



# Virtual Memory

## Process Abstraction, Part 2: Private Address Space

**Motivation:** why not direct physical memory access?

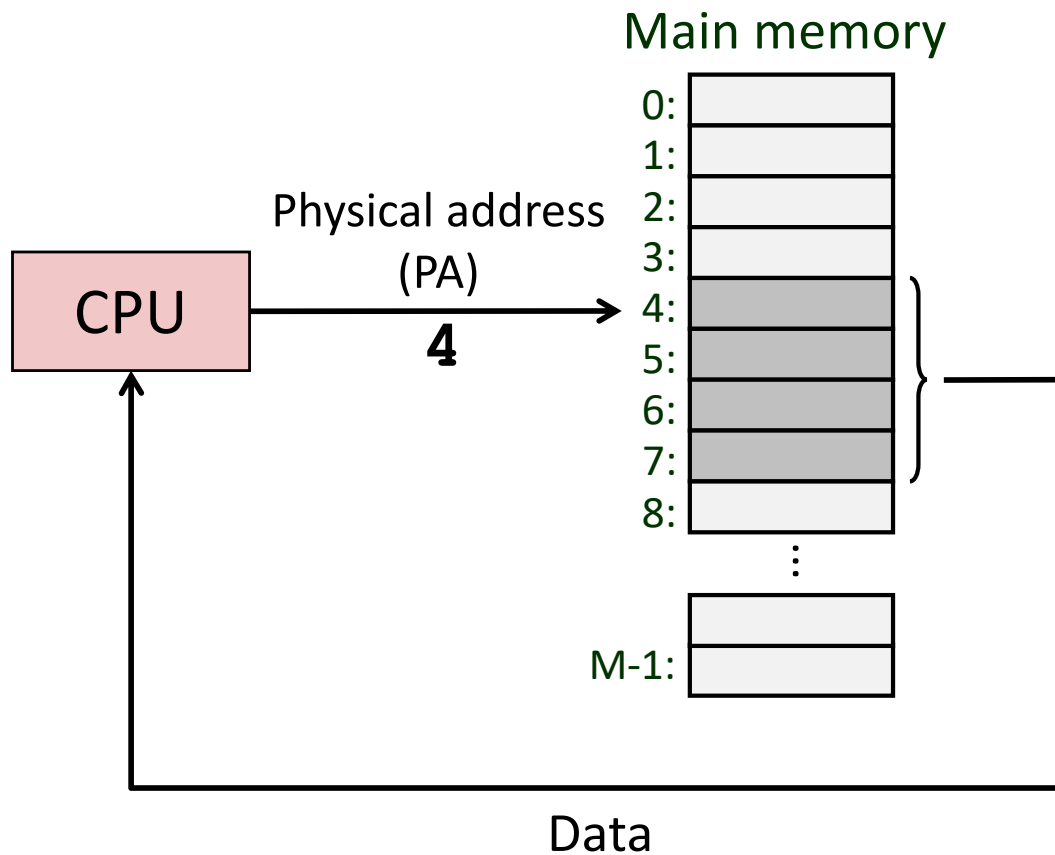
**Address translation** with pages

**Optimizing translation:** translation lookaside buffer

**Extra benefits:** sharing and protection

Memory as a contiguous array of bytes is a lie! Why?

# Problems with physical addressing



# Problem 1: memory management

Process 1  
Process 2  
Process 3  
...  
Process n



stack  
heap  
code  
globals  
...



Main memory



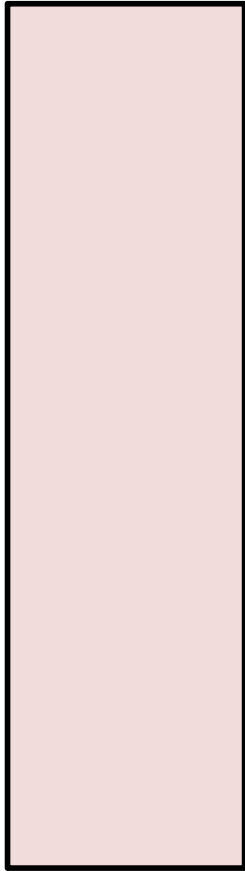
Also:

**Context switches** must swap out entire memory contents.

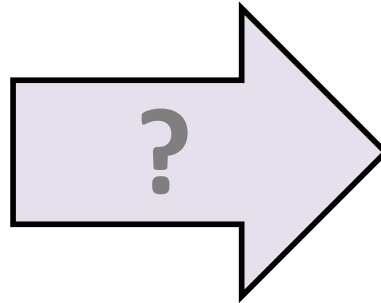
Isn't that **expensive**?

# Problem 2: capacity

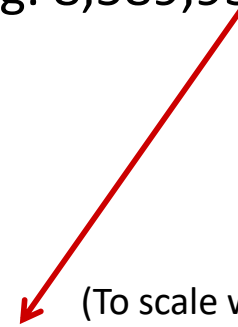
64-bit addresses can address  
several exabytes  
(18,446,744,073,709,551,616 bytes)



1 virtual address space per process,  
with many processes...

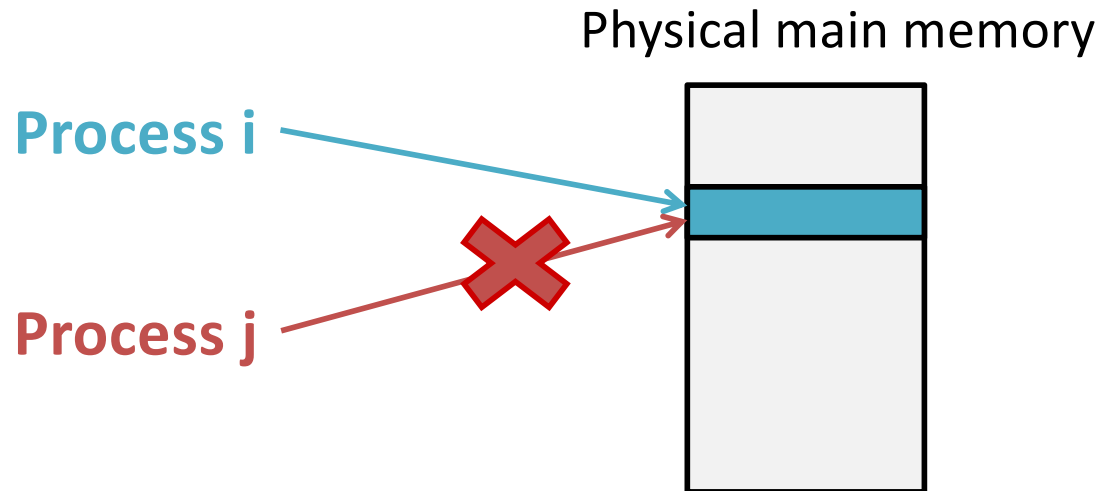


Physical main memory offers  
a few gigabytes  
(e.g. 8,589,934,592 bytes)

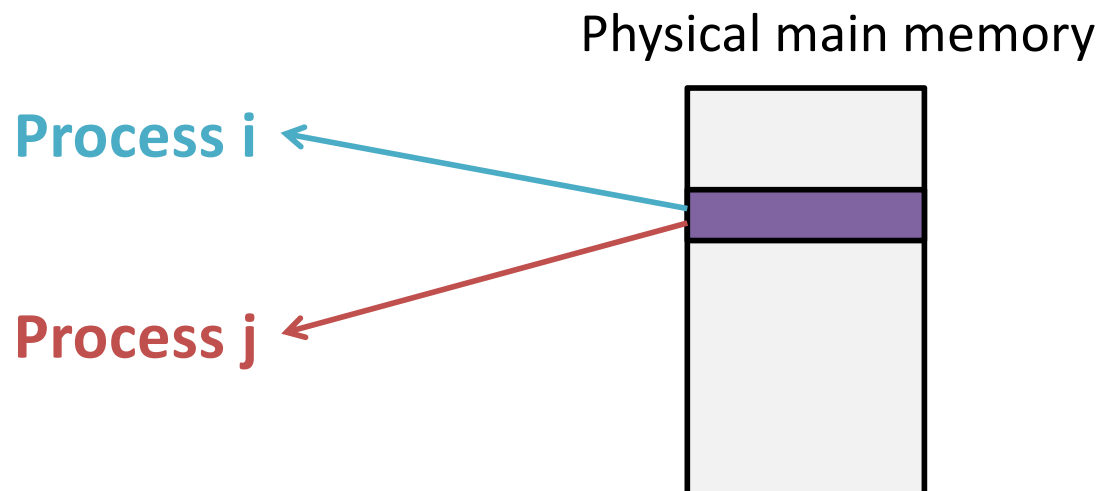


(To scale with 64-bit address space,  
you can't see it.)

