# W4118 Operating Systems

Instructor: Junfeng Yang

# Outline

- Paging
  - Overview
  - Page translation
  - Page allocation
  - Page protection
  - Translation Look-aside Buffers (TLB)
  - Page sharing
  - Page table structure
  - Combining paging with segmentation

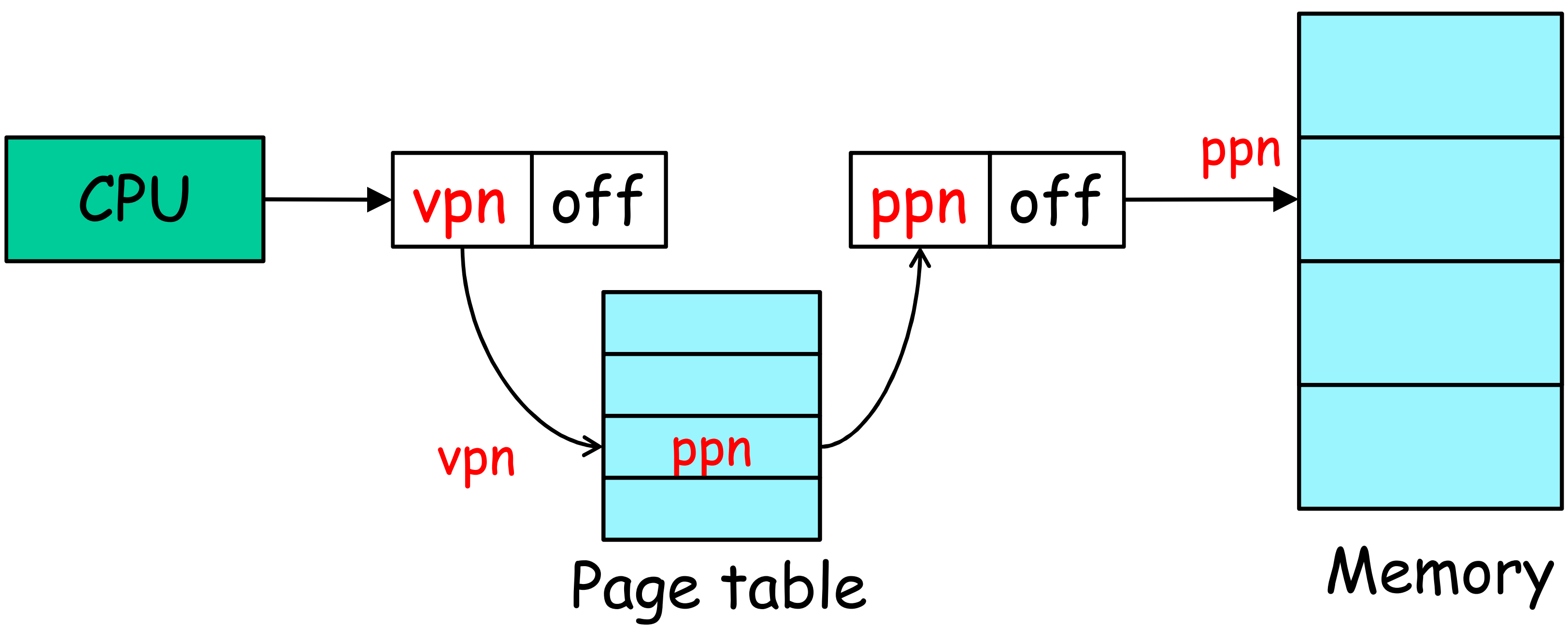
# Paging overview

- Goal
  - Eliminate external fragmentation
  - Don't allocate memory that will not be used
  - Enable sharing
- Paging: divide memory into fixed-sized pages
  - Both virtual and physical memory are composed of pages
- Another terminology
  - A virtual page: page
  - A physical page: frame

