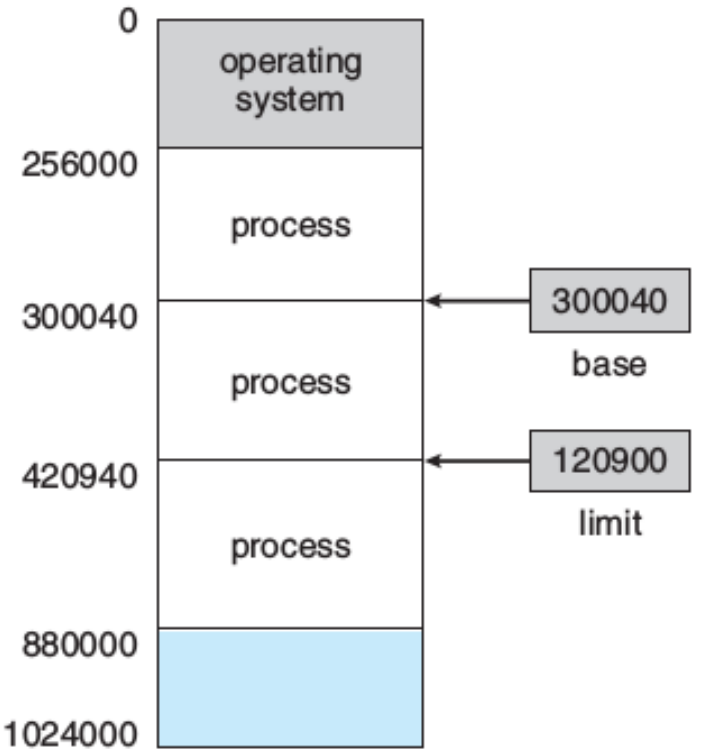
Main Memory

- Memory consists of a large array of bytes, each with its own address.
- The CPU fetches instructions from memory according to the value of the program counter.
- These instructions may cause additional loading from and storing to specific memory addresses.
- **Instruction-execution cycle**, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory.
- Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly.
- Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.
- If the data are not in memory, they must be moved there before the CPU can operate on them

- Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.
- To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
- We can provide this protection by using two registers, usually a base and a limit.
- The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range.
- For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).



- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the base and limit registers.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error.
- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction.
- Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers.

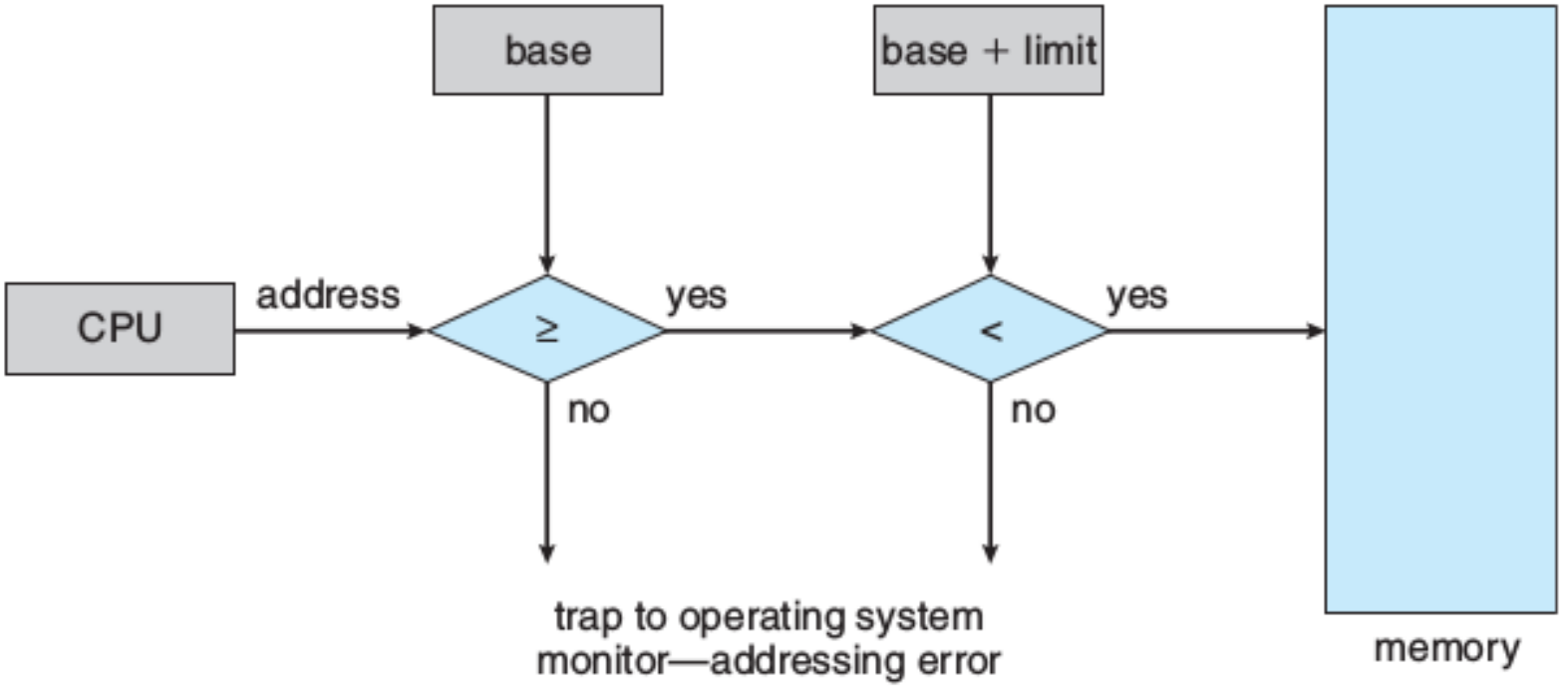


Figure 8.2 Hardware address protection with base and limit registers.

- The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory.
- This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services.

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 **job or input queue**.
- The normal single-tasking procedure is to select one of the processes in the input queue and to load that process into memory.
- As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.

- Addresses may be represented in different ways. Addresses in the source program are generally symbolic (such as the variable count).
- A compiler typically binds these symbolic addresses to relocatable addresses (such as “14 bytes from the beginning of this module”).
- 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.
- **Compile time Binding.** If you know at compile time where the process will reside in memory, then absolute code can be generated.
- For example, if you know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there.
- If, at some later time, the starting location changes, then it will be necessary to recompile this code.

- **Load time Binding.** If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code.
- In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.
- **Execution time Binding.** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

