Come, First-Served (FCFS)**, little can be done to optimize seek time. However, another strategy is possible when the disk is heavily loaded. It is likely that while the arm is seeking on behalf of one request, other disk requests may be generated by other processes. Many disk drivers maintain a table, indexed by cylinder number, with all the pending requests for each cylinder chained together in a linked list headed by the table entries.

Given this kind of data structure, we can improve upon the first-come, first-served scheduling algorithm. To see how, consider a disk with 40 cylinders. A request comes in to read a block on cylinder 11. While the seek to cylinder 11 is in progress, new requests come in for cylinders 1, 36, 16, 34, 9, and 12, in that order. They are entered into the table of pending requests, with a separate linked list for each cylinder. The requests are shown in Fig. 3-20.

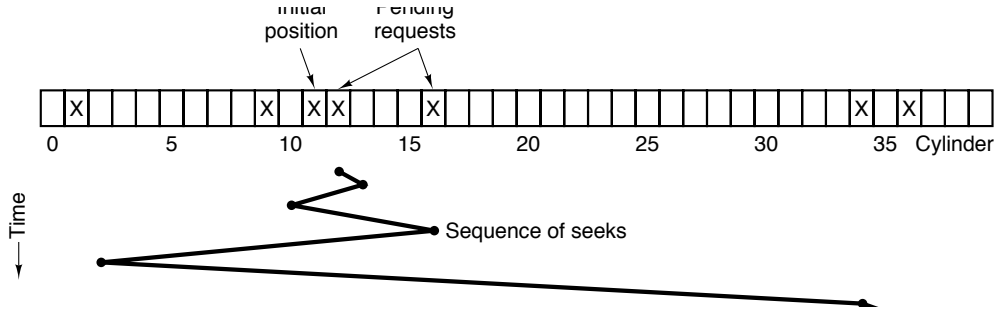


Figure 3-20. Shortest Seek First (SSF) disk scheduling algorithm.

When the current request (for cylinder 11) is finished, the disk driver has a choice of which request to handle next. Using FCFS, it would go next to cylinder 1, then to 36, and so on. This algorithm would require arm motions of 10, 35, 20, 18, 25, and 3, respectively, for a total of 111 cylinders.

Alternatively, it could always handle the closest request next, to minimize seek time. Given the requests of Fig. 3-20, the sequence is 12, 9, 16, 1, 34, and 36, as shown as the jagged line at the bottom of Fig. 3-20. With this sequence, the arm motions are 1, 3, 7, 15, 33, and 2, for a total of 61 cylinders. This algorithm, **Shortest Seek First (SSF)**, cuts the total arm motion almost in half compared to FCFS.

Unfortunately, SSF has a problem. Suppose more requests keep coming in while the requests of Fig. 3-20 are being processed. For example, if, after going to cylinder 16, a new request for cylinder 8 is present, that request will have priority over cylinder 1. If a request for cylinder 13 then comes in, the arm will next go to 13, instead of 1. With a heavily loaded disk, the arm will tend to stay in the middle of the disk most of the time, so requests at either extreme will have to wait until a statistical fluctuation in the load causes there to be no requests near the middle. Requests far from the middle may get poor service. The goals of minimal response time and fairness are in conflict here.

Tall buildings also have to deal with this trade-off. The problem of scheduling an elevator in a tall building is similar to that of scheduling a disk arm. Requests come in continuously calling the elevator to floors (cylinders) at random. The microprocessor running the elevator could easily keep track of the sequence in which customers pushed the call button and service them using FCFS. It could also use SSF.

However, most elevators use a different algorithm to reconcile the conflicting goals of efficiency and fairness. They keep moving in the same direction until there are no more outstanding requests in that direction, then they switch directions. This algorithm, known both in the disk world and the elevator world as the **elevator algorithm**, requires the software to maintain 1 bit: the current direction bit, *UP* or *DOWN*. When a request finishes, the disk or elevator driver checks the bit. If it is *UP*, the arm or cabin is moved to the next highest pending request. If no requests are pending at higher positions, the direction bit is reversed. When the bit is set to *DOWN*, the move is to the next lowest requested position, if any.

Figure 3-21 shows the elevator algorithm using the same seven requests as Fig. 3-20, assuming the direction bit was initially *UP*. The order in which the cylinders are serviced is 12, 16, 34, 36, 9, and 1, which yields arm motions of 1, 4, 18, 2, 27, and 8, for a total of 60 cylinders. In this case the elevator algorithm is slightly better than SSF, although it is usually worse. One nice property that the elevator algorithm has is that given any collection of requests, the upper bound on the total motion is fixed: it is just twice the number of cylinders.

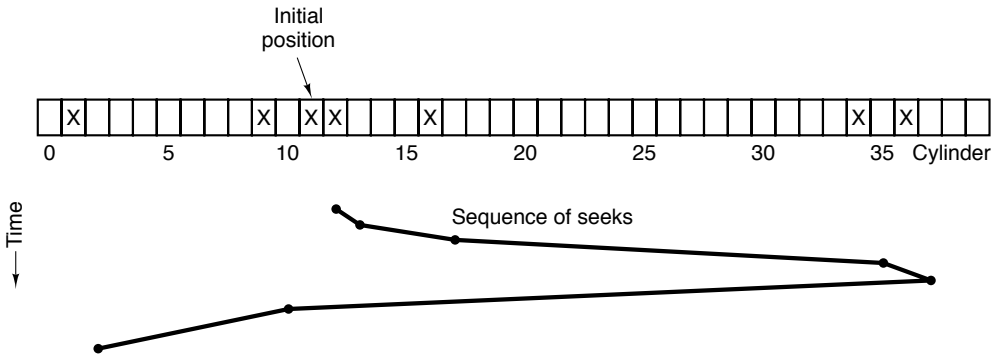


Figure 3-21. The elevator algorithm for scheduling disk requests.

A slight modification of this algorithm that has a smaller variance in response times (Teory, 1972) is to always scan in the same direction. When the highest numbered cylinder with a pending request has been serviced, the arm goes to the lowest-numbered cylinder with a pending request and then continues moving in an upward direction. In effect, the lowest-numbered cylinder is thought of as being just above the highest-numbered cylinder.

Some disk controllers provide a way for the software to inspect the current sector number under the head. With such a controller, another optimization is possible. If two or more requests for the same cylinder are pending, the driver can issue a request for the sector that will pass under the head next. Note that when multiple tracks are present in a cylinder, consecutive requests can be for different tracks with no penalty. The controller can select any of its heads instantaneously, because head selection involves neither arm motion nor rotational delay.

With a modem hard disk, the data transfer rate is so much faster than that of a floppy disk that some kind of automatic caching is necessary. Typically any request to read a sector will cause that sector and up to the rest of the current track to be read, depending upon how much space is available in the controller's cache memory. The 540M disk described in Fig. 3-19 has a 64K or 128K cache. The use of the cache is determined dynamically by the controller. In its simplest mode, the cache is divided into two sections, one for reads and one for writes.

When several drives are present, a pending request table should be kept for each drive separately. Whenever any drive is idle, a seek should be issued to move its arm to the cylinder where it will be needed next (assuming the controller allows overlapped seeks). When the current transfer finishes, a check can be made to see if any drives are positioned on the correct cylinder. If one or more are, the next transfer can be started on a drive that is already on the right cylinder. if none of the arms is in the right place, the driver should issue a new seek on the drive that just completed a transfer and wait until the next interrupt to see which arm gets to its destination first.

Error Handling

RAM disks do not have to worry about seek or rotational optimization: at any instant all blocks can be read or written without any physical motion. Another area in which RAM disks are simpler than real disks is error handling. RAM disks always work; real ones do not always work. They are subject to a wide variety of errors. Some of the more common ones are:

1. Programming error (e.g., request for nonexistent sector).
2. Transient checksum error (e.g., caused by dust on the head).
3. Permanent checksum error (e.g., disk block physically damaged).
4. Seek error (e.g., the arm sent to cylinder 6 but it went to 7).
5. Controller error (e.g., controller refuses to accept commands).

It is up to the disk driver to handle each of these as best it can.

Programming errors occur when the driver tells the controller to seek to a nonexistent cylinder, read from a nonexistent sector, use a nonexistent head, or transfer to or from nonexistent memory. Most controllers check the parameters given to them and complain if they are invalid. In theory, these errors should never occur, but what should the driver do if the controller indicates that one has happened? For a home-grown system, the best thing to do is stop and print a message like "Call the programmer" so the error can be tracked down and fixed. For a commercial software product in use at thousands of sites around the world, this approach is less attractive. Probably the only thing to do is terminate the current disk request with an error and hope it will not recur too often.

Transient checksum errors are caused by specks of dust in the air that get between the head and the disk surface. Most of the time they can be eliminated by just repeating the operation a few times. If the error persists, the block has to be marked as a **bad block** and avoided.

One way to avoid bad blocks is to write a very special program that takes a list of bad blocks as input and carefully hand crafts a file containing all the bad blocks. Once this file has been made, the disk allocator will think these blocks are occupied and never allocate them. As long as no one ever tries to read the bad block file, no problems will occur.

Not reading the bad block file is easier said than done. Many disks are backed up by copying their contents a track at a time to a backup tape or disk drive. If this procedure is followed, the bad blocks will cause trouble. Backing up the disk one file at a time is slower but will solve the problem, provided that the backup program knows the name of the bad block file and refrains from copying it.

Another problem that cannot be solved with a bad block file is the problem of a bad block in a file system data structure that must be in a fixed location. Almost every file system has at least one data structure whose location is fixed, so it can be found easily. On a partitioned file system it may be possible to repartition and work around a bad track, but a permanent error in the first few sectors of either a floppy or hard disk generally means the disk is unusable.

"Intelligent" controllers reserve a few tracks not normally available to user programs. When a disk drive is formatted, the controller determines which blocks are bad and automatically substitutes one of the spare tracks for the bad one. The table that maps bad tracks to spare tracks is kept in the controller's internal memory and on the disk. This

substitution is transparent (invisible) to the driver, except that its carefully worked out elevator algorithm may perform poorly if the controller is secretly using cylinder 800 whenever cylinder 3 is requested. The technology of manufacturing disk recording surfaces is better than it used to be, but it is still not perfect. However, the technology of hiding the imperfections from the user has also improved. On hard disks such as the one described in Fig. 3-19, the controller also manages new errors that may develop with use, permanently assigning substitute blocks when it determines that an error is unrecoverable. With such disks the driver software rarely sees any indication that there any bad blocks.

Seek errors are caused by mechanical problems in the air. The controller keeps track of the arm position internally. To perform a seek, it issues a series of pulses to the arm motor, one pulse per cylinder, to move the arm to the new cylinder. When the arm gets to its destination, the controller reads the actual cylinder number (written when the drive was formatted). If the arm is in the wrong place, a seek error has occurred.

Most hard disk controllers correct seek errors automatically, but many floppy controllers (including the IBM PCs) just set an error bit and leave the rest to the driver. The driver handles this error by issuing a **RECALIBRATE** command, to move the arm as far out as it will go and reset the controller's internal idea of the current cylinder to 0. Usually this solves the problem. If it does not, the drive must be repaired.

As we have seen. the controller is really a specialized little computer, complete with software, variables, buffers, and occasionally, bugs. Sometimes an unusual sequence of events such as an interrupt on one drive occurring simultaneously with a **RECALIBRATE** command for another drive will trigger a bug and cause the controller to go into a loop or lose track of what it was doing. Controller designers usually plan for the worst and provide a pin on the chip which, when asserted, forces the controller to forget whatever it was doing and reset itself. If all else fails, the disk driver can set a bit to invoke this signal and reset the controller. If that does not help, all the driver can do is print a message and give up.

Track-at-a-Time Cashing

The time required to seek to a new cylinder is usually much more than the rotational delay, and always much more than the transfer time. In other words, once the driver has gone to the trouble of moving the arm somewhere, it hardly matters whether it reads one sector or a whole track. This effect is especially true I if the controller provides rotational sensing, so the driver can see which sector is currently under the head and issue a request for the next sector, thereby making it possible to read a track in one rotation time. (Normally it takes half a rotation plus one sector time just to read a single sector, on the average.)

Some disk drivers take advantage of this property by maintaining a secret track-at-a-time cache, unknown to the device-independent software. If a sector that is in the cache is needed, no disk transfer is required. A disadvantage of track-at-a-time caching (in addition to the software complexity and buffer space needed) is that transfers from the cache to the calling program will have to be done by the CPU using a programmed loop, rather than letting the DMA hardware do the job.

Some controllers take this process a step further, and do track-at-a-time caching in their own internal memory, transparent to the driver, so that transfer between the controller and memory can use DMA. If the controller works this way, there is little point in having the disk driver do it as well. Note that both the controller and the driver are in a good position to read and write entire tracks in one command, but that the device-independent

software cannot, because it regards a disk as a linear sequence of blocks, without regard to how they are divided up into tracks and cylinders.

3.7.3 Overview of the Hard Disk Driver in MINIX

The hard disk driver is the first part of MINIX we have looked at that has to deal with a wild range of different types of hardware. Before we discuss the driver, we will briefly consider some of the problems hardware differences can cause. The "IBM-PC" is really a family of different computers. Not only are different processors used in different members of the family, there are also some major differences in the basic hardware. The earliest members of the family, the original PC and PC-XT, used an 8-bit bus, appropriate for the 8-bit external interface of the 8088 processor. The next generation, the PC-AT, used a 16-bit bus, which was cleverly designed so older 8-bit peripherals could still be used. Newer 16-bit peripherals generally cannot be used on older PC-XT systems, however. The AT bus was originally designed for systems using the 80286 processor, and many systems based on 80386, 80486 and Pentium use the AT bus. However, since these newer processors have a 32-bit interface, there are now several different 32-bit bus systems available, such as Intel's PCI bus.

For every bus there is a different family of **I/O adapters**, which plug into the system parentboard. All the peripherals for a particular bus design must be compatible with the standards for that design but need not be compatible with older designs. In the IBM PC family, as in most other computer systems, each bus design also comes with firmware in the Basic I/O System Read-Only Memory (the BIOS ROM) which is designed to bridge the gap between the operating system and the peculiarities of the hardware. Some peripheral devices may even provide extensions to the BIOS in ROM chips on the peripheral cards themselves. The difficulty faced by an operating system implementer is that the BIOS in IBM-type computers (certainly the early ones) was designed for an operating system, MSDOS, that does not support multiprogramming and that runs in 16-bit real mode, the lowest common denominator of the various modes of operation available from the 80x86 family of CPUs.

The implementer of a new operating system for the IBM PC is thus faced with several choices. One is whether to use the driver support for peripherals in the BIOS or to write new drivers from scratch. This was not a hard choice in the design of early versions of MINIX, since the BIOS was in many ways not suitable to the needs of MINIX. Of course, in order to get started, the MINIX boot monitor uses the BIOS to do the initial loading of the system, whether from hard disk or floppy disk—there is no practical alternative to doing it this way. Once we have loaded the system, including our own I/O drivers, we can do much better than the BIOS.

The second choice then must be faced: without the BIOS support how are we going to make our drivers adapt to the varied kinds of hardware on different systems? To make the discussion concrete, consider that there are at least four fundamentally different types of hard disk controllers that we might find on a system which is otherwise suitable for MINIX: the original 8-bit XT-type controller, the 16-bit AT-type controller, and two different controllers for two different types of IBM PS/2 series computers. There are several possible ways to deal with all this:

1. Recompile a unique version of the operating system for each type of hard disk controller we need to accommodate.

2. Compile several different hard disk drivers into the boot image and have the system automatically determine at startup time which one to use.

3. Compile several different hard disk drivers into the boot image and provide a way for the user to determine which one to use.

As we shall see, these are not mutually exclusive.

The first way is really the best way in the long run. For use on a particular installation there is no need to use up disk and memory space with code for alternative drivers that will never be used. However, it is a nightmare for the distributor of the software. Supplying four different startup disks and advising users on how to use them is expensive and difficult. Thus, one of the other alternatives is advisable, at least for the initial installation.

The second method is to have the operating system probe the peripherals, by reading the ROM on each card or writing and reading I/O ports to identify each card. This is feasible on some systems but does not works well on IBM-type systems, because there are too many nonstandard I/O devices available. Probing I/O ports to identify one device may, in some cases, activate another device which seizes control and disables the system. This method complicates the startup code for each device, and yet still does not work very well. Operating systems that do use this method generally have to provide some kind of override, typically a mechanism such as we use with MINIX.

The third method, used in MINIX, is to allow compilation of several drivers, with one of them being the default. The MINIX boot monitor allows various **boot parameters** to be read at startup time. These can be entered by hand, or stored permanently on the disk. At startup time, if a boot parameter of the form

```
hd = xt
```

is found, this forces use of the XT hard disk driver. If no `hd` boot parameter is found, the default driver is used.

There are two other things MINIX does to try to minimize problems with multiple hard disk drivers. One is that there is, after all, a driver that interfaces between MINIX and the ROM BIOS hard disk support. This driver is almost guaranteed to work on any system and can be selected by use of an

```
hd = bios
```

boot parameter. Generally, this should be a last resort, however. MINIX runs in protected mode on systems with an 80286 or better processor, but the BIOS code always runs in real (8086) mode. Switching out of protected mode and back again whenever a routine in the BIOS is called is very slow.

The other strategy MINIX uses in dealing with drivers is to postpone initialization until the last possible moment. Thus, if on some hardware configuration none of the hard disk drivers work, we can still start MINIX from a floppy disk and do some useful work. MINIX will have no problems as long as no attempt is made to access the hard disk. This may not seem like a major breakthrough in user friendliness, but consider this: if all the drivers try to initialize immediately on system startup, the system can be totally paralyzed by improper configuration of some device we do not need anyway. By postponing initialization of each driver until it is needed, the system can continue with whatever does work, while the user tries to resolve the problems.

As an aside, we learned this lesson the hard way: earlier versions of MINIX tried to initialize the hard disk as soon as the system was booted. If no hard disk was present,

the system hung. This behavior was especially unfortunate because MINIX would run quite happily on a system without a hard disk, albeit with restricted storage capacity and reduced performance.

In the discussion in this section and the next, we will take as our model the AT-style hard disk driver, which is the default driver in the standard MINIX distribution. This is a versatile driver that handles hard disk controllers from the ones used in the earliest 80286 systems to modern **EIDE** (**Extended Integrated Drive Electronics**) controllers that handle gigabyte capacity hard disks. The general aspects of hard disk operation we discuss in this section apply to the other supported drivers as well.

The main loop of the hard disk task is the same common code we have already discussed, and the standard six kinds of requests can be made. A *DEV_OPEN* request can entail a substantial amount of work, as there are always partitions and may be subpartitions on a hard disk. These must be read when a device is opened, (i.e., when it is first accessed). Some hard disk controllers can also support CD-ROM drives, which have removable media, and on a *DEV_OPEN* the presence of the medium must be verified. On a CD-ROM a *DEV_CLOSE* operation also has meaning: it requires that the door be unlocked and the CD-ROM ejected. There are other complications of removable media that are more applicable to floppy drives, so we will discuss these in a later section. For a hard disk, the *DEV_IOCTL* operation is used to set a flag to mark that the medium should be ejected from the drive upon a *DEV_CLOSE*. It is also used to read and write partition tables.

The *DEV_READ*, *DEV_WRITE*, and *SCATTERED_IO* requests are each handled in three phases, prepare, schedule, and finish, as we saw previously. The hard, unlike the memory devices, makes a real distinction between the schedule and finish phases. The hard disk driver does not use SSF or the elevator algorithm, but it does do a more limited form of scheduling, gathering requests for consecutive sectors. Requests normally come from the MINIX file system and are for multiples of 1024 bytes, but the driver is able to handle requests for any multiple of a sector (512 bytes). As long as each request is for a starting sector immediately following the last sector requested, each request is appended to a list requests. The list is maintained as an array, and when it is full, or when a nonconsecutive sector is requested, a call is made to the finish routine.

In a simple *DEV_READ* or *DEV_WRITE* request, more than a single block may be requested, but each call to the schedule routine is immediately followed by a call to the finish routine, which ensures the current request list is fulfilled. In the case of a *SCATTERED_IO* request, there may be multiple calls to the schedule routine before the finish routine is called. As long as they are for consecutive blocks of data, the list will be extended until the array becomes full. Recall that in a *SCATTERED_IO* request a flag can signify that request for a particular block is optional. The hard disk driver, like the memory driver, ignores the *OPTIONAL* flag and delivers all the data requested.

The rudimentary scheduling performed by the hard disk driver, postponing actual transfers while consecutive blocks are being requested, should be seen as the second step of a potential three-step process of scheduling. The file system itself, by using scattered I/O, can implement something similar to Teory's version of the elevator algorithm—recall that in a scattered I/O request the list of requests is sorted on the block number. The third step in scheduling takes place in the controller of a modern hard disk, like the one described in Fig. 3-19. Such controllers are "smart" and can buffer large quantities of data, using internally programmed algorithms to retrieve data in the most efficient order, irrespective of the order of receipt of the requests.

3.7.4 Implementation of the Hard Disk Driver in MINIX

Small hard disks used on microcomputers are sometimes called "winchester" disks. There are several different stories about the origin of the name. It was apparently an IBM's code name for the project that developed the disk technology in which the read/write heads fly on a thin cushion of air and land on the recording medium when the disk stops spinning. The explanation of the name is that an early model had two data modules, a 30-Mbyte fixed and a 30-Mbyte removable one. Supposedly this reminded the developers of the Winchester 30-30 firearm which figures in many tales of the United States' western frontier. Whatever the origin of the name, the basic technology remains the same, although today's typical PC disk is much smaller and the capacity is much larger than the 14-inch disks that were typical of the early 1970s when the winchester technology was developed.

The file *wini.c* has the job of hiding the actual hard disk driver used from the rest of the kernel. This allows us to follow the strategy discussed in the previous section, compiling several hard disk drivers into a single kernel image, and selecting the one to use at boot time. Later, a custom installation can be recompiled with only the one driver actually needed.

Wini.c contains one data definition, *hdmap* (line 10013), an array that associates a name with the address of a function. The array is initialized by the compiler with an many elements as are needed for the number of hard disk drivers enabled in *include/minix/config.h*. The array is used by the function *winchester_tast*, which is the name entered in the *task_tab* table used when the kernel is first initialized. When *winchester_task* (line 10040) is called, it tries to find an *hd* environment variable, using a kernel function that works similarly to the mechanism used by ordinary C program, reading the environment created by the MINIX boot monitor. It no *hd* is defined, the first entry in the array is used; otherwise, the array is searched for a matching name. The corresponding function is then called indirectly. In the rest of this section we will discuss the *at_winchester_task*, which is the first entry in the *hdmap* array in the standard distribution of MINIX.

The AT-style driver is in *at_wini.c* (line 10100). This is a complicated driver for a sophisticated device, and there are several pages of macro definitions specifying controller registers, status bits and commands, data structures, and prototypes. As with other block device drivers, a *driver* structure, *w_dtab* (lines 10274 to 10284), is initialized with pointers to the functions that actually do the work. Most of them are defined in *at_wini.c*, but as the hard disk requires no special cleanup operation, its *dr_cleanup* entry points to the common *nop_cleanup* in *driver.c*, shared with other drivers that have no special cleanup requirement. The entry function, *at_winchester_task* (line 10294), calls a procedure that does hardware-specific initialization and then calls the main loop in *driver.c*. This runs forever, dispatching calls to the various functions pointed to by the *driver* table.

Since we are now dealing with real electromechanical storage devices, there is a substantial amount of work to be done to initialize the hard disk driver. Various parameters about the hard disks are kept in the *wini* array defined on lines 10214 to 10230. As part of the policy of postponing initialization steps that could fail until the first time they are truly necessary, *init_params* (line 10307), which is called during kernel initialization, does not do anything that requires accessing the disk device itself. The main thing it does is to copy information about the hard disk logical configuration into the *wini* array. This is the information that is retrieved by the ROM BIOS from the CMOS memory that Pentium-class computers use to preserve basic configuration data. The BIOS actions take place when the computer is first turned on, before the first part of the MINIX loading process

begins. It is not necessarily fatal if this information cannot be retrieved. If the disk is a modern one, the information can be retrieved directly from the disk.

After the call to the common main loop, nothing may happen for a while until the first attempt is made to access the hard disk. Then a message requesting a *DEV_OPEN* operation is received and *w_do_open* (line 10353) is indirectly called. In turn, *w_do_open* calls *w_prepare* to determine if the device requested is valid, and then *w_identify* to identify the type of device and initialize some more parameters in the *wini* array. Finally, a counter in the *wini* array is used to test whether this is first time the device has been opened since MINIX was started. After being examined, the counter is incremented. If it is the first *DEV_OPEN* operation, the *partition* function (in *drvlib.c*) is called.

The next function, *w_prepare* (line 10388), accepts an integer argument, *device*, which is the minor device number of the drive or partition to be used, and returns a pointer to the *device* structure that indicates the base address and size of the device. In C, the use of an identifier to name a structure does not preclude use of the same identifier to name a variable. Whether a device is a drive, a partition, or a subpartition can be determined from the minor device number. Once *w_prepare* has completed its job, none of the other functions used to read or write the disk need to concern themselves with partitioning. As we have seen, *w_prepare* is called when a *DEV_OPEN* request is made; it is also one phase of the prepare/transfer cycle used by all data transfer requests. In that context its initialization of *w_count* to zero is important.

Software-compatible AT-type disks have been in use for quite a while, and *w_identify* (line 10415) has to distinguish between a number of different designs that have been introduced over the years. The first step is to see that a readable and writeable I/O port exists where one should exist on all disk controllers in this family (lines 10435 to l0437). If this condition is met, the address of the hard disk interrupt handler is installed in the interrupt descriptor table and the interrupt controller is enabled to respond to that interrupt. Then an *ATA_IDENTIFY* command is issued to the disk controller. If the result is OK, various pieces of information are retrieved, including a string that identities the model of the disk, and the physical cylinder, head, and sector parameters for the device. (Note that the "physical" configuration reported may not be the true physical configuration, but we have no alternative to accepting what the disk drive claims.) The disk information also indicates whether or not the disk is capable of **Linear Block Addressing (LBA)**. If it is, the driver can ignore the cylinder, head, and sector parameters and can address the disk using absolute sector numbers, which is much simpler.

As we mentioned earlier, it is possible that *init_params* may not recover the logical disk configuration information from the BIOS tables. If that happens, the code at lines 10469 to 10477 tries to create an appropriate set of parameters based on what it reads from the drive itself. The idea is that the maximum cylinder, head, and sector numbers can be 1023, 255, and 63 respectively, due to the number of bits allowed for these fields in the original BIOS data structures.

If the *ATA_IDENTIFY* command fails, it may simply mean that the disk is an older model that does not support the command. In this case the logical configuration values previously read by *init_params* are all we have. If they are valid, they are copied to the physical parameter fields of *wini*; otherwise an error is returned and the disk is not usable.

Finally, MINIX uses a *u32_t* variable to count addresses in bytes. The size of device the driver can handle, expressed as a count of sectors, must be limited if the product of cylinders $\times$ heads $\times$ sectors is too large (line 10490). Although at the time of writing this code devices of 4-GB capacity were rarely found on machines that one might expect to be used for MINIX, experience has taught that software should be written to test for limits

such as this, unnecessary as such tests may appear at the time the code is written. The base and size of the whole drive are then entered into the *wini* array, and *w_specify* is called, twice if necessary, to pass the parameters to be used back to the disk controller. Finally, the name of the device (determined by *w_name*) and the identification string found by *identify* (if it is an advanced device) or the cylinder head and sector parameters reported by the BIOS (if an old device) are printed on the console.

W_name (line 10511) returns a pointer to a string containing the device name, which will be either "at-hd0," "at-hd5," "at-hd10," or "at-hdl5." *W_specify* (line 10531), in addition to passing the parameters to the controller, also recalibrates the drive (if it is an older model), by doing a seek to cylinder zero.

Now we are ready to discuss the functions called in satisfying a data transfer request. *W_prepare*, which we have already discussed, is called first. Its initialization of the variable *w_count* to zero is important here. The next function called during a transfer is *w_schedule* (line 10567). It sets up the basic parameters: where the data are to come from, where they are to go to, the count of bytes to transfer (which must be a multiple of the sector size, and is tested on line 10584), and whether the transfer is a read or write. The bit that may be present in a *SCATTERED_IO* request to indicate an optional transfer is reset in the operation code to be passed to the controller (line 10595), but note that it is retained in the *io_request* field of the *io_request_s* structure. For the hard disk an attempt is made to honor all requests, but, as we will see, the driver may later decide not to do so if there have been errors. The last thing in the setup is to check that the request does not go beyond the last byte on the device and to reduce the request if it does. At this point the first sector to be read can be calculated.

On line 10602 the process of scheduling begins in earnest. If there are already requests pending (tested by seeing if *w_count* is greater than zero), and if the sector to read next is not consecutive with the last one requested, then *w_finish* is called to complete the previous requests. Otherwise, *w_nextblock*, which holds the sector number of the next sector, is updated, and the loop on lines 10611 to 10640 is entered to add new sector requests to the array of requests. Within maximum allowable number of requests has been reached (line 10614). The limit is kept in a variable, *max_count*, since, as we will see later, it is sometimes helpful to be able to adjust the limit. Here again, a call to *w_finish* may result.

As we have seen, there are two places within *w_prepare* where a call to *w_finish* may be made. Normally *w_prepare* terminates without calling *w_finish*, but whether or not it is called from within *w_prepare*, *w_finish* (line 10649) is always called eventually from the main loop in *driver.c*. If it has just been called, it may have no work, so there is a test on line 10659 to check this. If there are still requests in the request array, the main part of *w_finish* is entered.

As one might expect, since there may be a considerable number of requests queued, the main part of *w_finish* is a loop, on lines 10664 to 10761. Before entering the loop, the variable *r* is preset to a value signifying an error, to force reinitialization of the controller. If a call to *w_specify* succeeds the *command* structure, *cmd* is initialized to do a transfer. This structure is used to pass all the required parameters to the function that actually operates the disk controller. The *cmd.precomp* parameter is used by some drives to compensate for differences in the performance of the magnetic recording medium with differences in speed of passage of the medium under the disk heads as they move from outer to inner cylinders. It is always the same for a particular drive and is ignored by many drives. *Cmd.count* receives the number of sectors to transfer, masked to a quantity that fits in an 8-bit byte, since that is the size of all the command and status registers of the

controller. The code on lines 10675 to 10689 specifies the first sector to transfer, either as a 28-bit logical block number (lines 10676 to 10679), or as cylinder, head, and sector parameters (lines 10681 to 10688). In either case the same fields in the *cmd* structure are used.

Finally, the command itself, read or write, is loaded and *com_out* is called at line 10692 to initiate the transfer. The call to *com_out* may fail if the controller is not ready or does not become ready within a preset timeout period. In this case the count of errors is incremented and the attempt is aborted if *MAX_ERRORS* is reached. Otherwise, the

```
continue;
```

statement on line 10697 causes the loop to start over again at line 10665. If the controller accepts the command passed in the call to *com_out*, it may be a while before the data are available, so (assuming the command is *DEV_READ*) on line 10706 *w_intr_wait* is called. We will discuss this function in detail later, but for now just note that it calls *receive*, so at this point the disk task blocks.

Some time later, more or less, depending upon whether a seek was involved, the call to *w_intr_wait* will return. This driver does not use DMA, although some controllers support it. Instead, programmed I/O is used. If there is no error returned from *w_intr_wait*, the assembly language function *port_read* transfers *SECTOR_SIZE* bytes of data from the controller's data port to their final destination, which should be a buffer in the file system's block cache. Next, various addresses and counts are adjusted to account for the successful transfer. Finally, if the count of bytes in the current request goes to zero, the pointer to the array of requests is advanced to point to the next request (line 10714).

In the case of a *DEV_WRITE* command, the first part, setting up the command parameters and sending the command to the controller, is the same as for a read, except for the command operation code. The order of subsequent events is different for a write, however. First there is a wait for the controller to signal it is ready to receive data (line 10724). *Waitfor* is a macro, and normally will return very quickly. We will say more about it later; for now we will just note that the wait will time out eventually, but that long delays are expected to be extremely rare. Then the data are transferred from memory to the controller data port using *port_write* (line 10729), and at this point *w_intr_wait* is called and the disk task blocks. When the interrupt arrives and the disk task is awakened, the bookkeeping is done (lines 10736 to 10739).

Finally, if there have been errors in reading or writing, they must be dealt with. If the controller informs the driver that the error was due to a bad sector, there is no point in trying again, but other types of errors are worth a retry, at least up to a point. That point is determined by counting the errors and giving up if *MAX_ERRORS* is reached. When *MAX_ERRORS*/2 is reached, *w_need_reset* is called to force reinitialization when the retry is made. However, if the request was originally an optional one (made by a *SCATTERED_IO* request), no retry is attempted.

Whether *w_finish* terminates without errors or because of an error, the variable *w_command* is always set to *CMD_IDLE*. This allows other functions to determine that the failure was not because of a mechanical or electrical malfunction of the disk itself causing failure to generate an interrupt following an attempted operation.

The disk controller is controlled through a set of registers, which could be memory mapped on some systems, but on an IBM compatible appear as I/O ports. The registers used by a standard IBM-AT class hard disk controller are shown in Fig. 3-22.

This is our first encounter with I/O hardware, and it may be helpful to mention some ways I/O ports may behave differently from memory addresses. In general, input and out-

Register	Read Function	Write Function
0	Data	Data
1	Error	Write Precompensation
2	Sector Count	Sector Count
3	Sector Number (0-7)	Sector Number (0-7)
4	Cylinder Low (8-15)	Cylinder Low (8-15)
5	Cylinder High (16-23)	Cylinder High (16-23)
6	Select Drive/Head (24-27)	Select Drive/Head (24-27)
7	Status	Command

(a)

7	6	5	4	3	2	1	0
1	LBA	1	D	HS3	HS2	HS1	HS0

LBA: 0 = Cylinder/Head/Sector Mode
1 = Logical Block Addressing Mode
D: 0 = master drive
1 = slave drive
HSn: CHS mode: Head Select in CHS mode
LBA mode: Block select bits 24 - 27

(b)

Figure 3-22. (a) The control registers of an IDE hard disk controller. The numbers in parentheses are the bits of the logical block address selected by each register in LBA mode. (b) The fields of the Select Drive/Head register.

put registers that happen to have the same I/0 port address are not the same register. Thus, the data written to a particular address cannot necessarily be retrieved by a subsequent read operation. For example, the last register address shown in Fig. 3-22 shows the status of the disk controller when read and is used to issue commands to the controller when written to. It is also common that the very act of reading or writing an I/O device register causes an action to occur, independently of the details of the data transferred. This is true of the command register on the AT disk controller. In use, data are written to the lower-numbered registers to select the disk address to be read from or written to, and then the command register is written last with an operation code. The act of writing the operation code into the command register starts the operation.

It is also the case that the use of some registers or fields in the registers may vary with different modes of operation. In the example given in the figure, writing a 0 or a 1 to the LBA bit, bit 6 of register 6, selects whether CHS (Cylinder-Head-Sector) or LBA (Linear Block Addressing) made is used. The data written to or read from registers 3, 4, and 5, and the low four bits of register 6 are interpreted differently according to the setting of the LBA bit.

Now let us look at how a command is sent to the controller by calling *com_out* (line 10771). Before changing any registers, the status register is read to determine that the controller is not busy. This is done by testing the *STATUS_BSY* bit. Speed is important here, and normally the disk controller is ready or will be ready in a short time, so busy waiting is used. On line 10779 *waitfor* is called to test *STATUS_BSY*. To maximize the speed of response, *wairfor* is a macro, defined on line 10268. It makes the required test once, avoiding an expensive function call on most calls, when the disk is ready. On the rare occasions when a wait is necessary, it then calls *w_waitffor*, which executes the test in a loop until it is true or a predefined timeout period elapses. Thus the returned value will be true with the minimum possible delay if the controller is ready, true after a delay if it is temporarily unavailable, or false if it is not ready after the timeout period. We will have more to say about the timeout when we discuss *w_waitfor* itself.

A controller can handle more than one drive, so once it is determined that the controller is ready, a byte is written to select the drive, head, and mode of operation (line l0785) and then *wairfor* is called again. A disk drive sometimes fails to carry out a command or to properly return an error code—it is, after all, a mechanical device that can stick, jam, or break internally—and as insurance a message is sent to the clock task to schedule a call to a wakeup routine. Following this, the command is issued by first writing all the parameters to the various registers and finally writing the command code itself to the command register. The latter step and the subsequent modification of the *w_command* and *w_status* variables is a critical section, so the entire sequence is bracketed by calls to *lock* and *unlock* (lines 10801 to 10805) which disable and then reenable interrupts.

The next several functions are short. We noted that *w_need_reset* (line 10813) is called by *w_finish* when the failure count hits half of *MAX_ERRORS*. It is also called when timeouts occur while waiting for the disk to interrupt or become ready. The action of *w_need_reset* is just to mark the *state* variable for every drive in the *wini* array to force initialization on the next access.

W_do_close (line 10828) has very little to do for a conventional hard disk. When support is added for CD-ROMs or other removable devices, this routine will have to be extended to generate a command to unlock the door or eject the CD, depending upon what the hardware supports.

Com_simple is called to issue controller commands that terminate immediately without a data transfer phase. Commands that fall into this category include those that retrieve the disk identification, setting of some parameters, and recalibration.

When *com_out* calls the clock task to prepare for a possible rescue after a disk controller failure, it passes the address of *w_timeout* (line 10858) as the function for the clock task to awaken when the timeout period expires. Usually the disk completes the requested operation and when the timeout occurs, *w_command* will be found to have the value CMD_IDLE, meaning the disk completed its operation, and *w_timeout* can then terminate. If the command does not complete and the operation is a read or write, it may help to reduce the size of I/O requests. This is done in two steps, first reducing the maximum number of sectors that can be requested to 8, and then to 1. For all timeouts a message is printed, *w_need_reset* is called to force re-initialization of all drives on the next attempted access, and interrupt is called to deliver a message to the disk task and simulate the hardware-generated interrupt that should have occurred at the end of the disk operation.

When a reset is required, *w_reset* (line 10889) is called. This function makes use of a function provided by the clock driver, *milli_delay*. After an initial delay to give the drive time to recover from previous operations, a bit in the disk controller's control register is

strobed—that is, brought to a logical 1 level for a definite period, then returned to the logical 0 level. Following this operation, *waitfor* is called to give the drive a reasonable period to signal it is ready. In case the reset does not succeed, a message is printed and an error status returned. It is left to the caller to decide what to do next.

Commands to the disk that involve data transfer normally terminate by generating an interrupt, which sends a message back to the disk task. In fact, an interrupt is generated for each sector read or written. Thus, after issuing such a command, *w_intr_wait* (line 10925) will always be called. In turn, *w_intr_wait* calls *receive* in a loop, ignoring the contents of each message, waiting for an interrupt that sets *w_status* to "not busy." Once such a message is received, the status of the request is checked. This is another critical section, so *lock* and *unlock* are used here to guarantee that a new interrupt will not occur and change *w_stutus* before the various steps involved are complete.

We have seen several places where the macro *waitfor* is called to do busy waiting on a bit in the disk controller status register. After the initial test, the *waitfor* macro calls *w_waitfor* (line 10955), which calls *milli_start* to begin a timer and then enters a loop that alternately checks the status register and the timer. If a timeout occurs, *w_need_reset* is called to set things up for a reset of the disk controller the next time its services are requested.

The *TIMEOUT* parameter used by *w_waitfor* is defined on line 10206 as 32 seconds. A similar parameter, *WAKEUP* (line 10193), used to schedule wakeups from the clock task, is set to 31 seconds. These are very long periods of time to spend busy waiting, when you consider that an ordinary process only gets 100 msec to run before it will be evicted. But, these numbers are based upon the published standard for interfacing disk devices to AT-class computers, which states that up to 31 seconds must be allowed for a disk to "spin up" to speed. The fact is, of course, that this is a worst-case specification, and that on most systems spin up will only occur at power-on time, or possibly after long periods of inactivity. MINIX is still being developed. It is possible that a new way of handling timeouts may be called for when support for CD-ROMs (or other devices which must spin up frequently) is added.

W_handler (line 10976) is the interrupt handler. The address of this function is put into the Interrupt Descriptor Table by *w_identify* when the hard disk task is first activated. When a disk interrupt occurs, the disk controller status register is copied to *w_status* and then the *interrupt* function in the kernel is called to reschedule the hard disk task. When this occurs, of course, the hard disk task is already blocked as a result of a previous call to *receive* from *w_intr_wait* after initiation of a disk operation.

The last function in *at_wini.c* is *w_geometry*. It returns the logical maximum cylinder, head, and sector values of the selected hard disk device. In this case the numbers are real ones, not made up as they were for the RAM disk driver.

3.7.5 Floppy Disk Handling

The floppy disk driver is longer and more complicated than the hard disk driver. This may seem paradoxical, since floppy disk mechanisms would appear to be simpler than those of hard disks, but the simpler mechanism has a simpler controller that requires more attention from the operating system, and the fact that the medium is removable adds some complications. In this section we will describe some of the things an implementor has to consider in dealing with floppy disks. However, we will not go into the details of the MINIX floppy disk driver code. The most important parts are similar to those for the hard disk.

One of the things we do not have to worry about with the floppy driver is the multiple types of controller to support that we had to deal with in the case of the hard disk driver. Although the high-density floppy disks currently used were not supported in the design of the original IBM PC, the floppy disk controllers of all computers in the IBM PC family are supported by a single software driver. The contrast with the hard disk situation is probably due to lack of pressure to increase floppy disk performance. Floppy disks are rarely used as working storage during operation of a computer system; their speed and data capacity are too limited compared to those of hard disks. Floppy disks remain important for distribution of new software and for backup, so almost all small computer systems are equipped with at least one floppy drive.

The floppy disk driver does not use the SSF or the elevator algorithm. It is strictly sequential, accepting a request and carrying it out before even accepting another request. In the original design of MINIX it was felt that, since MINIX was intended for use on personal computer, most of the time there would be only one process active, and the chance of a disk request arriving while another was being carried out was small. Thus there would be little to gain from the considerable increase in software complexity that would be required for queueing requests. It is even less worthwhile now, since floppy disks are rarely used for anything but transferring data into or out of a system with a hard disk.

That said, even though there is no support in the driver software for reordering requests, the floppy driver, like any other block driver, can handle a request for scattered I/O, and just like the hard disk driver, the floppy driver collects requests in an array and continues to collect such requests as long as sequential sectors are requested. However, in the case of the floppy driver the array of requests is smaller than for the hard disk, limited to the maximum number of sectors per track on a floppy diskette. In addition, the floppy driver pays attention to the *OPTIONAL* flag in scattered I/O requests and does not proceed to a new track if all current requests are optional.

The simplicity of the floppy disk hardware is responsible for some of the complications in floppy disk driver software. Cheap, slow, low-capacity floppy drives do not justify the sophisticated integrated controllers that are part of modem hard drives, so the driver software has to deal explicitly with aspects of disk operation that are hidden in the operation of a hard drive. As an example of a complication caused by the simplicity of floppy drives, consider positioning the read/write head to a particular track during a *SEEK* operation. No hard disk has ever required the driver software to explicitly call for a *SEEK*. For a hard disk the cylinder, head, and sector geometry visible to the programmer may not correspond to the physical geometry, and, in fact, the physical geometry may be quite complicated, with more sectors on outer cylinders than on inner ones. This is not visible to the user, however. Hard disks may accept Logical Block Addressing (LBA), addressing by the absolute sector number on the disk, as an alternative to cylinder, head, and sector addressing. Even if addressing is done by cylinder, head, and sector, any geometry that does not address nonexistent sectors may be used, since the integrated controller on the disk calculates where to move the read/write heads and does a seek operation when required.

For a floppy disk, however, explicit programming of *SEEK* operations is needed. In case a *SEEK* fails, it is necessary to provide a routine to perform a *RECALIBRATE* operation, which forces the heads to cylinder 0. This makes it possible for the controller to advance them to a desired track position by stepping the heads a known number of times. Similar operations are necessary for the hard drive, of course, but the drive controller handles them without detailed guidance from the device driver software.

Some characteristics of a floppy disk drive that complicate its driver are:

1. Removable media.

2. Multiple disk formats.

3. Motor control.

Some hard disk controllers provide for removable media, for instance, on a CD-ROM drive, but the drive controller is generally able to handle any complications without much support in the device driver software. With a floppy disk, however, the built-in support is not there, and yet it is needed more. Some of the most common uses for floppy disks—installing new software or backing up files—are likely to require switching of disks in and out of the drives. It can cause grief if data that were intended for one diskette are written onto another diskette. The device driver should do what it can to prevent this, although this is not always possible, as not all floppy drive hardware allows determination of whether the drive door has been opened since the last access. Another problem that can be caused by removable media is that a system can become hung up if an attempt is made to access a floppy drive that currently has no diskette inserted. This can be solved if an open door can be detected, but since this is not always possible some provision must be made for a timeout and an error return if an operation on a floppy disk does not terminate in a reasonable time.

Removable media can be replaced with other media, and in the case of floppy disks there are many different possible formats. MINIX hardware supports both 3.5-inch and 5.25-inch disk drives and the diskettes can be formatted in a variety of ways to hold from 360 KB up to 1.2 MB (on a 5.25-inch diskette) or 1.44 MB (on a 3.5-inch diskette). MINIX supports seven different floppy disk formats.There are two possible solutions to the problem this causes, and MINIX allows for both of them. One way is to refer to each possible format as a distinct drive and provide multiple minor devices. MINIX does this, and in the device directory you will find fourteen different devices defined, ranging from */dev/pc0*, a 360K 5.25-inch diskette in the first drive, to */dev/PS1*, a 1.44M 3.5-inch diskette in the second drive. Remembering the different combinations is cumbersome, and a second alternative is provided. When the first floppy disk drive is addressed as */dev/fd0*, or the second as */dev/fd1*, the floppy disk driver tests the diskette currently in the drive when it is accessed, in order to determine the format. Some formats have more cylinders, and others have more sectors per track than other formats. Determination of the format of a diskette is done by attempting to read the higher numbered sectors and tracks. By a process of elimination the format can be determined. This does, of course, take time, and a disk with bad sectors could be misidentified.

The final complication of the floppy disk driver is motor control. Diskettes cannot be read or written unless they are revolving. Hard disks are designed to run for thousands of hours on end without wearing out, but leaving the motors on all the time causes a floppy drive and diskette to wear out quickly. If the motor is not already on when a drive is accessed, it is necessary to issue a command to start the drive and then to wait about a half second before attempting to read or write data. Turning the motors on or off is slow, so MINIX leaves a drive motor on for a few seconds after a drive is used. If the drive is used again within this interval, the timer is extended for another few seconds. If the drive is not used in this interval, the motor is turned off.

3.8 Clocks

Clocks (also called **timers**) are essential to the operation of any timesharing system for a variety of reasons. They maintain the time of day and prevent one process from monopolizing the CPU, among other things. The clock software has the form of a device driver, even though a clock is neither a block device, like a disk, nor a character device, like a terminal. Our examination of clocks will follow the same pattern as in the previous sections: first a look at clock hardware and software in general, and then a closer look at how these ideas are applied in MINIX.

3.8.1 Clock Hardware

Two types of clocks are commonly used in computers, and both are quite different from the clocks and watches used by people. The simpler clocks are tied to the 110- or 220-volt power line, and cause an interrupt on every voltage cycle, at 50 or 60 Hz.

The other kind of clock is built out of three components: a crystal oscillator, a counter, and a holding register, as shown in Fig. 3-23. When a piece of quartz crystal is properly cut and mounted under tension, it can be made to generate a periodic signal of very high accuracy, typically in the range of 5 to 100 MHz, depending on the crystal chosen. At least one such circuit is usually found in any computer, providing a synchronizing signal to the computer's various circuits. This signal is fed into the counter to make it count down to zero. When the counter gets to zero, it causes a CPU interrupt.

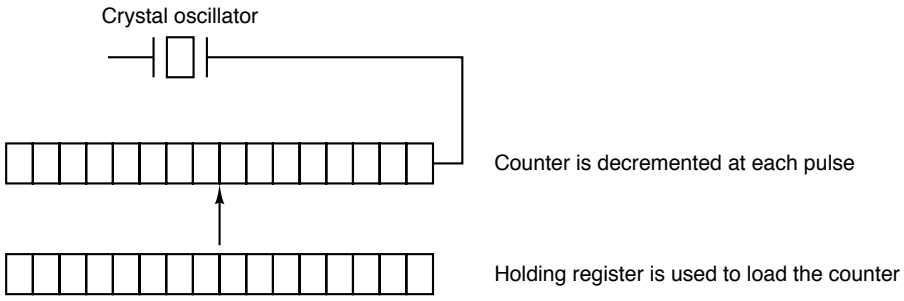


Figure 3-23. A programmable clock.

Programmable clocks typically have several modes of operation. In one-shot mode, when the clock is started, it copies the value of the holding register into the counter and then decrements the counter at each pulse from the crystal. When the counter gets to zero, it causes an interrupt and stops until it is explicitly started again by the software. In square-wave mode, after getting to zero and causing the interrupt, the holding register is automatically copied into the counter, and the whole process is repeated again indefinitely. These periodic interrupts are called clock ticks.

The advantage of the programmable clock is that its interrupt frequency can be controlled by software. If a 1-MHz crystal is used, then the counter is pulsed every microsecond. With 16-bit registers, interrupts can be programmed to occur at intervals from 1 microsecond to 65,536 milliseconds. Programmable clock chips usually contain two or three independently programmable clocks and have many other options as well (e.g., counting up instead of down, interrupts disabled, and more).

To prevent the current time from being lost when the computer's power is turned off, most computers have a battery-powered backup clock, implemented with the kind of low-power circuitry used in digital watches. The battery clock can be read at startup. If the

backup clock is not present, the software may ask the user for the current date and time. There is also a standard protocol for a networked system to get the current time from a remote host. In any case the time is then translated into the number of clock ticks since 12 A.M. **Universal Coordinated Time (UTC)** (formerly known as Greenwich Mean Time) on Jan. 1, 1970, as UNIX and MINIX do, or since some other benchmark. At every clock tick, the real time is incremented by one count. Usually utility programs are provided to manually set the system clock and the backup clock and to synchronize the two clocks.

3.8.2 Clock Software

All the clock hardware does is generate interrupts at known intervals. Everything else involving time must be done by the software, the clock driver. The exact duties of the clock driver vary among operating systems, but usually include most of the following:

1. Maintaining the time of day.
2. Preventing processes from running longer than they are allowed to.
3. Accounting for CPU usage.
4. Handling the ALARM system call made by user processes.
5. Providing watchdog timers for parts of the system itself.
6. Doing profiling, monitoring, and statistics gathering.

The first clock function, maintaining the time of day (also called the real time) is not difficult. It just requires incrementing a counter at each clock tick, as mentioned before. The only thing to watch out for is the number of bits in the time-of-day counter. With a clock rate of 60 Hz, a 32-bit counter will overflow in just over 2 years. Clearly the system cannot store the real time as the number of ticks since Jan. 1, 1970 in 32 bits.

Three approaches can be taken to solve this problem. The first way is to use a 64-bit counter, although doing so makes maintaining the counter more expensive since it has to be done many times a second. The second way is to maintain the time of day in seconds, rather than in ticks, using a subsidiary counter to count ticks until a whole second has been accumulated. Because 2^{32} seconds is more than 136 years, this method will work until well into the twenty-second century.

The third approach is to count in ticks, but to do that relative to the time the system was booted, rather than relative to a fixed external moment. When the backup clock is read or the user types in the real time, the system boot time is calculated from the current time-of-day value and stored in memory in any convenient form. Later, when the time of day is requested, the stored time of day is added to the counter to get the current time of day. All three approaches are shown in Fig. 3-24.

The second clock function is preventing processes from running too long. Whenever a process is started, the scheduler should initialize a counter to the value of that process' quantum in clock ticks. At every clock interrupt, the clock driver decrements the quantum counter by 1. When it gets to zero, the clock driver calls the scheduler to set up another process.

The third clock function is doing CPU accounting. The most accurate way to do it is to start a second timer, distinct from the main system timer, whenever a process is started. When that process is stopped, the timer can be read out to tell how long the process has

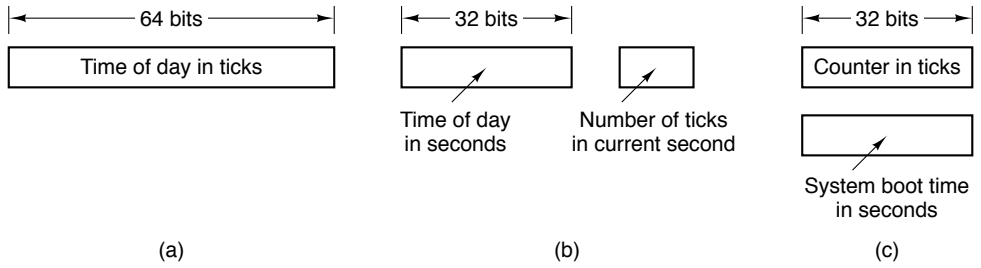


Figure 3-24. Three ways to maintain the time of day.

run. To do things right, the second timer should be saved when an interrupt occurs and restored afterward.

A less accurate, but much simpler, way to do accounting is to maintain a pointer to the process table entry for the currently running process in a global variable. At every clock tick, a field in the current process' entry is incremented. In this way, every clock tick is "charged" to the process running at the time of the tick. A minor problem with this strategy is that if many interrupts occur during a process' run, it is still charged for a full tick, even though it did not get much work done. Properly accounting for the CPU during interrupts is too expensive and is never done.

In MINIX and many other systems, a process can request that the operating system give it a warning after a certain interval. The warning is usually a signal, interrupt, message, or something similar. One application requiring such warnings is networking, in which a packet not acknowledged within a certain time interval must be retransmitted. Another application is computer-aided instruction, where a student not providing a response within a certain time is told the answer.

If the clock driver had enough clocks, it could set a separate clock for each request. This not being the case, it must simulate multiple virtual clocks with a single physical clock. One way is to maintain a table in which the signal time for all pending timers is kept, as well as a variable giving the time of the next one. Whenever the time of day is updated, the driver checks to see if the closest signal has occurred. If it has, the table is searched for the next one to occur.

If many signals are expected, it is more efficient to simulate multiple clocks by chaining all the pending clock requests together, sorted on time, in a linked list, as shown in Fig. 3-25. Each entry on the list tells how many clock ticks following the previous one to wait before causing a signal. In this example, signals are pending for 4203, 4207, 4213, 4215. and 4216.

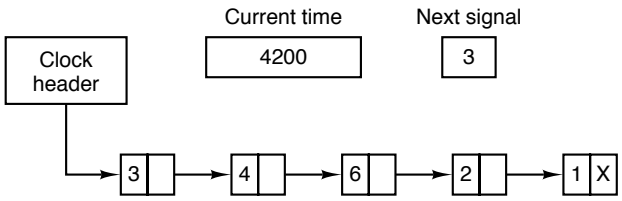


Figure 3-25. Simulating multiple timers with a single clock.

In Fig. 3-25, the next interrupt occurs in 3 ticks. On each tick, *Next signal* is decremented. When it gets to 0; the signal corresponding to the first item on the list is caused, and that item is removed from the list. Then *Next signal* is set to the value in the entry now at the head of the list, in this example, 4.

Note that during a clock interrupt, the clock driver has several things to do— increment the real time, decrement the quantum and check for 0, do CPU accounting, and decrement the alarm counter. However, each of these operations has been carefully arranged to be very fast because they have to be repeated many times a second.

Parts of the operating system also need to set timers. These are called **watchdog timers**. When studying the hard disk driver, we saw that a wakeup call is scheduled each time the disk controller is sent a command, so an attempt at recovery can be made if the command fails completely. We also mentioned that floppy disk drivers have to wait for the disk motor to get up to speed and must shut down the motor if no activity occurs for a while. Some printers with a movable print head can print at 120 characters/sec (8.3 msec/character) but cannot return the print head to the left margin in 8.3 msec, so the terminal driver must delay after typing a carriage return.

The mechanism used by the clock driver to handle watchdog timers is the same as for user signals. The only difference is that when a timer goes off, instead of causing a signal, the clock driver calls a procedure supplied by the caller. The procedure is part of the caller's code, but since all the drivers are in the same address space, the clock driver can call it anyway. The called procedure can do whatever is necessary, even causing an interrupt, although within the kernel interrupts are often inconvenient and signals do not exist. That is why the watchdog mechanism is provided.

The last thing in our list is profiling. Some operating systems provide a mechanism by which a user program can have the system build up a histogram of its program counter, so it can see where it is spending its time. When profiling is a possibility, at every tick the driver checks to see if the current process is being profiled, and if so, computes the bin number (a range of addresses) corresponding to the current program counter. It then increments that bin by one. This mechanism can also be used to profile the system itself.

3.8.3 Overview of the Clock Driver in MINIX

The MINIX clock driver is contained in the file clock.c. The clock task accepts these six message types, with the parameters shown:

1. HARD_INT
2. GET_UPTIME
3. GET_TIME
4. SET_TIME (new time in seconds)
5. SET_ALARM (process number, procedure to call, delay)
6. SET_SYN_AL (process number, delay)

HARD_INT is the message sent to the driver when a clock interrupt occurs and there is work to do, such as when an alarm must be sent or a process has run too long.

GET_UPTIME is used to get the time in ticks since boot time, *GET_TIME* returns the current real time as the number of seconds elapsed since Jan. 1, 1970 at 12:00 A.M., and *SET_TIME* sets the real time. It can only be invoked by the super-user.

Internal to the clock driver, the time is kept track of using the method of Fig. 3-24(c). When the time is set, the driver computes when the system was booted. It can make this computation because it has the current real time and it also knows how many ticks the

system has been running. The system stores the real time of the boot in a variable. Later, when *GET_TIME* is called, it converts the current value of the tick counter to seconds and adds it to the stored boot time.

SET_ALARM allows a process to set a timer that goes off in a specified number of clock ticks. When a user process does an ALARM call, it sends a message to the memory manager, which then sends this message to the clock driver. When the alarm goes off, the clock driver sends a message back to the memory manager, which then takes care of making the signal happen.

SET_ALARM is also used by tasks that need to start a watchdog timer. When the timer goes off, the procedure provided is simply called. The clock driver has no knowledge of what the procedure does.

SET_SYN_AL is similar to *SET_ALARM*, but is used to set a **synchronous alarm**. A synchronous alarm sends a message to a process, rather than generating a signal or calling a procedure. The synchronous alarm task handles dispatching messages to the processes that require them. Synchronous alarms will be discussed in detail later.

The clock task uses no major data structures, but there are several variables used to keep track of time. Only one is a global variable, *lost_ticks*, defined in *glo.h* (line 5031). This variable is provided for the use of any driver that may be added to MINIX in the future that might disable interrupts long enough that one or more clock ticks could be lost. It currently is not used, but if such a driver were to be written the programmer could cause *lost_ticks* to be incremented to compensate for the time during which clock interrupts were inhibited.

Obviously, clock interrupts occur very frequently, and fast handling of the clock interrupt is important. MINIX achieves this by doing the bare minimum amount of processing on most clock interrupts. Upon receipt of an interrupt the handler sets a local variable, *ticks*, to *lost_ticks + 1* and then uses this quantity to update accounting times and *pending_ticks* (line 11079) and resets *lost_ticks* to zero. *Pending_ticks* is a *PRIVATE* variable, declared outside of all function definitions, but known only to functions defined in *clock.c*. Another *PRIVATE* variable, *sched_ticks*, is decremented on each tick to keep track of execution time. The interrupt handler sends a message to the clock task only if an alarm is due or an execution quantum has been used. This scheme results in the interrupt handler returning almost immediately on most interrupts.

When the clock task receives any message, it adds *pending_ticks* to the variable *realtime* (line 11067) and then zeroes *pending_ticks*. *Realtime*, together with the variable *boot_time* (line 11068), allows the current time of day to be computed. These are both *PRIVATE* variables, so the only way for any other part of the system to get the time is by sending a message to the clock task. Although at any instant *realtime* may be inaccurate, this mechanism ensures it is always accurate when needed. If your watch is correct when you look at it, does it matter if it is incorrect when you are not looking?

To handle alarms, *next_alarm* records the time when the next signal or watchdog call may happen. The driver has to be careful here, because the process requesting the signal may exit or be killed before the signal happens. When it is time for the signal, a check is made to see if it is still needed. If it is not needed, it is not carried out.

Each user process is allowed to have only one outstanding alarm timer. Executing an ALARM call while the timer is still running cancels the first timer. Therefore, a convenient way to store the timers is to reserve one word in the process table entry for each process for its timer, if any. For tasks, the function to be called must also be stored somewhere, so an array, *watch_dog*, has been provided for this purpose. A similar array, *syn_table*, stores flags to indicate for each process if it is due to receive a synchronous

alarm.

The overall logic of the clock driver follows the same pattern as the disk drivers. The main program is an endless loop that gets messages, dispatches on the message type, and then sends a reply (except for *CLOCK_TICK*). Each message type is handled by a separate procedure, following our standard naming convention of naming all the procedures called from the main loop *do_xxx*, where *xxx* is different for each one, of course. As an aside, unfortunately, many linkers truncate procedure names to seven or eight characters, so the names *do_set_time* and *do_set_alarm* are potentially in conflict. The latter has been renamed *do_setalarm*. This problem occurs throughout MINIX and is usually solved by mangling one of the names.

The Synchronous Alarm Task

There is a second task to be discussed in this section, the **synchronous alarm task**. A synchronous alarm is similar to an alarm, but instead of sending a signal or calling a watchdog function when the timeout period expires, the synchronous alarm task sends a message. A signal may arrive or a watchdog task may be called without any relation to what part of a task is executing, so alarms of these types are **asynchronous**. In contrast, a message is received only when the receiver has executed a *receive* call.

The synchronous alarm mechanism was added to MINIX to support the network server, which, like the memory manager and the file server, runs as a separate process. Frequently there is a need to set a limit on the time a process may be blocked while waiting for input. For instance, in a network, failure to receive an acknowledgement of a data packet within a definite period is probably due to a failure of transmission. A network server can set a synchronous alarm before it tries to receive a message and blocks. Since the synchronous alarm is delivered as a message, it will unblock the server eventually if no message is received from the network. Upon receiving any message the server must first reset the alarm. Then by examining the type or origin of the message, it can determine a packet has arrived or if it has been unblocked by a timeout. If it is the latter, then the server can try to recover, usually by resending the last unacknowledged packet.

