Agenda

- Virtual memory

Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

Abstraction: Virtual vs. Physical Memory

- Programmer sees virtual memory
  - Can assume the memory is "infinite"
- Reality: Physical memory size is much smaller than what the programmer assumes

- The system (system software + hardware, cooperatively) maps virtual memory addresses to physical memory
  - The system automatically manages the physical memory space transparently to the programmer

+ Programmer does not need to know the physical size of memory nor manage it → A small physical memory can appear as a huge one to the programmer → Life is easier for the programmer

-- More complex system software and architecture

Benefits of Automatic Management of Memory

- Programmer does not deal with physical addresses
- Each process has its own mapping from virtual → physical addresses

- Enables
  - Code and data to be located anywhere in physical memory
    (relocation)
  - Isolation/separation of code and data of different processes in physical processes
    (protection and isolation)
  - Code and data sharing between multiple processes
    (sharing)

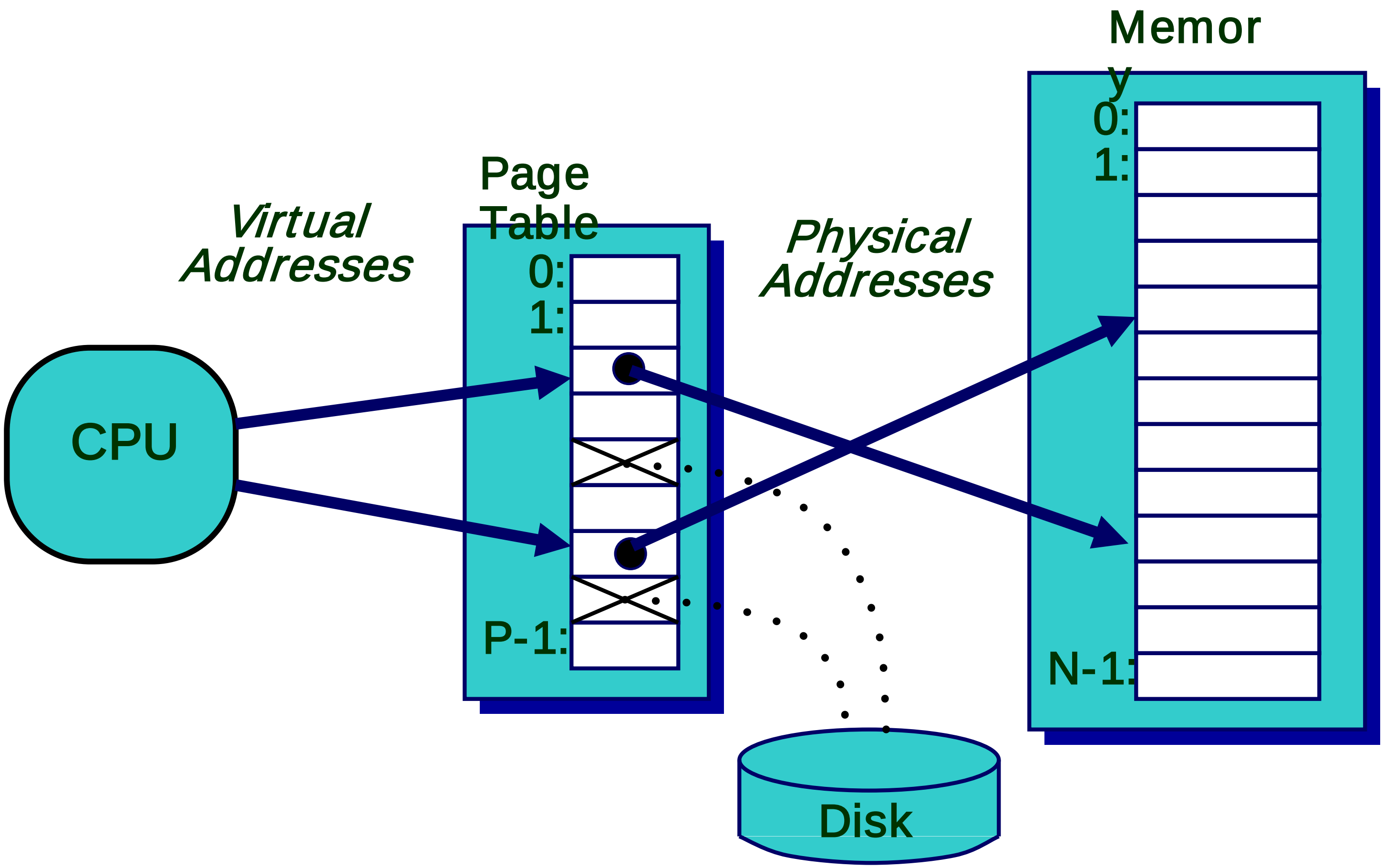
Basic Mechanism

- Indirection (in addressing)

- Address generated by each instruction in a program is a "virtual address"
  - i.e., it is not the physical address used to address main memory

- An "address translation" mechanism maps this address to a "physical address"
  - Address translation mechanism can be implemented in hardware and software together

A System with Virtual Memory (Page based)



- Address Translation: The hardware converts virtual addresses into physical addresses via an OS-managed lookup table (page table)

