

# W4118: segmentation and paging



Instructor: Junfeng Yang

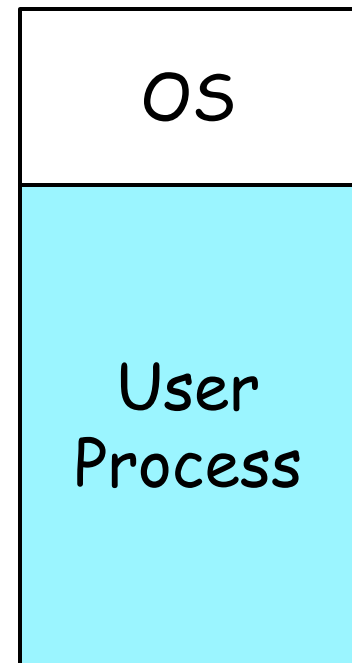
References: Modern Operating Systems (3<sup>rd</sup> edition), Operating Systems Concepts (8<sup>th</sup> edition), previous W4118, and OS at MIT, Stanford, and UWisc

# Outline

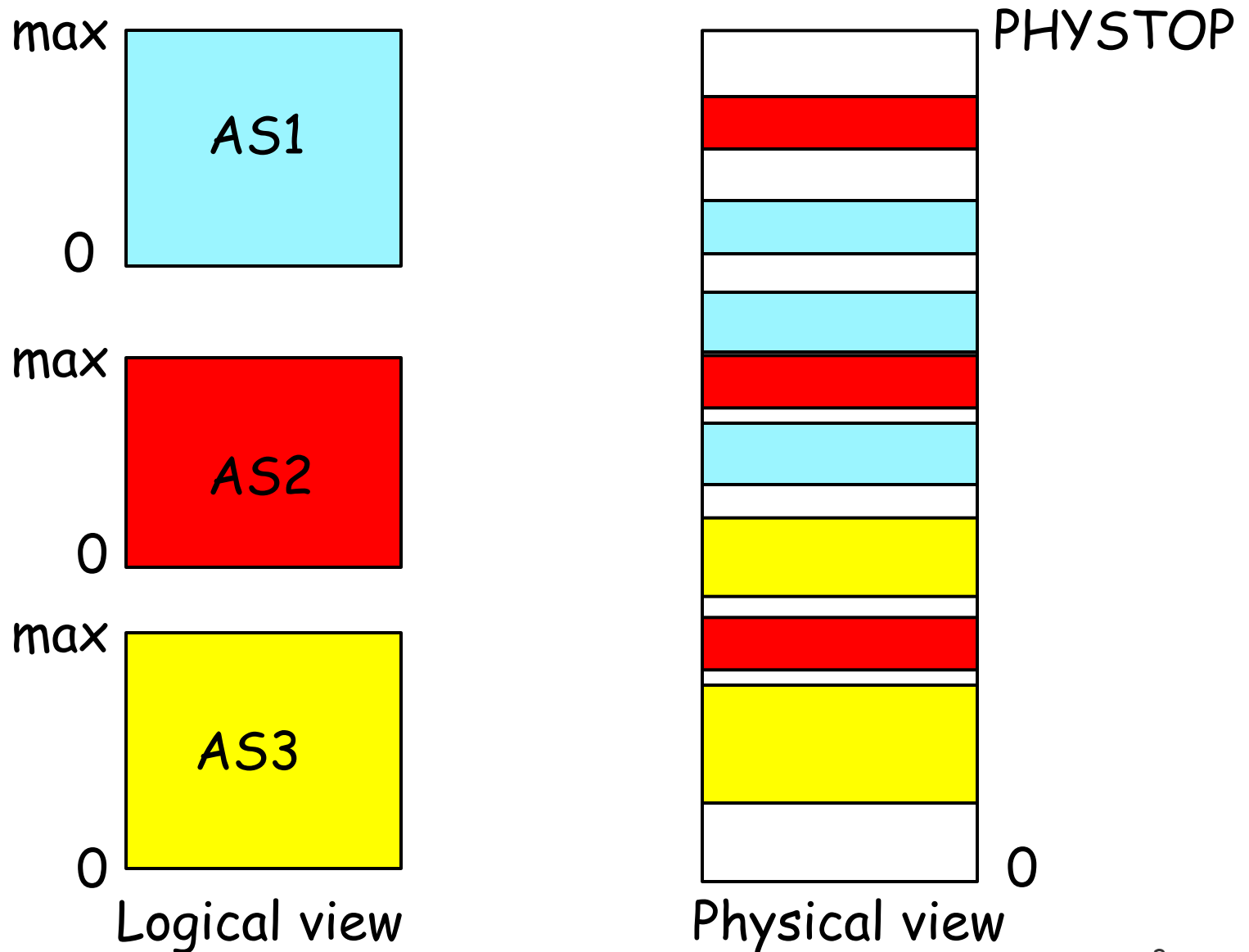
- ❑ Memory management goals
- ❑ Segmentation
- ❑ Paging
- ❑ TLB

# Uni- v.s. multi-programming

- ❑ Simple uniprogramming with a single segment per process
- ❑ Uniprogramming disadvantages
  - Only one process can run a time
  - Process can destroy OS
- ❑ Want multiprogramming!



# Multiple address spaces co-exist



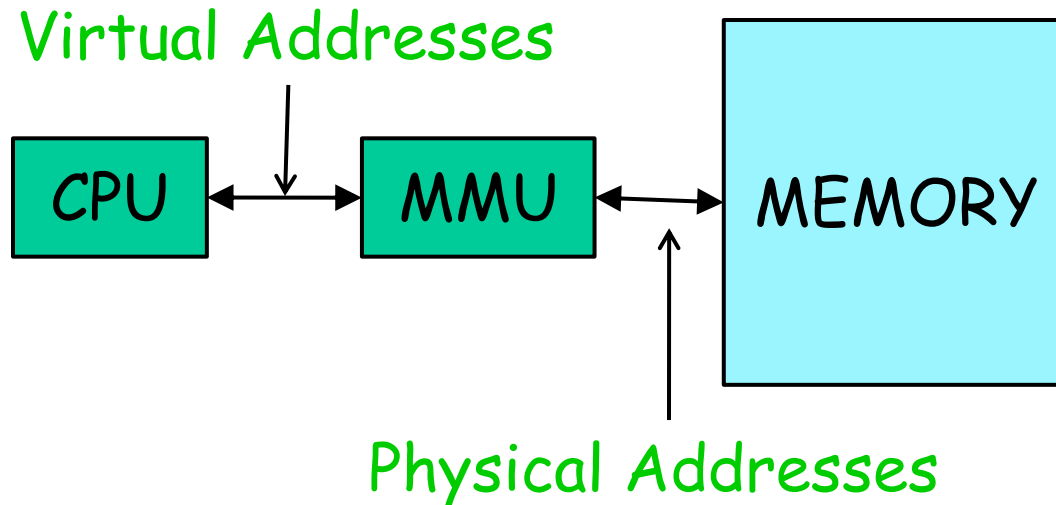
# Memory management wish-list

- ❑ Sharing
  - multiple processes **coexist** in main memory
- ❑ Transparency
  - Processes **are not aware** that memory is shared
  - Run **regardless of number/locations** of other processes
- ❑ Protection
  - **Cannot access** data of OS or other processes
- ❑ Efficiency: should have reasonable performance
  - Purpose of sharing is to increase efficiency
  - **Do not waste** CPU or memory resources (**fragmentation**)

# Outline

- Memory management goals
- Segmentation
- Paging
- TLB

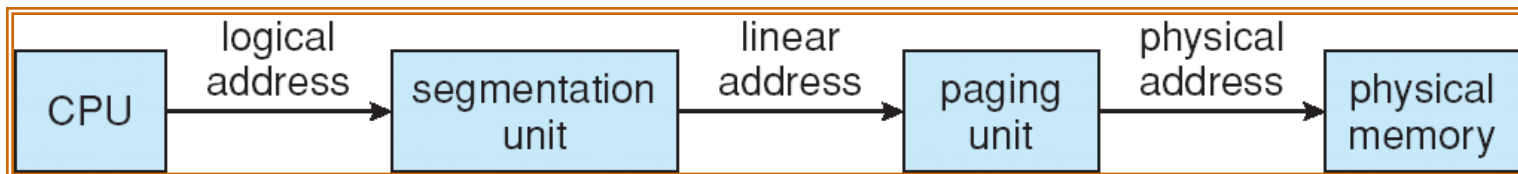
# Memory Management Unit (MMU)



