PART 3 Memory

CHAPTER 7

MEMORY MANAGEMENT

ISBN 1-256-52023-3

I cannot guarantee that I carry all the facts in my mind. Intense mental concentration has a curious way of blotting out what has passed. Each of my cases displaces the last, and Mlle. Carère has blurred my recollection of Baskerville Hall. Tomorrow some other little problem may be submitted to my notice which will in turn dispossess the fair French lady and the infamous Upwood.

—*The Hound of the Baskervilles*, Arthur Conan Doyle.

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.
- Summarize key security issues related to memory management.
- 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 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 with the requirements that memory management is intended to satisfy. Next, we discuss a variety of simple schemes that have been used for memory management

Table 7.1 introduces some key terms for our discussion. A set of animations that illustrate concepts in this chapter is available online. Click on the rotating globe at WilliamStallings.com/OS/OS7e.html for access.

ISBN 1-256-52023-3

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 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).

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
- 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

ISBN 1-256-52023-3

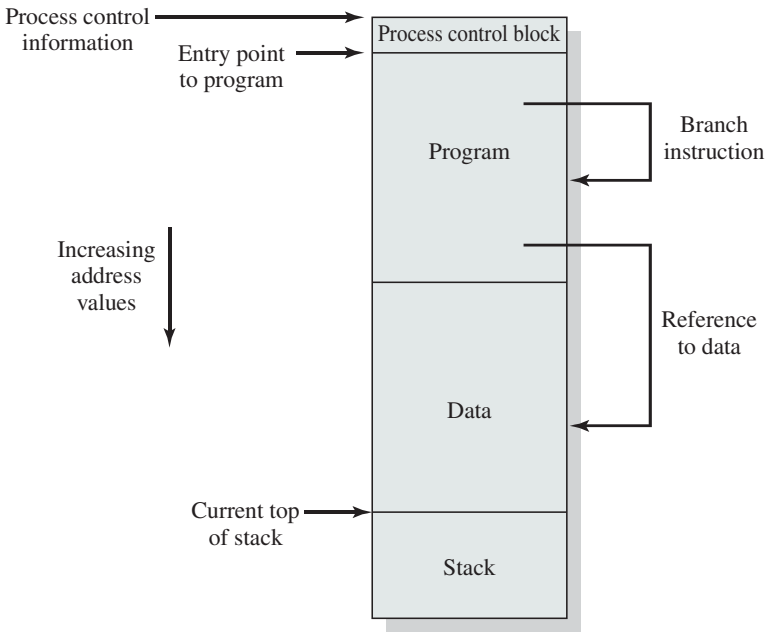


Figure 7.1 Addressing Requirements for a Process

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 (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 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.

ISBN 1-256-52023-3

Note that 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 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

ISBN 1-256-52023-3

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 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 that the OS 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

ISBN 1-256-52023-3

Table 7.2 Memory Management Techniques

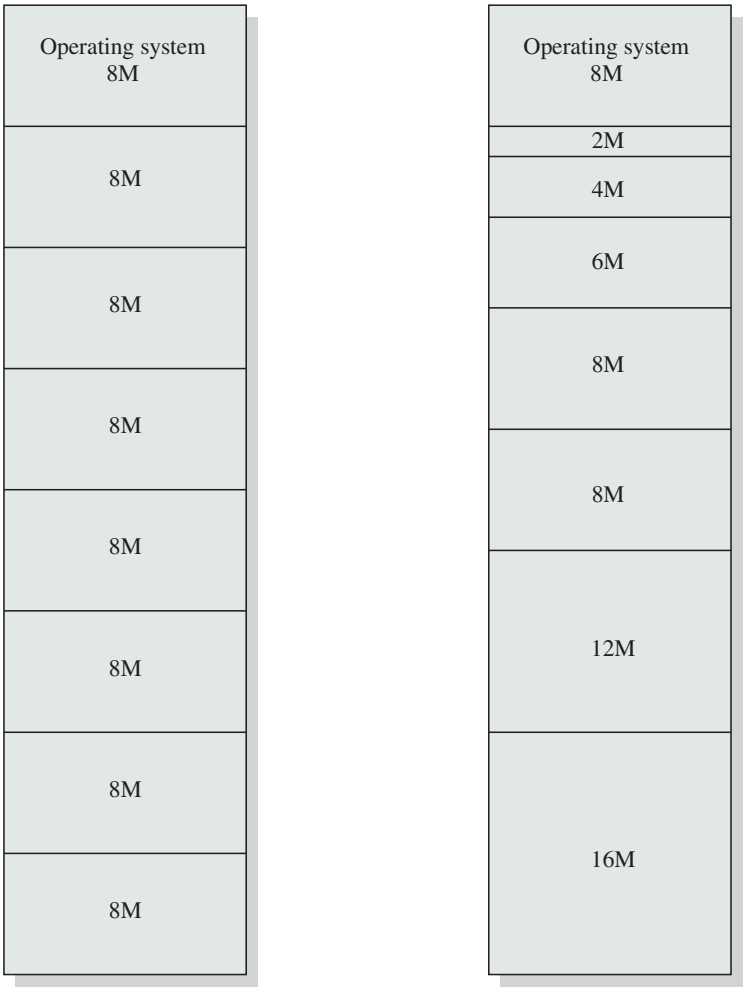
Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to 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.

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

ISBN 1-256-52023-3



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

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

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 (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.

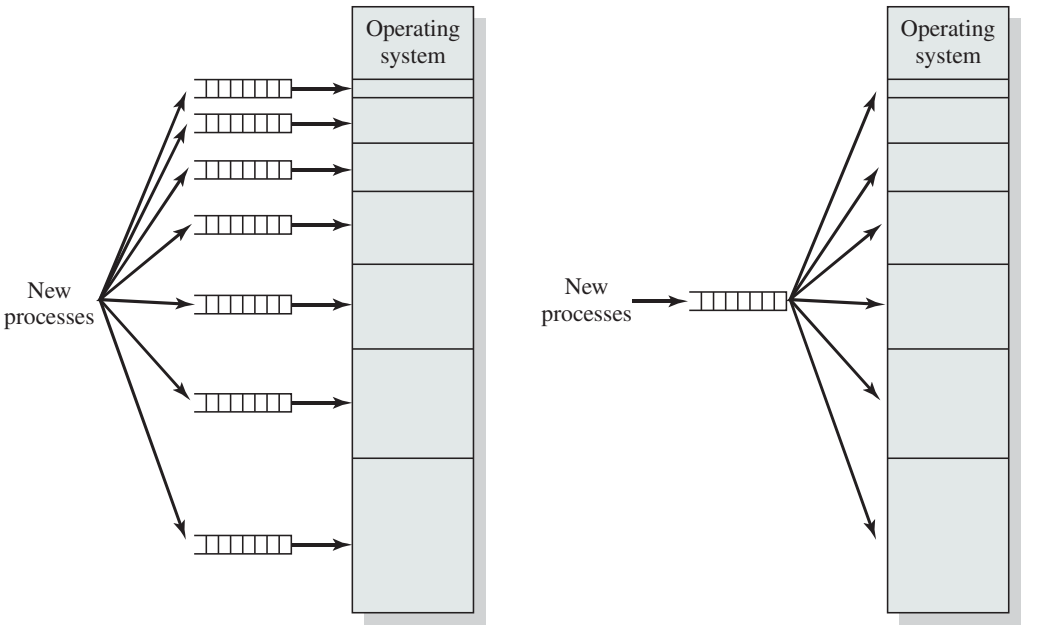
Placement Algorithm With equal-size partitions, the placement of processes in memory is trivial. As long as there is any available partition, a process can be

ISBN 1-256-52023-3

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 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



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

Figure 7.3 Memory Assignment for Fixed Partitioning

[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.

ISBN 1-256-52023-3

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 (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 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 OS 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

ISBN 1-256-52023-3

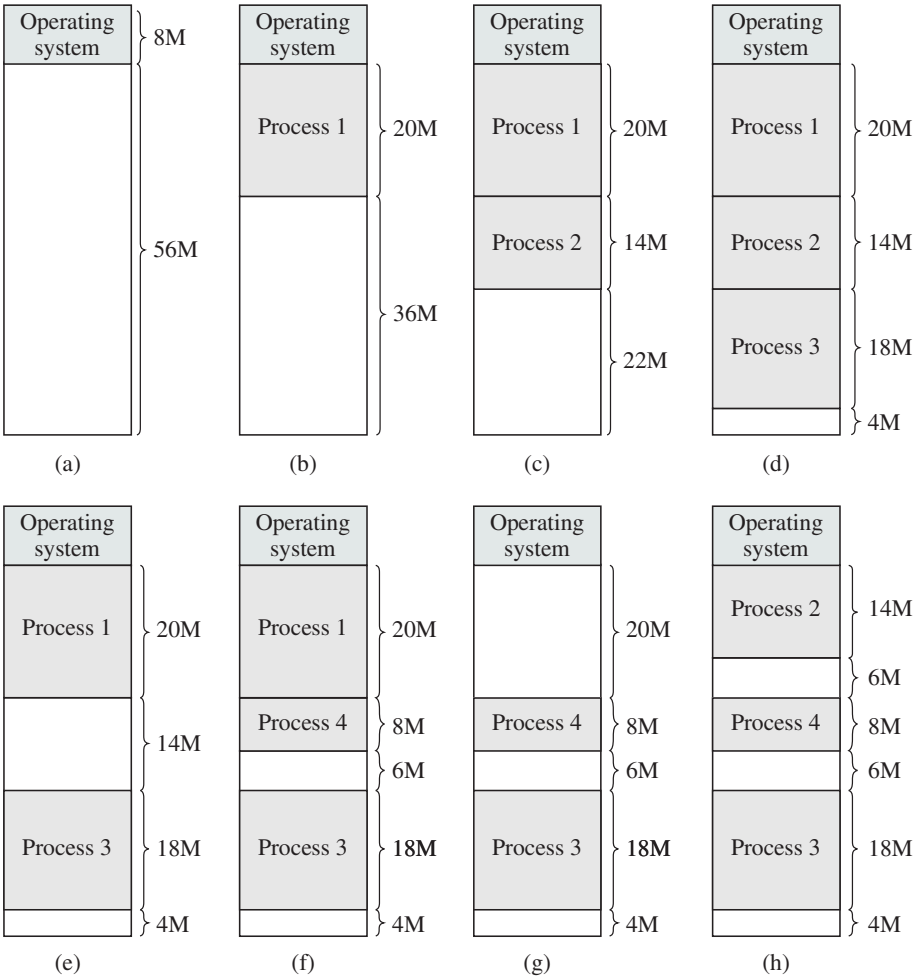


Figure 7.4 The Effect of Dynamic Partitioning

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

ISBN 1-256-52023-3

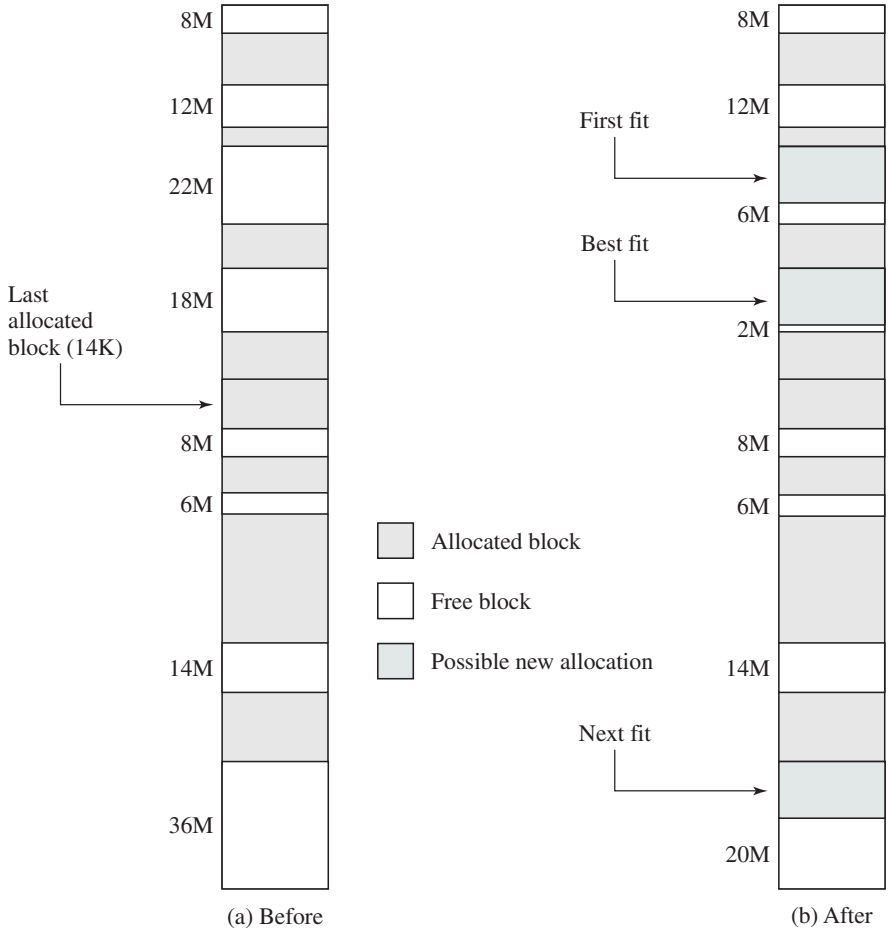


Figure 7.5 **Example Memory Configuration before and after Allocation of 16-Mbyte Block**

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

ISBN 1-256-52023-3

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 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 $(i + 1)$

ISBN 1-256-52023-3

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 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.

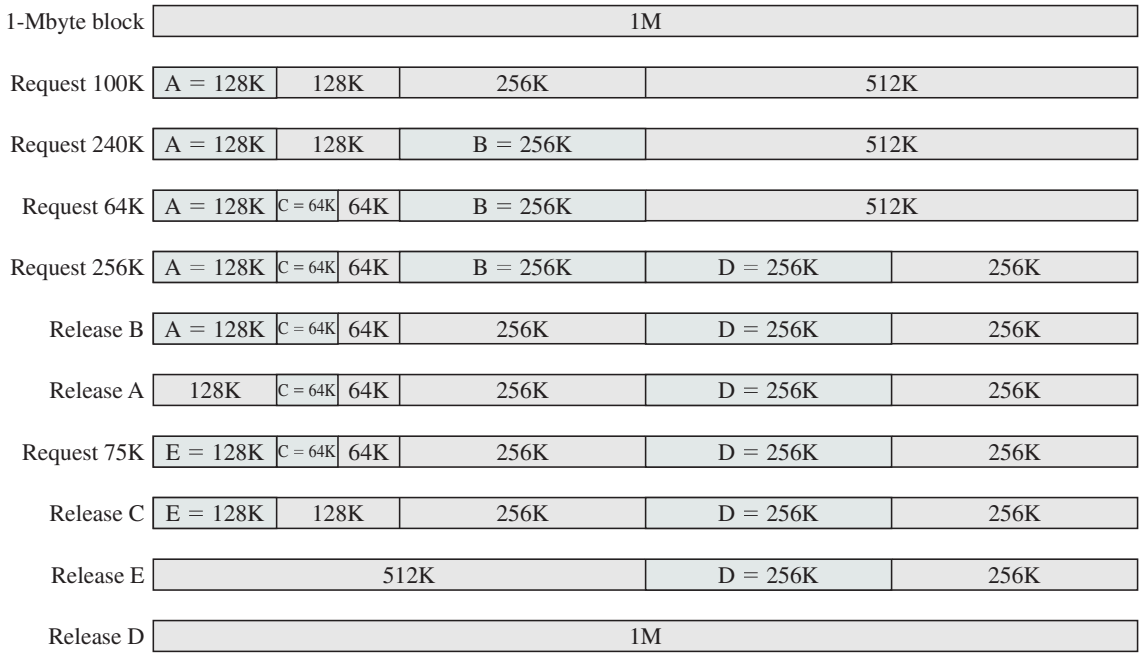


Figure 7.6 Example of Buddy System

ISBN 1-256-52023-3

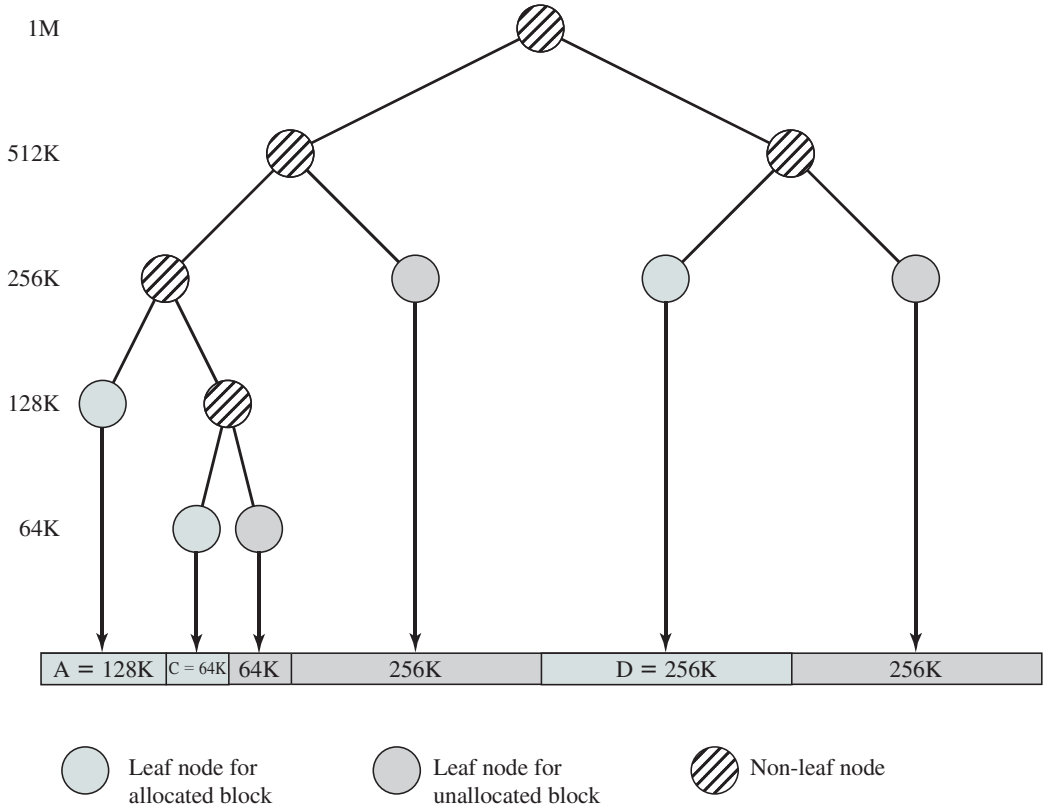


Figure 7.7 Tree Representation of Buddy System

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

ISBN 1-256-52023-3

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 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

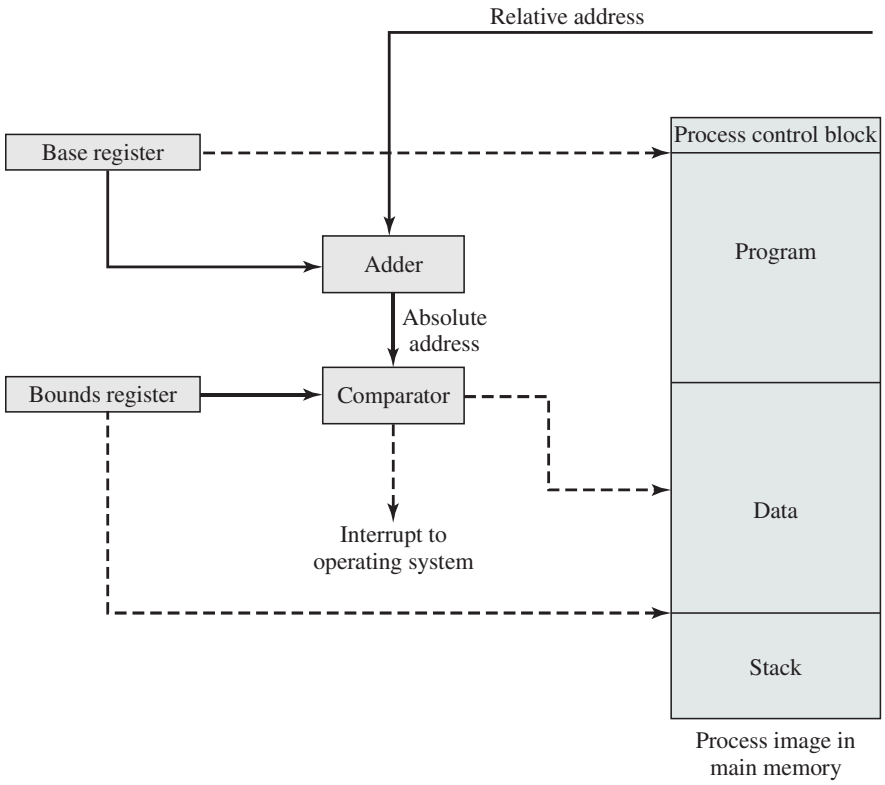


Figure 7.8 Hardware Support for Relocation

ISBN 1-256-52023-3

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 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 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 (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, 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

ISBN 1-256-52023-3

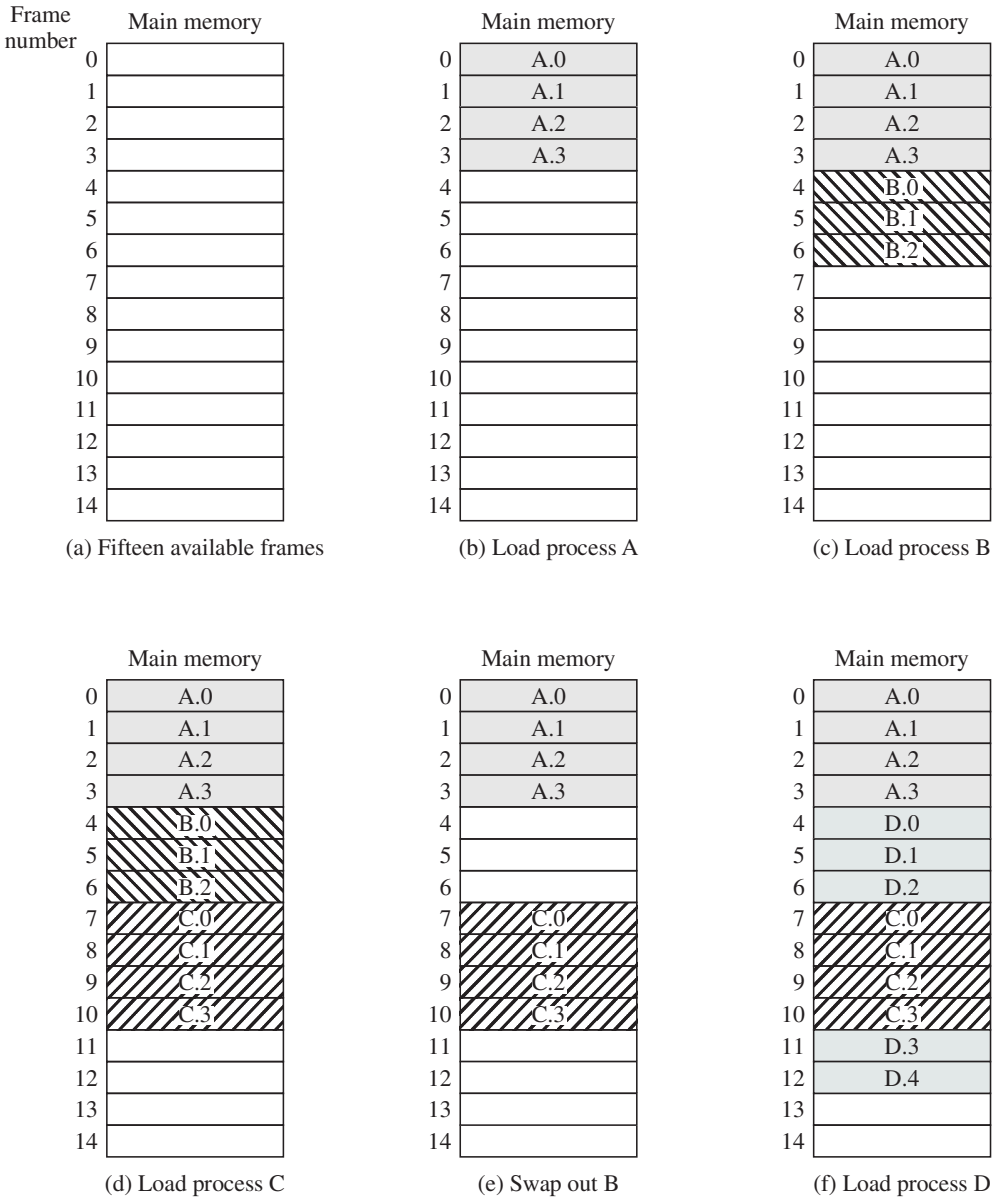


Figure 7.9 Assignment of Process to Free Frames

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 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 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

