Computer Architecture

Week 13: Virtual Memory and System Calls

Fenerbahçe University

Professor & TAs

Prof: Dr. Vecdi Emre Levent

Office: 311

Email: emre.levent@fbu.edu.tr

TA: Arş. Gör. Uğur Özbalkan

Office: 311

Email: ugur.ozbalkan@fbu.edu.tr

Course Plan

- Virtual Memory
- System Calls

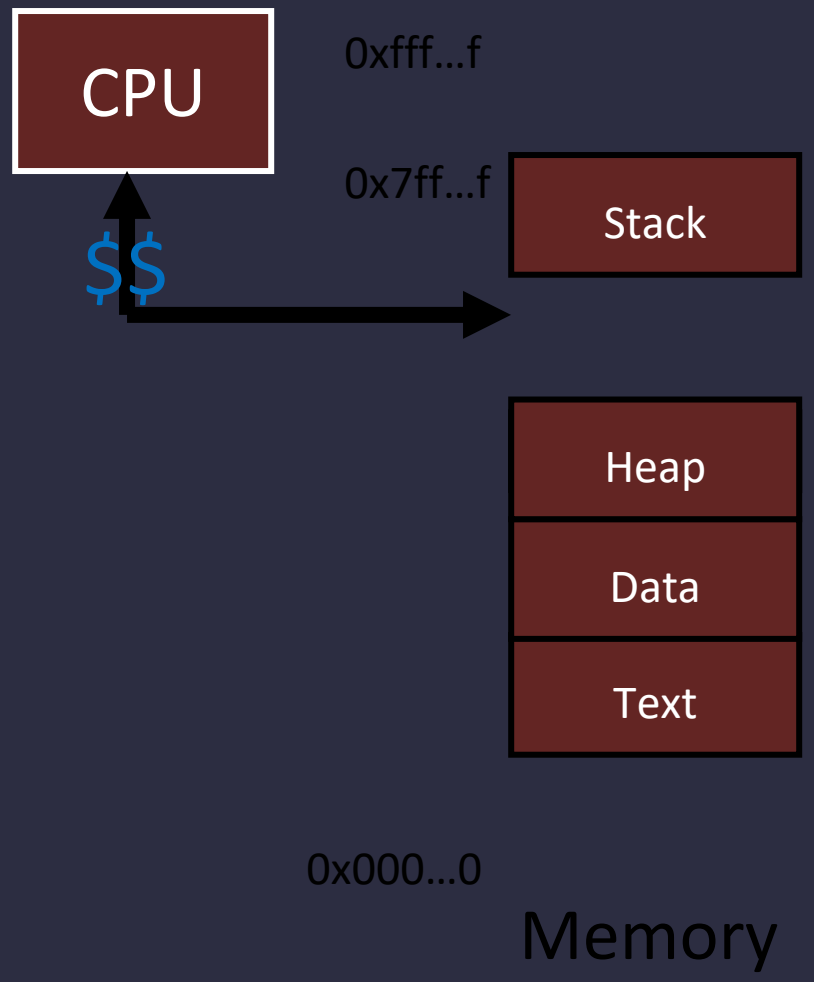
Big Picture: Multiple Processes

How to run multiple processes?

- *Time-multiplex* a single CPU core (multi-tasking)
  - Web browser, skype, office, … all must co-exist
- Many cores per processor (multi-core)
  or many processors (multi-processor)
  - Multiple programs run *simultaneously*

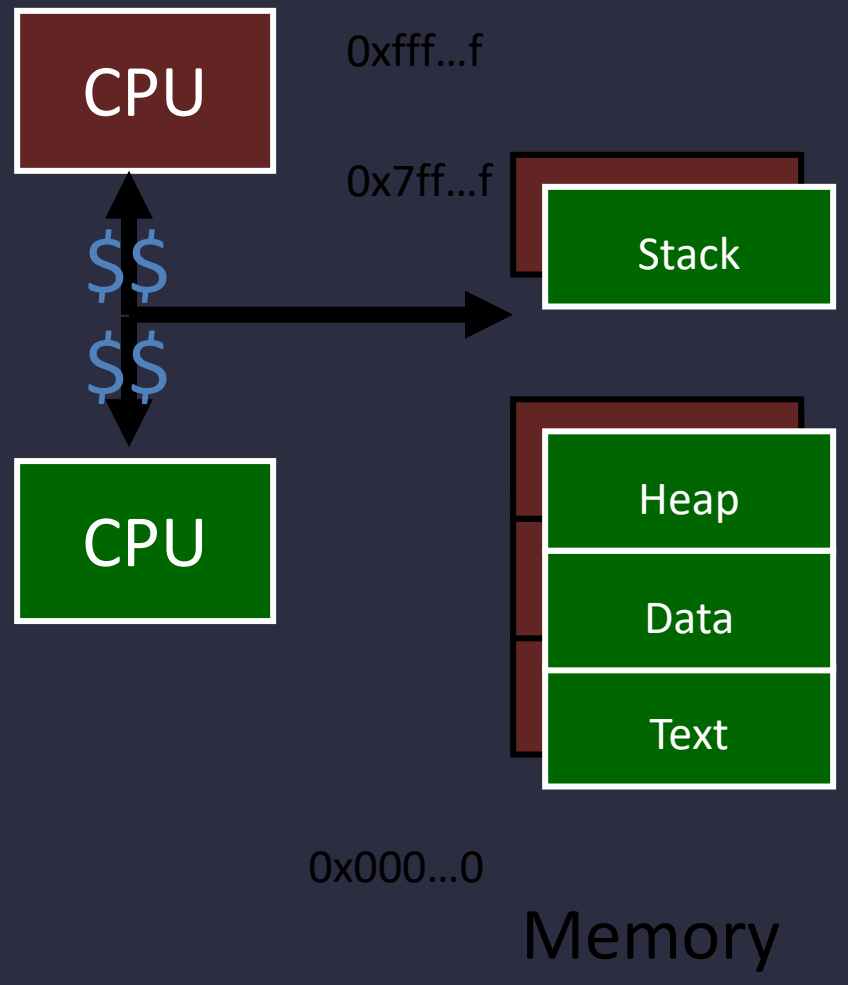
Processor & Memory

- CPU address/data bus…
- … routed through caches
- … to main memory
  - Simple, fast, but…



Multiple Processes

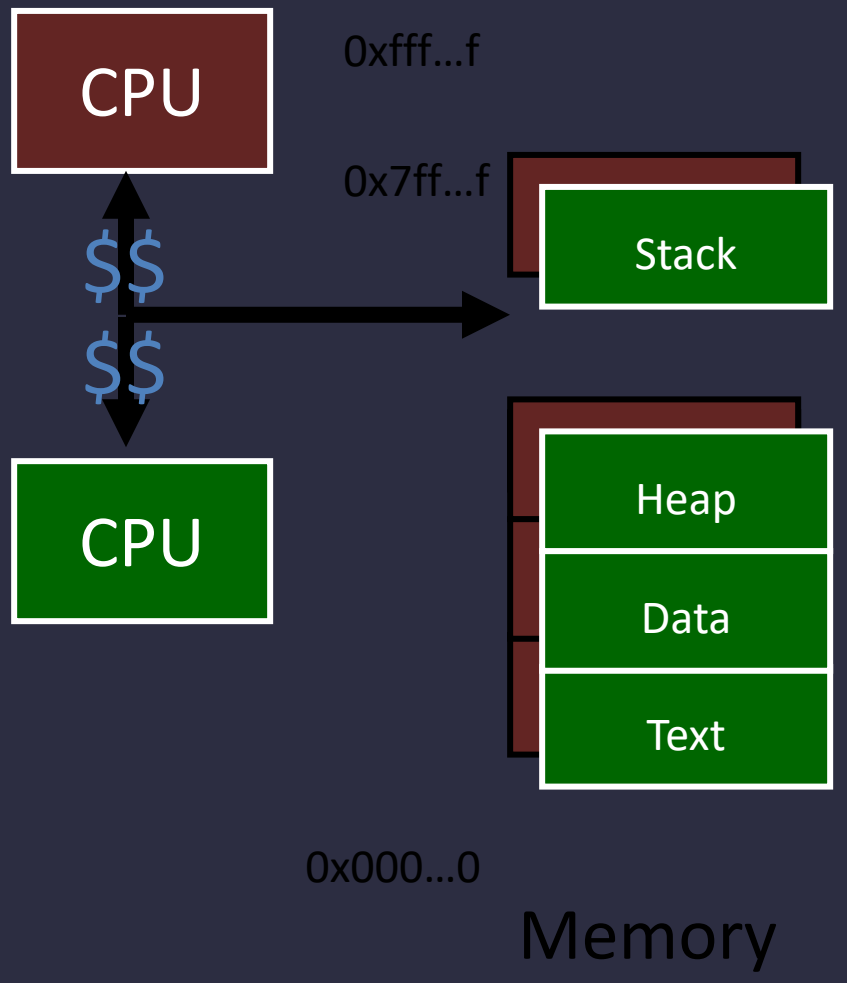
- Q: What happens when another program is executed concurrently on another processor?



