Memory Management I
Paging

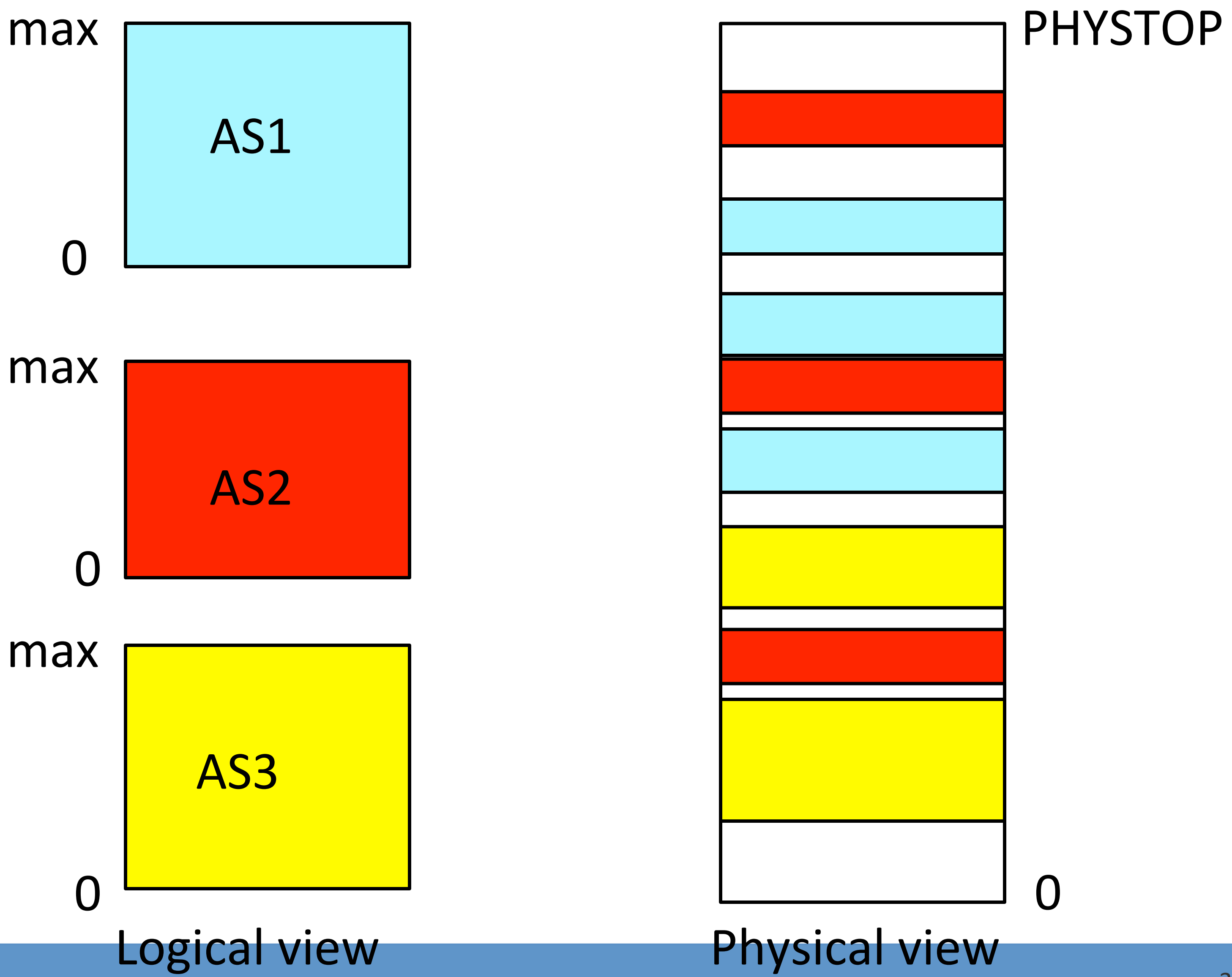
COMS W4118

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

Outline

- Overview
- Paging
- TLB
- Page sharing

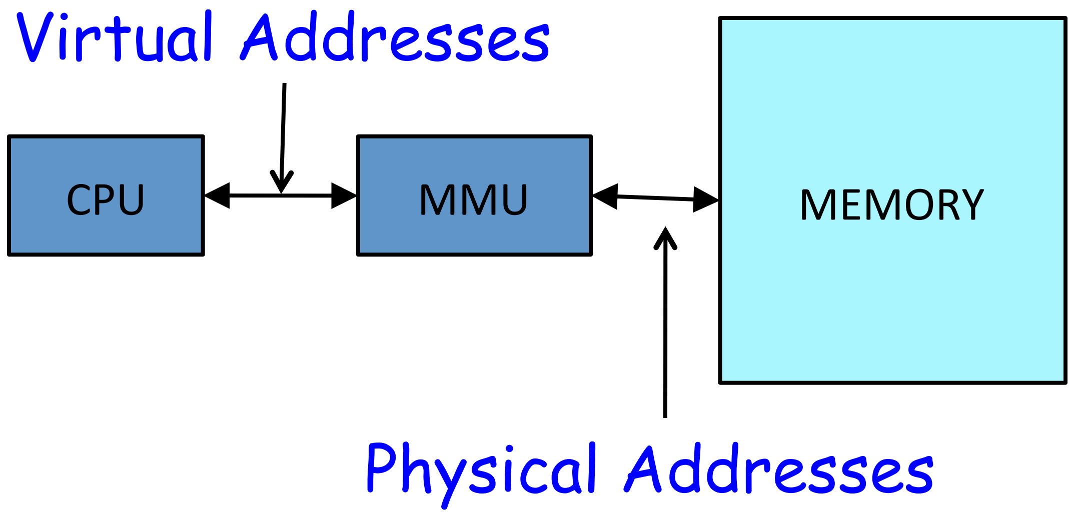
Multiple address spaces co-exist



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)

