CS162
Operating Systems and Systems Programming
Lecture 15

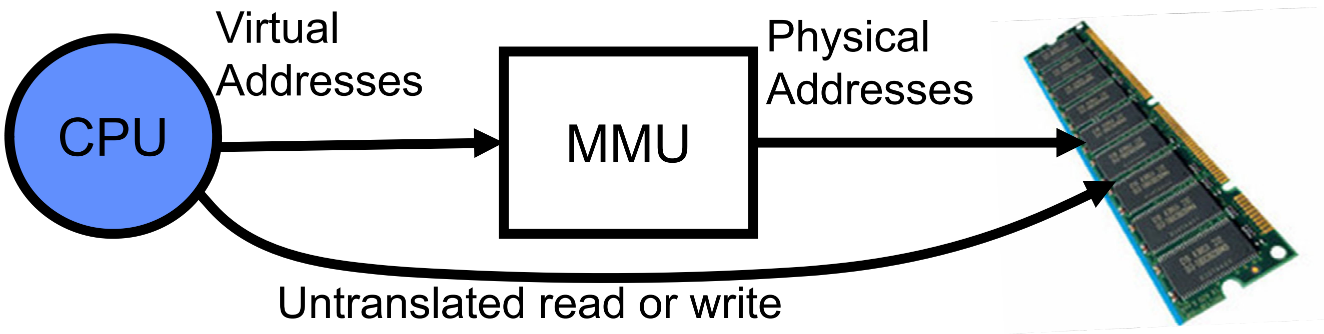
Memory 2: Paging (Con’t), Caching and TLBs

March 7th, 2024

Prof. John Kubiatowicz

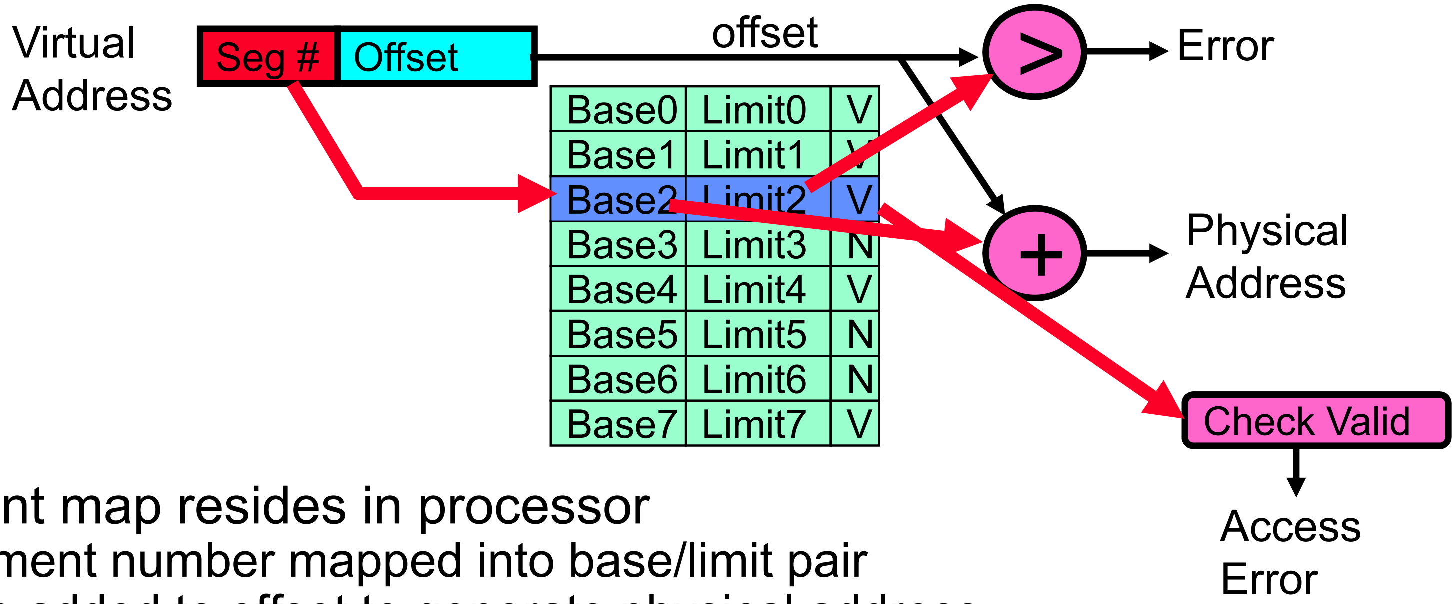
http://cs162.eecs.Berkeley.edu

Recall: General Address translation



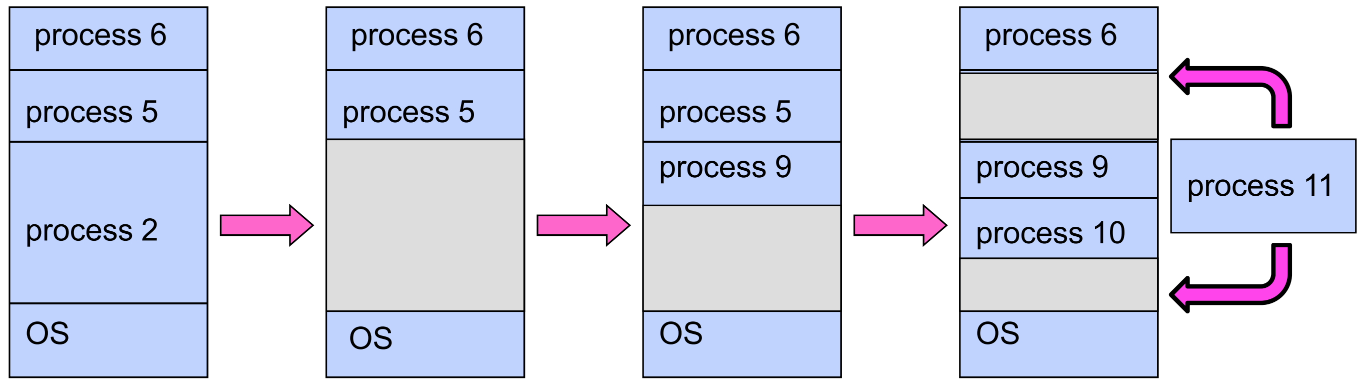
- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (Memory Management Unit or MMU) converts between the two views
- Translation ⇒ much easier to implement protection!
  - If task A cannot even gain access to task B's data, no way for A to adversely affect B
  - Extra benefit: every program can be linked/loaded into same region of user address space

Recall: 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



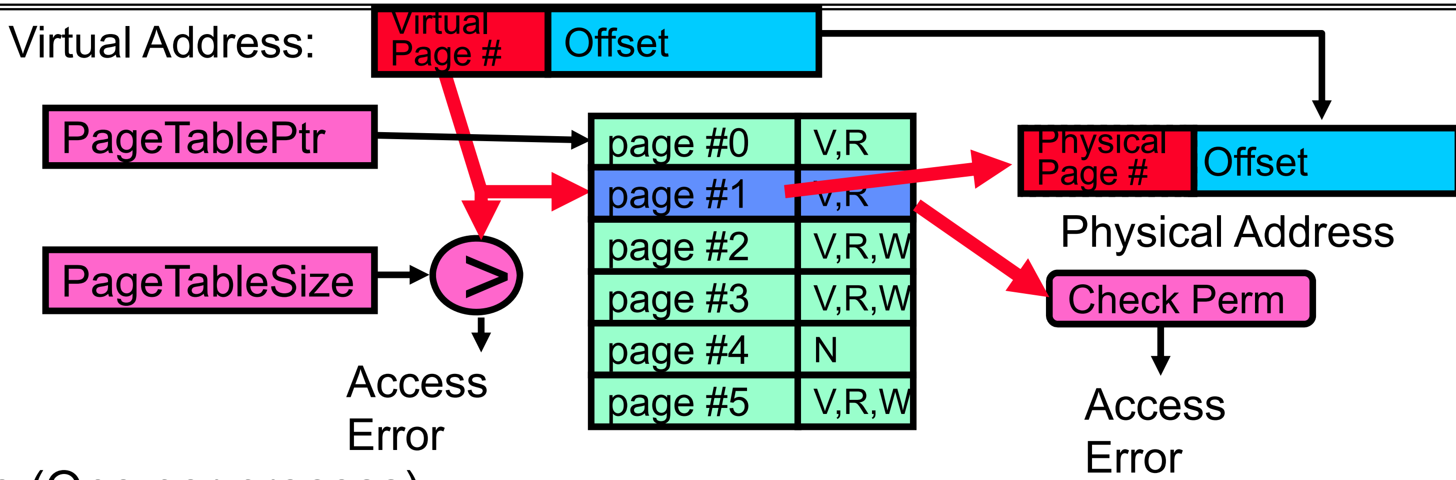
- Must fit variable-sized chunks into physical memory
  - Complicated allocation algorithms in kernel
- May move processes multiple times to fit everything
  - Lots of wasted time copying
- Limited options for swapping to disk
  - All or nothing: Can't have *part* of a segment
- Fragmentation: wasted 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` ⇒ allocated, `0` ⇒ free

- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment

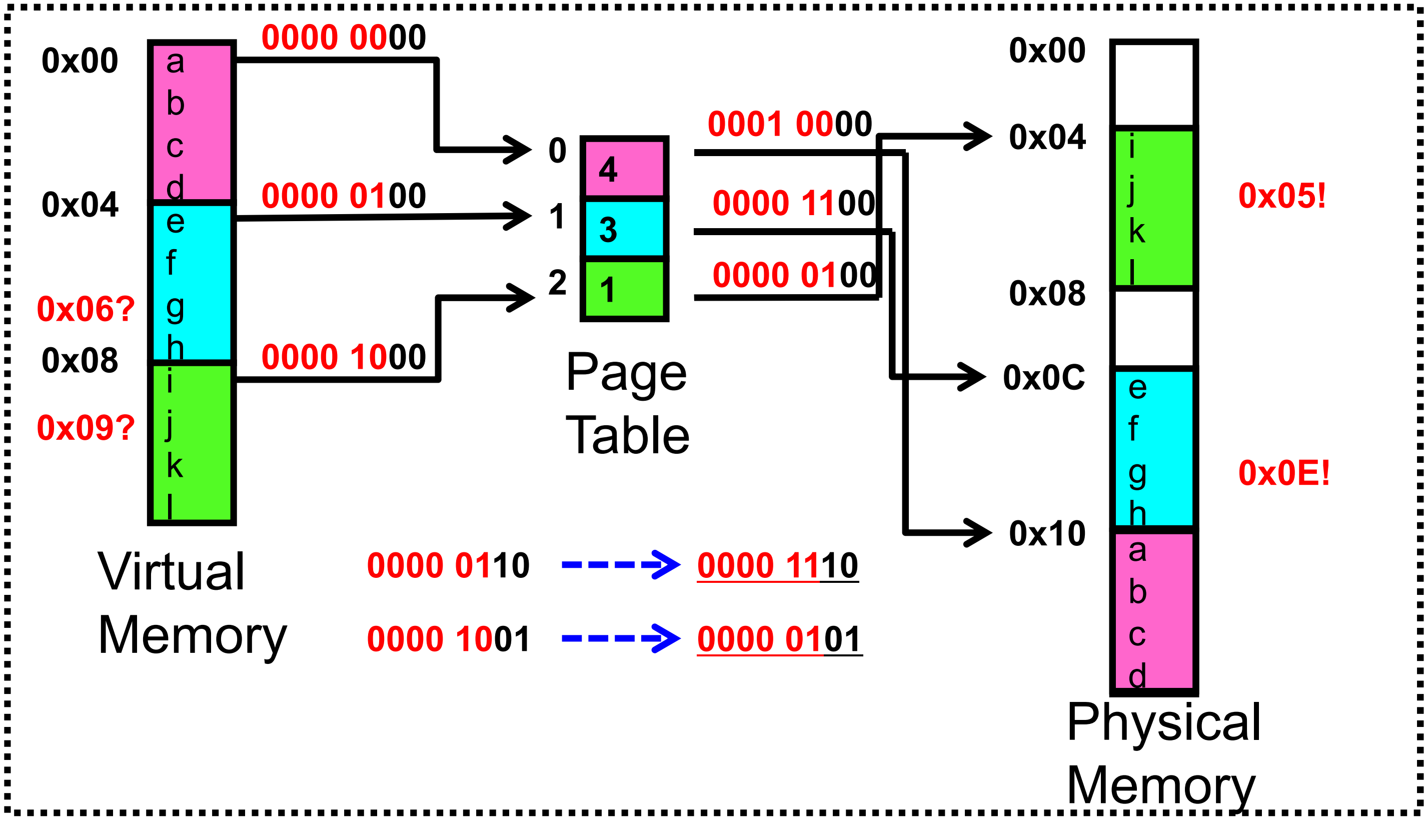
How to Implement Simple Paging?



