CSC290/420 ML Systems for Efficient AI
Virtual Memory

Sreepathi Pai

September 23, 2025

URCS

Outline

Cache Coherence

Memory Virtualization

x86-64 Implementation of Virtual Memory

Outline

Multiple Processors and Cores

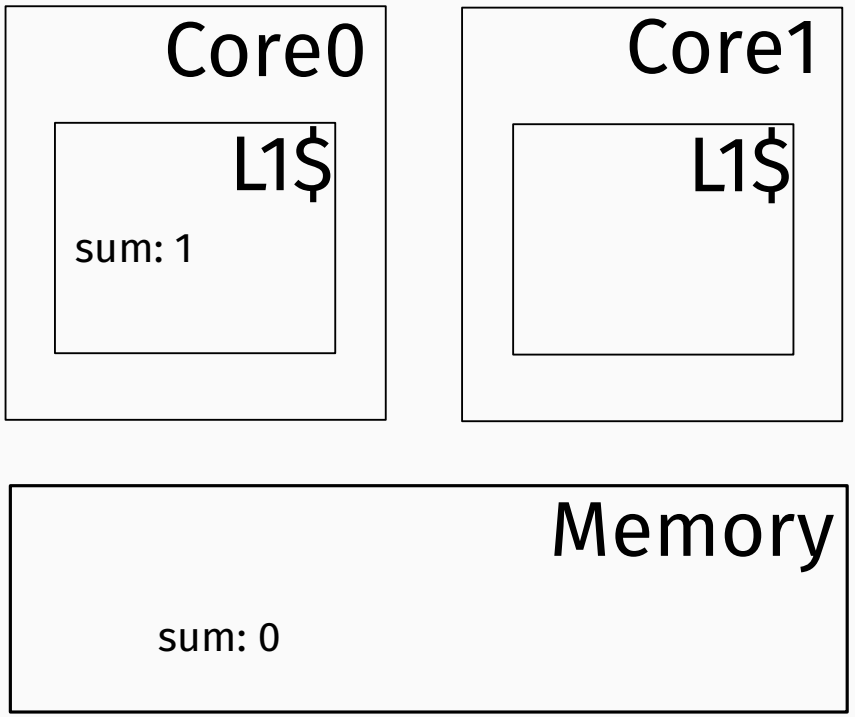
- You can run concurrent code on a system with 1 processor
  - Thanks to time sharing
- But most computers have multiple cores today
  - Each core is an independent computational unit
- Systems can also have multiple processors
  - Each processor contains multiple cores
  - Rare in consumer-grade systems

Mapping Processes and Threads to Cores

- The OS scheduler maps processes and threads to cores
- It is possible to "pin" threads/processes to certain cores
  - Avoids scheduling overhead
  - Can improve performance in some situations
- On Linux, the `sched_setaffinity` function allows you to set thread affinities
  - Can also use the `pthread_setaffinity_np` function

Cache Coherence

- Recall that caches contain copies of data variables
  - This is fine when only one process/thread is accessing the data
- What happens when different threads access shared data?
  - Core 1 has shared variable `sum` in its cache
  - Will Core 2 try to get `sum` from memory?



Cache Coherence

- Cache coherence is a hardware mechanism to locate copies of a piece of data and use the "latest" version
  - Usually, the last written version
- Core 2 will send a request for `sum`
  - Core 1 will reply to that request
  - RAM may also reply, but Core 1 has more recent version and will be used by Core 2
- Coherence protocols also prevent multiple cores from writing to the same piece of data

The MESI Cache Coherence Protocol

- Every cache block (or cache line) is in one of four states:
  - Modified (M), this line contains updates
  - Exclusive (E), this line is owned by this core and is identical to RAM
  - Shared (S), multiple identical copies of this line exist
  - Invalid (I), this line does not contain any data
- Real processors use slight variants of the MESI protocol
  - MOESI, etc.

How the MESI protocol works

- Only lines in Exclusive state can be written to by a core
  - achieved by invalidating all other copies of the line in other caches
  - then reading the latest copy from RAM
  - a line read from RAM is loaded into "Exclusive" state
- A line that is written moves to Modified state
  - when the line is written to RAM, it moves to Shared state
  - usually when another core wants to read the line
- All copies of a line in Shared state match RAM contents

Cache Line Bouncing

```
// global shared variable
uint32_t counter = 0;

// called by different threads
void inc_counter() {
     // this is an atomic read-modify-write, not a plain write
     counters++;
}
```

- What happens when thread 0 calls `inc_counter`?
- What happens when thread 1 calls `inc_counter`?
- What happens when thread 2 calls `inc_counter`?

