CHAPTER 8

VIRTUAL MEMORY

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.

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 will focus first on the hardware aspect of virtual memory, looking at the use of paging, segmentation, and combined paging and segmentation. Then we will 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.

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:

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.

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.

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 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 (input/output) 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 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.

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.

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 not partitioned.	
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).	
Internal fragmentation within frames.		No internal fragmentation.	
No 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 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 list of free holes in main memory.	
Processor uses page 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.

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 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.

The principle of locality suggests that a virtual memory scheme may be effective. 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 will examine the hardware aspect and look at the 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 will be 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 will be 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. Recall from Chapter 7 that with 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. With virtual memory paging, we again have equal-size

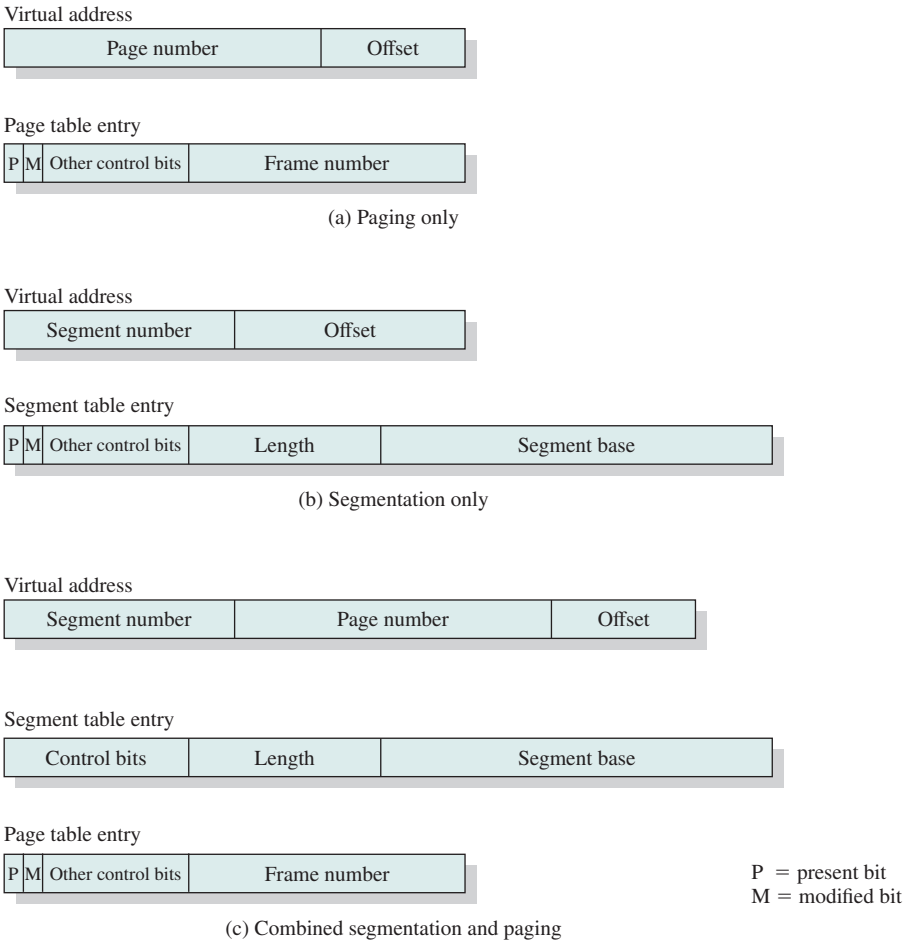


Figure 8.1 Typical Memory Management Formats

pages of the same length as frames; however, not all pages need to be loaded into main memory frames for execution.

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 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 (see Figure 8.1a). 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

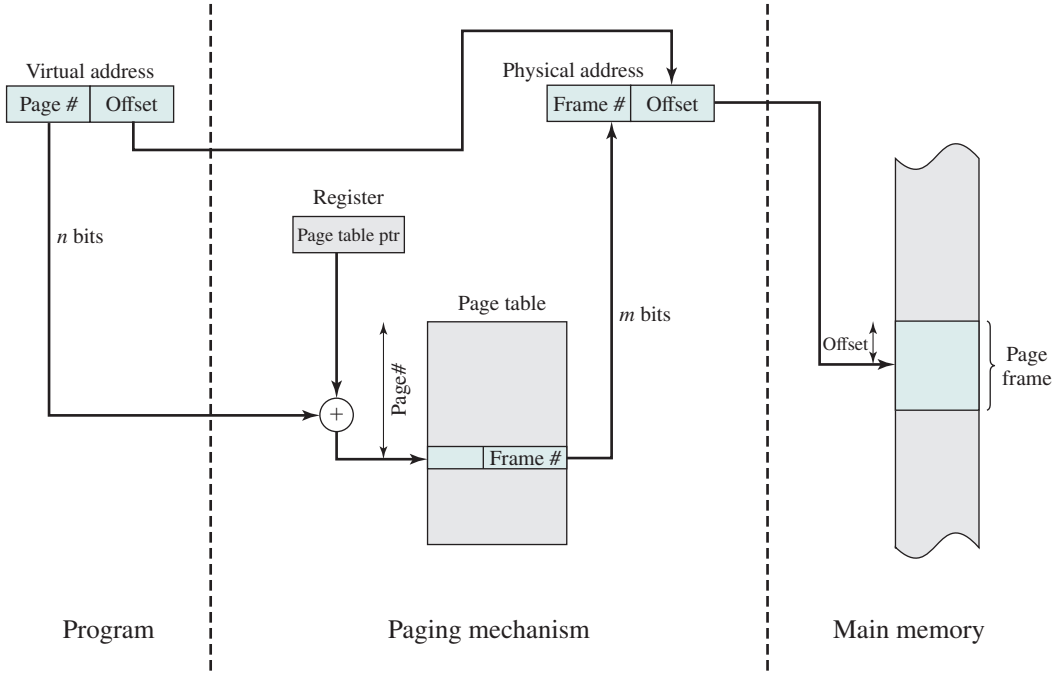


Figure 8.2 Address Translation in a Paging System

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.

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 8.2 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 ($n > m$). This inequality results from the fact that the number of pages in a process may exceed the number of frames in main memory.

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 (Virtual Address Extension) architecture, each process can have up to $2^{31} = 2$ GB 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

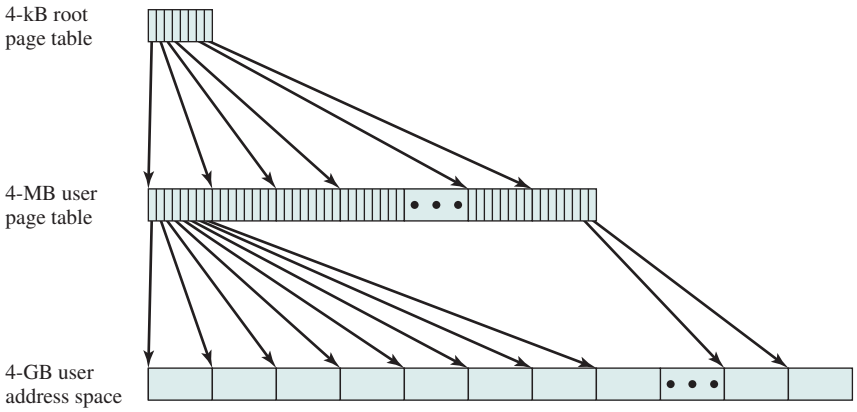


Figure 8.3 **A Two-Level Hierarchical Page Table**

page tables in virtual memory rather than real memory. This means 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 number of entries in the page directory is X, and if the maximum number of entries in a page table is Y, then a process can 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 8.3 shows an example of a two-level scheme typical for use with a 32-bit address. If we assume byte-level addressing and 4-kB (2^{12}) pages, then the 4-GB (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 MB (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 kB (2^{12}) of main memory. Figure 8.4 shows the steps involved in address 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

