




# Operating System Concepts

Che-Wei Chang

[chewei@mail.cgu.edu.tw](mailto:chewei@mail.cgu.edu.tw)

Department of Computer Science and Information Engineering, Chang Gung University

# Contents

1. Introduction
2. System Structures
3. Process Concept
4. Multithreaded Programming
5. Process Scheduling
6. Synchronization
7. Deadlocks
-  8. Memory-Management Strategies
9. Virtual-Memory Management
10. File System
11. Implementing File Systems
12. Secondary-Storage Systems

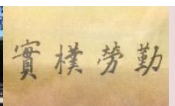




# Chapter 8. Memory– Management Strategies

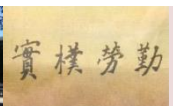
# Objectives

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



# Background

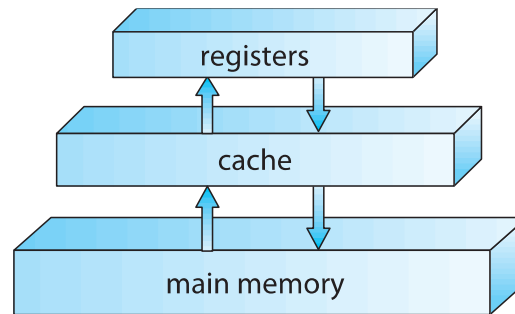
- ▶ Program must be brought (from disk) into memory and placed within a process for it to be run
- ▶ Main memory and registers are the storages which CPU can access directly
- ▶ Register access is in one CPU clock (or less)
- ▶ Main memory access can take cycles, causing a **stall**
- ▶ **Cache** sits between main memory and CPU registers



# Storage-Device Hierarchy

## Primary Storage

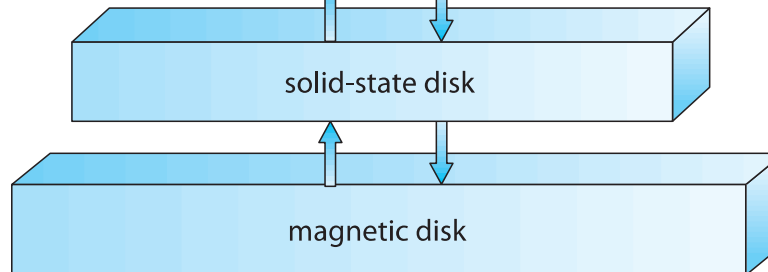
- volatile storage



- Access time: a cycle
- Access time: several cycles
- Access time: many cycles

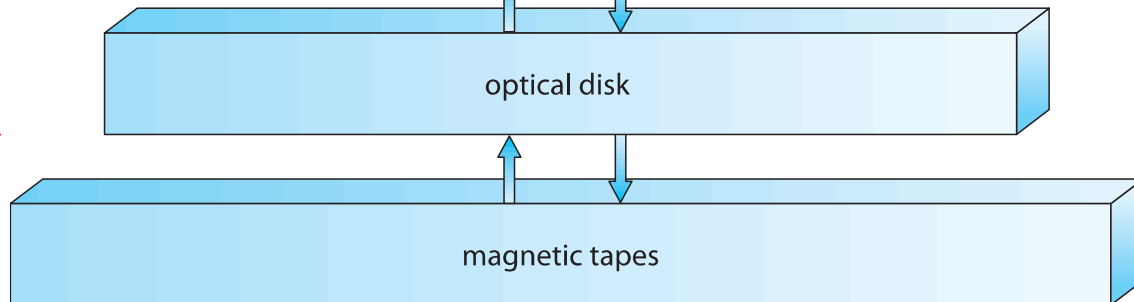
## Secondary Storage

- nonvolatile storage



## Tertiary Storage

- removable media



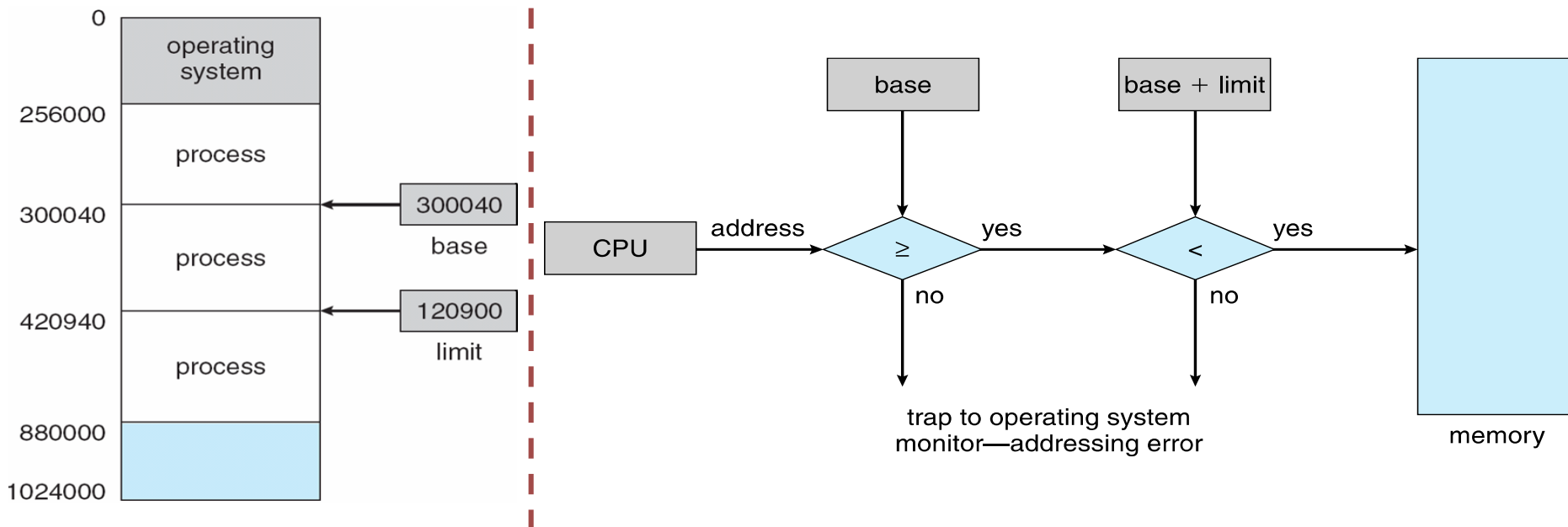
# Memory Management

- ▶ Motivation
  - Keep several processes in memory to improve a system's performance
- ▶ Selection of different memory management methods
  - Application-dependent
  - Hardware-dependent
- ▶ Memory – A large array of words or bytes, each with its own address
  - Memory is always too small
- ▶ What should be done
  - Know which areas are free or used
  - Decide which processes to get memory
  - Perform allocation and de-allocation



# Base and Limit Registers

- ▶ A pair of **base** and **limit registers** define the logical address space
- ▶ CPU must check every memory access generated in user mode is between base and limit for that user



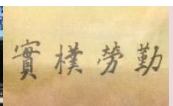


# Binding of Instructions and Data to Memory

- ▶ Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time:** If memory location is 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

# Logical and Physical Address Space

- ▶ **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; logical and physical addresses differ in execution-time address-binding scheme
- ▶ **Logical address space** is the set of all logical addresses generated by a program
- ▶ **Physical address space** is the set of all physical addresses generated by a program



# Dynamic Linking

- ▶ Static linking – system libraries and program code combined by the loader into the binary program image
- ▶ Dynamic linking –linking postponed until execution time
- ▶ Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- ▶ Stub replaces itself with the address of the routine, and executes the routine
- ▶ Dynamic linking is particularly useful for shared libraries

