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 the equivalent 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. 4-24(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.

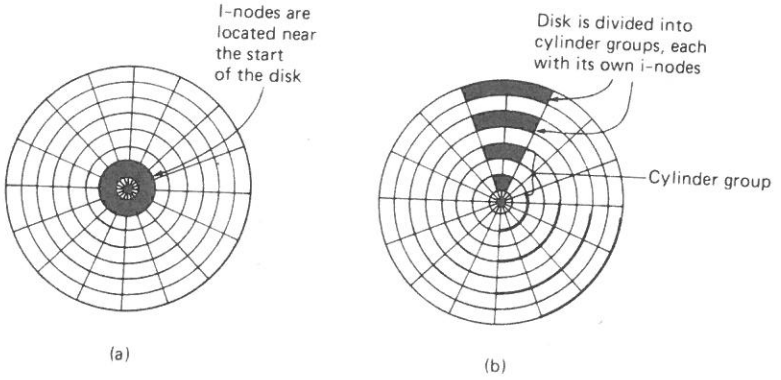

Fig. 4-24. (a) I-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

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. 4-24(b), is to divide the disk into cylinder groups, each with its own i-nodes, blocks, and free list (McKusick et al., 1984). When creating a new file, any i-node can be chosen, but having done this, 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 cylinder group close by is used.

4.4. SECURITY

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.

4.4.1. 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.

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 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 the KGB 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.

4.4.2. 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 four different operating systems: UNIX, MULTICS, 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 file. 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 file. In 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 file. By putting the necessary commands in a shell script, they could be tried over and over until the trick worked.

The MULTICS security problem had to do with the fact that the system designers always perceived MULTICS as a timesharing system, with batch facilities thrown in as an afterthought to pacify some old batch diehards. The timesharing security was excellent; the batch security was nonexistent. It was possible for anyone to submit a batch job that read a deck of cards into an arbitrary user's directory.

To steal someone's files, all one had to do was get a copy of the editor source code, modify it to steal files (but still work perfectly as an editor), compile it, and put it into the victim's *bin* directory. The next time the victim called the editor, he got the intruder's version, which edited fine, but stole all his files as well. The idea of modifying a normal program to do nasty things in addition to its usual function and arranging for the victim to use the modified version is now known as the **Trojan horse attack**.

The TENEX operating system used to be very popular on the DEC-10 computers. It is no longer used much, 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 system to call a user function on each page fault.

TENEX also used passwords to protect files. To access a file, a program had to present the proper password. The operating system checked passwords one character 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. 4-25(a), with the first character at the end of one page, and the rest at the start of the next page.

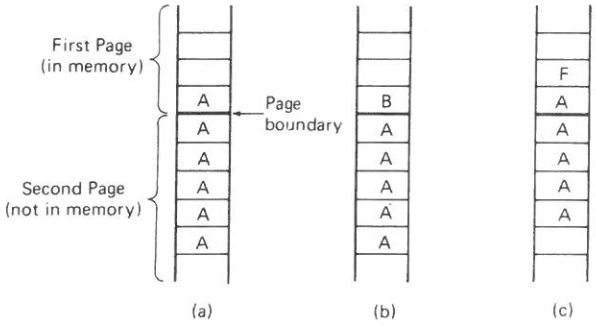

Fig. 4-25. The TENEX password problem.

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. 4-25(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 the first character was an *F*. The memory layout of Fig. 4-25(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, instead 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 file, 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*.

4.4.3. The Internet Worm

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. This action brought down thousands of computers at universities, corporations, and government laboratories all over the world before it was tracked down and removed. It also started a controversy that has not yet died down. We will discuss the highlights of this event below. For more technical information see (Spafford, 1989). For the story viewed as a police thriller, see (Hafner and Markoff, 1991).

The story began sometime in 1988 when Morris discovered two bugs in Berkeley UNIX that made it possible to gain unauthorized access to machines all over the Internet, the vast research network connecting hundreds of thousands of machines in the U.S., Europe, and the Far East. Working alone, he wrote a self replicating program, called a **worm**, that would exploit these errors and replicate itself in seconds on every machine it could gain access to. He worked on the program for months, carefully tuning it and having it try to hide its tracks.