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

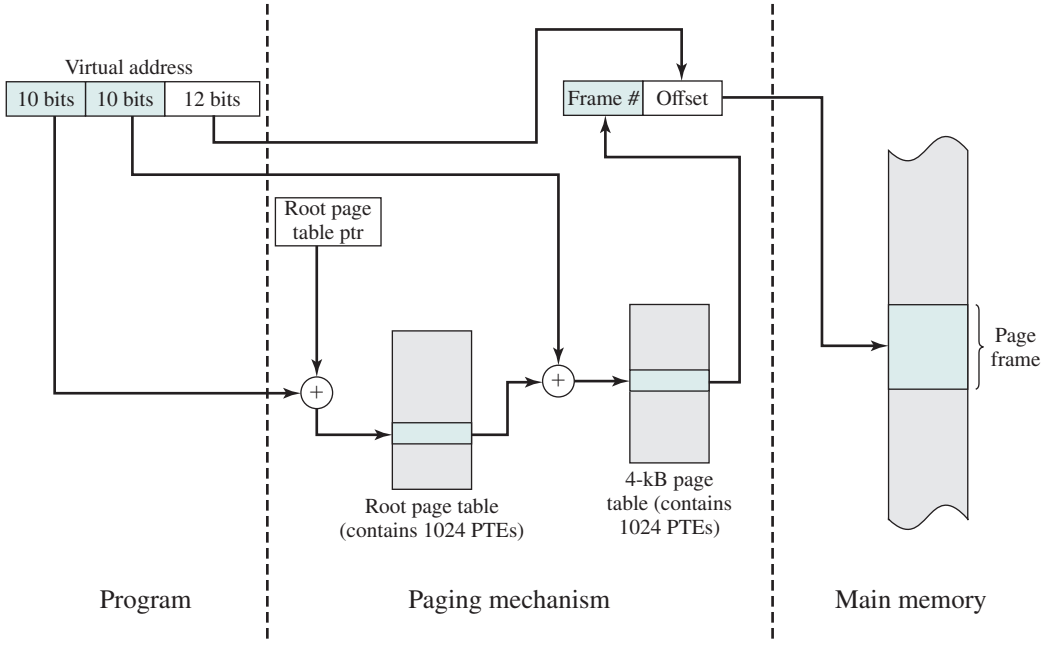


Figure 8.4 Address Translation in a Two-Level Paging System

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 8.5 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 identifies 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.

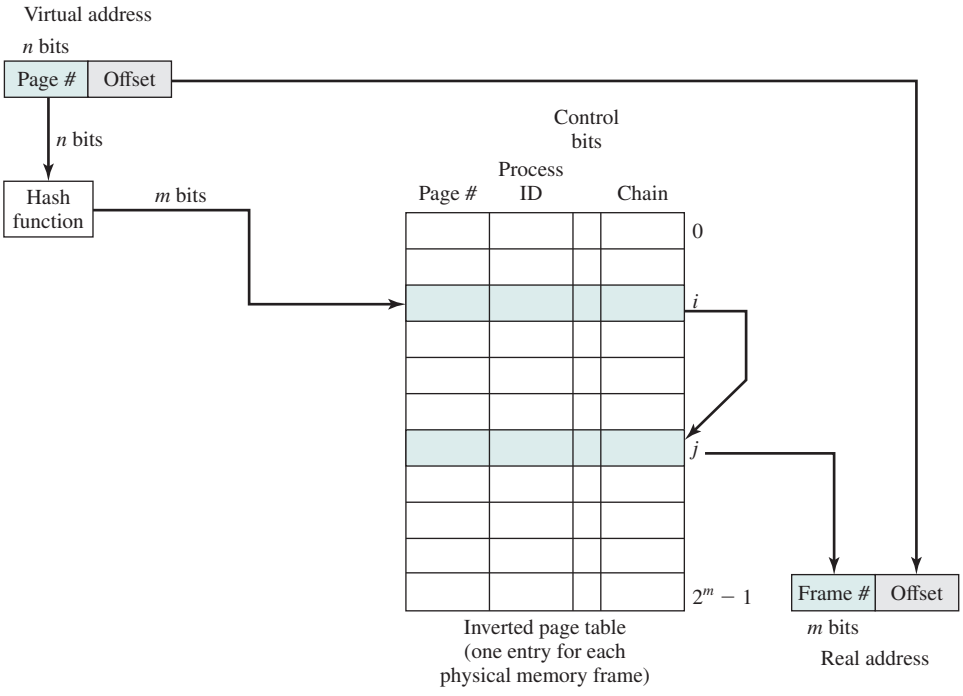


Figure 8.5 **Inverted Page Table Structure**

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 another to fetch the desired data. Thus, a straightforward virtual memory 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 8.6. 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

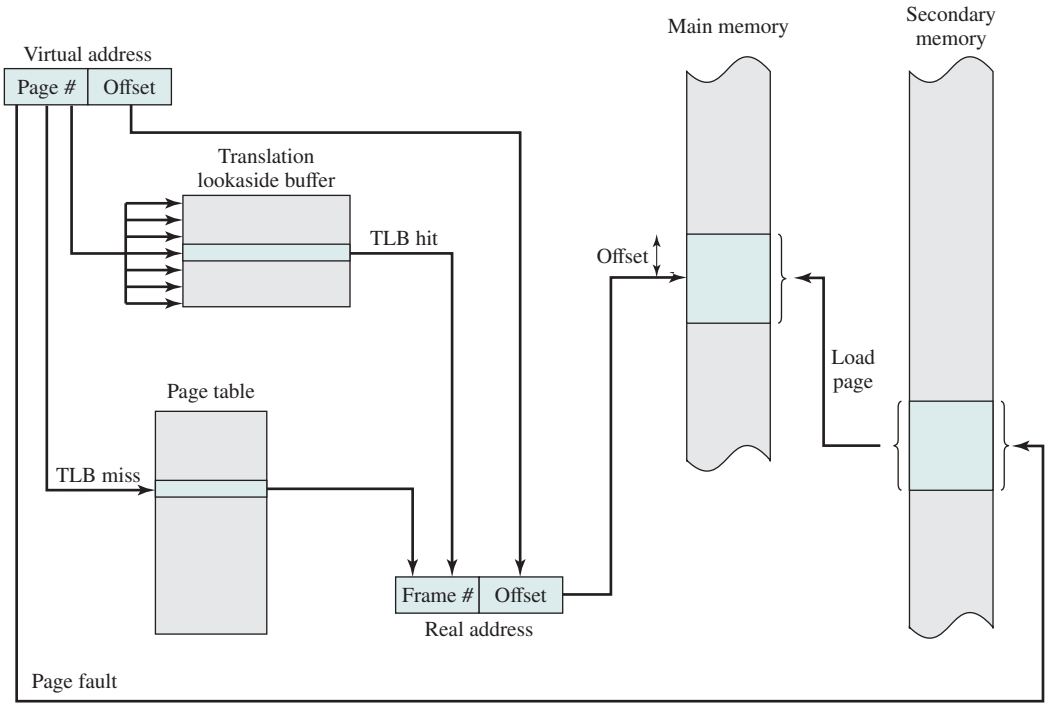


Figure 8.6 **Use of a Translation Lookaside Buffer**

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 8.7 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 recently used pages. Therefore, most references will involve page table entries in the cache. Studies of the VAX TLB have shown 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 8.8. 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., [STAL16a]).

