Virtual Memory

Concept and TLB

主讲人: 邓倩妮

上海交通大学

内容来自：

计算机组成与设计，第四版， 第五章

Virtual Memory

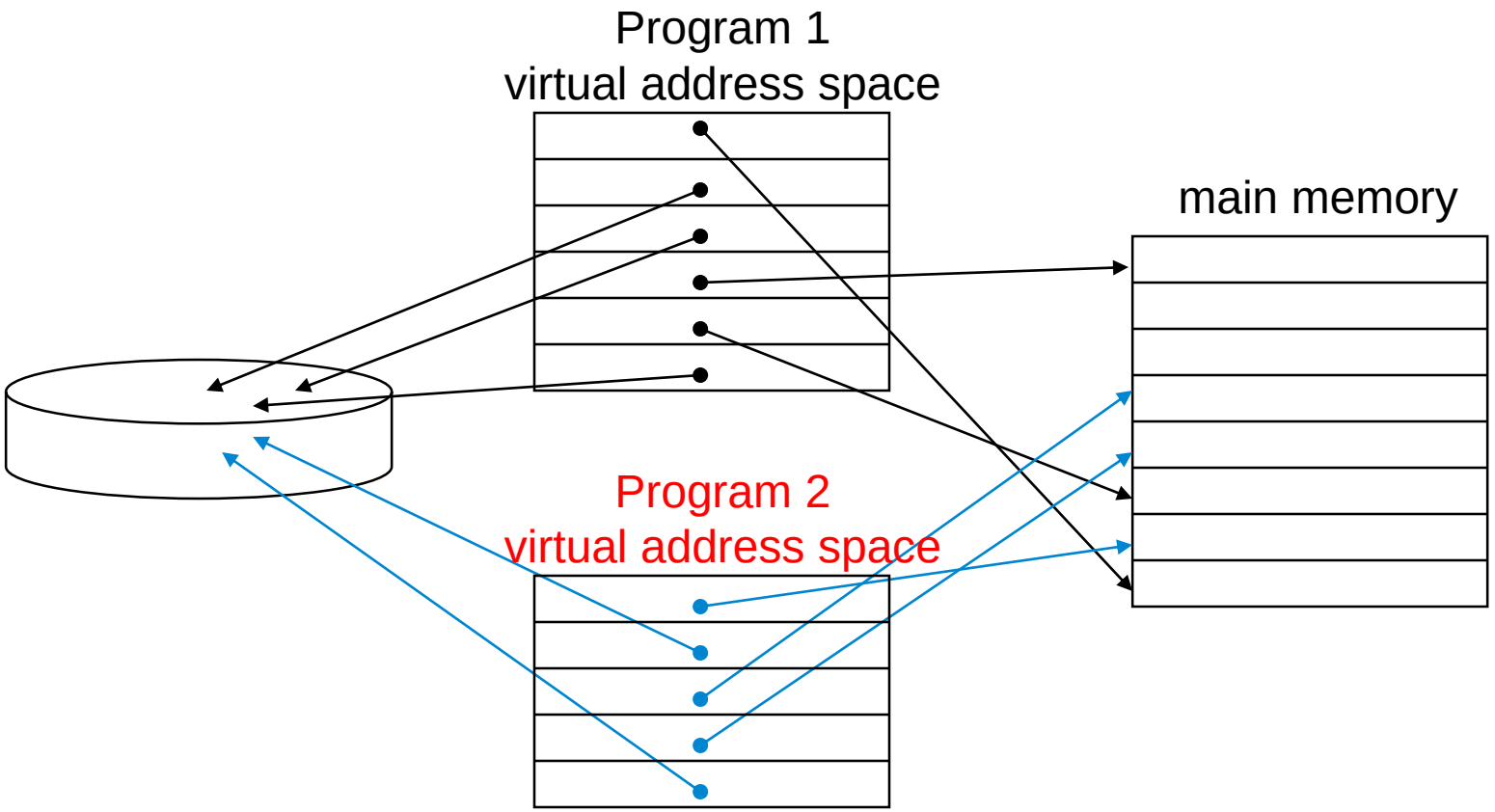
- Use main memory as a “cache” for secondary memory
  - Location-independent programs: Provides the ability to easily run programs larger than the size of physical memory
  - Protection: Allows efficient and safe sharing of memory among multiple programs
  - Simplifies loading a program for execution by providing for code relocation (i.e., the code can be loaded anywhere in main memory)

Virtual Memory

- What makes it work? – again the Principle of Locality
  - A program is likely to access a relatively small portion of its address space during any period of time
- Each program is compiled into its own address space – a "virtual" address space
  - During run-time each virtual address must be translated to a physical address (an address in main memory)

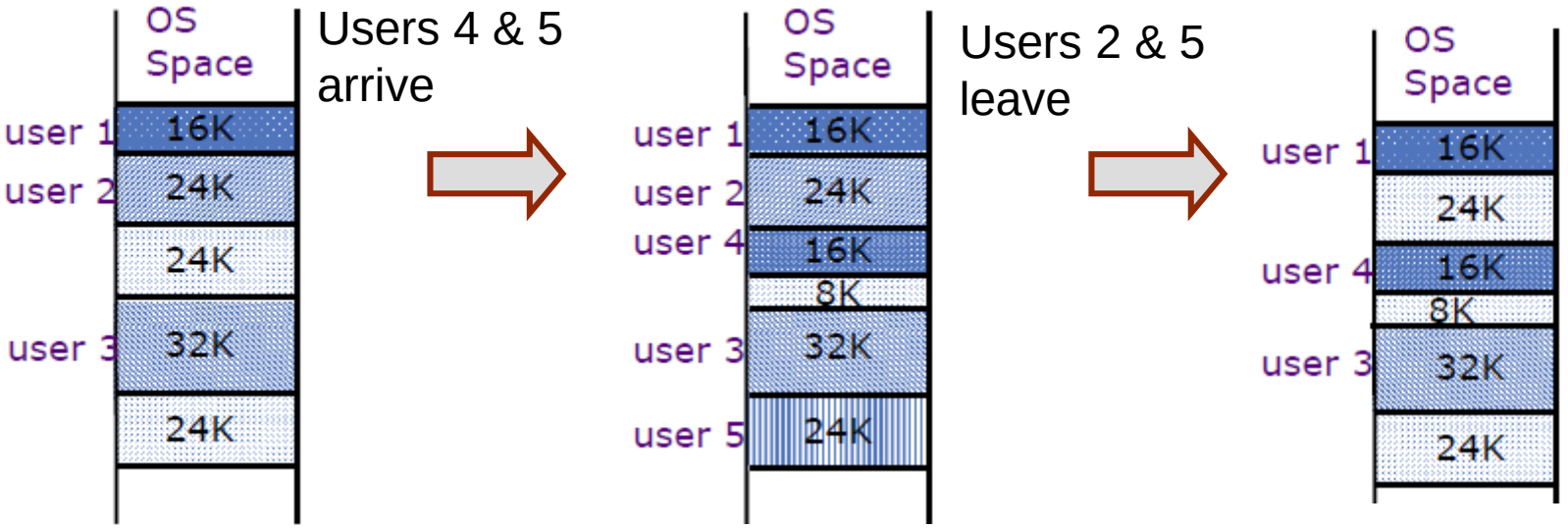
Two Programs Sharing Physical Memory

- A program's address space is divided into pages (all one fixed size) or segments (variable sizes)



