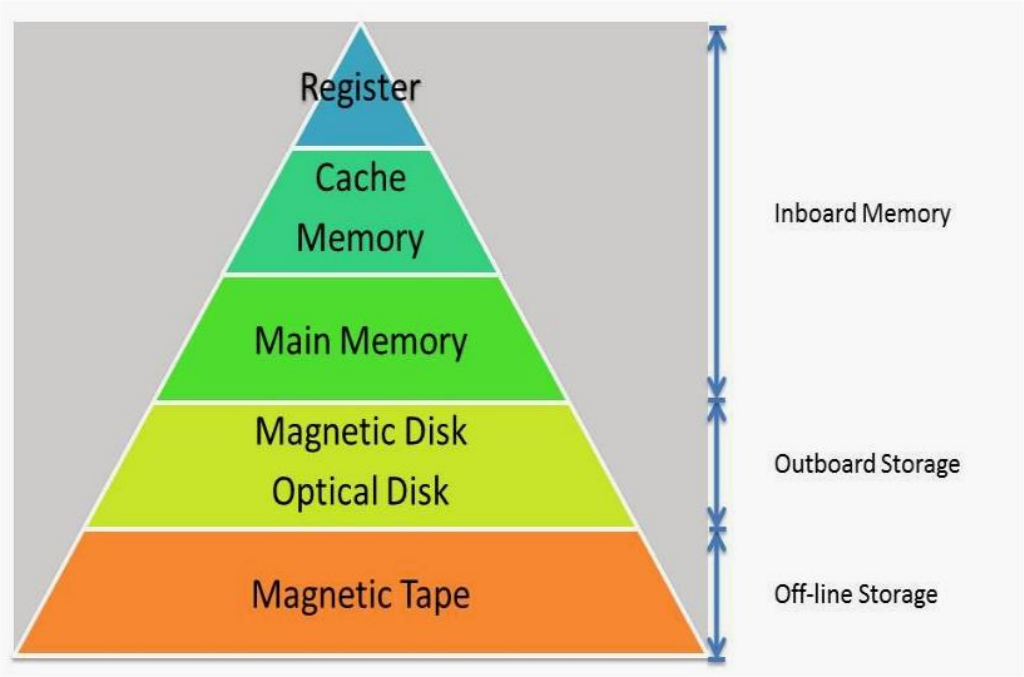
MODULE 4

MEMORY MANAGEMENT

INTRODUCTION

- CPU can load instructions only from memory, so any programs to run must be stored there.
- General-purpose computers run most of their programs from rewriteable memory, called **main memory** (also called Random Access Memory or **RAM**).
- Memory provides an array of words. Each word has its own **address**.(A numerical number)
- The **load** instruction moves a word from main memory to an internal register within the CPU, whereas the **store** instruction moves the content of a register to main memory.
- A typical **instruction-execution cycle** consist of 5 steps:
  1. First **fetches an instruction** from memory and stores that instruction in the instruction register (IR)
  2. The instruction is then **decoded**
  3. May cause **operands to be fetched** from memory and stored in some internal register.
  4. The instruction on the operands has been **executed**
  5. The result may be **stored back** in memory.
- The wide variety of storage systems in a computer system can be organized in a hierarchy according to speed and cost. The higher levels are expensive, but they are fast



BASIC HARDWARE

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.
- If the data are not in memory, they must be moved there before the CPU can operate on them.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. **CPUs can perform simple operations on register contents at the rate of one or more operations per clock tick**
- The same cannot be said of main memory, which is accessed via a transaction on the memory bus. **Completing a memory access may take many cycles of the CPU clock.**
- In such cases, the processor normally needs to **stall** (wait)

- The remedy is to add fast memory between the CPU and main memory. **A memory buffer used to accommodate a speed differential is called a cache.**
- We also must ensure to **protect** the OS from access by user processes and to protect user processes from one another.
- We first need to make sure that each process has a **separate memory space.**
- We can provide this protection by using two registers, a **base register** and a **limit register**.
- 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 3000 and the limit register is 1200, then the program can legally access all addresses from 3000 through 4199 (inclusive).
- Any attempt by a program executing in user mode to access OS memory or other users' memory results in a trap to the OS, 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 OS or other users.
- **Only the OS can load the base and limit registers.**

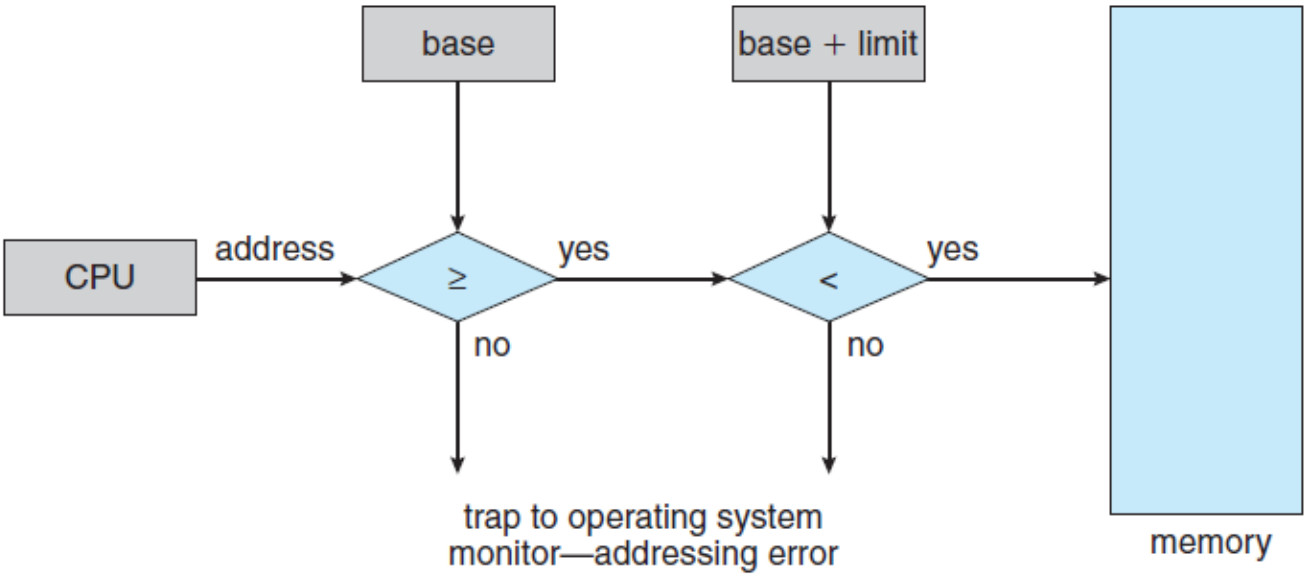


Figure 8.2 Hardware address protection with base and limit registers.

