PART 3 Memory

CHAPTER 7

MEMORY MANAGEMENT

Learning Objectives

After studying this chapter, you should be able to:

- Discuss the principal requirements for memory management.
- Understand the reason for memory partitioning and explain the various techniques that are used.
- Understand and explain the concept of paging.
- Understand and explain the concept of segmentation.
- Assess the relative advantages of paging and segmentation.
- Describe the concepts of loading and linking.

In a uniprogramming system, main memory is divided into two parts: one part for the operating system (resident monitor, kernel) and other part for the program currently being executed. In a multiprogramming system, the "user" part of memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as **memory management**.

Effective memory management is vital in a multiprogramming system. If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O (input/output), and the processor will be idle. Thus, memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time.

We begin with the requirements that memory management is intended to satisfy. Next, we will discuss a variety of simple schemes that have been used for memory management.

Table 7.1 introduces some key terms for our discussion.

7.1 MEMORY MANAGEMENT REQUIREMENTS

While surveying the various mechanisms and policies associated with memory management, it is helpful to keep in mind the requirements that memory management is intended to satisfy. These requirements include the following:

- Relocation
- Protection

Table 7.1 Memory Management Terms

Frame	A fixed-length block of main memory.
Page	A fixed-length block of data that resides in secondary memory (such as a disk). A page of data may temporarily be copied into a frame of main memory.
Segment	A variable-length block of data that resides in secondary memory. An entire segment may temporarily be copied into an available region of main memory (segmentation) or the segment may be divided into pages, which can be individually copied into main memory (combined segmentation and paging).

- Sharing
- Logical organization
- Physical organization

Relocation

In a multiprogramming system, the available main memory is generally shared among a number of processes. Typically, it is not possible for the programmer to know in advance which other programs will be resident in main memory at the time of execution of his or her program. In addition, we would like to be able to swap active processes in and out of main memory to maximize processor utilization by providing a large pool of ready processes to execute. Once a program is swapped out to disk, it would be quite limiting to specify that when it is next swapped back in, it must be placed in the same main memory region as before. Instead, we may need to **relocate** the process to a different area of memory.

Thus, we cannot know ahead of time where a program will be placed, and we must allow for the possibility that the program may be moved about in main memory due to swapping. These facts raise some technical concerns related to addressing, as illustrated in Figure 7.1. The figure depicts a process image. For simplicity, let us assume that the process image occupies a contiguous region of main memory. Clearly, the operating system will need to know the location of process control information and of the execution stack, as well as the entry point to begin execution of the program for this process. Because the operating system is managing memory and is responsible for bringing this process into main memory, these addresses are easy to come by. In addition, however, the processor must deal with memory references

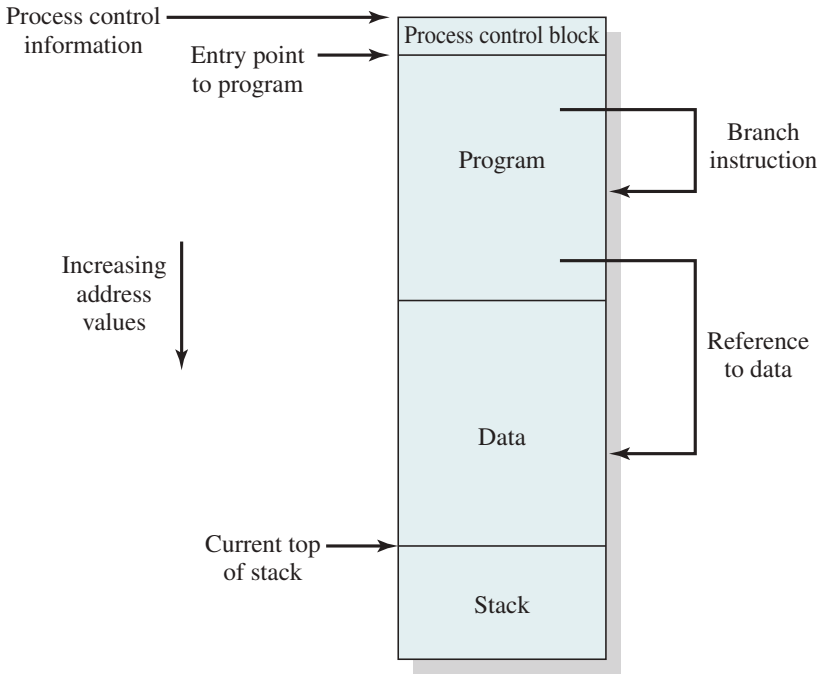


Figure 7.1 Addressing Requirements for a Process

within the program. Branch instructions contain an address to reference the instruction to be executed next. Data reference instructions contain the address of the byte or word of data referenced. Somehow, the processor hardware and operating system software must be able to translate the memory references found in the code of the program into actual physical memory addresses, reflecting the current location of the program in main memory.

Protection

Each process should be protected against unwanted interference by other processes, whether accidental or intentional. Thus, programs in other processes should not be able to reference memory locations in a process for reading or writing purposes without permission. In one sense, satisfaction of the relocation requirement increases the difficulty of satisfying the protection requirement. Because the location of a program in main memory is unpredictable, it is impossible to check absolute addresses at compile time to assure protection. Furthermore, most programming languages allow the dynamic calculation of addresses at run time (e.g., by computing an array subscript or a pointer into a data structure). Hence, all memory references generated by a process must be checked at run time to ensure they refer only to the memory space allocated to that process. Fortunately, we shall see that mechanisms that support relocation also support the protection requirement.

Normally, a user process cannot access any portion of the operating system, neither program nor data. Again, usually a program in one process cannot branch to an instruction in another process. Without special arrangement, a program in one process cannot access the data area of another process. The processor must be able to abort such instructions at the point of execution.

Note the memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software). This is because the OS cannot anticipate all of the memory references that a program will make. Even if such anticipation were possible, it would be prohibitively time consuming to screen each program in advance for possible memory-reference violations. Thus, it is only possible to assess the permissibility of a memory reference (data access or branch) at the time of execution of the instruction making the reference. To accomplish this, the processor hardware must have that capability.

Sharing

Any protection mechanism must have the flexibility to allow several processes to access the same portion of main memory. For example, if a number of processes are executing the same program, it is advantageous to allow each process to access the same copy of the program, rather than have its own separate copy. Processes that are cooperating on some task may need to share access to the same data structure. The memory management system must therefore allow controlled access to shared areas of memory without compromising essential protection. Again, we will see that the mechanisms used to support relocation also support sharing capabilities.

Logical Organization

Almost invariably, main memory in a computer system is organized as a linear or one-dimensional address space, consisting of a sequence of bytes or words. Secondary memory, at its physical level, is similarly organized. While this organization closely mirrors the actual machine hardware, it does not correspond to the way in which programs are typically constructed. Most programs are organized into modules, some of which are unmodifiable (read only, execute only) and some of which contain data that may be modified. If the operating system and computer hardware can effectively deal with user programs and data in the form of modules of some sort, then a number of advantages can be realized:

1. Modules can be written and compiled independently, with all references from one module to another resolved by the system at run time.
2. With modest additional overhead, different degrees of protection (read only, execute only) can be given to different modules.
3. It is possible to introduce mechanisms by which modules can be shared among processes. The advantage of providing sharing on a module level is that this corresponds to the user's way of viewing the problem, hence it is easy for the user to specify the sharing that is desired.

The tool that most readily satisfies these requirements is segmentation, which is one of the memory management techniques explored in this chapter.

Physical Organization

As we discussed in Section 1.5, computer memory is organized into at least two levels, referred to as main memory and secondary memory. Main memory provides fast access at relatively high cost. In addition, main memory is volatile; that is, it does not provide permanent storage. Secondary memory is slower and cheaper than main memory, but is usually not volatile. Thus, secondary memory of large capacity can be provided for long-term storage of programs and data, while a smaller main memory holds programs and data currently in use.

In this two-level scheme, the organization of the flow of information between main and secondary memory is a major system concern. The responsibility for this flow could be assigned to the individual programmer, but this is impractical and undesirable for two reasons:

1. The main memory available for a program and its data may be insufficient. In that case, the programmer must engage in a practice known as **overlaying**, in which the program and data are organized in such a way that various modules can be assigned the same region of memory, with a main program responsible for switching the modules in and out as needed. Even with the aid of compiler tools, overlay programming wastes programmer time.
2. In a multiprogramming environment, the programmer does not know at the time of coding how much space will be available or where that space will be.

It is clear, then, that the task of moving information between the two levels of memory should be a system responsibility. This task is the essence of memory management.

7.2 MEMORY PARTITIONING

The principal operation of memory management is to bring processes into main memory for execution by the processor. In almost all modern multiprogramming systems, this involves a sophisticated scheme known as virtual memory. Virtual memory is, in turn, based on the use of one or both of two basic techniques: segmentation and paging. Before we can look at these virtual memory techniques, we must prepare the ground by looking at simpler techniques that do not involve virtual memory (Table 7.2 summarizes all the techniques examined in this chapter and the next). One of these techniques, partitioning, has been used in several variations in some now-obsolete operating systems. The other two techniques, simple paging and simple segmentation, are not used by themselves. However, it will clarify the discussion of virtual memory if we look first at these two techniques in the absence of virtual memory considerations.

Fixed Partitioning

In most schemes for memory management, we can assume the OS occupies some fixed portion of main memory, and the rest of main memory is available for use by multiple processes. The simplest scheme for managing this available memory is to partition it into regions with fixed boundaries.

Partition Sizes Figure 7.2 shows examples of two alternatives for fixed partitioning. One possibility is to make use of equal-size partitions. In this case, any process whose size is less than or equal to the partition size can be loaded into any available partition. If all partitions are full, and no process is in the Ready or Running state, the operating system can swap a process out of any of the partitions and load in another process, so there is some work for the processor.

There are two difficulties with the use of equal-size fixed partitions:

- A program may be too big to fit into a partition. In this case, the programmer must design the program with the use of overlays so only a portion of the program need be in main memory at any one time. When a module is needed that is not present, the user's program must load that module into the program's partition, overlaying whatever programs or data are there.
- Main memory utilization is extremely inefficient. Any program, no matter how small, occupies an entire partition. In our example, there may be a program whose length is less than 2 Mbytes; yet it occupies an 8-Mbyte partition whenever it is swapped in. This phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition, is referred to as **internal fragmentation**.