False Sharing

```
uint32_t counters[NTHREADS]; // each thread gets its own counter

void inc_counter() {
     // atomic RMW
     counters[mythreadid]++;
}
```

- What is the size of `counters`?
- How many cache lines does it occupy if each cache line is 32 bytes?
- What happens when thread 0 writes to location `counters[0]` and thread 1 writes to location `counters[1]`?

Cache Coherence Domain

- Data is kept coherent within a cache coherence domain
- Traditionally, only the CPU's caches
- Then, extended to other devices
- Now many machines support cache coherence across CPUs and GPUs
    - Notably, the newer Macs.

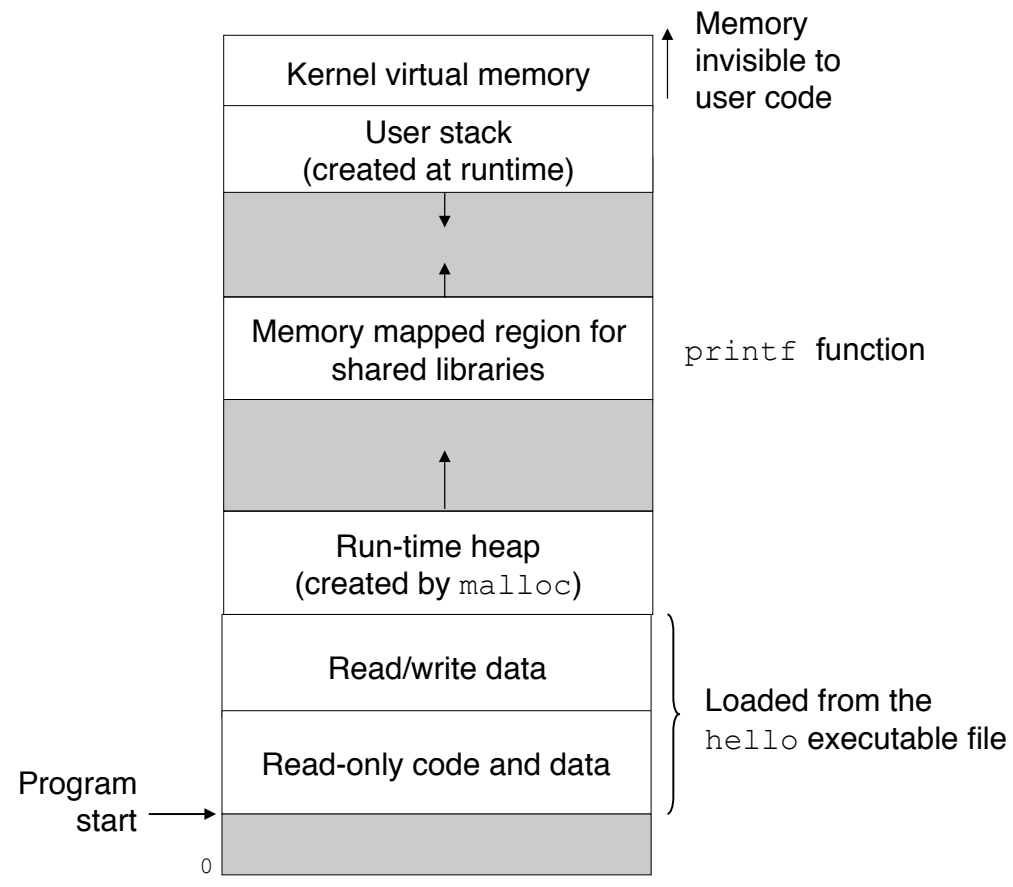
Outline

Virtualization

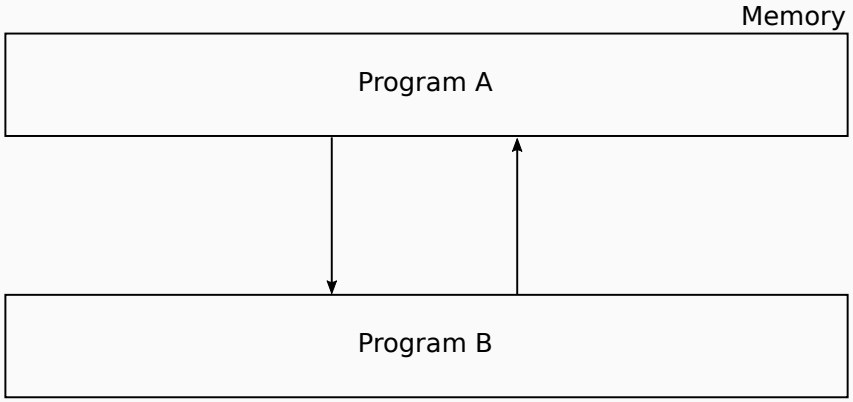
- Virtualization decouples physical resources from logical resources
  - Physical CPUs vs Virtual CPUs
  - Physical Memory vs Virtual Memory
  - Physical Computers vs Virtual Computers
- The operating system and CPU cooperate to perform virtualization