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.
- The processes on the disk that are waiting to be brought into memory for execution form the **input queue**
- The normal procedure is to select one of the processes in the input queue and to load that process into memory.
- A user program will go through several steps before being executed

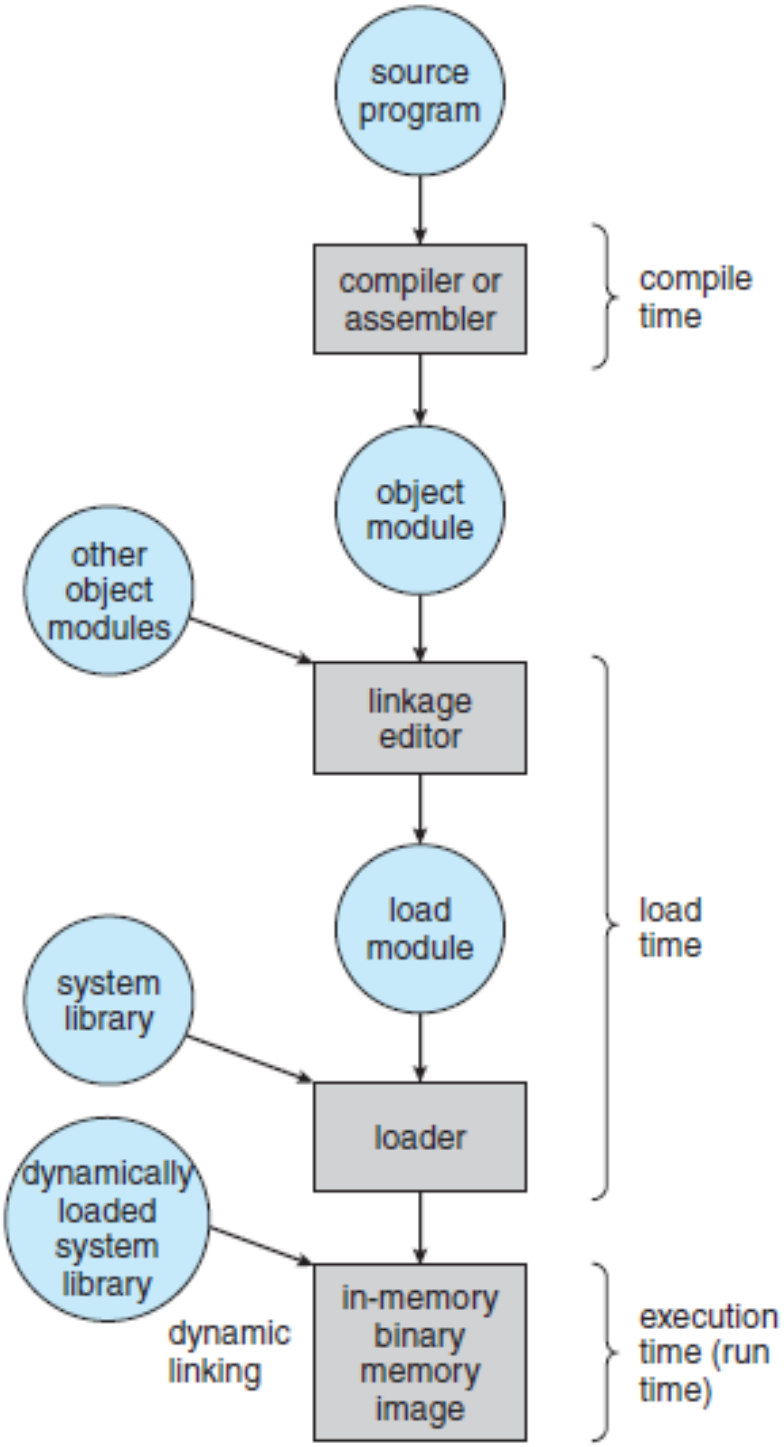


Figure 8.3 Multistep processing of a user program.

- Addresses may be represented in different ways during these steps.

- Addresses in the source program are generally symbolic (such as *count).* **It should bind with correct memory address**
- Binding can be done in 3 situations

1. **Compile time**. If we know at compile time where the process will reside in memory, then **absolute code** can be generated. If, at some later time, the starting location changes, then it will be necessary to recompile this code.
2. **Load time**. 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
3. **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. **Special hardware must be available for this scheme to work.** Most general-purpose OS use this method.

LOGICAL VERSUS PHYSICAL ADDRESS SPACE

- An address generated by the CPU is called **Logical address** whereas an address seen by the memory unit is called **Physical address**.
- Physical address is stored in a special register named **memory address register (MAR)**.