- ❑ Map program-generated address (**virtual address**) to hardware address (**physical address**) dynamically at every reference
- ❑ Check range and permissions
- ❑ Programmed by OS

# x86 address translation

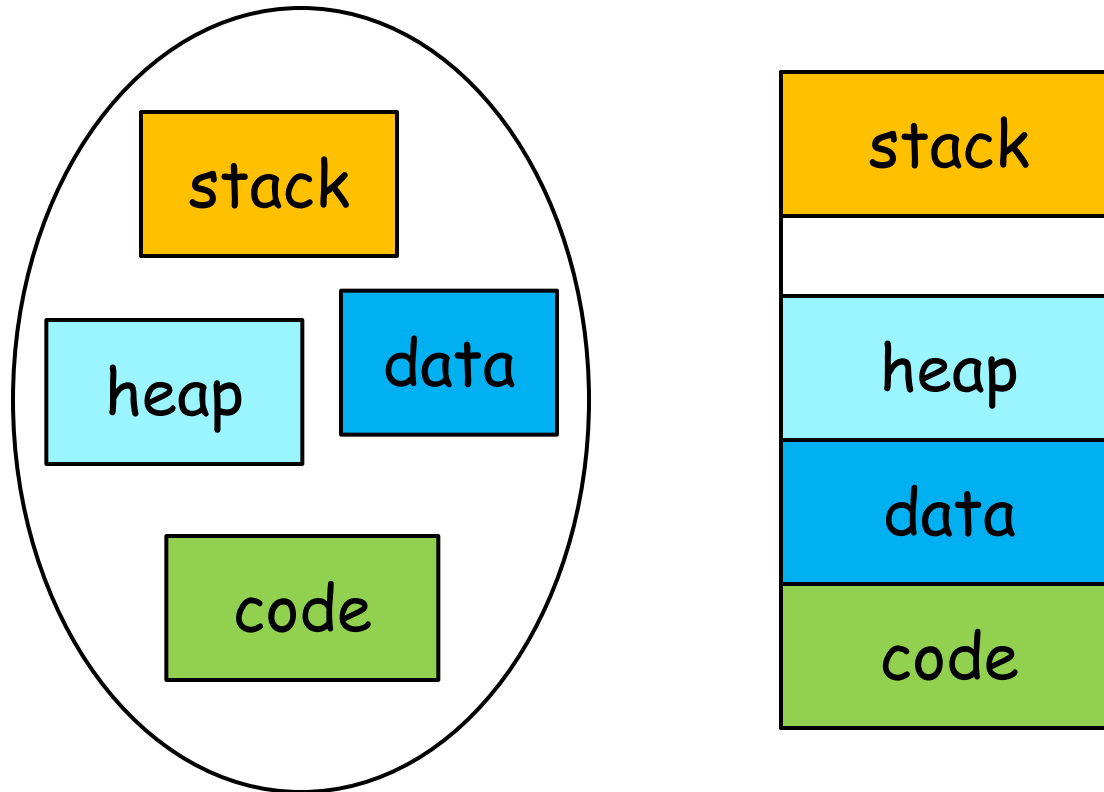
- CPU generates virtual address (seg, offset)
  - Given to segmentation unit
    - Which produces **linear addresses**
  - Linear address given to paging unit
    - Which generates physical address in main memory





# Segmentation

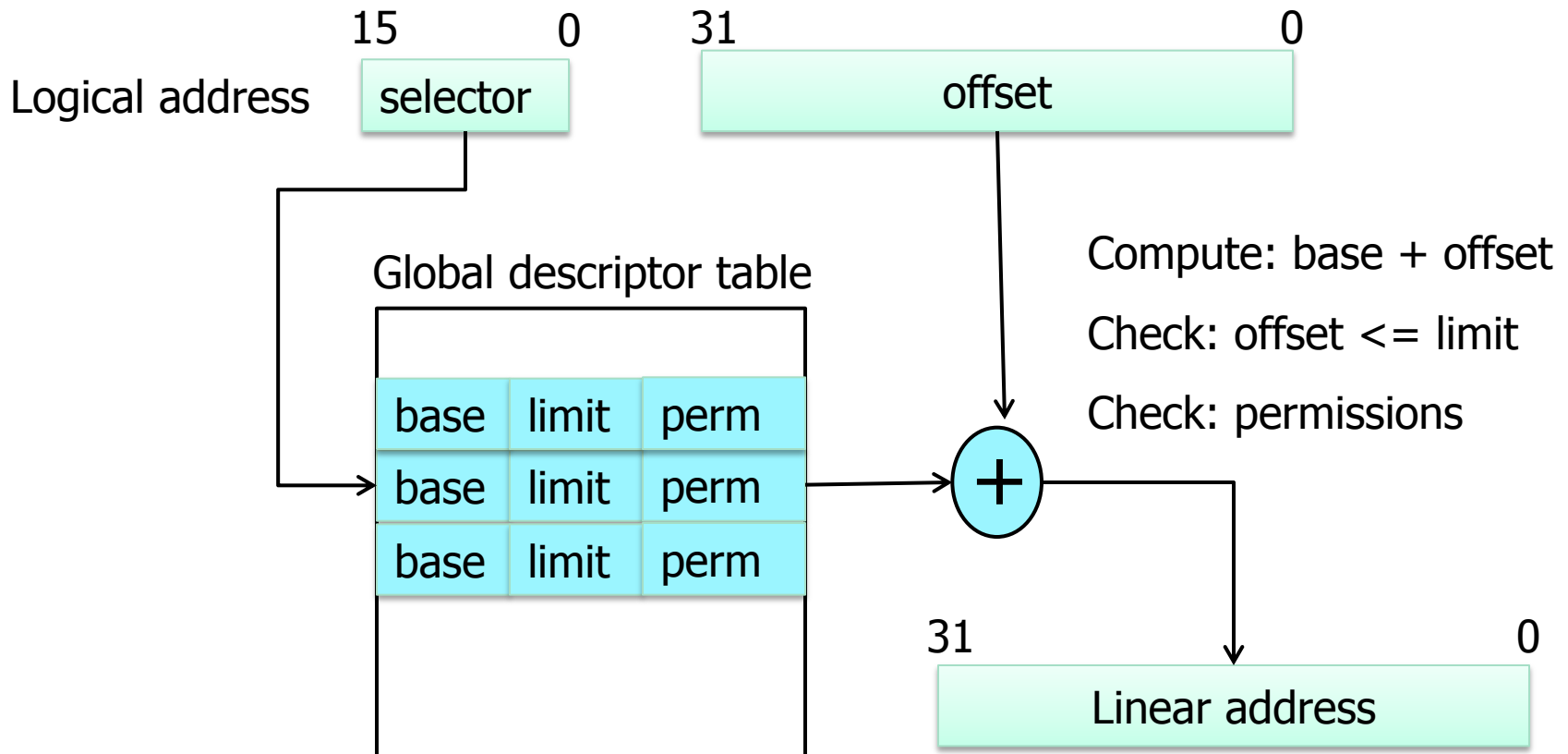
- Divide virtual address space into separate logical segments; each is part of physical mem



# Segmentation translation

- ❑ Virtual address:  $\langle \text{segment-number}, \text{offset} \rangle$
- ❑ Segment table maps segment number to segment information
  - **Base**: starting address of the segment in physical memory
  - **Limit**: length of the segment
  - Additional metadata includes **protection bits**
- ❑ Limit & protection checked on each access

# x86 segmentation hardware



# xv6 segments

- ❑ `vm.c, seginit()`
- ❑ Kernel code: readable + executable in kernel mode
- ❑ Kernel data: writable in kernel mode
- ❑ User code: readable + executable in user mode
- ❑ User data: writable in user mode
- ❑ These are all **null** mappings
  - Map to `[0, 0xFFFFFFFF)`
  - Linear address = Offset
- ❑ Kernel CPU: shortcuts to per-CPU data
  - Base: `&c->cpu`
  - Limit: 8 bytes

