18-447
Computer Architecture
Lecture 20: Virtual Memory

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2015, 3/4/2015

Assignment and Exam Reminders

- Lab 4: Due March 6 (this Friday!)
  - Control flow and branch prediction
- Lab 5: Due March 22
  - Data cache
- HW 4: March 18
- Exam: March 20
- Advice: Finish the labs early
  - You have almost a month for Lab 5
- Advice: Manage your time well

Debugging, Testing and Piazza Etiquette

- You are supposed to debug your code
- You are supposed to develop your own test cases
- As in real life
- Piazza is not the place to ask debugging questions
  - And expect answers to them
- And, TAs are not your debuggers
  - Ask for help only if you have done due diligence

Late Days

- Bonus: 2 more late days for everyone

- Even if you run out of late days, please submit your lab
  - Showing that you are working and caring counts
  - Making it work counts

Agenda for the Rest of 447

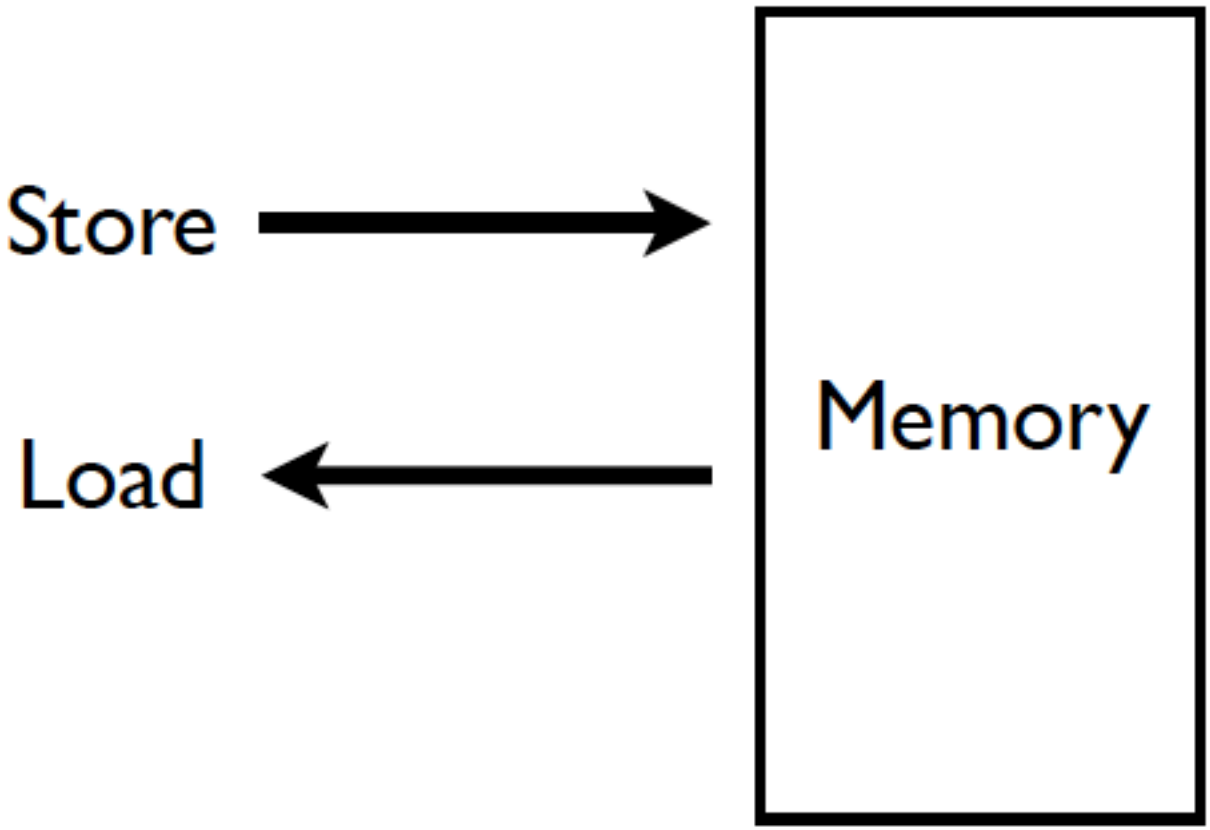
- The memory hierarchy
- Caches, caches, more caches
- Virtualizing the memory hierarchy: Virtual Memory
- Main memory: DRAM
- Main memory control, scheduling
- Memory latency tolerance techniques
- Non-volatile memory

- Multiprocessors
- Coherence and consistency
- Interconnection networks
- Multi-core issues

Readings

- Section 5.4 in P&H
- Optional: Section 8.8 in Hamacher et al.
- Your 213 textbook for brush-up

Memory (Programmer's View)



Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

Abstraction: Virtual vs. Physical Memory

- Programmer sees virtual memory
  - Can assume the memory is "infinite"
- Reality: Physical memory size is much smaller than what the programmer assumes
- The system (system software + hardware, cooperatively) maps virtual memory addresses are to physical memory
  - The system automatically manages the physical memory space transparently to the programmer

+ Programmer does not need to know the physical size of memory nor manage it → A small physical memory can appear as a huge one to the programmer → Life is easier for the programmer

-- More complex system software and architecture

A classic example of the programmer/(micro)architect tradeoff

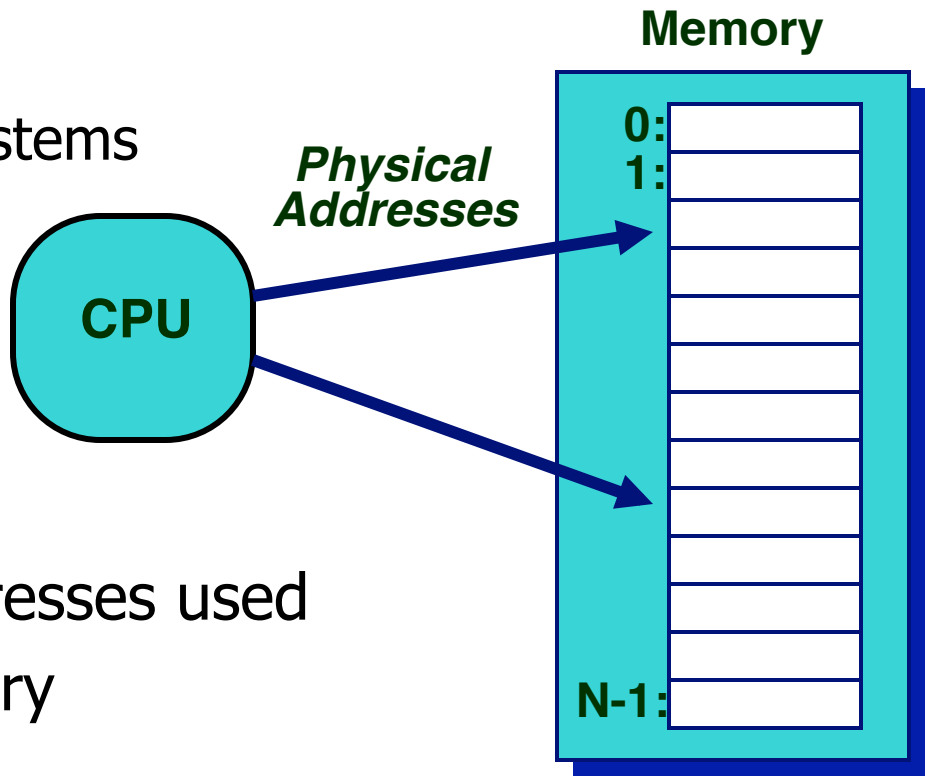
Benefits of Automatic Management of Memory

- Programmer does not deal with physical addresses
- Each process has its own mapping from virtual→physical addresses

- Enables
  - Code and data to be located anywhere in physical memory (relocation)
  - Isolation/separation of code and data of different processes in physical processes (protection and isolation)
  - Code and data sharing between multiple processes (sharing)

A System with Physical Memory Only

- Examples:
  - most Cray machines
  - early PCs
  - nearly all embedded systems



CPU's load or store addresses used directly to access memory

The Problem

- Physical memory is of limited size (cost)
  - What if you need more?
  - Should the programmer be concerned about the size of code/ data blocks fitting physical memory?
  - Should the programmer manage data movement from disk to physical memory?
  - Should the programmer ensure two processes do not use the same physical memory?

- Also, ISA can have an address space greater than the physical memory size
  - E.g., a 64-bit address space with byte addressability
  - What if you do not have enough physical memory?

Difficulties of Direct Physical Addressing

- Programmer needs to manage physical memory space
  - Inconvenient & hard
  - Harder when you have multiple processes
- Difficult to support code and data relocation
- Difficult to support multiple processes
  - Protection and isolation between multiple processes
  - Sharing of physical memory space
- Difficult to support data/code sharing across processes

Virtual Memory

- Idea: Give the programmer the illusion of a large address space while having a small physical memory
  - So that the programmer does not worry about managing physical memory

- Programmer can assume he/she has "infinite" amount of physical memory

- Hardware and software cooperatively and automatically manage the physical memory space to provide the illusion
  - Illusion is maintained for each independent process

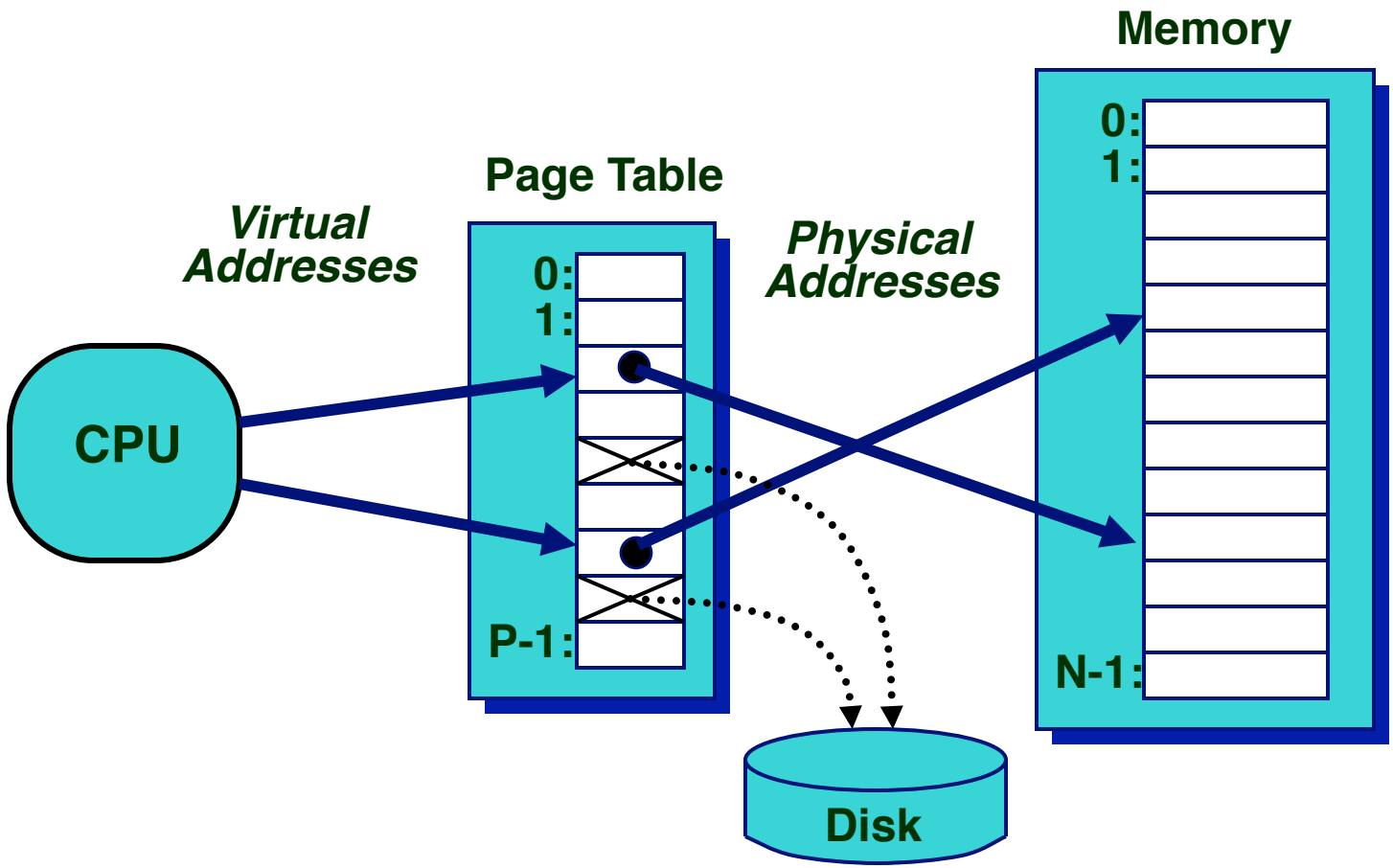
Basic Mechanism

- Indirection (in addressing)

- Address generated by each instruction in a program is a "virtual address"
  - i.e., it is not the physical address used to address main memory
  - called "linear address" in x86

- An "address translation" mechanism maps this address to a "physical address"
  - called "real address" in x86
  - Address translation mechanism can be implemented in hardware and software together

A System with Virtual Memory (Page based)



- Address Translation: The hardware converts virtual addresses into physical addresses via an OS-managed lookup table (page table)

Virtual Pages, Physical Frames

- Virtual address space divided into pages
- Physical address space divided into frames

- A virtual page is mapped to
  - A physical frame, if the page is in physical memory
  - A location in disk, otherwise

- If an accessed virtual page is not in memory, but on disk
  - Virtual memory system brings the page into a physical frame and adjusts the mapping → this is called demand paging

- Page table is the table that stores the mapping of virtual pages to physical frames

Physical Memory as a Cache

- In other words…

- Physical memory is a cache for pages stored on disk
  - In fact, it is a fully associative cache in modern systems (a virtual page can be mapped to any physical frame)

- Similar caching issues exist as we have covered earlier:
  - Placement: where and how to place/find a page in cache?
  - Replacement: what page to remove to make room in cache?
  - Granularity of management: large, small, uniform pages?
  - Write policy: what do we do about writes? Write back?

