Course website temporary issues + updated slides #875

Giulia Guidi STAFF
4 minutes ago in General

UNPIN STAR WATCHING 3 VIEWS

Hi,

The course website is not updating and deploying correctly. The staff is looking into it but in the meantime please find updated slides for lectures 19-21 in this post.

Best,

Prof. Guidi

19-guidi-syscalls.pdf

20-guidi-syscalls-vm.pdf

21-guidi-vm.pdf

Cornell Bowers C·IS
Computer Science

CS3410: Computer Systems and Organization

LEC21: Virtual Memory

Professor Giulia Guidi
Monday, November 10, 2025

Credits: Bala, Bracy, Garcia, Guidi, Kao, Sampson, Sirer, Weatherspoon

Plan for Today

- Review of system call
- On to virtual memory!

Review of system call

Ex: `fork()`

- `fork()` is used to create a new process by duplicating the calling process
  - The new process is called the **child** process
  - The original process is called the **parent** process

- `fork()` function prototype:

```
pid_t fork(pid_t pid);
```

- `fork()` is called, then both processes continue executing the code **after the `fork()` call**, but they have **different PIDs**

Ex: `exec()`

- `exec()` replaces the current process image with a new process image
  - Commonly used functions: `execl()`, `execp()`, `execv()`, etc.

- `exec()` function prototype:

```
int execl(const char *path, const char *arg, ...);
```

- `exec()` basically ***changes*** what a process does

Ex: `waitpid()`

- `waitpid()` is used to wait for state changes in a child process
  - It can be used to wait for a specific child process to terminate
- `waitpid()` function prototype:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Return value	**Return value of `waitpid()`**
`> 0`	`PID` of the child whose state has changed
`0`	(only if `WNOHANG` used) — no child has exited yet
`-1`	-1

`waitpid()` vs `wait()`

- `waitpid()` is used to wait for state changes in a child process
  - It can be used to wait for a specific child process to terminate

- `waitpid()` function prototype:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Call	It means…
`wait(&status)`	It waits for **any** child to terminate
`waitpid(-1, &status, 0)`	Same as `wait()`
`waitpid(pid, &status, 0)`	It waits for **that specific `pid` child**

How Processes Are Terminated?

- The system calls for termination are:
  - **`exit()`**: used by a process to terminate itself
  - **`abort()`**: used by a parent process to terminate a child process
  - **`wait()`** and **`waitpid()`**: used by a parent process to wait for the termination of a child process and retrieve its exit status

False Friend `kill()`

- Despite its name, `kill()` doesn't necessarily *kill* a process; it can send **any** signal, including ones that don't terminate the target
- `kill()` function prototype:

```
int kill(pid_t pid, int sig);
```

Outcome	**Return value of `kill()`**
Success	`0`
Error	`-1`

Conceptual RISC-V Print “Hello”

Conventionally, a7 holds the **system call number**: it tells the OS **which** service the program is asking for

```
# RISC assembly pseudo-code
li    a7, 4            # Load system call code for 'print string'
la    a0, msg          # Load address of message
ecall                  # Call to the OS
...
msg:  .asciiz "Hello!"
```

PC moves from **user** space to **kernel** space (more on this later)

This is a **special instruction** that triggers a **trap** to the operating system (OS)

- The OS looks at `a7` (system call number = 4) and `a0` (address of string)
- It performs the requested service: writing “`Hello!`” to the terminal

Interrupt

An **interrupt** is a signal sent to the processor by hardware or software indicating an event that needs immediate attention

Poll

What is the key difference between a hardware interrupt and a software interrupt?

[PollEv.com /gguidi](https://PollEv.com/gguidi)

Or send **gguidi** to **22333**

Types of Interrupt

- **Hardware Interrupt:** Generated by external hardware devices to get the CPU's attention
    - Keyboard input (key press)
    - Mouse movement or click
    - Disk I/O completion
    - Network packet arrival

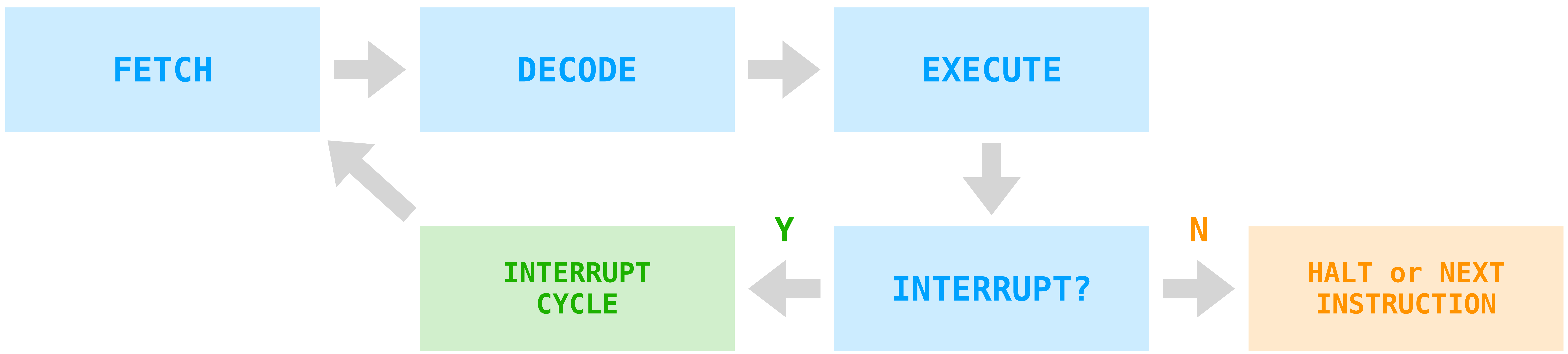
Type	Description
Maskable Interrupt	Can be temporarily "masked" (disabled) by the CPU; used for regular hardware events
Non-Maskable Interrupt	Cannot be disabled; reserved for critical events like hardware failure or power issues

Types of Interrupt

- **Software Interrupt**: Generated by a program or the operating system, typically to request a system service or perform an exception
    - System calls (e.g., syscall instruction)
    - Exceptions like divide-by-zero or invalid memory access
    - Breakpoints during debugging

Type	Description
System Call Interrupt	Triggered by a program to request OS services (e.g., file read/write)
Exception Interrupt	Generated automatically when an error or specific condition occurs (divide by zero, page fault)
Software-Generated Signal	Explicitly triggered in software (e.g., raise(), kill() in Unix)

Instruction Cycle and Interrupt



How Interrupts Work

- **Interrupt Signal**
  - An interrupt signal is sent to the CPU by a hardware device or software
- **Saving State**
  - The CPU saves the current state of the running process (e.g., program counter, registers) so it can resume execution later
- **Interrupt Handling**
  - The CPU transfers control to the interrupt handler associated with the interrupt. The interrupt handler processes the event (e.g., reading data from a device)
- **Restoring State**
  - The CPU restores the saved state and resumes execution of the interrupted process

Ok, introduction to virtual memory

Process = running program

Syscall = how it asks the kernel to do privileged work on its behalf

Big Picture: Processes

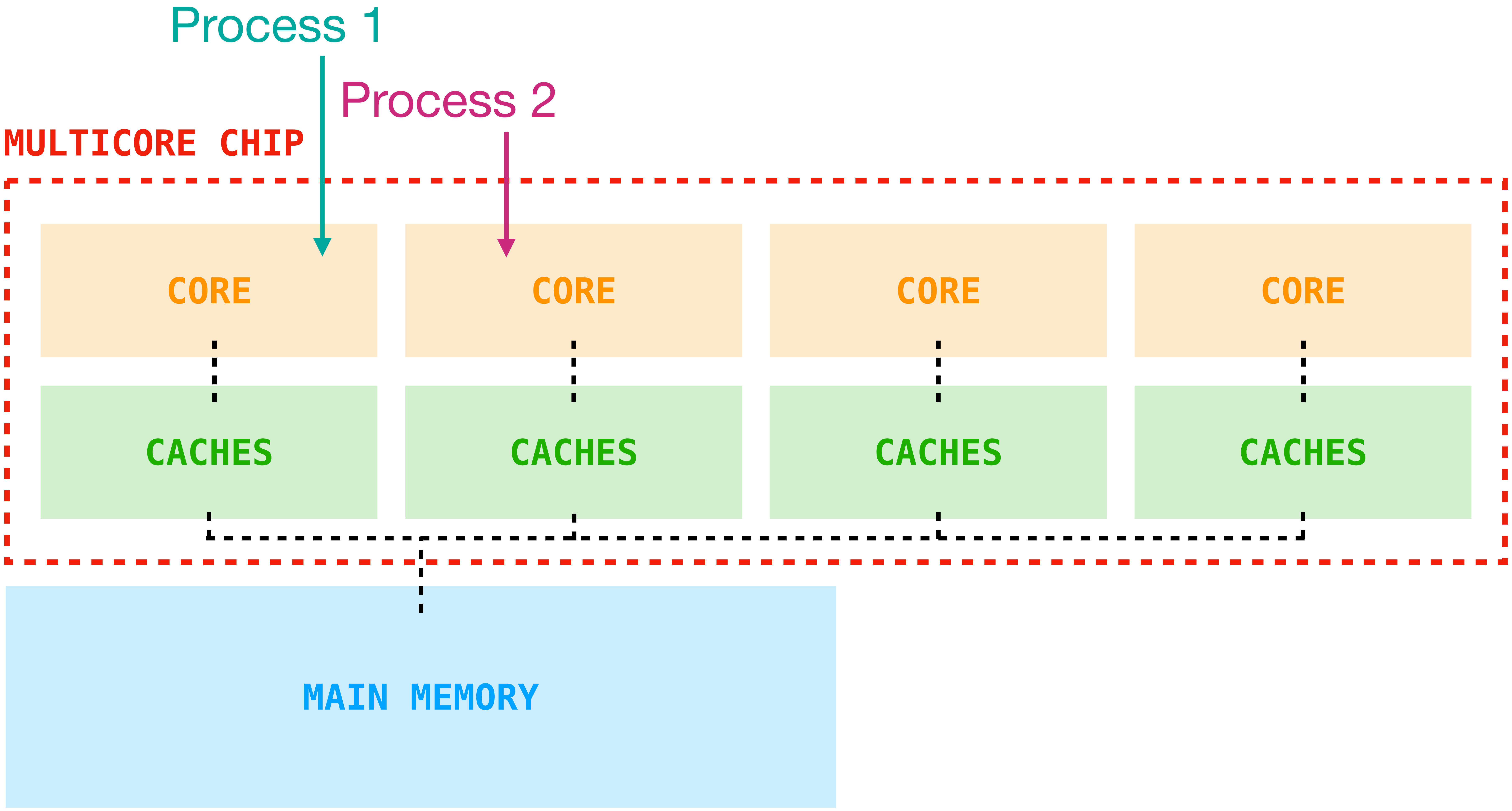
Each process requires memory to hold:

- Its **instructions** (the code to run)
- Its **data** (variables, heap)
- Its **stack** (function calls, local variables)

The **problem**:

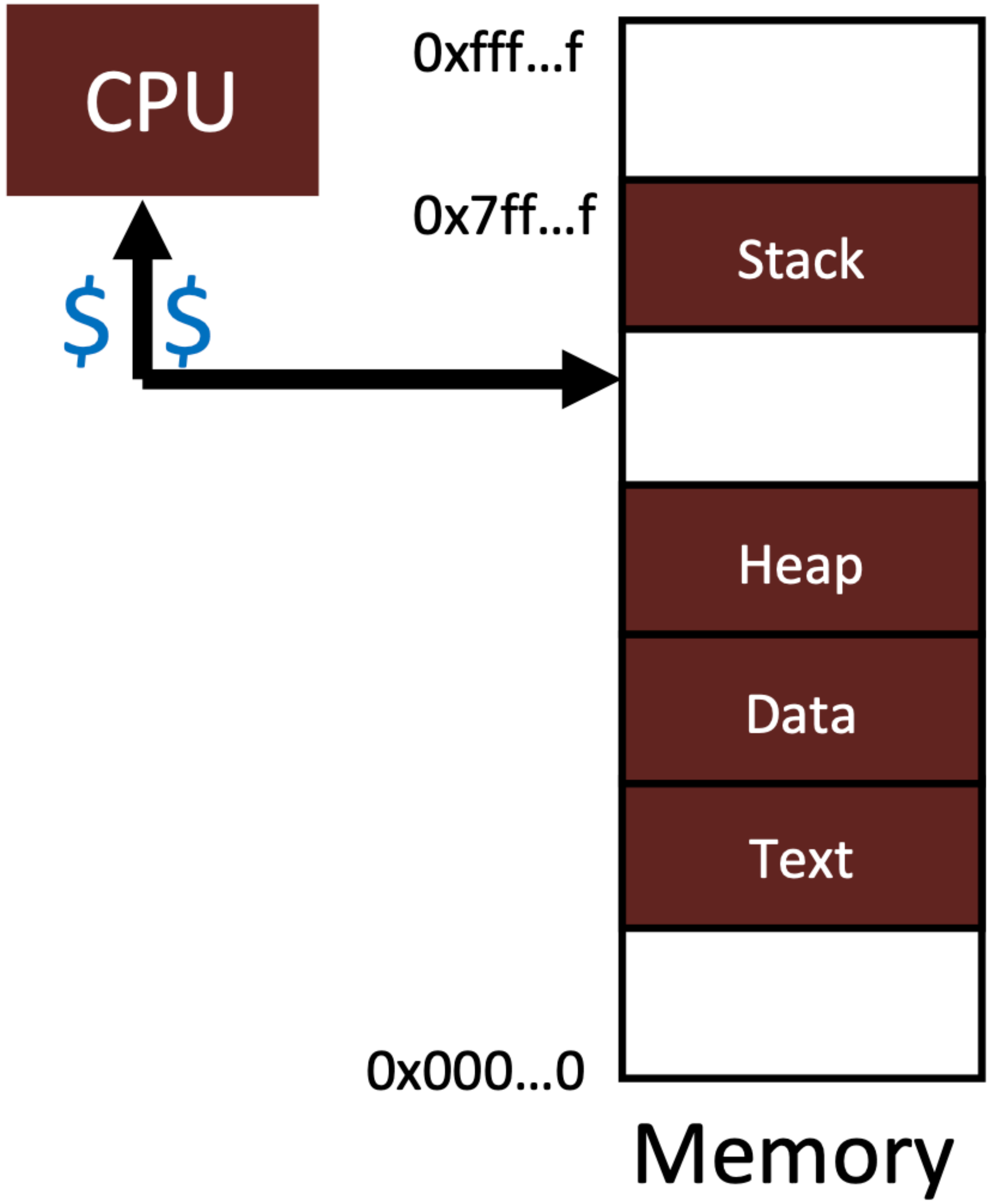
- In reality, multiple processes run **at the same time**
- They all think they're using the same memory addresses (like address `0x400000`)

