Digital Logic Design + Computer Architecture

Sayandeep Saha

Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Virtual Memory

Creating Illusion

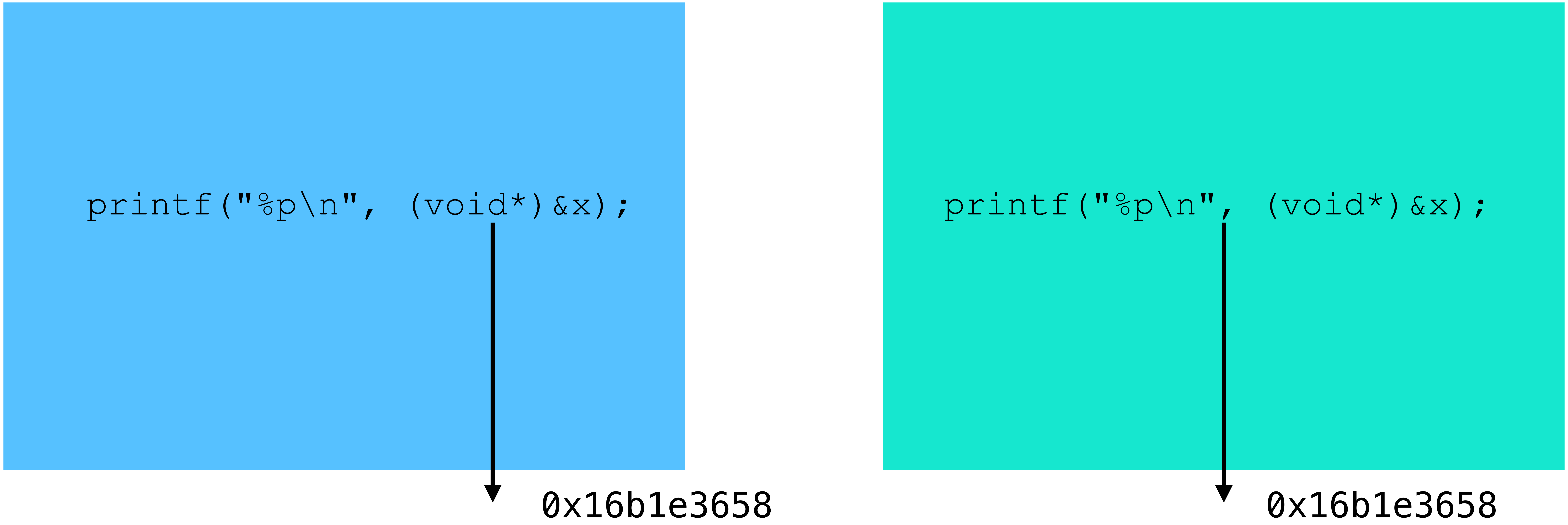
- Consider

```
printf("The address of x is: %p\n", (void*)&x);

printf("The address of x is: %llx\n”, (unsigned long long)&x);
```

- These are not the actual addresses in DRAM !!! these are virtual addresses

Creating Illusion



Program 1 (or better Process 1)

```
printf("%p\n", (void*)&x);
```

```
0x16b1e3658
```

Program 2 (or better Process 2)

```
printf("%p\n", (void*)&x);
```

```
0x16b1e3658
```

These two maps in completely different locations in the DRAM

Creating Illusion

- Modern systems are multiprogramming-based
  - Run multiple programs on the same processor at the same time
  - Virtual memory simplifies programming
  - More importantly: it provides safe memory isolation between two processes.
    - Isolation is handled under the hood

Creating Illusion

View of the Programmer

- Large contiguous address space
  - Can allocate blocks of contiguous addresses, say using `malloc`
  - Do not need to think about how these addresses are organized physically.
  - Allows a program to (virtually) exceed the amount of physical memory — though not very significant these days.

Inside the System

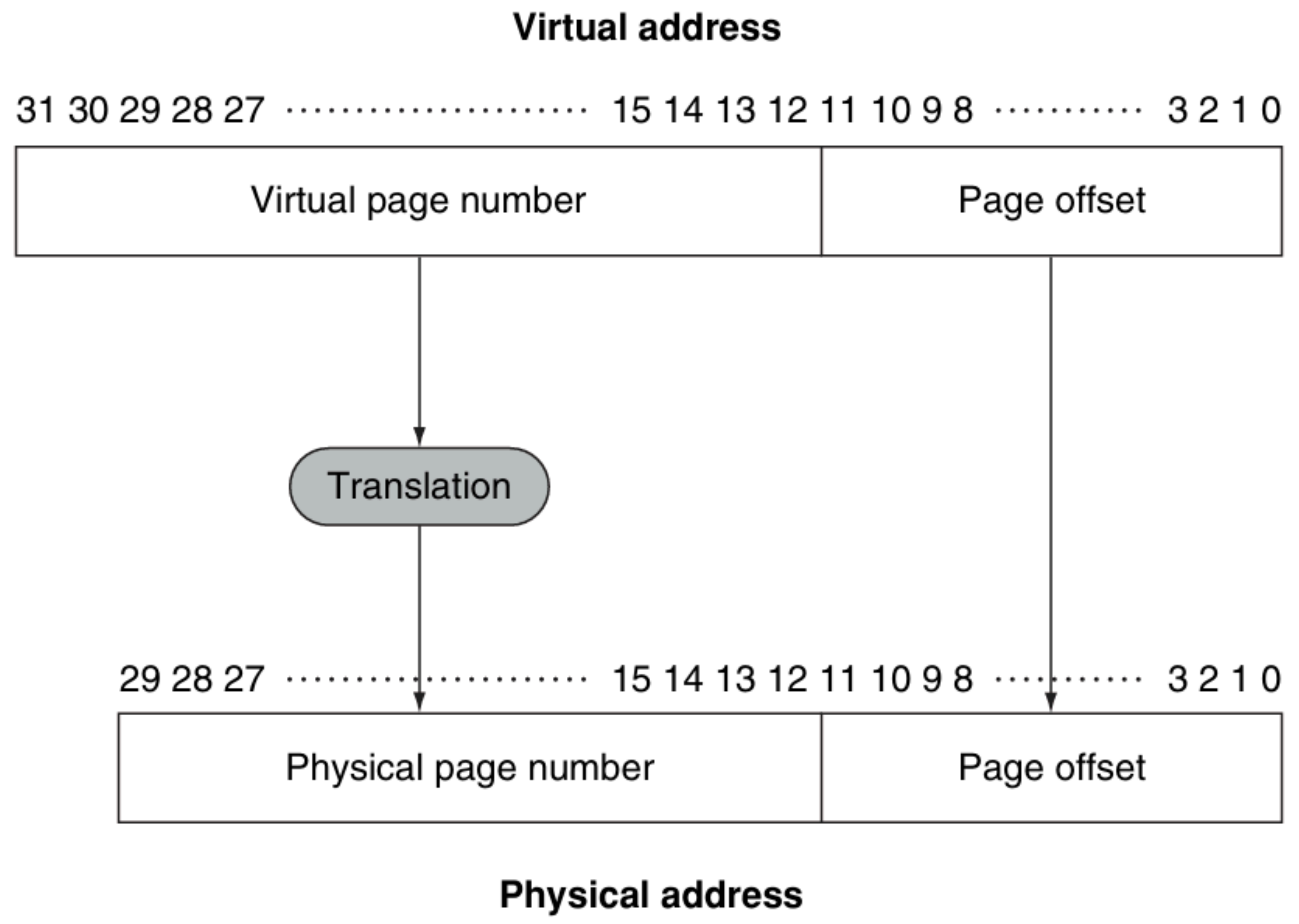
- User virtual address space created by mapping to set of **pages**
  - Addresses in main memory need not be contiguous
  - Two different process has isolated address space. They do not interfere with each other
- OS manages many processes simultaneously
  - Continually switching among processes (context switching)

How to Create Illusion: Paged Memory System

- Processor generates Virtual Address
- The address is translated to Physical Address
- Translation happens through a table called Page Table
- What is a “page”?
    - A fixed size contiguous set of virtual addresses — typically 4KiB (4*1024 bytes) to 16 KB. But can be large as 1 Gib (huge pages). — why?
    - Physical memory is also broken into page-sized chunks, called frames

How to Create Illusion: Paged Memory System

- A Page Table contains the address at the start of each page

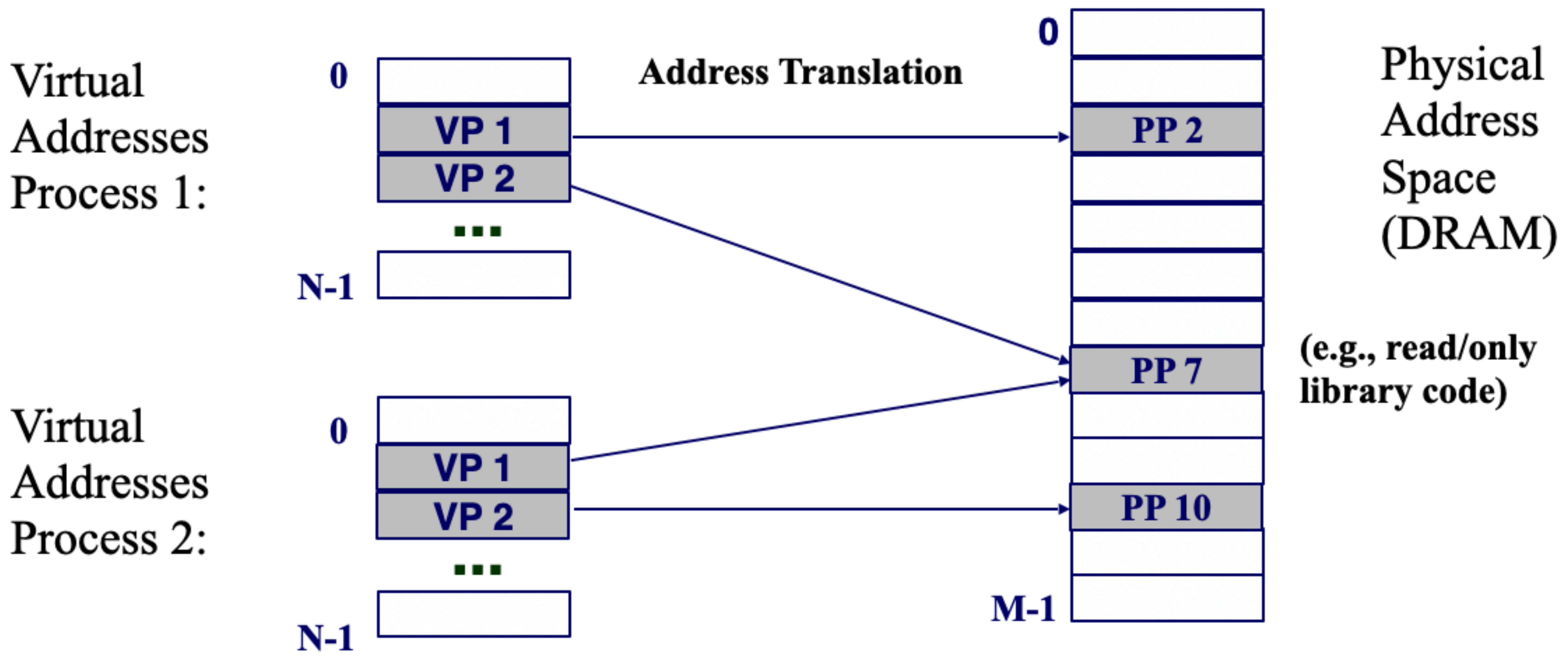


- Physical page number is also called “frame number”

How to Create Illusion: Page Table

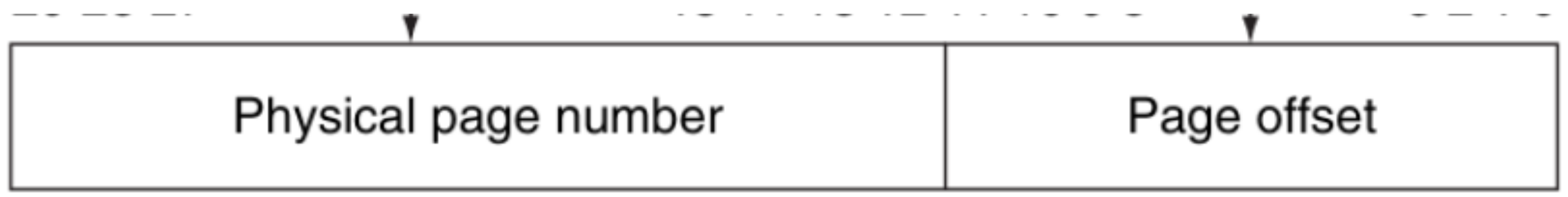
- Processor generates VA =

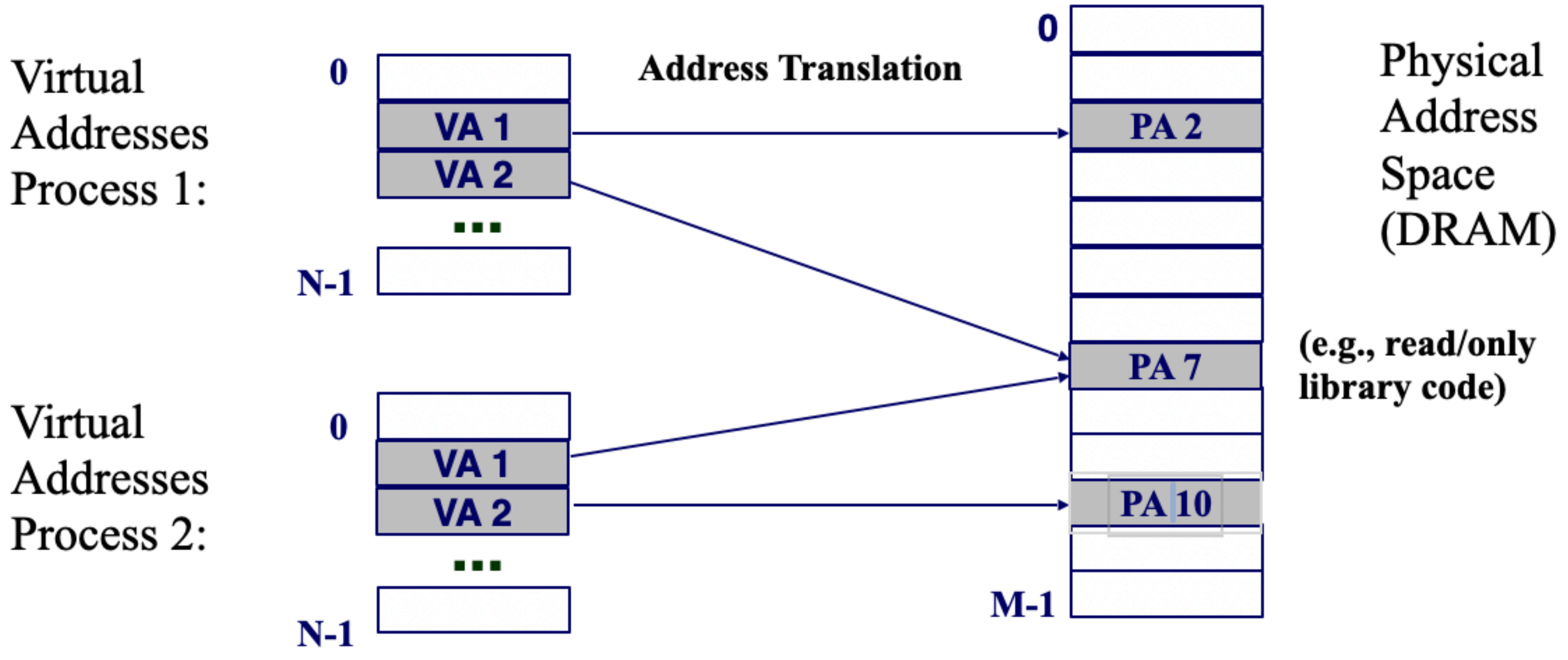
Page Number	Offset


Virtual Addresses Process 1:
0
VP 1
VP 2
...
N-1
Address Translation
0
PP 2
PP 7
PP 10
M-1
Physical Address Space (DRAM)
(e.g., read/only library code)
Virtual Addresses Process 2:
0
VP 1
VP 2
...
N-1