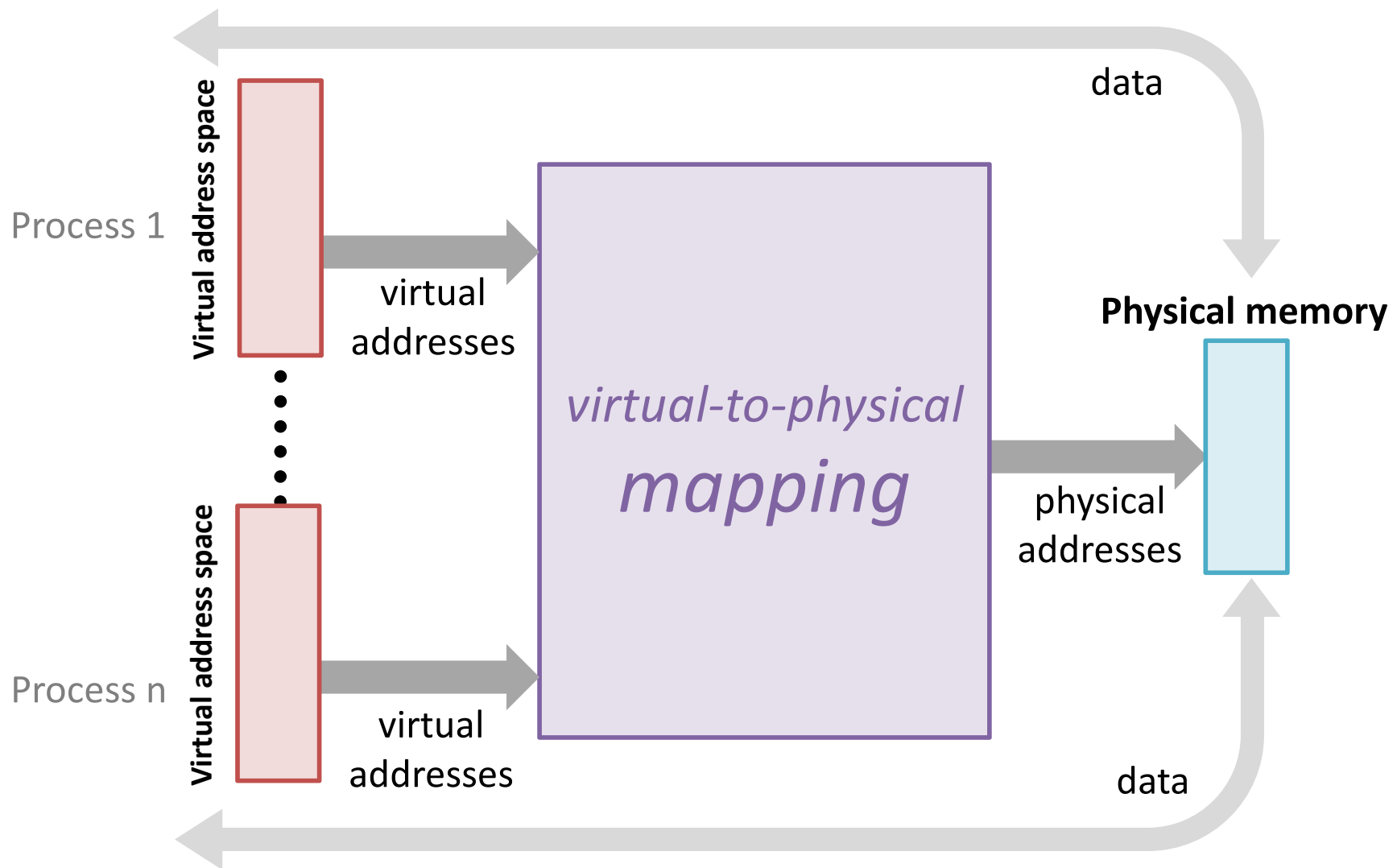
# Problem 3: protection



# Problem 4: sharing



# Solution: Virtual Memory (address *indirection*)



**Private virtual address space per process.**

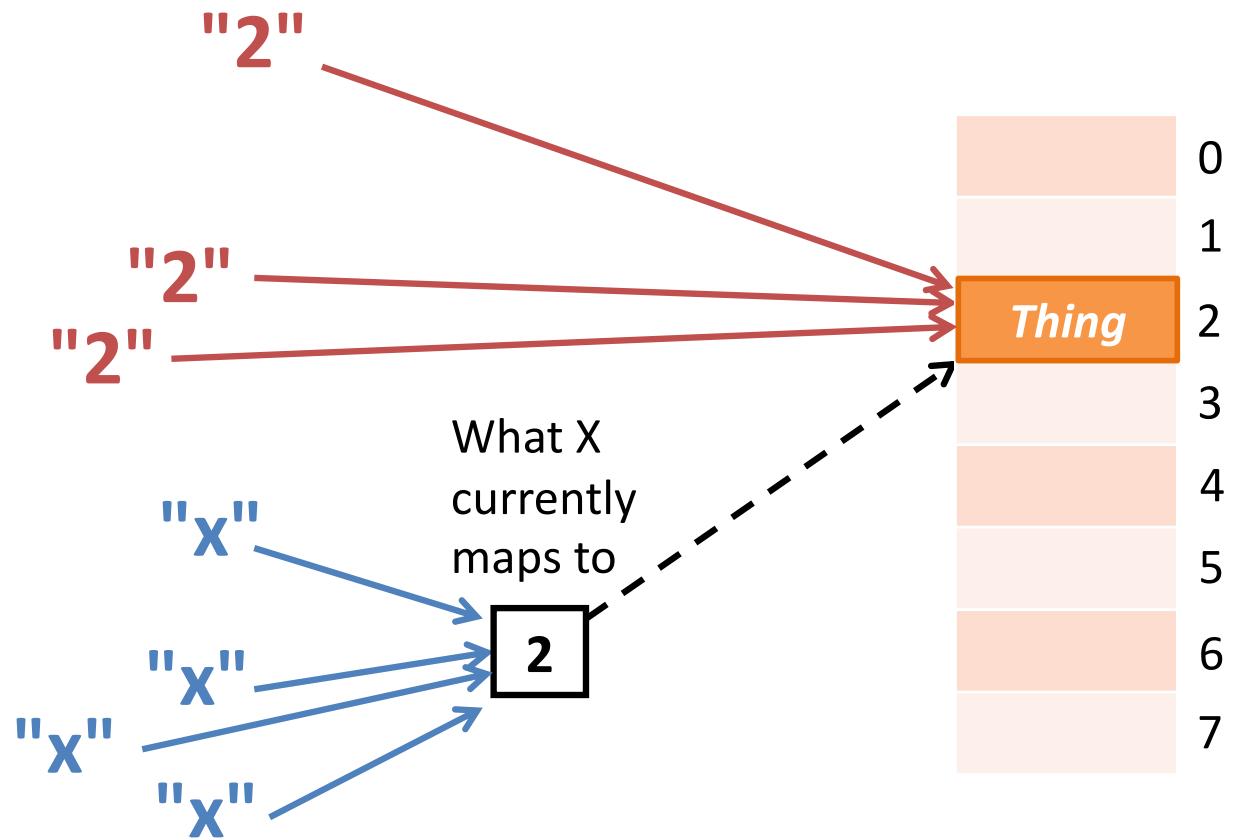
**Single physical address space managed by OS/hardware.**

# Indirection

(it's everywhere!)

Direct naming

Indirect naming



What if we move *Thing*?

# Tangent: **indirection everywhere**

- Pointers
- Constants
- Procedural abstraction
- Domain Name Service (DNS)
- Dynamic Host Configuration Protocol (DHCP)
- Phone numbers
- 911
- Call centers
- Snail mail forwarding

**“Any problem in computer science can be solved by adding another level of indirection.”**

*–David Wheeler, inventor of the subroutine, or Butler Lampson*

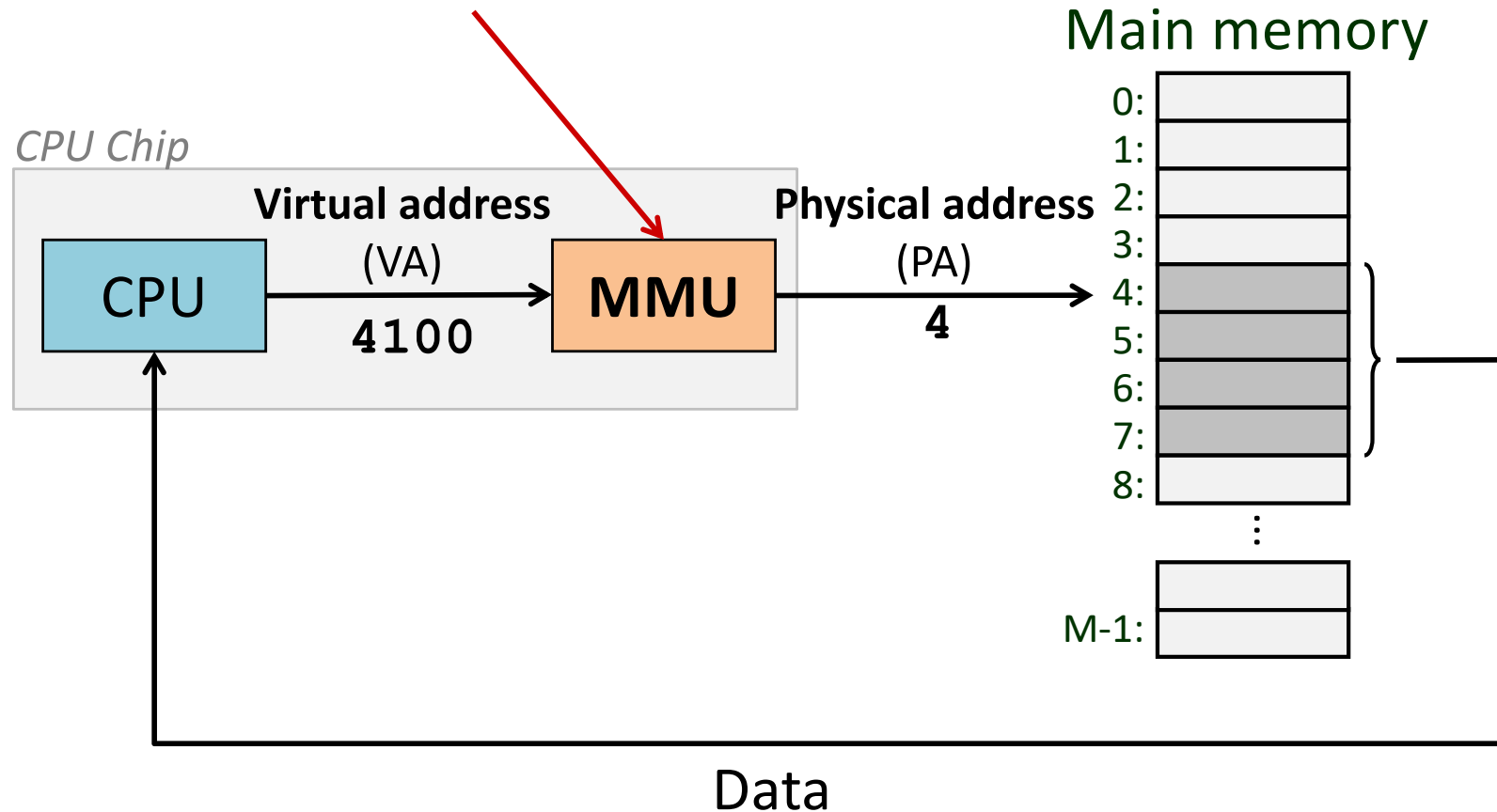
Another Wheeler quote? "Compatibility means deliberately repeating other people's mistakes."



# Virtual addressing and address translation

## Memory Management Unit

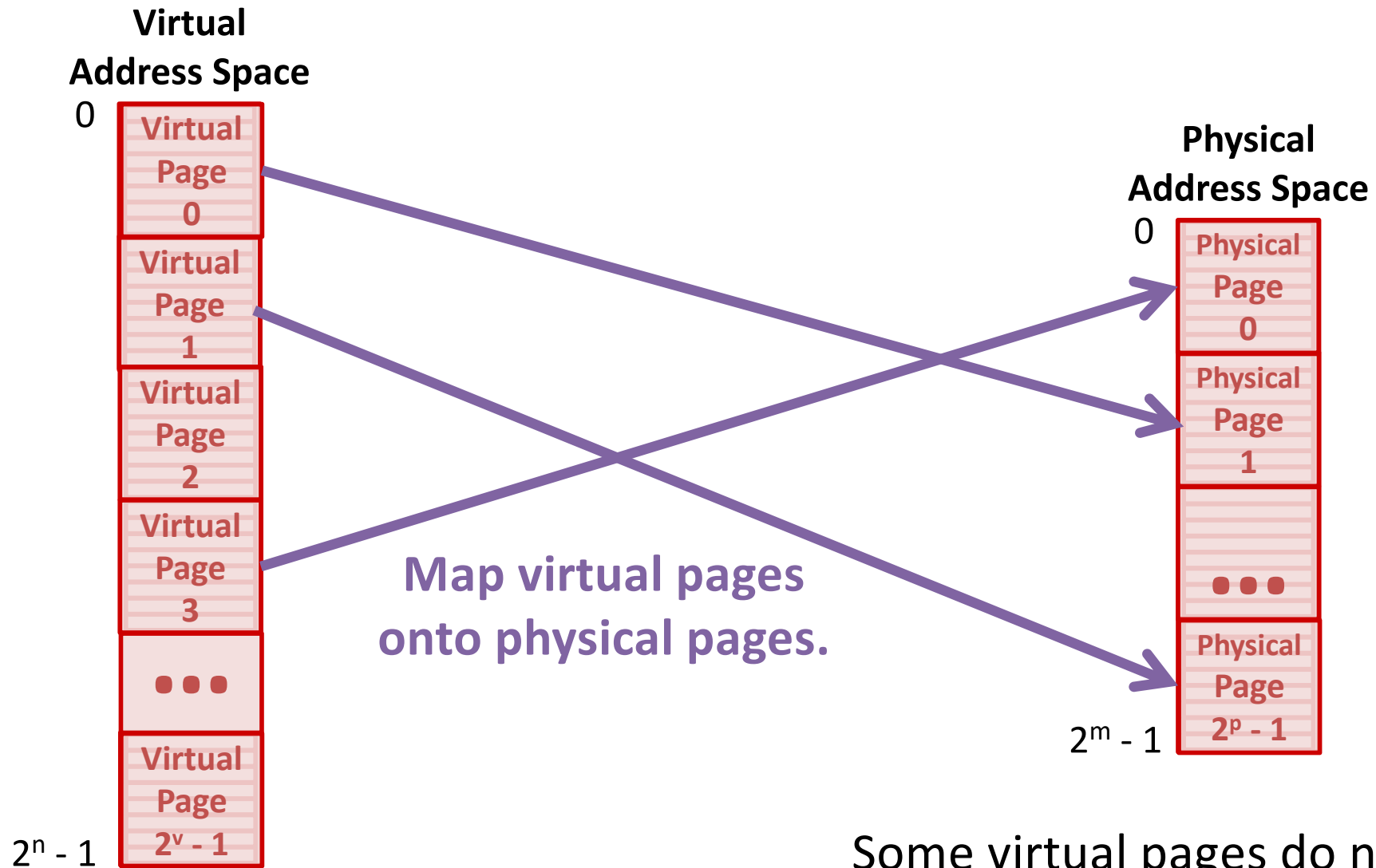
translates virtual address to physical address



Physical addresses are *invisible* to programs.

