Memory Management

Outline of this lecture

In this lecture, we will discuss the following:

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Segmentation
- Segmentation with Paging

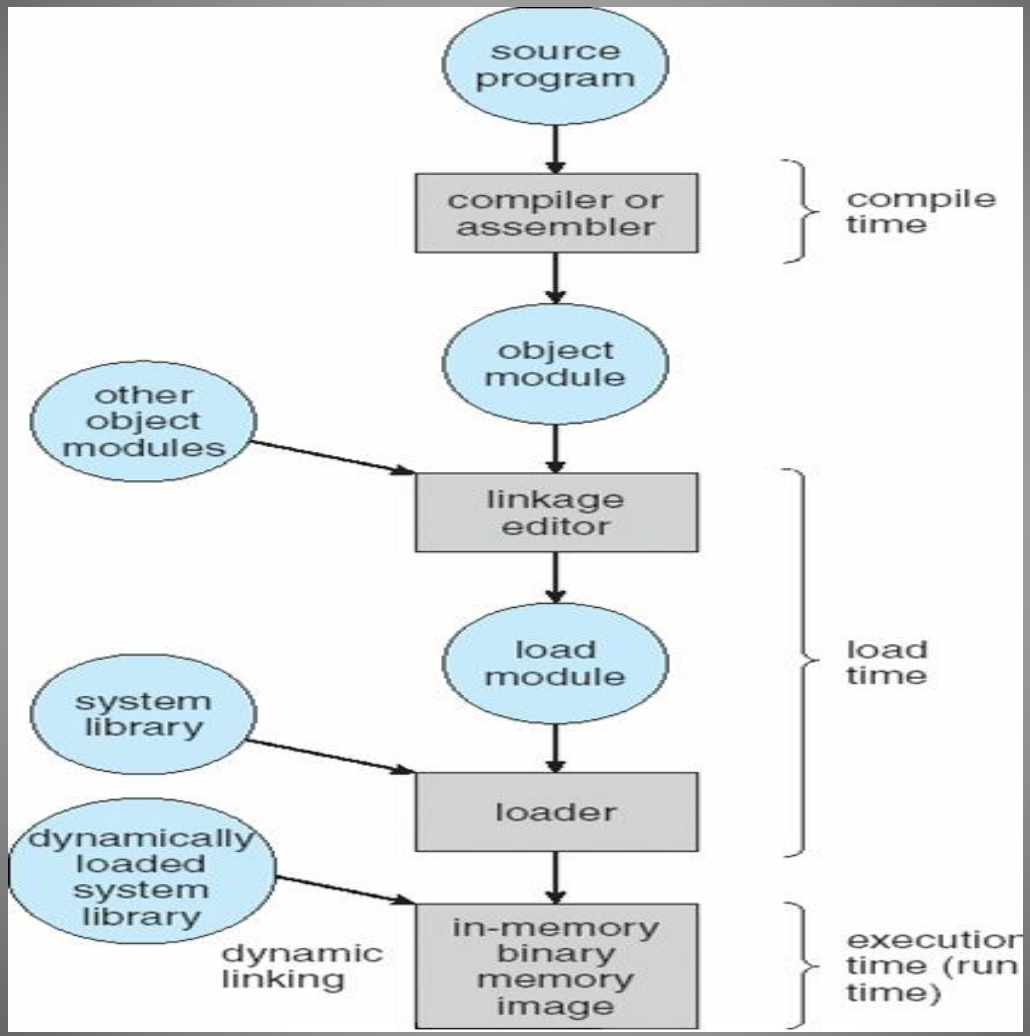
Background

- Program must be brought into memory and placed within a process for it to be executed.
- A process may be moved between memory and disk during execution.
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory for execution.
- User programs go through several steps before being executed.