Multi-Core Processor



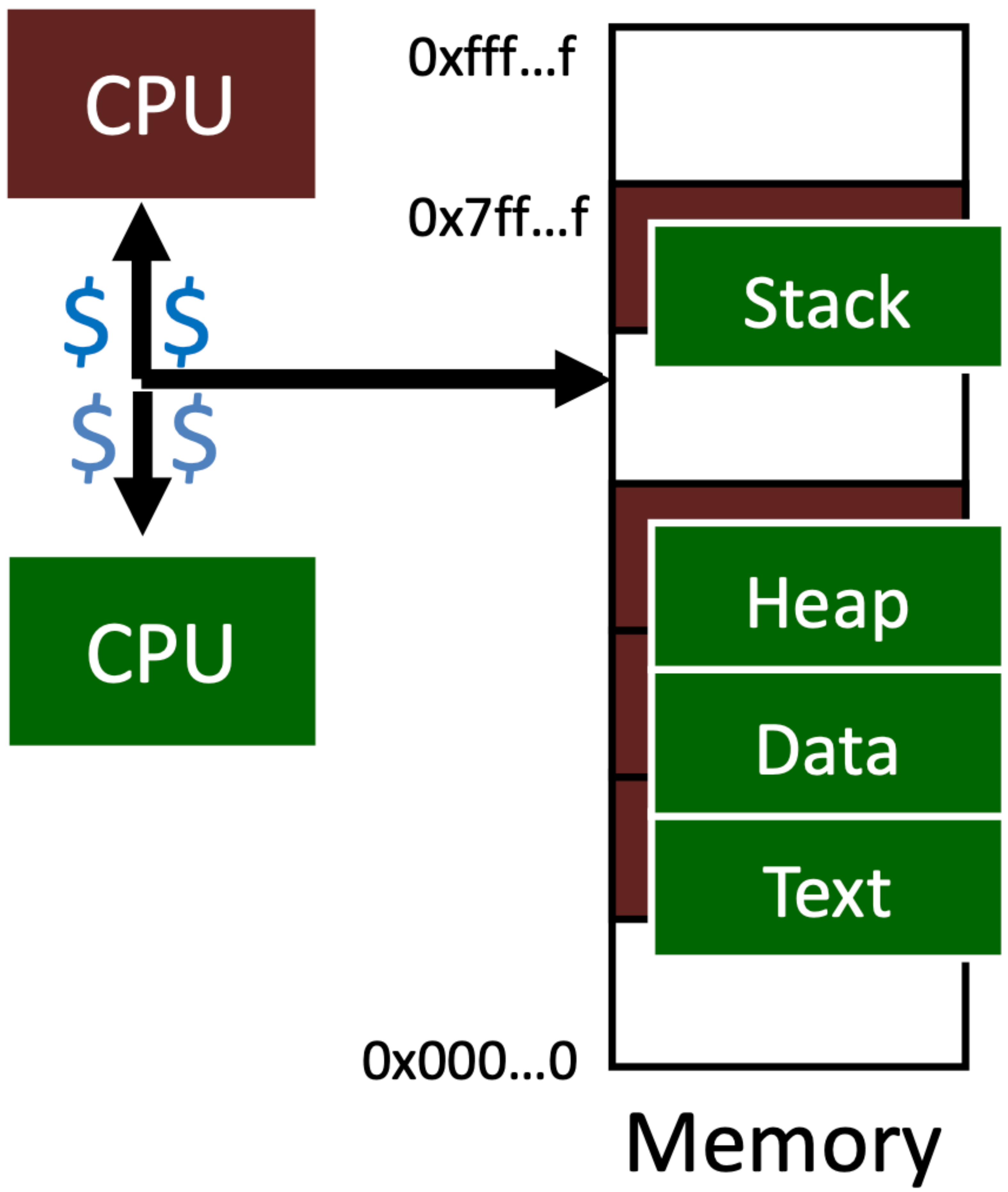
Processor & Memory

- CPU address/data bus...
  - … routed through caches
  - … to main memory
  - It's simple, fast, but…



Processor & Memory

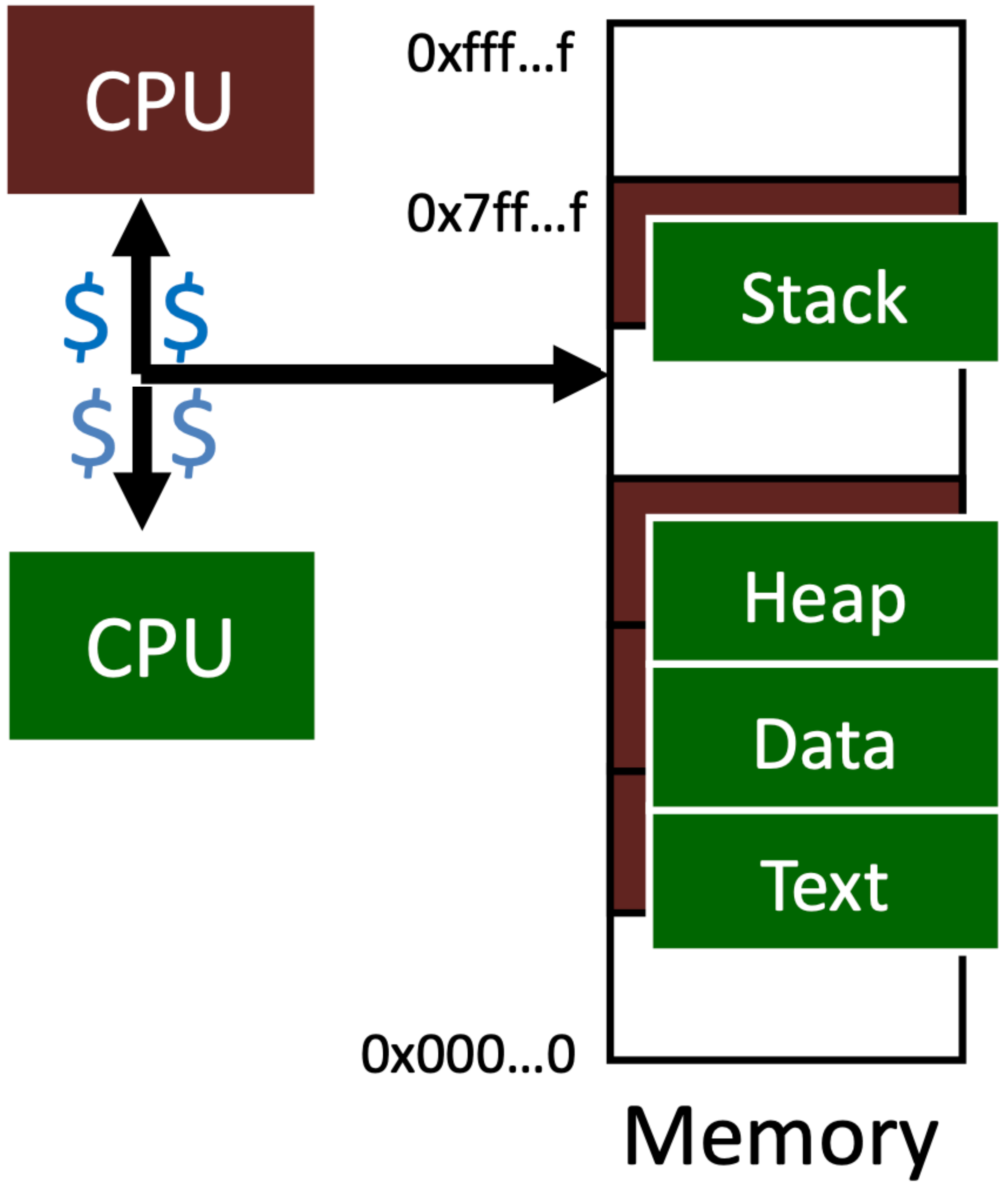
- So what happens when when another program is executed concurrently on another processor?
- The addresses will conflict
  - Even if CPUs take turns using memory bus



Processor & Memory

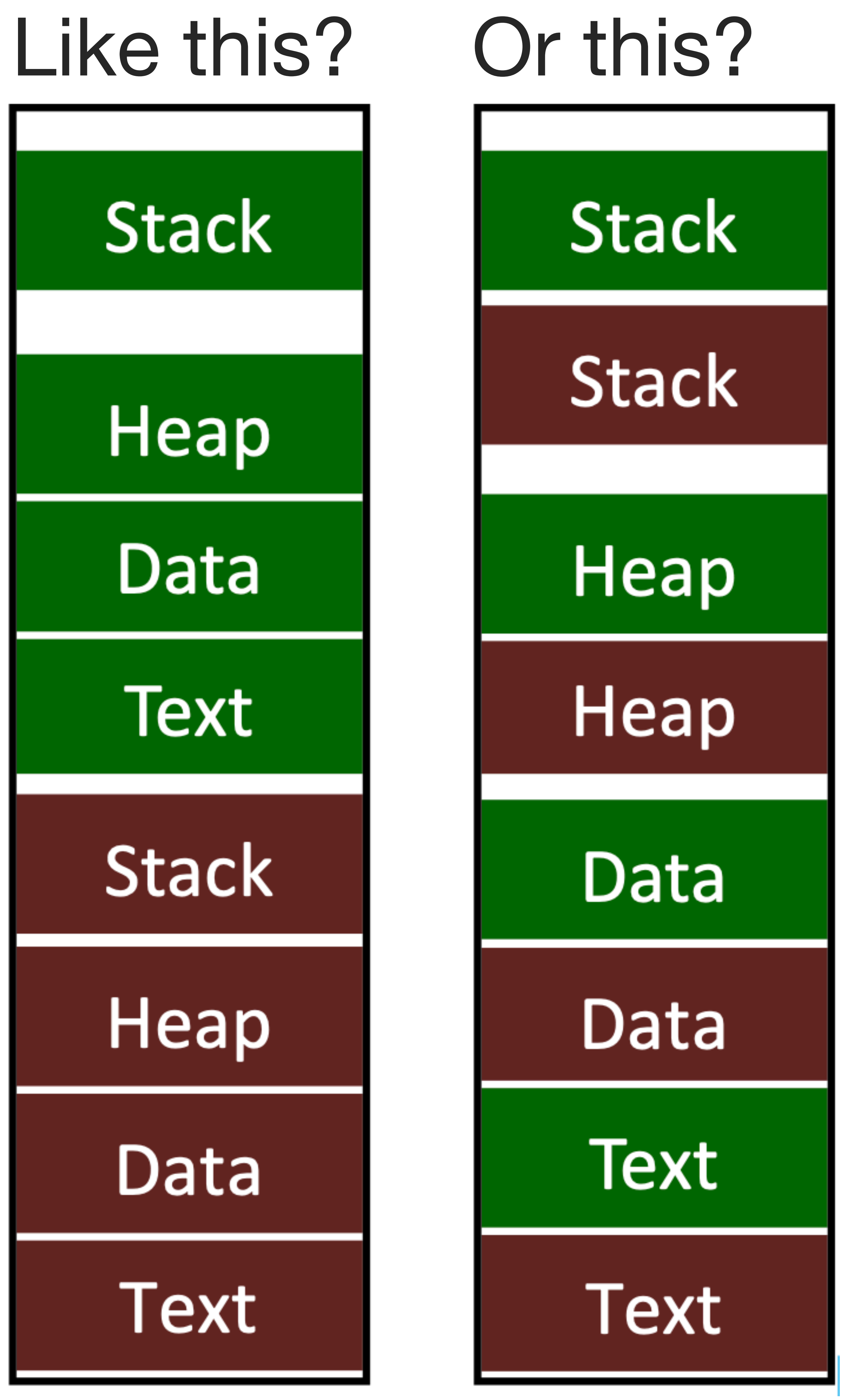
- So what happens when when another program is executed concurrently on another processor?
- The addresses will conflict
  - Even if CPUs take turns using memory bus

