# CSE 520
# Computer Architecture II

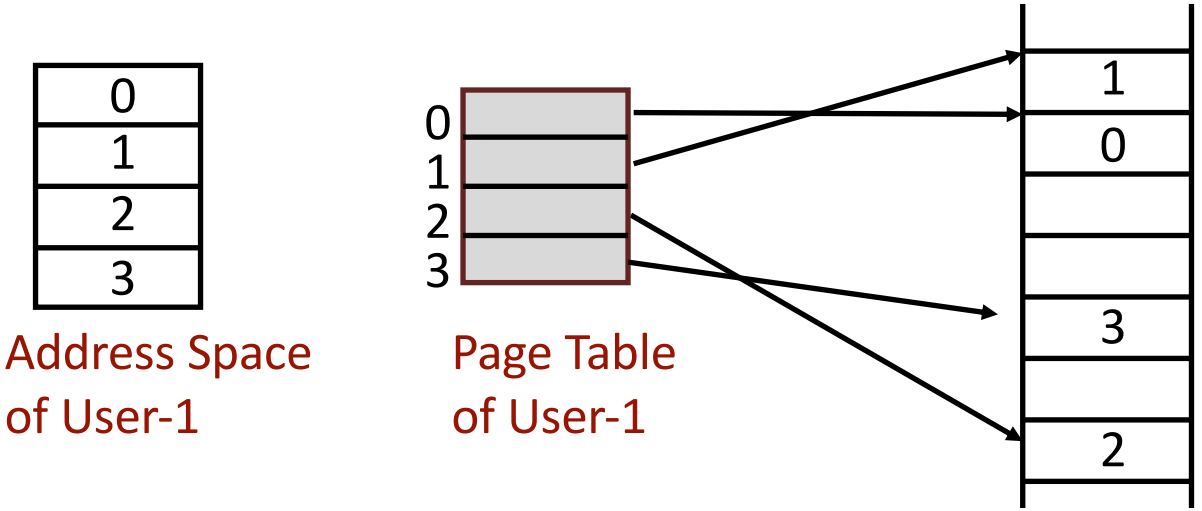
## Advanced Memory Operations
Supplement

