Memory Management I
Segmentation and Paging

COMS W4118

Prof. Kaustubh R. Joshi

krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

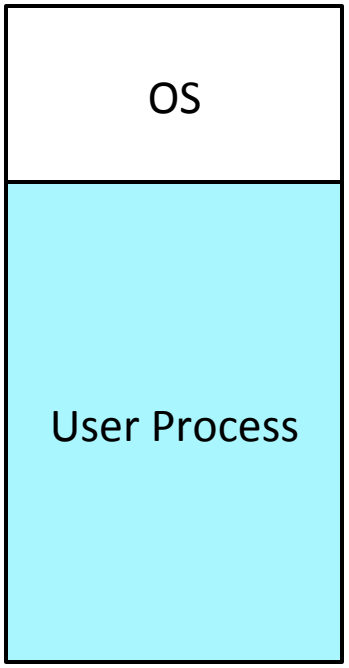
References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s

Outline

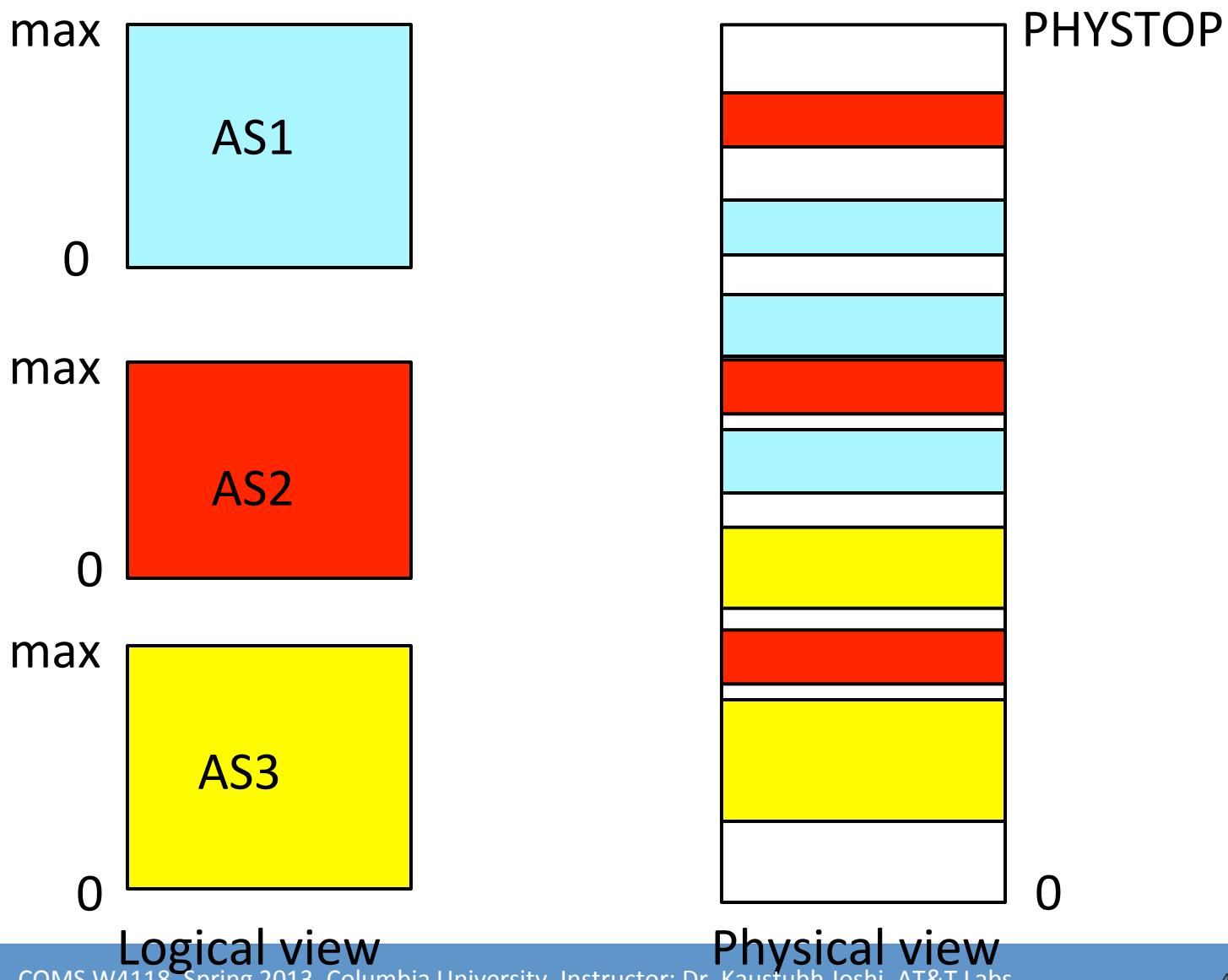
- Memory management goals
- Segmentation
- Paging
- TLB
- Page sharing

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!



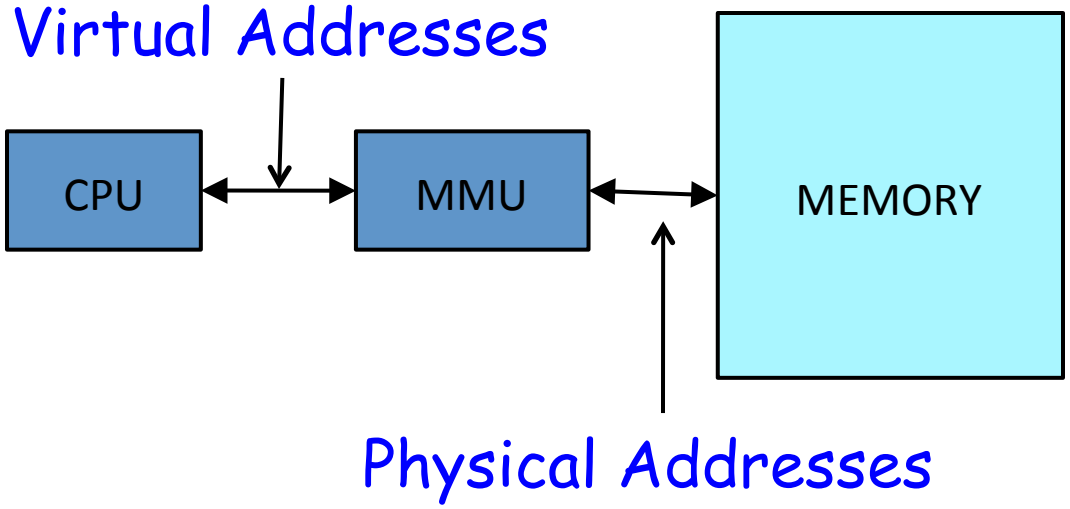
Multiple address spaces co-exist



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)

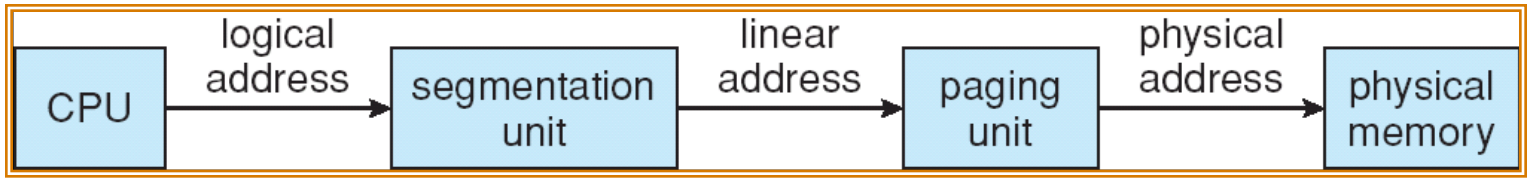
Memory Management Unit (MMU)



