

CSI 62  
Operating Systems and  
Systems Programming  
Lecture 13

Address Translation (Con't),  
Caching and TLBs

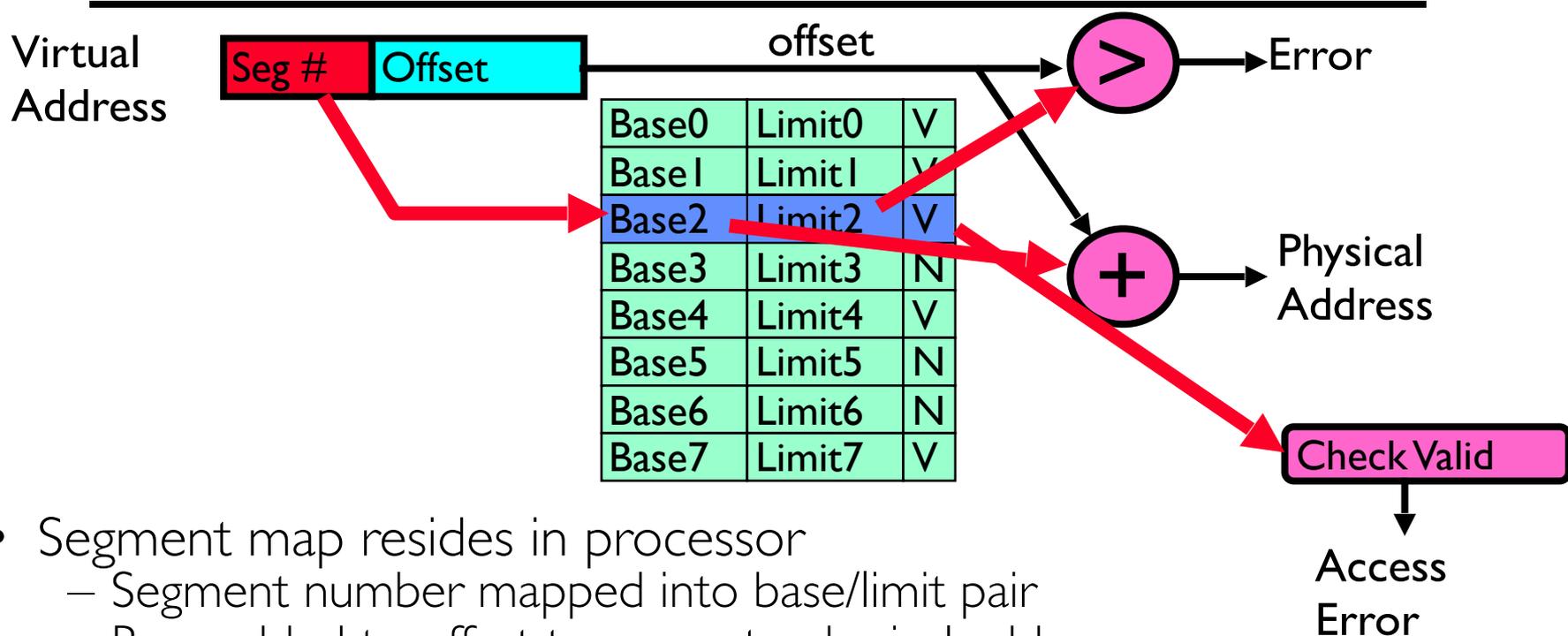
March 10<sup>th</sup>, 2020

Prof. John Kubiawicz

<http://cs162.eecs.Berkeley.edu>

*Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiawicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.*

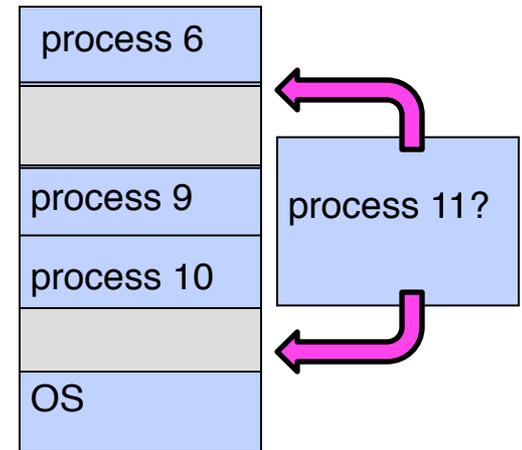
# Recall: Implementation of Multi-Segment Model



- 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

# Recall: Problems with Segmentation

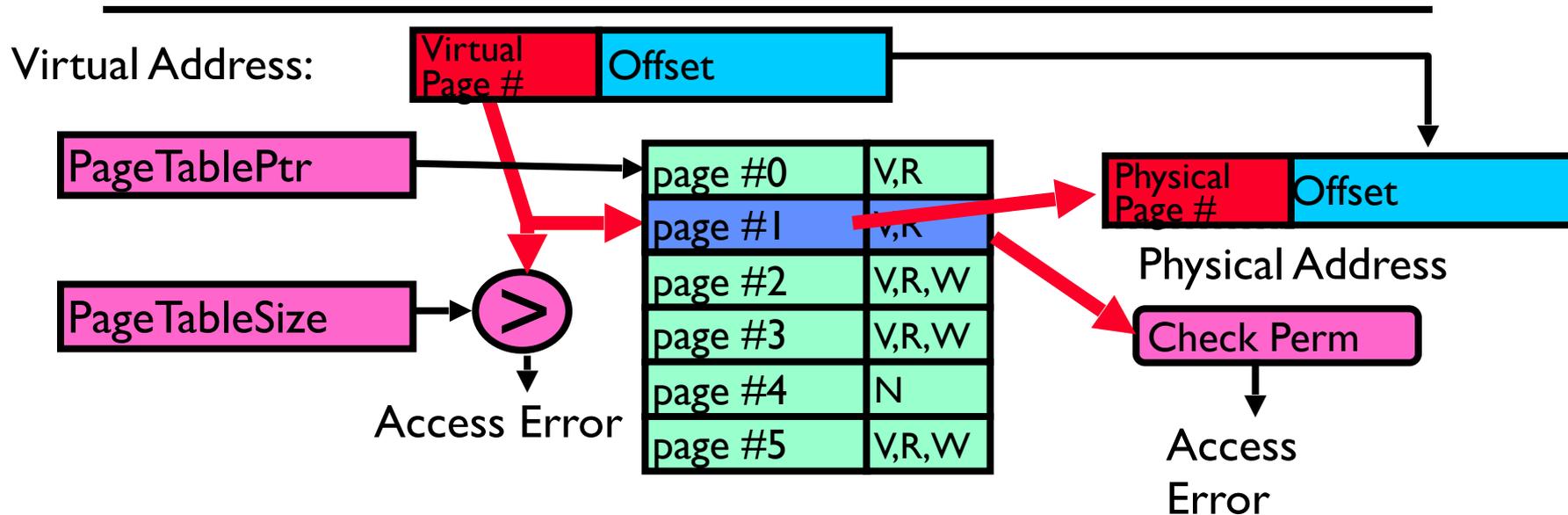
- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- **Fragmentation**: wasted space
  - **External**: free gaps between allocated chunks
  - **Internal**: don't need all memory within allocated chunks



# Paging: Physical Memory in Fixed Size Chunks

- Solution to fragmentation from segments?
  - Allocate physical memory in **fixed size** chunks (“pages”)
  - Every chunk of physical memory is equivalent
    - » Can use simple vector of bits to handle allocation:  
00110001110001101 ... 110010
    - » Each bit represents page of physical memory  
1  $\Rightarrow$  allocated, 0  $\Rightarrow$  free
- Should pages be as big as our previous segments?
  - No: Could lead to lots of **internal fragmentation**
    - » Typically have small pages (1K-16K)
    - » Consequently: need multiple pages/segment

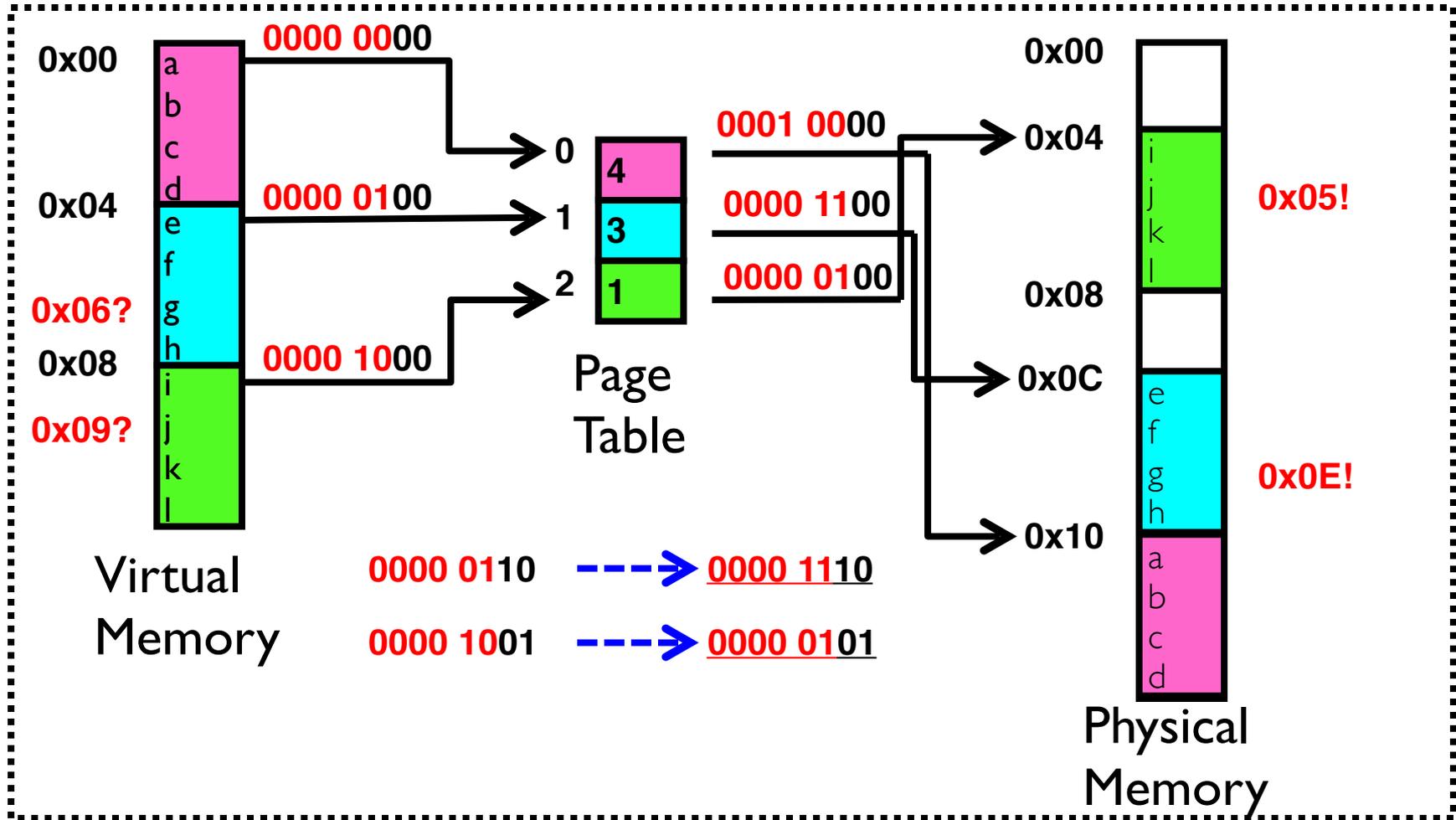
# How to Implement Simple Paging?



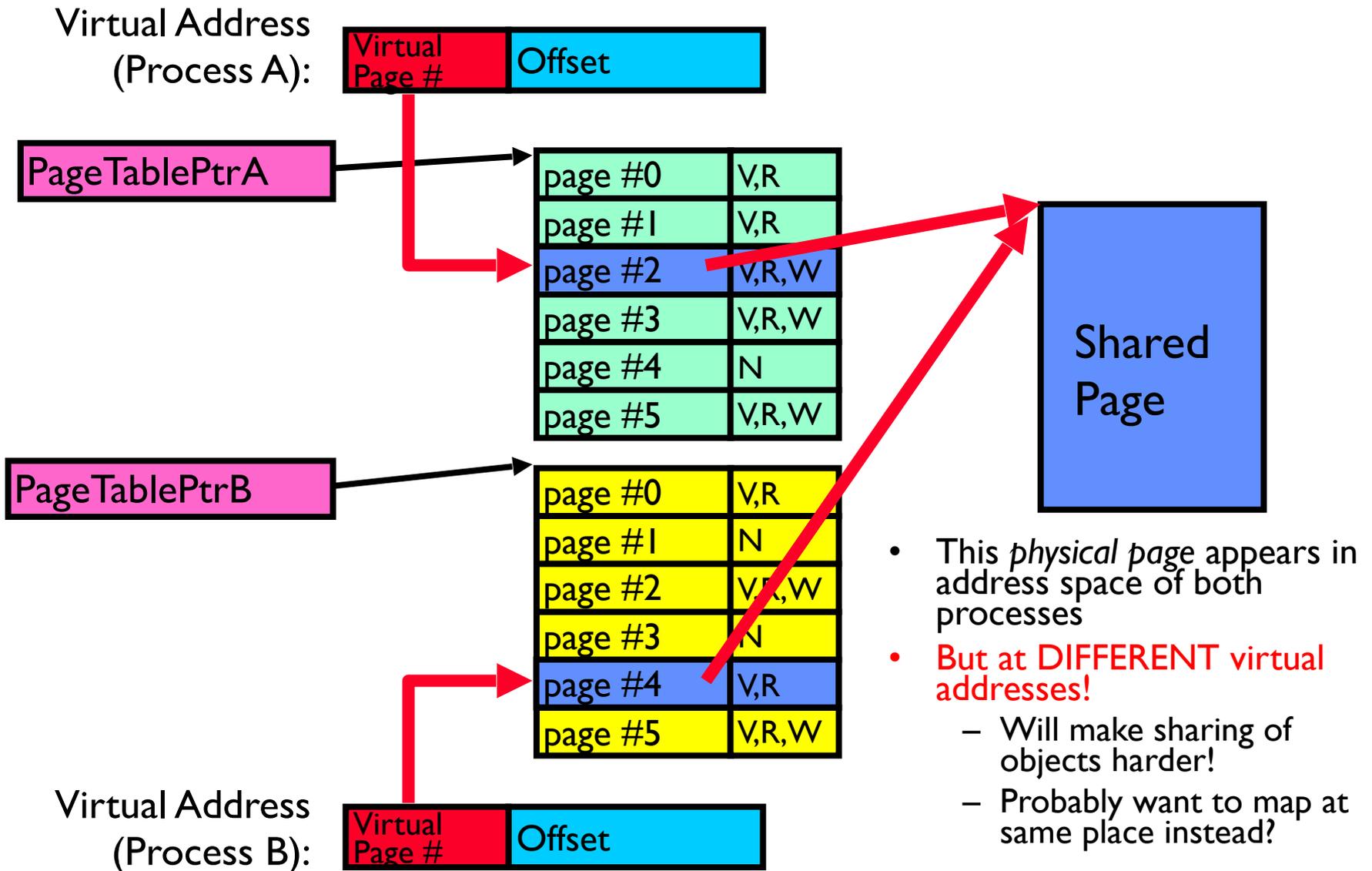
- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - » Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset  $\Rightarrow$  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

# Simple Page Table Example

Example (4 byte pages)



# What about Sharing?



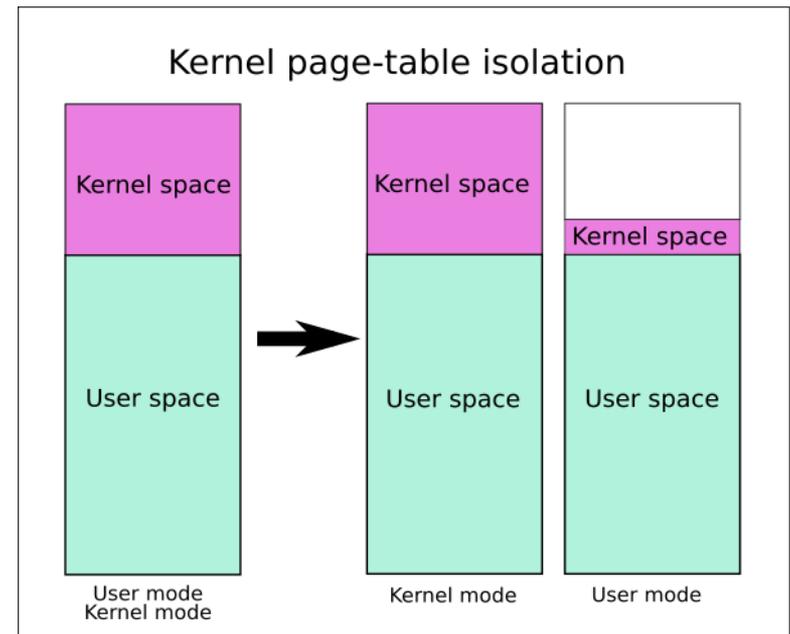
# Where is page sharing used ?

---

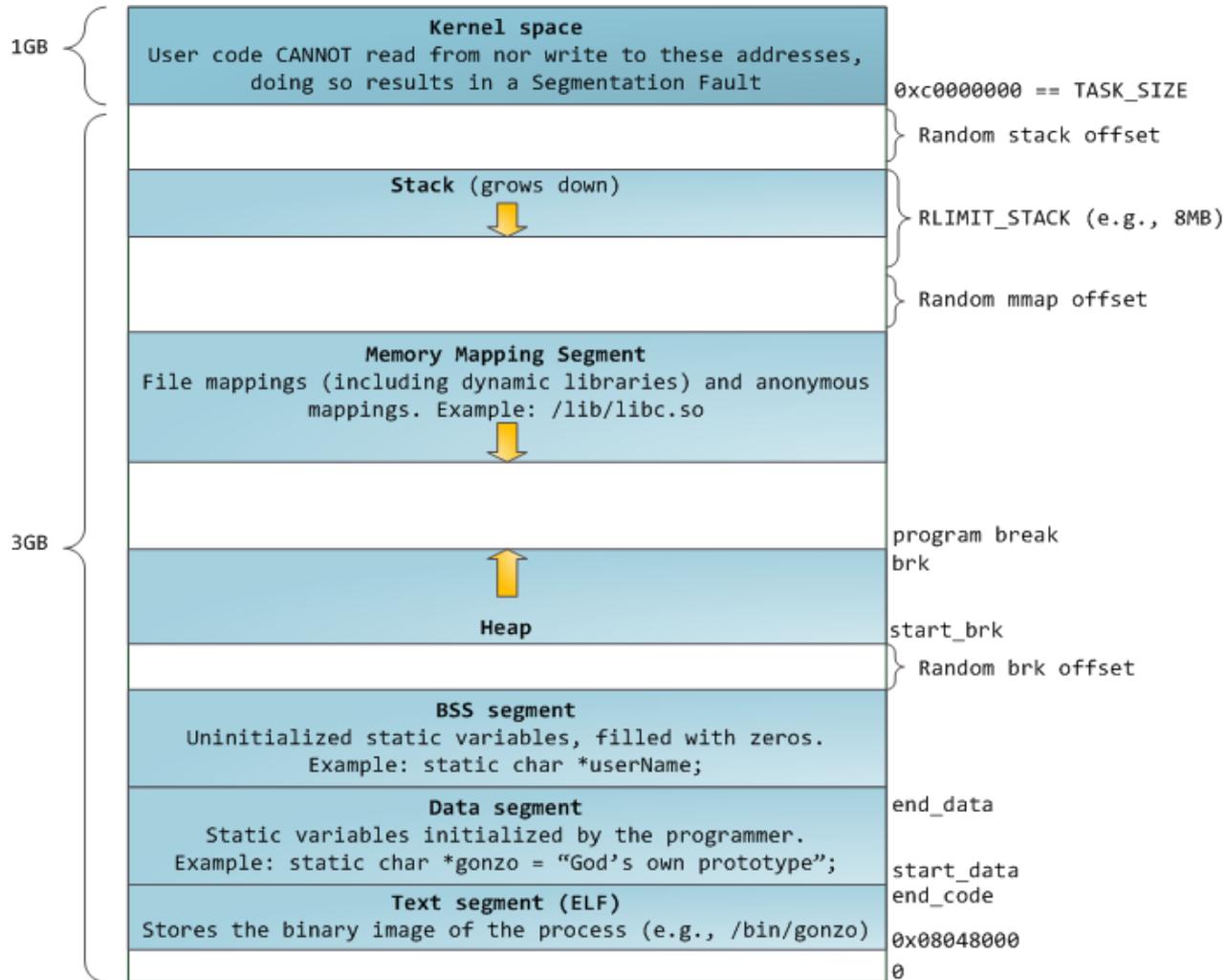
- The “kernel region” of every process has the same page table entries
  - The process cannot access it at user level
  - But on U->K switch, kernel code can access it AS WELL AS the region for THIS user
    - » What does the kernel need to do to access other user processes?
- Different processes running same binary!
  - Execute-only, but do not need to duplicate code segments
- User-level system libraries (execute only)
- Shared-memory segments between different processes
  - Can actually share objects directly between processes
    - » Must map page into same place in address space!
  - This is a limited form of the sharing that threads have within a single process

# Some simple security measures

- Address Space Randomization: Limit the damage of buffer overflow attacks (e.g. overwriting stack to point to arbitrary code)
  - Position-Independent Code => can place user code region anywhere in the address space
    - » Random start address makes much harder for attacker to cause jump to code that it seeks to take over
  - Stack & Heap can start anywhere, so randomize placement
- Kernel address space isolation
  - Don't map whole kernel into each process (Provide separate kernel page table)
  - Meltdown protection => map none of kernel into user mode!