# Pros and cons of segmentation

## □ Advantages

- Segment sharing
- Easier to relocate segment than entire program
- Avoids allocating unused memory
- Flexible protection
- Efficient translation
  - Segment table small → fit in MMU

## □ Disadvantages

- Segments have variable lengths → how to fit?
- Segments can be large → fragmentation

# Outline

- ❑ Memory management goals
- ❑ Segmentation
- ❑ Paging
- ❑ TLB

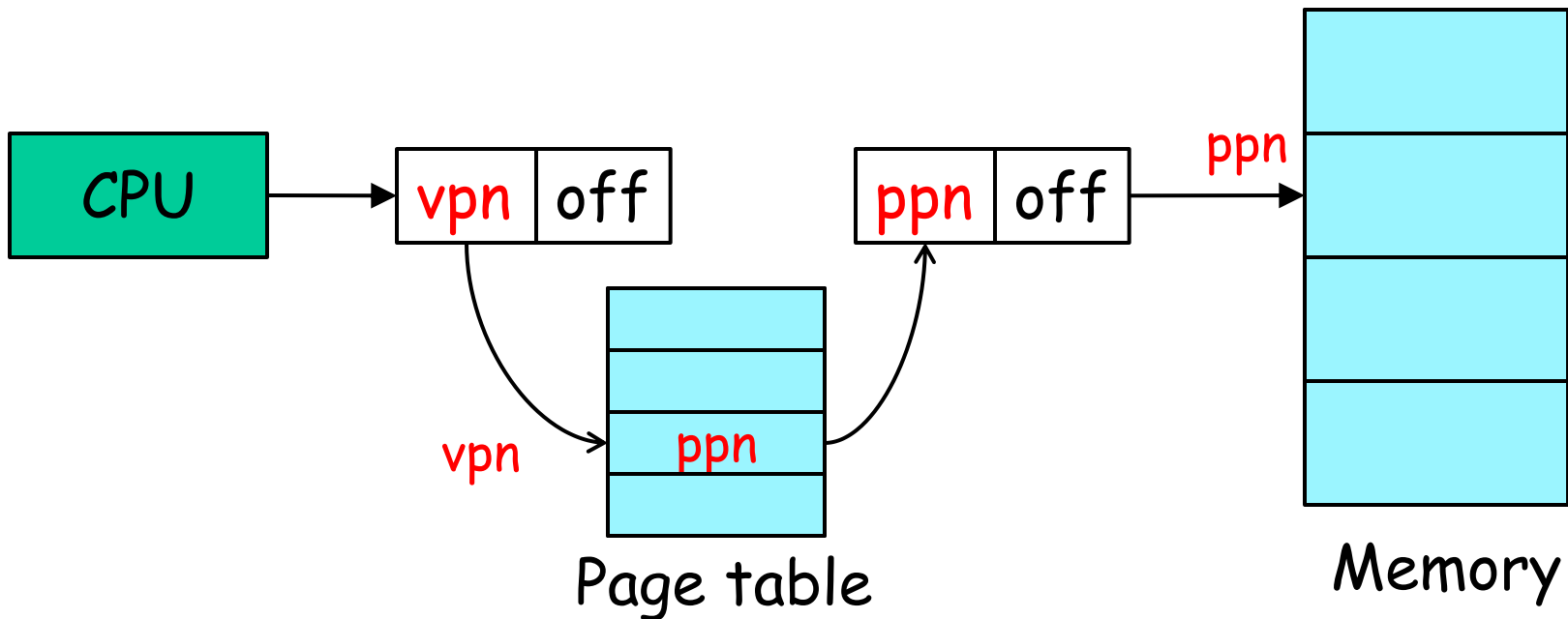
# Paging overview

- Goal
  - Eliminate fragmentation due to large segments
  - Don't allocate memory that will not be used
  - Enable fine-grained sharing
  
- Paging: divide memory into fixed-sized pages
  - For both virtual and physical memory
  
- Another terminology
  - A virtual page: page
  - A physical page: frame

# Page translation

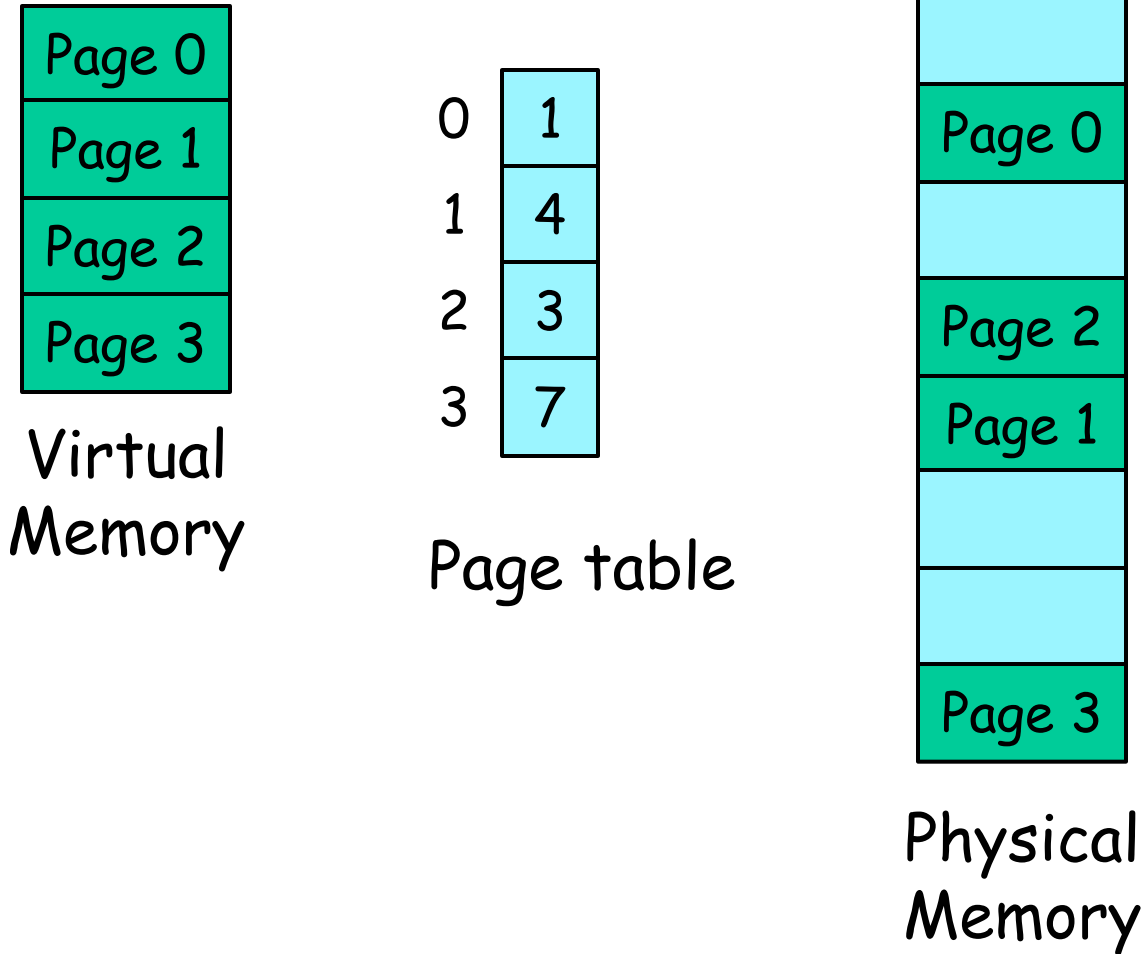
- Address bits = page number + page offset
- Translate virtual page number (vpn) to physical page number (ppn) using page table

