Memory Management

Address Binding

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process.

Depending on the memory management in use, the process may be moved between disk and memory during its execution.

The processes on the disk that are waiting to be brought into memory for execution form the **input queue**.

In most cases, a user program goes through several steps before being executed (Figure 8.3).

Addresses may be represented in different ways during these steps.

Addresses in the source program are generally symbolic (such as the variable count).

A ***compiler*** typically **binds** these symbolic addresses to relocatable addresses.

The ***linkage editor or loader*** in turn binds the relocatable addresses to absolute addresses (such as 74014).

Each binding is a mapping from one address space to another.

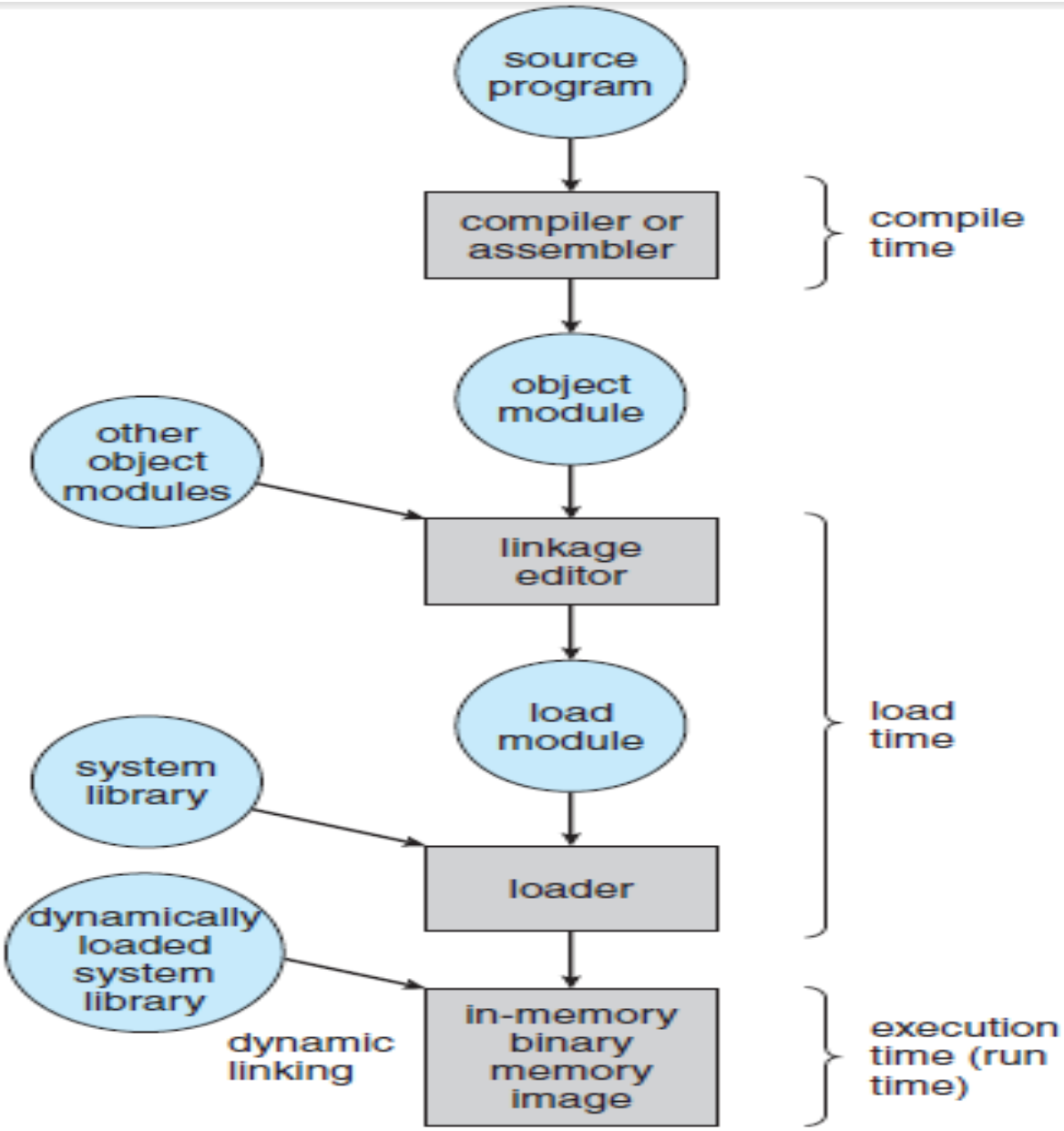

Figure 8.3 Multistep processing of a user program.

Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into the memory-address register of the memory—is commonly referred to as **a physical address**.

The set of all logical addresses generated by a program is a **logical (virtual) address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU).**

The base register is called a **relocation register.** The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (see Figure 8.4).

The user program never sees the real physical addresses.

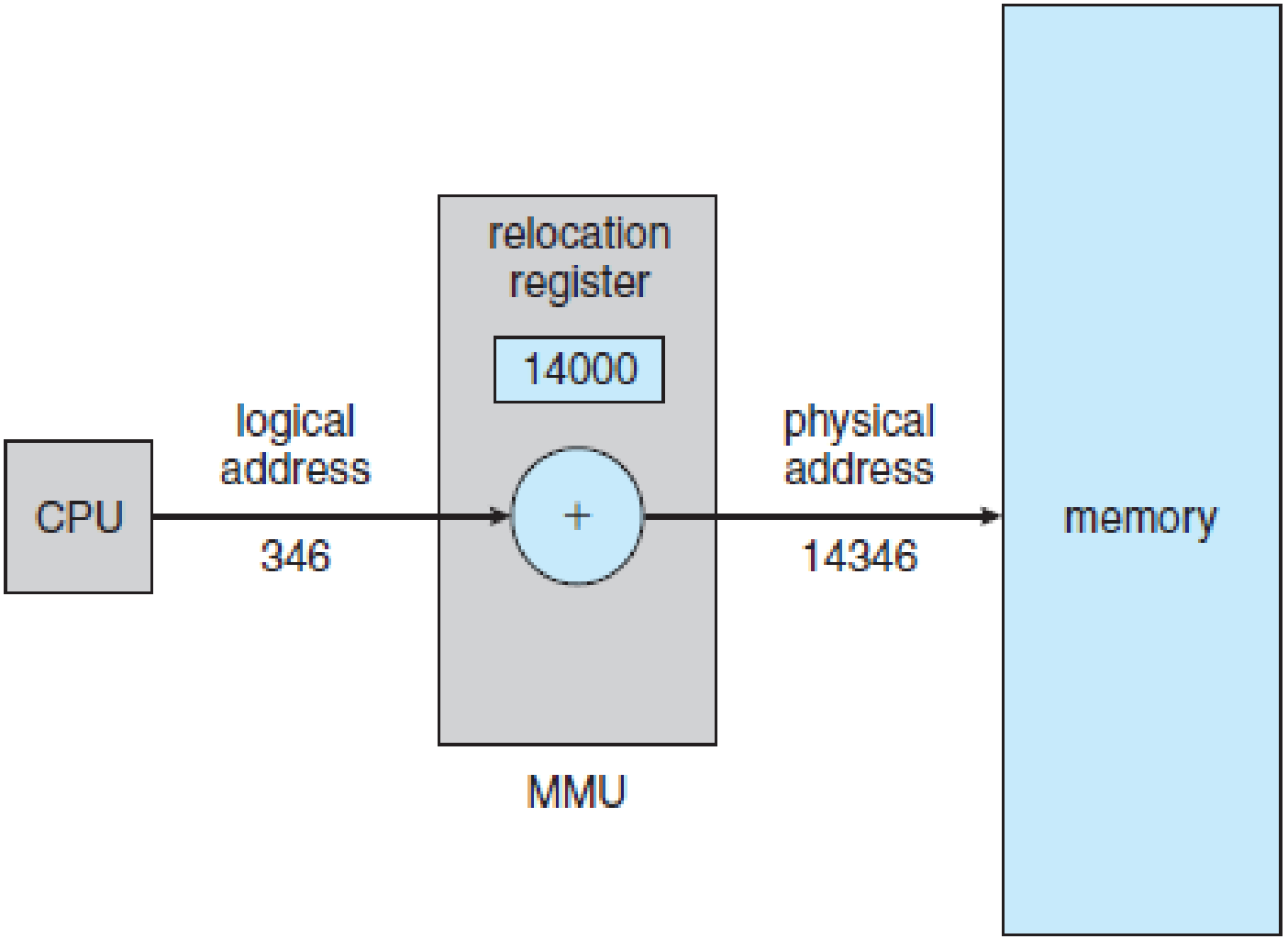

Figure 8.4 Dynamic relocation using a relocation register.

Dynamic Loading

The size of a process is limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading.**

With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change.

Contiguous Memory Allocation

In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.

One of the simplest methods for allocating memory is to divide memory into several **fixed-sized partitions**.

Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

Another allocation method is the **variable-partition scheme.** OS keeps a table indicating which parts of memory are available and which are occupied.

Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**. Eventually, memory contains a set of holes of various sizes.

The problem of Contiguous Memory Allocation: Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces.

Fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes.

This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes.

If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Segmentation

Segmentation is a memory-management scheme that permits the physical address space of a process to be noncontiguous, it is a collection of segments.

Each segment has a name (number) and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities: a segment name and an offset.

Thus, a logical address consists of a *two* tuple:

<*segment-number, offset*>

This mapping is effected by a **segment table.** Each entry in the segment table has a segment base and a segment limit.

The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.

The use of a segment table is illustrated in Figure 8.8. A logical address consists of two parts: a segment number, *s, and an offset into that segment, d.*