- CPU virtualization
  - Time sharing
- Memory virtualization
  - Today
- Whole computer virtualization
  - Not in this class, but basis of cloud computing

The High-Level Problem

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



Time Sharing

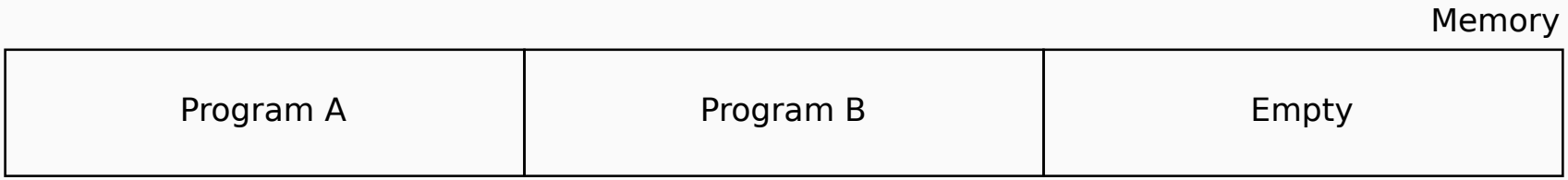


- Program A starts executing with full access to memory
- Timer interrupt
- All memory for Program A is copied to “swap area”
  - swap area could be hard disk, for example
- All memory for Program B is loaded from “swap area”
- Program B starts executing
- Repeat

The Problem with Time Sharing

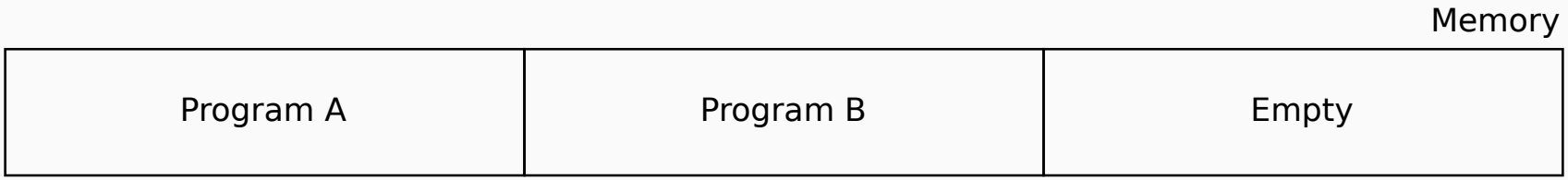
- My laptop has 8GB of RAM
  - Worst case save and restore data size
- My HDD writes about 500MB/s
- 16s to save full contents of memory
- 16s to load full contents of memory
- 32s to switch between programs

Space Sharing



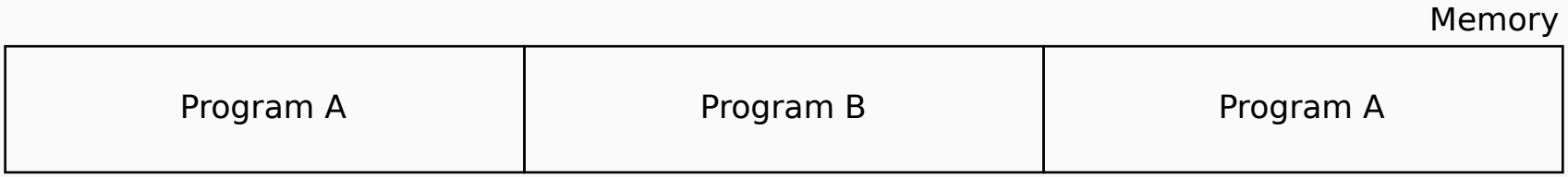
- Divide memory into portions
- Each program gets some portion of memory

