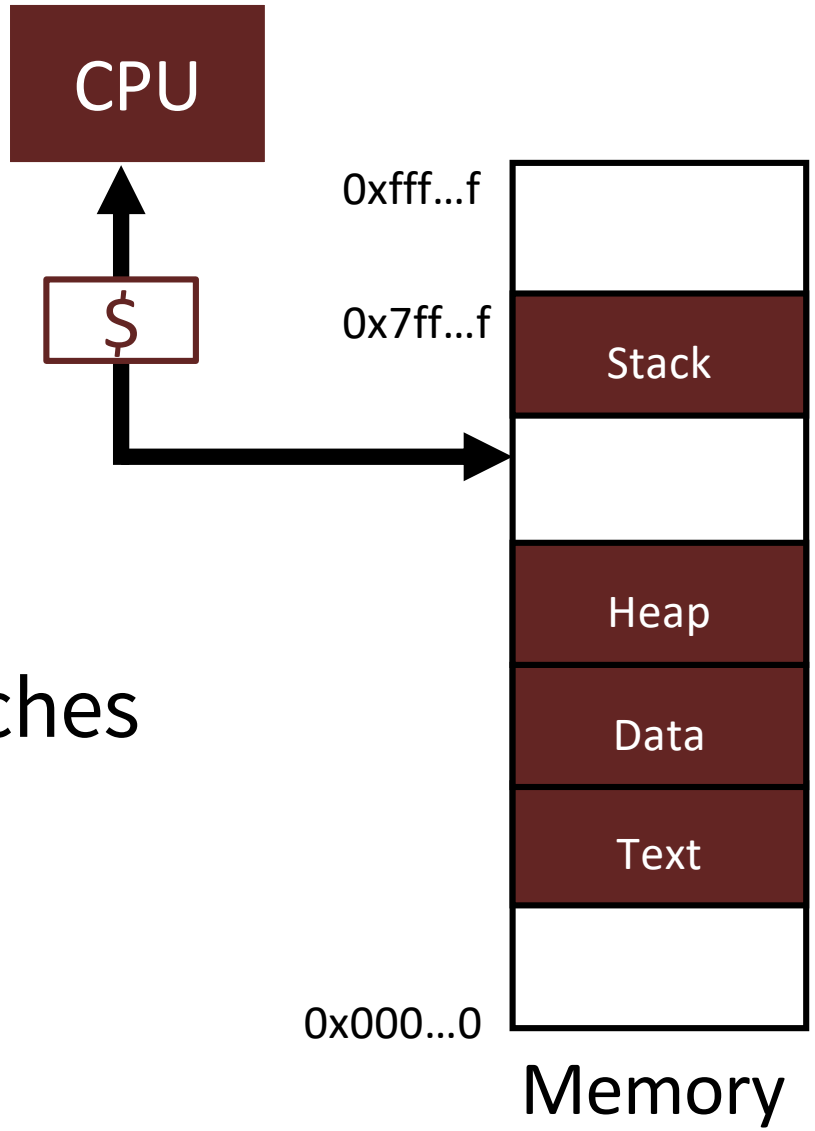
Virtual Memory

ECE 4750
Computer Architecture

[K. Bala, A. Bracy, E. Sirer, H. Weatherspoon, D. Zagieboylo]

Processor & Memory



CPU address/data bus…

… routed through caches

… to main memory

Simple, fast, but…

Multiple Processes

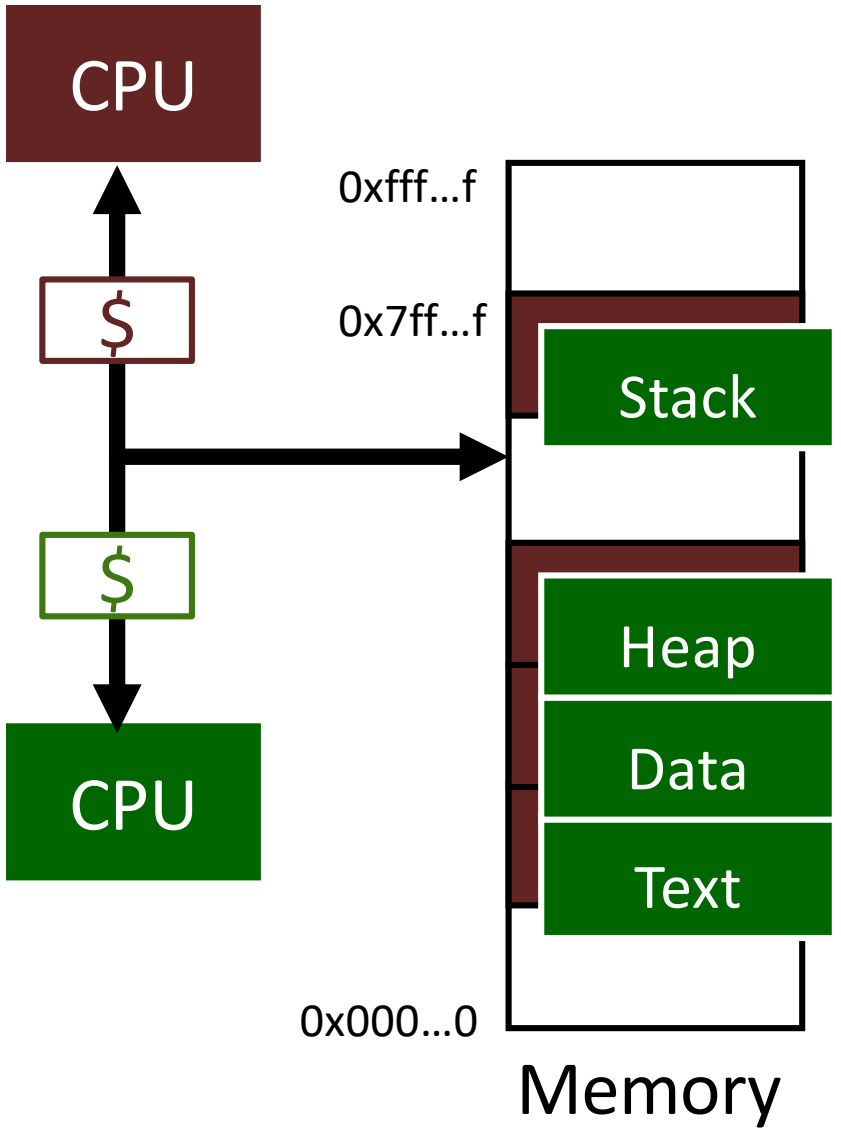
Q: What happens when another program is executed concurrently on another processor?

A: Addresses will conflict

Even if CPUs take turns using memory bus

Solutions?

Can we relocate second program?



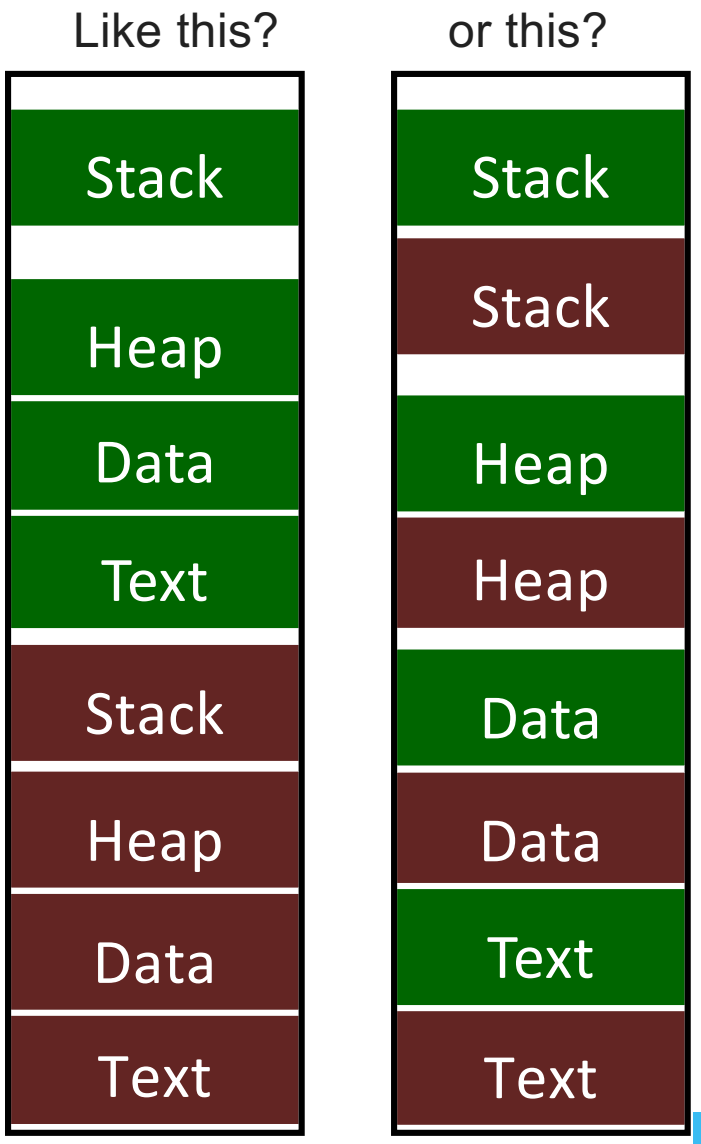
Can we relocate second program?

Yes, but… how?

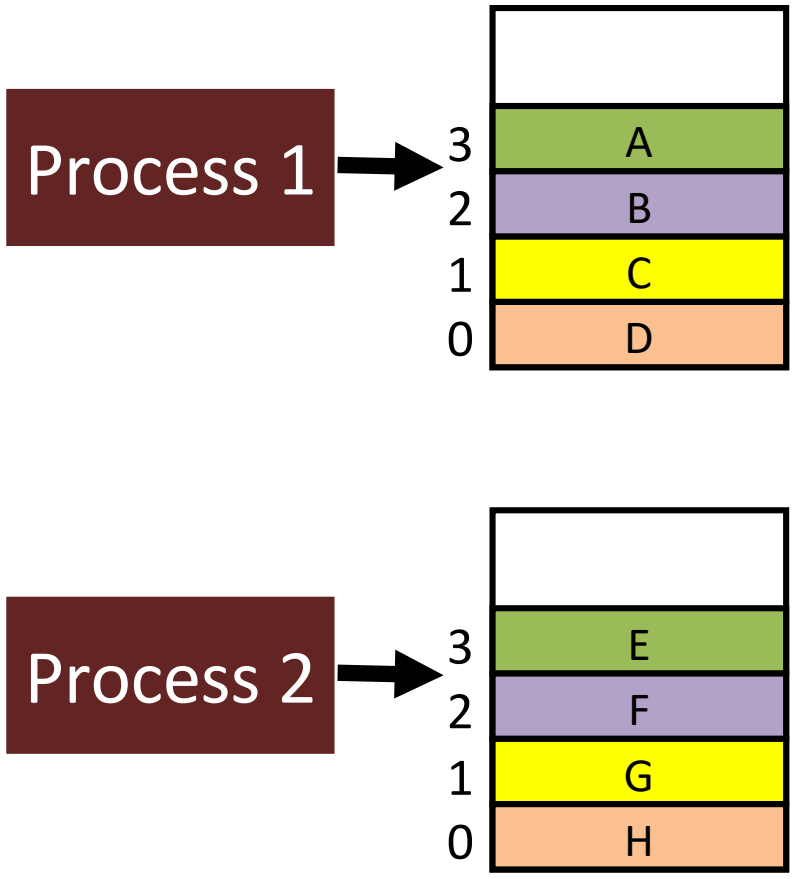
- Split 50/50?
- What if they don't fit?
- What if not contiguous?
- Need to recompile/relink?
- …

This is a problem even on a single core machine (runs multiple processes at a time)

(1990's programming & crashes)

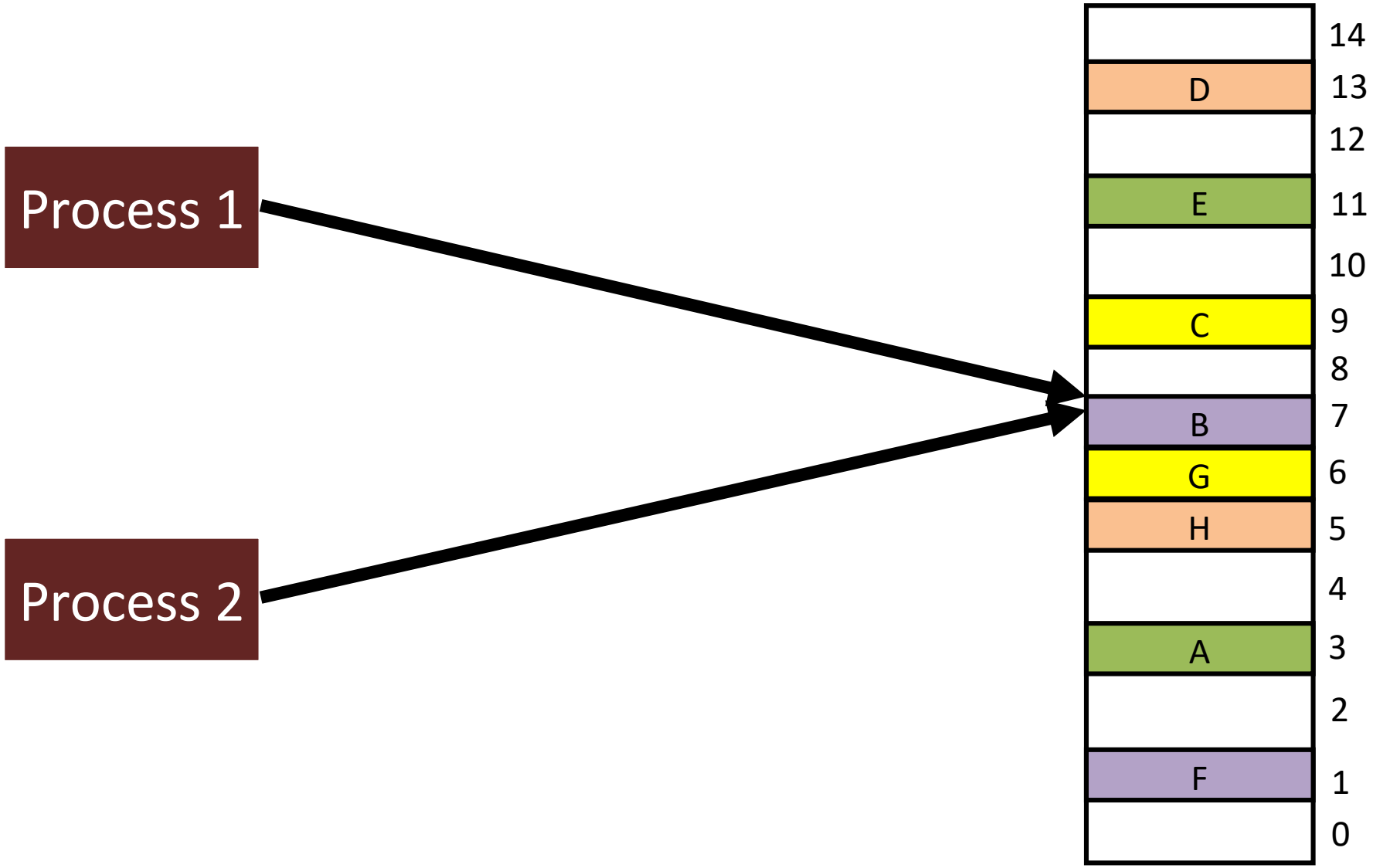


Big Picture: (Virtual) Memory

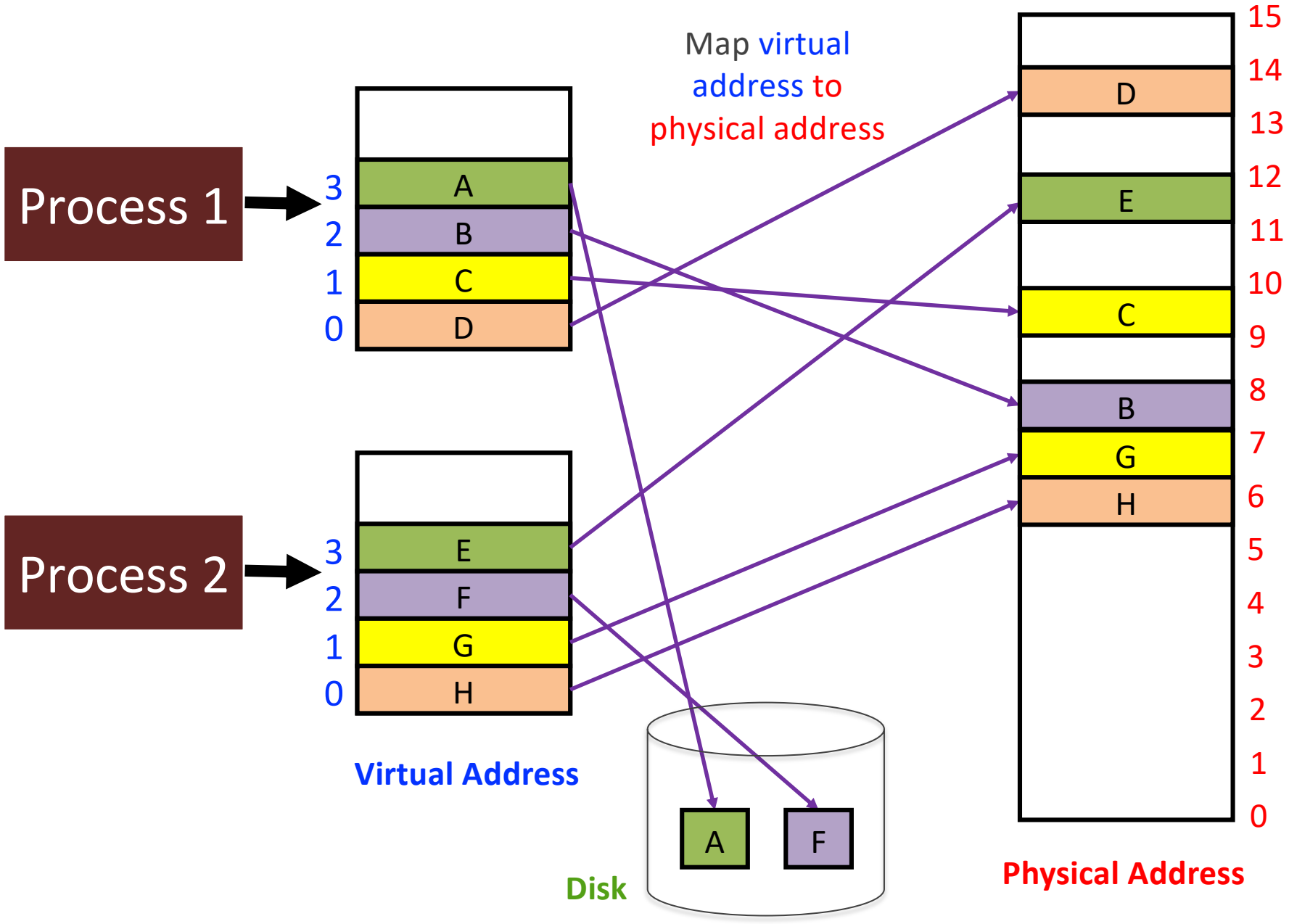


Give each process an illusion that it has exclusive access to entire main memory

But In Reality…

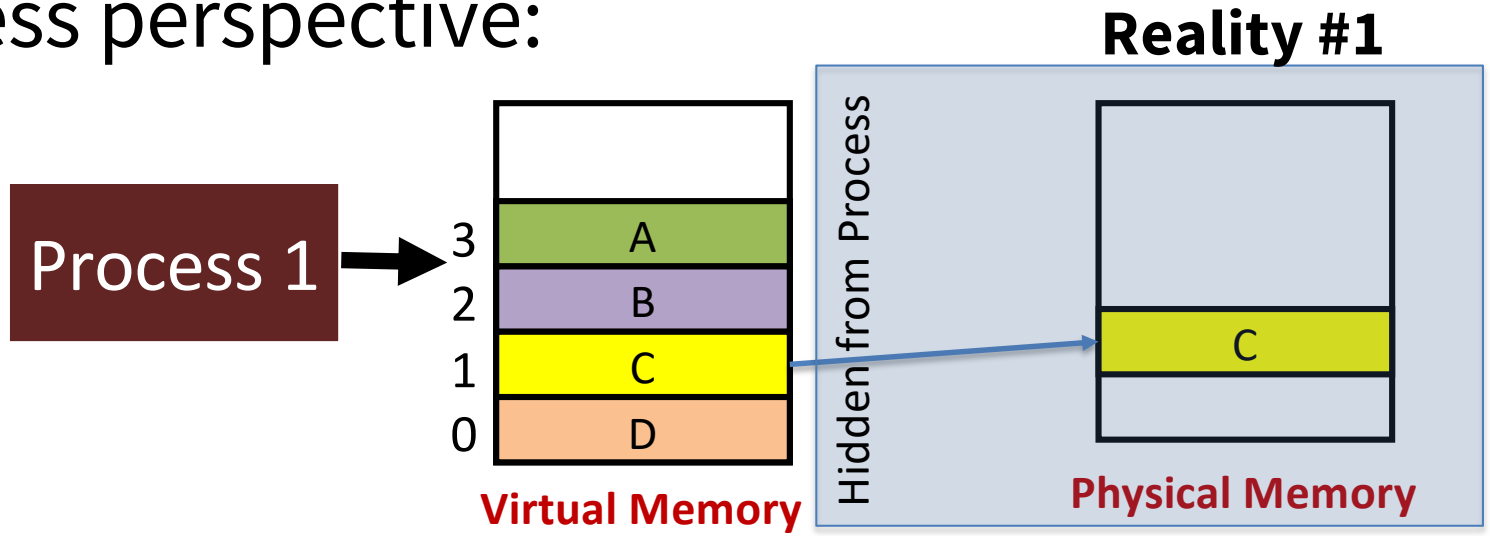


How do we create the illusion?



Big Picture: (Virtual) Memory

Process perspective:

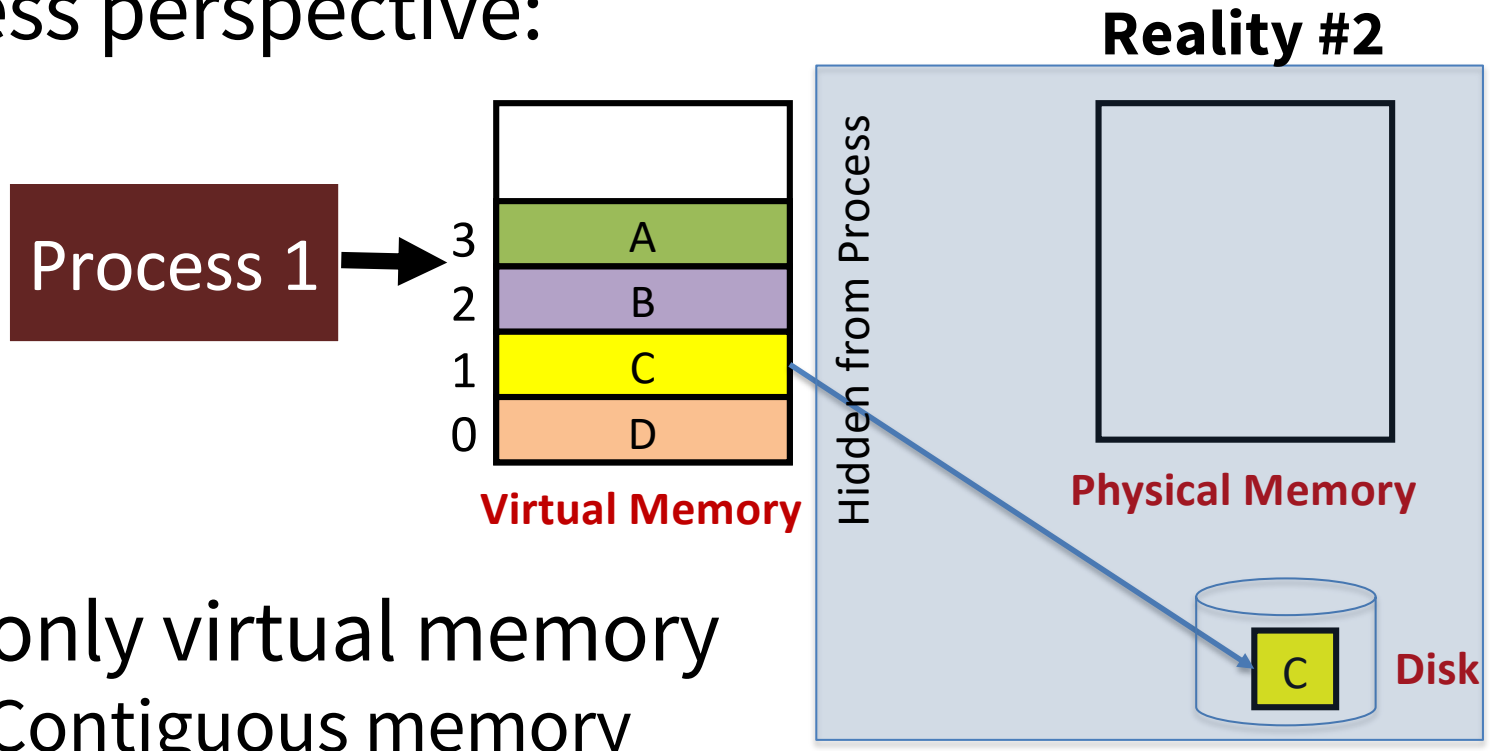


sees only virtual memory

- ✓ Contiguous memory
- ✓ No need to recompile - only mappings need to be updated

Big Picture: (Virtual) Memory

Process perspective:



sees only virtual memory

- ✓ Contiguous memory
- ✓ No need to recompile - only mappings need to be updated
- ✓ When run out of memory, map data on disk in a transparent manner

(1990's "would you like to enable virtual memory?)

Virtual Memory: a Solution for All Problems

Each process has its own virtual address space

- Program/CPU can access any address from $0 \ldots 2^N$
- 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

Provides Protection and Virtualization

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 processes / cores

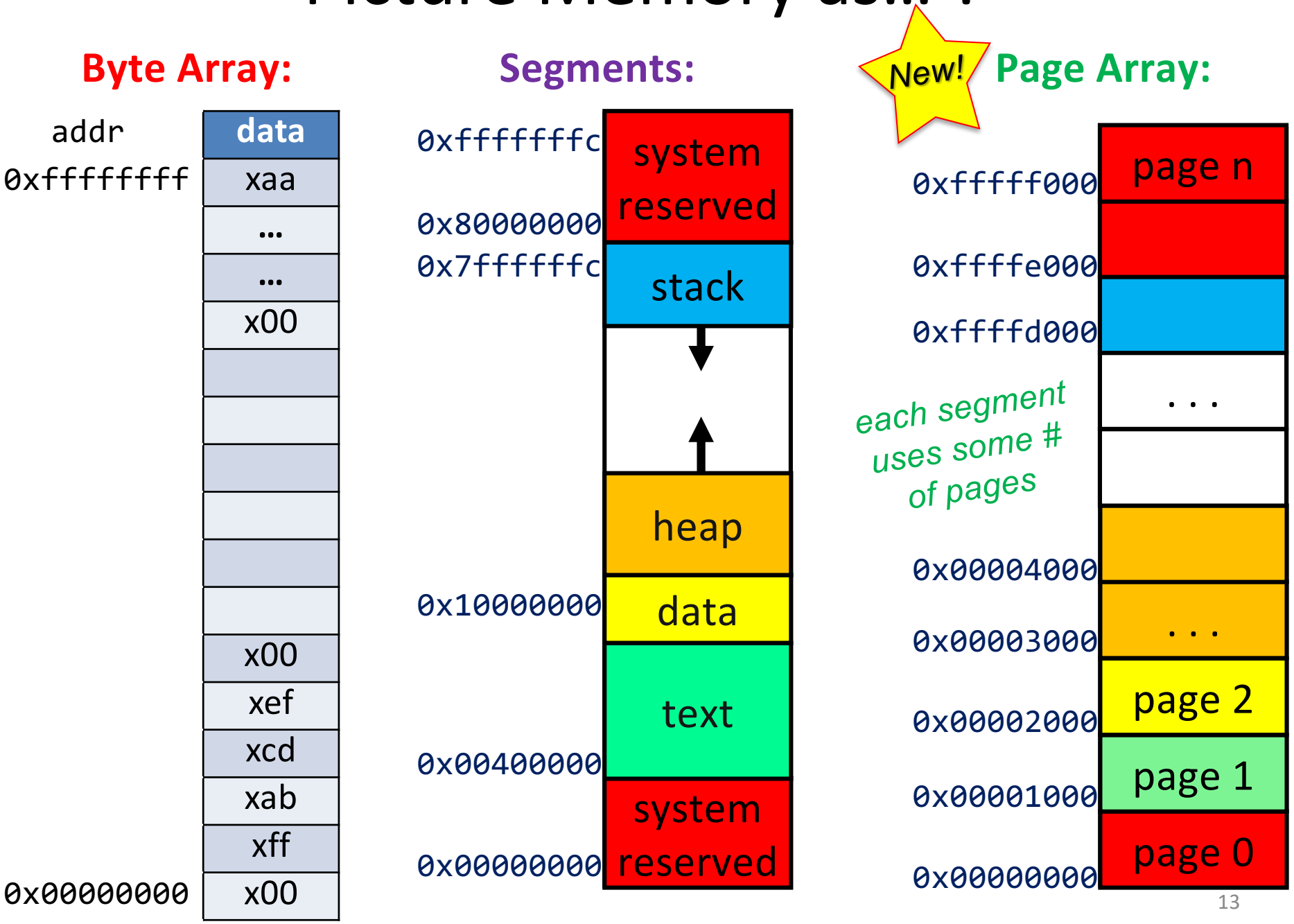
Virtual Memory Agenda

What is Virtual Memory?

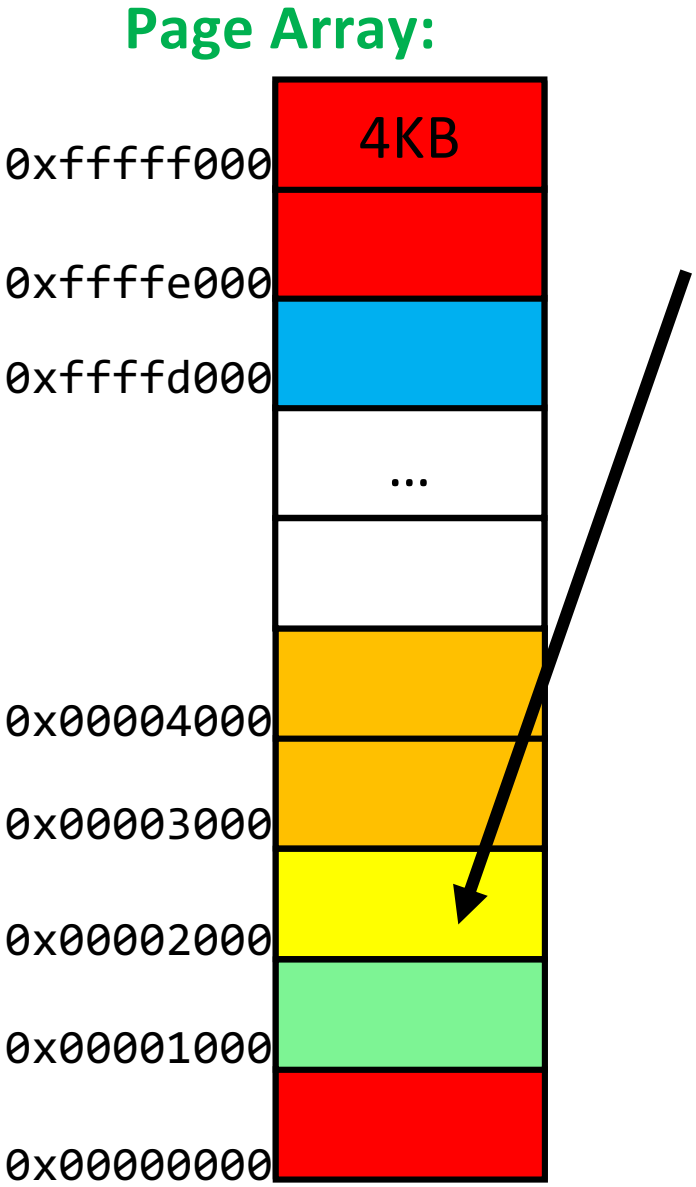
How does Virtual memory Work?

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

Picture Memory as… ?



A Little More About Pages



Suppose each page = 4KB

Anything in page 2 has address:
0x00002xxx

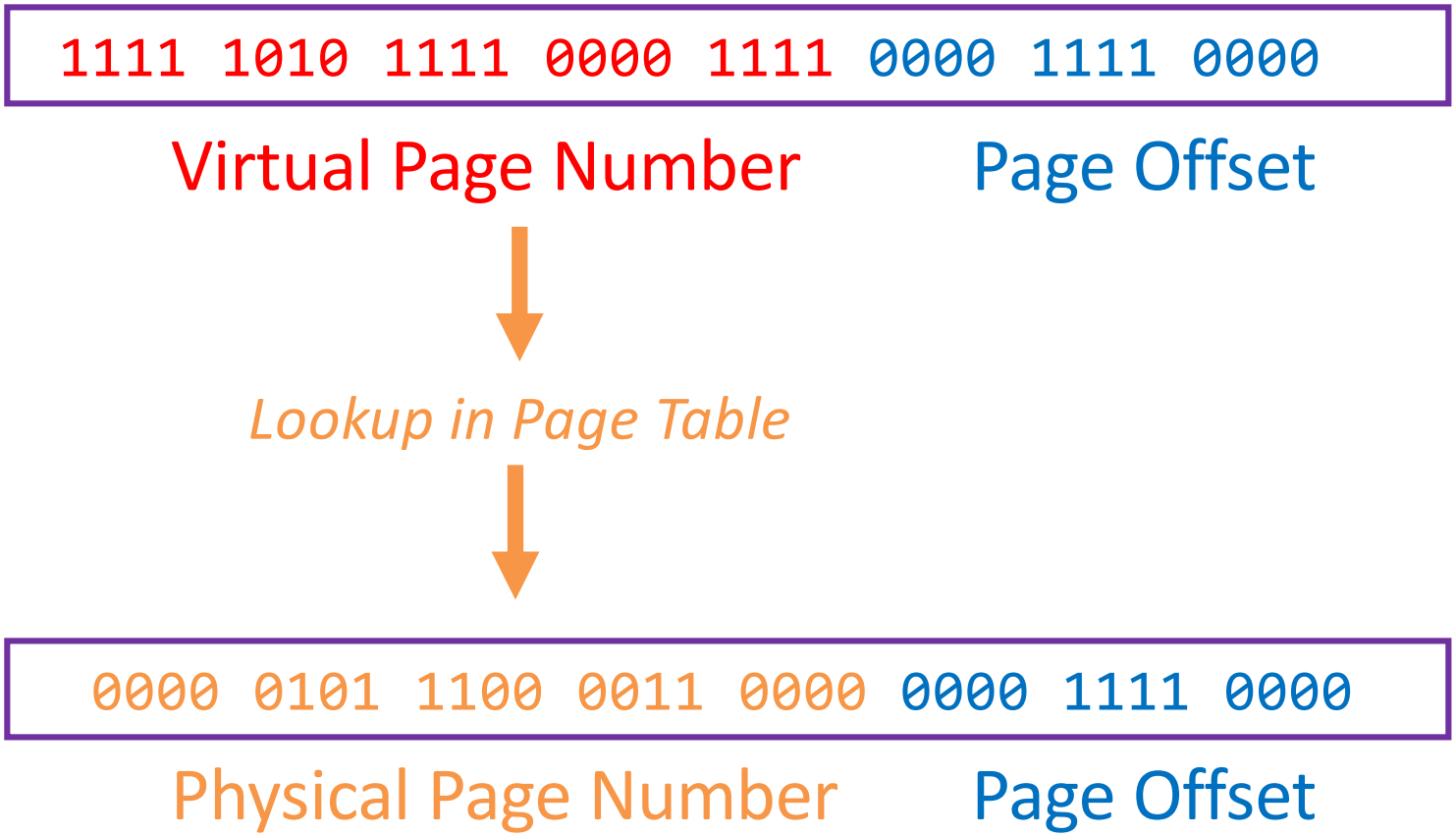
Lower 12 bits specify which byte you are in the page:

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

upper bits = page number
lower bits = page offset

(should sound familiar)

Simple Address Translation



Assuming each page = 4KB

Translations stored in the Page Table

OS-Managed Mapping of Virtual → Physical Pages

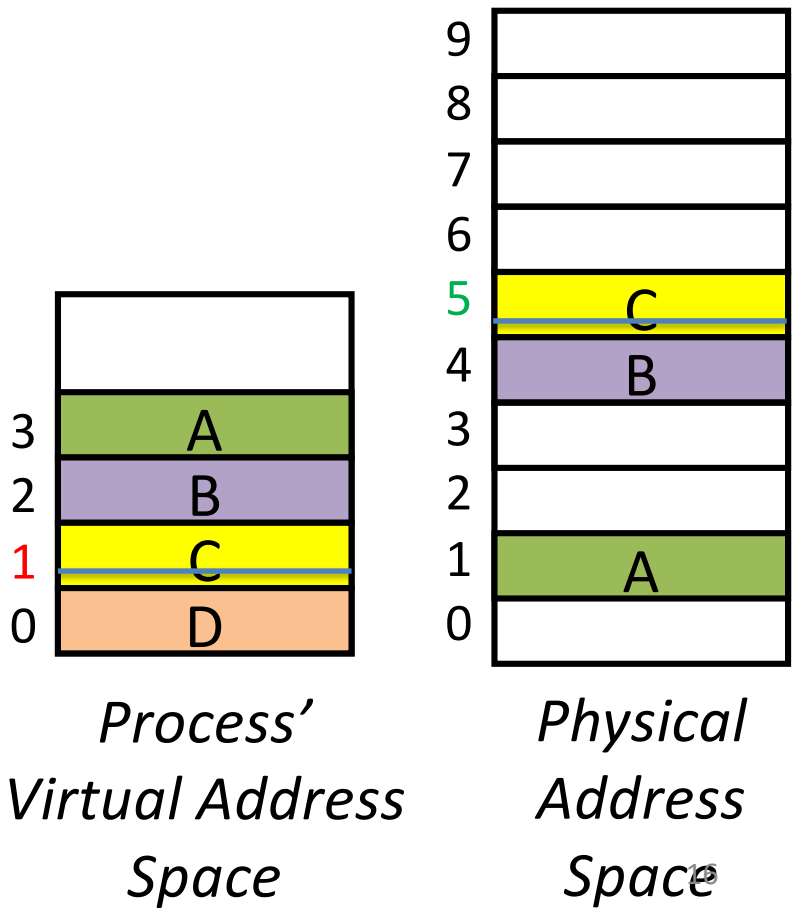
```
int page_table[2^20] = { 0, 5, 4, 1, … };
. . .
ppn = page_table[vpn];
```

Offset does not change:

VA 0x00001234

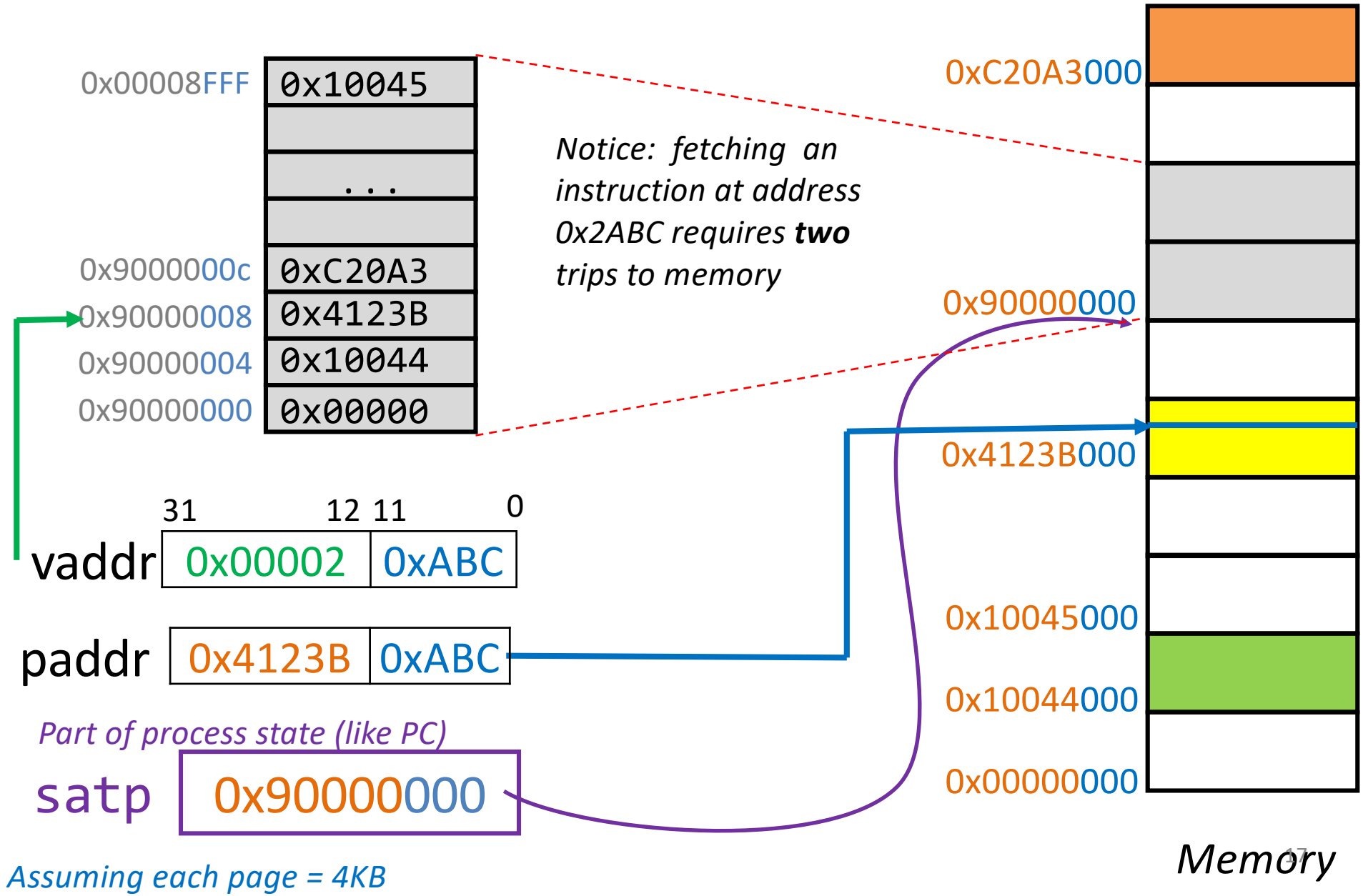
PA 0x00005234

both @ x234 in page C



Assuming each page = 4KB, lower 12 bits → offset

Simple Page Table Translation

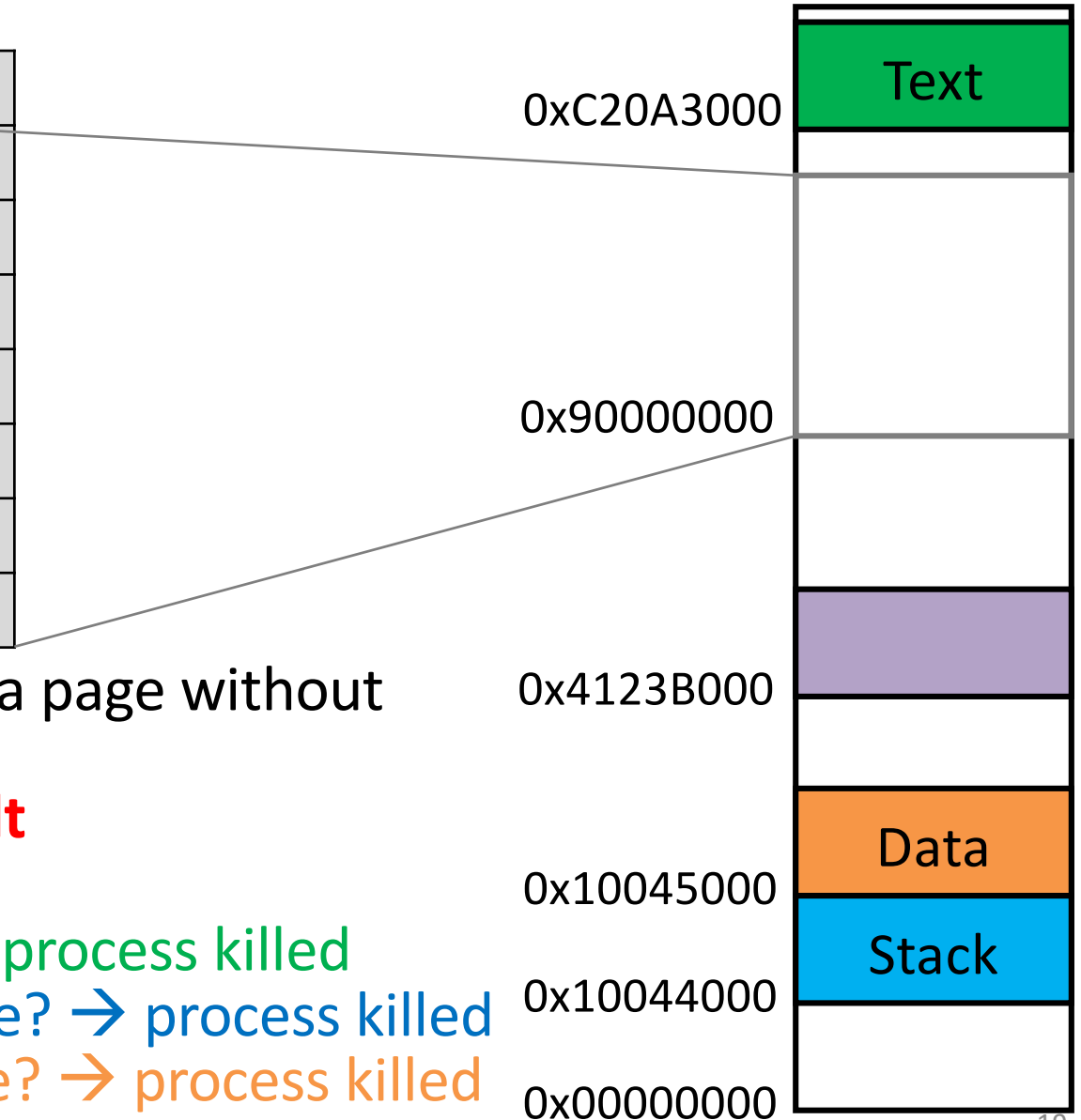


But Wait... There's more!

- 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

V	R	W	X	D	PPN
0					
1	1	0	1	0	0xC20A3
0					
0	1	1	0	1	0x10045
0					
1	1	1	1	1	0x4123B
1	1	1	0	1	0x10044
0					



Process tries to access a page without proper permissions

Segmentation Fault

Examples:

Write to read-only? → process killed

Execute non-executable? → process killed

Read non-existant page? → process killed

Page Fault

Valid bit in Page Table = 0 (page is not in memory)

Why? Maybe the page…

- wasn't needed yet (still part of the Text section)
- didn't exist before (growing Stack or Heap)
- was sent to disk (OS swapped it out b/c it needed room)

OS takes over, solves the problem, updates Page Table

Performance-wise page faults are *really* bad!

Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

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

Cost of Page Tables

- How large is a Page Table?
- Virtual address space (for each process):

  Given: total virtual memory: 2^{32} bytes = 4GB

  Given: page size: 2^{12} bytes = 4KB

  - **# PT entries?** 2^{20} = 1 million entries
  - **Size of each entry?** PTE size = 4B
  - **size of Page Table?** 4B x 2^{20} = 4MB

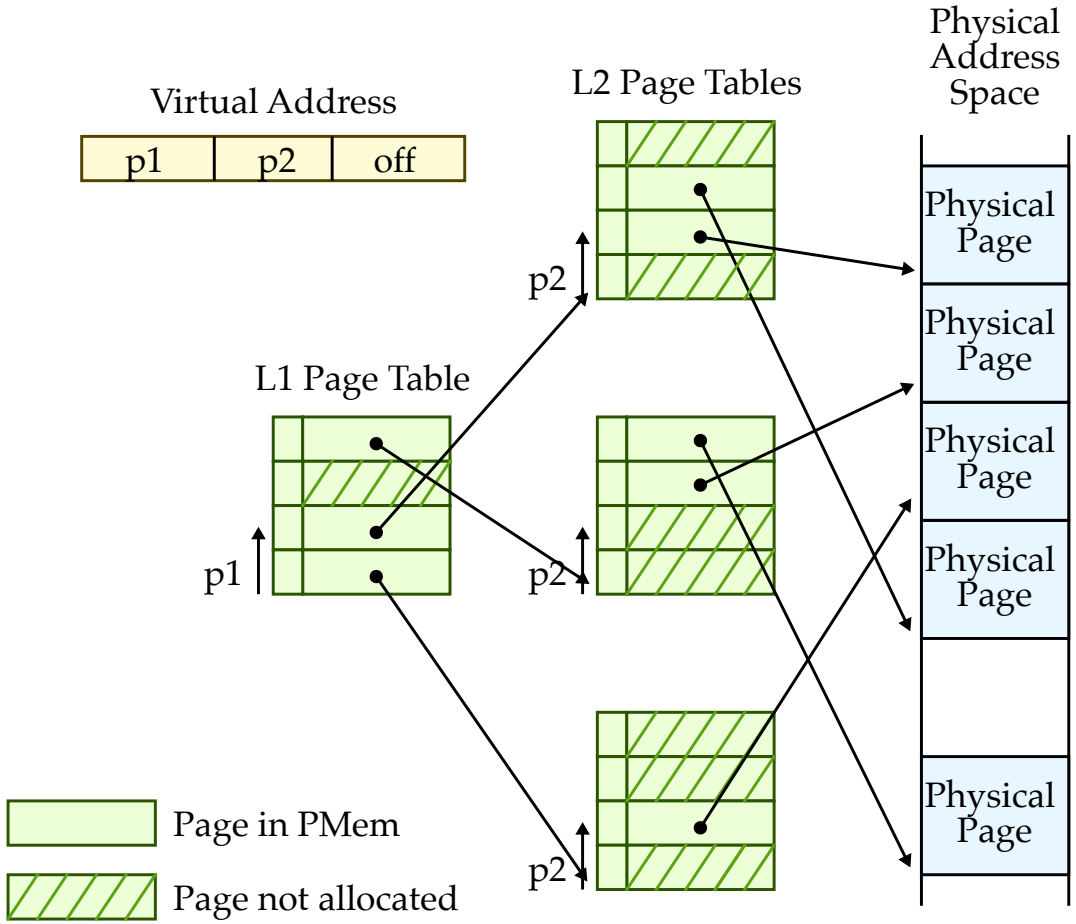
  What if we have 10 processes?

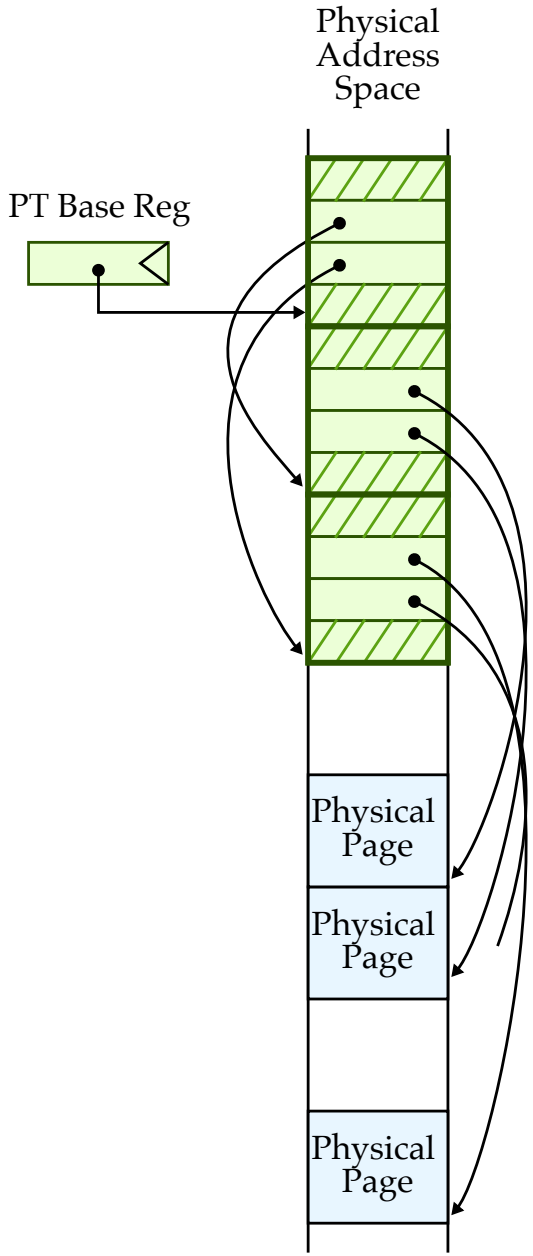
  10 x 4MB = 40 MB of Page Tables…. ☹

helloworld.c has a 4MB page table??

Multi-Level Page Tables to the Rescue!

+ Allocate only PTEs in use
+ Simple memory allocation
− ***more*** lookups per memory reference





Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

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

Watch Your Performance Tank!

For *every instruction*:

1. Translate address
2. Fetch the instruction using physical address
   - Access Memory Hierarchy (I$ → L2 → Memory)

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

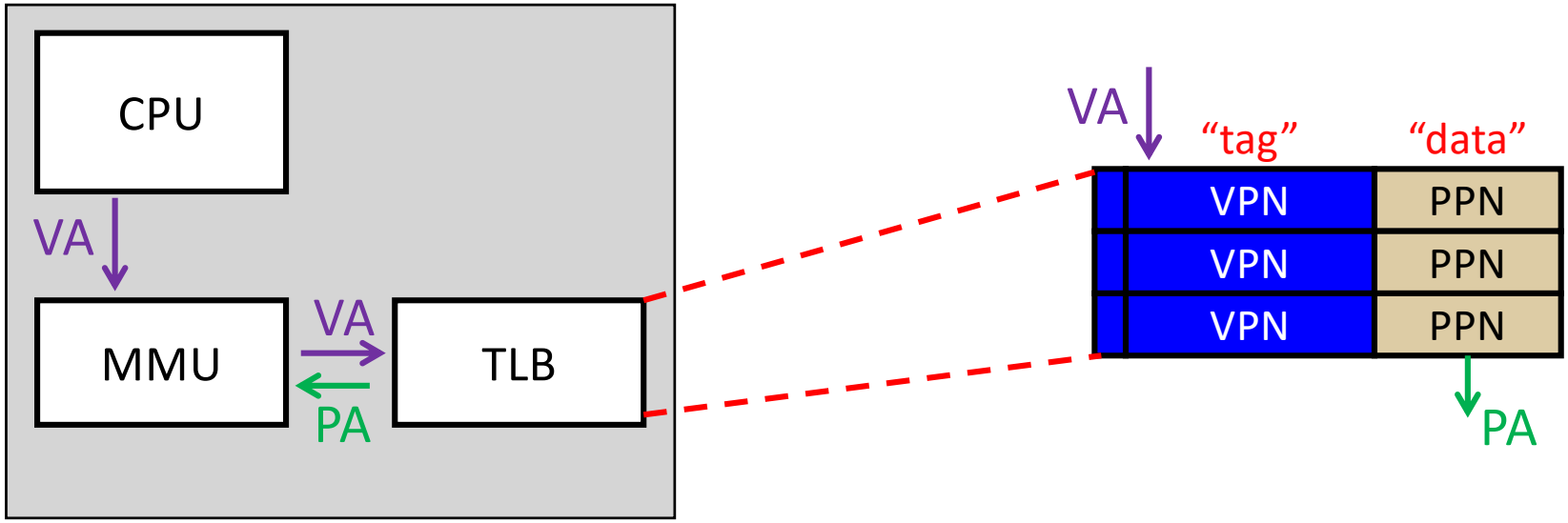
Best case? 1-2 memory accesses

- Go to memory to access Page Table
- Instructions & Data in Cache

Worst case? 4+ memory accesses

- everything in memory
- even worse for multi-level PT

Translation Lookaside Buffer (TLB)



- Small, fast cache
- Holds VPN→PPN translations
- Exploits temporal locality in pagetable
- TLB Hit: huge performance savings
- TLB Miss: puts translation in TLB
  - Handled in software (exception: OS walks Page Table)
  - Handled in hardware (MMU walks Page Table)

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

Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

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

Translation in Action