ISBN 1-256-52023-3

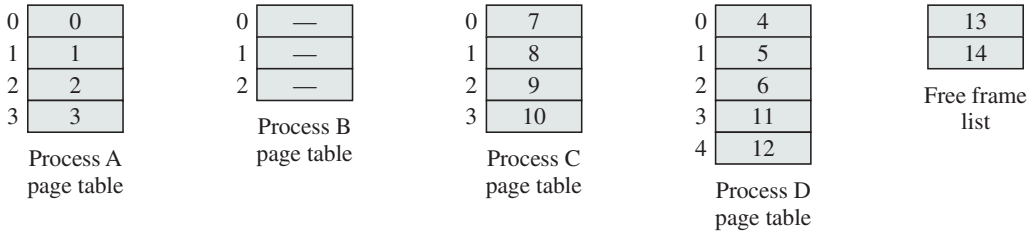


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

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 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 = 1,024 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

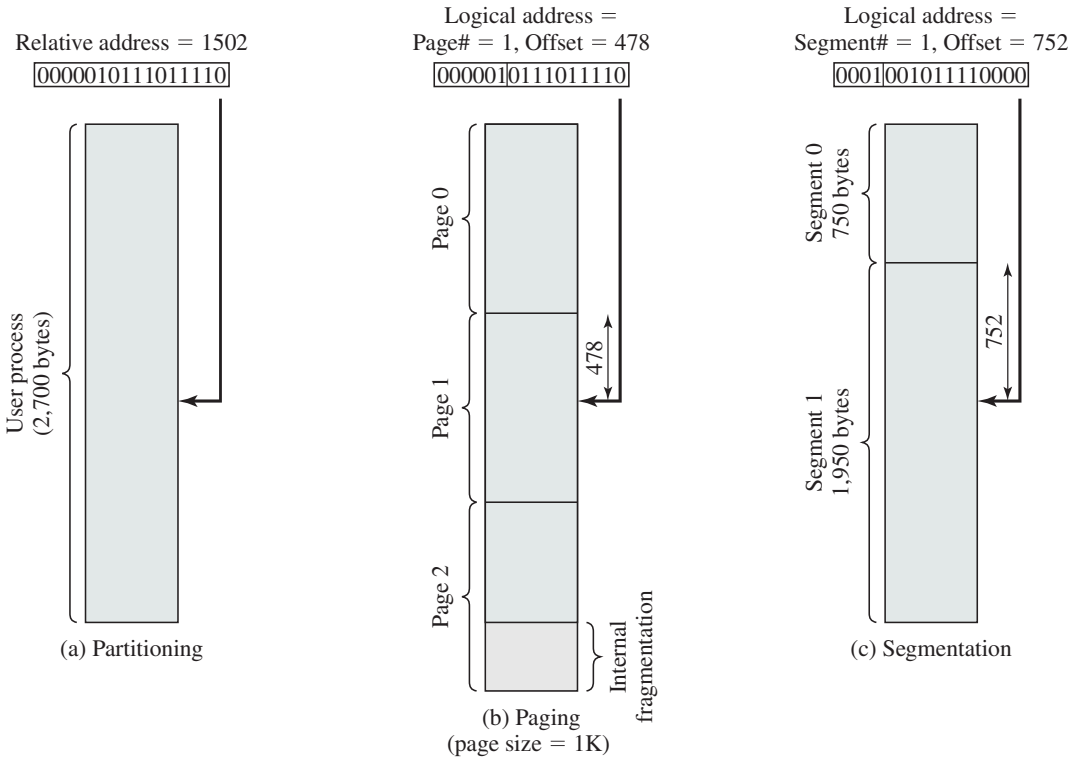


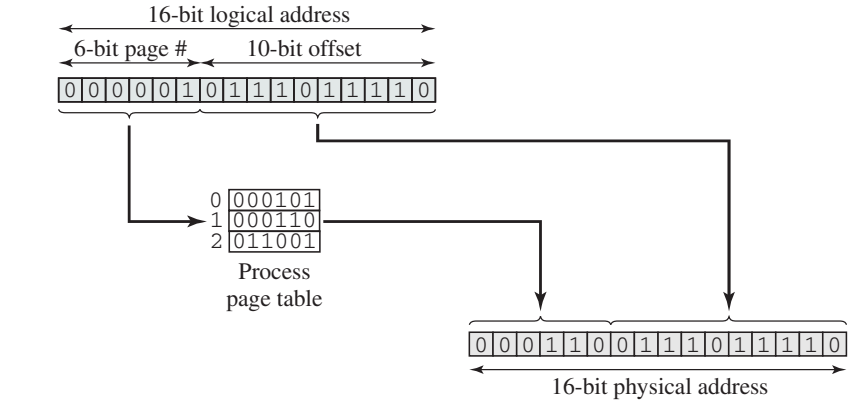
Figure 7.11 Logical Addresses

ISBN 1-256-52023-3

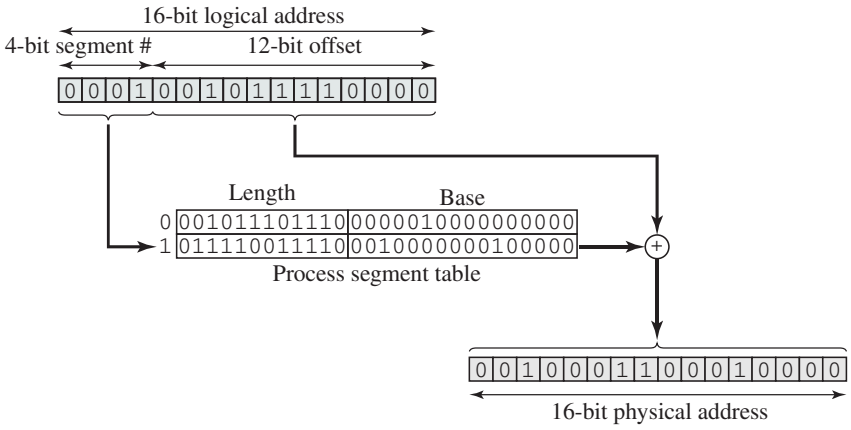
the linker. Each logical address (page number, offset) of a program is identical to its relative address. Second, it is a relatively easy matter to implement a function in hardware to perform dynamic address translation at run time. Consider an address of $n + m$ bits, where the leftmost n bits are the 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

ISBN 1-256-52023-3

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.