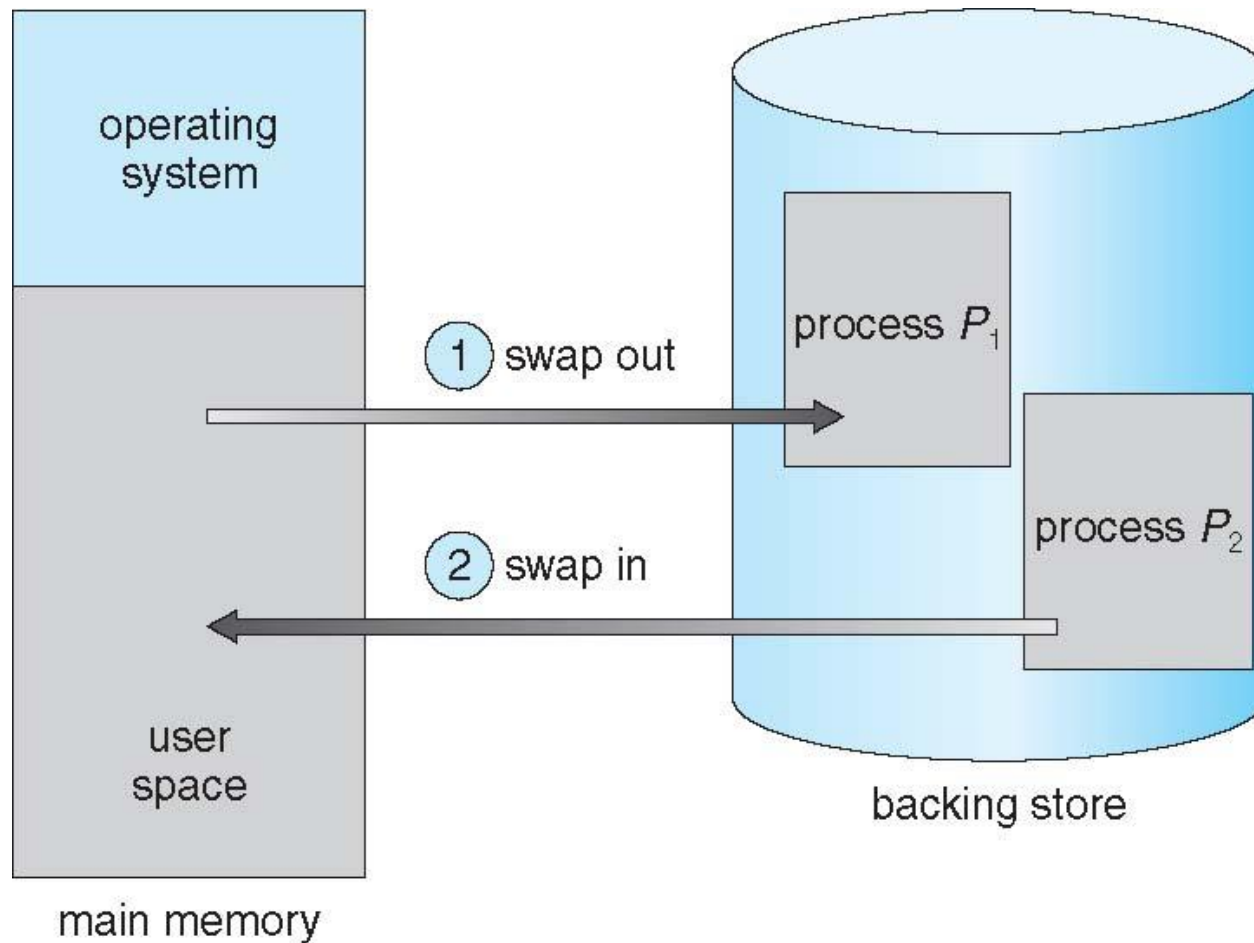


# Swapping

- ▶ A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- ▶ Does the swapped out process need to swap back in to the same physical addresses?
- ▶ Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold



# Schematic View of Swapping



# Swapping on Mobile Systems

- ▶ Not typically supported
  - Flash memory
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- ▶ Instead use other methods to free memory if it is low
  - iOS asks apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes application state to flash for fast restart



# Contiguous Allocation (1 / 2)

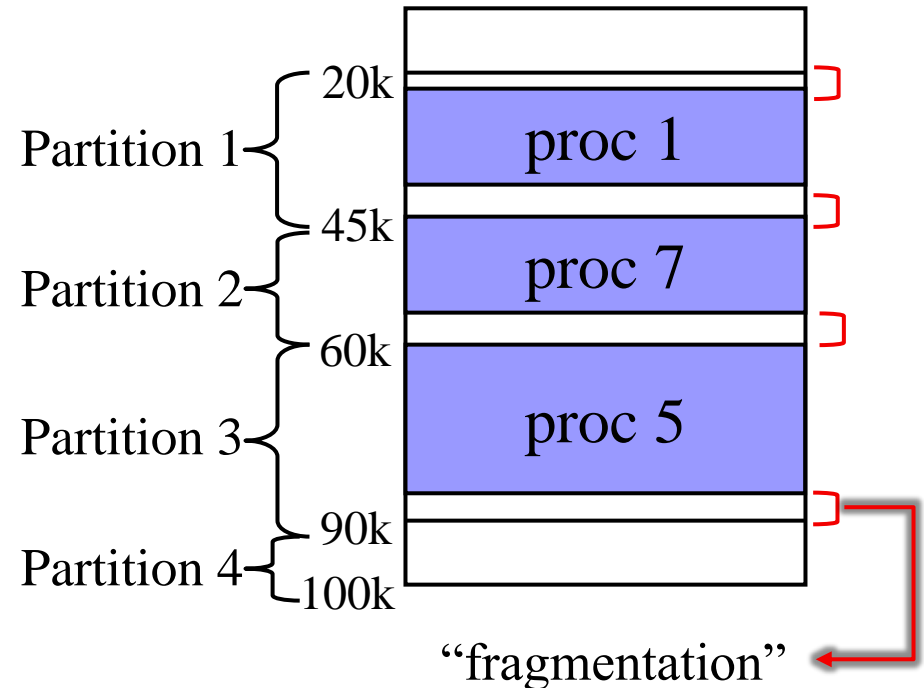
## ► Fixed Partitions

- Memory is divided into fixed partitions, e.g., OS/360
- A process is allocated on an entire partition