Supporting Virtual Memory

- Virtual memory requires both HW+SW support
  - Page Table is in memory
  - Can be cached in special hardware structures called Translation Lookaside Buffers (TLBs)

- The hardware component is called the MMU (memory management unit)
  - Includes Page Table Base Register(s), TLBs, page walkers

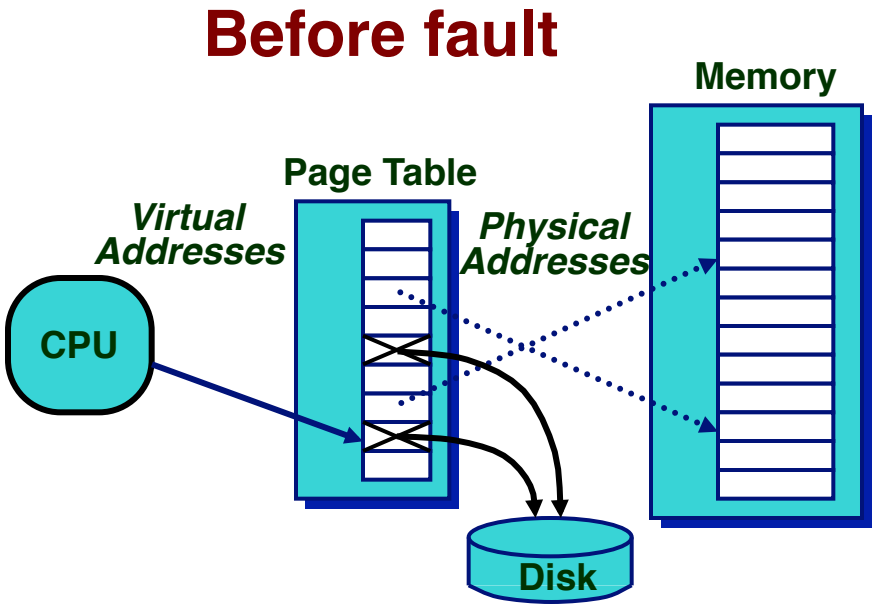
- It is the job of the software to leverage the MMU to
  - Populate page tables, decide what to replace in physical memory
  - Change the Page Table Register on context switch (to use the running thread's page table)
  - Handle page faults and ensure correct mapping

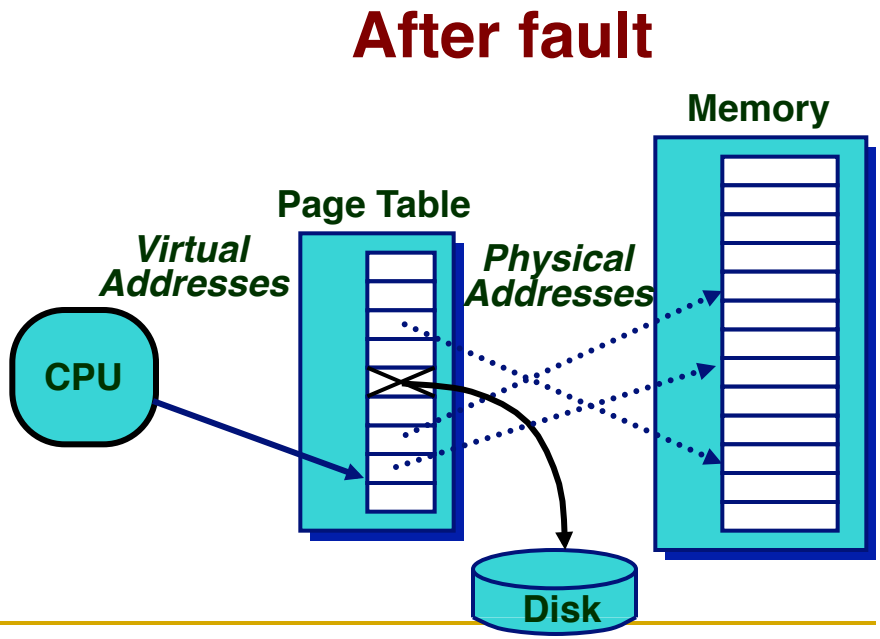
Some System Software Jobs for VM

- Keeping track of which physical frames are free
- Allocating free physical frames to virtual pages
- Page replacement policy
  - When no physical frame is free, what should be swapped out?
- Sharing pages between processes
- Copy-on-write optimization
- Page-flip optimization

Page Fault ("A Miss in Physical Memory")

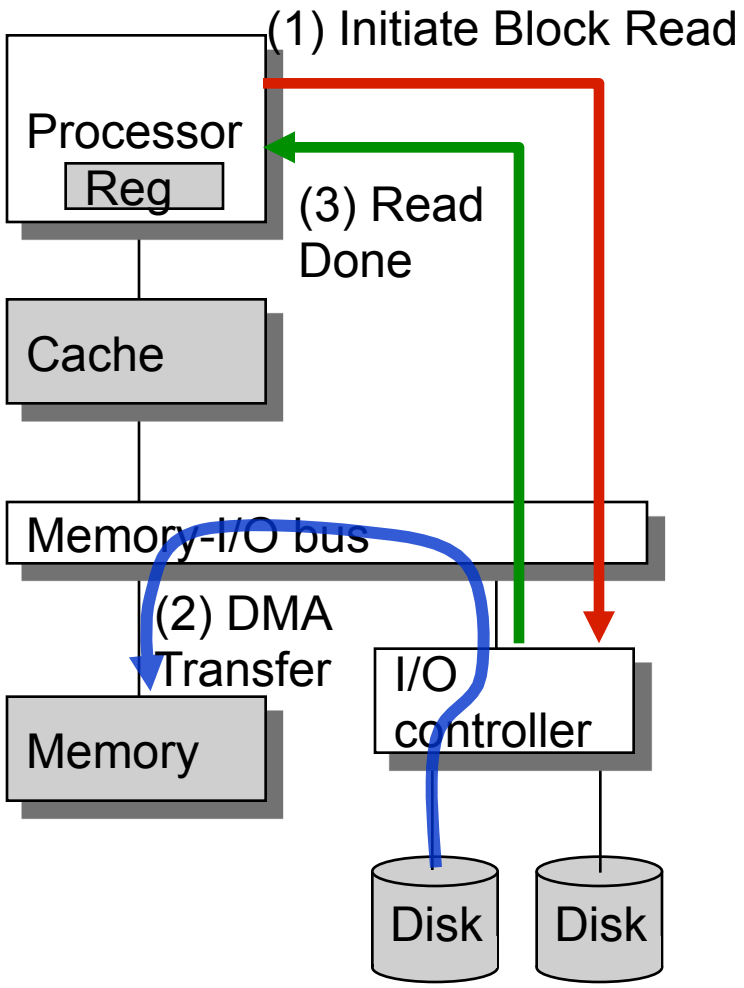
- If a page is not in physical memory but disk
  - Page table entry indicates virtual page not in memory
  - Access to such a page triggers a page fault exception
  - OS trap handler invoked to move data from disk into memory
    - Other processes can continue executing
    - OS has full control over placement





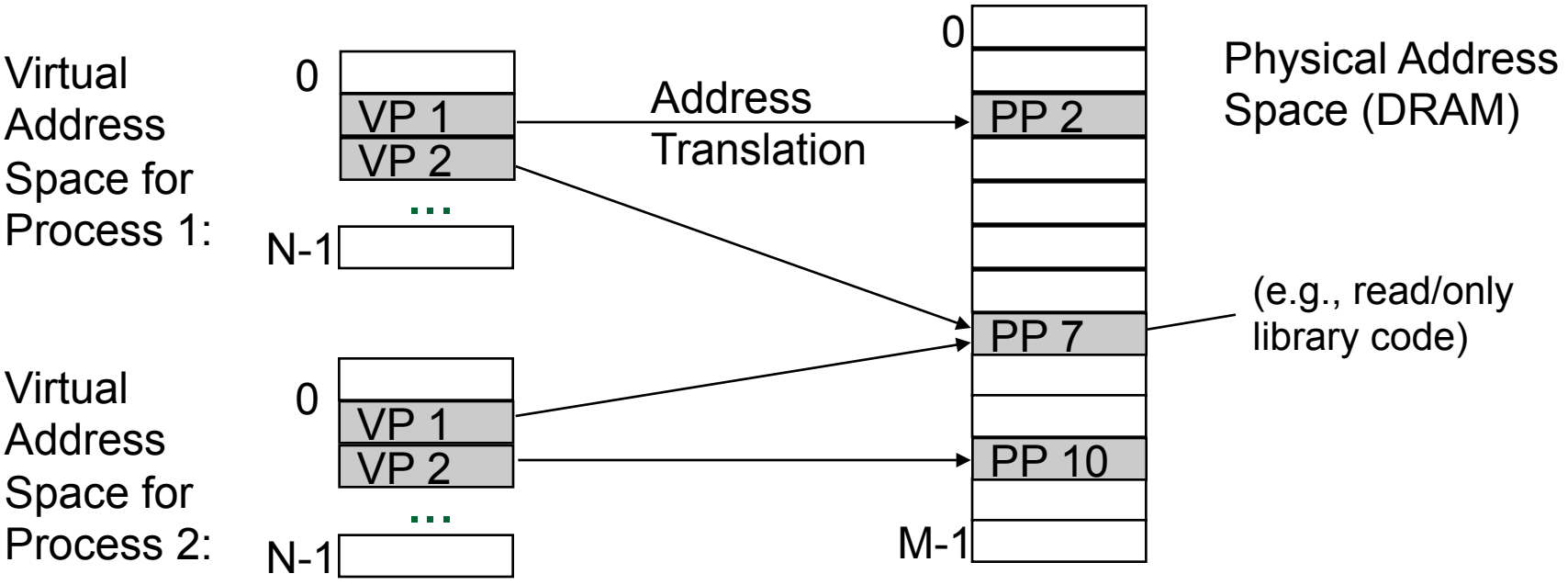
Servicing a Page Fault

- (1) Processor signals controller
  - Read block of length P starting at disk address X and store starting at memory address Y
- (2) Read occurs
  - Direct Memory Access (DMA)
  - Under control of I/O controller
- (3) Controller signals completion
  - Interrupt processor
  - OS resumes suspended process



Page Table is Per Process

- Each process has its own virtual address space
  - Full address space for each program
  - Simplifies memory allocation, sharing, linking and loading.



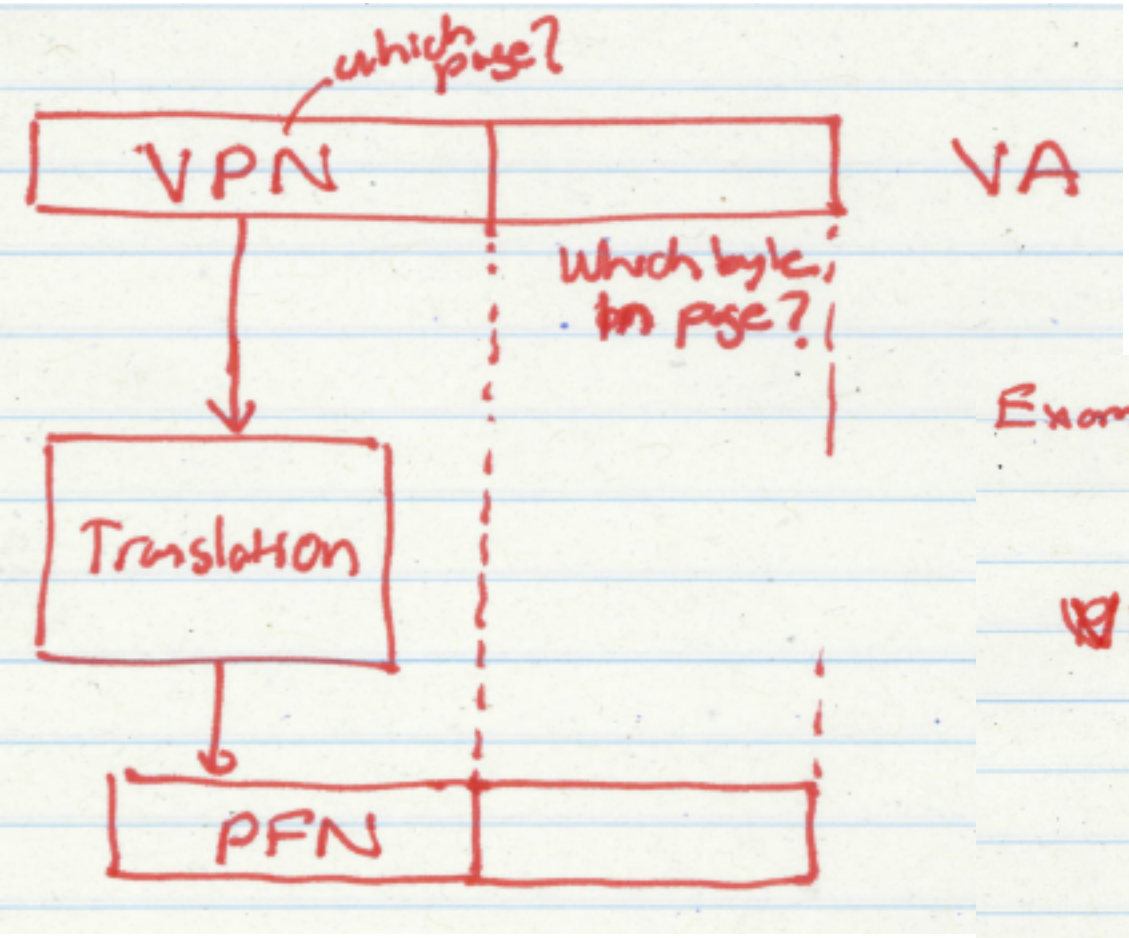
Address Translation

- How to obtain the physical address from a virtual address?

- Page size specified by the ISA
  - VAX: 512 bytes
  - Today: 4KB, 8KB, 2GB, … (small and large pages mixed together)
  - Trade-offs? (remember cache lectures)

- Page Table contains an entry for each virtual page
  - Called Page Table Entry (PTE)
  - What is in a PTE?

Address Translation (II)

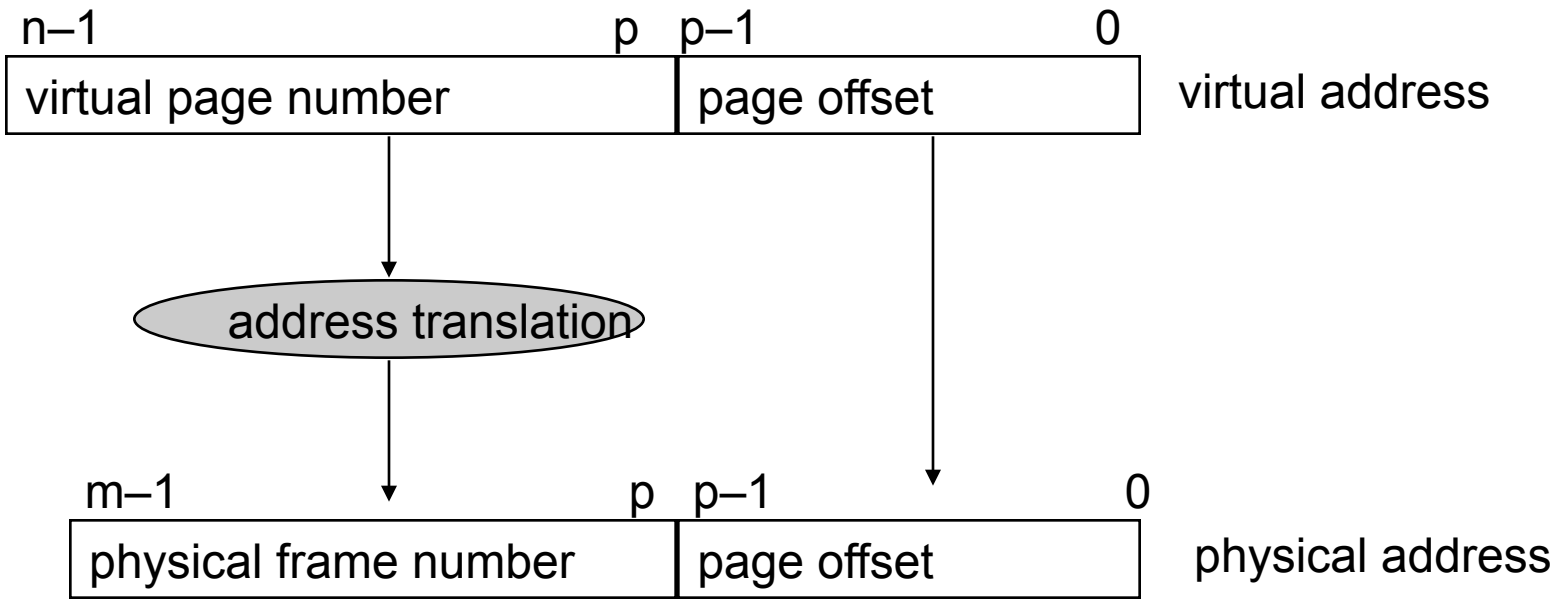


Example: 8K page size
32-bit virtual address space

VPN → 19 bits
→ 2^{19} virtual pages
→ 2^{19} PTEs in page table (for each process)

Address Translation (III)