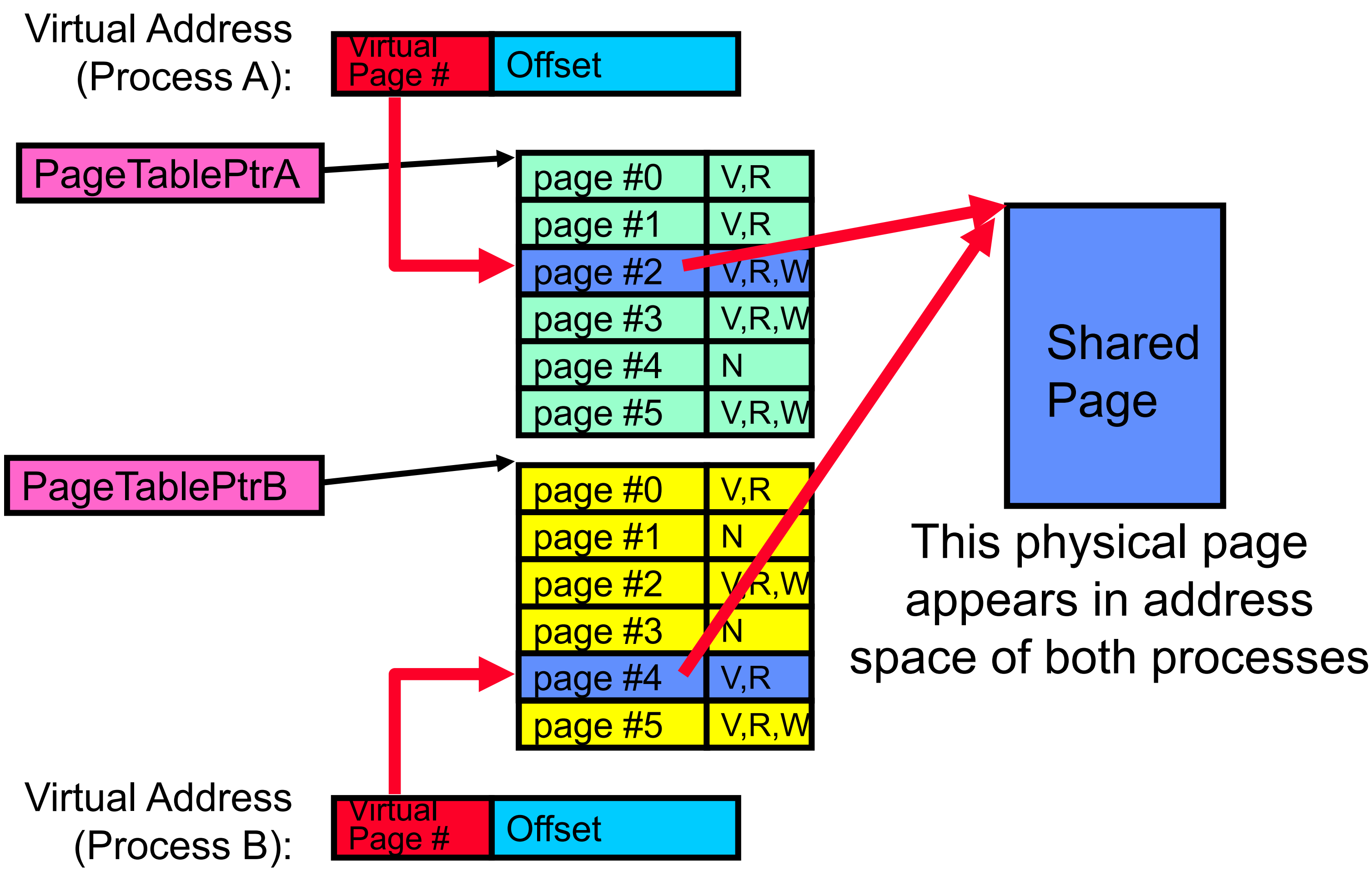
- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc)
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual 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

Simple Page Table Example

Example (4 byte pages)



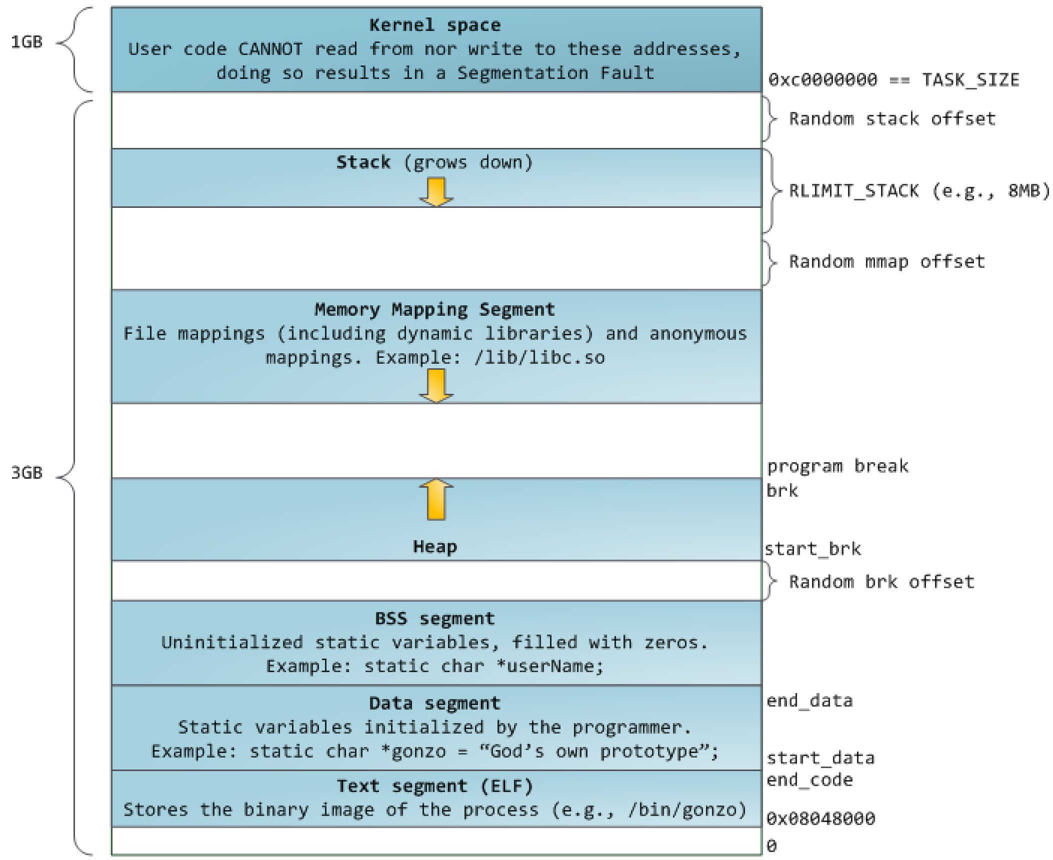
What about Sharing?



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

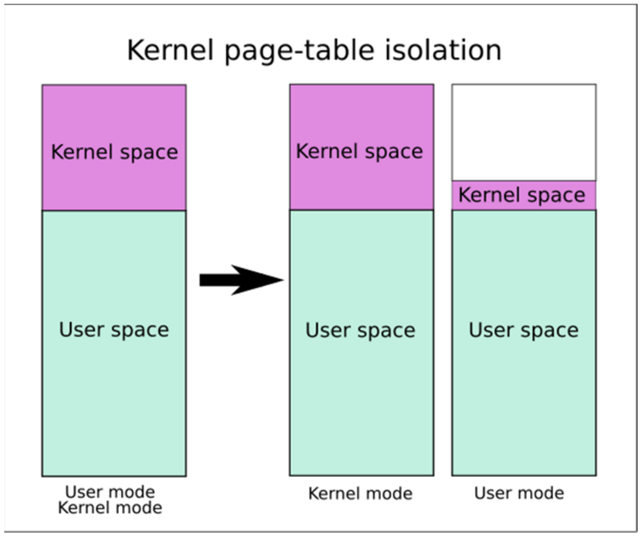
Recall: Memory Layout for Linux 32-bit (Pre-Meltdown patch!)



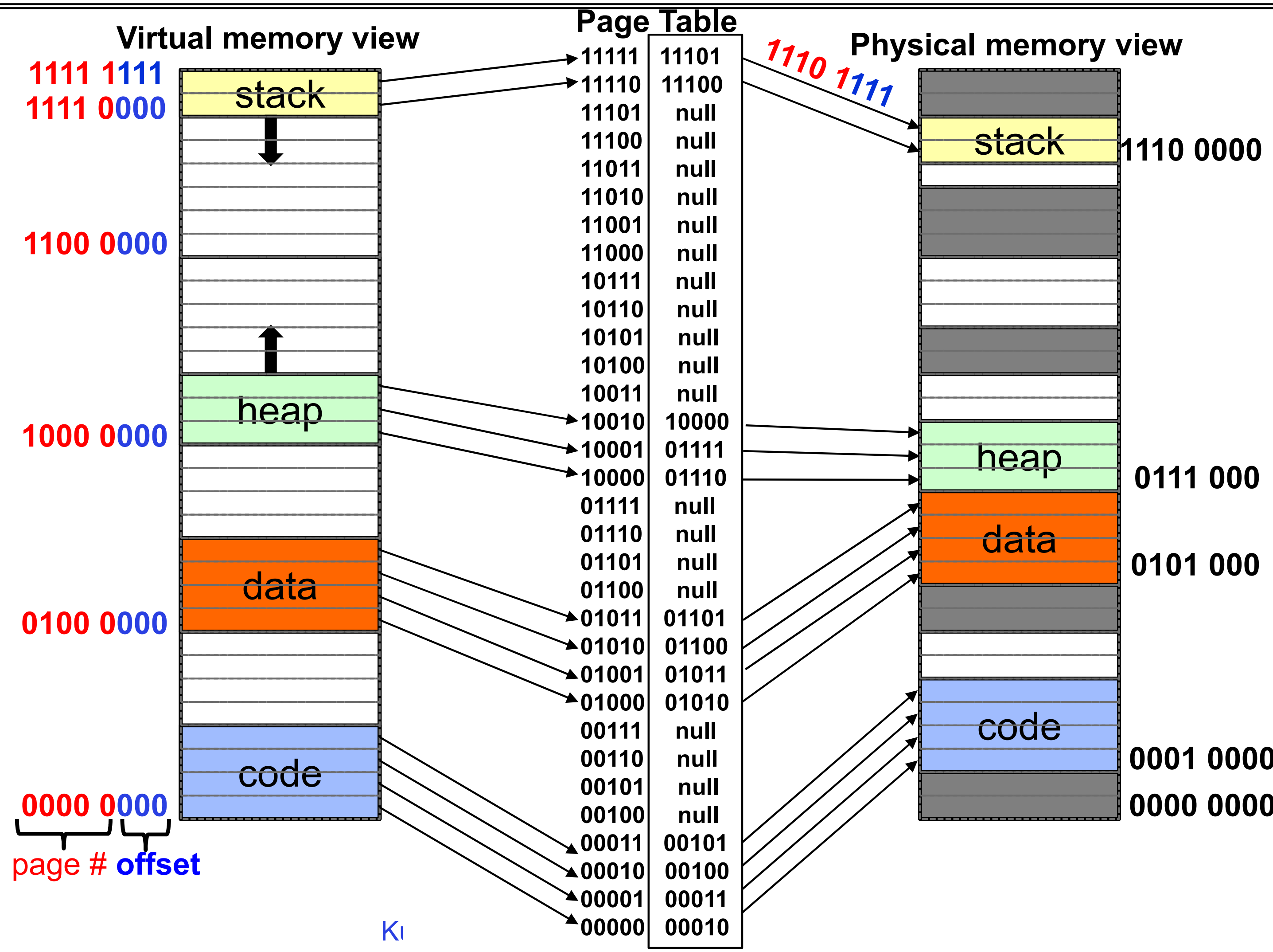
http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png

