CPSC313: Computer Hardware and Operating Systems

Unit 5: Virtual Memory

Virtual Memory: The x86

Administrivia

- Quiz 5 coming soon!
- Lab 10 due Sunday
- Exam period office hours schedule coming soon (or already posted!)

Check Canvas/Piazza for details!

Today

- Roadmap:
  - So far, we've described page tables as huge arrays, cached in the TLB -- let's figure out why representing tables that way could be a problem.
  - We'll solve that problem and examine how the x86 represents page tables.
- Learning Outcomes
  - Describe the similarities and differences between a multi-level index for a file and multi-level page tables.
  - Describe the x86 Virtual Memory architecture
  - Generalize from the x86 to other configurations of virtual memory systems.

Things you know so far:

Virtual pages and physical pages are the same size.

- The page size is set by the hardware (CPU).
- The (maximum) size of the virtual address space is also determined by the hardware (CPU).
- Therefore, the number of bits or size of a virtual page number is also determined by the CPU.

Things you know so far:

- The maximum number of physical pages and therefore the size of physical page numbers is also determined by the hardware (CPU).
- The actual number of physical pages is specific to a particular machine (not only determined by its CPU).
- Virtual page numbers and physical page numbers can be different sizes.

Pre-class video implementation: one flat array

- There is one page-table per process.
- The OS must track page tables of all running processes.
- So, how big would those page tables be?
- Assume Intel x86-64
  - 4 KB pages
  - 48-bit address space
  - How many entries would we need in our page table?

Pre-class video implementation: one flat array

- Assume Intel x86-64
  - 4 KB pages
  - 48-bit address space
  - How many entries would we need in our page table?
    - 48 (bits in address space) – 12 bits (per 4 KB page) = 36 bits
    - 36 bits = 2^{36} = 64 G pages; that's more than 64 billion pages
    - We need 1 PTE per page; if each PTE is 8 bytes, **that's 512 GB per page table! Per process!**

Observations about Address Spaces

- What feature of address spaces might enable a better solution than a huge, VPN-indexed table?

Observations about Address Spaces

- What feature of address spaces might enable a better solution than a huge, VPN-indexed table?
    - Most processes won't use all or even most of their address space.
    - Many only use a few pages! Consider the hello program:
        - It consumes only 1 page each for text, data, stack
        - BUT: it also needs virtual address space for:
            - Shared libraries (code and data)
        - Still: total number of pages is measured in "a few hundred."
    - **Page Tables are Sparse (i.e., most entries are invalid)**

How do we represent a sparse address space?

- Requirements:
  - **Simple** enough data structure to work with in hardware (on the x86).
  - **Efficient** for a large address space.
  - Supports gaps in the address space: **sparse address spaces.**
- Insight:
  - This is similar to the problem we had representing **files**: it had to be efficient for both small and large files and for sparse files.
  - *We were able to ignore whole chunks of the logical block number space by having NULL (invalid daddr_t) values for indirect block addresses.*
  - Maybe we could do something similar?

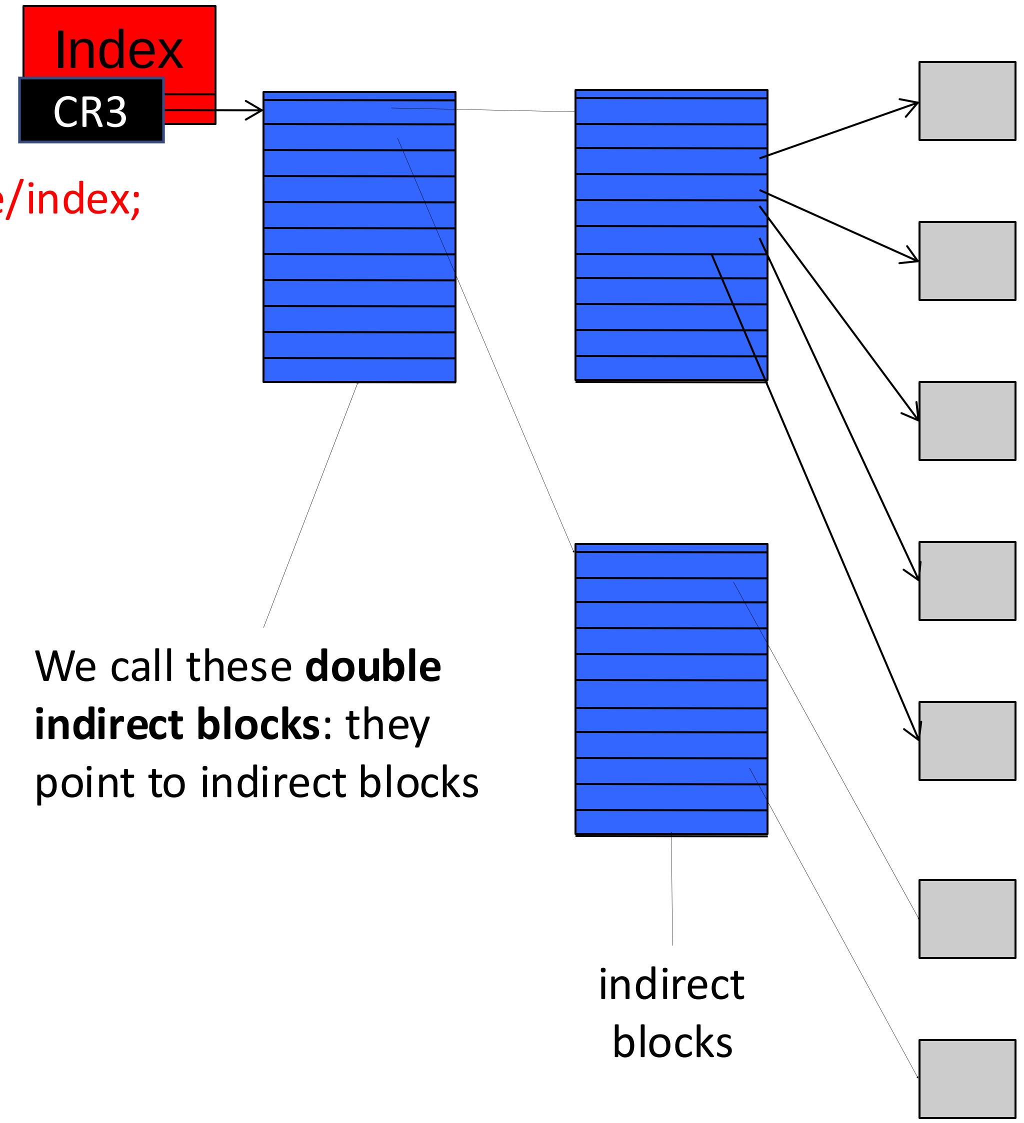
When implementing in HW

- Guiding principles:
  - Using bit values directly is good, cheap, simple
  - Computing things is relatively slower and more expensive

When implementing in HW

- Guiding principles:
  - Using bit values directly is good, cheap, simple
  - Computing things is relatively slower and more expensive
- We are essentially going to use a multi-level index, but we are going to choose the sizes of those indexes to match the hardware.
- In particular:
  - All blocks in the tree to match our virtual memory page size (4 KB)
  - We use specific bits in an address to index into each level of the tree

From a file index to a page table!



