Lab this week is REVIEW and OPTIONAL #876

Pedro Pontes García STAFF
19 hours ago in General

UNPIN STAR WATCH 149 VIEWS

1

Hi everyone!

Lab this week will be a review of topics introduced after the midterm, in preparation for the final exam. We will have a worksheet and work through solutions together. It will NOT be open office hours like the previous optional lab. However, we will not take attendance, so there is no obligation to attend (but we strongly encourage it!).

See (most of) you at lab!

PS. Congrats on all the work you've done so far! Only a few more weeks and you'll be done with 3410, which is a thorough survey of systems topics and is not at all easy, especially as a first systems class. Good job :)

Comment Edit Delete ···

Add comment

Cornell Bowers CIS
Computer Science

Happy 2nd Birthday Nina!

CS3410: Computer Systems and Organization

LEC21: Virtual Memory

Professor Giulia Guidi
Wednesday, November 12, 2025

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

Plan for Today

- Review of virtual memory so far
- Paging and virtual memory

Review of virtual memory

Big Picture: Processes

Each process requires memory to hold:

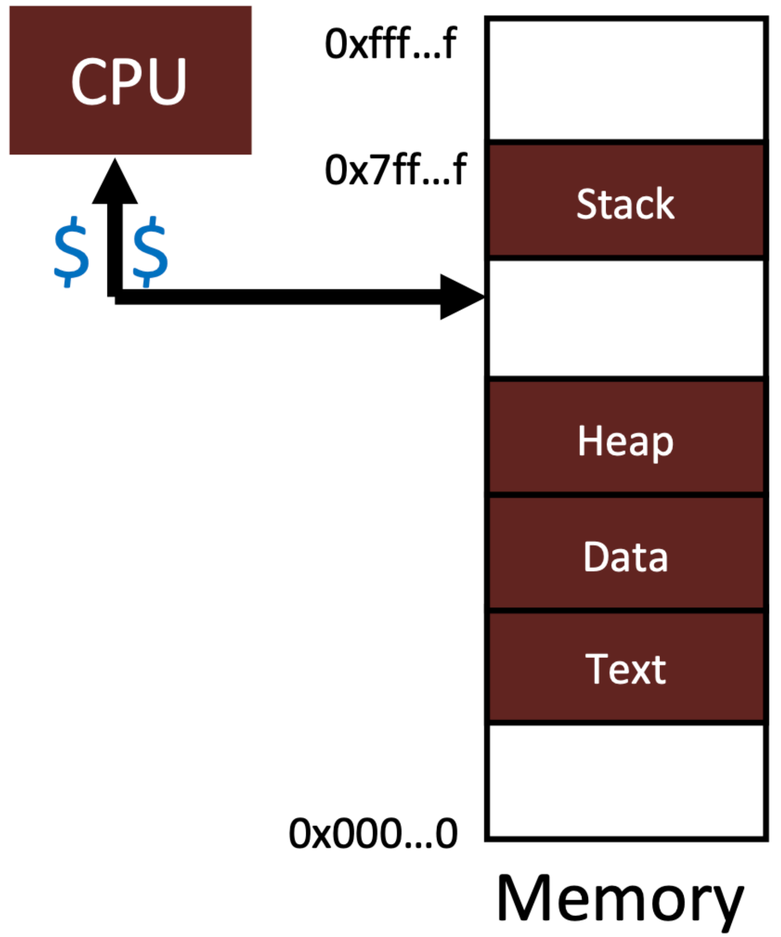
- Its **instructions** (the code to run)
- Its **data** (variables, heap)
- Its **stack** (function calls, local variables)

The **problem**:

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

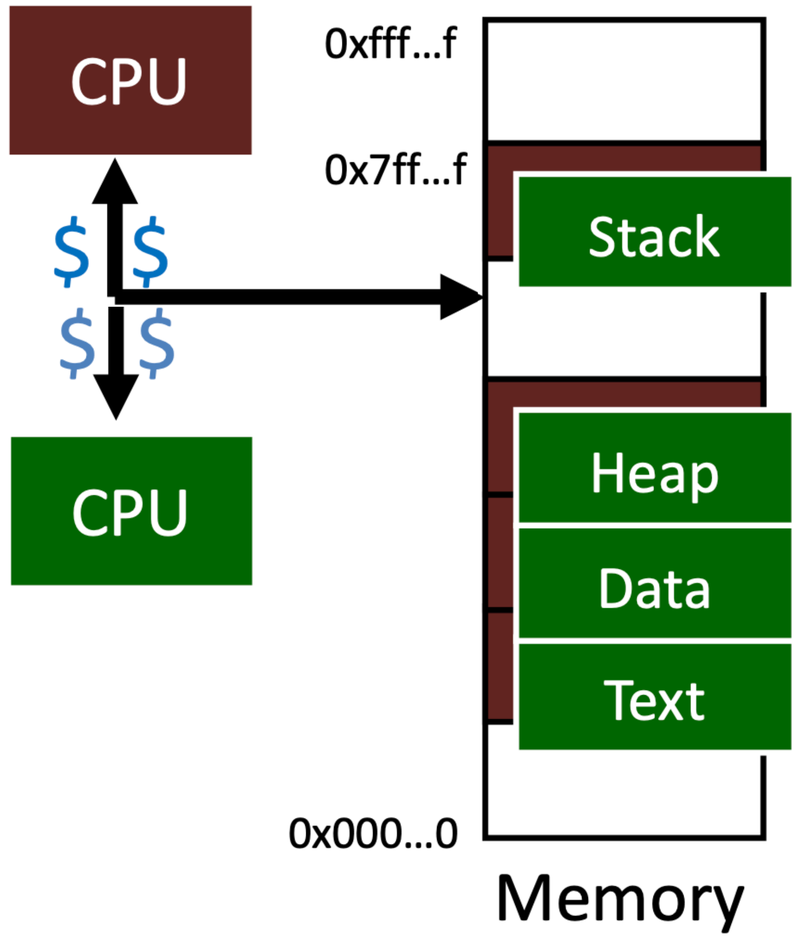
Processor & Memory

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



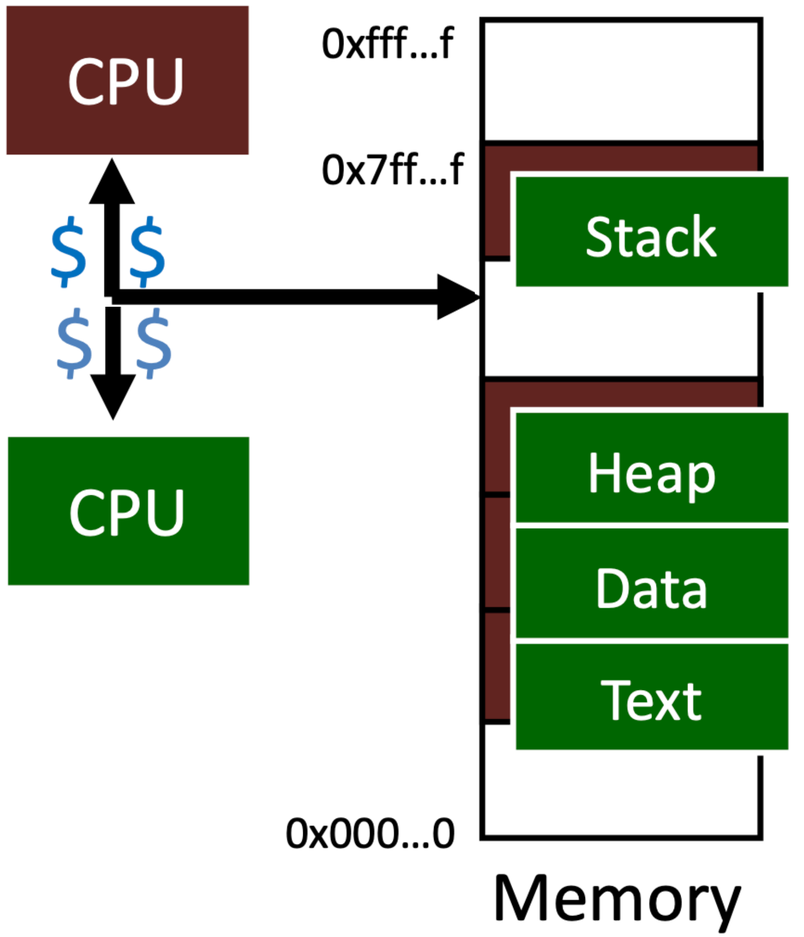
Processor & Memory

- 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

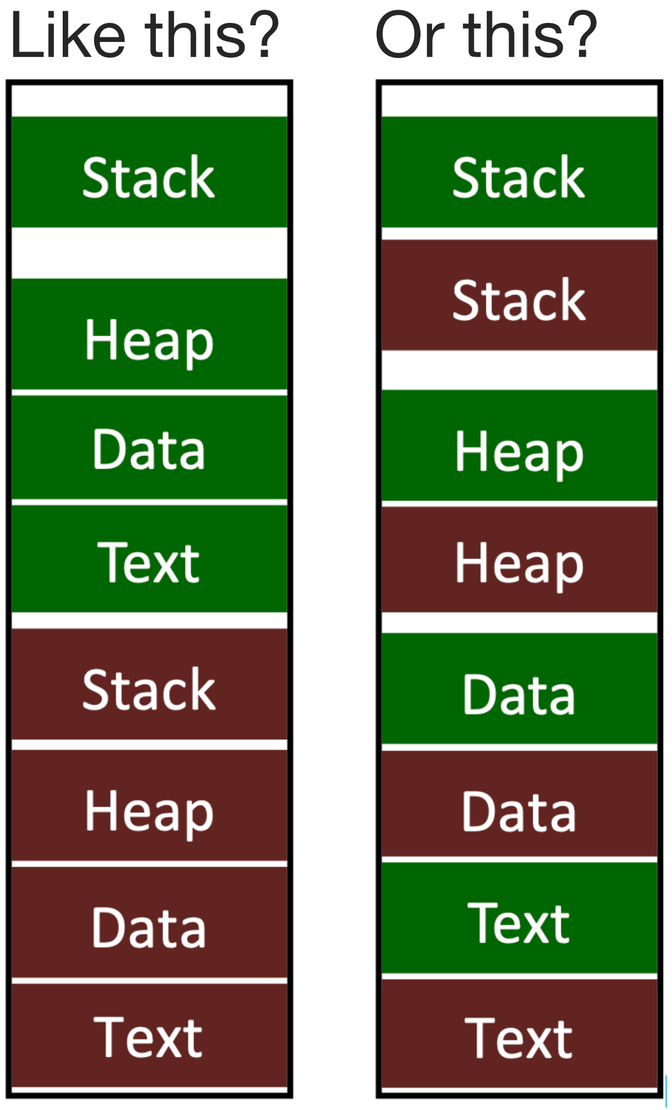
- So what happens when when another program is executed concurrently on another processor?
- The addresses will conflict
  - Even if CPUs take turns using memory bus
  - Solutions?
    - **Can we relocate second program?**



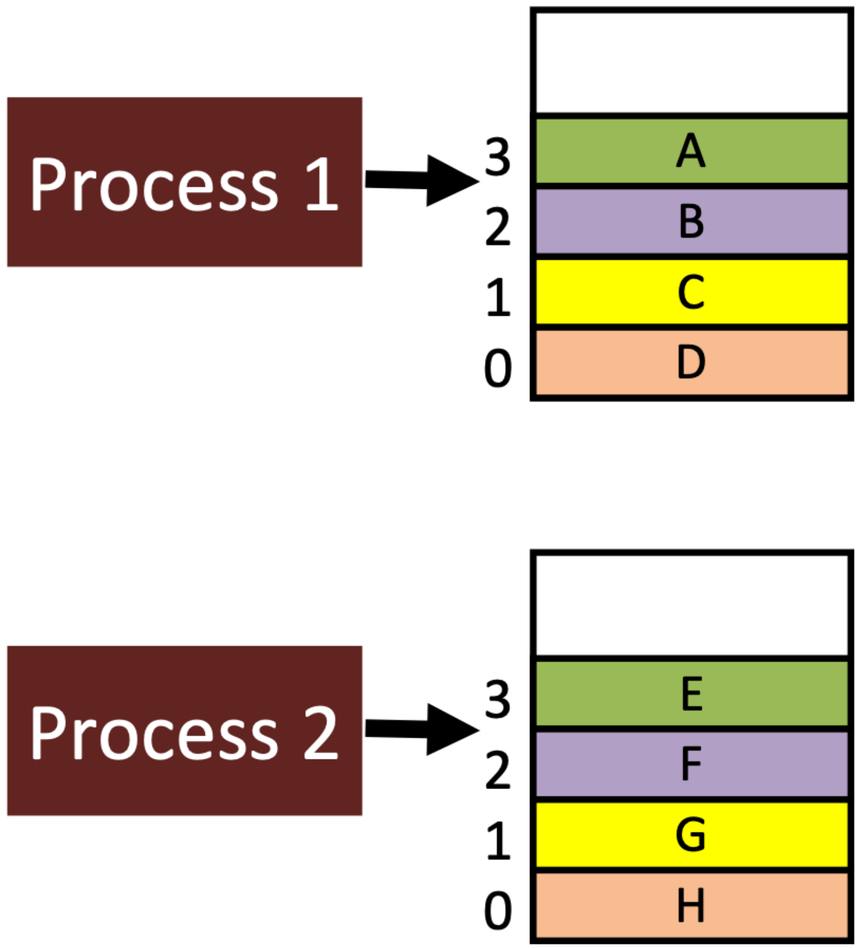
Can We Relocate Second Program?