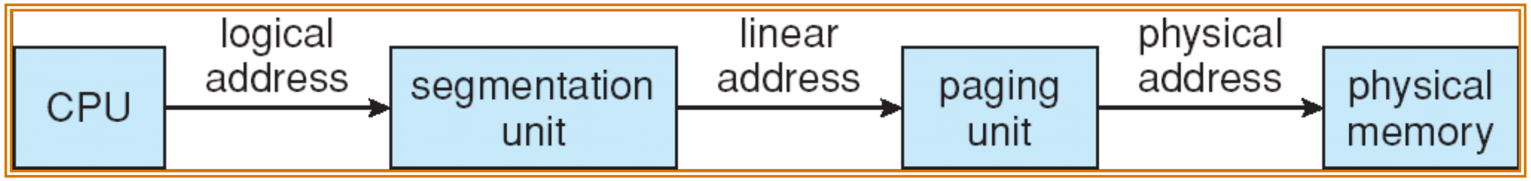
Memory Management Unit (MMU)



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

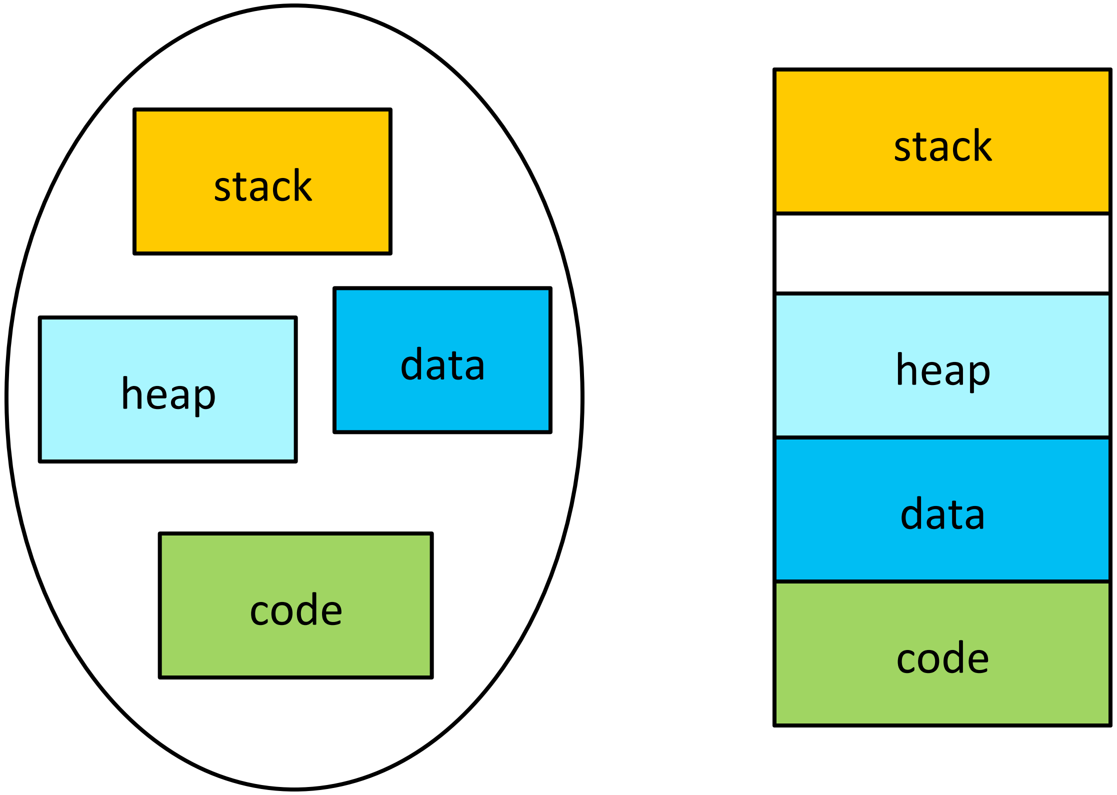
x86 address translation

- 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

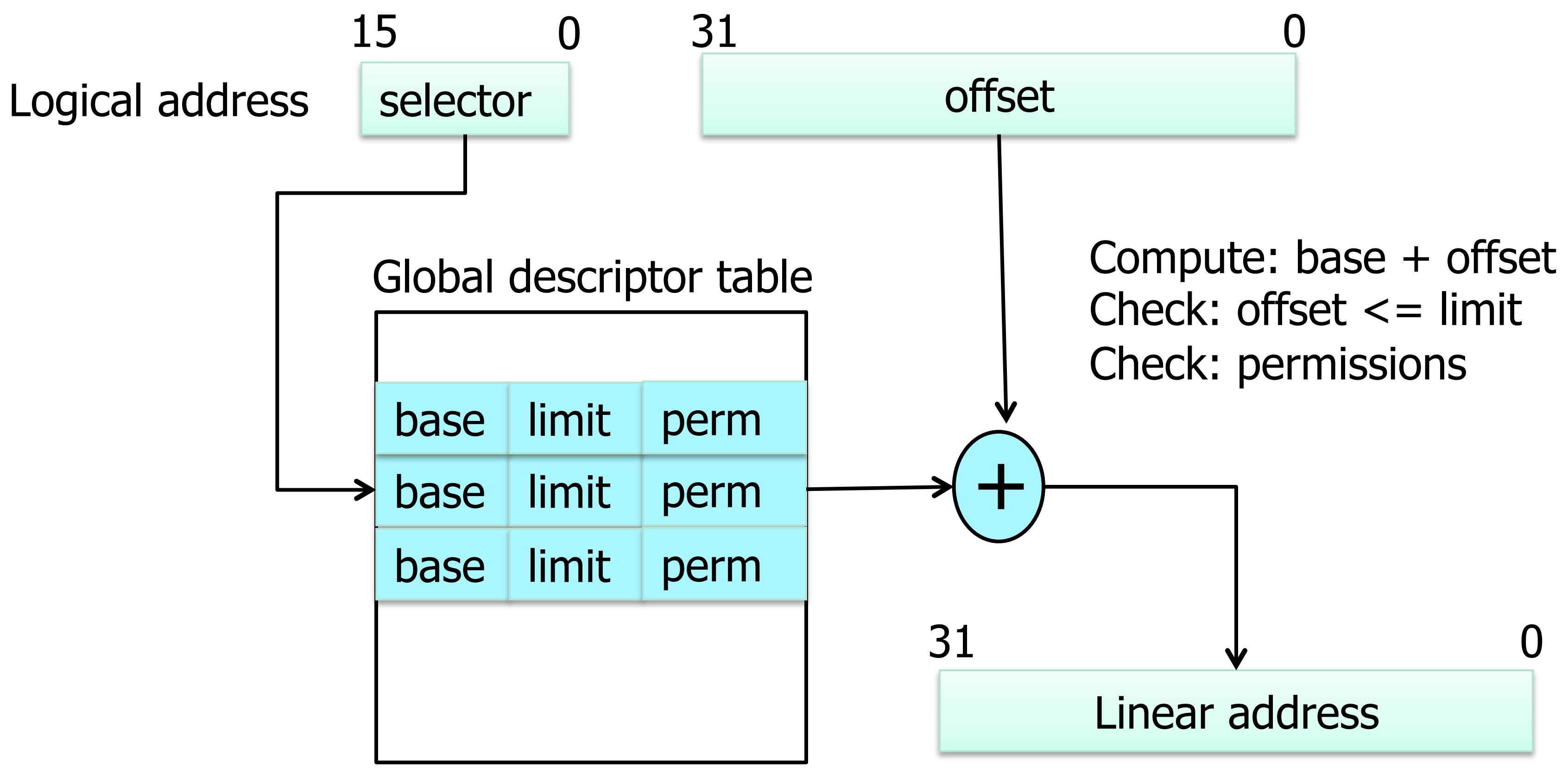


Segmentation

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