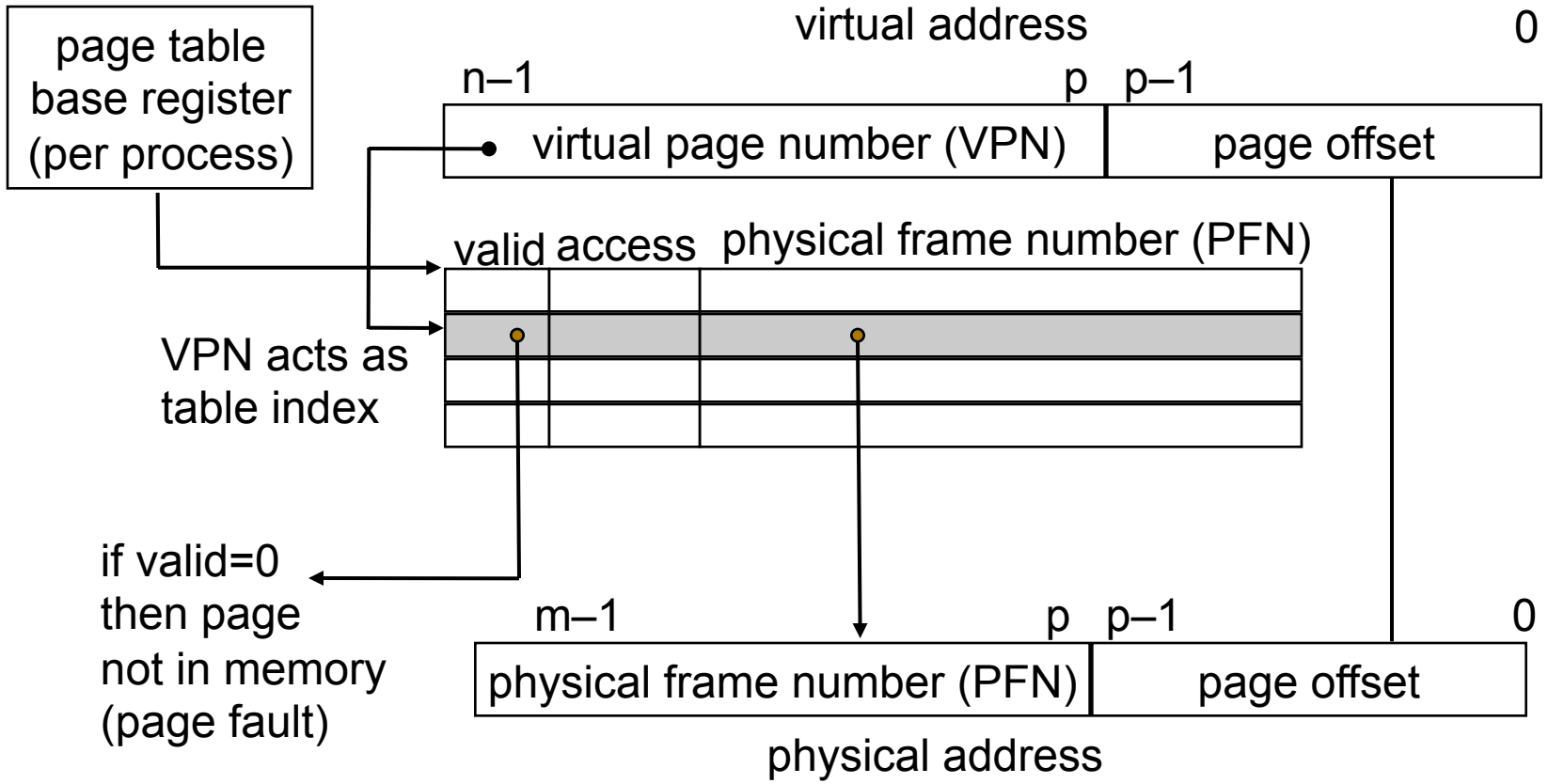
- Parameters
  - P = 2^p = page size (bytes).
  - N = 2^n = Virtual-address limit
  - M = 2^m = Physical-address limit



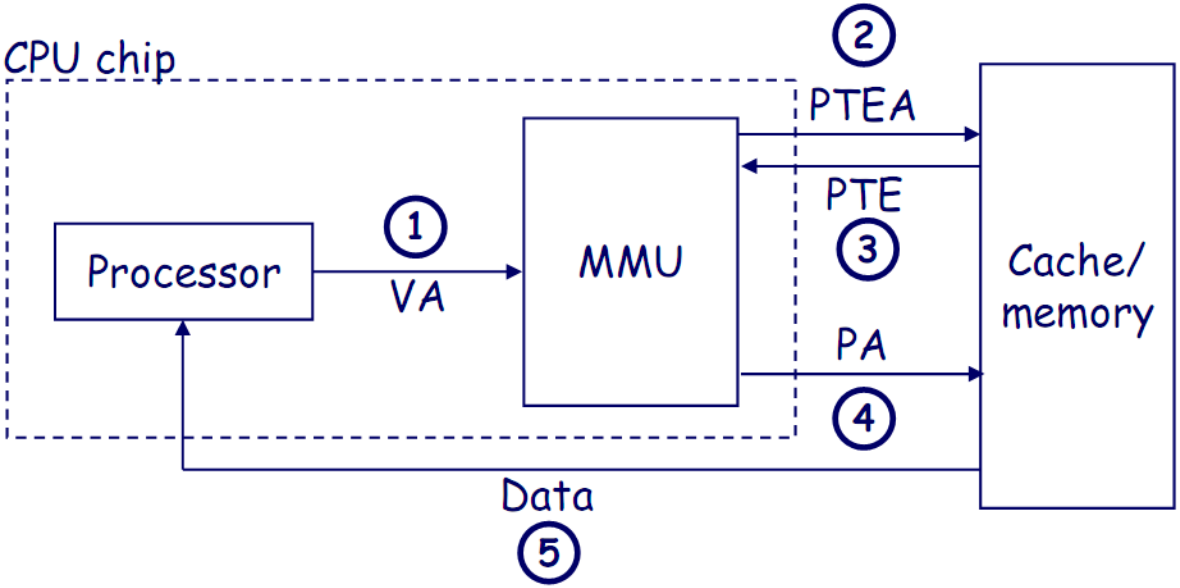
Page offset bits don't change as a result of translation

Address Translation (IV)

- Separate (set of) page table(s) per process
- VPN forms index into page table (points to a page table entry)
- Page Table Entry (PTE) provides information about page



Address Translation: Page Hit



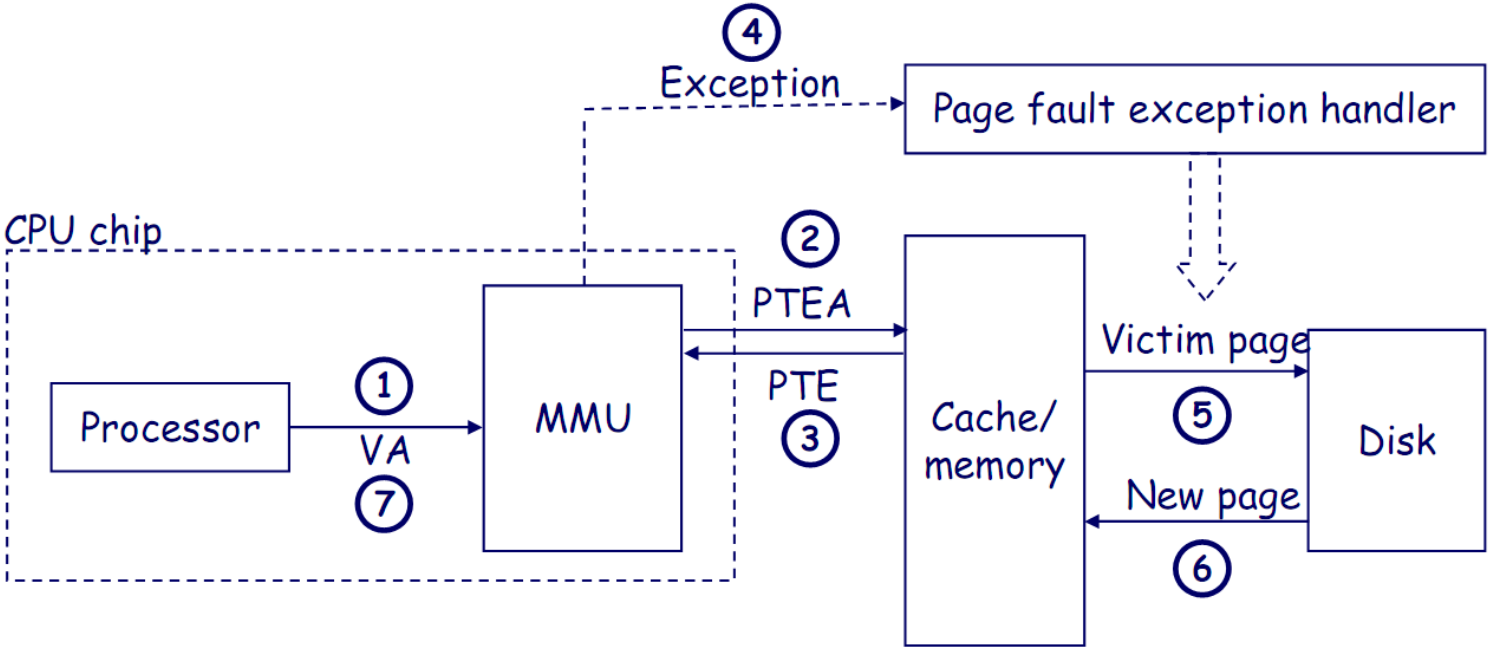
1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to L1 cache

5) L1 cache sends data word to processor

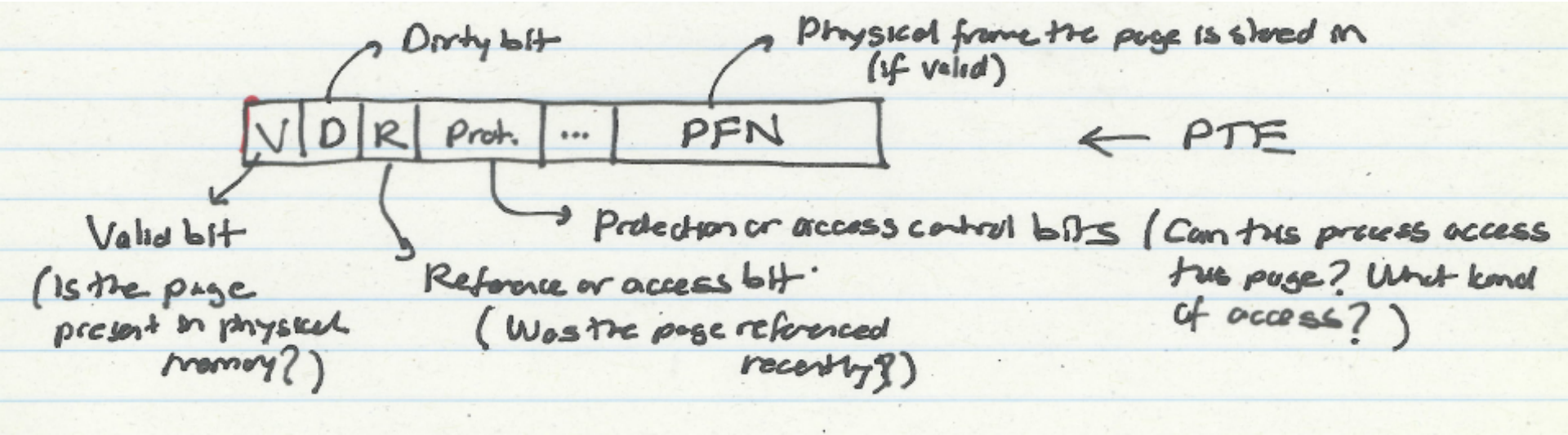
Address Translation: Page Fault



1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim, and if dirty pages it out to disk
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction.

What Is in a Page Table Entry (PTE)?

- Page table is the "tag store" for the physical memory data store
  - A mapping table between virtual memory and physical memory
- PTE is the "tag store entry" for a virtual page in memory
  - Need a valid bit → to indicate validity/presence in physical memory
  - Need tag bits (PFN) → to support translation
  - Need bits to support replacement
  - Need a dirty bit to support "write back caching"
  - Need protection bits to enable access control and protection



Remember: Cache versus Page Replacement

- Physical memory (DRAM) is a cache for disk
  - Usually managed by system software via the virtual memory subsystem

- Page replacement is similar to cache replacement
- Page table is the "tag store" for physical memory data store

- What is the difference?
  - Required speed of access to cache vs. physical memory
  - Number of blocks in a cache vs. physical memory
  - "Tolerable" amount of time to find a replacement candidate (disk versus memory access latency)
  - Role of hardware versus software

Page Replacement Algorithms

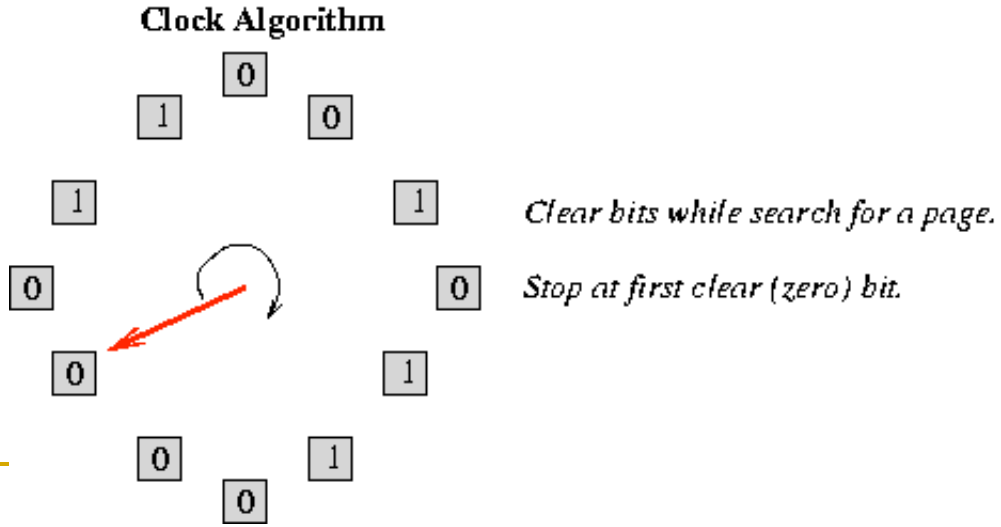
- If physical memory is full (i.e., list of free physical pages is empty), which physical frame to replace on a page fault?

- Is True LRU feasible?
  - 4GB memory, 4KB pages, how many possibilities of ordering?

- Modern systems use approximations of LRU
  - E.g., the CLOCK algorithm
- And, more sophisticated algorithms to take onto account "frequency" of use
  - E.g., the ARC algorithm
  - Megiddo and Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," FAST 2003.

CLOCK Page Replacement Algorithm

- Keep a circular list of physical frames in memory
- Keep a pointer (hand) to the last-examined frame in the list
- When a page is accessed, set the R bit in the PTE
- When a frame needs to be replaced, replace the first frame that has the reference (R) bit not set, traversing the circular list starting from the pointer (hand) clockwise
  - During traversal, clear the R bits of examined frames
  - Set the hand pointer to the next frame in the list



Aside: Page Size Trade Offs

- What is the granularity of management of physical memory?
- Large vs. small pages
- Tradeoffs have analogies to large vs. small cache blocks

- Many different tradeoffs with advantages and disadvantages
  - Size of the Page Table (tag store)
  - Reach of the Translation Lookaside Buffer (we will see this later)
  - Transfer size from disk to memory (waste of bandwidth?)
  - Waste of space within a page (internal fragmentation)
  - Waste of space within the entire physical memory (external fragmentation)
  - Granularity of access protection
  - …

Access Protection/Control via Virtual Memory

Page-Level Access Control (Protection)

- Not every process is allowed to access every page
  - E.g., may need supervisor level privilege to access system pages

- Idea: Store access control information on a page basis in the process's page table

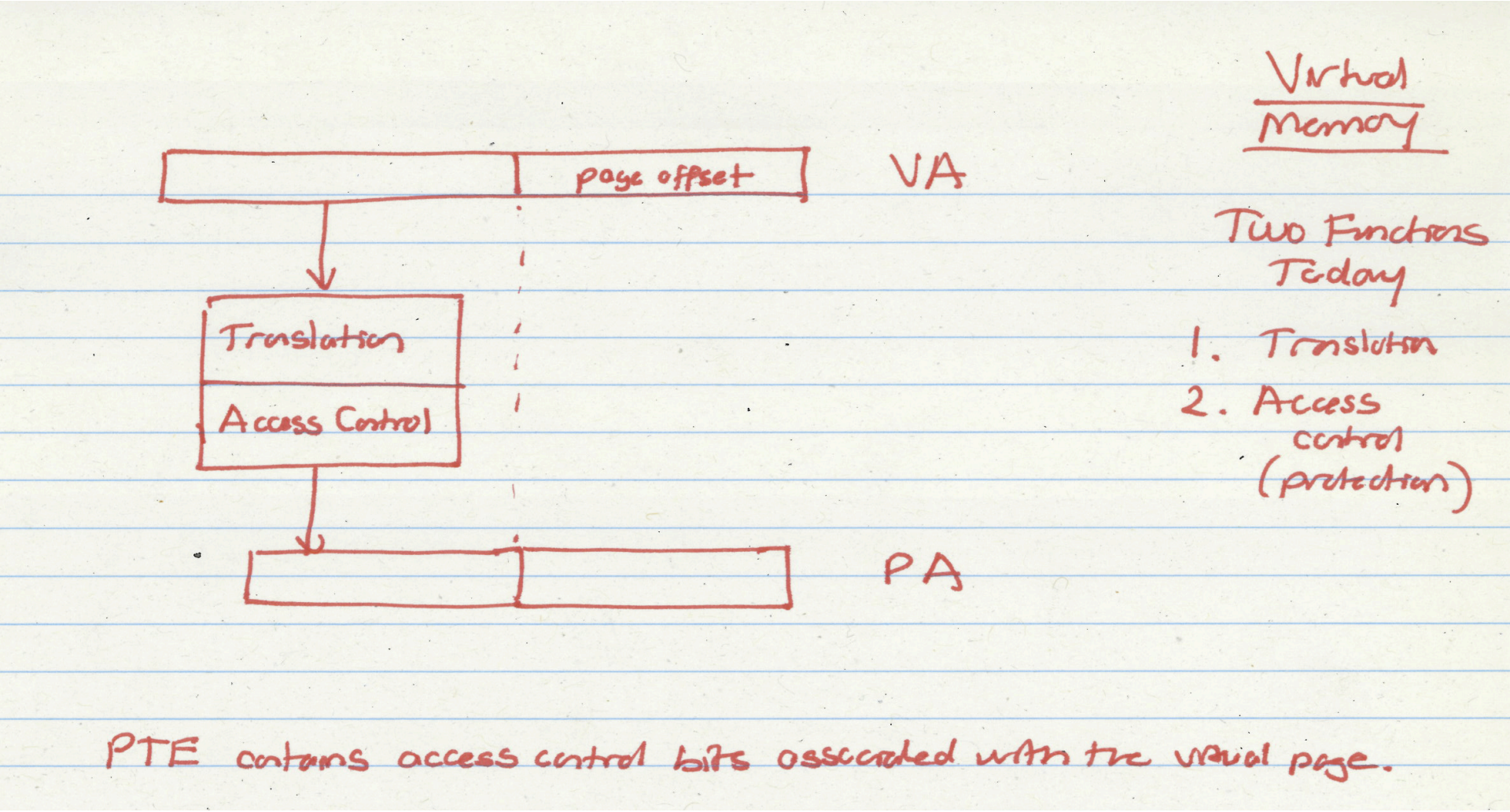
- Enforce access control at the same time as translation

→ Virtual memory system serves two functions today

Address translation (for illusion of large physical memory)

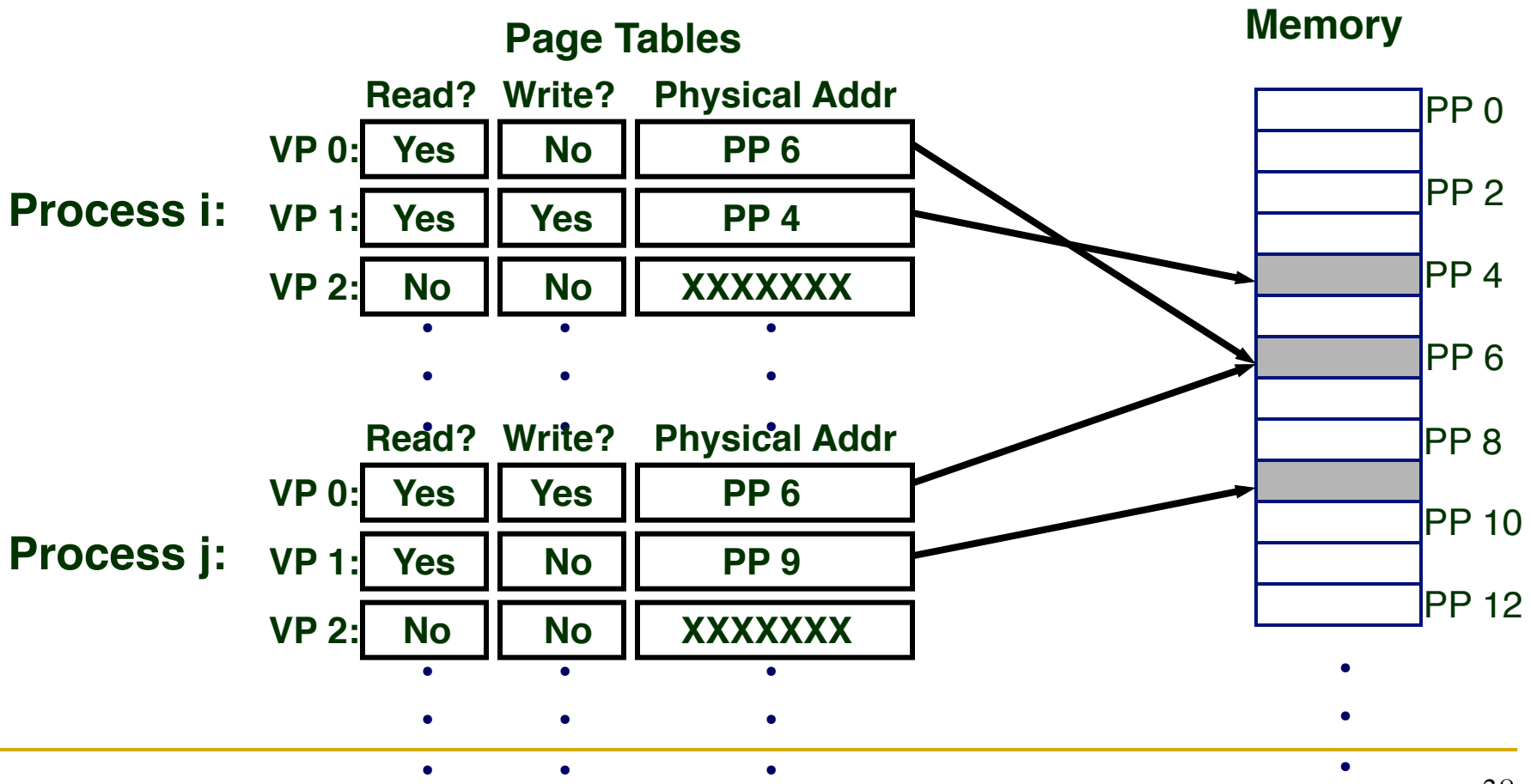
Access control (protection)

Two Functions of Virtual Memory

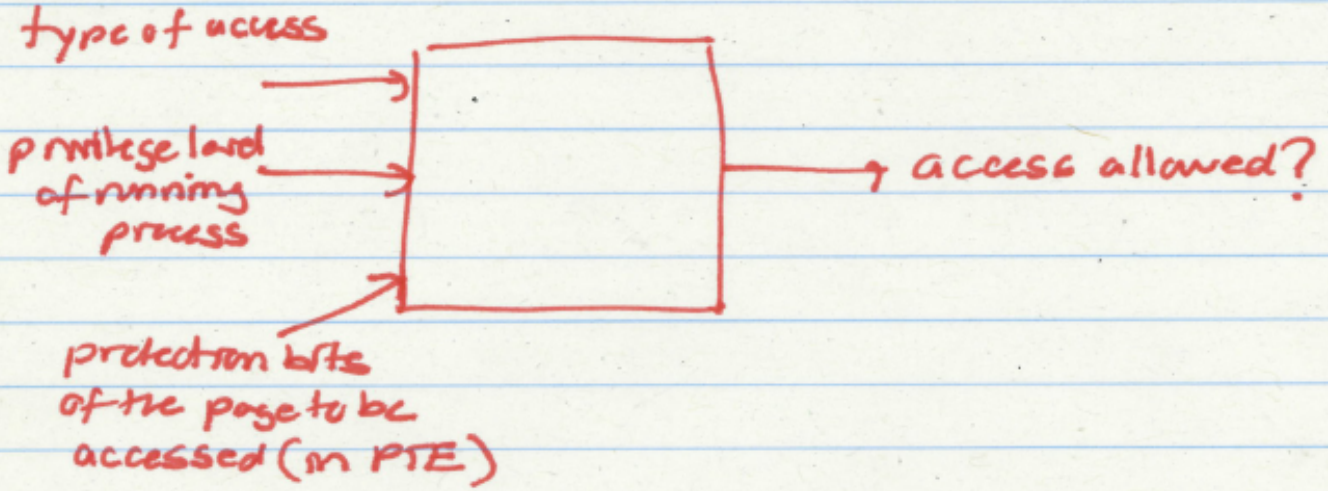


VM as a Tool for Memory Access Protection

- Extend Page Table Entries (PTEs) with permission bits
- Check bits on each access and during a page fault
  - If violated, generate exception (Access Protection exception)



Access Control Logic



type of access : R, W, E, ~~none~~
ordered: ~~none~~, E, R, W

privilege level: specified by ISA
VAX: Kernel (K), Executive (E)
Supervisor (S), User (U)

protection bits: Specify ~~whether this page can be accessed~~
what type of access can be made to this page
& at what privilege level

Privilege Levels in x86

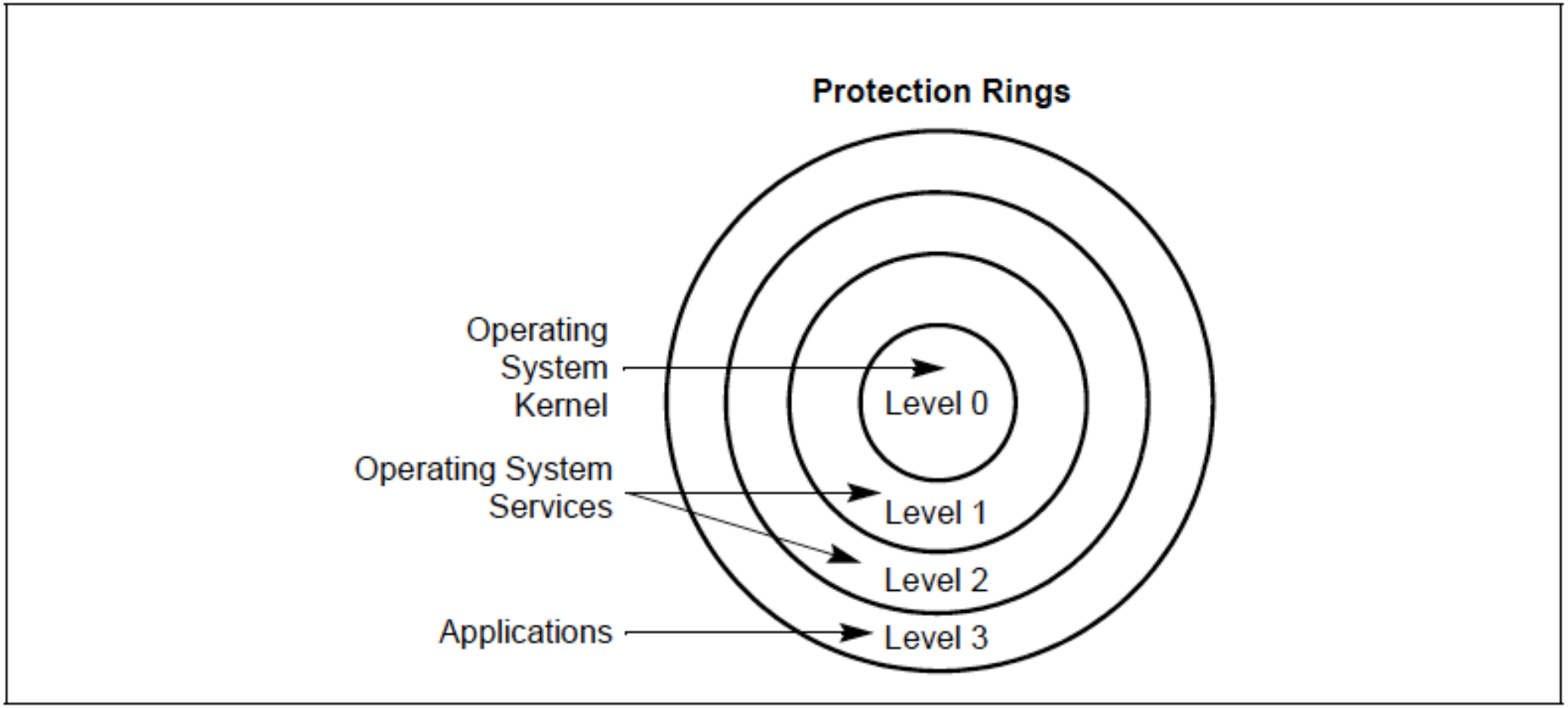


Figure 5-3. Protection Rings

Page Level Protection in x86

Table 5-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

Some Issues in Virtual Memory

Three Major Issues

- How large is the page table and how do we store and access it?
- How can we speed up translation & access control check?
- When do we do the translation in relation to cache access?
- There are many other issues we will not cover in detail
  - What happens on a context switch?
  - How can you handle multiple page sizes?
  - …

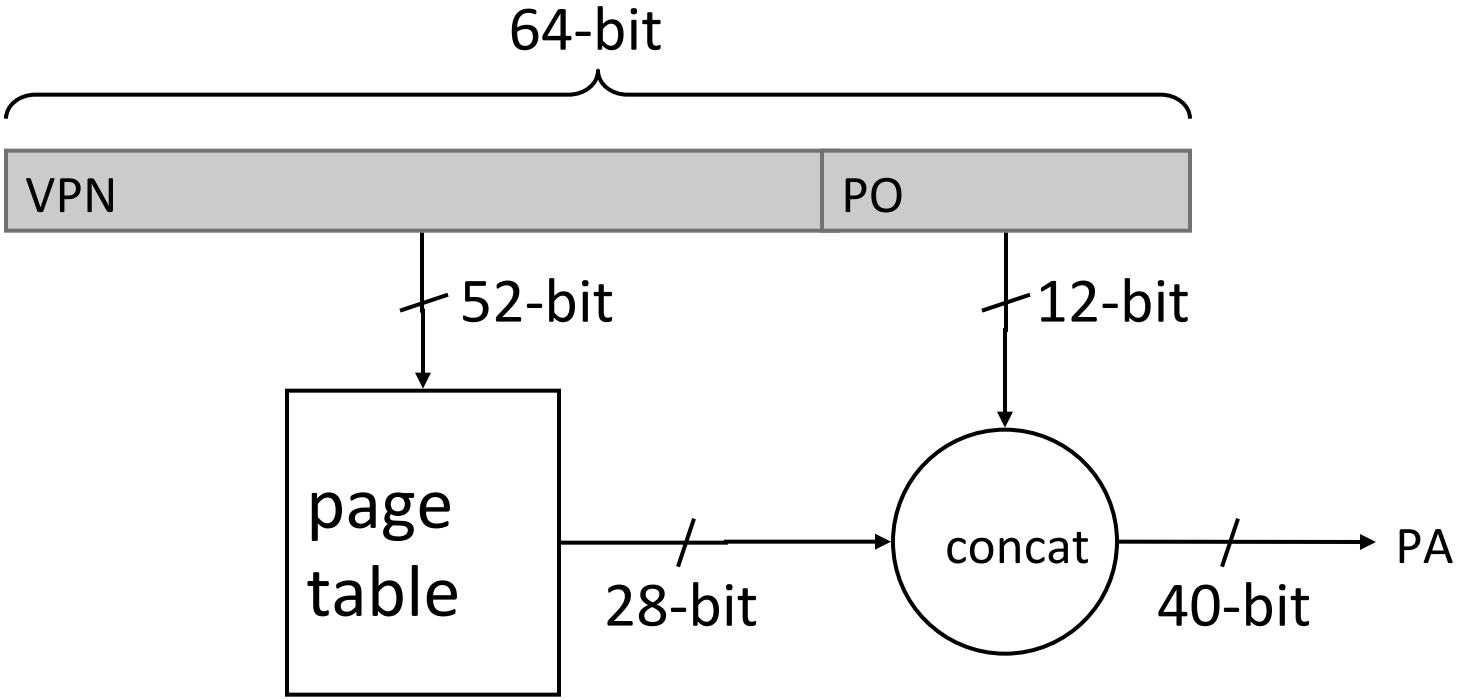
Virtual Memory Issue I

- How large is the page table?