- Yes but how?
- 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)

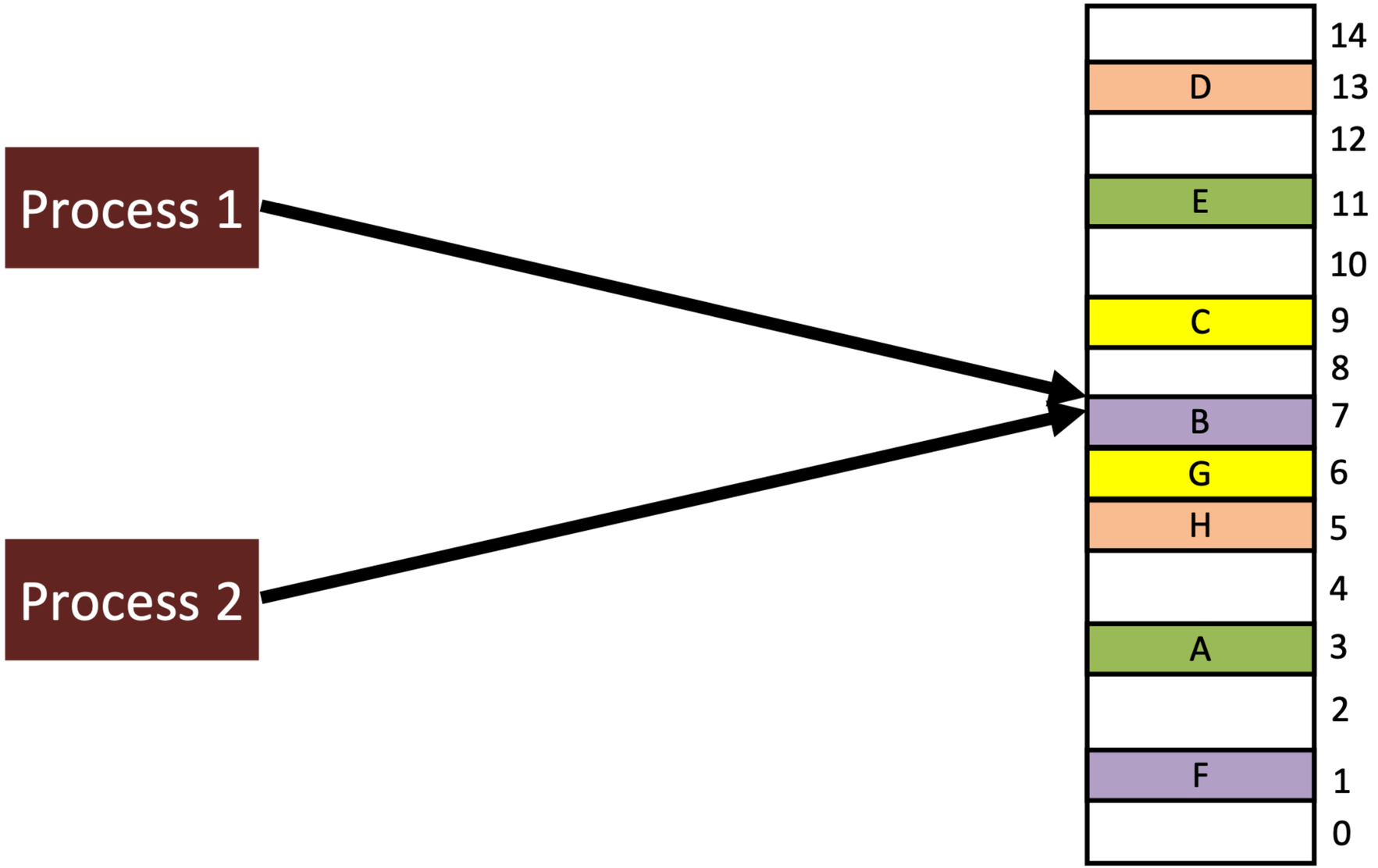


Big Picture: Virtual Memory

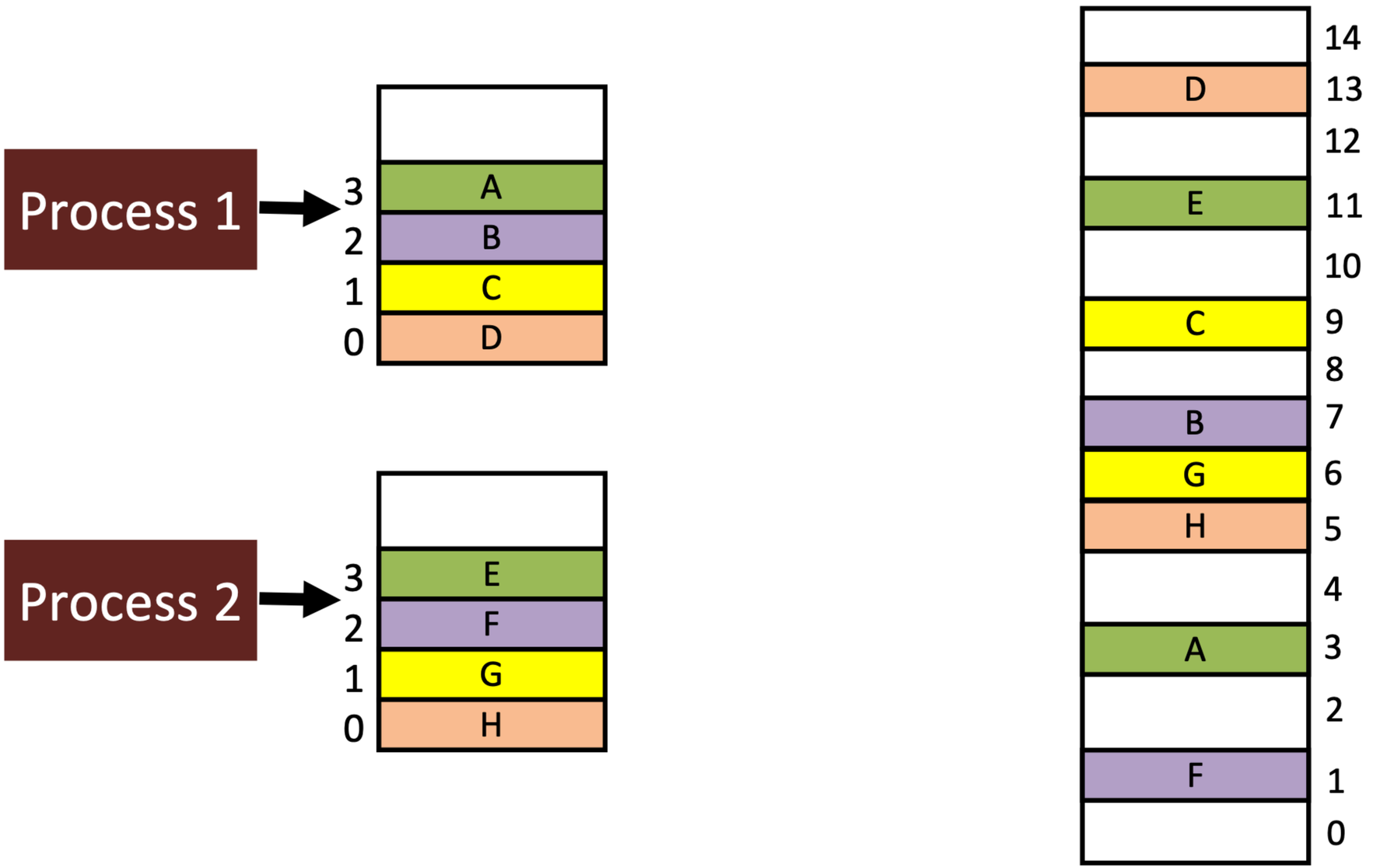


Give each process an **illusion** that it has **exclusive** access to entire main memory