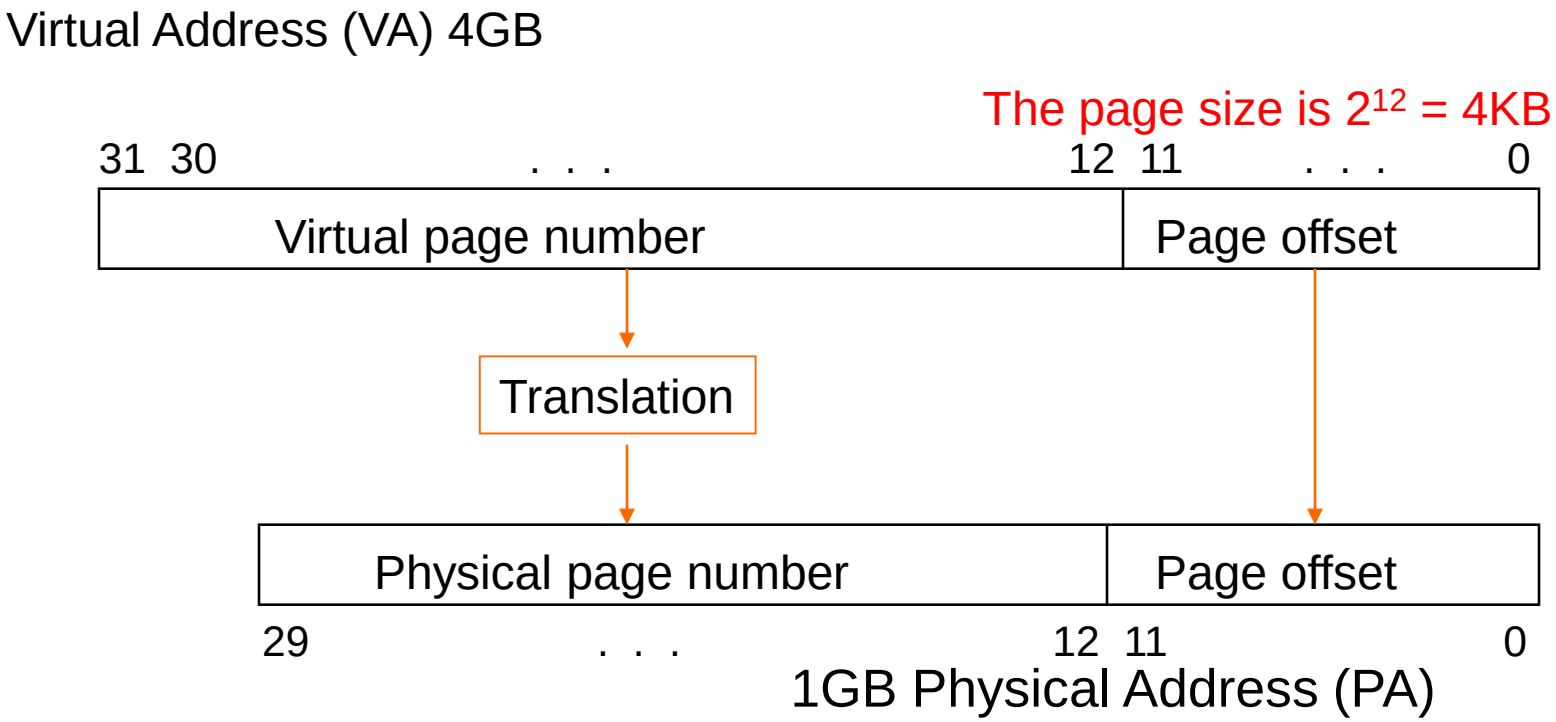
Why Paged Memory Systems

To Avoid Memory Fragmentation



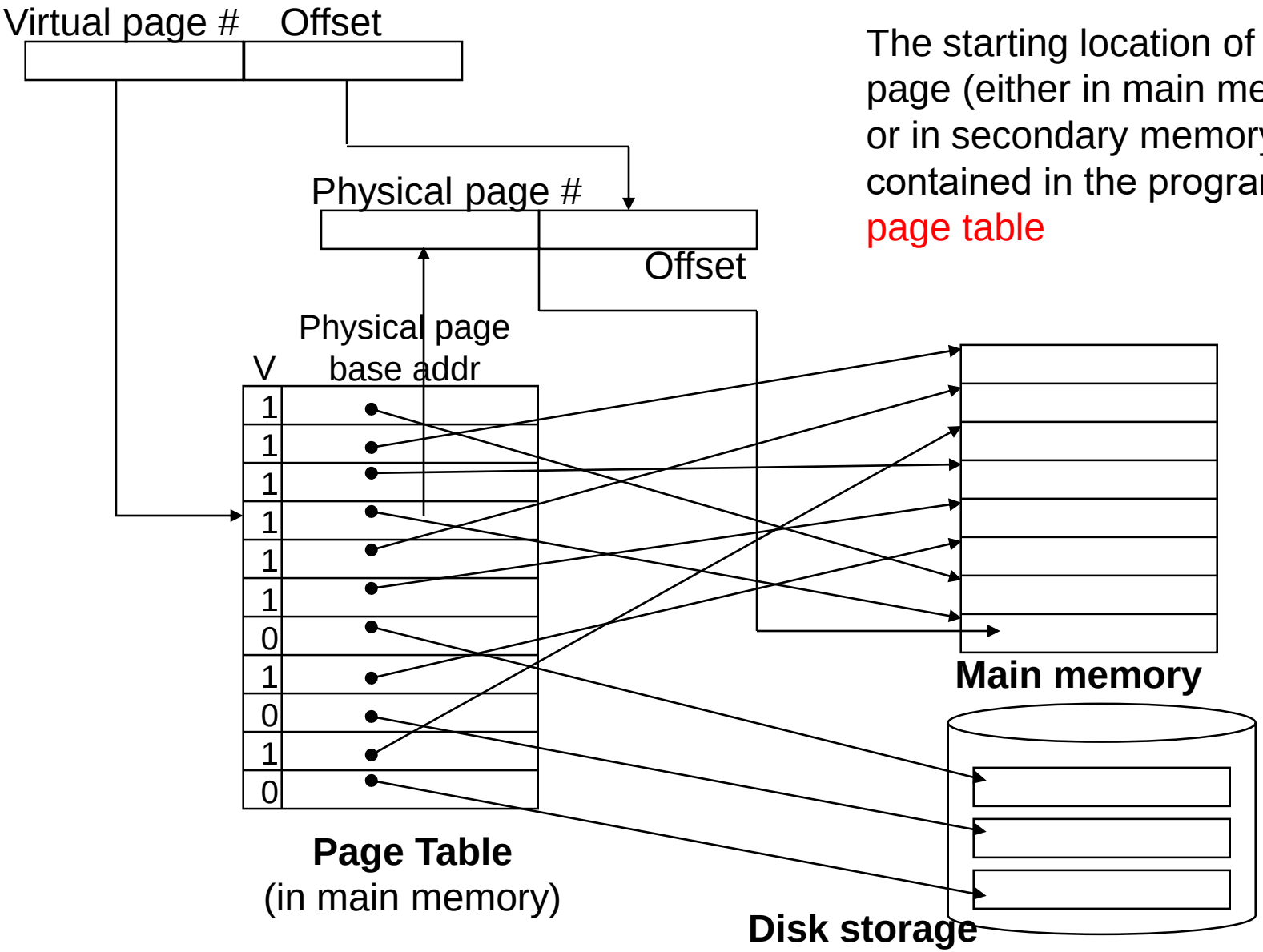
As users come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage.