$$pa = \text{page\_table}[va/pg\_sz] + va \% pg\_sz$$





# Page translation example



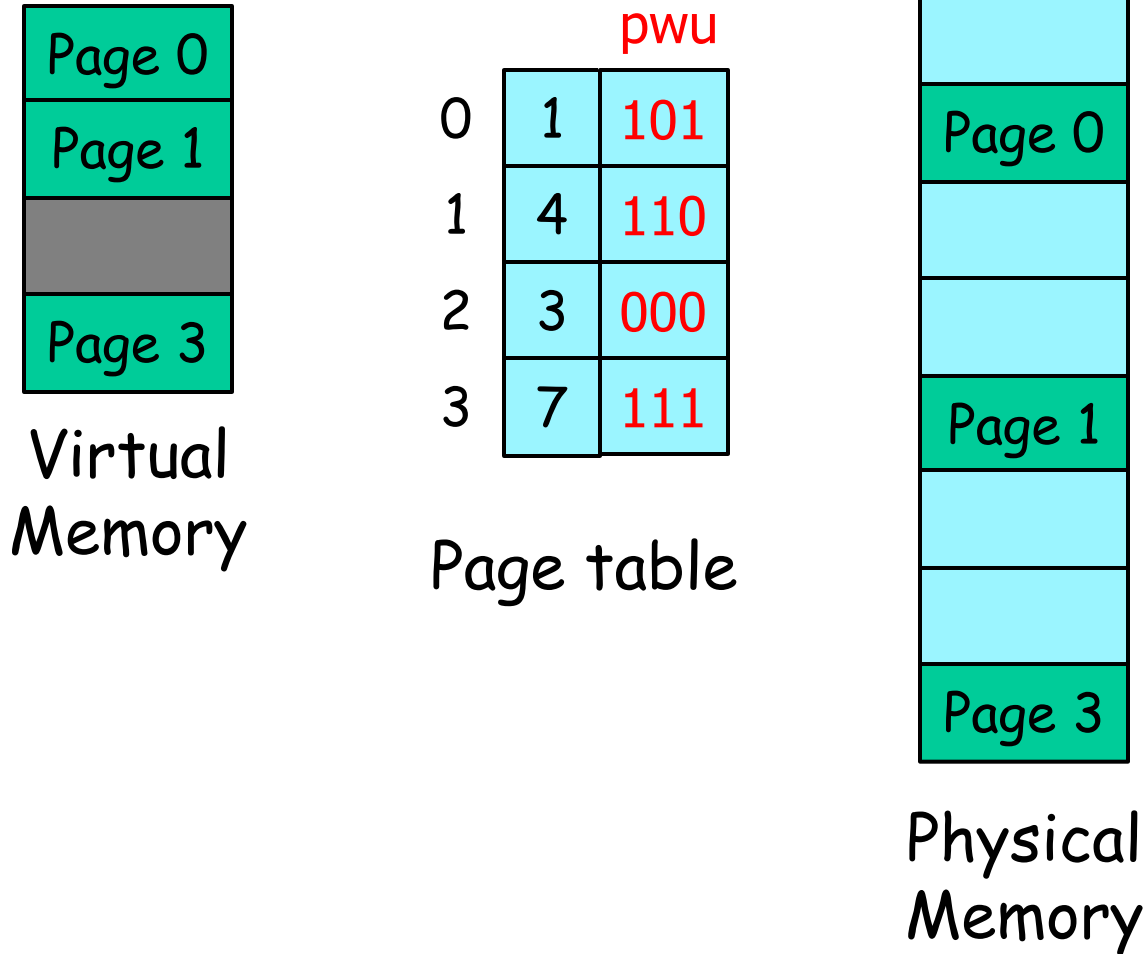
# Page translation exercise

- 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

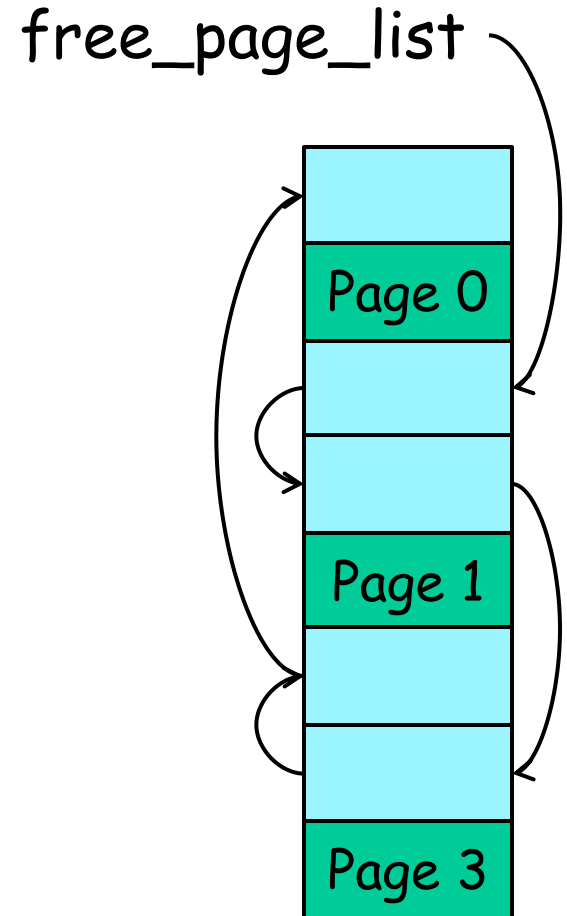
- ❑ Implemented by associating **protection bits** with each virtual page in page table
- ❑ Protection bits
  - **present bit**: map to a valid physical page?
  - **read/write/execute bits**: can read/write/execute?
  - **user bit**: can access in user mode?
  - **x86: PTE\_P, PTE\_W, PTE\_U**
- ❑ Checked 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
- ❑ xv6: [kalloc.c](#)



2, 3, 6, 5, 0

# Implementation of page table

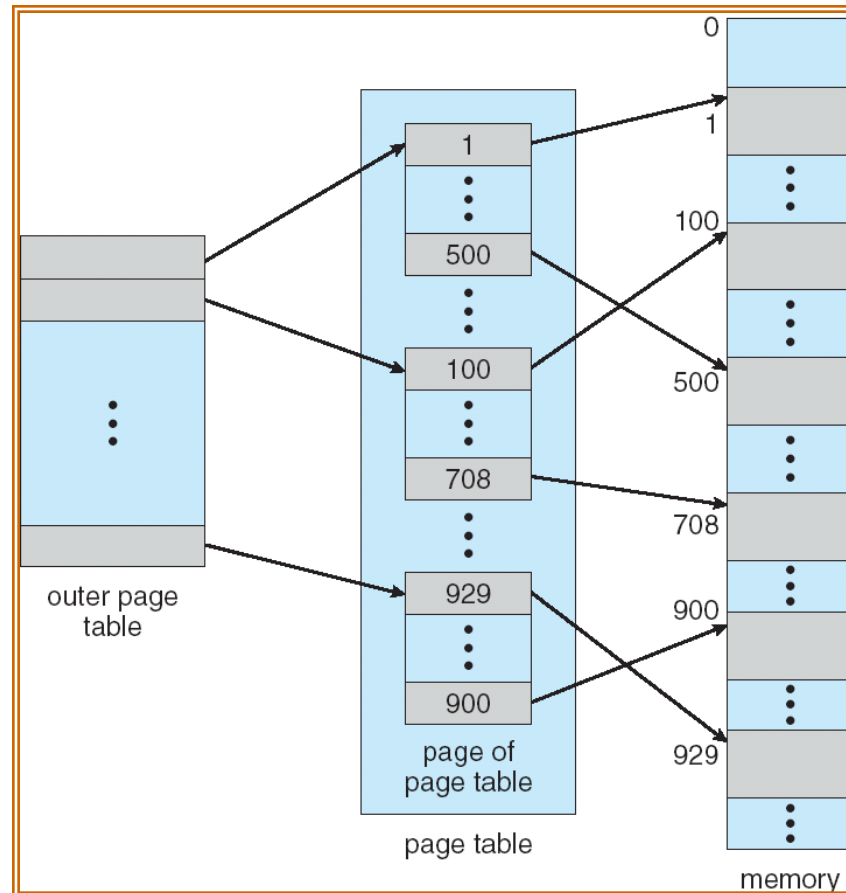
- Page table is stored in memory
  - Page table base register (PTBR) points to the base of page table
    - x86: cr3
  - OS stores base in process control block (PCB)
  - OS switches PTBR on each context switch
- Problem: each data/instruction access requires two memory accesses
  - Extra memory access for page table