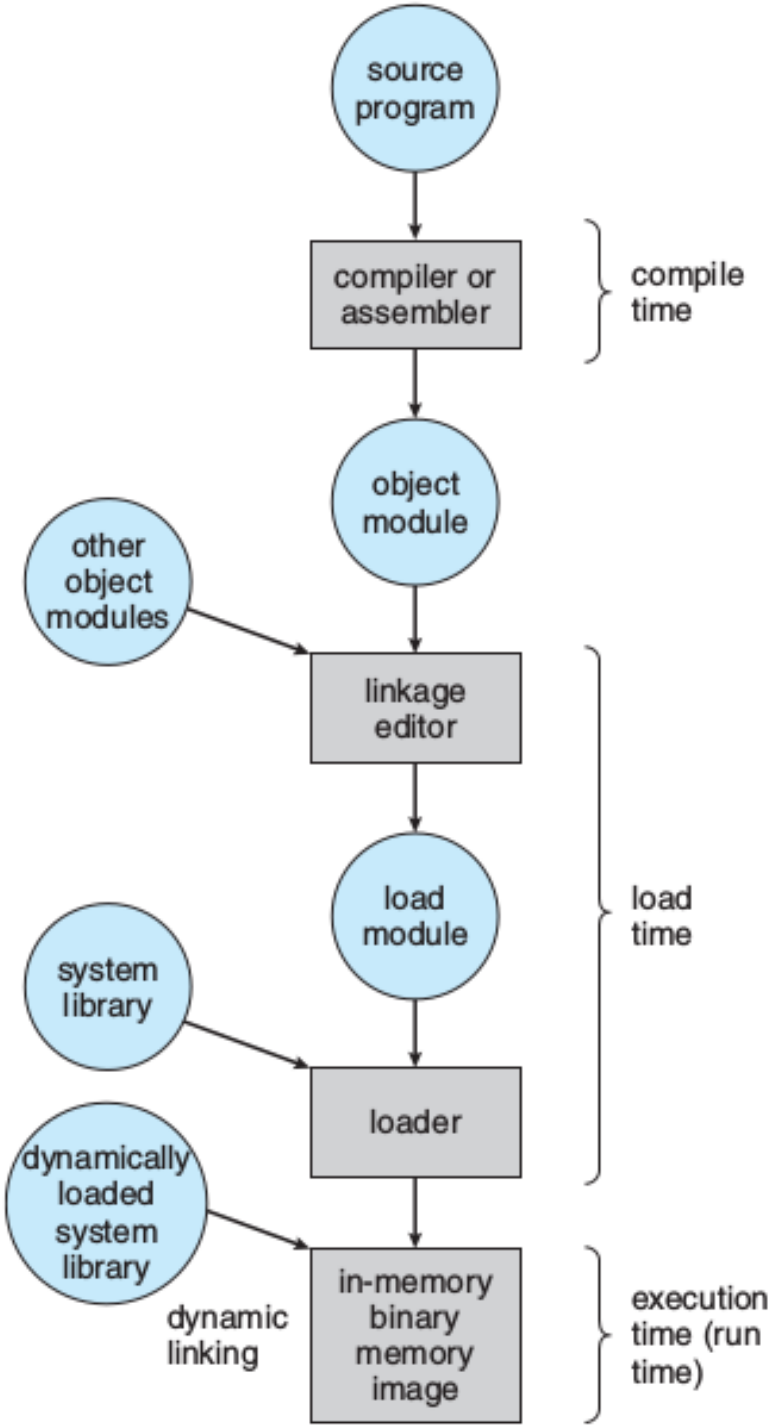


Figure 8.3 Multistep processing of a user program.

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 compile-time and load-time address-binding methods generate identical logical and physical addresses.
- However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**.
- The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**.

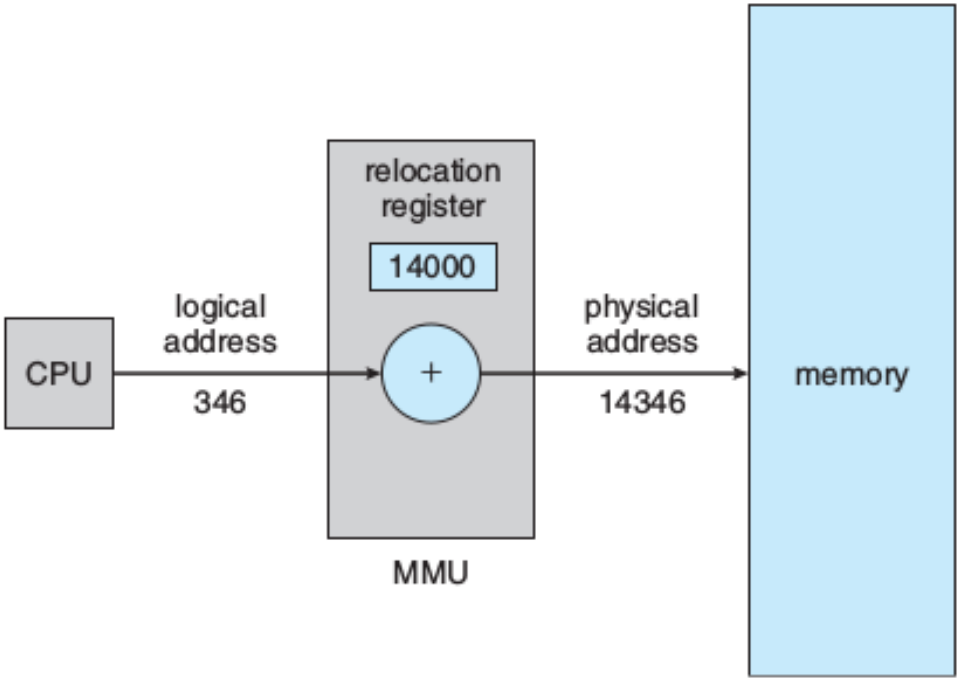


Figure 8.4 Dynamic relocation using a relocation register.

- 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 now 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.
- For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

- We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a base value R).
- The user program generates only logical addresses and thinks that the process runs in locations 0 to max.
- However, these logical addresses must be mapped to physical addresses before they are used.
- The memory-mapping hardware converts logical addresses into physical addresses.

Dynamic Loading

- It has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been 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. Then control is passed to the newly loaded routine.

- The advantage of dynamic loading is that a routine is loaded only when it is needed.
- This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.
- In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.
- Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method.

Dynamic Linking and Shared Libraries

- Dynamically linked libraries are system libraries that are linked to user programs when the programs are run.
- Some operating systems support only static linking, in which system libraries are treated like any other object module and are combined by the loader into the binary program image.
- Here, linking is postponed until execution time.
- This feature is usually used with system libraries, such as language subroutine libraries.
- Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image.
- This requirement wastes both disk space and main memory.

- With dynamic linking, a stub is included in the image for each library- routine reference.
- The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.
- When the stub is executed, it checks to see whether the needed routine is already in memory. If it is not, the program loads the routine into memory.
- Either way, the stub replaces itself with the address of the routine and executes the routine. Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.
- Under this scheme, all processes that use a language library execute only one copy of the library code.

- A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.
- Without dynamic linking, all such programs would need to be relinked to gain access to the new library.
- Version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use.
- Versions with minor changes retain the same version number, whereas versions with major changes increment the number.
- This system is also known as **shared libraries**.
- Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process’s memory space or that can allow multiple processes to access the same memory addresses.

Swapping

- A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
- Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.
- Standard swapping involves moving processes between main memory and a backing store. The backing store is commonly a fast disk.
- It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.

- The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.
- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory.
- If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.
- The context-switch time in such a swapping system is fairly high. Let the user process is 100 MB in size and the backing store is a standard hard disk with a transfer rate of 50 MB per second. The actual transfer of the 100- MB process to or from main memory takes

  100 MB /50 MB per second = 2 seconds

- The swap time is 2000 milliseconds. Since we must swap both out and in, the total swap time is about 4,000 milliseconds.
- The total transfer time is directly proportional to the amount of memory swapped. A 100-MB process could be swapped out in 2 seconds, compared with the 60 seconds required for swapping 3 GB .
- If we want to swap a process, we must be sure that it is completely idle. A process may be waiting for an I/O operation when we want to swap that process to free up memory.
- Assume that the I/O operation is queued because the device is busy. If we were to swap out process P1 and swap in process P2 , the I/O operation might then attempt to use memory that now belongs to process P2 .
- There are two main solutions to this problem: never swap a process with pending I/O , or execute I/O operations only into operating-system buffers.
- Transfers between operating-system buffers and process memory then occur only when the process is swapped in. This **double buffering** itself adds overhead. We now need to copy the data again, from kernel memory to user memory, before the user process can access it.

Contiguous Memory Allocation

- The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.
- We can place the operating system in either low memory or high memory. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.
- In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.
- The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600).
- Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register.