x86 segmentation hardware



Compute: base + offset
Check: offset <= limit
Check: permissions

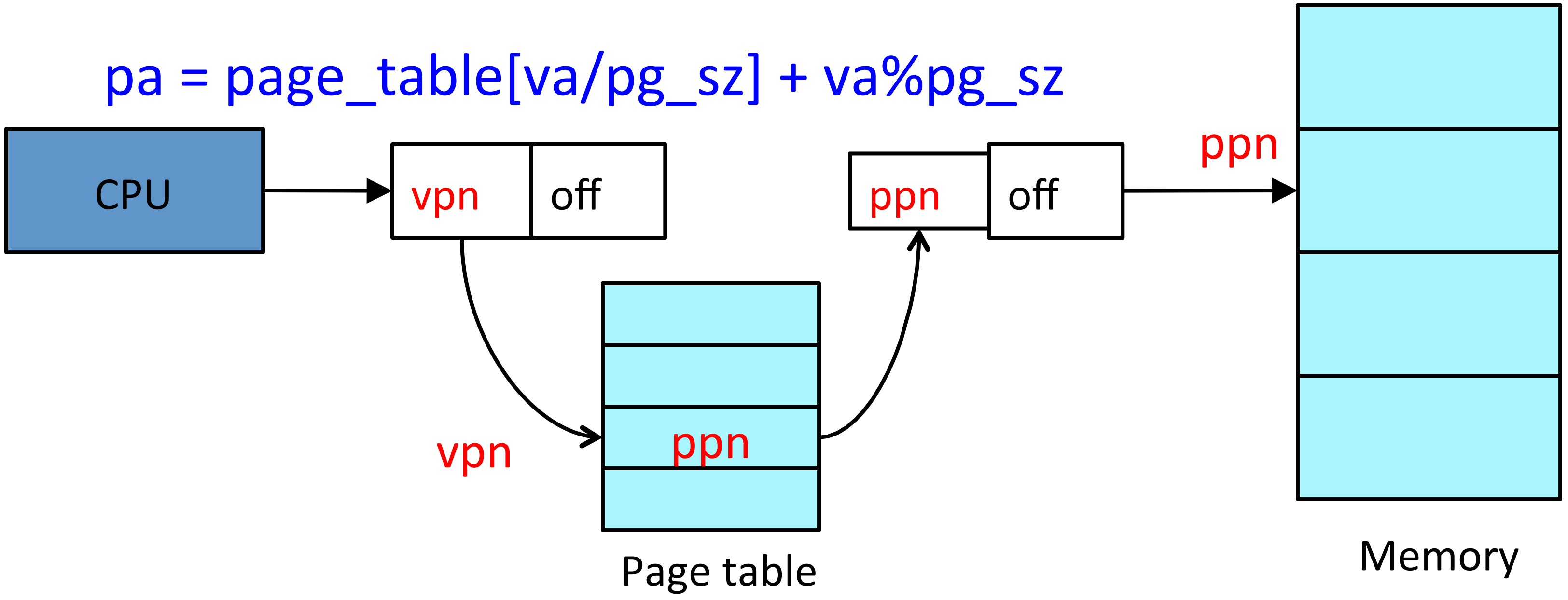
Paging overview

- 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

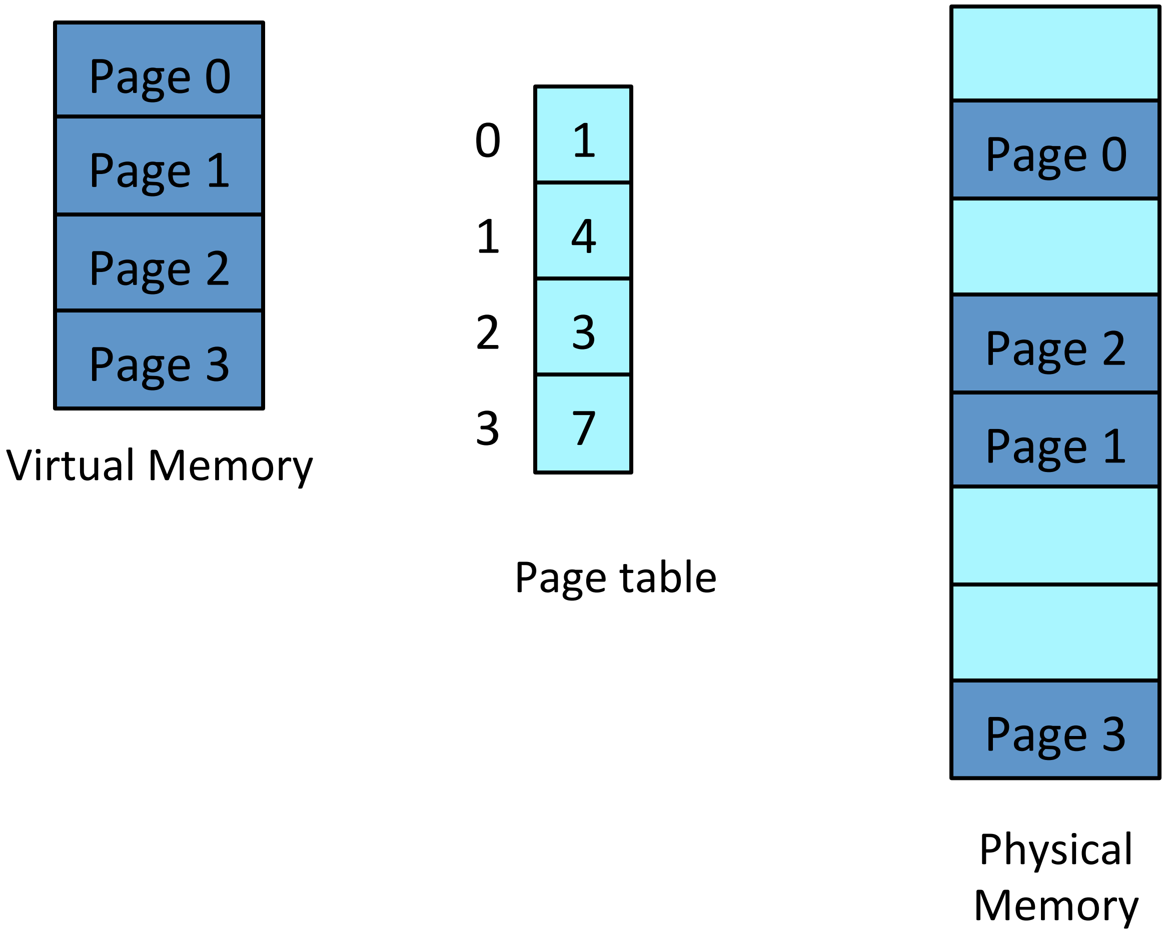
Page translation

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