Some simple security measures

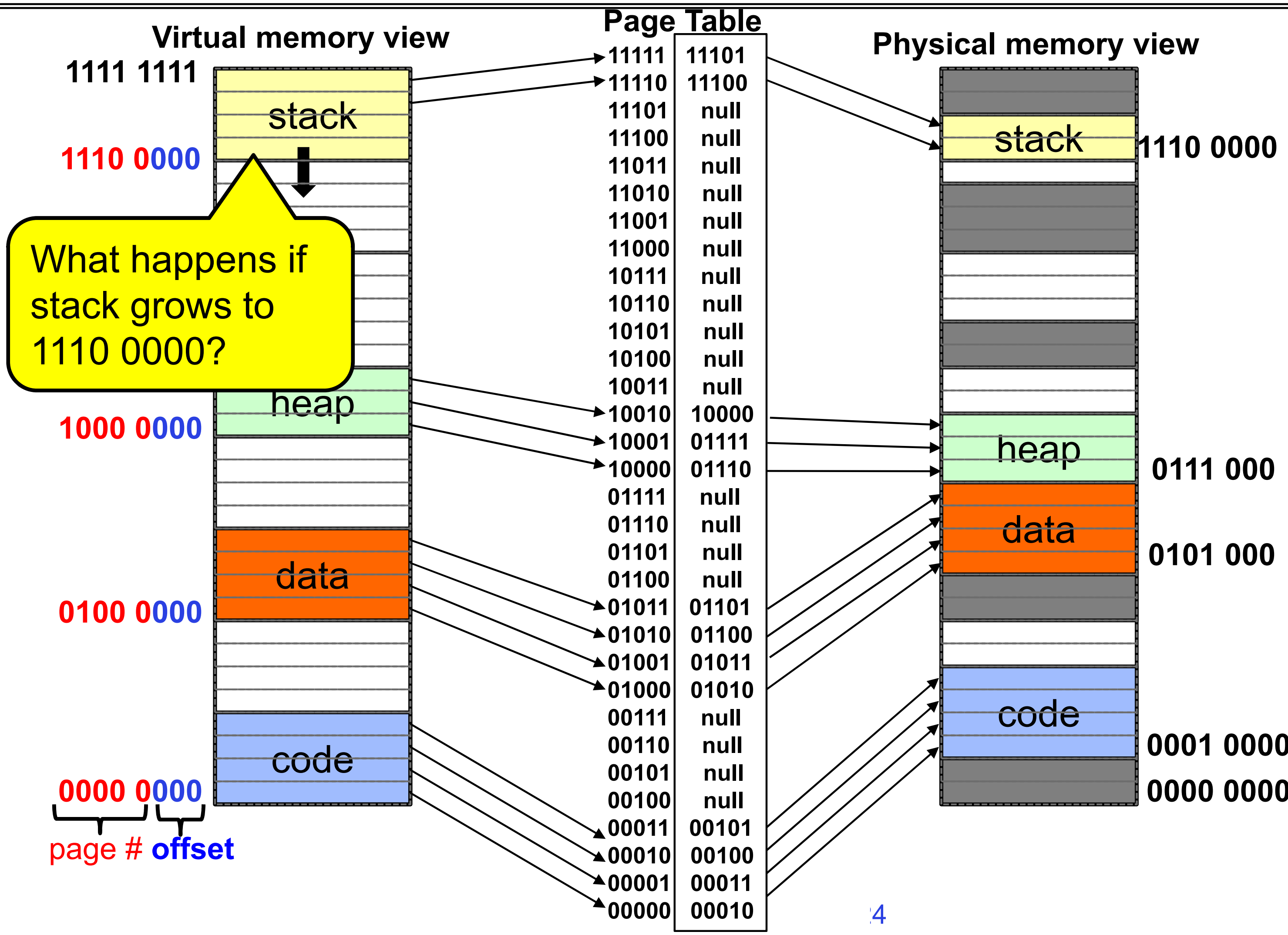
- Address Space Randomization
  - Position-Independent Code ⇒ can place user code anywhere in address space
    - » Random start address makes much harder for attacker to cause jump to code that it seeks to take over
  - Stack & Heap can start anywhere, so randomize placement
- Kernel address space isolation
  - Don't map whole kernel space into each process, switch to kernel page table
  - Meltdown⇒map none of kernel into user mode!



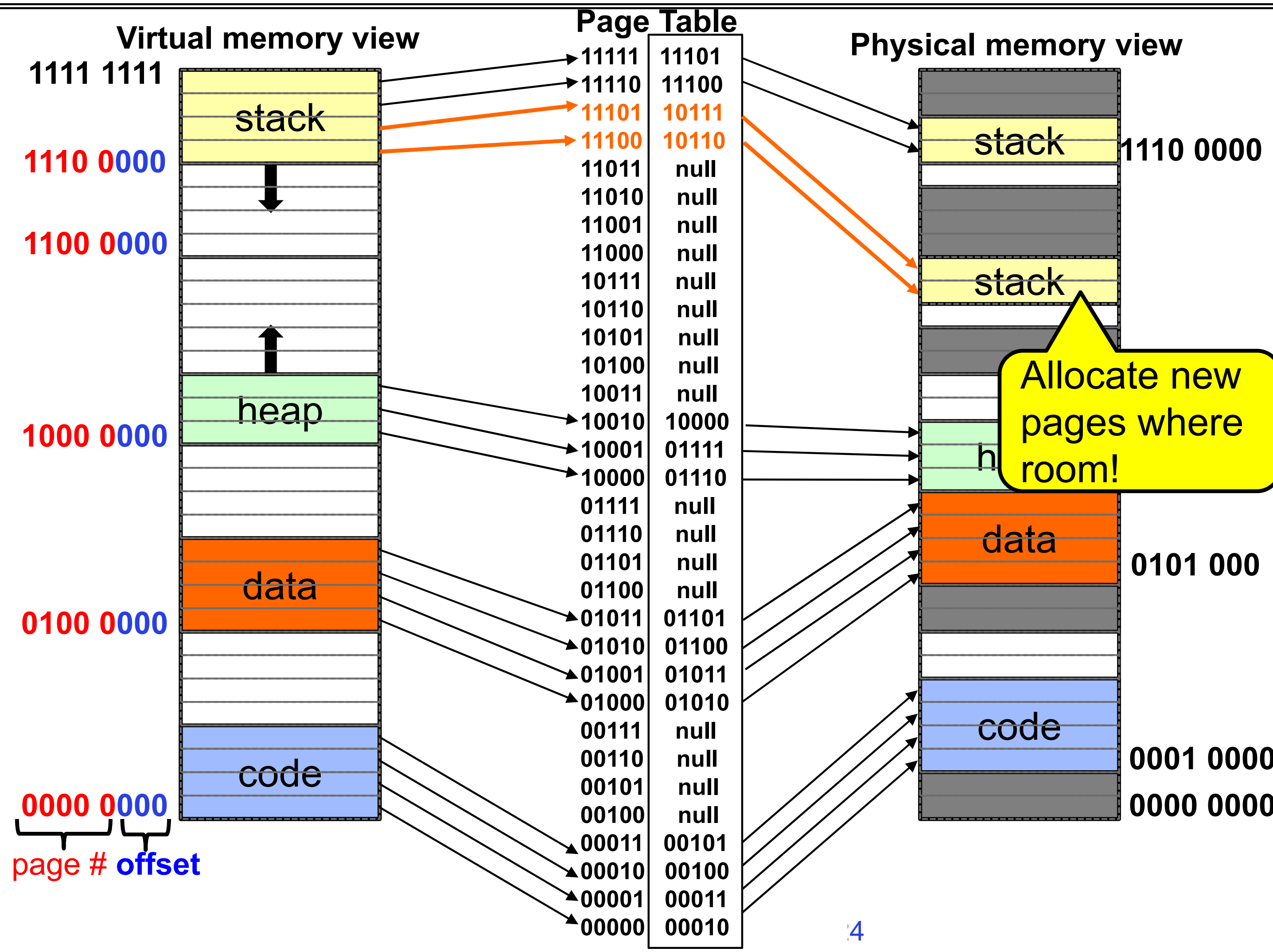
Summary: Paging



Summary: Paging



Summary: Paging



Administrivia

- Midterm 2: Thursday 3/14 from 8-10PM
  – A week from today!!!
  – All material up to Lecture 16 technically in bounds
  – Closed book: with two double-sided handwritten sheets of notes
- Homework 4 is out
  – Released yesterday, Wednesday 3/6
- Project 2 design document due this Friday!
- Starting next week – will have an opportunity to get extra credit participation points by attending lecture
  – Details to follow
- Reminder: Kubi Office Hours
  – Monday 1:00PM—2:00PM
  – Thursday 3:00PM—4:00PM

How big do things get?

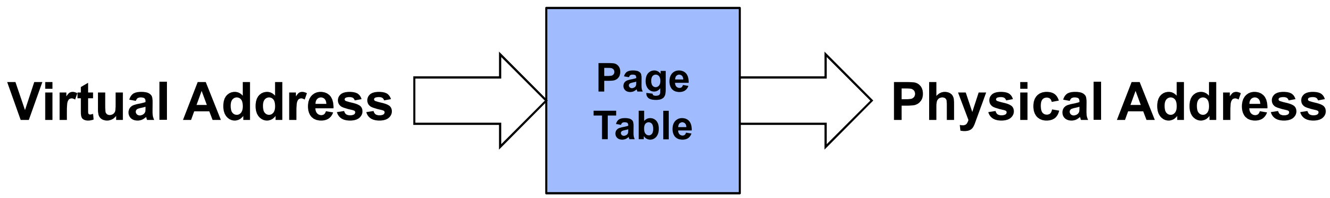
- 32-bit address space => 2^{32} bytes (4 GB)
  - Note: "b" = bit, and "B" = byte
  - And *for memory*:
    - » "K"(kilo) = 2^{10} = 1024 ≈ 10^3 (But not quite!): Sometimes called "Ki" (Kibi)
    - » "M"(mega) = 2^{20} = $(1024)^2$ = 1,048,576 ≈ 10^6 (But not quite!): Sometimes called "Mi" (Mibi)
    - » "G"(giga) = 2^{30} = $(1024)^3$ = 1,073,741,824 ≈ 10^9 (But not quite!): Sometimes called "Gi" (Gibi)
- Typical page size: 4 KB
  - how many bits of the address is that ? (remember 2^{10} = 1024)
  - Ans – 4KB = $4\times 2^{10} = 2^{12} \Rightarrow$ 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) x 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 4.5×10^{15} or 4.5 exa-entries)$\times$8 bytes each =
    36×10^{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!!!
- The address space is *sparse*, i.e. has holes 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 2 million page table entries!
  - Con: What if table really big?
    - Not all pages used all the time ⇒ 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