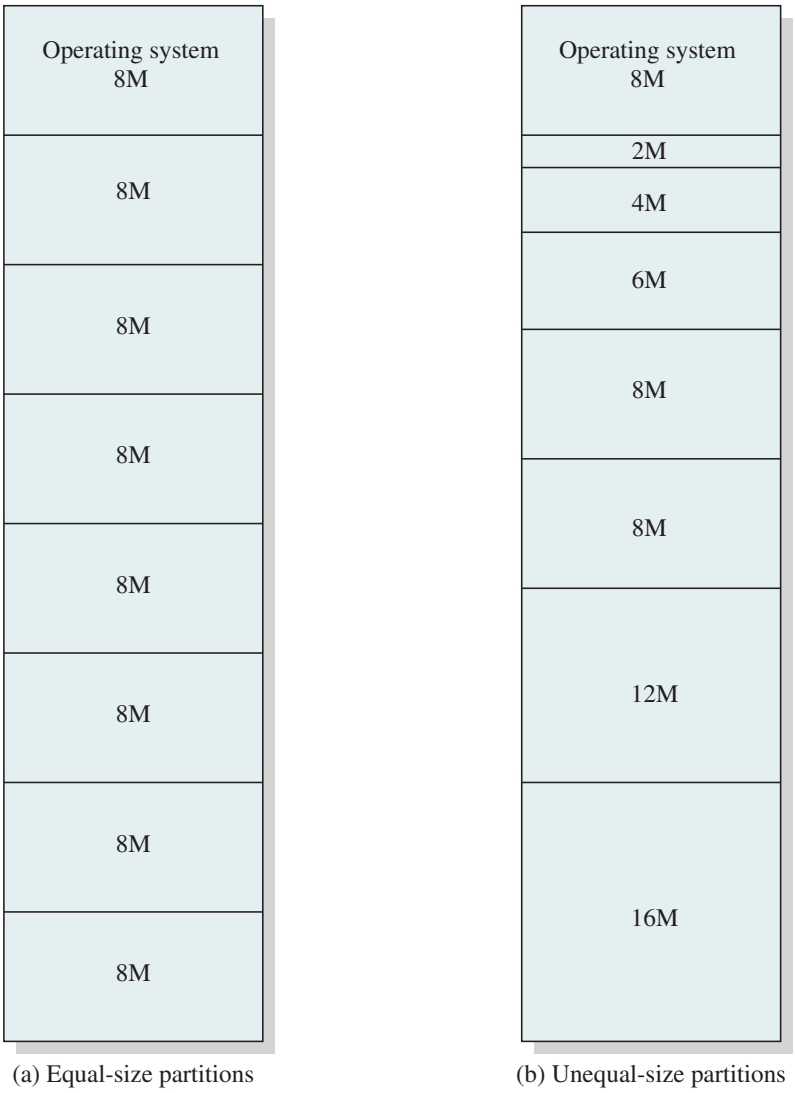
Both of these problems can be lessened, though not solved, by using unequal-size partitions (see Figure 7.2b). In this example, programs as large as 16 Mbytes can

Table 7.2 Memory Management Techniques

Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation.
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are automatically brought in later.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are automatically brought in later.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.

be accommodated without overlays. Partitions smaller than 8 Mbytes allow smaller programs to be accommodated with less internal fragmentation.

Placement Algorithm With equal-size partitions, the placement of processes in memory is trivial. As long as there is any available partition, a process can be loaded into that partition. Because all partitions are of equal size, it does not matter which partition is used. If all partitions are occupied with processes that are not ready



(a) Equal-size partitions (b) Unequal-size partitions

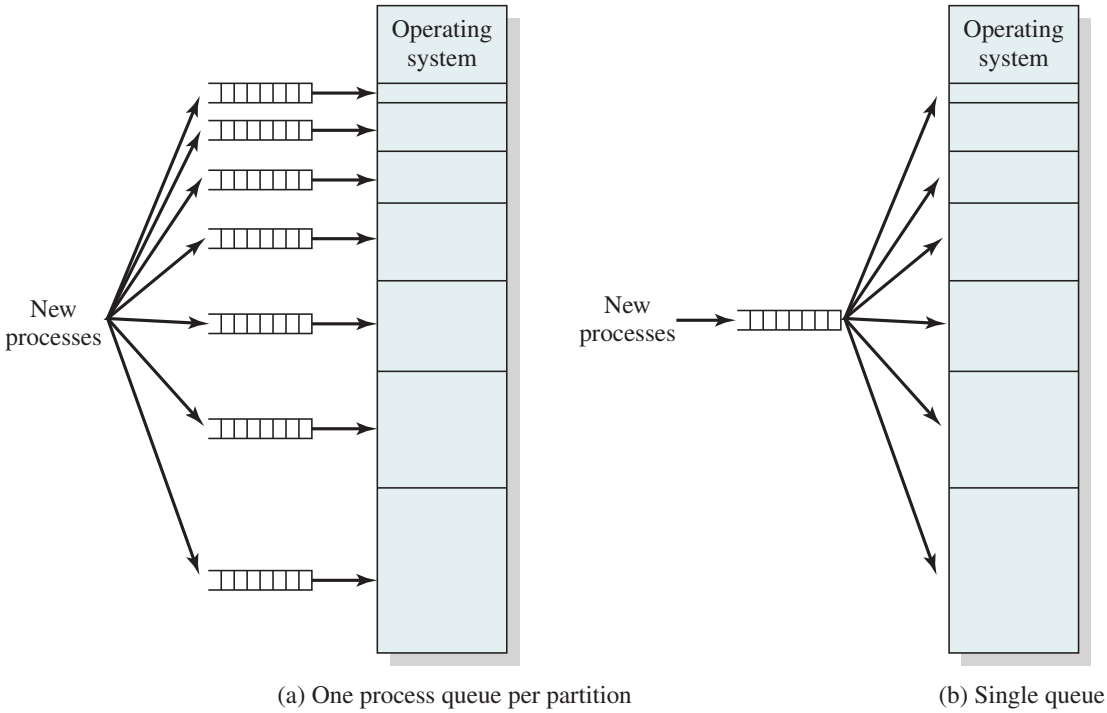
Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

to run, then one of these processes must be swapped out to make room for a new process. Which one to swap out is a scheduling decision; this topic will be explored in Part Four.

With unequal-size partitions, there are two possible ways to assign processes to partitions. The simplest way is to assign each process to the smallest partition within which it will fit.[1] In this case, a scheduling queue is needed for each partition to hold swapped-out processes destined for that partition (see Figure 7.3a). The advantage of this approach is that processes are always assigned in such a way as to minimize wasted memory within a partition (internal fragmentation).

Although this technique seems optimum from the point of view of an individual partition, it is not optimum from the point of view of the system as a whole.

[1]This assumes one knows the maximum amount of memory that a process will require. This is not always the case. If it is not known how large a process may become, the only alternatives are an overlay scheme or the use of virtual memory.



(a) One process queue per partition (b) Single queue

Figure 7.3 Memory Assignment for Fixed Partitioning

In Figure 7.2b, for example, consider a case in which there are no processes with a size between 12 and 16M at a certain point in time. In that case, the 16M partition will remain unused, even though some smaller process could have been assigned to it. Thus, a preferable approach would be to employ a single queue for all processes (see Figure 7.3b). When it is time to load a process into main memory, the smallest available partition that will hold the process is selected. If all partitions are occupied, then a swapping decision must be made. Preference might be given to swapping out of the smallest partition that will hold the incoming process. It is also possible to consider other factors, such as priority, and a preference for swapping out blocked processes versus ready processes.

The use of unequal-size partitions provides a degree of flexibility to fixed partitioning. In addition, it can be said that fixed partitioning schemes are relatively simple and require minimal OS software and processing overhead. However, there are disadvantages:

- The number of partitions specified at system generation time limits the number of active (not suspended) processes in the system.
- Because partition sizes are preset at system generation time, small jobs will not utilize partition space efficiently. In an environment where the main storage requirement of all jobs is known beforehand, this may be reasonable, but in most cases, it is an inefficient technique.

The use of fixed partitioning is almost unknown today. One example of a successful operating system that did use this technique was an early IBM mainframe operating system, OS/MFT (Multiprogramming with a Fixed Number of Tasks).

Dynamic Partitioning

To overcome some of the difficulties with fixed partitioning, an approach known as dynamic partitioning was developed. Again, this approach has been supplanted by more sophisticated memory management techniques. An important operating system that used this technique was IBM's mainframe operating system, OS/MVT (Multiprogramming with a Variable Number of Tasks).

With dynamic partitioning, the partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more. An example, using 64 Mbytes of main memory, is shown in Figure 7.4. Initially, main memory is empty, except for the OS (see Figure 7.4a). The first three processes are loaded in, starting where the operating system ends and occupying just enough space for each process (see Figure 7.4b, c, d). This leaves a "hole" at the end of memory that is too small for a fourth process. At some point, none of the processes in memory is ready. The operating system swaps out process 2 (see Figure 7.4e), which leaves sufficient room to load a new process, process 4 (see Figure 7.4f). Because process 4 is smaller than process 2, another small hole is created.

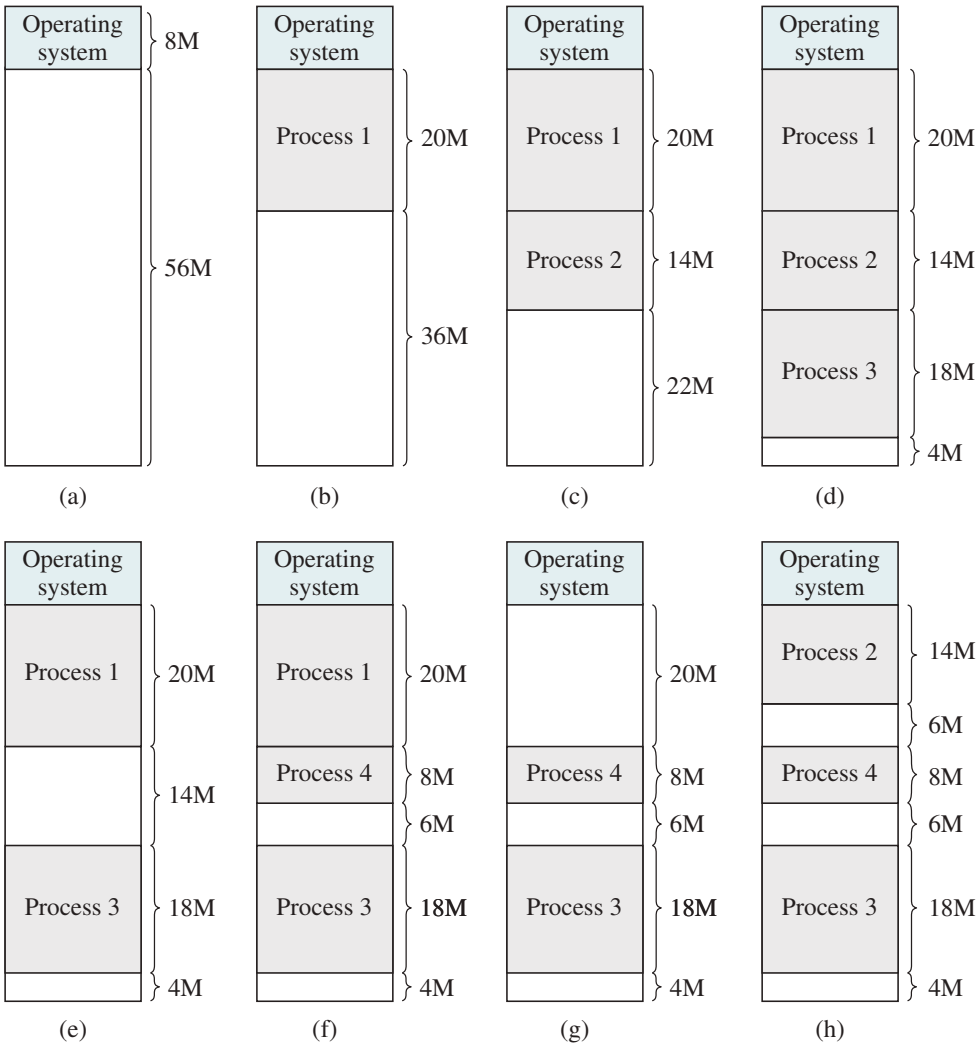


Figure 7.4 The Effect of Dynamic Partitioning

Later, a point is reached at which none of the processes in main memory is ready, but process 2, in the Ready-Suspend state, is available. Because there is insufficient room in memory for process 2, the operating system swaps process 1 out (see Figure 7.4g) and swaps process 2 back in (see Figure 7.4h).