Partitions

number   size   location   status

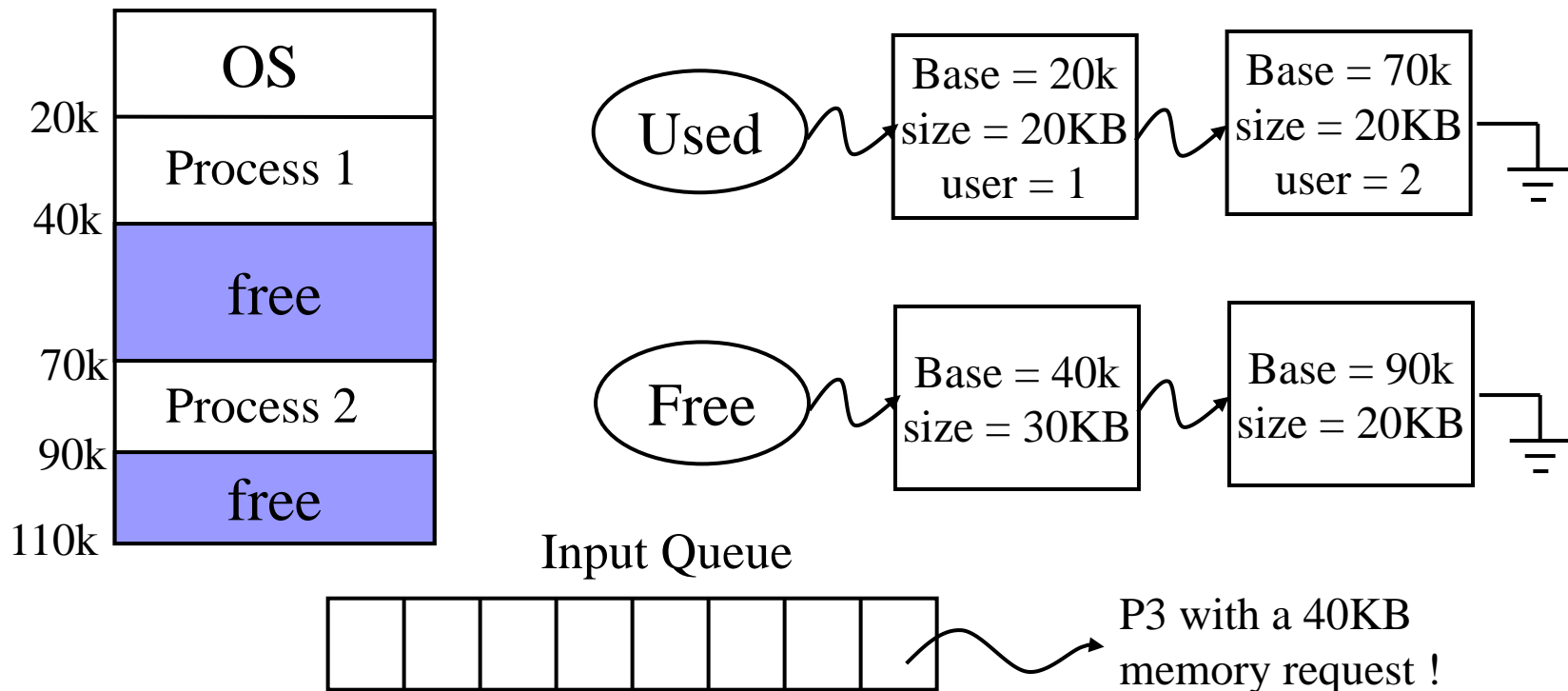
1	25KB	20k	Used
2	15KB	45k	Used
3	30KB	60k	Used
4	10KB	90k	Free



# Contiguous Allocation (2/2)

## ► Dynamic Partitions

- Partitions are dynamically created
- OS tables record free and used partitions





# Dynamic Allocation

- ▶ **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 are better than worst-fit in terms of speed and storage utilization

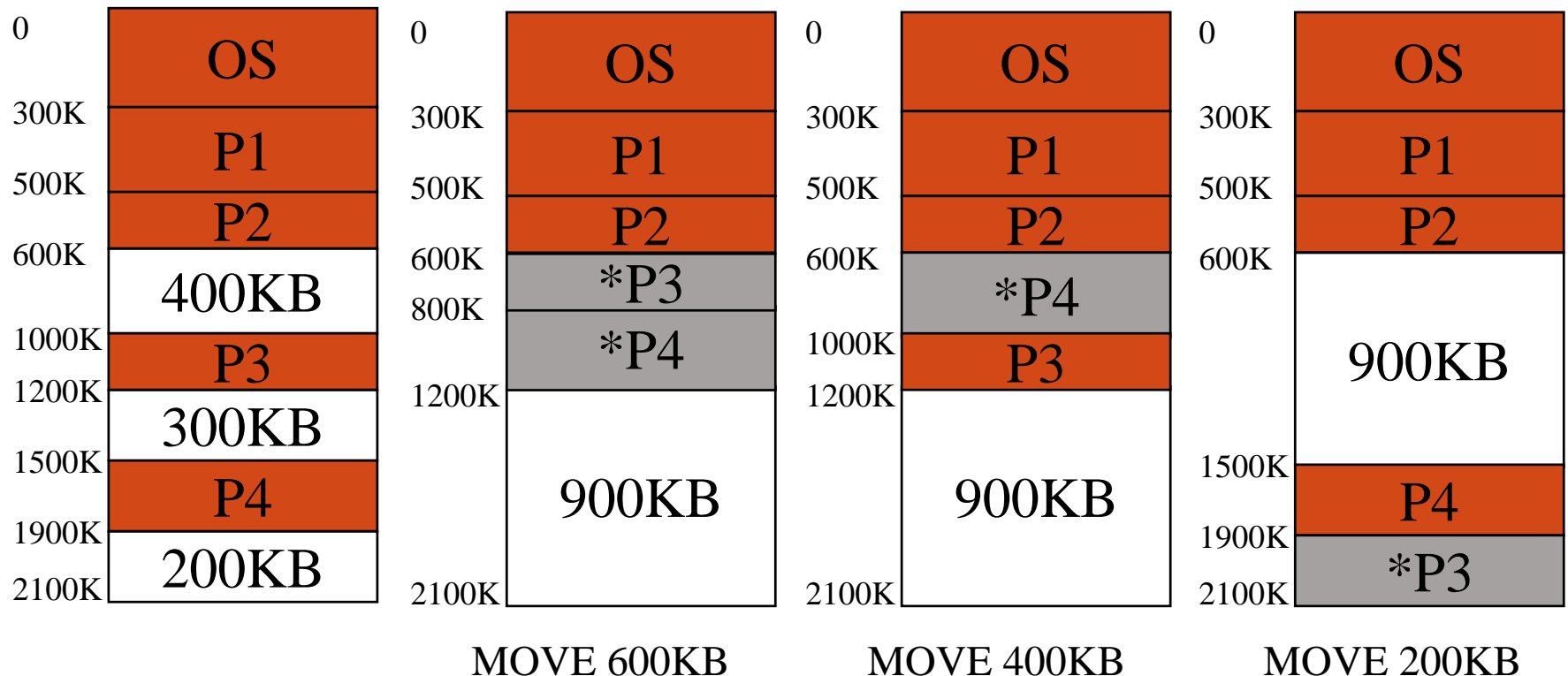


# Fragmentation

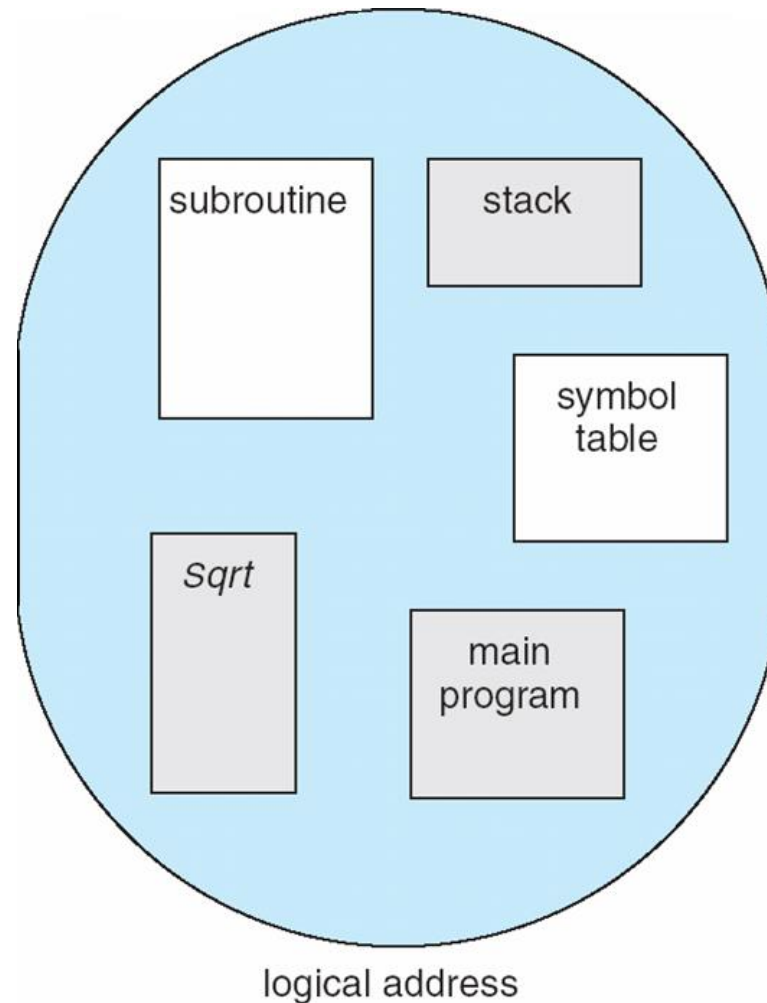
- ▶ **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- ▶ **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- ▶ First fit analysis reveals that given  $N$  blocks allocated,  $0.5 N$  blocks lost to fragmentation
  - $1/3$  may be unusable -> **50-percent rule**