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

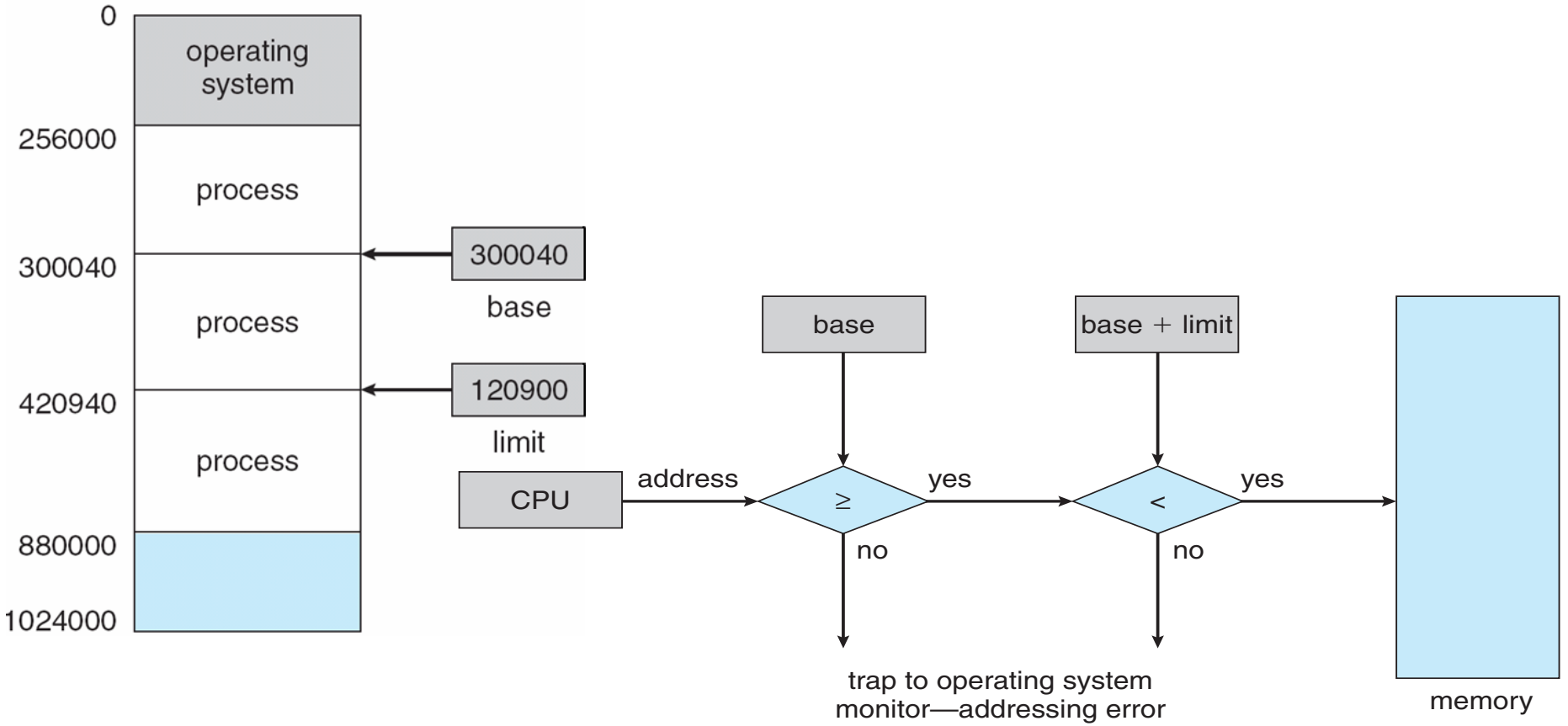
x86 address translation

- 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



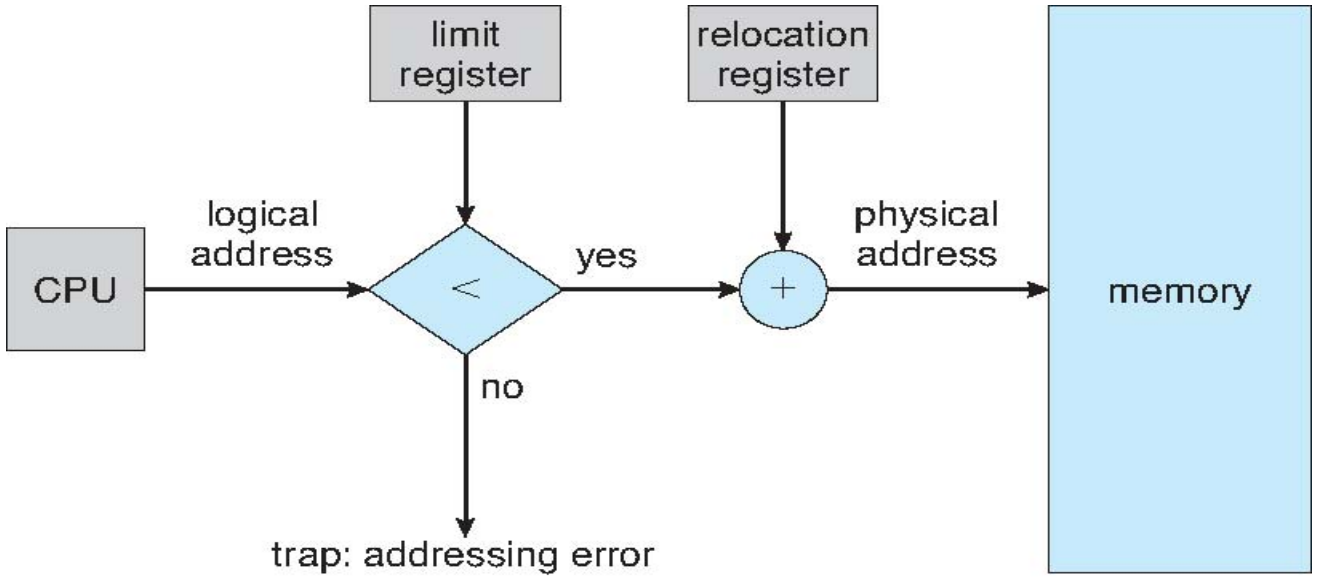
A Simple MMU: Base/Limit Registers

- **Base** and **limit registers** define logical address space
- CPU checks every memory access generated in user mode to be sure it is between base and limit for that user



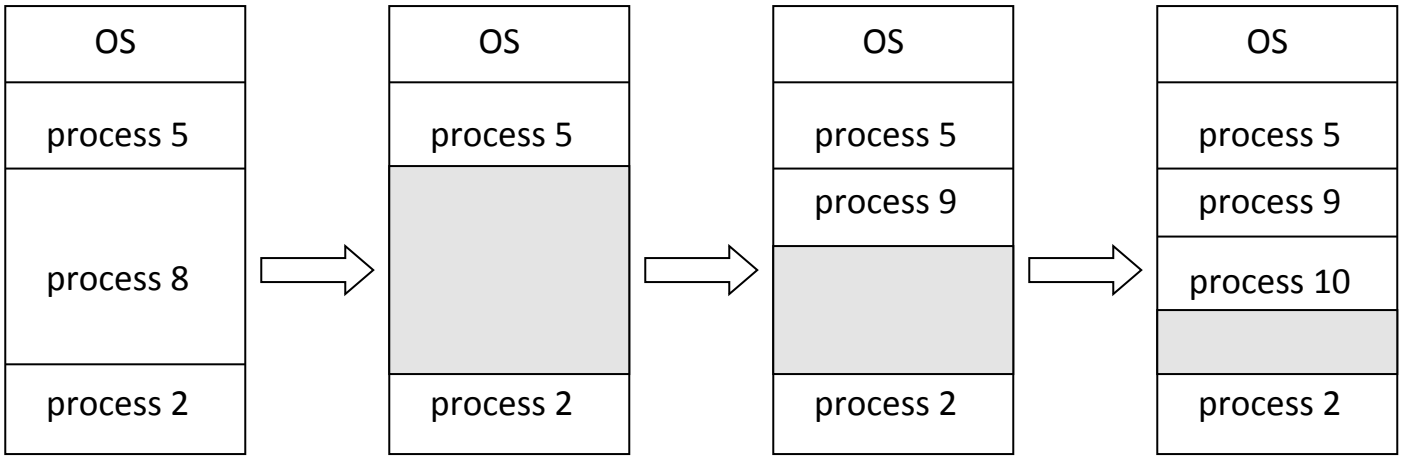
A better MMU: Relocatable Code

- Problem with base limit register solution?
  - Need to know address at which program will be before hand: linker/ loader must rewrite instructions
  - Can't change location once loaded, prone to fragmentation
- Solution: add a relocation register
  - Programmer uses addresses that are offsets from base
  - Hardware adds actual value of base at runtime to get final address



Problems with contiguous allocation

- Partition per program: how big should each partition be?
  - Entire size of address space? Impractical
  - How much program actually uses? May not know in advance
- Have to be conservative
  - Too small: must reallocate and move program (expensive)
  - Too big: wasted memory
- Fragmentation over time
  - **Hole** – block of available memory; scattered throughout memory
  - Need hole large enough to accommodate new processes