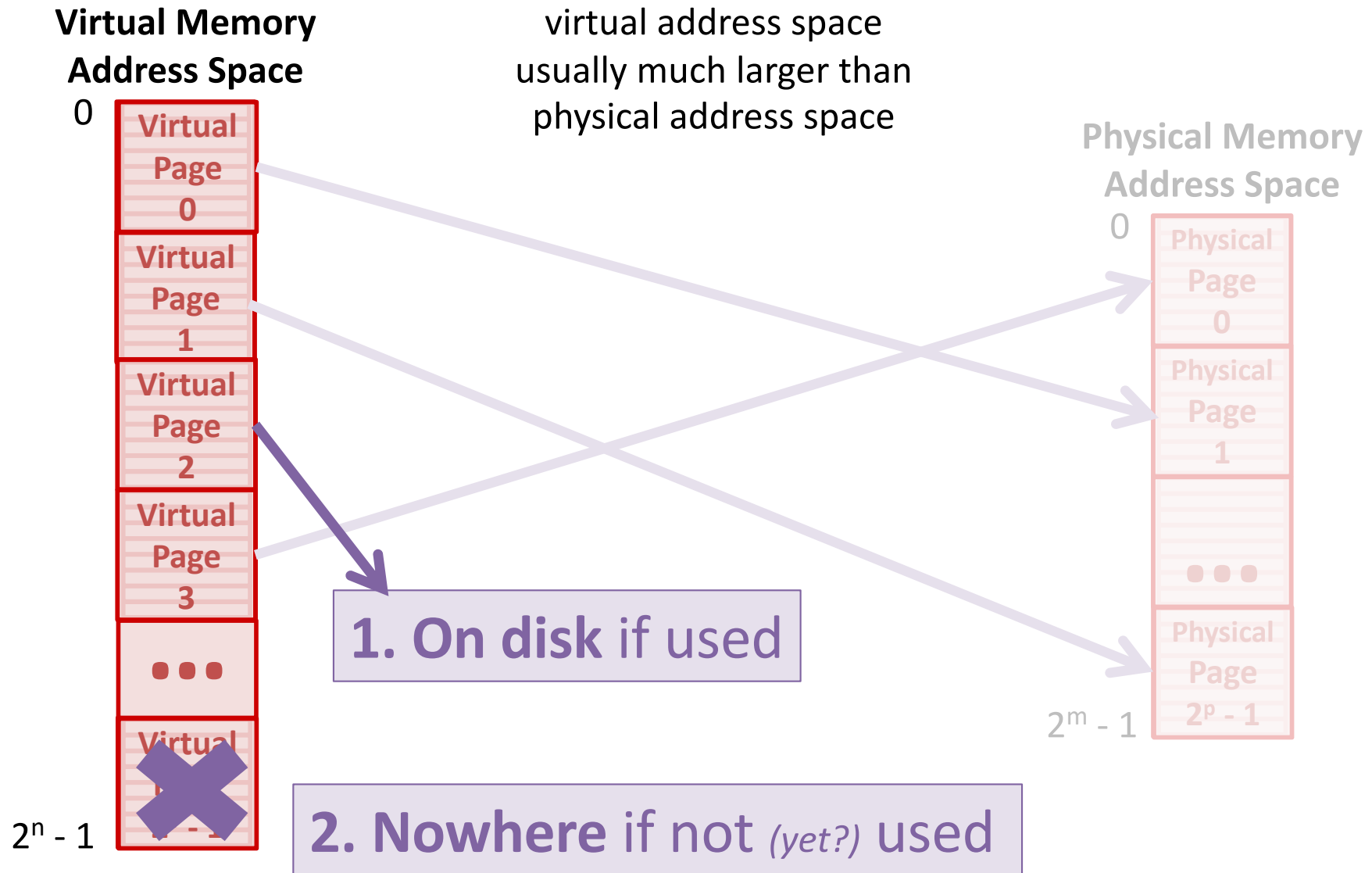
# Page-based mapping

fixed-size, aligned *pages*  
page size = power of two



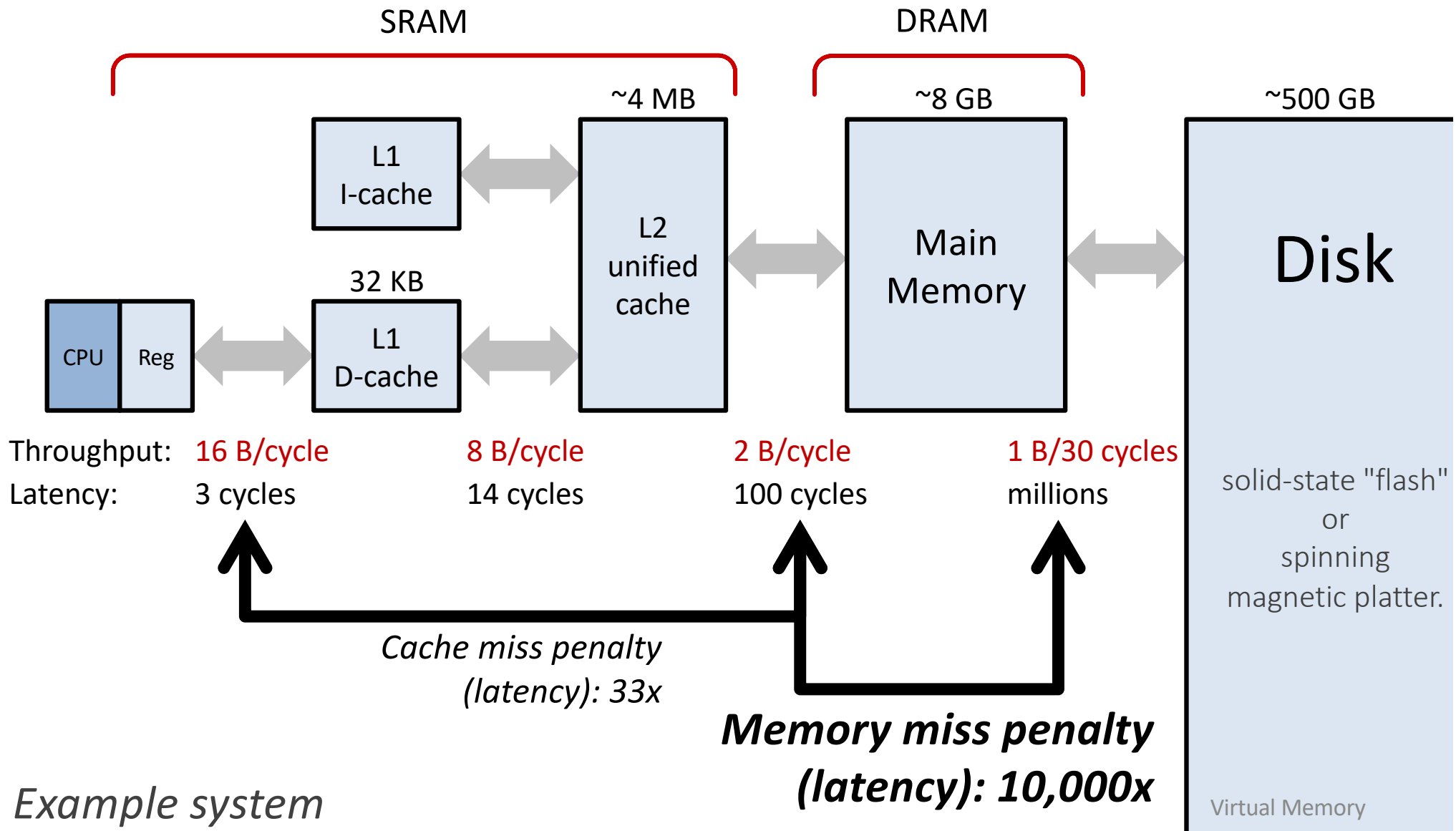
Some virtual pages do not fit!  
Where are they stored?

# Cannot fit all virtual pages! Where are the rest stored?

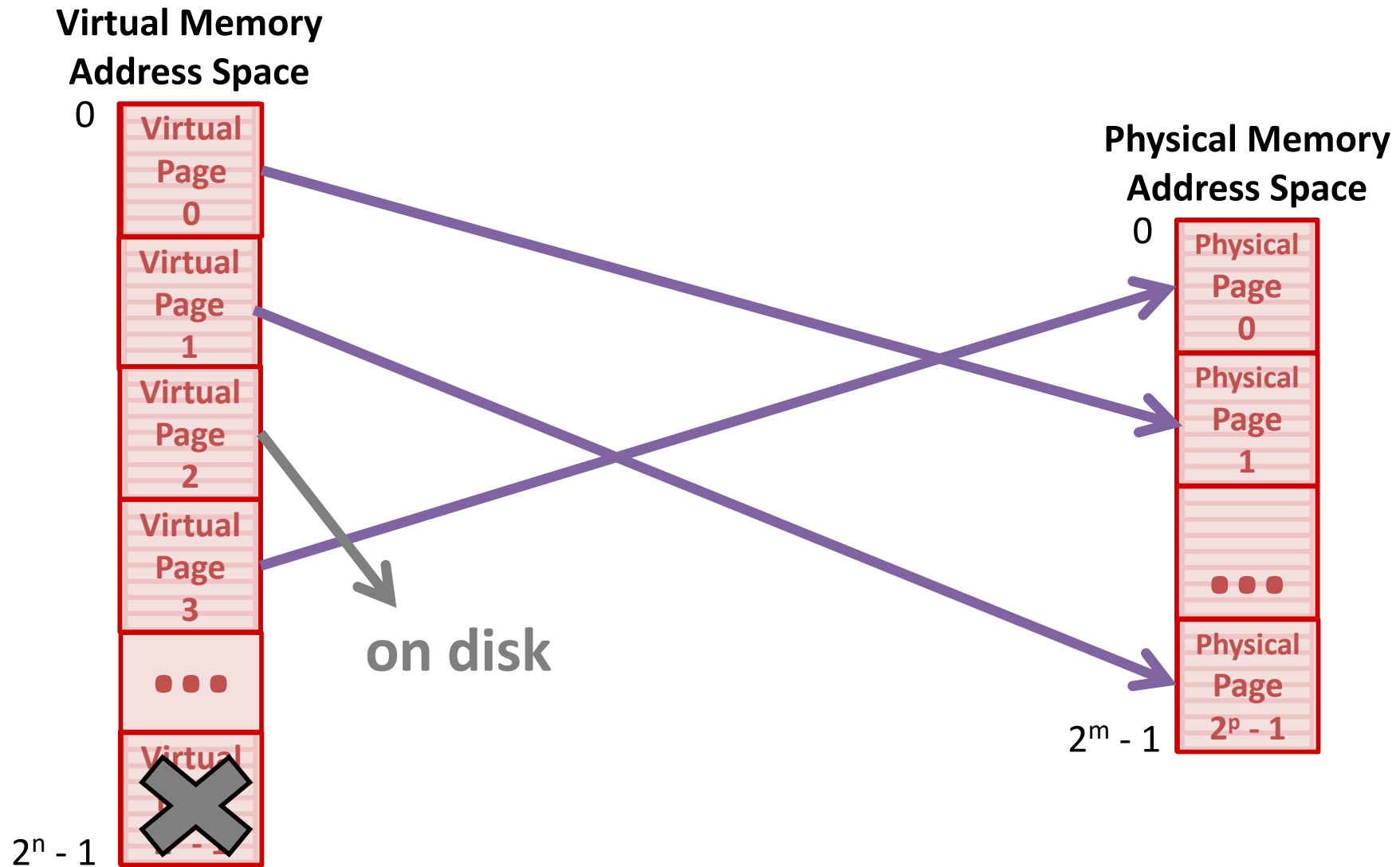


Not drawn to scale

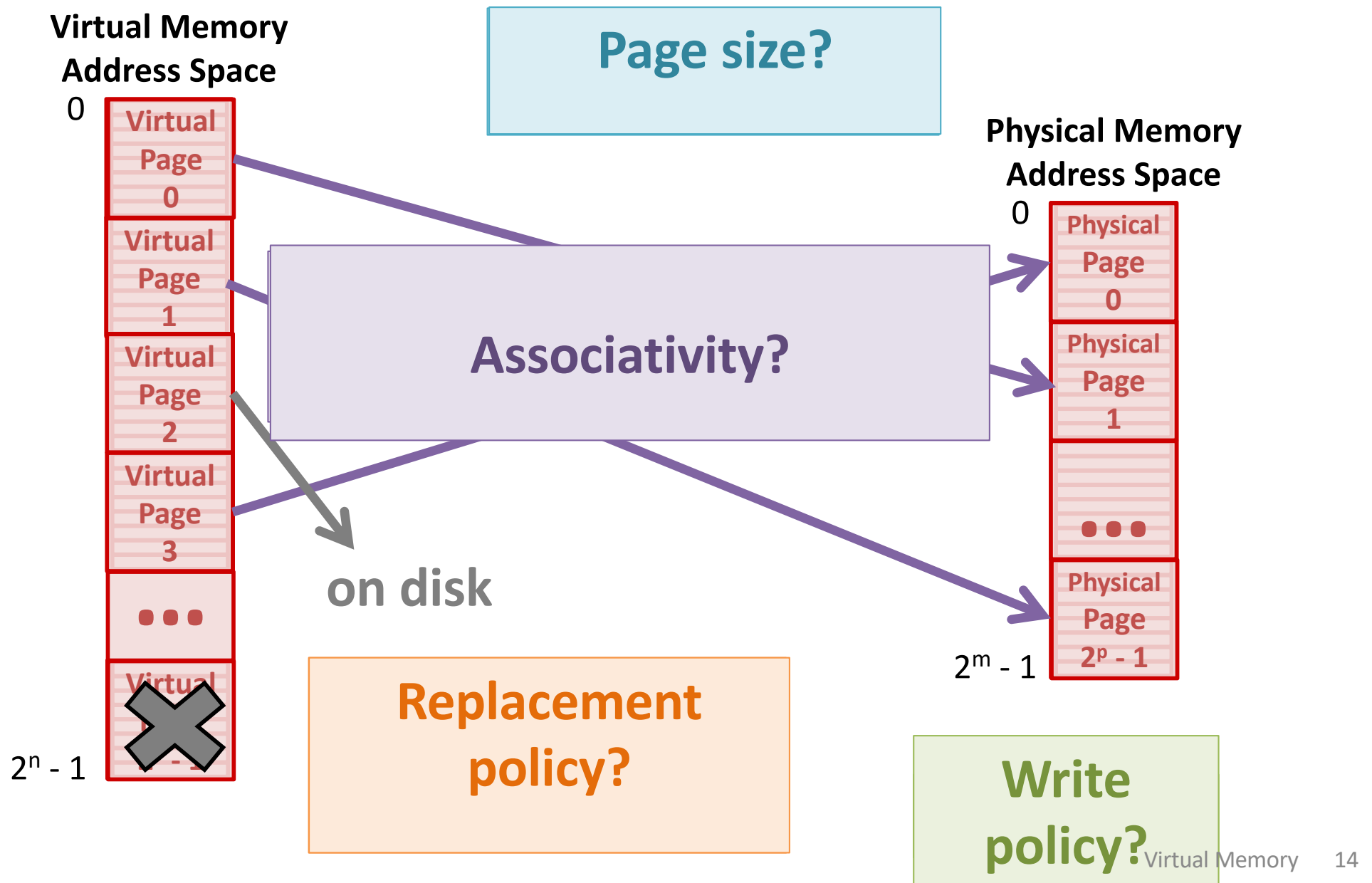
# Virtual memory: cache for disk?



