

# 9 Memory Management Strategies

# Contents

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation
- Example: The Intel Pentium

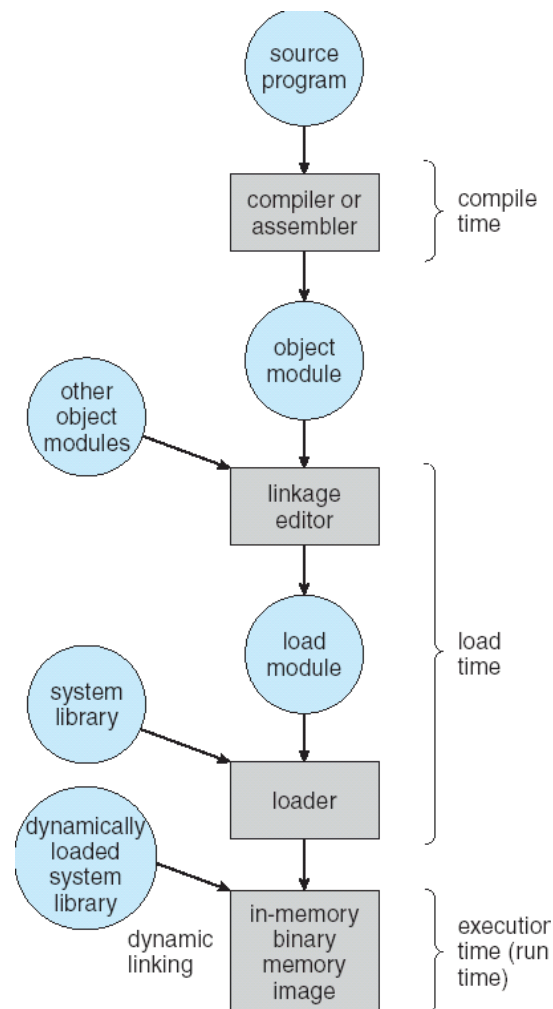
# Background

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

# 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)

# Multi-step Processing of a User Program



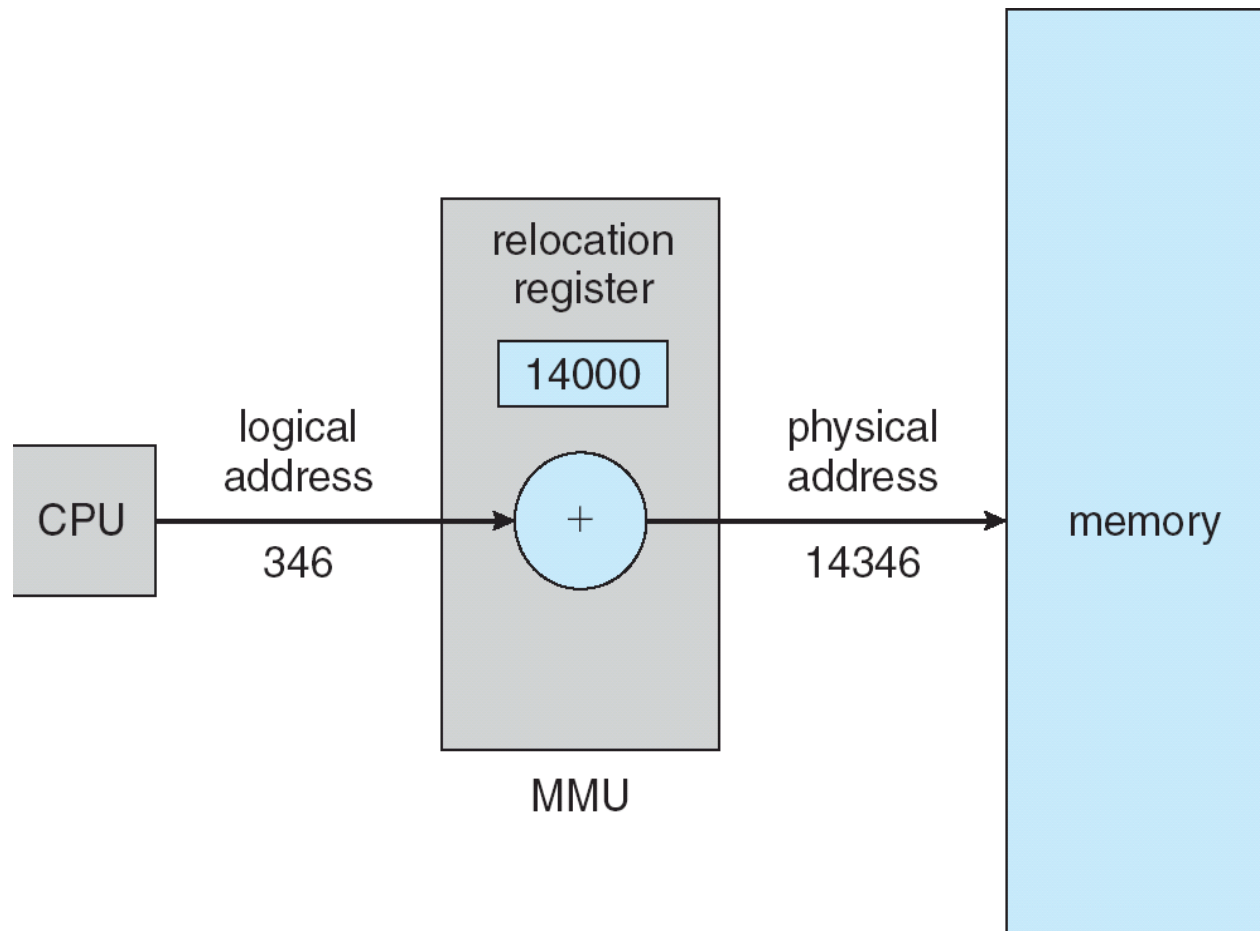
# Logical vs. Physical Address Space

- 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
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; They differ in execution-time address binding scheme

# Memory Management Unit (MMU)

- Hardware 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
- The user program deals with logical addresses; it never sees the real physical addresses

# Dynamic Relocation Using a Relocation Register

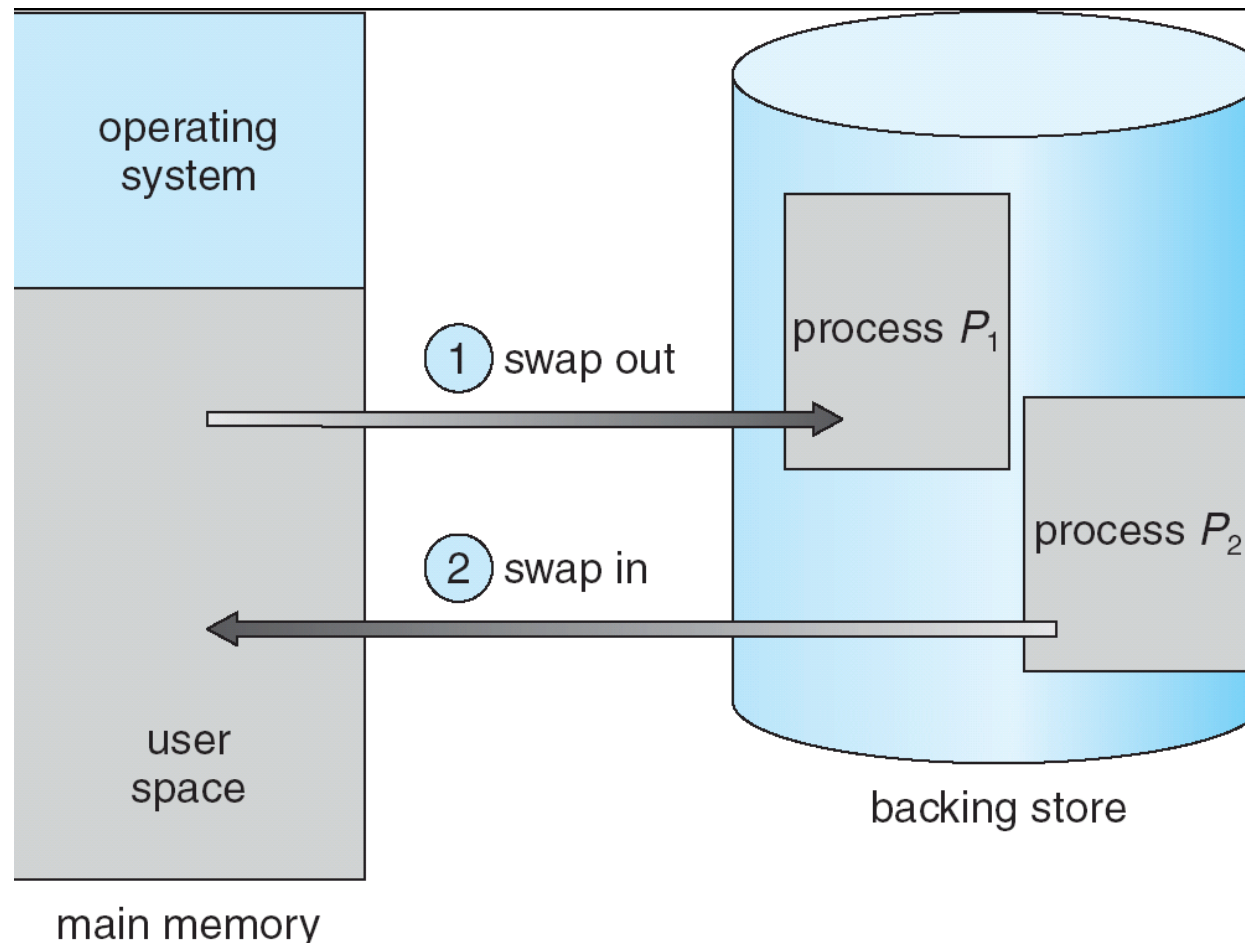




# Swapping

- 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

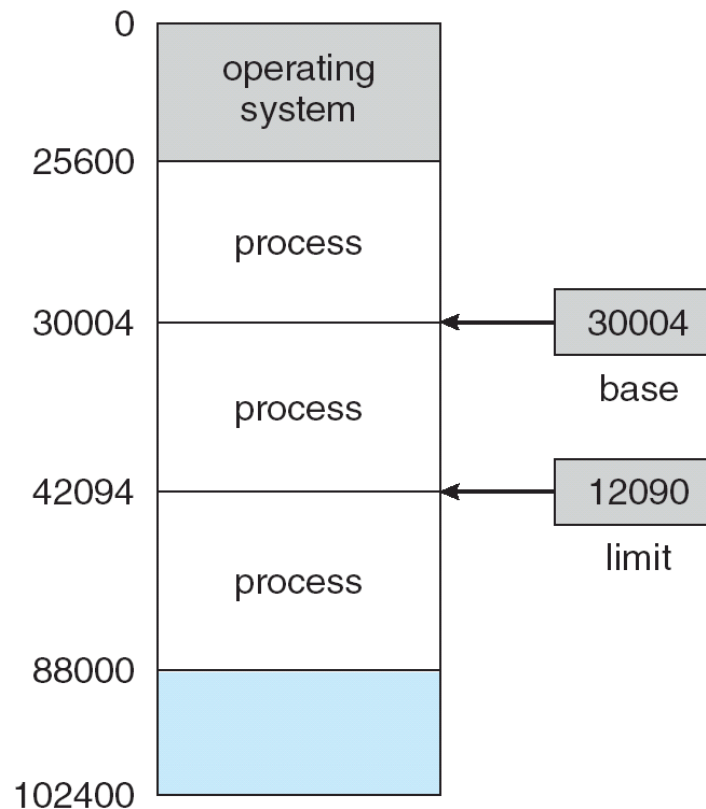


# Contiguous Allocation

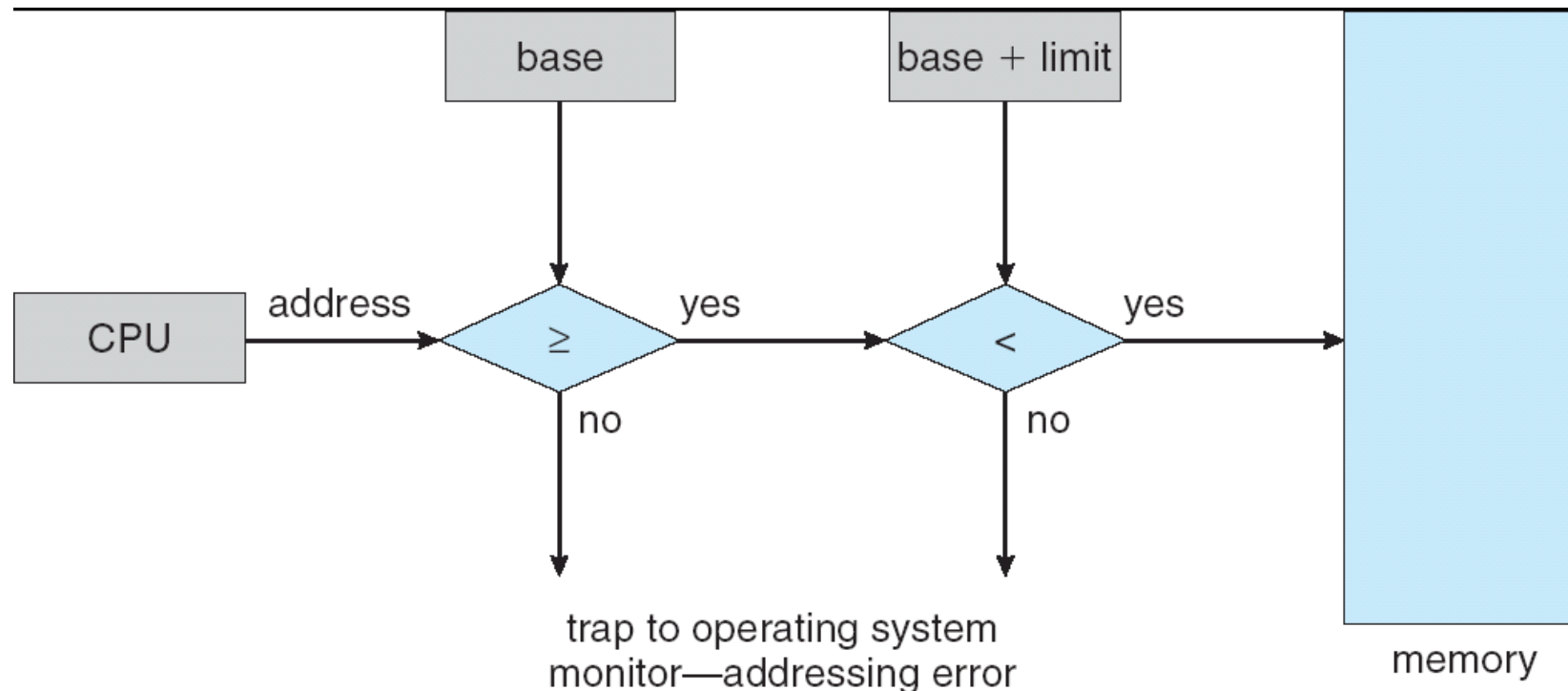
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Relocation registers used to protect user processes from each other, and from changing operating system code and data
  - Base register contains value of 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

# Base and Limit Registers

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

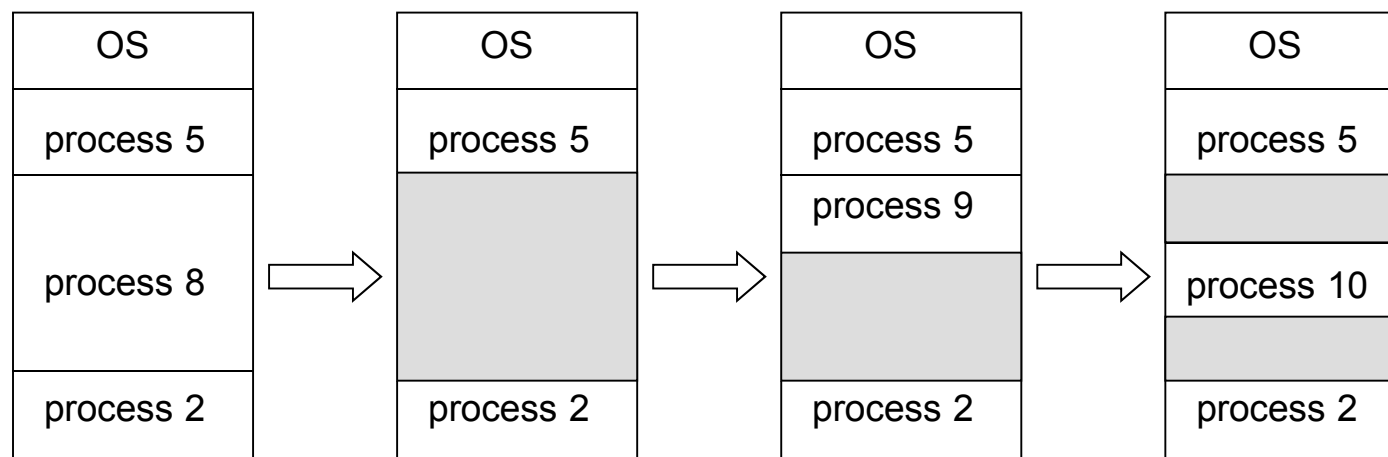


# HW Address Protection with Base and Limit Registers



# 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

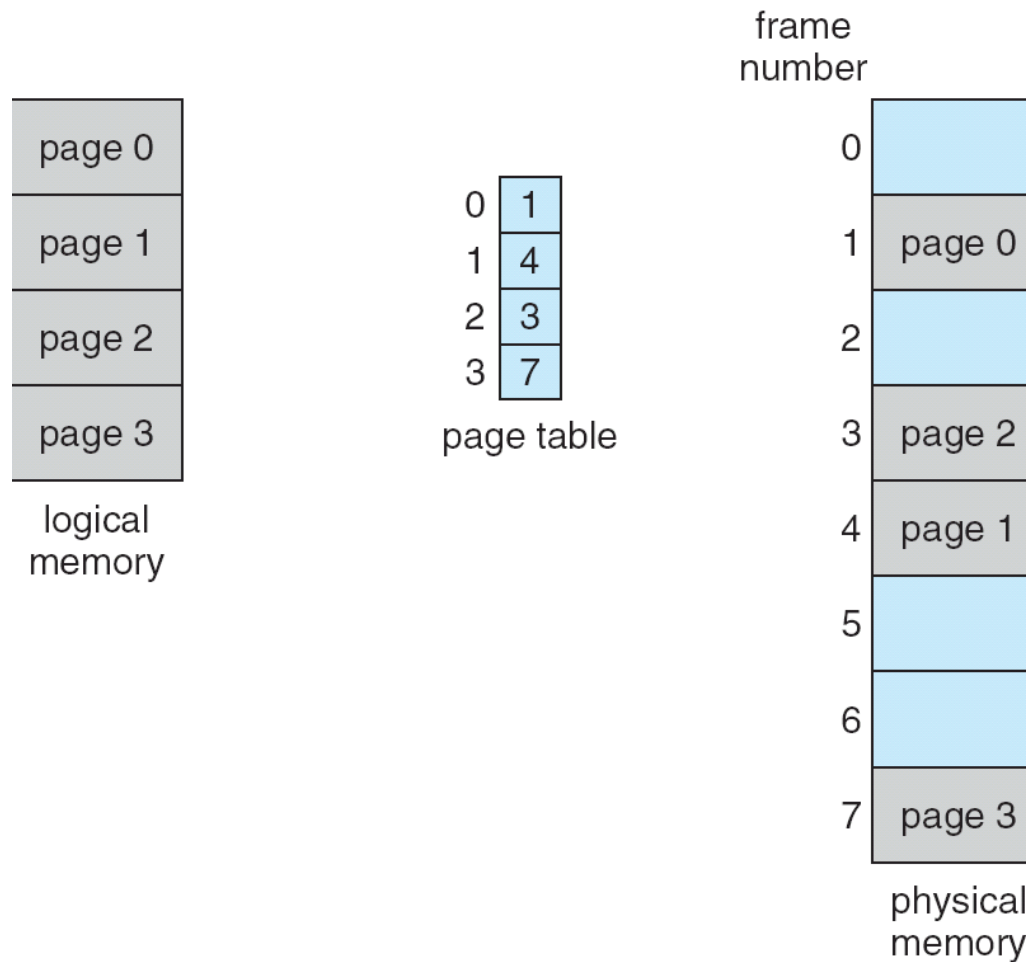
- External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation – allocated memory may be slightly larger than requested memory
- Reduce external fragmentation by **garbage collection** and **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time
  - I/O problem: (1) Latch job in memory while it is involved in I/O, (2) Do I/O only into OS buffers



# Paging

- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, 512 bytes ~ 4KB)
- Divide logical memory into blocks of same size called **pages**
- 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; but no external fragmentation
- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available

# Paging Model of Logical and Physical Memory



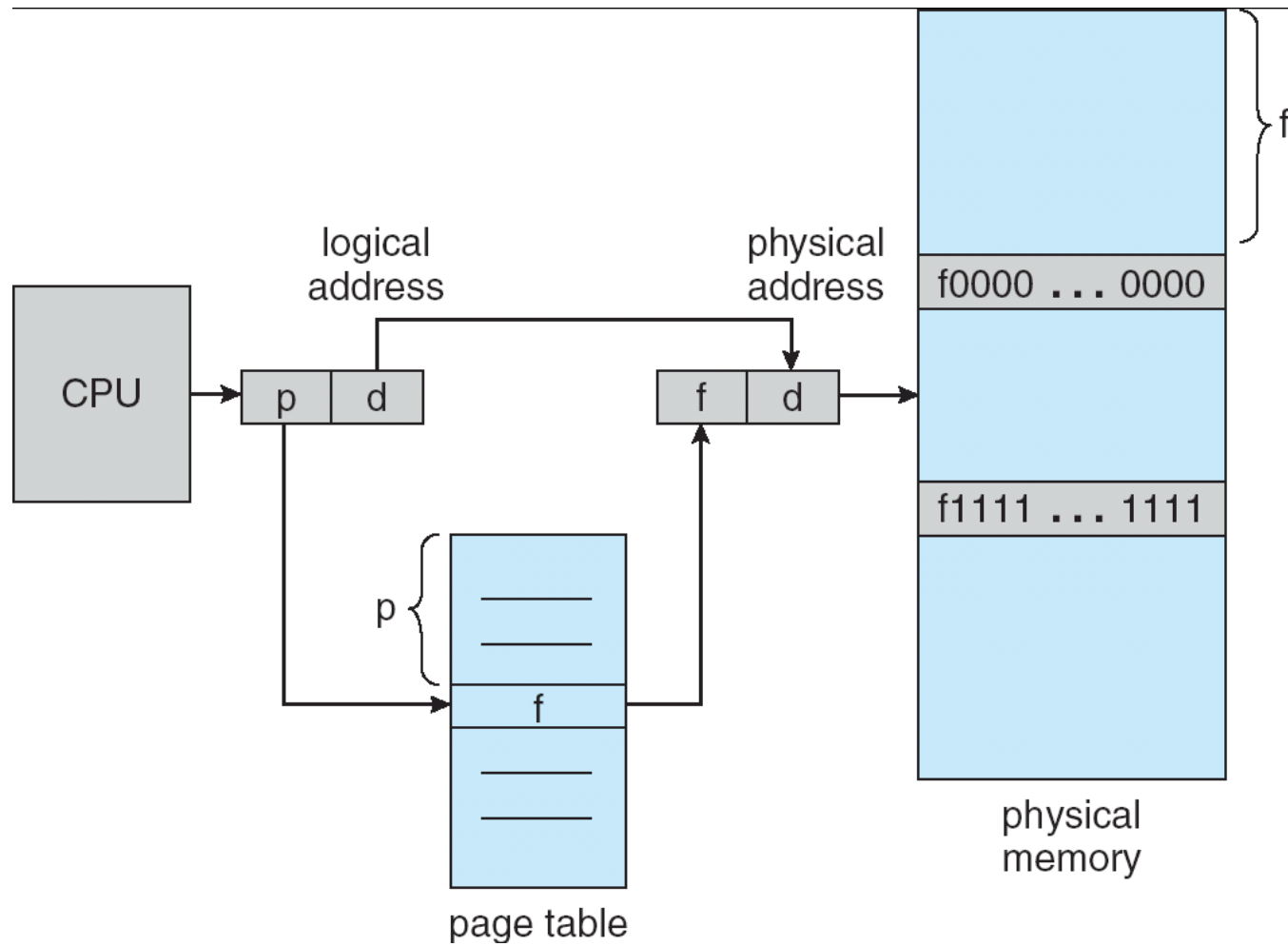
# Address Translation Scheme

- 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

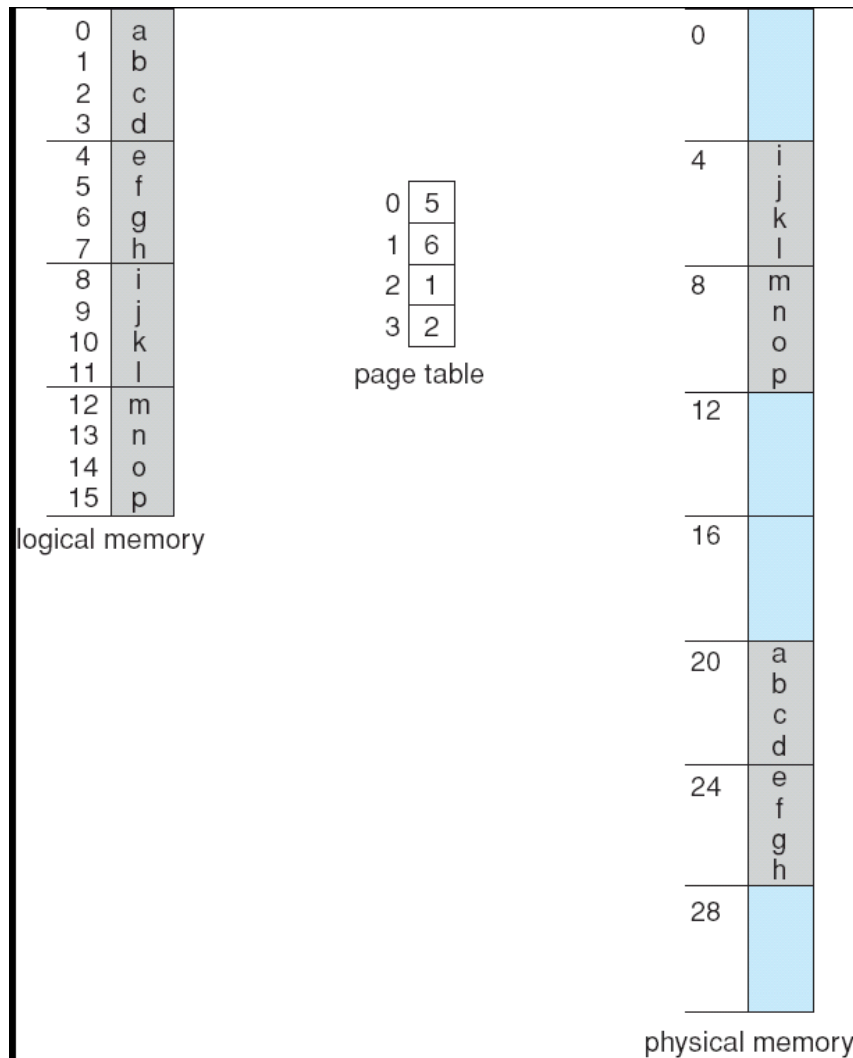
page number	page offset
$p$	$d$
$m - n$	$n$

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

# Paging Hardware

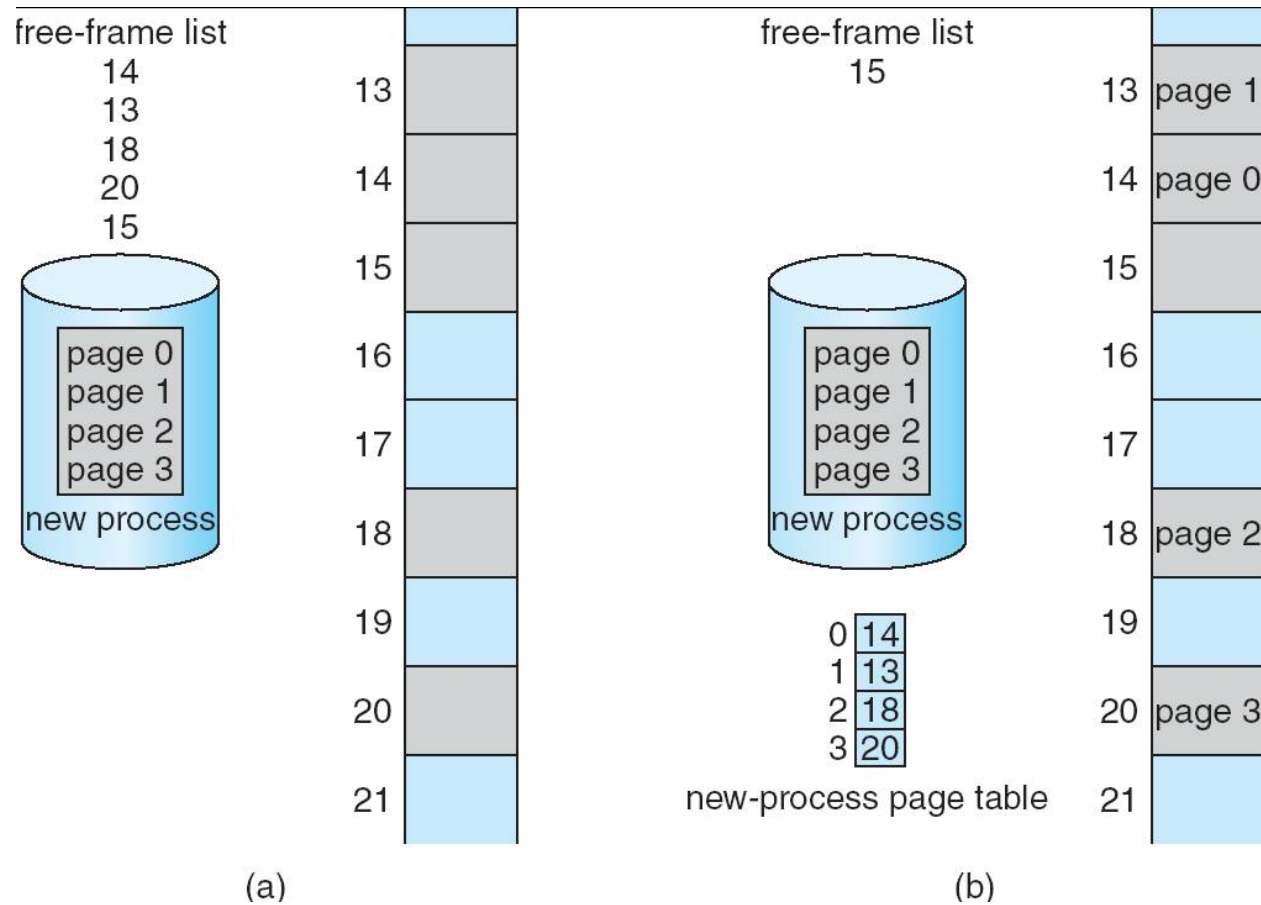


# Paging Example



32-byte memory and 4-byte pages

# Free Frames



# Implementation of Page Table

- Page table is a part of kernel data, one for each process; must be kept in main memory
- **Page-table base register (PTBR)**
- **Page-table length register (PRLR)**
- Every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- A special fast-lookup hardware cache called associative memory or **translation lookaside buffers (TLBs)** must be used
- 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

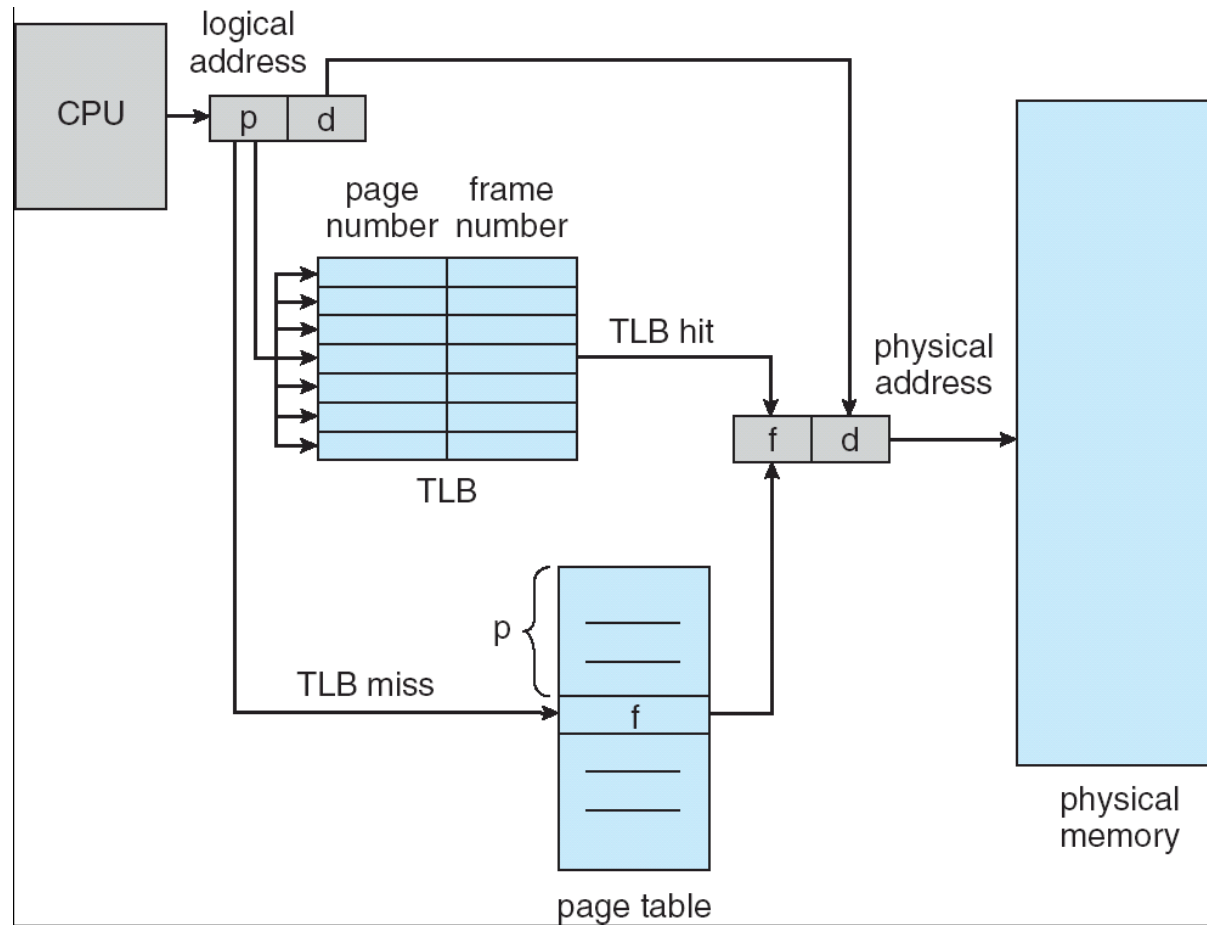
- Associative memory – parallel search

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

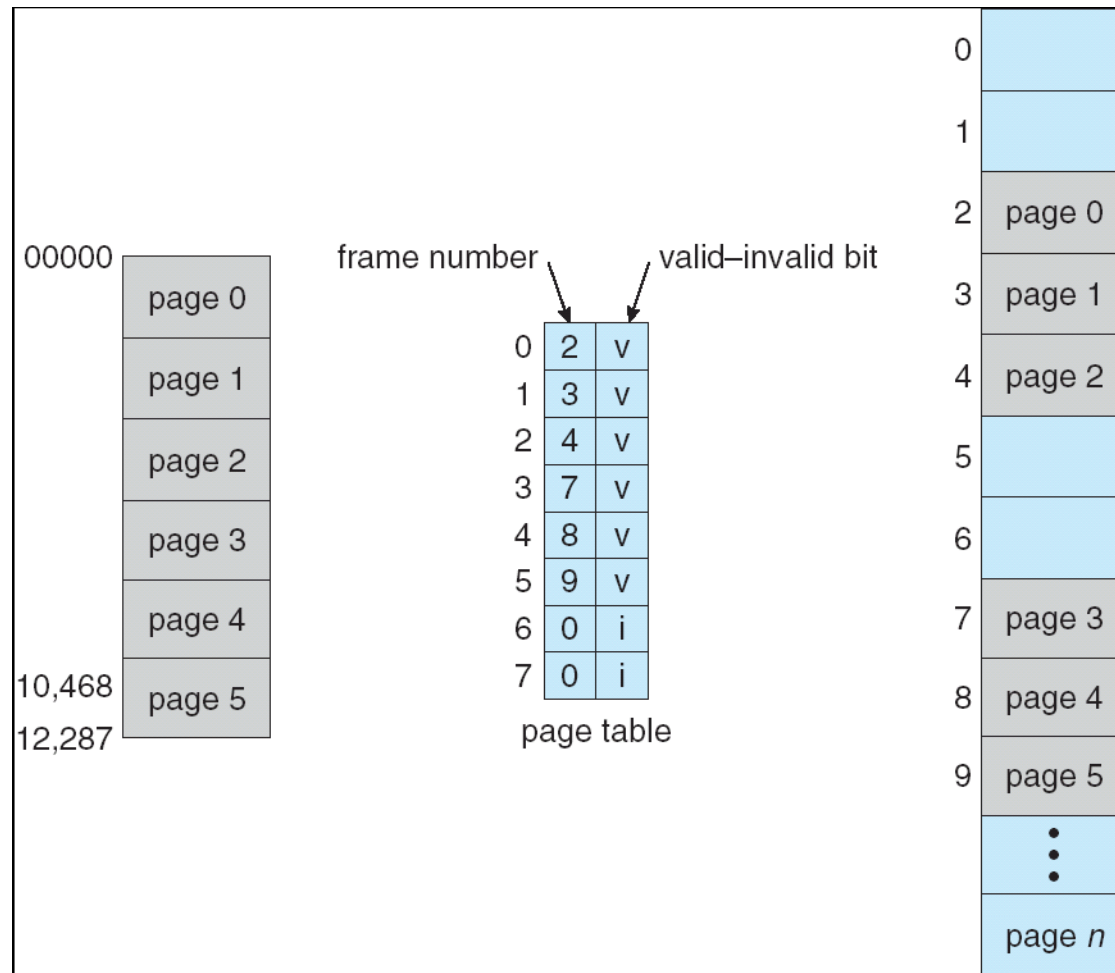
- Associative Lookup =  $\Sigma$  time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio =  $\langle$
- **Effective Access Time (EAT)**

$$\begin{aligned} \text{EAT} &= (1 + \Sigma) \langle + (2 + \Sigma)(1 - \langle) \\ &= 2 + \Sigma - \langle \end{aligned}$$

# 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

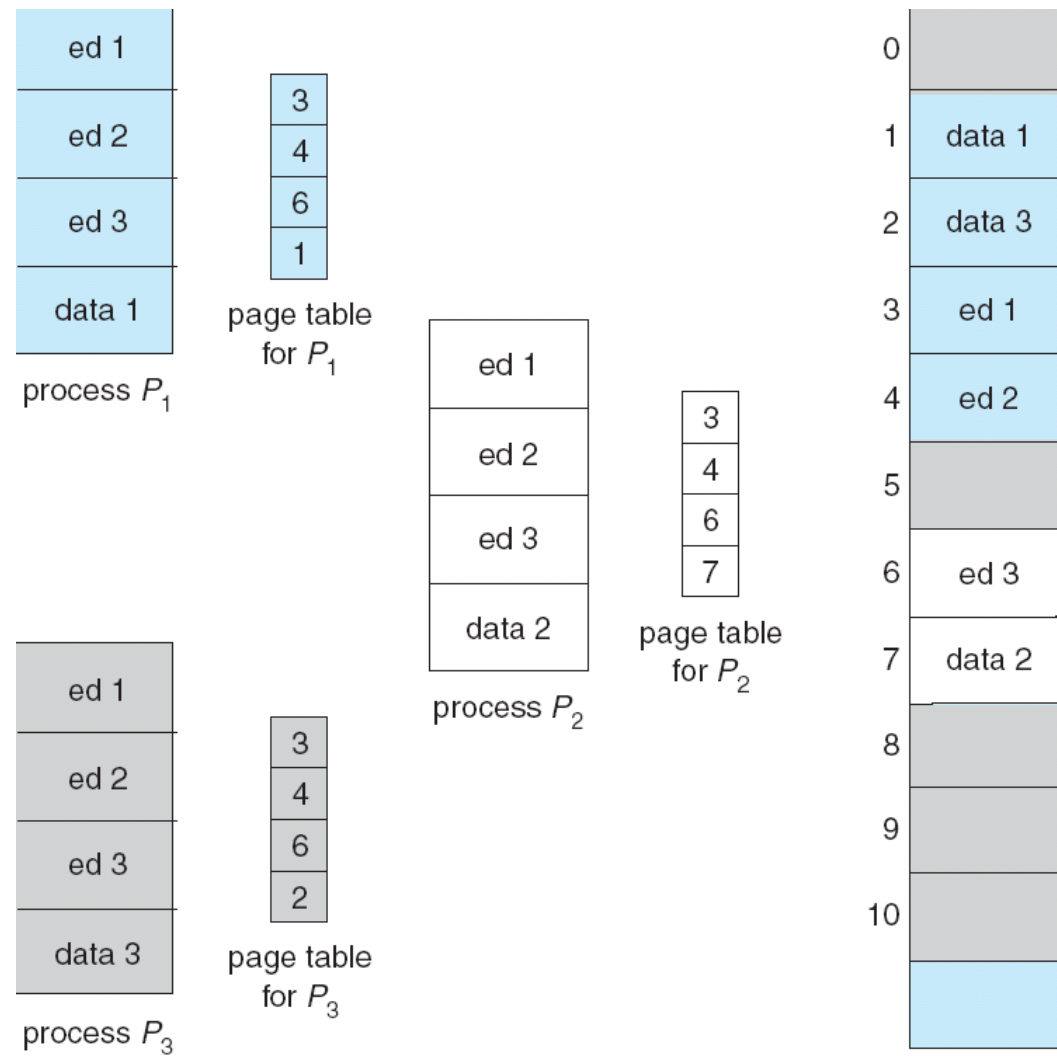
# Valid(v) or Invalid(i) Bit in a Page Table



# Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, ...).
  - 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

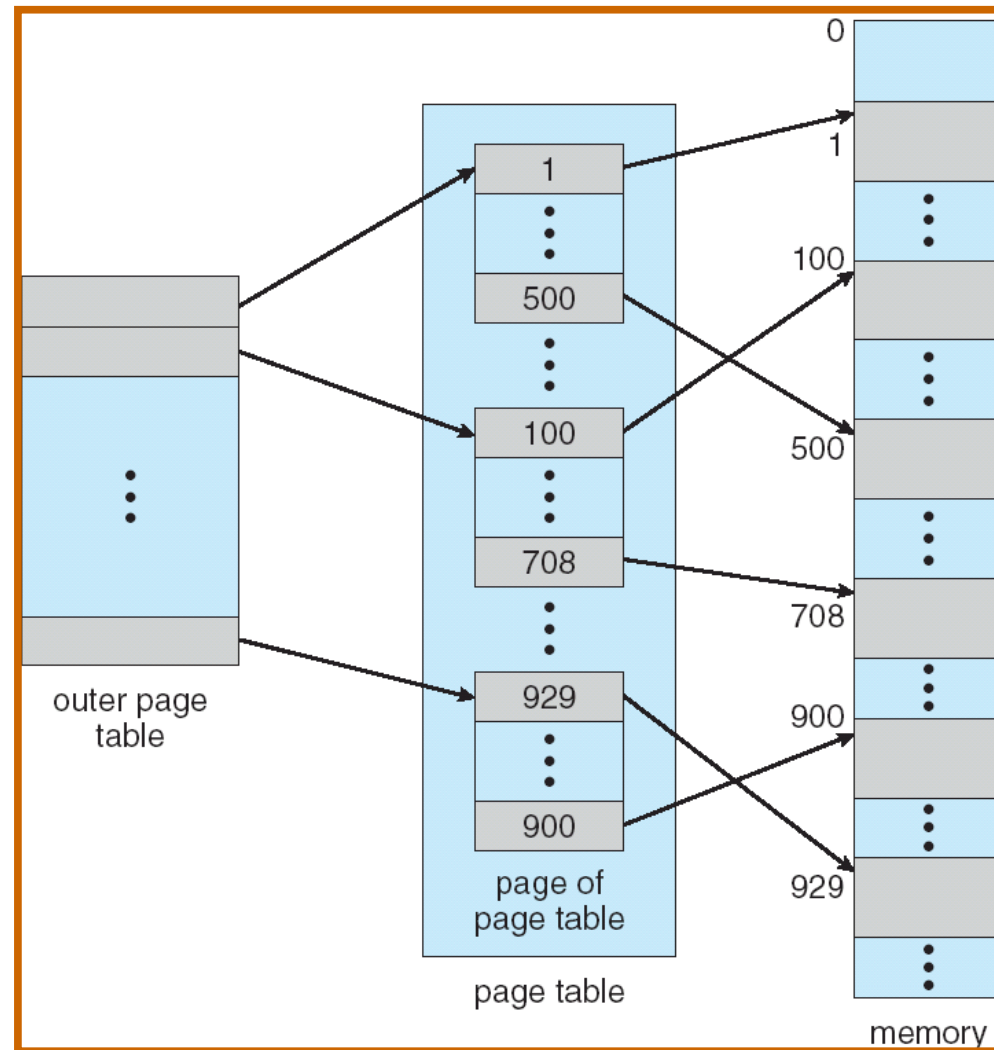
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

# 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



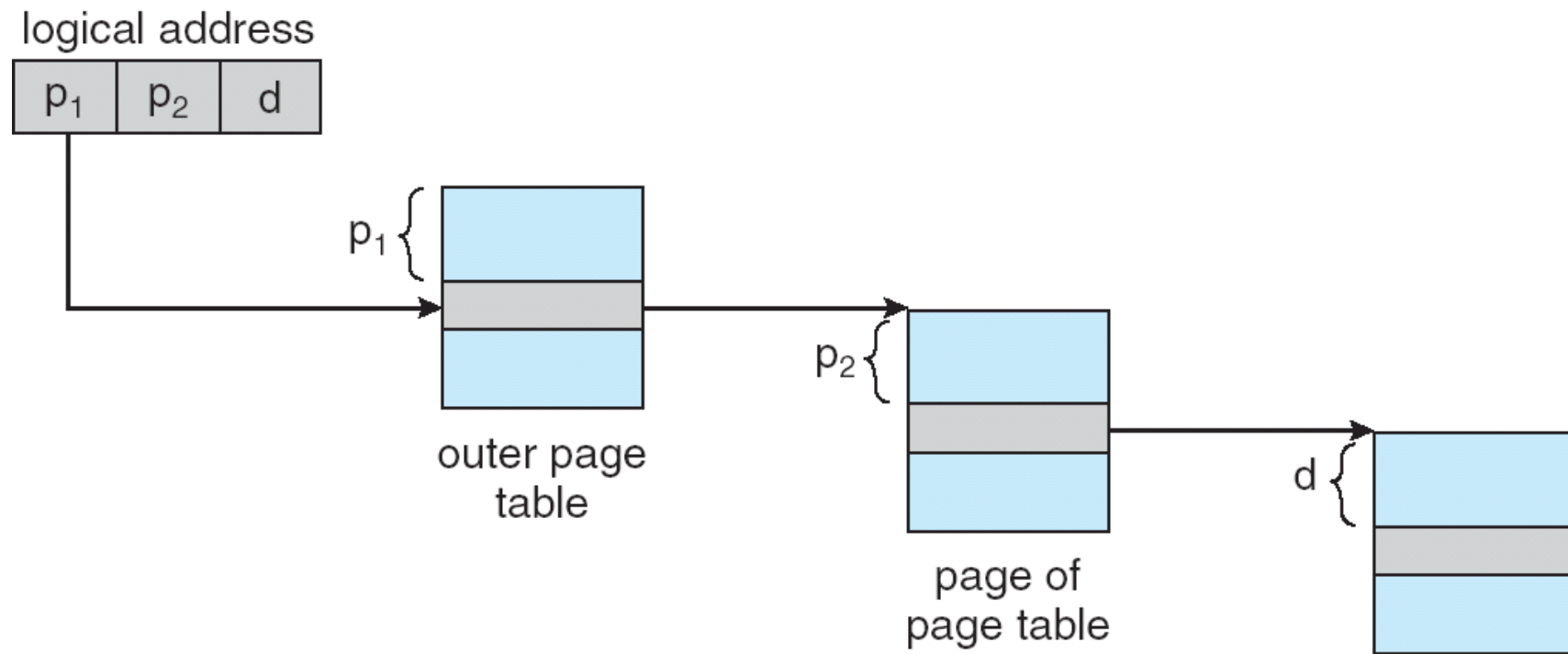
# Two-Level Paging Example

- 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 itself is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

page number		page offset
$p_1$	$p_2$	$d$
12	10	10

where  $p_1$  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

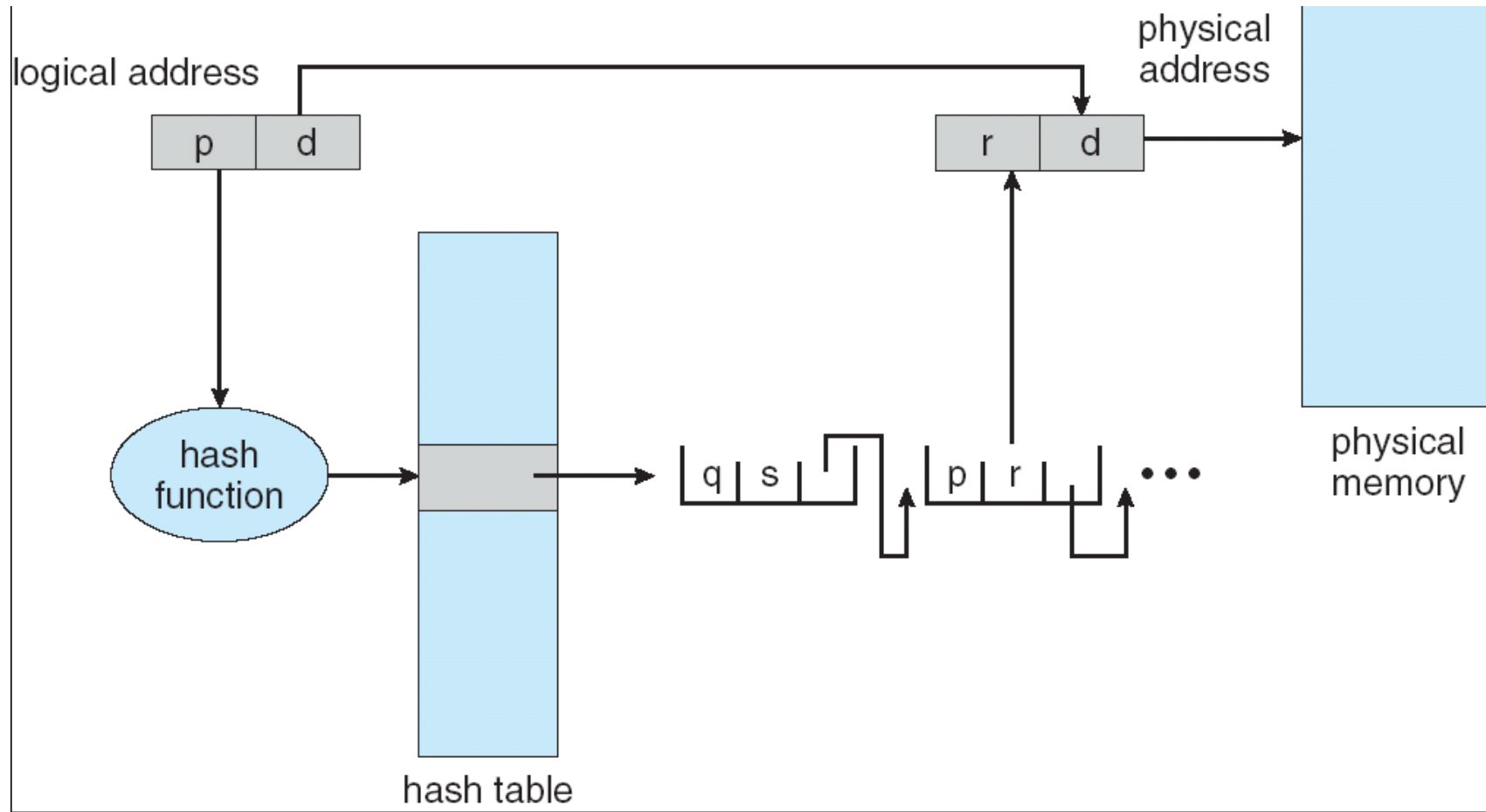
outer page	inner page	offset
$p_1$	$p_2$	$d$
42	10	12

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.

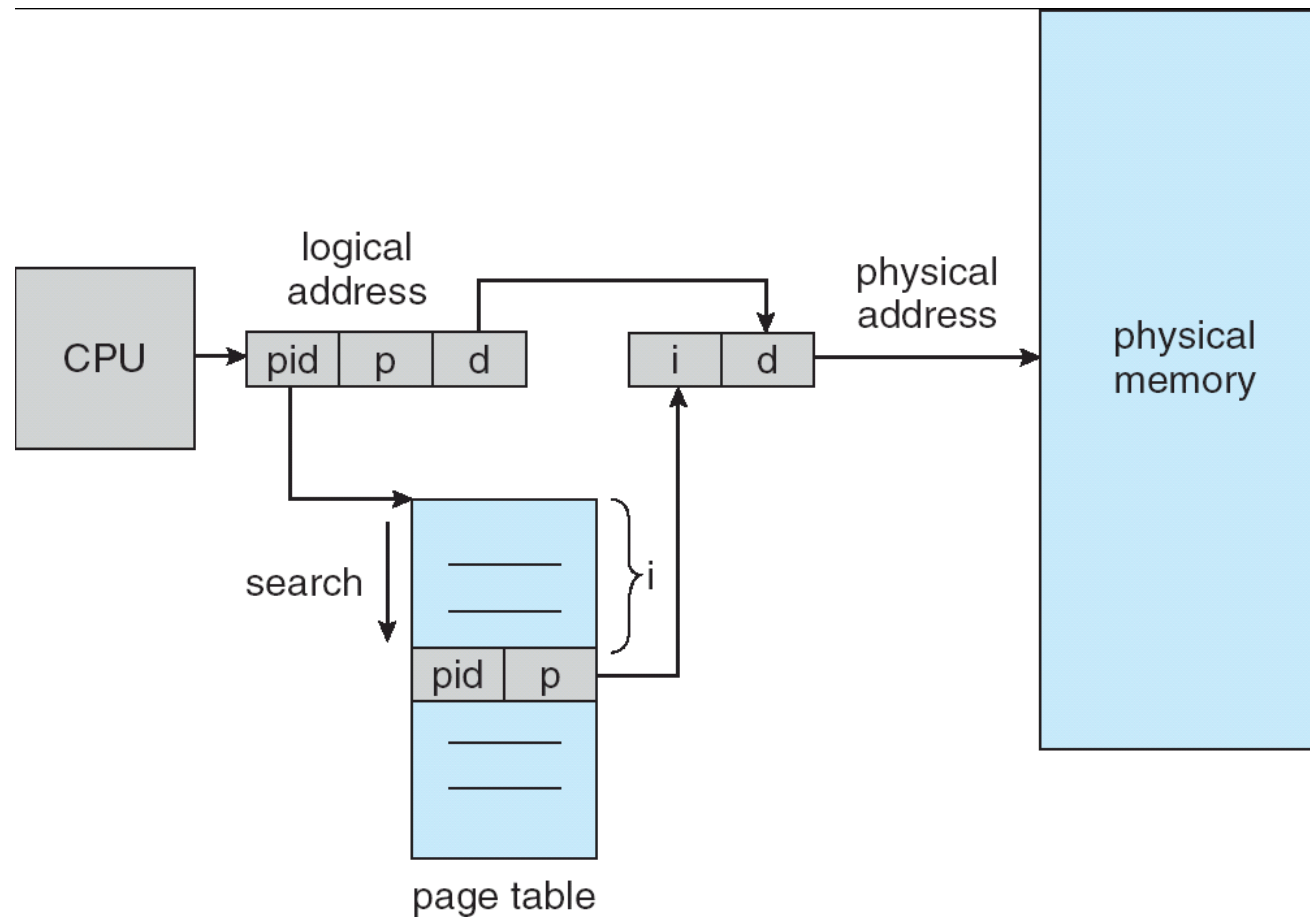
# Hashed Page Table



# 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

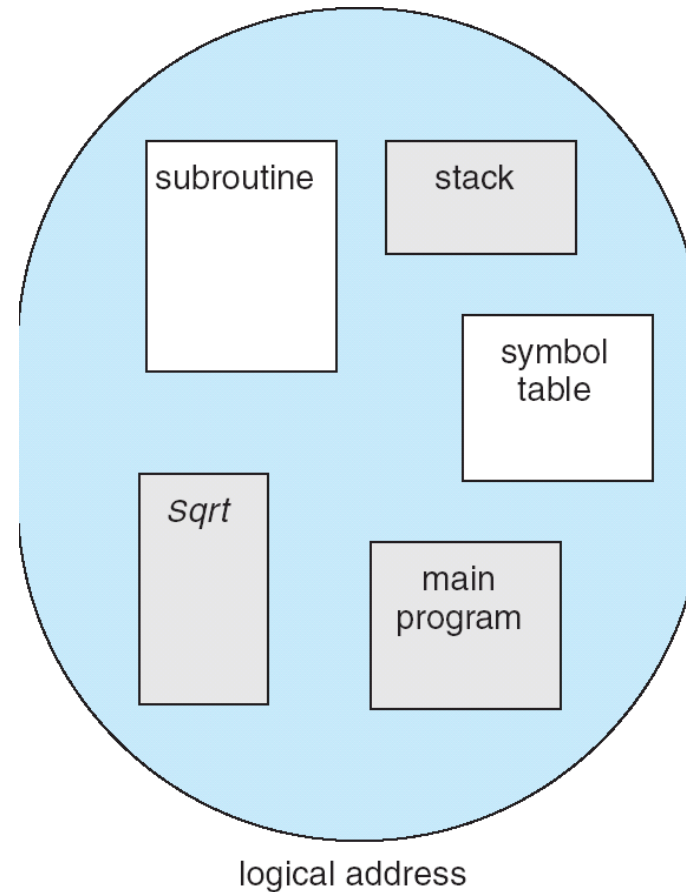




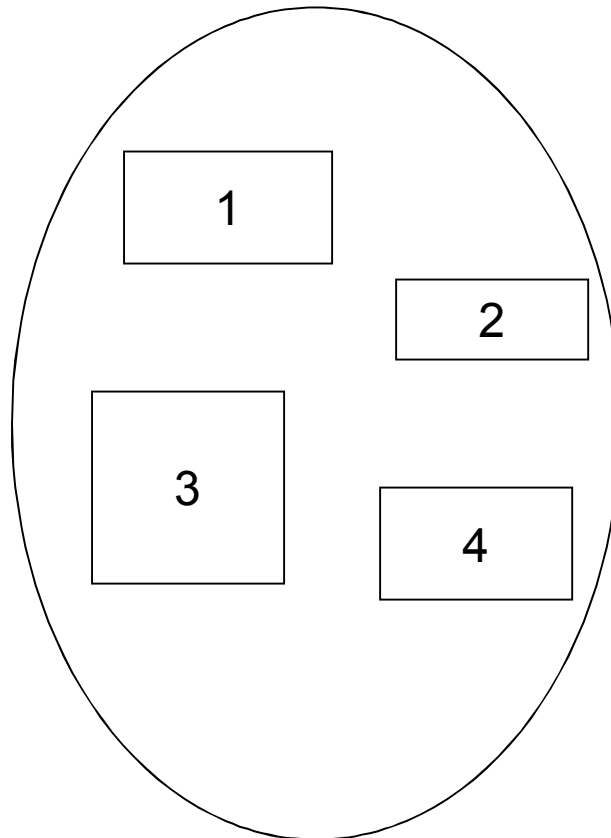
# Segmentation

- 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 / global variables, common block, stack, symbol table, arrays, ...

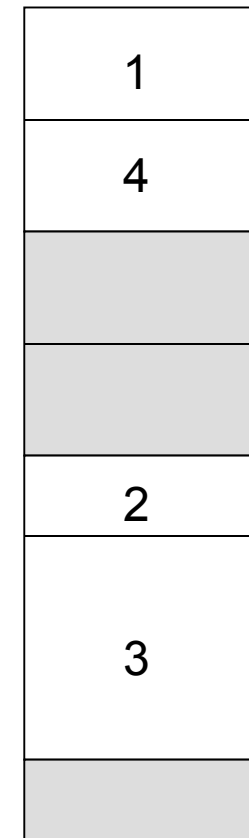
User's view of a Program



# Logical View of Segmentation



user space



physical memory space

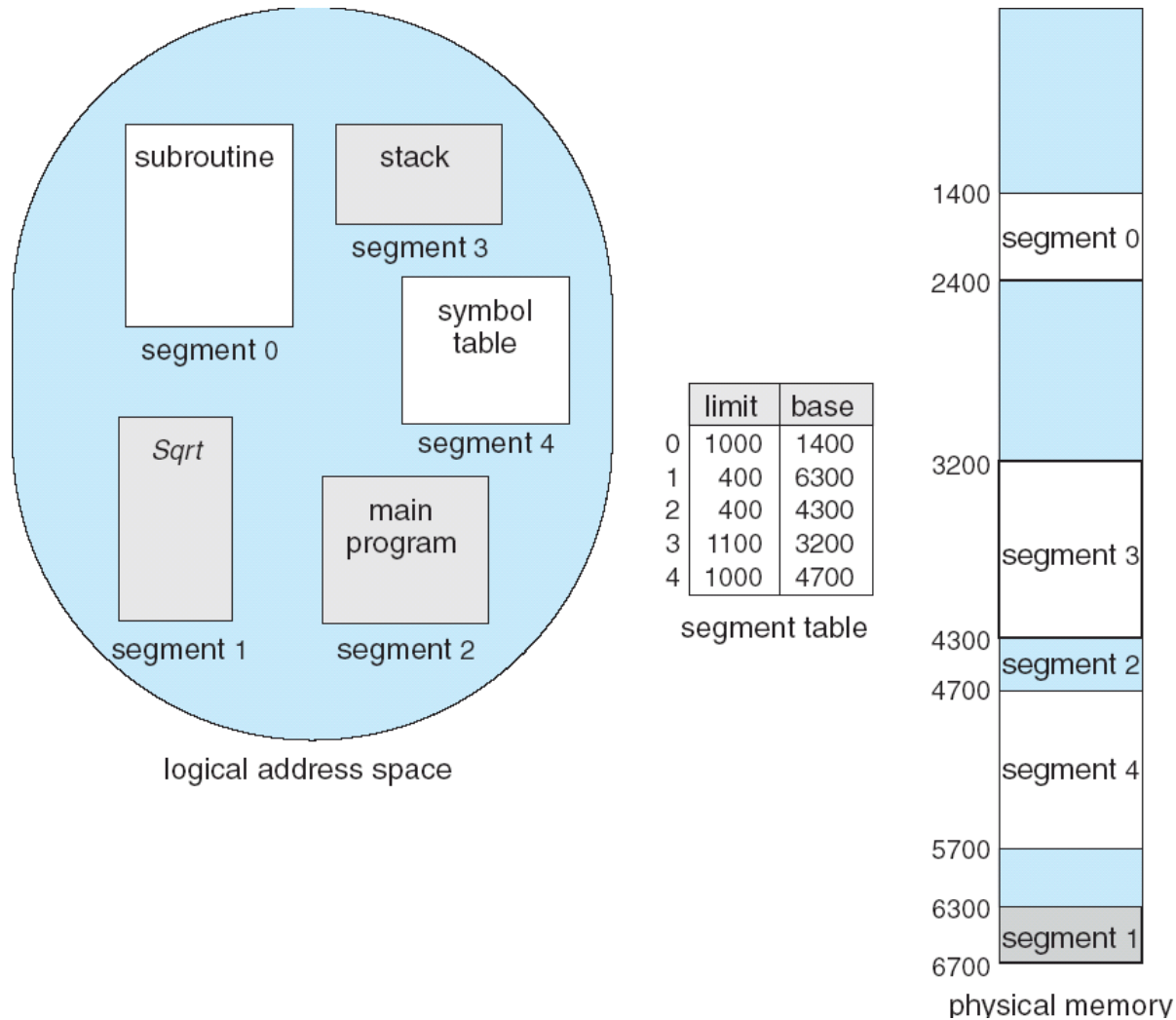
# Segmentation Architecture

- 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**

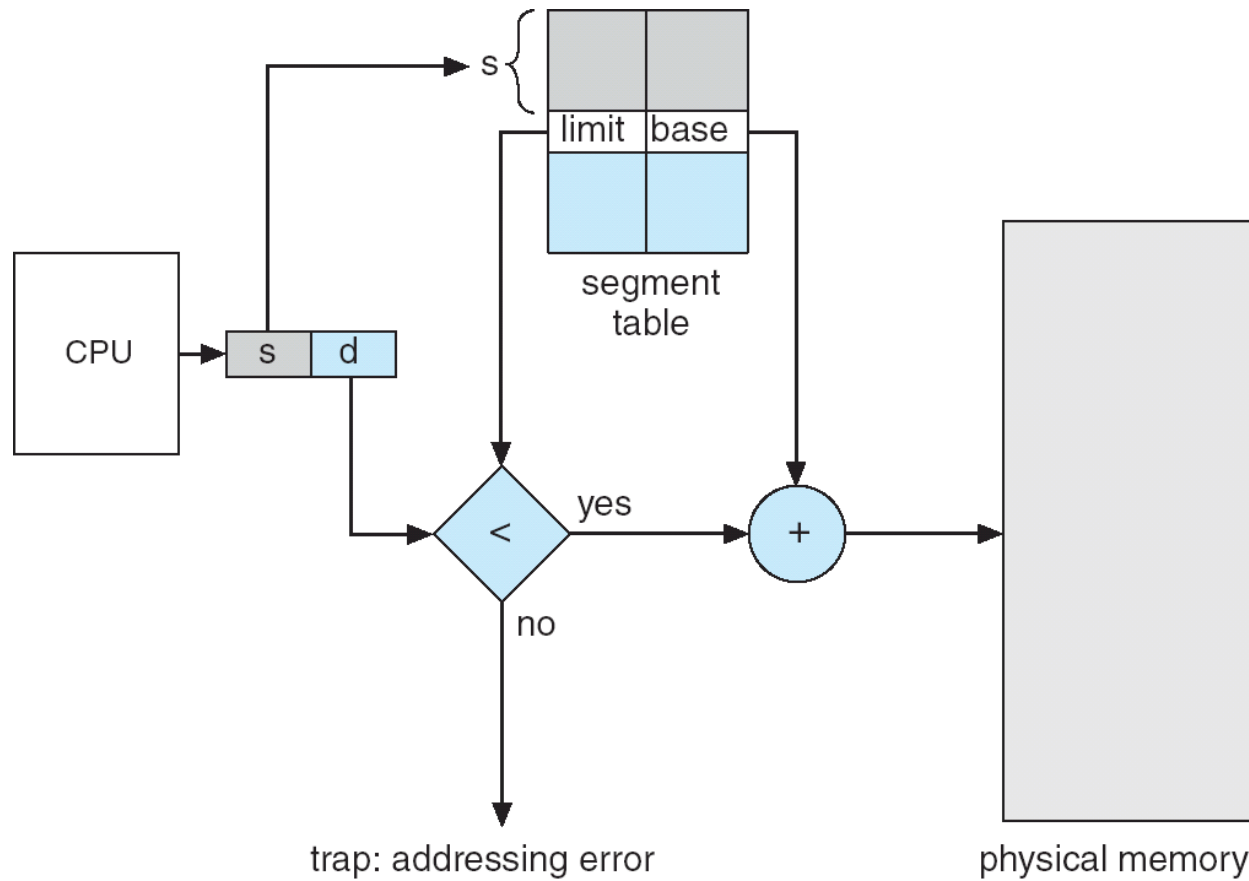
# Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ® 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

# Example of Segmentation



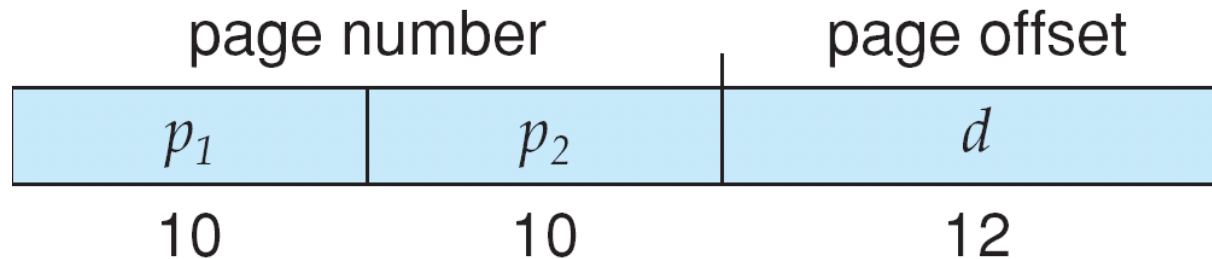
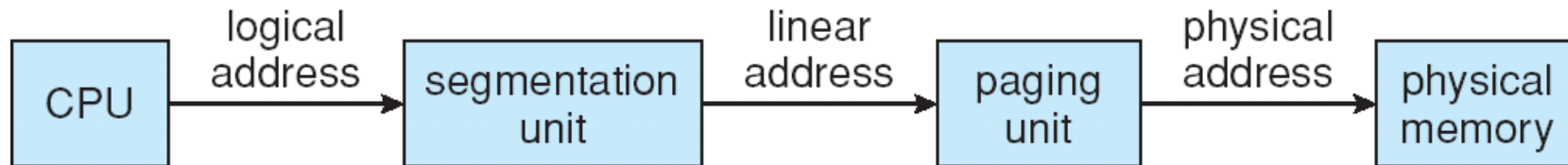
# Segmentation Hardware



# Example: The Intel Pentium

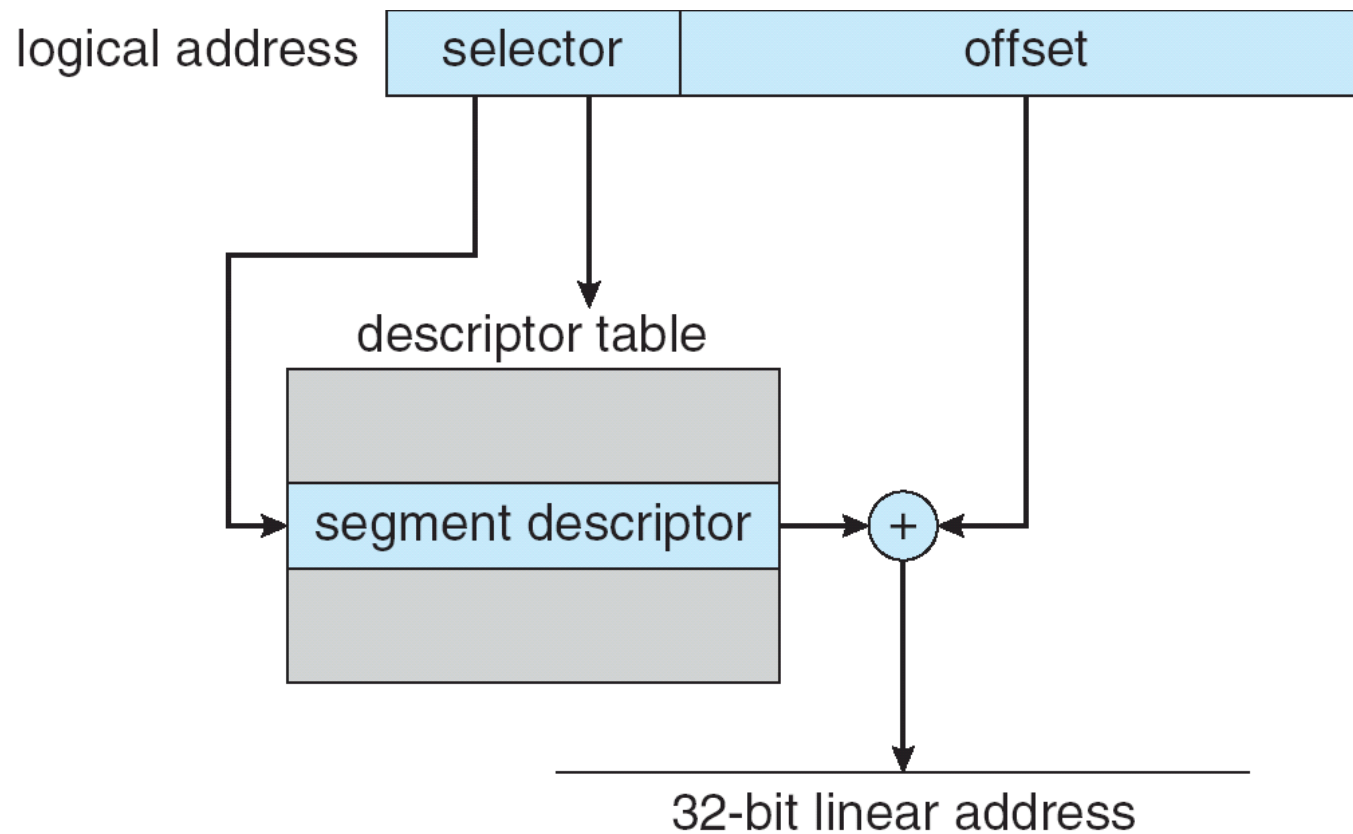
- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU

# Logical to Physical Address Translation in Pentium

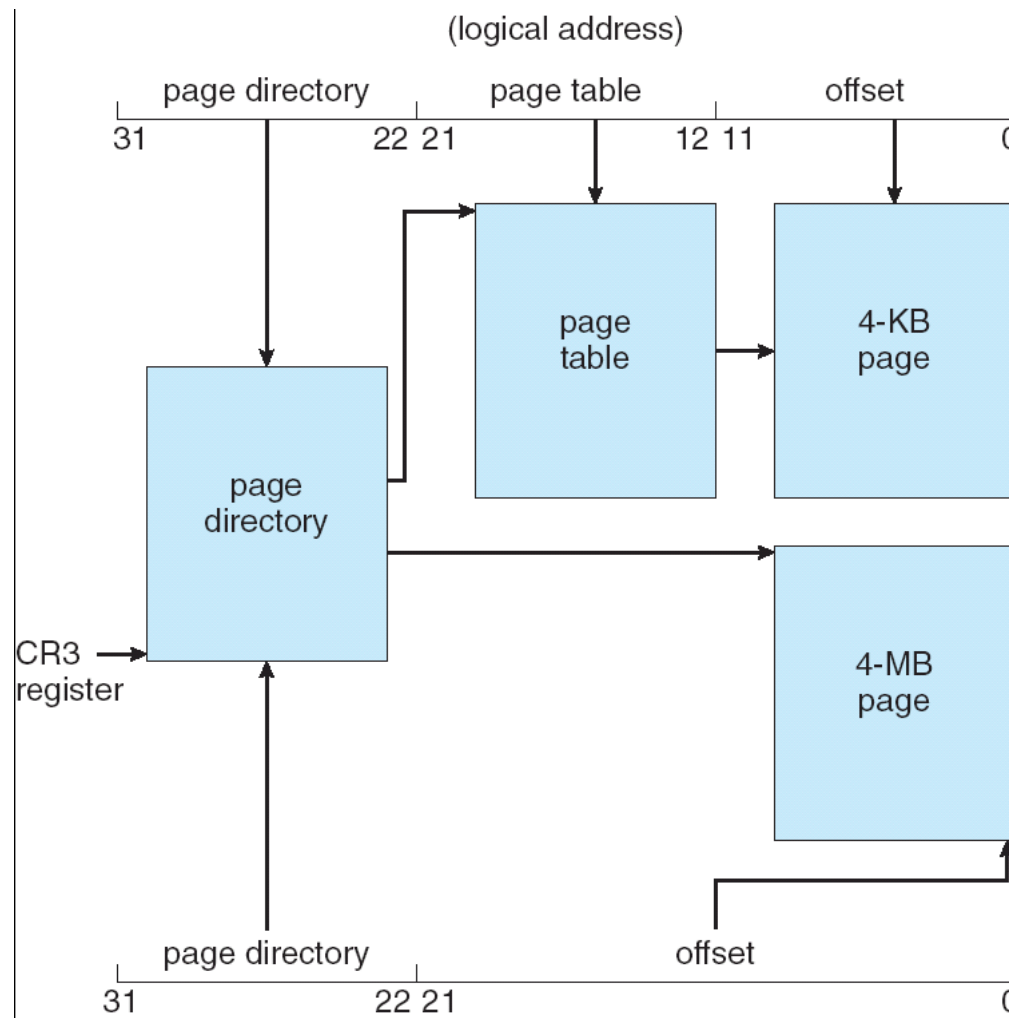




# Intel Pentium Segmentation



# Pentium Paging Architecture



# Linear Address in Linux

Broken into four parts:

global directory	middle directory	page table	offset
---------------------	---------------------	---------------	--------

# Three-level Paging in Linux

