Part Three

Memory

One of the most difficult aspects of operating system design is memory management. Although the cost of memory has dropped dramatically and, as a result, the size of main memory on modern machines has grown, reaching into the gigabyte range, there is never enough main memory to hold all of the programs and data structures needed by active processes and by the operating system. Accordingly, a central task of the operating system is to manage memory, which involves bringing in and swapping out blocks of data from secondary memory. However, memory I/O is a slow operation, and its speed relative to the processor's instruction cycle time lags further and further behind with each passing year. To keep the processor or processors busy and thus to maintain efficiency, the operating system must cleverly time the swapping in and swapping out to minimize the effect of memory I/O on performance.

ROAD MAP FOR PART THREE

Chapter 7 Memory Management

Chapter 7 provides an overview of the fundamental mechanisms used in memory management. First, the basic requirements of any memory management scheme are summarized. Then the use of memory partitioning is introduced. This technique is not much used except in special cases, such as kernel memory management. However, a review of memory partitioning illuminates many of the design issues involved in memory management. The remainder of the chapter deals with two techniques that form the basic building blocks of virtually all memory management systems: paging and segmentation.

Chapter 8 Virtual Memory

Virtual memory, based on the use of either paging or the combination of paging and segmentation, is the almost universal approach to memory management on contemporary machines. Virtual memory is a scheme that is transparent to the application processes and allows each process to behave as if it had unlimited memory at its disposal. To achieve this, the operating system creates for each process a virtual address space, or virtual memory, on disk. Part of the virtual memory is brought into real main memory as needed. In this way, many processes can share a relatively small amount of main memory. For virtual memory to work effectively, hardware mechanisms are needed to perform the basic paging and segmentation functions, such as address translation between virtual and real addresses. Chapter 8 begins with an overview of these hardware mechanisms. The remainder of the chapter is devoted to operating system design issues relating to virtual memory.

CHAPTER 7

Memory Management

In a uniprogramming system, main memory is divided into two parts: one part for the operating system (resident monitor, kernel) and one 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 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 this chapter with a look at the requirements that memory management is intended to satisfy. Next, we approach the technology of memory management by looking at a variety of simple schemes that have been used. Our focus is the requirement that a program must be loaded into main memory to be executed. This discussion introduces some of the fundamental principles of 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. [LIST93] suggests five requirements:

- Relocation
- Protection
- 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

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 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 my temporariliy be copied into a an available region of main memory (segmentation) or that segment may be divided into pages which can be individually copied into main memory (combined segmentation and paging).

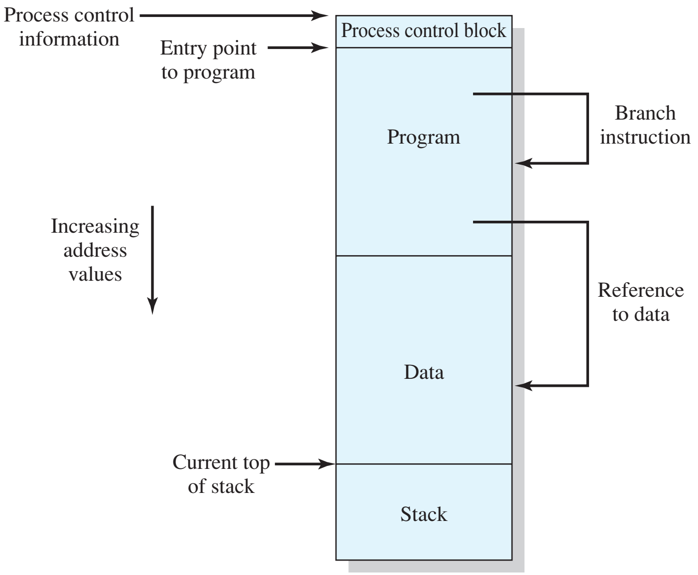


Figure 7.1 Addressing Requirments for a Process

a large pool of ready processes to execute. Once a program has been swapped out to disk, it would be quite limiting to declare 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 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 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 (for example, 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 that 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 that the memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software). This is because the operating system 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 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, and 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 and 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 plus 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 examine in this chapter and the next). One of these techniques, partitioning, has been used in several variations in

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 that 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 brought in later automatically.	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 brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.

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.