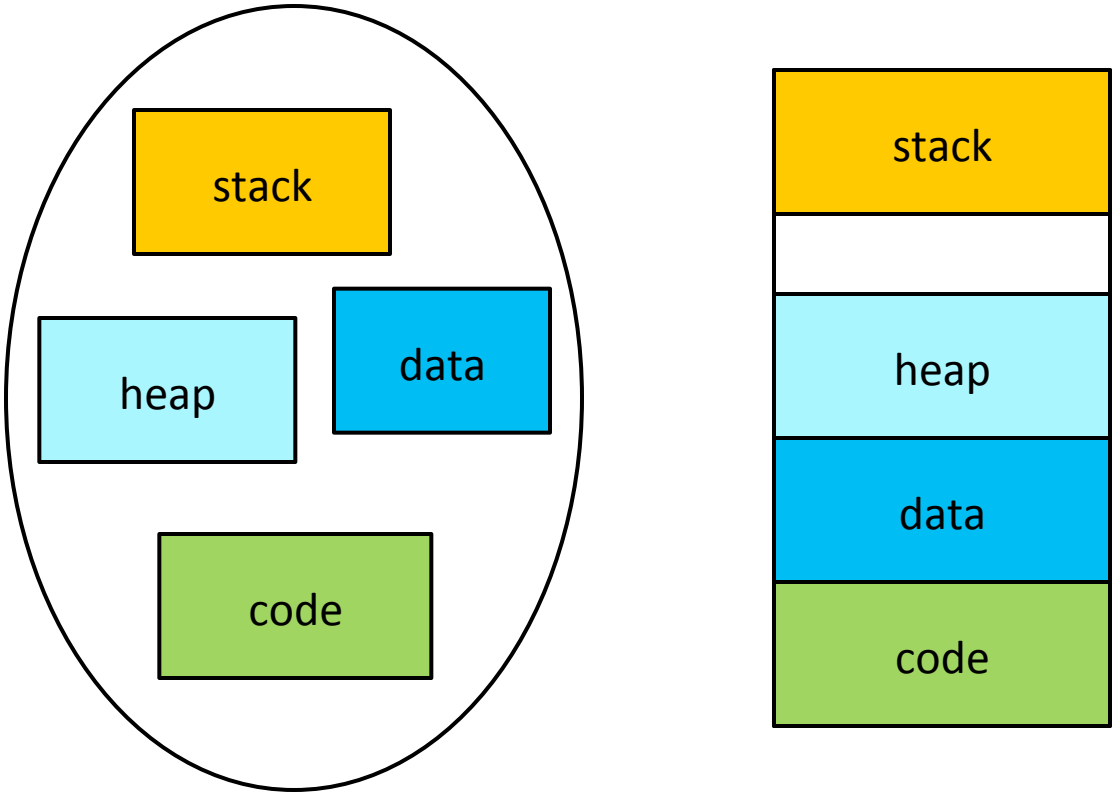


Outline

- Memory management goals
- Segmentation
- Paging
- TLB
- Page sharing

Segmentation

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



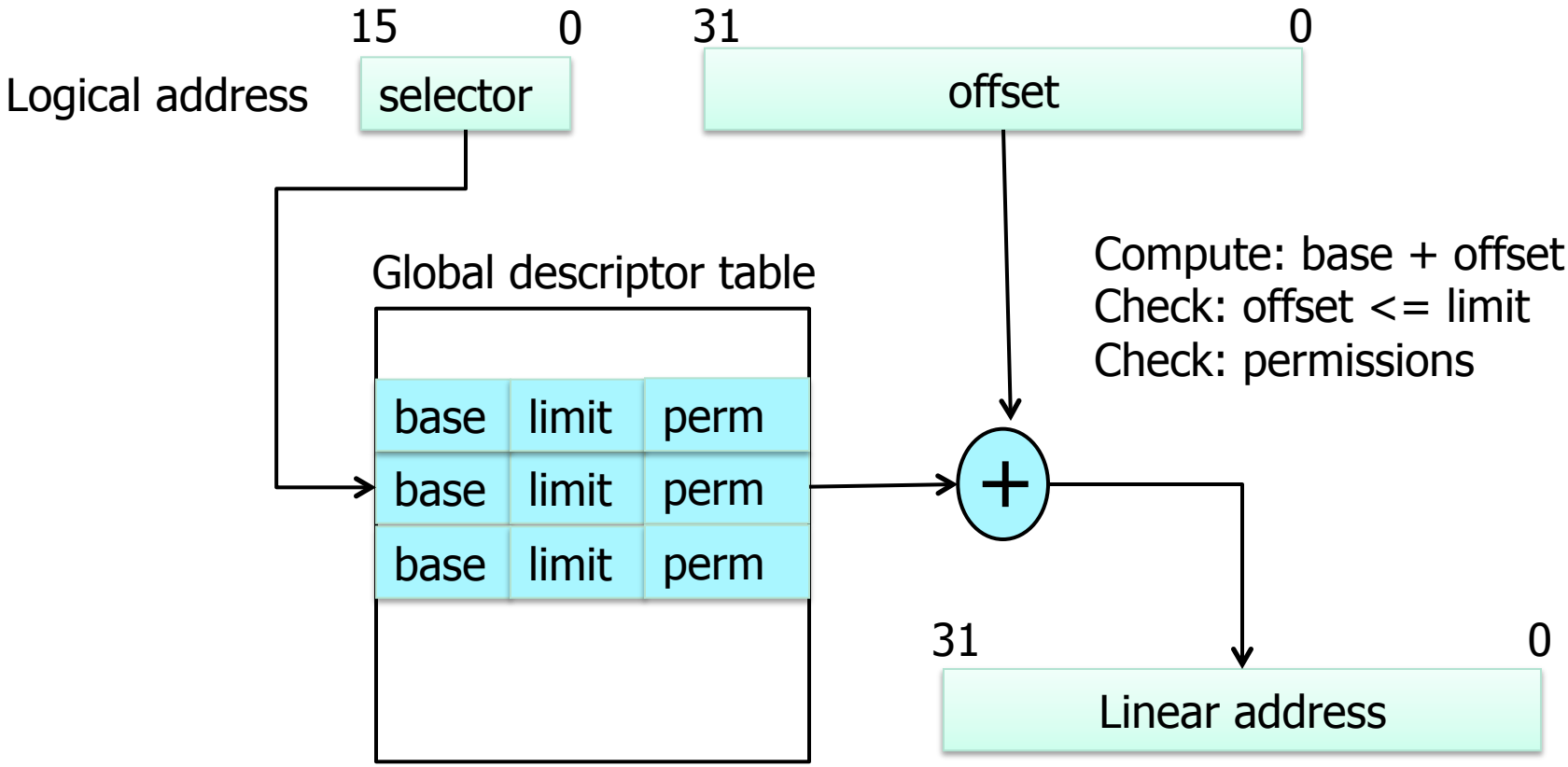
Segmentation translation

- Virtual address: <segment-number, offset>
- Segment table maps segment number to segment information
  - Base: starting address of the segment in physical memory
  - Limit: length of the segment
  - Addition metadata includes protection bits
- Limit & protection checked on each access

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
- Support for segmentation removed in x86-64

x86 segmentation hardware



Linux Segments

- Not much to see
  - Rely mainly on paging (next topic)
  - Basic common segments that span entire memory
- Different permissions dependent on use
  - Kernel code: read + executein 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

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

17

Outline

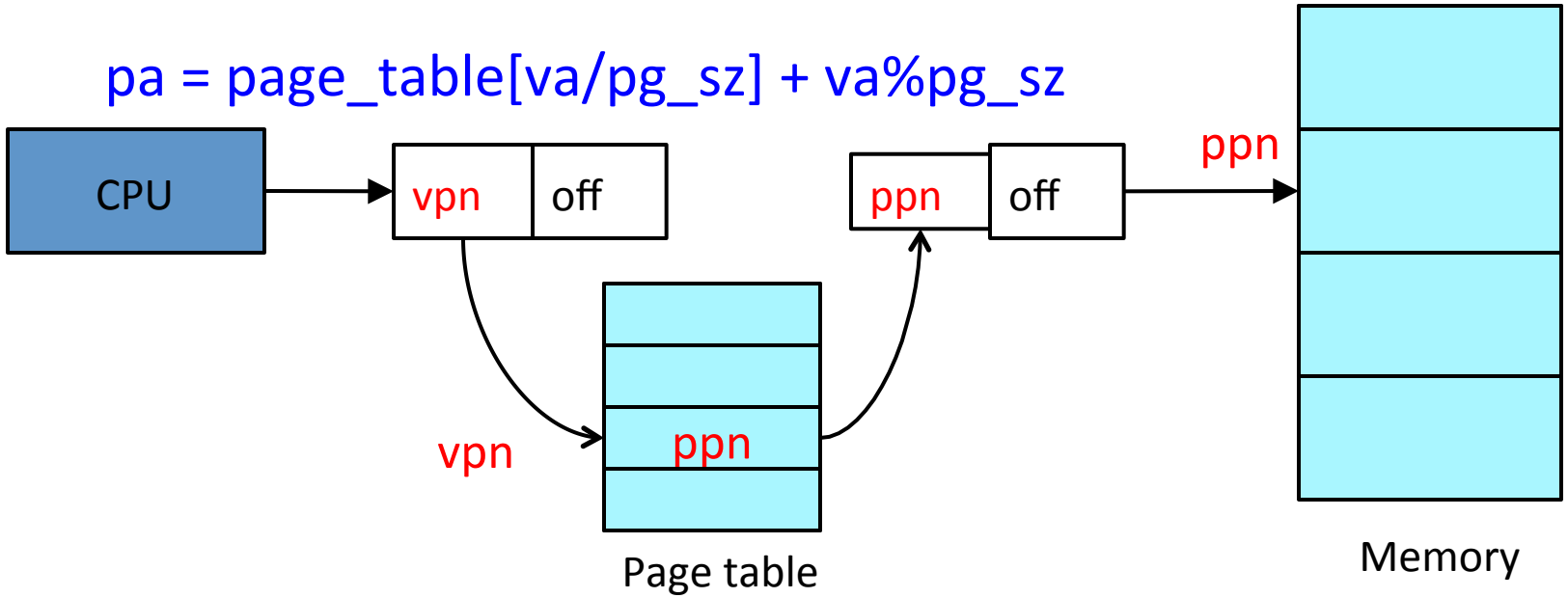
- Memory management goals
- Segmentation
- Paging
- TLB
- Page sharing

Paging overview

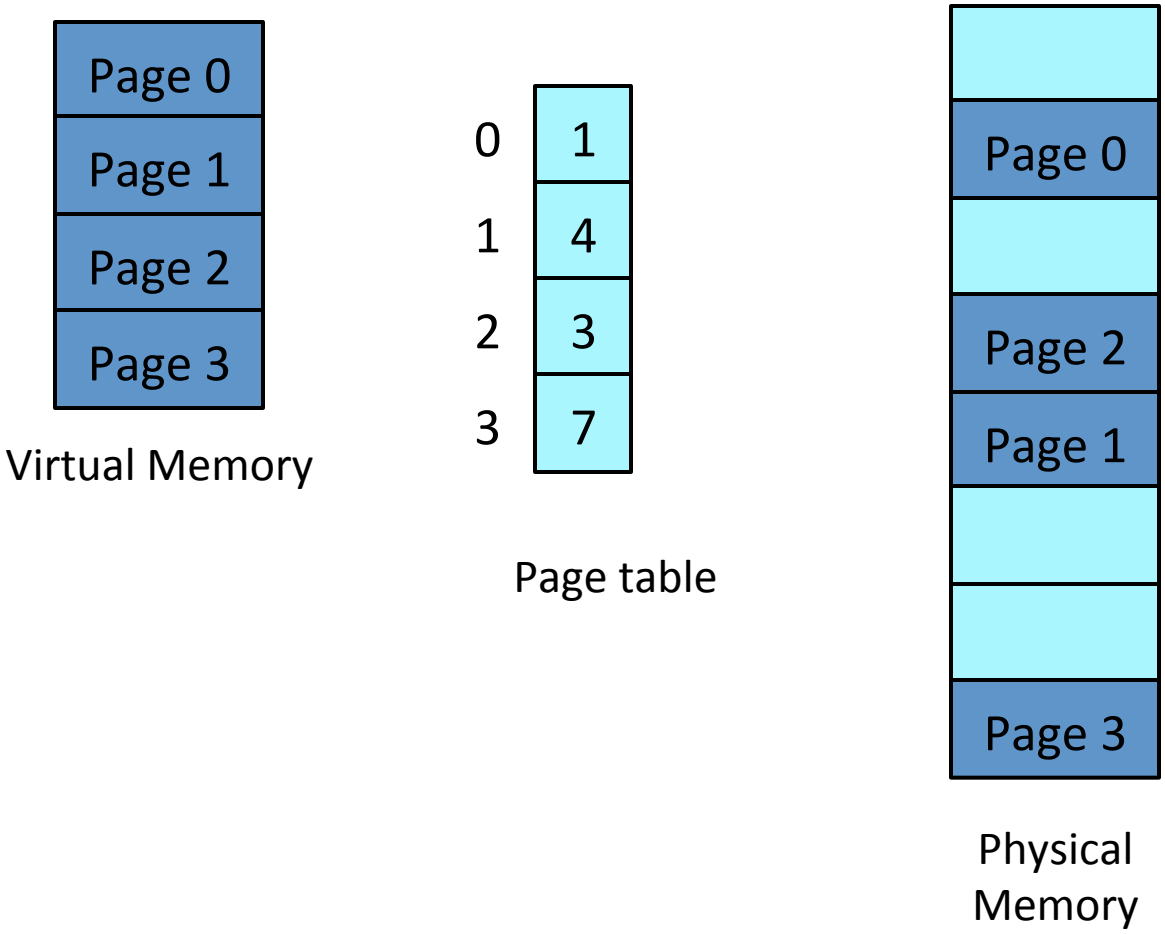
- Goal
  - Eliminate fragmentation due to large segments
  - Don't allocate memory that will not be used
  - Enable fine-grained sharing
- Paging: divide memory into fixed-sized pages
  - For both virtual and physical memory
- Another terminology
  - A virtual page: page
  - A physical page: frame

Page translation

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



Page translation example



Page translation exercise

- 8-bit virtual address, 10-bit physical address, each page is 64 bytes

1. How many virtual pages?
   - 2^8 / 64 = 4 virtual pages
2. How many physical pages?
   - 2^10/64 = 16 physical pages
3. How many entries in page table?
   - Page table contains 4 entries
4. Given page table = [2, 5, 1, 8], what's the physical address for virtual address 241?
   - 241 = 11110001b
   - 241/64 = 3 = 11b
   - 241%64 = 49 = 110001b
   - page_table[3] = 8 = 1000b
   - Physical address = 8 * 64 + 49 = 561 = 1000110001b

Page translation exercise

m-bit virtual address, n-bit physical address, k-bit page size

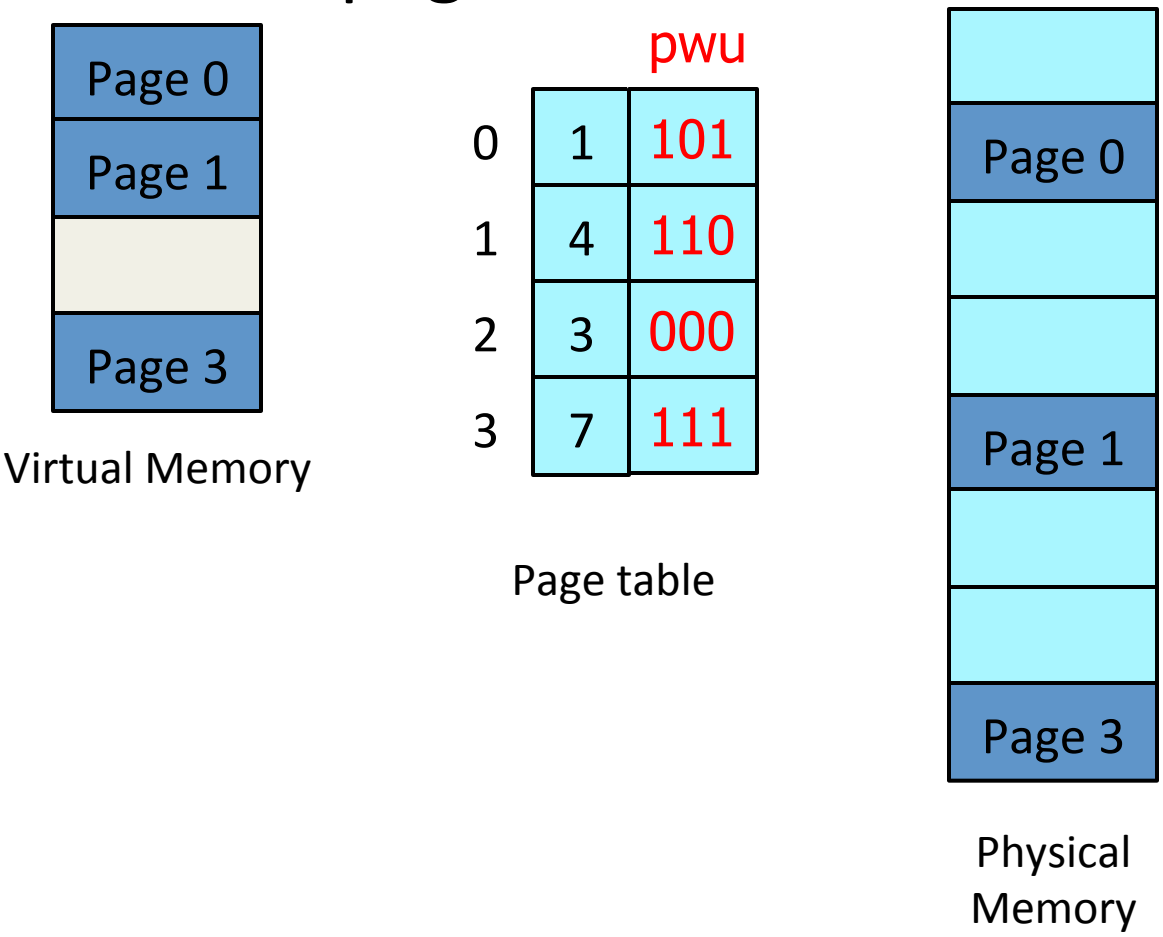
- # of virtual pages: $2^{(m-k)}$
- # of physical pages: $2^{(n-k)}$
- # of entries in page table: $2^{(m-k)}$
- vpn = va / 2^k
- offset = va % 2^k
- ppn = page_table[vpn]
- pa = ppn * 2^k + offset

Page protection

- Implemented by associating protection bits with each virtual page in page table
- Why do we need protection bits?
- 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

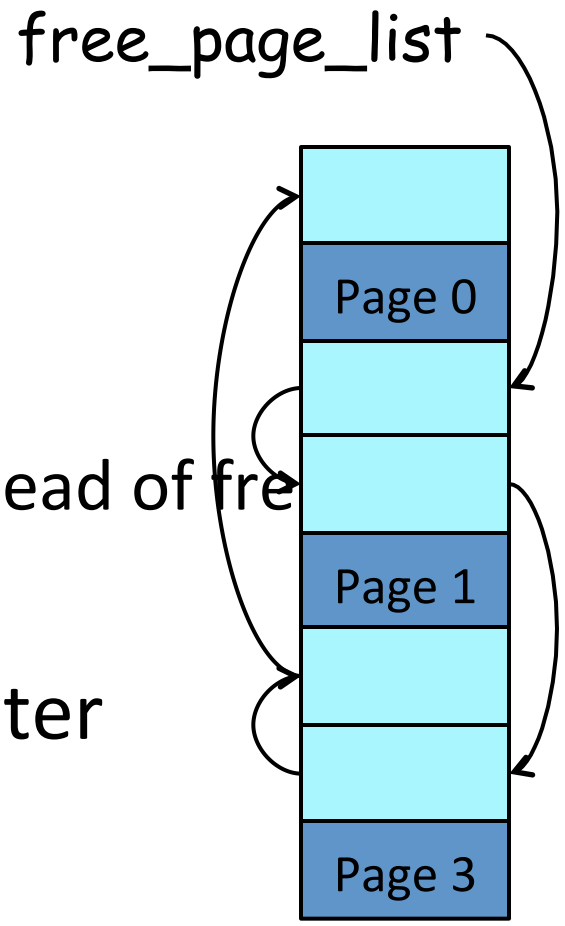
Page protection example

- What kind of pages?



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 fre
- We'll see allocation policies later



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

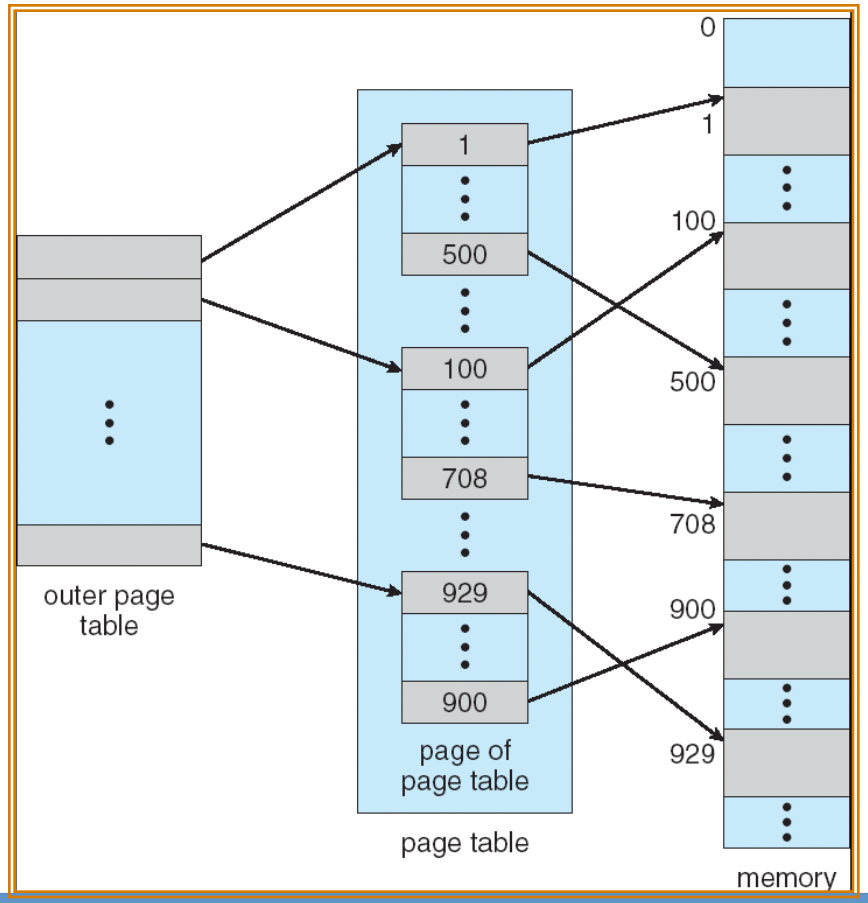
- Given:
  - A 32 bit address space (4 GB)
  - 4 KB pages
  - A page table entry of 4 bytes
- Implication: page table is 4 MB per process!
- Observation: address space are often sparse
  - Few programs use all of 2^{32} bytes
- Change page table structures to save memory
  - Trade translation time for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

Page table structures

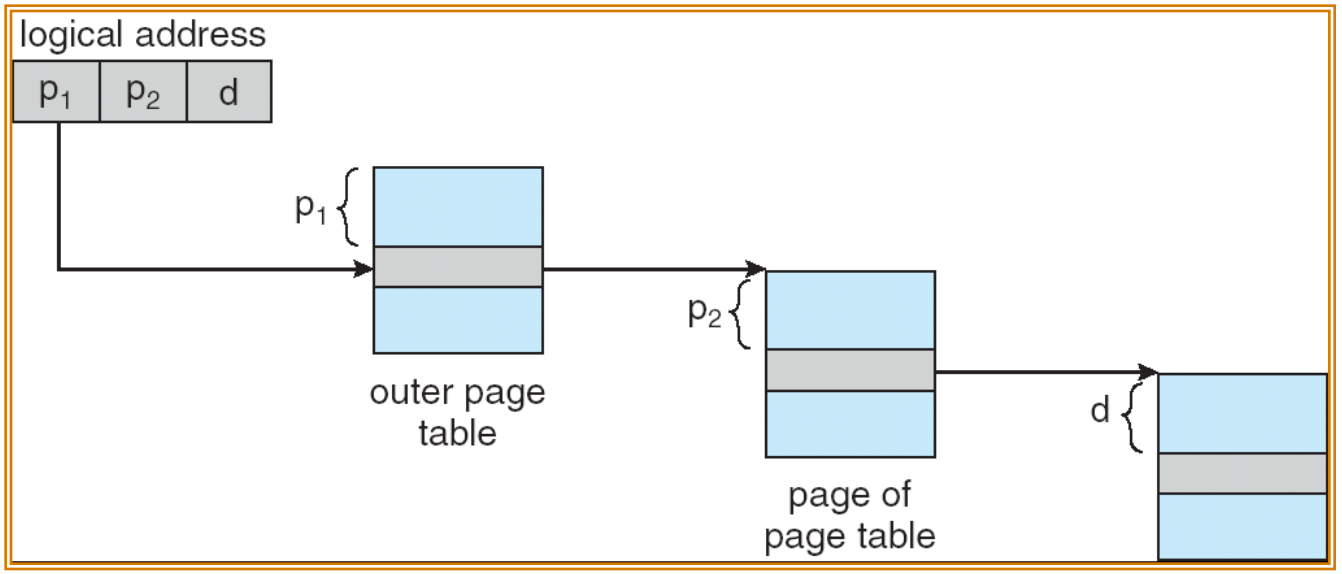
- Hierarchical paging
- Hashed page tables
- Inverted page tables