Address Translation



- So each memory request *first* requires an address translation from the virtual space to the physical space
  - A virtual memory miss (i.e., when the page is not in physical memory) is called a page fault

Address Translation Mechanisms



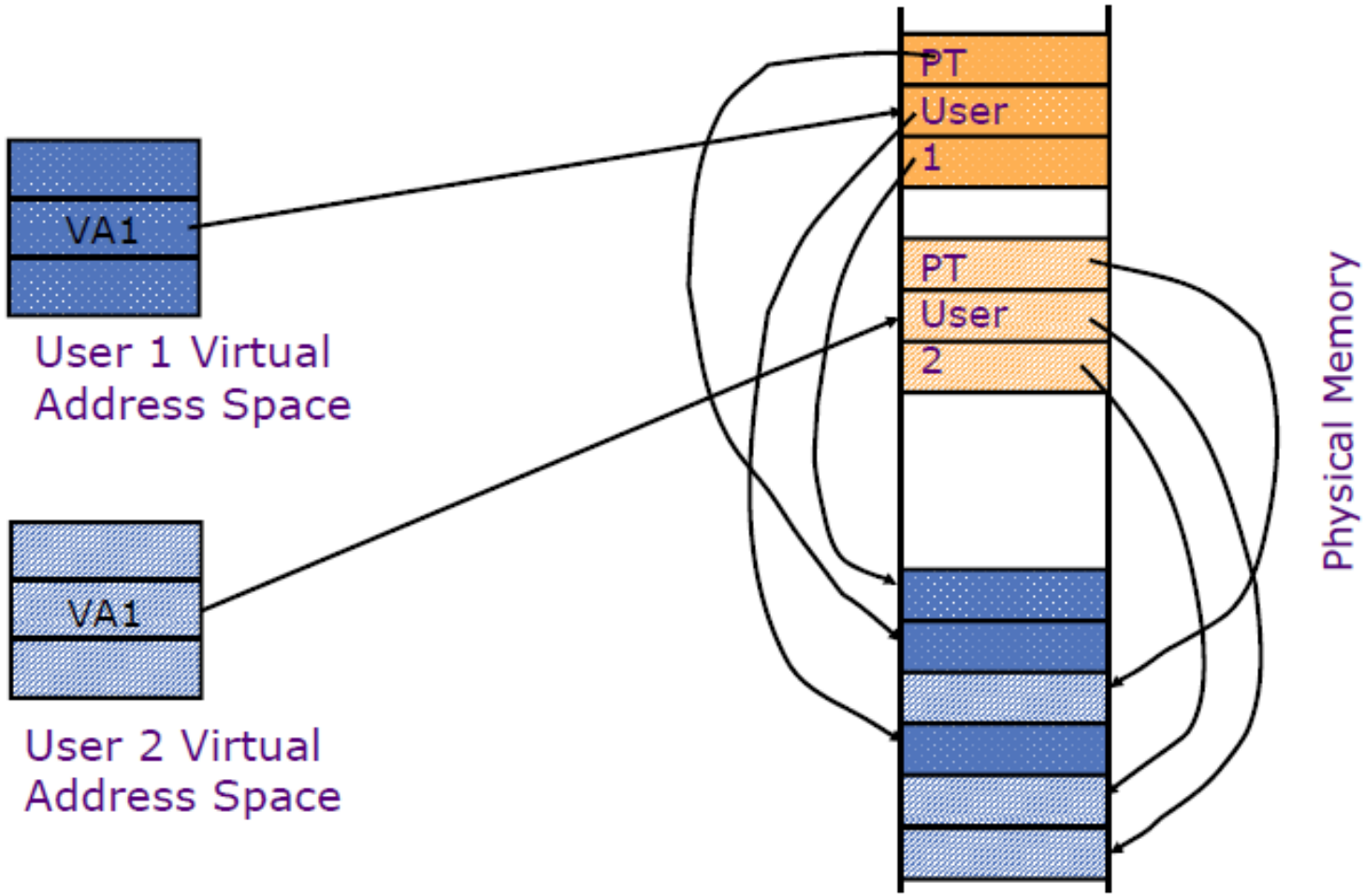
The starting location of each page (either in main memory or in secondary memory) is contained in the program's page table

Where Should Page Tables Reside?

- Space required by the page tables (PT) is proportional to the address space, number of users, (inverse to) size of each page, …
  - Space requirement is large
  - Too expensive to keep in registers

- Idea: Keep PTs in the main memory

Page Tables in Physical Memory

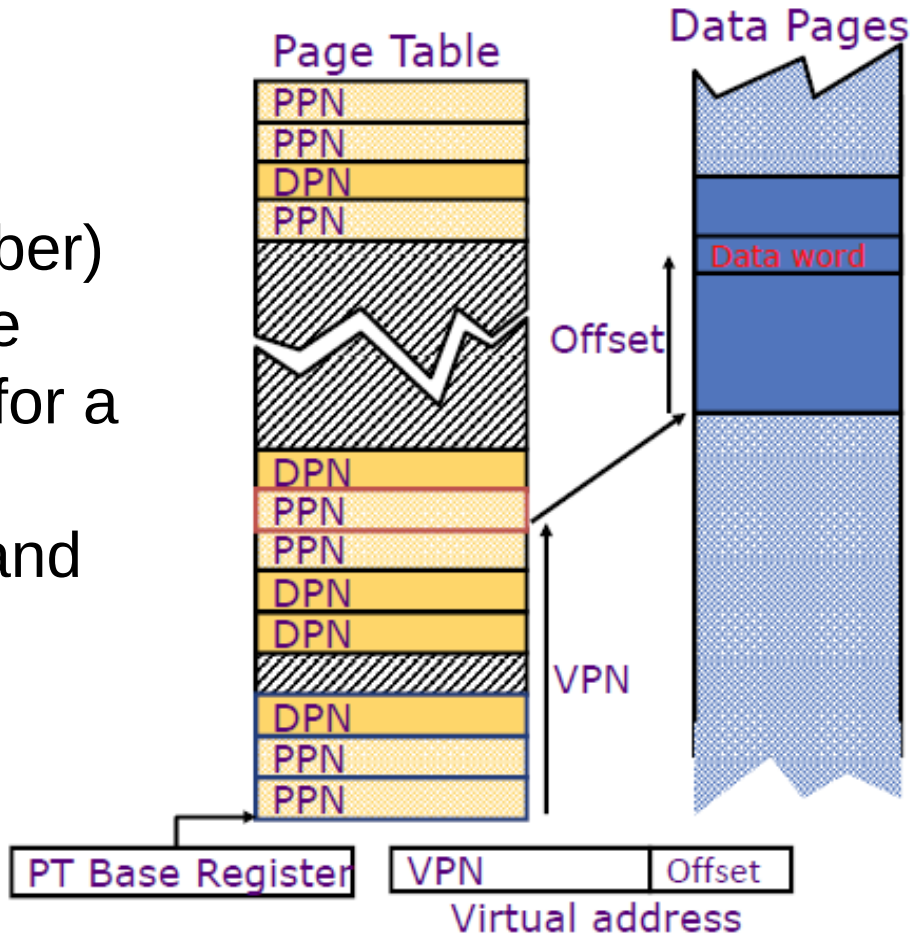


Linear Page Table

Page Table Entry (PTE) contains:

- A bit to indicate if a page exists
- PPN (physical page number) for a memory-resident page
- DPN (disk page number) for a page on the disk
- Status bits for protection and usage