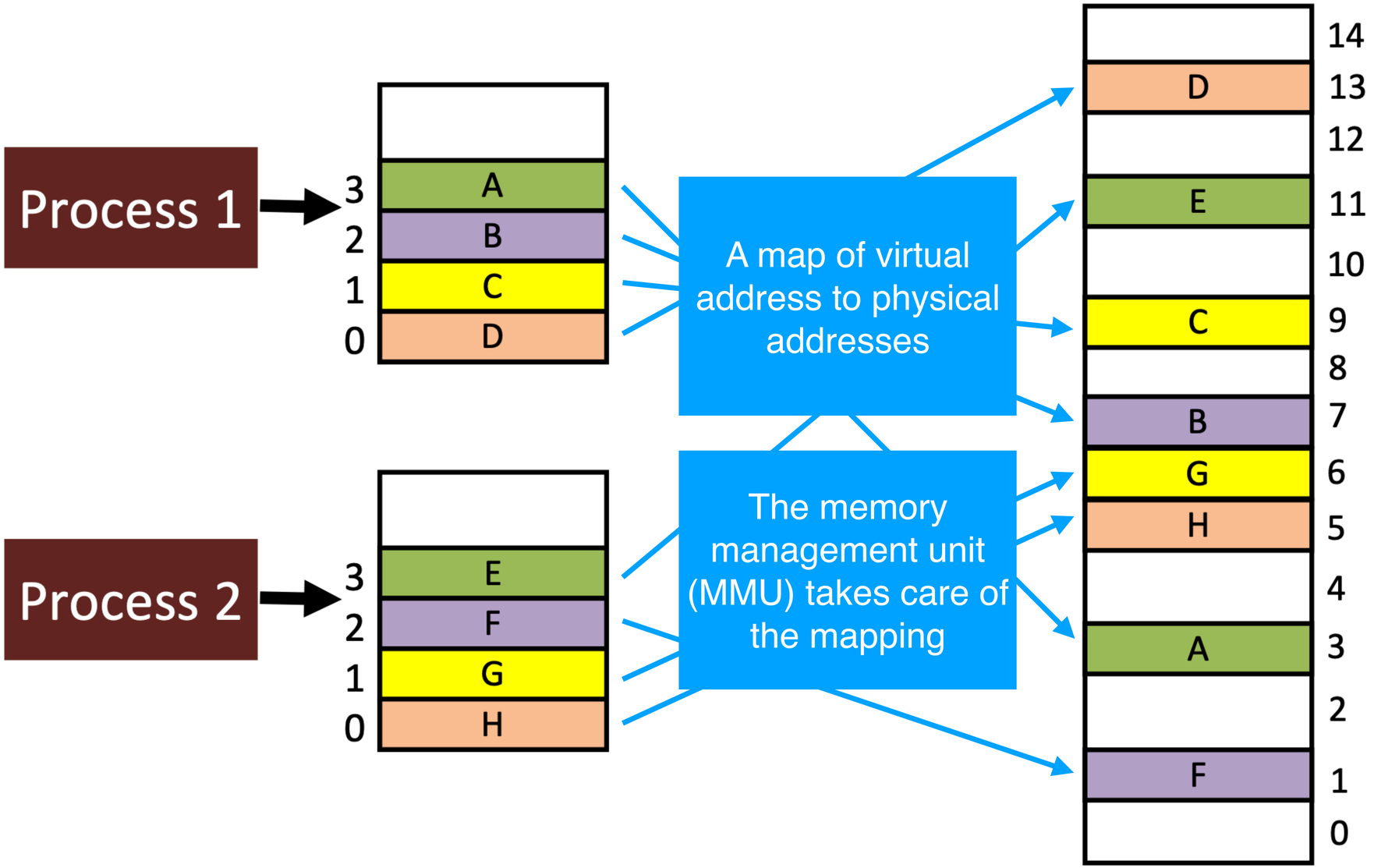
But in Reality



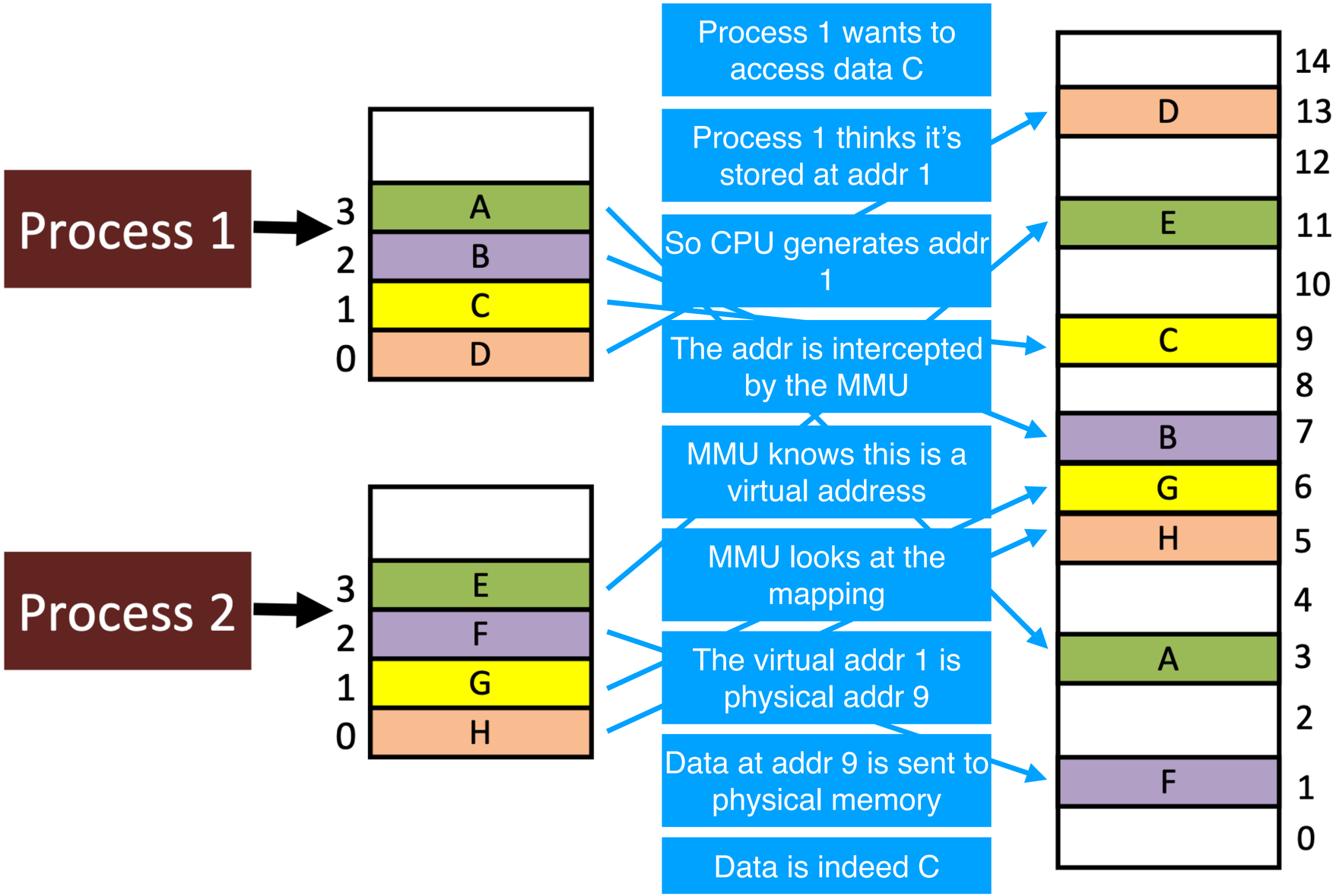
How Do We Create the Illusion?



How Do We Create the Illusion?



How Do We Create the Illusion?



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

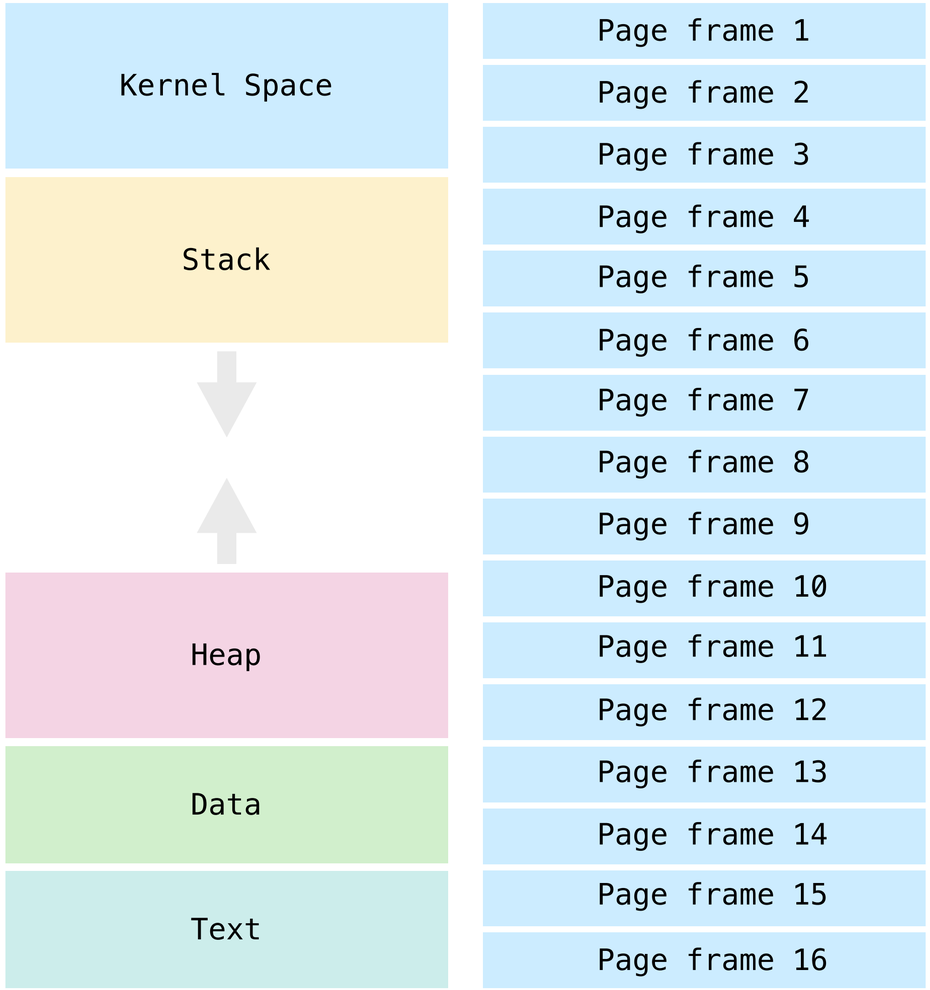
The Advantages of Virtual Memory

- **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

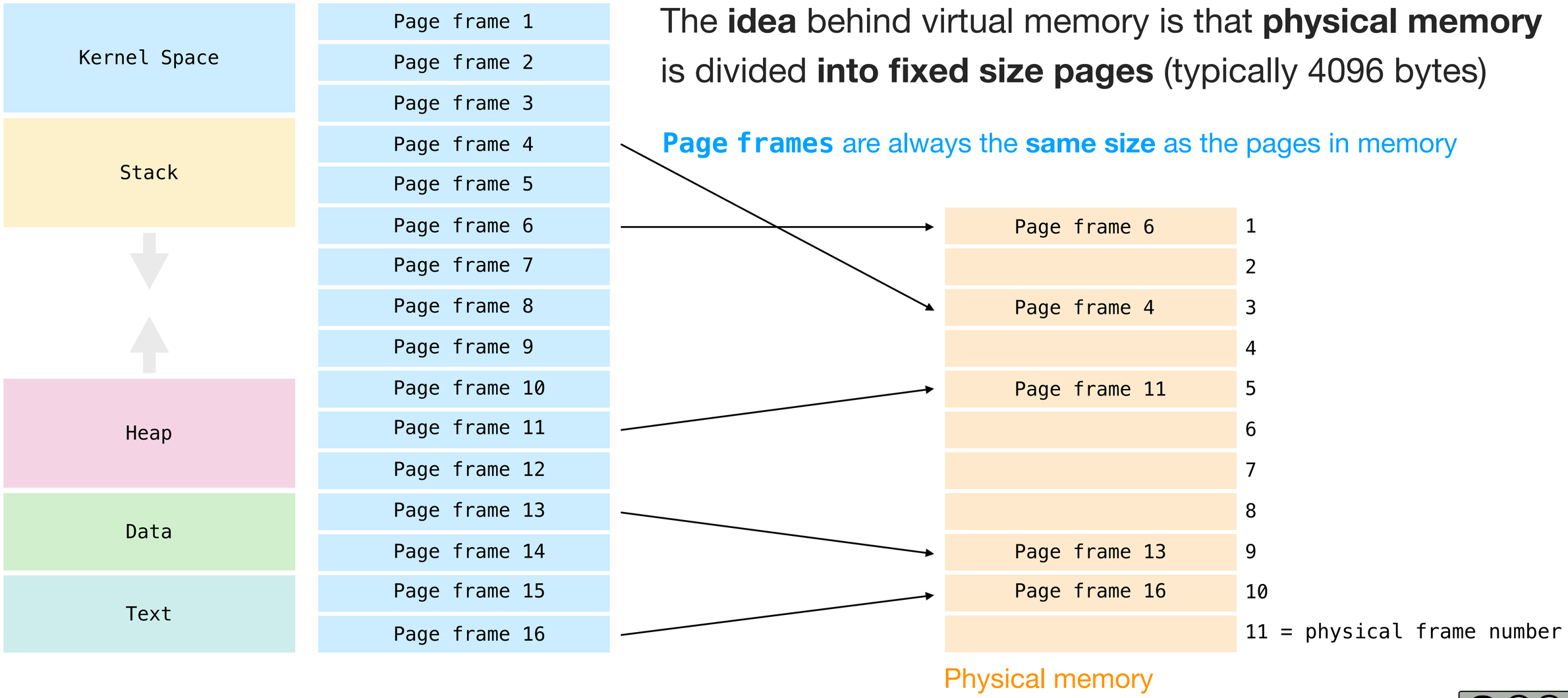
Picture Memory As …



The **idea** behind virtual memory is that **physical memory** is divided **into fixed size pages** (typically 4096 bytes)