ISBN 1-256-52023-3

- 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

ISBN 1-256-52023-3

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.

```
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.

ISBN 1-256-52023-3

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

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 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

[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.

[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.

ISBN 1-256-52023-3

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 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

[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.

ISBN 1-256-52023-3

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].

ISBN 1-256-52023-3

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	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 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.

ISBN 1-256-52023-3

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.

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-Mbyte 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, worst-fit. For each algorithm, draw a horizontal segment under the memory strip and label it clearly.

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 $\text{buddy}_k(x)$ = 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:

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

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?

ISBN 1-256-52023-3

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 Write the binary translation of the logical address `0001010010111010` under the following hypothetical memory management schemes, and explain your answer:
- **a.** a paging system with a 256-address page size, using a page table in which the frame number happens to be four times smaller than the page number
- **b.** a segmentation system with a 1K-address maximum segment size, using a segment table in which bases happen to be regularly placed at real addresses: 22 + 4,096 × segment #

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

Starting Address	Length (bytes)
660	248
1,752	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,…,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 S1, S2,…,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 $f = 0.2$, $t = 1{,}000$, and $s = 50$.

ISBN 1-256-52023-3

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 library routines themselves may be incorporated into the program or referenced as shared 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 7.16, 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:

- Absolute loading
- Relocatable loading
- 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

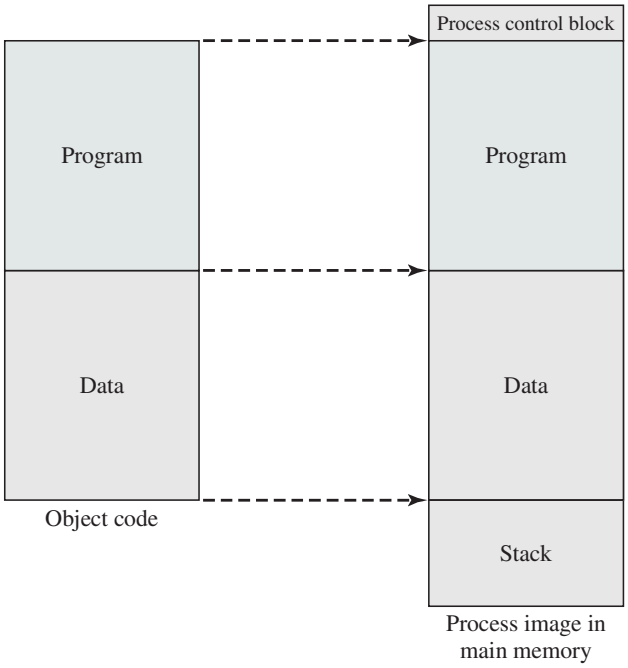


Figure 7.15 The Loading Function

ISBN 1-256-52023-3

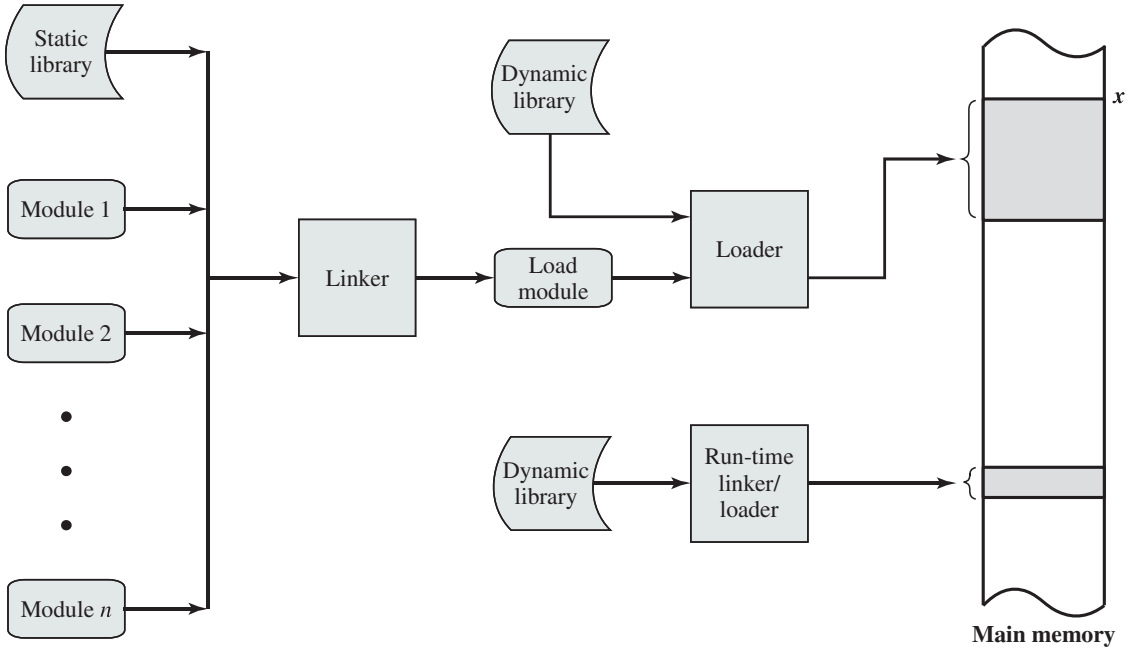


Figure 7.16 **A Linking and Loading Scenario**

memory addresses. For example, if *x* in Figure 7.16 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 (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 to be altered. Accordingly, it is preferable to allow memory references within programs to be expressed symbolically and then resolve those symbolic references at the time of compilation or assembly. This is illustrated in Figure 7.17. 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 7.17b.

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 7.17c. The start of the load module is assigned the relative address 0, and

ISBN 1-256-52023-3

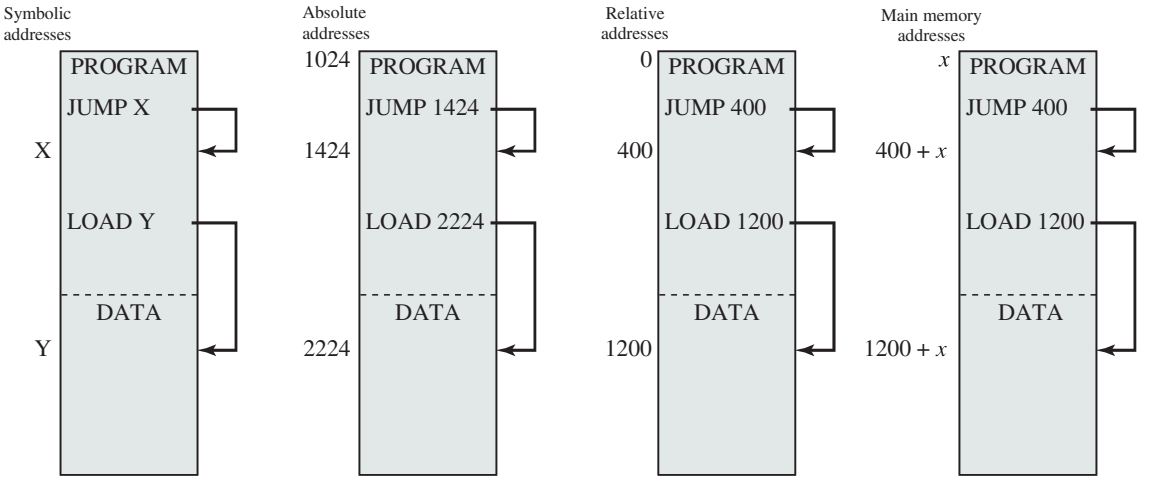
Table 7.3 Address Binding

(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.



(a) Object module (b) Absolute load module (c) Relative load module (d) Relative load module loaded into main memory starting at location x

Figure 7.17 Absolute and Relocatable Load Modules

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 and 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 (Figure 7.17c). 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 7.18a 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.

Linkage Editor The nature of this address linkage will depend on the type of load module to be created and when the linkage occurs (Table 7.3b). If, as is

ISBN 1-256-52023-3

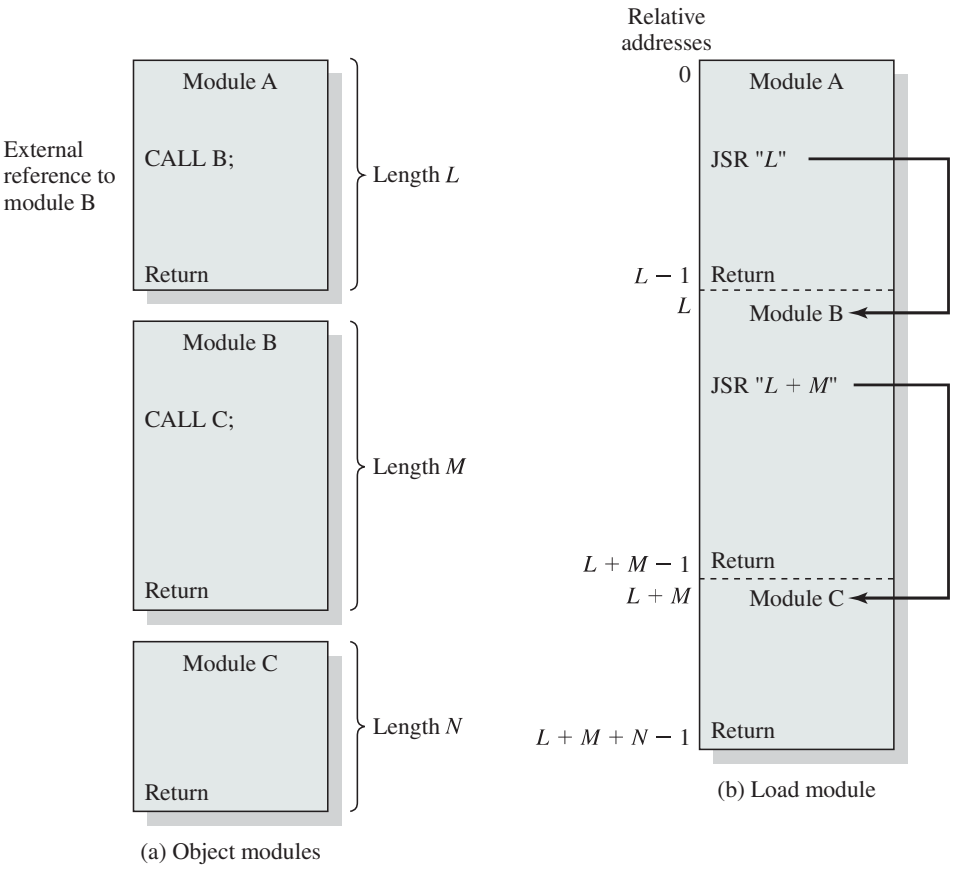


Figure 7.18 The Linking Function

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 7.18 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 upper dynamic library in Figure 7.16), 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:

ISBN 1-256-52023-3

- 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 lower dynamic library in Figure 7.16), 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 call 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.

ISBN 1-256-52023-3

CHAPTER 8

VIRTUAL MEMORY

ISBN 1-256-52023-3

You're gonna need a bigger boat.

—STEVEN SPIELBERG, *JAWS*, 1975

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Define virtual memory.
- Describe the hardware and control structures that support virtual memory.
- Describe the various OS mechanisms used to implement virtual memory.
- Describe the virtual memory management mechanisms in UNIX, Linux, and Windows 7.

Chapter 7 introduced the concepts of paging and segmentation and analyzed their shortcomings. We now move to a discussion of virtual memory. An analysis of this topic is complicated by the fact that memory management is a complex interrelationship between processor hardware and operating system software. We focus first on the hardware aspect of virtual memory, looking at the use of paging, segmentation, and combined paging and segmentation. Then we look at the issues involved in the design of a virtual memory facility in operating systems.

Table 8.1 defines some key terms related to virtual memory. A set of animations that illustrate concepts in this chapter is available online. Click on the rotating globe at WilliamStallings.com/OS/OS7e.html for access.

8.1 HARDWARE AND CONTROL STRUCTURES

Comparing simple paging and simple segmentation, on the one hand, with fixed and dynamic partitioning, on the other, we see the foundation for a fundamental breakthrough in memory management. Two characteristics of paging and segmentation are the keys to this breakthrough:

Table 8.1 Virtual Memory Terminology

Virtual memory	A storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.
Virtual address	The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory.
Virtual address space	The virtual storage assigned to a process.
Address space	The range of memory addresses available to a process.
Real address	The address of a storage location in main memory.

ISBN 1-256-52023-3

1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process may be swapped in and out of main memory such that it occupies different regions of main memory at different times during the course of execution.
2. A process may be broken up into a number of pieces (pages or segments) and these pieces need not be contiguously located in main memory during execution. The combination of dynamic run-time address translation and the use of a page or segment table permits this.

Now we come to the breakthrough. *If the preceding two characteristics are present, then it is not necessary that all of the pages or all of the segments of a process be in main memory during execution.* If the piece (segment or page) that holds the next instruction to be fetched and the piece that holds the next data location to be accessed are in main memory, then at least for a time execution may proceed.

Let us consider how this may be accomplished. For now, we can talk in general terms, and we will use the term *piece* to refer to either page or segment, depending on whether paging or segmentation is employed. Suppose that it is time to bring a new process into memory. The OS begins by bringing in only one or a few pieces, to include the initial program piece and the initial data piece to which those instructions refer. The portion of a process that is actually in main memory at any time is called the **resident set** of the process. As the process executes, things proceed smoothly as long as all memory references are to locations that are in the resident set. Using the segment or page table, the processor always is able to determine whether this is so. If the processor encounters a logical address that is not in main memory, it generates an interrupt indicating a memory access fault. The OS puts the interrupted process in a blocking state. For the execution of this process to proceed later, the OS must bring into main memory the piece of the process that contains the logical address that caused the access fault. For this purpose, the OS issues a disk I/O read request. After the I/O request has been issued, the OS can dispatch another process to run while the disk I/O is performed. Once the desired piece has been brought into main memory, an I/O interrupt is issued, giving control back to the OS, which places the affected process back into a Ready state.

It may immediately occur to you to question the efficiency of this maneuver, in which a process may be executing and have to be interrupted for no other reason than that you have failed to load in all of the needed pieces of the process. For now, let us defer consideration of this question with the assurance that efficiency is possible. Instead, let us ponder the implications of our new strategy. There are two implications, the second more startling than the first, and both lead to improved system utilization:

1. **More processes may be maintained in main memory.** Because we are only going to load some of the pieces of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in a Ready state at any particular time.
2. **A process may be larger than all of main memory.** One of the most fundamental restrictions in programming is lifted. Without the scheme we have been discussing, a programmer must be acutely aware of how much memory is available. If the program being written is too large, the programmer must devise ways to

ISBN 1-256-52023-3

structure the program into pieces that can be loaded separately in some sort of overlay strategy. With virtual memory based on paging or segmentation, that job is left to the OS and the hardware. As far as the programmer is concerned, he or she is dealing with a huge memory, the size associated with disk storage. The OS automatically loads pieces of a process into main memory as required.

Because a process executes only in main memory, that memory is referred to as **real memory.** But a programmer or user perceives a potentially much larger memory—that which is allocated on disk. This latter is referred to as **virtual memory.** Virtual memory allows for very effective multiprogramming and relieves the user of the unnecessarily tight constraints of main memory. Table 8.2 summarizes characteristics of paging and segmentation, with and without the use of virtual memory.

Table 8.2 Characteristics of Paging and Segmentation

Simple Paging	Virtual Memory Paging	Simple Segmentation	Virtual Memory Segmentation
Main memory partitioned into small fixed-size chunks called frames	Main memory partitioned into small fixed-size chunks called frames	Main memory not partitioned	Main memory not partitioned
Program broken into pages by the compiler or memory management system	Program broken into pages by the compiler or memory management system	Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)	Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)
Internal fragmentation within frames	Internal fragmentation within frames	No internal fragmentation	No internal fragmentation
No external fragmentation	No external fragmentation	External fragmentation	External fragmentation
Operating system must maintain a page table for each process showing which frame each page occupies	Operating system must maintain a page table for each process showing which frame each page occupies	Operating system must maintain a segment table for each process showing the load address and length of each segment	Operating system must maintain a segment table for each process showing the load address and length of each segment
Operating system must maintain a free frame list	Operating system must maintain a free frame list	Operating system must maintain a list of free holes in main memory	Operating system must maintain a list of free holes in main memory
Processor uses page number, offset to calculate absolute address	Processor uses page number, offset to calculate absolute address	Processor uses segment number, offset to calculate absolute address	Processor uses segment number, offset to calculate absolute address
All the pages of a process must be in main memory for process to run, unless overlays are used	Not all pages of a process need be in main memory frames for the process to run. Pages may be read in as needed	All the segments of a process must be in main memory for process to run, unless overlays are used	Not all segments of a process need be in main memory for the process to run. Segments may be read in as needed
	Reading a page into main memory may require writing a page out to disk		Reading a segment into main memory may require writing one or more segments out to disk

ISBN 1-256-52023-3

Locality and Virtual Memory

The benefits of virtual memory are attractive, but is the scheme practical? At one time, there was considerable debate on this point, but experience with numerous operating systems has demonstrated beyond doubt that virtual memory does work. Accordingly, virtual memory, based on either paging or paging plus segmentation, has become an essential component of contemporary operating systems.

To understand the key issue and why virtual memory was a matter of much debate, let us examine again the task of the OS with respect to virtual memory. Consider a large process, consisting of a long program plus a number of arrays of data. Over any short period of time, execution may be confined to a small section of the program (e.g., a subroutine) and access to perhaps only one or two arrays of data. If this is so, then it would clearly be wasteful to load in dozens of pieces for that process when only a few pieces will be used before the program is suspended and swapped out. We can make better use of memory by loading in just a few pieces. Then, if the program branches to an instruction or references a data item on a piece not in main memory, a fault is triggered. This tells the OS to bring in the desired piece.

Thus, at any one time, only a few pieces of any given process are in memory, and therefore more processes can be maintained in memory. Furthermore, time is saved because unused pieces are not swapped in and out of memory. However, the OS must be clever about how it manages this scheme. In the steady state, practically all of main memory will be occupied with process pieces, so that the processor and OS have direct access to as many processes as possible. Thus, when the OS brings one piece in, it must throw another out. If it throws out a piece just before it is used, then it will just have to go get that piece again almost immediately. Too much of this leads to a condition known as **thrashing**: The system spends most of its time swapping pieces rather than executing instructions. The avoidance of thrashing was a major research area in the 1970s and led to a variety of complex but effective algorithms. In essence, the OS tries to guess, based on recent history, which pieces are least likely to be used in the near future.

This reasoning is based on belief in the **principle of locality**, which was introduced in Chapter 1 (see especially Appendix 1A). To summarize, the principle of locality states that program and data references within a process tend to cluster. Hence, the assumption that only a few pieces of a process will be needed over a short period of time is valid. Also, it should be possible to make intelligent guesses about which pieces of a process will be needed in the near future, which avoids thrashing.

One way to confirm the principle of locality is to look at the performance of processes in a virtual memory environment. Figure 8.1 is a rather famous diagram that dramatically illustrates the principle of locality [HATF72]. Note that, during the lifetime of the process, references are confined to a subset of pages.

Thus we see that the principle of locality suggests that a virtual memory scheme may work. For virtual memory to be practical and effective, two ingredients are needed. First, there must be hardware support for the paging and/or segmentation scheme to be employed. Second, the OS must include software for managing the movement of pages and/or segments between secondary memory and main memory. In this section, we examine the hardware aspect and look at the

ISBN 1-256-52023-3

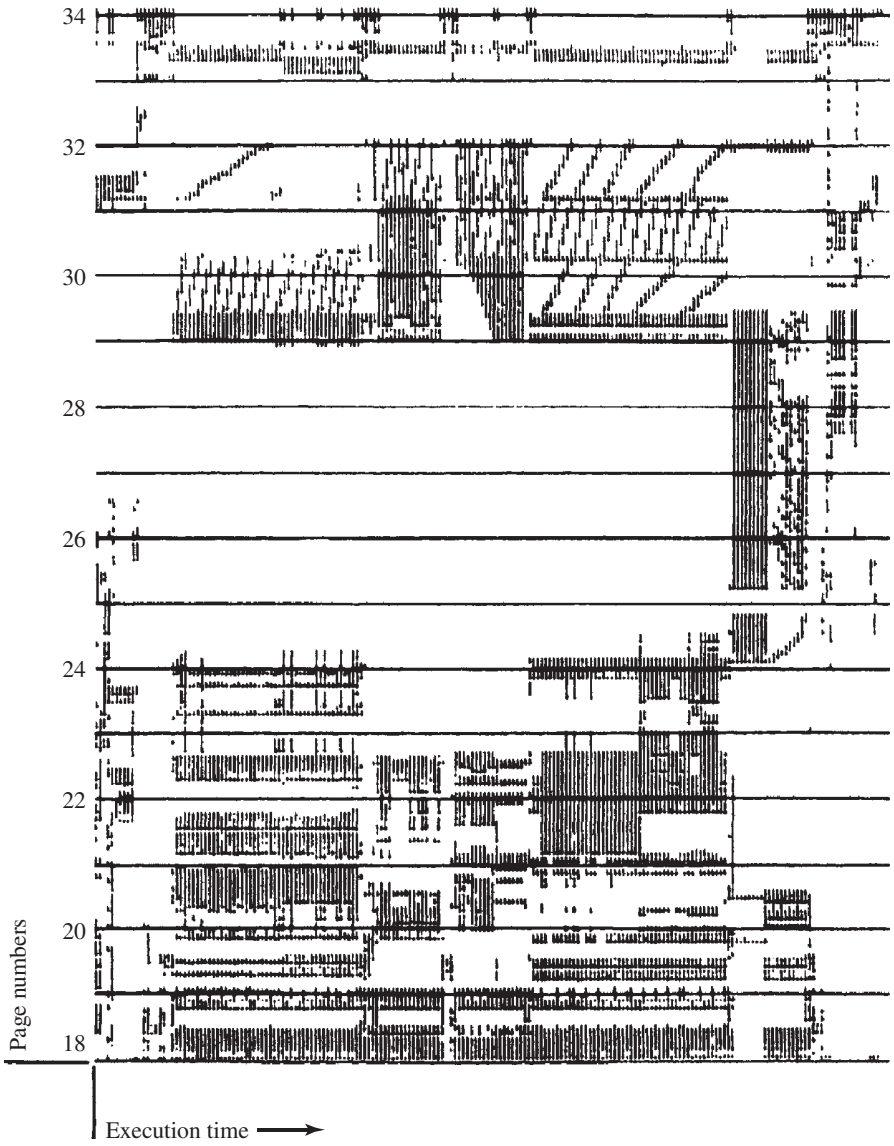


Figure 8.1 Paging Behavior

necessary control structures, which are created and maintained by the OS but are used by the memory management hardware. An examination of the OS issues is provided in the next section.

Paging

The term *virtual memory* is usually associated with systems that employ paging, although virtual memory based on segmentation is also used and is discussed next. The use of paging to achieve virtual memory was first reported for the Atlas computer [KILB62] and soon came into widespread commercial use.

In the discussion of simple paging, we indicated that each process has its own page table, and when all of its pages are loaded into main memory, the page

ISBN 1-256-52023-3

table for a process is created and loaded into main memory. Each page table entry (PTE) contains the frame number of the corresponding page in main memory. A page table is also needed for a virtual memory scheme based on paging. Again, it is typical to associate a unique page table with each process. In this case, however, the page table entries become more complex (Figure 8.2a). Because only some of the pages of a process may be in main memory, a bit is needed in each page table entry to indicate whether the corresponding page is present (P) in main memory or not. If the bit indicates that the page is in memory, then the entry also includes the frame number of that page.

The page table entry includes a modify (M) bit, indicating whether the contents of the corresponding page have been altered since the page was last loaded into main memory. If there has been no change, then it is not necessary to write the page out when it comes time to replace the page in the frame that it currently occupies. Other control bits may also be present. For example, if protection or sharing is managed at the page level, then bits for that purpose will be required.

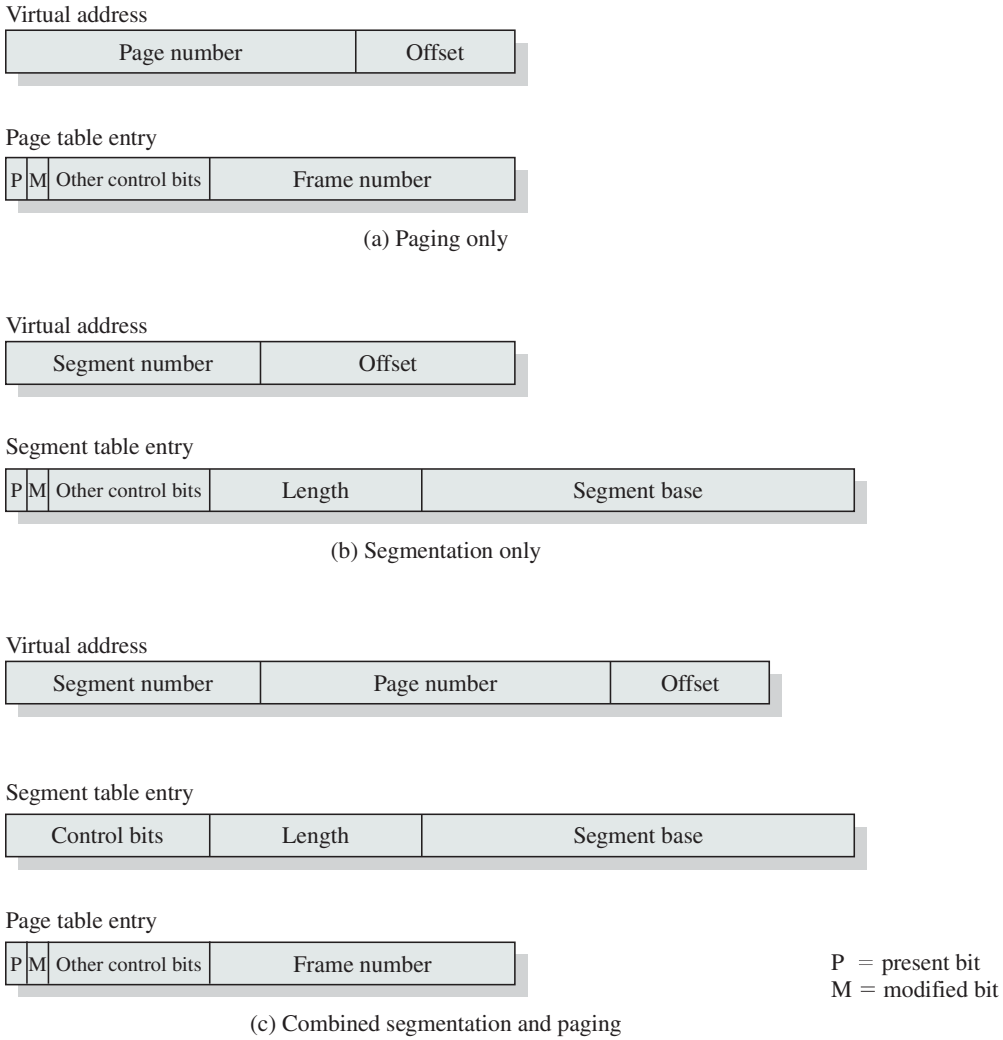


Figure 8.2 Typical Memory Management Formats

ISBN 1-256-52023-3

PAGE TABLE STRUCTURE The basic mechanism for reading a word from memory involves the translation of a virtual, or logical, address, consisting of page number and offset, into a physical address, consisting of frame number and offset, using a page table. Because the page table is of variable length, depending on the size of the process, we cannot expect to hold it in registers. Instead, it must be in main memory to be accessed. Figure 8.3 suggests a hardware implementation. When a particular process is running, a register holds the starting address of the page table for that process. The page number of a virtual address is used to index that table and look up the corresponding frame number. This is combined with the offset portion of the virtual address to produce the desired real address. Typically, the page number field is longer than the frame number field ($n > m$).

In most systems, there is one page table per process. But each process can occupy huge amounts of virtual memory. For example, in the VAX architecture, each process can have up to $2^{31} = 2$ Gbytes of virtual memory. Using $2^9 = 512$-byte pages means that as many as 2^{22} page table entries are required *per process*. Clearly, the amount of memory devoted to page tables alone could be unacceptably high. To overcome this problem, most virtual memory schemes store page tables in virtual memory rather than real memory. This means that page tables are subject to paging just as other pages are. When a process is running, at least a part of its page table must be in main memory, including the page table entry of the currently executing page. Some processors make use of a two-level scheme to organize large page tables. In this scheme, there is a page directory, in which each entry points to a page table. Thus, if the length of the page directory is X, and if the maximum length of a page table is Y, then a process can

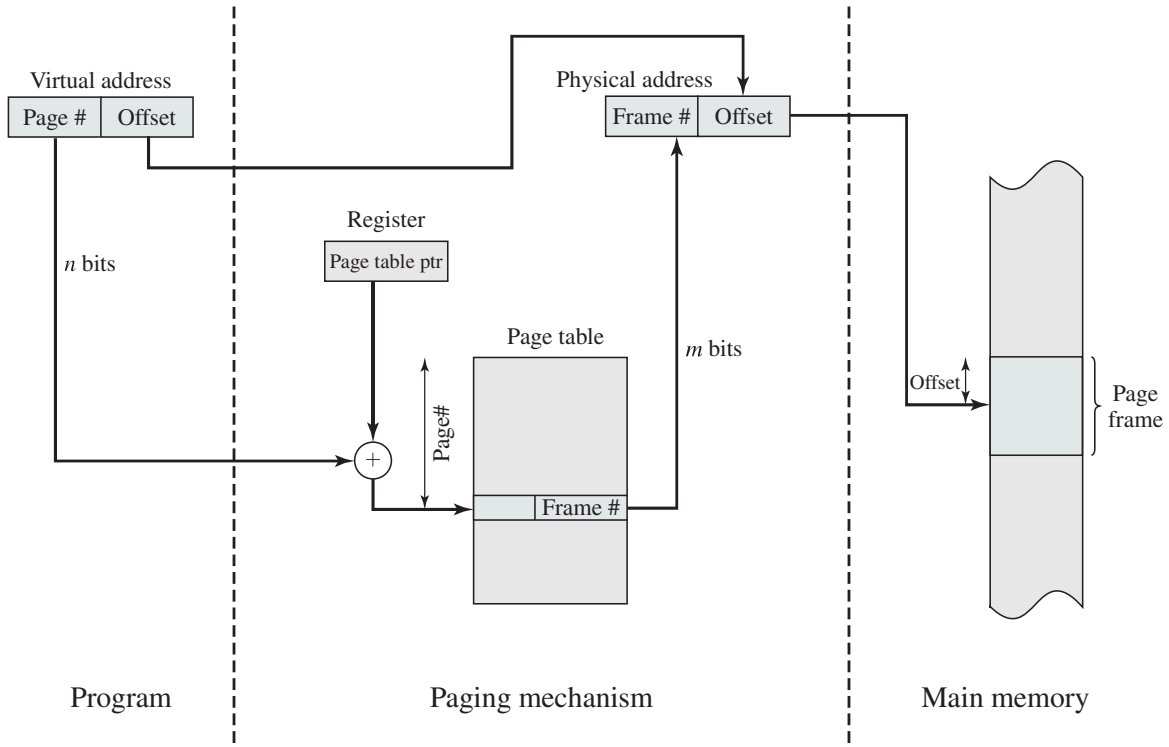


Figure 8.3 Address Translation in a Paging System

ISBN 1-256-52023-3

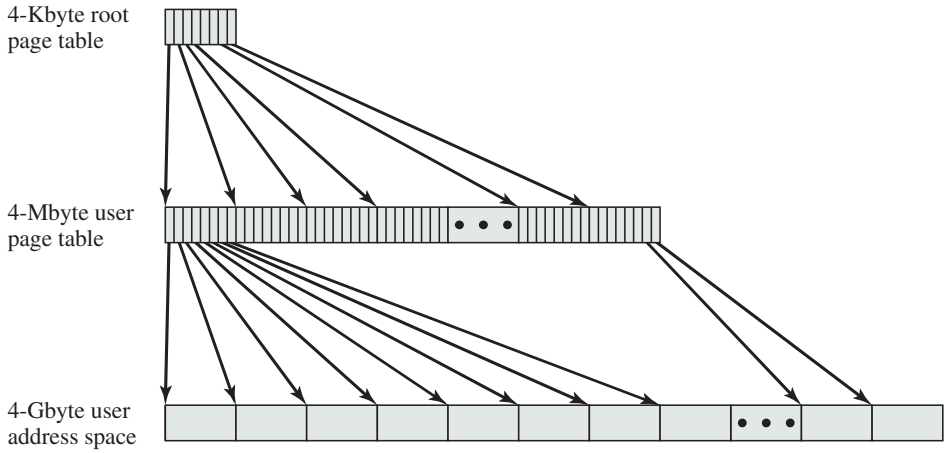


Figure 8.4 **A Two-Level Hierarchical Page Table**

consist of up to $X \times Y$ pages. Typically, the maximum length of a page table is restricted to be equal to one page. For example, the Pentium processor uses this approach.

Figure 8.4 shows an example of a two-level scheme typical for use with a 32-bit address. If we assume byte-level addressing and 4-Kbyte (2^{12}) pages, then the 4-Gbyte (2^{32}) virtual address space is composed of 2^{20} pages. If each of these pages is mapped by a 4-byte page table entry, we can create a user page table composed of 2^{20} PTEs requiring 4 Mbytes (2^{22}). This huge user page table, occupying 2^{10} pages, can be kept in virtual memory and mapped by a root page table with 2^{10} PTEs occupying 4 Kbytes (2^{12}) of main memory. Figure 8.5 shows the steps involved in address

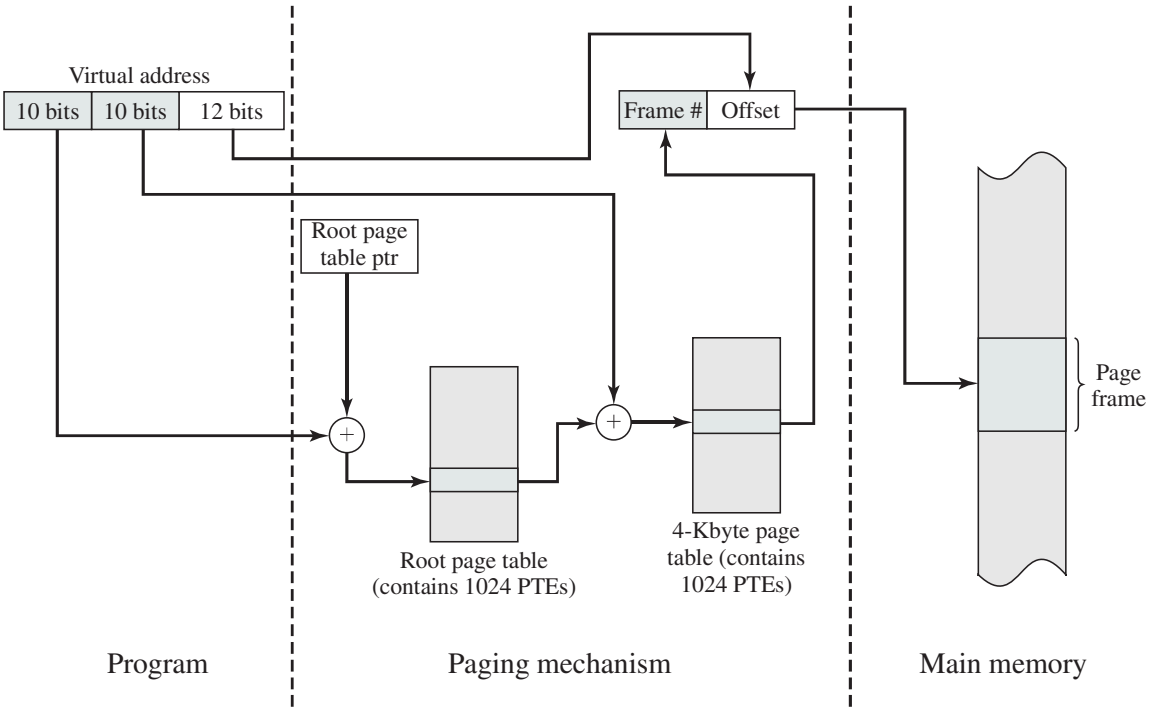


Figure 8.5 **Address Translation in a Two-Level Paging System**

ISBN 1-256-52023-3

translation for this scheme. The root page always remains in main memory. The first 10 bits of a virtual address are used to index into the root page to find a PTE for a page of the user page table. If that page is not in main memory, a page fault occurs. If that page is in main memory, then the next 10 bits of the virtual address index into the user PTE page to find the PTE for the page that is referenced by the virtual address.

Inverted Page Table A drawback of the type of page tables that we have been discussing is that their size is proportional to that of the virtual address space.

An alternative approach to the use of one or multiple-level page tables is the use of an **inverted page table** structure. Variations on this approach are used on the PowerPC, UltraSPARC, and the IA-64 architecture. An implementation of the Mach operating system on the RT-PC also uses this technique.

In this approach, the page number portion of a virtual address is mapped into a hash value using a simple hashing function.[1] The hash value is a pointer to the inverted page table, which contains the page table entries. There is one entry in the inverted page table for each real memory page frame rather than one per virtual page. Thus, a fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages supported. Because more than one virtual address may map into the same hash table entry, a chaining technique is used for managing the overflow. The hashing technique results in chains that are typically short—between one and two entries. The page table's structure is called *inverted* because it indexes page table entries by frame number rather than by virtual page number.

Figure 8.6 shows a typical implementation of the inverted page table approach. For a physical memory size of 2^m frames, the inverted page table contains 2^m entries, so that the ith entry refers to frame i. Each entry in the page table includes the following:

- **Page number:** This is the page number portion of the virtual address.
- **Process identifier:** The process that owns this page. The combination of page number and process identifier identify a page within the virtual address space of a particular process.
- **Control bits:** This field includes flags, such as valid, referenced, and modified; and protection and locking information.
- **Chain pointer:** This field is null (perhaps indicated by a separate bit) if there are no chained entries for this entry. Otherwise, the field contains the index value (number between 0 and $2^m - 1$) of the next entry in the chain.

In this example, the virtual address includes an n-bit page number, with $n > m$. The hash function maps the n-bit page number into an m-bit quantity, which is used to index into the inverted page table.

Translation Lookaside Buffer In principle, every virtual memory reference can cause two physical memory accesses: one to fetch the appropriate page table entry and one to fetch the desired data. Thus, a straightforward virtual memory

[1]See Appendix F for a discussion of hashing.

ISBN 1-256-52023-3

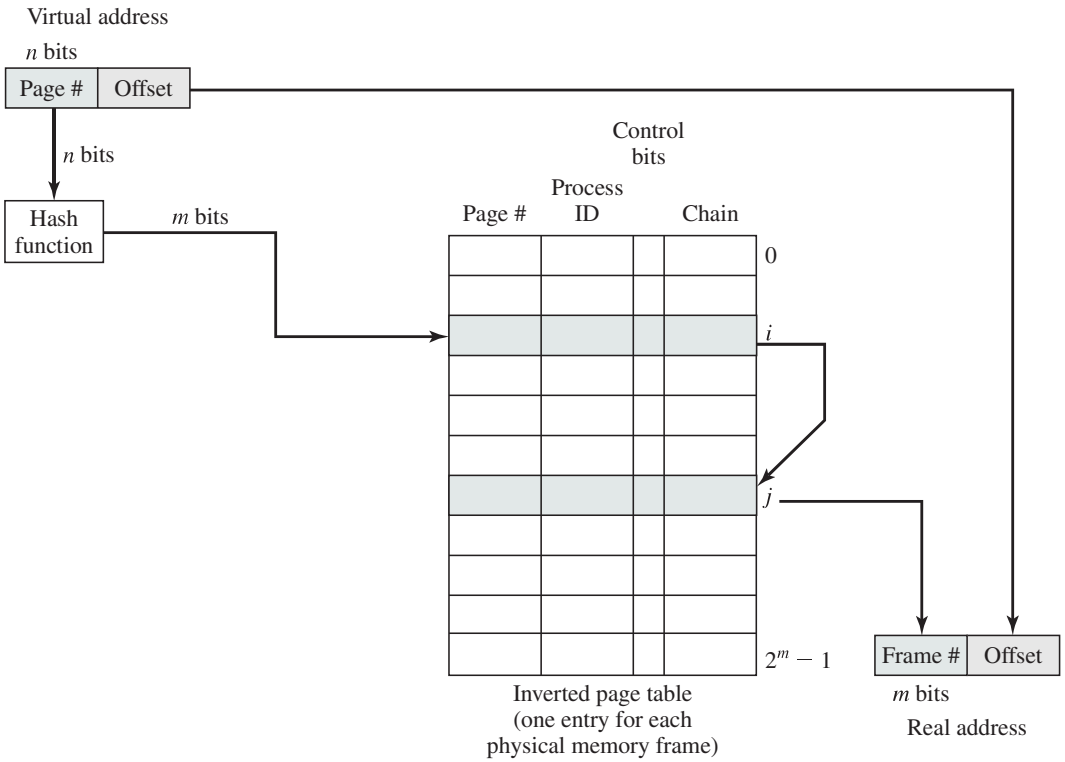


Figure 8.6 **Inverted Page Table Structure**

scheme would have the effect of doubling the memory access time. To overcome this problem, most virtual memory schemes make use of a special high-speed cache for page table entries, usually called a **translation lookaside buffer (TLB)**. This cache functions in the same way as a memory cache (see Chapter 1) and contains those page table entries that have been most recently used. The organization of the resulting paging hardware is illustrated in Figure 8.7. Given a virtual address, the processor will first examine the TLB. If the desired page table entry is present (*TLB hit*), then the frame number is retrieved and the real address is formed. If the desired page table entry is not found (*TLB miss*), then the processor uses the page number to index the process page table and examine the corresponding page table entry. If the "present bit" is set, then the page is in main memory, and the processor can retrieve the frame number from the page table entry to form the real address. The processor also updates the TLB to include this new page table entry. Finally, if the present bit is not set, then the desired page is not in main memory and a memory access fault, called a **page fault**, is issued. At this point, we leave the realm of hardware and invoke the OS, which loads the needed page and updates the page table.

Figure 8.8 is a flowchart that shows the use of the TLB. The flowchart shows that if the desired page is not in main memory, a page fault interrupt causes the page fault handling routine to be invoked. To keep the flowchart simple, the fact that the OS may dispatch another process while disk I/O is underway is not shown. By the principle of locality, most virtual memory references will be to locations in

ISBN 1-256-52023-3

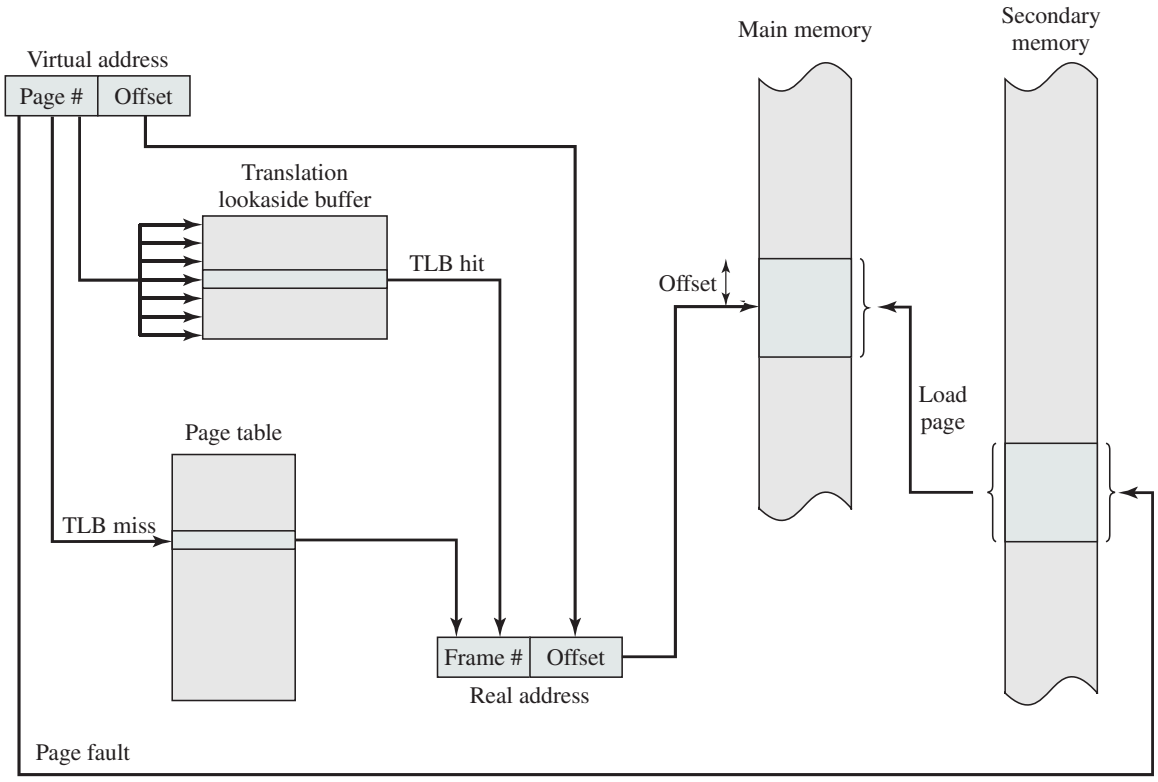


Figure 8.7 **Use of a Translation Lookaside Buffer**

recently used pages. Therefore, most references will involve page table entries in the cache. Studies of the VAX TLB have shown that this scheme can significantly improve performance [CLAR85, SATY81].

There are a number of additional details concerning the actual organization of the TLB. Because the TLB contains only some of the entries in a full page table, we cannot simply index into the TLB based on page number. Instead, each entry in the TLB must include the page number as well as the complete page table entry. The processor is equipped with hardware that allows it to interrogate simultaneously a number of TLB entries to determine if there is a match on page number. This technique is referred to as **associative mapping** and is contrasted with the direct mapping, or indexing, used for lookup in the page table in Figure 8.9. The design of the TLB also must consider the way in which entries are organized in the TLB and which entry to replace when a new entry is brought in. These issues must be considered in any hardware cache design. This topic is not pursued here; the reader may consult a treatment of cache design for further details (e.g., [STAL10]).

Finally, the virtual memory mechanism must interact with the cache system (not the TLB cache, but the main memory cache). This is illustrated in Figure 8.10. A virtual address will generally be in the form of a page number, offset. First, the memory system consults the TLB to see if the matching page table entry is present. If it is, the real (physical) address is generated by combining the frame number with the offset. If not, the entry is accessed from a page table. Once the real address is

ISBN 1-256-52023-3

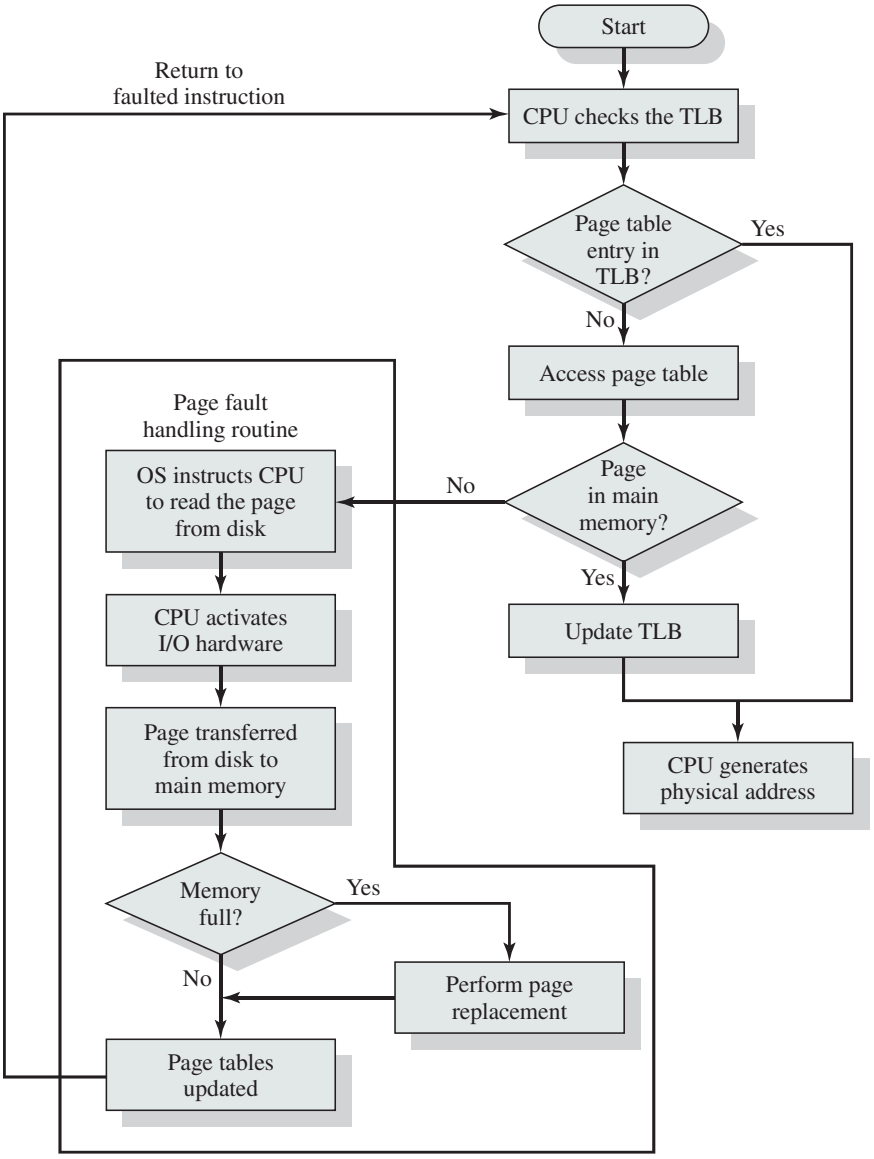


Figure 8.8 Operation of Paging and Translation Lookaside Buffer (TLB)

generated, which is in the form of a tag[2] and a remainder, the cache is consulted to see if the block containing that word is present. If so, it is returned to the CPU. If not, the word is retrieved from main memory.

The reader should be able to appreciate the complexity of the CPU hardware involved in a single memory reference. The virtual address is translated into a real address. This involves reference to a page table entry, which may be in the TLB, in main memory, or on disk. The referenced word may be in cache, main memory, or on disk. If the referenced word is only on disk, the page containing the word must

[2]See Figure 1.17. Typically, a tag is just the leftmost bits of the real address. Again, for a more detailed discussion of caches, see [STAL10].

ISBN 1-256-52023-3

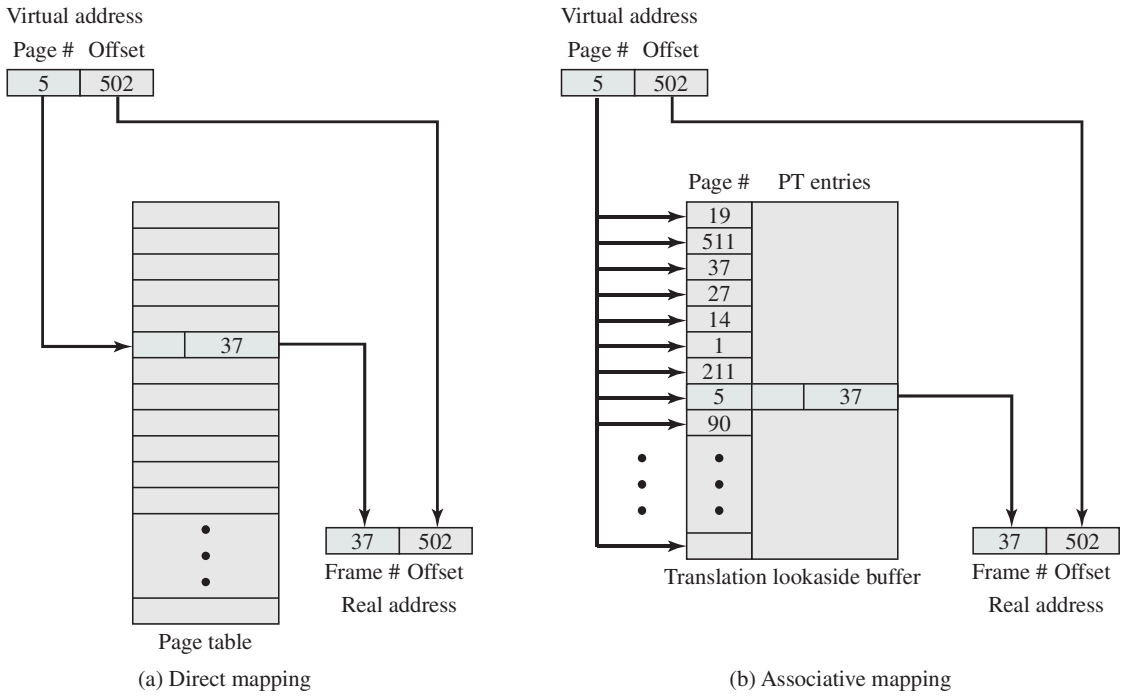


Figure 8.9 Direct versus Associative Lookup for Page Table Entries

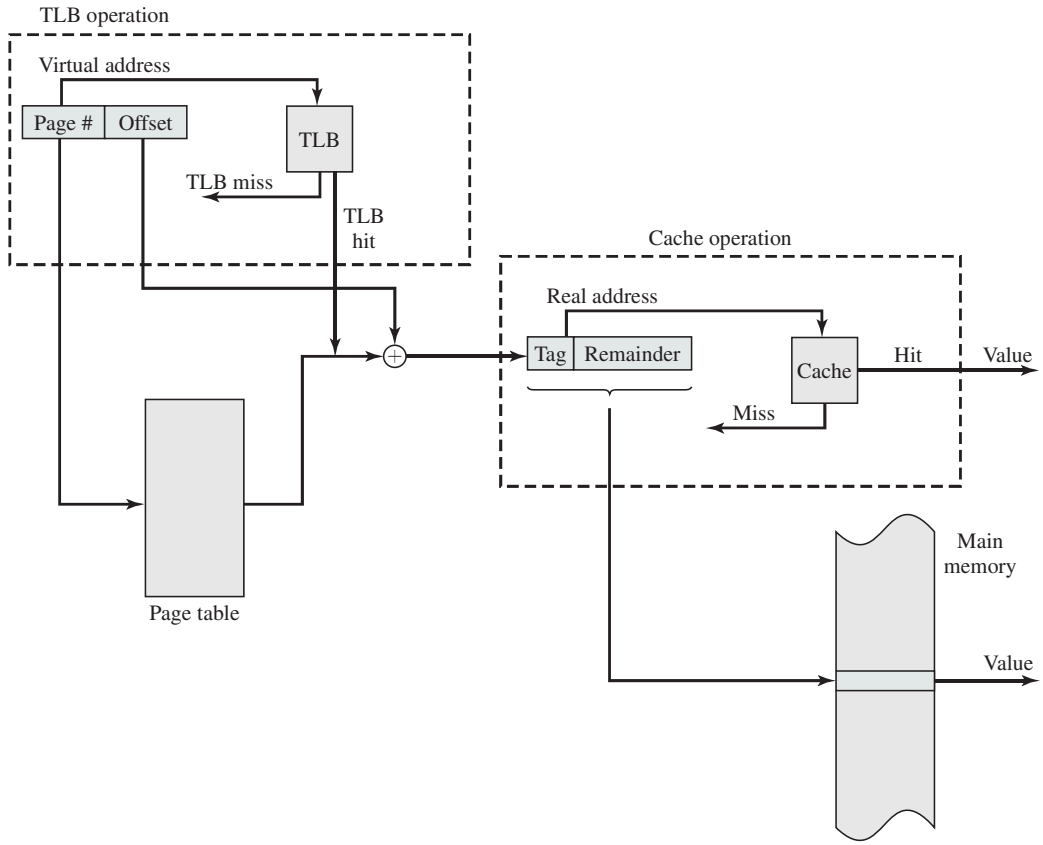


Figure 8.10 Translation Lookaside Buffer and Cache Operation

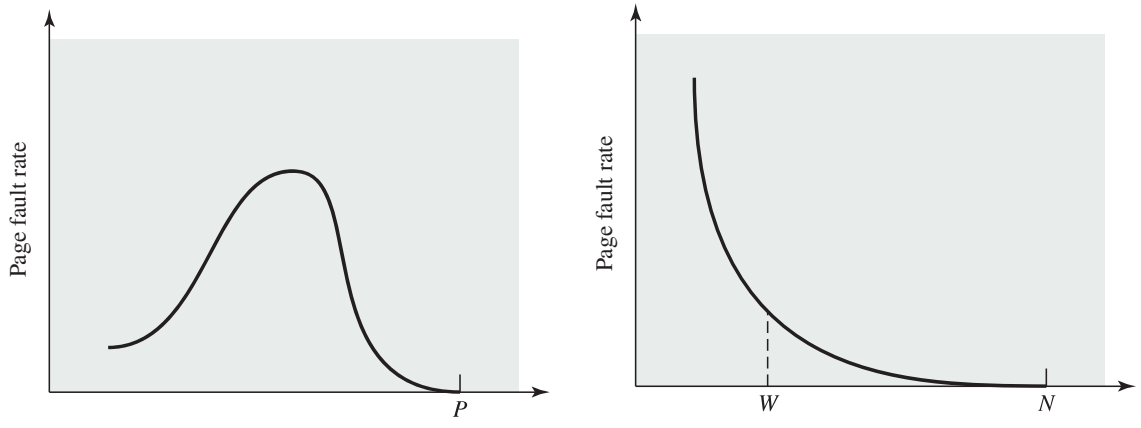
ISBN 1-256-52023-3

be loaded into main memory and its block loaded into the cache. In addition, the page table entry for that page must be updated.