As this example shows, this method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as **external fragmentation**, indicating the memory that is external to all partitions becomes increasingly fragmented. This is in contrast to internal fragmentation, referred to earlier.

One technique for overcoming external fragmentation is **compaction**: From time to time, the OS shifts the processes so they are contiguous and all of the free memory is together in one block. For example, in Figure 7.4h, compaction will result in a block of free memory of length 16M. This may well be sufficient to load in an additional process. The difficulty with compaction is that it is a time-consuming procedure and wasteful of processor time. Note that compaction implies the need for a dynamic relocation capability. That is, it must be possible to move a program from one region to another in main memory, without invalidating the memory references in the program (see Appendix 7A).

Placement Algorithm Because memory compaction is time consuming, the OS designer must be clever in deciding how to assign processes to memory (how to plug the holes). When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate.

Three placement algorithms that might be considered are best-fit, first-fit, and next-fit. All, of course, are limited to choosing among free blocks of main memory that are equal to or larger than the process to be brought in. **Best-fit** chooses the block that is closest in size to the request. **First-fit** begins to scan memory from the beginning and chooses the first available block that is large enough. **Next-fit** begins to scan memory from the location of the last placement and chooses the next available block that is large enough.

Figure 7.5a shows an example memory configuration after a number of placement and swapping-out operations. The last block that was used was a 22-Mbyte block from which a 14-Mbyte partition was created. Figure 7.5b shows the difference between the best-, first-, and next-fit placement algorithms in satisfying a 16-Mbyte allocation request. Best-fit will search the entire list of available blocks and make use of the 18-Mbyte block, leaving a 2-Mbyte fragment. First-fit results in a 6-Mbyte fragment, and next-fit results in a 20-Mbyte fragment.

Which of these approaches is best will depend on the exact sequence of process swappings that occurs and the size of those processes. However, some general comments can be made (see also [BREN89], [SHOR75], and [BAYS77]). The first-fit algorithm is not only the simplest but usually the best and fastest as well. The next-fit algorithm tends to produce slightly worse results than the first-fit. The next-fit algorithm will more frequently lead to an allocation from a free block at the end of memory. The result is that the largest block of free memory, which usually appears at the end of the memory space, is quickly broken up into small fragments. Thus,

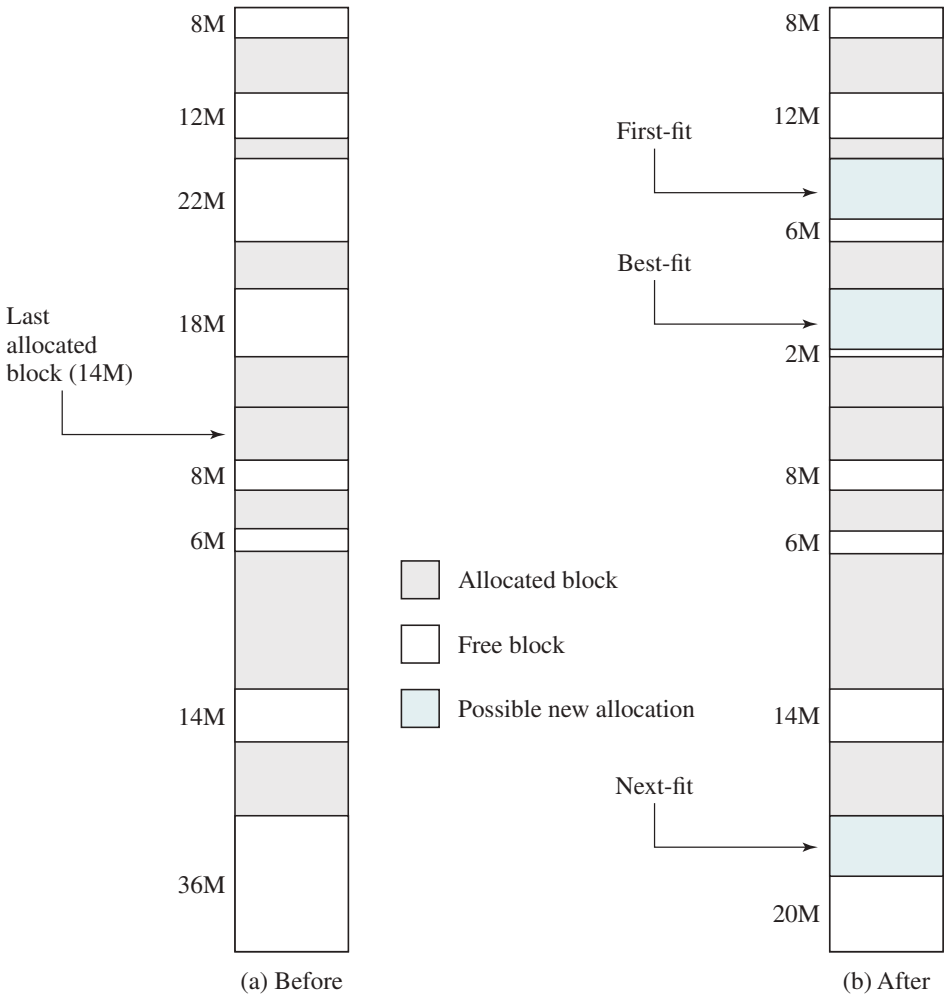


Figure 7.5 **Example of Memory Configuration before and after Allocation of 16-Mbyte Block**

compaction may be required more frequently with next-fit. On the other hand, the first-fit algorithm may litter the front end with small free partitions that need to be searched over on each subsequent first-fit pass. The best-fit algorithm, despite its name, is usually the worst performer. Because this algorithm looks for the smallest block that will satisfy the requirement, it guarantees that the fragment left behind is as small as possible. Although each memory request always wastes the smallest amount of memory, the result is that main memory is quickly littered by blocks too small to satisfy memory allocation requests. Thus, memory compaction must be done more frequently than with the other algorithms.

Replacement Algorithm In a multiprogramming system using dynamic partitioning, there will come a time when all of the processes in main memory are in a blocked state and there is insufficient memory, even after compaction, for an additional process. To avoid wasting processor time waiting for an active process to become unblocked, the OS will swap one of the processes out of main memory to make room for a new process or for a process in a Ready-Suspend state. Therefore,

the operating system must choose which process to replace. Because the topic of replacement algorithms will be covered in some detail with respect to various virtual memory schemes, we defer a discussion of replacement algorithms until then.

Buddy System

Both fixed and dynamic partitioning schemes have drawbacks. A fixed partitioning scheme limits the number of active processes and may use space inefficiently if there is a poor match between available partition sizes and process sizes. A dynamic partitioning scheme is more complex to maintain and includes the overhead of compaction. An interesting compromise is the buddy system ([KNUT97], [PETE77]).

In a buddy system, memory blocks are available of size 2^K words, $L \leq K \leq U$, where

2^L = smallest size block that is allocated

2^U = largest size block that is allocated; generally 2^U is the size of the entire memory available for allocation

To begin, the entire space available for allocation is treated as a single block of size 2^U. If a request of size s such that $2^{U-1} < s \leq 2^U$ is made, then the entire block is allocated. Otherwise, the block is split into two equal buddies of size 2^{U-1}. If $2^{U-2} < s \leq 2^{U-1}$, then the request is allocated to one of the two buddies. Otherwise, one of the buddies is split in half again. This process continues until the smallest block greater than or equal to s is generated and allocated to the request. At any time, the buddy system maintains a list of holes (unallocated blocks) of each size 2^i. A hole may be removed from the $(i + 1)$ list by splitting it in half to create two buddies of size 2^i in the i list. Whenever a pair of buddies on the i list both become unallocated, they are removed from that list and coalesced into a single block on the $(i + 1)$ list. Presented with a request for an allocation of size k such that $2^{i-1} < k \leq 2^i$, the following recursive algorithm is used to find a hole of size 2^i:

```
void  get_hole(int  i)
{
  if  (i == (U + 1)) <failure>;
  if  (<i_list empty>) {
       get_hole(i + 1);
       <split hole into buddies>;
       <put buddies on i_list>;
  }
  <take first hole on i_list>;
}
```

Figure 7.6 gives an example using a 1-Mbyte initial block. The first request, A, is for 100 Kbytes, for which a 128K block is needed. The initial block is divided into two 512K buddies. The first of these is divided into two 256K buddies, and the first of these is divided into two 128K buddies, one of which is allocated to A. The next request, B, requires a 256K block. Such a block is already available and is allocated. The process continues with splitting and coalescing occurring as needed. Note that