`Page frames` are always the **same size** as the pages in memory

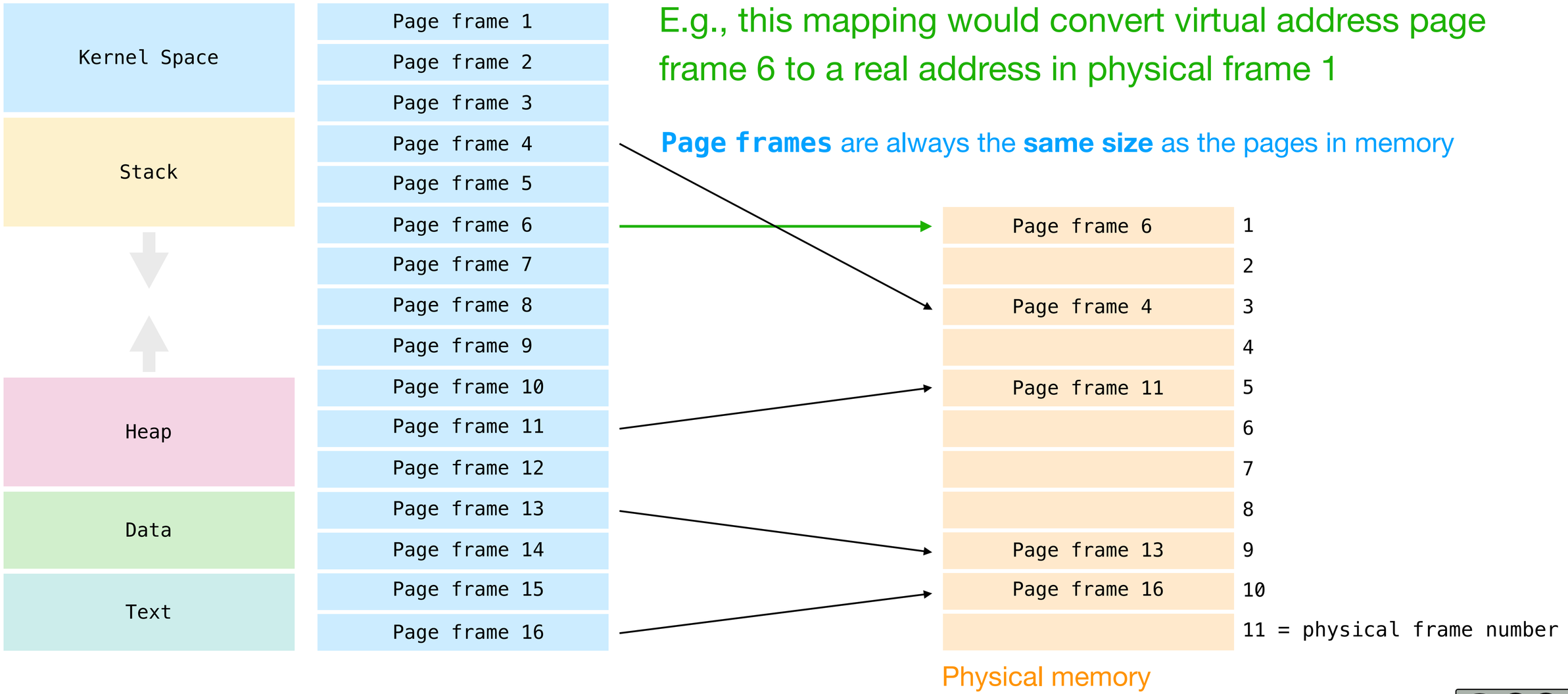
Picture Memory As …



The **idea** behind virtual memory is that **physical memory** is divided **into fixed size pages** (typically 4096 bytes)

`Page frames` are always the **same size** as the pages in memory

Picture Memory As …



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

The virtual address breakdown

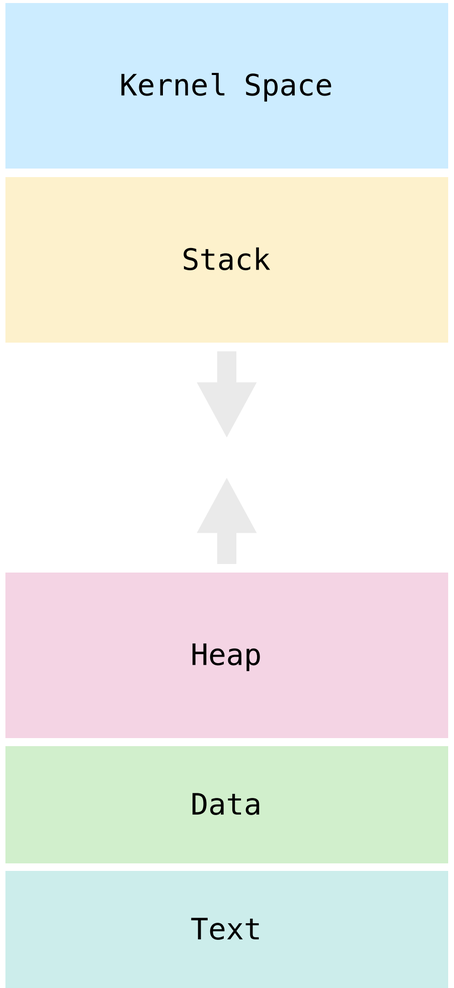
A virtual address is divided into:

- **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]
```

This assuming each **page = 4KB**, lower 12 bits is the **offset**

Picture Memory As …



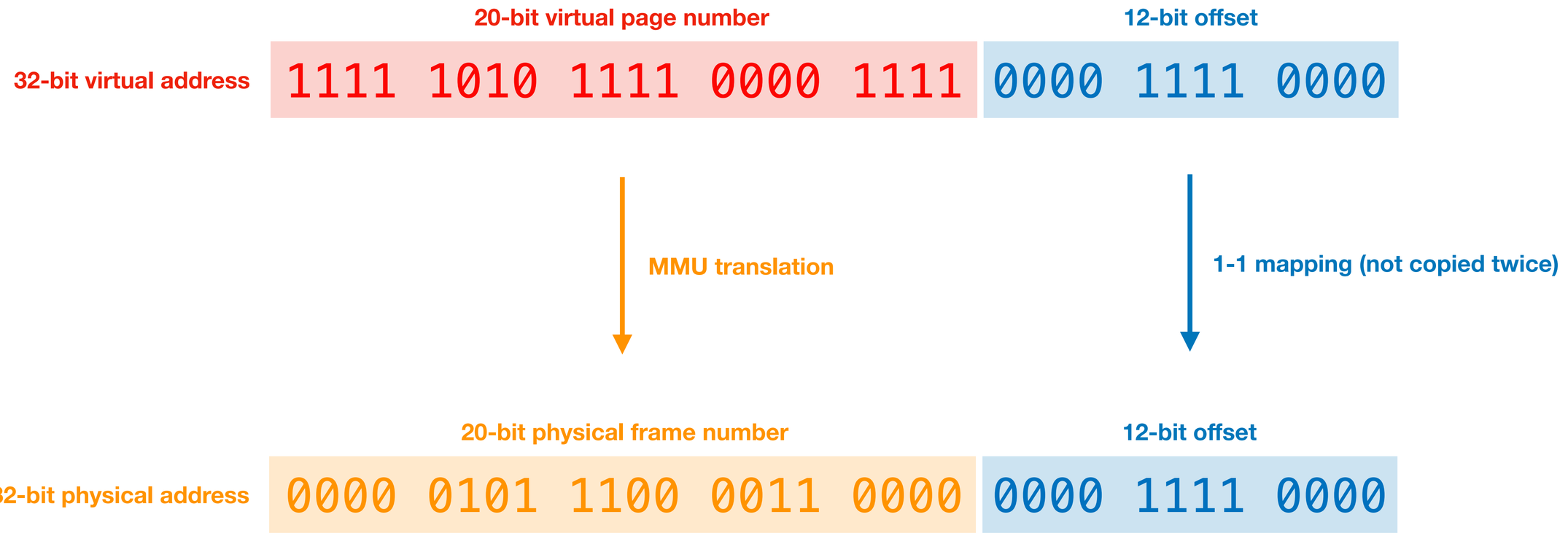
Page frame 1	0xFFFFF000 = page starting address
Page frame 2	0xFFFFE000
Page frame 3	0xFFFFD000
Page frame 4	0xFFFFC000
Page frame 5	0xFFFFB000
Page frame 6	0xFFFFA000 = virtual address = [page number][offset]
Page frame 7	…
Page frame 8	…
Page frame 9	…
Page frame 10	0x00006000
Page frame 11	0x00005000
Page frame 12	0x00004000
Page frame 13	0x00003000
Page frame 14	0x00002000
Page frame 15	0x00001000
Page frame 16	0x00000000

The values stored in **page frame 6** have addresses:

0xFFFFAxxx

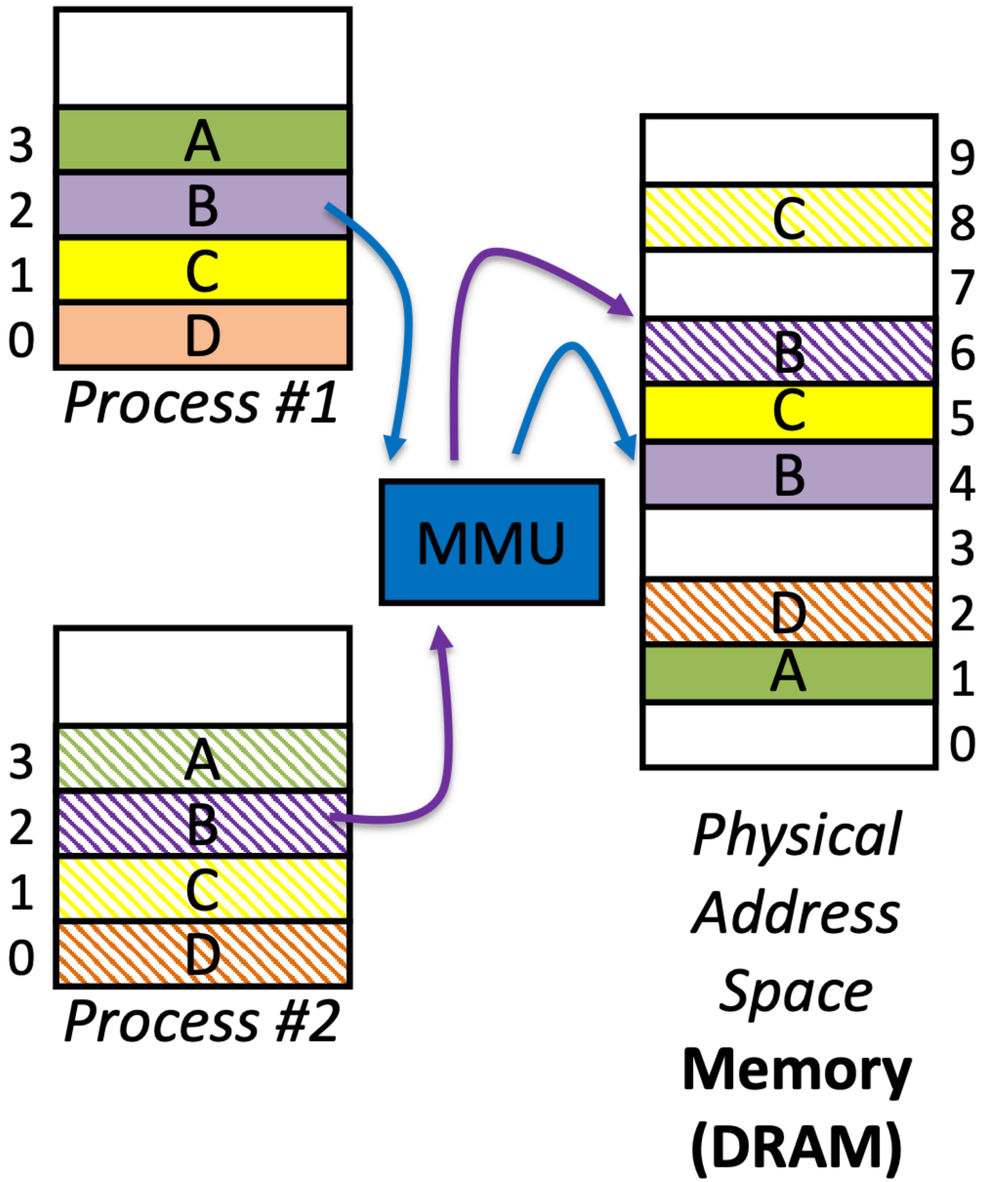
This assuming each **page = 4KB**, lower 12 bits is the **offset**

Address Translation



This assuming each **page = 4KB**, lower 12 bits is the **offset**

Address Translator (MMU)



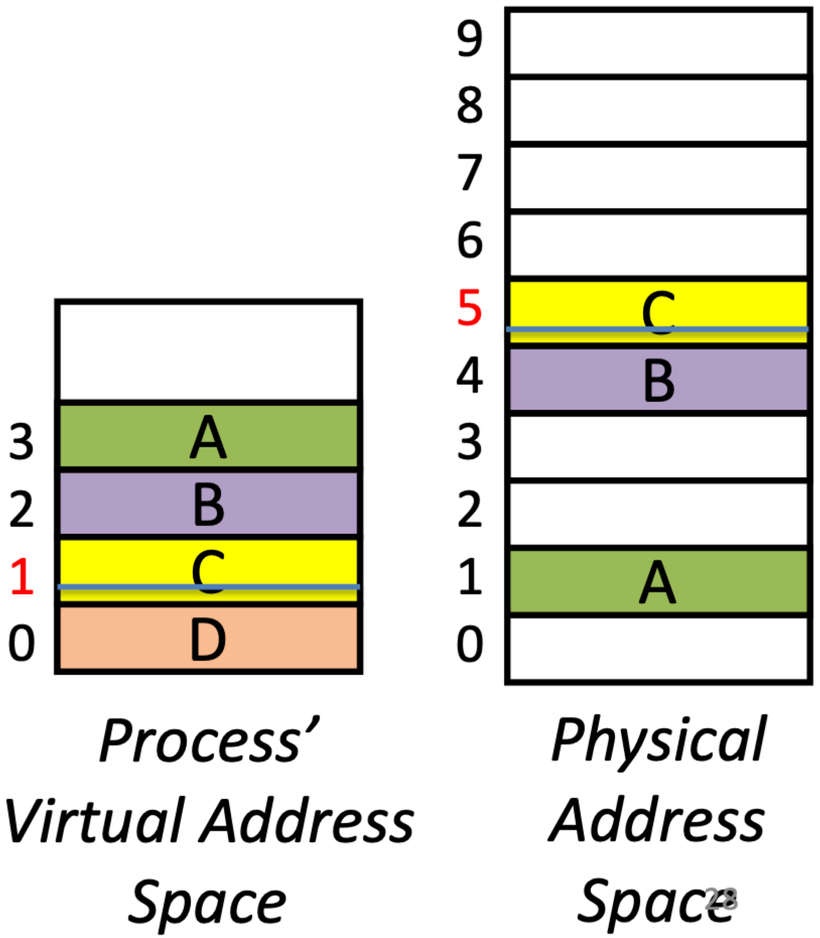
- Processes use virtual memory
- DRAM uses physical memory

- **Memory Management Unit (MMU)**
  - It's hardware!
  - It translates virtual addresses to physical addresses on the fly

Address Translation: in Page Table

- OS is responsible for managing the page tables, which the MMU uses to translate virtual addresses to physical addresses

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



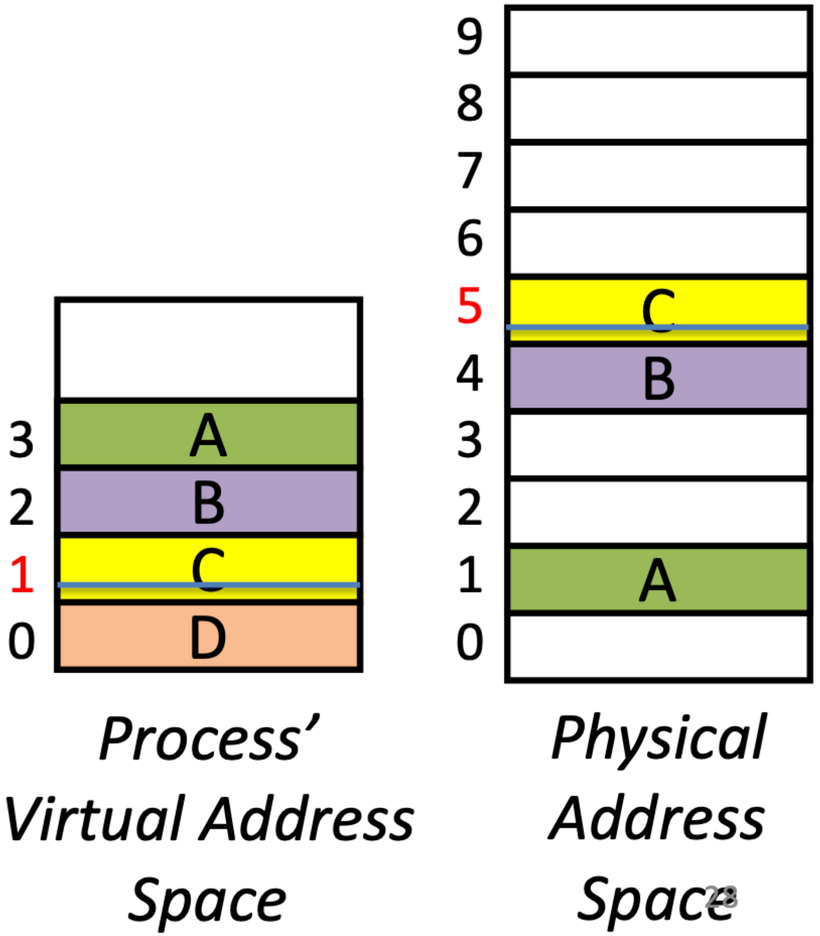
Address Translation: in Page Table

- OS is responsible for managing the page tables, which the MMU uses to translate virtual addresses to physical addresses

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

- **Remember:**
  - That any address 0x00001234 is x234 bytes into Page C both **virtual** & **physical**

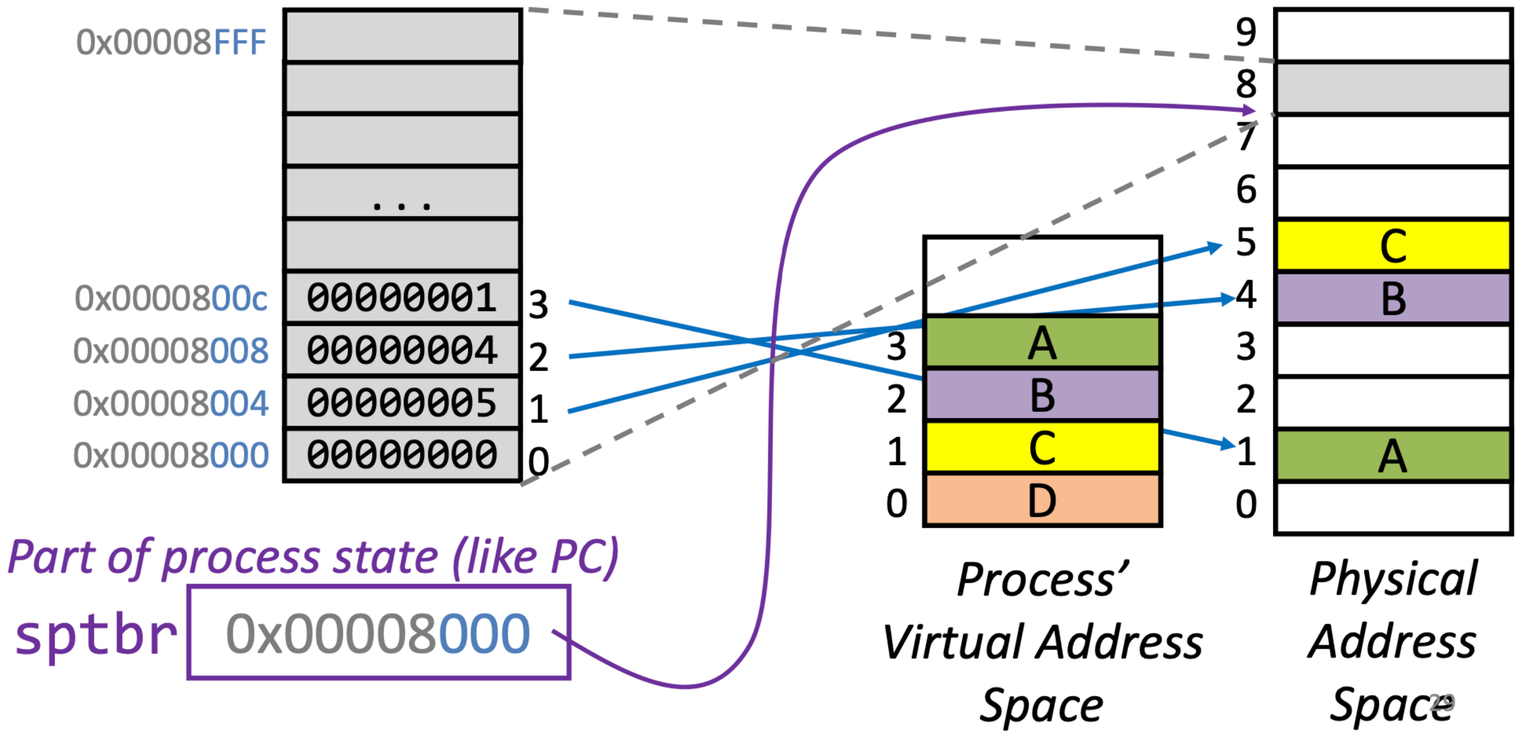
    VP 1 into PP 5



This assuming each **page = 4KB**, lower 12 bits is the **offset**

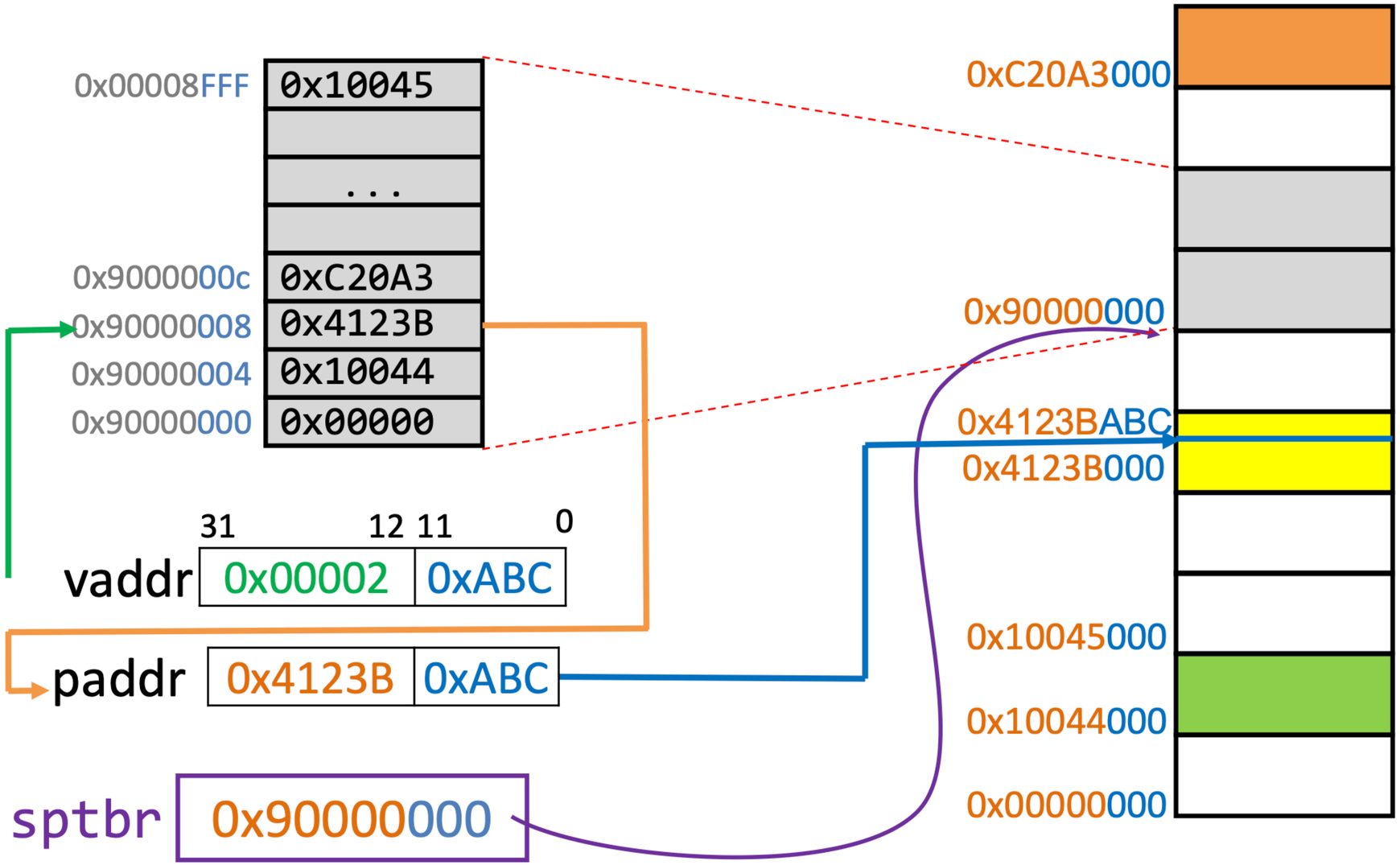
Page Table Basics

- 1 page per process
  - It lives in memory, i.e., *in a page (or more)*