Hierarchical page table

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

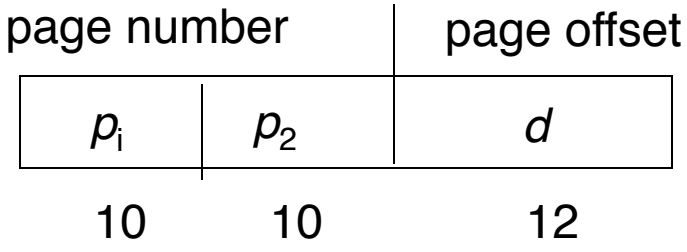


Hierarchical page tables

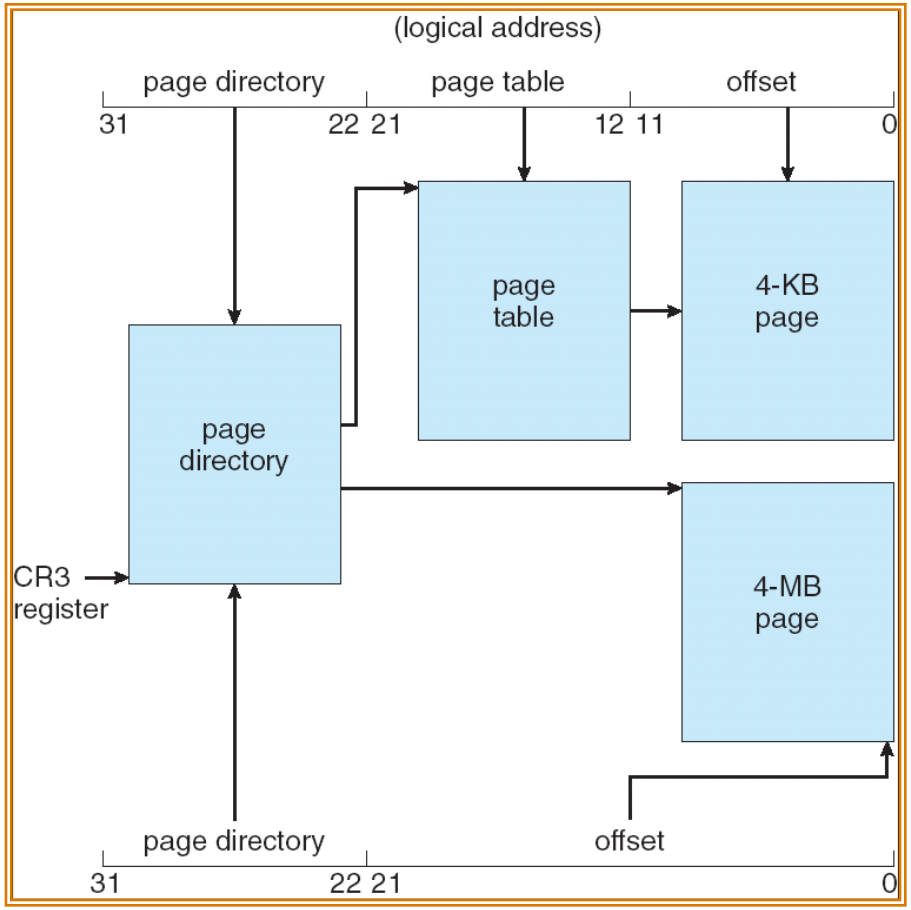


x86 page translation with 4KB pages

- 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 – 10 – 12 = 10

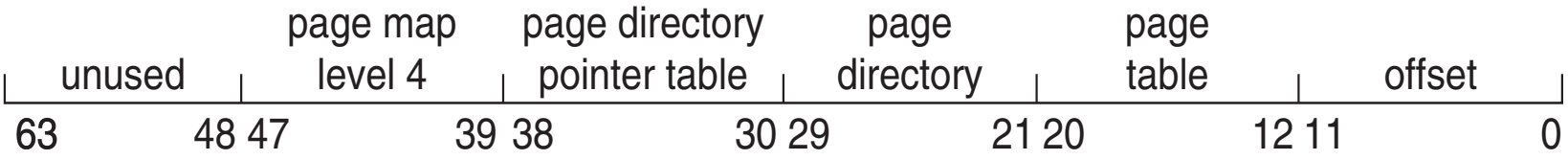


x86 paging architecture



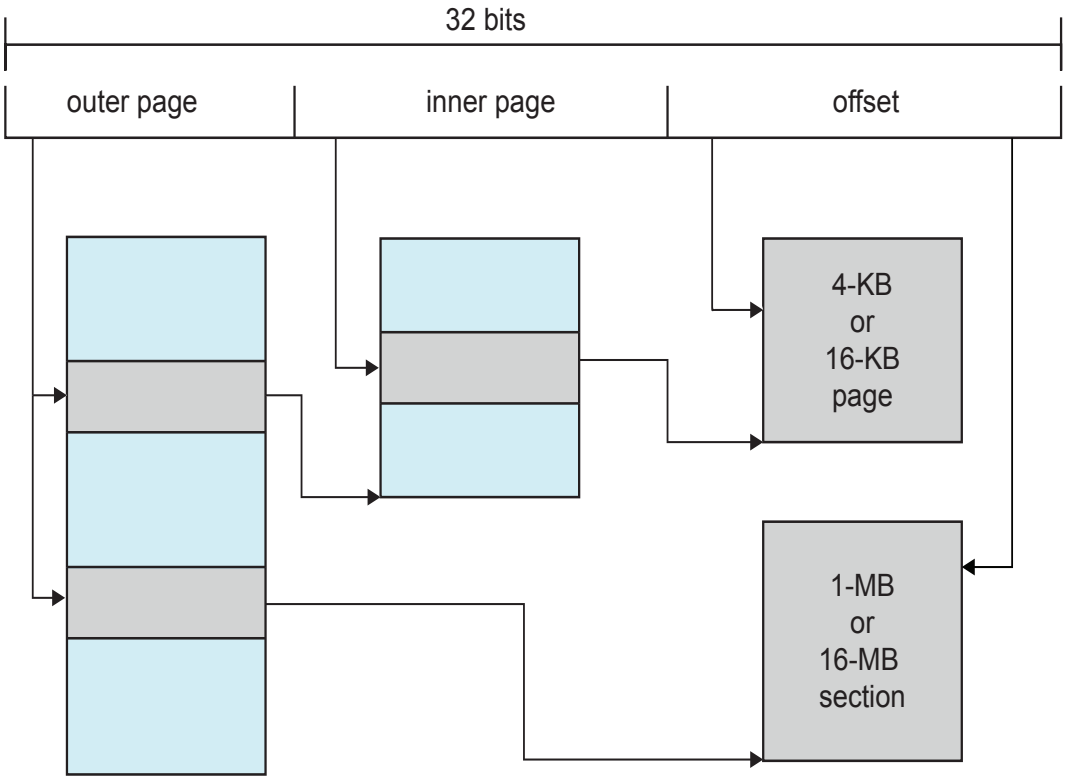
Intel x86-64 Paging

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



ARM Paging

- 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



Four-level Paging in Linux

- Abstracts paging across architecture
  - pgd: page global directory
  - pud: page upper directory
  - pmd: page middle directory
  - pte: page table entry
- Each architecture defines
  - Size of each directory, number of entries, bits
  - Bypass levels that arch doesn't have

