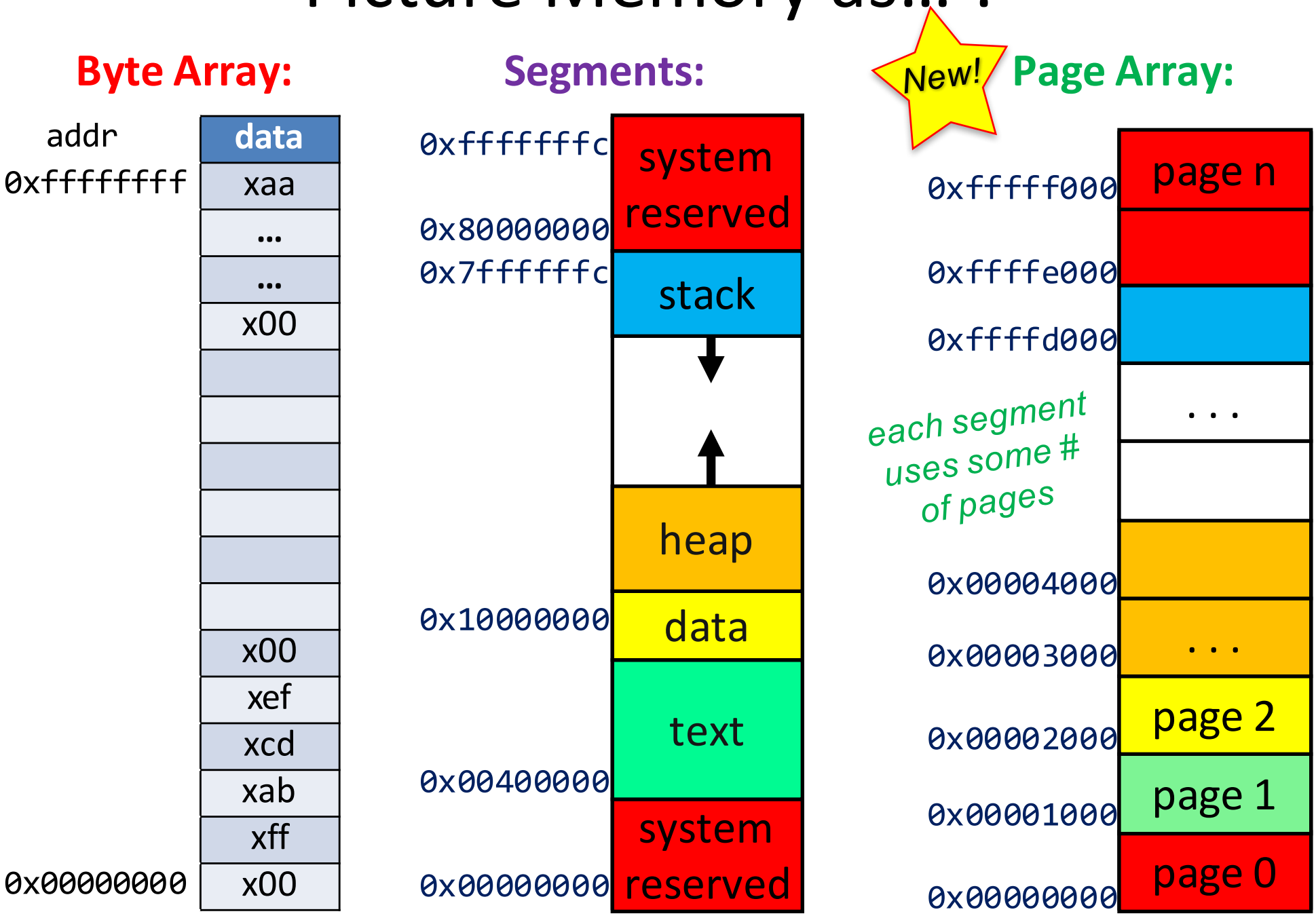
Virtual Memory

Anne Bracy
CS 3410
Computer Science
Cornell University

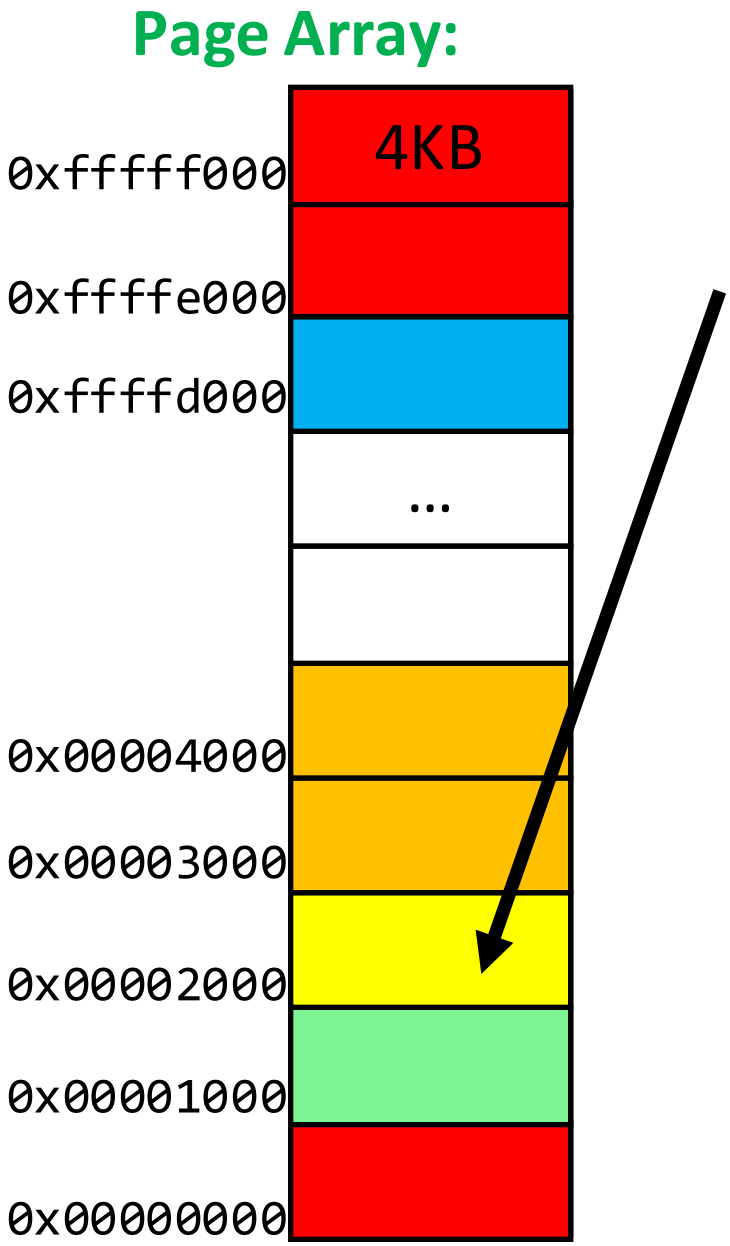
The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.

P & H Chapter 5.7

Picture Memory as… ?



A Little More About Pages



Suppose each page = 4KB

Anything in page 2 has address:
0x00002xxx

Lower 12 bits specify which byte you are in the page:

0x00002200 = 0010 0000 0000
= byte 512

upper bits = page number
lower bits = page offset

Sound familiar?