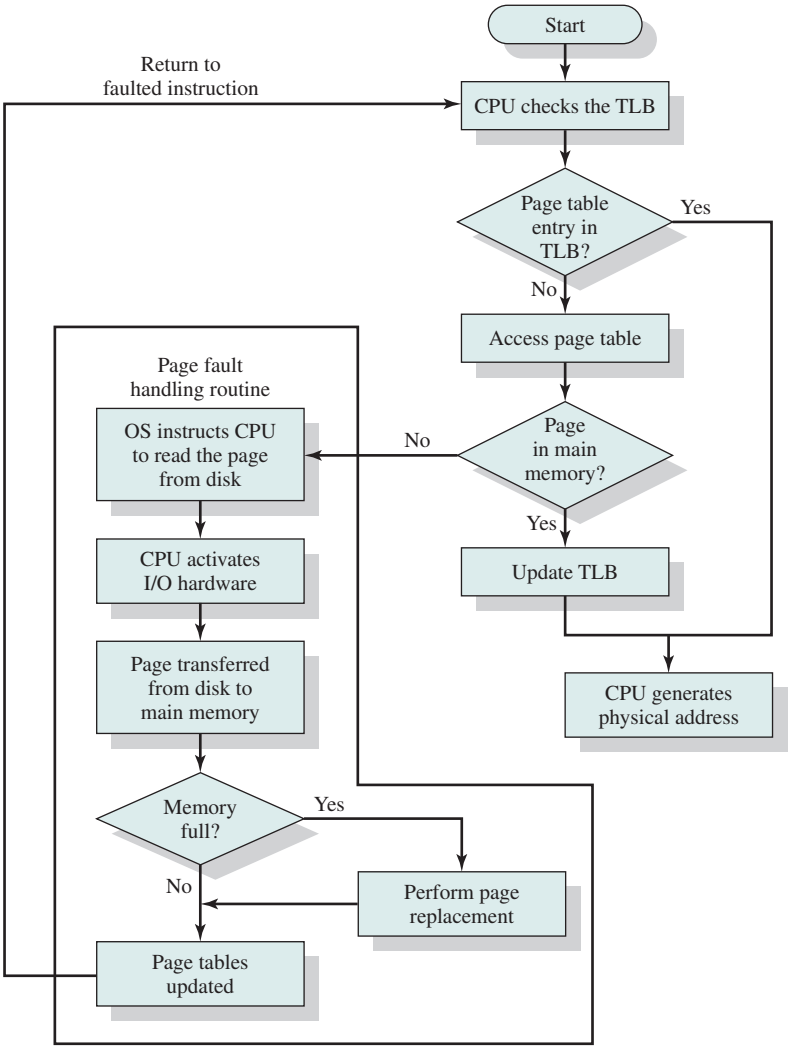


Figure 8.7 Operation of Paging and Translation Lookaside Buffer (TLB)

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 8.9. 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 generated, which is in the form of a tag[2] and a remainder, the cache is consulted to see if the

[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 [STAL16a].