A synchronous alarm is faster than an alarm sent using a signal, which requires several messages and a considerable amount of processing. A watchdog function is fast, but is only useful for tasks compiled into the same address space as the clock task. When a process is waiting for a message, a synchronous alarm is more appropriate and simpler than either signals or watchdog functions, and it is easily handled with little additional processing.

The Clock Interrupt Handler

As described earlier, when a clock interrupt occurs, *realtime* is not updated immediately. The interrupt service routine maintains the *pending_ticks* counter and does simple jobs like charging the current tick to a process and decrementing the quantum timer. A message is sent to the clock task only when more complicated jobs must be done. This is something of a compromise with the ideal of MINIX tasks that communicate totally by messages, but it is a practical concession to the reality that servicing clock ticks consumes CPU time. On a slow machine it was found that doing it this way resulted in a 15% increase in system speed relative to an implementation that sent a message to the clock task on every clock interrupt.

Millisecond Timing

As another concession to reality, a few routines are provided in *clock.c* that provide millisecond resolution timing. Delays as short as a millisecond are needed by various I/O devices. There is no practical way to do this using alarms and the message passing interface. The functions here are meant to be called directly by tasks. The technique used is the oldest and simplest I/O technique: polling. The counter that is used for generating the clock interrupts is read directly, as rapidly as possible, and the count is converted to milliseconds. The caller does this repeatedly until the desired time has elapsed.

Summary of Clock Services

Figure 3-26 summarizes the various services provided by *clock.c*. There are several ways to access the clock, and several ways the request can be honored. Some services are available to any process, with results returned in a message.

Service	Access	Response	Clients
Gettime	System call	Message	Any process
Uptime	System call	Message	Any process
Uptime	Function call	Function value	Kernel or task
Alarm	System call	Signal	Any process
Alarm	System call	Watchdog activation	Task
Synchronous alarm	System call	Message	Server process
Milli_delay	Function call	Busy wait	Kernel or task
Milli_elapsed	Function call	Function value	Kernel or task

Figure 3-26. The clock code supports a number of time-related services.

Uptime can be obtained by a function call from the kernel or a task, avoiding the overhead of a message. An alarm can be requested by a user process, with the eventual result being a signal, or by a task, causing activation of a watchdog function. Neither of these mechanisms can be used by a server process, but a server can ask for a synchronous alarm. A task or the kernel can request a delay using the *milli_delay* function, or it can incorporate calls to *milli_elapsed* into a polling routine, for instance, while waiting for input from a port.

3.8.4 Implementation of the Clock Driver in MINIX

When MINIX starts up, all the drivers are called. Most of them just try to get a message and block. The clock driver, *clock_task* (line 11098), does that too, but first it calls *init_clock* to initialize the programmable clock frequency to 60 Hz. When any message is received, it adds *pending_ticks* to *realtime* and then resets *pending_ticks* before doing anything else. This operation could potentially conflict with a clock interrupt, so calls to *lock* and *unlock* are used to prevent a race (lines 11115 to 11118). Otherwise the main loop of the clock driver is essentially the same as the other drivers; a message is received, a function to do the required work is called, and a reply message is sent.

Do_clocktick (line 11140) is not called on each tick of the clock, so its name is not an exact description of its function. It is called when the interrupt handler has determined there might be something important to do. First a check is made to see if a signal or watchdog timer has gone off. If one has, all the alarm entries in the process table are inspected. Because ticks are not processed individually, several alarms may go off in one pass over the table. It is also possible that the process that was to receive the next alarm has already exited. When a process is found whose alarm is less than the current time, but not zero, the slot in the watchdog array corresponding to that process is checked. In the C programming language a numeric value also has a logical value, so the test on line 11163 returns *TRUE* if a valid address is stored in the *watch_dog* slot, and the corresponding function is called indirectly on line 11163. If a **null pointer** is found (represented in C by a value of zero), the test evaluates to *FALSE* and *cause_sig* is called to send a *SIGALRM* signal. The *watch_dog* slot is also used when a synchronous alarm is needed. In that case the address stored is the address of *cause_alarm*, rather than the address of a watchdog function belonging to a particular task. For sending a signal we could have stored the address of *cause_sig* but then we would have had to have written *cause_sig* differently, to expect no arguments and get the target process number from a global variable. Alternatively, we could have required all watchdog processes to expect an argument they do not need.

We will discuss *cause_sig* when we discuss the system task in a subsequent section. Its job is to send a message to the memory manager. This requires a check to see if the memory manager is currently waiting for a message. If so, it sends a message telling about the alarm. If the memory manager is busy, a note is made to inform it at the first opportunity.

While looping through the process table inspecting the *p_alarm* value for each process, *next_alarm* is updated. Before starting the loop it is set to a very large number (line 11151), and then, for each process whose alarm value is nonzero after sending alarms or signals, a comparison is made between the process' alarm and *next_alarm*, which is set to the smaller value (lines 11171 and 11172).

After processing alarms, *do_clocktick* goes on to see if it is time to schedule another process. The execution quantum is maintained in the *PRIVATE* variable *sched_ticks*, which is normally decremented by the clock interrupt handler on every clock tick. However, on those ticks when *do_clocktick* is activated, it is not decremented by the handler, allowing *do_clocktick* itself to do this and test for a zero result on line 11178. *Sched_ricks* is not reset whenever a new process is scheduled (because the file system and memory manager are allowed to run to completion). Instead it is reset after every *SCHED_RATE* ticks. The comparison on line 11179 is to make sure that the current process has actually run at least one full scheduler tick before taking the CPU away from it.

The next procedure, *do_get_uptime* (line 11189), is just one line; it puts the current value of *realtime* (the number of ticks since boot) into the proper field in the message to be returned. Any process can find the elapsed time this way, but the message overhead is a big price to ask of tasks, so a related function, *get_uptime* (line 11200) is provided that can be called directly by tasks. Since it is not invoked via a message to the clock task, it has to add pending ticks to the current *realtime* itself. *Lock* and *unlock* are necessary here to prevent a clock interrupt occurring while *pending_ticks* is being accessed.

To get the current real time, *do_get_time* (line 11219) computes the current real time from *realtime* and *boot_time* (the system boot time in seconds). *Do_set_time* (line 11230) is its complement. It computes a new value for *boot_time* based on the given current real time and number of ticks since booting.

The procedures *do_setalarm* (line 11242) and *do_setsyn_alrm* (line 11269) are so similar we will discuss them together. Both extract the parameters that specify the process to be signaled and the time to wait from the message. *Do_setalarm* also extracts a function to call (line 11257), although a few lines farther on it replaces this value with a null pointer if the target process is a user process and not a task. We have already seen how this pointer is later tested in *do_clocktick* to determine whether the target should get a signal or a call to a watchdog. The time remaining to the alarm (in seconds) is also calculated by both functions and set into the return message. Both then call *common_setalarm* to finish up. In the case of the do_setsyn_alarm call, the function parameter passed to *common_setalarm* is always *cause_alarm*.

Common_setalarm (line 11291) finishes the work started by either of the two functions just discussed. Then it stores the alarm time in the process table and the pointer to the watchdog procedure (which may also be a pointer to *cause_alarm* or a null pointer) in the *watch_dog* array. Then it scans the entire process table to find the next alarm, just as is done by *do_clocktick*.

Cause_alarm (line 11318) is simple; it sets to *TRUE* an entry in the *syn_table* array corresponding to the target of the synchronous alarm. If the synchronous alarm task is not alive, it is sent a message to wake it up.

Implementation of the Synchronous Alarm Task

The synchronous alarm task, *syn_alarm_task* (line 11333), follows the basic model of all tasks. It initializes and then enters an endless loop in which it receives and sends messages. The initialization consists of declaring itself alive by setting the variable *syn_al_alive* to *TRUE* and then declaring that it has nothing to do by setting all the entries in *syn_table* to *FALSE*. There is a slot in *syn_table* for each slot in the process table. In begins its outer loop by declaring it has completed its work and then enters an inner loop where it checks all slots in *syn_table*. If it finds an entry indicating a synchronous alarm is expected, it resets the entry, sends a message of type *CLOCK_INT* to the appropriate process, and declares its work not complete. At the bottom of its outer loop it does not pause to wait for any new messages unless its *work_done* flag is set. A new message is not needed to tell it there is more work to do, since *cause_alarm* writes directly into *syn_table*. A message is needed only to wake it up after it has run out of work. The effect is that it cycles very rapidly as long as there are alarms to be delivered.

In fact, this task is not used by the distribution version of MINIX. If you recompile MINIX to add networking support, it will be used by the network server, however, which needs exactly this kind of mechanism to enforce rapid timeouts if packets are not received when expected. In addition to the need for speed, a server cannot be sent a signal, since servers must run forever, and the default action of most signals is to kill the target process.

Implementation of the Clock Interrupt Handler

The design of the clock interrupt handler is a compromise between doing very little (so the processing time will be minimized) and doing enough to make expensive activations of the clock task infrequent. It changes a few variables and tests a few others. *Clock_handler* (line 11374) starts off by doing system accounting. MINIX keeps track of both user time and system time. User time is charged against a process if it is running when the clock ticks. System time is charged if the file system or memory manager is running. The variable *bill_ptr* always points to the last user process scheduled (the two servers do

not count). The billing is done on lines 11447 and 11448. After billing is finished, the most important variable maintained by *clack_handler*, *pending_ricks*, is incremented (line 11450). The real time must be known for testing whether *clock_handler* should wake up the tty or send a message to the clock task, but actually updating realtime itself is expensive, because this operation must be done using locks. To avoid this, the handler calculates its own version of the real time in the local variable *now*. There is a small chance that the result will be incorrect once in a while, but the consequences of such an error would not be serious.

The rest of the handler's work depends upon various tests. The terminal and the printer both need to be awakened from time to time. *Tty_timeout* is a global variable maintained by the terminal task, which holds the next time the tty should be awakened. For the printer several variables which are *PRIVATE* within the printer module need to be checked, and they are tested in the call to *pr_restart*, which returns quickly even in the worst case of the printer being hung up. On lines 11455 to 11458 a test is made that activates the clock task if an alarm is due or if it is time to schedule another task. The latter test is complex, a logical AND of three simpler tests. The

```
interrupt(CLOCK);
```

code on line 11459 results in a *HARD_INT* message to the clock task.

In discussing *do_clocktick* we noted that it decrements *sched_ticks* and tests for zero to see if the execution quantum has expired. Testing whether *sched_ticks* is equal to one is part of the complex test we mentioned above; if the clock task is not activated, it is still necessary to decrement *sched_ticks* within the interrupt handler and, if it reaches zero, reset the quantum. If this occurs, it is also time to note that the current process was active at the start of the new quantum; this is done by the assignment of the current value of *bill_ptr* to *prev_ptr* on line 11466.

Time Utilities

Finally, *clock.c* contains some functions that provide various kinds of support. Many of these are hardware specific and will need to be replaced for a port of MINIX to non-Intel hardware. We will only describe the function of these, without going into details of their internals.

Init_clock (line 11474) is called by the timer task when it runs for the first time. It sets the mode and time delay of the timer chip to produce clock tick interrupts 60 times per second. Despite the fact that the "CPU speed" one sees in advertisements for PCs has increased from 4.77 Mhz for the original IBM PC to over 200 Mhz for modern systems, the constant *TIMER_COUNT*, used to initialize the timer, is the same no matter what PC model MINIX is run on. Every IBM compatible PC, no matter how fast its processor runs, provides a 14.3 Mhz signal for use by various devices that need a time reference. Serial communications lines and the video display also need such a timing reference.

The complement of *init_clock* is *clock_stop* (line 11489). It is not really necessary, but it is a concession to the fact that MINIX users may want to start another operating system at times. It simply resets the timer chip parameters to the default mode of operation that MS-DOS and other operating systems may expect the ROM BIOS to have provided when they first start.

Milli_delay (line 11502) is provided for use by any task that needs very short delays. It is written in C without any hardware-specific references, but it uses a technique one might expect to find only in a low-level assembly language routine. It initializes a counter

to zero and then rapidly polls it until a desired value is reached. In Chapter 2 we said that this technique of busy waiting should generally be avoided, but the necessities of implementation can require exceptions to general rules. The initialization of the counter is done by the next function, *milli_start* (line ll5l6), which simply zeroes two variables. The polling is done by calling the last function, *milli_elapsed* (line 11529), which accesses the timer hardware. The counter that is examined is the same one used to count down clock ticks, and thus it can underflow and be reset to its maximum value before the desired delay is complete. *Milli_elapsed* corrects for this.

3.9 Terminals

Every general purpose computer has one or more terminals used to communicate with it. Terminals come in an extremely large number of different forms. It is up to the terminal driver to hide all these differences, so that the device-independent part of the operating system and the user programs do not have to be rewritten for each kind of terminal. In the following sections we will follow our now-standard approach of first discussing terminal hardware and software in general, and then discussing the MINIX software.

3.9.1 Terminal Hardware

From the operating system's point of view, terminals can be divided into three broad categories based on how the operating system communicates with them. The first category consists of memory-mapped terminals, which consist of a keyboard and a display, both of which are hardwired to the computer. The second category consists of terminals that interface via a serial communication line using the RS-232 standard, most frequently over a modem. The third category consists of terminals that are connected to the computer via a network. This taxonomy is shown in Fig. 3-27.

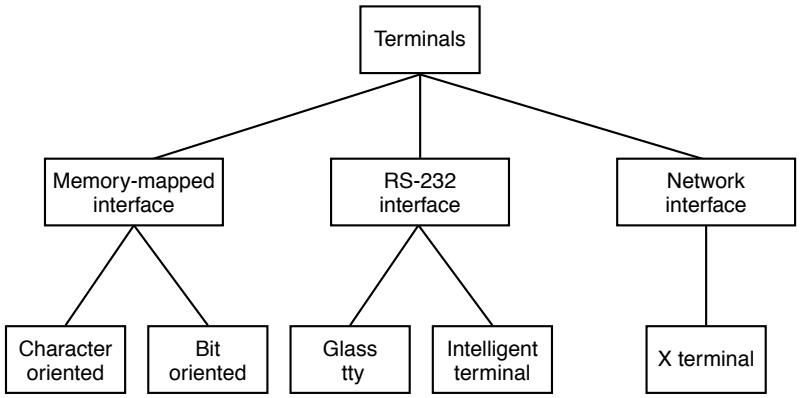


Figure 3-27. Terminal types.

Memory-Mapped Terminals

The first broad category of terminals named in Fig. 3-27 consists of memory-mapped terminals. These are an integral part of the computers themselves. Memory-mapped terminals are interfaced via a special memory called a **video RAM**, which forms part of the computer's address space and is addressed by the CPU the same way as the rest of memory (see Fig. 3-28).

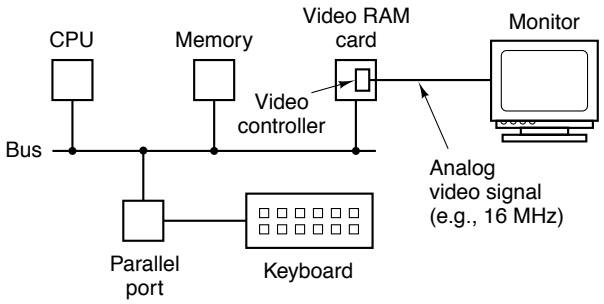


Figure 3-28. Memory-mapped terminals write directly into video RAM.

Also on the video RAM card is a chip called a **video controller**. This chip pulls character codes out of the video RAM and generates the video signal used to drive the display (monitor). The monitor generates a beam of electrons that scans horizontally across the screen, painting lines on it. Typically the screen has 480 to 1024 lines from top to bottom, with 640 to 1200 points per line. These points are called **pixels**. The video controller signal modulates the electron beam, determining whether a given pixel will be light or dark. Color monitors have three beams, for red, green, and blue, which are independently modulated.

A simple monochrome display might fit each character in a box 9 pixels wide by 14 pixels high (including the space between characters), and have 25 lines of 80 characters. The display would then have 350 scan lines of 720 pixels each. Each of these frames is redrawn 45 to 70 times a second. The video controller could be designed to fetch the first 80 characters from the video RAM, generate 14 scan lines, fetch the next 80 characters from the video RAM, generate the following 14 scan lines, and so on. In fact, most fetch each character once per scan line to eliminate the need for buffering in the controller. The 9-by-14 bit patterns for the characters are kept in a ROM used by the video controller. (RAM may also be used to support custom fonts.) The ROM is addressed by a 12-bit address, 8 bits from the character code and 4 bits to specify a scan line. The 8 bits in each byte of the ROM control 8 pixels; the 9th pixel between characters is always blank. Thus 14 x 80 = 1120 memory references to the video RAM are needed per line of text on the screen. The same number of references are made to the character generator ROM.

The IBM PC had several modes for the screen. In the simplest one, it uses a character-mapped display for the console. In Fig. 3-29(a) we see a portion of the video RAM. Each character on the screen of Fig. 3-29(b) occupies two characters in the RAM. The low-order character is the ASCII code for the character to be displayed. The high-order character is the attribute byte, which is used to specify the color, reverse video, blinking, and so on. The full screen of 25 by 80 characters required 4000 bytes of video RAM in this mode.

Bit-map terminals use the same principle, except that each pixel on the screen is individually controlled. In the simplest configuration, for a monochrome display, each pixel has a corresponding bit in the video RAM. At the other extreme, each pixel is represented by a 24-bit number, with 8 bits each for red, green, and blue. A 768 x 1024 color display with 24 bits per pixel requires 2 MB of RAM just to hold the image.

With a memory-mapped display, the keyboard is completely decoupled from the screen. It may be interfaced via a serial or parallel port. On every key action the CPU is interrupted, and the keyboard driver extracts the character typed by reading an I/O port.

On the IBM PC, the keyboard contains an embedded microprocessor which communicates through a specialized serial port with a controller chip on the main board. An

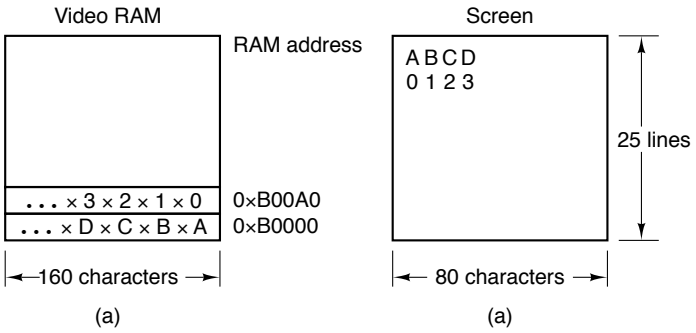


Figure 3-29. (a) A video RAM image for the IBM monochrome display. The xs are attribute bytes. (b) The corresponding screen.

interrupt is generated whenever a key is struck and also when one is released. Furthermore, all that the keyboard hardware provides is the key number, not the ASCII code. When the A key is struck, the key code (30) is put in an I/O register. It is up to the driver to determine whether it is lower case, upper case, CTRL-A, ALT-A, CTRL-ALT-A, or some other combination. Since the driver can tell which keys have been struck but not yet released (e.g., shift), it has enough information to do the job. Although this keyboard interface puts the full burden on the software, it is extremely flexible. For example, user programs may be interested in whether a digit just typed came from the top row of keys or the numeric key pad on the side. In principle, the driver can provide this information.

RS-232 Terminals

RS-232 terminals are devices containing a keyboard and a display that communicate using a serial interface, one bit at a time (see Fig. 3-30). These terminals use a 9-pin or 25-pin connector, of which one pin is used for transmitting data, one pin is for receiving data, and one pin is ground. The other pins are for various control functions, most of which are not used. To send a character to an RS-232 terminal, the computer must transmit it 1 bit at a time, prefixed by a start bit, and followed by 1 or 2 stop bits to delimit the character. A parity bit which provides rudimentary error detection may also be inserted preceding the stop bits, although this is commonly required only for communication with mainframe systems. Common transmission rates are 9600, l9,200, and 38,400 bps. RS-232 terminals are commonly used to communicate with a remote computer using at modem and a telephone line.

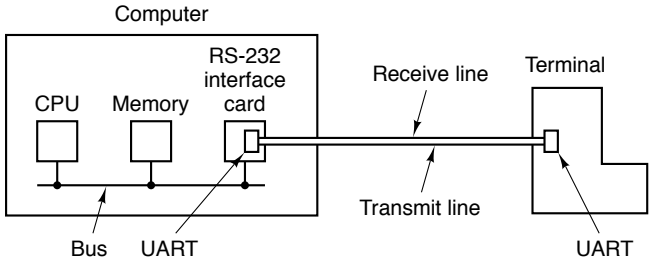


Figure 3-30. An RS-232 terminal communicates with a computer over a communication line, one bit at a time. The computer and the terminal are completely independent.

Since both computers and terminals work internally with whole characters but must

communicate over a serial line a bit at a time, chips have been developed to do the character-to-serial and serial-to-character conversions. They are called **UART**s (Universal Asynchronous Receiver Transmitters). UARTs are attached to the computer by plugging RS-232 interface cards into the bus as illustrated in Fig. 3-30. RS-232 terminals are gradually dying off, being replaced by PCs and X terminals, but they are still encountered on older mainframe systems, especially in banking, airline reservation, and similar applications.

To print a character, the terminal driver writes the character to the interface card, where it is buffered and then shifted out over the serial line one bit at a time by the UART. Even at 38,400 bps, it takes just over 250 microsec to send a character. As a result of this slow transmission rate, the driver generally outputs a character to the RS-232 card and blocks, waiting for the interrupt generated by the interface when the character has been transmitted and the UART is able to accept another character. The UART can simultaneously send and receive characters, as its name implies. An interrupt is also generated when a character is received, and usually a small number of input characters can be buffered. The terminal driver must check a register when an interrupt is received to determine the cause of the interrupt. Some interface cards have a CPU and memory and can handle multiple lines, taking over much of the I/O load from the main CPU.

RS-232 terminals can be subdivided into categories, as mentioned above. The simplest ones were hardcopy (printing) terminals. Characters typed on the keyboard were transmitted to the computer. Characters sent by the computer were typed on the paper. These terminals are obsolete and rarely seen any more.

Dumb CRT terminals work the same way, only with a screen instead of paper. These are often called "glass ttys" because they are functionally the same as hardcopy ttys. (The term "tty" is an abbreviation for Teletype® a former company that pioneered in the computer terminal business; "try" has come to mean any terminal.) Glass ttys are also obsolete.

Intelligent CRT terminals are in fact miniature, specialized computers. They have a CPU and memory and contain software, usually in ROM. From the operating system's viewpoint, the main difference between a glass tty and an intelligent terminal is that the latter understands certain escape sequences. For example, by sending the ASCII ESC character (033), followed by various other characters, it may be possible to move the cursor to any position on the screen, insert text in the middle of the screen, and so forth.

X Terminals

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 mouse. One common terminal of this type is the **X Terminal**, which runs M.I.T.'s X Window System. Usually, X terminals talk to the main computer over an Ethernet.

An X terminal is a computer that runs the X software. Some products are dedicated to running only X; others are general-purpose computers that simply run X as one program among many others. Either way, an X terminal has large bit-mapped screen, usually 960×1200 or better resolution, in black and white, gray-scale, or color, a full keyboard, and a mouse, normally with three buttons.

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 communicates over the network with **X clients** running on some remote host. 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. The arrangement of the client and server is shown in Fig. 3-31.

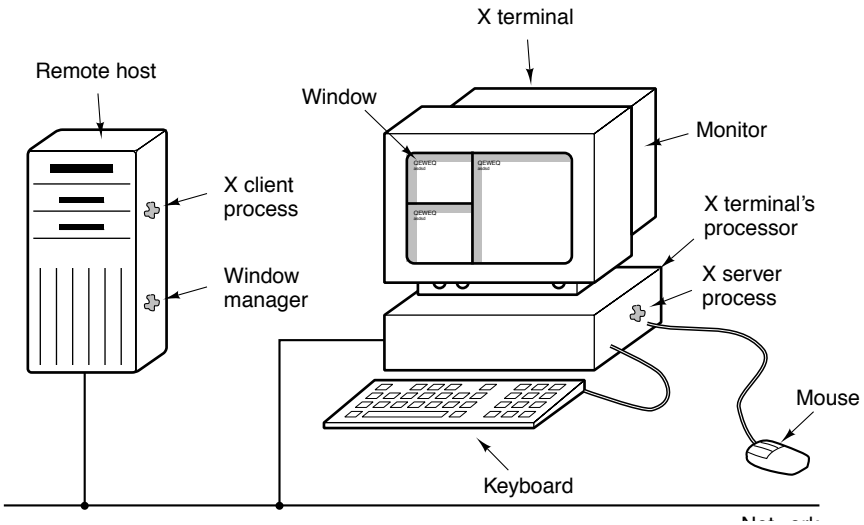


Figure 3-31. Clients and servers in the M.I.T. X Window System.

The screen of the X terminal contains some number of windows, each in the form of a rectangular grid of pixels. Each window usually has a title bar at the top, a scroll bar on the left, and a resizing box in the upper right-hand corner. One of the X clients is a program called a **window manager**. Its job is to control the creation, deletion, and movement of windows on the screen. To manage windows, it sends commands to the X server telling what to do. These commands include draw point, draw line, draw rectangle, draw polygon, fill rectangle, fill polygon, and so on.

The job of the X server is to coordinate input from the mouse, keyboard, and X clients and update the display accordingly. It has to keep track of which window is currently selected (where the mouse pointer is), so it knows which clients to send any new keyboard input to.

3.9.2 Terminal Software

The keyboard and display are almost independent devices, so we will treat them separately here. (They are not quite independent, since typed characters must be displayed on the screen.) In MINIX the keyboard and screen drivers are part of the same task; in other systems they may be split into distinct drivers.

Input Software

The basic job of the keyboard driver is to collect input from the keyboard and pass it to user programs when they read from the terminal. Two possible philosophies can be adopted for the driver. In the first one, the driver's job is just to accept input and pass it upward unmodified. A program reading from the terminal gets a raw sequence of ASCII codes. (Giving user programs the key numbers is too primitive, as well as being highly machine dependent.)

This philosophy is well suited to the needs of sophisticated screen editors such as *emacs*, which allow the user to bind an arbitrary action to any character or sequence of characters. It does, however, mean that if the user types *dste* instead of *date* and then

corrects the error by typing three backspaces and *ate*, followed by a carriage return, the user program will be given all ll ASCII codes typed.

Most programs do not want this much detail. They just want the corrected input, not the exact sequence of how it was produced. This observation leads to the second philosophy: the driver handles all the intraline editing, and just delivers corrected lines to the user programs. The first philosophy is character-oriented; the second one is line-oriented. Originally they were referred to as **raw mode** and **cooked mode**, respectively. The POSIX standard uses the less-picturesque term **canonical mode** to describe line-oriented mode. On most systems canonical mode refers to a well-defined configuration. **Noncanonical mode** is equivalent to raw mode, although many details of terminal behavior can be changed. POSIX-compatible systems provide several library functions that support selecting either mode and changing many aspects of terminal configuration. In MINIX the IOCTL system call supports these functions.

The first task of the keyboard driver is to collect characters. If every keystroke causes an interrupt, the driver can acquire the character during the interrupt. If interrupts are turned into messages by the low-level software, it is possible to put the newly acquired character in the message. Alternatively, it can be put in a small buffer in memory and the message used to tell the driver that something has arrived. The latter approach is actually safer if a message can be sent only to at waiting process and there is some chance that the keyboard driver might still be busy with the previous character.

Once the driver has received the character, it must begin processing it. If the keyboard delivers key numbers rather than the character codes used by application software, then the driver must convert between the codes by using a table. Not all IBM "compatibles" use standard key numbering, so if the driver wants to support these machines, it must map different keyboards with different tables. A simple approach is to compile a table that maps between the codes provided by the keyboard and ASCII (American Standard Code for Information Interchange) codes into the keyboard driver, but this is unsatisfactory for users of languages other than English. Keyboards are arranged differently in different countries, and the ASCII character set is not adequate even for the majority of people in the Western Hemisphere, where speakers of Spanish, Portuguese, and French need accented characters and punctuation marks not used in English. To respond to the need for flexibility of keyboard layouts to provide for different languages, many operating systems provide for loadable **keymaps** or **code pages**, which make it possible to choose the mapping between keyboard codes and codes delivered to the application, either when the system is booted or later.

If the terminal is in canonical (cooked) mode, characters must be stored until an entire line has been accumulated, because the user may subsequently decide to erase part of it. Even if the terminal is in raw mode, the program may not yet have requested input, so the characters must be buffered to allow type ahead. (System designers who do not allow users to type far ahead ought to be tarred and feathered, or worse yet, be forced to use their own system.)

Two approaches to character buffering are common. In the first one, the driver contains a central pool of buffers, each buffer holding perhaps 10 characters. Associated with each terminal is a data structure, which contains, among other items, a pointer to the chain of buffers for input collected from that terminal. As more characters are typed, more buffers are acquired and hung on the chain. When the characters are passed to a user program, the buffers are removed and put back in the central pool.

The other approach is to do the buffering directly in the terminal data structure itself, with no central pool of buffers. Since it is common for users to type a command that will

take a little while (say, a compilation) and then type a few lines ahead, to be safe the driver should allocate something like 200 characters per terminal. In a large-scale timesharing system with 100 terminals, allocating 20K all the time for type ahead is clearly overkill, so a central buffer pool with space for perhaps 5K is probably enough. On the other hand, a dedicated buffer per terminal makes the driver simpler (no linked list management) and is to be preferred on personal computers with only one or two terminals. Figure 3-32 shows the difference between these two methods.

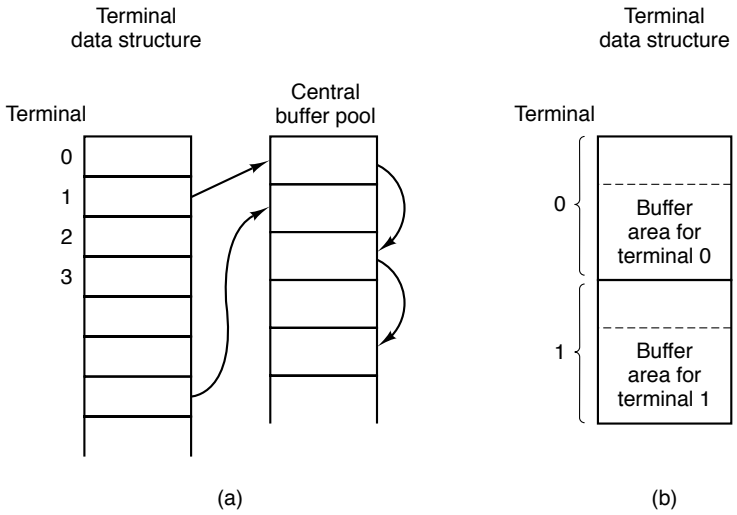


Figure 3-32. (a) Central buffer pool. (b) Dedicated buffer for each terminal.

Although the keyboard and display are logically separate devices, many users have grown accustomed to seeing the characters they have just typed appear on the screen. Some (older) terminals oblige by automatically displaying (in hardware) whatever has just been typed, which is not only a nuisance when passwords are being entered but greatly limits the flexibility of sophisticated editors and other programs. Fortunately, most modern terminals display nothing when keys are typed. It is therefore up to the software to display the input. This process is called **echoing**.

Echoing is complicated by the fact that a program may be writing to the screen while the user is typing. At the very least, the keyboard driver has to figure out where to put the new input without it being overwritten by program output.

Echoing also gets complicated when more than 80-characters are typed on a terminal with 80-character lines. Depending on the application, wrapping around to the next line may be appropriate. Some drivers just truncate lines to 80 charac- ters by throwing away all characters beyond column 80.

Another problem is tab handling. Most terminals have a tab key, but few can handle tab on output. It is up to the driver to compute where the cursor is currently located, taking into account both output from programs and output from echoing, and compute the proper number of spaces to be echoed.

Now we come to the problem of device equivalence. Logically, at the end of a line of text, one wants a carriage return, to move the cursor back to column 1, and a linefeed, to advance to the next line. Requiring users to type both at the end of each line would not sell well (although some terminals have a key which generates both, with a 50 percent chance of doing so in the order that the software wants them). It is up to the driver to convert whatever comes in to the standard internal format used by the operating system.

If the standard form is just to store a linefeed (the MINIX convention), then carriage

returns should be turned into linefeeds. If the internal format is to store both, then the driver should generate a linefeed when it gets a carriage return and a carriage return when it gets a linefeed. No matter what the internal convention, the terminal may require both a linefeed and a carriage return to be echoed in order to get the screen updated properly. Since a large computer may well have a wide variety of different terminals connected to it, it is up to the keyboard driver to get all the different carriage return/linefeed combinations converted to the internal system standard and arrange for all echoing to be done right.

A related problem is the timing of carriage return and linefeeds. On some terminals, it may take longer to display a carriage return or linefeed than a letter or number. If the microprocessor inside the terminal actually has to copy a large block of text to achieve scrolling, then linefeeds may be slow. If a mechanical print head has to be returned to the left margin of the paper, carriage returns may be slow. In both cases it is up to the driver to insert **filler characters** (dummy null characters) into the output stream or just stop outputting long enough for the terminal to catch up. The amount of time to delay is often related to the terminal speed, for example, at 4800 bps or slower, no delays may be, but at 9600 bps or higher one filler character might be required. Terminals with hardware tabs, especially hardcopy ones, may also require a delay after a tab.

When operating in canonical mode, a number of input characters have special meanings. Figure 3-33 shows all of the special characters required by POSIX and the additional ones recognized by MINIX. The defaults are all control characters that should not conflict with text input or codes used by programs, but all except the last two can be changed using the *stty* command, if desired. Older versions of UNIX used different defaults for many of these.

Character	POSIX name	Comment
CTRL-D	EOF	End of file
	EOL	End of line (undefined)
CTRL-H	ERASE	Backspace one character
DEL	INTR	Interrupt process (SIGINT)
CTRL-U	KILL	Erase entire line being typed
CTRL-\	QUIT	Force core dump (SIGQUIT)
CTRL-Z	SUSP	Suspend (ignored by MINIX)
CTRL-Q	START	Start output
CTRL-S	STOP	Stop output
CTRL-R	REPRINT	Redisplay input (MINIX extension)
CTRL-V	LNEXT	Literal next (MINIX extension)
CTRL-O	DISCARD	Discard output (MINIX extension)
CTRL-M	CR	Carriage return (unchangeable)
CTRL-J	NL	Linefeed (unchangeable)

Figure 3-33. Characters that are handled specially in canonical mode.

The *ERASE* character allows the user to rub out the character just typed. In MINIX it is the backspace (CTRL-H). It is not added to the character queue but instead removes the

previous character from the queue. It should be echoed as a sequence of three characters, backspace, space, and backspace, in order to remove the previous character from the screen. If the previous character was a tab, erasing it requires keeping track of where the cursor was prior to the tab. In most systems, backspacing will only erase characters on the current line. It will not erase a carriage retum and back up into the previous line.

When the user notices an error at the start of the line being typed in, it is often convenient to erase the entire line and start again. The *KILL* character (in MINIX CTRL-U) erases the entire line. MINIX makes the erased line vanish from the screen, but some systems echo it plus a carriage return and linefeed because some users like to see the old line. Consequently, how to echo *KILL* is a matter of taste. As with *ERASE* it is usually not possible to go further back than the current line. When a block of characters is killed, it may or may not be worth the trouble for the driver to return buffers to the pool, if one is used.

Sometimes the *ERASE* or *KILL* characters must be entered as ordinary data. The *LNEXT* character serves as an **escape character**. In MINIX CTRL-V is the default. As an example, older UNIX systems often used the @ sign for *KILL*, but the Internet mail system uses addresses of the form *linda@cs.washington.edu*. Someone who feels more comfortable with older conventions might redefine *KILL* as @, but then need to enter an @ sign literally to address e-mail. This can be done by typing CTRL-V @. The CTRL-V itself can be entered literally by typing CTRL-V CTRL-V. After seeing a CTRL-V, the driver sets a flag saying that the next character is exempt from special processing. The *LNEXT* character itself is not entered in the character queue.

To allow users to stop a screen image from scrolling out of view, control codes are provided to freeze the screen and restart it later. In MINIX these are *STOP*, (CTRL-S) and *START*, (CTRL-Q), respectively. They are not stored but are used to set and clear a flag in the terminal data structure. Whenever output is attempted, the flag is inspected. If it is set, no output occurs. Usually, echoing is also suppressed along with program output.

It is often necessary to kill a runaway program being debugged. The *INTR* (DEL) and *QUIT* (CTRL-\) characters can be used for this purpose. In MINIX, DEL sends the SIGINT signal to all the processes started up from the terminal. Implementing DEL can be quite tricky. The hard part is getting the information from the driver to the part of the system that handles signals, which, after all, has not asked for this information. CTRL-\ is similar to DEL, except that it sends the SIGQUIT signal, which forces a core dump if not caught or ignored. When either of these keys is struck, the driver should echo a carriage return and linefeed and discard all accumulated input to allow for a fresh start. The default value for *INTR* is often CTRL-C instead of DEL, since many programs use DEL interchangeably with the backspace for editing.

Another special character is *EOF* (CTRL-D), which in MINIX causes any pending read requests for the terminal to be satisfied with whatever is available in the buffer, even if the buffer is empty. Typing CTRL-D at the start of a line causes the program to get a read of 0 bytes, which is conventionally interpreted as end-of-file and causes most programs to act the same way as they would upon seeing end-of-file on an input file.

Some terminal drivers allow much fancier intraline editing than we have sketched here. They have special control characters to erase a word, skip backward or forward characters or words, go to the beginning or end of the line being typed, and so forth. Adding all these functions to the terminal driver makes it much larger and, furthermore, is wasted when using fancy screen editors that work in raw mode anyway.

To allow programs to control terminal parameters, POSIX requires that several functions be available in the standard library, of which the most important are *tcgetattr* and

tcsetattr. *Tcgetattr* retrieves a copy of the structure shown in Fig. 3-34, the *termios* structure, which contains all the information needed to change special characters, set modes, and modify other characteristics of a terminal. A program can examine the current settings and modify them as desired. *Tcsetattr* then writes the structure back to the terminal task.

```
struct termios {
  tcflag_t c_iflag;          /* input modes */
  tcflag_t c_oflag;          /* output modes */
  tcflag_t c_cflag;          /* control modes */
  tcflag_t c_lflag;          /* local modes */
  speed_t  c_ispeed;         /* input speed */
  speed_t  c_ospeed;         /* output speed */
  cc_t c_cc[NCCS];           /* control characters */
};
```

Figure 3-34. The termios structure. In MINIX tcflag_t is a short, speed_t is an int, cc_t is a char.

POSIX does not specify whether its requirements should be implemented through library functions or system calls. MINIX provides a system call, IOCTL, called by

```
ioctl(file_descriptor, request, argp);
```

that is used to examine and modify the configurations of many I/O devices. This call is used to implement the *tcgetattr* and *tcsetattr* functions. The variable *request* specifies whether the *termios* structure is to be read or written, and in the latter case, whether the request is to take effect immediately or should be deferred until all currently queued output is complete. The variable *argp* is a pointer to a *termios* structure in the calling program. This particular choice of communication between program and driver was chosen for its UNIX compatibility, rather than for its inherent beauty.

A few notes about the termios structure are in order. The four flag words provide a great deal of flexibility. The individual bits in *c_iflag* control various ways input is handled. For instance, the *ICRNL* bit causes *CR* characters to be converted into *NL* on input. This flag is set by default in MINIX. The *c_oflag* holds bits that affect output processing. For instance, the *OPOST* bit enables output processing. It and the *ONLCR* bit, which causes *NL* characters in the output to be converted into a *CR NL* sequence, are also set by default in MINIX. The *c_cflag* is the control flag. The default settings for MINIX enable a line to receive 8-bit characters and cause a modem to hang up if a user logs out on the line. The *c_lflag* is the *local mode* flags field. One bit, *ECHO*, enables echoing (this can be turned off during a login to provide security for entering a password). Its most important bit is the *ICANON* bit, which enables canonical mode. With *ICANON* off, several possibilities exist. If all other settings are left at their defaults, a mode identical to the traditional **cbreak mode** is entered. In this mode characters are passed to the program without waiting for a full line, but the *INTR*, *QUIT*, *START*, and *STOP* characters retain their effects. All of these can be disabled by resetting bits in the flags, however, to produce the equivalent of traditional raw mode.

The various special characters that can be changed, including those which are MINIX extensions, are held in the *c_cc* array. This array also holds two parameters which are used in noncanonical mode. The quantity *MIN*, stored in *c_cc*[*VMIN*], specifies the minimum

number of characters that must be received to satisfy a READ call. The quantity *TIME* in *c_cc*[*VTIME*] sets a time limit for such calls. *MIN* and *TIME* interact as shown in Fig. 3-35. A call that asks for *N* bytes is illustrated. With *TIME* = 0 and *MIN* = 1, the behavior is similar to the traditional raw mode.

	TIME = 0	**TIME > 0**
MIN = 0	Return immediately with whatever is available, 0 to N bytes	Timer starts immediately. Return with first byte entered or with 0 bytes after timeout
MIN > 0	Return with at least MIN and up to N bytes. Possible indefinite block.	Interbyte timer starts after first byte. Return N bytes if received by timeout, or at least 1 byte at timeout. Possible indefinite block

Figure 3-35. *MIN* and *TIME* determine when a call to read returns in non-canonical mode. *N* is the number of bytes requested.

Output Software

Output is simpler than input, but drivers for RS-232 terminals are radically different from drivers for memory-mapped terminals. The method that is commonly used for RS-232 terminals is to have output buffers associated with each terminal. The buffers can come from the same pool as the input buffers, or be dedicated, as with input. When programs write to the terminal, the output is first copied to the buffers. Similarly, output from echoing is also copied to the buffers. After all the output has been copied to the buffers (or the buffers are full), the first character is output, and the driver goes to sleep. When the interrupt comes in, the next character is output, and so on.

With memory-mapped terminals, a simpler scheme is possible. Characters to be printed are extracted one at a time from user space and put directly in the video RAM. With RS-232 terminals, each character to be output is just sent across the line to the terminal. With memory mapping, some characters require special treatment, among them, backspace, carriage return, linefeed, and the audible bell (CTRL-G). A driver for a memory-mapped terminal must keep track in software of the current position in the video RAM, so that printable characters can be put there and the current position advanced. Backspace, carriage return, and linefeed all require this position to be updated appropriately.

In particular, when a linefeed is output on the bottom line of the screen, the screen must be scrolled. To see how scrolling works, look at Fig. 3-29. If the video controller always began reading the RAM at 0xB0000, the only way to scroll the screen would be to copy 24 $\times$ 80 characters (each character requiring 2 bytes) from 0xB00A0 to 0xB0000, a time-consuming proposition.

Fortunately, the hardware usually provides some help here. Most video controllers contain a register that determines where in the video RAM to begin fetching bytes for the top line on the screen. By setting this register to point to 0xB00A0 instead of 0xB0000, the line that was previously number two moves to the top, and the whole screen scrolls up one line. The only other thing the driver must do is copy whatever is needed to the new bottom line. When the video controller gets to the top of the RAM, it just wraps around and continues fetching bytes starting at the lowest address.

Another issue that the driver must deal with on a memory-mapped terminal is cursor positioning. Again, the hardware usually provides some assistance in the form of a register that tells where the cursor is to go. Finally, there is the problem of the bell. It is sounded by outputting a sine or square wave to the loudspeaker, a part of the computer quite separate from the video RAM.

It is worth noting that many of the issues faced by the terminal driver for a memory-mapped display (scrolling, bell, and so on) are also faced by the microprocessor inside an RS-232 terminal. From the viewpoint of the microprocessor, it is the main processor in a system with a memory-mapped display.

Screen editors and many other sophisticated programs need to be able to update the screen in more complex ways than just scrolling text onto the bottom of the display. To accommodate them, many terminal drivers support a variety of escape sequences. Although some terminals support idiosyncratic escape sequence sets, it is advantageous to have a standard to facilitate adapting software from one system to another. The American National Standards Institute (ANSI) has defined a set of standard escape sequences, and MINIX supports a subset of the ANSI sequences, shown in Fig. 3-36, that is adequate for many common operations. When the driver sees the character that starts the escape sequences, it sets a flag and waits until the rest of the escape sequence comes in. When everything has arrived, the driver must carry it out in software. Inserting and deleting text require moving blocks of characters around the video RAM. The hardware is of no help with anything except scrolling and displaying the cursor.

Escape sequence	Meaning
ESC [*n* A	Move up *n* lines
ESC [*n* B	Move down *n* lines
ESC [*n* C	Move right *n* spaces
ESC [*n* D	Move left *n* spaces
ESC [*m* ; *n* H	Move cursor to (*m*,*n*)
ESC [*s* J	Clear screen from cursor (0 to end, 1 from start, 2 all)
ESC [*s* K	Clear line from cursor (0 to end, 1 from start, 2 all)
ESC [*n* L	Insert *n* lines at cursor
ESC [*n* M	Delete *n* lines at cursor
ESC [*n* P	Delete *n* chars at cursor
ESC [*n* @	Insert *n* chars at cursor
ESC [*n* m	Enable rendition *n* (0=normal, 4=bold, 5=blinking, 7=reverse)
ESC M	Scroll the screen backward if the cursor is on the top line

Figure 3-36. The ANSI escape sequences accepted by the terminal driver on output. ESC denotes the ASCII escape character (0x1B), and n, m, and s are optional numeric parameters.

3.9.3 Overview of the Terminal Driver in MINIX

The terminal driver is contained in four C files (six if RS-232 and pseudo terminal support are enabled) and together they far and away constitute the largest driver in MINIX. The size of the terminal driver is partly explained by the observation that the driver handles both the keyboard and the display, each of which is a complicated device in its own right, as well as two other optional types of terminals. Still, it comes as a surprise to most people to learn that terminal I/O requires thirty times as much code as the scheduler. (This

feeling is reinforced by looking at the numerous books on operating systems that devote thirty times as much space to scheduling as to all I/O combined.)

The terminal driver accepts seven message types:

1. Read from the terminal (from FS on behalf of a user process).
2. Write to the terminal (from FS on behalf of a user process).
3. Set terminal parameters for IOCTL (from FS on behalf of a user process).
4. I/O occurred during last clock tick (from the clock interrupt).
5. Cancel previous request (from the file system when a signal occurs).
6. Open a device.
7. Close a device.

The messages for reading and writing have the same format as shown in Fig. 3-15, except that no *POSITION* field is needed. With a disk, the program has to specify which block it wants to read. With a terminal, there is no choice: the program always gets the next character typed in. Terminals do not support seeks.

The POSIX functions *tcgetattr* and *tcsetattr*, used to examine and modify terminal attributes (properties), are supported by the IOCTL system call. Good programming practice is to use these functions and others in *include/termios.h* and leave it to the C library to convert library calls to IOCTL system calls. There are, however, some control operations needed by MINIX that are not provided for in POSIX, for example, loading an alternate keymap, and for these the programmer must use IOCTL explicitly.

The message sent to the driver by an IOCTL system call contains a function request code and a pointer. For the *tcsetattr* function, an IOCTL call is made with a *TCSETS*, *TCSETSW*, or *TCSETSF* request type, and a pointer to a *termios* structure like the one shown in Fig. 3-34. All such calls replace the current set of attributes with a new set, the differences being that a *TCSETS* request takes effect immediately, a *TCSETSW* request does not take effect until all output has been transmitted, and a *TCSETSF* waits for output to finish and discards all input that has not yet been read. *Tcgetattr* is translated into an IOCTL call with a *TCGETS* request type and returns a filled in *termios* structure to the caller, so the current state of a device can be examined. IOCTL calls that do not correspond to functions defined by POSIX, like the *KIOCSMAP* request used to load a new keymap, pass pointers to other kinds of structures, in this case to a *keymap_t* which is a 1536-byte structure (16-bit codes for 128 keys $\times$ 6 modifiers). Figure 3-43 summarizes how standard POSIX calls are converted into IOCTL system calls.

The terminal driver uses one main data structure, *tty_table*, which is an array of *tty* structures, one per terminal. A standard PC has only one keyboard and display, but MINIX can support up to eight virtual terminals, depending upon the amount of memory on the display adapter card. This permits the person at the console to log on multiple times, switching the display output and keyboard input from one "user" to another. With two virtual consoles, pressing ALT-F2 selects the second one and ALT-F1 returns to the first. ALT plus the arrow keys also can be used. In addition, serial lines can support two users at remote locations, connected by RS-232 cable or modem, and **pseudo terminals** can support users connected through a network. The driver has been written to make it easy to add additional terminals. The standard configuration illustrated in the source code in this text has two virtual consoles, with serial lines and pseudo terminals disabled.

Each *tty* structure in *tty_table* keeps track of both input and output. For input, it holds a queue of all characters that have been typed but not yet read by the program, information about requests to read characters that have not yet been received, and timeout information, so input can be requested without the task blocking permanently if no character is typed. For output, it holds the parameters of write requests that are not yet finished. Other fields hold various general variables, such as the *termios* structure discussed above, which affects many properties of both input and output. There is also a field in the *tty* structure to point to information which is needed for a particular class of devices but is not needed in the *tty_table* entry for every device. For instance, the hardware-dependent part of the console driver needs the current position on the screen and in the video RAM, and the current attribute byte for the display, but this information is not needed to support an RS-232 line. The private data structures for each device type are also where the buffers that receive input from the interrupt service routines are located. Slow devices, such as the keyboard, do not need buffers as large as those needed by fast devices.

Terminal Input

To better understand how the driver works, let us first look at how characters typed in on the terminal work their way through the system to the program that wants them.

When a user logs in on the system console, a shell is created for him with */dev/console* as standard input, standard output, and standard error. The shell starts up and tries to read from standard input by calling the library procedure *read*. This procedure sends a message that contains the file descriptor, buffer address, and count to the file system. This message is shown as (1) in Fig. 3-37. After sending the message, the shell blocks, waiting for the reply. (User processes execute only the SEND_REC primitive, which combines a SEND with a RECEIVE from the process sent to.)

The file system gets the message and locates the i-node corresponding to the specified file descriptor. This i-node is for the character special file */dev/console* and contains the major and minor device numbers for the terminal. The major device type for terminals is 4; for the console the minor device number is 0.

The file system indexes into its device map, *dmap*, to find the number of the terminal task. Then it sends a message to the terminal task, shown as (2) in Fig. 3-37. Normally, the user will not have typed anything yet, so the terminal driver will be unable to satisfy the request. It sends a reply back immediately to unblock the file system and report that no characters are available, shown as (3). The file system records the fact that a process is waiting for terminal input in the console's structure in *tty_table* and then goes off to get the next request for work. The user's shell remains blocked until the requested characters arrive, of course.

When a character is finally typed on the keyboard, it causes two interrupts, one when the key is depressed and one when it is released. This rule also applies to modifier keys such as CTRL and SHIFT, which do not transmit any data by themselves but still cause two interrupts per key. The keyboard interrupt is IRQ 1, and *_hwint01* in the assembly code file *mpx386.s* activates *kbd_hw_int* (line 13123), which in turn calls *scan_keyboard* (line 13432) to extract the key code from the keyboard hardware. If the code is for an ordinary character, it is put into the keyboard input queue, *ibuf*, if the interrupt was generated by a key being depressed, but it is ignored if the interrupt was generated by the release of a key. Codes for modifier keys like CTRL and SHIFT are put into the queue for both types of interrupt but can be distinguished later by a bit that is set only when a key is released. Note that at this point the codes received and stored in *ibuf* are not ASCII

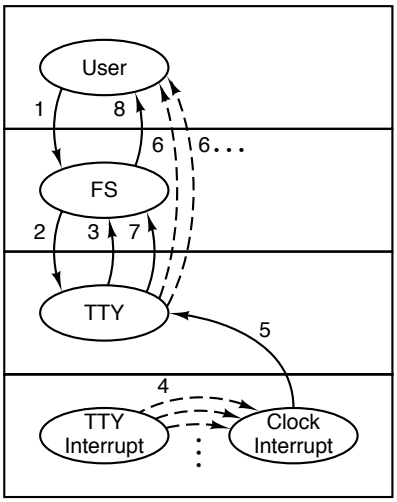


Figure 3-37. Read request from terminal when no characters are pending. FS is the file system. TTY is the terminal task. The interrupt handler for the terminal queues characters as they are entered, but it is the clock interrupt handler that awakens TTY.

codes; they are simply the scan codes produced by the IBM keyboard. *Kbd_hw_int* then sets a flag, *tty_events* (part of the keyboard's section of the *tty_table*), calls *force_timeout*, and returns.

Unlike some other interrupt service routines, *kbd_kw_int* does not send a message to wake up the terminal task. The call to *force_timeout* is indicated by the dashed lines marked (4) in the figure. These are not messages. They set the *tty_timeout* variable in the address space common to the interrupt service routines. On the next clock interrupt *clock_handler* finds that *tty_timeout* indicates it is time for a call to *tty_wakeup* (line 11452) which then sends a message (5) to the terminal task. Note that although the source code for *tty_wakeup* is in the file *tty.c*, it runs in response to the clock interrupt, and thus we say the clock interrupt sends the message to the terminal task. If input is arriving rapidly, a number of character codes may be queued this way, which is why multiple calls to *force_timeout* (4) are shown in the figure.

Upon receiving the wakeup message the terminal task inspects the *tty_events* flag for each terminal device, and, for each device which has the flag set, calls *handle_events* (line 12256). The *tty_events* flag can signal various kinds of activity (although input is the most likely), so *handle_events* always calls the device-specific functions for both input and output. For input from the keyboard this results in a call to *kb_read* (line 13165), which keeps track of keyboard codes that indicate pressing or releasing of the CTRL, SHIFT, and ALT keys and converts keyboard codes into ASCII codes. *Kb_read* in turn calls *in_process* (line 12367), which processes the ASCII codes, taking into account special characters and different flags that may be set, including whether or not canonical mode is in effect. The effect is normally to add characters to the console's input queue in *tty_table*, although some codes, for instance BACKSPACE, have other effects. Normally, also, *in_process* initiates echoing of the ASCII codes to the display.

When enough characters have come in, the terminal task calls the assembly language procedure *phys_copy* to copy the data to the address requested by the shell. This operation also is not message passing and for that reason is shown by dashed lines (6) in Fig. 3-37. There is more than one such line shown because there may be more than one such operation before the user's request has been completely fulfilled. When the operation is

finally complete, the terminal driver sends a message to the file system telling it that the work has been done (7), and the file system reacts to this message by sending a message back to the shell to unblock it (8).

The definition of when enough characters have come in depends upon the terminal mode. In canonical mode a request is complete when a linefeed, end-of-line, or end-of-file code is received, and, in order for proper input processing to be done, a line of input cannot exceed the size of the input queue. In noncanonical mode a read can request a much larger number of characters, and *in_process* may have to transfer characters more than once before a message is returned to the file system to indicate the operation is complete.

Note that the terminal driver copies the actual characters directly from its own address space to that of the shell. It does not first go through the file system. With block I/O, data do pass through the file system to allow it to maintain a buffer cache of the most recently used blocks. If a requested block happens to be in the cache, the request can be satisfied directly by the file system, without doing any disk I/O.

For terminal I/O, a cache makes no sense. Furthermore, a request from the file system to a disk driver can always be satisfied in at most a few hundred milliseconds, so there is no real harm in having the file system just wait. Terminal I/O may take hours to complete, or may never be complete (in canonical mode the terminal task waits for a complete line, and it may also wait a long time in noncanonical mode, depending upon the settings of *MIN* and *TIME*). Thus, it is unacceptable to have the file system block until a terminal input request is satisfied.

Later on, it may happen that the user has typed ahead, and that characters are available before they have been requested, from previous occurrences of events 4 and 5. In that case, events 1, 2, 6, 7, and 8 all happen in quick succession after the read request; 3 does not occur at all.

If the terminal task happens to be running at the time of a clock interrupt, no message can be sent to it because it will not be waiting for one. However, in order to keep input and output flowing smoothly when the terminal task is busy, the *tty_events* flags for all terminal devices are inspected at several other times, for instance, immediately after processing and replying to a message. Thus, it is possible for characters to be added to the console queue without the aid of a wakeup message from the clock. If two or more clock interrupts occur before the terminal driver finishes what it is doing, all the characters are stored in *ibuf* and *tty_flags* is repeatedly set. Ultimately, the terminal task gets one message; the rest are lost. But since all the characters are safely stored in the buffer, no typed input is lost. It is even possible that by the time a message is received by the terminal task the input is complete and a reply has already been sent to the user process.

The problem of what to do in an unbuffered message system (rendezvous principle) when an interrupt routine wants to send a message to a process that is busy is inherent in this kind of design. For most devices, such as disks, interrupts occur only in response to commands issued by the driver, so only one interrupt can be pending at any instant. The only devices that generate interrupts on their own are the clock and terminals (and when enabled, the network). The clock is handled by counting pending ticks, so if the clock task does not receive a message from the clock interrupt, it can compensate later. Terminals are handled by having the interrupt routine accumulate the characters in a buffer and raising a flag to indicate characters have been received. If the terminal task is running, it checks all of these flags before it goes to sleep and postpones going to sleep if there is more work it can do.

The terminal task is not awakened directly by terminal interrupts due to the excessive

overhead doing so would entail. The clock sends an interrupt to the terminal task on the next tick following each terminal interrupt. At 100 words per minute a typist enters fewer than l0 characters per second. Even with a fast typist the terminal task will probably be sent an interrupt message for each character typed at the keyboard, although some of these messages may be lost. If the buffer should fill before being emptied, excess characters are discarded, but experience shows that, for the keyboard, a 32-character buffer is adequate. In the case of other input devices higher data rates are probable—rates 1000 or more times faster than those of a typist are possible from a serial port connected to a 28,800-bps modem. At that speed approximately 48 characters may be received between clock ticks by the modem, but to allow for data compression on the modem link the serial port connected to the modem must be able to handle at least twice as many. For serial lines, MINIX provides a buffer of 1024 characters.

We have some regrets that the terminal task cannot be implemented without some compromise of our general design principles, but the method we use does the job without too much additional software complexity and no loss in performance. The obvious alternative, to throw away the rendezvous principle and have the system buffer all messages sent to destinations not waiting for them, is much more complicated and also slower.

Real system designers are often faced with a trade-off between using the general case, which is elegant all the time but somewhat slow, and using simpler techniques, which are usually fast but in one or two cases require a trick to make them work properly. Experience is really the only guide to which approach is better under given circumstances. A considerable amount of experience on designing operating systems is summarized by Lampson (1984) and Brooks (1975). While old, these references are still classics.

We will complete our overview of terminal input by summarizing the events that occur when the terminal task is first activated by a read request and when it is reactivated after receipt of keyboard input (see Fig. 3-38). In the first case, when a message comes in to the terminal task requesting characters from the keyboard, the main procedure, *tty_task* (line 11817) calls *do_read* (line 11891) to handle the request. *Do_read* stores the parameters of the call in the keyboard's entry in *tty_table*, in case there are insufficient characters buffered to satisfy the request.

Then it calls *in_transfer* (line 12303) to get any input already waiting, and then *handle_events* (line 12256) which in turn calls *kb_read* (line 13165) and then *in_transfer* once again, in order to try to milk the input stream for a few more characters. *Kb_read* calls several other procedures not shown in Fig. 3-38 to accomplish its work. The result is that whatever is immediately available is copied to the user. If nothing is available, nothing is copied. If the read is completed by *in_transfer* or by *handle_events*, a message is sent to the file system when all characters have been transferred, so the file system can unblock the caller. If the read was not completed (no characters, or not enough characters) *do_read* reports back to the file system, telling it whether it should suspend the original caller, or, if a nonblocking read was requested, cancel the read.

The right side of Fig. 3-38 summarizes the events that occur when the terminal task is awakened subsequent to an interrupt from the keyboard. When a character is typed, the interrupt procedure *kb_hw_int* (line 13123) puts the character code received into the keyboard buffer, sets a flag to identify that the console device has experienced an event, and then arranges for a timeout to occur on the next clock tick. The clock task sends a message to the terminal task telling it something has happened. Upon receiving this message, *tty_task* checks the event flags of all terminal devices and calls *handle_event* for each device with a raised flag. In the case of the keyboard, *handle_event* calls *kb_read* and *in_transfer*, just as was done on receipt of the original read request. The events shown on

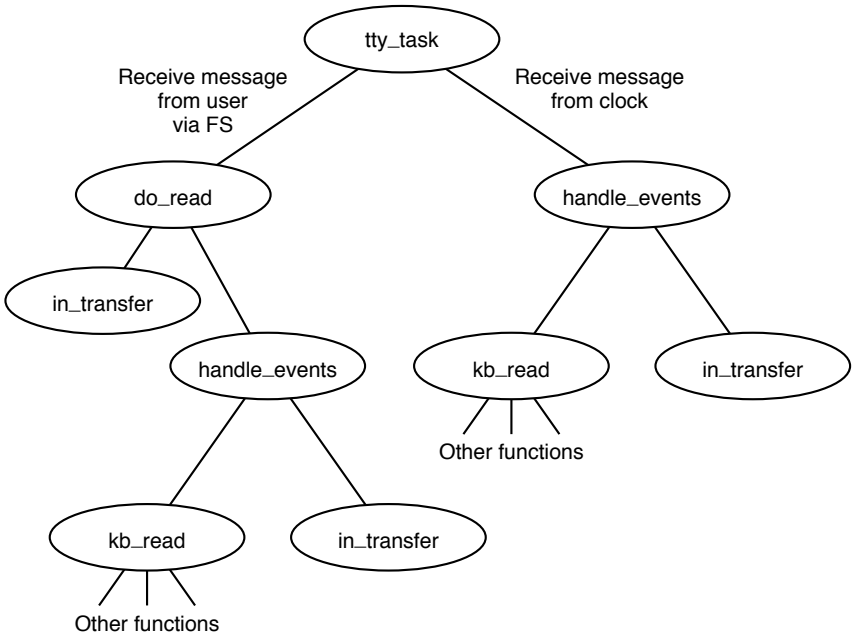


Figure 3-38. Input handling in the terminal driver. The left branch of the tree is taken to process a request to read characters. The right branch is taken when a character-has-been-typed message is sent to the driver.

the right side of the figure may occur several times, until enough characters are received to fulfill the request accepted by *do_read* after the first message from the FS. If the FS tries to initiate a request for more characters from the same device before the first request is complete, an error is returned. Of course, each device is independent; a read request on behalf of a user at a remote terminal is processed separately from one for a user at the console.