  - Solutions?
    - **Can we relocate second program?**



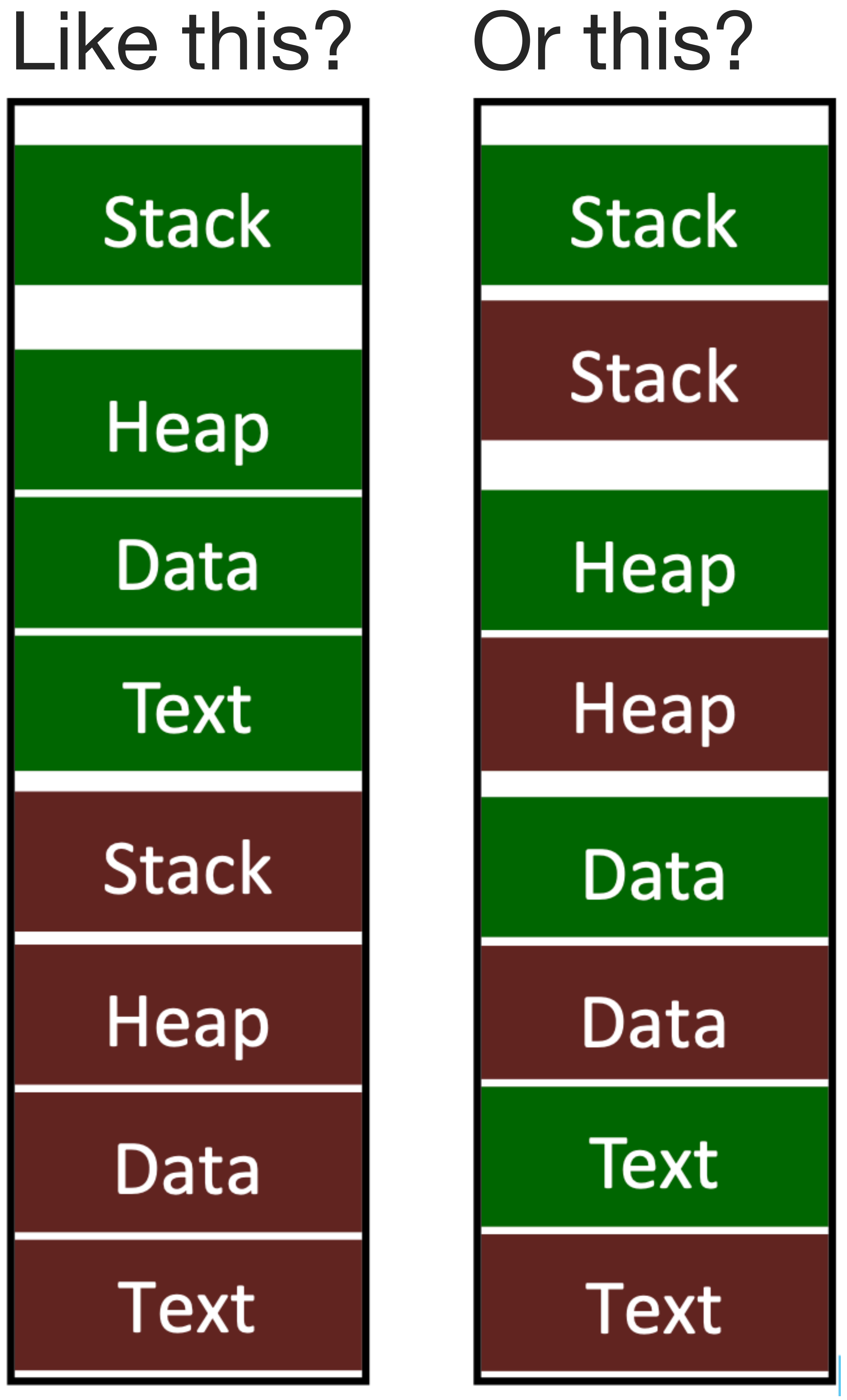
Can We Relocate Second Program?

- Yes but how?



Can We Relocate Second Program?

- Yes but how?
- Do we split 50/50?
  - If they don't fit?
  - If not contiguous?
  - Do I need to recompile?



Poll

True or False: The problem of two processes sharing DRAM only exists on a multicore machine

PollEv.com /gguidi

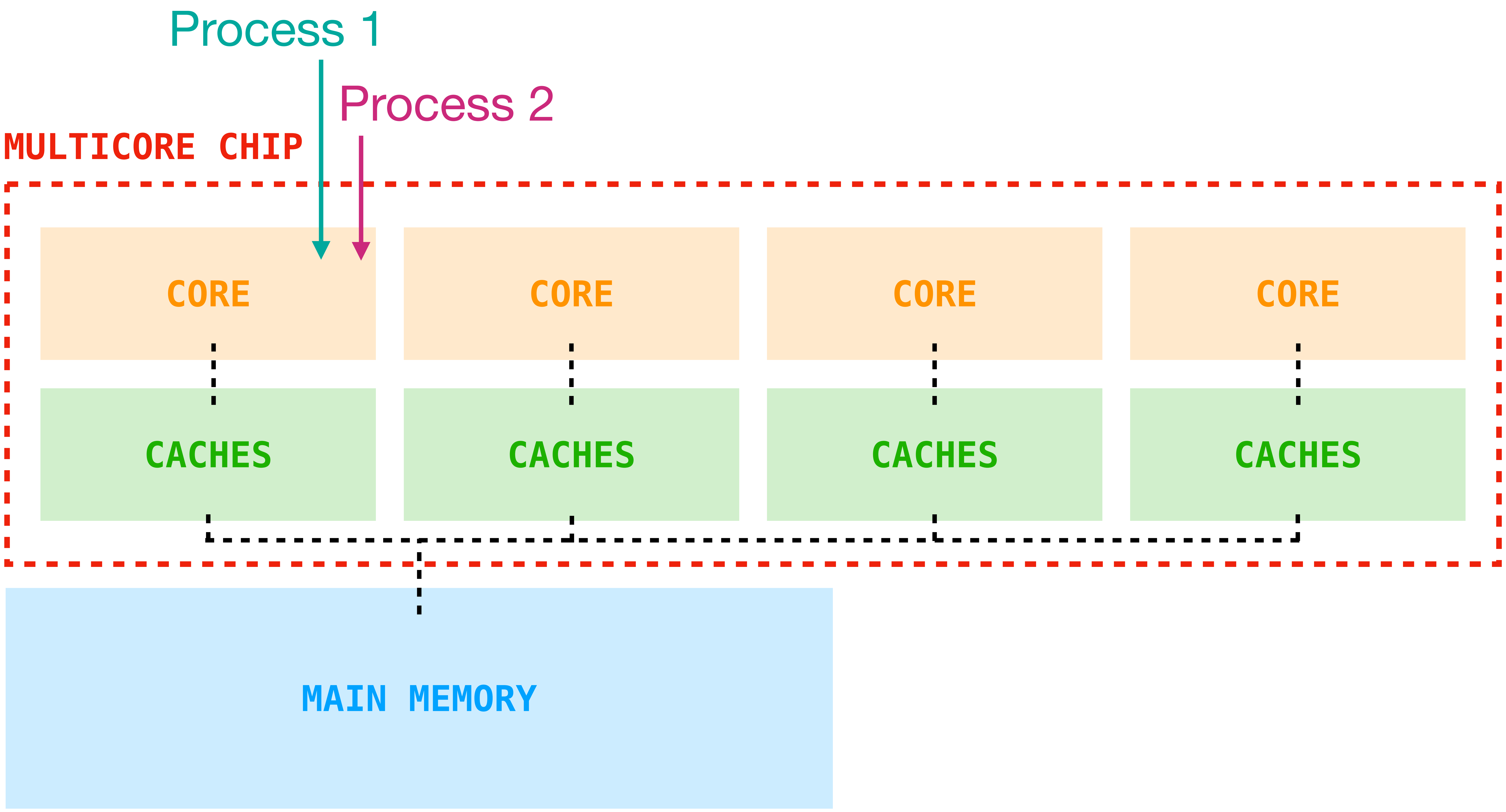
Or send **gguidi** to **22333**

Poll

True or False: The problem of two processes sharing DRAM only exists on a multicore machine

memory **contention** can occur even on a single-core machine due to time-sharing; multicore just increases simultaneous access

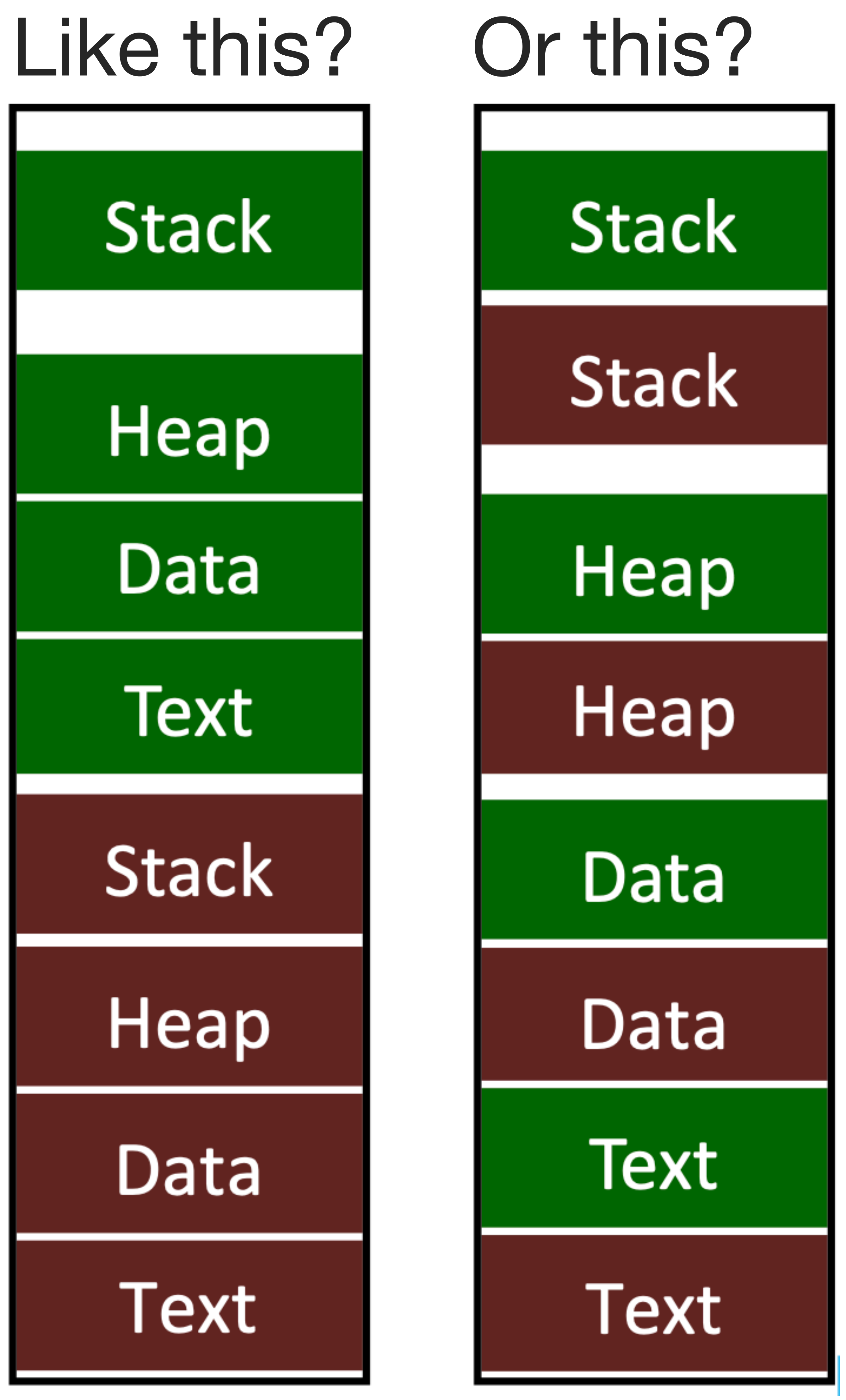
Multi-Core Processor



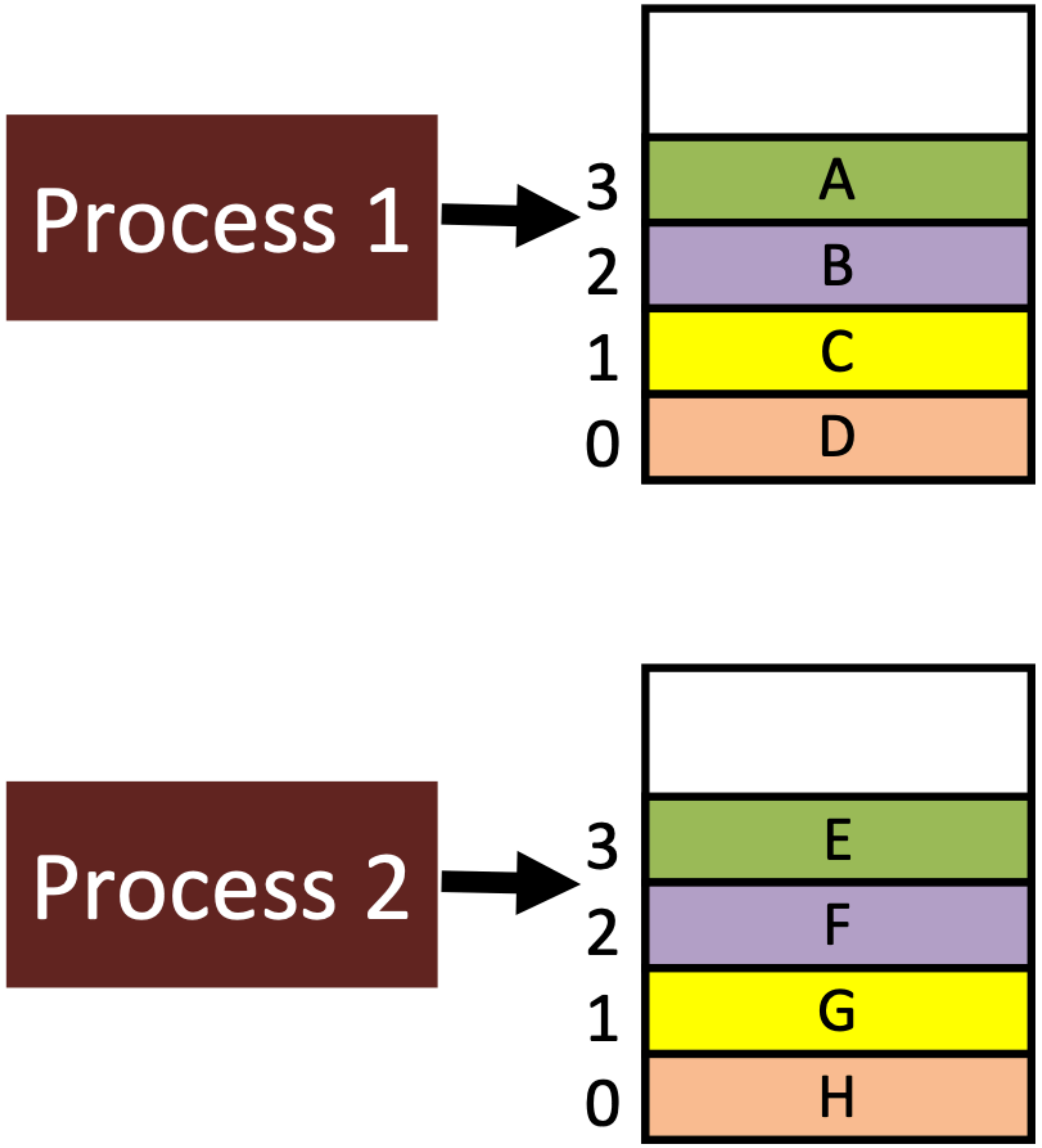
Can We Relocate Second Program?

