4.2. Memory and Address Protection

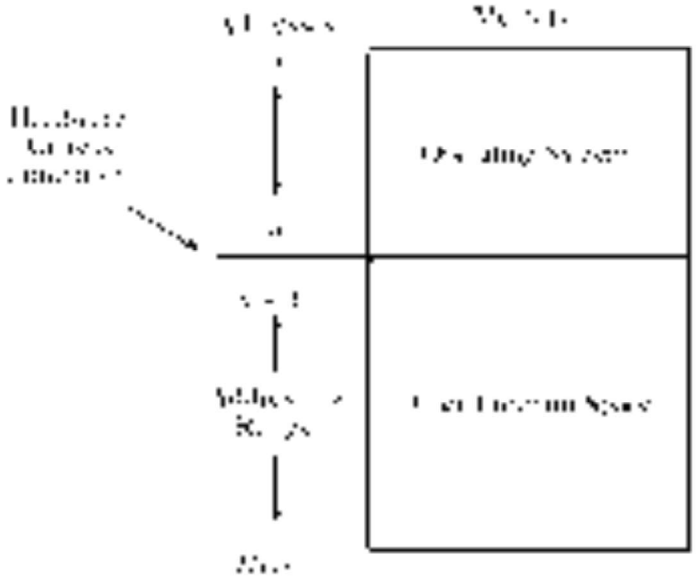
The most obvious problem of multiprogramming is preventing one program from affecting the data and programs in the memory space of other users. Fortunately, protection can be built into the hardware mechanisms that control efficient use of memory, so solid protection can be provided at essentially no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

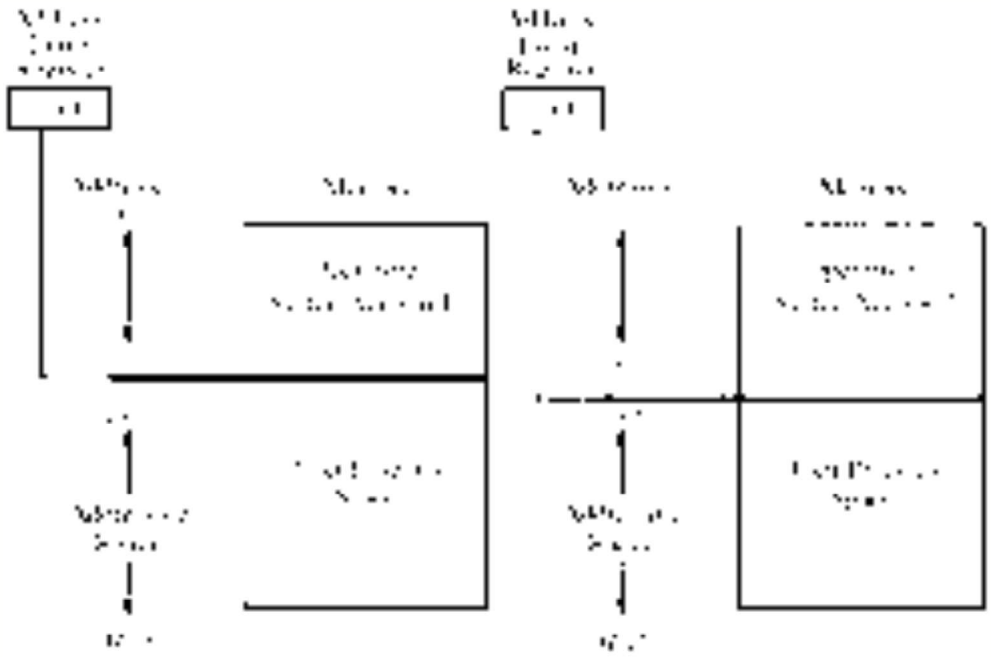
In one implementation, the fence was a predefined memory address, enabling the operating system to reside on one side and the user to stay on the other. An example of this situation is shown in Figure 4-1. Unfortunately, this kind of implementation was very restrictive because a predefined amount of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

Figure 4-1. Fixed Fence.



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.

Figure 4-2. Variable Fence Register.



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.