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



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

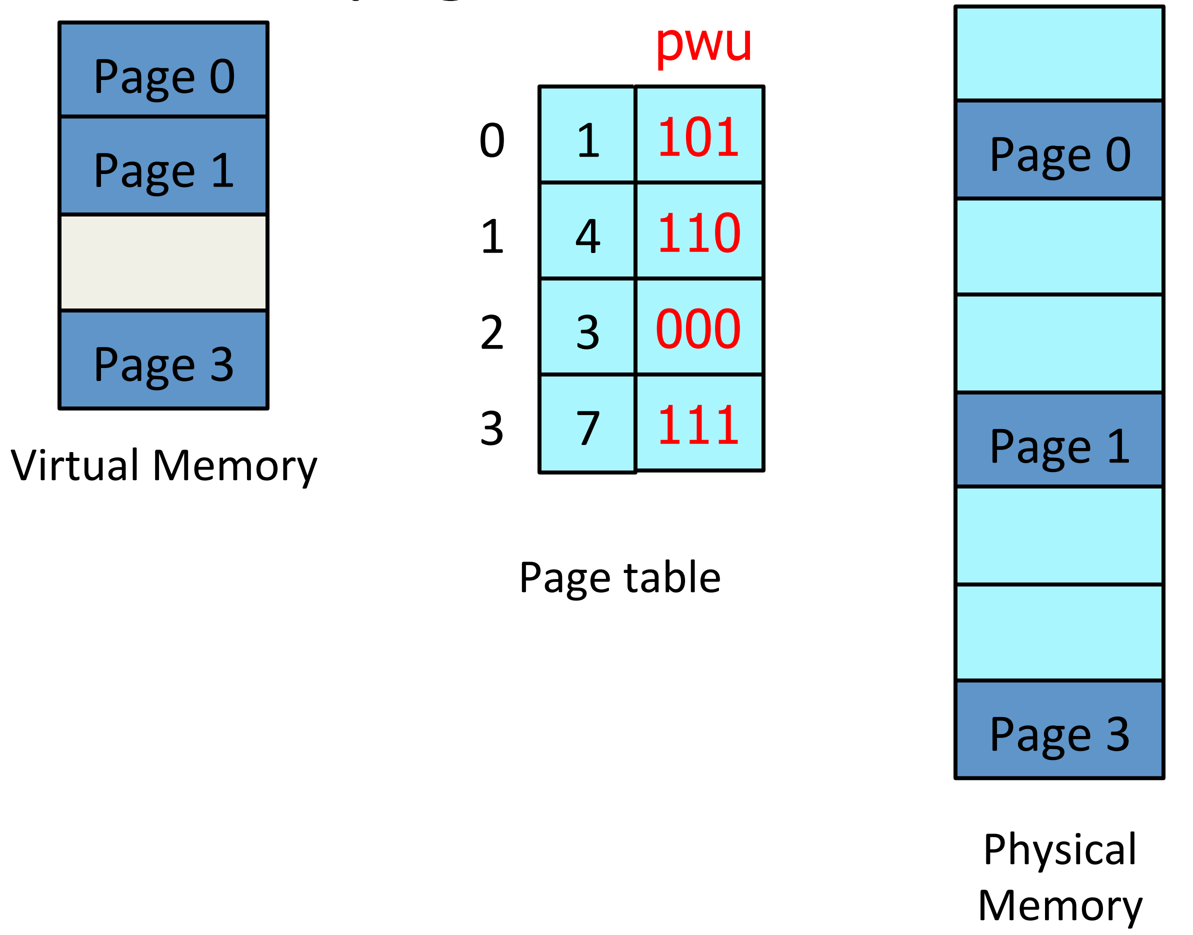
- # of virtual pages: $2^{(m-k)}$
- # of physical pages: $2^{(n-k)}$
- # of entries in page table: $2^{(m-k)}$
- vpn = va / 2^k
- offset = va % 2^k
- ppn = page_table[vpn]
- pa = ppn * 2^k + offset

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?



		pwu
0	1	101
1	4	110
2	3	000
3	7	111

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

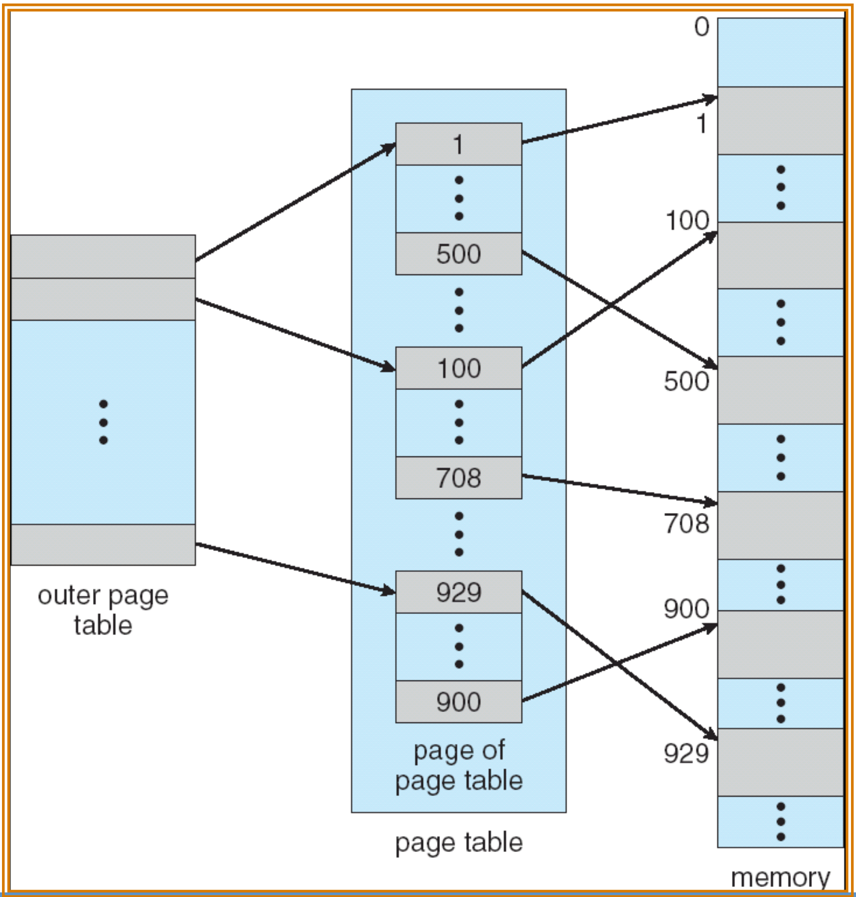
- Problem: each data/instruction access requires two memory accesses
  - Extra memory access for page table

Page table size issues

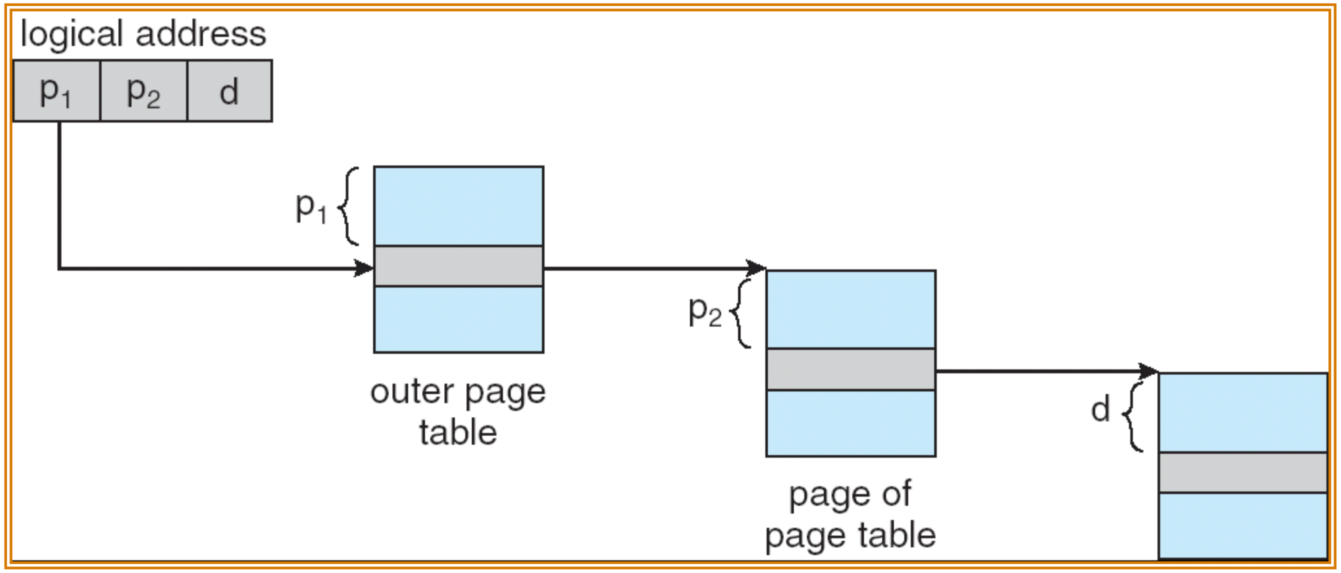
- Given:
  - A 32 bit address space (4 GB)
  - 4 KB pages
  - A page table entry of 4 bytes
- Implication: page table is 4 MB per process!
- Observation: address space are often sparse
  - Few programs use all of 2^{32} bytes
- Change page table structures to save memory
  - Trade translation time for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

