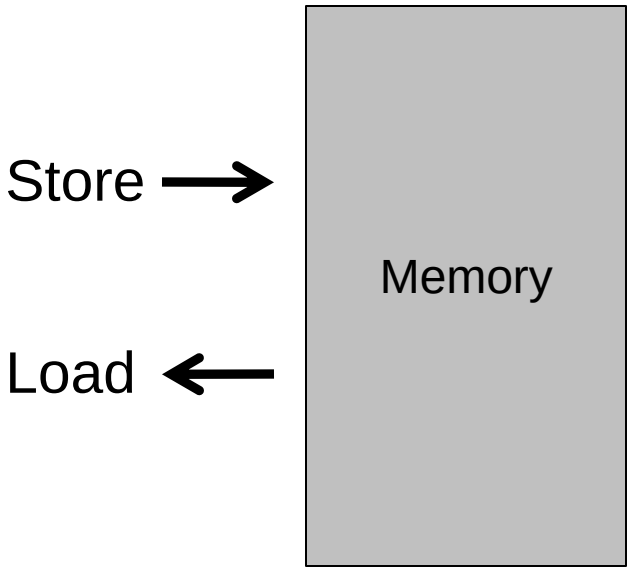
18-447
Computer Architecture
Lecture 12: Virtual Memory I

Lecturer: Rachata Ausavarungnirun
Carnegie Mellon University
Spring 2014, 2/14/2014
(with material from Onur Mutlu, Justin Meza and Yoongu Kim)