Virtual Pages, Physical Pages

- Virtual and physical address space divided into pages

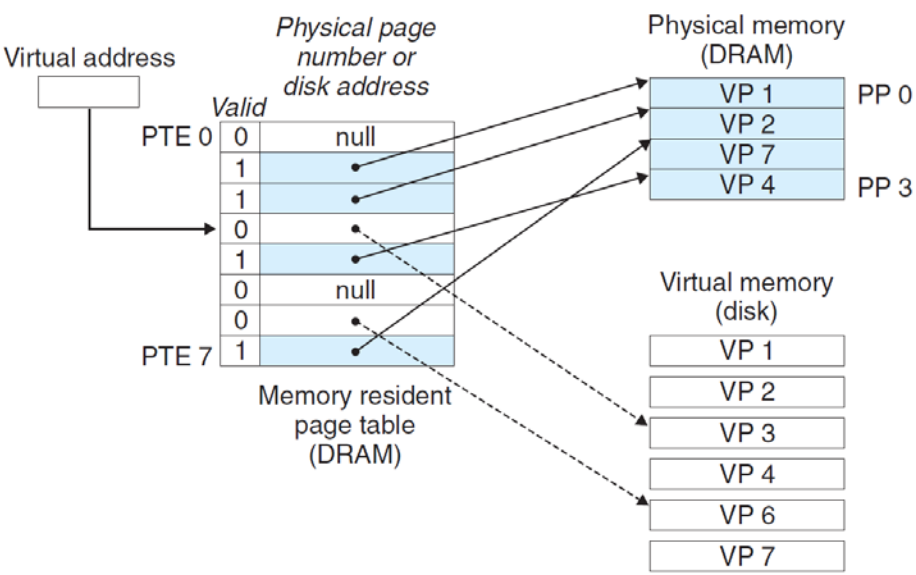
- A virtual page is mapped to
  - A physical page, if the page is in physical memory
  - A location in disk, otherwise

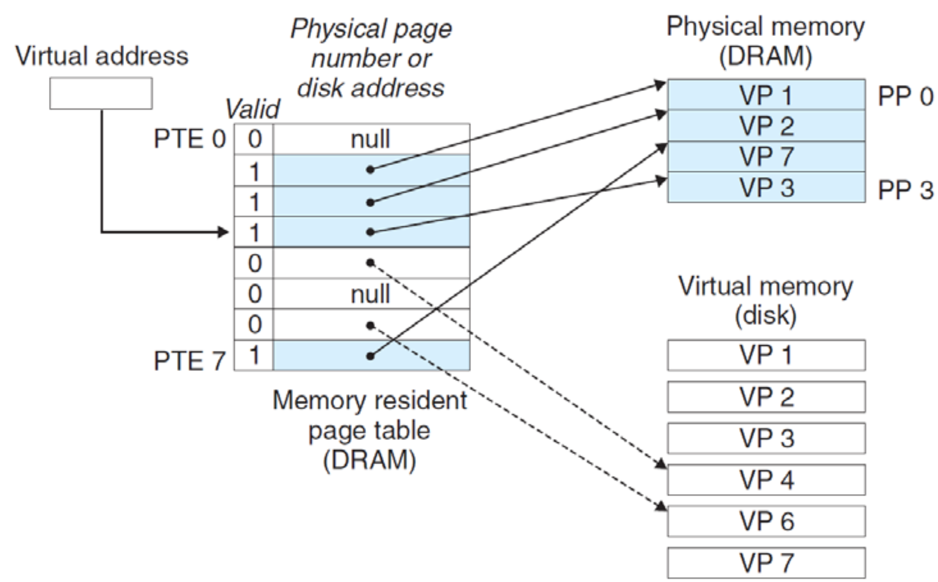
- If an accessed virtual page is not in memory, but on disk
  - Virtual memory system brings the page into a physical page and adjusts the mapping → this is called demand paging

- Page table is the table that stores the mapping of virtual pages to physical frames

Page Fault ("A Miss in Physical Memory")

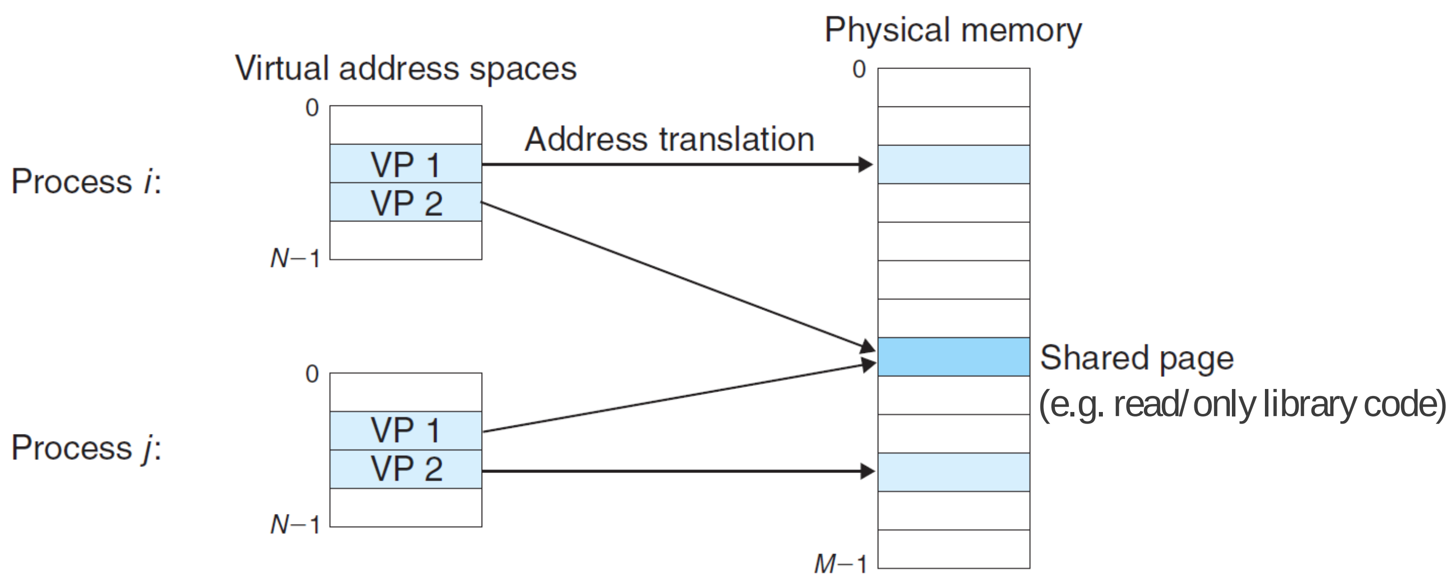
- If a page is not in physical memory but disk
  - Page table entry indicates virtual page not in memory
  - Access to such a page triggers a page fault exception
  - OS trap handler invoked to move data from disk into memory
    - Other processes can continue executing
    - OS has full control over placement





