4

Protection in General-Purpose Operating Systems

In this chapter:

- Protection features provided by general-purpose operating systems: protecting memory, files, and the execution environment
- Controlled access to objects
- User authentication

In the previous chapter, we looked at several types of security problems that can occur in programs. The problems may be unintentional, as with buffer overflows, or intentional, as when a virus or worm is inserted in code. In addition to these general problems, some kinds of programs may be vulnerable to certain kinds of security problems simply because of the nature of the program itself. For example, operating systems and databases offer security challenges beyond those in more general programs; these programs offer different access to different items by different kinds of users, so program designers must pay careful attention to defining access, granting access, and controlling intentional and unintentional corruption of data and relationships. For this reason, we devote three chapters to these specialized programs and their particular security concerns. In this chapter and the next, we study operating systems and their role in computer security; we look at databases in Chapter 6.

An operating system has two goals: controlling shared access and implementing an interface to allow that access. Underneath those goals are support activities, including identification and authentication, naming, filing objects, scheduling, communication among processes, and reclaiming and reusing objects. Operating system functions can be categorized as

- access control
- identity and credential management

- information flow
- audit and integrity protection

Each of these activities has security implications. Operating systems range from simple ones supporting a single task at a time (such an operating system might run a personal digital assistant) to complex multiuser, multitasking systems, and, naturally, security considerations increase as operating systems become more complex.

We begin by studying the contributions that operating systems have made to user security. An operating system supports multiprogramming (that is, the concurrent use of a system by more than one user), so operating system designers have developed ways to protect one user's computation from inadvertent or malicious interference by another user. Among those facilities provided for this purpose are memory protection, file protection, general control of access to objects, and user authentication. This chapter surveys the controls that provide these four features. We have oriented this discussion to the user: How do the controls protect users, and how do users apply those controls? In the next chapter, we see how operating system design is affected by the need to separate levels of security considerations for particular users.

There are many commercially available operating systems, but we draw examples largely from two families: the Microsoft Windows NT, 2000, XP, 2003 Server, and Vista operating systems (which we denote NT+) and Unix, Linux, and their derivatives (which we call Unix+). Other proprietary operating systems are in wide use, notably Apple's Mac OS X (based on a system called Darwin that is derived from Mach and FreeBSD) and IBM's z/OS, the successor to S/390, but for security purposes, NT+ and Unix+ are the most widely known.

4.1 PROTECTED OBJECTS AND METHODS OF PROTECTION

We begin by reviewing the history of protection in operating systems. This background helps us understand what kinds of things operating systems can protect and what methods are available for protecting them. (Readers who already have a good understanding of operating system capabilities may want to jump to Section 4.3.)

A Bit of History

Once upon a time, there were no operating systems: Users entered their programs directly into the machine in binary by means of switches. In many cases, program entry was done by physical manipulation of a toggle switch; in other cases, the entry was performed with a more complex electronic method, by means of an input device such as a keyboard. Because each user had exclusive use of the computing system, users were required to schedule blocks of time for running the machine. These users were responsible for loading their own libraries of support routines—assemblers, compilers, shared subprograms—and "cleaning up" after use by removing any sensitive code or data.

The first operating systems were simple utilities, called **executives**, designed to assist individual programmers and to smooth the transition from one user to another. The early executives provided linkers and loaders for relocation, easy access to compilers and assemblers, and automatic loading of subprograms from libraries. The executives

handled the tedious aspects of programmer support, focusing on a single programmer during execution.

Operating systems took on a much broader role (and a different name) as the notion of multiprogramming was implemented. Realizing that two users could interleave access to the resources of a single computing system, researchers developed concepts such as scheduling, sharing, and parallel use. **Multiprogrammed operating systems**, also known as **monitors**, oversaw each program's execution. Monitors took an active role, whereas executives were passive. That is, an executive stayed in the background, waiting to be called into service by a requesting user. But a monitor actively asserted control of the computing system and gave resources to the user only when the request was consistent with general good use of the system. Similarly, the executive waited for a request and provided service on demand; the monitor maintained control over all resources, permitting or denying all computing and loaning resources to users as they needed them.

Multiprogramming brought another important change to computing. When a single person was using a system, the only force to be protected against was the user himself or herself. A user making an error may have felt foolish, but one user could not adversely affect the computation of any other user. However, multiple users introduced more complexity and risk. User A might rightly be angry if User B's programs or data had a negative effect on A's program's execution. Thus, protecting one user's programs and data from other users' programs became an important issue in multiprogrammed operating systems.

Protected Objects

In fact, the rise of multiprogramming meant that several aspects of a computing system required protection:

- memory
- sharable I/O devices, such as disks
- serially reusable I/O devices, such as printers and tape drives
- sharable programs and subprocedures
- networks
- sharable data

As it assumed responsibility for controlled sharing, the operating system had to protect these objects. In the following sections, we look at some of the mechanisms with which operating systems have enforced these objects' protection. Many operating system protection mechanisms have been supported by hardware. But, as noted in Sidebar 4-1, that approach is not always possible.

Security Methods of Operating Systems

The basis of protection is **separation**: keeping one user's objects separate from other users. Rushby and Randell [RUS83] note that separation in an operating system can occur in several ways:

Sidebar 4-1 Hardware-Enforced Protection

From the 1960s to the 1980s, vendors produced both hardware and the software to run on it. The major mainframe operating systems—such as IBM's MVS, Digital Equipment's VAX, and Burroughs's and GE's operating systems, as well as research systems such as KSOS, PSOS, KVM, Multics, and SCOMP—were designed to run on one family of hardware. The VAX family, for example, used a hardware design that implemented four distinct protection levels: Two were reserved for the operating system, a third for system utilities, and the last went to users' applications. This structure put essentially three distinct walls around the most critical functions, including those that implemented security. Anything that allowed the user to compromise the wall between user state and utility state still did not give the user access to the most sensitive protection features. A BiiN operating system from the late 1980s offered an amazing 64,000 different levels of protection (or separation) enforced by the hardware.

Two factors changed this situation. First, the U.S. government sued IBM in 1969, claiming that IBM had exercised unlawful monopolistic practices. As a consequence, during the 1970s IBM made its hardware available to run with other vendors' operating systems (thereby opening its specifications to competitors). This relaxation encouraged more openness in operating system selection: Users were finally able to buy hardware from one manufacturer and go elsewhere for some or all of the operating system. Second, the Unix operating system, begun in the early 1970s, was designed to be largely independent of the hardware on which it ran. A small kernel had to be recoded for each different kind of hardware platform, but the bulk of the operating system, running on top of that kernel, could be ported without change.

These two situations together meant that the operating system could no longer depend on hardware support for all its critical functionality. So, although an operating system might still be structured to reach several states, the underlying hardware might enforce separation between only two of those states, with the remainder being enforced in software.

Today three of the most prevalent families of operating systems—the Windows NT/2000/XP series, Unix, and Linux—run on many different kinds of hardware. (Only Apple's Mac OS is strongly integrated with its hardware base.) The default expectation is one level of hardware-enforced separation (two states). This situation means that an attacker is only one step away from complete system compromise through a "get_root" exploit. (See this chapter's "Where the Field Is Headed" section to read of a recent Microsoft initiative to reintroduce hardware-enforced separation for security-critical code and data.)

- *physical separation*, in which different processes use different physical objects, such as separate printers for output requiring different levels of security
- *temporal separation*, in which processes having different security requirements are executed at different times
- *logical separation*, in which users operate under the illusion that no other processes exist, as when an operating system constrains a program's accesses so that the program cannot access objects outside its permitted domain

- *cryptographic separation*, in which processes conceal their data and computations in such a way that they are unintelligible to outside processes

Of course, combinations of two or more of these forms of separation are also possible.

The categories of separation are listed roughly in increasing order of complexity to implement, and, for the first three, in decreasing order of the security provided. However, the first two approaches are very stringent and can lead to poor resource utilization. Therefore, we would like to shift the burden of protection to the operating system to allow concurrent execution of processes having different security needs.

But separation is only half the answer. We want to separate users and their objects, but we also want to be able to provide sharing for some of those objects. For example, two users with different security levels may want to invoke the same search algorithm or function call. We would like the users to be able to share the algorithms and functions without compromising their individual security needs. An operating system can support separation and sharing in several ways, offering protection at any of several levels.

- *Do not protect.* Operating systems with no protection are appropriate when sensitive procedures are being run at separate times.
- *Isolate.* When an operating system provides isolation, different processes running concurrently are unaware of the presence of each other. Each process has its own address space, files, and other objects. The operating system must confine each process somehow so that the objects of the other processes are completely concealed.
- *Share all or share nothing.* With this form of protection, the owner of an object declares it to be public or private. A public object is available to all users, whereas a private object is available only to its owner.
- *Share via access limitation.* With protection by access limitation, the operating system checks the allowability of each user's potential access to an object. That is, access control is implemented for a specific user and a specific object. Lists of acceptable actions guide the operating system in determining whether a particular user should have access to a particular object. In some sense, the operating system acts as a guard between users and objects, ensuring that only authorized accesses occur.
- *Share by capabilities.* An extension of limited access sharing, this form of protection allows dynamic creation of sharing rights for objects. The degree of sharing can depend on the owner or the subject, on the context of the computation, or on the object itself.
- *Limit use of an object.* This form of protection limits not just the access to an object but the use made of that object after it has been accessed. For example, a user may be allowed to view a sensitive document, but not to print a copy of it. More powerfully, a user may be allowed access to data in a database to derive statistical summaries (such as average salary at a particular grade level), but not to determine specific data values (salaries of individuals).