Announcements

- Lab 3 due Friday (Feb 21)
- HW 3 is out

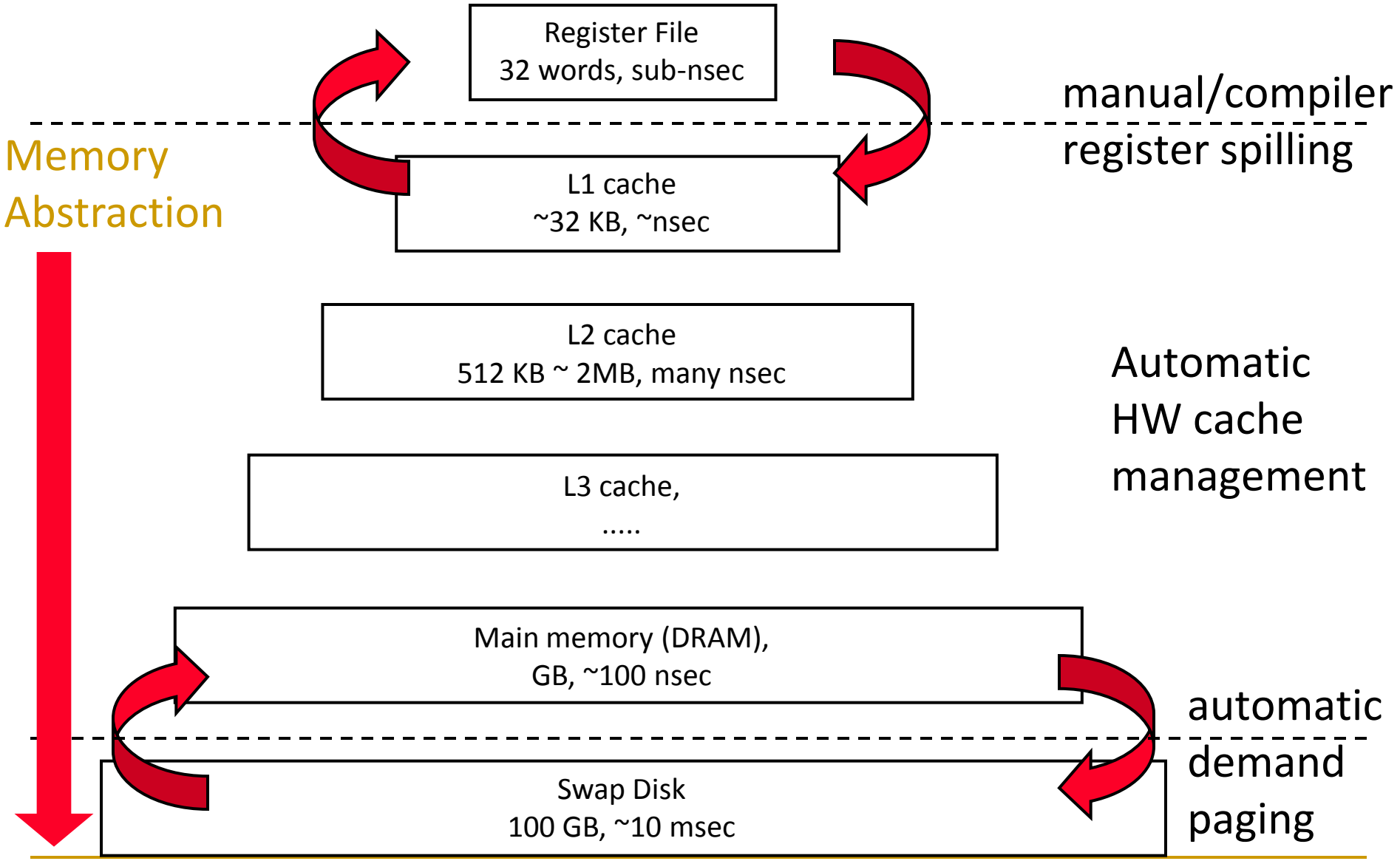
Memory: Programmer's View



Ideal Memory

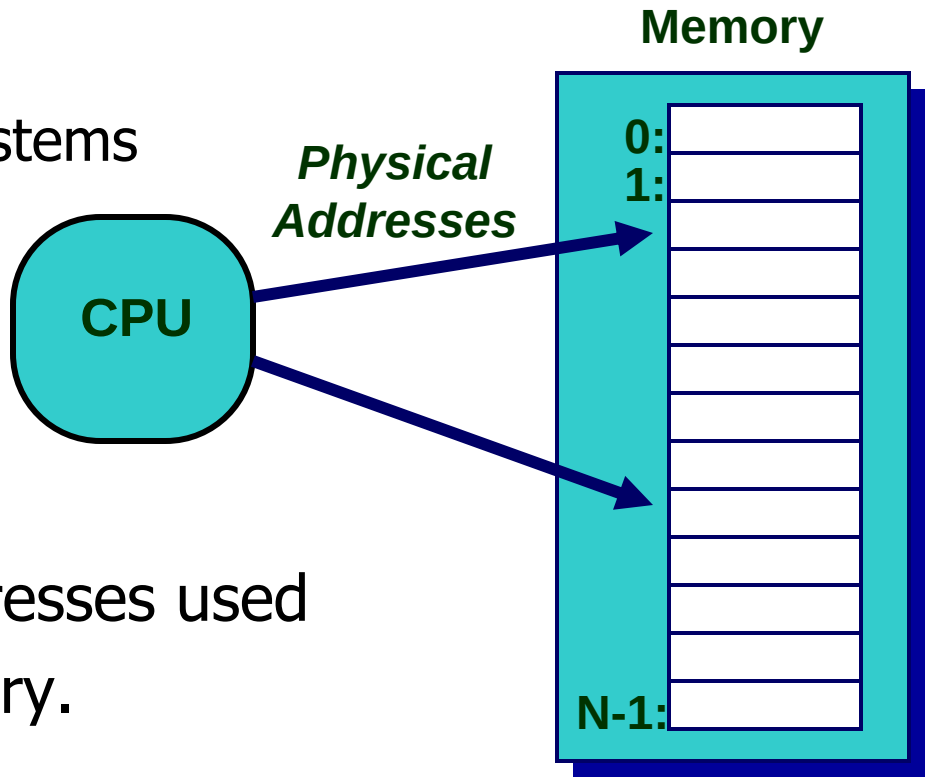
- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

A Modern Memory Hierarchy



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? (overlay programming, programming with some embedded systems)
  - Should the programmer manage data movement from disk to physical memory?

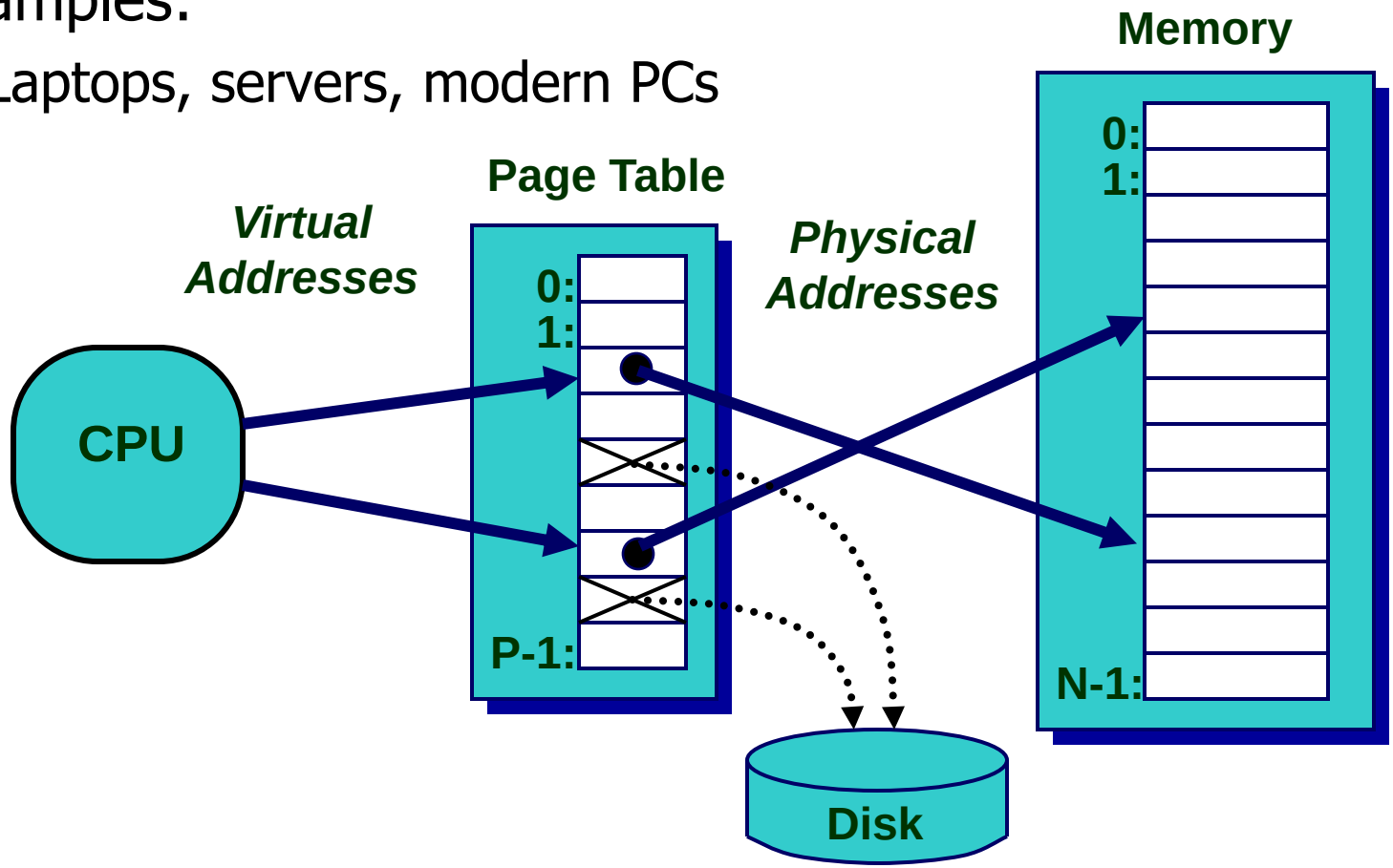
- Also, ISA can have an address space greater than the physical memory size
  - E.g., a 64-bit address space with byte addressability
  - What if you do not have enough physical memory?

Basic Mechanism

- Indirection
- 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 is implemented in hardware and software together

A System with Virtual Memory (page-based)

- Examples:
  - Laptops, servers, modern PCs



- 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
  - Assuming the page is in memory
- 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 → demand paging
- Page table is the table that stores the mapping of virtual pages to physical frames

What do we need to support VM?

