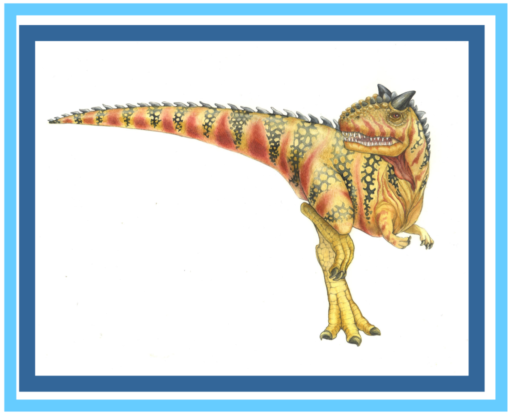
Chapter 7: Main Memory



Chapter 7: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation

Objectives

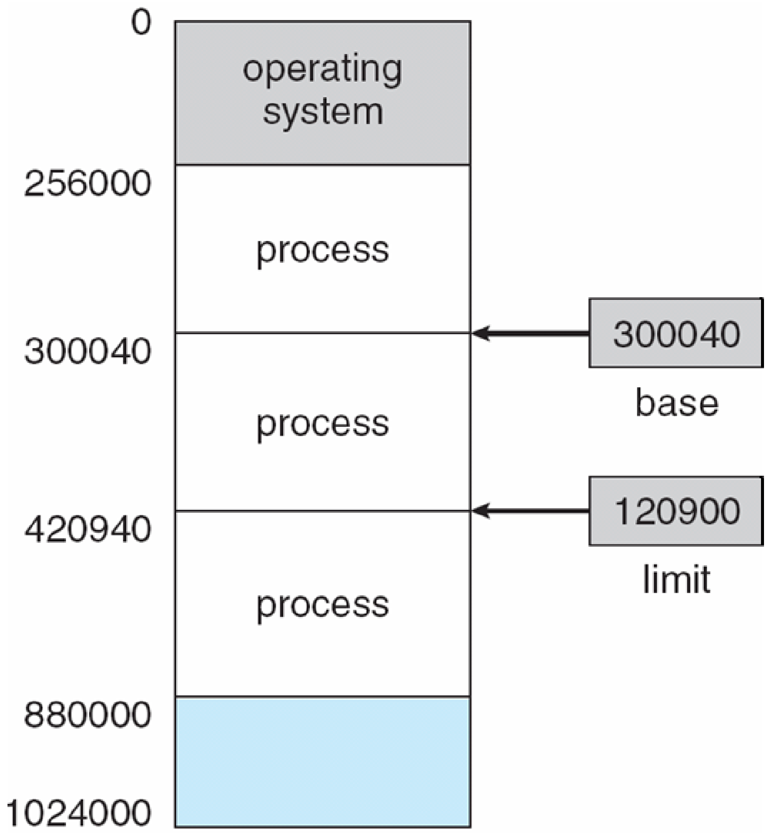
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers and cashes are only storage CPU can access directly
- Register access in one CPU clock cycle (or less)
- Access to main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Base and Limit Registers

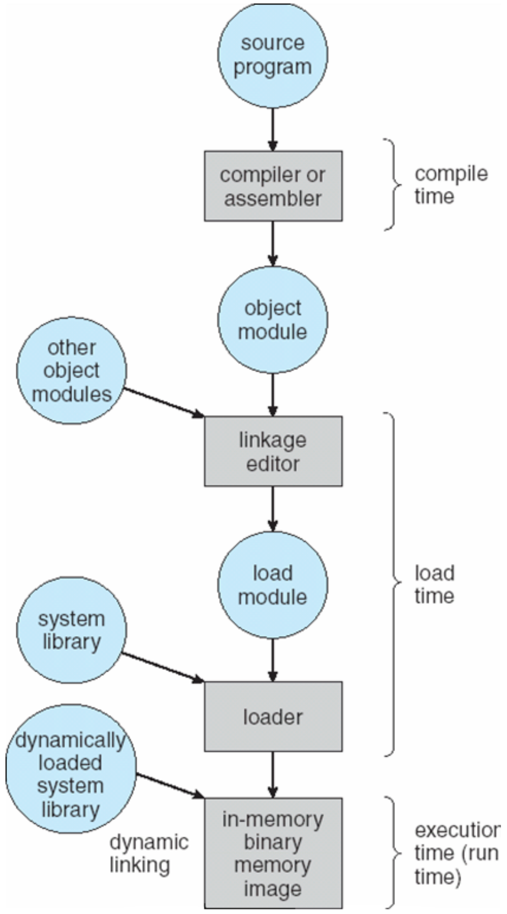
- A pair of **base** and **limit** registers define the logical address space

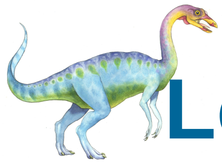


Binding of Instructions and Data to Memory

- 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





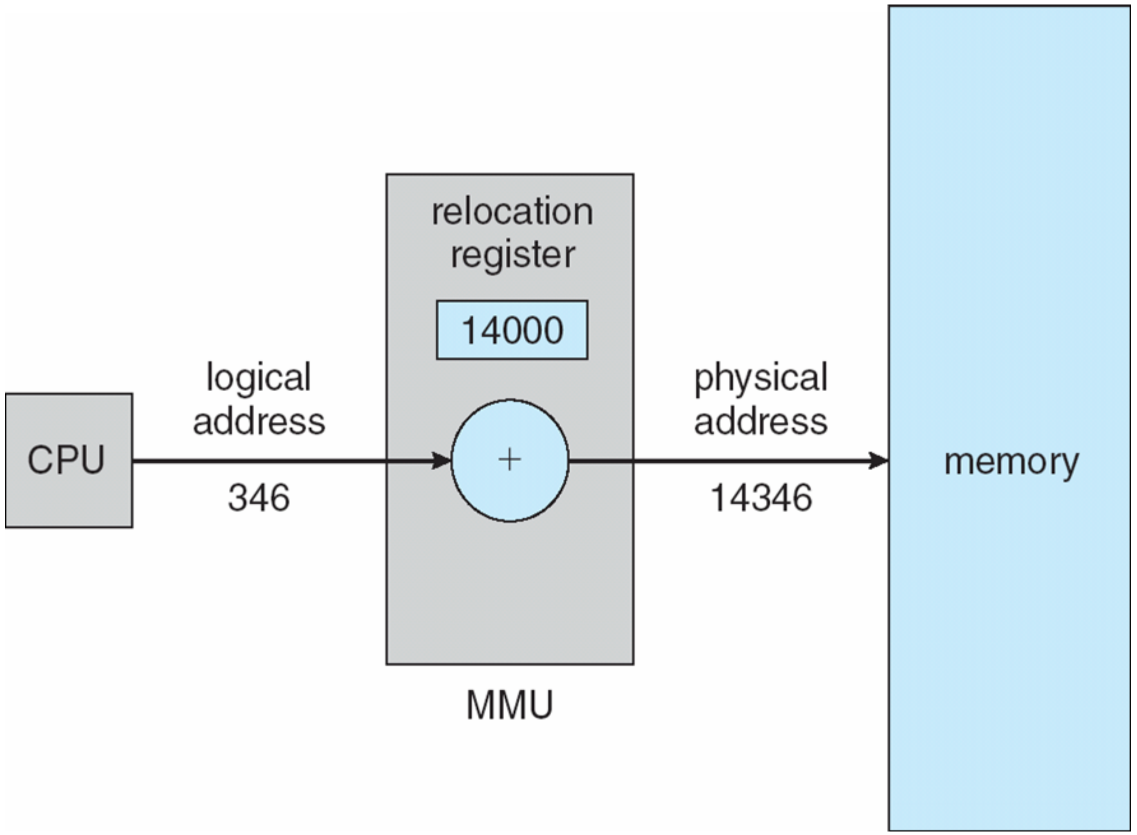
Logical vs. Physical Address Space

- 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;
  - **Physical address** – address seen by the memory management unit

Memory Management Unit (MMU)

- Hardware device that maps logical address to physical address
- In the simplest MMU scheme, the value in the relocation register (usually, the same as base register) is added to every address generated by CPU at the time it is sent to memory
- The user program deals with *logical* addresses because these are the addresses generated by the CPU; the user program never sees the *real* physical addresses

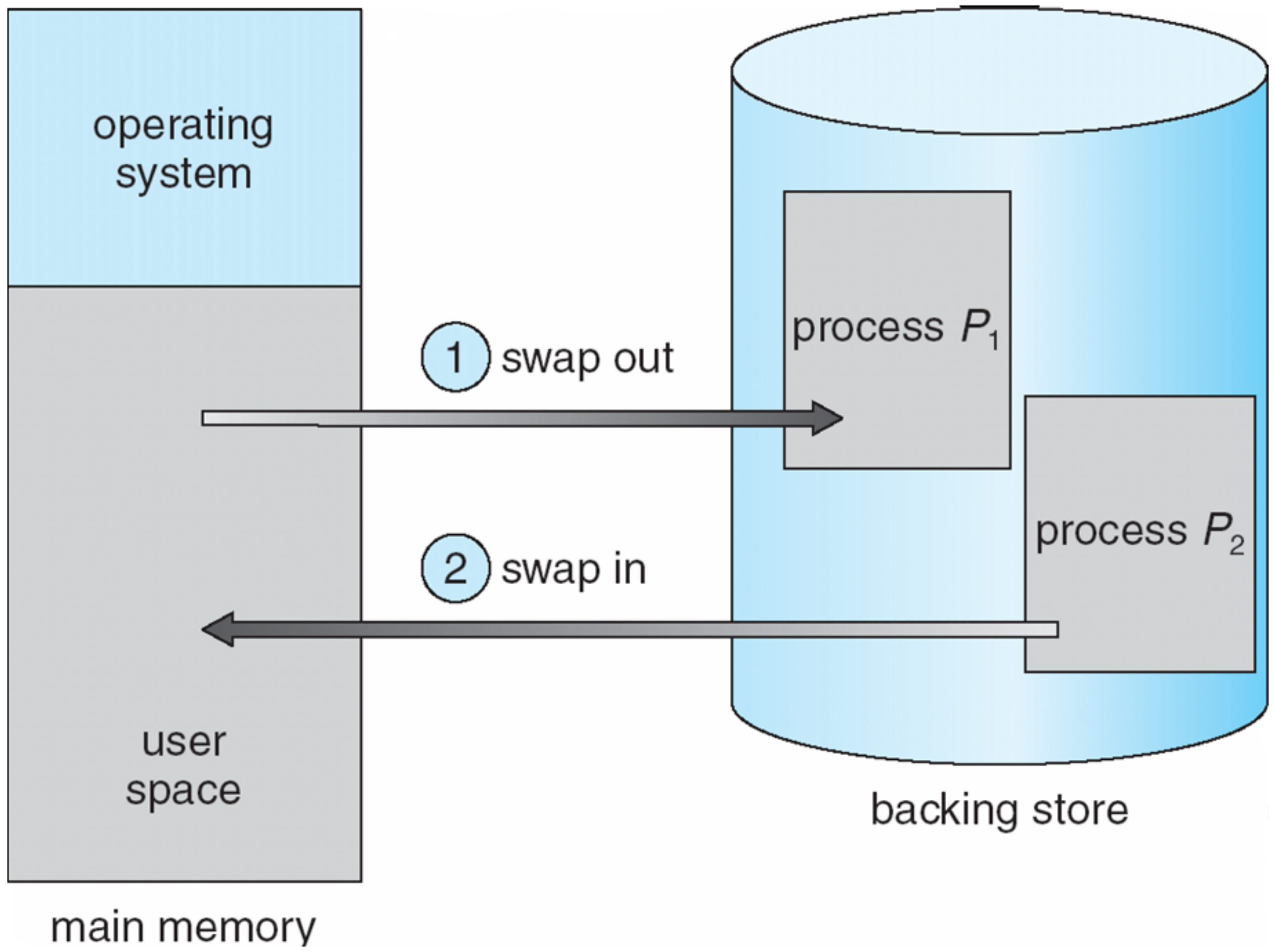
Dynamic relocation using a relocation register



Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **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
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

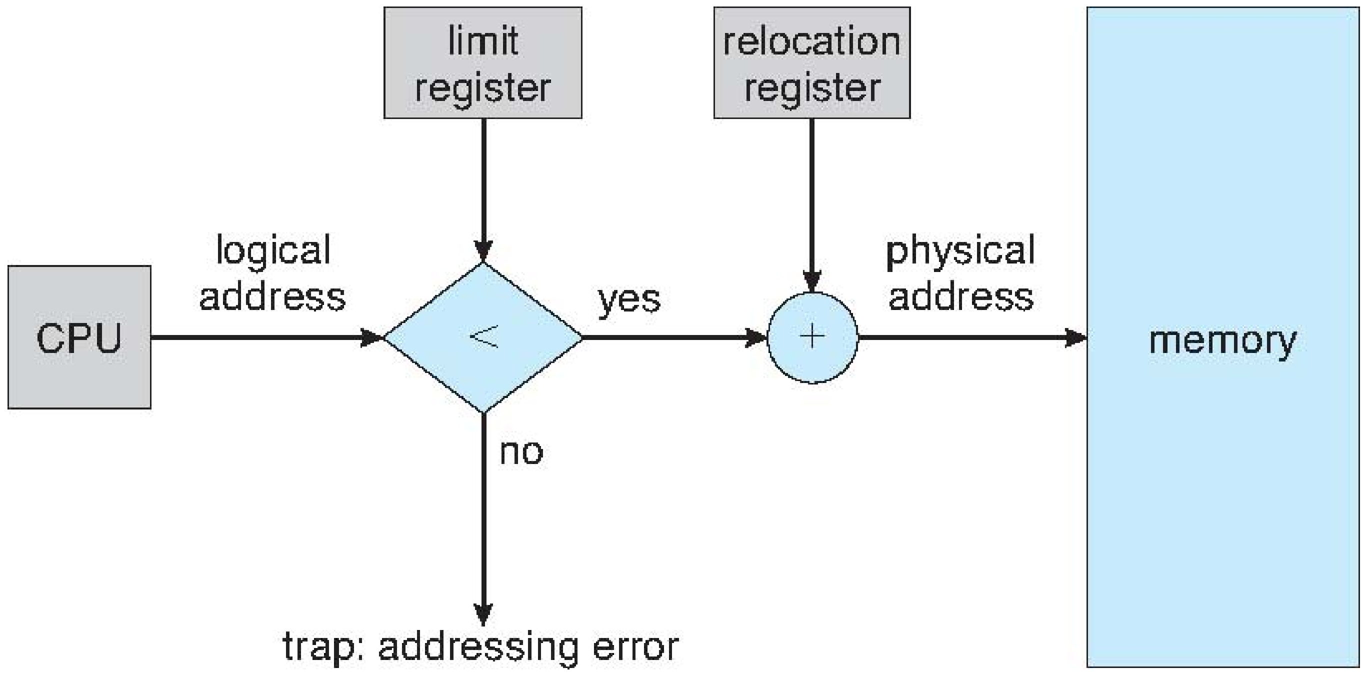
Schematic View of Swapping



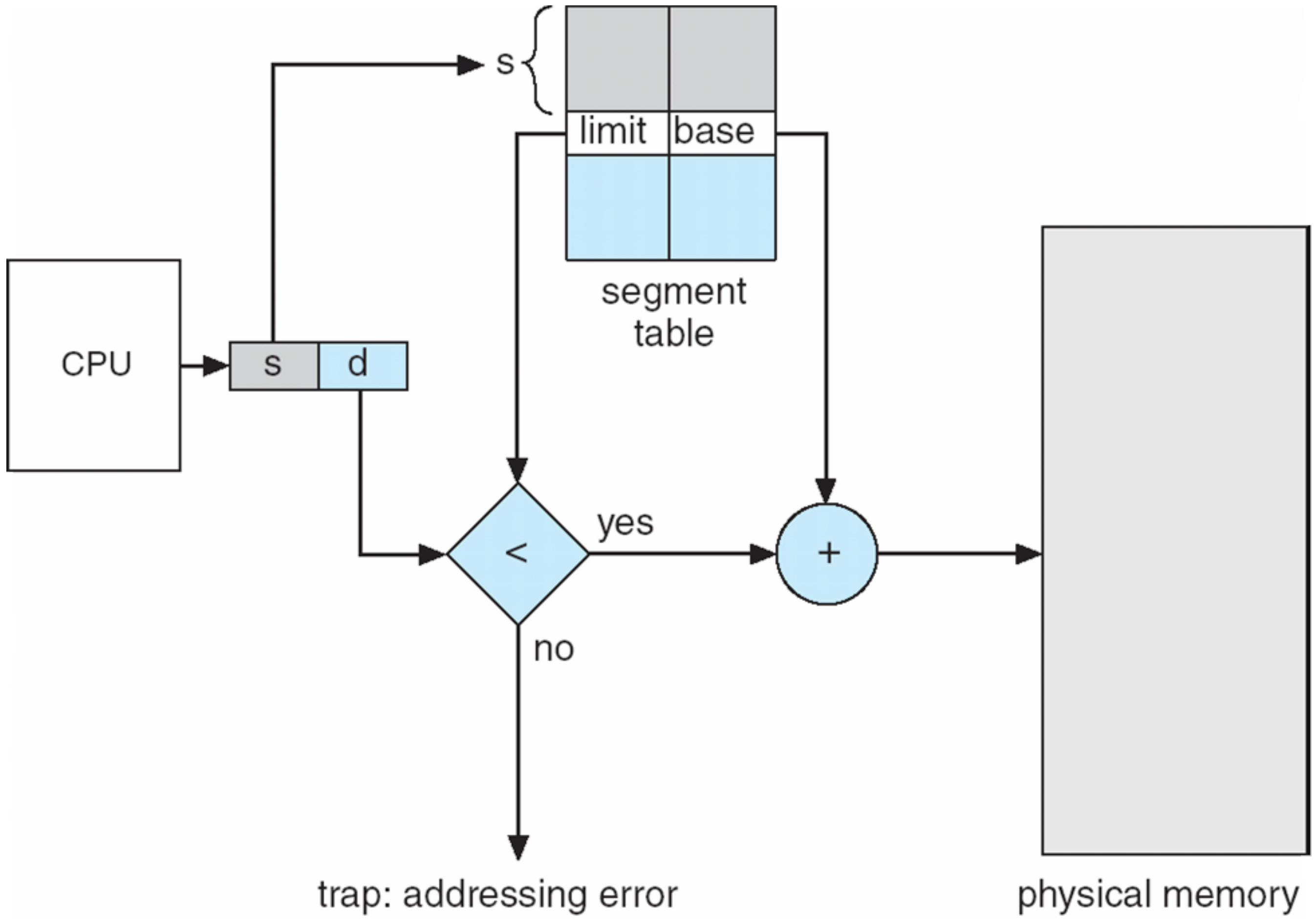
Contiguous Allocation

- Main memory is usually divided into two partitions:
  - Resident operating system, usually held in low memory, with interrupt vector
  - User processes, usually held in high memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of the smallest physical address within the block of memory in question
  - Limit register limits the 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

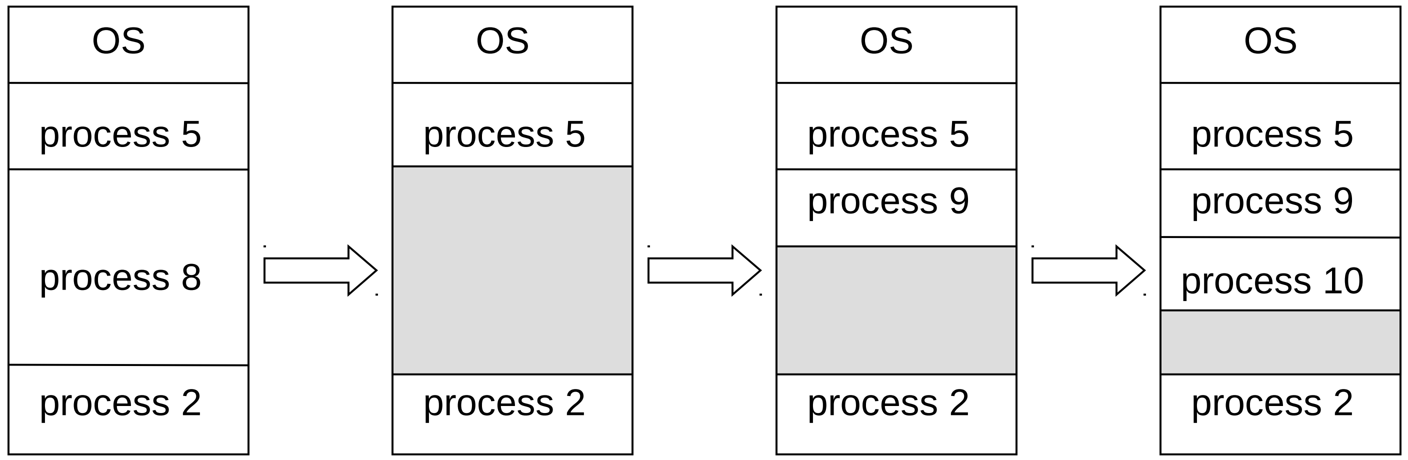


Segmentation Hardware



Contiguous Allocation (Cont)

- Multiple-partition allocation
  - 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

- **First-fit**: Allocate the *first* hole that is big enough
- **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

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous;
- http://csc.csudh.edu/suchenek/CSC341/Paging.pdf
- **Internal Fragmentation** – allocated memory is larger than requested memory;
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is practical *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

Paging

- Physical address space of a process can be noncontiguous; process is allocated a page physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, usually between 512 bytes and 8,192 bytes)
- Divide contiguous 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 into these frames
- Map the (logical) page addresses onto (physical) frame addresses, using a page table to translate logical to physical addresses
- May suffer from internal fragmentation

Address Translation Scheme

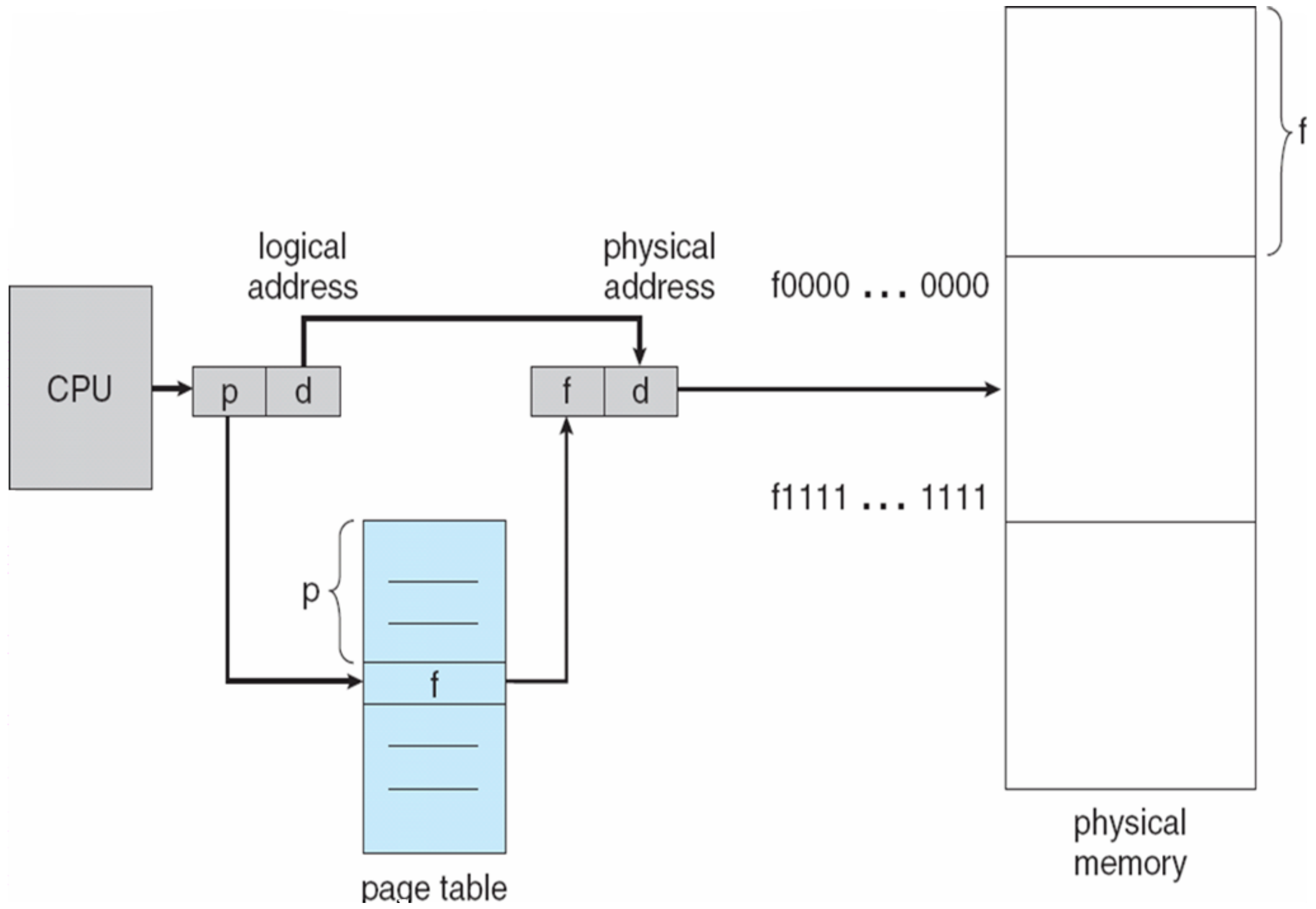
- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

page number	page offset
p	d
$m - n$	n

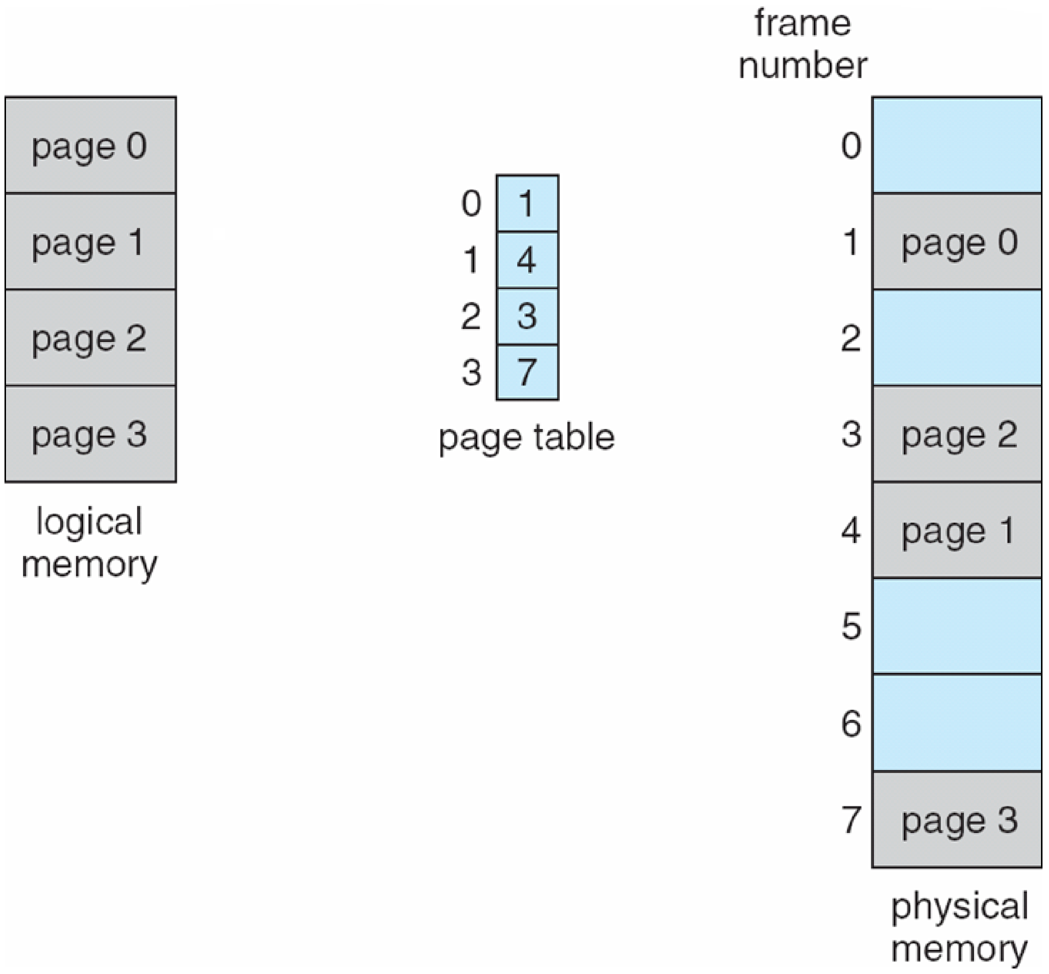
  - For given logical address space 2^m *and page size* 2^n

http://csc.csudh.edu/suchenek/CSC341/Paging.pdf

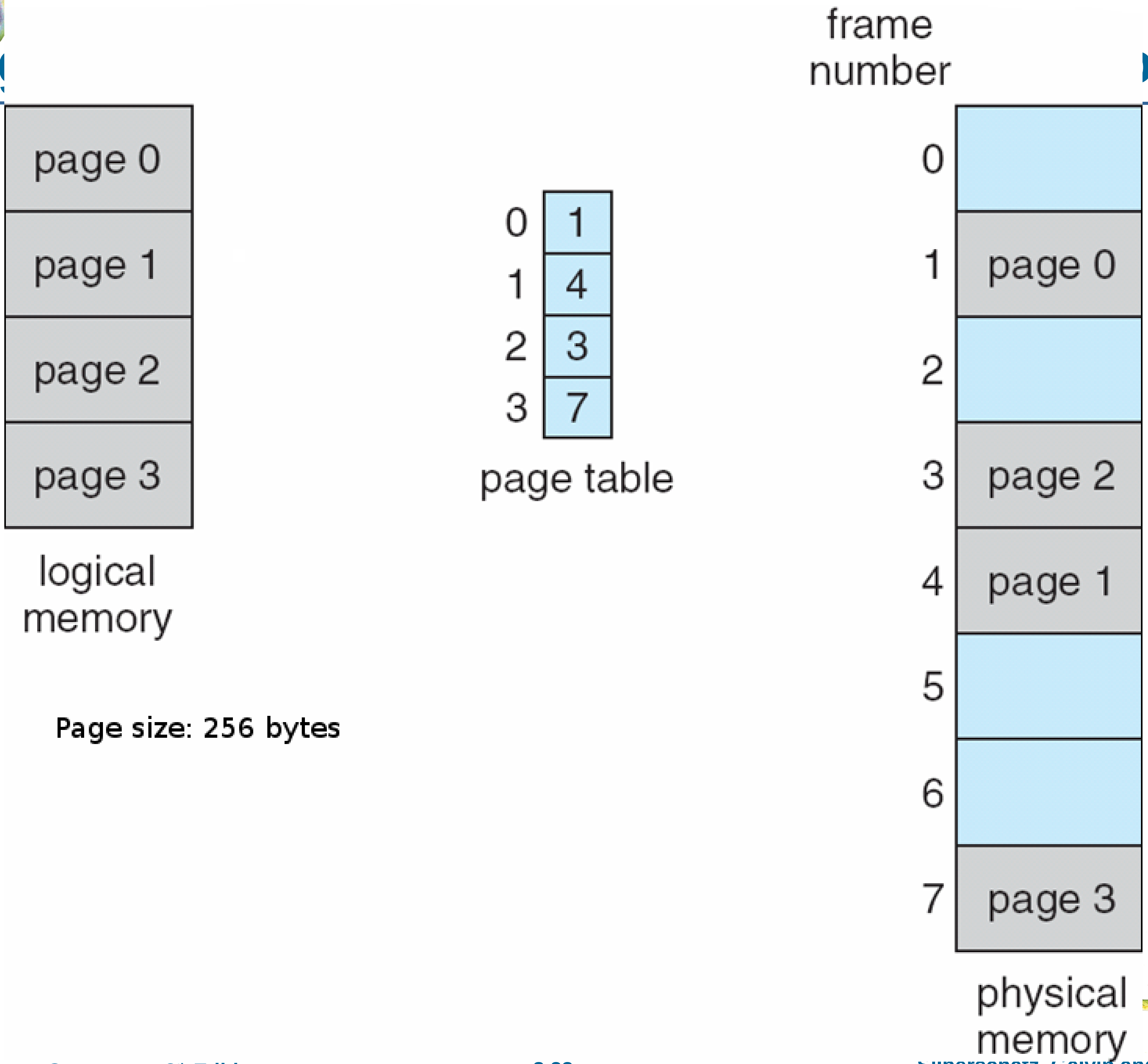
Paging Hardware



Paging Model of Logical and Physical Memory

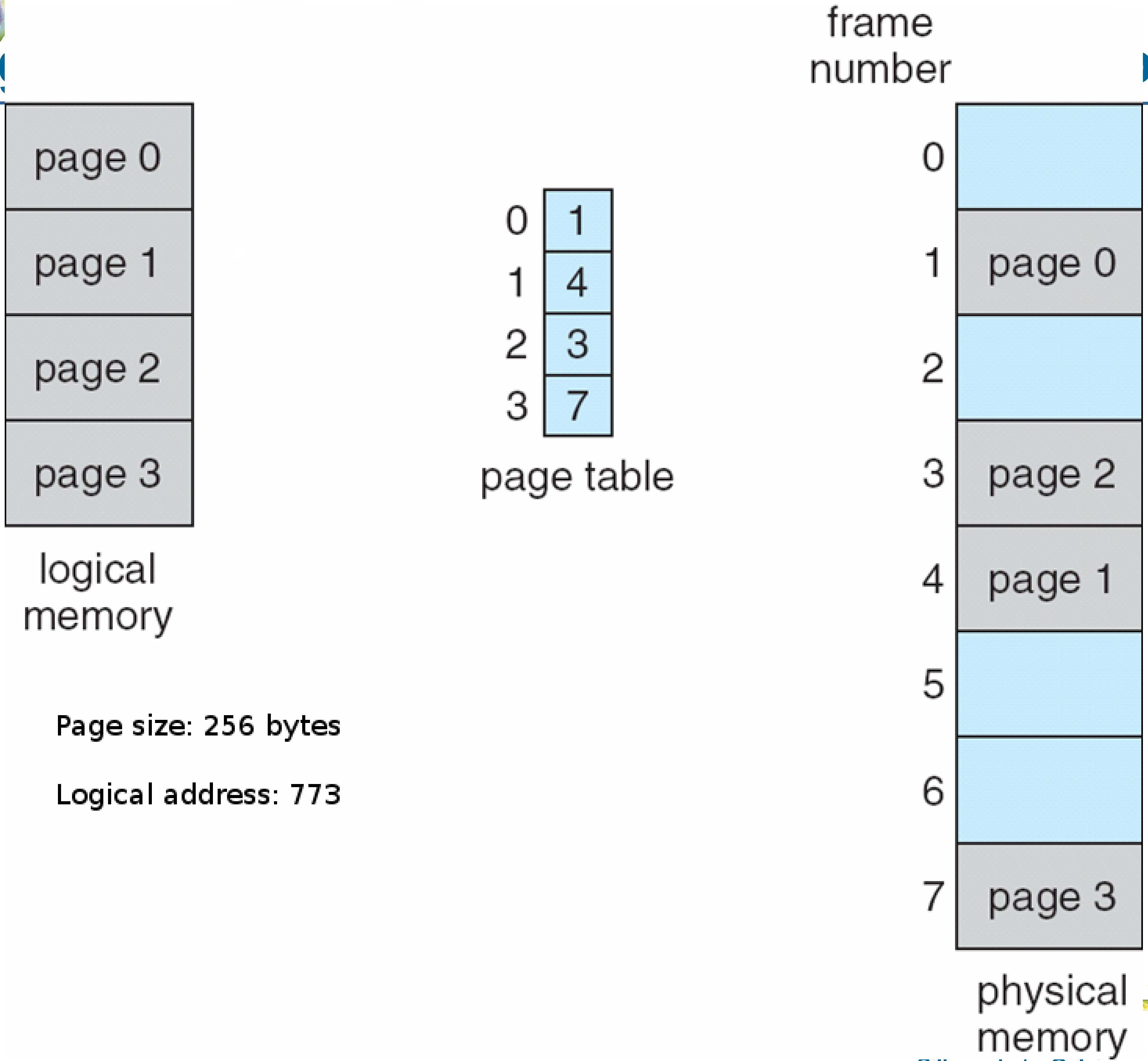


Pa ry



Page size: 256 bytes

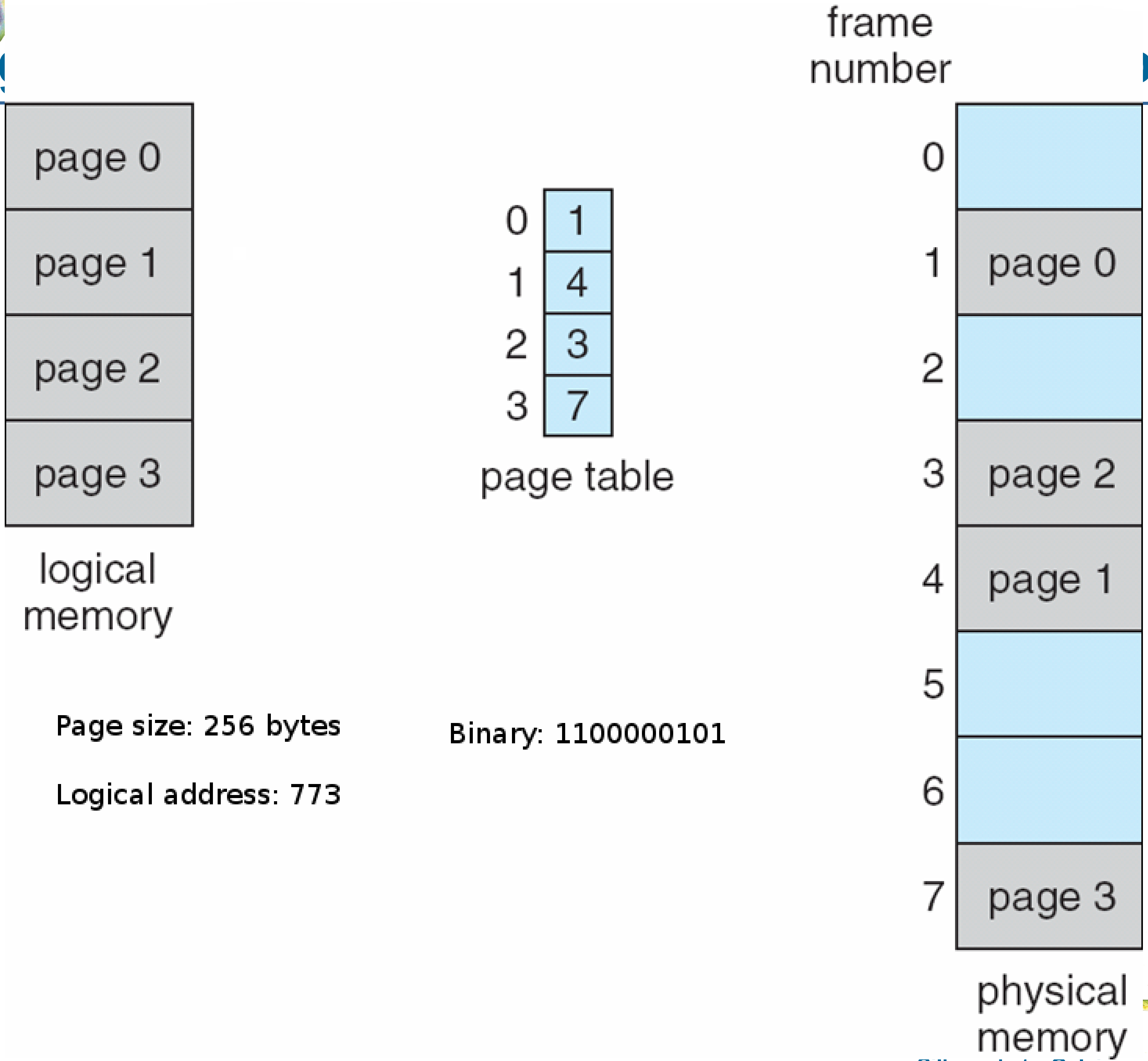
Pa ry



Page size: 256 bytes

Logical address: 773

Pa ry

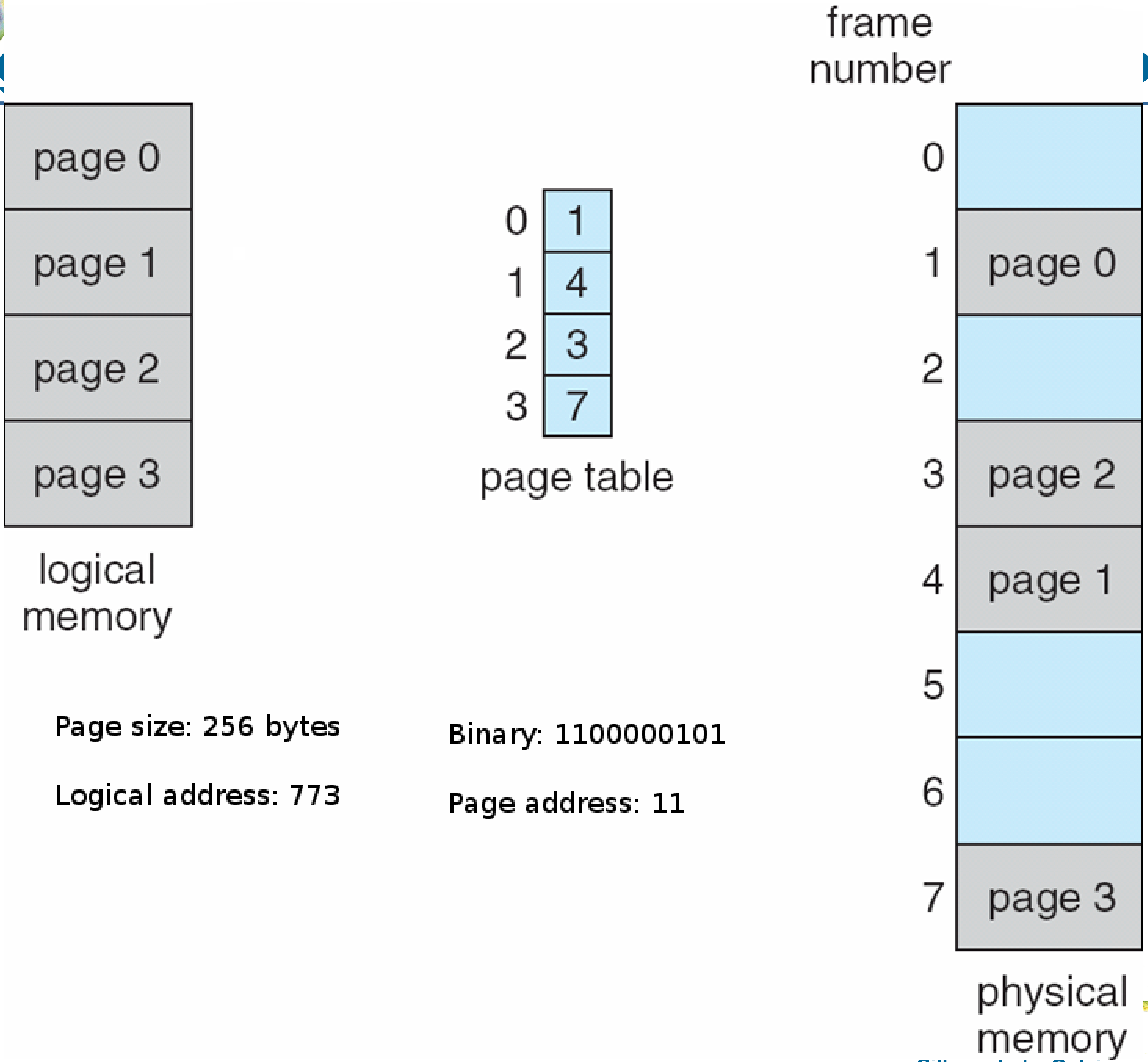


Page size: 256 bytes

Logical address: 773

Binary: 1100000101

Pa ry



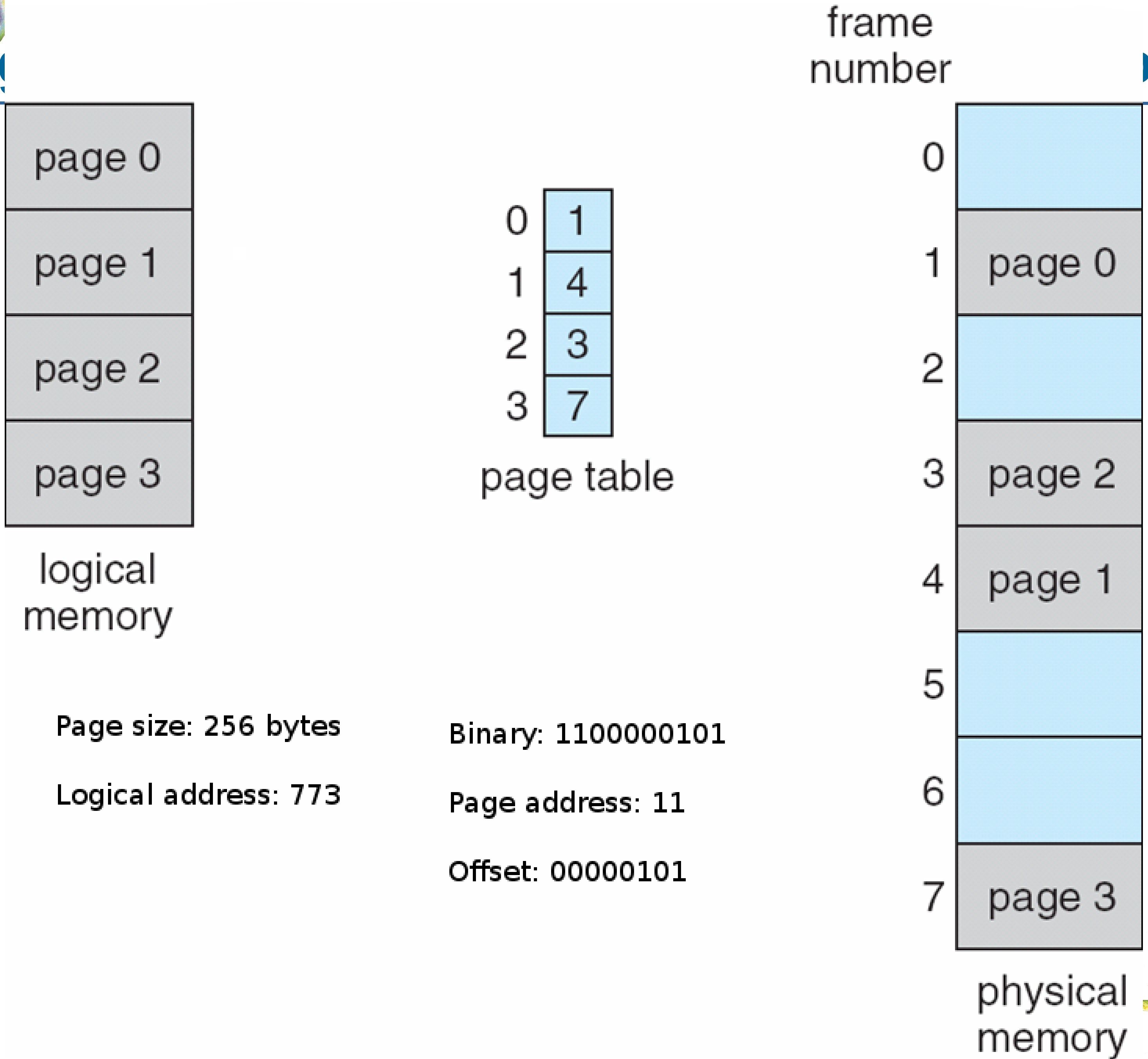
Page size: 256 bytes

Logical address: 773

Binary: 1100000101

Page address: 11

Pa ry



Page size: 256 bytes

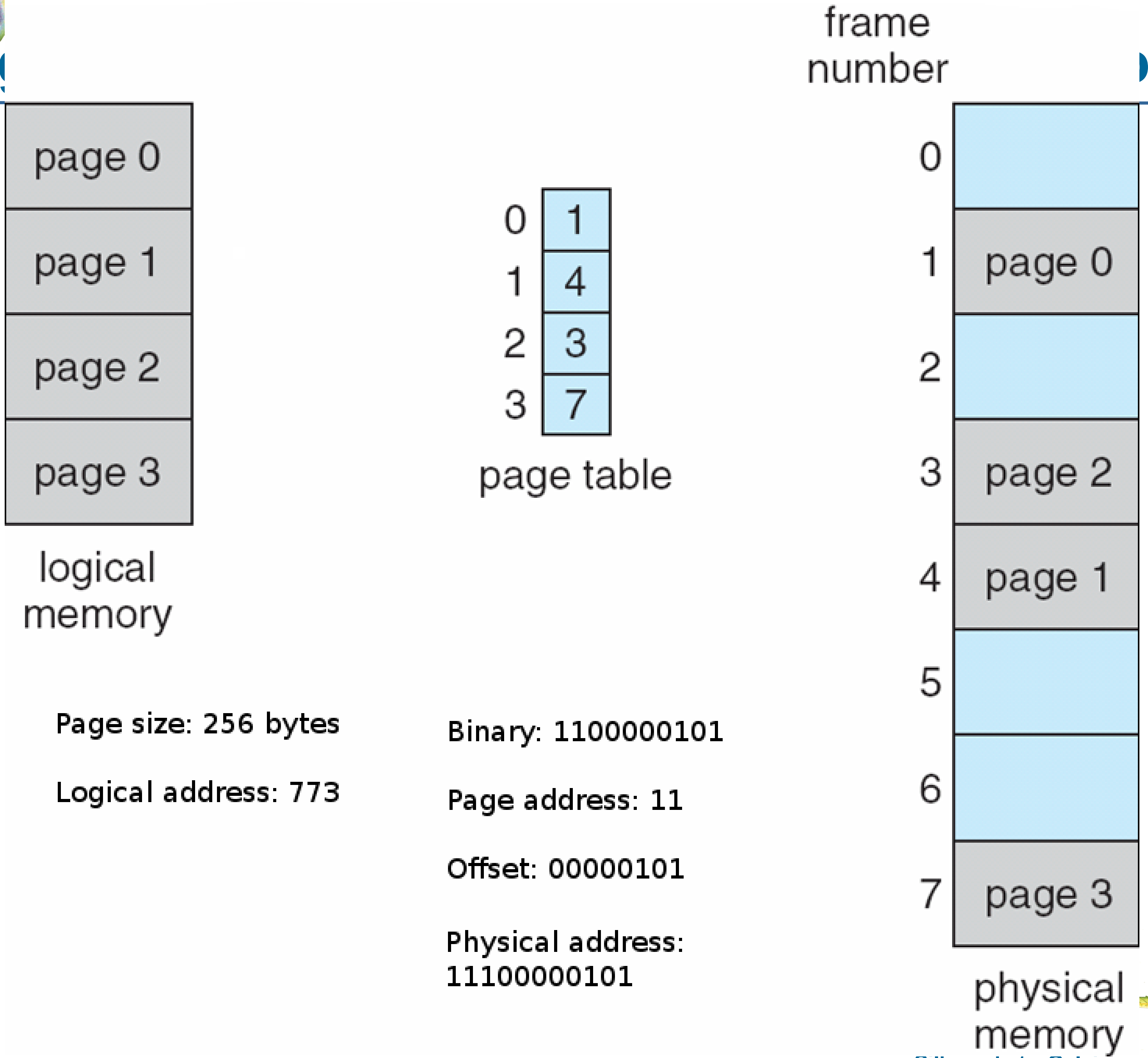
Logical address: 773

Binary: 1100000101

Page address: 11

Offset: 00000101

Pa ry



Page size: 256 bytes

Logical address: 773

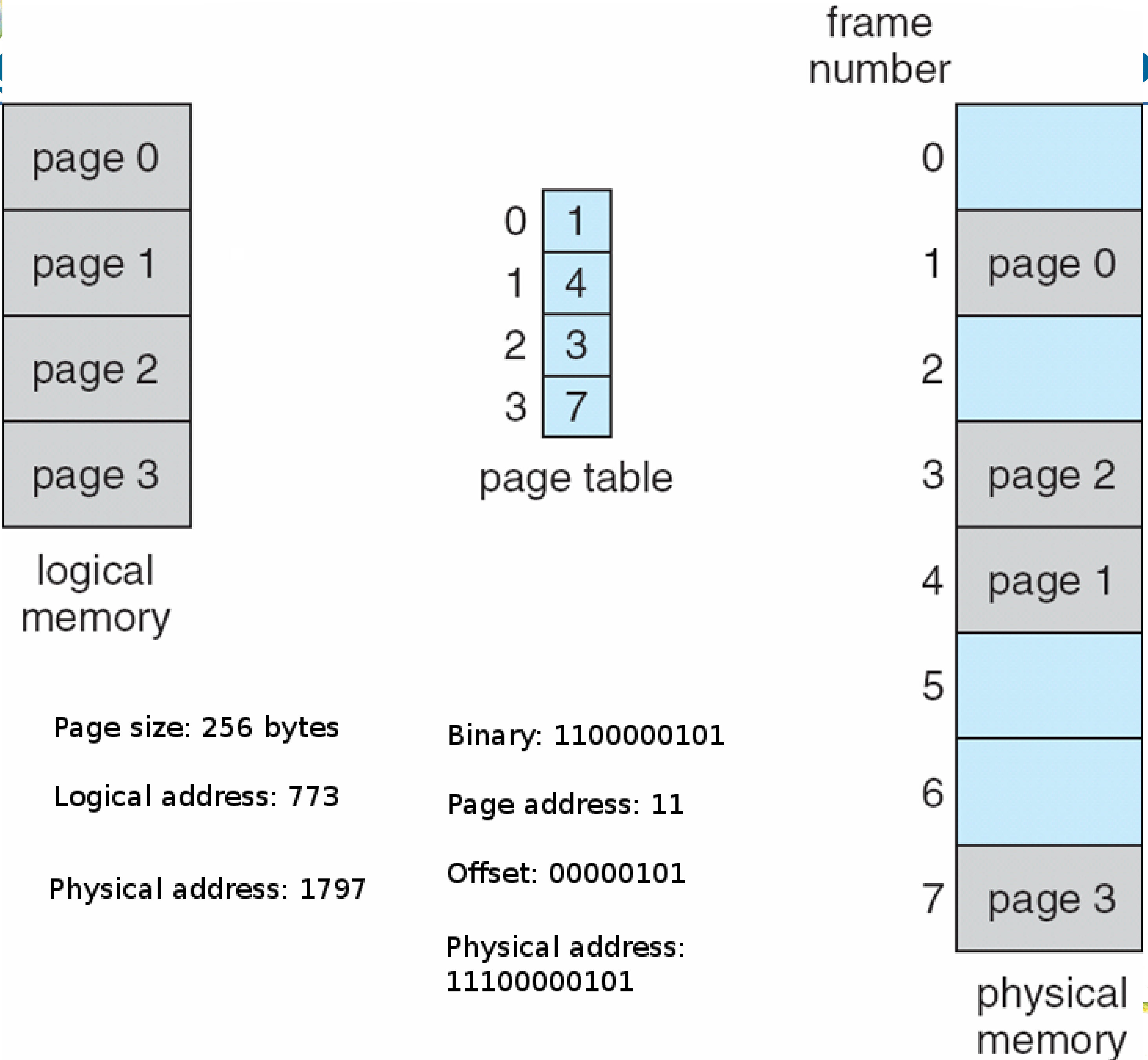
Binary: 1100000101

Page address: 11

Offset: 00000101

Physical address:
11100000101

Pa ry



Page size: 256 bytes

Logical address: 773

Physical address: 1797

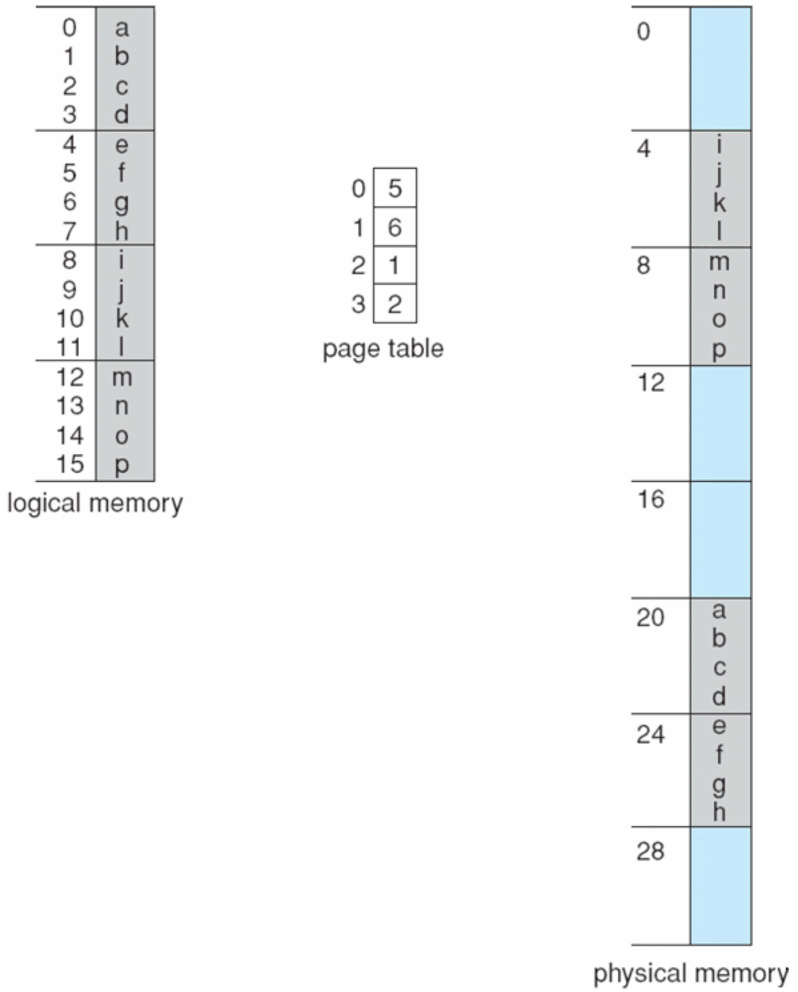
Binary: 1100000101

Page address: 11

Offset: 00000101

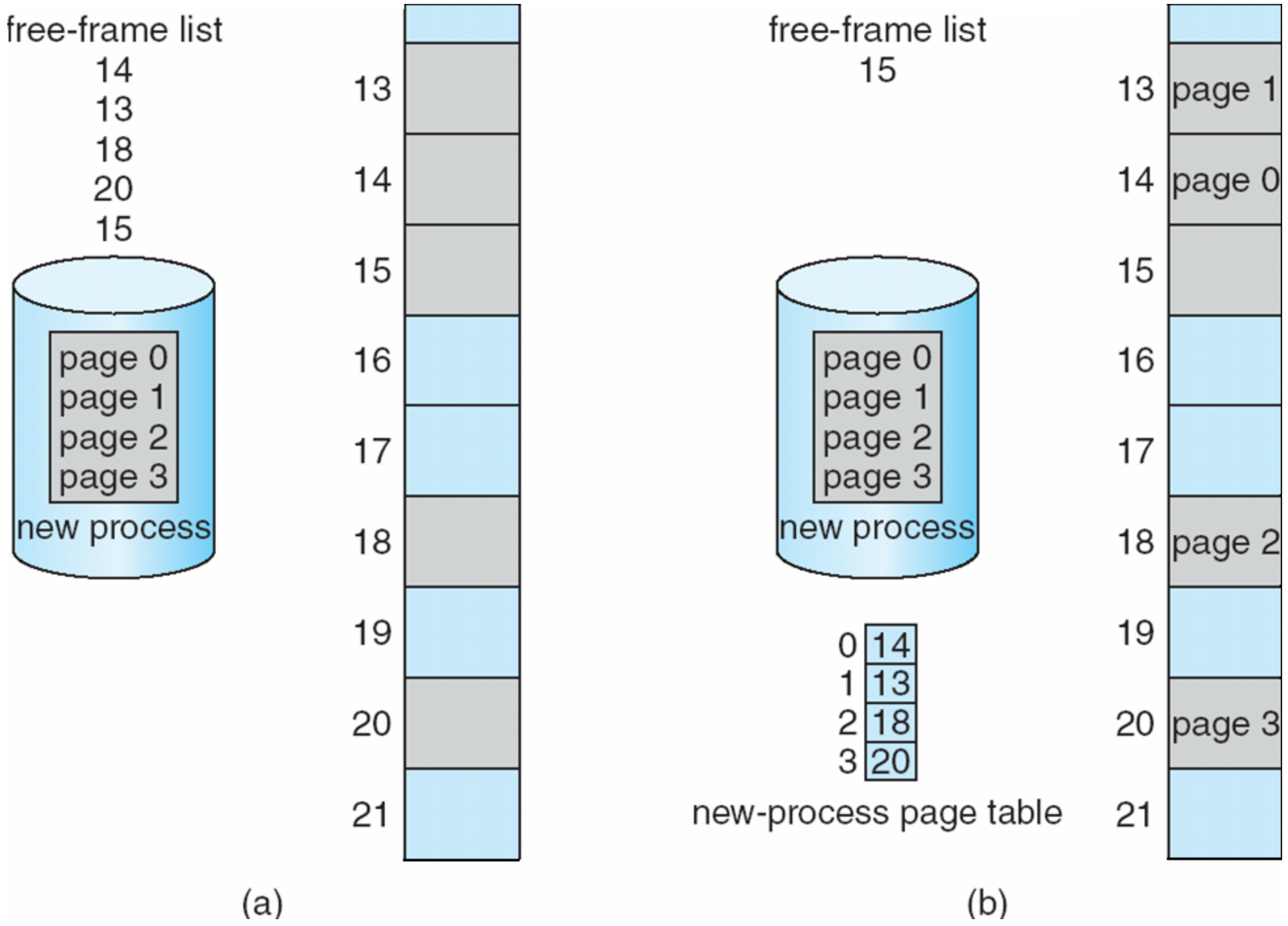
Physical address:
11100000101

Paging Example



32-byte memory and 4-byte pages

Free Frames



Before allocation

After allocation

Implementation of Page Table

- **Case 1:** Page table is in hardware page table
- It consists of a set of high-speed registers
- It is impractical if the page table is large (say, larger than a few hundred entries)

Implementation of Page Table 2

- Case 2: Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** 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 memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Associative Memory

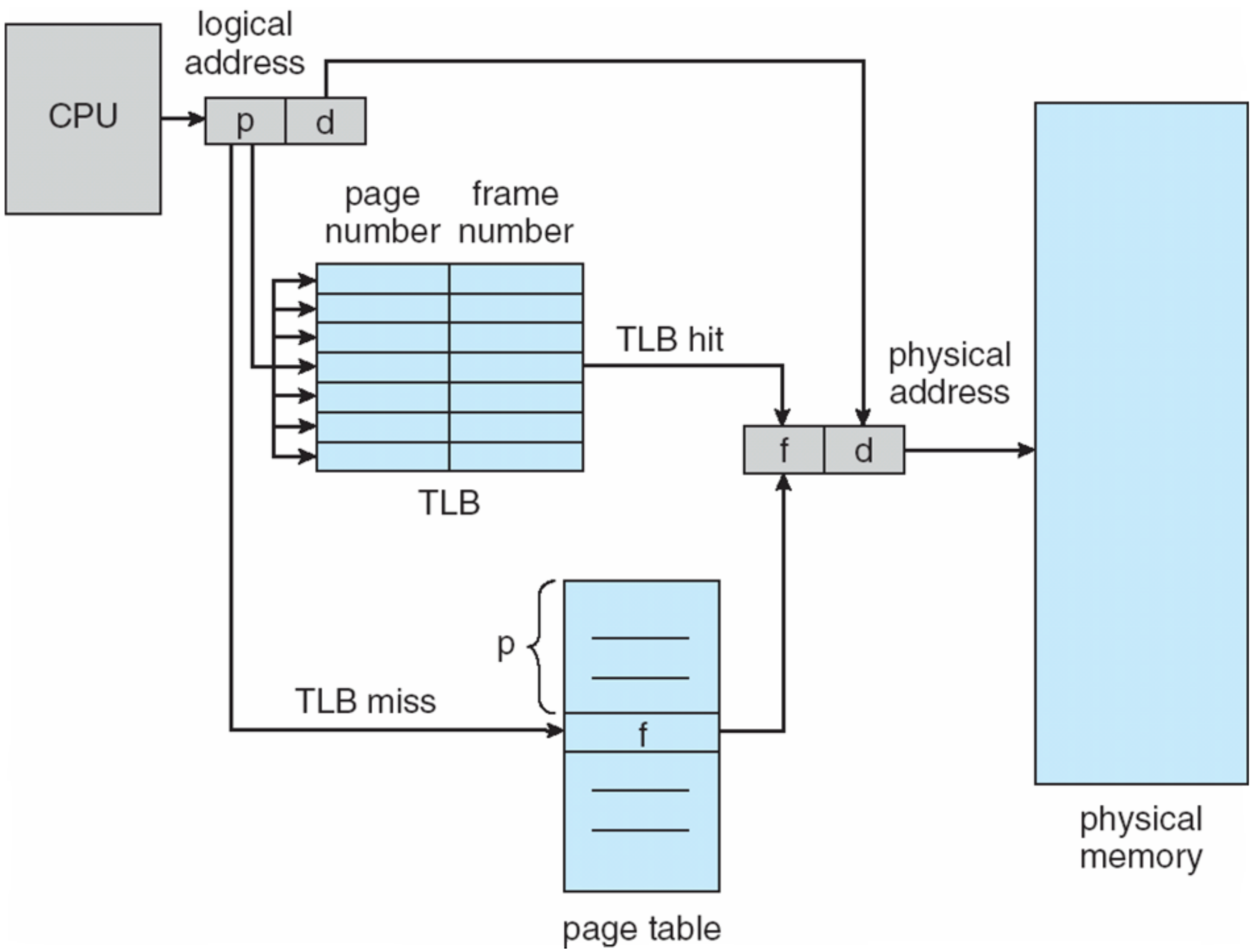
- Associative memory – parallel search (relative to the normal access)

Page #	Frame #

Address translation (p, d)

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

Paging Hardware With TLB



Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 time unit
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α

Effective Access Time (EAT)

$$\begin{aligned} EAT &= (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

More accurate formula:

$$\begin{aligned} EAT &= (1 + \varepsilon)\,\alpha + 2(1 - \alpha) \\ &= 2 + \varepsilon\alpha - \alpha \end{aligned}$$

Effective Access Time

Example

- Associative Lookup α = 0.05 time unit (= 10 ns)
- Assume time unit is 190 ns
- Hit ratio $\alpha = .99$

EAT = 190ns (2 + $\varepsilon\alpha - \alpha$) = 190ns (2 + 0.0495 -.99) =

= 201.305ns $\approx$ 200ns

Effective Access Time

Example

- Associative Lookup α = 0.05 time unit (= 10 ns)
- Assume time unit is 190 ns
- Hit ratio $\alpha = .99$

EAT = 190ns (2 + $\varepsilon\alpha - \alpha) =$ 190ns (2 + 0.0495 -.99) $=$

$=$ 201.305ns $\approx$ 200ns

We will use the above approximation in Chapter 8, Virtual Memory.

Memory Protection

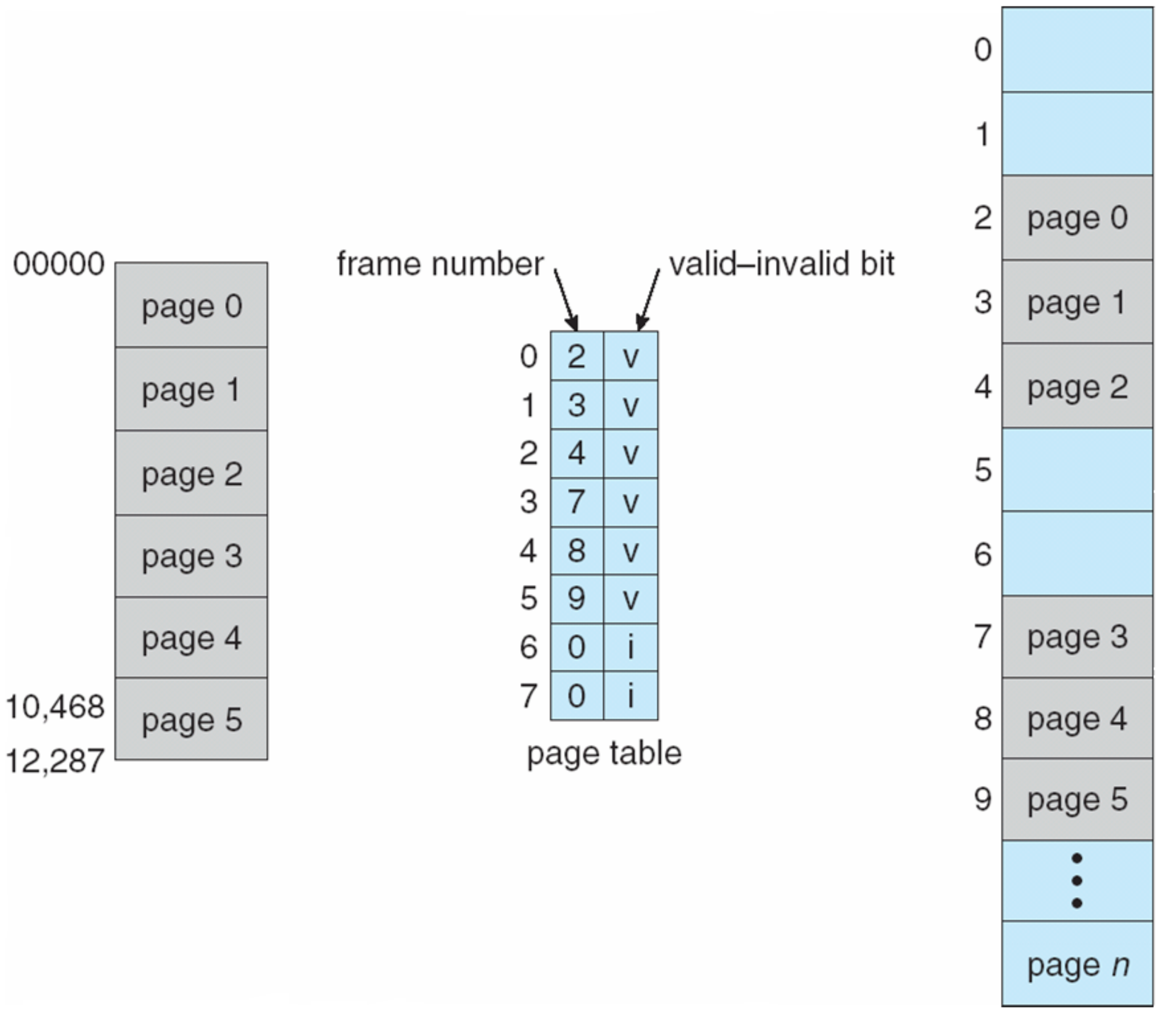
- Memory protection implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space

Memory Protection

- Memory protection 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

- **Note: In the next Chapter 8, Virtual Memory, the meaning of the valid-invalid bit is slightly changed. The Final will cover that other meaning, and NOT this one.**

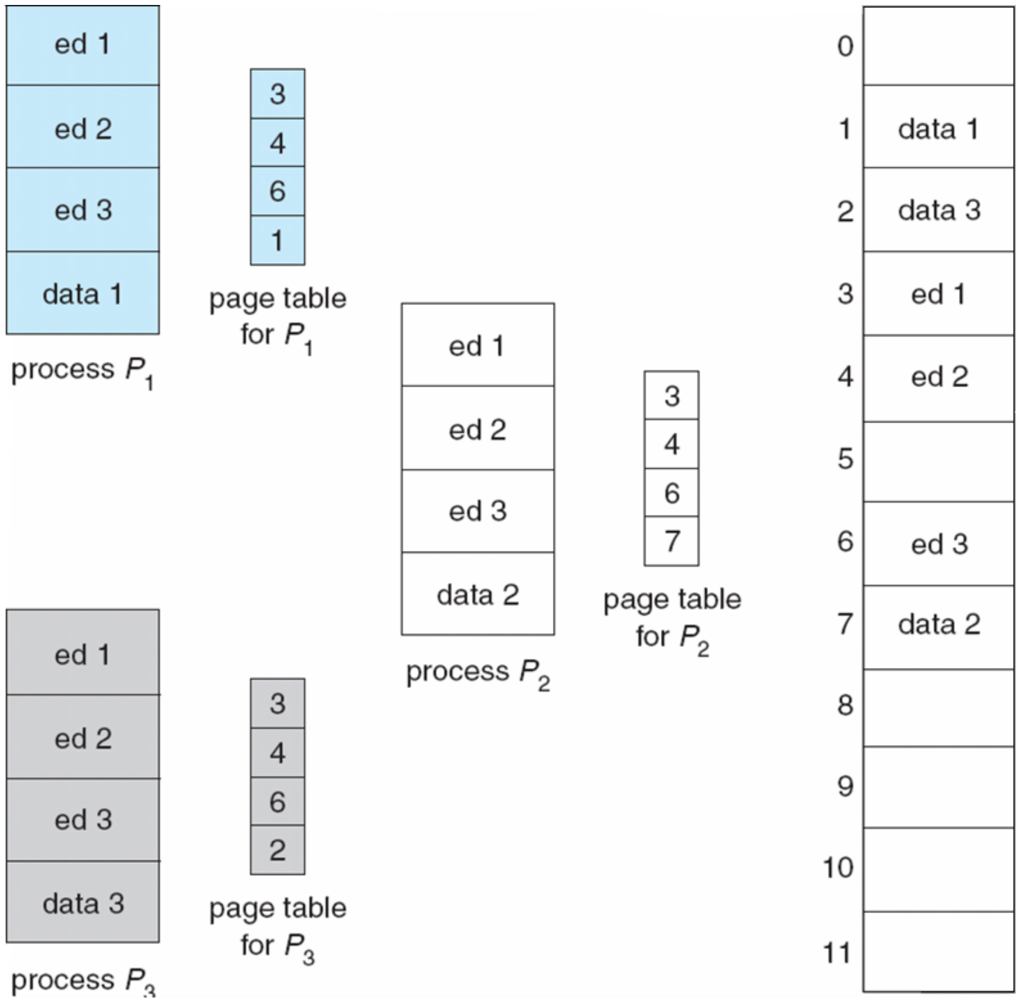
Valid (v) or Invalid (i) Bit In A Page Table



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



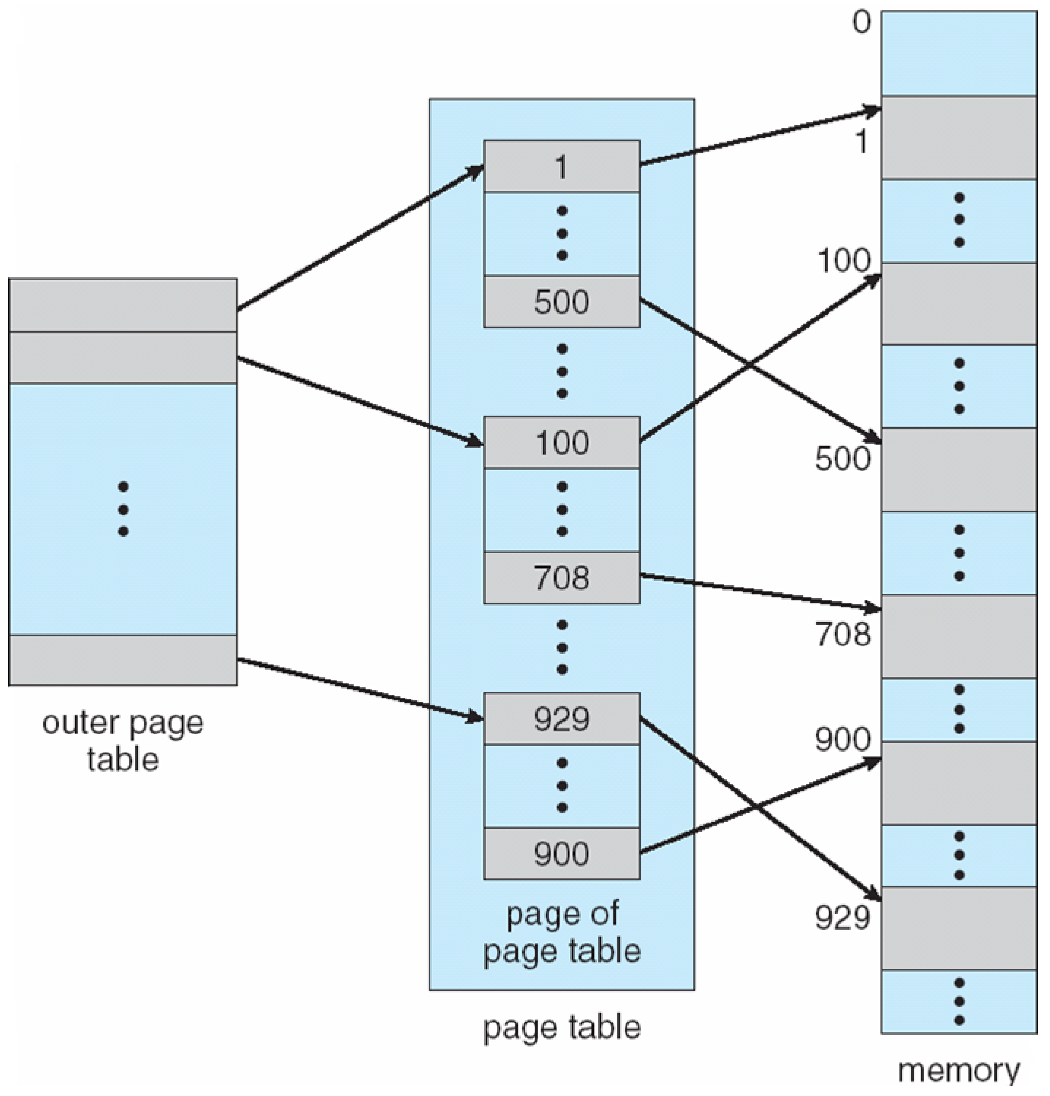
Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

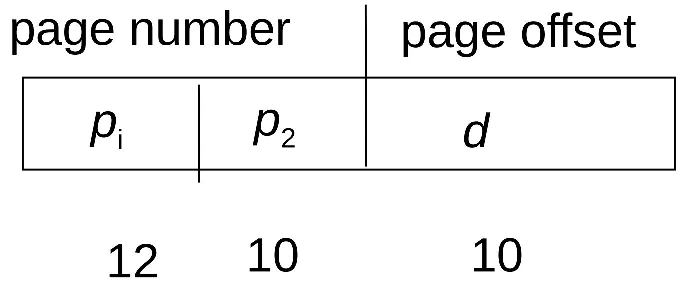
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

Two-Level Page-Table Scheme



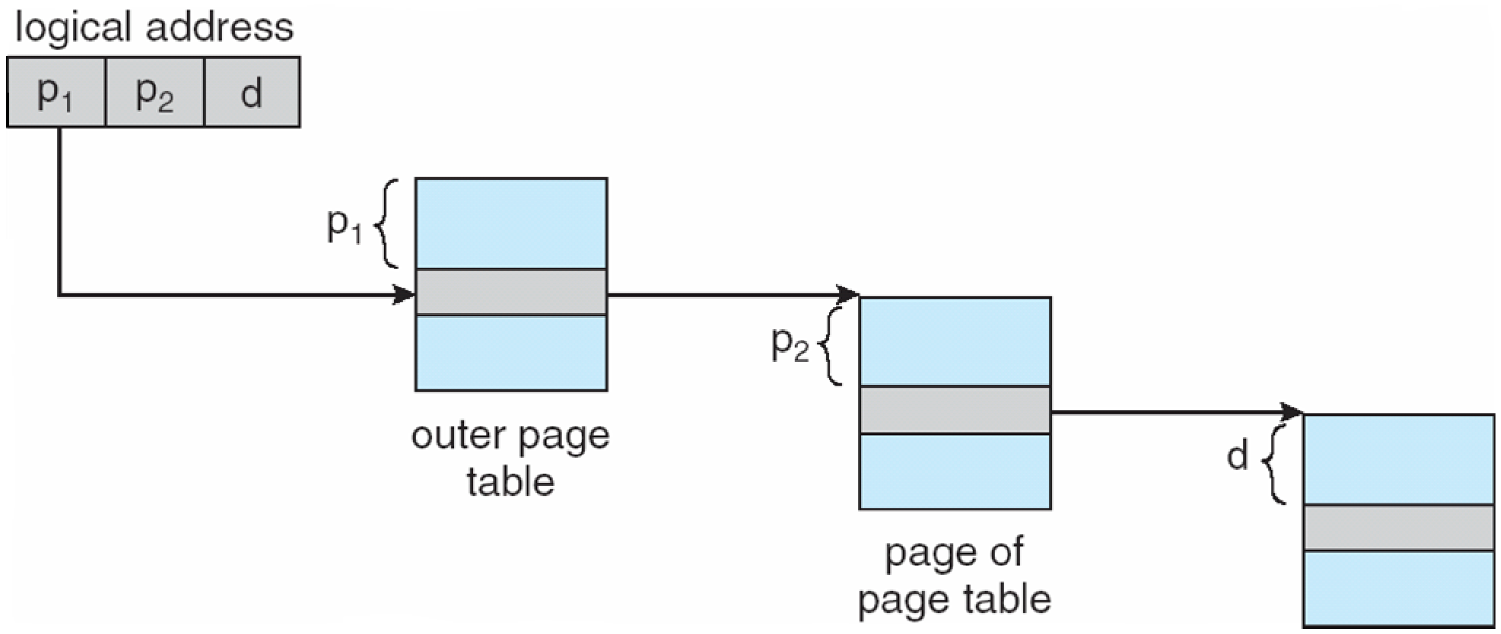
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset (a page number in the inner table)
- Thus, a logical address is as follows:



where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Address-Translation Scheme



Three-level Paging Scheme

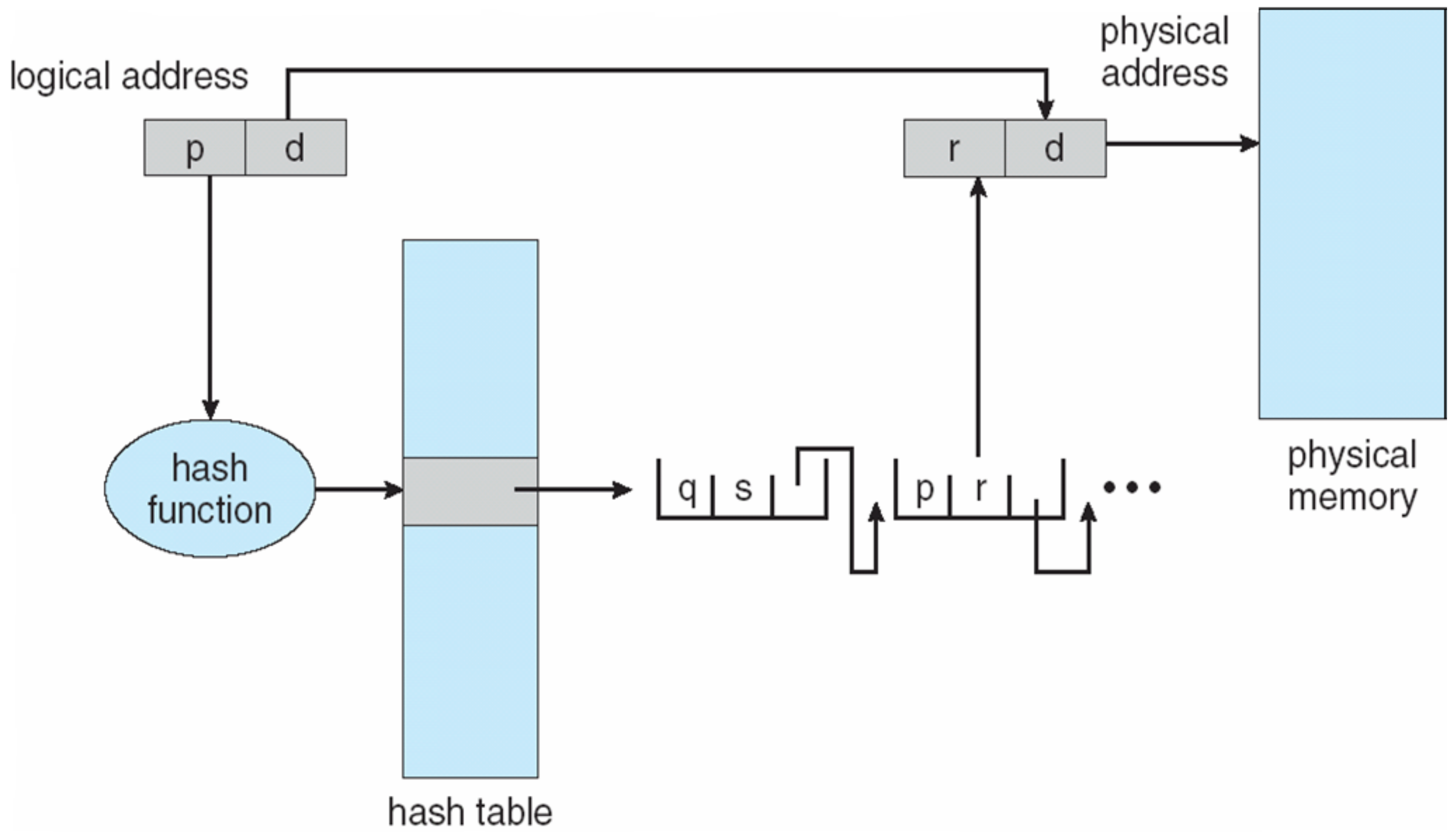
outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

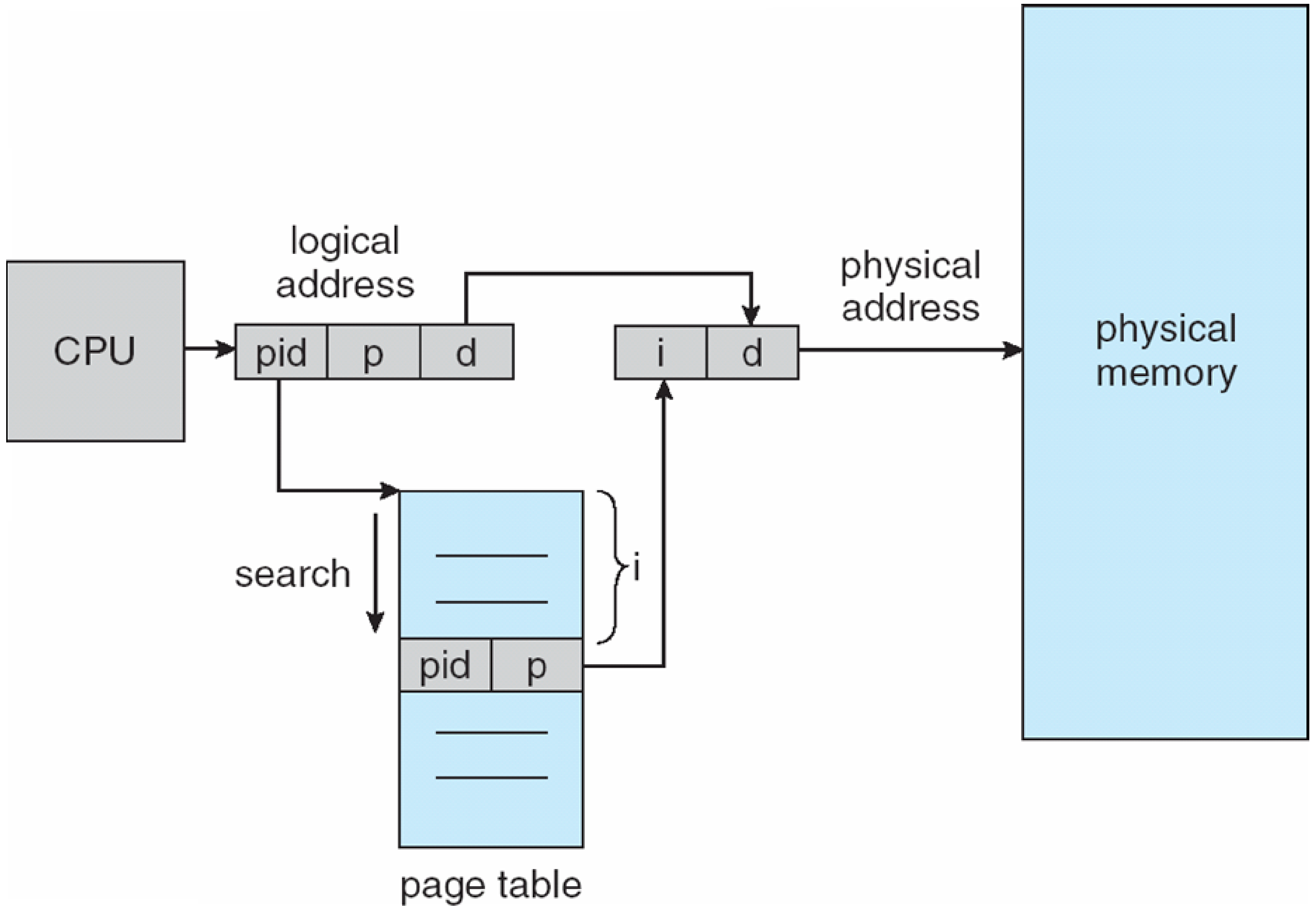
Hashed Page Table



Inverted Page Table

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

Inverted Page Table Architecture



Segmentation

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

    main program

    procedure

    function

    method

    object

    local variables, global variables

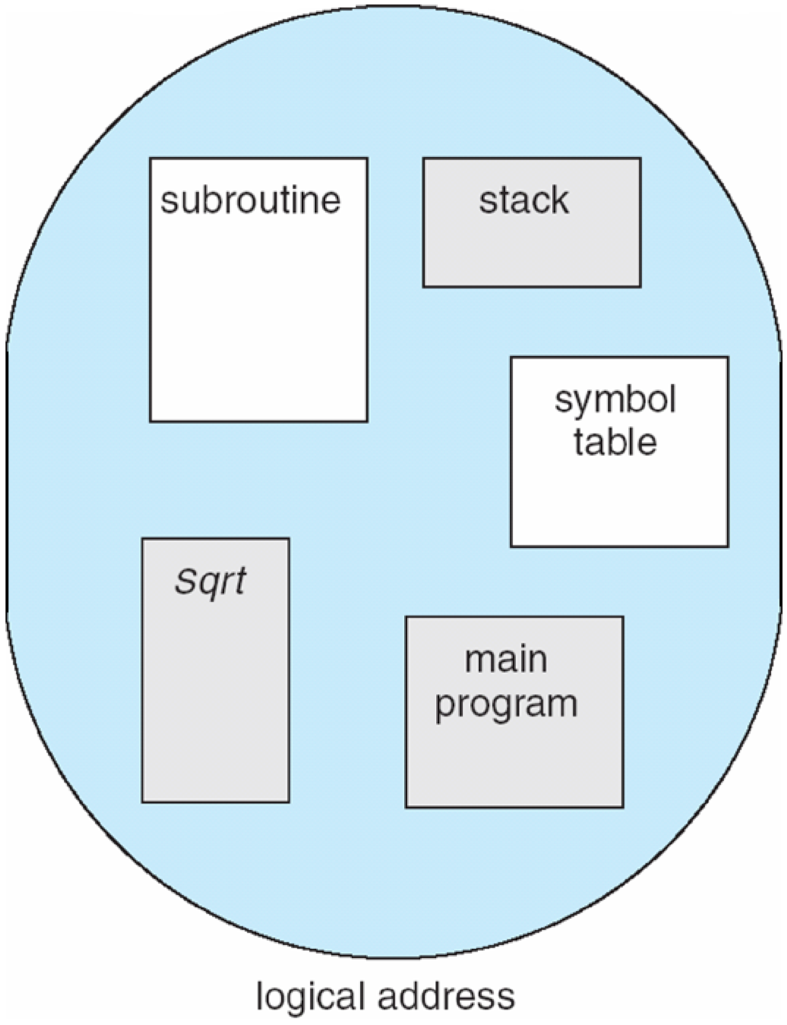
    common block

    stack

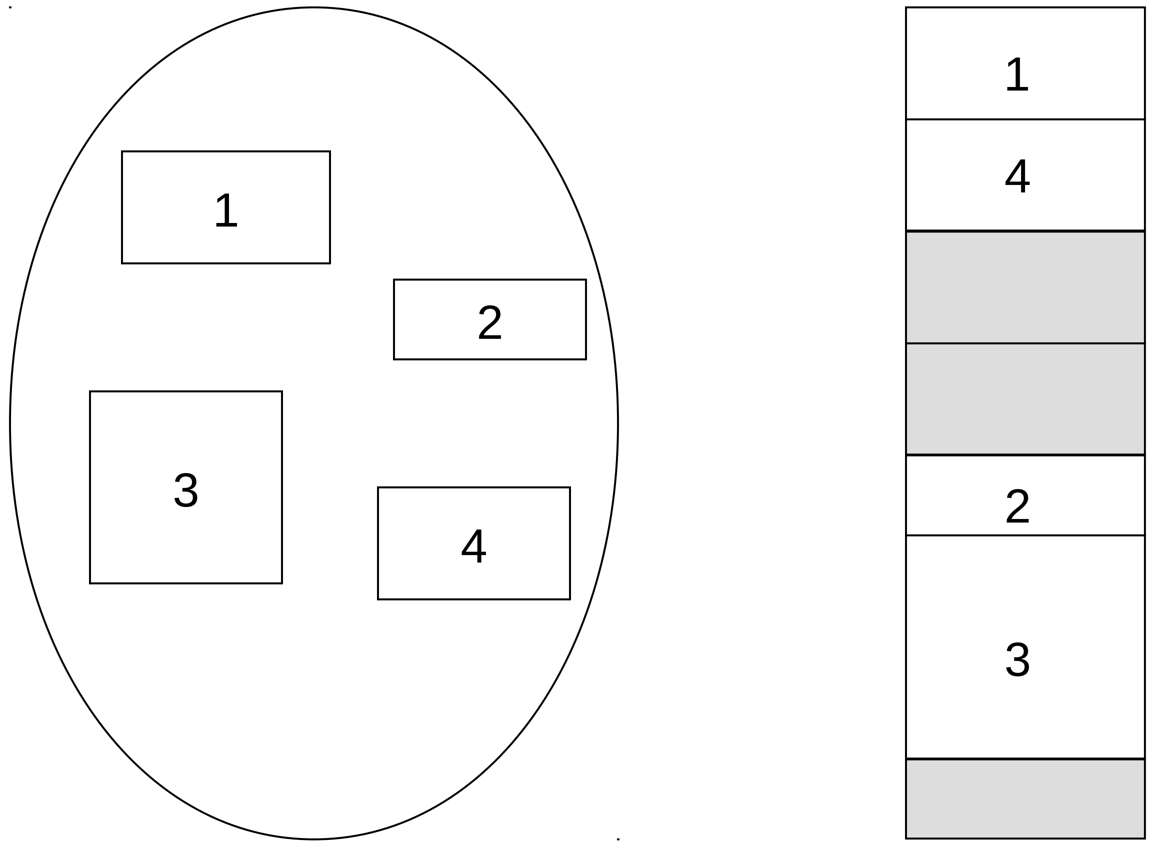
    symbol table

    arrays

User’s View of a Program



Logical View of Segmentation



user space

physical memory space

Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

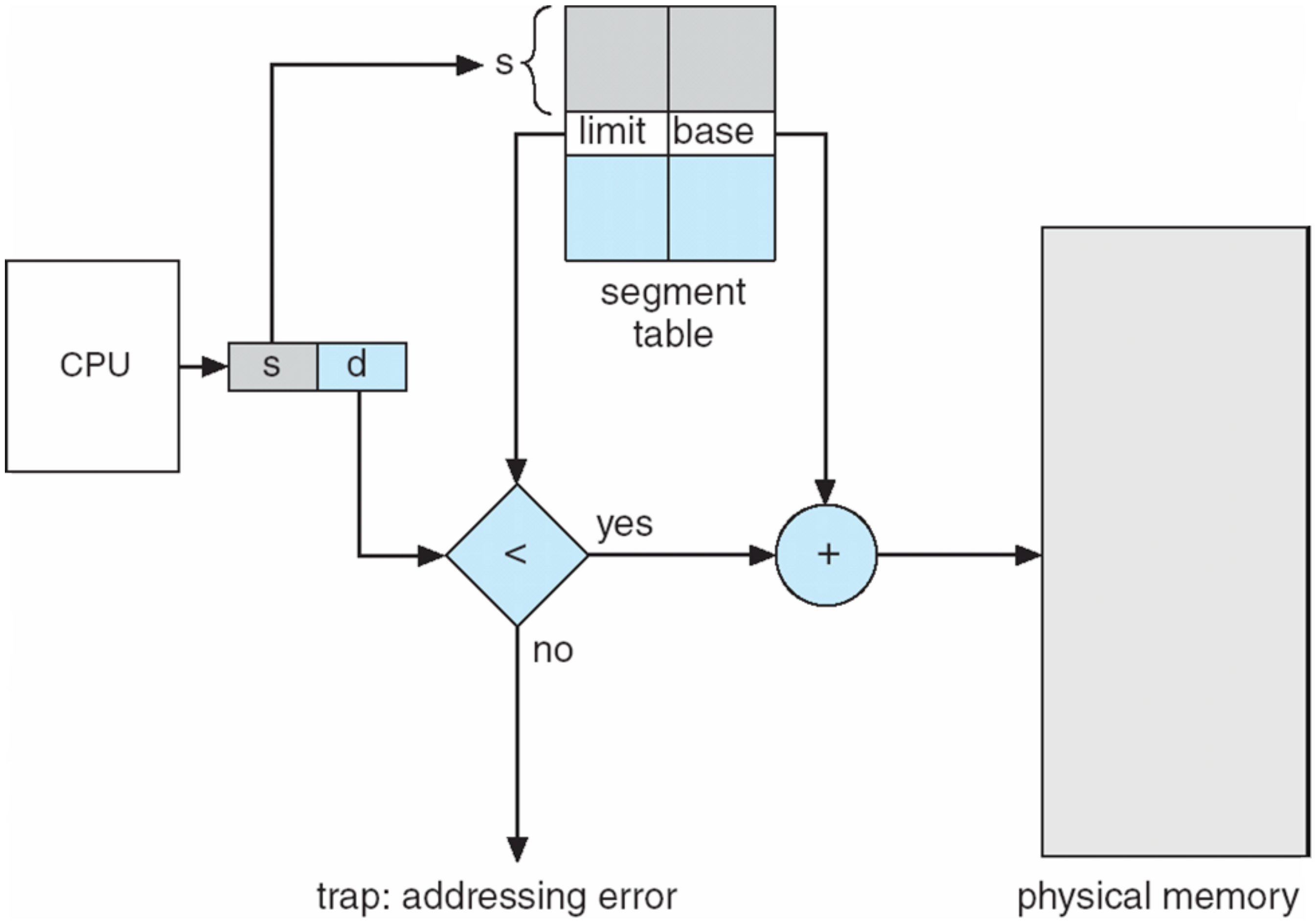
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location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

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

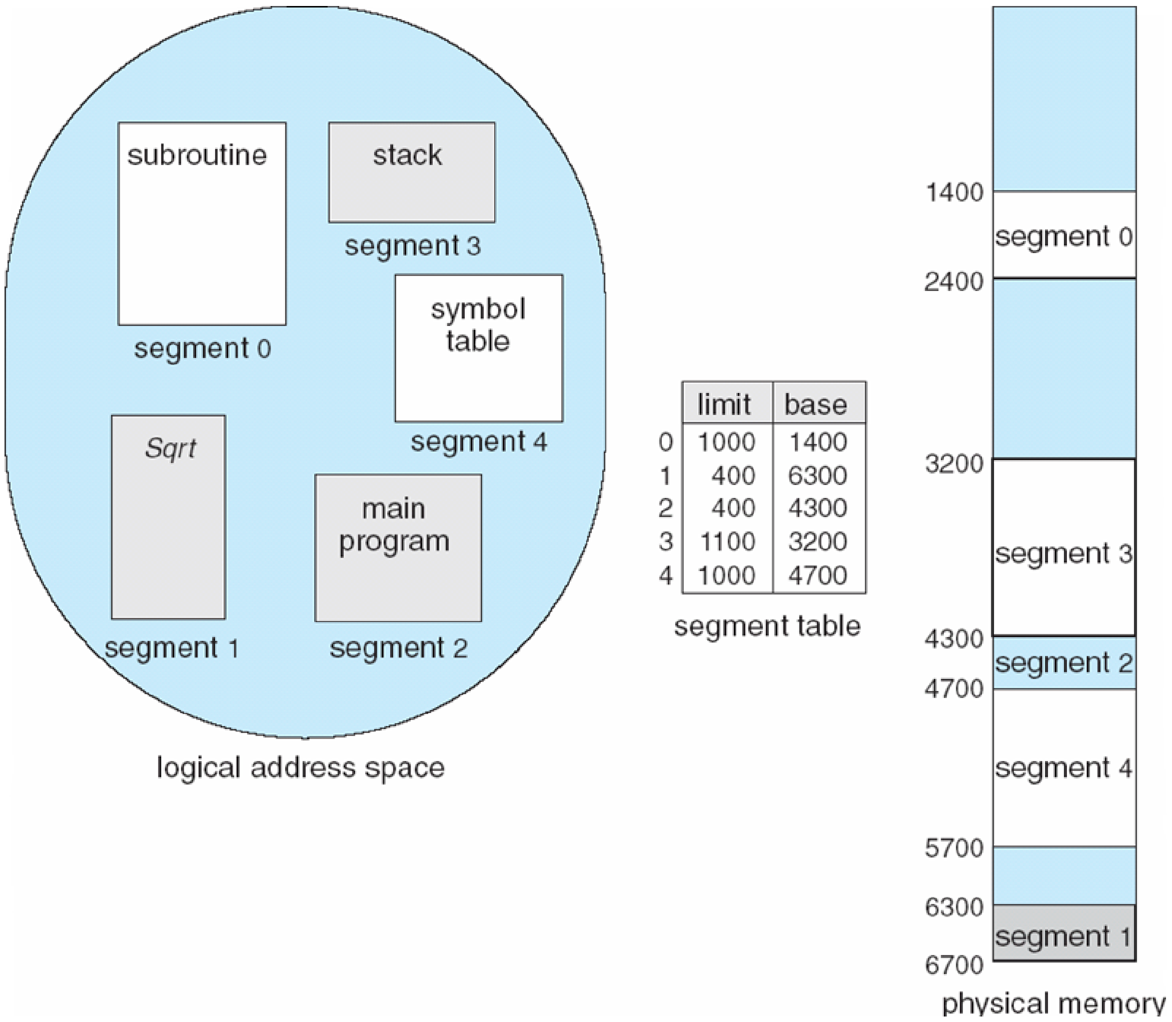
Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

End of Chapter 7