The functions not shown in Fig. 3-38 that are called by *kb_read* include *map_key*, which converts the key codes (scan codes) generated by the hardware into ASCII codes, *make_break*, which keeps track of the state of modifier keys such as the SHIFT key, and *in_process*, which handles complications such as attempts by the user to backspace over input entered by mistake, other special characters, and options available in different input modes. *In_process* also calls *echo* (line 12531), so the typed characters will be displayed on the screen.

Terminal output

In general, console output is simpler than terminal input, because the operating system is in control and does not need to be concerned with requests for output arriving at inconvenient times. Also, because the MINIX console is a memory-mapped display, output to the console is particularly simple. No interrupts are needed; the basic operation is to copy data from one memory region to another. On the other hand, all the details of managing the display, including handling escape sequences, must be handled by the driver software. As we did with keyboard input in the previous section we will trace through the steps involved in sending characters to the console display. We will assume in this example that the active display is being written; minor complications caused by virtual consoles will be discussed later.

When a process wants to print something, it generally calls *pringf*. *Printf* calls

WRITE to send a message to the file system. The message contains a pointer to the characters to be printed (not the characters themselves). The file system then sends a message to the terminal driver, which fetches them and copies them to the video RAM. Figure 3-39 shows the main procedures involved in output.

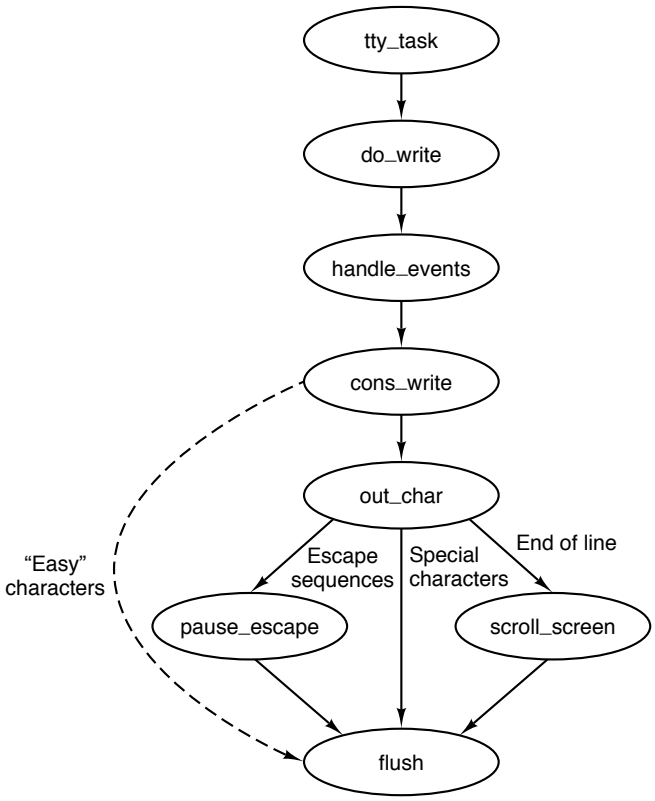


Figure 3-39. Major procedures used on terminal output. The dashed line indicates characters copied directly to *ramqueue* by *cons_write*.

When a message comes in to the terminal task requesting it to write on the screen, *do_write* (line 11964) is called to store the parameters in the console's *tty* struct in the *tty_table*. Then *handle_events* (the same function called whenever the *tty_events* flag is found set) is called. On every call this function calls both the input and output routines for the device selected in its argument. In the case of the console display this means that any keyboard input that is waiting is processed first. If there is input waiting, characters to be echoed are added to whatever characters are already awaiting output. Then a call is made to *cons_write* (line 13729), the output procedure for memory-mapped displays. This procedure uses *phys_copy* to copy blocks of characters from the user process to a local buffer, possibly repeating this and the following steps a number of times, since the local buffer holds only 64 bytes. When the local buffer is full, each 8-bit byte is transferred to another buffer, *ramqueue*. This is an array of 16-bit words. Alternate bytes are filled in with the current value of the screen attribute byte, which determines foreground and background colors and other attributes. When possible, characters are transferred directly into *ramqueue*, but certain characters, such as control characters or characters that are parts of escape sequences, need special handling. Special handling is also required when a character's screen position would exceed the width of the screen, or when *ramqueue* becomes full. In these cases *out_char* (line 13809) is called to transfer the characters and take whatever additional action is called for. For instance, *scroll_screen* (line 13896) is called when a linefeed is received while addressing the last line of the screen, and

parse_escape handles characters during an escape sequence. Usually *out_char* calls *flush* (line 13951) which copies the contents of *ramqueue* to the video display memory, using the assembly language routine *mem_vid_copy*. *Flush* is also called after the last character is transferred into *ramqueue* to be sure all output is displayed. The final result of *flush* is to command the 6845 video controller chip to display the cursor in the correct position.

Logically, the bytes fetched from the user process could be written into the video RAM one per loop iteration. However, accumulating the characters in *ramqueue* and then copying the block with a call to *mem_vid_copy* are more efficient in the protected memory environment of Pentium-class processors. Interestingly, this technique was introduced in early versions of MINIX that ran on older processors without protected memory. The precursor of *mem_vid_copy* dealt with a timing problem—with older video displays the copy into the video memory had to be done when the screen was blanked during vertical retrace of the CRT beam to avoid generating visual garbage all over the screen. MINIX no longer provides this support for obsolete equipment as the performance penalty is too great. However, the modem version of MINIX benefits in other ways from copying *ramqueue* as a block.

The video RAM available to a console is delimited in the console structure by the fields *c_start* and *c_limit*. The current cursor position is stored in the *c_column* and *c_row* fields. The coordinate (0, 0) is in the upper left corner of the screen, which is where the hardware starts to fill the screen. Each video scan begins at the address given by *c_org* and continues for 80 $\times$ 25 characters (4000 bytes). In other words, the 6845 chip pulls the word at offset *c_org* from the video RAM and displays the character byte in the upper left-hand corner, using the attribute byte to control color, blinking, and so forth. Then it fetches the next word and displays the character at (1, 0). This process continues until it gets to (79, 0), at which time it begins the second line on the screen, at coordinate (0, 1).

When the computer is first started, the screen is cleared, output is written into the video RAM starting at location *c_start*, and *c_org* is assigned the same value as *c_start*. Thus the first line appears on the top line of the screen. When output must go to a new line, either because the first line is full or because a newline character is detected by *out_char*, output is written into the location given by *c_start* plus 80. Eventually all 25 lines are filled, and **scrolling** of the screen is required. Some programs, editors, for example, require scrolling in the downward direction too, when the cursor is on the top line and further movement upward within the text is required.

There are two ways scrolling the screen can be managed. In **software scrolling** the character to be displayed at position (0, 0) is always in the first location in video memory, word 0 relative to the position pointed to by *c_start*, and the video controller chip is commanded to display this location first by keeping the same address in *c_org*. When the screen is to be scrolled, the contents of relative location 80 in the video RAM, the beginning of the second line on the screen, is copied to relative location 0, word 81 is copied to relative location 1, and so on. The scan sequence is unchanged, putting the data at location 0 in the memory at screen position (0, 0) and the image on the screen appears to have moved up one line. The cost is that the CPU has moved 80 $\times$ 24 = 1920 words. In **hardware scrolling** the data are not moved in the memory; instead the video controller chip is instructed to start the display at a different point, for instance, with the data at word 80. The bookkeeping is done by adding 80 to the contents of *c_org*, saving it for future reference, and writing this value into the correct register of the video controller chip. This requires either that the controller be smart enough to wrap around the video RAM, taking data from the beginning of the RAM (the address in *c_start*) when it reaches the end (the address in *c_limit*), or that the video RAM have more capacity than just the 80 x 2000

words necessary to store a single screen of display. Older display adapters generally have smaller memory but are able to wrap around and do hardware scrolling. Newer adapters generally have much more memory than needed to display a single screen of text, but the controllers are not able to wrap. Thus an adapter with 32768 bytes of display memory can hold 204 complete lines of 160 bytes each, and can do hardware scrolling 179 times before the inability to wrap becomes a problem. But, eventually a memory copy operation will be needed to move the data for the last 24 lines back to location 0 in the video memory. Whichever method is used, a row of blanks is copied to the video RAM to ensure that the new line at the bottom of the screen is empty.

When virtual consoles are configured, the available memory within a video adapter is divided equally between the number of consoles desired by properly The position of the cursor relative to the start of the video RAM can be derived from *c_column* and *c_row*, but it is faster to store it explicitly (in *c_cur*). When a character is to be printed, it is put into the video RAM at location *c_cur*, which is then updated, as is *c_column*. Figure 3-40 summarizes the fields of the console structure that affect the current position and the display origin.

Field	Meaning
c_start	Start of video memory for this console
c_limit	Limit of video memory for this console
c_column	Current column (0-79) with 0 at left
c_row	Current row (0-24) with 0 at top
c_cur	Offset into video RAM for cursor
c_org	Location in RAM pointed to by 6845 base register

Figure 3-40. Fields of the console structure that relate to the current screen position.

The characters that affect the cursor position (e.g., linefeed, backspace) are handled by adjusting the values of *c_column*, *c_row*, and *c_cur*. This work is done at the end of flush by a call to *set_6845* which sets the registers in the video controller chip.

The terminal driver supports escape sequences to allow screen editors and other interactive programs to update the screen in a flexible way. The sequences supported are a subset of an ANSI standard and should be adequate to allow many programs written for other hardware and other operating systems to be easily parted to MINIX. There are two categories of escape sequences: those that never contain a variable parameter, and those that may contain parameters. In the first category the only representative supported by MINIX is ESC M, which reverse indexes the screen, moving the cursor up one line and scrolling the screen downward if the cursor is already on the first line. The other category can have one or two numeric parameters. Sequences in this group all begin with ESC [. The "[" is the **control sequence introducer**. A table of escape sequences defined by the ANSI standard and recognized by MINIX was shown in Fig. 3-36. Parsing escape sequences is not trivial. Valid escape sequences in MINIX can be as short as two characters, as in ESC M, or up to 8 characters long in the case of a sequence that accepts two numeric parameters that each can have a two-digit values as in ESC [20;60H, which moves the cursor to line 20, column 60. In a sequence that accepts a parameter, the parameter may be omitted, and in a sequence that accepts two parameters either or both may

be omitted; When a parameter is omitted or one that is outside the valid range is used, a default is substituted. The default is the lowest valid value.

Consider the following ways one could construct a sequence to move to the upper-left corner of the screen:

1. ESC [H is acceptable, because if no parameters are entered the lowest valid parameters are assumed.

2. ESC [1;1H will correctly send the cursor to row l and column I (with ANSI, the row and column numbers start at 1).

3. Both ESC [1;H and ESC [;1H have an omitted parameter, which defaults to 1 as in the first example.

4. ESC [0;0H will do the same, since each parameter is less than the minimum valid value the minimum is substituted.

These examples are presented not to suggest one should deliberately use invalid parameters but to show that the code that parses such sequences is nontrivial.

MINIX implements a finite state automaton to do this parsing. The variable *c_esc_state* in the console structure normally has a value of 0. When *out_char* detects an ESC character, it changes *c_esc_state* to 1, and subsequent characters are processed by *parse_escape* (line 13986); If the next character is the control sequence introducer, state 2 is entered; otherwise the sequence is considered complete, and *do_escape* (line 14045) is called. In state 2, as long as incoming characters are numeric, a parameter is calculated by multiplying the previous value of the parameter (initially 0) by 10 and adding the numeric value of the current character. The parameter values are kept in an array and when a semicolon is detected the processing shifts to the next cell in the array. (The array in MINIX has only two elements, but the principle is the same). When a nonnumeric character that is not a semicolon is encountered the sequence is considered complete, and again *do_escape* is called. The current character on entry to *do_escape* then is used to select exactly what action to take and how to interpret the parameters, either the defaults or those entered in the character stream. This is illustrated in Fig. 3-48.

Loadable Keymaps

The IBM PC keyboard does not generate ASCII codes directly. The keys are each identified with a number, starting with the keys that are located in the upper left of the original PC keyboard—1 for the "ESC" key, 2 for the "1", and so on. Each key is assigned a number, including modifier keys like the left SHIFT and right SHIFT keys, numbers 42 and 54. When a key is pressed, MINIX receives the key number as a scan code. A scan code is also generated when a key is released, but the code generated upon release has the most significant bit set (equivalent to adding 128 to the key number). Thus a key press and a key release can be distinguished. By keeping track of which modifier keys have been pressed and not yet released, a large number of combinations are possible. For ordinary purposes, of course, two-finger combinations, such as SHIFT-A or CTRL-D, are most manageable for two-handed typists, but for special occasions three-key (or more) combinations are possible, for instance, CTRL-SHIFT-A, or the well-known CTRL-ALT-DEL combination that PC users recognize as the way to reset and reboot the system.

The complexity of the PC keyboard allows for a great deal of flexibility in how it is used. A standard keyboard has 47 ordinary character keys defined (26 alphabetic, 10

numeric, and 11 punctuation). If we are willing to use three-fingered modifier key combinations, such as CTRL-ALT-SHIFT, we can support a character set of 376 (8 × 47) members. This is by no means the limit of what is possible, but for now let us assume we do not want to distinguish between the left- and right-hand modifier keys, or use any of the numeric keypad or function keys. Indeed, we are not limited to using just the CTRL, ALT, and SHIFT keys as modifiers; we could retire some keys from the set of ordinary keys and use them as modifiers if we desired to write a driver that supported such a system.

Operating systems that use such keyboards use a **keymap** to determine what character code to pass to a program based upon the key being pressed and the modifiers in effect. The MINIX keymap logically is an array of 128 rows, representing possible scan code values (this size was chosen to accommodate Japanese keyboards; U.S. and European keyboards do not have this many keys) and 6 columns. The columns represent no modifier, the SHIFT key, the Control key, the left ALT key, the right ALT key, and a combination of either ALT key plus the SHIFT key. There are thus 720 ((128 - 6) × 6) character codes that can be generated by this scheme, given an adequate keyboard. This requires that each entry in the table be a 16-bit quantity. For U.S. keyboards the ALT and ALT2 columns are identical. ALT2 is named ALTGR on keyboards for other languages, and many of these keymaps support keys with three symbols by using this key as a modifier.

A standard keymap (determined by the line

```
#include keymaps/us-std.src
```

in keyboard.c) is compiled into the minix kernel at compilation time, but an

```
ioctl(0, KIOCSMAP, keymap)
```

call can be used to load a different map into the kernel at address keymap. A full keymap occupies 1536 bytes (128 × 6 × 2). Extra keymaps are stored in compressed form. A program called *genmap* is used to make a new compressed keymap. When compiled, *genmap* includes the *keymap.src* code for a particular keymap, so the map is compiled within *genmap*. Normally, *genmap* is executed immediately after being compiled, at which time it outputs the compressed version to a file, and then the *genmap* binary is deleted. The command *loadkeys* reads a compressed keymap, expands it internally, and then calls IOCTL to transfer the keymap into the kernel memory. MINIX can execute *loadkeys* automatically upon starting, and the program can also be invoked by the user at any time.

The source code for a keymap defines a large initialized array, and in the interest of saving space a keymap file is not printed with the source code. Figure 3-41 shows in tabular form the contents of a few lines of *src/kernel/keymaps/us-std.src* which illustrate several aspects of keymaps. There is no key on the IBM-PC keyboard that generates a scan code of 0. The entry for code 1, the ESC key, shows that the value returned is unchanged when the SHIFT key or CTRL key are pressed, but that a different code is returned when an ALT key is pressed simultaneously with the ESC key. The values compiled into the various columns are determined by macros defined in *include/minix/keymap.h*:

```
#define C(c)     ((c) & 0x1F)     /* Map to control code */
#define A(c)     ((c) | 0x80)     /* Set eight bit (ALT) */
#define CA(c)    A(C(c))          /* CTRL-ALT */
#define L(c)     ((c) | HASCAPS)  /* Add "Caps Lock has effect" attribute */
```

The first three of these macros manipulate bits in the code for the quoted character to produce the necessary code to be returned to the application. The last one sets the HASCAPS

Scan code	Character	Regular	SHIFT	ALT1	ALT2	ALT+SHIFT	CTRL
00	none	0	0	0	0	0	0
01	ESC	C('[')	C('[')	CA('[')	CA('[')	CA('[')	C('[')
02	'1'	'1'	'!'	A('1')	A('1')	A('!')	C('A')
13	'='	'='	'+'	A('=')	A('=')	A('+')	C('@')
16	'q'	L('q')	'Q'	A('q')	A('q')	A('Q')	C('Q')
28	CR/LF	C('M')	C('M')	CA('M')	CA('M')	CA('M')	C('J')
29	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL
59	F1	F1	SF1	AF1	AF1	ASF1	CF1
127	???	0	0	0	0	0	0

Figure 3-41. A few entries from a keymap source file.

bit in the high byte of the 16-bit value. This is a flag that indicates that the state of the capslock variable has to be checked and the code possibly modified before being returned. In the figure, the entries for scan codes 2, 13, and 16 show how typical numeric, punctuation, and alphabetic keys are handled. For code 28 a special feature is seen—normally the ENTER key produces a CR (0x0D) code, represented here as C('M'). Because the newline character in UNIX files is the LF (Ox0A) code, and it is sometimes necessary to enter this directly, this keyboard map provides for a CTRL-ENTER combination, which produces this code, C('J').

Scan code 29 is one of the modifier codes and must be recognized no matter what other key is pressed, so the CTRL value is returned regardless of any other key that may be pressed. The function keys do not return ordinary ASCII values, and the row for scan code 59 shows symbolically the values (defined in *include/minix/keymap.h*) that are returned for the F1 key in combination with other modifiers. These values are F1: 0x0110, SF1: 0x1010, AF1: 0x0810, ASF1: 0x0C10, and CF1: 0x0210. The last entry shown in the figure, for scan code l27, is typical of many entries near the end of the array. For many keyboards, certainly most of those used in Europe and the Americas, there are not enough keys to generate all the possible codes, and these entries in the table are filled with zeroes.

Loadable Fonts

Early PCs had the patterns for generating characters on a video screen stored only in ROM, but the displays used on modern systems provide RAM on the video display adapters into which custom character generator patterns can be loaded. This is supported by MINIX with a

```
ioctl(0, TIOCSFON, font)
```

IOCTL operation. MINIX supports an 80 lines $\times$ 25 rows video mode, and font files contain 4096 bytes. Each byte represents a line of 8 pixels that are illuminated if the bit value is 1, and 16 such lines are needed to map each character. However the video display adapter uses 32 bytes to map each character, to provide higher resolution in modes not currently supported by MINIX. The *loadfont* command is provided to convert these files into the 8192-byte *font* structure referenced by the IOCTL call and to use that call to load

the font. As with the keymaps, a font can be loaded at startup time, or at any time during normal operation. However, every video adapter has a standard font built into its ROM that is available by default. There is no need to compile a font into MINIX itself, and the only font support necessary in the kernel is the code to carry out the *TIOCSFON* IOCTL operation.

3.9.4 Implementation of the Device-Independent Terminal Driver

In this section we will begin to look at the source code of the terminal driver in detail. We saw when we studied the block devices that multiple tasks supporting several different devices could share a common base of software. The case with the terminal devices is similar, but with the difference that there is one terminal task that supports several different kinds of terminal device. Here we will start with the device-independent code. In later sections we will look at the device-dependent code for the keyboard and the memory-mapped console display.

Terminal Task Data Structures

The file *tty.h* contains definitions used by the C files which implement the terminal drivers. Most of the variables declared in this file are identified by the prefix *tty_*. There is also one such variable declared in *glo.h* as *EXTERN*. This is *tty_timeout*, which is used by both the clock and terminal interrupt handlers.

Within *tty.h*, the definitions of the *O_NOCTTY* and *O_NONBLOCK* flags (which are optional arguments to the OPEN call) are duplicates of definitions in *include/fcntl.h* but they are repeated here so as not to require including another file. The *devfun_t* and *devfunarg_t* types (lines 11611 and 11612) are used to define pointers to functions, in order to provide for indirect calls using a mechanism similar to what we saw in the code for the main loop of the disk drivers.

The most important definition in *tty.h* is the *tty* structure (lines 11614 to 11668). There is one such structure for each terminal device (the console display and keyboard together count as a single terminal). The first variable in the *tty* structure, *tty_events*, is the flag that is set when an interrupt causes a change that requires the terminal task to attend to the device. When this flag is raised, the global variable *tty_timeout* is also manipulated to tell the clock interrupt handler to awaken the terminal task on the next clock tick.

The rest of the *tty* structure is organized to group together variables that deal with input, output, status, and information about incomplete operations. In the input section, *tty_inhead* and *tty_intail* define the queue where received characters are buffered. *Tty_incount* counts the number of characters in this queue, and *tty_eotct* counts lines or characters, as explained below. All device-specific calls are done indirectly, with the exception of the routines that initialize the terminals, which are called to set up the pointers used for the indirect calls. The *tty_devread* and *tty_icancel* fields hold pointers to device-specific code to perform the read and input cancel operations. *Tty_min* is used in comparisons with *tty_eotct*. When the latter becomes equal to the former, a read operation is complete. During canonical input, *tty_min* is set to 1 and *tty_eotct* counts lines entered. During noncanonical input, *tty_eotct* counts characters and *tty_min* is set from the *MIN* field of the *termios* structure. The comparison of the two variables thus tells when a line is ready or when the minimum character count is reached, depending upon the mode.

Tty_time holds the timer value that determines when the terminal task should be awakened by the clock interrupt handler, and *tty_timenext* is a pointer used to chain the active

tty_time fields together in a linked list. The list is sorted whenever a timer is set, so the clock interrupt handler only has to look at the first entry. MINIX can support many remote terminals, of which only a few may have timers set at any time. The list of active timers makes the job of the clock handler easier than it would be if it had to check each entry in *tty_table*.

Since queueing of output is handled by the device-specific code, the output section of *tty* declares no variables and consists entirely of pointers to device-specific functions that write, echo, send a break signal, and cancel output. In the status section the flags *tty_reprint*, *tty_escaped*, and *tty_inhibited* indicate that the last character seen has a special meaning; for instance, when a CTRL-V (LNEXT) character is seen, *tty_escaped* is set to 1 to indicate that any special meaning of the next character is to be ignored.

The next part of the structure holds data about *DEV_READ*, *DEV_WRITE*, and *DEV_IOCTL* operations in progress. There are two processes involved in each of these operations. The server managing the system call (normally FS) is identified in *tty_incaller* (line 11644). The server calls the tty task on behalf of another process that needs to do an I/O operation, and this client is identified in *tty_inproc* (line 11645). As described in Fig. 3-37, during a READ, characters are transferred directly from the terminal task to a buffer within the memory space of the original caller. *Tty_inproc* and *tty_in_vir* locate this buffer. The next two variables, *tty_inleft* and *tty_incum*, count the characters still needed and those already transferred. Similar sets of variables are needed for a WRITE system call. For IOCTL there maybe an immediate transfer of data between the requesting process and the task, so a virtual address is needed, but there is no need for variables to mark the progress of an operation. An IOCTL request may be postponed, for instance, until current output is complete, but when the time is right the request is carried out in a single operation. Finally, the *tty* structure includes some variables that fall into no other category, including pointers to the functions to handle the *DEV_IOCTL* and *DEV_CLOSE* operations at the device level, a POSIX-style *termios* structure, and a *winsize* structure that provides support for window-oriented screen displays. The last part of the structure provides storage for the input queue itself in the array *tty_inbuf*. Note that this is an array of *u16_t*, not of 8-bit *char* characters. Although applications and devices use 8-bit codes for characters, the C language requires the input function *getchar* to work with a larger data type so it can return a symbolic *EOF* value in addition to all 256 possible byte values.

The *tty_table*, an array of *tty* structures, is declared using the *EXTERN* macro (line 11670). There is one element for each terminal enabled by the *NR_CONS*, *NR_RS_LINES*, and *NR_PTYS* definitions in *include/minix/config.h*. For the configuration discussed in this book, two consoles are enabled, but MINIX may be recompiled to add up to 2 serial lines, and up to 64 pseudo terminals.

There is one other *EXTERN* definition in *tty.h*. *Tty_timelist* (line 11690) is a pointer used by the timer to hold the head of the linked list of *tty_time* fields. The *tty.h* header file is included in many files and storage for *tty_table* and *tty_timelist* is allocated during compilation of *table.c*, in the same way as the *EXTERN* variables that are defined in the *glo.h* header file.

At the end of *tty.h* two macros, *buflen* and *bufend*, are defined. These are used frequently in the terminal task code, which does much copying of data into and out of buffers.

The Device-Independent Terminal Driver

The main terminal task and the device-independent supporting functions are all in *tty.c*. Since the task supports many different devices, the minor device numbers must be

used to distinguish which device is being supported on a particular call, and they are defined on lines 11760 to 11764. Following this there are a number of macro definitions. If a device is not initialized, the pointers to that device's device-specific functions will contain zeroes put there by the C compiler. This makes it possible to define the *tty_active* macro (line 11774) which return *FALSE* if a null pointer is found. Of course, the initialization code for a device cannot be accessed indirectly if part of its job is to initialize the pointers that make indirect access possible. On lines 11777 to 11783 are conditional macro definitions to equate initialization calls for RS-232 or pseudo terminal devices to calls to a null function when these devices are not configured. *Do_pty* may be similarly disabled in this section. This makes it possible to omit the code for these devices entirely if it is not needed.

Since there are so many configurable parameters for each terminal, and there may be quite a few terminals on a networked system, a *termios_defaults* structure is declared and initialized with default values (all of which are defined in *include/termios.h*) on lines 11803 to 11310. This structure is copied into the *tty_table* entry for a terminal whenever it is necessary to initialize or reinitialize it. The defaults for the special characters were shown in Fig. 3-33. Figure 3-42 shows the default values for the various flags. On the following line the *winsize_defaults* structure is similarly declared. It is left to be initialized to all zeroes by the C compiler. This is the proper default action; it means "window size is unknown, use */etc/termcap*."

Field	Default values
c_iflag	BRKINT ICRNL IXON IXANY
c_oflag	OPOST ONLCR
c_cflag	CREAD CS8 HUPCL
c_lflag	ISIG IEXTEN ICANON ECHO ECHOE

Figure 3-42. Default termios flag values.

The entry point for the terminal task is *tty_task* (line 11817). Before entering the main loop, a call is made to *tty_init* for each configured terminal (in the loop on line 11826), and then the MINIX startup message is displayed (lines 11829 to 11831). Although the source code shows a call to printf when this code is compiled the macro that converts calls to the printf library routine into calls to printk is in effect. Printk uses a routine called putk within the console driver, so the FS is not involved. This message goes only to the primary console display and cannot be redirected.

The main loop on lines 11833 to 11884 is, in principle, like the main loop of any task—it receives a message, executes a switch on the message type to call the appropriate function, and then generates a return message. However, there are some complications. First, much work is done by low-level interrupt routines, especially in handling terminal input. In the previous section we saw that individual characters from the keyboard are accepted and buffered without sending a message to the terminal task for each character. Thus, before attempting to receive a message, the main loop always sweeps through the entire *tty_table*, inspecting each terminal's *tp->tty_events* flag and calling *handle_events* as necessary (lines 11835 to 11837), to take care of unfinished business. Only when there is nothing demanding immediate attention is a call made to receive. If the message received is from the hardware a continue statement short-circuits the loop, and the check for events is repeated.

Second, this task services several devices. If a received message is from a hardware interrupt, the device or devices that need service are identified by checking the *tp->tty_events* flags. If the interrupt is not a hardware interrupt the *TTY_LINE* field in the message is used to determine which device should respond to the message. The minor device number is decoded by a series of comparisons, by means of which *tp* is pointed to the correct entry in the *tty_table* (lines 11845 to 11864). If the device is a pseudo terminal, *do_pty* (in *pty.c*) is called and the main loop is restarted. In this case *do_pty* generates the reply message. Of course, if pseudo terminals are not enabled, the call to *do_pty* uses the dummy macro defined earlier. One would hope that attempts to access nonexistent devices would not occur, but it is always easier to add another check than to verify that there are no errors elsewhere in the system. In case the device does not exist or is not configured, a reply message with an *ENXIO* error message is generated and, again, control returns to the top of the loop.

The rest of the task resembles what we have seen in the main loop of other tasks, a switch on the message type (lines 11874 to 11883). The appropriate function for the type of request, *do_read*, *do_write*, and so on, is called. In each case the called function generates the reply message, rather than pass the information needed to construct the message back to the main loop. A reply message is generated at the end of the main loop only if a valid message type was not received, in which case an *EINVAL* error message is sent. Because reply messages are sent from many different places within the terminal task a common routine, *tty_reply*, is called to handle the details of constructing reply messages.

If the message received by *tty_task* is a valid message type, not the result of an interrupt, and does not come from a pseudo terminal, the switch at the end of the main loop will dispatch to one of the functions *do_read*, *do_write*, *do_ioctl*, *do_open*, *do_close*, or *do_cancel*. The arguments to each of these calls are *tp*, a pointer to a *tty* structure, and the address of the message. Before looking at each of them, we will mention a few general considerations. Since *tty_task* may service multiple terminal devices, these functions must return quickly so the main loop can continue. However, *do_read*, *do_write*, and *do_ioctl* may not be able to complete immediately all the requested work. In order to allow FS to service other calls, an immediate reply is required. If the request cannot be completed immediately, the *SUSPEND* code is returned in the status field of the reply message. This corresponds to the message marked (3) in Fig. 3-37 and suspends the process that initiated the call, while unblocking the FS. Messages corresponding to (7) and (8) in the figure will be sent later when the operation can be completed. If the request can be fully satisfied, or an error occurs, either the count of bytes transferred or the error code is returned in the status field of the return message to the FS. In this case a message will be sent immediately from the FS back to the process that made the original call, to wake it up.

Reading from a terminal is fundamentally different from reading from a disk device. The disk driver issues a command to the disk hardware and eventually data will be returned, barring a mechanical or electrical failure. The computer can display a prompt upon the screen, but there is no way for it to force a person sitting at the keyboard to start typing. For that matter, there is no guarantee that anybody will be sitting there at all. In order to make the speedy return that is required, *do_read* (line 11891) starts by storing information that will enable the request to be completed later, when and if input arrives. There are a few error checks to be made first. It is an error if the device is still expecting input to fulfill a previous request, or if the parameters in the message are invalid (lines 11901 to 11908). If these tests are passed, information about the request is copied into the proper fields in the device's *tp->tty_table* entry on lines 11911 to 11915. The last step,

setting *tp->tty_inleft* to the number of characters requested, is important. This variable is used to determine when the read request is satisfied. In canonical mode *tp->tty_inleft* is decremented by one for each character returned, until an end of line is received, at which point it is suddenly reduced to zero. In noncanonical mode it is handled differently, but in any case it is reset to zero whenever the call is satisfied, whether by a timeout or by receiving at least the minimum number of bytes requested. When *tp->tty_inleft* reaches zero, a reply message is sent. As we will see, reply messages can be generated in several places. It is sometimes necessary to check whether a reading process still expects a reply; a nonzero value of *tp->tty_inleft* serves as a flag for that purpose.

In canonical mode a terminal device waits for input until either the number of characters asked for in the call has been received, or the end of a line or the end of the file is reached. The *ICANON* bit in the *termios* structure is tested on line 11917 to see if canonical mode is in effect for the terminal. If it is not set, the termios *MIN* and *TIME* values are checked to determine what action to take.

In Fig. 3-35 we saw how *MIN* and *TIME* interact to provide different ways a read call can behave. *TIME* is tested on line 11918. A value of zero corresponds to the left-hand column in Fig. 3-35, and in this case no further tests are needed at this point. If *TIME* is nonzero, then *MIN* is tested. If it is zero, *settimer* is called on to start the timer that will terminate the *DEV_READ* request after a delay, even if no bytes have been received. *Tp->tty_min* is set to 1 here, so the call will terminate immediately if one or more bytes are received before the timeout. At this point no check for possible input has yet been made, so more than one character may already be waiting to satisfy the request. In that case, as many characters as are ready, up to the number specified in the READ call, will be returned as soon as the input is found. If both *TIME* and *MIN* are nonzero, the timer has a different meaning. The timer is used as an inter-character timer in this case. It is started only after the first character is received and is restarted after each successive character. *Tp->tty_eotct* counts characters in noncanonical mode, and if it is zero at line 11931, no characters have been received yet and the inter-byte timer is inhibited. *Lock* and *unlock* are used to protect both of these calls to *settimer*, to prevent clock interrupts when *settimer* is running.

In any case, at line 11941, *in_transfer* is called to transfer any bytes already in the input queue directly to the reading process. Next there is a call to *handle_events*, which may put more data into the input queue and which calls *in_transfer* again. This apparent duplication of calls requires some explanation. Although the discussion so far has been in terms of keyboard input, *do_read* is in the device-independent part of the code and also services input from remote terminals connected by serial lines. It is possible that previous input has filled the RS-232 input buffer to the point where input has been inhibited. The first call to *in_transfer* does not start the flow again, but the call to *handle_events* can have this effect. The fact that it then causes a second call to *in_transfer* is just a bonus. The important thing is to be sure the remote terminal is allowed to send again. Either of these calls may result in satisfaction of the request and sending of the reply message to the FS. *Tp->tty_inleft* is used as a flag to see if the reply has been sent; if it is still nonzero at line 11944, *do_read* generates and sends the reply message itself. This is done on lines 11949 to 11957. If the original request specified a nonblocking read, the FS is told to pass an *EAGAIN* error code back to original caller. If the call is an ordinary blocking read, the FS receives a *SUSPEND* code, unblocking it but telling it to leave the original caller blocked. In this case the terminal's *tp->tty_inrepcode* field is set to *REVIVE*. When and if the READ is later satisfied, this code will be placed in the reply message to the FS to indicate that the original caller was put to sleep and needs to be revived.

Do_write (line 11964) is similar to *do_read*, but simpler, because there are fewer options to be concerned about in handling a WRITE system call. Checks similar to those made by *do_read* are made to see that a previous write is not still in progress and that the message parameters are valid, and then the parameters of the request are copied into the *tty* structure. *Handle_events* is then called, and *tp->tty_outleft* is checked to see if the work was done (lines 11991 and 11992). If so, a reply message already has been sent by *handle_events* and there is nothing left to do. If not, a reply message is generated. with the message parameters depending upon whether or not the original WRITE call was called in nonblocking mode.

POSIX function	POSIX operation	IOCTL type	IOCTL parameter
tcdrain	(none)	TCDRAIN	(none)
tcflow	TCOOFF	TCFLOW	int=TCOOFF
tcflow	TCOON	TCFLOW	int=TCOON
tcflow	TCIOFF	TCFLOW	int=TCIOFF
tcflow	TCION	TCFLOW	int=TCION
tcflush	TCIFLUSH	TCFLSH	int=TCIFLUSH
tcflush	TCOFLUSH	TCFLSH	int=TCOFLUSH
tcflush	TCIOFLUSH	TCFLSH	int=TCIOFLUSH
tcgetattr	(none)	TCGETS	termios
tcsetattr	TCSANOW	TCSETS	termios
tcsetattr	TCSADRAIN	TCSETSW	termios
tcsetattr	TCSAFLUSH	TCSETSF	termios
tcsendbreak	(none)	TCSBRK	int=duration

Figure 3-43. POSIX calls and IOCTL operations.

The next function, *do_ioctl* (line 12012), is a long one, but not difficult to understand. The body of *do_ioctl* is two switch statements. The first determines the size of the parameter pointed to by the pointer in the request message (lines 12033 to 12064). If the size is not zero, the parameter's validity is tested. The contents cannot be tested here, but what can be tested is whether a structure of the required size beginning at the specified address fits within the segment it is specified to be in. The rest of the function is another switch on the type of IOCTL operation requested (lines 12075 to 12161). Unfortunately, supporting the POSIX-required operations with the IOCTL call meant that names for IOCTL operations had to be invented that suggest, but do not duplicate, names required by POSIX. Figure 3-43 shows the relationship between the POSIX request names and the names used by the MINIX IOCTL call. A *TCGETS* operation services a *tcgetattr* call by the user and simply returns a copy of the terminal device's *tp->tty_termios* structure. The next four request types share code. The *TCSETSW*, *TCSETSF*, and *TCSETS* request types correspond to user calls to the POSIX-defined function tcsetattr, and all have the basic action of copying a new *termios* structure into a terminal's *tty* structure. The copying is done immediately for *TCSETS* calls and may be done for *TCSETSW* and *TCSETSF* calls if output is complete, by a *phys_copy* call to get the data from the user, followed by a call to *setattr*, on lines 12098 and 12099. If *tcsetattr* was called with a modifier

requesting postponement of the action until completion of current output, the parameters for the request are placed in the terminal's *tty* structure for later processing if the test of *tp->tty_outleft* on line 12084 reveals output is not complete. *Tcdrain* suspends a program until output is complete and is translated into an IOCTL call of type *TCDRAIN*. If output is already complete, it has nothing more to do. If not, it also must leave information in the *tty* structure.

The POSIX *tcflush* function discards unread input and/or unsent output data, according to its argument, and the IOCTL translation is straightforward, consisting of a call to the *tty_icancel* function that services all terminals, and/or the device-specific function pointed to by *tp->tty_ocancel* (lines 12102 to 12109). *Tcflow* is similarly translated in a straightforward way into an IOCTL call. To suspend or restart output, it sets a *TRUE* or *FALSE* value into *tp->tty_inhibited* and then sets the *tp->tty_events* flag. To suspend or restart input, it sends the appropriate *STOP* (normally CTRL-S) or *START* (CTRL-Q) code to the remote terminal, using the device-specific echo routine pointed to by *tp->tty_echo* (lines 12120 to 12125).

Most of the rest of the operations handled by *do_ioctl* are handled in one line of code, by calling an appropriate function. In the cases of the *KIOCSMAP* (load keymap) and *TIOCSFON* (load font) operations, a test is made to be sure the device really is a console, since these operations do not apply to other terminals. If virtual terminals are in use the same keymap and font apply to all consoles, the hardware does not permit any easy way of doing otherwise. The window size operations copy a *winsize* structure between the user process and the terminal task. Note, however, the comment under the code for the *TIOCSWINSZ* operation. When a process changes its window size, the kernel is expected to send a *SIGWINCH* signal to the process group under some versions of UNIX. The signal is not required by the POSIX standard. But, anyone thinking of using these structures should consider adding code here to initiate this signal.

The last two cases in *do_ioctl* support the POSIX required *tcgetpgrp* and *rcsetpgrp* functions. There is no action associated with these cases, and they always return an error. There is nothing wrong with this. These functions support job control, the ability to suspend and restart a process from the keyboard. job control is not required by POSIX and is not supported by MINIX. However, POSIX requires these functions, even when job control is not supported, to ensure portability of programs.

Do_open (line 12171) has a simple basic action to perform—it increments the variable *tp->tty_openct* for the device so it can be verified that it is open. However, there are some tests to be done first. POSIX specifies that for ordinary terminals the first process to open a terminal is the **session leader**, and when a session leader dies, access to the terminal is revoked from other processes in its group. Daemons need to be able to write error messages, and if their error output is not redirected to a file, it should go to a display that cannot be closed. For this purpose a device called */dev/log* exists in MINIX. Physically it is the same device as */dev/console*, but it is addressed by a separate minor device number and is treated differently. It is a write-only device, and thus *do_open* returns an *EACCESS* error if an attempt is made to open it for reading (line 12183). The other test done by *do_open* is to test the *O_NOCTTY* flag. If it is not set and the device is not */dev/log*, the terminal becomes the controlling terminal for a process group. This is done by putting the process number of the caller into the *tp->tty_pgrp* field of the *tty_table* entry. Following this, the *tp->tty_openct* variable is incremented and the reply message is sent.

A terminal device may be opened more than once, and the next function, *do_close* (line 12198), has nothing to do except decrement *tp->tty_openct*. The test on line 12204 foils an attempt to close the device if it happens to be */dev/log*. If this operation is the last

close, input is canceled by calling *tp->tc_icancel*. Device-specific routines pointed to by *tp->tty_ocancel* and *tp->tty_close* are also called. Then various fields in the *tty* structure for the device are set back to their default values and the reply message is sent.

The last message type handler is *do_cancel* (line 12220). This is invoked when a signal is received for a process that is blocked trying to read or write. There are three states that must be checked:

1. The process may have been reading when killed.

2. The process may have been writing when killed.

3. The process may have been suspended by tcdrain until its output was complete.

A test is made for each case, and the general *tp->tty_icancel*, or the device-specific routine pointed to by *tp->tty_ocancel*, is called as necessary. In the last case the only action required is to reset the flag *tp->tty_ioreq*, to indicate the IOCTL operation is now complete. Finally, the *tp->tty_events* flag is set and a reply message is sent.

Terminal Driver Support Code

Now that we have looked at the top-level functions called in the main loop of *tty_task*, it is time to look at the code that supports them. We will start with *handle_events* (line 12256). As mentioned earlier, on each pass through the main loop of the terminal task, the *tp->tty_events* flag for each terminal device is checked and *handle_events* is called if it shows that attention is required for a particular terminal. *Do_read* and *do_write* also call *handle_events*. This routine must work fast. It resets the *tp->tty_events* flag and then calls device-specific routines to read and write, using the pointers to the functions *tp->tty_devread* and *tp->tty_devwrite* (lines 12279 to 12282). These are called unconditionally, because there is no way to test whether a read or a write caused the raising of the flag—a design choice was made here, that checking two flags for each device would be more expensive than making two calls each time a device was active. Also, most of the time a character received from a terminal must be echoed, so both calls will be necessary. As noted in the discussion of the handling of *tcsetattr* calls by *do_ioctl*, POSIX may postpone control operations on devices until current output is complete, so immediately after calling the device-specific *tty_devwrite* function is a good time take care of ioctl operations. This is done on line 12285, where *dev_ioctl* is called if there is a pending control request.

Since the *tp->tty_events* flag is raised by interrupts, and characters may arrive in a rapid stream from a fast device, there is a chance that by the time the calls to the device-specific read and write routines and *dev_ioctl* are completed, another interrupt will have raised the flag again. A high priority is placed on getting input moved along from the buffer where the interrupt routine places it initially. Thus *handle_events* repeats the calls to the device-specific routines as long as the tp->*tty_events* flag is found raised at the end of the loop (line 12286). When the flow of input stops (it also could be output, but input is more likely to make such repeated demands), *in_transfer* is called to transfer characters from the input queue to the buffer within the process that called for a read operation. *In_transfer* itself sends a reply message if the transfer completes the request, either by transferring the maximum number of characters requested or by reaching the end of a line (in canonical mode). If it does so, *tp->tty_left* will be zero upon the return to *handle_events*. Here a further test is made and a reply message is sent if the number of

characters transferred has reached the minimum number requested. Testing *tp->tty_inleft* prevents sending a duplicate message.

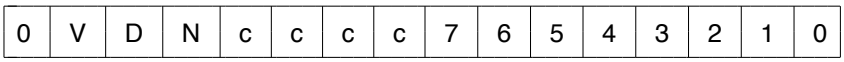
Next we will look at *in_transfer* (line 12303), which is responsible for moving data from the input queue in the task's memory space to the buffer of the user process that requested the input. However, a straightforward block copy is not possible. The input queue is a circular buffer and characters have to be checked to see that the end of the file has not been reached, or, if canonical mode is in effect, that the transfer only continues up through the end of a line. Also, the input queue is a queue of 16-bit quantities, but the recipient's buffer is an array of 8-bit characters. Thus an intermediate local buffer is used. Characters are checked one by one as they are placed in the local buffer, and when it fills up or when the input queue has been emptied, *phys_copy* is called to move the contents of the local buffer to the receiving process' buffer (lines 12319 to 12345).

Three variables in the *tty* structure, *tp->tty_inleft*, *tp->tty_eotct*, and *tp->tty_min*, are used to decide whether *in_transfer* has any work to do, and the first two of these control its main loop. As mentioned earlier, *tp->tty_inleft* is set initially to the number of characters requested by a READ call. Normally, it is decremented by one whenever a character is transferred out it may be abruptly decreased to zero when a condition signaling the end of input is reached. Whenever it becomes zero, a reply message to the reader is generated, so it also serves as a flag to indicate whether or not a message has been sent. Thus in the test on line 12314, finding that *tp->tty_inleft* is already zero is a sufficient reason to abort execution of *in_transfer* without sending a reply.

In the next part of the test, *tp->tty_eotct* and *tp->tty_min* are compared. In canonical mode both of these variables refer to complete lines of input, and in noncanonical mode they refer to characters. *Tp->tty_eotct* is incremented whenever a "line break" or a byte is placed in the input queue and is decremented by *in_transfer* whenever a line or byte is removed from the queue. Thus it counts the number of lines or bytes that have been received by the terminal task but not yet passed on to a reader. *Tp->tty_min* indicates the minimum number of lines (in canonical mode) or characters (in noncanonical mode) that must be transferred to complete a read request. Its value is always 1 in canonical mode and may be any value from 0 up to *MAX_INPUT* (255 in MINIX) in noncanonical mode. The second half of the test on line 12314 causes *in_transfer* to return immediately in canonical mode if a full line has not yet been received. The transfer is not done until a line is complete so the queue contents can be modified if, for instance, an ERASE or KILL character is subsequently typed in by the user before the ENTER key is pressed. In noncanonical mode an immediate return occurs if the minimum number of characters is not yet available.

A few lines later, *tp->tty_inleft* and *tp->tty_eotct* are used to control the main loop of *in_transfer*. In canonical mode the transfer continues until there is no longer a complete line left in the queue. In noncanonical mode *tp->tty_eotct* is a count of pending characters. *Tp->tty_min* controls whether the loop is entered but is not used in determining when to stop. Once the loop is entered, either all available characters or the number of characters requested in the original call will be transferred, whichever is smaller.

Characters are 16-bit quantities in the input queue. The actual character code to be transferred to the user process is in the low 8 bits. Fig. 3-44 shows how the high bits are used. Three are used to flag whether the character is being escaped (by CTRL-V), whether it signifies end-of-file, or whether it represents one of several codes that signify a line is complete. Four bits are used for a count to show how much screen space is used when the character is echoed. The test on line 12322 checks whether the *IN_EOF* bit (*D* in the figure) is set. This is tested at the top of the inner loop because an end-of-file



V: IN_ESC, escaped by LNEXT (CTRL-V)
D: IN_EOF, end of file (CTRL-D)
N: IN_EOT, line break (NL and others)
cccc: count of characters echoed
7: Bit 7, may be zeroed if ISTRIP is set
6-0: Bits 0-6, ASCII code

Figure 3-44. The fields in a character code as it is placed into the input queue.

(CTRL-D) is not itself transferred to a reader, nor is it counted in the character count. As each character is transferred, a mask is applied to zero the upper 8 bits, and only the ASCII value in the low 8 bits is transferred into the local buffer (line 12324).

There is more than one way to signal the end of input, but the device-specific input routine is expected to determine whether a character received is a linefeed, CTRL-D, or other such character and to mark each such character. *In_transfer* only needs to test for this mark, the *IN_EOT* bit (*N* in Fig. 3-44), on line 12340. if this is detected, *tp->tty_eotct* is decremented. In noncanonical mode every character is counted this way as it is put into the input queue, and every character is also marked with the *IN_EOT* bit at that time, so *tp->tty_eotct* counts characters not yet removed from the queue. The only difference in the operation of the main loop of *in_transfer* in the two different modes is found on line 12343. Here *tp->tty_inleft* is zeroed in response to finding a character marked as a line break, but only if canonical mode is in effect. Thus when control returns to the top of the loop. the loop terminates properly after a line break in canonical mode, but in noncanonical line breaks are ignored.

When the loop terminates there is usually a partially full local buffer to be transferred (lines l2347 to 12353). Then a reply message is sent if *tp->tty_inleft* has reached zero. This is always the case in canonical mode, but if noncanonical mode is in effect and the number of characters transferred is less than the full request, the reply is not sent. This may be puzzling if you have a good enough memory for details to remember that where we have seen calls to *in_transfer* (in *do_read* and *handle_events*). the code following the call to *in_transfer* sends a reply message if *in_transfer* returns having transferred more than the amount specified in *tp->tty_min*, which will certainly be the case here. The reason why a reply is not made unconditionally from *in_transfer* will be seen when we discuss the next function, which calls *in_transfer* under a different set of circumstances.

That next function is *in_process* (line 12367). It is called from the device-specific software to handle the common processing that must be done on all input. Its parameters are a pointer to the *tty* structure for the source device, a pointer to the array of 8-bit characters to be processed, and a count. The count is returned to the caller. *In_process* is a long function, but its actions are not complicated. It adds l6-bit characters to the input queue that is later processed by *in_transfer*.

There are several categories of treatment provided by *in_transfer*.

1. Normal characters are added to the input queue, extended to 16 bits.

2. Characters which affect later processing modify flags to signal the effect but are not placed in the queue.

3. Characters which control echoing are acted upon immediately without being placed in the queue.

4. Characters with special significance have codes such as the EOT bit added to their high byte as they are placed in the input queue.

Let us look first at a completely normal situation, an ordinary character, such as "x" (ASCII code Ox78), typed in the middle of a short line, with no escape sequence in effect, on a terminal that is set up with the standard MINIX default properties. As received from the input device this character occupies bits 0 through 7 in Fig. 3-44. On line 12385 it would have its most significant bit, bit 7, reset to zero if the *ISTRIP* bit were set, but the default in MINIX is not to strip the bit, allowing full 8-bit codes to be entered. This would not affect our "x" anyway. The MINIX default is to allow extended processing of input, so the test of the *IEXTEN* bit in tp->*tty_termios.c_lflag* (line 12388) passes, but the succeeding tests fail under the conditions we postulate: no character escape is in effect (line 12391), this input is not itself the character escape character (line 12397), and this input is not the *REPRINT* character (line 12405).

Tests on the next several lines find that the input character is not the special *_POSIX_VDISABLE* character, nor is it a *CR* or an *NL*. Finally, a positive result: canonical mode is in effect, this is the normal default (line 12424). However our "x" is not the *ERASE* character, nor is it any of the *KILL*, *EOF* (CTRL-D), *NL*, or *EOL* characters, so by line 12457 still nothing will have happened to it. Here it is found that the *IXON* bit is set, by default, allowing use of the *STOP* (CTRL-S) and *START* (CTRL-Q) characters, but in the ensuing tests for these no match is found. On line 12478 it is found that the *ISIG* bit, enabling the use of the *INTR* and *QUIT* characters, is set by default, but again no match is found.

In fact, the first interesting thing that might happen to an ordinary character occurs on line 12491; where a test is made to see if the input queue is already full. If this were the case, the character would be discarded at this point, since canonical mode is in effect, and the user would not see it echoed on the screen. (The continue statement discards the character, since it causes the outer loop to restart). However, since we postulate completely normal conditions for this illustration, let us assume the buffer is not full yet. The next test, to see if special noncanonical mode processing is needed (line 12497), fails, causing a jump forward to line 12512. Here *echo* is called to display the character to the user, since the *ECHO* bit in *tp->tty_termios.c_lflag* is set by default.

Finally, on lines 12515 to 12519 the character is disposed of by being put into the input queue. At this time *tp->tty_incount* is incremented. but since this is an ordinary character, not marked by the *EOT* bit. *tp->tty_eotct* is not changed.

The last line in the loop calls *in_transfer* if the character just transferred into the queue fills it. However, under the ordinary conditions we postulate for this example. *in_transfer* would do nothing, even if called, since (assuming the queue has been serviced normally and previous input was accepted when the previous line of input was complete) *tp->tty_eotct* is zero. *tp->tty_min* is one, and the test at the start of *in_transfer* (line 12314) causes an immediate return.

Having passed through *in_process* with an ordinary character under ordinary conditions, let us now go back to the start of *in_process* and look at what happens in less ordinary circumstances. First, we will look at the character escape, which allows a character which ordinarily has a special effect to be passed on to the user process. If a character escape is in effect, the *tp->tty_escaped* flag is set, and when this is detected (on line 12391) the flag is reset immediately and the *IN_ESC* bit, bit V in Fig. 3-44, is added to the current character. This causes special processing when the character is echoed—escaped control

characters are displayed as "^" plus the character to make them visible. The *IN_ESC* bit also prevents the character from being recognized by tests for special characters. The next few lines process the escape character itself, the *LNEXT* character (CTRL-V by default). When the *LNEXT* code is detected the tp->*tty_escaped* flag is set, and *rawecho* is called twice to output a "^" followed by a backspace. This reminds the user at the keyboard that an escape is in effect, and when the following character is echoed, it overwrites the "^". The *LNEXT* character is an example of one that affects later characters (in this case, only the very next character). It is not placed in the queue, and the loop restarts after the two calls to *rawecho*. The order of these two tests is important, making it possible to enter the *LNEXT* character itself twice in a row, in order to pass the second copy on to a process.

The next special character processed by *in_process* is the *REPRINT* character (CTRL-R). When it is found a call to reprint ensues (line 12406), causing the current echoed output to be redisplayed. The *REPRINT* itself is then discarded with no effect upon the input queue.

Going into detail on the handling of every special character would be tedious, and the source code of *in_process* is straightforward. We will mention just a few more points. One is that the use of special bits in the high byte of the 16-bit value placed in the input queue makes it easy to identify a class of characters that have similar effects. Thus, *EOF* (CTRL-D), *LF*, and the alternate *EOL* character (undefined by default) are all marked by the EOT bit, bit D in Fig. 3-44 (lines 12447 to 12453), making later recognition easy. Finally, we will justify the peculiar behavior of *in_transfer* noted earlier. A reply is not generated each time it terminates, although in the calls to *in_transfer* we have seen previously, it seemed that a reply would always be generated upon return. Recall that the call to *in_transfer* made by *in_process* when the input queue is full (line 12522) has no effect when canonical mode is in effect. But if noncanonical processing is desired, every character is marked with the *EOT* bit on line 12499, and thus every character is counted by *tp->tty_eotct* on line 12519. In turn, this causes entry into the main loop of *in_transfer* when it is called because of a full input queue in noncanonical mode. On such occasions no message should be sent at the termination of *in_transfer*, because there are likely to be more characters read after returning to *in_process*. Indeed, although in canonical mode input to a single READ is limited by the size of the input queue (255 characters in MINIX), in noncanonical mode a READ call must be able to deliver the POSIX-required *_POSIX_SSIZE_MAX* number of characters. Its value in MINIX is 32767.

The next few functions in *tty.c* support character input. *Echo* (line 12531) treats a few characters in a special way, but most just get displayed on the output side of the same device being used for input. Output from a process may be going to a device at the same time input is being echoed, which makes things messy if the user at the keyboard tries to backspace. To deal with this, the *tp->tty_reprint* flag is always set to *TRUE* by the device-specific output routines when normal output is produced, so the function called to handle a backspace can tell that mixed output has been produced. Since *echo* also uses the device-output routines, the current value of *tp->tty_reprint* is preserved while echoing, using the local variable *rp* (lines 12552 to 12585). However, if a new line of input has just begun, *rp* is set to *FALSE* instead of taking on the old value, thus assuring that *tp->tty_reprint* will be reset when echo terminates.

You may have noticed that *echo* returns a value, for instance, in the call on line 12512 in *in_process*:

```
ch = echo(tp, ch)
```

The value returned by *echo* contains the number of spaces used on the screen for the echo

display, which may be up to eight if the character is a *TAB*. This count is placed in the *cccc* field in Fig. 3-44. Ordinary characters occupy one space on the screen, but if a control character (other than *TAB*, *NL*, or *CR* or a *DEL* (0x7F) is echoed, it is displayed as a "^" plus a printable ASCII character and occupies two positions on the screen. On the other hand an *NL* or *CR* occupies zero spaces. The actual echoing must be done by a device-specific routine, of course, and whenever a character must be passed to the device, an indirect call is made using *tp->tty_echo*, as, for instance, on line 12580, for ordinary characters.

The next function, *rawecho*, is used to bypass the special handling done by *echo*. It checks to see if the *ECHO* flag is set, and if it is, sends the character along to the device-specific *tp->tty_echo* routine without any special processing. A local variable *rp* is used here to prevent *rawecho*'s own call to the output routine from changing the value of *tp->tty_reprint*.

When a backspace is found by *in_process*, the next function, *backover* (line 12607), is called. It manipulates the input queue to remove the previous head of the queue if backing up is possible—if the queue is empty or if the last character is a line break, then backing up is not possible. Here the *tp->tty_reprint* flag mentioned in the discussions of *echo* and *rawecho* is tested. If it is *TRUE*, then *reprint* is called (line 12618) to put a clean copy of the output line on the screen. Then the *len* field of the last character displayed (the *cccc* field of Fig. 3-44) is consulted to find out how many characters have to be deleted on the display, and for each character a sequence of backspace-space-backspace characters is sent through *rawecho* to remove the unwanted character from the screen.

Reprint is the next function. In addition to being called by *backover*, it may be invoked by the user pressing the *REPRINT* key (CTRI-R). The loop on lines 12651 to 12656 searches backward through the input queue for the last line break. If it is found in the last position filled, there is nothing to do and *reprint* returns. Otherwise, it echos the CTRL-R, which appears on the display as the two character sequence "^R", and then moves to the next line and redisplays the queue from the last line break to the end.

Now we have arrived at *out_process* (line 12677). Like *in_process*, it is called by device-specific output routines, but it is simpler. It is called by the RS232 and pseudo terminal device-specific output routines, but not by the console routine. *Out_process* works upon a circular buffer of bytes but does not remove them from the buffer. The only change it makes to the array is to insert a *CR* character ahead of an *NL* character in the buffer if the *OPOST* (enable output processing) and *ONLCR* (map NL to CR-NL) bits in *tp->tty_termios.oflag* are set. Both bits are set by default in MINIX. Its job is to keep the *tp->tty_position* variable in the device's *tty* structure up to date. Tabs and backspaces complicate life.

The next routine is *dev_ioctl* (line 12763). It supports *do_ioctl* in carrying out the *tcdrain* function and the *tcsetattr* function when it is called with either the *TCSADRAIN* or *TCSAFLUSH* options. In these cases, *do_ioctl* cannot complete the action immediately if output is incomplete, so information about the request is stored in the parts of the *tty* structure reserved for delayed IOCTL operations. Whenever *handle_events* runs, it checks the *tp->tty_ioreq* field after calling the device-specific output routine and calls *dev_ioctl* if an operation is pending. *Dev_ioctl* tests *tp->tty_outleft* to see if output is complete, and if so, carries out the same actions that *do_ioctl* would have carried out immediately if there had been no delay. To service *tcdrain*, the only action is to reset the *tp->tty_ioreq* field and send the reply message to the FS, telling it to wake up the process that made the original call. The *TCSAFLUSH* variant of *tcsetattr* calls *tty_icancel* to cancel input. For both variants of *tcsetattr*, the *termios* structure whose address was

passed in the original call to IOCTL is copied to the device's *tp->tty_termios* structure. *Setattr* is then called, followed, as with *tcdrain*, by sending a reply message to wake up the blocked original caller.

Setattr (line 12789) is the next procedure. As we have seen, it is called by *do_ioctl* or *dev_ioctl* to change the attributes of a terminal device, and by *do_close* to reset the attributes back to the default settings. *Setattr* is always called after copying a new *termios* structure into a device's *tty* structure, because merely copying the parameters is not enough. If the device being controlled is now in noncanonical mode, the first action is to mark all characters currently in the input queue with the *IN_EOT* bit, as would have been done when these characters were originally entered in the queue if noncanonical mode had been in effect then. It is easier just to go ahead and do this (lines 12803 to 12809) than to test whether the characters already have the bit set. There is no way to know which attributes have just been changed and which still retain their old values.

The next action is to check the *MIN* and *TIME* values. In canonical mode *tp->tty_min* is always 1; that is set on line 12818. In noncanonical mode the combination of the two values allows for four different modes of operation, as we saw in Fig. 3-35. On lines 12823 to 12825 *tp->tty_min* is first set up with the value passed in *tp->tty_termiso.cc*[*VMIN*], which is then modified if it is zero and *tp->tty_termiso.cc*[*VTIME*] is not zero.

Finally, *setattr* makes sure output is not stopped if XON/XOFF control is disabled, sends a *SIGHUP* signal if the output speed is set to zero, and makes an indirect call to the device-specific routine pointed to by *tp->tty_ioctl* to do what can only be done at the device level.

The next function, *tty_reply* (line 12845) has been mentioned many times in the preceding discussion. Its action is entirely straightforward, constructing a message and sending it. If for some reason the reply fails, a panic ensues. The following functions are equally simple. *Sigchar* (line 12866) asks MM to send a signal. If the *NOFLSH* flag is not set, queued input is removed—the count of characters or lines received is zeroed and the pointers to the tail and head of the queue are equated. This is the default action. When a *SIGHUP* signal is to be caught, *NOFLSH* can be set, to allow input and output to resume after catching the signal. *Tty_icancel* (line 12891) unconditionally discards pending input in the way described for *sigchar*, and in addition calls the device-specific function pointed to by *tp->tty_icancel*, to cancel input that may exist in the device itself or be buffered in the low-level code.

Tty_init (line 12905) is called once for each device when *tty_task* first starts. It sets up defaults. Initially a pointer to *tty_devnop*, a dummy function that does nothing, is set into the *tp->tty_icancel*, *tp->tty_ocancel*, *tp->tty_ioctl*, and *tp->tty_close* variables. *Tty_init* then calls a device-specific initialization functions for the appropriate category of terminal (console, serial line, or pseudo terminal). These set up the real pointers to indirectly called device-specific functions. Recall that if there are no devices at all configured in a particular category, a macro that returns immediately is created, so no part of the code for a nonconfigured device need be compiled. The call to *scr_init* initializes the console driver and also calls the initialization routine for the keyboard.