PAGE SIZE An important hardware design decision is the size of page to be used. There are several factors to consider. One is internal fragmentation. Clearly, the smaller the page size, the lesser is the amount of internal fragmentation. To optimize the use of main memory, we would like to reduce internal fragmentation. On the other hand, the smaller the page, the greater is the number of pages required per process. More pages per process means larger page tables. For large programs in a heavily multiprogrammed environment, this may mean that some portion of the page tables of active processes must be in virtual memory, not in main memory. Thus, there may be a double page fault for a single reference to memory: first to bring in the needed portion of the page table and second to bring in the process page. Another factor is that the physical characteristics of most secondary-memory devices, which are rotational, favor a larger page size for more efficient block transfer of data.

Complicating these matters is the effect of page size on the rate at which page faults occur. This behavior, in general terms, is depicted in Figure 8.11a and is based on the principle of locality. If the page size is very small, then ordinarily a relatively large number of pages will be available in main memory for a process. After a time, the pages in memory will all contain portions of the process near recent references. Thus, the page fault rate should be low. As the size of the page is increased, each individual page will contain locations further and further from any particular recent reference. Thus the effect of the principle of locality is weakened and the page fault rate begins to rise. Eventually, however, the page fault rate will begin to fall as the size of a page approaches the size of the entire process (point P in the diagram). When a single page encompasses the entire process, there will be no page faults.

A further complication is that the page fault rate is also determined by the number of frames allocated to a process. Figure 8.11b shows that, for a fixed page



(a) Page size

(b) Number of page frames allocated

P = size of entire process
W = working set size
N = total number of pages in process

Figure 8.11 Typical Paging Behavior of a Program

ISBN 1-256-52023-3

Table 8.3 Example Page Sizes

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1,024 36-bit words
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
Intel Itanium	4 Kbytes to 256 Mbytes
Intel core i7	4 Kbytes to 1 Gbyte

size, the fault rate drops as the number of pages maintained in main memory grows.[3] Thus, a software policy (the amount of memory to allocate to each process) interacts with a hardware design decision (page size).

Table 8.3 lists the page sizes used on some machines.

Finally, the design issue of page size is related to the size of physical main memory and program size. At the same time that main memory is getting larger, the address space used by applications is also growing. The trend is most obvious on personal computers and workstations, where applications are becoming increasingly complex. Furthermore, contemporary programming techniques used in large programs tend to decrease the locality of references within a process [HUCK93]. For example,

- Object-oriented techniques encourage the use of many small program and data modules with references scattered over a relatively large number of objects over a relatively short period of time.
- Multithreaded applications may result in abrupt changes in the instruction stream and in scattered memory references.

For a given size of TLB, as the memory size of processes grows and as locality decreases, the hit ratio on TLB accesses declines. Under these circumstances, the TLB can become a performance bottleneck (e.g., see [CHEN92]).

One way to improve TLB performance is to use a larger TLB with more entries. However, TLB size interacts with other aspects of the hardware design, such as the main memory cache and the number of memory accesses per instruction cycle [TALL92]. The upshot is that TLB size is unlikely to grow as rapidly as main memory size. An alternative is to use larger page sizes so that each page table entry in the TLB refers to a larger block of memory. But we have just seen that the use of large page sizes can lead to performance degradation.

[3]The parameter *W* represents working set size, a concept discussed in Section 8.2.

ISBN 1-256-52023-3

Accordingly, a number of designers have investigated the use of multiple page sizes [TALL92, KHAL93], and several microprocessor architectures support multiple pages sizes, including MIPS R4000, Alpha, UltraSPARC, Pentium, and IA-64. Multiple page sizes provide the flexibility needed to use a TLB effectively. For example, large contiguous regions in the address space of a process, such as program instructions, may be mapped using a small number of large pages rather than a large number of small pages, while thread stacks may be mapped using the small page size. However, most commercial operating systems still support only one page size, regardless of the capability of the underlying hardware. The reason for this is that page size affects many aspects of the OS; thus, a change to multiple page sizes is a complex undertaking (see [GANA98] for a discussion).

Segmentation

Virtual Memory Implications Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments. Segments may be of unequal, indeed dynamic, size. Memory references consist of a (segment number, offset) form of address.

This organization has a number of advantages to the programmer over a nonsegmented address space:

1. It simplifies the handling of growing data structures. If the programmer does not know ahead of time how large a particular data structure will become, it is necessary to guess unless dynamic segment sizes are allowed. With segmented virtual memory, the data structure can be assigned its own segment, and the OS will expand or shrink the segment as needed. If a segment that needs to be expanded is in main memory and there is insufficient room, the OS may move the segment to a larger area of main memory, if available, or swap it out. In the latter case, the enlarged segment would be swapped back in at the next opportunity.
2. It allows programs to be altered and recompiled independently, without requiring the entire set of programs to be relinked and reloaded. Again, this is accomplished using multiple segments.
3. It lends itself to sharing among processes. A programmer can place a utility program or a useful table of data in a segment that can be referenced by other processes.
4. It lends itself to protection. Because a segment can be constructed to contain a well-defined set of programs or data, the programmer or system administrator can assign access privileges in a convenient fashion.

Organization In the discussion of simple segmentation, we indicated that each process has its own segment table, and when all of its segments are loaded into main memory, the segment table for a process is created and loaded into main memory. Each segment table entry contains the starting address of the corresponding segment in main memory, as well as the length of the segment. The same device, a segment table, is needed when we consider a virtual memory scheme based on segmentation. Again, it is typical to associate a unique segment table with each process. In this

ISBN 1-256-52023-3

case, however, the segment table entries become more complex (Figure 8.2b). Because only some of the segments of a process may be in main memory, a bit is needed in each segment table entry to indicate whether the corresponding segment is present in main memory or not. If the bit indicates that the segment is in memory, then the entry also includes the starting address and length of that segment.

Another control bit in the segmentation table entry is a modify bit, indicating whether the contents of the corresponding segment have been altered since the segment was last loaded into main memory. If there has been no change, then it is not necessary to write the segment out when it comes time to replace the segment in the frame that it currently occupies. Other control bits may also be present. For example, if protection or sharing is managed at the segment level, then bits for that purpose will be required.

The basic mechanism for reading a word from memory involves the translation of a virtual, or logical, address, consisting of segment number and offset, into a physical address, using a segment table. Because the segment table is of variable length, depending on the size of the process, we cannot expect to hold it in registers. Instead, it must be in main memory to be accessed. Figure 8.12 suggests a hardware implementation of this scheme (note similarity to Figure 8.3). When a particular process is running, a register holds the starting address of the segment table for that process. The segment number of a virtual address is used to index that table and look up the corresponding main memory address for the start of the segment. This is added to the offset portion of the virtual address to produce the desired real address.

Combined Paging and Segmentation

Both paging and segmentation have their strengths. Paging, which is transparent to the programmer, eliminates external fragmentation and thus provides efficient use of main memory. In addition, because the pieces that are moved in and out of

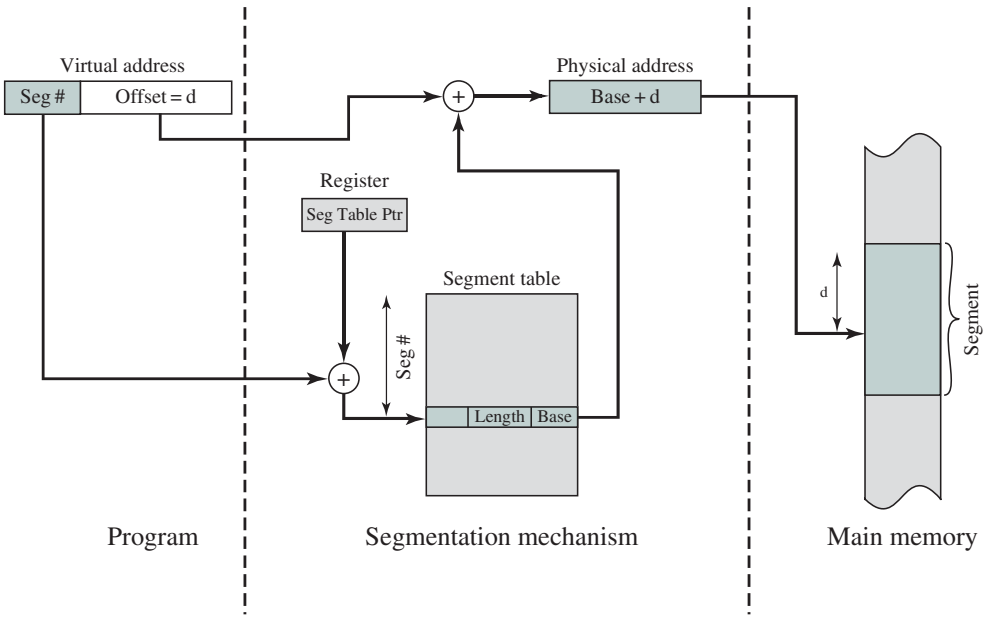


Figure 8.12 Address Translation in a Segmentation System

ISBN 1-256-52023-3

main memory are of fixed, equal size, it is possible to develop sophisticated memory management algorithms that exploit the behavior of programs, as we shall see. Segmentation, which is visible to the programmer, has the strengths listed earlier, including the ability to handle growing data structures, modularity, and support for sharing and protection. To combine the advantages of both, some systems are equipped with processor hardware and OS software to provide both.

In a combined paging/segmentation system, a user's address space is broken up into a number of segments, at the discretion of the programmer. Each segment is, in turn, broken up into a number of fixed-size pages, which are equal in length to a main memory frame. If a segment has length less than that of a page, the segment occupies just one page. From the programmer's point of view, a logical address still consists of a segment number and a segment offset. From the system's point of view, the segment offset is viewed as a page number and page offset for a page within the specified segment.

Figure 8.13 suggests a structure to support combined paging/segmentation (note similarity to Figure 8.5). Associated with each process is a segment table and a number of page tables, one per process segment. When a particular process is running, a register holds the starting address of the segment table for that process. Presented with a virtual address, the processor uses the segment number portion to index into the process segment table to find the page table for that segment. Then the page number portion of the virtual address is used to index the page table and look up the corresponding frame number. This is combined with the offset portion of the virtual address to produce the desired real address.

Figure 8.2c suggests the segment table entry and page table entry formats. As before, the segment table entry contains the length of the segment. It also contains

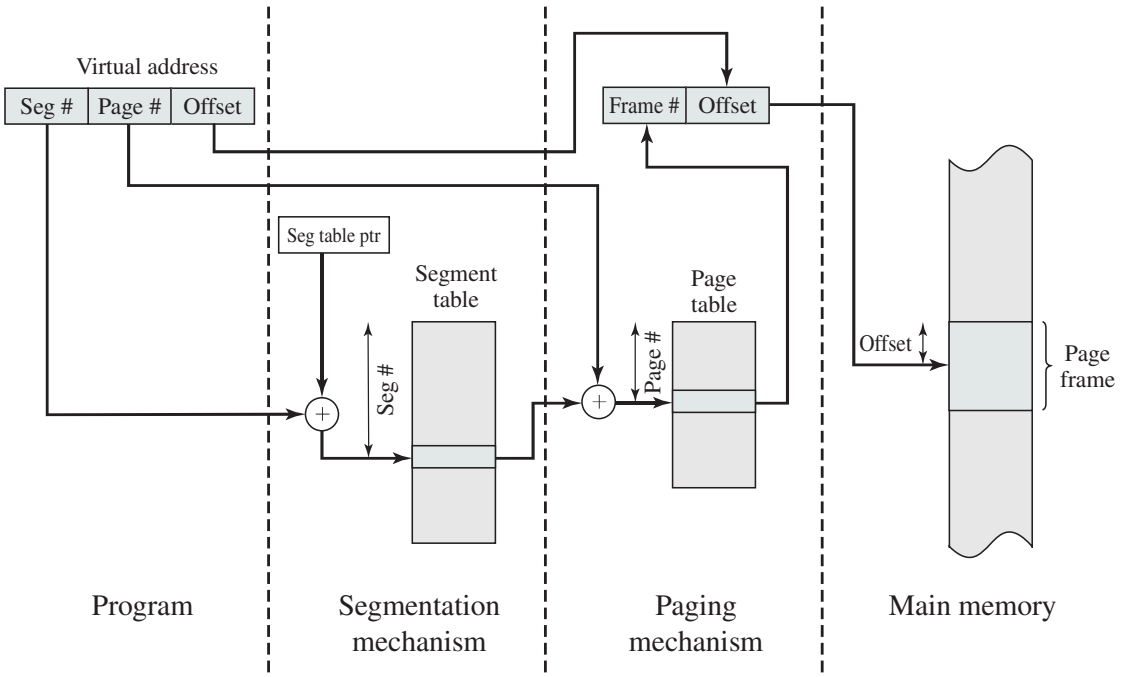


Figure 8.13 Address Translation in a Segmentation/Paging System

a base field, which now refers to a page table. The present and modified bits are not needed because these matters are handled at the page level. Other control bits may be used, for purposes of sharing and protection. The page table entry is essentially the same as is used in a pure paging system. Each page number is mapped into a corresponding frame number if the page is present in main memory. The modified bit indicates whether this page needs to be written back out when the frame is allocated to another page. There may be other control bits dealing with protection or other aspects of memory management.

Protection and Sharing

Segmentation lends itself to the implementation of protection and sharing policies. Because each segment table entry includes a length as well as a base address, a program cannot inadvertently access a main memory location beyond the limits of a segment. To achieve sharing, it is possible for a segment to be referenced in the segment tables of more than one process. The same mechanisms are, of course, available in a paging system. However, in this case the page structure of programs and data is not visible to the programmer, making the specification of protection and sharing requirements more awkward. Figure 8.14 illustrates the types of protection relationships that can be enforced in such a system.

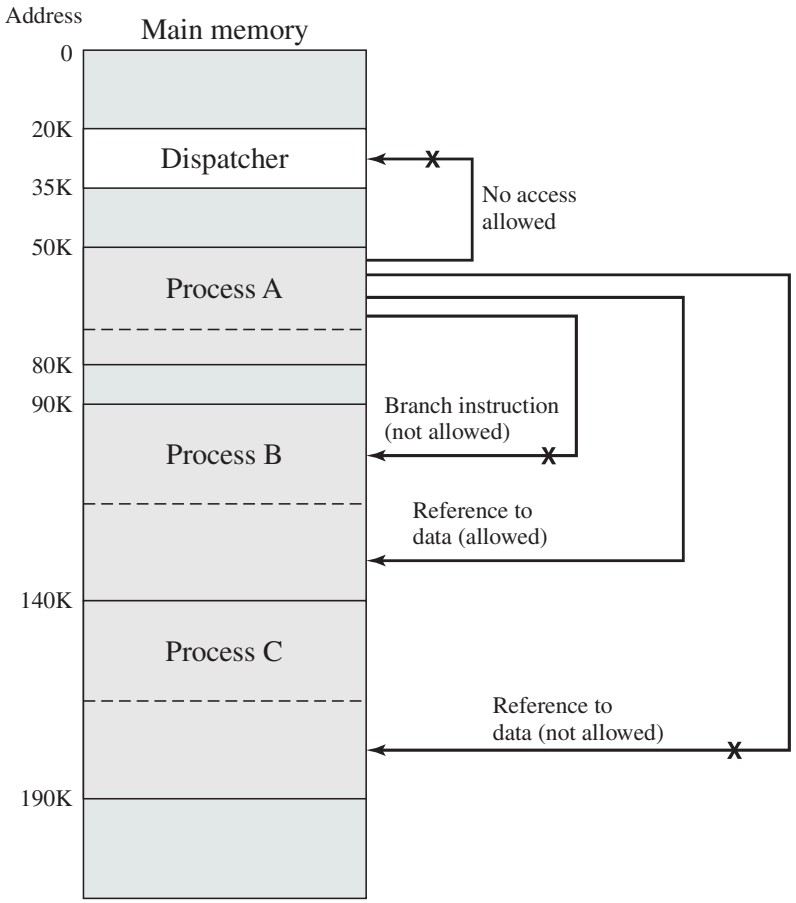


Figure 8.14 Protection Relationships between Segments

ISBN 1-256-52023-3

More sophisticated mechanisms can also be provided. A common scheme is to use a ring-protection structure, of the type we referred to in Chapter 3 (Figure 3.18). In this scheme, lower-numbered, or inner, rings enjoy greater privilege than higher-numbered, or outer, rings. Typically, ring 0 is reserved for kernel functions of the OS, with applications at a higher level. Some utilities or OS services may occupy an intermediate ring. Basic principles of the ring system are as follows:

1. A program may access only data that reside on the same ring or a less privileged ring.
2. A program may call services residing on the same or a more privileged ring.

8.2 OPERATING SYSTEM SOFTWARE

The design of the memory management portion of an OS depends on three fundamental areas of choice:

- Whether or not to use virtual memory techniques
- The use of paging or segmentation or both
- The algorithms employed for various aspects of memory management

The choices made in the first two areas depend on the hardware platform available. Thus, earlier UNIX implementations did not provide virtual memory because the processors on which the system ran did not support paging or segmentation. Neither of these techniques is practical without hardware support for address translation and other basic functions.

Two additional comments about the first two items in the preceding list: First, with the exception of operating systems for some of the older personal computers, such as MS-DOS, and specialized systems, all important operating systems provide virtual memory. Second, pure segmentation systems are becoming increasingly rare. When segmentation is combined with paging, most of the memory management issues confronting the OS designer are in the area of paging.[4] Thus, we can concentrate in this section on the issues associated with paging.

The choices related to the third item are the domain of operating system software and are the subject of this section. Table 8.4 lists the key design elements that we examine. In each case, the key issue is one of performance: We would like to minimize the rate at which page faults occur, because page faults cause considerable software overhead. At a minimum, the overhead includes deciding which resident page or pages to replace, and the I/O of exchanging pages. Also, the OS must schedule another process to run during the page I/O, causing a process switch. Accordingly, we would like to arrange matters so that, during the time that a process is executing, the probability of referencing a word on a missing page is minimized. In all of the areas referred to in Table 8.4, there is no definitive policy that works best.

[4]Protection and sharing are usually dealt with at the segment level in a combined segmentation/paging system. We will deal with these issues in later chapters.

ISBN 1-256-52023-3

Table 8.4 Operating System Policies for Virtual Memory

Fetch Policy	**Resident Set Management**
Demand paging	Resident set size
Prepaging	Fixed
	Variable
Placement Policy	Replacement Scope
	Global
Replacement Policy	Local
Basic Algorithms	
Optimal	**Cleaning Policy**
Least recently used (LRU)	Demand
First-in-first-out (FIFO)	Precleaning
Clock	
Page Buffering	**Load Control**
	Degree of multiprogramming

As we shall see, the task of memory management in a paging environment is fiendishly complex. Furthermore, the performance of any particular set of policies depends on main memory size, the relative speed of main and secondary memory, the size and number of processes competing for resources, and the execution behavior of individual programs. This latter characteristic depends on the nature of the application, the programming language and compiler employed, the style of the programmer who wrote it, and, for an interactive program, the dynamic behavior of the user. Thus, the reader must expect no final answers here or anywhere. For smaller systems, the OS designer should attempt to choose a set of policies that seems "good" over a wide range of conditions, based on the current state of knowledge. For larger systems, particularly mainframes, the operating system should be equipped with monitoring and control tools that allow the site manager to tune the operating system to get "good" results based on site conditions.

Fetch Policy

The fetch policy determines when a page should be brought into main memory. The two common alternatives are demand paging and prepaging. With **demand paging**, a page is brought into main memory only when a reference is made to a location on that page. If the other elements of memory management policy are good, the following should happen. When a process is first started, there will be a flurry of page faults. As more and more pages are brought in, the principle of locality suggests that most future references will be to pages that have recently been brought in. Thus, after a time, matters should settle down and the number of page faults should drop to a very low level.

With **prepaging,** pages other than the one demanded by a page fault are brought in. Prepaging exploits the characteristics of most secondary memory devices, such as disks, which have seek times and rotational latency. If the pages of a process are stored contiguously in secondary memory, then it is more efficient to bring in a number of contiguous pages at one time rather than bringing them in one at a time over an extended period. Of course, this policy is ineffective if most of the extra pages that are brought in are not referenced.

ISBN 1-256-52023-3

The prepaging policy could be employed either when a process first starts up, in which case the programmer would somehow have to designate desired pages, or every time a page fault occurs. This latter course would seem preferable because it is invisible to the programmer. However, the utility of prepaging has not been established [MAEK87].

Prepaging should not be confused with swapping. When a process is swapped out of memory and put in a suspended state, all of its resident pages are moved out. When the process is resumed, all of the pages that were previously in main memory are returned to main memory.

Placement Policy

The placement policy determines where in real memory a process piece is to reside. In a pure segmentation system, the placement policy is an important design issue; policies such as best-fit, first-fit, and so on, which were discussed in Chapter 7, are possible alternatives. However, for a system that uses either pure paging or paging combined with segmentation, placement is usually irrelevant because the address translation hardware and the main memory access hardware can perform their functions for any page-frame combination with equal efficiency.

There is one area in which placement does become a concern, and this is a subject of research and development. On a so-called nonuniform memory access (NUMA) multiprocessor, the distributed, shared memory of the machine can be referenced by any processor on the machine, but the time for accessing a particular physical location varies with the distance between the processor and the memory module. Thus, performance depends heavily on the extent to which data reside close to the processors that use them [LARO92, BOLO89, COX89]. For NUMA systems, an automatic placement strategy is desirable to assign pages to the memory module that provides the best performance.

Replacement Policy

In most operating system texts, the treatment of memory management includes a section entitled "replacement policy," which deals with the selection of a page in main memory to be replaced when a new page must be brought in. This topic is sometimes difficult to explain because several interrelated concepts are involved:

- How many page frames are to be allocated to each active process
- Whether the set of pages to be considered for replacement should be limited to those of the process that caused the page fault or encompass all the page frames in main memory
- Among the set of pages considered, which particular page should be selected for replacement

We shall refer to the first two concepts as *resident set management*, which is dealt with in the next subsection, and reserve the term *replacement policy* for the third concept, which is discussed in this subsection.

The area of replacement policy is probably the most studied of any area of memory management. When all of the frames in main memory are occupied and it is necessary to bring in a new page to satisfy a page fault, the replacement policy

ISBN 1-256-52023-3

determines which page currently in memory is to be replaced. All of the policies have as their objective that the page that is removed should be the page least likely to be referenced in the near future. Because of the principle of locality, there is often a high correlation between recent referencing history and near-future referencing patterns. Thus, most policies try to predict future behavior on the basis of past behavior. One trade-off that must be considered is that the more elaborate and sophisticated the replacement policy, the greater will be the hardware and software overhead to implement it.

Frame Locking One restriction on replacement policy needs to be mentioned before looking at various algorithms: Some of the frames in main memory may be locked. When a frame is locked, the page currently stored in that frame may not be replaced. Much of the kernel of the OS, as well as key control structures, are held in locked frames. In addition, I/O buffers and other time-critical areas may be locked into main memory frames. Locking is achieved by associating a lock bit with each frame. This bit may be kept in a frame table as well as being included in the current page table.

Basic Algorithms Regardless of the resident set management strategy (discussed in the next subsection), there are certain basic algorithms that are used for the selection of a page to replace. Replacement algorithms that have been discussed in the literature include

- Optimal
- Least recently used (LRU)
- First-in-first-out (FIFO)
- Clock

The **optimal** policy selects for replacement that page for which the time to the next reference is the longest. It can be shown that this policy results in the fewest number of page faults [BELA66]. Clearly, this policy is impossible to implement, because it would require the OS to have perfect knowledge of future events. However, it does serve as a standard against which to judge real-world algorithms.

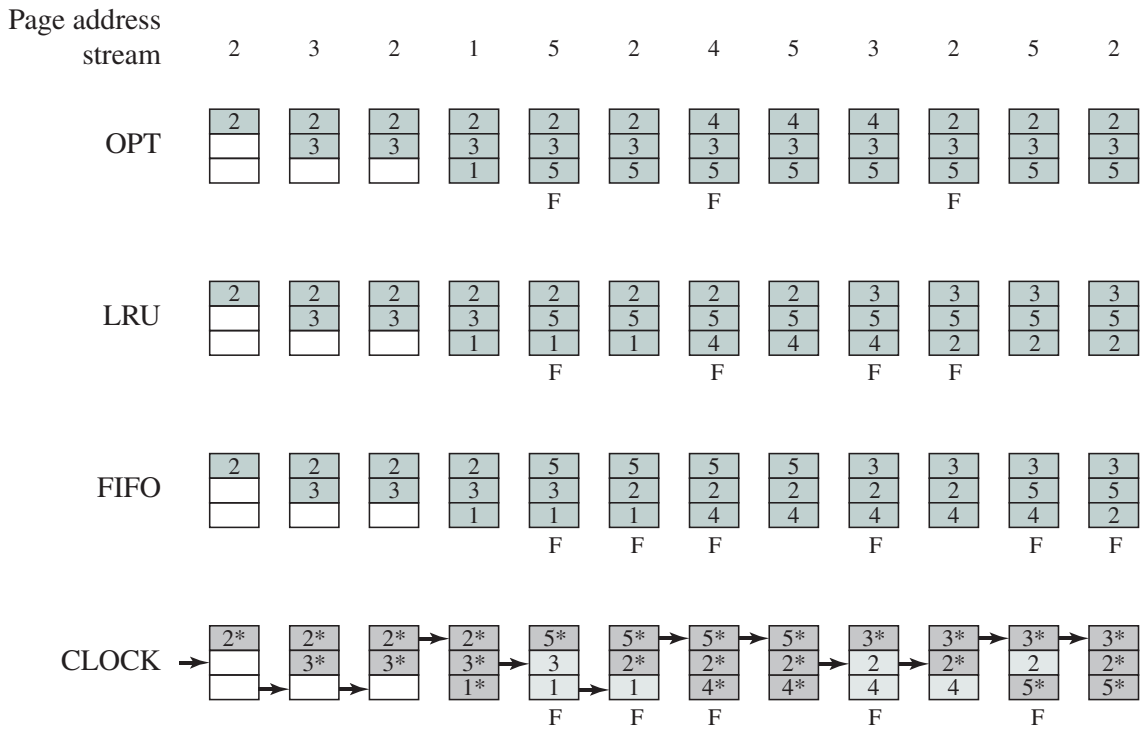
Figure 8.15 gives an example of the optimal policy. The example assumes a fixed frame allocation (fixed resident set size) for this process of three frames. The execution of the process requires reference to five distinct pages. The page address stream formed by executing the program is

2 3 2 1 5 2 4 5 3 2 5 2

which means that the first page referenced is 2, the second page referenced is 3, and so on. The optimal policy produces three page faults after the frame allocation has been filled.

The **least recently used (LRU)** policy replaces the page in memory that has not been referenced for the longest time. By the principle of locality, this should be the page least likely to be referenced in the near future. And, in fact, the LRU policy does nearly as well as the optimal policy. The problem with this approach is the difficulty in implementation. One approach would be to tag each page with the

ISBN 1-256-52023-3



F = page fault occurring after the frame allocation is initially filled

Figure 8.15 Behavior of Four Page Replacement Algorithms

time of its last reference; this would have to be done at each memory reference, both instruction and data. Even if the hardware would support such a scheme, the overhead would be tremendous. Alternatively, one could maintain a stack of page references, again an expensive prospect.

Figure 8.15 shows an example of the behavior of LRU, using the same page address stream as for the optimal policy example. In this example, there are four page faults.

The **first-in-first-out (FIFO)** policy treats the page frames allocated to a process as a circular buffer, and pages are removed in round-robin style. All that is required is a pointer that circles through the page frames of the process. This is therefore one of the simplest page replacement policies to implement. The logic behind this choice, other than its simplicity, is that one is replacing the page that has been in memory the longest: A page fetched into memory a long time ago may have now fallen out of use. This reasoning will often be wrong, because there will often be regions of program or data that are heavily used throughout the life of a program. Those pages will be repeatedly paged in and out by the FIFO algorithm.

Continuing our example in Figure 8.15, the FIFO policy results in six page faults. Note that LRU recognizes that pages 2 and 5 are referenced more frequently than other pages, whereas FIFO does not.

Although the LRU policy does nearly as well as an optimal policy, it is difficult to implement and imposes significant overhead. On the other hand, the FIFO

ISBN 1-256-52023-3

policy is very simple to implement but performs relatively poorly. Over the years, OS designers have tried a number of other algorithms to approximate the performance of LRU while imposing little overhead. Many of these algorithms are variants of a scheme referred to as the **clock policy**.

The simplest form of clock policy requires the association of an additional bit with each frame, referred to as the use bit. When a page is first loaded into a frame in memory, the use bit for that frame is set to 1. Whenever the page is subsequently referenced (after the reference that generated the page fault), its use bit is set to 1. For the page replacement algorithm, the set of frames that are candidates for replacement (this process: local scope; all of main memory: global scope[5]) is considered to be a circular buffer, with which a pointer is associated. When a page is replaced, the pointer is set to indicate the next frame in the buffer after the one just updated. When it comes time to replace a page, the OS scans the buffer to find a frame with a use bit set to 0. Each time it encounters a frame with a use bit of 1, it resets that bit to 0 and continues on. If any of the frames in the buffer have a use bit of 0 at the beginning of this process, the first such frame encountered is chosen for replacement. If all of the frames have a use bit of 1, then the pointer will make one complete cycle through the buffer, setting all the use bits to 0, and stop at its original position, replacing the page in that frame. We can see that this policy is similar to FIFO, except that, in the clock policy, any frame with a use bit of 1 is passed over by the algorithm. The policy is referred to as a clock policy because we can visualize the page frames as laid out in a circle. A number of operating systems have employed some variation of this simple clock policy (e.g., Multics [CORB68]).

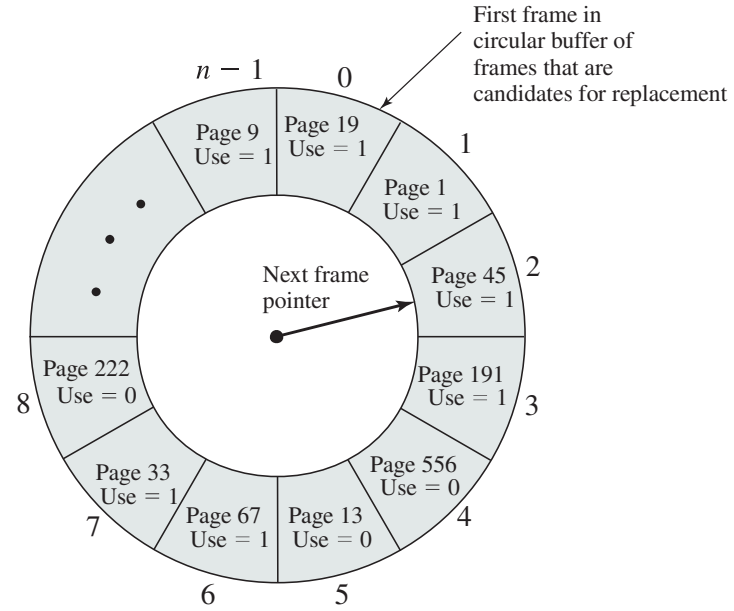
Figure 8.16 provides an example of the simple clock policy mechanism. A circular buffer of n main memory frames is available for page replacement. Just prior to the replacement of a page from the buffer with incoming page 727, the next frame pointer points at frame 2, which contains page 45. The clock policy is now executed. Because the use bit for page 45 in frame 2 is equal to 1, this page is not replaced. Instead, the use bit is set to 0 and the pointer advances. Similarly, page 191 in frame 3 is not replaced; its use bit is set to 0 and the pointer advances. In the next frame, frame 4, the use bit is set to 0. Therefore, page 556 is replaced with page 727. The use bit is set to 1 for this frame and the pointer advances to frame 5, completing the page replacement procedure.

The behavior of the clock policy is illustrated in Figure 8.15. The presence of an asterisk indicates that the corresponding use bit is equal to 1, and the arrow indicates the current position of the pointer. Note that the clock policy is adept at protecting frames 2 and 5 from replacement.

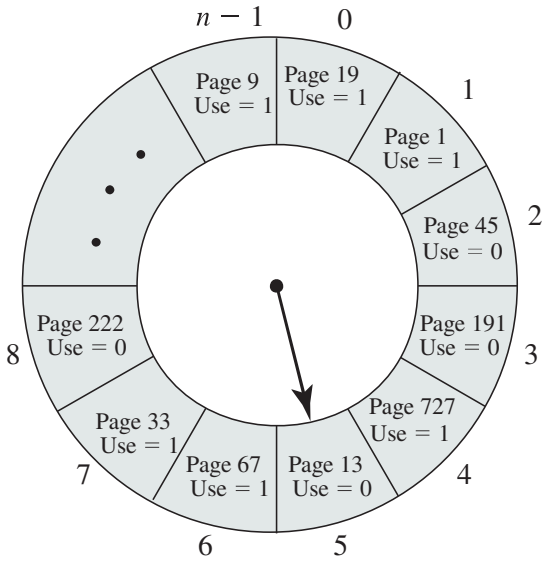
Figure 8.17 shows the results of an experiment reported in [BAER80], which compares the four algorithms that we have been discussing; it is assumed that the number of page frames assigned to a process is fixed. The results are based on the execution of 0.25×10^6 references in a FORTRAN program, using a page size of 256 words. Baer ran the experiment with frame allocations of 6, 8, 10, 12, and 14 frames. The differences among the four policies are most striking at small allocations, with

[5]The concept of scope is discussed in the subsection "Replacement Scope," subsequently.

ISBN 1-256-52023-3



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

Figure 8.16 Example of Clock Policy Operation

FIFO being over a factor of 2 worse than optimal. All four curves have the same shape as the idealized behavior shown in Figure 8.11b. In order to run efficiently, we would like to be to the right of the knee of the curve (with a small page fault rate) while keeping a small frame allocation (to the left of the knee of the curve). These two constraints indicate that a desirable mode of operation would be at the knee of the curve.