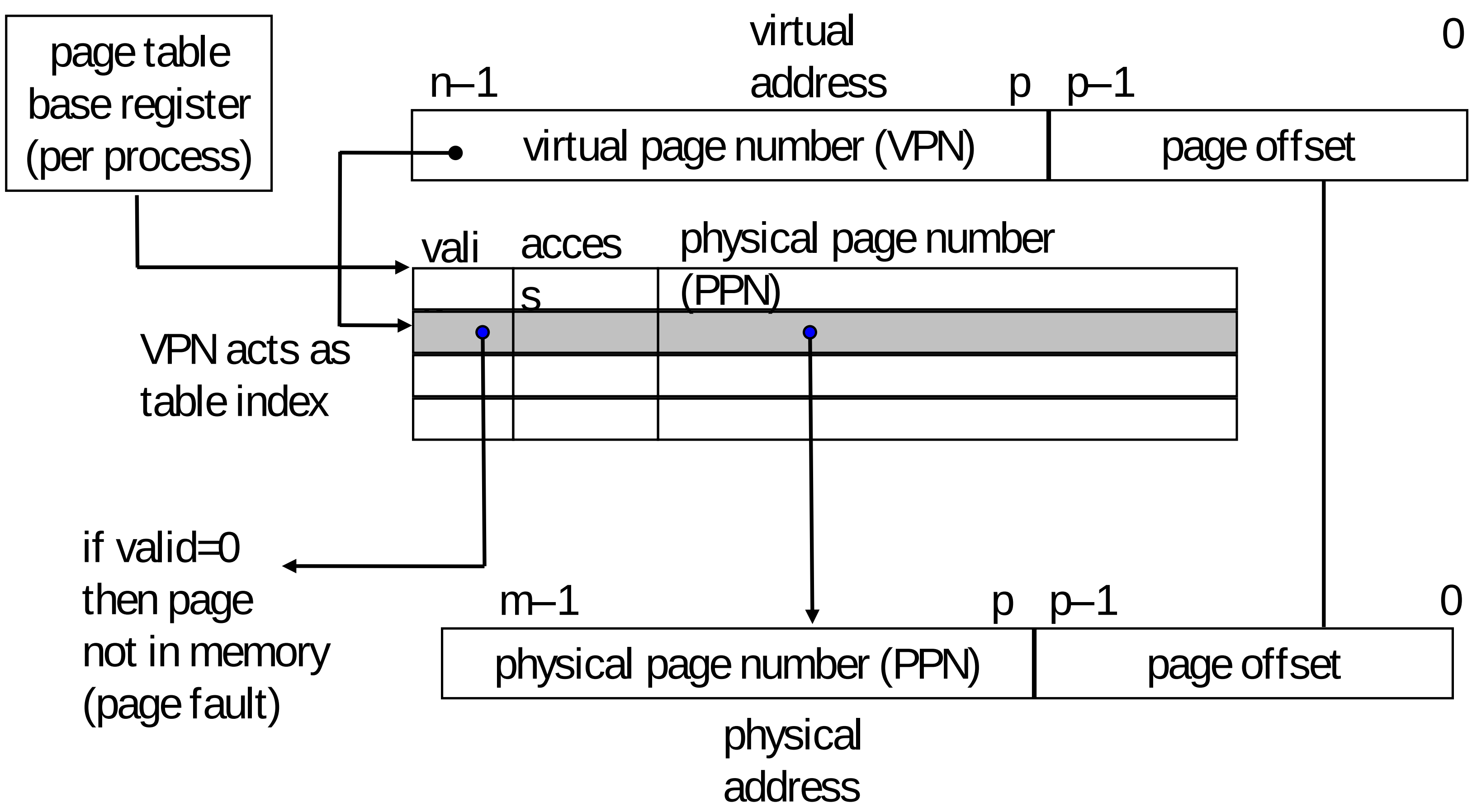
Page Table is Per Process

- Each process has its own virtual address space
  - Full address space for each program
  - Simplifies memory allocation, sharing, linking and loading



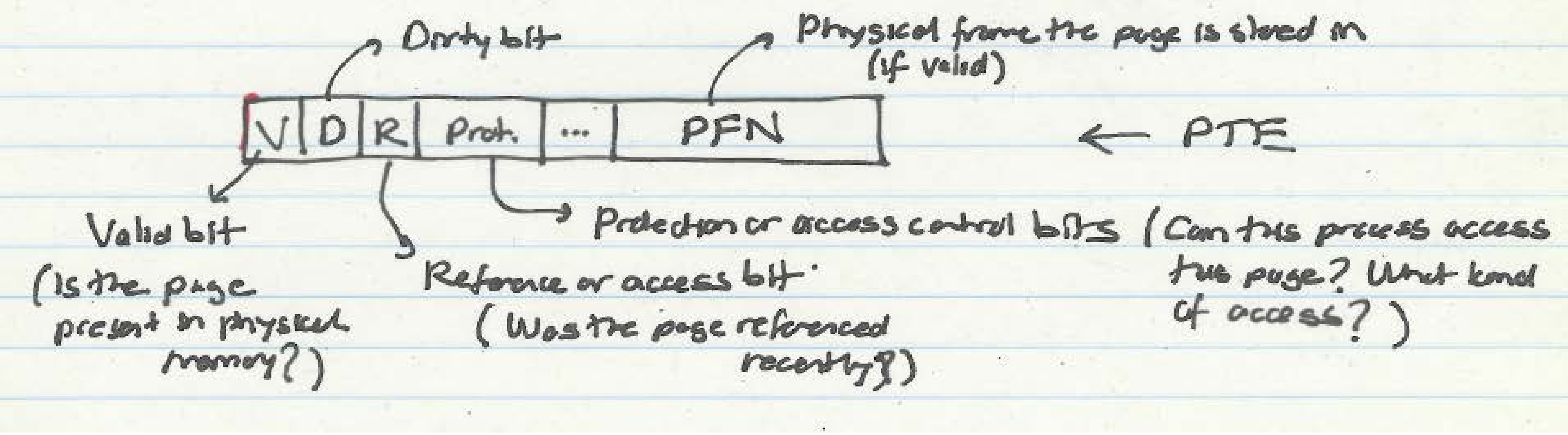
Address Translation

- Separate (set of) page table(s) per process
- VPN forms index into page table (points to a page table entry)
- Page Table Entry (PTE) provides information about page



What Is in a Page Table Entry (PTE)?