- 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
- Logical address is also called **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**.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the **Memory Management unit (MMU)**

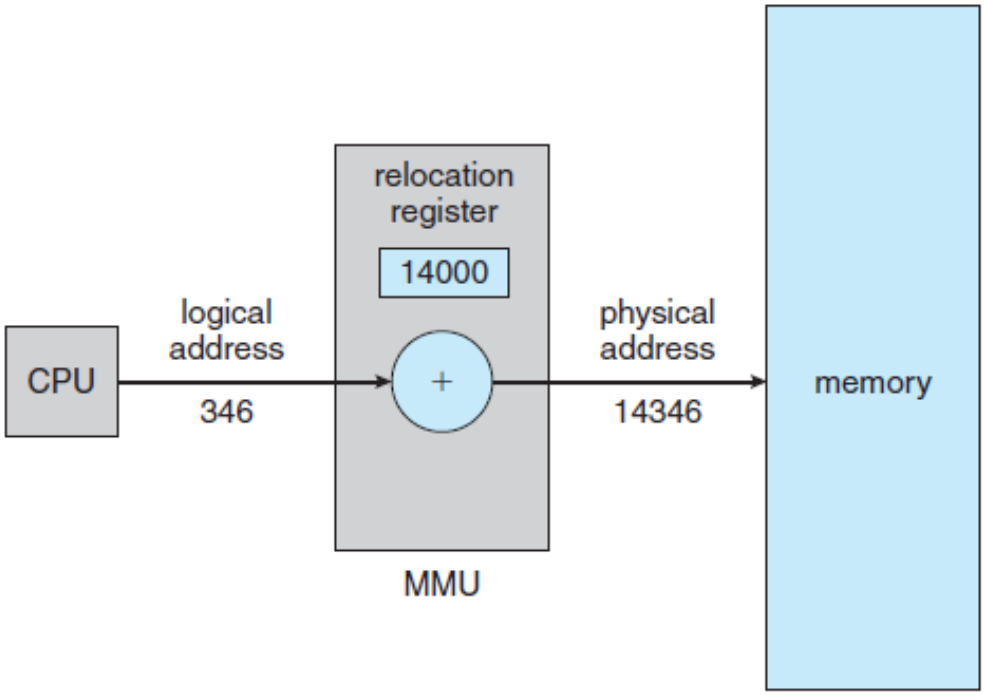


Figure 8.4 Dynamic relocation using a relocation register.

- 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
- 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.
- The **user program deals with logical** addresses. The memory-mapping hardware converts logical addresses into physical addresses.

DYNAMIC LOADING

- 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 an unused routine is never loaded.
- This method is particularly **useful when large amounts of code are needed to handle in infrequently occurring cases**
- **Dynamic loading does not require special support from the OS.** It is the responsibility of the users to design their programs

DYNAMIC LINKING AND SHARED LIBRARIES

- Some OS 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.**
- In dynamic linking, the linking is postponed until execution time. This feature is usually used with **system libraries**.
- Without this facility, each program on a system must include a copy of its 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.**
- 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.
- **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.
- Unlike dynamic loading, **dynamic linking generally requires help from the OS**.

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.
- This is called **swap in and swap out**
- **Swapping is needed in many situations, especially during context switch in multi-programming.**
- Example, assume round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to

swap out the process that just finished and to swap another process into the memory space that has been freed

- It is also used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called **roll out and roll in**

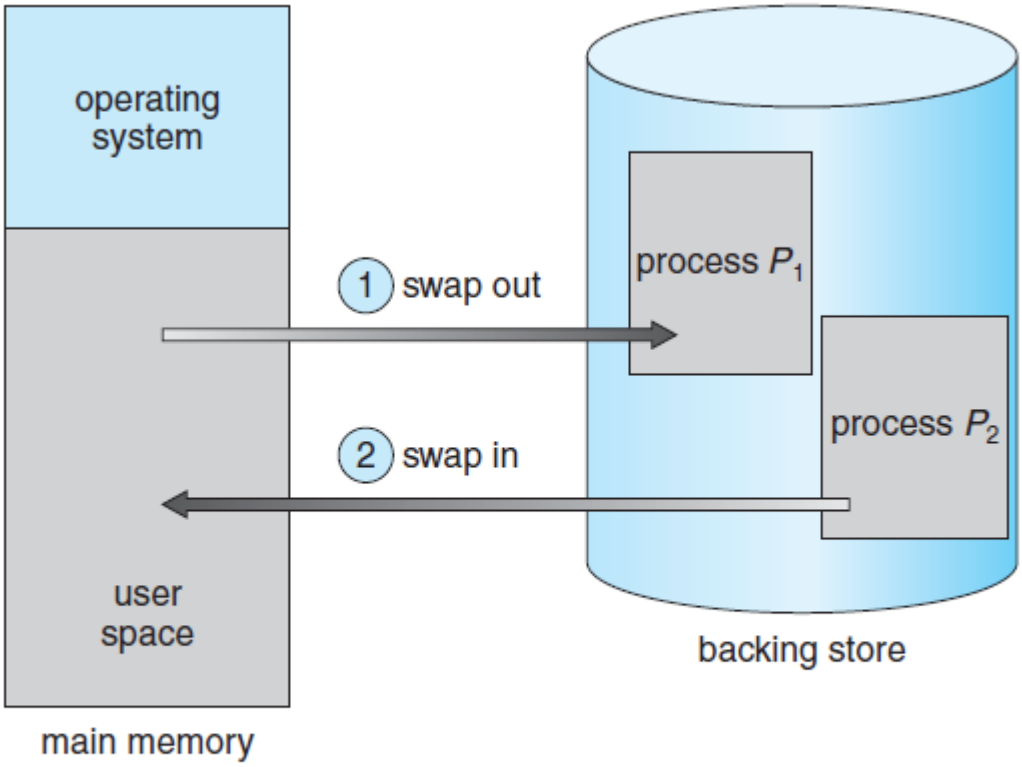


Figure 8.5 Swapping of two processes using a disk as a backing store.

- Normally, a process that is swapped out will be **swapped back into the same memory space** it occupied previously.

- If address binding is done at compile time or load time, then the process cannot be easily moved to a different location.
- **If execution-time binding is being used, however, then a process can be swapped into a different memory space**, because the physical addresses are computed during execution time.
- 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.**
- The major part of the swap time is transfer time between main memory and backing store. The total transfer time is directly proportional to the *amount* of memory swapped.
- It would be useful to know exactly how much memory a user process *is presently* using, not simply how much it *might be* using.
- Then we would need to swap only what is actually used, reducing swap time.
- For this method to be effective, the user must keep the system informed of any changes in memory requirements. Thus, a process with dynamic memory requirements will

need to issue system calls (request memory and release memory) to inform the OS of its changing memory needs.

- A process may be waiting for an I/0 operation when we want to swap that process to free up memory.
- There are **two solutions to this problem**:

1. Never swap a process with pending I/0
2. Execute I/0 operations only into OS buffers. Transfers between OS buffers and process memory occur only when the process is swapped in.

CONTIGUOUS MEMORY ALLOCATION

- The **memory is usually divided into two partitions**: one for the resident OS and one for the **user processes**.
- We can place the OS in either low memory or high memory. Programmers **usually place the OS in low memory**.
- We need to consider how to allocate available memory to the processes that are in the **input queue** waiting to be brought into memory.
- Each process is contained in a single contiguous section of memory.

Memory Mapping and Protection

- We can provide these features by using a **relocation register** together with a **limit register**

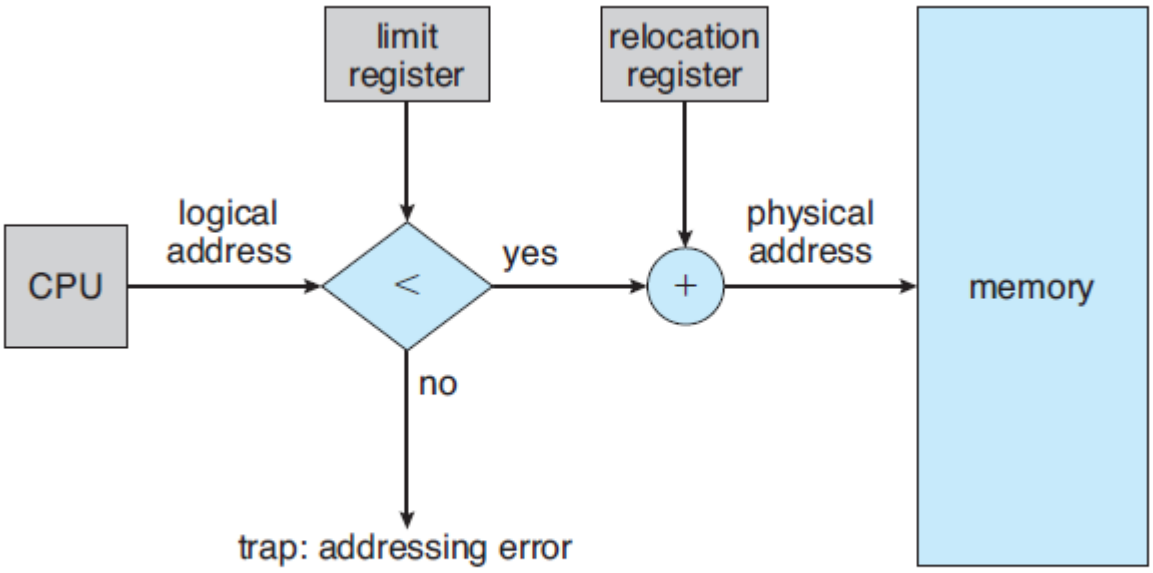


Figure 8.6 Hardware support for relocation and limit registers.

- Each logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory
- Because every address generated by a CPU is checked against these registers, we can protect both the OS 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 OS size to change dynamically.
- Eg: If a device driver 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 OS code**; it comes and goes as needed. So the size of OS may vary.

Memory Allocation

- Can be done in 2 ways

1. Fixed sized partitions
2. Variable sized partitions

- **Simplest method** 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.
- 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 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**.
- After allocating memory for a number of processes, a **set of holes** will be there.
- We have a list of available block sizes and an input queue. The OS can order the input queue according to a scheduling algorithm.
- **Memory is allocated to processes until 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 OS 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.**
- This is called **dynamic storage allocation problem** which concerns how to satisfy a request of size *n* from a list of free holes
- **3 strategies** are commonly used to select a free hole from the set of available holes.
    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.

- 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

- **Wastage of memory** is called fragmentation
- 2 types
  1. External fragmentation
  2. Internal 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.
- Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation.
- Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem.
- **Statistical analysis of first fit reveals that, even with some optimization, given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the 50 percent rule.**
- Internal fragmentation may also occur.
- 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.
- 2 bytes may not be enough to hold any other process and thus causes wastage.
- **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 compile or load time, compaction cannot be done; **compaction is possible *only* if relocation is dynamic and is done at execution time.**
- If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address.
- The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.
- 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 techniques achieve this solution: paging and segmentation**

PAGING

- Paging is a memory-management scheme that permits the **physical address space of a process to be non-contiguous.**
- **Paging avoids external fragmentation and the need for compaction.**
- When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store.
- **The backing store also has the same fragmentation problems** same as that of main memory, but since access is much slower, compaction is impossible.

Basic Method

- 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
- The backing store also is divided into fixed-sized blocks that are of the same size as the memory frames.
- A page table is used to keep track of the pages stored in frames.

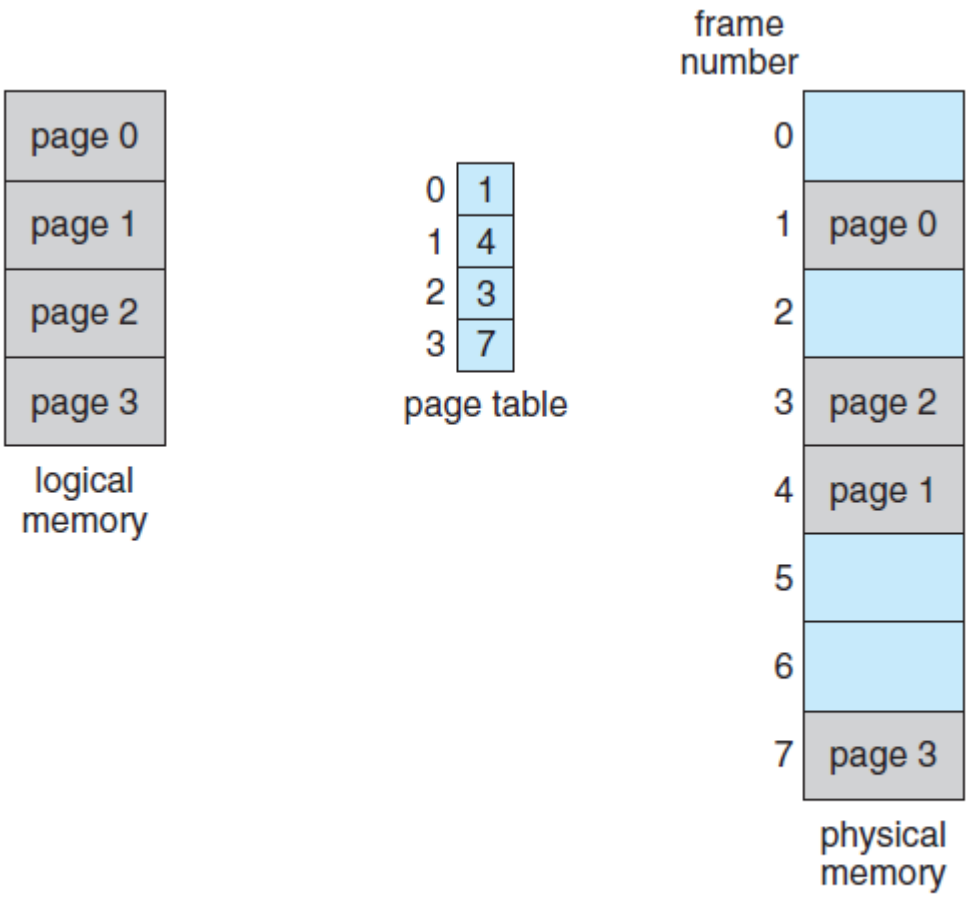


Figure 8.11 Paging model of logical and physical memory.

- Every address generated by the CPU is divided into two parts:
    1. Page number (p)
    2. 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.

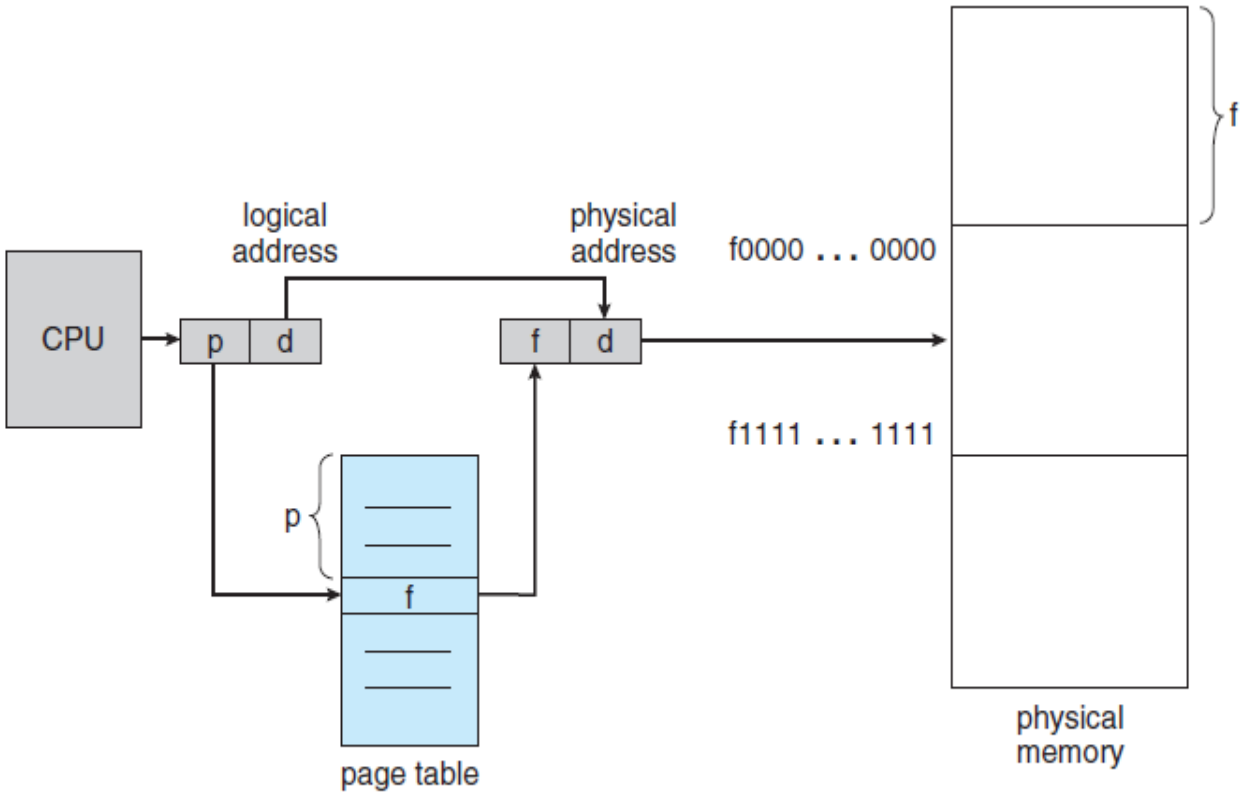
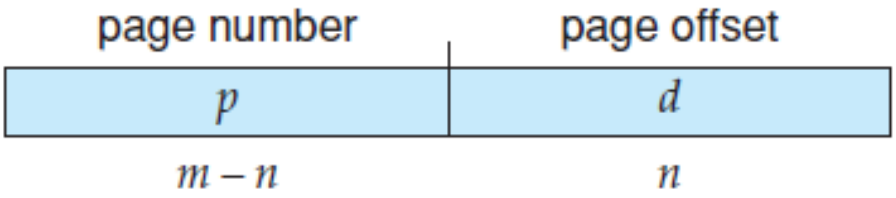


Figure 8.10 Paging hardware.

- The size of a page is typically a power of 2 (Eg 512KB or 1024KB etc) depending on the computer architecture.
- If the size of the **logical address space is 2^m**, and a **page size is 2^n** addressing units, 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.**



- For example, consider the memory in following figure

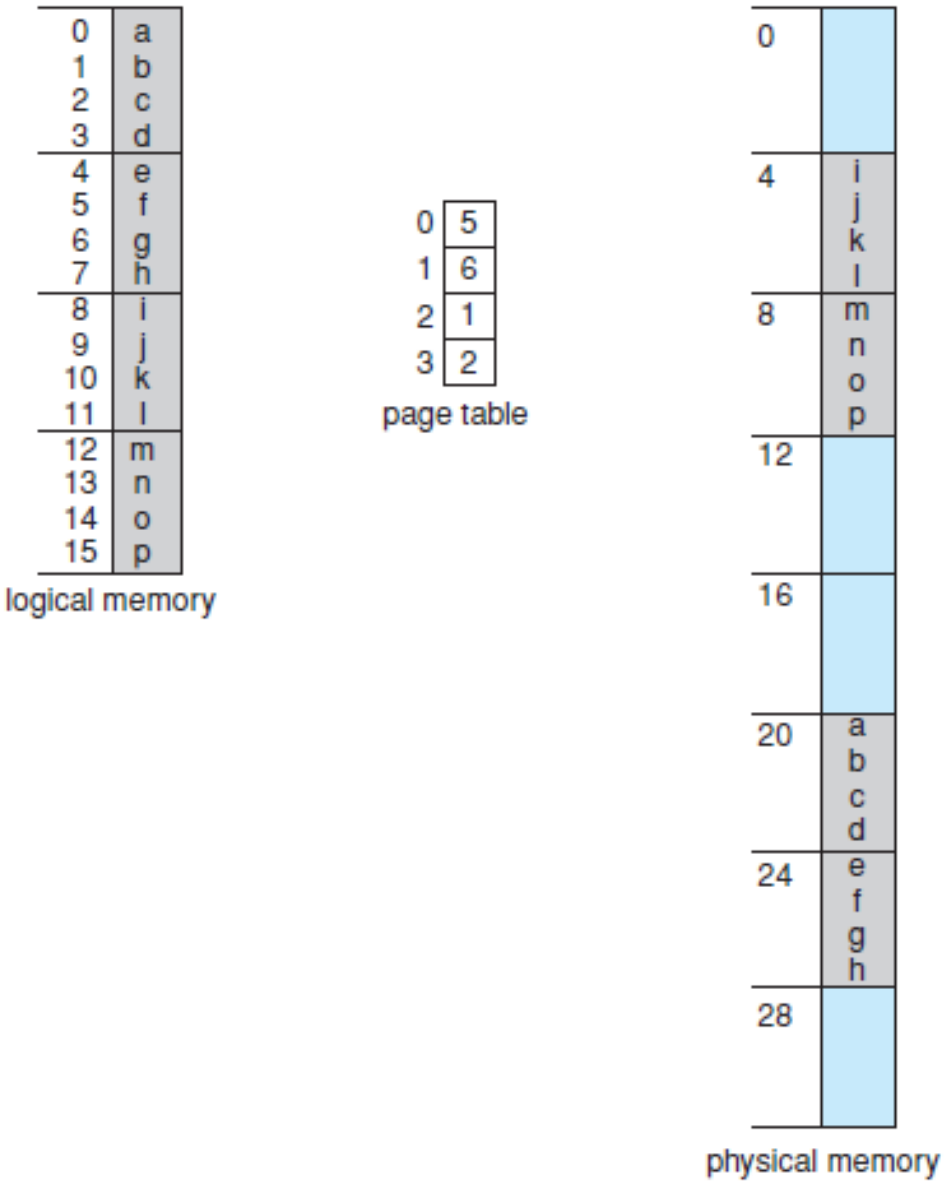


Figure 8.12 Paging example for a 32-byte memory with 4-byte pages.

- *Here m* = 4 (logical address space is 16) and n = 2 (page size is 4). Physical memory is of 32 bytes (8 frames)
- 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 (5 x 4) + 0 = 20
- Logical address 3 (page 0, offset 3) maps to physical address (5 x 4) + 3 = 23

- 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 (6 x 4) + 0 = 24
- Logical address 13 maps to physical address 9.
- If the size of the **logical address space is 2^m**, and a **page size is 2^n** addressing units, 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.**
- Eg: Logical address = 13 (Binary 1101).
- In our example, m-n bits (4-2=2 bits) are used for page and n bits (2 bits) for offset.
- **We can easily find out the page number and offset value.** Page = 3 (binary 11). Offset = 1 (Binary 01)
- Paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.
- In 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**.
- The *last* frame allocated may not be completely full.
- 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
- This consideration suggests that 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.
- Today, pages typically are between 4 KB and 8 KB in size and some systems support even larger page sizes.
- Some OS even support **multiple page sizes based on the application**. Eg: Solaris
- OS 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 frame is free or allocated and, if it is allocated, to which page of which process
- In every memory access, the logical address is converted into physical address through paging. **Paging therefore increases the context-switch time.**
- A pointer to the page table is stored in PCB of each process

Hardware Support

- The simplest method, 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. Every access to memory must go through the paging map, so **efficiency is a major consideration.**
- This implementation is **satisfactory if the page table is reasonably small**
- Some systems allow the page table to be very large, and then the page table is **kept in main memory. It takes memory access for page table access also.**
- The standard solution to this problem is to use a special, small, **fast lookup hardware cache**, called a **Translation Look-aside Buffer** (**TLB**).
- TLB is associative, high-speed cache memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is given with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the hardware is expensive.
- The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is given to the TLB. **If the page number is found (TLB hit),** its frame number is immediately available and is used to access memory. If the page number is not in the

TLB **(TLB miss)** a memory reference to the page table must be made. System **adds 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, the operating system must select one for replacement. **Various replacement algorithms are there**

- The percentage of times that a particular page number is found in the TLB is called the **hit ratio**. Eg 80%

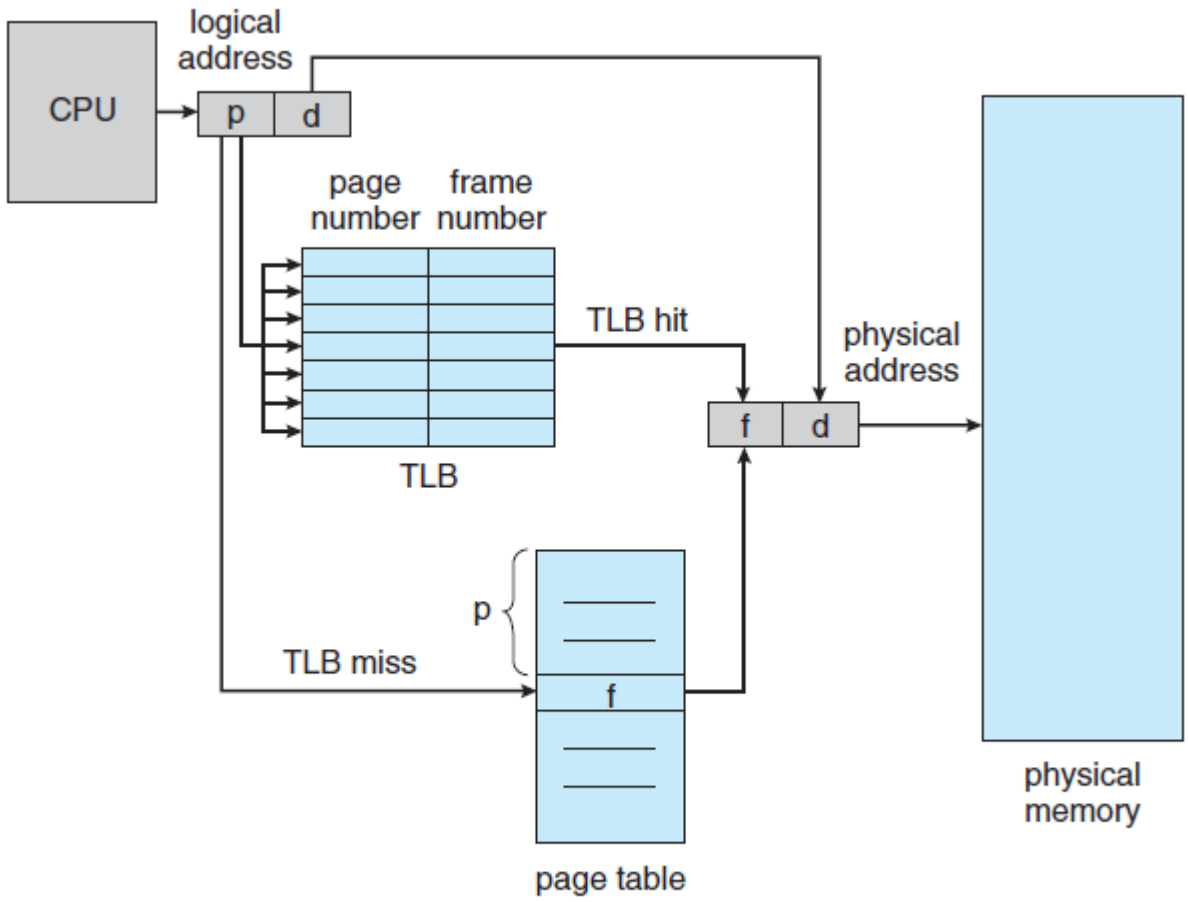


Figure 8.14 Paging hardware with TLB.

Protection

- User process is **unable to access memory it does not own**.
- Also it should specify the **set of operations permitted** on a memory location
- **Protection bits** are added with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read-write or read-only. Eg: 0 – read only. 1 – read write
- An attempt to write to a read-only page causes a hardware trap
- One additional bit is generally attached to each entry in the page table called **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.

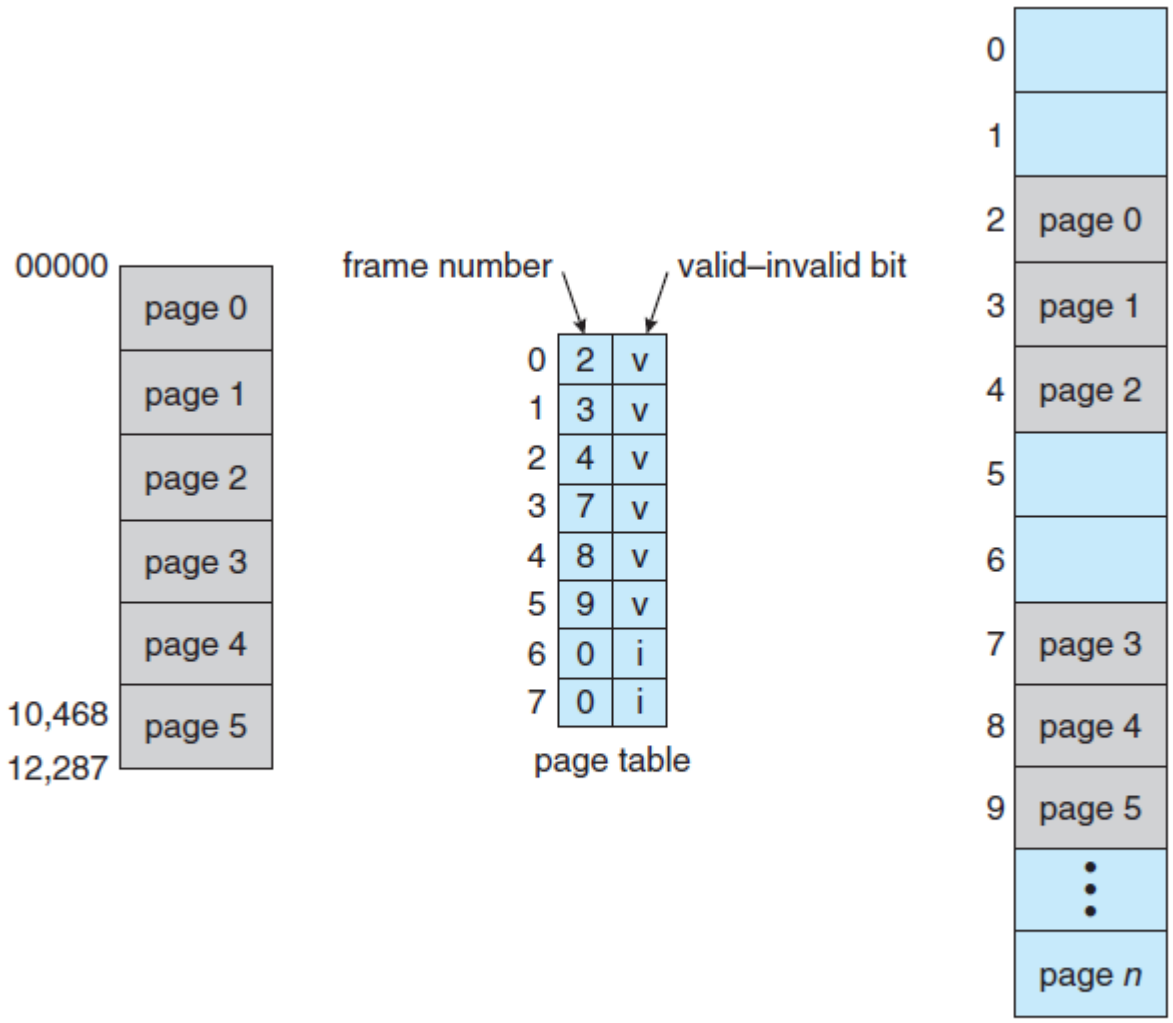


Figure 8.15 Valid (v) or invalid (i) bit in a page table.

- Any attempt to generate an address in pages 6 or 7 the computer will trap to operating system error

Shared Pages

- **An advantage of paging is the possibility of *sharing* common code.**
- 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.

- 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 *2150* KB instead of 8,000 KB-a significant savings.
- Other heavily used programs can also be shared - compilers, window systems, run-time libraries, database systems, and so on.
- **Some operating systems implement shared memory using shared pages**.

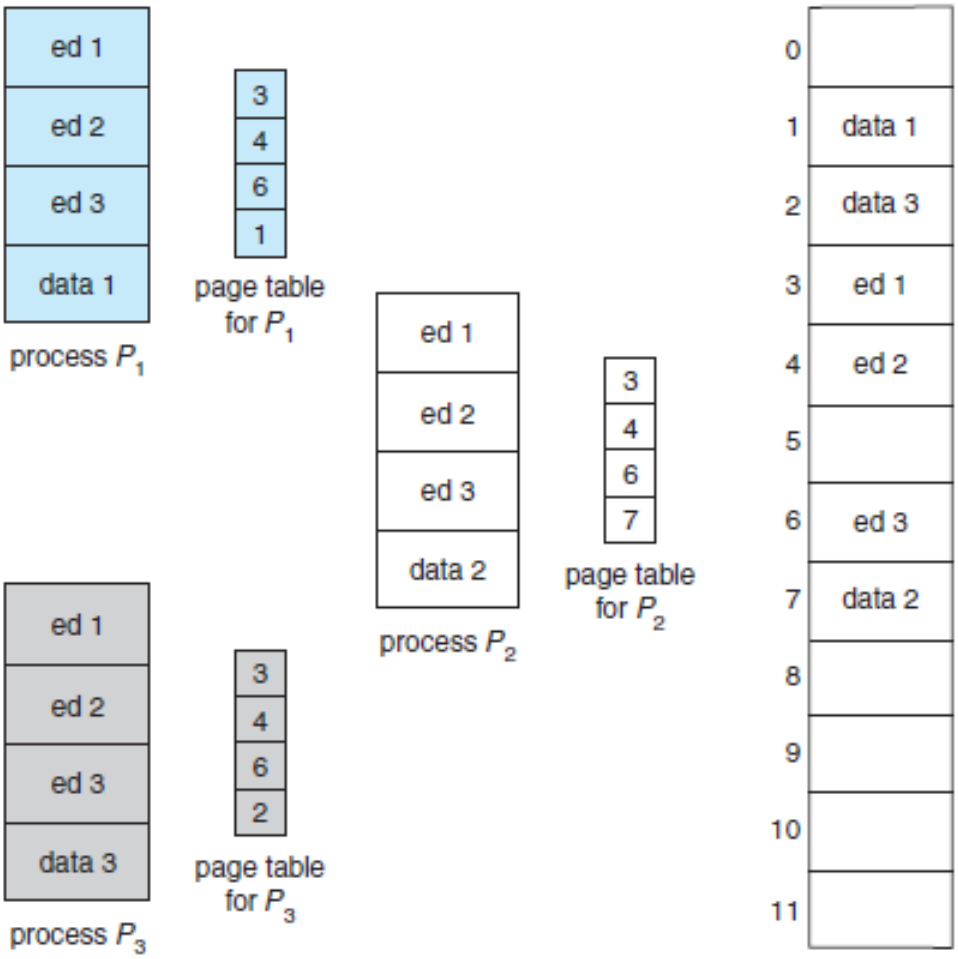


Figure 8.16 Sharing of code in a paging environment.