Data Granularity

ISA: instruction specific: LB, LH, LW (MIPS)

Registers: 32 bits (MIPS)

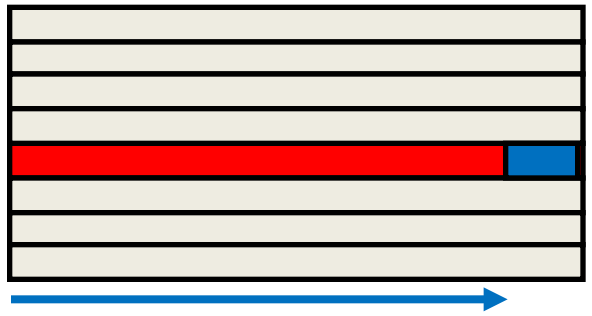
Caches: cache line/block

Address bits divided into:

index: which entry in the cache

tag: sanity check for address match

offset: which byte in the line

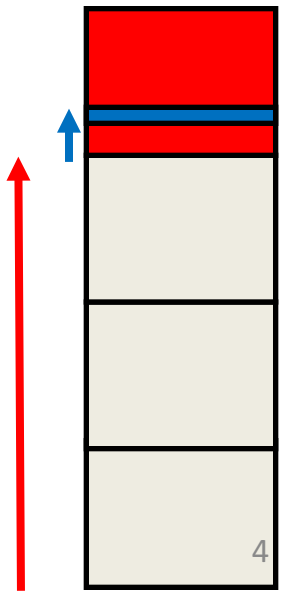


Memory: page

Address bits divided into:

page number: which page in memory

index: which byte in the page



Program's View of Memory

32-bit machine:

0x00000000 – 0xffffffff to play with

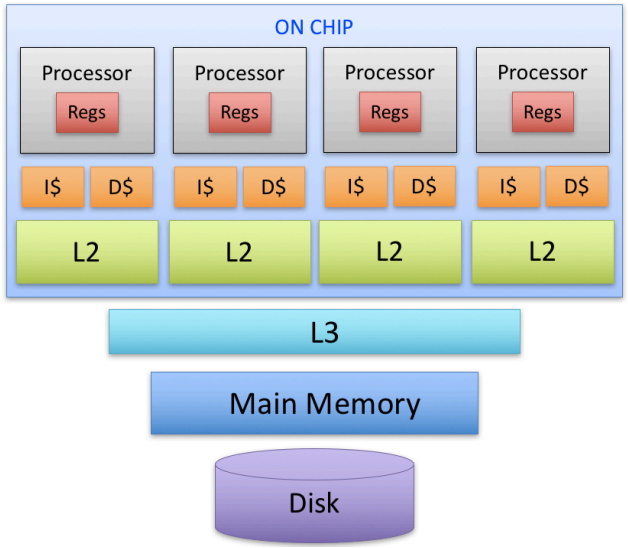
(modulo system reserved)

64-bits: 16 EB ???

2 Interesting/Dubious Assumptions:

The machine I'm running on has 4GB of DRAM.

I am the only one using this DRAM.



These assumptions are embedded in the executable!

If they are wrong, things will break!

~~Recompile? Relink?~~

Indirection* to the Rescue!

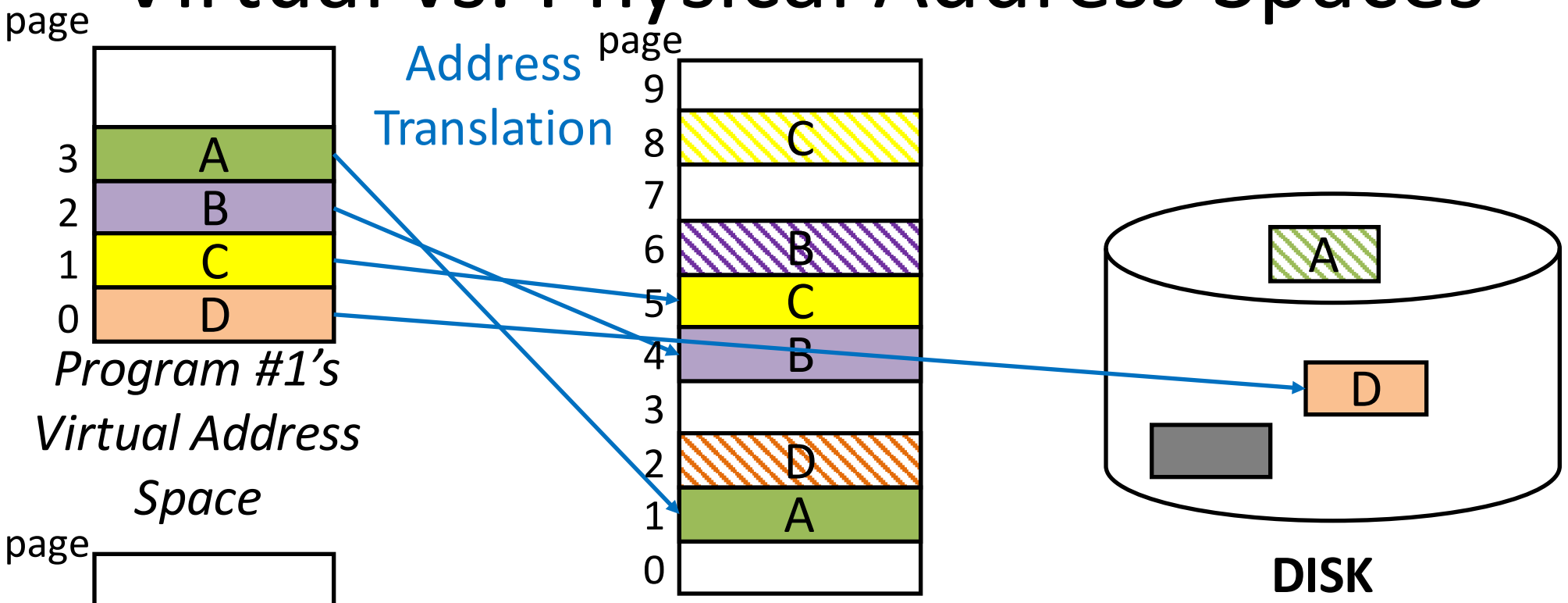
Virtual Memory: a Solution for All Problems

- Each process has its own virtual address space
  - Program/CPU can access any address from 0…2^N
  - A process is a program being executed
  - Programmer can code as if they own all of memory

- On-the-fly at runtime, for each memory access
  - all accesses are *indirect* through a virtual address
  - translate fake virtual address to a real physical address
  - redirect load/store to the physical address

*google David Wheeler, Butler Lampson, Leslie Lamport, and Steve Bellovin

Virtual vs. Physical Address Spaces



- Not contiguous
- Page vs. Address?