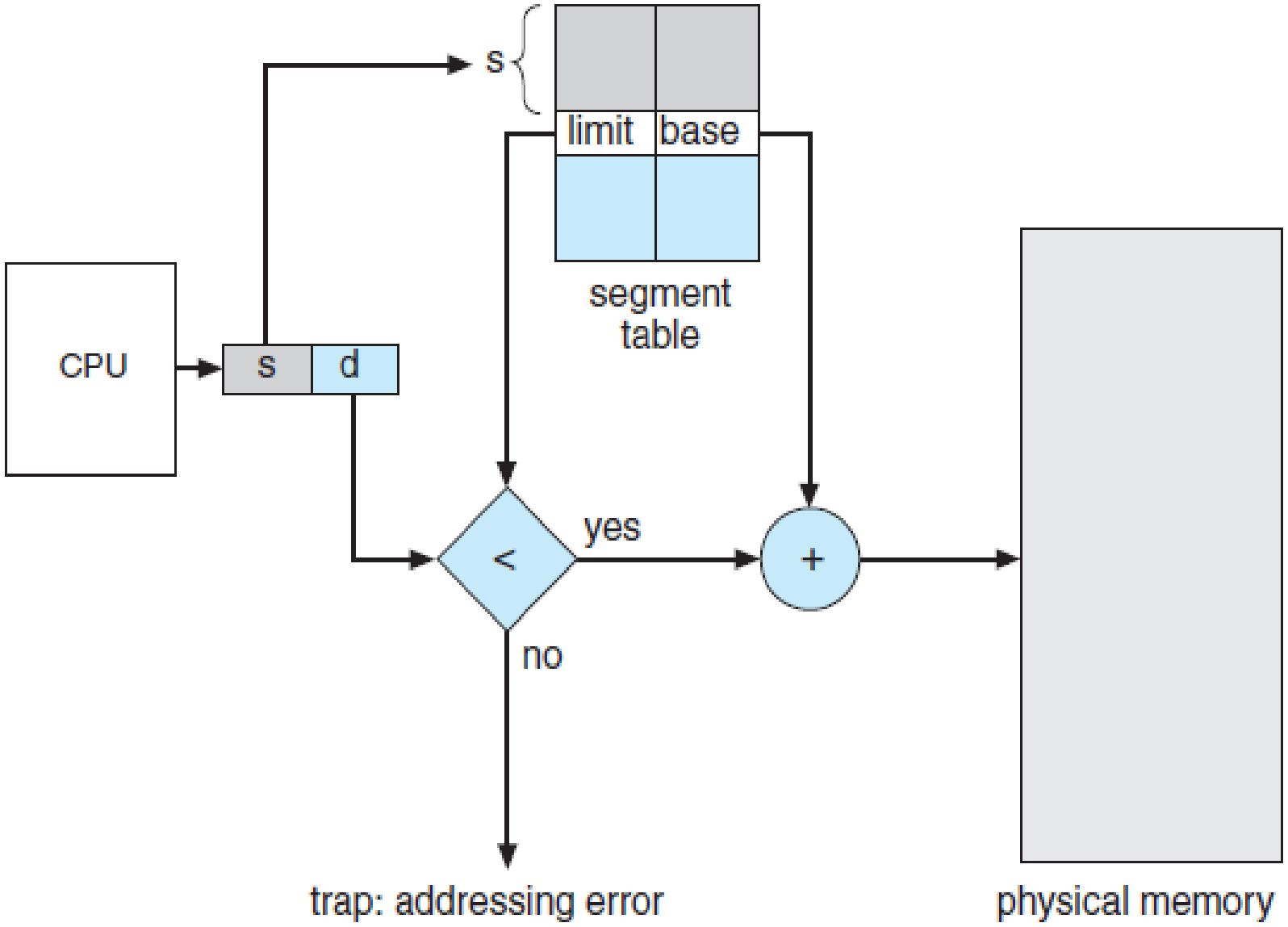

Figure 8.8 Segmentation hardware.