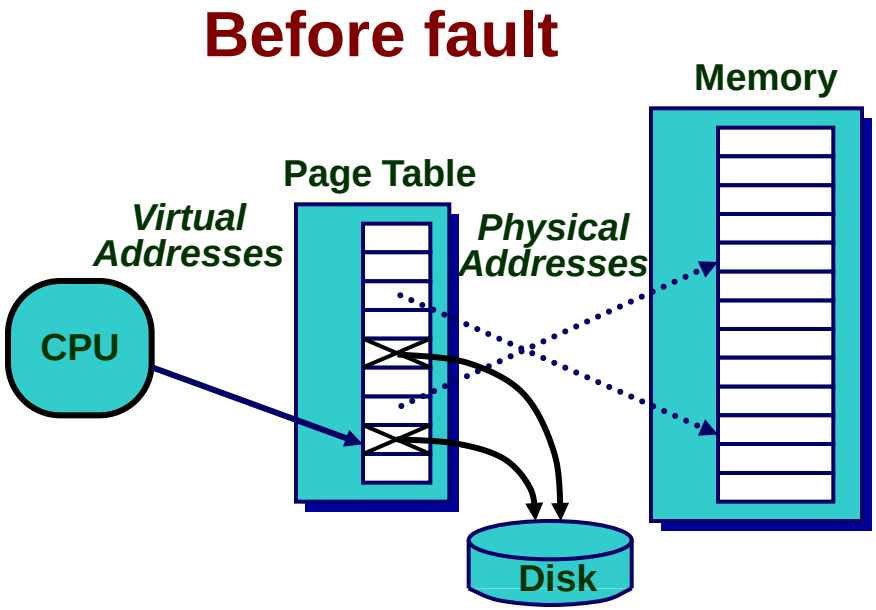
- Virtual memory requires both HW+SW support
- The hardware component is called the MMU
  - Most of what's been explained today is done by the MMU
- It is the job of the software to leverage the MMU
  - Populate page directories and page tables
  - Modify the Page Directory Base Register on context switch
  - Set correct permissions
  - Handle page faults
  - Etc.

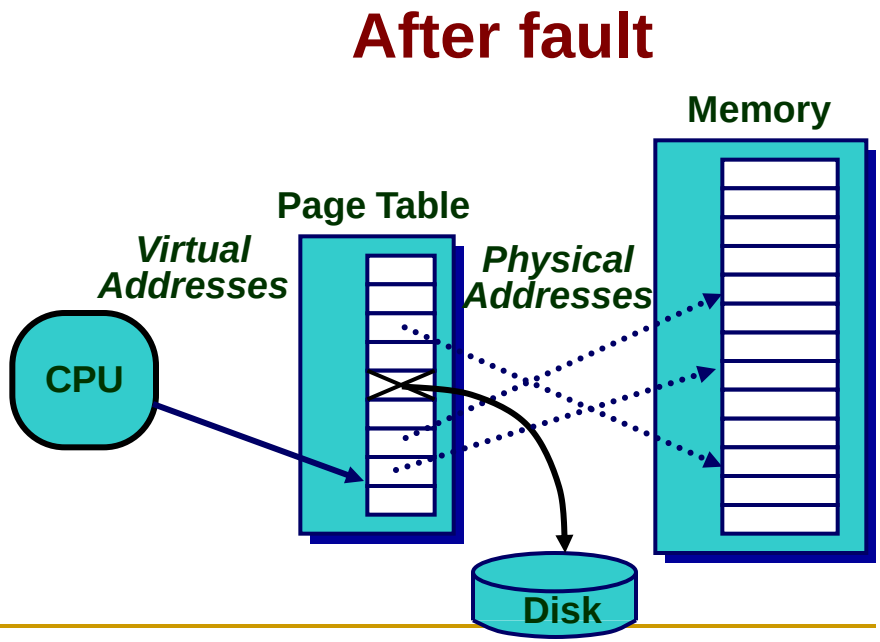
Additional Jobs from the Software Side

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

Page Fault ("A miss in physical memory")

- What if object is on disk rather than in memory?
  - Page table entry indicates virtual page not in memory → page fault exception
  - OS trap handler invoked to move data from disk into memory
    - Current process suspends, others can resume
    - OS has full control over placement





Servicing a Page Fault

(1) Processor signals controller

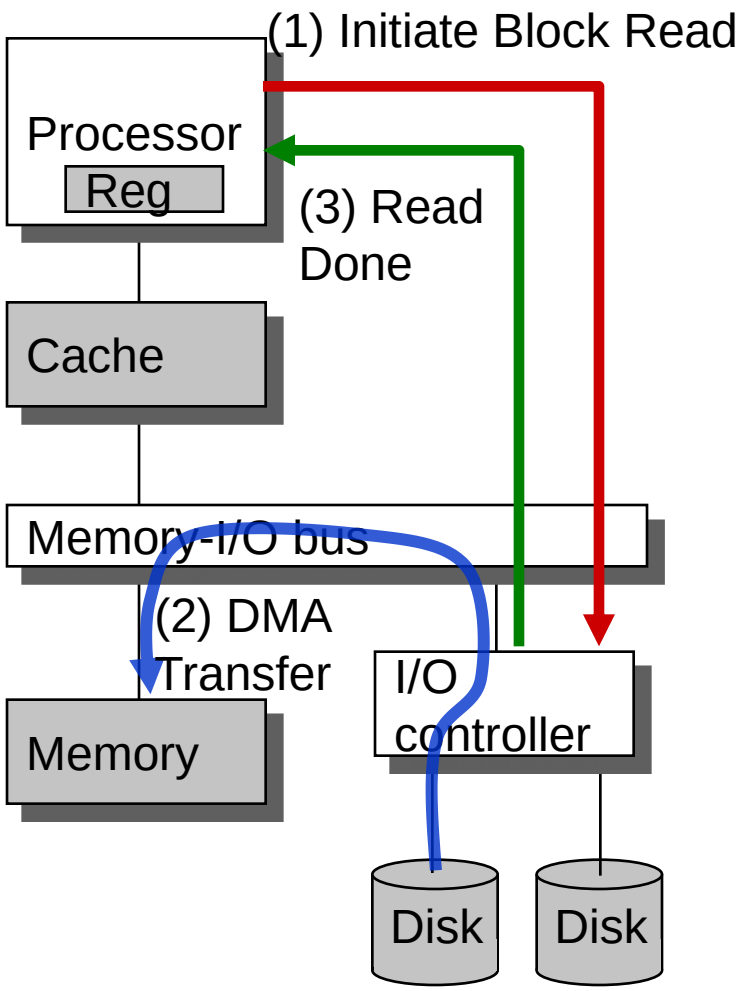
- Read block of length P starting at disk address X and store starting at memory address Y

(2) Read occurs

- Direct Memory Access (DMA)
- Under control of I/O controller

(3) Controller signals completion

- Interrupt processor
- OS resumes suspended process



Page Swap

- Swapping
  - You are running many programs that require lots of memory
- What happens if you try to run another program?
  - Some physical pages are "swapped out" to disk
  - The data in some physical pages are migrated to disk
  - This frees up those physical pages
  - As a result, their PTEs become invalid
- When you access a physical page that has been swapped out, only then is it brought back into physical memory
  - This may cause another physical page to be swapped out
  - If this "ping-ponging" occurs frequently, it is called thrashing
  - Extreme performance degradation

Address Translation

- How to get 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?