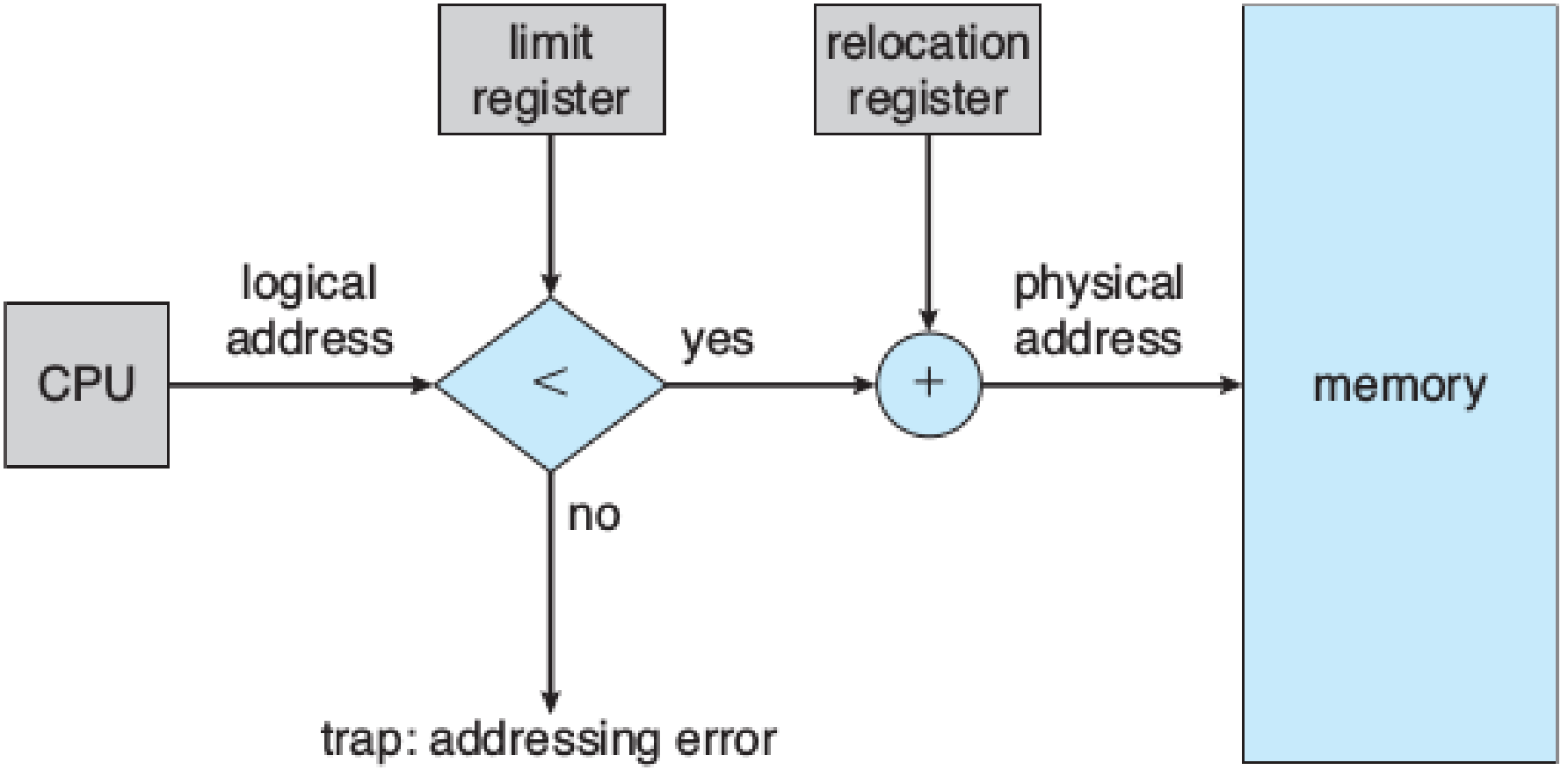


Figure 8.6 Hardware support for relocation and limit registers.

- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.
- Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.
- The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically.
- For example, the operating system contains code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes.
- Such code is sometimes called **transient operating-system code**; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution.

Memory Allocation

- Memory is divided 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.
- In the **variable-partition** scheme, the operating system 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.

- As processes enter the system, they are put into an input queue.
- The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.
- When a process is allocated space, it is loaded into memory, and it can then compete for CPU time.
- When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.
- Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied —that is, no available block of memory (or hole) is large enough to hold that process.
- The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

- The memory blocks available comprise a set of holes of various sizes scattered throughout memory.
- When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
- If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes.
- When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.
- At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

Dynamic storage-allocation problem

- It concerns how to satisfy a request of size n from a list of free holes. There are three solutions - first-fit, best-fit, and worst-fit.

  1. **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

  2. **Best fit**. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

  3. **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

- Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- External 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.

- Even with some optimization, given N allocated blocks, another 0.5 N blocks will be lost to fragmentation. This property is known as the **50-percent rule**.
- Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes.
- If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself.
- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.
- With this approach, the memory allocated to a process may be slightly larger than the requested memory.
- The difference between these two numbers is **internal fragmentation** —unused memory that is internal to a partition.

- One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time.
- Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available.
- Two complementary techniques achieve this solution: **segmentation** and **paging**.

Segmentation

- Segmentation is a memory-management scheme that supports programmer view of memory.
- When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on.
- The programmer talks about “the stack,” “the math library,” and “the main program” without caring what addresses in memory these elements occupy.
- A logical address space is a collection of segments.
- Each segment has a name 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**.

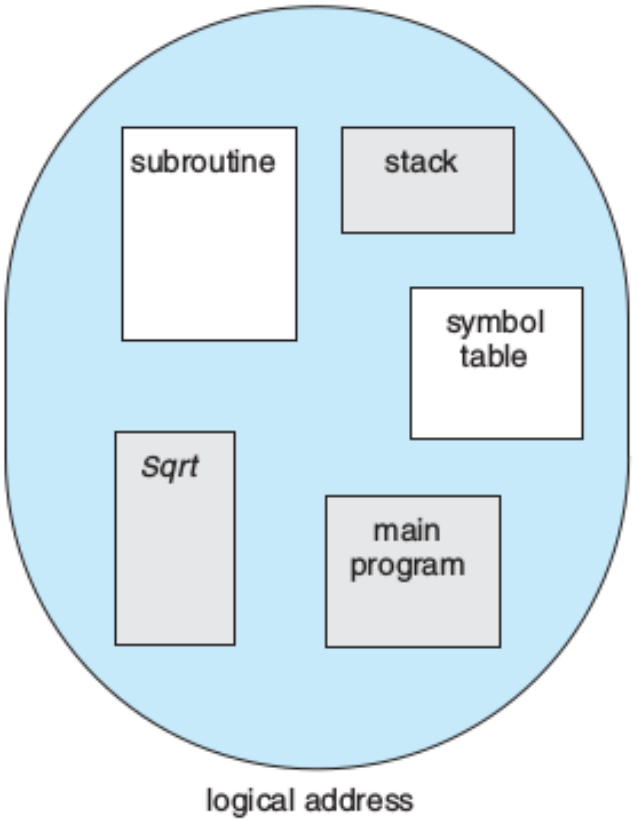


Figure 8.7 Programmer's view of a program.

- Segments are numbered and are referred to by a segment number, rather than by a segment name.
- Thus, a logical address consists of a two tuple:

  <segment-number, offset>.

- A C compiler might create separate segments for the following:
  1. The code
  2. Global variables
  3. The heap, from which memory is allocated
  4. The stacks used by each thread
  5. The standard C library
- Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.
- An implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses is done 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.