It is not known whether the release on Nov. 2, 1988 was intended as a test, or was the real thing. In any event, it did bring most of the Sun and VAX systems on the Internet to their knees within a few hours of its release. Morris' motivation is

unknown, but it is possible that he intended the whole idea as a high-tech practical joke, but which due to a programming error got completely out of hand.

Technically, 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.

Three methods were tried to infect new machines. Method 1 was to try to run a remote shell using the *rsh* command. Some machines trust other machines, and willingly run *rsh* without any further authentication. If this worked, the remote shell uploaded the worm program and continued infecting new machines from there.

Method 2 made use of a program present on all BSD systems called *finger*, that allows a user anywhere on the Internet to type

```
finger name@site
```

to display information about a person at a particular installation. This information usually includes the person's real name, login, home and work addresses and telephone numbers, secretary's name and telephone number, FAX number, and similar information. It is the electronic equivalent of the phone book.

Finger works as follows. At every BSD site a background process called the **finger daemon** runs all the time fielding and answering queries from all over the Internet. What the worm did was call *finger* with a specially handcrafted 536-byte string as parameter. This long string overflowed the daemon's buffer and overwrote its stack. The bug exploited here is the daemon's failure to check for overflow. When the daemon returned from the procedure it was in at the time it got the request, it returned not to *main*, but to a procedure inside the 536-byte string on the stack. This procedure tried to execute */bin/sh*. If it succeeded, the worm now had a shell running on the machine under attack.

Method 3 depended on a bug in the mail system, *sendmail*, which allowed the worm to mail a copy of the bootstrap and get it executed.

Once established, 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 code breaking agency, for a reprint of a classic paper on the subject that Morris Sr. and Ken Thompson wrote 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.

Every time the worm gained access to a new machine, it checked to see if any other copies of the worm were already active there. If so, the new copy exited, except one time in seven it kept going, possibly in an attempt to keep the worm propagating even if the system administration there started up their own version of the worm to fool the real worm. The use of 1 in 7 created far too many worms, and was the reason all the infected machines ground to a halt: they were infested with worms.

If Morris had left this out and just exited whenever another worm was sighted, the worm would probably have gone undetected.

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 friend inadvertently let slip that the perpetrator's login was *rtm*. Converting *rtm* into the owner's name was easy—all that had to be done was to run *finger*. 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.

4.4.4. 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. In many systems, 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 trick or bribe her. The secretary probably has easy access to all kinds of wonderful information, and is usually poorly paid. Do not underestimate the 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 will supply in exchange for money.

Virus problems are easier to prevent than to cure. The safest course is only to buy 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 16, preferably 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.

4.4.5. 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 in to 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.

4.4.6. 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.

Morris and Thompson (1979) made a study of passwords on UNIX systems. They compiled a list of likely passwords: first names, last names, street names, city names, words from a moderate-sized dictionary (also words spelled backward), valid 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^{13}. 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 characters long would be a big improvement on 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 of *f* 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 *fotally*, *garbungy*, or *bipitty* 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 near the terminal. What is less obvious is that passwords should never be stored in the computer in unencrypted form, and that 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. (It 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. 4-26. The user inserts his hand into it, and the length of all his fingers is measured and checked against the data base.

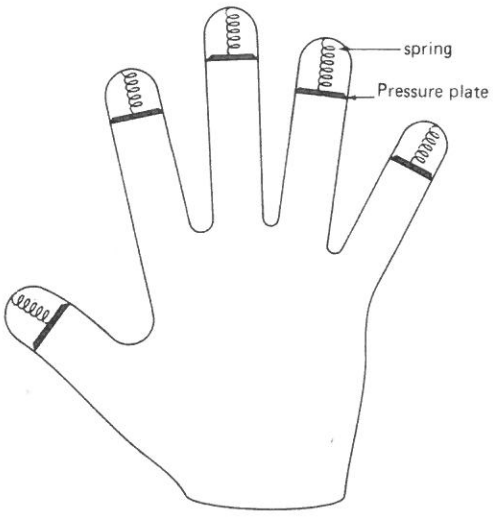

Fig. 4-26. A device for measuring finger length.

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 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 discrete 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.

Dialup 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 dialup 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.

4.5. 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.

4.5.1. Protection Domains

A computer system contains many **objects** that need to be protected. These objects can be hardware, such as CPUs, memory segments, terminals, disk drives, or printers, or they can be software, such as processes, files, data bases, or semaphores.