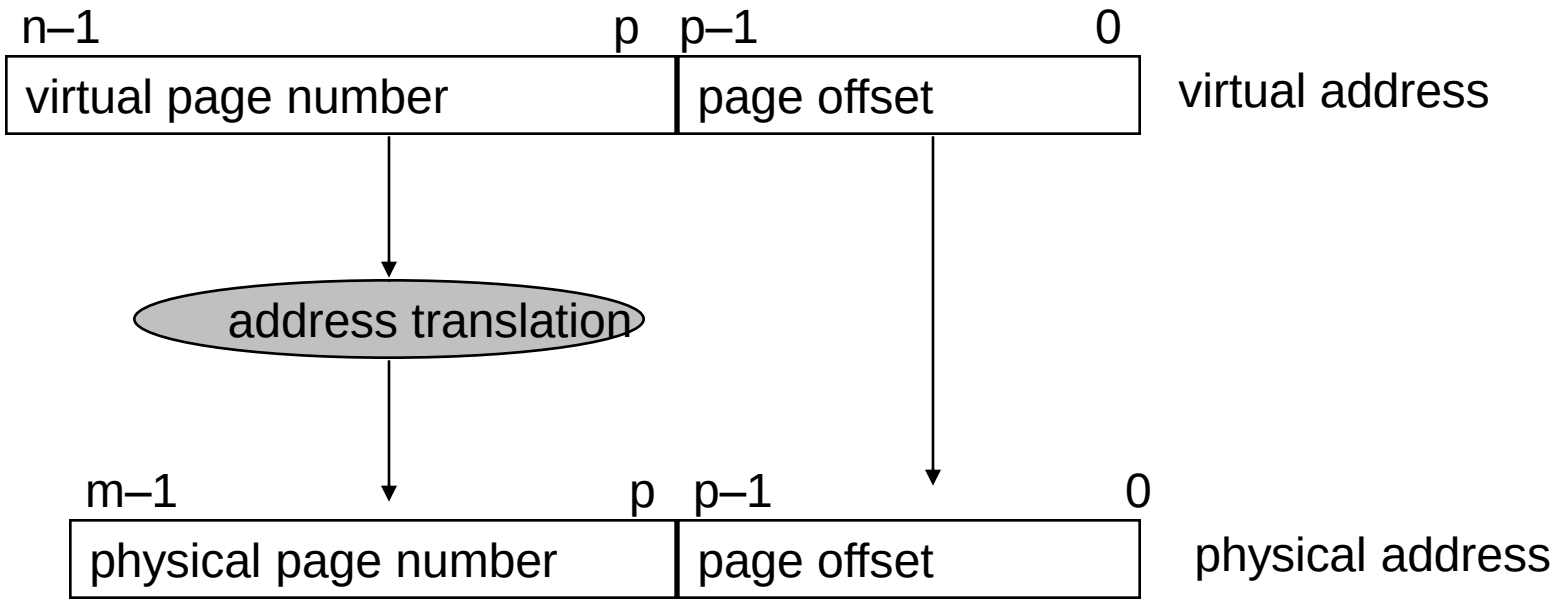
- Page Table contains an entry for each virtual page
  - Called Page Table Entry (PTE)
  - What is in a PTE?

Trade-Offs in Page Size

- Large page size (e.g., 1GB)
  - Pro: Fewer PTEs required · Saves memory space
  - Pro: Fewer TLB misses · Improves performance
  - Con: Large transfers to/from disk
    - Even when only 1KB is needed, 1GB must be transferred
    - Waste of bandwidth/energy
    - Reduces performance
  - Con: Internal fragmentation
    - Even when only 1KB is needed, 1GB must be allocated
    - Waste of space
    - Q: What is external fragmentation?
  - Con: Cannot have fine-grained permissions

VM Address Translation

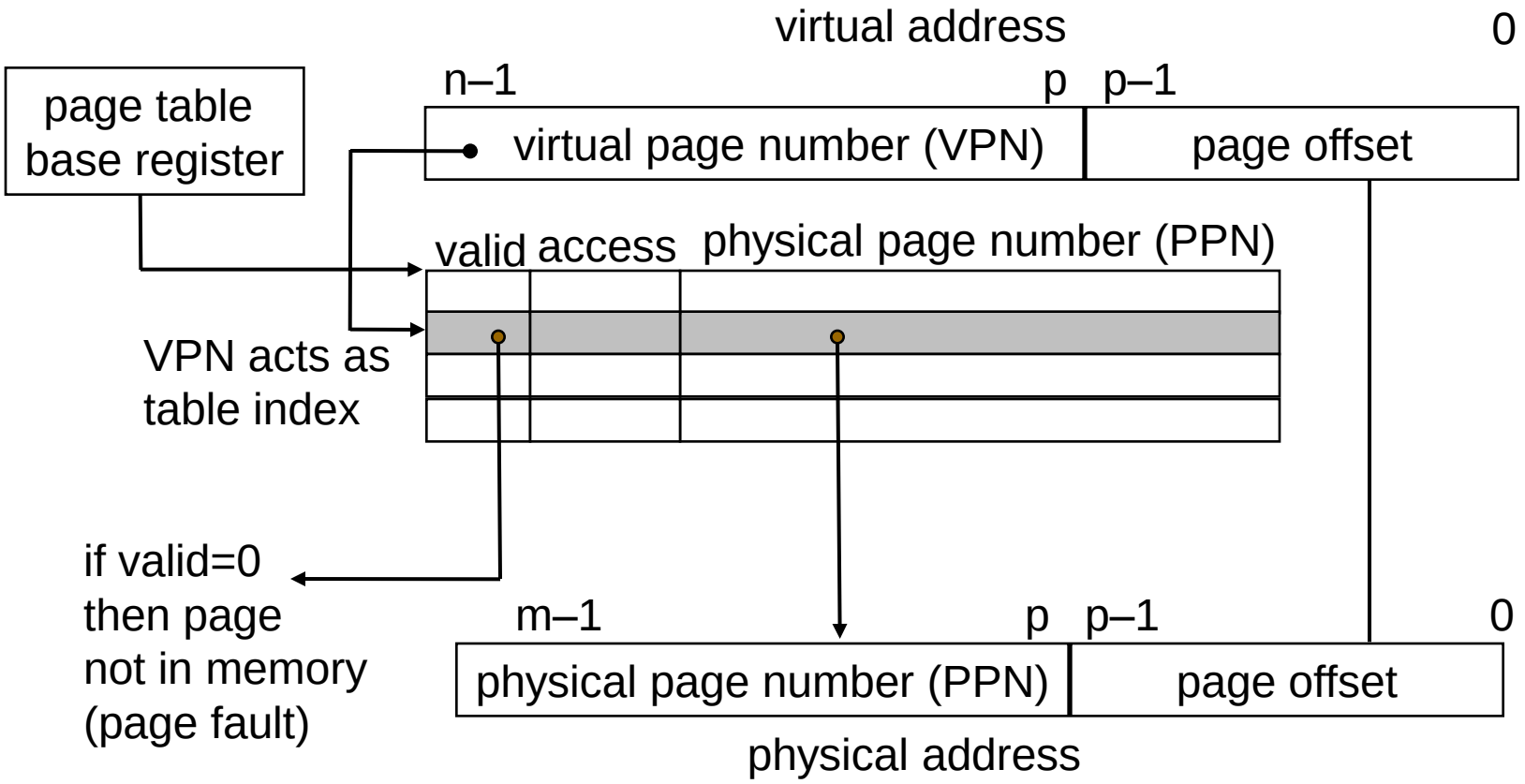
- Parameters
  - $P = 2^p$ = page size (bytes).
  - $N = 2^n$ = Virtual-address limit
  - $M = 2^m$ = Physical-address limit