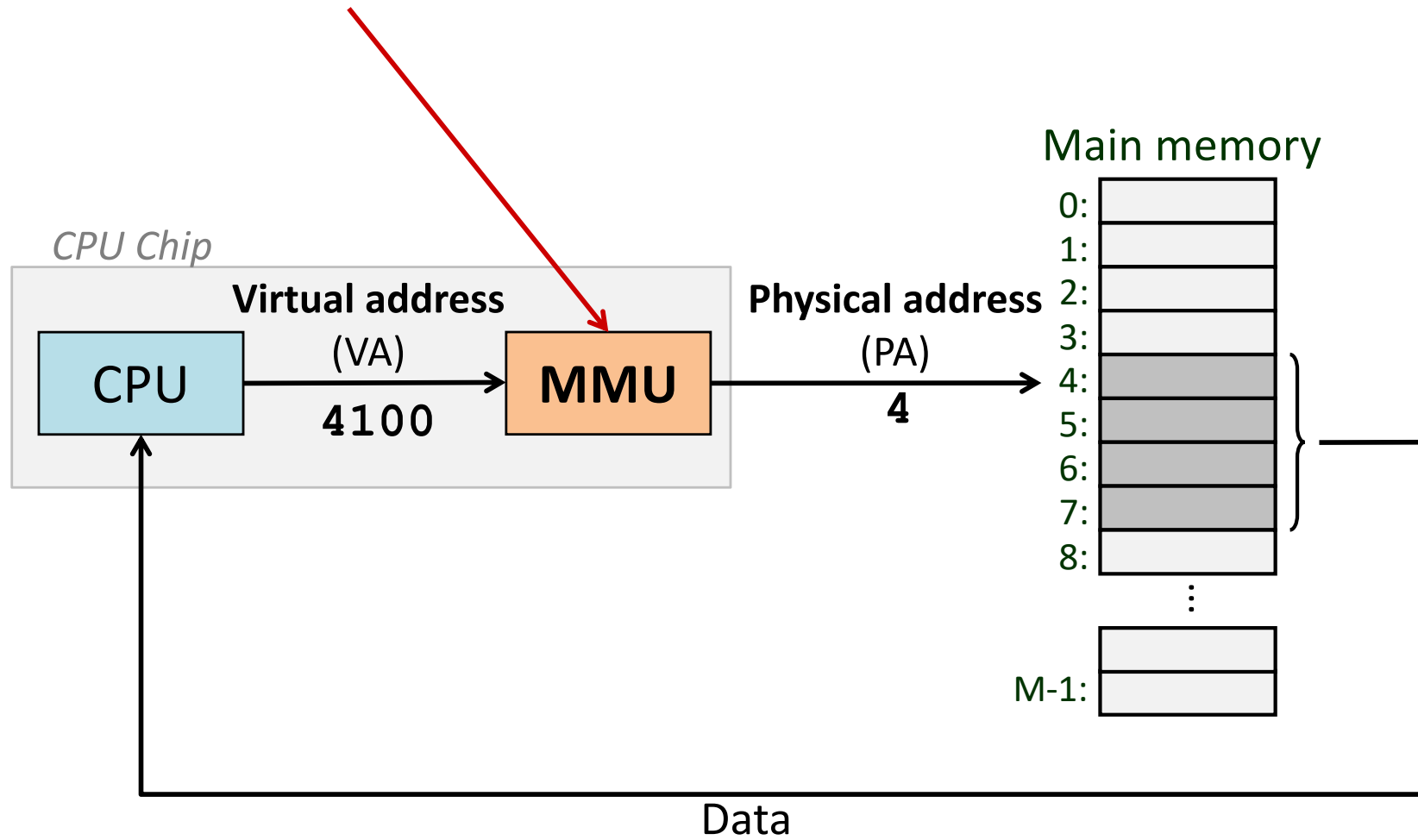
# Design for a slow disk: exploit locality



# Design for a slow disk: exploit locality

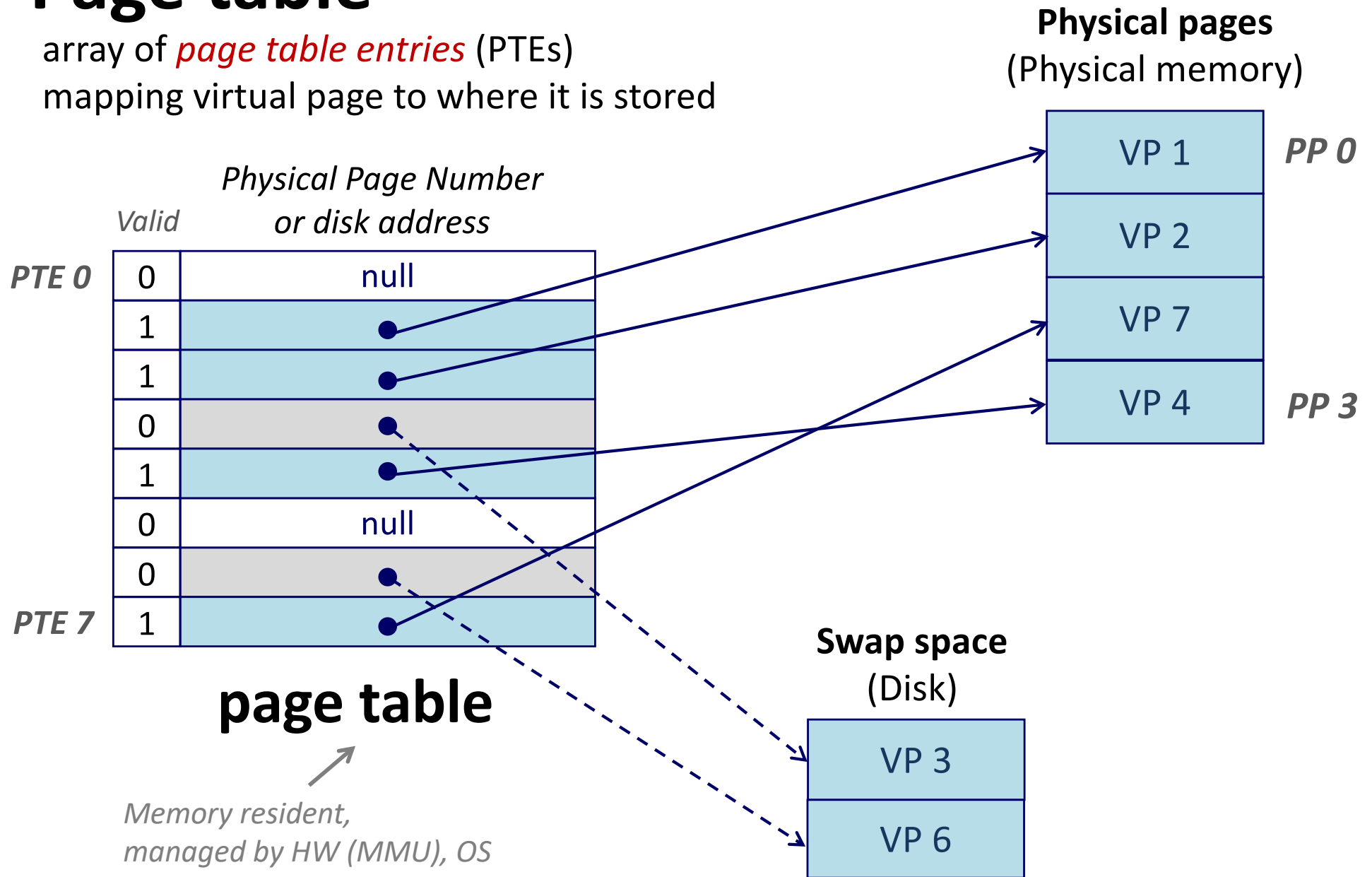


# Address translation



# Page table

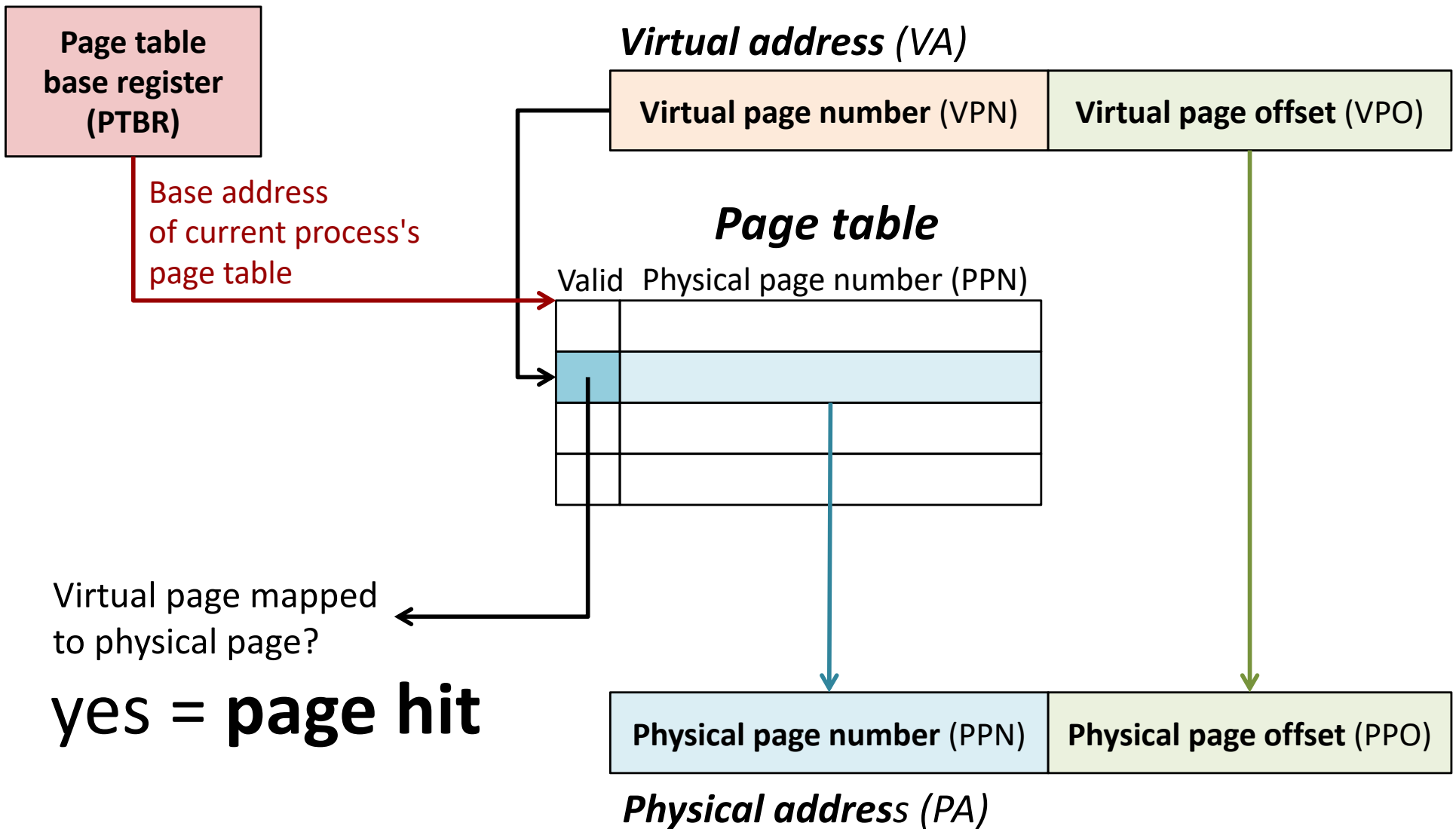
array of *page table entries* (PTEs)  
mapping virtual page to where it is stored



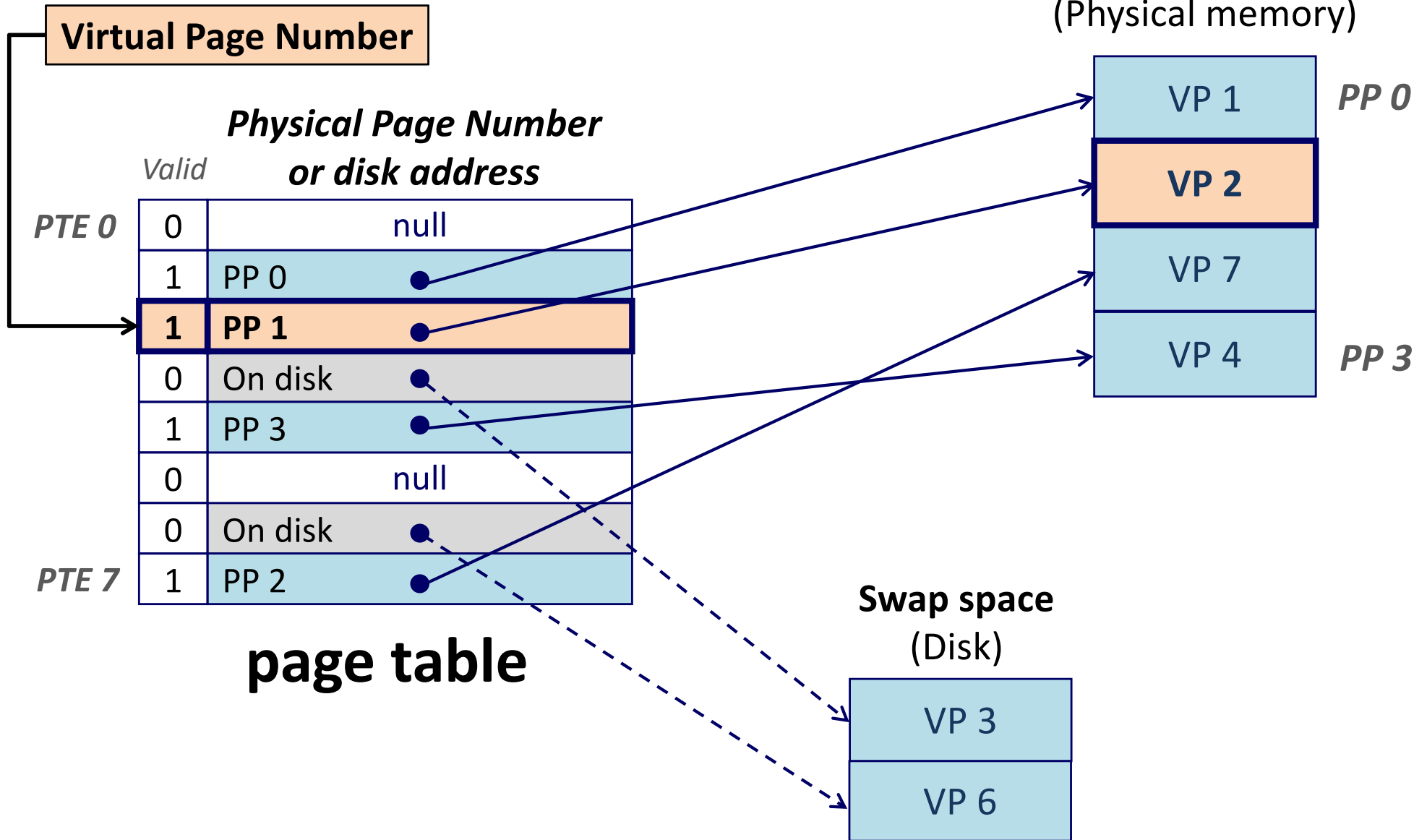
*How many page tables are in the system?*