- Where do we store it?
  - In hardware?
  - In physical memory? (Where is the PTBR?)
  - In virtual memory? (Where is the PTBR?)

- How can we store it efficiently without requiring physical memory that can store all page tables?
  - Idea: multi-level page tables
  - Only the first-level page table has to be in physical memory
  - Remaining levels are in virtual memory (but get cached in physical memory when accessed)

Issue: Page Table Size



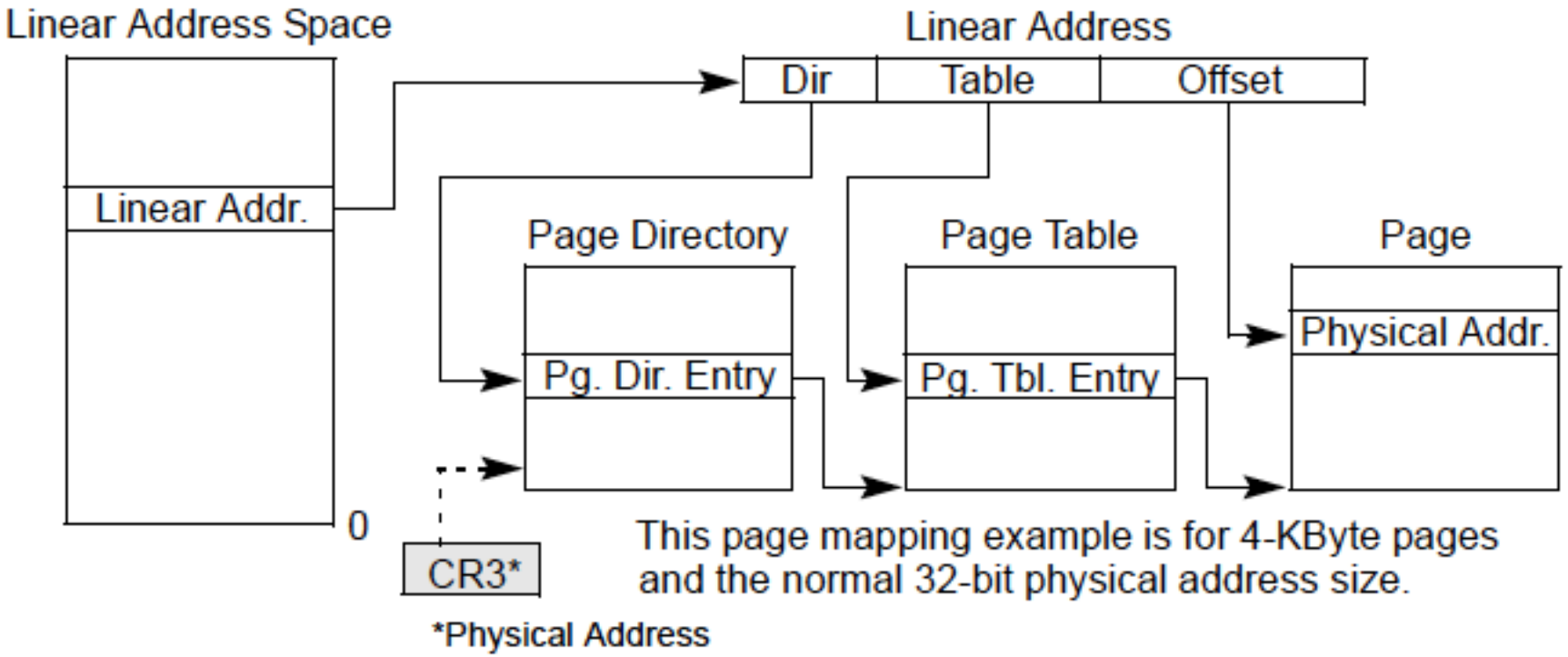
- Suppose 64-bit VA and 40-bit PA, how large is the page table?
  2^{52} entries x ~4 bytes ≈ 16×10^{15} Bytes

  and that is for just one process!

  and the process many not be using the entire VM space!

Solution: Multi-Level Page Tables

Example from x86 architecture



Page Table Access

- How do we access the Page Table?

- Page Table Base Register (CR3 in x86)
- Page Table Limit Register

- If VPN is out of the bounds (exceeds PTLR) then the process did not allocate the virtual page → access control exception

- Page Table Base Register is part of a process's context
  - Just like PC, status registers, general purpose registers
  - Needs to be loaded when the process is context-switched in

More on x86 Page Tables (I): Small Pages

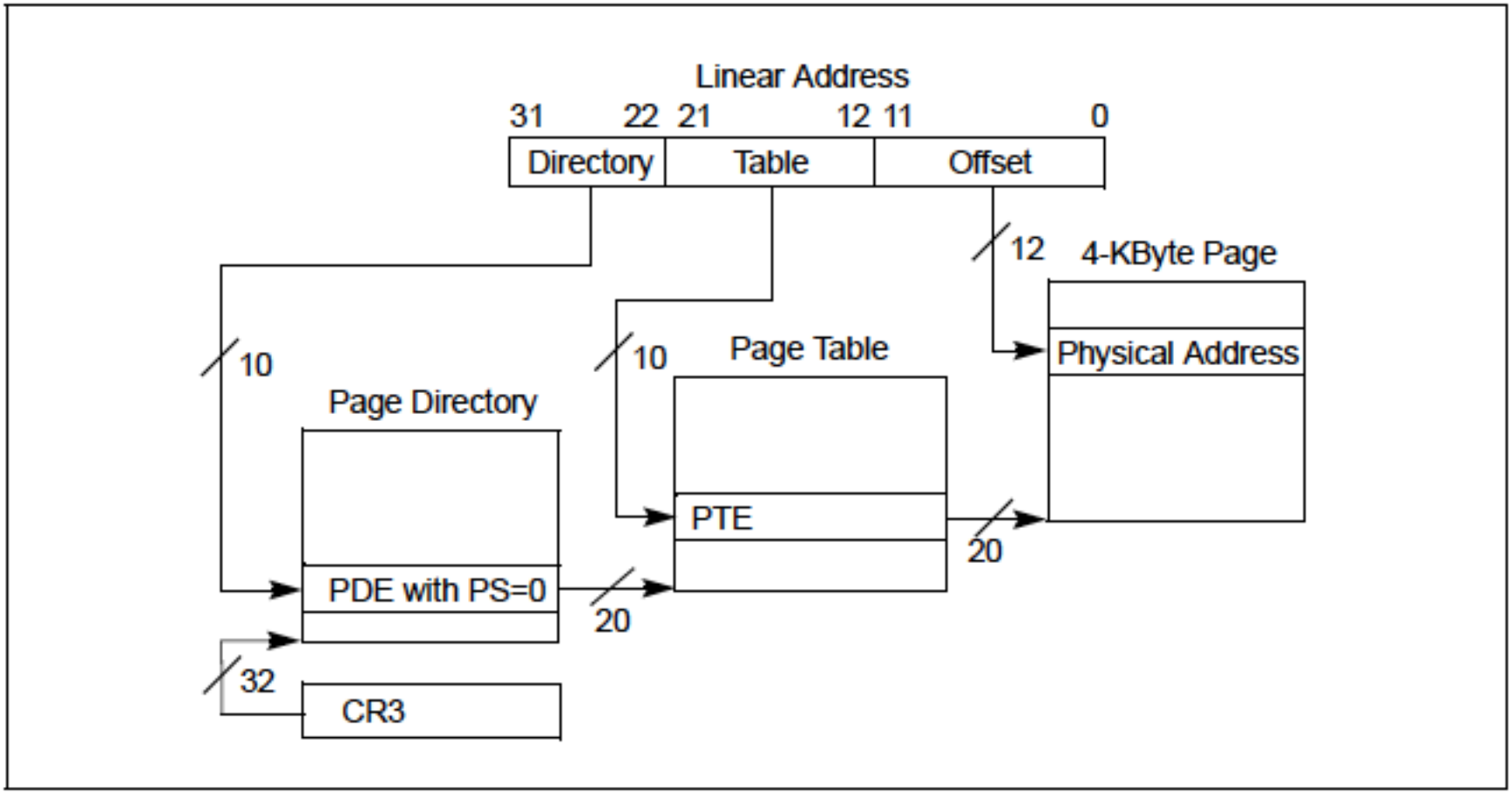


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

More on x86 Page Tables (II): Large Pages

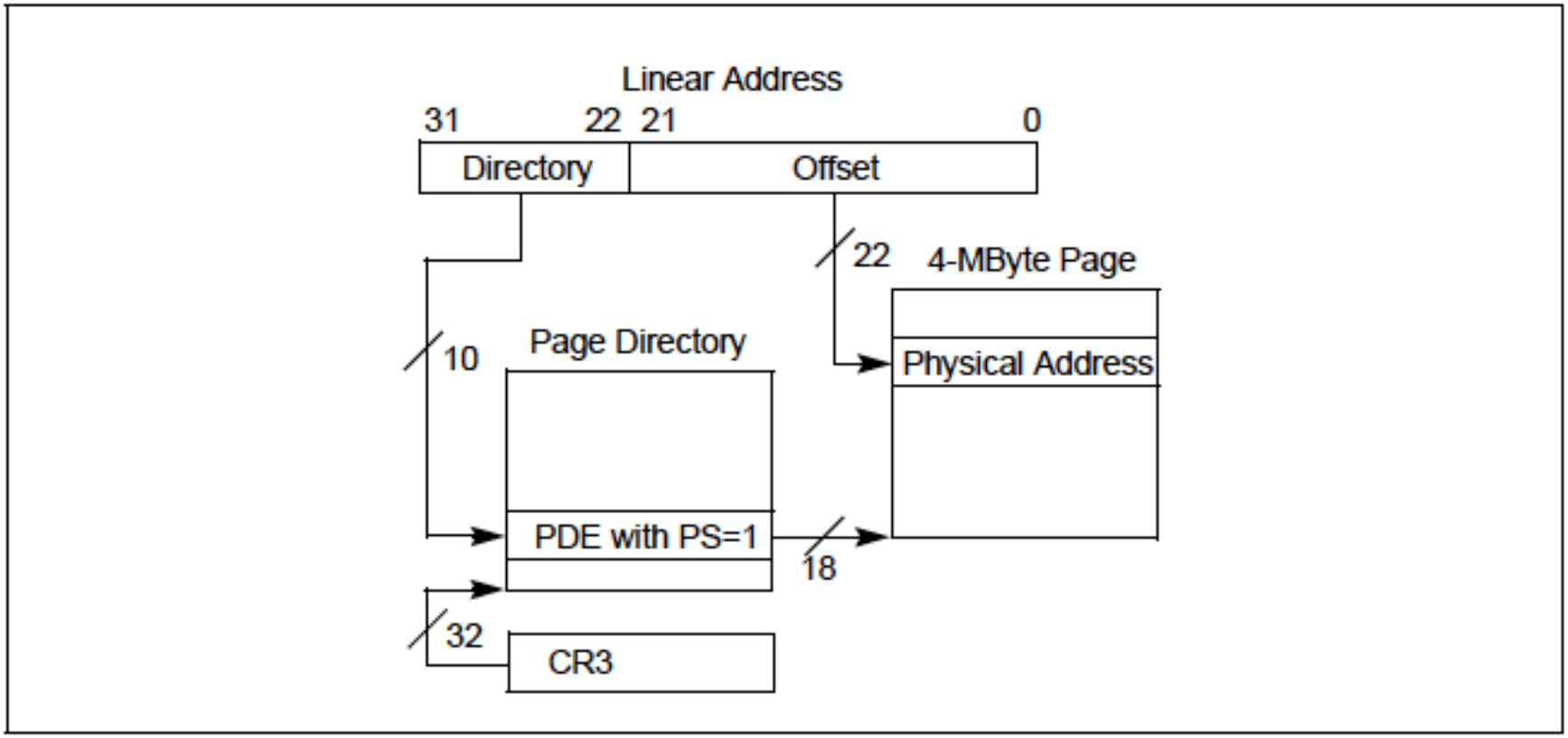


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

x86 Page Table Entries

Figure 4-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are "not present"; bit 0 (P) and bit 7 (PS) are highlighted because they determine how such an entry is used.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame										Reserved (must be 0)						Bits 39:32 of address[2]			PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

x86 PTE (4KB page)

Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

x86 Page Directory Entry (PDE)

Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)

Four-level Paging in x86

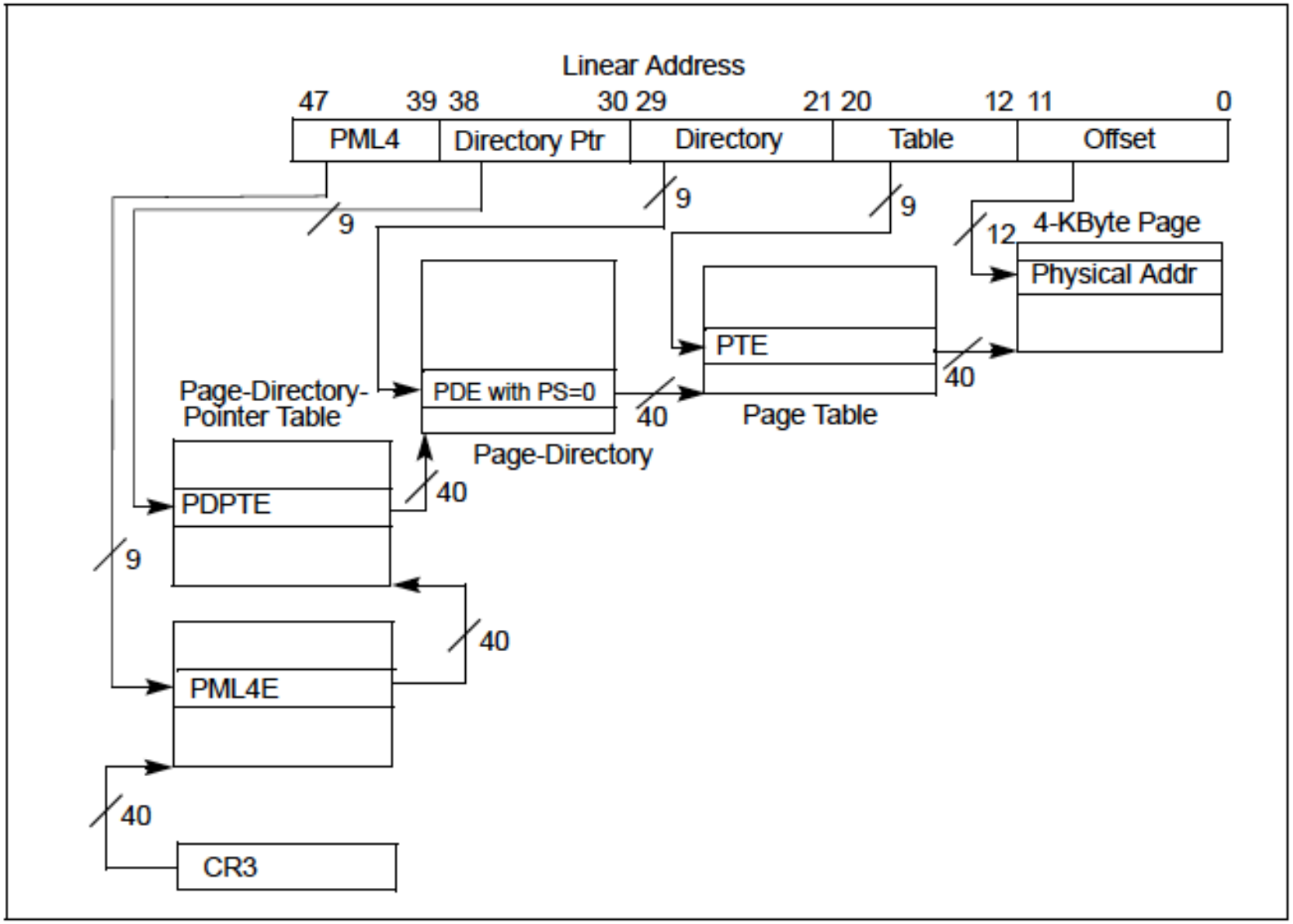


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Four-level Paging and Extended Physical Address Space in x86

A logical processor uses IA-32e paging if CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 1. With IA-32e paging, linear address are translated using a hierarchy of in-memory paging structures located using the contents of CR3. IA-32e paging translates 48-bit linear addresses to 52-bit physical addresses.[1] Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 48 bits; at most 256 TBytes of linear-address space may be accessed at any given time.

IA-32e paging uses a hierarchy of paging structures to produce a translation for a linear address. CR3 is used to locate the first paging-structure, the PML4 table. Use of CR3 with IA-32e paging depends on whether process-context identifiers (PCIDs) have been enabled by setting CR4.PCIDE:

Virtual Memory Issue II

- How fast is the address translation?
  - How can we make it fast?

- Idea: Use a hardware structure that caches PTEs → Translation lookaside buffer

- What should be done on a TLB miss?
  - What TLB entry to replace?
  - Who handles the TLB miss? HW vs. SW?

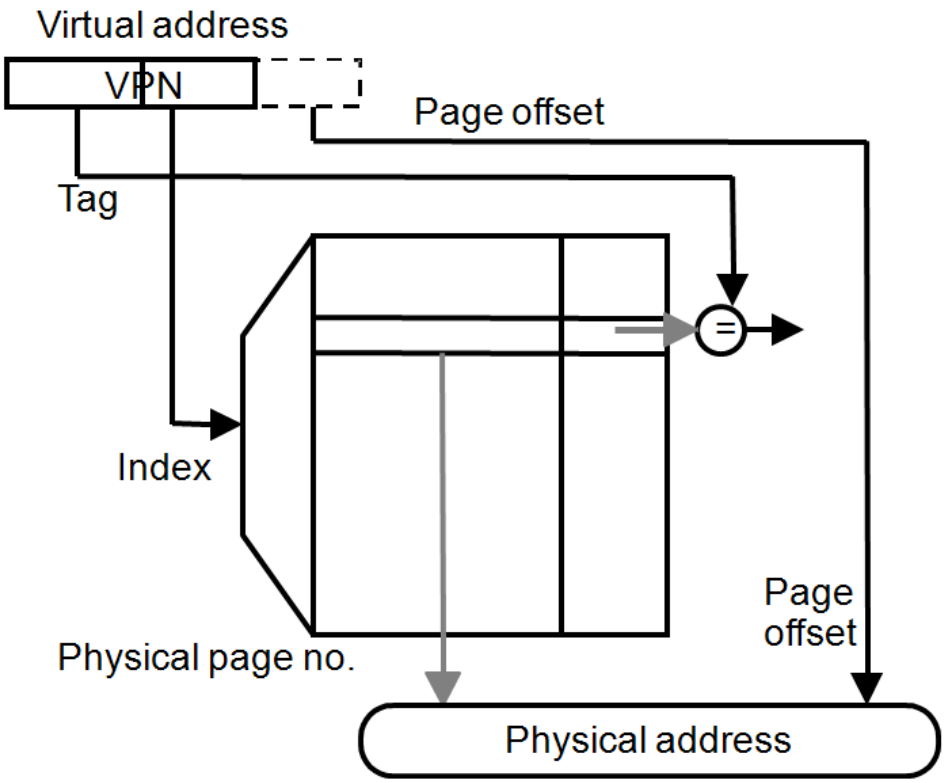
- What should be done on a page fault?
  - What virtual page to replace from physical memory?
  - Who handles the page fault? HW vs. SW?

Speeding up Translation with a TLB

- Essentially a cache of recent address translations
  - Avoids going to the page table on every reference

- **Index** = lower bits of VPN (virtual page #)
- **Tag** = unused bits of VPN + process ID
- **Data** = a page-table entry
- **Status** = valid, dirty

The usual cache design choices (placement, replacement policy, multi-level, etc.) apply here too.



Handling TLB Misses

- The TLB is small; it cannot hold all PTEs
  - Some translations will inevitably miss in the TLB
  - Must access memory to find the appropriate PTE
    - Called **walking** the page directory/table
    - Large performance penalty

- Who handles TLB misses? Hardware or software?

Handling TLB Misses (II)

- Approach #1. **Hardware-Managed** (e.g., x86)
  - The hardware does the **page walk**
  - The hardware fetches the PTE and inserts it into the TLB
    - If the TLB is full, the entry **replaces** another entry
  - Done transparently to system software

- Approach #2. **Software-Managed** (e.g., MIPS)
  - The hardware raises an exception
  - The operating system does the **page walk**
  - The operating system fetches the PTE
  - The operating system inserts/evicts entries in the TLB

Handling TLB Misses (III)

- Hardware-Managed TLB
  - Pro: No exception on TLB miss. Instruction just stalls
  - Pro: Independent instructions may continue
  - Pro: No extra instructions/data brought into caches.
  - Con: Page directory/table organization is etched into the system: OS has little flexibility in deciding these

- Software-Managed TLB
  - Pro: The OS can define page table oganization
  - Pro: More sophisticated TLB replacement policies are possible
  - Con: Need to generate an exception → performance overhead due to pipeline flush, exception handler execution, extra instructions brought to caches

Virtual Memory Issue III

- When do we do the address translation?
  - Before or after accessing the L1 cache?

Virtual Memory and Cache Interaction

Address Translation and Caching

- When do we do the address translation?
  - Before or after accessing the L1 cache?

- In other words, is the cache virtually addressed or physically addressed?
  - Virtual versus physical cache

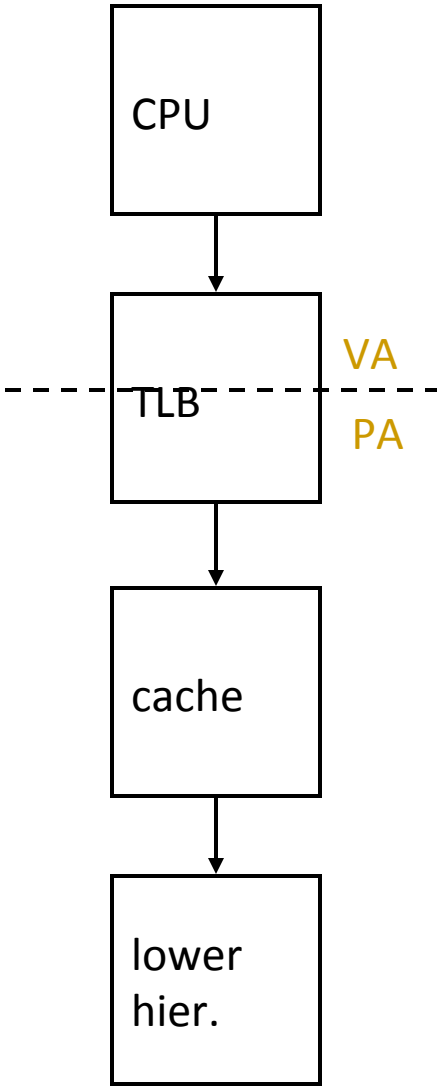
- What are the issues with a virtually addressed cache?

- Synonym problem:
  - Two different virtual addresses can map to the same physical address → same physical address can be present in multiple locations in the cache → can lead to inconsistency in data

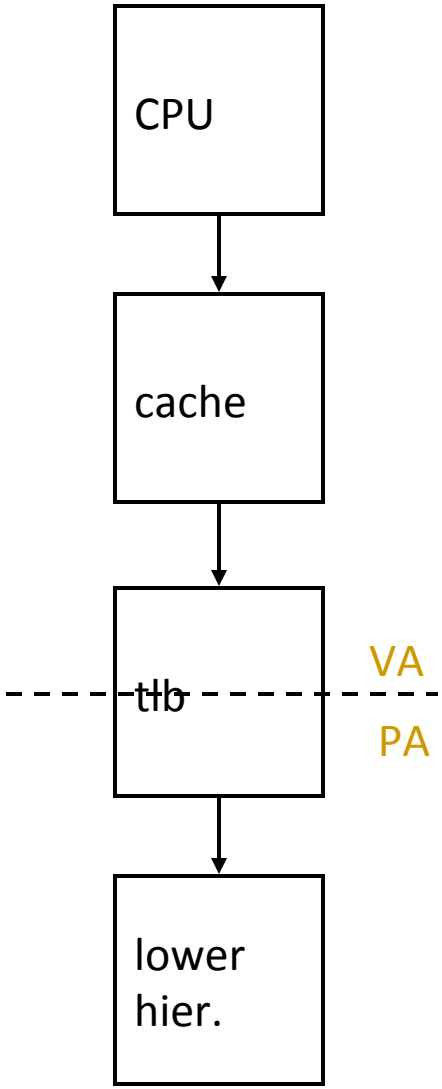
Homonyms and Synonyms

- Homonym: Same VA can map to two different PAs
  - Why?
    - VA is in different processes
- Synonym: Different VAs can map to the same PA
  - Why?
    - Different pages can share the same physical frame within or across processes
    - Reasons: shared libraries, shared data, copy-on-write pages within the same process, …
- Do homonyms and synonyms create problems when we have a cache?
  - Is the cache virtually or physically addressed?

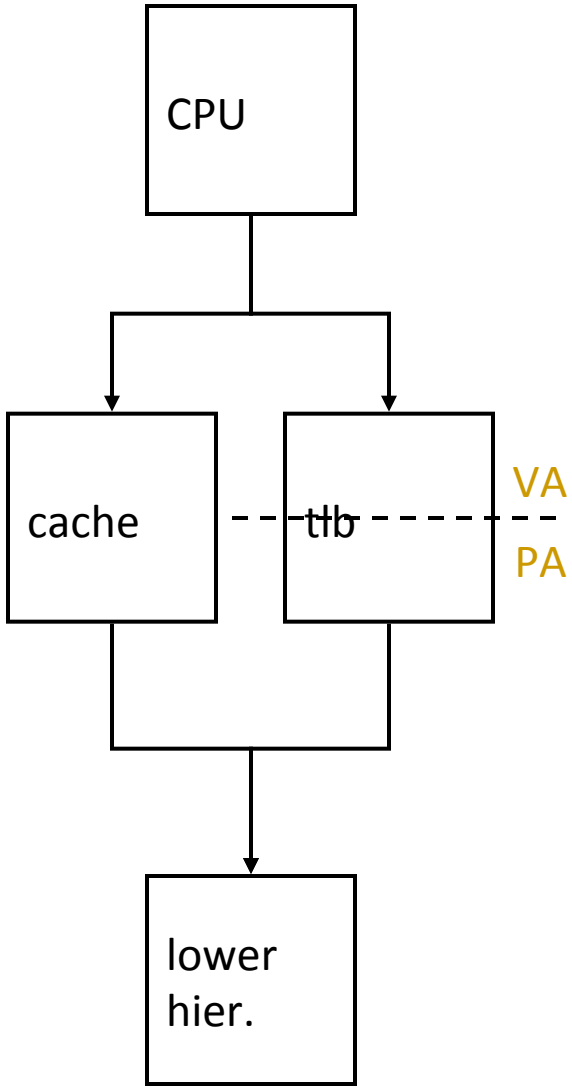
Cache-VM Interaction



physical cache

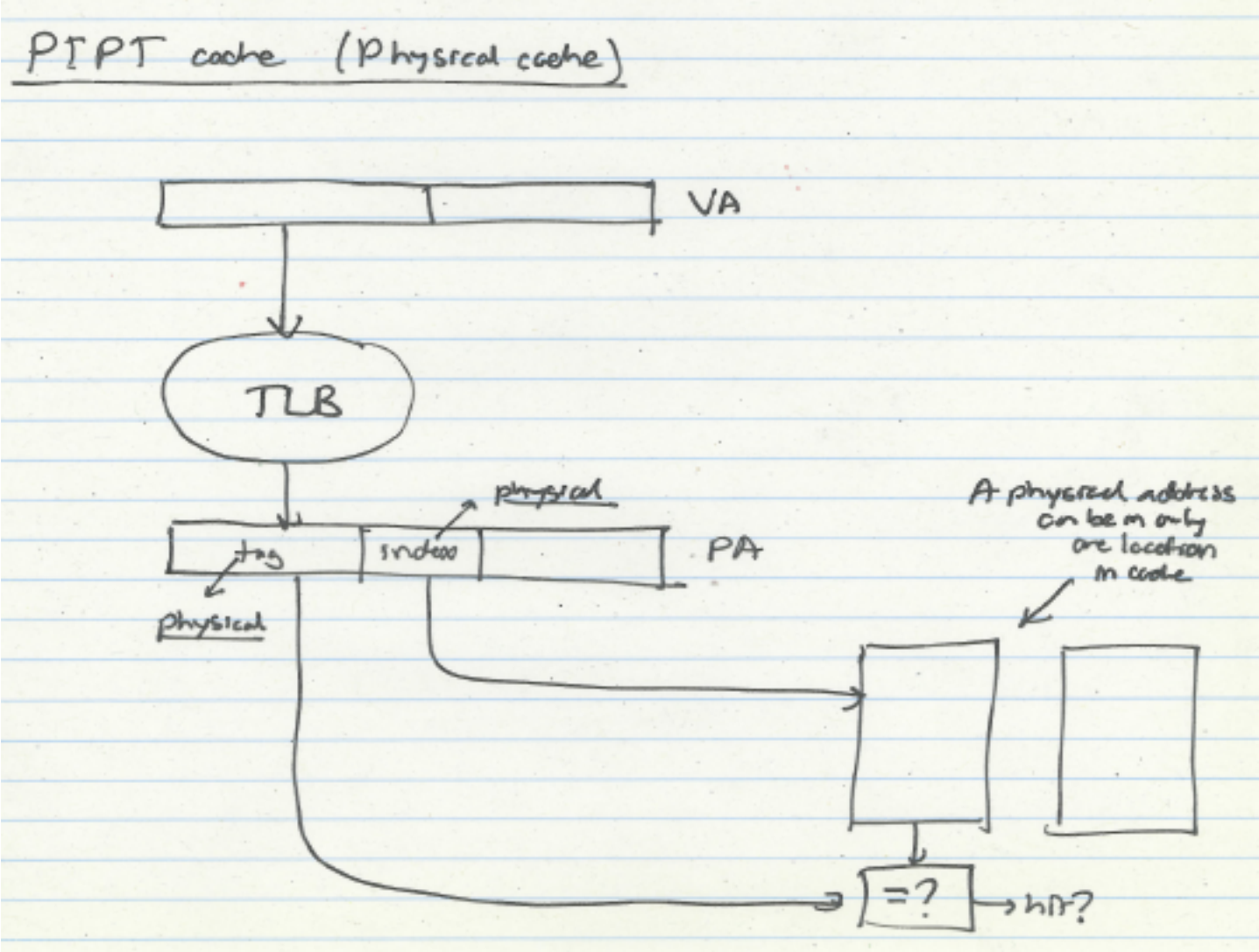


virtual (L1) cache

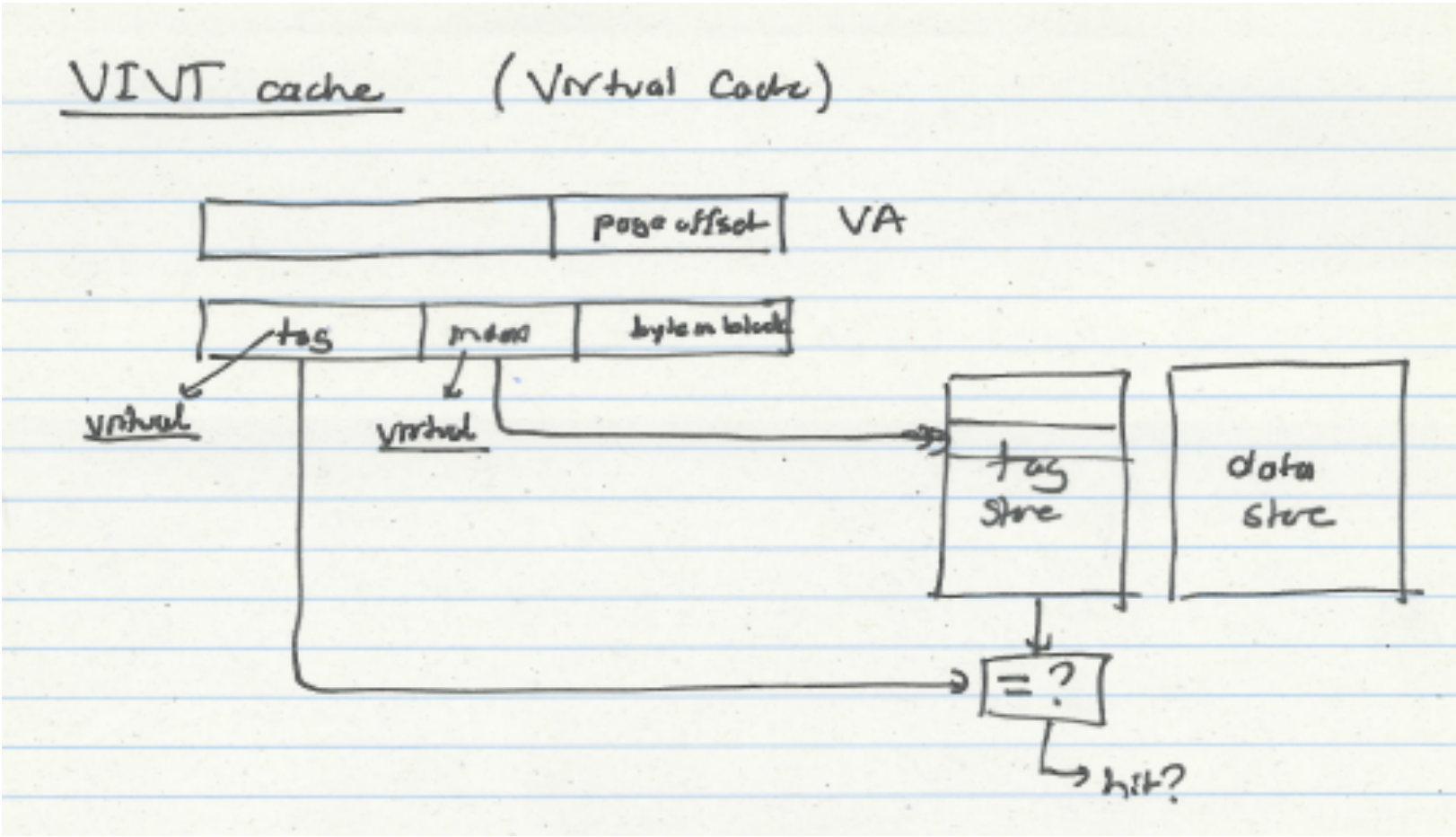


virtual-physical cache

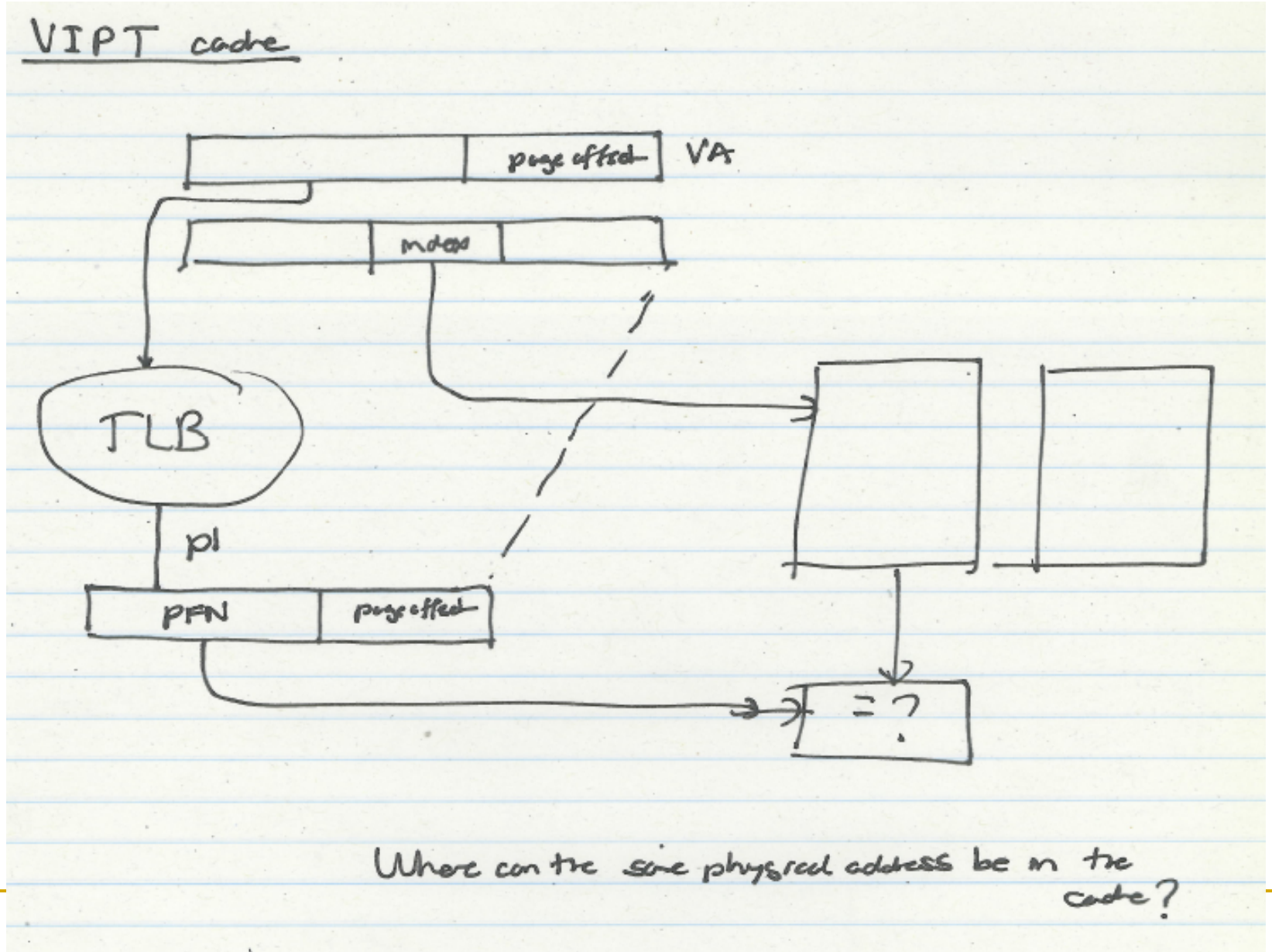
Physical Cache



Virtual Cache

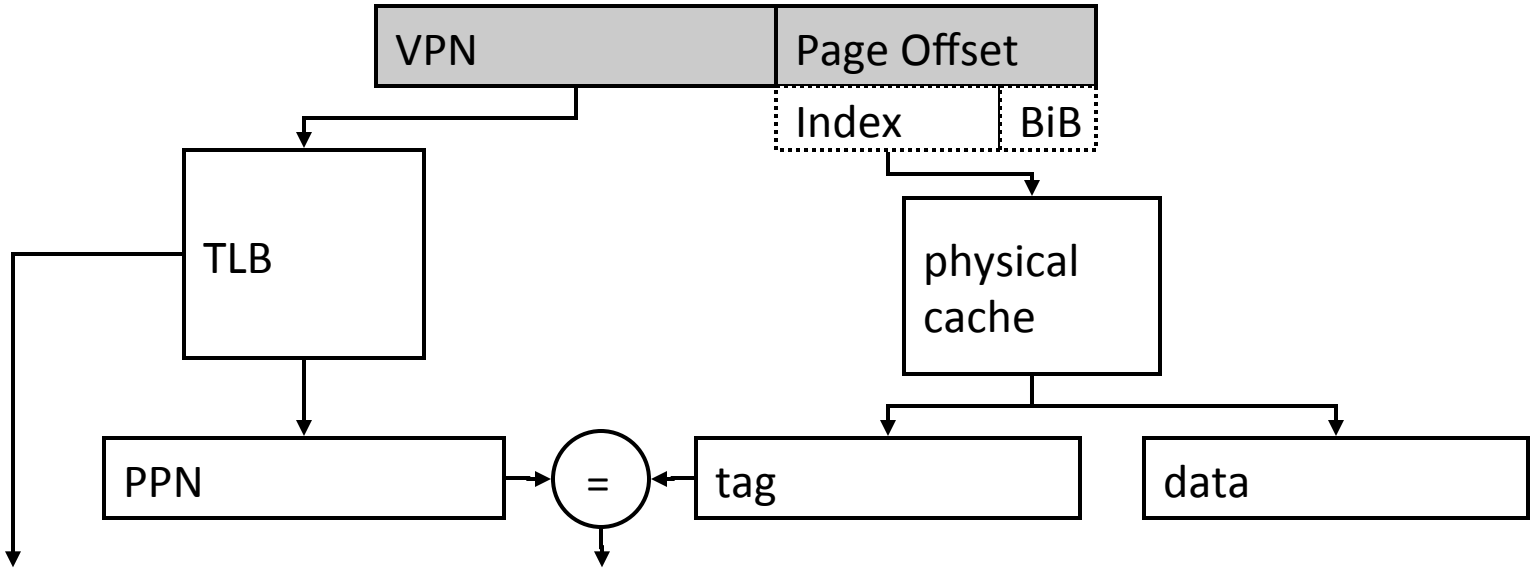


Virtual-Physical Cache



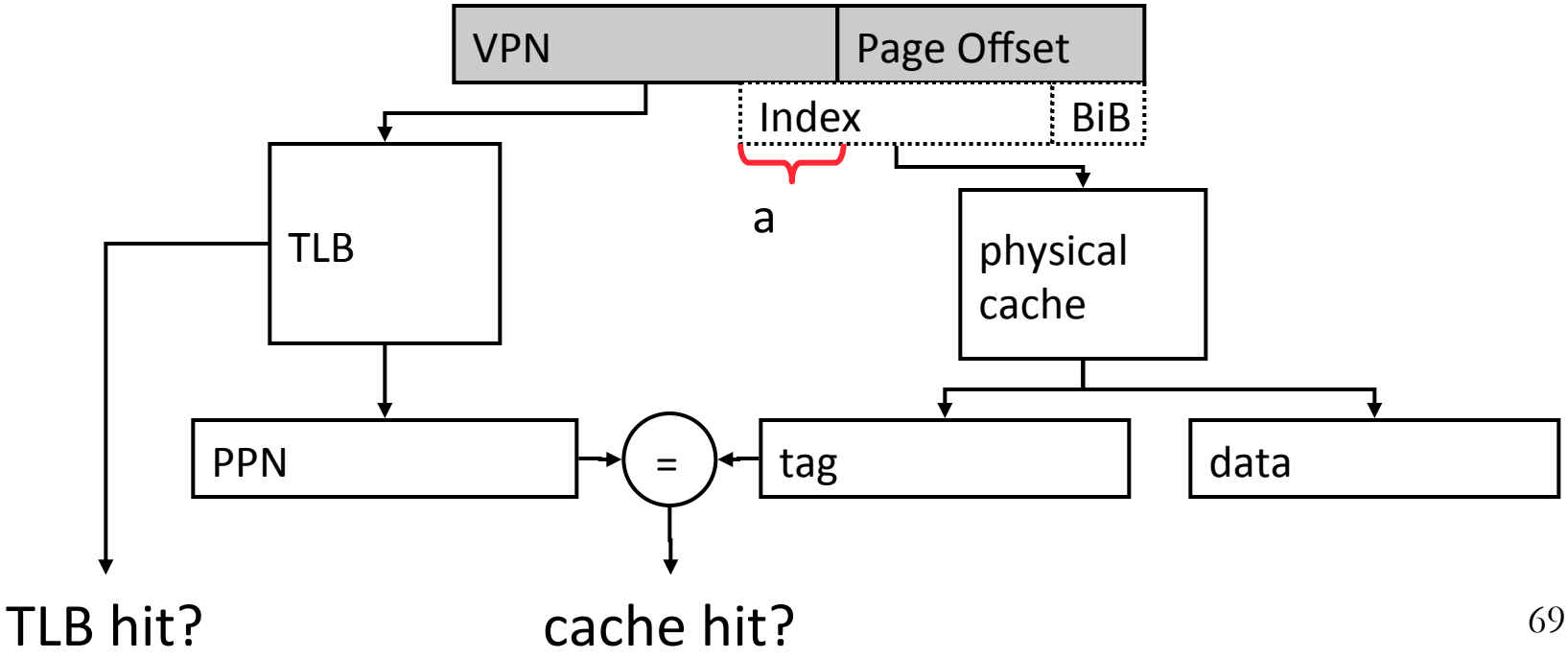
Virtually-Indexed Physically-Tagged

- If C≤(page_size × associativity), the cache index bits come only from page offset (same in VA and PA)
- If both cache and TLB are on chip
  - index both arrays concurrently using VA bits
  - check cache tag (physical) against TLB output at the end



Virtually-Indexed Physically-Tagged

- If C>(page_size × associativity), the cache index bits include VPN
  ⇒ Synonyms can cause problems
  - The same physical address can exist in two locations
- Solutions?



Some Solutions to the Synonym Problem

- Limit cache size to (page size times associativity)
  - get index from page offset
- On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate
  - Used in Alpha 21264, MIPS R10K
- Restrict page placement in OS
  - make sure index(VA) = index(PA)
  - Called page coloring
  - Used in many SPARC processors

An Exercise

- Problem 5 from
  - Past midterm exam Problem 5, Spring 2009
  - http://www.ece.cmu.edu/~ece740/f11/lib/exe/fetch.php?media=wiki:midterm:midterm_s09.pdf

An Exercise (I)

We have a byte-addressable toy computer that has a physical address space of 512 bytes. The computer uses a simple, one-level virtual memory system. The page table is always in physical memory. The page size is specified as 8 bytes and the virtual address space is 2 KB.

Part A.

i. (1 point)
How many bits of each virtual address is the virtual page number?

ii. (1 point)
How many bits of each physical address is the physical frame number?

We would like to add a 128-byte *write-through* cache to enhance the performance of this computer. However, we would like the cache access and address translation to be performed simultaneously. In other words, we would like to index our cache using a virtual address, but do the tag comparison using the physical addresses (virtually-indexed physically-tagged). The cache we would like to add is direct-mapped, and has a block size of 2 bytes. The replacement policy is LRU. Answer the following questions:

iii. (1 point)
How many bits of a virtual address are used to determine which byte in a block is accessed?

iv. (2 point)
How many bits of a virtual address are used to index into the cache? Which bits exactly?

v. (1 point)
How many bits of the virtual page number are used to index into the cache?

vi. (5 points)
What is the size of the tag store in bits? Show your work.

Part B.

Suppose we have two processes sharing our toy computer. These processes share some portion of the physical memory. Some of the virtual page-physical frame mappings of each process are given below:

PROCESS 0	
Virtual Page	Physical Frame
Page 0	Frame 0
Page 3	Frame 7
Page 7	Frame 1
Page 15	Frame 3

PROCESS 1	
Virtual Page	Physical Frame
Page 0	Frame 4
Page 1	Frame 5
Page 7	Frame 3
Page 11	Frame 2

vii. (2 points)
Give a complete physical address whose data can exist in two different locations in the cache.

viii. (3 points)
Give the indexes of those two different locations in the cache.

An Exercise (Concluded)

ix. (5 points)

We do not want the same physical address stored in two different locations in the 128-byte cache. We can prevent this by increasing the associativity of our virtually-indexed physically-tagged cache. What is the minimum associativity required?

x. (4 points)

Assume we would like to use a direct-mapped cache. Describe a solution that ensures that the same physical address is never stored in two different locations in the 128-byte cache.

Solutions to the Exercise

- http://www.ece.cmu.edu/~ece740/f11/lib/exe/fetch.php?media=wiki:midterm:midterm_s09_solution.pdf

- And, more exercises are in past exams and in your homeworks…

Review: Solutions to the Synonym Problem

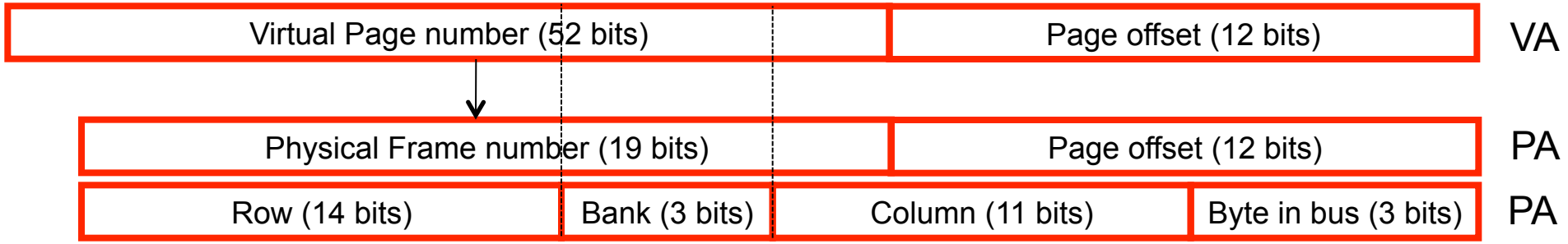
- Limit cache size to (page size times associativity)
  - get index from page offset
- On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate
  - Used in Alpha 21264, MIPS R10K
- Restrict page placement in OS
  - make sure index(VA) = index(PA)
  - Called page coloring
  - Used in many SPARC processors

Some Questions to Ponder

- At what cache level should we worry about the synonym and homonym problems?
- What levels of the memory hierarchy does the system software's page mapping algorithms influence?
- What are the potential benefits and downsides of page coloring?

Fast Forward: Virtual Memory – DRAM Interaction

- Operating System influences where an address maps to in DRAM



- Operating system can control which bank/channel/rank a virtual page is mapped to.

- It can perform page coloring to minimize bank conflicts
- Or to minimize inter-application interference

We did not cover the following slides. They are for your benefit.

Protection and Translation without Virtual Memory

Aside: Protection w/o Virtual Memory

- Question: Do we need virtual memory for protection?
- Answer: No
- Other ways of providing memory protection
  - Base and bound registers
  - Segmentation
- None of these are as elegant as page-based access control
  - They run into complexities as we need more protection capabilites

Very Quick Overview: Base and Bound

In a multi-tasking system

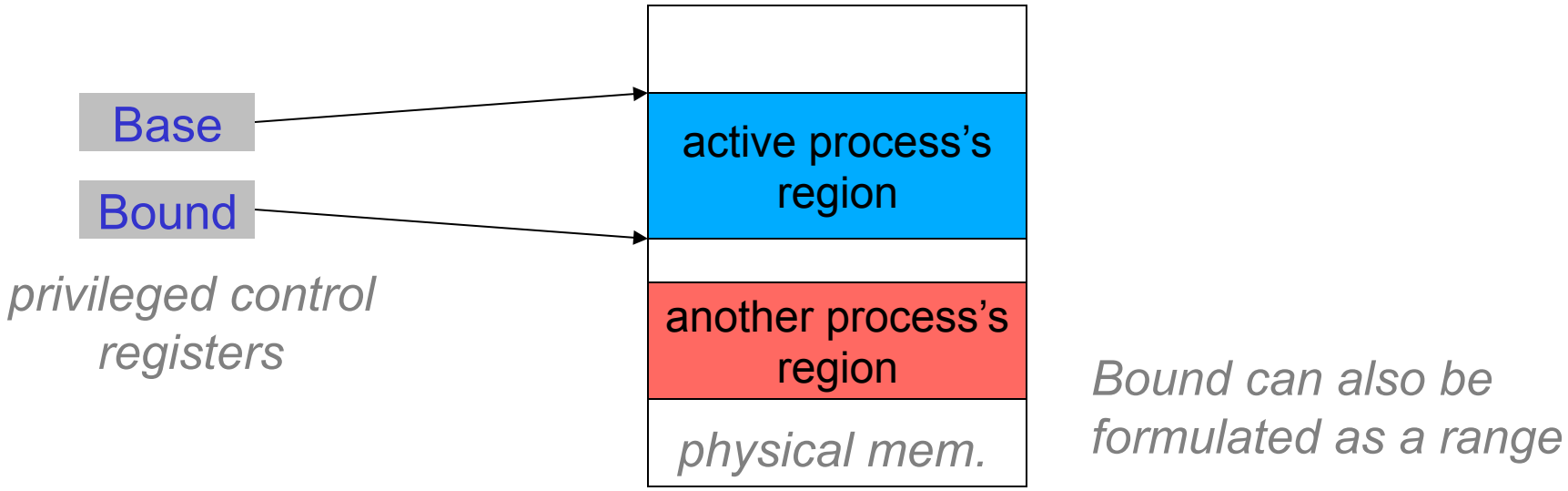
Each process is given a non-overlapping, contiguous physical memory region, *everything belonging to a process must fit in that region*

When a process is swapped in, OS sets base to the start of the process's memory region and bound to the end of the region

HW translation and protection check *(on each memory reference)*

PA = EA + base, provided (PA < bound), else violations

⇒ *Each process sees a private and uniform address space (0 .. max)*

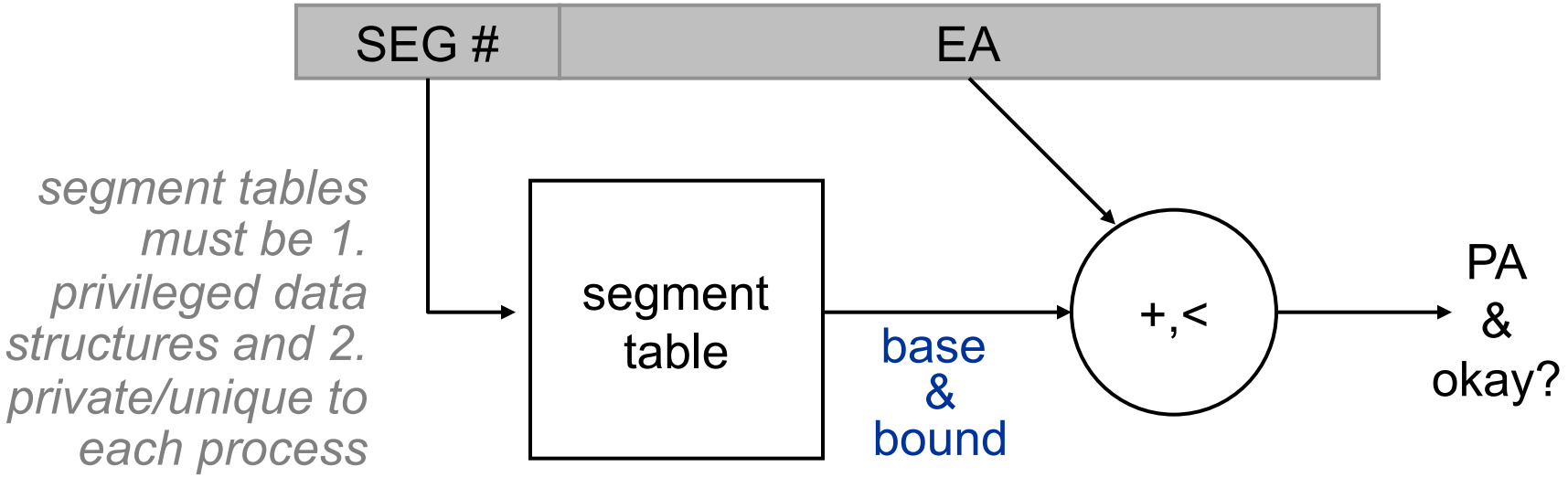


Very Quick Overview: Base and Bound (II)

- Limitations of the base and bound scheme
  - large contiguous space is hard to come by after the system runs for a while---free space may be fragmented
  - how do two processes share some memory regions but not others?

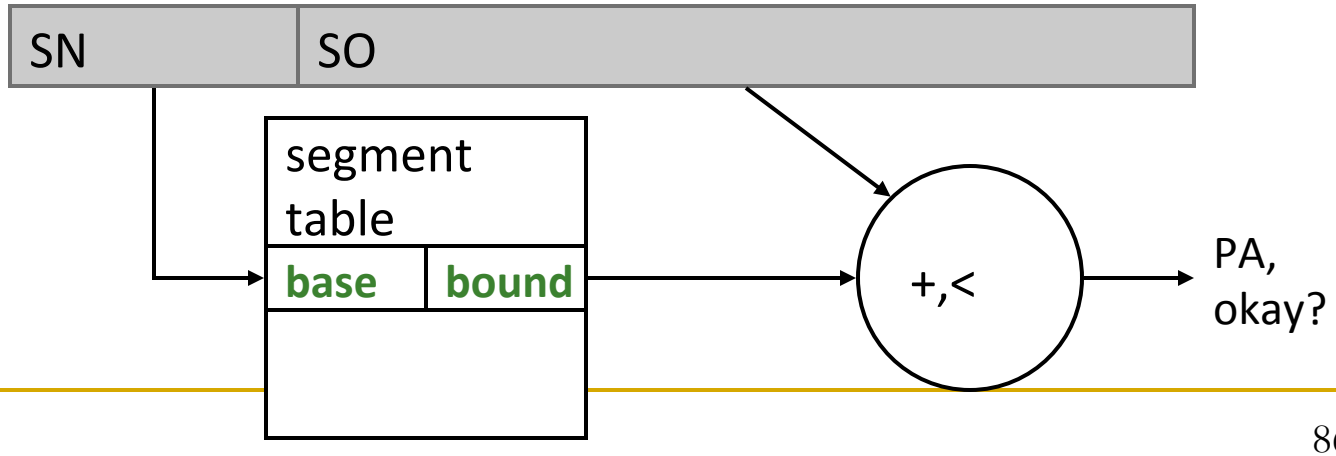
Segmented Address Space

- segment == a base and bound pair
- segmented addressing gives each process multiple segments
  - initially, separate code and data segments
    - *2 sets of base-and-bound reg's for inst and data fetch*
    - *allowed sharing code segments*
  - became more and more elaborate: *code, data, stack, etc.*



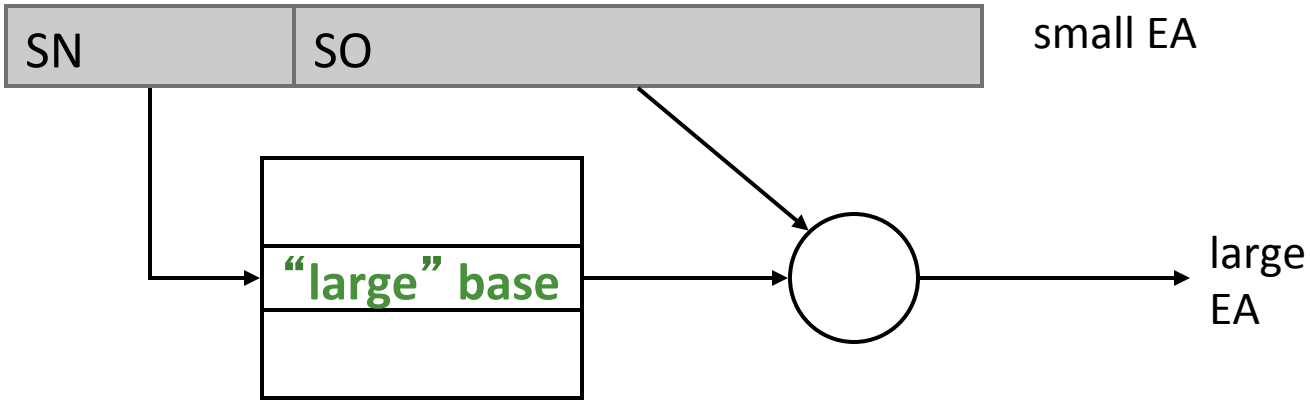
Segmented Address Translation

- EA: segment number (SN) and a segment offset (SO)
  - SN may be specified explicitly or implied (code vs. data)
  - segment size limited by the range of SO
  - segments can have different sizes, not all SOs are meaningful
- Segment translation and protection table
  - maps SN to corresponding base and bound
  - separate mapping for each process
  - must be a privileged structure



Segmentation as a Way to Extend Address Space

- How to extend an old ISA to support larger addresses for new applications while remaining compatible with old applications?



Issues with Segmentation

- Segmented addressing creates fragmentation problems:
  - a system may have plenty of unallocated memory locations
  - they are useless if they do not form a contiguous region of a sufficient size

- Page-based virtual memory solves these issues
  - By ensuring the address space is divided into fixed size "pages"
  - And virtual address space of each process is contiguous
  - The key is the use of indirection to give each process the illusion of a contiguous address space

Page-based Address Space

- In a Paged Memory System:
- PA space is divided into fixed size "segments" (e.g., 4kbyte),
  more commonly known as "page frames"
- VA is interpreted as page number and page offset