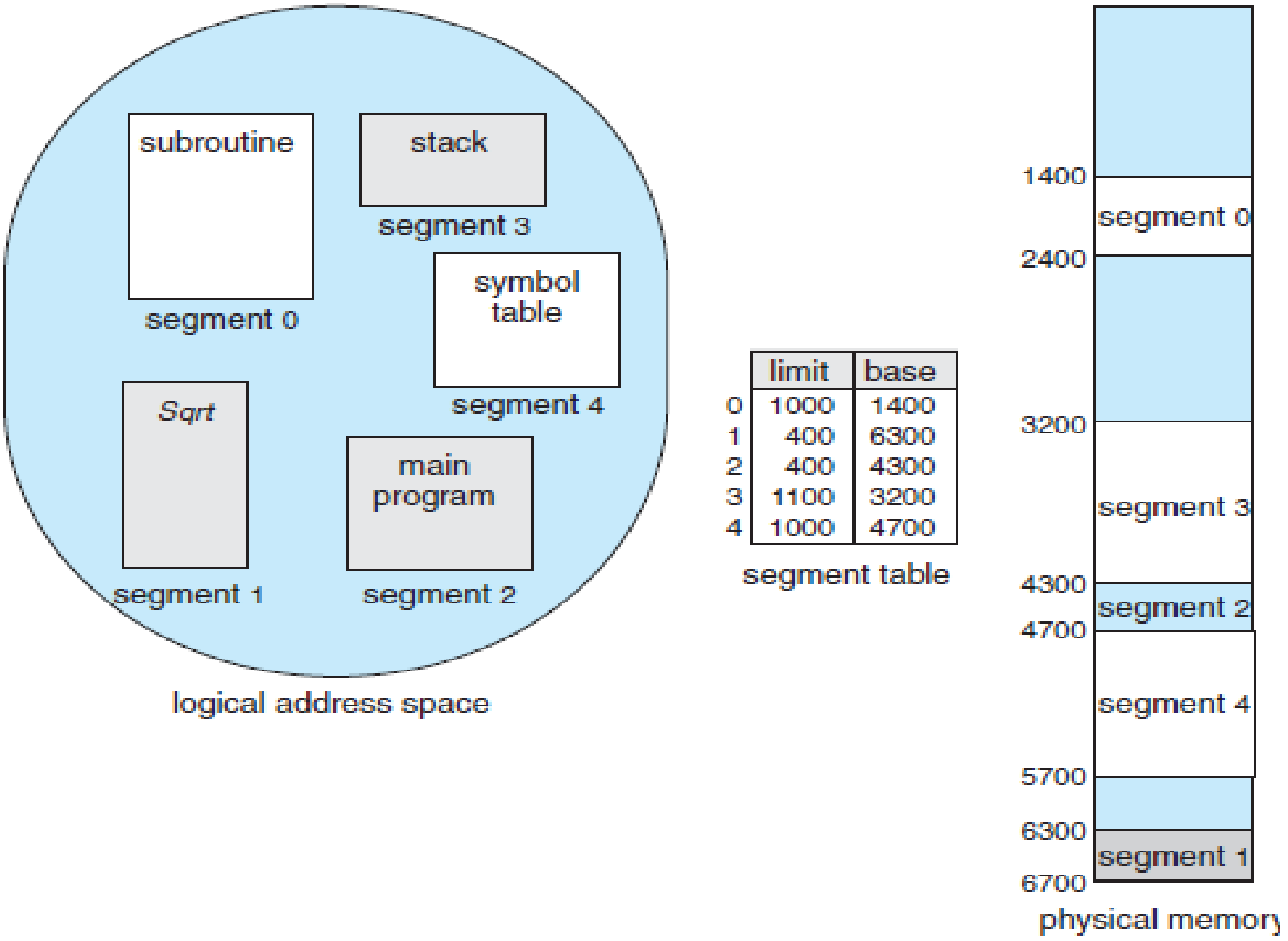

Figure 8.9 Example of segmentation.

Paging

Paging is another memory-management scheme. Paging avoids external fragmentation and the need for compaction, whereas segmentation does not.

It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store.

.

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**.

When a process is to be executed, its pages are loaded into any available memory frames from their source

The hardware support for paging is illustrated in Figure 8.10.

Every address generated by the CPU is divided into two parts: a **page number (p) and a page offset (d).**

The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure 8.11.