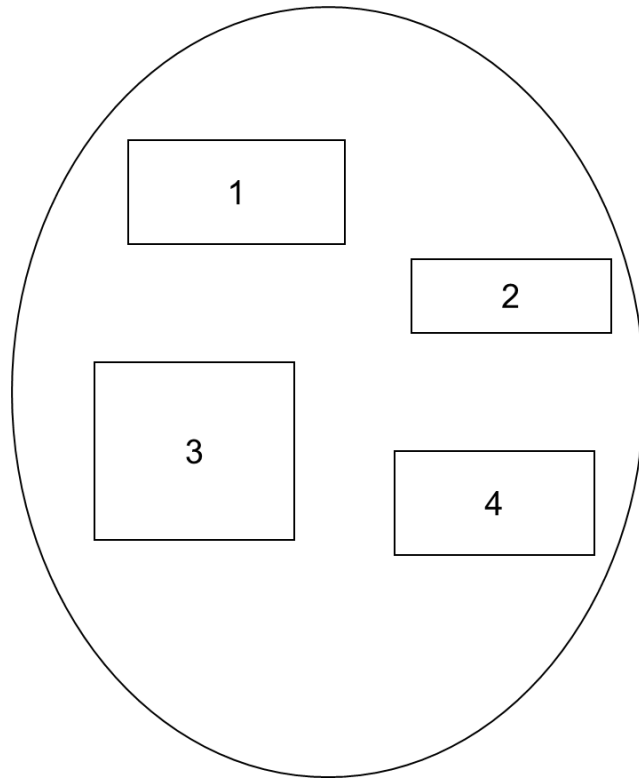
# Fragmentation — Compaction



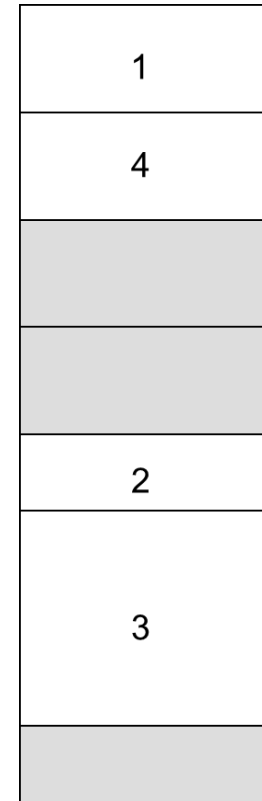
# User's View of a Program



# Logical View of Segmentation



user space

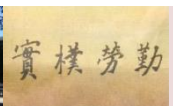


physical memory space

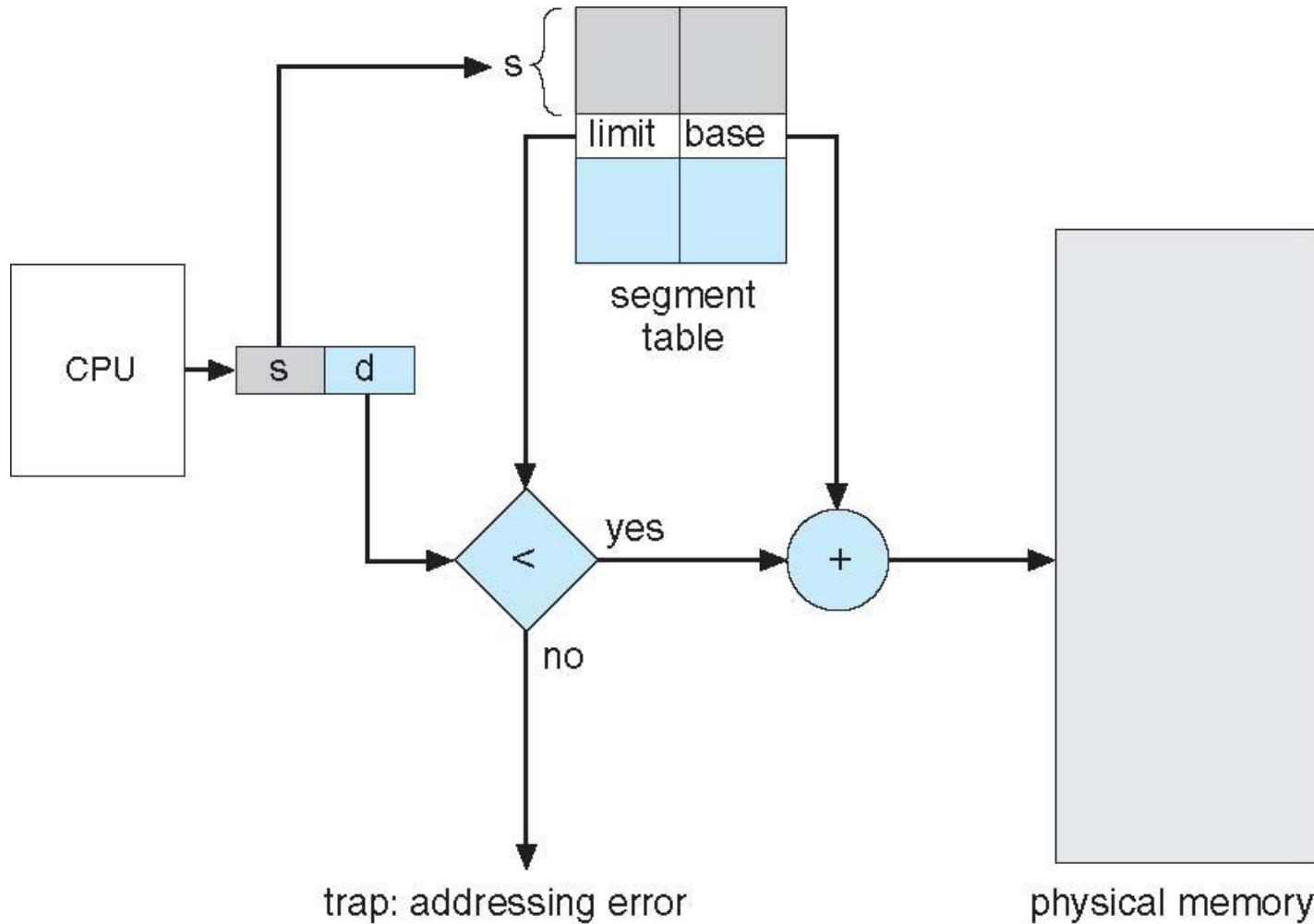


# Segmentation

- ▶ Segmentation is a memory management scheme that supports the user view of memory
  - A logical address space is a collection of segments with variable lengths
- ▶ Logical address consists of a 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



# Segmentation Architecture



# Paging

## ► Objective

- Users see a logically contiguous address space although its physical addresses are throughout physical memory

## ► Units of Memory and Backing Store

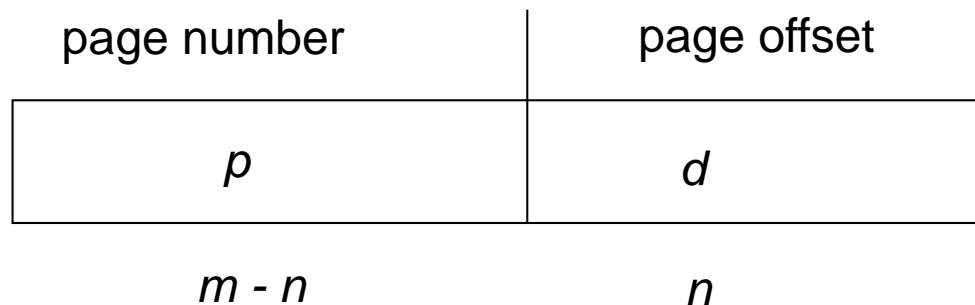
- Physical memory is divided into fixed-sized blocks called **frames**
- The logical memory space of each process is divided into blocks of the same size called **pages**
- The backing store is also divided into blocks of the same size if used