Tty_wakeup (line 12929), although short, is extremely important in the functioning of the terminal task. Whenever the clock interrupt handler runs, that is to say, for every tick of the clock, the global variable *tty_timeout* (defined in *glo.h* on line 5032), is checked to see if it contains a value less than the present time. If so *tty_wakeup* is called. *Tty_timeout* is set to zero by the interrupt service routines for terminal drivers, so wakeup is forced to run at the next clock tick after any terminal device interrupt. *Tty_timeout* is also altered by *settimer* when a terminal device is servicing a READ call in noncanonical

mode and needs to set a timeout, as we will see shortly. When *tty_wakeup* runs, it first disables the next wakeup by assigning *TIME_NEVER*, a value very far in the future, to *tty_timeout*. Then it scans the linked list of timer values, which is sorted with the earliest scheduled wakeups first, until it comes to one that is later than the current time. This is the next wakeup, and it is then put into *tty_timeout*. *Tty_wakeup* also sets *tp->tty_min* for that device to 0, which ensures that the next read will succeed even if no bytes have been received, sets the *tp->tty_events* flag for the device to ensure it gets attention when the terminal task runs next, and removes the device from the timer list. Finally, it calls *interrupt* to send the wakeup message to the task. As mentioned in the discussion of the clock task, *tty_wakeup* is logically part of the clock interrupt service code, since it is called only from there.

The next function, *settimer* (line 12958), sets timers for determining when to return from a READ call in noncanonical mode. It is called with parameters of *tp*, a pointer to a *tty* structure, and *on*, an integer which represents *TRUE* or *FALSE*. First the linked list of *tty* structures pointed to by *timelist* is scanned, searching for an existing entry that matches the *tp* parameter. If one is found, it is removed from the list (lines 12968 to 12973). If *settimer* is called to unset a timer, this is all it must do. If it is called to set a timer, the *tp->tty_time* element in the *tty* structure of the device is set to the current time plus the increment in tenths of a second specified in the *TIME* value in the device's *termios* structure. Then the entry is put into the list, which is maintained in sorted order. Finally, the timeout just entered on the list is compared with the value in the global *tty_timeout*, and the latter is replaced if the new timeout is due sooner.

Finally, the last definition in *tty.c* is *tty_devnop* (line 12992), a "no-operation" function to be indirectly addressed where a device does not require a service. We have seen *tty_devnop* used in *tty_init* as the default value entered into various function pointers before calling the initialization routine for a device.

3.9.5 Implementation of the Keyboard Driver

Now we turn to the device-dependent code that supports the MINIX console, which consists of an IBM PC keyboard and a memory-mapped display. The physical devices that support these are entirely separate: on a standard desktop system the display uses an adapter card (of which there are at least a half-dozen basic types) plugged into the backplane, while the keyboard is supported by circuitry built into the parentboard which interfaces with an 8-bit single-chip computer inside the keyboard unit. The two subdevices require entirely separate software support, which is found in the files *keyboard.c* and *console.c*.

The operating system sees the keyboard and console as parts of the same device, */dev/console*. If there is enough memory available on the display adapter, **virtual console** support may be compiled, and in addition to */dev/console* there may be additional logical devices, */dev/ttyc1*, */dev/ttyc2*, and so on. Output from only one goes to the display at any given time, and there is only one keyboard to use for input to whichever console is active. Logically the keyboard is subservient to the console, but this is manifested in only two relatively minor ways. First, *tty_table* contains a *tty* structure for the console, and where separate fields are provided for input and output, for instance, the *tty_devread* and *tty_devwrite* fields, pointers to functions in *keyboard.c* and *console.c* are filled in at startup time. However, there is only one *tty_priv* field, and this points to the console's data structures only. Second, before entering its main loop, *tty_task* calls each logical device once to initialize it. The routine called for */dev/console* is in *console.c*, and the

initialization code for the keyboard is called from there. The implied hierarchy could just as well have been reversed, however. We have always looked at input before output in dealing with I/O devices and we will continue that pattern, discussing *keyboard.c* in this section and leaving the discussion of *console.c* for the following section.

Keyboard.c begins, like most source files we have seen, with several `#include` statements. One of these is unusual, however. The file *keymaps/us-std.src* (included on line 13014) is not an ordinary header; it is a C source file that results in compilation of the default keymap within *keyboard.o* as an initialized array. The keymap source file is not included in the listings at the end of the book because of its size, but some representative entries are illustrated in Fig. 3-41. Following the `#includes` are macros to define various constants. The first group are used in low-level interaction with the keyboard controller. Many of these are I/O port addresses or bit combinations that have meaning in these interactions. The next group includes symbolic names for special keys. The macro *kb_addr* (line 13041) always returns a pointer to the first element of the *kb_lines* array, since the IBM hardware supports only one keyboard. On the next line the size of the keyboard input buffer is symbolically defined as *KB_IN_BYTES*, with a value of 32. The next 11 variables are used to hold various states that must be remembered to properly interpret a key press. They are used in different ways. For instance, the value of the *capslock* flag (line 13046) is toggled between *TRUE* and *FALSE* each time the Caps Lock key is pressed. The *shift* flag (line 13054) is set to *TRUE* when the Shift key is pressed and to *FALSE* when the Shift key is released. The *esc* variable is set when a scan code escape is received. It is always reset upon receipt of the following character.

The *kb_s* structure on lines 13060 to 13065 is used to keep track of scan codes as they are entered. Within this structure the codes are held in a circular buffer, in the array *ibuf*, of size *KB_IN_BYTES*. An array *kb_lines*[*NR_CONS*] of these structures is declared, one per console, but in fact only the first one is used, since the *kb_addr* macro is always used to determine the address of the current *kb_s*. However, we usually refer to variables within *kb_lines*[0] using a pointer to the structure, for example, *kb->ihead*, for consistency with the way we treat other devices and to make the references in the text consistent with those in the source code listing. A small amount of memory is wasted because of the unused array elements, of course. However, if someone manufactures a PC with hardware support for multiple keyboards, MINIX is ready; only a modification of the *kb_addr* macro is required.

Map_key0 (line 13084) is defined as a macro. It returns the ASCII code that corresponds to a scan code, ignoring modifiers. This is equivalent to the first column (unshifted) in the keymap array. Its big brother is *map_key* (line 13091), which performs the complete mapping of a scan code to an ASCII code, including accounting for (multiple) modifier keys that are depressed at the same time as ordinary keys.

The keyboard interrupt service routine is *kbd_hw_int* (line 13123), called whenever a key is pressed or released. It calls *scan_keyboard* to get the scan code from the keyboard controller chip. The most significant bit of the scan code is set when a key release causes the interrupt, and in this case the key is ignored unless it is one of the modifier keys. If the interrupt is caused by a press of any, key, or the release of a modifier key, the raw scan code is placed in the circular buffer if there is space, the *tp->tty_events* flag for the current console is raised (line 13154), and then *force_timeout* is called to make sure the clock task will start the terminal task on the next clock tick. Figure 3-45 shows scan codes in the buffer for a short line of input that contains two upper case characters, each preceded by the scan code for depression of a shift key and followed by the code for the release of the shift key.

42	35	170	18	38	38	24	57	54	17	182	24	19	38	32	28
L+	h	L-	e	l	l	o	SP	R+	w	R-	o	r	l	d	CR

Figure 3-45. Scan codes in the input buffer, with corresponding key presses below, for a line of text entered at the keyboard. L+, L-, R+, and R- represent, respectively, pressing and releasing the left and right Shift keys. The code for a key release is 128 more than the code for a press of the same key.

When the clock interrupt occurs, the terminal task itself runs, and upon finding the *tp->tty_events* flag for the console device set, it calls *kb_read* (line 13165), the device-specific routine, using the pointer in the *tp->tty_devread* field of the console's *tty* structure. *Kb_read* takes scan codes from the keyboard's circular buffer and places ASCII codes in its local buffer, which is large enough to hold the escape sequences that must be generated in response to some scan codes from the numeric keypad. Then it calls *in_process* in the hardware-independent code to put the characters into the input queue. On lines 13181 to 13183 *lock* and *unlock* are used to protect the decrement of *kb->icount* from a possible keyboard interrupt arriving at the same time. The call to *make_break* returns the ASCII code as an integer. Special keys, such as keypad and function keys, have values greater than 0xFF at this point. Codes in the range from *HOME* to *INSERT* (0x101 to 0x10C, defined in *include/minix/keymap.h*) result from pressing the numeric keypad, and are converted into 3-character escape sequences shown in Fig. 3-46 using the *numpad_map* array. The sequences are then passed to *in_process* (lines 13196 to 13201). Higher codes are not passed on to *in_process*, but a check is made for the codes for ALT-LEFT-ARROW, ALT-RIGHT-ARROW, or ALT-F1 through ALT-F12, and if one of these is found, *select_console* is called to switch virtual consoles.

Key	Scan code	"ASCII"	Escape sequence
Home	71	0x101	ESC [H
Up Arrow	72	0x103	ESC [A
Pg Up	73	0x107	ESC [V
–	74	0x10A	ESC [S
Left Arrow	75	0x105	ESC [D
5	76	0x109	ESC [G
Right Arrow	77	0x106	ESC [C
+	78	0x10B	ESC [T
End	79	0x102	ESC [Y
Down Arrow	80	0x104	ESC [B
Pg Dn	81	0x108	ESC [U
Ins	82	0x10C	ESC [@

Figure 3-46. Escape codes generated by the numeric keypad. When scan codes for ordinary keys are translated into ASCII codes the special keys are assigned "pseudo ASCII" codes with values greater than 0xFF.

Make_break (line 13222) converts scan codes into ASCII and then updates the variables that keep track of the state of modifier keys. First, however, it checks for the magic CTRL-ALT-DEL combination that PC users all know as the way to force a reboot under MS-DOS. An orderly shutdown is desirable, however, so rather than try to start the PC BIOS routines, a *SIGABRT* signal is sent to *init*, the parent process of all other processes. *Init* is expected to catch this signal and interpret it as a command to begin an orderly process of shutting down, prior to causing a return to the boot monitor, from which a full restart of the system or a reboot of MINIX can be commanded. Of course, it is not realistic to expect this to work every time. Most users understand the dangers of an abrupt shutdown and do not press CTRL-ALT-DEL until something is really going wrong and normal control of the system has become impossible. At this point it is likely that the system may be so disrupted that an orderly sending of a signal to another process may be impossible. This is why there is a static variable *CAD_count* in *make_break*. Most system crashes leave the interrupt system still functioning, so keyboard input can still be received and the clock task can keep the terminal task running. Here MINIX takes advantage of the expected behavior of computer users, who are likely to bang on the keys repeatedly when something does not seem to work correctly. If the attempt to send the *SIGABRT* to *init* fails and the user presses CTRL-ALT-DEL twice more, a call to *wreboot* is made directly, causing a return to the monitor without going through the call to *init*.

The main part of *make_break* is not hard to follow. The variable *make* records whether the scan code was generated by a key press or a key release, and then the call to *map_key* returns the ASCII code to *ch*. Next is a `switch` on *ch* (lines 13248 to 13294). Let us consider two cases, an ordinary key and a special key. For an ordinary key, none of the cases match, and nothing should happen in the default case either (line 13292), since ordinary keys codes are supposed to be accepted only on the make (press) phase of a key press and release. If somehow an ordinary key code is accepted at key release, a value of -1 is substituted here, and this is ignored by the caller, *kb_read*. A special key, for example *CTRL*, is identified at the appropriate place in the `switch`, in this case on line 13249. The corresponding variable, in this case *control*, records the state of *make*, and -1 is substituted for the character code to be returned (and ignored). The handling of the *ALT*, *CALOCK*, *NLOCK*, and *SLOCK* keys is more complicated, but for all of these special keys the effect is similar: a variable records either the current state (for keys that are only effective while pressed) or toggles the previous state (for the lock keys).

There is one more case to consider, that of the *EXTKEY* code and the *esc* variable. This is not to be confused with the ESC key on the keyboard, which returns the ASCII code 0x1B. There is no way to generate the *EXTKEY* code alone by pressing any key or combination of keys; it is the PC keyboard's **extended key prefix**, the first byte of a 2-byte scan code that signifies that a key that was not part of the original PC's complement of keys but that has the same scan code, has been pressed. In many cases software treats the two keys identically. For instance, this is almost always the case for the normal "/" key and the gray "/" key on the numeric keyboard. In other cases, one would like to distinguish between such keys. For instance, many keyboard layouts for languages other than English treat the left and right ALT keys differently, to support keys that must generate three different character codes. Both ALT keys generate the same scan code (56), but the *EXTKEY* code precedes this when the right-hand ALT is pressed. When the *EXTKEY* code is returned, the *esc* flag is set. In this case, *make_break* returns from within the switch, thus bypassing the last step before a normal return, which sets *esc* to zero in every other case (line 13295). This has the effect of making the *esc* effective only for the very next code received. If you are familiar with the intricacies of the PC keyboard as it

is ordinarily used, this will be both familiar and yet a little strange, because the PC BIOS does not allow one to read the scan code for an ALT key and returns a different value for the extended code than does MINIX.

Set_leds (line 13303) turns on and off the lights that indicate whether the Num Lock, Caps Lock, or Scroll Lock keys on a PC keyboard have been pressed. A control byte, *LED_CODE*, is written to an output port to instruct the keyboard that the next byte written to that port is for control of the lights, and the status of the three lights is encoded in 3 bits of that next byte. The next two functions support this operation. *Kb_wait* (line 13327) is called to determine that the keyboard is ready to receive a command sequence, and *kb_ack* (line 13343) is called to verify that the command has been acknowledged. Both of these commands use busy waiting, continually reading until a desired code is seen. This is not a recommended technique for handling most I/O operations, but turning lights on and off on the keyboard is not going to be done very often and doing it inefficiently does not waste much time. Note also that both *kb_wait* and *kb_ack* could fail, and one can determine from the return code if this happens. But setting the light on the keyboard is not important enough to merit checking the value returned by either call, and *set_leds* just proceeds blindly.

Since the keyboard is part of the console, its initialization routine, *kb_init* (line 13359), is called from *scr_init* in *console.c*, not directly from *tty_init* in *tty.c*. If virtual consoles are enabled, (i.e., *NR_CONS* in *include/minix/config.h* is greater than 1), *kb_init* is called once for each logical console. After the first time the only part of *kb_init* that is essential for additional consoles is setting the address of *kb_read* into *tp->tty_devread* (line 13367), but no harm is done by repeating the rest of the function. The rest of *kb_init* initializes some variables, sets the lights on the keyboard, and scans the keyboard to be sure no leftover keystroke is read. When all is ready, it calls *put_irq_handler* and then *enuble_irq*, so *kbd_hw_int* will be executed whenever a key is pressed or released.

The next three functions are all rather simple. *Kbd_loadmap* (line 13392) is almost trivial. It is called by *do_ioctl* in *tty.c* to do the copying of a keymap from user space to overwrite the default keymap compiled by the inclusion of a keymap source file at the start of *keyboard.c*.

Func_key (line 13405) is called from *kb_read* to see if a special key meant for local processing has been pressed. Figure 3-47 summarizes these keys and their effects. The code called is found in several files. The *F1* and *F2* codes activate code in *dmp.c*, which we will discuss in the next section. The *F3* code activates *toggle_scroll*, which is in *console.c*, also to be discussed in the next section. The *CF7*, *CF8*, and *CF9* codes cause calls to *sigchar*, in *tty.c*. When networking is added to MINIX, an additional case, to detect the *F5* code, is added to display Ethernet statistics. A large number of other scan codes are available that could be used to trigger other debugging messages or special events from the console.

Scan_keyboard (line 13432) works at the hardware interface level, by reading and writing bytes from I/O ports. The keyboard controller is informed that a character has been read by the sequence on lines 13440 to 13442, which reads a byte, writes it again with the most significant bit set to 1, and then rewrites it with the same bit rest to 0. This prevents the same data from being read on a subsequent read. There is no status checking in reading the keyboard, but there should be no problems in any case, since *scan_keyboard* is only called in response to an interrupt, with the exception of the call from *kb_init* to clear out any garbage.

The last function in *keyboard.c* is *wreboot* (line 13450). If invoked as a result of a system panic, it provides an opportunity for the user to use the function keys to display

Key	Purpose
F1	Display process table
F2	Display details of process memory use
F3	Toggle between hardware and software scrolling
F5	Show Ethernet statistics (if network support compiled)
CF7	Send SIGQUIT, same effect as CTRL-\
CF8	Send SIGINT, same effect as DEL
CF9	Send SIGKILL, same effect as CTRL-U

Figure 3-47. The function keys detected by *func_keys()*.

debugging information. The loop on lines 13478 to 13487 is another example of busy waiting. The keyboard is read repeatedly until an ESC is typed. Certainly no one can claim that a more efficient technique is needed after a crash, while awaiting a command to reboot. Within the loop, *func_key* is called to provide a possibility of obtaining information that might help analyze the cause of a crash. We will not discuss further details of the return to the monitor. The details are very hardware-specific and do not have a lot to do with the operating system.

3.9.6 Implementation of the Display Driver

The IBM PC display may be configured as several virtual terminals, if sufficient memory is available. We will examine the console's device-dependent code in this section. We will also look at the debug dump routines that use low-level services of the keyboard and display. These provide support for limited interaction with the user at the console, even when other parts of the MINIX system are not functioning and can provide useful information even following a near-total system crash.

Hardware-specific support for console output to the PC memory-mapped screen is in *console.c*. The *console* structure is defined on lines 13677 to 13693. In a sense this structure is an extension of the *tty* structure defined in *tty.c*. At initialization the *tp->tty_priv* field of a console's *tty* structure is assigned a pointer to its own *console* structure. The first item in the *console* structure is a pointer back to the corresponding *tty* structure. The components of a *console* structure are what one would expect for a video display: variables to record the row and column of the cursor location, the memory addresses of the start and limit of memory used for the display, the memory address pointed to by the controller chip's base pointer, and the current address of the cursor. Other variables are used for managing escape sequences. Since characters are initially received as 8-bit bytes and must be combined with attribute bytes and transferred as 16-bit words to video memory, a block to be transferred is built up in *c_ramqueue*, an array big enough to hold an entire 80-column row of 16-bit character-attribute pairs. Each virtual console needs one *console* structure, and the storage is allocated in the array *cons_table* (line 13696). As we did with the *tty* and *kb_s* structures, we will usually refer to the elements of a *console* structure using a pointer, for example, *cons->c_tty*.

The function whose address is stored in each console's *tp->tty_devwrite* entry is *cons_write* (line 13729). It is called from only one place, *handle_events* in *tty.c*. Most of the other functions in *console.c* exist to support this function. When it is called for the first time

after a client process makes a WRITE call, the data to be output are in the client's buffer, which can be found using the *tp->tty_ourproc* and *tp->out_vir* fields in the *tty* structure. The *tp->tty_outleft* field tells how many characters are to be transferred, and the *tp->tty_outcum* field is initially zero, indicating none have yet been transferred. This is the usual situation upon entry to *cons_write*, because normally, once called, it transfers all the data requested in the original call. However, if the user wants to slow the process in order to review the data on the screen, he may enter a *STOP* (CTRL-S) character at the keyboard, resulting in raising of the *tp->tty_inhibited* flag. *Cons_write* returns immediately when this flag is raised, even though the WRITE has not been completed. In such a case *handle_events* will continue to call *cons_write*, and when *tp->tty_inhibited* is finally reset, by the user entering a *START* (CTRL-Q) character, *cons_write* continues with the interrupted transfer.

Cons_write's sole argument is a pointer to the particular console's *tty* structure, so the first thing that must be done is to initialize *cons*, the pointer to this console's *console* structure (line 13741). Then, because *handle_events* calls *cons_write* whenever it runs, the first action is a test to see if there really is work to be done. A quick return is made if not (line 13746). Following this the main loop on lines 13751 to 13778 is entered. This loop is very similar in structure to the main loop of *in_transfer* in *tty.c*. A local buffer that can hold 64 characters is filled by calling *phys_copy* to get the data from the client's buffer, the pointer to the source and the counts are updated, and then each character in the local buffer is transferred to the *cons->c_ramqueue* array, along with an attribute byte, for later transfer to the screen by *flush*. There is more than one way to do this transfer, as we saw in Fig. 3-39. *Out_char* can be called to do this for each character, but it is predictable that none of the special services of *out_char* will be needed if the character is a visible character, an escape sequence is not in progress, the screen width has not been exceeded, and *cons->c_ramqueue* is not full. If the full service of *out_char* is not needed, the character is placed directly into *cons->c_ramqueue*, along with the attribute byte (retrieved from *cons->c_attr*), and *cons->c_rwords* (the index into the queue), *cons->c_column* (which keeps track of the column on the screen), and *tbuf* the pointer into the buffer, are all incremented. This direct placement of characters into *cons->c_ramqueue* corresponds to the dashed line on the left side of Fig. 3-39. If needed, *out_char* is called (lines 13766 to 13777). It does all of the bookkeeping, and additionally calls *flush*, which does the final transfer to screen memory, when necessary. The transfer from the user buffer to the local buffer to the queue is repeated as long as *tp->ttv_outleft* indicates there are still characters to be transferred and the flag *tp->tty_inhibited* has not been raised. When the transfer stops, whether because the WRITE operation is complete or because *tp->tty_inhibited* has been raised, *flush* is called again to transfer the last characters in the queue to screen memory. If the operation is complete (tested by seeing if *tp->tty_outleft* is zero), a reply message is sent by calling *tty_reply* (lines 13784 and 13785).

In addition to calls to *cons_write* from *handle_events*, characters to be displayed are also sent to the console by *echo* and *rawecho* in the hardware-independent part of the terminal task. If the console is the current output device, calls via the *tp->tty_echo* pointer are directed to the next function, *cons_echo* (line 13794). *Cons_echo* does all of its work by calling *out_char* and then *flush*. Input from the keyboard arrives character by character and the person doing the typing wants to see the echo with no perceptible delay, so putting characters into the output queue would be unsatisfactory.

Now we arrive at *our_char* (line 13809). It does a test to see if an escape sequence is in progress, calling *parse_escape* and then returning immediately if so (lines 13814 to 13816). Otherwise, a `switch` is entered to check for special cases: null, backspace, the

bell character, and so on. The handling of most of these is easy to follow. The linefeed and the tab are the most complicated, since they involve complicated changes, to the position of the cursor on the screen and may require scrolling as well. The last test is for the *ESC* code. If it is found, the *cons->c_esc_state* flag is set (line 13871), and future calls to *our_char* are diverted to *parse_escape* until the sequence is complete. At the end, the default is taken for printable characters. If the screen width has been exceeded, the screen may need to be scrolled, and *flush* is called. Before a character is placed in the output queue a test is made to see that the queue is not full, and *flush* is called if it is. Putting a character into the queue requires the same bookkeeping we saw earlier in *cons_write*.

The next function is *scroll_screen* (line 13896). *Scroll_screen* handles both scrolling up, the normal situation that must be dealt with whenever the bottom line on the screen is full, and scrolling down, which occurs when cursor positioning commands attempt to move the cursor beyond the top line of the screen. For each direction of scroll there are three possible methods. These are required to support different kinds of video cards.

We will look at the scrolling up case. To begin, *chars* is assigned the size of the screen minus one line. Softscrolling is accomplished by a single call to *vid_vid_copy* to move *chars* characters lower in memory, the size of the move being the number of characters in a line. *Vid_vid_copy* can wrap, that is, if asked to move a block of memory that overflows the upper end of the block assigned to the video display, it fetches the overflow portion from the low end of the memory block and moves it to an address higher than the part that is moved lower, treating the entire block as a circular array. The simplicity of the call hides a fairly slow operation. Even though *vid_vid_copy* is an assembly language routine defined in *klib386.s*, this call requires the CPU to move 3840 bytes, which is a large job even in assembly language.

The softscroll method is never the default; the operator is supposed to select it only if hardware scrolling does not work or is not desired for some reason. One such reason might be a desire to use the *screendump* command to save the screen memory in a file. When hardware scrolling is in effect, *screendump* is likely to give unexpected results, because the start of the screen memory is likely not to coincide with the start of the visible display.

On line 13917 the *wrap* variable is tested as the first part of a compound test. *Wrap* is true for older displays that can support hardware scrolling, and if the test fails, simple hardware scrolling occurs on line 13921, where the origin pointer used by the video controller chip, *cons->c_org*, is updated to point to the first character to be displayed at the upper-left corner of the display. If *wrap* is *FALSE*, the compound test continues with a test of whether the block to be moved up in the scroll operation overflows the bounds of the memory block designated for this console. If this is so, *vid_vid_copy* is called again to make a wrapped move of the block to the start of the console's allocated memory, and the origin pointer is updated. If there is no overlap, control passes to the simple hardware scrolling method always used by older video controllers. This consists of adjusting *cons->c_org* and then putting the new origin in the correct register of the controller chip. The call to do this is done later, as is a call to blank the bottom line on the screen.

The code for scrolling down is very similar to that for scrolling up. Finally, *mem_vid_copy* is called to blank out the line at the bottom (or top) addressed by *new_line*. Then *set_6845* is called to write the new origin from *cons->c_org* into the appropriate registers, and *flush* makes sure all changes become visible on the screen.

We have mentioned *flush* (line 13951) several times. It transfers the characters in the queue to the video memory using *mem_vid_copy*, updates some variables, and then makes sure the row and column numbers are reasonable, adjusting them if, for instance, an escape sequence has tried to move the cursor to a negative column position. Finally a

calculation of where the cursor ought to be is made and is compared with *cons->c_cur*. If they do not agree, and if the video memory that is currently being handled belongs to the current virtual console, a call to *set_6845* is made to set the correct value in the controller's cursor register.

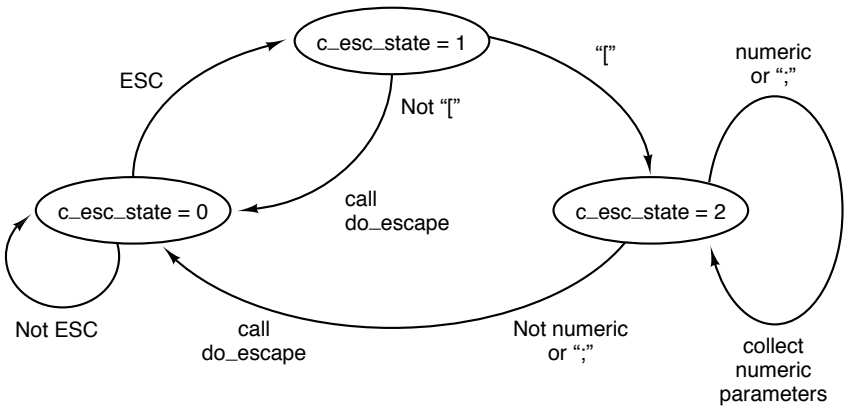


Figure 3-48. Finite state machine for processing escape sequences.

Figure 3-48 shows how escape sequence handling can be represented as a finite state machine. This is implemented by *parse_escape* (line 13986) which is called at the start of *out_char* if *cons->c_esc_state* is nonzero. An ESC itself is detected by *out_char* and makes *cons->c_esc_state* equal to 1. When the next character is received, *parse_escape* prepares for further processing by putting a '\0' in *cons->c_esc_intro*, a pointer to the start of the array of parameters, *cons->c_esc_parmv*[0] into *cons->c_esc_parmp*, and zeroes into the parameter array itself. Then the first character following the ESC is examined—valid values are either "[" or "M". In the first case the "[" is copied to *cons->c_esc_intro* and the state is advanced to 2. In the second case, *do_escape* is called to carry out the action, and the escape state is reset to zero. If the first character after the ESC is not one of the valid ones, it is ignored and succeeding characters are once again displayed normally.

When an ESC [sequence has been seen, the next character entered is processed by the escape state 2 code. There are three possibilities at this point. If the character is a numeric character, its value is extracted and added to 10 times the existing value in the position currently pointed to by *cons->c_esc_parmp*, initially *cons->c_esc_parmv*[O] (which was initialized to zero). The escape state does not change. This makes it possible to enter a series of decimal digits and accumulate a large numeric parameter, although the maximum value currently recognized by MINIX is 80, used by the sequence that moves the cursor to an arbitrary position (lines 14027 to 14029). If the character is a semicolon, the pointer to the parameter string is advanced, so succeeding numeric values can be accumulated in the second parameter (lines 14031 to 14033). If *MAX_ESC_PARMS* were to be changed to allocate a larger array for the parameters, this code would not have to be altered to accumulate additional numeric values after entry of additional parameters. Finally, if the character is neither a numeric digit nor a semicolon, *do_escape* is called.

Do_escape (line 14045) is one of the longer functions in the MINIX system source code, even though MINIX's complement of recognized escape sequences is relatively modest. For all its length, however, the code should be easy to follow. After an initial call to *flush* to make sure the video display is fully updated, there is a simple `if` choice, depending upon whether the character immediately following the ESC character was a special control sequence introducer or not. If not, there is only one valid action, moving the cursor up one line if the sequence was ESC M. Note that the test for the "M" is

done within a switch with a default action, as a validity check and in anticipation of addition of other sequences that do not use the ESC [format. The action is typical of many escape sequences: the *cons->c_row* variable is inspected to determine if scrolling is required. If the cursor is already on row 0, a *SCROLL_DOWN* call is made to *scroll_screen*; otherwise the cursor is moved up one line. The latter is accomplished just by decrementing *cons->c_row* and then calling *flush*. If a control sequence introducer is found, the code following the `else` on line 14069 is taken. A test is made for "[", the only control sequence introducer currently recognized by MINIX. If the sequence is valid, the first parameter found in the escape sequence, or zero if no numeric parameter was entered, is assigned to *value* (line 14072). If the sequence is invalid, nothing happens except that the large `switch` that ensues (lines 14073 to 14272) is skipped and the escape state is reset to zero before returning from *do_escape*. In the more interesting case that the sequence is valid, the `switch` is entered. We will not discuss all the cases; we will just note several that are representative of the types of actions governed by escape sequences.

The first five sequences are generated, with no numeric arguments, by the four "arrow" keys and the Home key on the IBM PC keyboard. The first two, ESC [A and ESC [B, are similar to ESC M, except they can accept a numeric parameter and move up and down by more than one line, and they do not scroll the screen if the parameter specifies a move that exceeds the bounds of the screen. In such cases, *flush* catches requests to move out of bounds and limits the move to the last row or the first row, as appropriate. The next two sequences, ESC [C and ESC [D, which move the cursor right and left, are similarly limited by *flush*. When generated by the "arrow" keys there is no numeric argument, and thus the default movement of one line or column occurs.

The next sequence, ESC [H, can take two numeric parameters, for instance, ESC [20;60H. The parameters specify an absolute position rather than one relative to the current position and are converted from 1-based numbers to 0-based numbers for proper interpretation. The Home key generates the default (no parameters) sequence which moves the cursor to position (1, 1).

The next two sequences, ESC [*s*J and ESC [*s*K, clear a part of either the entire screen or the current line, depending upon the parameter that is entered. In each case a count of characters is calculated. For instance, for ESC [1J, *count* gets the number of characters from the start of the screen to the cursor position, and the count and a position parameter, *dst*, which may be the start of the screen, *cons->c_org*, or the current cursor position, *cons->c_cur*, are used as parameters to a call to *mem_vid_copy*. This procedure is called with a parameter that causes it to fill the specified region with the current background color.

The next four sequences insert and delete lines and spaces at the cursor position, and their actions do not require detailed explanation. The last case, ESC [*n*m (note the *n* represents a numeric parameter, but the "m" is a literal character) has its effect upon *cons->c_attr*, the attribute byte that is interleaved between the character codes when they are written to video memory.

The next function, *set_6845* (line 14280), is used whenever it is necessary to update the video controller chip. The 6845 has internal 16-bit registers that are programmed 8 bits at a time, and writing a single register requires four I/O port write operations. *Lock* and *unlock* calls are used to disable interrupts, which can cause problems if allowed to disrupt the sequence. Some of the registers of the 6845 video controller chip are shown in Fig. 3-49

The *beep* function (line 14300) is called when a CTRL-G character must be output. It takes advantage of the built-in support provided by the PC for making sounds by send-

Registers	Function
10 – 11	Cursor size
12 – 13	Start address for drawing screen
14 – 15	Cursor position

Figure 3-49. Some of the 6845's registers.

ing a square wave to the speaker. The sound is initiated by more of the kind of magic manipulation of I/O ports that only assembly language programmers can love, again with some concern that a critical part of the process should be protected from interrupts. The more interesting part of the code is the use of the clock task's capability to set an alarm, which can be used to initiate a function. The next routine, *stop_beep* (line 14329), is the one whose address is put into the message to the clock task. It stops the beep after the designated time has elapsed and also resets the *beeping* flag which is used to prevent superfluous calls to the beep routine from having any effect.

Scr_init (line 14343) is called by *tty_init NR_CONS* times. Each time its argument is a pointer to a *tty* structure, one element of the *tty_table*. On lines 14354 and 14355 *line*, to be used as the index into the *cons_table* array, is calculated, tested for validity, and, if valid, used to initialize *cons*, the pointer to the current console table entry. At this point the *cons->c_tty* field can be initialized with the pointer to the main *tty* structure for the device, and, in turn, *tp->tty_priv* can be pointed to this device's *console_t* structure. Next, *kb_init* is called to initialize the keyboard, and then the pointers to device specific routines are set up, *tp->tty_devwrite* pointing to *cons_write* and *tp->tty_echo* pointing to *cons_echo*. The I/O address of the base register of the CRT controller is fetched and the address and size of the video memory are determined on lines 14368 to 14378, and the *wrap* flag (used to determine how to scroll) is set according to the class of video controller in use. On lines 14382 to 14384 the segment descriptor for the video memory is initialized in the global descriptor table.

Next comes the initialization of virtual consoles. Each time *scr_init* is called, the argument is a different value of *tp*, and thus a different *line* and *cons* are used on lines 14393 to 14396 to provide each virtual console with its own share of the available video memory. Each screen is then blanked, ready to start, and finally console 0 is selected to be the first active one.

The remaining routines in *console.c* are short and simple and we will review them quickly. *Putk* (line 14408) has been mentioned earlier. It prints a character on behalf of any code linked into the kernel image that needs the service, without going through the FS. *Toggle_scroll* (line 14429) does what its name says, it toggles the flag that determines whether hardware or software scrolling is used. It also displays a message at the current cursor position to identify the selected mode. *Cons_stop* (line 14442) reinitializes the console to the state that the boot monitor expects, prior to a shutdown or reboot. *Cons_org0* (line 14456) is used only when a change of scrolling mode is forced by the F3 key, or when preparing to shut down. *Select_console* (line 14482) selects a virtual console. It is called with the new index and calls *set_6845* twice to get the video controller to display the proper part of the video memory.

The last two routines are highly hardware-specific. *Con_loadfont* (line 14497) loads a font into a graphics adapter, in support of the IOCTL *TIOCSFON* operation. It calls *ga_program* (line 14540) to do a series of magical writes to an I/O port that cause the

video adapter's font memory, which is normally not addressable by the CPU, to be visible. Then *phys_copy* is called to copy the font data to this area of memory, and another magic sequence is invoked to return the graphics adapter to its normal mode of operation.

Debugging Dumps

The final group of procedures we will discuss in the terminal task were originally intended only for temporary use when debugging MINIX. They can be removed when this assistance is no longer needed, but many users find them useful and leave them in place. They are particularly helpful when modifying MINIX.

As we have seen, *func_key* is called at the start of *kb_read* to detect scan codes used for control and debugging. The dump routines called when the F1 and F2 keys are detected are in *dmp.c*. The first, *p_dmp* (line 14613) displays basic process information for all processes, including some information on memory use, when the F1 key is pressed. The second, *map_dmp* (line 14660), provides more detailed information on memory use in response to F2. *Proc_name* (line 14690) supports *p_dmp* by looking up process names.

Since this code is completely contained within the kernel binary itself and does not run as a user process or task, it frequently continues to function correctly, even after a major system crash. Of course, these routines are accessible only from the console. The information provided by the dump routines cannot be redirected to a file or to any other device, so hardcopy or use over a network connection are not options.

We suggest that the first step in trying to add any improvement to MINIX might very well be to extend the dumping routines to provide more information on the aspect of the system you wish to improve.

3.10 The System Task in MINIX

One consequence of making the file system and memory manager server processes outside the kernel is that occasionally they have some piece of information that the kernel needs. This structure, however, forbids them from just writing it into a kernel table. For example, the FORK system call is handled by the memory manager. When a new process is created, the kernel must know about it, in order to schedule it. How can the memory manager tell the kernel?

The solution to this problem is to have a kernel task that communicates with the file system and memory manager via the standard message mechanism and which also has access to all the kernel tables. This task, called the **system task**, is in layer 2 in Fig. 2-26, and functions like the other tasks we have studied in this chapter. The only difference is that it does not control any I/O device. But, like I/O tasks, it implements an interface, in this case not to the external world, but to the most internal part of the system. It has the same privileges as the I/O tasks and is compiled with them into the kernel image, and it makes more sense to study it here than in any other chapter.

The system task accepts 19 kinds of messages, shown in Fig. 3-50. The main program of the system task, *sys_task* (line 14837), is structured like other tasks. It gets a message, dispatches to the appropriate service procedure, and then sends a reply. We will now look at each of these messages and its service procedure.

Figure 3-50. The message types accepted by the system task.

The *SYS_FORK* message is used by the memory manager to tell the kernel that a new process has come into existence. The kernel needs to know this in order to schedule it. The message contains the slot numbers within the process table corresponding to the parent and child. The memory manager and file system also have process tables, with entry *k* referring to the same process in all three. In this manner, the memory manager can specify just the parent and child slot numbers, and the kernel will know which processes are meant.

The procedure *do_fork* (line 14877) first makes a check (line 14886) to see if the memory manager is feeding the kernel garbage. The test uses a macro, *isoksusern*, defined in *proc.h*, to test that the process table entries of the parent and child are valid. Similar tests are made by most of the service procedures in *system.c*. This is pure paranoia, but a little internal consistency checking does no harm. Then *do_fork* copies the parent's process table entry to the child's slot. Some things need adjustment here. The child is freed from any pending signals for the parent, and the child does not inherit the parent's trace status. And, of course, all the child's accounting information is set to zero.

After a FORK, the memory manager allocates memory for the child. The kernel must know where the child is located in memory so it can set up the segment registers properly when running the child. The *SYS_NEWMAP* message allows the memory manager to give the kernel any process' memory map. This message can also be used after a BRK system call changes the map.

The message is handled by *do_newmap* (line 14921), which must first copy the new map from the memory manager's address space. The map is not contained in the message itself because it is too big. In theory, the memory manager could tell the kernel that the map is at address *m*, where m is an illegal address. The memory manager is not supposed to do this, but the kernel checks anyway. The map is copied directly into the *p_map* field of the process table entry for the process getting the new map. The call to *alloc_segments* extracts information from the map and loads it into the *p_reg* fields that hold the segment registers. This is not complicated, but the details are processor dependent and are segregated in a separate function for this reason.

The *SYS_NEWMAP* message is much used in the normal operation of a MINIX system. A similar message, *SYS_GETMAP*, is used only when the file system initially starts up. This message requests a transfer of the process map information in the opposite direction, from the kernel to the memory manager. It is carried out by *do_getmap* (line 14957). The code of the two functions is similar, differing mainly in the swapping of the source and destination arguments of the call to *phys_copy* used by each function.

When a process does an EXEC system call, the memory manager sets up a new stack for it containing the arguments and environment. It passes the resulting stack pointer to the kernel using *SYS_EXEC*, which is handled by *do_exec* (line 14990). After the usual check for a valid process, there is a test of the *PROC2* field in the message. This field is used here as a flag to indicate whether the process is being traced and has nothing to do with identifying a process. If tracing is in force, *cause_sig* is called to send a *SIGTRAP* signal to the process. This does not have the usual consequences of this signal, which would normally terminate a process and cause a core dump. In the memory manager all signals to a traced process except for *SIGKILL* are intercepted and cause the signaled process to stop so a debugging program can then control its further execution.

The EXEC call causes a slight anomaly. The process invoking the call sends a message to the memory manager and blocks. With other system calls, the resulting reply unblocks it. With EXEC there is no reply, because the newly loaded core image is not expecting a reply. Therefore, *do_exec* unblocks the process itself on line 15009. A The

next line makes the new image ready to run, using the *lock_ready* function that protects against a possible race condition. Finally, the command string is saved so the process can be identified when the user presses the F1 function key to display the status of all processes.

Processes can exit in MINIX either by doing an EXIT system call, which sends a message to the memory manager, or by being killed by a signal. In both cases, the memory manager tells the kernel by the *SYS_XIT* message. The work is done by *do_xit* (line 15027), which is more complicated than you might expect. Taking care of the accounting information is straightforward. The alarm timer, if any, is killed by storing a zero on top of it. It is for this reason that the clock task always checks when a timer has run out to see if anybody is still interested. The tricky part of *do_xit* is that the process might have been queued trying to send or receive at the time it was killed. The code on lines 15056 to 15076 checks for this possibility. If the exiting process is found on any other process' message queue, it is carefully removed.

In contrast to the previous message, which is slightly complicated, *SYS_GETSP* is completely trivial. It is used by the memory manager to find out the value of the current stack pointer for some process. This value is needed for the BRK and SBRK system calls to see if the data segment and stack segment have collided. The code is in *do_getsp* (line 15089).

Now we come to one of the few message types used exclusively by the file system, *SYS_TIMES*. It is needed to implement the TIMES system call, which returns the accounting times to the caller. All *do_times* (line 15106) does is put the requested times into the reply message. Calls to *lock* and *unlock* are used to protect against a possible race while accessing the time counters.

It can happen that either the memory manager or the file system discovers an error that makes it impossible to continue operation. For example, if upon first starting up the file system sees that the super-block on the root device has been fatally corrupted, it panics and sends a *SYS_ABORT* message to the kernel. It is also possible for the super-user to force a return to the boot monitor and/or a reboot, using the *reboot* command, which in turn calls the REBOOT system call. In any of these cases, the system task executes *do_abort* (line 15131), which copies instructions to the monitor, if necessary, and then calls wreboot to complete the process.

Most of the work of signal handling is done by the memory manager, which checks to see if the process to be signaled is enabled to catch or ignore the signal, if the sender of the signal is entitled to do so, and so on. However, the memory manager cannot actually cause the signal, which requires pushing some information onto the stack of the signaled process.

Signal handling previous to POSIX was problematic, because catching a signal restored the default response to signals. If continued special handling of subsequent signals were required, the programmer could not guarantee reliability. Signals are asynchronous, and a second signal could very well arrive before the handling were reenabled. POSIX-style signal handling solves this problem, but the price is a more complicated mechanism. Old-style signal handling could be implemented by the operating system pushing some information onto the signaled process' stack, similar to the information pushed by an interrupt. The programmer would then write a handler that ended with a return instruction, popping the information needed to resume execution. POSIX saves more information when a signal is received than can be conveniently handled this way. There is additional work to do afterward, before the signaled process can resume what it was doing. The memory manager thus has to send two messages to the system task to process a signal.

The payoff for this effort is more reliable handling of signals.

When a signal is to be sent to a process, the *SYS_SENDSIG* message is sent to the system task. It is handled by *do_sendsig* (line 15157). The information needed to handle POSIX-style signals is in a *sigcontext* structure, which contains the processor register contents, and a *sigframe* structure, which contains information about how signals are to be handled by the process. Both of these structures need some initialization, but the basic work of *do_sendsig* is just to put the required information on the signaled process' stack, and adjust the signaled process' program counter and stack pointer so the signal handling code will be executed the next time the scheduler allows the process to execute.

When a POSIX-style signal handler completes its work, it does not pop the address where execution of the interrupted process resumes, as is the case with old-style signals. The programmer writing the handler writes a *return* instruction (or the high-level language equivalent), but the manipulation of the stack by the SENDSIG call causes the *return* to execute a SIGRETURN system call. The memory manager then sends the system task a *SYS_SIGRETURN* message. This is handled by *do_sigreturn* (line 15221), which copies the *sigcontext* structure back into the kernel's space and then restores the signaled process' registers. The interrupted process will resume execution at the point where it was interrupted the next time the scheduler allows it to run, retaining any special signal handling that was previously set up.

The SIGRETURN system call, unlike most of the others discussed in this section, is not required by POSIX. It is a MINIX invention, a convenient way to initiate the processing needed when a signal handler is complete. Programmers should not use this call; it will not be recognized by other operating systems, and in any case there is no need to refer to it explicitly.

Some signals come from within the kernel image, or are handled by the kernel before they go to the memory manager. These include signals originating from tasks, such as alarms from the clock task, or signal-causing key presses detected by the terminal task, as well as signals caused by exceptions (such as division by zero or illegal instructions) detected by the CPU. Signals originating from the file system are also handled first by the kernel. The *SYS_KILL* message is used by the file system to request that such a signal be generated. The name is perhaps a bit misleading. This has nothing to do with handling of the KILL system call, used by ordinary processes to send signals. This message is handled by *do_kill* (line 15276), which makes the usual check for a valid origin of the message, and then calls *cause_sig* to actually pass the signal on to the process. Signals originating in the kernel are also passed on by a call to this function, which initiates signals by sending a *KSIG* message to the memory manager.

Whenever the memory manager has finished with one of these *KSIG*-type signals, it sends a *SYS_ENDSIG* message back to the system task. This message is handled by *do_endsig* (line 15294), which decrements the count of pending signals, and, if it reaches zero, resets the *SIG_PENDING* bit for the signaled process. If there are no other flags set indicating reasons the process should not be runnable, *lock_ready* is then called to allow the process to run again.

The *SYS_COPY* message is the most heavily used one. It is needed to allow the file system and memory manager to copy information to and from user processes.

When a user does a READ call, the file system checks its cache to see if it has the block needed. If not, it sends a message to the appropriate disk task to load it into the cache. Then the file system sends a message to the system task telling it to copy the block to the user process. In the worst case, seven messages are needed to read a block; in the best case, four messages are needed. Both cases are shown in Fig. 3-51. These

messages are a significant source of overhead in MINIX and are the price paid for the highly modular design.

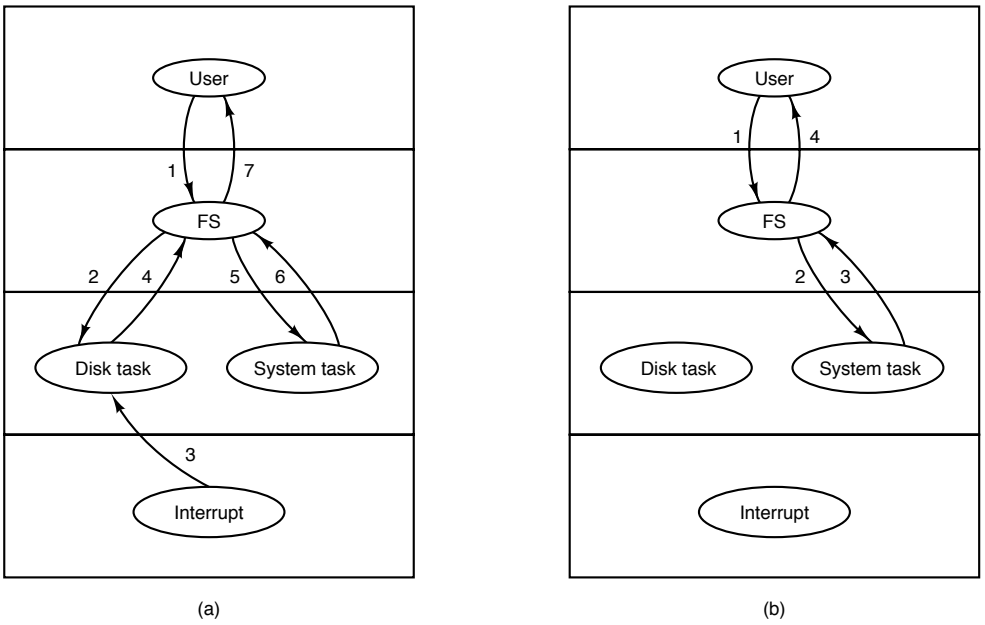


Figure 3-51. (a) Worst case for reading a block requires seven messages. (b) Best case for reading a block requires four messages.

As an aside, on the 8088, which had no protection, it would have been easy enough to cheat and let the file system copy the data to the caller's address space, but this would have violated the design principle. Anyone with access to such an antique machine who is interested in improving the performance of MINIX should look carefully at this mechanism to see how much improper behavior one can tolerate for how much gain in performance. Of course, this means of improvement is not available on Pentium-class machines with protection mechanisms.

Handling a *SYS_COPY* request is straightforward. It is done by *do_copy* (line 15316) and consists of little more than extracting the message parameters and calling *phys_copy*.

One way to deal with some of the inefficiency of the message passing mechanism is to pack multiple requests into a message. The *SYS_VCOPY* message does this. The content of this message is a pointer to a vector specifying multiple blocks to be copied between memory locations. The function *do_vcopy* (line 15364) executes a loop, extracting source and destination addresses and block lengths and calling *phys_copy* repeatedly until all the copies are complete. This is similar to the capability of disk devices to handle multiple transfers based on a single request.

There are several more message types received by the system task, most of which are fairly simple. Two of these are normally used only during system startup. The file system sends a *SYS_GBOOT* message to request the boot parameters. This is a structure, *bparam_s*, declared in *include/minix/boot.h*, which allows various aspects of system configuration to be specified by the boot monitor program before MINIX is started. The *do_gboot* (line 15403) function carries out this operation, which is just a copy from one part of memory to another. Also at startup time, the memory manager sends the system task a series of *SYS_MEM* messages to request the base and size of the available chunks of memory. *Do_mem* (line l5424) handles this request.

The *SYS_UMAP* message is used by a nonkernel process to request calculation of the

physical memory address for a given virtual address. *Do_umap* (line l5445) carries this out by calling *umap*, which is the function called from within the kernel to handle this conversion.

The last message type we will discuss is *SYS_TRACE*, which supports the PTRACE system call, used for debugging. Debugging is not a fundamental operating system function, but operating system support can make it easier. With help from the operating system, a debugger can examine and modify the memory used by a process under test, as well as the contents of the processor registers that are stored in the process table whenever the debugged program is not running.

Normally, a process runs until it blocks to wait for I/O or uses up a time quantum. Most CPU designs also provide means by which a process can be limited to executing just a single instruction, or can be made to execute only until a particular instruction is reached, by setting a **breakpoint**. Taking advantage of such facilities makes possible detailed analysis of a program.

There are eleven operations that can be carried out using PTRACE. A few are carried out totally by the memory manager, but for most of them the memory manager sends a *SYS_TRACE* message to the system task, which then calls *do_trace* (line l5467). This function implements a `switch` on the trace operation code. The operations are generally simple. A *P_STOP* bit in the process table is used by MINIX to recognize that debugging is in progress and is set by the command to stop the process (case *T_STOP*) or reset to restart it (case *T_RESUME*). Debugging depends upon hardware support, and on Intel processors is controlled by a bit in the CPU's flag register. When the bit is set, the processor executes just one instruction, then generates a *SIGTRAP* exception. As mentioned earlier, the memory manager stops a program being traced when a signal is sent to it. This *TRACEBIT* is manipulated by the *T_STOP* and *T_STEP* commands. Breakpoints can be set in two ways: either by using the *T_SETINS* command to replace an instruction with a special code that generates a *SIGTRAP*, or by using the *T_SETUSER* command to modify special breakpoint registers. On any kind of system to which MINIX may be ported, it will probably be possible to implement a debugger using similar techniques, but porting these functions will require study of the particular hardware.

Most of the commands carried out by *do_trace* either return or modify values in the traced process' text or data space, or in its process table entry, and the code is straightforward. Altering certain registers and certain bits of the CPU flags is too dangerous to allow, so there are many checks in the code that handle the *T_SETUSER* command to prevent such operations.

At the end of *system.c* are several utility procedures used in various places throughout the kernel. When a task needs to cause a signal (e.g., the clock task needs to cause a *SIGALRM* signal, or the terminal task needs to cause a *SIGINT* signal), it calls *cause_sig* (line 15586). This procedure sets a bit in the *p_pending* field of the process table entry for the process to be signaled and then checks to see if the memory manager is currently waiting for a message from ANY, that is, if it is idle and waiting for the next request to process. If it is idle, *inform* is called to tell the memory manager to handle the signal.

Inform (line 15627) is called only after a check that the memory manager is not busy, as described above. In addition to the call from *cause_sig*, it is called from *mini_rec*(in *proc.c*), whenever the memory manager blocks and there are kernel signals pending. *Inform* builds a message of type *KSIG* and sends it to the memory manager. The task or process calling *cause_sig* continues running as soon as the message has been copied into the memory manager's receive buffer. It does not wait for the memory manager to run, as would be the case if the normal send mechanism, which causes the sender to block,

were to be used. Before it returns, however, inform calls *lock_pick_proc*, which schedules the memory manager to run. Since tasks have a higher priority than servers, the memory manager will not run until all tasks are satisfied. When the signaling task finishes, the scheduler will be entered. If the memory manager is the highest priority runnable process, it will run and process the signal.

The procedure *umap* (line 15658) is a generally useful procedure that maps a virtual address onto a physical address. As we have noted, it is called by *do_umap*, which services the *SYS_UMAP* message. Its parameters are a pointer to the process table entry for the process or task whose virtual address space is to be mapped, a flag specifying the text, data, or stack segment, the virtual address itself, and a byte count. The byte count is useful because *umap* checks to make sure that the entire buffer starting at the virtual address is within the process' address space. To do this, it must know the buffer's size. The byte count is not used for the mapping itself, just this check. All the tasks that copy data to or from user space compute the physical address of the buffer using *umap*. For device drivers it is convenient to be able to get the services of *umap* starting with the process number instead of a pointer to a process table entry. *Numap* (line 15697) does this. It calls *proc_addr* to convert its first argument and then calls *umap*.

The last function defined in *system.c* is *alloc_segments* (line 15715). It is called by *do_newmap*. It is also called by the main routine of the kernel during initialization. This definition is very hardware dependent. It takes the segment assignments that are recorded in a process table entry and manipulates the registers and descriptors the Pentium processor uses to support protected segments at the hardware level.

3.11 Summary

Input/output is an often neglected, but important, topic. A substantial fraction of any operating system is concerned with I/O. We started out by looking at I/O hardware, and the relation of I/O devices to I/O controllers, which are what the software has to deal with. Then we looked at the four levels of I/O software: the interrupt routines, the device drivers, the device-independent I/O software, and the I/O libraries and spoolers that run in user space.

Next we studied the problem of deadlock and how it can be tackled. Deadlock occurs when a group of processes each have been granted exclusive access to some resources, and each one wants yet another resource that belongs to another process in the group. All of them are blocked and none will ever run again. Deadlock can be prevented by structuring the system so it can never occur, for example, by allowing a process to hold only one resource at any instant. It can also be avoided by examining each resource request to see if it leads to a situation I in which deadlock is possible (an unsafe state) and denying or delaying those that lead to trouble.

Device drivers in MINIX are implemented as processes embedded in the kernel. We have looked at the RAM disk driver, hard disk driver, clock driver, and terminal driver. The synchronous alarm task and the system task are not device drivers but are structurally very similar to one. Each of these tasks a main loop that gets requests and processes them, eventually sending back replies to report on what happened. All the tasks are located in the same address space. The RAM disk, hard disk, and floppy disk driver tasks all use a single copy of the same main loop and also share common functions. Nevertheless, each one is an independent process. Several different terminals, using the system console, the serial lines, and network connections are all supported by a single terminal task.

Device drivers have varying relationships to the interrupt system. Devices which can complete their work rapidly, such as the RAM disk and the memory-mapped display, do not use interrupts at all. The hard disk driver task does most of its work in the task code itself, and the interrupt handlers just return status information. The clock interrupt handler does a number of bookkeeping operations itself and only sends a message to the clock task when there is some work that cannot be taken care of by the handler. The keyboard interrupt handler buffers input and never sends a message to its task. Instead it changes a variable inspected by the clock interrupt handler; the latter sends a message to the terminal task.

Chapter 4

MEMORY MANAGEMENT

Memory is an important resource that must be carefully managed. While the average home computer nowadays has fifty times as much memory as the IBM 7094, the largest computer in the world in the early 1960s, programs are getting bigger just as fast as memories. To paraphrase Parkinson's law, "Programs expand to fill the memory available to hold them." In the chapter we will study how operating systems manage their memory.

Ideally, what every programmer would like is an infinitely large, fast memory that is also nonvolatile, that is, does not lose its contents when the electric power fails. While we are at it, why not also ask for it to be inexpensive, too. Unfortunately technology does not provide such memories. Consequently, most computers have a **memory hierarchy**, with a small amount of very fast, expensive, volatile cache memory, some number of megabytes of medium-speed, medium-price, volatile main memory (RAM), and hundreds or thousands of megabytes of slow, cheap, nonvolatile disk storage. It is the job of the operating system to coordinate how these memories are used.

The part of the operating system that manages the memory hierarchy is called the **memory manager**. Its job is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and deallocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes.

In this chapter we will investigate a number of different memory management schemes, ranging from very simple to highly sophisticated. We will start at the beginning and look first at the simplest possible memory management system and gradually progress to more and more elaborate ones.

4.1 Basic Memory Management $\surd$

Memory management systems can be divided into two basic classes: those that move processes back and forth between main memory and disk during execution (swapping and paging), and those that do not. The latter are simpler, so we will study them first. Later in the chapter we will examine swapping and paging. Throughout this chapter the reader should keep in mind that swapping and paging are largely artifacts caused by the lack of sufficient main memory to hold all programs and data at once. As main memory gets cheaper, the arguments in favor of one kind of memory management scheme or another may become obsolete—unless programs get bigger faster than memory gets cheaper.

4.1.1 Monoprogramming without Swapping or Paging

The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the operating system. Three variations on this theme are shown in Fig. 4-1. The operating system may be at the bottom of memory in RAM (Random Access Memory), as shown in Fig. 4-1(a), or it may be in ROM (Read-Only Memory) at the top of memory, as shown in Fig. 4-1(b), or the device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below, as shown in Fig. 4-1(c). The latter model is used by small MS-DOS systems, for example. On IBM PCs, the portion of the system in the ROM is called the **BIOS** (Basic Input Output System).

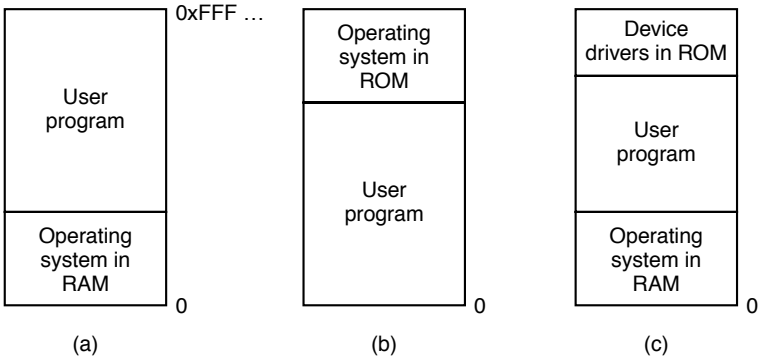


Figure 4-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

When the system is organized in this way, only one process at a time can be running. As soon as the user types a command, the operating system copies the requested program from disk to memory and executes it. When the process finishes, the operating system displays a prompt character and waits for a new command. When it receives the command, it loads a new program into memory, overwriting the first one.

4.1.2 Multiprogramming wiith Fixed Partitions

Although monoprogramming is sometimes used on small computers with simple operating systems, often it is desirable to allow multiple processes to run at once. On timesharing systems, having multiple processes in memory at once means that when one process is blocked waiting for I/O to finish, another one can use the CPU. Thus multiprogramming increases the CPU utilization. However, even on personal computers it is often useful to be able to run two or more programs at once.

The easiest way to achieve multiprogramming is simply to divide memory up into n (possibly unequal) partitions. This partitioning can, for example, be done manually when the system is started up.

When a job arrives, it can be put into the input queue for the smallest partition large enough to hold it. Since the partitions are fixed in this scheme, any space in a partition not used by a job is lost. In Fig. 4-2(a) we see how this system of fixed partitions and separate input queues looks.

The disadvantage of sorting the incoming jobs into separate queues becomes apparent when the queue for a large partition is empty but the queue for a small partition is full, as is the case for partitions 1 and 3 in Fig. 4-2(a). An alternative organization is to maintain

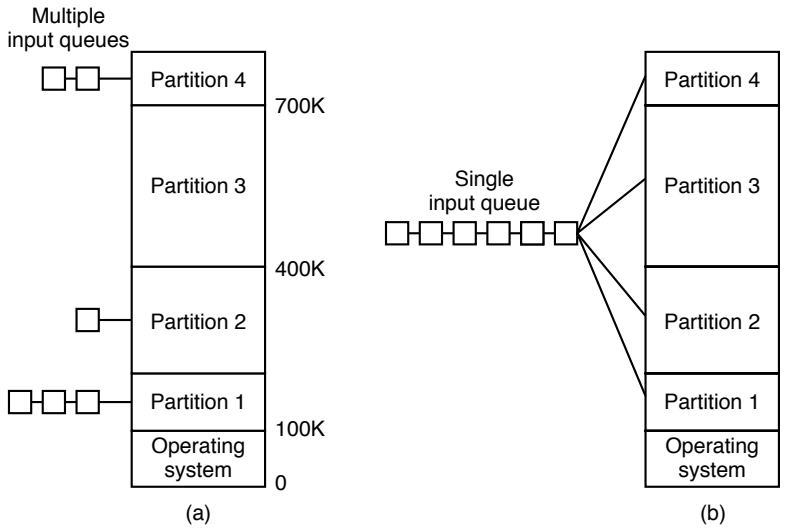


Figure 4-2. (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

a single queue as in Fig. 4-2(b). Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run. Since it is undesirable to waste a large partition on a small job, a different strategy is to search the whole input queue whenever a partition becomes free and pick the largest job that fits. Note that the latter algorithm discriminates against small jobs as being unworthy of having a whole partition, whereas usually it is desirable to give the smallest jobs (assumed to be interactive jobs) the best service, not the worst.

One way out is to have at least one small partition around. Such a partition will allow small jobs to run without having to allocate a large partition for them. Another approach is to have a rule stating that a job that is eligible to run may not be skipped over more than k times. Each time it is skipped over, it gets one point. When it has acquired k points, it may not be skipped again.