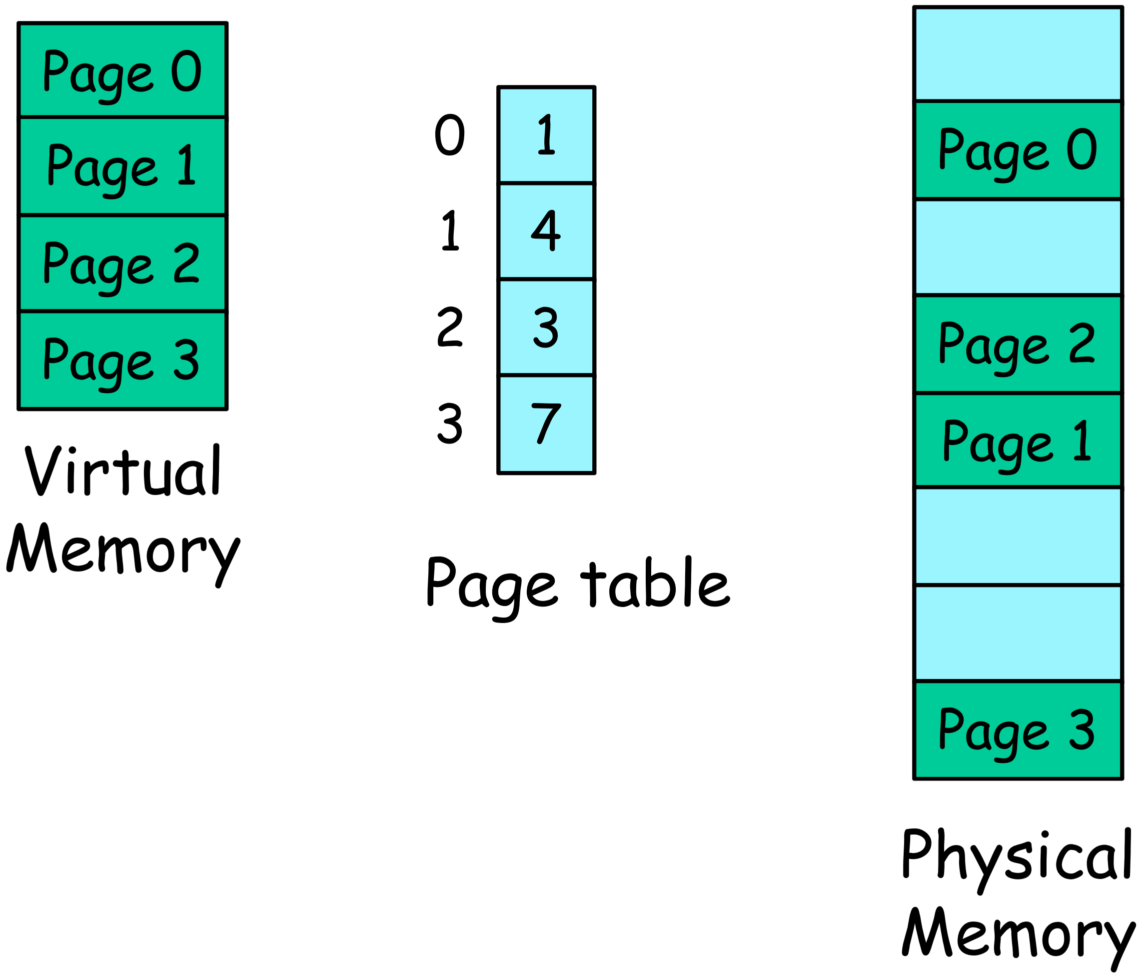
# Page translation

- Address bits = page number + page offset
- Translate virtual page number (vpn) to physical page number (ppn) using page table

  pa = page_table[va/pg_sz] + va%pg_sz



# Page translation example



# Page translation exercise

- 8-bit virtual address, 10-bit physical address, and each page is 64 bytes
  - How many virtual pages?
  - How many physical pages?
  - How many entries in page table?
  - Given page table = [2, 5, 1, 8], what's the physical address for virtual address 241?

- m-bit virtual address, n-bit physical address, k-bit page size
  - What are the answers to the above questions?