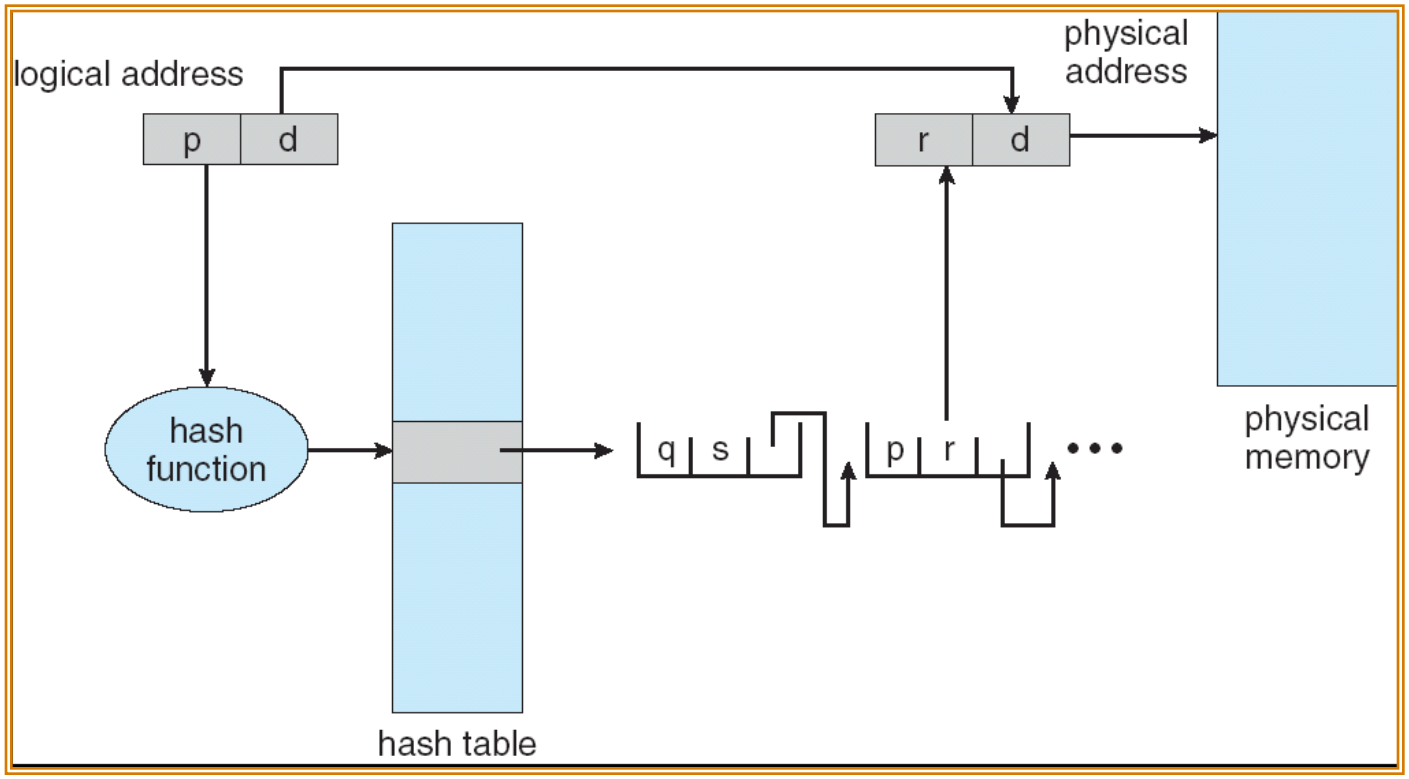
Other page table structures

- 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

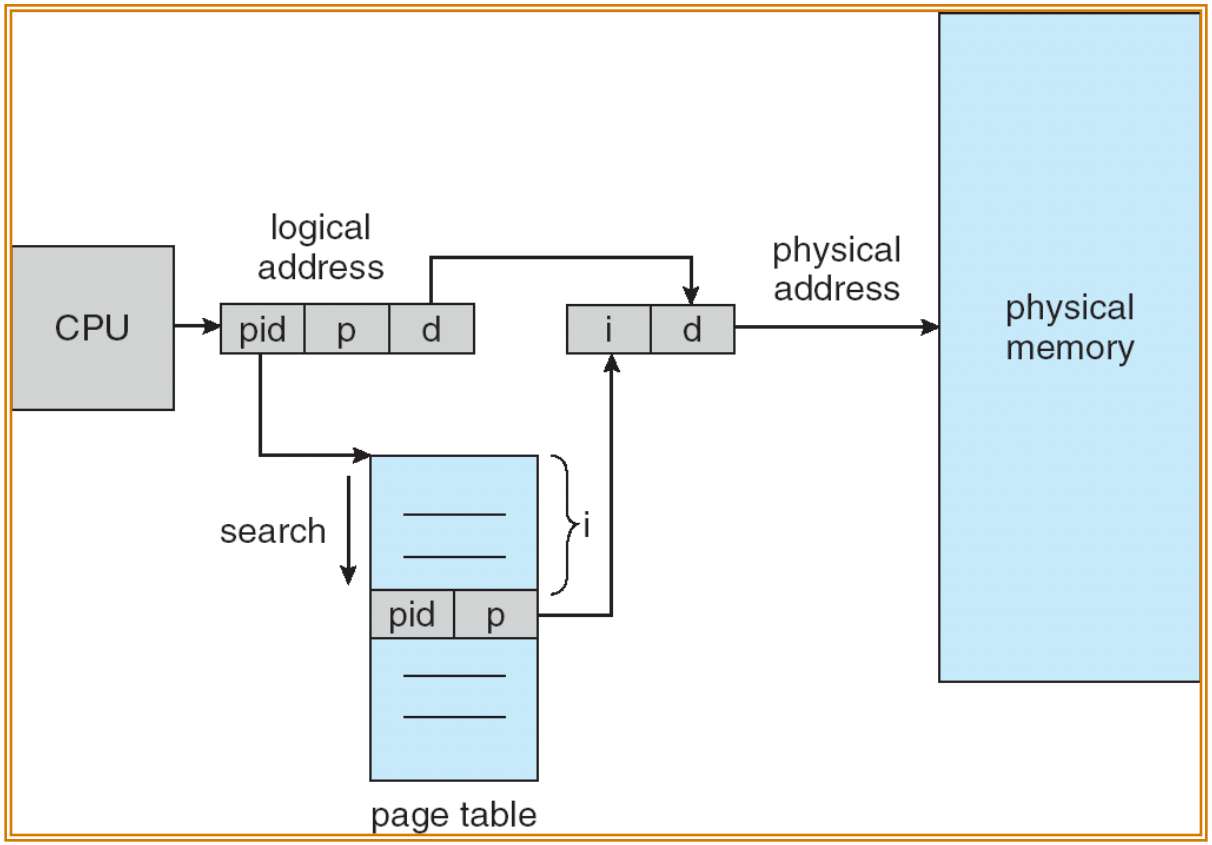
Hashed page table example



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
- Same page table shared by all processes
  - Need owner information
- Can use hash table to limit the search to one or at most a few page-table entries

Inverted page table example



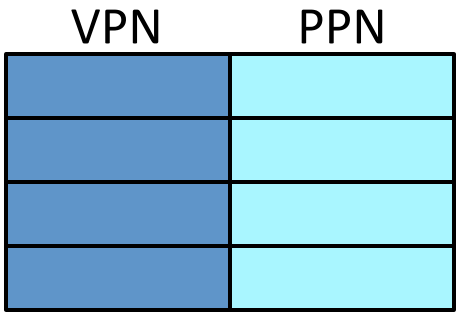
Outline

- Memory management goals
- Segmentation
- Paging
- TLB
- Page sharing

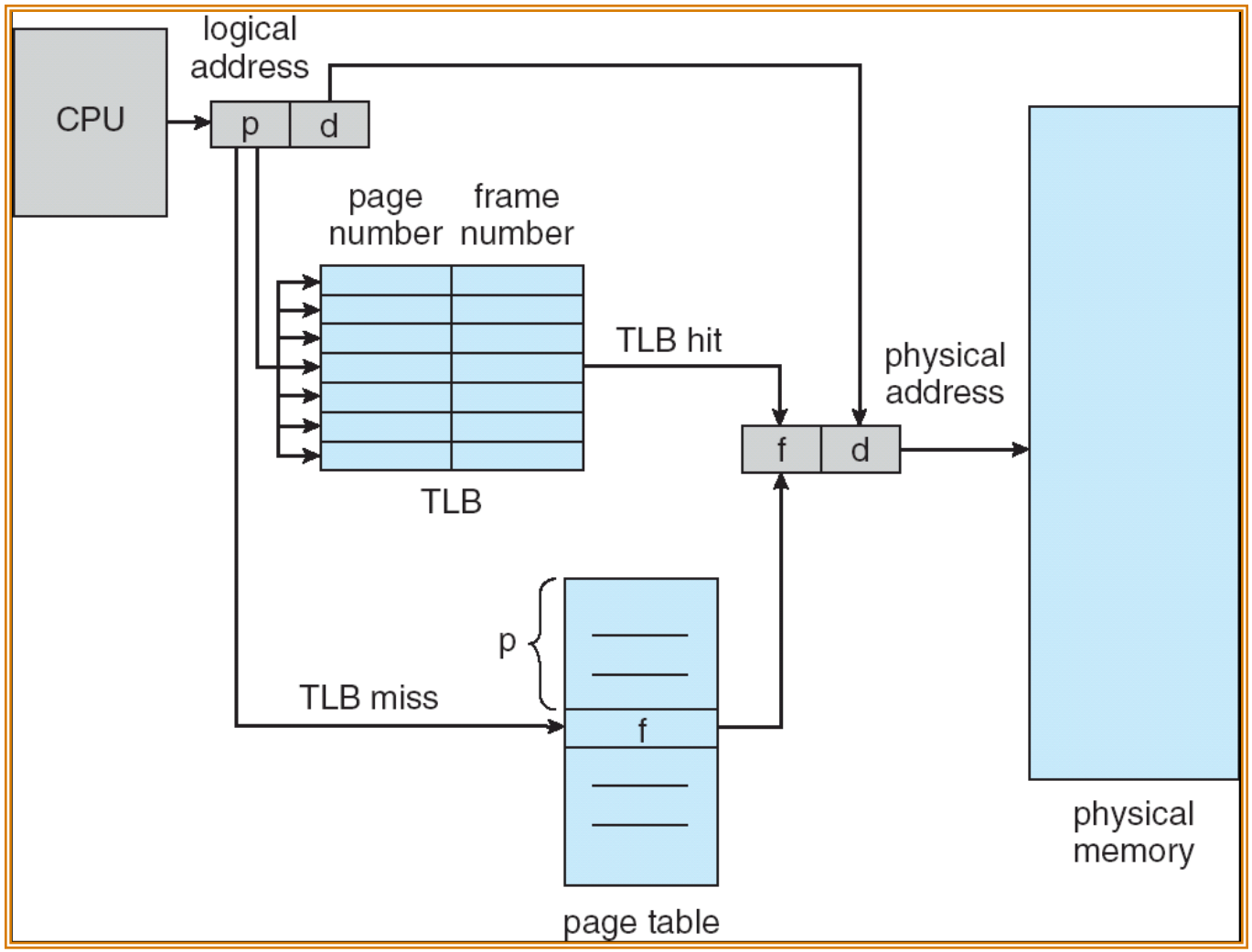
Avoiding extra memory accesses

- 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



Paging hardware with TLB



Effective access time with TLB

- Assume memory cycle time is 1 unit time
- TLB Lookup time = ε
- TLB Hit ratio = α
  - Percentage of times that a vpn➔ppn mapping is found in TLB

- Effective Access Time (EAT)

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

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)
  - In effect, TLB is hardware page table
  - Pros: simple hardware, flexible
  - Cons: code may have bug!

