Module 13 – Main Memory

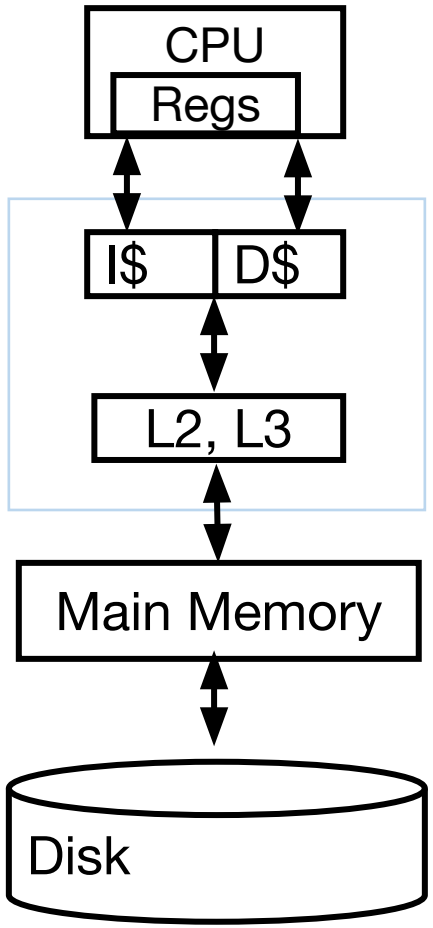
(ECE M116C- CS M151B) Computer Architecture Systems

Nader Sehatbakhsh
Department of Electrical and Computer Engineering
University of California, Los Angeles

Last time…

- Use memory hierarchy to achieve both *fast* and *large* memory system.
  - Cache performance: hit time, miss rate, and miss penalty.
  - Replacement policy
  - How to improve each metric

Main Memory



Technology

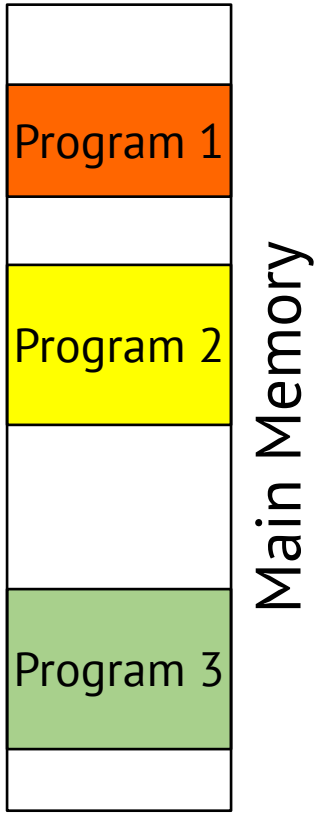
- We see combination of DRAM and DISK as the main memory (usually call the DRAM part Main Memory and the HardDrive part DISK).

- Access to the Main Memory (combination of disk and DRAM) *is always a hit*.
  - As we will describe later, data could be in disk and/or DRAM, and we need to handle that.
  - Huge difference between latency of DRAM to DISK (and of course to cache).

Where to store different programs?

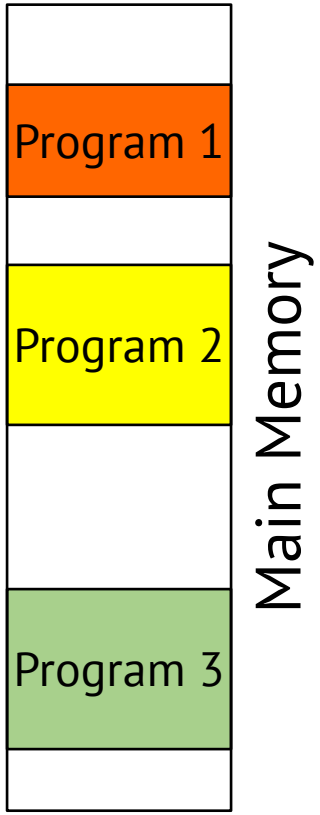
➔ *How to run different programs in a computer?*

- Fundamental Requirement: *Isolation*



Where to store different programs?

- Each program typically has *different regions:*
  - code
  - data
  - heap
  - stack
  - …
- Portion of the memory code be shared among multiple programs.
- Portion of the memory can be reserved for the OS and/or other privileged activities.



Memory Management

- Early (and simple) machines only run one program with unrestricted access to ALL memory.

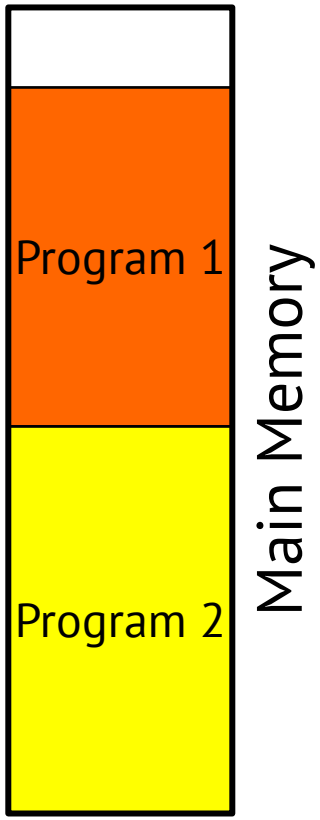
- Modern systems run several programs, and memory should be shared between multiple programs.

- Memory management is controlled by OS or system software, or hardware.

How to share the memory between multiple programs?

How to share the memory between multiple programs?

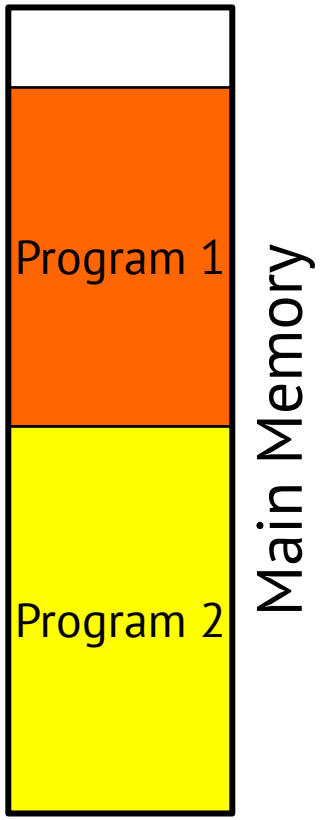
★ *Option 1: partition the memory (statically)!*



How to share the memory between multiple programs?

★ *Option 1: partition the memory (statically)!*

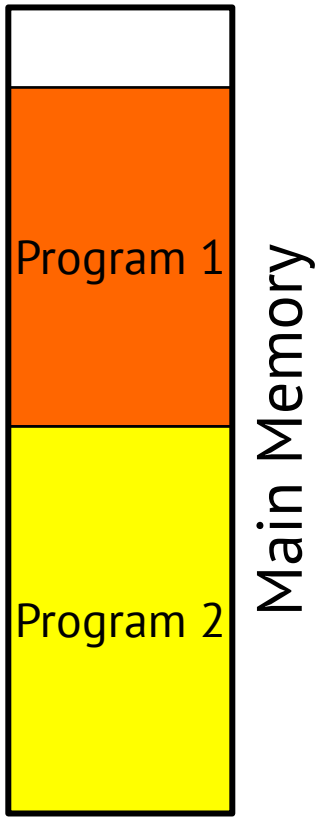
- Considerations:
  - *Protection:* Independent program should not be able to affect each other inadvertently.
  - *Location-Independence:* Program might be moved anywhere in the memory.



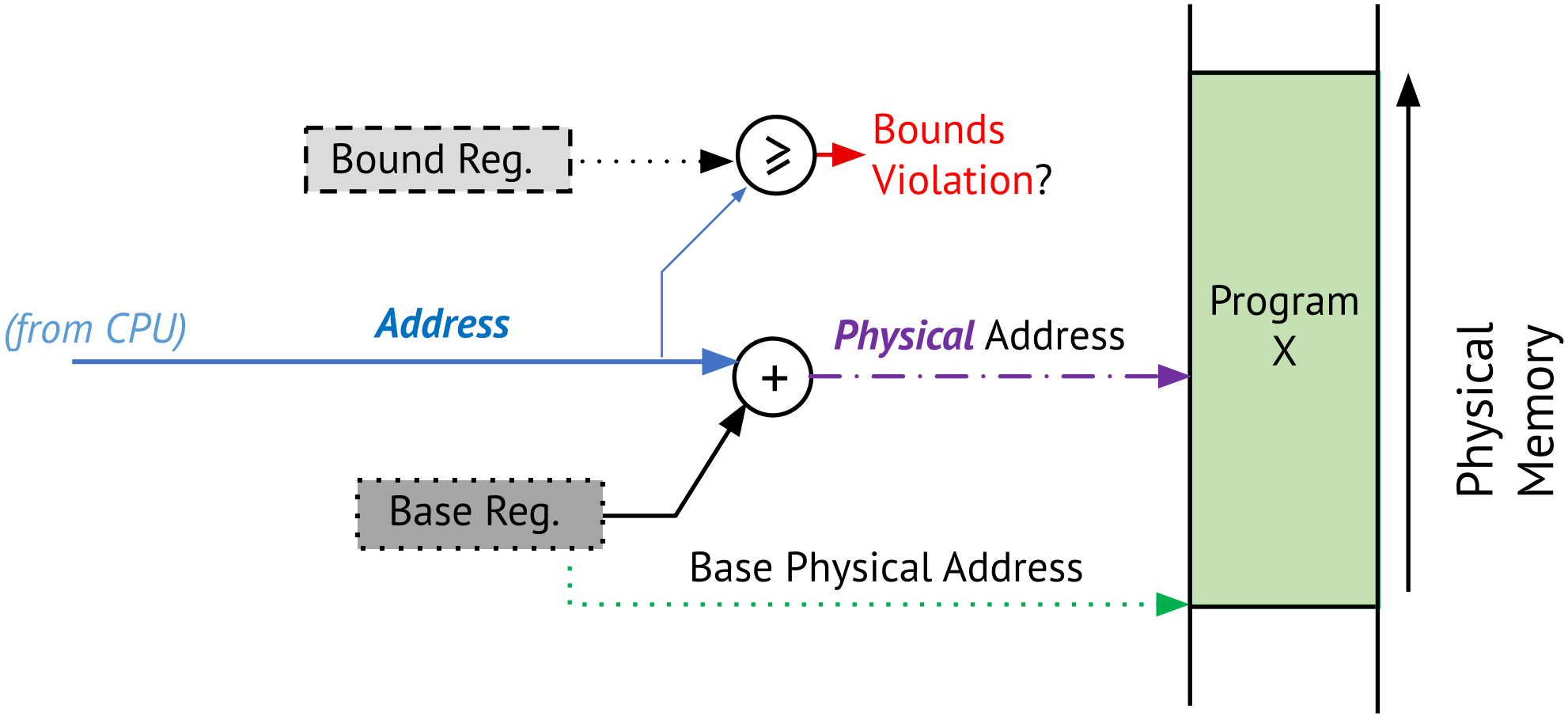
How to share the memory between multiple programs?

★ *Option 1: partition the memory (statically)!*

- Considerations:
  - *Protection:* Independent program should not be able to affect each other inadvertently.
    → *check bounds!*
  - *Location-Independence:* Program might be moved anywhere in the memory.
    → *use a base pointer!*



Static Partitioning



Nader Sehatbakhsh <nsehat@ee.ucla.edu>

Address Translation

- To access the memory, addresses in a program (typically called logical or effective address) should be translated into a physical address.

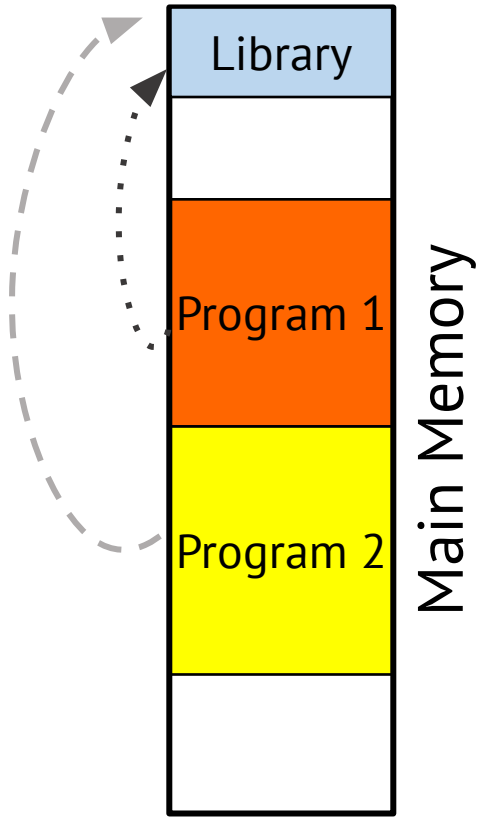
Example:

$$PA = Base\_Reg + Logical\_Adr$$

- When switching programs, OS updates base and bound registers.

How to share code/libraries?

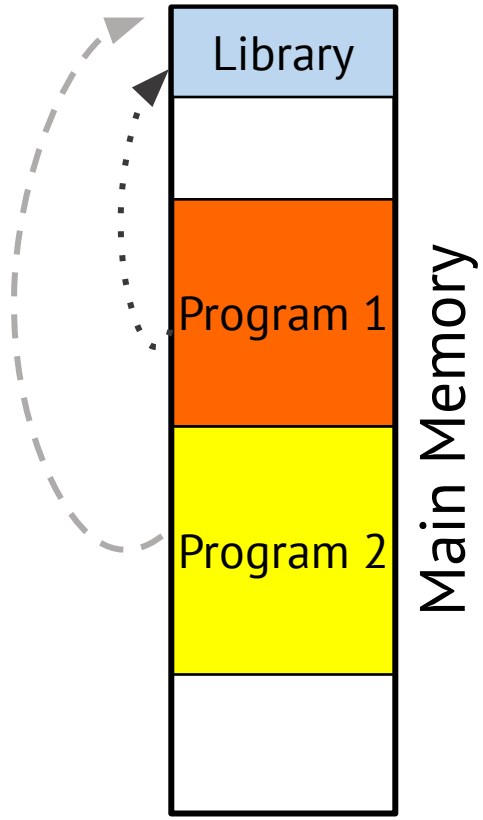
- Multiple programs can access (read-only) shared memory spaces.
  - Standard libraries (e.g., printf)
  - Drivers
  - etc.



How to share code/libraries?

- Multiple programs can access (read-only) shared memory spaces.
  - Standard libraries (e.g., printf)
  - Drivers
  - etc.

❖ *What about the OS?*

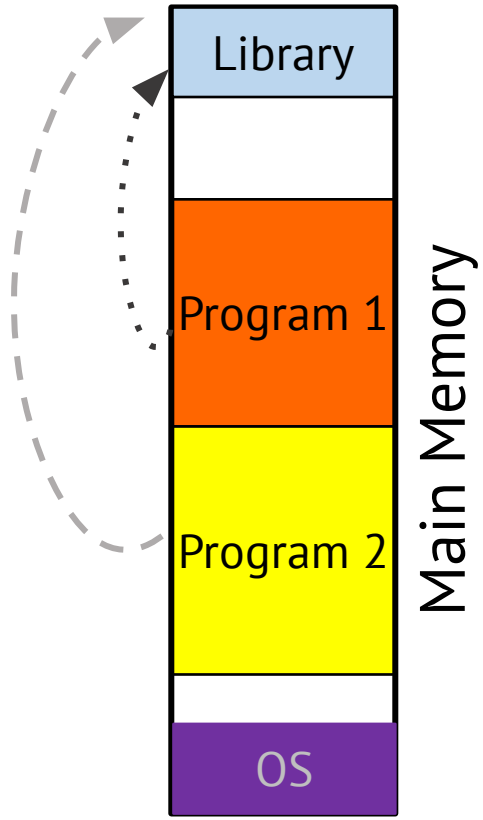


How to share code/libraries?