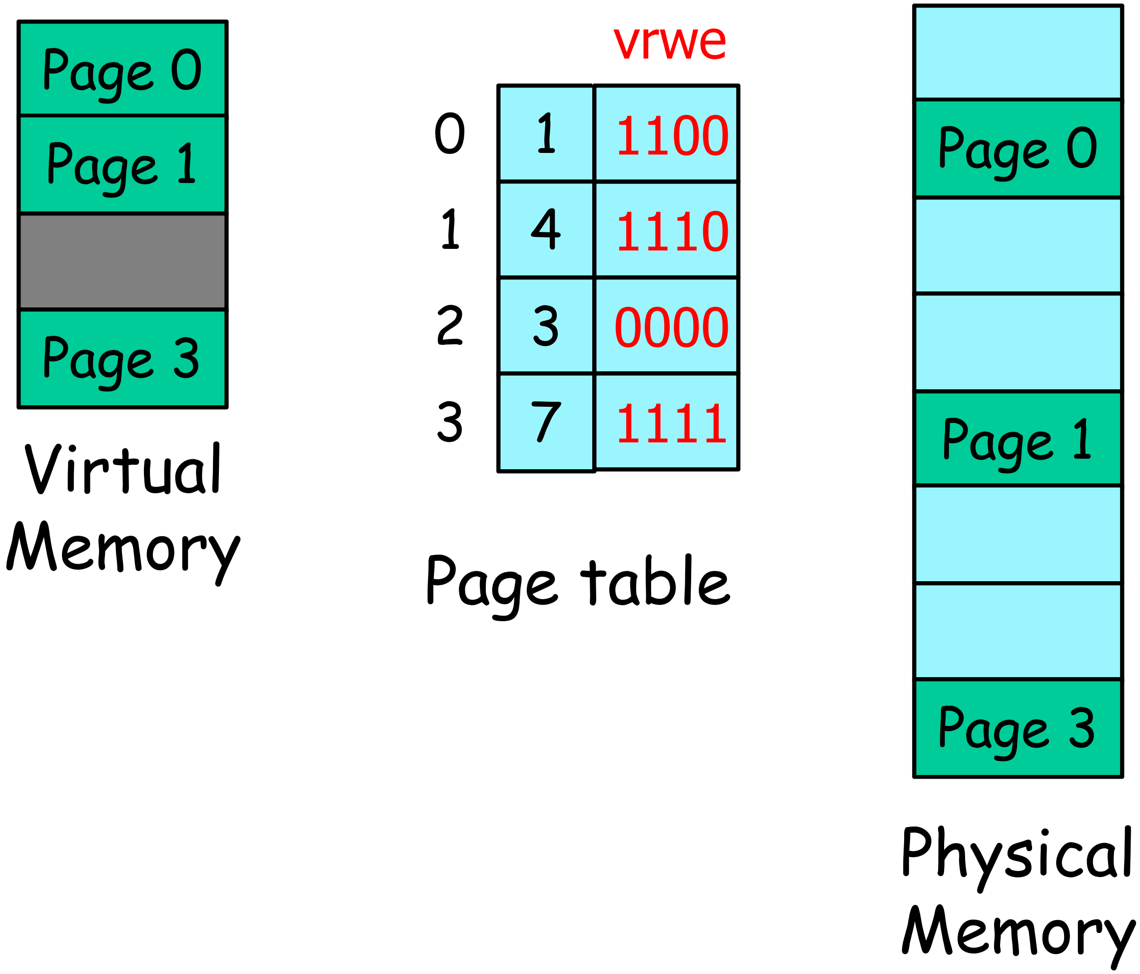
# Page protection

- Implemented by associating protection bits with each virtual page in page table

- Protection bits
  - valid bit: map to a valid physical page?
  - read/write/execute bits: can read/write/execute?

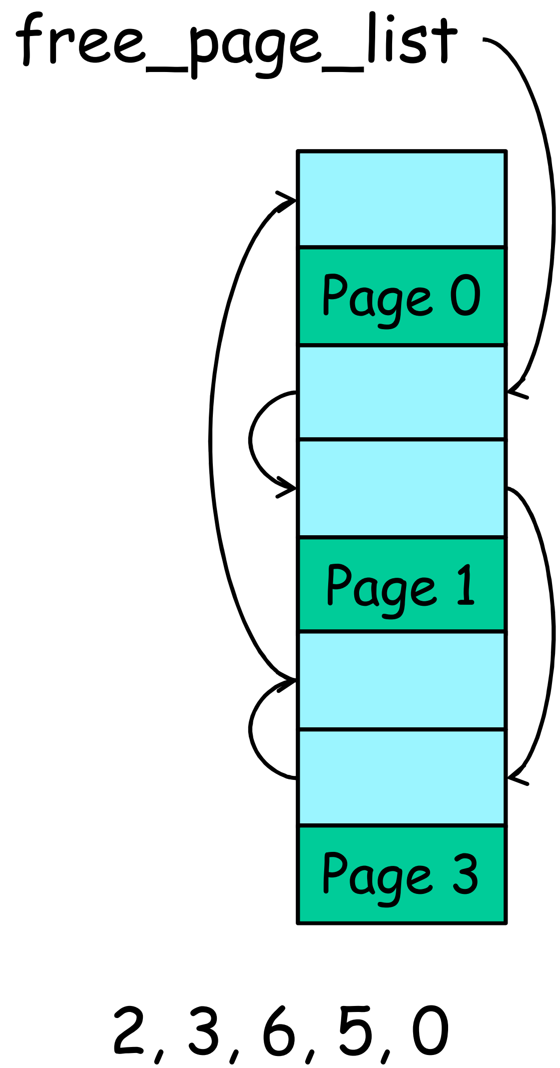
- Checked by MMU on each memory access

# Page protection example



# Page allocation

- Free page management
  - E.g., can put page on a free list
- Allocation policy
  - E.g., one page at a time, from head of free list



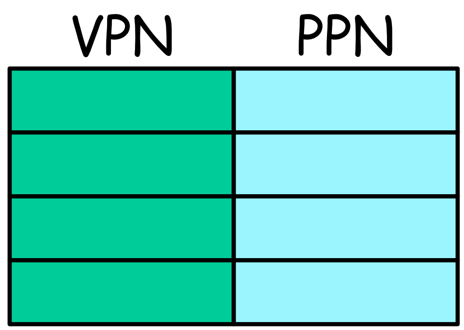
# Implementation of page table

- Page table is stored in memory
  - Page table base register (PTBR) points to the base of page table
  - OS stores the value of this register in process control block (PCB)
  - OS switches PTBR on each context switch

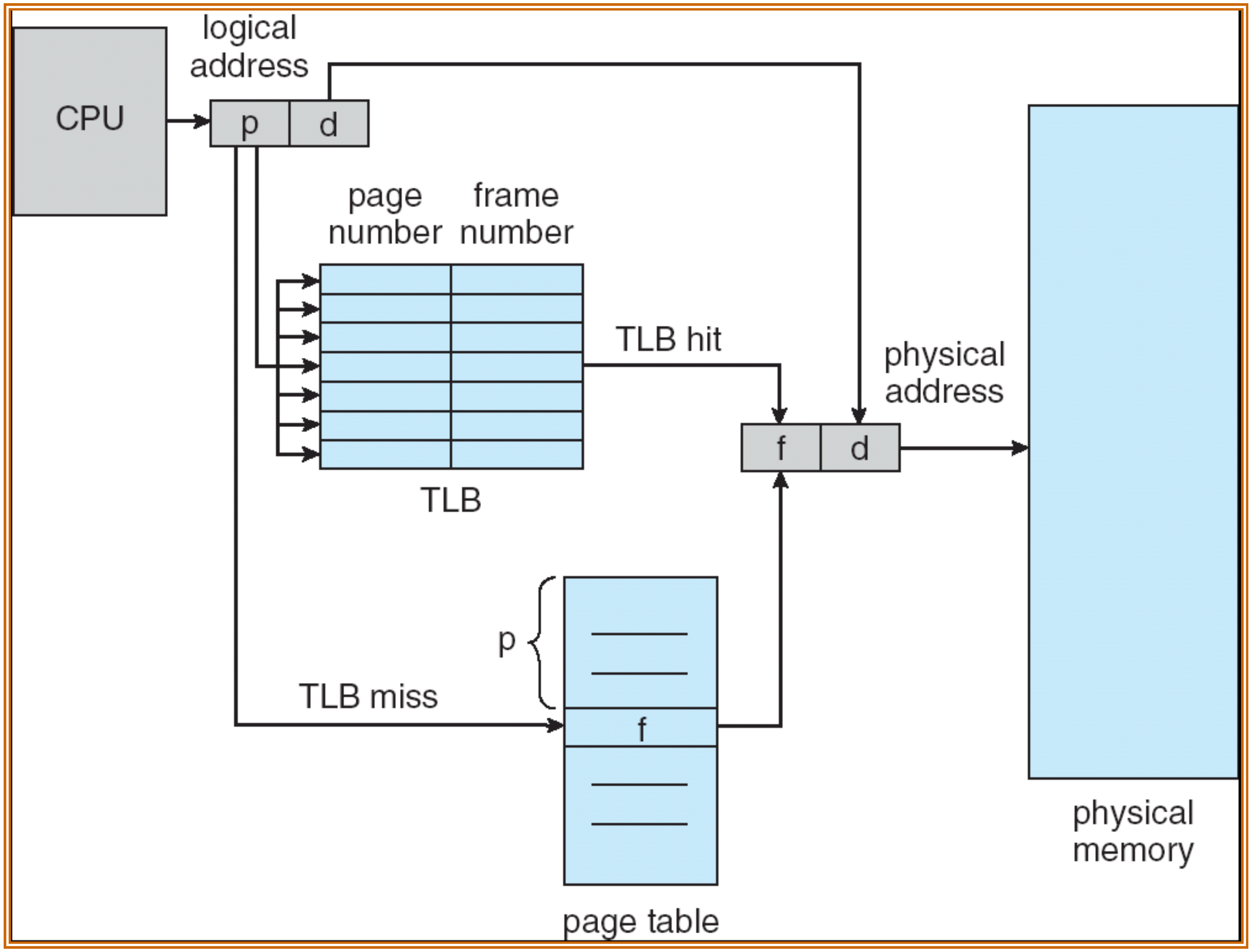
- Problem: each data/instruction access requires two memory accesses
  - Extra memory access for page table

# Avoiding extra memory access

- Fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)
- Fast parallel search (CPU speed)
- Small



# Paging hardware with TLB



# TLB Miss

- Can be handled in hardware and software
- Hardware (CISC: x86)
  - Pros: hardware doesn't have to trust OS !
  - Cons: complexity
- Software (RISC: MIPS, SPARC)
  - Pros: flexibility
  - Cons: code may have bug
  - Question: what can't a TLB miss handler do?

# TLB and context switches

- What happens to TLB on context switches?

- Option 1: flush entire TLB
  - x86

- Option 2: attach process ID to TLB entries
  - ASID: Address Space Identifier
  - MIPS, SPARC

# Effective access time

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

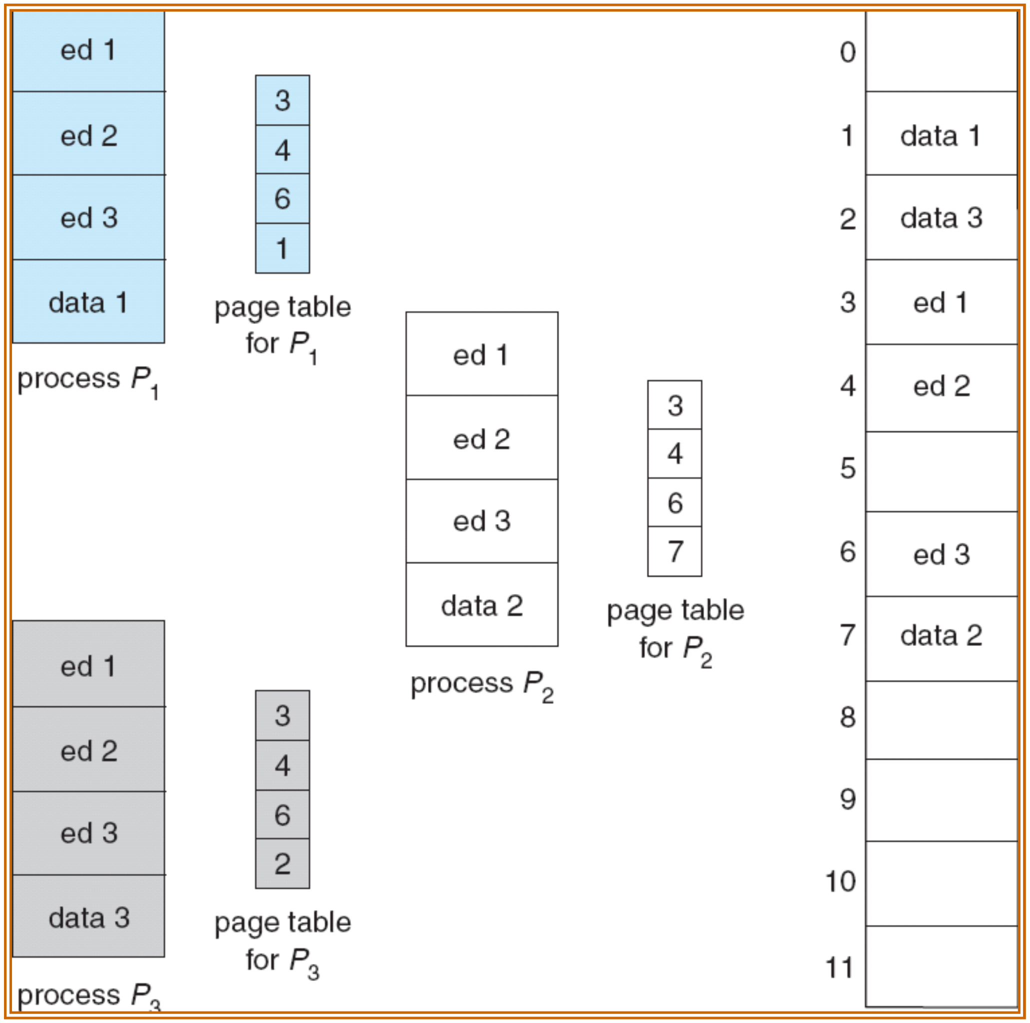
$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= \alpha + \varepsilon\alpha + 2 + \varepsilon - \varepsilon\alpha - 2\alpha$$
$$= 2 + \varepsilon - \alpha$$

# Motivation for page sharing

- Efficient communication. Processes communicate by write to shared pages

- Memory efficiency. One copy of read-only code/data shared among processes
  - Example 1: multiple instances of the shell program
  - Example 2: parent and forked child share AS

# Page sharing example



# Page table size issues

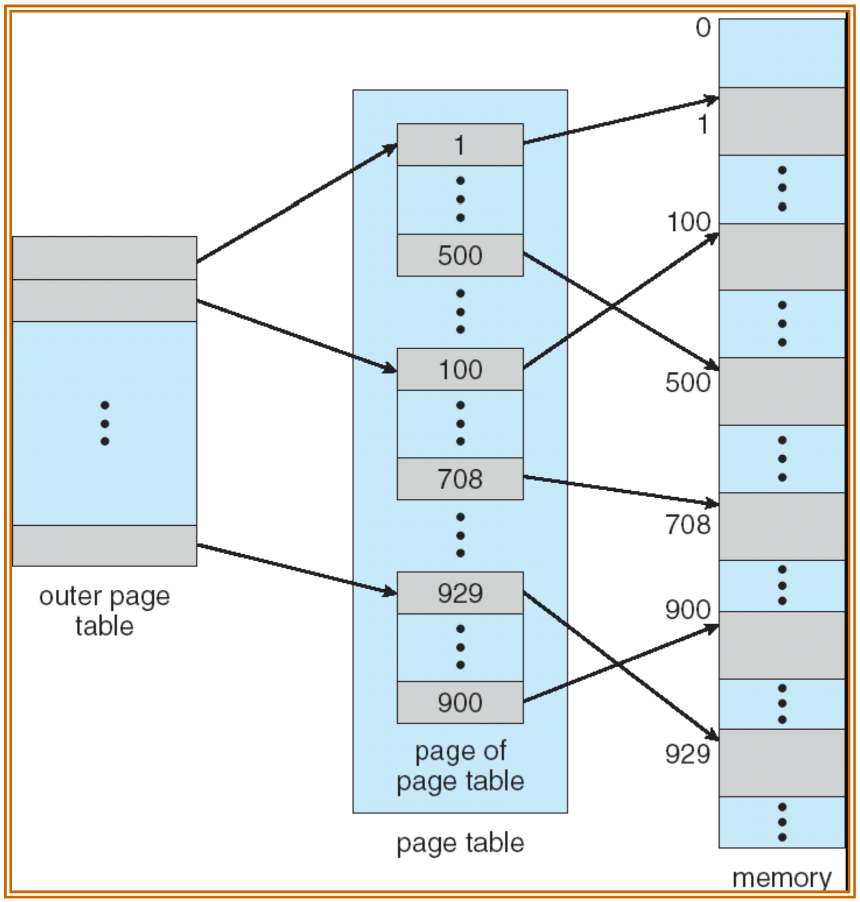
- Given:
  - A 32 bit address space (4 GB)
  - 4 KB pages
  - A page table entry of 4 bytes
- Implication: page table is 4 MB per process!
- Observation: address space are often sparse
  - Few programs use all of 2^32 bytes
- Change page table structures to save memory
  - Trade translation time for page table space

# Page table structures

- Hierarchical paging
- Hashed page tables
- Inverted page tables

# Hierarchical page table

- Break up virtual address space into multiple page tables at different levels

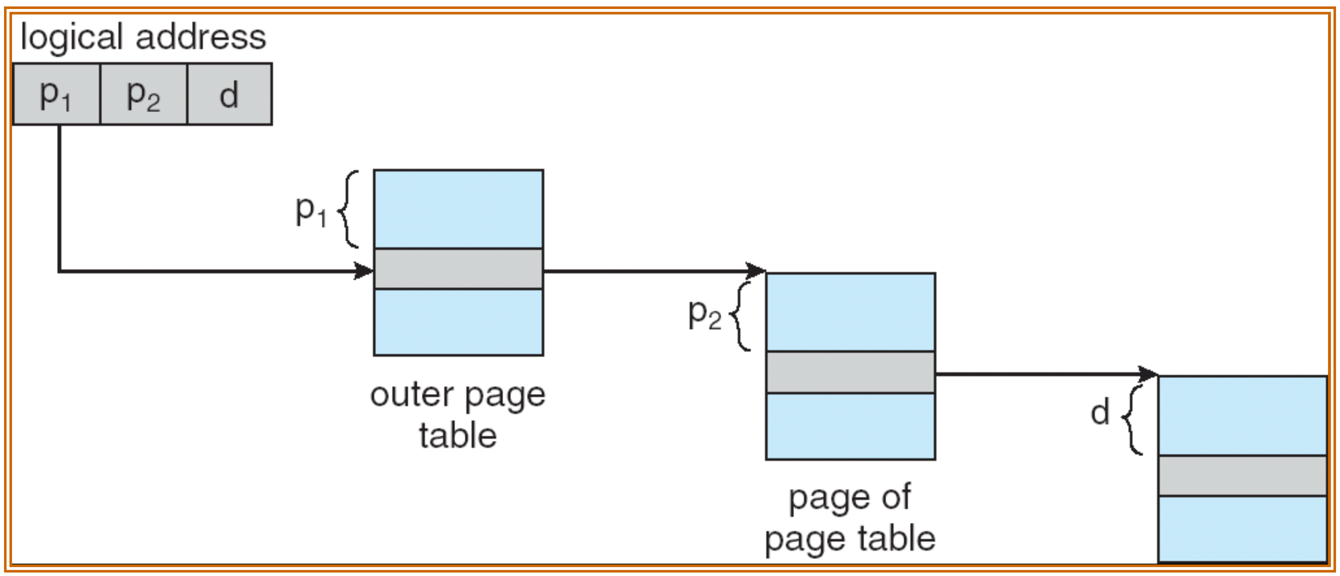


# Two-level paging example

- 32-bit address space, 4 KB page
  - 4KB page ➔ 12 bits for page offset
- How many bits for 2nd-level page table?
  - Desirable to fit a 2nd-level page table in one page
  - 4KB/4B = 1024 ➔ 10 bits for 2nd-level page table
- Address bits for top-level page table: 32 – 12 – 12 = 10

| page number | | page offset |
|---|---|---|
| $p_i$ | $p_2$ | $d$ |
| 10 | 10 | 12 |

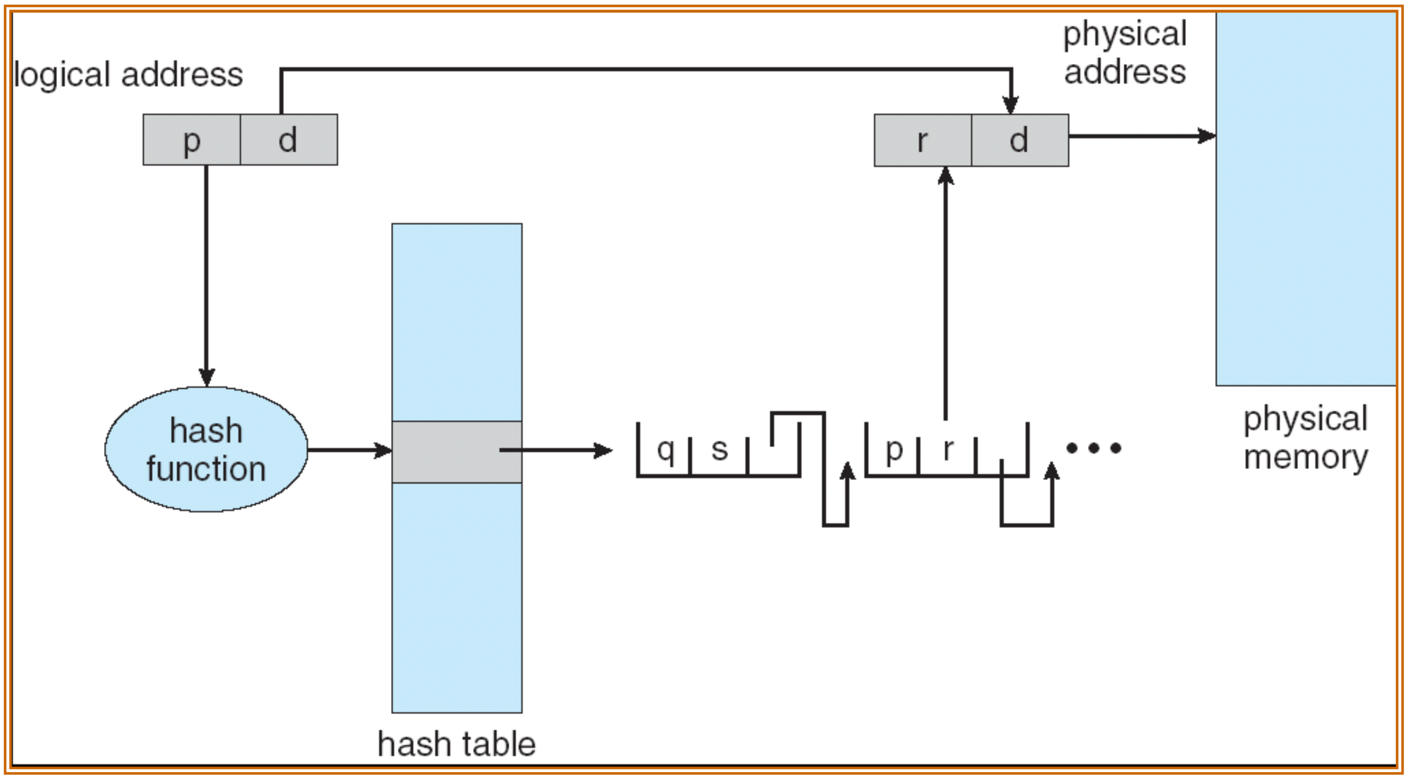
# Address-translation scheme



# Hashed page table

- Common in address spaces > 32 bits
- Page table contains a chain of elements hashing to the same location
- On page translation
  - Hash virtual page number into page table
  - Search chain for a match on virtual page number