Almost identical results have been reported in [FINK88], again showing a maximum spread of about a factor of 2. Finkel's approach was to simulate the effects of various policies on a synthesized page-reference string of 10,000 references selected

ISBN 1-256-52023-3

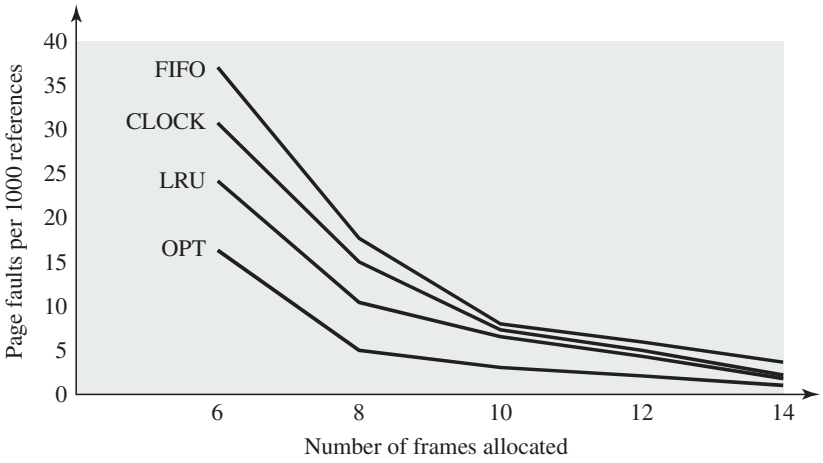


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

from a virtual space of 100 pages. To approximate the effects of the principle of locality, an exponential distribution for the probability of referencing a particular page was imposed. Finkel observes that some might be led to conclude that there is little point in elaborate page replacement algorithms when only a factor of 2 is at stake. But he notes that this difference will have a noticeable effect either on main memory requirements (to avoid degrading operating system performance) or operating system performance (to avoid enlarging main memory).

The clock algorithm has also been compared to these other algorithms when a variable allocation and either global or local replacement scope (see the following discussion of replacement policy) is used [CARR81, CARR84]. The clock algorithm was found to approximate closely the performance of LRU.

The clock algorithm can be made more powerful by increasing the number of bits that it employs.[6] In all processors that support paging, a modify bit is associated with every page in main memory and hence with every frame of main memory. This bit is needed so that, when a page has been modified, it is not replaced until it has been written back into secondary memory. We can exploit this bit in the clock algorithm in the following way. If we take the use and modify bits into account, each frame falls into one of four categories:

- Not accessed recently, not modified ($u = 0; m = 0$)
- Accessed recently, not modified ($u = 1; m = 0$)
- Not accessed recently, modified ($u = 0; m = 1$)
- Accessed recently, modified ($u = 1; m = 1$)

With this classification, the clock algorithm performs as follows:

1. Beginning at the current position of the pointer, scan the frame buffer. During this scan, make no changes to the use bit. The first frame encountered with ($u = 0; m = 0$) is selected for replacement.

[6]On the other hand, if we reduce the number of bits employed to zero, the clock algorithm degenerates to FIFO.

ISBN 1-256-52023-3

2. If step 1 fails, scan again, looking for the frame with ($u = 0$; $m = 1$). The first such frame encountered is selected for replacement. During this scan, set the use bit to 0 on each frame that is bypassed.
3. If step 2 fails, the pointer should have returned to its original position and all of the frames in the set will have a use bit of 0. Repeat step 1 and, if necessary, step 2. This time, a frame will be found for the replacement.

In summary, the page replacement algorithm cycles through all of the pages in the buffer looking for one that has not been modified since being brought in and has not been accessed recently. Such a page is a good bet for replacement and has the advantage that, because it is unmodified, it does not need to be written back out to secondary memory. If no candidate page is found in the first sweep, the algorithm cycles through the buffer again, looking for a modified page that has not been accessed recently. Even though such a page must be written out to be replaced, because of the principle of locality, it may not be needed again anytime soon. If this second pass fails, all of the frames in the buffer are marked as having not been accessed recently and a third sweep is performed.

This strategy was used on an earlier version of the Macintosh virtual memory scheme [GOLD89], illustrated in Figure 8.18. The advantage of this algorithm over

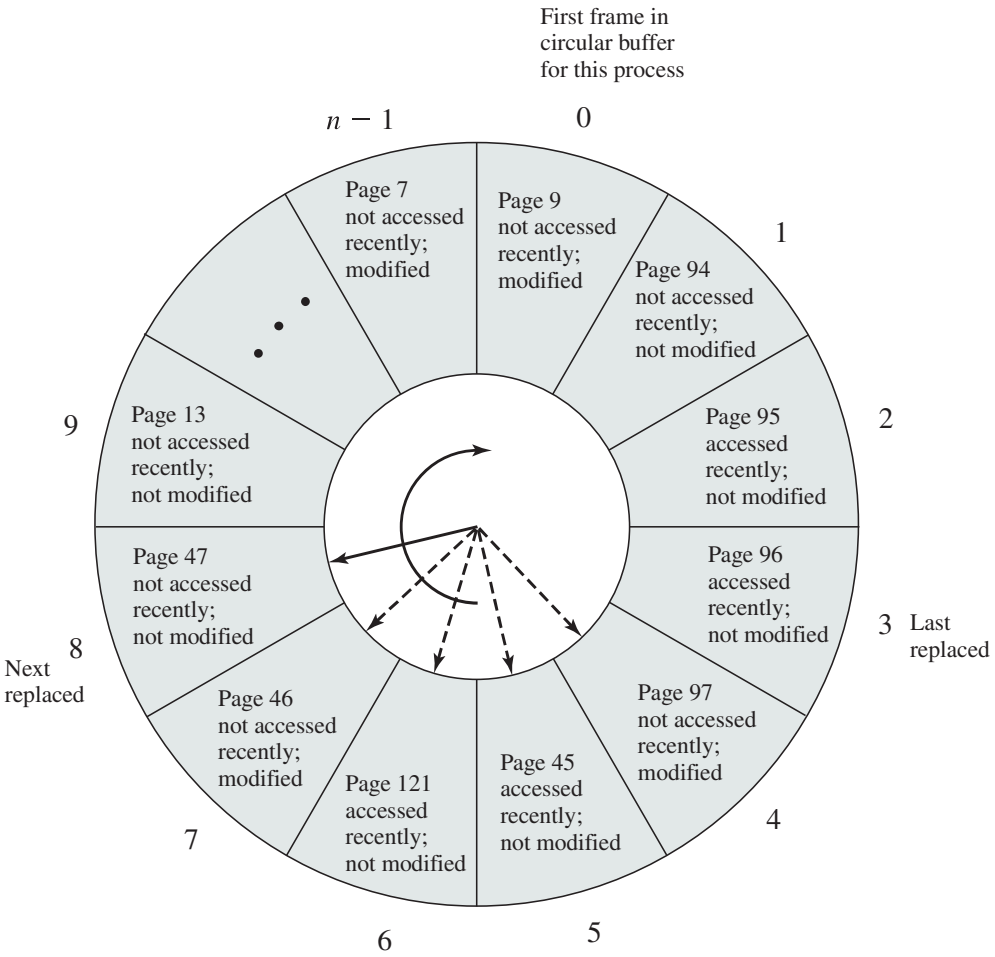


Figure 8.18 The Clock Page Replacement Algorithm [GOLD89]

ISBN 1-256-52023-3

the simple clock algorithm is that pages that are unchanged are given preference for replacement. Because a page that has been modified must be written out before being replaced, there is an immediate saving of time.

PAGE BUFFERING Although LRU and the clock policies are superior to FIFO, they both involve complexity and overhead not suffered with FIFO. In addition, there is the related issue that the cost of replacing a page that has been modified is greater than for one that has not, because the former must be written back out to secondary memory.

An interesting strategy that can improve paging performance and allow the use of a simpler page replacement policy is page buffering. The VAX VMS approach is representative. The page replacement algorithm is simple FIFO. To improve performance, a replaced page is not lost but rather is assigned to one of two lists: the free page list if the page has not been modified, or the modified page list if it has. Note that the page is not physically moved about in main memory; instead, the entry in the page table for this page is removed and placed in either the free or modified page list.

The free page list is a list of page frames available for reading in pages. VMS tries to keep some small number of frames free at all times. When a page is to be read in, the page frame at the head of the list is used, destroying the page that was there. When an unmodified page is to be replaced, it remains in memory and its page frame is added to the tail of the free page list. Similarly, when a modified page is to be written out and replaced, its page frame is added to the tail of the modified page list.

The important aspect of these maneuvers is that the page to be replaced remains in memory. Thus if the process references that page, it is returned to the resident set of that process at little cost. In effect, the free and modified page lists act as a cache of pages. The modified page list serves another useful function: Modified pages are written out in clusters rather than one at a time. This significantly reduces the number of I/O operations and therefore the amount of disk access time.

A simpler version of page buffering is implemented in the Mach operating system [RASH88]. In this case, no distinction is made between modified and unmodified pages.

REPLACEMENT POLICY AND CACHE SIZE As discussed earlier, main memory size is getting larger and the locality of applications is decreasing. In compensation, cache sizes have been increasing. Large cache sizes, even multimegabyte ones, are now feasible design alternatives [BORG90]. With a large cache, the replacement of virtual memory pages can have a performance impact. If the page frame selected for replacement is in the cache, then that cache block is lost as well as the page that it holds.

In systems that use some form of page buffering, it is possible to improve cache performance by supplementing the page replacement policy with a policy for page placement in the page buffer. Most operating systems place pages by selecting an arbitrary page frame from the page buffer; typically a first-in-first-out discipline is used. A study reported in [KESS92] shows that a careful page placement strategy can result in 10–20% fewer cache misses than naive placement.

ISBN 1-256-52023-3

Several page placement algorithms are examined in [KESS92]. The details are beyond the scope of this book, as they depend on the details of cache structure and policies. The essence of these strategies is to bring consecutive pages into main memory in such a way as to minimize the number of page frames that are mapped into the same cache slots.

Resident Set Management

Resident Set Size With paged virtual memory, it is not necessary and indeed may not be possible to bring all of the pages of a process into main memory to prepare it for execution. Thus, the OS must decide how many pages to bring in, that is, how much main memory to allocate to a particular process. Several factors come into play:

- The smaller the amount of memory allocated to a process, the more processes that can reside in main memory at any one time. This increases the probability that the OS will find at least one ready process at any given time and hence reduces the time lost due to swapping.
- If a relatively small number of pages of a process are in main memory, then, despite the principle of locality, the rate of page faults will be rather high (see Figure 8.11b).
- Beyond a certain size, additional allocation of main memory to a particular process will have no noticeable effect on the page fault rate for that process because of the principle of locality.

With these factors in mind, two sorts of policies are to be found in contemporary operating systems. A **fixed-allocation** policy gives a process a fixed number of frames in main memory within which to execute. That number is decided at initial load time (process creation time) and may be determined based on the type of process (interactive, batch, type of application) or may be based on guidance from the programmer or system manager. With a fixed-allocation policy, whenever a page fault occurs in the execution of a process, one of the pages of that process must be replaced by the needed page.

A **variable-allocation** policy allows the number of page frames allocated to a process to be varied over the lifetime of the process. Ideally, a process that is suffering persistently high levels of page faults, indicating that the principle of locality only holds in a weak form for that process, will be given additional page frames to reduce the page fault rate; whereas a process with an exceptionally low page fault rate, indicating that the process is quite well behaved from a locality point of view, will be given a reduced allocation, with the hope that this will not noticeably increase the page fault rate. The use of a variable-allocation policy relates to the concept of replacement scope, as explained in the next subsection.

The variable-allocation policy would appear to be the more powerful one. However, the difficulty with this approach is that it requires the OS to assess the behavior of active processes. This inevitably requires software overhead in the OS and is dependent on hardware mechanisms provided by the processor platform.

ISBN 1-256-52023-3

Replacement Scope The scope of a replacement strategy can be categorized as global or local. Both types of policies are activated by a page fault when there are no free page frames. A **local replacement policy** chooses only among the resident pages of the process that generated the page fault in selecting a page to replace. A **global replacement policy** considers all unlocked pages in main memory as candidates for replacement, regardless of which process owns a particular page. While it happens that local policies are easier to analyze, there is no convincing evidence that they perform better than global policies, which are attractive because of their simplicity of implementation and minimal overhead [CARR84, MAEK87].

There is a correlation between replacement scope and resident set size (Table 8.5). A fixed resident set implies a local replacement policy: To hold the size of a resident set fixed, a page that is removed from main memory must be replaced by another page from the same process. A variable-allocation policy can clearly employ a global replacement policy: The replacement of a page from one process in main memory with that of another causes the allocation of one process to grow by one page and that of the other to shrink by one page. We shall also see that variable allocation and local replacement is a valid combination. We now examine these three combinations.

Fixed Allocation, Local Scope For this case, we have a process that is running in main memory with a fixed number of frames. When a page fault occurs, the OS must choose which page from among the currently resident pages for this process is to be replaced. Replacement algorithms such as those discussed in the preceding subsection can be used.

With a fixed-allocation policy, it is necessary to decide ahead of time the amount of allocation to give to a process. This could be decided on the basis of the type of application and the amount requested by the program. The drawback to this approach is twofold: If allocations tend to be too small, then there will be a high page fault rate, causing the entire multiprogramming system to run slowly. If allocations tend to be unnecessarily large, then there will be too few programs in main memory and there will be either considerable processor idle time or considerable time spent in swapping.

Table 8.5 Resident Set Management

	Local Replacement	Global Replacement
Fixed Allocation	• Number of frames allocated to a process is fixed. • Page to be replaced is chosen from among the frames allocated to that process.	• Not possible.
Variable Allocation	• The number of frames allocated to a process may be changed from time to time to maintain the working set of the process. • Page to be replaced is chosen from among the frames allocated to that process.	• Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.

ISBN 1-256-52023-3

Variable Allocation, Global Scope This combination is perhaps the easiest to implement and has been adopted in a number of operating systems. At any given time, there are a number of processes in main memory, each with a certain number of frames allocated to it. Typically, the OS also maintains a list of free frames. When a page fault occurs, a free frame is added to the resident set of a process and the page is brought in. Thus, a process experiencing page faults will gradually grow in size, which should help reduce overall page faults in the system.

The difficulty with this approach is in the replacement choice. When there are no free frames available, the OS must choose a page currently in memory to replace. The selection is made from among all of the frames in memory, except for locked frames such as those of the kernel. Using any of the policies discussed in the preceding subsection, the page selected for replacement can belong to any of the resident processes; there is no discipline to determine which process should lose a page from its resident set. Therefore, the process that suffers the reduction in resident set size may not be optimum.

One way to counter the potential performance problems of a variable-allocation, global-scope policy is to use page buffering. In this way, the choice of which page to replace becomes less significant, because the page may be reclaimed if it is referenced before the next time that a block of pages are overwritten.

Variable Allocation, Local Scope The variable-allocation, local-scope strategy attempts to overcome the problems with a global-scope strategy. It can be summarized as follows:

1. When a new process is loaded into main memory, allocate to it a certain number of page frames as its resident set, based on application type, program request, or other criteria. Use either prepaging or demand paging to fill up the allocation.
2. When a page fault occurs, select the page to replace from among the resident set of the process that suffers the fault.
3. From time to time, reevaluate the allocation provided to the process, and increase or decrease it to improve overall performance.

With this strategy, the decision to increase or decrease a resident set size is a deliberate one and is based on an assessment of the likely future demands of active processes. Because of this evaluation, such a strategy is more complex than a simple global replacement policy. However, it may yield better performance.

The key elements of the variable-allocation, local-scope strategy are the criteria used to determine resident set size and the timing of changes. One specific strategy that has received much attention in the literature is known as the **working set strategy.** Although a true working set strategy would be difficult to implement, it is useful to examine it as a baseline for comparison.

The working set is a concept introduced and popularized by Denning [DENN68, DENN70, DENN80b]; it has had a profound impact on virtual memory management design. The working set with parameter Δ for a process at virtual time t, which we designate as $W(t, \Delta)$, is the set of pages of that process that have been referenced in the last Δ virtual time units.

ISBN 1-256-52023-3

Sequence of Page References W	Window Size, Δ 2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

Figure 8.19 **Working Set of Process as Defined by Window Size**

Virtual time is defined as follows. Consider a sequence of memory references, $r(1)$, $r(2)$, …., in which $r(i)$ is the page that contains the ith virtual address generated by a given process. Time is measured in memory references; thus $t = 1, 2, 3, \ldots$ measures the process's internal virtual time.

Let us consider each of the two variables of W. The variable Δ is a window of virtual time over which the process is observed. The working set size will be a nondecreasing function of the window size. The result is illustrated in Figure 8.19 (based on [BACH86]), which shows a sequence of page references for a process. The dots indicate time units in which the working set does not change. Note that the larger the window size, the larger is the working set. This can be expressed in the following relationship:

$$W(t,\Delta + 1) \supseteq W(t,\Delta)$$

The working set is also a function of time. If a process executes over Δ time units and uses only a single page, then $|W(t,\Delta)| = 1$. A working set can also grow as large as the number of pages N of the process if many different pages are rapidly addressed and if the window size allows. Thus,

$$1 \leq |W(t,\Delta)| \leq \min(\Delta,N)$$

Figure 8.20 indicates the way in which the working set size can vary over time for a fixed value of Δ. For many programs, periods of relatively stable working set

ISBN 1-256-52023-3

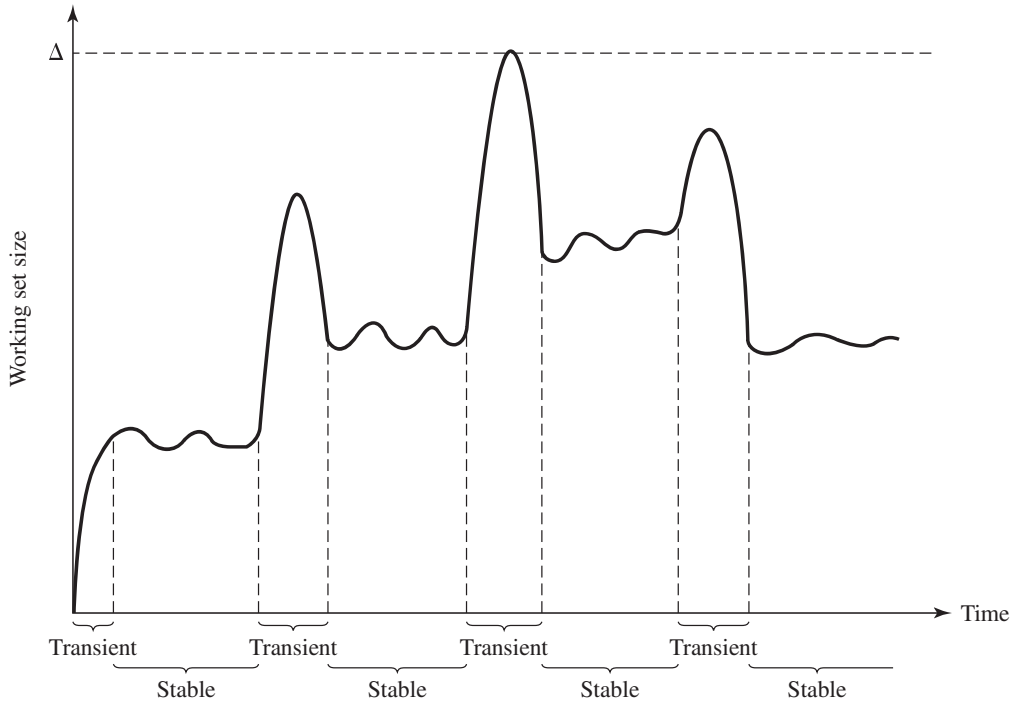


Figure 8.20 Typical Graph of Working Set Size [MAEK87]

sizes alternate with periods of rapid change. When a process first begins executing, it gradually builds up to a working set as it references new pages. Eventually, by the principle of locality, the process should stabilize on a certain set of pages. Subsequent transient periods reflect a shift of the program to a new locality. During the transition phase, some of the pages from the old locality remain within the window, Δ, causing a surge in the size of the working set as new pages are referenced. As the window slides past these page references, the working set size declines until it contains only those pages from the new locality.

This concept of a working set can be used to guide a strategy for resident set size:

1. Monitor the working set of each process.
2. Periodically remove from the resident set of a process those pages that are not in its working set. This is essentially an LRU policy.
3. A process may execute only if its working set is in main memory (i.e., if its resident set includes its working set).

This strategy is appealing because it takes an accepted principle, the principle of locality, and exploits it to achieve a memory management strategy that should minimize page faults. Unfortunately, there are a number of problems with the working set strategy:

1. The past does not always predict the future. Both the size and the membership of the working set will change over time (e.g., see Figure 8.20).
2. A true measurement of working set for each process is impractical. It would be necessary to time-stamp every page reference for every process using the

ISBN 1-256-52023-3

virtual time of that process and then maintain a time-ordered queue of pages for each process.

3. The optimal value of Δ is unknown and in any case would vary.

Nevertheless, the spirit of this strategy is valid, and a number of operating systems attempt to approximate a working set strategy. One way to do this is to focus not on the exact page references but on the page fault rate of a process. As Figure 8.11b illustrates, the page fault rate falls as we increase the resident set size of a process. The working set size should fall at a point on this curve such as indicated by W in the figure. Therefore, rather than monitor the working set size directly, we can achieve comparable results by monitoring the page fault rate. The line of reasoning is as follows: If the page fault rate for a process is below some minimum threshold, the system as a whole can benefit by assigning a smaller resident set size to this process (because more page frames are available for other processes) without harming the process (by causing it to incur increased page faults). If the page fault rate for a process is above some maximum threshold, the process can benefit from an increased resident set size (by incurring fewer faults) without degrading the system.

An algorithm that follows this strategy is the **page fault frequency (PFF)** algorithm [CHU72, GUPT78]. It requires a use bit to be associated with each page in memory. The bit is set to 1 when that page is accessed. When a page fault occurs, the OS notes the virtual time since the last page fault for that process; this could be done by maintaining a counter of page references. A threshold F is defined. If the amount of time since the last page fault is less than F, then a page is added to the resident set of the process. Otherwise, discard all pages with a use bit of 0, and shrink the resident set accordingly. At the same time, reset the use bit on the remaining pages of the process to 0. The strategy can be refined by using two thresholds: an upper threshold that is used to trigger a growth in the resident set size, and a lower threshold that is used to trigger a contraction in the resident set size.

The time between page faults is the reciprocal of the page fault rate. Although it would seem to be better to maintain a running average of the page fault rate, the use of a single time measurement is a reasonable compromise that allows decisions about resident set size to be based on the page fault rate. If such a strategy is supplemented with page buffering, the resulting performance should be quite good.

Nevertheless, there is a major flaw in the PFF approach, which is that it does not perform well during the transient periods when there is a shift to a new locality. With PFF, no page ever drops out of the resident set before F virtual time units have elapsed since it was last referenced. During interlocality transitions, the rapid succession of page faults causes the resident set of a process to swell before the pages of the old locality are expelled; the sudden peaks of memory demand may produce unnecessary process deactivations and reactivations, with the corresponding undesirable switching and swapping overheads.

An approach that attempts to deal with the phenomenon of interlocality transition with a similar relatively low overhead to that of PFF is the **variable-interval sampled working set (VSWS)** policy [FERR83]. The VSWS policy evaluates the working set of a process at sampling instances based on elapsed virtual time. At the beginning of a sampling interval, the use bits of all the resident pages for the process are reset; at the end, only the pages that have been referenced during the interval

ISBN 1-256-52023-3

will have their use bit set; these pages are retained in the resident set of the process throughout the next interval, while the others are discarded. Thus the resident set size can only decrease at the end of an interval. During each interval, any faulted pages are added to the resident set; thus the resident set remains fixed or grows during the interval.

The VSWS policy is driven by three parameters:

M: The minimum duration of the sampling interval

L: The maximum duration of the sampling interval

Q: The number of page faults that are allowed to occur between sampling instances

The VSWS policy is as follows:

1. If the virtual time since the last sampling instance reaches L, then suspend the process and scan the use bits.
2. If, prior to an elapsed virtual time of L, Q page faults occur,
   a. If the virtual time since the last sampling instance is less than M, then wait until the elapsed virtual time reaches M to suspend the process and scan the use bits.
   b. If the virtual time since the last sampling instance is greater than or equal to M, suspend the process and scan the use bits.

The parameter values are to be selected so that the sampling will normally be triggered by the occurrence of the Qth page fault after the last scan (case 2b). The other two parameters (M and L) provide boundary protection for exceptional conditions. The VSWS policy tries to reduce the peak memory demands caused by abrupt interlocality transitions by increasing the sampling frequency, and hence the rate at which unused pages drop out of the resident set, when the page fault rate increases. Experience with this technique in the Bull mainframe operating system, GCOS 8, indicates that this approach is as simple to implement as PFF and more effective [PIZZ89].

Cleaning Policy

A cleaning policy is the opposite of a fetch policy; it is concerned with determining when a modified page should be written out to secondary memory. Two common alternatives are demand cleaning and precleaning. With **demand cleaning**, a page is written out to secondary memory only when it has been selected for replacement. A **precleaning** policy writes modified pages before their page frames are needed so that pages can be written out in batches.

There is a danger in following either policy to the full. With precleaning, a page is written out but remains in main memory until the page replacement algorithm dictates that it be removed. Precleaning allows the writing of pages in batches, but it makes little sense to write out hundreds or thousands of pages only to find that the majority of them have been modified again before they are replaced. The transfer capacity of secondary memory is limited and should not be wasted with unnecessary cleaning operations.

ISBN 1-256-52023-3

On the other hand, with demand cleaning, the writing of a dirty page is coupled to, and precedes, the reading in of a new page. This technique may minimize page writes, but it means that a process that suffers a page fault may have to wait for two page transfers before it can be unblocked. This may decrease processor utilization.

A better approach incorporates page buffering. This allows the adoption of the following policy: Clean only pages that are replaceable, but decouple the cleaning and replacement operations. With page buffering, replaced pages can be placed on two lists: modified and unmodified. The pages on the modified list can periodically be written out in batches and moved to the unmodified list. A page on the unmodified list is either reclaimed if it is referenced or lost when its frame is assigned to another page.

Load Control

Load control is concerned with determining the number of processes that will be resident in main memory, which has been referred to as the multiprogramming level. The load control policy is critical in effective memory management. If too few processes are resident at any one time, then there will be many occasions when all processes are blocked, and much time will be spent in swapping. On the other hand, if too many processes are resident, then, on average, the size of the resident set of each process will be inadequate and frequent faulting will occur. The result is thrashing.

Multiprogramming Level Thrashing is illustrated in Figure 8.21. As the multiprogramming level increases from a small value, one would expect to see processor utilization rise, because there is less chance that all resident processes are blocked. However, a point is reached at which the average resident set is inadequate. At this point, the number of page faults rises dramatically, and processor utilization collapses.

There are a number of ways to approach this problem. A working set or PFF algorithm implicitly incorporates load control. Only those processes whose resident set is sufficiently large are allowed to execute. In providing the required resident set

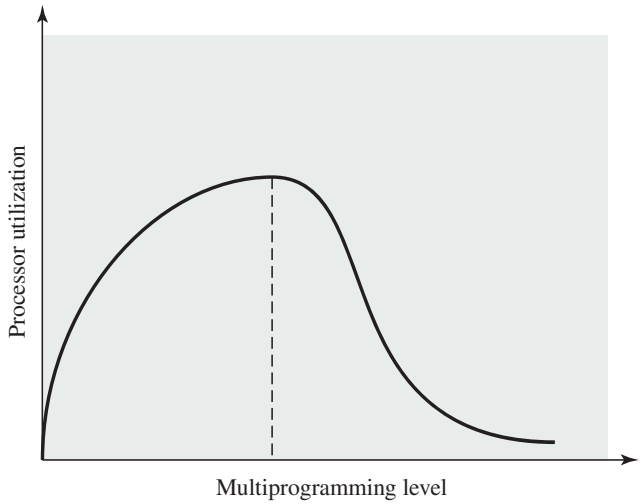


Figure 8.21 Multiprogramming Effects

ISBN 1-256-52023-3

size for each active process, the policy automatically and dynamically determines the number of active programs.

Another approach, suggested by Denning and his colleagues [DENN80b], is known as the *L = S criterion*, which adjusts the multiprogramming level so that the mean time between faults equals the mean time required to process a page fault. Performance studies indicate that this is the point at which processor utilization attained a maximum. A policy with a similar effect, proposed in [LERO76], is the *50% criterion*, which attempts to keep utilization of the paging device at approximately 50%. Again, performance studies indicate that this is a point of maximum processor utilization.

Another approach is to adapt the clock page replacement algorithm described earlier (Figure 8.16). [CARR84] describes a technique, using a global scope, that involves monitoring the rate at which the pointer scans the circular buffer of frames. If the rate is below a given lower threshold, this indicates one or both of two circumstances:

1. Few page faults are occurring, resulting in few requests to advance the pointer.
2. For each request, the average number of frames scanned by the pointer is small, indicating that there are many resident pages not being referenced and are readily replaceable.

In both cases, the multiprogramming level can safely be increased. On the other hand, if the pointer scan rate exceeds an upper threshold, this indicates either a high fault rate or difficulty in locating replaceable pages, which implies that the multiprogramming level is too high.

Process Suspension If the degree of multiprogramming is to be reduced, one or more of the currently resident processes must be suspended (swapped out). [CARR84] lists six possibilities:

- **Lowest-priority process:** This implements a scheduling policy decision and is unrelated to performance issues.
- **Faulting process:** The reasoning is that there is a greater probability that the faulting task does not have its working set resident, and performance would suffer least by suspending it. In addition, this choice has an immediate payoff because it blocks a process that is about to be blocked anyway and it eliminates the overhead of a page replacement and I/O operation.
- **Last process activated:** This is the process least likely to have its working set resident.
- **Process with the smallest resident set:** This will require the least future effort to reload. However, it penalizes programs with strong locality.
- **Largest process:** This obtains the most free frames in an overcommitted memory, making additional deactivations unlikely soon.
- **Process with the largest remaining execution window:** In most process scheduling schemes, a process may only run for a certain quantum of time before being interrupted and placed at the end of the Ready queue. This approximates a shortest-processing-time-first scheduling discipline.

ISBN 1-256-52023-3

As in so many other areas of OS design, which policy to choose is a matter of judgment and depends on many other design factors in the OS as well as the characteristics of the programs being executed.

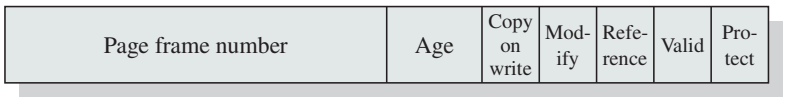
8.3 UNIX AND SOLARIS MEMORY MANAGEMENT

Because UNIX is intended to be machine independent, its memory management scheme will vary from one system to the next. Earlier versions of UNIX simply used variable partitioning with no virtual memory scheme. Current implementations of UNIX and Solaris make use of paged virtual memory.

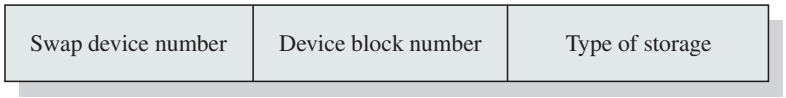
In SVR4 and Solaris, there are actually two separate memory management schemes. The **paging system** provides a virtual memory capability that allocates page frames in main memory to processes and also allocates page frames to disk block buffers. Although this is an effective memory management scheme for user processes and disk I/O, a paged virtual memory scheme is less suited to managing the memory allocation for the kernel. For this latter purpose, a **kernel memory allocator** is used. We examine these two mechanisms in turn.

Paging System

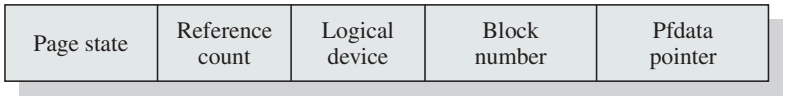
Data Structures For paged virtual memory, UNIX makes use of a number of data structures that, with minor adjustment, are machine independent (Figure 8.22 and Table 8.6):



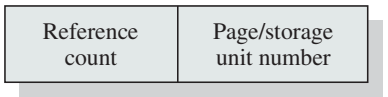
(a) Page table entry



(b) Disk block descriptor



(c) Page frame data table entry



(d) Swap-use table entry

Figure 8.22 UNIX SVR4 Memory Management Formats

ISBN 1-256-52023-3

Table 8.6 UNIX SVR4 Memory Management Parameters

Page Table Entry

Page frame number
Refers to frame in real memory.

Age
Indicates how long the page has been in memory without being referenced. The length and contents of this field are processor dependent.

Copy on write
Set when more than one process shares a page. If one of the processes writes into the page, a separate copy of the page must first be made for all other processes that share the page. This feature allows the copy operation to be deferred until necessary and avoided in cases where it turns out not to be necessary.

Modify
Indicates page has been modified.

Reference
Indicates page has been referenced. This bit is set to 0 when the page is first loaded and may be periodically reset by the page replacement algorithm.

Valid
Indicates page is in main memory.

Protect
Indicates whether write operation is allowed.

Disk Block Descriptor

Swap device number
Logical device number of the secondary device that holds the corresponding page. This allows more than one device to be used for swapping.

Device block number
Block location of page on swap device.

Type of storage
Storage may be swap unit or executable file. In the latter case, there is an indication as to whether the virtual memory to be allocated should be cleared first.

Page Frame Data Table Entry

Page state
Indicates whether this frame is available or has an associated page. In the latter case, the status of the page is specified: on swap device, in executable file, or DMA in progress.

Reference count
Number of processes that reference the page.

Logical device
Logical device that contains a copy of the page.

Block number
Block location of the page copy on the logical device.

Pfdata pointer
Pointer to other pfdata table entries on a list of free pages and on a hash queue of pages.

Swap-Use Table Entry

Reference count
Number of page table entries that point to a page on the swap device.

Page/storage unit number
Page identifier on storage unit.

ISBN 1-256-52023-3

- **Page table:** Typically, there will be one page table per process, with one entry for each page in virtual memory for that process.
- **Disk block descriptor:** Associated with each page of a process is an entry in this table that describes the disk copy of the virtual page.
- **Page frame data table:** Describes each frame of real memory and is indexed by frame number. This table is used by the replacement algorithm.
- **Swap-use table:** There is one swap-use table for each swap device, with one entry for each page on the device.

Most of the fields defined in Table 8.6 are self-explanatory. A few warrant further comment. The Age field in the page table entry is an indication of how long it has been since a program referenced this frame. However, the number of bits and the frequency of update of this field are implementation dependent. Therefore, there is no universal UNIX use of this field for page replacement policy.

The Type of Storage field in the disk block descriptor is needed for the following reason: When an executable file is first used to create a new process, only a portion of the program and data for that file may be loaded into real memory. Later, as page faults occur, new portions of the program and data are loaded. It is only at the time of first loading that virtual memory pages are created and assigned to locations on one of the devices to be used for swapping. At that time, the OS is told whether it needs to clear (set to 0) the locations in the page frame before the first loading of a block of the program or data.

PAGE REPLACEMENT The page frame data table is used for page replacement. Several pointers are used to create lists within this table. All of the available frames are linked together in a list of free frames available for bringing in pages. When the number of available frames drops below a certain threshold, the kernel will steal a number of frames to compensate.

The page replacement algorithm used in SVR4 is a refinement of the clock policy algorithm (Figure 8.16) known as the two-handed clock algorithm (Figure 8.23). The algorithm uses the reference bit in the page table entry for each page in memory that is eligible (not locked) to be swapped out. This bit is set to 0 when the page is first brought in and set to 1 when the page is referenced for a read or write. One hand in the clock algorithm, the fronthand, sweeps through the pages on the list of eligible pages and sets the reference bit to 0 on each page. Sometime later, the backhand sweeps through the same list and checks the reference bit. If the bit is set to 1, then that page has been referenced since the fronthand swept by; these frames are ignored. If the bit is still set to 0, then the page has not been referenced in the time interval between the visit by fronthand and backhand; these pages are placed on a list to be paged out.

Two parameters determine the operation of the algorithm:

- **Scanrate:** The rate at which the two hands scan through the page list, in pages per second
- **Handspread:** The gap between fronthand and backhand

These two parameters have default values set at boot time based on the amount of physical memory. The scanrate parameter can be altered to meet changing

ISBN 1-256-52023-3

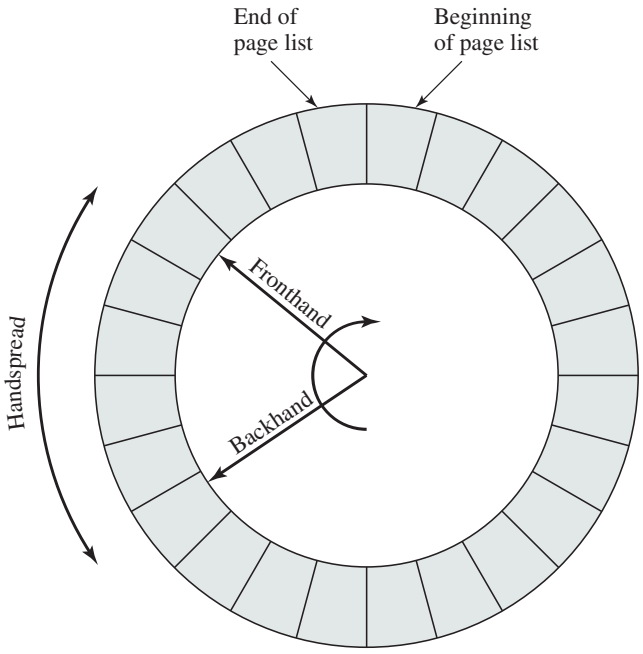


Figure 8.23 **Two-Handed Clock Page Replacement Algorithm**

conditions. The parameter varies linearly between the values slowscan and fastscan (set at configuration time) as the amount of free memory varies between the values *lotsfree* and *minfree*. In other words, as the amount of free memory shrinks, the clock hands move more rapidly to free up more pages. The handspread parameter determines the gap between the fronthand and the backhand and therefore, together with scanrate, determines the window of opportunity to use a page before it is swapped out due to lack of use.

Kernel Memory Allocator

The kernel generates and destroys small tables and buffers frequently during the course of execution, each of which requires dynamic memory allocation. [VAHA96] lists the following examples:

- The pathname translation routing may allocate a buffer to copy a pathname from user space.
- The `allocb()` routine allocates STREAMS buffers of arbitrary size.
- Many UNIX implementations allocate zombie structures to retain exit status and resource usage information about deceased processes.
- In SVR4 and Solaris, the kernel allocates many objects (such as proc structures, vnodes, and file descriptor blocks) dynamically when needed.

Most of these blocks are significantly smaller than the typical machine page size, and therefore the paging mechanism would be inefficient for dynamic kernel memory allocation. For SVR4, a modification of the buddy system, described in Section 7.2, is used.

ISBN 1-256-52023-3

In buddy systems, the cost to allocate and free a block of memory is low compared to that of best-fit or first-fit policies [KNUT97]. However, in the case of kernel memory management, the allocation and free operations must be made as fast as possible. The drawback of the buddy system is the time required to fragment and coalesce blocks.

Barkley and Lee at AT&T proposed a variation known as a lazy buddy system [BARK89], and this is the technique adopted for SVR4. The authors observed that UNIX often exhibits steady-state behavior in kernel memory demand; that is, the amount of demand for blocks of a particular size varies slowly in time. Therefore, if a block of size 2^i is released and is immediately coalesced with its buddy into a block of size 2^{i+1}, the kernel may next request a block of size 2^i, which may necessitate splitting the larger block again. To avoid this unnecessary coalescing and splitting, the lazy buddy system defers coalescing until it seems likely that it is needed, and then coalesces as many blocks as possible.

The lazy buddy system uses the following parameters:

N_i = current number of blocks of size 2^i.

A_i = current number of blocks of size 2^i that are allocated (occupied).

G_i = current number of blocks of size 2^i that are globally free; these are blocks that are eligible for coalescing; if the buddy of such a block becomes globally free, then the two blocks will be coalesced into a globally free block of size 2^{i+1}. All free blocks (holes) in the standard buddy system could be considered globally free.

L_i = current number of blocks of size 2^i that are locally free; these are blocks that are not eligible for coalescing. Even if the buddy of such a block becomes free, the two blocks are not coalesced. Rather, the locally free blocks are retained in anticipation of future demand for a block of that size.

The following relationship holds:

$$N_i = A_i + G_i + L_i$$

In general, the lazy buddy system tries to maintain a pool of locally free blocks and only invokes coalescing if the number of locally free blocks exceeds a threshold. If there are too many locally free blocks, then there is a chance that there will be a lack of free blocks at the next level to satisfy demand. Most of the time, when a block is freed, coalescing does not occur, so there is minimal bookkeeping and operational costs. When a block is to be allocated, no distinction is made between locally and globally free blocks; again, this minimizes bookkeeping.

The criterion used for coalescing is that the number of locally free blocks of a given size should not exceed the number of allocated blocks of that size (i.e., we must have $L_i \leqslant A_i$). This is a reasonable guideline for restricting the growth of locally free blocks, and experiments in [BARK89] confirm that this scheme results in noticeable savings.

To implement the scheme, the authors define a delay variable as follows:

$$D_i = A_i - L_i = N_i - 2L_i - G_i$$

Figure 8.24 shows the algorithm.

ISBN 1-256-52023-3

Initial value of D_i is 0
After an operation, the value of D_i is updated as follows

(I) if the next operation is a block allocate request:
    if there is any free block, select one to allocate
        if the selected block is locally free
            then $D_i := D_i + 2$
            else $D_i := D_i + 1$
    otherwise
        first get two blocks by splitting a larger one into two (recursive operation)
        allocate one and mark the other locally free
        D_i remains unchanged (but D may change for other block sizes because
                of the recursive call)

(II) if the next operation is a block free request
    Case $D_i \geq 2$
        mark it locally free and free it locally
        $D_i = 2$
    Case $D_i = 1$
        mark it globally free and free it globally; coalesce if possible
        $D_i = 0$
    Case $D_i = 0$
        mark it globally free and free it globally; coalesce if possible
        select one locally free block of size 2^i and free it globally; coalesce if possible
        $D_i := 0$

Figure 8.24 Lazy Buddy System Algorithm

8.4 LINUX MEMORY MANAGEMENT

Linux shares many of the characteristics of the memory management schemes of other UNIX implementations but has its own unique features. Overall, the Linux memory management scheme is quite complex [DUBE98]. In this section, we give a brief overview of the two main aspects of Linux memory management: process virtual memory and kernel memory allocation.

Linux Virtual Memory

VIRTUAL MEMORY ADDRESSING Linux makes use of a three-level page table structure, consisting of the following types of tables (each individual table is the size of one page):

- **Page directory:** An active process has a single page directory that is the size of one page. Each entry in the page directory points to one page of the page middle directory. The page directory must be in main memory for an active process.

ISBN 1-256-52023-3

- **Page middle directory:** The page middle directory may span multiple pages. Each entry in the page middle directory points to one page in the page table.
- **Page table:** The page table may also span multiple pages. Each page table entry refers to one virtual page of the process.

To use this three-level page table structure, a virtual address in Linux is viewed as consisting of four fields (Figure 8.25). The leftmost (most significant) field is used as an index into the page directory. The next field serves as an index into the page middle directory. The third field serves as an index into the page table. The fourth field gives the offset within the selected page of memory.

The Linux page table structure is platform independent and was designed to accommodate the 64-bit Alpha processor, which provides hardware support for three levels of paging. With 64-bit addresses, the use of only two levels of pages on the Alpha would result in very large page tables and directories. The 32-bit Pentium/x86 architecture has a two-level hardware paging mechanism. The Linux software accommodates the two-level scheme by defining the size of the page middle directory as one. Note that all references to an extra level of indirection are optimized away at compile time, not at run time. Therefore, there is no performance overhead for using generic three-level design on platforms which support only two levels in hardware.

PAGE ALLOCATION To enhance the efficiency of reading in and writing out pages to and from main memory, Linux defines a mechanism for dealing with contiguous blocks of pages mapped into contiguous blocks of page frames. For this purpose, the buddy system is used. The kernel maintains a list of contiguous page frame groups of fixed size; a group may consist of 1, 2, 4, 8, 16, or 32 page frames. As pages

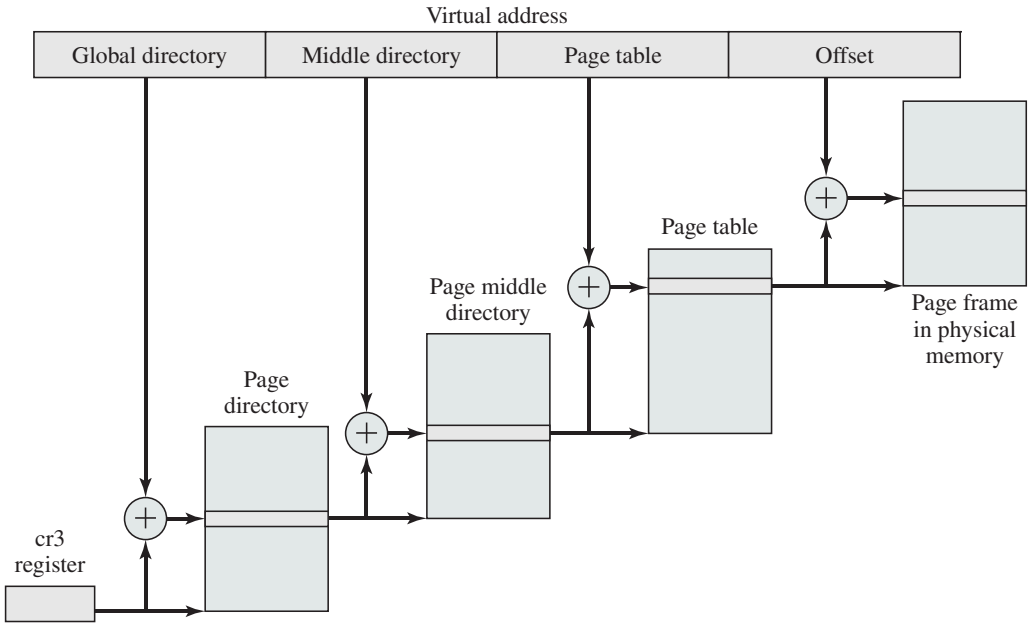


Figure 8.25 Address Translation in Linux Virtual Memory Scheme

ISBN 1-256-52023-3

are allocated and deallocated in main memory, the available groups are split and merged using the buddy algorithm.

PAGE REPLACEMENT ALGORITHM The Linux page replacement algorithm is based on the clock algorithm described in Section 8.2 (see Figure 8.16). In the simple clock algorithm, a use bit and a modify bit are associated with each page in main memory. In the Linux scheme, the use bit is replaced with an 8-bit age variable. Each time that a page is accessed, the age variable is incremented. In the background, Linux periodically sweeps through the global page pool and decrements the age variable for each page as it rotates through all the pages in main memory. A page with an age of 0 is an "old" page that has not been referenced in some time and is the best candidate for replacement. The larger the value of age, the more frequently a page has been used in recent times and the less eligible it is for replacement. Thus, the Linux algorithm is a form of least frequently used policy.

Kernel Memory Allocation

The Linux kernel memory capability manages physical main memory page frames. Its primary function is to allocate and deallocate frames for particular uses. Possible owners of a frame include user-space processes (i.e., the frame is part of the virtual memory of a process that is currently resident in real memory), dynamically allocated kernel data, static kernel code, and the page cache.[7]

The foundation of kernel memory allocation for Linux is the page allocation mechanism used for user virtual memory management. As in the virtual memory scheme, a buddy algorithm is used so that memory for the kernel can be allocated and deallocated in units of one or more pages. Because the minimum amount of memory that can be allocated in this fashion is one page, the page allocator alone would be inefficient because the kernel requires small short-term memory chunks in odd sizes. To accommodate these small chunks, Linux uses a scheme known as *slab allocation* [BONW94] within an allocated page. On a Pentium/x86 machine, the page size is 4 Kbytes, and chunks within a page may be allocated of sizes 32, 64, 128, 252, 508, 2,040, and 4,080 bytes.

The slab allocator is relatively complex and is not examined in detail here; a good description can be found in [VAHA96]. In essence, Linux maintains a set of linked lists, one for each size of chunk. Chunks may be split and aggregated in a manner similar to the buddy algorithm and moved between lists accordingly.

8.5 WINDOWS MEMORY MANAGEMENT

The Windows virtual memory manager controls how memory is allocated and how paging is performed. The memory manager is designed to operate over a variety of platforms and to use page sizes ranging from 4 Kbytes to 64 Kbytes. Intel

[7]The page cache has properties similar to a disk buffer, described in this chapter, as well as a disk cache, described in Chapter 11. We defer a discussion of the Linux page cache to Chapter 11.

ISBN 1-256-52023-3

and AMD64 platforms have 4 Kbytes per page and Intel Itanium platforms have 8 Kbytes per page.

Windows Virtual Address Map

On 32-bit platforms, each Windows user process sees a separate 32-bit address space, allowing 4 Gbytes of virtual memory per process. By default, half of this memory is reserved for the OS, so each user actually has 2 Gbytes of available virtual address space and all processes share most of the upper 2 Gbytes of system space when running in kernel-mode. Large memory intensive applications, on both clients and servers, can run more effectively using 64-bit Windows. Other than netbooks, most modern PCs use the AMD64 processor architecture which is capable of running as either a 32-bit or 64-bit system.

Figure 8.26 shows the default virtual address space seen by a normal 32-bit user process. It consists of four regions:

- **0x00000000 to 0x0000FFFF**: Set aside to help programmers catch NULL-pointer assignments.
- **0x00010000 to 0x7FFEFFFF**: Available user address space. This space is divided into pages that may be loaded into main memory.

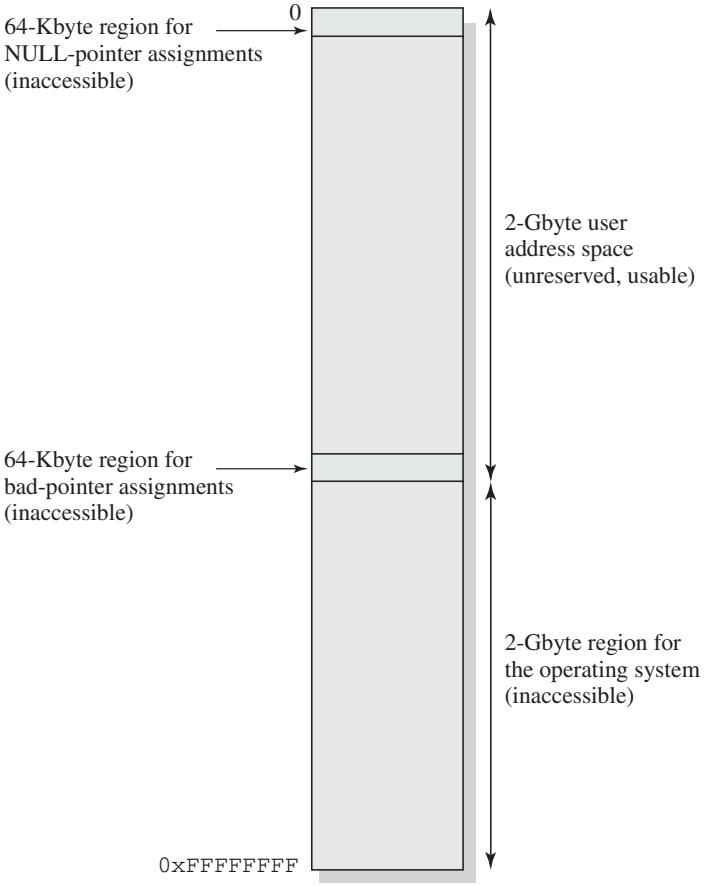


Figure 8.26 Windows Default 32-Bit Virtual Address Space

ISBN 1-256-52023-3

- **0x7FFF0000 to 0x7FFFFFFF**: A guard page inaccessible to the user. This page makes it easier for the OS to check on out-of-bounds pointer references.
- **0x80000000 to 0xFFFFFFFF**: System address space. This 2-Gbyte process is used for the Windows Executive, Kernel, HAL, and device drivers.
- On 64-bit platforms, 8 Tbytes of user address space is available in Windows 7.

Windows Paging

When a process is created, it can in principle make use of the entire user space of almost 2 Gbytes (or 8 Tbytes on 64-bit Windows). This space is divided into fixed-size pages, any of which can be brought into main memory, but the OS manages the addresses in contiguous regions allocated on 64-Kbyte boundaries. A region can be in one of three states:

- **Available:** addresses not currently used by this process.
- **Reserved:** addresses that the virtual memory manager has set aside for a process so they cannot be allocated to another use (e.g., saving contiguous space for a stack to grow).
- **Committed:** addresses that the virtual memory manager has initialized for use by the process to access virtual memory pages. These pages can reside either on disk or in physical memory. When on disk they can be either kept in files (mapped pages) or occupy space in the paging file (i.e., the disk file to which it writes pages when removing them from main memory).

The distinction between reserved and committed memory is useful because it (1) reduces the amount of total virtual memory space needed by the system, allowing the page file to be smaller; and (2) allows programs to reserve addresses without making them accessible to the program or having them charged against their resource quotas.

The resident set management scheme used by Windows is variable allocation, local scope (see Table 8.5). When a process is first activated, it is assigned data structures to manage its working set. As the pages needed by the process are brought into physical memory the memory manager uses the data structures to keep track of the pages assigned to the process. Working sets of active processes are adjusted using the following general conventions:

- When main memory is plentiful, the virtual memory manager allows the resident sets of active processes to grow. To do this, when a page fault occurs, a new physical page is added to the process but no older page is swapped out, resulting in an increase of the resident set of that process by one page.
- When memory becomes scarce, the virtual memory manager recovers memory for the system by removing less recently used pages out of the working sets of active processes, reducing the size of those resident sets.
- Even when memory is plentiful, Windows watches for large processes that are rapidly increasing their memory usage. The system begins to remove

ISBN 1-256-52023-3

pages that have not been recently used from the process. This policy makes the system more responsive because a new program will not suddenly cause a scarcity of memory and make the user wait while the system tries to reduce the resident sets of the processes that are already running.

8.6 SUMMARY

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 way to address both of these concerns is virtual memory. With virtual memory, all address references are logical references that are translated at run time to real addresses. This allows a process to be located anywhere in main memory and for that location to change over time. Virtual memory also allows a process to be broken up into pieces. These pieces need not be contiguously located in main memory during execution and, indeed, it is not even necessary for all of the pieces of the process to be in main memory during execution.

Two basic approaches to providing virtual memory 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.

A virtual memory management scheme requires both hardware and software support. The hardware support is provided by the processor. The support includes dynamic translation of virtual addresses to physical addresses and the generation of an interrupt when a referenced page or segment is not in main memory. Such an interrupt triggers the memory management software in the OS.

A number of design issues relate to OS support for memory management:

- **Fetch policy:** Process pages can be brought in on demand, or a prepaging policy can be used, which clusters the input activity by bringing in a number of pages at once.
- **Placement policy:** With a pure segmentation system, an incoming segment must be fit into an available space in memory.
- **Replacement policy:** When memory is full, a decision must be made as to which page or pages are to be replaced.
- **Resident set management:** The OS must decide how much main memory to allocate to a particular process when that process is swapped in. This can be a static allocation made at process creation time, or it can change dynamically.
- **Cleaning policy:** Modified process pages can be written out at the time of replacement, or a precleaning policy can be used, which clusters the output activity by writing out a number of pages at once.
- **Load control:** Load control is concerned with determining the number of processes that will be resident in main memory at any given time.

ISBN 1-256-52023-3

8.7 RECOMMENDED READING AND WEB SITES

As might be expected, virtual memory receives good coverage in most books on operating systems. [MILE92] provides a good summary of various research areas. [CARR84] provides an excellent in-depth examination of performance issues. The classic paper [DENN70] is still well worth a read. [DOWD93] provides an instructive performance analysis of various page replacement algorithms. [JACO98a] is a good survey of issues in virtual memory design; it includes a discussion of inverted page tables. [JACO98b] looks at virtual memory hardware organizations in various microprocessors.

It is a sobering experience to read [IBM86], which gives a detailed account of the tools and options available to a site manager in optimizing the virtual memory policies of MVS. The document illustrates the complexity of the problem.

[VAHA96] is one of the best treatments of the memory management schemes used in the various flavors of UNIX. [GORM04] is a thorough treatment of Linux memory management.

CARR84 Carr, R. *Virtual Memory Management.* Ann Arbor, MI: UMI Research Press, 1984.

DENN70 Denning, P. "Virtual Memory." *Computing Surveys*, September 1970.

DOWD93 Dowdy, L., and Lowery, C. *P.S. to Operating Systems.* Upper Saddle River, NJ: Prentice Hall, 1993.

GORM04 Gorman, M. *Understanding the Linux Virtual Memory Manager.* Upper Saddle River, NJ: Prentice Hall, 2004.

IBM86 IBM National Technical Support, Large Systems. *Multiple Virtual Storage (MVS) Virtual Storage Tuning Cookbook.* Dallas Systems Center Technical Bulletin G320-0597, June 1986.

JACO98a Jacob, B., and Mudge, T. "Virtual Memory: Issues of Implementation." *Computer*, June 1998.

JACO98b Jacob, B., and Mudge, T. "Virtual Memory in Contemporary Microprocessors." *IEEE Micro*, August 1998.

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

VAHA96 Vahalia, U. *UNIX Internals: The New Frontiers.* Upper Saddle River, NJ: Prentice Hall, 1996.

Recommended Web site:

- **The Memory Management Reference:** A good source of documents and links on all aspects of memory management.

ISBN 1-256-52023-3

8.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

associative mapping	page	resident set management
demand paging	page fault	segment
external fragmentation	page placement policy	segment table
fetch policy	page replacement policy	segmentation
frame	page table	slab allocation
hash table	paging	thrashing
hashing	prepaging	translation lookaside buffer
internal fragmentation	real memory	virtual memory
locality	resident set	working set

Review Questions

8.1 What is the difference between simple paging and virtual memory paging?
8.2 Explain thrashing.
8.3 Why is the principle of locality crucial to the use of virtual memory?
8.4 What elements are typically found in a page table entry? Briefly define each element.
8.5 What is the purpose of a translation lookaside buffer?
8.6 Briefly define the alternative page fetch policies.
8.7 What is the difference between resident set management and page replacement policy?
8.8 What is the relationship between FIFO and clock page replacement algorithms?
8.9 What is accomplished by page buffering?
8.10 Why is it not possible to combine a global replacement policy and a fixed allocation policy?
8.11 What is the difference between a resident set and a working set?
8.12 What is the difference between demand cleaning and precleaning?

Problems

8.1 Suppose the page table for the process currently executing on the processor looks like the following. All numbers are decimal, everything is numbered starting from zero, and all addresses are memory byte addresses. The page size is 1,024 bytes.

Virtual page number	Valid bit	Reference bit	Modify bit	Page frame number
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

ISBN 1-256-52023-3

a. Describe exactly how, in general, a virtual address generated by the CPU is translated into a physical main memory address.
b. What physical address, if any, would each of the following virtual addresses correspond to? (Do not try to handle any page faults, if any.)
(i) 1,052
(ii) 2,221
(iii) 5,499

8.2 Consider the following program.

```
#define Size 64
int A[Size; Size], B[Size; Size], C[Size; Size];
int register i, j;

for (j = 0; j< Size; j ++)
   for (i = 0; i< Size; i++)
C[i; j] = A[i; j] + B[i; j];
```

Assume that the program is running on a system using demand paging and the page size is 1 Kilobyte. Each integer is 4 bytes long. It is clear that each array requires a 16-page space. As an example, A[0, 0]-A[0, 63], A[1, 0]-A[1, 63], A[2, 0]-A[2, 63], and A[3, 0]-A[3, 63] will be stored in the first data page. A similar storage pattern can be derived for the rest of array A and for arrays B and C. Assume that the system allocates a 4-page working set for this process. One of the pages will be used by the program and three pages can be used for the data. Also, two index registers are assigned for i and j (so, no memory accesses are needed for references to these two variables).

a. Discuss how frequently the page fault would occur (in terms of number of times C[i, j] = A[i, j] + B[i, j] are executed).
b. Can you modify the program to minimize the page fault frequency?
c. What will be the frequency of page faults after your modification?

8.3 **a.** How much memory space is needed for the user page table of Figure 8.4?
b. Assume you want to implement a hashed inverted page table for the same addressing scheme as depicted in Figure 8.4, using a hash function that maps the 20-bit page number into a 6-bit hash value. The table entry contains the page number, the frame number, and a chain pointer. If the page table allocates space for up to 3 overflow entries per hashed entry, how much memory space does the hashed inverted page table take?

8.4 Consider the following string of page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2. Complete a figure similar to Figure 8.15, showing the frame allocation for:

a. FIFO (first-in-first-out)
b. LRU (least recently used)
c. Clock
d. Optimal (assume the page reference string continues with 1, 2, 0, 1, 7, 0, 1)
e. List the total number of page faults and the miss rate for each policy. Count page faults only after all frames have been initialized.

8.5 A process references five pages, A, B, C, D, and E, in the following order:

A; B; C; D; A; B; E; A; B; C; D; E

Assume that the replacement algorithm is first-in-first-out and find the number of page transfers during this sequence of references starting with an empty main memory with three page frames. Repeat for four page frames.

8.6 A process contains eight virtual pages on disk and is assigned a fixed allocation of four page frames in main memory. The following page trace occurs:

1, 0, 2, 2, 1, 7, 6, 7, 0, 1, 2, 0, 3, 0, 4, 5, 1, 5, 2, 4, 5, 6, 7, 6, 7, 2, 4, 2, 7, 3, 3, 2, 3

ISBN 1-256-52023-3

a. Show the successive pages residing in the four frames using the LRU replacement policy. Compute the hit ratio in main memory. Assume that the frames are initially empty.
b. Repeat part (a) for the FIFO replacement policy.
c. Compare the two hit ratios and comment on the effectiveness of using FIFO to approximate LRU with respect to this particular trace.

8.7 In the VAX, user page tables are located at virtual addresses in the system space. What is the advantage of having user page tables in virtual rather than main memory? What is the disadvantage?

8.8 Suppose the program statement

```
for (i = 1; i <= n; i ++)
        a[i] = b[i] + c[i];
```

is executed in a memory with page size of 1,000 words. Let $n = 1{,}000$. Using a machine that has a full range of register-to-register instructions and employs index registers, write a hypothetical program to implement the foregoing statement. Then show the sequence of page references during execution.

8.9 The IBM System/370 architecture uses a two-level memory structure and refers to the two levels as segments and pages, although the segmentation approach lacks many of the features described earlier in this chapter. For the basic 370 architecture, the page size may be either 2 Kbytes or 4 Kbytes, and the segment size is fixed at either 64 Kbytes or 1 Mbyte. For the 370/XA and 370/ESA architectures, the page size is 4 Kbytes and the segment size is 1 Mbyte. Which advantages of segmentation does this scheme lack? What is the benefit of segmentation for the 370?

8.10 Assuming a page size of 4 Kbytes and that a page table entry takes 4 bytes, how many levels of page tables would be required to map a 64-bit address space, if the top level page table fits into a single page?

8.11 Consider a system with memory mapping done on a page basis and using a single-level page table. Assume that the necessary page table is always in memory.

a. If a memory reference takes 200 ns, how long does a paged memory reference take?
b. Now we add an MMU that imposes an overhead of 20 ns on a hit or a miss. If we assume that 85% of all memory references hit in the MMU TLB, what is the Effective Memory Access Time (EMAT)?
c. Explain how the TLB hit rate affects the EMAT.

8.12 Consider a page reference string for a process with a working set of M frames, initially all empty. The page reference string is of length P with N distinct page numbers in it. For any page replacement algorithm,

a. What is a lower bound on the number of page faults?
b. What is an upper bound on the number of page faults?

8.13 In discussing a page replacement algorithm, one author makes an analogy with a snowplow moving around a circular track. Snow is falling uniformly on the track and a lone snowplow continually circles the track at constant speed. The snow that is plowed w the track disappears from the system.

a. For which of the page replacement algorithms discussed in Section 8.2 is this a useful analogy?
b. What does this analogy suggest about the behavior of the page replacement algorithm in question?

8.14 In the S/370 architecture, a storage key is a control field associated with each page-sized frame of real memory. Two bits of that key that are relevant for page replacement are the reference bit and the change bit. The reference bit is set to 1 when any address within the frame is accessed for read or write, and is set to 0 when a new page is loaded into the frame. The change bit is set to 1 when a write operation is performed on any location within the frame. Suggest an approach for determining which page frames are least-recently-used, making use of only the reference bit.

ISBN 1-256-52023-3

8.15 Consider the following sequence of page references (each element in the sequence represents a page number):

1 2 3 4 5 2 1 3 3 2 3 4 5 4 5 1 1 3 2 5

Define the *mean working set size* after the kth reference as $s_k(\Delta) = \frac{1}{k}\sum_{t=1}^{k}|W(t,\Delta)|$ and define the *missing page probability* after the kth reference as $m_k(\Delta) = \frac{1}{k}\sum_{t=1}^{k}F(t,\Delta)$ where $F(t, \Delta) = 1$ if a page fault occurs at virtual time t and 0 otherwise.

a. Draw a diagram similar to that of Figure 8.19 for the reference sequence just defined for the values $\Delta = 1, 2, 3, 4, 5, 6$.
b. Plot $s_{20}(\Delta)$ as a function of Δ.
c. Plot $m_{20}(\Delta)$ as a function of Δ.

8.16 A key to the performance of the VSWS resident set management policy is the value of Q. Experience has shown that, with a fixed value of Q for a process, there are considerable differences in page fault frequencies at various stages of execution. Furthermore, if a single value of Q is used for different processes, dramatically different frequencies of page faults occur. These differences strongly indicate that a mechanism that would dynamically adjust the value of Q during the lifetime of a process would improve the behavior of the algorithm. Suggest a simple mechanism for this purpose.

8.17 Assume that a task is divided into four equal-sized segments and that the system builds an eight-entry page descriptor table for each segment. Thus, the system has a combination of segmentation and paging. Assume also that the page size is 2 Kbytes.

a. What is the maximum size of each segment?
b. What is the maximum logical address space for the task?
c. Assume that an element in physical location 00021ABC is accessed by this task. What is the format of the logical address that the task generates for it? What is the maximum physical address space for the system?

8.18 Consider a paged logical address space (composed of 32 pages of 2 Kbytes each) mapped into a 1-Mbyte physical memory space.

a. What is the format of the processor's logical address?
b. What is the length and width of the page table (disregarding the "access rights" bits)?
c. What is the effect on the page table if the physical memory space is reduced by half?

8.19 The UNIX kernel will dynamically grow a process's stack in virtual memory as needed, but it will never try to shrink it. Consider the case in which a program calls a C subroutine that allocates a local array on the stack that consumes 10 K. The kernel will expand the stack segment to accommodate it. When the subroutine returns, the stack pointer is adjusted and this space could be released by the kernel, but it is not released. Explain why it would be possible to shrink the stack at this point and why the UNIX kernel does not shrink it.

ISBN 1-256-52023-3