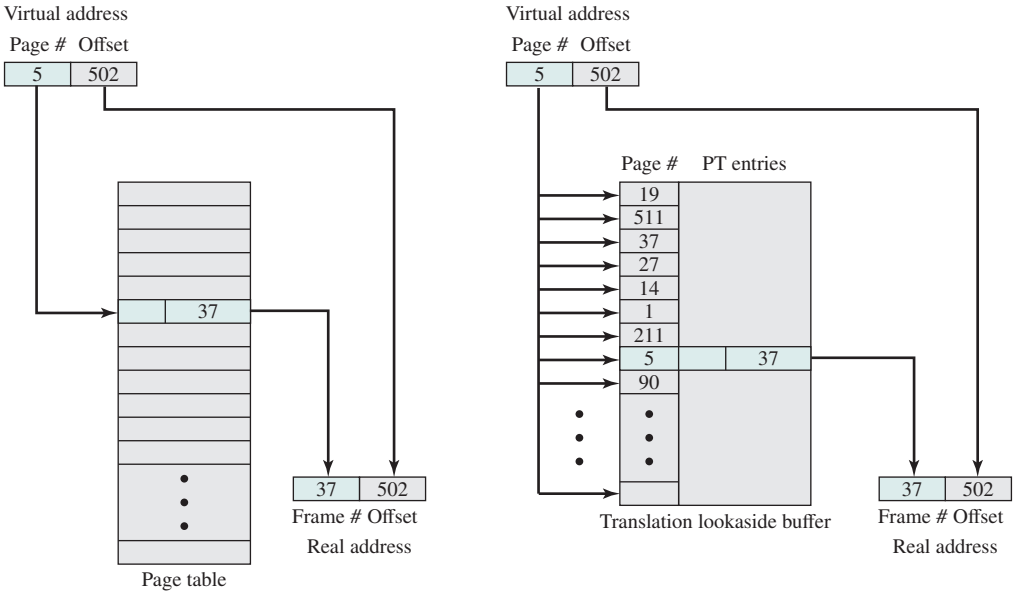


Figure 8.8 Direct versus Associative Lookup for Page Table Entries

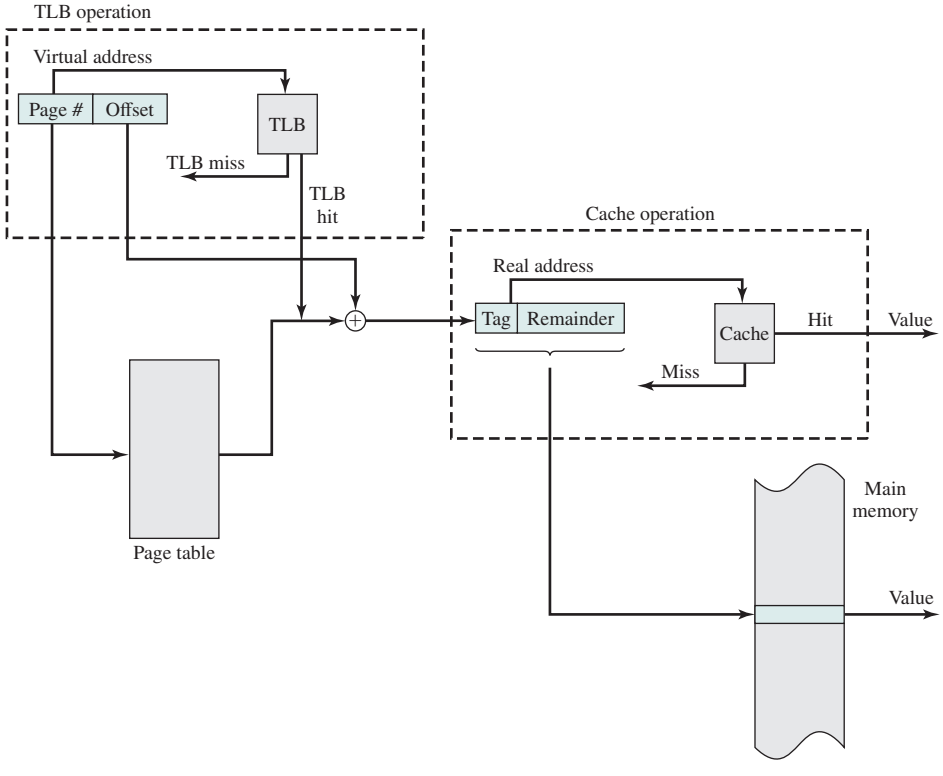


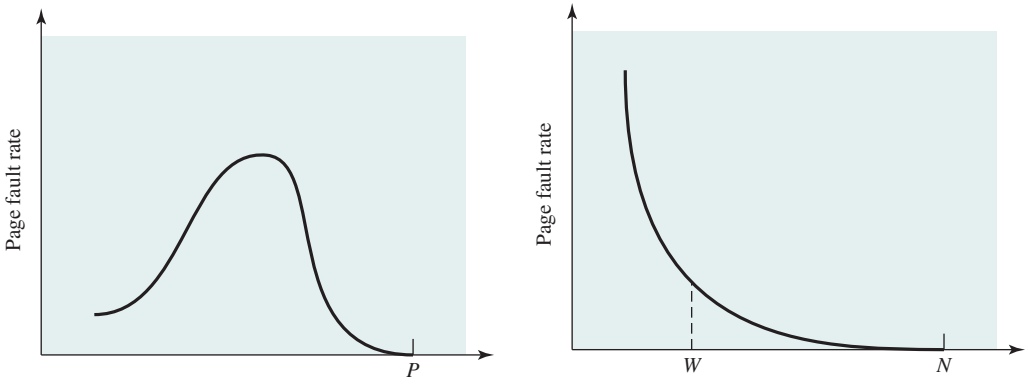
Figure 8.9 Translation Lookaside Buffer and Cache Operation

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 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 8.10a 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,



(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 8.10 Typical Paging Behavior of a Program

Table 8.3 Example of Page Sizes

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

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 8.10b shows that for a fixed page 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.

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

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 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.

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, x86, 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 non-segmented 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 case, however, the segment table entries become more complex (see Figure 8.1b). 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 8.11 suggests a hardware implementation of this scheme (note similarity to Figure 8.2). When a particular process is running, a register holds the starting address of the segment table for that process.

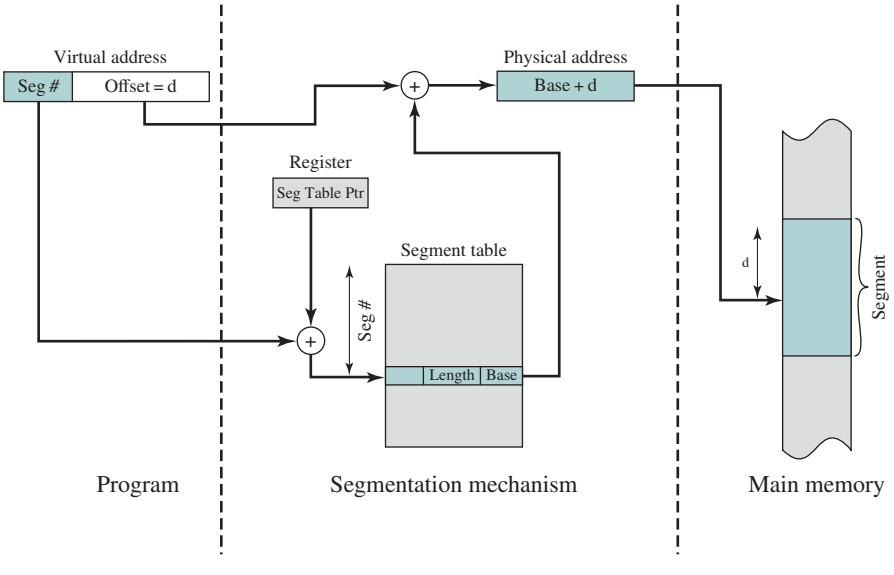


Figure 8.11 Address Translation in a Segmentation System

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 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 8.12 suggests a structure to support combined paging/segmentation (note the similarity to Figure 8.4). Associated with each process is a segment table and a number of page tables, one per process segment. When a particular process is

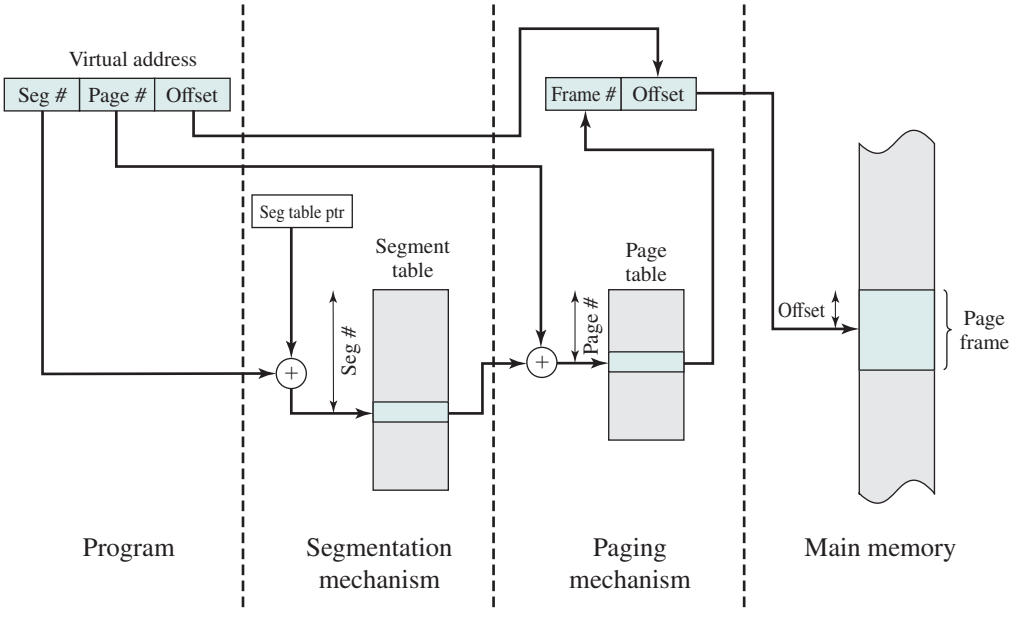


Figure 8.12 Address Translation in a Segmentation/Paging System

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 8.1c 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 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 8.13 illustrates the types of protection relationships that can be enforced in such a system.

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 (see 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:

- A program may access only data that reside on the same ring or a less-privileged ring.
- 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:

1. Whether or not to use virtual memory techniques
2. The use of paging or segmentation or both
3. The algorithms employed for various aspects of memory management

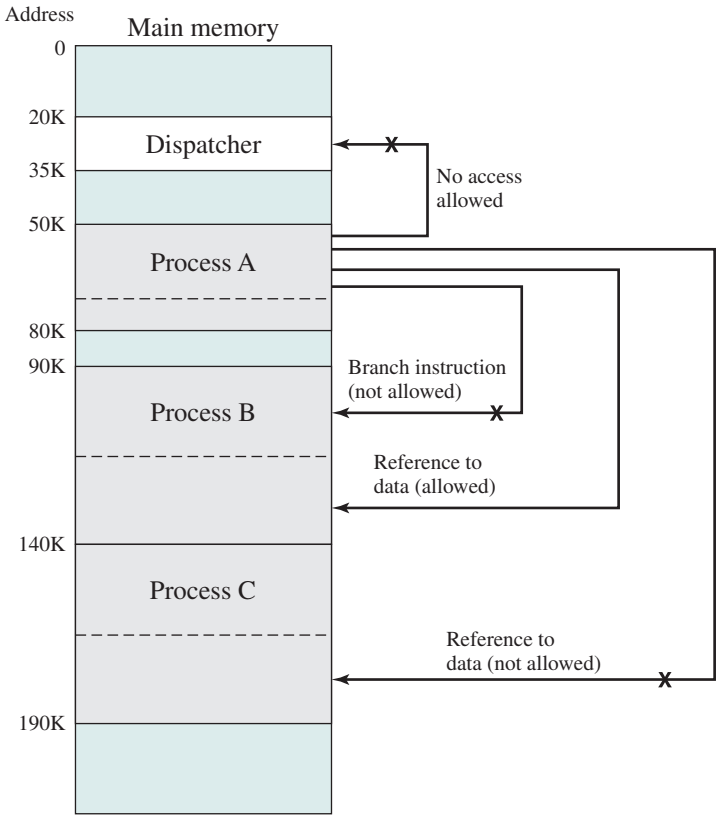


Figure 8.13 **Protection Relationships between Segments**

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

[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.

Table 8.4 Operating System Policies for Virtual Memory

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

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 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.

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.

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.

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 will be 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 determines which page currently in memory is to be replaced. All of the policies have as their objective that the page to be 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.

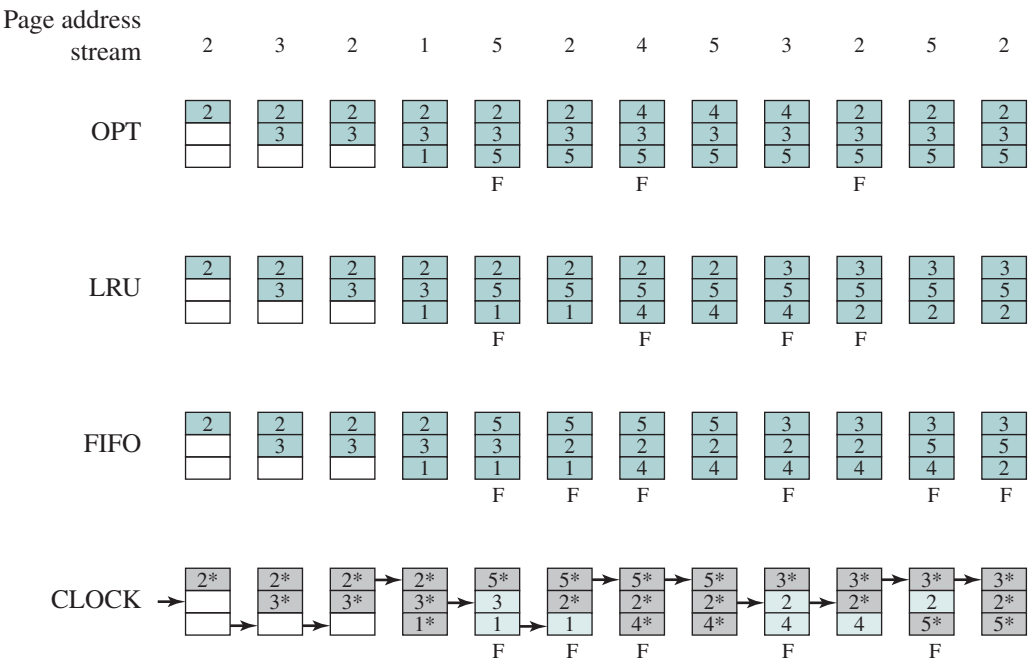


Figure 8.14 Behavior of Four Page Replacement Algorithms

Figure 8.14 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 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 8.14 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.

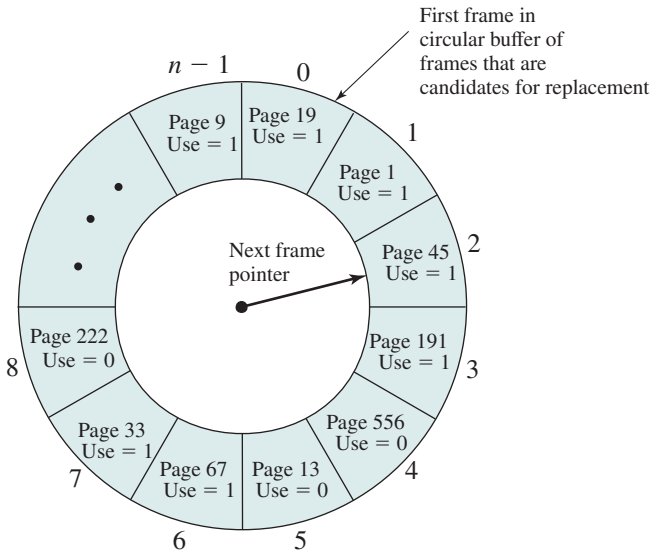
Continuing our example in Figure 8.14, the FIFO policy results in six page faults. Note that LRU recognizes that pages 2 and 5 are referenced more frequently than other pages, whereas FIFO does not.

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 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**.

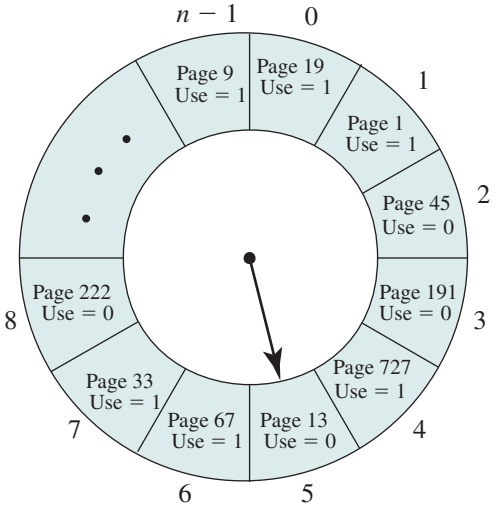
The simplest form of clock policy requires the association of an additional bit with each frame, referred to as the use bit. When a page is first loaded into a frame in memory, the use bit for that frame is set to 1. Whenever the page is subsequently referenced (after the reference that generated the 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 8.15 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,

[5]The concept of scope will be discussed in the subsection "Replacement Scope."



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



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

Figure 8.15 Example of Clock Policy Operation

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 8.14. 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 the clock policy is adept at protecting frames 2 and 5 from replacement.

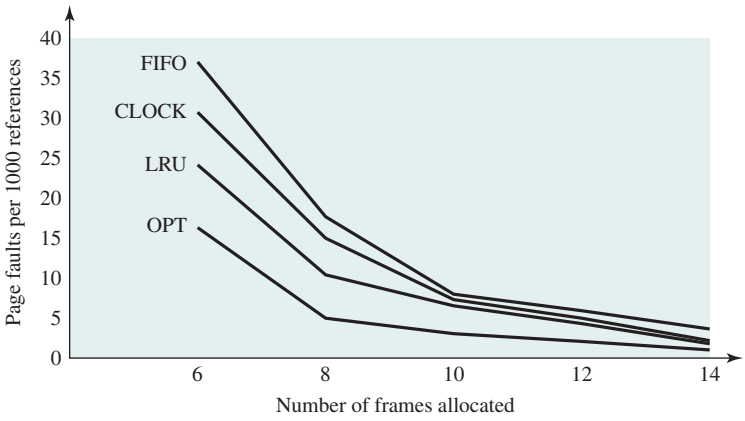


Figure 8.16 **Comparison of Fixed-Allocation, Local Page Replacement Algorithms**

Figure 8.16 shows the results of an experiment reported in [BAER80], which compares the four algorithms that we have been discussing; it is assumed 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 FIFO being over a factor of 2 worse than optimal. All four curves have the same shape as the idealized behavior shown in Figure 8.10b. 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 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 [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

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

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:

1. Not accessed recently, not modified ($u = 0; m = 0$)
2. Accessed recently, not modified ($u = 1; m = 0$)
3. Not accessed recently, modified ($u = 0; m = 1$)
4. 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.
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]. The advantage of this algorithm over 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 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 free page 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.

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

As was stated earlier in this chapter, the portion of a process that is actually in main memory at any time is defined to be the resident set of the process.

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 8.10b).
- 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.

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. As mentioned earlier, when a frame is locked, the page currently stored in that frame may not be replaced. An unlocked page is simply a page in a frame of main memory that is not locked. 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 (see 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

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.

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 will 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 is to be replaced from among the currently resident pages for this process. 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.

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.

Virtual time is defined as follows. Consider a sequence of memory references, $r(1), r(2), \ldots$, 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 8.17 (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

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 8.17 Working Set of Process as Defined by Window Size

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 8.18 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 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.

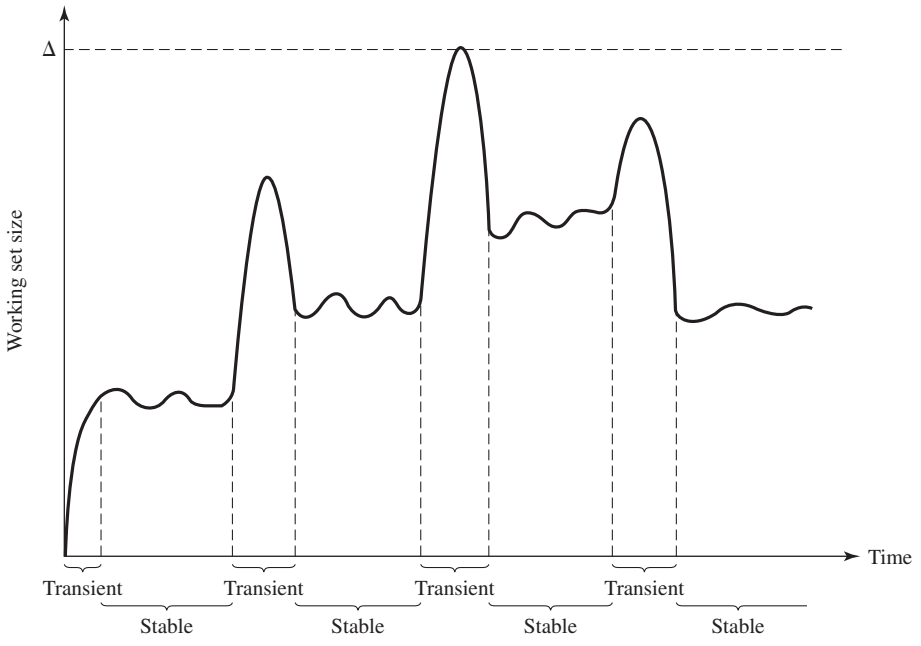


Figure 8.18 Typical Graph of Working Set Size [MAEK87]

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 (see Figure 8.18).
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 virtual time of that process 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 8.10b 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 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 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 pages can be written out in batches.

Both precleaning and demand cleaning have drawbacks. 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.

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 8.19. 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 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 the mean time between faults equals the mean time required to process a page fault. Performance studies indicate 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 this is a point of maximum processor utilization.

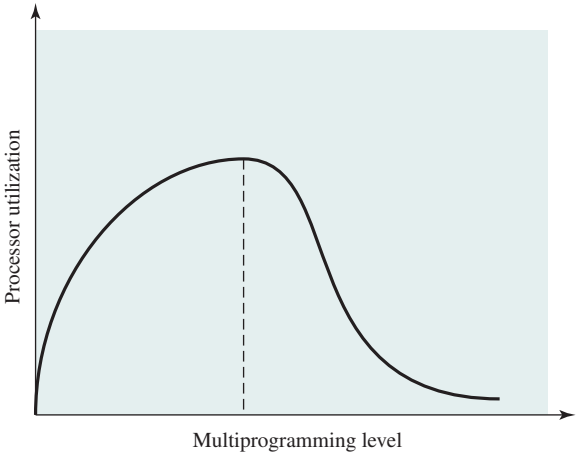


Figure 8.19 Multiprogramming Effects

Another approach is to adapt the clock page replacement algorithm described earlier (see Figure 8.15). [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 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 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.

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 will 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 (see Figure 8.20 and Table 8.6):

- **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.

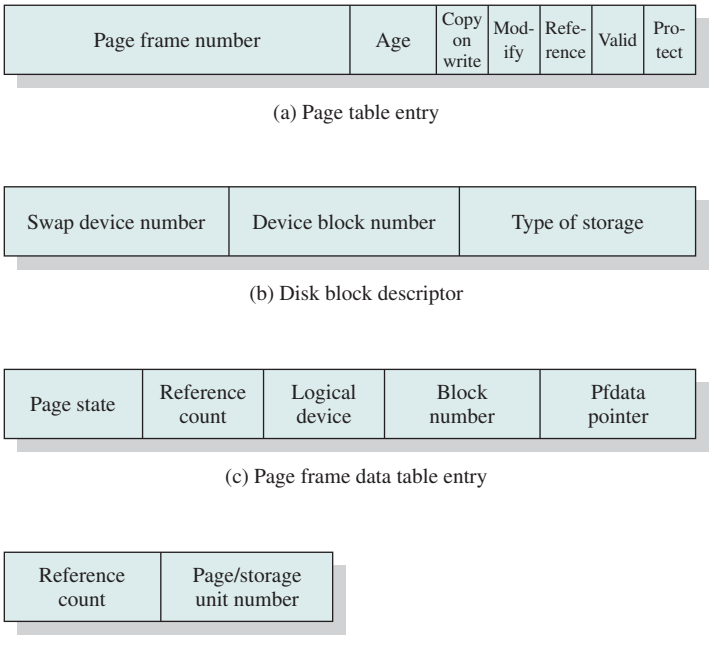


Figure 8.20 UNIX SVR4 Memory Management Formats

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 or not 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.

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 (see Figure 8.15) known as the two-handed clock algorithm (see Figure 8.21). The algorithm uses the reference bit in the page table entry for each page

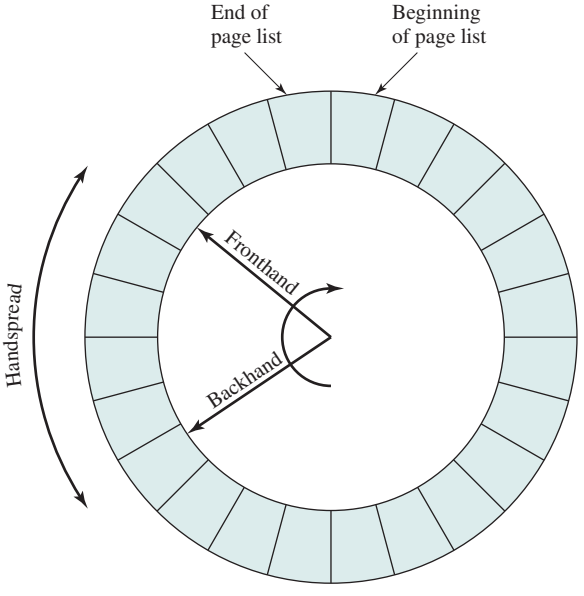


Figure 8.21 Two-Handed Clock Page Replacement Algorithm

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:

1. **Scanrate:** The rate at which the two hands scan through the page list, in pages per second
2. **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 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.

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, 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 $L_i \leq A_i$). 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 8.22 shows the algorithm.

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 $D_i > 2$
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 8.22 **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 will give a brief overview of the two main aspects of Linux memory management: process virtual memory and kernel memory allocation. The basic unit of memory is a physical page, which is represented in the Linux kernel by struct page. The size of this page depends on the architecture; typically it is 4kB . Linux also supports Hugepages, which enables one to set larger sizes for pages (for example, 2MB). There are several projects which use Hugepages in order to improve performance. For example, Data Plane Development Kit (http://dpdk.org/) uses Hugepages for packet buffers, and this decreases the number of Translation Lookaside Buffers accesses on the system, comparing to when using the 4kB page size.

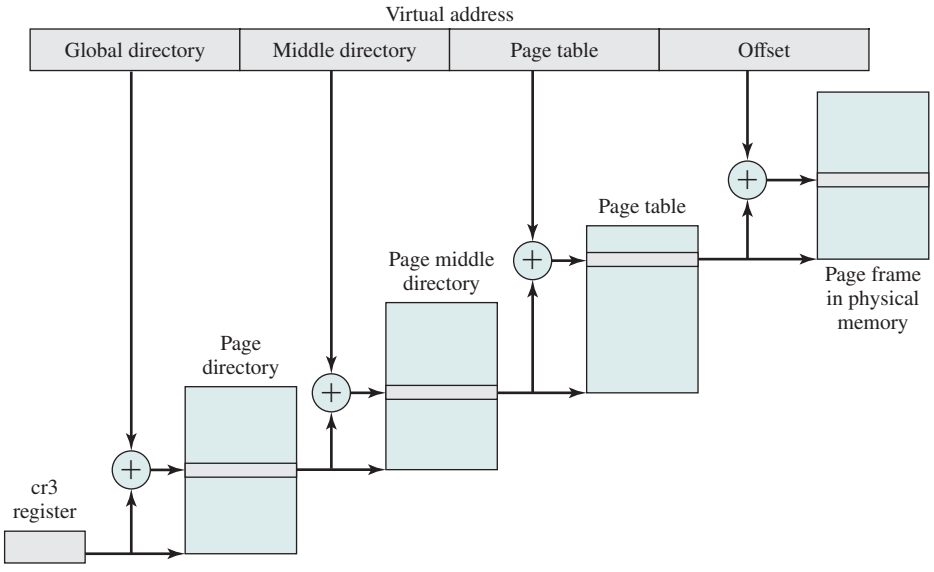


Figure 8.23 Address Translation in Linux Virtual Memory Scheme

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.
- **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 (see Figure 8.23). 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 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 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 are allocated and deallocated in main memory, the available groups are split and merged using the buddy algorithm.

Page Replacement Algorithm Prior to Linux release 2.6.28, the Linux page replacement algorithm was based on the clock algorithm described in Section 8.2 (see Figure 8.15). 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 was 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 was a form of least frequently used policy.

Beginning with Linux release 2.6.28, the page replacement algorithm described in the preceding paragraph was scrapped and a new algorithm, referred to as a split LRU algorithm, was merged into the kernel. One problem with the older algorithm is that the periodic sweeps through the page pool consumes increasing amounts of processor time for increasingly large memories.

The new algorithm makes use of two flags added to each page table entry: PG_active and PG_referenced. The entire physical memory is divided into different "zones" in Linux based on their address. Two linked lists, namely the active and inactive lists, are used in each zone for page reclamation by the memory manager. A kernel daemon `kswapd` runs in the background periodically to perform periodic page reclamation in each zone. This daemon sweeps through the page table entries to which the system page frames are mapped. For all page table entries marked as accessed, PG_referenced bit is set. This bit is set by the processor the first time a page is accessed. For each iteration of kswapd, it checks whether the page accessed bit is set in the page table entry. Every time it reads the page accessed bit, `kswapd` clears the bit. We can summarize the steps involved in page management as follows (see Figure 8.24):

1. The first time a page on the inactive list is accessed, the PG_referenced flag is set.
2. The next time that page is accessed, it is moved to the active list. That is, it takes two accesses for a page to be declared active. More precisely, it takes two accesses in different scans for a page to become active.
3. If the second access doesn't happen soon enough, PG_referenced is reset.
4. Similarly, for active pages, two timeouts are required to move the page to the inactive list.

Pages on the inactive list are then available for page replacement, using an LRU type of algorithm.

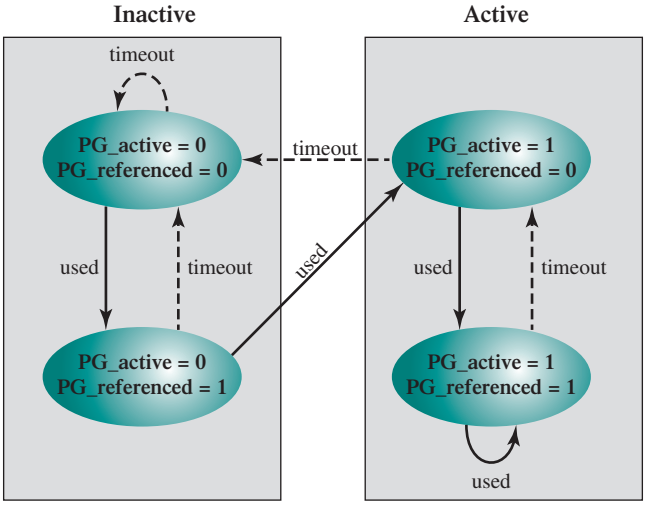


Figure 8.24 Linux Page Reclaiming

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 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 x86 machine, the page size is 4 kB, 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.

While SLAB is the most commonly used, there are three memory allocators in Linux for allocating small chunks of memory:

1. SLAB: Designed to be as cache-friendly as possible, minimizing cache misses.
2. SLUB (unqueued slab allocator): Designed to be simple and minimize instruction count [CORB07].

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

3. SLOB (simple list of blocks): Designed to be as compact as possible; intended for systems with memory limitations [MACK05].

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 kB to 64 kB. Intel and AMD64 platforms have 4 kB per page, and Intel Itanium platforms have 8 kB per page.

Windows Virtual Address Map

On 32-bit platforms, each Windows user process sees a separate 32-bit address space, allowing 4 GB of virtual memory per process. By default, half of this memory is reserved for the OS, so each user actually has 2 GB of available virtual address space and all processes share most of the upper 2 GB 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 8.25 shows the default virtual address space seen by a normal 32-bit user process. It consists of four regions:

1. **0x00000000 to 0x0000FFFF:** Set aside to help programmers catch NULL-pointer assignments.
2. **0x00010000 to 0x7FFEFFFF:** Available user address space. This space is divided into pages that may be loaded into main memory.
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.
4. **0x80000000 to 0xFFFFFFFF:** System address space. This 2-GB process is used for the Windows Executive, Kernel, HAL, and device drivers.