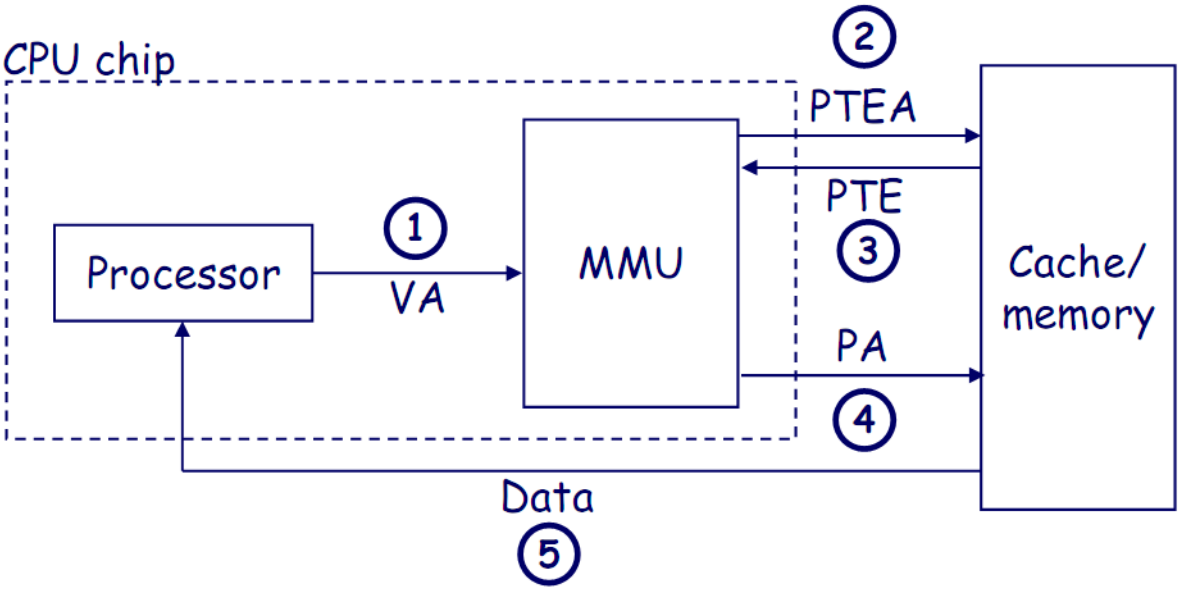
Page offset bits don't change as a result of translation

VM Address Translation

- 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



VM Address Translation: Page Hit



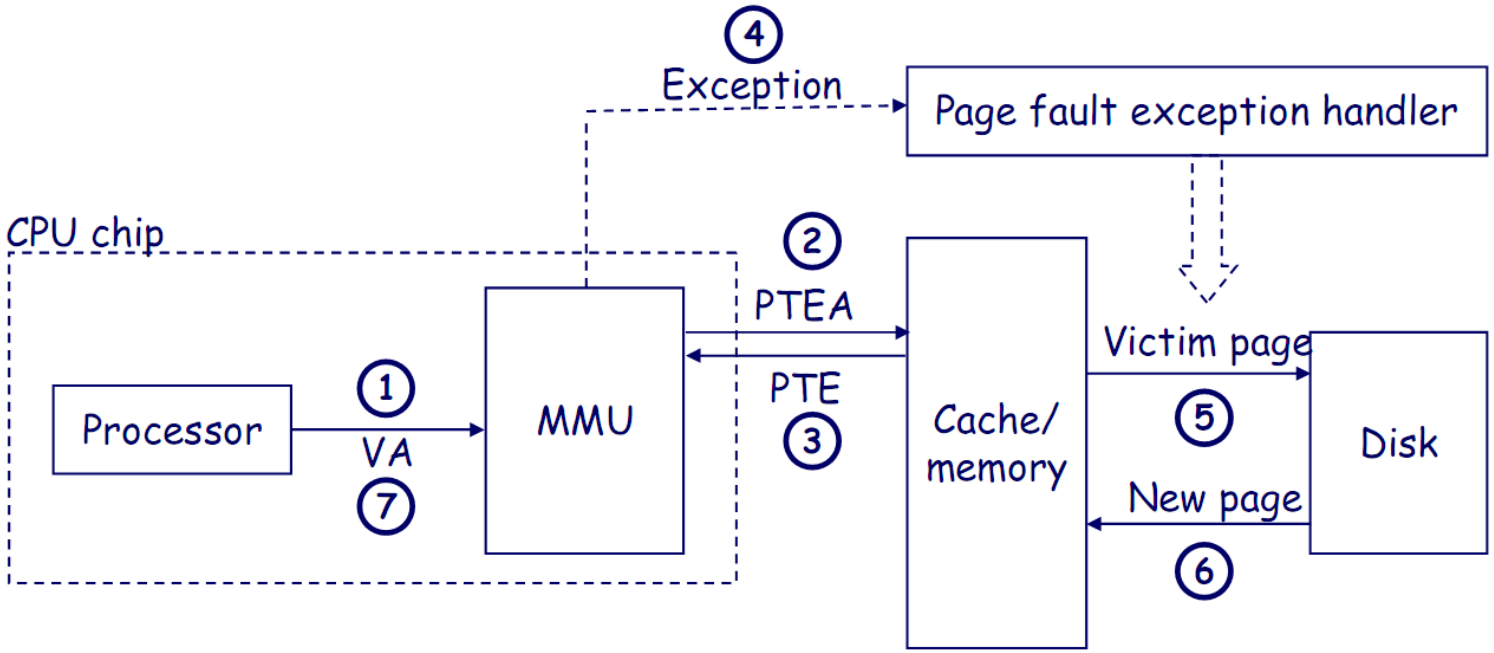
1) Processor sends virtual address to MMU

