is placed in the appropriate I/O queue. If it is suspended because of a timeout or because the OS must attend to pressing business, then it is placed in the ready state and put into the short-term queue.

Finally, we mention that the OS also manages the I/O queues. When an I/O operation is completed, the OS removes the satisfied process from that I/O queue and places it in the short-term queue. It then selects another waiting process (if any) and signals for the I/O device to satisfy that process's request.

8.3 MEMORY MANAGEMENT

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

Effective memory management is vital in a multiprogramming system. If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O and the processor will be idle. Thus, memory needs to be allocated efficiently to pack as many processes into memory as possible.

Swapping

Referring back to Figure 8.11, we have discussed three types of queues: the long-term queue of requests for new processes, the short-term queue of processes ready to use the processor, and the various I/O queues of processes that are not ready to use the processor. Recall that the reason for this elaborate machinery is that I/O activities are much slower than computation and therefore the processor in a uniprogramming system is idle most of the time.

But the arrangement in Figure 8.11 does not entirely solve the problem. It is true that, in this case, memory holds multiple processes and that the processor can move to another process when one process is waiting. But the processor is so much faster than I/O that it will be common for *all* the processes in memory to be waiting on I/O. Thus, even with multiprogramming, a processor could be idle most of the time.

What to do? Main memory could be expanded, and so be able to accommodate more processes. But there are two flaws in this approach. First, main memory is expensive, even today. Second, the appetite of programs for memory has grown as fast as the cost of memory has dropped. So larger memory results in larger processes, not more processes.

Another solution is **swapping**, depicted in Figure 8.12. We have a long-term queue of process requests, typically stored on disk. These are brought in, one at a time, as space becomes available. As processes are completed, they are moved out of main memory. Now the situation will arise that none of the processes in memory are in the ready state (e.g., all are waiting on an I/O operation). Rather than remain idle, the processor *swaps* one of these processes back out to disk into an *intermediate queue*. This is a queue of existing processes that have been temporarily

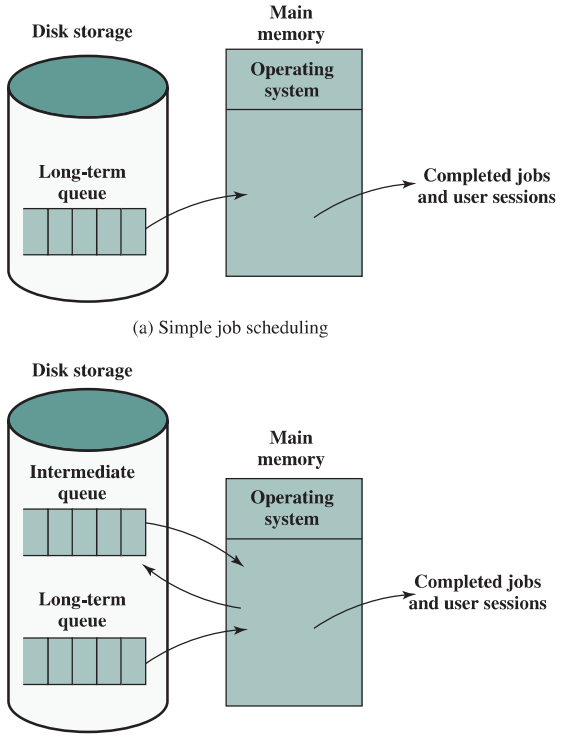


Figure 8.12 The Use of Swapping

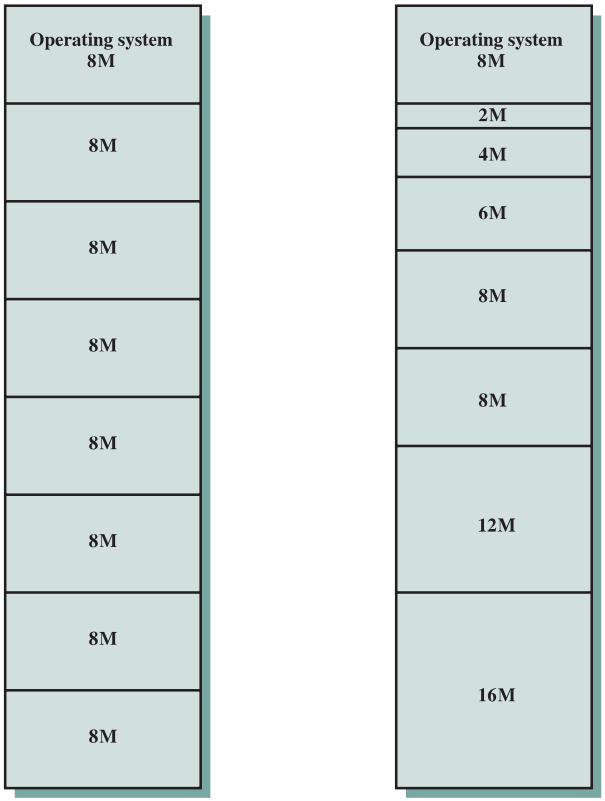
kicked out of memory. The OS then brings in another process from the intermediate queue, or it honors a new process request from the long-term queue. Execution then continues with the newly arrived process.