The Problems with Space Sharing: #1



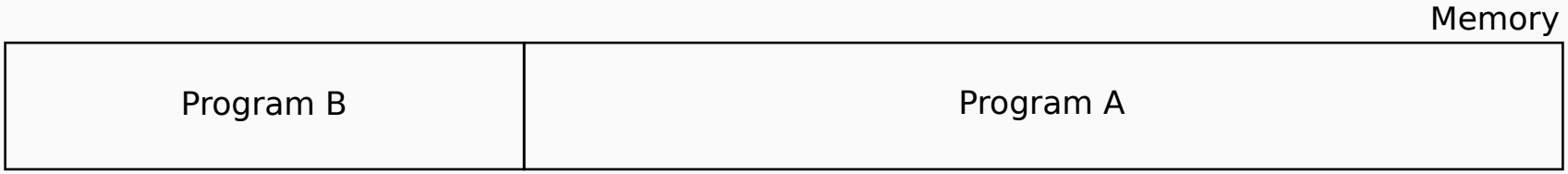
- How do we size each portion?
  - Fixed-size allocations waste space
  - Known as the "trapped-capacity" problem

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

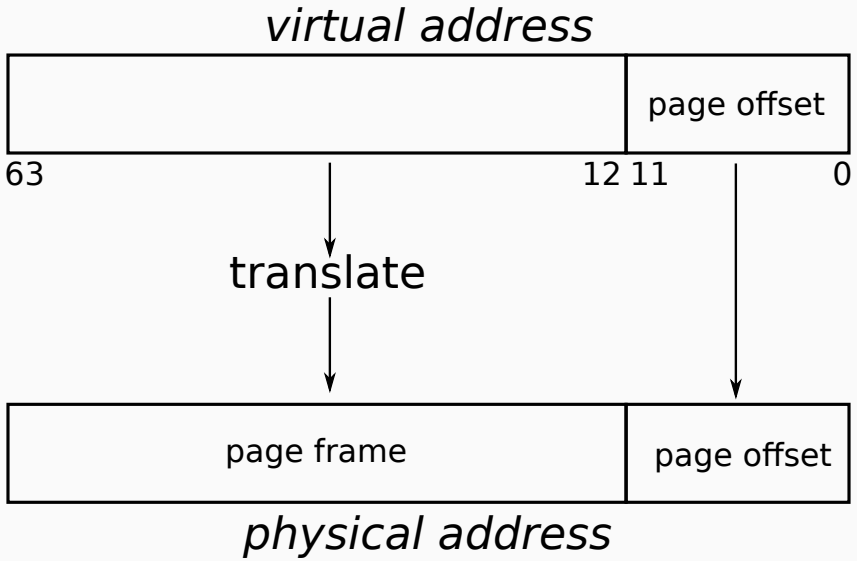
Virtual and Physical Addresses

- 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
  - (Devices like this are increasingly uncommon)

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 new M1 Macs have a 16384 byte page size

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 to 52-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 (if using 48-bit virtual addresses)
  - 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