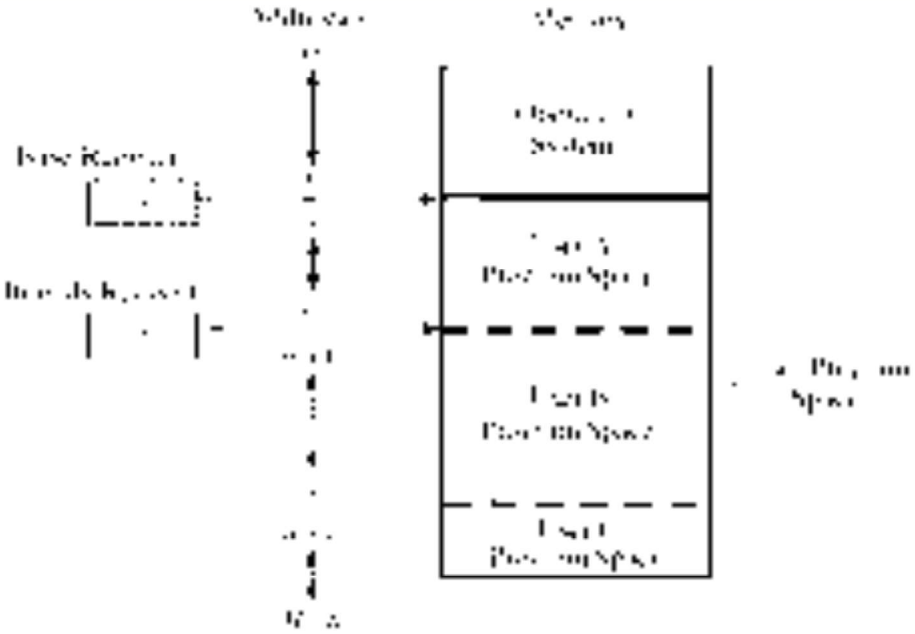
Base/Bounds Registers

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.

Figure 4-3. Pair of Base/Bounds Registers.



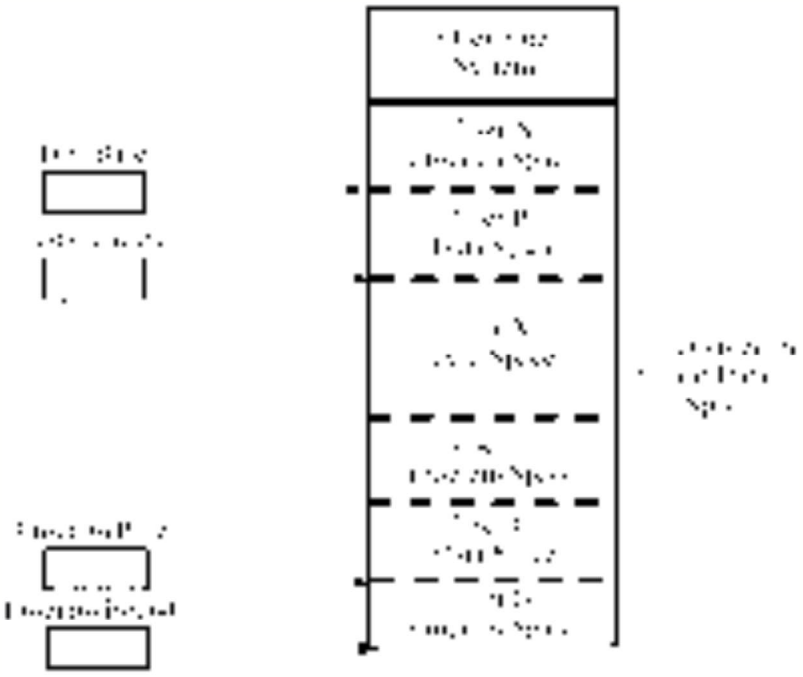
This technique protects a program's addresses from modification by another user. When execution changes from one user's program to another's, the operating system must change the contents of the base and bounds registers to reflect the true address space for that user. This change is part of the general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

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



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.

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.