- Yes but how?
- Do we split 50/50?
  - If they don't fit?
  - If not contiguous?
  - Do I need to recompile?

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

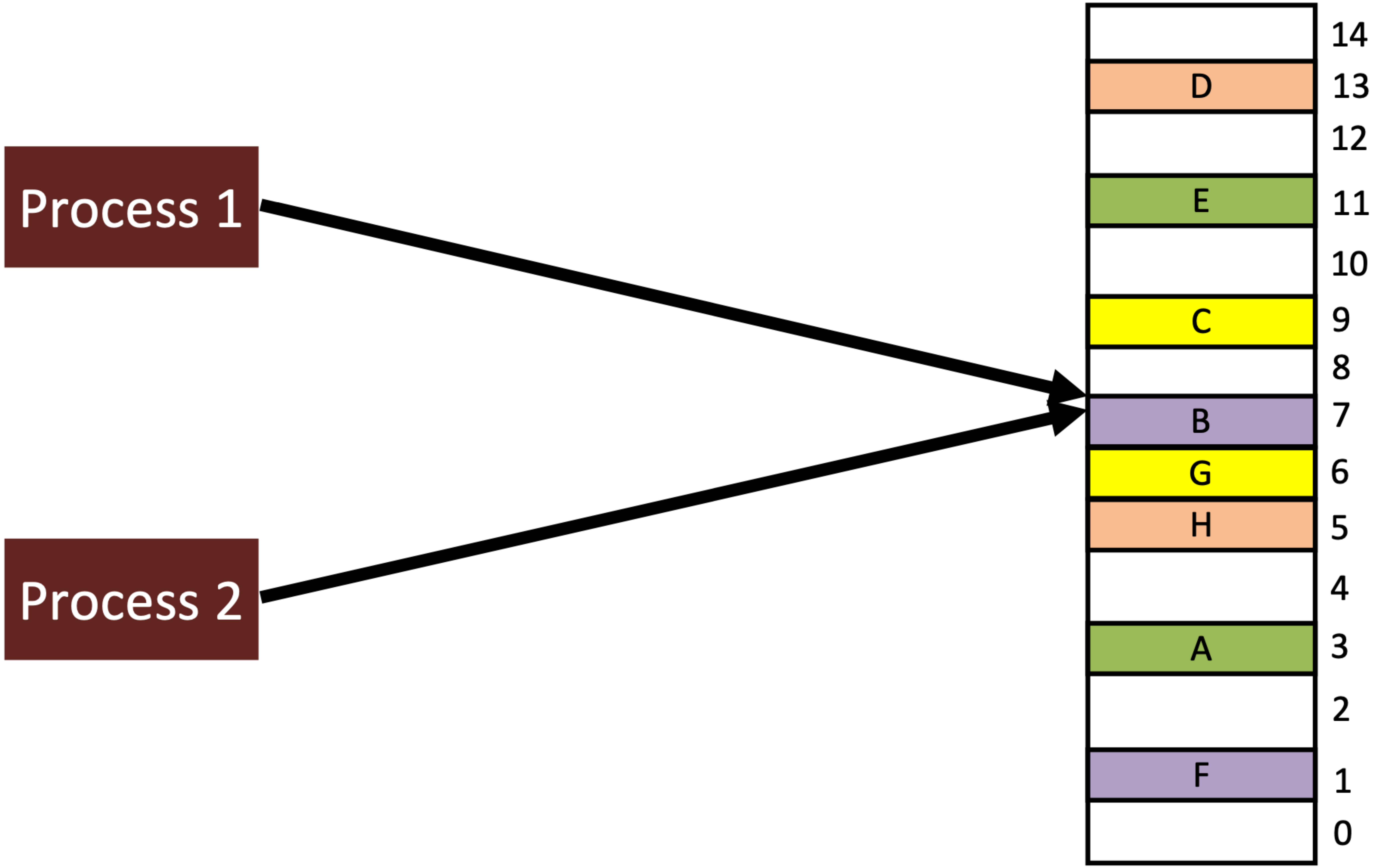


Big Picture: Virtual Memory

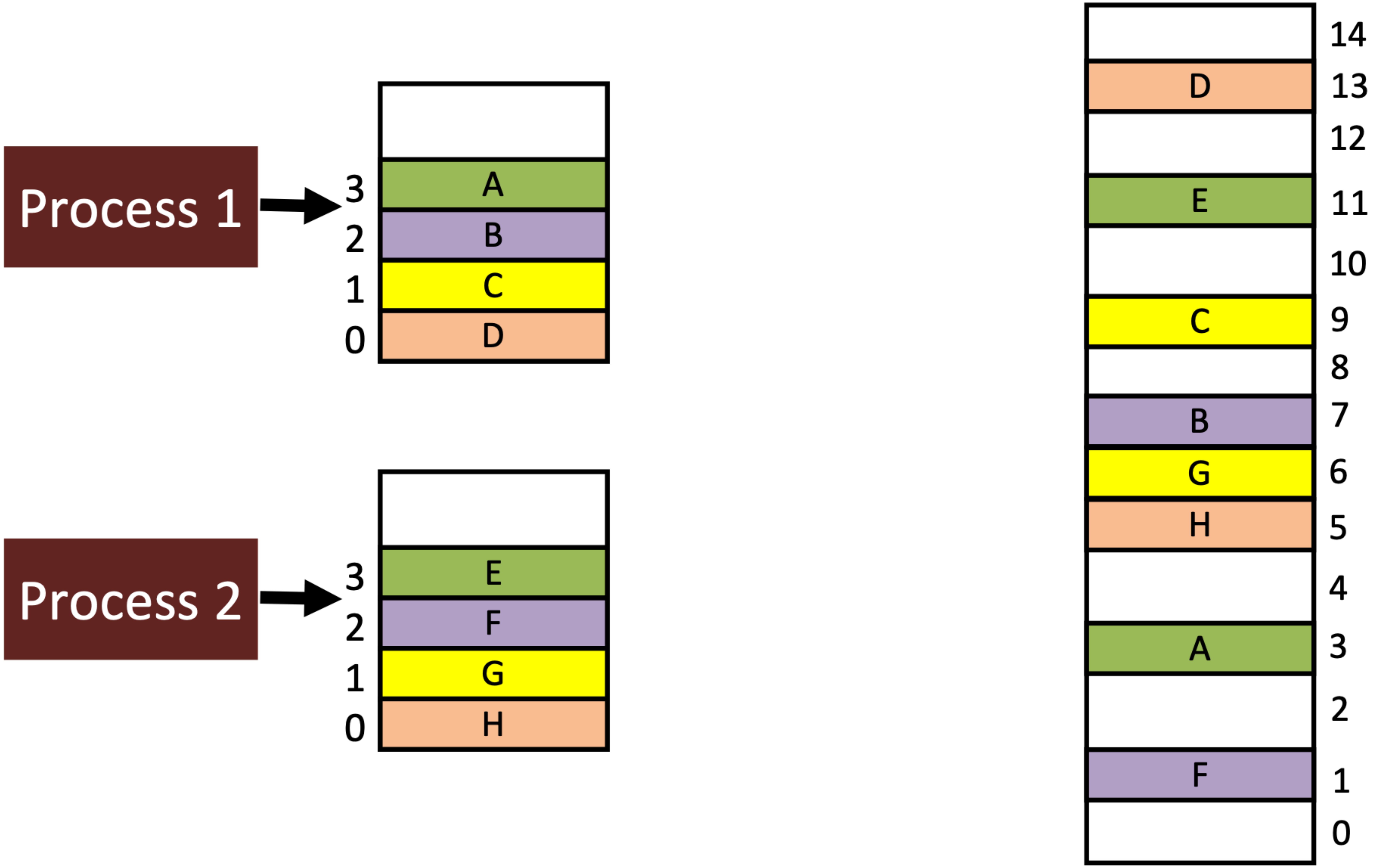


Give each process an **illusion** that it has **exclusive** access to entire main memory

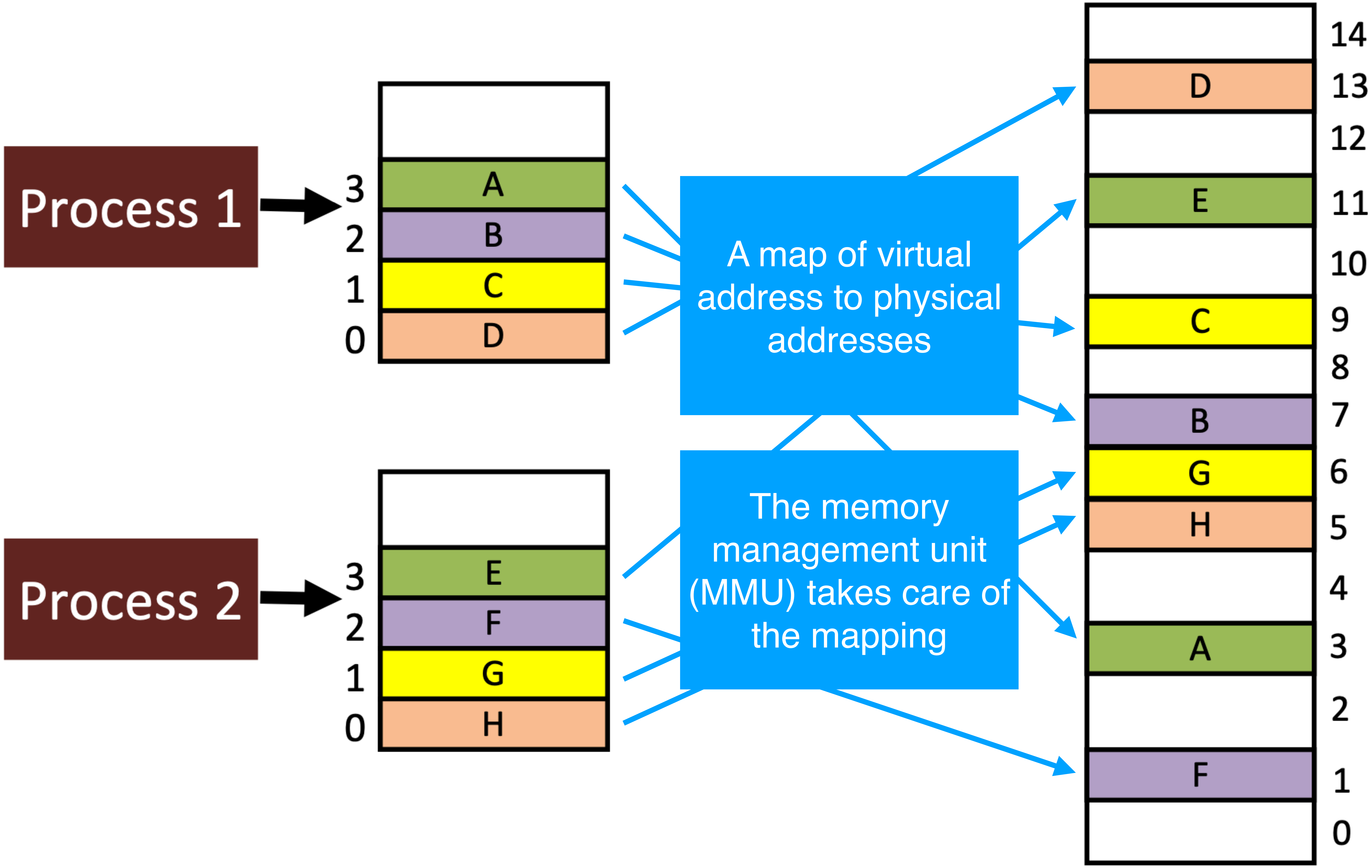
But in Reality



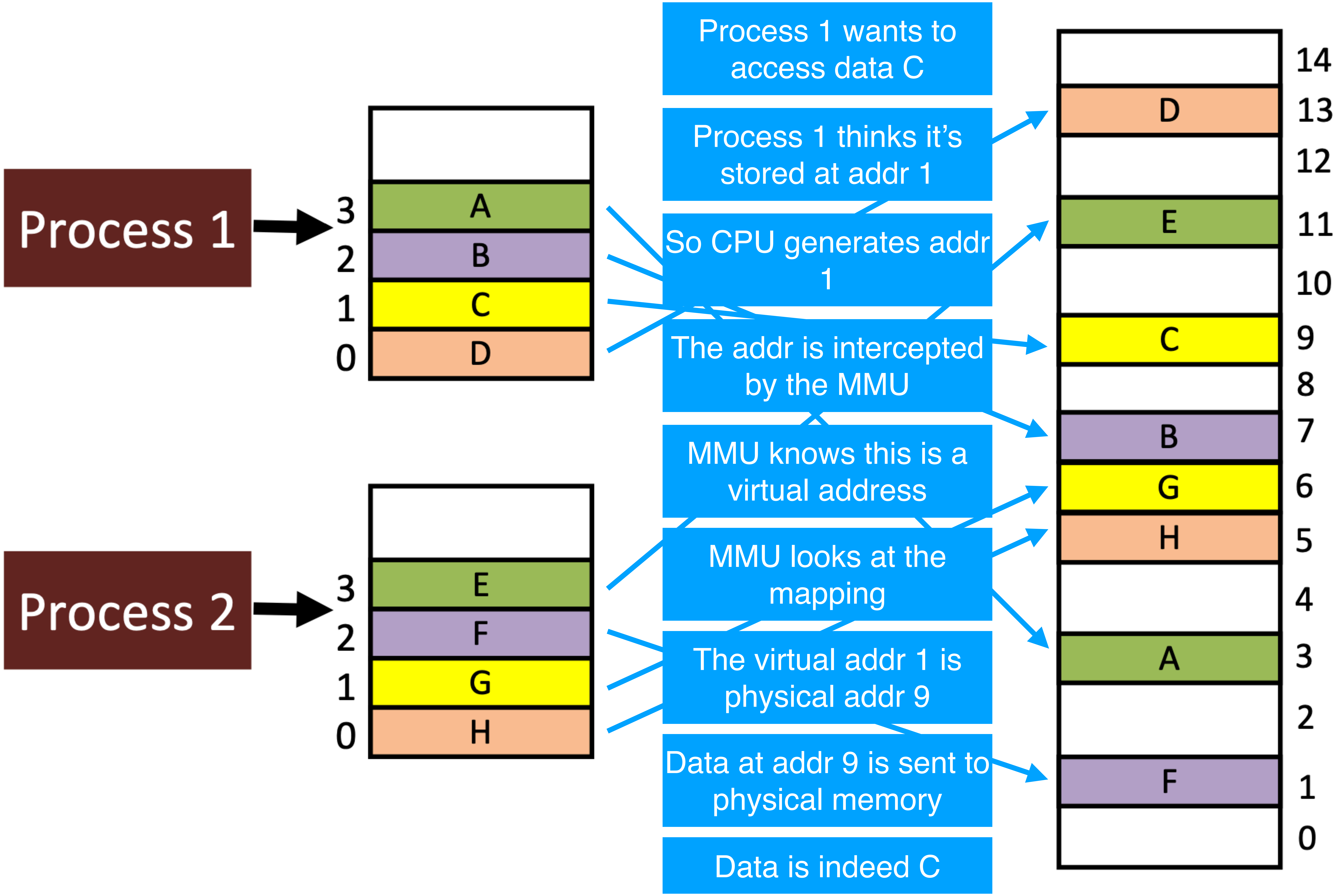
How Do We Create the Illusion?