- Multiple programs can access (read-only) shared memory spaces.
  - Standard libraries (e.g., printf)
  - Drivers
  - etc.

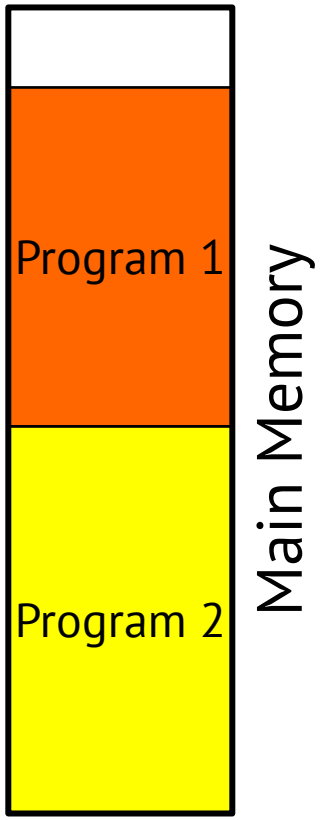
❖ *What about the OS?*

- It's a program as well!



How to share the memory between multiple programs?

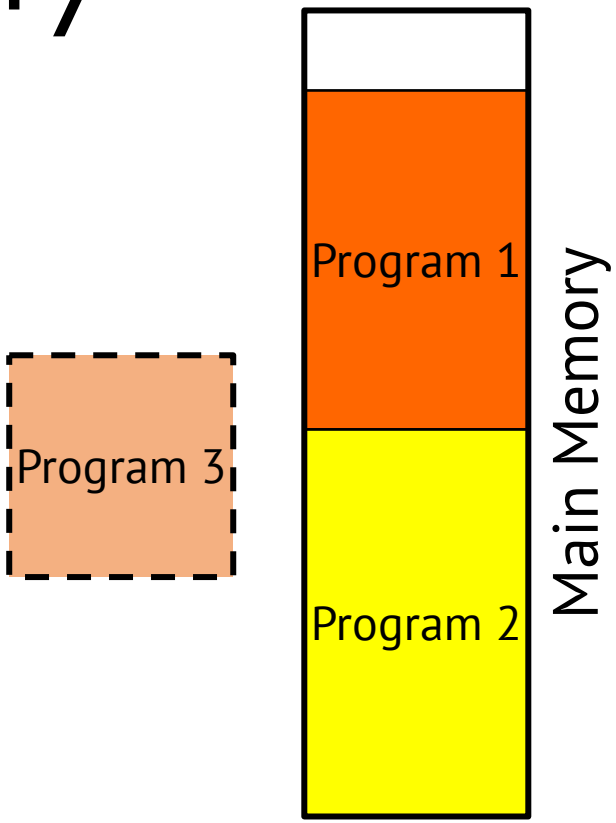
★ *Option 1: partition the memory (statically)!*



How to share the memory between multiple programs?

★ *Option 1: partition the memory (statically)!*

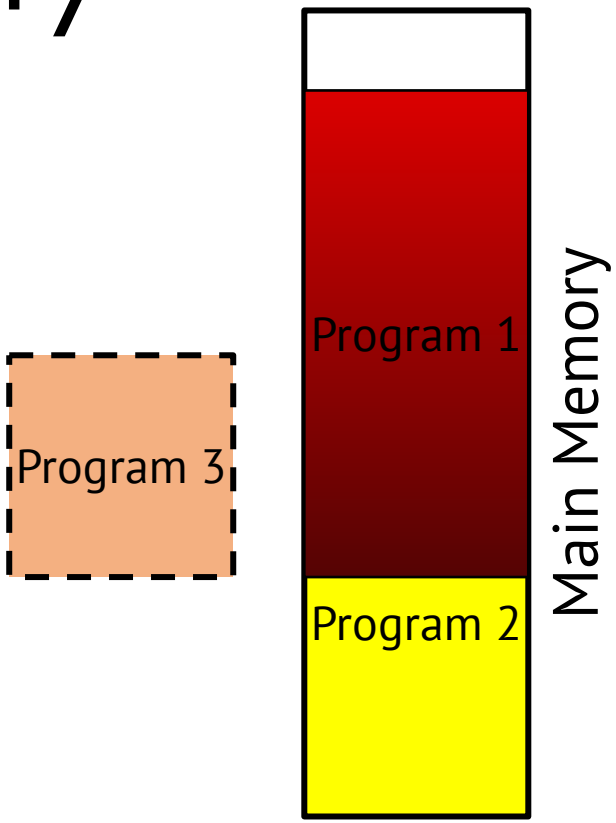
- What happens if we add a new program?



How to share the memory between multiple programs?

★ *Option 1: partition the memory (statically)!*

- *What happens if we add a new program?*
- *What if one program need more (less) space?*



How to share the memory between multiple programs?

★ *Option 2:*

How to share the memory between multiple programs?

★ *Option 2: partition the memory (dynamically)!*

How to share the memory between multiple programs?

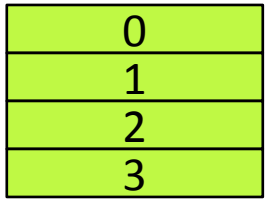
★ *Option 2: partition the memory (dynamically)!*

- Divide the memory into fixed-size blocks (called page).

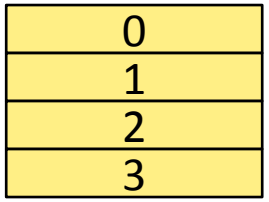
How to share the memory between multiple programs?

★ *Option 2: partition the memory (dynamically)!*

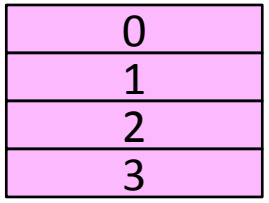
- Divide the memory into fixed-size blocks (called page).



Program A



Program B

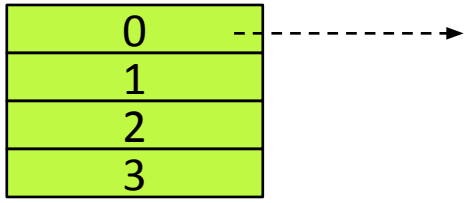


Program C

How to share the memory between multiple programs?

★ *Option 2: partition the memory (dynamically)!*

- Divide the memory into fixed-size blocks (called page).



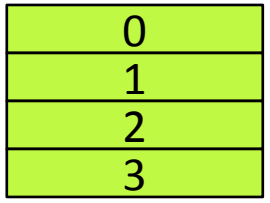
Program A

Each page is typically 4KB. For example, we can use the lower 12 bits of each address to compute its page number.

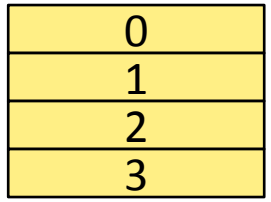
How to share the memory between multiple programs?

★ *Option 2: partition the memory (dynamically)!*

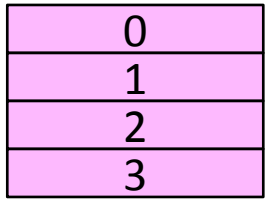
- Divide the memory into fixed-size blocks (called page).



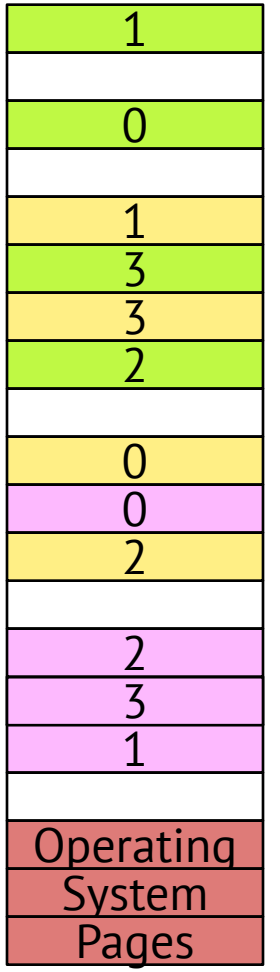
Program A



Program B



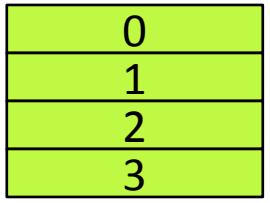
Program C



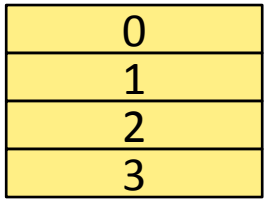
How to share the memory between multiple programs?

★ *Option 2: partition the memory (dynamically)!*

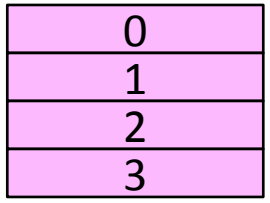
- Divide the memory into fixed-size blocks (called page).
- Assign pages to each program as needed.
- Pages can be *scattered* in the memory.
- Typically 4KB per page.



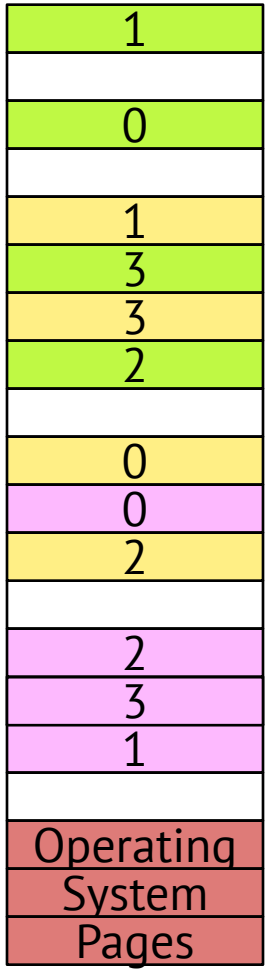
Program A



Program B

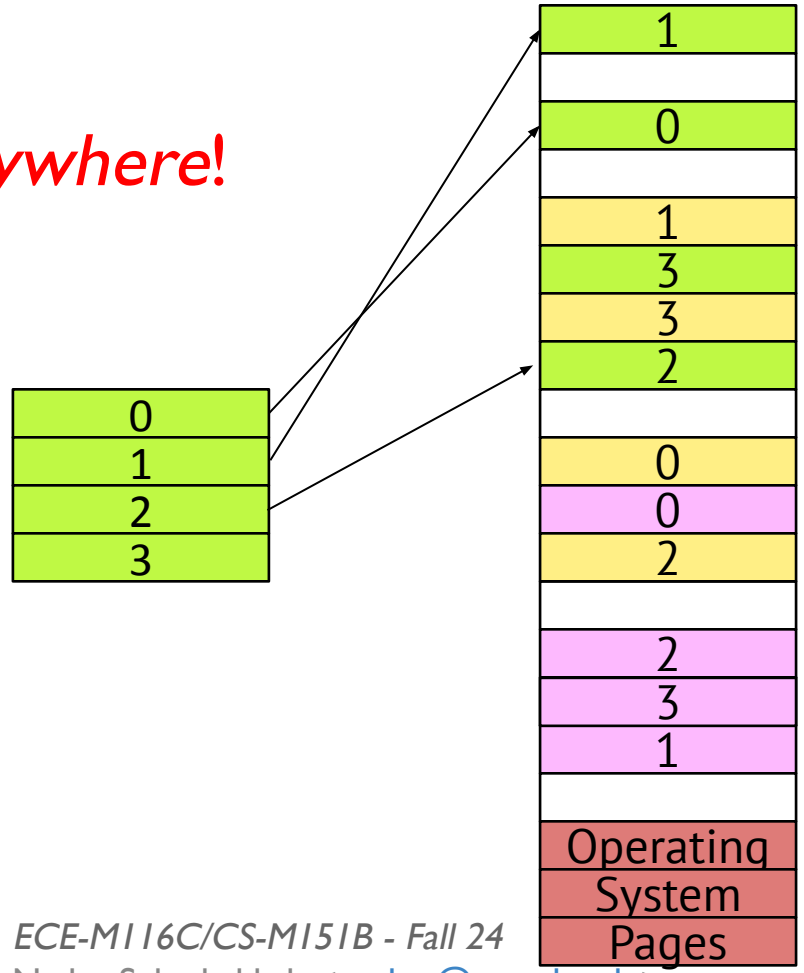


Program C



How to track which page is where?

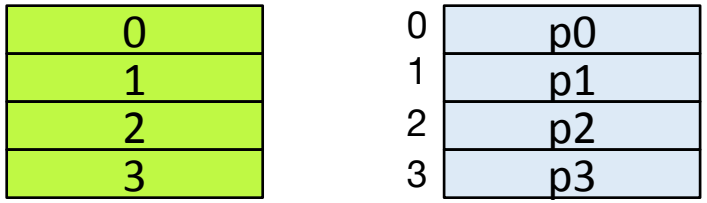
- Each page could be *anywhere*!



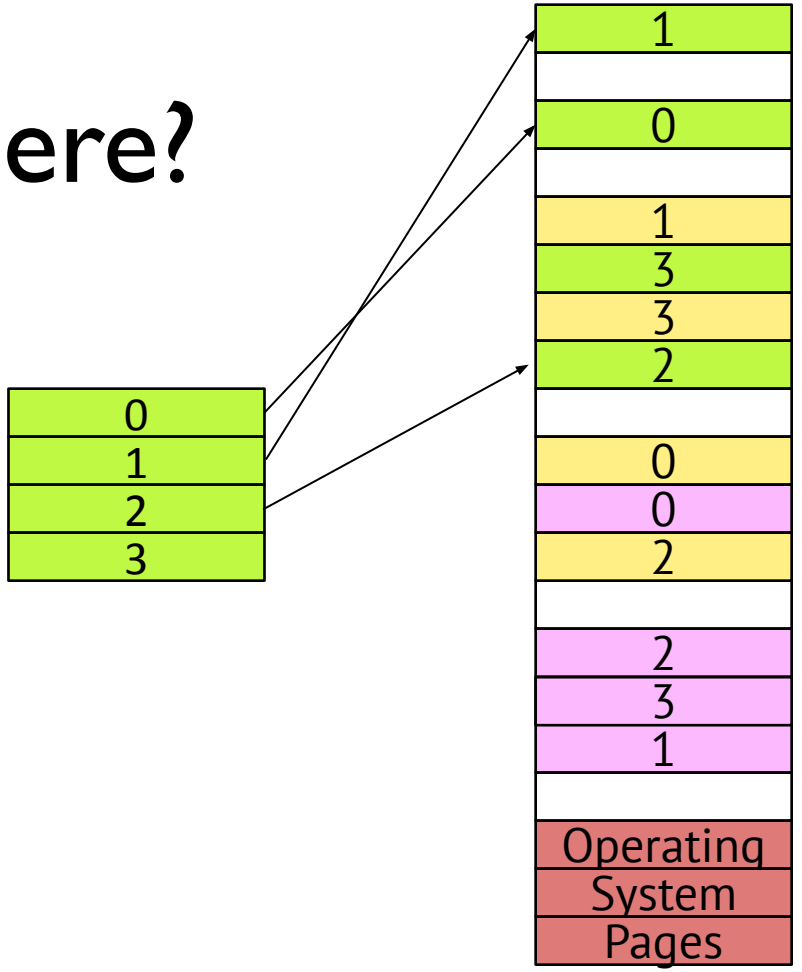
How to track which page is where?