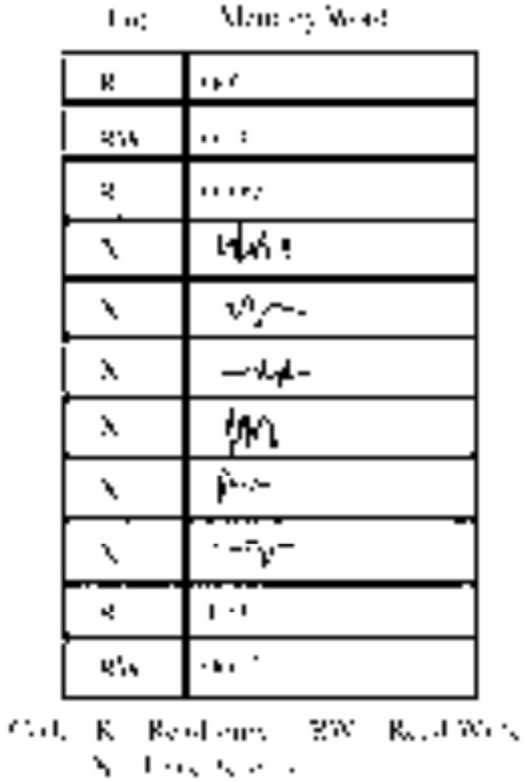
Additional, operating-system-specific design features can help, too. Base/bounds registers create an all-or-nothing situation for sharing: Either a program makes all its data available to be accessed and modified or it prohibits access to all. Even if there were a third set of registers for shared data, 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 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.

Figure 4-5. Example of Tagged Architecture.



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

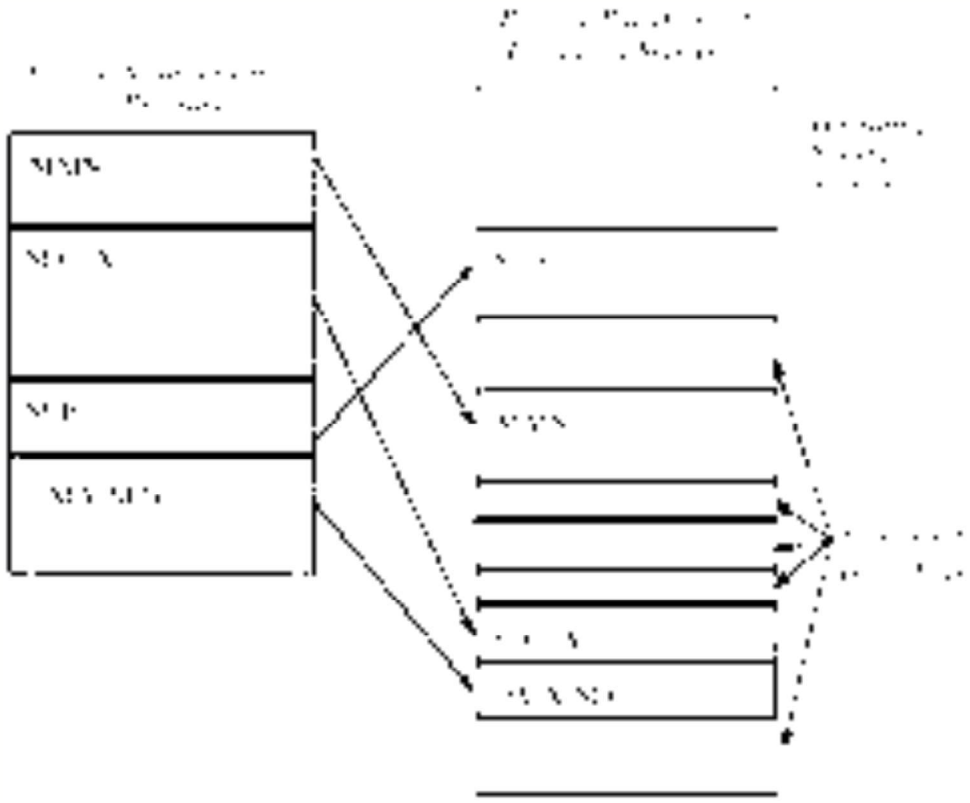
We present two more approaches to protection, each of which can be implemented on top of a conventional machine structure, suggesting a better chance of acceptance. Although these approaches are ancient by computing's standardsthey were designed between 1965 and 1975they have been implemented on many 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).