Swapping, however, is an I/O operation, and therefore there is the potential for making the problem worse, not better. But because disk I/O is generally the fastest I/O on a system (e.g., compared with tape or printer I/O), swapping will usually enhance performance. A more sophisticated scheme, involving virtual memory, improves performance over simple swapping. This will be discussed shortly. But first, we must prepare the ground by explaining partitioning and paging.

Partitioning

The simplest scheme for partitioning available memory is to use *fixed-size partitions*, as shown in Figure 8.13. Note that, although the partitions are of fixed size, they need not be of equal size. When a process is brought into memory, it is placed in the smallest available partition that will hold it.

Even with the use of unequal fixed-size partitions, there will be wasted memory. In most cases, a process will not require exactly as much memory as provided



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

Figure 8.13 Example of Fixed Partitioning of a 64-Mbyte Memory

by the partition. For example, a process that requires 3M bytes of memory would be placed in the 4M partition of Figure 8.13b, wasting 1M that could be used by another process.

A more efficient approach is to use *variable-size partitions.* When a process is brought into memory, it is allocated exactly as much memory as it requires and no more.

EXAMPLE 8.2 An example, using 64 Mbytes of main memory, is shown in Figure 8.14. Initially, main memory is empty, except for the OS (a). The first three processes are loaded in, starting where the OS ends and occupying just enough space for each process (b, c, d). This leaves a "hole" at the end of memory that is too small for a fourth process. At some point, none of the processes in memory is ready. The OS swaps out process 2 (e), which leaves sufficient room to load a new process, process 4 (f). Because process 4 is smaller than process 2, another small hole is created. Later, a point is reached at which none of the processes in main memory is ready, but process 2, in the ready-suspend state, is available. Because there is insufficient room in memory for process 2, the OS swaps process 1 out (g) and swaps process 2 back in (h).

As this example shows, this method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. One technique for overcoming this problem is **compaction**: From time to time, the OS shifts the processes in memory to place all the free memory together in one block. This is a time-consuming procedure, wasteful of processor time.

Before we consider ways of dealing with the shortcomings of partitioning, we must clear up one loose end. Consider Figure 8.14; it should be obvious that a process is not likely to be loaded into the same place in main memory each time it is swapped in. Furthermore, if compaction is used, a process may be shifted while in main memory. A process in memory consists of instructions plus data. The instructions will contain addresses for memory locations of two types:

- Addresses of data items
- Addresses of instructions, used for branching instructions

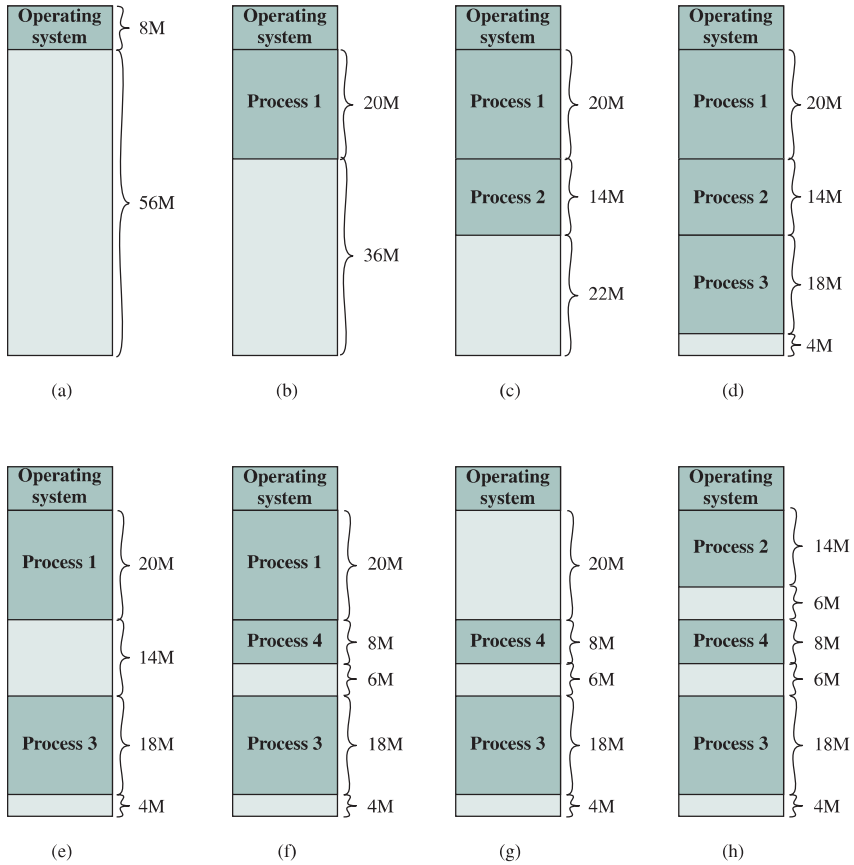


Figure 8.14 The Effect of Dynamic Partitioning

But these addresses are not fixed. They will change each time a process is swapped in. To solve this problem, a distinction is made between logical addresses and physical addresses. A **logical address** is expressed as a location relative to the beginning of the program. Instructions in the program contain only logical addresses. A **physical address** is an actual location in main memory. When the processor executes a process, it automatically converts from logical to physical address by adding the current starting location of the process, called its **base address**, to each logical address. This is another example of a processor hardware feature designed to meet an OS requirement. The exact nature of this hardware feature depends on the memory management strategy in use. We will see several examples later in this chapter.