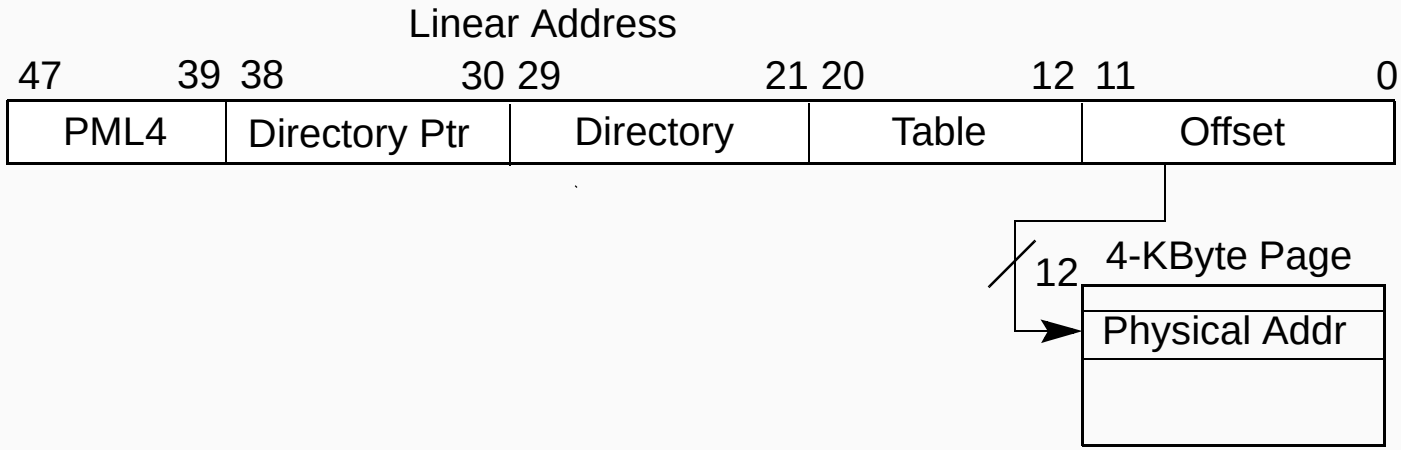
512GB

- 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

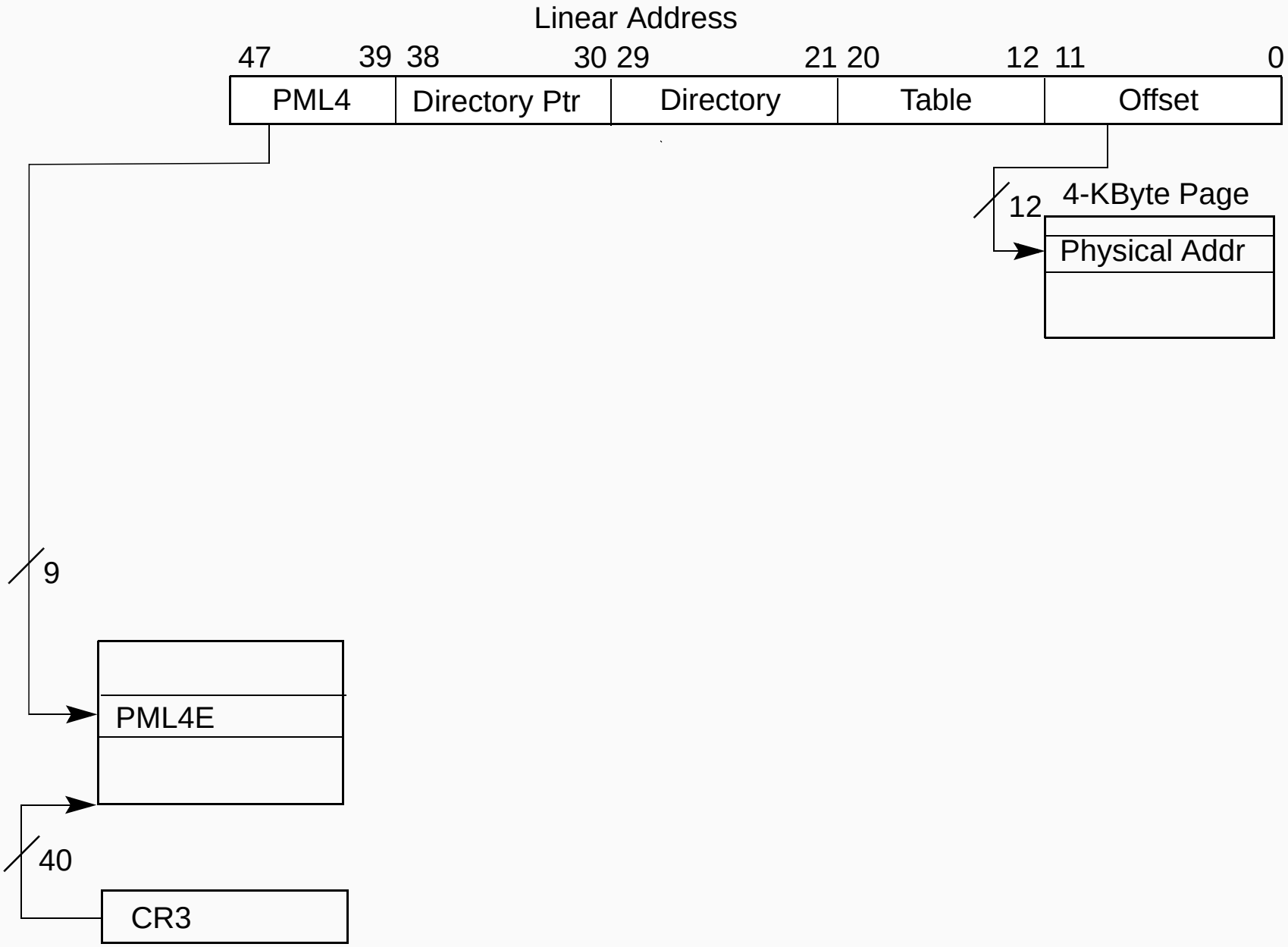
Hierarchical Page Tables

- 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: x86-64 supports 52-bits in physical addresses