This system, with fixed partitions set up by the operator in the morning and not changed thereafter, was used by OS/360 on large IBM mainframes for many years. It was called **MFT** (Multiprogramming with a Fixed number of Tasks or OS/MFT). It is simple to understand and equally simple to implement: incoming jobs are queued until a suitable partition is available, at which time the job is loaded into that partition and run until it terminates. Nowadays, few, if any, operating systems, support this model.

Relocation and Protection

Multiprogramming introduces two essential problems that must be solved—relocation and protection. Look at Fig. 4-2. From the figure it is clear that different jobs will be run at different addresses. When a program is linked (i.e., the main program, user-written procedures, and library procedures are combined into a single address space), the linker must know at what address the program will begin in memory.

For example, suppose that the first instruction is a call to a procedure at absolute address 100 within the binary file produced by the linker. If this program is loaded in partition 1, that instruction will jump to absolute address 100, which is inside the operating system. What is needed is a call to l00K + 100. If the program is loaded into partition 2, it must be carried out as a call to 200K + 100, and so on. This problem is known as the

relocation problem.

One possible solution is to actually modify the instructions as the program is loaded into memory. Programs loaded into partition 1 have 100K added to each address, programs loaded into partition 2 have 200K added to addresses, and so forth. To perform relocation during loading like this, the linker must include in the binary program a list or bit map telling which program words are addresses to be relocated and which are opcodes, constants, or other items that must not be relocated. OS/MFT worked this way. Some microcomputers also work like this.

Relocation during loading does not solve the protection problem. A malicious program can always construct a new instruction and jump to it; Because programs in this system use absolute memory addresses rather than addresses relative to a register, there is no way to stop a program from building an instruction that reads or writes any word in memory. In multiuser systems, it is undesirable to let processes read and write memory belonging to other users.

The solution that IBM chose for protecting the 360 was to divide memory into blocks of 2K bytes and assign a 4-bit protection code to each block. The PSW contained a 4-bit key. The 360 hardware trapped any attempt by a running process to access memory whose protection code differed from the PSW key. Since only the operating system could change the protection codes and key, user processes were prevented from interfering with one another and with the operating system itself.

An alternative solution to both the relocation and protection problems is to equip the machine with two special hardware registers, called the **base** and **limit** registers. When a process is scheduled, the base register is loaded with the address of the start of its partition, and the limit register is loaded with the length of the partition. Every memory address generated automatically has the base register contents added to it before being sent to memory. Thus if the base register is 100K, a CALL 100 instruction is effectively turned into a CALL 100K + 100 instruction, without the instruction itself being modified. Addresses are also checked against the limit register to make sure that they do not attempt to address memory outside the current partition. The hardware protects the base and limit registers to prevent user programs from modifying them.

The CDC 6600—the world's first supercomputer—used this scheme. The Intel 8088 CPU used for the original IBM PC used a weaker version of this scheme—base registers, but no limit registers. Starting with the 286, a better scheme was adopted.

4.2 Swapping $\surd$

With a batch system, organizing memory into fixed partitions is simple and effective. Each job is loaded into a partition when it gets to the head of the queue. It stays in memory until it has finished. As long as enough jobs can be kept in memory to keep the CPU busy all the time, there is no reason to use anything more complicated.

With timesharing systems or graphically oriented personal computers, the situation is different. Sometimes there is not enough main memory to hold all the currently active processes, so excess processes must be kept on disk and brought in to run dynamically.

Two general approaches to memory management can be used, depending (in part) on the available hardware. The simplest strategy, called **swapping**, consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk. The other strategy, called **virtual memory**, allows programs to run even when they are only partially in main memory. Below we will studying swapping; in Sec. 4-3 we will examine

virtual memory.

The operation of a swapping system is illustrated in Fig. 4-3. Initially only process *A* is in memory. Then processes *B* and *C* are created or swapped in from disk. In Fig. 4-3(d) *A* terminates or is swapped out to disk. Then *D* comes in and *B* goes out. Finally *E* comes in.

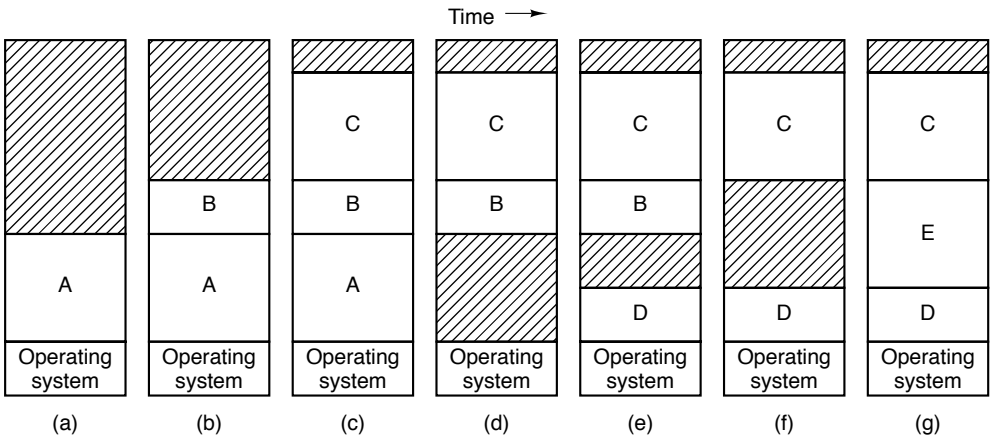


Figure 4-3. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

The main difference between the fixed partitions of Fig. 4-2 and the variable partitions of Fig. 4-3 is that the number, location, and size of the partitions vary dynamically in the latter as processes come and go, whereas they are fixed in the former. The flexibility of not being tied to a fixed number of partitions that may be too large or too small improves memory utilization, but it also complicates allocating and deallocating memory, as well as keeping track of it.

When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible. This technique is known as **memory compaction**. It is usually not done because it requires a lot of CPU time. For example, on a 32-MB machine that can copy 16 bytes per microsecond, it takes 2 sec to compact all of memory.

A point that is worth making concerns how much memory should be allocated for a process when it is created or swapped in. If processes are created with a fixed size that never changes, then the allocation is simple: you allocate exactly what is needed, no more and no less.

If, however, processes' data segments can grow, for example, by dynamically allocating memory from a heap, as in many programming languages, a problem occurs whenever a process tries to grow. If a hole is adjacent to the process, it can be allocated and the process allowed to grow into the hole. On the other hand, if the process is adjacent to another process, the growing process will either have to be moved to a hole in memory large enough for it, or one or more processes will have to be swapped out to create a large enough hole. If a process can not grow in memory and me swap area on the disk is full, the process will have to wait or be killed.

If it is expected that most processes will grow as they run, it is probably a good idea to allocate a little extra memory whenever a process is swapped in or moved, to reduce the overhead associated with moving or swapping processes that no longer fit in their allocated memory. However, when swapping processes to disk, only the memory actually in use should be swapped; it is wasteful to swap the extra memory as well. In Fig. 4-

4(a) we see a memory configuration in which space for growth has been allocated to two processes.

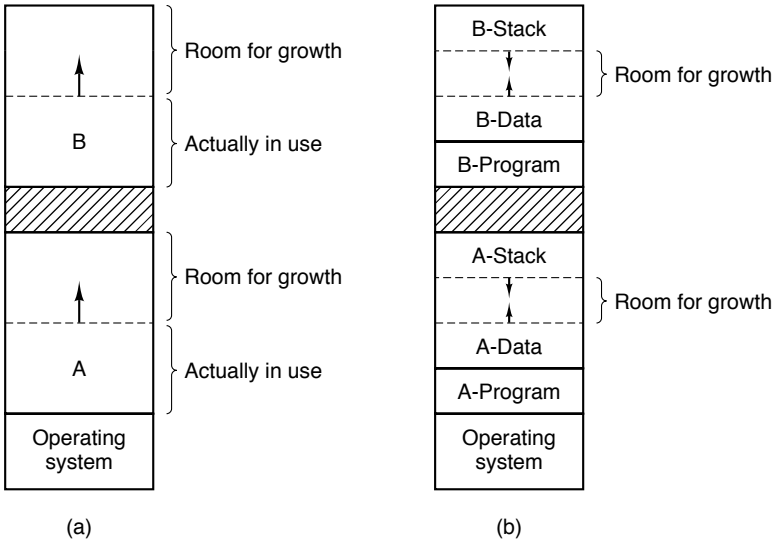


Figure 4-4. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

If processes can have two growing segments, for example, the data segment being used as a heap for variables that are dynamically allocated and released and a stack segment for the normal local variables and return addresses, an alternative arrangement suggests itself, namely that of Fig. 4-4(b). In this figure we see that each process illustrated has a stack at the top of its allocated memory that is growing downward, and a data segment just beyond the program text that is growing upward. The memory between them can be used for either segment. If it runs out, either the process will have to be moved to a hole with enough space, swapped out of memory until a large enough hole can be created, or killed.

4.2.1 Memory Management with Bit Maps

When memory is assigned dynamically, the operating system must manage it. In general terms, there are two ways to keep track of memory usage: bit maps and free lists. In this section and the next one we will look at these two methods in turn.

With a bit map, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes. Corresponding to each allocation unit is a bit in the bit map, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure 4-5 shows part of memory and the corresponding bit map.

The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bit map. However, even with an allocation unit as small as 4 bytes, 32 bits of memory will require only 1 bit of the map. A memory of $32n$ bits will use n map bits, so the bit map will take up only I/33 of memory. If the allocation unit is chosen large, the bit map will be smaller, but appreciable memory may be wasted in the last unit if the process size is not an exact multiple of the allocation unit.

A bit map provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bit map depends only on the size of memory and the

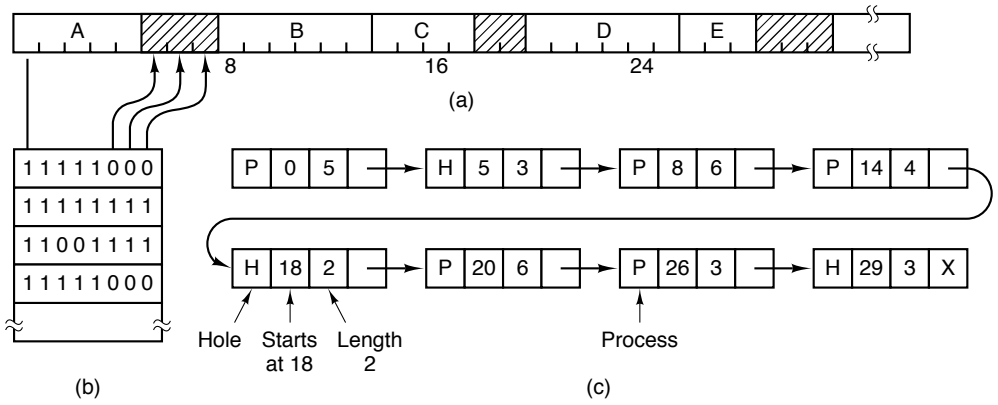


Figure 4-5. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bit map) are free. (b) The corresponding bit map. (c) The same information as a list.

size of the allocation unit. The main problem with it is that when it has been decided to bring a *k* unit process into memory, the memory manager must search the bit map to find a run of *k* consecutive O bits in the map. Searching a bit map for a run of a given length is a slow operation (because the run may straddle word boundaries in the map); this is an argument against bit maps.

4.2.2 Memory Management with Linked Lists

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes. The memory of Fig. 4-5(a) is represented in Fig. 4-5(c) as a linked list of segments. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry.

In this example, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward. A terminating process normally has two neighbors (except when it is at the very top or bottom of memory). These may be either processes or holes, leading to the four combinations of Fig. 4-6. In Fig. 4-6(a) updating the list requires replacing a P by an H. In Fig. 4-6(b) and Fig. 4-6(c), two entries are coalesced into one, and the list becomes one entry shorter. In Fig. 4-6(d), three entries are merged and two items are removed from the list. Since the process table slot for the terminating process will normally point to the list entry for the process itself, it may be more convenient to have the list as a double-linked list, rather than the single-linked list of Fig. 4-5(c). This structure makes it easier to find the previous entry and to see if a merge is possible.

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created or swapped in process. We assume that the memory manager knows how much memory to allocate. The simplest algorithm is **first fit**. The memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

A minor variation of first fit is **next fit**. it works the same way as first fit, except that it

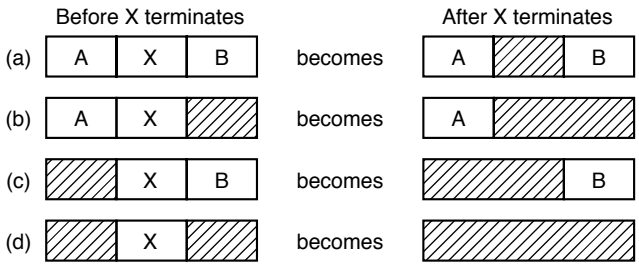


Figure 4-6. Four neighbor combinations for the terminating process, X.

keeps track of where it is whenever it finds a suitable hole. The next V time it is called to find a hole, it starts searching the list from the place where it left off last time. instead of always at the beginning, as first fit does. Simulations. by Bays (1977) show that next fit gives slightly worse performance than first fit.

Another well-known algorithm is **best fit**. Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.

As an example of first fit and best fit, consider Fig. 4-5 again. If a block of size 2 is needed, first fit will allocate the hole at 5, but best fit will allocate the hole at 18.

Best fit is slower than first fit because it must search the entire list every time it is called. Somewhat surprisingly, it also results in more wasted memory than first fit or next fit because it tends to fill up memory with tiny, useless holes. First fit generates larger holes on the average.

To get around the problem of breaking up nearly exact matches into a process and a tiny hole, one could think about **worst fit**, that is, always take the largest available hole, so that the hole broken off will be big enough to be useful. simulation has shown that worst fit is not a very good idea either.

All four algorithms can be speeded up by maintaining separate lists for processes and holes. In this way, all of them devote their full energy to inspecting holes, not processes. The inevitable price that is paid for this speedup on allocation is the additional complexity and slowdown when deallocating memory, since a freed segment has to be removed from the process list and inserted into the hole list.

If distinct lists are maintained for processes and holes, the hole list may be kept sorted on size, to make best fit faster. When best fit searches a list of holes from smallest to largest, as soon as it finds a hole that fits, it knows that the hole is the smallest one that will do the job, hence the best fit. No further searching is needed, as it is with the single list scheme. With a hole list sorted by size, first fit and best fit are equally fast, and next fit is pointless.

When the holes are kept on separate lists from the processes, a small optimization is possible. Instead of having a separate set of data structures for maintaining the hole list, as is done in Fig. 4-5(c), the holes themselves can be used. The first word of each hole could be the hole size, and the second word a pointer to the following entry. The nodes of the list of Fig. 4-5(c), which require three words and one bit (P/H), are no longer needed.

Yet another allocation algorithm is **quick fit**, which maintains separate lists for some of the more common sizes requested. For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4K holes, the second entry is a pointer to a list of 8K holes, the third entry a pointer to l2K holes, and so on. Holes of say, 21K, could either be put on the 20K list or on a special list of odd-sized holes. With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as

all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge is possible is expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

4.3 Virtual Memory √

Many years ago people were first confronted with programs that were too big to fit in the available memory. The solution usually adopted was to split the program into pieces, called **overlays**. Overlay 0 would start running first. When it was done, it would call another overlay. Some overlay systems were highly complex, allowing multiple overlays in memory at once. The overlays were kept on the disk and swapped in and out of memory by the operating system, dynamically, as needed.

Although the actual work of swapping overlays in and out was done by the system, the decision of how to split the program into pieces had to be done by the programmer. Splitting up large programs into small, modular pieces was time consuming and boring. It did not take long before someone thought of a way to turn the whole job over to the computer.

The method that was devised (Fotheringham, 1961) has come to be known as **virtual memory** . The basic idea behind virtual memory is that the combined size of the program, data, and stack may exceed the amount of physical memory available for it. The operating system keeps those parts of the program currently in use in main memory, and the rest on the disk. For example, a 16MB program can run on a 4MB machine by carefully choosing which 4MB to keep in memory at each instant, with pieces of the program being swapped between disk and memory as needed.

Virtual memory can also work in a multiprogramming system, with bits and pieces of many programs in memory at once. While a program is waiting for part of itself to be brought in, it is waiting for I/O and cannot run, so the CPU can be given to another process, the same way as in any other multiprogramming system.

4.3.1 Paging

Most virtual memory systems use a technique called **paging**, which we will now describe. On any computer, there exists a set of memory addresses that programs can produce. When a program uses an instruction like

```
MOVE REG,1000
```

it is copying the contents of memory address 1000 to REG (or vice versa, depending on the computer). Addresses can be generated using indexing, base registers, segment registers, and other ways.

These program-generated addresses are called **virtual addresses** and form the **virtual address space**. On computers without virtual memory, the virtual address is put directly onto the memory bus and causes the physical memory word with the same address to be read or written. When virtual memory is used, the virtual addresses do not go directly to the memory bus. Instead, they go to a **Memory Management Unit** (MMU), a chip or collection of chips that maps the virtual addresses onto the physical memory addresses as illustrated in Fig. 4-7.

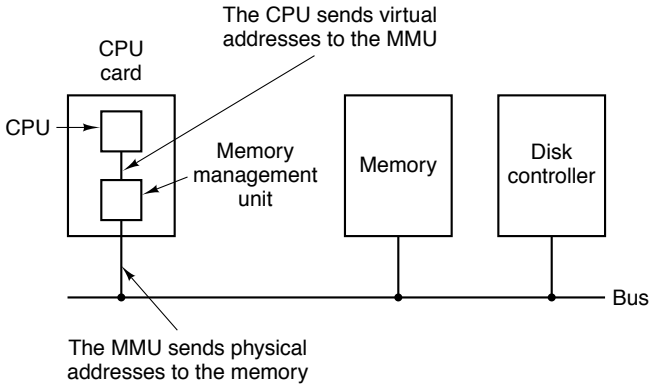


Figure 4-7. The position and function of the MMU.

A very simple example of how this mapping works is shown in Fig. 4-8. In this example, we have a computer that can generate 16-bit addresses, from 0 up to 64K. These are the virtual addresses. This computer, however, has only 32K of physical memory, so although 64k programs can be written, they cannot be loaded into memory in their entirety and run. A complete copy of a program's core image, up to 64K; must be present on the disk, however, so that pieces can be brought in as needed.

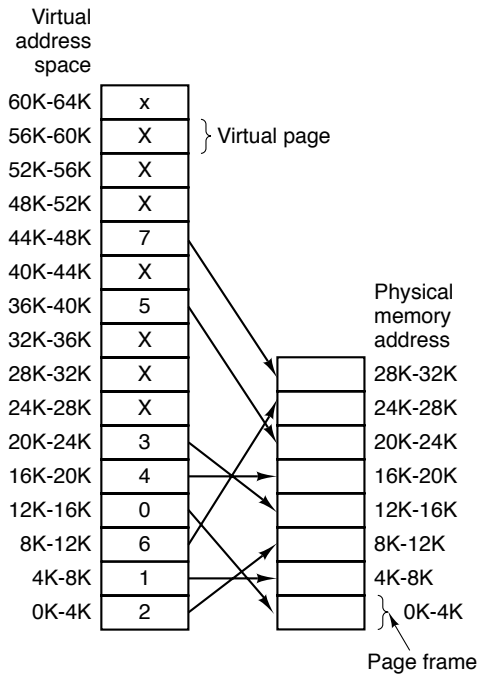


Figure 4-8. The relation between virtual addresses and physical memory addresses is given by the page table.

The virtual address space is divided up into units called **pages**. The corresponding units in the physical memory are called **page frames**. The pages and page frames are always exactly the same size. In this example they are 4k, but page sizes from 512 bytes to 64K are commonly used in existing systems. With 64K of virtual address space and 32K of physical memory, we have 16 virtual pages and 8 page frames. Transfers between memory and disk are always in units of a page.

When the program tries to access address 0, for example, using the instruction

```
MOVE REG,O
```

the virtual address O is sent to the MMU. The MMU sees that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287). It thus transforms the address to 8192 and outputs address 8192 onto the bus. The memory board knows nothing at all about the MMU and just sees a request for reading or writing address 8l92, which it honors. Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.

Similarly, an instruction

```
MOVE REG,8192
```

is effectively transformed into

```
MOVE REG,24576
```

because virtual address 8l92 is in virtual page 2 and this page is mapped onto physical page frame 6 (physical addresses 24576 to 28671). As a third example, virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address 12288 + 20 = 12308.

By itself, this ability to map the 16 virtual pages onto any of the eight page frames by setting the MMU's map appropriately does not solve the problem that the virtual address space is larger than the physical memory. Since we have only eight physical page frames, only eight of the virtual pages in Fig. 4-8 are mapped onto physical memory. The others, shown as a cross in the figure, are not mapped. In the actual hardware, a **Present/absent bit** in each entry keeps track of whether the page is mapped or not.

What happens if the program tries to use an unmapped page, for example, by using the instruction

```
MOVE REG,32780
```

which is byte 12 within virtual page 8 (starting at 3276S)? The MMU notices that the page is unmapped (indicated by a cross in the figure), and causes the CPU to trap to the operating system. This trap is called a **page fault**. The operating system picks a little-used page frame and writes its contents back to the disk. It then fetches the page just referenced into the page frame just freed, changes the map, and restarts the trapped instruction.

For example, if the operating system decided to evict page frame 1, it would load virtual page 8 at physical address 4K and make two changes to the MMU map. First, it would mark virtual page 1's entry as unmapped, to trap any future accesses to virtual addresses between 4K and 8K. Then it would replace the cross in virtual page 8's entry with a 1, so that when the trapped instruction is re-executed, it will map virtual address 32780 onto physical address 4108.

Now let us look inside the MMU to see how it works and why we have chosen to use a page size that is a power of 2. In Fig. 4-9 we see an example of a virtual address, 8196 (0011000000000100 in binary), being mapped using the MMU map of Fig. 4-8. The incoming 16-bit virtual address is split up into a 4-bit page number and a l2-bit offset. With 4 bits for the page number, we can represent 16 pages, and with 12 bits for the offset, we can address all 4096 bytes within a page.

The page number is used as an index into the **page table**, yielding the number of the page frame corresponding to that virtual page. If the *Present/absent* bit is 0, a trap to the operating system is caused. If the bit is 1, the page frame number found in the page

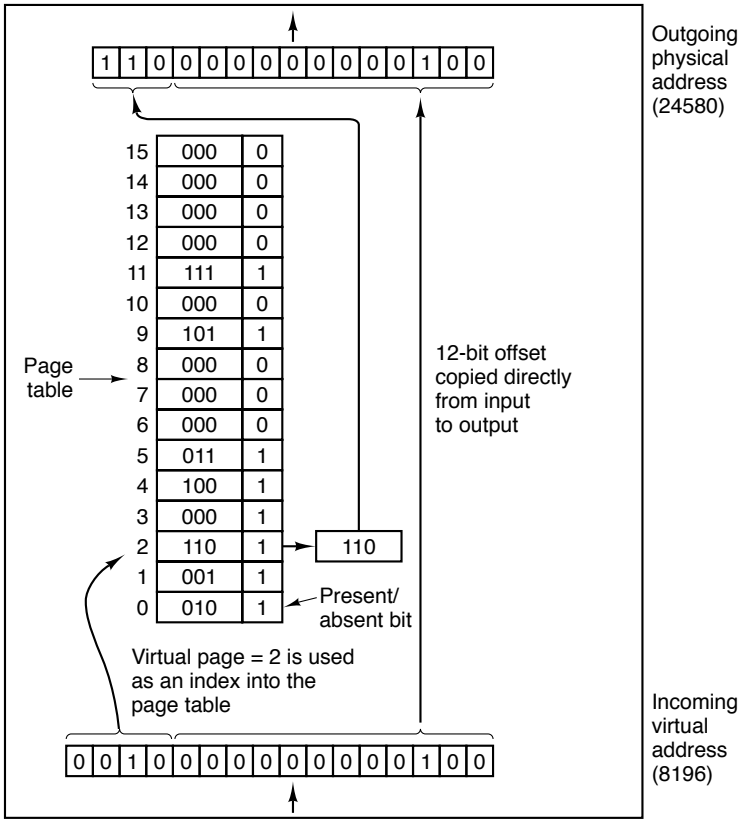


Figure 4-9. The internal operation of the MMU with 16 4K pages.

table is copied to the high-order 3 bits of the output register, along with the l2-bit offset, which is copied unmodified from the incoming virtual address. Together they form a 15-bit physical address. The output register is then put onto the memory bus as the physical memory address.

4.3.2 Page Tables

In theory, the mapping of virtual addresses onto physical addresses is as we have just described it. The virtual address is split into a virtual page number (high-order bits) and an offset (low-order bits). The virtual page number is used as an index into the page table to find the entry for that virtual page. From the page table entry, the page frame number (if any) is found. The page frame number is attached to the high-order end of the offset, replacing the virtual page number, to form a physical address that can be sent to the memory.

The purpose of the page table is to map virtual pages onto page frames. Mathematically speaking, the page table is a function, with the virtual page number as argument and the physical frame number as result. Using the result of this function, the virtual page field in a virtual address can be replaced by a page frame field, thus forming a physical memory address.

Despite this simple description, two major issues must be faced:

1. The page table can be extremely large.

2. The mapping must be fast.

The first point follows from the fact that modem computers use virtual addresses of at least 32 bits. With, say, a 4K page size, a 32-bit address space has 1 million pages, and a 64-bit address space has more than you want to contemplate. With 1 million pages in the virtual address space, the page table must have 1 million entries. And remember that each process needs its own page table.

The second point is a consequence of the fact that the virtual-to-physical mapping must be done on every memory reference. A typical instruction has an instruction word, and often a memory operand as well. Consequently, it is necessary to make 1, 2, or sometimes more page table references per instruction. If an instruction takes, say, 1O nsec, the page table lookup must be done in a few nano-seconds to avoid becoming a major bottleneck.

The need for large, fast page mapping is a significant constraint on the way computers are built. Although the problem is most serious with top-of-the-line machines, it is also an issue at the low end as well, where cost and price/performance are critical. In this section and the following ones, we will look at page table design in detail and show a number of hardware solutions that have been used in actual computers.

The simplest design (at least conceptually) is to have a single page table consisting of an array of fast hardware registers, with one entry for each virtual page, indexed by virtual page number. When a process is started up, the operating system loads the registers with the process' page table, taken from a copy kept in main memory. During process execution, no more memory references are needed for the page table. The advantages of this method are that it is straightforward and requires no memory references during mapping. A disadvantage is that it is potentially expensive (if the page table is large). Having to load the page table at every context switch can also hurt performance.

At the other extreme, the page table can be entirely in main memory. All the hardware needs then is a single register that points to the start of the page table. This design allows the memory map to be changed at a context switch by reloading one register. Of course, it has the disadvantage of requiring one or more memory references to read page table entries during the execution of each instruction. For this reason, this approach is rarely used in its most pure form, but below. we will study some variations that have much better performance.

Multilevel Page Tables

To get around the problem of having huge page tables in memory all the time, many computers use a multilevel page table. A simple example is shown in Fig. 4-10. In Fig. 4-10(a) we have a 32-bit virtual address that is partitioned into a 10-bit *PT1* field, a 10-bit *PT2* field, and a 12-bit *Offset* field. Since offsets are 12 bits, pages are 4K, and there are a total of 2^{20} of them.

The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time. In particular, those that are not needed should not be kept around. Suppose, for example, that a process needs l2 megabytes, the bottom 4 megabytes of memory for program text, the next 4 megabytes for data, and the top 4 megabytes for the stack. In between the top of the data and the bottom of the stack is a gigantic hole that is not used.

In Fig. 4-10(b) we see how the two-level page table works in this example. On the left we have the top-level page table, with 1024 entries, corresponding to the 10-bit *PT1* field. When a virtual address is presented to the MMU, it first extracts the *PT1* field and uses this value as an index into the top-level page table. Each of these 1024 entries represents

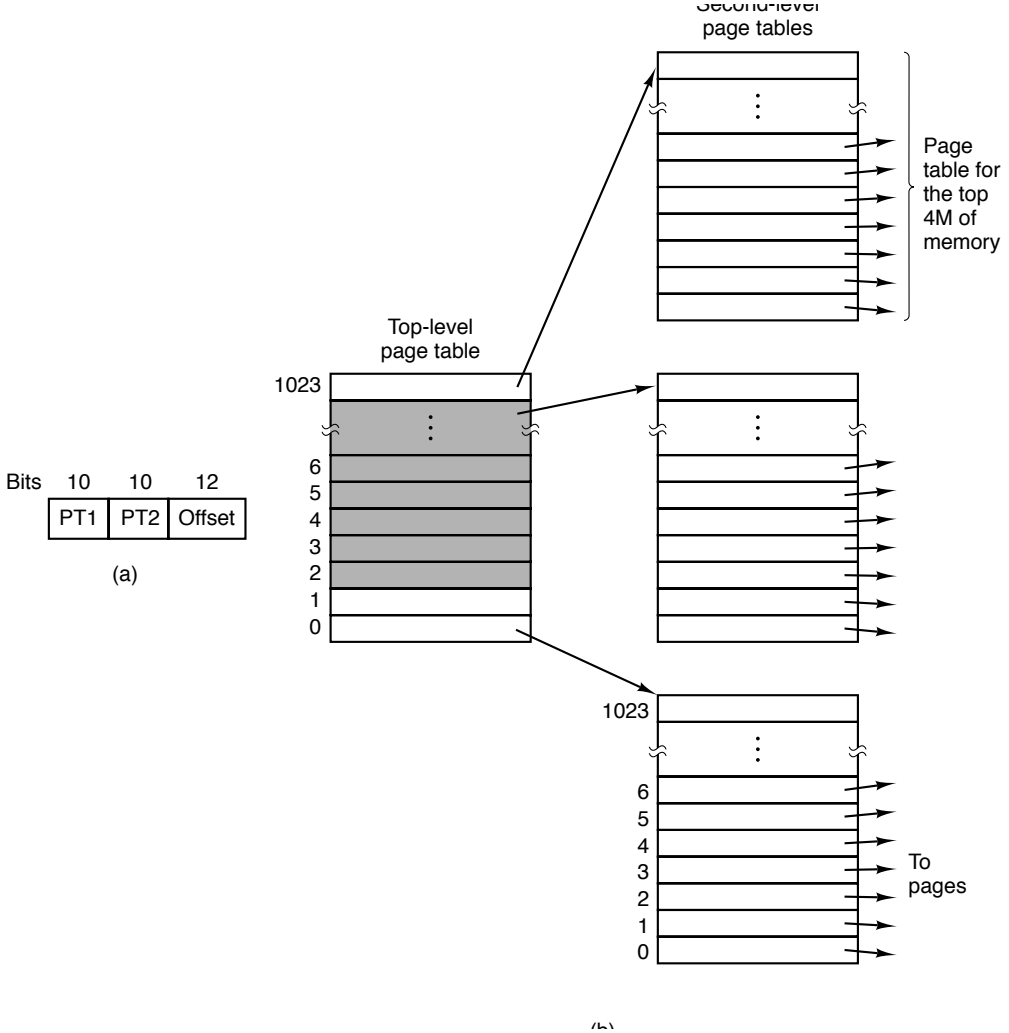


Figure 4-10. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

4M because the entire 4-gigabyte (i.e., 32- bit) virtual address space has been chopped into chunks of 1024 bytes.

The entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table. Entry 0 of the top-level page table points to the page table for the program text, entry 1 points to the page table for the data, and entry 1023 points to the page table for the stack. The other (shaded) entries are not used. The *PT2* field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

As an example, consider the 32-bit virtual address Ox00403004 (4,206,596 decimal), which is 12,292 bytes into the data. This address corresponds to *PT1* = 1, *PT2* = 3, and *Offset* = 4. The MMU first uses *PT1* to index into the top-level page table and obtain entry 1, which corresponds to addresses 4M to 8M. It then uses *PT2* to index into the second-level page table just found and extract entry 3, which corresponds to addresses 12288 to 16383 within its 4M chunk (i.e., absolute addresses 4,206,592 to 4,210,687). This entry contains the page frame number of the page containing virtual address Ox00403004. If that page is not in memory, the *Present/absent* bit in the page table entry will be zero, causing a page fault. If the page is in memory, the page frame number taken from the second-level page table is combined with the offset (4) to construct a physical address.

This address is put on the bus and sent to memory.

The interesting thing to note about Fig. 4-10 is that although the address space contains over a million pages, only four page tables are actually needed: the top-level table, and the second-level tables for 0 to 4M, 4M to 8M, and the top 4M. The *Present/absent* bits in l021 entries of the top-level page table are set to 0, forcing a page fault if they are ever accessed. Should this occur, the operating system will notice that the process is trying to reference memory that it is not supposed to and will take appropriate action, such as sending it a signal or killing it. In this example we have chosen round numbers for the various sizes and have picked *PT1* equal to *PT2* but in actual practice other values are also possible, of course.

The two-level page table system of Fig. 4-10 can be expanded to three, four, or more levels. Additional levels give more flexibility, but it is doubtful that the additional complexity is worth it beyond three levels.

Let us now turn from the structure of the page tables in the large, to the details of a single page table entry. The exact layout of an entry is highly machine dependent, but the kind of information present is roughly the same from machine to machine. In Fig. 4-11 we give a sample page table entry. The size varies from computer to computer, but 32 bits is a common size. The most important field is the *Page frame number*. After all, the goal of the page mapping is to locate this value. Next to it we have the *Present/absent* bit. If this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault.

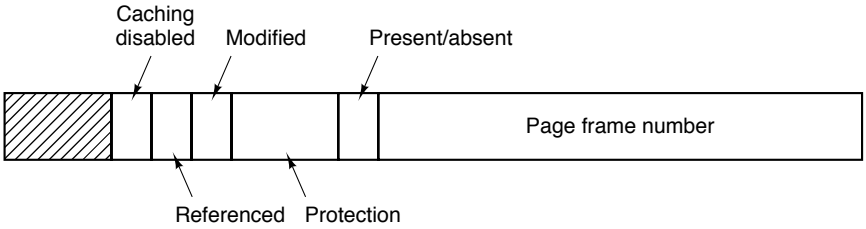


Figure 4-11. A typical page table entry.

The *Protection* bits tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only. A more sophisticated arrangement is having 3 bits, one bit each for enabling reading, writing, and executing the page.

The *Modified* and *Referenced* bits keep track of page usage. When a page is written to, the hardware automatically sets the *Modified* bit. This bit is of value when the operating system decides to reclaim a page frame. If the page in it has been modified (i.e., is "dirty"), it must be written back to the disk. If it has not. been modified (i.e., is "clean"), it can just be abandoned, since the disk copy is still valid. The bit is sometimes called the **dirty bit**, since it reflects the page's state.

The *Referenced* bit is set whenever a page is referenced, either for reading or writing. Its value is to help the operating system choose a page to evict when a page fault occurs. Pages that are not being used are better candidates than pages that are, and this bit plays an important role in several of the page replacement algorithms that we will study later in this chapter.

Finally, the last bit allows caching to be disabled for the page. This feature is important for pages that map onto device registers rather than memory. If the operating system is

sitting in a tight loop waiting for some I/O device to respond to a command it was just given, it is essential that the hardware keep fetching the word from the device, and not use an old cached copy. With this bit, caching can be turned off. Machines that have a separate I/O space and do not use memory mapped I/O do not need this bit.

Note that the disk address used to hold the page when it is not in memory is not part of the page table. The reason is simple. The page table holds only that information the hardware needs to translate a virtual address to a physical address. Information the operating system needs to handle page faults is kept in software tables inside the operating system.

4.3.3 TLBs—Translation Lookaside Buffers

In most paging schemes, the page tables are kept in memory, due to their large size. Potentially, this design has an enormous impact on performance. Consider, for example, an instruction that copies one register to another. In the absence of paging, this instruction makes only one memory reference, to fetch the instruction. With paging, additional memory references will be needed to access the page table. Since execution speed is generally limited by the rate the CPU can get instructions and data out of the memory, having to make two page table references per memory reference reduces performance by 2/3. Under these conditions, no one would use it.

Computer designers have known about this problem for years and have come up with a solution. Their solution is based on the observation that most programs tend to make a large number of references to a small number of pages, and not the other way around. Thus only a small fraction of the page table entries are heavily read; the rest are barely used at all.

The solution that has been devised is to equip computers with a small hardware device for mapping virtual addresses to physical addresses without going through the page table. The device, called a **TLB** (**Translation Lookaside Buffer**) or sometimes an **associative memory**, is illustrated in Fig. 4-12. It is usually inside the MMU and consists of a small number of entries, eight in this example, but rarely more than 64. Each entry contains information about one page. in particular, the virtual page number, a bit that is set when the page is modified, the protection code (read/write/execute permissions), and the physical page frame in which the page is located. These fields have a one-to-one correspondence with the fields in the page table. Another bit indicates whether the entry is valid (i.e., in use) or not.

Figure 4-12. A TLB to speed up paging.

An example that might generate the TLB of Fig. 4-12 is a process in a loop that spans virtual pages 19, 20, and 21, so these TLB entries have protection codes for reading and executing. The main data currently being used (say, an array being processed) are on pages 129 and 130. Page 140 contains the indices used in the array calculations. Finally, the stack is on pages 860 and 861.

Let us now see how the TLB functions. When a virtual address is presented to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries simultaneously (i.e., in parallel). If a valid match is found and the access does not violate the protection bits, the page frame is taken directly from the TLB, without going to the page table. If the virtual page number is

present in the TLB but the instruction is trying to write on a read-only page, a protection fault is generated, the same way as it would be from the page table itself.

The interesting case is what happens when the virtual page number is not in the TLB. The MMU detects the miss and does an ordinary page table lookup. It then evicts one of the entries from the TLB and replaces it with the page table entry just looked up. Thus if that page is used again soon, the second time it will result in a hit rather than a miss. When an entry is purged from the TLB, the modified bit is copied back into the page table entry in memory. The other values are already there. When the TLB is loaded from the page table, all the fields are taken from memory.

Software TLB Management

Up until now, we have assumed that every machine with paged virtual memory has page tables recognized by the hardware, plus a TLB. In this design, TLB management and handling TLB faults are done entirely by the MMU hardware. Traps to the operating system occur only when a page is not in memory.

In the past, this assumption was true. However, some modern RISC machines, including the MIPS, Alpha, and HP PA, do nearly all of this page management in software. On these machines, the TLB entries are explicitly loaded by the operating system. When a TLB miss occurs, instead of the MMU just going to the page tables to find and fetch the needed page reference, it just generates a TLB fault and tosses the problem into the lap of the operating system. The system must find the page, remove an entry from the TLB, enter the new one, and restart the instruction that faulted. And, of course, all of this must be done in a handful of instructions because TLB misses occur much more frequently than page faults.

Surprisingly enough, if the TLB is reasonably large (say, 64 entries) to reduce the miss rate, software management of the TLB turns out to be quite efficient. The main gain here is a much simpler MMU, which frees up a considerable amount of area on the CPU chip for caches and other features that can improve performance. Software TLB management is discussed at length by Uhlig et al. (1994).

Various strategies have been developed to improve performance on machines that do TLB management in software. One approach attacks both reducing TLB misses and reducing the cost of a TLB miss when it does occur (Bala et al., 1994). To reduce TLB misses, sometimes the operating system can use its intuition to figure out which pages are likely to be used next and to preload entries for them in the TLB. For example, when a client process does an RPC to a server process on the same machine, it is very likely that the server will have to run soon. Knowing this, while processing the trap to do the RPC, the system can also check to see where the server's code, data, and stack pages are, and map them in before they can cause TLB faults.

The normal way to process a TLB miss, whether in hardware or in software, is to go to the page table and perform the indexing operations to locate the page referenced. The problem with doing this search in software is that the pages holding the page table may not be in the TLB, which will cause additional TLB faults during the processing. These faults can be reduced by maintaining a large (e.g., 4K) software cache of TLB entries in a fixed location whose page is always kept in the TLB. By first checking the software cache, the operating system can substantially reduce TLB misses.

4.3.4 Inverted Page Tables

Traditional page tables of the type described so far require one entry per virtual page, since they are indexed by virtual page number. If the address space consists of 2^{32} bytes, with 4096 bytes per page, then over 1 million page table entries are needed. As a bare minimum, the page table will have to be at least 4 megabytes. On larger systems, this size is probably doable.

However, as 64-bit computers become more common, the situation changes drastically. If the address space is now 2^{64} bytes, with 4K pages, we need over 10^{15} bytes for the page table. Tying up 1 million gigabytes just for the page table is not doable, not now and not for decades to come, if ever. Consequently, a different solution is needed for 64-bit paged virtual address spaces.

One such solution is the **inverted page table**. In this design, there is one entry per page frame in real memory, rather than one entry per page of virtual address space. For example, with 64-bit virtual addresses, a 4K page, and 32 MB of RAM, an inverted page table only requires 8192 entries. The entry keeps track of which (process, virtual page) is located in the page frame.

Although inverted page tables save vast amounts of space, at least when the virtual address space is much larger than the physical memory, they have a serious downside: virtual-to-physical translation becomes much harder. When process n references virtual page p, the hardware can no longer find the physical page by using p as an index into the page table. Instead, it must search the entire inverted page table for an entry (n, p). Furthermore. this search must be done on every memory reference, not just on page faults. Searching an 8K table on every memory reference is not the way to make your machine blindingly fast.

The way out of this dilemma is to use the TLB. If the TLB can hold all of the heavily used pages, translation can happen just as fast as with regular page tables. On a TLB miss, however, the inverted page table has to be searched. Using a hash table as an index into the inverted page table, this search can be made reasonably fast, however. Inverted page tables are currently used on some IBM and Hewlett-Packard workstations and will become more common as 64-bit machines become widespread.

Other approaches to handling large virtual memories can be found in (Huck and Hays, 1993; Talluri and Hill, 1994; and Talluri et al., 1995).

4.4 Page Replacement Algorithms $\surd$

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted.

While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. Below we will describe some of the most important algorithms.

4.4.1 The Optimal Page Replacement Algorithm

The best possible page replacement algorithm is easy to describe but impossible to implement. It goes like this. At the moment that a page fault occurs, some set of pages is in memory. One of these pages will be referenced on the very next instruction (the page containing that instruction). Other pages may not be referenced until 10,100, or perhaps 1000 instructions later. Each page can be labeled with the number of instructions that will be executed before that page is first referenced.

The optimal page algorithm simply says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, removing the former pushes the page fault that will fetch it back as far into the future as possible. Computers, like people, try to put off unpleasant events for as long as they can.

The only problem with this algorithm is that it is unrealizable. At the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next. (We saw a similar situation earlier with the shortest job first scheduling algorithm—how can the system tell which job is shortest? Still, by running a program on a simulator and keeping track of all page references, it is possible to implement optimal page replacement on the *second* run by using the page reference information collected during the *first* run.

In this way it is possible to compare the performance of realizable algorithms with the best possible one. If an operating system achieves a performance of, say, only 1 percent worse than the optimal algorithm, effort spent in looking for a better algorithm will yield at most a 1 percent improvement.

To avoid any possible confusion, it should be made clear that this log of page references refers only to the one program just measured. The page replacement algorithm derived from it is thus specific to that one program. Although this method is useful for evaluating page replacement algorithms, it is of no use in practical systems. Below we will study algorithms that *are* useful on real systems.

4.4.2 The Not-Recently-Used Page Replacement Algorithm

In order to allow the operating system to collect useful statistics about which pages are being used and which ones are not, most computers with virtual memory have two status bits associated with each page. *R* is set whenever the page is referenced (read or written). *M* is set when the page is written to (i.e., modified). The bits are contained in each page table entry, as shown in Fig. 4-11. It is important to realize that these bits must be updated on every memory reference, so it is essential that they be set by the hardware. Once a bit has been set to 1, it stays 1 until the operating system resets it to 0 in software.

If the hardware does not have these bits, they can be simulated as follows. When a process is started up, all of its page table entries are marked as not in memory. As soon as any page is referenced, a page fault will occur. The operating system then sets the *R* bit (in its internal tables), changes the page table entry to point to the correct page, with mode READ ONLY, and restarts the instruction. If the page is subsequently written on, another page fault will occur, allowing the operating system to set the *M* bit and change the page‘s mode to READ/WRITE.

The *R* and *M* bits can be used to build a simple paging algorithm as follows. When a process is started up, both page bits for all its pages are set to 0 by the operating system. Periodically (e.g., on each clock interrupt), the *R* bit is cleared, to distinguish pages that

have not been referenced recently from those that have been.

When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their *R* and *M* bits:

Class 0: not referenced, not modified.
Class 1: not referenced, modified.
Class 2: referenced, not modified.
Class 3: referenced, modified.

Although class 1 pages seem, at first glance, impossible, they occur when a class 3 page has its *R* bit cleared by a clock interrupt. Clock interrupts do not clear the *M* bit because this information is needed to know whether the page has to be rewritten to disk or not.

The **NRU** (**Not Recently Used**) algorithm removes a page at random from the lowest numbered nonempty class. Implicit in this algorithm is that it is better to remove a modified page that has not been referenced in at least one clock tick (typically 20 msec) than a clean page that is in heavy use. The main attraction of NRU is that it is easy to understand, efficient to implement, and gives a performance that, while certainly not optimal, is often adequate.

4.4.3 The First-In, First-Out (FIFO) Page Replacement Algorithm

Another low-overhead paging algorithm is the **FIFO** (**First-In, First-Out**) algorithm. To illustrate how this works, consider a supermarket that has enough shelves to display exactly *k* different products. One day, some company introduces a new convenience food—instant, freeze-dried, organic yogurt that can be reconstituted in a microwave oven. It is an immediate success, so our finite supermarket has to get rid of one old product in order to stock it.

One possibility is to find the product that the supermarket has been stocking the longest (i.e.. something it began selling 120 years ago) and get rid of it on the grounds that no one is interested any more. In effect, the supermarket maintains a linked list of all the products it currently sells in the order they were introduced. The new one goes on the back of the list; the one at the front of the list is dropped.

As a page replacement algorithm, the same idea is applicable. The operating system maintains a list of all pages currently in memory, with the page at the head of the list the oldest one and the page at the tail the most recent arrival. On a page fault, the page at the head is removed and the new page added to the tail of the list. When applied to stores, FIFO might remove mustache wax, but it might also remove flour, salt, or butter. When applied to computers the same problem arises. For this reason, FIFO in its pure form is rarely used.

4.4.4 The Second Chance Page Replacement Algorithm

A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the *R* bit of the oldest page. If it is 0, the page is both old and unused, so it is replaced immediately. If the *R* bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.

The operation of this algorithm, called second chance, is shown in Fig. 4-13. In Fig. 4-13(a) we see pages *A* through *H* kept on a linked list and sorted by the time they arrived in memory.

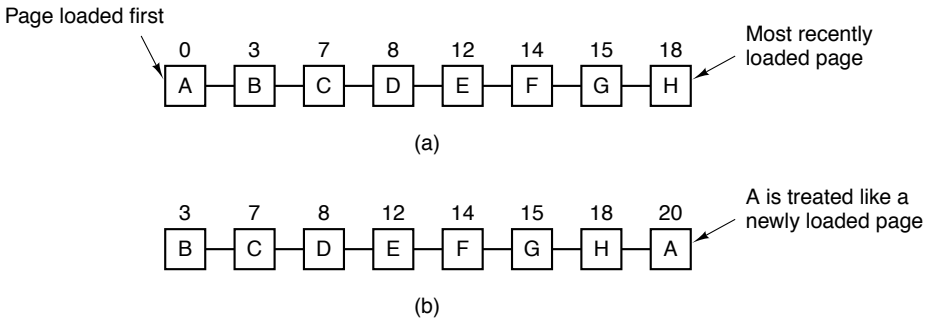


Figure 4-13. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set.

Suppose that a page fault occurs at time 20. The oldest page is *A*, which arrived at time 0, when the process started. If *A* has the *R* bit cleared, it is evicted from memory, either by being written to the disk (if it is dirty), or just abandoned (if it is clean). On the other hand, if the *R* bit is set, *A* is put onto the end of the list and its "load time" is reset to the current time (20). The *R* bit is also cleared. The search for a suitable page continues with *B*.

What second chance is doing is looking for an old page that has not been referenced in the previous clock interval. If all the pages have been referenced, second chance degenerates into pure FIFO. Specifically, imagine that all the pages in Fig. 4-13(a) have their *R* bits set. One by one, the operating system moves the pages to the end of the list, clearing the *R* bit each time it appends a page to the end of the list. Eventually, it comes back to page *A*, which now has its *R* bit cleared. At this point *A* is evicted. Thus the algorithm always terminates.

4.4.5 The Clock Page Replacement Algorithm

Although second chance is a reasonable algorithm, it is unnecessarily inefficient because it is constantly moving pages around on its list. A better approach is to keep all the pages on a circular list in the form of a clock, as shown in Fig. 4-14. A hand points to the oldest page.

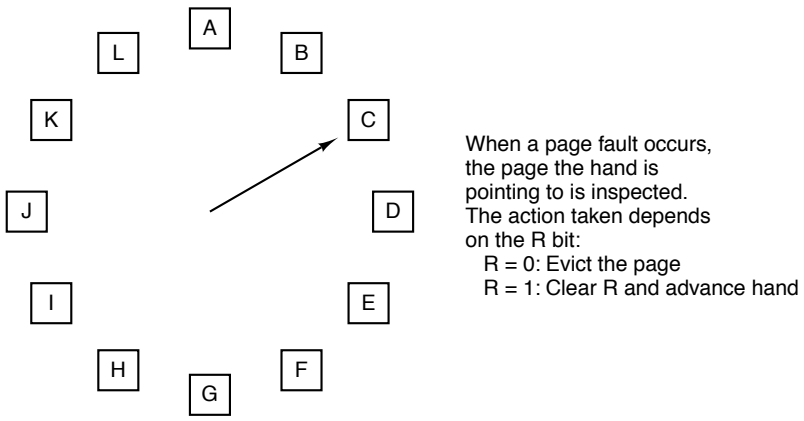


Figure 4-14. The clock page replacement algorithm.

When a page fault occurs, the page being pointed to by the hand is inspected. If its *R* bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the

hand is advanced one position. If R is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with $R = 0$. Not surprisingly, this algorithm is called **clock**. It differs from second chance only in the implementation.

4.4.6 The Least Recently Used (LRU) Page Replacement Algorithm

A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called **LRU** (**Least Recently Used**) paging.

Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation, even in hardware (assuming that such hardware could be built).

However, there are other ways to implement LRU with special hardware. Let us consider the simplest way first. This method requires equipping the hardware with a 64-bit counter, C, that is automatically incremented after each instruction. Furthermore, each page table entry must also have a field large enough to contain the counter. After each memory reference, the current value of C is stored in the page table entry for the page just referenced. When a page fault occurs, the operating system examines all the counters in the page table to find the lowest one. That page is the least recently used.

Now let us look at a second hardware LRU algorithm. For a machine with n page frames, the LRU hardware can maintain a matrix of $n \times n$ bits, initially all zero. Whenever page frame k is referenced, the hardware first sets all the bits of row k to 1, then sets all the bits of column k to 0. At any instant, the row whose binary value is lowest is the least recently used, the row whose value is next lowest is next least recently used, and so forth. The workings of this algorithm are given in Fig. 4-15 for four page frames and page references in the order

0 1 2 3 2 1 0 3 2 3

After page 0 is referenced we have the situation of Fig. 4-15(a), and so forth.

4.4.7 Simulating LRU in Software

Although both of the previous LRU algorithms are realizable in principle, few, if any, machines have this hardware, so they are of little use to the operating system designer who is making a system for a machine that does not have this hardware. Instead, a solution that can be implemented in software is needed. One possibility is called the **Not Frequently Used** or **NFU** algorithm. It requires a software counter associated with each page, initially zero. At each clock interrupt, the operating system scans all the pages in memory. For each page, the R bit, which is 0 or 1, is added to the counter. In effect, the counters are an attempt to keep track of how often each page has been referenced. When a page fault occurs, the page with the lowest counter is chosen for replacement.

The main problem with NFU is that it never forgets anything. For example, in a multipass compiler, pages that were heavily used during pass 1 may still have a high

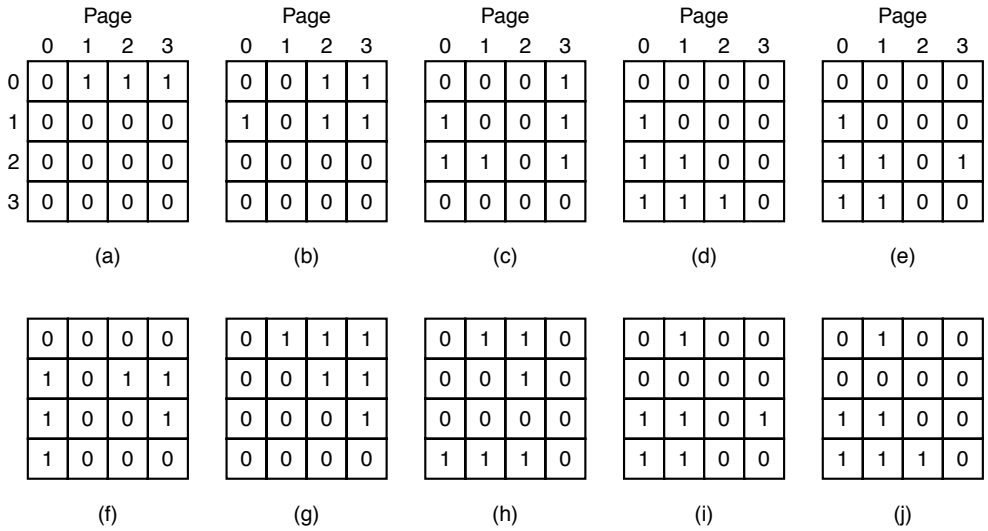


Figure 4-15. LRU using a matrix.

count well into later passes. In fact, if pass 1 happens to have the longest execution time of all the passes, the pages containing the code for subsequent passes may always have lower counts than the pass 1 pages. Consequently, the operating system will remove useful pages instead of pages no longer in use.

Fortunately, a small modification to NFU makes it able to simulate LRU quite well. The modification has two parts. First, the counters are each shifted right 1 bit before the *R* bit is added in. Second, the *R* bit is added to the leftmost, rather than the rightmost bit.

Figure 4-16 illustrates how the modified algorithm, known as **aging**, works. Suppose that after the first clock tick the *R* bits for pages 0 to 5 have the values 1, 0, 1, 0, 1, and 1 respectively (page 0 is 1, page 1 is 0, page 2 is 1, etc.). In other words, between tick 0 and tick 1, pages 0, 2, 4, and 5 were referenced, setting their *R* bits to 1, while the other ones remain 0. After the six corresponding counters have been shifted and the *R* bit inserted at the left, they have the values shown in Fig. 4-16(a). The four remaining columns show the six counters after the next four clock ticks.

When a page fault occurs, the page whose counter is the lowest is removed. It is clear that a page that has not been referenced for, say, four clock ticks will have four leading zeroes in its counter, and thus will have a lower value than a counter that has not been referenced for three clock ticks.

This algorithm differs from LRU in two ways. Consider pages 3 and 5 in Fig. 4-16(e). Neither has been referenced for two clock ticks; both were referenced in the tick prior to that. According to LRU, if a page must be replaced, we should choose one of these two. The trouble is, we do not know which of these two was referenced last in the interval between tick 1 and tick 2. By recording only one bit per time interval, we have lost the ability to distinguish references early in the clock interval from those occurring later. All we can do is remove page 3, because page 5 was also referenced two ticks earlier and page 3 was not.

The second difference between LRU and aging is that in aging the counters have a finite number of bits, 8 bits in this example. Suppose that two pages each have a counter value of 0. All we can do is pick one of them at random. In reality, it may well be that one of the pages was last referenced 9 ticks ago and the other was last referenced 1000 ticks ago. We have no way of seeing that. In practice, however, 8 bits is generally enough if a clock tick is around 20 msec. If a page has not been referenced in 160 msec, it probably

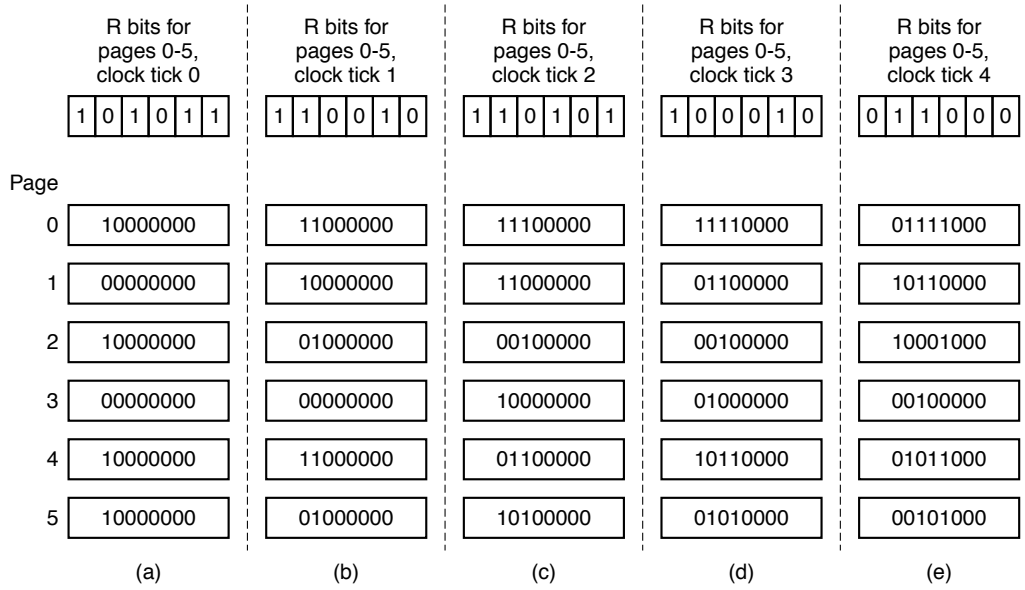


Figure 4-16. The aging algorithm simulates LRU in software. Shown are six pages for live clock ticks. The five clock ticks are represented by (a) to (e).

is not that important.

4.5 Design Issues for Paging Systems $\surd$

In the previous sections we have explained how paging works and have given a few of the basic page replacement algorithms and shown how to model them. But knowing the bare mechanics is not enough. To design a system, you have to know a lot more to make it work well. It is like the difference between knowing how to move the rook, knight, and other pieces in chess, and being a good player. In the following sections, we will look at other issues that operating system designers must consider in order to get good performance from a paging system.

4.5.1 The Working Set Model

In the purest form of paging, processes are started up with none of their pages in memory. As soon as the CPU tries to fetch the first instruction, it gets a page fault, causing the operating system to bring in the page containing the first instruction. Other page faults for global variables and the stack usually follow quickly. After a while, the process has most of the pages it needs and settles down to run with relatively few page faults. This strategy is called **demand paging** because pages are loaded only on demand, not in advance.

Of course, it is easy enough to write a test program that systematically reads all the pages in a large address space, causing so many page faults that there is not enough memory to hold them all. Fortunately, most processes do not work this way. They exhibit a **locality of reference**, meaning that during any phase of execution, the process references only a relatively small fraction of its pages. Each pass of a multipass compiler, for example, references only a fraction of all the pages, and a different fraction at that.

The set of pages that a process is currently using is called its **working set** (Denning, l968a; Denning, 1980). If the entire working set is in memory, the process will run without

causing many faults until it moves into another execution phase (e.g., the next pass of the compiler). If the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly since executing an instruction often takes a few nanoseconds and reading in a page from the disk typically takes tens of milliseconds. At a rate of one or two instructions per 20 milliseconds, it will take ages to finish. A program causing page faults every few instructions is said to be **thrashing** (Denning, 1968b).

In a timesharing system, processes are frequently moved to disk (i.e., all their pages are removed from memory) to let other processes have a turn at the CPU. The question arises of what to do when a process is brought back in again. Technically, nothing need be done. The process will just cause page faults until its working set has been loaded. The problem is that having 20, 50, or even 100 page faults every time a process is loaded is slow, and it also wastes considerable CPU time, since it takes the operating system a few milliseconds of CPU time to process a page fault.

Therefore, many paging systems try to keep track of each process' working set and make sure that it is in memory before letting the process run. This approach is called the **working set model** (Denning, 1970). It is designed to greatly reduce the page fault rate. Loading the pages before letting processes run is also called **prepaging**.

To implement the working set model, it is necessary for the operating system to keep track of which pages are in the working set. One way to monitor this information is to use the aging algorithm discussed above. Any page containing a 1 bit among the high order n bits of the counter is considered to be a member of the working set. If a page has not been referenced in n consecutive clock ticks, it is dropped from the working set. The parameter n has to be determined experimentally for each system, but the system performance is usually not especially sensitive to the exact value.

Information about the working set can be used to improve the performance of the clock algorithm. Normally, when the hand points to a page whose R bit is 0, the page is evicted. The improvement is to check to see if that page is part of the working set of the current process. If it is, the page is spared. This algorithm is called **wsclock**.

4.5.2 Local versus Global Allocation Policies

In the preceding sections we have discussed several algorithms for choosing a page to replace when a fault occurs. A major issue associated with this choice (which we have carefully swept under the rug until now) is how memory should be allocated among the competing runnable processes.

Take a look at Fig. 4-17(a). In this figure, three processes, A, B, and C, make up the set of runnable processes. Suppose A gets a page fault. Should the page replacement algorithm try to find the least recently used page considering only the six pages currently allocated to A, or should it consider all the pages in memory? If it looks only at A's pages, the page with the lowest age value is $A5$, so we get the situation of Fig. 4-17(b).

On the other hand, if the page with the lowest age value is removed without regard to whose page it is, page $B3$ will be chosen and we will get the situation of Fig. 4-17(c). The algorithm of Fig. 4-17(b) is said to be a **local** page replacement algorithm, whereas Fig. 4-17(c) is said to be a **global** algorithm. Local algorithms effectively correspond to allocating every process a fixed fraction of the memory. Global algorithms dynamically allocate page frames among the runnable processes. Thus the number of page frames assigned to each process varies in time.