Paging

Both unequal fixed-size and variable-size partitions are inefficient in the use of memory. Suppose, however, that memory is partitioned into equal fixed-size chunks that are relatively small, and that each process is also divided into small fixed-size chunks of some size. Then the chunks of a program, known as **pages**, could be assigned to available chunks of memory, known as **frames**, or page frames. At most, then, the wasted space in memory for that process is a fraction of the last page.

Figure 8.15 shows an example of the use of pages and frames. At a given point in time, some of the frames in memory are in use and some are free. The list of free frames is maintained by the OS. Process **A**, stored on disk, consists of four pages.

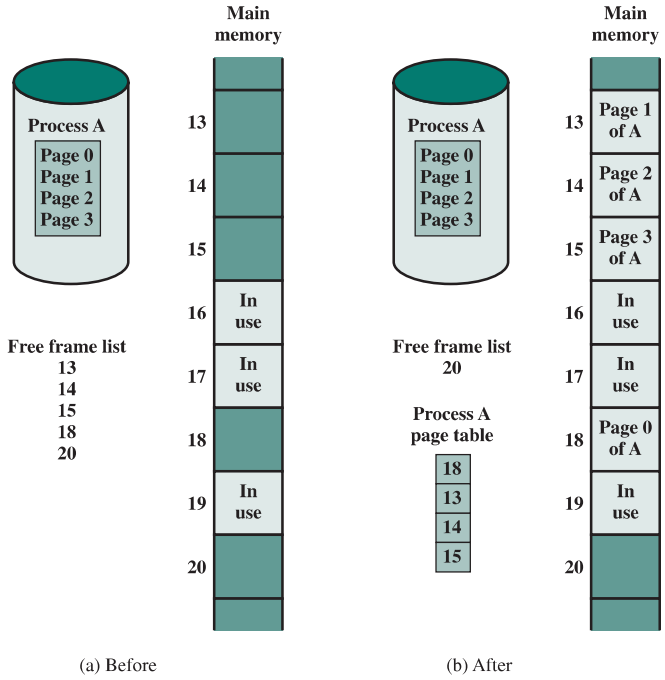


Figure 8.15 Allocation of Free Frames

When it comes time to load this process, the OS finds four free frames and loads the four pages of the process **A** into the four frames.

Now suppose, as in this example, that there are not sufficient unused contiguous frames to hold the process. Does this prevent the OS from loading **A**? The answer is no, because we can once again use the concept of logical address. A simple base address will no longer suffice. Rather, the OS maintains a **page table** for each process. The page table shows the frame location for each page of the process. Within the program, each logical address consists of a page number and a relative address within the page. Recall that in the case of simple partitioning, a logical address is the location of a word relative to the beginning of the program; the processor translates that into a physical address. With paging, the logical-to-physical address translation is still done by processor hardware. The processor must know how to access the page table of the current process. Presented with a logical address (page number, relative address), the processor uses the page table to produce a physical address (frame number, relative address). An example is shown in Figure 8.16.

This approach solves the problems raised earlier. Main memory is divided into many small equal-size frames. Each process is divided into frame-size pages: smaller processes require fewer pages, larger processes require more. When a process is brought in, its pages are loaded into available frames, and a page table is set up.

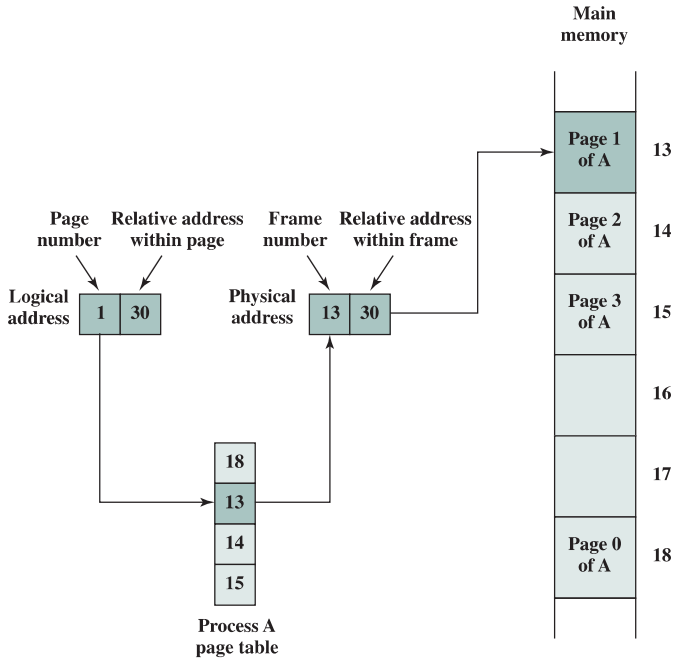


Figure 8.16 Logical and Physical Addresses

Virtual Memory

DEMAND PAGING With the use of paging, truly effective multiprogramming systems came into being. Furthermore, the simple tactic of breaking a process up into pages led to the development of another important concept: virtual memory.

To understand virtual memory, we must add a refinement to the paging scheme just discussed. That refinement is **demand paging**, which simply means that each page of a process is brought in only when it is needed, that is, on demand.

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 perhaps only one or two arrays of data are being used. This is the principle of locality, which we introduced in Appendix 4A. It would clearly be wasteful to load in dozens of pages for that process when only a few pages will be used before the program is suspended. We can make better use of memory by loading in just a few pages. Then, if the program branches to an instruction on a page not in main memory, or if the program references data on a page not in memory, a **page fault** is triggered. This tells the OS to bring in the desired page.