Each object has a unique name by which it is referenced, and a set of operations that can be carried out on it. READ and WRITE are operations appropriate to a file; UP and DOWN make sense on a semaphore.

It is obvious that a way is needed to prohibit processes from accessing objects that they are not authorized to access. Furthermore, this mechanism must also make it possible to restrict processes to a subset of the legal operations when that is needed. For example, process *A* may be entitled to read, but not write, file *F*.

To provide a way to discuss different protection mechanisms, it is convenient to introduce the concept of a domain. A **domain** is a set of (object, rights) pairs. Each pair specifies an object and some subset of the operations that can be performed on it. A **right** in this context means permission to perform one of the operations.

Figure 4-27 depicts three domains, showing the objects in each domain and the rights [Read, Write, eXecute] available on each object. Note that *Printer1* is in two domains at the same time. Although not shown in this example, it is possible for the same object to be in multiple domains, with *different* rights in each domain.

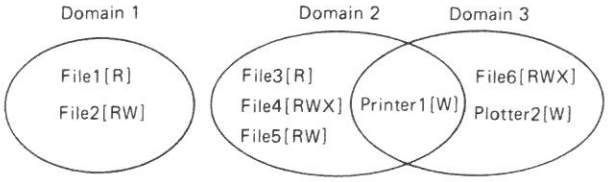

Fig. 4-27. Three protection domains.

At every instant of time, each process runs in some protection domain. In other words, there is some collection of objects it can access, and for each object it has some set of rights. Processes can also switch from domain to domain during execution. The rules for domain switching are highly system dependent.

To make the idea of a protection domain more concrete, let us look at UNIX. In UNIX, the domain of a process is defined by its uid and gid. Given any (uid, gid) combination, it is possible to make a complete list of all objects (files, including I/O devices represented by special files, etc.) that can be accessed, and whether they can be accessed for reading, writing, or executing. Two processes with the same (uid, gid) combination will have access to exactly the same set of objects. Processes with different (uid, gid) values will have access to a different set of files, although there

will be considerable overlap in most cases.

Furthermore, each process in UNIX has two halves: the user part and the kernel part. When the process does a system call, it switches from the user part to the kernel part. The kernel part has access to a different set of objects from the user part. For example, the kernel can access all the pages in physical memory, the entire disk, and all the other protected resources. Thus, a system call causes a domain switch.

When a process does an EXEC on a file with the SETUID or SETGID bit on, it acquires a new effective uid or gid. With a different (uid, gid) combination, it has a different set of files and operations available. Running a program with SETUID or SETGID is also a domain switch.

The division of a UNIX process into a kernel part and a user part is a remnant of a much more powerful domain switching mechanism that was used in MULTICS. In that system, the hardware supported not two domains (kernel and user) per process, but up to 64. A MULTICS process could consist of a collection of procedures, each one running in some domain, which were called **rings** (Schroeder and Saltzer, 1972). Procedures could also be linked dynamically to a running process during execution.

Figure 4-28 shows four rings. The innermost ring, the operating system kernel, had the most power. Moving outward from the kernel, the rings became successively less powerful. Ring 1, for example, might contain the code for functions that in UNIX are handled by SETUID programs owned by the root, such as *mkdir*. Ring 2 might contain the grading program used to evaluate student programs, and ring 3 might contain the student programs.

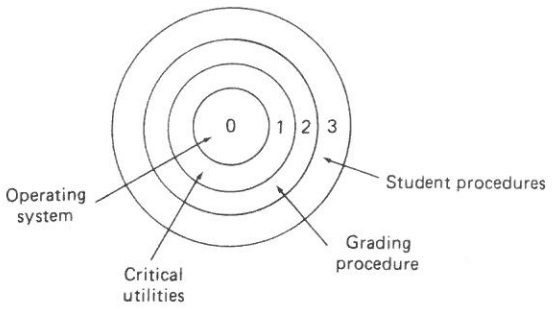

Fig. 4-28. A process in MULTICS occupying four rings. Each ring is a separate protection domain.

When a procedure in one ring called a procedure in another ring, a trap occurred, giving the system the opportunity to change the protection domain of the process. Thus, a MULTICS process could operate in as many as 64 different domains during its lifetime. (Actually, the situation was more complicated than we have sketched above; procedures could live in multiple consecutive rings, and parameter passing between rings was carefully controlled.) For a detailed description of MULTICS, see Organick (1972).

An important question is how the system keeps track of which object belongs to which domain. Conceptually, at least, one can envision a large matrix, with the rows

being the domains and the columns being the objects. Each box lists the rights, if any, that the domain contains for the object. The matrix for Fig. 4-27 is shown in Fig. 4-29. Given this matrix and the current domain number, the system can tell if an access to a given object in a particular way from a specified domain is allowed.

	Object							
Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Fig. 4-29. A protection matrix.

Domain switching itself, as in MULTICS, can be easily included in the matrix model by realizing that a domain is itself an object, with the operation ENTER. Figure 4-30 shows the matrix of Fig. 4-29 again, only now with the three domains as objects themselves. Processes in domain 1 can switch to domain 2, but once there, they cannot go back. This situation models executing a SETUID program in UNIX. No other domain switches are permitted in this example.

	Object										
Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain 3
1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

Fig. 4-30. A protection matrix with domains as objects.

4.5.2. Access Control Lists

In practice, actually storing the matrix of Fig. 4-30 is rarely done because it is large and sparse. Most domains have no access at all to most objects, so storing a big, empty matrix is a waste of disk space. Two methods that are practical, however, are storing the matrix by rows or by columns, and then storing only the nonempty elements. The two approaches are surprisingly different. In this section we will look at storing it by column; in the next one we will study storing it by row.

The first technique consists of associating with each object an (ordered) list containing all the domains that may access the object, and how. This list is called the

access control list or **ACL**. If it were to be implemented in UNIX, the easiest way would be to put the ACL for each file in a separate disk block, and include the block number in the file's i-node. As only the nonempty entries of the matrix are stored, the total storage required for all the ACLs combined is much less than what would be needed for the whole matrix.

As an example of how ACLs work, let us continue to imagine that they were used in UNIX, where a domain is specified by a (uid, gid) pair. Actually, ACLs were used in UNIX's role model, MULTICS, more or less in the way we will describe, so the example is not so hypothetical.

Let us now assume that we have four users (i.e., uids) *Jan, Els, Jelle*, and *Maaike*, who belong to groups *system, staff, student*, and *student*, respectively. Suppose some files have the following ACLs:

File0: (Jan, *, RWX)
File1: (Jan, system, RWX)
File2: (Jan, *, RW–), (Els, staff, R––), (Maaike, *, RW–)
File3: (*, student, R––)
File4: (Jelle, *, –––), (*, student, R––)

Each ACL entry, in parentheses, specifies a uid, a gid, and the allowed accesses (Read, Write, eXecute). An asterisk means all uids or gids. *File0* can be read, written, or executed by any process with uid = *Jan*, and any gid. *File1* can be accessed only by processes with uid = *Jan* and gid = *system*. A process that has uid = *Jan* and gid = *staff* can access *File0* but not *File1*. *File2* can be read or written by processes with uid = *Jan* and any gid, read by processes with uid = *Els* and gid = *staff*, or by processes with uid = *Maaike* and any gid. *File3* can be read by any student. *File4* is especially interesting. It says that anyone with uid = *Jelle*, in any group, has no access at all, but all other students can read it. By using ACLs it is possible to prohibit specific uids or gids from accessing an object, while allowing everyone else in the same class.

So much for what UNIX does not do. Now let us look at what it *does* do. It provides three bits, *rwx*, per file for the owner, the owner's group, and others. This scheme is just the ACL again, but compressed to 9 bits. It is a list associated with the object saying who may access it and how. While the 9-bit UNIX scheme is clearly less general than a full-blown ACL system, in practice it is adequate, and its implementation is much simpler and cheaper.

The owner of an object can change its ACL at any time, thus making it easy to prohibit accesses that were previously allowed. The only problem is that changing the ACL will probably not affect any users who are currently using the object (e.g., have the file open).

4.5.3. Capabilities

The other way of slicing up the matrix of Fig. 4-30 is by rows. When this method is used, associated with each process is a list of objects that may be accessed, along with an indication of which operations are permitted on each, in other words, its

domain. This list is called a **capability list**, and the individual items on it are called **capabilities** (Dennis and Van Horn, 1966; Fabry, 1974).

A typical capability list is shown in Fig. 4-31. Each capability has a *Type* field, which tells what kind of an object it is, a *Rights* field, which is a bit map indicating which of the legal operations on this type of object are permitted, and an *Object* field, which is a pointer to the object itself (e.g., its i-node number). Capability lists are themselves objects, and may be pointed to from other capability lists, thus facilitating sharing of subdomains. Capabilities are often referred to by their position in the capability list. A process might say: "Read 1K from the file pointed to by capability 2." This form of addressing is similar to using file descriptors in UNIX.

	Type	Rights	Object
0	File	R– –	Pointer to File3
1	File	RWX	Pointer to File4
2	File	RW–	Pointer to File5
3	Printer	–W–	Pointer to Printer1

Fig. 4-31. The capability list for domain 2 in Fig. 4-29.

It is fairly obvious that capability lists, or **C-lists** as they are often called, must be protected from user tampering. Three methods have been proposed to protect them. The first way requires a **tagged architecture**, a hardware design in which each memory word has an extra (or tag) bit that tells whether the word contains a capability or not. The tag bit is not used by arithmetic, comparison, or similar ordinary instructions, and it can be modified only by programs running in kernel mode (i.e., the operating system).

The second way is to keep the C-list inside the operating system, and just have processes refer to capabilities by their slot number, as mentioned above. Hydra (Wulf, 1974) worked this way.

The third way is to keep the C-list in user space, but encrypt each capability with a secret key unknown to the user. This approach is particularly suited to distributed systems, and is used extensively by Amoeba, which will be described in Chap. 14.

In addition to the specific object-dependent rights, such as read and execute, capabilities usually have **generic rights** which are applicable to all objects. Examples of generic rights are

1. Copy capability: create a new capability for the same object.
2. Copy object: create a duplicate object with a new capability.
3. Remove capability: delete an entry from the C-list; object unaffected.
4. Destroy object: permanently remove an object and a capability.

Many capability systems are organized as a collection of modules, with **type manager modules** for each type of object. Requests to perform operations on a file are sent to the file manager, whereas requests to do something with a mailbox go to

the mailbox manager. These requests are accompanied by the relevant capability. A problem arises here, because the type manager module is just an ordinary program, after all. The owner of a file capability can perform only some of the operations on the file, but cannot get at its internal representation (e.g., its i-node). It is essential that the type manager module be able to do more with the capability than an ordinary process.

This problem was solved in Hydra by a technique called **rights amplification**, in which type managers were given a rights template that gave them more rights to an object than the capability itself allowed. Other capability systems that have strong typing of objects also need something like this.

A last remark worth making about capability systems is that revoking access to an object is quite difficult. It is hard for the system to find all the outstanding capabilities for any object to take them back, since they may be stored in C-lists all over the disk. One approach is to have each capability point to an indirect object, rather than to the object itself. By having the indirect object point to the real object, the system can always break that connection, thus invalidating the capabilities. (When a capability to the indirect object is later presented to the system, the user will discover that the indirect object is now pointing to a null object.)

Another way to achieve revocation is the scheme used in Amoeba. Each object contains a long random number, which is also present in the capability. When a capability is presented for use, the two are compared. Only if they agree is the operation allowed. The owner of an object can request that the random number in the object be changed, thus invalidating existing capabilities. Neither scheme allows selective revocation, that is, taking back, say, John's permission, but nobody else's.

4.5.4. Protection Models

Protection matrices, such as that of Fig. 4-29, are not static. They frequently change as new objects are created, old objects are destroyed, and owners decide to increase or restrict the set of users for their objects. A considerable amount of attention has been paid to modeling protection systems in which the protection matrix is constantly changing. In the remainder of this section, we will touch briefly upon some of this work.

Harrison et al. (1976) identified six primitive operations on the protection matrix that can be used as a base to model any protection system. These operations are: CREATE OBJECT, DELETE OBJECT, CREATE DOMAIN, DELETE DOMAIN, INSERT RIGHT, and REMOVE RIGHT. The two latter primitives insert and remove rights from specific matrix elements, such as granting domain 1 permission to read *File6*.

These six primitives can be combined into **protection commands**. It is these protection commands that user programs can execute to change the matrix. They may not execute the primitives directly. For example, the system might have a command to create a new file, which would test to see if the file already existed, and if not, create a new object and give the owner all rights to it. There might also be a command to allow the owner to grant permission to read the file to everyone in the system, in effect, inserting the "read" right in the new file's entry in every domain.

At any instant, the matrix determines what a process in any domain can do, not what it is authorized to do. The matrix is what is enforced by the system; authorization has to do with management policy. As an example of this distinction, let us consider the simple system of Fig. 4-32 in which domains correspond to users (similar to the UNIX model). In Fig. 4-32(a) we see the intended protection policy: *Henry* can read and write *mailbox7*, *Robert* can read and write *secret*, and all three users can read and execute *compiler*.

Objects

	compiler	mailbox7	secret
Eric	Read Execute		
Henry	Read Execute	Read Write	
Robert	Read Execute		Read Write

(a)

Objects

	compiler	mailbox7	secret
Eric	Read Execute		
Henry	Read Execute	Read Write	
Robert	Read Execute	Read	Read Write

(b)

Fig. 4-32. (a) An authorized state. (b) An unauthorized state.

Now imagine that *Robert* is very clever and has found a way to issue commands to have the matrix changed to Fig. 4-32(b). He has now gained access to *mailbox7*, something he is not authorized to have. If he tries to read it, the operating system will carry out his request because it does not know that the state of Fig. 4-32(b) is unauthorized.

It should now be clear that the set of all possible matrices can be partitioned into two disjoint sets: the set of all authorized states and the set of all unauthorized states. A question around which much theoretical research has revolved is this: "Given an initial authorized state and a set of commands, can it be proven that the system can never reach an unauthorized state?"

In effect, we are asking if the available mechanism (the protection commands) is adequate to enforce some protection policy. As a simple example of a policy, consider the security scheme used by the military. Each object is unclassified, confidential, secret, or top secret. Each domain (and thus each process) also belongs to one of these four security levels. The security policy has two rules:

1. No process may read any object whose level is higher than its own, but it may freely read objects at a lower level or at its own level. A secret process may read confidential objects, but not top secret ones.
2. No process may write information into any object whose level is lower than its own. A secret process may write in a top secret file but not in a confidential one.

In military terms, if we assume that privates operate at confidential level, lieutenants at secret level, and generals at top secret level, then a lieutenant may look at a

private's papers, but not at a general's. A lieutenant may tell a general anything he knows, but he may not tell a private anything, because privates cannot be trusted.

Given this policy, some initial state of the matrix (including some way of telling which object is at which level), and the set of commands for modifying the matrix, what we would like is a way to prove that the system is secure. Such a proof turns out quite difficult to acquire; many general purpose systems are not theoretically secure. For more information, see Landwehr (1981) and Denning (1982).

4.5.5. Covert Channels

In the previous section we saw how it is possible to make formal models for protection systems. In this section we will see how futile it is to make such models. In particular, we will show that even in a system that has been rigorously proven to be absolutely secure, leaking information between processes that in theory cannot communicate at all is relatively straightforward. These ideas are due to Lampson (1973).

Lampson's model involves three processes, and is primarily applicable to large timesharing systems. The first process is the client, which wants some work performed by the second one, the server. The client and the server do not entirely trust each other. For example, the server's job is to help clients with filling out their tax forms. The clients are worried that the server will secretly record their financial data, for example, maintaining a secret list of who earns how much, and then selling the list. The server is worried that the clients will try to steal the valuable tax program.

The third process is the collaborator, which is conspiring with the server to indeed steal the client's confidential data. The collaborator and server are typically owned by the same person. These three processes are shown in Fig. 4-33. The object of this exercise is to design a system in which it is impossible for the server to leak to the collaborator the information that it has legitimately received from the client. Lampson called this the **confinement problem**.

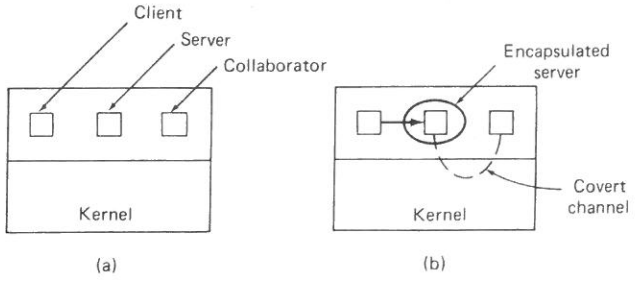

Fig. 4-33. (a) The client, server and collaborator processes. (b) The encapsulated server can still leak to the collaborator via covert channels.