- Need a valid bit → to indicate validity/presence in physical memory
- Need PPN → to support translation
- Need bits to support replacement
- Need a dirty bit to support "write back caching"
- Need protection bits to enable access control and protection



Some Issues in Virtual Memory

Three Major Issues

- How large is the page table and how do we store and access it?

- How can we speed up translation & access control check?

- When do we do the translation in relation to cache access?

- There are many other issues we will not cover in detail
  - What happens on a context switch?
  - How can you handle multiple page sizes?
  - …

Virtual Memory Issue I

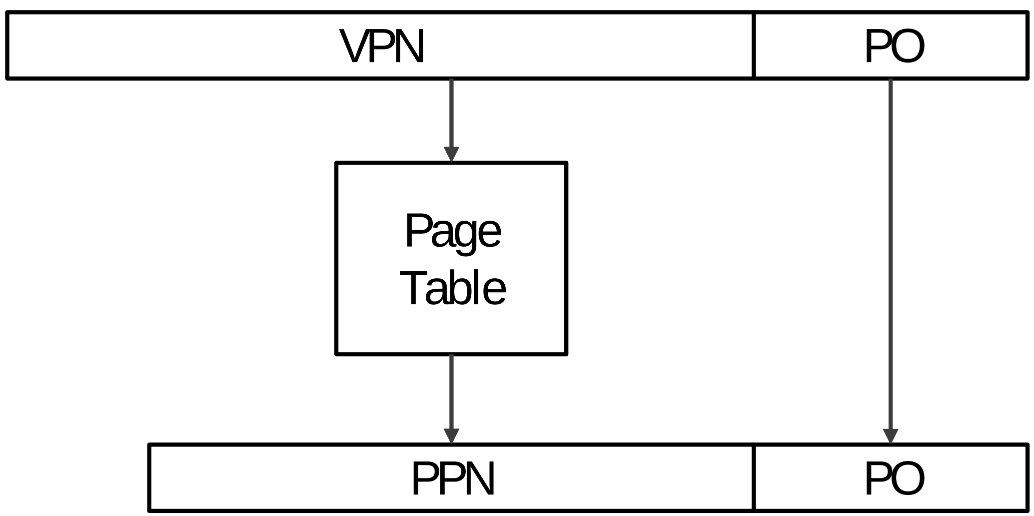
- How large is the page table?

- Where do we store it?
  - In hardware?
  - In physical memory? (Where is the PTBR?)
  - In virtual memory? (Where is the PTBR?)

- How can we store it efficiently without requiring physical memory that can store all page tables?
  - Idea: multi-level page tables
  - Only the first-level page table has to be in physical memory
  - Remaining levels are in virtual memory (but get cached in physical memory when accessed)

Issue: Page Table Size

- Suppose 64-bit VA, 40-bit PA and 4kB page size how large is the page table?
  - 2^52 entries * ~4 bytes per entry ≈16 * 10^15 bytes
  - And that is for just one process
  - And the process may not be using the entire VM space