Thus, at any one time, only a few pages of any given process are in memory, and therefore more processes can be maintained in memory. Furthermore, time is saved because unused pages are not swapped in and out of memory. However, the OS must be clever about how it manages this scheme. When it brings one page in, it must throw another page out; this is known as **page replacement**. If it throws out a page just before it is about to be used, then it will just have to go get that page again almost immediately. Too much of this leads to a condition known as **thrashing**: the processor spends most of its time swapping pages 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 pages are least likely to be used in the near future.

Page Replacement Algorithm Simulators

A discussion of page replacement algorithms is beyond the scope of this chapter. A potentially effective technique is least recently used (LRU), the same algorithm discussed in Chapter 4 for cache replacement. In practice, LRU is difficult to implement for a virtual memory paging scheme. Several alternative approaches that seek to approximate the performance of LRU are in use; see Appendix K for details.

With demand paging, it is not necessary to load an entire process into main memory. This fact has a remarkable consequence: *It is possible for a process to be larger than all of main memory.* One of the most fundamental restrictions in programming has been lifted. Without demand paging, 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 one at a time. With demand paging, 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.

Because a process executes only in main memory, that memory is referred to as **real memory**. But a programmer or user perceives a much larger memory—that which is allocated on the disk. This latter is therefore referred to as **virtual memory**. Virtual memory allows for very effective multiprogramming and relieves the user of the unnecessarily tight constraints of main memory.

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 8.16 suggests a hardware implementation of this scheme. 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.

In most systems, there is one page table per process. But each process can occupy huge amounts of virtual memory. For example, in the VAX architecture, each process can have up to $2^{31} = 2$ Gbytes of virtual memory. Using $2^9 = 512 -$ byte pages, that means that as many as 2^{22} page table entries are required *per process.* Clearly, the amount of memory devoted to page tables alone could be unacceptably high. To overcome this problem, most virtual memory schemes store page tables in virtual memory rather than real memory. This means that page tables are subject to paging just as other pages are. When a process is running, at least a part of its page table must be in main memory, including the page table entry of the currently executing page. Some processors make use of a two-level scheme to organize large page tables. In this scheme, there is a page directory, in which each entry points to a page table. Thus, if the length of the page directory is X, and if the maximum length of a page table is Y, then a process can consist of up to $X \times Y$ pages. Typically, the maximum length of a page table is restricted to be equal to one page. We will see an example of this two-level approach when we consider the Intel x86 later in this chapter.

An alternative approach to the use of one- or two-level page tables is the use of an inverted page table structure (Figure 8.17). Variations on this approach are used on the PowerPC, UltraSPARC, and the IA-64 architecture. An implementation of the Mach OS 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.[2] The hash value is a pointer to the inverted page table, which contains the page table entries. There is one entry in the

[2]A hash function maps numbers in the range 0 through M into numbers in the range 0 through N, where $M > N$. The output of the hash function is used as an index into the hash table. Since more than one input maps into the same output, it is possible for an input item to map to a hash table entry that is already occupied. In that case, the new item must *overflow* into another hash table location. Typically, the new item is placed in the first succeeding empty space, and a pointer from the original location is provided to chain the entries together. See Appendix L for more information on hash functions.

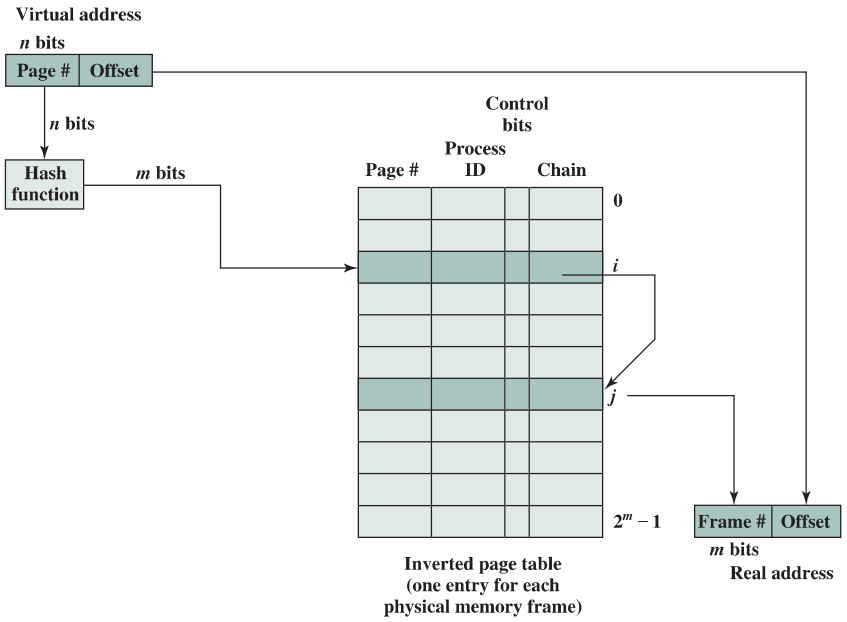


Figure 8.17 Inverted Page Table Structure

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.