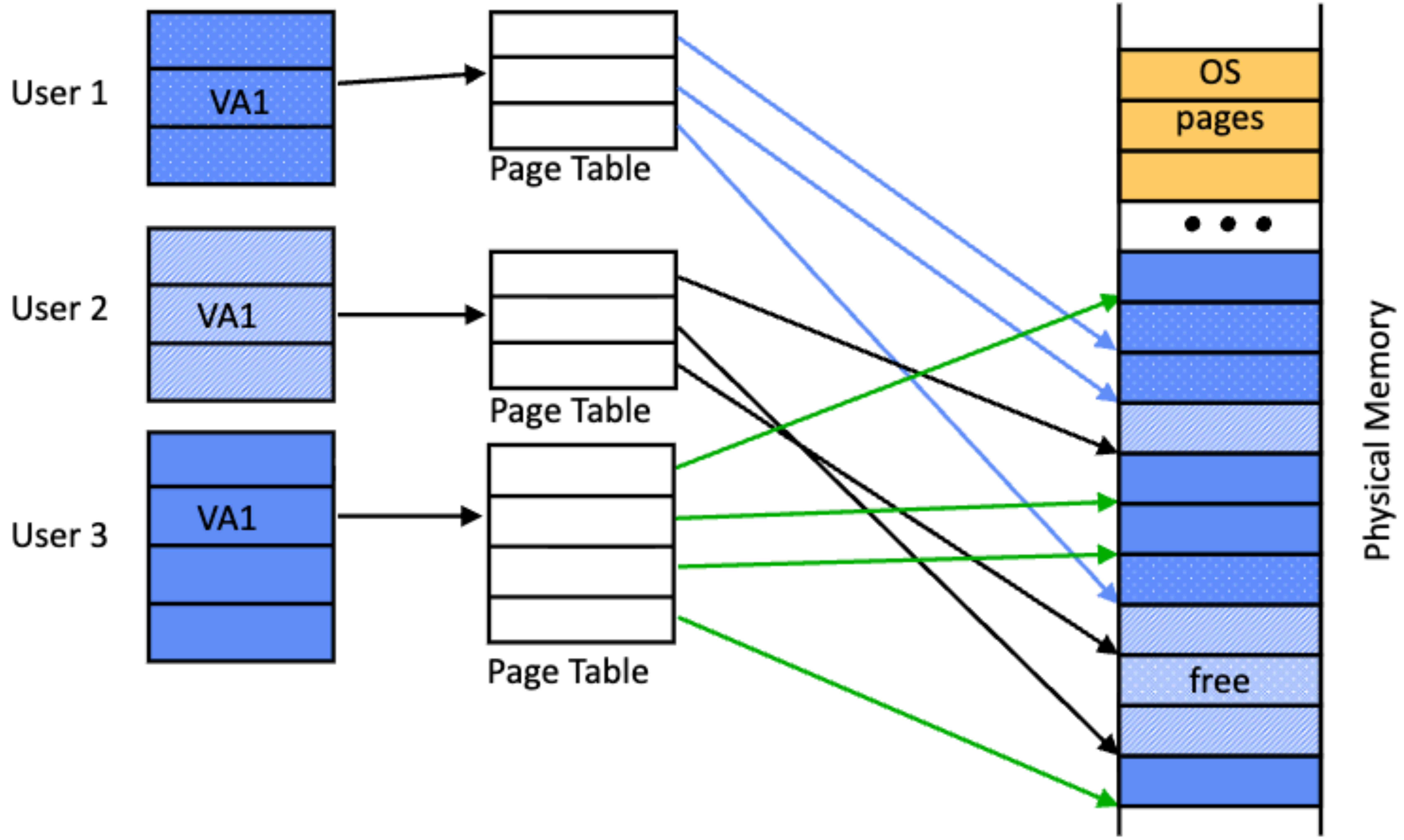
How to Create Illusion: Page Table

- Page table returns PA =




Virtual Addresses Process 1:
0
VA 1
VA 2
...
N-1
Virtual Addresses Process 2:
0
VA 1
VA 2
...
N-1
Address Translation
0
PA 2
PA 7
PA 10
M-1
Physical Address Space (DRAM)
(e.g., read/only library code)

How to Create Illusion: Page Table



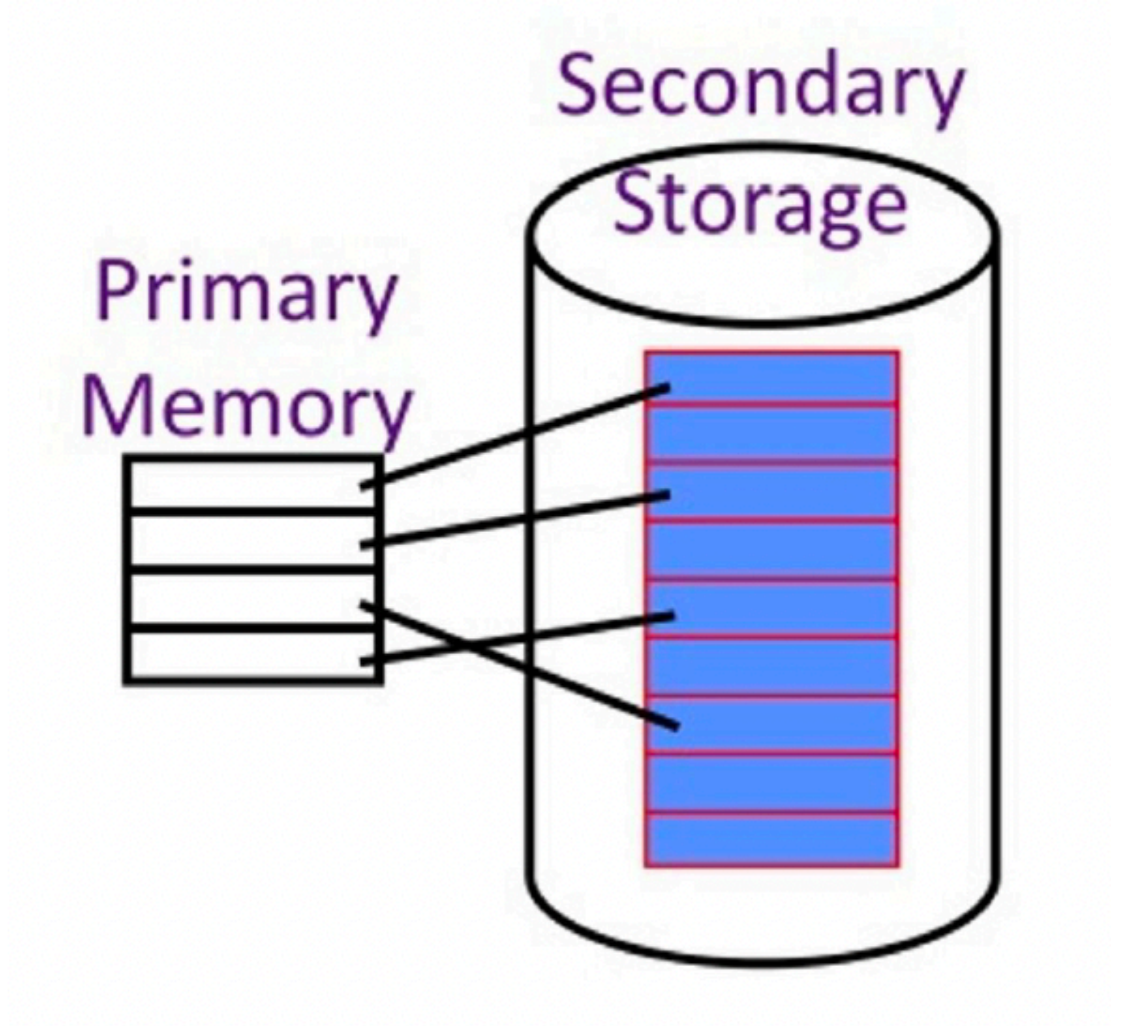
- Page table is user specific — each program has one.

Demand Paging — When Your DRAM is a Cache

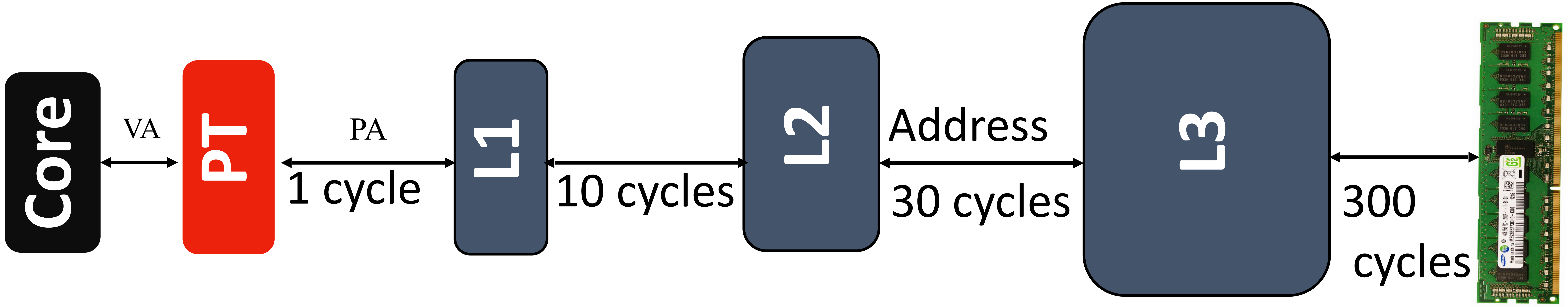
- Memory isolation through paging is understandable, but does all the pages in a program remains in the DRAM?

- Think about an LLM with 1 trillion parameters — size can be 300 GB..
  - But maybe you have only 128 GB RAM

- Demand Paging: Bring pages from disc to DRAM when you need it
- Disk has a space to store pages called swap space
- Page fault: A miss in the DRAM, bring page from disk
- Demand paging helps in running ultra large programs without bothering about your DRAM size.

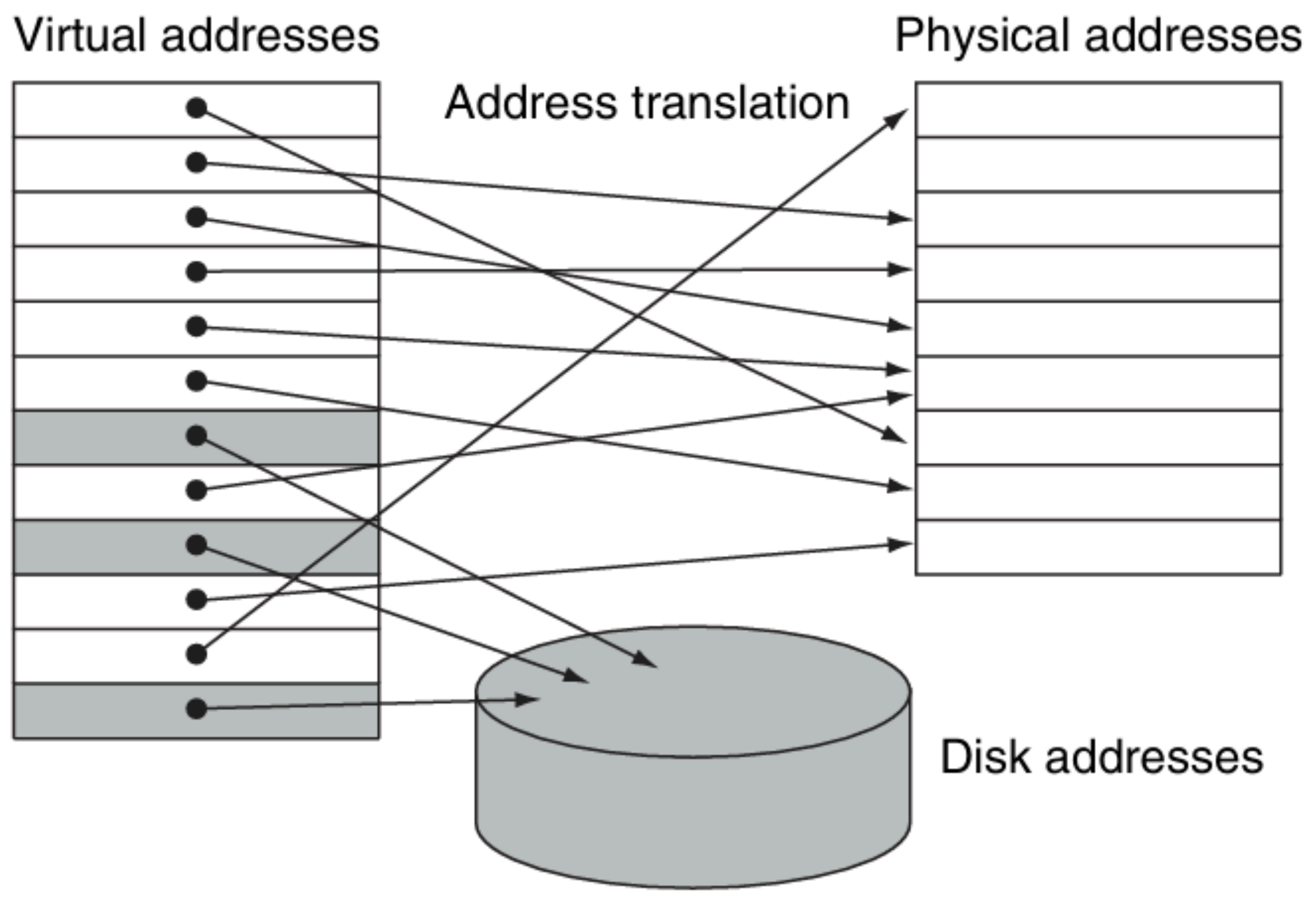


Effect of Address Translation on Program Execution



- This view is a bit simplistic — actually there are more thrills in the story
- But main question — how fast should be the PT access?
  - More fundamentally, what is the PT — hardware or software??

