Operating Systems
(Honor Track)

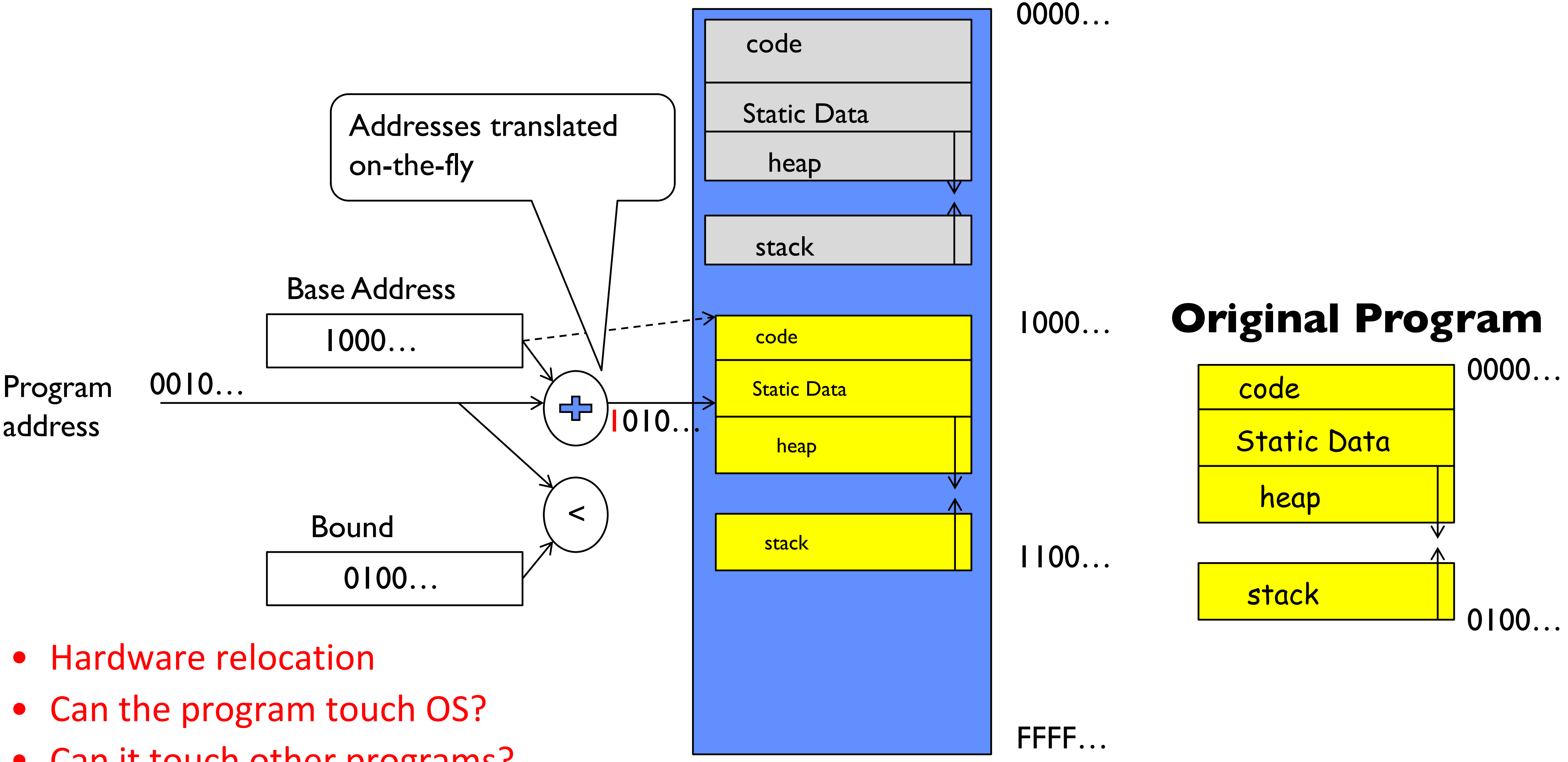
Memory 2: Virtual Memory (Con’t), Caching and TLBs

Xin Jin

Spring 2023

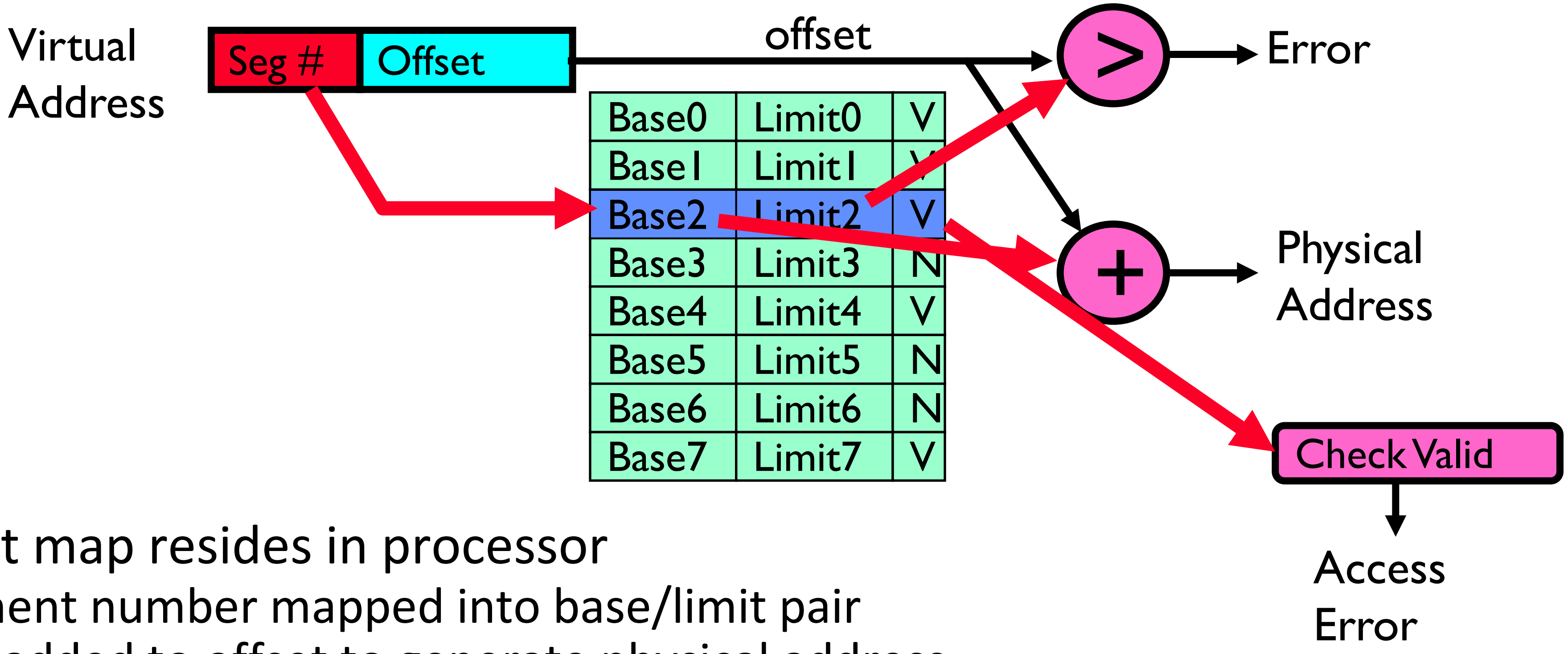
Acknowledgments: Ion Stoica, Berkeley CS 162

Recap: Base and Bound (with Translation)



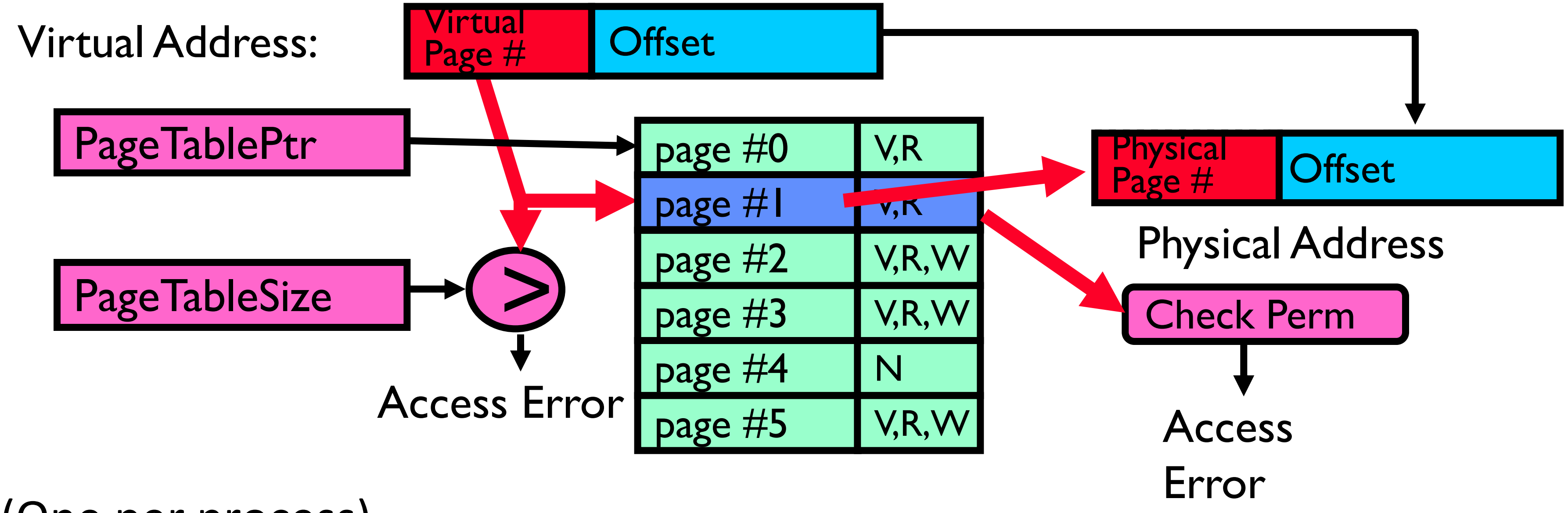
- Hardware relocation
- Can the program touch OS?
- Can it touch other programs?

Recap: Implementation of Multi-Segment Model



- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - » x86 Example: mov [es:bx],ax.
- What is "V/N" (valid / not valid)?
  - Can mark segments as invalid; requires check as well

Recap: How to Implement Simple Paging?



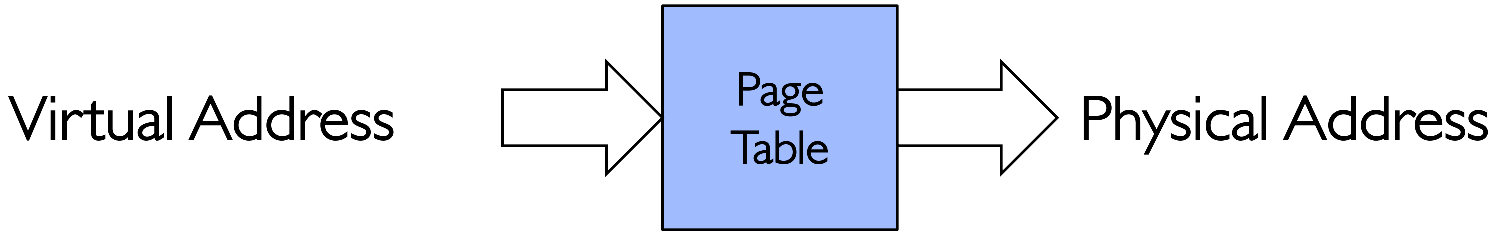
- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc.)
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

Recap: Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit
- What provides protection here?
  - Translation (per process) *and* dual-mode!
  - Can't let process alter its own page table!
- Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to share
  - Con: What if address space is sparse?
    - » E.g., on UNIX, code starts at 0, stack starts at (2^{31}-1)
    - » With 4KB pages, need 1 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time $\Rightarrow$ would be nice to have working set of page table in memory
- Simple Page table is way too big!
  - Does it all need to be in memory?
  - How about multi-level paging?
  - or combining paging and segmentation

How to Structure a Page Table

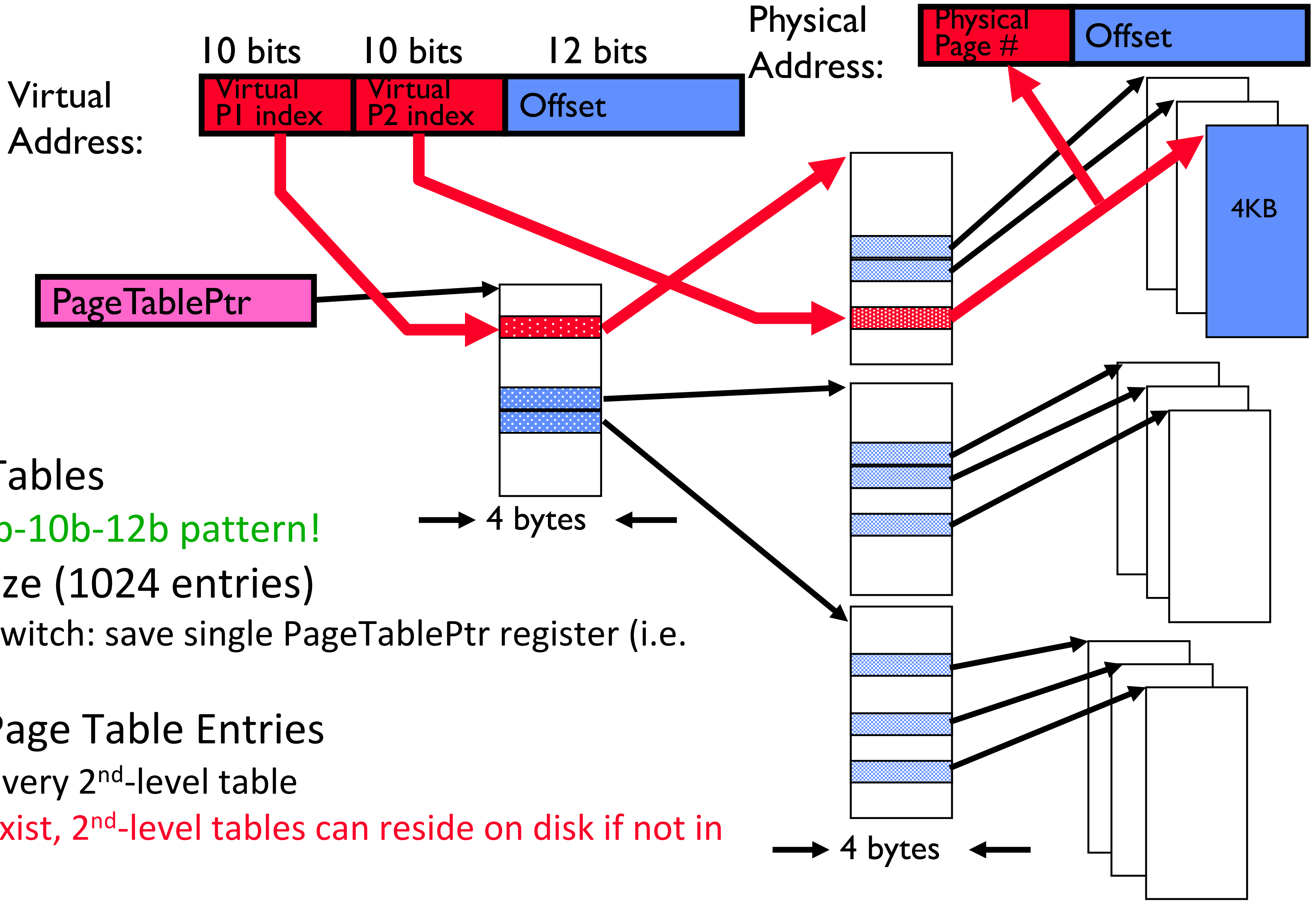
- Page Table is a *map* (function) from VPN to PPN



- Simple page table corresponds to a *very large* lookup table
  - VPN is index into table, each entry contains PPN

- What other map structures can you think of?
  - Trees?
  - Hash Tables?

Fix for sparse address space: The two-level page table



- Tree of Page Tables
  - "Magic" 10b-10b-12b pattern!
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register (i.e. CR3)
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

Page Table Entry (PTE)

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Flags: valid, read-only, read-write, write-only, etc.

- How do we use the PTE?
  - Invalid PTE can imply different things:
    - » Region of address space is actually invalid or
    - » Page/directory is just somewhere else than memory
  - Validity checked first
    - » OS can use other bits for location info

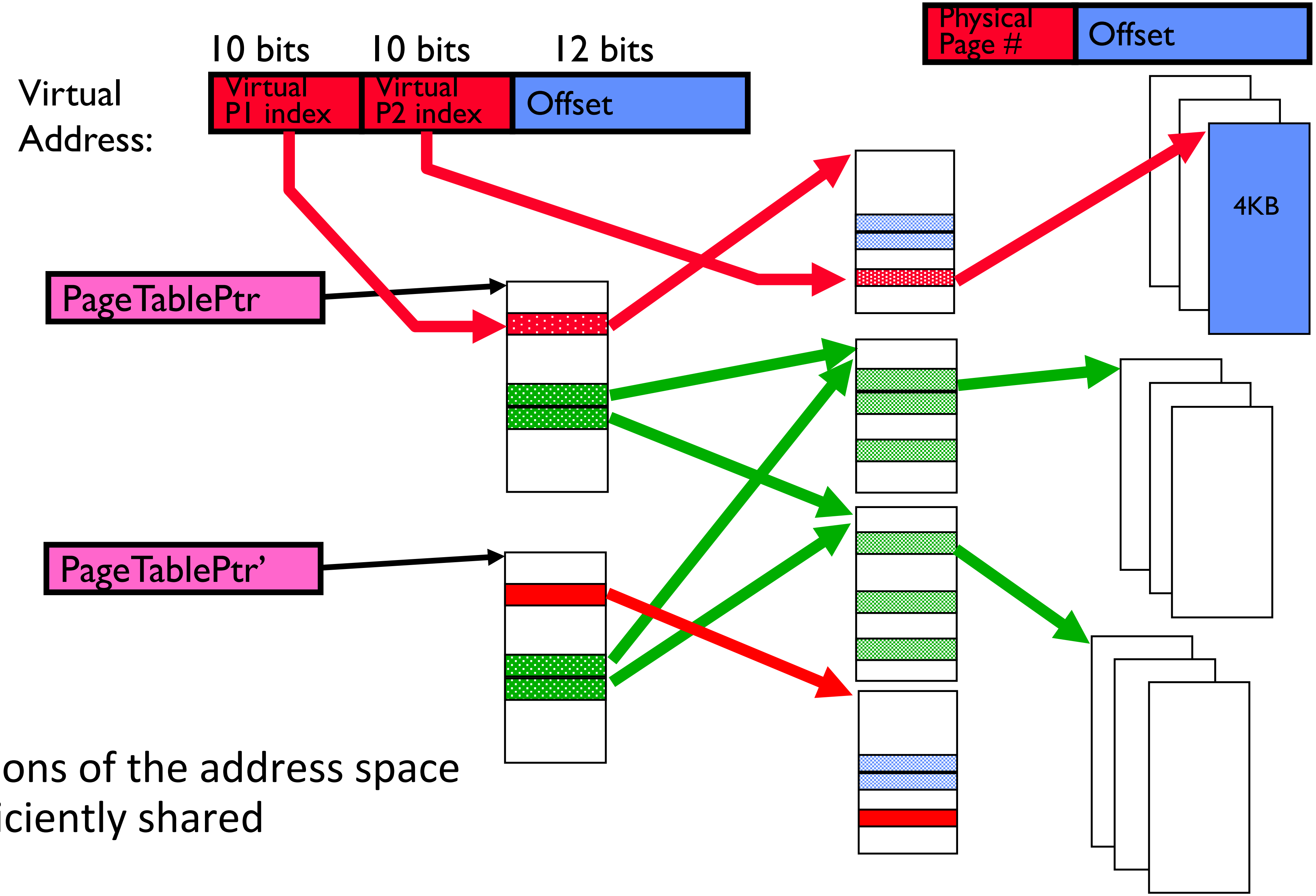
Page Table Entry (PTE)

- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid

- Usage Example: Copy on Write
  - UNIX fork gives *copy* of parent address space to child
    - » Address spaces disconnected after child created
  - How to do this cheaply?
    - » Make copy of parent's page tables (point at same memory)
    - » Mark entries in both sets of page tables as read-only
    - » Page fault on write creates two copies

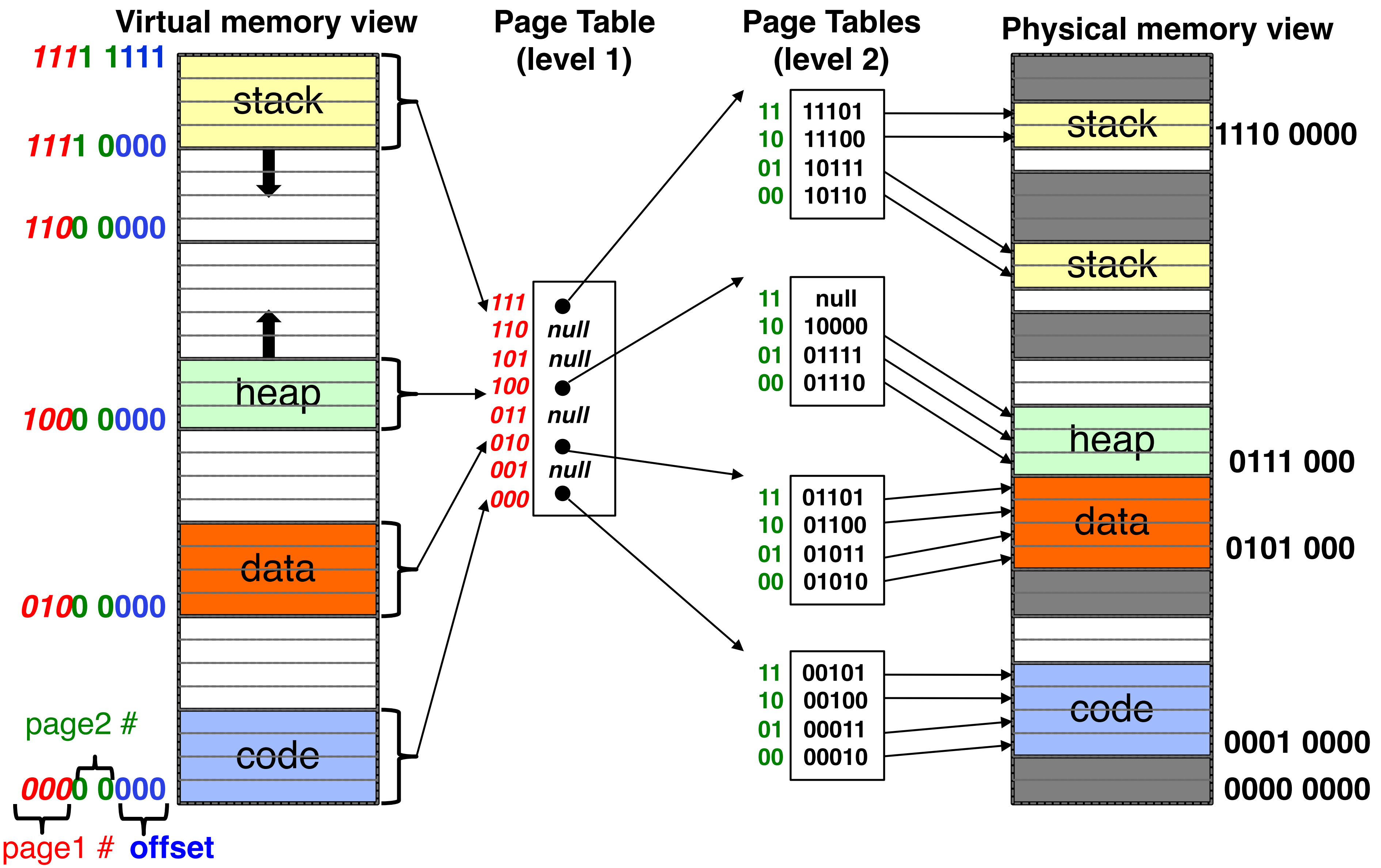
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

Sharing with multilevel page tables

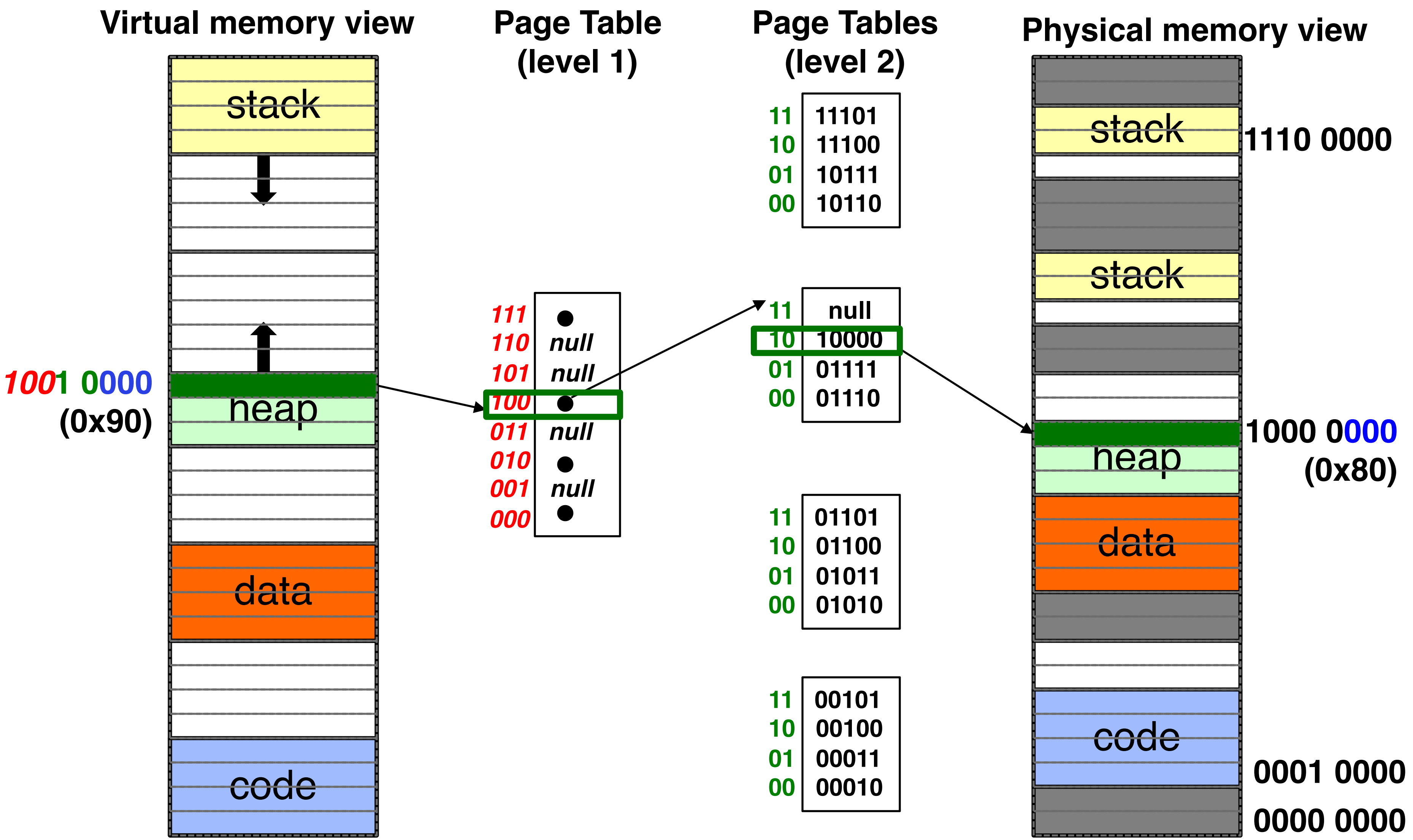


- Entire regions of the address space can be efficiently shared

Summary: Two-Level Paging

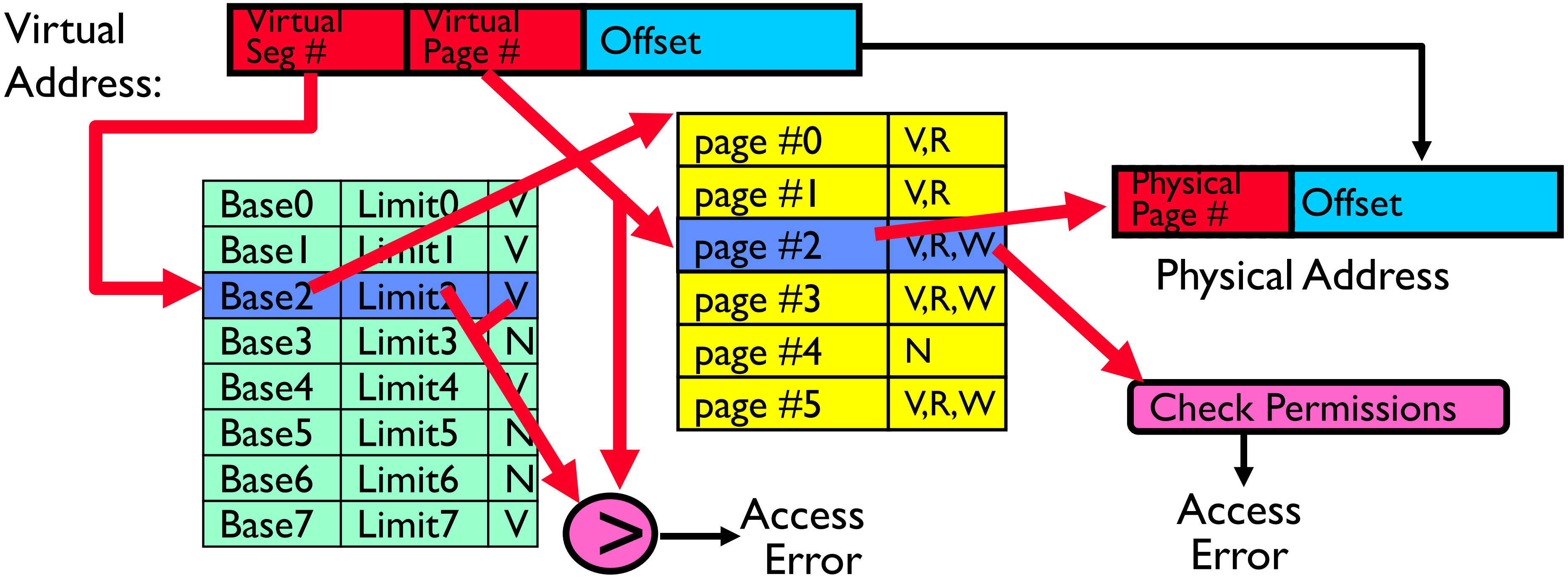


Summary: Two-Level Paging



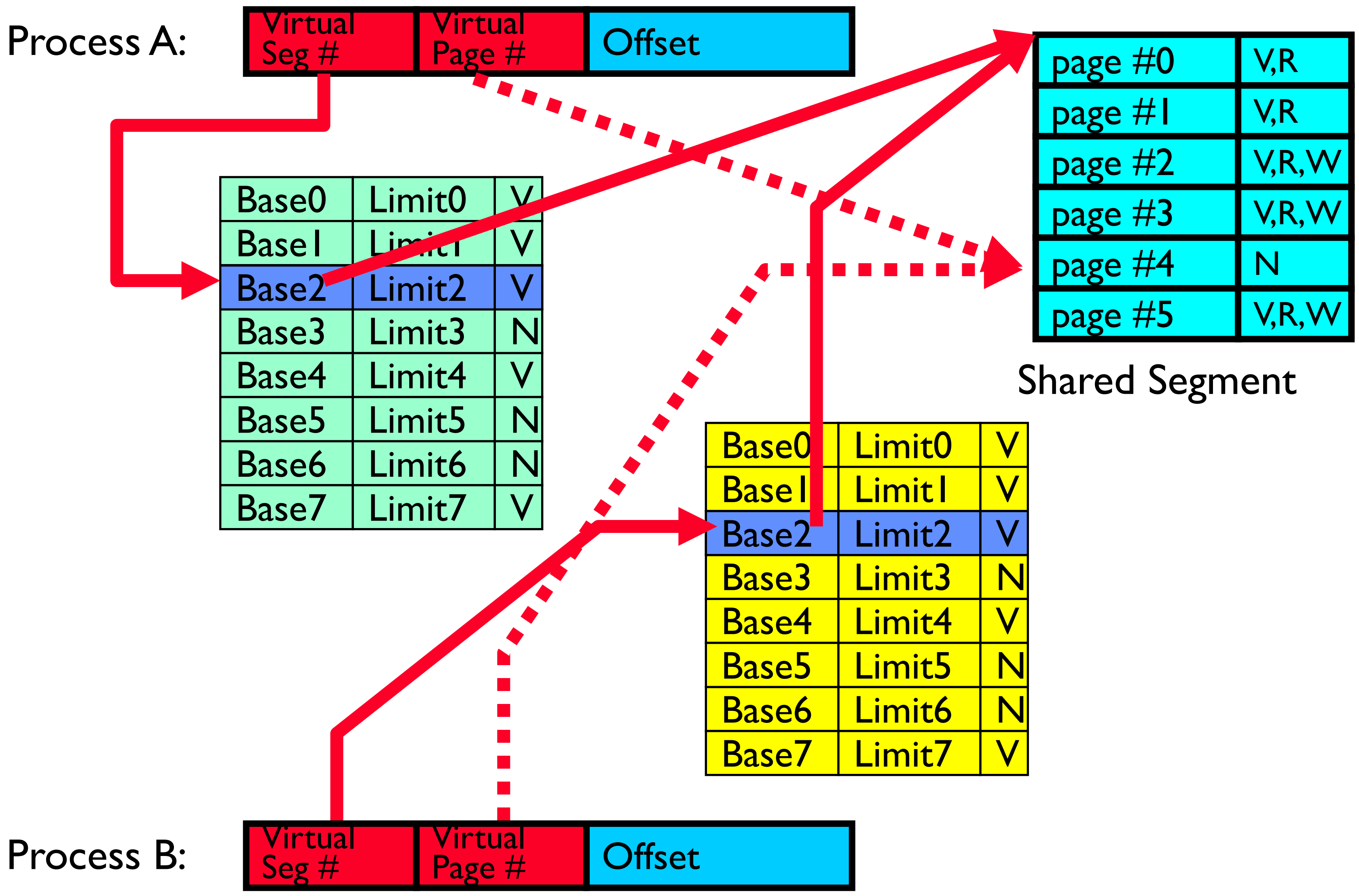
Multi-level Translation: Segments + Pages

- What about a tree of tables?
  - Lowest level page table ⇒ memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

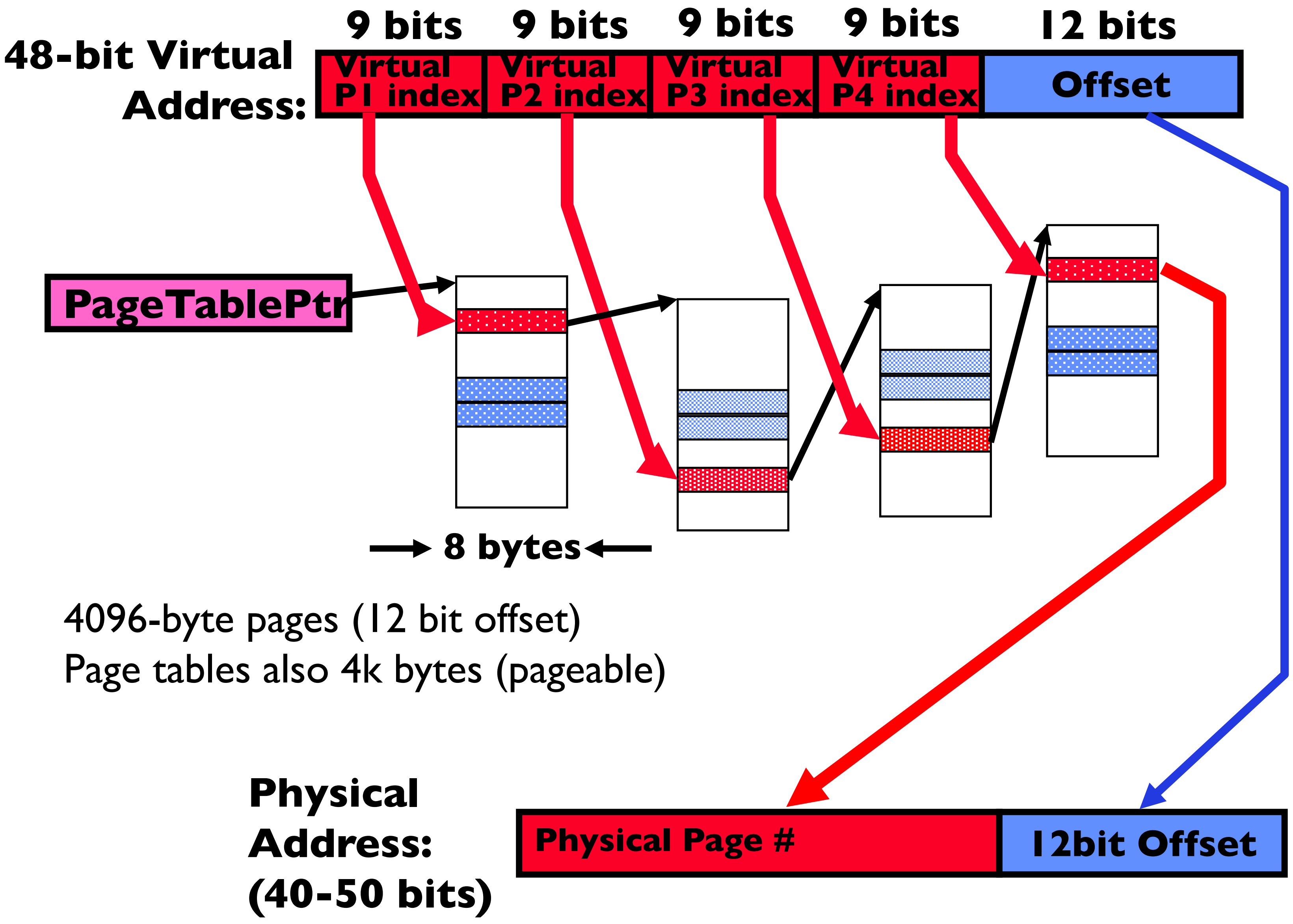


- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

What about Sharing (Complete Segment)?



X86_64: Four-level page table!



4096-byte pages (12 bit offset)
Page tables also 4k bytes (pageable)

IA64: 64bit addresses: Six-level page table?!?

64bit Virtual Address:

7 bits	9 bits	9 bits	9 bits	9 bits	9 bits	12 bits
Virtual P1 index	Virtual P2 index	Virtual P3 index	Virtual P4 index	Virtual P5 index	Virtual P6 index	Offset

No!

Too slow

Too many almost-empty tables

Group Discussion

- Topic: multi-level translation
  – What are the pros and cons of multi-level translation?

- Discuss in groups of two to three students
  – Each group chooses a leader to summarize the discussion
  – In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

Multi-level Translation Analysis

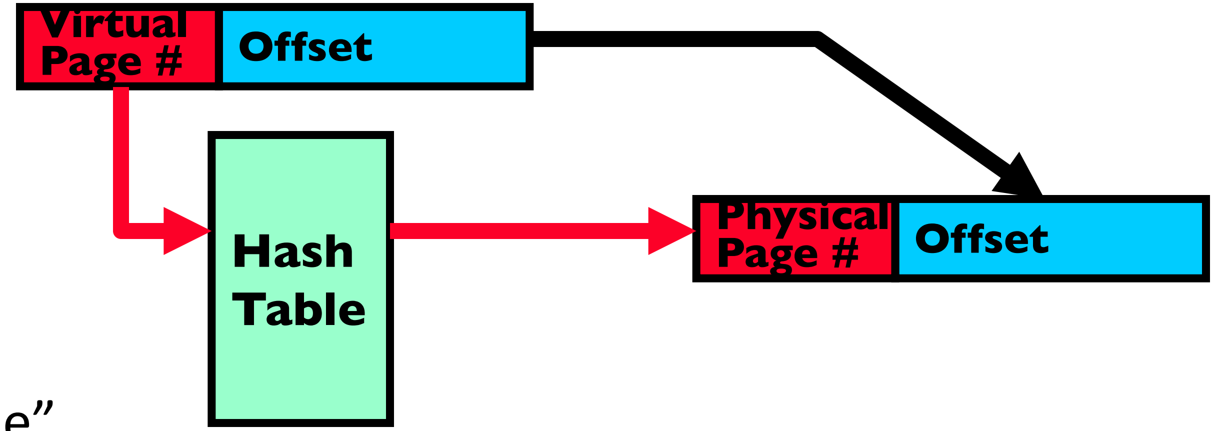
- Pros:
  - Only need to allocate as many page table entries as we need for application
    - » In other words, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - » Share at segment or page level (need additional reference counting)
- Cons:
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables need to be contiguous
    - » However, the 10b-10b-12b configuration keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    - » Seems very expensive!

Recall: Dual-Mode Operation

- Can a process modify its own translation tables? NO!
  - If it could, could get access to all of physical memory (no protection!)
- To Assist with Protection, Hardware provides at least two modes (Dual-Mode Operation):
  - “Kernel” mode (or “supervisor” or “protected”)
  - “User” mode (Normal program mode)
  - Mode set with bit(s) in control register only accessible in Kernel mode
  - Kernel can easily switch to user mode; User program must invoke an exception of some sort to get back to kernel mode
- Note that x86 model actually has more modes:
  - Traditionally, four “rings” representing priority; most OSes use only two:
    - » Ring 0 $\Rightarrow$ Kernel mode, Ring 3 $\Rightarrow$ User mode
    - » Called “Current Privilege Level” or CPL
  - Newer processors have additional mode for hypervisor (“Ring -1”)
- Certain operations restricted to Kernel mode:
  - Modifying page table base, and segment descriptor tables
    - » Have to transition into Kernel mode before you can change them!
  - Also, all page-table pages must be mapped only in kernel mode

Alternative: Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - » Much of process space may be out on disk or not in use



- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
    - » PowerPC, UltraSPARC, IA64
- Cons:
  - Complexity of managing hash chains: Often in hardware!
  - Poor cache locality of page table

Group Discussion

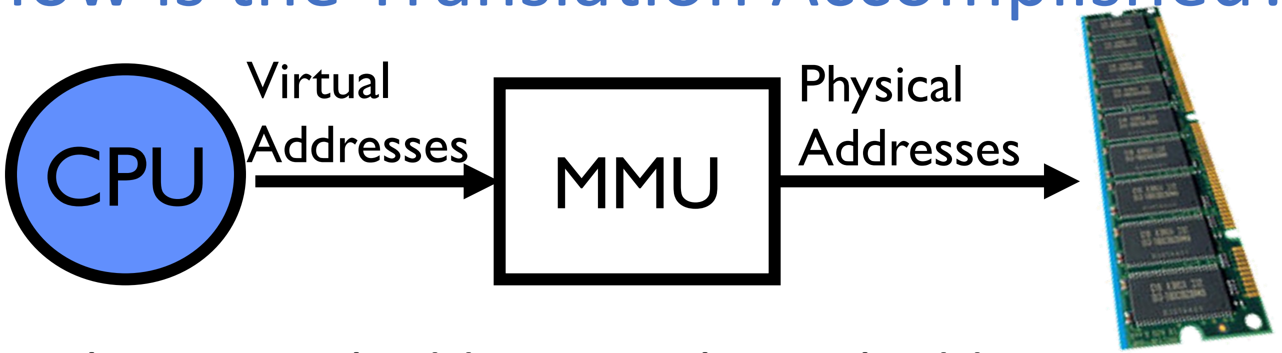
- Topic: simple segmentation, paging (single-level), paged segmentation, multi-level paging, inverted page tables
  - What are the pros and cons of each solution?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

Address Translation Comparison

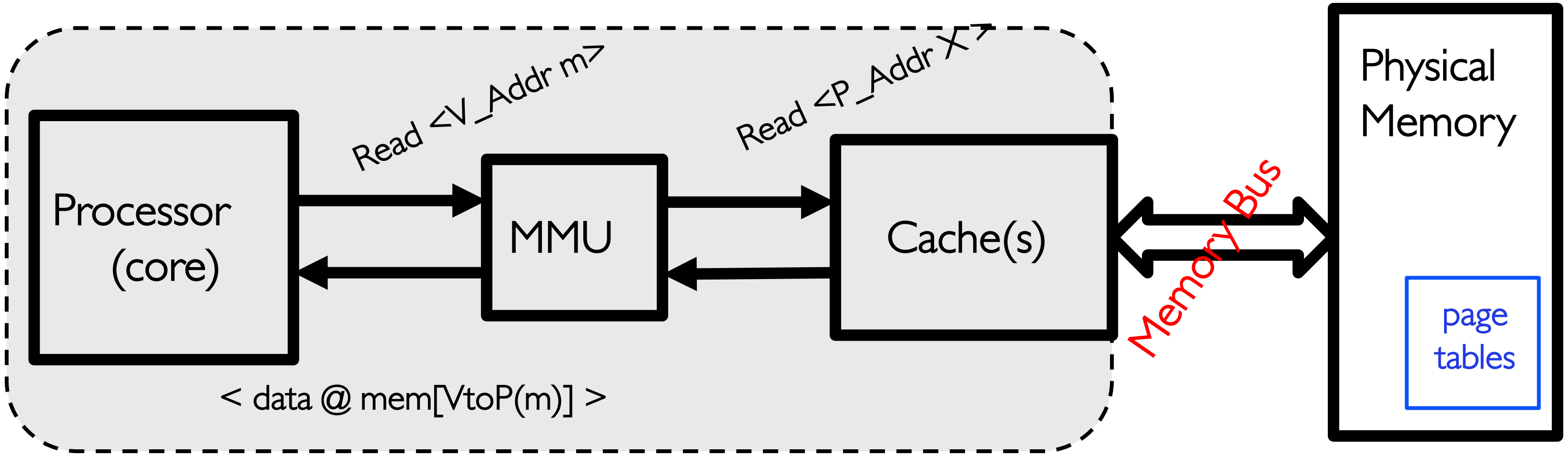
<table>
<tr><th></th><th>Advantages</th><th>Disadvantages</th></tr>
<tr><td>Simple Segmentation</td><td>Fast context switching (segment map maintained by CPU)</td><td>Internal/External fragmentation</td></tr>
<tr><td>Paging (Single-Level)</td><td>No external fragmentation
Fast and easy allocation</td><td>Large table size (~ virtual memory)
Internal fragmentation</td></tr>
<tr><td>Paged Segmentation</td><td rowspan="2">Table size ~ # of pages in virtual memory
Fast and easy allocation</td><td rowspan="2">Multiple memory references per page access</td></tr>
<tr><td>Multi-Level Paging</td></tr>
<tr><td>Inverted Page Table</td><td>Table size ~ # of pages in physical memory</td><td>Hash function more complex
No cache locality of page table</td></tr>
</table>

How is the Translation Accomplished?



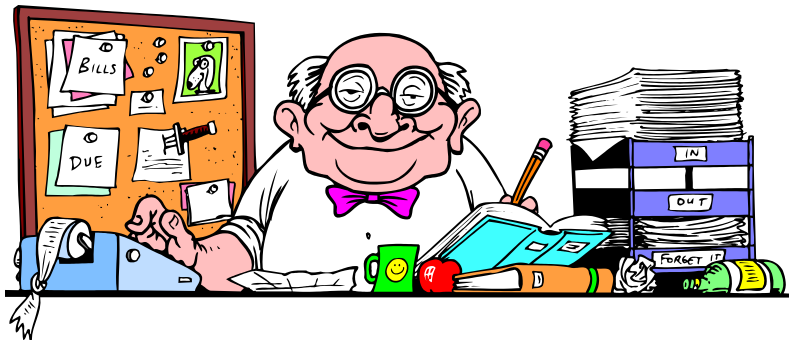
- The MMU must translate virtual address to physical address on:
  - Every instruction fetch
  - Every load
  - Every store
- What does the MMU need to do to translate an address?
  - 1-level Page Table
    - » Read PTE from memory, check valid, merge address
    - » Set "accessed" bit in PTE, Set "dirty bit" on write
  - 2-level Page Table
    - » Read and check first level
    - » Read, check, and update PTE
  - N-level Page Table …
- MMU does ***page table Tree Traversal*** to translate each address

Where and What is the MMU ?



- The processor requests READ Virtual-Address to memory system
  – Through the MMU to the cache (to the memory)
- Some time later, the memory system responds with the data stored at the physical address (resulting from virtual → physical) translation
  – Fast on a cache hit, slow on a miss
- So what is the MMU doing?
- On every reference (I-fetch, Load, Store) read (multiple levels of) page table entries to get physical frame or FAULT
  – Through the caches to the memory
  – Then read/write the physical location

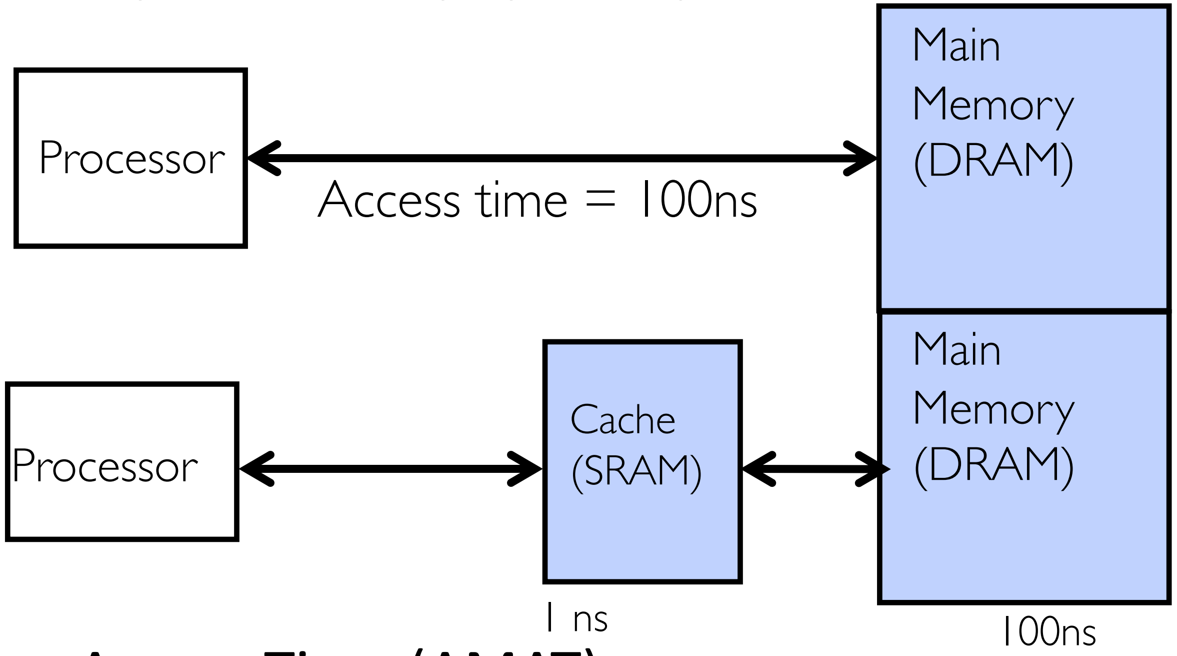
ICS: Caching Concept



- Cache: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many techniques used today to make computers fast
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Only good if:
  - Frequent case frequent enough
  - Infrequent case not too expensive
- Important measure: Average Access time =
  (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

ICS: In Machine Structures…

- Caching is the key to memory system performance



Average Memory Access Time (AMAT)

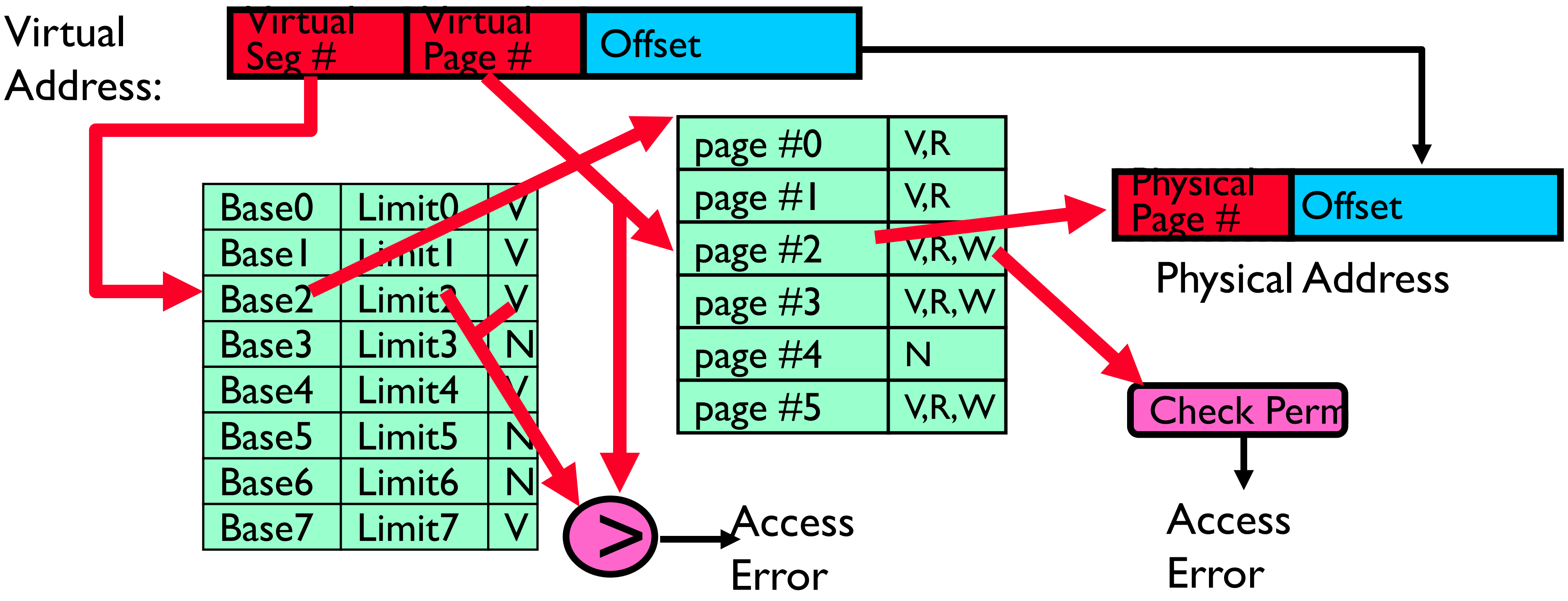
= (Hit Rate x HitTime) + (Miss Rate x MissTime)

Where HitRate + MissRate = 1

HitRate = 90% => AMAT = (0.9 x 1) + (0.1 x 101)=11.1 ns

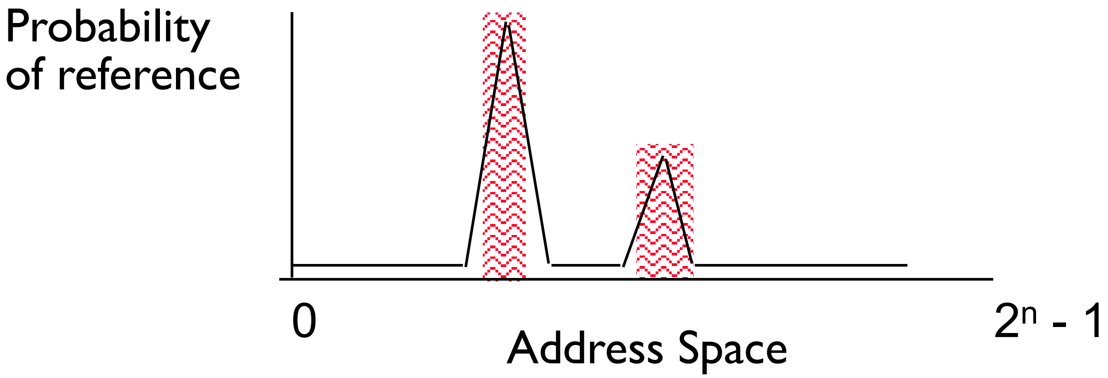
HitRate = 99% => AMAT = (0.99 x 1) + (0.01 x 101)=2.01 ns

Another Major Reason to Deal with Caching

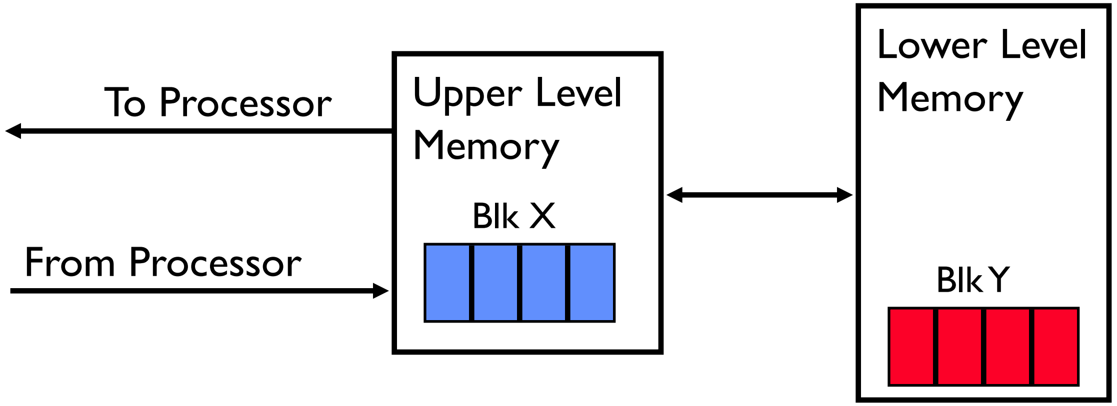


- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Solution? Cache translations!
  - Translation Cache: TLB ("Translation Lookaside Buffer")

Why Does Caching Help? Locality!

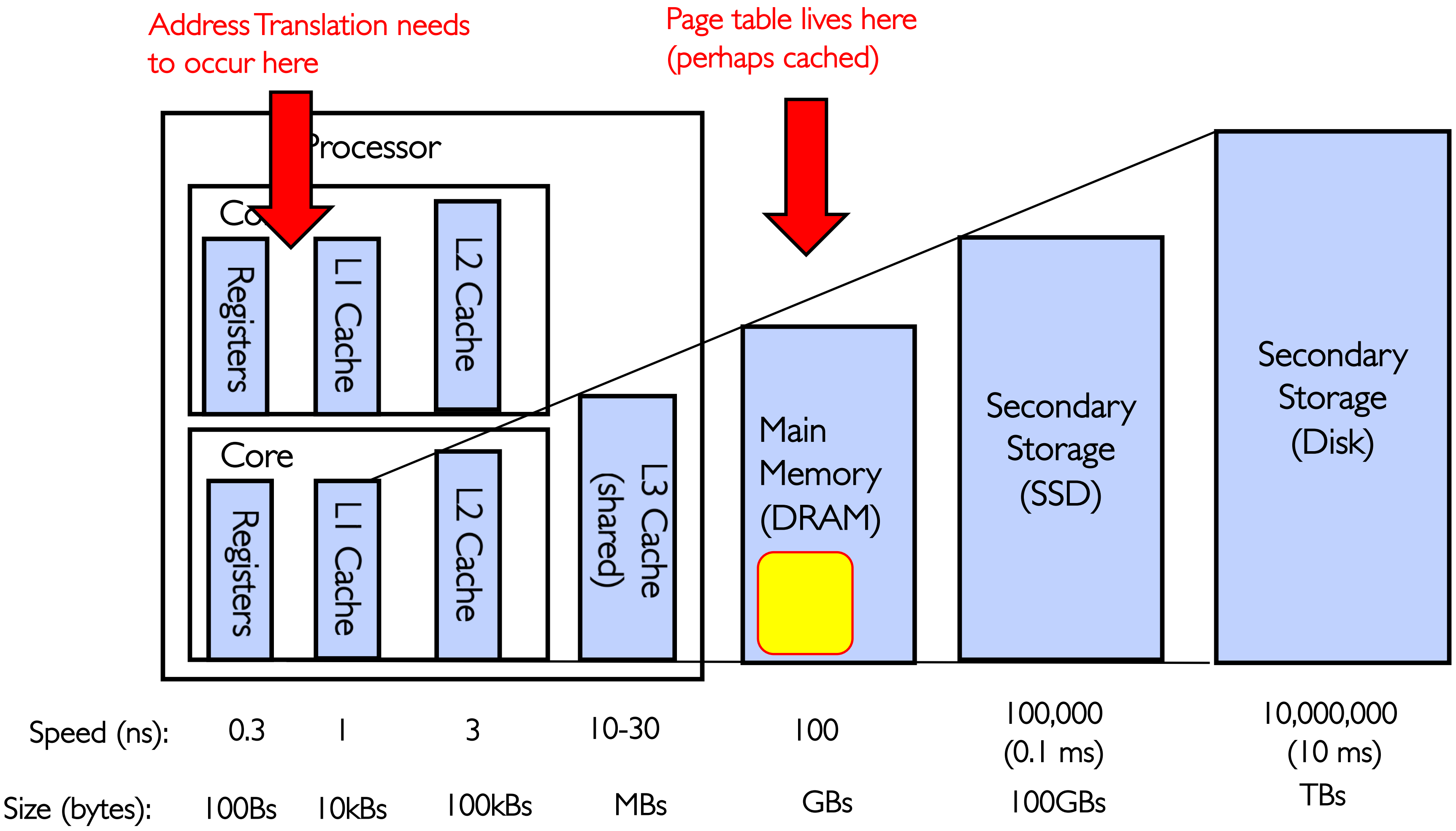


- Temporal Locality (Locality in Time):
  - Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - Move contiguous blocks to the upper levels



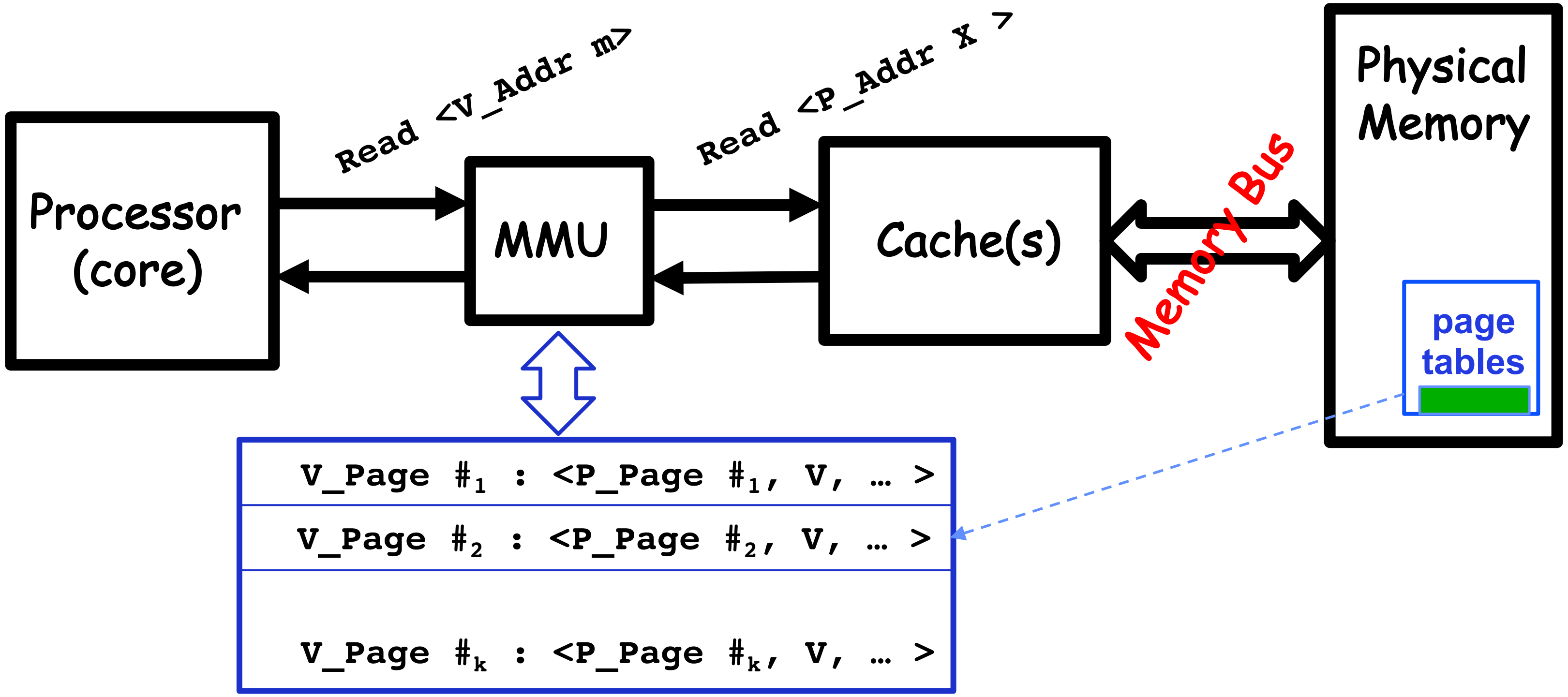
Recall: Memory Hierarchy

- Caching: Take advantage of the principle of locality to:
  - Present the illusion of having as much memory as in the cheapest technology
  - Provide average speed similar to that offered by the fastest technology



How do we make Address Translation Fast?

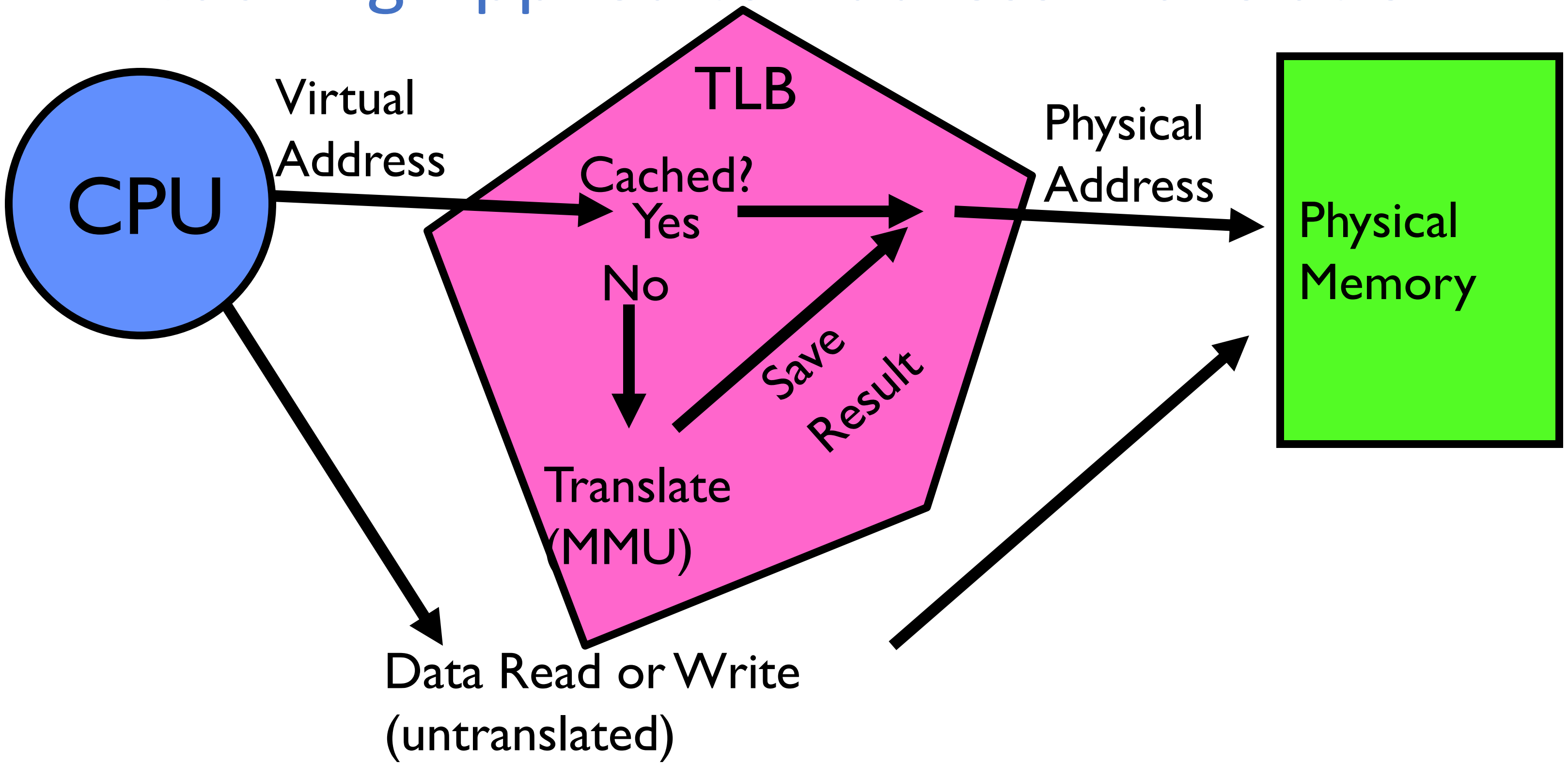
- Cache results of recent translations !
  - Different from a traditional cache
  - Cache Page Table Entries using Virtual Page # as the key



Translation Look-Aside Buffer

- Record recent Virtual Page # to Physical Page # translation
- If present, have the physical address without reading any of the page tables !!!
  - Even if the translation involved multiple levels
  - Caches the end-to-end result
- Was invented by Sir Maurice Wilkes – *prior to caches*
  - When you come up with a new concept, you get to name it!
- On a *TLB miss*, the page tables may be cached, so only go to memory when both miss

Caching Applied to Address Translation

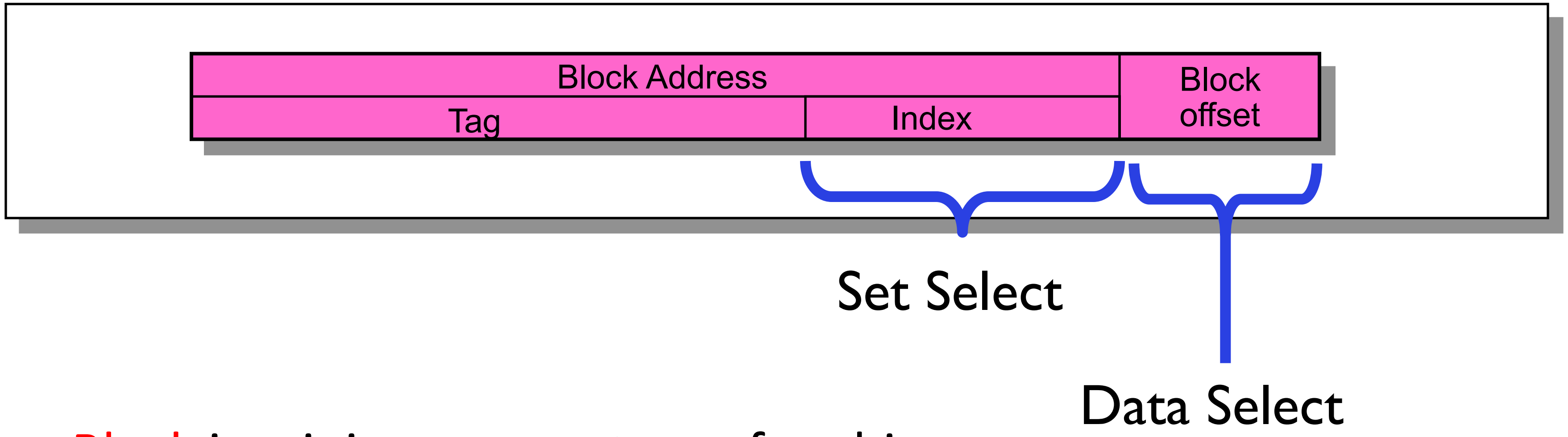


- Question is one of page locality: does it exist?
  – Instruction accesses spend a lot of time on the same page (since accesses sequential)
  – Stack accesses have definite locality of reference
  – Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  – Sure: multiple levels at different sizes/speeds

A Summary on Sources of Cache Misses

- Compulsory (cold start, first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Coherence (Invalidation): other process (e.g., I/O) updates memory

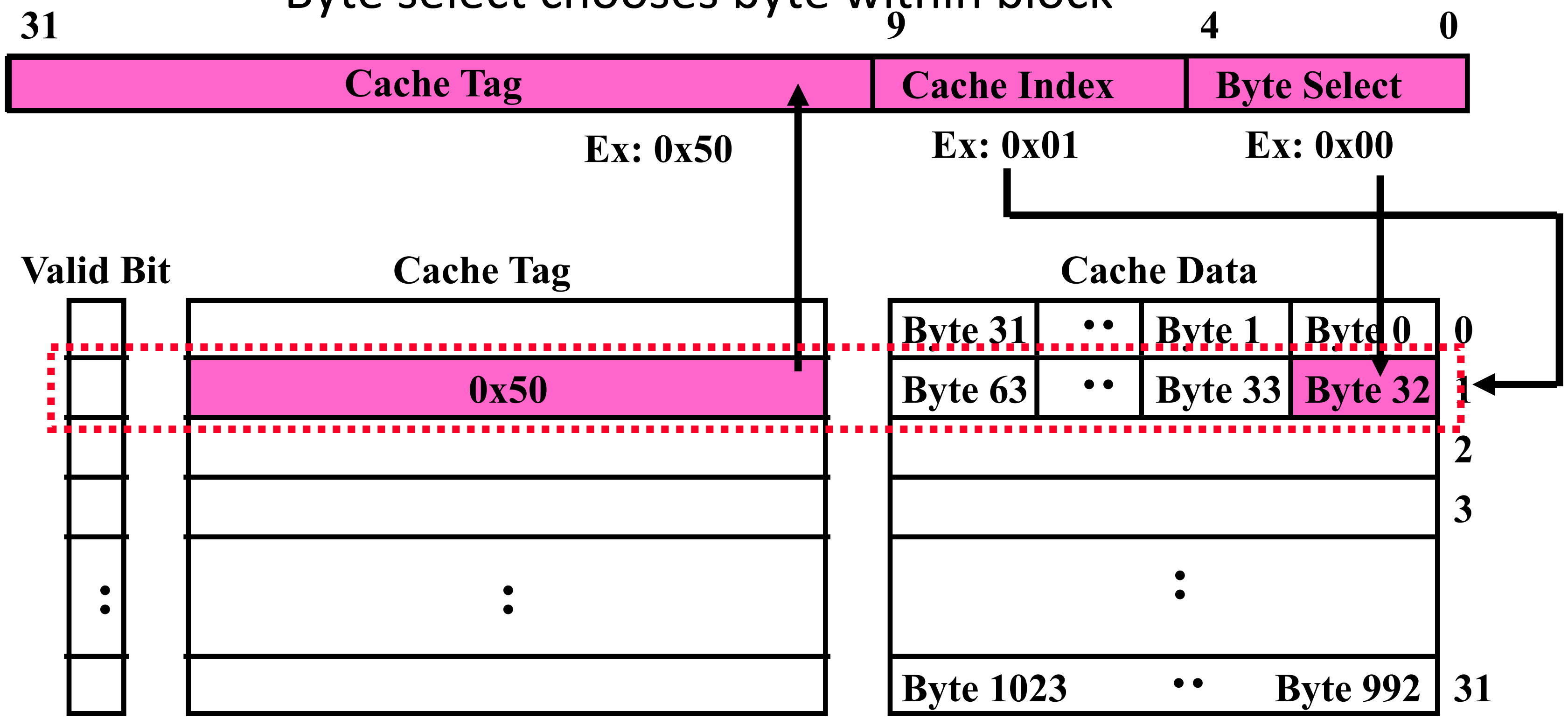
How is a Block found in a Cache?



- Block is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don't have data select field
- Index Used to Lookup Candidates in Cache
  - Index identifies the set
- Tag used to identify actual copy
  - If no candidates match, then declare cache miss

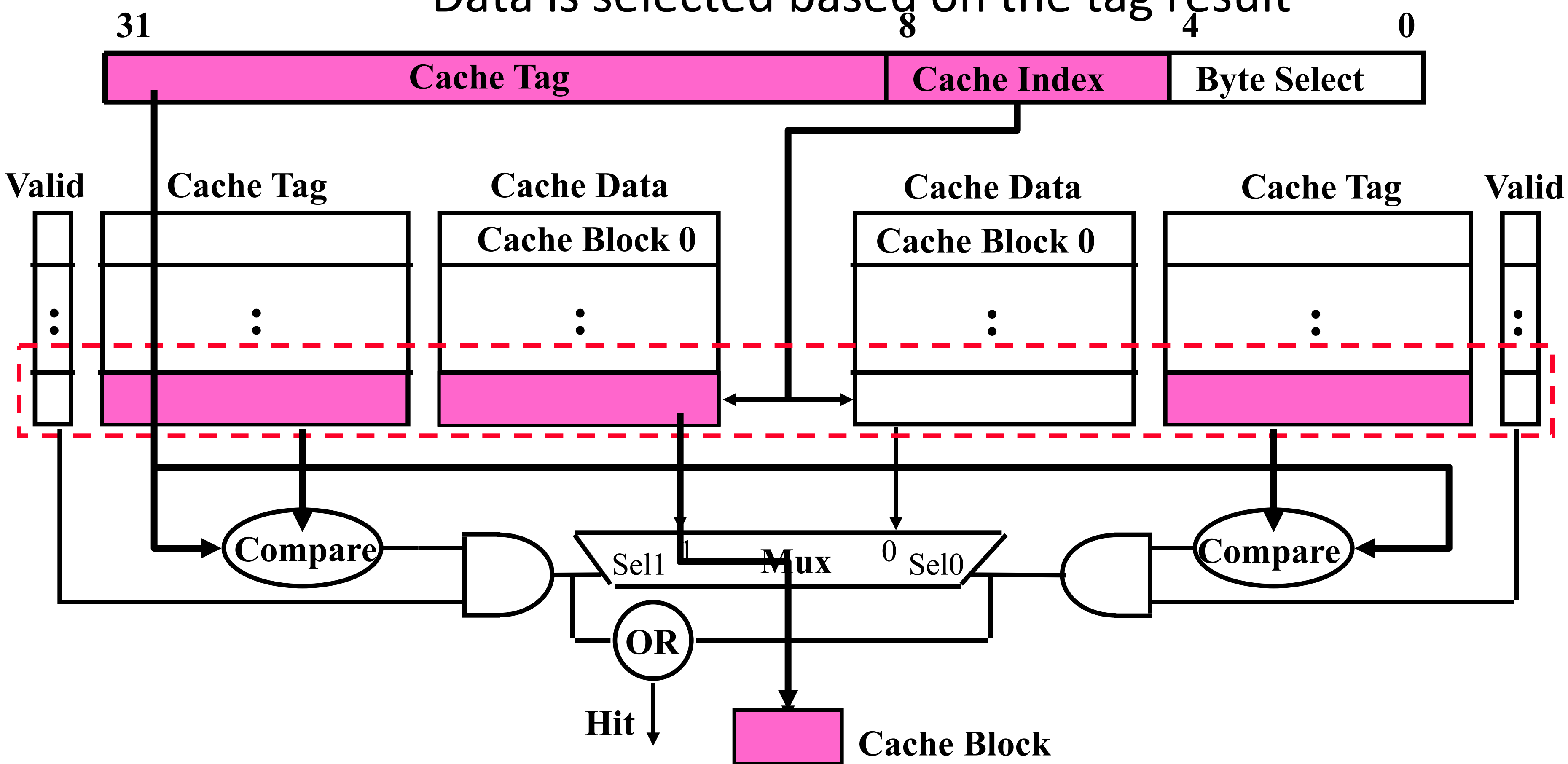
Review: Direct Mapped Cache

- Direct Mapped 2^N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2^M)
- Example: 1 KB Direct Mapped Cache with 32 B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block



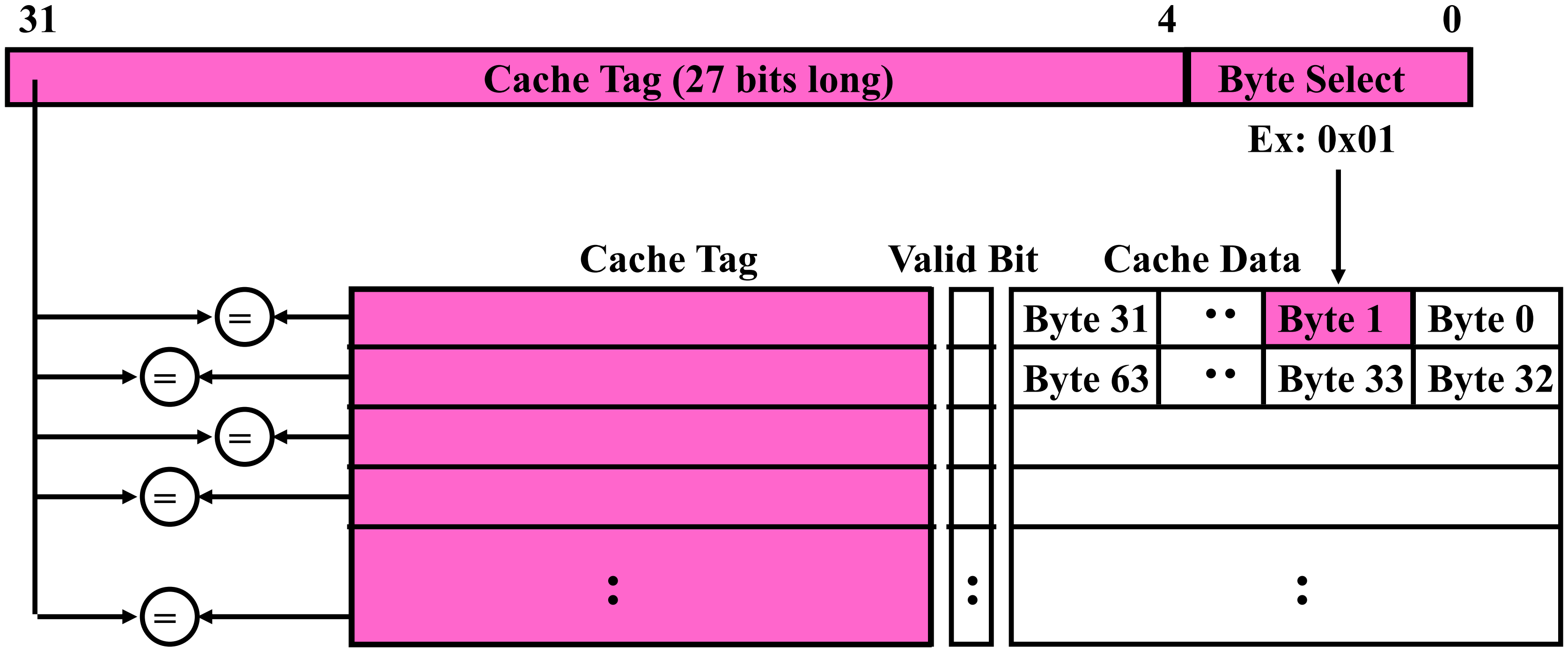
Review: Set Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result



Review: Fully Associative Cache

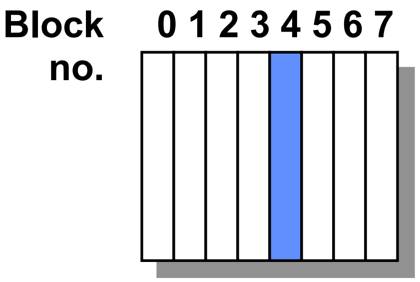
- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block



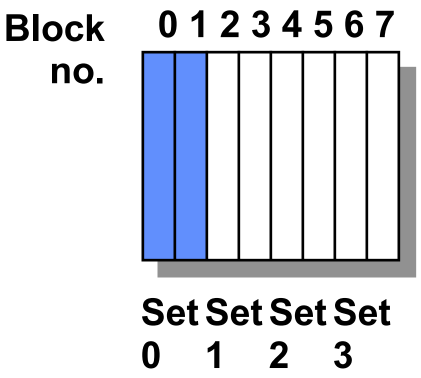
Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache

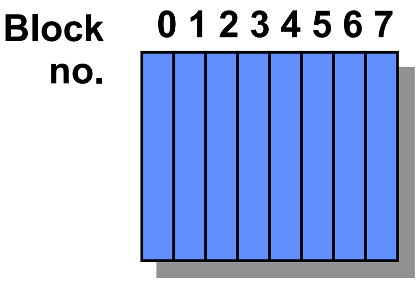
Direct mapped:
block 12 can go only into block 4 (12 mod 8)



Set associative:
block 12 can go anywhere in set 0 (12 mod 4)



Fully associative:
block 12 can go anywhere



Which block should be replaced on a miss?

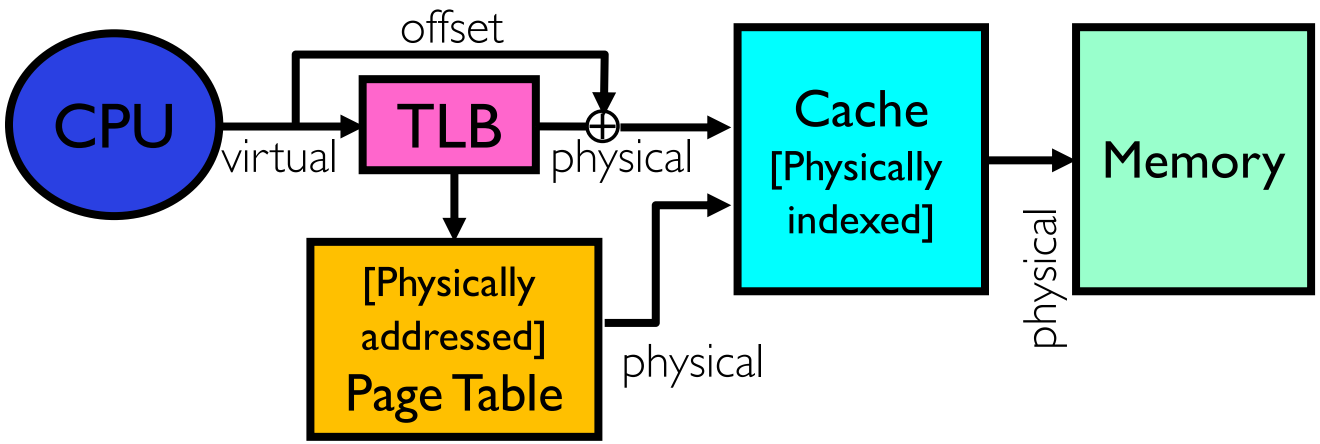
- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

Review: What happens on a write?

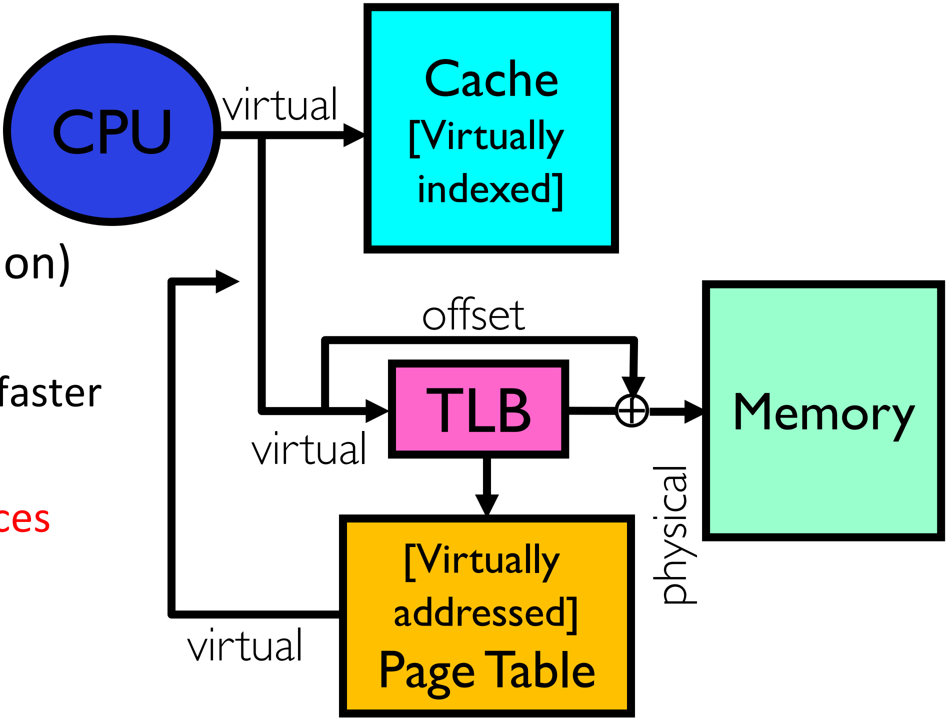
- Write through: The information is written to both the block in the cache and to the block in the lower-level memory
- Write back: The information is written only to the block in the cache
  - Modified cache block is written to main memory only when it is replaced
- Pros and Cons of each?
  - Write through:
    - » Pros: read misses cannot result in writes
    - » Cons: processor held up on writes
  - Write back:
    - » Pros: repeated writes not sent to DRAM
processor not held up on writes
    - » Cons: more complex
read miss may require writeback of dirty data

Physically-Indexed vs Virtually-Indexed Caches

- Physically-Indexed Caches
  - Address handed to cache *after translation*
  - Page Table holds *physical* addresses
  - Benefits:
    - » Every piece of data has single place in cache
    - » Cache can stay unchanged on context switch
  - *Challenges*:
    - » TLB is in critical path of lookup!
  - Pretty Common today (e.g., x86 processors)

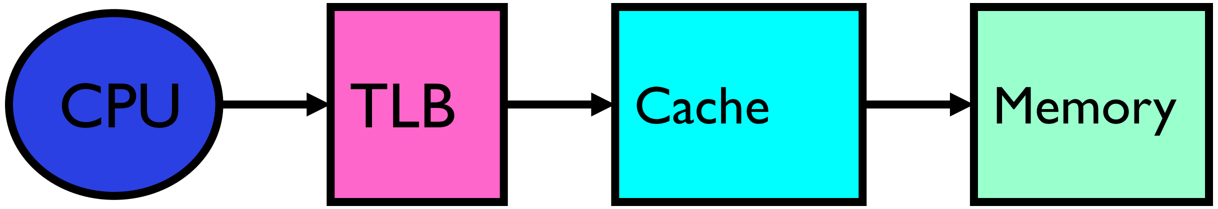


- Virtually-Indexed Caches
  - Address handed to cache *before translation*
  - Page Table holds *virtual* addresses (one option)
  - Benefits:
    - » TLB not in critical path of lookup, so can be faster
  - *Challenges:*
    - » Same data could be mapped in multiple places of cache
    - » May need to flush cache on context switch



- We will stick with Physically Addressed Caches for now!

What TLB Organization Makes Sense?

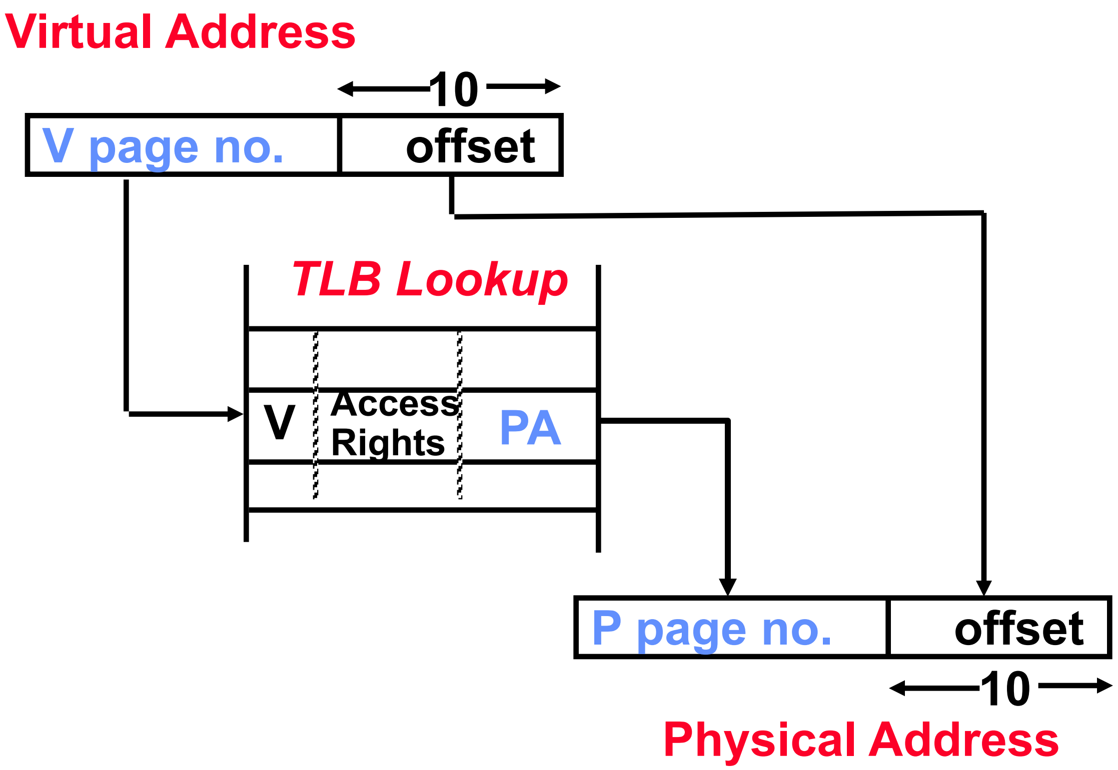


- Needs to be really fast
  - Critical path of memory access
    - » In simplest view: before the cache
    - » Thus, this adds to access time (reducing cache speed)
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high! (PT traversal)
  - Cost of Conflict (Miss Time) is high
  - Hit Time – dictated by clock cycle
- Thrashing: continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    - » First page of code, data, stack may map to same entry
    - » Need 3-way associativity at least?
  - What if use high order bits as index?
    - » TLB mostly unused for small programs

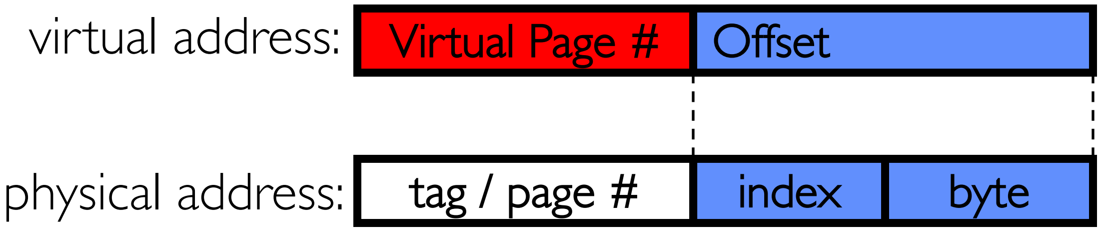
TLB organization: include protection

- How big does TLB actually have to be?
  - Usually small: 128-512 entries (larger now)
  - Not very big, can support higher associativity
- Small TLBs usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info
- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a "TLB Slice"

Reducing translation time for physically-indexed caches

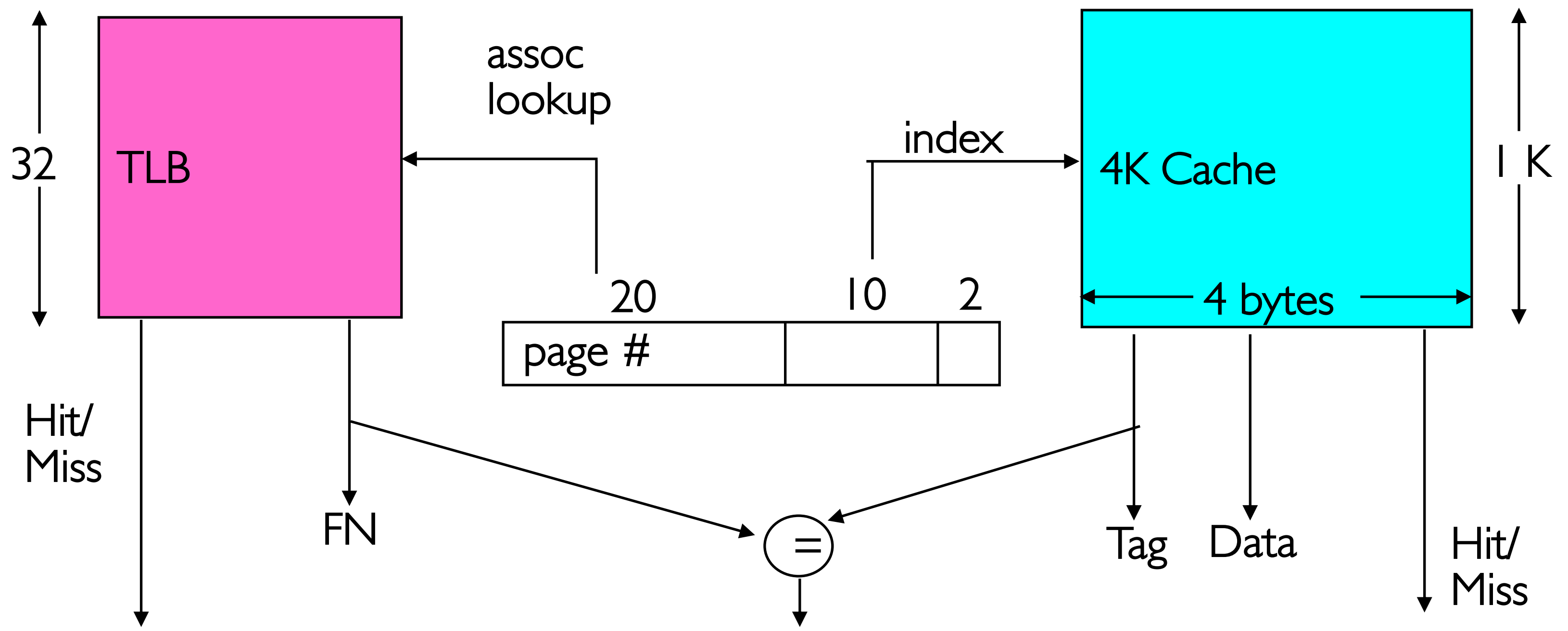


- As described, TLB lookup is in serial with cache lookup
  - Consequently, speed of TLB can impact speed of access to cache
- Machines with TLBs go one step further: overlap TLB lookup with cache access
  - Works because offset available early
  - Offset in virtual address exactly covers the “cache index” and “byte select”
  - Thus can select the cached byte(s) in parallel to perform address translation



Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:



- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else
- Another option: Virtual Caches would make this faster
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate TLB: simple but might be expensive
    - » What if switching frequently between processes?
  - Include ProcessID in TLB
    - » This is an architectural solution: needs hardware
- What if translation tables change?
  - For example, to move page from memory to disk or vice versa…
  - Must invalidate TLB entry!
    - » Otherwise, might think that page is still in memory!
  - Called “TLB Consistency”
- Aside: with Virtually-Indexed cache, need to flush cache!
  - Remember, everyone has their own version of the address “0”!

Summary (1/3)

- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory
- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- Inverted Page Table
  - Use of hash-table to hold translation entries
  - Size of page table ~ size of physical memory rather than size of virtual memory

Summary (2/3)

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - » Temporal Locality: Locality in Time
    - » Spatial Locality: Locality in Space
- Three (+1) Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity
  - Capacity Misses: increase cache size
  - Coherence Misses: Caused by external processors or I/O devices
- Cache Organizations:
  - Direct Mapped: single block per set
  - Set associative: more than one block per set
  - Fully associative: all entries equivalent

Summary (3/3)

- “Translation Lookaside Buffer” (TLB)
  - Small number of PTEs and optional process IDs (< 512)
  - Fully Associative (Since conflict misses expensive)
  - On TLB miss, page table must be traversed and if located PTE is invalid, cause Page Fault
  - On change in page table, TLB entries must be invalidated
  - TLB is logically in front of cache (need to overlap with cache access)