Prof. Michel A. Kinsy

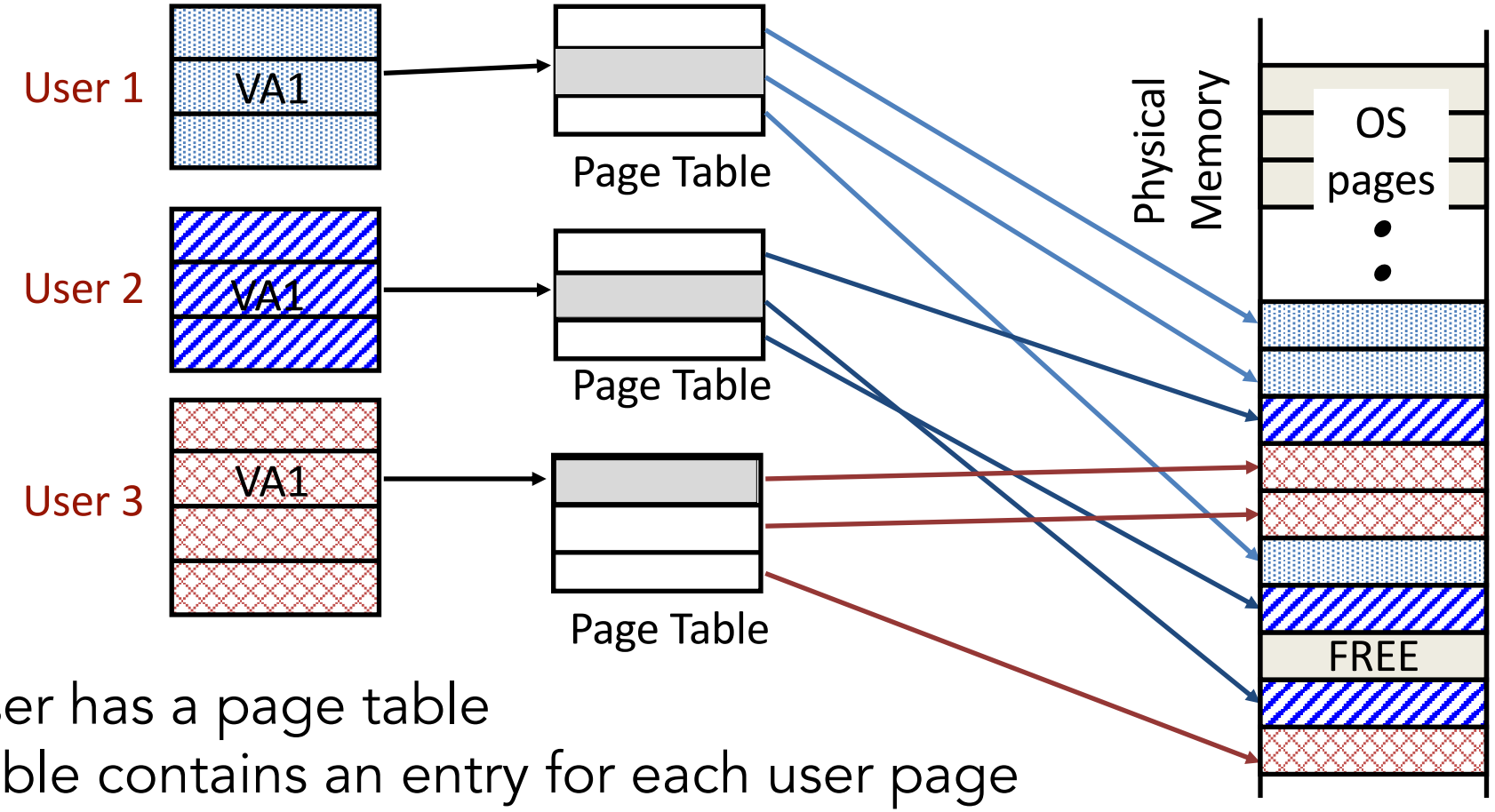
# Paged Memory Systems

- Processor generated address can be interpreted as a pair <page number,offset>


- A page table contains the physical address of the base of each page



# Private Address Space per User

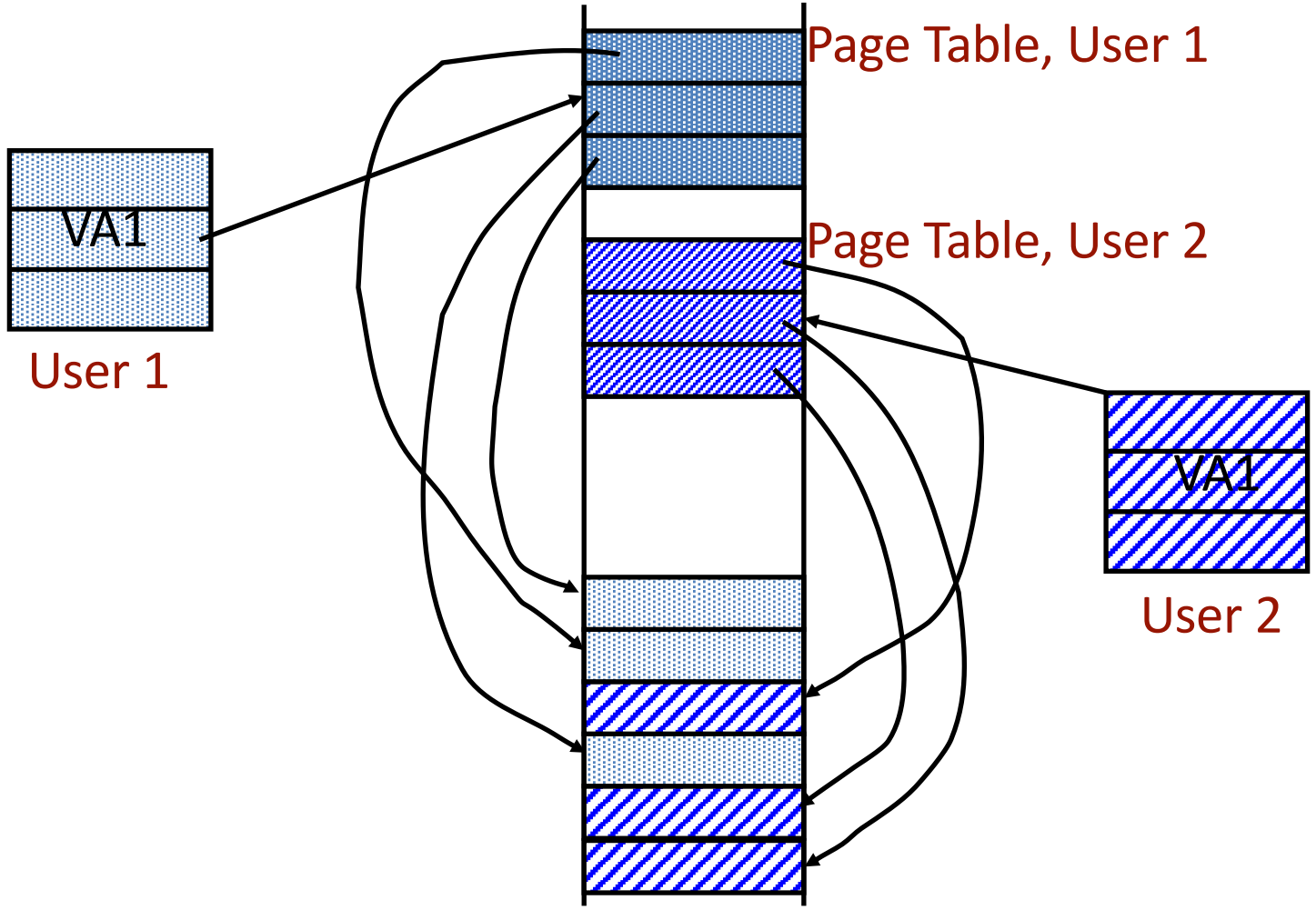



- Each user has a page table
- Page table contains an entry for each user page

# Where Should Page Tables Reside?

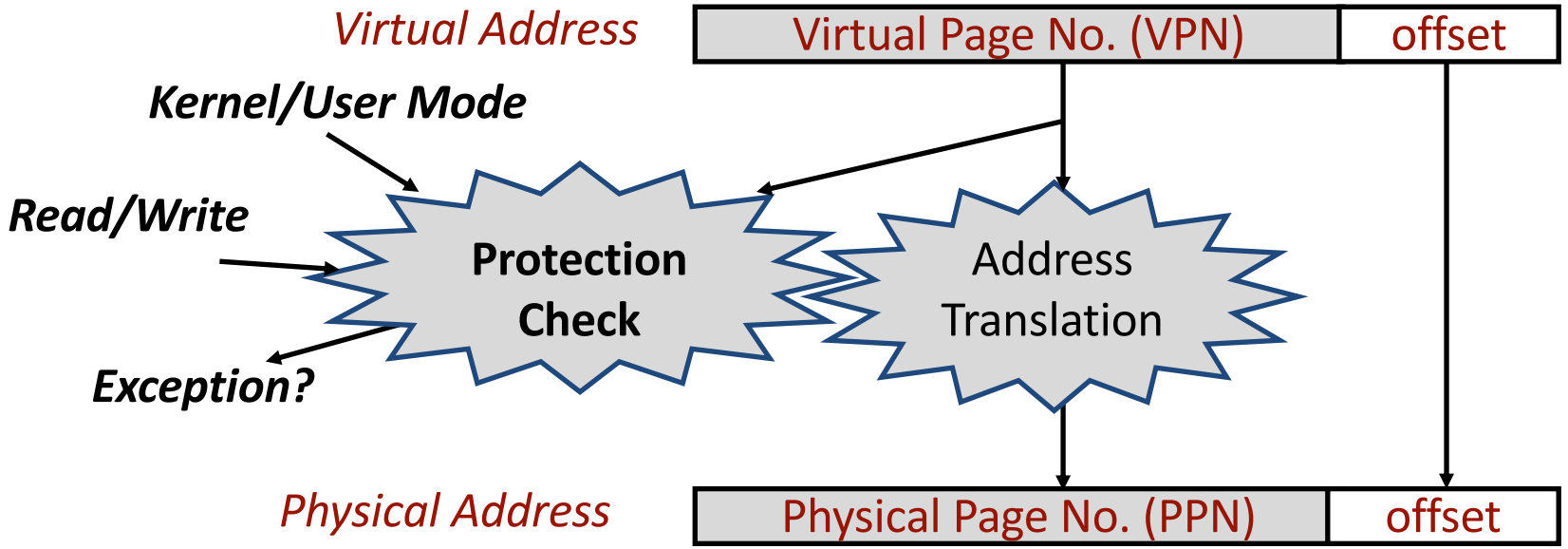
- Space required by the page tables is proportional to the address space, number of users, ...
  - Space requirement is large too expensive to keep in registers
- Special registers just for the current user:
  - What disadvantages does this have?
    - may not be feasible for large page tables
- Main memory:
  - Needs one reference to retrieve the page base address and another to access the data word
    - doubles number of memory references!

# Page Tables in Physical Memory



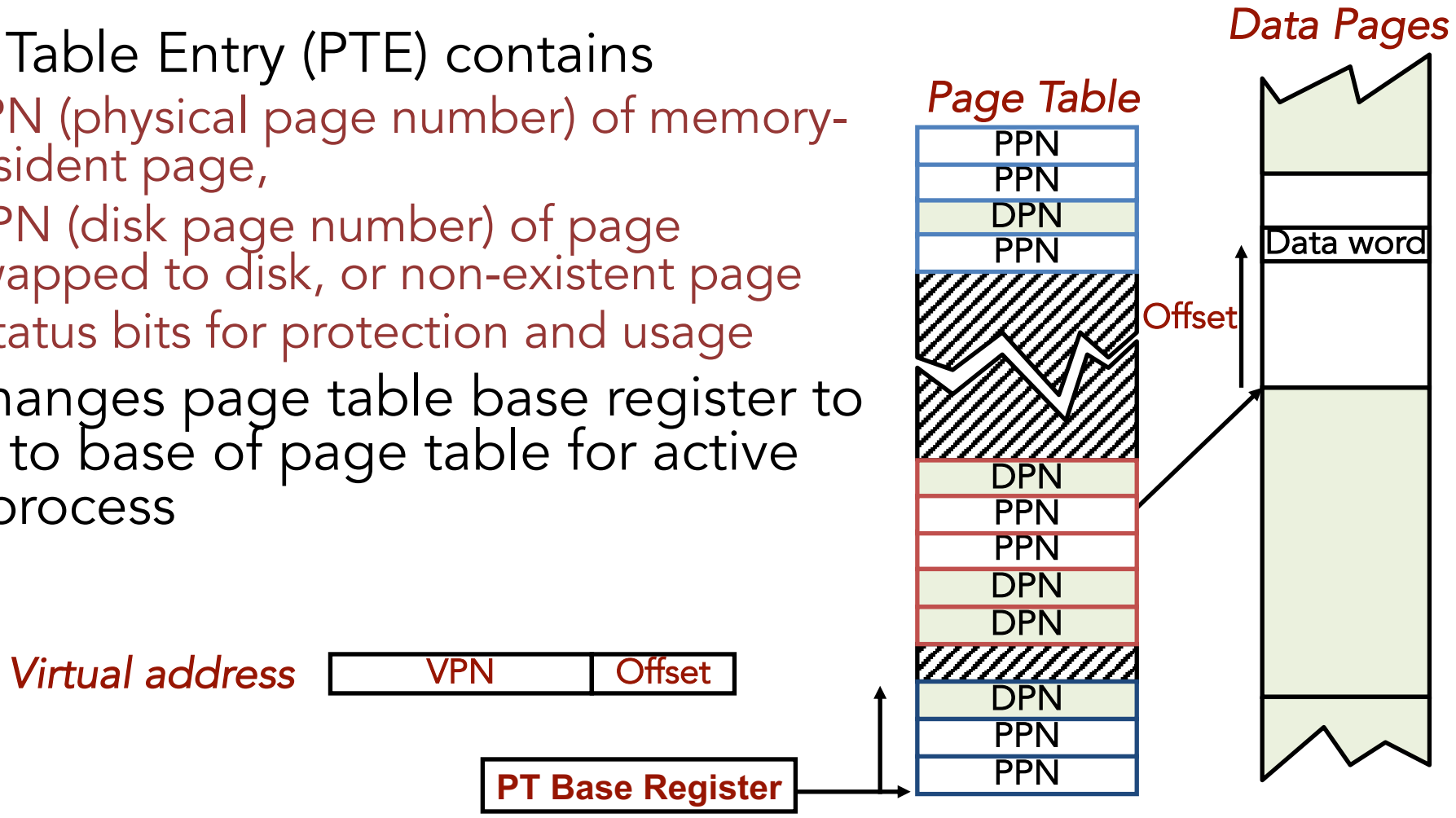
# Address Translation and Protection

- Every instruction and data access needs address translation and protection checks
- A good VM design needs to be fast (~ one cycle) and space efficient



# Linear Page Table

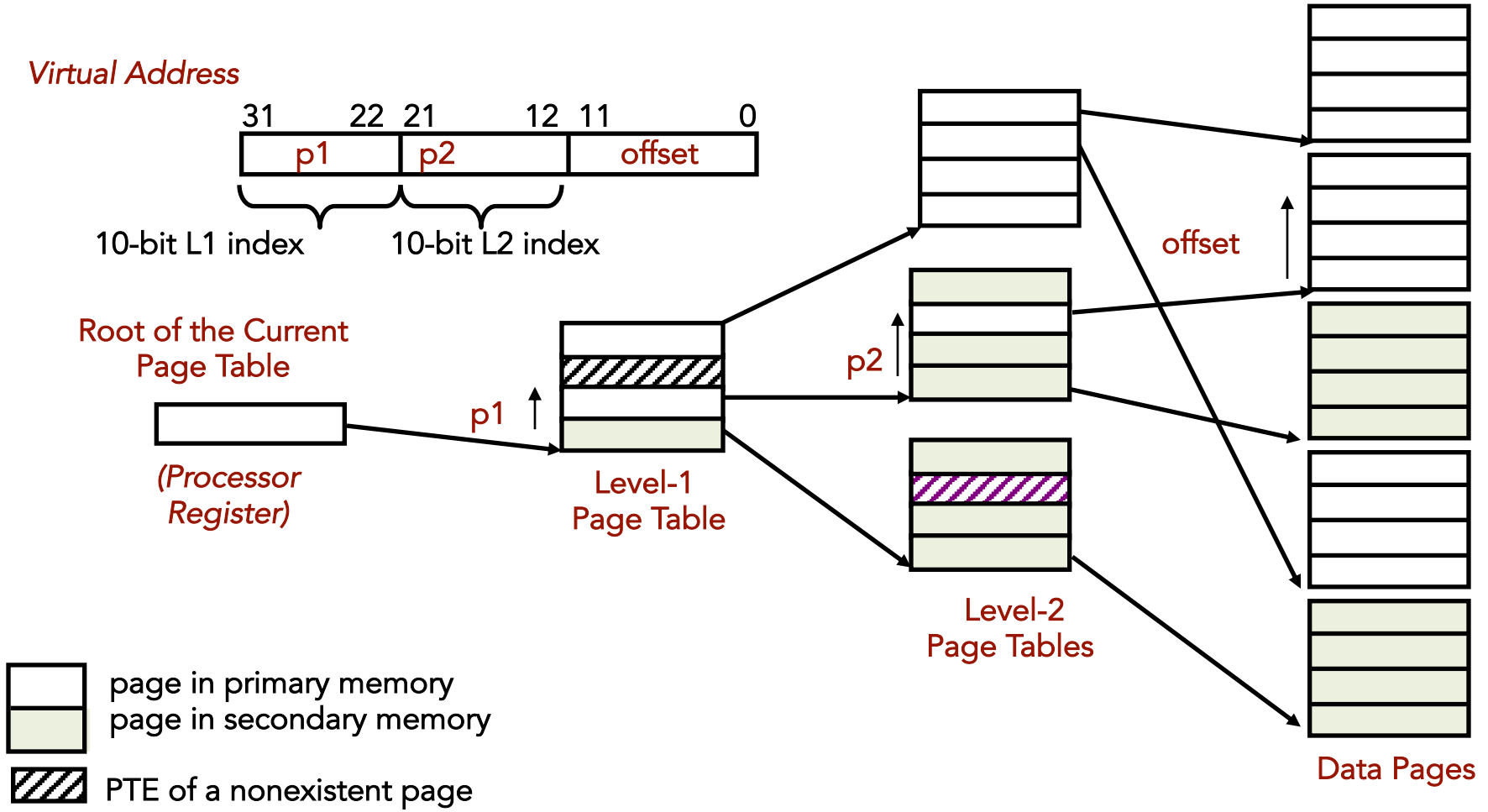
- Page Table Entry (PTE) contains
  - PPN (physical page number) of memory-resident page,
  - DPN (disk page number) of page swapped to disk, or non-existent page
  - Status bits for protection and usage
- OS changes page table base register to point to base of page table for active user process



# Size of Linear Page Table

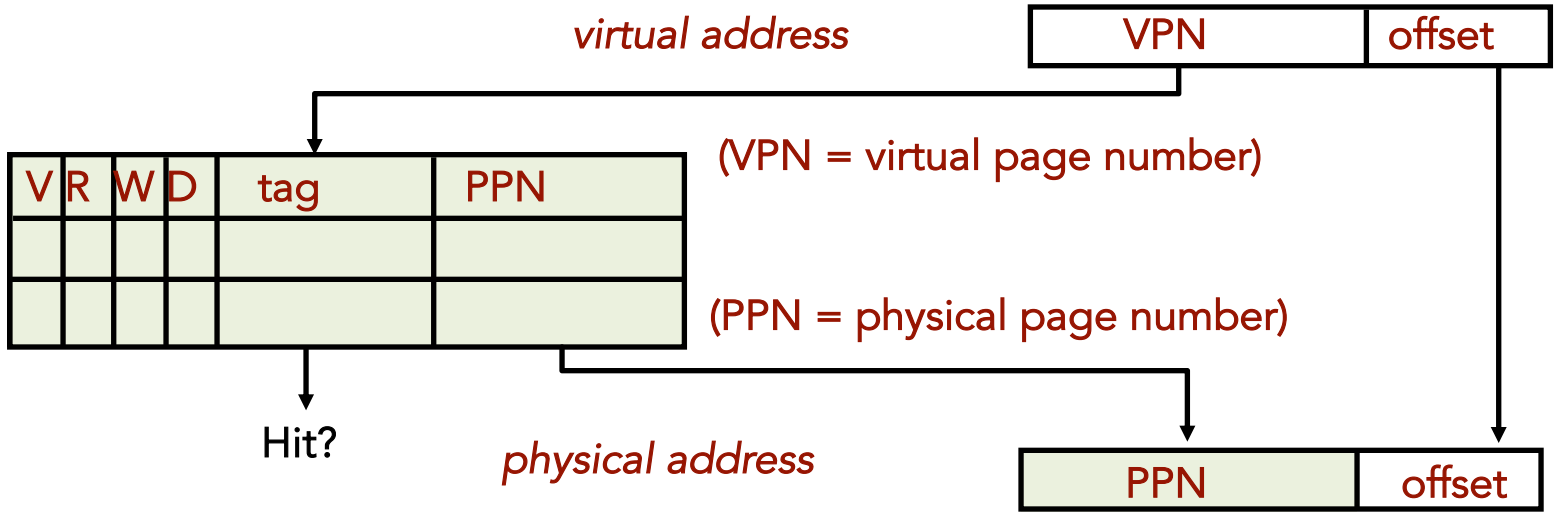
- With 32-bit addresses, 4-KB pages, and 4-byte PTEs:
  - $2^{20}$ PTEs, i.e, 4 MB page table per user
  - 4 GB of swap needed to back up full virtual address space
- Larger pages?
  - More internal fragmentation (don't use all memory in page)
  - Larger page fault penalty (more time to read from disk)
- What about 64-bit virtual address space???
  - Even 1MB pages would require $2^{44}$ 8-byte PTEs (35 TB!)

# Hierarchical Page Table



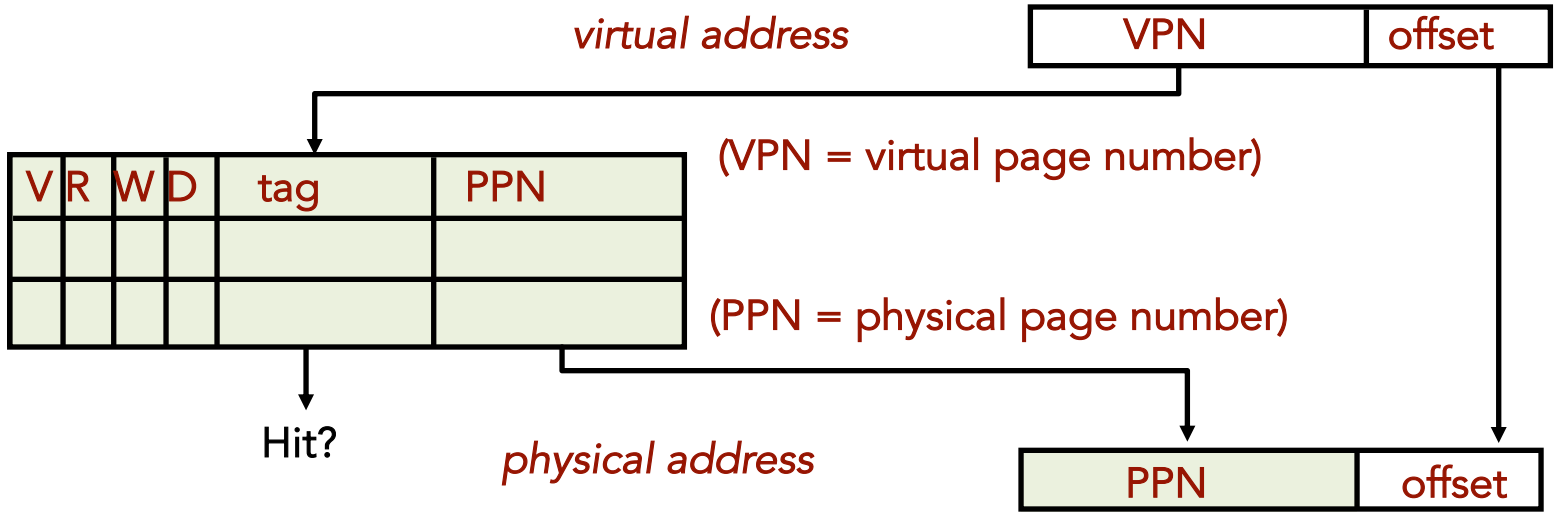
# Translation Lookaside Buffers

- Address translation is very expensive!
  - In a two-level page table, each reference becomes
    - Best case is _____ ?
    - Worst case is _____ ?
- Solution: Cache translations in TLB
  - TLB hit → Single Cycle Translation
  - TLB miss → Page Table Walk to refill



# Translation Lookaside Buffers

- Address translation is very expensive!
  - In a two-level page table, each reference becomes
    - Best case is ___3 memory references __ ?
    - Worst case is _____2-page faults___ ?
- Solution: Cache translations in TLB
  - TLB hit → Single Cycle Translation
  - TLB miss → Page Table Walk to refill



# TLB Designs

- Typically, 32-128 entries
- Usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages  more likely that two entries conflict
  - Sometimes larger TLBs are 4-8 way set-associative
- Random or FIFO replacement policy
  - Typically, only one page mapping per entry
  - No process information in TLB
- 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 = _____?

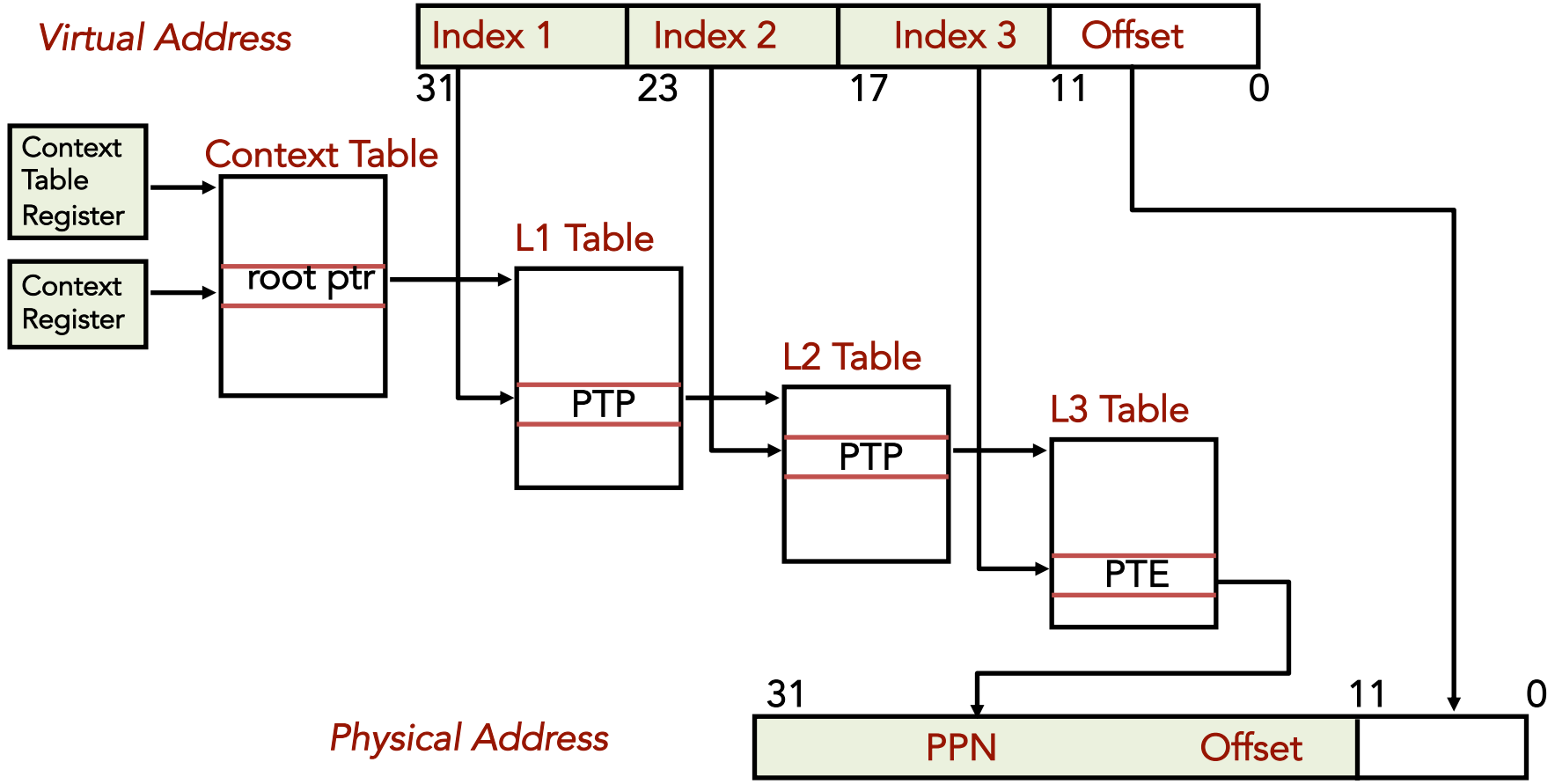
# TLB Designs

- Typically, 32-128 entries
- Usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages  more likely that two entries conflict
  - Sometimes larger TLBs are 4-8 way set-associative
- Random or FIFO replacement policy
  - Typically, only one page mapping per entry
  - No process information in TLB
- 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 = _64 entries * 4 KB = 256 KB (if contiguous)__?

# Handling A TLB Miss

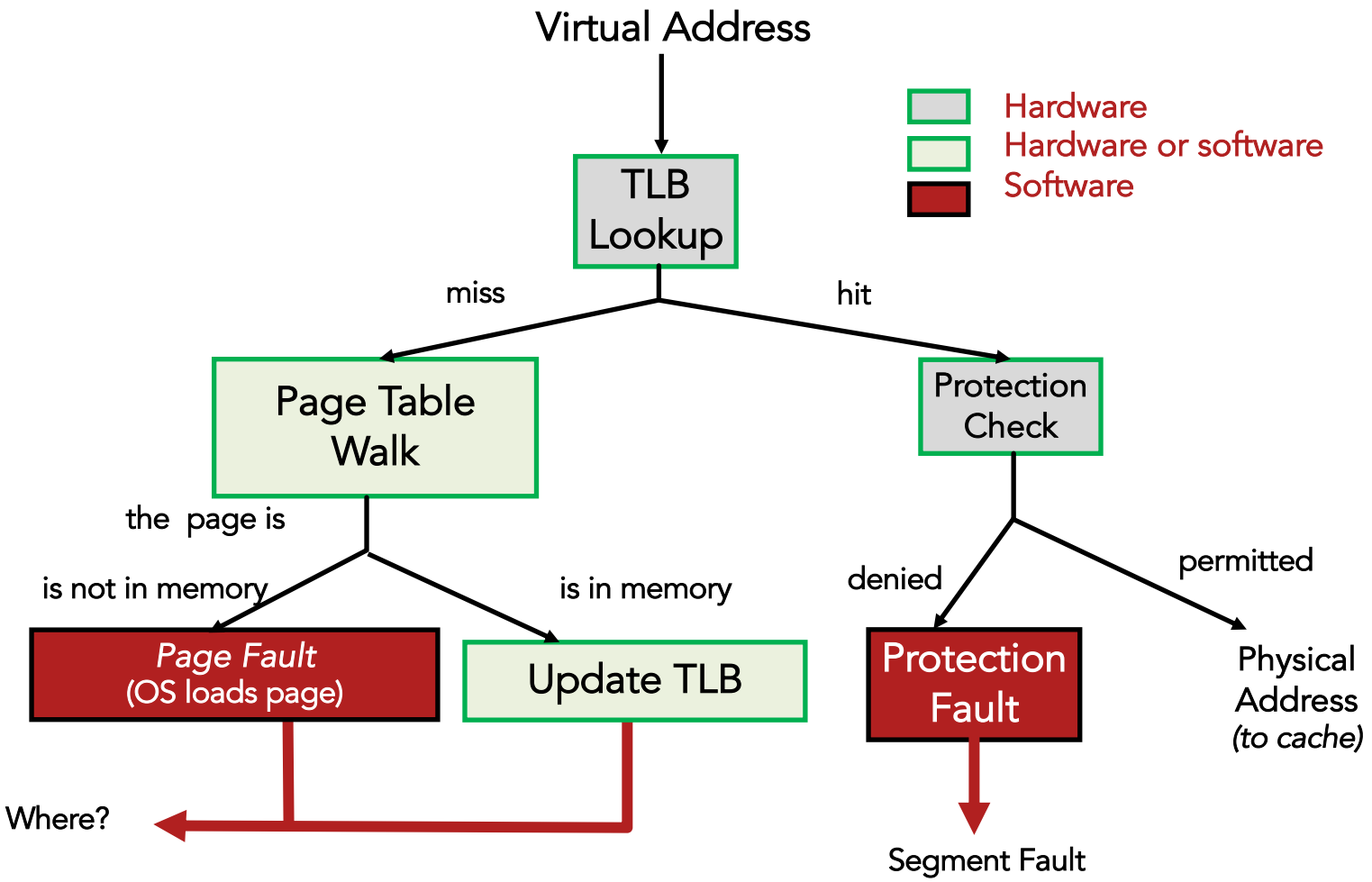
- Software (MIPS, Alpha)
  - TLB miss causes an exception and the operating system walks the page tables and reloads TLB privileged "untranslated" addressing mode used for walk
- Hardware (SPARC v8, x86, PowerPC)
  - A memory management unit (MMU) walks the page tables and reloads the TLB
  - If a missing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

# Hierarchical Page Table Walk: SPARC v8




- MMU does this table walk in hardware on a TLB miss

# Address Translation

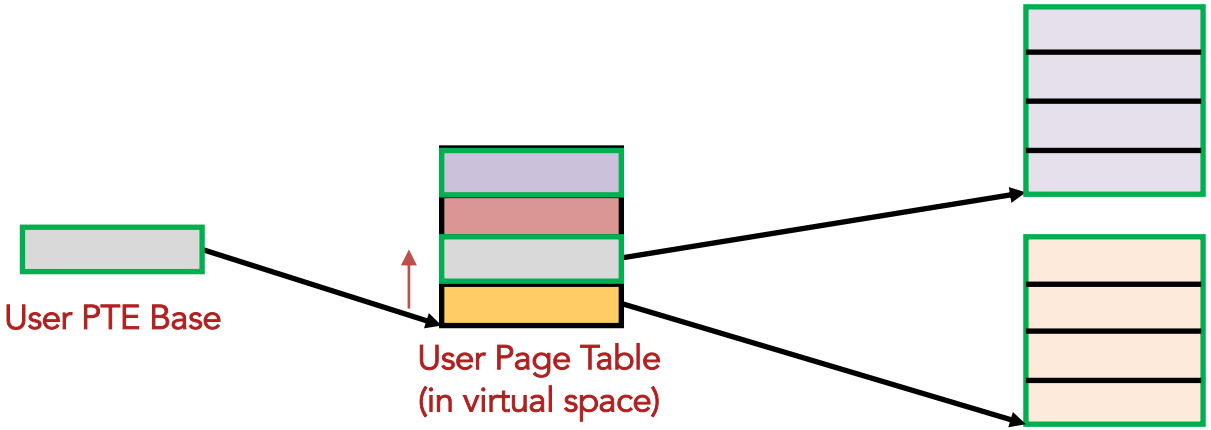


# Page Fault Handler

- When the referenced page is not in DRAM:
  - The missing page is located (or created)
  - It is brought in from disk, and page table is updated
    - Another job may be run on the CPU while the first job waits for the requested page to be read from disk
  - If no free pages are left, a page is swapped out
    - Approximate LRU replacement policy
- Since it takes a long time to transfer a page (msecs), page faults are handled completely in software by the OS
  - Untranslated addressing mode is essential to allow kernel to access page tables

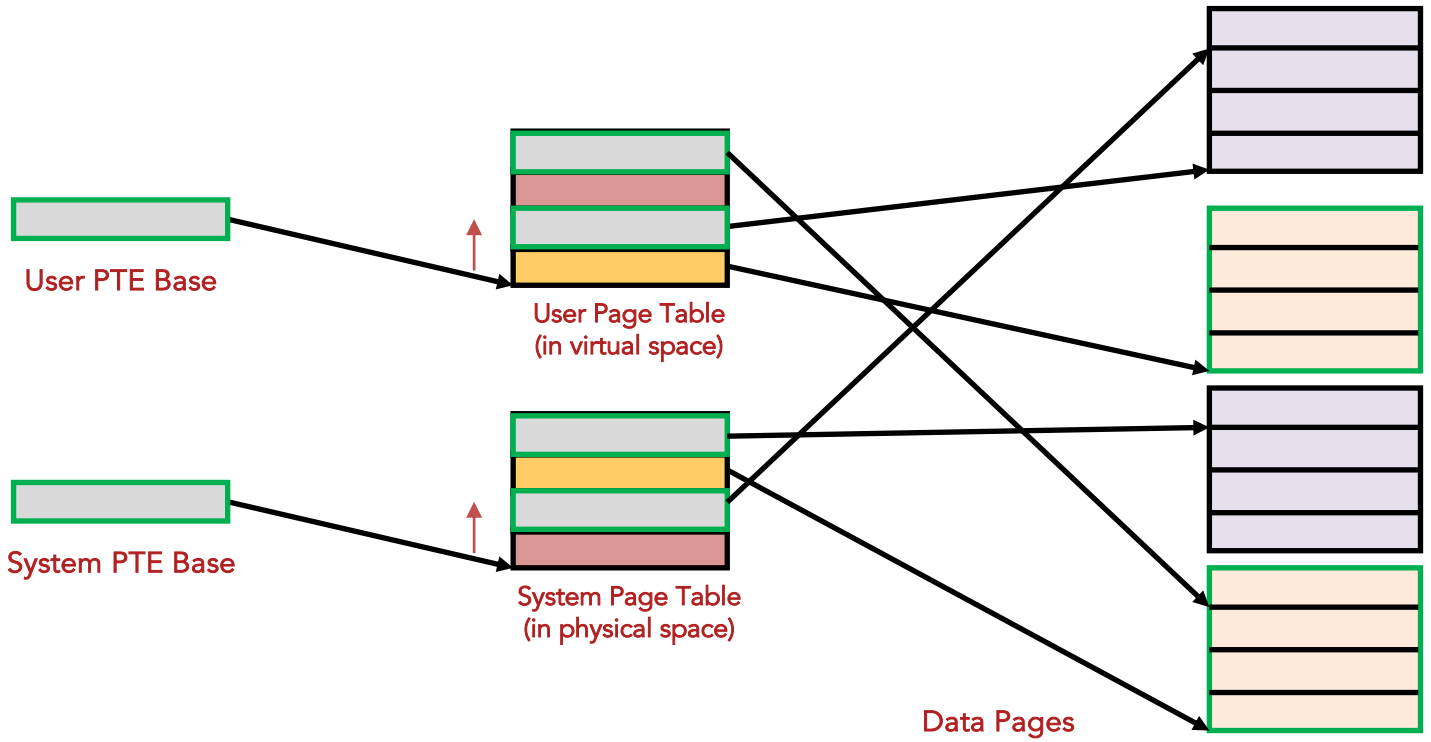
# Translation for Page Tables

- Can references to page tables cause TLB misses?




- User VA translation causes a TLB miss
- Page table walk: User PTE Base and appropriate bits from VA are used to obtain virtual address VP for page table entry
- Get a TLB miss when we try to translate VP

# Translation for Page Tables

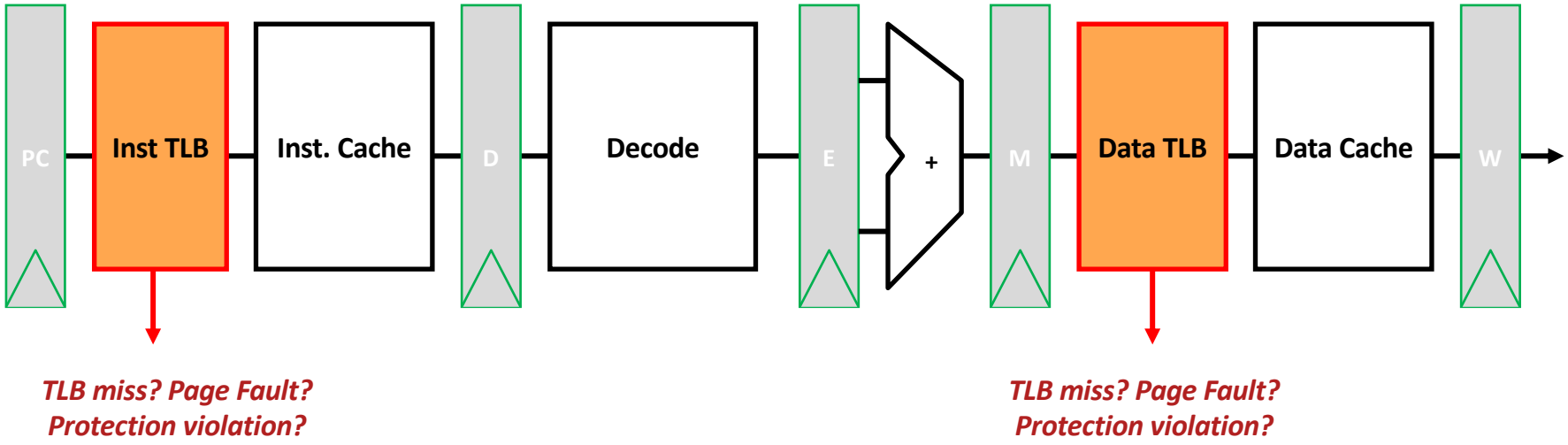



- When we get a TLB miss on VP translation, OS adds System PTE Base to bits from VP to find physical address of page table entry for VP

# Swapping a Page of a Page Table

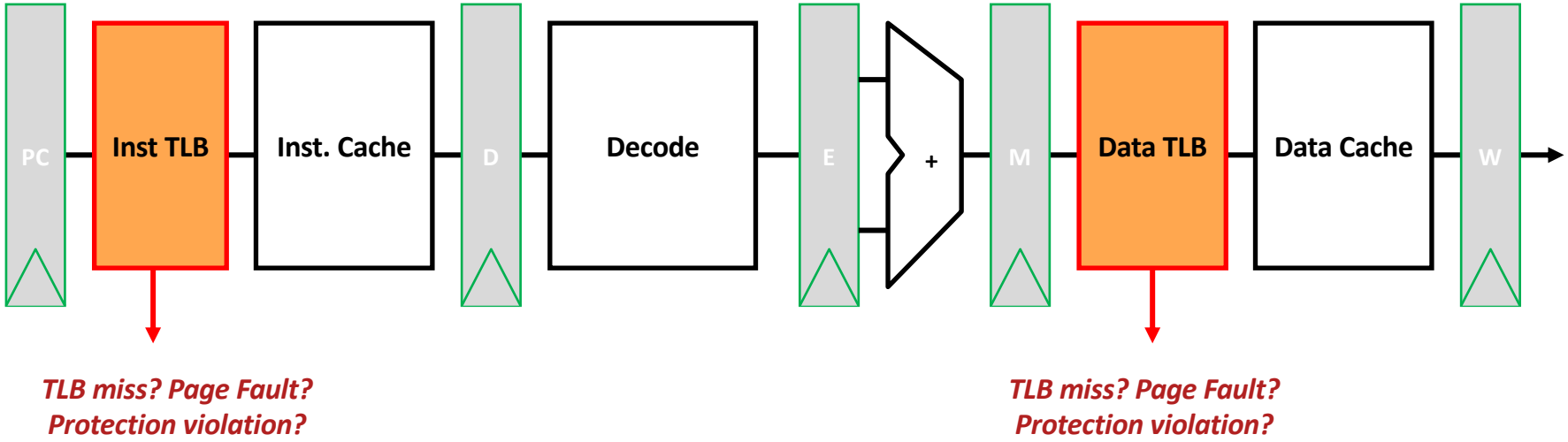
- A PTE in primary memory contains primary or secondary memory addresses
- A PTE in secondary memory contains only secondary memory addresses
- A page of a PT can be swapped out only if none its PTE's point to pages in the primary memory

# Address Translation in CPU Pipeline



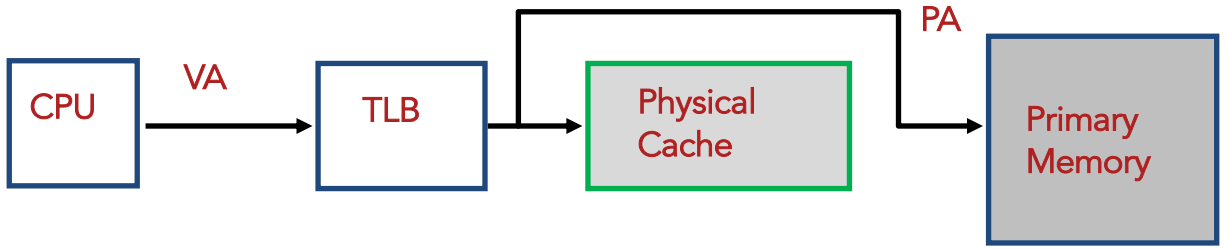

- Software handlers need a restartable exception on page fault or protection violation
- Handling a TLB miss needs a hardware or software mechanism to refill TLB

# Address Translation in CPU Pipeline

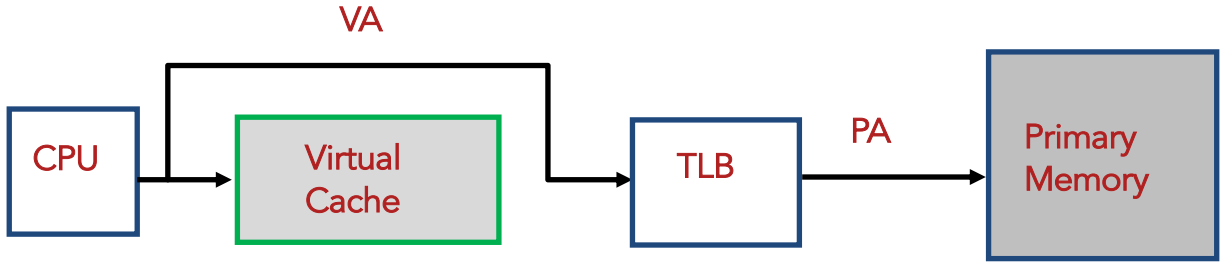



- Need mechanisms to cope with the additional latency of a TLB
  - Slow down the clock
  - Pipeline the TLB and cache access
  - Virtual address caches
  - Parallel TLB/cache access

# Physical or Virtual Address Caches?



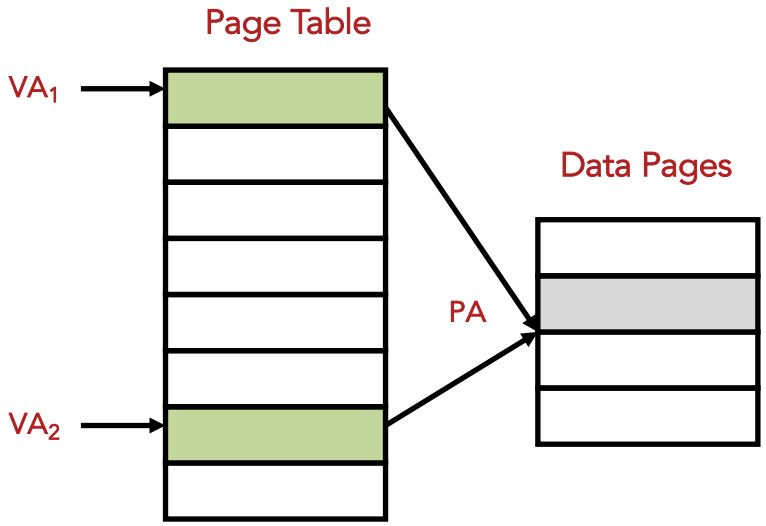

*Alternative: place the cache before the TLB*



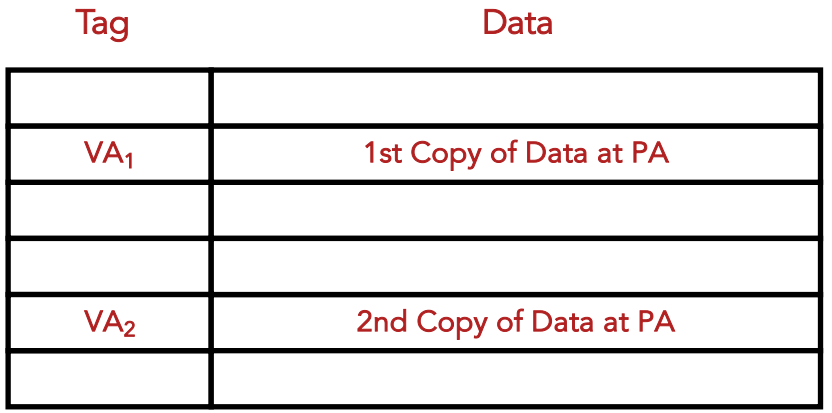

*(StrongARM)*

- One-step process in case of a hit (+)
- Cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- Aliasing problems due to the sharing of pages (-)

# Aliasing in Virtual-Address Caches



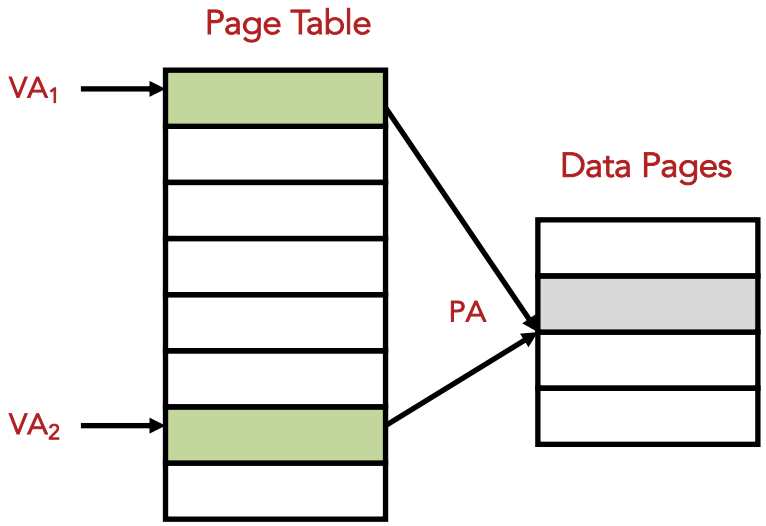

Two virtual pages share one physical page



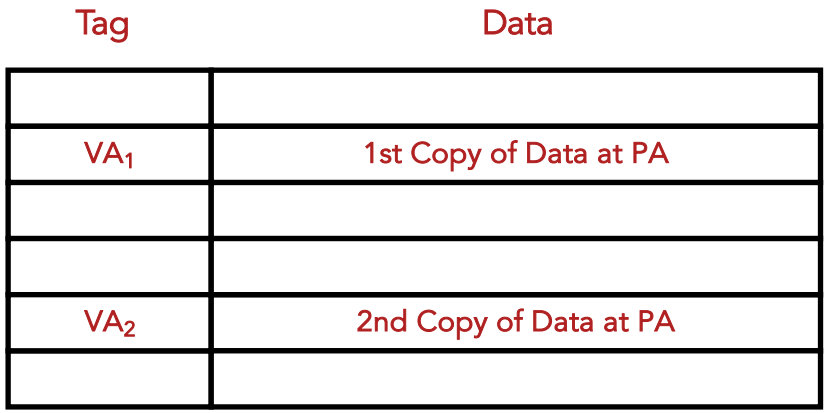

Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

- General Solution: Disallow aliases to coexist in cache

# Aliasing in Virtual-Address Caches



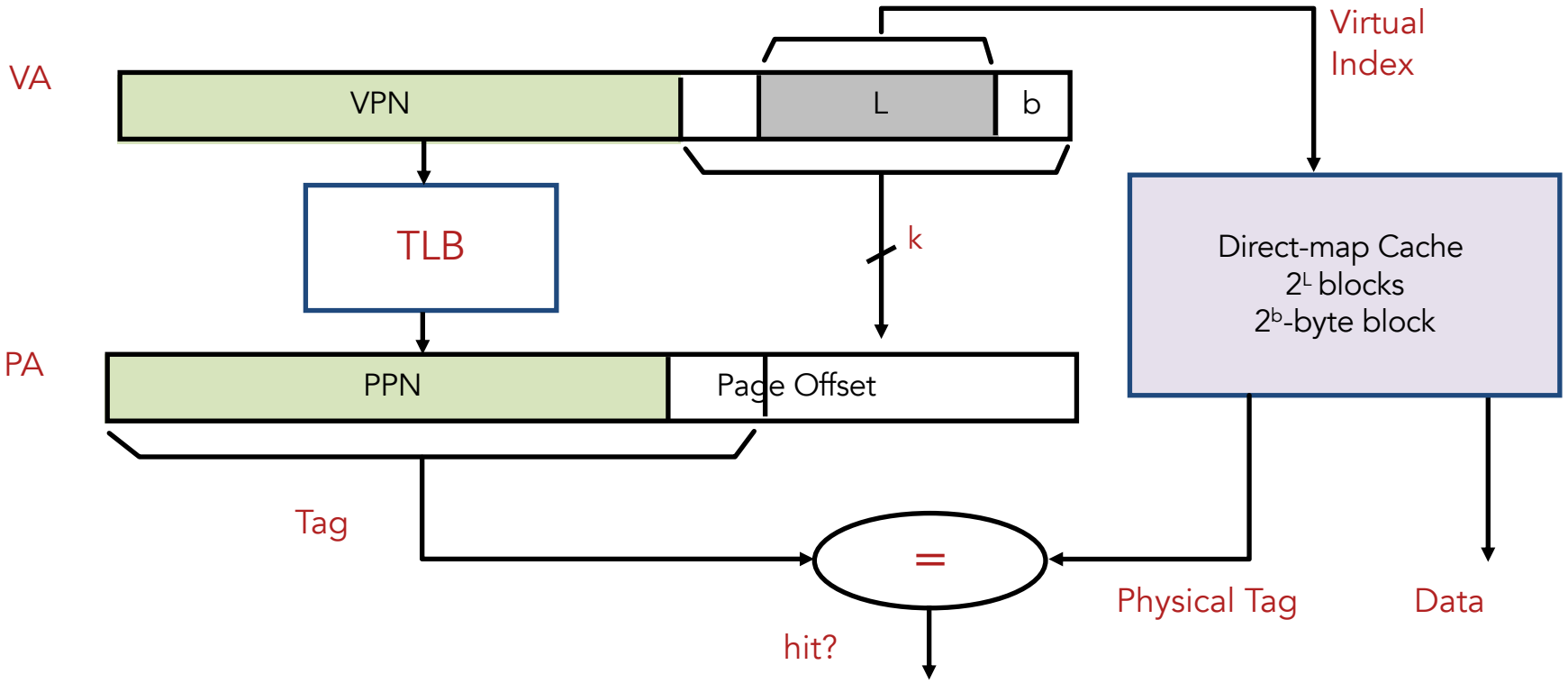

Two virtual pages share one physical page




Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

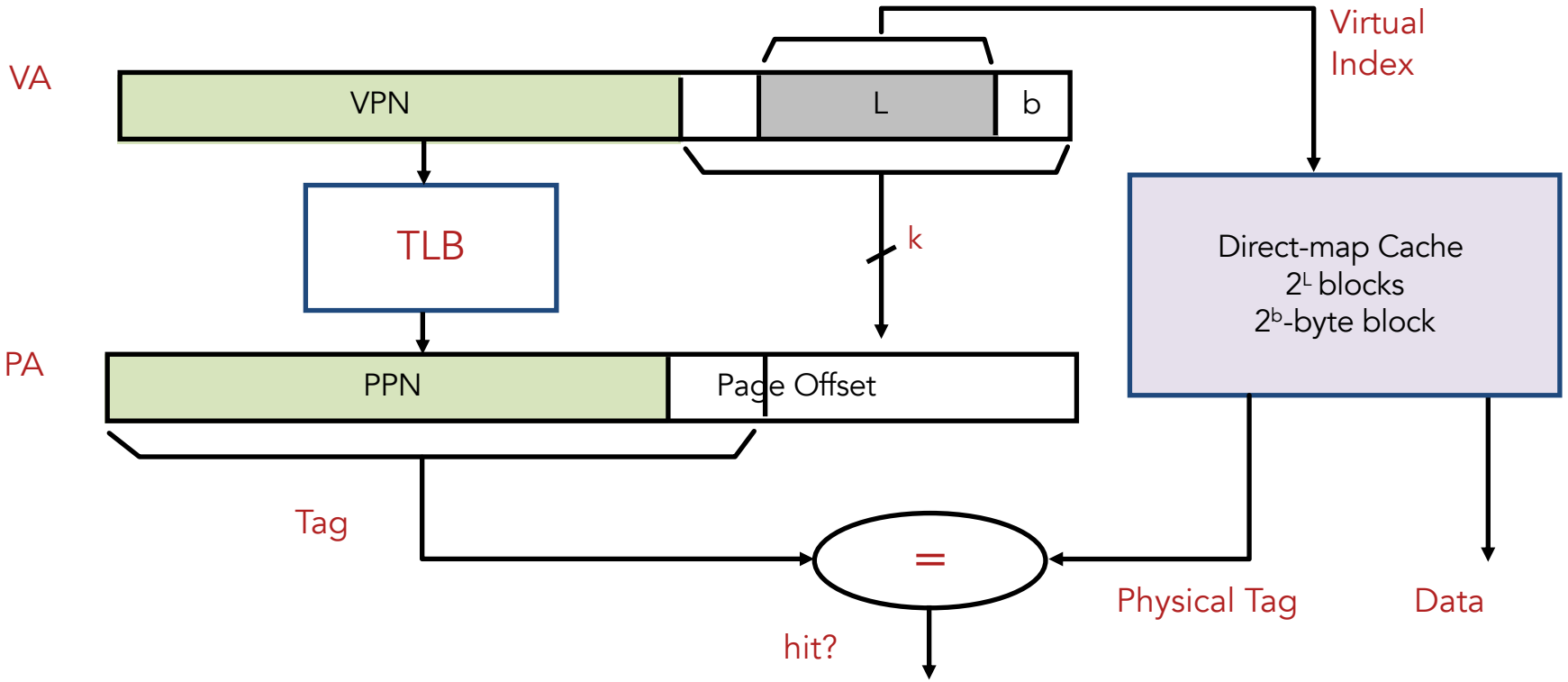
- Software (i.e., OS) solution for direct-mapped cache
  - VAs of shared pages must agree in cache index bits; this ensures all VAs accessing same PA will conflict in direct-mapped cache (early SPARCs)

# Concurrent Access to TLB & Cache



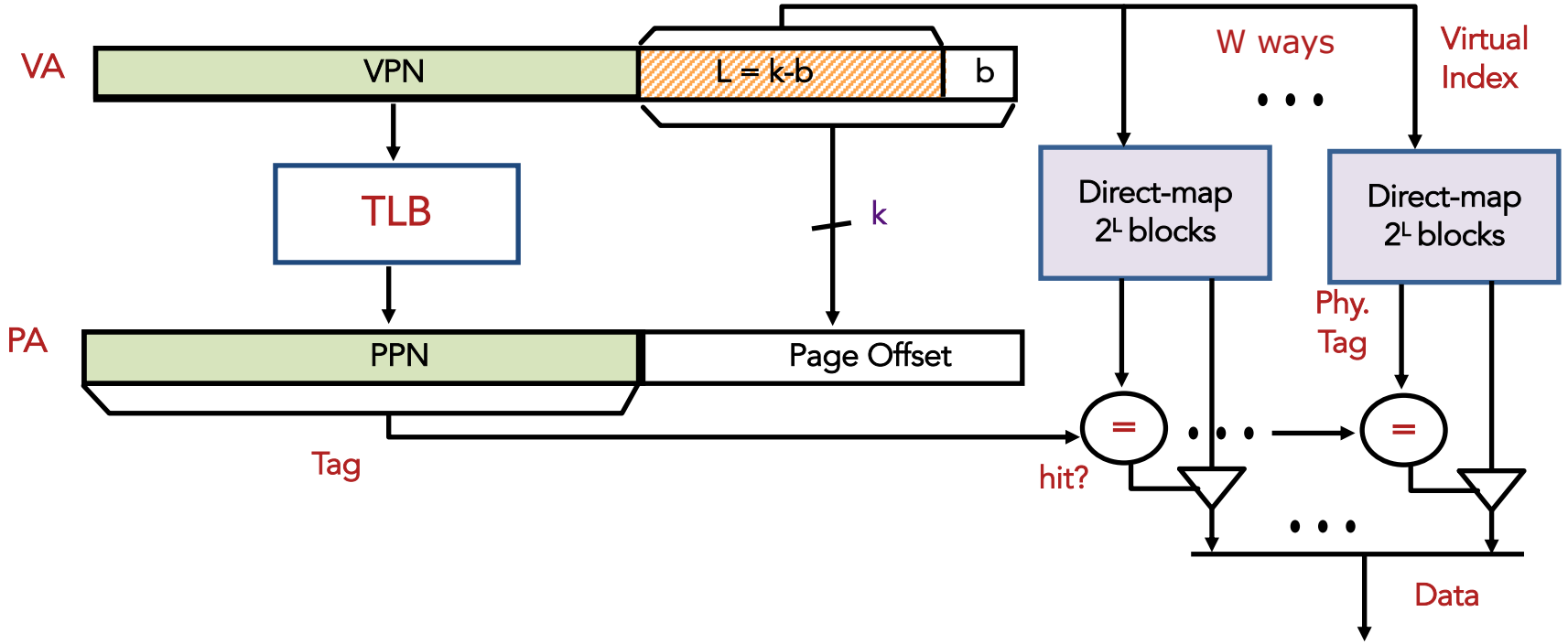

- Index L is available without consulting the TLB
  - Cache and TLB accesses can begin simultaneously

# Concurrent Access to TLB & Cache



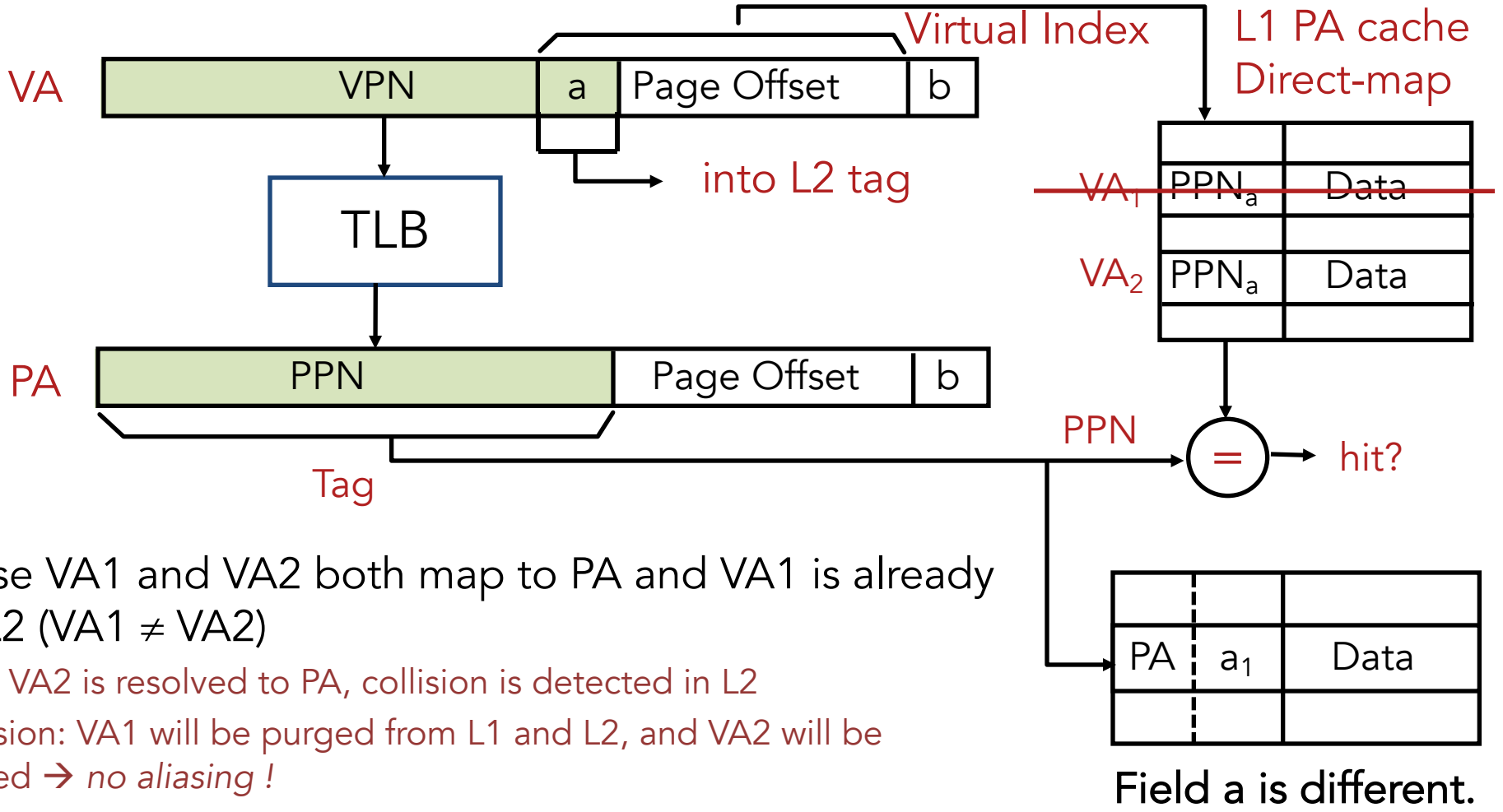

- Tag comparison is made after both accesses are completed
  - Cases: L + b = k  L + b < k
    L + b > k what happens here?
  - L + b > k: Partially VA cache. But it may be more effective to increase the way of the cache and decrease the index bits for same cache capacity

# Virtual-Index Physical-Tag Caches



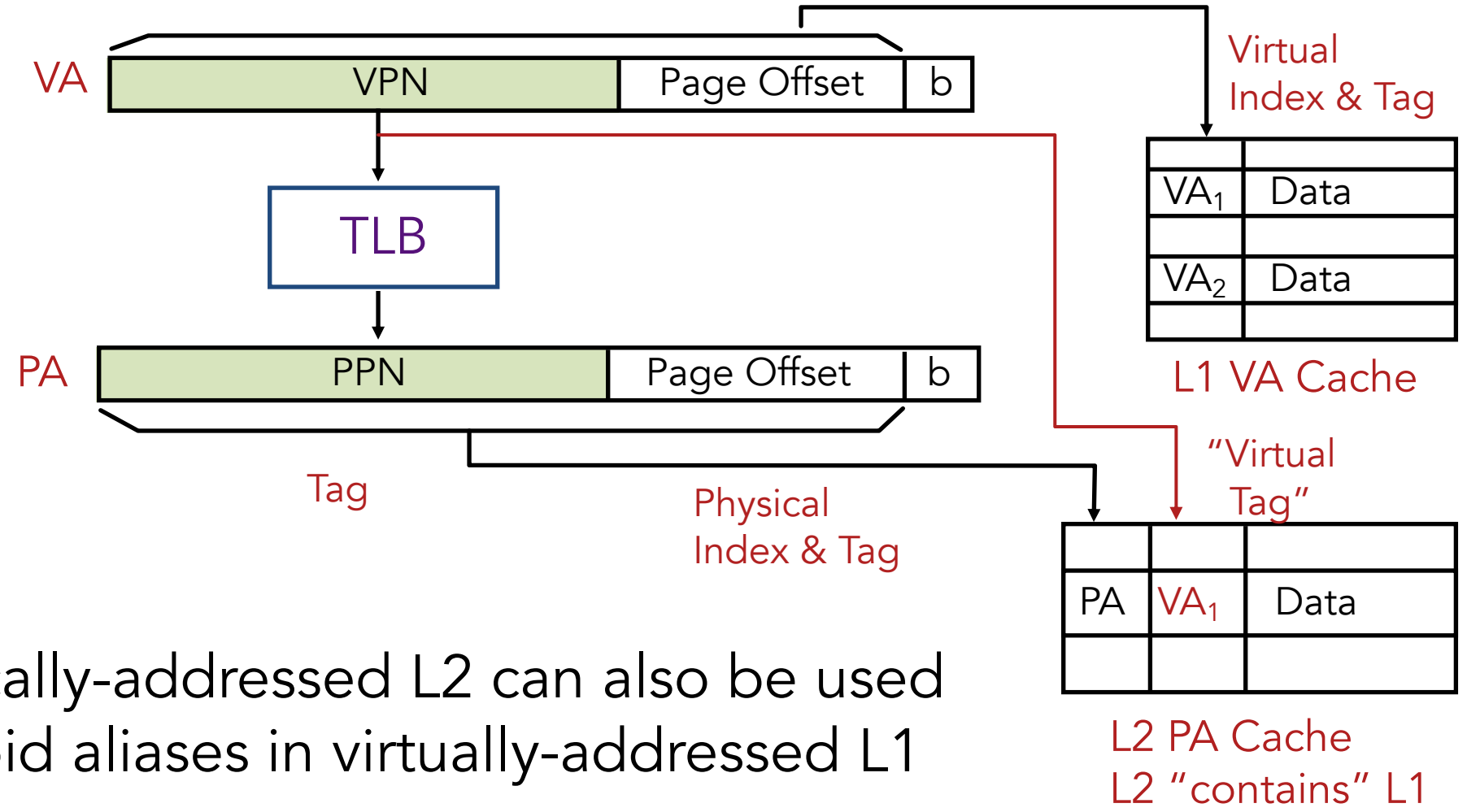

- After the PPN is known, W physical tags are compared
- Allows cache size to be greater than $2^{L+b}$ bytes

# Anti-Aliasing Using L2




- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 ≠ VA2)
  - After VA2 is resolved to PA, collision is detected in L2
  - Collision: VA1 will be purged from L1 and L2, and VA2 will be loaded → *no aliasing !*

# Virtually-Addressed L1




- Physically-addressed L2 can also be used to avoid aliases in virtually-addressed L1

# Virtual Memory Use Today

- Desktops/servers have full demand-paged virtual memory
  - Portability between machines with different memory sizes
  - Protection between multiple users or multiple tasks
  - Share small physical memory among active tasks
  - Simplifies implementation of some OS features

# Virtual Memory Use Today

- Vector supercomputers have translation and protection but not demand-paging (Older Crays: base&bound, Japanese & Cray X1: pages)
  - Do not waste expensive CPU time thrashing to disk (make jobs fit in memory)
  - Mostly run in batch mode (run set of jobs that fits in memory)
  - Difficult to implement restartable vector instructions

# Virtual Memory Use Today

- Most embedded processors and DSPs provide physical addressing only
  - Cannot afford area/speed/power budget for virtual memory support
  - Often there is no secondary storage to swap to!
  - Programs custom written for particular memory configuration in product
  - Difficult to implement restartable instructions for exposed architectures

# Next Learning Module

- Cache Coherence