From the system designer's point of view, the goal is to encapsulate or confine the server in such a way that it cannot pass information to the collaborator. Using a protection matrix scheme we can easily guarantee that the server cannot communicate with the collaborator by writing into a file to which the collaborator has read

access. We can probably also ensure that the server cannot communicate with the collaborator by using the system's interprocess communication mechanism.

Unfortunately, more subtle communication channels may be available. For example, the server can try to communicate a binary bit stream as follows. To send a 1 bit, it computes as hard as it can for a fixed interval of time. To send a 0 bit, it goes to sleep for the same length of time.

The collaborator can try to detect the bit stream by carefully monitoring its response time. In general, it will get better response when the server is sending a 0 than when the server is sending a 1. This communication channel is known as a **covert channel**, and is illustrated in Fig. 4-33(b).

Of course the covert channel is a noisy channel, containing a lot of extraneous information, but information can be reliably sent over a noisy channel by using an error-correcting code (e.g., a Hamming code, or even something more sophisticated). The use of an error-correcting code reduces the already low bandwidth of the covert channel even more, but it still may be enough to leak substantial information. It is fairly obvious that no protection model based on a matrix of objects and domains is going to prevent this kind of leakage.

Modulating the CPU usage is not the only covert channel. The paging rate can also be modulated (many page faults for a 1, no page faults for a 0). In fact, almost any way of degrading system performance in a clocked way is a candidate. If the system provides a way of locking files, then the server can lock some file to indicate a 1, and unlock it to indicate a 0. It may be possible to detect the status of a lock even on a file that you cannot access.

Acquiring and releasing dedicated resources (tape drives, plotters, etc.) can also be used for signaling. The server acquires the resource to send a 1 and releases it to send a 0. In UNIX, the server could create a file to indicate a 1 and remove it to indicate a 0; the collaborator could use the ACCESS system call to see if the file exists. This call works even though the collaborator has no permission to use the file. Unfortunately, many other covert channels exist.

Lampson also mentions a way of leaking information to the (human) owner of the server process. Presumably the server process will be entitled to tell its owner how much work it did on behalf of the client, so the client can be billed. If the actual computing bill is, say, 100 dollars and the client's income is 53K dollars, the server could report the bill as 100.53 to its owner.

Just finding all the covert channels, let alone blocking them, is extremely difficult. In practice, there is little that can be done. Introducing a process that causes page faults at random, or otherwise spends its time degrading system performance in order to reduce the bandwidth of the covert channels is not an attractive proposition.

4.6. 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. File naming,

structure, typing, access, and attributes are all important design issues. 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 implementers have to be concerned with keeping track of which disk blocks go with which file, how files can be shared, and how free disk space is managed. Directories can be arranged in various ways, ranging from putting the name, attributes, and disk addresses there to putting just the name and an i-node number there. Shared files, bad block management, backups, consistency, and caching are also important issues.

Security and protection are of vital concern to both the system users and implementers. We have 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.

PROBLEMS

1. Give 5 different path names for the file */etc/passwd*. (Hint: think about the directory entries "." and "..".)
2. Systems that support sequential files always have an operation to rewind files. Do systems that support random access files need this too?
3. In the list of file attributes in Fig. 4-4, one of the candidates is a bit that marks a file as temporary, and thus subject to automatic deletion when the process terminates. What is the point of having this? After all, the process can delete its own files when it is done, as is the case in Fig. 4-5.
4. Some operating systems provide a system call RENAME to give a file a new name. Is there any difference at all between using this call to rename a file, and just copying the file to a new file with the new name, followed by deleting the old one?
5. What happens if the program of Fig. 4-5 is called with a zero-length file as its first argument?
6. Consider the directory tree of Fig. 4-9. If */usr/jim* is the working directory, what is the absolute path name for the file whose relative path name is *../ast/x*?
7. Contiguous allocation of files leads to disk fragmentation, as mentioned in the text. Is this internal fragmentation or external fragmentation? Make an analogy with something discussed in the previous chapter.
8. An operating system only supports a single directory, but allows that directory to have arbitrarily many files with arbitrarily long file names. Can something approximating a hierarchical file system be simulated? How?
9. Free disk space can be kept track of using a free list or a bit map. Disk addresses require D bits. For a disk with B blocks, F of which are free, state the condition under which the free list uses less space than the bit map. For D having the value 16 bits, express your answer as a percentage of the disk space that must be free.