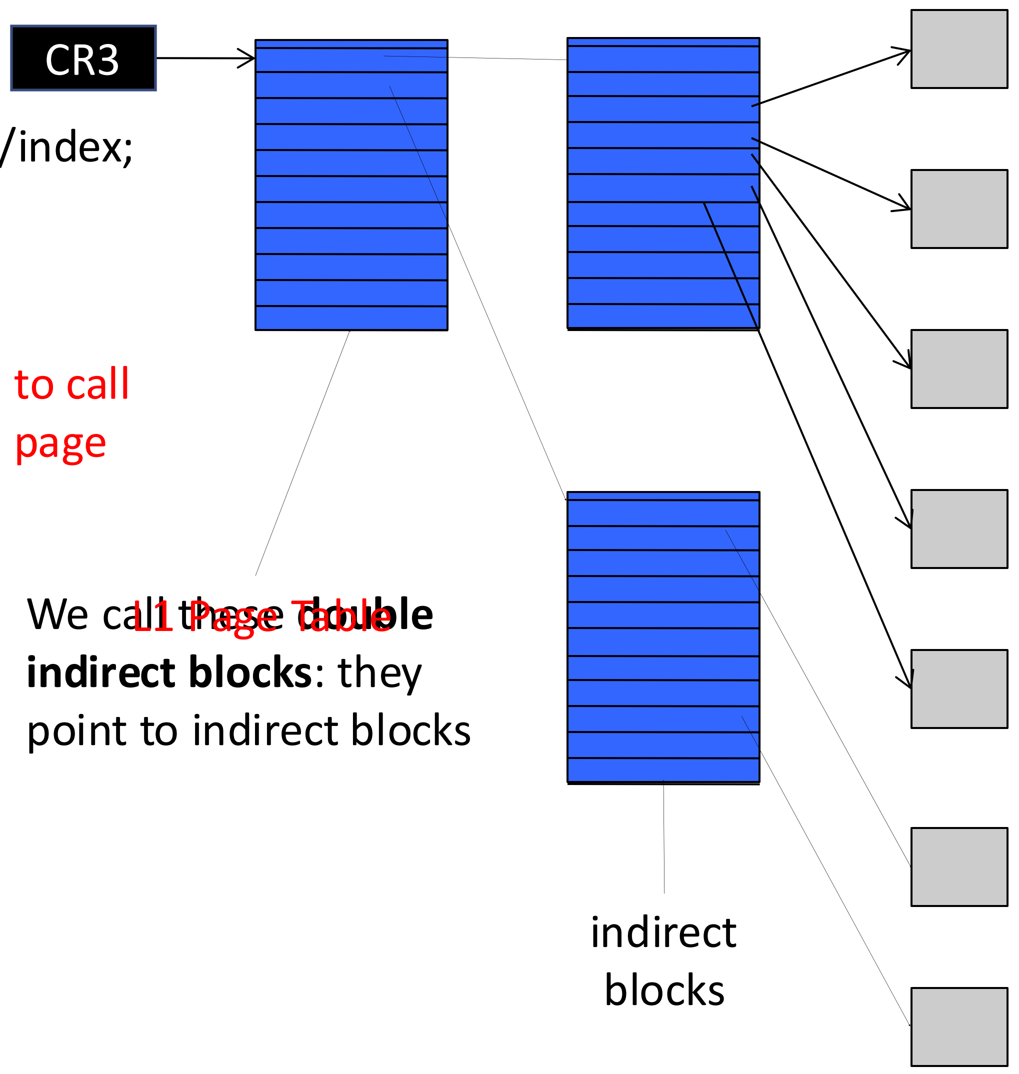
1. Instead of an inode/index; we have a hardware register: CR3

We call these **double indirect blocks**: they point to indirect blocks

From a file index to a page table!

1. Instead of an inode/index; we have a hardware register: CR3

2. Everything we used to call indirect blocks we call page tables.



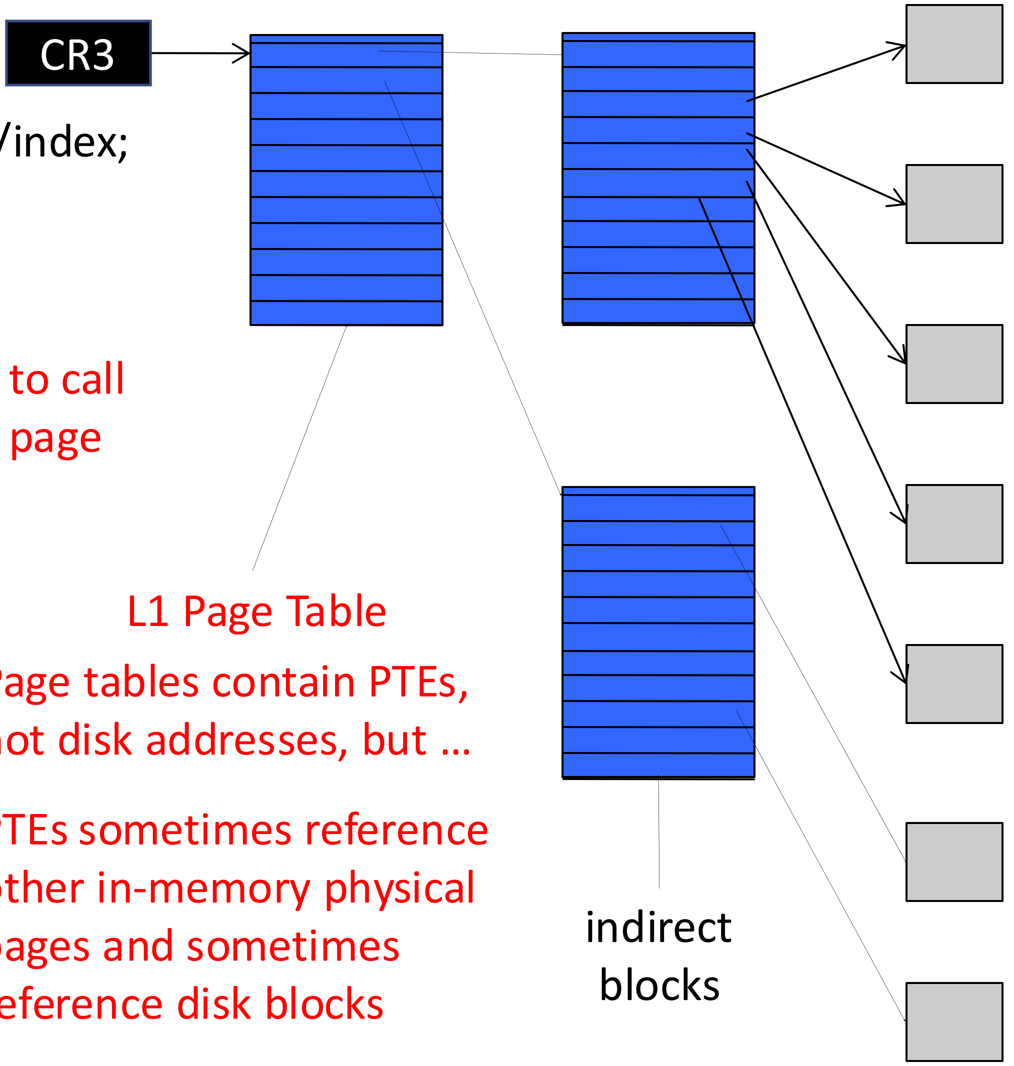
We call these **double indirect blocks**: they point to indirect blocks

L1 Page Table

From a file index to a page table!