Advantages of Virtual Memory

Easy relocation

- Loader puts code anywhere in physical memory
- Virtual mappings to give illusion of correct layout

Higher memory utilization

- Provide illusion of contiguous memory
- Use all physical memory, even physical address 0x0

Easy sharing

- Different mappings for different programs / cores

And more to come…

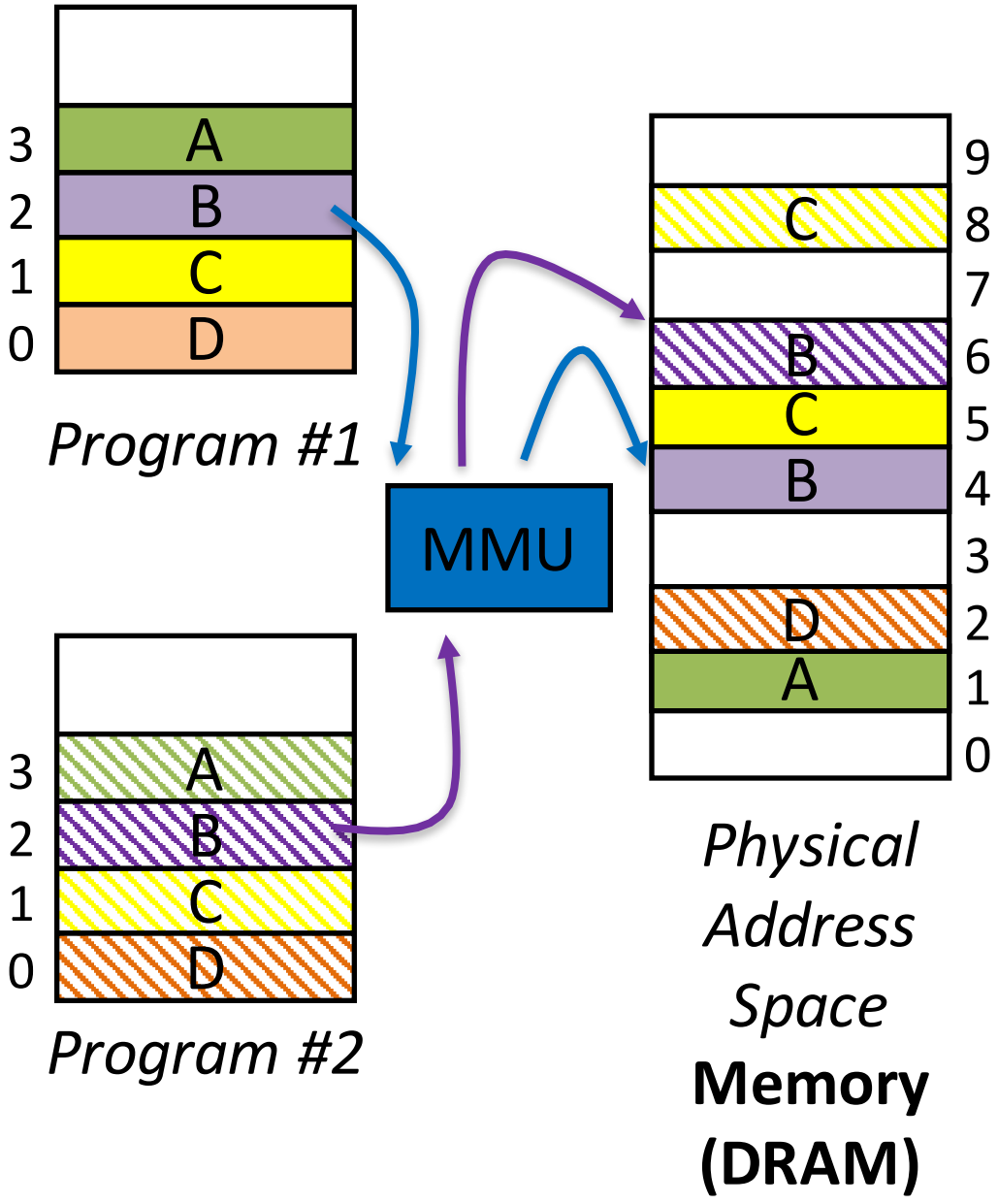
Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

- Address Translation
- Overhead
- Paging
- Performance
- Virtual Memory & Caches

Address Translator: MMU



- Programs use virtual addresses
- Actual memory uses physical addresses

Memory Management Unit (MMU)

- HW structure
- Translates virtual → physical address on the fly

Address Translation: in Page Table

OS-Managed Mapping of Virtual → Physical Pages

```
int page_table[2^20] = {0, 5, 4, 1, … };
. . .
ppn = page_table[vpn];
```

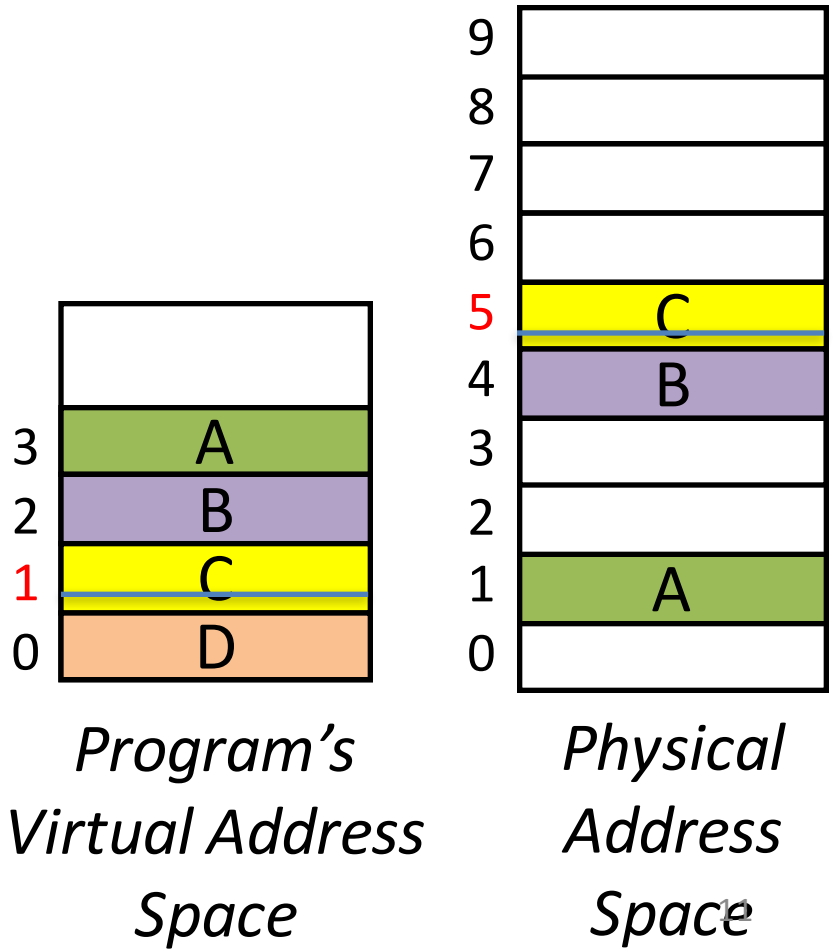
Remember:

any address 0x00001234

is x234 bytes into Page C

both virtual & physical

VP 1 → PP 5



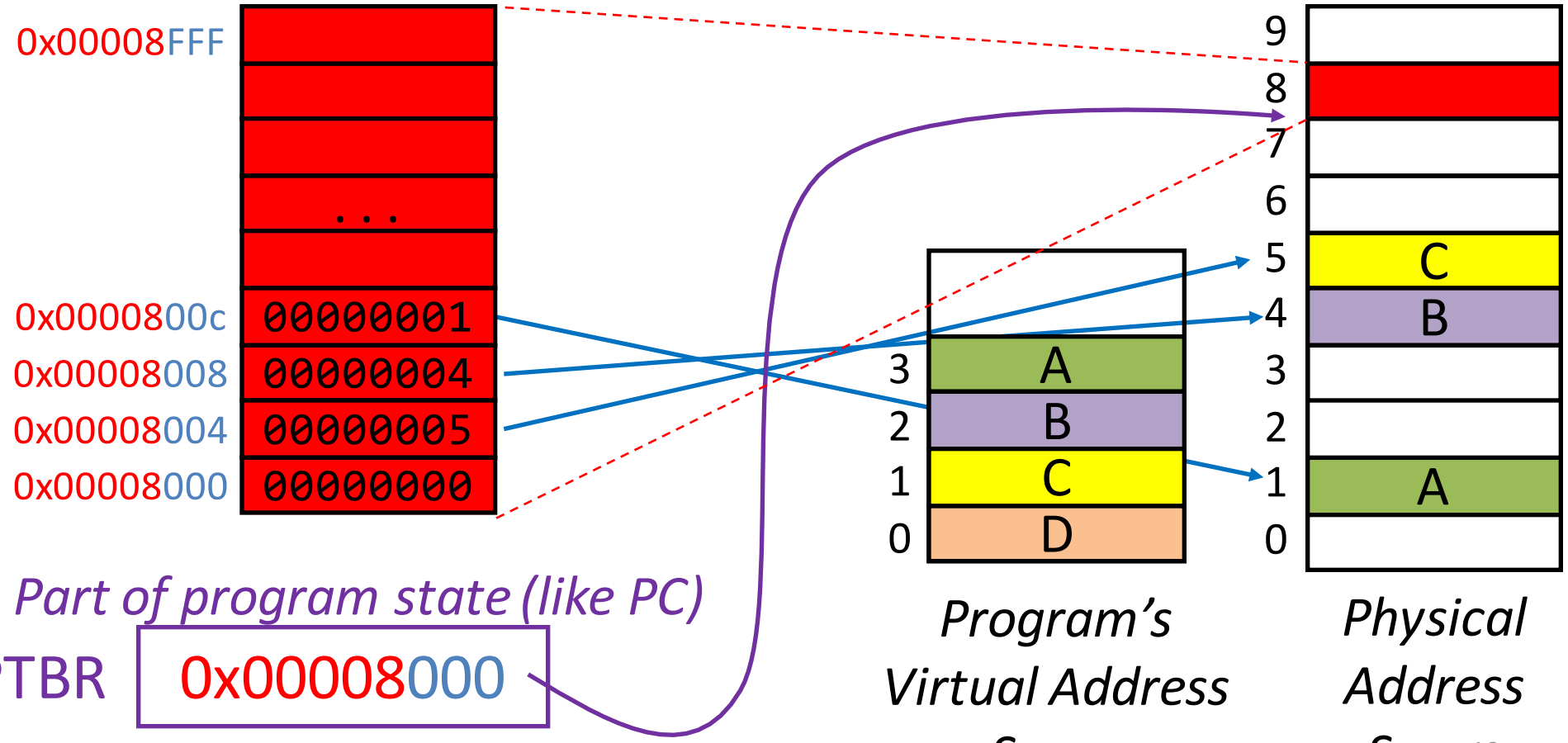
Assuming each page = 4KB

Page Table Basics

1 Page Table *per process*

Lives in Memory, *i.e. in a page (or more…)*

Location stored in **Page Table Base Register**

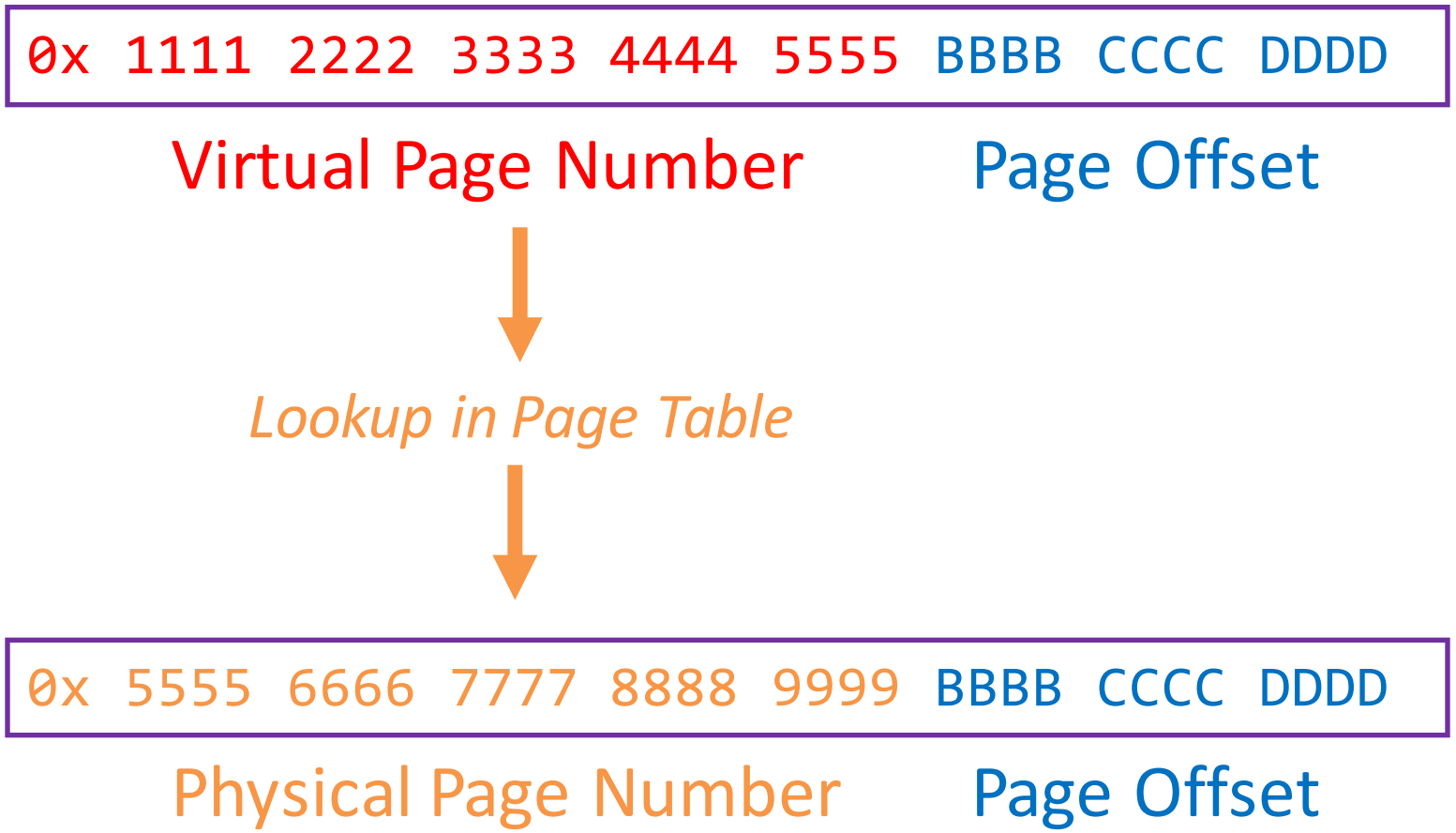


Part of program state (like PC)