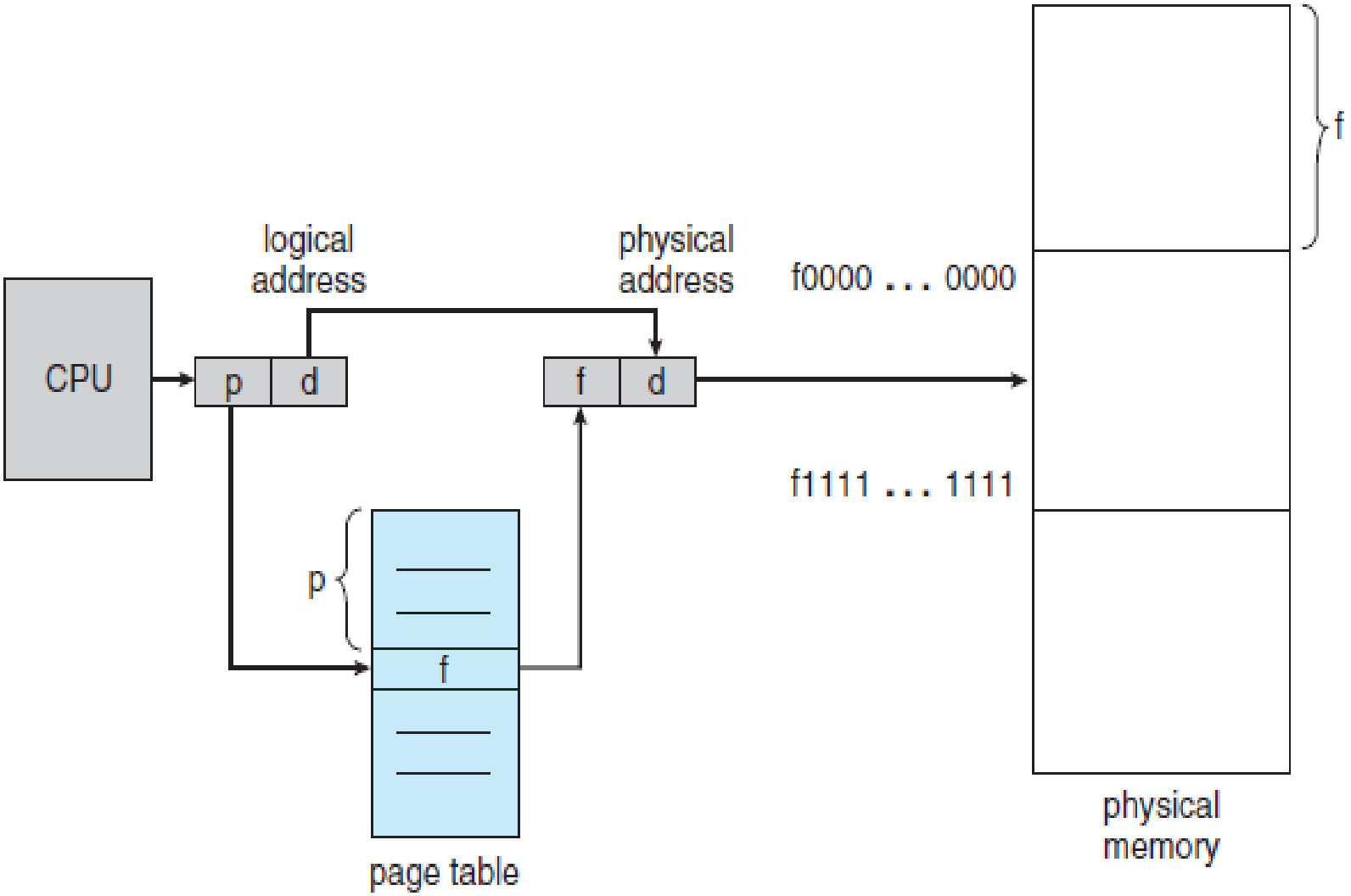

Figure 8.10 Paging hardware.

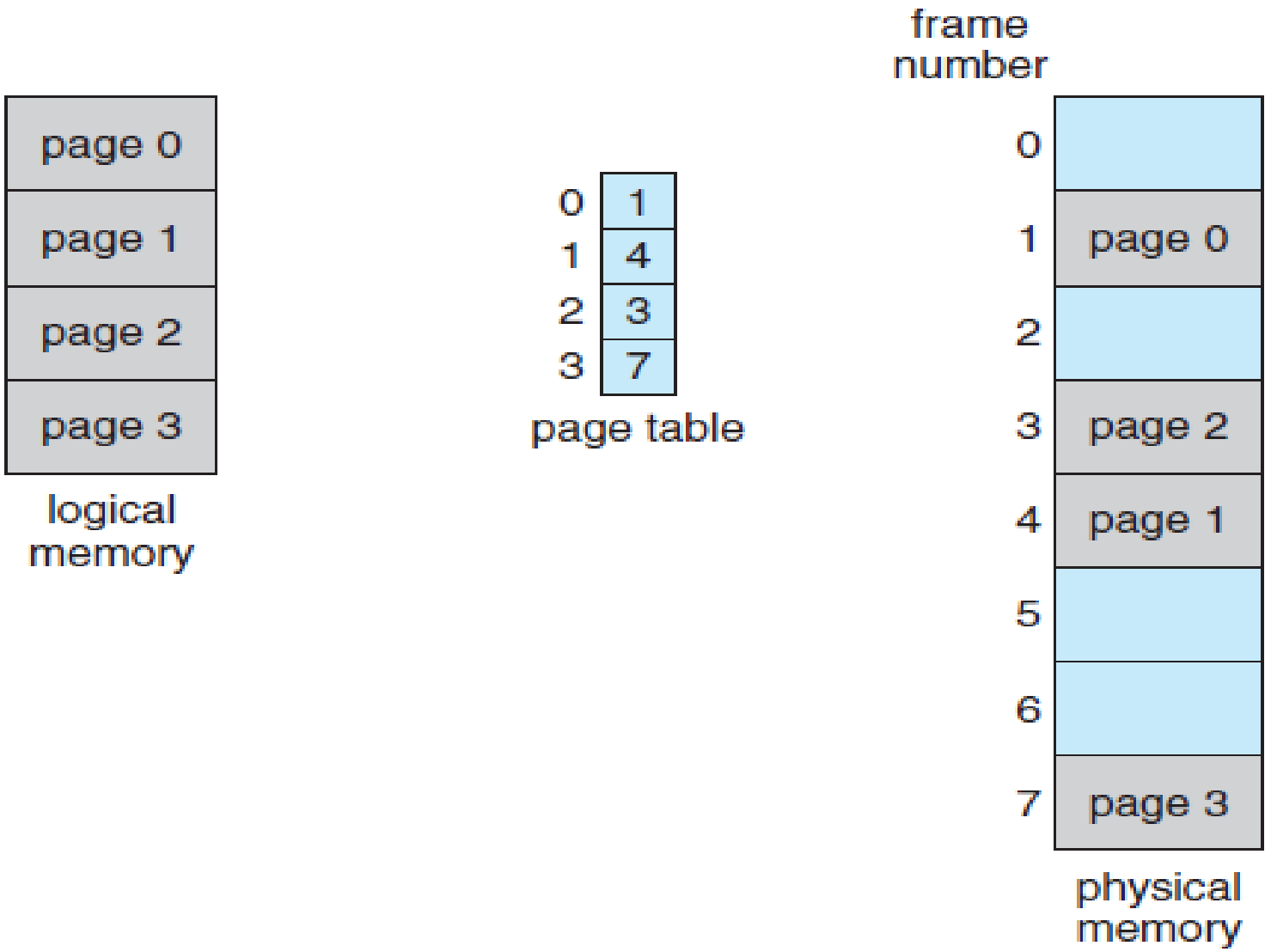

Figure 8.11 Paging model of logical and physical memory.

Virtual Memory

A virtual memory system attempts to optimize the use of the main memory (the higher speed portion) with the hard disk (the lower speed portion). In effect, virtual memory is a technique for using the secondary storage to extend the apparent limited size of the physical memory beyond its actual physical size.

It is usually the case that the available physical memory space will not be enough to host all the parts of a given active program. Those parts of the program that are currently active are brought to the main memory while those parts that are not active will be stored on the magnetic disk.

If the segment of the program containing the word requested by the processor is not in the main memory at the time of the request, then such segment will have to be brought from the disk to the main memory.

Movement of data between the disk and the main memory takes the form of **pages**. A page is a collection of memory words, which can be moved from the disk to the RAM when the processor requests accessing a word on that page.

A typical size of a page in modern computers ranges from 2K to 16K bytes.

A **page fault** occurs when the page containing the word required by the processor does not exist in the RAM and has to be brought from the disk.

Information about the main memory locations and the corresponding virtual pages are kept in a table called the **page table**. The page table is stored in the main memory.

Other information kept in the page table includes a bit indicating the validity of a page, modification of a page, and the authority for accessing a page.

The **valid bit** is set if the corresponding page is actually loaded into the main memory. Valid bits for all pages are reset when the computer is first powered on.

The other control bit that is kept in the page table is the **dirty bit**. It is set if the corresponding page has been altered while residing in the main memory. If while residing in the main memory a given page has not been altered, then its dirty bit will be reset. This can help in deciding whether to write the contents of a page back into the disk (at the time of replacement) or just to override its contents with another page.

Replacement Algorithms (Policies)

When a process needs a nonresident page, the operating system must decide which resident page is to be replaced by the requested page. The technique used in the virtual memory that makes this decision is called the **replacement policy**.

There exists a number of possible replacement mechanisms:

- Random Replacement According to this replacement policy, a page is selected randomly for replacement.
- First-In-First-Out (FIFO) Replacement According to this replacement policy, the page that was loaded before all the others in the main memory is selected for replacement.
- Least Recently Used (LRU) Replacement According to this technique, page replacement is based on the pattern of usage of a given page residing in the main memory regardless of the time spent in the main memory. The page that has not been referenced for the longest time while residing in the main memory is selected for replacement.