1. Instead of an inode/index; we have a hardware register: CR3

2. Everything we used to call indirect blocks we call page tables.



Page tables contain PTEs, not disk addresses, but …

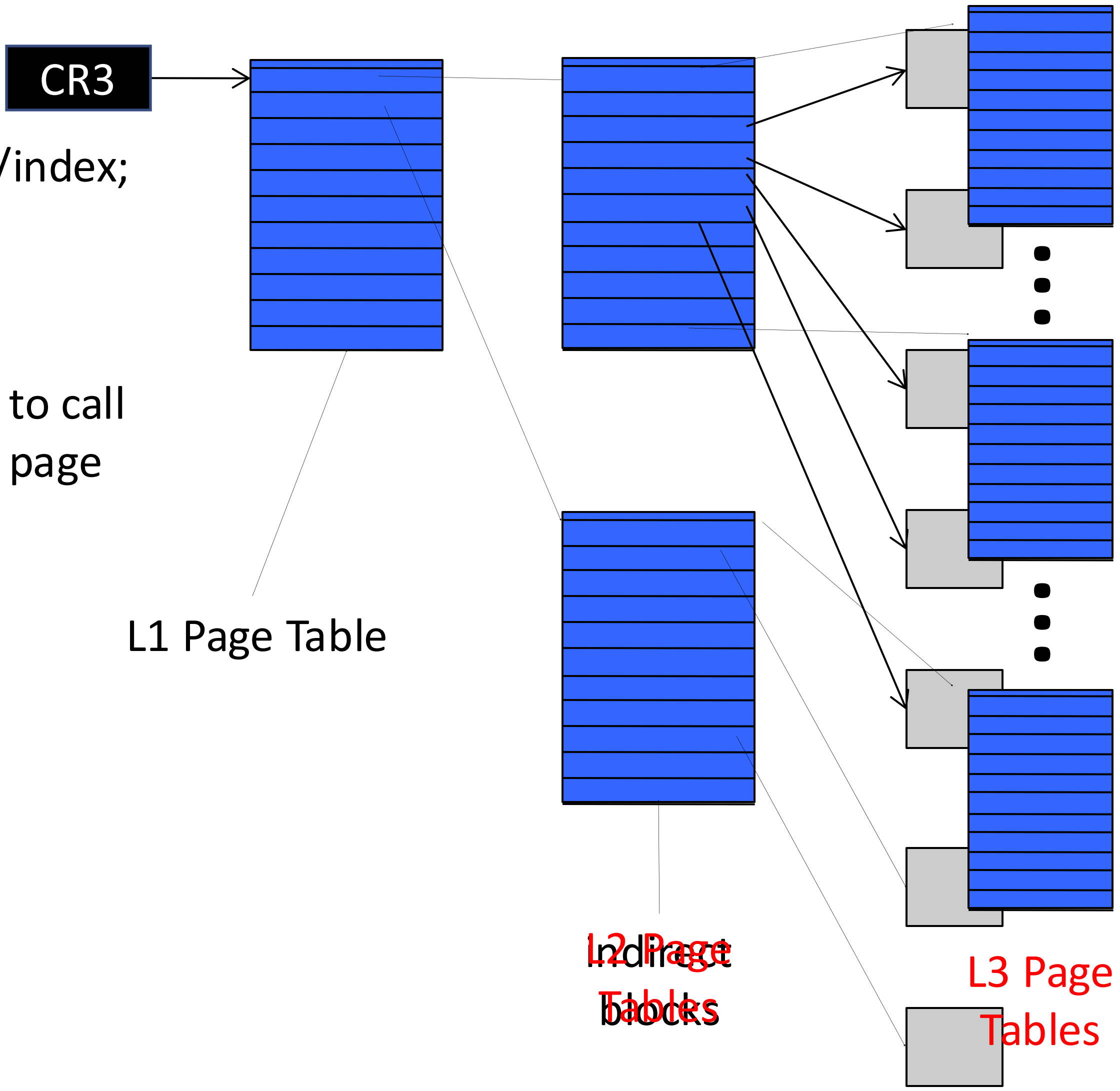
PTEs sometimes reference other in-memory physical pages and sometimes reference disk blocks

From a file index to a page table!

3. The x86-64 has 4-levels of page tables

1. Instead of an inode/index; we have a hardware register: CR3

2. Everything we used to call indirect blocks we call page tables.

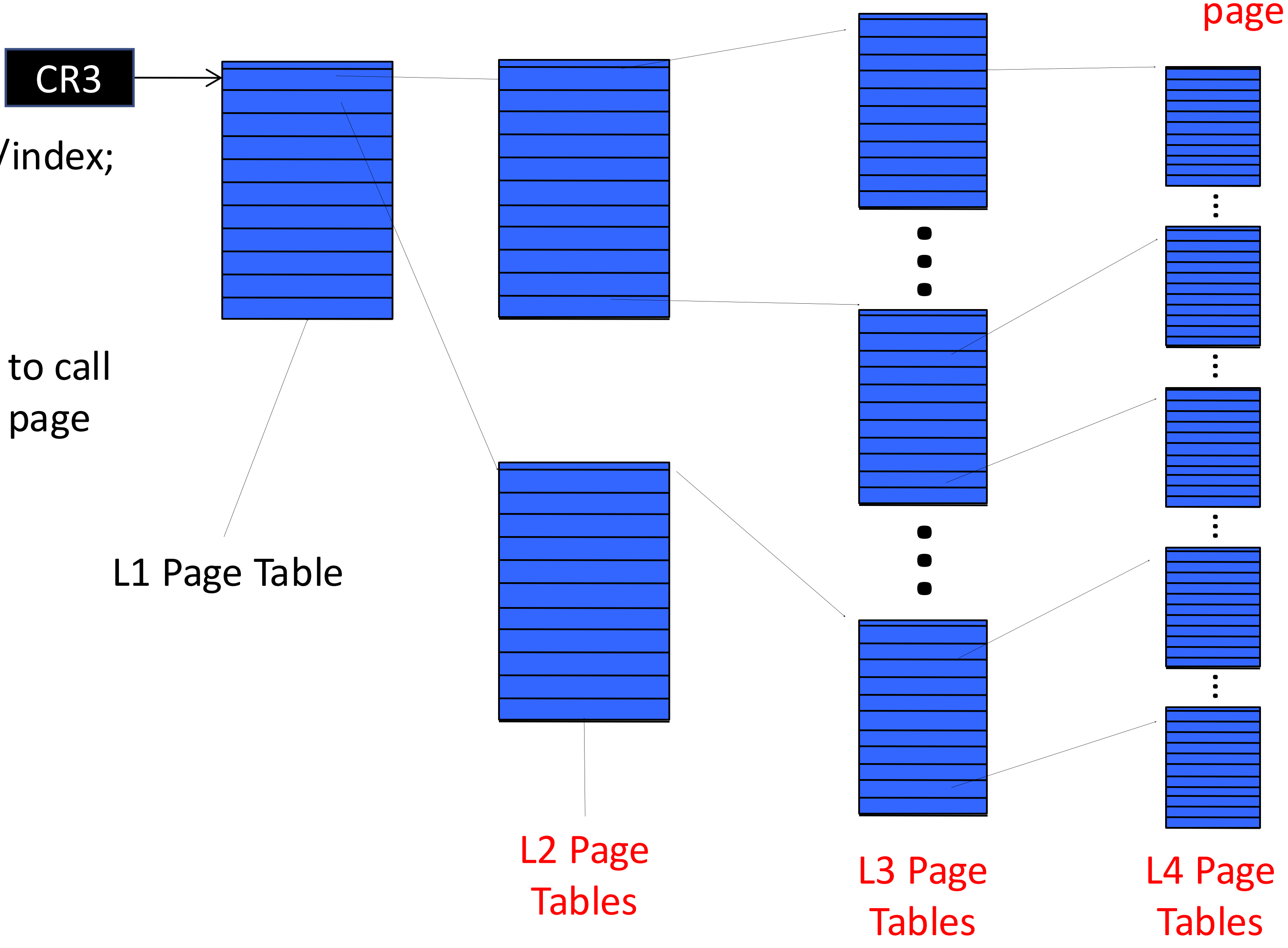


From a file index to a page table!