# 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

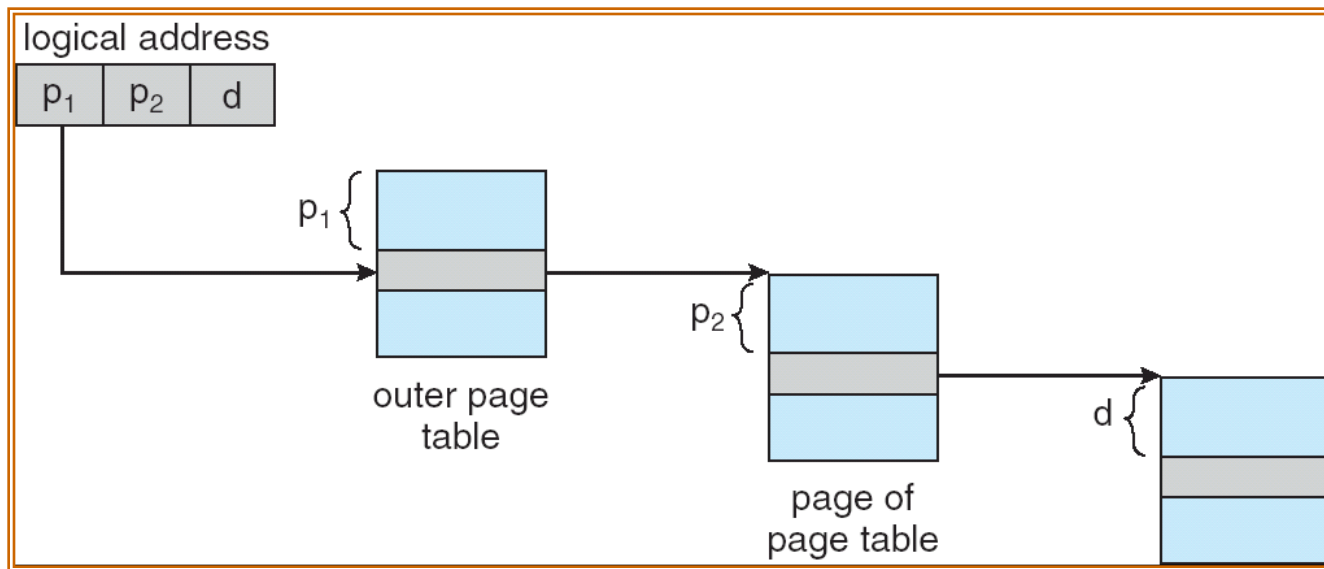
# Hierarchical page table

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



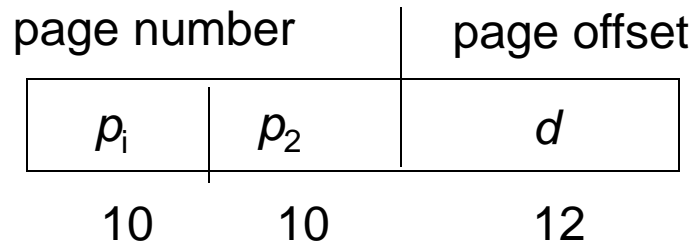


# Address translation with hierarchical page table

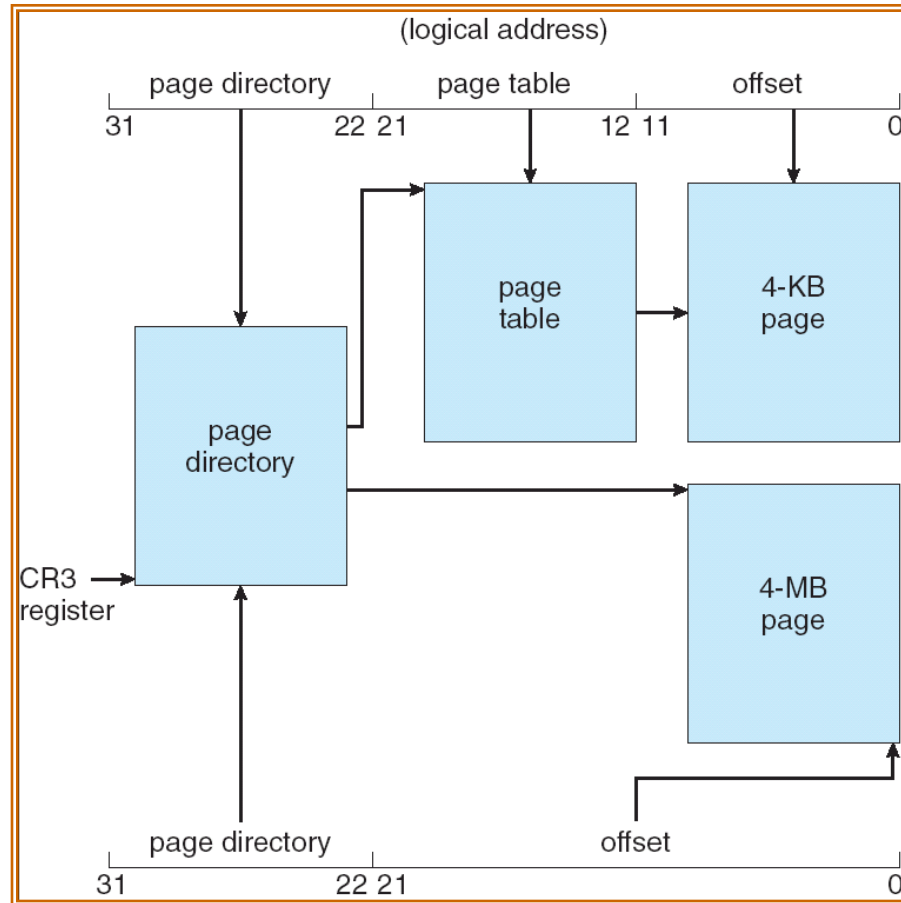


# x86 page translation with 4KB pages

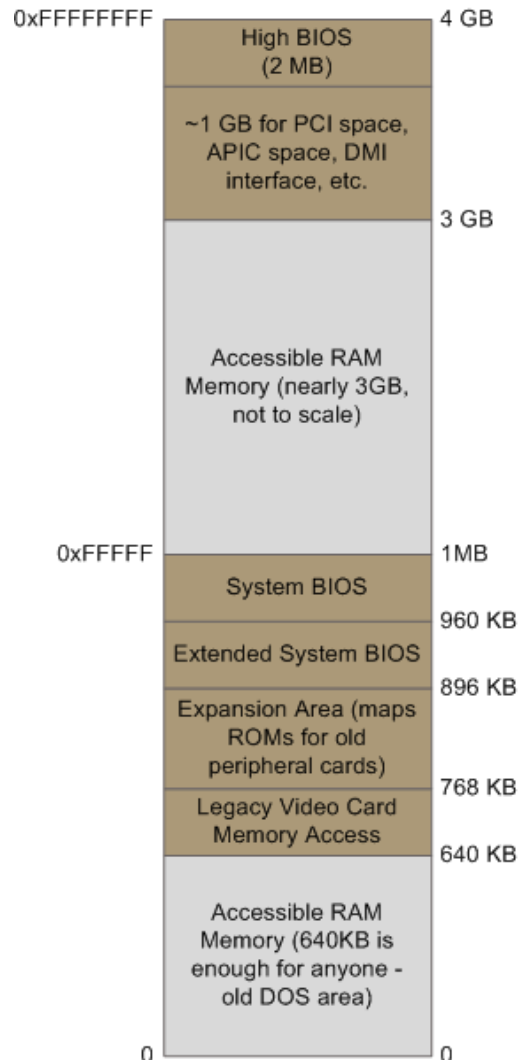
- 32-bit address space, 4 KB page
  - 4KB page → 12 bits for page offset
- How many bits for 2<sup>nd</sup>-level page table?
  - Desirable to fit a 2<sup>nd</sup>-level page table in one page
  - $4\text{KB}/4\text{B} = 1024 \rightarrow 10$  bits for 2<sup>nd</sup>-level page table
- Address bits for top-level page table:  $32 - 10 - 12 = 10$



# x86 paging architecture



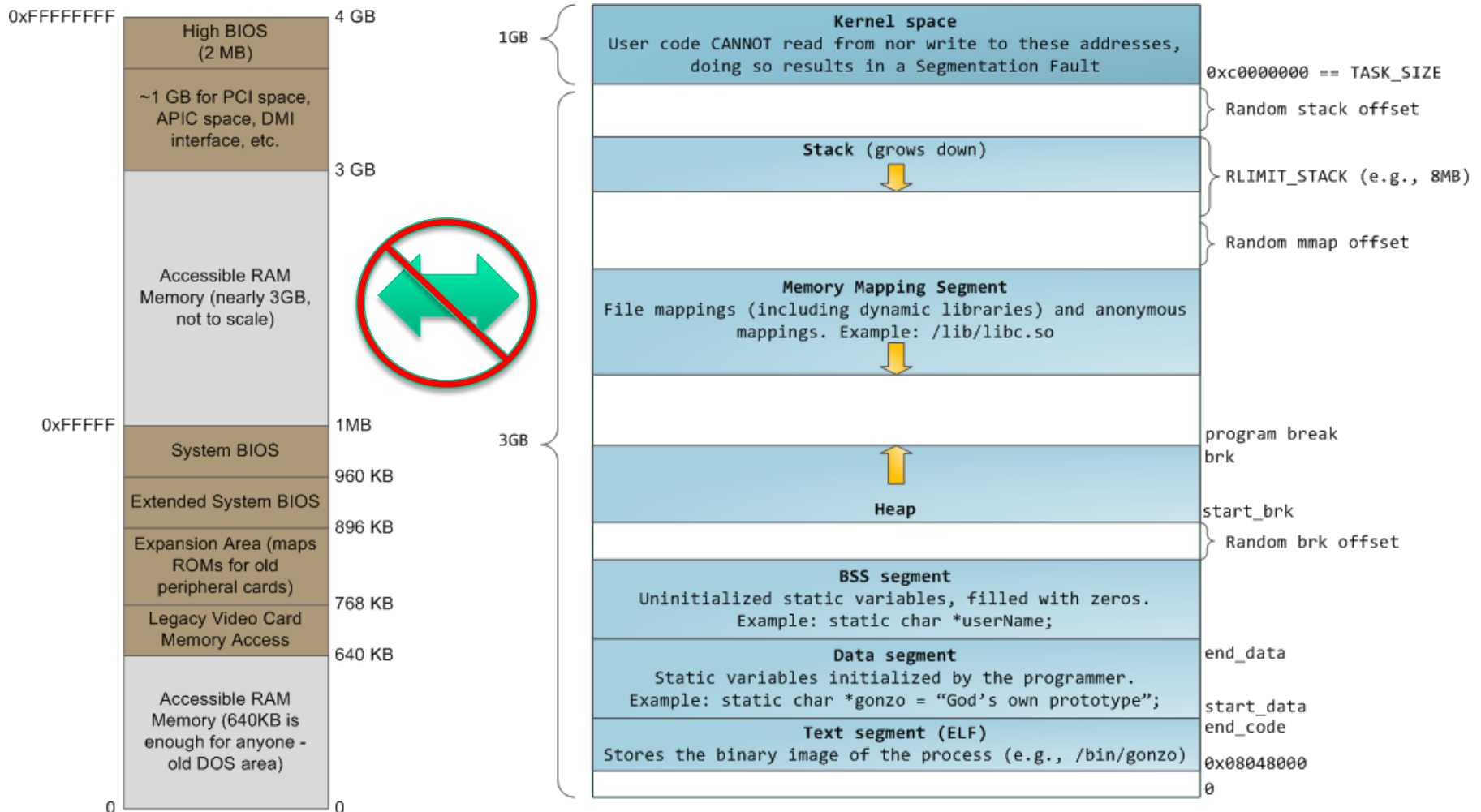
# x86 PC Physical Memory Layout



- ❑ Specific to each platform
- ❑ Different across architectures
- ❑ Different for machines with the same processor
- ❑ Firmware knows exact layout
- ❑ Passes to kernel at boot time (in Linux through `atag_mem` structures)

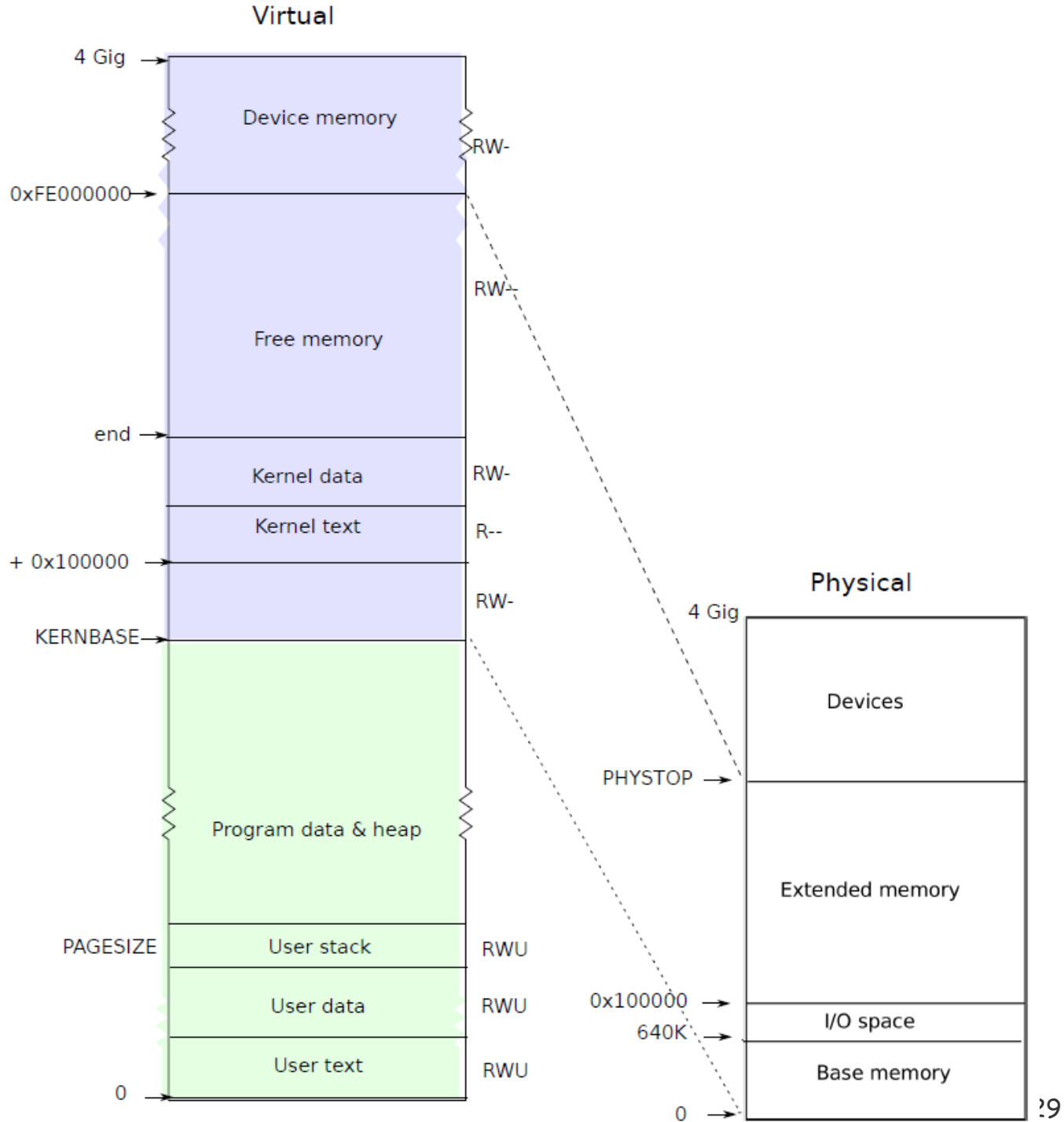
# Linux Address Space Layout

- Same address layout for all processes



Read: <http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory>

# xv6 address space (memlayout.h)



# xv6 address space implementation

- ❑ Split into kernel space and user space
- ❑ User: 0--KERNBASE
  - Map to physical pages
- ❑ Kernel: KERNBASE--(KERNBASE+PHYSTOP)
  - Virtual address = physical address + KERNBASE
- ❑ Kernel: 0xFE000000--4GB
  - Direct (virtual = physical)
- ❑ Kernel: `vm.c`, `setupkvm()`
- ❑ User: `vm.c`, `initvm()` and `exec.c`, `exec()`