  - The location is stored in **Supervisor Page-Table Base Register**



This assuming each **page = 4KB**, lower 12 bits is the **offset**

Page Table Translation



This assuming each **page = 4KB**, lower 12 bits is the **offset**

Page Table Overhead

- How large is Page Table?
- The virtual address space (for each process):
  - Given: total virtual memory: 2^{32} bytes = 4GB
  - Given: page size: 2^{12} bytes = 4KB
    - # entries in PageTable?
    - number of pages = virtual memory / page size
    - number of pages = 2^{32} / 2^{12} = 2^{20} = 1,048, 576 pages ~ 1 million pages

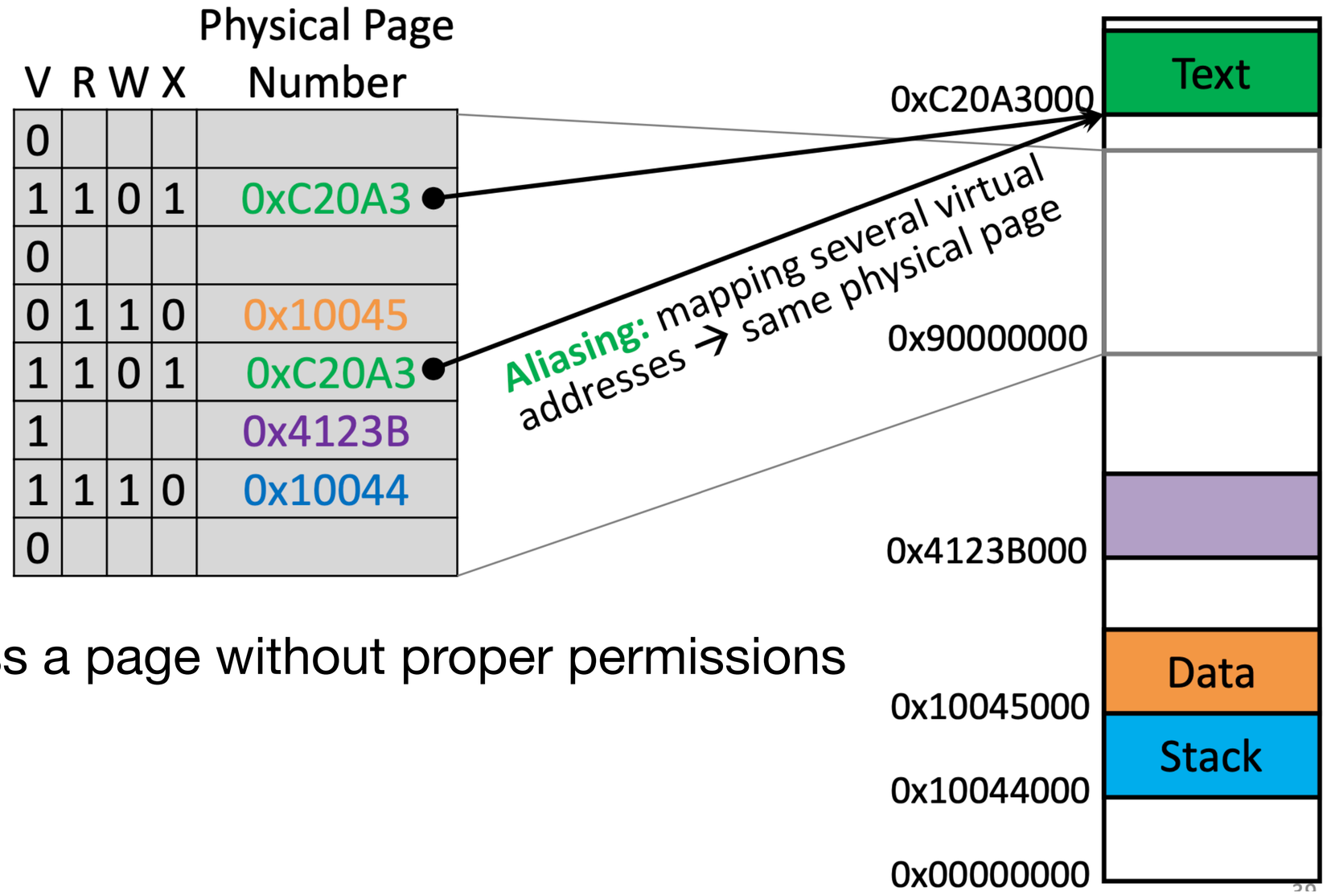
Page Table Overhead

- How large is Page Table?
- The 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? (in bytes)
    - A page table entry (PTE) usually stores the **PFN** plus some metadata (valid bit, protection bits, etc.)
    - Typically, we use 4 bytes (32 bits) per PTE (common for 32-bit physical addresses)
    - Page Table size = $2^{20} \times 4$ bytes = 4×2^{20} bytes = 4 MB

There's More!

- Page Table Entry won't be just an integer
- Meta-Data
  - Valid Bits
    - What PPN means "not mapped"?
    - **First:** not all virtual pages will be in physical memory
    - **Later:** might not have enough physical memory to map all virtual page
  - Page Permissions (e.g., Read/Write permission, Executable or not)

Less Simple Page Table



Process tries to access a page without proper permissions

Page Table Overhead

- How large is Page Table?
- The virtual address space (for each process):
  - Given: total virtual memory: 2^{32} bytes = 4GB
  - Given: page size: 2^{12} bytes = 4KB
    - # entries in PageTable? ~ 1 million pages
    - size of PageTable? (in bytes) ~ 4 MB
- The physical address space:
  - Total physical memory: 2^{29} bytes = 512MB
    - Overhead for 10 processes?

- number of physical frames = physical memory / page size = 2^{29} / 2^{12} = 2^{17} = 131, 072 frames
- pages per process = 2^{32} / 2^{12} = 2^{20} pages
- page table size per process = 4 MB
- page table **overhead** size= 4 MB x 10 = 40 MB
- fraction overhead = 40 / 512 MB = **7.8%**

Paging

- But what if process requirements > physical memory?
  - Then, *virtual starts earning its name*

- E.g., a process needs **1 GB**, but physical memory is only **512 MB**
- Can’t fit the entire virtual address space in DRAM at once
- **Paging** allows this to work by mapping only the pages that are actively used

Paging

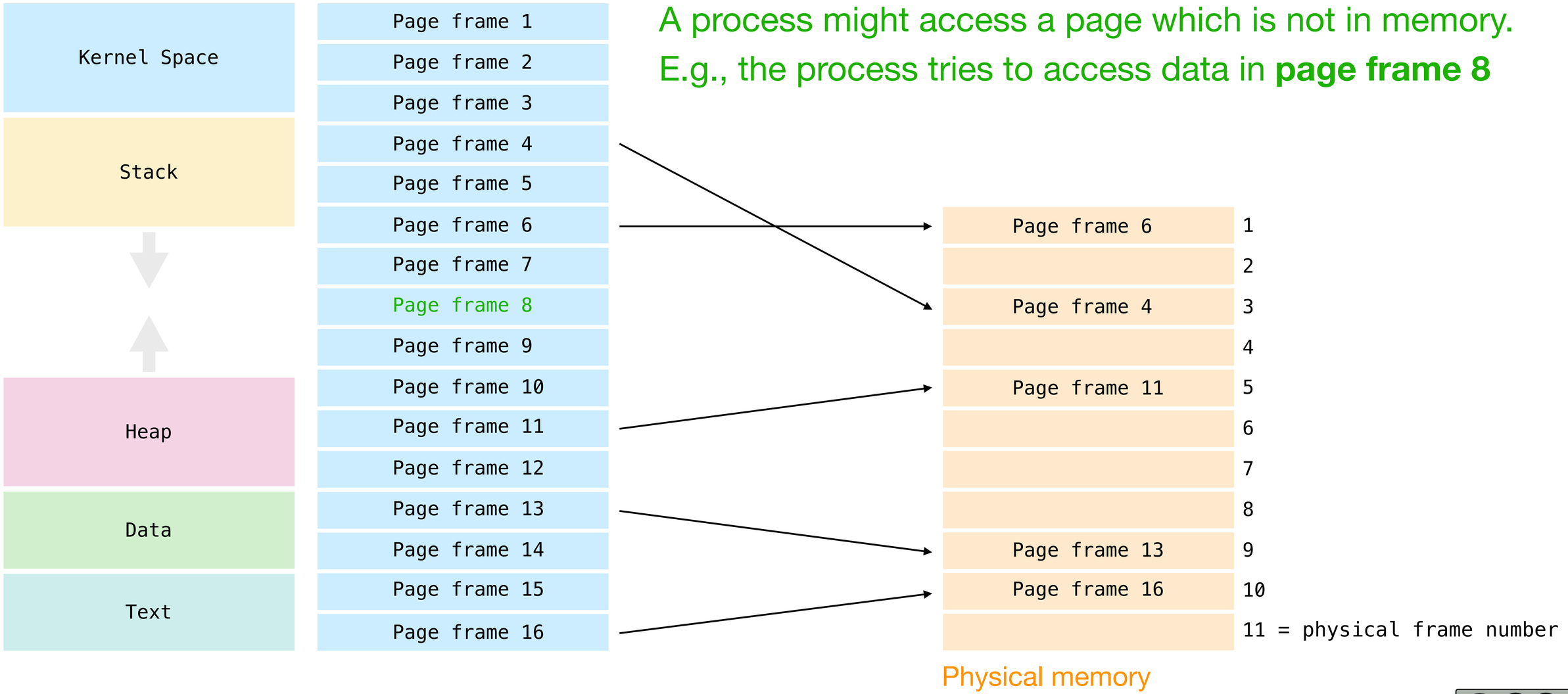
- But what if process requirements > physical memory?
  - Then, *virtual starts earning its name*

- The main 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
  - If a process accesses a page not in memory, a **page fault** occurs

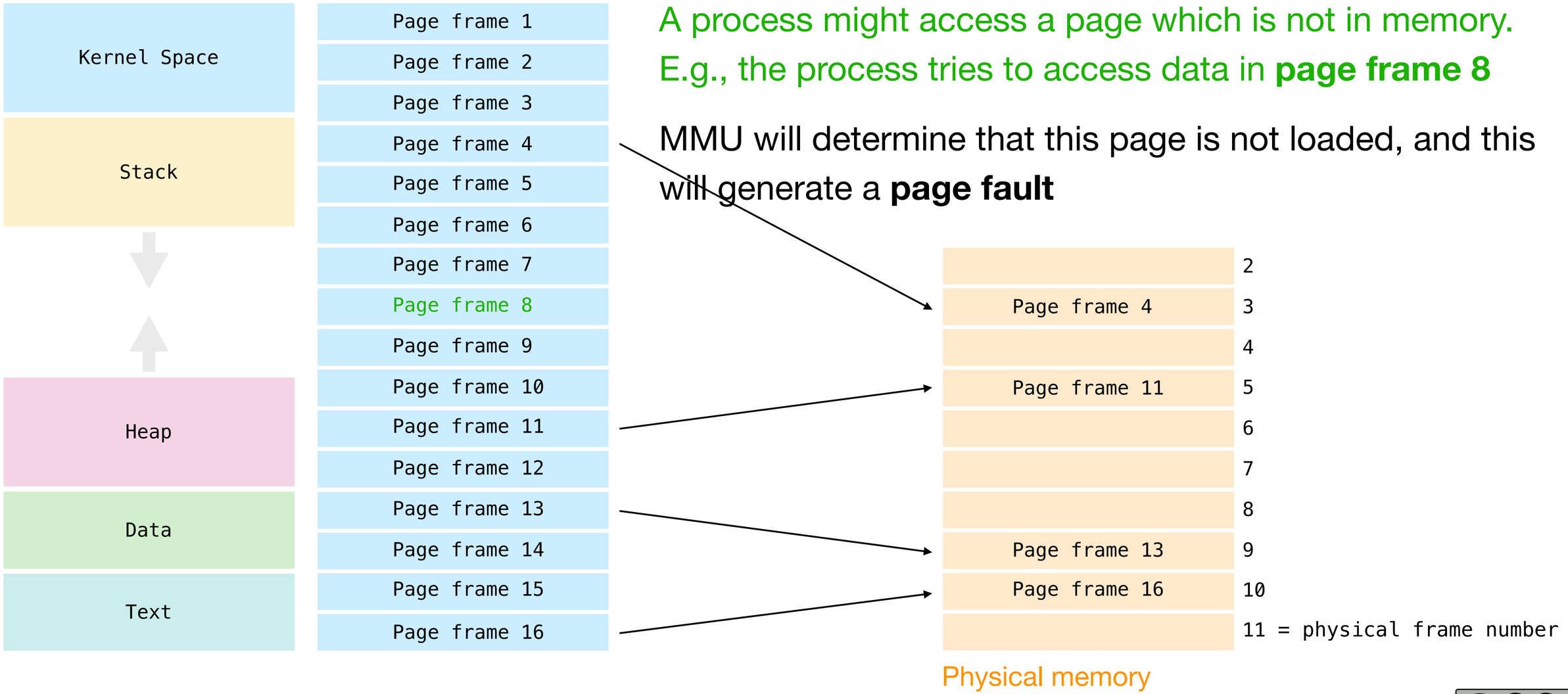
Paging

- But what if process requirements > physical memory?
  - Then, *virtual starts earning its name*
- The main 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
  - If a process accesses a page not in memory, a **page fault** occurs
- Courtesy of Temporal & Spatial Locality (again!)

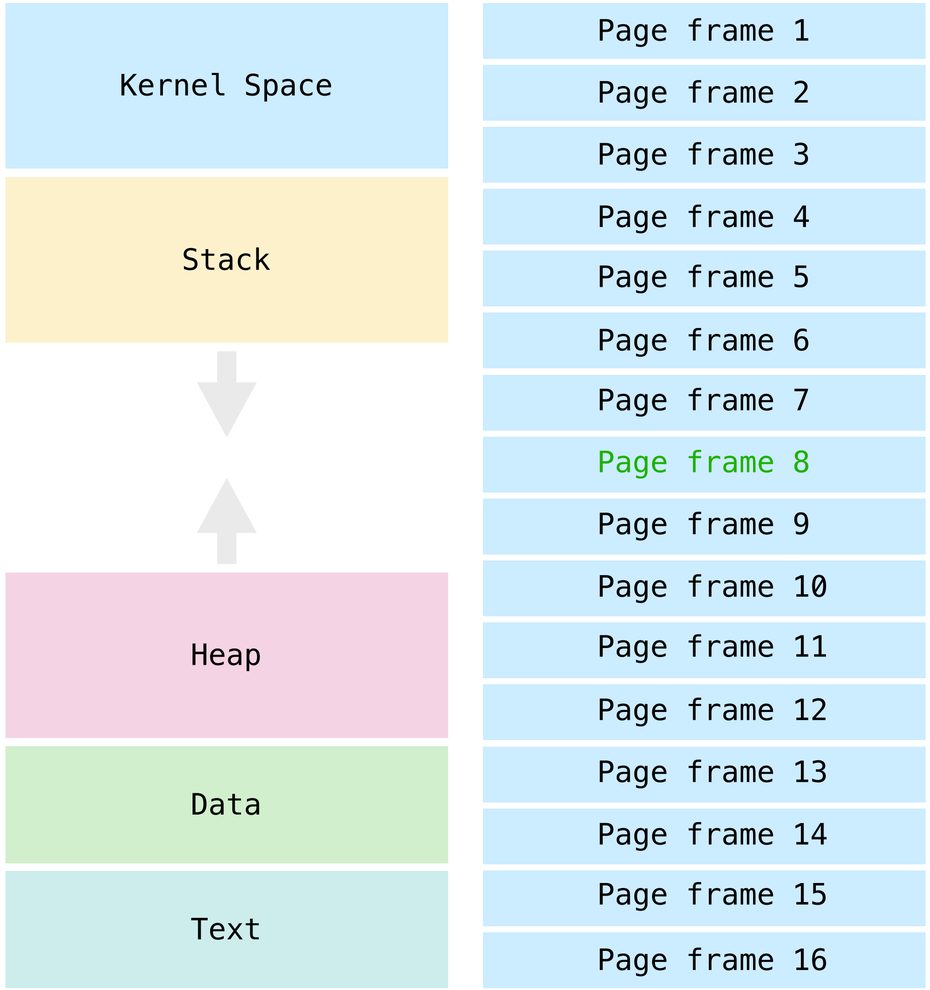
Picture Memory As …



Picture Memory As …



Picture Memory As …

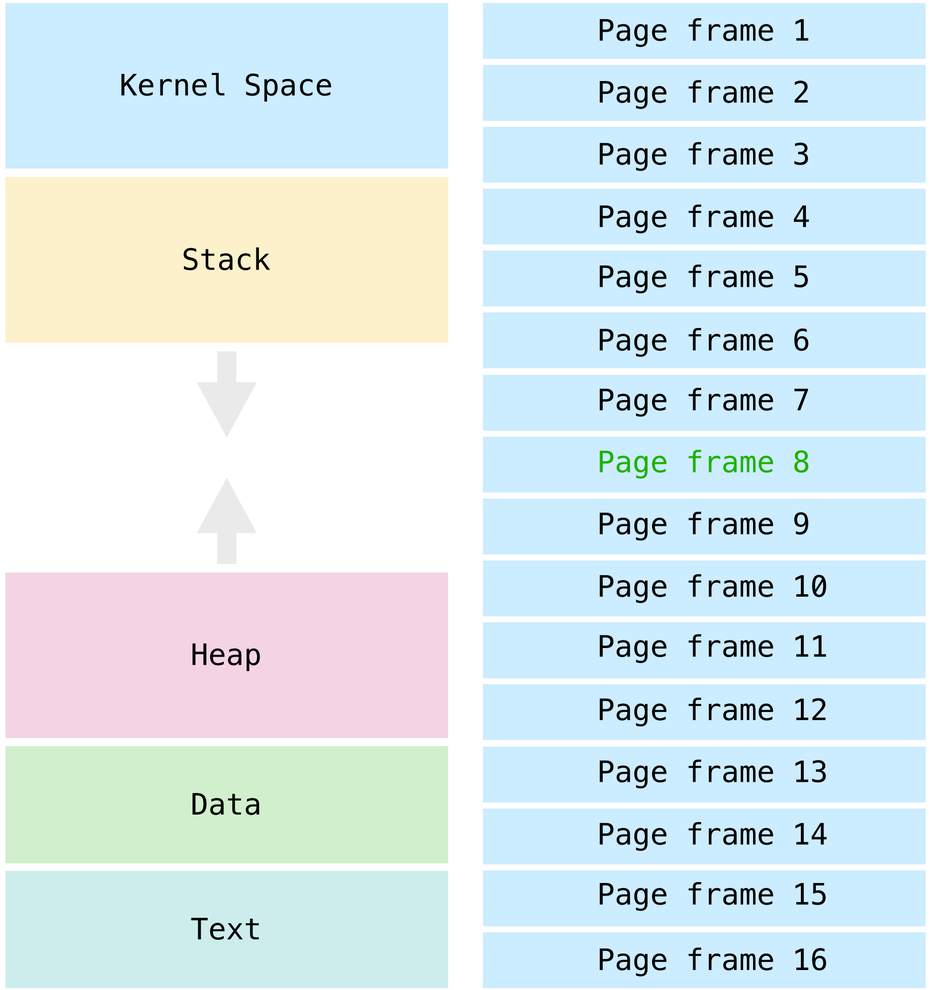


A process might access a page which is not in memory.
E.g., the process tries to access data in **page frame 8**

MMU will determine that this page is not loaded, and this will generate a **page fault**

A page fault is an **interrupt**

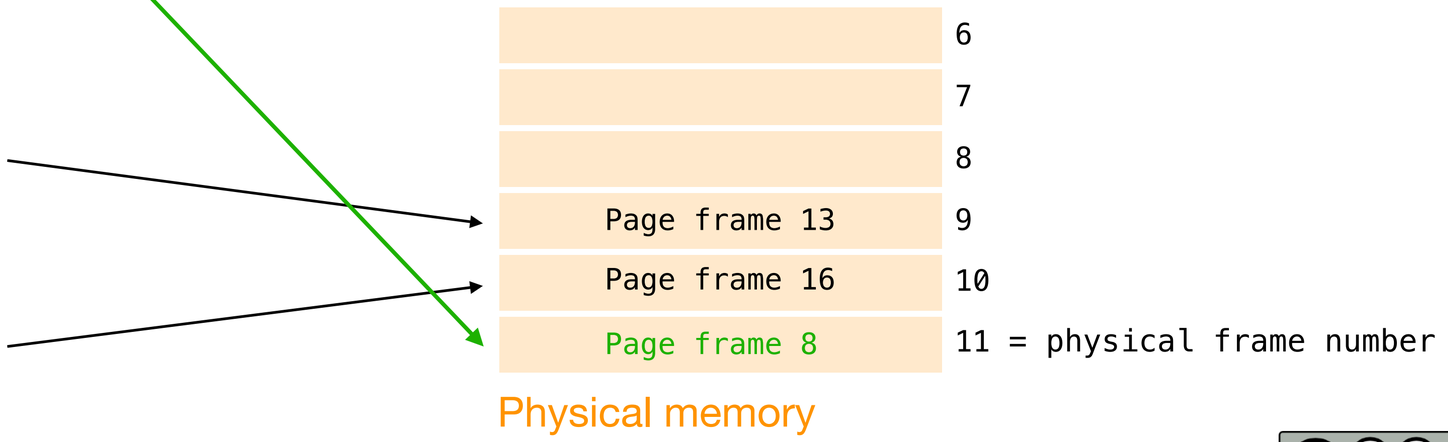
Picture Memory As …



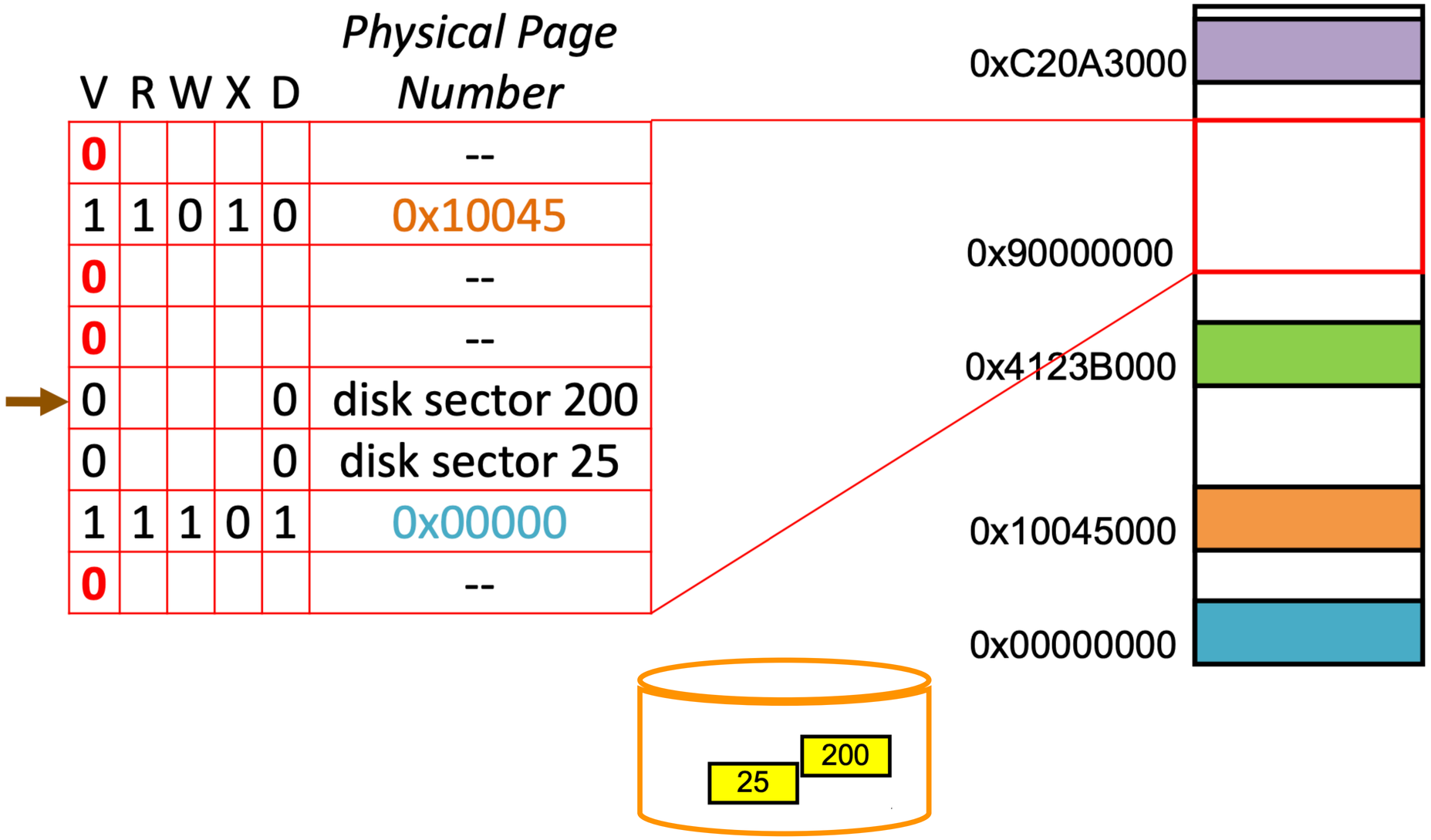
A process might access a page which is not in memory.
E.g., the process tries to access data in **page frame 8**

MMU will determine that this page is not loaded, and this will generate a **page fault**

The **OS handler** for a page fault locates the needed page frame on the disk, copies it to a page in memory, and updates the page table



Paging



