

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

²A 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.

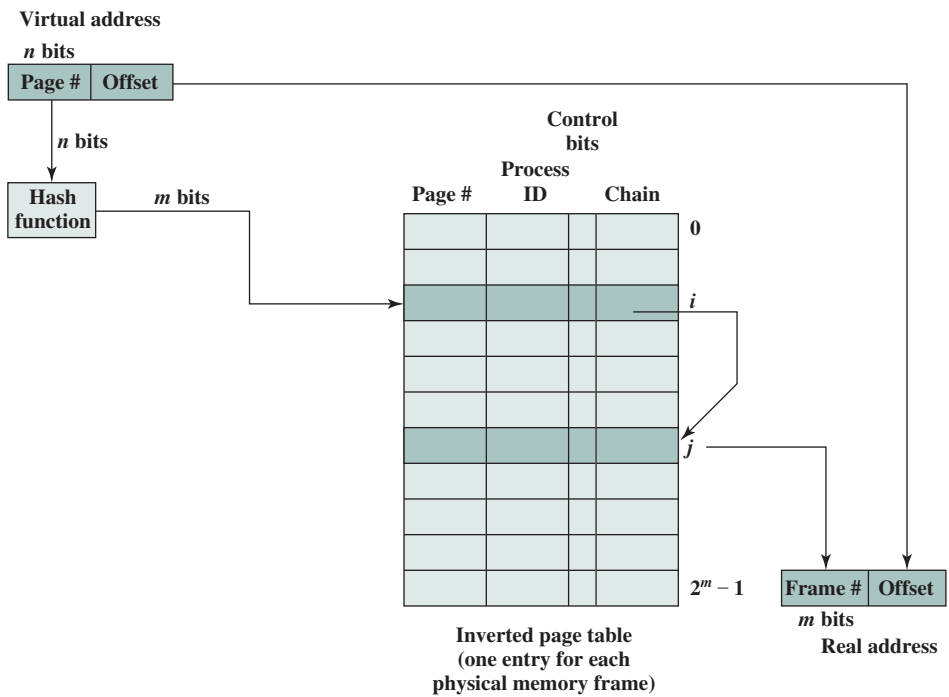


Figure 8.17 Inverted Page Table Structure

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].

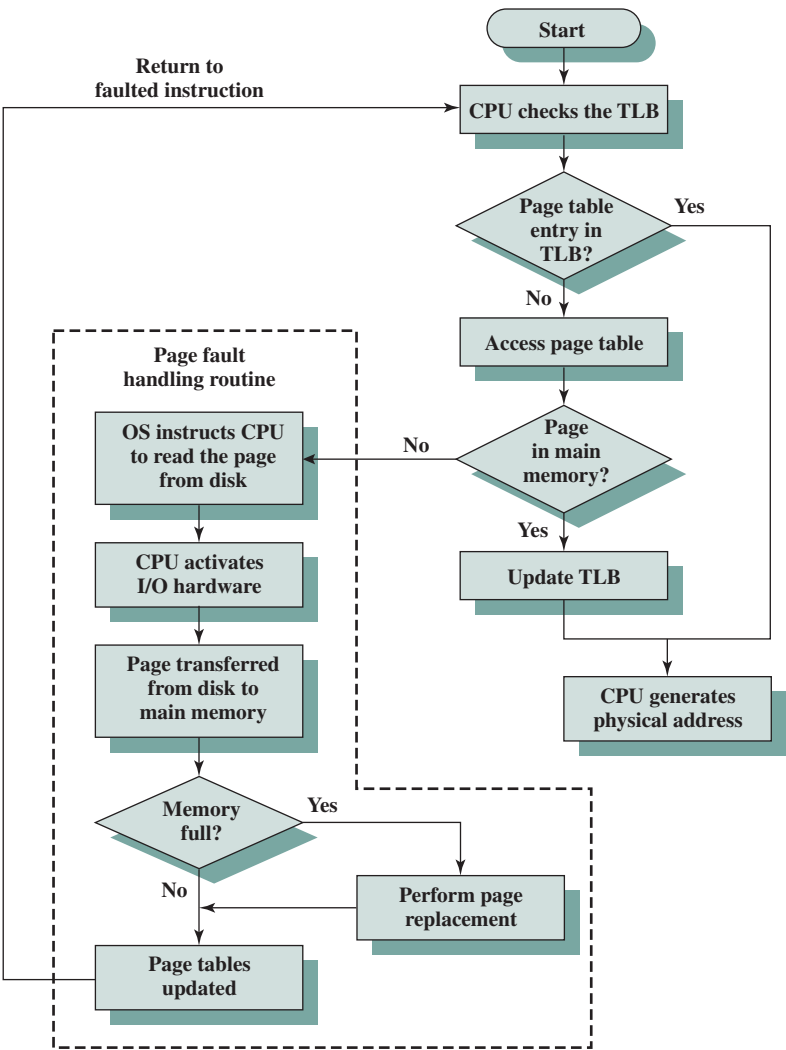


Figure 8.18 Operation of Paging and Translation Lookaside Buffer (TLB)

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

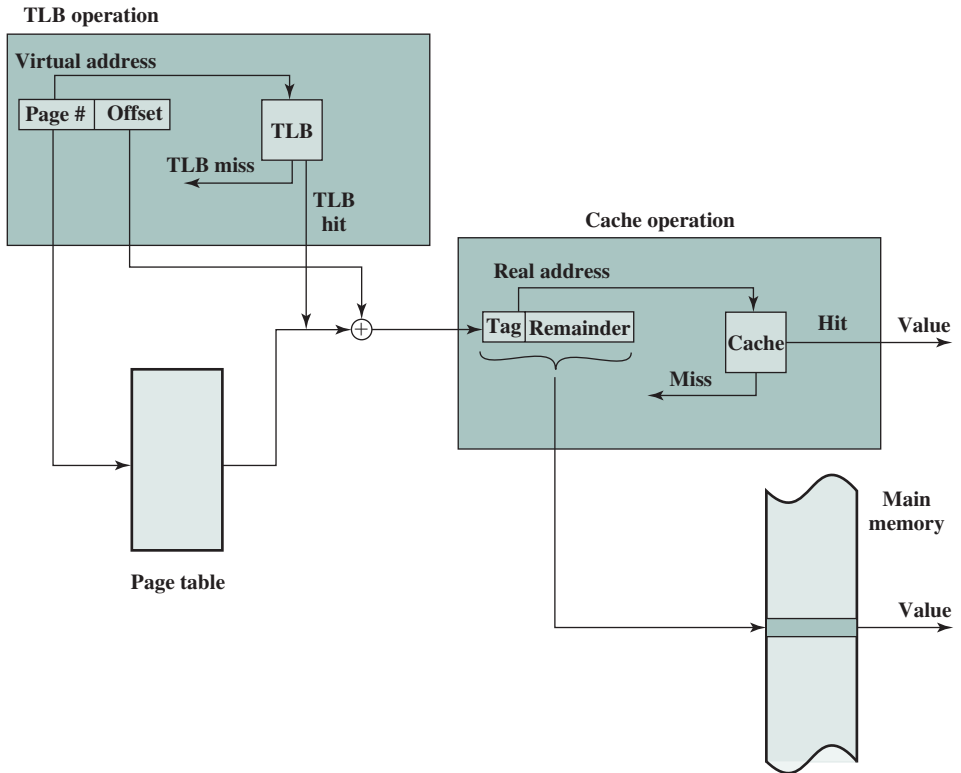


Figure 8.19 Translation Lookaside Buffer and Cache Operation

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.