- A: The addresses will conflict
  - Even though, CPUs may take turns using memory bus

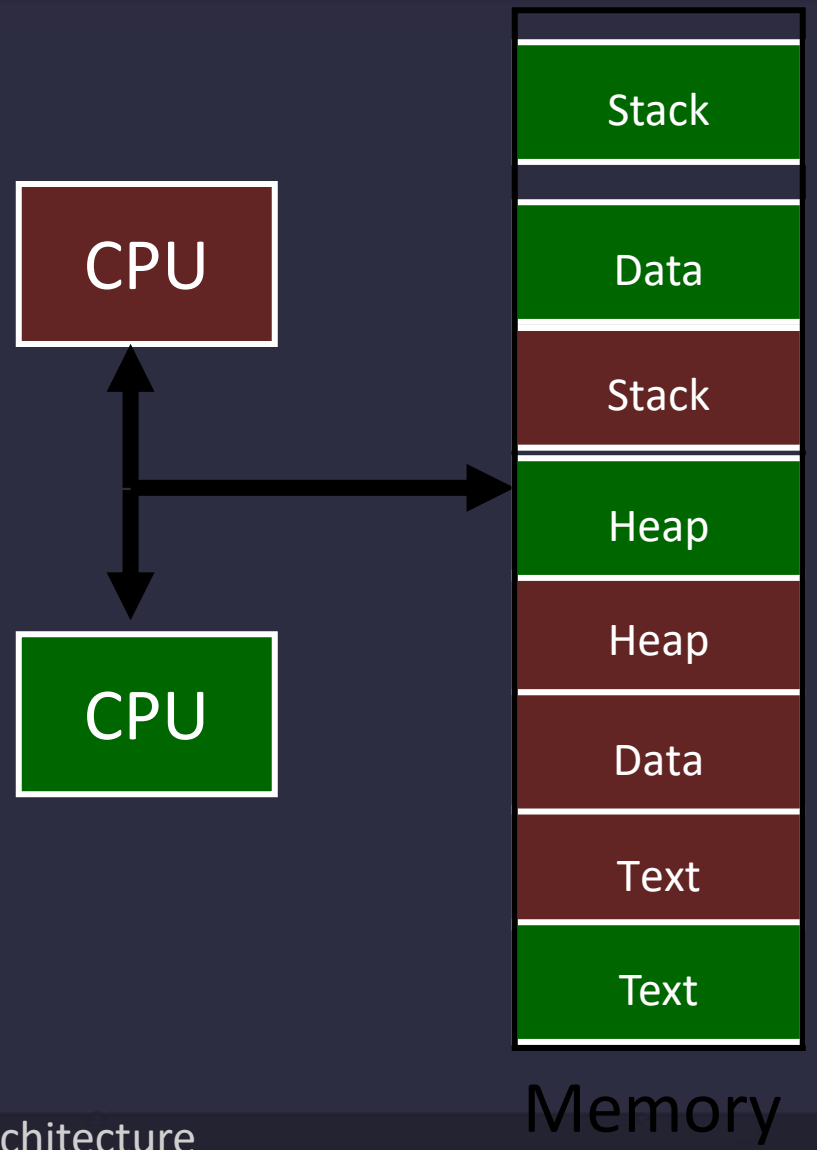
Multiple Processes

- Q: Can we relocate second program?

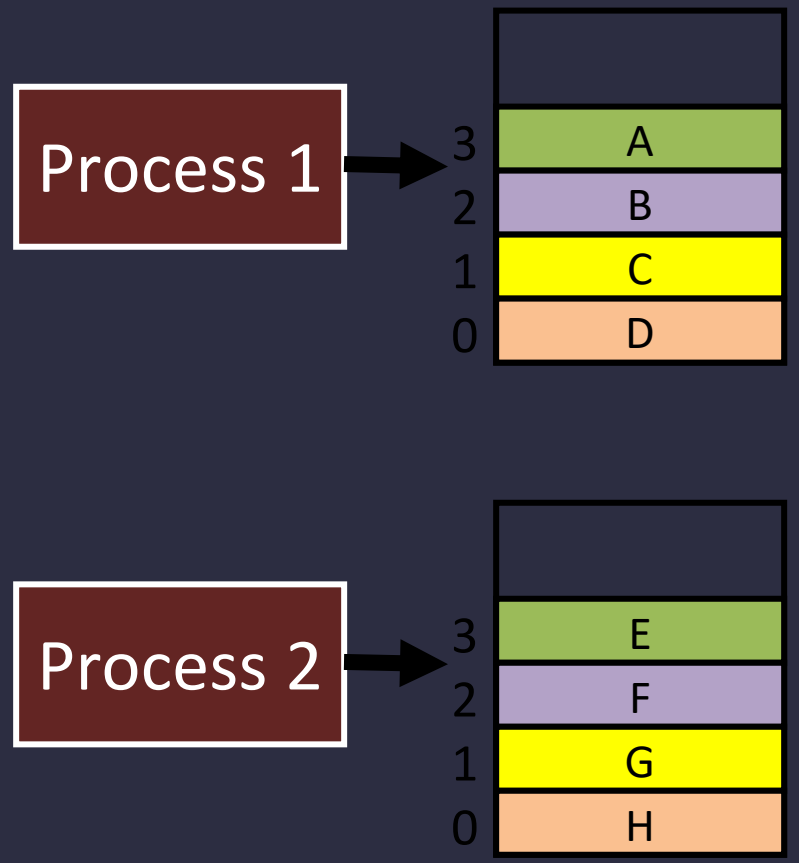


Solution? Multiple processes/processors

- Q: Can we relocate second program?
- A: Yes, but…
  - What if they don't fit?
  - What if not contiguous?
  - Need to recompile/relink?
  - …