In general, global algorithms work better, especially when the working set size can

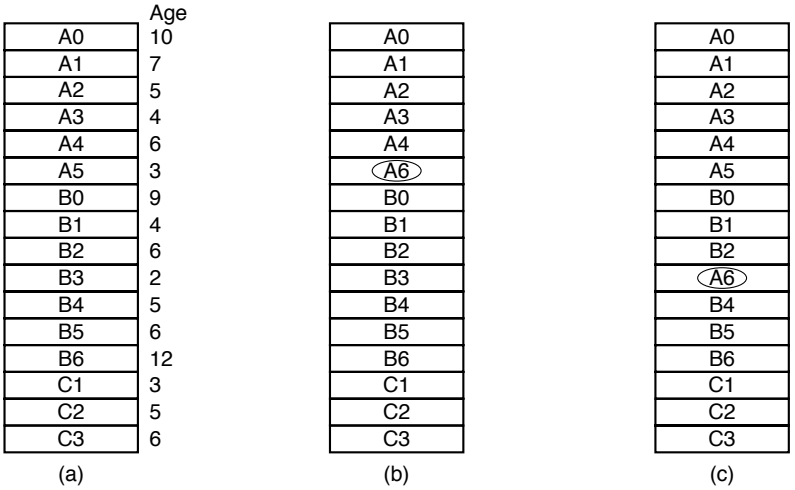


Figure 4-17. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

vary over the lifetime of a process. If a local algorithm is used and the working set grows, thrashing will result, even if there are plenty of free page frames. If the working set shrinks, local algorithms waste memory. If a global algorithm is used, the system must continually decide how many page frames to assign to each process. One way is to monitor the working set size as indicated by the aging bits, but this approach does not necessarily prevent thrashing. The working set may change size in microseconds, whereas the aging bits are a crude measure spread over a number of clock ticks.

Another approach is to have an algorithm for allocating page frames to processes. One way is to periodically determine the number of running processes and allocate each process an equal share. Thus with 475 available (i.e., non-operating system) page frames and 10 processes, each process gets 47 frames. The remaining 5 go into a pool to be used when page faults occur.

Although this method seems fair, it makes little sense to give equal shares of the memory to a 10K process and a 300K process. Instead, pages can be allocated in proportion to each process' total size, with a 300K process getting 30 times the allotment of a 10K process. It is probably wise to give each process some minimum number, so it can run, no matter how small it is. On some machines, for example, a single instruction may need as many as six pages because the instruction itself, the source operand. and the destination operand may all straddle page boundaries. With an allocation of only five pages, programs containing such instructions cannot execute at all.

Neither the equal allocation nor the proportional allocation method directly deals with the thrashing problem. A more direct way to control it is to use the **Page Fault Frequency** or **PFF** allocation algorithm. For a large class of page replacement algorithms, including LRU, it is known that the fault rate decreases as more pages are assigned, as we discussed above. This property is illustrated in Fig. 4-18.

The dashed line marked *A* corresponds to a page fault rate that is unacceptably high, so the faulting process is given more page frames to reduce the fault rate. The dashed line marked *B* corresponds to a page fault rate so low that it can be concluded that the process has too much memory. In this case page frames may be taken away from it. Thus, PFF tries to keep the paging rate within acceptable bounds.

If it discovers that there are so many processes in memory that it is not possible to keep

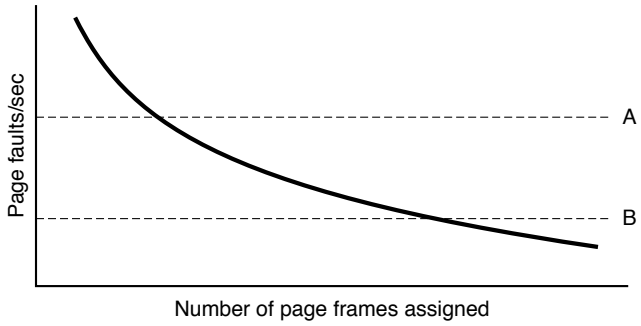


Figure 4-18. Page fault rate as a function of the number of page frames assigned.

all of them below *A*, then some process is removed from memory, and its page frames are divided up among the remaining processes or put into a pool of available pages that can be used on subsequent page faults. The decision to remove a process from memory is a form of load control. It shows that even with paging, swapping is still needed, only now swapping is used to reduce potential demand for memory, rather than to reclaim blocks of it for immediate use.

4.5.3 Page Size

The page size is often a parameter that can be chosen by the operating system, Even if the hardware has been designed with, for example, 512-byte pages, the operating system can easily regard pages O and 1, 2 and 3, 4 and 5, and so on, as 1K pages by always allocating two consecutive 512-byte page frames for them.

Determining the optimum page size requires balancing several competing factors. To start with, a randomly chosen text, data, or stack segment will not fill an integral number of pages. On the average, half of the final page will be empty. The extra space in that page is wasted This wastage is called **internal fragmentation**. With n segments in memory and a page size of p bytes, $np/2$ bytes will be wasted on internal fragmentation. This reasoning argues for a small page size.

Another argument for a small page size becomes apparent if we think about a program consisting of eight sequential phases of 4K each. With a 32K page size, the program must be allocated 32K all the time. With a 16K page size, it needs only 16K. With a page size of 4K or smaller, it requires only 4K at any instant. In general, a large page size will cause more unused program to be in memory than a small page size.

On the other hand, small pages mean that programs will need many pages, hence a large page table. A 32K program needs only four 8K pages, but 64 5l2-byte pages. Transfers to and from the disk are generally a page at a time, with most of the time being for the seek and rotational delay, so that transferring a small page takes almost as much time as transferring a large page. It might take 64×15 msec to load 64 512-byte pages, but only 4×25 msec to load four 8K pages.

On some machines, the page table must be loaded into hardware registers every time the CPU switches from one process to another. On these machines having a small page size means that the time required to load the page registers gets longer as the page size gets smaller. Furthermore, the space occupied by the page table increases as the page size decreases.

This last point can be analyzed mathematically. Let the average process size be s

bytes and the page size be p bytes. Furthermore, assume that each page entry requires e bytes. The approximate number of pages needed per process is then s/p, occupying se/p bytes of page table space. The wasted memory in the last page of the process due to internal fragmentation is $p/2$. Thus, the total overhead due to the page table and the internal fragmentation loss is given by

$$overhead = se/p + p/2$$

The first term (page table size) is large when the page size is small. The second term (internal fragmentation) is large when the page size is large. The optimum must lie somewhere in between. By taking the first derivative with respect to p and equating it to zero, we get the equation

$$-se/p^2 + 1/2 = 0$$

From this equation we can derive a formula that gives the optimum page size (considering only memory wasted in fragmentation and page table size). The result is:

$$p = \sqrt{2se}$$

For s = 128K and e = 8 bytes per page table entry, the optimum page size is l448 bytes. In practice 1K or 2K would be used, depending on the other factors (e.g., disk speed). Most commercially available computers use page sizes ranging from 512 bytes to 64K.

4.5.4 Virtual Memory Interface

Up until now, our whole discussion has assumed that virtual memory is transparent to processes and programmers, that is, all they see is a large virtual address space on a computer with a small(er) physical memory. With many systems, that is true, but in some advanced systems, programmers have some control over the memory map and can use it in nontraditional ways. In this section. we will briefly look at a few of these.

One reason for giving programmers control over their memory map is to allow two or more processes to share the same memory. If programmers can name regions of their memory, it may be possible for one process to give another process the name of a memory region so that process can also map it in. With two (or more) processes sharing the same pages, high bandwidth sharing becomes possible—one process writes into the shared memory and another one reads from it.

Sharing of pages can also be used to implement a high-performance message-passing system. Normally, when messages are passed, the data are copied from one address space to another, at considerable cost. If processes can control their page map, a message can be passed by having the sending process unmap the page(s) containing the message, and the receiving process mapping them in. Here only the page names have to be copied, instead of all the data.

Yet another advanced memory management technique is **distributed shared memory** (Feeley et al., 1995; Li and Hudak, 1989; Zekauskas et al., I994). The idea here is to allow multiple processes over a network to share a set of pages, possibly, but not necessarily, as a single shared linear address space. When a process references a page that is not currently mapped in, it gets a page fault. The page fault handler, which may be in the kernel or in user space, then locates the machine holding the page and sends it a message asking it to unmap the page and send it over the network. When the page arrives, it is mapped in and the faulting instruction is restarted.

4.6 Segmentation √

The virtual memory discussed so far is one-dimensional because the virtual addresses go from 0 to some maximum address, one address after another. For many problems, having two or more separate virtual address spaces may be much better than having only one. For example, a compiler has many tables that are built up as compilation proceeds, possibly including

1. The source text being saved for the printed listing (on batch systems).
2. The symbol table, containing the names and attributes of variables.
3. The table containing all the integer and floating-point constants used.
4. The parse tree. containing the syntactic analysis of the program.
5. The stack used for procedure calls within the compiler.

Each of the first four tables grows continuously as compilation proceeds. The last one grows and shrinks in unpredictable ways during compilation. In a one-dimensional memory, these five tables would have to be allocated contiguous chunks of virtual address space, as in Fig. 4-19.

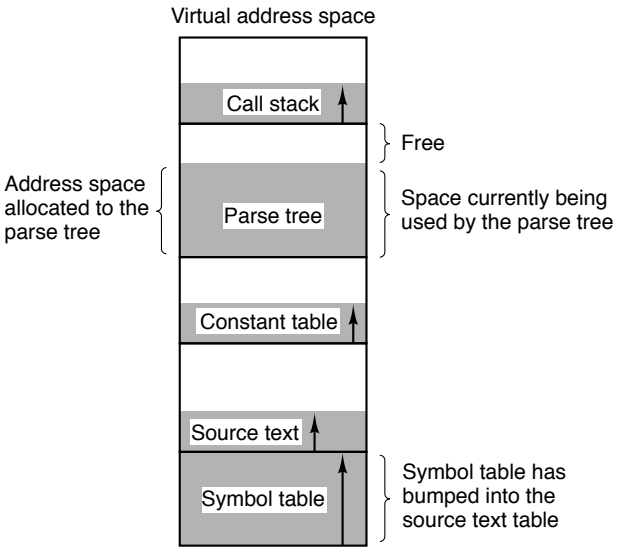


Figure 4-19. In a one-dimensional address space with growing tables, one table may bump into another.

Consider what happens if a program has an exceptionally large number of variables. The chunk of address space allocated for the symbol table may fill up, but there may be lots of room in the other tables. The compiler could, of course, simply issue a message saying that the compilation cannot continue due to too many variables. but doing so does not seem very sporting when unused space is left in the other tables.

Another possibility is to play Robin Hood, taking space from the tables with an excess of room and giving it to the tables with little room. This shuffling can be done, but it is analogous to managing one's own overlays—a nuisance at best and a great deal of tedious, unrewarding work at worst.

What is really needed is a way of freeing the programmer from having to manage the expanding and contracting tables, in the same way that virtual memory eliminates the worry of organizing the program into overlays.

A straightforward and extremely general solution is to provide the machine with many completely independent address spaces, called **segments**. Each segment consists of a linear sequence of addresses, from 0 to some maximum. The length of each segment may be anything from 0 to the maximum allowed. Different segments may, and usually do, have different lengths. Moreover, segment lengths may change during execution. The length of a stack segment may be increased whenever something is pushed onto the stack and decreased whenever something is popped off the stack.

Because each segment constitutes a separate address space, different segments can grow or shrink independently, without affecting each other. If a stack in a certain segment needs more address space to grow, it can have it, because there is nothing else in its address space to bump into. Of course, a segment can fill up but segments are usually very large, so this occurrence is rare. To specify an address in this segmented or two-dimensional memory, the program must supply a two-part address, a segment number, and an address within the segment. Figure 4-20 illustrates a segmented memory being used for the compiler tables discussed earlier.

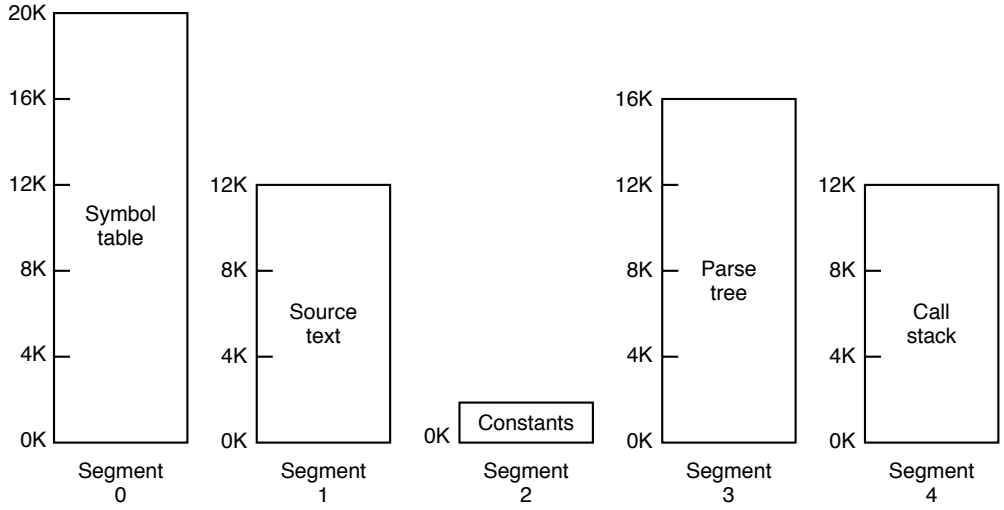


Figure 4-20. A segmented memory allows each table to grow or shrink independently of the other tables.

We emphasize that a segment is a logical entity, which the programmer is aware of and uses as a logical entity. A segment might contain a procedure, or an array, or a stack, or a collection of scalar variables, but usually it does not contain a mixture of different types.

A segmented memory has other advantages besides simplifying the handling of data structures that are growing or shrinking. If each procedure occupies a separate segment, with address 0 as its starting address, the linking up of procedures compiled separately is greatly simplified. After all the procedures that constitute a program have been compiled and linked up, a procedure call to the procedure in segment n will use the two-part address (n, 0) to address word 0 (the entry point).

If the procedure in segment n is subsequently modified and recompiled, no other procedures need be changed (because no starting addresses have been modified), even if the new version is larger than the old one. With a one-dimensional memory, the procedures

are packed tightly next to each other, with no address space between them. Consequently, changing one procedure's size can affect the starting address of other, unrelated procedures. This, in turn, requires modifying all procedures that call any of the moved procedures, in order to incorporate their new starting addresses. If a program contains hundreds of procedures, this process can be costly.

Segmentation also facilitates sharing procedures or data between several processes. A common example is the **shared library**. Modern workstations that run advanced window systems often have extremely large graphical libraries compiled into nearly every program. In a segmented system, the graphical library can be put in a segment and shared by multiple processes, eliminating the need for having it in every process' address space. While it is also possible to have shared libraries in pure paging systems, it is much more complicated. In effect, these systems do it by simulating segmentation.

Because each segment forms a logical entity of which the programmer is aware, such as a procedure, or an array, or a stack, different segments can have different kinds of protection. A procedure segment can be specified as execute only, prohibiting attempts to read from it or store into it. A floating-point array can be specified as read/write but not execute, and attempts to jump to it will be caught. Such protection is helpful in catching programming errors.

You should try to understand why protection makes sense in a segmented memory but not in a one-dimensional paged memory. In a segmented memory the user is aware of what is in each segment. Normally, a segment would not contain a procedure and a stack, for example, but one or the other. Since each segment contains only one type of object, the segment can have the protection appropriate for that particular type. Paging and segmentation are compared in Fig. 4-21.

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Figure 4-21. Comparison of paging and segmentation.

The contents of a page are, in a sense, accidental. The programmer is unaware of the fact that paging is even occurring. Although putting a few bits in each entry of the page table to specify the access allowed would be possible, to utilize this feature the programmer

would have to keep track of where in his address space the page boundaries were. That is precisely the sort of administration that paging was invented to eliminate. Because the user of a segmented memory has the illusion that all segments are in main memory all the time—that is, he can address them as though they were—he can protect each segment separately, without having to be concerned with the administration of overlaying them.

4.6.1 Implementation of Pure Segmentation

The implementation of segmentation differs from paging in an essential way: pages are fixed size and segments are not. Figure 4-22(a) shows an example of physical memory initially containing five segments. Now consider what happens if segment 1 is evicted and segment 7, which is smaller, is put in its place. We arrive at the memory configuration of Fig. 4-22(b). Between segment 7 and segment 2 is an unused area—that is, a hole. Then segment 4 is replaced by segment 5, as in Fig. 4-22(c), and segment 3 is replaced by segment 6, as in Fig. 4-22(d). After the system has been running for a while, memory will be divided up into a number of chunks, some containing segments and some containing holes. This phenomenon, called **checkerboarding** or **external fragmentation**, wastes memory in the holes. It can be dealt with by compaction, as shown in Fig. 4-22(e).

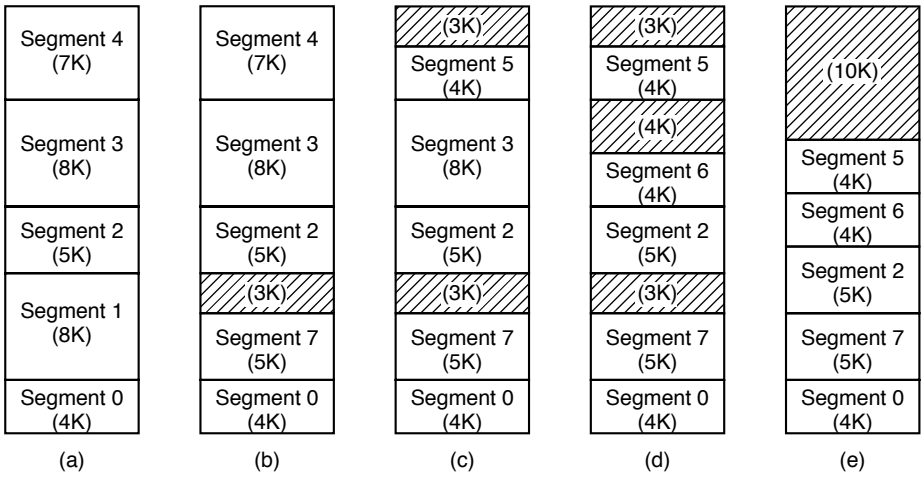


Figure 4-22. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

4.6.2 Segmentation with Paging: MULTICS

If the segments are large, it may be inconvenient, or even impossible, to keep them in main memory in their entirety. This leads to the idea of paging them, so that only those pages that are actually needed have to be around. Several significant systems have supported paged segments. In this section we will describe the first one: MULTICS. In the next one we will discuss a more recent one: the Intel Pentium.

MULTICS ran on the Honeywell 6000 machines and their descendants and provided each program with a virtual memory of up to 2^{18} segments (more than 250,000), each of which could be up to 65,536 (36-bit) words long. To implement this, the MULTICS designers chose to treat each segment as a virtual memory and to page it, combining the advantages of paging (uniform page size and not having to keep the whole segment in

memory if only part of it is being used) with the advantages of segmentation (case of programming, modularity, protection, and sharing).

Each MULTICS program has a segment table, with one descriptor per segment. Since there are potentially more than a quarter of a million entries in the table, the segment table is itself a segment and is paged. A segment descriptor contains an indication of whether the segment is in main memory or not. If any part of the segment is in memory, the segment is considered to be in memory, and its page table will be in memory. If the segment is in memory, its descriptor contains an 18-bit pointer to its page table [see Fig. 4-23(a)]. Because physical addresses are 24 bits and pages are aligned on 64-byte boundaries (implying that the low-order 6 bits of page addresses are 000000), only 18 bits are needed in the descriptor to store a page table address. The descriptor also contains the segment size, the protection bits, and a few other items. Figure 4-23(b) illustrates a MULTICS segment descriptor. The address of the segment in secondary memory is not in the segment descriptor but in another table used by the segment fault handler.

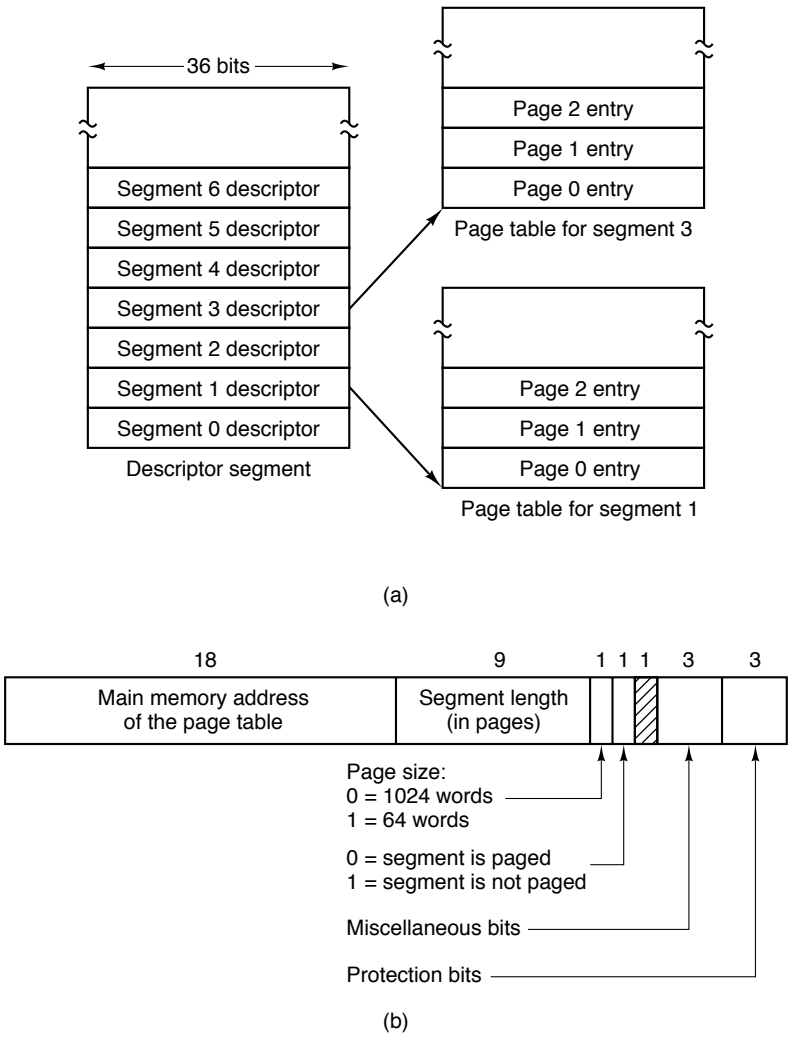


Figure 4-23. The MULTICS virtual memory. (a) The descriptor segment points to the page tables. (b) A segment descriptor. The numbers are the Held lengths.

Each segment is an ordinary virtual address space and is paged in the same way as the nonsegmented paged memory described earlier in this chapter. The normal page size is 1024 words (although a few small segments used by MULTICS itself are not paged or

are paged in units of 64 words to save physical memory).

An address in MULTICS consists of two parts: the segment and the address within the segment. The address within the segment is further divided into a page number and a word within the page, as shown in Fig. 4-24. When a memory reference occurs, the following algorithm is carried out.

1. The segment number is used to find the segment descriptor.
2. A check is made to see if the segment's page table is in memory. If the page table is in memory, it is located. If it is not, a segment fault occurs. If there is a protection violation, a fault (trap) occurs.
3. The page table entry for the requested virtual page is examined. If the page is not in memory, a page fault occurs. If it is in memory, the main memory address of the start of the page is extracted from the page table entry.
4. The offset is added to the page origin to give the main memory address where the word is located.
5. The read or store finally takes place.

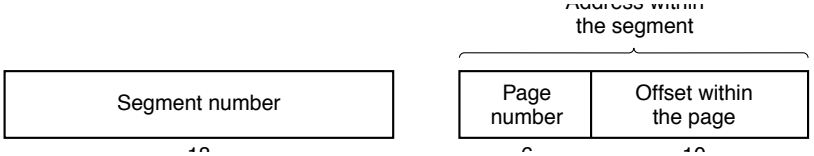


Figure 4-24. A 34-bit MULTICS virtual address.

This process is illustrated in Fig. 4-25. For simplicity, the fact that the descriptor segment is itself paged has been omitted. What really happens is that a register (the descriptor base register), is used to locate the descriptor segment's page table, which, in turn, points to the pages of the descriptor segment. Once the descriptor for the needed segment has been found, the addressing proceeds as shown in Fig. 4-25.

As you have no doubt guessed by now, if the preceding algorithm were actually carried out by the operating system on every instruction, programs would not run very fast. In reality, the MULTICS hardware contains a 16-word high-speed TLB that can search all its entries in parallel for a given key. It is illustrated in Fig. 4-26. When an address is presented to the computer, the addressing hardware first checks to see if the virtual address is in the TLB. If so, it gets the page frame number directly from the TLB and forms the actual address of the referenced word without having to look in the descriptor segment or page table. The addresses of the 16 most recently referenced pages are kept in the TLB.

Programs whose working set is smaller than the TLB size will come to equilibrium with the addresses of the entire working set in the TLB and therefore will run efficiently. If the page is not in the TLB, the descriptor and page tables are actually referenced to find the page frame address, and the TLB is updated to include this page, the least recently used page being thrown out. The age field keeps track of which entry is the least recently used. The reason that a TLB is used is for comparing the segment and page number of all the entries in parallel.

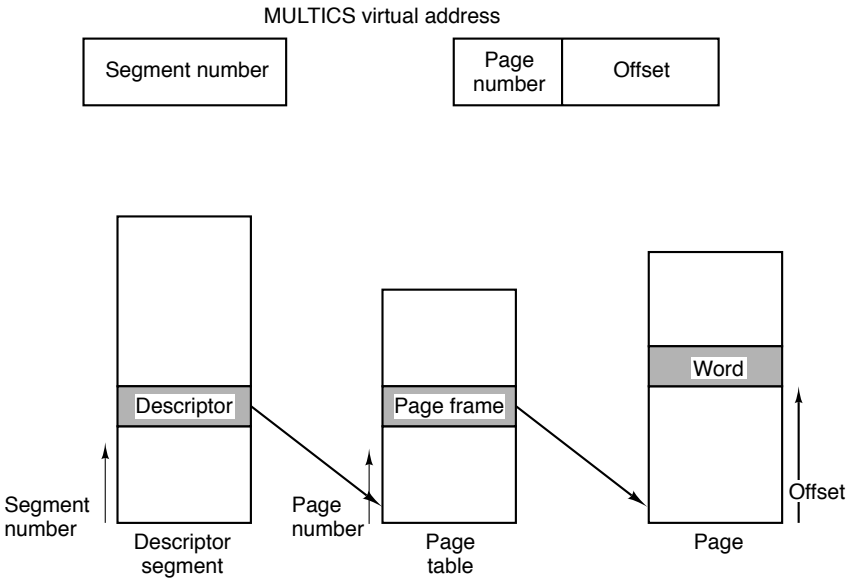


Figure 4-25. Conversion of a two-part MULTICS address into a main memory address.

Comparison field					Is this entry used?
Segment number	Virtual page	Page frame	Protection	Age	
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Figure 4-26. A simplified version of the MULTICS TLB. The existence of two page sizes makes the actual TLB more complicated.

4.6.3 Segmentation with Paging: The Intel Pentium

In many ways, the virtual memory on the Pentium (and Pentium Pro) resembles MULTICS, including the presence of both segmentation and paging. Whereas MULTICS has 256K independent segments, each up to 64K 36-bit words, the Pentium has 16K independent segments, each holding up to 1 billion 32-bit words. Although there are fewer segments, the larger segment size is far more important, as few programs need more than 1000 segments, but many programs need segments holding megabytes.

The heart of the Pentium virtual memory consists of two tables, the **LDT** (**Local Descriptor Table**) and the **GDT** (**Global Descriptor Table**). Each program has its own LDT, but there is a single GDT, shared by all the programs on the computer. The LDT describes segments local to each program, including its code, data, stack, and so on, whereas the GDT describes system segments, including the operating system itself.

To access a segment, a Pentium program first loads a selector for that segment into one of the machine's six segment-registers. During execution, the CS register holds the

selector for the code segment and the DS register holds the selector for the data segment. The other segment registers are less important. Each selector is a 16-bit number, as shown in Fig. 4-27.

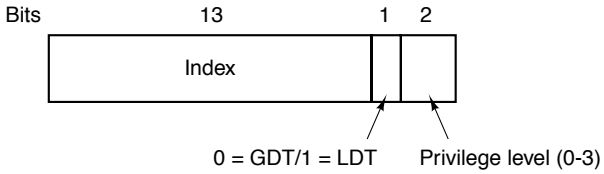


Figure 4-27. A Pentium selector.

One of the selector bits tells whether the segment is local or global (i.e., whether it is in the LDT or GDT). Thirteen other bits specify the LDT or GDT entry number, so these tables are each restricted to holding 8K segment descriptors. The other 2 bits relate to protection, and will be described later. Descriptor 0 is forbidden. It may be safely loaded into a segment register to indicate that the segment register is not currently available. It causes a trap if used.

At the time a selector is loaded into a segment register, the corresponding descriptor is fetched from the LDT or GDT and stored in microprogram registers, so it can be accessed quickly. A descriptor consists of 8 bytes, including the segment's base address, size, and other information, as depicted in Fig. 4-28.

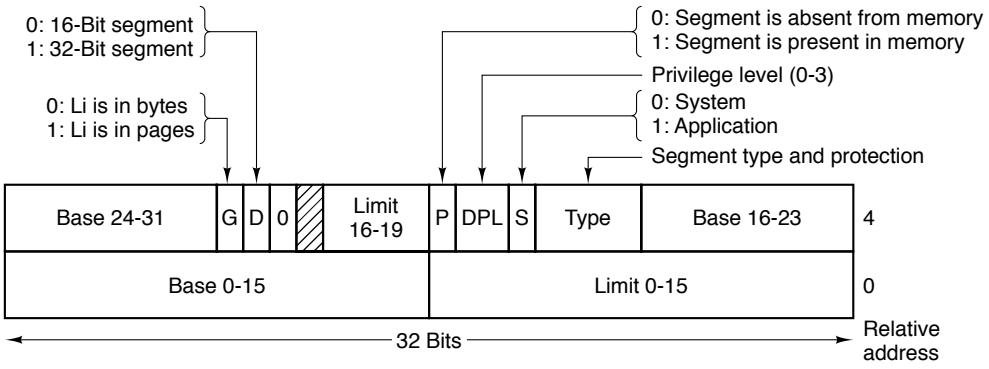


Figure 4-28. Pentium code segment descriptor. Data segments differ slightly.

The format of the selector has been cleverly chosen to make locating the descriptor easy. First either the LDT or GDT is selected, based on selector bit 2. Then the selector is copied to an internal scratch register, and the 3 low-order bits set to 0. Finally, the address of either the LDT or GDT table is added to it, to give a direct pointer to the descriptor. For example, selector 72 refers to entry 9 in the GDT, which is located at address GDT + 72.

Let us trace the steps by which a (selector, offset) pair is convened to a physical address. As soon as the microprogram knows which segment register is being used, it can find the complete descriptor corresponding to that selector in its internal registers. If the segment does not exist (selector 0), or is currently paged out, a trap occurs.

It then checks to see if the offset is beyond the end of the segment, in which case a trap also occurs. Logically, there should simply be a 32-bit field in the descriptor giving the size of the segment, but there are only 20 bits available, so a different scheme is used. If the *Gbit* (Granularity) field is O, the *Limit* field is the exact segment size, up to 1 MB.

If it is 1, the *Limit* field gives the segment size in pages instead of bytes. The Pentium page size is fixed at 4K bytes, so 20 bits are enough for segments up to 2^{32} bytes.

Assuming that the segment is in memory and the offset is in range, the Pentium then adds the 32-bit *Base* field in the descriptor to the offset to form what is called a **linear address**, as shown in Fig. 4-29. The *Base* field is broken up into three pieces and spread all over the descriptor for compatibility with the 286, in which the *Base* is only 24 bits. In effect, the *Base* field allows each segment to start at an arbitrary place within the 32-bit linear address space.

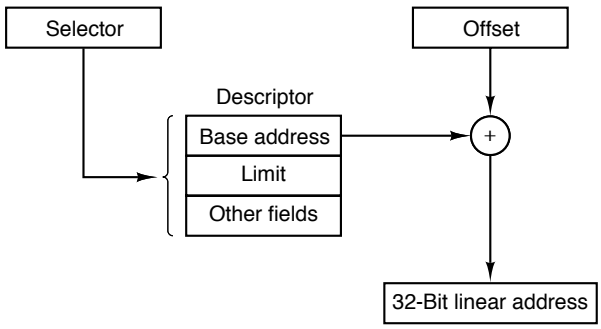


Figure 4-29. Conversion ofa (selector. offset) pair to a linear address.

If paging is disabled (by a bit in a global control register), the linear address is interpreted as the physical address and sent to the memory for the read or write. Thus with paging disabled, we have a pure segmentation scheme, with each segment's base address given in its descriptor. Segments are permitted to overlap, incidentally, probably because it would be too much trouble and take too much time to verify that they were all disjoint.

On the other hand, if paging is enabled, the linear address is interpreted as a virtual address and mapped onto the physical address using page tables, pretty much as in our earlier examples. The only real complication is that with a 32-bit virtual address and a 4K page, a segment might contain 1 million pages. so a two-level mapping is used to reduce the page table size for small segments.

Each running program has a **page directory** consisting of 1024 32-bit entries. It is located at an address pointed to by a global register. Each entry in this directory points to a page table also containing 1024 32-bit entries. The page table entries point to page frames. The scheme is shown in Fig. 4-30.

In Fig. 4-30(a) we see a linear address divided into three fields, *Dir*, *Page*, and *Off*. The *Dir* field is used to index into the page directory to locate a pointer to the proper page table. Then the *Page* field is used as an index into the page table to find the physical address of the page frame. Finally, *Off* is added to the address of the page frame to get the physical address of the byte or word needed.

The page table entries are 32 bits each, 20 of which contain a page frame number. The remaining bits contain access and dirty bits, set by the hardware for the benefit of the operating system, protection bits, and other utility bits.

Each page table has entries for 1024 4K page frames, so a single page table handles 4 megabytes of memory. A segment shorter than 4M will have a page directory with a single entry, a pointer to its one and only page table. In this way, the overhead for short segments is only two pages, instead of the million pages that would be needed in a one-level page table.

To avoid making repeated references to memory, the Pentium, like MULTICS, has a small TLB that directly maps the most recently used *Dir-Page* combinations onto the

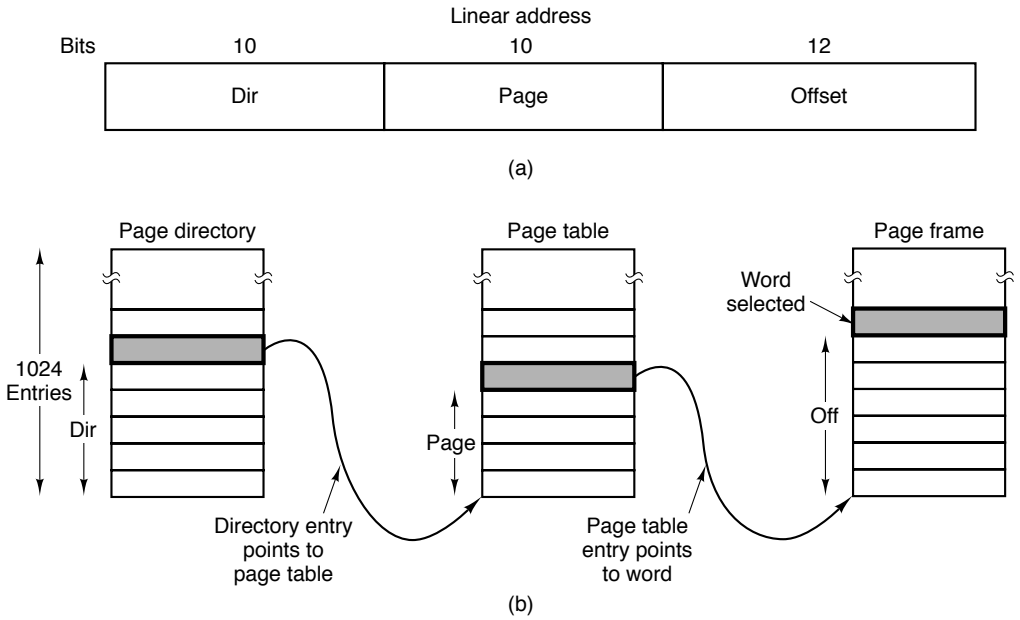


Figure 4-30. Mapping of a linear address onto a physical address.

physical address of the page frame. Only when the current combination is not present in the TLB is the mechanism of Fig. 4-30 actually carried out and the TLB updated.

A little thought will reveal the fact that when paging is used, there is really no point in having the *Base* field in the descriptor be nonzero. All that *Base* does is cause a small offset to use an entry in the middle of the page directory, instead of at the beginning. The real reason for including *Base* at all is to allow pure (non-paged) segmentation, and for compatibility with the 286, which always has paging disabled (i.e., the 286 has only pure segmentation, but not paging).

It is also worth noting that if some application does not need segmentation but is content with a single, paged, 32-bit address space, that model is possible. All the segment registers can be set up with the same selector, whose descriptor has *Base* = 0 and *Limit* set to the maximum. The instruction offset will then be the linear address, with only a single address space used—in effect, normal paging.

All in all, one has to give credit to the Pentium designers. Given the conflicting goals of implementing pure paging, pure segmentation, and paged segments, while at the same time being compatible with the 286, and doing all of this efficiently, the resulting design is surprisingly simple and clean.

Although we have covered the complete architecture of the Pentium virtual memory, albeit briefly, it is worth saying a few words about protection, since this subject is intimately related to the virtual memory. Just as the virtual memory scheme is closely modeled on MULTICS, so is the protection system. The Pentium supports four protection levels with level 0 being the most privileged and level 3 the least. These are shown in Fig. 4-31. At each instant, a running program is at a certain level, indicated by a 2-bit field in its PSW. Each segment in the system also has a level.

As long as a program restricts itself to using segments at its own level, everything works fine. Attempts to access data at a higher level are permitted. Attempts to access data at a lower level are illegal and cause traps. Attempts to call procedures at a different level (higher or lower) are allowed, but in a carefully controlled way. To make an interlevel call, the CALL instruction must contain a selector instead of an address. This selector

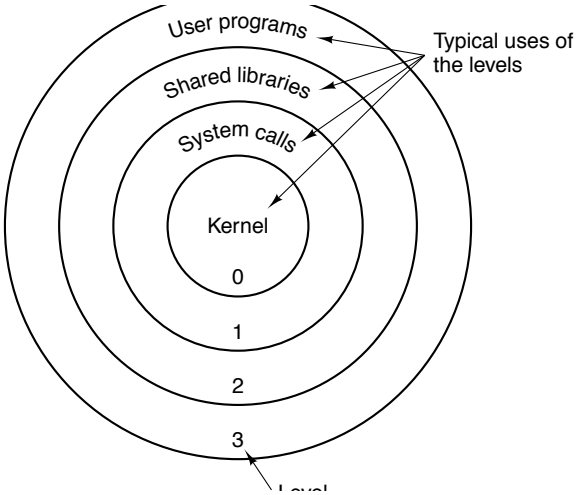


Figure 4-31. Protection on the Pentium.

designates a descriptor called a **call gate**, which gives the address of the procedure to be called. Thus it is not possible to jump into the middle of an arbitrary code segment at a different level. Only official entry points may be used. The concepts of protection levels and call gates were pioneered in MULTICS, where they were viewed as protection rings.

A typical use for this mechanism is suggested in Fig. 4-31. At level 0, we find the kernel of the operating system, which handles I/O, memory management, and other critical matters. At level 1, the system call handler is present. User programs may call procedures here to have system calls carried out, but only a specific and protected list of procedures may be called. Level 2 contains library procedures, possibly shared among many running programs. User programs may call these procedures and read their data, but they may not modify them. Finally, user programs run at level 3, which has the least protection.

Traps and interrupts use a mechanism similar to the call gates. They, too, reference descriptors, rather than absolute addresses, and these descriptors point to specific procedures to be executed. The *Type* field in Fig. 4-28 distinguishes between code segments, data segments, and the various kinds of gates.

4.7 Overview of Memory Management in MINIX

Memory management in MINIX is simple: neither paging or swapping is used. The memory manager maintains a list of holes sorted in memory address order. When memory is needed, either due to a FORK or an EXEC system call, the hole list is searched using first fit for a hole that is big enough. Once a process has been placed in memory, it remains in exactly the same place until it terminates. It is never swapped out and also never moved to another place in memory. Nor does the allocated area grow or shrink.

This strategy deserves some explanation. It derives from three factors: (1) the idea that MINIX is for personal computers, rather than for large timesharing systems. (2) the desire to have MINIX work on all IBM PCs. (3) a desire to make the system straightforward to implement on other small computers.

The first factor means that, on the average, the number of running processes will be small, so that typically enough memory will be available to hold all the processes with room left over. Swapping will not be needed then. Since it adds complexity to the system,

not swapping leads to simpler code.

The desire to have MINIX run on all IBM PC-compatible computers also had substantial impact on the memory management design. The simplest systems in this family use the 8086 processor, whose memory management architecture is very primitive. It does not support virtual memory in any form and does not even detect stack overflow, a defect that has major implications for the way processes are laid out in memory. These limitations do not exist in later designs which use the 80386, 80386, or Pentium processors. However, taking advantage of these features would make MINIX incompatible with many low-end machines that are still serviceable and in use.

The portability issue argues for as simple a memory management scheme as possible. If MINIX used paging or segmentation, it would be difficult, if not impossible, to port it to machines not having these features. By making a minimal number of assumptions about what the hardware can do, the number of machines to which MINIX can be ported is increased.

Another unusual aspect of MINIX is the way the memory management is implemented. It is not part of the kernel. Instead, it is handled by the memory manager process, which runs in user space and communicates with the kernel by the standard message mechanism. The position of the memory manager in the server level in shown in Fig. 2-26.

Moving the memory manager out of the kernel is an example of the separation of **policy** and **mechanism**. The decision about which process will be placed where in memory (policy) are made by the memory manager. The actual setting of memory maps for processes (mechanism) is done by the system task within the kernel. This split makes it relatively easy to change the memory management policy (algorithm, etc.) without having to modify the lowest layers of the operating system.

Most of the memory manager code is devoted to handling the MINIX system calls that involve memory management, primarily FORK and EXEC, rather than just manipulating lists of processes and holes. In the next section we will look at the memory layout, and in subsequent sections we will take a bird's-eye view of how the memory management system calls are processed by the memory manager.

4.7.1 Memory Layout

MINIX programs may be compiled to use combined I and D space, in which all parts of the process (text, data, and stack) share a block of memory which is allocated and released as one block. Processes can also be compiled to use separate I and D space. For clarity, allocation of memory for the simpler combined model will be discussed first. Processes using separate I and D space can use memory more efficiently, but taking advantage of this feature complicates things. We will discuss the complications after the simple case has been outlined.

Memory is allocated in MINIX on two occasions. First, when a process forks, the amount of memory needed by the child is allocated. Second, when a process changes its memory image via the EXEC system call, the space occupied by the old image is returned to the free list as a hole, and memory is allocated for the new image. The new image may be in a part of memory different from the released memory. Its location will depend upon where an adequate hole is found. Memory is also released whenever a process terminates, either by exiting or by being killed by a signal.

Figure 4-32 shows both ways of allocating memory. In Fig. 4-32(a) we see two processes, A and B, in memory. If A forks, we get the situation of Fig. 4-32(b). The

child is an exact copy of A. If the child now execs the file C, the memory looks like Fig. 4-32(c). The child's image is replaced by C.

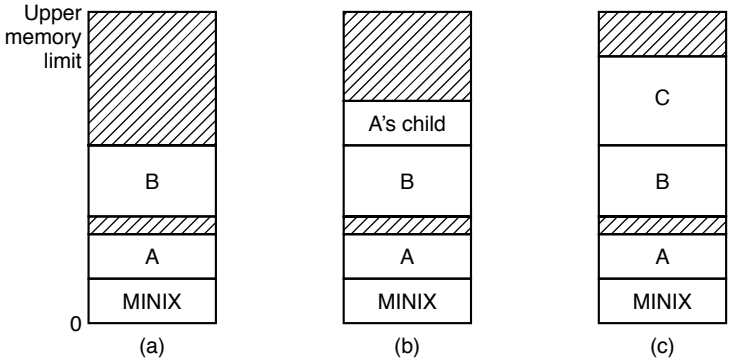


Figure 4-32. Memory allocation. (a) Originally. (b) After a FORK. (c) After the child does an EXEC. The shaded regions are unused memory. The process is a common I&D one.

Note that the old memory for the child is released before the new memory for C is allocated, so that C can use the child's memory. In this way, a series of fork and exec pairs (such as the shell setting up a pipeline) can result in all the processes being adjacent, with no holes between them, as would have been the case had been the case if the new memory had been allocated before the old memory had been released.

When memory is allocated, either by the FORK or EXEC system calls, a certain amount of it is taken for the new process. In the former case, the amount taken is identical to what the parent process has. In the latter case, the memory manager takes the amount specified in the header of the file executed. Once this allocation has been made, under no conditions is the process ever allocated any more total memory.

What has been said so far applies to programs that have been compiled with combined I and D space. Programs with separate I and D space take advantage of an enhanced mode of memory management called **shared text**. When such a process does a FORK, only the amount of memory needed for a copy of the new process' data and stack is allocated. Both the parent and the child share the executable code already in use by the parent. When such a process does an EXEC, the process table is searched to see if another process is already using the executable code needed. If one is found, new memory is allocated only for the data and stack, and the text already in memory is shared. Shared text complicates termination of a process. When a process terminates it always releases the memory occupied by its data and stack. But it only releases the memory occupied by its text segment after a search of the process table reveals that no other current process is sharing that memory. Thus a process may be allocated more memory when it starts than it releases when it terminates, if it loaded its own text when it started but that text is being shared by one or more other processes when the first process terminates.

Figure 4-33 shows how a program is stored as a disk file and how this is transferred to the internal memory layout of a MINIX process. The header on the disk file contains information about the sizes of the different parts of the image, as well as the total size. In the header of a program with common I and D space, a field specifies the total size of the text and data parts; these parts are copied directly to the memory image. The data part in the image is enlarged by the amount specified in the *bss* field in the header. This area is cleared to contain all zeroes and is used for uninitialized static data. The total amount of memory to be allocated is specified by the *total* field in the header. If, for example,

a program has 4K of text, 2K of data plus bss, and 1K of stack, and the header says to allocate 40K total, the gap of unused memory between the data segment and the stack segment will be 33K. A program file on the disk may also contain a symbol table. This is for use in debugging and is not copied into memory.

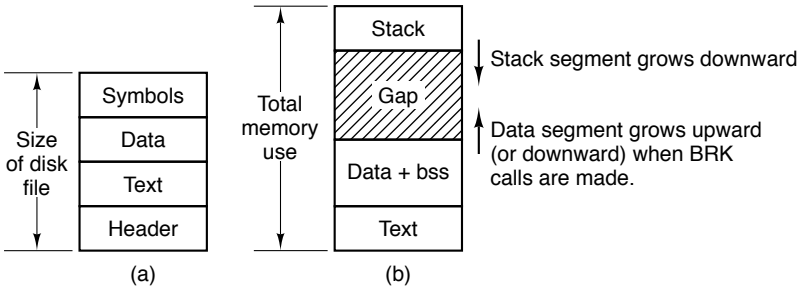


Figure 4-33. (a) A program as stored in a disk file. (b) Internal memory layout for a single process. In both parts of the figure the lowest disk or memory address is at the bottom and the highest address is at the top.

If the programmer knows that the total memory needed for the combined growth of the data and stack segments for the file *a.out* is at most 10K, he can give the command

```
chmem =10240 a.out
```

which changes the header field so that upon exec the memory manger allocates a space 10240 bytes more than the sum of the initial text and data segments. For the above example, a total of 16K will be allocated on all subsequent EXECs of the file. Of this amount, the topmost 1K will be used for the stack, and 9K will be in the gap, where it can be used by growth of the stack, the data area, or both.

For a program using separate I and D space (indicated by a bit in the header that is set by the linker), the total field in the header applies to the combined data and stack space only. A program with 4K of text, 2K of data, 1K of stack, and a total size of 64K will be allocated 68K (4K instruction space, 64K stack and data space), leaving 61K for the data segment and stack to consume during execution. The boundary of the data segment can be moved only by the brk system call. All brk does is check to see if the new data segment bumps into the current stack pointer, and if not, notes the change in some internal tables. This is entirely internal to the memory originally allocated to the process; no additional memory is allocated by the operating system. If the new data segment bumps into the stack, the call fails.

This strategy was chosen to make it possible to run MINIX on an IBM PC with an 8088 processor, which does not check for stack overflow in hardware. A user program can push as many words as it wants onto the stack without the operating system being aware of it. On computers with more sophisticated memory management hardware, the stack is allocated a certain amount of memory initially. If it attempts to grow beyond this amount, a trap to the operating system occurs, and the system allocates another piece of memory to the stack, if possible. This trap does not exist on the 8088, making it dangerous to have the stack adjacent to anything except a large chunk of unused memory, since the stack and grow quickly and without warning. MINIX has been designed so that when it is implemented on a computer with better memory management, it is straightforward to change the MINIX memory manager.

This is a good place to mention a possible semantic difficulty. When we use the word "segment," we refer to an area of memory defined by the operating system. The Intel

80x86 processors have a set of internal "segment registers" and (in the more advanced processors) "segment descriptor tables" which provide hardware support for "segments." The Intel hardware designers' concept of a segment is similar to, but not always the same as, the segments used and defined by MINIX. All references to segments in this text should be interpreted as references to memory areas delineated by MINIX data structures. We will refer explicitly to segment registers or segment descriptors when talking about the hardware.

This warning can be generalized. Hardware designers often try to provide support for the operating systems that they expect to be used on their machines, and the terminology used to describe registers and other aspects of a processor's architecture usually reflects an idea of how the features will be used. Such features are often useful to the implementer of an operating system, but they may not be used in the same way the hardware designer foresaw. This can lead to misunderstandings when the same word has different meanings when used to describe an aspect of an operating system or of the underlying hardware.

4.7.2 Message Handling

Like all the other components of MINIX, the memory manager is message driven. After the system has been initialized, the memory manager enters its main loop, which consists of waiting for a message, carrying out the request contained in the message, and sending a reply. Figure 4-34 gives the list of legal message types, input parameters, and values sent back in the reply message.

FORK, EXIT, WAIT, WAITPID, BRK, and EXEC are clearly closely related to memory allocation and deallocation. The calls KILL, ALARM, and PAUSE are all related to signals, as are SIGACTION, SIGSUSPEND, SIGPENDING, SIGMASK, and SIGRETURN. These also can affect what is in memory, because when a signal kills a process the memory used by that process is deallocated. REBOOT has effects throughout the operating system, but its first job is to send signals to terminate all processes in a controlled way, so the memory manager is a good place for it. The seven GET/SET calls have nothing to do with memory management at all. They also have nothing to do with the file system. But they had to go either in the file system or the memory manager, since every system call is handled by one or the other. They were put here simply because the file system was large enough already. PTRACE, which is used in debugging, is here for the same reason.

The final message, KSIG, is not a system call. KSIG is the message type used by the kernel to inform the memory manager of a signal originating in the kernel, such as SIGINT, SIGQUIT, or SIGALARM.

although there is a library routine *sbrk*, there is no system call SBRK. The library routine computes the amount of memory needed by adding the increment or decrement specified as parameter to the current size and makes a BRK call to set the size. Similarly, there are no separate system calls for *geteuid* and *getegid*. The calls GETUID and GETGID return both the effective and real identifiers. In like manner, GETPID returns the pid of both the calling process and its parent.

A key data structure used for message processing is the table *call_vec* declared in *table.c* (line 16515). It contains pointers to the procedures that handle the various message types. When a message comes in to the memory manager, the main loop extracts the message type and puts it in the global variable *mm_call*. This value is then used to index into *call_vec* to find the pointer to the procedure that handles the newly arrived message. That procedure is then called to execute the system call. The value that it returns is sent

back to the caller in the reply message to report on the success or failure of the call. The mechanism is similar to that of Fig. 1-16, only in user space rather than in the kernel.

Message type	Input parameters	Reply value
FORK	(none)	Child's pid, (to child: 0)
EXIT	Exit status	(No reply if successful)
WAIT	(none)	Status
WAITPID	(none)	Status
BRK	New size	New size
EXEC	Pointer to initial stack	(No reply if successful)
KILL	Process identifier and signal	Status
ALARM	Number of seconds to wait	Residual time
PAUSE	(none)	(No reply if successful)
SIGACTION	Sig. number, action, old action	Status
SIGSUSPEND	Signal mask	(No reply if successful)
SIGPENDING	(none)	Status
SIGMASK	How, set, old set	Status
SIGRETURN	Context	Status
GETUID	(none)	Uid, effective uid
GETGID	(none)	Gid, effective gid
GETPID	(none)	Pid, parent pid
SETUID	New uid	Status
SETGID	New gid	Status
SETSID	New sid	Process group
SETPGRP	New gid	Process group
PTRACE	Request, pid, address, data	Status
REBOOT	How (halt, reboot, or panic)	(No reply if successful)
KSIG	Process slot and signals	(No reply)

Figure 4-34. The message types, input parameters, and reply values used for communicating with the memory manager.

4.7.3 Memory Manager Data Structures and Algorithms

The memory manager has two key data structures: the process table and the hole table. We will now look at each of these in turn.

In Fig. 2-4 we saw that some process table fields are needed for process management, others for memory management, and yet others for the file system. In MINIX, each of these three pieces of the operating system has its own process table, containing just those fields that it needs. The entries correspond exactly, to keep things simple. Thus, slot *k* of the memory manager's table refers to the same process as slot *k* of the file system's table. When a process is created or destroyed, all three parts update their tables to reflect the new situation, in order to keep them synchronized.

The memory managers process table is called *mproc*; its definition is in */usr/src/mm/mproc.h*. It contains all the fields related to a process' memory allocation, as well as some additional items. The most important field is the array *mp_seg*, which has three entries, for the text, data, and stack segments, respectively. Each entry is a structure containing the virtual address, physical address, and length of the segment, all measured in clicks rather than in bytes. The size of a click is implementation dependent; for standard MINIX it is 256 bytes. All segments must start on a click boundary and occupy an integral number of clicks.

The method used for recording memory allocation is shown in Fig. 4-35. In this figure we have a process with 3K of text, 4K of data, a gap of 1K, and then a 2K stack, for a total memory allocation of 1OK. In Fig. 4-35(b) we see what the virtual, physical, and length fields for each of the three segments are, assuming that the process does not have separate I and D space. In this model, the text segment is always empty, and the data segment

contains both text and data. When a process references virtual address 0, either to jump to it or to read it (i.e., as instruction space or as data space), physical address Ox32000 (in decimal, 200K) will be used. This address is at click 0x320.

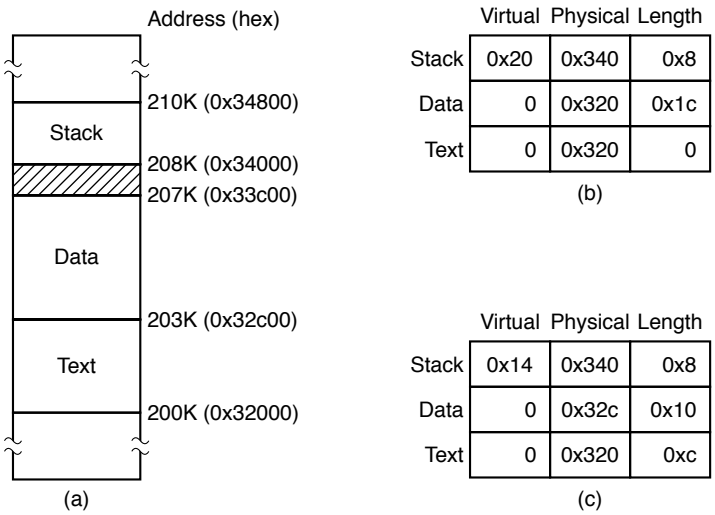


Figure 4-35. (a) A process in memory. (b) Its memory representation for non-separate I and D space. (c) Its memory representation for separate I and D space.

Note that the virtual address at which the stack begins depends initially on the total amount of memory allocated to the process. If the *chmem* command were used to modify the file header to provide a larger dynamic allocation area (bigger gap between data and stack segments), the next time the file was executed, the stack would start at a higher virtual address. If the stack grows longer by one click, the stack entry *should* change from the triple (Ox20, 0x340, 0x8) to the triple (Ox1F, 0x33F, 0x9).

The 8088 hardware does not have a stack limit trap, and MINIX defines the stack in a way that will not trigger the trap on 32-bit processors until the stack has already overwritten the data segment. Thus, this change will not be made until the next BRK system call, at which point the operating system explicitly reads SP and recomputes the segment entries. On a machine with a stack trap, the stack segments entry could be updated as soon as the stack outgrew its segment. This is not done by MINIX on 32-bit Intel processors, for reasons we will now discuss.

We mentioned previously that the efforts of hardware designers may not always produce exactly what the software designer needs. Even in protected mode on a Pentium, MINIX does not trap when the stack outgrows its segment. Although in protected mode the Intel hardware detects attempted access to memory outside a segment (as defined by a segment descriptor such as the one in Fig. 4-28), in MINIX the data segment descriptor and the stack segment descriptor are always identical. The MINIX-defined data and stack each use part of this space, and thus either or both can expand into the gap between them. However, only MINIX can manage this. The CPU has no way to detect errors involving the gap, since as far as the hardware is concerned the gap is a valid part of both the data area and the stack area. Of course, the hardware can detect a very large error, such as an attempt to access memory outside the combined data-gap-stack area. This will protect one process from another process' mistakes but is not enough to protect a process from itself.

A design decision was made here. We recognize an argument can be made for aban-

doning the shared hardware-defined segment that allows MINIX to dynamically reallocate the gap area. The alternative, using the hardware to define non-overlapping stack and data segments, would offer somewhat more security from certain errors but would make MINIX more memory-hungry. The source code is available to anybody who wants to evaluate the other approach.

Fig. 4-35(c) shows the segment entries for the memory layout of Fig. 4-35(a) for separate I and D space. Here both the text and data segments are nonzero in length. The *mp_seg* array shown in Fig. 4-35(b) or (c) is primarily used to map virtual addresses onto physical memory addresses. Given a virtual address and the space to which it belongs, it is a simple matter to see whether the virtual address is legal or not (i.e., falls inside a segment), and if legal, what the corresponding physical address is. The kernel procedure *umap* performs this mapping for the I/O tasks and for copying to and from user space, for example.

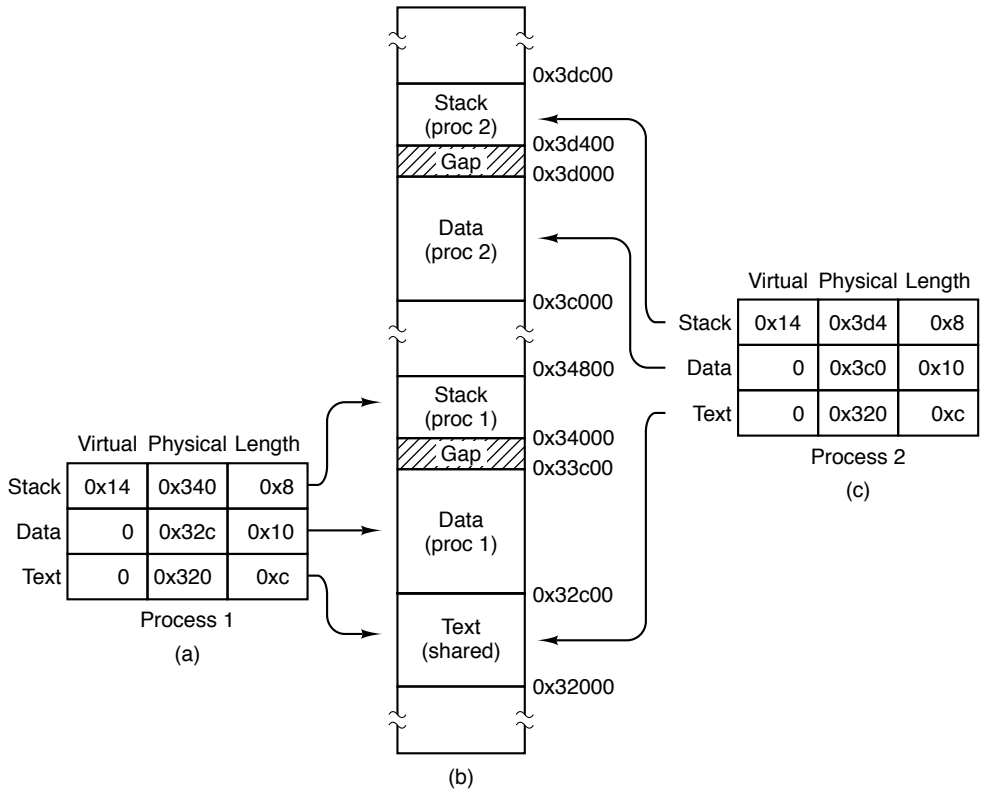


Figure 4-36. (a) The memory map of a separate I and D space process. as in the previous figure. (b) The layout in memory after a second process starts, executing the same program image with shared text. (c) The memory map of the second process.

The contents of the data and stack areas belonging to a process may change as the process executes, but the text does not change. It is common for several processes to be executing copies of the same program. for instance several users may be executing the same shell. Memory efficiency is improved by using shared text. When EXEC is about to load a process, it opens the file holding the disk image of the program to be loaded and reads the file header. If the process uses separate I and D space, a search of the *mp_dev*, *mp_ino*, and *mp_ctime* fields in each slot of *mproc* is made. These hold the device and i-node numbers and changed-status times of the images being executed by other processes. If a process already loaded is found to be executing the same program that is about to

be loaded, there is no need to allocate memory for another copy of the text. Instead the *mp_seg[T]* portion of the new process' memory map is initialized to point to the same place where the text segment is already loaded, and only the data and stack portions are set up in a new memory allocation. This is shown in Fig. 4-36. If the program uses combined I and D space or no match is found, memory is allocated as shown in Fig. 4-35 and the text and data for the new process are copied in from the disk.