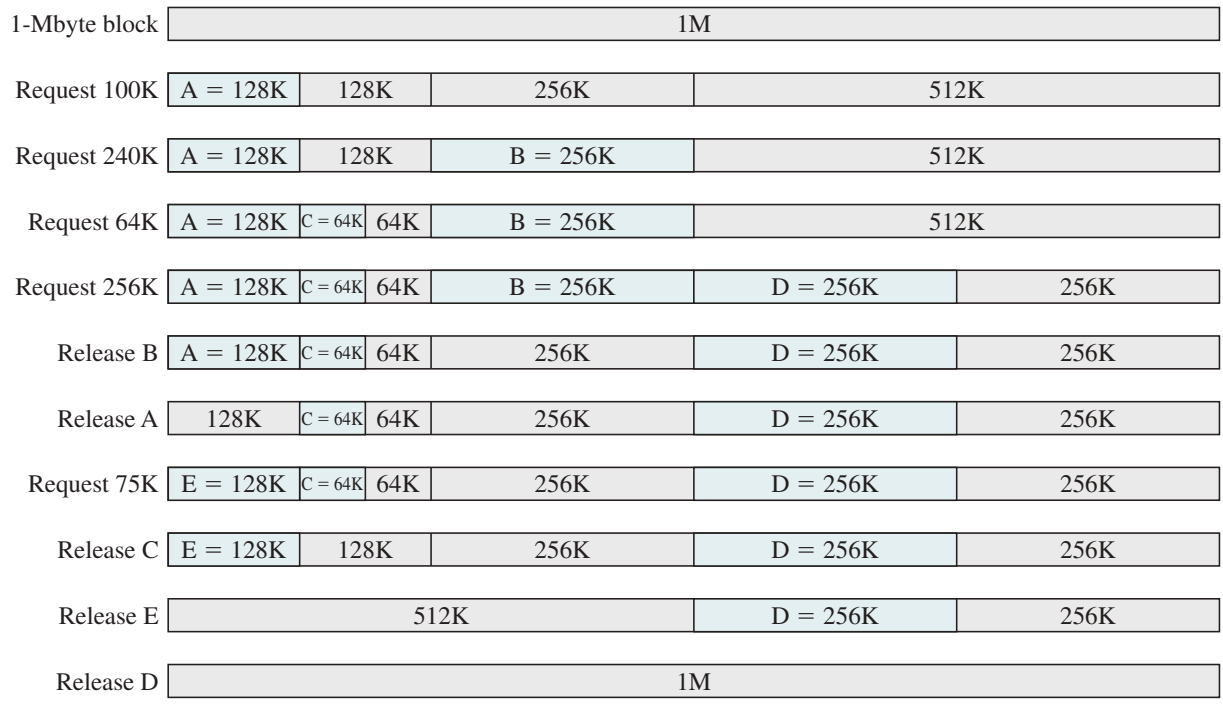


Figure 7.6 Example of the Buddy System

when E is released, two 128K buddies are coalesced into a 256K block, which is immediately coalesced with its buddy.

Figure 7.7 shows a binary tree representation of the buddy allocation immediately after the Release B request. The leaf nodes represent the current partitioning of the memory. If two buddies are leaf nodes, then at least one must be allocated; otherwise, they would be coalesced into a larger block.

The buddy system is a reasonable compromise to overcome the disadvantages of both the fixed and variable partitioning schemes, but in contemporary operating systems, virtual memory based on paging and segmentation is superior. However, the buddy system has found application in parallel systems as an efficient means of allocation and release for parallel programs (e.g., see [JOHN92]). A modified form of the buddy system is used for UNIX kernel memory allocation (described in Chapter 8).

Relocation

Before we consider ways of dealing with the shortcomings of partitioning, we must clear up one loose end, which relates to the placement of processes in memory. When the fixed partition scheme of Figure 7.3a is used, we can expect a process will always be assigned to the same partition. That is, whichever partition is selected when a new process is loaded will always be used to swap that process back into memory after it has been swapped out. In that case, a simple relocating loader, such as is described in Appendix 7A, can be used: When the process is first loaded, all relative memory references in the code are replaced by absolute main memory addresses, determined by the base address of the loaded process.

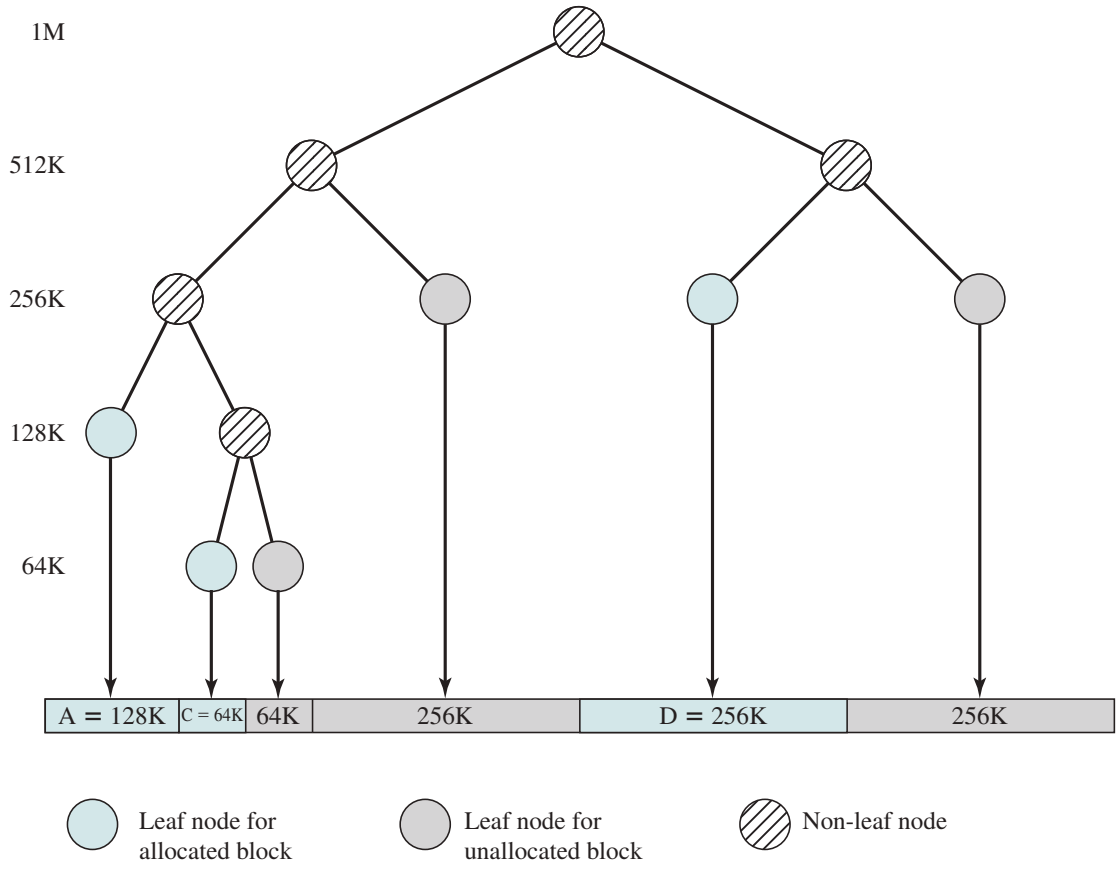


Figure 7.7 **Tree Representation of the Buddy System**

In the case of equal-size partitions (see Figure 7.2a) and in the case of a single process queue for unequal-size partitions (see Figure 7.3b), a process may occupy different partitions during the course of its life. When a process image is first created, it is loaded into some partition in main memory. Later, the process may be swapped out; when it is subsequently swapped back in, it may be assigned to a different partition than the last time. The same is true for dynamic partitioning. Observe in Figure 7.4c and Figure 7.4h that process 2 occupies two different regions of memory on the two occasions when it is brought in. Furthermore, when compaction is used, processes are shifted while they are in main memory. Thus, the locations (of instructions and data) referenced by a process are not fixed. They will change each time a process is swapped in or shifted. To solve this problem, a distinction is made among several types of addresses. A **logical address** is a reference to a memory location independent of the current assignment of data to memory; a translation must be made to a physical address before the memory access can be achieved. A **relative address** is a particular example of logical address, in which the address is expressed as a location relative to some known point, usually a value in a processor register. A **physical address**, or absolute address, is an actual location in main memory.

Programs that employ relative addresses in memory are loaded using dynamic run-time loading (see Appendix 7A for a discussion). Typically, all of the memory references in the loaded process are relative to the origin of the program. Thus, a

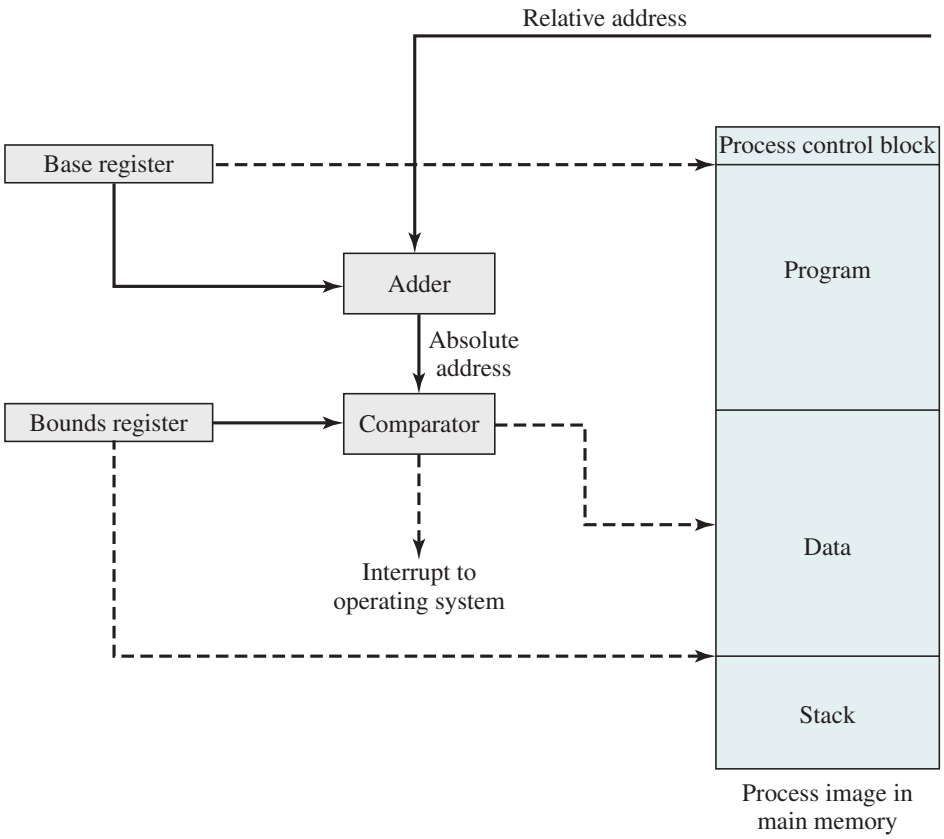


Figure 7.8 Hardware Support for Relocation

hardware mechanism is needed for translating relative addresses to physical main memory addresses at the time of execution of the instruction that contains the reference.

Figure 7.8 shows the way in which this address translation is typically accomplished. When a process is assigned to the Running state, a special processor register, sometimes called the base register, is loaded with the starting address in main memory of the program. There is also a "bounds" register that indicates the ending location of the program; these values must be set when the program is loaded into memory or when the process image is swapped in. During the course of execution of the process, relative addresses are encountered. These include the contents of the instruction register, instruction addresses that occur in branch and call instructions, and data addresses that occur in load and store instructions. Each such relative address goes through two steps of manipulation by the processor. First, the value in the base register is added to the relative address to produce an absolute address. Second, the resulting address is compared to the value in the bounds register. If the address is within bounds, then the instruction execution may proceed. Otherwise, an interrupt is generated to the operating system, which must respond to the error in some fashion.