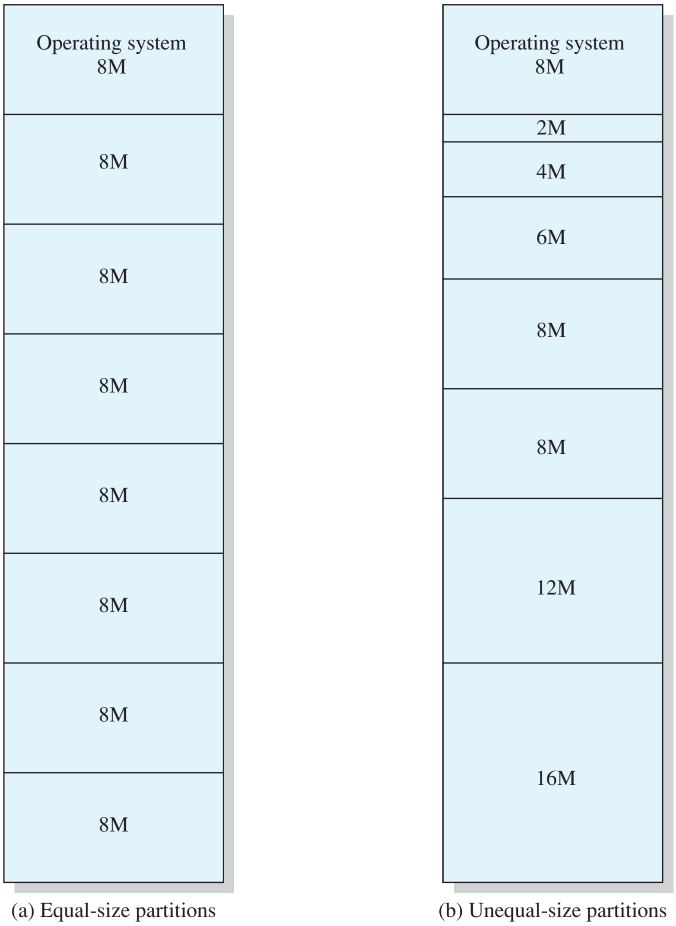
Fixed Partitioning

In most schemes for memory management, we can assume that the operating system occupies some fixed portion of main memory and that 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 that 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 that 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.



(a) Equal-size partitions

(b) Unequal-size partitions

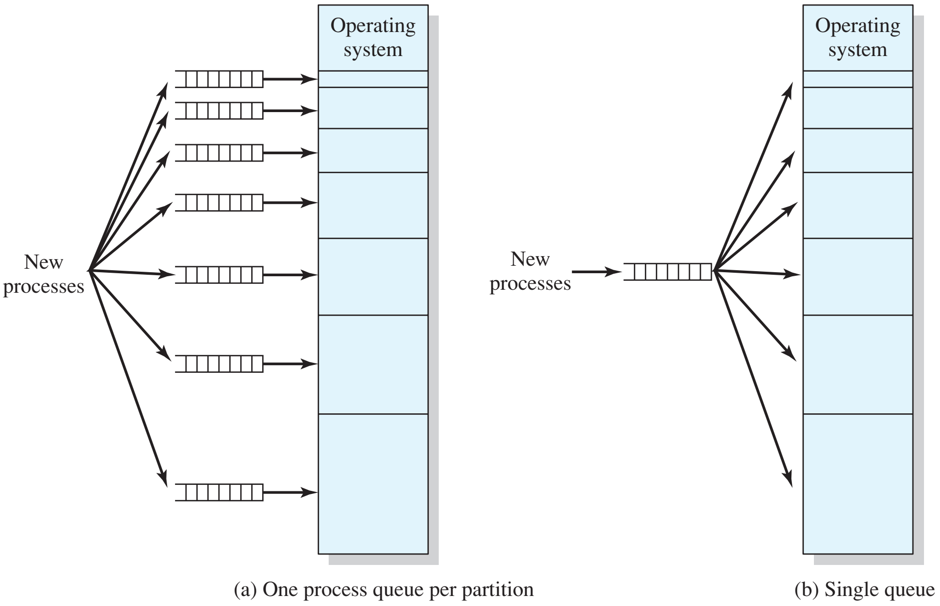
Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

- 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 (Figure 7.2b). In this example, programs as large as 16 Mbytes can 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 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 is 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 (Figure 7.3a). The



(a) One process queue per partition

(b) Single queue

Figure 7.3 **Memory Assignment for Fixed Partioning**

[1] This assumes that 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.

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. 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 (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 operating system 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 operating system (a). The first three processes are loaded in, starting where the operating system ends and occupying just enough space for each process (b, 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 (e), which leaves sufficient room to load a new process, process 4 (f). Because process 4 is smaller than process 2, another small hole is created. Later, a point is reached at which none of the processes in main memory is ready, but process 2, in the Ready-Suspend

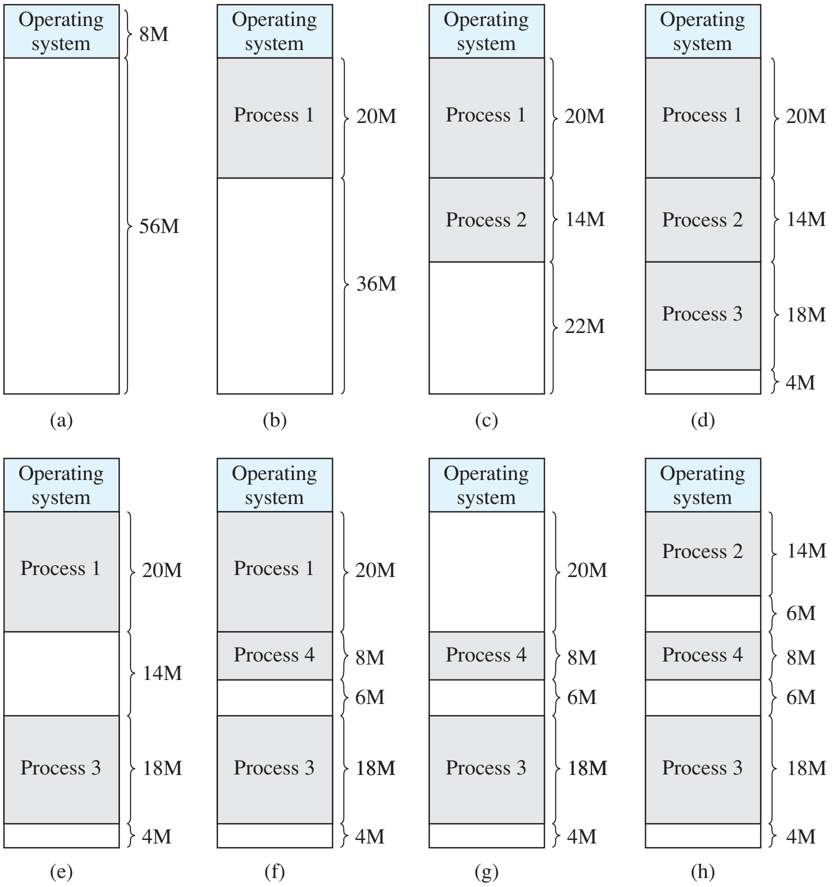


Figure 7.4 The Effect of Dynamic Partitioning

state, is available. Because there is insufficient room in memory for process 2, the operating system swaps process 1 out (g) and swaps process 2 back in (h).