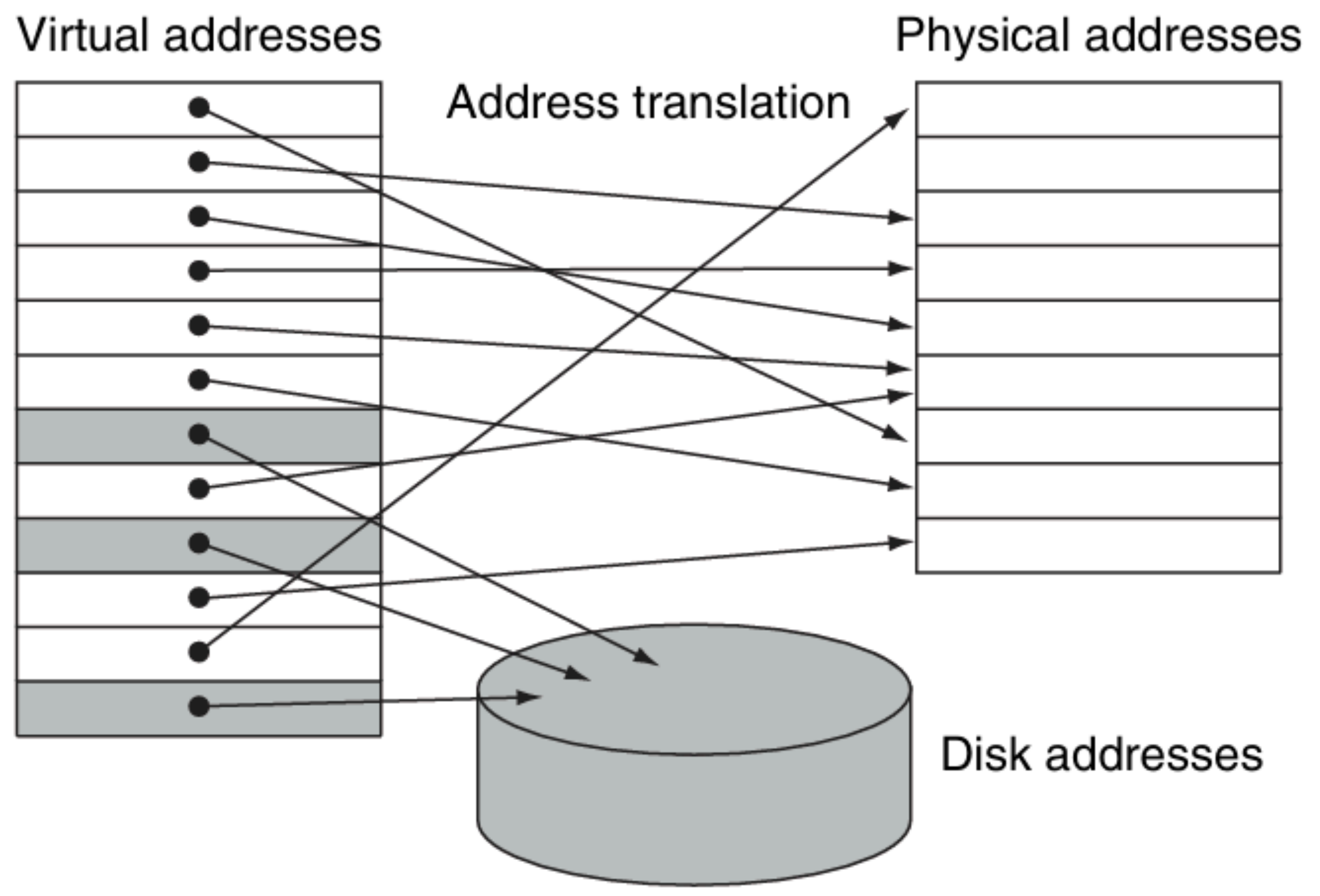
Let’s See…



- Addresses can be in DRAM or in Disk — for the later, the page is loaded into the DRAM (demand paging)

- But where the is the PT located?

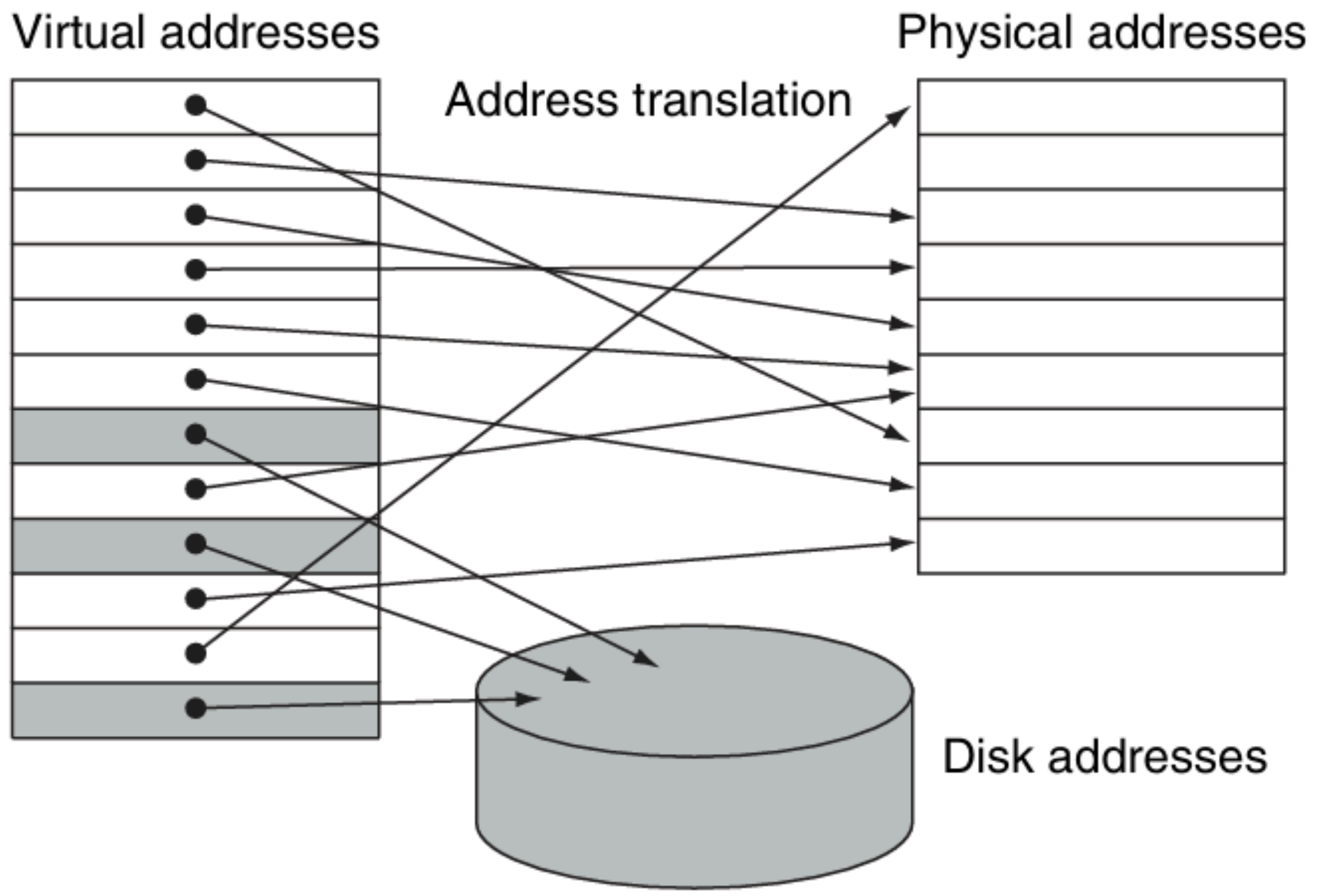
Let’s See…



- Addresses can be in DRAM or in Disk — for the later, the page is loaded into the DRAM (demand paging)

- But where the is the PT located? — in memory

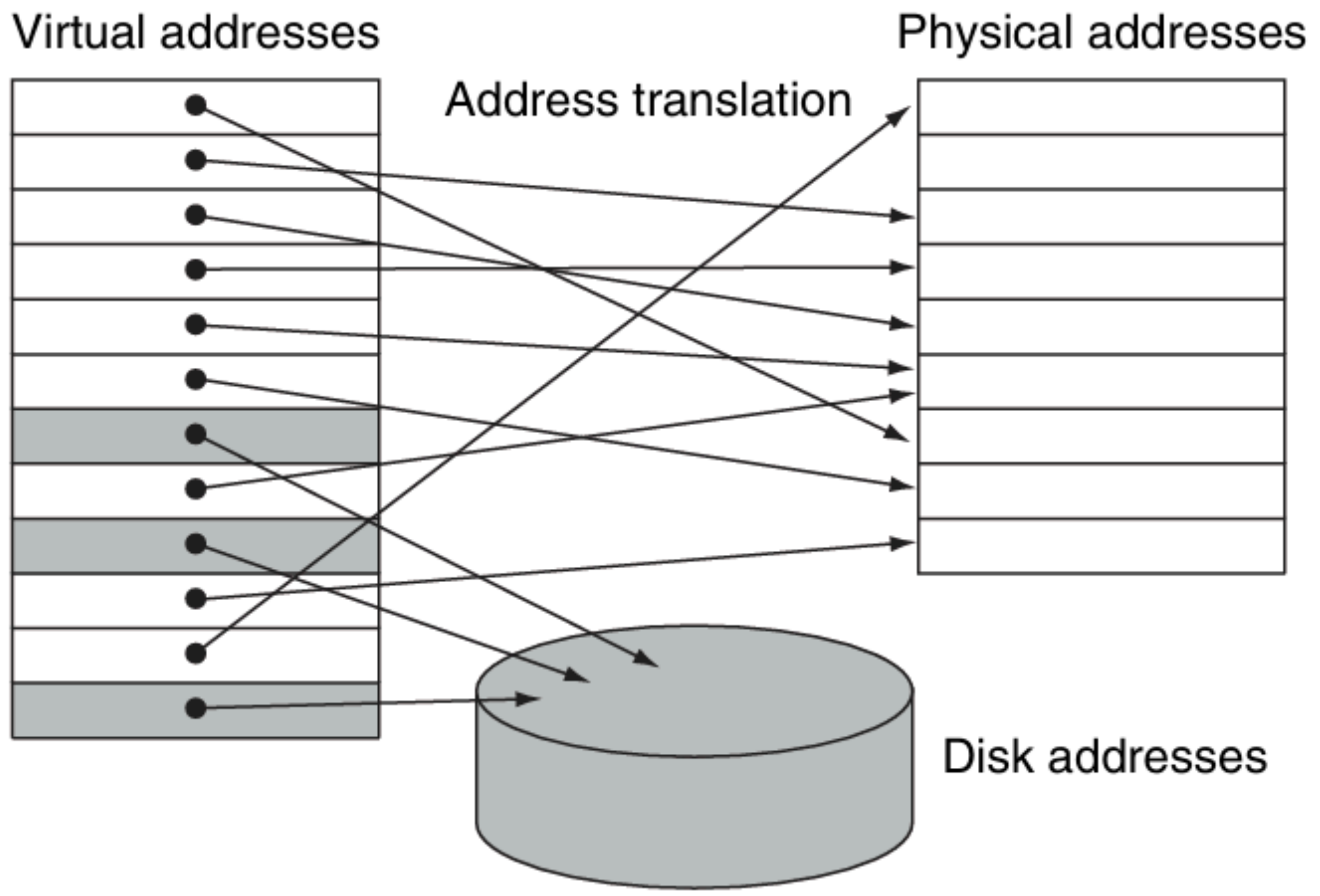
Let’s See…



- Addresses can be in DRAM or in Disk — for the later, the page is loaded into the DRAM (demand paging)

- But where the is the PT located? — in memory
- Does the PT fits in your DRAM?