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

VM Address Translation: Page Fault

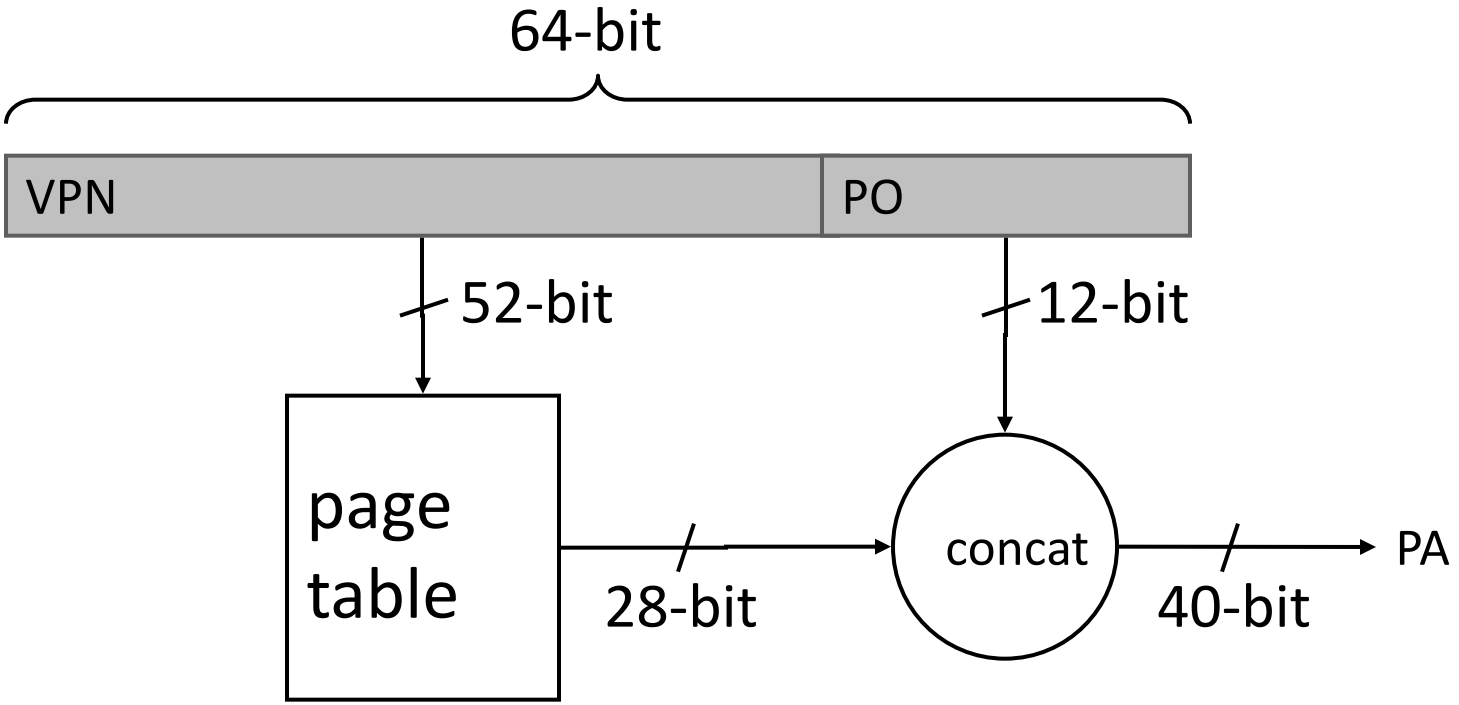


1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim, and if dirty pages it out to disk
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process,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

Issues

- 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)