The scheme of Figure 7.8 allows programs to be swapped in and out of memory during the course of execution. It also provides a measure of protection: Each process image is isolated by the contents of the base and bounds registers, and is safe from unwanted accesses by other processes.

7.3 PAGING

Both unequal fixed-size and variable-size partitions are inefficient in the use of memory; the former results in internal fragmentation, the latter in external fragmentation. Suppose, however, main memory is partitioned into equal fixed-size chunks that are relatively small, and each process is also divided into small fixed-size chunks of the same size. Then the chunks of a process, known as **pages**, could be assigned to available chunks of memory, known as **frames**, or page frames. We show in this section that the wasted space in memory for each process is due to internal fragmentation consisting of only a fraction of the last page of a process. There is no external fragmentation.

Figure 7.9 illustrates the use of pages and frames. At a given point in time, some of the frames in memory are in use, and some are free. A list of free frames is maintained by the OS. Process A, stored on disk, consists of four pages. When it is time to load this process, the OS finds four free frames and loads the four pages of process A into the four frames (see Figure 7.9b). Process B, consisting of three pages, and process C, consisting of four pages, are subsequently loaded. Then process B is suspended and is swapped out of main memory. Later, all of the processes in main memory are blocked, and the OS needs to bring in a new process, process D, which consists of five pages.

Now suppose, as in this example, there are not sufficient unused contiguous frames to hold the process. Does this prevent the operating system from loading D? The answer is no, because we can once again use the concept of logical address. A simple base address register will no longer suffice. Rather, the operating system maintains a **page table** for each process. The page table shows the frame location for each page of the process. Within the program, each logical address consists of a page number and an offset within the page. Recall that in the case of simple partition, a logical address is the location of a word relative to the beginning of the program; the processor translates that into a physical address. With paging, the logical-to-physical address translation is still done by processor hardware. Now the processor must know how to access the page table of the current process. Presented with a logical address (page number, offset), the processor uses the page table to produce a physical address (frame number, offset).

Continuing our example, the five pages of process D are loaded into frames 4, 5, 6, 11, and 12. Figure 7.10 shows the various page tables at this time. A page table contains one entry for each page of the process, so the table is easily indexed by the page number (starting at page 0). Each page table entry contains the number of the frame in main memory, if any, that holds the corresponding page. In addition, the OS maintains a single free-frame list of all the frames in main memory that are currently unoccupied and available for pages.

Thus, we see that simple paging, as described here, is similar to fixed partitioning. The differences are that, with paging, the partitions are rather small; a program may occupy more than one partition; and these partitions need not be contiguous.

To make this paging scheme convenient, let us dictate that the page size, hence the frame size, must be a power of 2. With the use of a page size that is a power of 2, it is easy to demonstrate that the relative address (which is defined with reference to the origin of the program) and the logical address (expressed as a page number

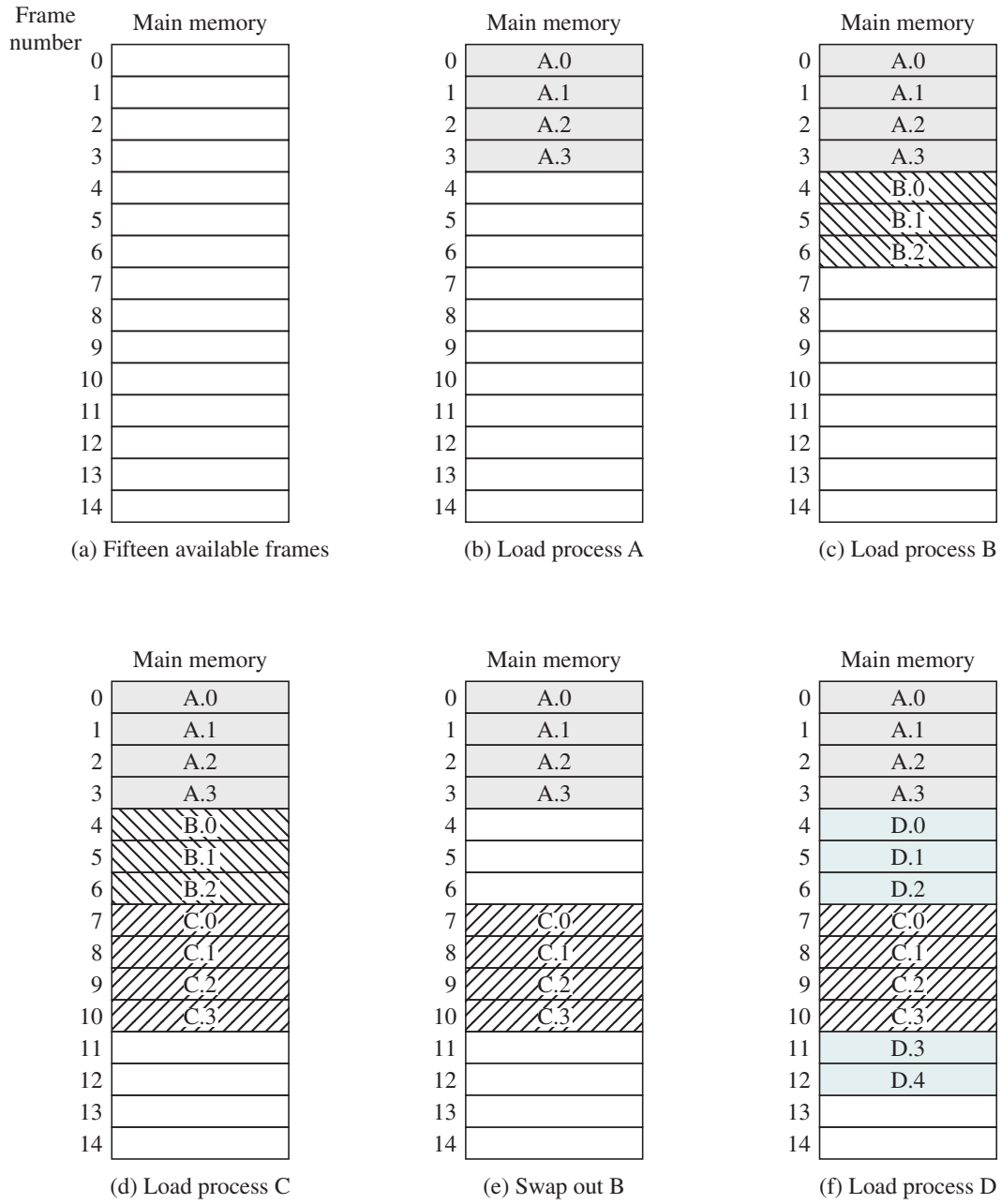


Figure 7.9 Assignment of Process to Free Frames

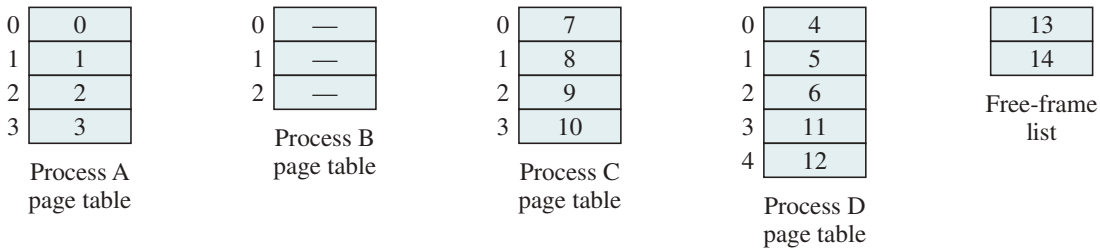


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

and offset) are the same. An example is shown in Figure 7.11. In this example, 16-bit addresses are used, and the page size is 1K = 1024 bytes. The relative address 1502 in binary form is 0000010111011110. With a page size of 1K, an offset field of 10 bits is needed, leaving 6 bits for the page number. Thus, a program can consist of a maximum of $2^6 = 64$ pages of size 1 Kbytes each. As Figure 7.11b shows, relative address 1502 corresponds to an offset of 478 (0111011110) on page 1 (000001), which yields the same 16-bit number, 0000010111011110.

The consequences of using a page size that is a power of 2 are twofold. First, the logical addressing scheme is transparent to the programmer, the assembler, and the linker. Each logical address (page number, offset) of a program is identical to its relative address. Second, it is a relatively easy matter to implement a function in hardware to perform dynamic address translation at run time. Consider an address of $n + m$ bits, where the leftmost n bits are the page number, and the rightmost m bits are the offset. In our example (see Figure 7.11b), $n = 6$ and $m = 10$. The following steps are needed for address translation:

1. Extract the page number as the leftmost n bits of the logical address.
2. Use the page number as an index into the process page table to find the frame number, k.
3. The starting physical address of the frame is $k \times 2_m$, and the physical address of the referenced byte is that number plus the offset. This physical address need not be calculated; it is easily constructed by appending the frame number to the offset.