Solution: Multi-Level Page Tables

- Four-level paging in x86

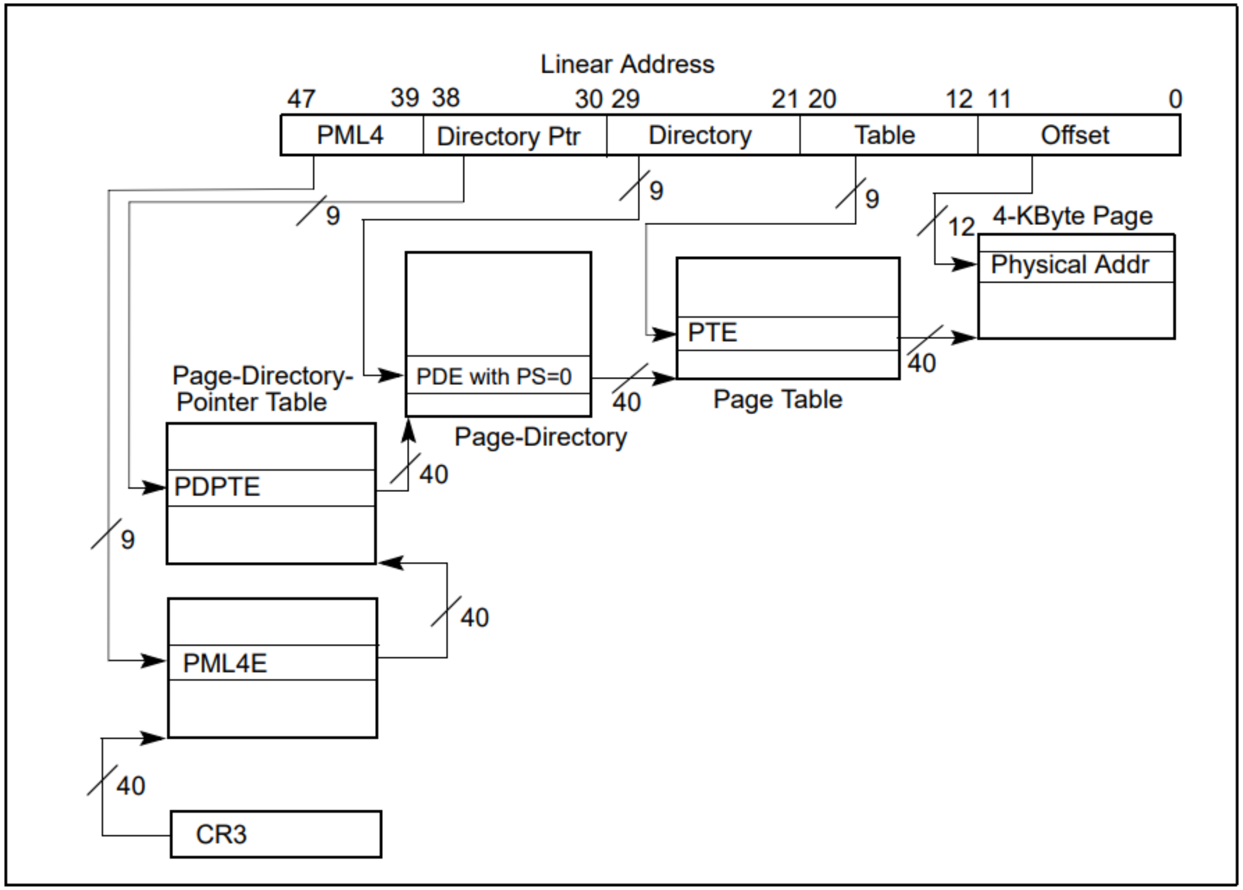


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

Intel® 64 and IA-32 Architectures Software Developer's Manual

Solution: Multi-Level Page Tables

- Four-level paging in x86

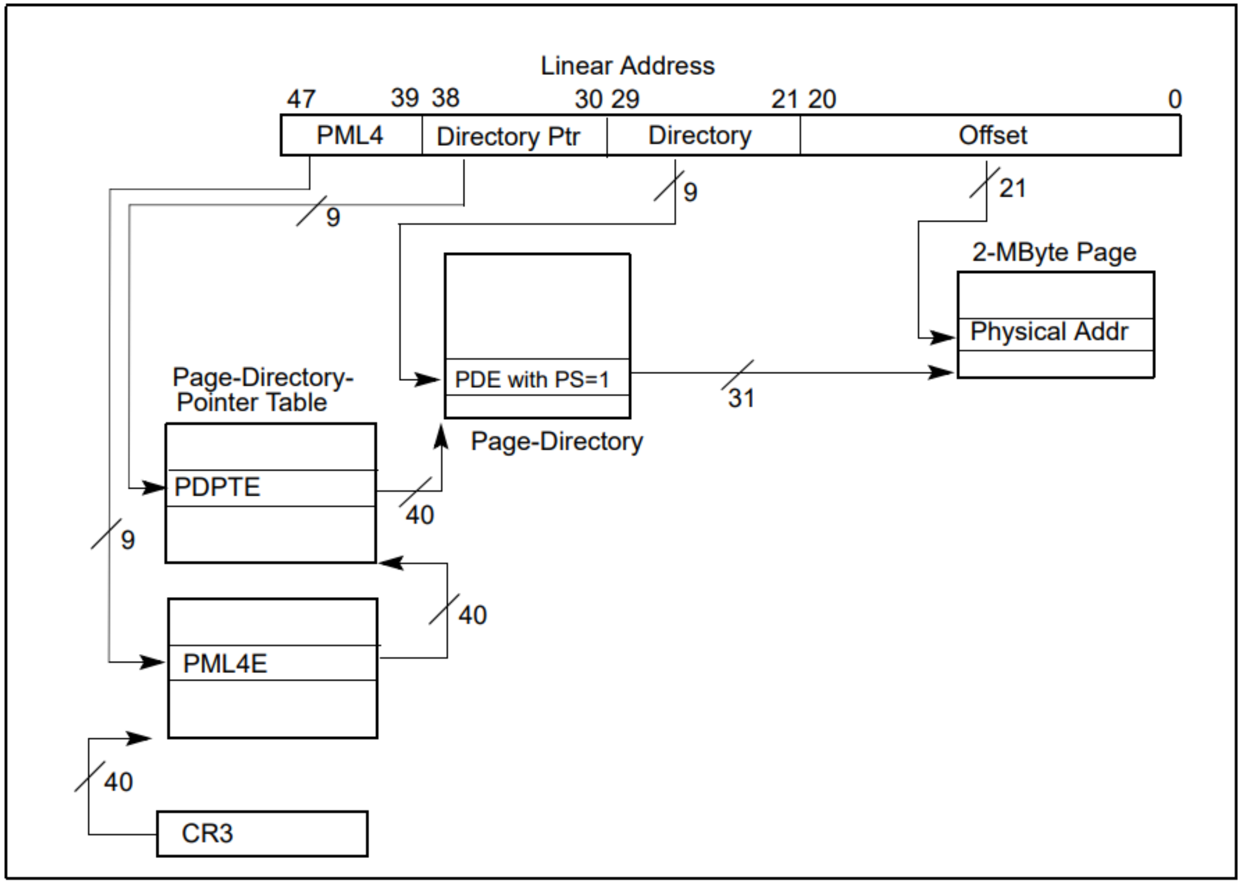


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

Intel® 64 and IA-32 Architectures Software Developer's Manual

Virtual Memory Issue II

- How fast is the address translation?
  - How can we make it fast?

- Idea: Use a hardware structure that caches PTEs → Translation lookaside buffer

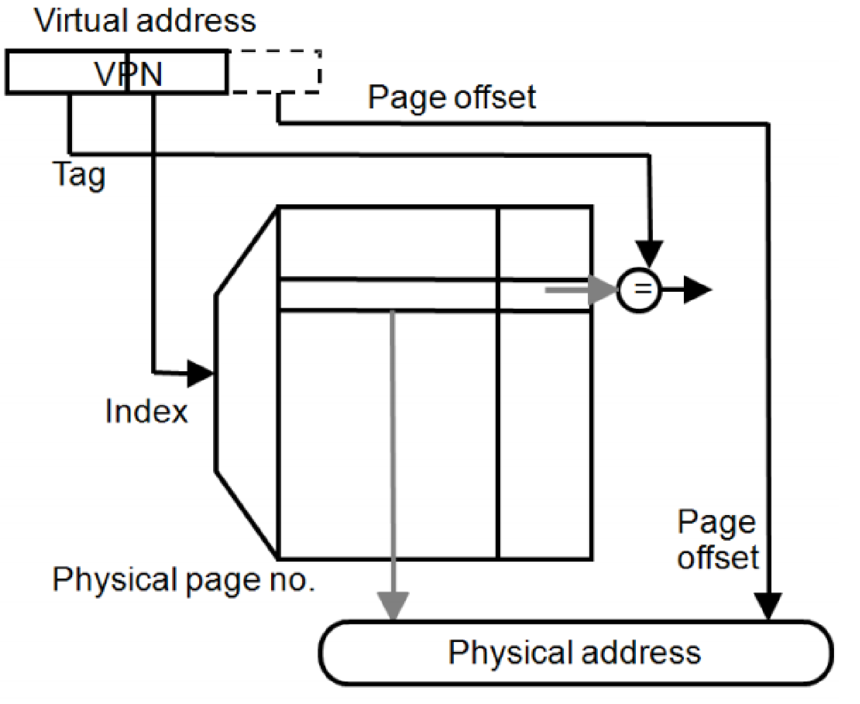
- What should be done on a TLB miss?
  - What TLB entry to replace?
  - Who handles the TLB miss? HW vs. SW?