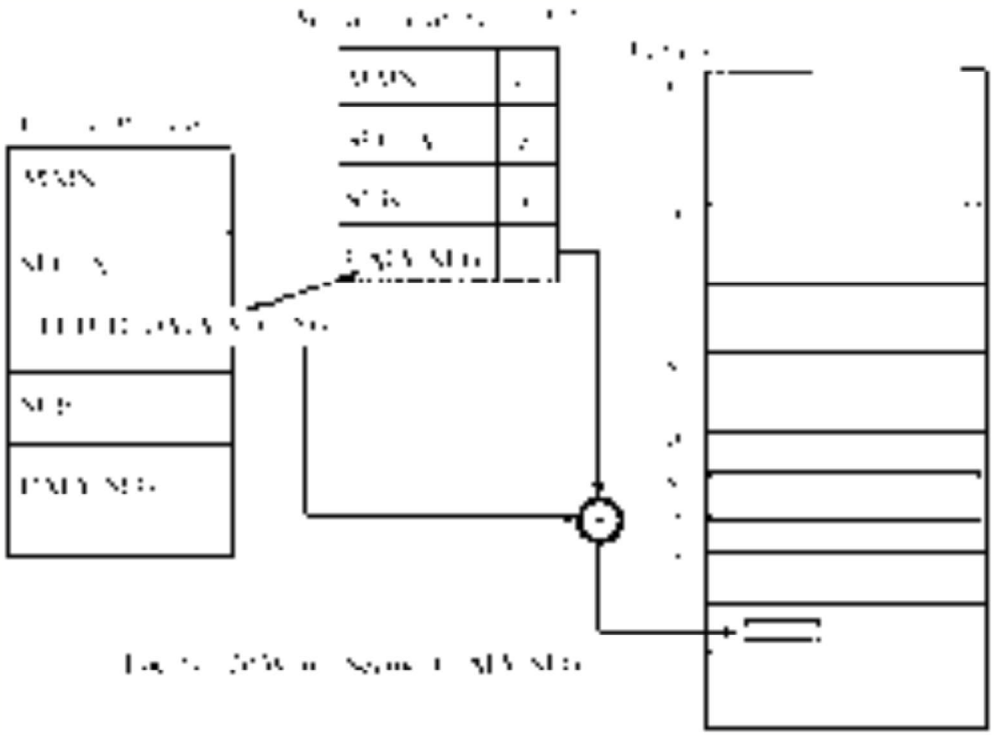
Logically, the programmer pictures a program as a long collection of segments. Segments can be separately relocated, allowing any segment to be placed in any available memory locations. The relationship between a logical segment and its true memory position is shown in Figure 4-6.

Figure 4-6. Logical and Physical Representation of Segments.



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.

Figure 4-7. Translation of Segment Address.



Thus, a user's program does not know what true memory addresses it uses. It has no wayand no needto 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.