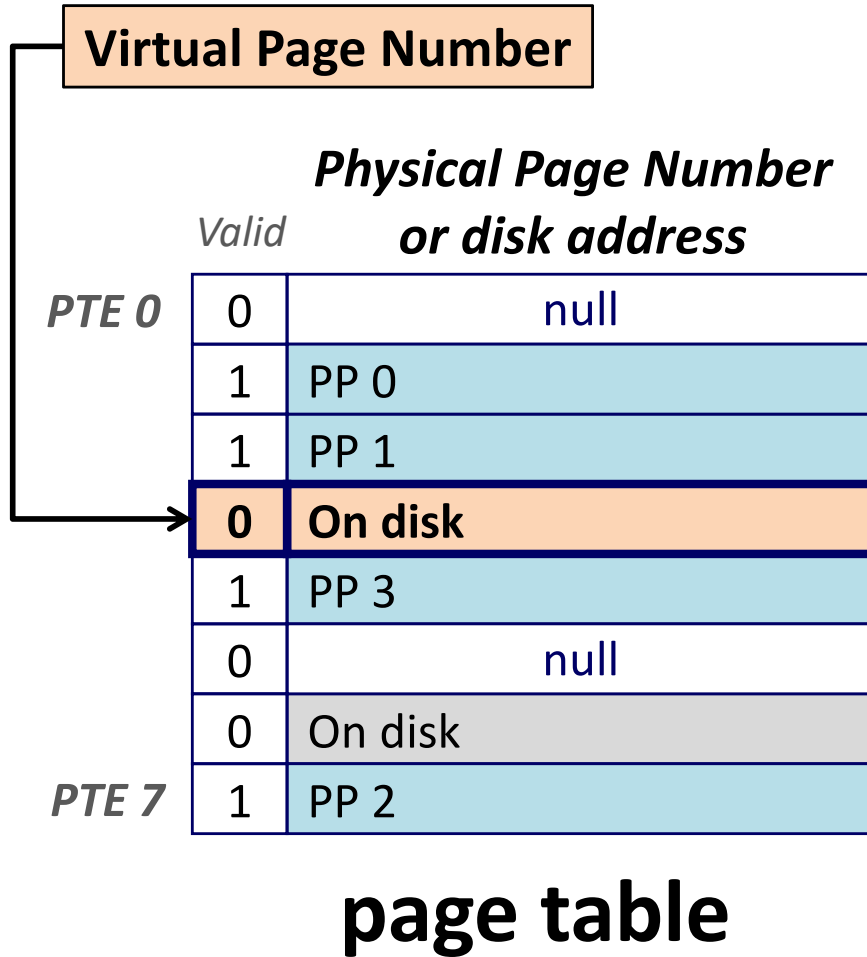
# Address translation with a page table



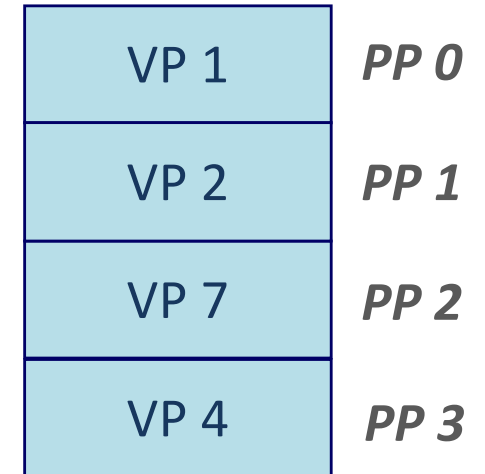
# Page *hit*: virtual page is in memory



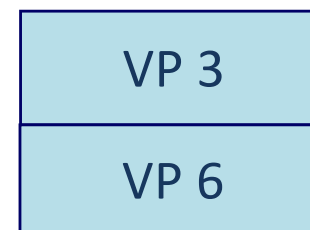
# Page fault:



Physical pages  
(Physical memory)

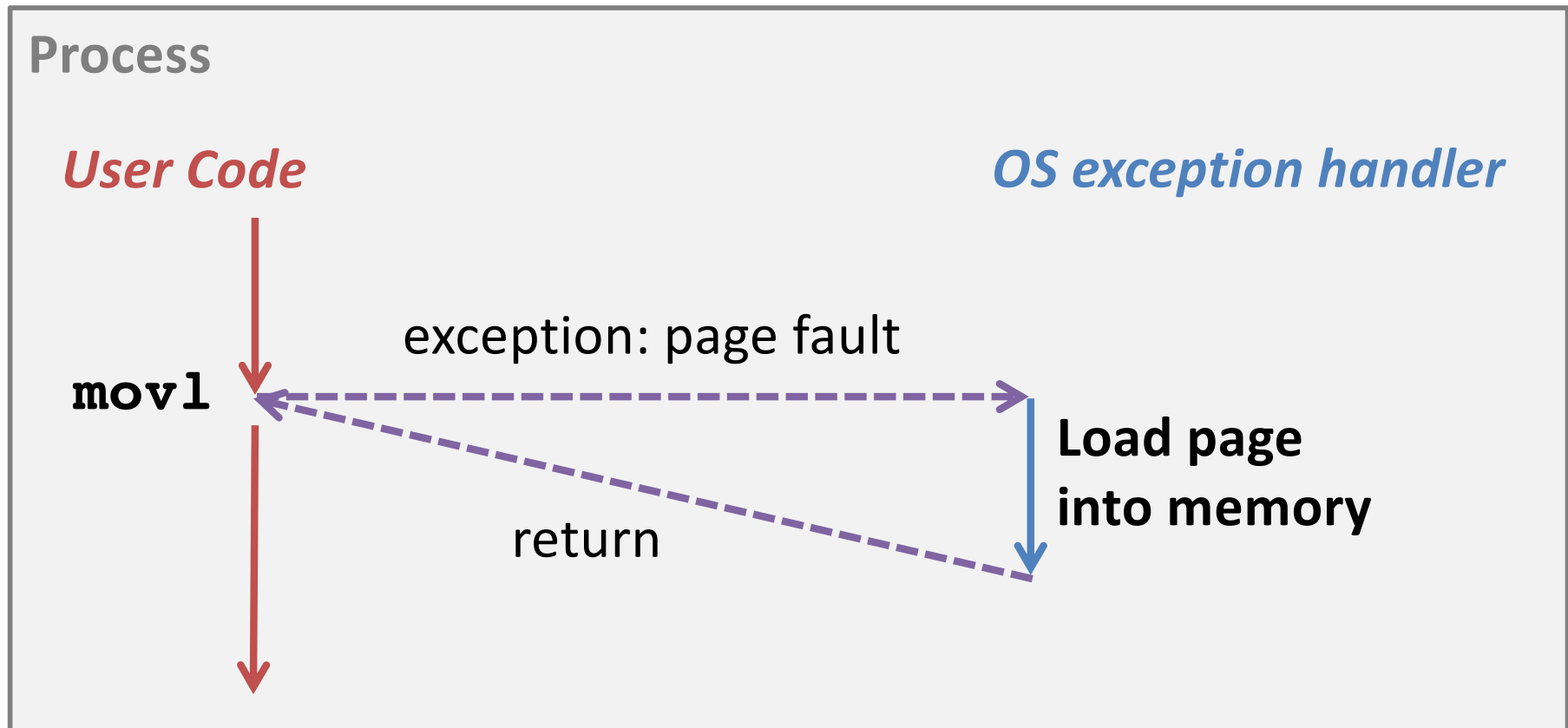


Swap space  
(Disk)



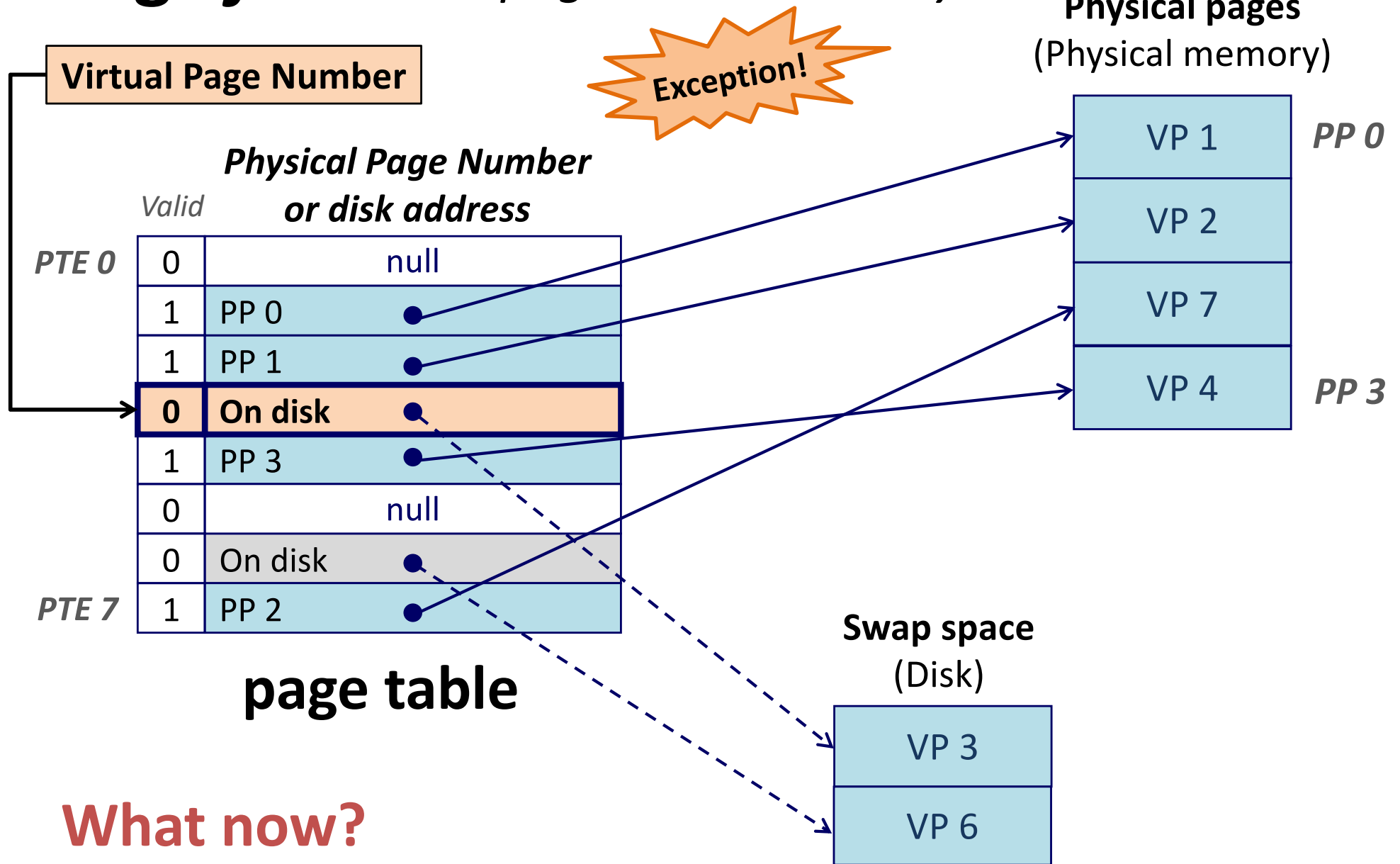
# Page *fault*: exceptional control flow

Process accessed virtual address in a page that is not in physical memory.