Translation Lookaside Buffer

In principle, then, every virtual memory reference can cause two physical memory accesses: one to fetch the appropriate page table entry, and one to fetch the desired data. Thus, a straightforward virtual memory scheme would have the effect of doubling the memory access time. To overcome this problem, most virtual memory schemes make use of a special cache for page table entries, usually called a **translation lookaside buffer (TLB)**. This cache functions in the same way as a memory cache and contains those page table entries that have been most recently used. Figure 8.18 is a flowchart that shows the use of the TLB. 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 that this scheme can significantly improve performance [CLAR85, SATY81].

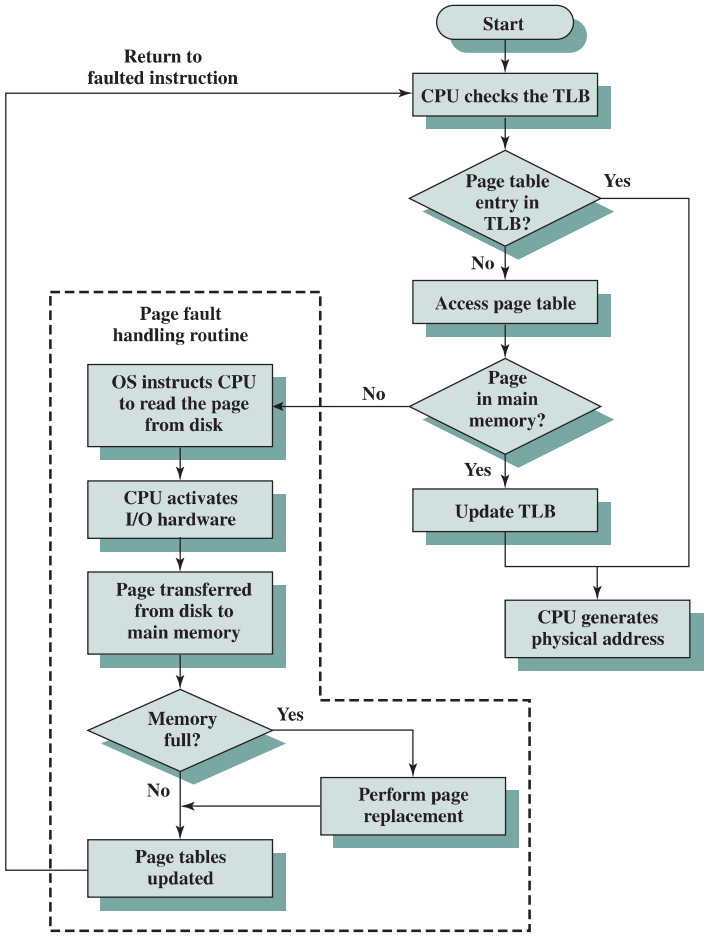


Figure 8.18 Operation of Paging and Translation Lookaside Buffer (TLB)

Note that the virtual memory mechanism must interact with the cache system (not the TLB cache, but the main memory cache). This is illustrated in Figure 8.19. 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 and a remainder, the cache is consulted to see if the block containing that word is present (see Figure 4.5). If so, it is returned to the processor. If not, the word is retrieved from main memory.

The reader should be able to appreciate the complexity of the processor hardware involved in a single memory reference. The virtual address is translated into a real address. This involves reference to a page table, which may be in the TLB, in

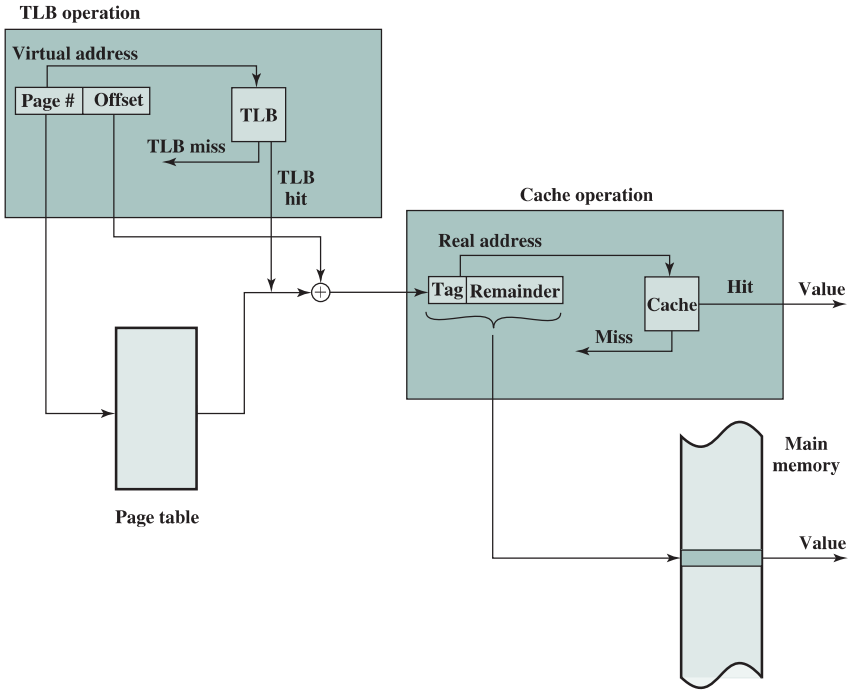


Figure 8.19 Translation Lookaside Buffer and Cache Operation

main memory, or on disk. The referenced word may be in cache, in main memory, or on disk. In the latter case, 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.