Again, these modes of sharing are arranged in increasing order of difficulty to implement, but also in increasing order of fineness of protection they provide. A given

operating system may provide different levels of protection for different objects, users, or situations.

When we think about data, we realize that access can be controlled at various levels: the bit, the byte, the element or word, the field, the record, the file, or the volume. Thus, the granularity of control concerns us. The larger the level of object controlled, the easier it is to implement access control. However, sometimes the operating system must allow access to more than the user needs. For example, with large objects, a user needing access only to part of an object (such as a single record in a file) must be given access to the entire object (the whole file).

Let us examine in more detail several different kinds of objects and their specific kinds of protection.

4.2 MEMORY AND ADDRESS PROTECTION

The most obvious problem of multiprogramming is preventing one program from affecting the data and programs in the memory space of other users. Fortunately, protection can be built into the hardware mechanisms that control efficient use of memory, so solid protection can be provided at essentially no additional cost.

Fence

The simplest form of memory protection was introduced in single-user operating systems to prevent a faulty user program from destroying part of the resident portion of the operating system. As its name implies, a **fence** is a method to confine users to one side of a boundary.

In one implementation, the fence was a predefined memory address, enabling the operating system to reside on one side and the user to stay on the other. An example of this situation is shown in Figure 4-1. Unfortunately, this kind of implementation was very restrictive because a predefined amount of space was always reserved for the

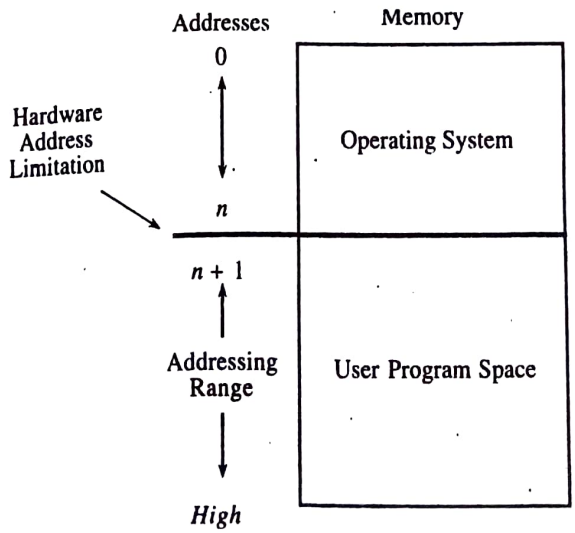

FIGURE 4-1 Fixed Fence.

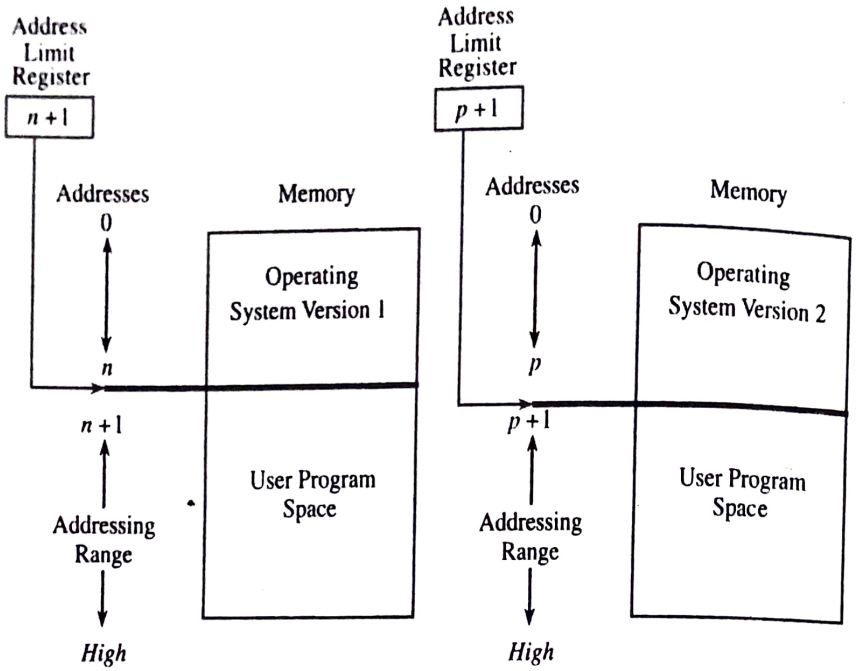

FIGURE 4-2 Variable Fence Register.

operating system, whether it was needed or not. If less than the predefined space was required, the excess space was wasted. Conversely, if the operating system needed more space, it could not grow beyond the fence boundary.

Another implementation used a hardware register, often called a **fence register**, containing the address of the end of the operating system. In contrast to a fixed fence, in this scheme the location of the fence could be changed. Each time a user program generated an address for data modification, the address was automatically compared with the fence address. If the address was greater than the fence address (that is, in the user area), the instruction was executed; if it was less than the fence address (that is, in the operating system area), an error condition was raised. The use of fence registers is shown in Figure 4-2.

A fence register protects only in one direction. In other words, an operating system can be protected from a single user, but the fence cannot protect one user from another user. Similarly, a user cannot identify certain areas of the program as inviolable (such as the code of the program itself or a read-only data area).

Relocation

If the operating system can be assumed to be of a fixed size, programmers can write their code assuming that the program begins at a constant address. This feature of the operating system makes it easy to determine the address of any object in the program. However, it also makes it essentially impossible to change the starting address if, for example, a new version of the operating system is larger or smaller than the old. If the

size of the operating system is allowed to change, then programs must be written in a way that does not depend on placement at a specific location in memory.

Relocation is the process of taking a program written as if it began at address 0 and changing all addresses to reflect the actual address at which the program is located in memory. In many instances, this effort merely entails adding a constant **relocation factor** to each address of the program. That is, the relocation factor is the starting address of the memory assigned for the program.

Conveniently, the fence register can be used in this situation to provide an important extra benefit: The fence register can be a hardware relocation device. The contents of the fence register are added to each program address. This action both relocates the address and guarantees that no one can access a location lower than the fence address. (Addresses are treated as unsigned integers, so adding the value in the fence register to any number is guaranteed to produce a result at or above the fence address.) Special instructions can be added for the few times when a program legitimately intends to access a location of the operating system.

Base/Bounds Registers

A major advantage of an operating system with fence registers is the ability to relocate; this characteristic is especially important in a multiuser environment. With two or more users, none can know in advance where a program will be loaded for execution. The relocation register solves the problem by providing a base or starting address. All addresses inside a program are offsets from that base address. A variable fence register is generally known as a **base register**.

Fence registers provide a lower bound (a starting address) but not an upper one. An upper bound can be useful in knowing how much space is allotted and in checking for overflows into "forbidden" areas. To overcome this difficulty, a second register is often added, as shown in Figure 4-3. The second register, called a **bounds register**, is an upper address limit, in the same way that a base or fence register is a lower address limit. Each program address is forced to be above the base address because the contents of the base register are added to the address; each address is also checked to ensure that it is below the bounds address. In this way, a program's addresses are neatly confined to the space between the base and the bounds registers.

This technique protects a program's addresses from modification by another user. When execution changes from one user's program to another's, the operating system must change the contents of the base and bounds registers to reflect the true address space for that user. This change is part of the general preparation, called a **context switch**, that the operating system must perform when transferring control from one user to another.

With a pair of base/bounds registers, a user is perfectly protected from outside users, or, more correctly, outside users are protected from errors in any other user's program. Erroneous addresses *inside* a user's address space can still affect that program because the base/bounds checking guarantees only that each address is inside the user's address space. For example, a user error might occur when a subscript is out of range or an undefined variable generates an address reference within the user's space but, unfortunately, inside the executable instructions of the user's program. In this

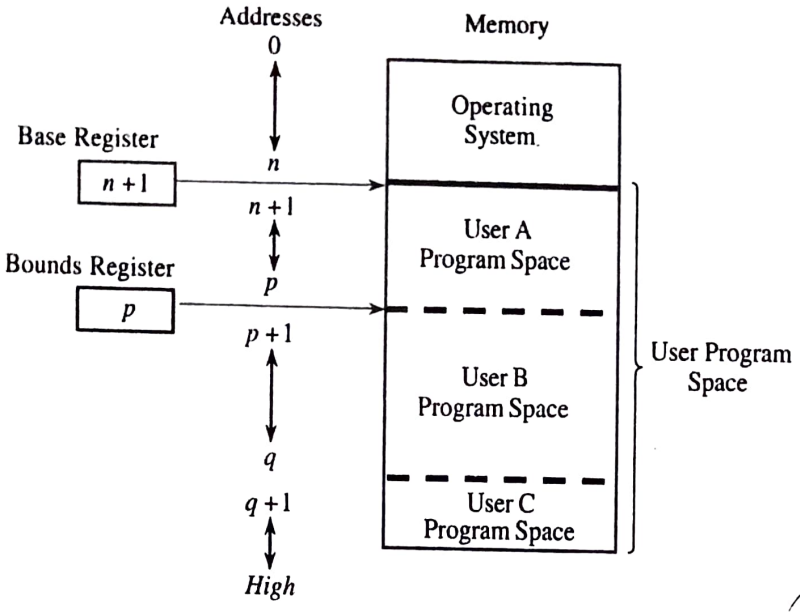

FIGURE 4-3 Pair of Base/Bounds Registers.

manner, a user can accidentally store data on top of instructions. Such an error can let a user inadvertently destroy a program, but (fortunately) only the user's own program.