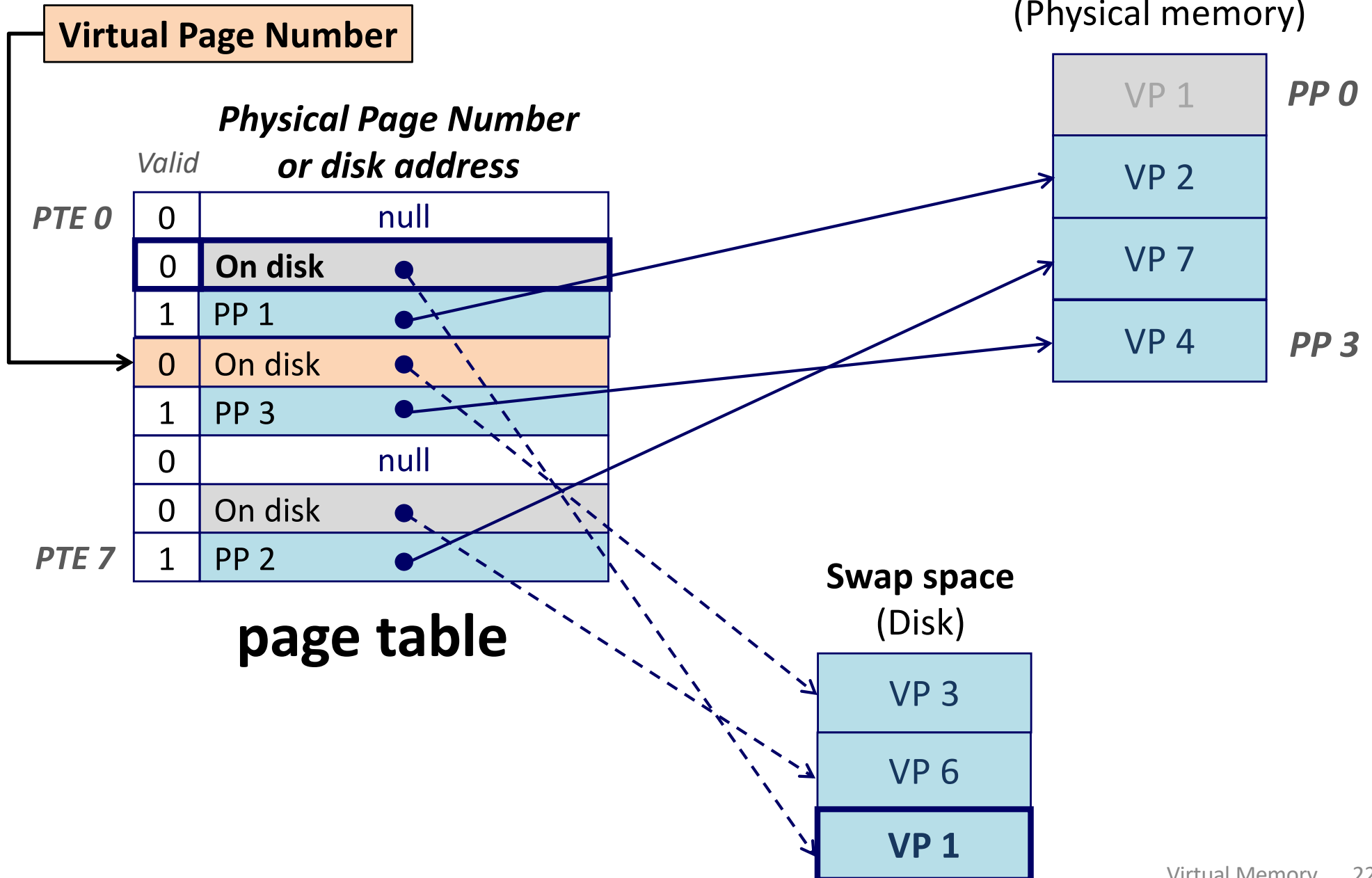
Returns to faulting instruction:  
`movl` is executed *again*!

# Page fault: 1. page not in memory

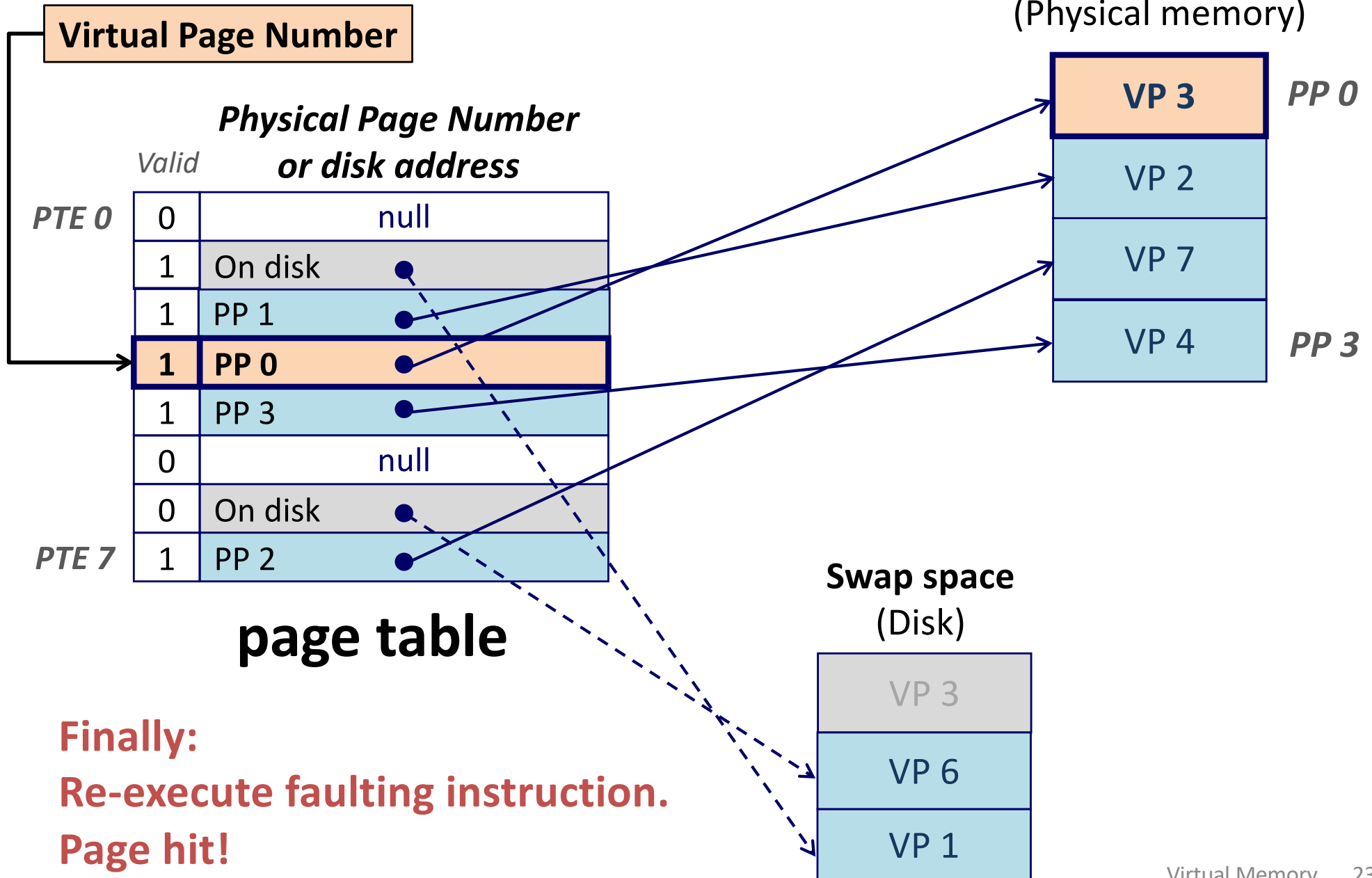


**What now?**  
**OS handles fault**

# Page fault: 2. OS evicts another page.



# Page fault: 3. OS loads needed page.



**Finally:**  
**Re-execute faulting instruction.**  
**Page hit!**

# Terminology

## context switch

Switch control between processes on the same CPU.

## page in

Move page of virtual memory from disk to physical memory.

## page out

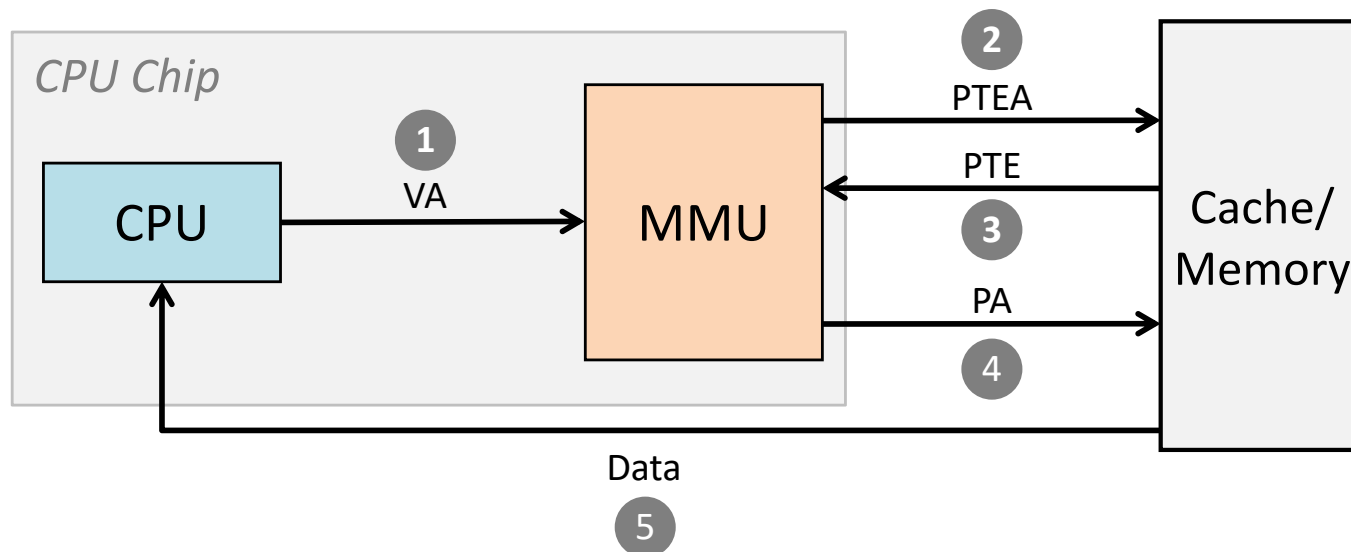
Move page of virtual memory from physical memory to disk.

## thrash

Total working set size of processes is larger than physical memory.  
Most time is spent paging in and out instead of doing useful work.

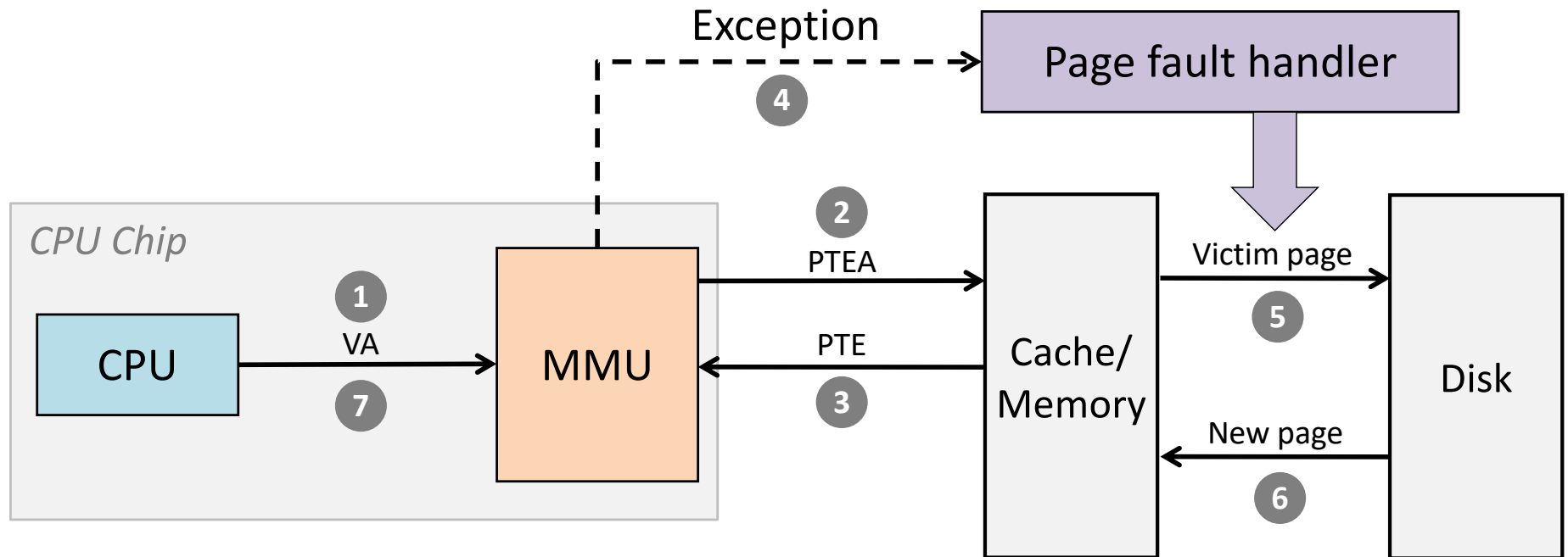


# Address translation: page *hit*



- 1) Processor sends virtual address to MMU (*memory management unit*)
- 2-3) MMU fetches PTE from page table in cache/memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

# Address Translation: Page *Fault*



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in cache/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

# How fast is translation?

How many physical memory accesses are required to complete one virtual memory access?