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 that 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 operating system shifts the processes so that they are contiguous and so that 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 operating system 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

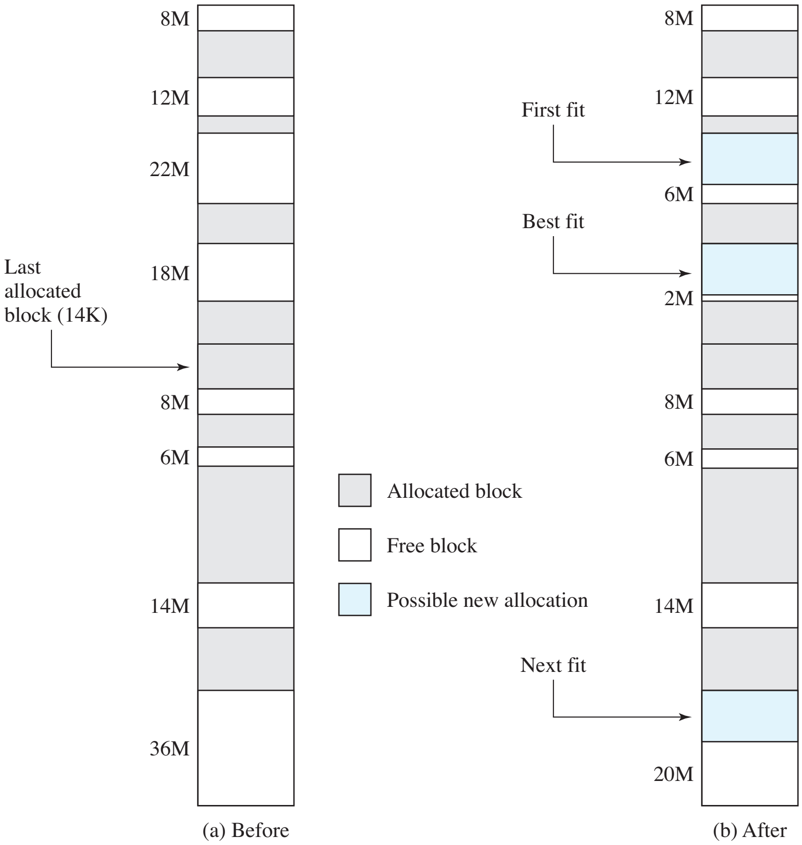


Figure 7.5 **Example Memory Configuration before and after Allocation of 16-Mbyte Block**

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, 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 operating system 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 2U 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 (from [LIST93]) 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 when E is released, two 128K buddies are coalesced into a 256K block, which is immediately coalesced with its buddy.

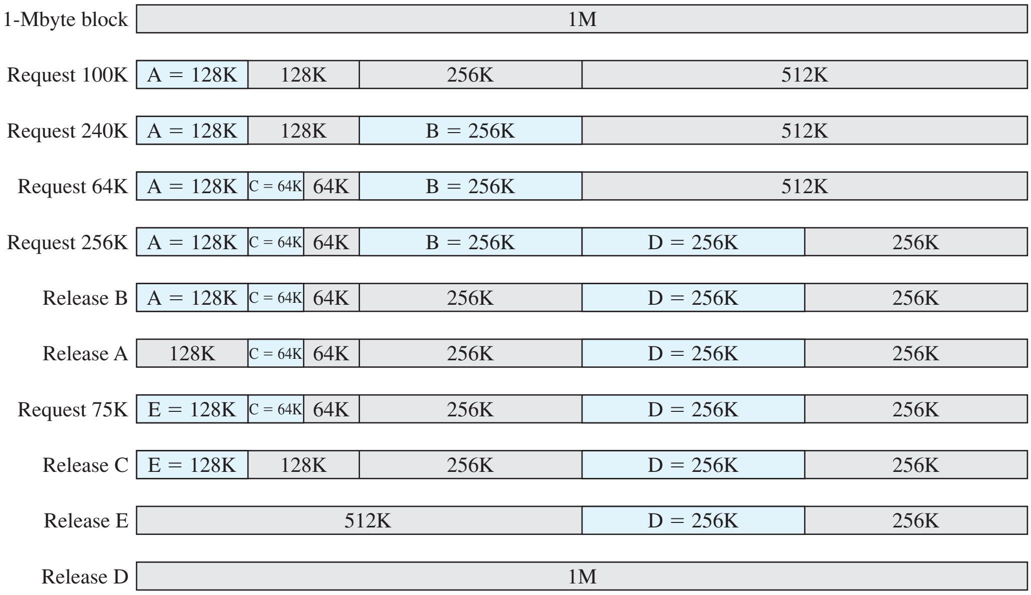

1-Mbyte block 1M
Request 100K A = 128K 128K 256K 512K
Request 240K A = 128K 128K B = 256K 512K
Request 64K A = 128K C = 64K 64K B = 256K 512K
Request 256K A = 128K C = 64K 64K B = 256K D = 256K 256K
Release B A = 128K C = 64K 64K 256K D = 256K 256K
Release A 128K C = 64K 64K 256K D = 256K 256K
Request 75K E = 128K C = 64K 64K 256K D = 256K 256K
Release C E = 128K 128K 256K D = 256K 256K
Release E 512K D = 256K 256K
Release D 1M