Reducing misses: TLB Reach

- Increase size of TLB
  - Content addressable memory (CAM) is expensive
- Increase amount of memory accessible from the TLB
  - TLB Reach = (TLB Size) X (Page Size)
  - Ideally, equal to working set
  - Otherwise lots of page faults
- Increase page size
  - More reach for same TLB size
  - Increase in fragmentation as well
- Provide multiple page sizes
  - Applications can choose which size fits their access pattern
  - Doesn't increase fragmentation

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
- Option 2: attach process ID to TLB entries
  - ASID: Address Space Identifier
  - MIPS, SPARC
- x86 "INVLPG addr" invalidates one TLB entry

Address Space IDs (ASID

Mechanism to reduce frequency of TLB invalidations

Without ASID:

VPN	PPN	valid	prot
0	10	1	rwx
----	----	0	---
0	17	1	rwx
----	----	0	---

With ASID:

VPN	PPN	valid	prot	ASID
0	10	1	rwx	1
----	----	0	---	----
0	17	1	rwx	2
----	----	0	---	----

Choosing a page size

- Many CPUs support multiple page sizes
- Page size selection affects (or is affected by):
  - Fragmentation?
    - Smaller is better.
  - Page table size?
    - Bigger is more efficient.
  - I/O overhead?
    - Larger is better (fewer seeks).
  - Resolution (locality)?
    - Smaller is better.
  - Number of page faults?
    - Larger or smaller could be better. 1 page per byte vs. 1 page for entire mem.
  - TLB size and effectiveness?
    - Larger is better.
- On average, growing over time

Outline

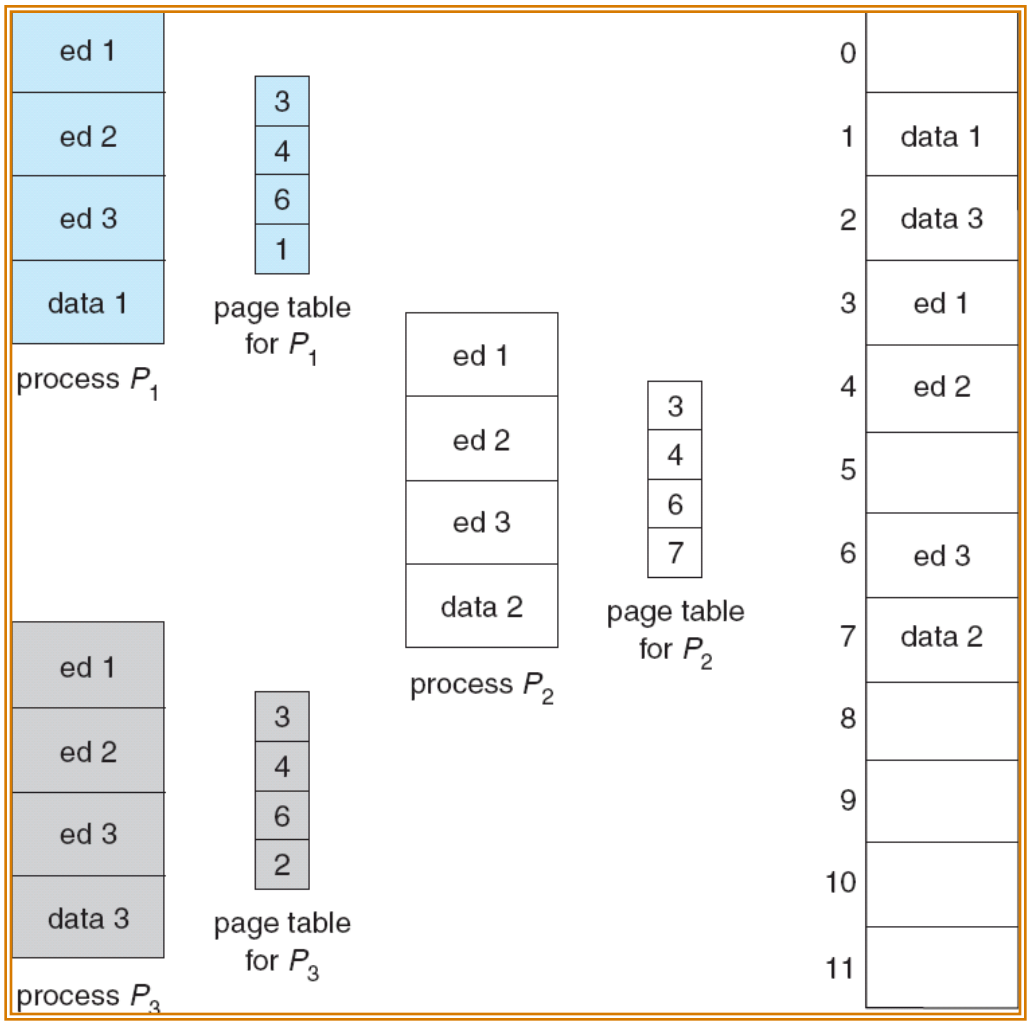
- Memory management goals
- Segmentation
- Paging
- TLB
- Page sharing

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



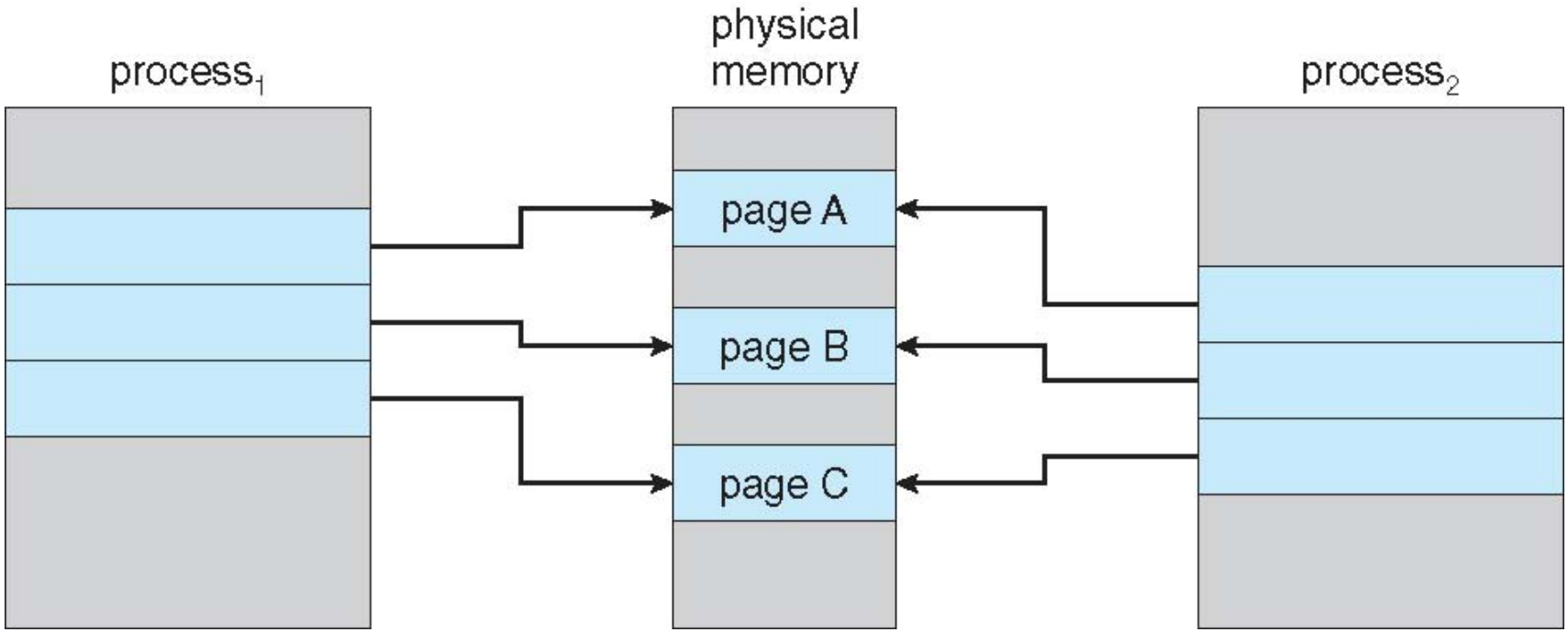
A cool trick: copy-on-write

- In fork(), parent and child often share significant amount of memory
  - Expensive to copy all pages
- COW Idea: exploit VA to PA indirection
  - Instead of copying all pages, share them
  - If either process writes to shared pages, only then is the page copied
- Real: used in virtually all modern OS

How to implement COW?

- (Ab)use page protection
- Mark pages as read-only in both parent and child address space
- On write, page fault occurs
- In OS page fault handler, distinguish COW fault from real fault
  - How?
- Copy page and update page table if COW fault
  - Always copy page?

Before Process 1 Modifies Page C



After Process 1 Modifies Page C