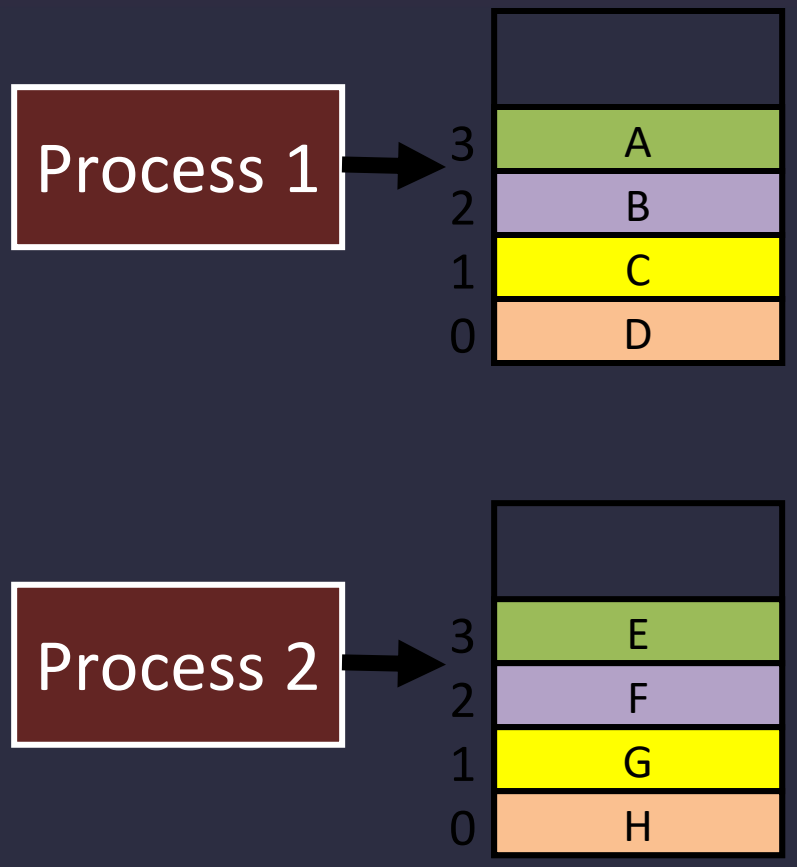
Big Picture: (Virtual) Memory

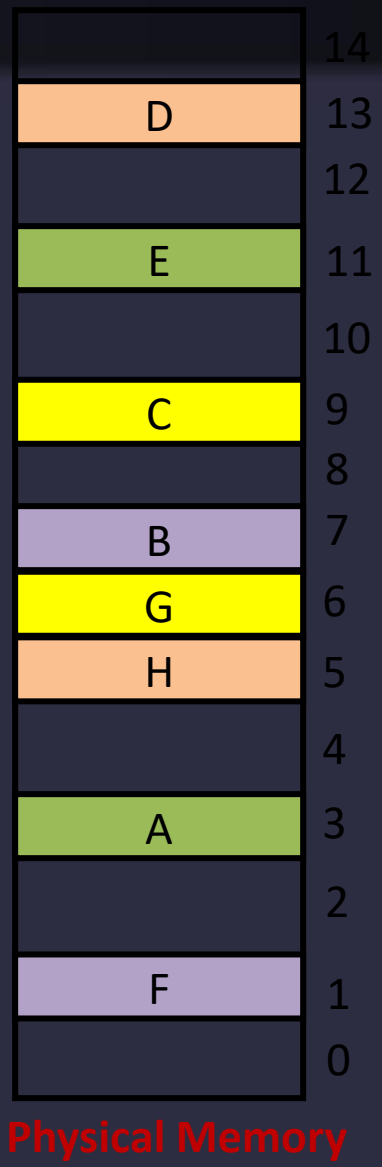


Give each process an illusion that it has exclusive access to entire main memory

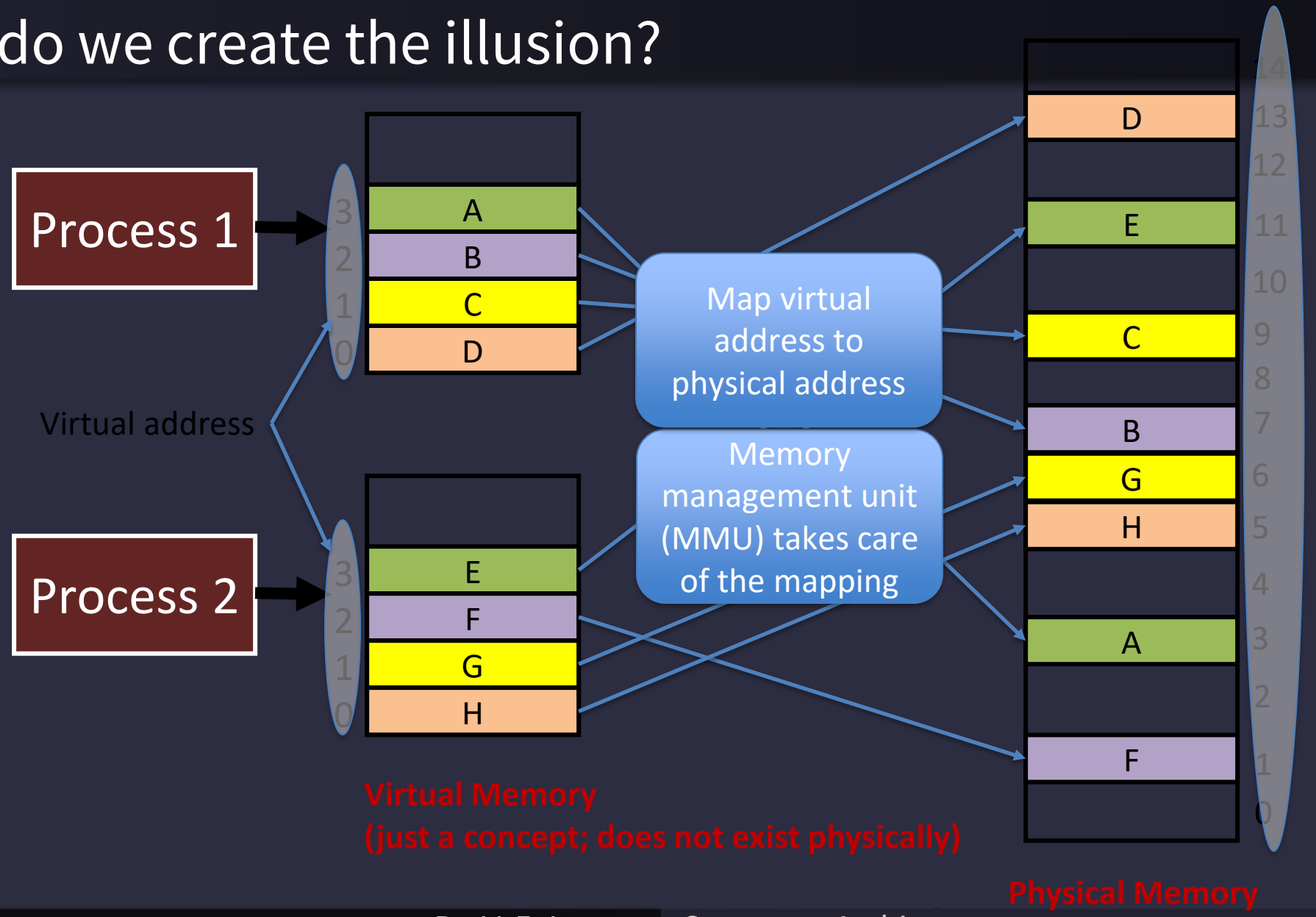
But In Reality…

How do we create the illusion?

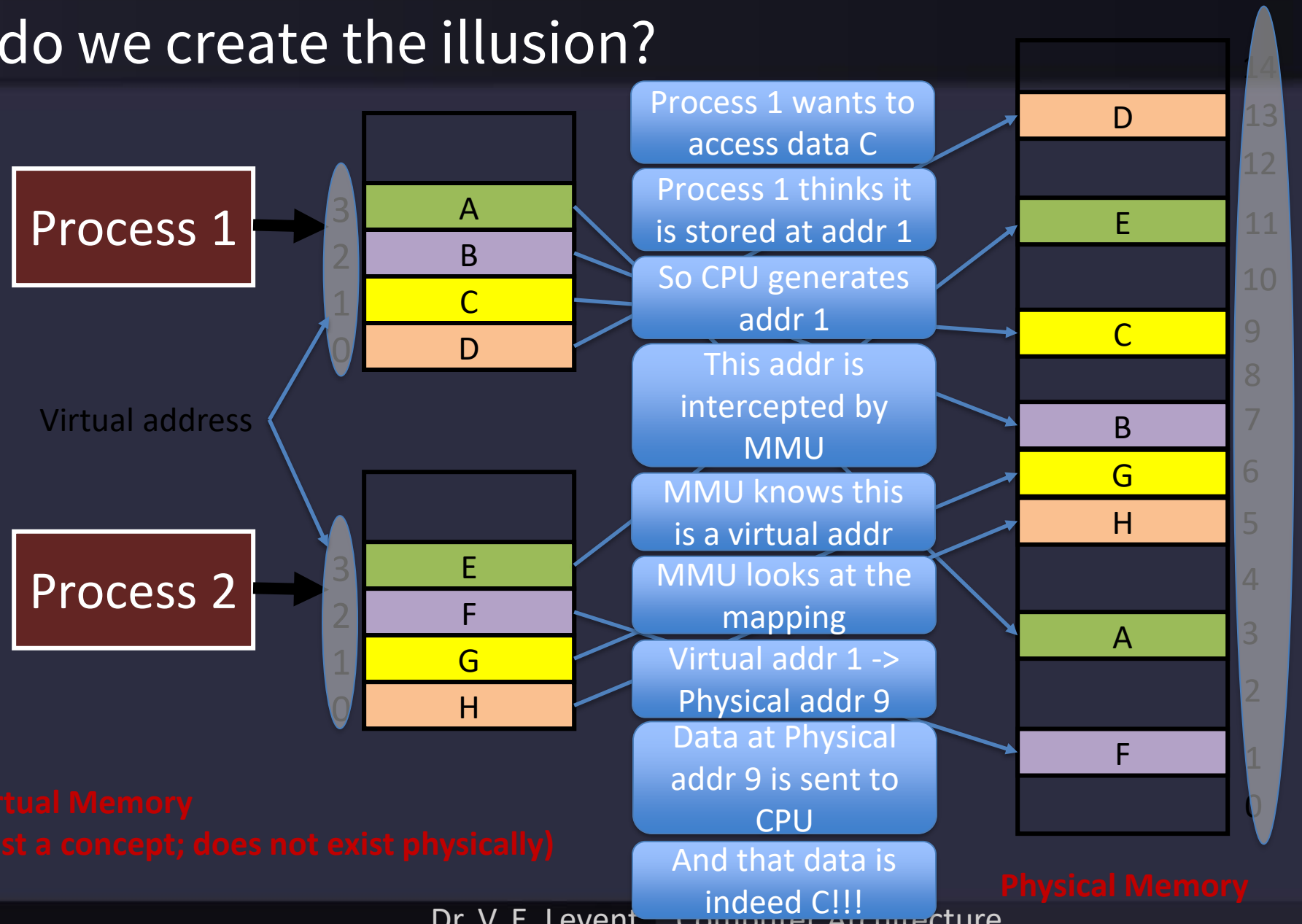




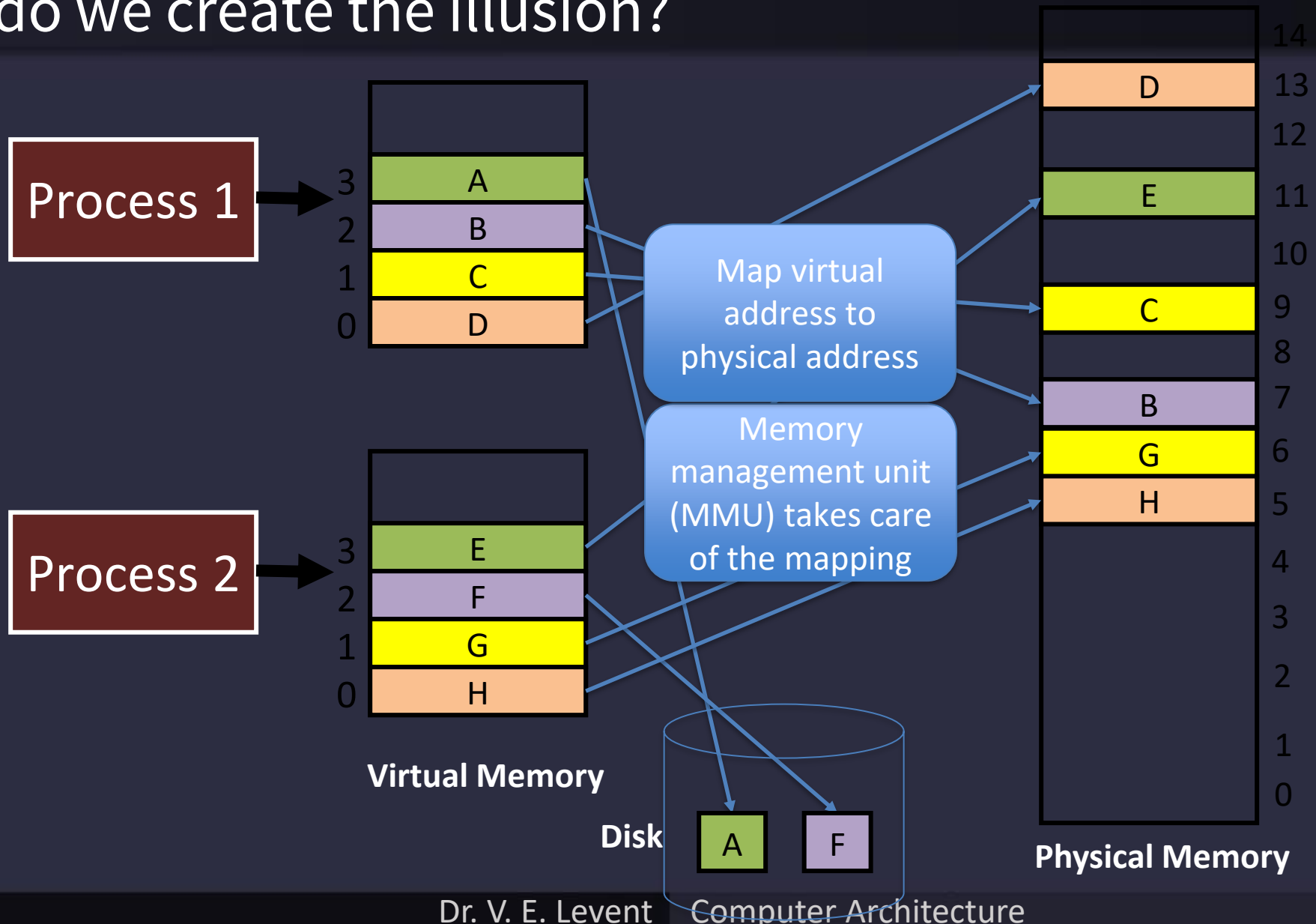
How do we create the illusion?



How do we create the illusion?

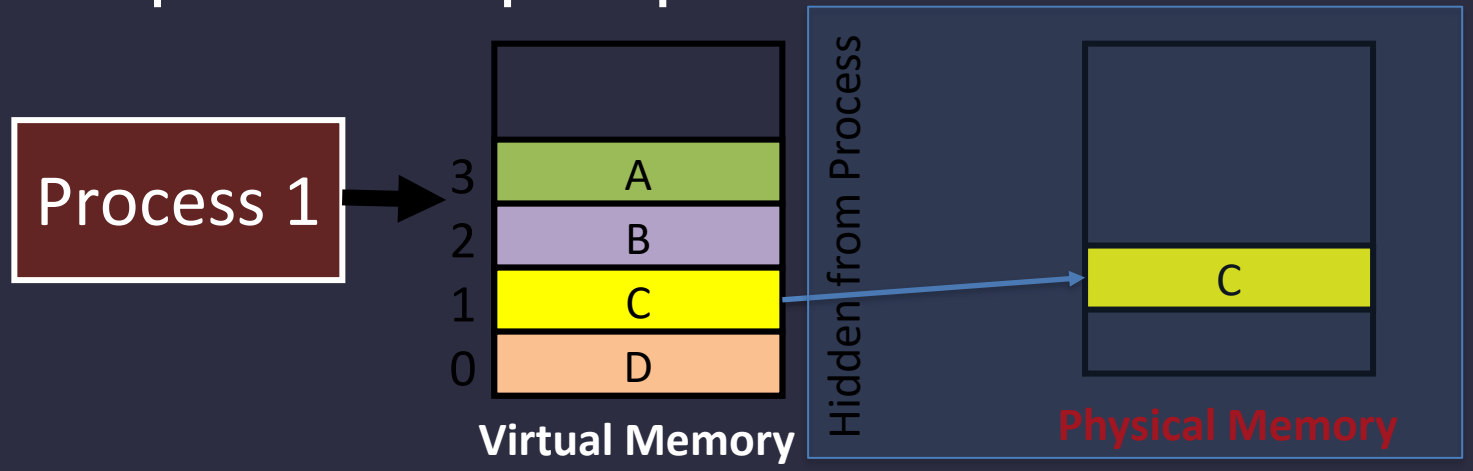


How do we create the illusion?



Big Picture: (Virtual) Memory

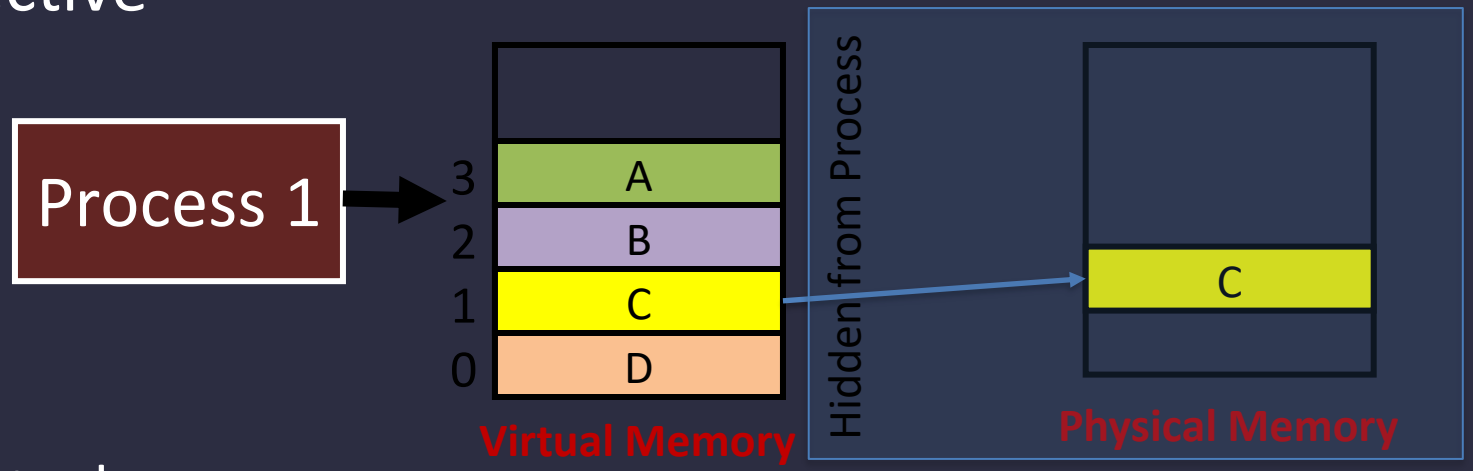
- From a process's perspective –



  - Process only sees the virtual memory
    - ✓Contiguous memory

Big Picture: (Virtual) Memory

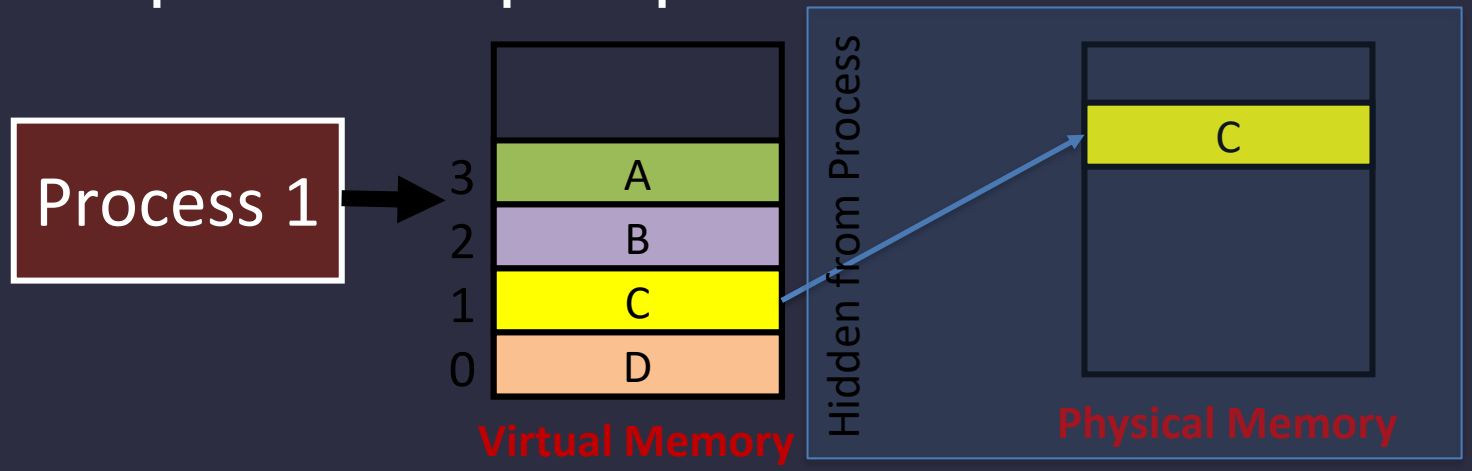
- From a process's perspective –



  - Process only sees the virtual memory
    - ✓ Contiguous memory
    - ✓ No need to recompile - only mappings need to be updated

Big Picture: (Virtual) Memory

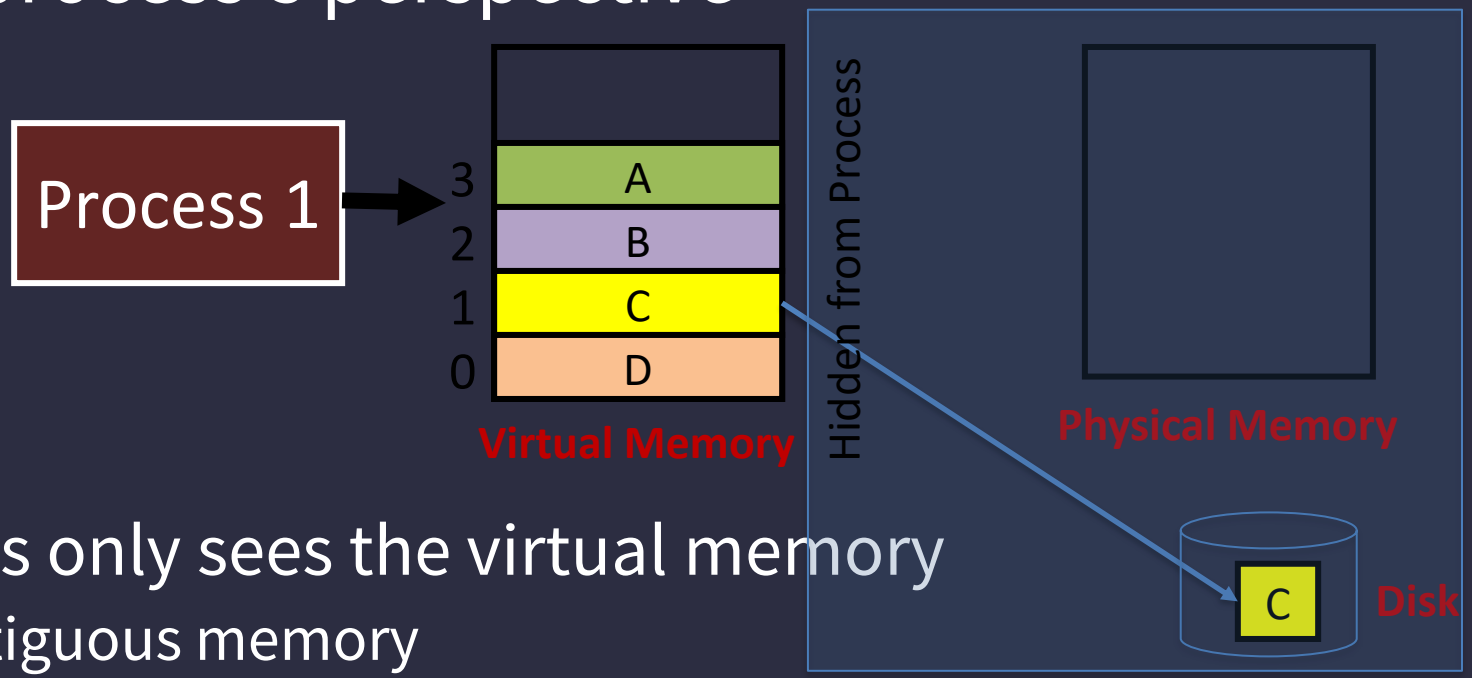
- From a process's perspective –



  - Process only sees the virtual memory
    - ✓Contiguous memory
    - ✓No need to recompile - only mappings need to be updated

Big Picture: (Virtual) Memory

- From a process's perspective –



  - Process only sees the virtual memory
    - ✓Contiguous memory
    - ✓No need to recompile - only mappings need to be updated

Next Goal

- How does Virtual Memory work?

- How do we create the “map” that maps a virtual address generated by the CPU to a physical address used by main memory?

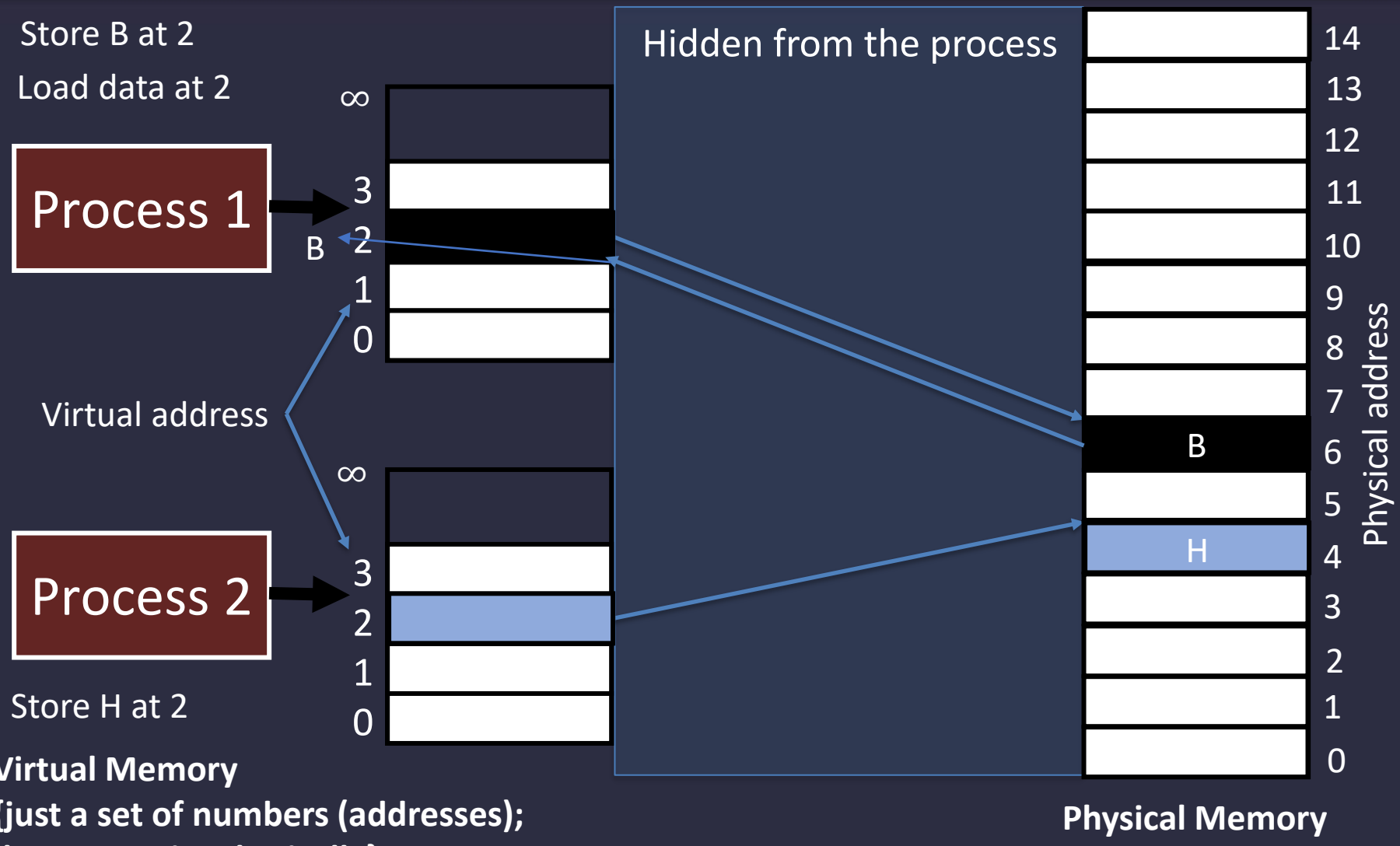
Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory work?

- Address Translation
- Overhead
- Paging
- Performance

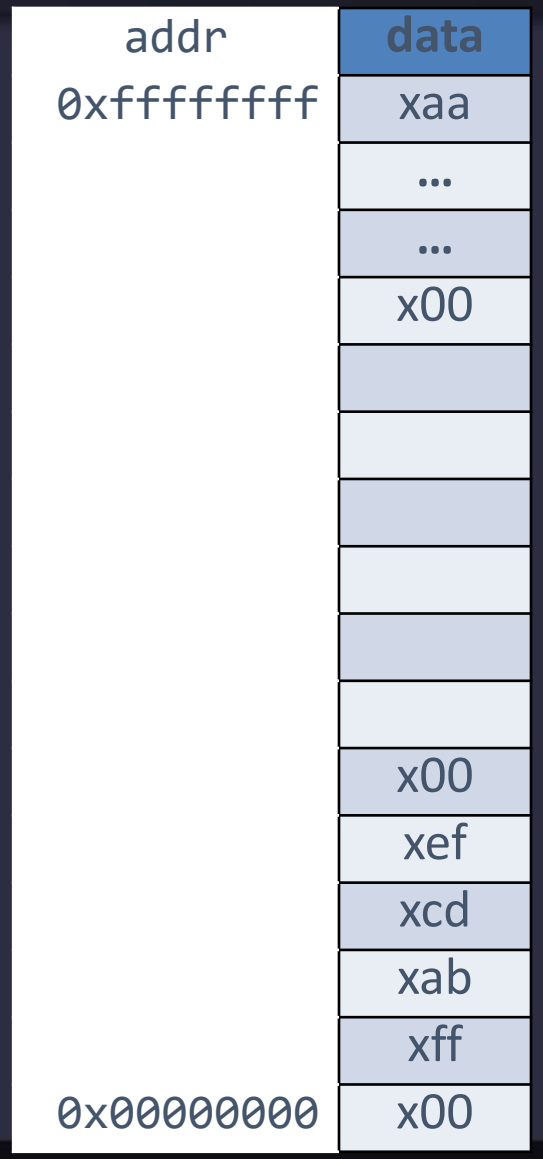
Virtual Memory



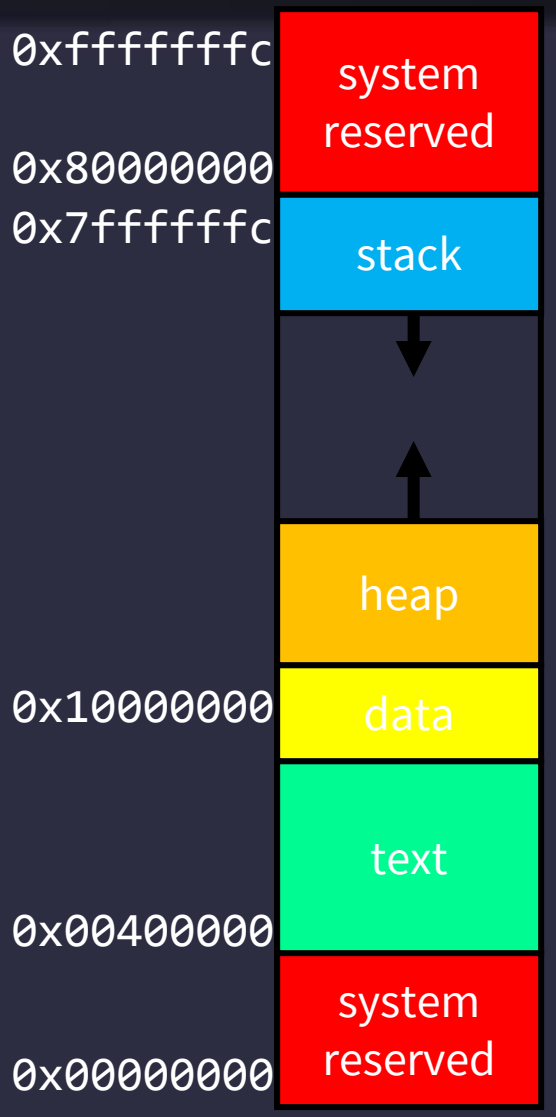
Virtual Memory
{just a set of numbers (addresses); does not exist physically}

Picture Memory as… ?

Byte Array:

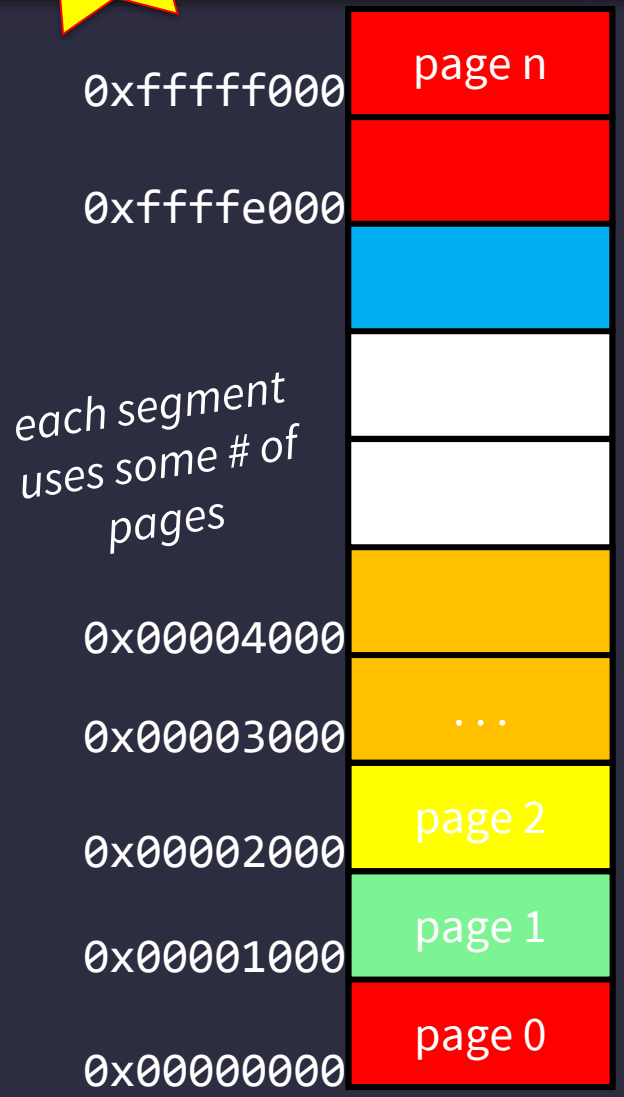


Segments:



New!

Page Array:



A Little More About Pages

Memory size = depends on system
say 4GB

Page size = 4KB (by default)

Then, # of pages = 2^20

Any data in a page # 2 has address of the form:
0x00002xxx

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

```
0x00002200  = 0010 0000 0000
                    = byte 512
```

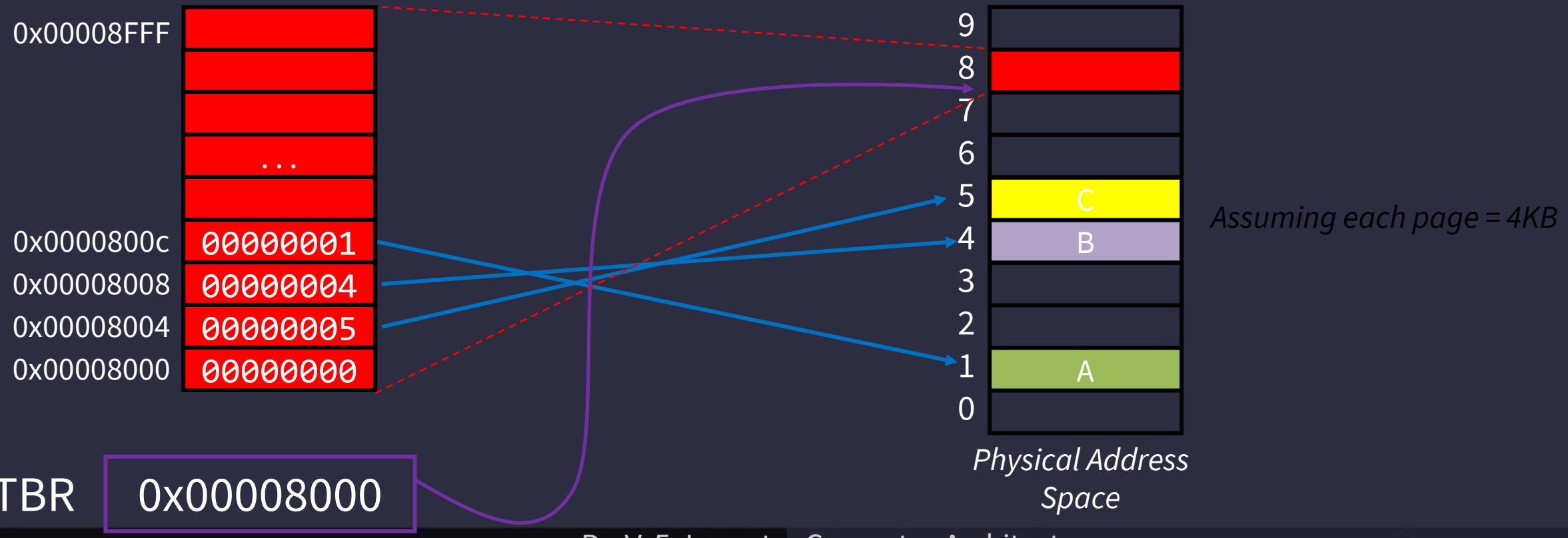
upper bits = page number (PPN)
lower bits = page offset

Page Table: Data structure to store mapping

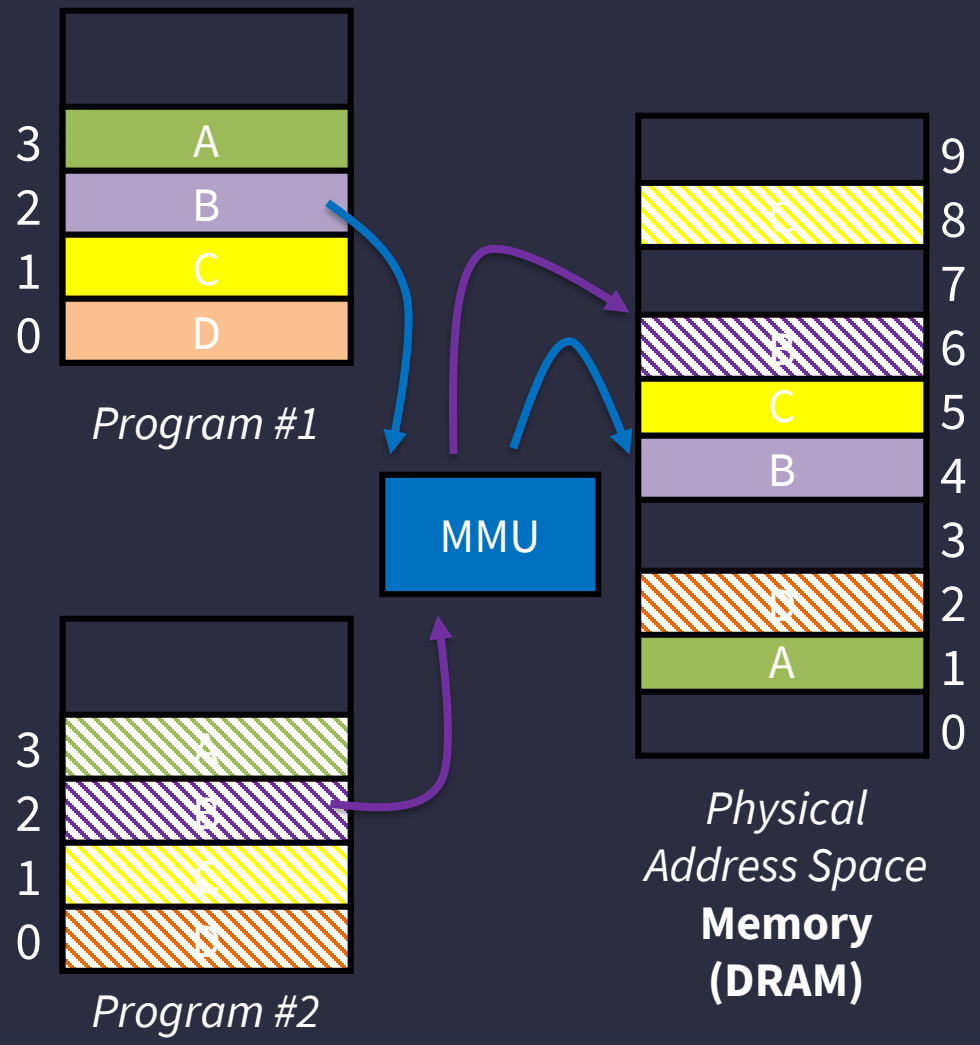
1 Page Table *per process*

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

Location stored in **Page Table Base Register**



Address Translator: MMU

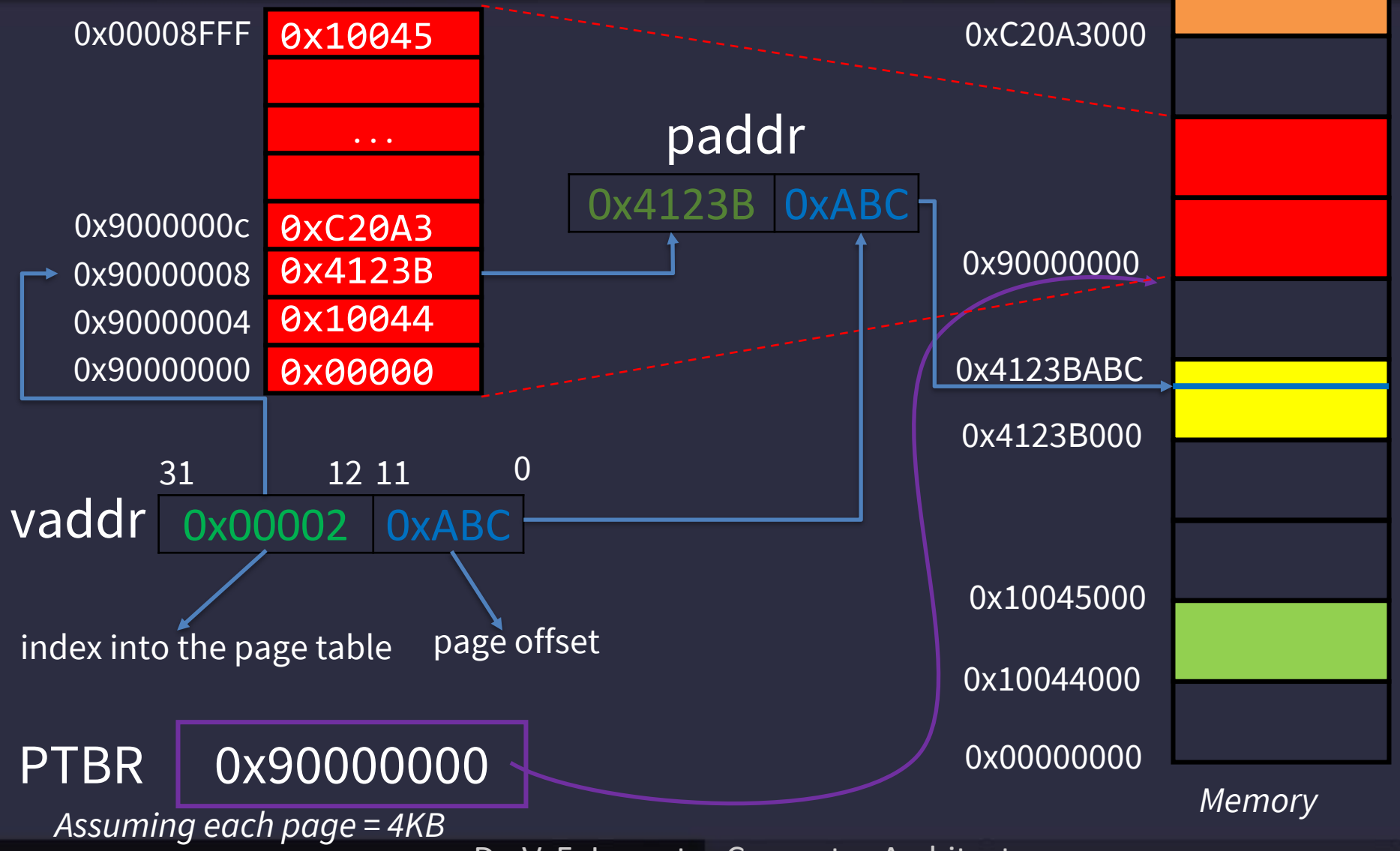


- Programs use virtual addresses
- Actual memory uses physical addresses

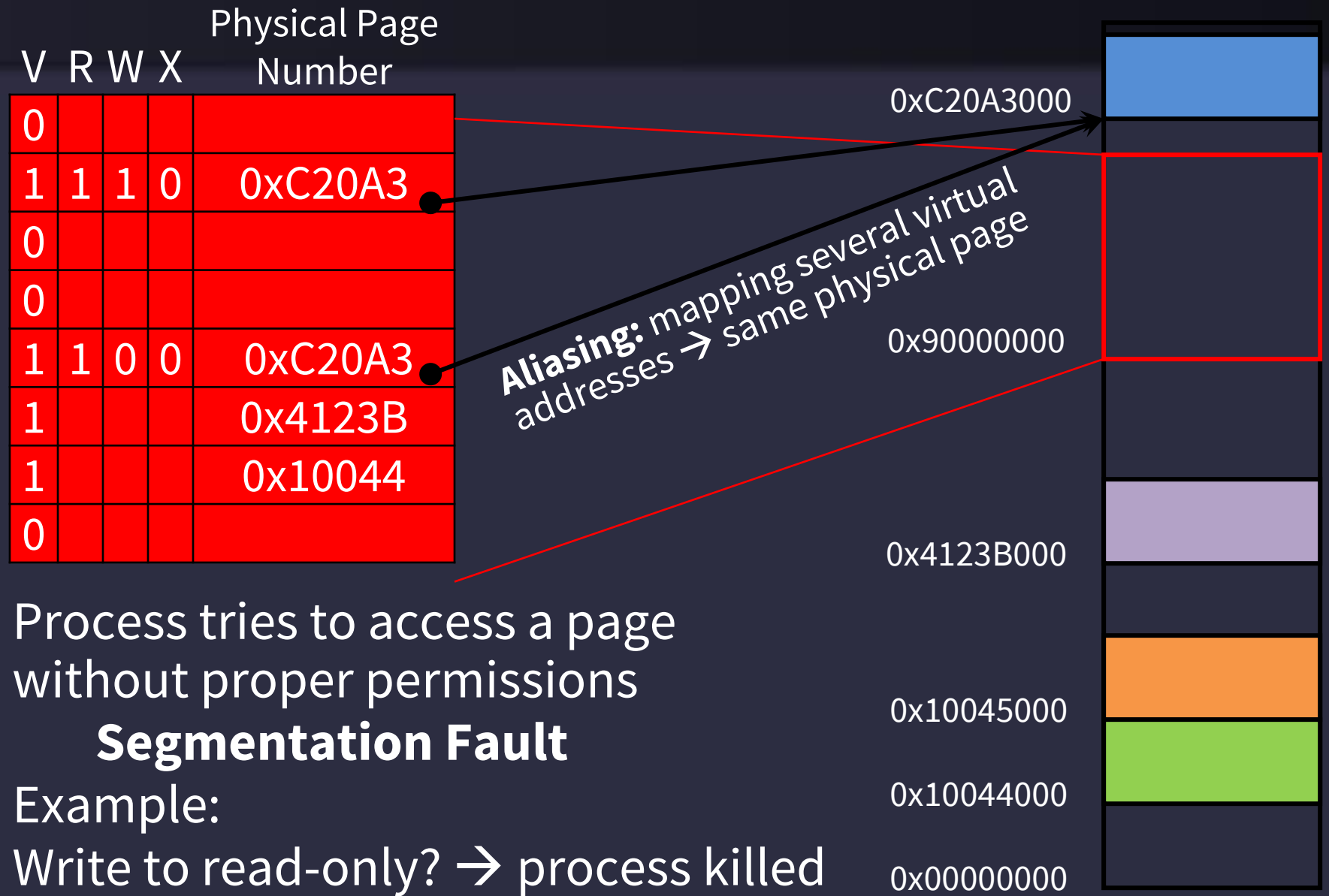
Memory Management Unit (MMU)

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

Simple Page Table Translation



Less Simple Page Table



Process tries to access a page without proper permissions

Segmentation Fault

Example:

Write to read-only? → process killed

Wait, how big is this Page Table?

page_table[2^{20}] = 8x2^{20} =2^{23} bytes

(Page Table = 8 MB in size)

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

2^{23} /2^{12} =2^{11} 2K pages! *Assuming each page = 4KB*

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!)

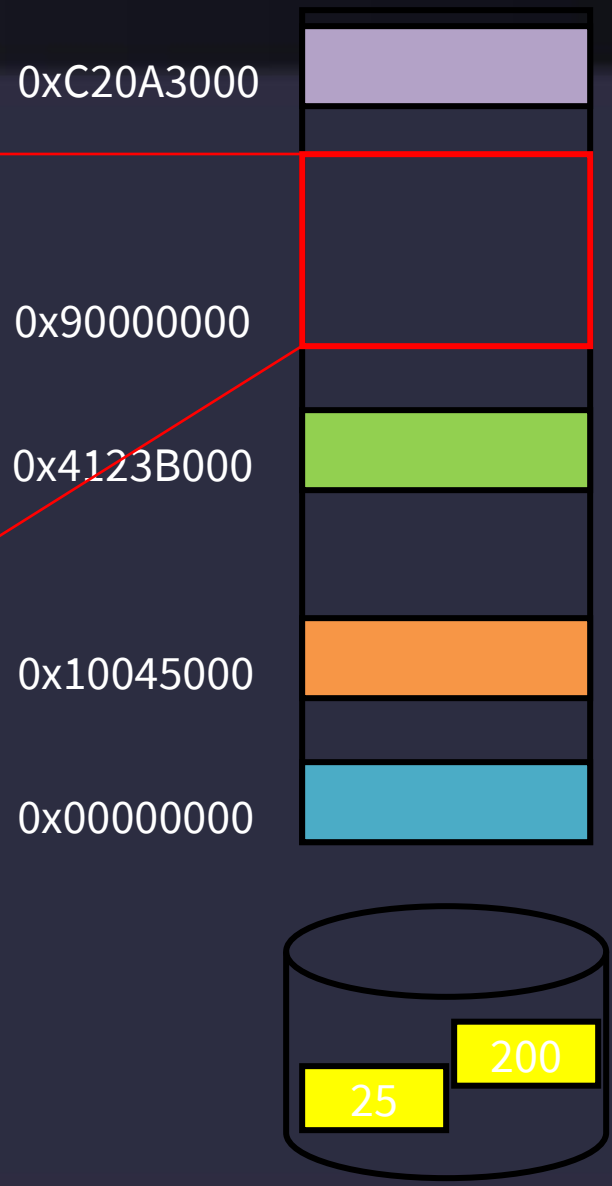
- Pages used recently mostly likely to be used again

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!

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

Performance

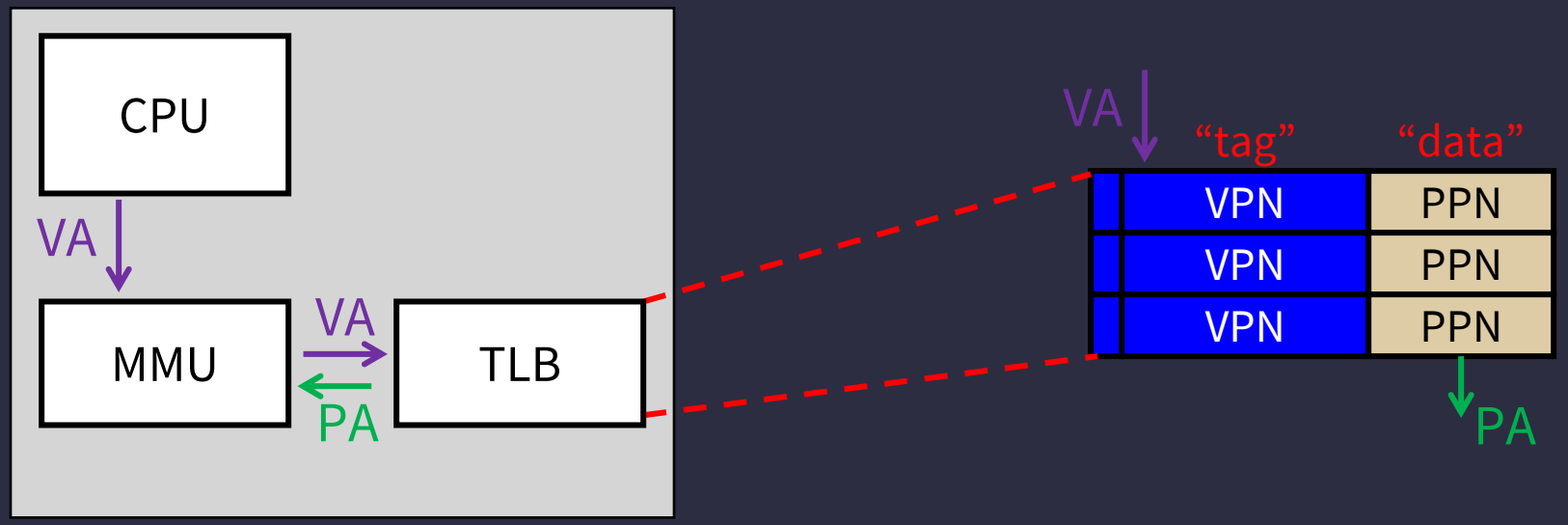
- Virtual Memory Summary
- PageTable for each process:
  - Page
    - Single-level (e.g. 4MB contiguous in physical memory)
    - or multi-level (e.g. less mem overhead due to page table),
    - …
  - every load/store translated to physical addresses
  - page table miss: load a swapped-out page and retry instruction, or kill program
- Performance?
  - terrible: memory is already slow
translation makes it slower
- Solution?
  - A cache, of course

Next Goal

- How do we speedup address translation?

Translation Lookaside Buffer (TLB)

- Small, fast cache
- Holds VPN→PPN translations
- Exploits 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

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