## Translation Lookaside Buffer (TLB)

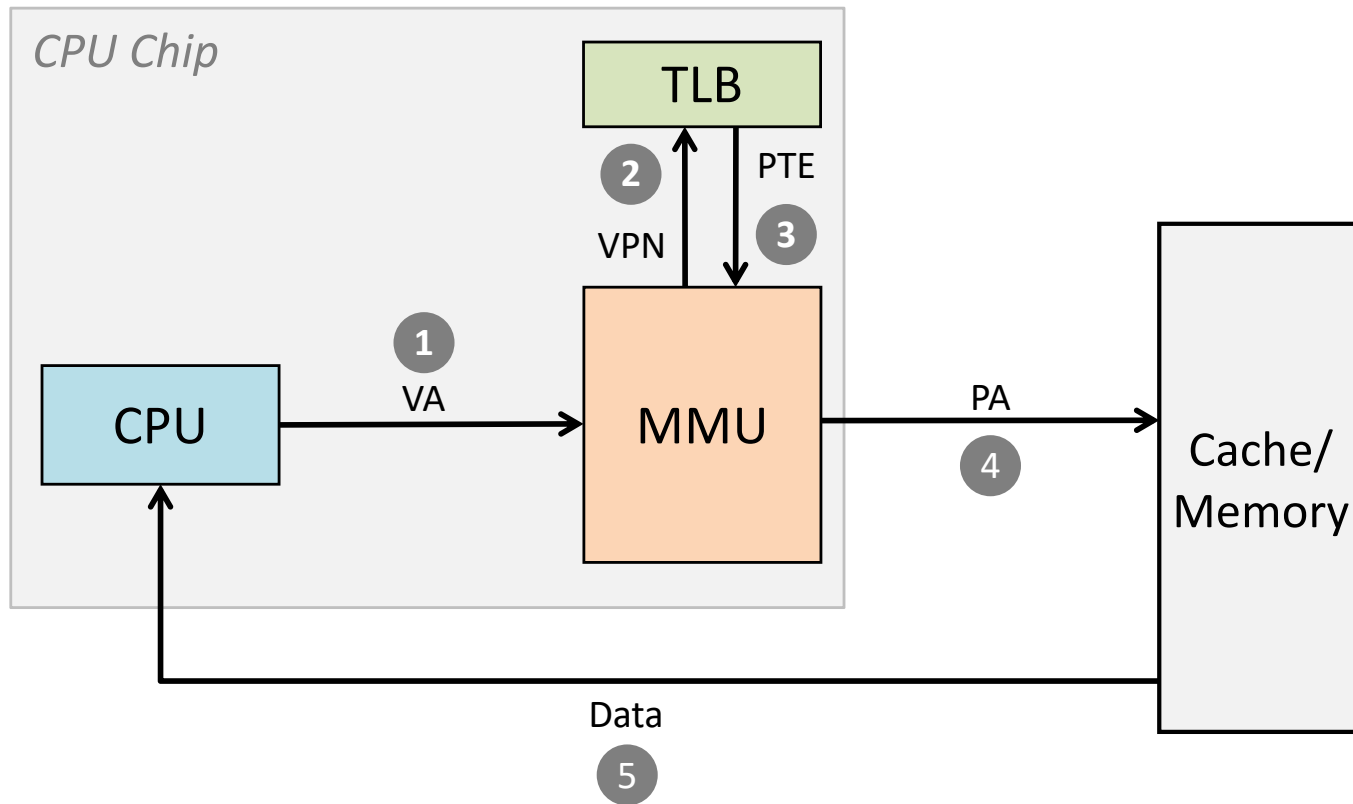
Small hardware cache in MMU just for page table entries

e.g., 128 or 256 entries

Much faster than a page table lookup in memory.

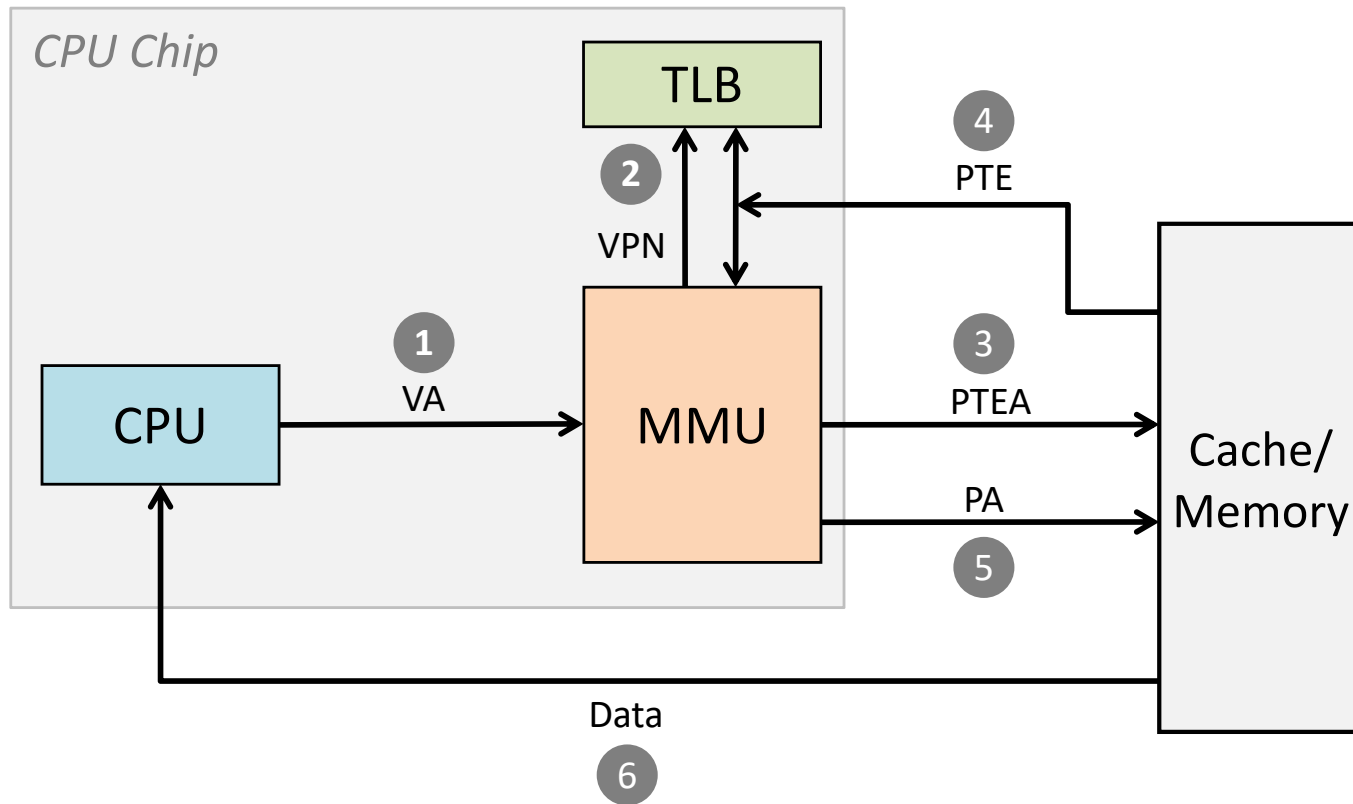
In the running for *"un/classiest name of a thing in CS"*

# TLB hit



**A TLB hit eliminates a memory access**

# TLB miss



**A TLB miss incurs an additional memory access (the PTE)**

Fortunately, TLB misses are rare. **Does a TLB miss require disk access?**

# Memory system example (small)

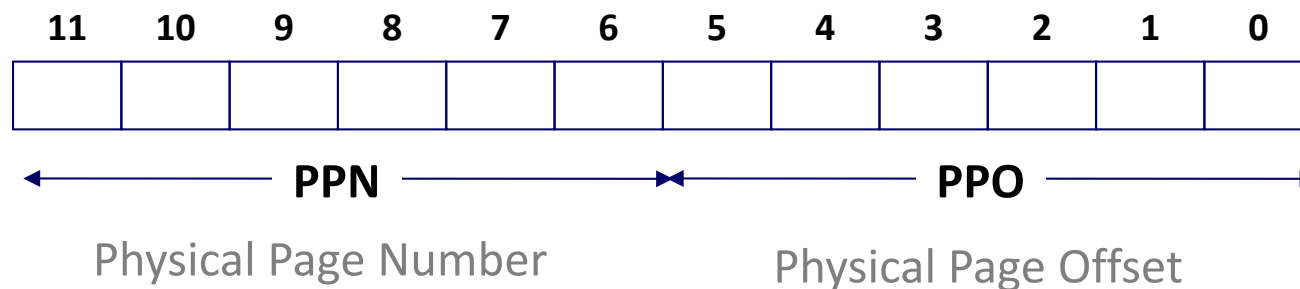
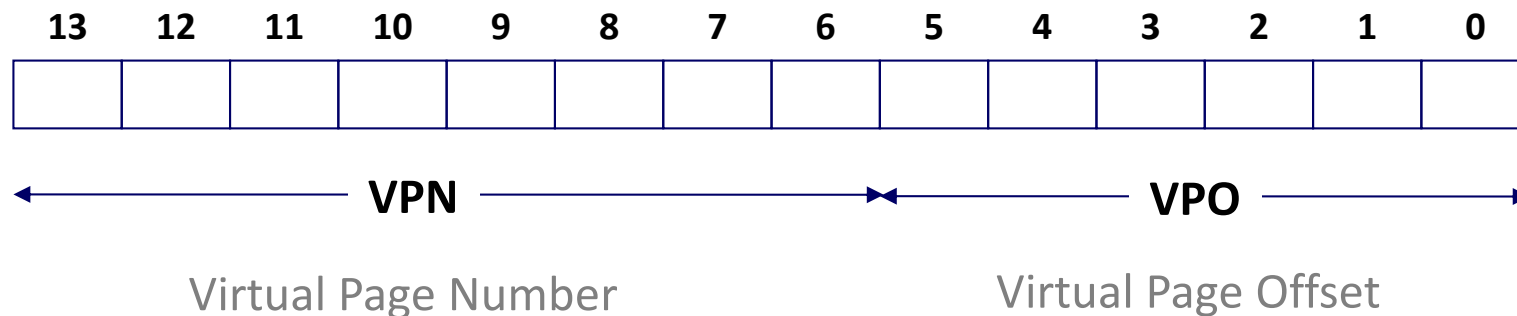
## Addressing

14-bit virtual addresses

12-bit physical address

Page size = 64 bytes

Simulate accessing these virtual addresses on the system: **0x03D4**, **0x0B8F**, **0x0020**



# Memory system example: page table

Only showing first 16 entries (out of 256 = 2<sup>8</sup>)

virtual page # \_\_\_ TLB index \_\_\_ TLB tag \_\_\_ TLB Hit? \_\_\_ Page Fault? \_\_\_ physical page #: \_\_\_

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

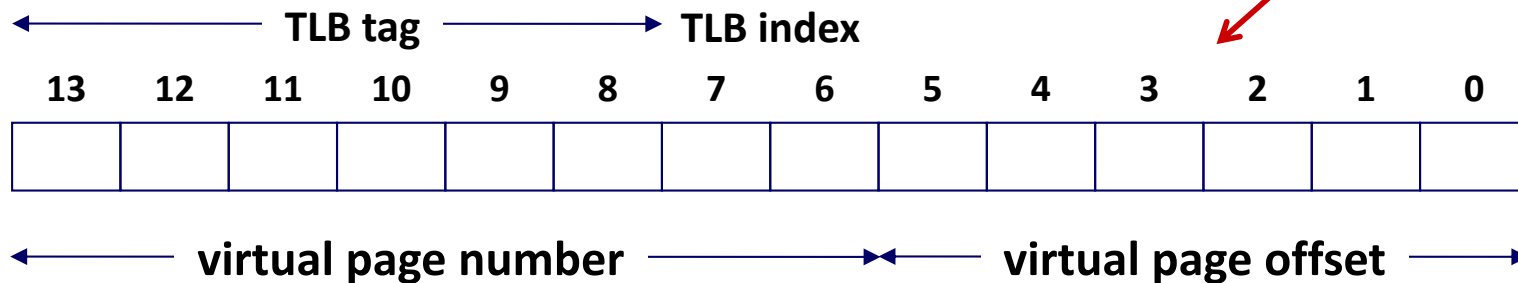
What about a real address space? Read more in the book...