PTBR 0x00008000

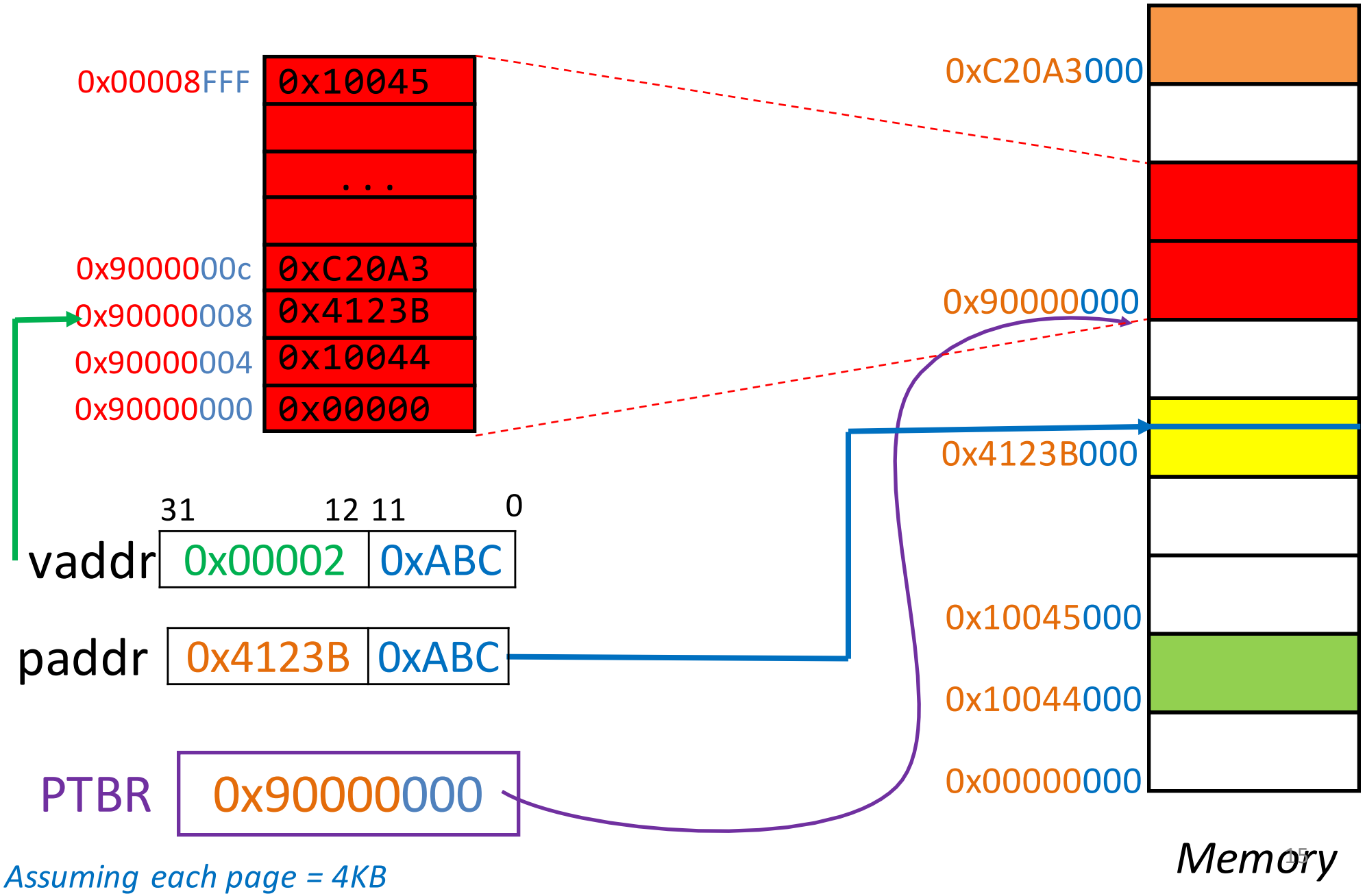
Assuming each page = 4KB

Simple Address Translation



Assuming each page = 4KB

Simple Page Table Translation



General Address Translation

- What if the page size is not 4KB?
  → Page offset is no longer 12 bits

Clicker Question:
Page size is 16KB → how many bits is page offset?
(a) 12 (b) 13 (c) 14 (d) 15 (e) 16

- What if Main Memory is not 4GB?
  → Physical page number is no longer 20 bits

Clicker Question:
Page size 4KB, Main Memory 512 MB
→ how many bits is PPN?
(a) 15 (b) 16 (c) 17 (d) 18 (e) 19

Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

- Address Translation
- Overhead
- Paging
- Performance
- Virtual Memory & Caches