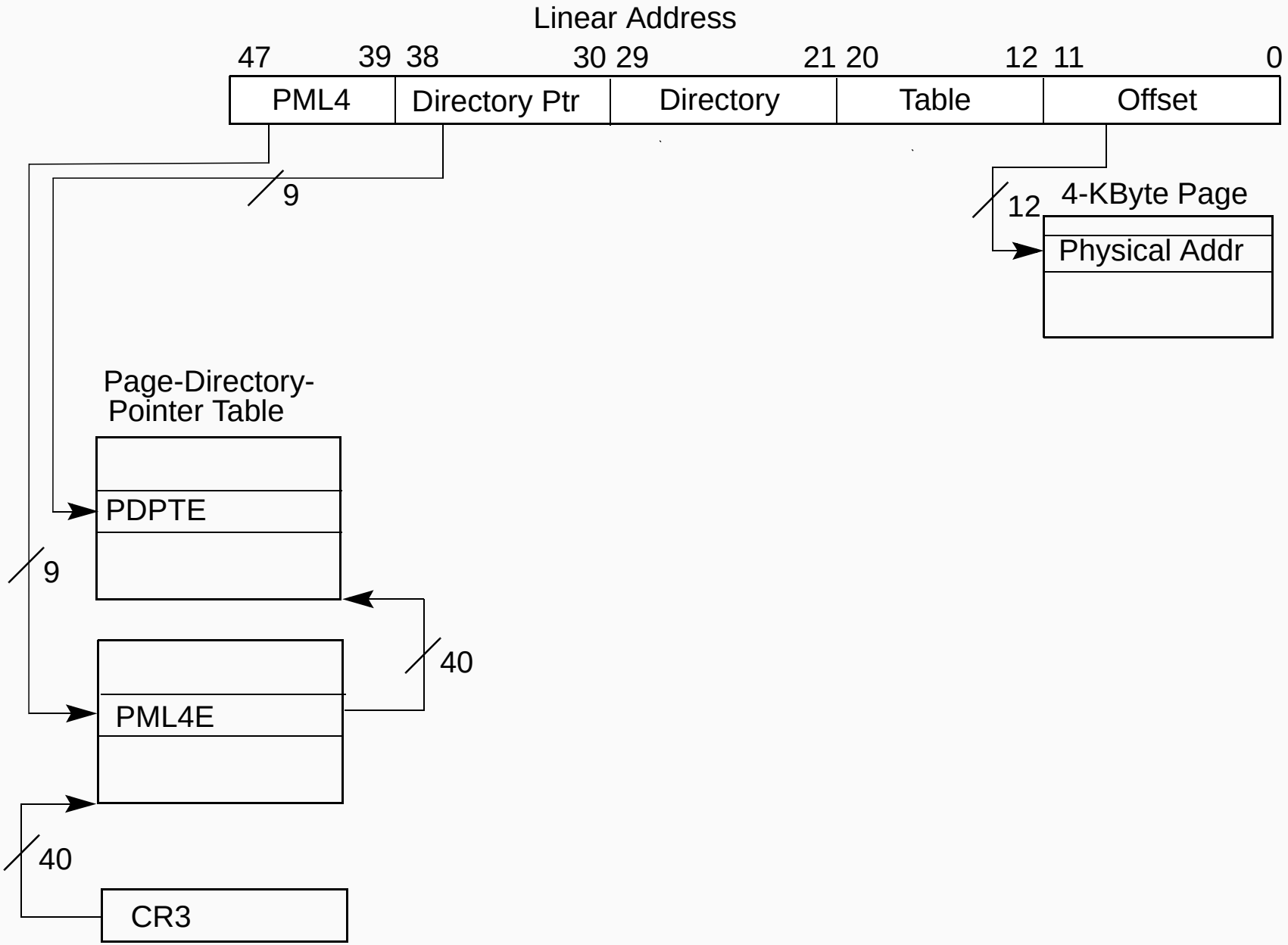
Translation: Goal



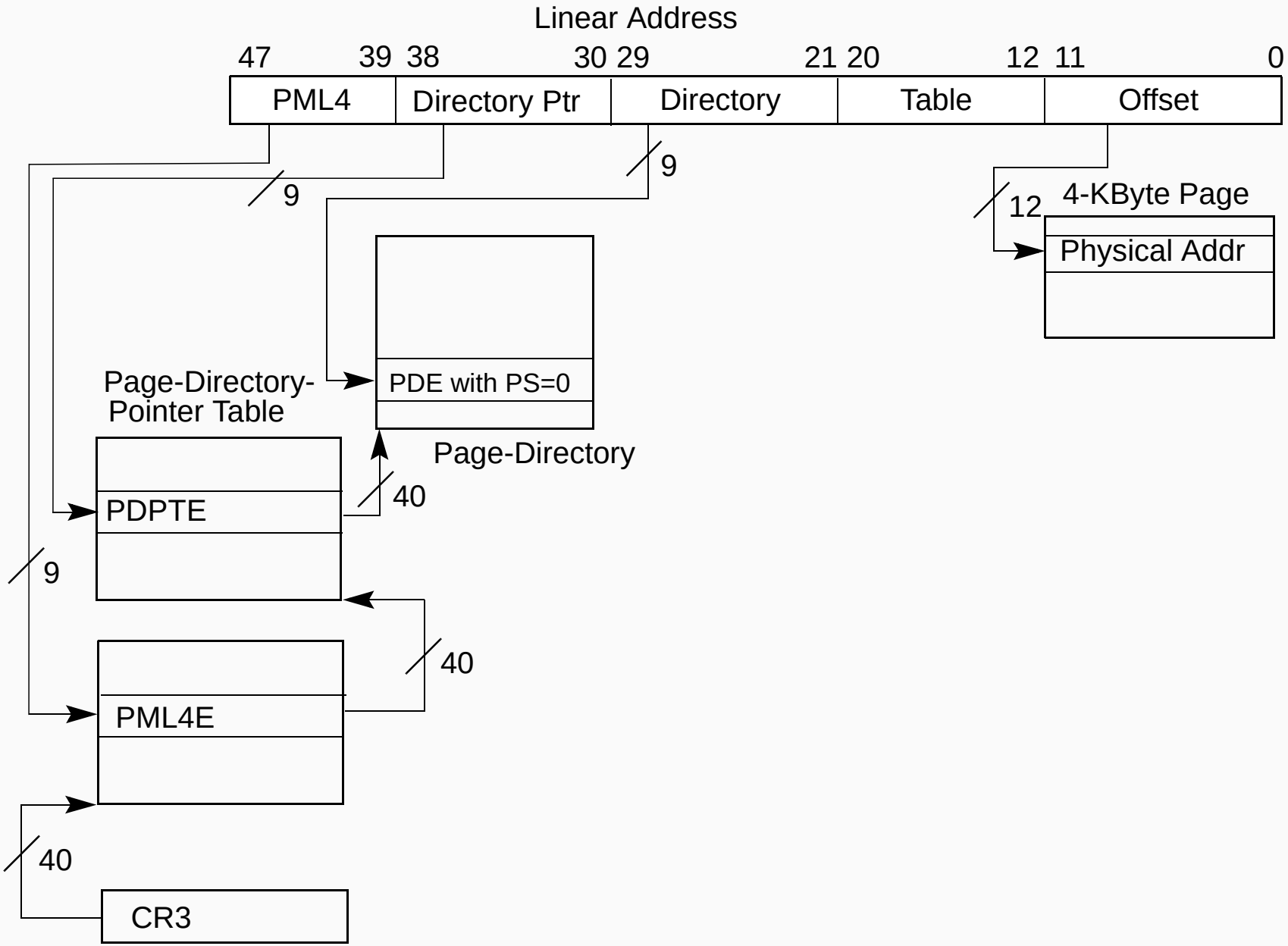
Translation: First level



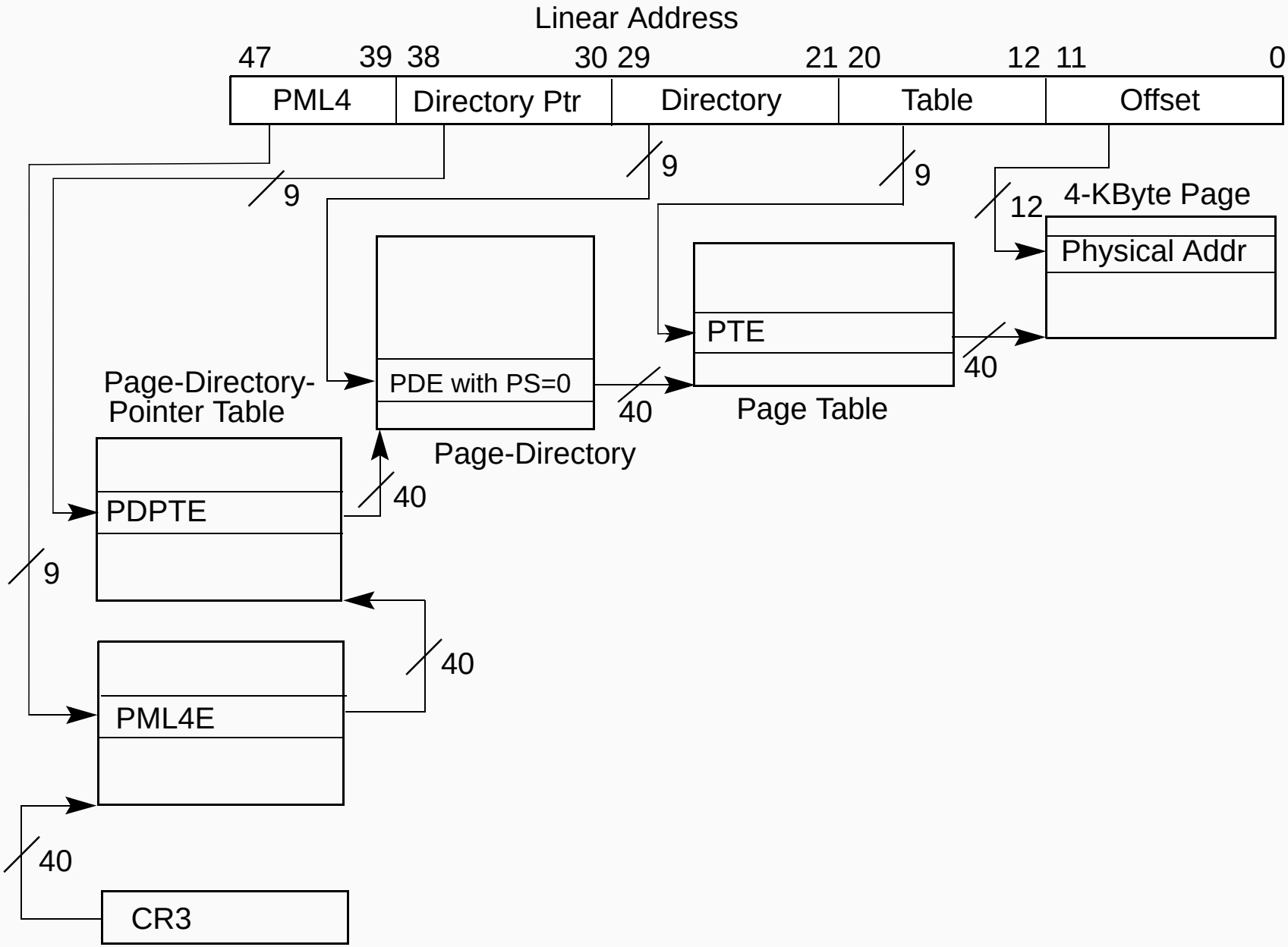
Translation: Second level



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
  - Correction: on Intel's Golden Cove, this is 96 entries, 4-way set 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

TLB Misses

- Very large data structures can cause excessive TLB misses
- At least one very fast Matrix Multiply routine was specifically optimized to minimize TLB misses
    - GotoBLAS

Thrashing

- The Working Set is the set of pages a program uses
- If the Working Set size is greater than physical memory, some pages will be swapped out
- On a page fault, the page will be brought in from disk, displacing an existing page
- In certain cases, the pages that were swapped out might be referenced immediately
- The program gets stuck just swapping pages in and out
  - "Thrashing" [its working set]

How Caches Change with Virtual Memory

- Should you use virtual addresses to index the cache?
  - Should you do this at all levels?
- Caches contain a tag (a part of the full address)
  - since multiple addresses can be mapped to the same cache set
- Which address should the tag be constructed from?
  - Virtual or physical?

Acknowledgements

- Acknowledgements
  - Figure of memory layout from the Computer Systems: A Programmer's Perspective
  - Figures of page tables and page table entries from Intel Software Developers Manual, Vol 3, System Programming Guide