- Each page could be *anywhere*!

-- Use a table to keep track of mapping!



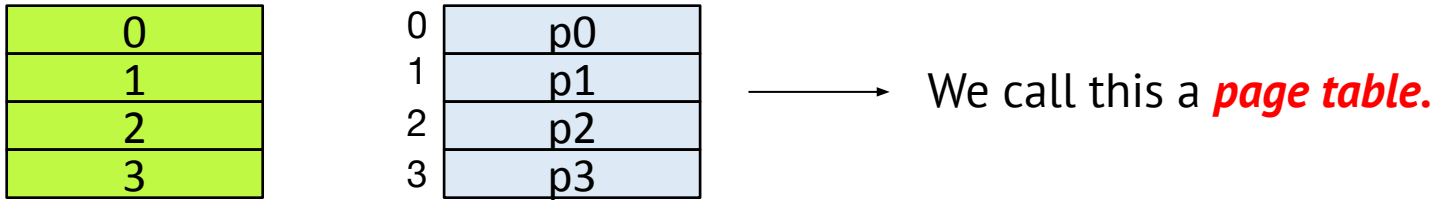
0	p0
1	p1
2	p2
3	p3

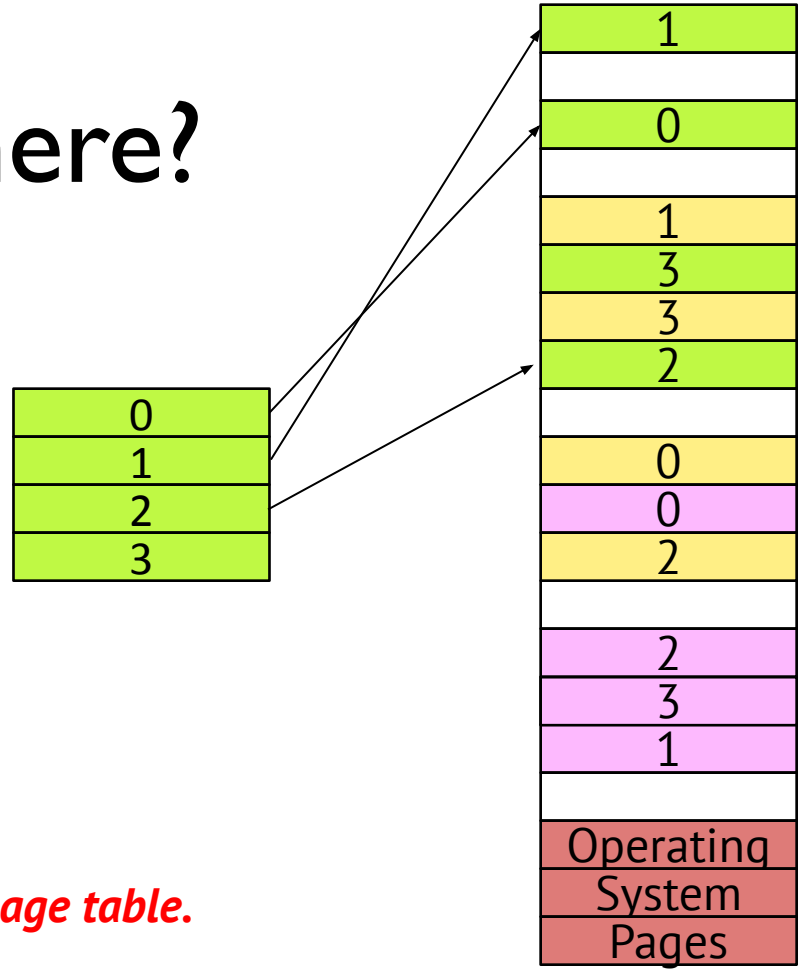


How to track which page is where?

- Each page could be *anywhere*!

-- Use a table to keep track of mapping!

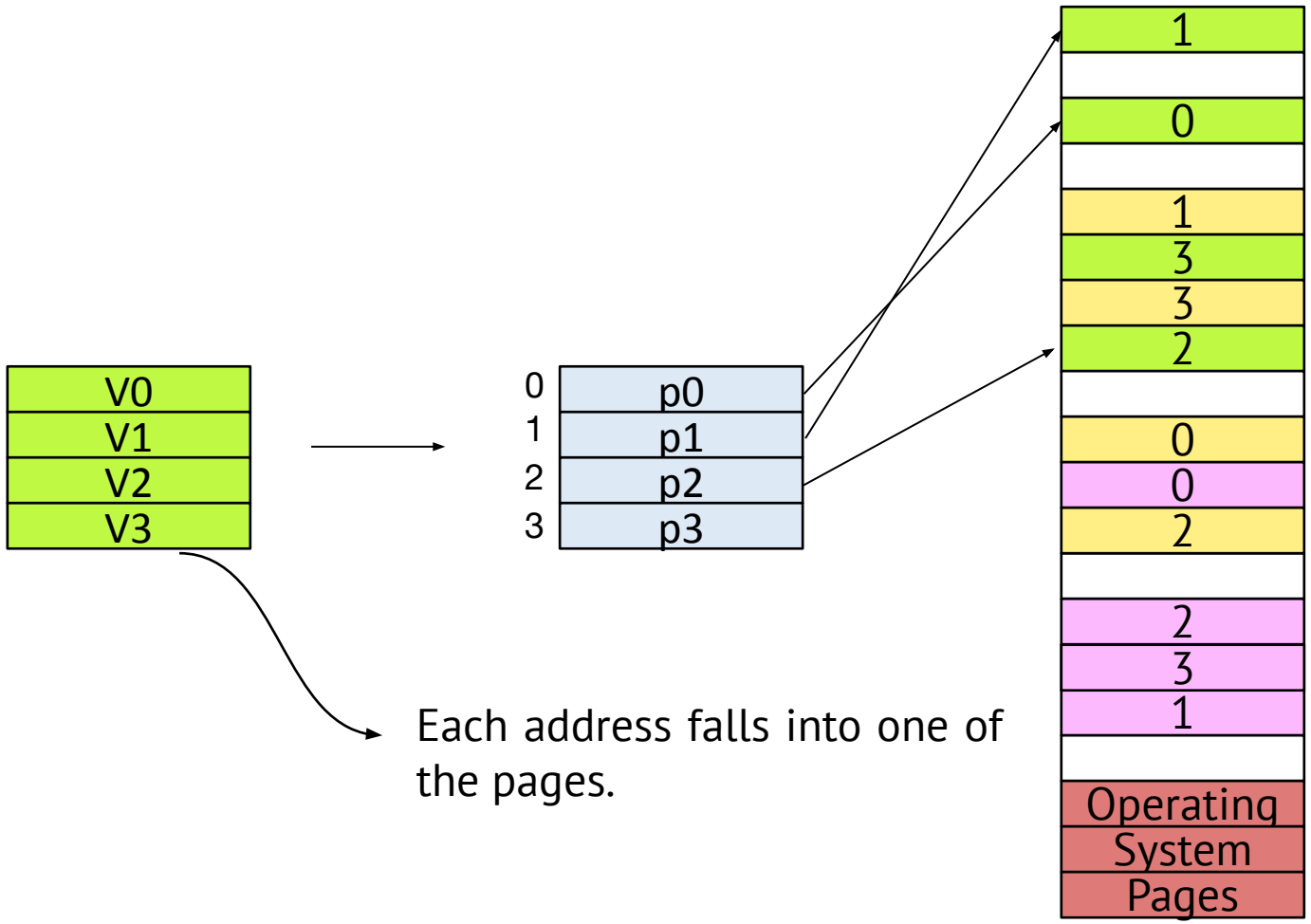




Virtual Address

- Since we are using page tables for translations, we no longer need to use real physical memories addresses in each program.
  - Each program can start from (virtual) address 0x00.
  - Each program sees a *large*, *private*, and *uniform* memory.
    (it's just an illusion though!)

Virtual to Physical Address Translation

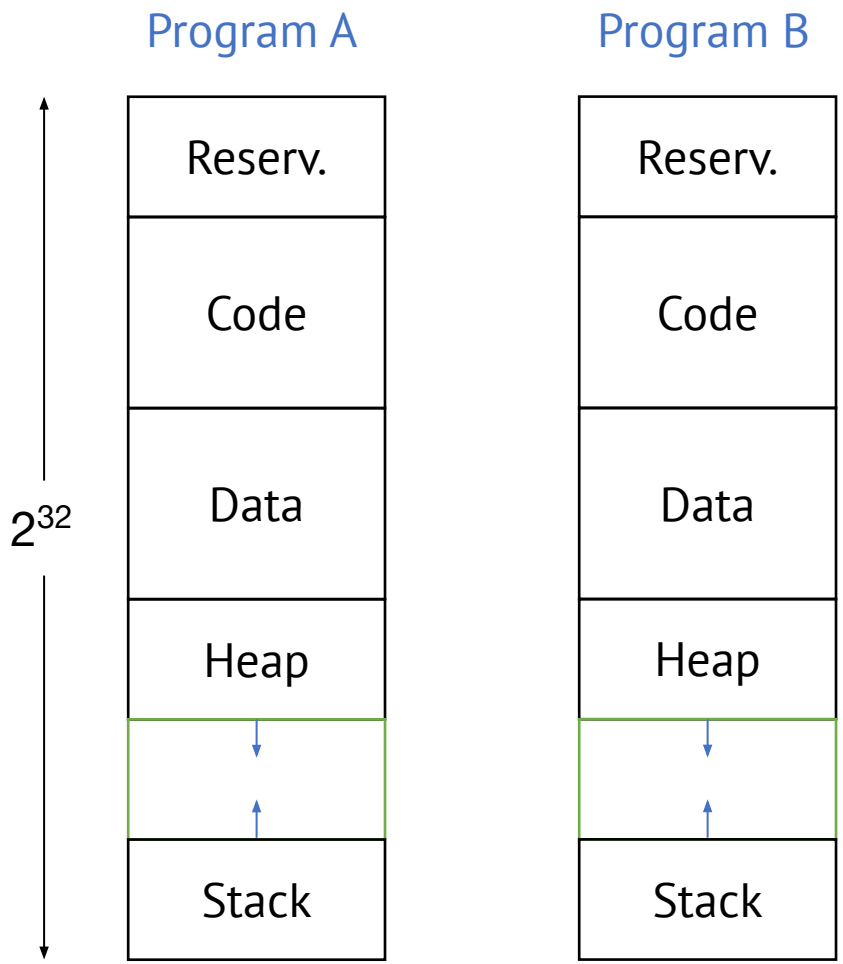


Address Lifecycle

- Program → VA → PA → Memory

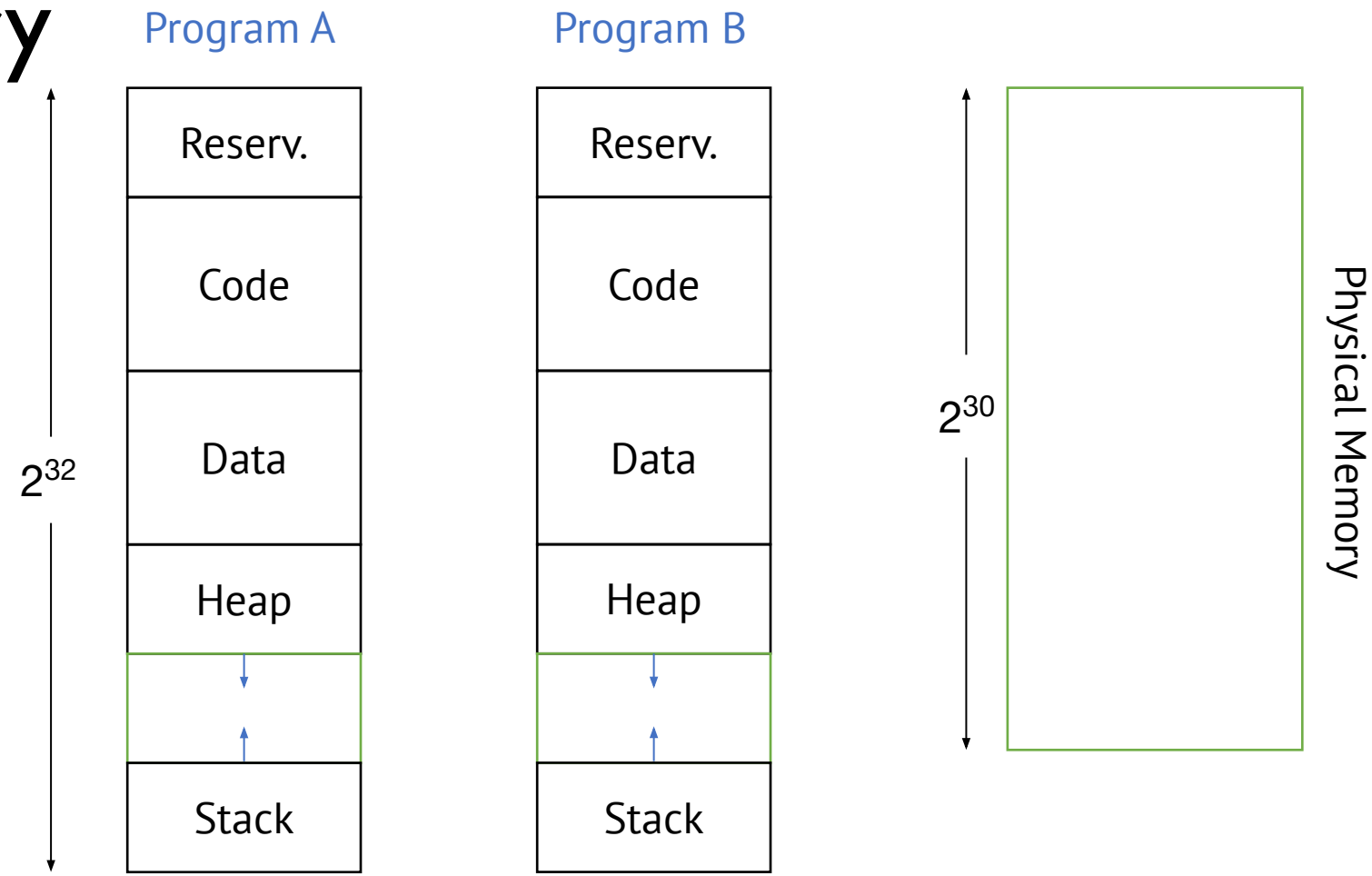
Virtual Memory

Private, large, and unified!