We can solve this overwriting problem by using another pair of base/bounds registers, one for the instructions (code) of the program and a second for the data space. Then, only instruction fetches (instructions to be executed) are relocated and checked with the first register pair, and only data accesses (operands of instructions) are relocated and checked with the second register pair. The use of two pairs of base/bounds registers is shown in Figure 4-4. Although two pairs of registers do not prevent all program errors, they limit the effect of data-manipulating instructions to the data space. The pairs of registers offer another more important advantage: the ability to split a program into two pieces that can be relocated separately.

These two features seem to call for the use of three or more pairs of registers: one for code, one for read-only data, and one for modifiable data values. Although in theory this concept can be extended, two pairs of registers are the limit for practical computer design. For each additional pair of registers (beyond two), something in the machine code of each instruction must indicate which relocation pair is to be used to address the instruction's operands. That is, with more than two pairs, each instruction specifies one of two or more data spaces. But with only two pairs, the decision can be automatic: instructions with one pair, data with the other.

Tagged Architecture

Another problem with using base/bounds registers for protection or relocation is their contiguous nature. Each pair of registers confines accesses to a consecutive range of addresses. A compiler or loader can easily rearrange a program so that all code sections are adjacent and all data sections are adjacent.

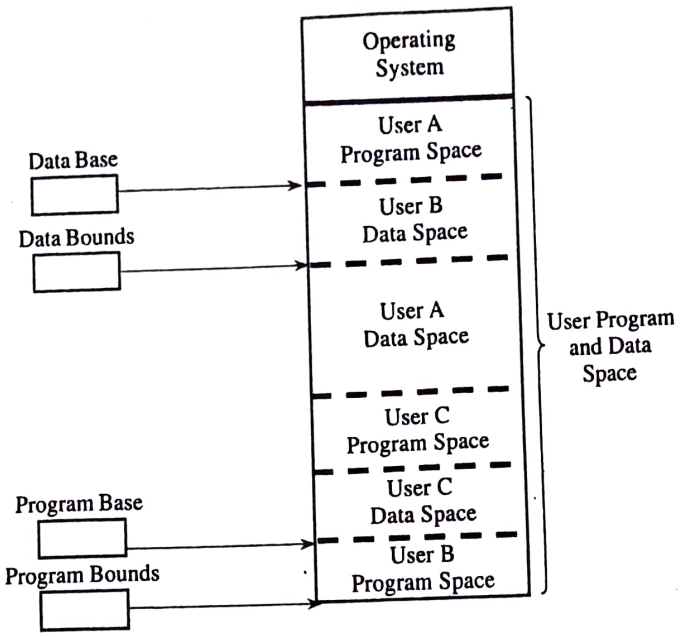

FIGURE 4-4 Two Pairs of Base/Bounds Registers.

However, in some cases you may want to protect *some* data values but not *all*. For example, a personnel record may require protecting the field for salary but not office location and phone number. Moreover, a programmer may want to ensure the integrity of certain data values by allowing them to be written when the program is initialized but prohibiting the program from modifying them later. This scheme protects against errors in the programmer's own code. A programmer may also want to invoke a shared subprogram from a common library. We can address some of these issues by using good design, both in the operating system and in the other programs being run. Recall that in Chapter 3 we studied good design characteristics such as information hiding and modularity in program design. These characteristics dictate that one program module must share with another module only the *minimum* amount of data necessary for both of them to do their work.

Additional, operating-system-specific design features can help, too. Base/bounds registers create an all-or-nothing situation for sharing: Either a program makes all its data available to be accessed and modified or it prohibits access to all. Even if there were a third set of registers for shared data, all data would need to be located together. A procedure could not effectively share data items *A*, *B*, and *C* with one module, *A*, *C*, and *D* with a second, and *A*, *B*, and *D* with a third. The only way to accomplish the kind of sharing we want would be to move each appropriate set of data values to some contiguous space. However, this solution would not be acceptable if the data items were large records, arrays, or structures.

An alternative is **tagged architecture**, in which every word of machine memory has one or more extra bits to identify the access rights to that word. These access bits

Tag	Memory Word
R	0001
RW	0137
R	0099
X	
X	
X	
X	
X	
X	
R	4091
RW	0002

Code: R = Read-only RW = Read/Write
X = Execute-only

FIGURE 4-5 Example of Tagged Architecture.

can be set only by privileged (operating system) instructions. The bits are tested every time an instruction accesses that location.

For example, as shown in Figure 4-5, one memory location may be protected as execute-only (for example, the object code of instructions), whereas another is protected for fetch-only (for example, read) data access, and another accessible for modification (for example, write). In this way, two adjacent locations can have different access rights. Furthermore, with a few extra tag bits, different classes of data (numeric, character, address or pointer, and undefined) can be separated, and data fields can be protected for privileged (operating system) access only.

This protection technique has been used on a few systems, although the number of tag bits has been rather small. The Burroughs B6500-7500 system used three tag bits to separate data words (three types), descriptors (pointers), and control words (stack pointers and addressing control words). The IBM System/38 used a tag to control both integrity and access.

A variation used one tag that applied to a group of consecutive locations, such as 128 or 256 bytes. With one tag for a block of addresses, the added cost for implementing tags was not as high as with one tag per location. The Intel I960 extended architecture processor used a tagged architecture with a bit on each memory word that marked the word as a "capability," not as an ordinary location for data or instructions. A capability controlled access to a variable-sized memory block or segment. This large number of possible tag values supported memory segments that ranged in size from 64 to 4 billion bytes, with a potential 2^{256} different protection domains.

Compatibility of code presented a problem with the acceptance of a tagged architecture. A tagged architecture may not be as useful as more modern approaches, as we see

shortly. Some of the major computer vendors are still working with operating systems that were designed and implemented many years ago for architectures of that era. Indeed, most manufacturers are locked into a more conventional memory architecture because of the wide availability of components and a desire to maintain compatibility among operating systems and machine families. A tagged architecture would require fundamental changes to substantially all the operating system code, a requirement that can be prohibitively expensive. But as the price of memory continues to fall, the implementation of a tagged architecture becomes more feasible.

Segmentation

We present two more approaches to protection, each of which can be implemented on top of a conventional machine structure, suggesting a better chance of acceptance. Although these approaches are ancient by computing's standards—they were designed between 1965 and 1975—they have been implemented on many machines since then. Furthermore, they offer important advantages in addressing, with memory protection being a delightful bonus.

The first of these two approaches, **segmentation**, involves the simple notion of dividing a program into separate pieces. Each piece has a logical unity, exhibiting a relationship among all of its code or data values. For example, a segment may be the code of a single procedure, the data of an array, or the collection of all local data values used by a particular module. Segmentation was developed as a feasible means to produce the effect of the equivalent of an unbounded number of base/bounds registers. In other words, segmentation allows a program to be divided into many pieces having different access rights.

Each segment has a unique name. A code or data item within a segment is addressed as the pair ⟨*name*, *offset*⟩, where *name* is the name of the segment containing the data item and *offset* is its location within the segment (that is, its distance from the start of the segment).

Logically, the programmer pictures a program as a long collection of segments. Segments can be separately relocated, allowing any segment to be placed in any available memory locations. The relationship between a logical segment and its true memory position is shown in Figure 4-6.

The operating system must maintain a table of segment names and their true addresses in memory. When a program generates an address of the form ⟨*name*, *offset*⟩, the operating system looks up *name* in the segment directory and determines its real beginning memory address. To that address the operating system adds *offset*, giving the true memory address of the code or data item. This translation is shown in Figure 4-7. For efficiency there is usually one operating system **segment address table** for each process in execution. Two processes that need to share access to a single segment would have the same segment name and address in their segment tables.

Thus, a user's program does not know what true memory addresses it uses. It has no way—and no need—to determine the actual address associated with a particular ⟨*name*, *offset*⟩. The ⟨*name*, *offset*⟩ pair is adequate to access any data or instruction to which a program should have access.

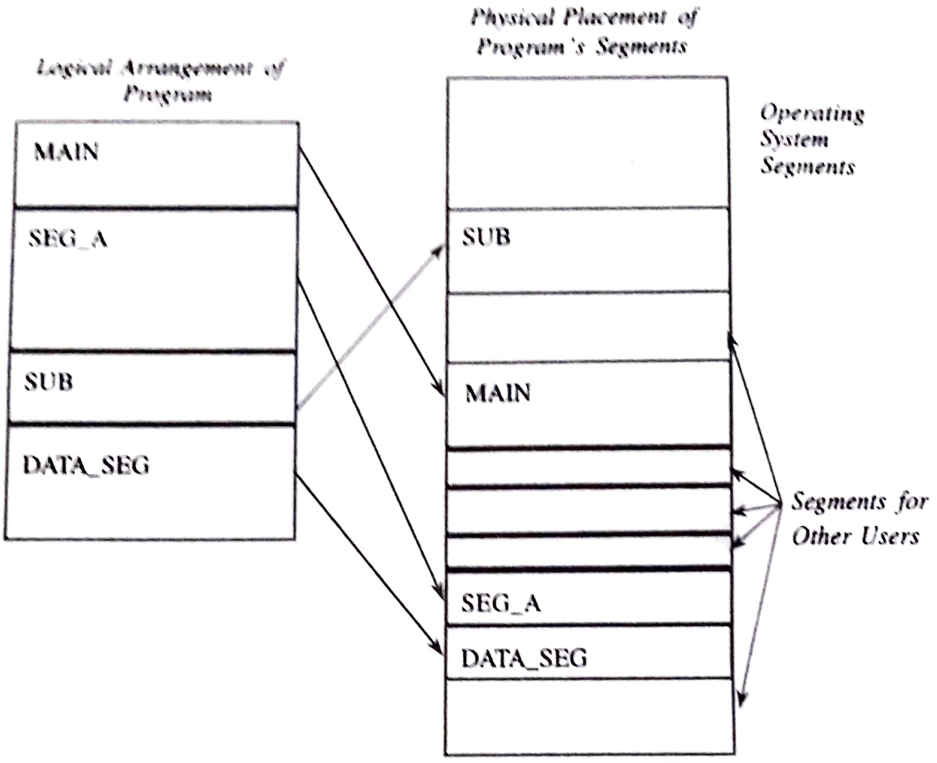

FIGURE 4-6 Logical and Physical Representation of Segments.

This hiding of addresses has three advantages for the operating system.

- The operating system can place any segment at any location or move any segment to any location, even after the program begins to execute. Because it translates all address references by a segment address table, the operating system needs only update the address in that one table when a segment is moved.
- A segment can be removed from main memory (and stored on an auxiliary device) if it is not being used currently.
- Every address reference passes through the operating system, so there is an opportunity to check each one for protection.

Because of this last characteristic, a process can access a segment only if that segment appears in that process's segment translation table. The operating system controls which programs have entries for a particular segment in their segment address tables. This control provides strong protection of segments from access by unpermitted processes. For example, program *A* might have access to segments *BLUE* and *GREEN* of user *X* but not to other segments of that user or of any other user. In a straightforward way we can allow a user to have different protection classes for different segments of a program. For example, one segment might be read-only data, a second might be execute-only code, and a third might be writeable data. In a situation like this

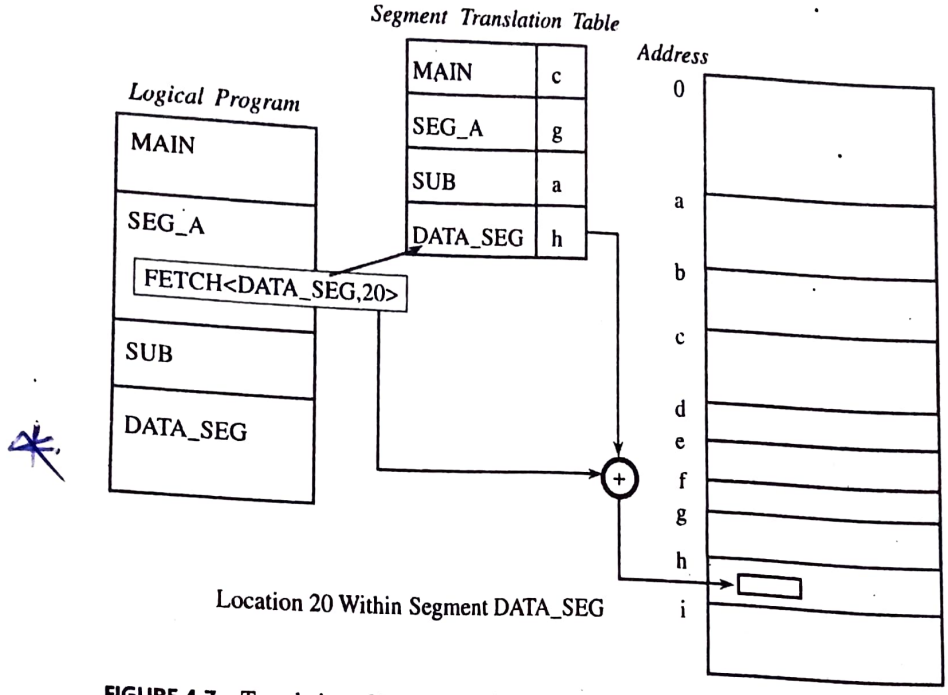

FIGURE 4-7 Translation of Segment Address.

one, segmentation can approximate the goal of separate protection of different pieces of a program, as outlined in the previous section on tagged architecture.

Segmentation offers these security benefits:

- Each address reference is checked for protection.
- Many different classes of data items can be assigned different levels of protection.
- Two or more users can share access to a segment, with potentially different access rights.
- A user cannot generate an address or access to an unpermitted segment.

One protection difficulty inherent in segmentation concerns segment size. Each segment has a particular size. However, a program can generate a reference to a valid segment *name*, but with an *offset* beyond the end of the segment. For example, reference ⟨*A*,9999⟩ looks perfectly valid, but in reality segment *A* may be only 200 bytes long. If left unplugged, this security hole could allow a program to access any memory address beyond the end of a segment just by using large values of *offset* in an address.

This problem cannot be stopped during compilation or even when a program is loaded, because effective use of segments requires that they be allowed to grow in size during execution. For example, a segment might contain a dynamic data structure such as a stack. Therefore, secure implementation of segmentation requires checking a generated address to verify that it is not beyond the current end of the segment referenced. Although this checking results in extra expense (in terms of time and resources), segmentation systems must perform this check; the segmentation process must main-

tain the current segment length in the translation table and compare every address generated.

Thus, we need to balance protection with efficiency, finding ways to keep segmentation as efficient as possible. However, efficient implementation of segmentation presents two problems: Segment names are inconvenient to encode in instructions, and the operating system's lookup of the name in a table can be slow. To overcome these difficulties, segment names are often converted to numbers by the compiler when a program is translated; the compiler also appends a linkage table matching numbers to true segment names. Unfortunately, this scheme presents an implementation difficulty when two procedures need to share the same segment because the assigned segment numbers of data accessed by that segment must be the same.

Paging

One alternative to segmentation is **paging**. The program is divided into equal-sized pieces called pages, and memory is divided into equal-sized units called page frames. (For implementation reasons, the page size is usually chosen to be a power of two between 512 and 4096 bytes.) As with segmentation, each address in a paging scheme is a two-part object, this time consisting of ⟨*page*, *offset*⟩.

Each address is again translated by a process similar to that of segmentation: The operating system maintains a table of user page numbers and their true addresses in memory. The *page* portion of every ⟨*page*, *offset*⟩ reference is converted to a page frame address by a table lookup; the *offset* portion is added to the page frame address to produce the real memory address of the object referred to as ⟨*page*, *offset*⟩. This process is illustrated in Figure 4-8.

Unlike segmentation, all pages in the paging approach are of the same fixed size, so fragmentation is not a problem. Each page can fit in any available page in memory, and thus there is no problem of addressing beyond the end of a page. The binary form of a ⟨*page*, *offset*⟩ address is designed so that the *offset* values fill a range of bits in the address. Therefore, an *offset* beyond the end of a particular page results in a carry into the *page* portion of the address, which changes the address.

To see how this idea works, consider a page size of 1024 bytes ($1024 = 2^{10}$), where 10 bits are allocated for the *offset* portion of each address. A program cannot generate an *offset* value larger than 1023 in 10 bits. Moving to the next location after ⟨*x*,1023⟩ causes a carry into the *page* portion, thereby moving translation to the next page. During the translation, the paging process checks to verify that a ⟨*page*, *offset*⟩ reference does not exceed the maximum number of pages the process has defined.

With a segmentation approach, a programmer must be conscious of segments. However, a programmer is oblivious to page boundaries when using a paging-based operating system. Moreover, with paging there is no logical unity to a page; a page is simply the next 2^n bytes of the program. Thus, a change to a program, such as the addition of one instruction, pushes all subsequent instructions to lower addresses and moves a few bytes from the end of each page to the start of the next. This shift is not something about which the programmer need be concerned because the entire mechanism of paging and address translation is hidden from the programmer.

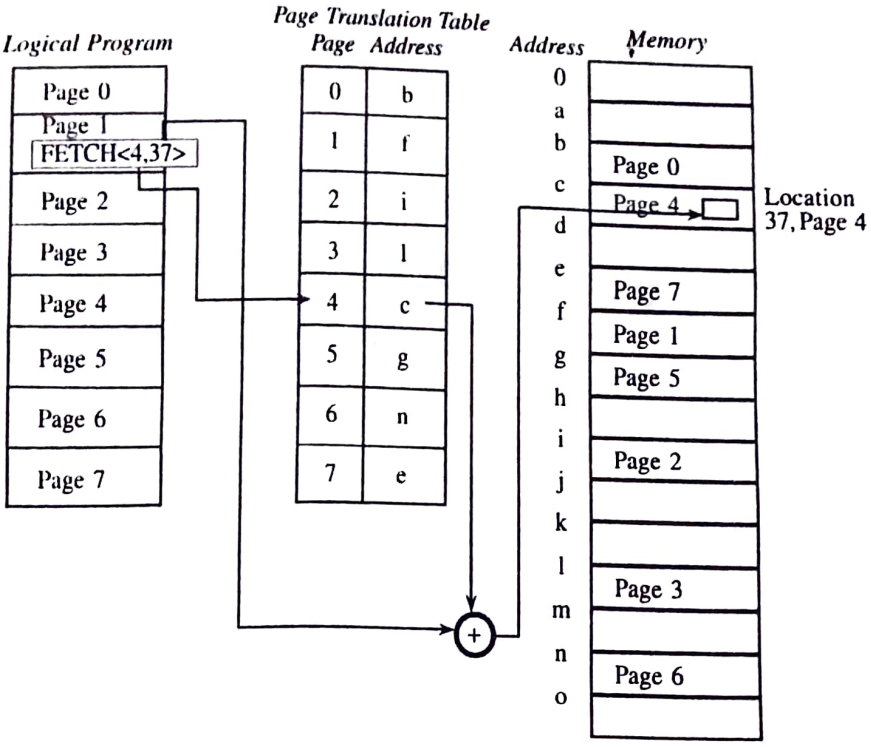

FIGURE 4-8 Page Address Translation.

However, when we consider protection, this shift is a serious problem. Because segments are logical units, we can associate different segments with individual protection rights, such as read-only or execute-only. The shifting can be handled efficiently during address translation. But with paging there is no necessary unity to the items on a page, so there is no way to establish that all values on a page should be protected at the same level, such as read-only or execute-only.

Combined Paging with Segmentation

We have seen how paging offers implementation efficiency, while segmentation offers logical protection characteristics. Since each approach has drawbacks as well as desirable features, the two approaches have been combined.

The IBM 390 family of mainframe systems used a form of paged segmentation. Similarly, the Multics operating system (implemented on a GE-645 machine) applied paging on top of segmentation. In both cases, the programmer could divide a program into logical segments. Each segment was then broken into fixed-size pages. In Multics, the segment *name* portion of an address was an 18-bit number with a 16-bit *offset*. The addresses were then broken into 1024-byte pages. The translation process is shown in Figure 4-9. This approach retained the logical unity of a segment and permitted differentiated protection for the segments, but it added an additional layer of translation for each address. Additional hardware improved the efficiency of the implementation.

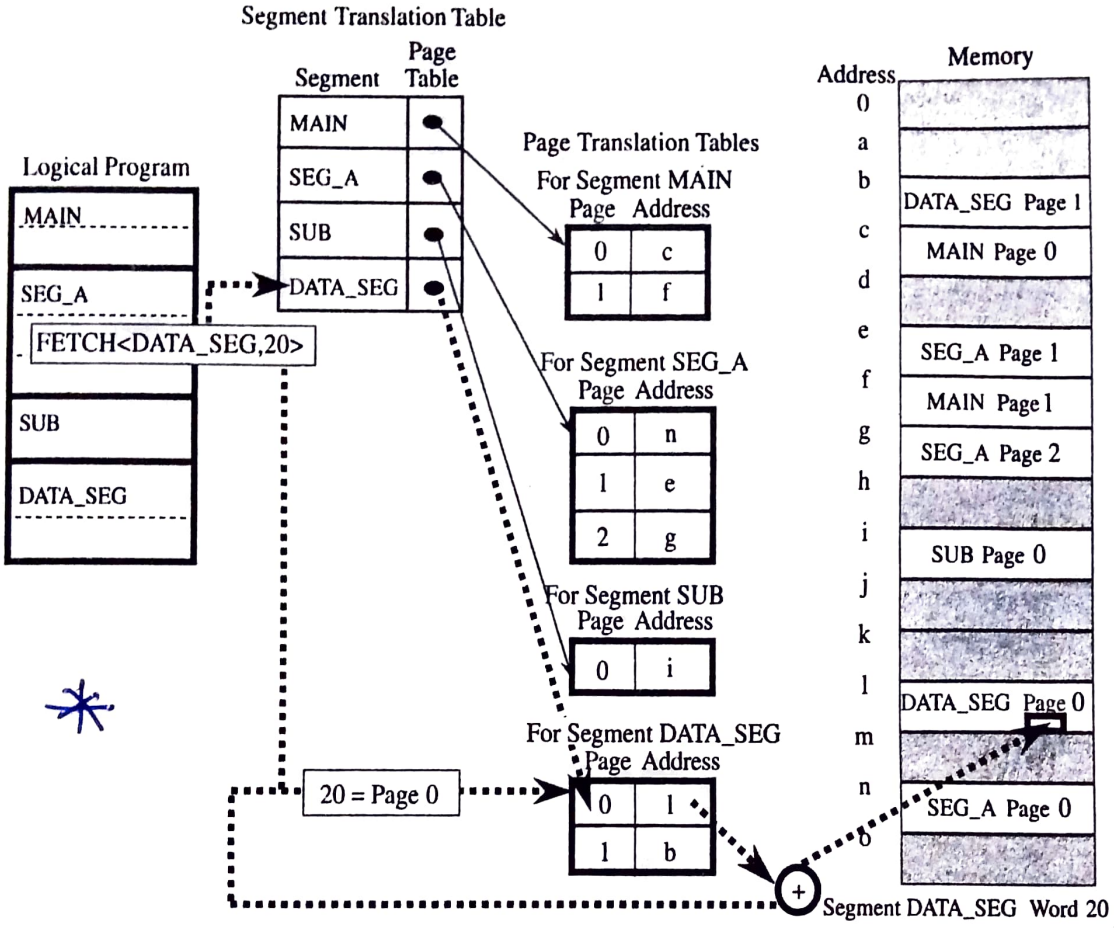

FIGURE 4-9 Paged Segmentation.

4.3 CONTROL OF ACCESS TO GENERAL OBJECTS

Protecting memory is a specific case of the more general problem of protecting *objects*. As multiprogramming has developed, the numbers and kinds of objects shared have also increased. Here are some examples of the kinds of objects for which protection is desirable:

- memory
- a file or data set on an auxiliary storage device
- an executing program in memory
- a directory of files
- a hardware device
- a data structure, such as a stack
- a table of the operating system
- instructions, especially privileged instructions

- passwords and the user authentication mechanism
- the protection mechanism itself

The memory protection mechanism can be fairly simple because every memory access is guaranteed to go through certain points in the hardware. With more general objects, the number of points of access may be larger, a central authority through which all accesses pass may be lacking, and the kind of access may not simply be limited to read, write, or execute.

Furthermore, all accesses to memory occur through a program, so we can refer to the program or the programmer as the accessing agent. In this book, we use terms like *the user* or *the subject* in describing an access to a general object. This user or subject could be a person who uses a computing system, a programmer, a program, another object, or something else that seeks to use an object.

There are several complementary goals in protecting objects.

- *Check every access.* We may want to revoke a user's privilege to access an object. If we have previously authorized the user to access the object, we do not necessarily intend that the user should retain indefinite access to the object. In fact, in some situations, we may want to prevent further access immediately after we revoke authorization. For this reason, every access by a user to an object should be checked.
- *Enforce least privilege.* The principle of least privilege states that a subject should have access to the smallest number of objects necessary to perform some task. Even if extra information would be useless or harmless if the subject were to have access, the subject should not have that additional access. For example, a program should not have access to the absolute memory address to which a page number reference translates, even though the program could not use that address in any effective way. Not allowing access to unnecessary objects guards against security weaknesses if a part of the protection mechanism should fail.
- *Verify acceptable usage.* Ability to access is a yes-or-no decision. But it is equally important to check that the activity to be performed on an object is appropriate. For example, a data structure such as a stack has certain acceptable operations, including *push*, *pop*, *clear*, and so on. We may want not only to control who or what has access to a stack but also to be assured that the accesses performed are legitimate stack accesses.

In the next section we consider protection mechanisms appropriate for general objects of unspecified types, such as the kinds of objects listed above. To make the explanations easier to understand, we sometimes use an example of a specific object, such as a file. Note, however, that a general mechanism can be used to protect any of the types of objects listed.

Directory

One simple way to protect an object is to use a mechanism that works like a file directory. Imagine we are trying to protect files (the set of objects) from users of a computing system (the set of subjects). Every file has a unique owner who possesses "control"

access rights (including the rights to declare who has what access) and to revoke access to any person at any time. Each user has a file directory, which lists all the files to which that user has access.

Clearly, no user can be allowed to write in the file directory because that would be a way to forge access to a file. Therefore, the operating system must maintain all file directories, under commands from the owners of files. The obvious rights to files are the common *read*, *write*, and *execute* familiar on many shared systems. Furthermore, another right, *owner*, is possessed by the owner, permitting that user to grant and revoke access rights. Figure 4-10 shows an example of a file directory.

This approach is easy to implement because it uses one list per user, naming all the objects that user is allowed to access. However, several difficulties can arise. First, the list becomes too large if many shared objects, such as libraries of subprograms or a common table of users, are accessible to all users. The directory of each user must have one entry for each such shared object, even if the user has no intention of accessing the object. Deletion must be reflected in all directories. (See Sidebar 4-2 for a different issue concerning deletion of objects.)

A second difficulty is **revocation of access**. If owner *A* has passed to user *B* the right to read file *F*, an entry for *F* is made in the directory for *B*. This granting of access implies a level of *trust* between *A* and *B*. If *A* later questions that trust, *A* may

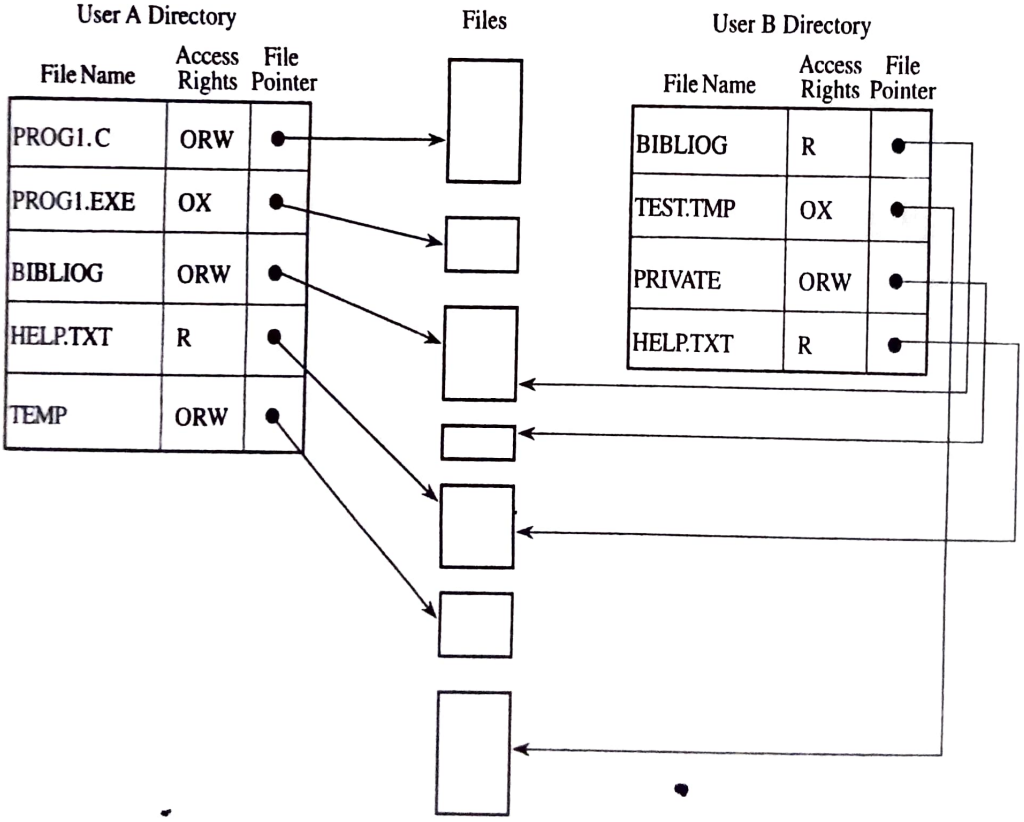

FIGURE 4-10 Directory Access.

Sidebar 4-2 "Out, Damned Spot! Out, I Say!"

Shakespeare's Lady Macbeth symbolically and obsessively sought to remove from her hands the blood of the man her husband had murdered at her instigation. As others have found (less dramatically), removing bits can be real, critical, and challenging.

Early Microsoft operating systems didn't actually erase a deleted file; they simply flagged its directory entry to show the file had been deleted. A user who accidentally deleted a file could recover the file by resetting the flag in the directory. Now Microsoft implements a recycle bin (originally an idea from Apple).

What happens to the bits of deleted files? In early multiuser operating systems, it was possible to retrieve someone else's data by looking through the trash. The technique was to create a large file, and then before writing to the file, read from it to get the contents previously written in that memory or storage space. Although an attacker had to restructure the data (blocks might not be stored contiguously, and a large space might include scraps of data from several other users), sensitive data could be found with some luck and some work. This flaw led to an operating system's enforcing "object reuse." The operating system had to ensure that no residue from a previous user was accessible by another. The operating system could erase (for example, overwrite with all 0s) all storage being assigned to a new user, or it could enforce a policy that a user could read from a space only after having written into it.

Magnetic devices retain some memory of what was written in an area. As President Nixon's secretary discovered with her 17-minute gap, with specialized equipment engineers can sometimes bring back something previously written and then written over. This property, called "magnetic remanence," causes organizations with sensitive data to require a seven- or more pass erasure, rewriting first with 0s, then with 1s, and then with a random pattern of 0s and 1s. And agencies with the most sensitive data opt to destroy the medium rather than risk inadvertent disclosure. Garfinkel and Shelat discuss sanitizing magnetic media in [GAR03a].

want to revoke the access right of *B*. The operating system can respond easily to the single request to delete the right of *B* to access *F* because that action involves deleting one entry from a specific directory. But if *A* wants to remove the rights of *everyone* to access *F*, the operating system must search each individual directory for the entry *F*, an activity that can be time consuming on a large system. For example, large timesharing systems or networks of smaller systems can easily have 5,000 to 10,000 active accounts. Moreover, *B* may have passed the access right for *F* to another user, so *A* may not know that *F*'s access exists and should be revoked. This problem is particularly serious in a network.

A third difficulty involves pseudonyms. Owners *A* and *B* may have two different files named *F*, and they may both want to allow access by *S*. Clearly, the directory for *S* cannot contain two entries under the same name for different files. Therefore, *S* has to be able to uniquely identify the *F* for *A* (or *B*). One approach is to include the original owner's designation as if it were part of the file name, with a notation such as *A*:*F* (or *B*:*F*).

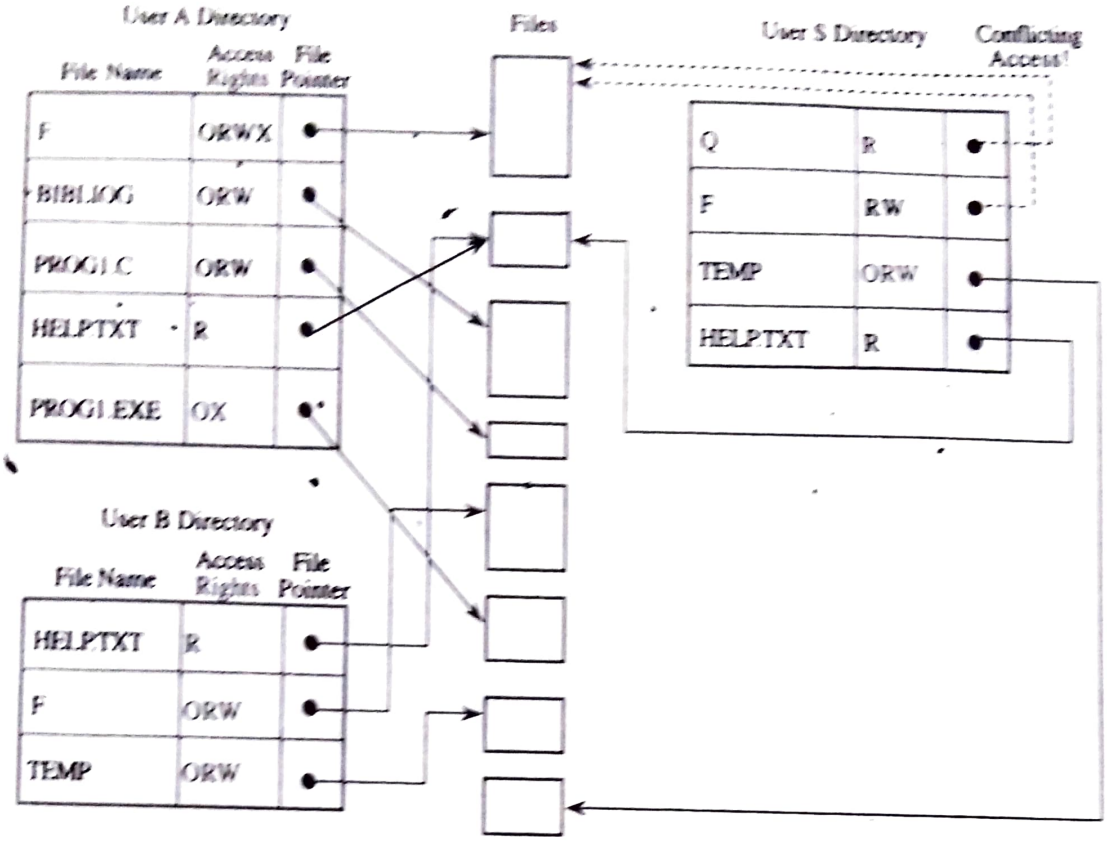

FIGURE 4-11 Alternative Access Paths.

Suppose, however, that *S* has trouble remembering file contents from the name *F*. Another approach is to allow *S* to name *F* with any name unique to the directory of *S*. Then, *F* from *A* could be called *Q* to *S*. As shown in Figure 4-11, *S* may have forgotten that *Q* is *F* from *A*, and so *S* requests access again from *A* for *F*. But by now *A* may have more trust in *S*, so *A* transfers *F* with greater rights than before. This action opens up the possibility that one subject, *S*, may have two distinct sets of access rights to *F*, one under the name *Q* and one under the name *F*. In this way, allowing pseudonyms leads to multiple permissions that are not necessarily consistent. Thus, the directory approach is probably too simple for most object protection situations.

Access Control List

An alternative representation is the access **control list**. There is one such list for each object, and the list shows all subjects who should have access to the object and what their access is. This approach differs from the directory list because there is one access control list per *object*; a directory is created for each *subject*. Although this difference seems small, there are some significant advantages.

To see how, consider subjects *A* and *S*, both of whom have access to object *F*. The operating system will maintain just one access list for *F*, showing the access rights for *A* and *S*, as shown in Figure 4-12. The access control list can include general default entries for any users. In this way, specific users can have explicit rights, and all other users can have a default set of rights. With this organization, a public file or program can be shared by all possible users of the system without the need for an entry for the object in the individual directory of each user.

The Multics operating system used a form of access control list in which each user belonged to three protection classes: a *user*, a *group*, and a *compartment*. The user designation identified a specific subject, and the group designation brought together subjects who had a common interest, such as coworkers on a project. The compartment confined an untrusted object; a program executing in one compartment could not access objects in another compartment without specific permission. The compartment was also a way to collect objects that were related, such as all files for a single project.

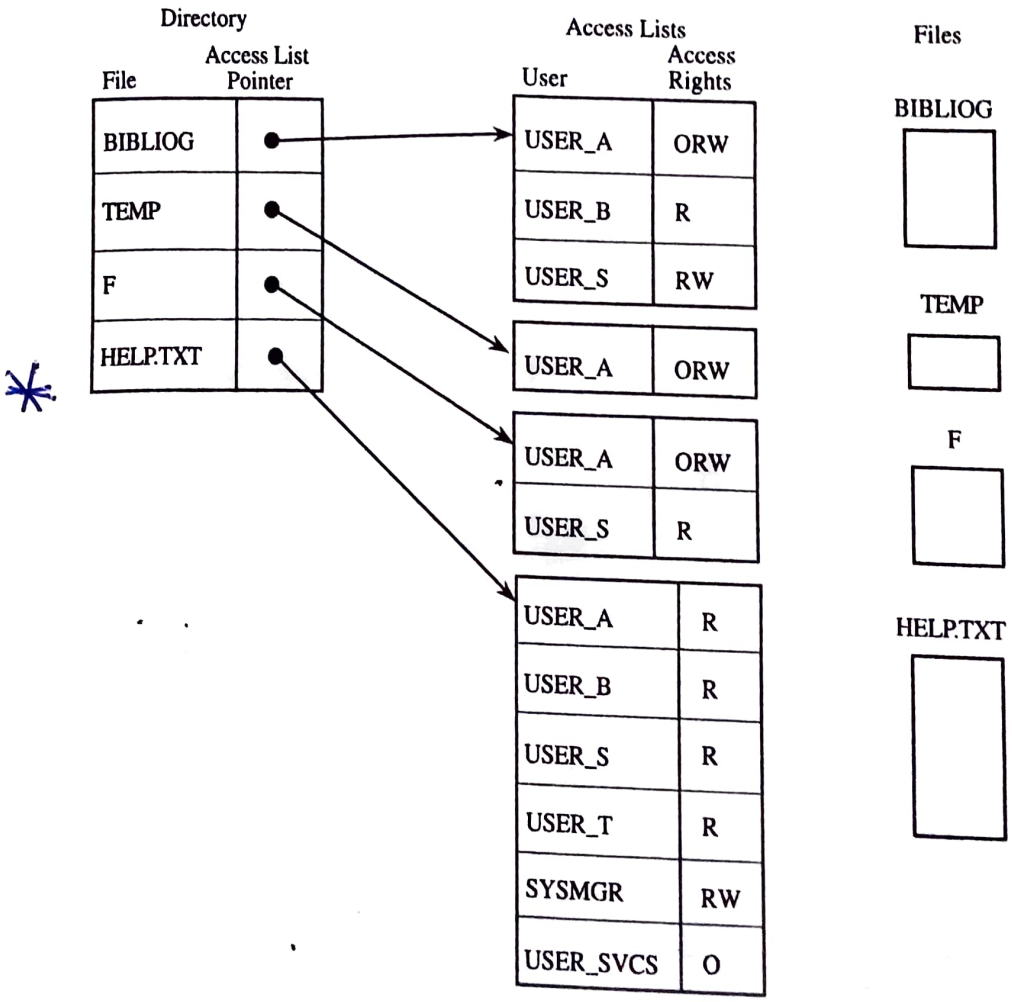

FIGURE 4-12 Access Control List.

To see how this type of protection might work, suppose every user who initiates access to the system identifies a group and a compartment with which to work. If Adams logs in as user *Adams* in group *Decl* and compartment *Art2*, only objects having *Adams-Decl-Art2* in the access control list are accessible in the session.

By itself, this kind of mechanism would be too restrictive to be usable. Adams cannot create general files to be used in any session. Worse yet, shared objects would have not only to list Adams as a legitimate subject but also to list Adams under all acceptable groups and all acceptable compartments for each group.

The solution is the use of **wild cards**, meaning placeholders that designate "any user" (or "any group" or "any compartment"). An access control list might specify access by *Adams-Decl-Art1*, giving specific rights to Adams if working in group *Decl* on compartment *Art1*. The list might also specify *Adams-*-Art1*, meaning that Adams can access the object from any group in compartment *Art1*. Likewise, a notation of **-Decl-** would mean "any user in group *Decl* in any compartment." Different placements of the wildcard notation * have the obvious interpretations.

The access control list can be maintained in sorted order, with * sorted as coming after all specific names. For example, *Adams-Decl-** would come after all specific compartment designations for Adams. The search for access permission continues just until the first match. In the protocol, all explicit designations are checked before wild cards in any position, so a specific access right would take precedence over a wildcard right. The last entry on an access list could be *-*-*, specifying rights allowable to any user not explicitly on the access list. By using this wildcard device, a shared public object can have a very short access list, explicitly naming the few subjects that should have access rights different from the default.

Access Control Matrix

We can think of the directory as a listing of objects accessible by a single subject, and the access list as a table identifying subjects that can access a single object. The data in these two representations are equivalent, the distinction being the ease of use in given situations.

As an alternative, we can use an **access control matrix**, a table in which each row represents a subject, each column represents an object, and each entry is the set of access rights for that subject to that object. An example representation of an access control matrix is shown in Table 4-1. In general, the access control matrix is sparse (meaning that most cells are empty): Most subjects do not have access rights to most objects. The access matrix can be represented as a list of triples, having the form ⟨*subject, object, rights*⟩. Searching a large number of these triples is inefficient enough that this implementation is seldom used.

Capability

So far, we have examined protection schemes in which the operating system must keep track of all the protection objects and rights. But other approaches put some of the burden on the user. For example, a user may be required to have a ticket or pass that enables access, much like a ticket or identification card that cannot be duplicated. More formally, we say that a **capability** is an unforgeable token that gives the possessor

TABLE 4-1 Access Control Matrix

	BIBLIOG	TEMP	F	HELPTXT	C_COMP	LINKER	SYS_CLOCK	PRINTER
USER A	ORW	ORW	ORW	R	X	X	R	W
USER B	R			R	X	X	R	W
USER S	RW		R	R	X	X	R	W
USER T				R	X	X	R	W
SYS_MGR				RW	OX	OX	ORW	O
USER_SVCS				O	X	X	R	W

certain rights to an object. The Multics [SAL74], CAL [LAM76], and Hydra [WUL74] systems used capabilities for access control. In theory, a subject can create new objects and can specify the operations allowed on those objects. For example, users can create objects, such as files, data segments, or subprocesses, and can also specify the acceptable kinds of operations, such as *read*, *write*, and *execute*. But a user can also create completely new objects, such as new data structures, and can define types of accesses previously unknown to the system.

A capability is a ticket giving permission to a subject to have a certain type of access to an object. For the capability to offer solid protection, the ticket must be unforgeable. One way to make it unforgeable is to not give the ticket directly to the user. Instead, the operating system holds all tickets on behalf of the users. The operating system returns to the user a pointer to an operating system data structure, which also links to the user. A capability can be created only by a specific request from a user to the operating system. Each capability also identifies the allowable accesses.

Alternatively, capabilities can be encrypted under a key available only to the access control mechanism. If the encrypted capability contains the identity of its rightful owner, user *A* cannot copy the capability and give it to user *B*.

One possible access right to an object is *transfer* or *propagate*. A subject having this right can pass copies of capabilities to other subjects. In turn, each of these capabilities also has a list of permitted types of accesses, one of which might also be *transfer*. In this instance, process *A* can pass a copy of a capability to *B*, who can then pass a copy to *C*. *B* can prevent further distribution of the capability (and therefore prevent further dissemination of the access right) by omitting the *transfer* right from the rights passed in the capability to *C*. *B* might still pass certain access rights to *C*, but not the right to propagate access rights to other subjects.

As a process executes, it operates in a **domain** or **local name space**. The domain is the collection of objects to which the process has access. A domain for a user at a given time might include some programs, files, data segments, and I/O devices such as a printer and a terminal. An example of a domain is shown in Figure 4-13.

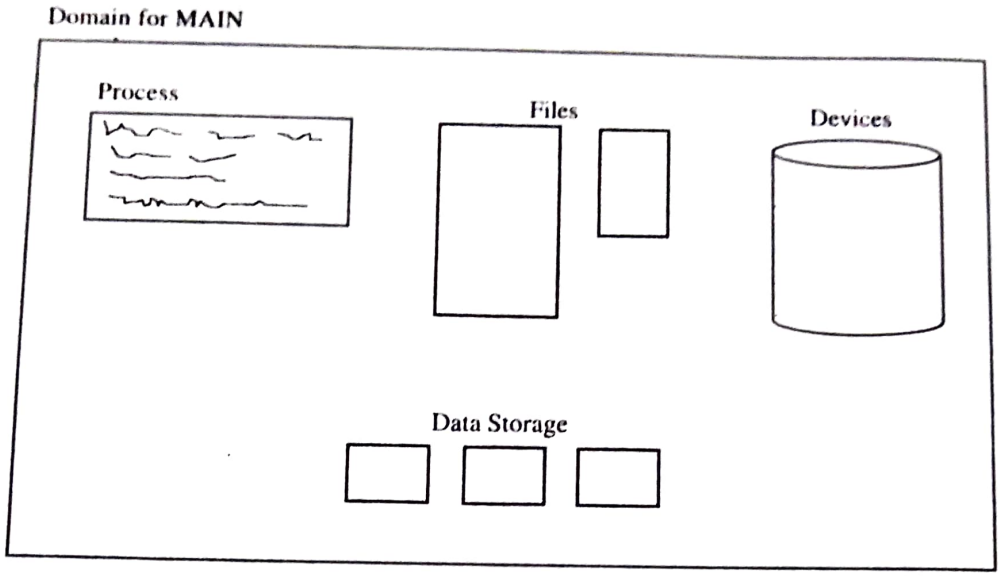

FIGURE 4-13 Process Execution Domain.

As execution continues, the process may call a subprocedure, passing some of the objects to which it has access as arguments to the subprocedure. The domain of the subprocedure is not necessarily the same as that of its calling procedure; in fact, a calling procedure may pass only some of its objects to the subprocedure, and the subprocedure may have access rights to other objects not accessible to the calling procedure. The caller may also pass only some of its access rights for the objects it passes to the subprocedure. For example, a procedure might pass to a subprocedure the right to read but not modify a particular data value.

Because each capability identifies a single object in a domain, the collection of capabilities defines the domain. When a process calls a subprocedure and passes certain objects to the subprocedure, the operating system forms a stack of all the capabilities of the current procedure. The operating system then creates new capabilities for the subprocedure, as shown in Figure 4-14.

Operationally, capabilities are a straightforward way to keep track of the access rights of subjects to objects during execution. The capabilities are backed up by a more comprehensive table, such as an access control matrix or an access control list. Each time a process seeks to use a new object, the operating system examines the master list of objects and subjects to determine whether the object is accessible. If so, the operating system creates a capability for that object.

Capabilities must be stored in memory inaccessible to normal users. One way of accomplishing this is to store capabilities in segments not pointed at by the user's segment table or to enclose them in protected memory as from a pair of base/bounds registers. Another approach is to use a tagged architecture machine to identify capabilities as structures requiring protection.

During execution, only the capabilities of objects that have been accessed by the current process are kept readily available. This restriction improves the speed with

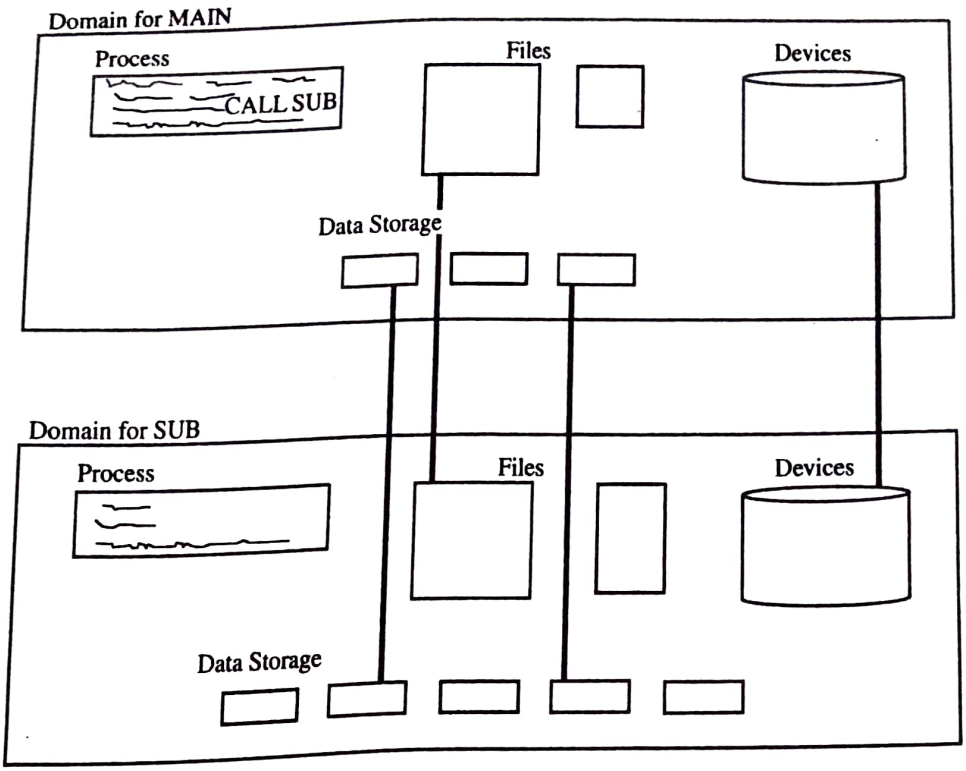

FIGURE 4-14 Passing Objects to a Subject.

which access to an object can be checked. This approach is essentially the one used in Multics, as described in [FAB74].

Capabilities can be revoked. When an issuing subject revokes a capability, no further access under the revoked capability should be permitted. A capability table can contain pointers to the active capabilities spawned under it so that the operating system can trace what access rights should be deleted if a capability is revoked. A similar problem is deleting capabilities for users who are no longer active.

Kerberos

Fundamental research on capabilities laid the groundwork for subsequent production use in systems such as Kerberos [STE88] (studied in greater detail in Chapter 7). Kerberos implements both authentication and access authorization by means of capabilities, called **tickets**, secured with symmetric cryptography. Microsoft has based much of its access control in NT+ on Kerberos.

Kerberos requires two systems, called the **authentication server** (AS) and the **ticket-granting server** (TGS), which are both part of the **key distribution center** (KDC). A user presents an authenticating credential (such as a password) to the authentication server and receives a ticket showing that the user has passed authentication. Obviously, the ticket must be encrypted to prevent the user from modifying or

forging one claiming to be a different user, and the ticket must contain some provision to prevent one user from acquiring another user's ticket to impersonate that user.

Now let us suppose that a user, Joe, wants to access a resource R (for example, a file, printer, or network port). Joe sends the TGS his authenticated ticket and a request to use R. Assuming Joe is allowed access, the TGS returns to Joe two tickets: One shows Joe that his access to R has been authorized, and the second is for Joe to present to R in order to access R.

Kerberos implements **single sign-on**; that is, a user signs on once and from that point on all the user's (allowable) actions are authorized without the user needing to sign on again. So if a user wants access to a resource in a different domain, say on a different system or in a different environment or even a different company or institution, as long as authorization rights have been established between the two domains, the user's access takes place without the user's signing on to a different system.

Kerberos accomplishes its local and remote authentication and authorization with a system of shared secret encryption keys. In fact, each user's password is used as an encryption key. (That trick also means that passwords are never exposed, reducing the risk from interception.) We study the exact mechanism of Kerberos in Chapter 7.

Procedure-Oriented Access Control

One goal of access control is restricting not just which subjects have access to an object, but also what they can *do* to that object. Read versus write access can be controlled rather readily by most operating systems, but more complex control is not so easy to achieve.

By **procedure-oriented** protection, we imply the existence of a procedure that controls access to objects (for example, by performing its own user authentication to strengthen the basic protection provided by the basic operating system). In essence, the procedure forms a capsule around the object, permitting only certain specified accesses.

Procedures can ensure that accesses to an object be made through a trusted interface. For example, neither users nor general operating system routines might be allowed direct access to the table of valid users. Instead, the only accesses allowed might be through three procedures: one to add a user, one to delete a user, and one to check whether a particular name corresponds to a valid user. These procedures, especially add and delete, could use their own checks to make sure that calls to them are legitimate.

Procedure-oriented protection implements the principle of information hiding because the means of implementing an object are known only to the object's control procedure. Of course, this degree of protection carries a penalty of inefficiency. With procedure-oriented protection, there can be no simple, fast access, even if the object is frequently used.

Our survey of access control mechanisms has intentionally progressed from simple to complex. Historically, as the mechanisms have provided greater flexibility, they have done so with a price of increased overhead. For example, implementing capabilities that must be checked on each access is far more difficult than implementing a simple directory structure that is checked only on a subject's first access to an object. This

complexity is apparent both to the user and to the implementer. The user is aware of additional protection features, but the naïve user may be frustrated or intimidated at having to select protection options with little understanding of their usefulness. The implementation complexity becomes apparent in slow response to users. The balance between simplicity and functionality is a continuing battle in security.

Role-Based Access Control

We have not yet distinguished among kinds of users, but we want some users (such as administrators) to have significant privileges, and we want others (such as regular users or guests) to have lower privileges. In companies and educational institutions, this can get complicated when an ordinary user becomes an administrator or a baker moves to the candlestick makers' group. **Role-based access control** lets us associate privileges with groups, such as all administrators can do this or candlestick makers are forbidden to do this. Administering security is easier if we can control access by job demands, not by person. Access control keeps up with a person who changes responsibilities, and the system administrator does not have to choose the appropriate access control settings for someone. For more details on the nuances of role-based access control, see [FER03].

4.4 FILE PROTECTION MECHANISMS

Until now, we have examined approaches to protecting a general object, no matter the object's nature or type. But some protection schemes are particular to the type. To see how they work, we focus in this section on file protection. The examples we present are only representative; they do not cover all possible means of file protection on the market.

Basic Forms of Protection

We noted earlier that all multiuser operating systems must provide some minimal protection to keep one user from maliciously or inadvertently accessing or modifying the files of another. As the number of users has grown, so also has the complexity of these protection schemes.

All–None Protection

In the original IBM OS operating systems, files were by default public. Any user could read, modify, or delete a file belonging to any other user. Instead of software- or hardware-based protection, the principal protection involved trust combined with ignorance. System designers supposed that users could be trusted not to read or modify others' files because the users would expect the same respect from others. Ignorance helped this situation, because a user could access a file only by name; presumably users knew the names only of those files to which they had legitimate access.

However, it was acknowledged that certain system files were sensitive and that the system administrator could protect them with a password. A normal user could exercise this feature, but passwords were viewed as most valuable for protecting operating