Modern Computer Architecture

Lecture5 Virtual Memory

Hongbin Sun

国家集成电路人才培养基地

Xi'an Jiaotong University

Recap: Memory Hierarchy

- **DRAM is dominant form of main memory today**
  - **Holds values on small capacitors, requires refresh, and long time to sense bit values (destructive reads)**
  - **Row access brings internal row into sense amps, column access reads out bits from sense amps.**
  - **Most DRAM interface innovations improve bandwidth of column accesses across chip pins, not access latency**
  - **Individual chips packaged in ranks on DIMMs, multiple independent banks/rank**
- **Many forms of cache optimization, targeting: hit time, miss rate, miss penalty**
  - **Increasing number of levels of cache, and sophisticated prefetching schemes - difficult to tune code for hierarchy**

Memory Management

- **From early absolute adressing schemes, to modern virtual memory systems with support for virtual machine monitors**

- **Can separate into orthogonal functions:**
  - **Translation (mapping of virtual address to physical address)**
  - **Protection (permission to access word in memory)**
  - **Virtual memory (transparent extension of memory space using slower disk storage)**
- **But most modern systems merge support for above functions with a common page-based system**

Absolute Addresses

EDSAC, early 50's

- **Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)**
- **Addresses in a program depended upon where the program was to be loaded in memory**
- ***But* it was more convenient for programmers to write location-independent subroutines**

How could location independence be achieved?

Linker and/or loader modify addresses of subroutines and callers when building a program memory image

Dynamic Address Translation

Motivation

In the early machines, I/O operations were slow and each word transferred involved the CPU

Higher throughput if CPU and I/O of 2 or more programs were overlapped.

How?⇒ multiprogramming

Location-independent programs

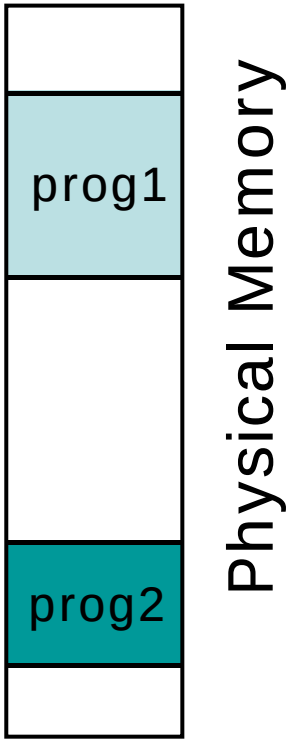
Programming and storage management ease

⇒ need for a *base register*

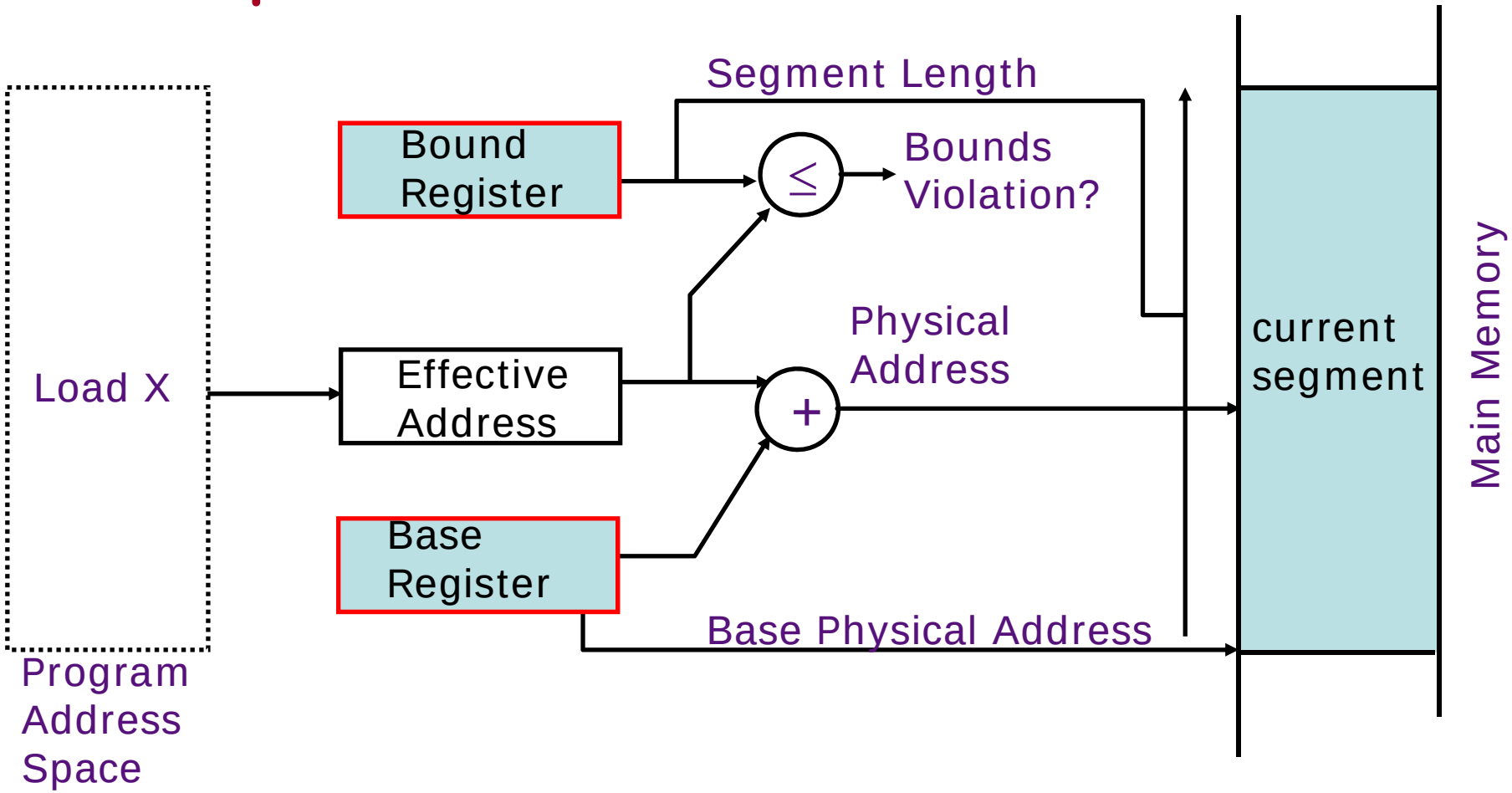
Protection

Independent programs should not affect each other inadvertently

⇒ need for a *bound register*

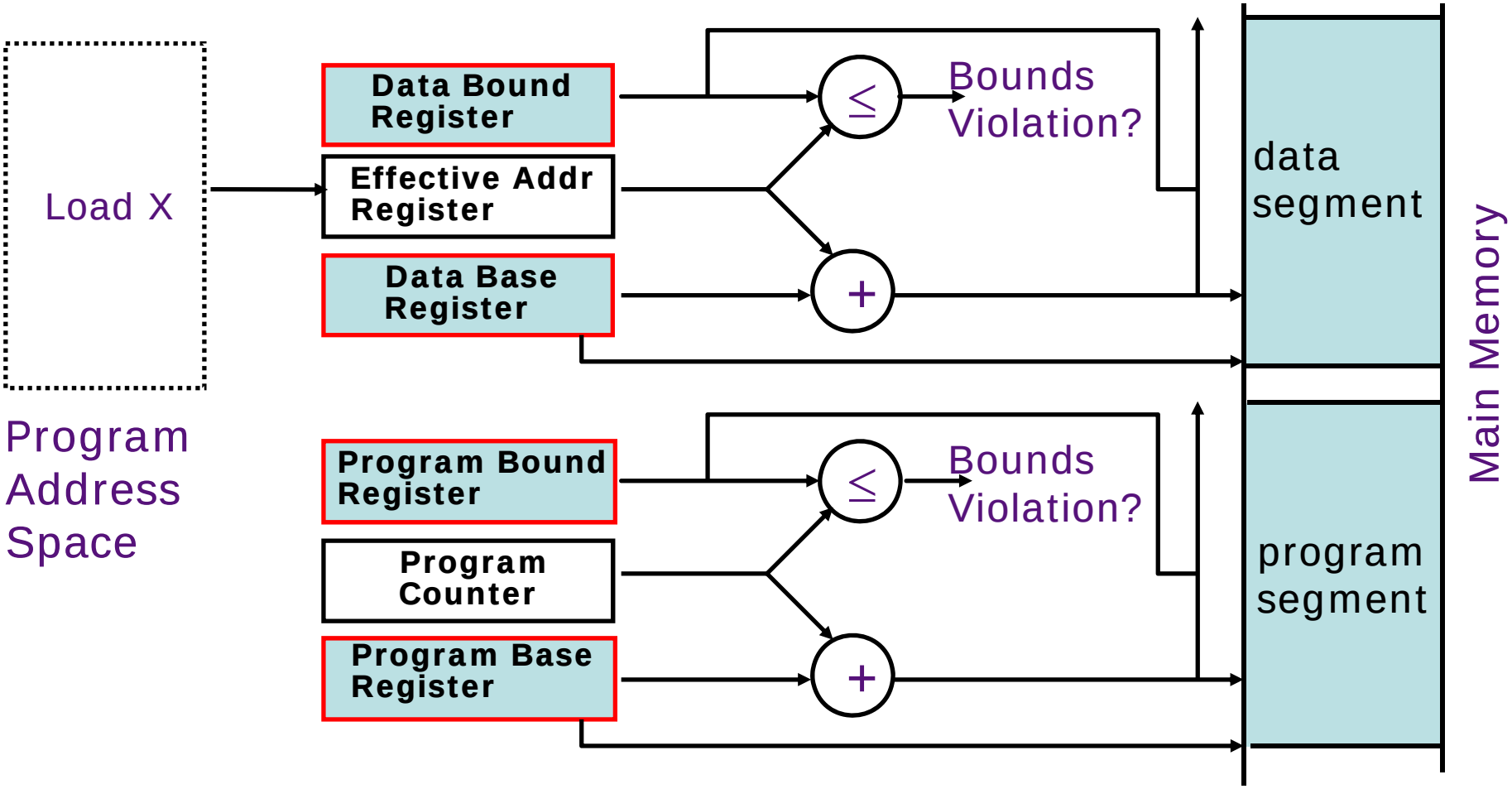


Simple Base and Bound Translation



Base and bounds registers are visible/accessible only when processor is running in the *supervisor mode*

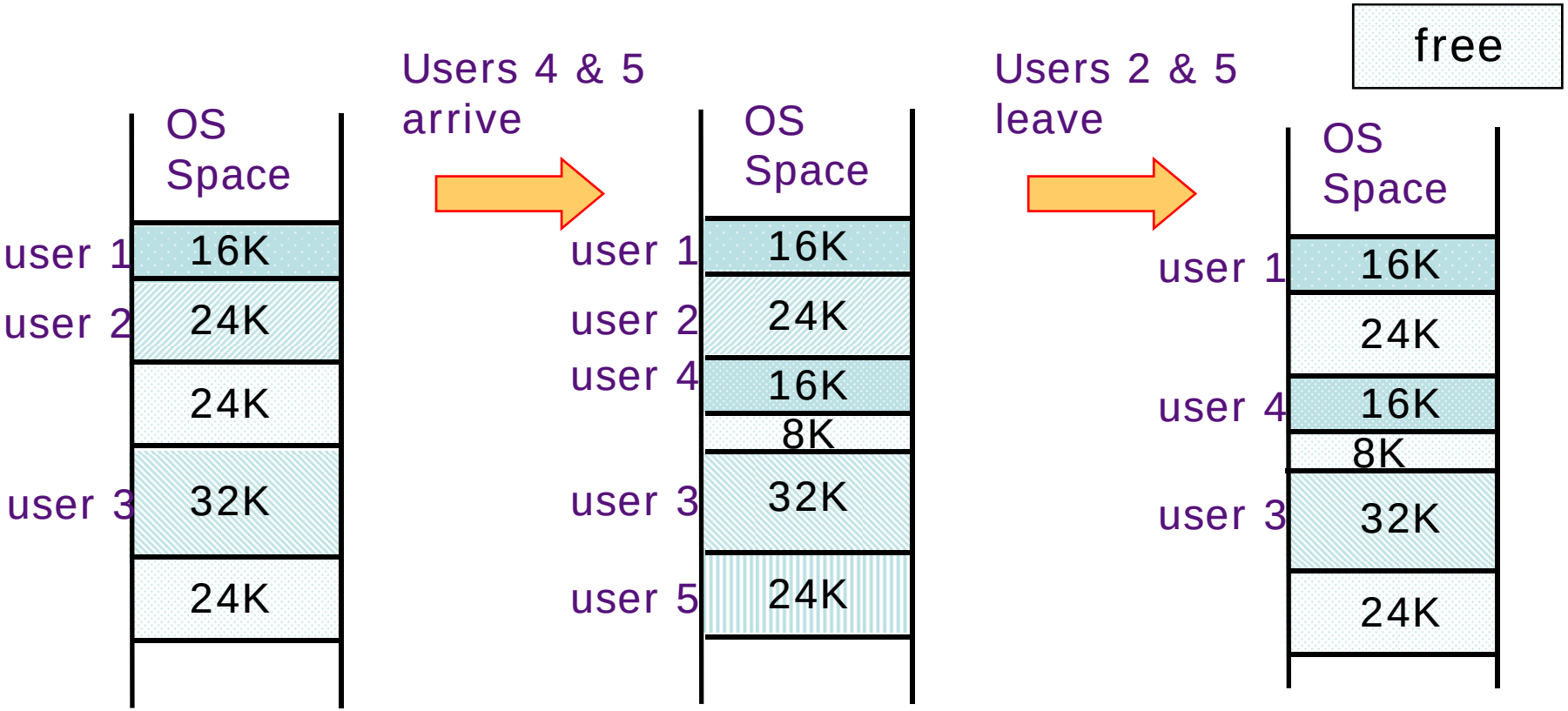
Separate Areas for Program and Data



What is an advantage of this separation?

(Scheme used on all Cray vector supercomputers prior to X1, 2002)

Memory Fragmentation

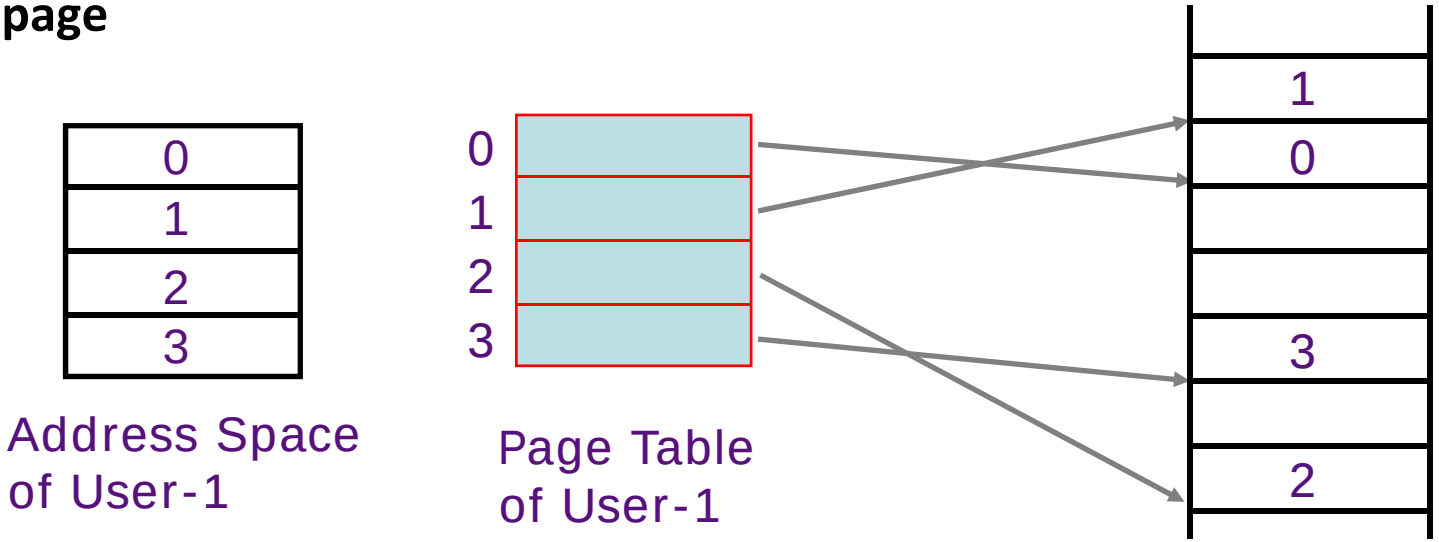


As users come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage.

Paged Memory Systems

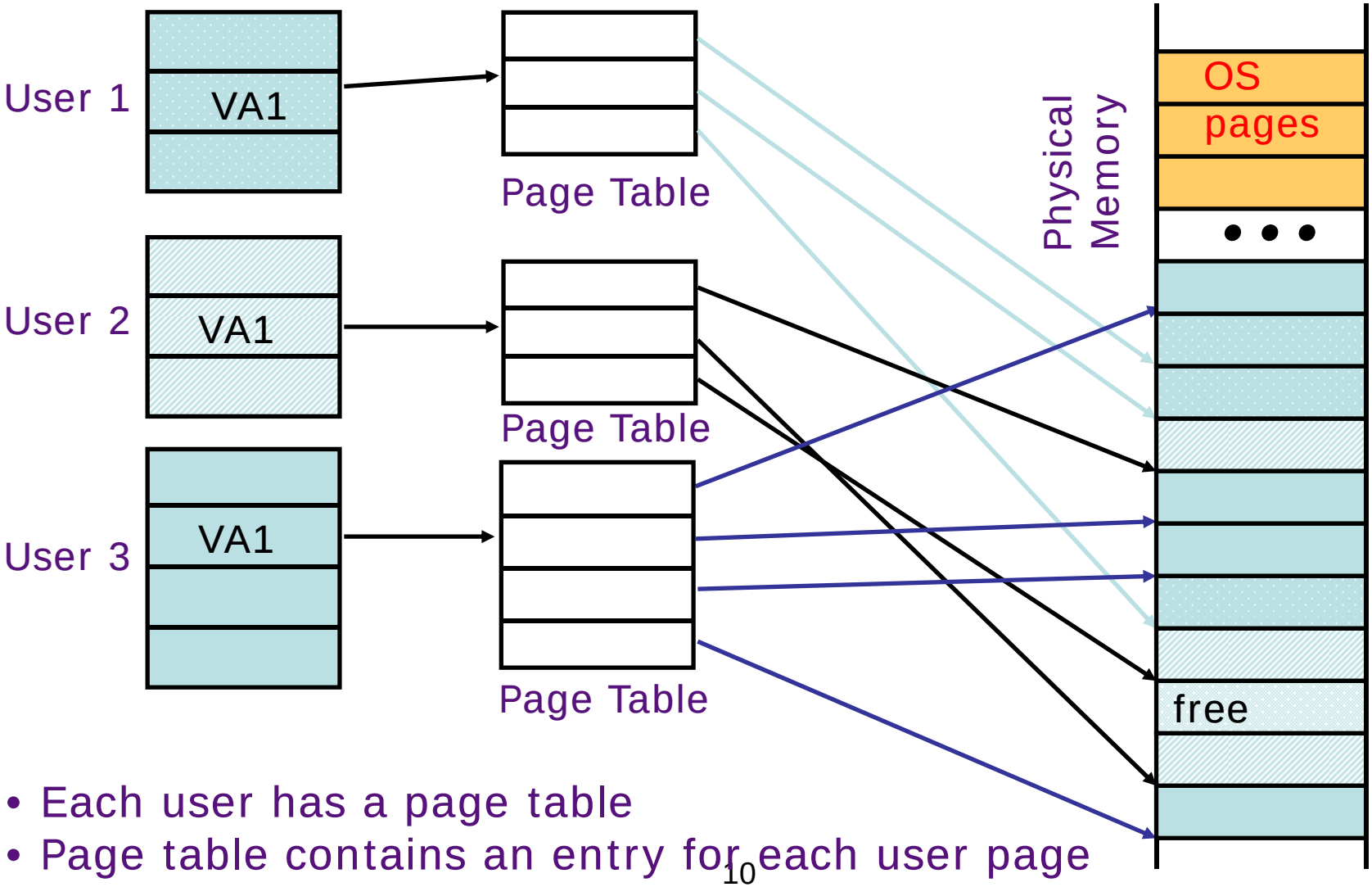
- **Processor generated address can be interpreted as a pair <page number, offset>**

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



Page tables make it possible to store the pages of a program non-contiguously.

Private Address Space per User



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

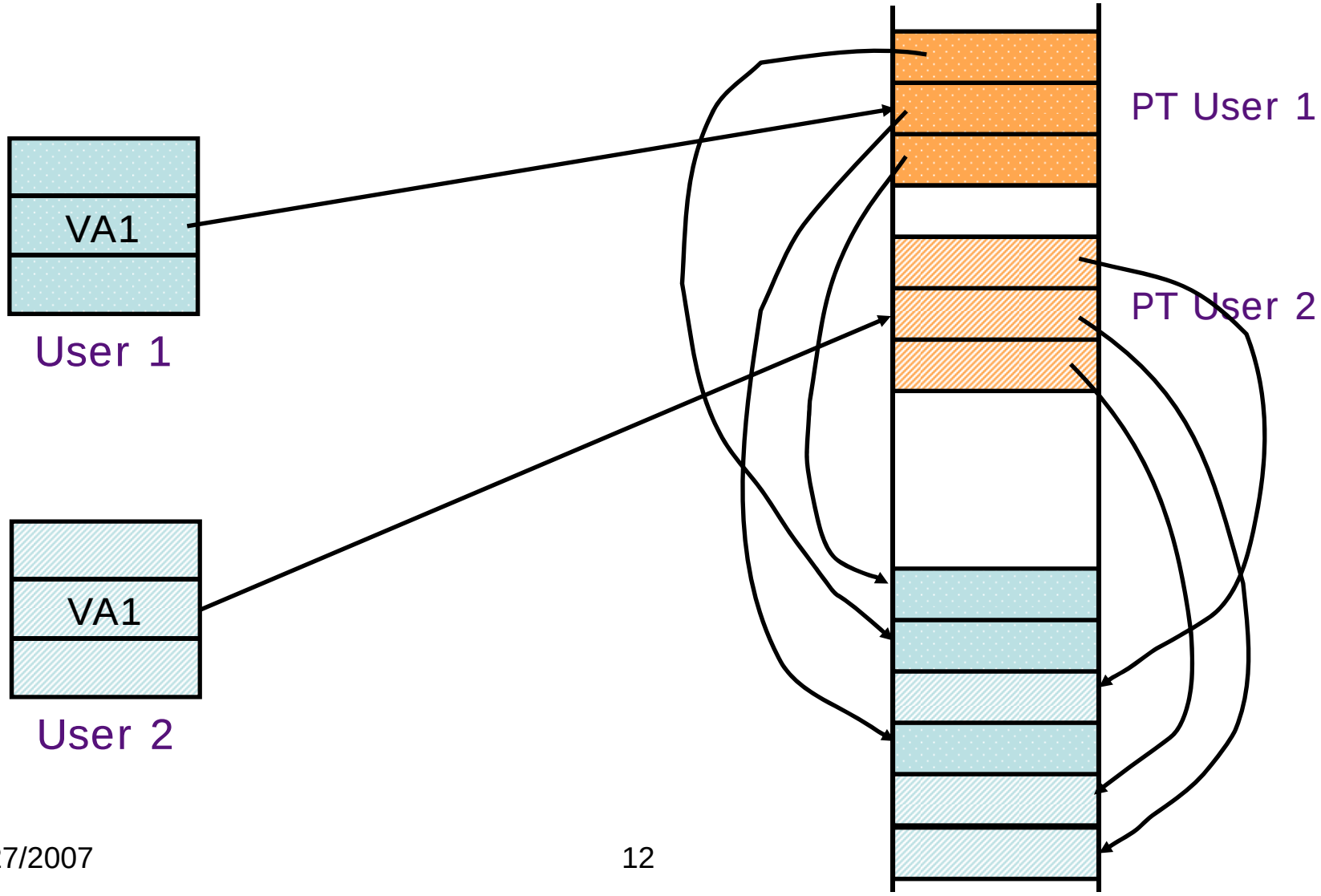
Where Should Page Tables Reside?

- **Space required by the page tables (PT) is proportional to the address space, number of users, ...**
  - ⇒ **Space requirement is large**
  - ⇒ **Too expensive to keep in registers**

- **Idea: Keep PTs in the main memory**
  - **needs one reference to retrieve the page base address and another to access the data word**

    ⇒ ***doubles the number of memory references!***

Page Tables in Physical Memory



A Problem in Early Sixties

- **There were many applications whose data could not fit in the main memory, e.g., payroll**
  - ***Paged memory system reduced fragmentation but still required the whole program to be resident in the main memory***

- **Programmers moved the data back and forth from the secondary store by *overlaying* it repeatedly on the primary store**

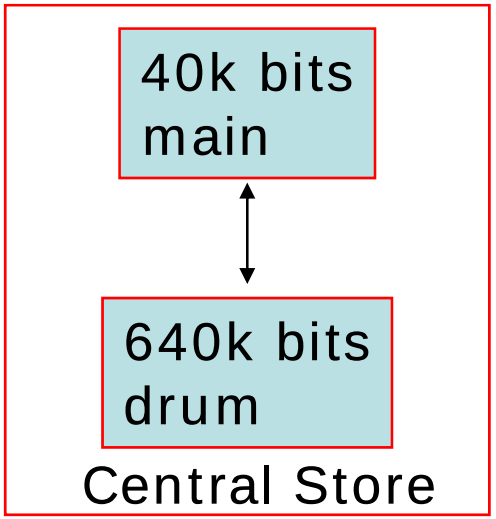
tricky programming!

Manual Overlays

- **Assume an instruction can address all the storage on the drum**
- ***Method 1:*** **programmer keeps track of addresses in the main memory and initiates an I/O transfer when required**
- ***Method 2:*** **automatic initiation of I/O transfers by software address translation**

Brooker's interpretive coding, 1960

Method1: Difficult, error prone
Method2: Inefficient



Ferranti Mercury
1956

Not just an ancient black art, e.g., IBM Cell microprocessor explicitly managed local store has same issues

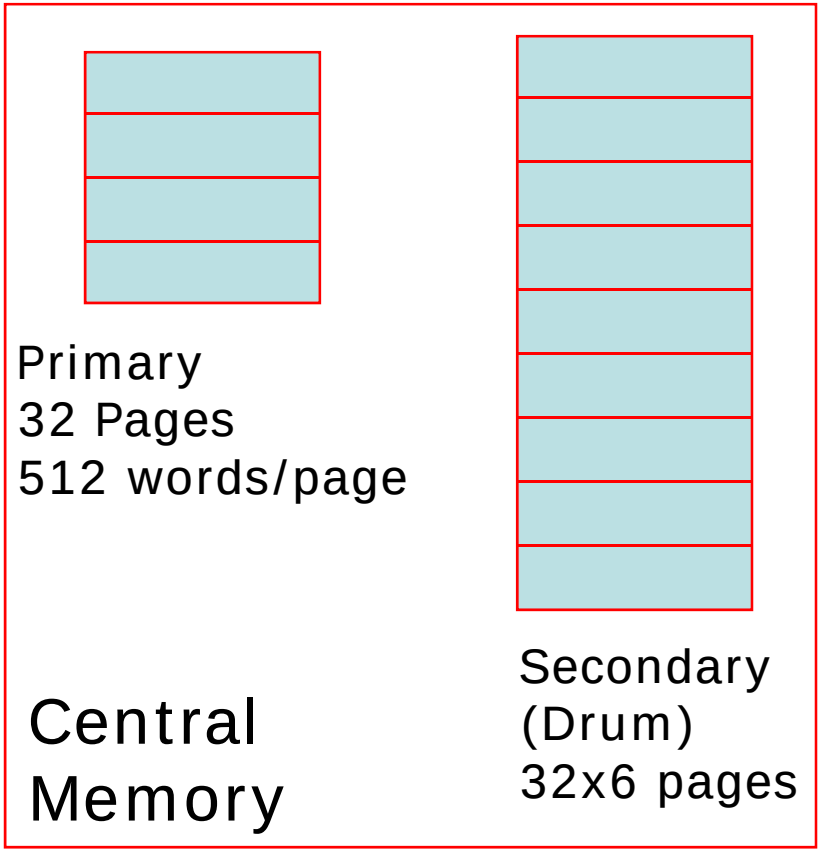
Demand Paging in Atlas (1962)

"A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor."

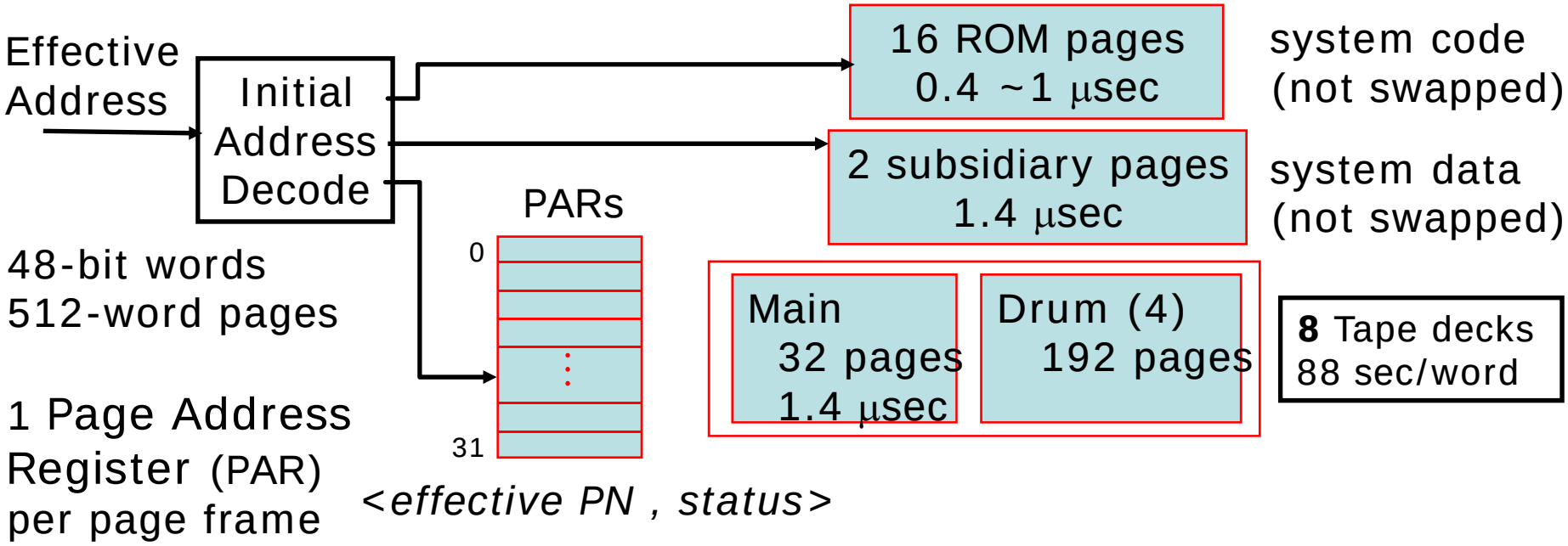
Tom Kilburn

Primary memory as a *cache* for secondary memory

User sees 32 x 6 x 512 words of storage



Hardware Organization of Atlas



Compare the effective page address against all 32 PARs

match ⇒ normal access

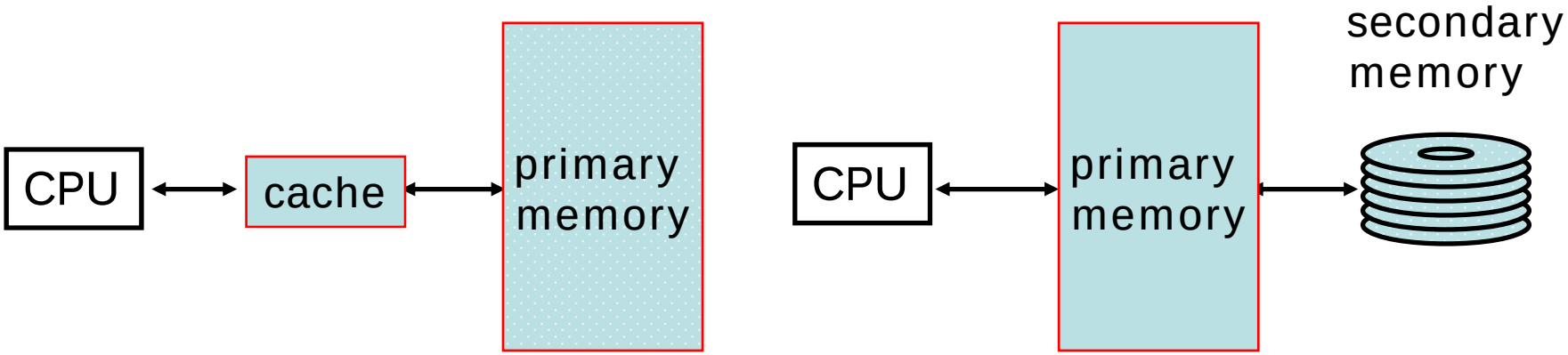
no match ⇒ *page fault*

save the state of the partially executed instruction

Atlas Demand Paging Scheme

- **On a page fault:**
  - **Input transfer into a free page is initiated**
  - **The Page Address Register (PAR) is updated**
  - **If no free page is left, a *page is selected to be replaced* (based on usage)**
  - **The replaced page is written on the drum**
    - to minimize drum latency effect, the first empty page on the drum was selected
  - **The *page table is updated* to point to the new location of the page on the drum**

Caching vs. Demand Paging



Caching

- cache entry
- cache block (~32 bytes)
- cache miss rate (1% to 20%)
- cache hit (~1 cycle)
- cache miss (~100 cycles)
- a miss is handled
  - in *hardware*

Demand paging

- page frame
- page (~4K bytes)
- page miss rate (<0.001%)
- page hit (~100 cycles)
- page miss (~5M cycles)
- a miss is handled
  - mostly in *software*

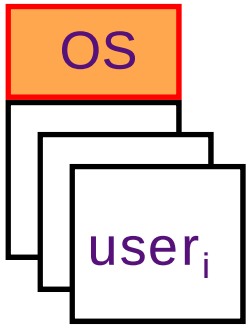
Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy

several users, each with their private address space and one or more shared address spaces

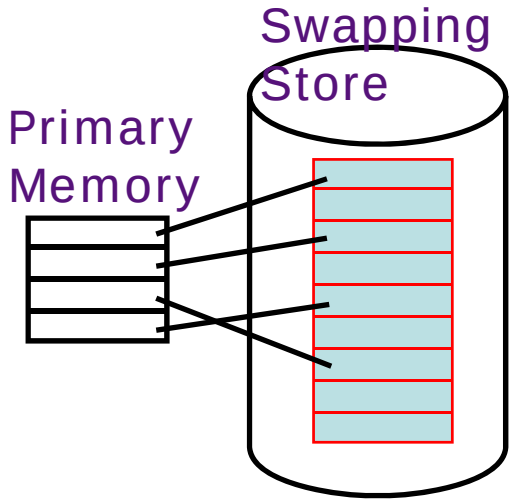
page table ≡ name space



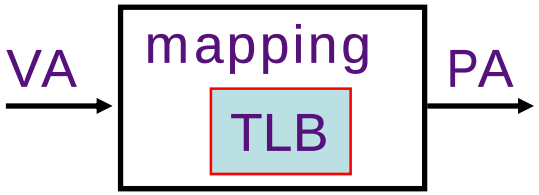
Demand Paging

Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations

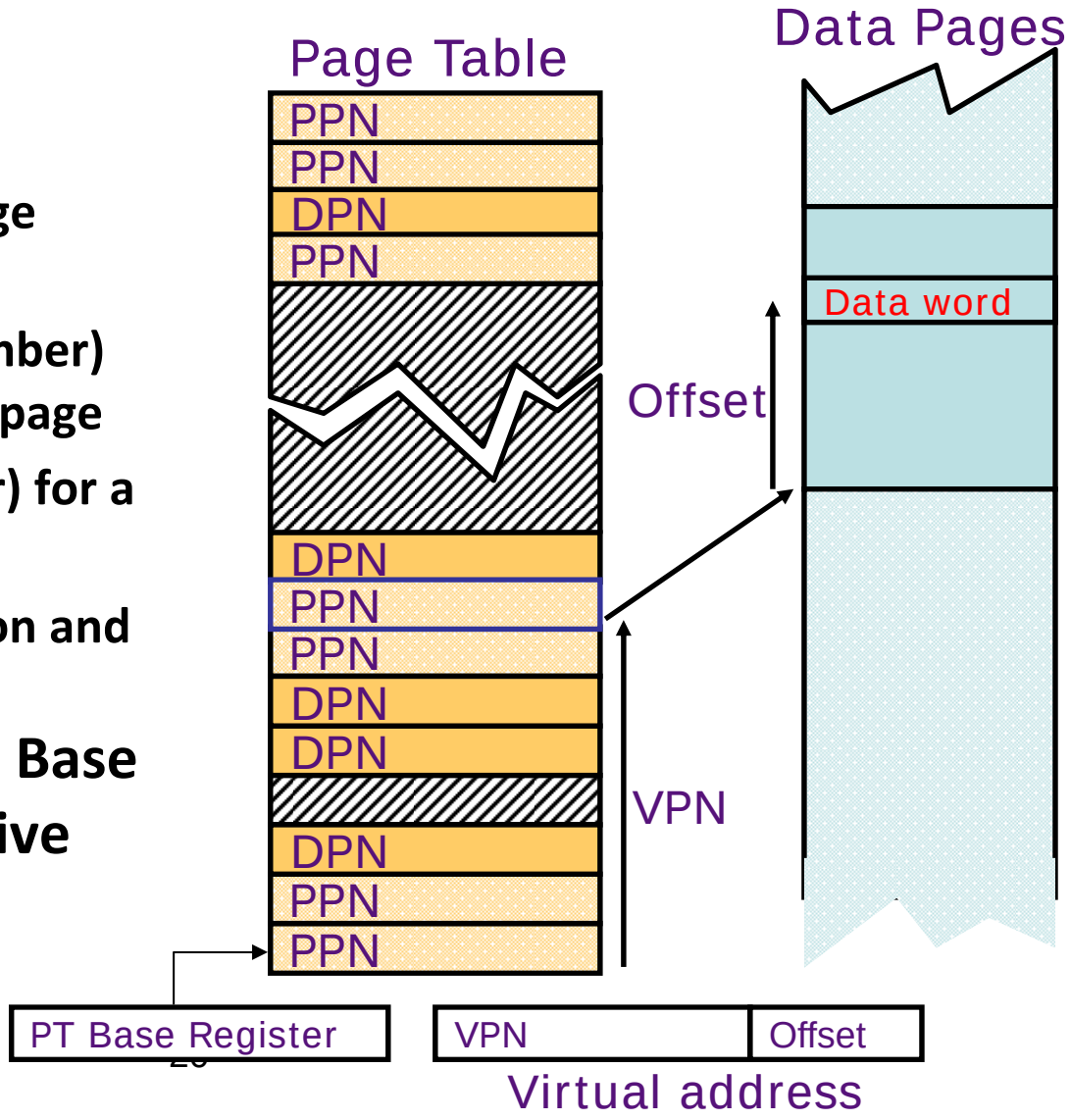


The price is address translation on each memory reference



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, 4-KB pages & 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?

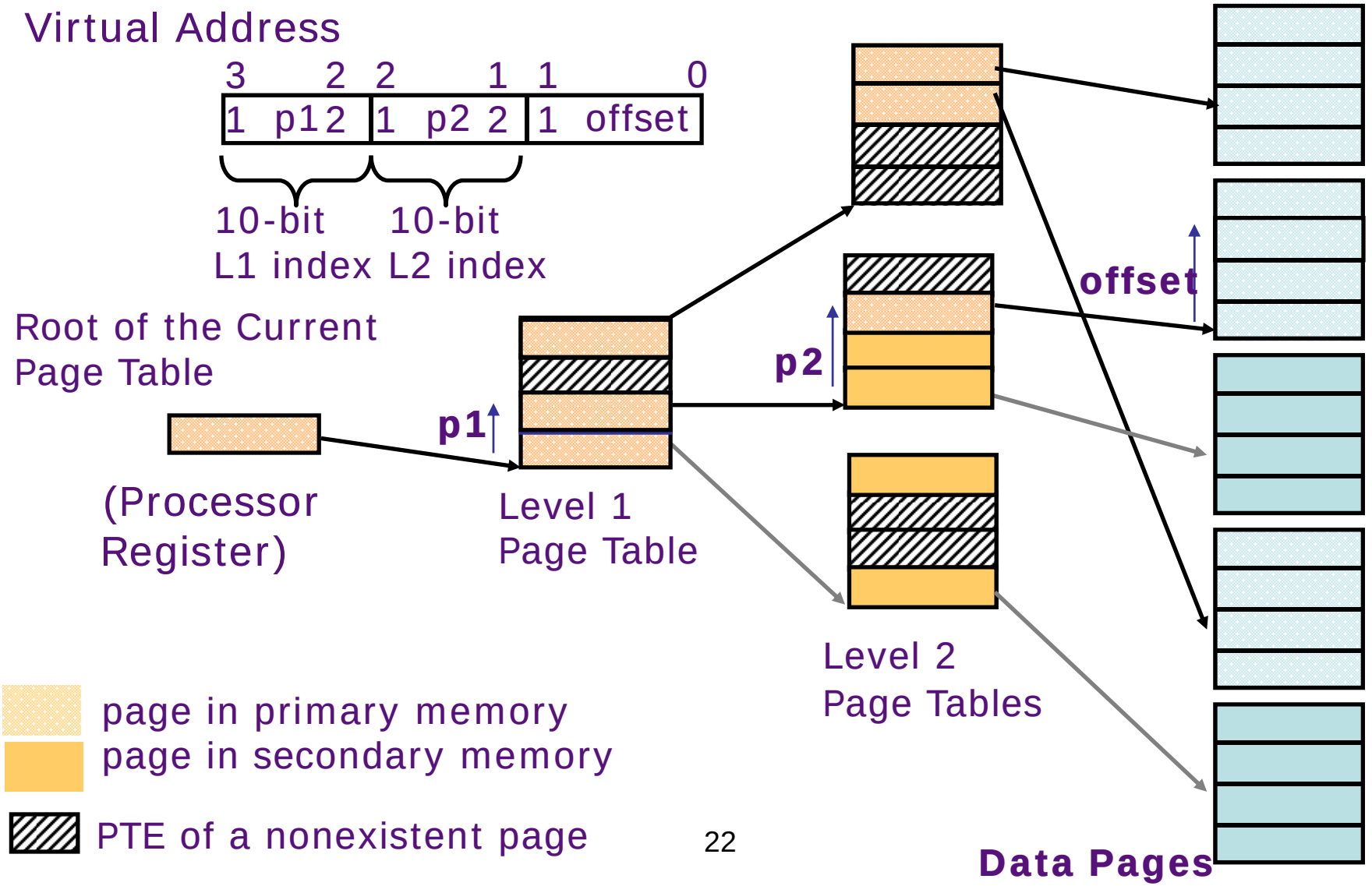
- **Internal fragmentation (Not all memory in a page is used)**
- **Larger page fault penalty (more time to read from disk)**

What about 64-bit virtual address space???

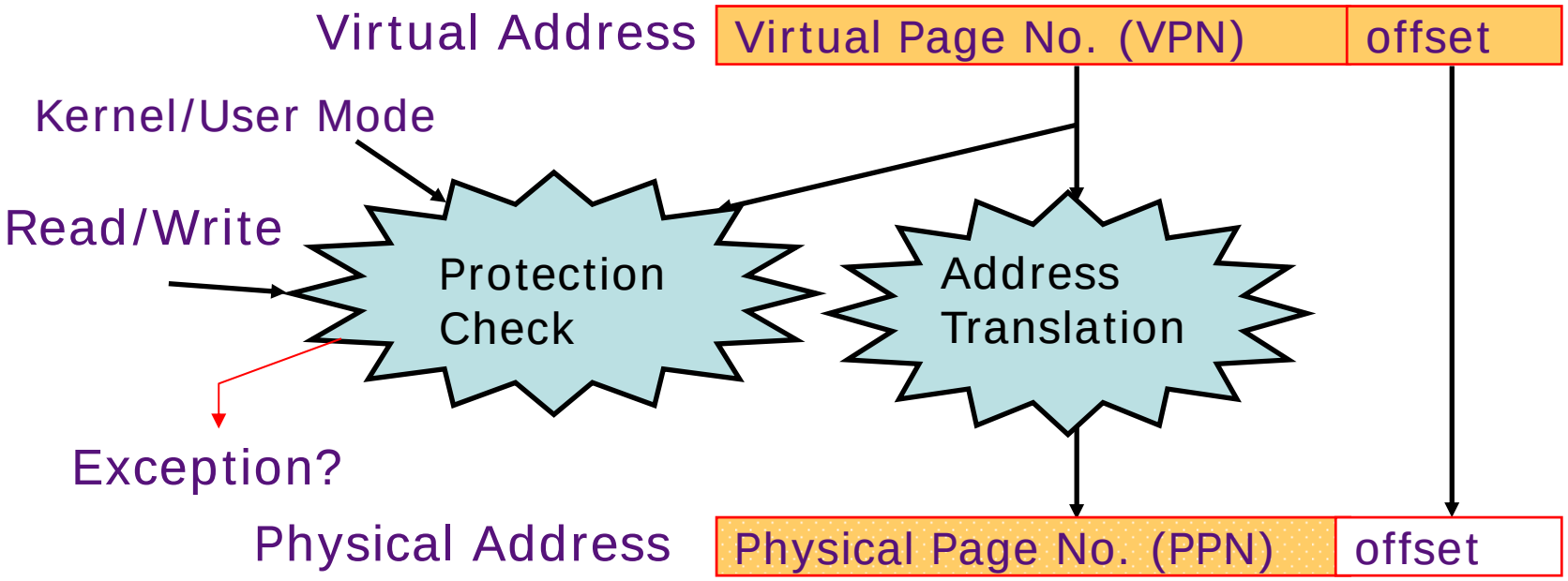
- **Even 1MB pages would require 2^{44} 8-byte PTEs (35 TB!)**

What is the "saving grace" ?

Hierarchical Page Table



Address Translation & 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

Translation Lookaside Buffers

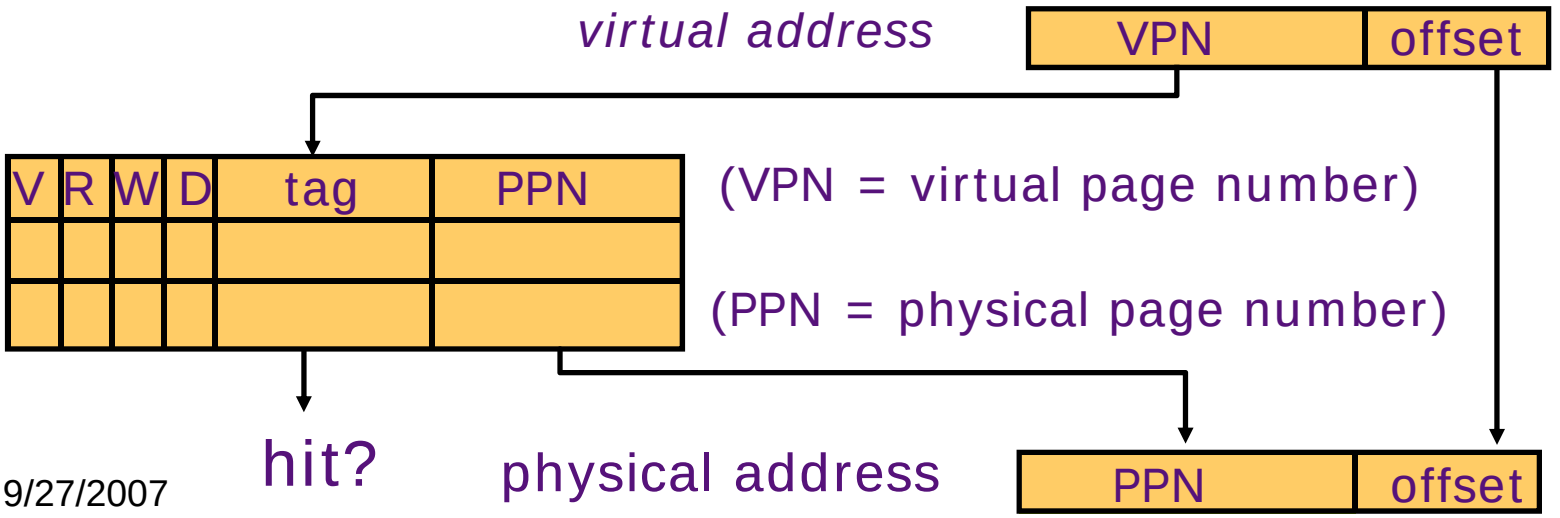
Address translation is very expensive!

In a two-level page table, each reference becomes several memory accesses

Solution: *Cache translations in TLB*

TLB hit ⇒ *Single Cycle Translation*

TLB miss ⇒ *Page Table Walk to refill*



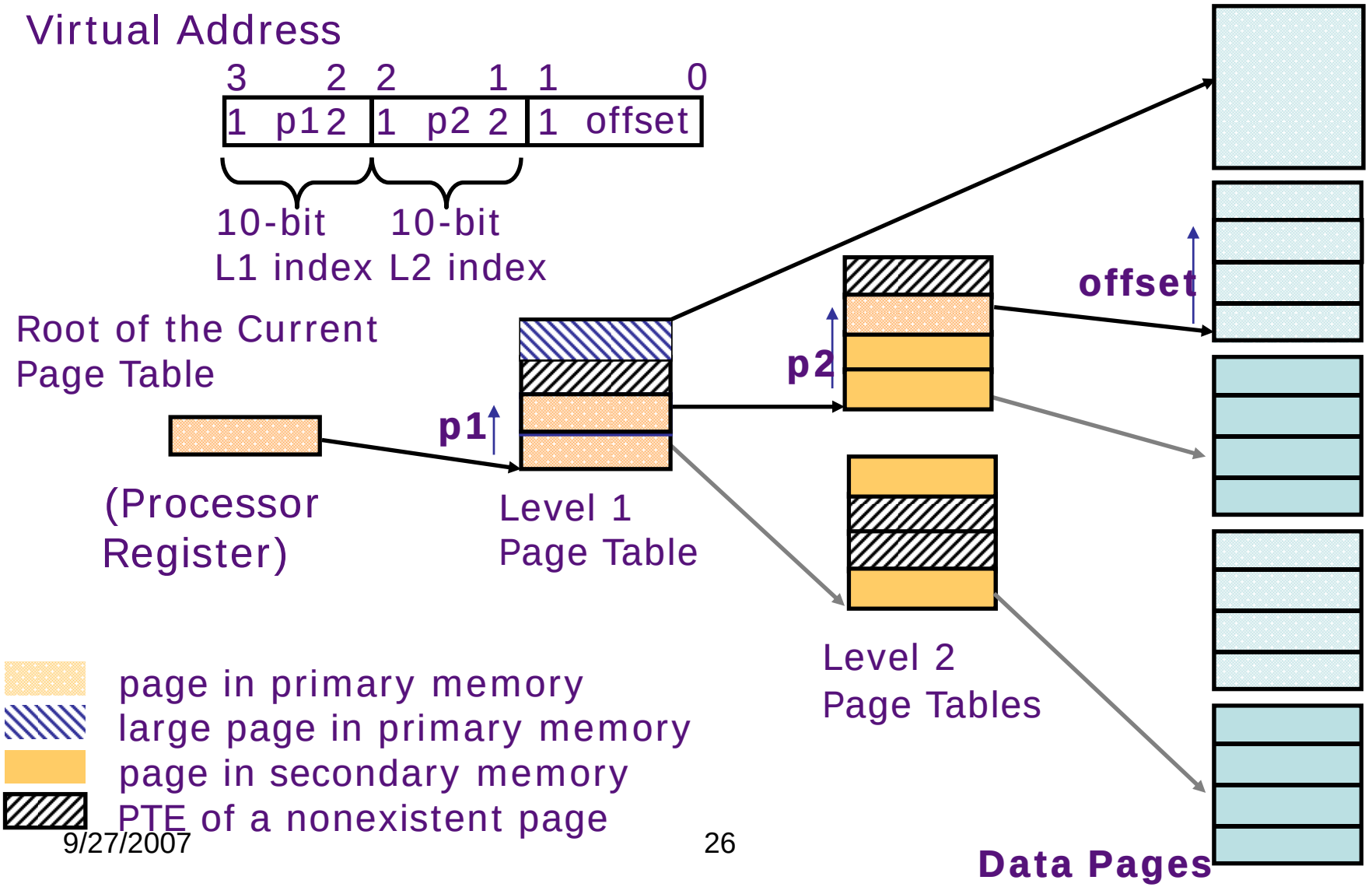
TLB Designs

- **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 (256-512 entries) are 4-8 way set-associative**
- **Random or FIFO replacement policy**
- **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

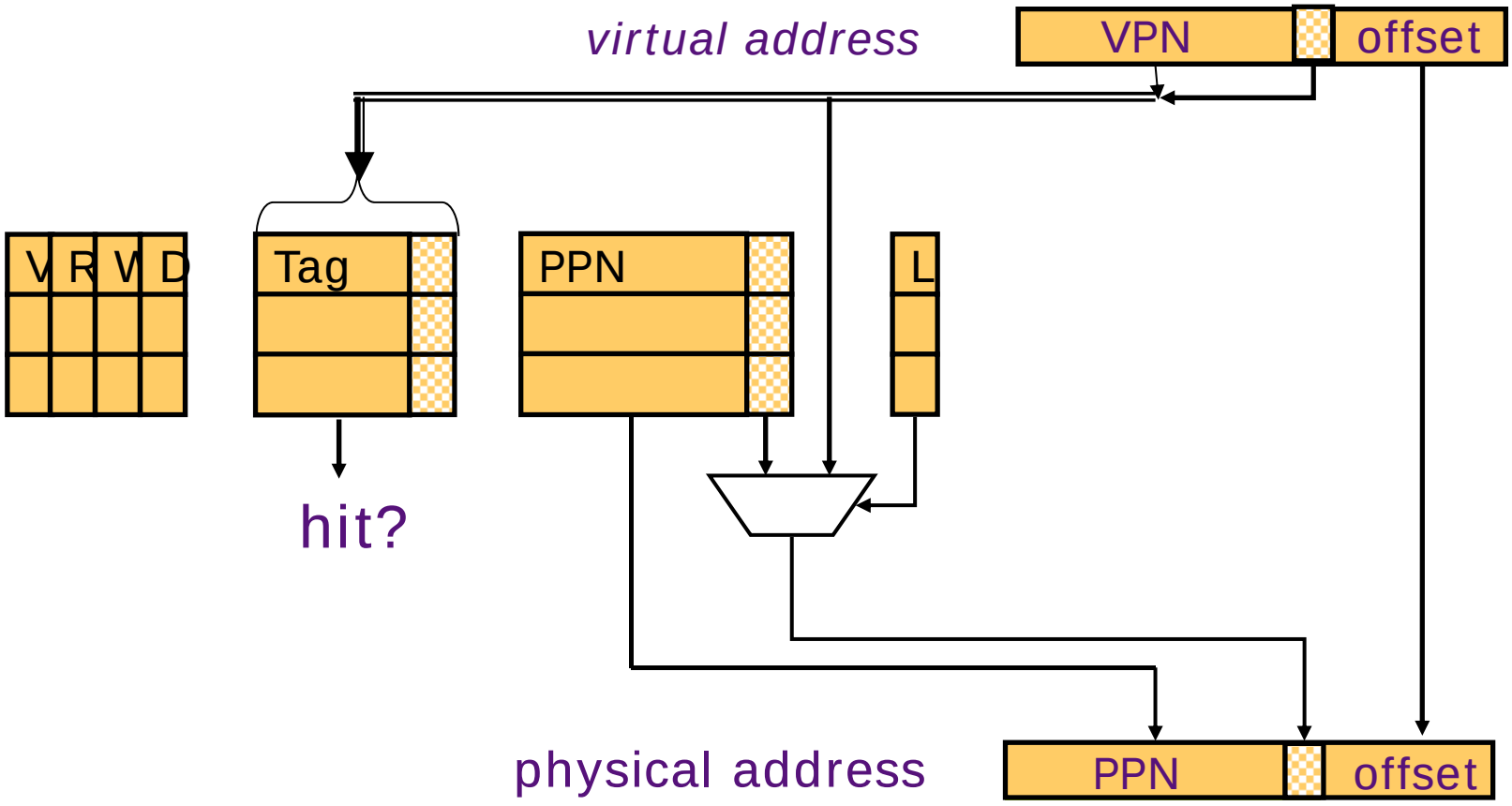
TLB Reach = *64 entries * 4 KB = 256 KB (if contiguous)* **?**

Variable-Sized Page Support



Variable-Size Page TLB

Some systems support multiple page sizes.



Handling a TLB Miss

Software (MIPS, Alpha)

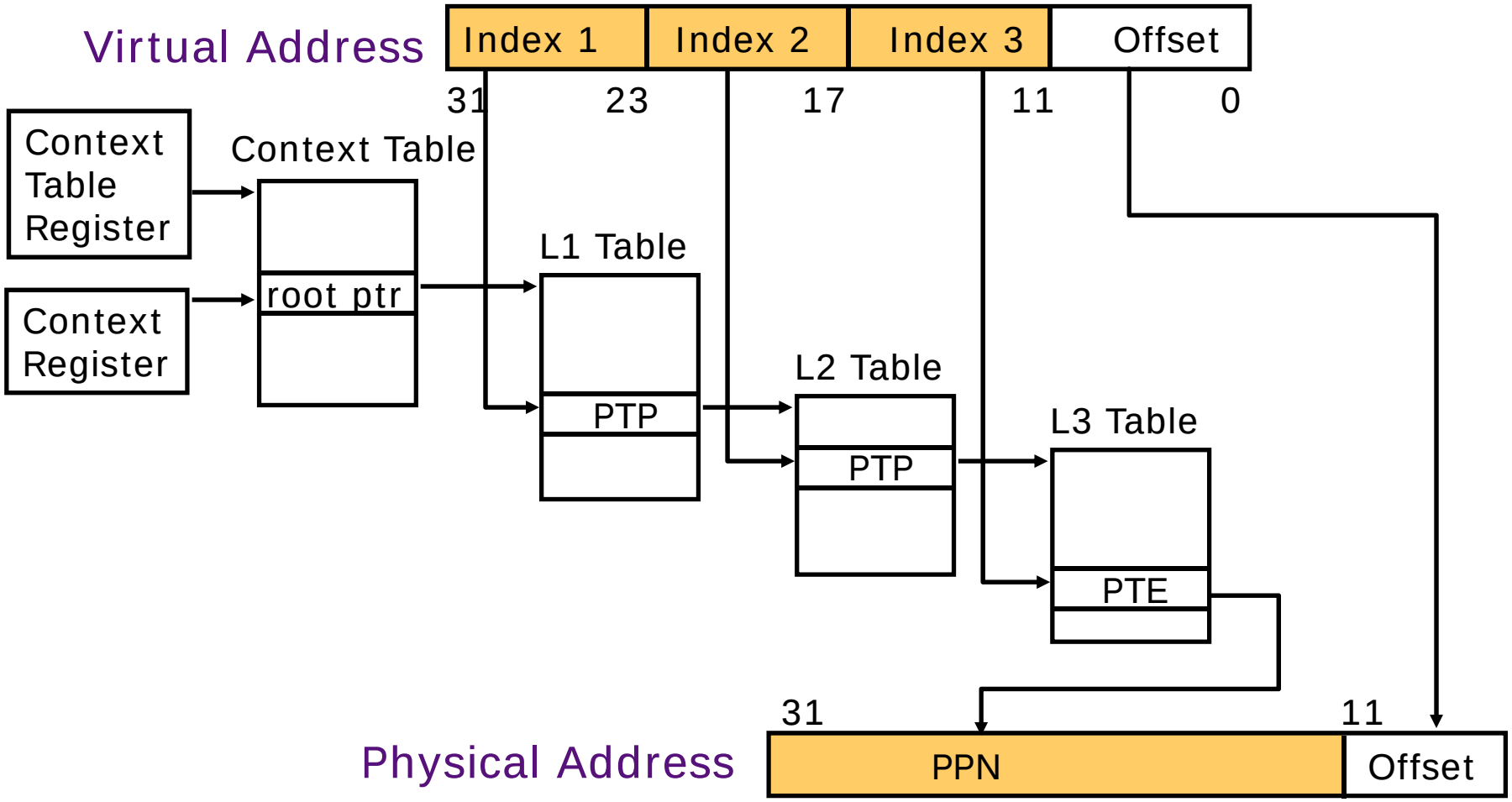
TLB miss causes an exception and the operating system walks the page tables and reloads TLB. *A privileged "untranslated" addressing mode used for walk*

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

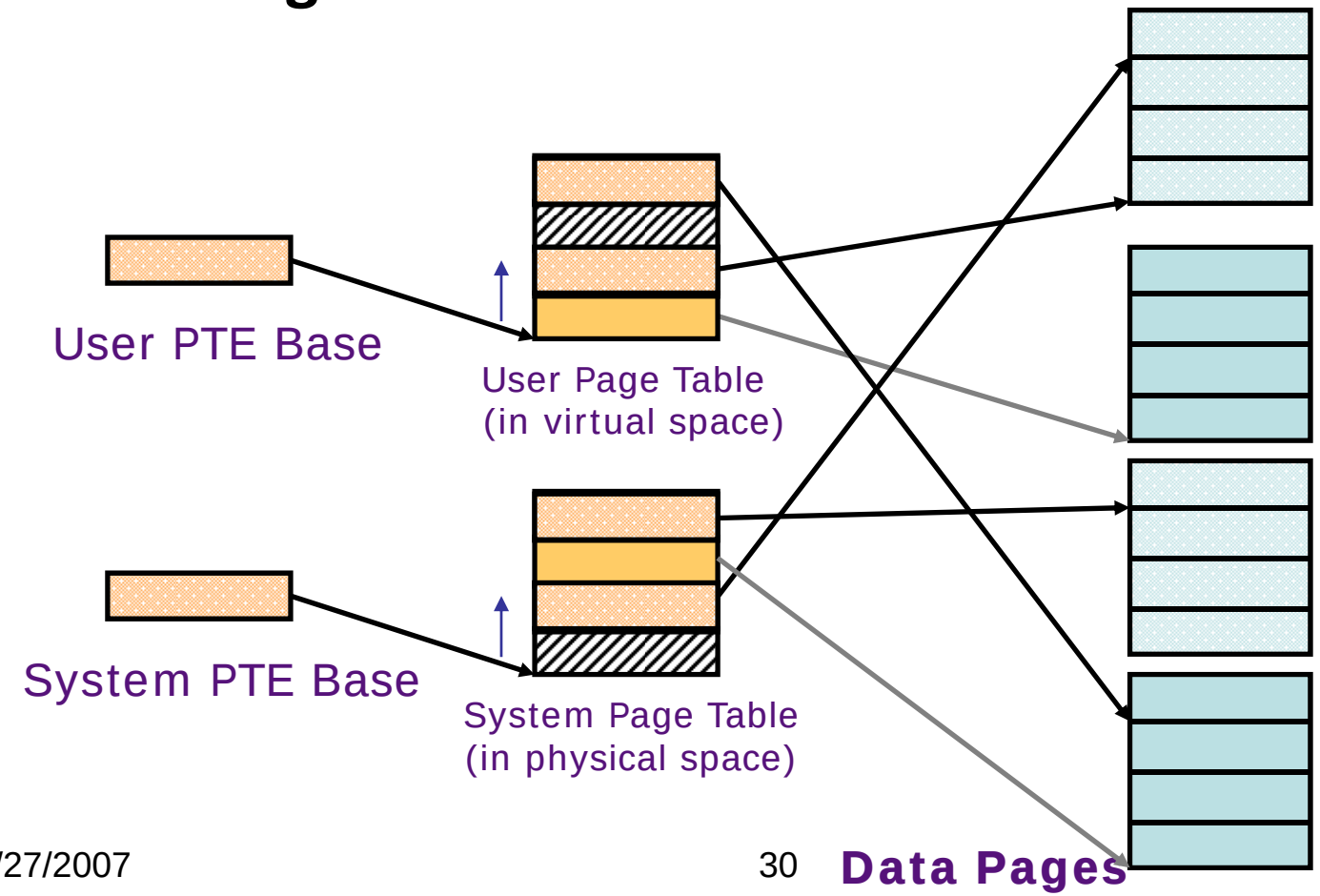
Hierarchical Page Table Walk: SPARC v8



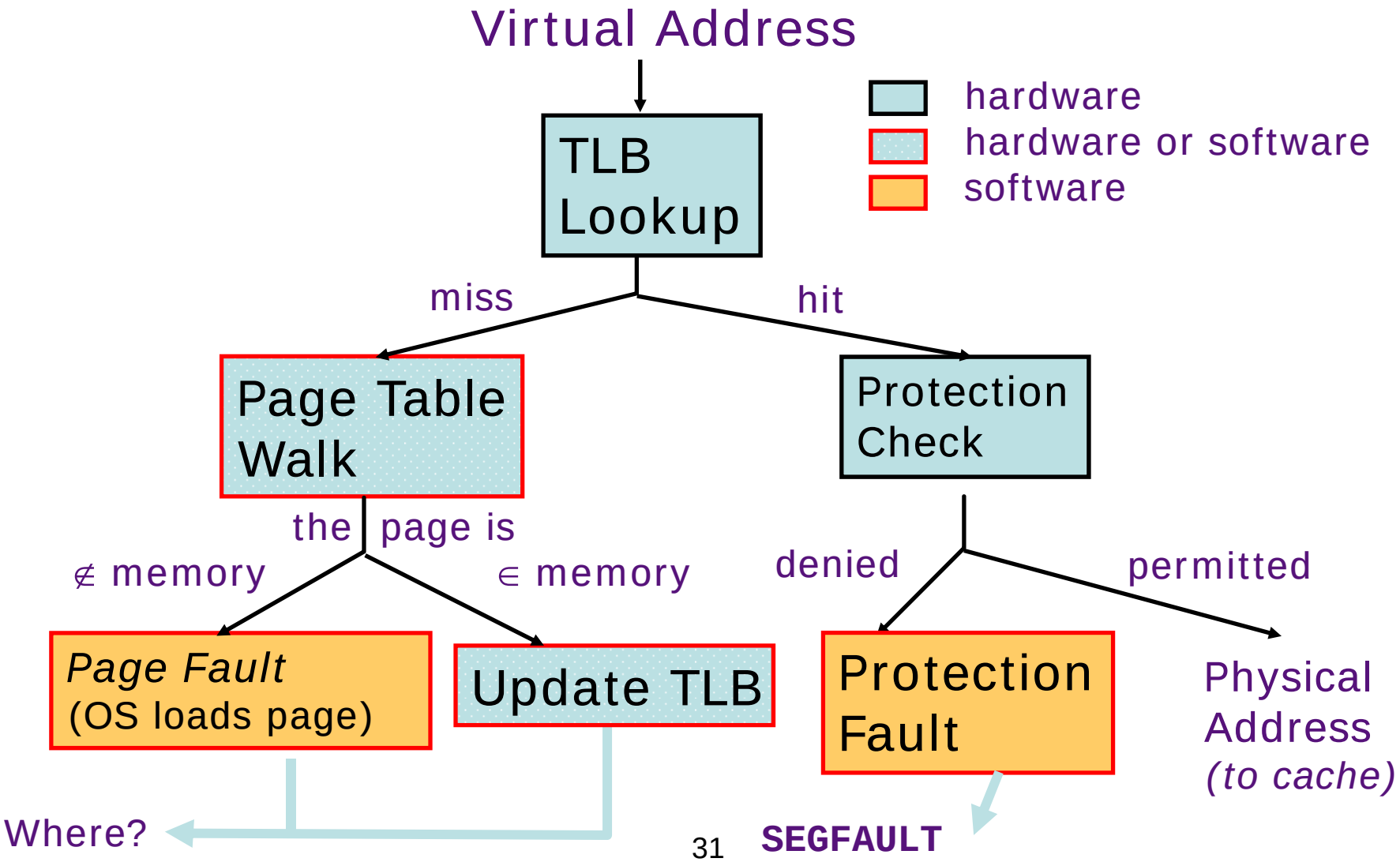
MMU does this table walk in hardware on a TLB miss

Translation for Page Tables

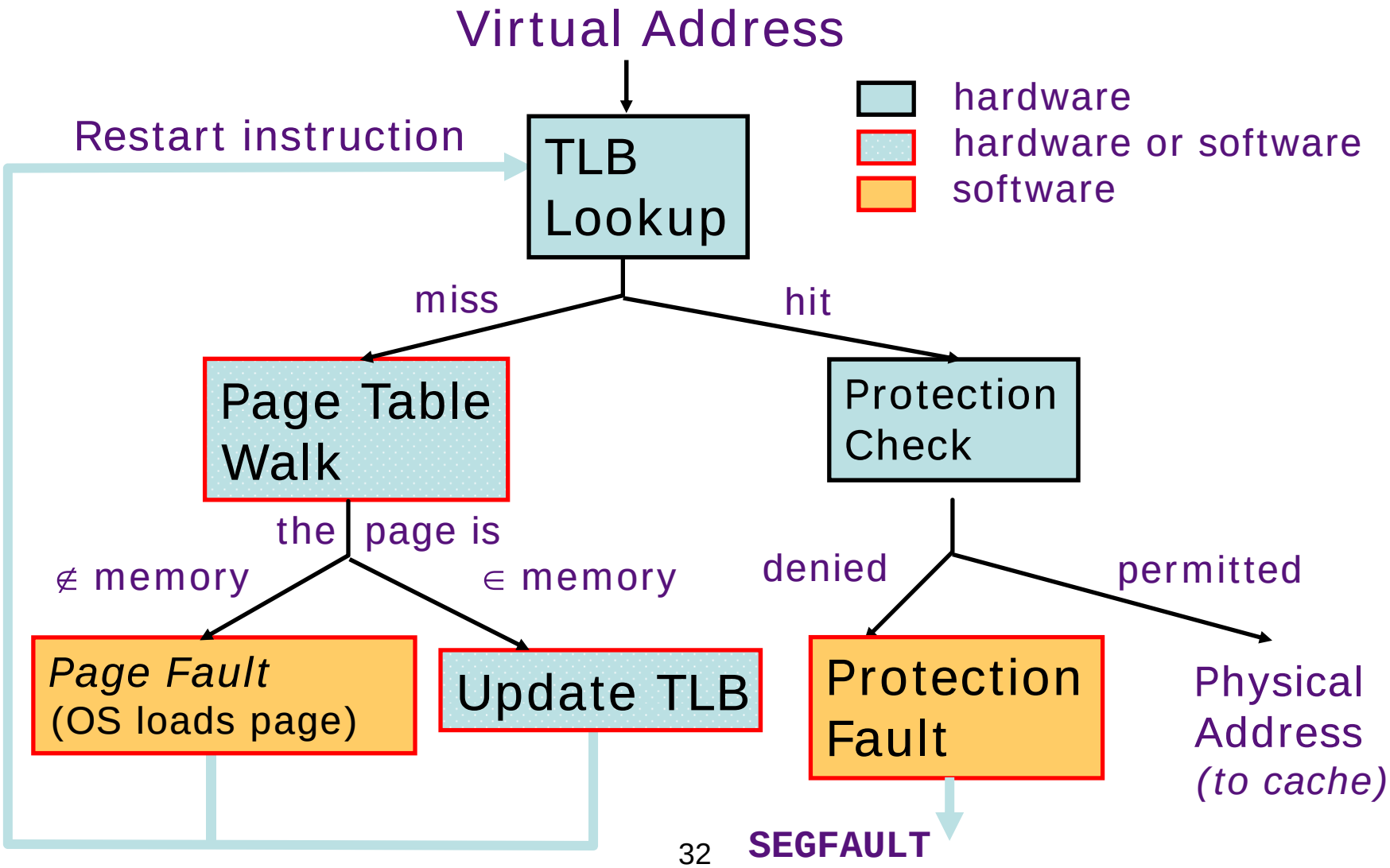
- **Can references to page tables cause TLB misses?**
- **Can this go on forever?**



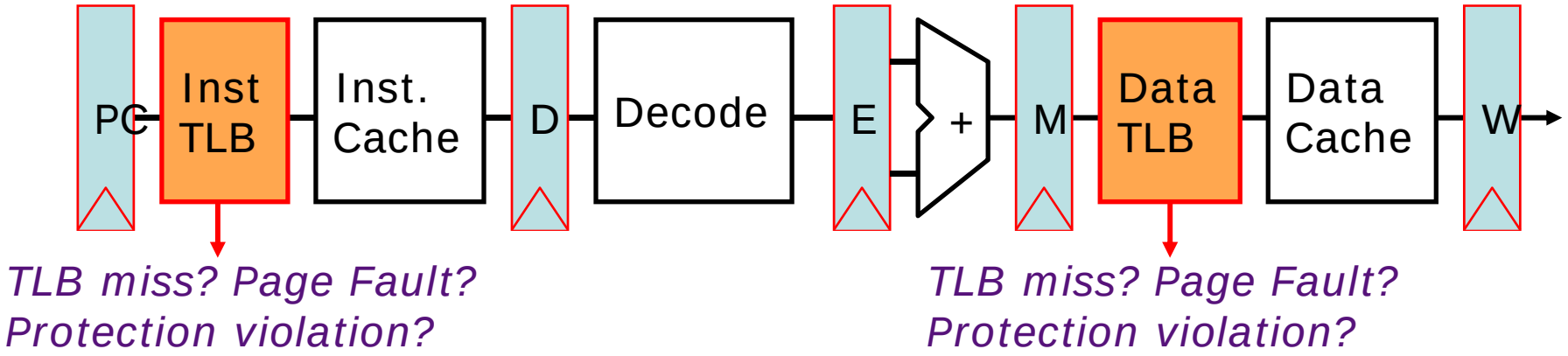
Address Translation:
putting it all together



Address Translation:
putting it all together

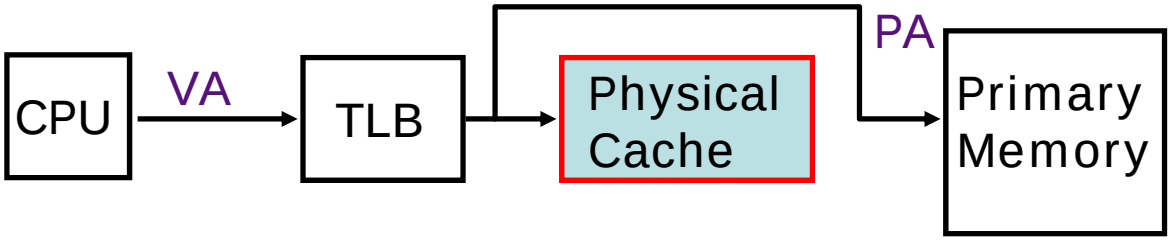


Address Translation in CPU Pipeline

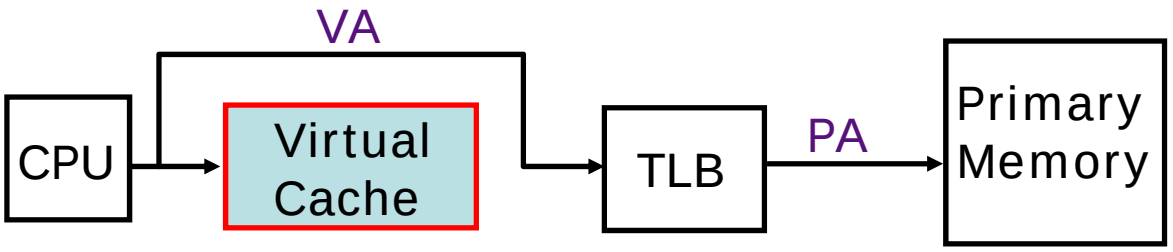


- **Software handlers need a *restartable* exception on page fault or protection violation**
- **Handling a TLB miss needs a *hardware* or *software* mechanism to refill TLB**
- **Need mechanisms to cope with the additional latency of a TLB:**
  - **slow down the clock**
  - **pipeline the TLB and cache access**
  - **virtual address caches**
  - **parallel TLB/cache access**

Virtual Address Caches



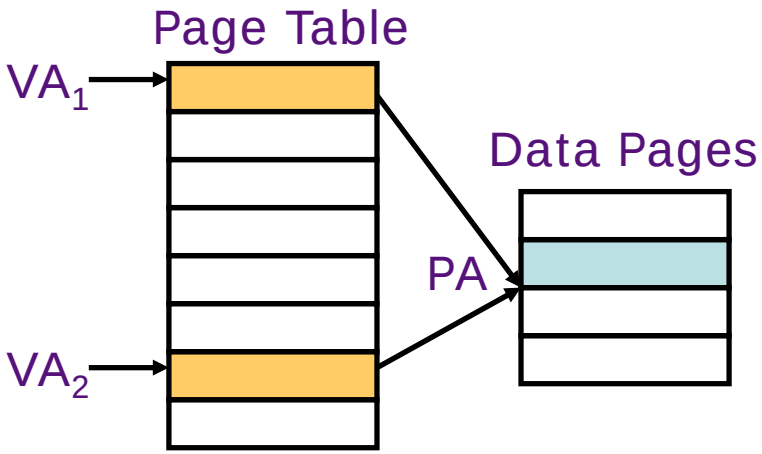
Alternative: place the cache before the TLB



(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

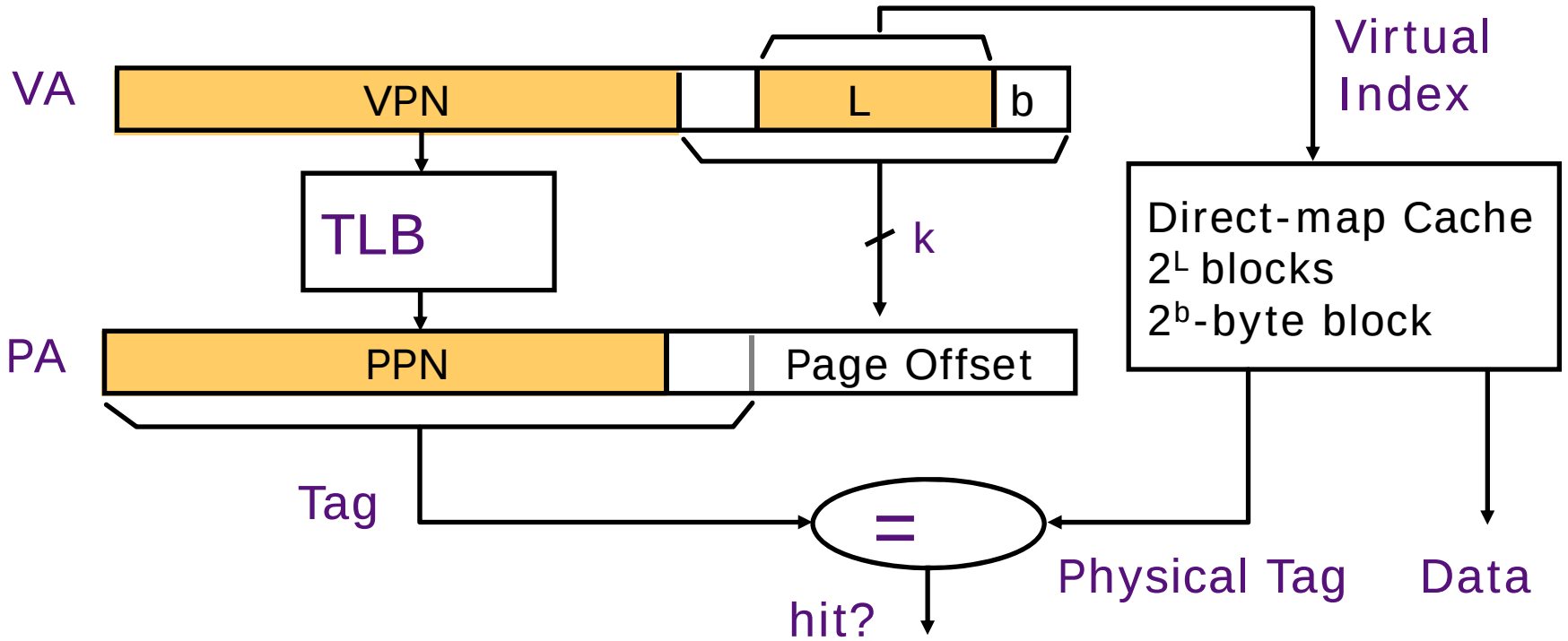
Tag	Data
VA_1	1st Copy of Data at PA
VA_2	2nd Copy of Data at PA

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*
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

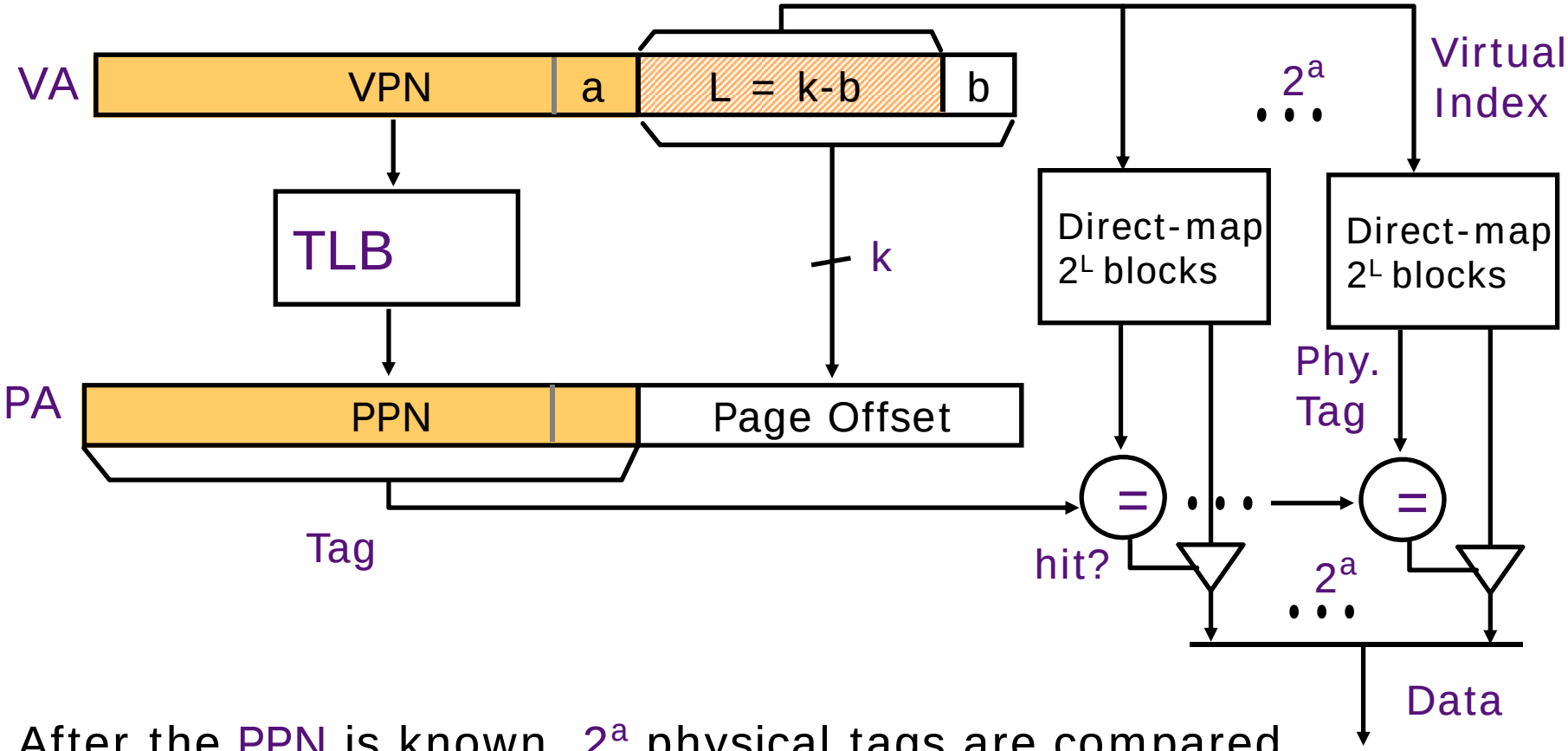
⇒ *cache and TLB accesses can begin simultaneously*

Tag comparison is made after both accesses are completed

Cases: $L + b = k$ $L + b < k$ $L + b > k$

Virtual-Index Physical-Tag Caches:

Associative Organization

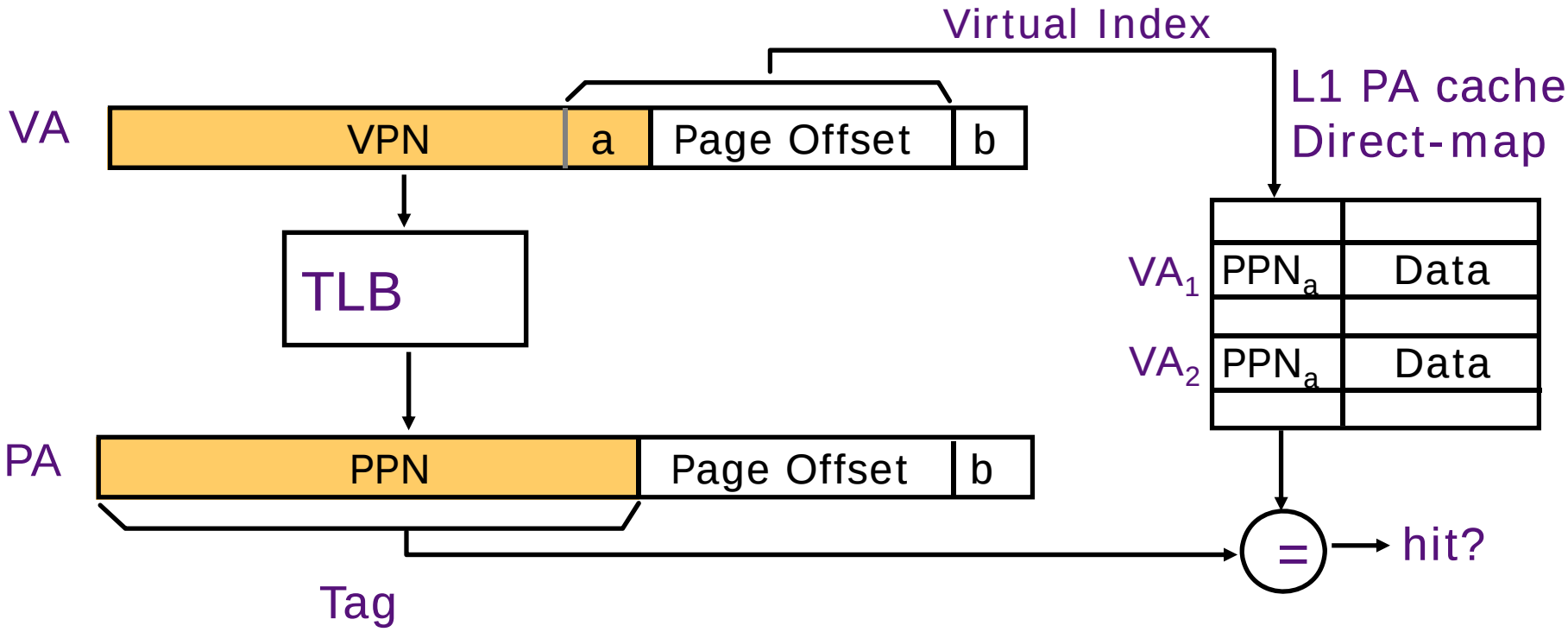


After the PPN is known, 2^a physical tags are compared

Is this scheme realistic?

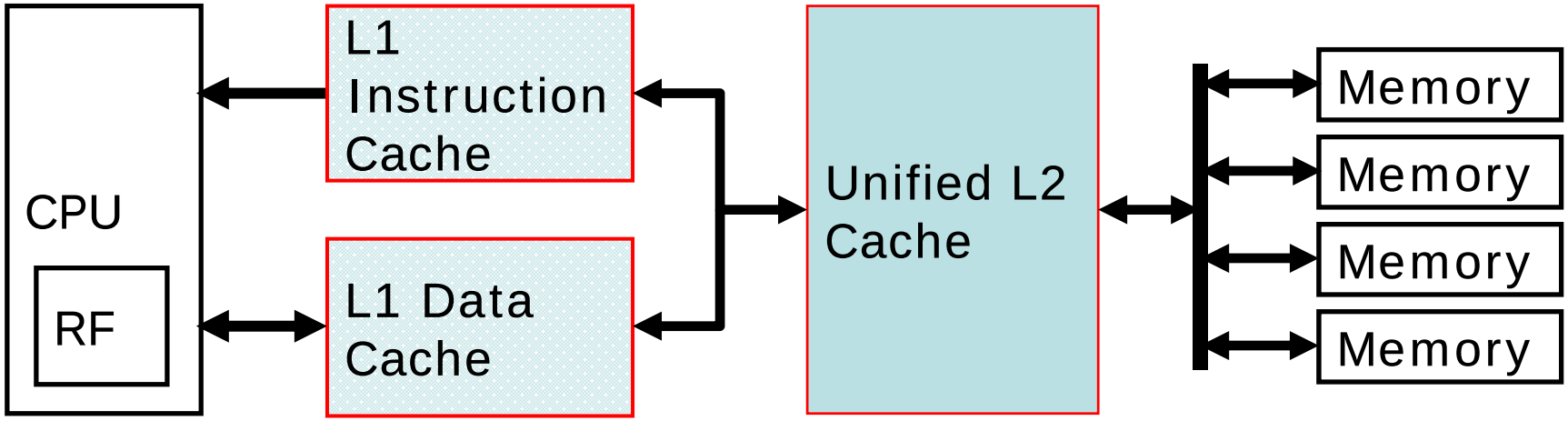
Concurrent Access to TLB & Large L1

The problem with L1 > Page size



Can VA_1 and VA_2 both map to PA *?*

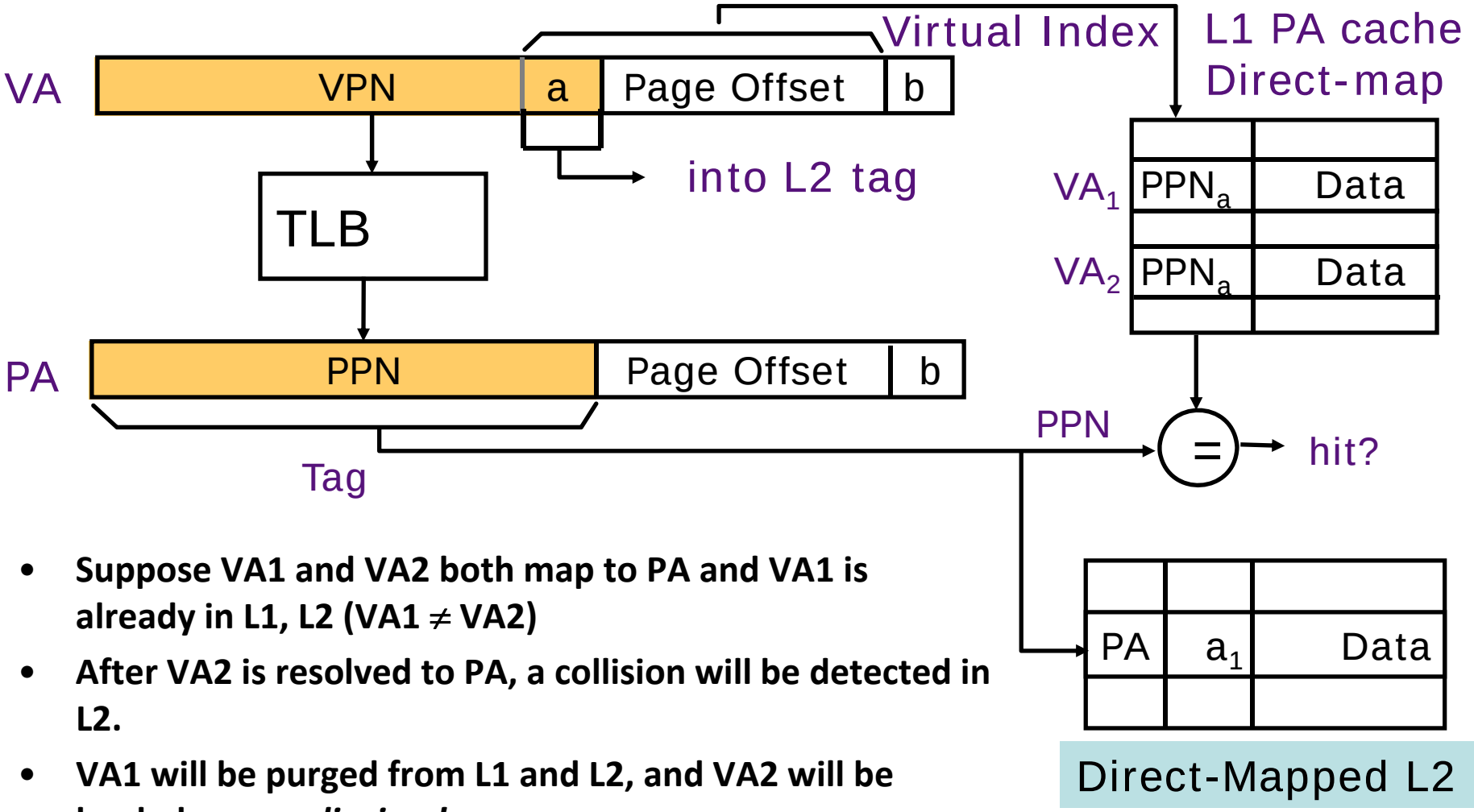
A solution via Second Level Cache



Usually a common L2 cache backs up both Instruction and Data L1 caches

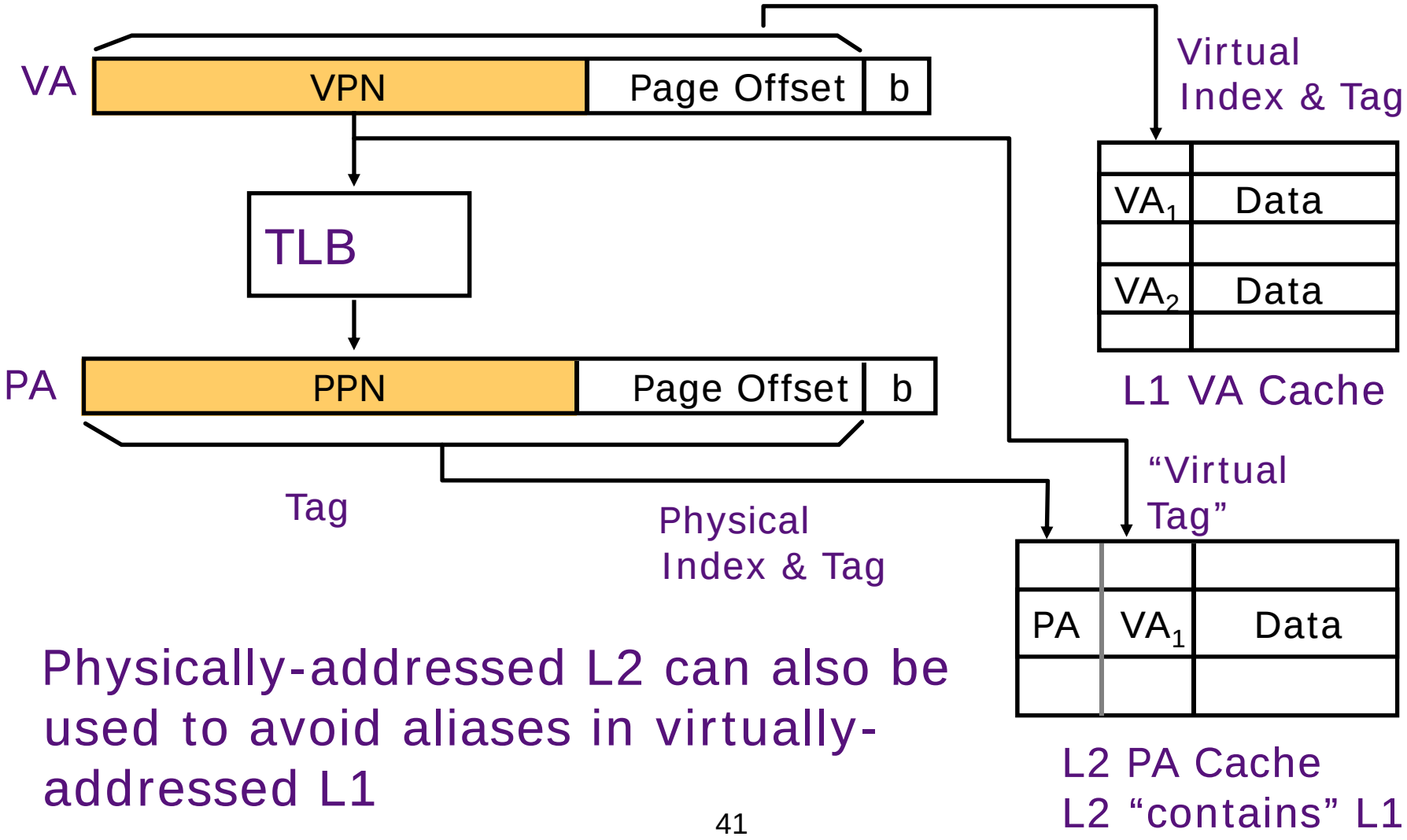
L2 is “inclusive” of both Instruction and Data caches

Anti-Aliasing Using L2: *MIPS R10000*



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

Virtually-Addressed L1:
Anti-Aliasing using L2



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

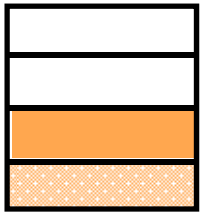
Page Fault Handler

- **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**

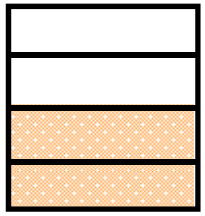
    Another job may be run on the CPU while the first job waits for the requested page to be read from disk
  - **If no free pages are left, a page is swapped out**

    Pseudo-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**

Swapping a Page of a Page Table



A PTE in primary memory contains primary or secondary memory addresses



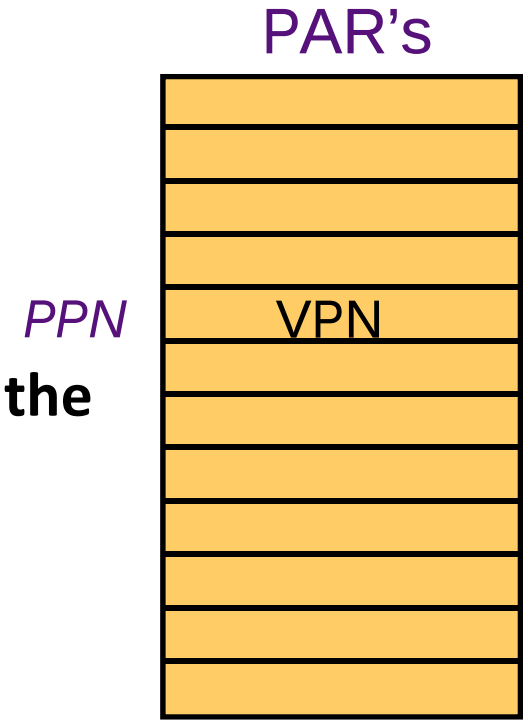
A PTE in secondary memory contains *only* secondary memory addresses

$\Rightarrow$ a page of a PT can be swapped out only if none its PTE's point to pages in the primary memory

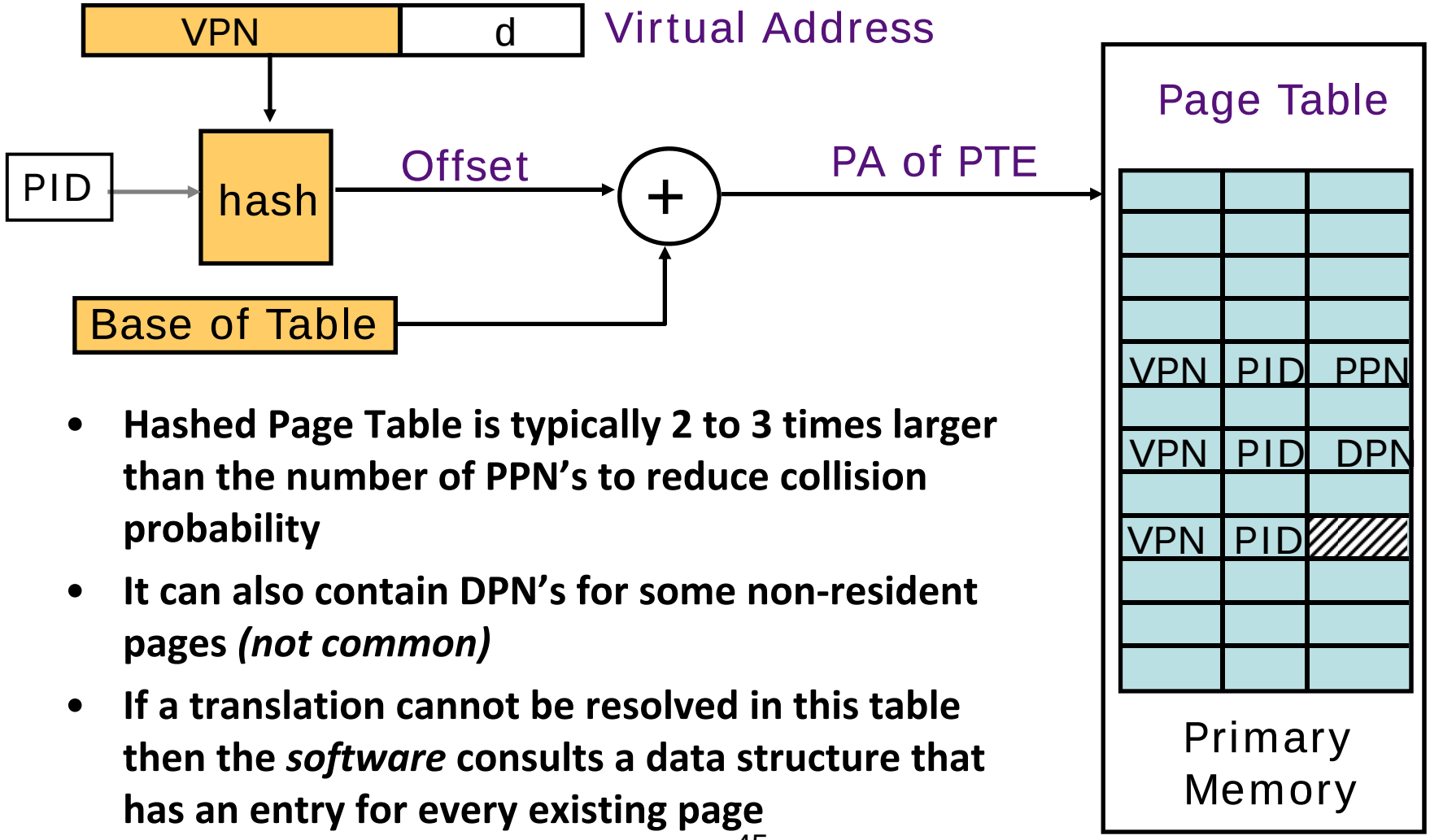
*Why?*_____

Atlas Revisited

- **One PAR for each physical page**
- **PAR's contain the VPN's of the pages *resident in primary memory***
- ***Advantage:*** **The size is proportional to the size of the primary memory**
- ***What is the disadvantage ?***

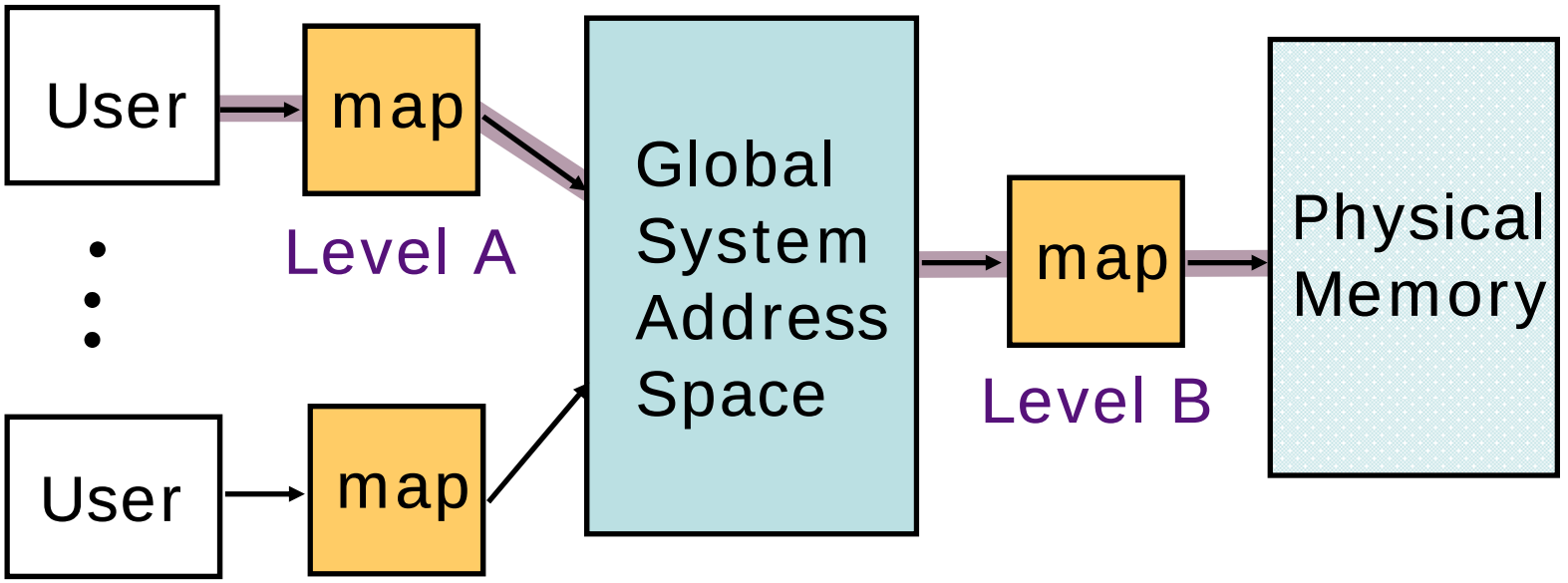


Hashed Page Table:
Approximating Associative Addressing



- **Hashed Page Table is typically 2 to 3 times larger than the number of PPN's to reduce collision probability**
- **It can also contain DPN's for some non-resident pages *(not common)***
- **If a translation cannot be resolved in this table then the *software* consults a data structure that has an entry for every existing page**

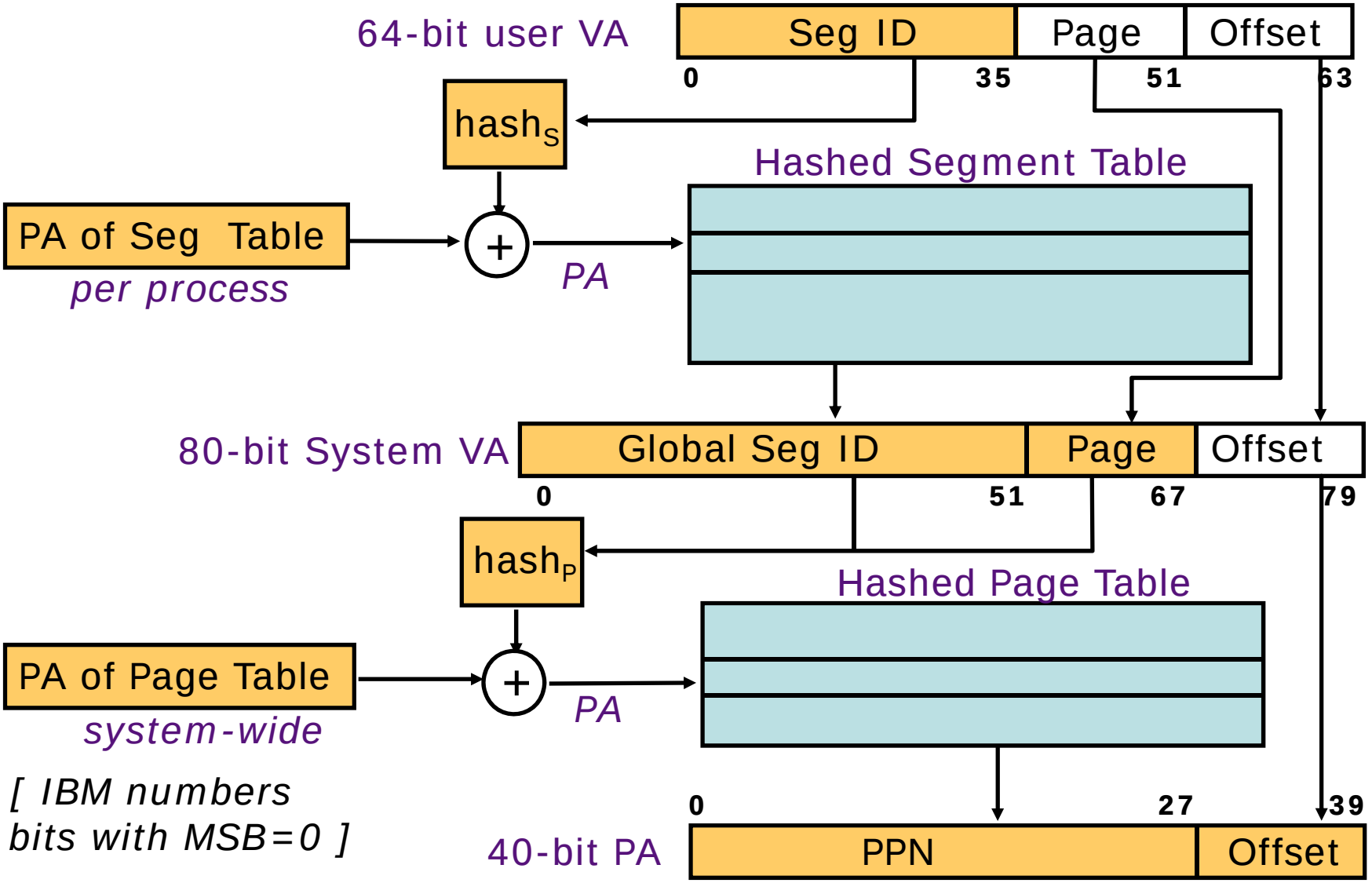
Global System Address Space



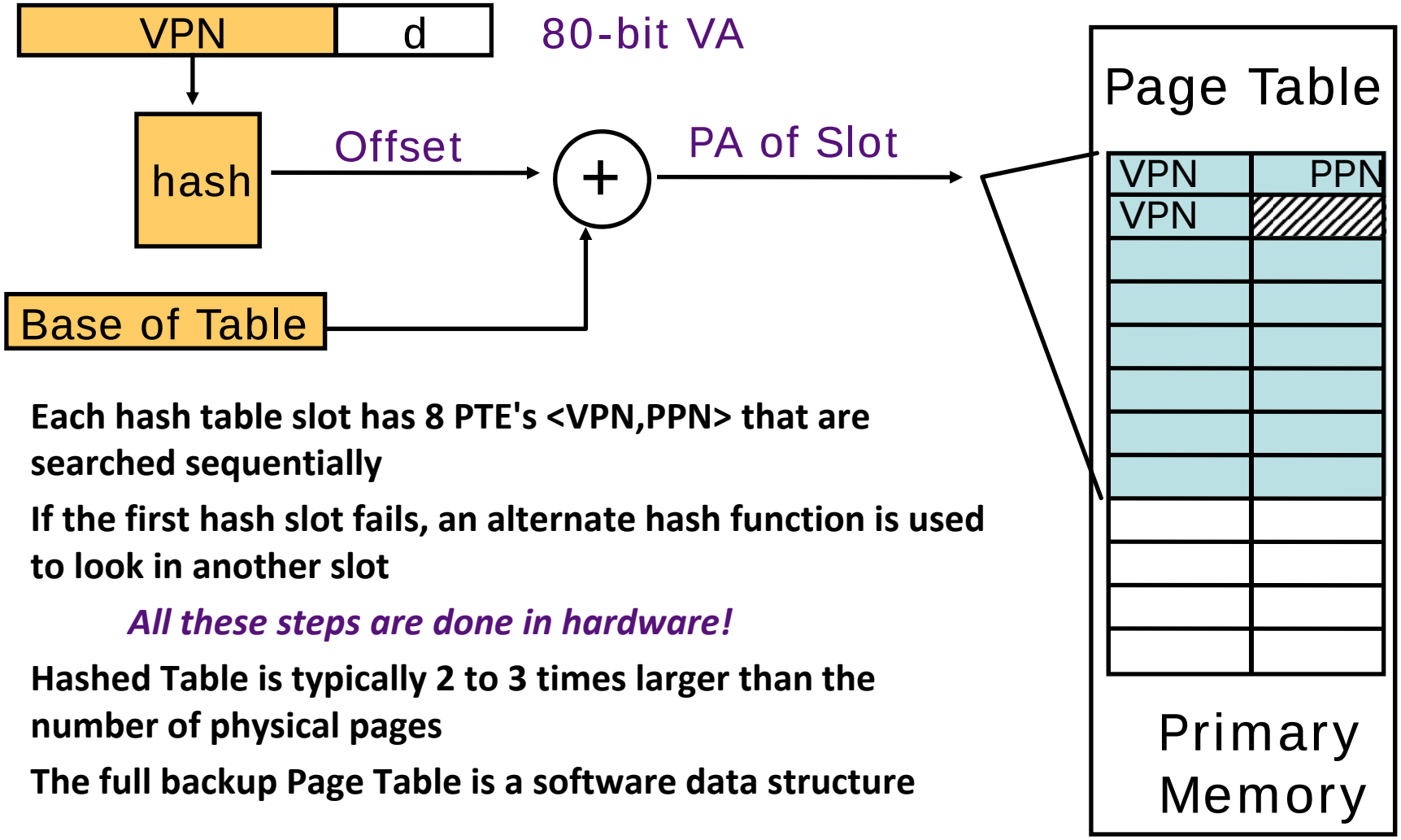
- **Level A maps users' address spaces into the global space providing privacy, protection, sharing etc.**
- **Level B provides demand-paging for the large global system address space**
- **Level A and Level B translations may be kept in separate TLB's**

Hashed Page Table Walk:

PowerPC Two-level, Segmented Addressing



Power PC: Hashed Page Table



- **Each hash table slot has 8 PTE's <VPN,PPN> that are searched sequentially**
- **If the first hash slot fails, an alternate hash function is used to look in another slot**

  All these steps are done in hardware!
- **Hashed Table is typically 2 to 3 times larger than the number of physical pages**
- **The full backup Page Table is a software data structure**

Virtual Memory Use Today - 1

- **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**
- **Vector supercomputers have translation and protection but not demand-paging**
  - **Older Crays: base&bound, Japanese & Cray X1: pages**
  - **Don't 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 - 2

- **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!**
  - **Programs custom written for particular memory configuration in product**
  - **Difficult to implement restartable instructions for exposed architectures**

But where software demands are more complex (e.g., cell phones, PDAs, routers), even embedded devices have TLBs!