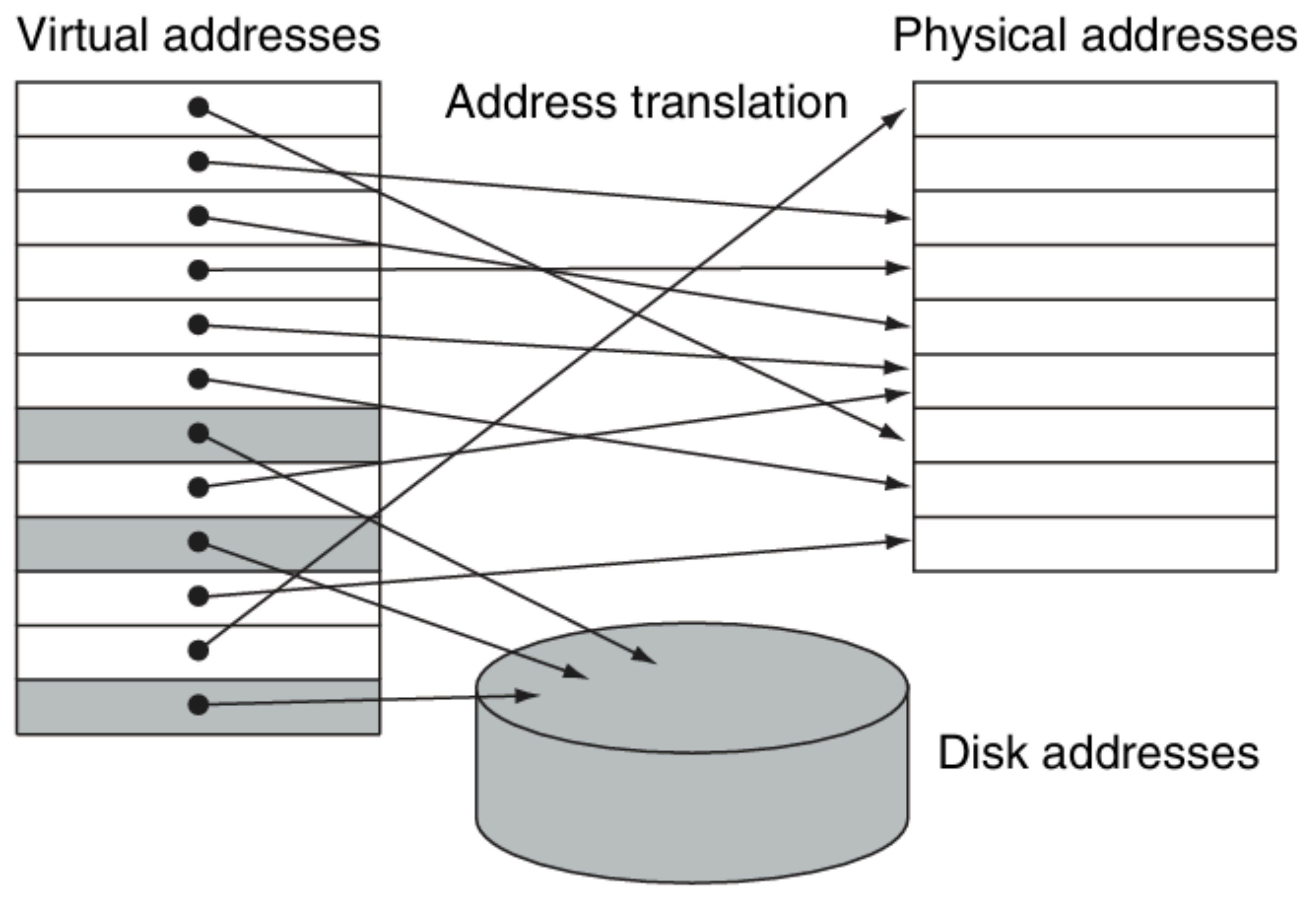
Let’s See…



- Addresses can be in DRAM or in Disk — for the later, the page is loaded into the DRAM (demand paging)

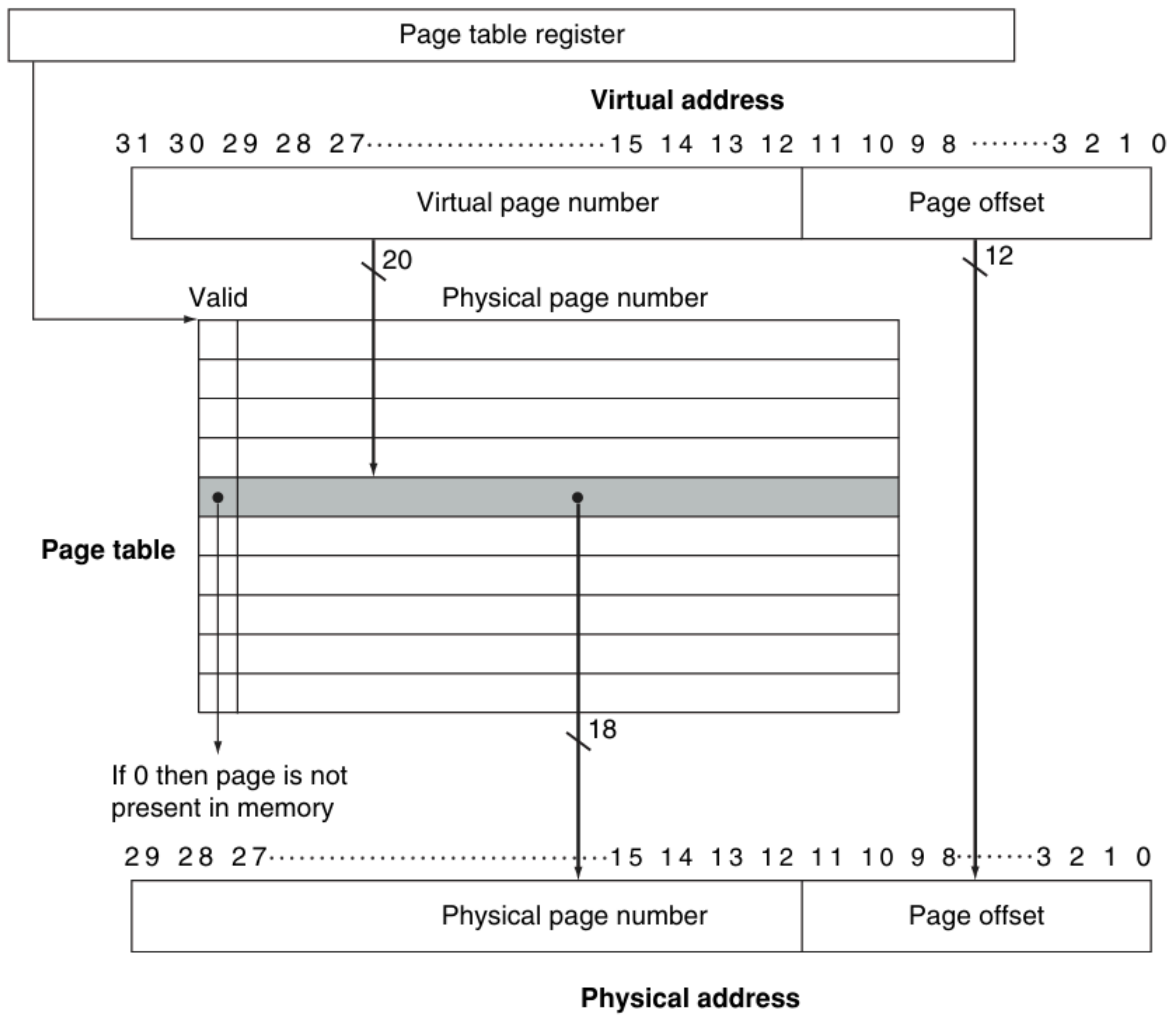
- But where the is the PT located? — in memory
- Does the PT fits in your DRAM? — depends

Let’s See…



- Ok, PT is in memory — but how to search the PT to have address translation?
- Point!! PT is a part of the memory allocated to a program (process). Also, each process has a register called *page-table register* which stores the address of its page table

Let’s See…



- What can be the size of a page table ??

Size of a Typical Page Table

- Let's assume a 32-bit virtual address space — 2^{32} bytes
- Page size is 4 KiB — 2^{12} bytes
- Let the physical address space is 1 GiB — 2^{32} bytes
- Number of pages in the page table — 2^{20} — 1 million!!
- Let's assume each entry is of 4 byte (32 bit) — VA is just an index
- Size of the page table — $2^{20} \times 4$ bytes $= 4$ MiB
- So each process needs 4 MiB for storing its page table

Size of a Typical Page Table

- So each process needs 4 MiB for storing its page table — not bad
- But you may have 100 processes
- Also, think about a 64-bit address space with 4KiB pages…
  - 2^{52} pages.
  - Typical size of each PT entry 8 bytes.
  - So size of page table — 2^{55} bytes — 32 petabytes..

The Page Table Dilemma

- How to store such large page tables efficiently?
- How to quickly search page tables?
- Intuition: Do you ever allocate 32 petabytes of memory?? — no!!

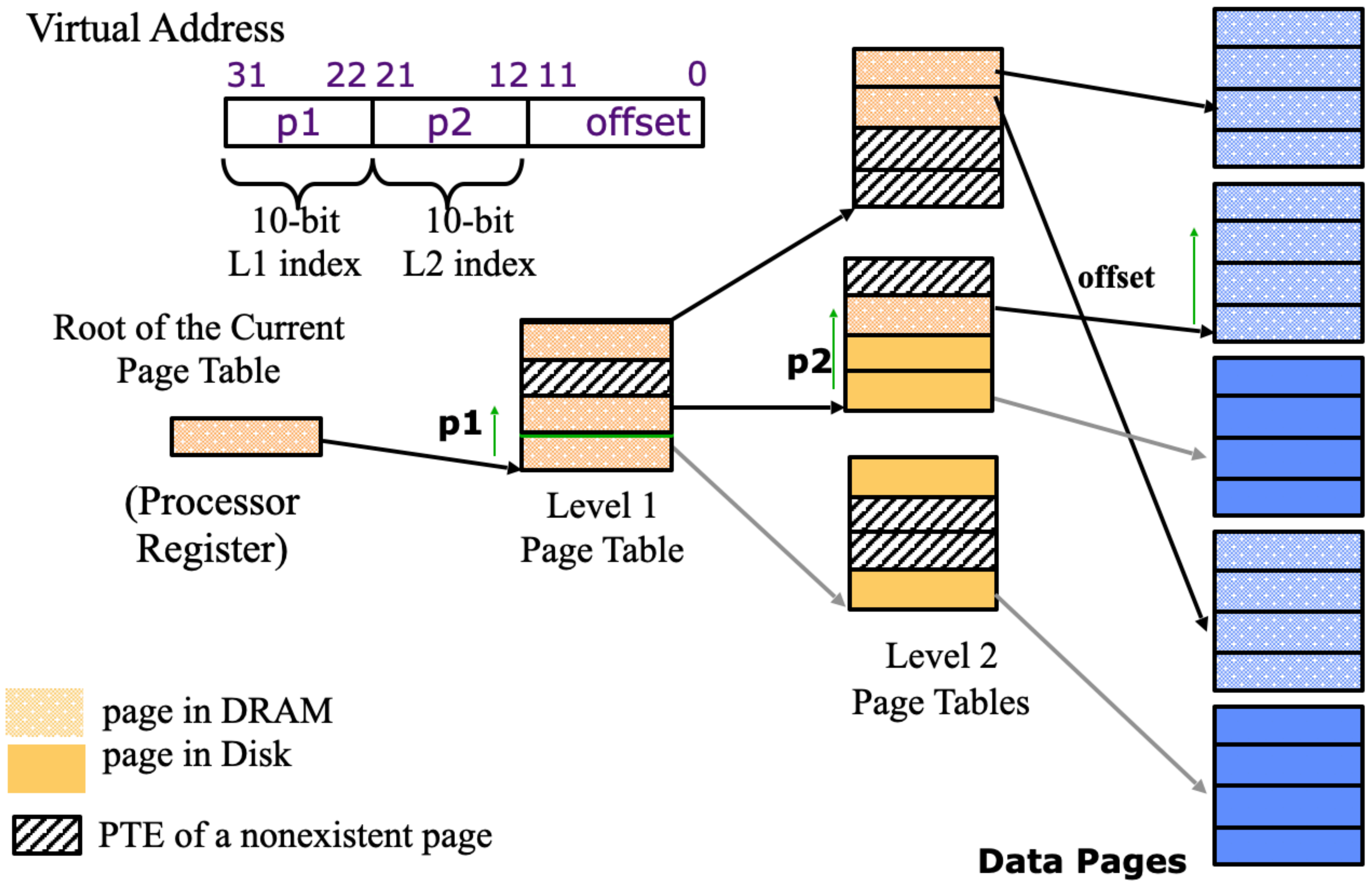
Hierarchical Page Table

- How to store such large page tables efficiently?
- How to quickly search page tables?
- Intuition: Do you ever allocate 32 petabytes of memory?? — no!!
- We should not store anything for a virtual address which maps to NONE
- Idea!! What if we only store a linked list of valid pages??

Hierarchical Page Table

- How to store such large page tables efficiently?
- How to quickly search page tables?
- Intuition: Do you ever allocate 32 petabytes of memory?? — no!!
- We should not store anything for a virtual address which maps to NONE
- Idea!! What if we only store a linked list of valid pages?? — bad idea; search complexity O(N).

Hierarchical Page Table



Hierarchical Page Table

- There can be multiple levels of the page table.
- Level 1 page table contains the address of the Level 2 table and so on..
- Let’s quantify the saving in space…
- Let’s consider a 4 level page table….

Hierarchical Page Table

- Page size is 4KiB
- L1 contains (8 byte) entries of L2 table — how many entries in L1?
  - 4 KiB / 8 bytes per PTE = 512 entries — each level must fit in one page
  - 512 entries ==> 9 bits for indexing
- L2 again contains 512 entries == > 9 bits indexing and so on…
- So the addressing looks like this…

```
63          47      38      29      20      11          0
+-----------+-------+-------+-------+-------+-----------+
| UNUSED    |  L1   |  L2   |  L3   |  L4   | offset    |
+-----------+-------+-------+-------+-------+-----------+
            |9 bits |9 bits |9 bits |9 bits | 12 bits
```

Hierarchical Page Table

```
63         47      38      29      20      11          0
+----------+-------+-------+-------+-------+-----------+
| UNUSED   |  L1   |  L2   |  L3   |  L4   | offset    |
+----------+-------+-------+-------+-------+-----------+
           |9 bits|9 bits|9 bits|9 bits| 12 bits
```

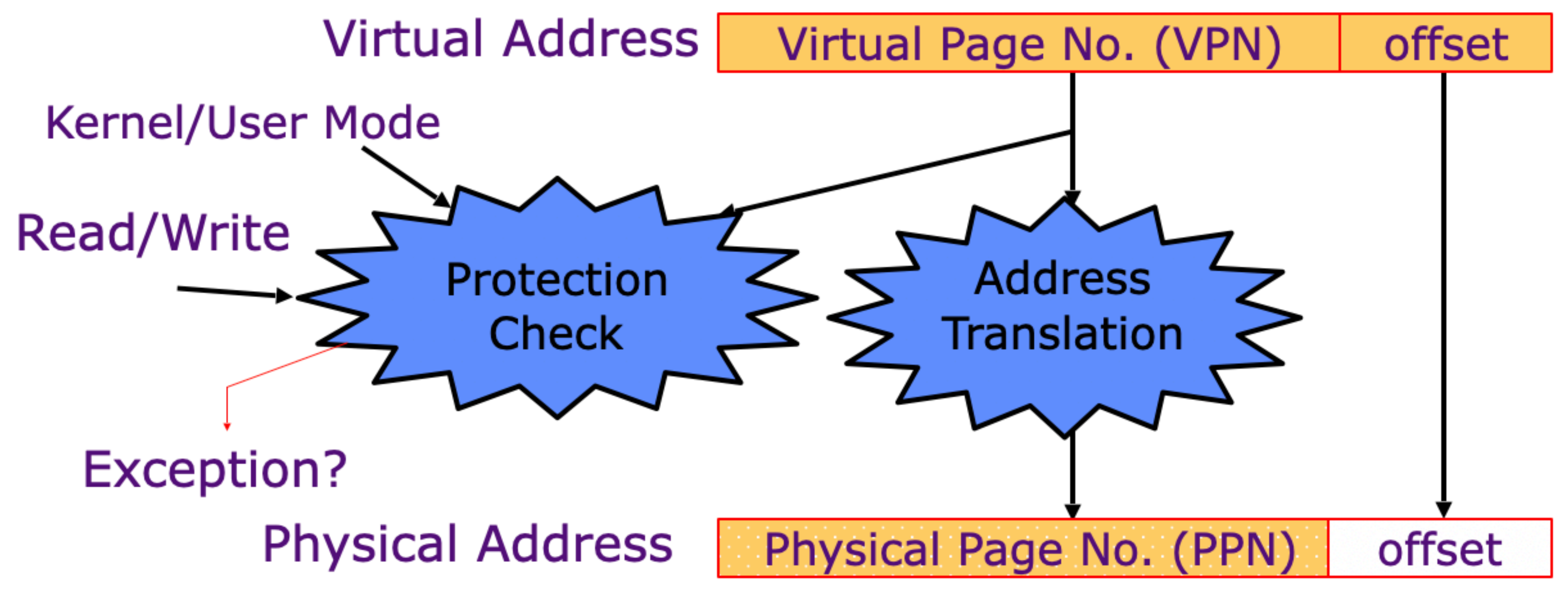
- Only 48 address bits are used (X86-64)
- There is 1 L1 table, 512 possible L2 tables, 512^3 possible L3 tables, and 512^4 possible L4 tables
- L4 actually holds the physical address…
- Note: Each level of table contains the physical address of the next level