In addition to the segment information, *mproc* also holds the process ID (pid) of the process itself and of its parent, the uids and gids (both real and effective). information about signals, and the exit status, if the process has already terminated but its parent has not yet done a WAIT for it.

The other major memory manager table is the hole table, *hole*, defined in *alloc.c*, which lists every hole in memory in order of increasing memory address. The gaps between the data and stack segments are not considered holes; they have already been allocated to processes. Consequently, they are not contained in the free hole list. Each hole list entry has three fields: the base address of the hole, in clicks; the length of the hole, in clicks; and a pointer to the next entry on the list. The list is singly linked, so it is easy to find the next hole starting from any given hole, but to find the previous hole, you have to search the entire list from the beginning until you come to the given hole.

The reason for recording everything about segments and holes in clicks rather than bytes is simple: it is much more efficient. In 16-bit mode, 16-bit integers are used for recording memory addresses, so with 256-bit clicks, up to l6 MB of memory can be supported. In 32-bit mode, address fields can refer to up to 240 bytes, which is 1024 gigabytes.

The principal operations on the hole list are allocating a piece of memory of a given size and returning an existing allocation. To allocate memory, the hole list is searched, starting at the hole with the lowest address, until a hole that is large enough is found (first fit). The segment is then allocated by reducing the hole by the amount needed for the segment, or in the rare case of an exact fit, removing the hole from the list. This scheme is fast and simple but suffers from both a small amount of internal fragmentation (up to 255 bytes may be wasted in the final click, since an integral number of clicks is always taken) and external fragmentation.

When a process terminates and is cleaned up, its data and stack memory are returned to the free list. If it uses common I and D, this releases all its memory, since such programs never have a separate allocation of memory for text. If the program uses separate I and D and a search of the process table reveals no other process is sharing the text, the text allocation will also be returned. Since with shared text the text and data regions are not necessarily contiguous, two regions of memory may be returned. For each region returned, if either or both of the region's neighbors are holes, they are merged, so adjacent holes never occur. In this way, the number, location, and sizes of the holes vary continuously during system operation. Whenever all user processes have terminated, all of available memory is once again ready for allocation. This isn't necessarily a single hole, however, since physical memory may be interrupted by regions unusable by the operating system, as in IBM compatible systems where read-only memory (ROM) and memory reserved for I/O transfers separate usable memory below address 640K from memory above 1M.

4.7.4 The FORK, EXIT, and WAIT System Calls

When processes are created or destroyed, memory must be allocated or deallocated. Also, the process table must be updated, including the parts held by the kernel and FS. The memory manager coordinates all this activity. Process creation is done by FORK, and carried out in the series of steps shown in Fig. 4-37.

1. Check to see if process table is full.
2. Try to allocate memory for the child's data and stack.
3. Copy the parent's data and stack to the child's memory.
4. Find a free process slot and copy parent's slot to it.
5. Enter child's memory map in process table.
6. Choose a pid for the child.
7. Tell kernel and file system about child.
8. Report child's memory map to kernel.
9. Send reply messages to parent and child.

Figure 4-37. The steps required to carry out the FORK system call.

It is difficult and inconvenient to stop a FORK call part way through, so the memory manager maintains a count at all times of the number of processes currently in existence in order to see easily if a process table slot is available. If the table is not full, an attempt is made to allocate memory for the child. If the program is one with separate I and D space, only enough memory for new data and stack allocations is requested. If this step also succeeds, the FORK is guaranteed to work. The newly allocated memory is then filled in, a process slot is located and filled in, a pid is chosen, and the other parts of the system are informed that a new process has been created.

A process fully terminates when two events have both happened: (1) the process itself has exited (or has been killed by a signal), and (2) its parent has executed a WAIT system call to find out what happened. A process that has exited or has been killed, but whose parent has not (yet) done a WAIT for it, enters a kind of suspended animation, sometimes known as **zombie state**. It is prevented from being scheduled and has its alarm timer turned off (if it was on), but it is not removed from the process table. Its memory is freed. Zombie state is temporary and rarely lasts long. When the parent finally does the WAIT, the process table slot is freed, and the file system and kernel are informed.

A problem arises if the parent of an exiting process is itself already dead. If no special action were taken, the exiting process would remain a zombie forever. Instead, the tables are changed to make it a child of the *init* process. When the system comes up, *init* reads the */etc/ttytab* tile to get a list of all terminals, and then forks off a login process to handle each one. It then blocks, waiting for processes to terminate. In this way, orphan zombies are cleaned up quickly.

4.7.5 The EXEC System Call

When a command is typed at the terminal, the shell forks off a new process, which then executes the command requested. It would have been possible to have a single system call to do both FORK and EXEC at once, but they were provided as two distinct calls for a very good reason: to make it easy to implement I/O redirection. When the shell forks, if standard input is redirected, the child closes standard input and then opens the

new standard input before executing the command. In this way the newly started process inherits the redirected standard input. Standard output is handled the same way.

EXEC is the most complex system call in MINIX. It must replace the current memory image with a new one, including setting up a new stack. It carries out its job in a series of steps, as shown in Fig. 4-38.

1. Check permissions—is the file executable?
2. Read the header to get the segment and total sizes.
3. Fetch the arguments and environment from the caller.
4. Allocate new memory and release unneeded old memory.
5. Copy stack to new memory image.
6. Copy data (and possibly text) segment to new memory image.
7. Check for and handle setuid, setgid bits.
8. Fix up process table entry.
9. Tell kernel that process is now runnable.

Figure 4-38. The steps required to carry out the EXEC system call.

Each step consists, in turn, of yet smaller steps, some of which can fail. For example, there might be insufficient memory available. The order in which the tests are made has been carefully chosen to make sure the old memory image is not released until it is certain that the EXEC will succeed, to avoid the embarrassing situation of not being able to set up a new memory image, but not having the old one to go back to, either. Normally EXEC does not return, but if it fails, the calling process must get control again, with an error indication.

There are a few steps in Fig. 4-38 that deserve some more comment. First is the question of whether or not there is enough room or not. After determining how much memory is needed, which requires determining if the text memory of another process can be shared, the hole list is searched to check whether there is sufficient physical memory *before* freeing the old memory—if the old memory were freed first and there were insufficient memory, it would be hard to get the old image back again.

However, this test is overly strict. It sometimes rejects EXEC calls that, in fact, could succeed. Suppose, for example, the process doing the EXEC call occupies 20K and its text is not shared by any other process. Further suppose that there is a 30K hole available and that the new image requires 50K. By testing before releasing, we will discover that only 30K is available and reject the call. If we had released first, we might have succeeded, depending on whether or not the new 20K hole were adjacent to, and thus now merged with, the 30K hole. A more sophisticated implementation could handle this situation a little better.

Another possible improvement would be to search for two holes, one for the text segment and one for the data segment, if the process to be EXECed uses separate I and D space. There is no need for the segments to be contiguous.

A more subtle issue is whether the executable file fits in the *virtual* address space. The problem is that memory is allocated not in bytes, but in 256-byte clicks. Each click must belong to a single segment, and may not be, for example, half data, half stack, because the entire memory administration is in clicks.

To see how this restriction can give trouble, note that the address space on 16-bit systems (8088 and 80286) is limited to 64K, which can be divided into 256 clicks. Suppose that a separate I and D space program has 40,000 bytes of text, 32,770 bytes of data, and 32,760 bytes of stack. The data segment occupies 129 clicks, of which the last one is

only partially used; still, the whole click is part of the data segment. The stack segment is 128 clicks. Together they exceed 256 clicks, and thus cannot co-exist, even though the number of *bytes* needed fits in the virtual address space (barely). In theory this problem exists on all machines whose click size is larger than 1 byte, but in practice it rarely occurs on Pentium-class processors, since they permit large (4-GB) segments.

Another important issue is how the initial stack is set up. The library call normally used to invoke EXEC with arguments and an environment is

```
execve(name, argv, envp);
```

where *name* is a pointer to the name of the file to be executed, *argv* is a pointer to an array of pointers, each one pointing to an argument, and *envp* is a pointer to an array of pointers, each one pointing to an environment string.

It would be easy enough to implement EXEC by just putting the three pointers in the message to the memory manager and letting it fetch the file name and two arrays by itself. Then it would have to fetch each argument and each string one at a time. Doing it this way requires at least one message to the system task per argument or string and probably more, since the memory manager has no way of knowing how big each one is in advance.

To avoid the overhead of multiple messages to read all these pieces, a completely different strategy has been chosen. The *execve* library procedure builds the entire initial stack inside itself and passes its base address and size to the memory manager. Building the new stack within the user space is highly efficient, because references to the arguments and strings are just local memory references, not references to a different address space.

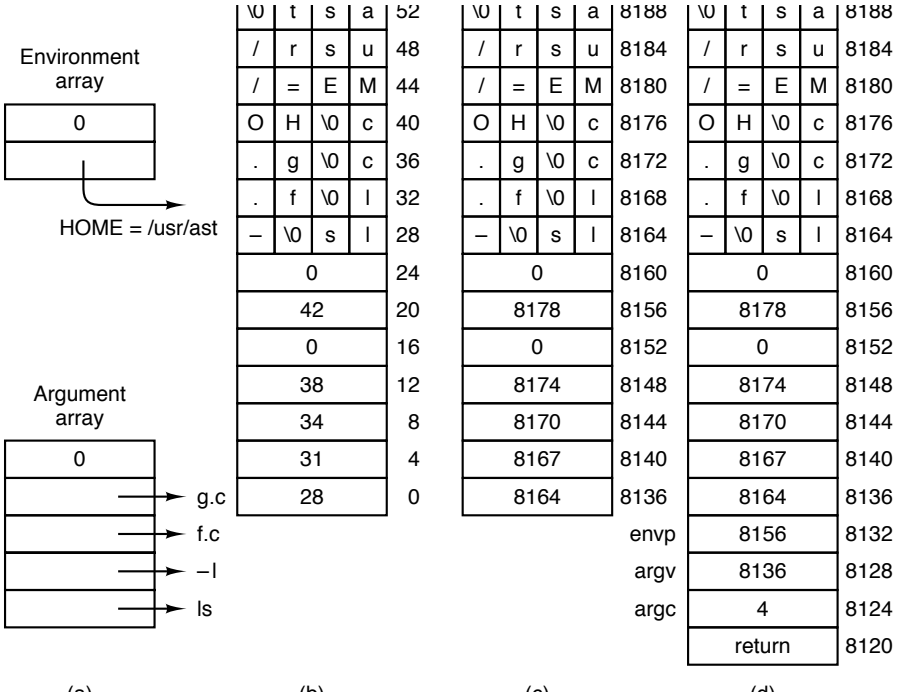


Figure 4-39. (a) The arrays passed to *execve*. (b) The stack built by *execve*. (c) The stack after relocation by the memory manager. (d) The stack as it appears to main at the start of execution.

To make this mechanism clearer, consider an example. When a user types

```
ls -l f.c g.c
```

to the shell, the shell interprets it and then makes the call

```
execve("/bin/ls", argv, envp);
```

to the library procedure. The contents of the two pointer arrays are shown in Fig. 4-39(a). The procedure *execve*, within the shell's address space, now builds the initial stack, as shown in Fig. 4-39(b). This stack is eventually copied intact to the memory manager during the processing of the EXEC call.

When the stack is finally copied to the user process, it will not be put at virtual address 0. Instead, it will be put at the end of the memory allocation. as determined by the total memory size field in the executable file's header. As an example, let us arbitrarily assume that the total size is 8192 bytes, so the last byte available to the program is at address 8191. It is up to the memory manager to relocate the pointers within the stack so that when deposited into the new address, the stack looks like Fig. 4-39(c).

When the EXEC call completes and the program starts running, the stack will indeed look exactly like Fig. 4-39(c), with the stack pointer having the value 8136. However, another problem is yet to be dealt with. The main program of the executed tile is probably declared something like this:

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

As far as the C compiler is concerned, *main* is just another function. It does not know that *main* is special, so it compiles code to access the three parameters on the assumption that they will be passed according to the standard C calling convention, last parameter first. With one integer and two pointers, the three parameters are expected to occupy the three words just before the return address. Of course, the stack of Fig. 4-39(c) does not look like that at all.

The solution is that programs do not begin with *main*. Instead. a small, assembly language routine called the C run-time, start-off procedure, *crtso*, is always linked in at text address 0 so it gets control first. Its job is to push three more words onto the stack and then to call *main* using the standard call instruction. This results in the stack of Fig. 4-39(d) at the time that *main* starts executing. Thus, *main* is tricked into thinking it was called in the usual way (actually, it is not really a trick; it is called that way).

If the programmer neglects to call *exit* at the end of *main*, control will pass back to the C run-time, start-off routine when *main* is finished. Again, the compiler just sees *main* as an ordinary procedure and generates the usual code to return from it after the last statement. Thus *main* returns to its caller, the C run-time, start-off routine which then calls *exit* itself. Most of the code of 32-bit *crtso* is shown in Fig. 4-40. The comments should make its operation clear. All that has been left out is the code that loads the registers that are pushed and a few lines that set a flag that indicates if a floating point coprocessor is present or not.

4.7.6 The BRK System Call

The library procedures *brk* and *sbrk* are used to adjust the upper bound of the data segment. The former takes an absolute size (in bytes) and calls BRK. The latter takes a positive or negative increment to the current size, computes the new data segment size, and then calls BRK. There is no actual SBRK system call.

An interesting question is: "How does *sbrk* keep track of the current size, so it can compute the new size?" The answer is that a variable, *brksize*, always holds the current

```
push ecx      ! push environ
push edx      ! push argv
push eax      ! push argc
call _main    ! main(argc, argv, envp)
push eax      ! push exit status
call _exit
hlt           ! force a trap if exit fails
```

Figure 4-40. The key part of the C run-time. start-off routine.

size so *sbrk* can find it. This variable is initialized to a compiler generated symbol giving the initial size of text plus data (non-separate l and D) or just data (separate I and D). The name, and, in fact, very existence of such a symbol is compiler dependent, and thus it will not be found defined in any header file in the source file directories. It is defined in the library, in the file *brksize.s*. Exactly where it will be found depends on the system, but it will be in the same directory as *crtso.s*.

Carrying out BRK is easy for the memory manager. All that must be done is to check to see that everything still fits in the address space, adjust the tables, and tell the kernel.

4.7.7 Signal Handling

4.7.8 Other System Calls

4.8 Implementation of Memory Management in MINIX

4.8.1 The Header Files and Data Structures

4.8.2 The Main Program

4.8.3 Implementation of FORK, EXIT, and WAIT

4.8.4 Implementation of EXEC

4.8.5 Implementation of BRK

4.8.6 Implementation of Signal Handling

4.8.7 Implementation of the Other System Calls

4.8.8 Memory Manager Utilities

4.9 Summary

Chapter 5

FILE SYSTEMS

All computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information within its own address space. However the storage capacity is restricted to the size of the virtual address space. For some applications this size is adequate, but for others, such as airline reservations, banking, or corporate record keeping, it is far too small.

A second problem with keeping information within a process' address space is that when the process terminates, the information is lost. For many applications, (e.g., for databases), the information must be retained for weeks, months, or even forever. Having it vanish when the process using it terminates is unacceptable. Furthermore, it must not go away when a computer crash kills the process.

A third problem is that it is frequently necessary for multiple processes to access (parts of) the information at the same time. If we have an on-line telephone directory stored inside the address space of a single process, only that process can access it. The way to solve this problem is to make the information itself independent of any one process.

Thus we have three essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. The information must survive the termination of the process using it.
3. Multiple processes must be able to access the information concurrently.

The usual solution to all these problems is to store information on disks and other external media in units called **files**. Processes can then read them and write new ones if need be. Information stored in files must be **persistent**, that is, not be affected by process creation and termination. A file should only disappear when its owner explicitly removes it.

Files are managed by the operating system. How they are structured, named, accessed, used, protected, and implemented are major topics in operating system design. As a whole, that part of the operating system dealing with files is known as the **file system** and is the subject of this chapter.

From the user's standpoint, the most important aspect of a file system is how it appears to them, that is, what constitutes a file, how files are named and protected, what operations are allowed on files, and so on. The details of whether linked lists or bitmaps are used to keep track of free storage and how many sectors there are in a logical block are of less interest, although they are of great importance to the designers of the file system. For this reason, we have structured the chapter as several sections. The first two are concerned with the user interface to files and directories, respectively. Then comes a detailed

discussion of how the file system is implemented. Finally, we give some examples of real file systems.

5.1 FILES √

In the following pages we will look at files from the user's point of view, that is, how they are used and what properties they have.

5.1.1 File Naming

Files are an abstraction mechanism. They provide a way to store information on the disk and read it back later. This must be done in such a way as to shield the user from the details of how and where the information is stored, and how the disks actually work.

Probably the most important characteristic of any abstraction mechanism is the way the objects being managed are named, so we will start our examination of file systems with the subject of file naming.

When a process creates a file, it gives the file a name. When the process terminates, the file continues to exist and can be accessed by other processes using its name.

The exact rules for file naming vary somewhat from system to system, but all current operating systems allow strings of one to eight letters as legal file names. Thus *andrea*, *bruce*, and *cathy* are possible file names. Frequently digits and special characters are also permitted, so names like *2*, *urgent!*, and *Fig.2-14* are often valid as well. Many file systems support names as long as 255 characters.

Some file systems distinguish between upper and lower case letters, whereas others do not. UNIX falls in the first category; MS-DOS falls in the second. Thus a UNIX system can have all of the following as three distinct files: *Barbara*, *BARBARA*, *BARbara* and *BarBaRa*. In MS-DOS they all designate the same file.

Many operating systems support two-part file names, with the two parts separated by a period, as in *prog.c*. The part following the period is called the **file extension** and usually indicates something about the file. In MS-DOS, for example, file names are 1 to 8 characters, plus an optional extension of 1 to 3 characters. In UNIX, the size of the extension, if any, is up to the user, and a file may even have two or more extensions, as in *prog.c.Z*, where *.Z* is commonly used to indicate that the file (*prog.c*) has been compressed using the Ziv-Lempel compression algorithm. Some of the more common file extensions and their meanings are shown in Fig. 5-1.

In some cases, file extensions are just conventions and are not enforced in any way. A file named *file.txt* might be some kind of text file, but that name is more to remind the owner than to convey any specific information to the computer. On the other hand, a C compiler may actually insist that files it is to compile end in *.c*, and it may refuse to compile them if they do not.

Conventions like this are especially useful when the same program can handle several different kinds of files. The C compiler, for example, can be given a list of files to compile and link together, some of them C files and some of them assembly language files. The extension then becomes essential for the compiler to tell which are C files, which are assembly files, and which are other files.

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.f77	Fortran 77 program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compile output, not yet linked)
file.ps	Postscript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Figure 5-1. Some typical file extensions.

5.1.2 File Structure

Files can be structured in any of several ways. Three common possibilities are depicted in Fig. 5-2. The file in Fig. 5-2(a) is just an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and MS-DOS use this approach. As an aside, WINDOWS 95 basically uses the MS-DOS file system, with a little syntactic sugar added (f.g., long file names), so nearly everything said in this chapter about ms-dos also holds for WINDOWS 95, WINDOWS NT is completely different, however.

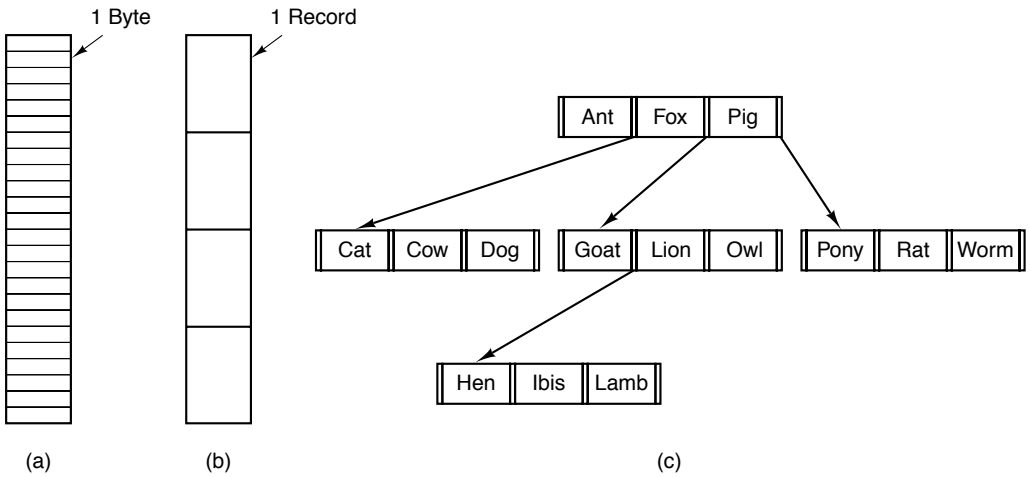


Figure 5-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Having the operating system regard files as nothing more than byte sequences provides the maximum flexibility. User programs can put anything they want in their files and name them any way that is convenient. The operating system does not help, but it also does not get in the way. For users who want to do unusual things, the latter can be very important.

The first step up in structure is shown in Fig. 5-2(b). In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record. In years gone by, when the 80-column punched card was king many operating systems based their file systems on files consisting of 80-character records, in effect, card images. These systems also supported files of 132-character records, which were intended for the line printer (which in those days were big chain printers having 132 columns). Programs read input in units of 80 characters and wrote it in units of 132 characters, although the final 52 could be spaces, of course.

An (old) system that viewed files as sequences of fixed-length records was CP/M. It used a 128-character record. Nowadays, the idea of a file as a sequence of fixed-length records is pretty much gone, although it was once the norm.

The third kind of file structure is shown in Fig. 5-2(c). In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a **key** field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

The basic operation here is not to get the "next" record, although that is also possible, but to get the record with a specific key. For the zoo file of Fig. 5-2(c), one could ask the system to get the record whose key is *pony*, for example, without worrying about its exact position in the file. Furthermore, new records can be added to the file, with the operating system, and not the user, deciding where to place them. This type of file is clearly quite different from the unstructured byte streams used in UNIX and MS-DOS but is widely used on the large mainframe computers still used in some commercial data processing.

5.1.3 File Types

Many operating systems support several types of files. UNIX and MS-DOS, for example, have regular files and directories. UNIX also has character and block special files. **Regular files** are the ones that contain user information. All the files of Fig. 5-2 are regular files. **Directories** are system files for maintaining the structure of the file system. We will study directories below. **Character special files** are related to input/output and used to model serial I/O devices such as terminals, printers, and networks. **Block special files** are used to model disks. In this chapter we will be primarily interested in regular files.

Regular files are generally either ASCII files or binary files. ASCII files consist of lines of text. In some systems each line is terminated by a carriage return character. In others, the line feed character is used. Occasionally both are required. Lines need not all be of the same length.

The great advantage of ASCII files is that they can be displayed and printed as is, and they can be edited with any text editor. Furthermore, if large numbers of programs use ASCII files for input and output, it is easy to connect the output of one program to the input of another, as in shell pipelines. (The interprocess plumbing is not any easier, but interpreting the information certainly is if a standard convention, such as ASCII, is used for expressing it.)

Other files are binary files, which just means that they are not ASCII files. Listing them on the printer gives an incomprehensible listing full of what is apparently random junk. Usually, they have some internal structure.

For example, in Fig. 5-3(a) we see a simple executable binary file taken from an early version of UNIX. Although technically the file is just a sequence of bytes, the operating system will only execute a file if it has the proper format. It has five sections: header, text,

data, relocation bits, and symbol table. The header starts with a so-called **magic number**, identifying the file as an executable file (to prevent the accidental execution of a file not in this format). Then come 16-bit integers giving the sizes of the various pieces of the file, the address at which execution starts, and some flag bits. Following the header are the text and data of the program itself. These are loaded into memory and relocated using the relocation bits. The symbol table is used for debugging.

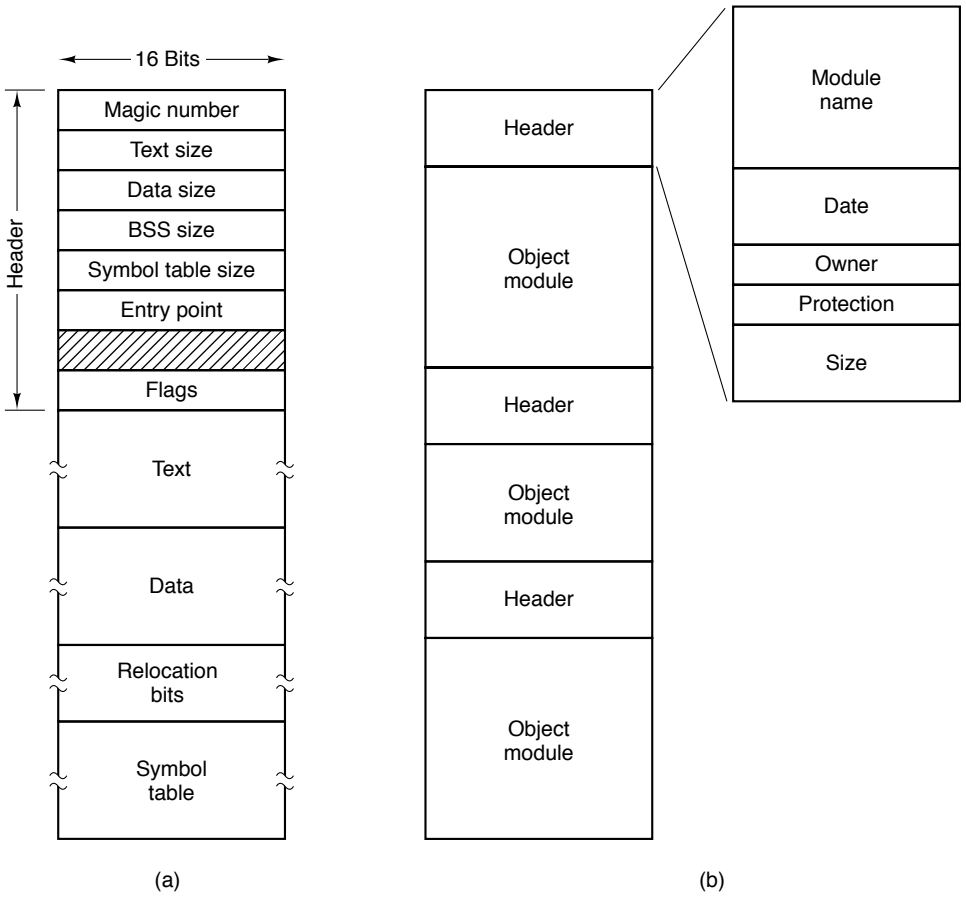


Figure 5-3. (a) An executable file. (b) An archive.

Our second example of a binary file is an archive, also from UNIX. It consists of a collection of library procedures (modules) compiled but not linked. Each one is prefaced by a header telling its name, creation date, owner, protection code, and size. Just as with the executable file, the module headers are full of binary numbers. Copying them to the printer would produce complete gibberish.

All operating systems must recognize at least one file type: its own executable file, but some operating systems recognize more. The old TOPS-20 system (for the DECsystem 20) went so far as to examine the creation time of any file to be executed. Then it located the source file and saw if the source had been modified since the binary was made. If it had been, it automatically recompiled the source. In UNIX terms, the *make* program had been built into the shell. The file extensions were mandatory so the operating system could tell which binary program was derived from which source.

In a similar vein, when a WINDOWS user double clicks on a file name, an appropriate program is launched with the file as parameter. The operating system determines which program to run based on the file extension.

Having strongly typed files like this causes problems whenever the user does anything

that the system designers did not expect. Consider, as an example, a system in which program output files have extension *.dat* (data files). If a user writes a program formatter that reads a *.pas* file, transforms it (e.g., by converting it to a standard indentation layout), and then writes the transformed file as output, the output file will be of type *.dat*. If the user tries to offer this to the Pascal compiler to compile it, the system will refuse because it has the wrong extension. Attempts to copy *file.dat* to *file.pas* will be rejected by the system as invalid (to protect the user against mistakes).

While this kind of "user friendliness" may help novices, it drives experienced users up the wall since they have to devote considerable effort to circumventing the operating system's idea of what is reasonable and what is not.

5.1.4 File Access

Early operating systems provided only a single kind of file access: **sequential access**. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order. Sequential files could be rewound, however, so they could be read as often as needed. Sequential files were convenient when the storage medium was magnetic tape, rather than disk.

When disks came into use for storing files, it became possible to read the bytes or records of a file out of order, or to access records by key, rather than by position. Files whose bytes or records can be read in any order are called **random access** files.

Random access files are essential for many applications, for example, database systems. If an airline customer calls up and wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight without having to read the records for thousands of other flights first.

Two methods are used for specifying where to start reading. In the first one, every READ operation gives the position in the file to start reading at. In the second one, a special operation, SEEK, is provided to set the current position. After a seek, the file can be read sequentially from the now-current position.

In some older mainframe operating systems, files are classified as being either sequential or random access at the time they are created. This allows the system to use different storage techniques for the two classes. Modern operating systems do not make this distinction. All their files are automatically random access.

5.1.5 File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was created and the file's size. We will call these extra items the file's attributes. The list of attributes varies considerably from system to system. The table of Fig. 5-4 shows some of the possibilities, but others also exist. No existing system has all of these, but each is present in some system.

The first four attributes relate to the file's protection and tell who may access it and who may not. All kinds of schemes are possible, some of which we will study later. In some systems the user must present a password to access a file, in which case the password must be one of the attributes.

The flags are bits or short fields that control or enable some specific property. Hidden files, for example, do not appear in listings of the files. The archive flag is a bit that keeps track of whether the file has been backed up. The backup program clears it, and

Field	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 5-4. Some possible file attributes.

the operating system sets it whenever a file is changed. In this way, the backup program can tell which files need backing up. The temporary flag allows a file to be marked for automatic deletion when the process that created it terminates.

The record length, key position, and key length fields are only present in files whose records can be looked up using a key. They provide the information required to find the keys.

The various times keep track of when the file was created, most recently accessed and most recently modified. These are useful for a variety of purposes. For example, a source file that has been modified after the creation of the corresponding object file needs to be recompiled. These fields provide the necessary information.

The current size tells how big the file is at present. Some old mainframe operating systems require the maximum size to be specified when the file is created, in order to let the operating system reserve the maximum amount of storage in advance. Modern operating systems are clever enough to do without this feature.

5.1.6 File Operations

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Below is a discussion of the most common system calls relating to files.

1. **CREATE**. The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.

2. **DELETE**. When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.

3. **OPEN**. Before using a file, a process must open it. The purpose of the OPEN call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.

4. **CLOSE**. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.

5. **READ**. Data are read from file. Usually, the bytes come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.

6. **WRITE**. Data are written to the file, again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.

7. **APPEND**. This call is a restricted form of WRITE. It can only add data to the end of the file. Systems that provide a minimal set of system calls do not generally have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have APPEND.

8. **SEEK**. For random access files, a method is needed to specify from where to take the data. One common approach is a system call, SEEK, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.

9. **GET ATTRIBUTES**. Processes often need to read file attributes to do their work. For example, the UNIX *make* program is commonly used to manage software development projects consisting of many source files. When make is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

10. **SET ATTRIBUTES**. Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example. Most of the flags also fall in this category.

11. **RENAME**. It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

5.2 DIRECTORIES $\surd$

To keep track of files, file systems normally have directories, which, in many systems, are themselves files. In this section we will discuss directories, their organization, their properties, and the operations that can be performed on them.

5.2.1 Hierarchical Directory Systems

A directory typically contains a number of entries, one per file. One possibility is shown in Fig. 5-5(a), in which each entry contains the file name, the file attributes, and the disk addresses where the data are stored. Another possibility is shown in Fig. 5-5(b). Here a directory entry holds the file name and a pointer to another data structure where the attributes and disk addresses are found. Both of these systems are commonly used.

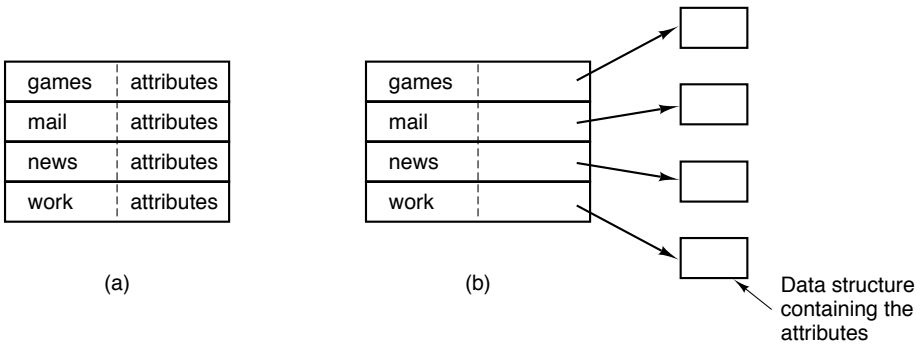


Figure 5-5. (a) Attributes in the directory entry. (b) Attributes elsewhere.

When a file is opened, the operating system searches its directory until it finds the name of the file to be opened. It then extracts the attributes and disk addresses, either directly from the directory entry or from the data structure pointed to, and puts them in a table in main memory. All subsequent references to the file use the information in main memory.

The number of directories varies from system to system. The simplest form of directory system is a single directory containing all files for all users, as illustrated in Fig. 5-6(a). If there are many users, and they choose the same file names (e.g., *mail* and *games*), conflicts and confusion will quickly make the system unworkable. This system model was used by the first microcomputer operating systems but is rarely seen any more.

An improvement on the idea of having a single directory for all the files in the entire system is to have one directory per user [see Fig. 5-6(b)]. This design eliminates name conflicts among users but is not satisfactory for users with a large number of files. It is quite common for users to want to group their files together in logical ways. A professor, for example, might have a collection of files that together form a book that he is writing for one course, a second collection of file containing student programs submitted for another course, a third group of files containing the code of an advanced compiler-writing system he is building, a fourth group of files containing grant proposals, as well as other files for electronic mail, minutes of meetings, papers he is writing, games, and so on. Some way is needed to group these files together in flexible ways chosen by the user.

What is needed is a general hierarchy (i.e., a tree of directories). With this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways. This approach is shown in Fig. 5-6(c). Here the directories *A*, *B* and *C* contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

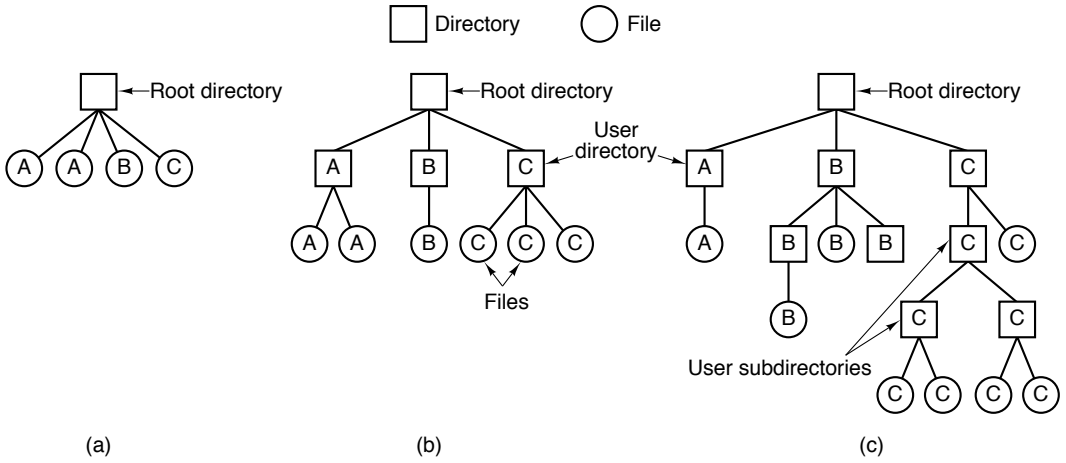


Figure 5-6. Three file system designs. (a) Single directory shared by all users. (b) One directory per user. (c) Arbitrary tree per user. The letters indicate the directory or file's owner.

5.2.2 Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file is given an **absolute path name** consisting of the path from the root directory to the file. As an example, the path */usr/ast/mailbox* means that the root directory contains a subdirectory *usr/*, which in turn contains a subdirectory *ast/*, which contains the file *mailbox*. Absolute path names always start at the root directory and are unique. In UNIX the components of the path are separated by /. In MS-DOS the separator is \ . In MULTICS it is ¿. No matter which character is used, if the first character of the path name is the separator, then the path is absolute.

The other kind of name is the **relative path name**. This is used in conjunction with the concept of the **working directory** (also called the **current directory**). A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For example, if the current working directory is */usr/ast*, then the file whose absolute path is */usr/ast/mailbox* can be referenced simply as *mailbox*. In other words, the UNIX command

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

and the command

```
cp mailbox mailbox.bak
```

do exactly the same thing if the working directory is */usr/ast/*. The relative form is often more convenient, but it does the same thing as the absolute form.

Some programs need to access a specific file without regard to what the working directory is. In that case, they should always use absolute path names. For example, a spelling checker might need to read */usr/lib/dictionary* to do its work. It should use the full, absolute path name in this case because it does not know what the working directory will be when it is called. The absolute path name will always work, no matter what the working directory is.

Of course, if the spelling checker needs a large number of files from */usr/lib/*, an alternative approach is for it to issue a system call to change its working directory to */usr/lib/*, and then use just dictionary as the first parameter to *open*. By explicitly changing the working directory, it knows for sure where it is in the directory tree, so it can then use relative paths.

In most systems, each process has its own working directory, so when a process changes its working directory and later exits, no other processes are affected and no traces of the change are left behind in the file system. In this way it is always perfectly safe for a process to change its working directory whenever that is convenient. On the other hand, if a library procedure changes the working directory and does not change back to where it was when it is finished, the rest of the program may not work since its assumption about where it is may now suddenly be invalid. For this reason, library procedures rarely change the working directory, and when they must, they always change it back again before returning.

Most operating systems that support a hierarchical directory system have two special entries in every directory, "." and "..", generally pronounced "dot" and "dotdot." Dot refers to the current directory; dotdot refers to its parent. To see how these are used, consider the UNIX file tree of Fig. 5-7. A certain process has */usr/ast/* as its working directory. It can use .. to go up the tree. For example, it can copy the file */usr/lib/dictionary* to its own directory using the command

```
cp ../lib/dictionary .
```

The first path instructs the system to go upward (to the *usr* directory), then to go down to the directory *lib* to find the file *dictionary*.

The second argument (dot) names the current directory. When the *cp* command gets a directory name (including dot) as its last argument, it copies all the files there. Of course, a more normal way to do the copy would be to type

```
cp /usr/lib/dictionary .
```

Here the use of dot saves the user the trouble of typing *dictionary* a second time.

5.2.3 Directory Operations

The system calls for managing directories exhibit more variation from system to system than system calls for files. To give an impression of what they are and how they work, we will give a sample (taken from UNIX).

1. **CREATE**. A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system (or in a few cases, by the *mkdir* program).

2. **DELETE**. A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot usually be deleted.

3. **OPENDIR**. Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.

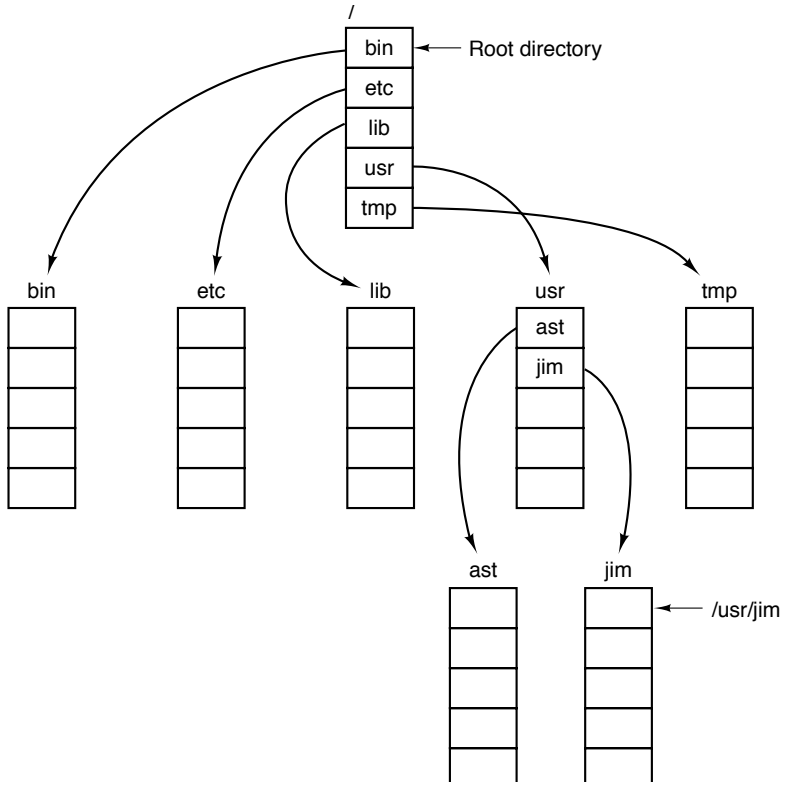


Figure 5-7. A UNIX directory tree.

4. **CLOSEDIR**. When a directory has been read, it should be closed to free up internal table space.

5. **READDIR**. This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual READ system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, READDIR always returns one entry in a standard format, no matter which of the possible directory structures is being used.

6. **RENAME**. In many respects, directories are just like files and can be renamed the same way files can be.

7. **LINK**. Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories.

8. **UNLINK**. A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, UNLINK.

The above list gives the most important calls, but there are a few others as well, for example, for managing the protection information associated with a directory.

5.3 FILE SYSTEM IMPLEMENTATION √

Now it is time to tum from the user's view of the file system to the implementor's view. Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like, and similar interface issues. Implementors are interested in how tiles and directories are stored, how disk space is managed, and how to make everything work efficiently and reliably. In the following sections we will examine a number of these areas to see what the issues and trade-offs are.

5.3.1 Implementing Files

Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems. In this section. we will examine a few of them.

Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous block of data on the disk. Thus on a disk with 1K blocks, a 50K tile would be allocated 50 consecutive blocks. This scheme has two significant advantages. First, it is simple to implement because keeping track of where a file's blocks are is reduced to remembering one number, the disk address of the first block. Second, the performance is excellent because the entire file can be read from the disk in a single operation.

Unfortunately, contiguous allocation also has two equally significant drawbacks. First, it is not feasible unless the maximum file size is known at the time the tile is created. Without this information, the operating system does not know how much disk space to reserve. In systems where files must be written in a single blow, it can be used to great advantage, however.

The second disadvantage is the fragmentation of the disk that results from this allocation policy. Space is wasted that might otherwise have been used. Compaction of the disk is usually prohibitively expensive, although it can conceivably be done late at night when the system is otherwise idle.

Linked List Allocation

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig. 5-8. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block). Also, it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there.

On the other hand, although reading a file sequentially is straightforward, random access is extremely slow. Also, the amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two.

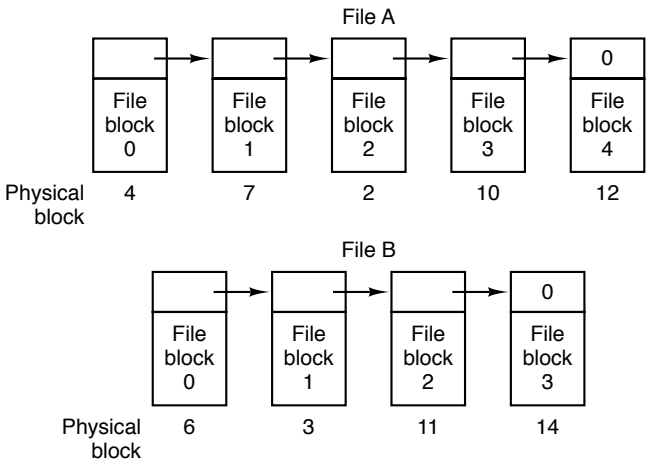


Figure 5-8. Storing a file as a linked list of disk blocks.

Linked List Allocation Using an Index

Both disadvantages of the linked list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table or index in memory. Figure 5-9 shows what the table looks like for the example of Fig. 5-8. In both figures, we have two files. File A uses disk blocks 4, 7, 2, 10, and 12, in that order, and file B uses disk blocks 6, 3, 11, and 14, in that order. Using the table of Fig. 5-9, we can start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6.

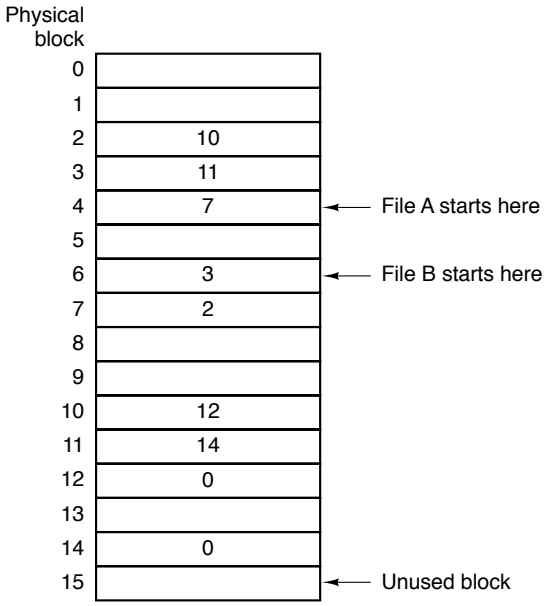


Figure 5-9. Linked list allocation using a table in main memory.

Using this organization, the entire block is available for data. Furthermore, random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is. MS-DOS uses this method for disk allocation.

The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a large disk, say, 500,000 1K blocks (500M), the table will have 500,000 entries, each of which will have to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 1.5 or 2 megabytes all the time depending on whether the system is optimized for space or time. Although MS-DOS uses this mechanism, it avoids huge tables by using large blocks (up to 32K) on large disks.

I-nodes

Our last method for keeping track of which blocks belong to which file is to associate with each file a little table called an i-node (index-node), which lists the attributes and disk addresses of the file's blocks, as shown in Fig. 5-10.

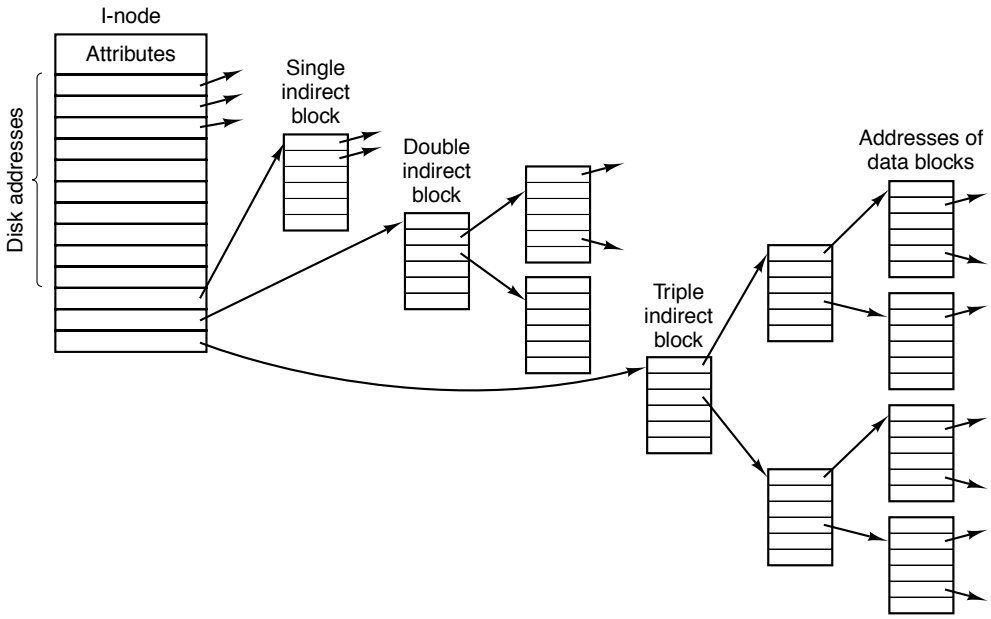


Figure 5-10. An i-node.

The first few disk addresses are stored in the i-node itself, so for small files, all the necessary information is right in the i-node, which is fetched from disk to main memory when the file is opened. For somewhat larger tiles, one of the addresses in the i-node is the address of a disk block called a single indirect block. This block contains additional disk addresses. If this still is not enough, another address in the i-node, called a double indirect block, contains the address of a block that contains a list of single indirect blocks. Each of these single indirect blocks points to a few hundred data blocks. If even this is not enough, a triple indirect block can also be used. UNIX uses this scheme.

5.3.2 Implementing Directories

Before a file can be read, it must be opened. When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry. The directory entry provides the information needed to find the disk blocks. Depending on the system, this information may be the disk address of the entire file (contiguous allocation), the number of the first block (both linked list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

A closely related issue is where the attributes should be stored. One obvious possibility is to store them directly in the directory entry. Many systems do precisely that. For systems that use i-nodes, another possibility is to store the attributes in the i-node, rather than in the directory entry. As we shall see later, this method has certain advantages over putting them in the directory entry.

Directories in CP/M

Let us start our study of directories with a particularly simple example, that of CP/M (Golden and Pechura, 1986), illustrated in Fig. 5-11. In this system, there is only one directory, so all the file system has to do to look up a file name is search the one and only directory. When it finds the entry, it also has the disk block numbers, since they are stored right in the directory entry, as are all the attributes. If the tile uses more disk blocks than fit in one entry, the file is allocated additional directory entries.

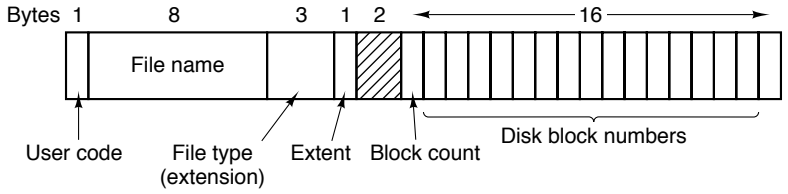


Figure 5-11. A directory entry that contains the disk block numbers for each file.

The fields in Fig. 5-11 have the following meanings. The User code field keeps track of which user owns the file. During a search, only those entries belonging to the currently logged-in user are checked. The next two fields give the name and extension of the file. The Extent field is needed because a file larger than 16 blocks occupies multiple directory entries. This field is used to tell which entry comes first, second, and soon. The Block count field tells how many of the 16 potential disk block entries are in use. The final 16 fields contain the disk block numbers themselves. The last block may not be full, so the system has no way to determine the exact size of a file down to the last byte (i.e., it keeps track of file sizes in blocks, not bytes).

Directories in MS-DOS

Now let us consider some examples of systems with hierarchical directory trees. Figure 5-12 shows an MS-DOS directory entry. It is 32 bytes long and contains the file name, attributes, and the number of the first disk block. The first block number is used as an index into a table of the type of Fig. 5-9. By following the chain, all the blocks can be found.

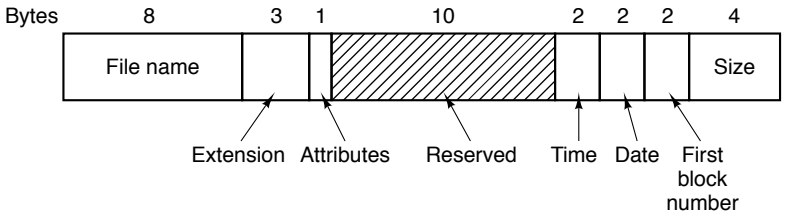


Figure 5-12. The MS-DOS directory entry.

In MS-DOS, directories may contain other directories, leading to a hierarchical file system. It is common in MS-DOS that different application programs each start out by creating a directory in the root directory and putting all their files there, so that different applications do not conflict.

Directories in UNIX

The directory structure traditionally used in UNIX is extremely simple, as shown in Fig. 5-13. Each entry contains just a file name and its i-node number. All the information about the type, size, times, ownership, and disk blocks is contained in the i-node. Some UNIX systems have a different layout, but in all cases, a directory entry ultimately contains only an ASCII string and an i-node number.

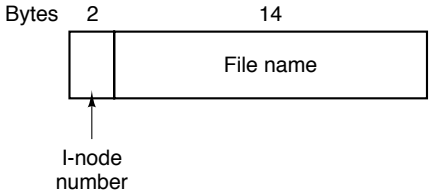


Figure 5-13. A UNIX directory entry.

When a file is opened, the file system must take the file name supplied and locate its disk blocks. Let us consider how the path name */usr/ast/mbox* is looked up. We will use UNIX as an example, but the algorithm is basically the same for all hierarchical directory systems. First the file system locates the root directory. In UNIX its i-node is located at a fixed place on the disk.

Then it looks up the first component of the path, *usr*, in the root directory to find the i-node number of the file */usr*. Locating an i-node from its number is straightforward, since each one has a fixed location on the disk. From this i-node, the system locates the directory for */usr* and looks up the next component, *ast*, in it. When it has found the entry for *ast*, it has the i-node for the directory */usr/ast*. From this i-node it can find the directory itself and look up *mbox*. The i-node for this file is then read into memory and kept there until the file is closed. The lookup process is illustrated in Fig. 5-14.

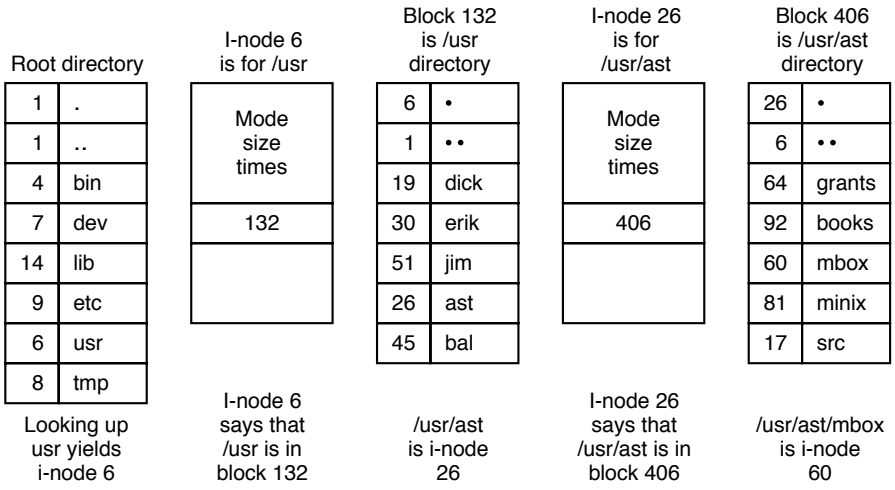


Figure 5-14. The steps in looking up */usr/ast/mbox*.

Relative path names are looked up the same way as absolute ones, only starting from the working directory instead of starting from the root directory. Every directory has entries for . and .. which are put there when the directory is created. The entry . has the i-node number for the current directory, and the entry for .. has the i-node number for the parent directory. Thus, a procedure looking up *../dick/prog.c* simply looks up .. in the working directory, finds the i-node number for the parent directory, and searches that directory for *dick*. No special mechanism is needed to handle these names. As far as the directory system is concerned, they are just ordinary ASCII strings, just the same as any other names.

5.3.3 Disk Space Management

Files are normally stored on disk, so management of disk space is a major concern to file system designers. Two general strategies are possible for storing an n byte file: n consecutive bytes of disk space are allocated, or the tile is split up into a number of (not necessarily) contiguous blocks. The same trade-off is present in memory management systems between pure segmentation and paging.

Storing a file as a contiguous sequence of bytes has the obvious problem that if a file grows, it will probably have to be moved on the disk. The same problem holds for segments in memory, except that moving a segment in memory is a relatively fast operation compared to moving a file from one disk position to another. For this reason, nearly all file systems chop files up into fixed-size blocks that need not be adjacent.

Block Size

Once it has been decided to store files in fixed-size blocks, the question arises of how big the block should be. Given the way disks are organized, the sector, the track and the cylinder are obvious candidates for the unit of allocation. In a paging system, the page size is also a major contender.

Having a large allocation unit, such as a cylinder, means that every file, even a I-byte file, ties up an entire cylinder. Studies (Mullender and Tanenbaum, l984) have shown that the median file size in UNIX environments is about IK, so allocating a 32K cylinder for each file would waste 31/32 or 97 percent of the total disk space. On the other hand, using a small allocation unit means that each file will consist of many blocks. Reading each block normally requires a seek and a rotational delay, so reading a file consisting of many small blocks will be slow.

As an example, consider a disk with 32,768 bytes per track, a rotation time of 16.67 msec, and an average seek time of 30 msec. The time in milliseconds to read a block of k bytes is then the sum of the seek, rotational delay, and transfer times:

$$30 + 8.3 + (k/32768) \text{ x } 16.67$$

The solid curve of Fig. 5-15 shows the data rate for such a disk as a function of block size. If we make the gross assumption that all files are 1K (the measured median size), the dashed curve of Fig. 5-15 gives the disk space efficiency. The bad news is that good space utilization (block size ¡ 2K) means low data rates and vice versa. Time efficiency and space efficiency are inherently in conflict.

The usual compromise is to choose a block size of 512, 1K or 2K bytes. If a 1K block size is chosen on a disk with a 512-byte sector size, then the file system will always read or write two consecutive sectors and treat them as a single, indivisible unit. Whatever

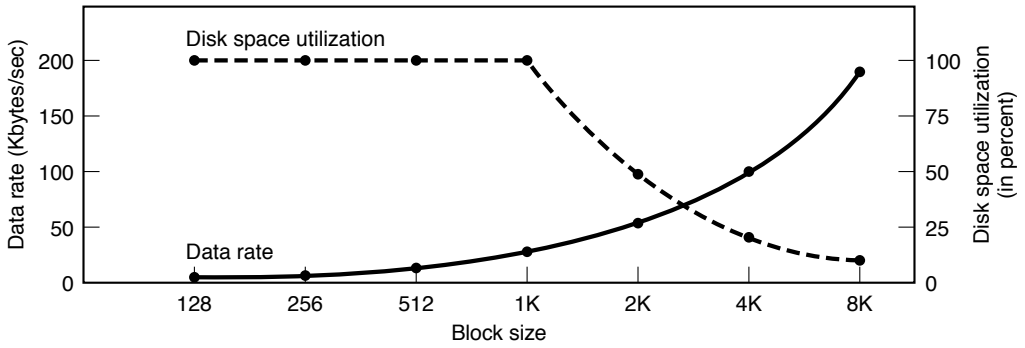


Figure 5-15. The solid curve (left-hand scale) gives the data rate of a disk. The dashed curve (right-hand scale) gives the disk space efficiency. All tiles are 1K.

decision is made, it should probably be reevaluated periodically, since, as with all aspects of computer technology, users take advantage of more abundant resources by demanding even more. One system manager reports that the average size of files in the university system he manages has increased slowly over the years, and that in 1997 the average size of files has grown to 12K for students and 15K for faculty.

Keeping Track of Free Blocks

Once a block size has been chosen, the next issue is how to keep track of free blocks. Two methods are widely used, as shown in Fig. 5-16. The first one consists of using a linked list of disk blocks, with each block holding as many free disk block numbers as will fit. With a 1K block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is needed for the pointer to the next block). A 200M disk needs a free list of maximum 804 blocks to hold all 200K disk block numbers. Often free blocks are used to hold the free list.

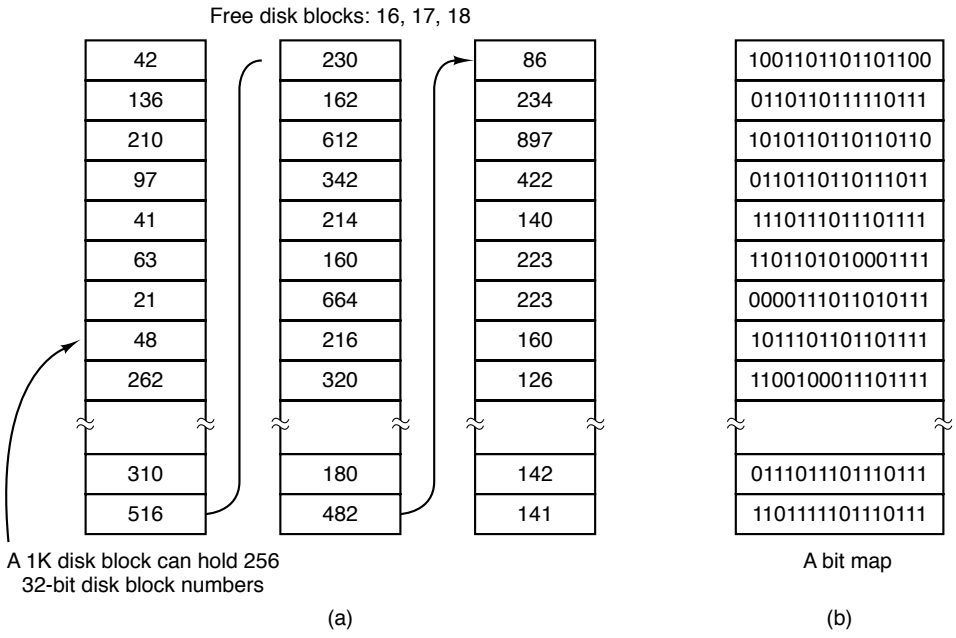


Figure 5-16. (a) Storing the free list on a linked list. (b) A bit map.

The other free space management technique is the bit map. A disk with n blocks requires a bit map with n bits. Free blocks are represented by 1s in the map, allocated blocks by 0s (or, vice versa). A 200M disk requires 200K bits for the map, which requires only 25 blocks. It is not surprising that the bit map requires less space, since it uses l bit per block, versus 32 bits in the linked list model. Only if the disk is nearly full will the linked list scheme require fewer blocks than the bit map.

If there is enough main memory to hold the bit map, that method is generally preferable. If, however, only 1 block of memory can be spared for keeping track of free disk blocks, and the disk is nearly full, then the linked list may be better. With only 1 block of the bit map in memory, it may turn out that no free blocks can be found on it, causing disk accesses to read the rest of the bit map. When a fresh block of the linked list is loaded into memory, 255 disk blocks can be allocated before having to go to the disk to fetch the next block from the list.

5.3.4 File System Reliability

Destruction of a file system is often a far greater disaster than destruction of a computer. If a computer is destroyed by fire, lightning surges, or a cup of coffee poured onto the keyboard, it is annoying and will cost money, but generally a replacement can be purchased with a minimum of fuss. Inexpensive personal computers can even be replaced within a few hours by just going to the dealer (except at universities where issuing a purchase order takes three committees, five signatures, and 90'days).