- What should be done on a page fault?
  - What virtual page to replace from physical memory?
  - Who handles the page fault? HW vs. SW?

Speeding up Translation with a TLB

- Essentially a cache of recent address translations
  - Avoids going to the page table on every reference

- Index = lower bits of VPN
- Tag = unused bits of VPN + process ID
- Data = a PTE
- Status = valid, dirty



Virtual Memory and Cache Interaction

Address Translation and Caching

- When do we do the address translation?
  - Before or after accessing the L1 cache?
- In other words, is the cache virtually addressed or physically addressed?
  - Virtual versus physical cache
- What are the issues with a virtually addressed cache?
- Synonym problem:
  - Two different virtual addresses can map to the same physical address $\rightarrow$ same physical address can be present in multiple locations in the cache $\rightarrow$ can lead to inconsistency in data

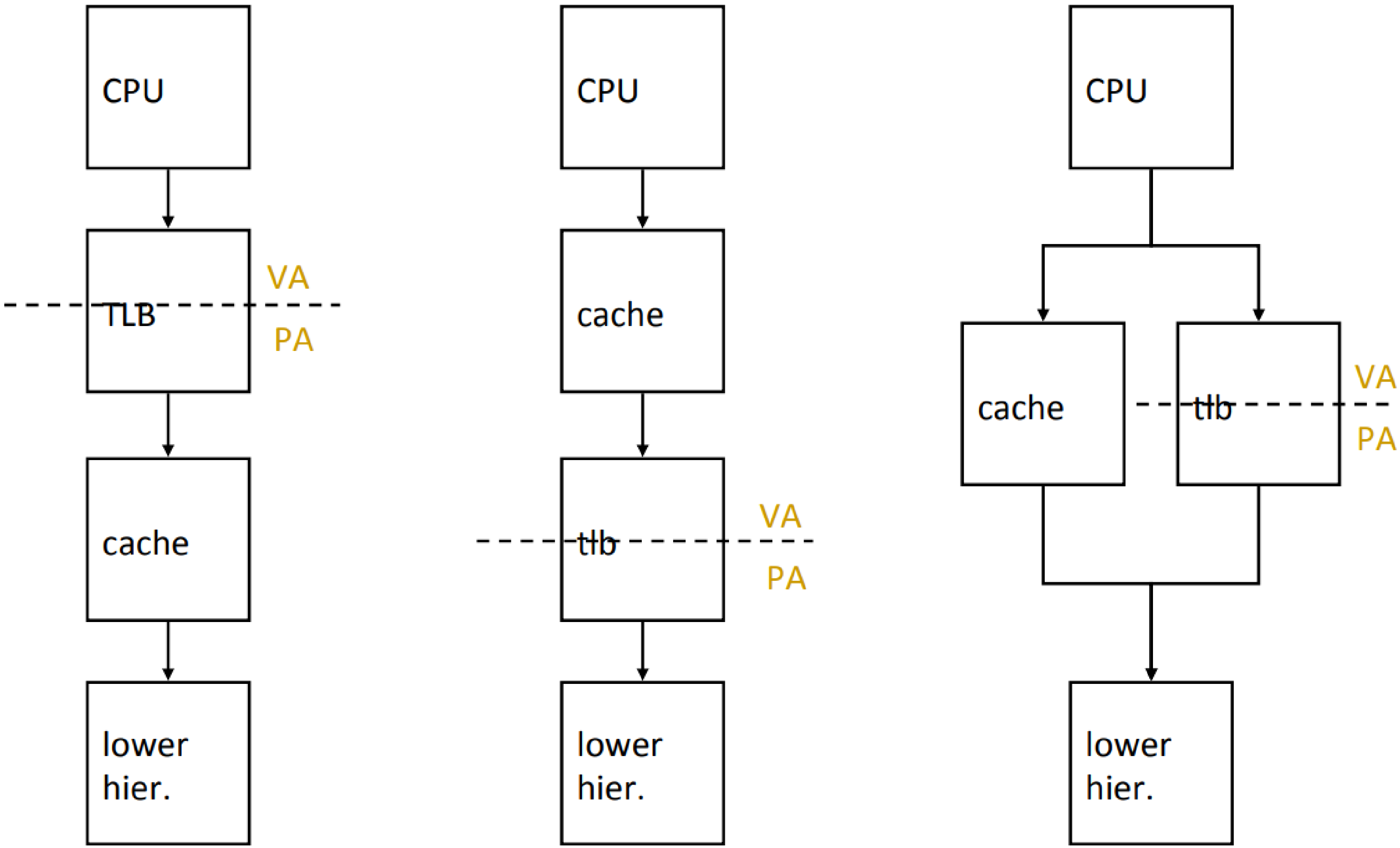
Homonyms and Synonyms

- Homonym: Same VA can map to two different PAs
  - Why?
    - VA is in different processes

- Synonym: Different VAs can map to the same PA
  - Why?
    - Different pages can share the same physical frame within or across processes
    - Reasons: shared libraries, shared data, copy-on-write pages within the same process, …

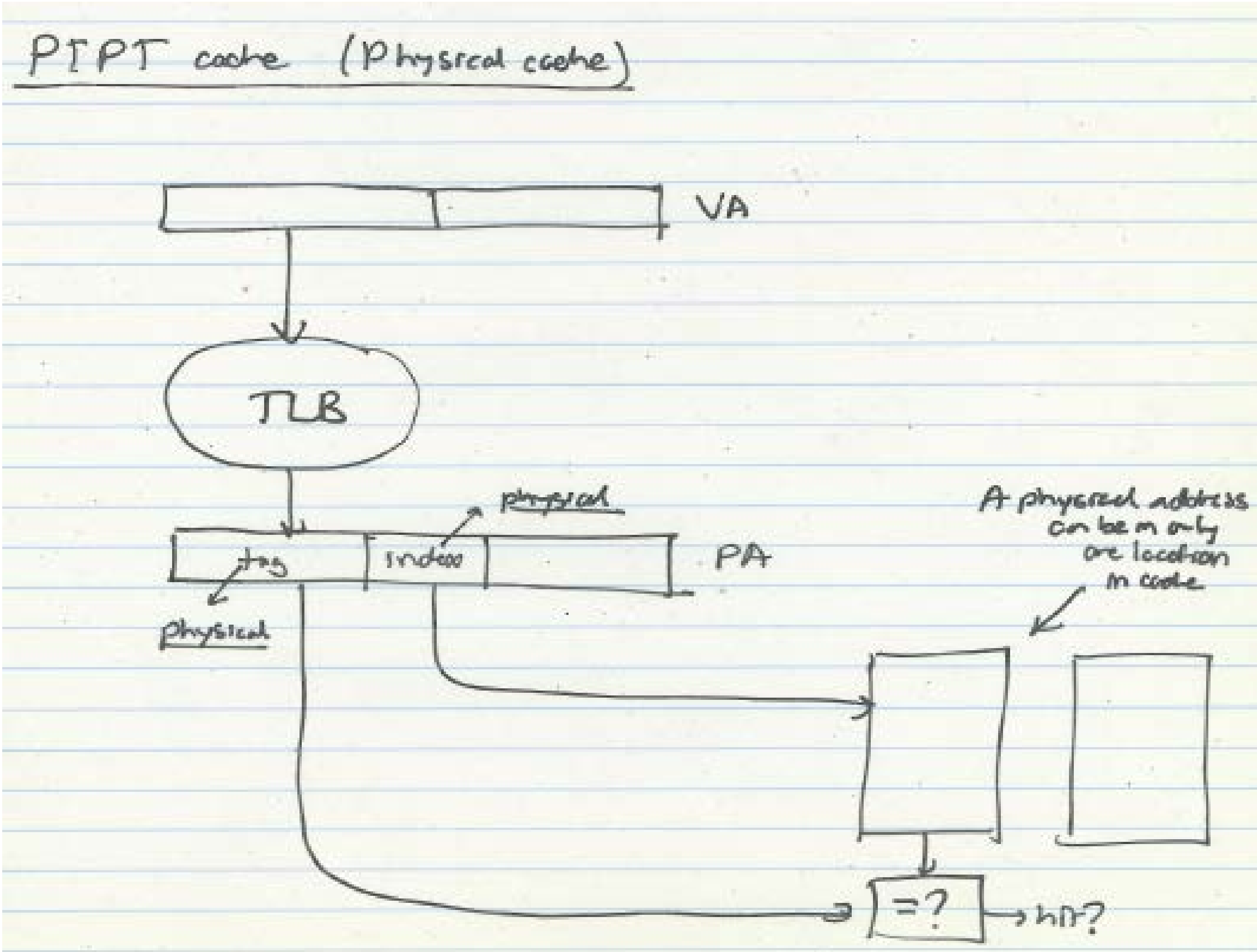
- Do homonyms and synonyms create problems when we have a cache?
  - Is the cache virtually or physically addressed?

Cache-VM Interaction

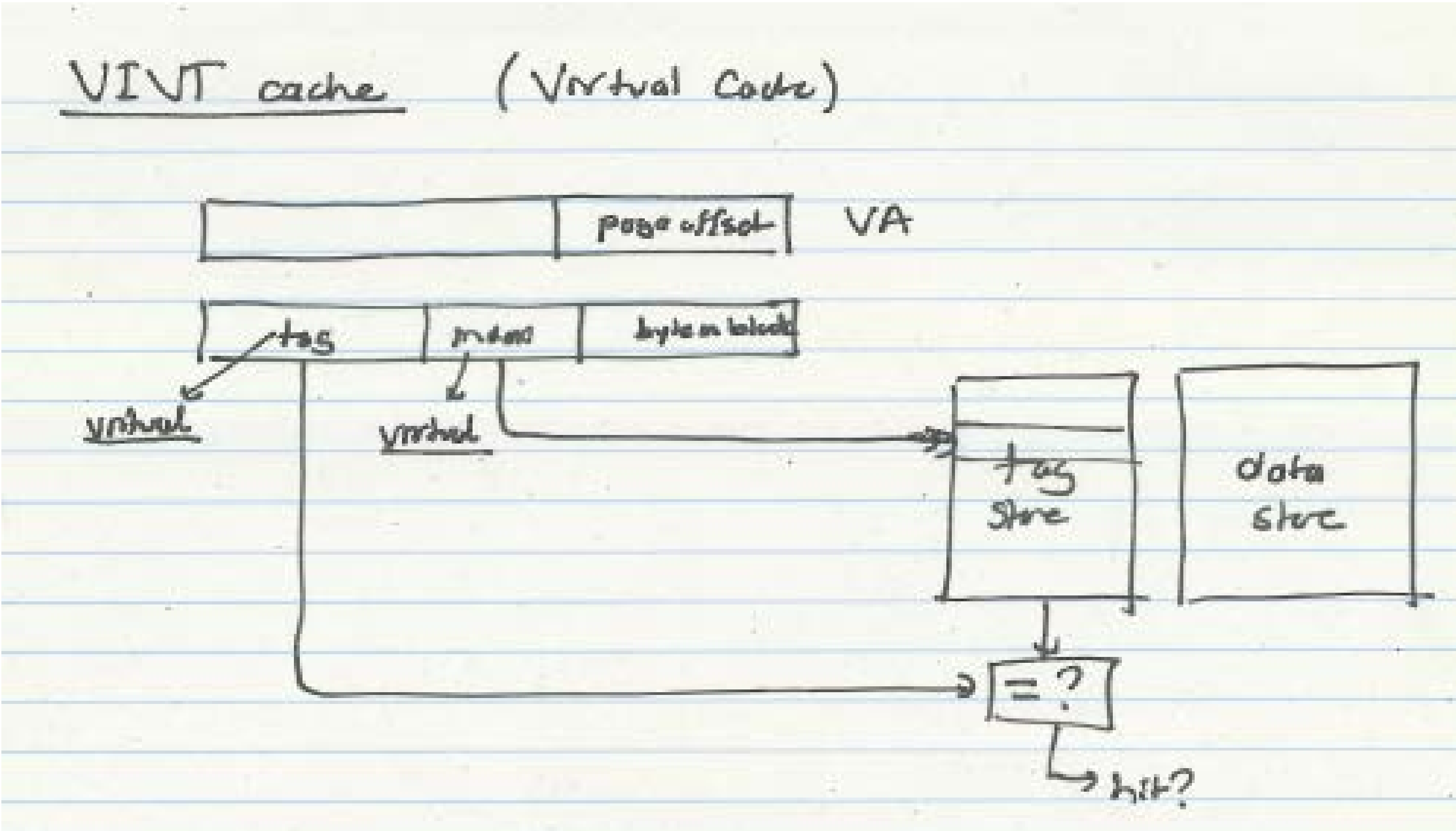


PIPT VIVT VIPT

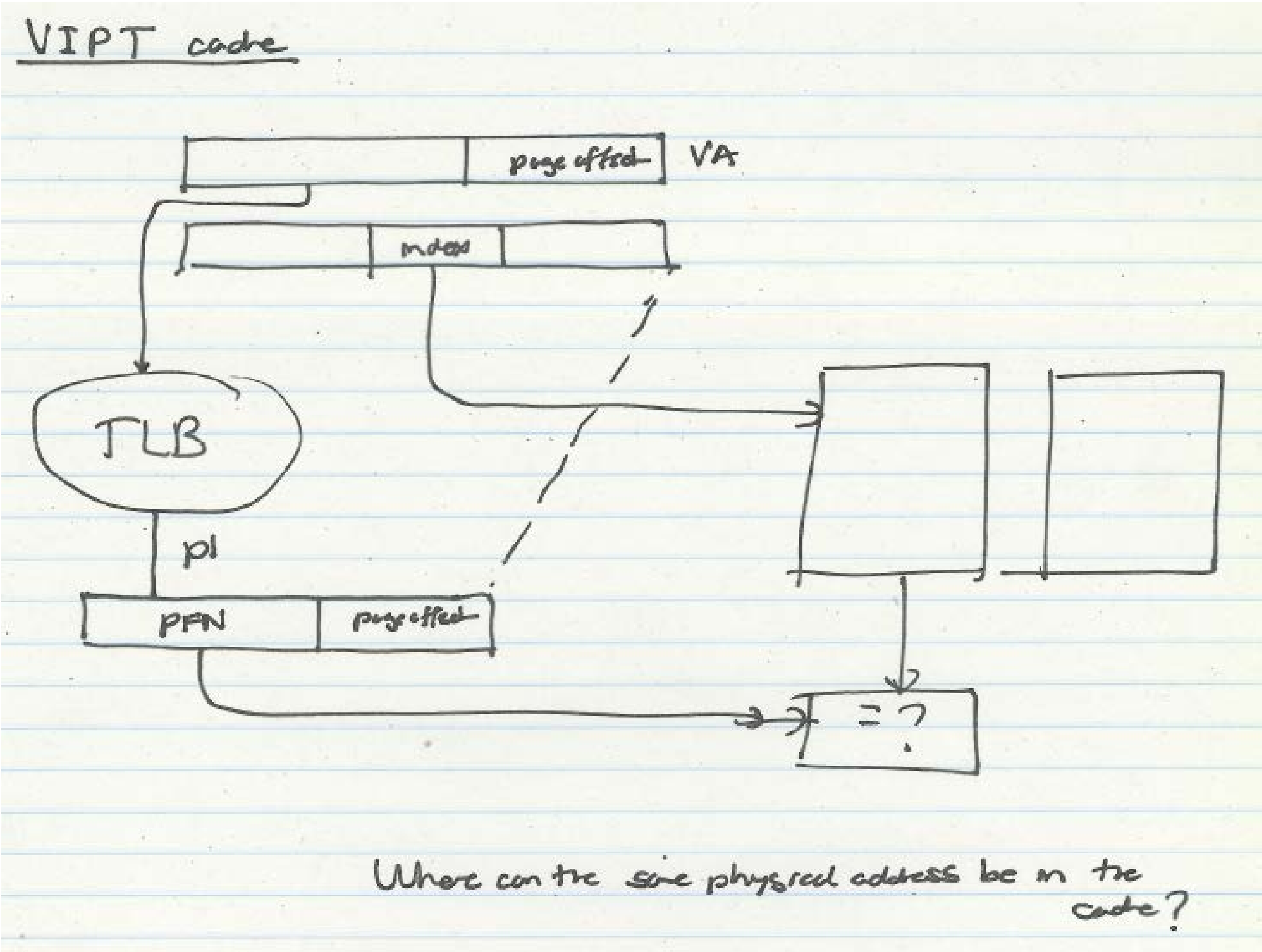
Physical Cache



Virtual Cache



VIPT


VIPT cache
page offset
VA
index
TLB
pl
PFN
page offset
=?
Where can the same physical address be in the cache?

Slides Credit

- Attiano Purpura-Pontoniere
- Yuchen Hao
- Onur Mutlu