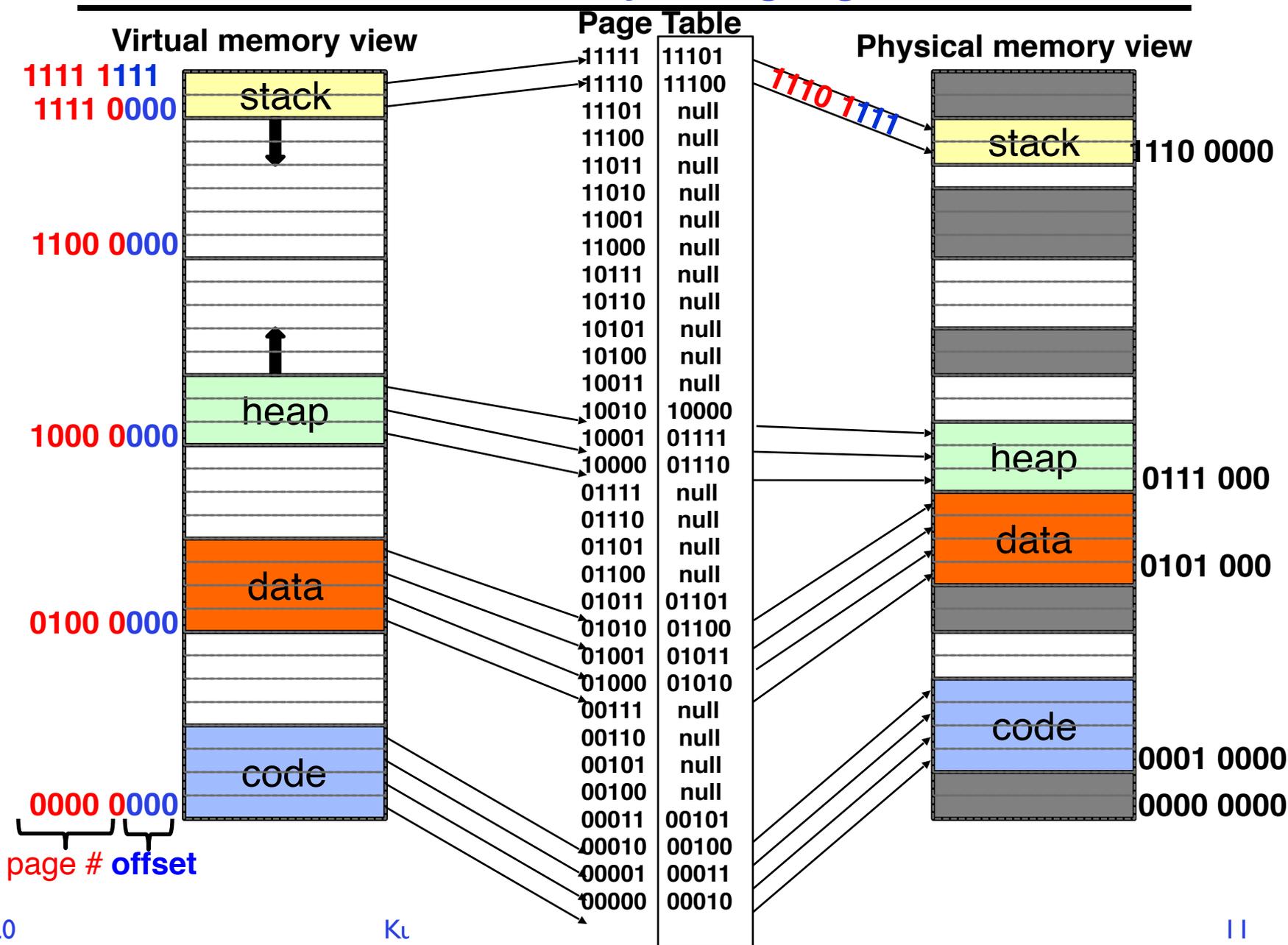


# Example: Memory Layout for Linux 32-bit (Pre-Meltdown patch!)

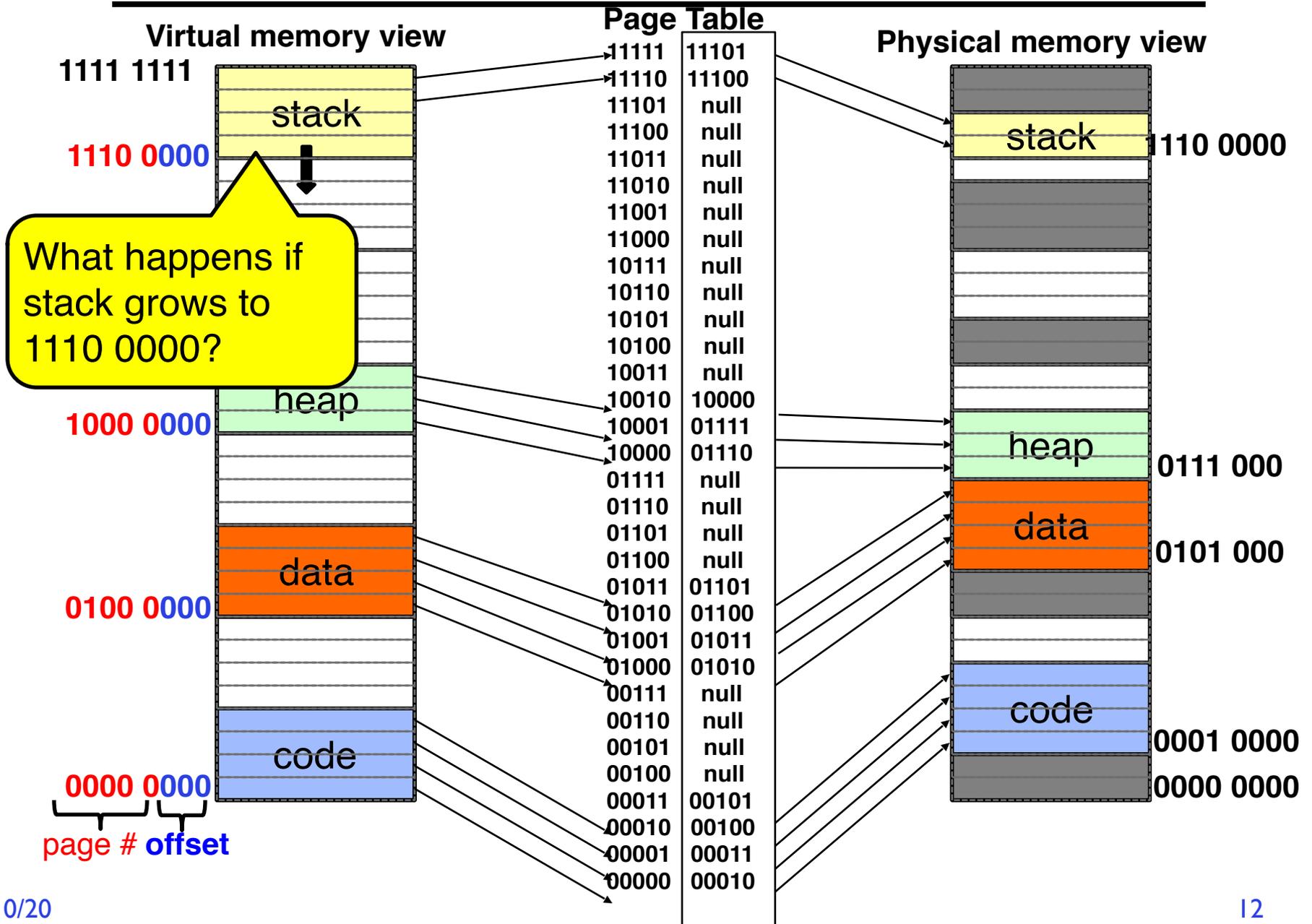


<http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png>

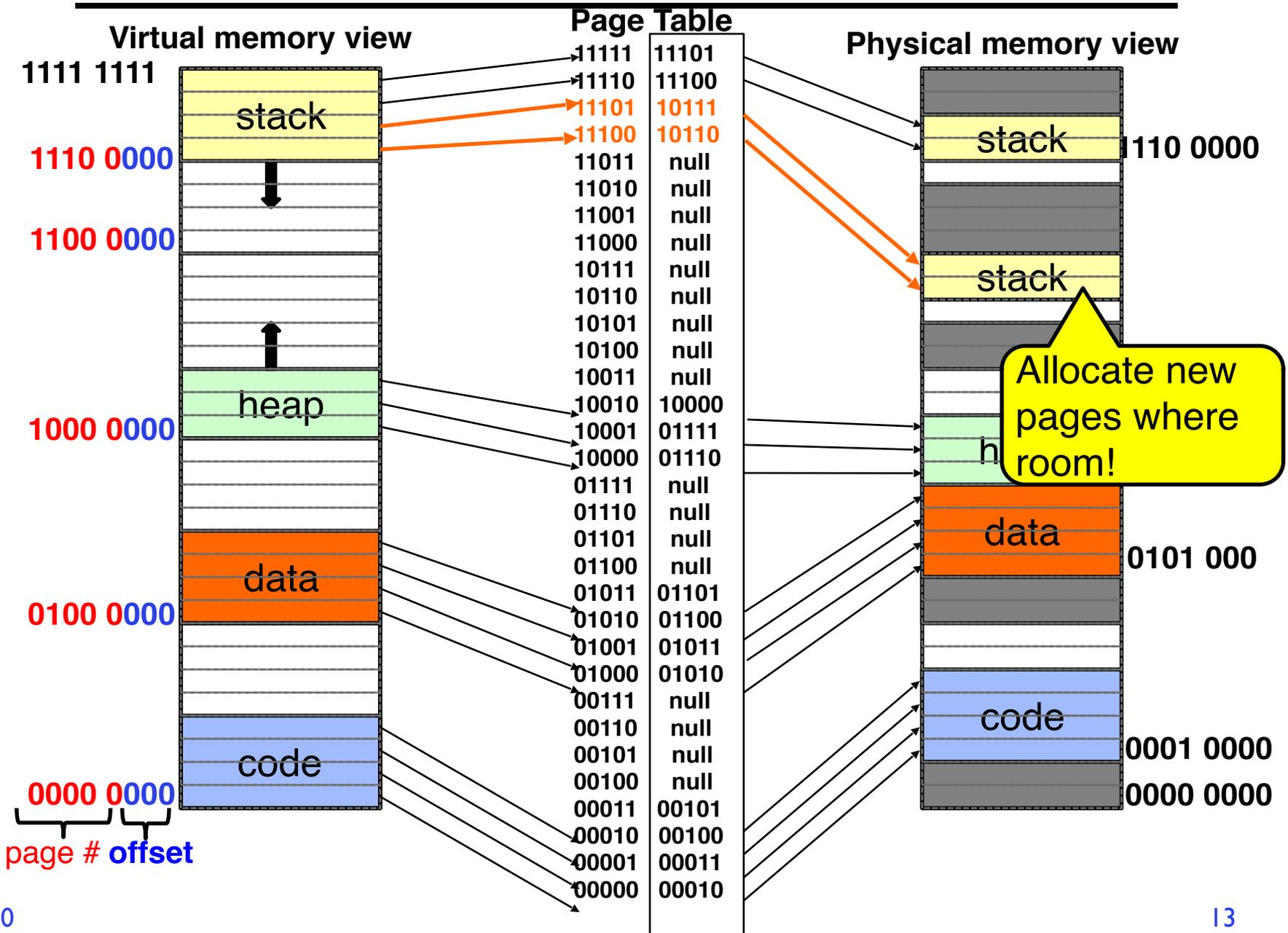
# Summary: Paging



# Summary: Paging



# Summary: Paging



# How big do things get?

---

- 32-bit address space =>  $2^{32}$  bytes (4 GB)
  - Note: “b” = bit, and “B” = byte
  - And *for memory*:
    - » “K”(kilo) =  $2^{10} = 1024 \approx 10^3$  (But not quite!)
    - » “M”(mega) =  $2^{20} = (1024)^2 = 1,048,576 \approx 10^6$  (But not quite!)
    - » “G”(giga) =  $2^{30} = (1024)^3 = 1,073,741,824 \approx 10^9$  (But not quite!)
- Typical page size: 4 KB
  - how many bits of the address is that ? (remember  $2^{10} = 1024$ )
  - Ans –  $4\text{KB} = 4 \times 2^{10} = 2^{12} \Rightarrow 12$  bits of the address
- So how big is the simple page table for each process?
  - $2^{32}/2^{12} = 2^{20}$  (that’s about a million entries)  $\times 4$  bytes each => 4 MB
  - When 32-bit machines got started (vax 11/780, intel 80386), 16 MB was a LOT of memory
- How big is a simple page table on a 64-bit processor (x86\_64)?
  - $2^{64}/2^{12} = 2^{52}$  (that’s  $4.5 \times 10^{15}$  or 4.5 exa-entries)  $\times 8$  bytes each =  
 **$36 \times 10^{15}$  bytes or 36 exa-bytes!!!! This is a ridiculous amount of memory!**
  - This is really a lot of space – for only the page table!!!
- Mostly, the address space is *sparse*, i.e. has holes in it that are not mapped to physical memory
  - So, most of this space is taken up by page tables mapped to nothing

# Page Table Discussion

---

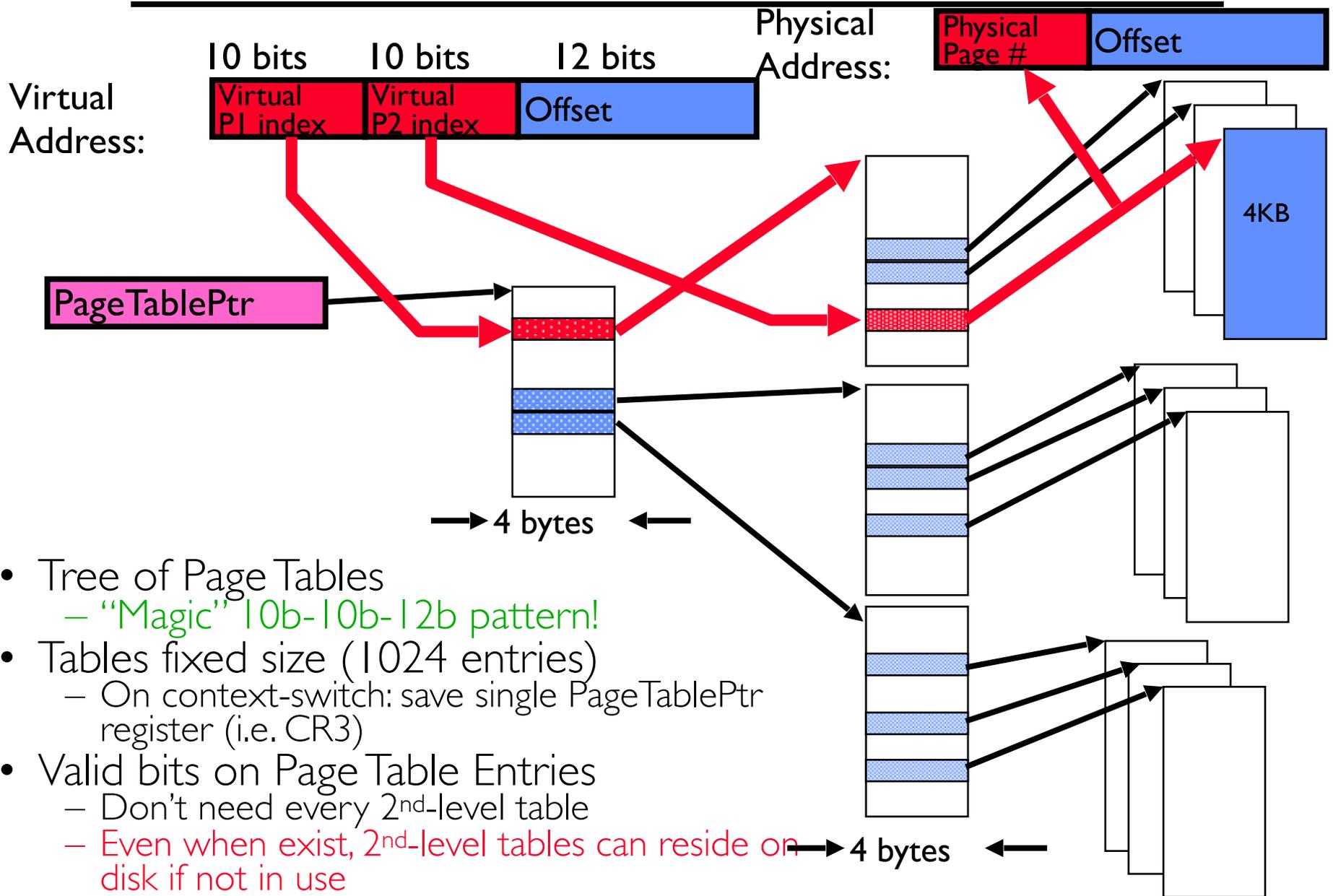
- What needs to be switched on a context switch?
  - Page table pointer and limit
- What provides protection here?
  - Translation (per process) *and* dual-mode!
  - Can't let process alter its own page table!
- Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to share
  - Con: What if address space is sparse?
    - » E.g., on UNIX, code starts at 0, stack starts at  $(2^{31}-1)$
    - » With 1K pages, need 4 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

# Administrivia

---

- Hope everyone is doing well
- HW2 will be released today.
- What are your ideas on midterm evaluation?

# 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 2<sup>nd</sup>-level table
  - Even when exist, 2<sup>nd</sup>-level tables can reside on disk if not in use

# Example: x86 classic 32-bit address translation

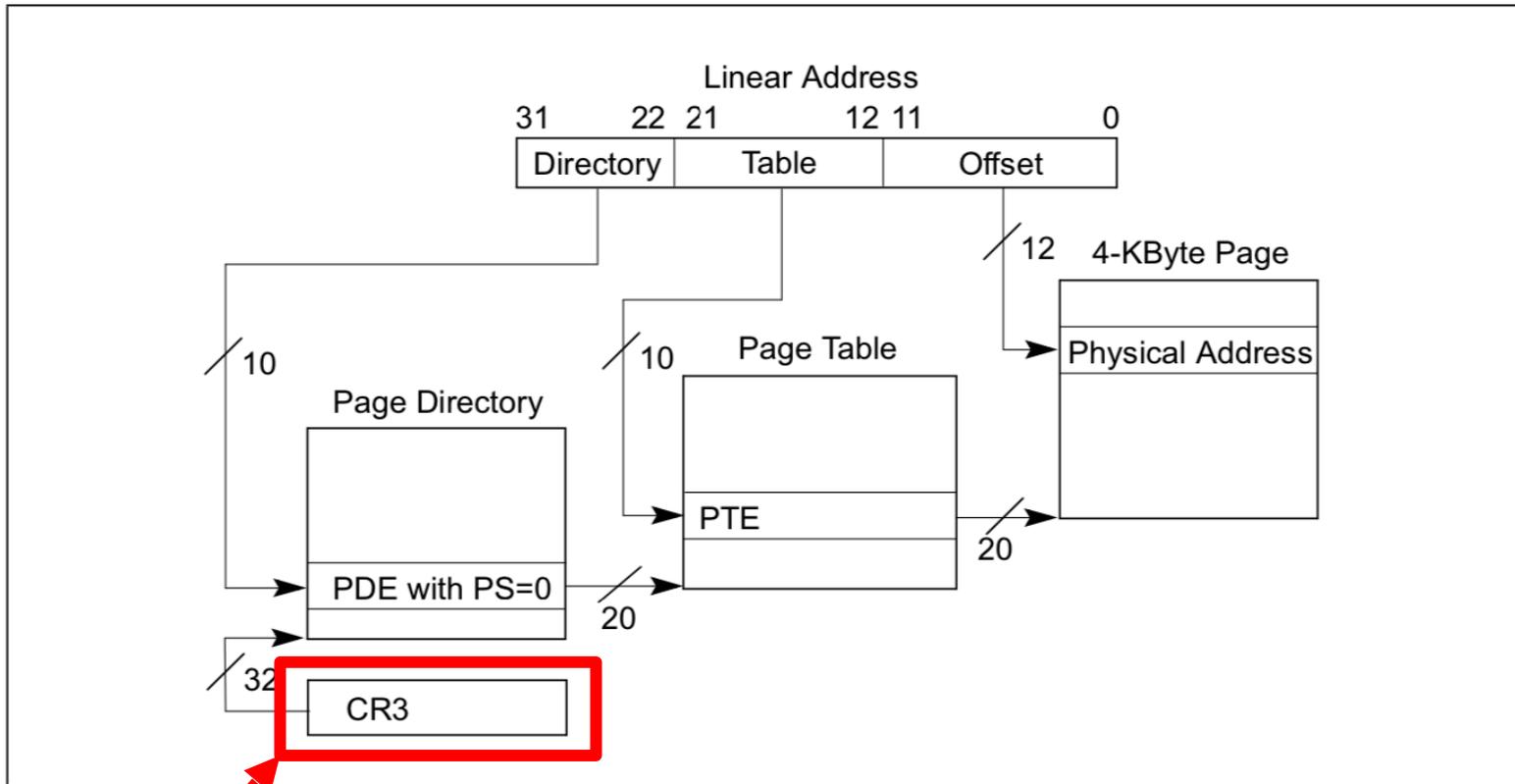


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

- Intel terminology: Top-level page-table called a “Page Directory”
  - With “Page Directory Entries”
- CR3 provides physical address of the page directory
  - This is what we have called the “PageTablePtr” in previous slides
  - Change in CR3 changes the whole translation table!

# What is in a Page Table Entry (PTE)?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called “Directories”



P: Present (same as “valid” bit in other architectures)

W: Writeable

U: User accessible

PWT: Page write transparent: external cache write-through

PCD: Page cache disabled (page cannot be cached)

A: Accessed: page has been accessed recently

D: Dirty (PTE only): page has been modified recently

PS: Page Size: PS=1  $\Rightarrow$  4MB page (directory only).

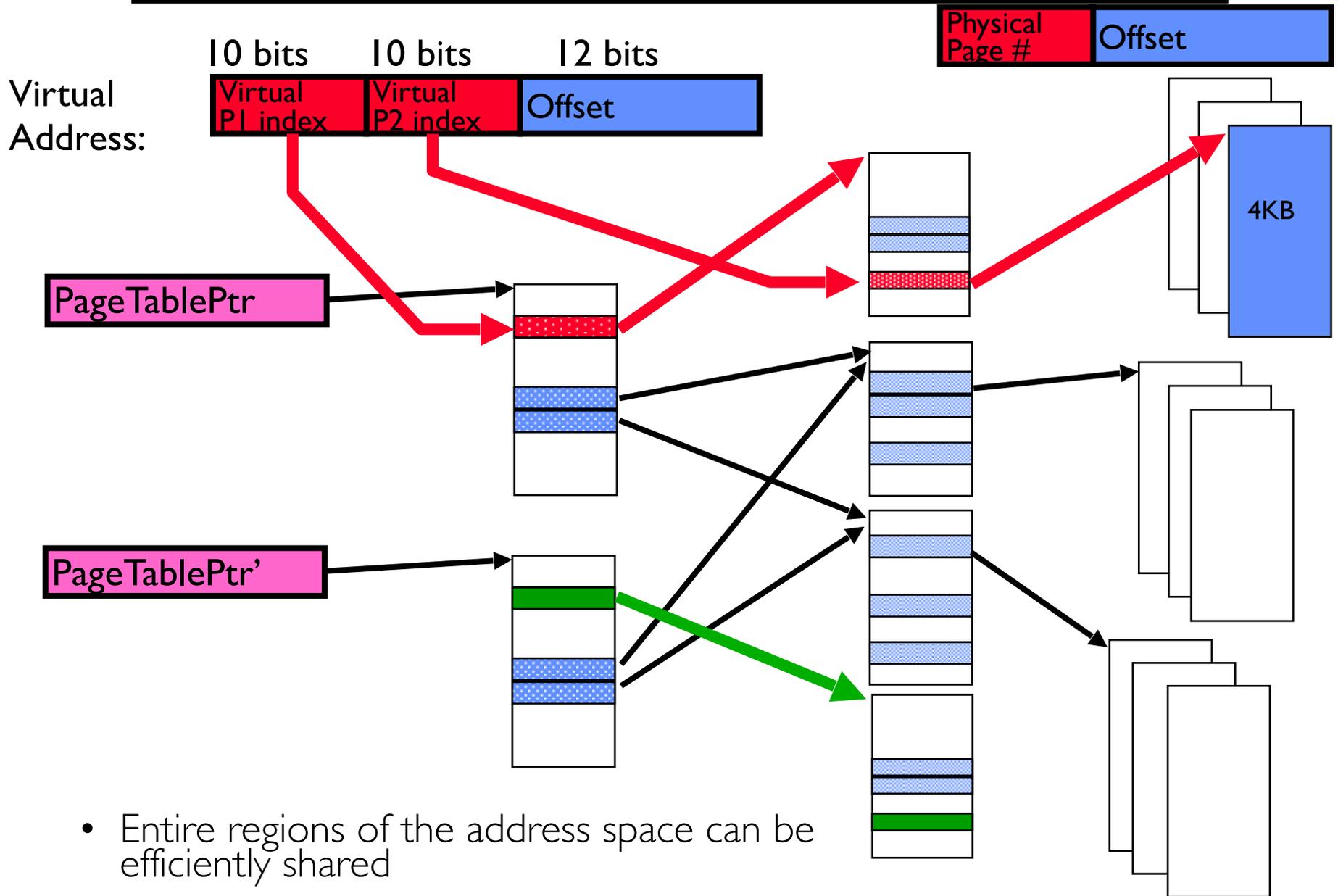
Bottom 22 bits of virtual address serve as offset

# Examples of how to use a PTE

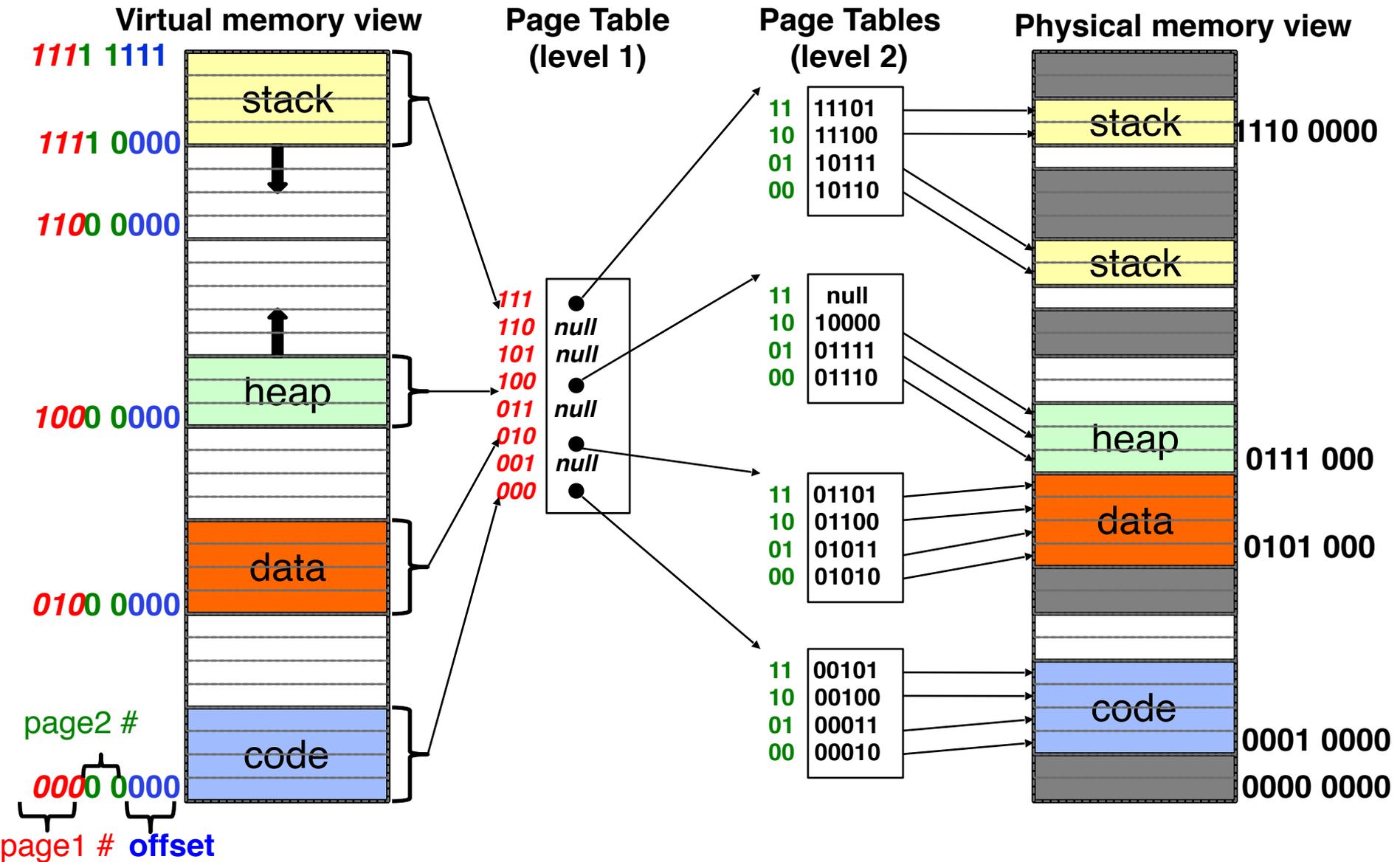
---

- 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 (say) 31 bits for location info
- 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

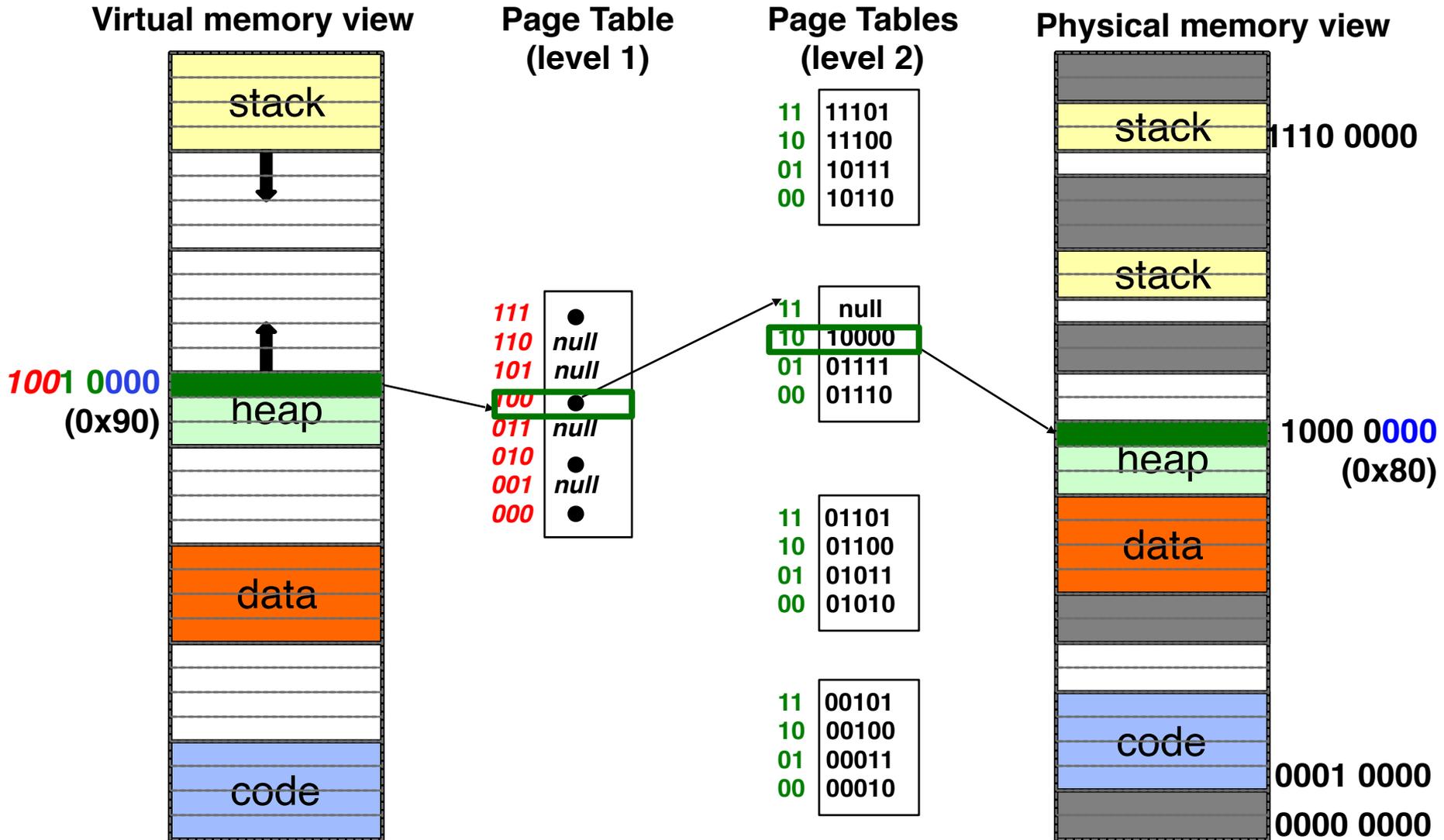
# Sharing with multilevel page tables



# Summary: Two-Level Paging

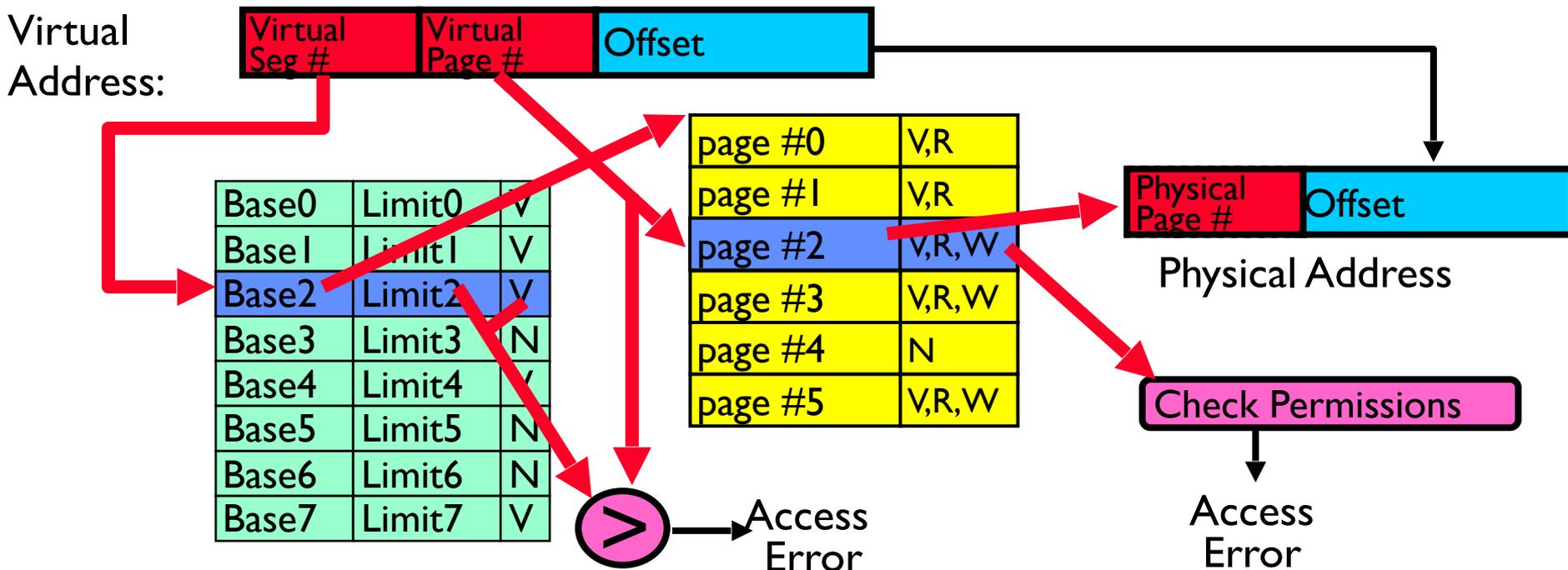


# Summary: Two-Level Paging



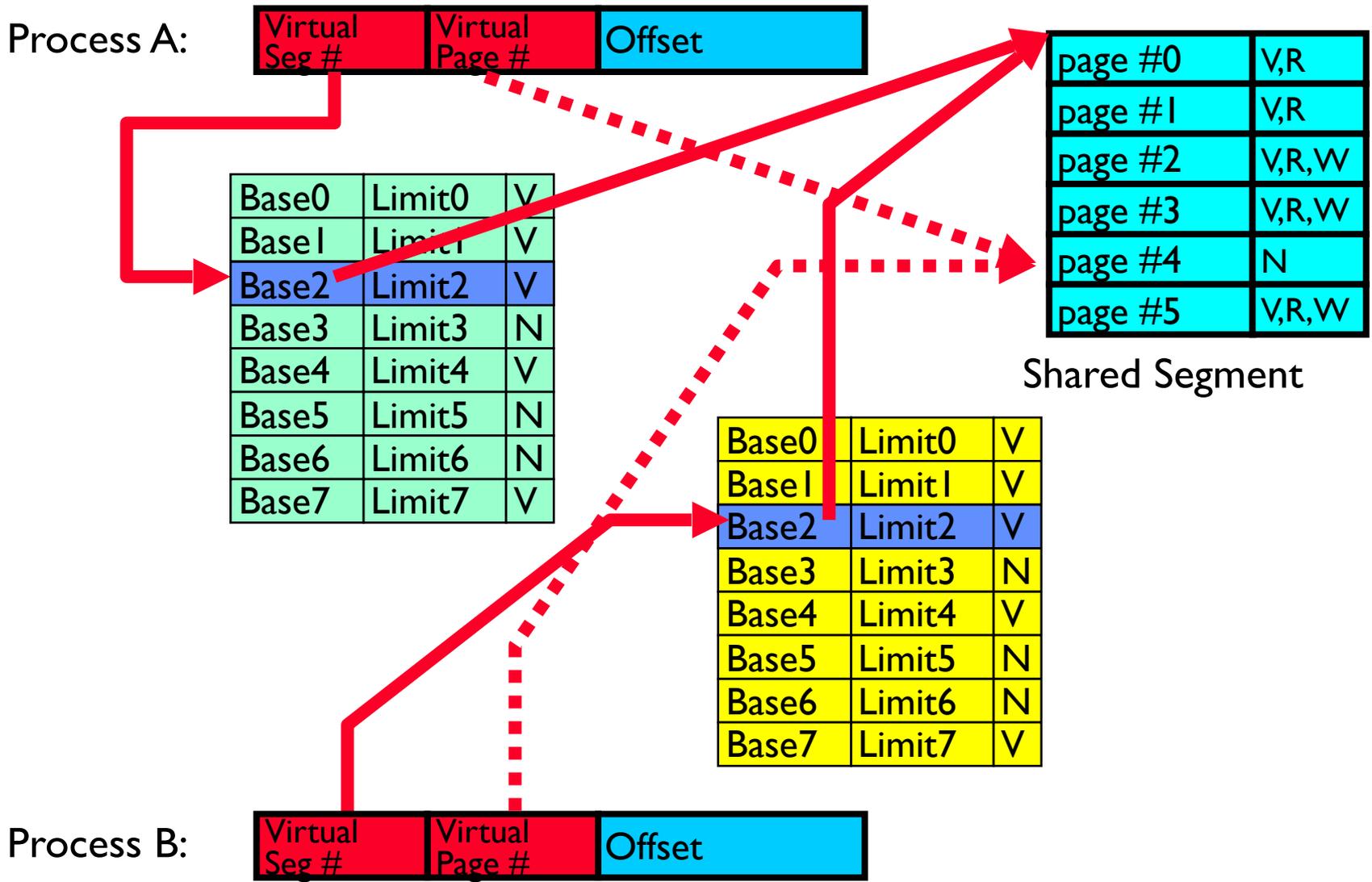
# Multi-level Translation: Segments + Pages

- What about a tree of tables?
  - Lowest level page table  $\Rightarrow$  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)?



# Multi-level Translation Analysis

---

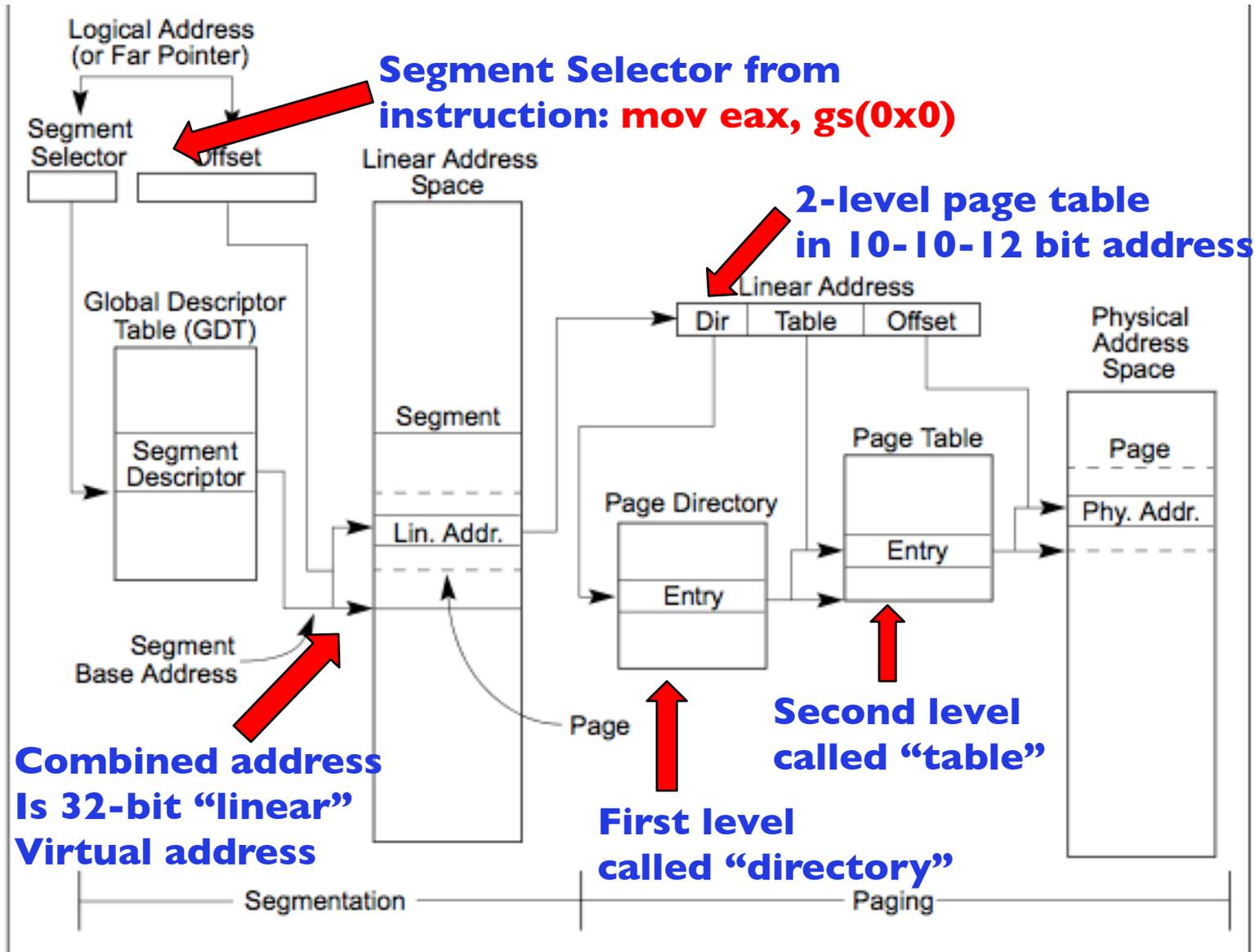
- Pros:
  - Only need to allocate as many page table entries as we need for application
    - » In other words, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - » Share at segment or page level
- Cons:
  - One pointer per page (typically 4K – 16K pages today)
  - Page 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

---

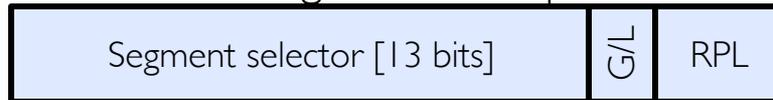
- Can a process modify its own translation tables? **NO!**
  - If it could, could get access to all of physical memory (no protection!)
- To Assist with Protection, **Hardware** provides at least two modes (Dual-Mode Operation):
  - “Kernel” mode (or “supervisor” or “protected”)
  - “User” mode (Normal 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 (CR3 in x86), 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

Making it real:  
X86 Memory model with segmentation (16/32-bit)



# X86 Segment Descriptors (32-bit Protected Mode)

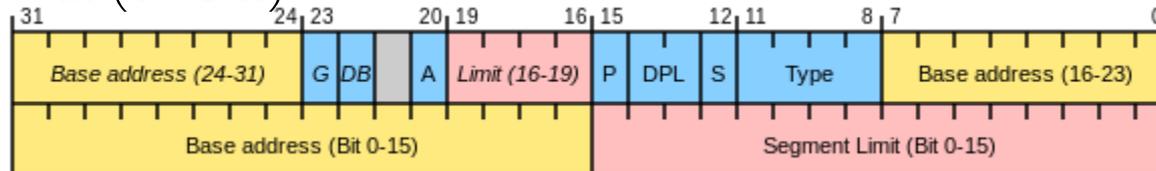
- Segments are either implicit in the instruction (say for code segments) or actually part of the instruction
  - There are 6 registers: SS, CS, DS, ES, FS, GS
- What is in a segment register?
  - A *pointer* to the actual segment description:



G/L selects between GDT and LDT tables (global vs local descriptor tables)

- RPL: Requestor's Privilege Level (**RPL of CS  $\Rightarrow$  Current Privilege Level**)

- Two registers: GDTR and LDTR hold pointers to the global and local descriptor tables in memory
  - Includes length of table (for  $< 2^{13}$ ) entries
- Descriptor format (64 bits):



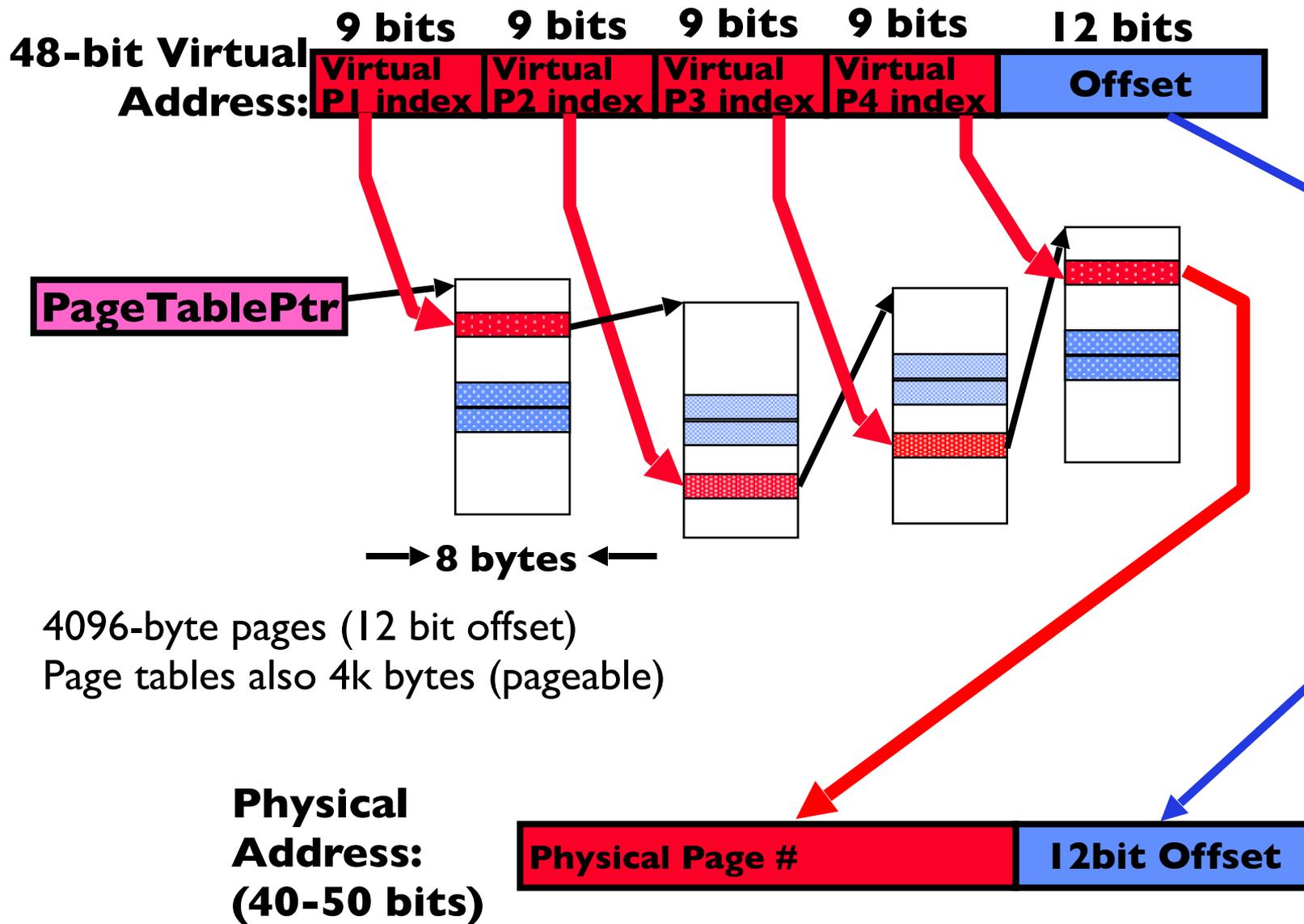
- G: Granularity of segment [ Limit Size ] (0: 16bit, 1: 4KiB unit)
- DB: Default operand size (0: 16bit, 1: 32bit)
- A: Freely available for use by software
- P: Segment present
- DPL: Descriptor Privilege Level: Access requires  $\text{Max}(\text{CPL}, \text{RPL}) \leq \text{DPL}$
- S: System Segment (0: System, 1: code or data)
- Type: Code, Data, Segment

# How are segments used?

---

- One set of global segments (GDT) for everyone, different set of local segments (LDT) for every process
- In legacy applications (16-bit mode):
  - Segments provide protection for different components of user programs
  - Separate segments for chunks of code, data, stacks
    - » RPL of Code Segment  $\Rightarrow$  CPL (Current Privilege Level)
  - Limited to 64K segments
- Modern use in 32-bit Mode:
  - Even though there is full segment functionality, segments are set up as “flattened”, i.e. every segment is 4GB in size
  - One exception: Use of GS (or FS) as a pointer to “Thread Local Storage” (TLS)
    - » A thread can make accesses to TLS like this:  
`mov eax, gs(0x0)`
- Modern use in 64-bit (“long”) mode
  - Most segments (SS, CS, DS, ES) have zero base and no length limits
  - Only FS and GS retain their functionality (for use in TLS)

# X86\_64: Four-level page table!



# From x86\_64 architecture specification

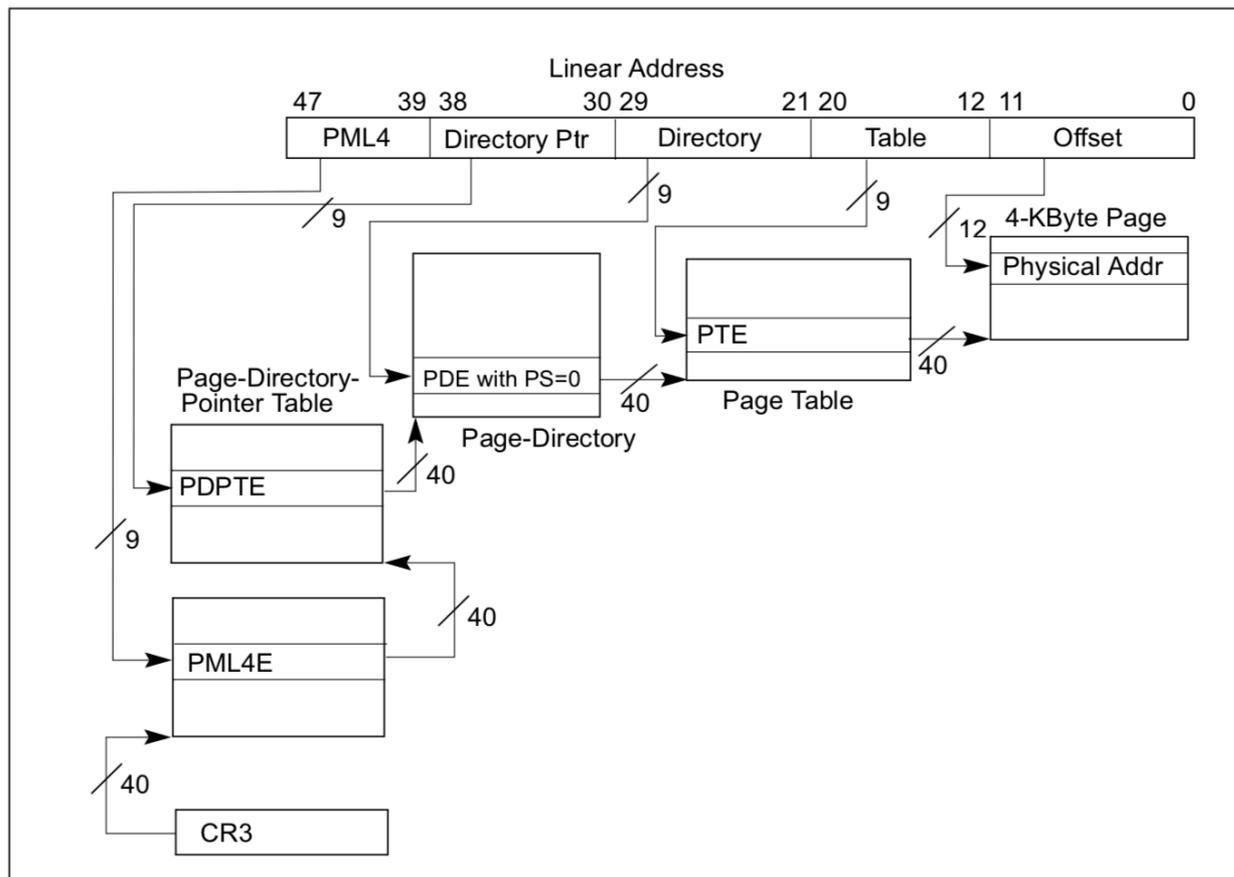


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

- All current x86 processor support a 64 bit operation
- 64-bit words (so ints are 8 bytes) but 48-bit addresses

# Larger page sizes supported as well

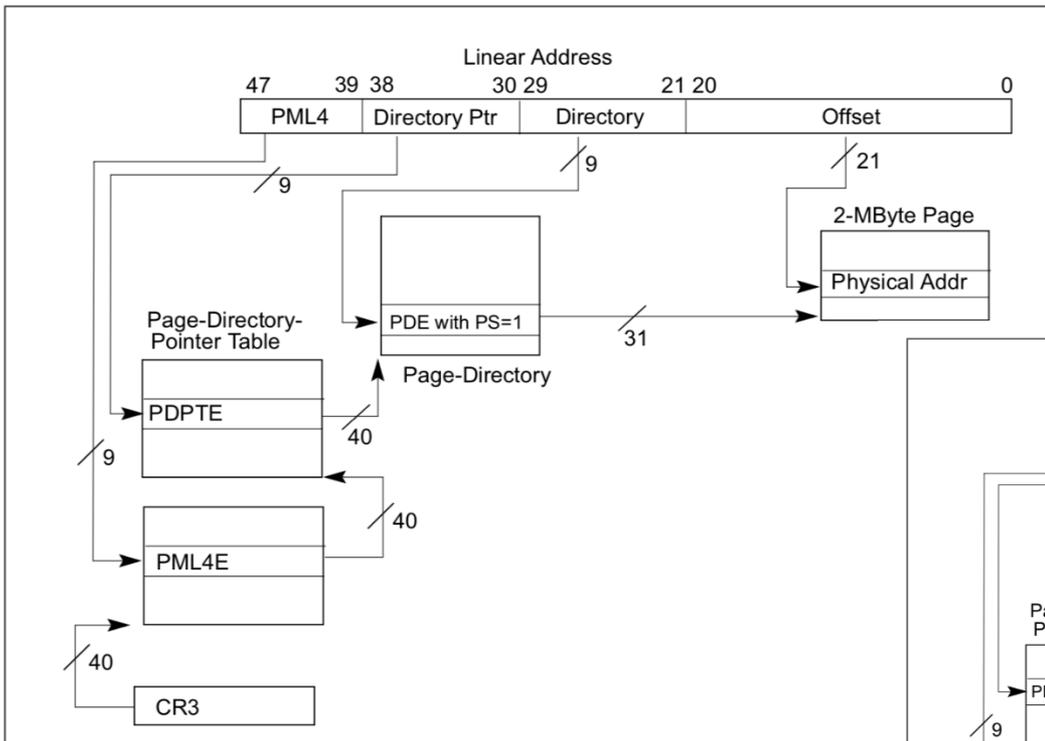


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-L

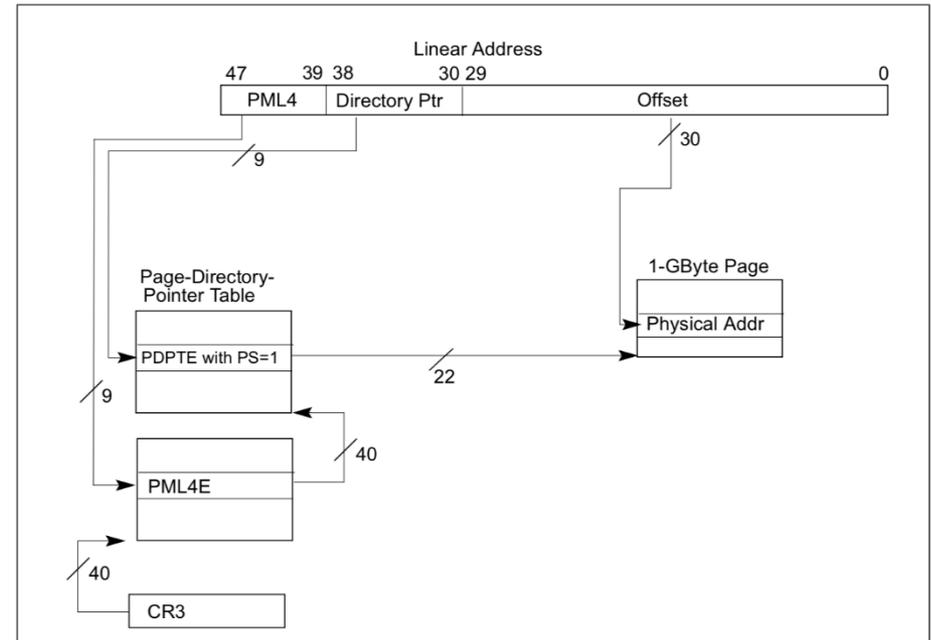


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

- Or larger page sizes, memory is now cheap

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

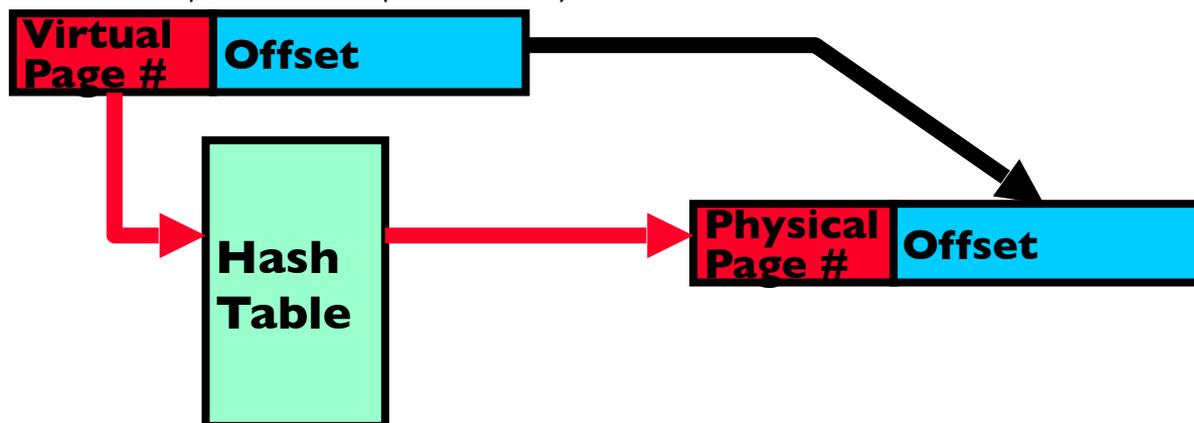


No!

Too slow  
Too many almost-empty tables

# Inverted Page Table

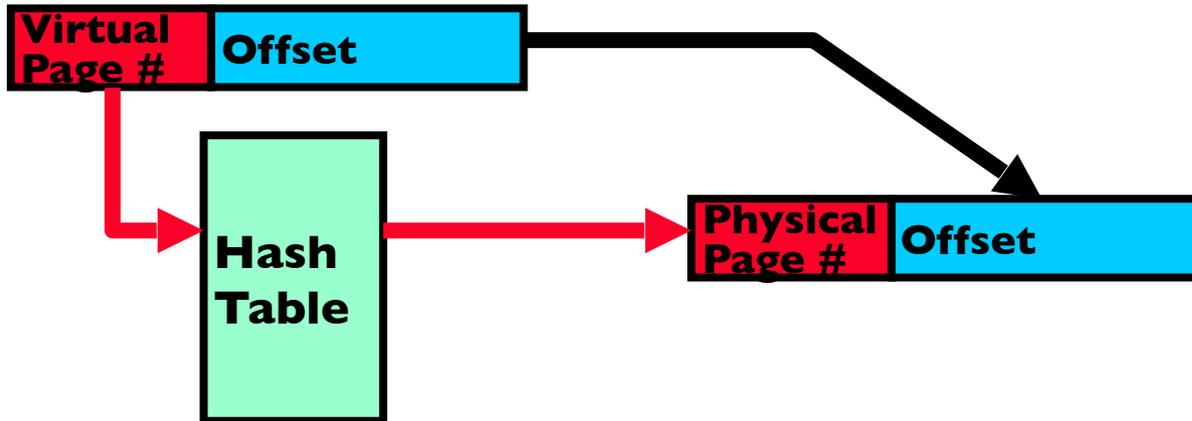
- With all previous examples (“Forward Page Tables”)
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - » Much of process space may be out on disk or not in use



- 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

# 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



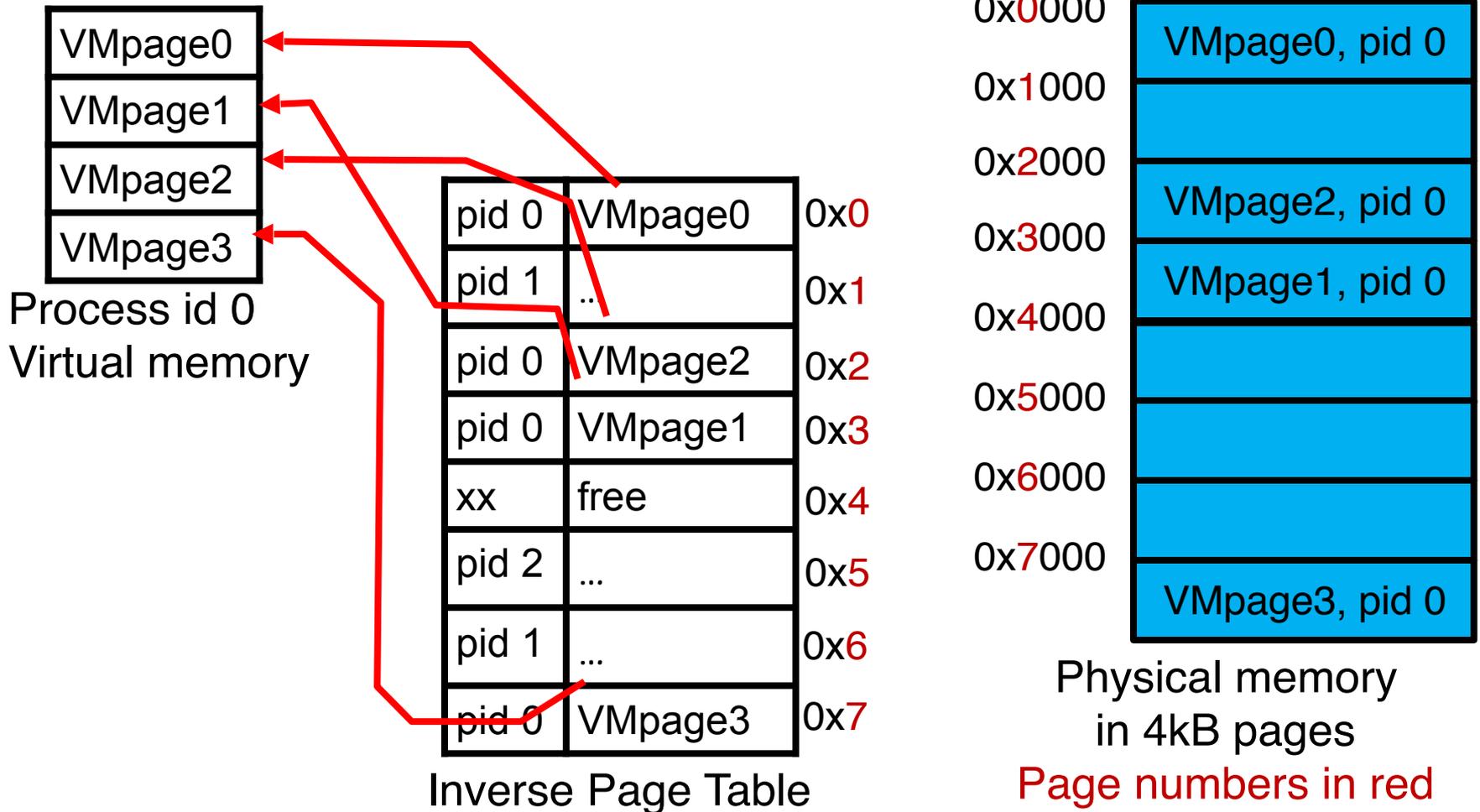
- Answer: use a hash table
  - Called an “Inverted Page Table”
  - Size
  - Dir
  - Ver
  - »

Total size of page table  $\approx$  number of pages **used** by program in **physical memory**. Hash more complex

- Cons:
  - Complexity of managing hash chains: Often in hardware!
  - Poor cache locality of page table

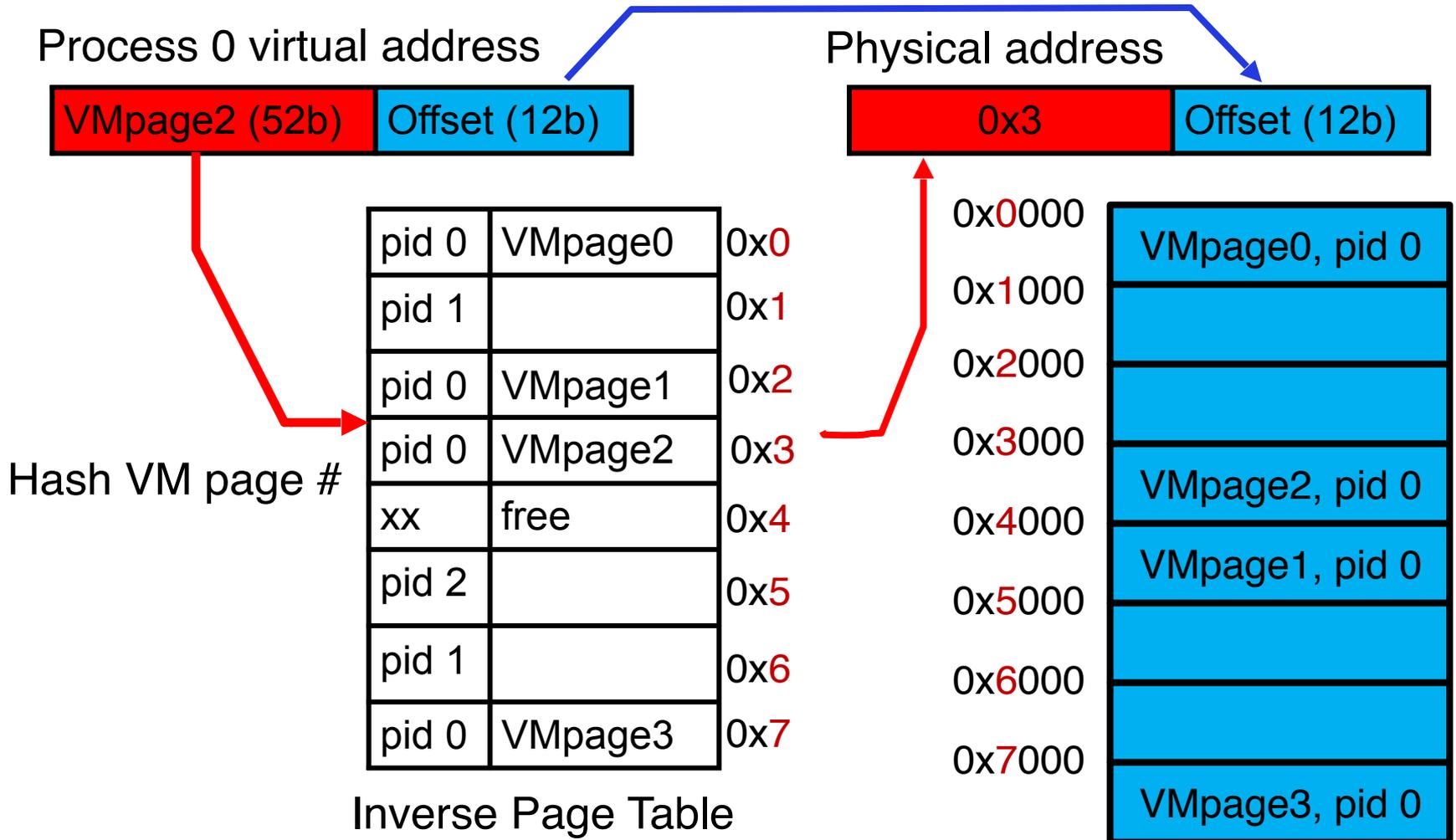
# IA64: Inverse Page Table (IPT)

Idea: index page table by physical pages instead of VM



# IPT address translation

- Need an associative map from VM page to IPT address:
  - Use a hash map



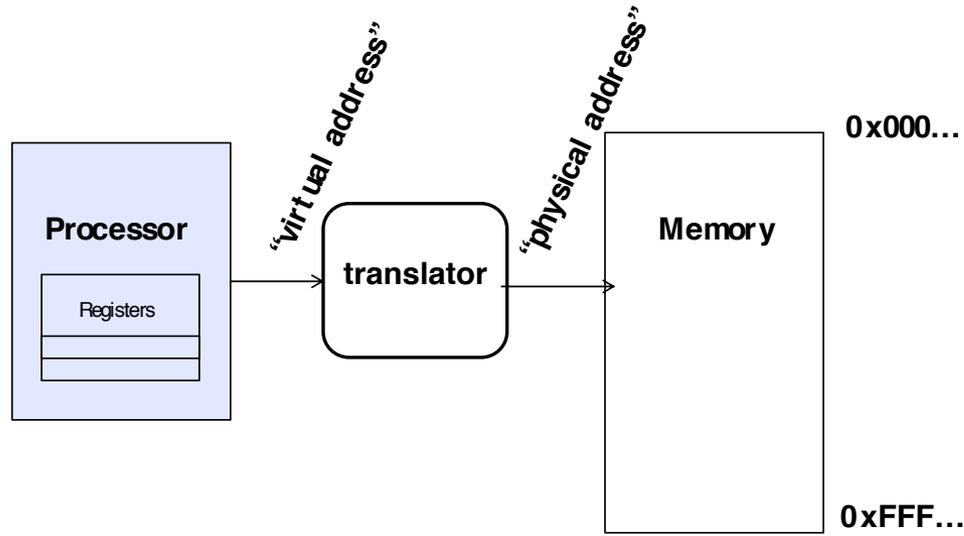
# Address Translation Comparison

---

	Advantages	Disadvantages
Simple Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Paging (single-level page)	No external fragmentation, fast easy allocation	Large table size ~ virtual memory Internal fragmentation
Paged segmentation	Table size ~ # of pages in <b>virtual memory</b> , fast easy allocation	Multiple memory references per page access
Two-level pages		
Inverted Table	Table size ~ # of pages in <b>physical memory</b>	Hash function more complex No cache locality of page table

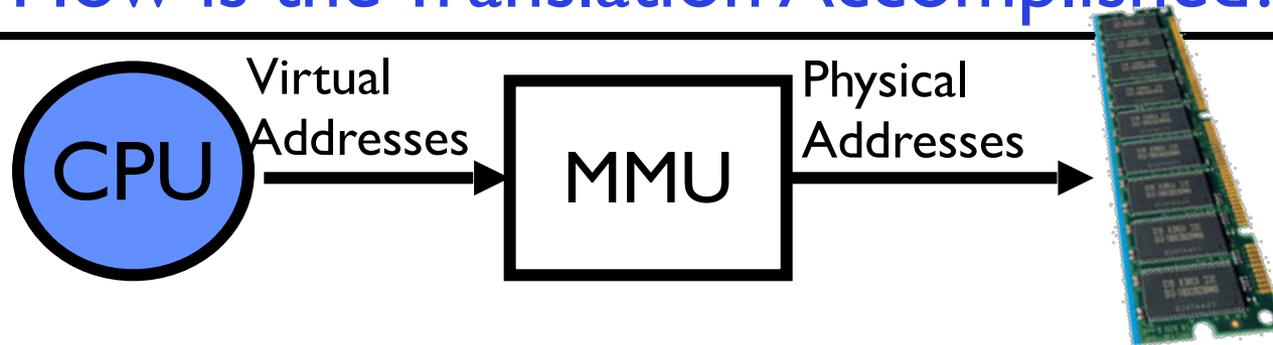
# Two Critical Issues in Address Translation

---



- How to translate addresses fast enough?
  - Every instruction fetch
  - Plus every load / store
  - EVERY MEMORY REFERENCE !
  - More than one translation for EVERY instruction
- What to do if the translation fails?
  - Page fault (Later!)

# How is the Translation Accomplished?



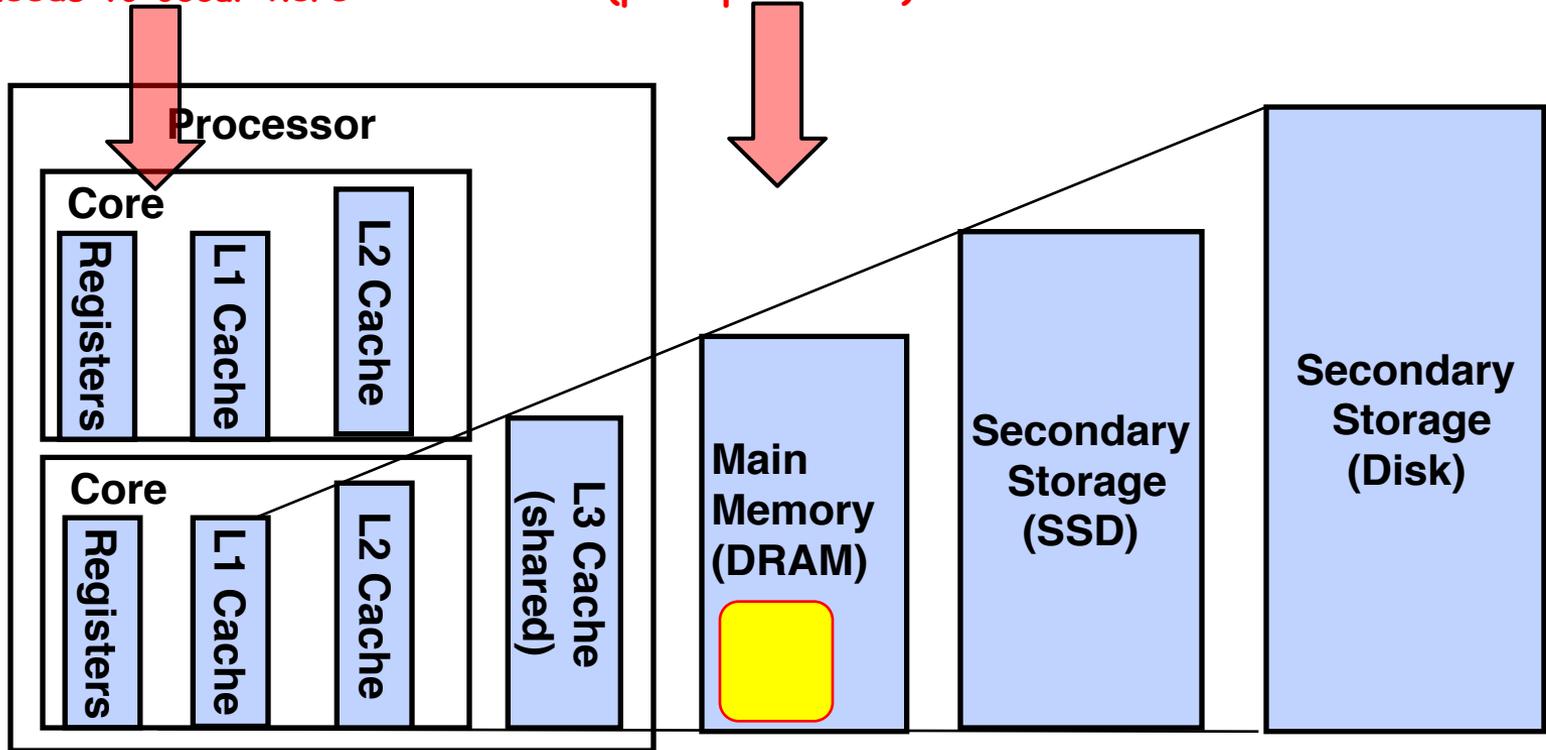
- What does the MMU need to do to translate an address?
- 1-level Page Table
  - Read PTE from memory, check valid, merge address
  - Set “accessed” bit in PTE, Set “dirty bit” on write
- 2-level Page Table
  - Read and check first level
  - Read, check, and update PTE
- N-level Page Table ...
- MMU does *page table Tree Traversal* to translate each address
- How can we make this go REALLY fast?
  - Fraction of a processor cycle

# Recall: Memory Hierarchy

- Large memories are slow, only small memory is fast

Address Translation needs to occur here

Page table lives here (perhaps cached)



Speed (ns): 0.3

1

3

10-30

100

100,000  
(0.1 ms)

10,000,000  
(10 ms)

Size (bytes): 100Bs

10kB

100kB

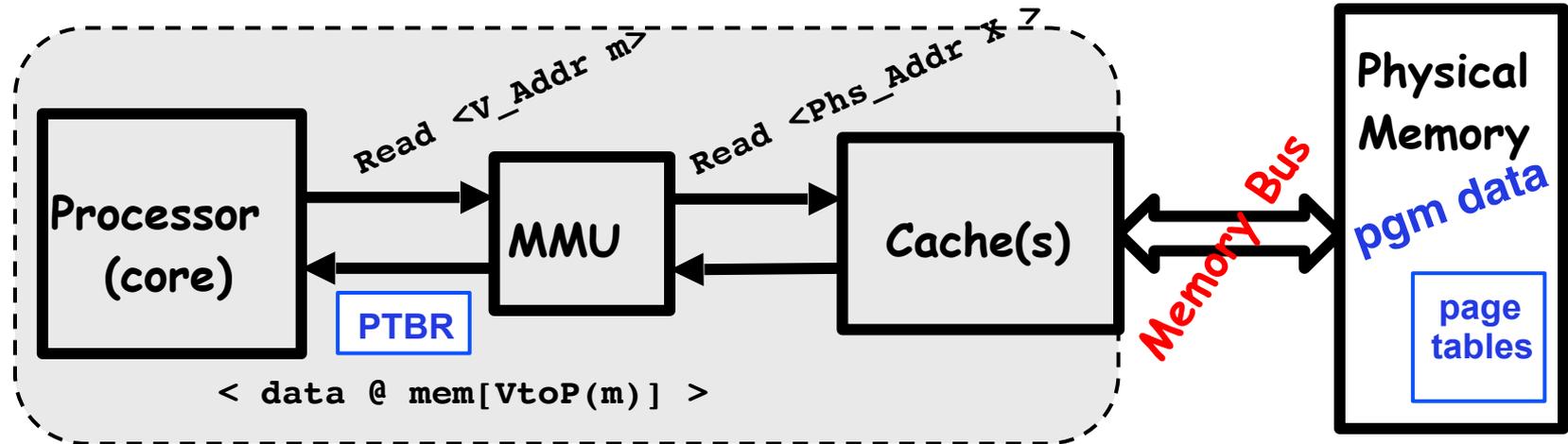
MB

GB

100GB

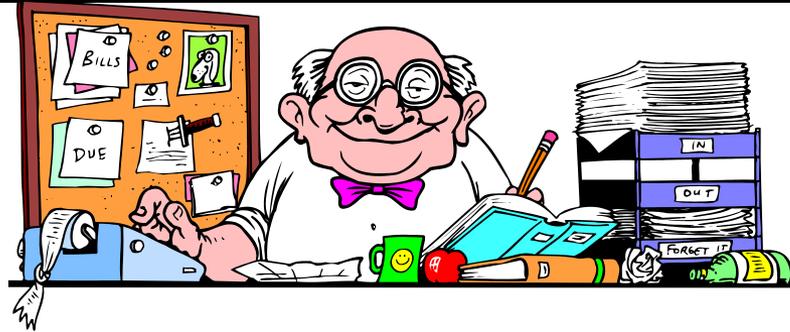
TB

# 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  $\rightarrow$  physical) translation
  - Fast on a cache hit, slow on a miss
- So what is the MMU doing?
- On every reference (l-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

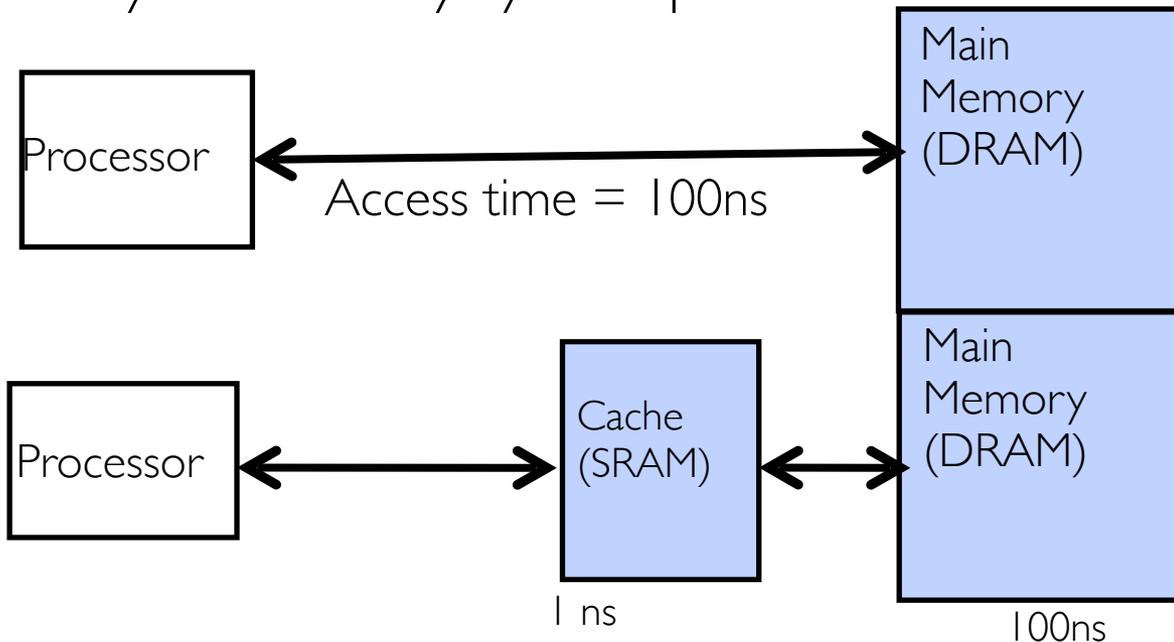
# Recall: CS61c Caching Concept



- **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 and
  - Infrequent case not too expensive
- Important measure: Average Access time =  
(Hit Rate × **Hit Time**) + (Miss Rate × **Miss Time**)

# Recall: In Machine Structures (eg. 6 | C) ...

- Caching is the key to memory system performance



Average Memory Access Time (AMAT)

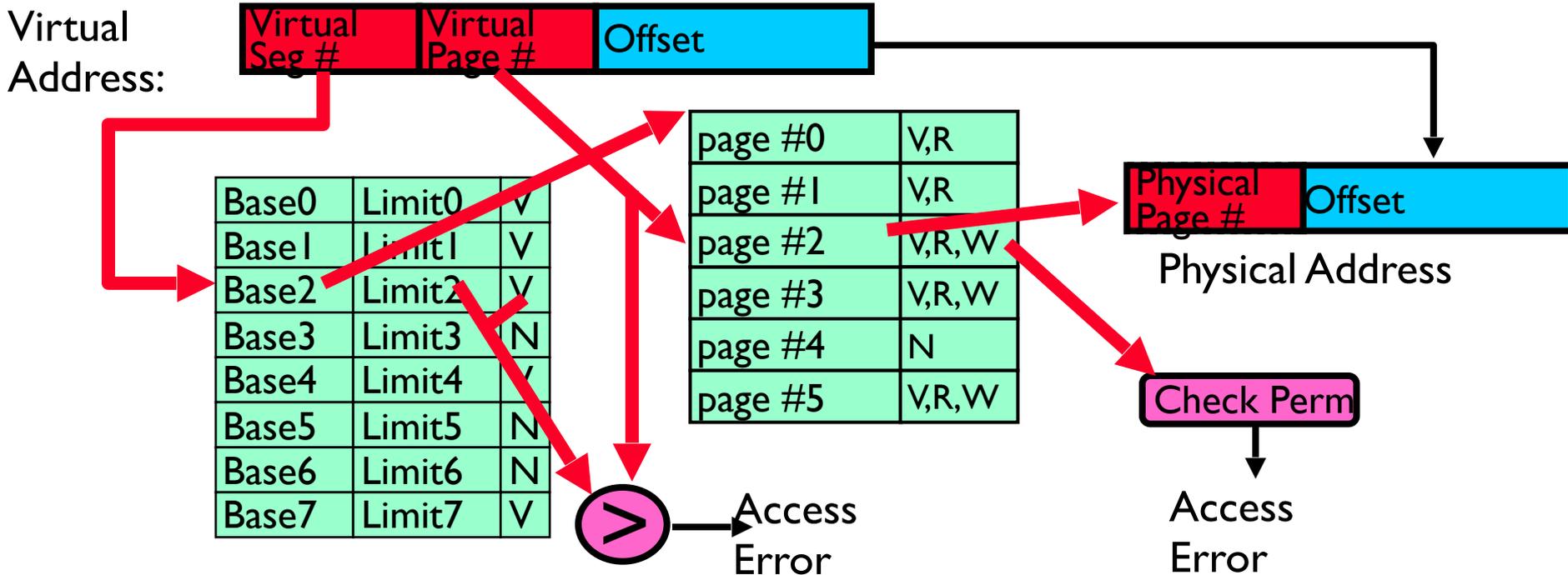
$$= (\text{Hit Rate} \times \text{HitTime}) + (\text{Miss Rate} \times \text{MissTime})$$

Where  $\text{HitRate} + \text{MissRate} = 1$

$$\text{HitRate} = 90\% \Rightarrow \text{AMAT} = (0.9 \times 1) + (0.1 \times 101) = 11.1 \text{ ns}$$

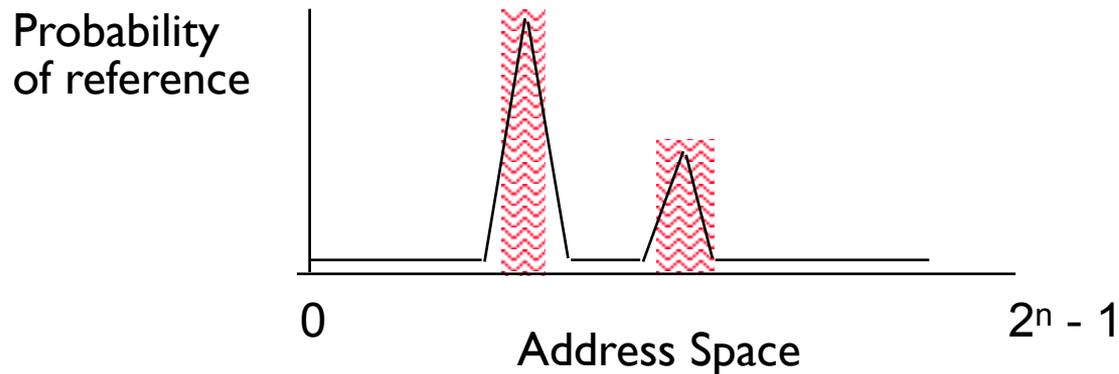
$$\text{HitRate} = 99\% \Rightarrow \text{AMAT} = (0.99 \times 1) + (0.01 \times 101) = 2.01 \text{ ns}$$

# Another Major Reason to Deal with Caching

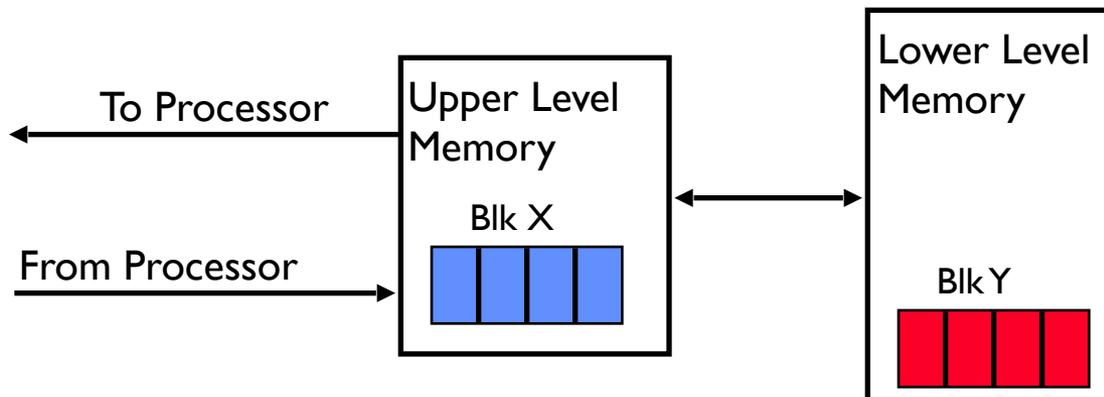


- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access?
- Solution? Cache translations!
  - Translation Cache: TLB (“Translation Lookaside Buffer”)

# Why Does Caching Help? Locality!

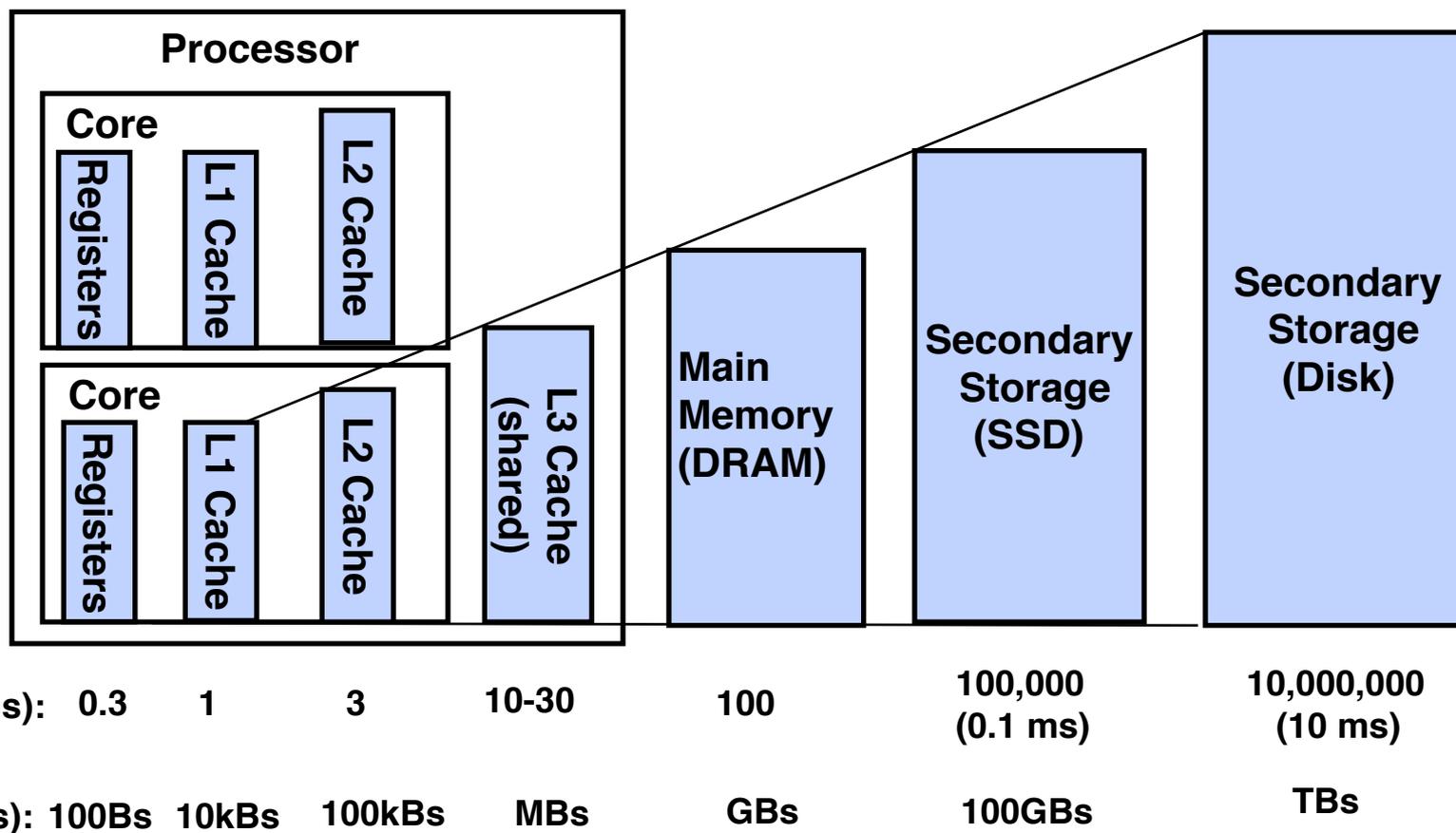


- **Temporal Locality** (Locality in Time):
  - Keep recently accessed data items closer to processor
- **Spatial Locality** (Locality in Space):
  - Move contiguous blocks to the upper levels



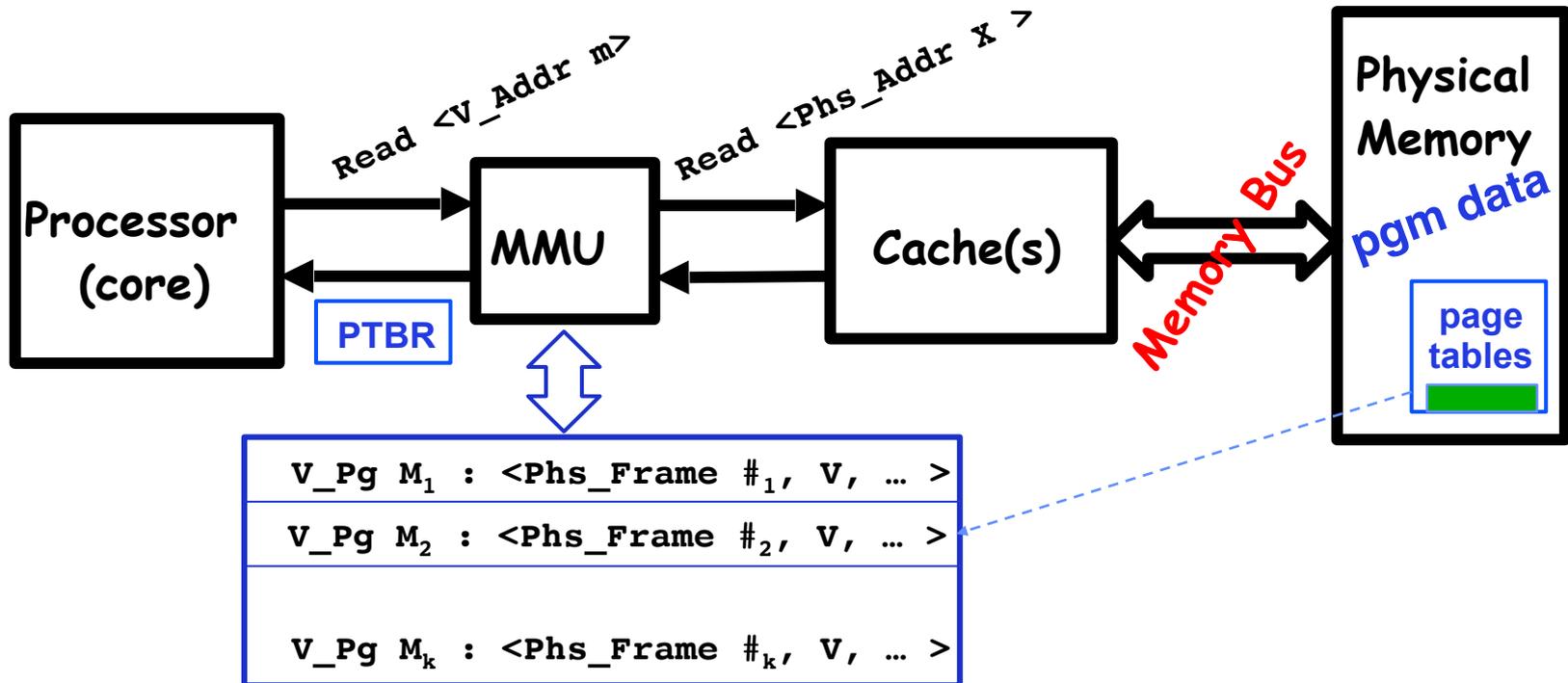
# Recall: Memory Hierarchy

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed 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

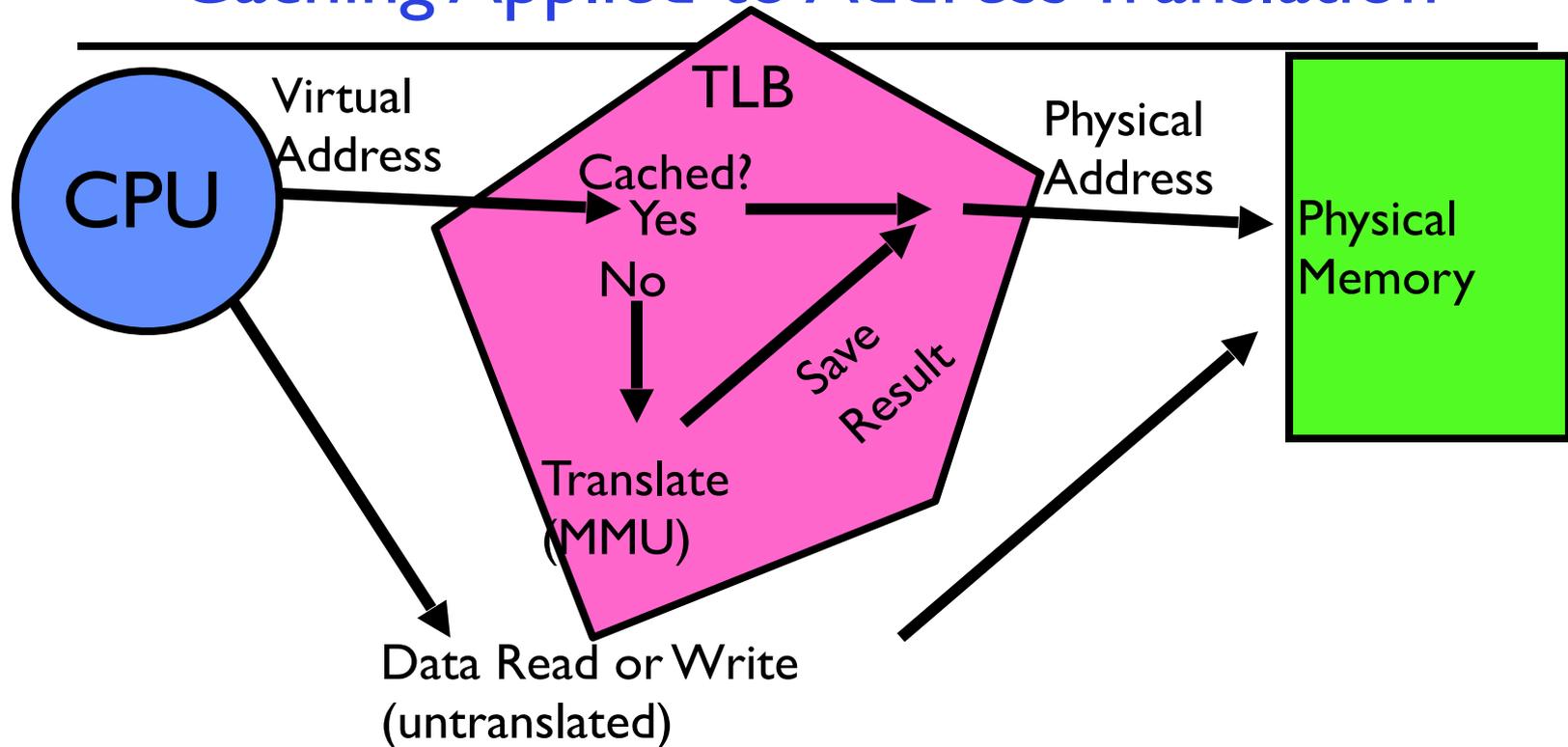


# Translation Look-Aside Buffer

---

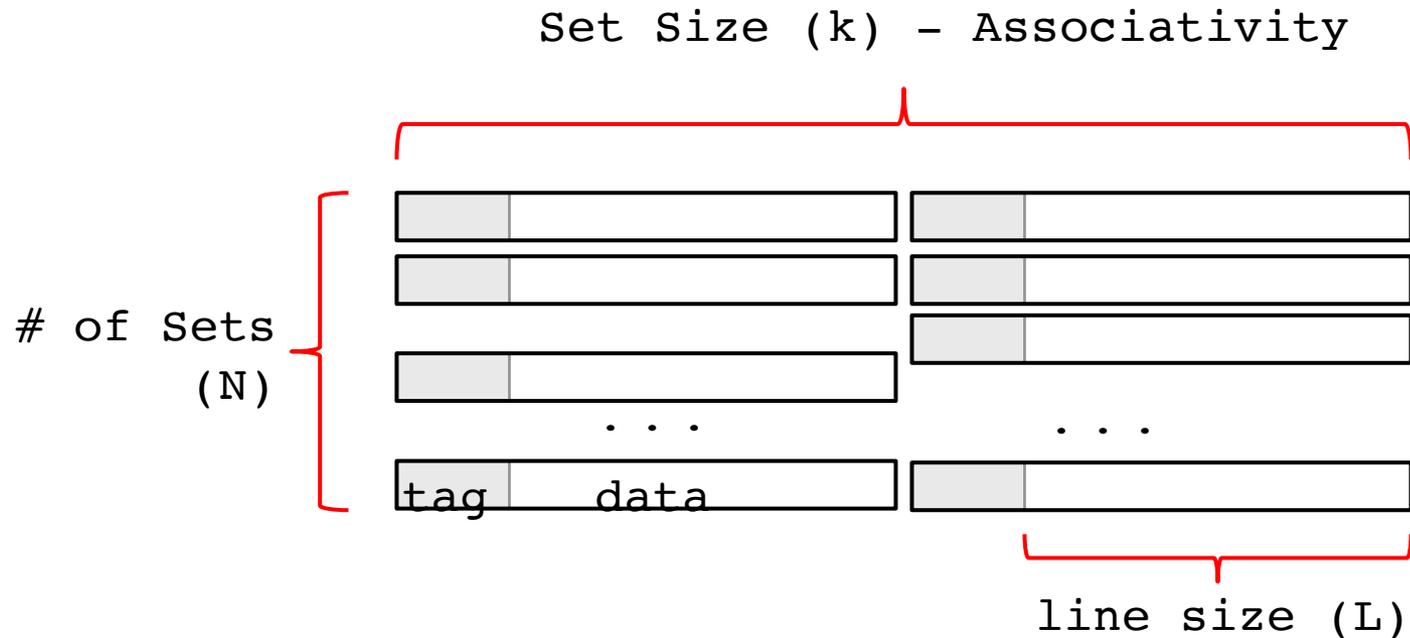
- Record recent Virtual Page # to Physical Frame # translation
- If present, have the physical address without reading any of the page tables !!!
  - Even if the translation involved multiple levels
  - Caches the end-to-end result
- Was invented by Sir Maurice Wilkes – *prior to caches*
  - People realized “if it’s good for page tables, why not the rest of the data in memory?”
- 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

# What kind of Cache for TLB?



- Remember all those cache design parameters and trade-offs?
  - Amount of Data =  $N * L * K$
  - Tag is portion of address that identifies line (w/o line offset)
  - Write Policy (write-thru, write-back), Eviction Policy (LRU, ...)

# How might organization of TLB differ from that of a conventional instruction or data cache?

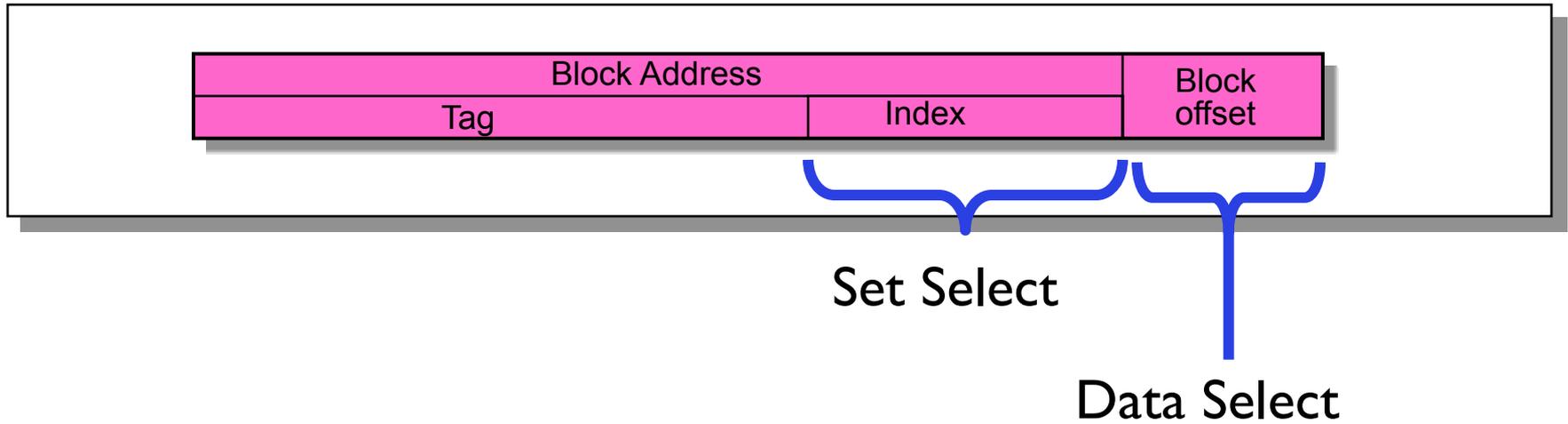
- Let's do some review ...

# A Summary on Sources of Cache Misses

---

- **Compulsory** (cold start or process migration, 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

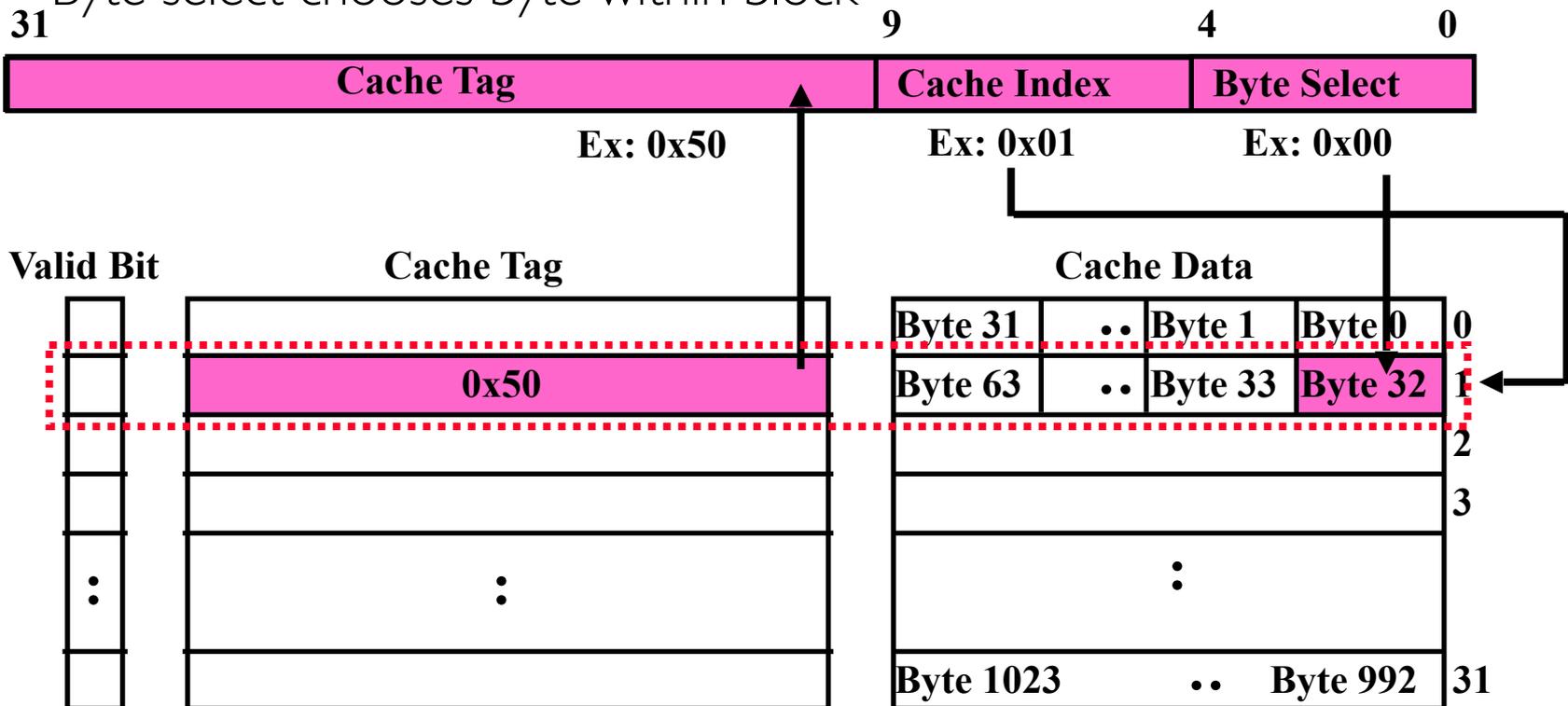
## How is a Block found in a Cache?



- **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

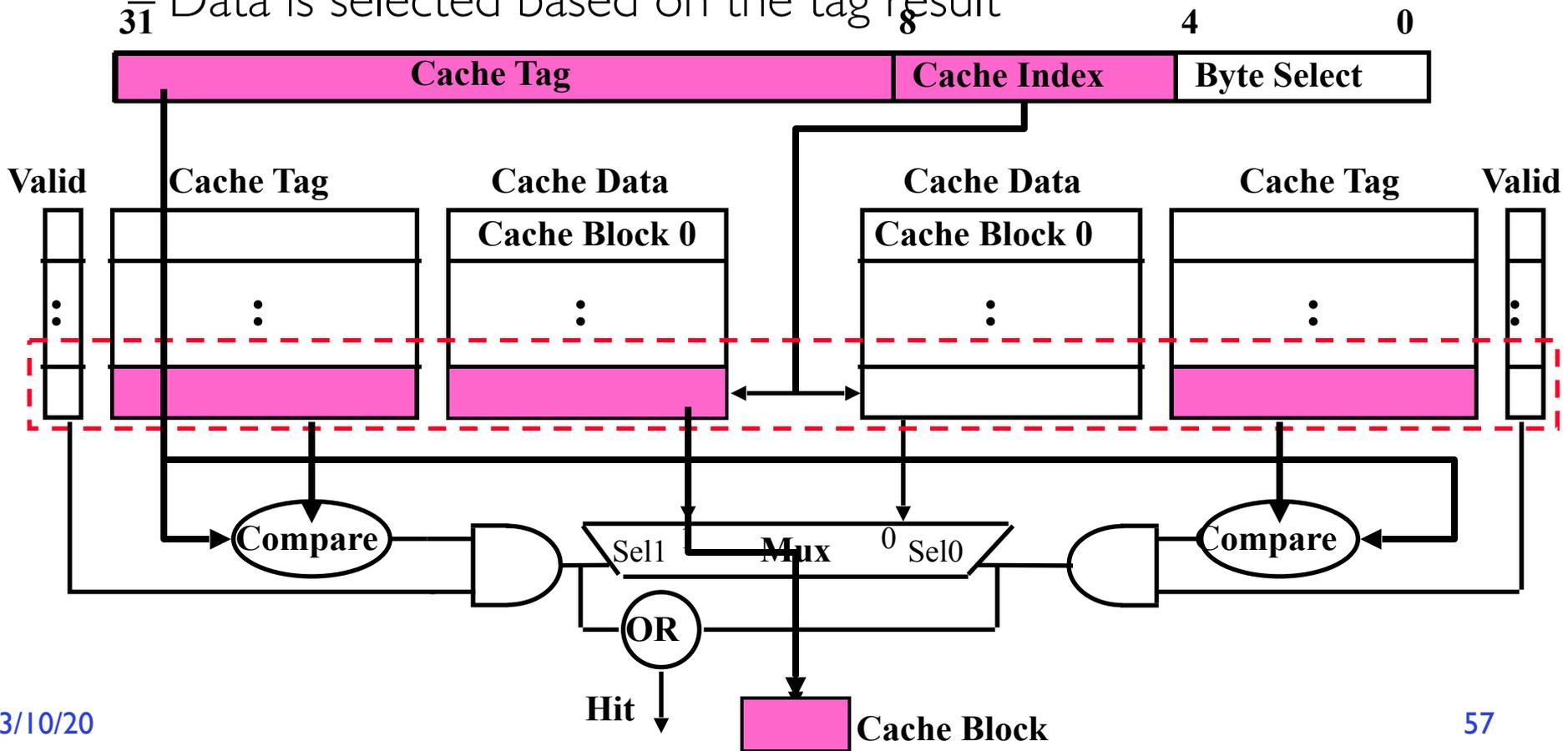
# Review: Direct Mapped Cache

- **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

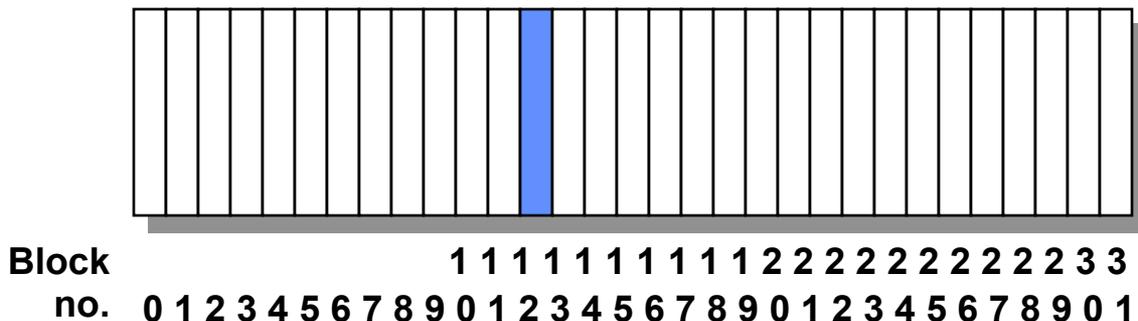




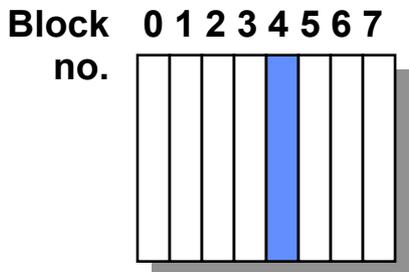
## Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache

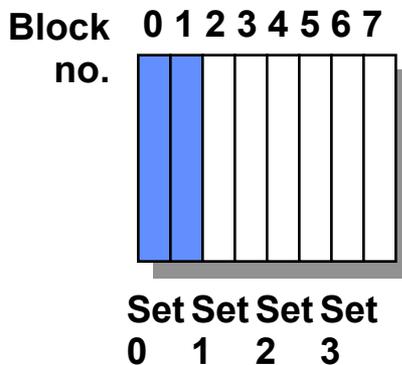
### 32-Block Address Space:



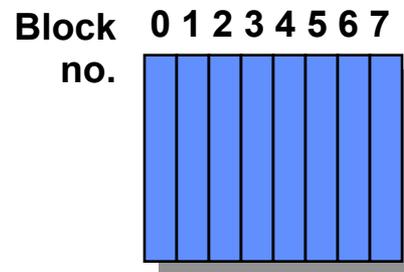
**Direct mapped:**  
 block 12 can go  
 only into block 4  
 ( $12 \bmod 8$ )



**Set associative:**  
 block 12 can go  
 anywhere in set 0  
 ( $12 \bmod 4$ )



**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)

- Miss rates for a workload:

Size	2-way		4-way		8-way	
	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

## Review: What happens on a write?

---

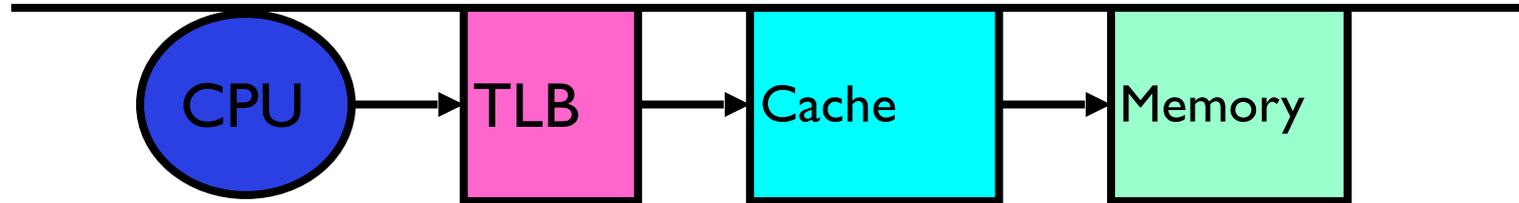
- **Write through**: The information is written to both the block in the cache and to the block in the lower-level memory
- **Write back**: The information is written only to the block in the cache
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean or dirty?
- Pros and Cons of each?
  - WT:
    - » PRO: read misses cannot result in writes
    - » CON: Processor held up on writes unless writes buffered
  - WB:
    - » PRO: repeated writes not sent to DRAM  
processor not held up on writes
    - » CON: More complex  
Read miss may require writeback of dirty data

# Questions about caches ?

---

- How does operating system behavior affect cache performance?
- Switching threads?
- Switching contexts?
- Cache design? What addresses are used?
- What does our understanding of caches tell us about TLB organization?

# 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”
- Example for MIPS R3000:

Virtual Address	Physical Address	Dirty	Ref	Valid	Access	ASID
<b>0xFA00</b>	<b>0x0003</b>	<b>Y</b>	<b>N</b>	<b>Y</b>	<b>R/W</b>	<b>34</b>
<b>0x0040</b>	<b>0x0010</b>	<b>N</b>	<b>Y</b>	<b>Y</b>	<b>R</b>	<b>0</b>
<b>0x0041</b>	<b>0x0011</b>	<b>N</b>	<b>Y</b>	<b>Y</b>	<b>R</b>	<b>0</b>

# Example: R3000 pipeline includes TLB “stages”

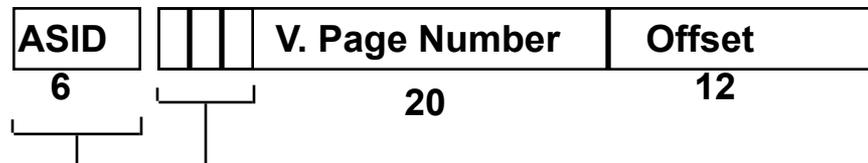
## MIPS R3000 Pipeline

Inst Fetch		Dcd/ Reg	ALU / E.A	Memory	Write Reg
TLB	I-Cache	RF	Operation		WB
			E.A.   TLB	D-Cache	

## TLB

64 entry, on-chip, fully associative, software TLB fault handler

## Virtual Address Space



- 0x User segment (caching based on PT/TLB entry)
- 100 Kernel physical space, cached
- 101 Kernel physical space, uncached
- 11x Kernel virtual space

Allows context switching among  
64 user processes without TLB flush

# Example: Pentium-M TLBs (2003)

---

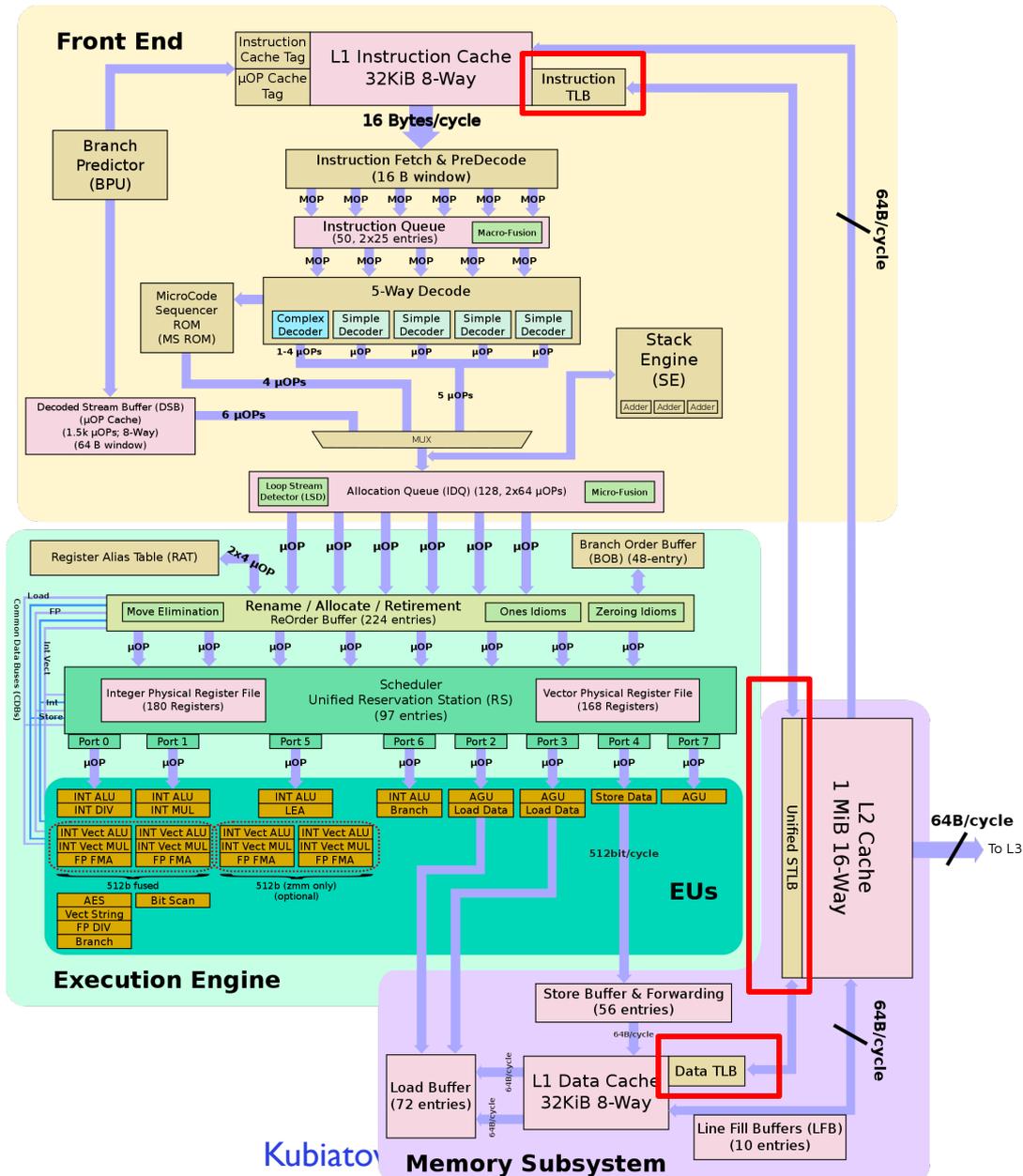
- Four different TLBs
  - Instruction TLB for 4K pages
    - » 128 entries, 4-way set associative
  - Instruction TLB for large pages
    - » 2 entries, fully associative
  - Data TLB for 4K pages
    - » 128 entries, 4-way set associative
  - Data TLB for large pages
    - » 8 entries, 4-way set associative
- All TLBs use LRU replacement policy
- Why different TLBs for instruction, data, and page sizes?

# Intel Nahelem (2008)

---

- L1 DTLB
  - 64 entries for 4 K pages and
  - 32 entries for 2/4 M pages,
- L1 ITLB
  - 128 entries for 4 K pages using 4-way associativity and
  - 14 fully associative entries for 2/4 MiB pages
- unified 512-entry L2 TLB for 4 KiB pages, 4-way associative.

# Current Intel x86 (Skylake, Cascade Lake)



# Current Example: Memory Hierarchy

---

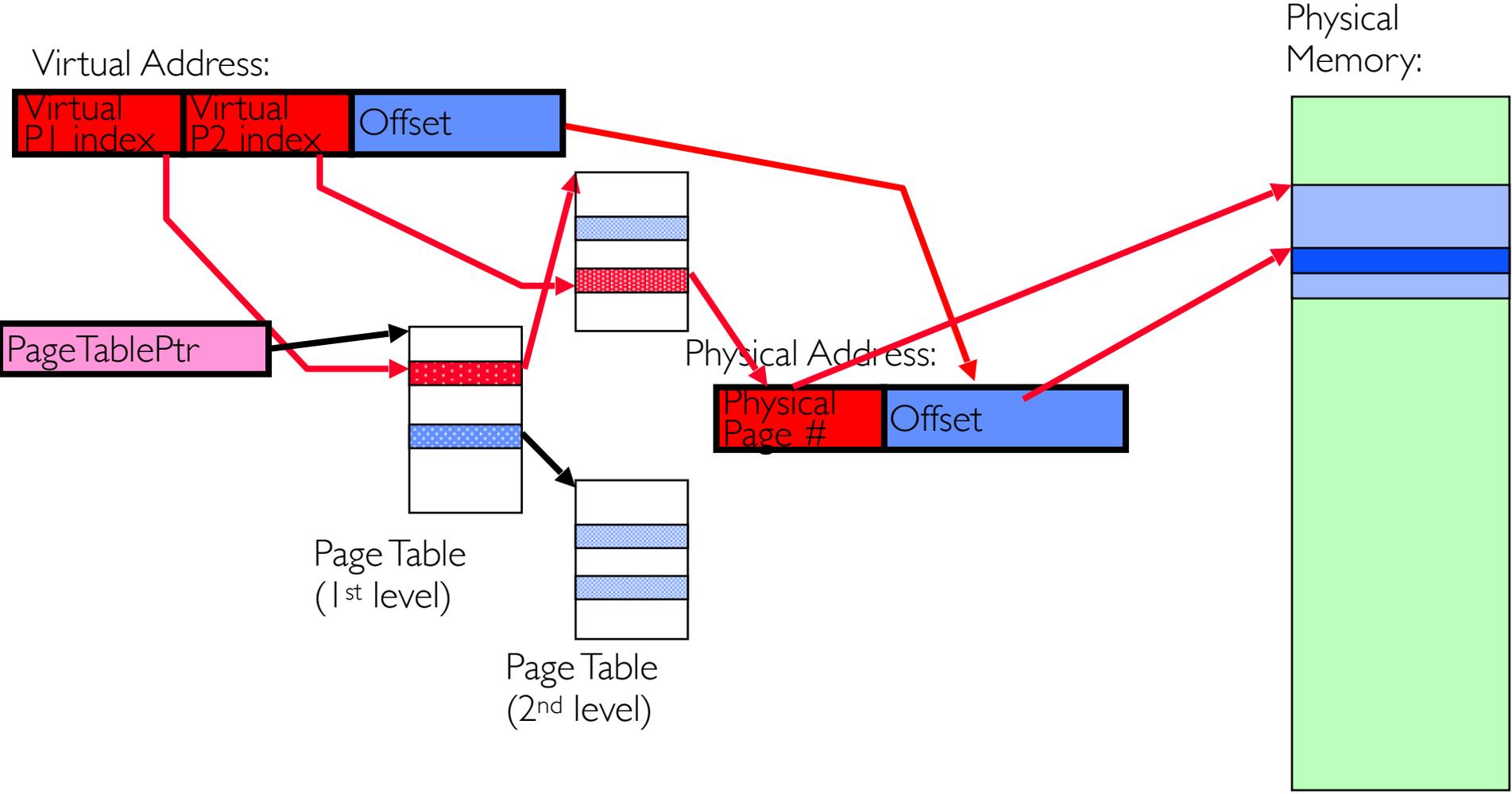
- Caches (all 64 B line size)
  - L1 I-Cache: 32 **KiB**/core, 8-way set assoc.
  - L1 D Cache: 32 KiB/core, 8-way set assoc., 4-5 cycles load-to-use, Write-back policy
  - L2 Cache: 1 MiB/core, 16-way set assoc., Inclusive, Write-back policy, 14 cycles latency
  - L3 Cache: 1.375 MiB/core, 11-way set assoc., shared across cores, Non-inclusive victim cache, Write-back policy, 50-70 cycles latency
- TLB
  - L1 ITLB, 128 entries; 8-way set assoc. for 4 KB pages
    - » 8 entries per thread; fully associative, for 2 MiB / 4 MiB page
  - L1 DTLB 64 entries; 4-way set associative for 4 KB pages
    - » 32 entries; 4-way set associative, 2 MiB / 4 MiB page translations:
    - » 4 entries; 4-way associative, 1 G page translations:
  - L2 STLB: 1536 entries; 12-way set assoc. 4 KiB + 2 MiB pages
    - » 16 entries; 4-way set associative, 1 GiB page translations:

# What happens on a Context Switch?

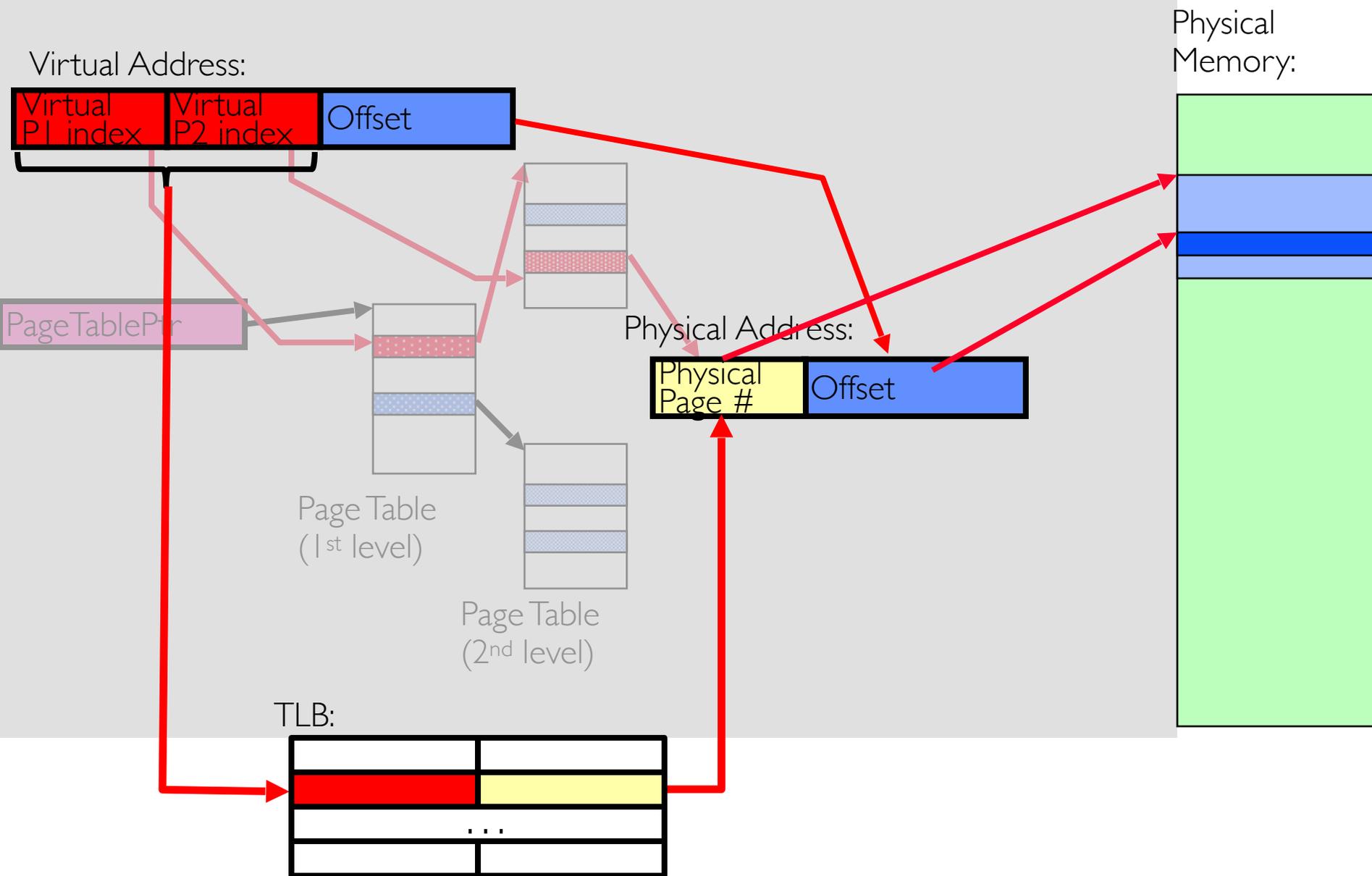
---

- 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”

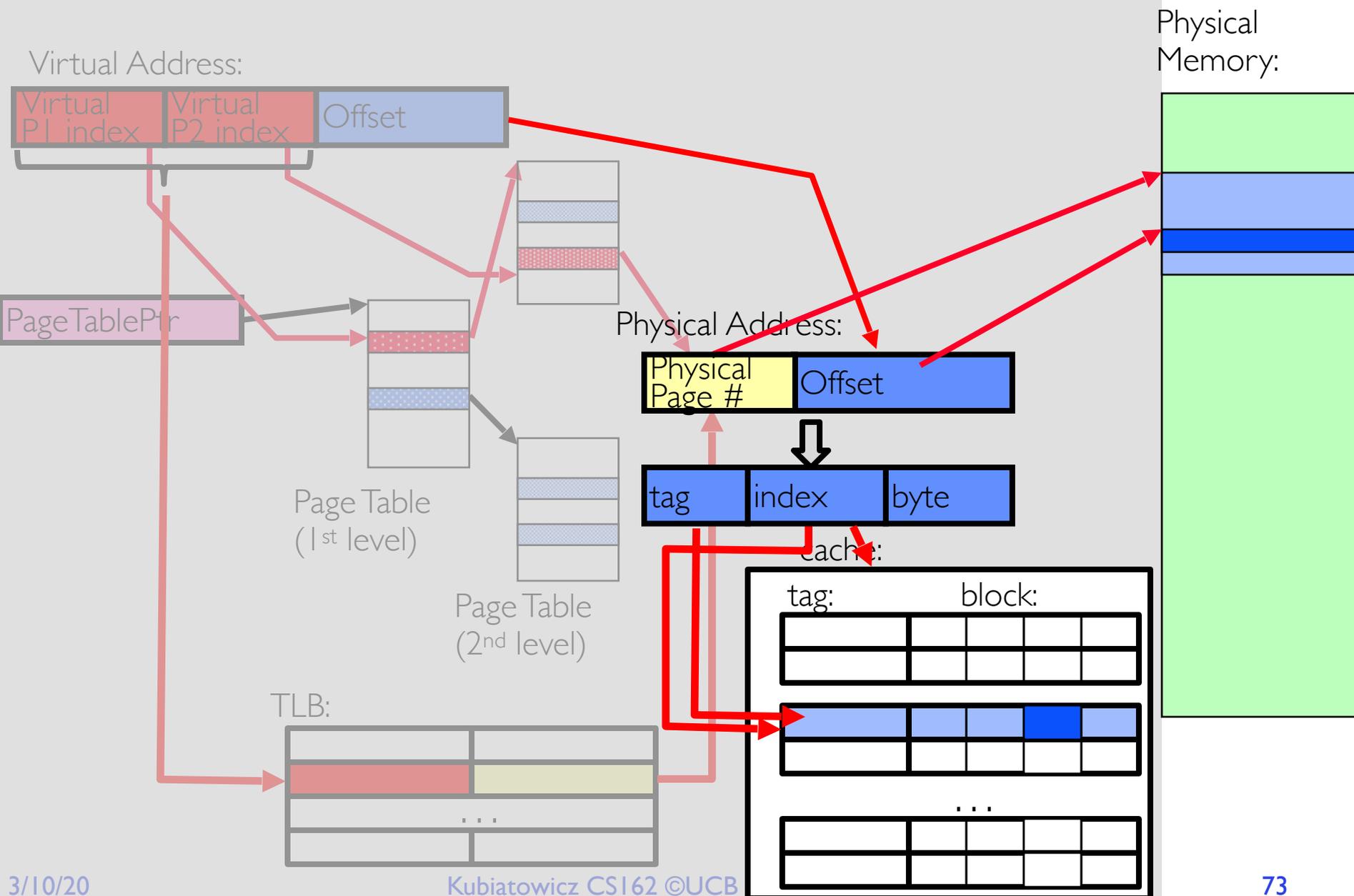
# Putting Everything Together: Address Translation



# Putting Everything Together: TLB

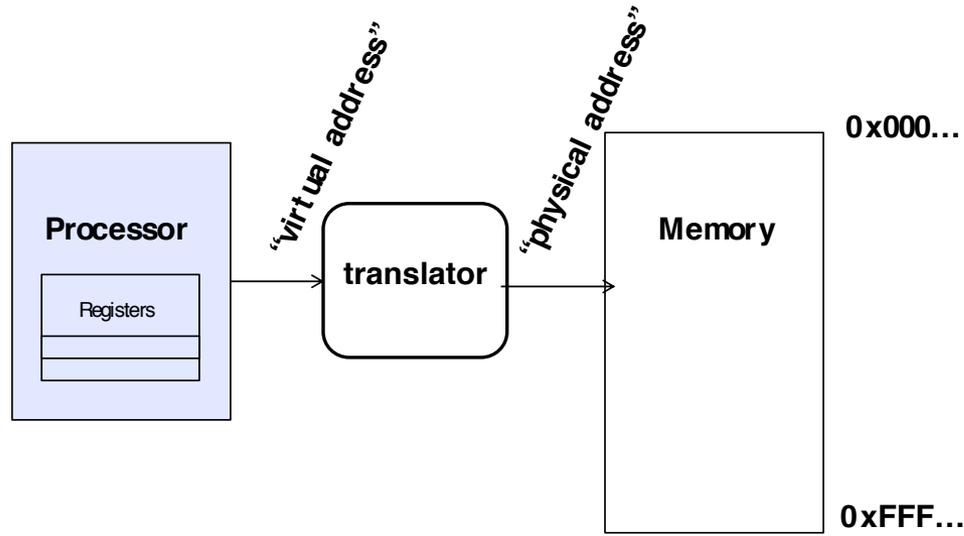


# Putting Everything Together: Cache



# Two Critical Issues in Address Translation

---



- How to translate addresses fast enough?
  - Every instruction fetch
  - Plus every load / store
  - EVERY MEMORY REFERENCE !
  - More than one translation for EVERY instruction
- Next: What to do if the translation fails?
  - Page fault! This is a synchronous exception!

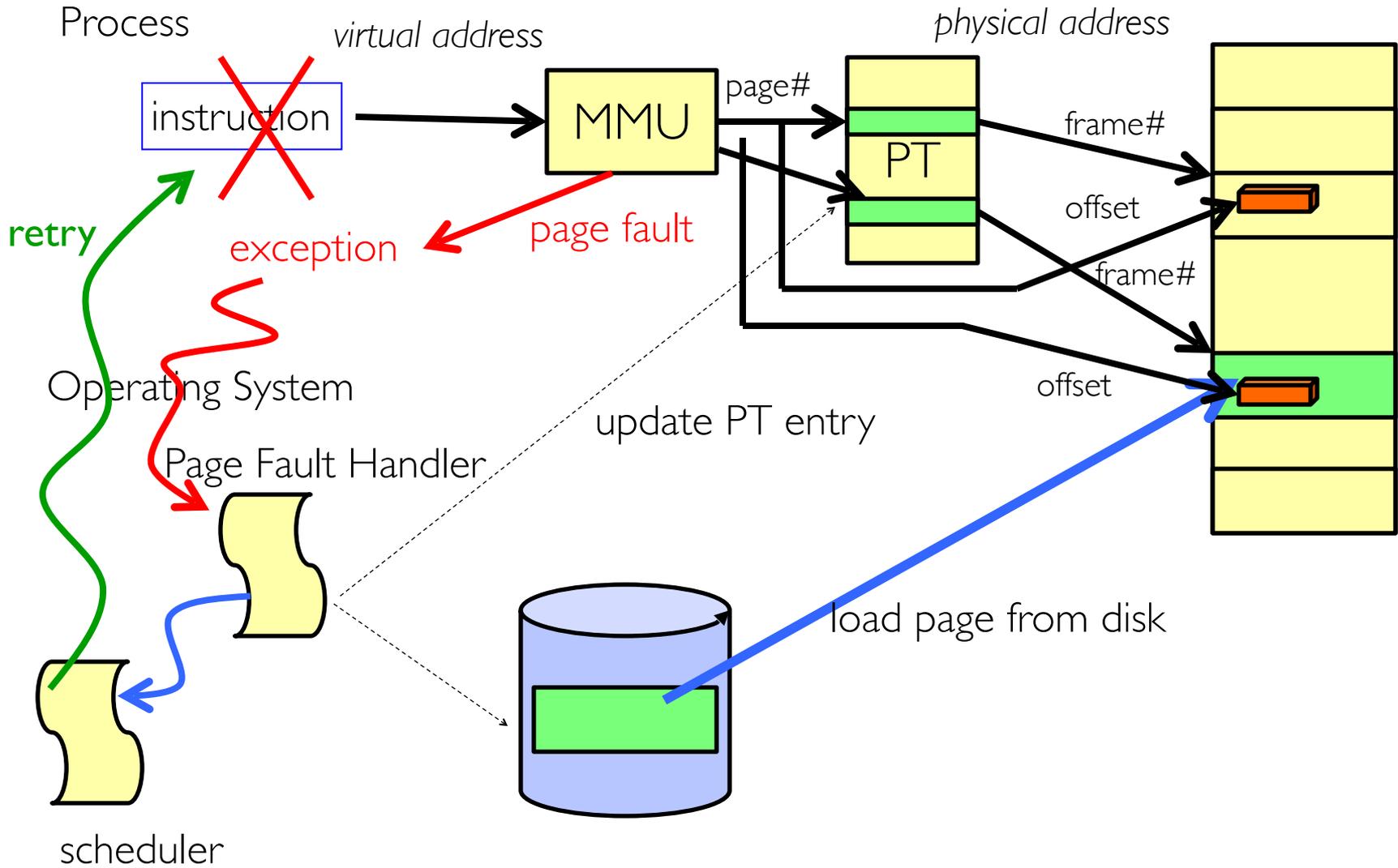
- A system call instruction causes a synchronous exception (or “trap”)
  - In fact, often called a software “trap” instruction
- Other sources of **Synchronous Exceptions (“Trap”)**:
  - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault (for illusion of infinite-sized memory)
- Interrupts are **Asynchronous Exceptions**:
  - Examples: timer, disk ready, network, etc....
  - **Interrupts can be disabled, traps cannot!**
- On system call, exception, or interrupt:
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - Some processors (e.g. x86) also save registers, changes stack
- Handler does any required state preservation not done by CPU:
  - Might save registers, other CPU state, and switches to kernel stack

# Page Fault

---

- The Virtual-to-Physical Translation fails
  - PTE marked invalid, Priv. Level Violation, Access violation, or does not exist
  - Causes an Fault / Trap
    - » Not an interrupt because synchronous to instruction execution
  - May occur on instruction fetch or data access
  - Protection violations typically terminate the instruction
- Other Page Faults engage operating system to fix the situation and retry the instruction
  - Allocate an additional stack page, or
  - Make the page accessible - Copy on Write,
  - Bring page in from secondary storage to memory – demand paging
- Fundamental inversion of the hardware / software boundary

# Next Up: What happens when ...



# 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  $\sim$  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)
- Next Time: What to do on a page fault?