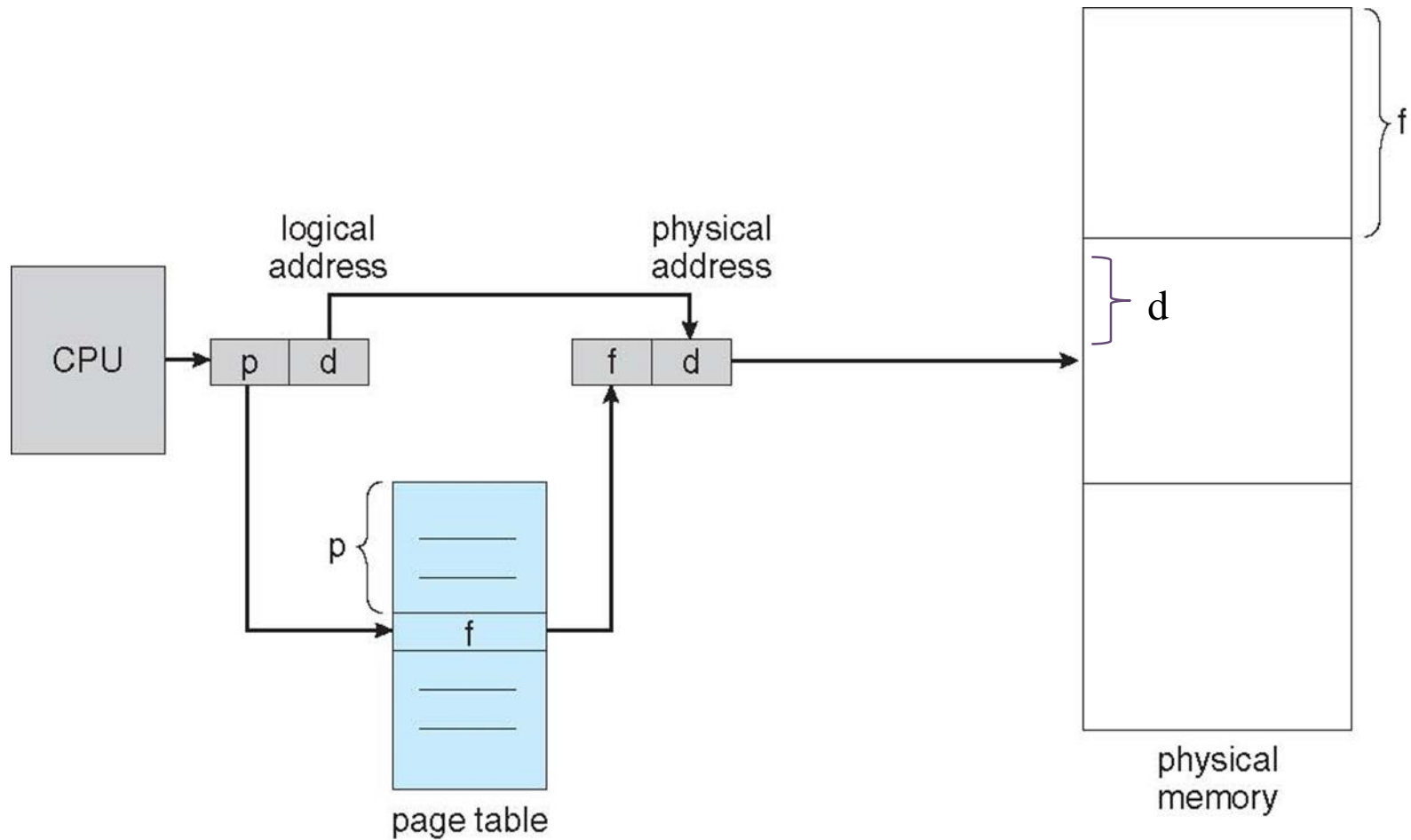
# Address Translation Scheme

- ▶ 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  $2^n$

# Paging Hardware



# Paging Model of Logical and Physical Memory

Page 0
Page 1
Page 2
Page 3

Logical  
Memory

0	1
1	4
2	3
3	7

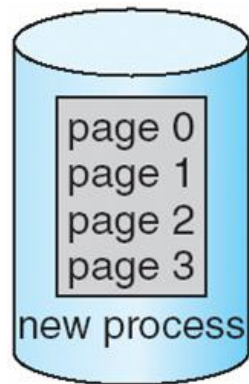
Page  
Table

0	
1	Page 0
2	
3	Page 2
4	Page 1
5	
6	
7	Page 3

Physical  
Memory



# Free Frames

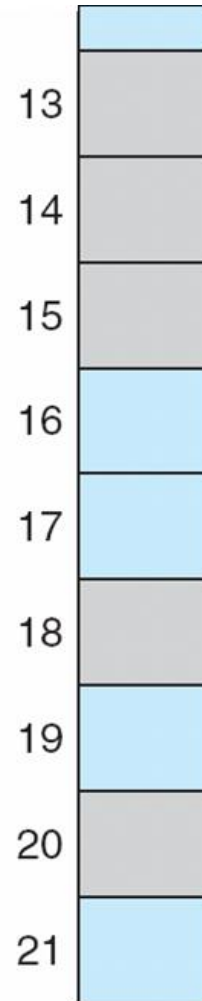


0	14
1	13
2	18
3	20

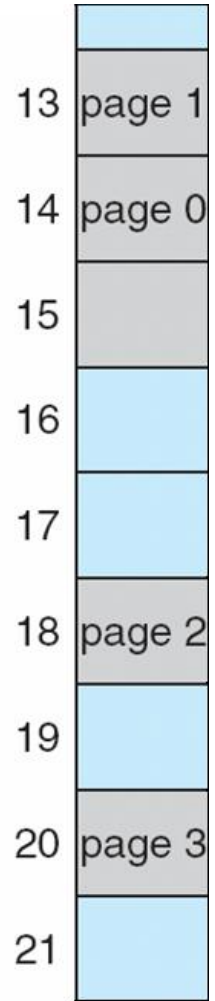
new-process page table

free-frame list

14  
13  
18  
20  
15



free-frame list  
15

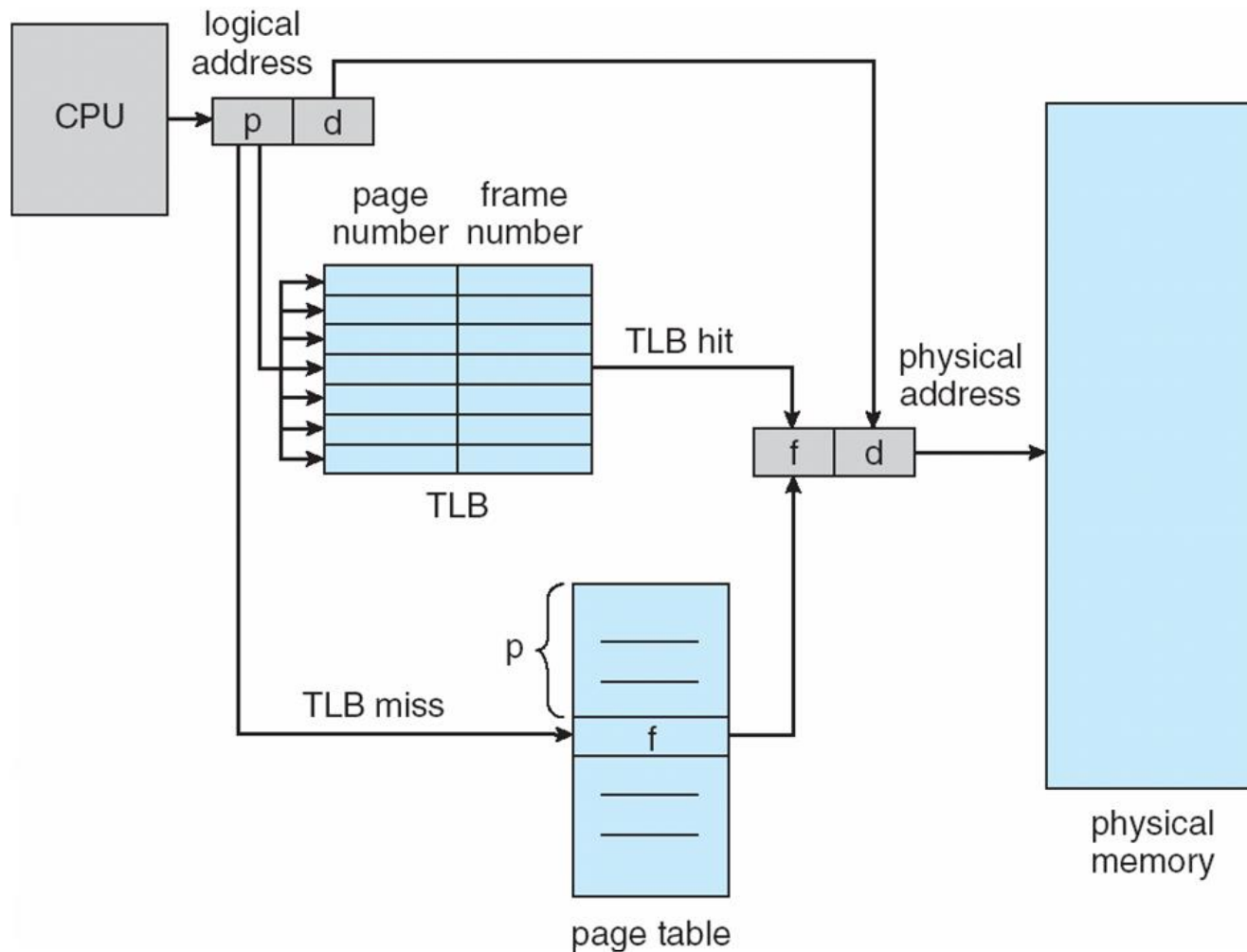


# Implementation of Page Table

- ▶ Page table is kept in main memory