3. The x86-64 has 4-levels of page tables

1. Instead of an inode/index; we have a hardware register: CR3

2. Everything we used to call indirect blocks we call page tables.



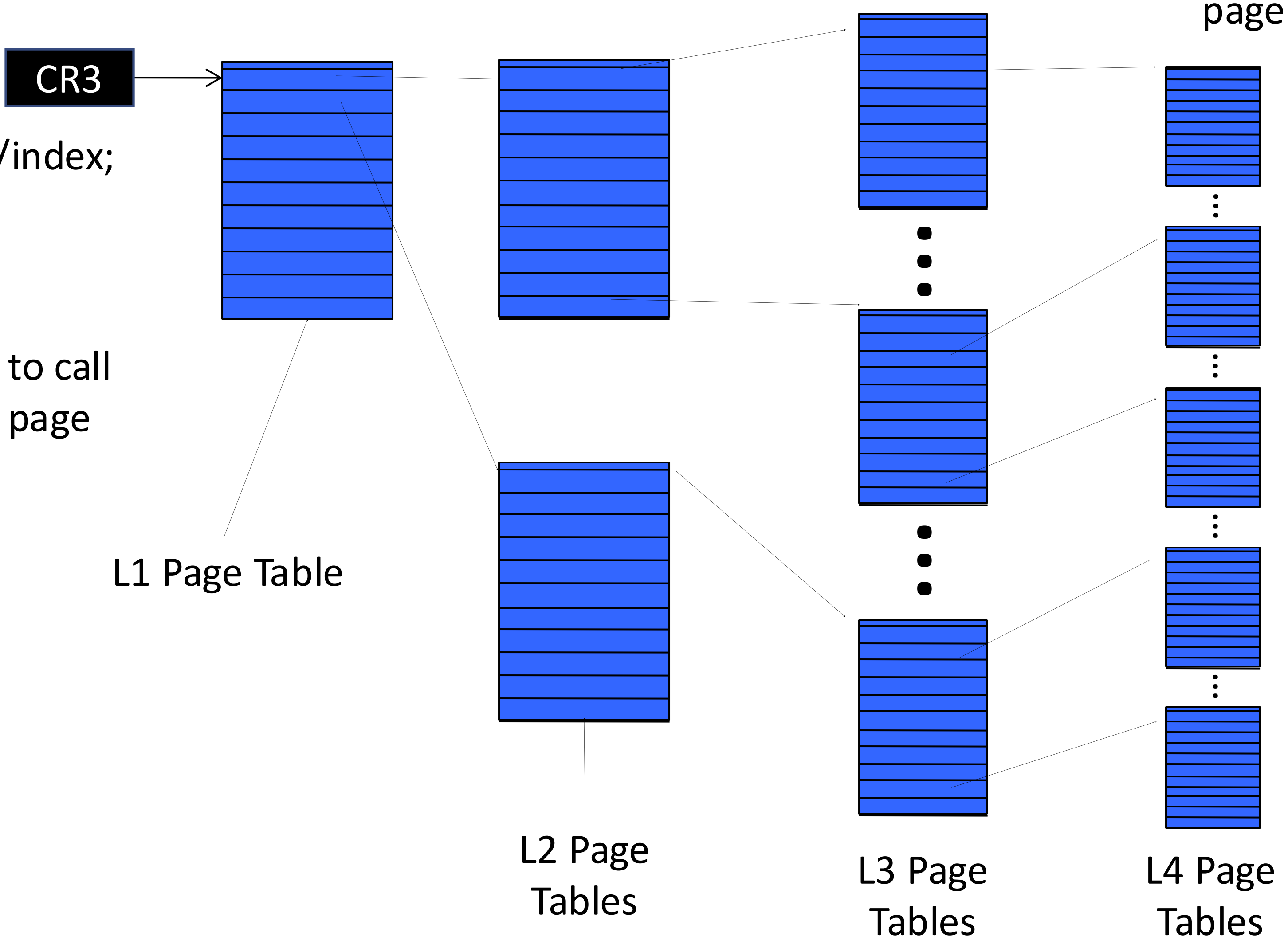
From a file index to a page table!

1. Instead of an inode/index; we have a hardware register: CR3

2. Everything we used to call indirect blocks we call page tables.

3. The x86-64 has 4-levels of page tables

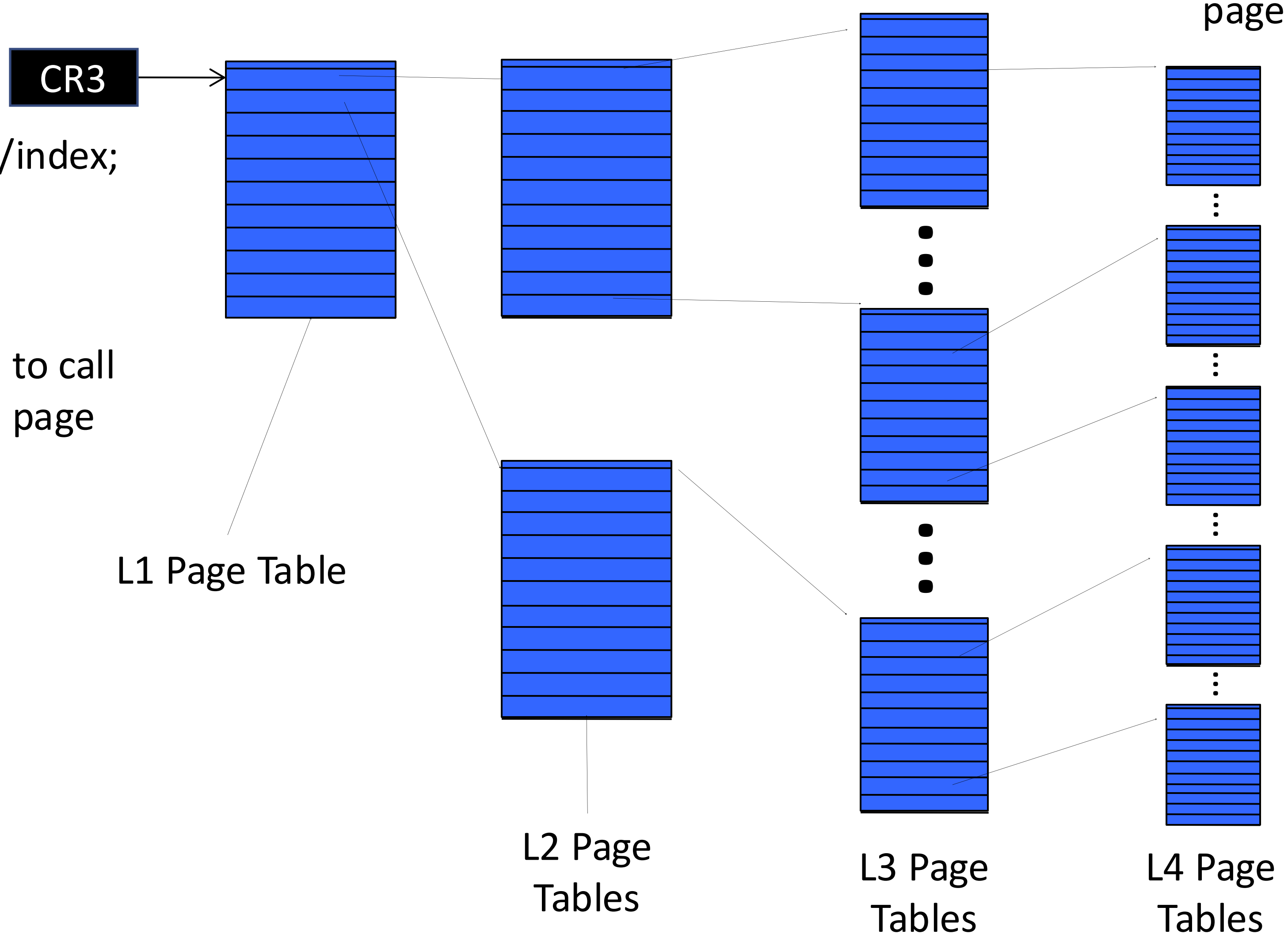
4. All page tables are indexed by bits in the virtual address.



From a file index to a page table!