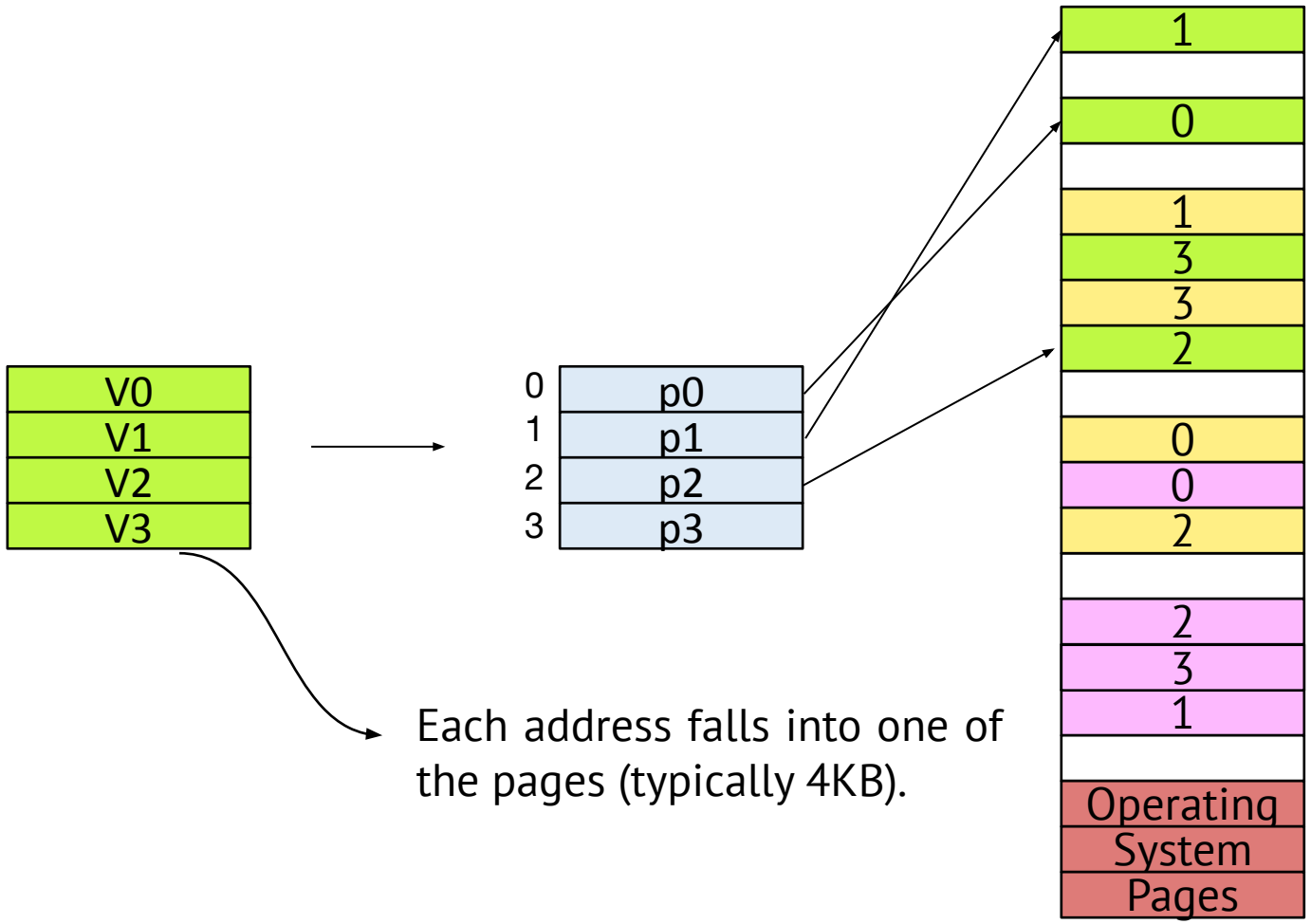


Physical Memory

What we actually have!



Virtual to Physical Address Translation

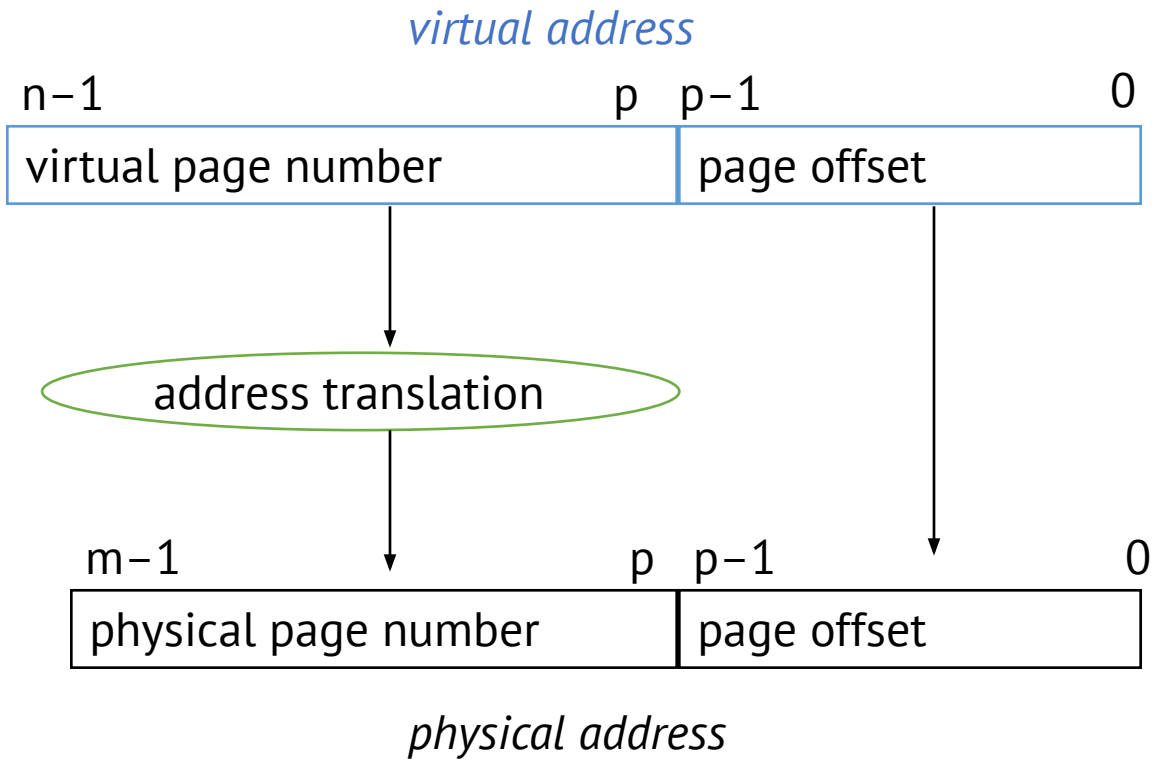


Virtual to Physical Address Translation

2^n: virtual address size

2^m: physical address size

2^p: page size



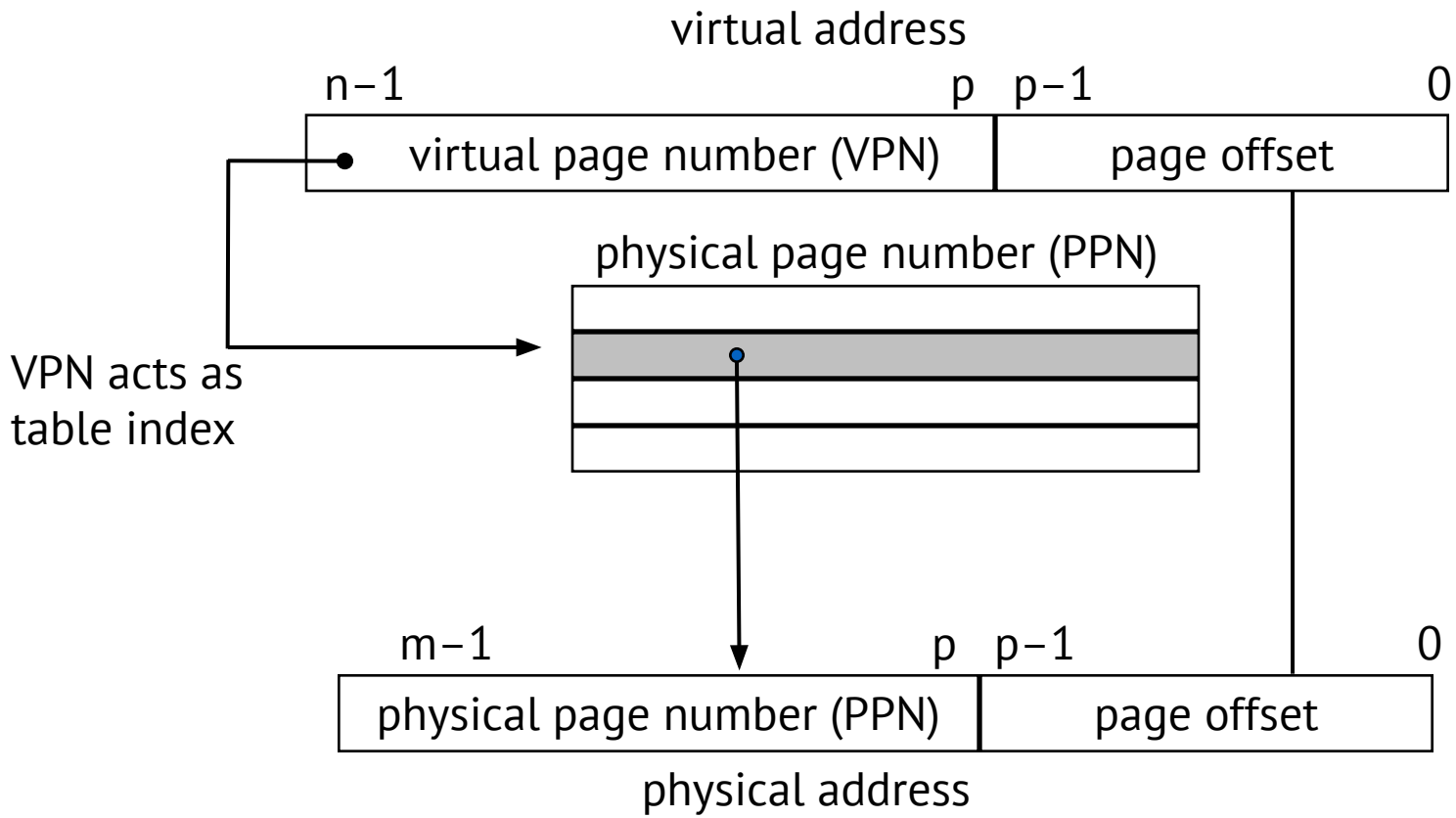
Nader Sehatbakhsh <nsehat@ee.ucla.edu>

Virtual and Physical Memory

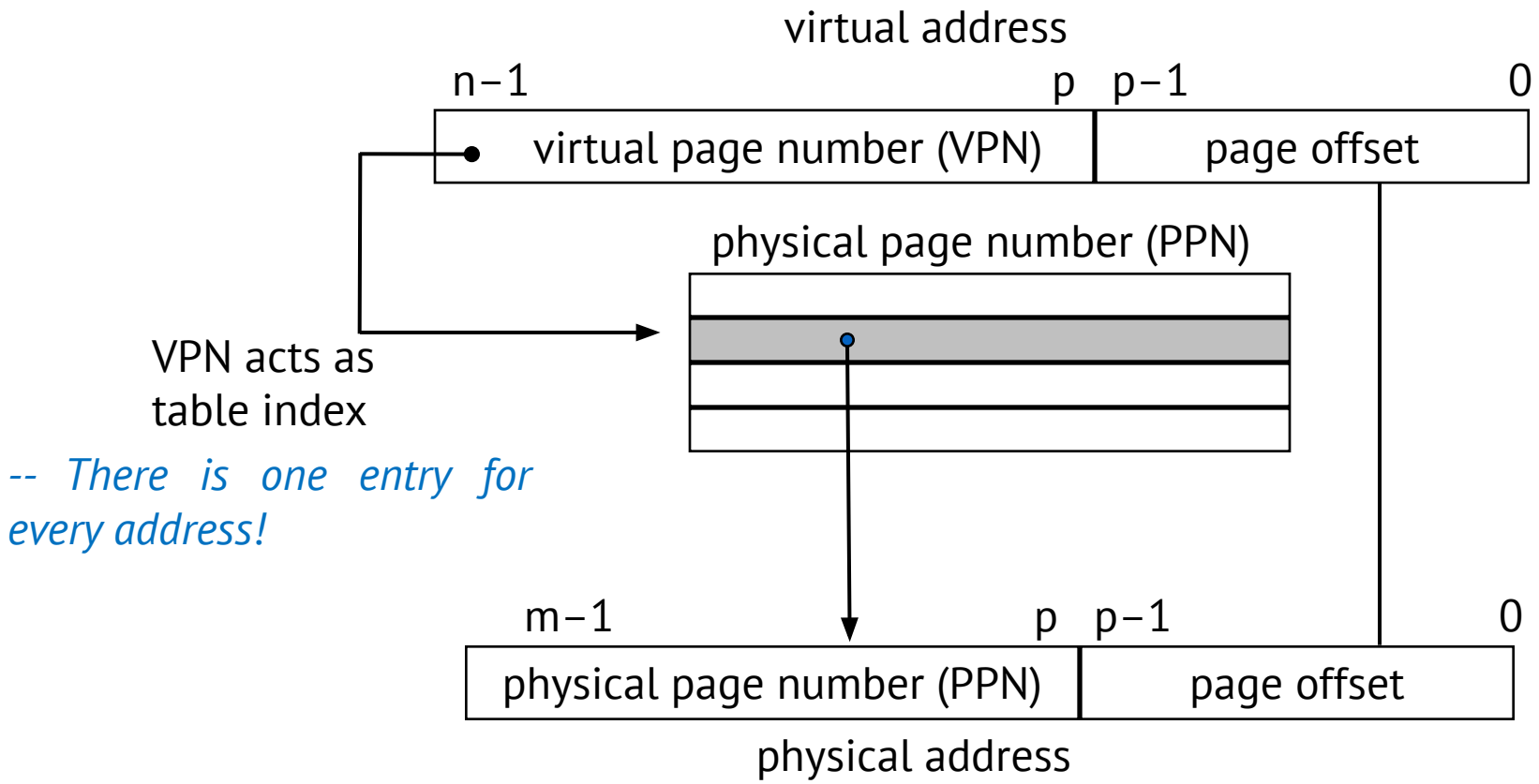
- System:
  - Virtual memory size: 4 GB = 2^{32} bytes
  - Physical memory size: 256 MB = 2^{28} bytes
  - Page size: 4 KB = 2^{12} bytes

- Organization:
  - Virtual address: **32** bits
  - Physical address: **28** bits
  - Page offset: **12** bits
  - #Virtual pages = $2^{32}/2^{12}$ = $\mathbf{2^{20}}$ (VPN = 20 bits)
  - # Physical pages = $2^{28}/2^{12}$ = $\mathbf{2^{16}}$ (PPN = 16 bits)

Page Table


virtual address
n–1
p
p–1
0
virtual page number (VPN)
page offset
physical page number (PPN)
VPN acts as
table index
m–1
p
p–1
0
physical page number (PPN)
page offset
physical address