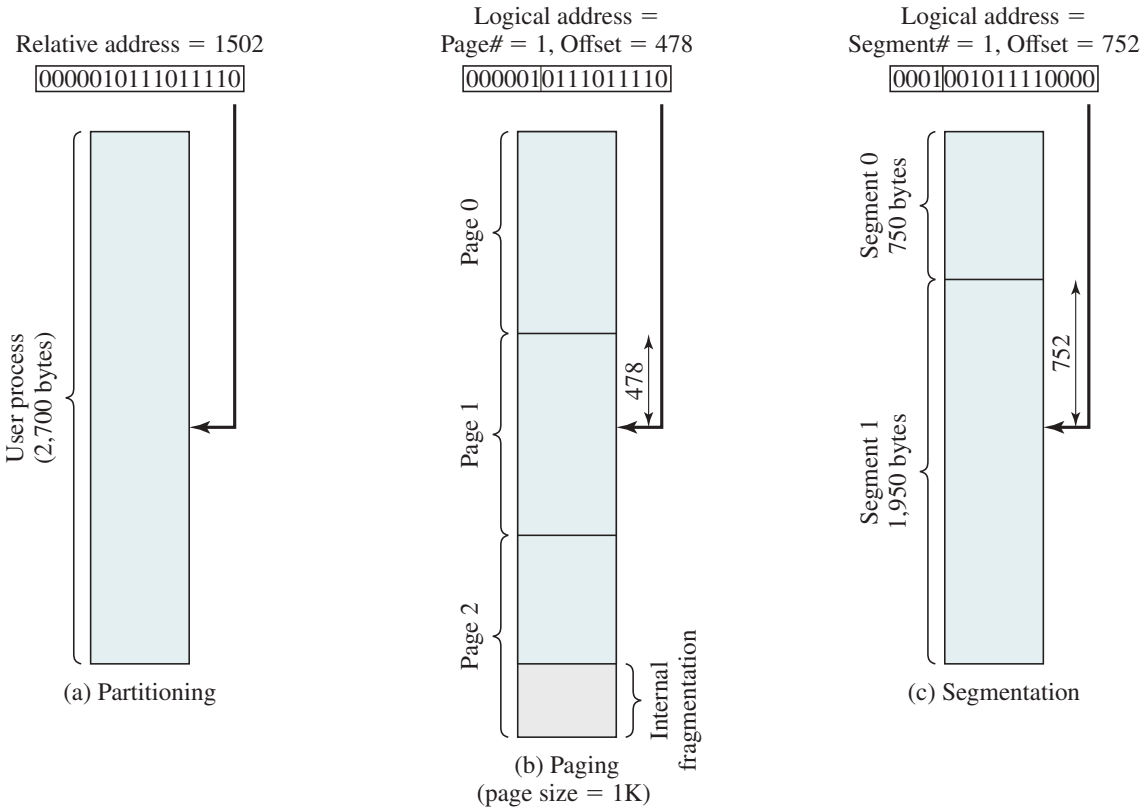


Figure 7.11 **Logical Addresses**

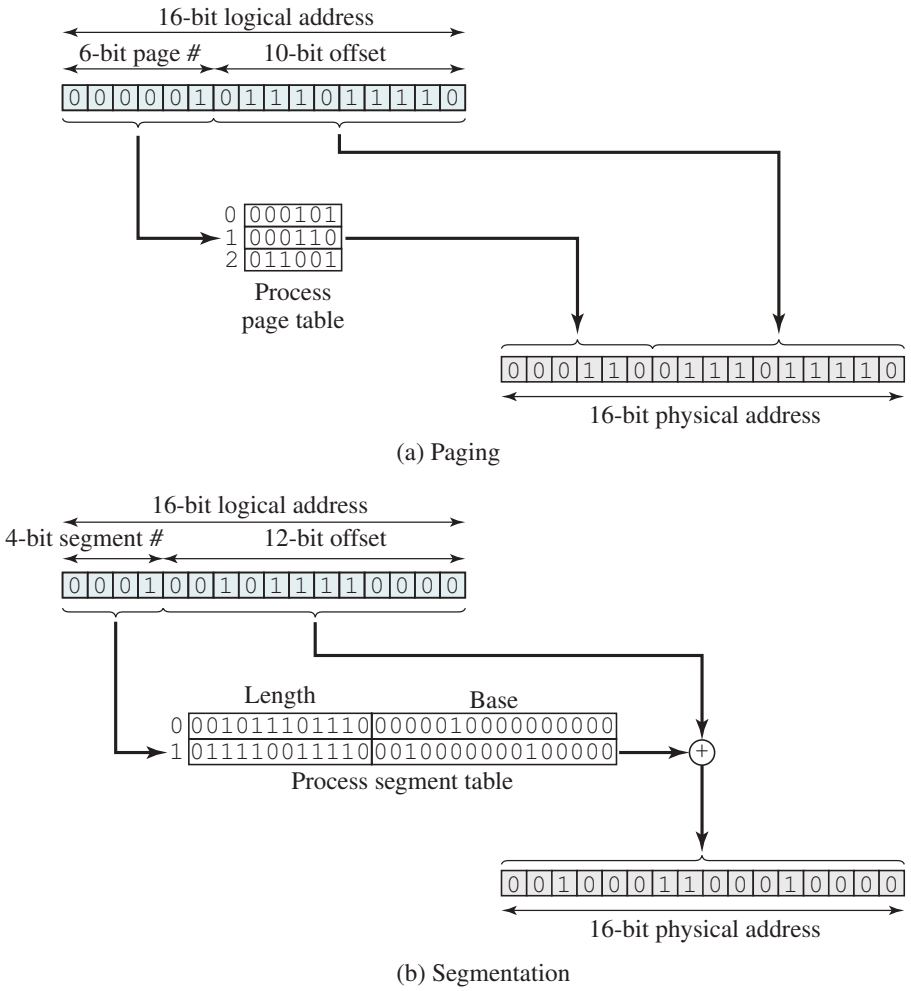


Figure 7.12 Examples of Logical-to-Physical Address Translation

In our example, we have the logical address 0000010111011110, which is page number 1, offset 478. Suppose this page is residing in main memory frame 6 = binary 000110. Then the physical address is frame number 6, offset 478 = 0001100111011110 (see Figure 7.12a).

To summarize, with simple paging, main memory is divided into many small equal-size frames. Each process is divided into frame-size pages. Smaller processes require fewer pages; larger processes require more. When a process is brought in, all of its pages are loaded into available frames, and a page table is set up. This approach solves many of the problems inherent in partitioning.

7.4 SEGMENTATION

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of **segments**. It is not required that all segments of all programs be of the same length, although there is a maximum segment

length. As with paging, a logical address using segmentation consists of two parts, in this case, a segment number and an offset.

Because of the use of unequal-size segments, segmentation is similar to dynamic partitioning. In the absence of an overlay scheme or the use of virtual memory, it would be required that all of a program's segments be loaded into memory for execution. The difference, compared to dynamic partitioning, is that with segmentation a program may occupy more than one partition, and these partitions need not be contiguous. Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation. However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less.

Whereas paging is invisible to the programmer, segmentation is usually visible and is provided as a convenience for organizing programs and data. Typically, the programmer or compiler will assign programs and data to different segments. For purposes of modular programming, the program or data may be further broken down into multiple segments. The principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation.

Another consequence of unequal-size segments is that there is no simple relationship between logical addresses and physical addresses. Analogous to paging, a simple segmentation scheme would make use of a segment table for each process, and a list of free blocks of main memory. Each segment table entry would have to give the starting address in main memory of the corresponding segment. The entry should also provide the length of the segment to assure that invalid addresses are not used. When a process enters the Running state, the address of its segment table is loaded into a special register used by the memory management hardware. Consider an address of $n + m$ bits, where the leftmost n bits are the segment number and the rightmost m bits are the offset. In our example (see Figure 7.11c), $n = 4$ and $m = 12$. Thus, the maximum segment size is $2^{12} = 4096$. The following steps are needed for address translation:

1. Extract the segment number as the leftmost n bits of the logical address.
2. Use the segment number as an index into the process segment table to find the starting physical address of the segment.
3. Compare the offset, expressed in the rightmost m bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid.
4. The desired physical address is the sum of the starting physical address of the segment plus the offset.

In our example, we have the logical address 0001001011110000, which is segment number 1, offset 752. Suppose this segment is residing in main memory starting at physical address 0010000000100000. Then the physical address is 0010000000100000 + 001011110000 = 0010001100010000 (see Figure 7.12b).

To summarize, with simple segmentation, a process is divided into a number of segments that need not be of equal size. When a process is brought in, all of its segments are loaded into available regions of memory, and a segment table is set up.

7.5 SUMMARY

One of the most important and complex tasks of an operating system is memory management. Memory management involves treating main memory as a resource to be allocated to and shared among a number of active processes. To use the processor and the I/O facilities efficiently, it is desirable to maintain as many processes in main memory as possible. In addition, it is desirable to free programmers from size restrictions in program development.

The basic tools of memory management are paging and segmentation. With paging, each process is divided into relatively small, fixed-size pages. Segmentation provides for the use of pieces of varying size. It is also possible to combine segmentation and paging in a single memory management scheme.

7.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

absolute loading	linkage editor	physical address
buddy system	linking	physical organization
compaction	loading	protection
dynamic linking	logical address	relative address
dynamic partitioning	logical organization	relocatable loading
dynamic run-time loading	memory management	relocation
external fragmentation	page	segment
fixed partitioning	page table	segmentation
frame	paging	sharing
internal fragmentation	partitioning	

Review Questions

7.1. What requirements is memory management intended to satisfy?
7.2. What is relocation of a program?
7.3. What are the advantages of organizing programs and data into modules?
7.4. What are some reasons to allow two or more processes to all have access to a particular region of memory?
7.5. In a fixed partitioning scheme, what are the advantages of using unequal-size partitions?
7.6. What is the difference between internal and external fragmentation?
7.7. What is address binding? State the different timings when address binding may occur.
7.8. What is the difference between a page and a frame?
7.9. What is the difference between a page and a segment?

Problems

7.1. In Section 2.3, we listed five objectives of memory management, and in Section 7.1, we listed five requirements. Argue that each list encompasses all of the concerns addressed in the other.

7.2. Consider a fixed partitioning scheme with equal-size partitions of 2^{16} bytes and a total main memory size of 2^{24} bytes. A process table is maintained that includes a pointer to a partition for each resident process. How many bits are required for the pointer?

7.3. Consider a dynamic partitioning scheme. Show that, on average, the memory contains half as many holes as segments.

7.4. To implement the various placement algorithms discussed for dynamic partitioning (see Section 7.2), a list of the free blocks of memory must be kept. For each of the three methods discussed (best-fit, first-fit, next-fit), what is the average length of the search?

7.5. Another placement algorithm for dynamic partitioning is referred to as worst-fit. In this case, the largest free block of memory is used for bringing in a process.

a. Discuss the pros and cons of this method compared to first-, next-, and best-fit.

b. What is the average length of the search for worst-fit?

7.6. This diagram shows an example of memory configuration under dynamic partitioning, after a number of placement and swapping-out operations have been carried out. Addresses go from left to right; gray areas indicate blocks occupied by processes; white areas indicate free memory blocks. The last process placed is 2 Mbytes and is marked with an X. Only one process was swapped out after that.

a. What was the maximum size of the swapped-out process?

b. What was the size of the free block just before it was partitioned by X?

c. A new 3-Mbyte allocation request must be satisfied next. Indicate the intervals of memory where a partition will be created for the new process under the following four placement algorithms: best-fit, first-fit, next-fit, and worst-fit. For each algorithm, draw a horizontal segment under the memory strip and label it clearly.

7.7. A 512 KB block of memory is allocated using the buddy system. Show the results of the following sequence of requests and returns in a figure that is similar to Figure 7.6: Request A: 100; Request B: 40; Request C: 190; Return A; Request D: 60; Return B; Return D; Return C. Also, find the internal fragmentation at each stage of allocation/de-allocation.

7.8. Consider a memory-management system that uses simple paging strategy and employs registers to speed up page lookups. The associative registers have a lookup performance of 120 ns and a hit ratio of 80%. The main-memory page-table lookup takes 600 ns. Compute the average page-lookup time.

7.9. Let $\text{buddy}_k(x)$ = address of the buddy of the block of size 2^k whose address is x. Write a general expression for $\text{buddy}_k(x)$.

7.10. The Fibonacci sequence is defined as follows:

$$F_0 = 0, F_1 = 1, F_{n+2} = F_{n+1} + F_n, n \geq 0$$

a. Could this sequence be used to establish a buddy system?

b. What would be the advantage of this system over the binary buddy system described in this chapter?

7.11. During the course of execution of a program, the processor will increment the contents of the instruction register (program counter) by one word after each instruction fetch, but will alter the contents of that register if it encounters a branch or call instruction that causes execution to continue elsewhere in the program. Now consider Figure 7.8. There are two alternatives with respect to instruction addresses:

1. Maintain a relative address in the instruction register and do the dynamic address translation using the instruction register as input. When a successful branch or call is encountered, the relative address generated by that branch or call is loaded into the instruction register.
2. Maintain an absolute address in the instruction register. When a successful branch or call is encountered, dynamic address translation is employed, with the results stored in the instruction register.

Which approach is preferable?

7.12. Consider a memory-management system based on paging. The total size of the physical memory is 2 GB, laid out over pages of size 8 KB. The logical address space of each process has been limited to 256 MB.

a. Determine the total number of bits in the physical address.

b. Determine the number of bits specifying page replacement and the number of bits for page frame number.

c. Determine the number of page frames.

d. Determine the logical address layout.

7.13. Write the binary translation of the logical address 0011000000110011 under the following hypothetical memory management schemes, and explain your answer:

a. A paging system with a 512-address page size, using a page table in which the frame number happens to be half of the page number.

b. A segmentation system with a 2K-address maximum segment size, using a segment table in which bases happen to be regularly placed at real addresses: segment # + 20 + offset + 4,096.

7.14. Consider a simple segmentation system that has the following segment table:

Starting Address	Length (bytes)
830	346
648	110
1,508	408
770	812

For each of the following logical addresses, determine the physical address or indicate if a segment fault occurs:

a. 0, 228

b. 2, 648

c. 3, 776

d. 1, 98

e. 1, 240

7.15. Consider a memory in which contiguous segments $S_1, S_2, \ldots, S_n$ are placed in their order of creation from one end of the store to the other, as suggested by the following figure:

When segment S_{n+1} is being created, it is placed immediately after segment S_n even though some of the segments $S_1, S_2, \ldots, S_n$ may already have been deleted. When the boundary between segments (in use or deleted) and the hole reaches the other end of the memory, the segments in use are compacted.

a. Show that the fraction of time F spent on compacting obeys the following inequality:

$$F \geq \frac{1 - f}{1 + kf} \text{ where } k = \frac{t}{2s} - 1$$

where
s = average length of a segment, in words
t = average lifetime of a segment, in memory references
f = fraction of the memory that is unused under equilibrium conditions

Hint: Find the average speed at which the boundary crosses the memory and assume that the copying of a single word requires at least two memory references.

b. Find F for $f = 0.25$, $t = 1{,}200$, and $s = 75$.

APPENDIX 7A LOADING AND LINKING

The first step in the creation of an active process is to load a program into main memory and create a process image (see Figure 7.13). Figure 7.14 depicts a scenario typical for most systems. The application consists of a number of compiled or assembled modules in object-code form. These are linked to resolve any references between modules. At the same time, references to library routines are resolved. The library routines themselves may be incorporated into the program or referenced as shared

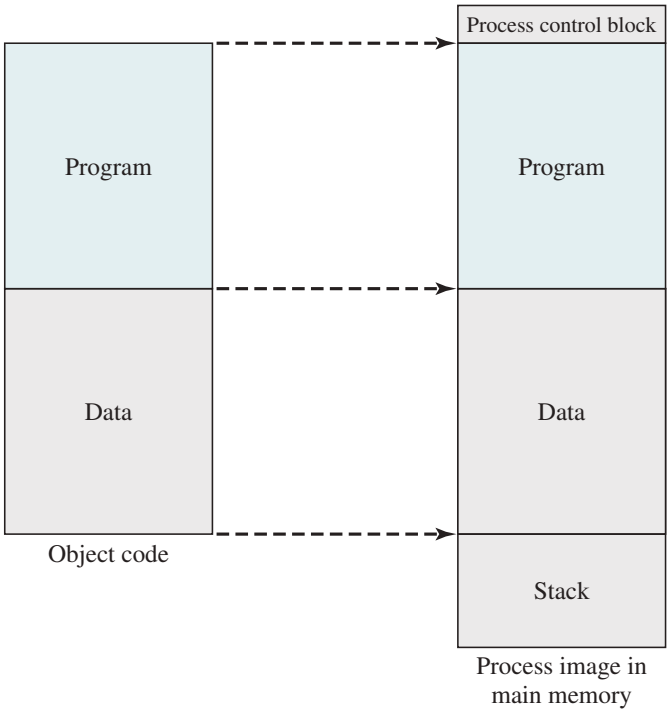


Figure 7.13 The Loading Function

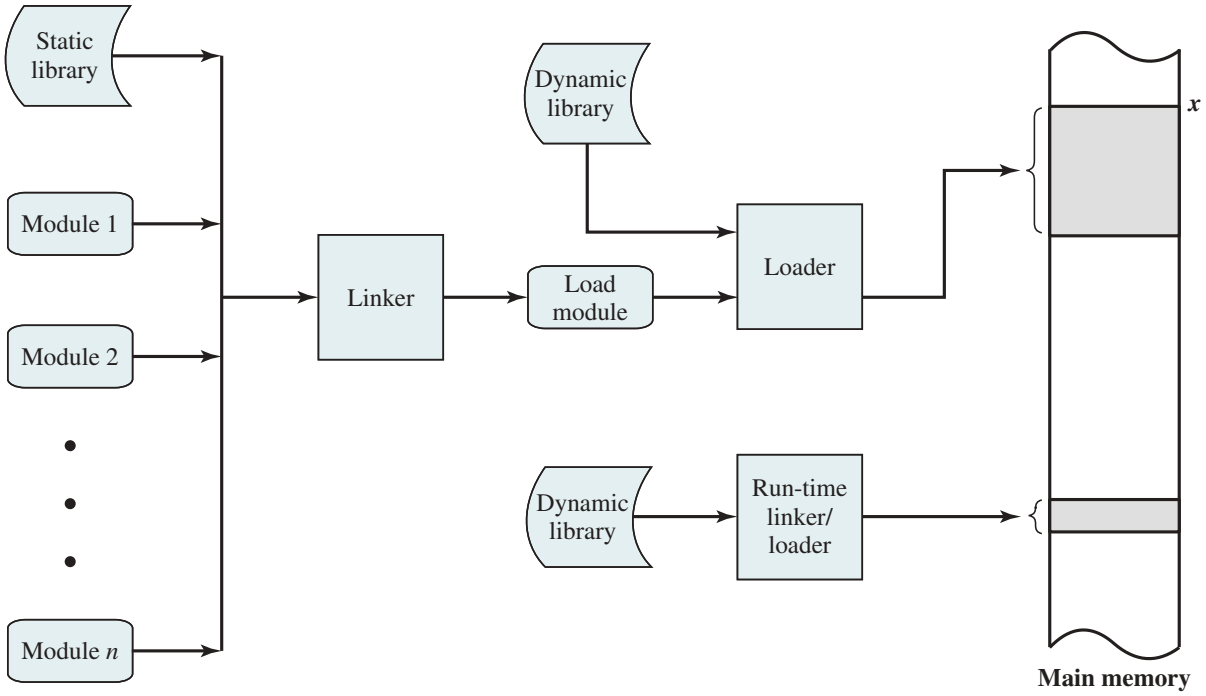


Figure 7.14 A Linking and Loading Scenario

code that must be supplied by the operating system at run time. In this appendix, we summarize the key features of linkers and loaders. For clarity in the presentation, we begin with a description of the loading task when a single program module is involved; no linking is required.

Loading

In Figure 7.14, the loader places the load module in main memory starting at location x. In loading the program, the addressing requirement illustrated in Figure 7.1 must be satisfied. In general, three approaches can be taken:

1. Absolute loading
2. Relocatable loading
3. Dynamic run-time loading

Absolute Loading An absolute loader requires that a given load module always be loaded into the same location in main memory. Thus, in the load module presented to the loader, all address references must be to specific, or absolute, main memory addresses. For example, if x in Figure 7.14 is location 1024, then the first word in a load module destined for that region of memory has address 1024.

The assignment of specific address values to memory references within a program can be done either by the programmer or at compile or assembly time (see Table 7.3a). There are several disadvantages to the former approach. First, every programmer would have to know the intended assignment strategy for placing modules into main memory. Second, if any modifications are made to the program that involve insertions or deletions in the body of the module, then all of the addresses will have

Table 7.3 Address Binding

(a) Loader

Binding Time	Function
Programming time	All actual physical addresses are directly specified by the programmer in the program itself.
Compile or assembly time	The program contains symbolic address references, and these are converted to actual physical addresses by the compiler or assembler.
Load time	The compiler or assembler produces relative addresses. The loader translates these to absolute addresses at the time of program loading.
Run time	The loaded program retains relative addresses. These are converted dynamically to absolute addresses by processor hardware.

(b) Linker

Linkage Time	Function
Programming time	No external program or data references are allowed. The programmer must place into the program the source code for all subprograms that are referenced.
Compile or assembly time	The assembler must fetch the source code of every subroutine that is referenced and assemble them as a unit.
Load module creation	All object modules have been assembled using relative addresses. These modules are linked together and all references are restated relative to the origin of the final load module.
Load time	External references are not resolved until the load module is to be loaded into main memory. At that time, referenced dynamic link modules are appended to the load module, and the entire package is loaded into main or virtual memory.
Run time	External references are not resolved until the external call is executed by the processor. At that time, the process is interrupted and the desired module is linked to the calling program.

to be altered. Accordingly, it is preferable to allow memory references within programs to be expressed symbolically, then resolve those symbolic references at the time of compilation or assembly. This is illustrated in Figure 7.15. Every reference to an instruction or item of data is initially represented by a symbol. In preparing the module for input to an absolute loader, the assembler or compiler will convert all of these references to specific addresses (in this example, for a module to be loaded starting at location 1024), as shown in Figure 7.15b.

Relocatable Loading The disadvantage of binding memory references to specific addresses prior to loading is that the resulting load module can only be placed in one region of main memory. However, when many programs share main memory, it may not be desirable to decide ahead of time into which region of memory a particular module should be loaded. It is better to make that decision at load time. Thus, we need a load module that can be located anywhere in main memory.

To satisfy this new requirement, the assembler or compiler produces not actual main memory addresses (absolute addresses) but addresses that are relative to some known point, such as the start of the program. This technique is illustrated in

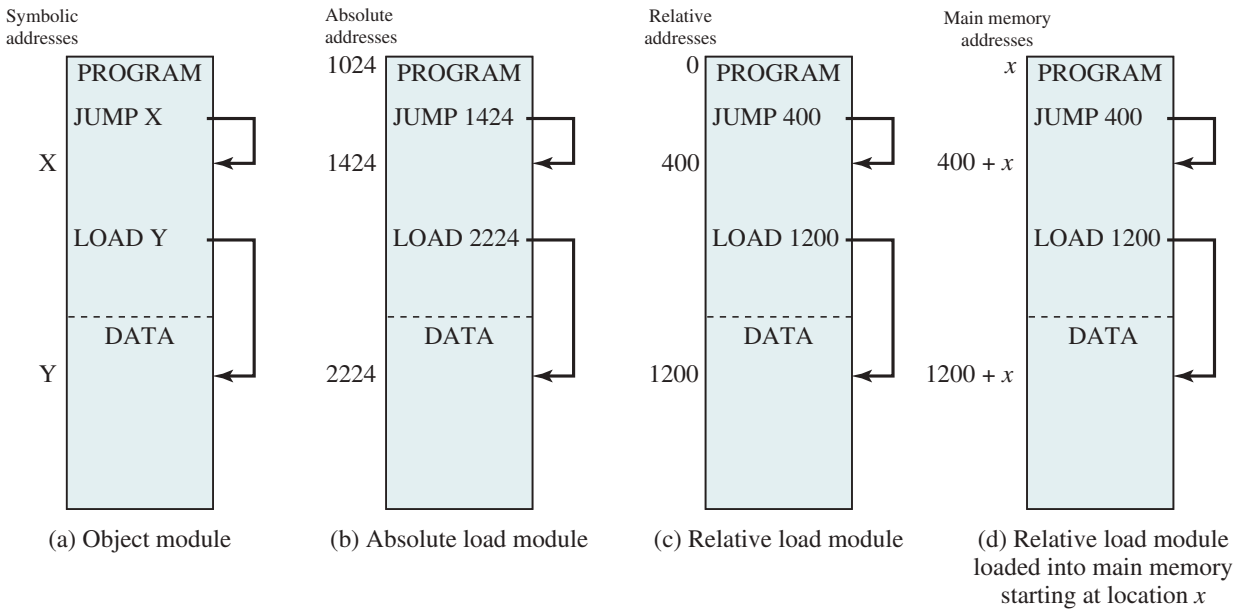


Figure 7.15 Absolute and Relocatable Load Modules

Figure 7.15c. The start of the load module is assigned the relative address 0, and all other memory references within the module are expressed relative to the beginning of the module.

With all memory references expressed in relative format, it becomes a simple task for the loader to place the module in the desired location. If the module is to be loaded beginning at location x, then the loader must simply add x to each memory reference as it loads the module into memory. To assist in this task, the load module must include information that tells the loader where the address references are and how they are to be interpreted (usually relative to the program origin, but also possibly relative to some other point in the program, such as the current location). This set of information is prepared by the compiler or assembler, and is usually referred to as the relocation dictionary.

Dynamic Run-Time Loading Relocatable loaders are common and provide obvious benefits relative to absolute loaders. However, in a multiprogramming environment, even one that does not depend on virtual memory, the relocatable loading scheme is inadequate. We have referred to the need to swap process images in and out of main memory to maximize the utilization of the processor. To maximize main memory utilization, we would like to be able to swap the process image back into different locations at different times. Thus, a program, once loaded, may be swapped out to disk then swapped back in at a different location. This would be impossible if memory references had been bound to absolute addresses at the initial load time.

The alternative is to defer the calculation of an absolute address until it is actually needed at run time. For this purpose, the load module is loaded into main memory with all memory references in relative form (see Figure 7.15d). It is not until an instruction is actually executed that the absolute address is calculated. To assure

that this function does not degrade performance, it must be done by special processor hardware rather than software. This hardware is described in Section 7.2.

Dynamic address calculation provides complete flexibility. A program can be loaded into any region of main memory. Subsequently, the execution of the program can be interrupted and the program can be swapped out of main memory, to be later swapped back in at a different location.

Linking

The function of a linker is to take as input a collection of object modules and produce a load module, consisting of an integrated set of program and data modules, to be passed to the loader. In each object module, there may be address references to locations in other modules. Each such reference can only be expressed symbolically in an unlinked object module. The linker creates a single load module that is the contiguous joining of all of the object modules. Each intramodule reference must be changed from a symbolic address to a reference to a location within the overall load module. For example, module A in Figure 7.16a contains a procedure invocation of module B. When these modules are combined in the load module, this symbolic reference to module B is changed to a specific reference to the location of the entry point of B within the load module.

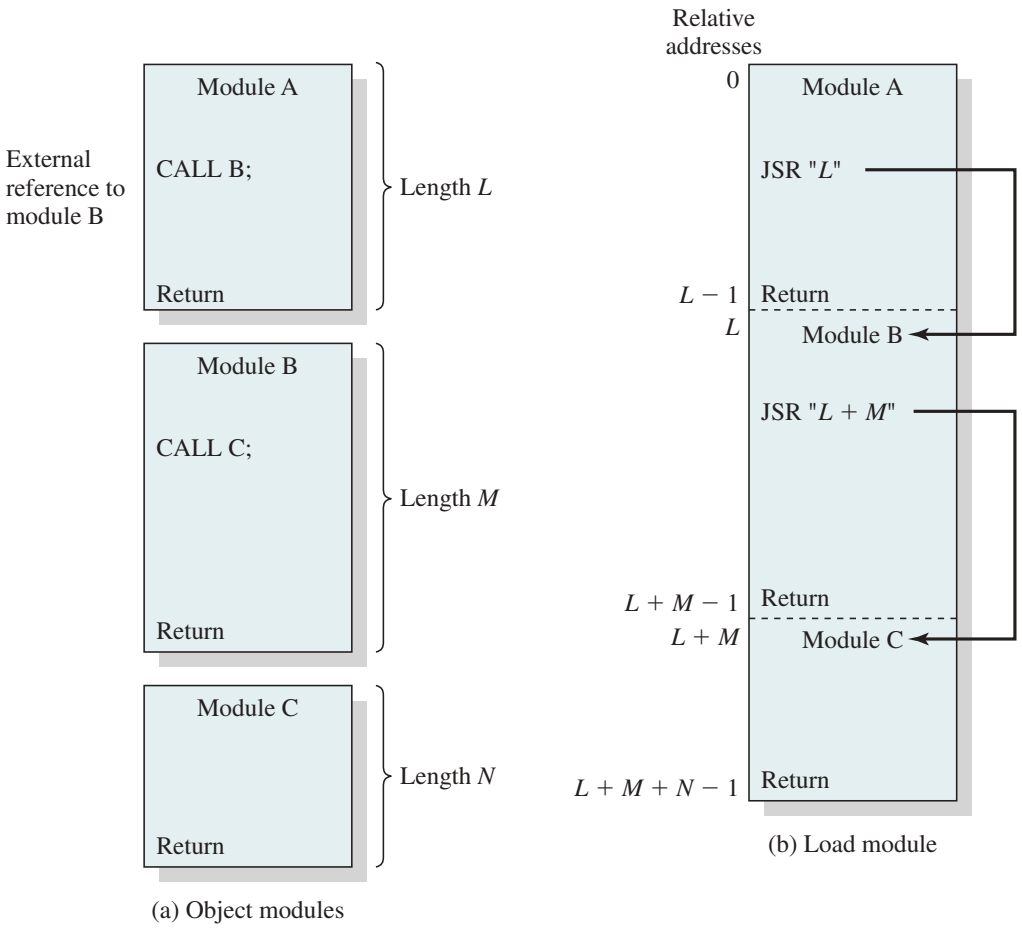


Figure 7.16 The Linking Function

Linkage Editor The nature of this address linkage will depend on the type of load module to be created and when the linkage occurs (see Table 7.3b). If, as is usually the case, a relocatable load module is desired, then linkage is usually done in the following fashion. Each compiled or assembled object module is created with references relative to the beginning of the object module. All of these modules are put together into a single relocatable load module with all references relative to the origin of the load module. This module can be used as input for relocatable loading or dynamic run-time loading.

A linker that produces a relocatable load module is often referred to as a linkage editor. Figure 7.16 illustrates the linkage editor function.

Dynamic Linker As with loading, it is possible to defer some linkage functions. The term *dynamic linking* is used to refer to the practice of deferring the linkage of some external modules until after the load module has been created. Thus, the load module contains unresolved references to other programs. These references can be resolved either at load time or run time.

For *load-time dynamic linking* (involving the upper dynamic library in Figure 7.14), the following steps occur. The load module (application module) to be loaded is read into memory. Any reference to an external module (target module) causes the loader to find the target module, load it, and alter the reference to a relative address in memory from the beginning of the application module. There are several advantages to this approach over what might be called static linking:

- It becomes easier to incorporate changed or upgraded versions of the target module, which may be an operating system utility or some other general-purpose routine. With static linking, a change to such a supporting module would require the relinking of the entire application module. Not only is this inefficient, but it may be impossible in some circumstances. For example, in the personal computer field, most commercial software is released in load module form; source and object versions are not released.
- Having target code in a dynamic link file paves the way for automatic code sharing. The operating system can recognize that more than one application is using the same target code, because it loaded and linked that code. It can use that information to load a single copy of the target code and link it to both applications, rather than having to load one copy for each application.
- It becomes easier for independent software developers to extend the functionality of a widely used operating system such as Linux. A developer can come up with a new function that may be useful to a variety of applications, and package it as a dynamic link module.

With **run-time dynamic linking** (involving the lower dynamic library in Figure 7.14), some of the linking is postponed until execution time. External references to target modules remain in the loaded program. When a call is made to the absent module, the operating system locates the module, loads it, and links it to the calling module. Such modules are typically shareable. In the Windows environment, these are called dynamic link libraries (DLLs). Thus, if one process is already making use

of a dynamically linked shared module, then that module is in main memory and a new process can simply link to the already-loaded module.

The use of DLLs can lead to a problem commonly referred to as **DLL hell**. DLL hell occurs if two or more processes are sharing a DLL module but expect different versions of the module. For example, an application or system function might be reinstalled and bring in with it an older version of a DLL file.

We have seen that dynamic loading allows an entire load module to be moved around; however, the structure of the module is static, being unchanged throughout the execution of the process and from one execution to the next. However, in some cases, it is not possible to determine prior to execution which object modules will be required. This situation is typified by transaction-processing applications, such as an airline reservation system or a banking application. The nature of the transaction dictates which program modules are required, and they are loaded as appropriate and linked with the main program. The advantage of the use of such a dynamic linker is that it is not necessary to allocate memory for program units unless those units are referenced. This capability is used in support of segmentation systems.

One additional refinement is possible: An application need not know the names of all the modules or entry points that may be called. For example, a charting program may be written to work with a variety of plotters, each of which is driven by a different driver package. The application can learn the name of the plotter that is currently installed on the system from another process or by looking it up in a configuration file. This allows the user of the application to install a new plotter that did not exist at the time the application was written.