If a computer's file system is irrevocably lost, whether due to hardware, software, or rats gnawing on the floppy disks, restoring all the information will be difficult, time consuming, and in many cases, impossible. For the people whose programs, documents, customer files, tax records, data bases, marketing plans, or other data are gone forever, the consequences can be catastrophic. While the file system cannot offer any protection against physical destruction of the equipment and media, it can help protect the information. In this section we will look at some of the issues involved in safeguarding the file system.

Disks may have bad blocks, as we pointed out in Chap. 3. Floppy disks are generally perfect when they leave the factory, but they can develop bad blocks during use. Winchester disks frequently have bad blocks right from the start: it is just too expensive to manufacture them completely free of all defects. In fact, older hard disks used to be supplied with a list of the bad blocks discovered by the manufacturer's tests. On such disks a sector is reserved for a bad block list. When the controller is first initialized, it reads the bad block list and picks a spare block (or track) to replace the defective ones, recording the mapping in the bad block list. Henceforth, all requests for the bad block will use the spare. When new errors are discovered this list is updated as part of a low-level format.

There has been a steady improvement in manufacturing techniques, so bad blocks are less common than they once were. However, they still occur. The controller on a modem disk drive is very sophisticated, as noted in Chap. 3. On these disks, tracks are at least one sector bigger than needed, so that at least one bad spot can be skipped by leaving it in a gap between two consecutive sectors. There are also a few spare sectors per cylinder so the controller can do automatic sector remapping if it notices that a sector needs more than a certain number of retries to-be read or written. Thus the user is usually unaware of bad blocks or their management. Nevertheless, when a modem IDE or SCSI disk fails, it will usually fail horribly, because it has run out of spare sectors. SCSI disks provide a

"recovered error" when they remap a block. If the driver notes this and prints a message on the keyboard the user will know it is time to buy a new disk when these messages begin to appear frequently.

There is a simple software solution to the bad block problem, suitable for use on older disks. This approach requires the user or file system to carefully construct a file containing all the bad blocks. This technique removes them from the free list, so they will never occur in data files. As long as the bad block file is never read or written, no problems will arise. Care has to be taken during disk backups to avoid reading this tile.

Backups

Even with a clever strategy for dealing with bad blocks, it is important to back up the files frequently. After all, automatically switching to a spare track after a crucial data block has been ruined is somewhat akin to locking the barn door after the prize race horse has escaped.

File systems on floppy disk can be backed up by just copying the entire floppy disk to a blank one. File systems on small winchester disks can be backed up by dumping the entire disk to magnetic tape. Current technologies include l50M cartridge tapes, and 8G Exabyte or DAT tapes.

For large winchesters (e.g., 10GB), backing up the entire drive on tape is awkward and time consuming. One strategy that is easy to implement but wastes half the storage is to provide each computer with two drives instead of one. Both drives are divided into two halves: data and backup. Each night the data portion of drive 0 is copied to the backup portion of drive I, and vice versa, as shown in Fig. 5-17. In this way, if one drive is completely ruined, no information is lost.

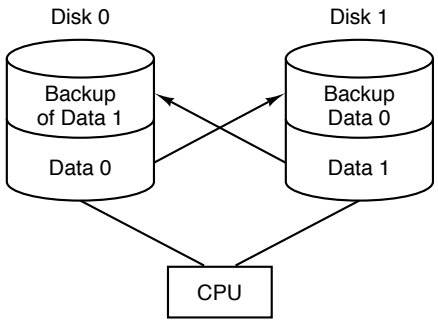


Figure 5-17. Backing up each drive on the other one wastes half the storage.

An alternative to dumping the entire file system every day is to make **incremental dumps**. The simplest form of incremental dumping is to make a complete dump periodically, say weekly or monthly, and to make a daily dump of only those files that have been modified since the last full dump. A better scheme is to dump only those files that have changed since they were last dumped.

To implement this method, a list of the dump times for each file must be kept on disk. The dump program then checks each file on the disk. If it has been modified since it was last dumped, it is dumped again and its time-of-last-dump is changed to the current time. If done on a monthly cycle, this method requires 3l daily dump tapes, one per day, plus enough tapes to hold a full dump, made once a month. Other more complex schemes that use fewer tapes are also in use.

Automatic methods using multiple disks are also used. For example, **mirroring** uses two disks. Writes go to both disks, and reads come from one. The write to the mirror disk is delayed a bit, so it can be done when the system is idle. Such a system can continue to run in "degraded mode" when one disk fails, allowing a failed disk to be swapped and data to be recovered with no downtime.

File System Consistency

Another area where reliability is an issue is file system consistency. Many file systems read blocks, modify them, and write them out later. If the system crashes before all the modified blocks have been written out, the file system can be left in an inconsistent state. This problem is especially critical if some of the blocks that have not been written out are i-node blocks, directory blocks, or 4 blocks containing the free list.

To deal with the problem of inconsistent file systems, most computers have a utility program that checks file system consistency. It can be run whenever the system is booted, especially after a crash. The description below tells how such a utility works in UNIX and MINIX; other systems have something similar. These file system checkers verify each file system (disk) independently of the other ones.

Two kinds of consistency checks can be made: blocks and files. To check for block consistency, the program builds two tables, each one containing a counter for each block, initially set to 0. The counters in the first table keep track of how many times each block is present in a file; the counters in the second table record how often each block is present in the free list (or the bit map of free blocks).

The program then reads all the i-nodes. Starting from an i-node, it is possible to build a list of all the block numbers used in the corresponding file. As each block number is read, its counter in the first table is incremented. The program then examines the free list or bit map, to find all the blocks that are not in use. Each occurrence of a block in the free list results in its counter in the second table being incremented.

If the tile system is consistent, each block will have a 1 either in the first table or in the second table, as illustrated in Fig. 5-18(a). However, as a result of a crash, the tables might look like Fig. 5-18(b), in which block 2 does not occur in either table. It will be reported as being a **missing block**. While missing blocks do no real harm, they do waste space and thus reduce the capacity of the disk. The solution to missing blocks is straightforward: the file system checker just adds them to the free list.

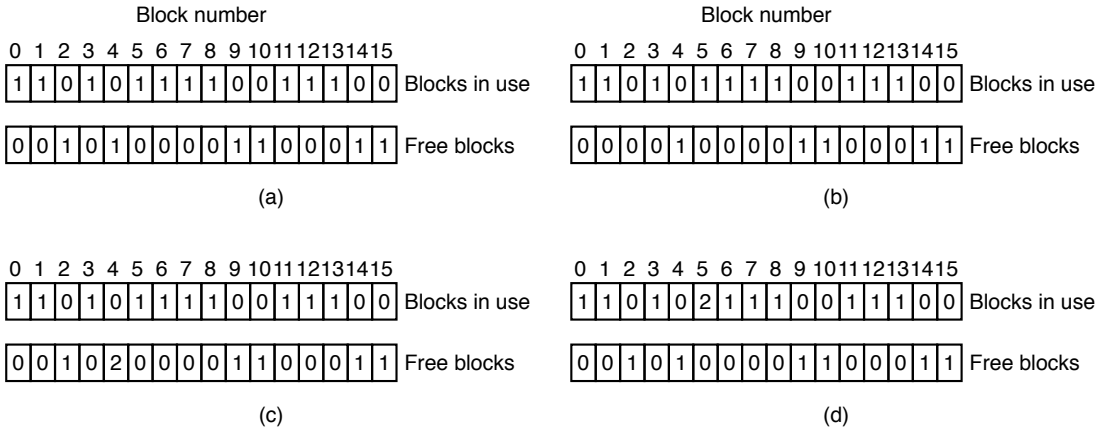


Figure 5-18. File system states. (a) Consistent. (b) Missing block. (c) duplicate block in free list. (d) Duplicate data block.

Another situation that might occur is that of Fig. 5-18(c). Here wet see a block, number 4, that occurs twice in the free list. (Duplicates can occur only if the free list is really a list; with a bit map it is impossible.) The solution here is also simple: rebuild the free list.

The worst thing that can happen is that the same data block is present in two or more files, as shown in Fig. 5-18(d) with block 5. If either of these files is removed, block 5 will be put on the free list, leading to a situation in which the same block is both in use and free at the same time. If both files are removed, the block will be put onto the free list twice.

The appropriate action for the file system checker to take is to allocate a free block, copy the contents of block 5 into it, and insert the copy into one of the files. In this way, the information content of the files is unchanged (although almost assuredly garbled), but the file system structure is at least made consistent. The error should be reported, to allow the user to inspect the damage.

In addition to checking to see that each block is properly accounted for, the file system checker also checks the directory system. It too, uses a table of counters, but these are per file, rather than per block. It starts at the root directory and recursively descends the tree, inspecting each directory in the file system. For every file in every directory, it increments the counter for that file's i-node (see Fig. 5-13 for the layout of a directory entry).

When it is all done, it has a list, indexed by i-node number, telling how many directories point to that i-node. It then compares these numbers with the link counts stored in the i-nodes themselves. In a consistent file system, both counts will agree. However, two kinds of errors can occur: the link count in the i-node can be too high or it can be too low.

If the link count is higher than the number of directory entries, then even if all the files are removed from the directories, the count will still be nonzero and the i-node will not be removed. This error is not serious, but it wastes space on the disk with tiles that are not in any directory. It should be fixed by setting the link count in the i-node to the correct value.

The other error is potentially catastrophic. If two directory entries are linked to a file, but the i-node says that there is only one, when either directory entry is removed, the i-node count will go to zero. When an i-node count goes to zero, the file system marks it as unused and releases all of its blocks. This action will result in one of the directories now pointing to an unused i-node, whose blocks may soon be assigned to other files. Again, the solution is just to force the link count in the i-node to the actual number of directory entries.

These two operations, checking blocks and checking directories, are often integrated for efficiency reasons (i.e., only one pass over the i-nodes is required). Other heuristic checks are also possible. For example, directories have a definite format, with i-node numbers and ASCII names. If an i-node number is larger than the number of i-nodes on the disk, the directory has been damaged.

Furthermore, each i-node has a mode, some of which are legal but strange, such as 0007, which allows the owner and his group no access at all, but allows outsiders to read, write, and execute the tile. It might be useful to at least report files that give outsiders more rights than the owner. Directories with more than, say, l000 entries are also suspicious. Files located in user directories, but which are owned by the super-user and have the SETUID bit on, are potential security problems. With a little effort, one can put together a fairly long list of legal, but peculiar, situations that might be worth reporting.

The previous paragraphs have discussed the problem of protecting the user against crashes. Some file systems also worry about protecting the user against himself. If the

user intends to type

```
rm *.o
```

to remove all the files ending with *.o* (compiler generated object files), but accidentally types

```
rm * .o
```

(note the space after the asterisk), *rm* will remove all the files in the current directory and then complain that it cannot find *.o*. In MS-DOS and some other systems, when a file is removed, all that happens is that a bit is set in the directory or i-node marking the file as removed. No disk blocks are returned to the free list until they are actually needed. Thus, if the user discovers the error immediately, it is possible to run a special utility program that "unremoves" (i.e., restores) the removed files. In WINDOWS 95, files that are removed are placed in a special recycled directory, from which they can later be retrieved if need be. Of course, no storage is reclaimed until they are actually deleted from this directory.

5.3.5 File System Performance

Access to disk is much slower than access to memory. Reading a memory word typically takes tens of nanoseconds. Reading a block from a hard disk may take fifty microseconds, a factor of four slower per 32-bit word, but to this must be added l0 to 20 milliseconds to seek to the track and then wait for the desired sector to arrive under the read head. If only a single word is needed, the memory access is of the order of 100,000 times as fast as disk access. As a result of this A difference in access time, many file systems have been designed to reduce the number of disk accesses needed.

The most common technique used to reduce disk accesses is the **block cache** or **buffer cache**. (Cache is pronounced "cash," and is derived from the French cacher, meaning to hide.) In this context, a cache is a collection of blocks that logically belong on the disk but are being kept in memory for performance reasons.

Various algorithms can be used to manage the cache, but a common one is to check all read requests to see if the needed block is in the cache. If it is, the read request can be satisfied without a disk access. If the block is not in the cache, it is first read into the cache, and then copied to wherever it is needed. Subsequent requests for the same block can be satisfied from the cache.

When a block has to be loaded into a full cache, some block has to be removed and rewritten to the disk if it has been modified since being brought in. This situation is very much like paging, and all the usual paging algorithms described in Chap. 4, such as FIFO, second chance, and LRU, are applicable. One pleasant difference between paging and caching is that cache references are relatively infrequent, so that it is feasible to keep all the blocks in exact LRU order with linked lists.

Unfortunately, there is a catch. Now that we have a situation in which exact LRU is possible, it turns out that LRU is undesirable. The problem has to do with the crashes and file system consistency discussed in the previous section. If a critical block, such as an i-node block, is read into the cache and modified, but not rewritten to the disk, a crash will leave the file system in an inconsistent state. If the i-node block is put at the end of the LRU chain, it may be quite a while before it reaches the front and is rewritten to the disk.

Furthermore, some blocks, such as double indirect blocks, are rarely referenced two times within a short interval. These considerations lead to a modified LRU scheme, taking two factors into account:

1. Is the block likely to be needed again soon?

2. Is the block essential to the consistency of the file system?

For both questions, blocks can be divided into categories such as i-node blocks, indirect blocks, directory blocks, full data blocks, and partly full data blocks. Blocks that will probably not be needed again soon go on the front, rather than the rear of the LRU list, so their buffers will be reused quickly. Blocks that might be needed again soon, such as a partly full block that is being written, go on the end of the list, so they will stay around for a long time.

The second question is independent of the first one. If the block is essential to the tile system consistency (basically, everything except data blocks), and it has been modified, it should be written to disk immediately, regardless of which end of the LRU list it is put on. By writing critical blocks quickly, we greatly reduce the probability that a crash will wreck the file system.

Even with this measure to keep the file system integrity intact, it is undesirable to keep data blocks in the cache too long before writing them out. Consider the plight of someone who is using a personal computer to write a book. Even if our writer periodically tells the editor to write the file being edited to the disk, there is a good chance that everything will still be in the cache and nothing on the disk. If the system crashes, the file system structure will not be corrupted, but a whole day's work will be lost.

This situation need not happen very often before we have a fairly unhappy user. Systems take two approaches to dealing with it. The UNIX way is to have a system call, SYNC, which forces all the modified blocks out onto the disk immediately. When the system is started up, a program, usually called *update*, is started up in the background to sit in an endless loop issuing SYNC calls, sleeping for 30 sec between calls. As a result, no more than 30 seconds of work is lost due to a crash.

The MS-DOS way is to write every modified block to disk as soon as it has been written. Caches in which all modified blocks are written back to the disk immediately are called **write-through caches**. They require much more disk I/O than nonwrite-through caches. The difference between these two approaches can be seen when a program writes a 1K block full, one character at a time. UNIX will , collect all the characters in the cache and write the block out once every 30 seconds, or whenever the block is removed from the cache. MS.DOS will make a disk access for every character written. Of course, most programs do internal buffering, so they normally write not a character, but a line or a larger unit on each WRITE system call.

A consequence of this difference in caching strategy is that just removing a (floppy) disk from a UNIX system without doing a SYNC will almost always result in lost data, and frequently in a corrupted file system as well. With MS-DOS, no problem arises. These differing strategies were chosen because UNIX was developed in an environment in which all disks were hard disks and not removable, whereas MS-DOS started out in the floppy disk world. As hard disks become the norm, even on small microcomputers, the UNIX approach, with its better efficiency, will definitely be the way to go.

Caching is not the only way to increase the performance of a file system. Another important technique is to reduce the amount of disk arm motion by putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder.

When an output file is written, the file system has to allocate the blocks one at a time, as they are needed. If the free blocks are recorded in a bit map, and the whole bit map is in main memory, it is easy enough to choose a free block as close as possible to the previous block. With a free list, part of which is on disk, it is much harder to allocate blocks close together.

However, even with a free list, some block clustering can be done. The trick is to keep track of disk storage not in blocks, but in groups of consecutive blocks. If a track consists of 64 sectors of 512 bytes, the system could use 1K blocks (2 sectors), but allocate disk storage in units of 2 blocks (4 sectors). This is not the same as having a 2K disk block, since the cache would still use 1K blocks and disk transfers would still be 1K but reading a file sequentially on an otherwise idle system would reduce the number of seeks by a factor of two, considerably improving performance.

A variation on the same theme is to take account of rotational positioning. When allocating blocks, the system attempts to place consecutive blocks in a file in the same cylinder, but interleaved for maximum throughput. Thus, if a disk has a rotation time of 16.67 msec and it takes about 4 msec for a user process to request and get a disk block, each block should be placed at least a quarter of the way around from its predecessor.

Another performance bottleneck in systems that use i-nodes or anything equivalent to i-nodes is that reading even a short file requires two disk accesses: one for the i-node and one for the block. The usual i-node placement is shown in Fig. 5-19(a). Here all the i-nodes are near the beginning of the disk, so the average distance between an i-node and its blocks will be about half the number of cylinders, requiring long seeks.

One easy performance improvement is to put the i-nodes in the middle of the disk, rather than at the start, thus reducing the average seek between the i-node and the first block by a factor of two. Another idea, shown in Fig. 5-19(b), is to divide the disk into

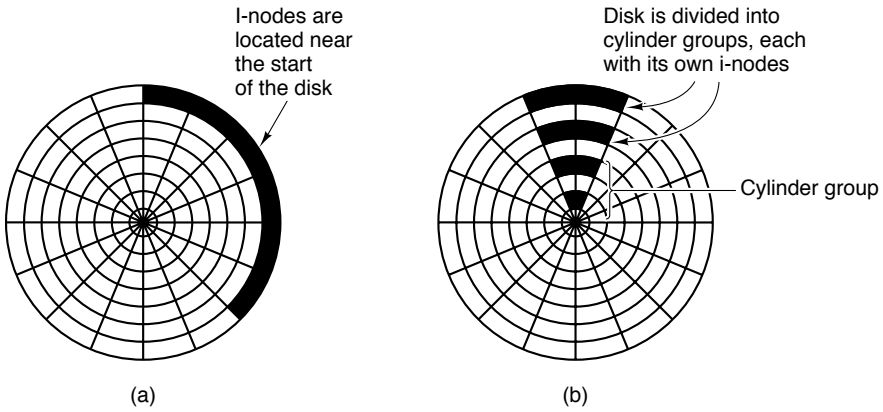


Figure 5-19. (a) l-nodes placed at the start of the disk. (b) Disk divided into cylinder groups. each with its own blocks and i-nodes.

cylinder groups, each with its own i-nodes. blocks. and free list (McKusick et al., l984). When creating a new file, any i-node can be chosen, but an attempt is made to find a block in the same cylinder group as the i-node. If none is available, then a block in a nearby cylinder group is used.

5.3.6 Log-Structured File Systems

Changes in technology are putting pressure on current file systems. In particular, CPUs keep getting faster, disks are becoming much bigger and cheaper (but not much

faster), and memories are growing exponentially in size. The one parameter that is not improving by leaps and bounds is disk seek time. The combination of these factors means that a performance bottleneck is arising in many file systems. Research done at Berkeley attempted to alleviate this problem by designing a completely new kind of file system, LFS (the **Log-structured File System**). In this section we will briefly describe how LFS works. For a more complete treatment, see (Rosenblum and Ousterhout. l99l).

The idea that drove the LFS design is that as CPUs get faster and RAM memories get larger, disk caches are increasingly rapidly. As a consequence, it is now possible to satisfy a very substantial fraction of all read requests directly from the file system cache, with no disk access needed. It follows from this observation, that in the future, most disk accesses will be writes, so the read-ahead mechanism used in some file systems to fetch blocks before they are needed no longer gains much performance.

To make matters worse, in most file systems, writes are done in very small chunks. Small writes are highly inefficient, since a 50-microsec disk write is typically preceded by a 10-msec seek and a 6-msec rotational delay. With these parameters, disk efficiency drops to a fraction of l percent.

To see where all the small writes come from, consider creating a new file on a UNIX system. To write this tile, the i-node for the directory, the directory block, the i-node for the tile, and the tile itself must all be written. While these writes can be delayed, doing so exposes the file system to serious consistency problems if a crash occurs before the writes are done. For this reason, the i-node writes are generally done immediately.

From this reasoning, the LFS designers decided to re-implement the UNIX file system in such a way as to achieve the full bandwidth of the disk, even in the face of a workload consisting in large part of small random writes. The basic idea is to structure the entire disk as a log. Periodically, and when there is a special need for it, all the pending writes being buffered in memory are collected into a single segment and written to the disk as a single contiguous segment at the end of the log. A single segment may thus contain i-nodes, directory blocks, and data blocks, all mixed together. At the start of each segment is a segment summary, telling what can be found in the segment. If the average segment can be made to be about 1 MB, almost the full bandwidth of the disk can be utilized.

In this design, i-nodes still exist and have the same structure as in UNIX, but they are now scattered all over the log, instead of being at a fixed position on the disk. Nevertheless, when an i-node is located, locating the blocks is done in the usual way. Of course, finding an i-node is now much harder, since its address cannot simply be calculated from its i-number, as in UNIX. To make it possible to find i-nodes, an i-node map, indexed by i-number, is maintained. Entry i in this map points to i-node I on the disk. The map is kept on disk, but it is also cached, so the most heavily used parts will be in memory most of the time.

To summarize what we have said so far, all writes are initially buffered in memory, and periodically all the buffered writes are written to the disk in a single segment, at the end of the log. Opening a file now consists of using the map to locate the i-node for the file. Once the i-node has been located, the addresses of the blocks can be found from it. All of the blocks will themselves be in segments, somewhere in the log.

If disks were infinitely large, the above description would be the entire story. However, real disks are finite, so eventually the log will occupy the entire disk, at which time no new segments can be written to the log. Fortunately, many existing segments may have blocks that are no longer needed, for example, if a file is overwritten, its i-node will now point to the new blocks, but the old ones will still be occupying space in previously written segments.

To deal with both of these problems, LFS has a **cleaner** thread that spends its time scanning the log circularly to compact it. It starts out by reading the summary of the first segment in the log to see which i-nodes and files are there. It then checks the current i-node map to see if the i-nodes are still current and file blocks are still in use. If not, that information is discarded. The i-nodes and blocks that are still in use go into memory to be written out in the next segment. The original segment is then marked as free, so the log can use it for new data. In this manner, the cleaner moves along the log, removing old segments from the back and putting any live data into memory for rewriting in the next segment. Consequently, the disk is a big circular buffer, with the writer thread adding new segments to the front and the cleaner thread removing old ones from the back.

The bookkeeping here is nontrivial, since when a tile block is written back to a new segment, the i-node of the file (somewhere in the log) must be located, updated, and put into memory to be written out in the next segment. The i-node map must then be updated to point to the new copy. Nevertheless, it is possible to do the administration, and the performance results show that all this complexity is worthwhile. Measurements given in the papers cited above show that LFS outperforms UNIX by an order of magnitude on small writes, while having a performance that is as good or better than UNIX for reads and large writes.

5.4 SECURITY $\surd$

File systems often contain information that is highly valuable to their users. Protecting this information against unauthorized usage is therefore a major concern of all file systems. In the following sections we will look at a variety of issues concerned with security and protection. These issues apply equally well to timesharing systems as to networks of personal computers connected to shared servers via local area networks.

5.4.1 The Security Environment

The terms "security" and "protection" are often used interchangeably. Nevertheless, it is frequently useful to make a distinction between the general problems involved in making sure that files are not read or modified by unauthorized persons, which include technical, managerial, legal, and political issues on the one hand, and the specific operating system mechanisms used to provide security, on the other. To avoid confusion, we will use the term **security** to refer to the overall problem, and the term **protection mechanisms** to refer to the specific operating system mechanisms used to safeguard information in the computer. The boundary between them is not well defined, however. First we will look at security; later on in the chapter we will look at protection.

Security has many facets. Two of the more important ones are data loss and intruders. Some of the common causes of data loss are:

1. Acts of God: fires, floods, earthquakes, wars, riots, or rats gnawing tapes or floppy disks.

2. Hardware or software errors: CPU malfunctions, unreadable disks or tapes, telecommunication errors, program bugs.

3. Human errors: incorrect data entry, wrong tape or disk mounted, wrong program run, lost disk or tape, or some other mistake.

Most of these can be dealt with by maintaining adequate backups, preferably far away from the original data.

A more interesting problem is what to do about intruders. These come in two varieties. Passive intruders just want to read files they are not authorized to read. Active intruders are more malicious; they want to make unauthorized changes to data. When designing a system to be secure against intruders, it is important to keep in mind the kind of intruder one is trying to protect against. Some common categories are:

1. Casual prying by nontechnical users. Many people have terminals to timesharing systems or networked personal computers on their desks, and human nature being what it is, some of them will read other people's electronic mail and other files if no barriers are placed in the way. Most UNIX systems, for example, have the default that all files are publicly readable.

2. Snooping by insiders. Students, system-programmers, operators, and other technical personnel often consider it to be a personal challenge to break the security of the local computer system. They often are highly skilled and are willing to devote a substantial amount of time to the effort.

3. Determined attempt to make money. Some bank programmers have attempted to break into a banking system to steal from the bank. Schemes have varied from changing the software to truncate rather than round interest, keeping the fraction of a cent for themselves, to siphoning off accounts not used in years, to blackmail ("Pay me or I will destroy all the bank's records.").

4. Commercial or military espionage. Espionage refers to a serious and well-funded attempt by a competitor or a foreign country to steal programs, trade secrets, patents, technology, circuit designs, marketing plans, and so forth. Often this attempt will involve wiretapping or even erecting antennas directed at the computer to pick up its electromagnetic radiation.

It should be clear that trying to keep a hostile foreign government from stealing military secrets is quite a different matter from trying to keep students from inserting a funny message-of-the-day into the system. The amount of effort that one puts into security and protection clearly depends on who the enemy is thought to be.

Another aspect of the security problem is **privacy**: protecting individuals from misuse of information about them. This quickly gets into many legal and moral issues. Should the government compile dossiers on everyone in order to catch X-cheaters, where X is "welfare" or "tax," depending on your politics? Should the police be able to look up anything on anyone in order to stop organized crime? Do employers and insurance companies have rights? What happens when these rights conflict with individual rights? All of these issues are extremely important but are beyond the scope of this book.

5.4.2 Famous Security Flaws

Just as the transportation industry has the *Titanic* and the *Hindenburg*, computer security experts have a few things they would rather forget about. In this section we will look at some interesting security problems that have occurred in three different operating systems: UNIX, TENEX, and OS/360.

The UNIX utility *lpr*, which prints a file on the line printer, has an option to remove the file after it has been printed. In early versions of UNIX it was possible for anyone to use *lpr* to print, and then have the system remove, the password file.

Another way to break into UNIX was to link a file called *core* in the working directory to the password tile. The intruder then forced a core dump of a SETUID program. which the system wrote on the *core* file, that is, on top of the password tile. ln this way, a user could replace the password file with one containing a few strings of his own choosing (e.g., command arguments).

Yet another subtle flaw in UNIX involved the command

```
mkdir foo
```

Mkdir, which was a SETUID program owned by the root, first created the i-node for the directory *foo* with the system call MKNOD and then changed the owner of *foo* from its effective uid (i.e., root) to its real uid (the user's uid). When the system was slow, it was sometimes possible for the user to quickly remove the directory i-node and make a link to the password file under the name *foo* after the MKNOD but before the CHOWN. When *mkdir* did the CHOWN it made the user the owner of the password tile. By putting the necessary commands in a shell script, they could be tried over and over until the trick worked.

The TENEX operating system used to be very popular on the DEC-10 com- puters. It is no longer used. but it will live on forever in the annals of computer security due to the following design error, TENEX supported paging. To allow users to monitor the behavior of their programs, it was possible to instruct the sys- tem to call a user function on each page fault.

TENEX also used passwords to protect files. To access a tile, a program had to present the proper password. The operating system checked passwords one char- acter at a time, stopping as soon as it saw that the password was wrong. To break into TENEX an intruder would carefully position a password as shown in Fig. 5-20(a), with the first character at the end of one page, and the rest at the start of the next page.

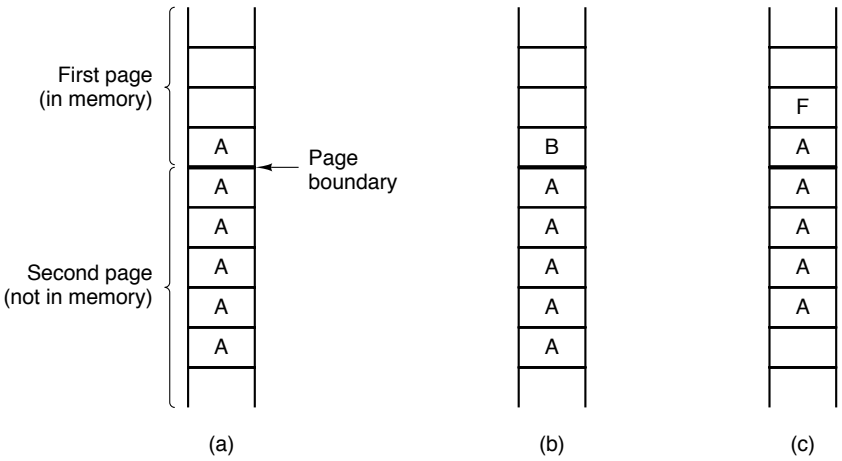


Figure 5-20. The TENEX password problem.

The next step was to make sure that the second page was not in memory, for example, by referencing so many other pages that the second page would surely be evicted to make room for them. Now the program tried to open the victim's file, using the carefully aligned password. If the first character of the real password was anything but *A*, the system would

stop checking at the first character and report back with ILLEGAL PASSWORD. If, however, the real password did begin with *A*, the system continued reading, and got a page fault, about which the intruder was informed.

If the password did not begin with *A*, the intruder changed the password to that of Fig. 5-20(b) and repeated the whole process to see if it began with *B*. It took at most 128 tries to go through the whole ASCII character set and thus determine the first character.

Suppose that the first character was an *F*. The memory layout of Fig. 5-20(c) allowed the intruder to test strings of the form *FA*, *FB*, and so on. Using this approach it took at most $128n$ tries to guess an n character ASCII password, in- stead of 128^n.

Our last flaw concerns OS/360. The description that follows is slightly simplified but preserves the essence of the flaw. In this system it was possible to start up a tape read and then continue computing while the tape drive was transferring data to the user space. The trick here was to carefully start up a tape read and then do a system call that required a user data structure, for example, a file to read and its password.

The operating system first verified that the password was indeed the correct one for the given file. Then it went back and read the file name again for the actual access (it could have saved the name internally, but it did not). Unfortunately, just before the system went to fetch the file name the second time, the file name was overwritten by the tape drive. The system then read the new tile, for which no password had been presented. Getting the timing right took some practice, but it was not that hard. Besides, if there is one thing that computers are good at, it is repeating the same operation over and over ad nauseam.

In addition to these examples, many other security problems and attacks have turned up over the years. One that has appeared in many contexts is the **Trojan horse**, in which a seemingly innocent program that is widely distributed also performs some unexpected and undesirable function, such as stealing data and emailing it to some distant site where it can be collected later.

Another security problem in these times of job insecurity is the **logic bomb**. This device is a piece of code written by one of a company's (currently employed) programmers and secretly inserted into the production operating system. As long as the programmer feeds it its daily password, it does nothing. However, if the programmer is suddenly fired and physically removed from the premises without warning, the next day the logic bomb does not get its password, so it goes off.

Going off might involve clearing the disk, erasing files at random, carefully making hard-to-detect changes to key programs, or encrypting essential files. In the latter case, the company has a tough choice about whether to call the police (which may or may not result in a conviction many months later) or to give in to this blackmail and to rehire the ex-programmer as a "consultant" for an astronomical sum to fix the problem (and hope that he does not plant new logic bombs while doing so).

Probably the greatest computer security violation of all time began in the evening of Nov. 2, 1988 when a Cornell graduate student, Robert Tappan Morris, released a worm program into the Internet that eventually brought down thousands of machines all over the world.

The worm consisted of two programs, the bootstrap and the worm proper. The bootstrap was 99 lines of C called *l1.c*. It was compiled and executed on the system under attack. Once running, it connected to the machine from which it came, uploaded the main worm, and executed it. After going to some trouble to hide its existence, the worm then looked through its new host's routing tables to see what machines that host was connected to and attempted to spread the bootstrap to those machines.

Once established on a machine, the worm tried to break user passwords. Morris did

not have to do much research on how to accomplish this. All he had to do was ask his father, a security expert at the National Security Agency, the U.S. government's top-secret code breaking agency, for a reprint of a classic paper on the subject that Morris Sr. and Ken Thompson had written a decade earlier at Bell Labs (Morris and Thompson, 1979). Each broken password allowed the worm to log in on any machines the password's owner had accounts on.

Morris was caught when one of his friends spoke with the New York Times computer reporter, John Markoff, and tried to convince Markoff that the incident was an accident, the worm was harmless, and the author was sorry. The next day the story was the lead on page one, even upstaging the presidential election three days later. Morris was tried and convicted in federal court. He was sentenced to a 10,000 dollar fine, 3 years probation, and 400 hours of community service. His legal costs probably exceeded 150,000 dollars.

This sentence generated a great deal of controversy. Many in the computer community felt that he was a bright graduate student whose harmless prank had gotten out of control. Nothing in the worm suggested that Morris was trying to steal or damage anything. Others felt he was a serious criminal and should have gone to jail.

One permanent effect of this incident was the establishment of **CERT (Computer Emergency Response Team)**, which provides a central place to report break-in attempts, and a group of experts to analyze security problems and design fixes. While this action was certainly a step forward, it also has its downside. CERT collects information about system flaws that can be attacked and how to fix them. Of necessity, it circulates this information widely to thousands of system administrators on the Internet, which means that the bad guys may also be able to get it and exploit the loopholes in the hours (or even days) before they are closed.

5.4.3 Generic Security Attacks

The flaws described above have been fixed but the average operating system still leaks like a sieve. The usual way to test a system's security is to hire a group of experts, known as tiger teams or penetration teams, to see if they can break in. Hebbard et al. (1980) tried the same thing with graduate students. In the course of the years, these penetration teams have discovered a number of areas in which systems are likely to be weak. Below we have listed some of the more common attacks that are often successful. When designing a system, be sure it can withstand attacks like these.

1. Request memory pages, disk space, or tapes and just read them. Many systems do not erase them before allocating them, and they may be full of interesting information written by the previous owner.

2. Try illegal system calls, or legal system calls with illegal parameters, or even legal system calls with legal but unreasonable parameters. Many systems can easily be confused.

3. Start logging in and then hit DEL, RUBOUT or BREAK halfway through the login sequence. In some systems, the password checking program will be killed and the login considered successful.

4. Try modifying complex operating system structures kept in user space (if any). In some systems (especially on mainframes), to open a file, the program builds a large data structure containing the file name and many other parameters and passes it

to the system. As the file is read and written, the system sometimes updates the structure itself. Changing these fields can wreak havoc with the security.

5. Spoof the user by writing a program that types "login:" on the screen and go away. Many users will walk up to the terminal and willingly tell it their login name and password, which the program carefully records for its evil master.

6. Look for manuals that say "Do not do X." Try as many variations of X as possible.

7. Convince a system programmer to change the system to skip certain vital security checks for any user with your login name. This attack is known as a **trapdoor**.

8. All else failing, the penetrator might find the computer center director's secretary and offer a large bribe. The secretary probably has easy access to all kinds of wonderful information, and is usually poorly paid. Do not underestimate problems caused by personnel.

These and other attacks are discussed by Linde (1975).

Viruses

A special category of attack is the computer virus, which has become a major problem for many computer users. A **virus** is a program fragment that is attached to a legitimate program with the intention of infecting other programs. It differs from a worm only in that a virus piggybacks on an existing program, whereas a worm is a complete program in itself. Viruses and worms both attempt to spread themselves and both can do severe damage.

A typical virus works as follows. The person writing the virus first produces a useful new program, often a game for MS-DOS. This program contains the virus code hidden away in it. The game is then uploaded to a public bulletin board system or offered for free or for a modest price on floppy disk. The program is then advertised, and people begin downloading and using it. Constructing a virus is not easy, so the people doing this are invariably quite bright, and the quality of the game or other program is often excellent.

When the program is started up, it immediately begins examining all the binary programs on the hard disk to see if they are already infected. When an uninfected program is found, it is infected by attaching the virus code to the end of the file, and replacing the first instruction with a jump to the virus. When the virus code is finished executing, it executes the instruction that had previously been first and then jumps to the second instruction. In this way, every time an infected program runs, it tries to infect more programs.

In addition to just infecting other programs, a virus can do other things, such as erasing, modifying, or encrypting files. One virus even displayed an extortion note on the screen, telling the user to send 500 dollars in cash to a post office box in Panama or face the permanent loss of his data and damage to the hardware.

It is also possible for a virus to infect the hard disk's boot sector, making it impossible to boot the computer. Such a virus may ask for a password, which the virus' writer may offer to supply in exchange for some small unmarked bills.

Virus problems are easier to prevent than to cure. The safest course is to buy only shrink-wrapped software from respectable stores. Uploading free software from bulletin boards or getting pirated copies on floppy disk is asking for trouble. Commercial antivirus packages exist, but some of these work by just looking for specific known viruses.

A more general approach is to first reformat the hard disk completely, including the boot sector. Next, install all the trusted software and compute a checksum for each file. The algorithm does not matter, as long as it has enough bits (at least 32). Store the list of (file, checksum) pairs in a safe place, either offline on a floppy disk, or online but encrypted. Starting at that point, whenever the system is booted, all the checksums should be recomputed and compared to the secure list of original checksums. Any file whose current checksum differs from the original one is immediately suspect. While this approach does not prevent infection, it at least allows early detection.

Infection can be made more difficult if the directory where binary programs reside is made unwritable for ordinary users. This technique makes it difficult for the virus to modify other binaries. Although it can be used in UNIX, it is not applicable to MS-DOS because the latter's directories cannot be made unwritable at all.

5.4.4 Design Principles for Security

Viruses mostly occur on desktop systems. On larger systems other problems occur and other methods are needed for dealing with them. Saltzer and Schroeder (1975) have identified several general principles that can be used as a guide to designing secure systems. A brief summary of their ideas (based on experience with Multics) is given below.

First, the system design should be public. Assuming that the intruder will not know how the system works serves only to delude the designers.

Second, the default should be no access. Errors in which legitimate access is refused will be reported much faster than errors in which unauthorized access is allowed.

Third, check for current authority. The system should not check for permission, determine that access is permitted, and then squirrel away this information for subsequent use. Many systems check for permission when a file is opened, and not afterward. This means that a user who opens a file, and keeps it open for weeks, will continue to have access, even if the owner has long since changed the file protection.

Fourth, give each process the least privilege possible. If an editor has only the authority to access the file to be edited (specified when the editor is invoked), editors with Trojan horses will not be able to do much damage. This principle implies a fine-grained protection scheme. We will discuss such schemes later in this chapter.

Fifth, the protection mechanism should be simple, uniform, and built into the lowest layers of the system. Trying to retrofit security to an existing insecure system is nearly impossible. Security, like correctness, is not an add-on feature.

Sixth, the scheme chosen must be psychologically acceptable. If users feel that protecting their files is too much work, they just will not do it. Nevertheless, they will complain loudly if something goes wrong. Replies of the form "It is your own fault" will generally not be well received.

5.4.5 User Authentication

Many protection schemes are based on the assumption that the system knows the identity of each user. The problem of identifying users when they log in is called user authentication. Most authentication methods are based on identifying something the user knows, something the user has, or something the user is.

Passwords

The most widely used form of authentication is to require the user to type a password. Password protection is easy to understand and easy to implement. In UNIX it works like this. The login program asks the user to type his name and password. The password is immediately encrypted. The login program then reads the password file, which is a series of ASCII lines, one per user, until it finds the line containing the user's login name. If the (encrypted) password contained in this line matches the encrypted password just computed, the login is permitted, otherwise it is refused.

Password authentication is easy to defeat. One frequently reads about groups of high school, or even junior high school students who, with the aid of their trusty home computers, have just broken into some top secret system owned by a giant corporation or government agency. Virtually all the time the break-in consists of guessing a user name and password combination.

Although more recent, studies have been made (e.g., Klein, l990) the classic work on password security remains the one done by Morris and Thompson (1979) on UNIX systems. They compiled a list of likely passwords: first and last names, street names, city names, words from a moderate-sized dictionary (also words spelled backward), license plate numbers, and short strings of random characters.

They then encrypted each of these using the known password encryption algorithm and checked to see if any of the encrypted passwords matched entries in their list. Over 86 percent of all passwords turned up in their list.

If all passwords consisted of 7 characters chosen at random from the 95 printable ASCII characters, the search space becomes 95^7, which is about 7×10^3. At 1000 encryptions per second, it would take 2000 years to build the list to check the password file against. Furthermore, the list would fill 20 million magnetic tapes. Even requiring passwords to contain at least one lowercase character, one uppercase character, and one special character, and be at least seven or eight characters long would be a major improvement over unrestricted user-chosen passwords.

Even if it is considered politically impossible to require users to pick reasonable passwords, Morris and Thompson have described a technique that renders their own attack (encrypting a large number of passwords in advance) almost useless. Their idea is to associate an n-bit random number with each password. The random number is changed whenever the password is changed. The random number is stored in the password file in unencrypted form, so that everyone can read it. Instead of just storing the encrypted password in the password file, the password and the random number are first concatenated and then encrypted together. This encrypted result is stored in the password file.

Now consider the implications for an intruder who wants to build up a list of likely passwords, encrypt them, and save the results in a sorted file, *f*, so that any encrypted password can be looked up easily. If an intruder suspects that *Marilyn* might be a password, it is no longer sufficient just to encrypt *Marilyn* and put the result in *f*. He has to encrypt 2^n strings, such as *Marilyn0000*, *Marilyn0001*, *Marilyn0002*, and so forth and enter all of them in *f*. This technique increases the size off by 2^n. UNIX uses this method with $n = 12$. It is known as **salting** the password file. Some versions of UNIX make the password file itself unreadable but provide a program to look up entries upon request, adding just enough delay to greatly slow down any attacker.

Although this method offers protection against intruders who try to precompute a large list of encrypted passwords, it does little to protect a user *David* whose password is also *David*. One way to encourage people to pick better passwords is to have the

computer offer advice. Some computers have a program that generates random easy-to-pronounce nonsense words, such as *fatally*, *garbungy*, or *bipirty* that can be used as passwords (preferably with some upper case and special characters thrown in).

Other computers require users to change their passwords regularly, to limit the damage done if a password leaks out. The most extreme form of this approach is the **one-time password**. When one-time passwords are used, the user gets a book containing a list of passwords. Each login uses the next password in the list. If an intruder ever discovers a password, it will not do him any good, since next time a different password must be used. It is suggested that the user try to avoid losing the password book.

It goes almost without saying that while a password is being typed in, the computer should not display the typed characters, to keep them from prying eyes hear the terminal. What is less obvious is that passwords should never be stored in the computer in unencrypted form. Furthermore, not even the computer center management should have unencrypted copies. Keeping unencrypted passwords anywhere is looking for trouble.

A variation on the password idea is to have each new user provide a long list of questions and answers that are then stored in the computer in encrypted form. The questions should be chosen so that the user does not need to write them down. Typical questions are:

1. Who is Marjolein's sister?

2. On what street was your elementary school?

3. What did Mrs. Woroboff teach?

At login, the computer asks one of them at random and checks the answer.

Another variation is **challenge-response**. When this is used, the user picks an algorithm when signing up as a user, for example x^2. When the user logs in, the computer types an argument, say 7, in which case the user types 49. The algorithm can be different in the morning and afternoon, on different days of the week, from different terminals, and so on.

Physical Identification

A completely different approach to authorization is to check to see if the user has some item, normally a plastic card with a magnetic stripe on it. The card is inserted into the terminal, which then checks to see whose card it is. This method can be combined with a password, so a user can only log in if he (1) has the card and (2) knows the password. Automated cash-dispensing machines usually work this way.

Yet another approach is to measure physical characteristics that are hard to forge. For example, a fingerprint or a voiceprint reader in the terminal could verify the user's identity. (lt makes the search go faster if the user tells the computer who he is, rather than making the computer compare the given fingerprint to the entire data base.) Direct visual recognition is not yet feasible but may be one day.

Another technique is signature analysis. The user signs his name with a special pen connected to the terminal, and the computer compares it to a known specimen stored on line. Even better is not to compare the signature, but compare the pen motions made while writing it. A good forger may be able to copy the signature, but will not have a clue as to the exact order in which the strokes were made.

Finger length analysis is surprisingly practical. When this is used, each terminal has a device like the one of Fig. 5-21. The user inserts his hand into it, and the length of all his fingers is measured and checked against the data base.

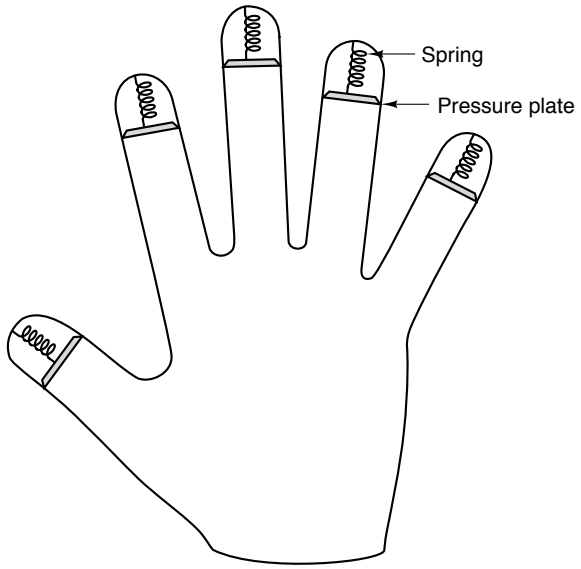


Figure 5-21. A device for measuring finger length.

We could go on and on with more examples, but two more will help make an important point. Cats and other animals mark off their territory by urinating around its perimeter. Apparently cats can identify each other this way. Suppose that someone comes up with a tiny device capable of doing an instant urinalysis, thereby providing a foolproof identification. Each terminal could be equipped with one of these devices, along with a discreet sign reading: "For login, please deposit sample here." This might be an absolutely unbreakable system, but it would probably have a fairly serious user acceptance problem.

The same could be said of a system consisting of a thumbtack and a small spectrograph. The' user would be requested to press his thumb against the thumbtack, thus extracting a drop of blood for spectrographic analysis. The point is that any authentication scheme must be psychologically acceptable to the user community. Finger-length measurements probably will not cause any problem, but even something as nonintrusive as storing fingerprints on line may be unacceptable to many people.

Countermeasures

Computer installations that are really serious about security, something that frequently happens the day after an intruder has broken in and done major damage, often take steps to make unauthorized entry much harder. For example, each user could be allowed to log in only from a specific terminal, and only during certain days of the week and hours of the day.

Dial-up telephone lines could be made to work as follows. Anyone can dial up and log in, but after a successful login, the system immediately breaks the connection and calls the user back at an agreed upon number. This measure means than an intruder cannot just try breaking in from any phone line; only the user's (home) phone will do. In any event, with or without call back, the system should take at least 10 seconds to check any password typed in on a dial-up line, and should increase this time after several consecutive unsuccessful login attempts, in order to reduce the rate at which intruders can try. After

three failed login attempts, the line should be disconnected for 10 minutes and security personnel notified.

All logins should be recorded. When a user logs in, the system should report the time and terminal of the previous login, so he can detect possible break ins.

The next step up is laying baited traps to catch intruders. A simple scheme is to have one special login name with an easy password (e.g., login name: guest, password: guest). Whenever anyone logs in using this name, the system security specialists are immediately notified. Other traps can be easy-to-find bugs in the operating system and similar things, designed for the purpose of catching intruders in the act. Stoll (1989) has written an entertaining account of the traps he set to track down a spy who broke into a university computer in search of military secrets.

5.5 PROTECTION MECHANISMS

In the previous sections we have looked at many potential problems, some of them technical and some of them not. In the following sections we will concentrate on some of the detailed technical ways that are used in operating systems to protect files and other things. All of these techniques make a clear distinction between policy (whose data are to be protected from whom) and mechanism (how the system enforces the policy). The separation of policy and mechanism is discussed in (Levin et al., 1975). Our emphasis will be on the mechanism, not the policy. For more advanced material, see (Sandhu, 1993).

In some systems, protection is enforced by a program called a **reference monitor**. Every time an access to a potentially protected resource is attempted, the system first asks the reference monitor to check its legality. The reference monitor then looks at its policy tables and makes a decision. Below we will describe the environment in which a reference monitor operates.

5.5.1 Protection Domains

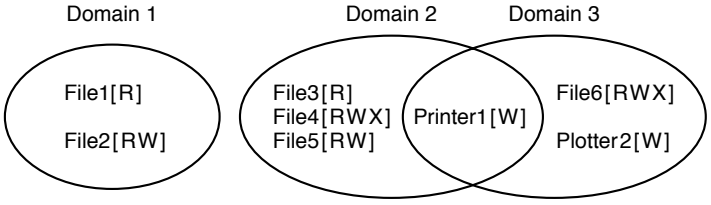


Figure 5-22. Three Protection domains.

Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
	Object							
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Figure 5-23. A Protection matrix.

Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
	Object										
1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

Figure 5-24. A Protection matrix with domains as objects.

5.5.2 Access Control Lists

5.5.3 Capabilities

5.5.4 Covert Channels

5.6 OVERVIEW OF THE MINIX FILE SYSTEM

Like any file system, the MINIX file system must deal with all the issues we have just studied. It must allocate and deallocate space for files, keep track of disk blocks and free space, provide some way to protect files against unauthorized usage, and so on. In the remainder of this chapter we will look closely at MINIX to see how it accomplishes these goals.

In the first part of this chapter, we have repeatedly referred to UNIX rather than to MINIX for the sake of generality, although the external interfaces of the two is virtually identical. Now we will concentrate on the internal design of MINIX. For information about the UNIX internals, see Thompson (1978), Bach (1987), Lions (1996), and Vahalia (1996).

The MINIX file system is just a big C program that runs in user space (see Fig. 2-26). To read and write files, user processes send messages to the file system telling what they want done. The file system does the work and then sends back a reply. The file system is, in fact, a network file server that happens to be running on the same machine as the caller.

This design has some important implications. For one thing, the file system can be modified, experimented with, and tested almost completely independently of the rest of MINIX. For another, it is very easy to move the whole file system to any computer that has a C compiler, compile it there, and use it as a free-standing UNIX-like remote file server. The only changes that need to be made are in the area of how messages are sent

Figure 5-25. The capability list for domain 2 in Fig. 5-23.

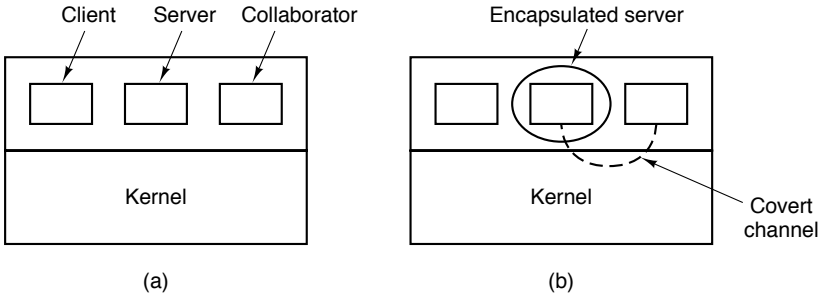


Figure 5-26. (a) The client, server, and collaborator processes. (b) The encapsulated server can still leak to the collaborator via convert channels.

and received, which differs from system to system.

In the following sections, we will present an overview of many of the key areas of the file system design. Specifically, we will look at messages, the file system layout, the bitmaps, i-nodes, the block cache, directories and paths, file descriptors, file locking, and special files (plus pipes). After studying these topics, we will show a simple example of how the pieces fit together by tracing what happens when a user process executes the READ system call.

5.6.1 Messages

The file system accepts 39 types of messages requesting work. All but two are for MINIX system calls. The two exceptions are messages generated by other parts of MINIX. Of the system calls, 31 are accepted from user processes. Six system call messages are for system calls which are handled first by the process manager, which then calls the file system to do a part of the work. Two other messages are also handled by the file system. The messages are shown in Fig. 5-27.

Figure 5-27. File system messages. File name parameters are always pointers to the name. The code status as reply value means OK or ERROR.

The structure of the file system is basically the same as that of the process manager and all the I/O device drivers. It has a main loop that waits for a message to arrive. When a message arrives, its type is extracted and used as an index into a table containing pointers to the procedures within the file system that handle all the types. Then the appropriate procedure is called, it does its work and returns a status value. The file system then sends a reply back to the caller and goes back to the top of the loop to wait for the next message.

5.6.2 File System Layout

A MINIX file system is a logical, self-contained entity with i-nodes, directories, and data blocks. It can be stored on any block device, such as a floppy disk or a (portion of a) hard disk. In all cases, the layout of the file system has the same structure. Fig. 5-28 shows this layout for a 360K floppy disk with 128 i-nodes and a 1K block size. Larger

file systems, or those with more or fewer i-nodes or a different block size, will have the same six components in the same order, but their relative sizes may be different.

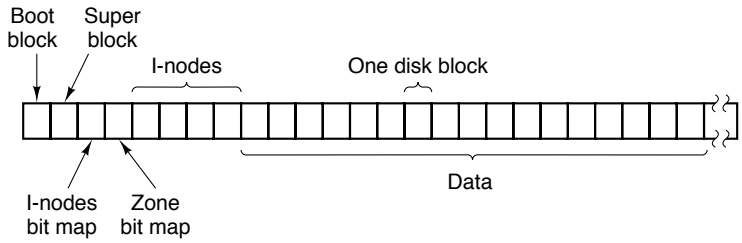


Figure 5-28. Disk layout for the simplest disk: a 360K floppy disk, with 128 i-nodes and a 1K block size (i.e., two consecutive 512-byte sectors are treated as a single block).

Each file system begins with a **boot block**. This contains executable code. When the computer is turned on, the hardware reads the boot block from the boot device into memory, jumps to it, and begins executing its code. The boot block code begins the process of loading the operating system itself. Once the system has been booted, the boot block is not used any more. Not every disk drive can be used as a boot device, but to keep the structure uniform, every block device has a block reserved for boot block code. At worst this strategy wastes one block. To prevent the hardware from trying to boot an unbootable device, a **magic number** is placed at a known location in the boot block when and only when the executable code is written to the device. When booting from a device, the hardware (actually, the BIOS code) will refuse to attempt to load from a device lacking the magic number. Doing this prevents inadvertently using garbage as a boot program.

The super-block contains information describing the layout of the file system. It is illustrated in Fig. 5-29. The main function of the super-block is to tell the file system how big the various pieces of the file system are. Given the block size and the number of i-nodes, it is easy to calculate the size of the i-node bitmap and the number of blocks of inodes. For example, for a 1K block, each block of the bitmap has 1K bytes (8K bits), and thus can keep track of the status of up to 8192 i-nodes. (Actually the first block can handle only up to 8191 i-nodes, since there is no 0th i-node, but it is given a bit in the bitmap, anyway). For 10,000 i-nodes, two bitmap blocks are needed. Since i-nodes each occupy 64 bytes, a 1K block holds up to 16 i-nodes. With 128 i-nodes, 8 disk blocks are needed to contain them all.

We will explain the difference between zones and blocks in detail later, but for the time being it is sufficient to say that disk storage can be allocated in units (zones) of 1, 2, 4, 8, or in general 2^n blocks. The zone bitmap keeps track of free storage in zones, not blocks. For all standard floppy disks used by MINIX the zone and block sizes are the same (1K), so for a first approximation a zone is the same as a block on these devices. Until we come to the details of storage allocation later in the chapter, it is adequate to think "block" whenever you see "zone."

Note that the number of blocks per zone is not stored in the super-block, as it is never needed. All that is needed is the base 2 logarithm of the zone to block ratio, which is used as the shift count to convert zones to blocks and vice versa. For example, with 8 blocks per zone, $\log_2 8 = 3$, so to find the zone containing block 128 we shift 128 right 3 bits to get zone 16.

The zone bitmap includes only the data zones (i.e., the blocks used for the bitmaps

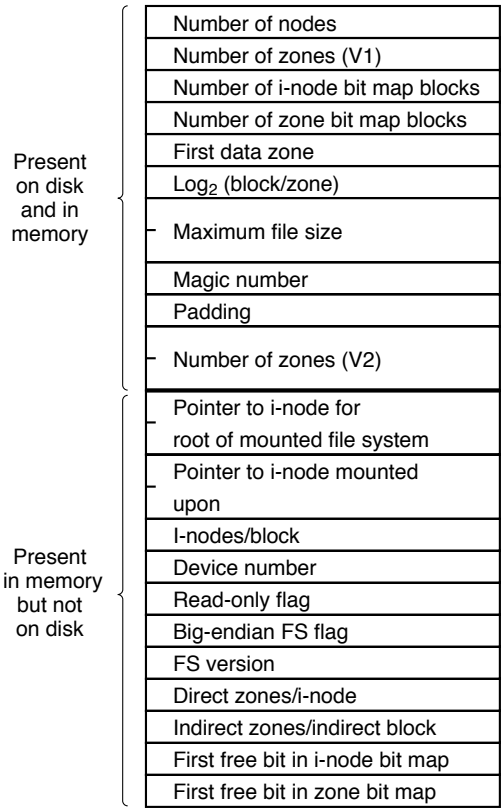


Figure 5-29. The MINIX super-block.

and i-nodes are not in the map), with the first data zone designated zone 1 in the bitmap. As with the i-node bitmap, bit 0 in the map is unused, so the first block in the zone bitmap can map 8191 zones and subsequent blocks can map 8192 zones each. If you examine the bitmaps on a newly formatted disk, you will find that both the i-node and zone bitmaps have 2 bits set to 1. One is for the nonexistent 0th i-node or zone; the other is for the i-node and zone used by the root directory on the device, which is placed there when the file system is created.

The information in the super-block is redundant because sometimes it is needed in one form and sometimes in another. With 1K devoted to the super-block, it makes sense to compute this information in all the forms it is needed, rather than having to recompute it frequently during execution. The zone number of the first data zone on the disk, for example, can be calculated from the block size, zone size, number of i-nodes, and number of zones, but it is faster just to keep it in the super-block. The rest of the super-block is wasted anyhow, so using up another word of it costs nothing.

When MINIX is booted, the super-block for the root device is read into a table in memory. Similarly, as other file systems are mounted, their super-blocks are also brought into memory. The super-block table holds a number of fields not present on the disk. These include flags that allow a device to be specified as read-only or as following a byte-order convention opposite to the standard, and fields to speed access by indicating points in the bitmaps below which all bits are marked used. In addition, there is a field describing the device from which the super-block came.

Before a disk can be used as a MINIX file system, it must be given the structure of Fig. 5-28. The utility program *mkfs* has been provided to build file systems. This program can be called either by a command like

```
mkfs /dev/fd1 1440
```

to build an empty 1440 block file system on the floppy disk in drive 1, or it can be given a prototype file listing directories and files to include in the new file system. This command also puts a magic number in the super-block to identify the file system as a valid MINIX file system. The MINIX file system has evolved, and some aspects of the file system (for instance, the size of i-nodes) were different in earlier versions. The magic number identifies the version of *mkfs* that created the file system, so differences can be accommodated. Attempts to mount a file system not in MINIX format, such as an MS-DOS diskette, will be rejected by the MOUNT system call, which checks the super-block for a valid magic number and other things.

5.6.3 Bit Maps

5.6.4 I-nodes

5.6.5 The Block Cache

5.6.6 Directories and Paths

5.6.7 File Descriptors

5.6.8 File Locking

5.6.9 Pipes and Special Files

5.6.10 An Example: The READ SYSTEM CALL

5.7 IMPLEMENTATION OF THE MINIX FILE SYSTEM

The MINIX file system is relatively large (more than 100 pages of C) but quite straightforward. Requests to carry out system calls come in, are carried out, and replies are sent. In the following sections we will go through it a file at a time, pointing out the highlights. The code itself contains many comments to aid the reader.

In looking at the code for other parts of MINIX we have generally looked at the main loop of a process first and then looked at the routines that handle the different message types. We will organize our approach to the file system differently. First we will go through the major subsystems (cache management, i-node management, etc.). Then we will look at the main loop and the system calls that operate upon files. Next we will look at systems call that operate upon directories. Finally we will see how device special files are handled.

5.7.1 Header Files and Global Data Structures

5.7.2 Table Management

5.7.3 The Main Program

5.7.4 Operations on Individual Files

5.7.5 Directories and Paths

5.7.6 Other System Calls

5.7.7 The I/O Device Interface

5.7.8 General Utilities

5.8 SUMMARY

When seen from the outside, a file system is a collection of files and directories, plus operations on them. Files can be read and written, directories can be created and destroyed, and files can be moved from directory to directory. Most modern file systems support a hierarchical directory system, in which directories may have subdirectories ad infinitum.

When seen from the inside, a file system looks quite different. The file system-designers have to be concerned with how storage is allocated, and how the system keeps track of which block goes with which file. We have also seen how different systems have different directory structures. File system reliability and performance are also important issues.

Security and protection are of vital concern to both the system users and system designers. We discussed some security flaws in older systems, and generic problems that many systems have. We also looked at authentication, with and without passwords, access control lists, and capabilities, as well as a matrix model for thinking about protection.

Finally, we studied the MINIX file system in detail. It is large but not very complicated. It accepts requests for work from user processes, indexes into a table of procedure pointers, and calls that procedure to carry out the requested system call. Due to its modular structure and position outside the kernel, it can be removed from MINIX and used as a free-standing network file server with only minor modifications.

Internally, MINIX buffers data in a block cache and attempts to read ahead when making sequential access to file. If the cache is made large enough, most program text will be found to be already in memory during operations that repeatedly access a particular set of programs, such as a compilation.

Chapter 6

READING LIST AND BIBLIOGRAPHY

6.1 Suggestions for Further Reading

6.1.1 Introduction and General Works

6.1.2 Processes

6.1.3 Input/Output

6.1.4 Memory Management

6.1.5 File Systems

6.2 Alphabetical Bibliography