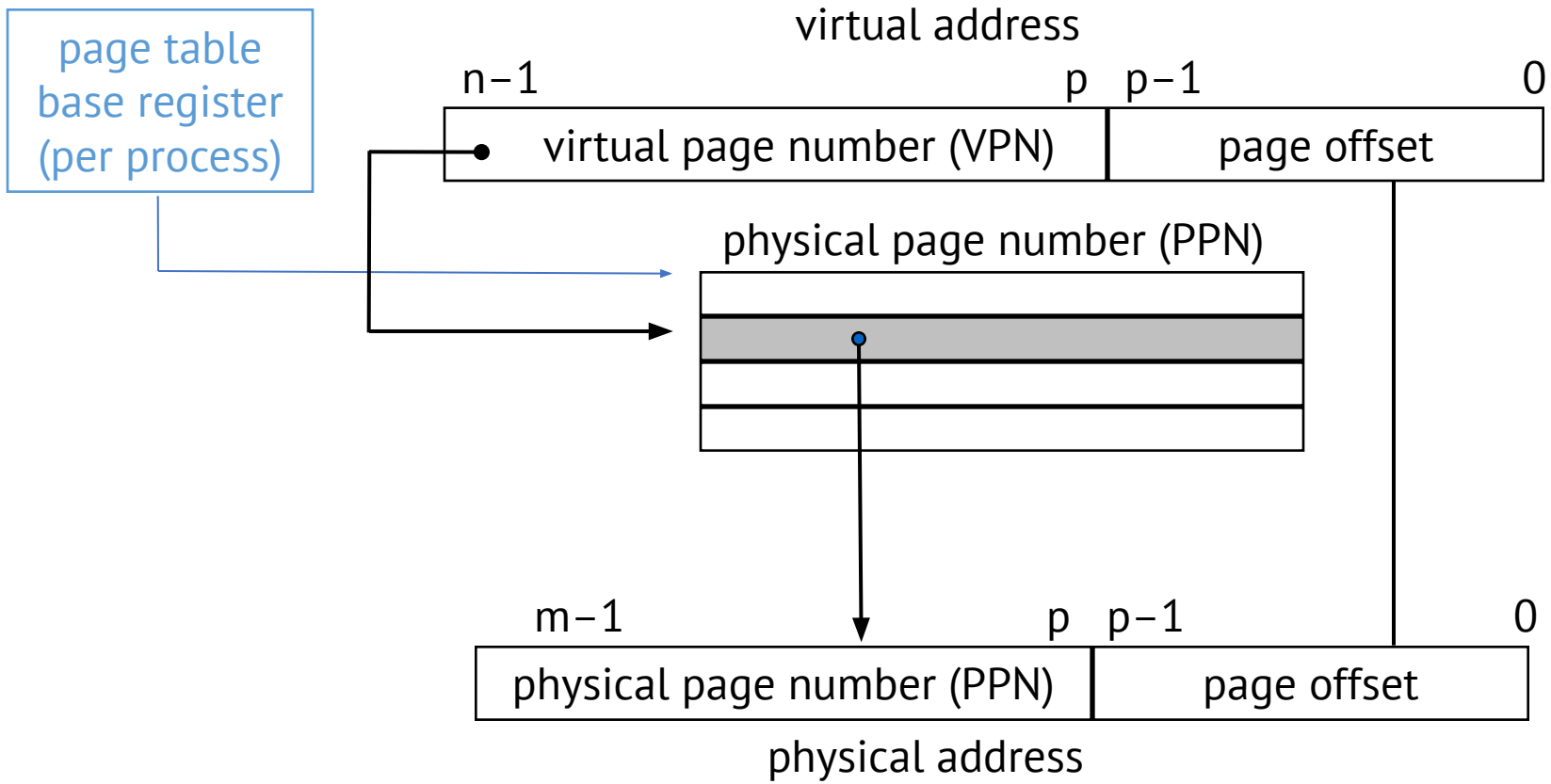
Page Table



-- There is one entry for every address!

Nader Sehatbakhsh <nsehat@ee.ucla.edu>

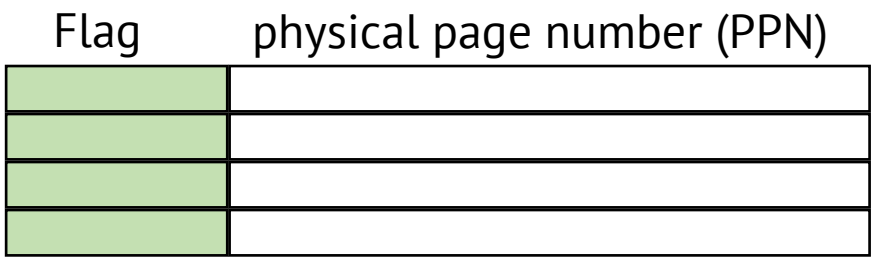
Page Table



Nader Sehatbakhsh <nsehat@ee.ucla.edu>

What else to store in each page table entry (PTE)?

- Protections and flags
  - Whether we can write to this page or not (read-only)?
  - Who is the owner? Is it shared?
  - Can we execute it?



Where to store page tables?

- Inside the memory itself.

Where to store page tables?

- Inside the memory itself.

★ Each memory access becomes *two accesses*, one for accessing the page table (i.e., adr = VPN + base). The other for accessing the translated address (i.e., adr = PPN + page offset).

How large is a page table?

- #Virtual pages = $2^{32}/2^{12} = 2^{20}$ (VPN = 20 bits)
- Physical address: 28
- Each entry in page table (PTE) = 28 - offset + flags ≃ 32 bits (4 bytes)

→ 2^{20} x 32 = 4 MB for each application!!

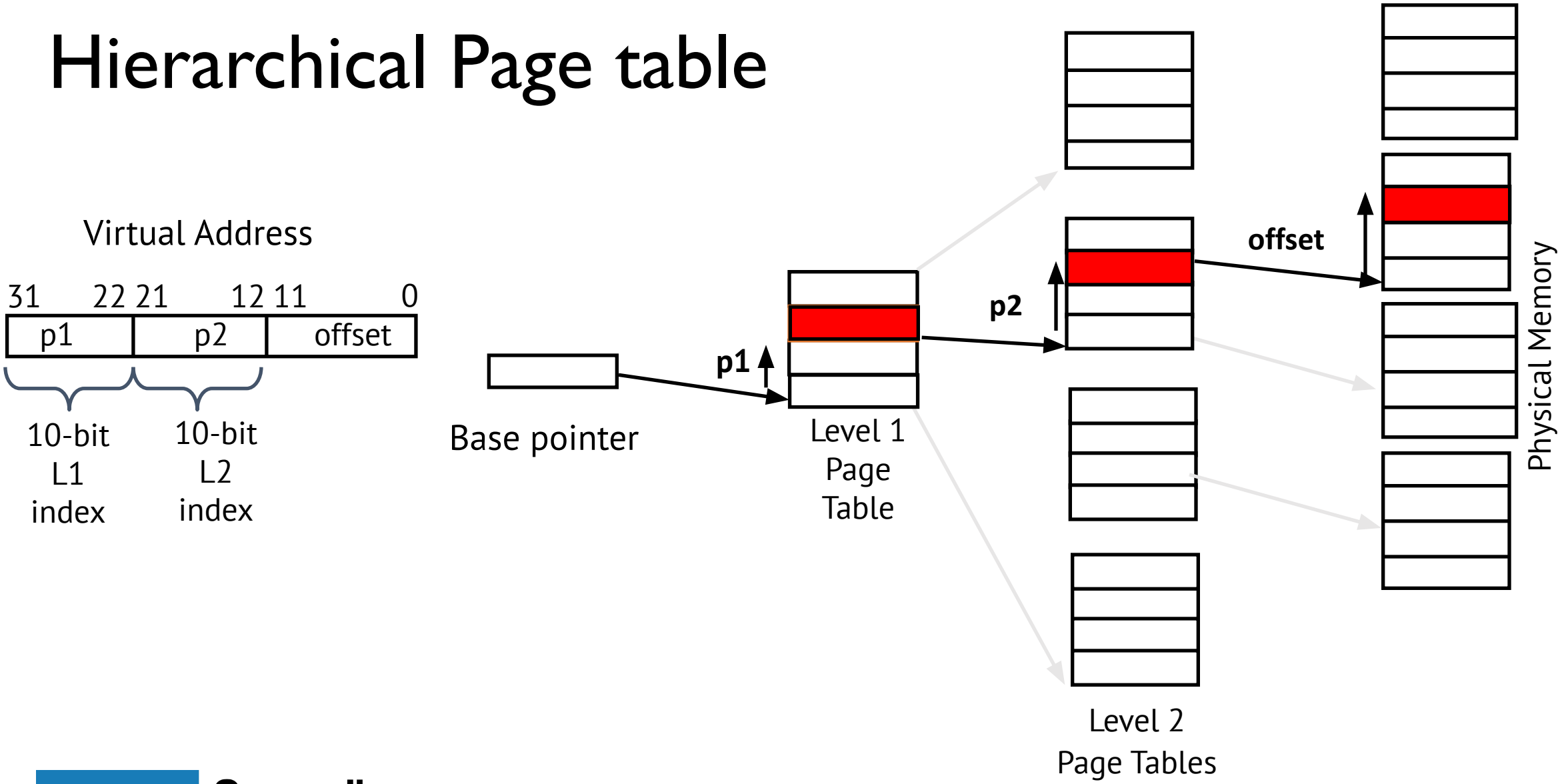
(What about 64-bit address?)

How to compress page tables?

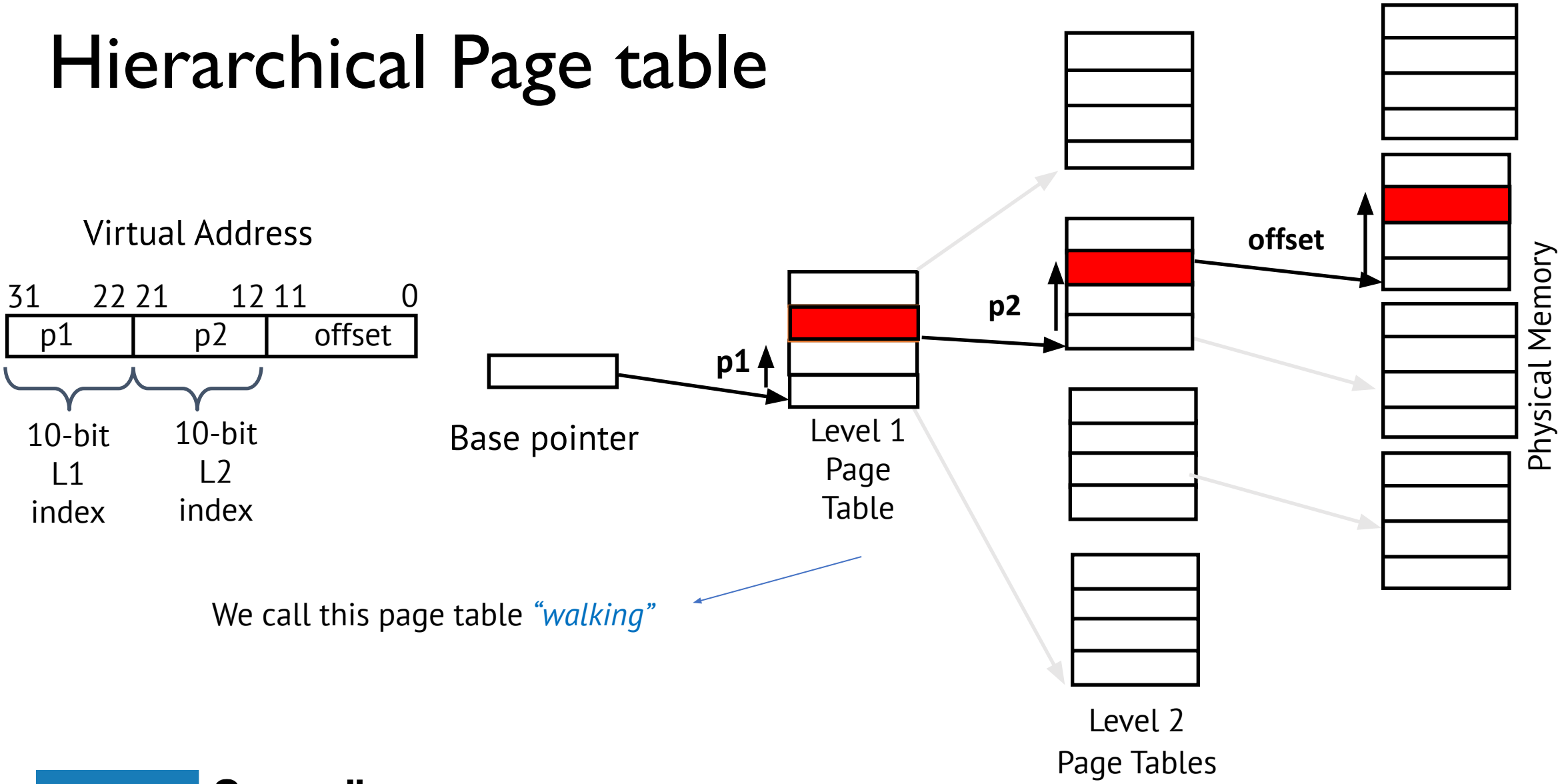
Hierarchical Page table

- Store the page table as a *page table*.