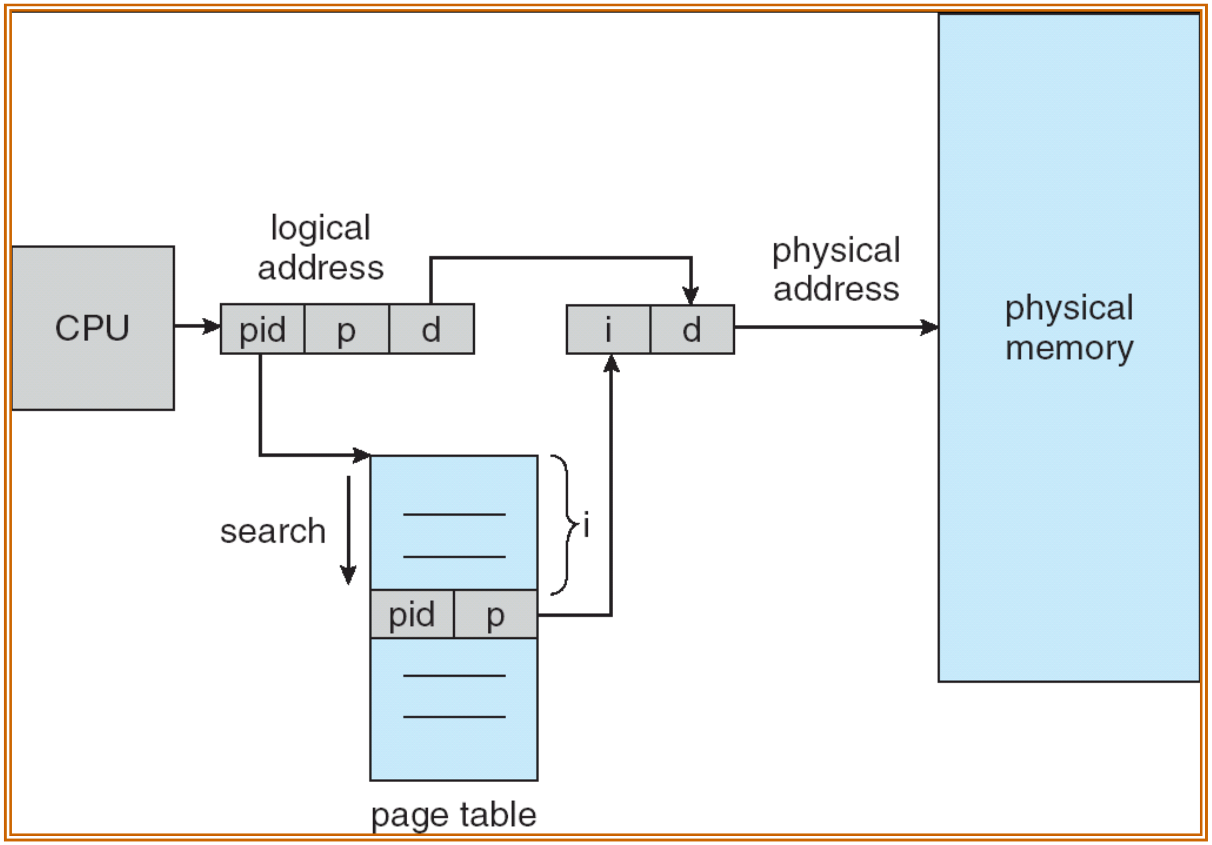
# Hashed page table example



# 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
- Can use hash table to limit the search to one or at most a few page-table entries

# Inverted page table example



# Combine paging and segmentation

- Structure
  - Segments: logical units in program, such as code, data, and stack
    - Size varies; can be large
  - Each segment contains one or more pages
    - Pages have fixed size

- Two levels of mapping to reduce page table size
  - Page table for each segment
  - Base and limit for each page table
  - Similar to multi-level page table

- Logical address divided into three portions

| | | |
|---|---|---|
| seg # | page # | offset |