- ▶ **Page-table base register (PTBR)** points to the page table
- ▶ **Page-table length register (PTLR)** indicates size of the page table
- ▶ The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **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
  - Otherwise need to flush at every context switch



# Paging Hardware With TLB



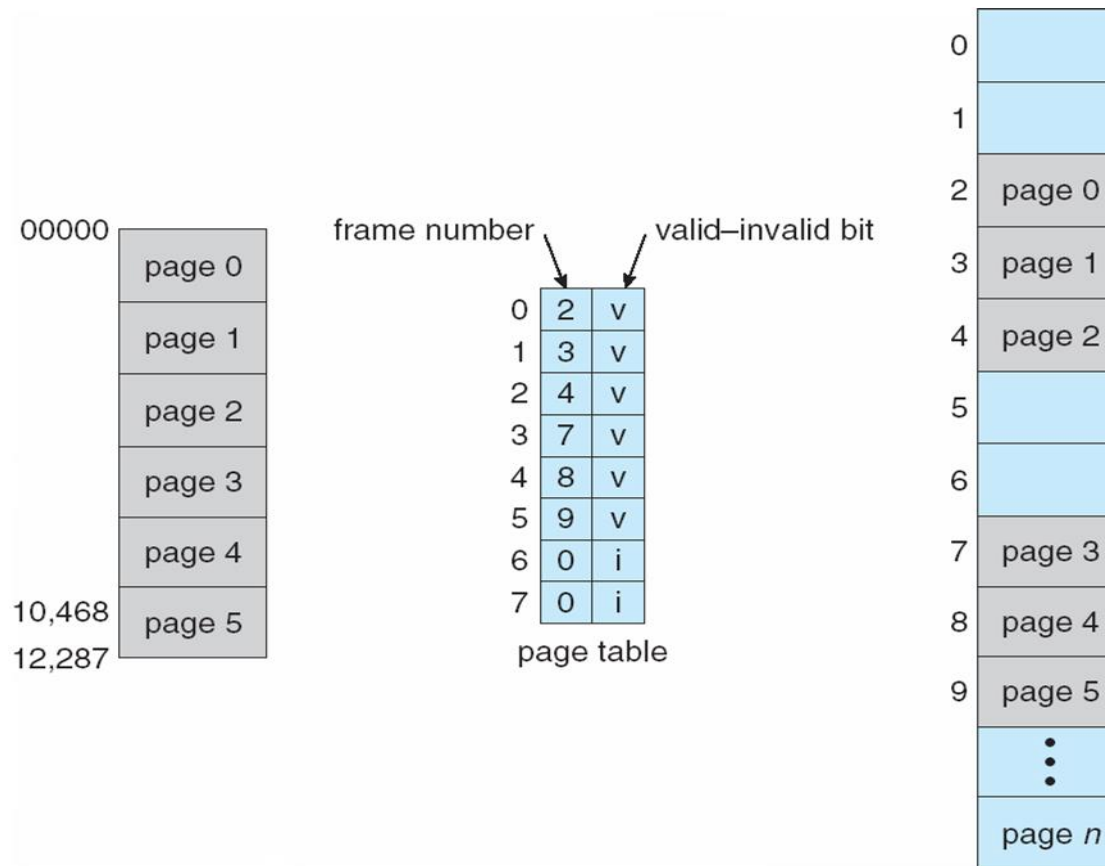
# Effective Access Time With TLB

- ▶ TLB Hit ratio =  $p$
- ▶ Consider  $p = 80\%$ , 100ns for memory access
  - Effective Access Time (EAT)  
 $= 0.80 \times 100 + 0.20 \times 200 = 120\text{ns}$
- ▶ Consider more realistic hit ratio  $p = 99\%$ , 100ns for memory access
  - $\text{EAT} = 0.99 \times 100 + 0.01 \times 200 = 101\text{ns}$



# Memory Protection

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





# Shared Pages

## ▶ Shared code

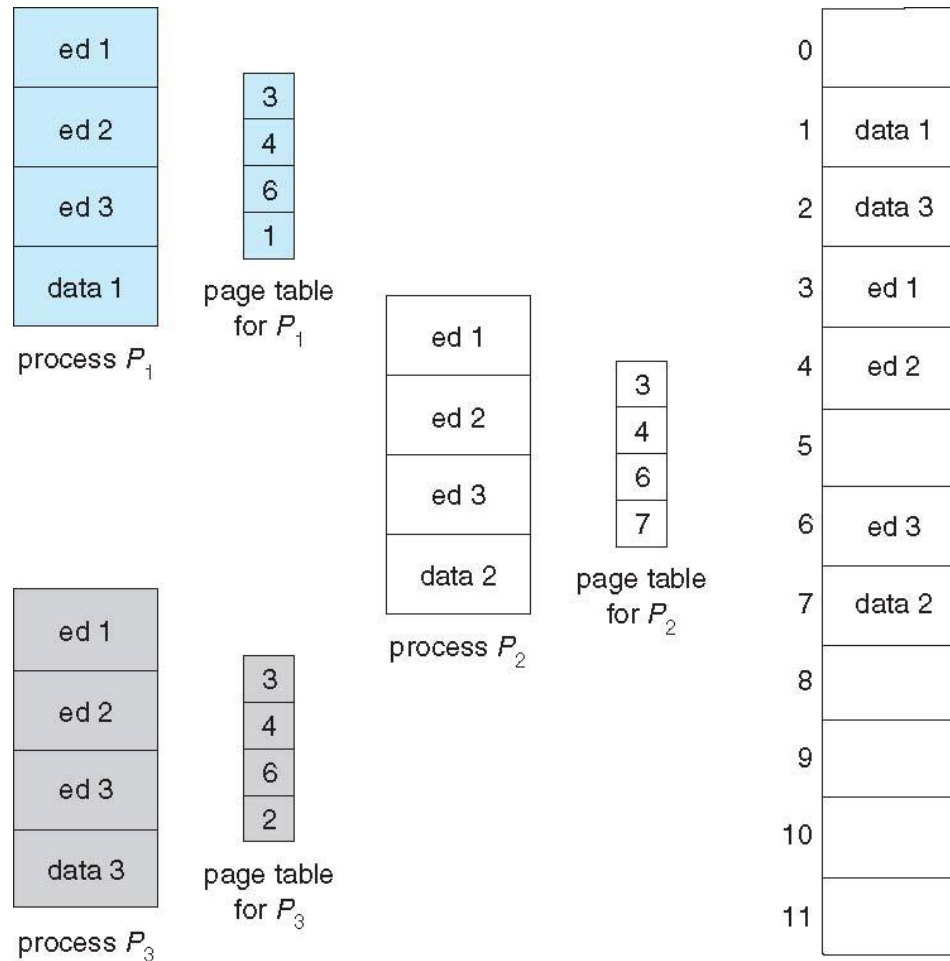
- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for inter-process communication if sharing of read-write pages is allowed

## ▶ 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



# An Example of Shared Pages



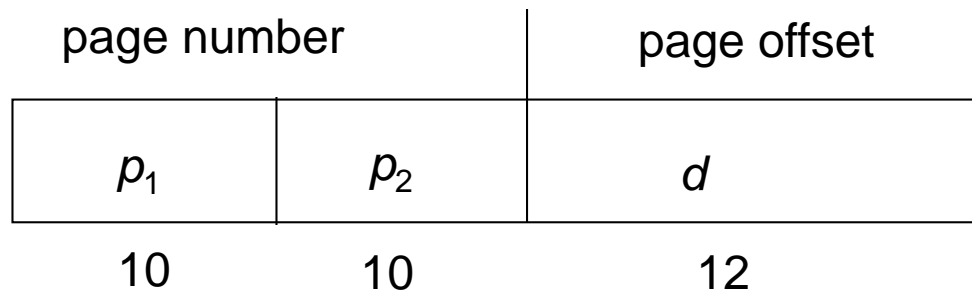
# Structure of the Page Table

- ▶ Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers, and the page size is 4 KB ( $2^{12}$ )
  - Page table would have 1 million entries ( $2^{32} / 2^{12}$ )
  - If each entry is 4 bytes → 4 MB of physical memory space for a page table
- ▶ Advanced structure of the page table
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables



# Two-Level Page-Table Scheme

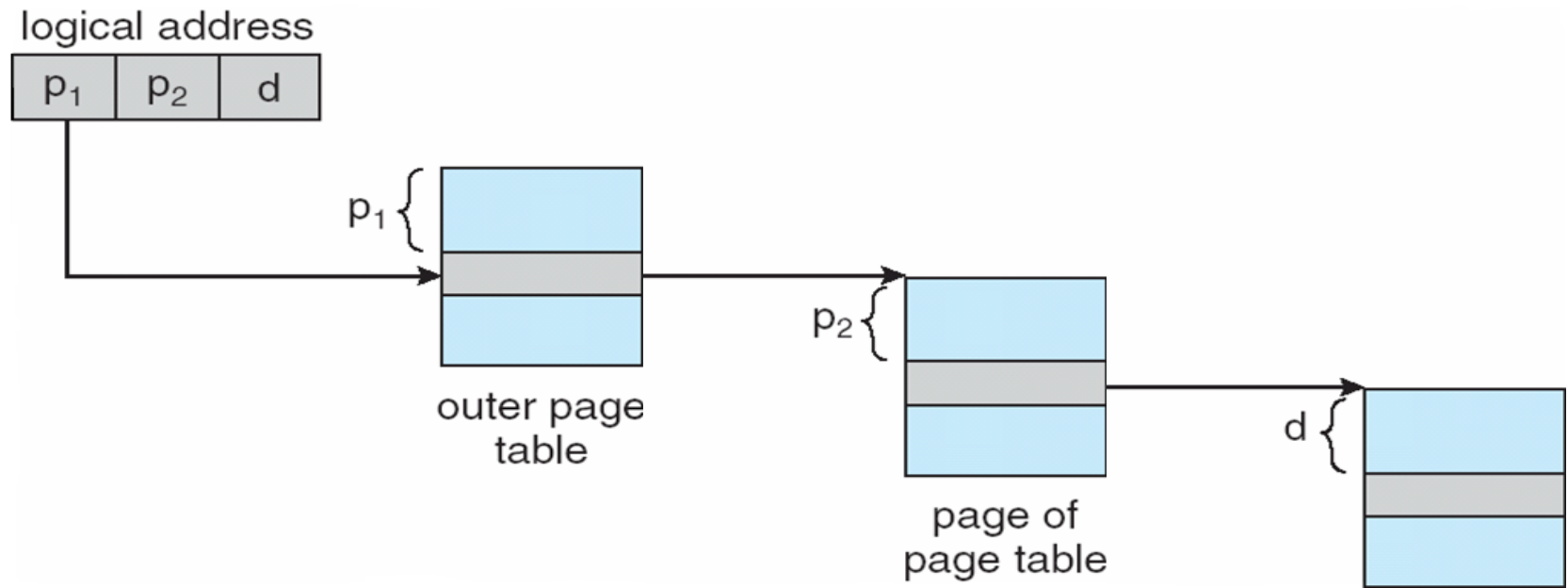
- ▶ A logical address on 32-bit machine with 4K page size is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- ▶ Thus, a logical address is as follows:



- ▶ Where  $p_1$  is an index into the outer page table, and  $p_2$  is the index into the inner page table



# Address Translation Scheme of Two-Level Paging

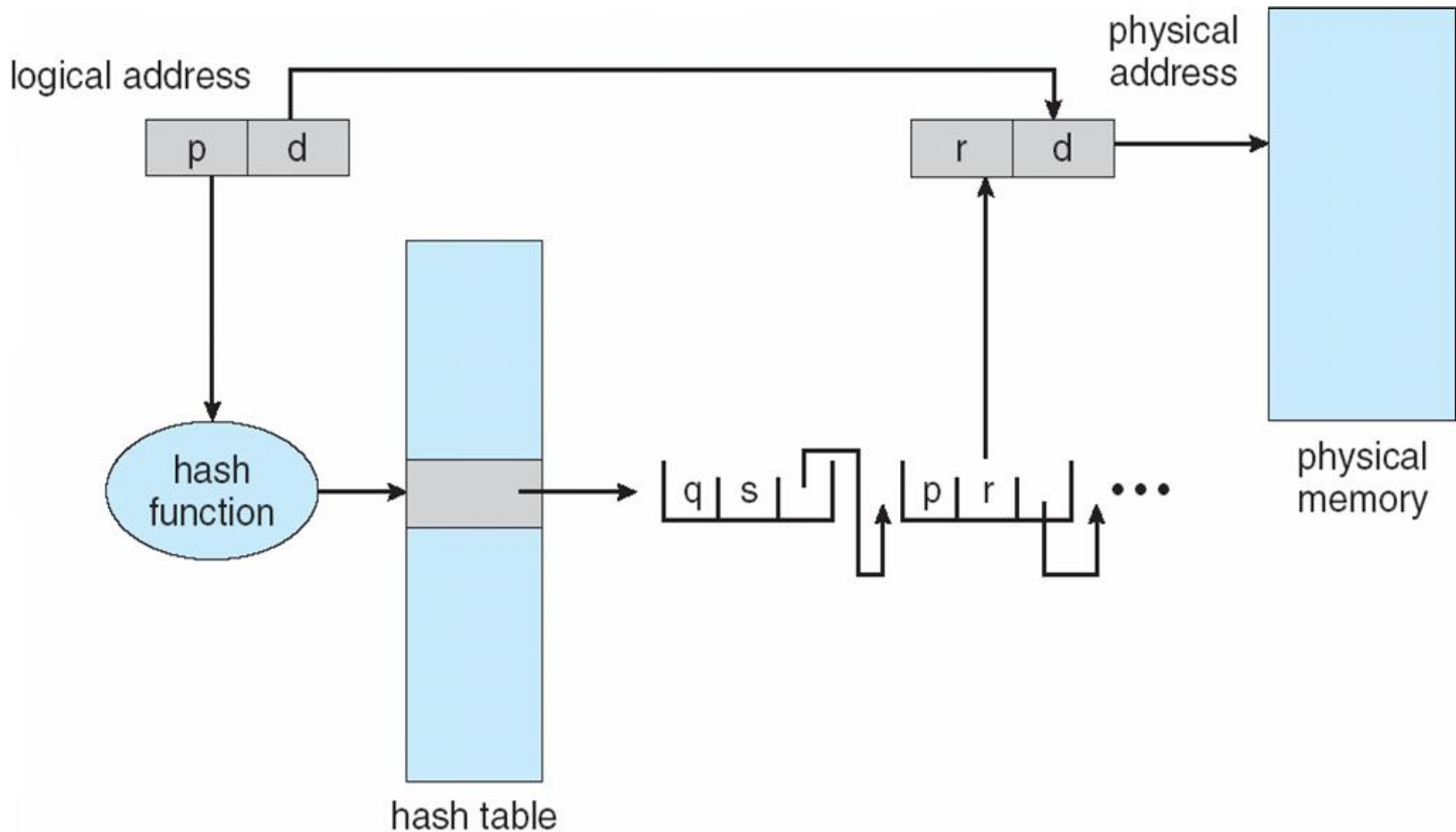


- ▶ The size of each table is 4KB if each entry has 4 Bytes
- ▶ The total size of the inner page tables is still 4MB, but each inner page table is created when it is used

# Hashed Page Tables (1 / 2)

- ▶ Objective:
  - To handle large address spaces
- ▶ Virtual address  $\rightarrow$  hash function  $\rightarrow$  a linked list of elements: (virtual page number, frame number, a pointer)
- ▶ Clustered Page Tables
  - Each entry contains the mappings for several physical-page frames, e.g., 16

# Hashed Page Tables (2/2)



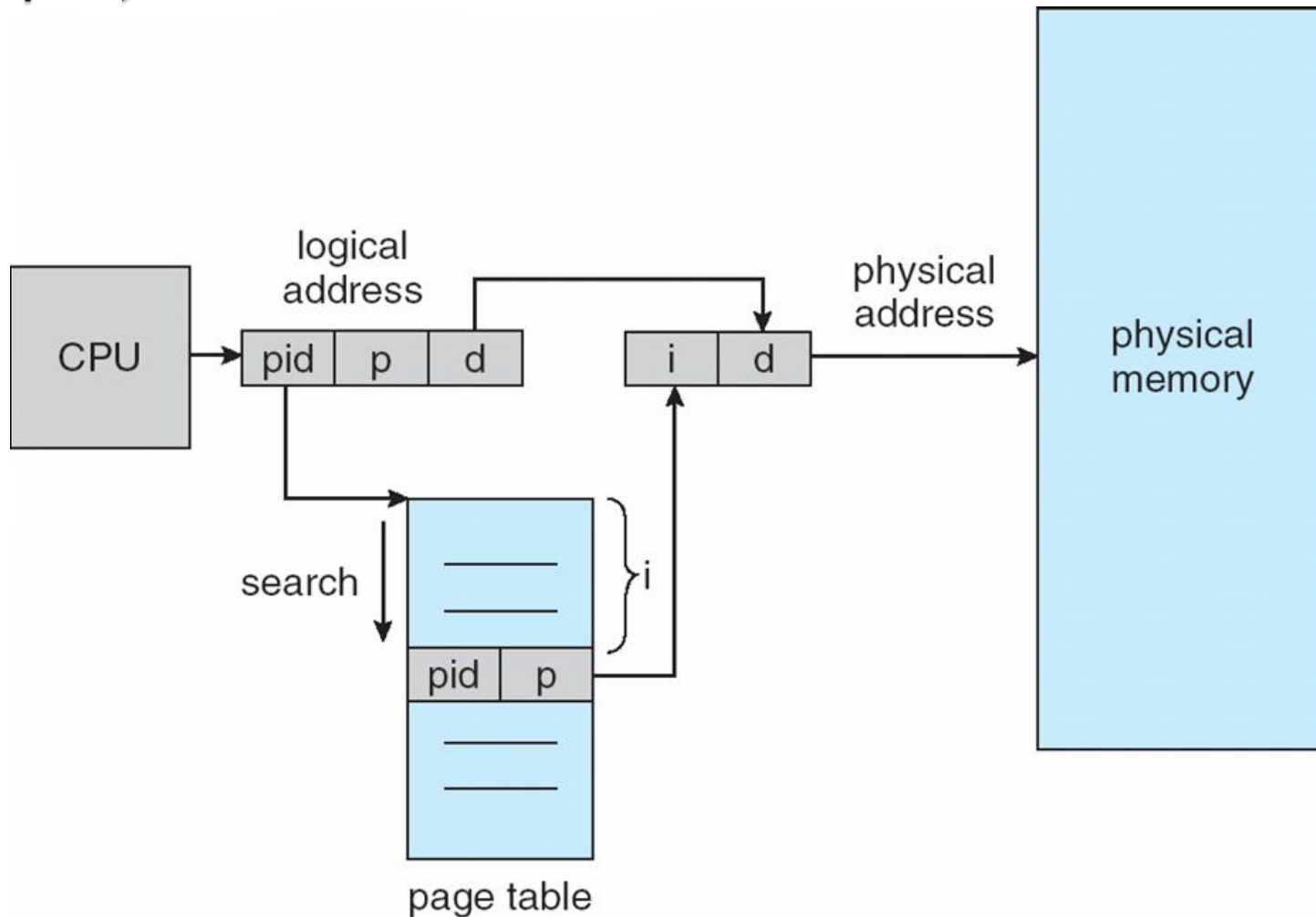
# Inverted Page Table Architecture (1 / 2)

- ▶ Only one page table for all processes
- ▶ Each entry corresponds to a physical frame.
  - Virtual Address: <Process ID, Page Number, Offset>
- Long search time to find out the match
- Difficult to implement with shared memory



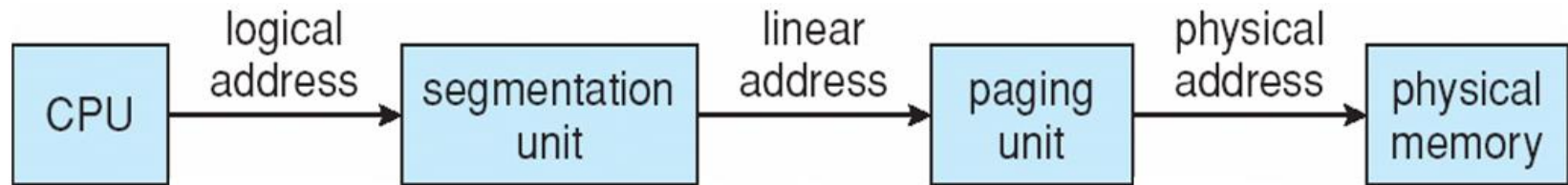


# Inverted Page Table Architecture (2/2)



# Example: The Intel IA-32 Architecture

- ▶ Supports both segmentation and paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process



- ▶ Two-level paging