Hierarchical Page table

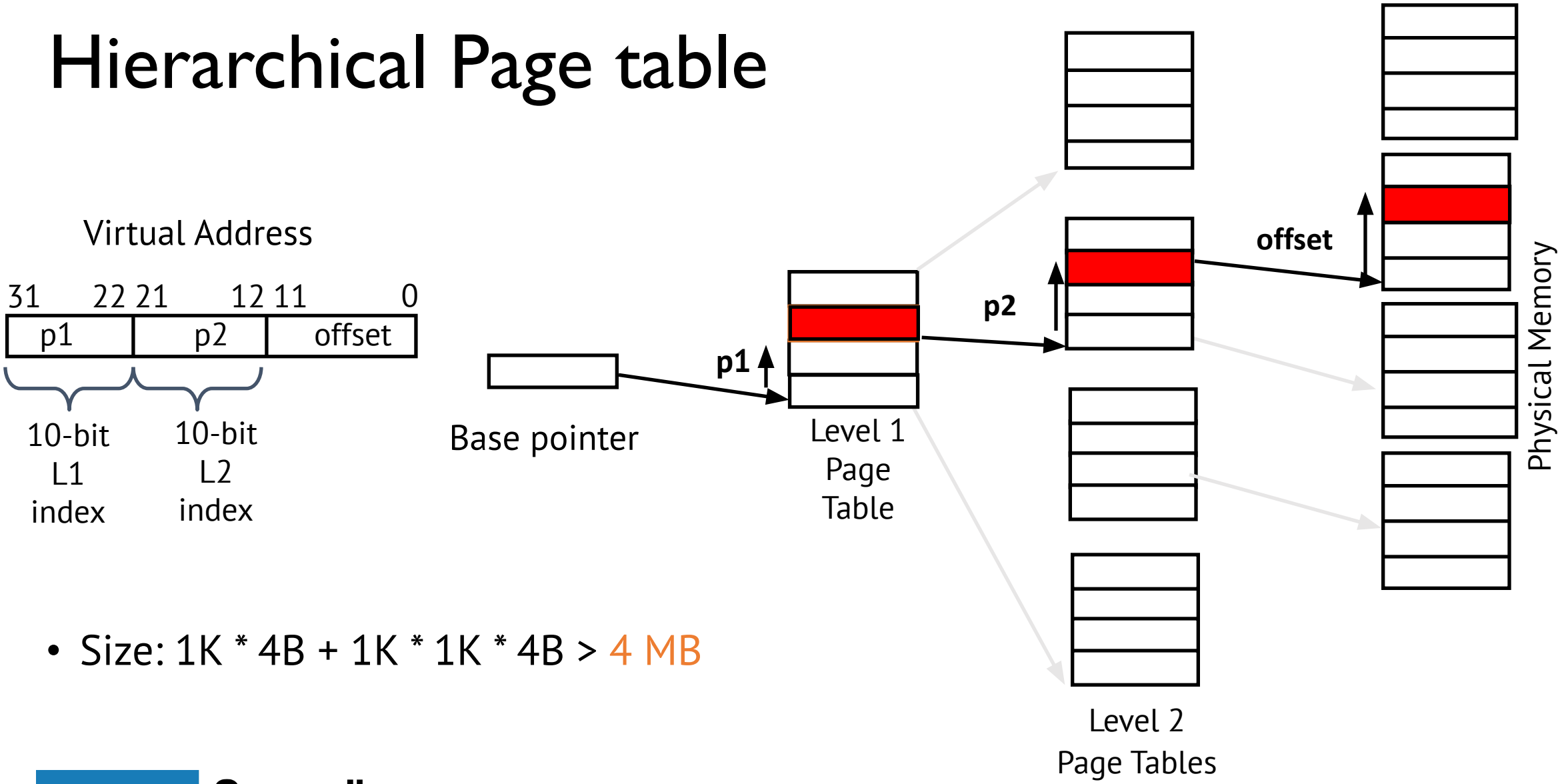


Hierarchical Page table



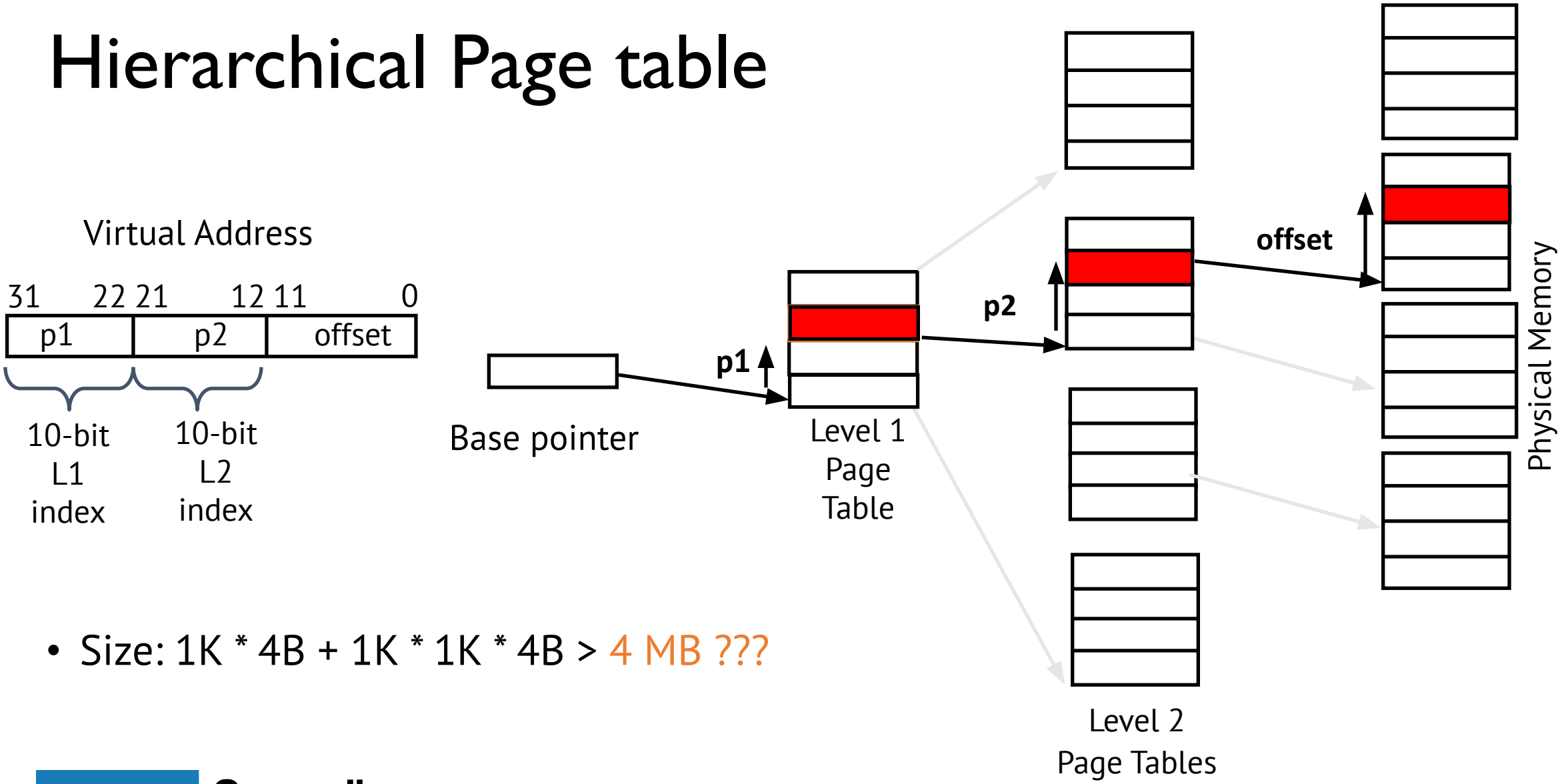
We call this page table *"walking"*

Hierarchical Page table



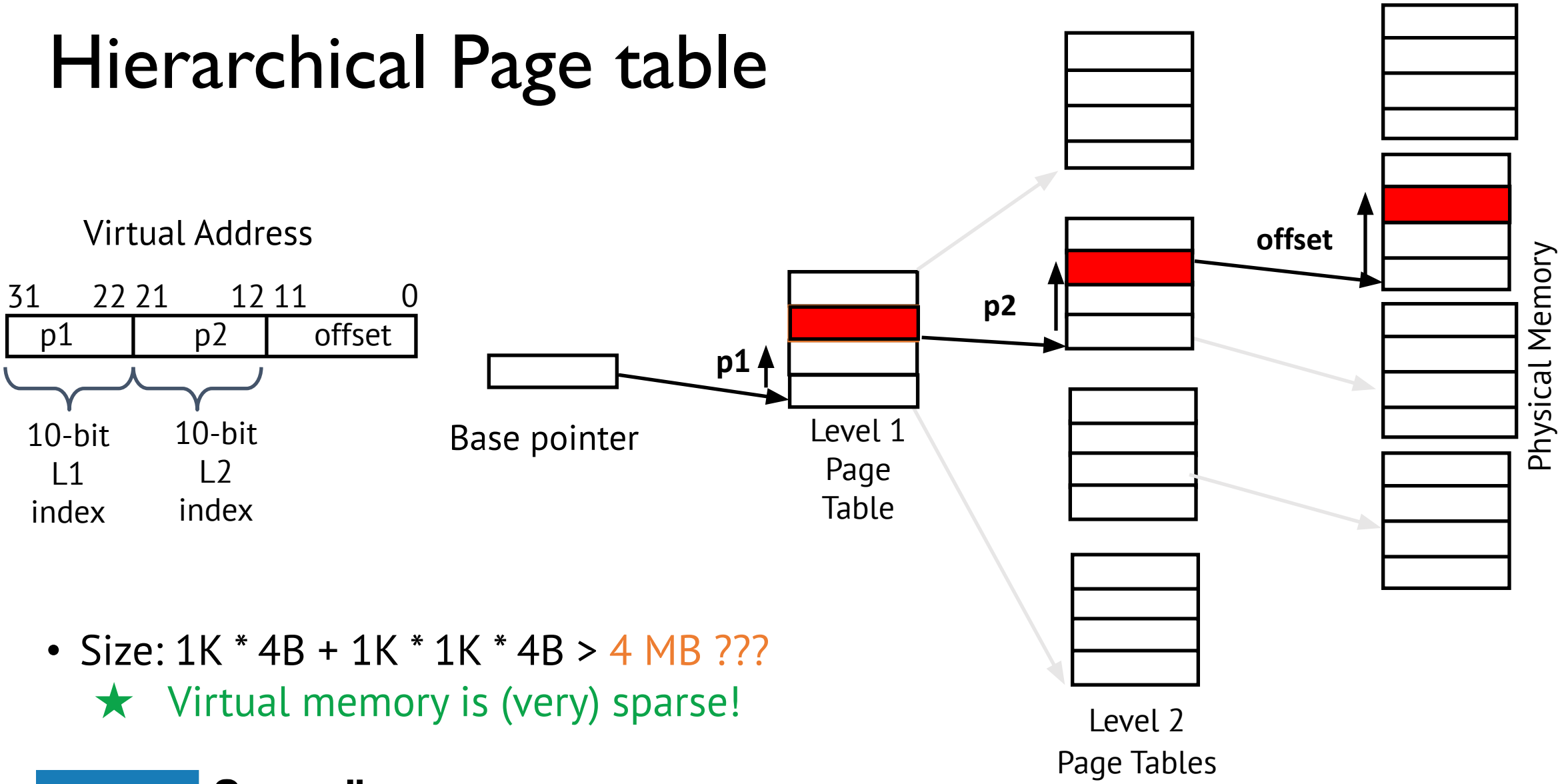
- Size: 1K * 4B + 1K * 1K * 4B > 4 MB

Hierarchical Page table



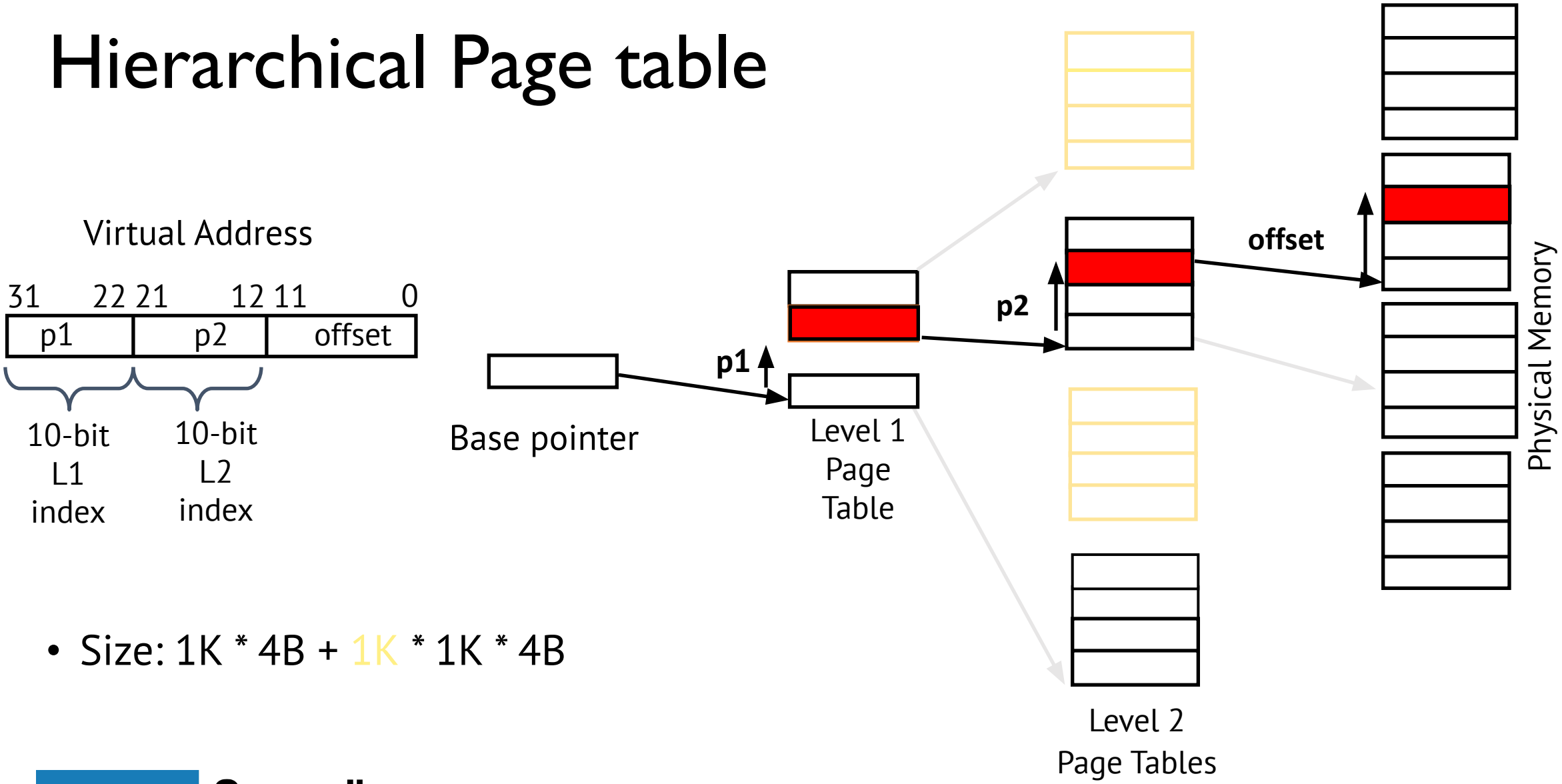
- Size: 1K * 4B + 1K * 1K * 4B > 4 MB ???

Hierarchical Page table



- Size: 1K * 4B + 1K * 1K * 4B > 4 MB ???
  - ★ Virtual memory is (very) sparse!

Hierarchical Page table



- Size: 1K * 4B + 1K * 1K * 4B

Hierarchical Page Tables

- We need page tables for address translation.
- Page tables are large and should be in the memory itself.
- Hierarchical Page tables can leverage the existing sparsity of virtual addresses and makes the page table storage compact!

Memory Access

- Each memory access (e.g., L1 cache) needs multiple (three or more) memory accesses!

➔ *What can we do?*

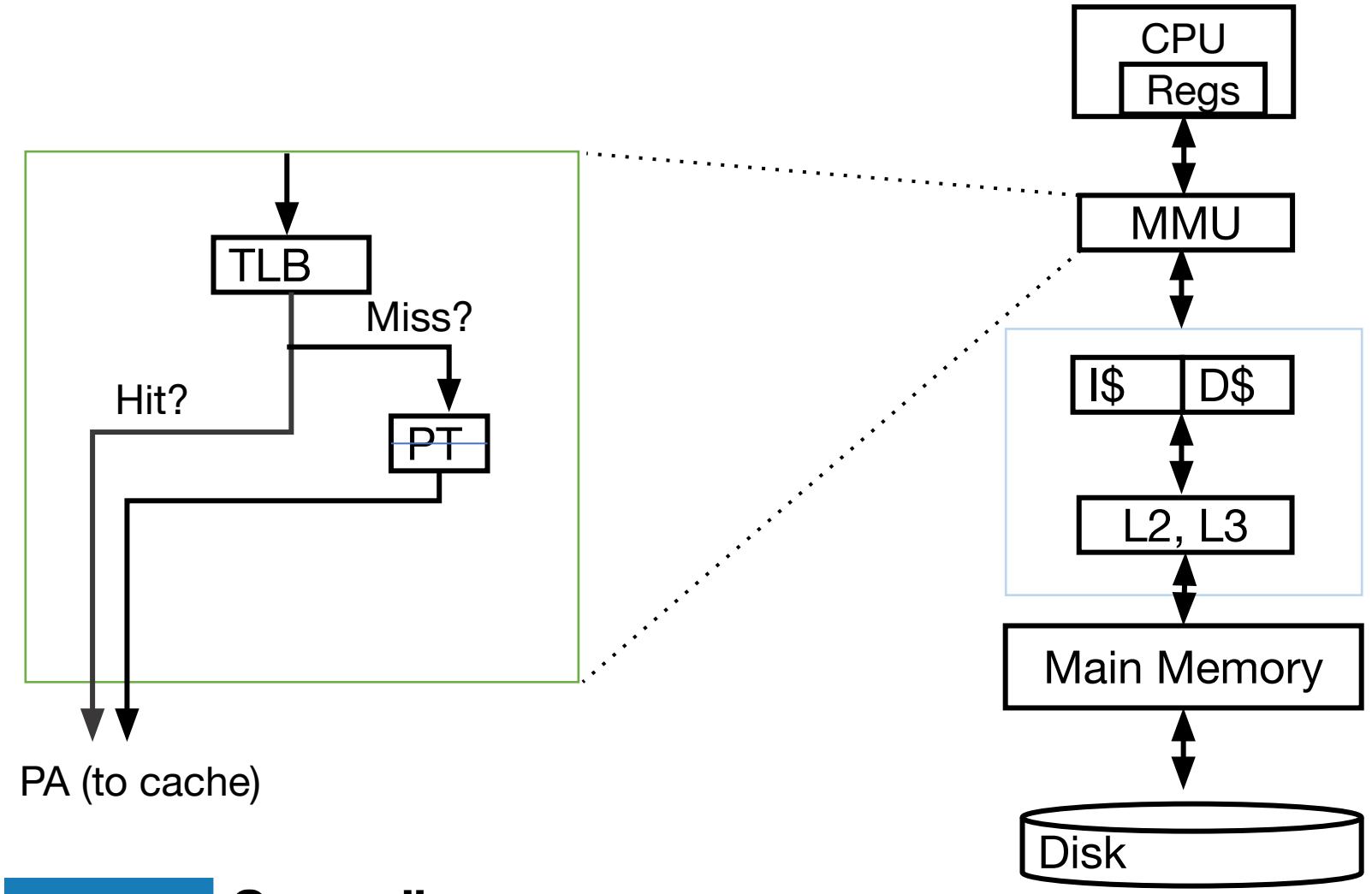
Memory Access

- Each memory access (e.g., L1 cache) needs multiple (three or more) memory accesses!

➔ *What can we do?*

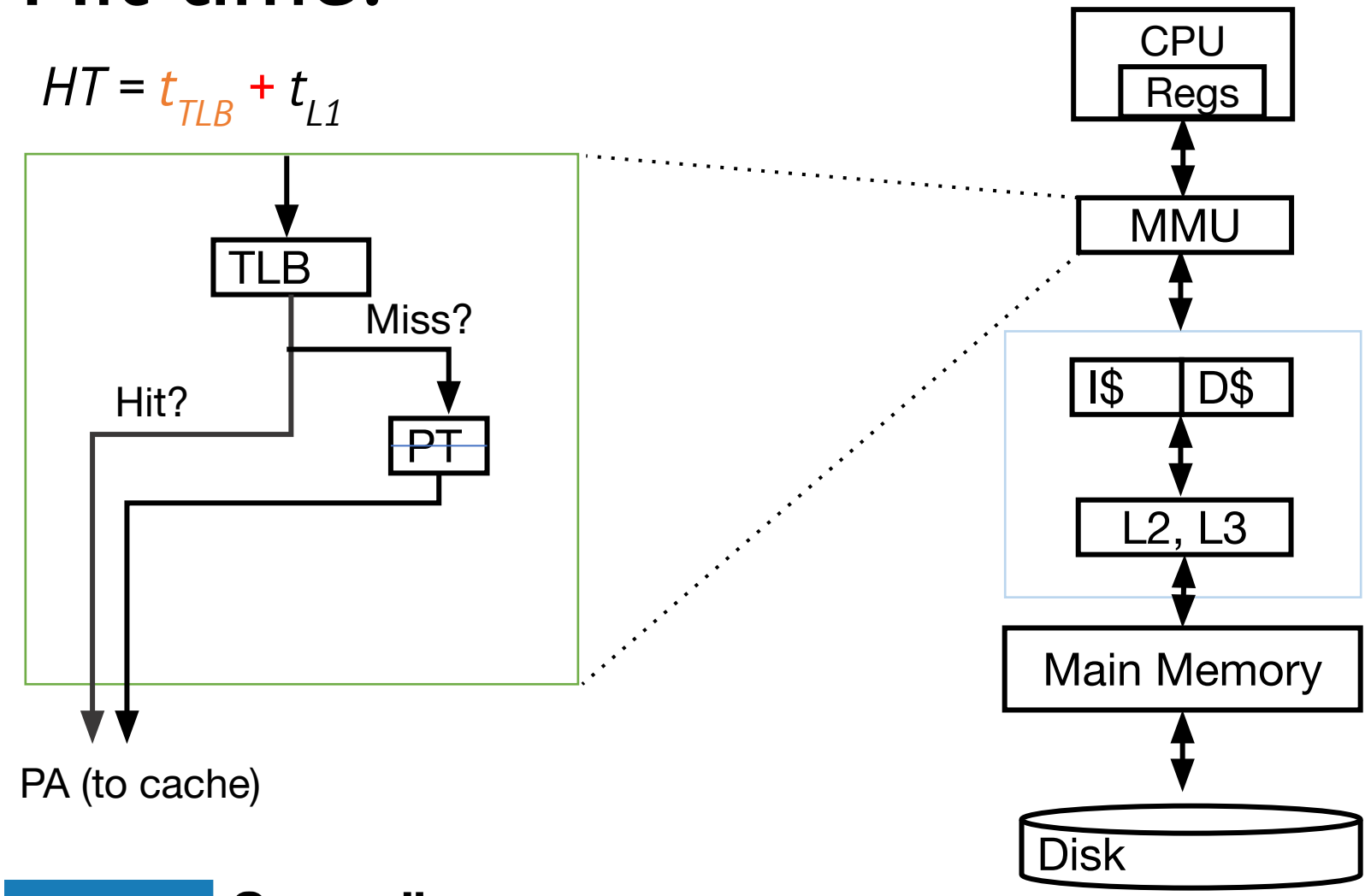
★ *Cache the most recent translations!*

Translation Lookaside Buffer (TLB)



Hit time?

$HT = t_{TLB} + t_{L1}$

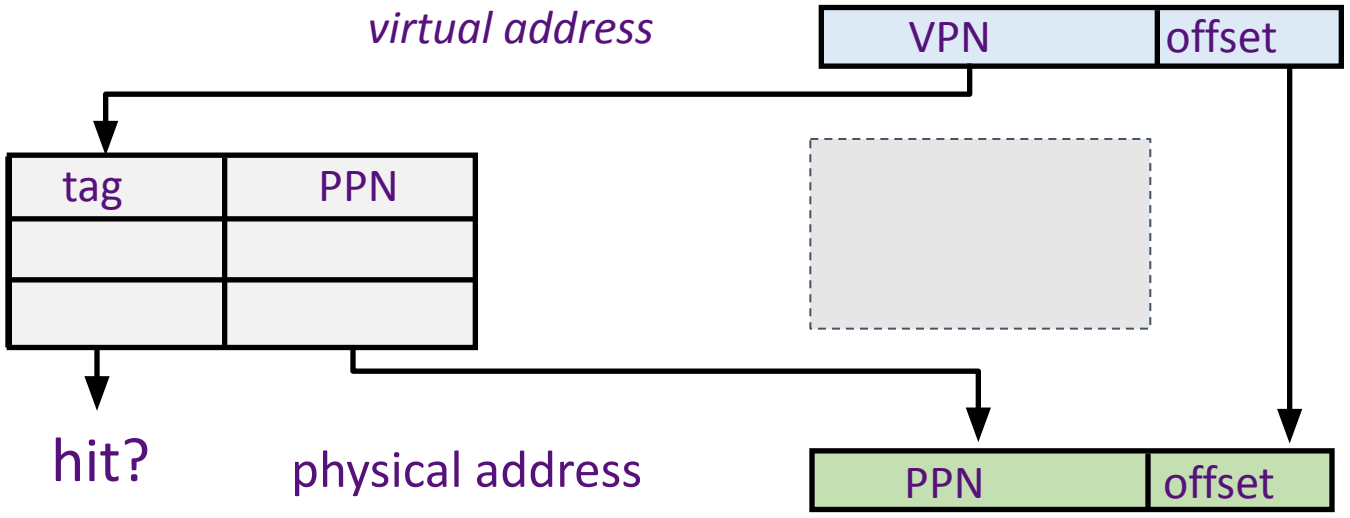


Translation Lookaside Buffer (TLB)

- *Cache translations in TLB*

  TLB hit ⇒ *Single-Cycle Translation*

  TLB miss ⇒ *Page-Table Walk to refill*



TLB Design

- Typically 32-128 entries, usually fully associative.
  - Each entry maps a large page, hence less spatial locality across pages.
  - Larger TLBs (256-512 entries) are 4 or 8 way set-associative.
  - Larger systems sometimes have multi-level (L1 and L2) TLBs.
  - Random or FIFO replacement policy.
  - ***TLB Reach:***
    - Size of largest virtual address space that can be simultaneously mapped by TLB
    - Example: 64 TLB entries, 4KB pages, one page per entry

      TLB Reach = 256 KB

- TLB miss causes an exception and results in a page table walk.

- OS typically is responsible to handle TLB miss (software handling).

- Alternatively, memory management unit (MMU) can handle TLB miss.

What about the Disk?

How to utilize disk?

- Disk is our secondary storage unit with much bigger size, but much larger access time.

How to utilize disk?

- Disk is our secondary storage unit with much bigger size, but much larger access time.

➔ *How to efficiently use the disk?*

Demand Paging

★ Use main memory and "swap" pages in the disk as automatically managed memory hierarchy levels.

→ Analogous to cache vs. main memory

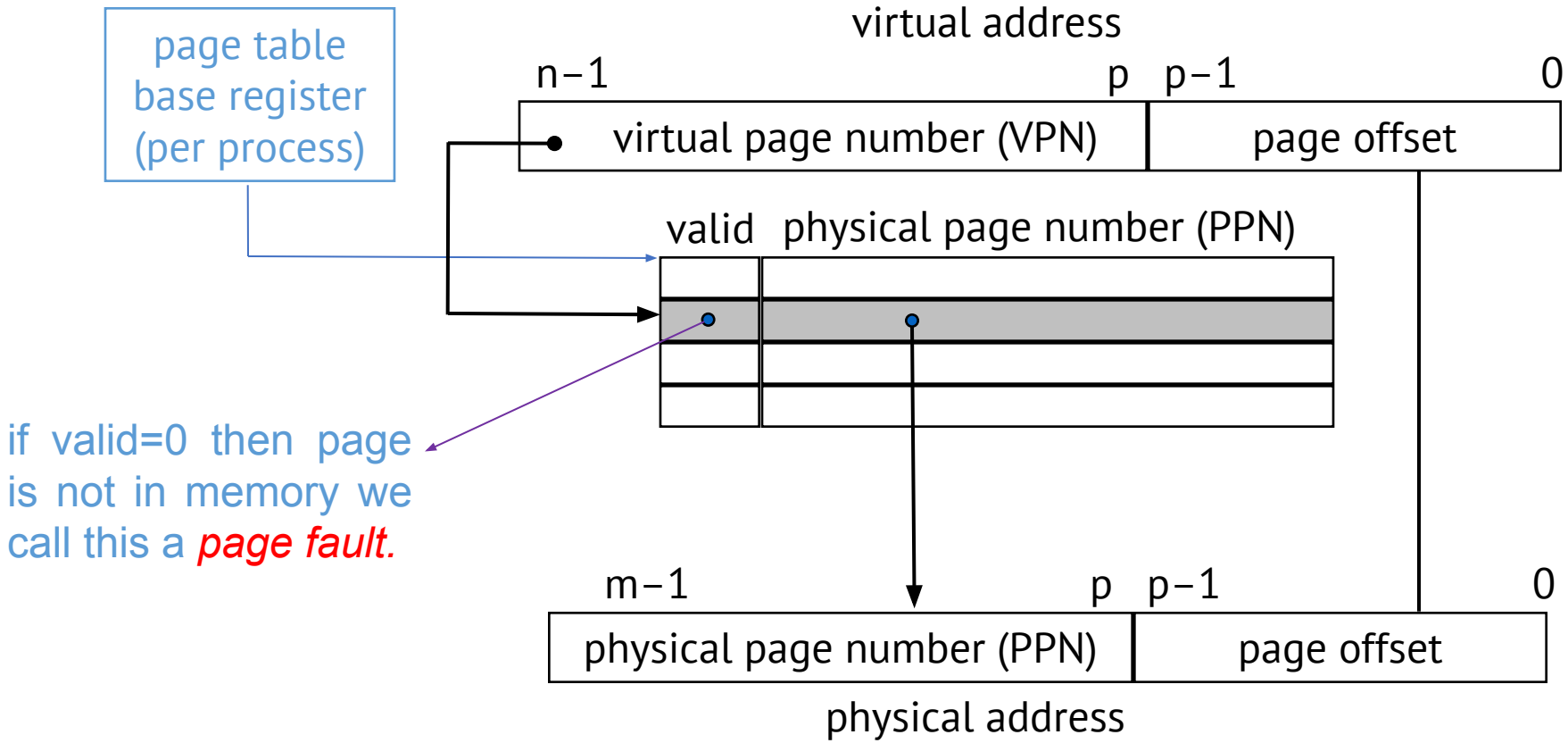
Demand Paging

★ Use main memory and "swap" pages in the disk as automatically managed memory hierarchy levels.

→ Analogous to cache vs. main memory

- M (DRAM + Disk capacity) bytes of storage,
  - keep most frequently used C bytes in DRAM (C << M)
  - Keep the rest in disk.
  - If the page is not in DRAM, we call it a *page fault*.
  - Bring a page (from disk to main memory) when "demanded".

Page Table


page table base register (per process)
virtual address
n–1
p
p–1
0
virtual page number (VPN)
page offset
valid
physical page number (PPN)
if valid=0 then page is not in memory we call this a page fault.
m–1
p
p–1
0
physical page number (PPN)
page offset
physical address

Demand Paging Design

- *Same basic issues as before (in cache)*
  - When to bring a page into DRAM?
  - Which page to evict (we call it "swap") from DRAM to disk to free-up DRAM for new pages?
  - Page Size?
  - …

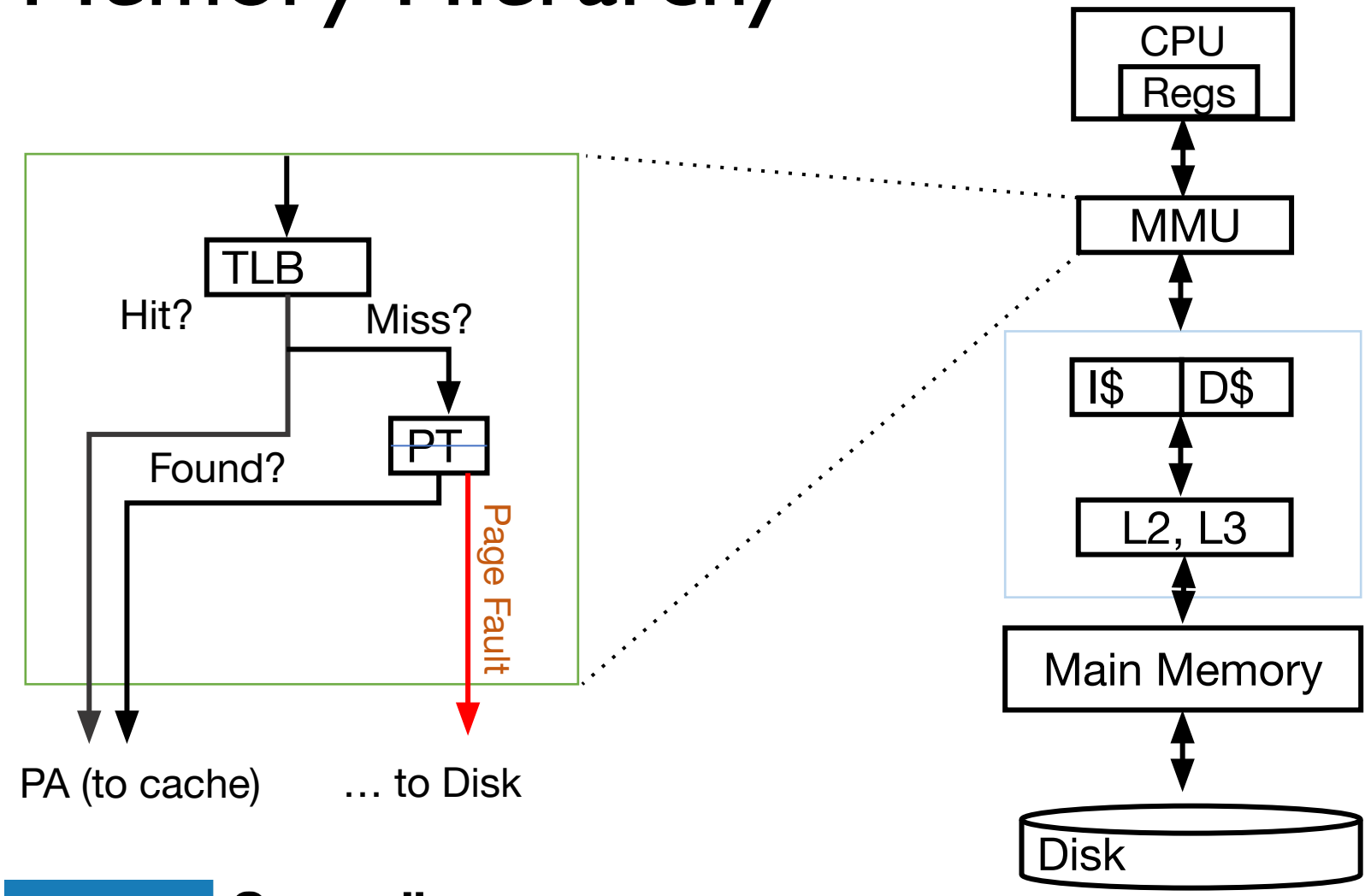
Demand Paging Design

- *Same basic issues as before (in cache)*

★ *OS handles everything (easier, fast enough)*

- Pseudo-LRU replacement policy

Memory Hierarchy



Recap

- Page tables
  - Why we need them and how to translate?
  - How to store them?
  - How to reduce access time?

Recap

- Page tables
  - Why we need them and how to translate?
  - How to store them?
  - How to reduce access time?

➔ Can we do better?

How to decrease hit time?

- Can we access L1 before TLB?

How to decrease hit time?

- Can we access L1 before TLB?
  - Address in L1 should be stored with VA (i.e., no translation)!

    Problem?

How to decrease hit time?

- Can we access L1 before TLB?
  - Address in L1 should be stored with VA (i.e., no translation)!

    Problem?

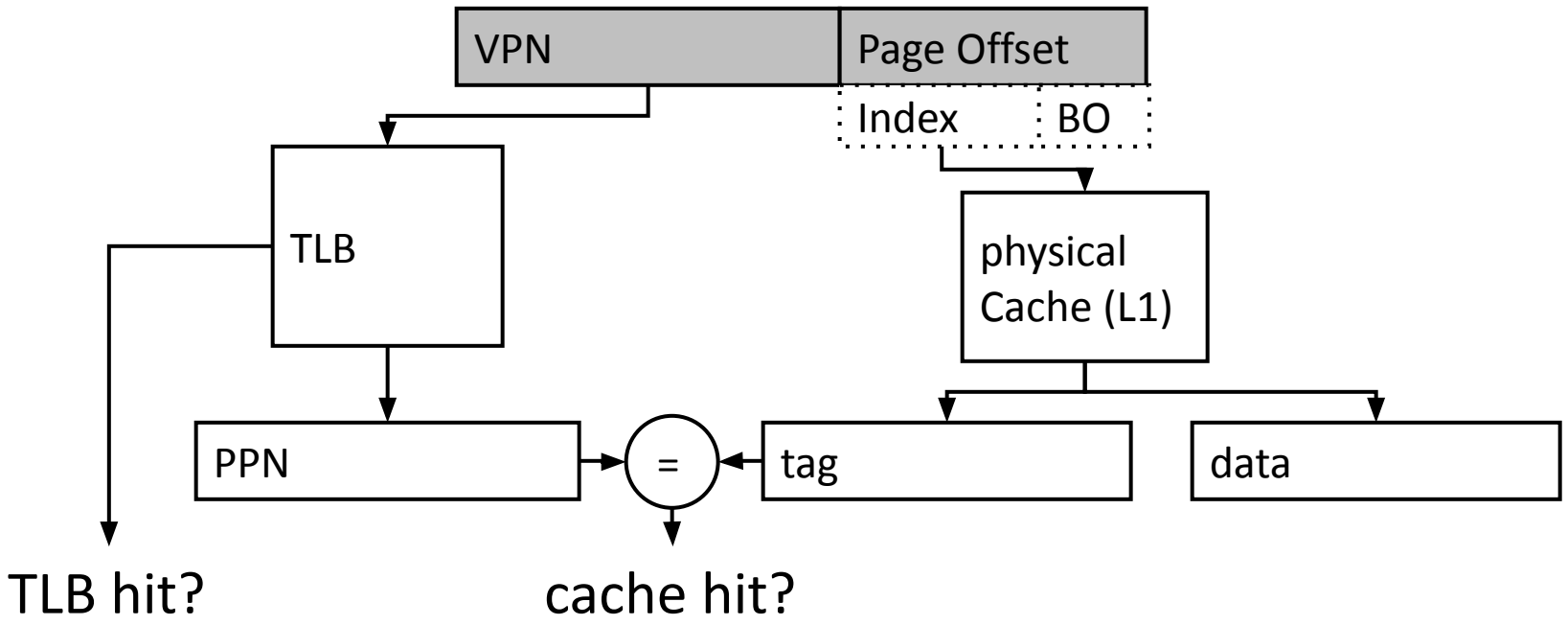
    - *Aliasing*

How to decrease hit time?

- Can we access L1 before TLB?
  - Address in L1 should be stored with VA!
    - NO!

- Can we access TLB and L1 *in parallel* ?

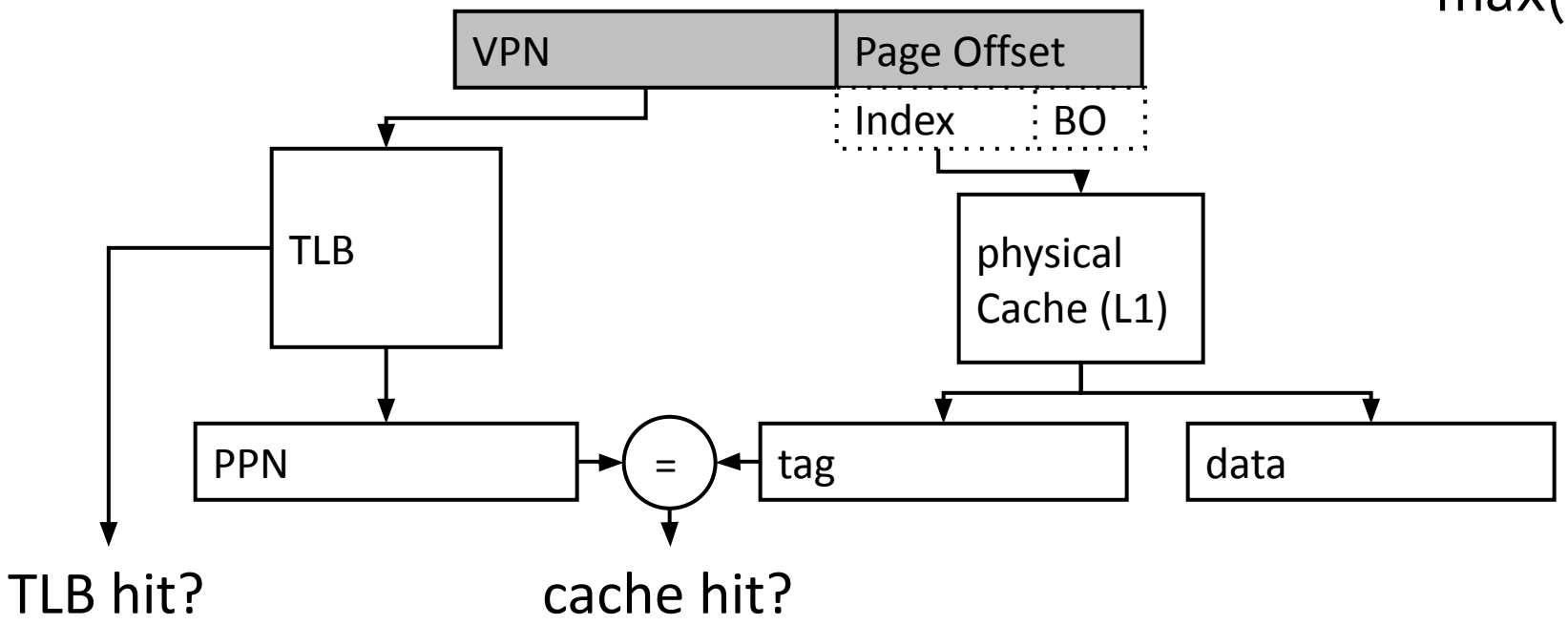
Virtually-Indexed *Physically*-Tagged Cache



Virtually-Indexed *Physically*-Tagged Cache

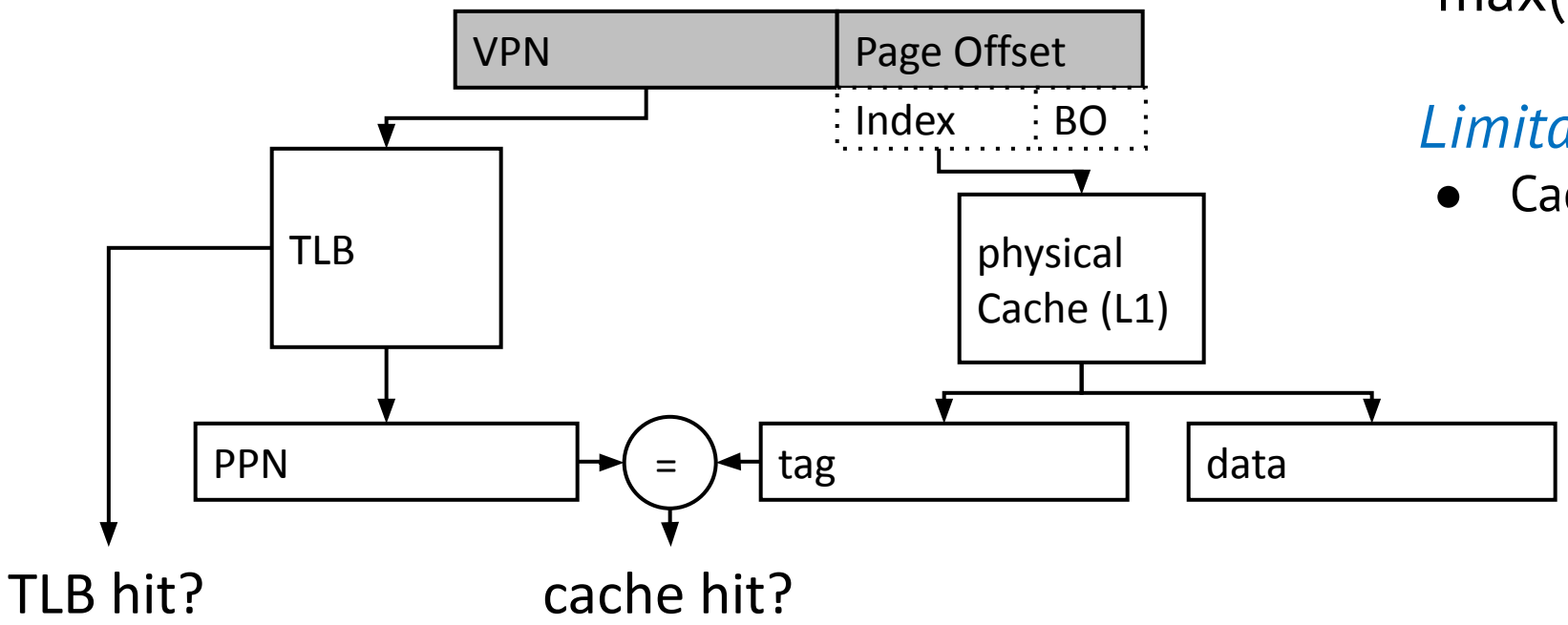
Hit time?

$= \max(t_{TLB}, t_{L1}) = t_{L1}$



Nader Sehatbakhsh <nsehat@ee.ucla.edu>

Virtually-Indexed *Physically*-Tagged Cache



Hit time?

$= \max(t_{TLB}, t_{L1}) = t_{L1}$

Limitation?

- Cache size ⩽ Assoc. x Page Size

With and Without MMU

- Most embedded processors and DSPs provide *physical addressing* only!
  - Can't afford area/speed/power budget for virtual memory support.
  - Often there is no secondary storage to swap to!

End of Presentation

Acknowledgement

- This course is partly inspired by the following courses created by my colleagues:
  - CS152, Krste Asanovic (UCB)
  - 18-447, James C. Hoe (CMU)
  - CSE141, Steven Swanson (UCSD)
  - CIS 501, Joe Devietti (Upenn)
  - CS4290, Tom Conte (Georgia Tech)
  - 252-0028-00L, Onur Mutlu (ETH)