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

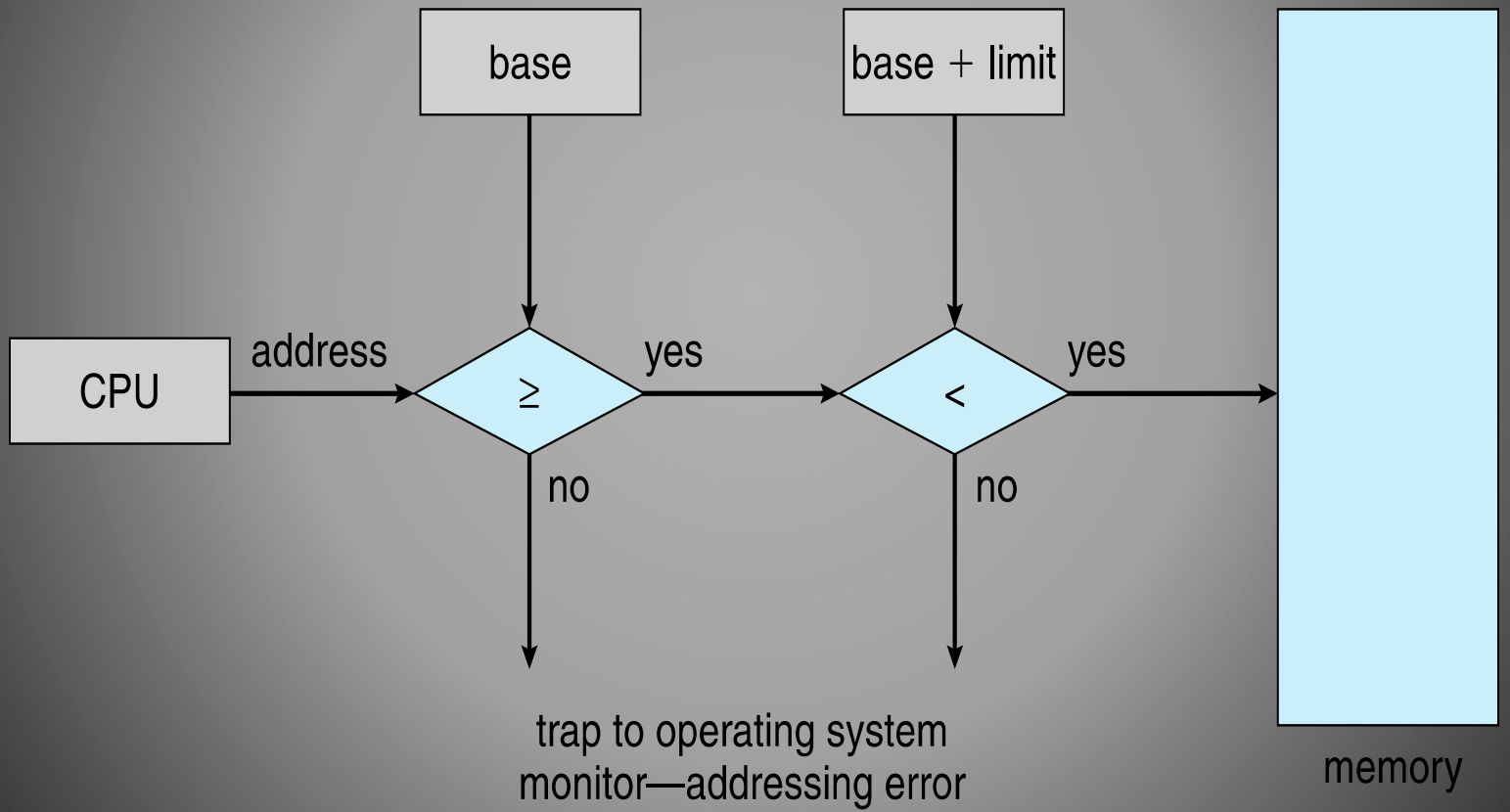
Multistep Processing of a User Program



Logical vs. Physical Address Space

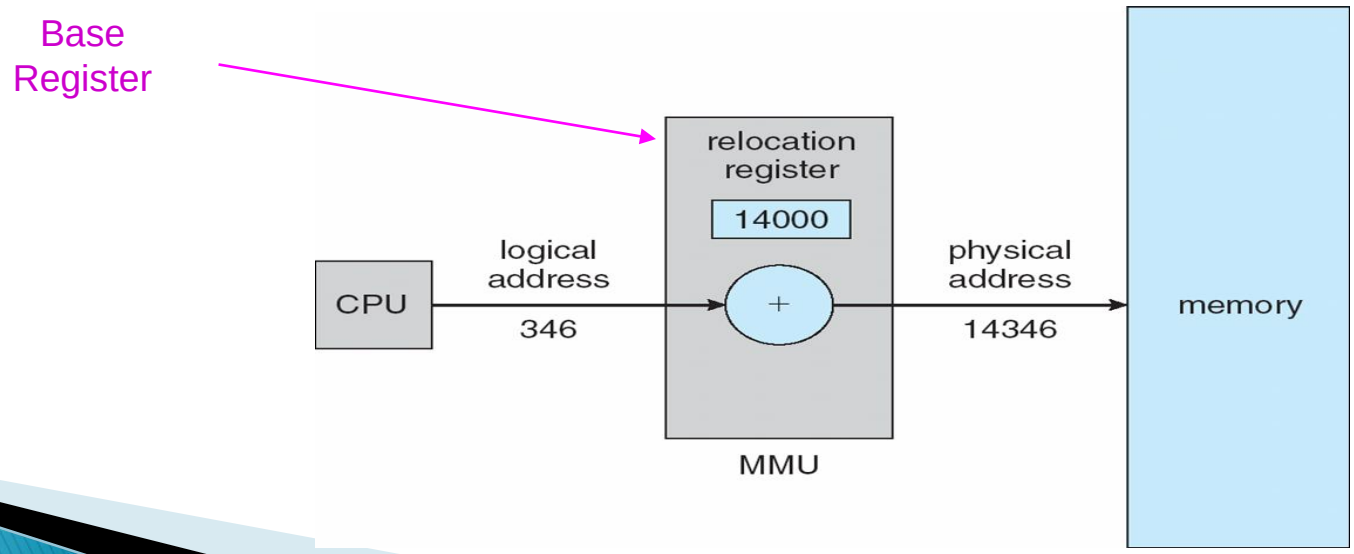
- The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management.
  - *Logical address* – generated by the CPU; also referred to as *virtual address*.
  - *Physical address* – address seen by the memory unit- the one loaded into memory-address register of the memory.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes.
- logical (virtual) and physical addresses differ in execution-time address-binding scheme.

Hardware Address Protection with Base and Limit Registers



Memory-Management Unit (MMU)

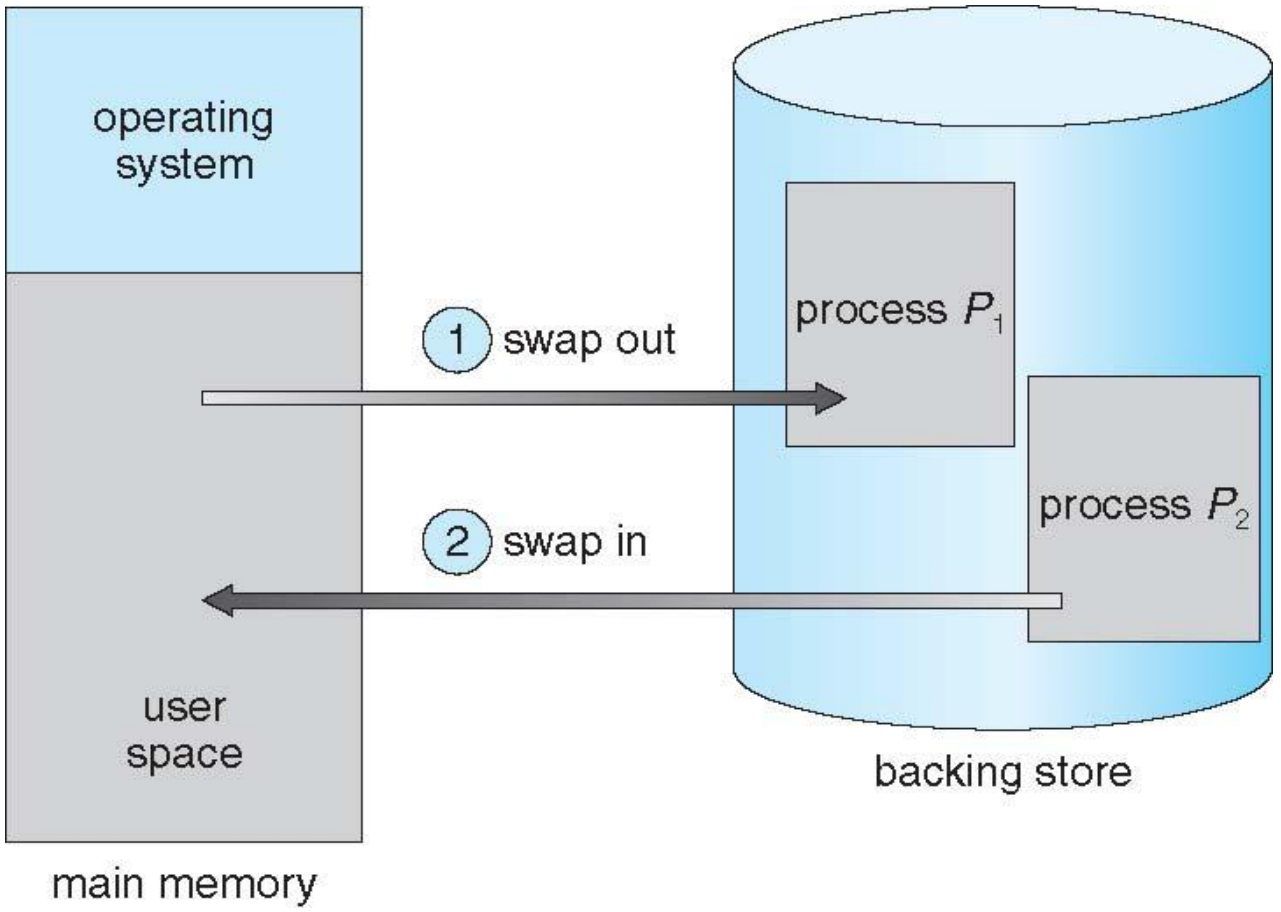
- MMU is a HW device that maps virtual to physical address.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.



Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution. (ex: Round Robin Algorithm)
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- *Roll out, roll in* – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method, if binding is done
  - At assembly or load time, p moves to the same location
  - At execution time, process can be swapped into diff memory space because physical add are computed during execution time
- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.

Schematic View of Swapping



Context Switch Time including Swapping

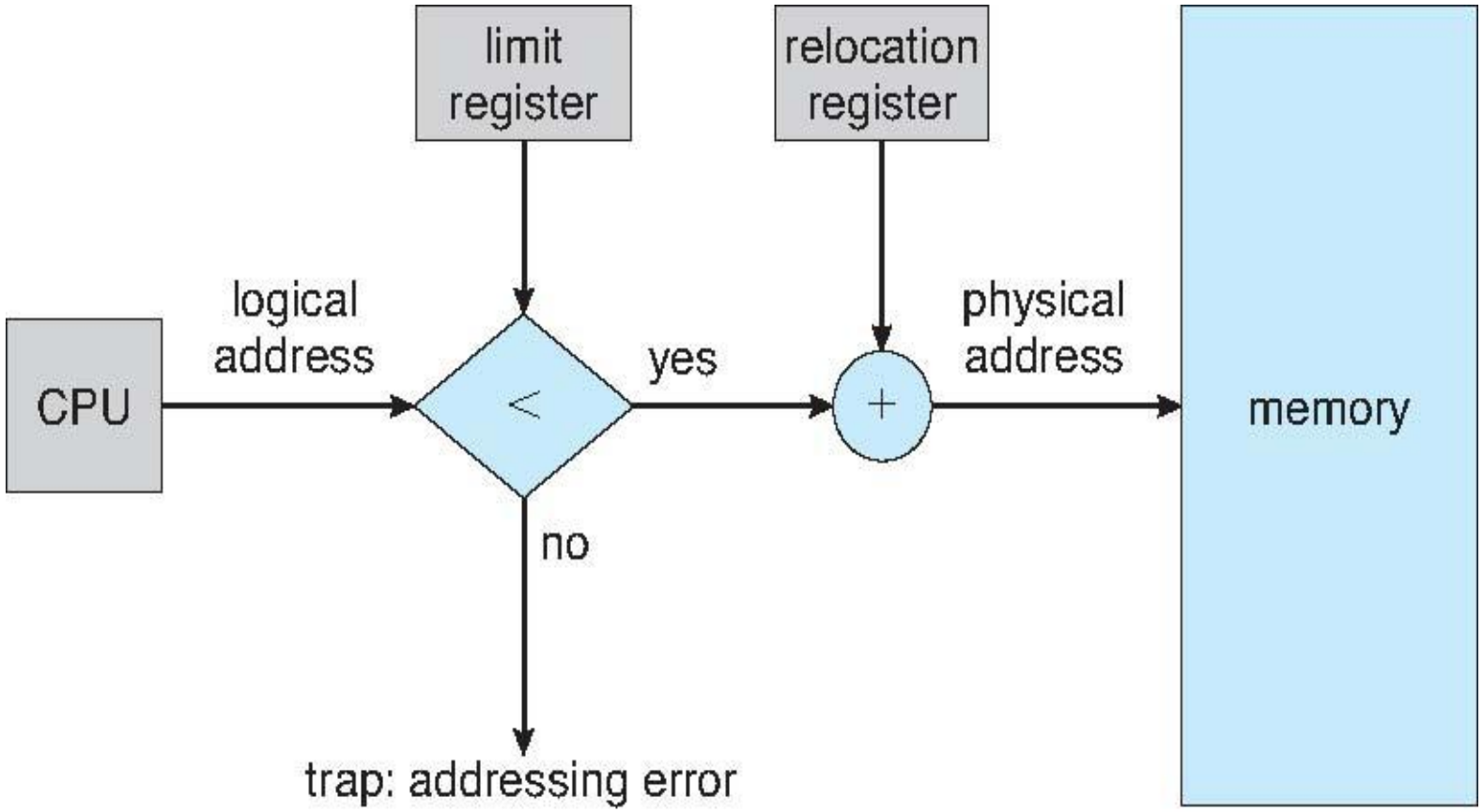
- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high, to get an idea of this time
- 100MB/sec user process swapping to hard disk with transfer rate of 50MB/sec
  - The actual transfer of this p to/from main memory takes 2sec (100/50)
  - Average latency of 8ms
  - Total context switch swapping component time of 4016ms
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request memory()` and `release memory()`
- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
- Standard swapping not used in modern operating systems
  - But modified version common –Swap only when free memory extremely low

Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method→ each p is contained in a single section of memory that is contiguous to the section containing the next p.
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

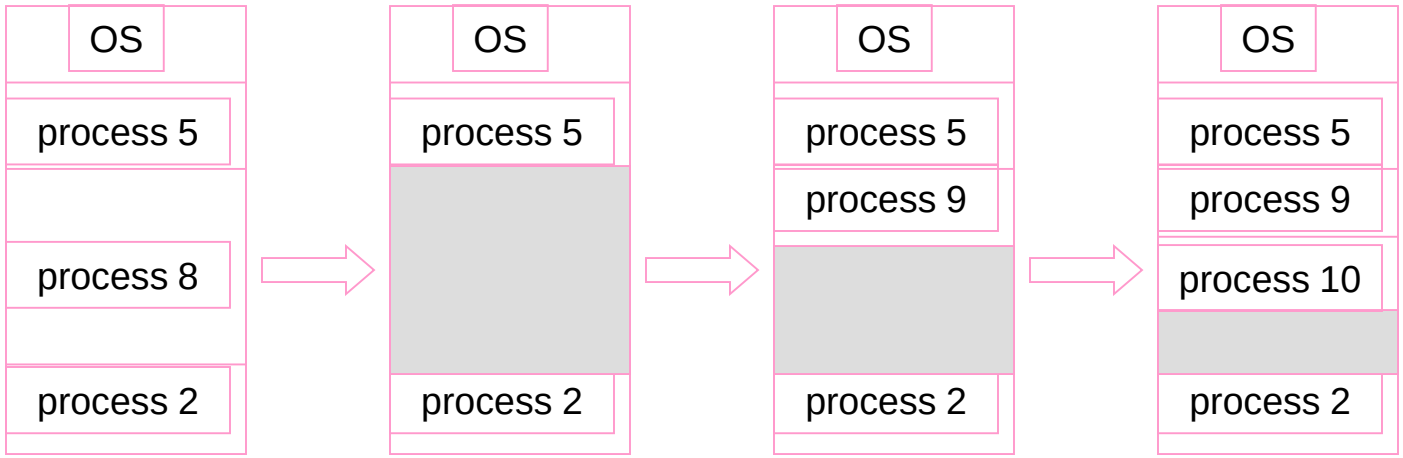
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

Hardware Support for Relocation and Limit Registers



Contiguous Allocation (cont.)

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

How to satisfy a request of size *n* from a list of free holes --→ placement policy.

Want to select the best locations in which to place the processes

- **First-fit**: Allocate the *first* hole that is big enough, search can start at the beginning of the set of holes or where prev *first-fit* search ended.
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

First-fit and best-fit better than worst-fit in terms of speed and storage utilization.

Placement policies – an example

- A variable partition memory system has the following hole sizes:
  - 20 15 40 60 10 25
- Show the state of memory after processes of 25K, 30K, 23K, and 37K (in request order) arrive using the best, first and worst fit placement policy?

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.**

Fragmentation

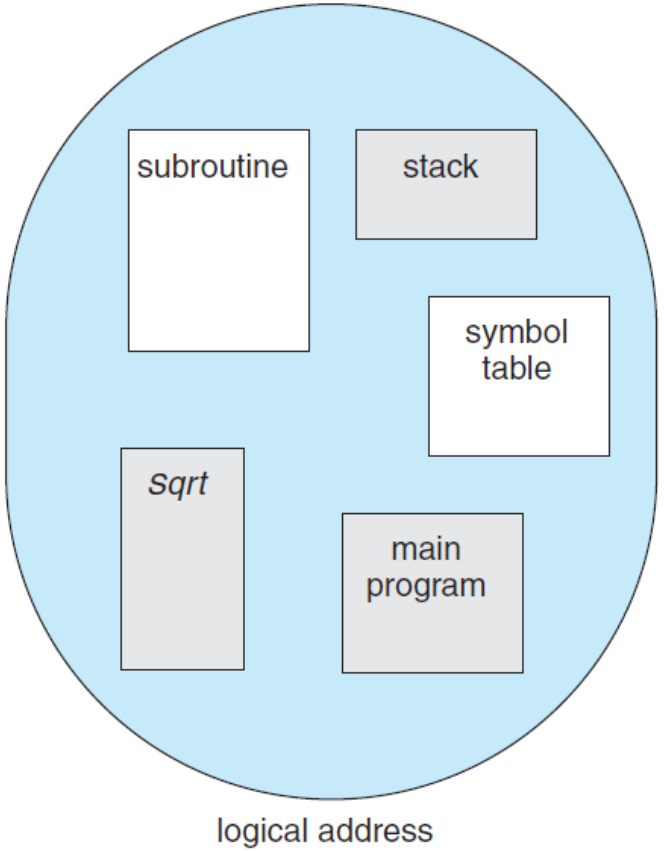
- **External Fragmentation**: Enough total memory space exists to satisfy a request, but it is not contiguous.
  - Solutions
    - Compaction- Shuffle memory contents to place all free memory together in one large block.

      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.
    - Allow the logical-address space of a process to be noncontiguous. Ex: Paging and Segmentation

- **Internal Fragmentation**: when memory allocated is slightly larger than the size needed by the process. the size difference between the process size and the allocated memory is memory that is internal to a partition, but not being used.

Segmentation

- Do programmers think of memory as a linear array of bytes, some containing instructions and others containing data? Most programmers would say “no.”
- Rather, they prefer to view memory as a collection of variable-sized segments, with no necessary ordering among the segments



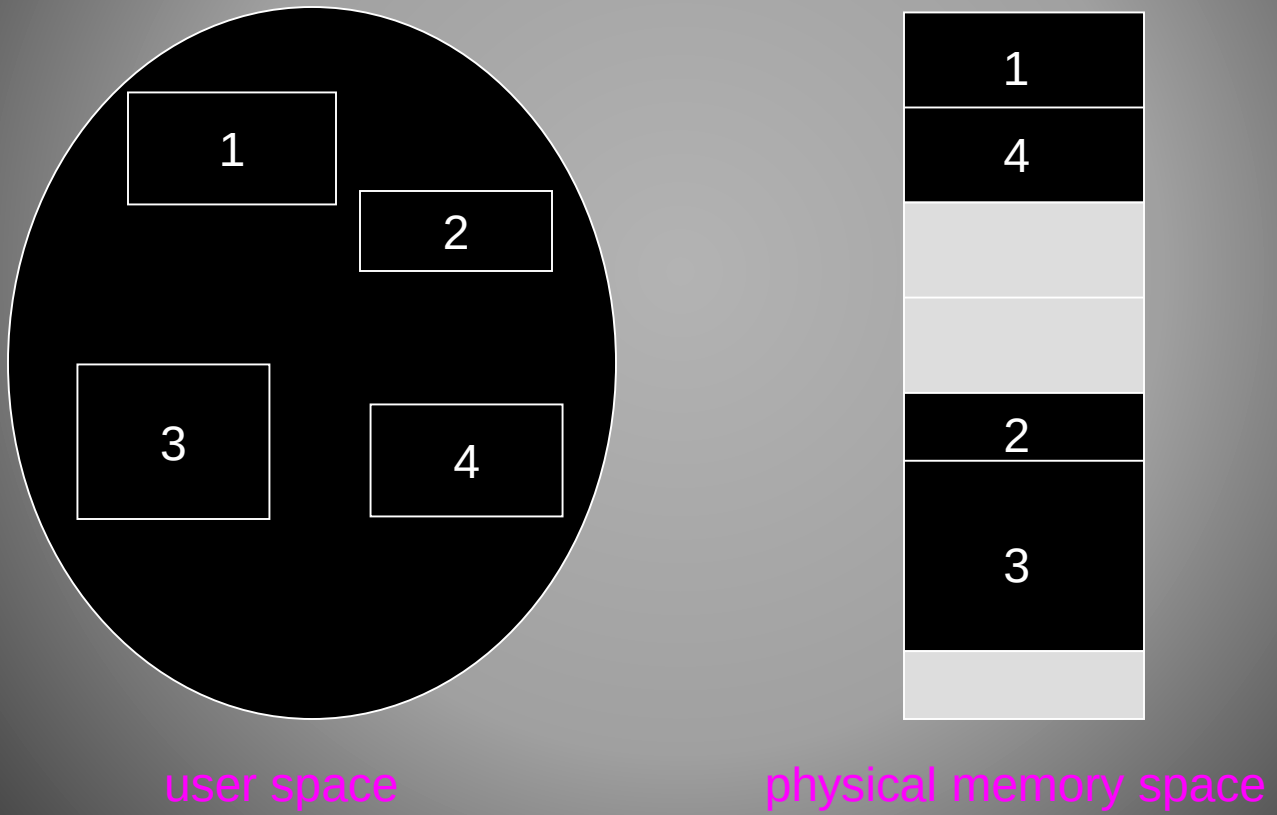
Programmer’s view of a program.

Segmentation

- Memory-management scheme that supports user view of memory.
- A logical address space is a collection of segments.
- A program is a collection of segments. A segment is a logical unit such as:

  main program,
  procedure,
  function,
  local variables, global variables,
  common block,
  stack,
  symbol table, arrays
- Each segment has a name and a length. For simplicity, the segments are given numbers.

Logical View of Segmentation



Segmentation Architecture

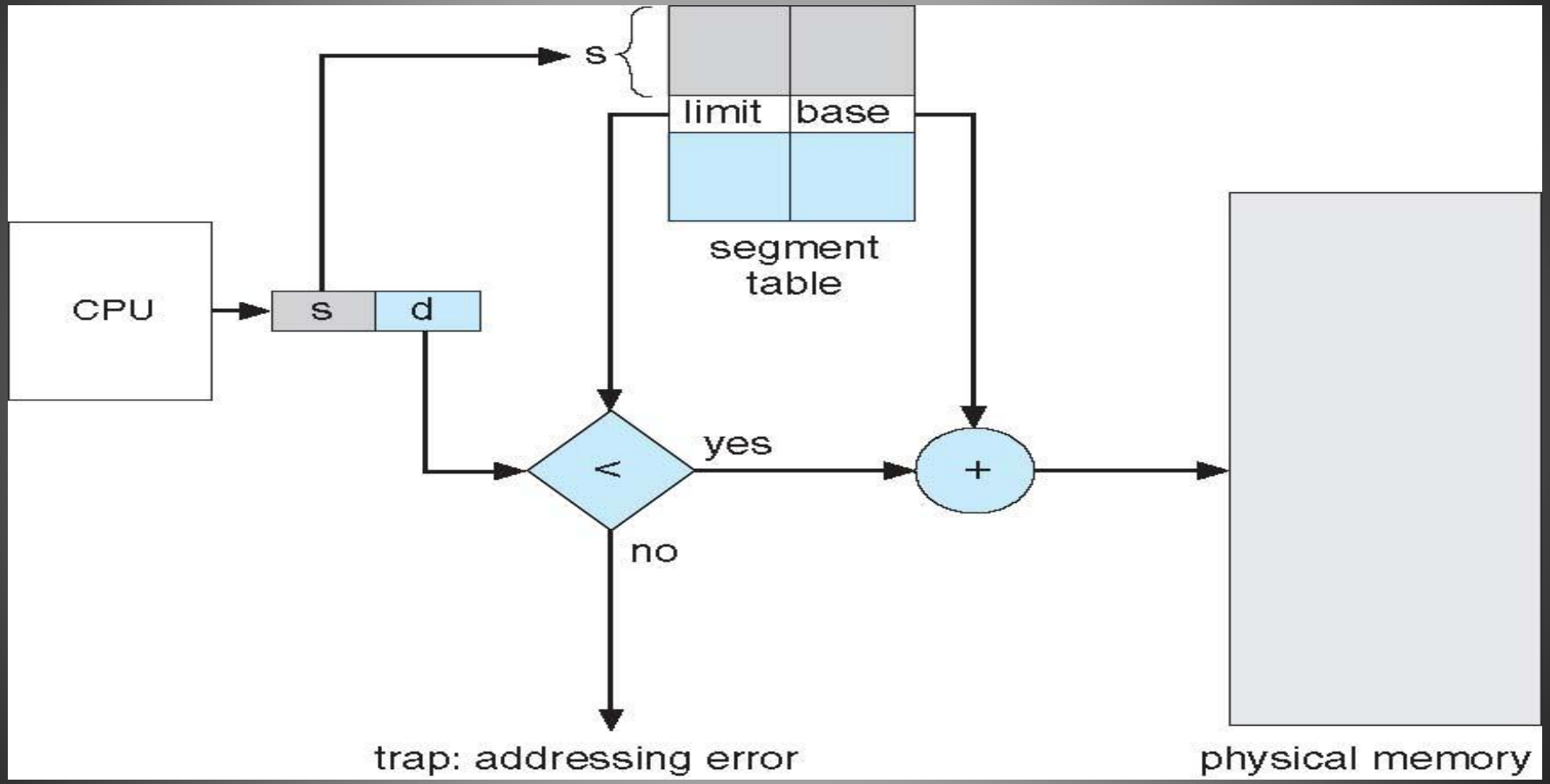
- Logical address consists of a two tuple:

  <segment-number, offset>,

- *Segment table* – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory.
  - limit – specifies the length of the segment.
- *Segment-table base register (STBR)* points to the segment table's location in memory.
- *Segment-table length register (STLR)* indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**.

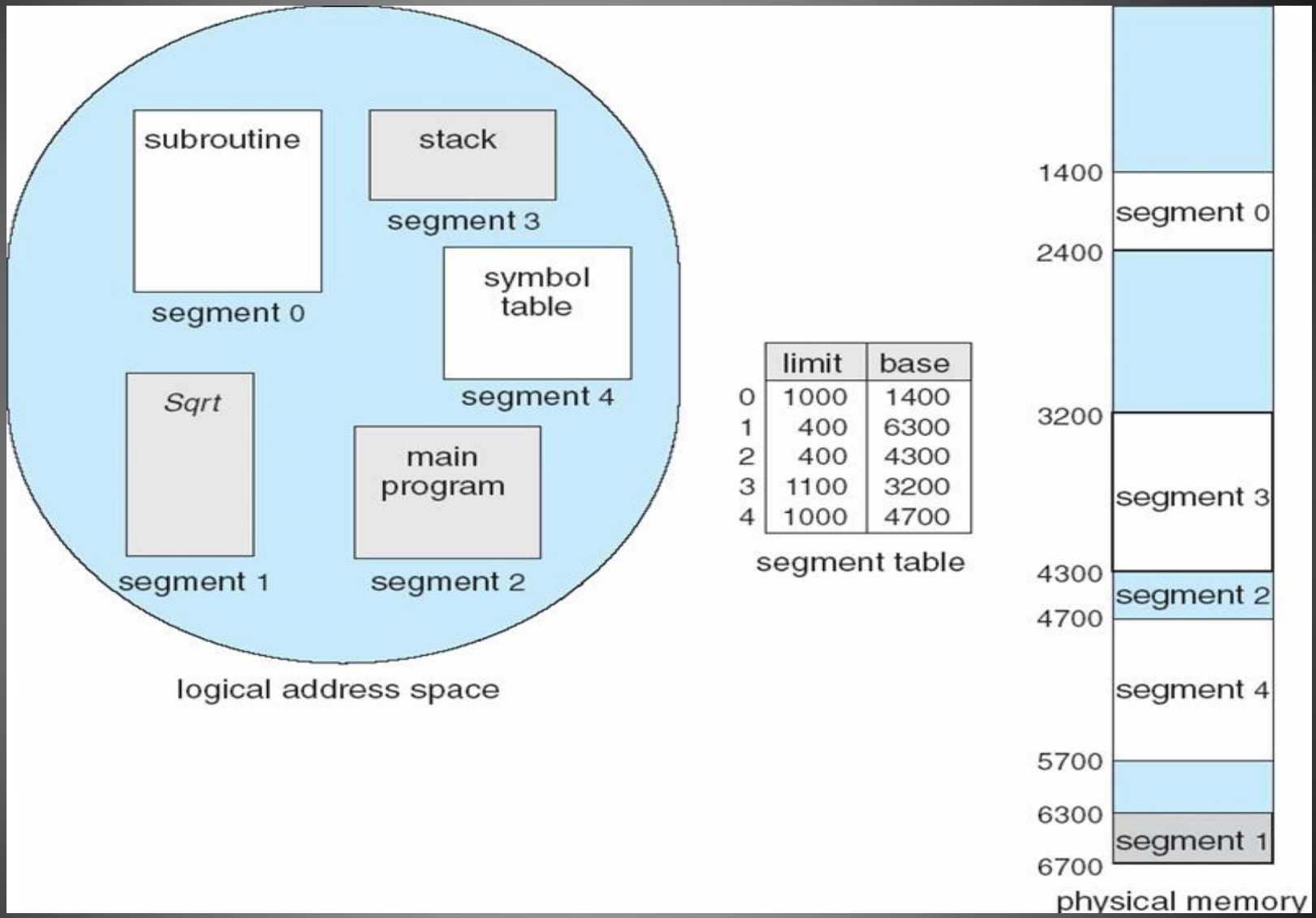
Segmentation Hardware



s : segment number

d: offset. Must be between 0 and the segment limit

Example of Segmentation



Exercise

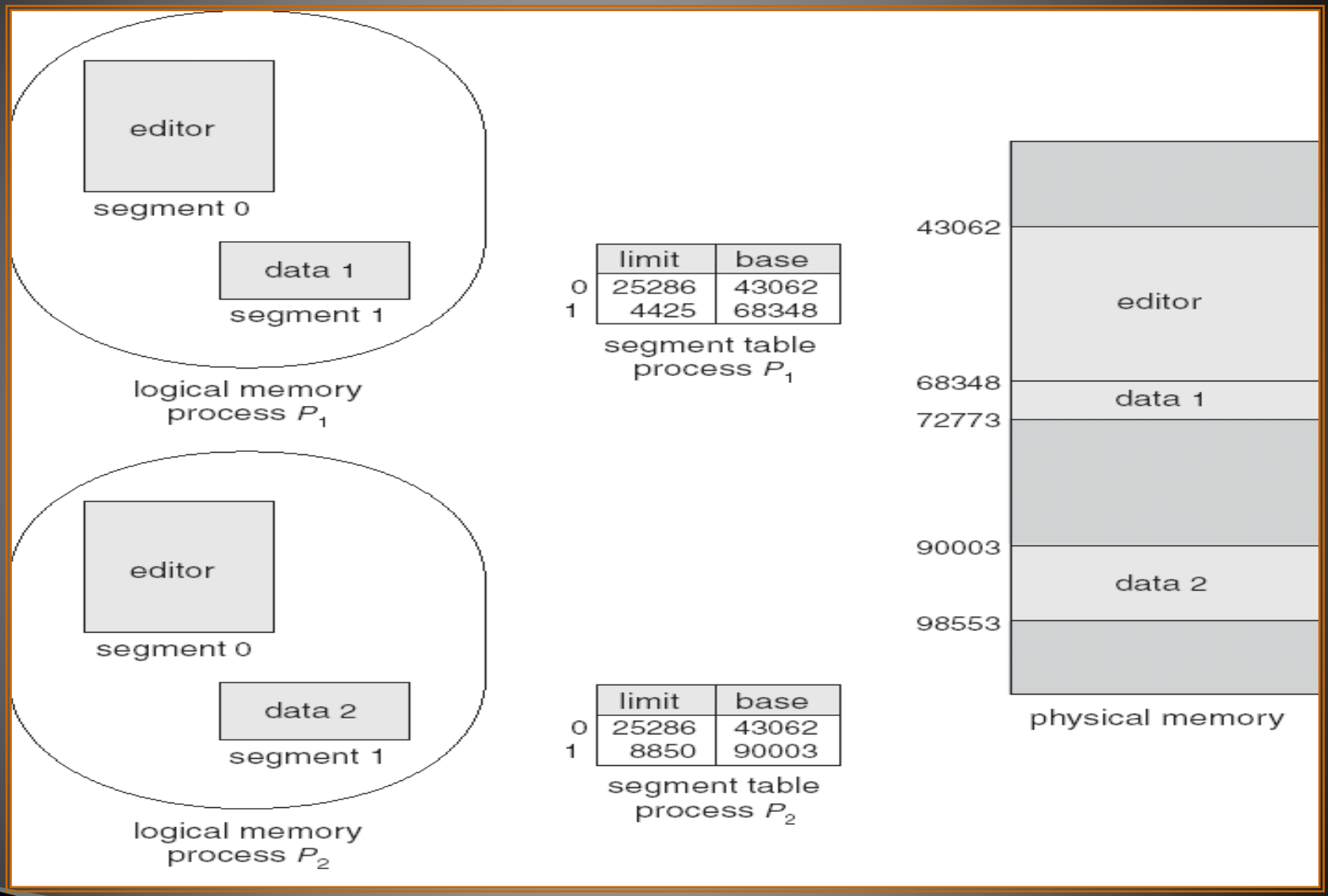
•Consider the following segment table:

Segment	Base	Limit
0	222	800
1	9900	14
2	90	500
3	1524	580
4	2784	400

- What are the physical addresses for the following logical addresses?

Logical address <s, d>	Physical address
0, 400	
1, 15	
2, 320	
3, 850	
4, 346	

Sharing of Segments

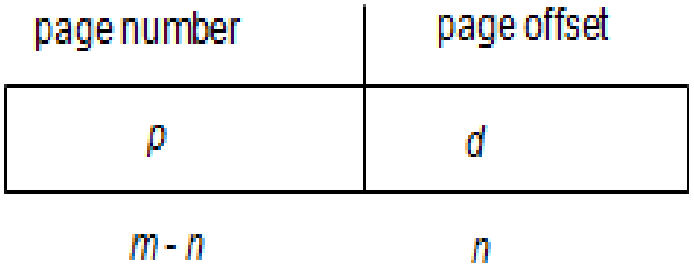


Paging

- Paging allows the physical address space of a process to be non-contiguous
- Paging relieves fragmentation issue. Single process can be distributed over a number of holes.
- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 1 GB per page).
- Divide logical memory into blocks of same size called pages.
- Keep track of all free frames.
- To run a program of size *N* pages, need to find *N* free frames and load program.
- Set up a page table to translate logical to physical addresses
- Internal fragmentation

Address Translation Scheme

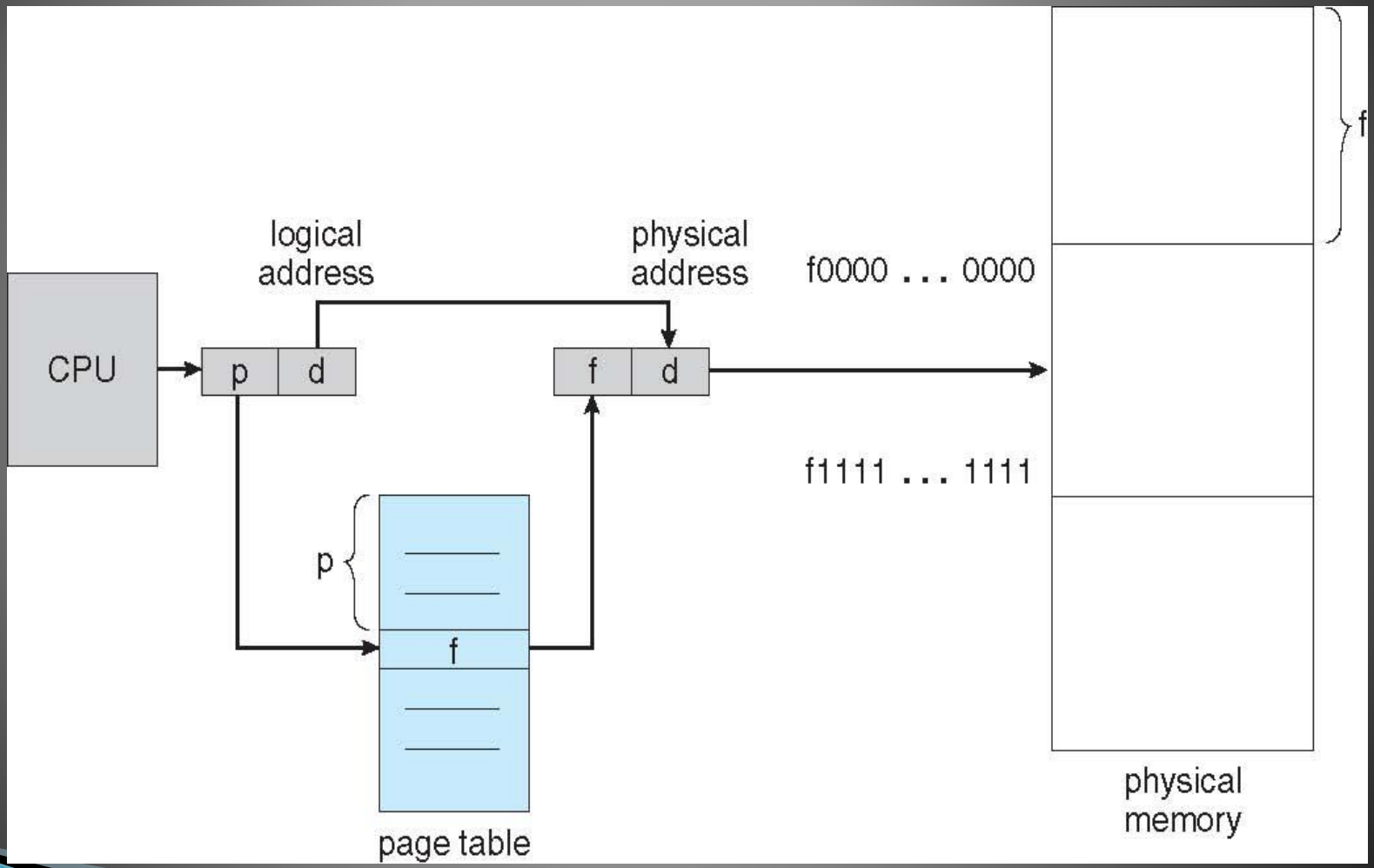
- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory.
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit.



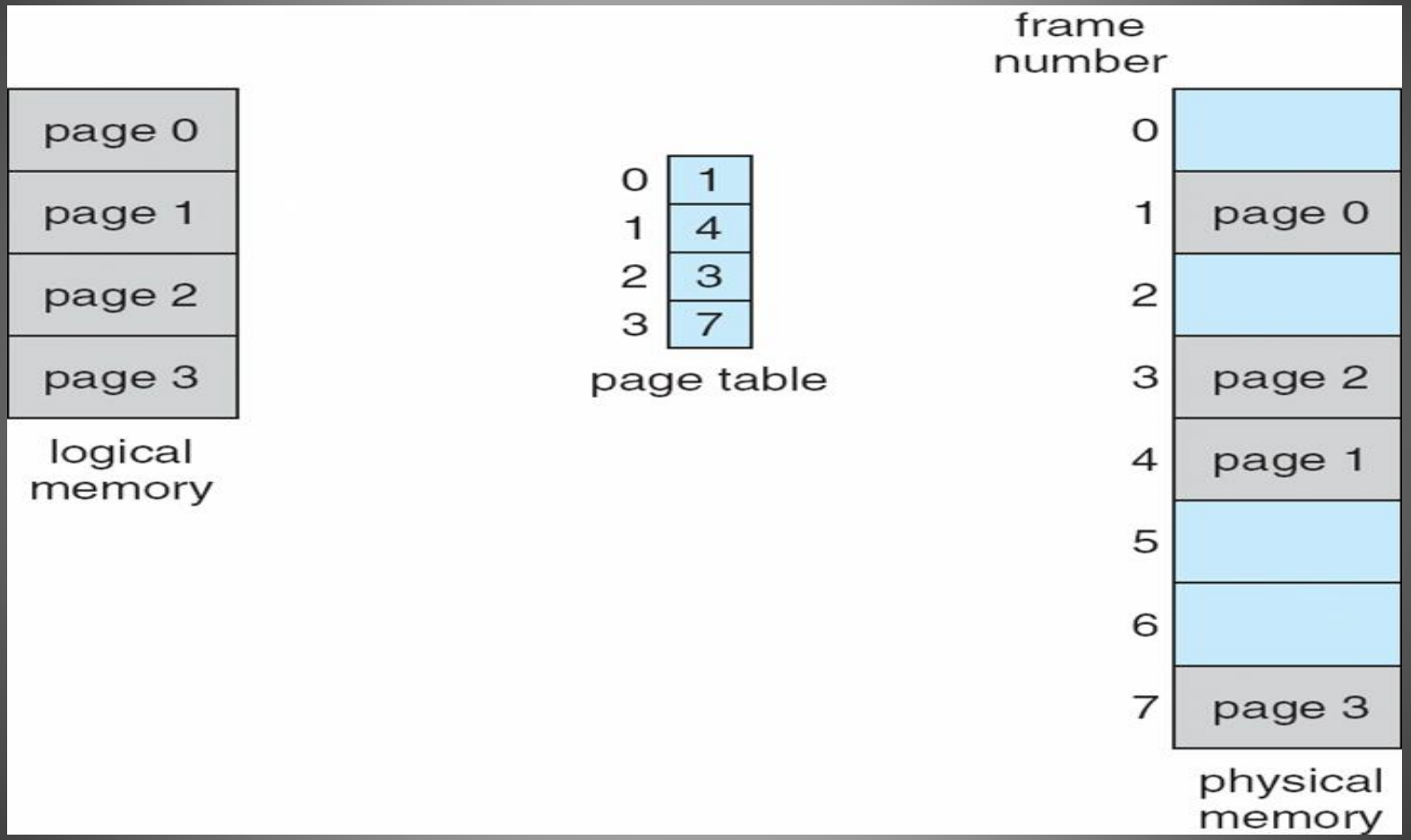
For given logical address space 2^m and page size 2^n bytes

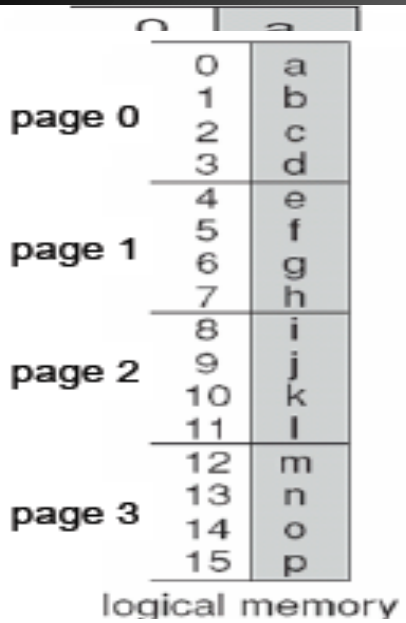
Page size and frame size are defined by the HW

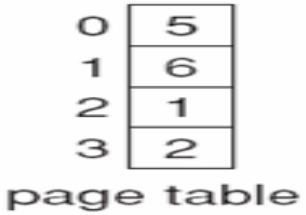
Paging Hardware

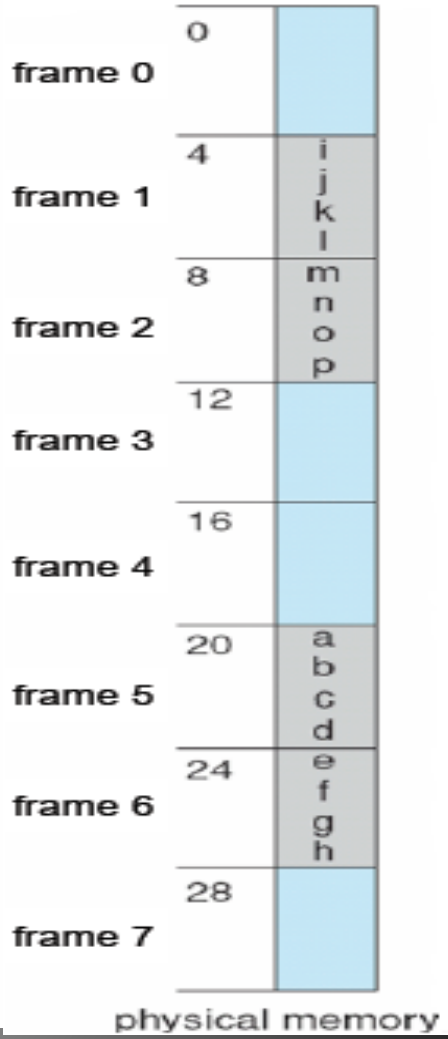


Paging Model of Logical and Physical Memory









- Logical address 10 is in page 2 at offset 2
- According to the page table, page 2 is located in frame 1
- Physical memory address is: (Frame # * Page size) + Offset
- Physical memory address for logical address 10 is : (1 * 4 bytes) + 2 byte offset = 6

n=2 and *m*=4 32-byte memory and 4-byte pages

- Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.
  a. How many bits are required in the logical address?

1. Consider that the physical memory is 256K, logical memory (per process) is 64K and the page size is 2K. Calculate the following values

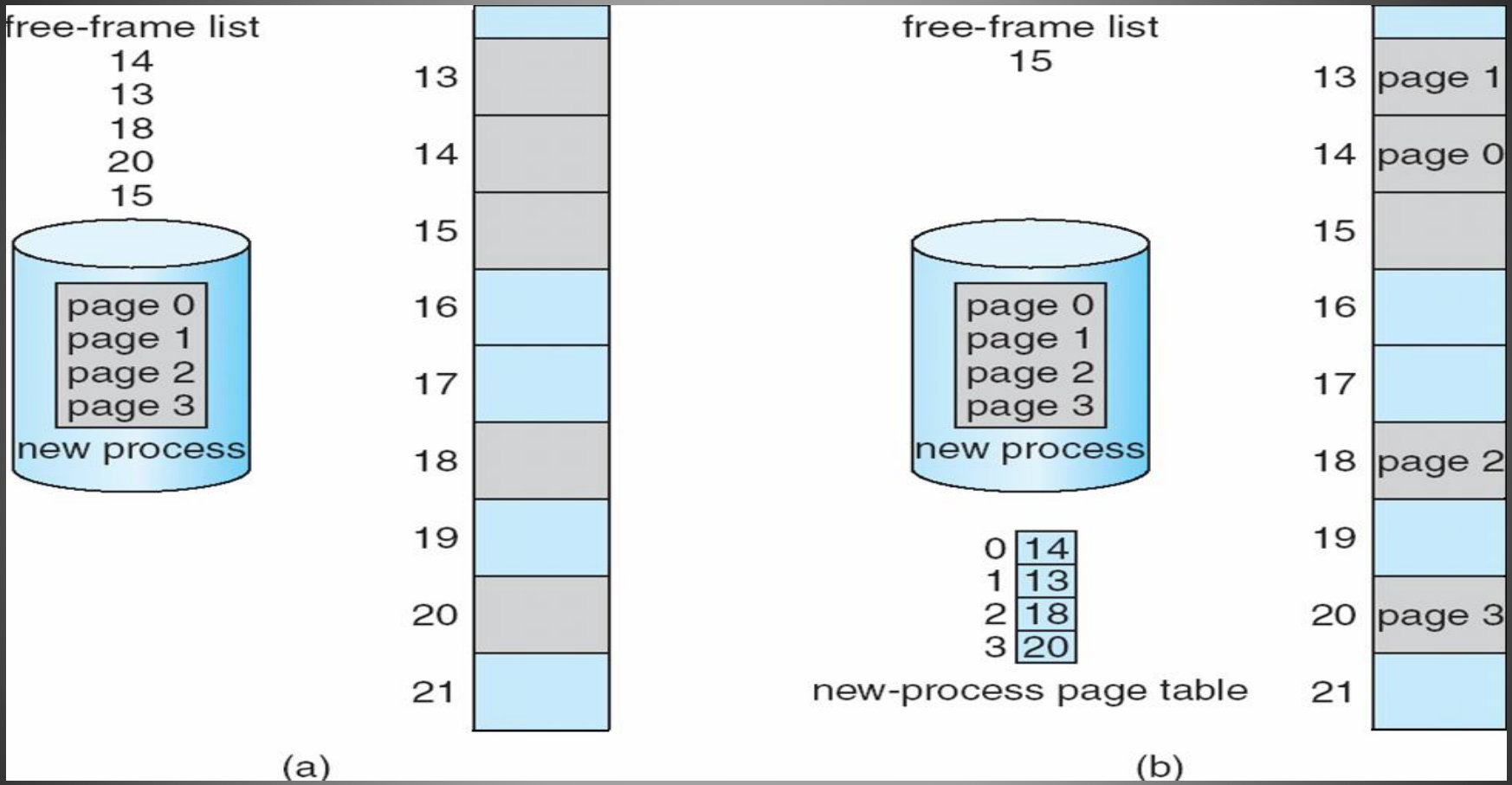
a) Number of pages per process

b) Number of frames in physical memory

c) Number of bits indicating page number

d) Number of bits indicating offset

Frame Allocation



Before allocation

After allocation

Implementation of Page Table

- Page table is kept in main memory.
- *Page-table base register (*PTBR) points to the page table.
- *Page-table length register* (PRLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called ***associative registers*** or *translation look-aside buffers (**TLBs**)*

Associative Register

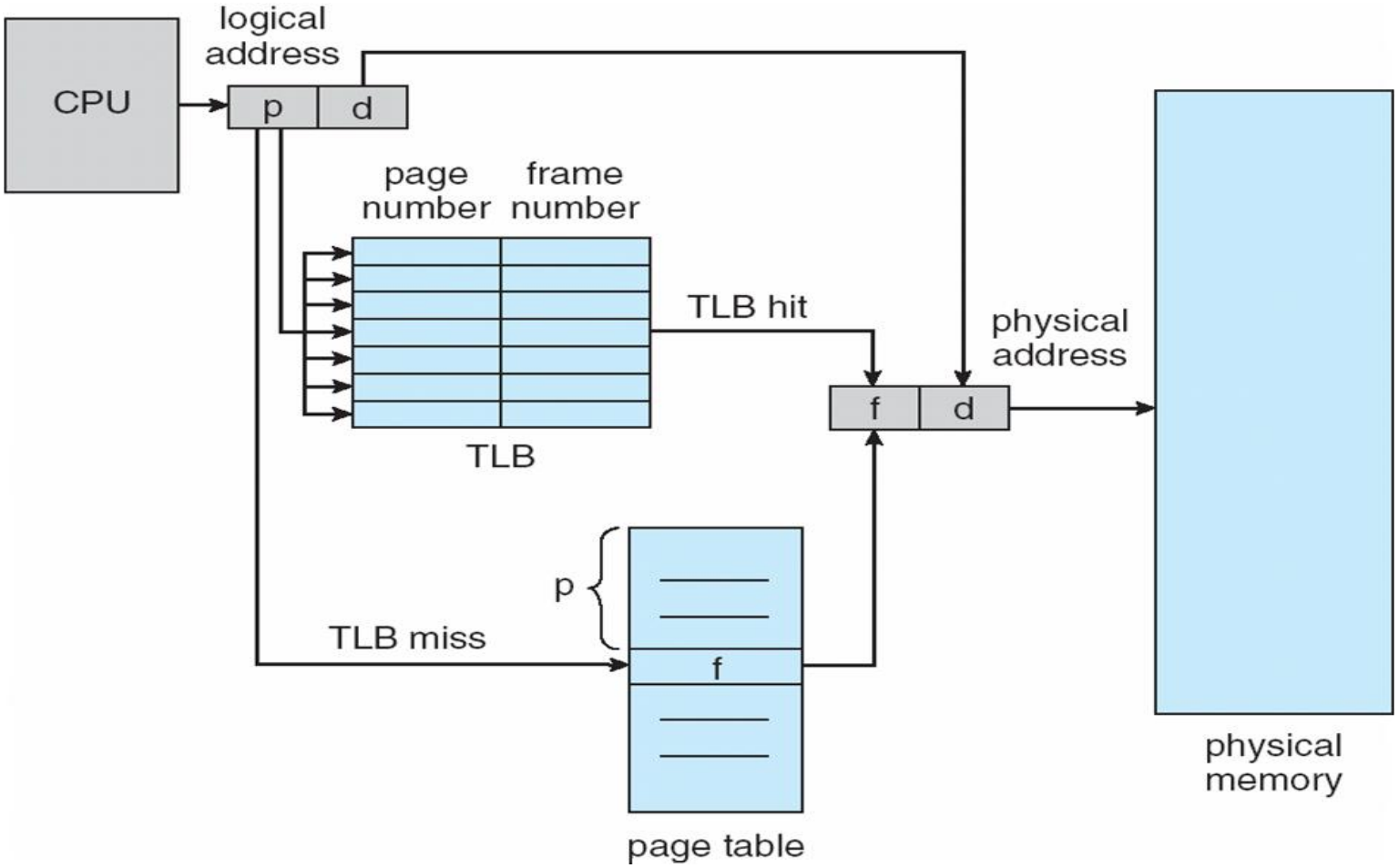
- Associative registers

Page #	Frame #

Address translation (page #, frame #)

- If page# is in associative register, get frame # out.
- Otherwise get frame # from page table in memory

Paging HW with TLB



Effective Access Time

- **Effective access time (EAT)** is the average time needed to access memory.
- **Hit Ratio = h :** The percentage of times that a particular page number is found in the TLB.
- Effective access time can be calculated based on:
  - The time it takes to access main memory
  - The time it takes to access the TLB
  - The hit ratio for the TLB
- Example: Consider h = 80%, ε = 20ns for TLB search, 100ns for memory access
  - Time to access main memory = 100ns
  - Time to access TLB = 20 ns
  - Hit ratio = 0.8 (80%) we found the desired page No in the TLB 80% of time
  - If page is found in TLB, total access time = 120?
  - If page is not found in TLB, total access time = 220?need to access memory for page table & frame No(100) & access the desire byte in memory (100)
  - Effective access time = ? 0.8 * 120 + 0.2 *220 = 140 ns

EAT = h*(c+m) + (1-h)*(c+2m)

where, h = hit ratio of TLB
m = Memory access time
c = TLB access time

Effective Access Time

Question

- A paging scheme uses a Translation Look-aside Buffer (TLB). A TLB-access takes 10 ns and a main memory access takes 50 ns. What is the effective access time (in ns) if the TLB hit ratio is 90% and there is no page-fault?

Effective Access Time

Question

- A paging scheme uses a Translation Look-aside Buffer (TLB). A TLB-access takes 10 ns and a main memory access takes 50 ns. What is the effective access time (in ns) if the TLB hit ratio is 90% and there is no page-fault?

Solution

Effective access time = hit ratio * (TLB access time + Memory access time) + miss ratio * (TLB access time + Page table access time + Memory access time)

=0.9 * (10 + 50) + 0.1 (10 + 50 + 50)
= 54 + 11 = 65 ns

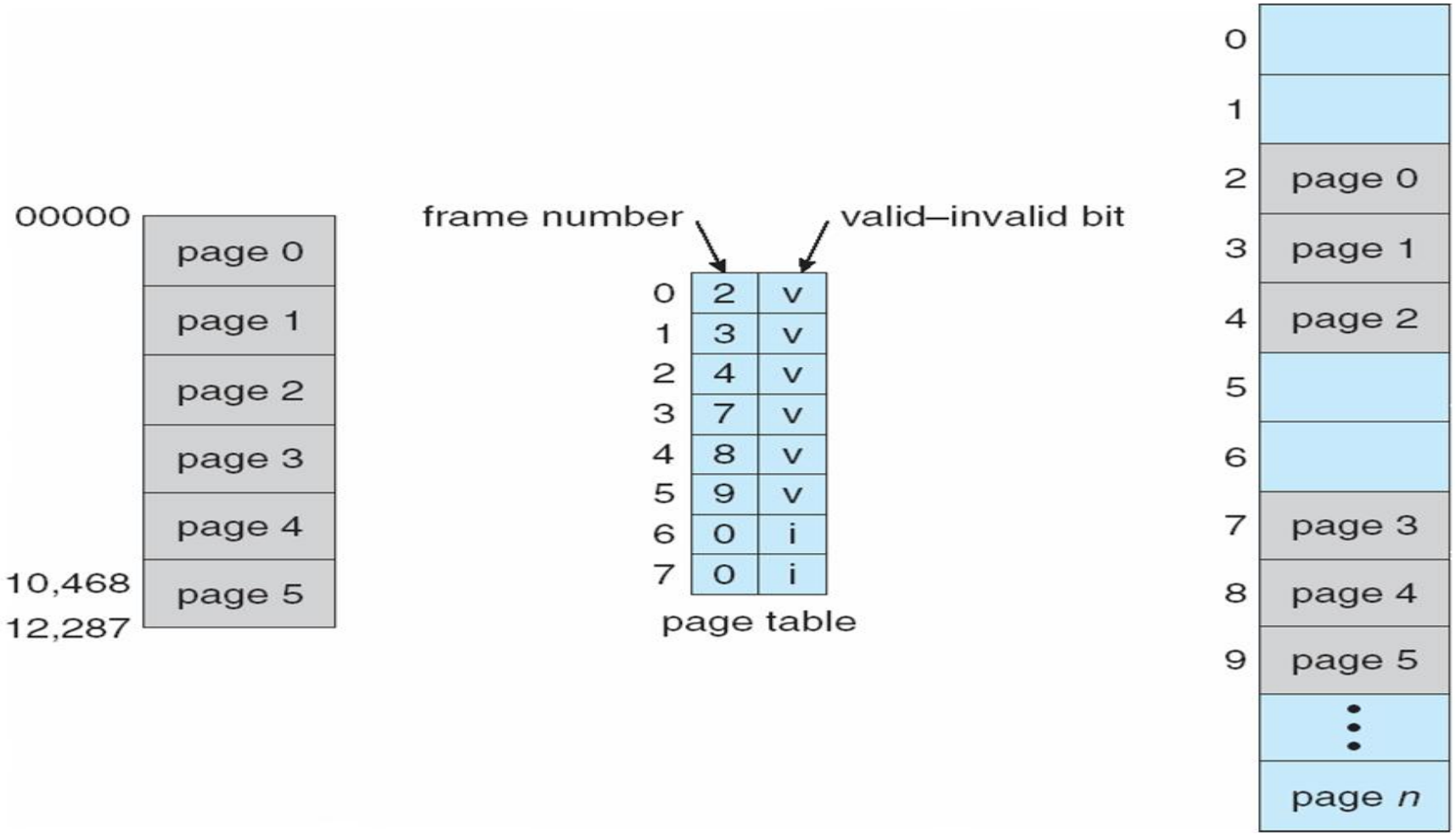
Memory Protection

- Memory protection in a paged environment is implemented by associating protection bit with each frame.

- *Valid-invalid* bit attached to each entry in the page table:

  - "*valid*" indicates that the associated page is in the process' logical address space, and is thus a *legal page*.
  - "*invalid*" indicates that the page is not in the process' logical address space.

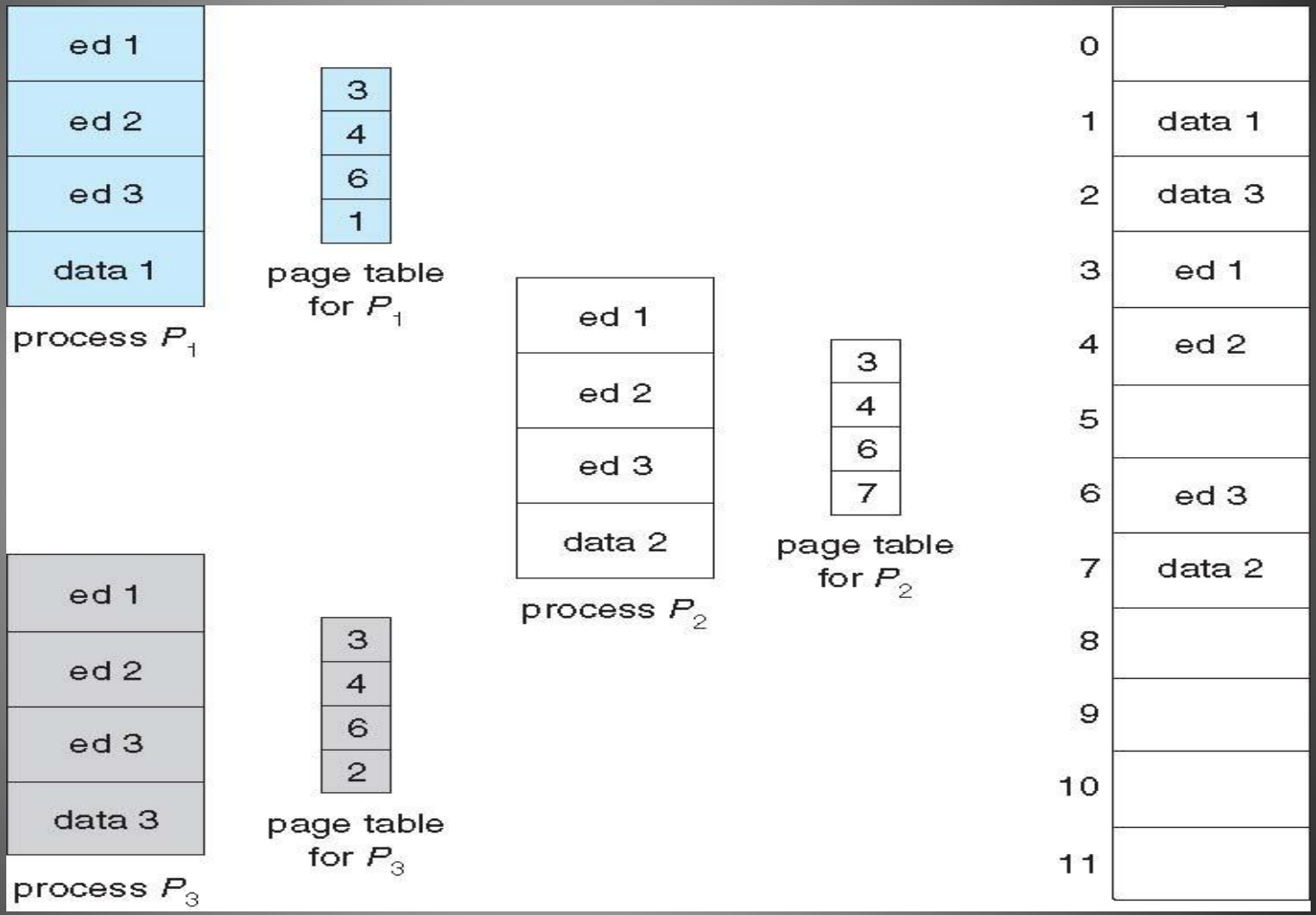
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.
- Private code and data
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.

Shared Pages Example



Problems that still Remain

- In simple paging and segmentation all pages and/or segments must be loaded
  - Limits the number of active processes
  - External fragmentation still possible
  - Swapping is a time consuming process
- Solution: Virtual Memory
  - Load pages only when needed
  - Less need to swap processes out to disk