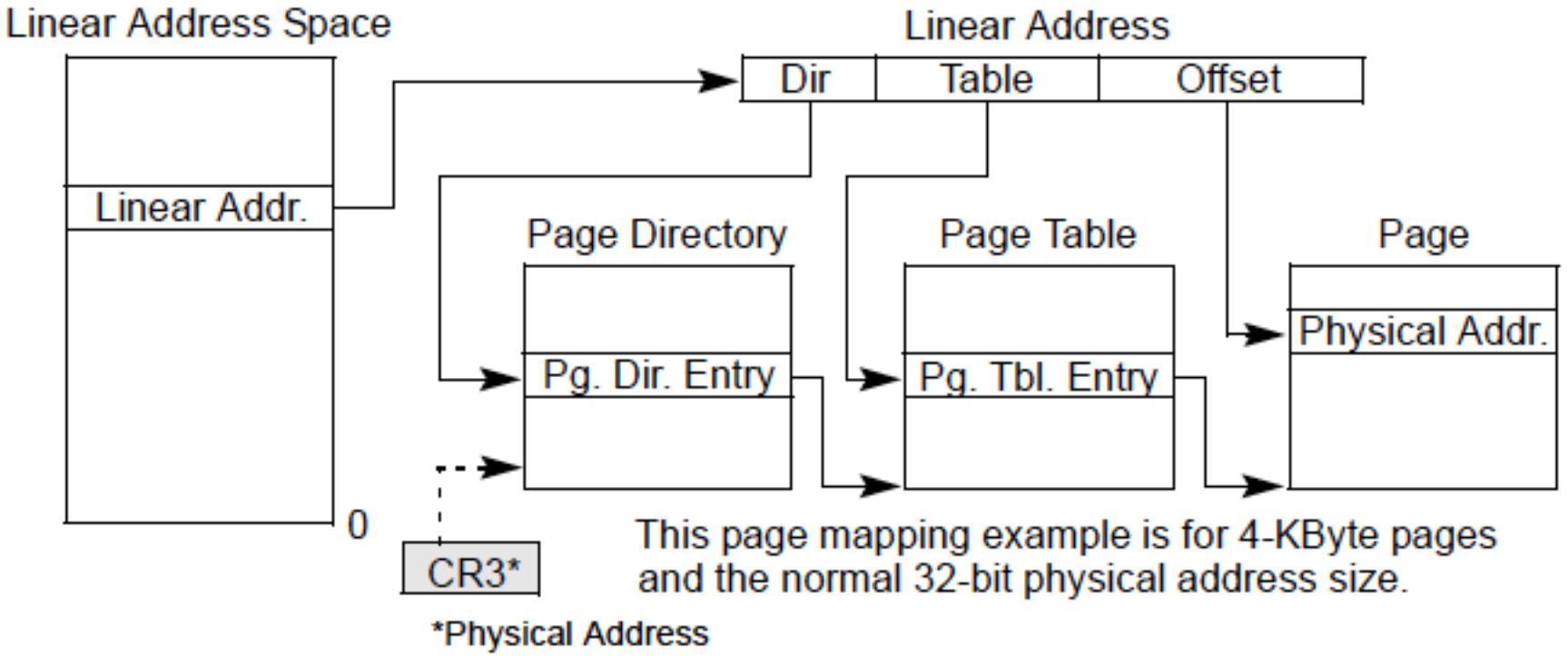
Issue: Page Table Size



- 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!!?

Multi-Level Page Tables in x86



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, PSR, GPRs
  - Needs to be loaded when the process is context-switched in

More on x86 Page Tables (I)

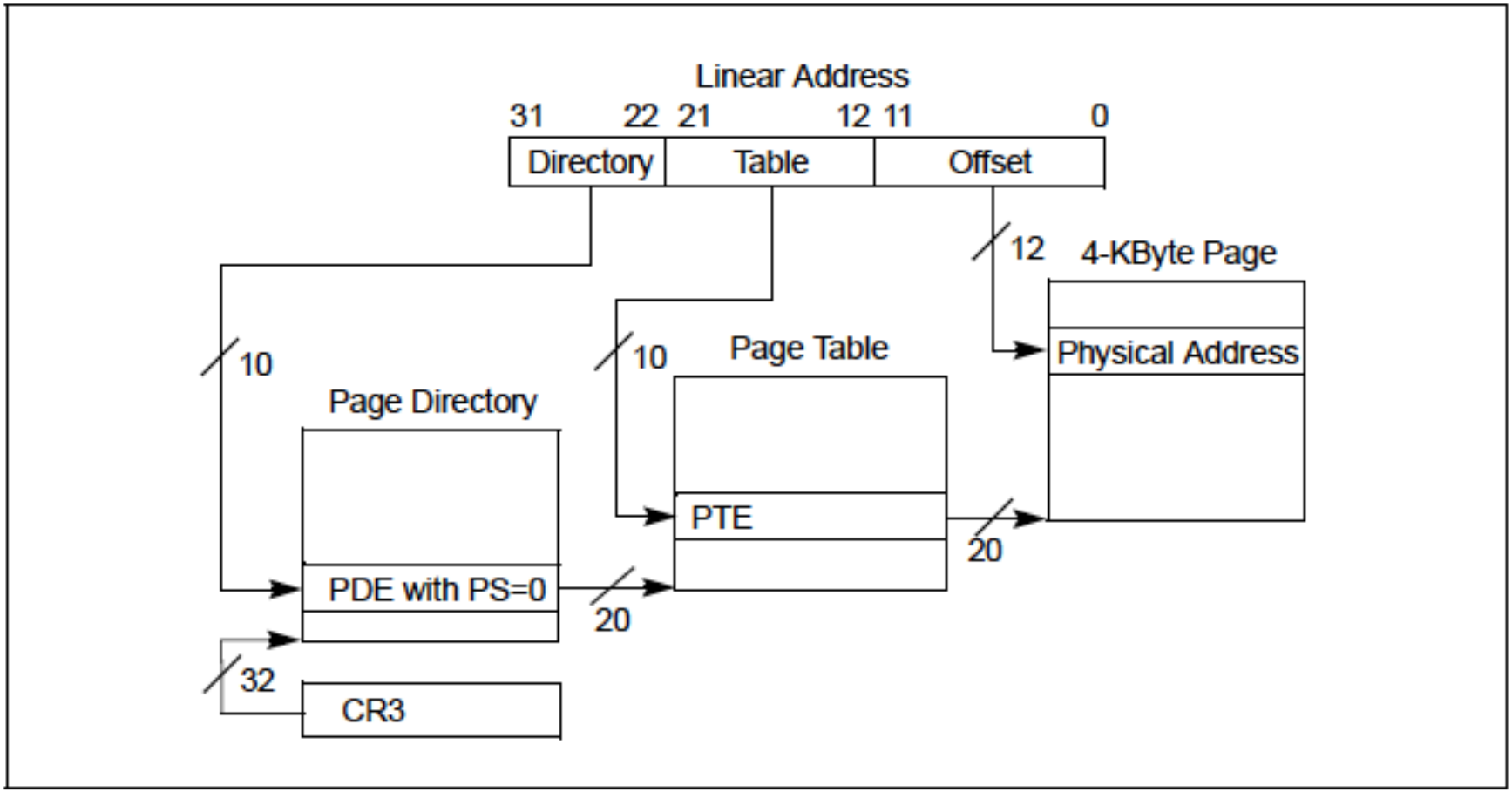


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

More on x86 Page Tables (II): Large Pages

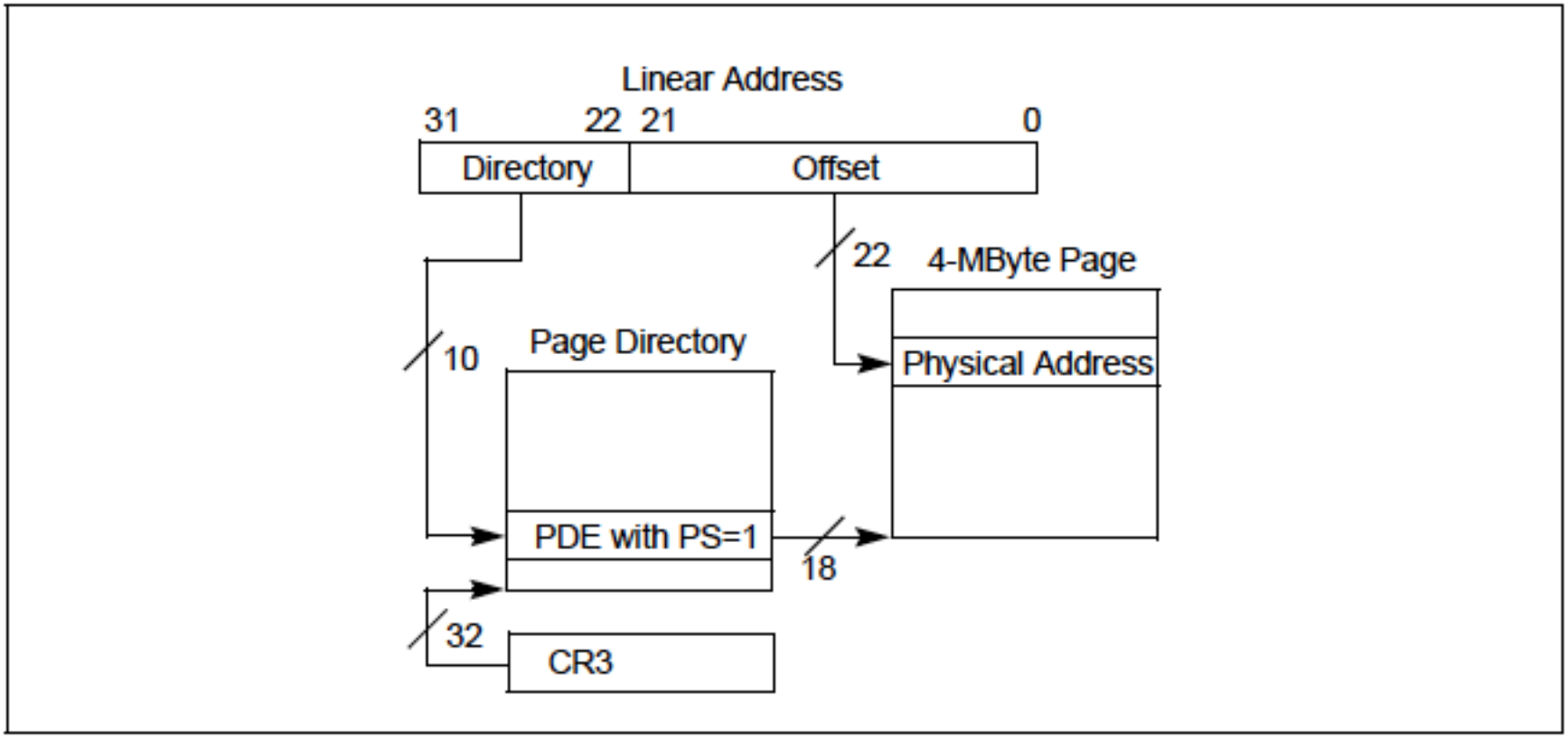


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

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.

31–12	11–9	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

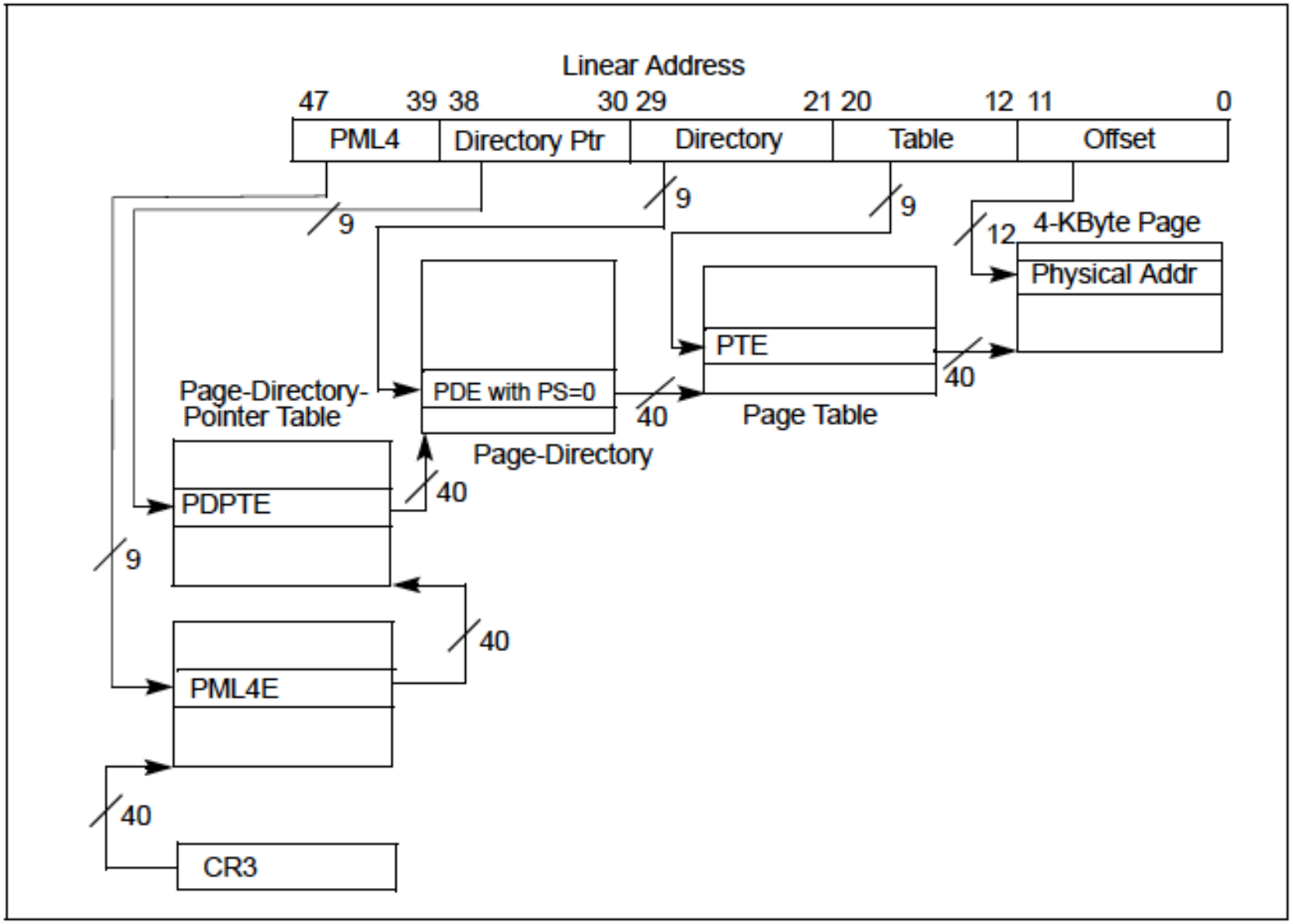


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

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: