CSC2/452 Computer Organization
Virtual Memory

Sreepathi Pai

URCS

October 30, 2019

Outline

Outline

Administrivia

- Assignment #3 is already out
  - Due Nov 8, 2019 at 7PM
- Homework #5 out today
  - Due Monday, Nov 11
  - I have printed copies with me today

Outline

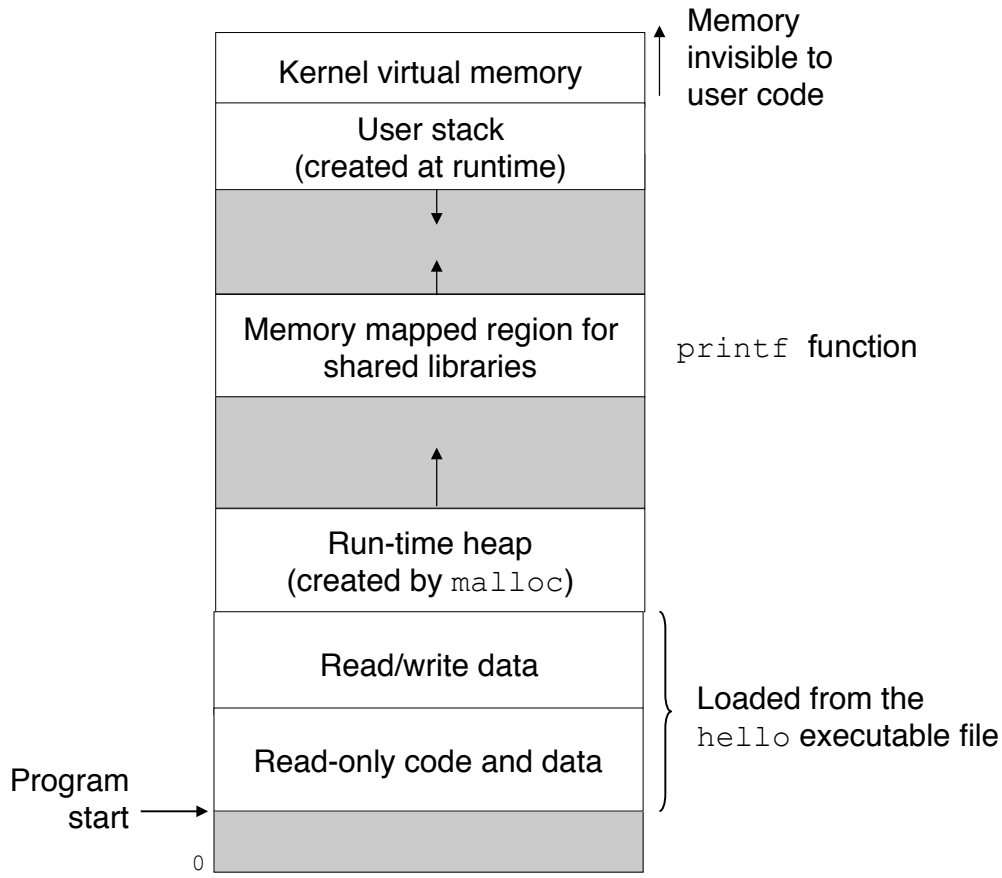
The OS and virtualization

- The operating system and CPU cooperate to perform virtualization
- CPU virtualization
  - Time sharing
- Memory virtualization
  - Today

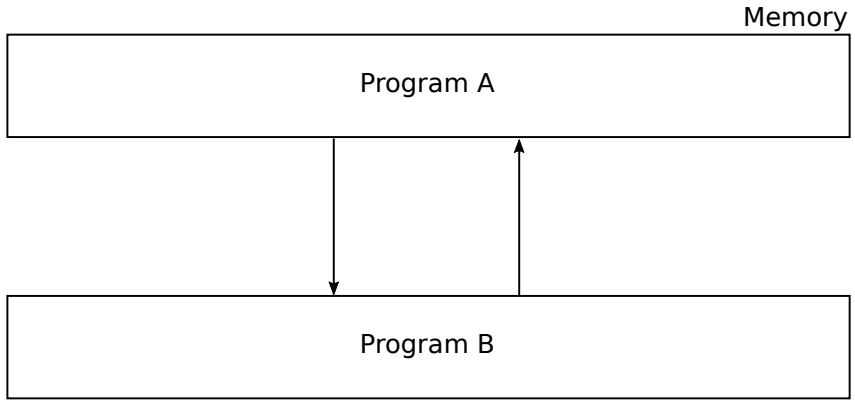
Outline

The High-Level Problem

How do you make every program believe it has access to the full RAM?



Time Sharing

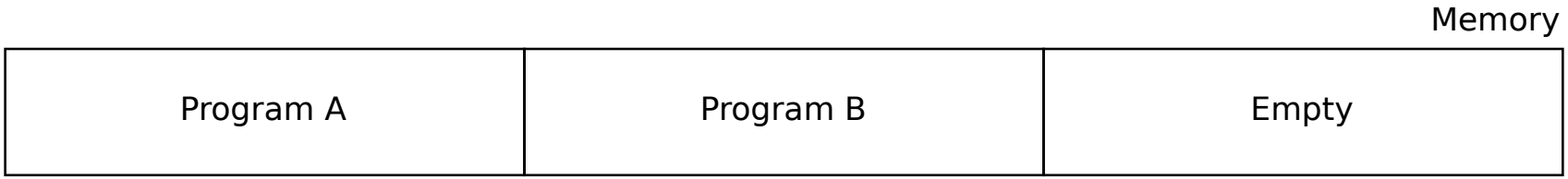


- Program A starts executing with full access to memory
- Timer interrupt
- All memory for Program A is copied to “swap area”
  - swap area could be hard disk, for example
- All memory for Program B is loaded from “swap area”
- Program B starts executing
- Repeat

The Problem with Time Sharing

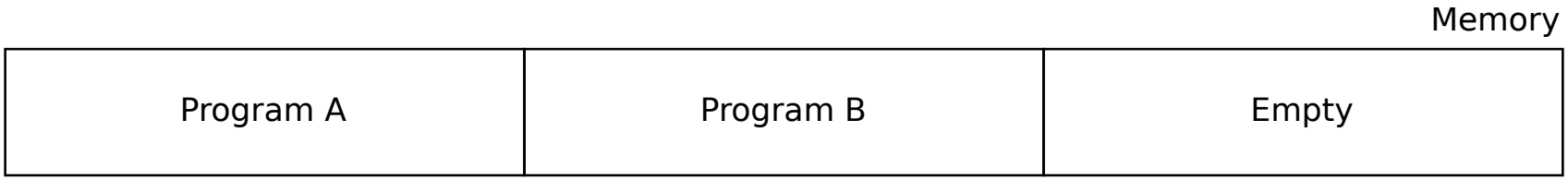
- My laptop has 8GB of RAM
  - Worst case save and restore data size
- My HDD writes about 500MB/s
- 16s to save full contents of memory
- 16s to load full contents of memory
- 32s to switch between programs

Space Sharing



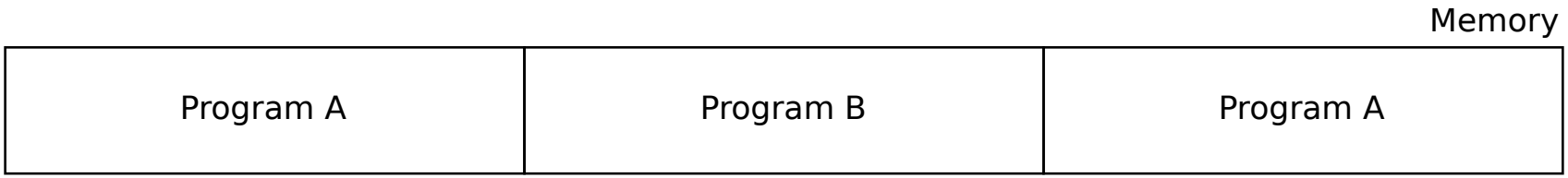
- Divide memory into portions
- Each program gets some portion of memory

The Problems with Space Sharing: #1



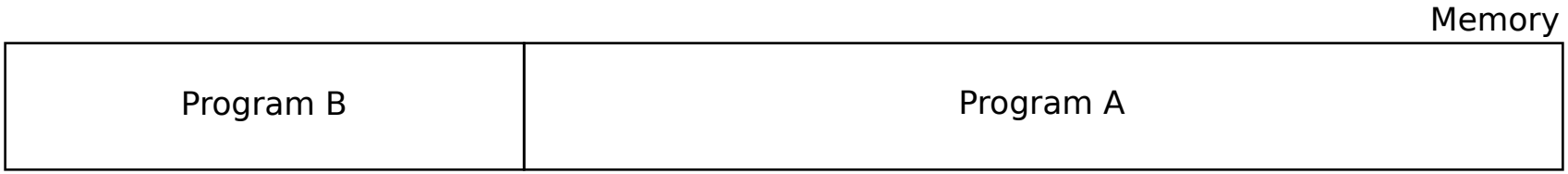
- How do we size each portion?
  - Fixed-size allocations waste space

Problem #2: Contiguous address space requirements



- Can't change size of allocations as programs are running
  - Atleast not easily
  - Need contiguous address spaces
- Think of an array that is bigger than each portion, but smaller than two portions combined

Problem #3: Can't move allocations



- Can't move allocations
  - Pointers in programs would need to be updated

Adding a Translation Layer

- Programs need to see one contiguous address space
  - We will call this the virtual address space
- We will translate from this virtual address space to actual physical address space
- Programs use virtual addresses, only the OS sees physical addresses

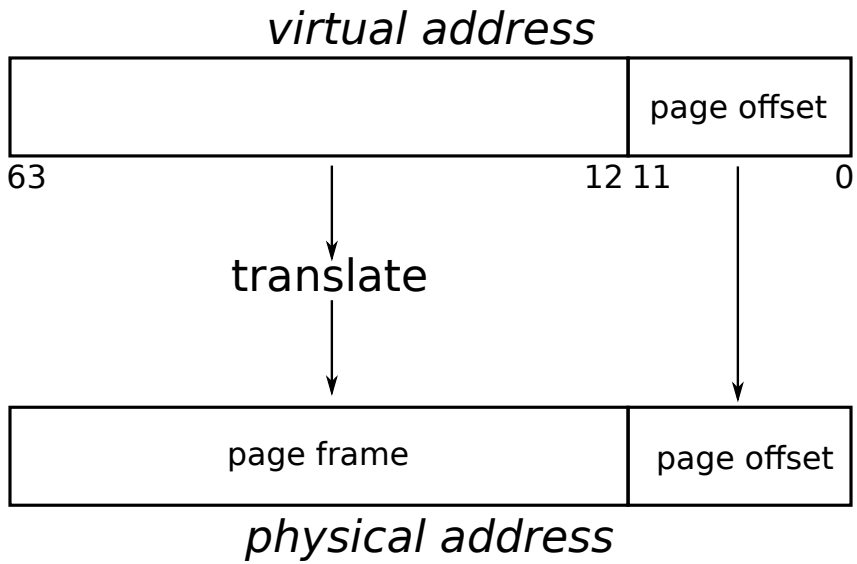
Virtual and Physical Addresses

- A virtual address and a physical address are "physically" indistinguishable
  - Both are 64-bit
  - However, virtual addresses span the whole 64-bit range
  - Physical addresses only span the actual amount of physical memory present
- All addresses used in programs are virtual
  - Except in very special cases when values of virtual addresses and its translated physical address are the same
  - Usually, when doing I/O with devices that don't understand virtual addresses

Translation Granularity

- We could translate any virtual address to any physical address
  - I.e. at byte level granularity
- But, it is more efficient to translate larger regions of memory
- Memory is divided into non-overlapping contiguous regions called *pages*
  - Most common page size is 4096 bytes (or 4KB)
  - But modern systems support large (or huge) pages (2MB or more)

The Memory Management Unit



- A load or store instruction uses virtual addresses
- The memory management unit (MMU) translates this virtual address to a physical address
  - Nearly everything "downstream" of the MMU sees physical addresses

Who maintains the translations?

- Although the CPU performs the translations, they are actually set up by the OS
- Page translations can change
  - Virtual address remains the same
  - Physical address changes
- This allows:
  - Allocation sizes to shrink and grow at page size granularity
  - Physical addresses can be non-contiguous

Swapping Pages Out

- The OS can mark virtual addresses as "not present"
  - The pages corresponding to these virtual addresses are not "mapped in"
  - Their contents may be on disk
  - No physical addresses are assigned to these virtual addresses
- Accessing these "swapped out" pages causes a page fault
  - CPU "suspends" processing of load/store instruction that caused fault
  - MMU notifies OS

Swapping Pages Back In

- When the OS receives a page fault notification, it can:
  - identify a page in physical memory
  - create a new mapping from the faulting virtual address to this page
  - load the contents of the newly mapped page from disk (if it was swapped out)
  - tell MMU that a new mapping has been set up
- CPU can then resume processing of load/store instruction

Summary

- Virtual memory uses a virtual address space
  - One, contiguous, linear address space
- Addresses are in the virtual address space are translated to physical addresses at page granularity
- Translations are setup by OS
- CPU MMU performs the translation on every load/store
- Virtual addresses can be marked as not present
  - Allows system to support allocating more physical memory than actually present!
  - CPU notifies OS whenever these addresses are accessed
- Programs do not notice these translations (except as loss in performance)

Outline

x86-64 VM Implementation

- Pages are 4KB (4096 bytes) in size
  - How many bits?
  - Also supports 2MB and 1GB pages, but we will not discuss these
- Uses a structure called a page table to maintain translations
  - Note, current implementations only use 48-bit virtual addresses
  - How many entries in page table?

x86-64 Page Table Design

- 12 bits for offset within page (4096 bytes)
- 36 bits remaining
  - 16 bits not used in current x86-64 implementations
- Page table will contain 2^{36} entries
  - Each program will require 8×2^{36} bytes for its page table
  - How much is this?

Space requirements for the page table

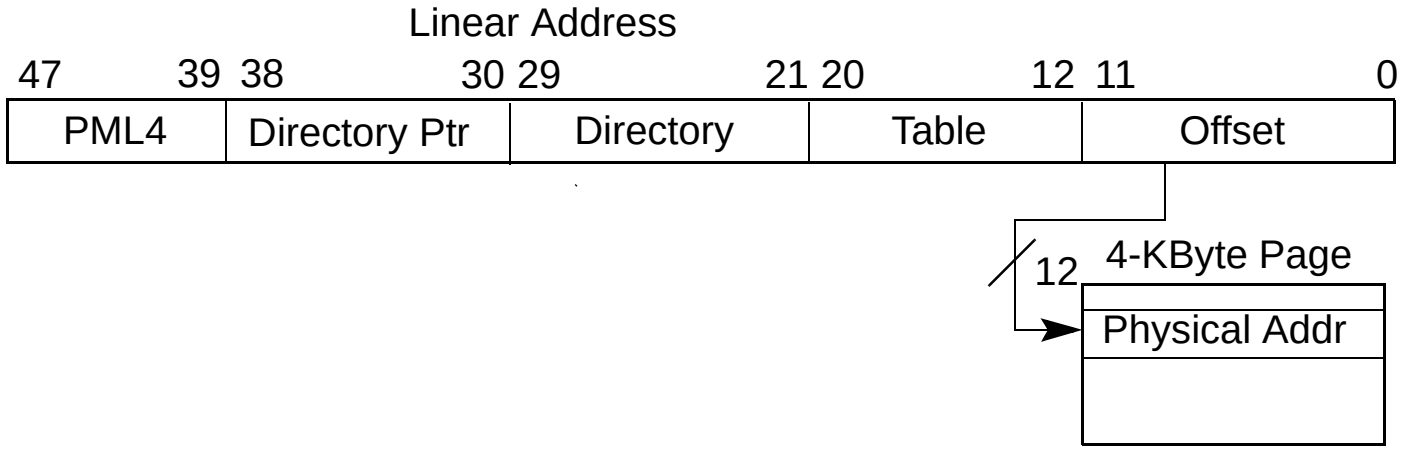
512GB

- Ideally, we only need to store as many translations as there are physical pages
  - e.g., if 8GB physical RAM, then 2097152 pages, so 16MB for page table entries
  - Called an *inverted page table* design
- Not used by x86-64

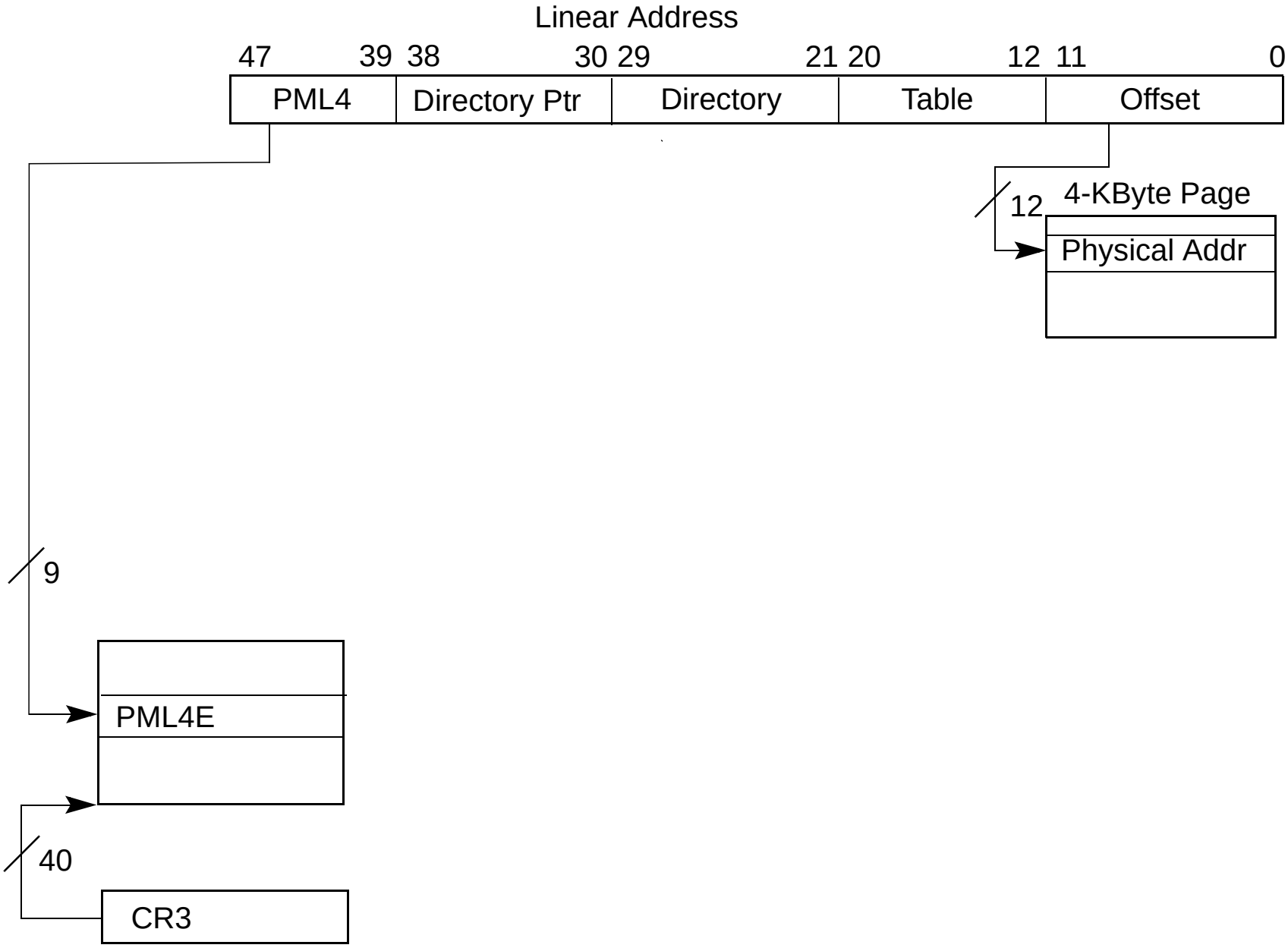
Hierarchical Page Tables

- Instead of a single page table, multi-level page tables are used
- On the x86-64, each level contains 512 entries
  - How many bits required to index into each level?
  - How many levels (given we have 36 bits)?
- Each entry is 64-bits wide
  - Total size of each level?
- NOTE: (updated after class), x86-64 supports 52-bits in physical addresses

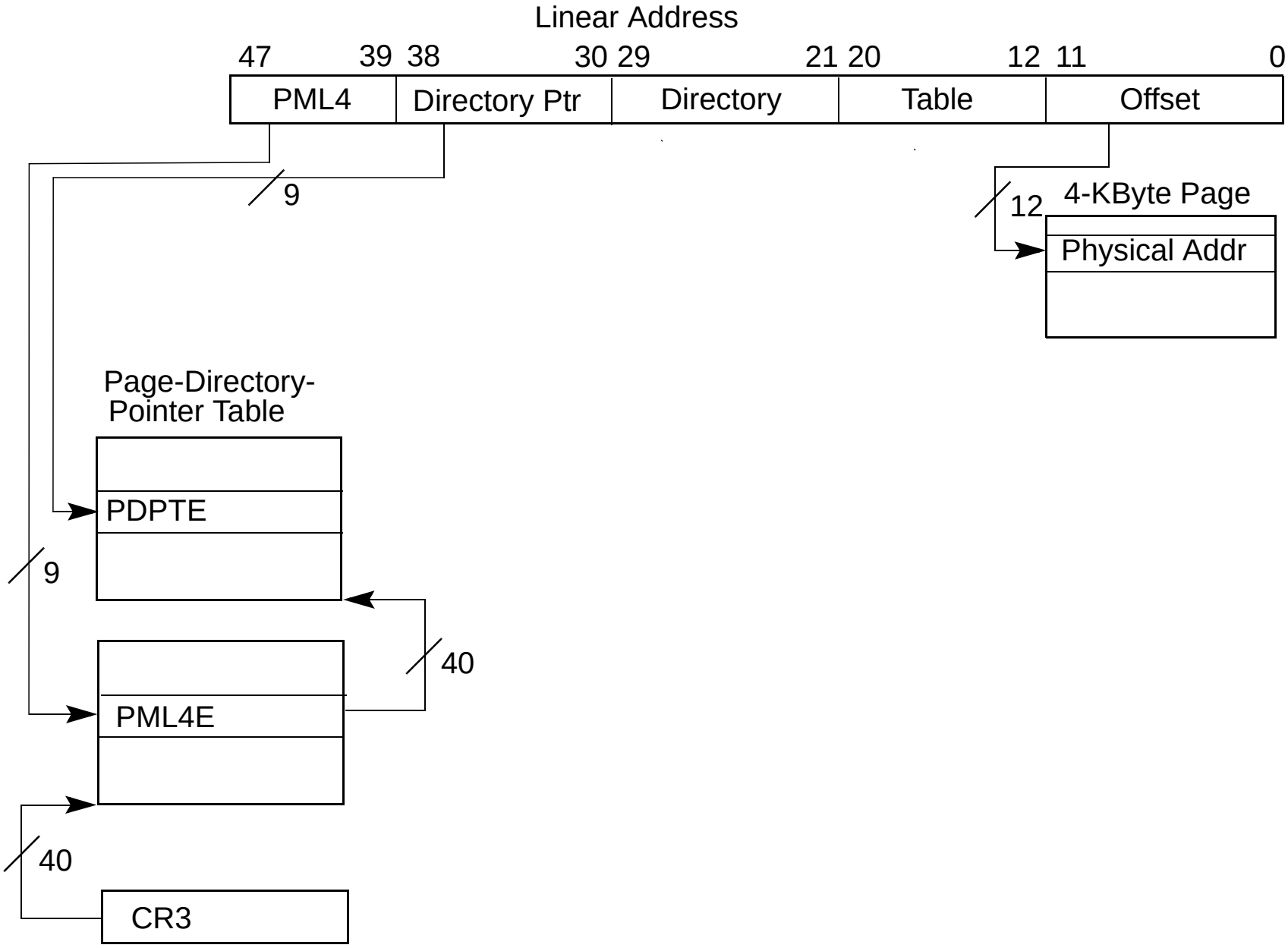
Translation: Goal



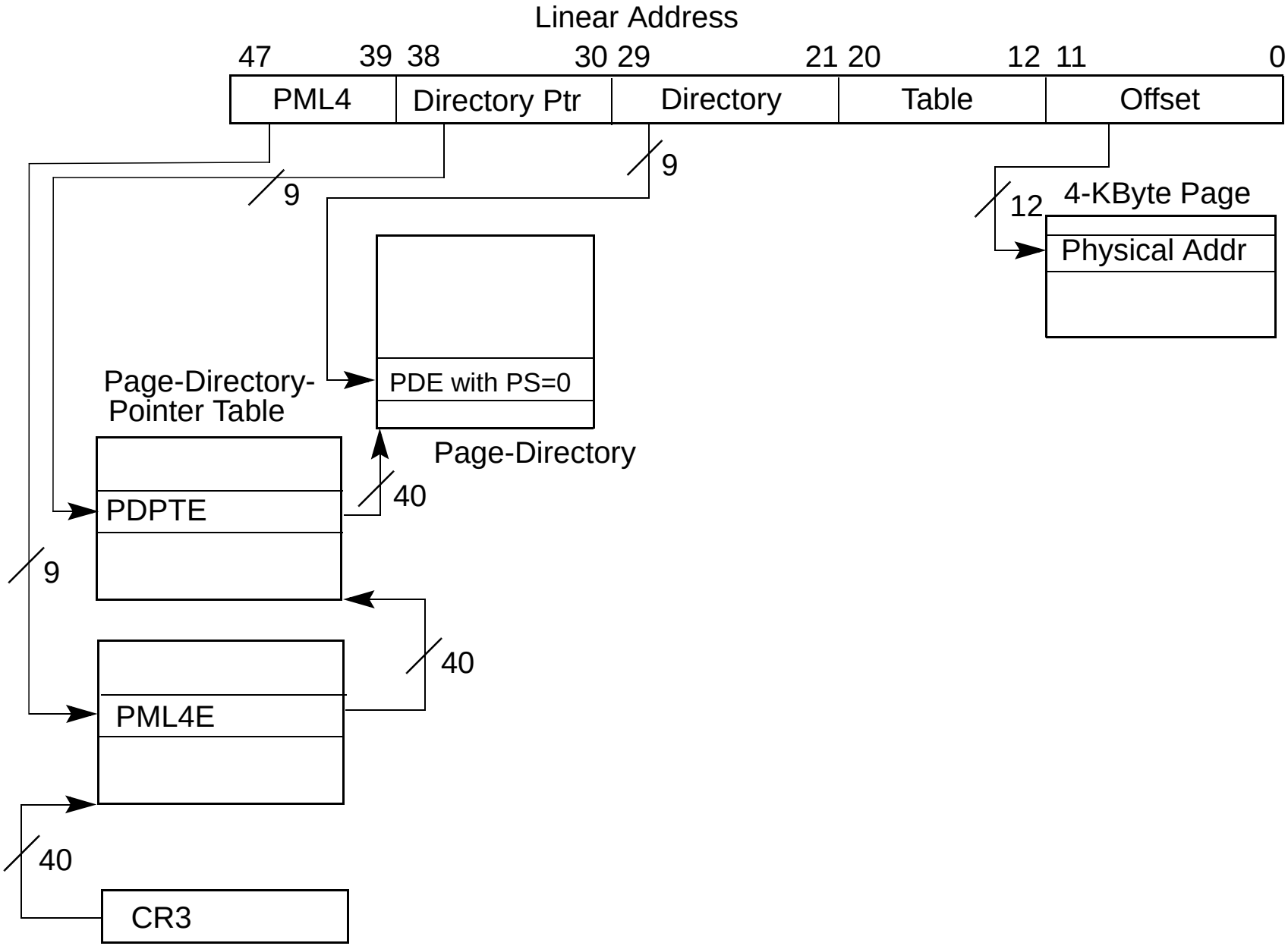
Translation: First level



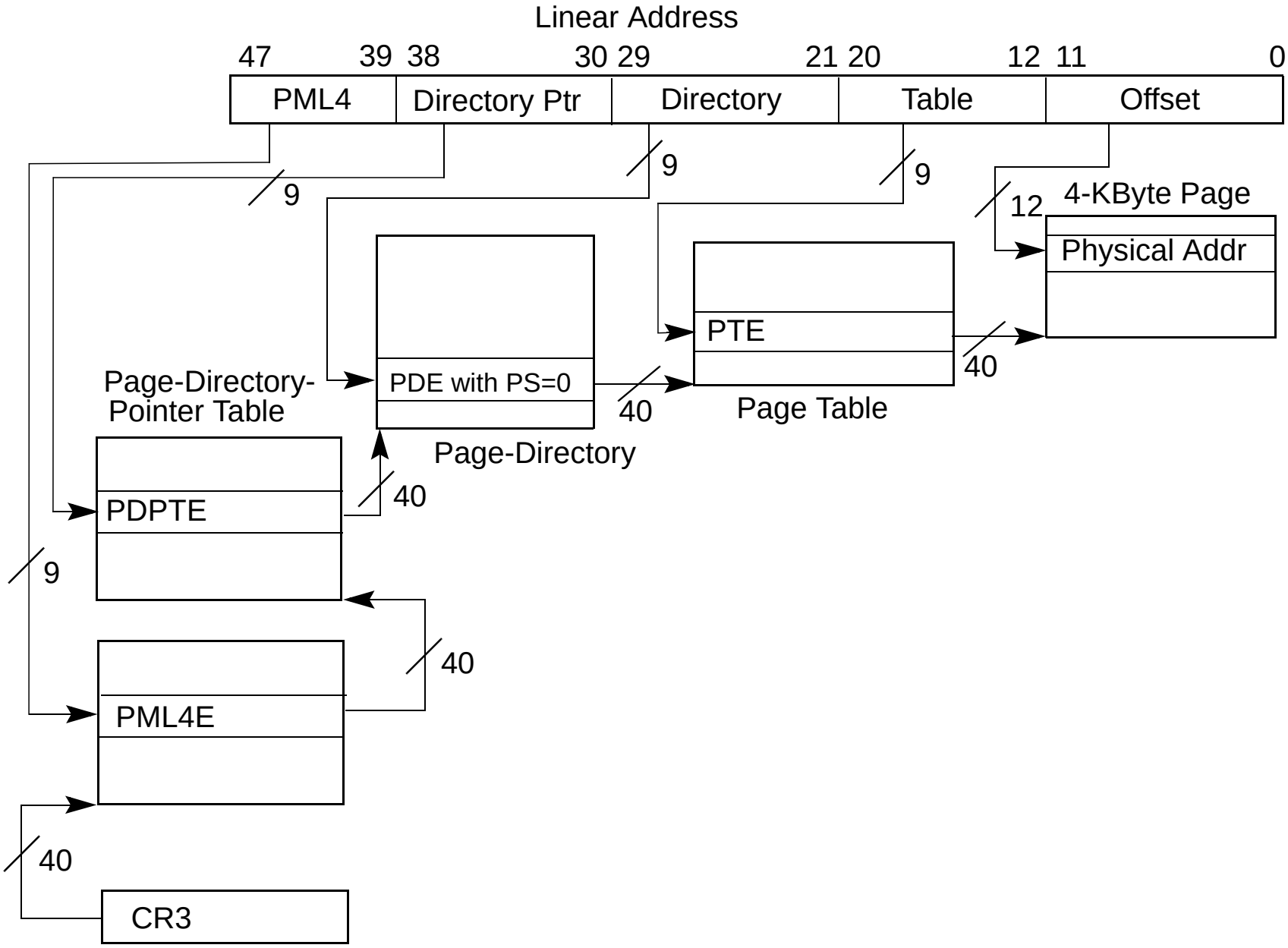
Translation: Second level



Translation: Third level



Translation: Fourth level



Space requirements for multi-level page tables

- Each level contains 512 8-byte entries containing physical addresses
  - 4096 bytes
- A minimal program could get away with 4096*4 bytes for the page tables
  - No need for 512GB or even 16MB
- Note some of these levels can be "paged out"
  - I.e. each entry in these tables contains a present bit

Translation Overheads

- Translating one memory address requires reading 4 other addresses!
  - This is called a “page table walk”, performed by the MMU
- Can we avoid reading the page table on every read access?

The Translation Look-aside Buffer (TLB)

- The TLB is a small cache used by the MMU
  - Usually fewer than 10 entries, fully associative
- It caches the contents of final translation
  - Must be invalidated whenever the translation changes (the `invlpg` instruction)
- MMU checks TLB if it contains translation
  - If it does, no page table walk is performed

Entry formats

Bit positions: 63 62 61 60 59 58 57 56 55 54 53 52 51 … M[1] M-1 … 32 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

Reserved[2]	Address of PML4 table	Ignored	PCD	PWT	Ign.												CR3
XD[3]	Ignored	Rsvd.	Address of page-directory-pointer table	Ign.	Rsvd	Ign	A	PCD	PWT	U/S	R/W	1					PML4E: present
Ignored	0																PML4E: not present
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 1GB page frame	Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE: 1GB page
XD	Ignored	Rsvd.	Address of page directory	Ign.	0	Ign	A	PCD	PWT	U/S	R/W	1					PDPTE: page directory
Ignored	0																PDTPE: not present
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 2MB page frame	Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Ignored	Rsvd.	Address of page table	Ign.	0	Ign	A	PCD	PWT	U/S	R/W	1					PDE: page table
Ignored	0																PDE: not present
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1			PTE: 4KB page
Ignored	0																PTE: not present

Page Permissions

- U/S: User/Supervisor
  - if this bit is 0, a user-space process cannot read/write this page
  - this is how the kernel protects itself from user programs (among other "defenses")
- R/W: Read/Write
  - if this bit is 0, this page cannot be written
  - e.g. pages containing code or read-only data
- XD: eXecute Disable (Intel terminology)
  - if this bit is 1 and the machine supports XD, then instructions cannot be fetched from this page

Attempting to perform forbidden actions causes the CPU to generate a fault.

How Caches Change with Virtual Memory

- Should you use virtual addresses to index the cache?
  - Should you do this at all levels?
- Which address should the tag be constructed from?
  - Virtual or physical?

How Prefetchers Change with Virtual Memory

- Recall, a prefetcher fetches data before it is required
- What happens if a prefetcher tries to fetch data from a page that is swapped out?

Outline

OS Implementations of Virtual Memory

- `mmap`
- `mprotect`

References and Acknowledgements

- Chapter 9
- Acknowledgements
  - Figure of memory layout from the textbook
  - Figures of page tables and page table entries from Intel Software Developers Manual, Vol 3, System Programming Guide