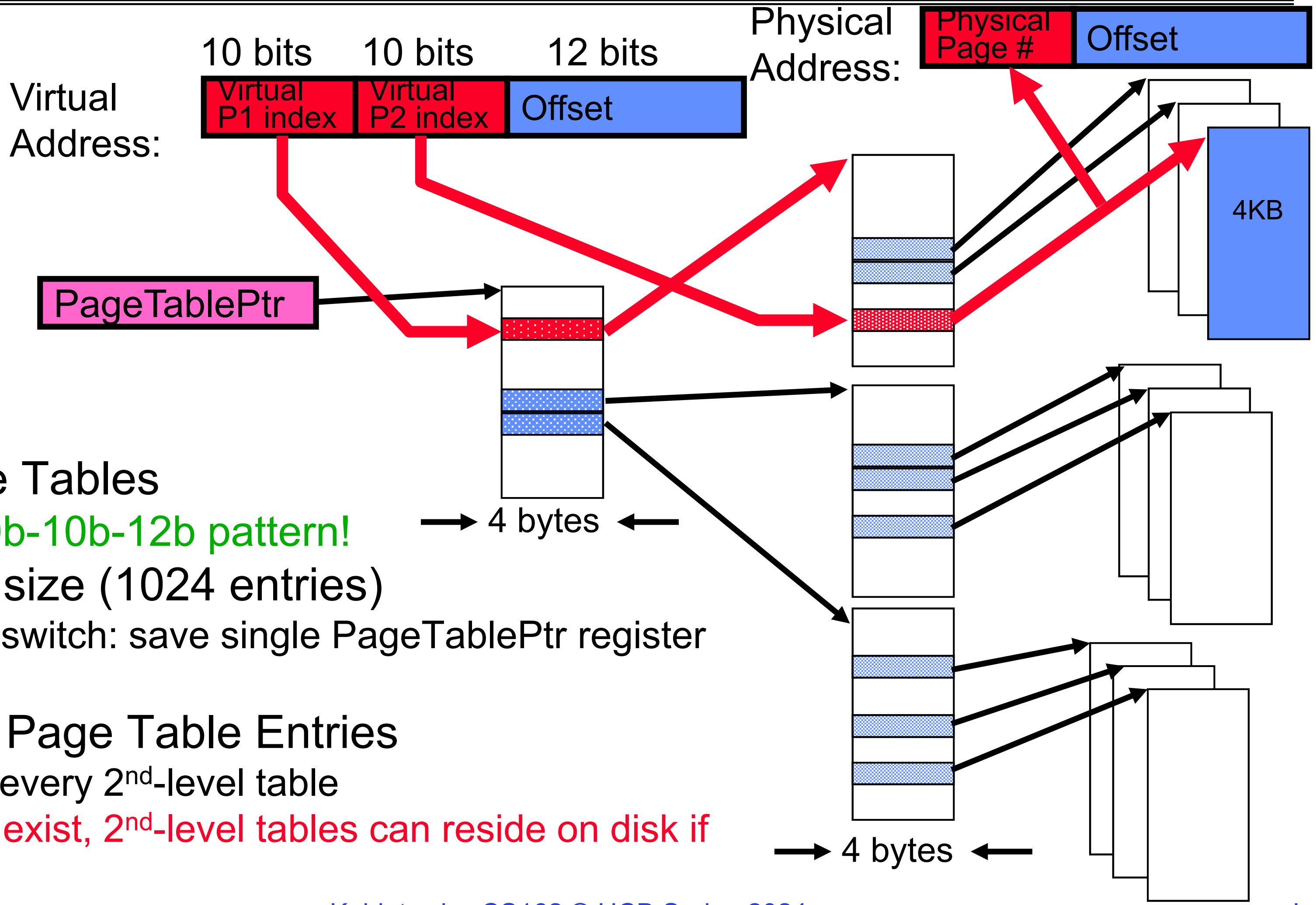
- Page Table is a *map* (function) from VPN to PPN



- 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

Example: x86 classic 32-bit address translation

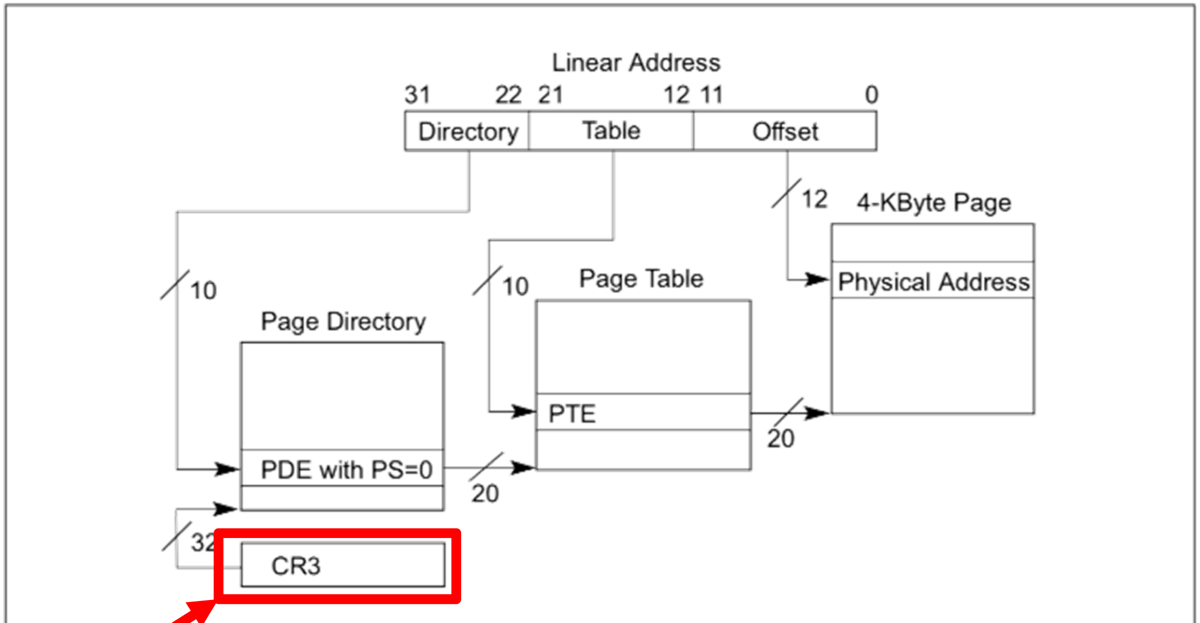


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

- 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"

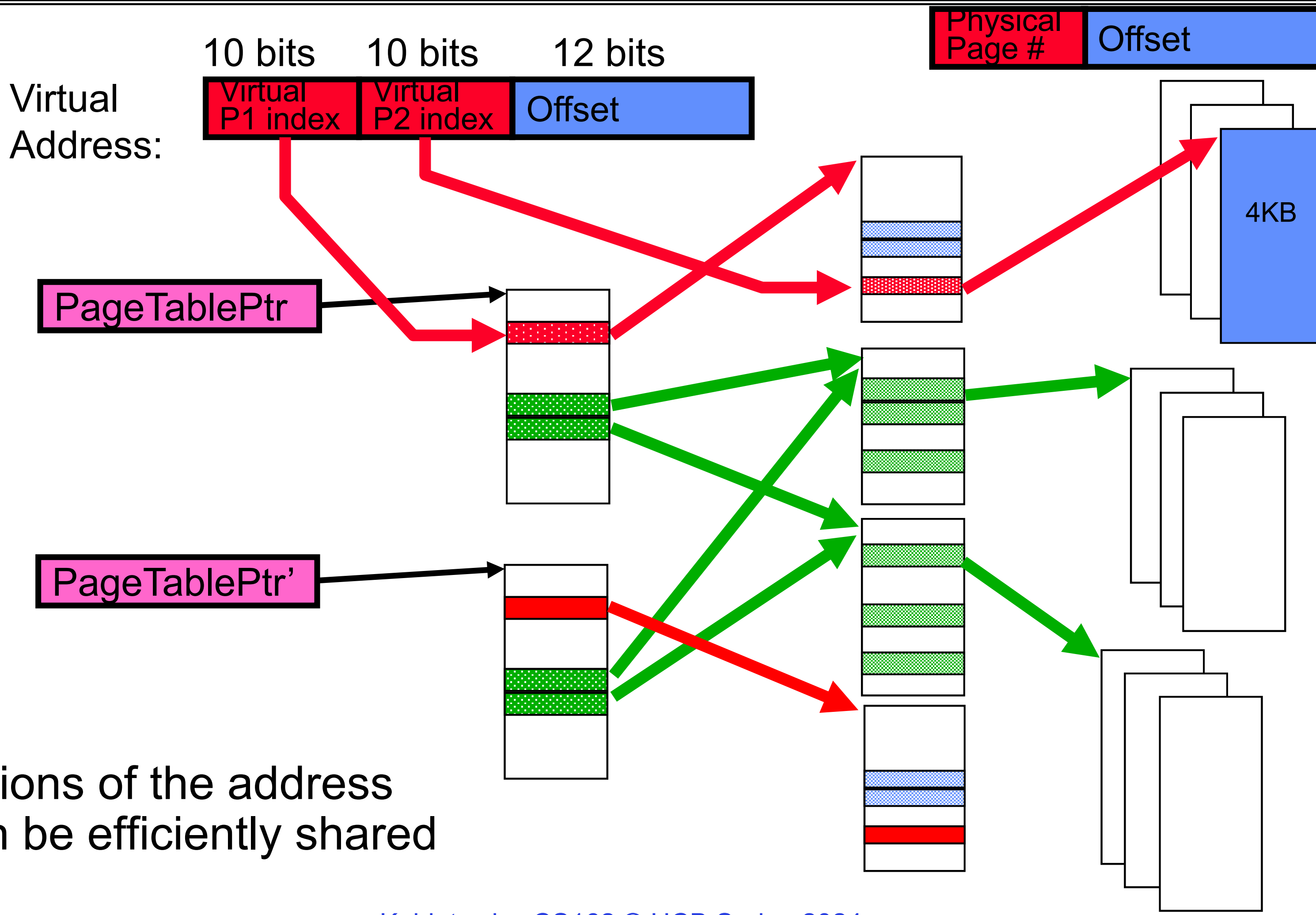
Page Frame Number (Physical Page Number)	**Free (OS)**	**0**	**PS**	**D**	**A**	**PCD**	**PWT**	**U**	**W**	**P**
31-12	**11-9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

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

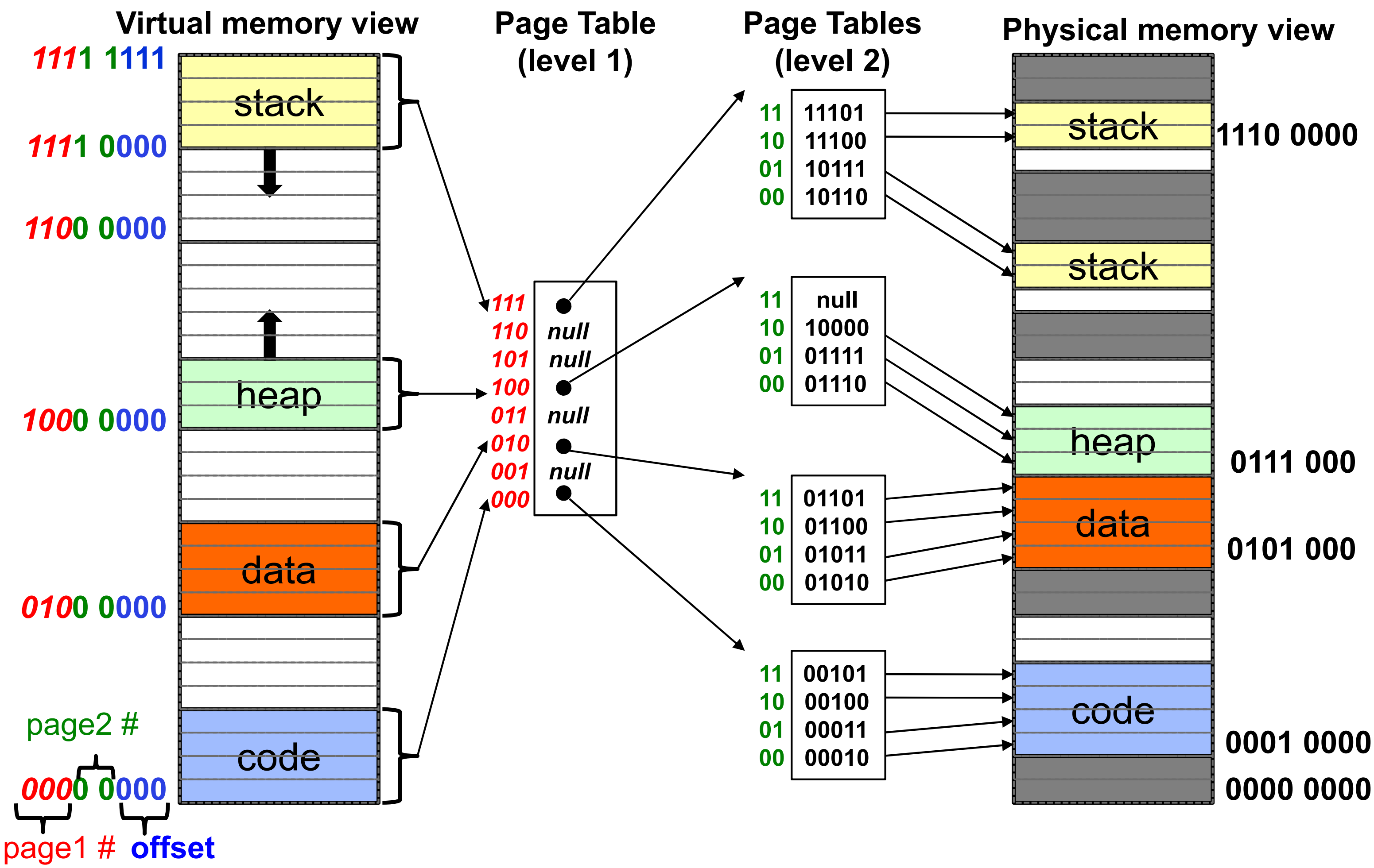
- How do we use the PTE?
  - Invalid PTE can imply different things:
    - » Region of address space is actually invalid or
    - » Page/directory is just somewhere else than memory
  - Validity checked first
    - » OS can use other (say) 31 bits for location info
- Usage Example: Demand Paging
  - 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

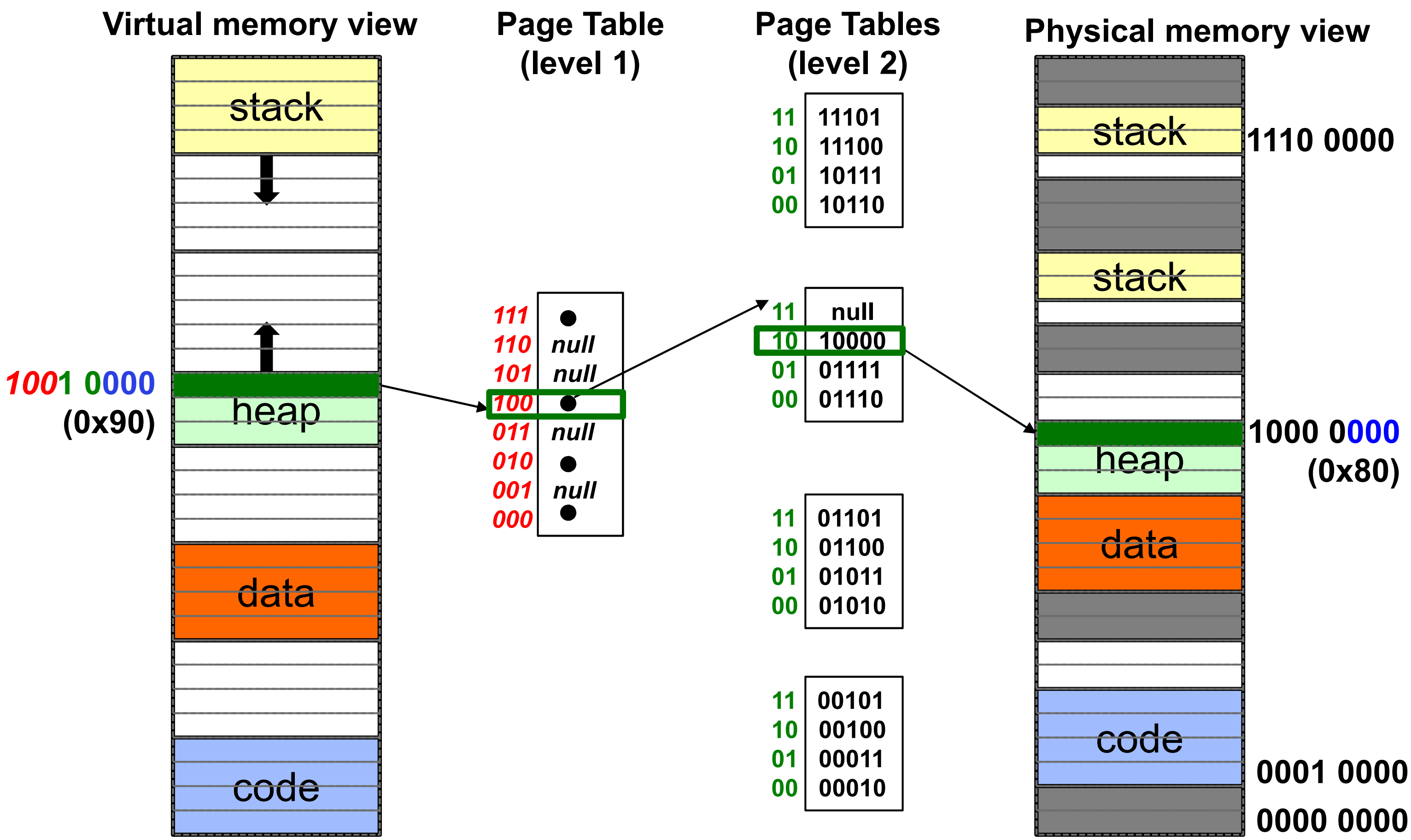


- Entire regions of the address space can be efficiently shared

Summary: Two-Level Paging

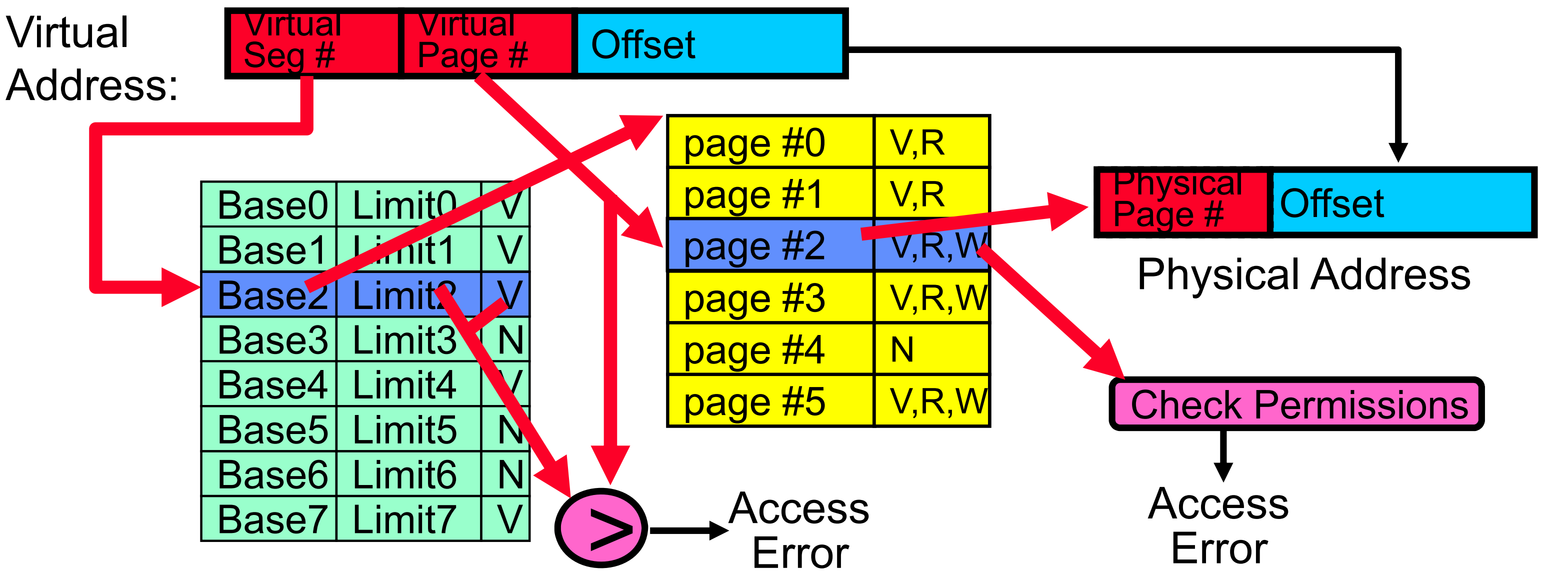


Summary: Two-Level Paging



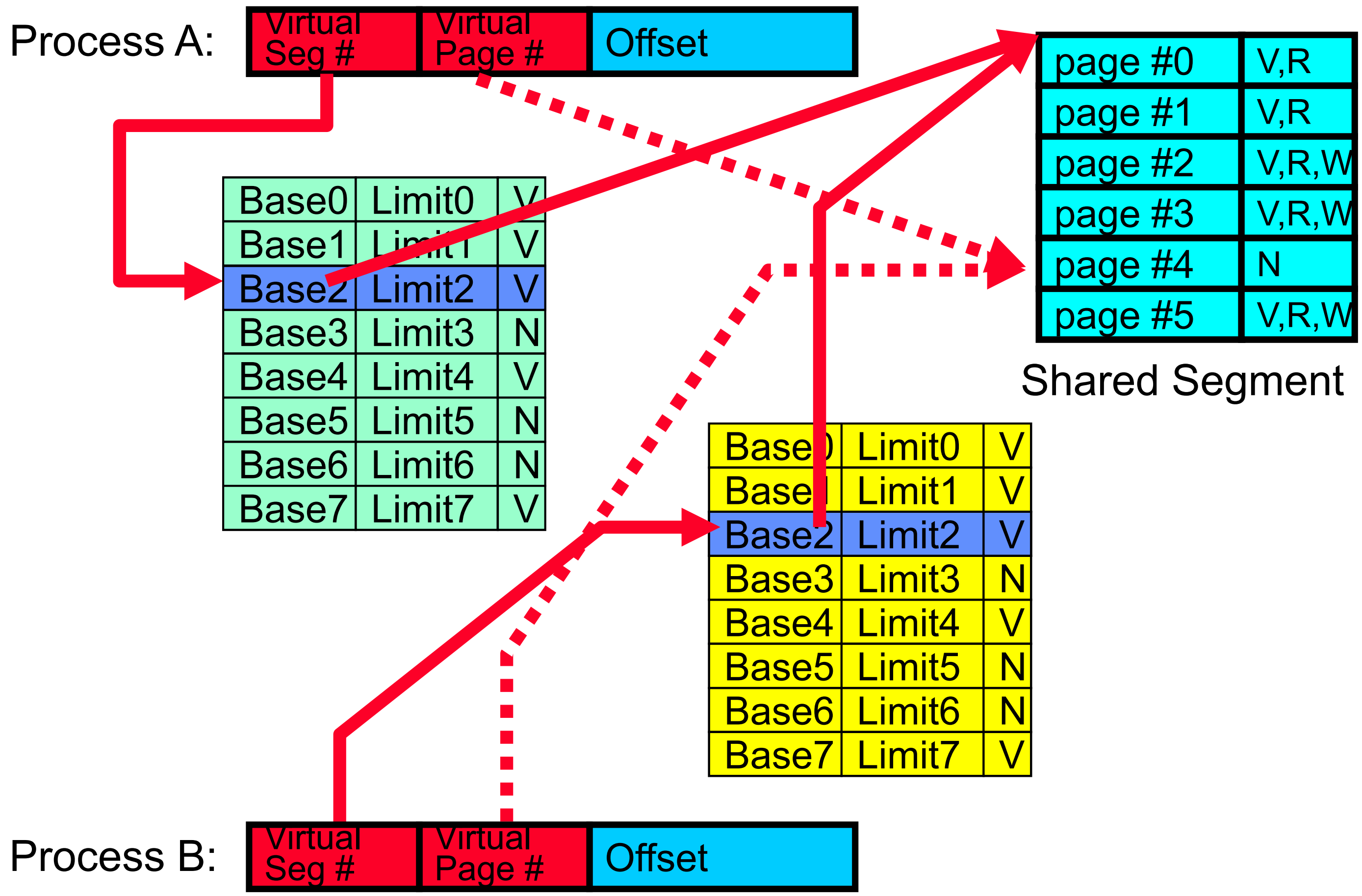
Multi-level Translation: Segments + Pages

- 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)?



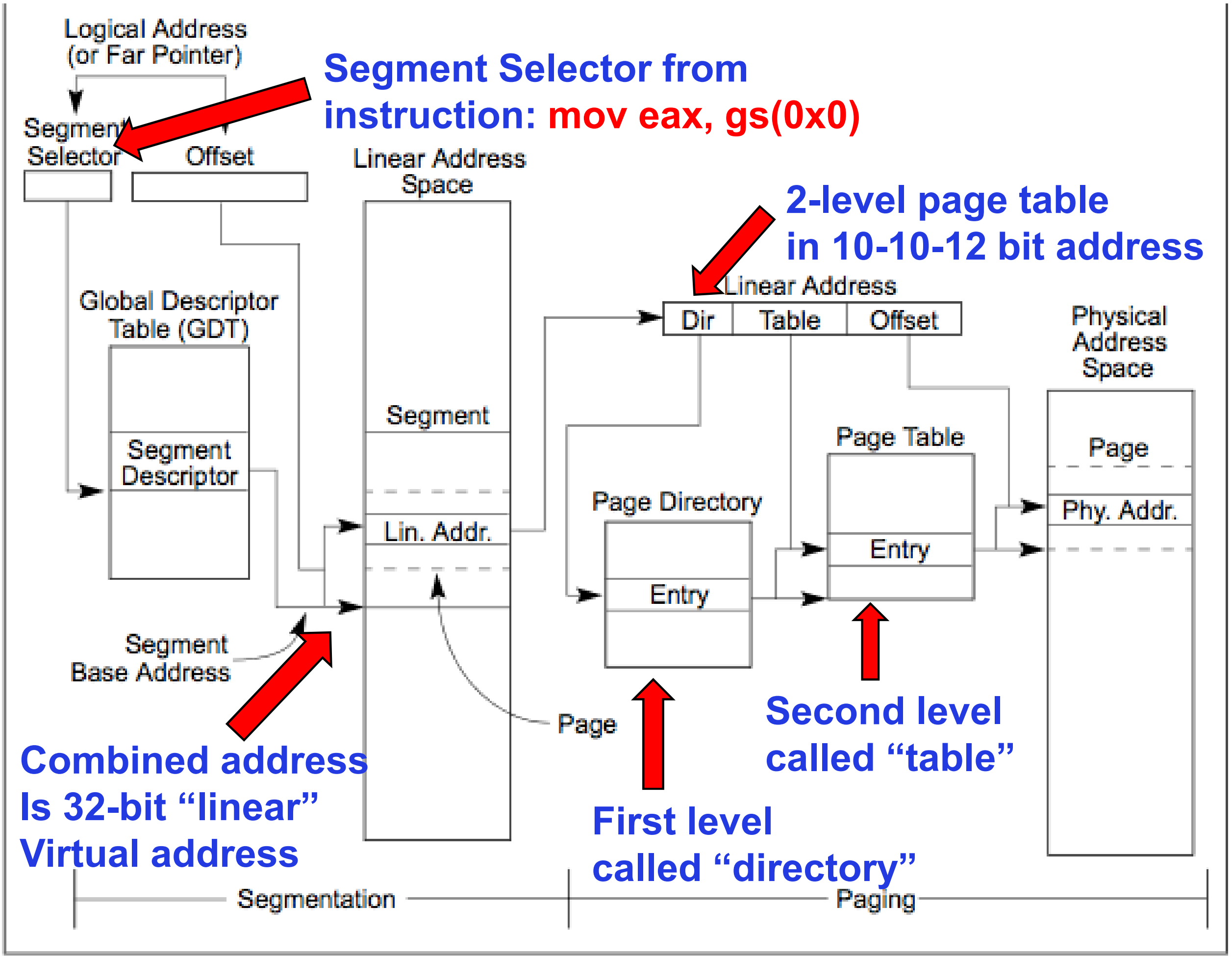
Multi-level Translation Analysis

- Pros:
  - Only need to allocate as many page table entries as we need for application
    - » In other wards, 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

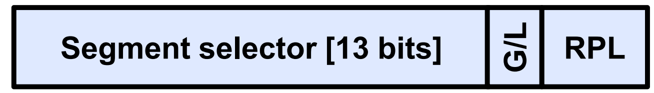
- Can a process modify its own translation tables? NO!
  - If it could, could get access to all of physical memory (no protection!)
- To Assist with Protection, Hardware provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program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



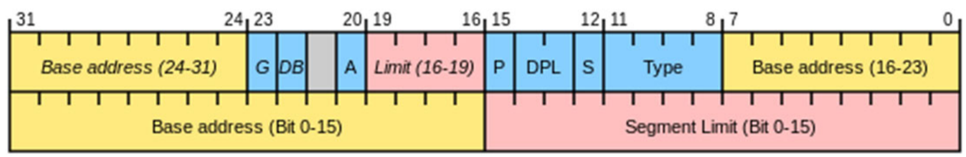
X86 Segment Descriptors (32-bit Protected Mode)

- Segments are implicit in the instruction (e.g. code segments) or part of the instruction
  – There are 6 registers: SS, CS, DS, ES, FS, GS



Segment Register

- 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 ⇒ Current Privilege Level)
- Two registers: GDTR/LDTR hold pointers to global/local descriptor tables in memory
  – Descriptor format (64 bits):

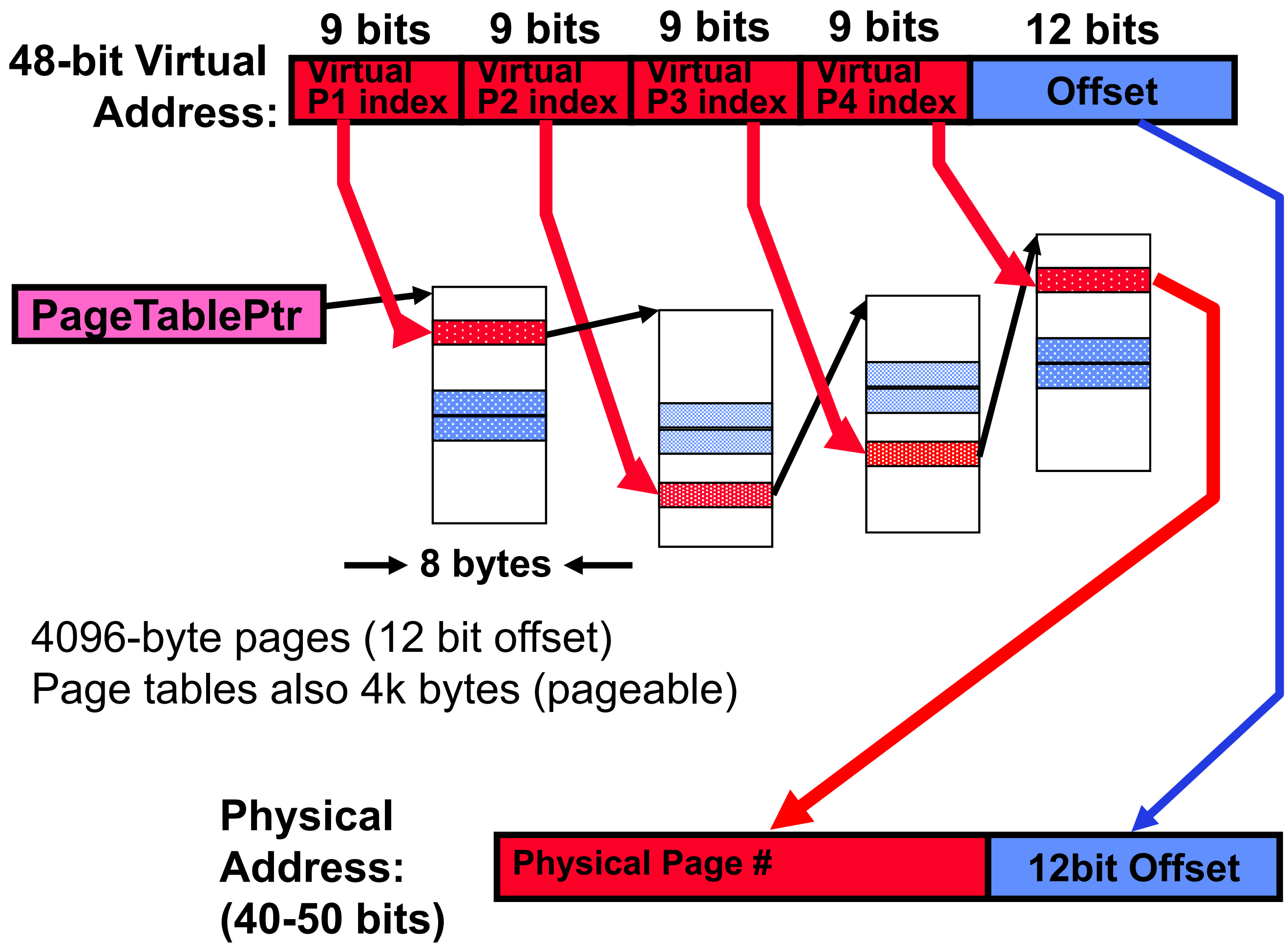


G: Granularity of segment [Limit Size] (0: bytes, 1: 4KiB unit)
DB: Default operand size (0: 16bit, 1: 32bit)
A: Programmer definable (no hardware meaning)
P: Segment present
DPL: Descriptor Privilege Level: Access requires Max(CPL,RPL)≤DPL
S: System Segment (0: System, 1: code or data)
Type: Code, Data, 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 ⇒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 TLS

X86_64: Four-level page table!



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

From x86_64 architecture specification

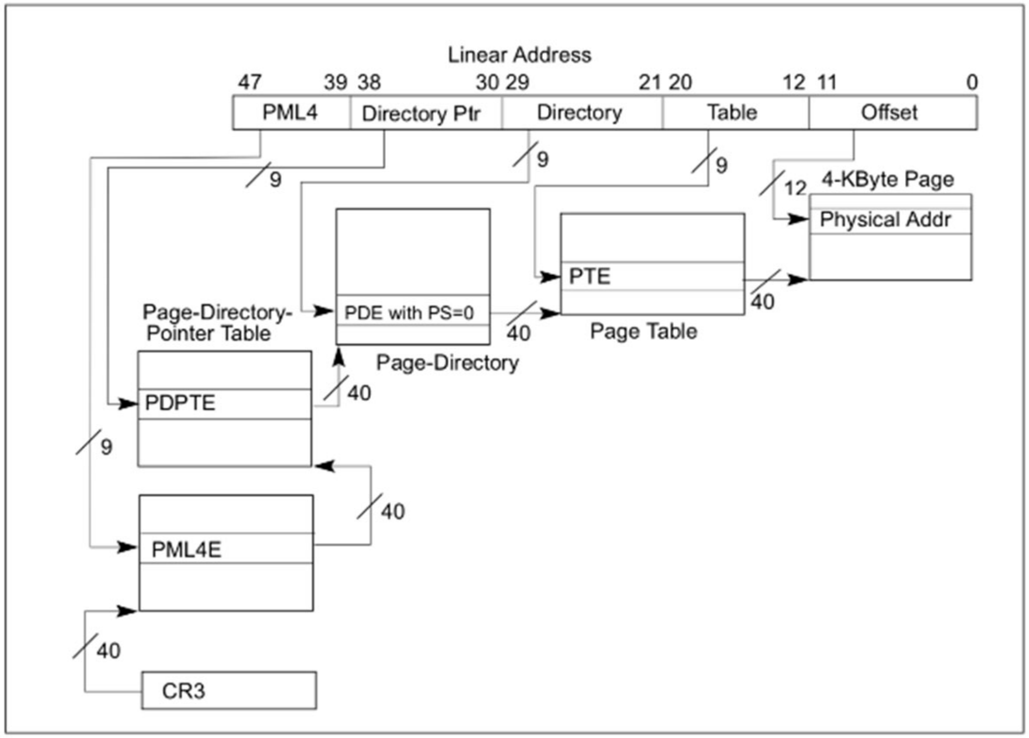


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

- 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

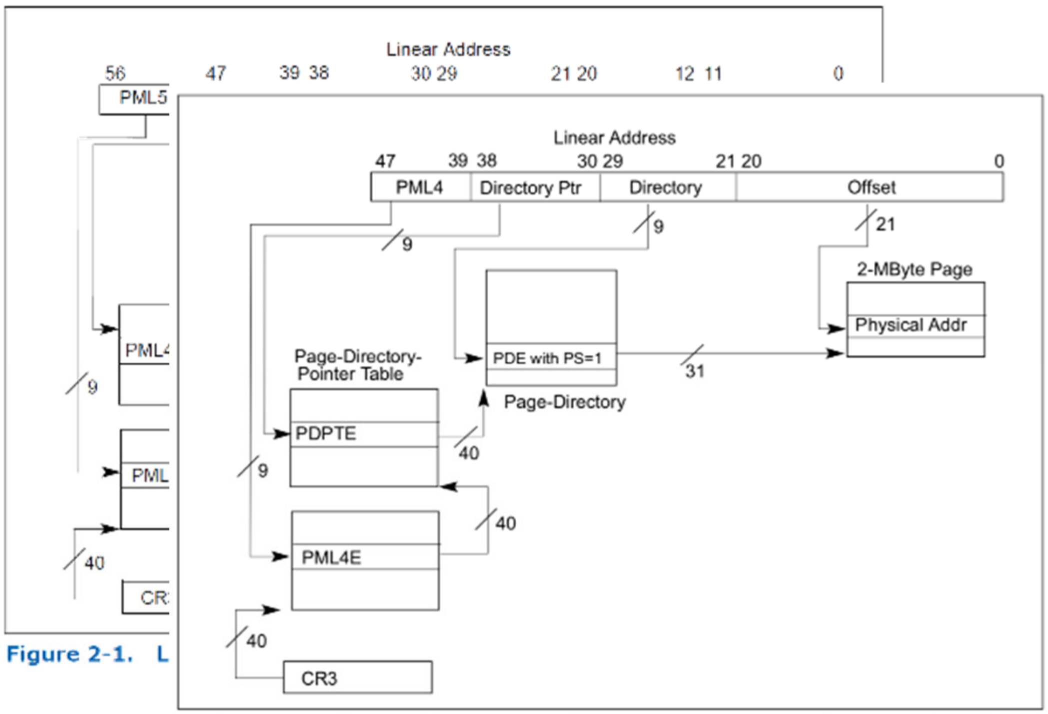


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

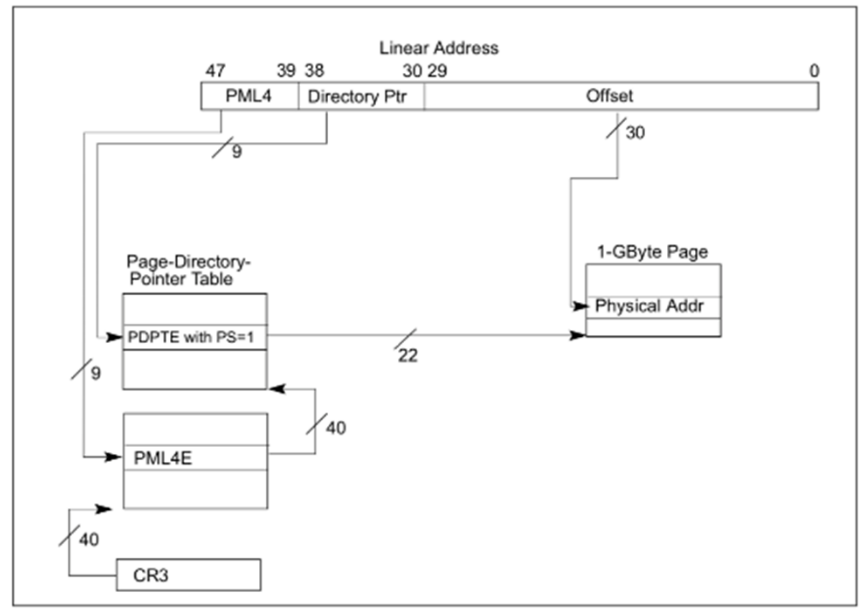


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

- Larger page sizes (2MB, 1GB) make sense since memory is now cheap
  - Great for kernel, large libraries, etc
  - Use limited primarily by internal fragmentation…

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

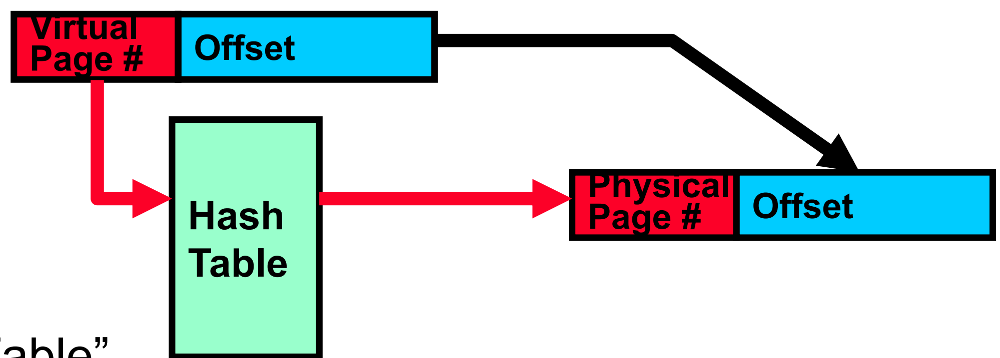
64bit Virtual Address:	7 bits	9 bits	9 bits	9 bits	9 bits	9 bits	12 bits
	Virtual P1 index	Virtual P2 index	Virtual P3 index	Virtual P4 index	Virtual P5 index	Virtual P6 index	Offset

No!

Too slow
Too many almost-empty tables

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

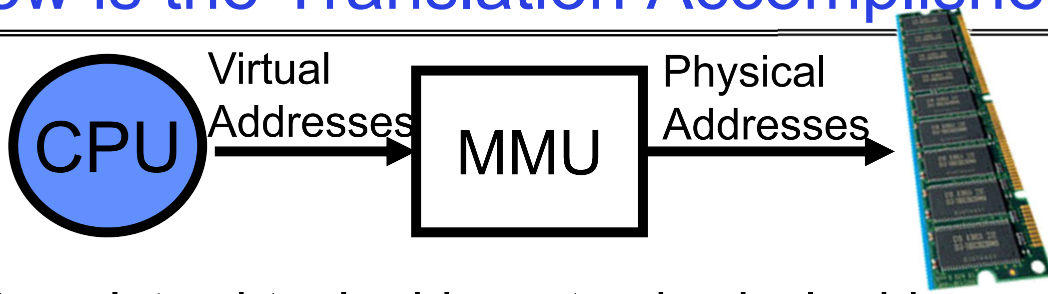


- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
    - » PowerPC, UltraSPARC, IA64
- Cons:
  - Complexity of managing hash chains: Often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

Address Translation Comparison

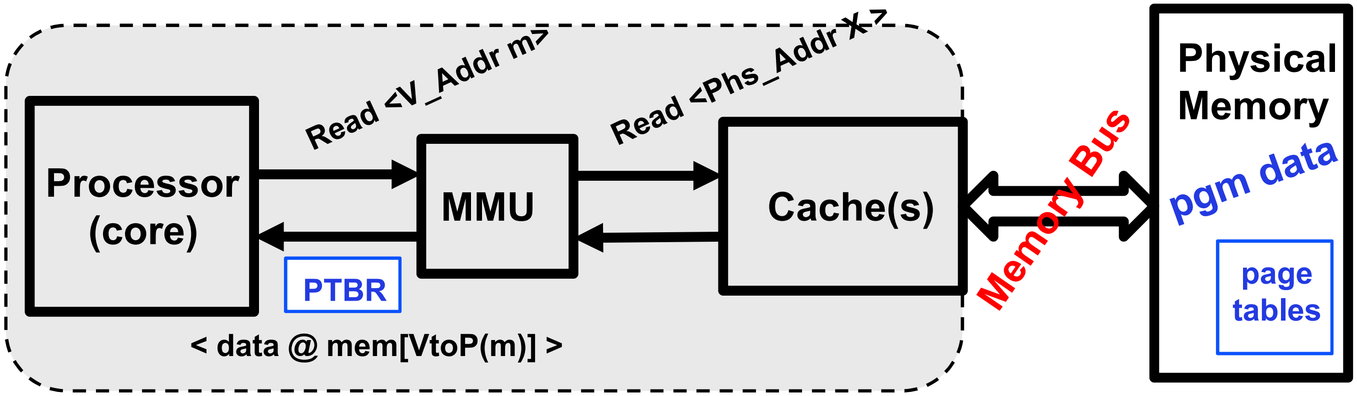
	Advantages	Disadvantages
Simple Segmentation	Fast context switching (segment map maintained by CPU)	External fragmentation
Paging (Single-Level)	No external fragmentation Fast and easy allocation	Large table size (~ virtual memory) Internal fragmentation
Paged Segmentation	Table size ~ # of pages in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

How is the Translation Accomplished?



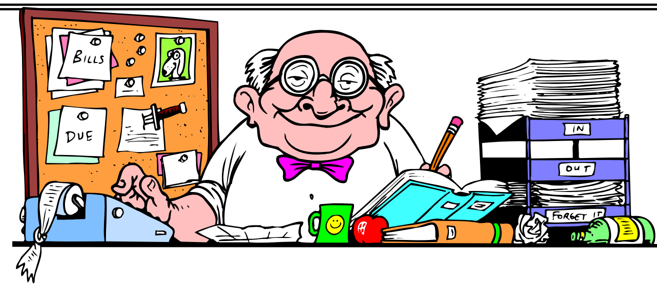
- The MMU must translate virtual address to physical address on:
  - Every instruction fetch
  - Every load
  - Every store
- What does the MMU need to do to translate an address?
  - 1-level Page Table
    - » Read PTE from memory, check valid, merge address
    - » Set "accessed" bit in PTE, Set "dirty 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

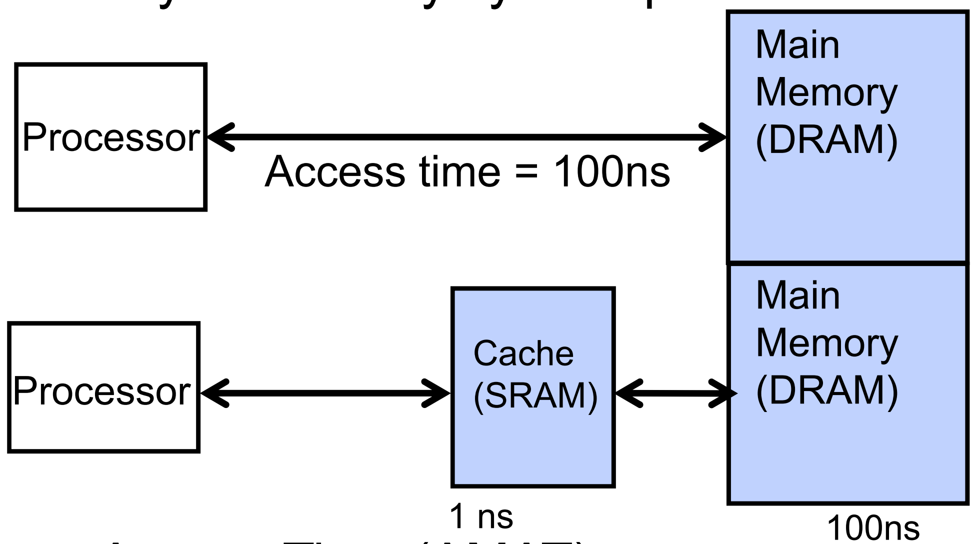
Recall: CS61c Caching Concept



- 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 x Hit Time) + (Miss Rate x Miss Time)

Recall: In Machine Structures (eg. 61C) …

- Caching is the key to memory system performance



Average Memory Access Time (AMAT)

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

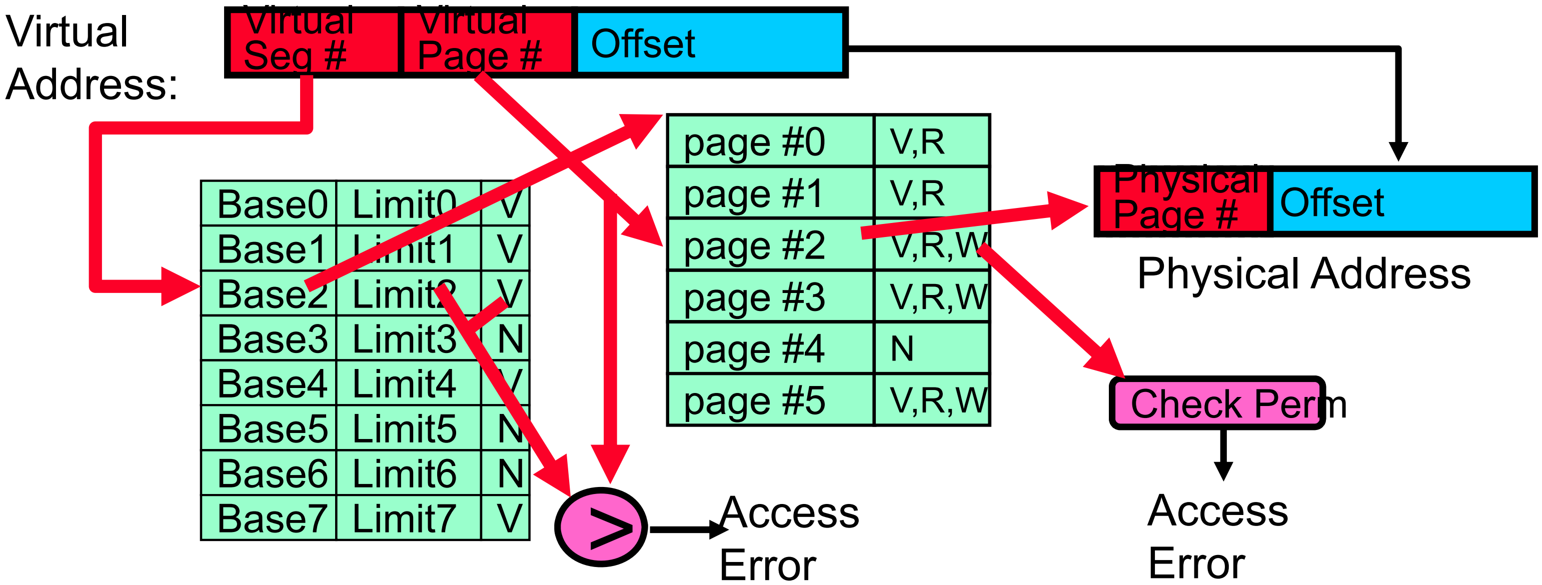
Where HitRate + MissRate = 1

HitRate = 90% => AMAT = (0.9 x 1) + (0.1 x 101)=11 ns

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

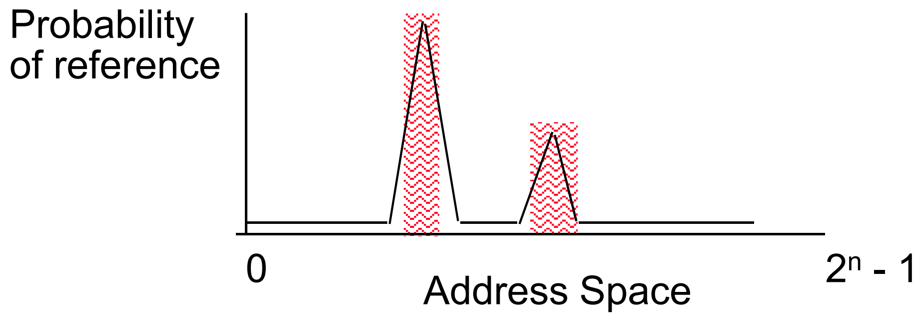
MissTime_{L1} includes $\text{HitTime}_{L1}+\text{MissPenalty}_{L1} \equiv \text{HitTime}_{L1} + \text{AMAT}_{L2}$

Another Major Reason to Deal with Caching

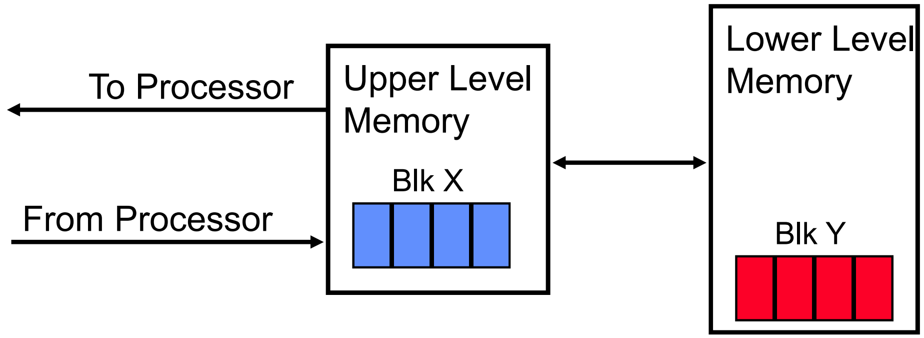


- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access?
- Solution? Cache translations!
  - Translation Cache: TLB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

Why Does Caching Help? Locality!

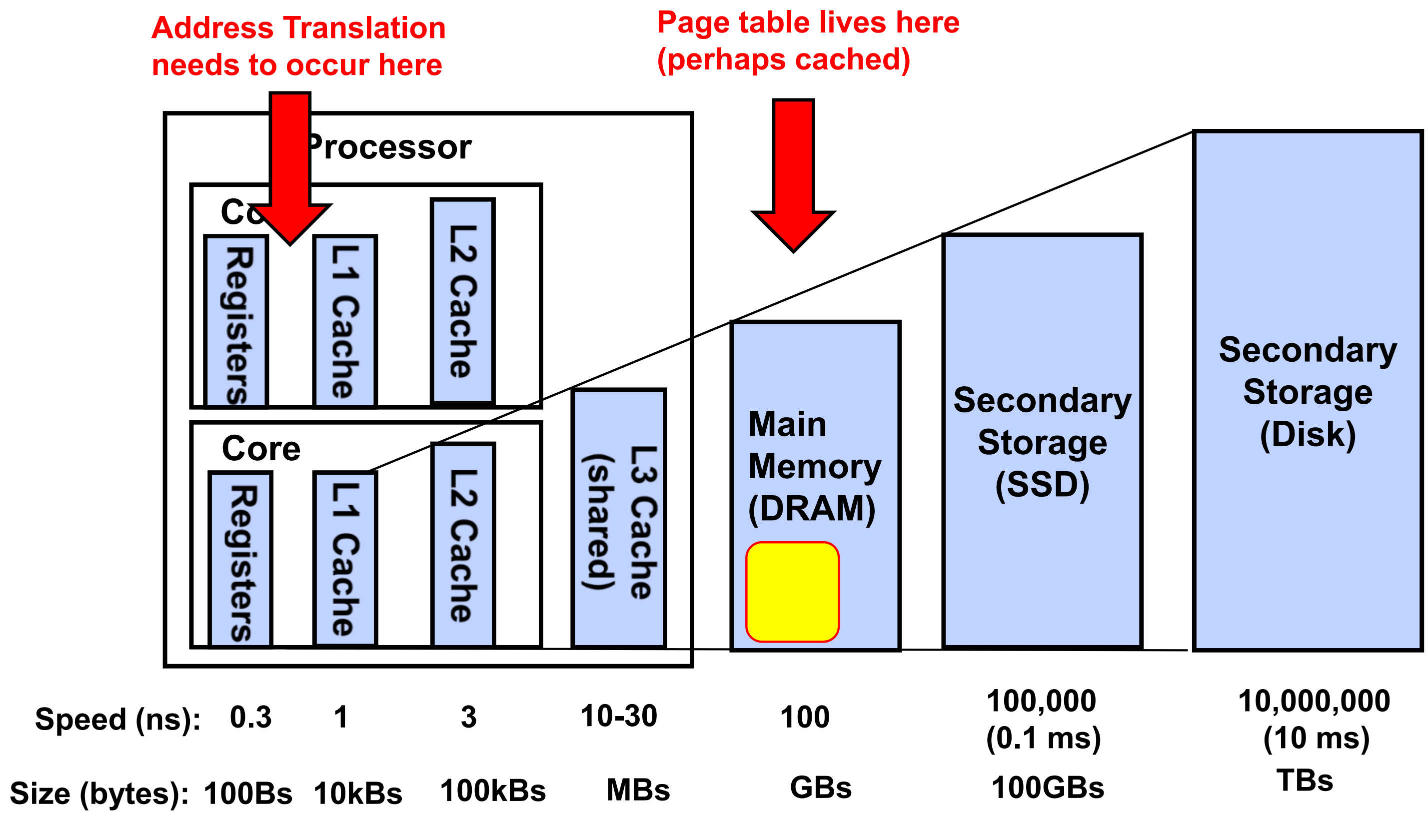


- Temporal Locality (Locality in Time):
  - Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - Move contiguous blocks to the upper levels



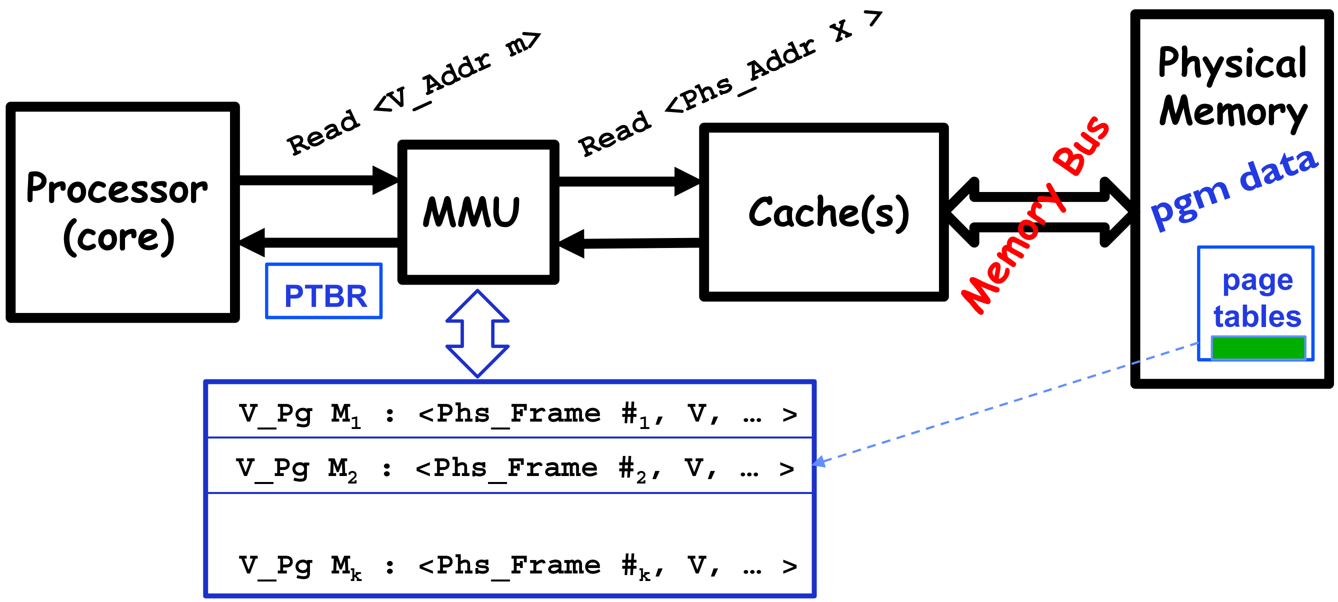
Recall: Memory Hierarchy

- 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?

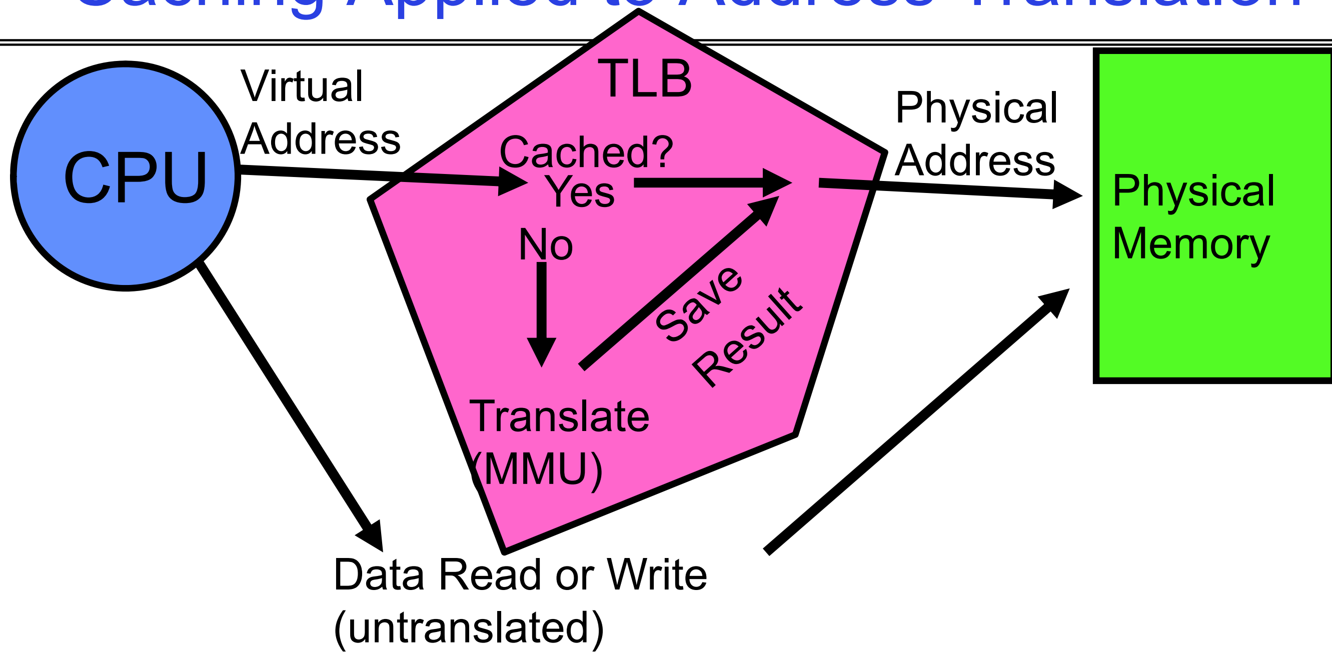
- Cache results of recent translations !
  – Different from a traditional cache
  – Cache Page Table Entries using Virtual Page # as the key



Translation Look-Aside Buffer

- Record recent Virtual Page # to Physical 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*
  – When you come up with a new concept, you get to name it!
  – 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

Conclusion

- 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
- 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