# Memory system example: TLB

16 entries

4-way associative

TLB ignores page offset. Why?



virtual page # \_\_\_ TLB index \_\_\_ TLB tag \_\_\_ TLB Hit? \_\_ Page Fault? \_\_ physical page #: \_\_\_

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	09	0D	1	00	-	0	07	02	1
1	03	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	03	0D	1	0A	34	1	02	-	0



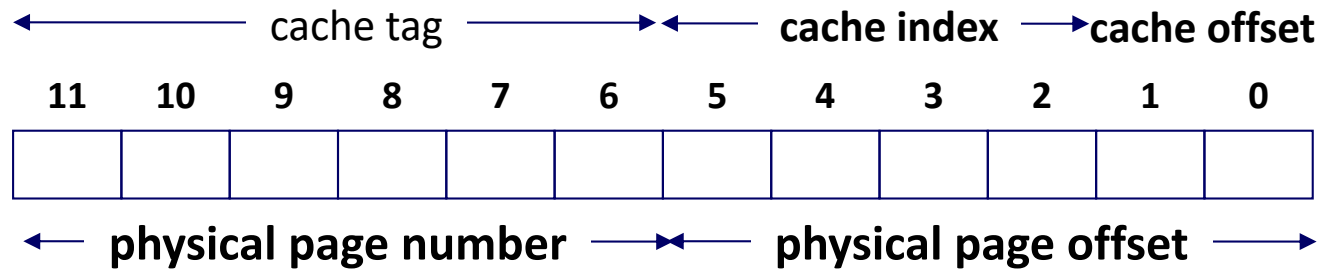
# Memory system example: cache

16 lines

4-byte block size

Physically addressed

Direct mapped



cache offset\_\_\_ cache index\_\_\_ cache tag\_\_\_ Hit? \_\_\_ Byte: \_\_\_\_\_

<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

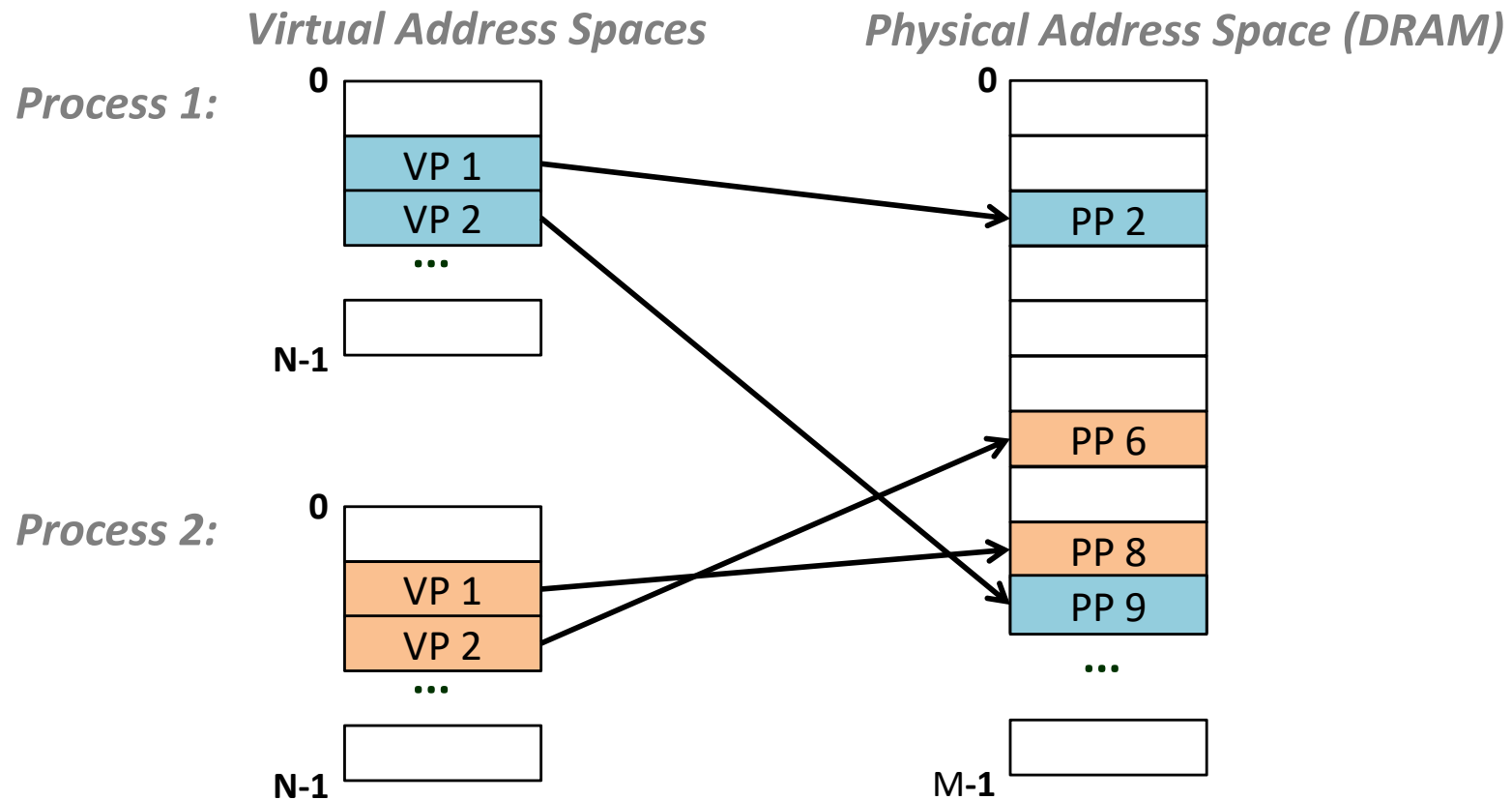
<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

# Virtual memory benefits:

## Simple address space allocation

Process needs private *contiguous* address space.


Storage of virtual pages in physical pages is **fully associative**.




# Virtual memory benefits:

## Simple cached access to storage > memory

Good locality, or least "small" working set = mostly page hits



All necessary  
page table entries  
fit in TLB



Working set pages  
fit in physical memory

If combined working set > physical memory:

**Thrashing:** Performance meltdown. CPU always waiting or paging.

Full indirection quote:

“Every problem in computer science can be solved by adding another level of indirection, ***but that usually will create another problem.***”

# Virtual memory benefits:

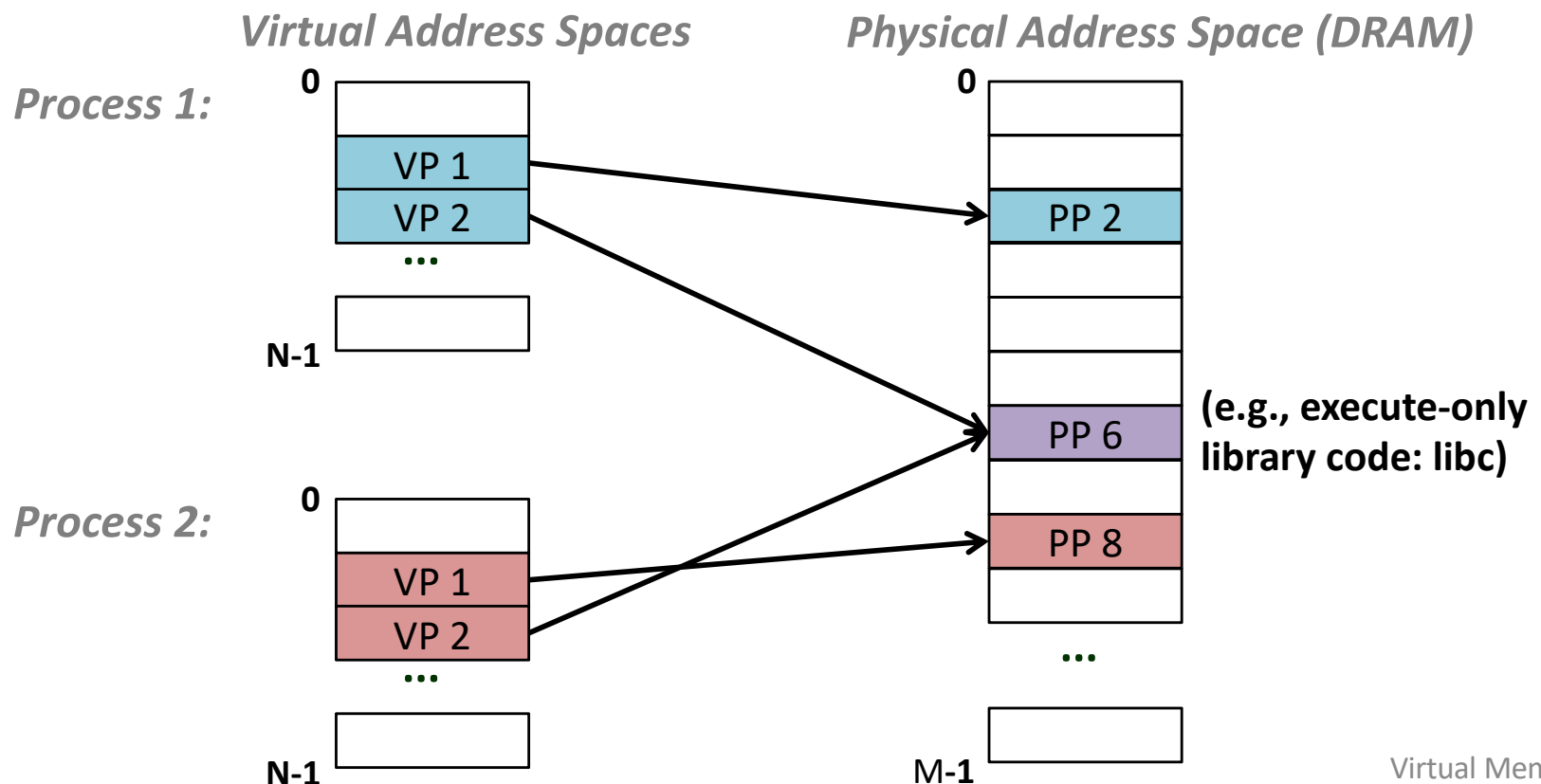
## Protection:

All accesses go through translation.

Impossible to access physical memory not mapped in virtual address space.

## Sharing:

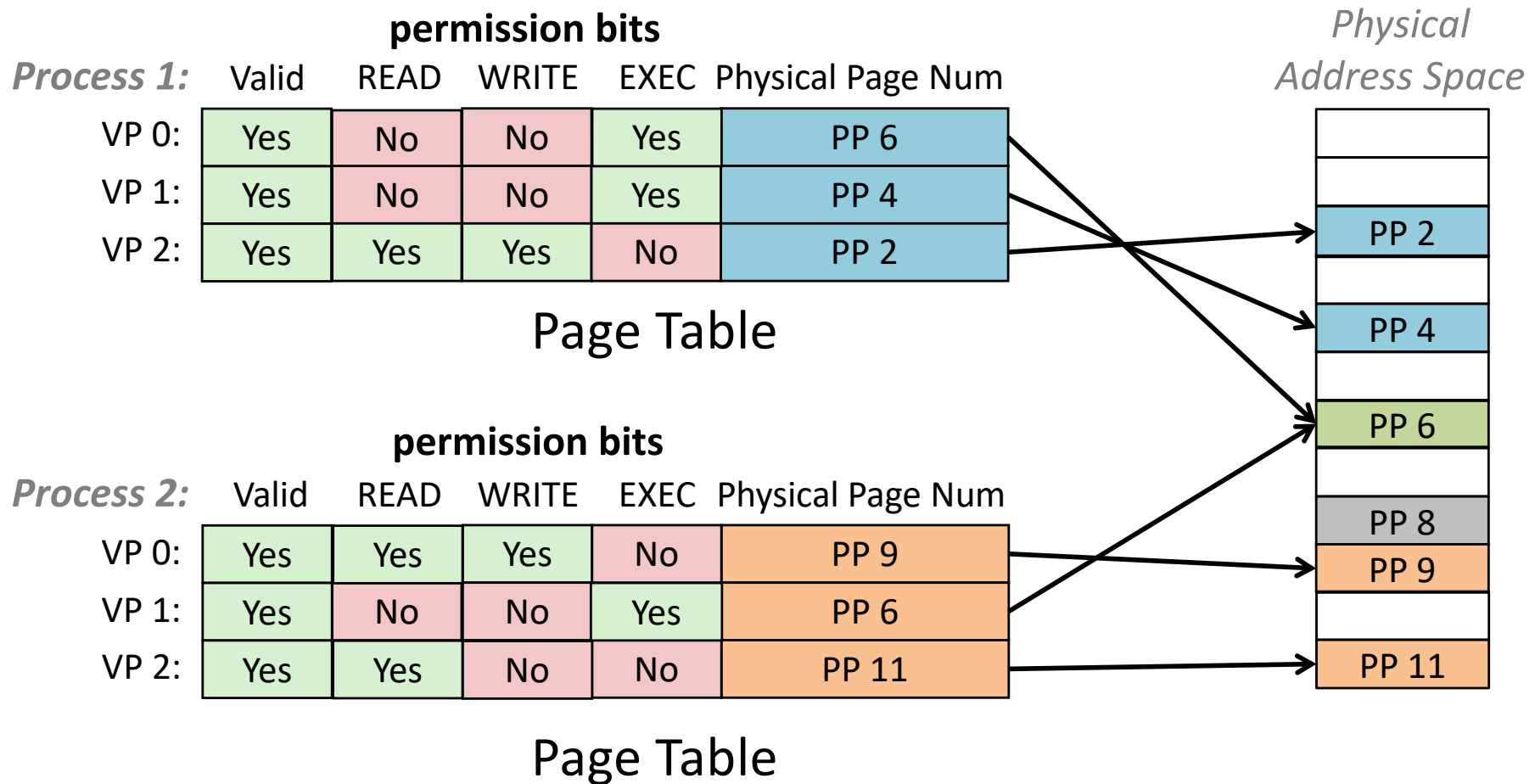
Map virtual pages in separate address spaces to same physical page (*PP 6*).



# Virtual memory benefits:

## Memory permissions

MMU checks on every access.  
**Exception if not allowed.**



How would you set permissions for the stack, heap, global variables, literals, code?

# Summary: **virtual memory**

## Programmer's view of virtual memory

Each process has its own private linear address space  
Cannot be corrupted by other processes

## System view of virtual memory

Uses memory efficiently (due to locality) by caching virtual memory pages

Simplifies memory management and sharing

Simplifies protection -- easy to interpose and check permissions

More goodies:

- Memory-mapped files
- Cheap fork() with copy-on-write pages (COW)

# Summary: memory hierarchy

## L1/L2/L3 Cache: Pure Hardware

Purely an optimization

"Invisible" to program and OS, no direct control

Programmer cannot control caching, can write code that fits well

## Virtual Memory: Software-Hardware Co-design

Supports processes, memory management

Operating System (**software**) manages the mapping

- Allocates physical memory

- Maintains page tables, permissions, metadata

- Handles exceptions

Memory Management Unit (**hardware**) does translation and checks

- Translates virtual addresses via page tables, enforces permissions

- TLB caches the mapping

Programmer cannot control mapping, can control sharing/protection via OS