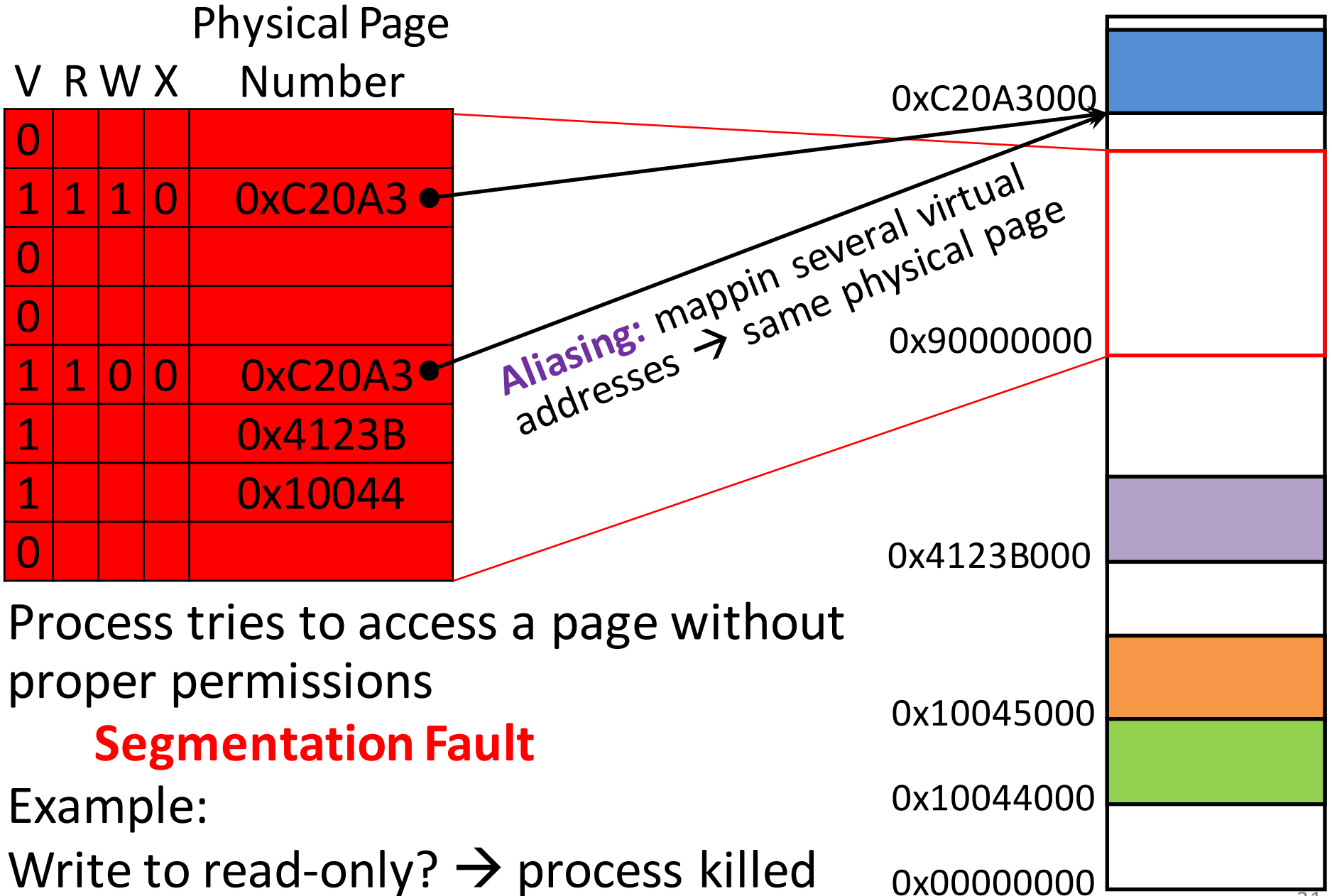
Page Table Overhead

- How large is PageTable?
- Virtual address space (for each process):
  - Given: total virtual memory: 2^{32} bytes = 4GB
  - Given: page size: 2^{12} bytes = 4KB
  - **# entries in PageTable?**
  - **size of PageTable?**
  - *This is one, big contiguous array, by the way!*
- Physical address space:
  - Given: total physical memory: 2^{29} bytes = 512MB
  - overhead for 10 processes?

But Wait... There's more!

- Page Table Entry won't be just an integer
- Meta-Data
  - Valid Bits
    - *What PPN means "not mapped"?* No such number…
    - **At first:** not all virtual pages will be in physical memory
    - **Later:** might not have enough physical memory to map all virtual pages
  - Page Permissions
    - R/W/X permission bits for each PTE
    - **Code:** read-only, executable
    - **Data:** writeable, not executable

Less Simple Page Table



Process tries to access a page without proper permissions

Segmentation Fault

Example:

Write to read-only? → process killed

Now how big is this Page Table?

`struct pte_t page_table[`2^{20}`]`

`Each PTE = 8 bytes`

How many pages in memory will the page table take up?

Clicker Question:

(a) 4 million (2^{22}) pages

(b) 2048 (2^{11}) pages

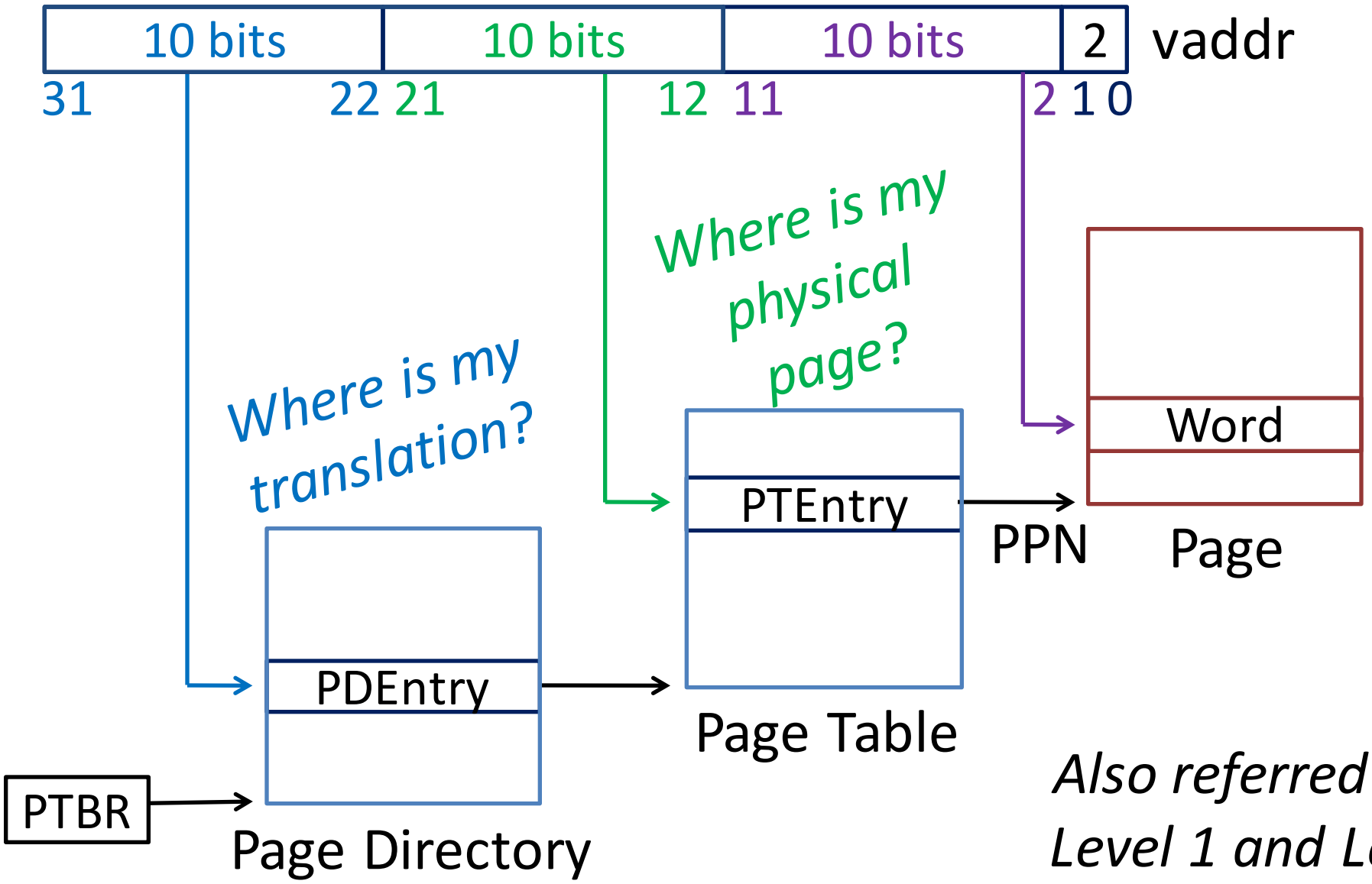
(c) 1024 (2^{10}) pages

(d) 4 billion (2^{32}) pages

(e) 4K (2^{12}) pages

Assuming each page = 4KB

Multi-Level Page Table



Also referred to as Level 1 and Level 2 Page Tables

* *Indirection to the Rescue, AGAIN!*

Multi-Level Page Table

Doesn't this take up more memory than before?

Benefits

- Don't need 4MB contiguous physical memory
- Don't need to allocate every PageTable, only those containing valid PTEs

Drawbacks

- Performance: Longer lookups

Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

- Address Translation
- Overhead
- Paging
- Performance
- Virtual Memory & Caches

Paging

What if process requirements > physical memory?

Virtual starts earning its name

Memory acts as a cache for secondary storage (disk)

- Swap memory pages out to disk when not in use
- Page them back in when needed

Courtesy of Temporal & Spatial Locality (again!)

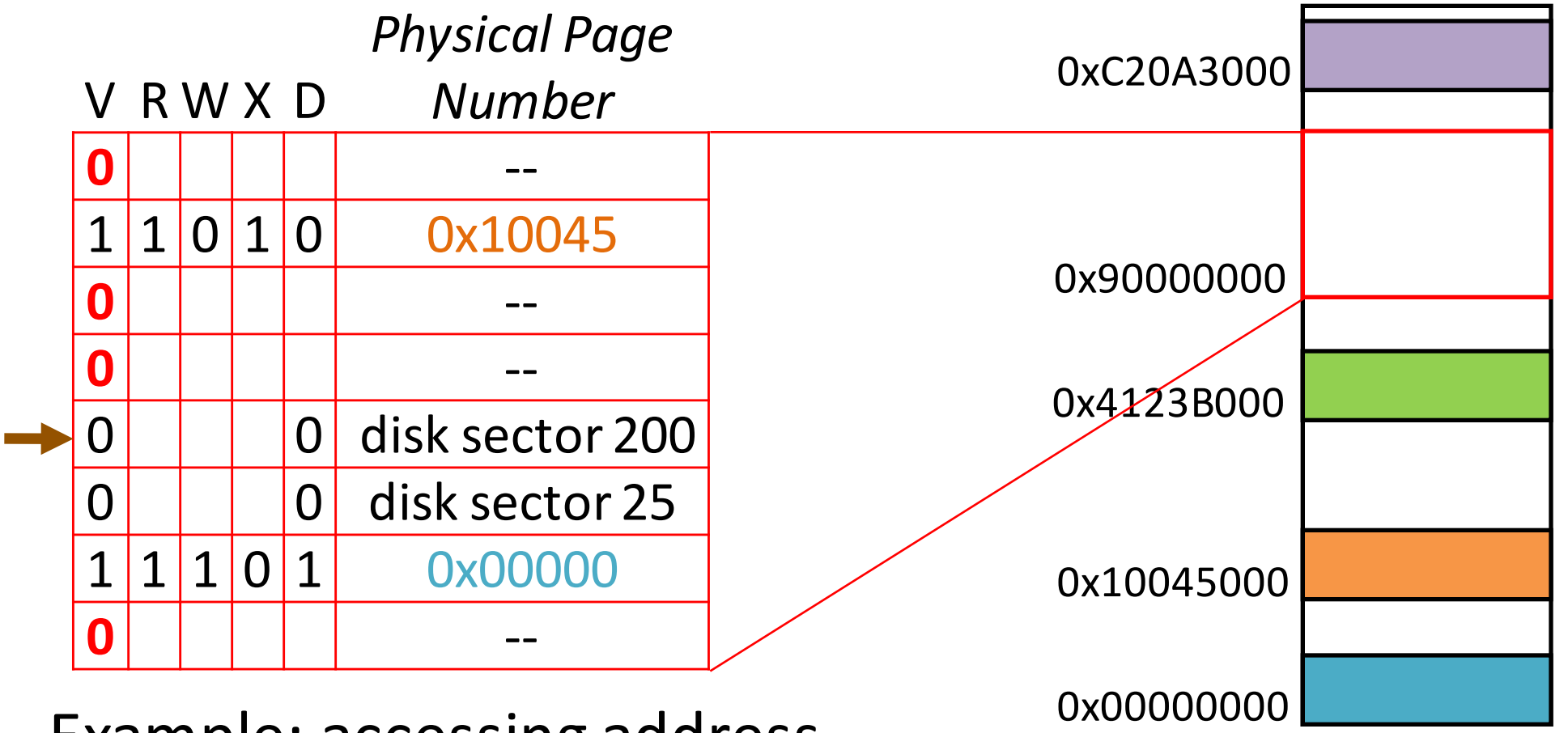
- Pages used recently mostly likely to be used again

More Meta-Data:

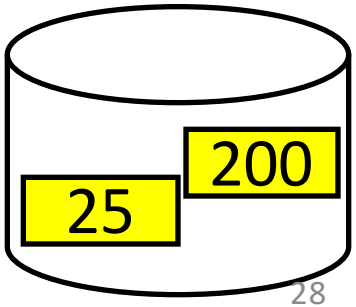
- Dirty Bit, Recently Used, *etc.*
- OS may access this meta-data to choose a victim

Paging

V	R	W	X	D	*Physical Page Number*
0					--
1	1	0	1	0	0x10045
0					--
0					--
0				0	disk sector 200
0				0	disk sector 25
1	1	1	0	1	0x00000
0					--



Example: accessing address beginning with 0x00003 (PageTable[3]) results in a Page Fault which will page the data in from disk sector 200



Page Fault

Valid bit in Page Table = 0

→ means page is not in memory

OS takes over:

- Choose a physical page to replace
  - **"Working set"**: refined LRU, tracks page usage
- If dirty, write to disk
- Read missing page from disk
  - Takes so long (~10ms), OS schedules another task

Performance-wise page faults are *really* bad!

Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

- Address Translation
- Overhead
- Paging
- Performance
- Virtual Memory & Caches

Watch Your Performance Tank!

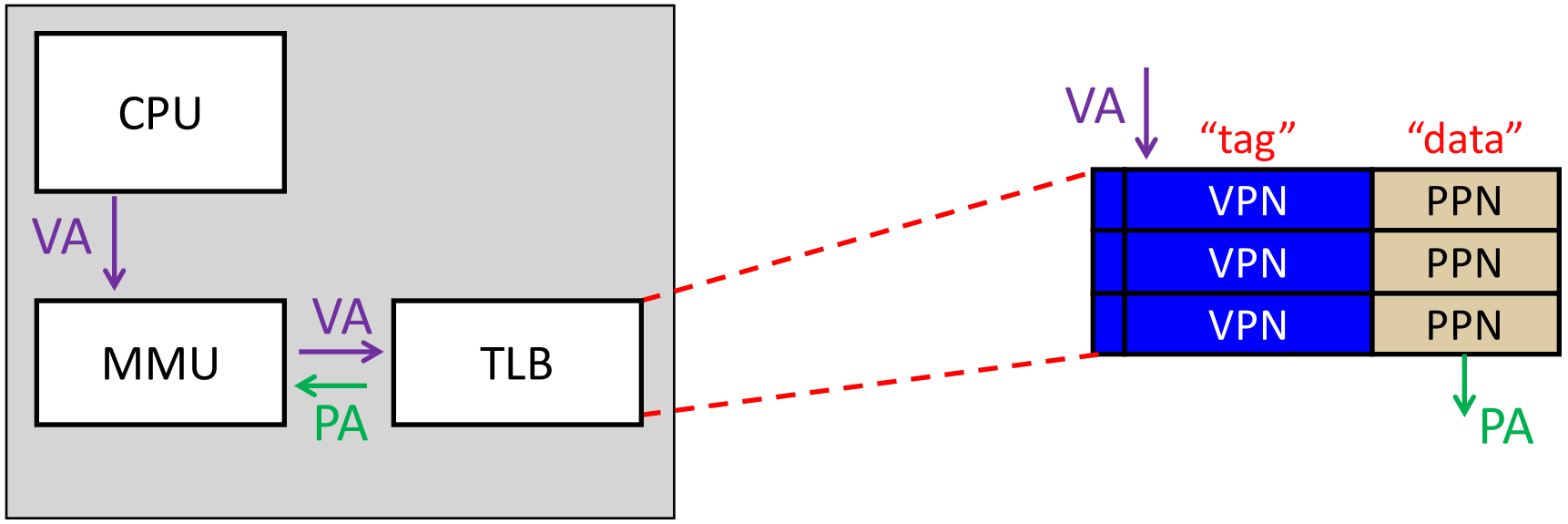
For *every instruction*:

- MMU translates address (virtual → physical)
  - Uses PTBR to find Page Table in memory
  - Looks up entry for that virtual page
- Fetch the instruction using physical address
  - Access Memory Hierarchy (I$ → L2 → Memory)

- Repeat at Memory stage for load/store insns
  - Translate address
  - **Now** you perform the load/store

Translation Lookaside Buffer (TLB)

- Small, fast cache
- Holds VPN→PPN translations
- Exploints temporal locality in pagetable
- TLB Hit: huge performance savings
- TLB Miss: invoke TLB miss handler
  - *Put translation in TLB for later*



TLB Parameters

Typical

- very small (64 – 256 entries) → *very fast*
- fully associative, or at least set associative
- tiny block size: why?

Example: Intel Nehalem TLB

- 128-entry L1 Instruction TLB, 4-way LRU
- 64-entry L1 Data TLB, 4-way LRU
- 512-entry L2 Unified TLB, 4-way LRU

TLB to the Rescue!

For every instruction:

- Translate the address (virtual → physical)
  - CPU checks TLB
  - That failing, walk the Page Table
    - Use PTBR to find Page Table in memory
    - Look up entry for that virtual page
    - Cache the result in the TLB
- Fetch the instruction using physical address
  - Access Memory Hierarchy (I$ → L2 → Memory)

- Repeat at Memory stage for load/store insns
  - CPU checks TLB, translate if necessary
  - **Now** perform load/store

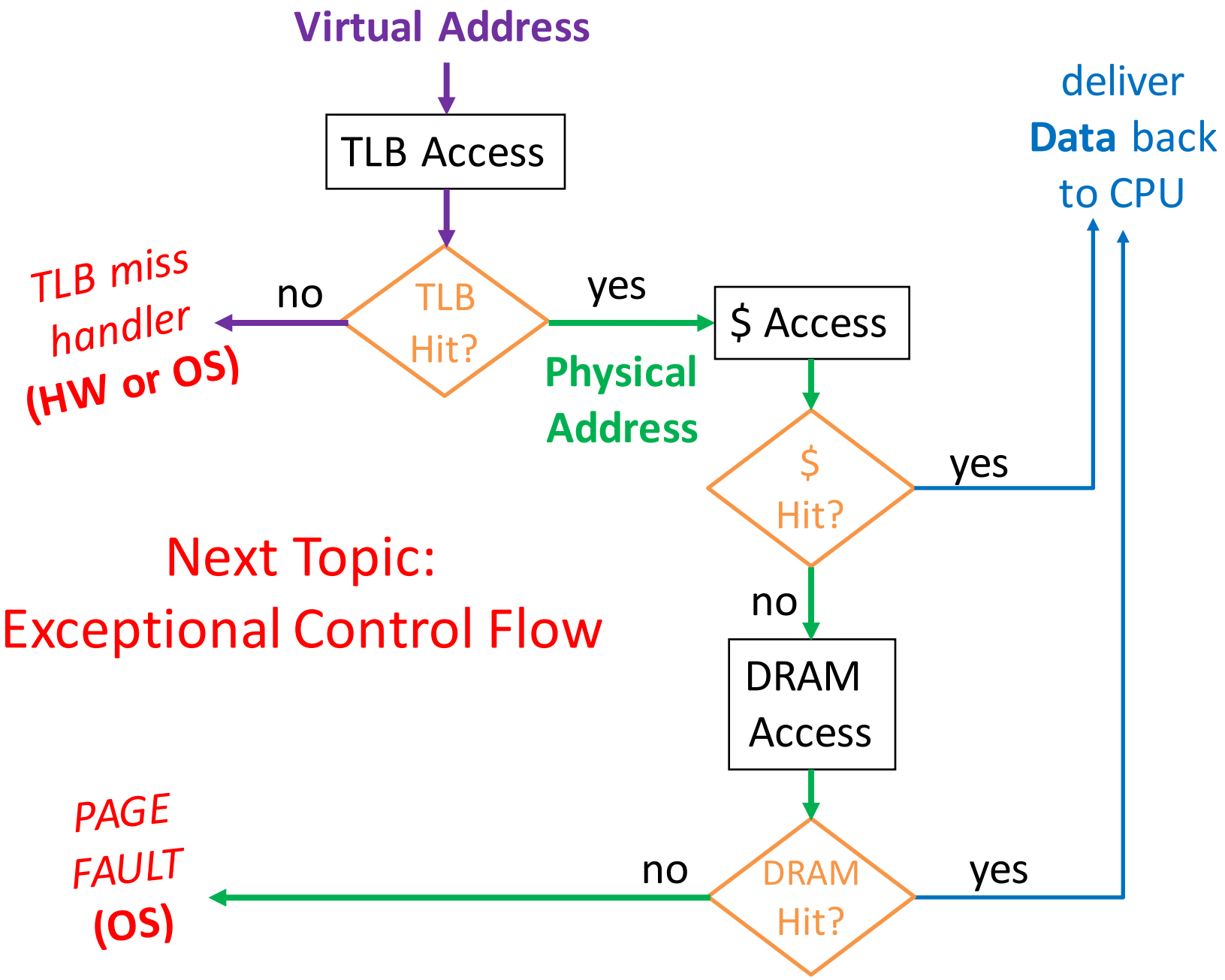
Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

- Address Translation
- Overhead
- Paging
- Performance
- Virtual Memory & Caches
  - Caches use physical addresses
  - Prevents sharing except when intended
  - *Works beautifully!*

Translation in Action



Next Topic:
Exceptional Control Flow

Takeaways

Need a map to translate a "fake" virtual address (from process) to a "real" physical Address (in memory).

The map is a **Page Table:** `ppn = PageTable[vpn]`

A page is constant size block of virtual memory. Often ~4KB to reduce the number of entries in a PageTable.

Page Table can enforce Read/Write/Execute permissions on a per page basis. Can allocate memory on a per page basis. Also need a valid bit, and a few others.

Space overhead due to Page Table is significant.
Solution: another level of indirection!
Two-level of Page Table significantly reduces overhead.

Time overhead due to Address Translations also significant.
Solution: caching! **Translation Lookaside Buffer (TLB)** acts as a cache for the Page Table and significantly improves performance.

November 1988: Internet Worm

- Internet Worm attacks thousands of Internet hosts
- Best Wikipedia quotes:

"According to its creator, the Morris worm was not written to cause damage, but to gauge the size of the Internet. The worm was released from MIT to disguise the fact that the worm originally came from Cornell."

"The worm …determined whether to invade a new computer by asking whether there was already a copy running. But just doing this would have made it trivially easy to kill: everyone could run a process that would always answer "yes". To compensate for this possibility, Morris directed the worm to copy itself even if the response is "yes" 1 out of 7 times. This level of replication proved excessive, and the worm spread rapidly, infecting some computers multiple times. Morris remarked, when he heard of the mistake, that he "should have tried it on a simulator first"."

Computer Virus TV News Report 1988