Hierarchical Page Table Walk

```
63         47      38      29      20      11         0
+----------+-------+-------+-------+-------+----------+
| UNUSED   |  L1   |  L2   |  L3   |  L4   | offset   |
+----------+-------+-------+-------+-------+----------+
           |9 bits|9 bits|9 bits|9 bits| 12 bits
```

- Use the address in the *page table register (cr3)* to find the L1 page table address
  - Actually this register contains the physical address of the L1 page table
- Use the L1 index from the virtual address to get the L2 page table address
- Use the L2 page table address and L2 index to get a L3 page table address
- Use the L3 page table address and L3 index to get a L4 page table address
- Use the L4 page table address and L4 index to get the destination physical page
- Use the destination physical page and the offset to get actual physical address within that destination physical page

Address Translation and Protection



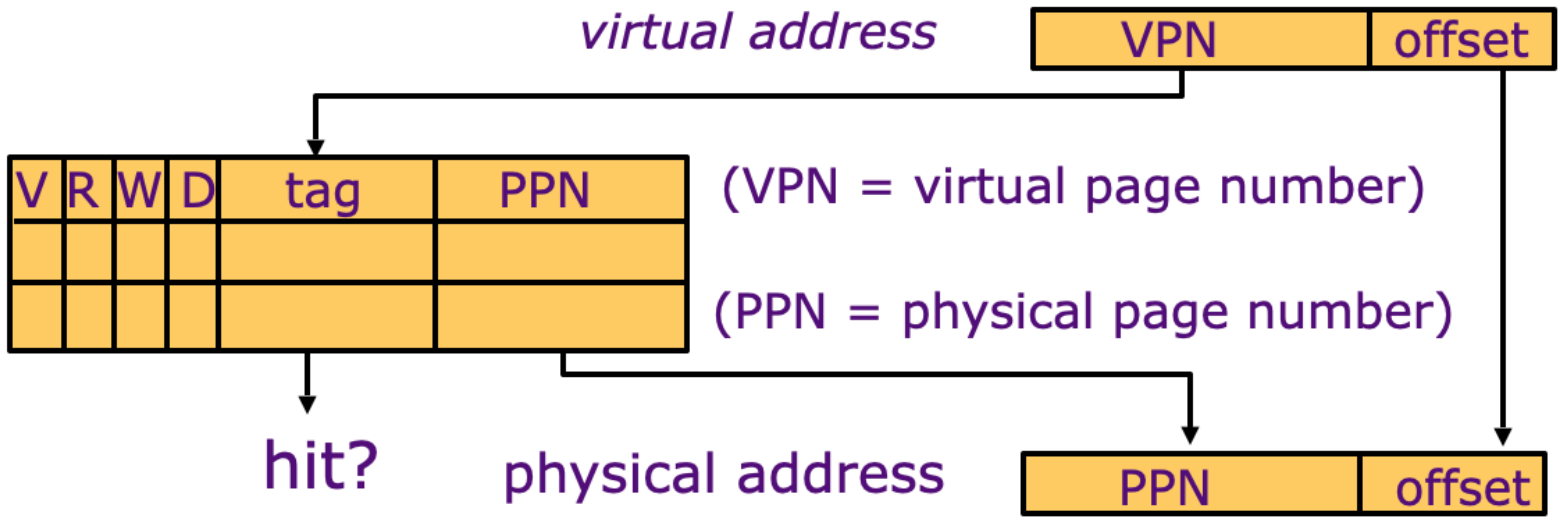
- Every address translation goes through a protection check

Translation Look Aside Buffer

- Address translation is extremely costly
  - DRAM access
  - Also disk access which is orders of magnitude slower than DRAM.
  - How frequently you need to do it?
    - Depends on the size of each page
  - How to make it faster?

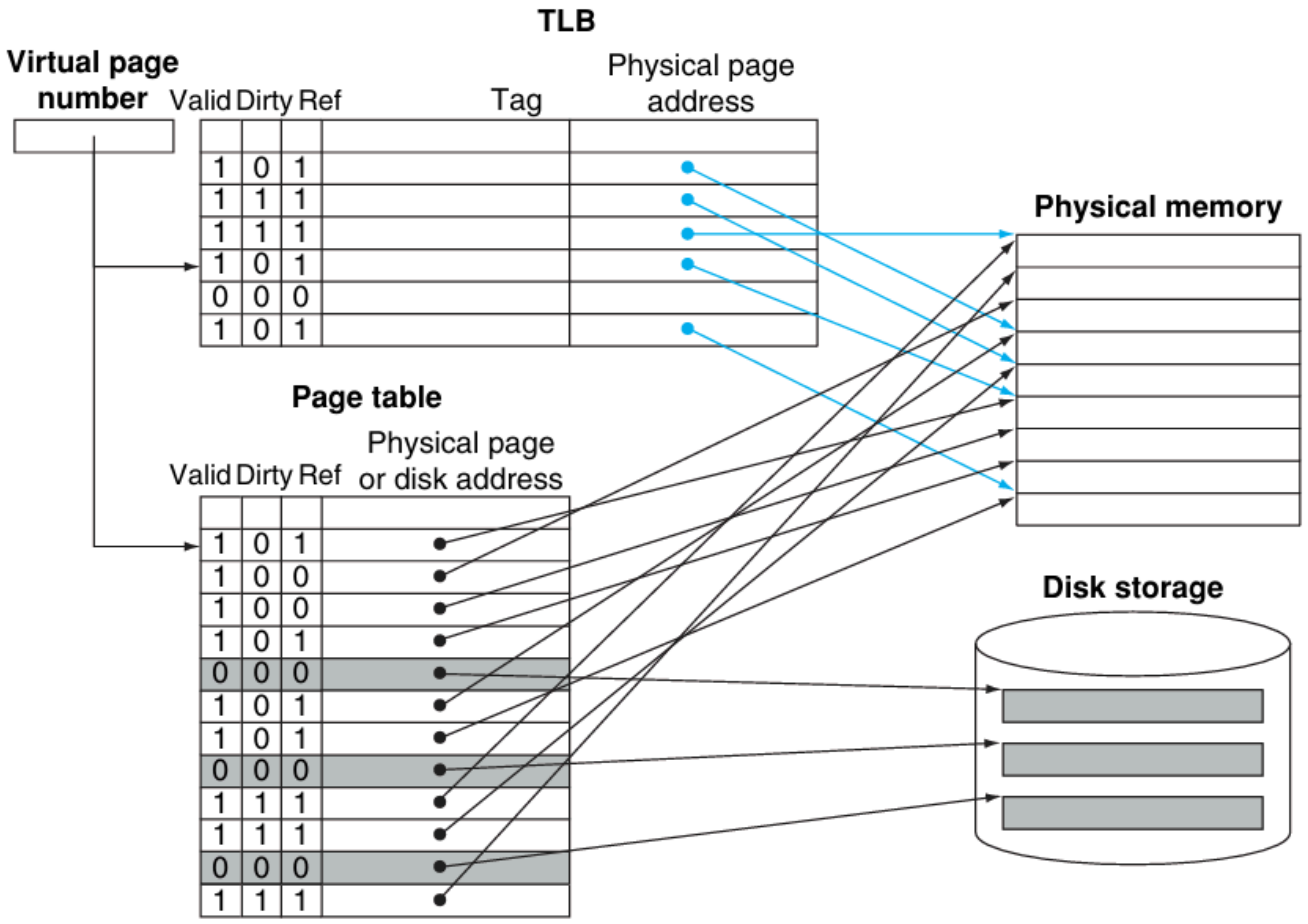
Translation Look Aside Buffer

TLB hit ⇒ *Single Cycle Translation*
TLB miss ⇒ *Page Table Walk to refill*



- Simple Solution: Use a Cache

Translation Look Aside Buffer — More Elaborate



Translation Look Aside Buffer — More Elaborate

- Typically 32-128 entries, usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages
    - more likely that two entries conflict–
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs
- Random or FIFO replacement policy
- No process information in TLB? — usually TLB is flushed in context switch..
- TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB

How to Handle a TLB Miss

Software (MIPS, Alpha)

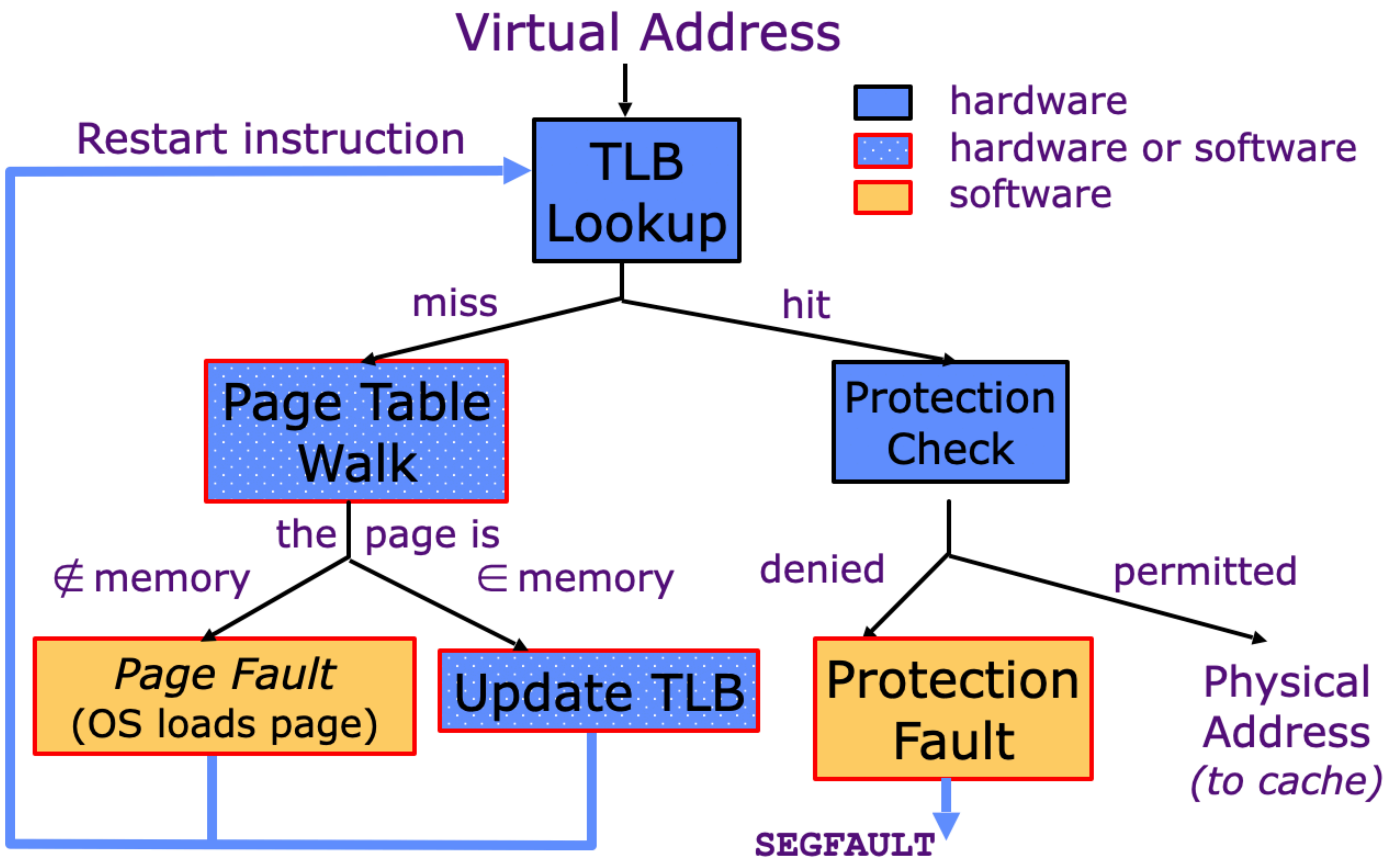
TLB miss causes an exception and the operating system walks the page tables and reloads TLB. *A privileged "untranslated" addressing mode used for walk*

Hardware (SPARC v8, x86, PowerPC)

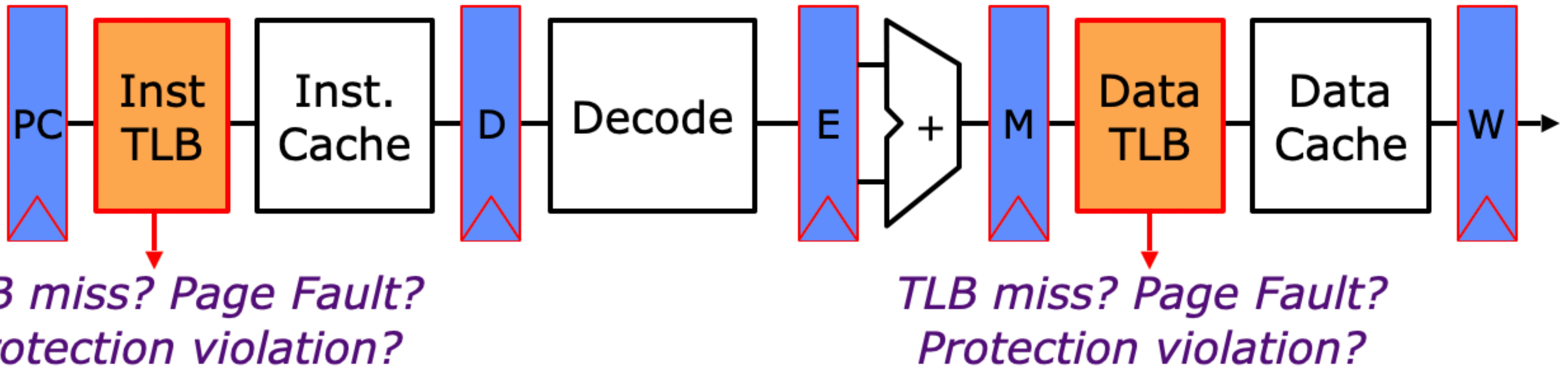
A memory management unit (MMU) walks the page tables and reloads the TLB

If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page-Fault exception for the original instruction

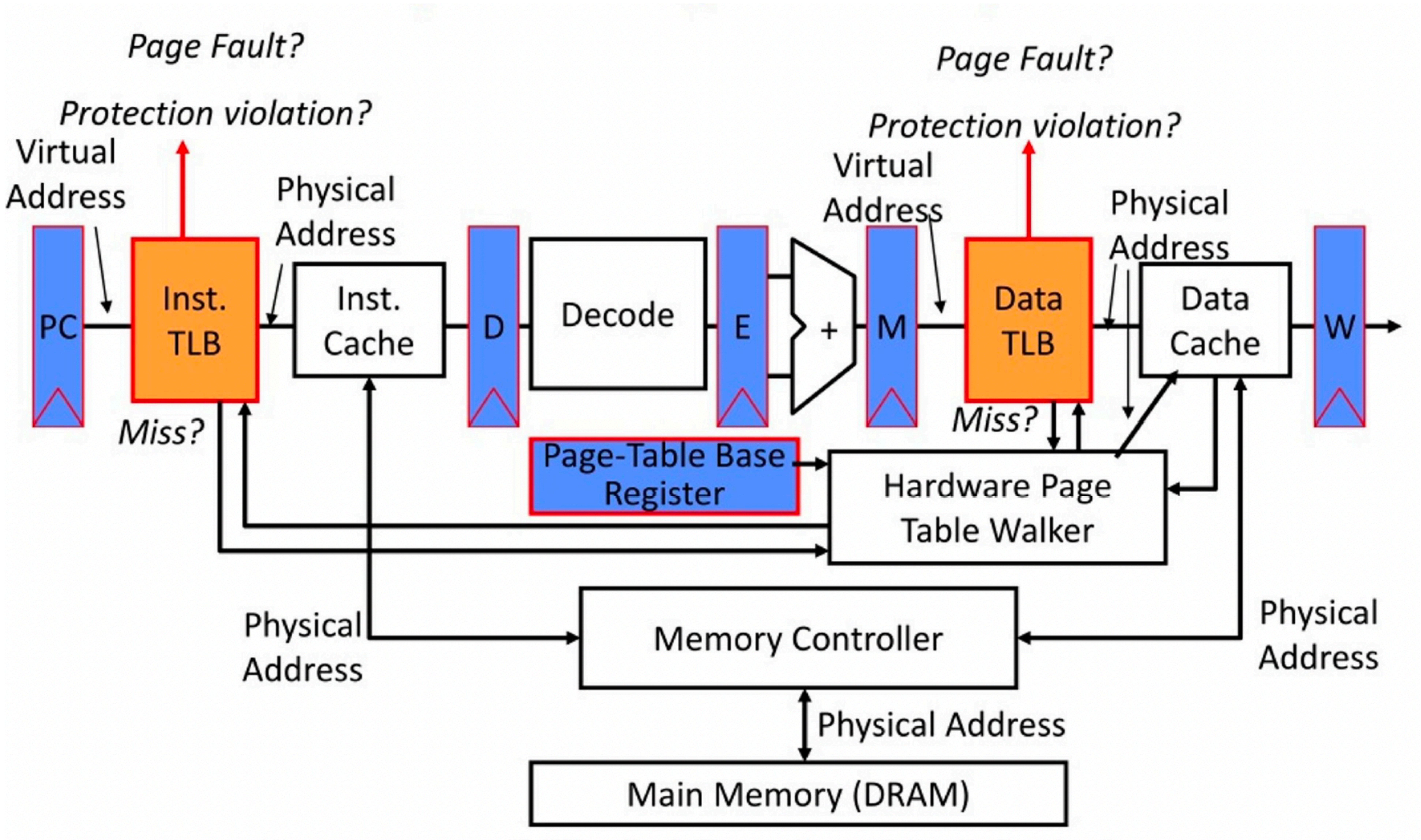
The Entire Picture



TLB, Cache and Pipeline

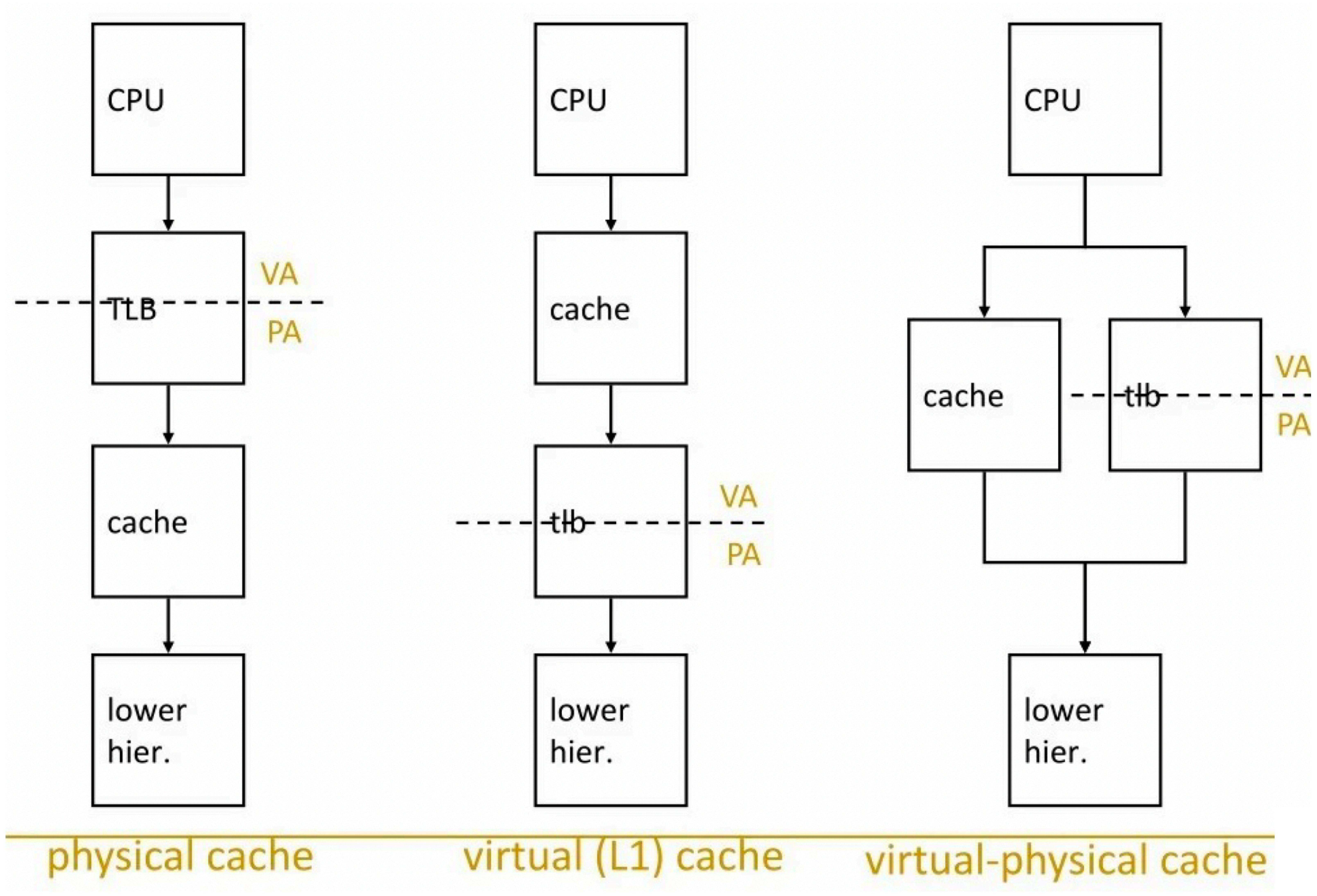

PC
Inst TLB
Inst. Cache
D
Decode
E
+
M
Data TLB
Data Cache
W
TLB miss? Page Fault? Protection violation?
TLB miss? Page Fault? Protection violation?

TLB, Cache and Pipeline



Virtual Caches

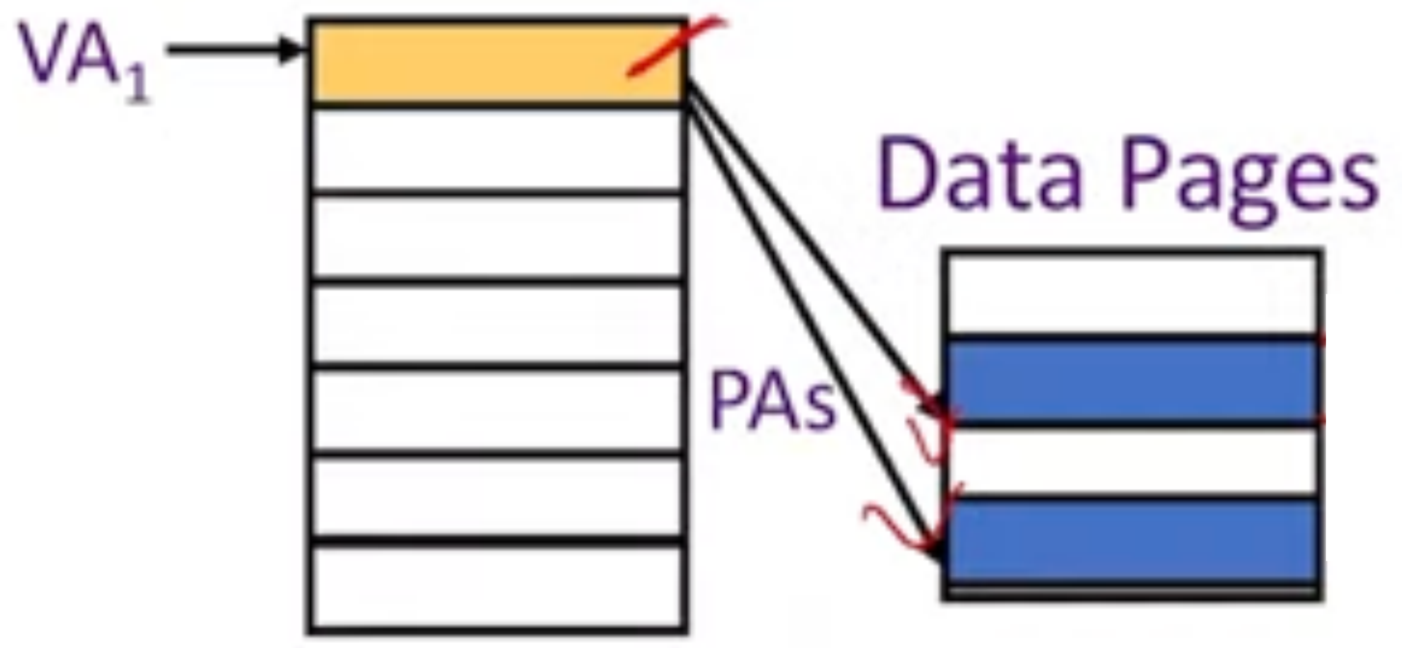
How Virtual Memory Interacts with Cache



Homonym and Synonym

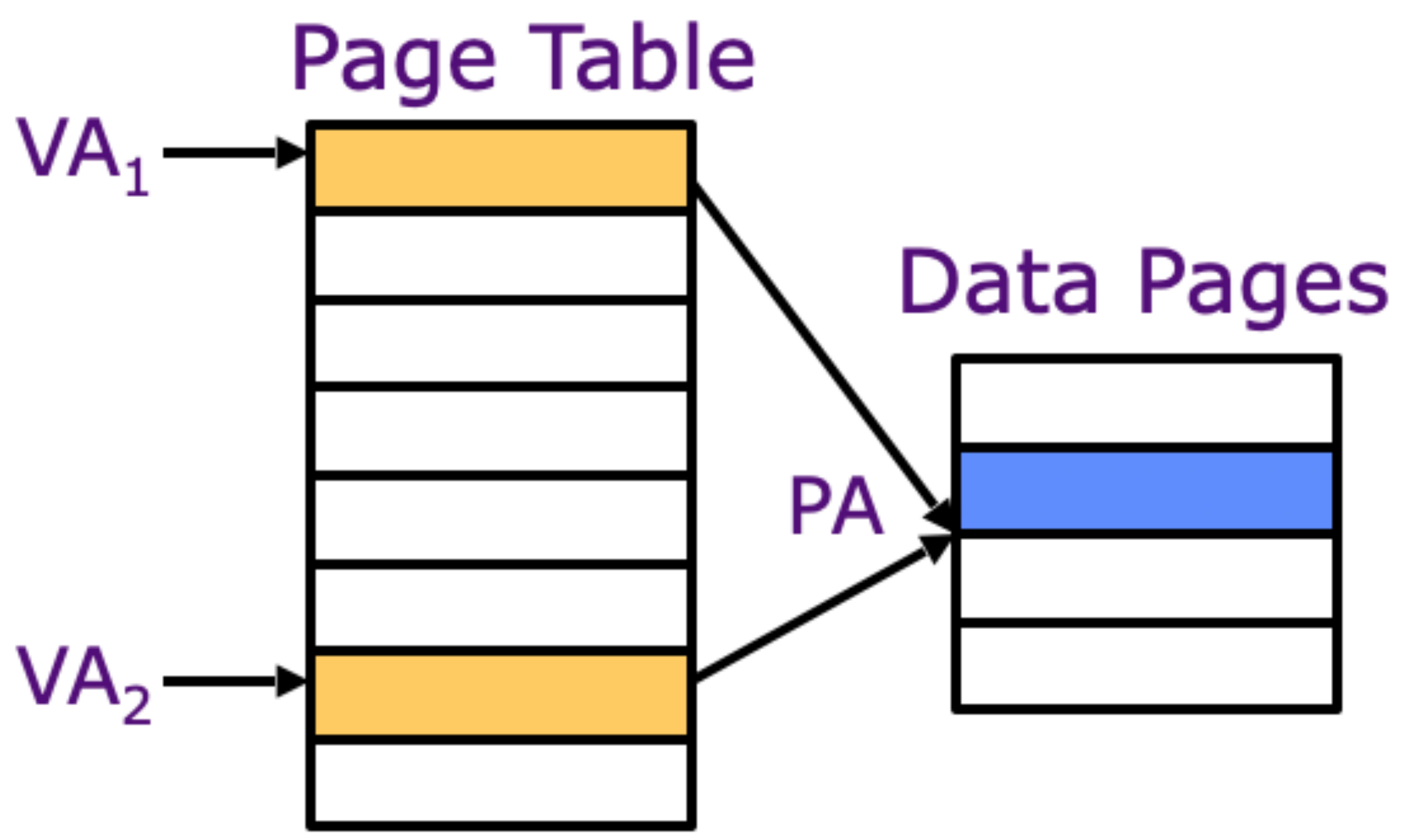
Homonym

- Same VA can map to two different PA
- VAs are from two different processes
- Tag may not uniquely identify the cache data
- Solution:
  - Add process ID
  - Flush cache on context switch
  - Anything else?

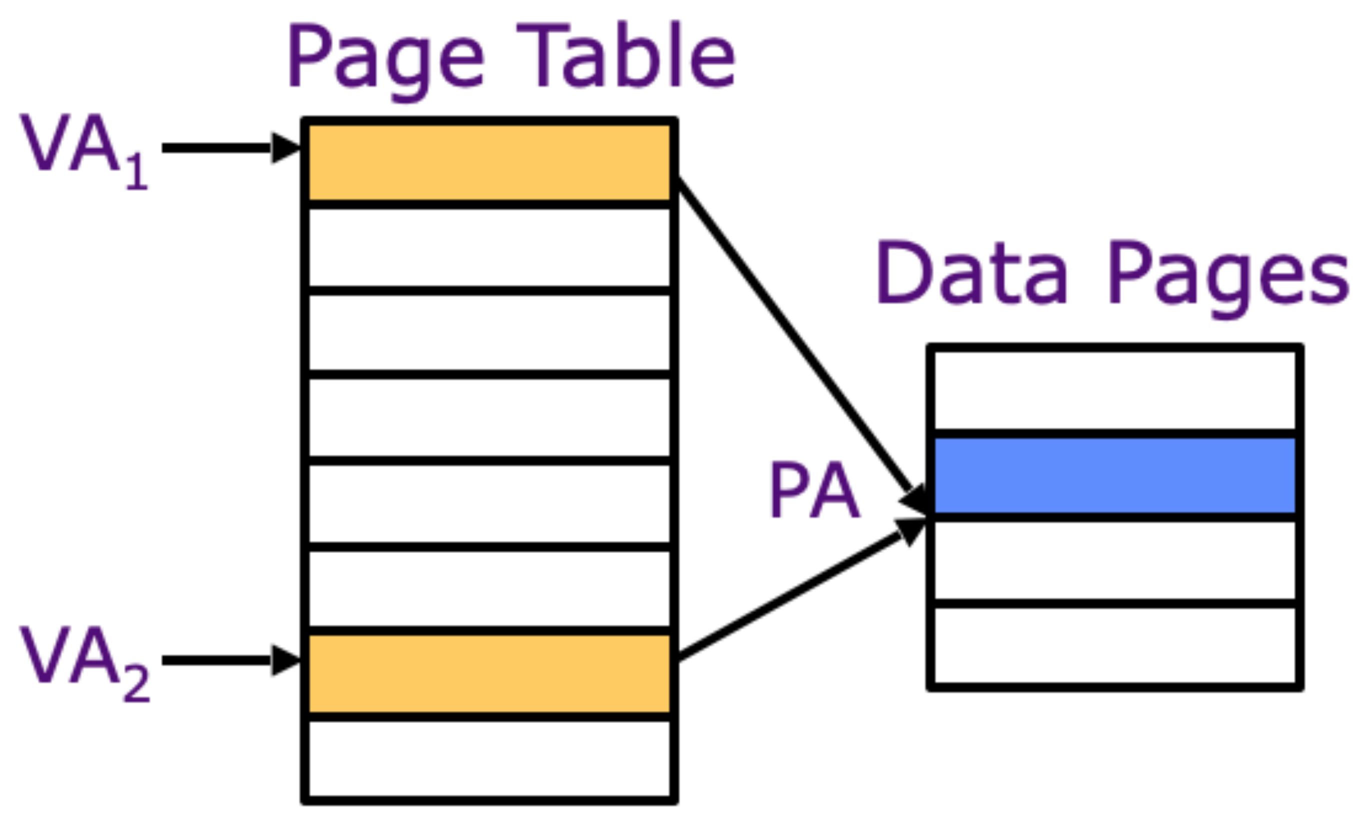


Synonym (or Alising)

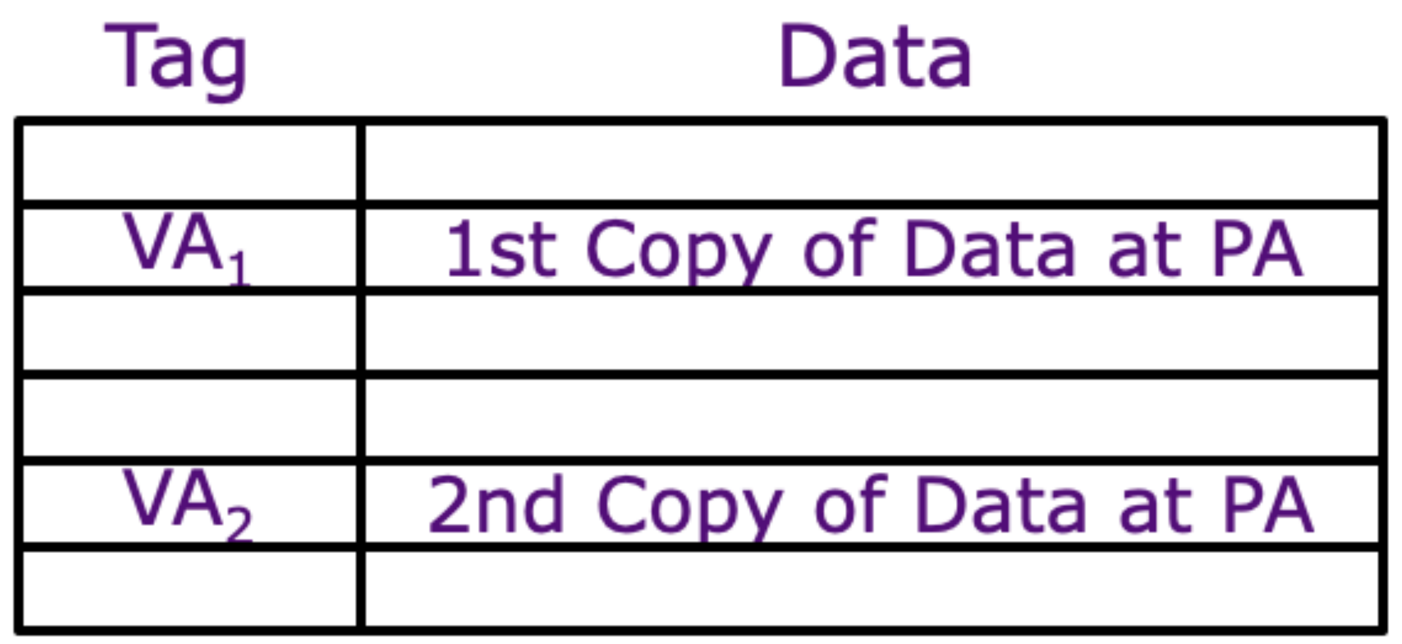
- Different VAs can map to the same PA
- Shared library, shared data



Issues with Synonym



Two virtual pages share one physical page



Tag	Data
VA_1	1st Copy of Data at PA
VA_2	2nd Copy of Data at PA

Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

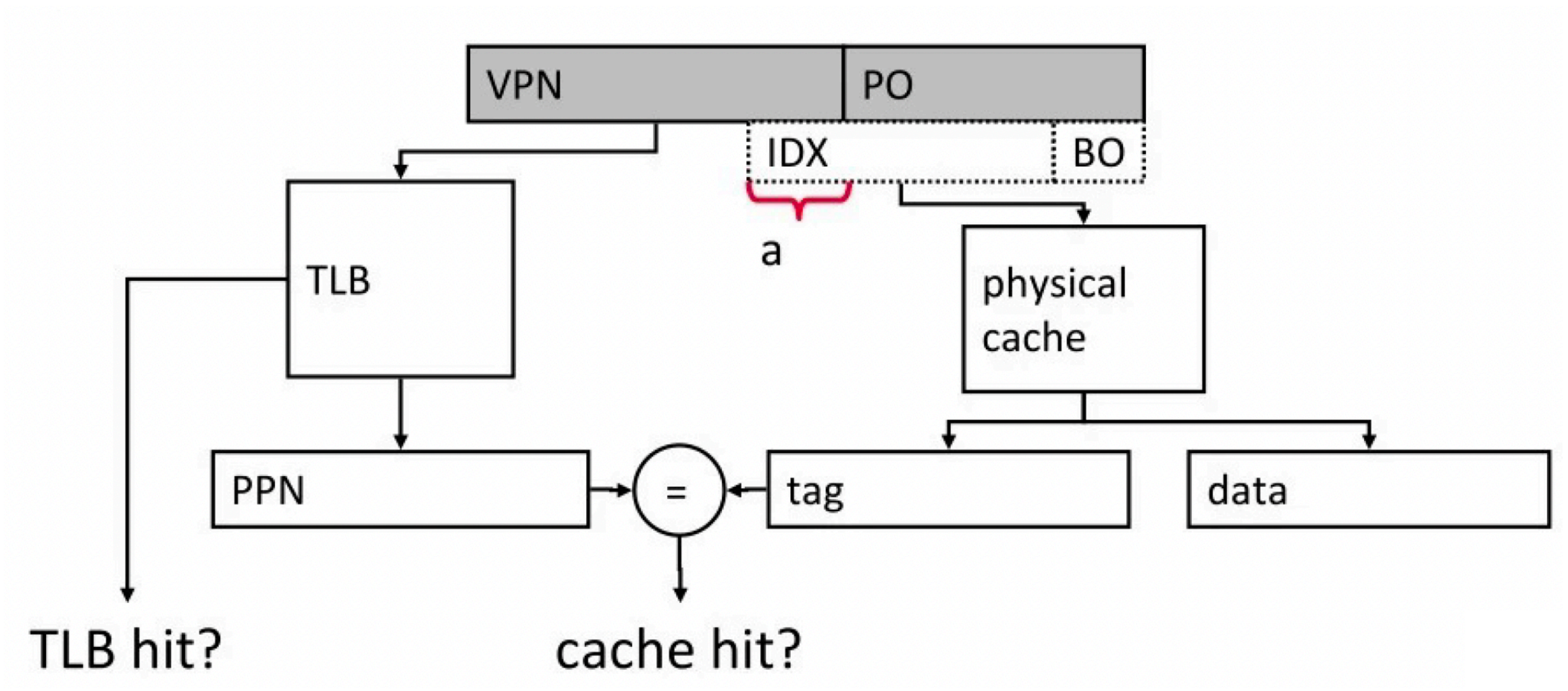
<u>Synonym (or Alising)</u>

- Different VAs can map to the same PA
- <u>Solution:</u>
  - Disallow alisas coexist in the cache
  - Direct mapped cache
  - Anything else?

Virtually Indexed Physically Tagged Cache

Idea

- Perform (L1) indexing based on virtual address
- Simultaneously, look into the TLB for the address translation and get the physical tag bits.
- Perform the tag check based on the physical tag.

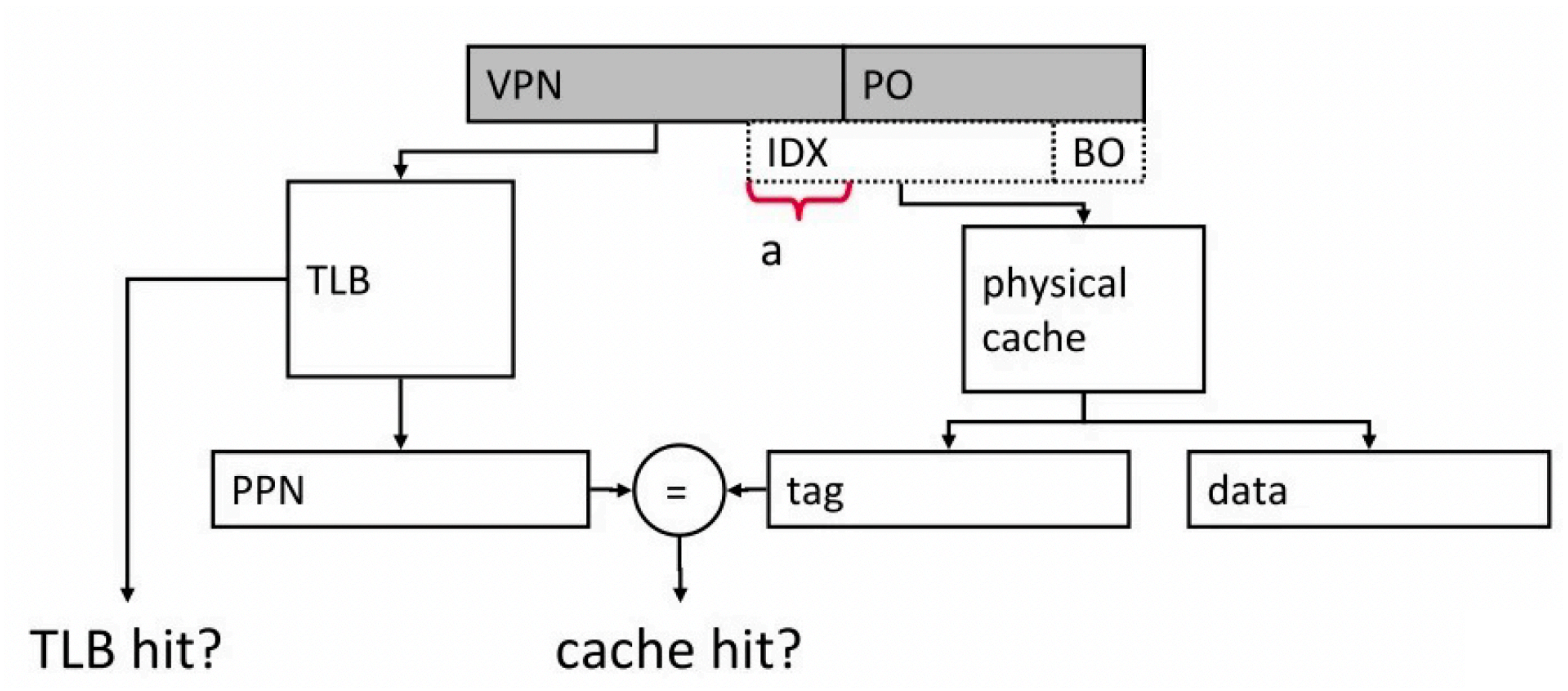


Virtually Indexed Physically Tagged Cache: Catch

Idea

- If $|C| < |page| \times associativity$, then the index bits comes from the page offset — page offset is same in VA and PA

- What does it solve?

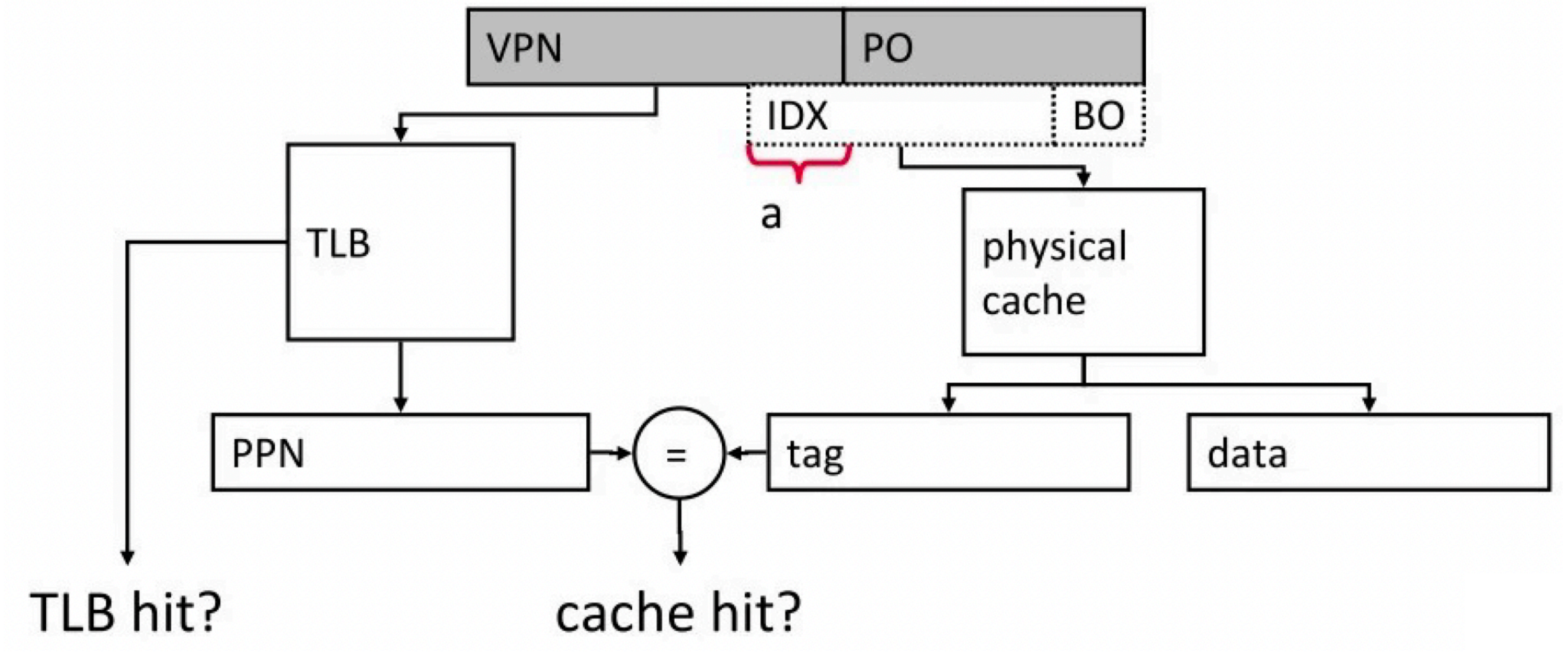


Virtually Indexed Physically Tagged Cache: Catch

Idea

- If $|C| < |page| \times associativity$, then the index bits comes from the page offset — page offset is same in VA and PA

- What does it solve? — Homonym, even if the VAs are the same, physical tags for two different processes will never match.

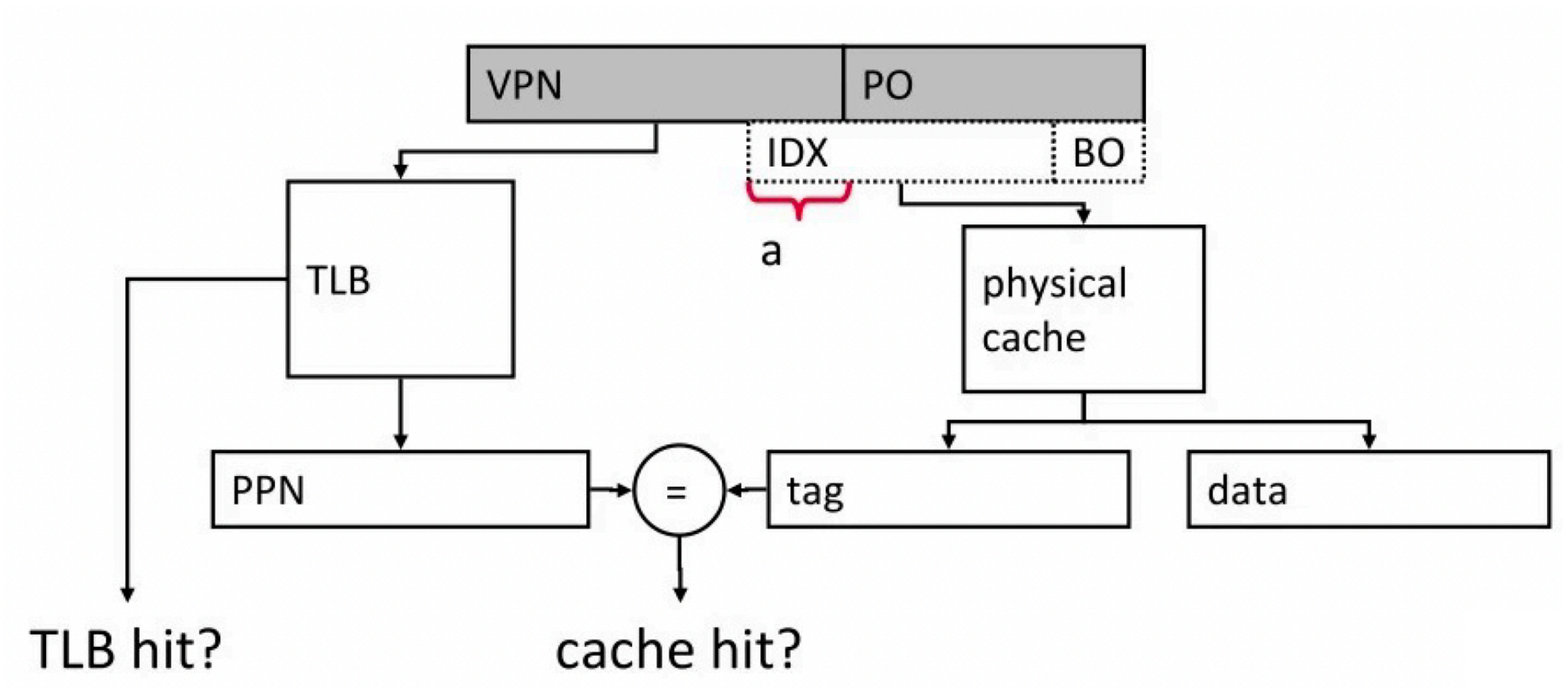


Virtually Indexed Physically Tagged Cache: Catch

Idea

- If $|C| < |page| \times associativity$, then the index bits comes from the page offset — page offset is same in VA and PA

- What about synonym?

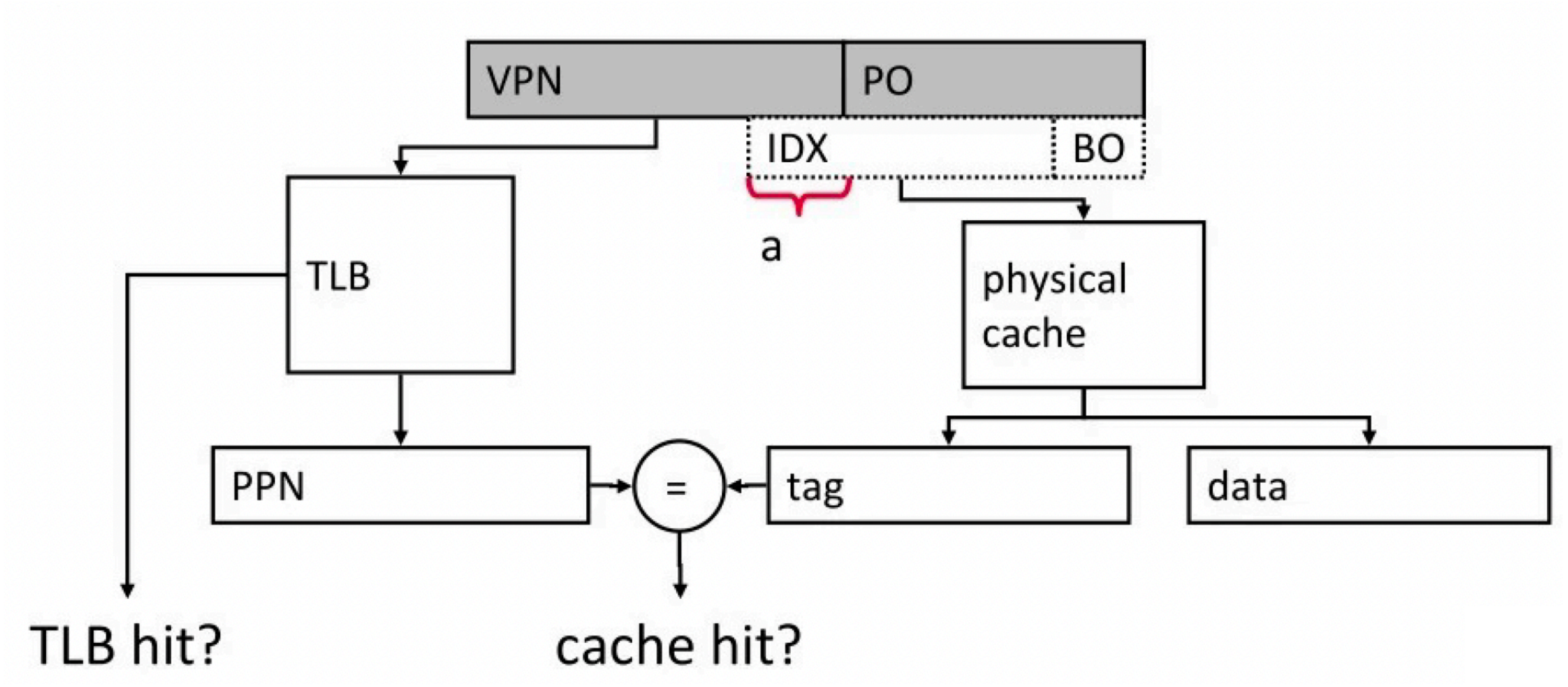


Virtually Indexed Physically Tagged Cache: Catch

<u>Idea</u>

- If $|C| < |page| \times associativity$, then the index bits comes from the page offset — page offset is same in VA and PA

- What about synonym? — it is still there, but..

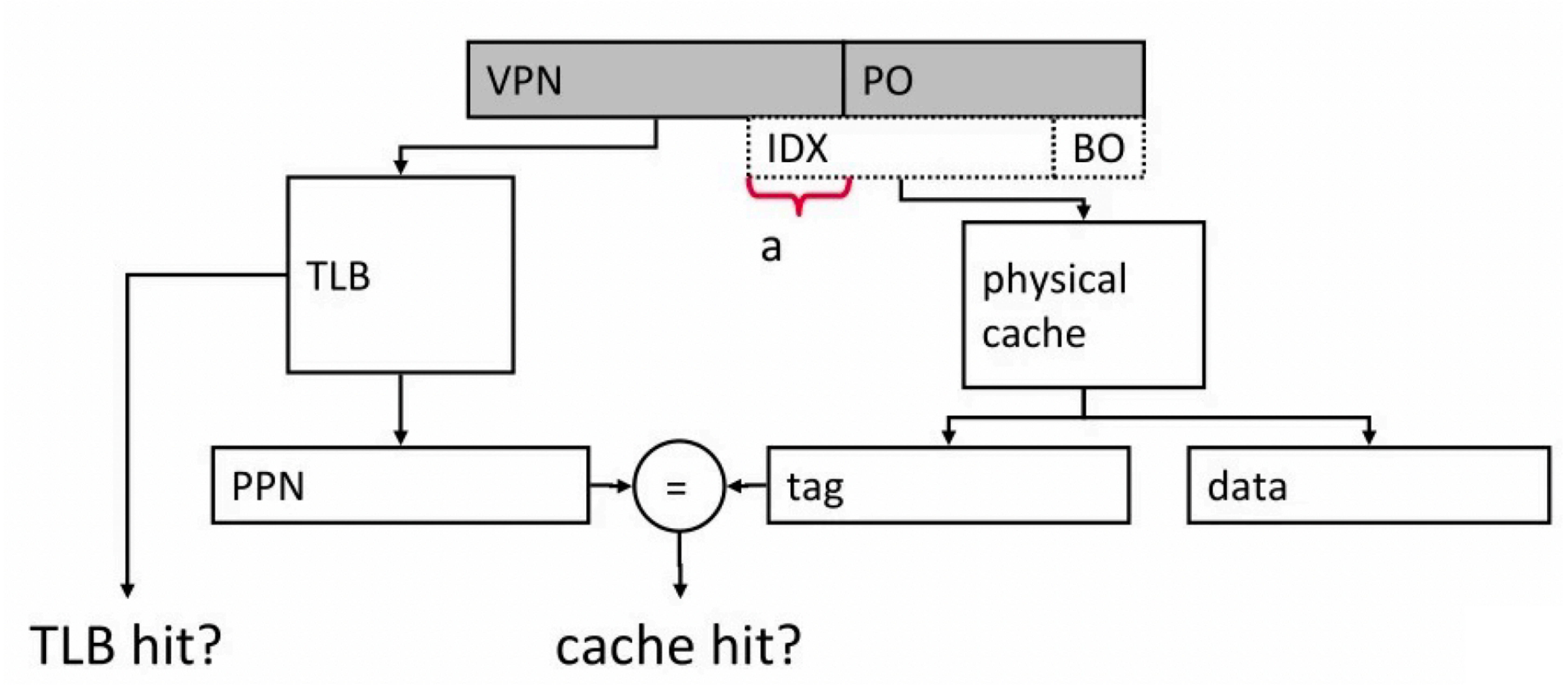


Virtually Indexed Physically Tagged Cache: Catch

<u>Idea</u>

- If $|C| < |page| \times associativity$, then the index bits comes from the page offset — page offset is same in VA and PA

- What about synonym? — it is still there, it will never happen if the indexing happens from page offset bits only



Overall…

VIVT

- Fastest — but homonym and synonym

PIPT

- Classical

VIPT

- No homonyms; synonyms can also be managed if small cache

PIVT?

- Extremely slow — cannot overlap TLB and cache access
- Homonym — as virtually tagged
- Synonym??

Book

P & H, Chapter 5