Segmentation

There is another way in which addressable memory can be subdivided, known as *segmentation*. Whereas paging is invisible to the programmer and serves the purpose of providing the programmer with a larger address space, segmentation is usually visible to the programmer and is provided as a convenience for organizing programs and data and as a means for associating privilege and protection attributes with instructions and data.

Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments. Segments are of variable, indeed dynamic, size. Typically, the programmer or the OS will assign programs and data to different segments. There may be a number of program segments for various types of programs as well as a number of data segments. Each segment may be assigned access and usage rights. 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 not necessary to guess. The data structure can be assigned its own segment, and the OS will expand or shrink the segment as needed.
2. It allows programs to be altered and recompiled independently without requiring that an entire set of programs 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 addressed 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 a system administrator can assign access privileges in a convenient fashion.

These advantages are not available with paging, which is invisible to the programmer. On the other hand, we have seen that paging provides for an efficient form of memory management. To combine the advantages of both, some systems are equipped with the hardware and OS software to provide both.

8.4 INTEL x86 MEMORY MANAGEMENT

Since the introduction of the 32-bit architecture, microprocessors have evolved sophisticated memory management schemes that build on the lessons learned with medium- and large-scale systems. In many cases, the microprocessor versions are superior to their larger-system antecedents. Because the schemes were developed by the microprocessor hardware vendor and may be employed with a variety of operating systems, they tend to be quite general purpose. A representative example is the scheme used on the Intel x86 architecture.

Address Spaces

The x86 includes hardware for both segmentation and paging. Both mechanisms can be disabled, allowing the user to choose from four distinct views of memory:

- **Unsegmented unpaged memory:** In this case, the virtual address is the same as the physical address. This is useful, for example, in low-complexity, high-performance controller applications.
- **Unsegmented paged memory:** Here memory is viewed as a paged linear address space. Protection and management of memory is done via paging. This is favored by some operating systems (e.g., Berkeley UNIX).
- **Segmented unpaged memory:** Here memory is viewed as a collection of logical address spaces. The advantage of this view over a paged approach is that it affords protection down to the level of a single byte, if necessary. Furthermore, unlike paging, it guarantees that the translation table needed (the segment table) is on-chip when the segment is in memory. Hence, segmented unpaged memory results in predictable access times.

- **Segmented paged memory:** Segmentation is used to define logical memory partitions subject to access control, and paging is used to manage the allocation of memory within the partitions. Operating systems such as UNIX System V favor this view.

Segmentation

When segmentation is used, each virtual address (called a logical address in the x86 documentation) consists of a 16-bit segment reference and a 32-bit offset. Two bits of the segment reference deal with the protection mechanism, leaving 14 bits for specifying a particular segment. Thus, with unsegmented memory, the user's virtual memory is $2^{32} = 4$ Gbytes. With segmented memory, the total virtual memory space as seen by a user is $2^{46} = 64$ terabytes (Tbytes). The physical address space employs a 32-bit address for a maximum of 4 Gbytes.

The amount of virtual memory can actually be larger than the 64 Tbytes. This is because the processor's interpretation of a virtual address depends on which process is currently active. Virtual address space is divided into two parts. One-half of the virtual address space (8K segments $\times$ 4 Gbytes) is global, shared by all processes; the remainder is local and is distinct for each process.

Associated with each segment are two forms of protection: privilege level and access attribute. There are four privilege levels, from most protected (level 0) to least protected (level 3). The privilege level associated with a data segment is its "classification"; the privilege level associated with a program segment is its "clearance." An executing program may only access data segments for which its clearance level is lower than (more privileged) or equal to (same privilege) the privilege level of the data segment.

The hardware does not dictate how these privilege levels are to be used; this depends on the OS design and implementation. It was intended that privilege level 1 would be used for most of the OS, and level 0 would be used for that small portion of the OS devoted to memory management, protection, and access control. This leaves two levels for applications. In many systems, applications will reside at level 3, with level 2 being unused. Specialized application subsystems that must be protected because they implement their own security mechanisms are good candidates for level 2. Some examples are database management systems, office automation systems, and software engineering environments.

In addition to regulating access to data segments, the privilege mechanism limits the use of certain instructions. Some instructions, such as those dealing with memory-management registers, can only be executed in level 0. I/O instructions can only be executed up to a certain level that is designated by the OS; typically, this will be level 1.

The access attribute of a data segment specifies whether read/write or read-only accesses are permitted. For program segments, the access attribute specifies read/execute or read-only access.

The address translation mechanism for segmentation involves mapping a virtual address into what is referred to as a linear address (Figure 8.20b). A virtual address consists of the 32-bit offset and a 16-bit segment selector (Figure 8.20a). An instruction fetching or storing an operand specifies the offset and a register containing the segment selector. The segment selector consists of the following fields:

- **Table Indicator (TI):** Indicates whether the global segment table or a local segment table should be used for translation.