Hierarchical page table

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

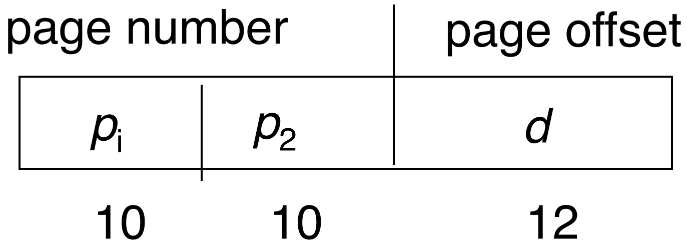


Hierarchical page tables

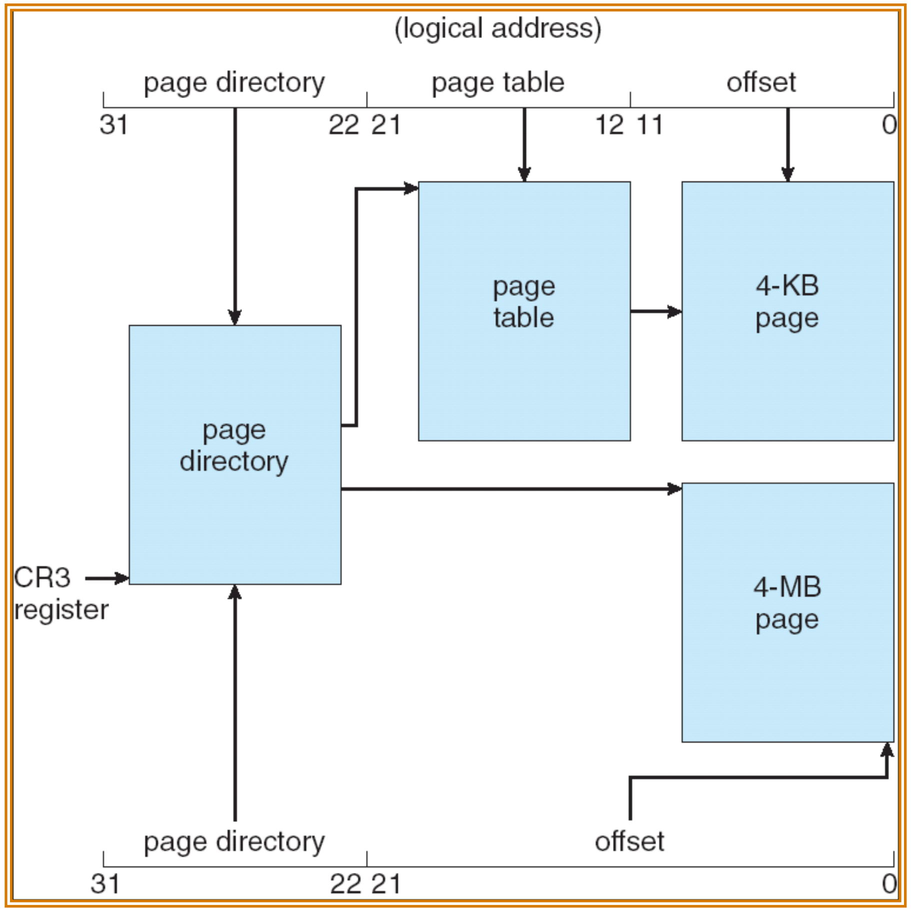


x86 page translation with 4KB pages

- 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



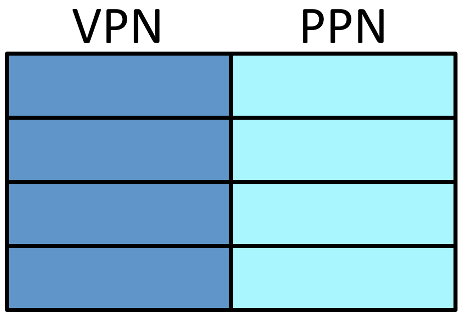
x86 paging architecture



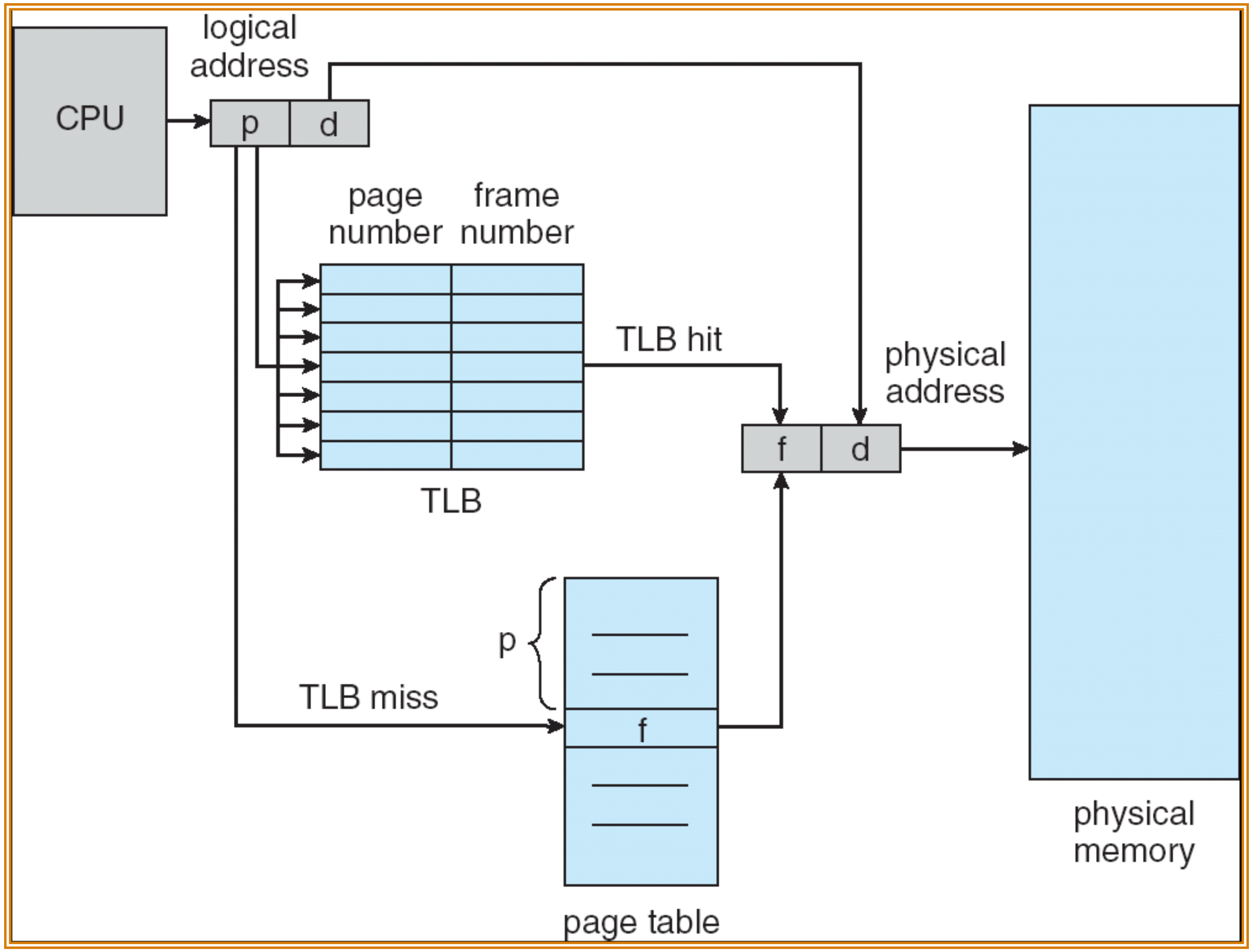
- A better picture here (page 26):
  - http://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf

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)

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

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

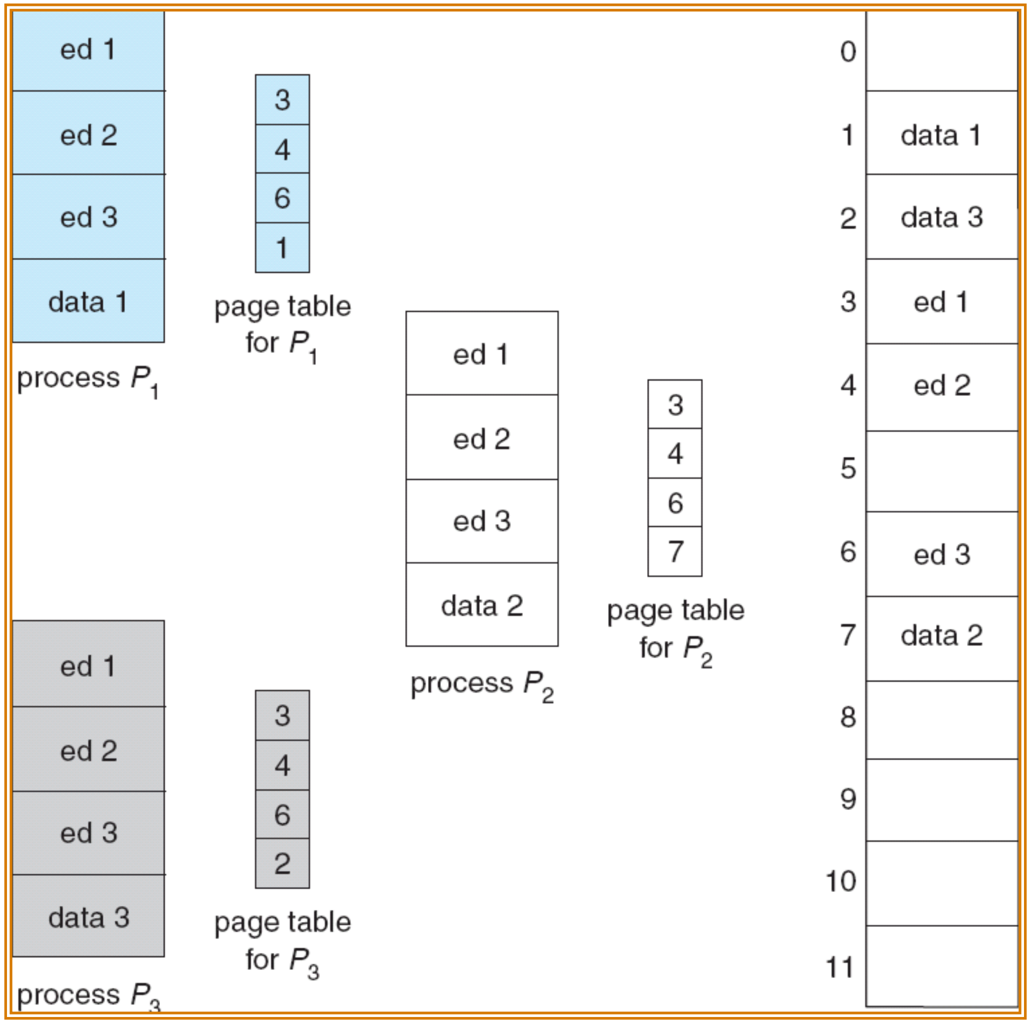
- x86 “INVLPG addr” invalidates one TLB entry

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

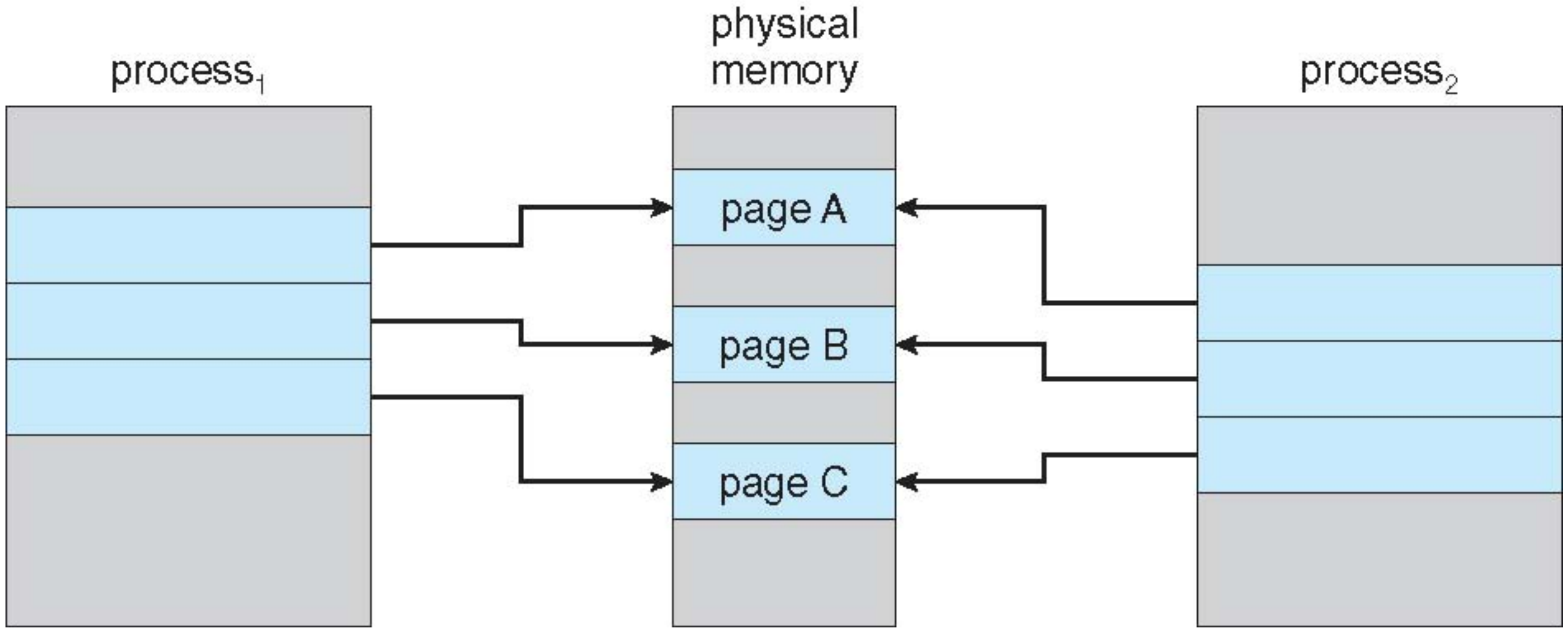
Page sharing example



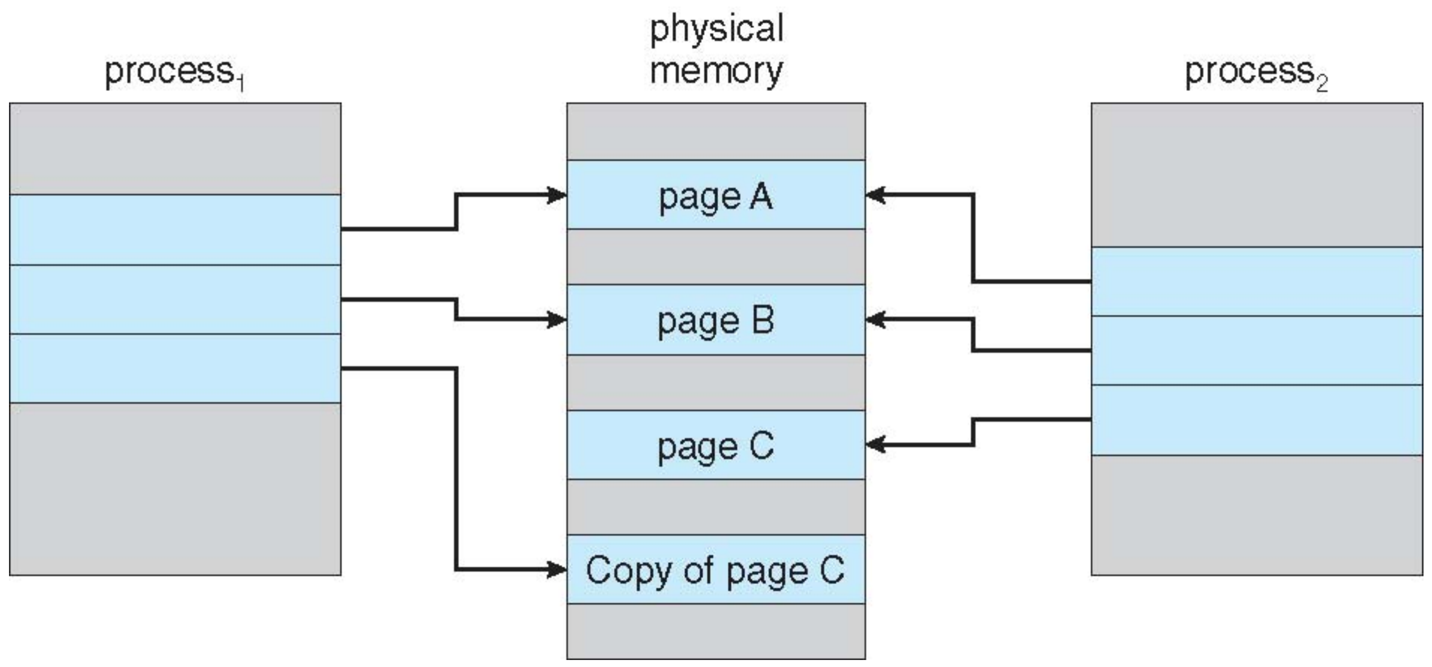
A cool trick: copy-on-write

- 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

Before Process 1 Modifies Page C



After Process 1 Modifies Page C


process$_1$
physical memory
process$_2$
page A
page B
page C
Copy of page C