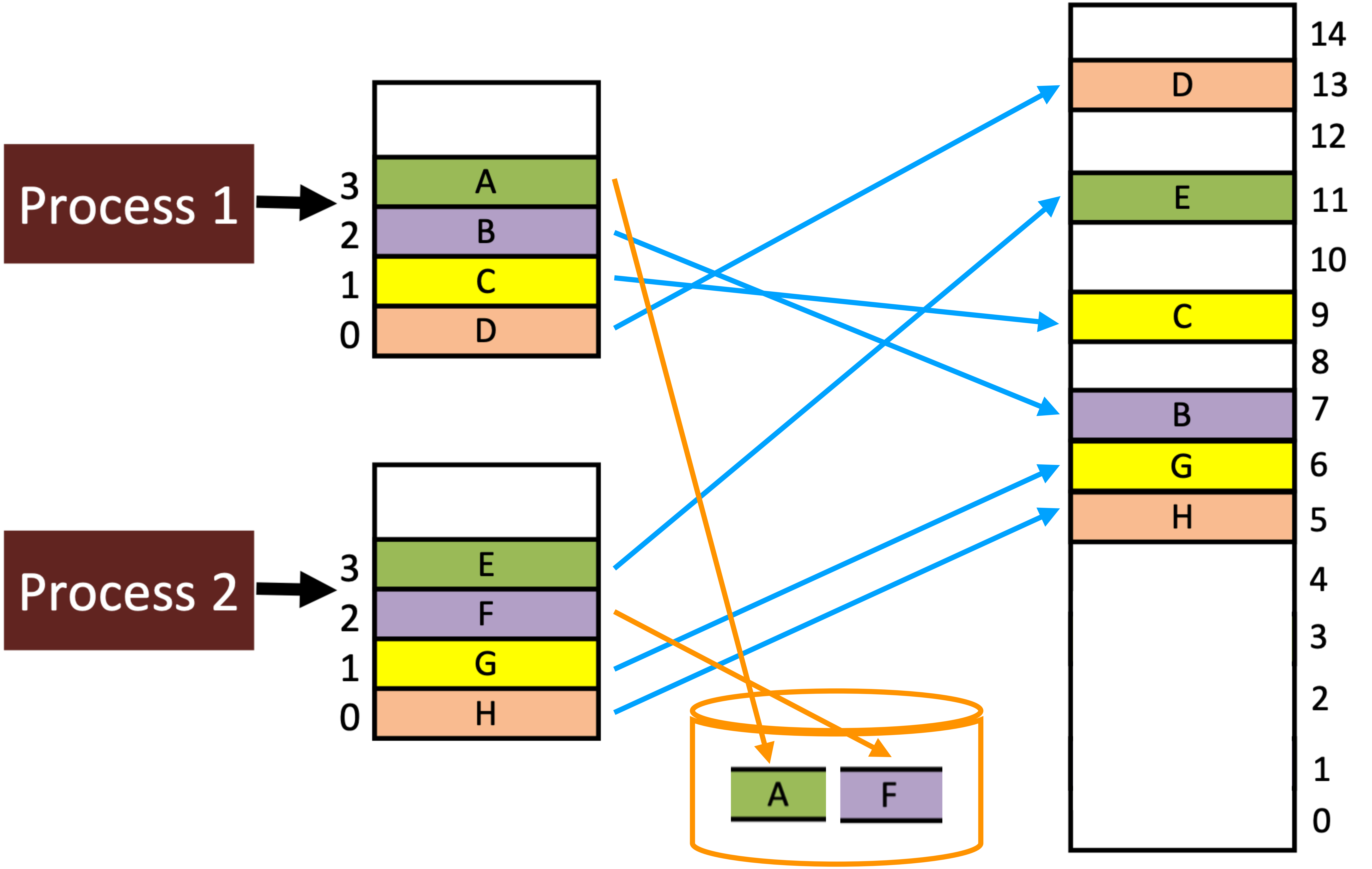
How Do We Create the Illusion?



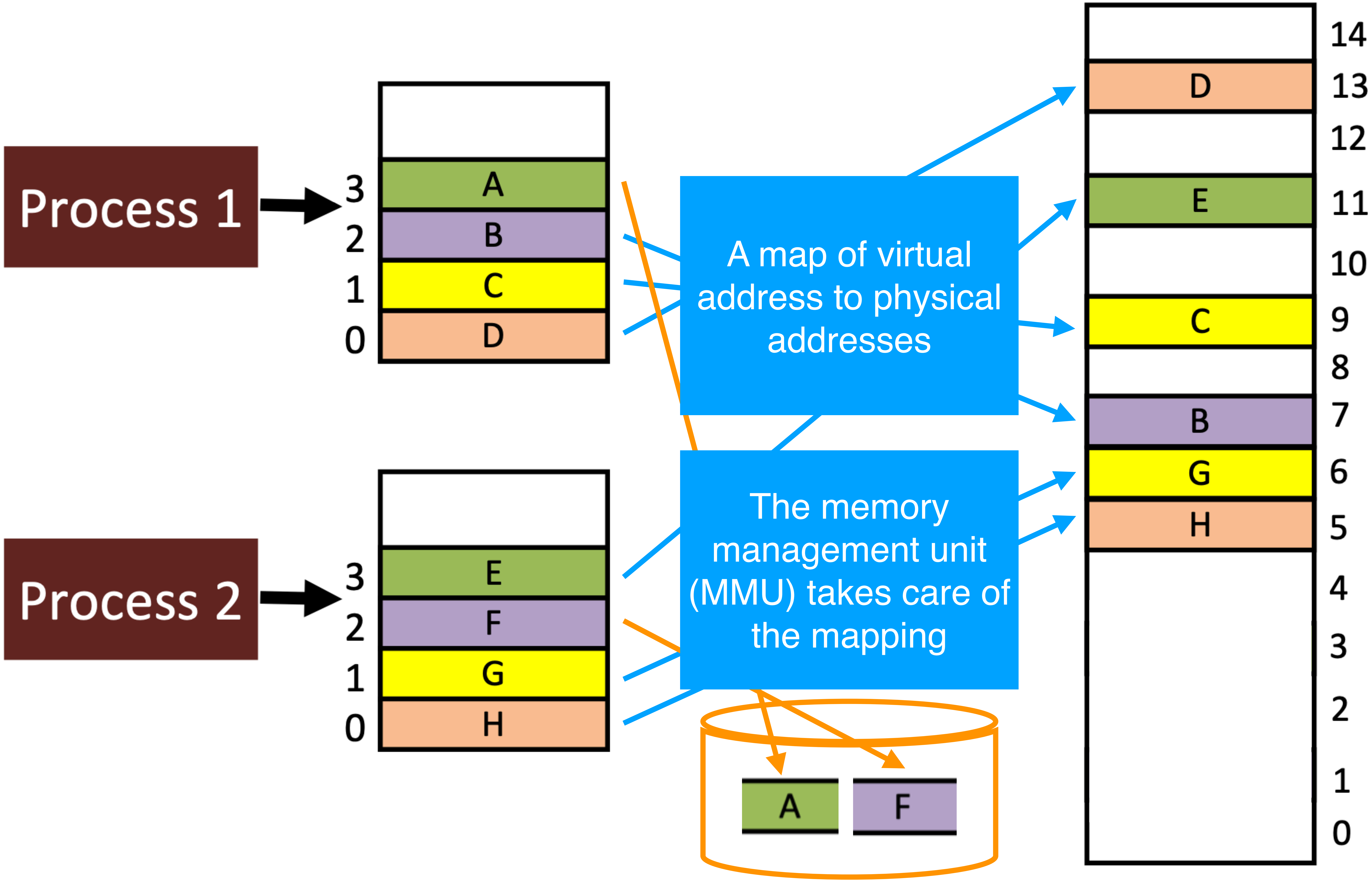
How Do We Create the Illusion?



How Do We Create the Illusion?

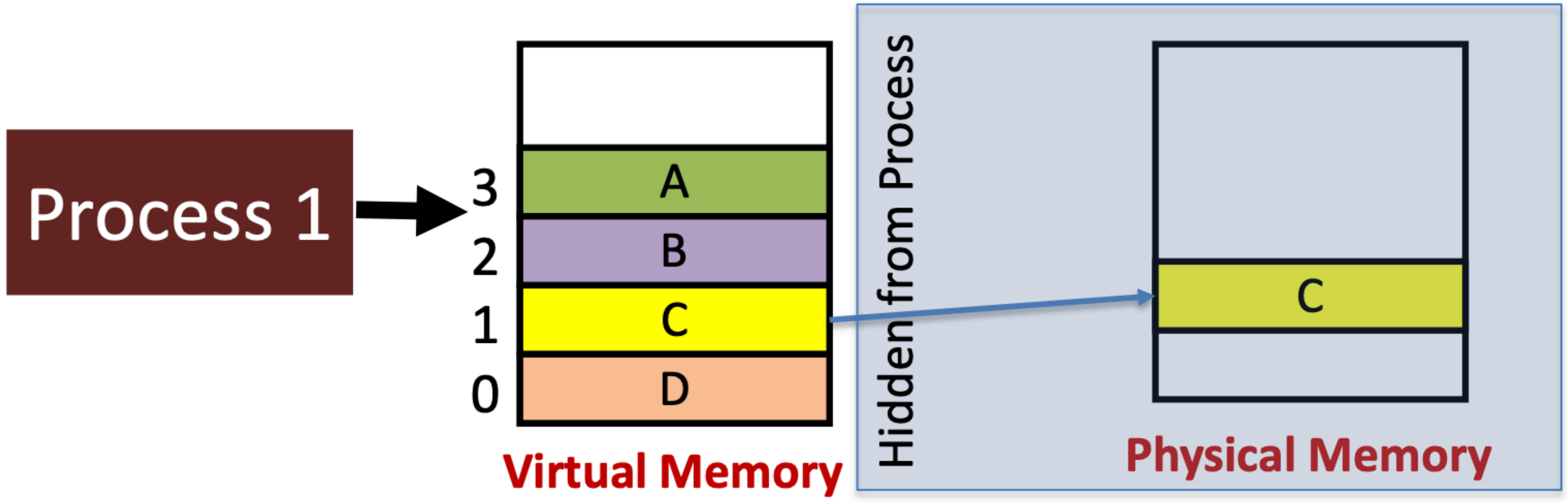


How Do We Create the Illusion?



Big Picture: (Virtual) Memory

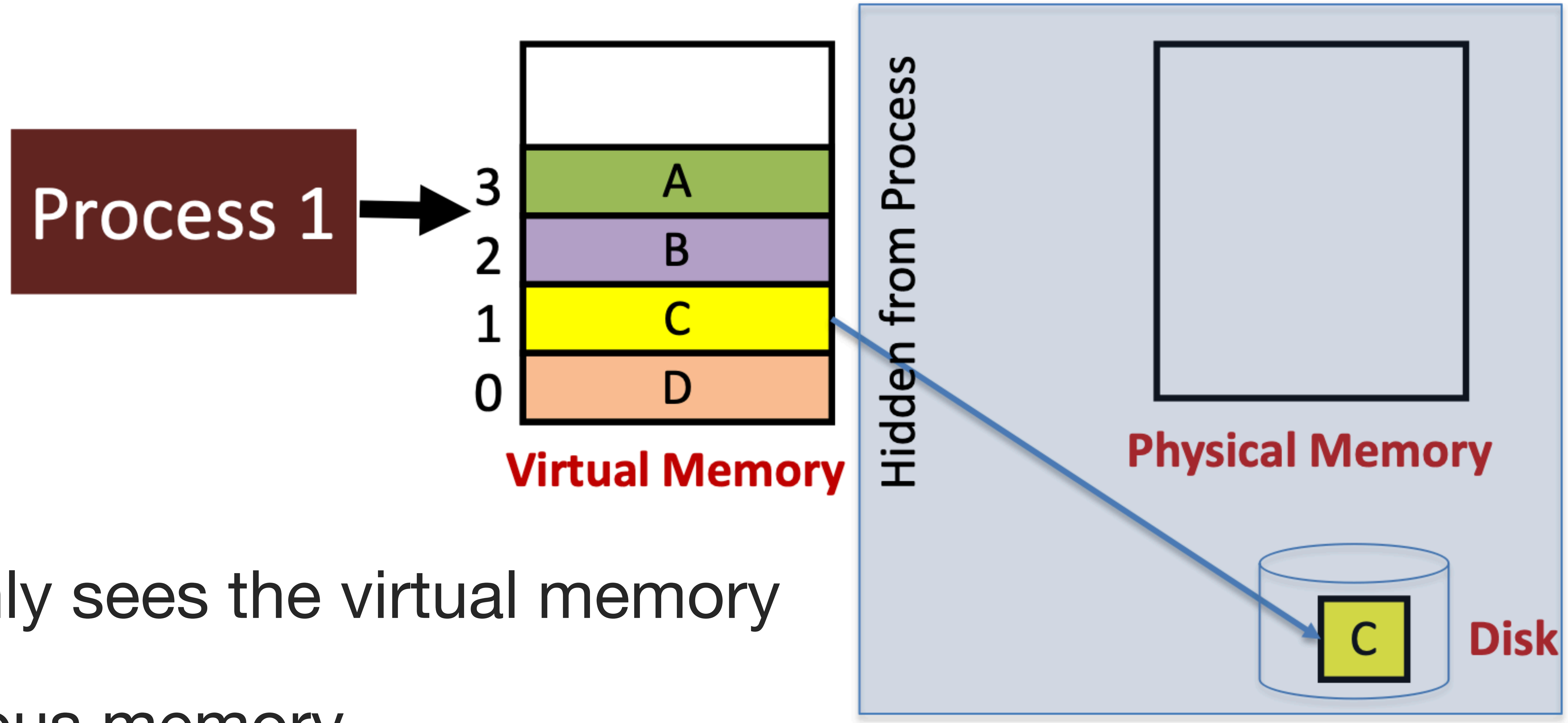
- From a process's perspective –



- Process only sees the virtual memory
    - If run out of memory, MMU maps data on disk in a transparent manner

Big Picture: (Virtual) Memory

- From a process's perspective –



- Process only sees the virtual memory
  - Contiguous memory
  - There's no need to recompile — Only mappings need to be updated

Virtual Memory

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

If the CPU uses 32-bit virtual addresses, then:

$$N = 32$$

The virtual addresses go from:

$$0 \text{ to } 2^{32} - 1$$

~4 GB of addressable space per process

Virtual Memory

- Each process has its own virtual address space:
  - Program/CPU can access any address from 0 … 2^N N = number of bits in the virtual address
  - 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:
  - So all accesses are *indirect* through a virtual address
  - Then, **map** translate fake virtual address to a real physical address
  - Redirect load/store to the physical address

The Advantages of Virtual Memory

- **Easy relocation**
  - Can put code/data anywhere in physical memory
  - The virtual addresses are the same; the MMU translates them

E.g.:

- Program A is mapped to virtual addresses 0-1 MB
- Program B is also mapped to virtual addresses 0-1 MB

A and B's "pieces" of memory are physically at different DRAM places, but both see a contiguous 0-1 MB address space

The Advantages of Virtual Memory

- **Higher memory utilization**
    - The virtual memory to only load “pieces” of program that are used into RAM
    - Physical memory can be **overcommitted**

E.g.:

- Program A allocates 1 GB of memory but only actually accesses 100 MB
- Only those 100 MB are loaded into DRAM; the rest stays on disk

This enables multiple programs to run **concurrently**, even if total memory exceeds DRAM

The Advantages of Virtual Memory

- **Easy sharing**
  - Processes can share the **same physical memory** via virtual memory mapping

E.g.:

- Shared libraries (like `libc.so`) are mapped into many processes' virtual address spaces
- Each process sees the library at its own virtual address, but there's only one copy in DRAM

Outline

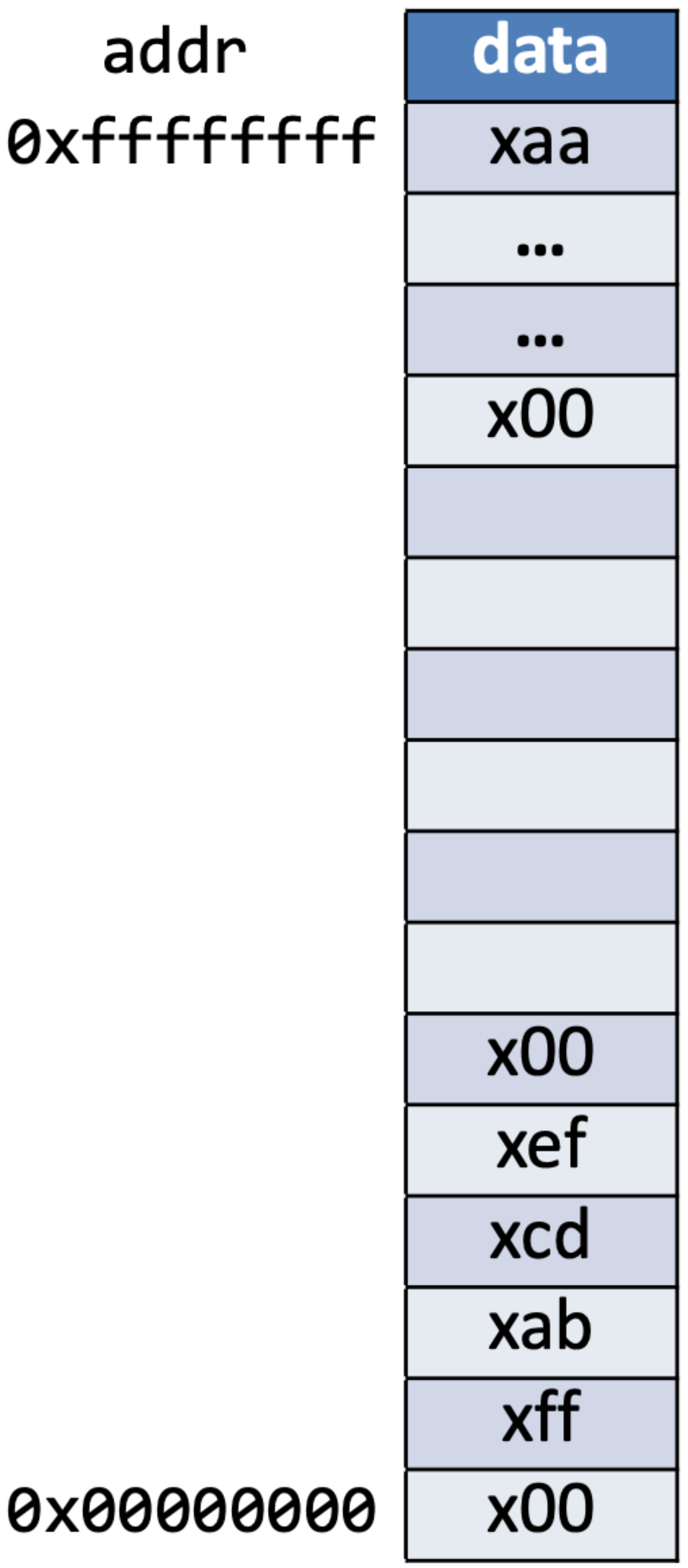
- How do we run multiple processes together?
- How does virtual memory work?
  - i.e., how do we create the “map” that maps a **virtual address** generated by the CPU to a **physical address** used by main memory?

Outline

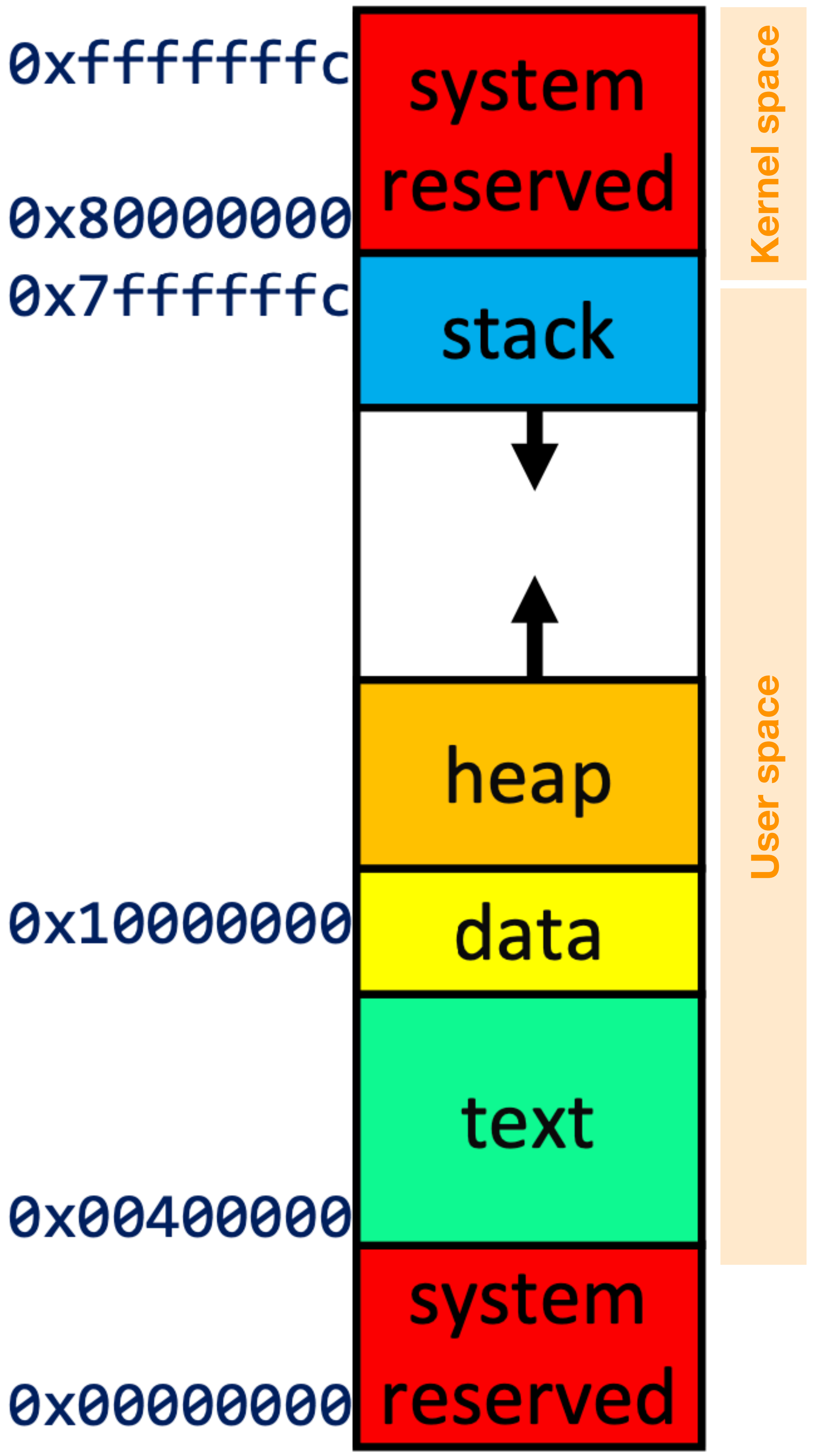
- How do we run multiple processes together?
- How does virtual memory work?
  - i.e., how do we create the “map” that maps a **virtual address** generated by the CPU to a **physical address** used by main memory?
    - Address Translation
    - Overhead
    - Paging

Picture Memory As …

Byte Array:



Segments:

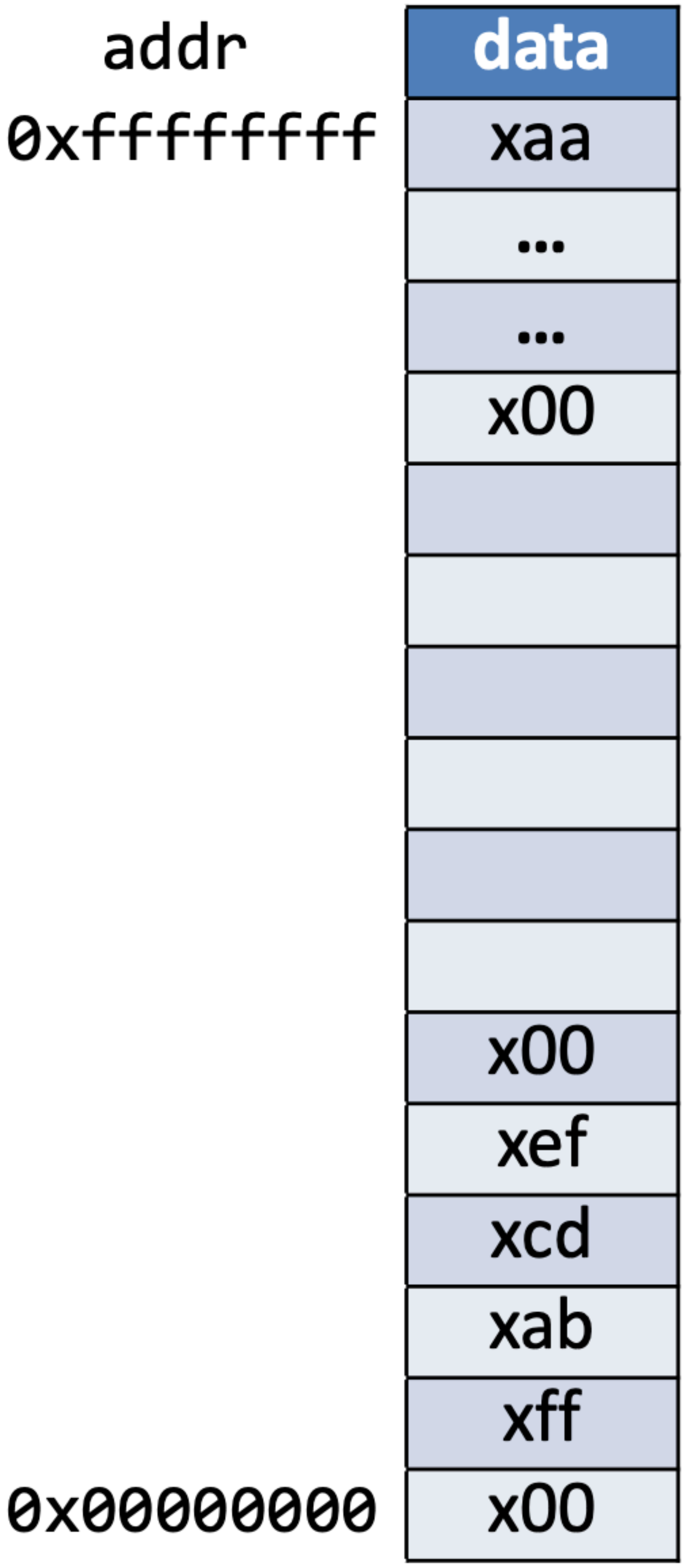


The virtual-to-physical address mapping

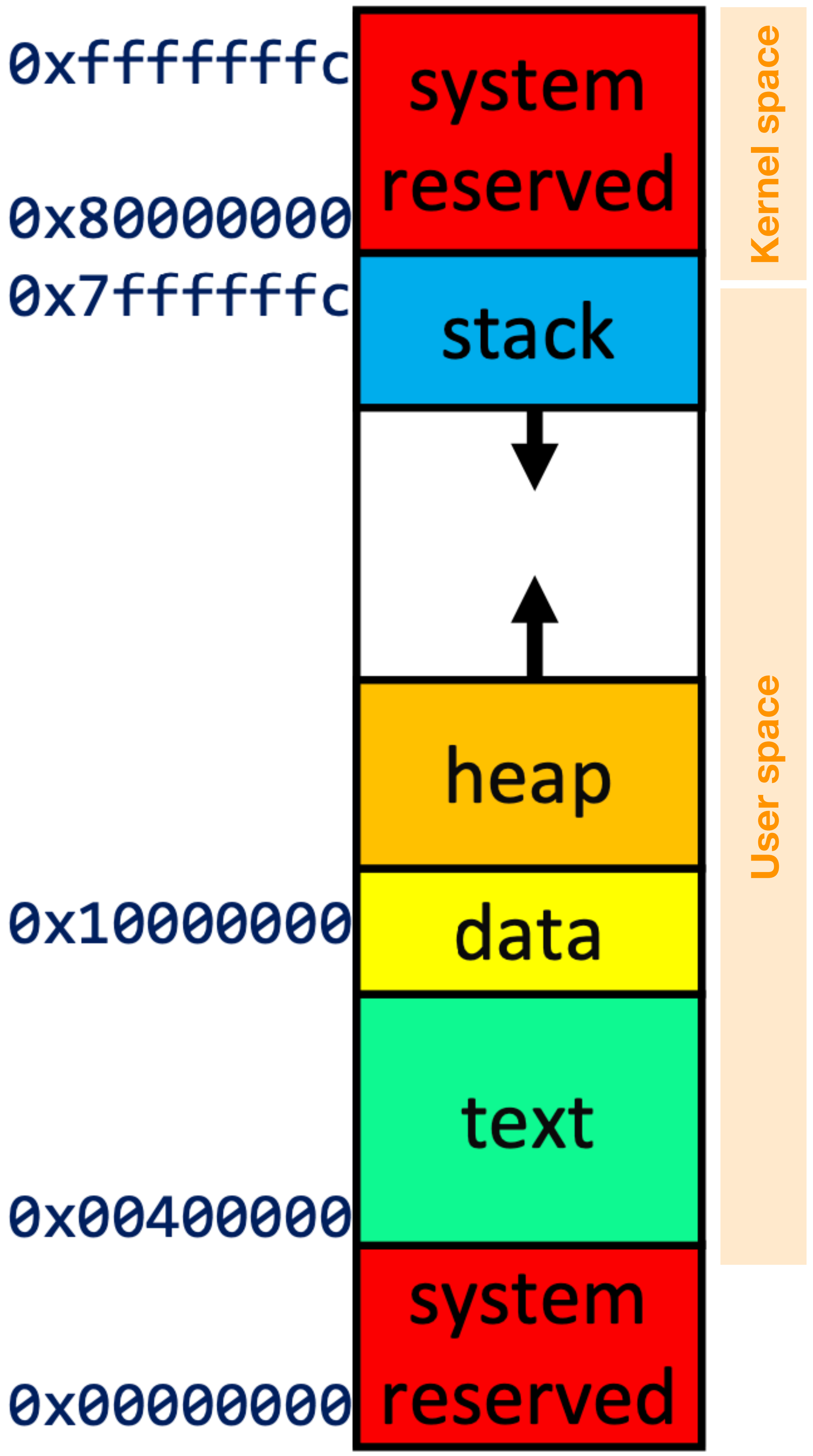
- The **CPU generates a virtual address** when running a program
- The OS wants to give each process the illusion of a contiguous, linear memory space
- To do this, the CPU uses a **page table**, which is a “map” maintained by the OS that links virtual pages to physical frames

Picture Memory As …

Byte Array:

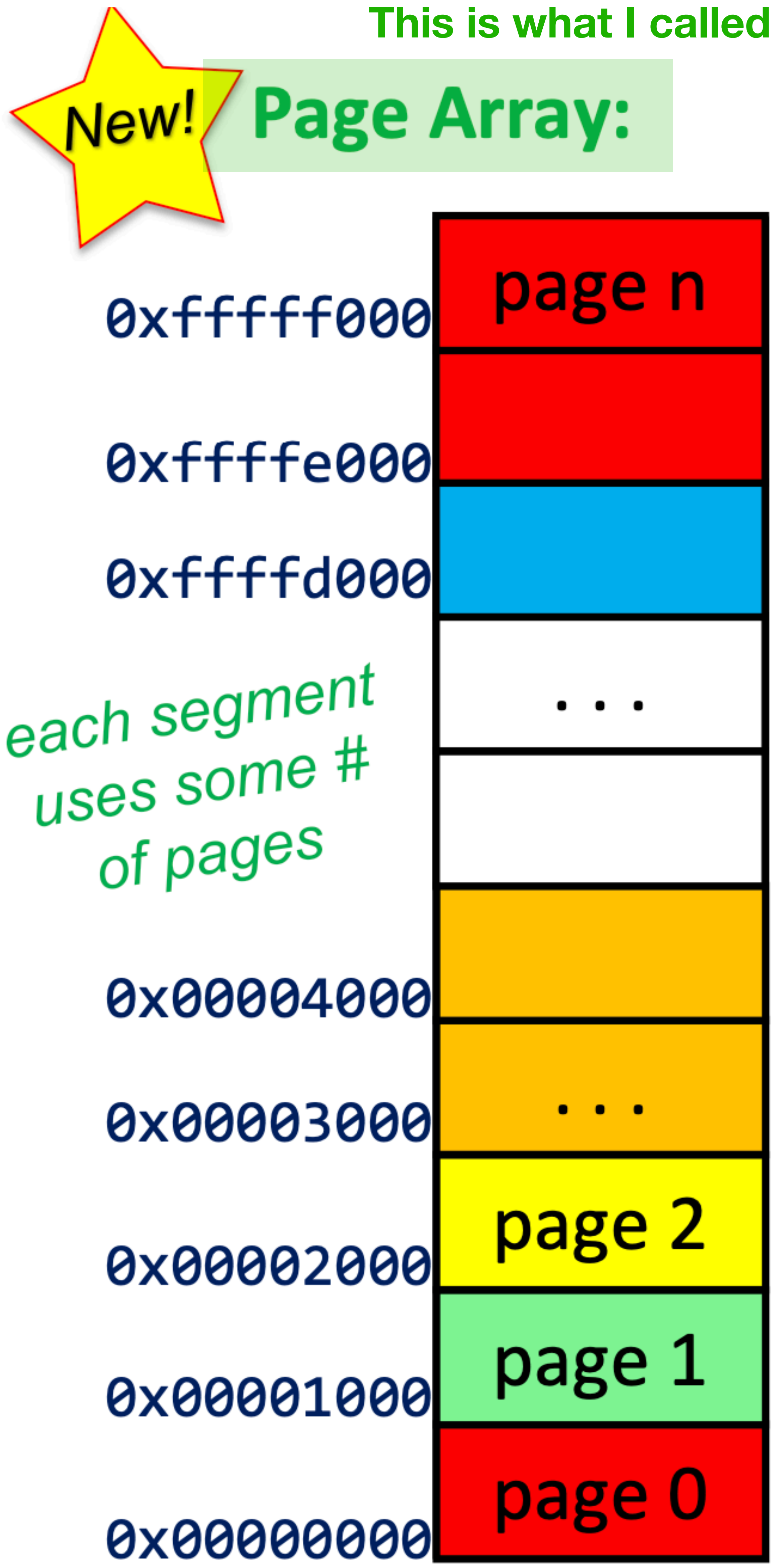


Segments:



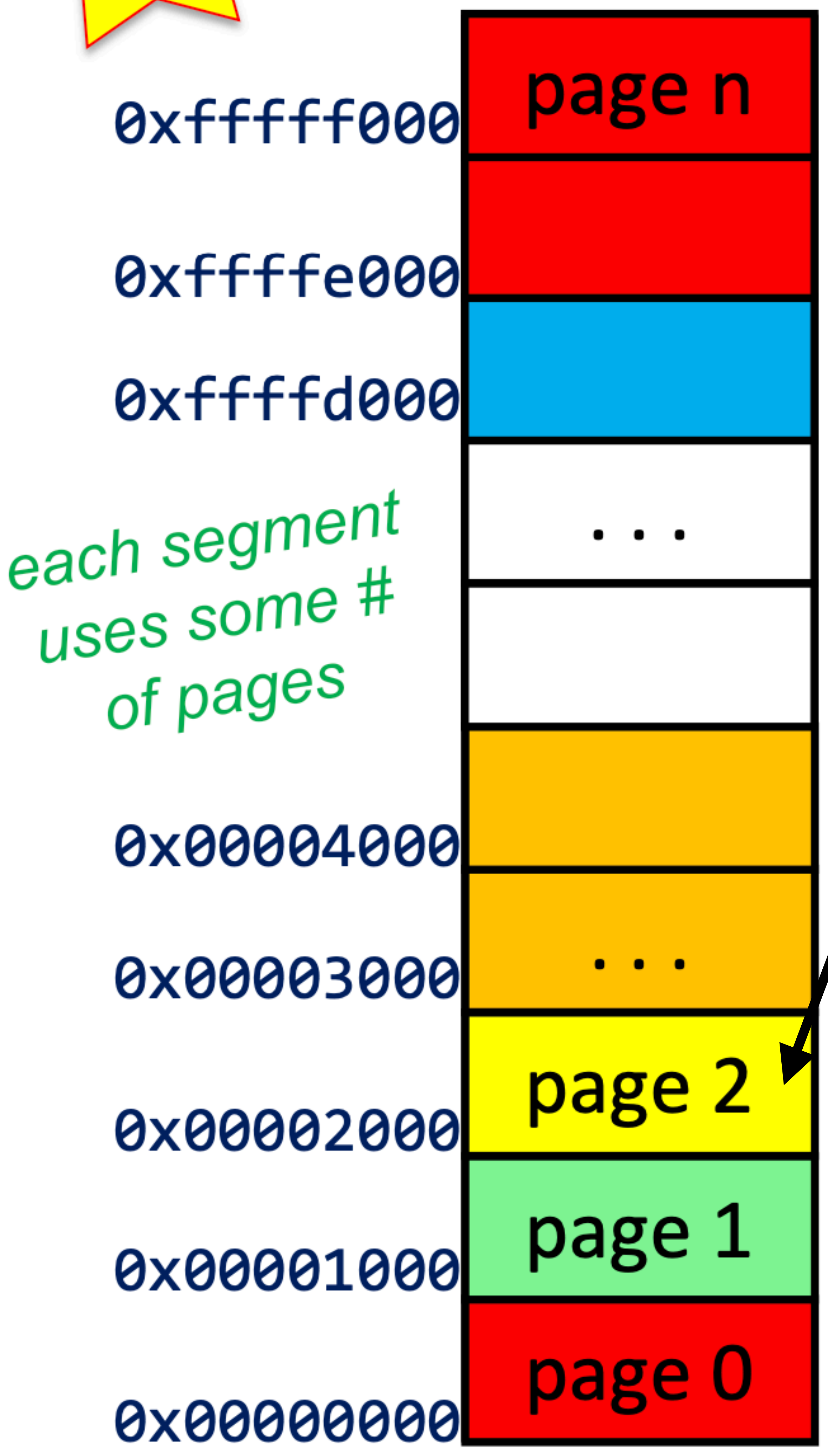
This is what I called "pieces" earlier!

Page Array:



Pages

New! **Page Array:**



- Let's suppose each page is 4KB
- Then anything in **page 2** has (hex) address: 0x00002**xxx**
- The lower 12 bits specify which byte you are in the page

  0x00002**200 = 0010 0000 0000 = byte 512**

- The upper bits = page **number** (PPN)
- The lower bits = page **offset**

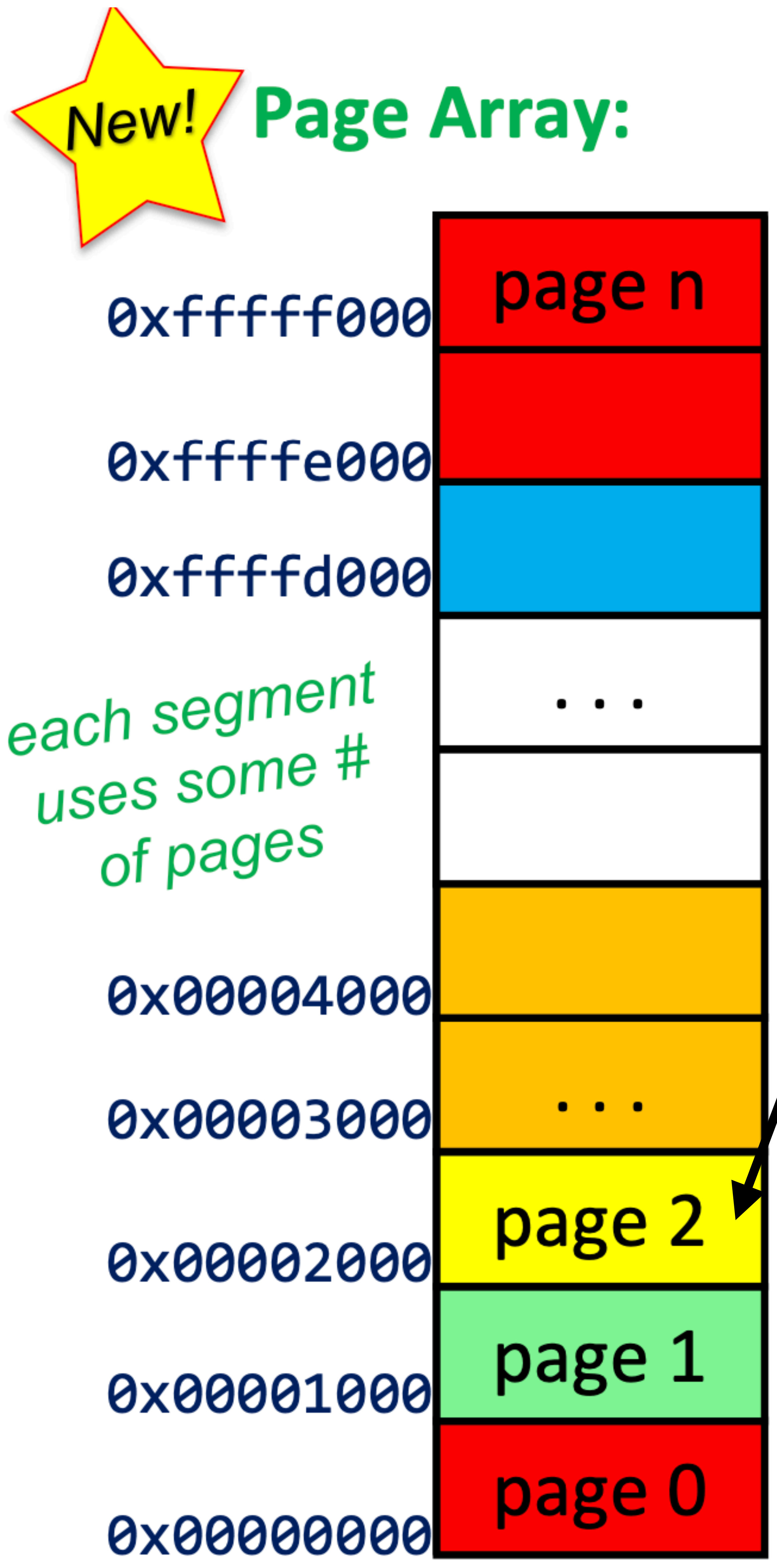
The virtual address breakdown

A virtual address is divided into:

- **Page number** → identifies which virtual page is being accessed
- **Offset** within the page → identifies the exact byte inside that page

```
        32-bit = [20 bit][12-bit]
virtual address = [page number][offset]
```

Sounds Familiar?



- Let's suppose each page is 4KB
- Then anything in page 2 has address: 0x00002**xxx**
- The lower 12 bits specify which byte you are in the page

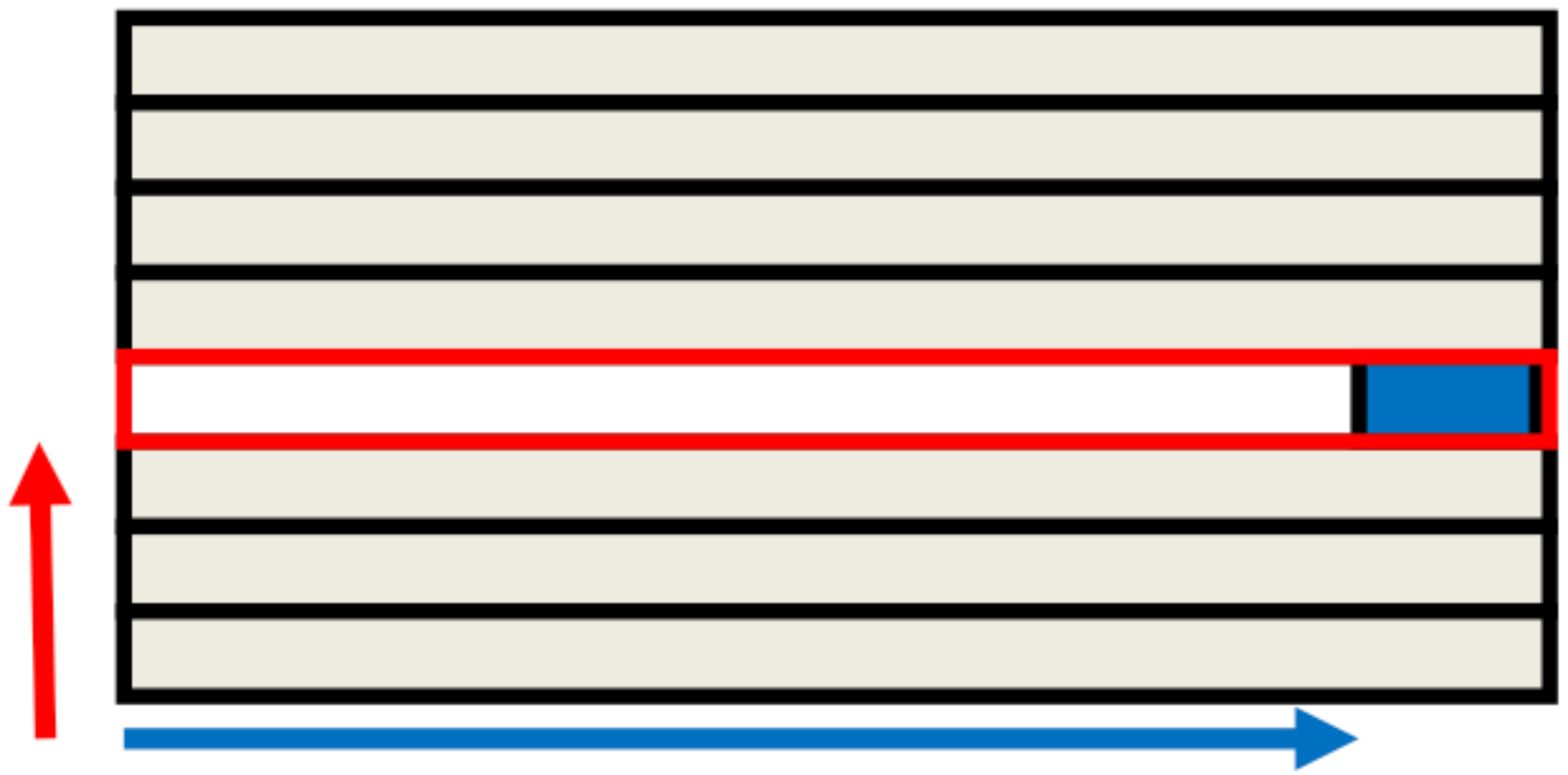
  0x00002**200 = 0010 0000 0000 = byte 512**

- The upper bits = page **number** (PPN)
- The lower bits = page **offset**

Data Granularity

- **ISA:** instruction specific: LB, LH, LW (ISA)
- **Registers:** 32 bits or 64 bits (ISA)

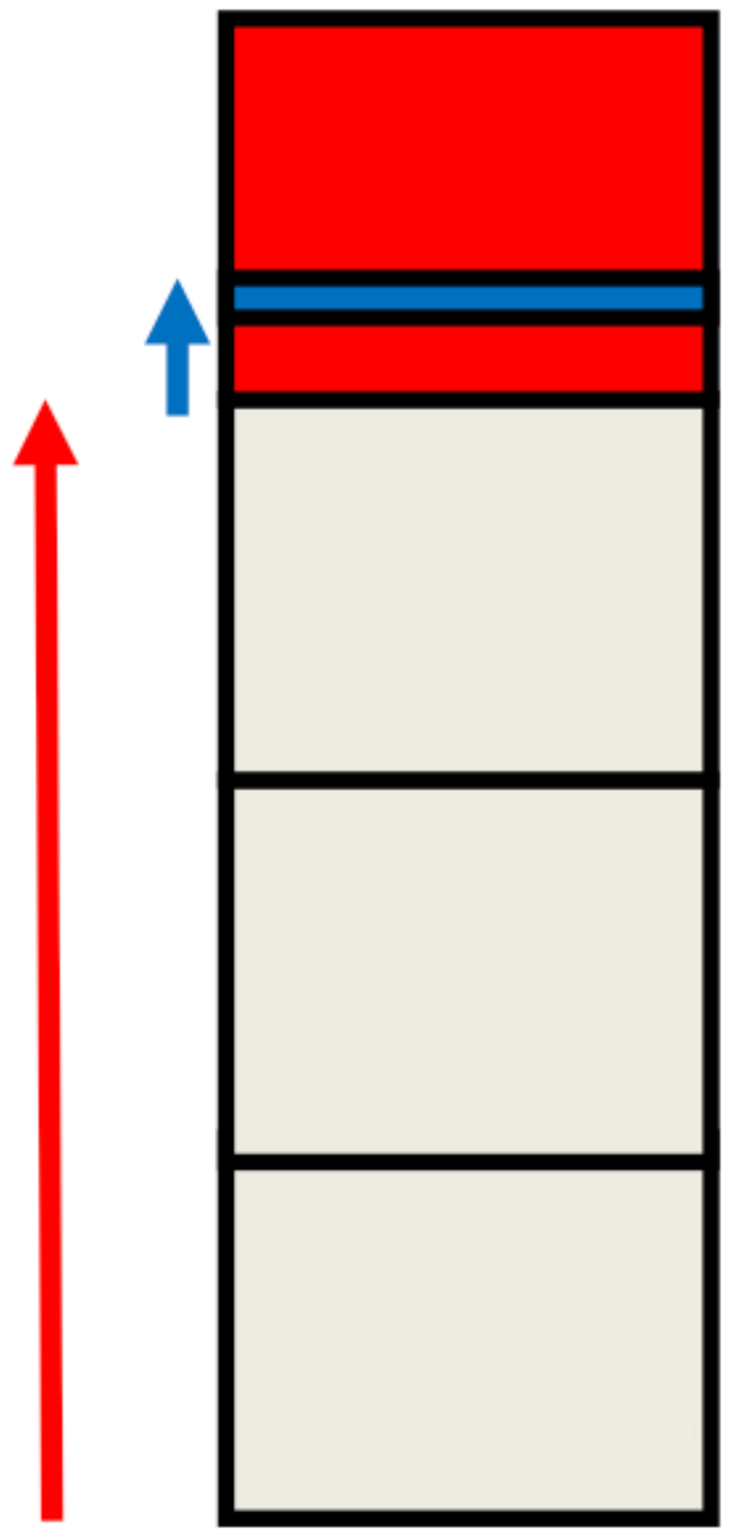
- **Caches:** Cache line/block (µarch)
  - The address bits divided into:
    - tag: sanity check for address match
    - index: which entry in the cache
    - offset: which byte in the line



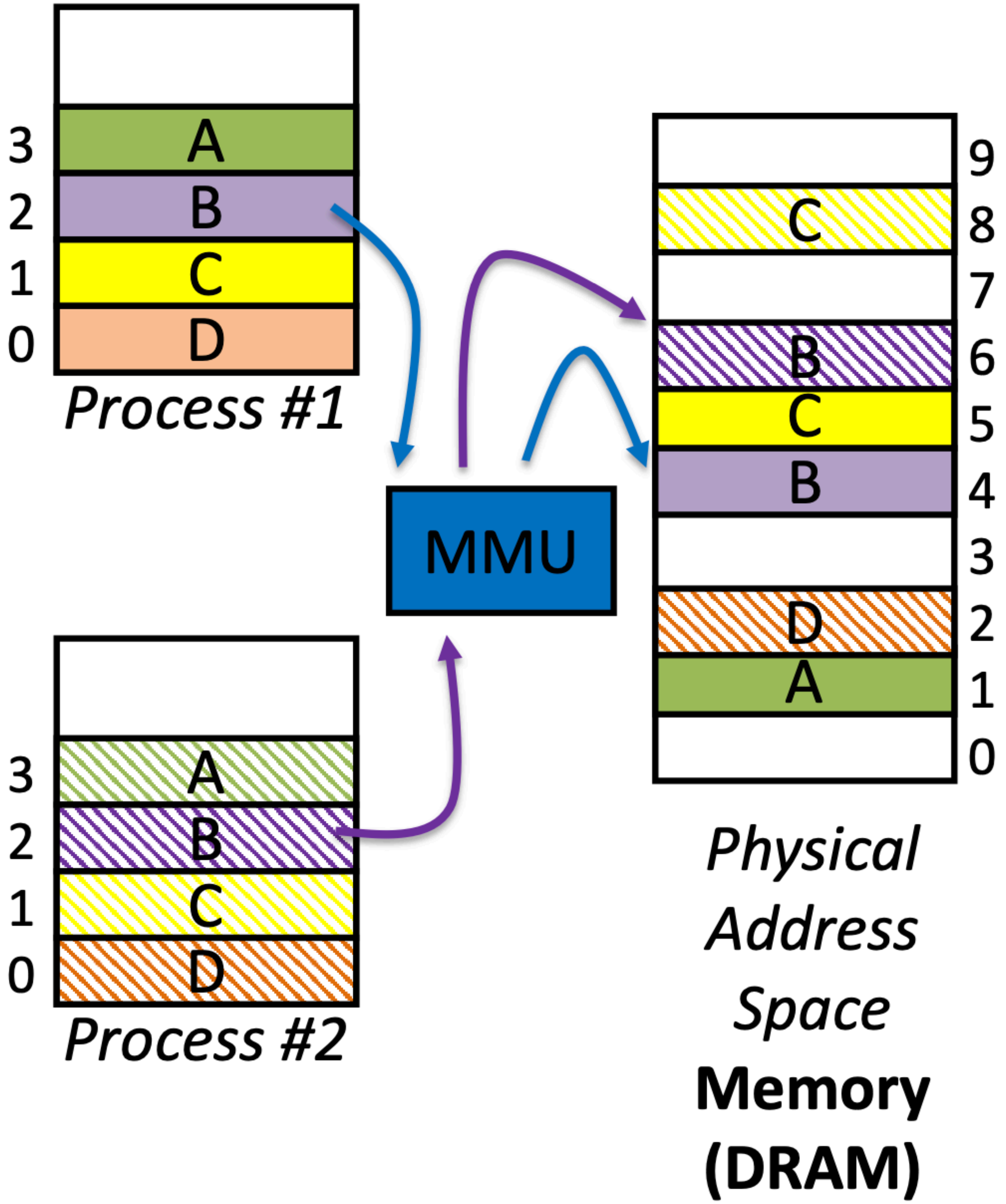
Data Granularity

- **ISA:** instruction specific: LB, LH, LW (ISA)
- **Registers:** 32 bits or 64 bits (ISA)

- **Memory:** Page
  - The address bits divided into:
    - page number: which page in memory
    - offset: which byte in the page



Address Translator (MMU)



- Processes use virtual memory
- DRAM uses physical memory
- **Memory Management Unit (MMU)**
  - It's <u>hardware</u>!
  - It translates virtual addresses to physical addresses on the fly