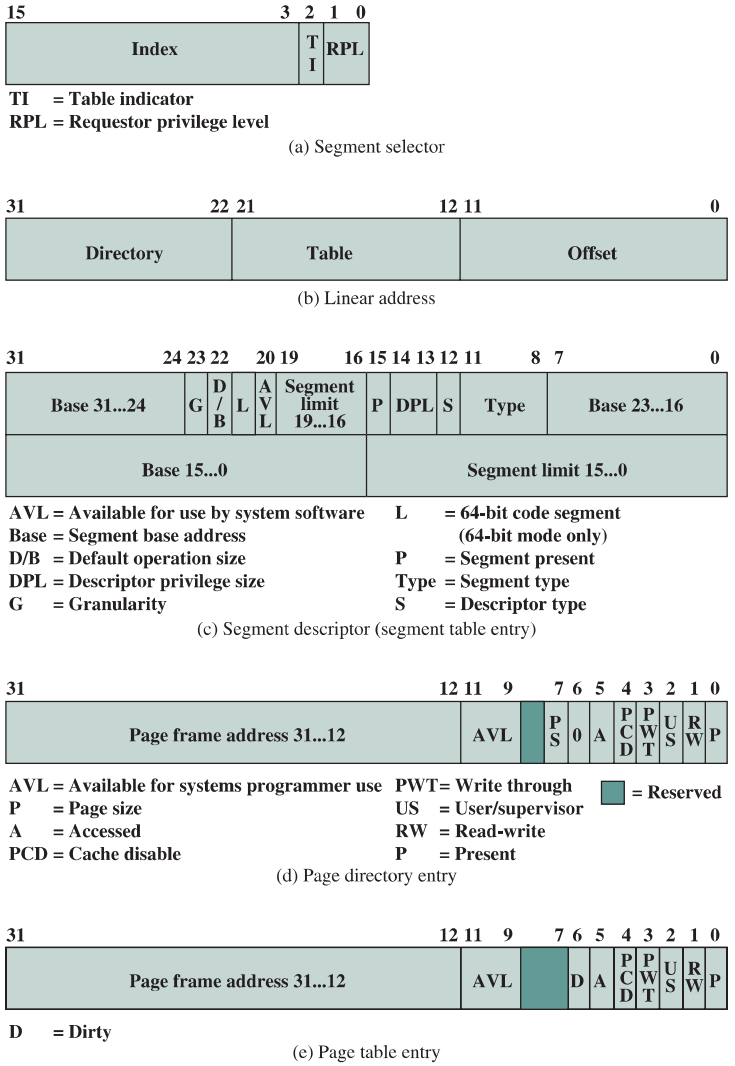


Figure 8.20 Intel x86 Memory Management Formats

- **Segment Number:** The number of the segment. This serves as an index into the segment table.
- **Requested Privilege Level (RPL):** The privilege level requested for this access.

Each entry in a segment table consists of 64 bits, as shown in Figure 8.20c. The fields are defined in Table 8.5.

Table 8.5 x86 Memory Management Parameters

Segment Descriptor (Segment Table Entry)
Base Defines the starting address of the segment within the 4-Gbyte linear address space.
D/B bit In a code segment, this is the D bit and indicates whether operands and addressing modes are 16 or 32 bits.
Descriptor Privilege Level (DPL) Specifies the privilege level of the segment referred to by this segment descriptor.
Granularity bit (G) Indicates whether the Limit field is to be interpreted in units by one byte or 4 Kbytes.
Limit Defines the size of the segment. The processor interprets the limit field in one of two ways, depending on the granularity bit: in units of one byte, up to a segment size limit of 1 Mbyte, or in units of 4 Kbytes, up to a segment size limit of 4 Gbytes.
S bit Determines whether a given segment is a system segment or a code or data segment.
Segment Present bit (P) Used for nonpaged systems. It indicates whether the segment is present in main memory. For paged systems, this bit is always set to 1.
Type Distinguishes between various kinds of segments and indicates the access attributes.
Page Directory Entry and Page Table Entry
Accessed bit (A) This bit is set to 1 by the processor in both levels of page tables when a read or write operation to the corresponding page occurs.
Dirty bit (D) This bit is set to 1 by the processor when a write operation to the corresponding page occurs.
Page Frame Address Provides the physical address of the page in memory if the present bit is set. Since page frames are aligned on 4K boundaries, the bottom 12 bits are 0, and only the top 20 bits are included in the entry. In a page directory, the address is that of a page table.
Page Cache Disable bit (PCD) Indicates whether data from page may be cached.
Page Size bit (PS) Indicates whether page size is 4 Kbyte or 4 Mbyte.
Page Write Through bit (PWT) Indicates whether write-through or write-back caching policy will be used for data in the corresponding page.
Present bit (P) Indicates whether the page table or page is in main memory.
Read/Write bit (RW) For user-level pages, indicates whether the page is read-only access or read/write access for user-level programs.
User/Supervisor bit (US) Indicates whether the page is available only to the operating system (supervisor level) or is available to both operating system and applications (user level).

Paging

Segmentation is an optional feature and may be disabled. When segmentation is in use, addresses used in programs are virtual addresses and are converted into linear addresses, as just described. When segmentation is not in use, linear addresses are used in programs. In either case, the following step is to convert that linear address into a real 32-bit address.

To understand the structure of the linear address, you need to know that the x86 paging mechanism is actually a two-level table lookup operation. The first level is a page directory, which contains up to 1024 entries. This splits the 4-Gbyte linear memory space into 1024 page groups, each with its own page table, and each 4 Mbytes in length. Each page table contains up to 1024 entries; each entry corresponds to a single 4-Kbyte page. Memory management has the option of using one page directory for all processes, one page directory for each process, or some combination of the two. The page directory for the current task is always in main memory. Page tables may be in virtual memory.

Figure 8.20 shows the formats of entries in page directories and page tables, and the fields are defined in Table 8.5. Note that access control mechanisms can be provided on a page or page group basis.