# Outline

- ❑ Memory management goals
- ❑ Segmentation
- ❑ Paging
- ❑ TLB

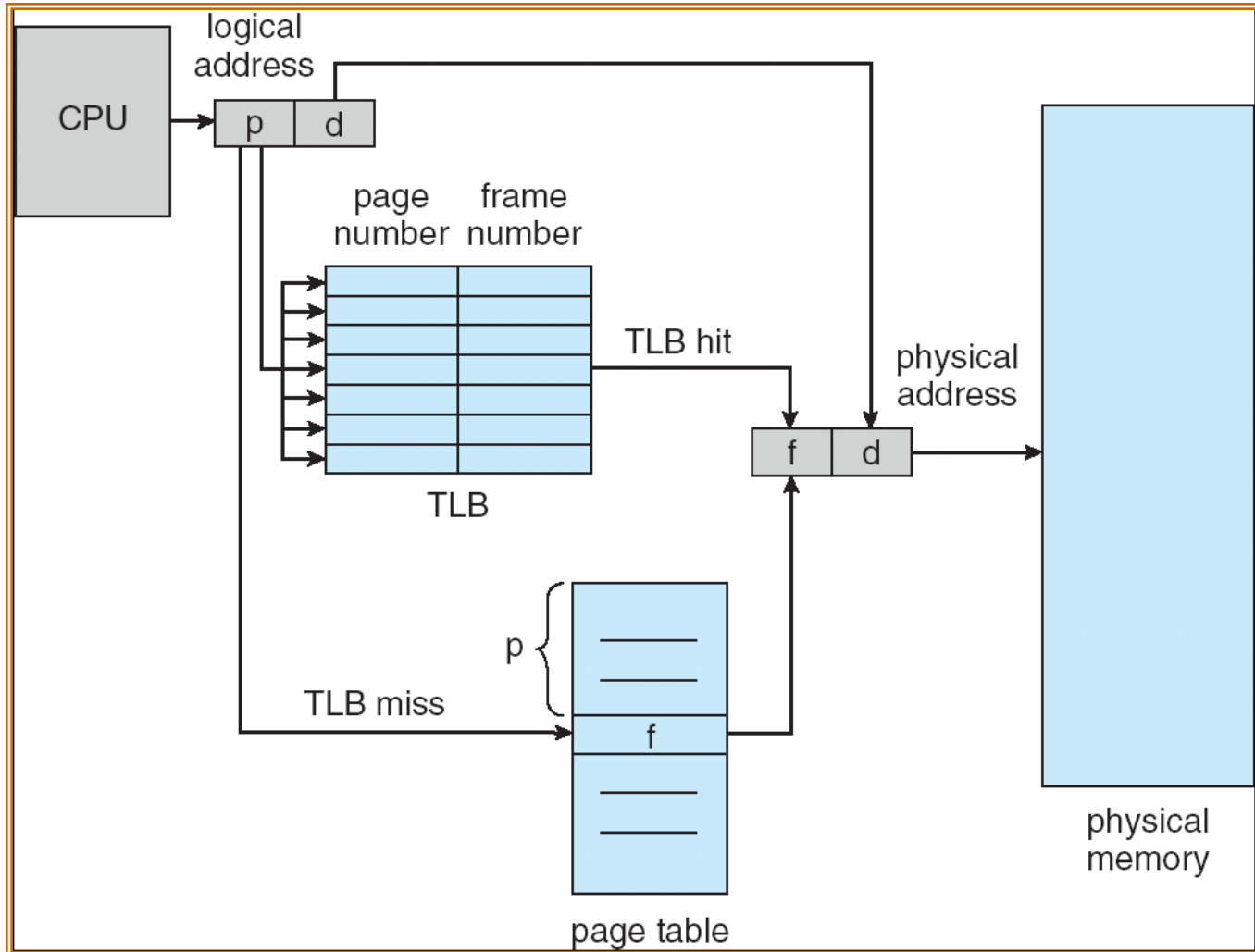


# Avoiding extra memory accesses

- Observation: **locality**
  - **Temporal**: access locations accessed **just now**
  - **Spatial**: access locations **adjacent** to locations accessed just now
  - Process often needs only **a small number** of vpn → ppn mappings at any moment!
- Fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
  - **Fast** parallel search (CPU speed)
  - **Small**

VPN	PPN

# Paging hardware with TLB



# Effective access time with TLB

- Assume memory cycle time is **1 unit time**
- **TLB Lookup time =  $\epsilon$**
- **TLB Hit ratio =  $\alpha$** 
  - Percentage of times that a  $\text{vpn} \rightarrow \text{ppn}$  mapping is found in TLB

## □ **Effective Access Time (EAT)**

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

# TLB Miss

- Depending on the architecture, TLB misses are handled in either hardware or software
- Hardware (CISC: x86)
  - Pros: hardware doesn't have to trust OS !
  - Cons: complex hardware, inflexible
- Software (RISC: MIPS, SPARC)
  - Pros: simple hardware, flexible
  - Cons: code may have bug!
  - Question: what can't a TLB miss handler do?

# TLB and context switches

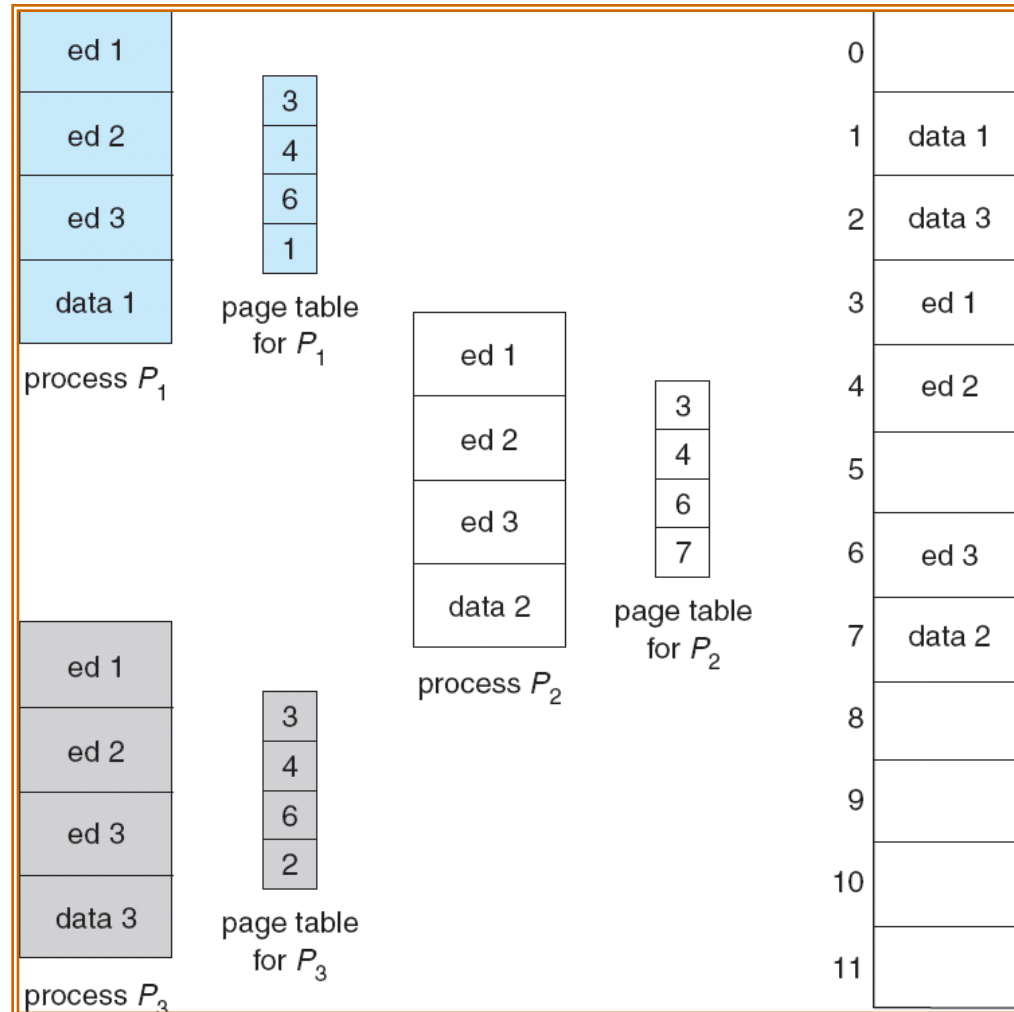
- What happens to TLB on context switches?
- Option 1: flush entire TLB
  - x86
    - “load cr3” (load page table base) flushes TLB
  - x86 “INVLPG addr” invalidates one TLB entry
- Option 2: attach process ID to TLB entries
  - ASID: Address Space Identifier
  - MIPS, SPARC

# Backup Slides

# 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: **copy-on-write fork.** Parent and child processes share pages right after fork; copy only when either writes to a page

# Page sharing example

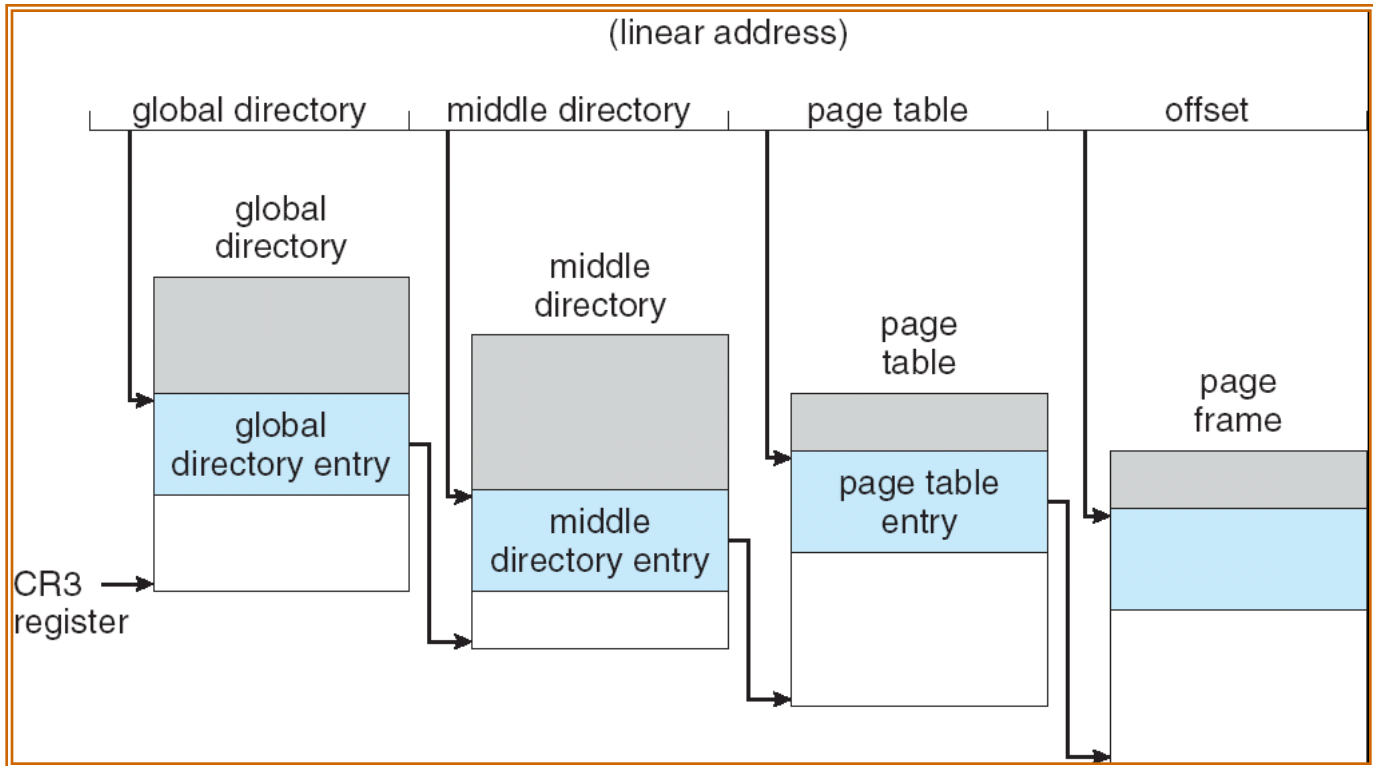




# 80x86 segment selector

- Logical address: **segment selector** + offset
- **Segment selector** stored in **segment registers** (16-bit)
  - **cs**: code segment selector
  - **ss**: stack segment selector
  - **ds**: data segment selector
  - **es, fs, gs**
- Segment register can be implicitly or explicitly specified
  - Implicit by type of memory reference (**jmp**)
    - `mov $8049780, %eax // implicitly use ds`
  - Through special registers (**cs, ss, es, ds, fs, gs** on x86)
  - `mov %ss:$8049780, %eax // explicitly use ss`

# Three-level Paging in Linux



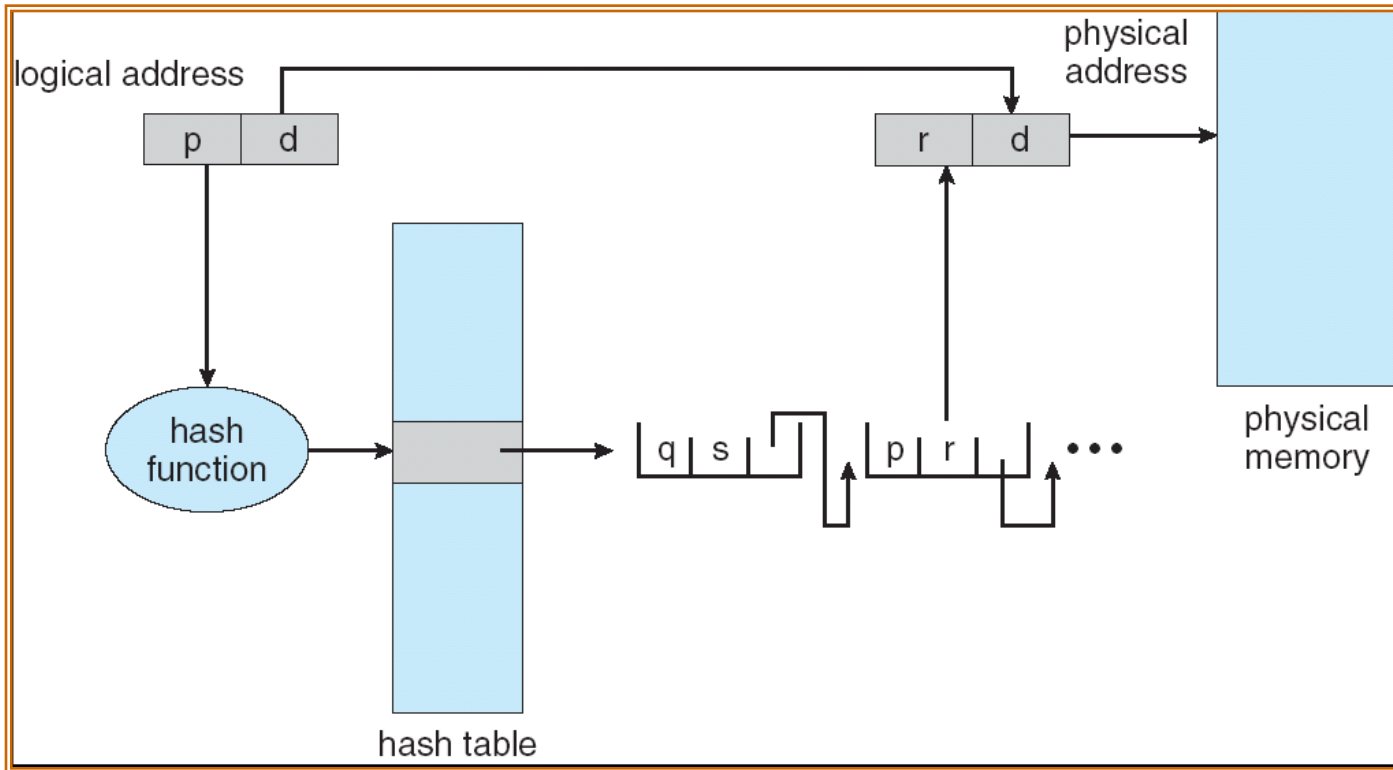
# Page table structures

- ❑ Hierarchical paging
- ❑ Hashed page tables
- ❑ Inverted page tables

# 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