   On 64-bit platforms, 8 TB of user address space is available in Windows.

Windows Paging

When a process is created, it can in principle make use of the entire user space of almost 2 GB (or 8 TB 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-kB boundaries. A region can be in one of three states:

1. **Available:** addresses not currently used by this process.
2. **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).

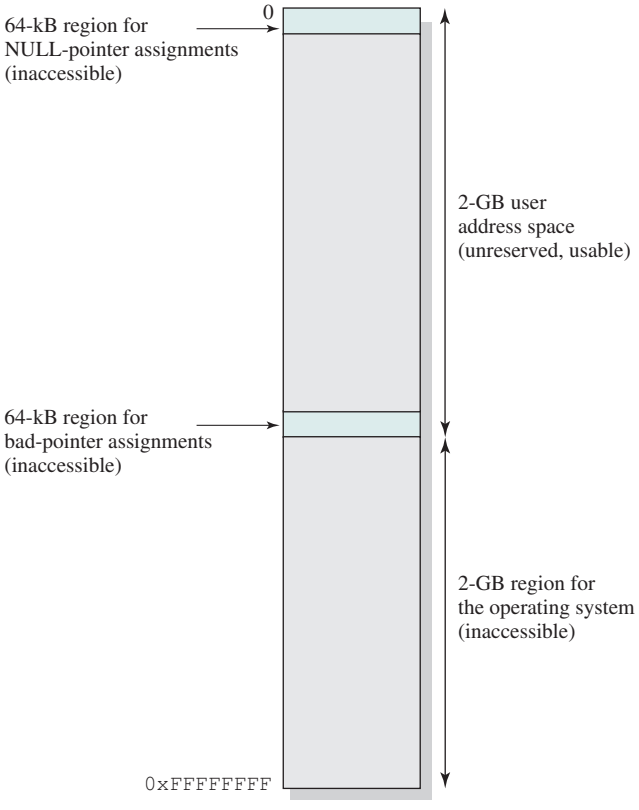


Figure 8.25 Windows Default 32-Bit Virtual Address Space

3. **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 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.

Windows Swapping

With the Metro UI comes a new virtual memory system to handle the interrupt requests from Windows Store apps. Swapfile.sys joins its familiar Windows counterpart pagefile.sys to provide access to temporary memory storage on the hard drive. Paging will hold items that haven't been accessed in a long time, whereas swapping holds items that were recently taken out of memory. The items in pagingfile may not be accessed again for a long time, whereas the items in swapfile might be accessed much sooner. Only Store apps use the swapfile.sys file, and because of the relatively small size of Store apps, the fixed size is only 256MB. The pagefile.sys file will be roughly one to two times the size of the amount of physical RAM found in the system. Swapfile.sys operates by swapping the entire process from system memory into the swapfile. This immediately frees up memory for other applications to use. By contrast, paging files function by moving "pages" of a program from system memory into the paging file. These pages are 4kB in size. The entire program does not get swapped wholesale into the paging file.

8.6 ANDROID MEMORY MANAGEMENT

Android includes a number of extensions to the normal Linux kernel memory management facility. These include the following:

- **ASHMem:** This feature provides anonymous shared memory, which abstracts memory as file descriptors. A file descriptor can be passed to another process to share memory.
- **ION:** ION is a memory pool manager and also enables its clients to share buffers. ION manages one or more memory pools, some of which are set aside at boot time to combat fragmentation or to serve special hardware needs. GPUs, display controllers, and cameras are some of the hardware blocks that may have special memory requirements. ION presents its memory pools as ION heaps. Each type of Android device can be provisioned with a different set of ION heaps according to the memory requirements of the device.
- **Low Memory Killer:** Most mobile devices do not have a swap capability (because of flash memory lifetime considerations). When main memory is

exhausted, the application or applications using the most memory must either back off their use of memory or be terminated. This feature enables the system to notify an app or apps that they need to free up memory. If an app does not cooperate, it is terminated.

8.7 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.

8.8 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

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

Review Questions

8.1. How does the use of virtual memory improve system utilization?
8.2. Explain thrashing.
8.3. Why is the principle of locality crucial to the use of virtual memory?
8.4. Which considerations determine the size of a page?
8.5. What is the purpose of a translation lookaside buffer?
8.6. What is demand paging?
8.7. What are the drawbacks of using either only a precleaning policy or only a demand cleaning policy?
8.8. What is the relationship between FIFO and clock page replacement algorithms?
8.9. How is a page fault trap dealt with?
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 2,048 bytes.

Virtual page number	Valid bit	Reference bit	Modify bit	Page frame number
0	1	1	1	7
1	0	0	0	–
2	0	0	0	–
3	1	0	0	6
4	1	1	0	0
5	1	0	1	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) 6,204
(ii) 3,021
(iii) 9,000

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 the program is running on a system using demand paging, and the page size is 1 kB. 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 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 8.3?
b. Assume you want to implement a hashed inverted page table for the same addressing scheme as depicted in Figure 8.3, using a hash function that maps the 24-bit page number into an 8-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 4 overflow entries per hashed entry, how much memory space does the hashed inverted page table take?

8.4. Consider the following page-reference string: *a, b, d, c, b, e, d, b, d, b, a, c, b, c, a, c, f, a, f, d*. Assume that there are 3 frames available and that they are all initially empty. Complete a figure, similar to Figure 8.14, showing the frame allocation for each of the following page replacement policies:

a. First-in-first-out
b. Optimal
c. Least recently used

Then, find the relative performance of each policy with respect to page faults.

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 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

a. Show the successive pages residing in the four frames using the LRU replacement policy. Compute the hit ratio in main memory. Assume 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. A system has a total of 128 frames. There are 4 processes in the system with the following memory requirements:

$$p_1 : 45 \qquad p_2 : 75 \qquad p_3 : 33 \qquad p_4 : 135$$

Using the following allocation methods, compute the number of frames allocated to each of the processes stated above:

a. Equal Allocation Algorithm
b. Proportional Allocation Algorithm

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 kB or 4 kB, and the segment size is fixed at either 64 kB or 1 MB. For the 370/XA and 370/ESA architectures, the page size is 4 kB and the segment size is 1 MB. Which advantages of segmentation does this scheme lack? What is the benefit of segmentation for the 370?

8.10. Suppose the virtual space accessed by memory is 6 GB, the page size is 8 KB, and each page table entry is 6 bytes. Compute the number of virtual pages that is implied. Also, compute the space required for the whole page table.

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 250 ns, how long does a paged memory reference take?
b. Now we add an MMU that imposes an overhead of 30 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 four frames, initially all empty. The page reference string is of length 20 with six distinct page numbers in it. For any page replacement algorithm,

a. What is the lower bound on the number of page faults? Justify your answer.
b. What is the upper bound on the number of page faults? Justify your answer.

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 off 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.

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

a. What is the maximum size of each segment?
b. What is the maximum logical address space for the task?
c. Assume 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 the following sequence of page references:

A, B, B, C, A, E, D, B, D, E, A, C, E, B, A, C, A, F, D, F

and consider that a working set strategy is used for page replacement. What will the contents of the working set at each stage be for the following?

a. Window Size = 2
b. Window Size = 3
c. Window Size = 4

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.