The x86 also makes use of a translation lookaside buffer. The buffer can hold 32 page table entries. Each time that the page directory is changed, the buffer is cleared.

Figure 8.21 illustrates the combination of segmentation and paging mechanisms. For clarity, the translation lookaside buffer and memory cache mechanisms are not shown.

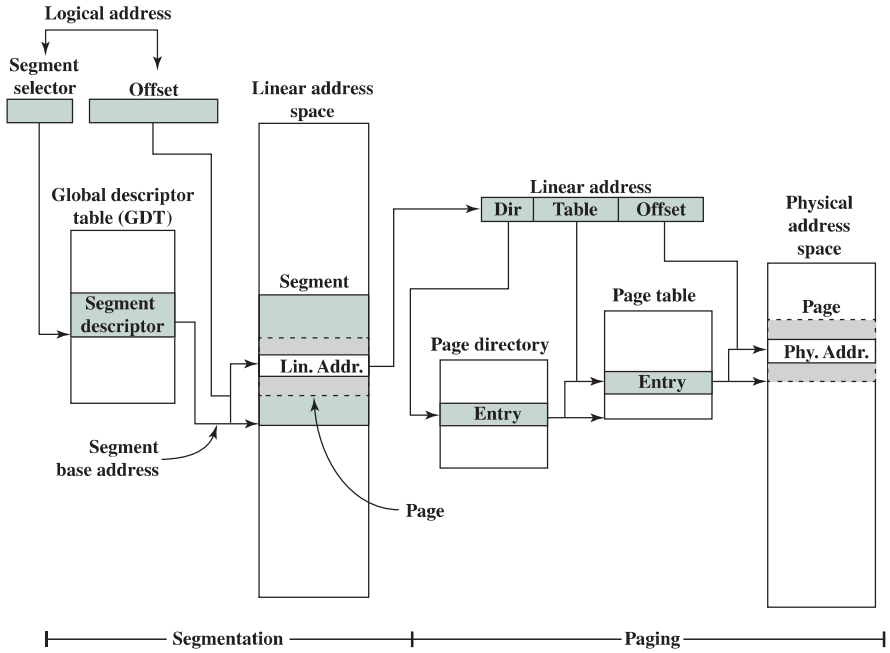


Figure 8.21 Intel x86 Memory Address Translation Mechanisms

Finally, the x86 includes a new extension not found on the earlier 80386 or 80486, the provision for two page sizes. If the PSE (page size extension) bit in control register 4 is set to 1, then the paging unit permits the OS programmer to define a page as either 4 Kbyte or 4 Mbyte in size.

When 4-Mbyte pages are used, there is only one level of table lookup for pages. When the hardware accesses the page directory, the page directory entry (Figure 8.20d) has the PS bit set to 1. In this case, bits 9 through 21 are ignored and bits 22 through 31 define the base address for a 4-Mbyte page in memory. Thus, there is a single page table.

The use of 4-Mbyte pages reduces the memory-management storage requirements for large main memories. With 4-Kbyte pages, a full 4-Gbyte main memory requires about 4 Mbytes of memory just for the page tables. With 4-Mbyte pages, a single table, 4 Kbytes in length, is sufficient for page memory management.

8.5 ARM MEMORY MANAGEMENT

ARM provides a versatile virtual memory system architecture that can be tailored to the needs of the embedded system designer.

Memory System Organization

Figure 8.22 provides an overview of the memory management hardware in the ARM for virtual memory. The virtual memory translation hardware uses one or two levels of tables for translation from virtual to physical addresses, as explained subsequently. The translation lookaside buffer (TLB) is a cache of recent page table entries. If an entry is available in the TLB, then the TLB directly sends a physical address to main memory for a read or write operation. As explained in Chapter 4, data is exchanged

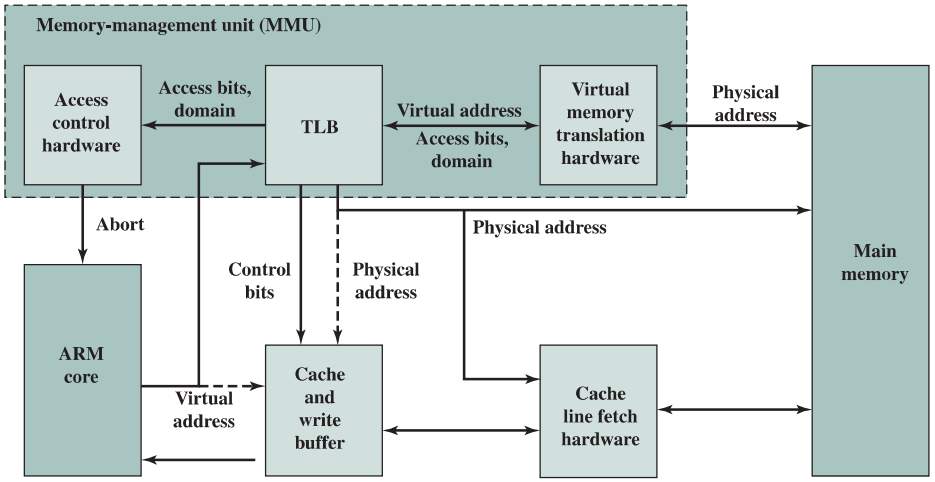


Figure 8.22 ARM Memory System Overview