Figure 7.6 Example of Buddy System

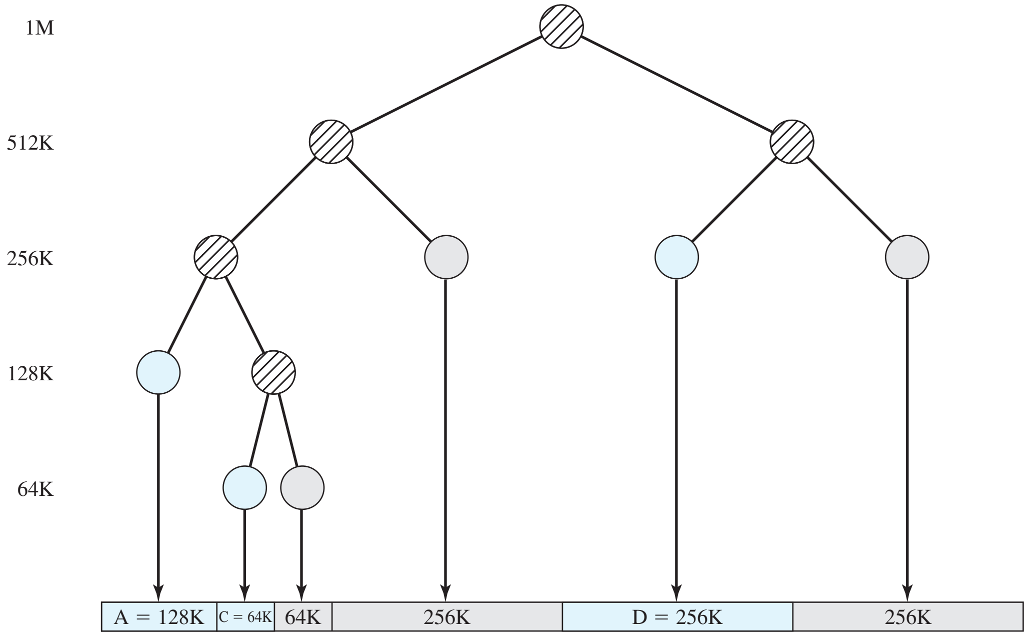


Figure 7.7 Free Representation of Buddy System

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 that 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.

In the case of equal-size partitions (Figure 7.2), and in the case of a single process queue for unequal-size partitions (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 Figures 7.4c and h 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 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

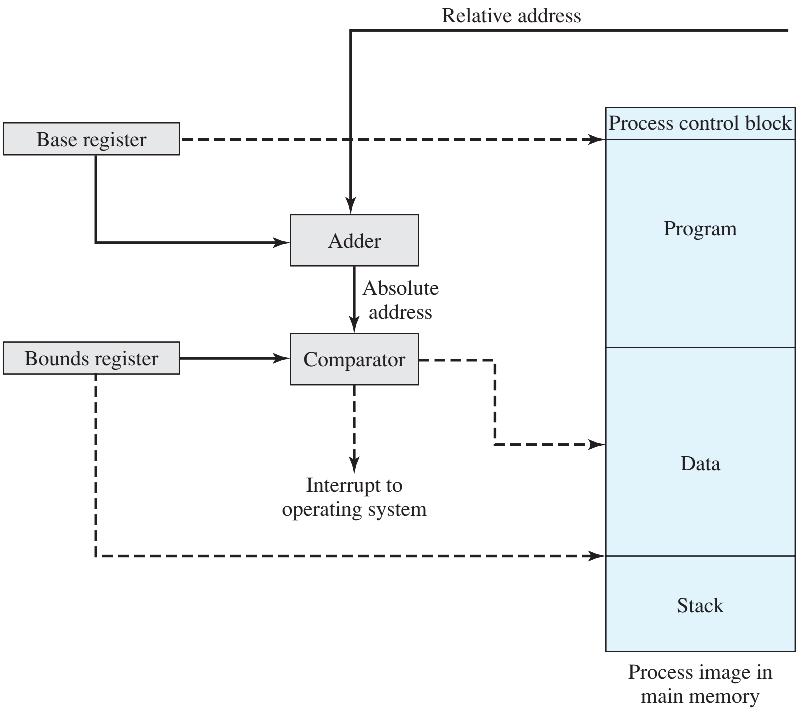


Figure 7.8 Hardware Support for Relocation

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.

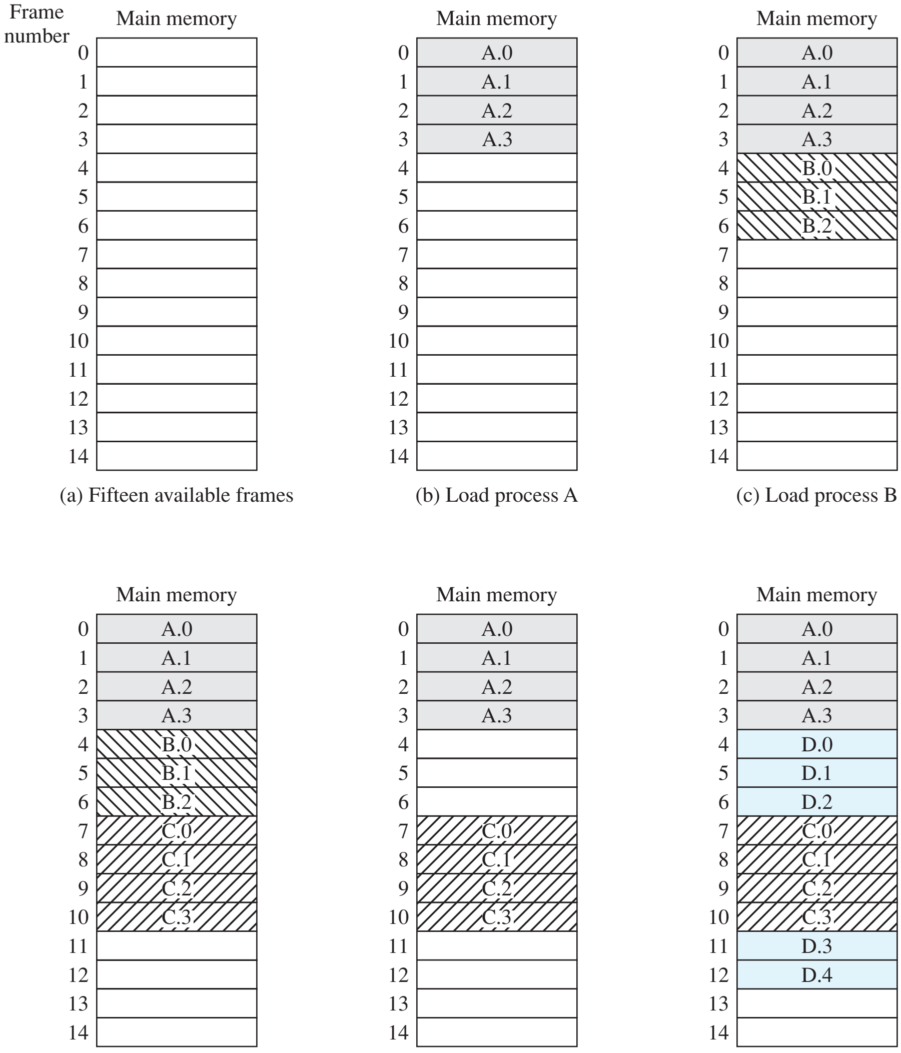
The scheme of Figure 7.8 allows programs to be swapped in and out of memory during the course of execution. It also provides a measure of protection: Each process image is isolated by the contents of the base and bounds registers and safe from unwanted accesses by other processes.

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, that main memory is partitioned into equal fixed-size chunks that are relatively small, and that 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 operating system. Process A, stored on disk, consists of four pages. When it comes time to load this process, the operating system finds four free frames and loads the four pages of process A into the four frames (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 operating system needs to bring in a new process, process D, which consists of five pages.

Now suppose, as in this example, that 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).



(a) Fifteen available frames

(b) Load process A

(c) Load process B

(d) Load process C

(e) Swap out B

(f) Load process D

Figure 7.9 Assignment of Process to Free Frames

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 that 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 operating system maintains a single free-frame list of all 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.

0	0
1	1
2	2
3	3

Process A
page table

0	—
1	—
2	—

Process B
page table

0	7
1	8
2	9
3	10

Process C
page table

0	4
1	5
2	6
3	11
4	12

Process D
page table

13
14

Free frame
list

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

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 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 1K bytes 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

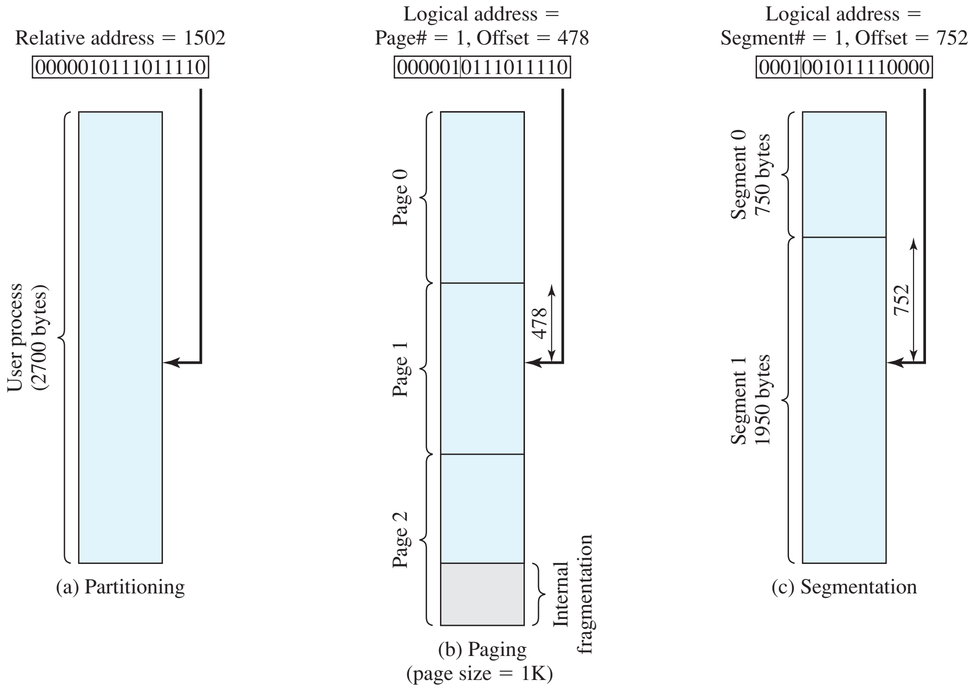
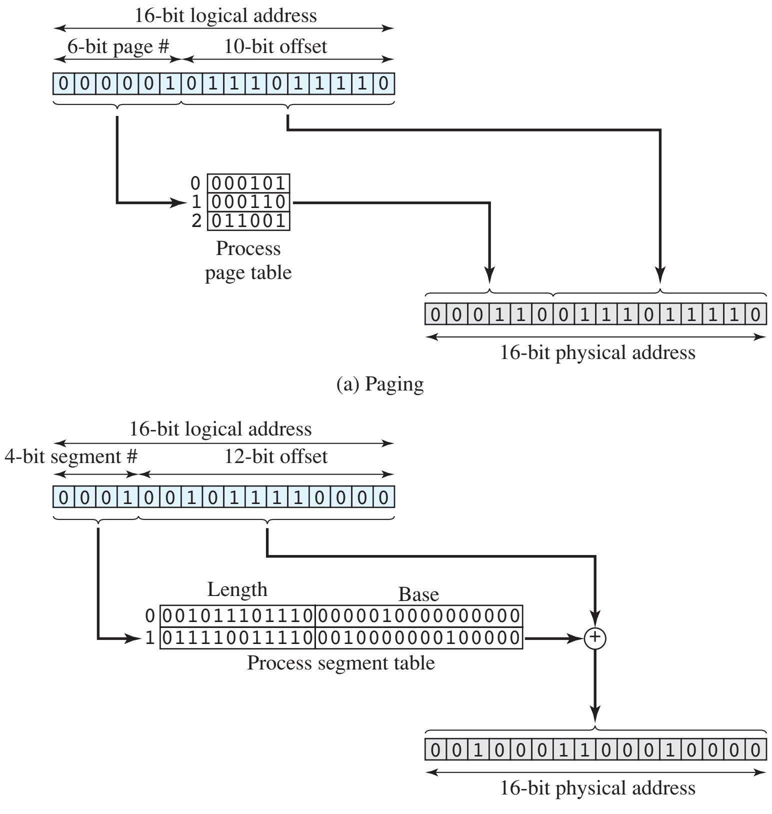


Figure 7.11 **Logical Addresses**

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 (Figure 7.11b), $n = 6$ and $m = 10$. The following steps are needed for address translation:

- Extract the page number as the leftmost n bits of the logical address.
- Use the page number as an index into the process page table to find the frame number, k.
- 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.

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



(a) Paging

(b) Segmentation

Figure 7.12 Examples of Logical-to Physical Address Translation

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 (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:

- Extract the segment number as the leftmost n bits of the logical address.
- Use the segment number as an index into the process segment table to find the starting physical address of the segment.
- 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.
- 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 that this segment is residing in main memory starting at physical address 0010000000100000. Then the physical address is 0010000000100000 + 001011110000 = 0010001100010000 (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 SECURITY ISSUES

Main memory and virtual memory are system resources subject to security threats and for which security countermeasures need to be taken. The most obvious security requirement is the prevention of unauthorized access to the memory contents of processes. If a process has not declared a portion of its memory to be sharable, then no other process should have access to the contents of that portion of memory. If a process declares that a portion of memory may be shared by other designated processes, then the security service of the OS must ensure that only the designated processes have access. The security threats and countermeasures discussed in Chapter 3 are relevant to this type of memory protection.

In this section, we summarize another threat that involves memory protection. Part Seven provides more detail.

Buffer Overflow Attacks

One serious security threat related to memory management remains to be introduced: **buffer overflow**, also known as a **buffer overrun,** which is defined in the NIST (National Institute of Standards and Technology) *Glossary of Key Information Security Terms* as follows:

> **buffer overrun:** A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.

A buffer overflow can occur as a result of a programming error when a process attempts to store data beyond the limits of a fixed-sized buffer and consequently overwrites adjacent memory locations. These locations could hold other program variables or parameters or program control flow data such as return addresses and pointers to previous stack frames. The buffer could be located on the stack, in the heap, or in the data section of the process. The consequences of this error include corruption of data used by the program, unexpected transfer of control in the program, possibly memory access violations, and very likely eventual program termination. When done deliberately as part of an attack on a system, the transfer of control could be to code of the attacker's choosing, resulting in the ability to execute arbitrary code with the privileges of the attacked process. Buffer overflow attacks are one of the most prevalent and dangerous types of security attacks.

To illustrate the basic operation of a common type of buffer overflow, known as **stack overflow**, consider the C main function given in Figure 7.13a. This contains three variables (`valid`, `str1`, and `str2`),[2] whose values will typically be saved in adjacent memory locations. Their order and location depends on the type of variable (local or global), the language and compiler used, and the target machine architecture. For this example, we assume that they are saved in consecutive memory locations, from highest to lowest, as shown in Figure 7.14.[3] This is typically the case for local variables in a C function on common processor architectures such as the Intel Pentium family. The purpose of the code fragment is to call the function `next_tag(str1)` to copy into `str1` some expected tag value. Let's assume this will be the string `START`. It then reads the next line from the standard input for the program using the C library `gets()` function, and then compares the string read with the expected tag. If the next line did indeed contain just the string `START`, this comparison would succeed, and the variable `valid` would be set to `TRUE`.[4] This

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
   printf(“buffer1: str1(%s), str2(%s), valid(%d)\n”, str1, str2, valid);
}
```

(a) Basic buffer overflow C code

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

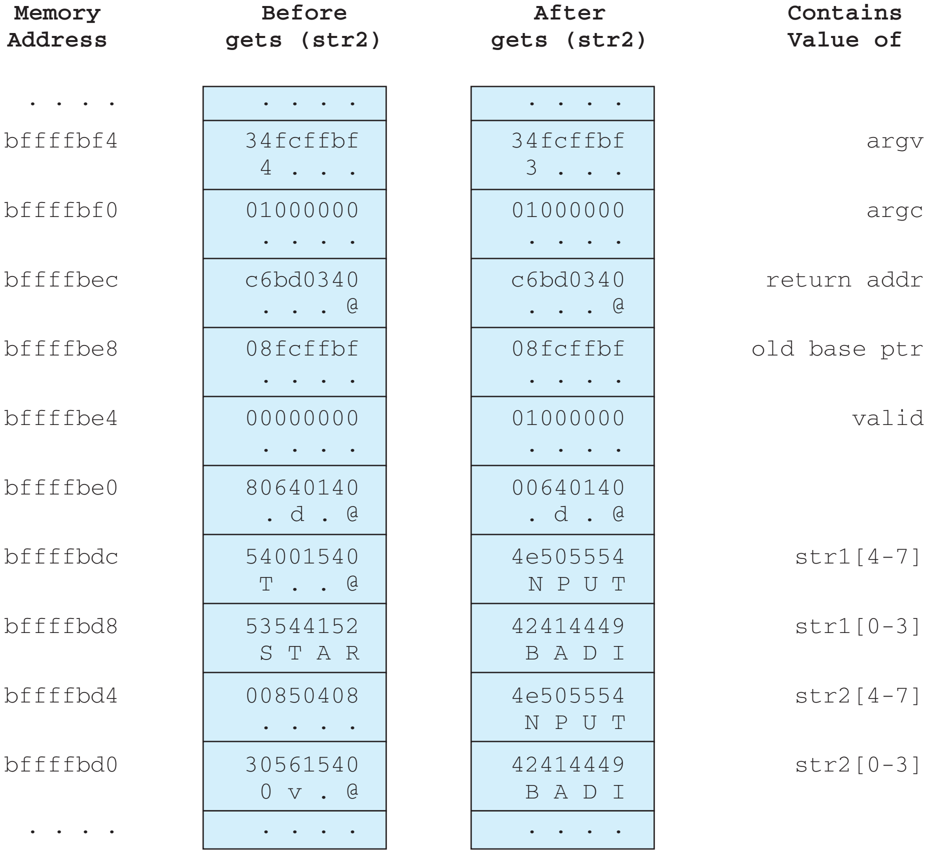
(b) Basic buffer overflow example runs

Figure 7.13 Basic Buffer Overflow Example

[2] In this example, the flag variable is saved as an integer rather than a Boolean. This is done both because it is the classic C style and to avoid issues of word alignment in its storage. The buffers are deliberately small to accentuate the buffer overflow issue being illustrated.

[3] Address and data values are specified in hexadecimal in this and related figures. Data values are also shown in ASCII where appropriate.

[4] In C the logical values FALSE and TRUE are simply integers with the values 0 and 1 (or indeed any nonzero value), respectively. Symbolic defines are often used to map these symbolic names to their underlying value, as was done in this program.



Memory Address	Before gets (str2)	After gets (str2)	Contains Value of
. . . .	. . . .	. . . .	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000	01000000	argc
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bffffbe8	08fcffbf	08fcffbf	old base ptr
bffffbe4	00000000	01000000	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 v . @	42414449 B A D I	str2[0-3]
. . . .	. . . .	. . . .	

Figure 7.14 **Basic Buffer Overflow Stack Values**

case is shown in the first of the three example program runs in Figure 7.13b. Any other input tag would leave it with the value FALSE. Such a code fragment might be used to parse some structured network protocol interaction or formatted text file.

The problem with this code exists because the traditional C library gets() function does not include any checking on the amount of data copied. It reads the next line of text from the program's standard input up until the first newline[5] character occurs and copies it into the supplied buffer followed by the NULL terminator used with C strings.[6] If more than seven characters are present on the input line, when read in they will (along with the terminating NULL character) require more room than is available in the str2 buffer. Consequently, the extra characters will

[5] The newline (NL) or linefeed (LF) character is the standard end of line terminator for UNIX systems, and hence for C, and is the character with the ASCII value 0x0a.

[6] Strings in C are stored in an array of characters and terminated with the NULL character, which has the ASCII value 0x00. Any remaining locations in the array are undefined, and typically contain whatever value was previously saved in that area of memory. This can be clearly seen in the value in the variable str2 in the "Before" column of Figure 7.14.

overwrite the values of the adjacent variable, str1 in this case. For example, if the input line contained EVILINPUTVALUE, the result will be that str1 will be overwritten with the characters TVALUE, and str2 will use not only the eight characters allocated to it but seven more from str1 as well. This can be seen in the second example run in Figure 7.13b. The overflow has resulted in corruption of a variable not directly used to save the input. Because these strings are not equal, valid also retains the value FALSE. Further, if 16 or more characters were input, additional memory locations would be overwritten.

The preceding example illustrates the basic behavior of a buffer overflow. At its simplest, any unchecked copying of data into a buffer could result in corruption of adjacent memory locations, which may be other variables, or possibly program control addresses and data. Even this simple example could be taken further. Knowing the structure of the code processing it, an attacker could arrange for the overwritten value to set the value in str1 equal to the value placed in str2, resulting in the subsequent comparison succeeding. For example, the input line could be the string BADINPUTBADINPUT. This results in the comparison succeeding, as shown in the third of the three example program runs in Figure 7.13b, and illustrated in Figure 7.14, with the values of the local variables before and after the call to gets(). Note also that the terminating NULL for the input string was written to the memory location following str1. This means the flow of control in the program will continue as if the expected tag was found, when in fact the tag read was something completely different. This will almost certainly result in program behavior that was not intended. How serious this is depends very much on the logic in the attacked program. One dangerous possibility occurs if instead of being a tag, the values in these buffers were an expected and supplied password needed to access privileged features. If so, the buffer overflow provides the attacker with a means of accessing these features without actually knowing the correct password.

To exploit any type of buffer overflow, such as those we have illustrated here, the attacker needs

1. To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attackers control, and
2. To understand how that buffer will be stored in the processes memory, and hence the potential for corrupting adjacent memory locations and potentially altering the flow of execution of the program.

Identifying vulnerable programs may be done by inspection of program source, tracing the execution of programs as they process oversized input, or using tools such as *fuzzing*, which we discuss in Part Seven, to automatically identify potentially vulnerable programs. What the attacker does with the resulting corruption of memory varies considerably, depending on what values are being overwritten.

Defending against Buffer Overflows

Finding and exploiting a stack buffer overflow is not that difficult. The large number of exploits over the previous couple of decades clearly illustrates this. There is consequently a need to defend systems against such attacks by either preventing them, or at least detecting and aborting such attacks. Countermeasures can be broadly classified into two categories:

- Compile-time defenses, which aim to harden programs to resist attacks in new programs
- Run-time defenses, which aim to detect and abort attacks in existing programs

While suitable defenses have been known for a couple of decades, the very large existing base of vulnerable software and systems hinders their deployment. Hence the interest in run-time defenses, which can be deployed in operating systems and updates and can provide some protection for existing vulnerable programs.

7.6 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.

7.7 RECOMMENDED READING

Because partitioning has been supplanted by virtual memory techniques, most OS books offer only cursory coverage. One of the more complete and interesting treatments is in [MILE92]. A thorough discussion of partitioning strategies is found in [KNUT97].

The topics of linking and loading are covered in many books on program development, computer architecture, and operating systems. A particularly detailed treatment is [BECK97]. [CLAR98] also contains a good discussion. A thorough practical discussion of this topic, with numerous OS examples, is [LEVI00].

BECK97 Beck, L. *System Software.* Reading, MA: Addison-Wesley, 1997.

CLAR98 Clarke, D., and Merusi, D. *System Software Programming: The Way Things Work.* Upper Saddle River, NJ: Prentice Hall, 1998.

KNUT97 Knuth, D. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Reading, MA: Addison-Wesley, 1997.

LEVI00 Levine, J. *Linkers and Loaders.* San Francisco: Morgan Kaufmann, 2000.

MILE92 Milenkovic, M. *Operating Systems: Concepts and Design.* New York: McGraw-Hill, 1992.

7.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

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

Review Questions

7.1 What requirements is memory management intended to satisfy?

7.2 Why is the capability to relocate processes desirable?

7.3 Why is it not possible to enforce memory protection at compile time?

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 are the distinctions among logical, relative, and physical addresses?

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.

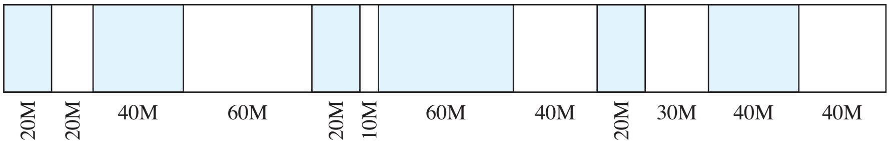
7.2 Consider a fixed partitioning scheme with equal-size partitions of 2^{16} bytes and a total main memory size of 2^{24} bytes. A process table is maintained that includes a pointer to a partition for each resident process. How many bits are required for the pointer?

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 (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. Discuss the pros and cons of this method compared to first-, next-, and best-fit. What is the average length of the search for worst-fit?

7.6 A dynamic partitioning scheme is being used, and the following is the memory configuration at a given point in time:



The shaded areas are allocated blocks; the white areas are free blocks. The next three memory requests are for 40M, 20M, and 10M. Indicate the starting address for each of the three blocks using the following placement algorithms:

a. First-fit
b. Best-fit
c. Next-fit. Assume the most recently added block is at the beginning of memory.
d. Worst-fit

7.7 A 1-Mbyte block of memory is allocated using the buddy system.

a. Show the results of the following sequence in a figure similar to Figure 7.6: Request 70; Request 35; Request 80; Return A; Request 60; Return B; Return D; Return C.
b. Show the binary tree representation following Return B.

7.8 Consider a buddy system in which a particular block under the current allocation has an address of 011011110000.

a. If the block is of size 4, what is the binary address of its buddy?
b. If the block is of size 16, what is the binary address of its buddy?

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

7.10 The Fibonacci sequence is defined as follows:

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

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:

- 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.
- 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 simple paging system with the following parameters: 2^{32} bytes of physical memory; page size of 2^{10} bytes; 2^{16} pages of logical address space.

a. How many bits are in a logical address?
b. How many bytes in a frame?
c. How many bits in the physical address specify the frame?
d. How many entries in the page table?
e. How many bits in each page table entry? Assume each page table entry contains a valid/invalid bit.

7.13 A virtual address a in a paging system is equivalent to a pair (p, w), in which p is a page number and w is a byte number within the page. Let z be the number of bytes in a page. Find algebraic equations that show p and w as functions of z and a.

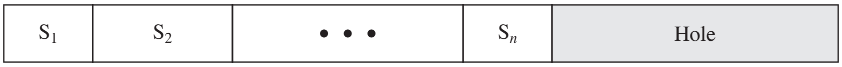
7.14 Consider a simple segmentation system that has the following segment table:

Starting Address	Length (bytes)
660	248
1752	422
222	198
996	604

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

a. 0, 198
b. 2, 156
c. 1, 530
d. 3, 444
e. 0, 222

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:

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

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 $f = 0.2$, $t = 1000$, and $s = 50$.

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 (Figure 7.15). Figure 7.16 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