OS sets the Page Table Base Register whenever active user process changes

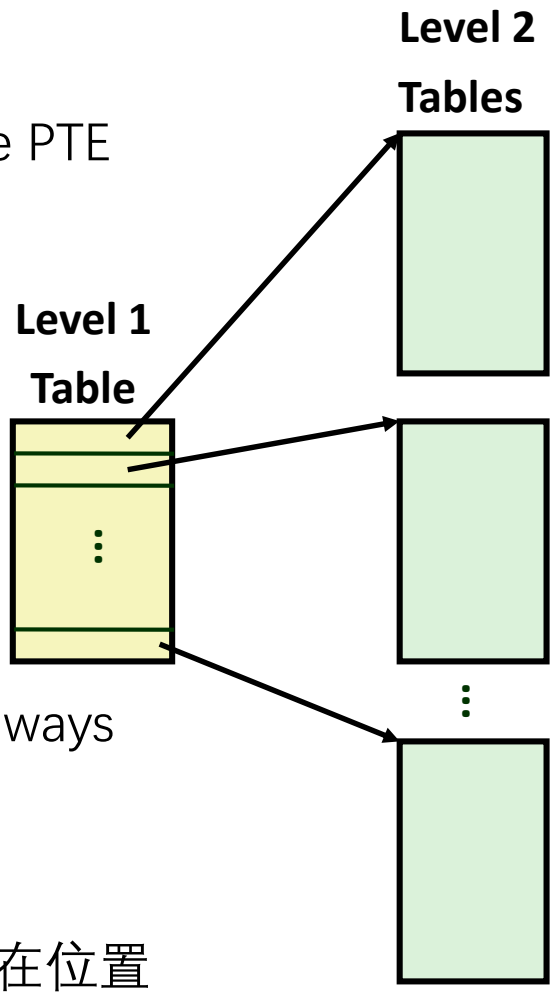


Size of Linear Page Table

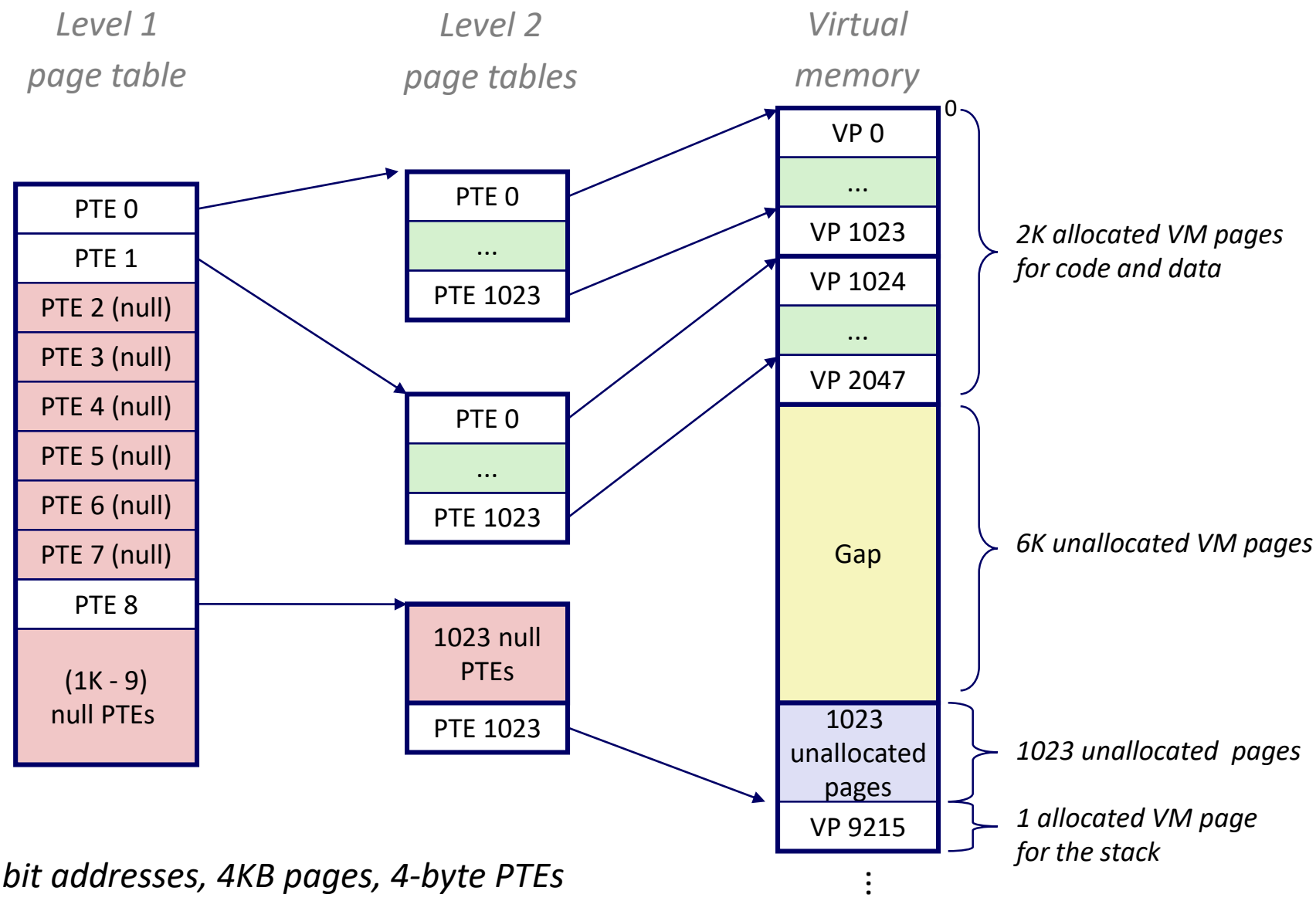
- With 32-bit addresses, 4KB pages & 4byte PTEs:
  - 2^{20} PTEs, i.e, 4 MB page table per user per process
- What about 64-bit virtual address space???
  - Even 1MB pages would require 2^{44} 8byte PTEs (128 TB!)
- Larger pages?
  - Internal fragmentation (Not all memory in page is used)
  - Larger page fault penalty (more time to read from disk)

Multi-Level Page Tables

- Suppose:
  - 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE
- Problem:
  - Would need a 512 GB page table!
    - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes
- Common solution: Multi-level page table
- Example: 2-level page table
  - Level 1 table: each PTE points to a page table (always memory resident) 记录页表所在位置
  - Level 2 table: each PTE points to a page (paged in and out like any other data) 记录页所在位置



A Two-Level Page Table Hierarchy



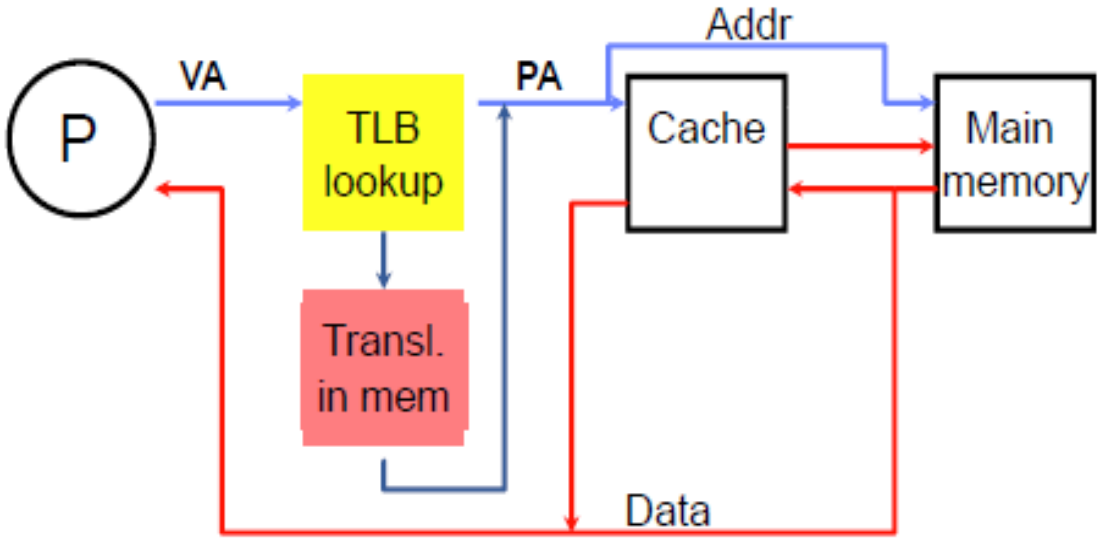
32 bit addresses, 4KB pages, 4-byte PTEs

Fast address translation

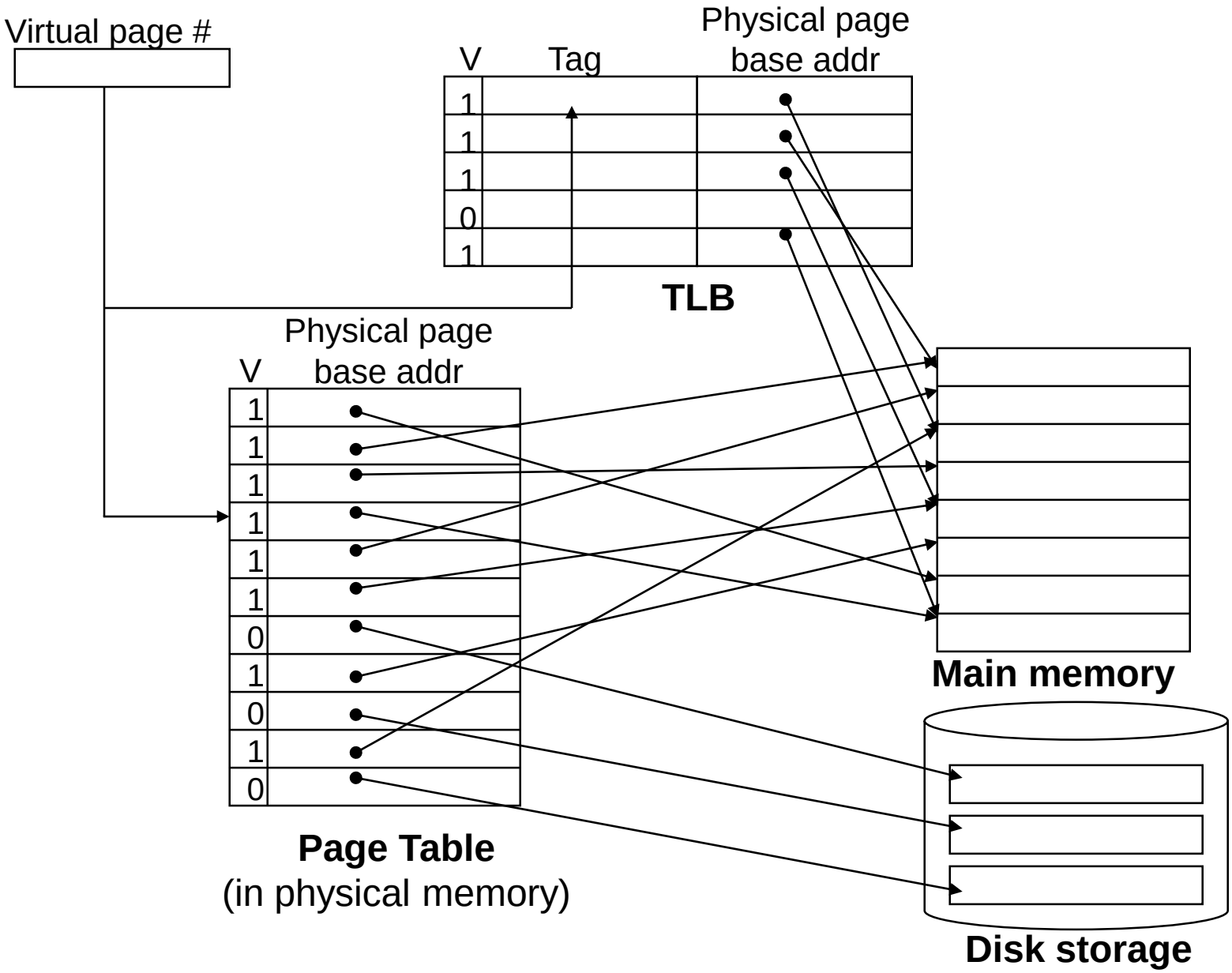
How can we avoid three extra memory references for each original memory reference?

- Store the most commonly used address translations in a cache—***Translation Look-aside Buffer*** (TLB)

==> *The caches rears their ugly faces again!*



Making Address Translation Fast



Translation Lookaside Buffers (TLBs)

- Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

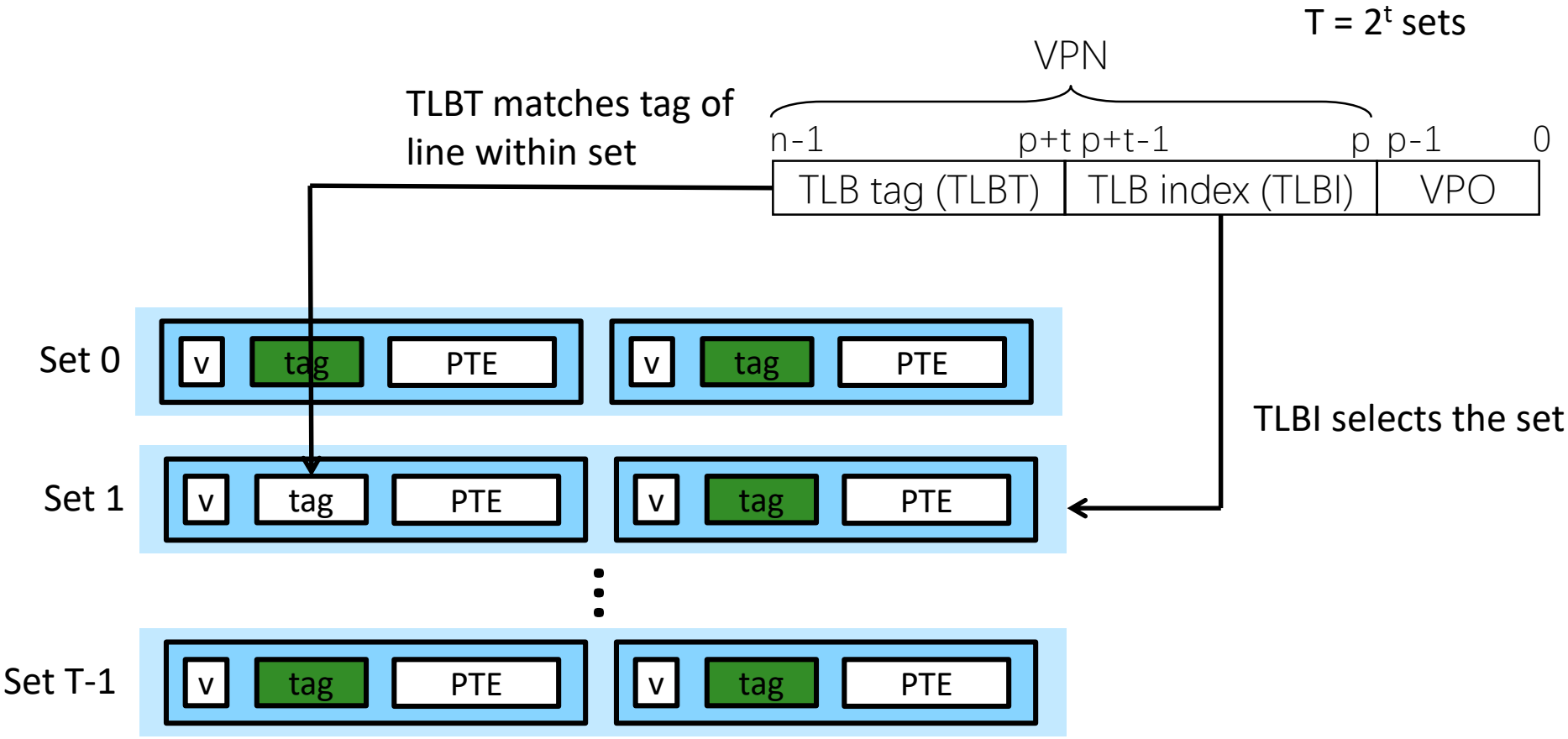
V	Virtual Page #	Physical Page #	Dirty	Ref	

- TLB access time is typically smaller than cache access time (because TLBs are much smaller than caches)

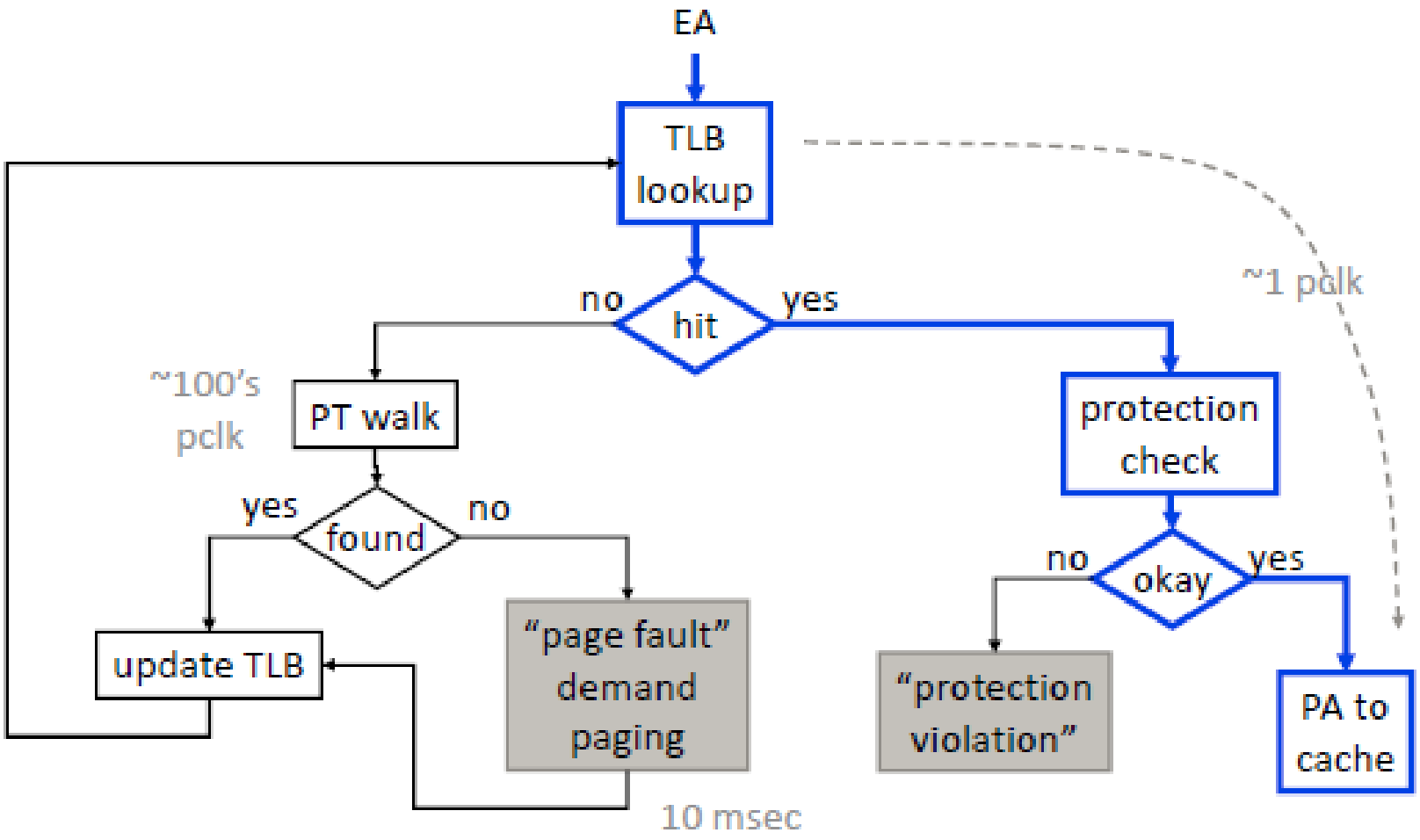
  TLBs are typically not more than 128 to 256 entries even on high end machines

Accessing the TLB

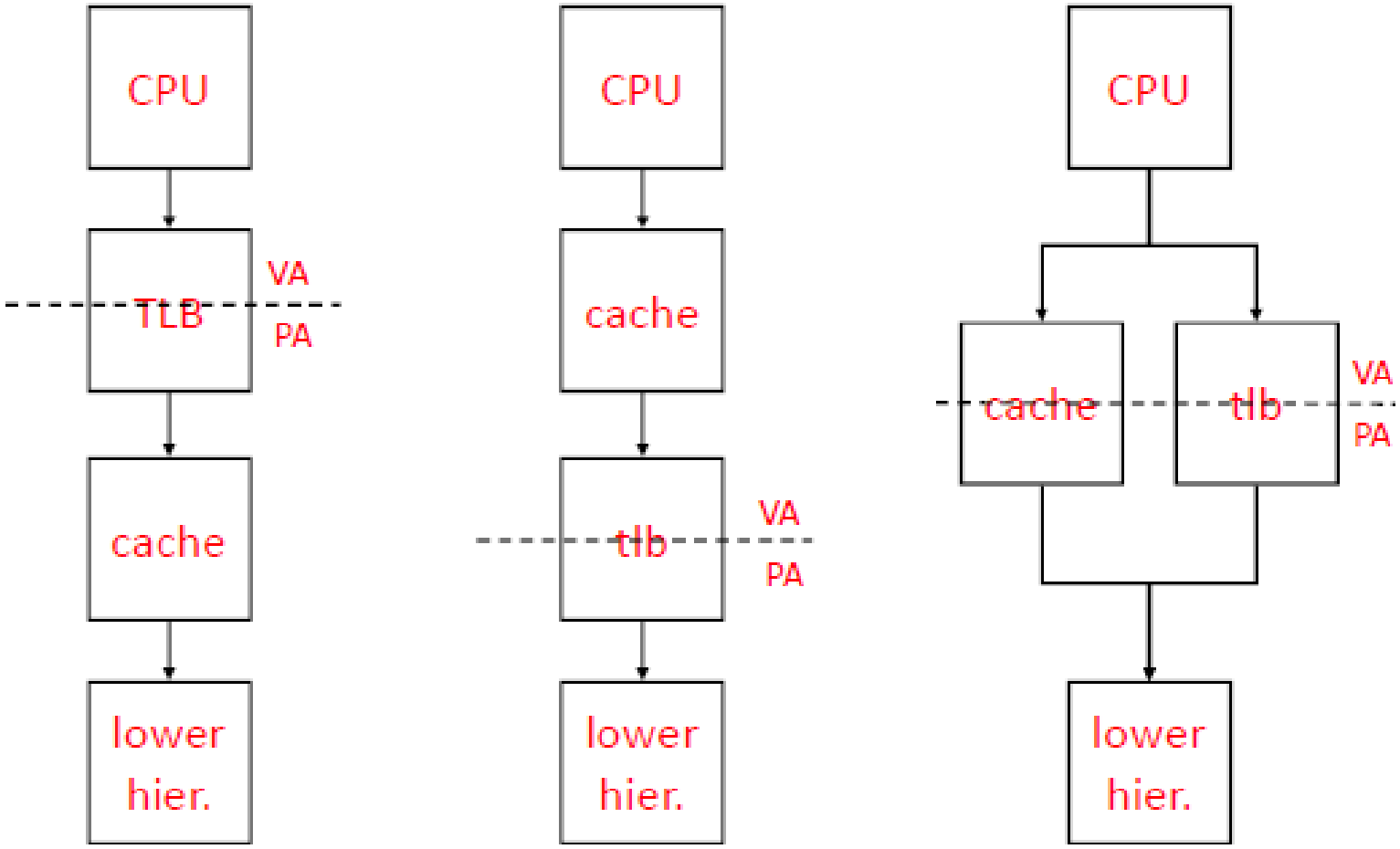
- Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped



VA to PA Translation



How should VM and Caches Interact?

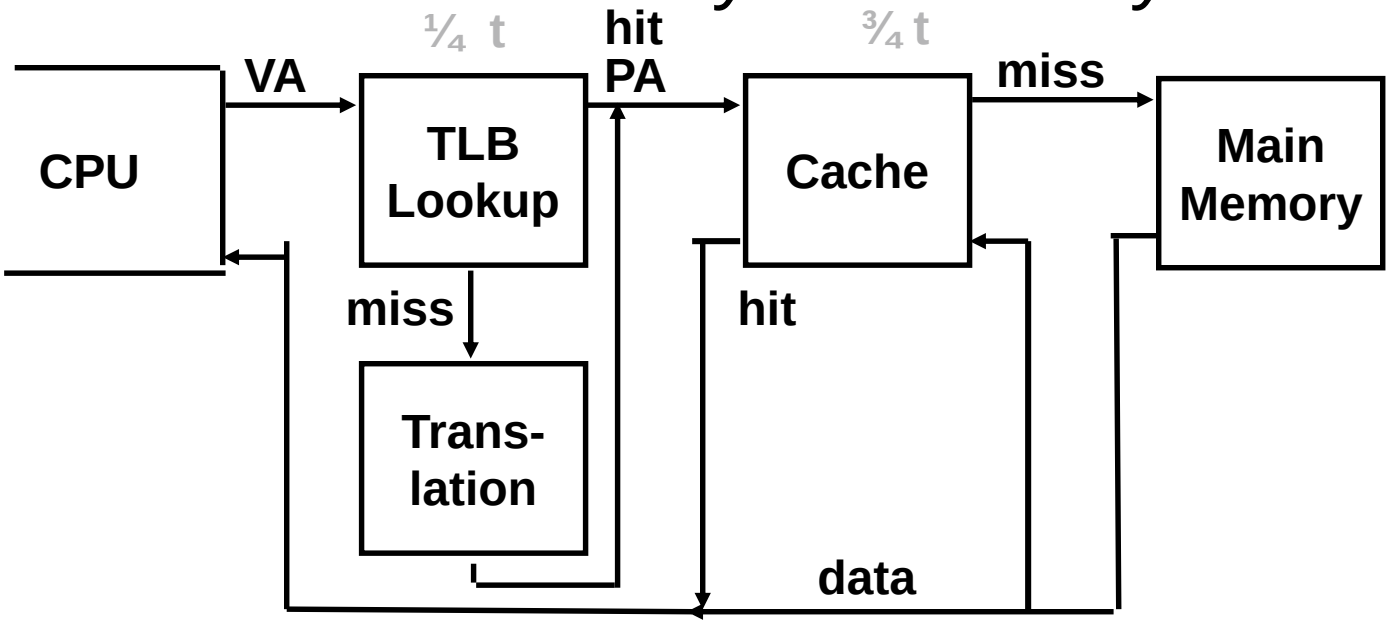


physical cache

virtual (L1) cache

hybrid??

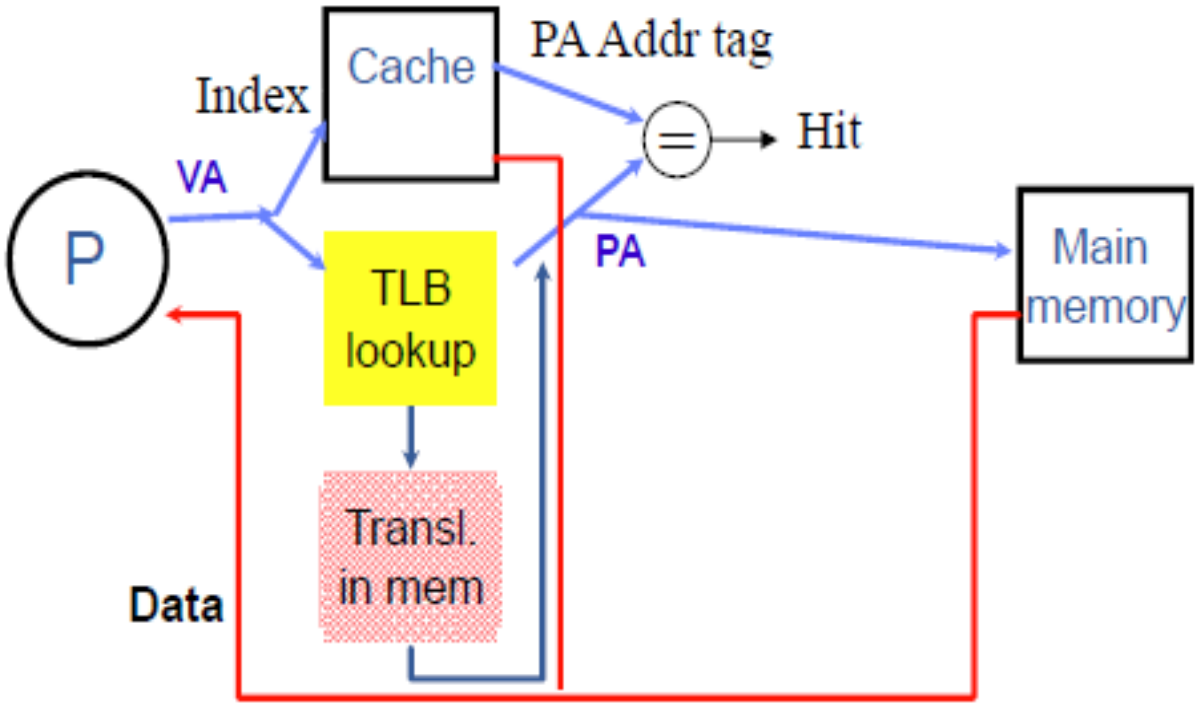
A TLB in the Memory Hierarchy



- A TLB miss – is it a page fault or merely a TLB miss?
  - If the page is loaded into main memory, then the TLB miss can be handled (in hardware or software) by loading the translation information from the page table into the TLB
    - Takes 10's of cycles to find and load the translation info into the TLB
  - If the page is not in main memory, then it's a true page fault
    - Takes 1,000,000's of cycles to service a page fault

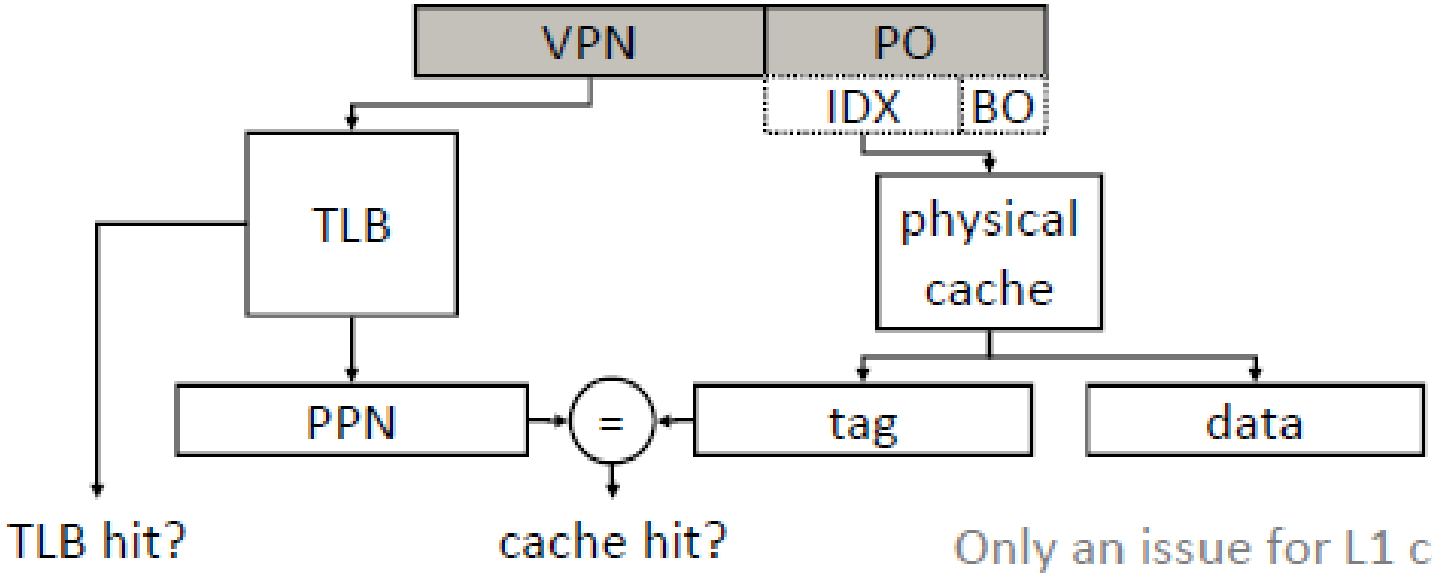
Virtually Indexed Physically Tagged: VIPT

- Have to guarantee that all aliases have the same index
- L1_cach



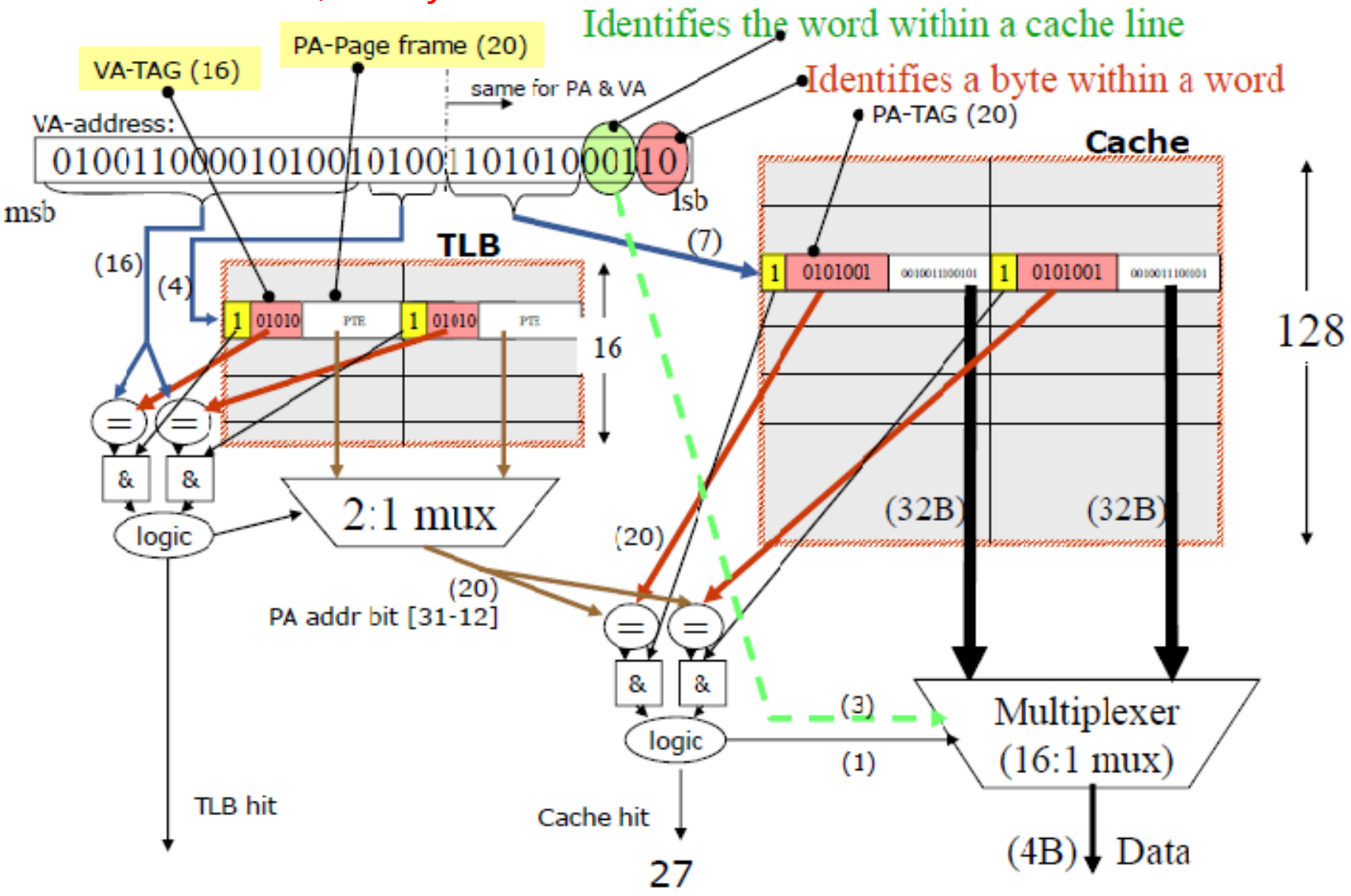
Virtually-Indexed Physically-Tagged

- If C≤(page_size × associativity), the cache index bits come only from page offset (same in VA and PA)
- If both cache and TLB are on chip
  - index both arrays concurrently using VA bits
  - check cache tag (physical) against TLB output at the end



Putting it all together: VIPT

Cache: 8kB, 2-way, CL=32B, word=4B, page =4kB
TLB: 32 entries, 2-way



TLB Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	
Hit	Hit	Miss	
Miss	Hit	Hit	
Miss	Hit	Miss	
Miss	Miss	Miss	
Hit	Miss	Miss/ Hit	
Miss	Miss	Hit	

TLB Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	Yes – what we want!
Hit	Hit	Miss	Yes – although the page table is not checked if the TLB hits
Miss	Hit	Hit	Yes – TLB miss, PA in page table
Miss	Hit	Miss	Yes – TLB miss, PA in page table, but data not in cache
Miss	Miss	Miss	Yes – page fault
Hit	Miss	Miss/ Hit	Impossible – TLB translation not possible if page is not present in memory
Miss	Miss	Hit	Impossible – data not allowed in cache if page is not in memory

Handling a TLB Miss

- Software (MIPS, Alpha)
  - TLB miss causes an exception and the operating system walks the page tables and reloads TLB. A privileged "untranslated" addressing mode used for walk
- Hardware (SPARC v8, x86, PowerPC)
  - A memory management unit (MMU) walks the page tables and reloads the TLB
  - If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page-Fault exception for the original instruction

Summary

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.

- Caches, TLBs, Virtual Memory all understood by examining how they deal with the four questions
  1. Where can block be placed?
  2. How is block found?
  3. What block is replaced on miss?
  4. How are writes handled?

- Page tables map virtual address to physical address
  - TLBs are important for fast translation

谢谢！