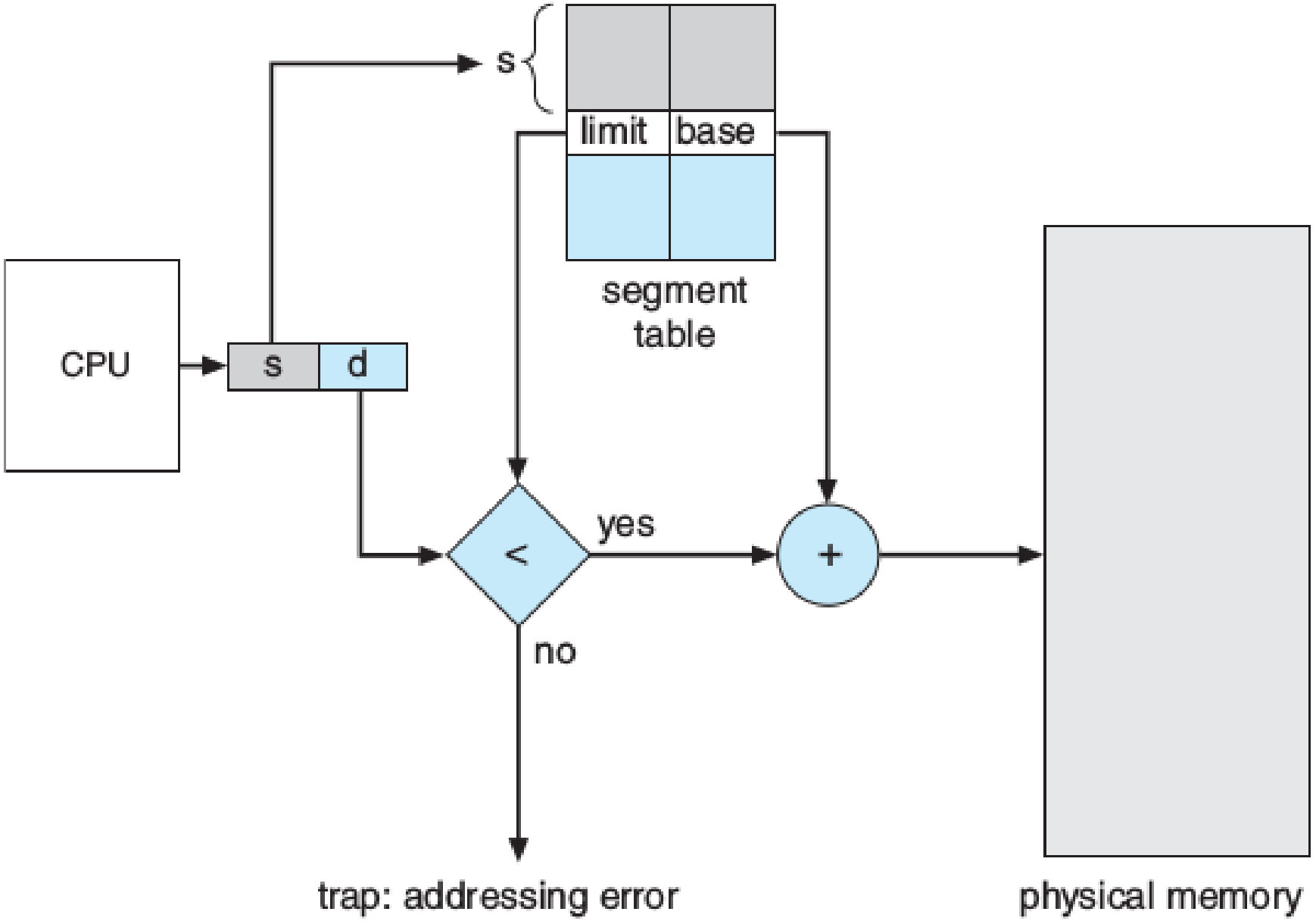
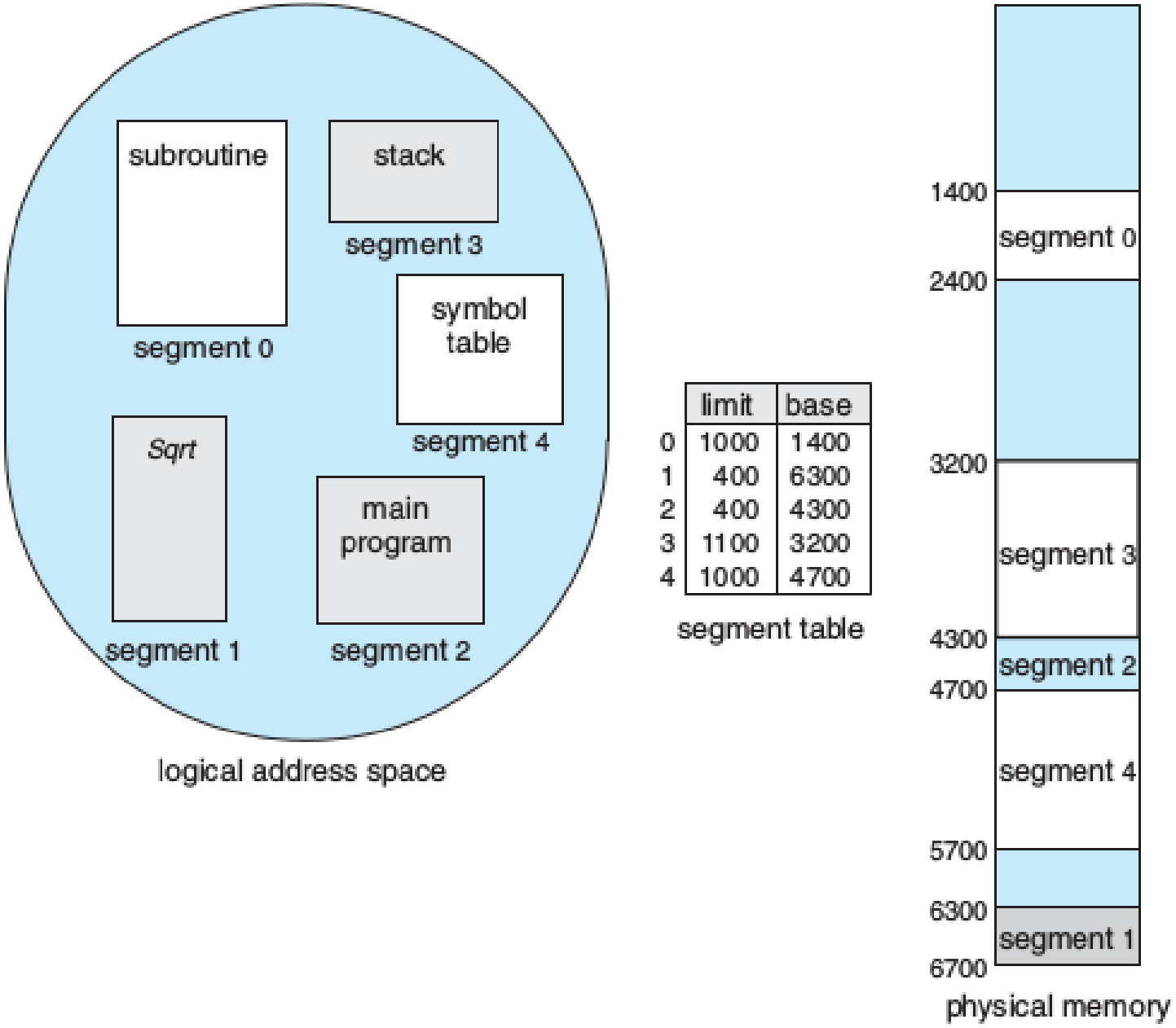


Figure 8.8 Segmentation hardware.

- A logical address consists of two parts: **a segment number, s, and an offset into that segment, d**.
- The segment number is used as an index to the segment table.
- The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.
- The segment table is thus essentially an array of base –limit register pairs.
- Example - We have five segments numbered from 0 through 4 stored in physical memory. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).
- Segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.



- A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

Paging

- Paging avoids external fragmentation and the need for compaction, whereas segmentation does not.
- In segmentation, when code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. This leads to the problem of fitting memory chunks of varying sizes onto the backing store.
- 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 (a file system or the backing store).
- The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.

- 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.