Page Fault Flow

1. CPU accesses a virtual address → **MMU** looks at page table → ***valid* bit = 0**
2. **MMU** signals a **page fault** to the **OS**
3. **OS** chooses a physical frame to load the page from disk (may evict an old page)
4. **OS** updates the page table → ***valid* bit = 1**, PPN points to the physical frame
5. CPU retries the instruction → now it succeeds!

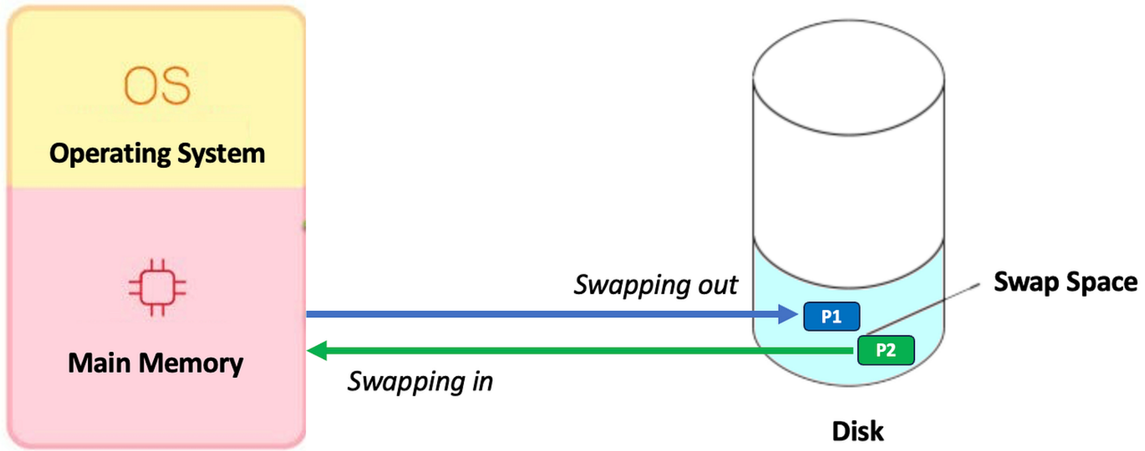
Page Fault Flow

- The valid bit in the Page = 0
  - I.e., the page is **not** in memory
- **OS takes over!**
  - Choose a physical page to replace (tracks page usage)
  - If **dirty**, write to disk
  - Read missing page from disk
    - Takes **long** (~10ms), **OS** schedules another task

Swap Space

A backup area where the OS can temporarily store parts of a process's memory that don't fit in DRAM

- The OS keeps **active pages** in DRAM and moves **inactive pages** to swap when RAM is full
- This way, the total "usable" memory = **DRAM + swap**

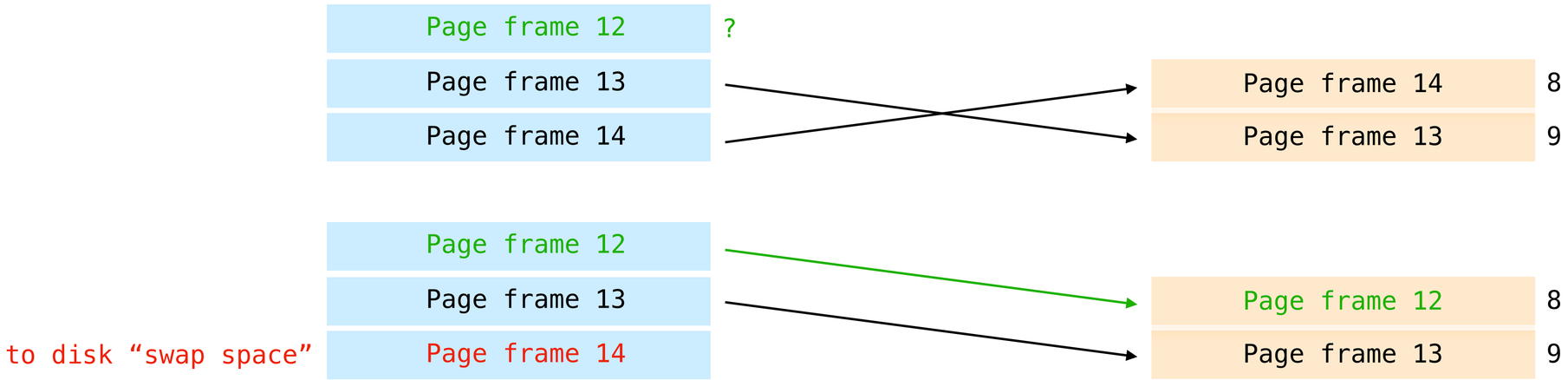


https://medium.com/@ahmedmansouri/the-diminishing-role-of-swap-space-in-cloud-environments-ab2e413c4013

Swap Space

If DRAM is full and a process needs a new page:

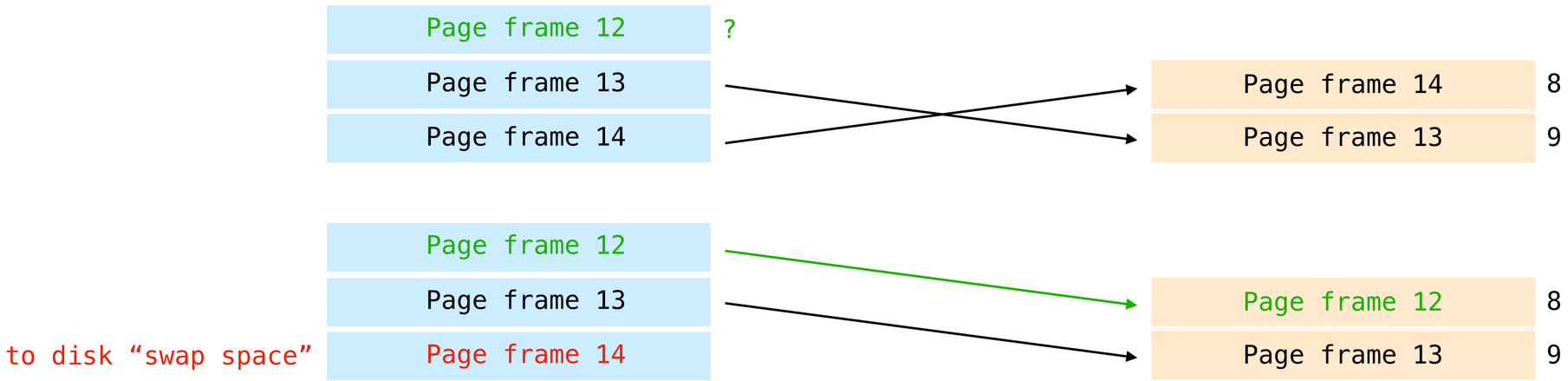
- The OS picks a page in DRAM to evict (LRU, etc.)
- If the page to be evicted has been modified, it's written to **swap**
- Then, the new page is loaded into that freed DRAM frame



Swap Space

Then, the OS updates the **page table** entry for the evicted page:

- It marks the page as not present in memory
- If stores the swap location (disk block) where the page lives



Conclusion

- The need of a map to translate a “fake” virtual address (from process) to a “real” physical address (in memory)

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

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

- The space overhead due to Page Table is significant.
  - Two-level of Page Table significantly reduces overhead **(CS4410)**