1. Instead of an inode/index; we have a hardware register: CR3

2. Everything we used to call indirect blocks we call page tables.

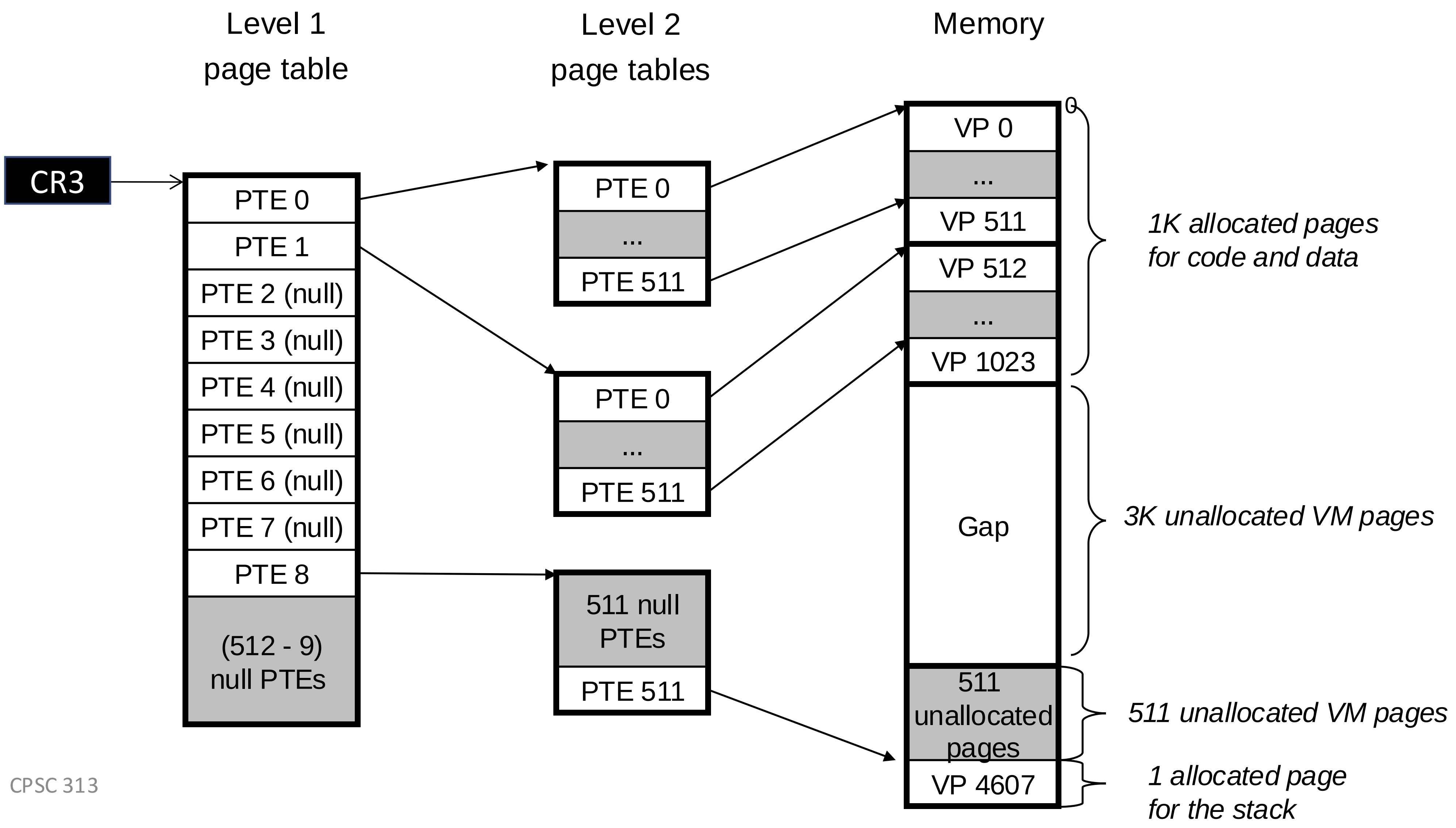


3. The x86-64 has 4-levels of page tables

4. All page tables are indexed by bits in the virtual address.

5. PTEs are 8 bytes

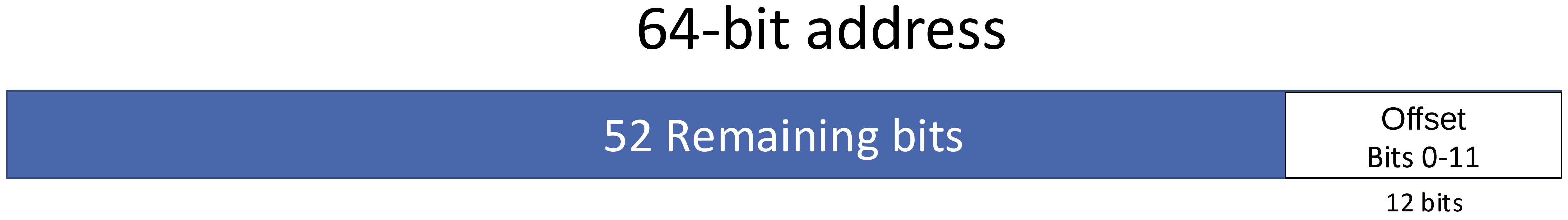
Example for one process (using 2 levels only)



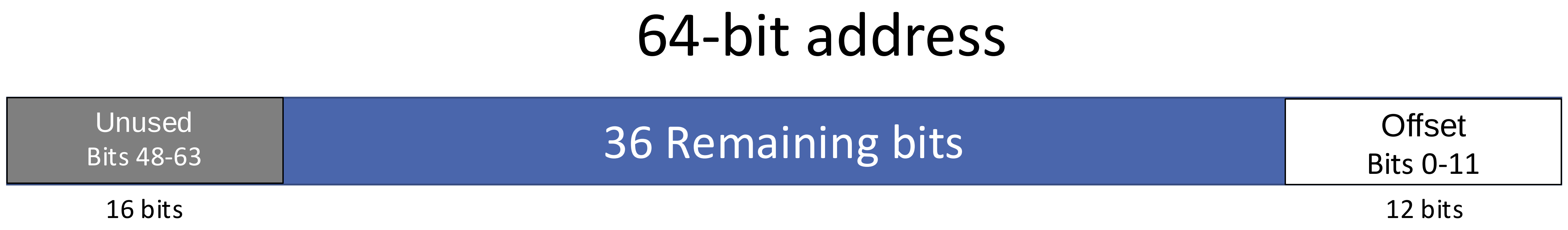
X86 Address Translation

64-bit Virtual Address

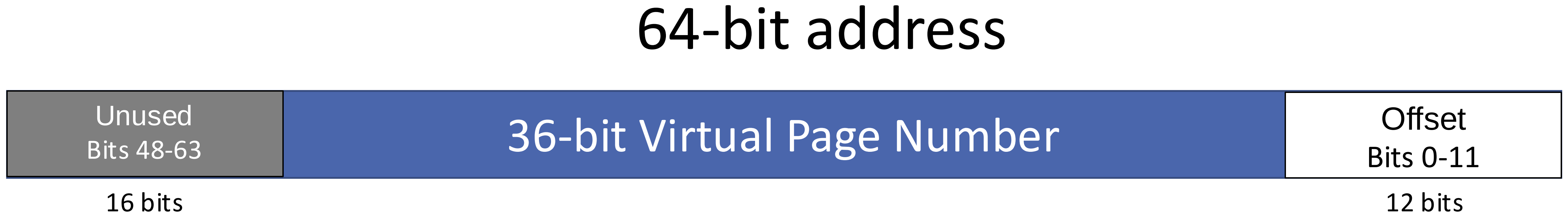
X86 Address Translation



X86 Address Translation



X86 Address Translation



X86 Address Translation

64-bit address

Unused Bits 48-63	L1 Index Bits 39-47	L2 Index Bits 30-38	L3 Index Bits 21-29	L4 Index Bits 12-20	Offset Bits 0-11
16 bits	9 bits	9 bits	9 bits	9 bits	12 bits

X86 Address Translation

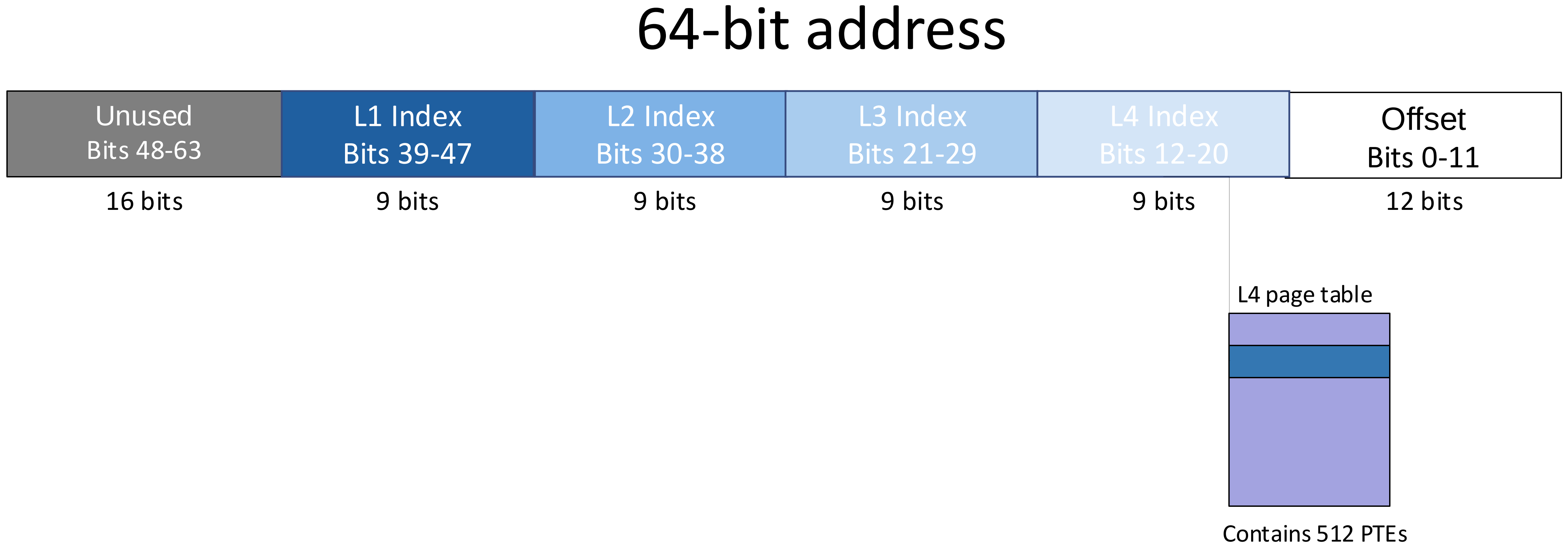
64-bit address

Unused Bits 48-63	L1 Index Bits 39-47	L2 Index Bits 30-38	L3 Index Bits 21-29	L4 Index Bits 12-20	Offset Bits 0-11
16 bits	9 bits	9 bits	9 bits	9 bits	12 bits

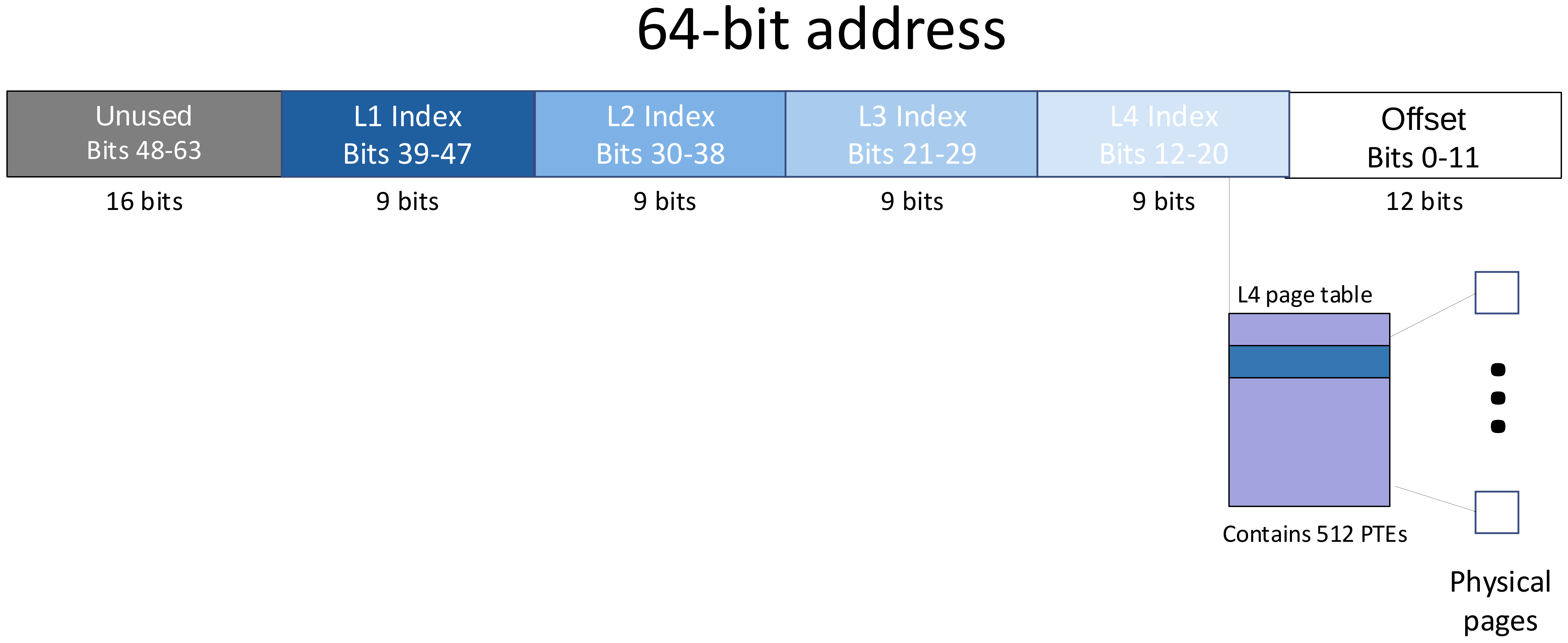
Each of these 9-bit chunks is going to be an index into a part of a page table.
Analogy to indirect blocks:

- An L4 index is for a page table containing PTEs that direct you to blocks in the address space
- An L3 index is for a page table containing PTEs that direct you to L4 page tables
- An L2 index is for a page table containing PTEs that direct you to L3 page tables
- An L1 index is for THE page table containing PTEs that direct you to L2 page tables

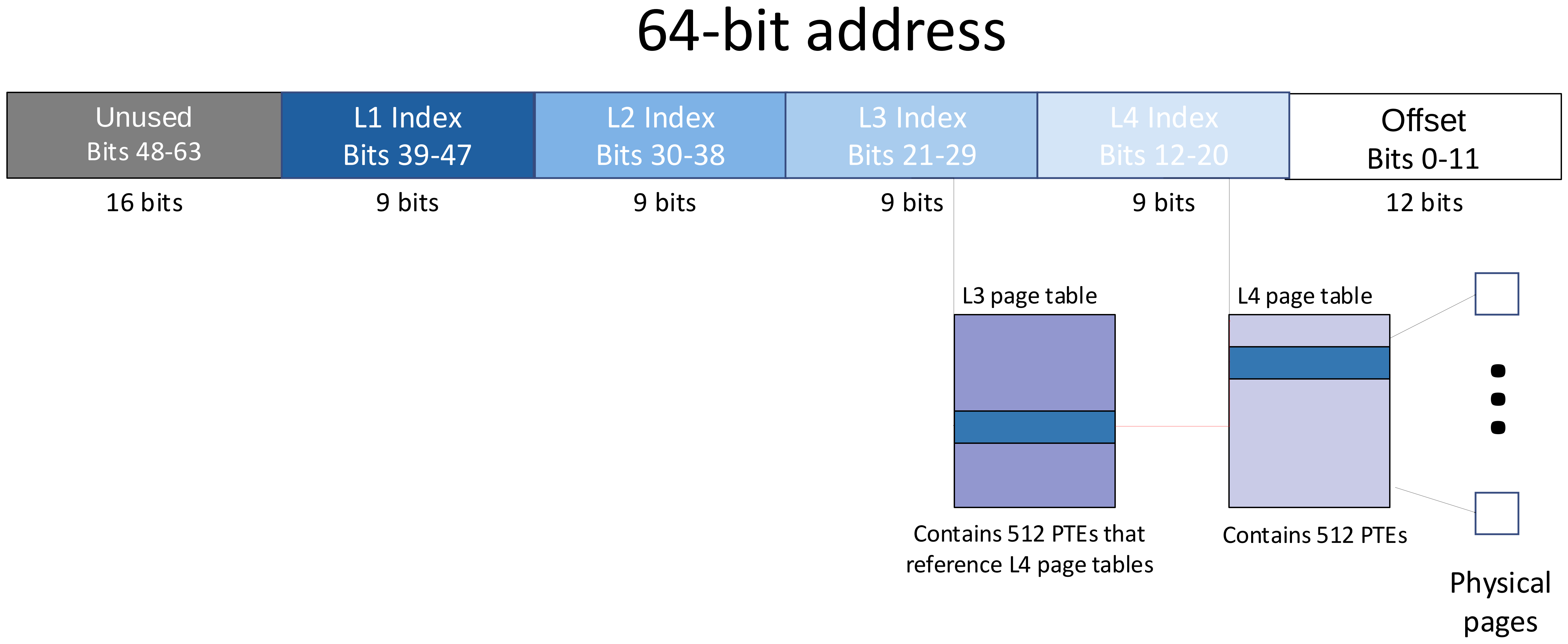
X86 Address Translation



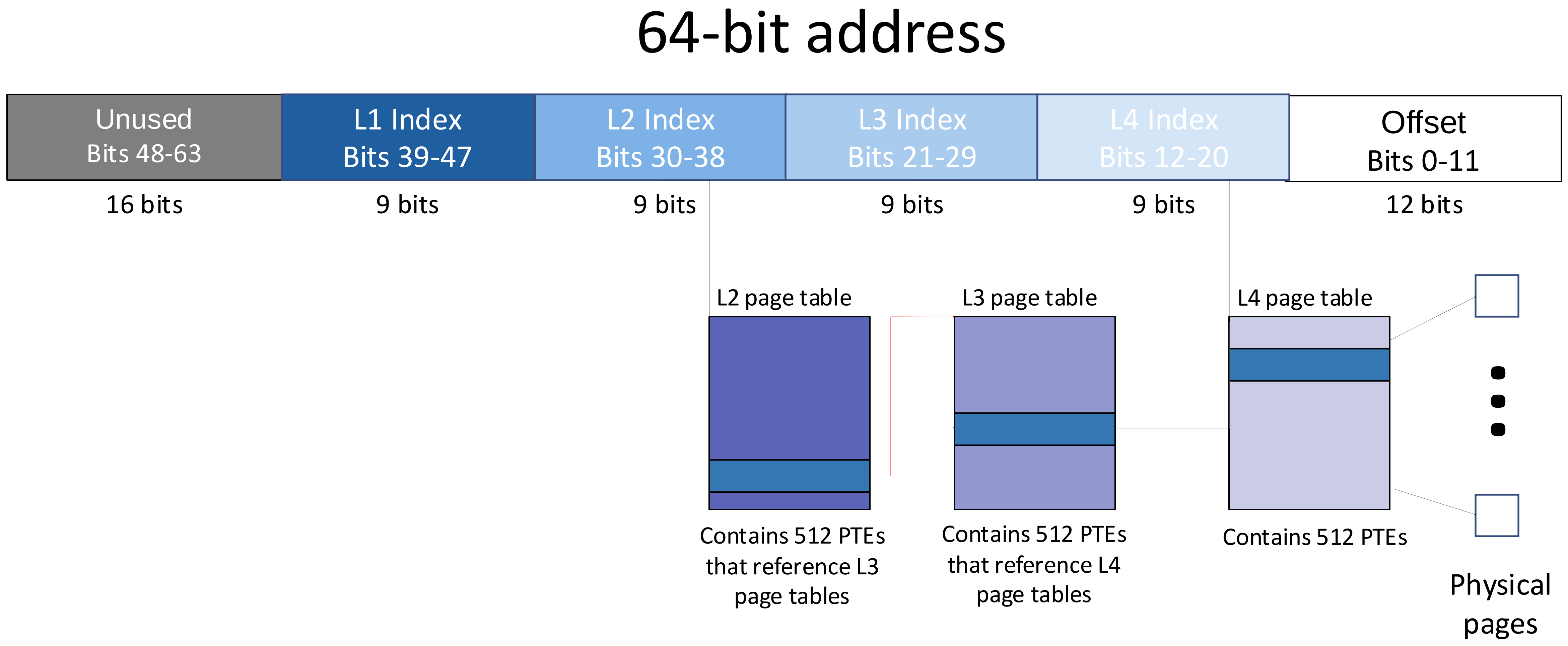
X86 Address Translation



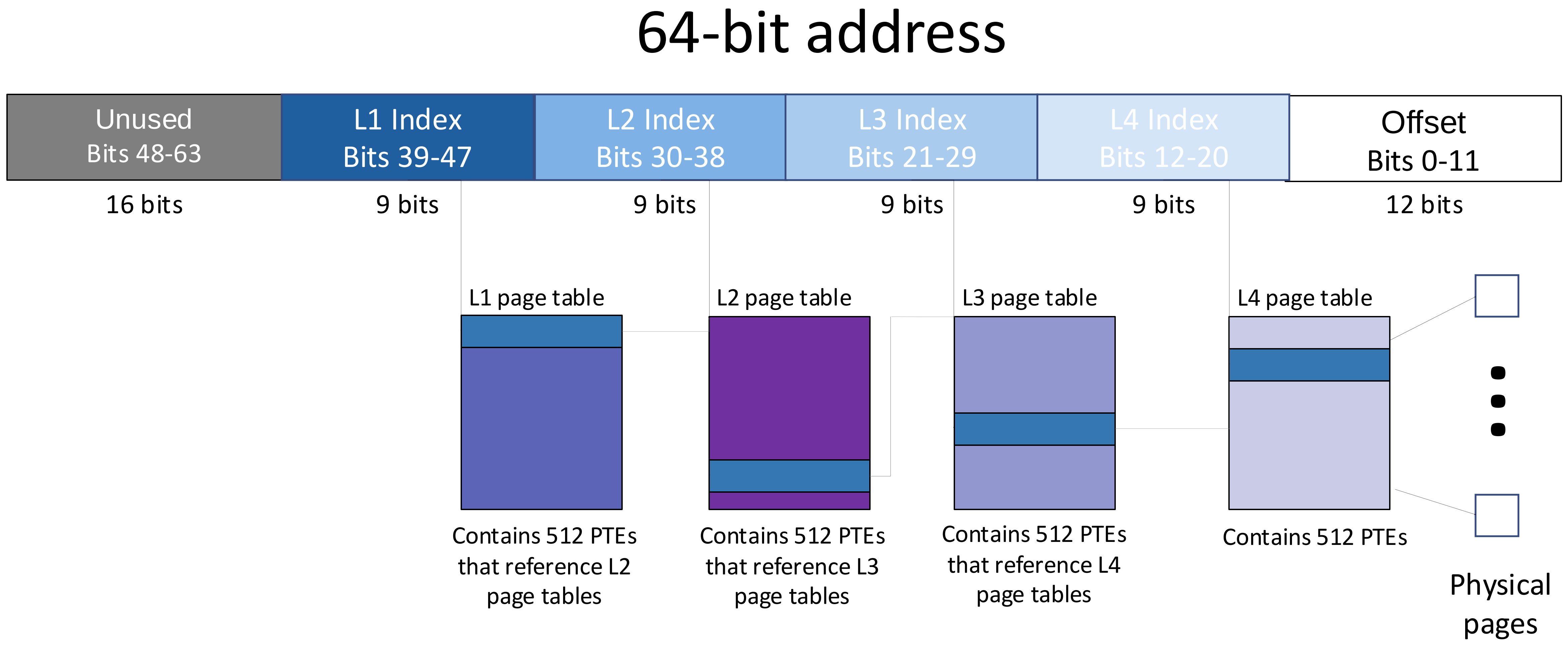
X86 Address Translation



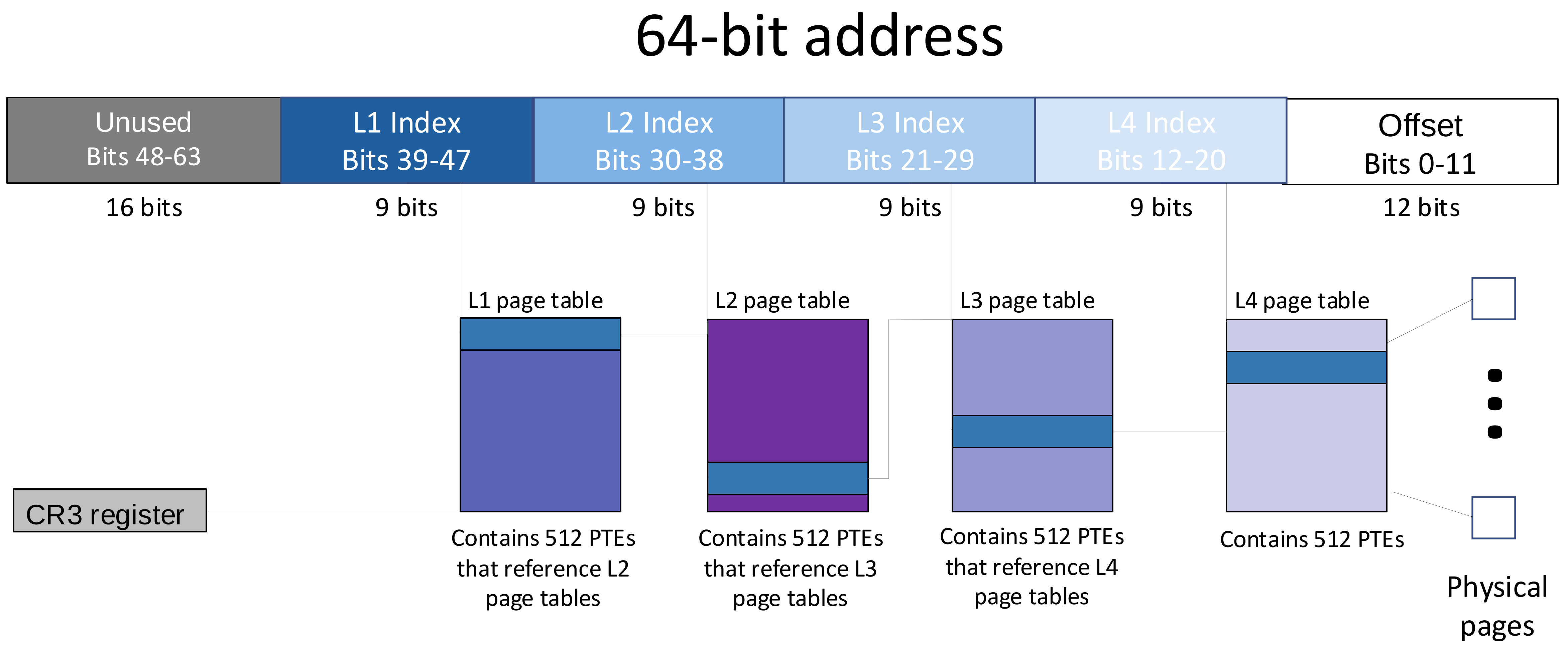
X86 Address Translation



X86 Address Translation



X86 Address Translation



This is **exactly** the division/modulo process we use to find indices in a multi-level index!

A Closer Look at PTEs: Levels 1, 2, and 3

- Levels 1, 2, and 3 each refer to other page tables, if P=1:

XD	Unused	Page table physical base address	Unused		PS		A	CD	WT	U/S	R/W	P
63	52-62	12-51	9-11	8	7	6	5	4	3	2	1	0

XD: Execute disable (can be used to disable instruction fetches from, e.g., stack)
Bits 12-51 contain the Physical page number of the page table referenced by this PTE
PS: Page size (safe to ignore this)
A: Reference bit (used for page replacement; check out the video); set on every access
CD: Are we allowed to cache the page table we reference
WT: Write through or write-back cache policy for the child page table
U/S: User or Supervisor mode for all pages reachable by the child table
R/W: Read-only or read/write permissions for all reachable pages
P: Is the child table present in main memory?

A Closer Look at PTEs: Levels 1, 2, and 3

- Levels 1, 2, and 3 each refer to other page tables:

XD	Unused	Page table physical base address	Unused		PS		A	CD	WT	U/S	R/W	P
63	52-62	12-51	9-11	8	7	6	5	4	3	2	1	0

If P = 0, that says that the page table is not in memory (DRAM); the OS is free to use the rest of the entry in whatever way it wants; what kinds of information do you think the OS wants to put there?

The OS can do whatever it wants with these bits!	P
Bits 1-63	0

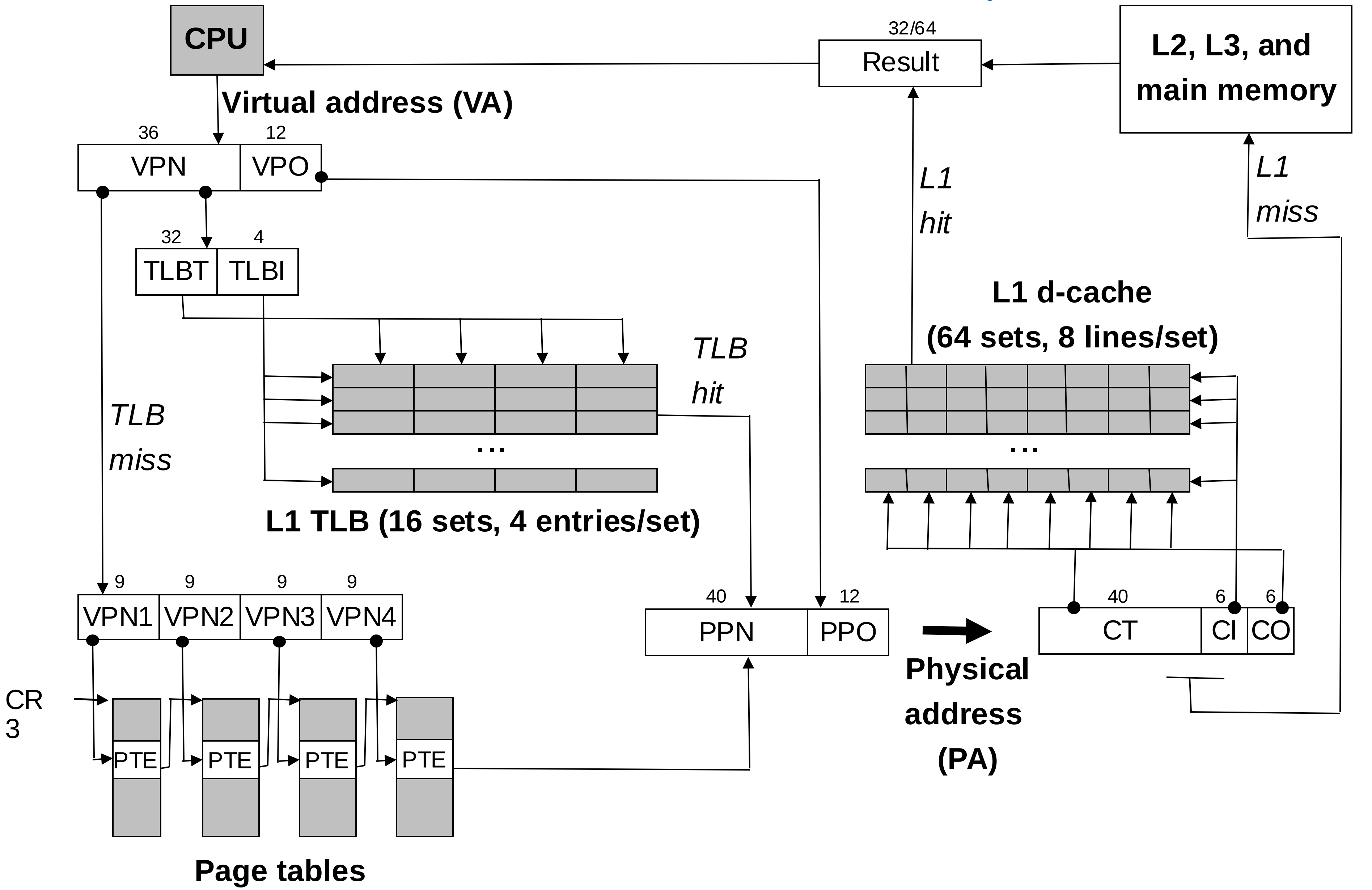
A Closer Look at PTEs: Level 4

- Levels 4 PTEs refer to pages that belong to the process' VAS, if P=1

XD	Unused	Page physical base address	Unused	G	PS	D	A	CD	WT	U/S	R/W	P
63	52-62	12-51	9-11	8	7	6	5	4	3	2	1	0

XD: Execute disable (can be used to disable instruction fetches from, e.g., stack)
Bits 12-51 contain the Physical page number of the page referenced by this PTE
G: Global page (is not evicted from TLB on a process switch)
PS: Page size (safe to ignore this)
D: Dirty bit (set on every write)
A: Reference bit (used for page replacement; stay tuned until next week); set on every access
CD: Are we allowed to cache the page table we reference
WT: Write through or write-back cache policy for the child page table
U/S: User or Supervisor mode for all pages reachable by the child table
R/W: Read-only or read/write permissions for all reachable pages
P: Is the page present in main memory? (If P=0, the OS again uses the rest for whatever it wants!)

A closer look at the x64-64 VM system



Questions to test your understanding:

- Think about these before the next class!

1. Must cr3 contain a physical or virtual address? Why?
2. How large is a single page table? Why?
3. Why do PTEs contain physical page numbers?
4. Why don’t PTEs need an offset? (I.e., why is a page number sufficient?)
5. For each level page table, how much memory is reachable?
6. On the previous slide, the CI + CO happened to fit in the VPO.
   Is that necessary? Is it useful in any way?

Intel x86 VM Historical Notes

- The Intel x86 Virtual Memory Architecture reflects many major revisions that have occurred over various generations of Intel microprocessors.
- While this makes the system a bit more complicated than some others, it is the most widely used platform today, so understanding it will serve you well.
- Note:
  - This presentation did not cover x86 in its full glory.
  - We cover it sufficiently so should you ever need to dig into the details, they will make sense.