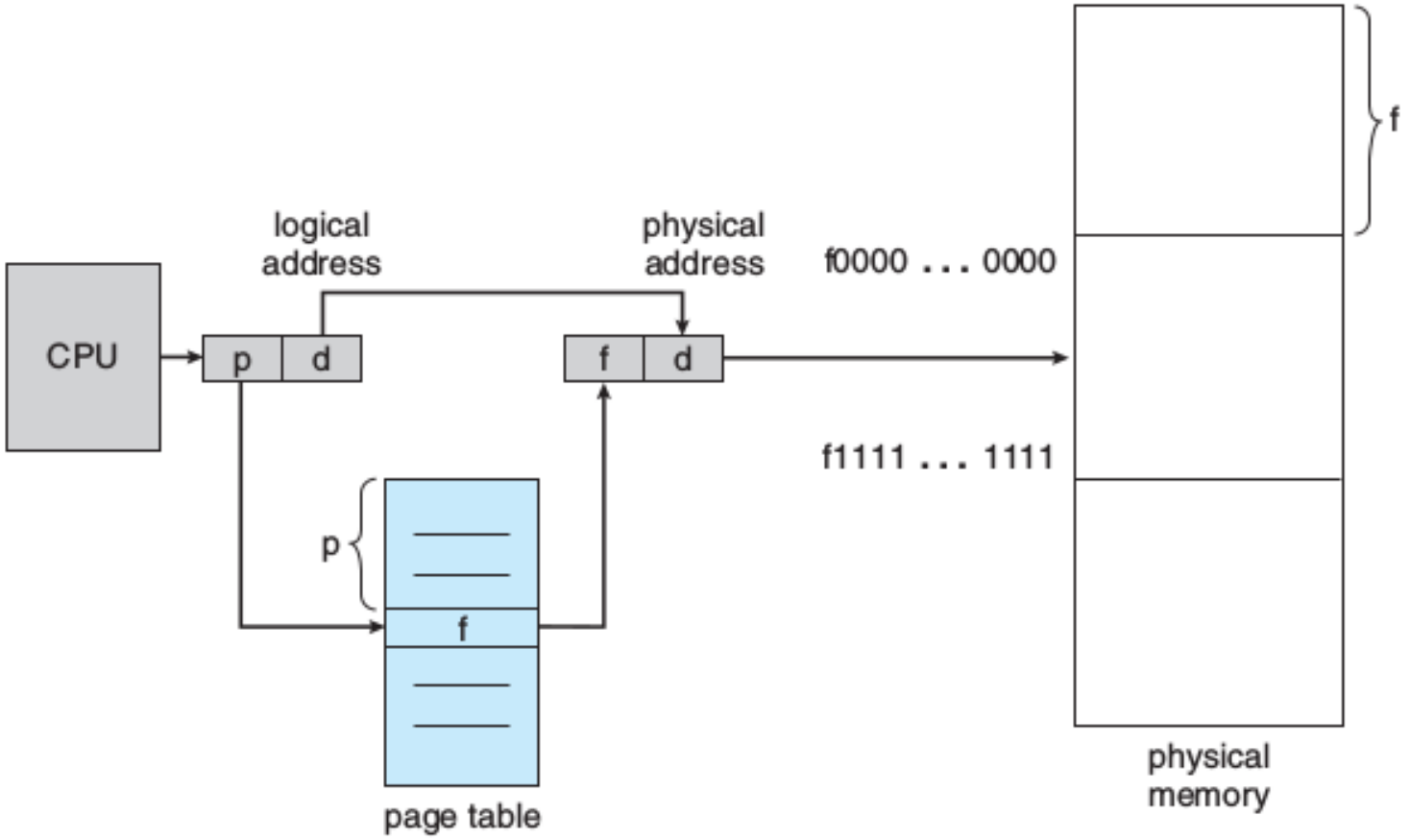


Figure 8.10 Paging hardware.

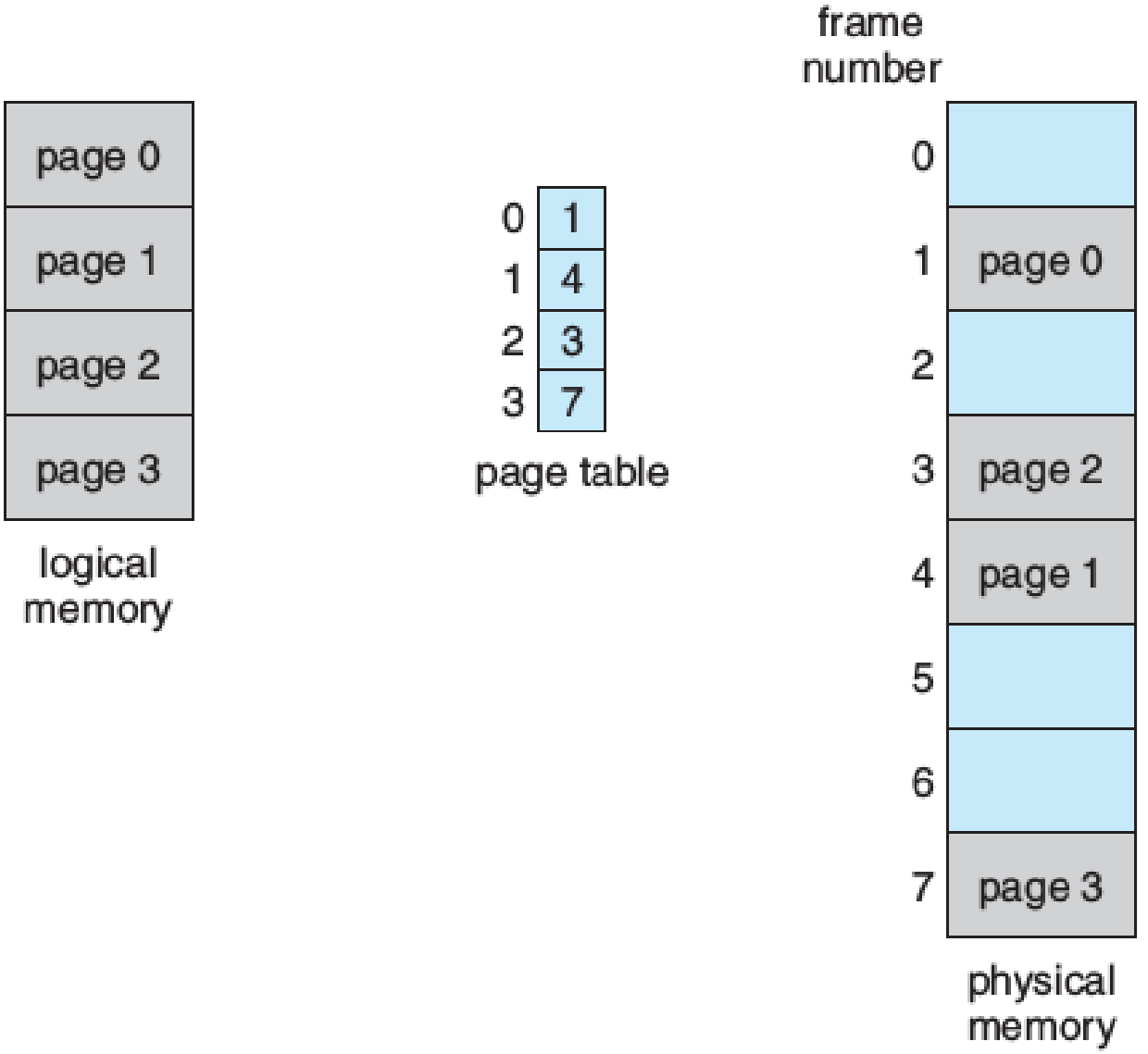
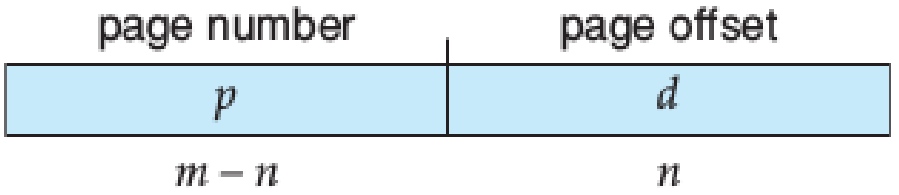


Figure 8.11 Paging model of logical and physical memory.

The size of a page is a power of 2, varying between 512 bytes and 1 GB per page, depending on the computer architecture.