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

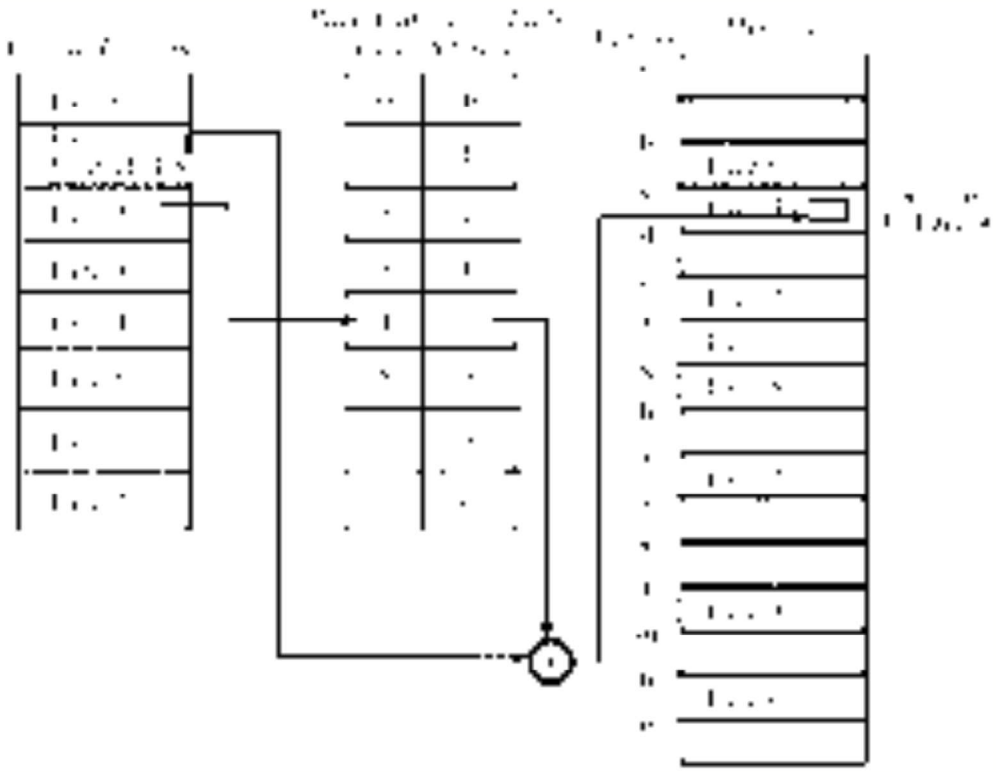
Thus, we need to balance protection with efficiency, finding ways to keep segmentation as efficient as possible. However, efficient implementation of segmentation presents two problems: Segment names are inconvenient to encode in instructions, and the operating system's lookup of the name in a table 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.

Figure 4-8. Page Address Translation.



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.

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

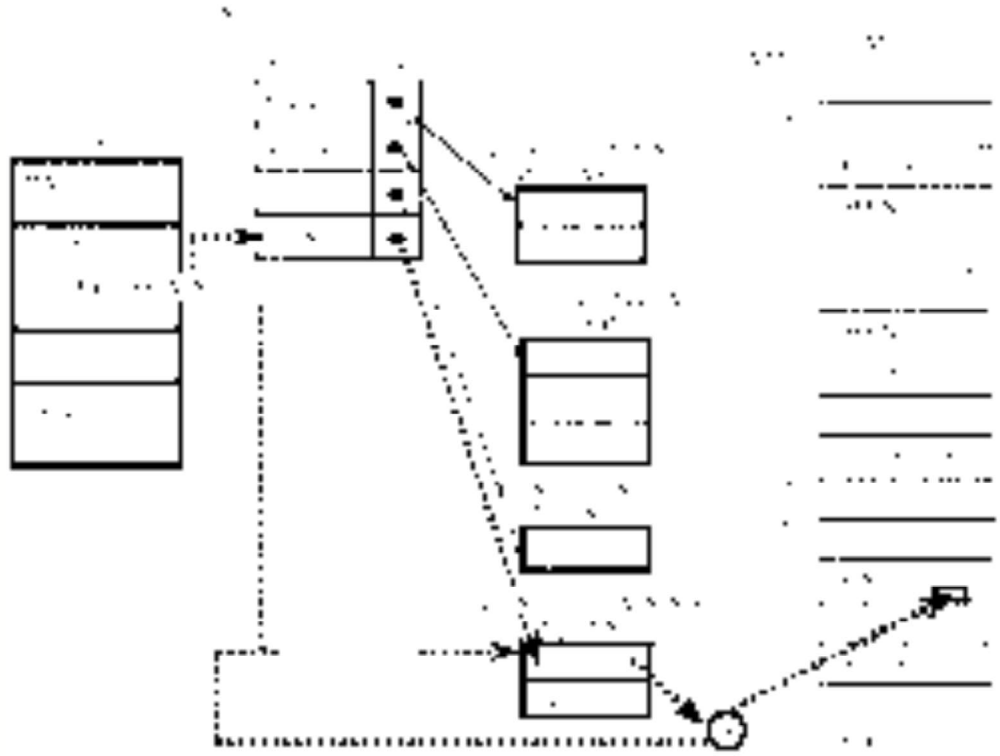
We have seen how paging offers implementation efficiency, while segmentation offers logical

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.

[View full size image]



PREV NEXT