- The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.
- If the size of the logical address space is 2^m , and a page size is 2^n bytes, then the high-order m − n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



where p is an index into the page table and d is the displacement within the page.

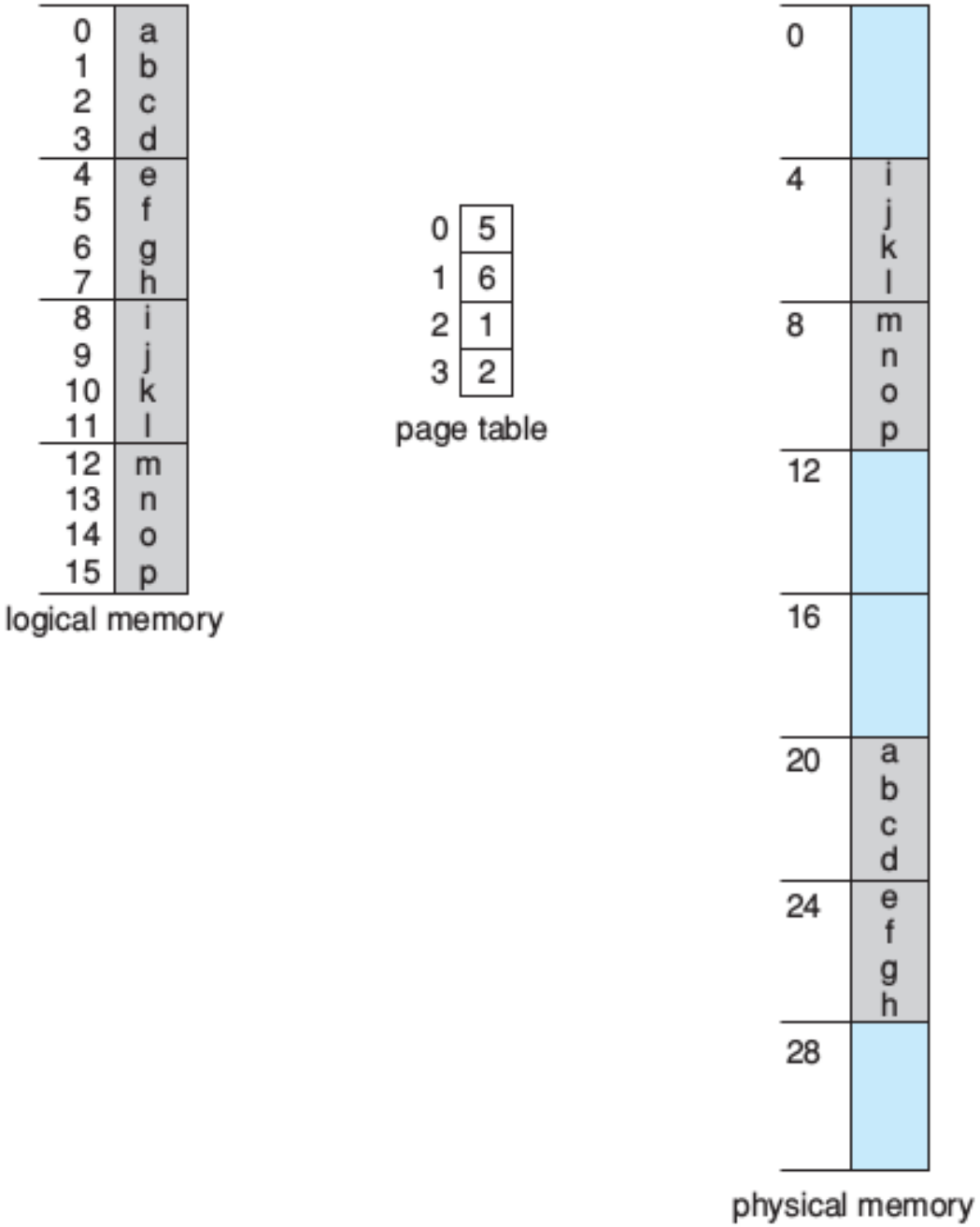


Figure 8.12 Paging example for a 32-byte memory with 4-byte pages.

- Consider the memory in Figure. Here, in the logical address, n= 2 and m = 4.
- Using a page size of $2^n = 4$ bytes and a physical memory of 32 bytes (8 pages),logical address 0 is page 0, offset 0.
- Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0].
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9.
- When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation.

- For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 − 1,086 = 962 bytes.
- In the worst case, a process would need n pages plus 1 byte. It would be allocated n + 1 frames, resulting in internal fragmentation of almost an entire frame.
- Therefore, small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases.
- Also, disk I/O is more efficient when the amount data being transferred is larger.
- When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process.

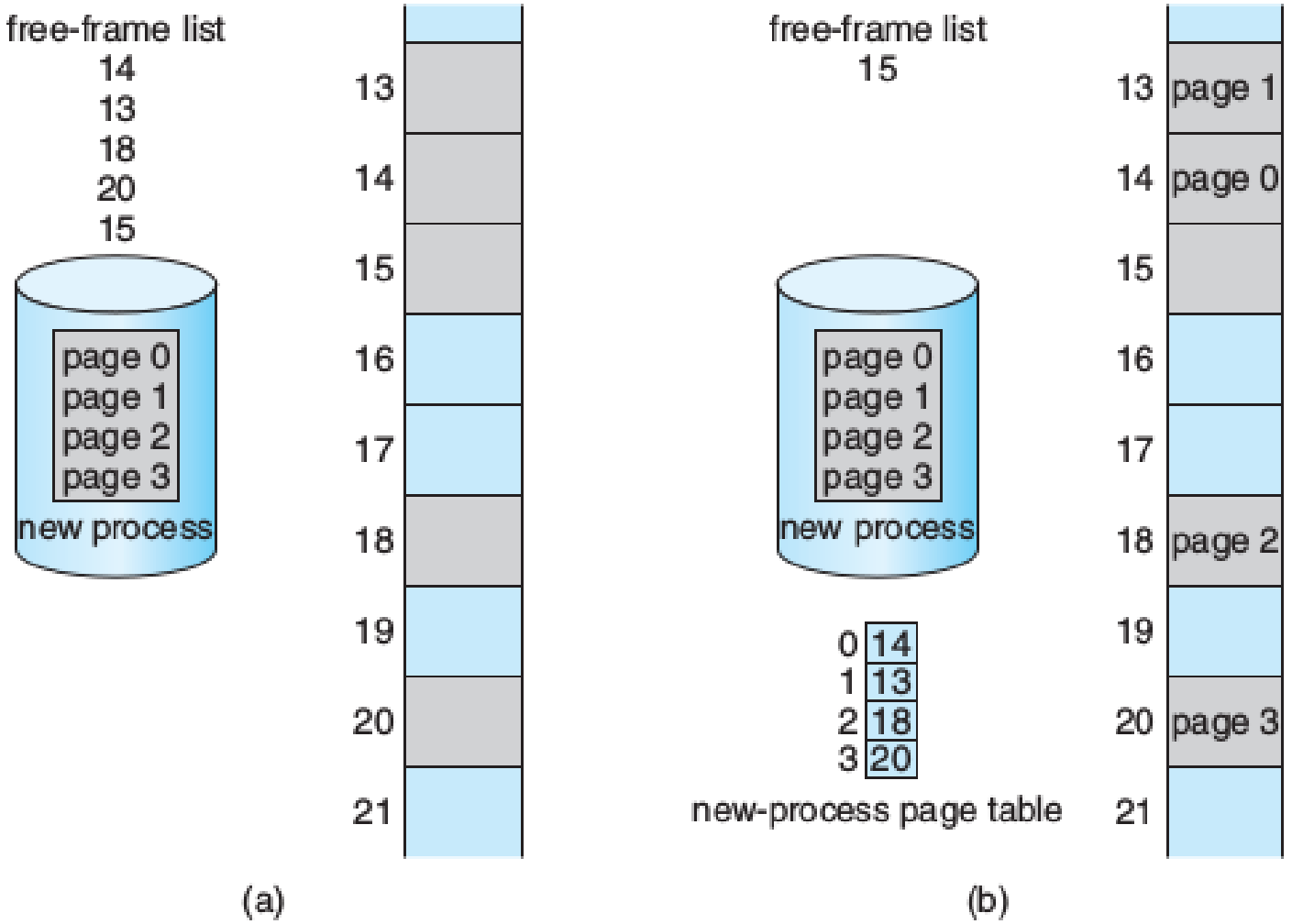


Figure 8.13 Free frames (a) before allocation and (b) after allocation.

- The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on.

- Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on.
- This information is generally kept in a data structure called a **frame table**. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.
- The operating system maintains a copy of the page table for each process. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually.
- Paging therefore increases the context-switch time.

Hardware Support

- In the simplest case, the page table is implemented as a set of dedicated registers. These registers should be built with very high-speed logic to make the paging-address translation efficient.
- The CPU dispatcher reloads these registers, just as it reloads the other registers.
- The use of registers for the page table is satisfactory if the page table is reasonably small.
- To allow the page table to be very large, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table.
- Changing page tables requires changing only this one register, substantially reducing context-switch time.

- The problem with this approach is the time required to access a user memory location.
- If we want to access location i, we must first index into the page table, using the value in the PTBR offset by the page number for i. This task requires a memory access.
- It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory.
- With this scheme, two memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2.

- The standard solution to this problem is to use a special, small, fast- lookup hardware cache called a **translation look-aside buffer (TLB)**.
- The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned.
- The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU , its page number is presented to the TLB .
- If the page number is found, its frame number is immediately available and is used to access memory.
- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory.

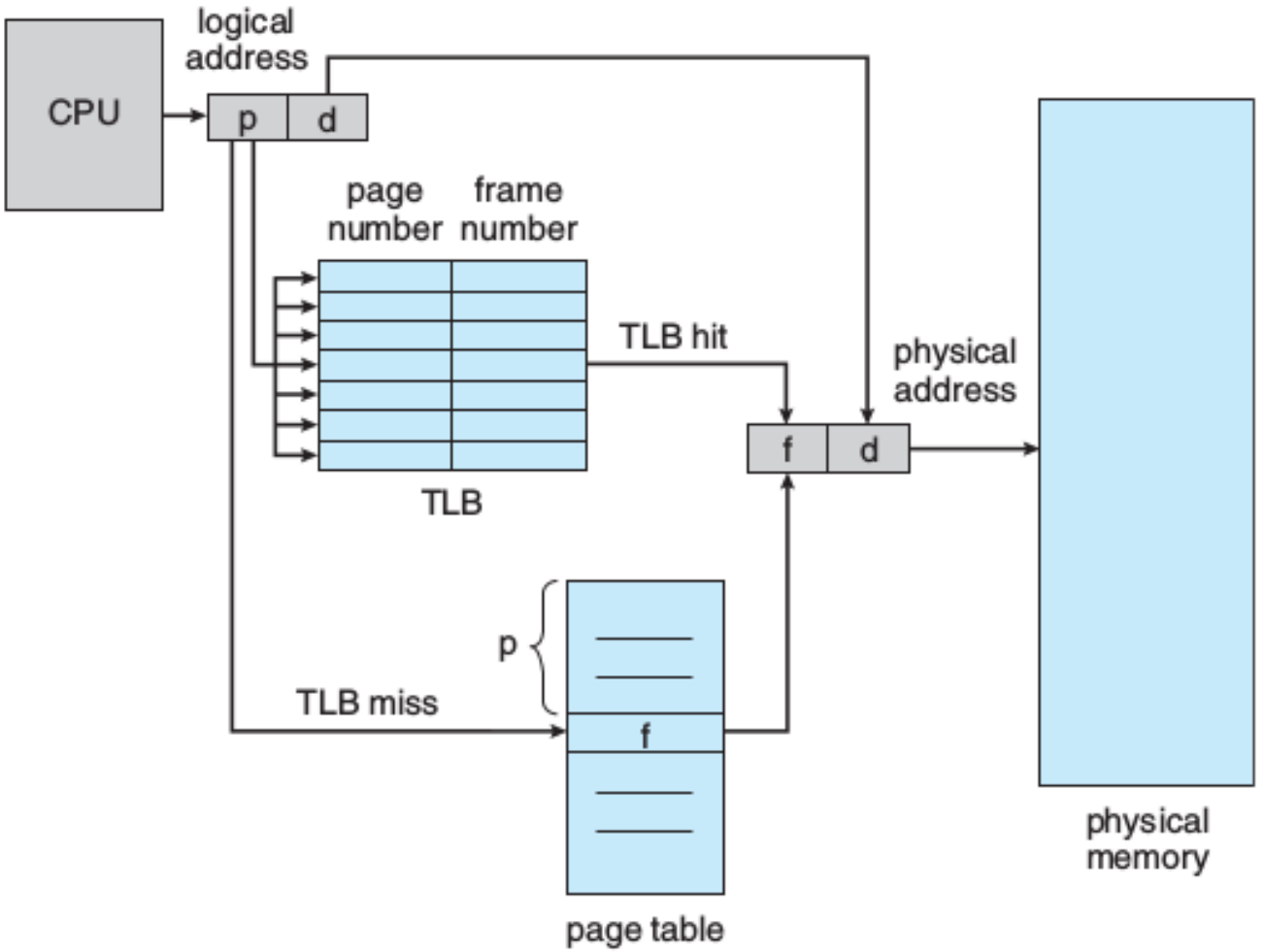


Figure 8.14 Paging hardware with TLB.

- In case of TLB miss, we add the page number and frame number to the TLB , so that they will be found quickly on the next reference.
- If the TLB is already full of entries, an existing entry must be selected for replacement.

- Some TLB s allow certain entries to be wired down, meaning that they cannot be removed from the TLB . Typically, TLB entries for key kernel code are wired down.
- Some TLB s store **address-space identifiers (ASID s)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process.
- When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page.
- If the ASID s do not match, the attempt is treated as a TLB miss.
- In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously.
- If the TLB does not support separate ASID s, then every time a new page table is selected (for instance, with each context switch), the TLB must be flushed (or erased) to ensure that the next executing process does not use the wrong translation information.
- Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

- The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time.
- If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB .
- If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds.
- To find the effective memory-access time, we weight the case by its probability:

  effective access time = 0.80 × 100 + 0.20 × 200 = 120 nanoseconds
- In this example, we suffer a 20-percent slowdown in average memory-access time (from 100 to 120 nanoseconds).

- For a 99-percent hit ratio, which is much more realistic, we have effective access time = 0.99 × 100 + 0.01 × 200 = 101 nanoseconds.
- This increased hit rate produces only a 1 percent slowdown in access time.

Memory Protection

- Memory protection in a paged environment is accomplished by protection bits associated with each frame. One bit can define a page to be read –write or read-only.
- Every reference to memory goes through the page table to find the correct frame number.
- At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.
- An attempt to write to a read-only page causes a hardware trap to the operating system
- One additional bit is generally attached to each entry in the page table: a **valid –invalid bit**.
- When this bit is set to valid, the associated page is in the process's logical address space and is thus a legal (or valid) page.
- When the bit is set to invalid, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid –invalid bit.

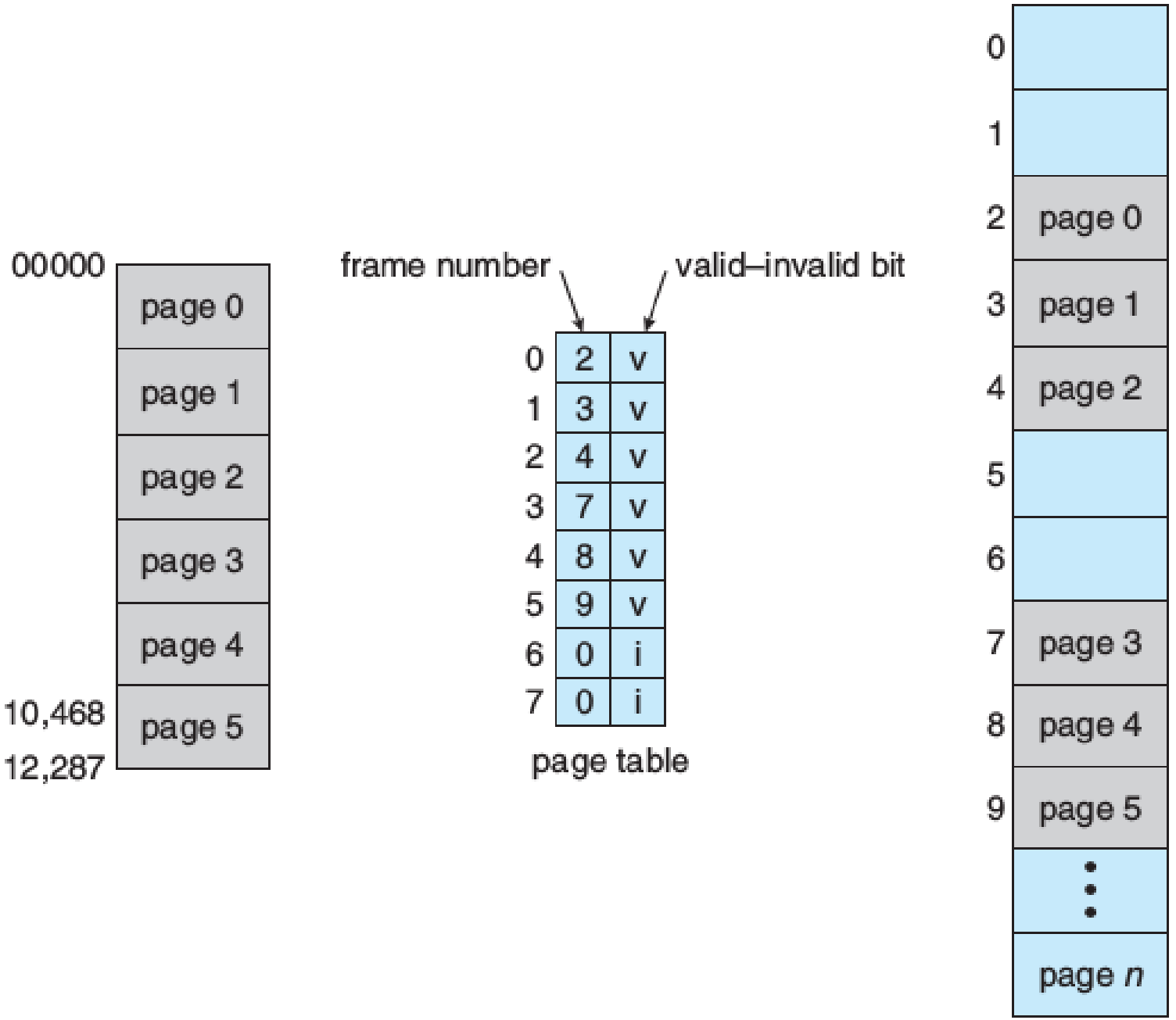


Figure 8.15 Valid (v) or invalid (i) bit in a page table.

- Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB.
- Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the valid –invalid bit is set to invalid, and the computer will trap to the operating system.
- This scheme has created a problem. Because the program extends only to address 10468, any reference beyond that address is illegal.
- However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid.
- This problem is a result of the 2- KB page size and reflects the internal fragmentation of paging.

Shared Pages

- An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment.
- Consider a system that supports 40 users, each of whom executes a text editor.
- If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is reentrant code (or pure code), however, it can be shared.
- Each process has its own data page.
- **Reentrant code** is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time.

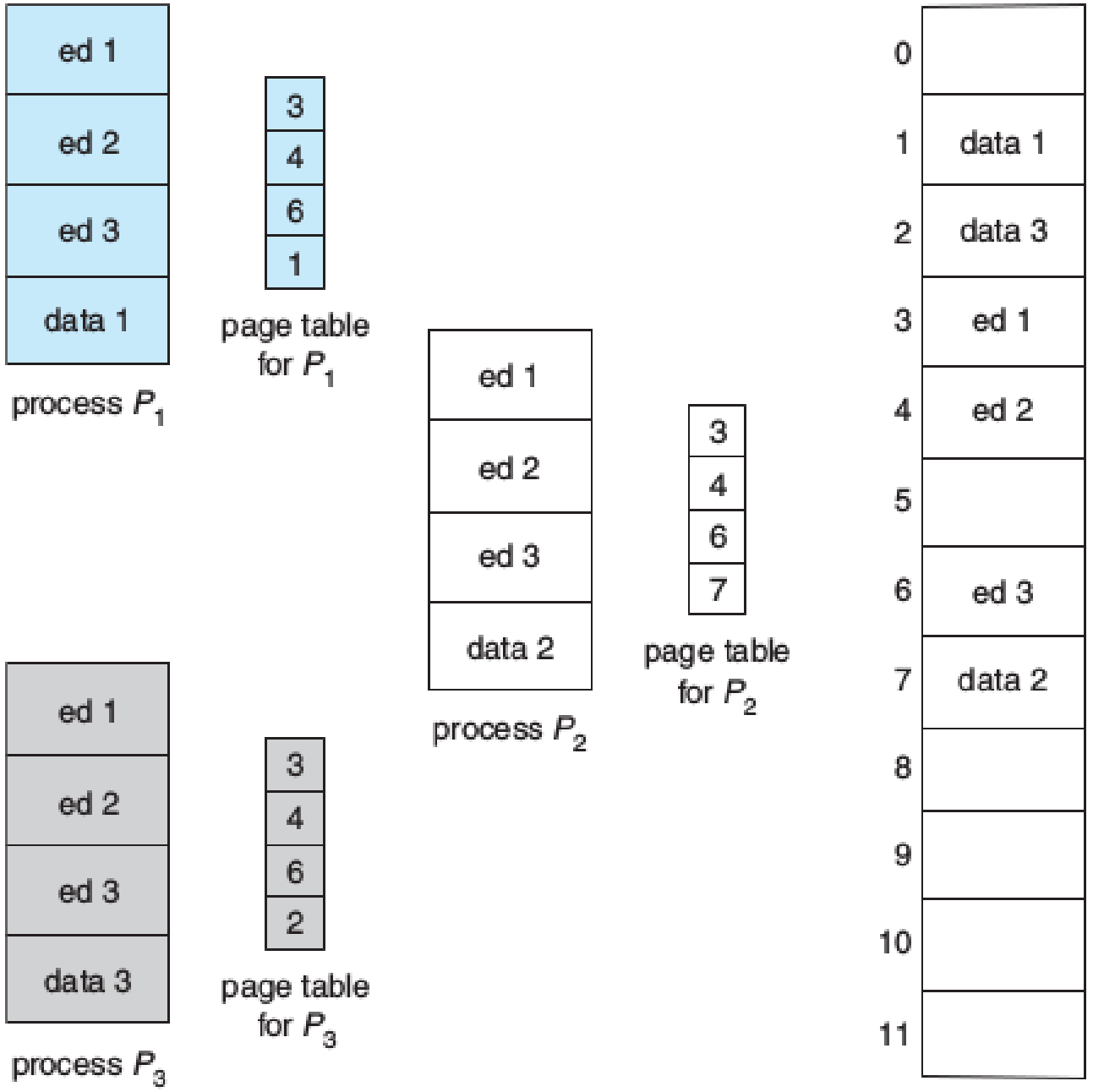


Figure 8.16 Sharing of code in a paging environment.

- Each process has its own copy of registers and data storage to hold the data for the process's execution.
- Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.
- Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user.
- The total space required is now 2,150 KB instead of 8,000 KB —a significant saving.
- Other heavily used programs can also be shared — compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant.