Virtual Memory

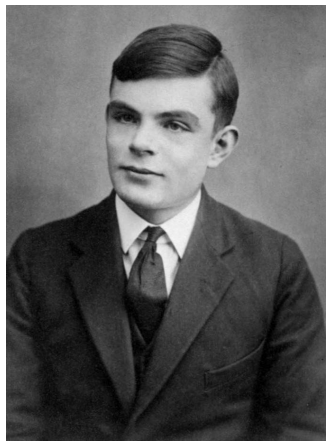
CS 3410

Computer System Organization & Programming

Welcome Back!

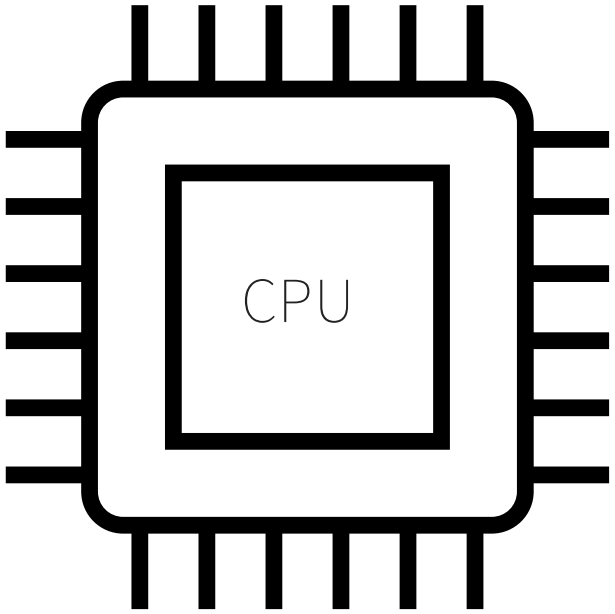
- I hope that you had a great Spring Break!

Big Picture: How to ~~Design~~ Program a Processor

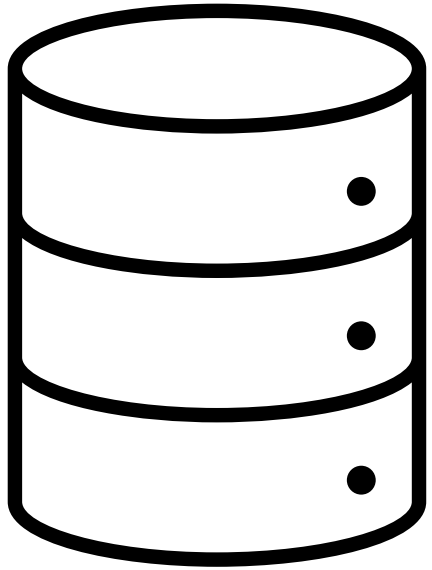


Loads/Stores are very slooooow

Processor

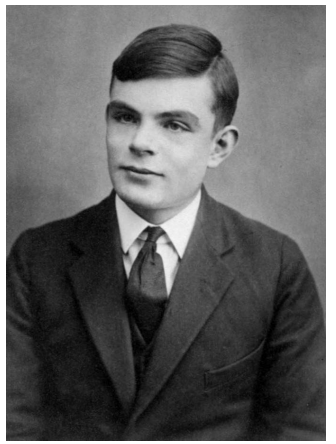


Memory

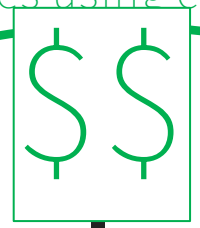


- ✓ Runs code; does computations
- ✗ Doesn't remember anything
- ✗ Can't compute anything
- ✓ Stores data

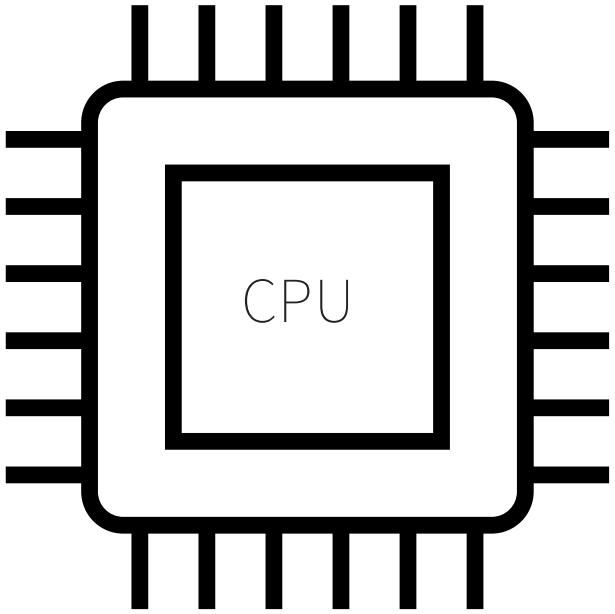
Big Picture: How to ~~Design~~ Program a Processor



Loads/Stores using caches are fast!

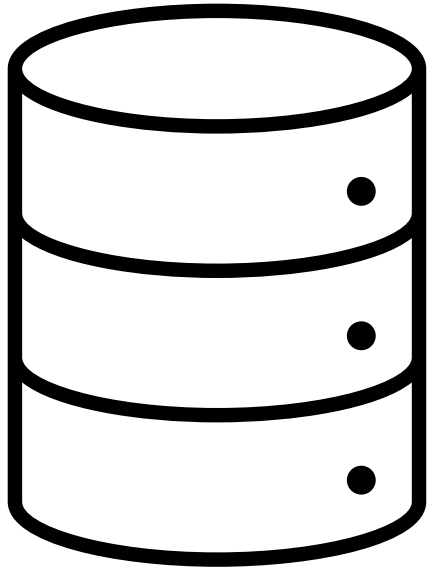


Processor



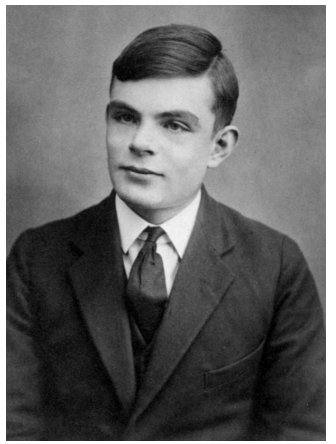
- ✓ Runs code; does computations
- ✗ Doesn't remember anything

Memory

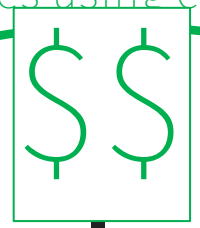


- ✗ Can't compute anything
- ✓ Stores data

Big Picture: How to ~~Design~~ Program a Processor

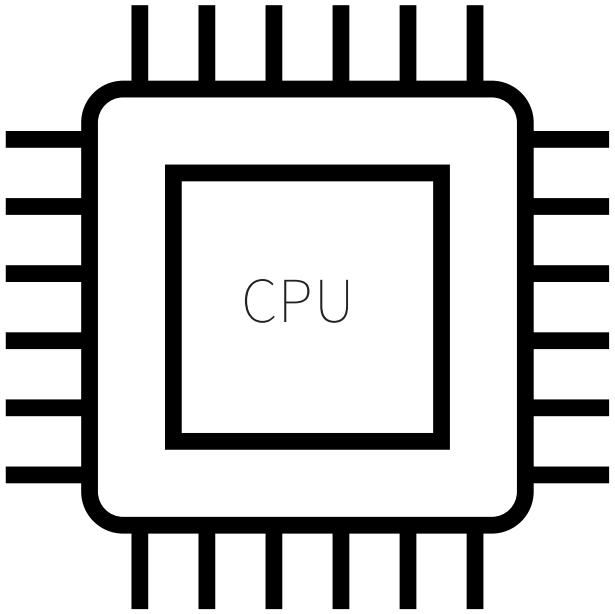


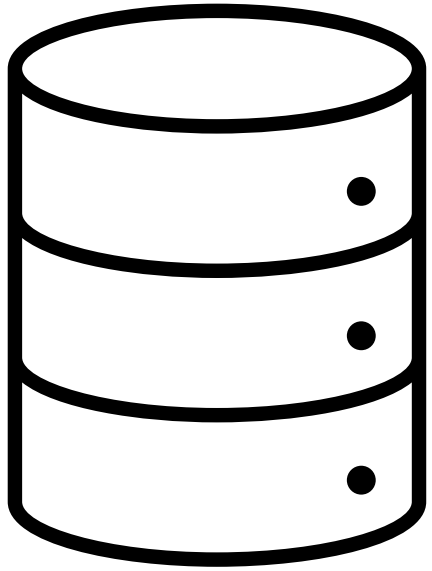
Loads/Stores using caches are fast!



Processor

Memory





Next, how do we run more than one process?

- ✓ Runs code; does computations
- ✗ Doesn't remember anything
- ✗ Can't compute anything
- ✓ Stores data

Virtual Memory Agenda

What do we run multiple processes together?

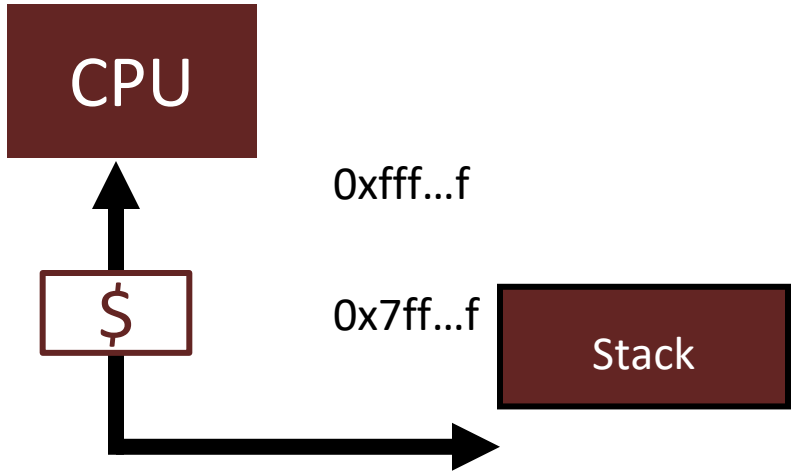
How does Virtual memory Work?

Big Picture: Multiple Processes

How to run multiple processes?

- *Time-multiplex* a single CPU core (multi-tasking)
  - Web browser, zoom, 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…

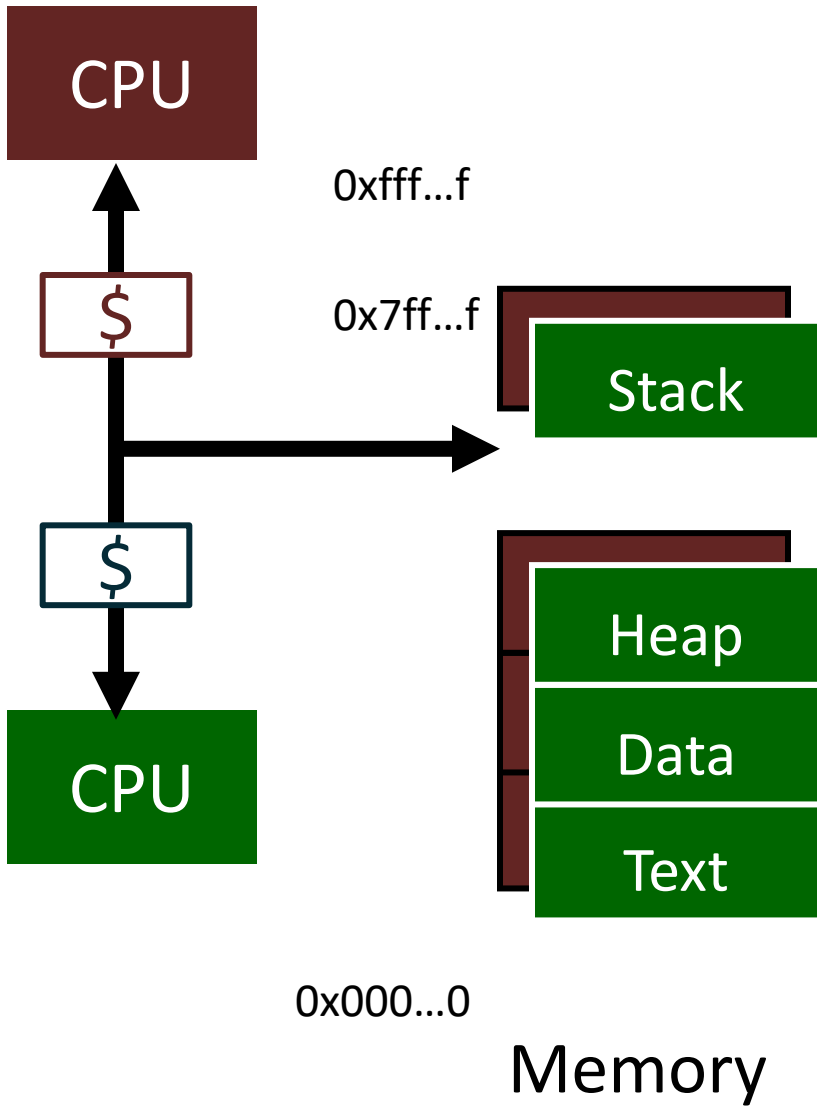
… routed through caches

… to main memory

Simple, fast, but…

Multiple Processes

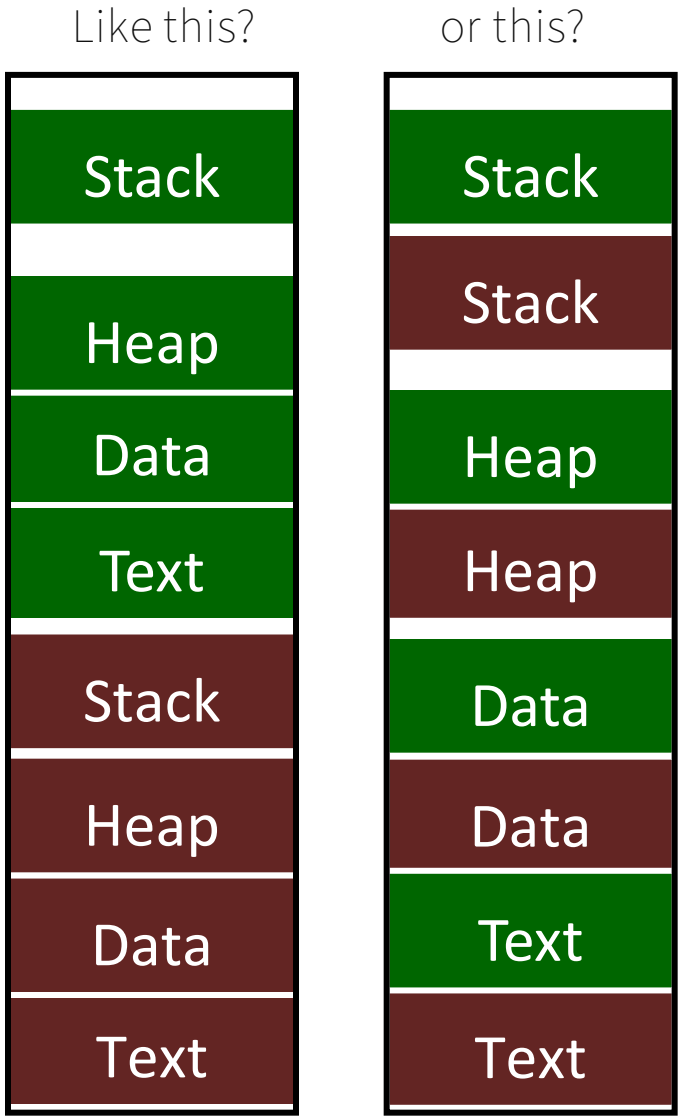
Q: What happens when another program is executed concurrently on another processor?



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?
- …

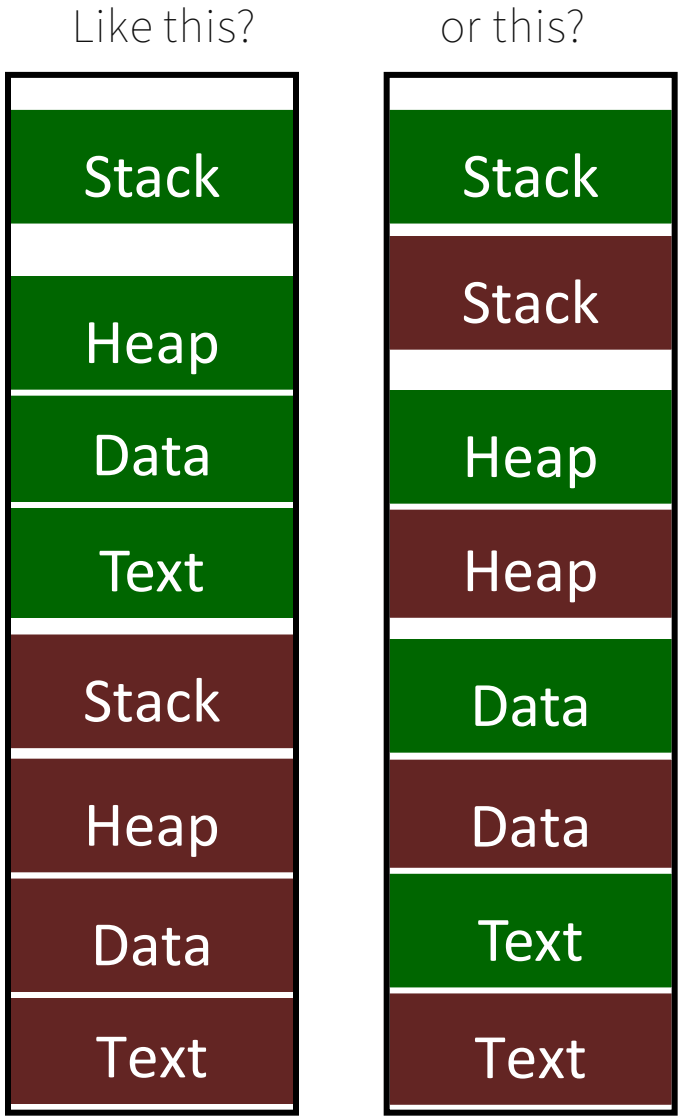


Can we relocate second program?

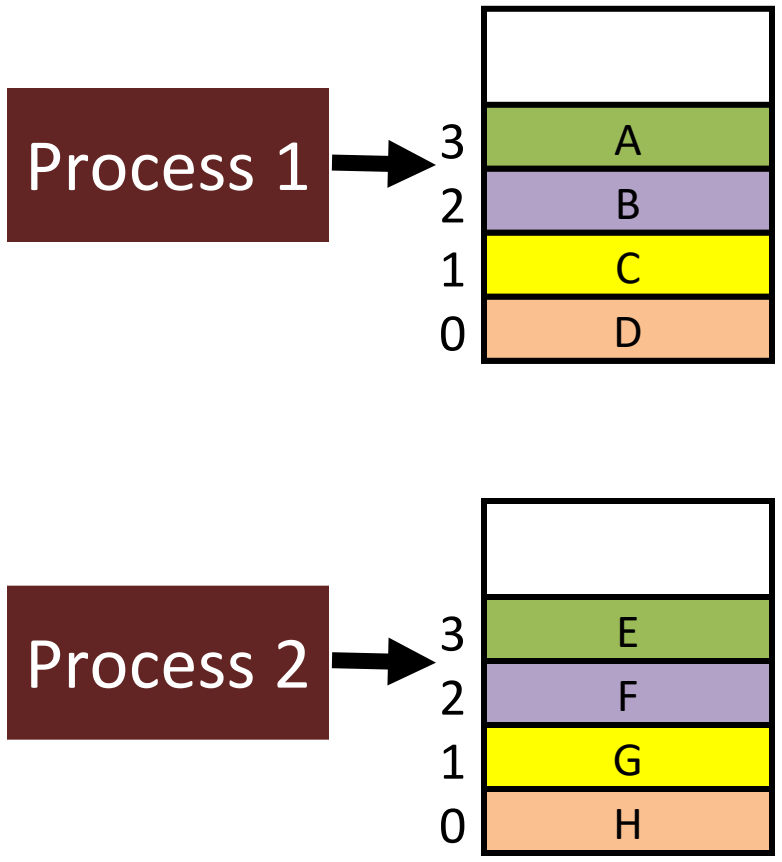
Yes, but… how?

- 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)

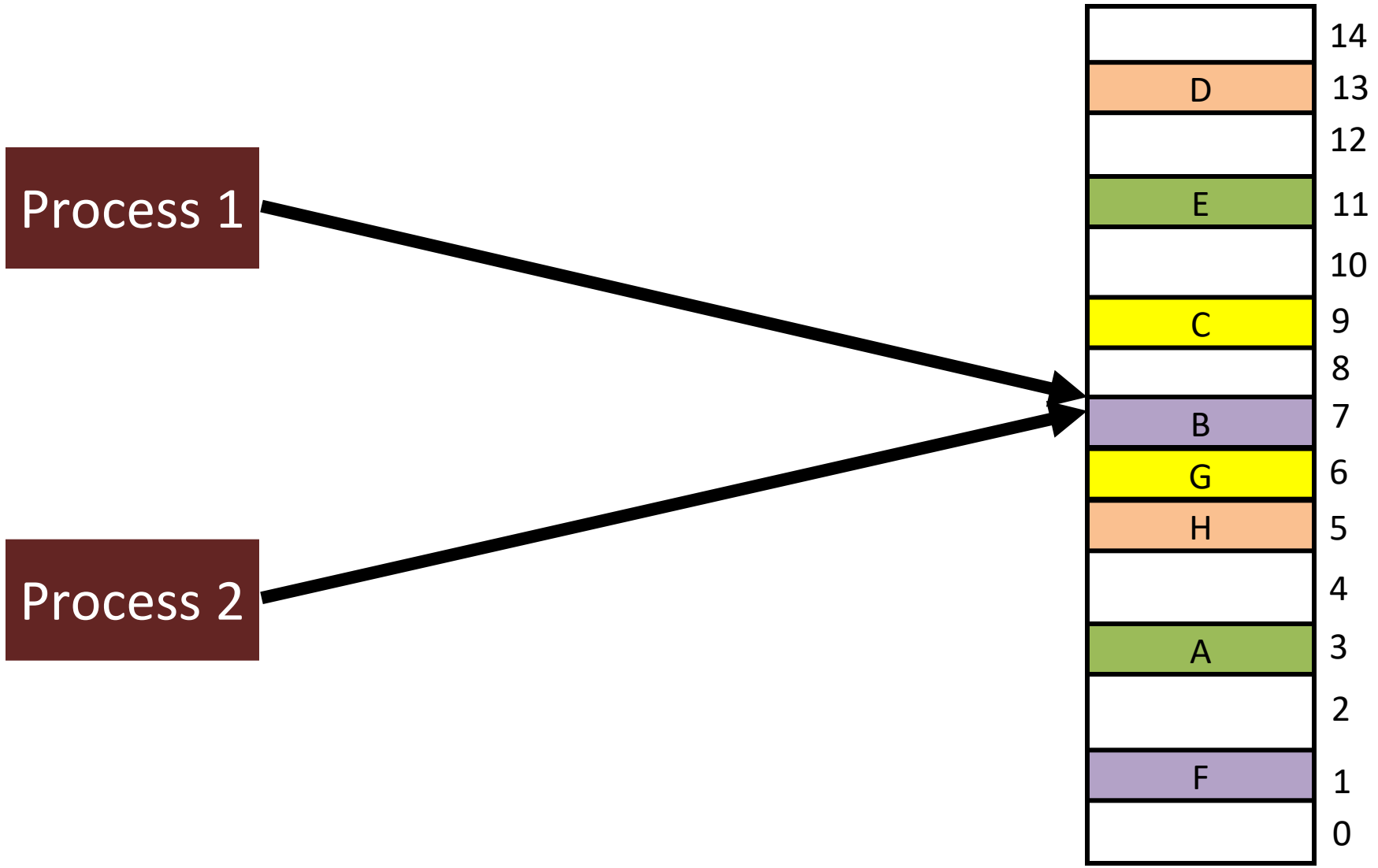


Big Picture: (Virtual) Memory

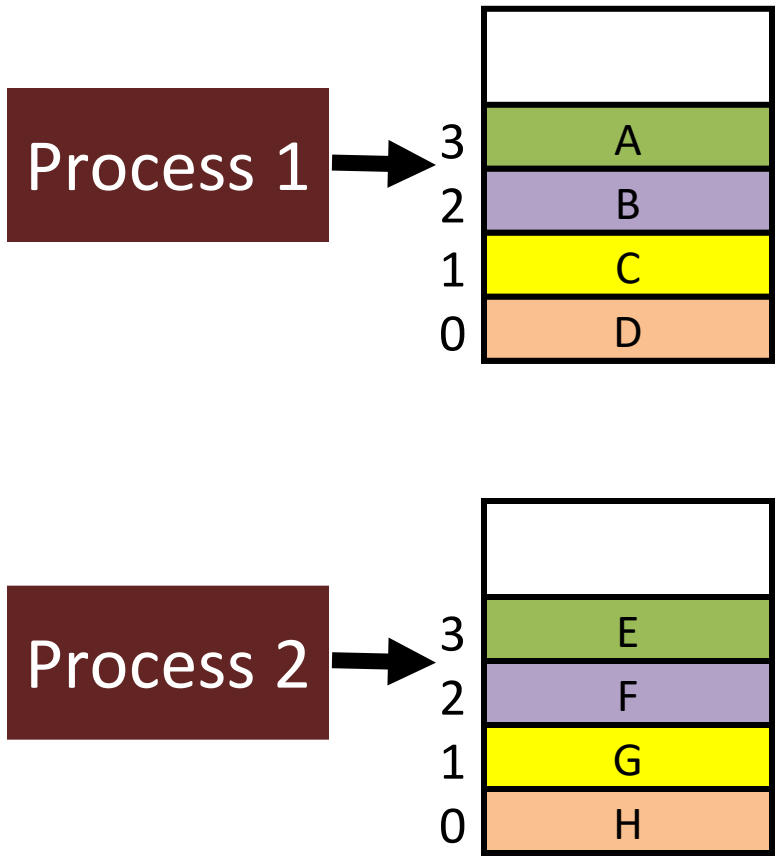


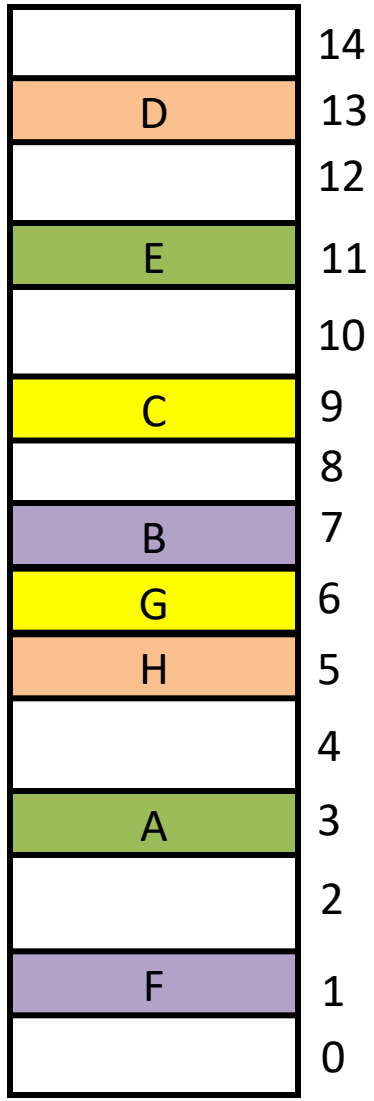
Give each process an illusion that it has exclusive access to entire main memory

But In Reality…



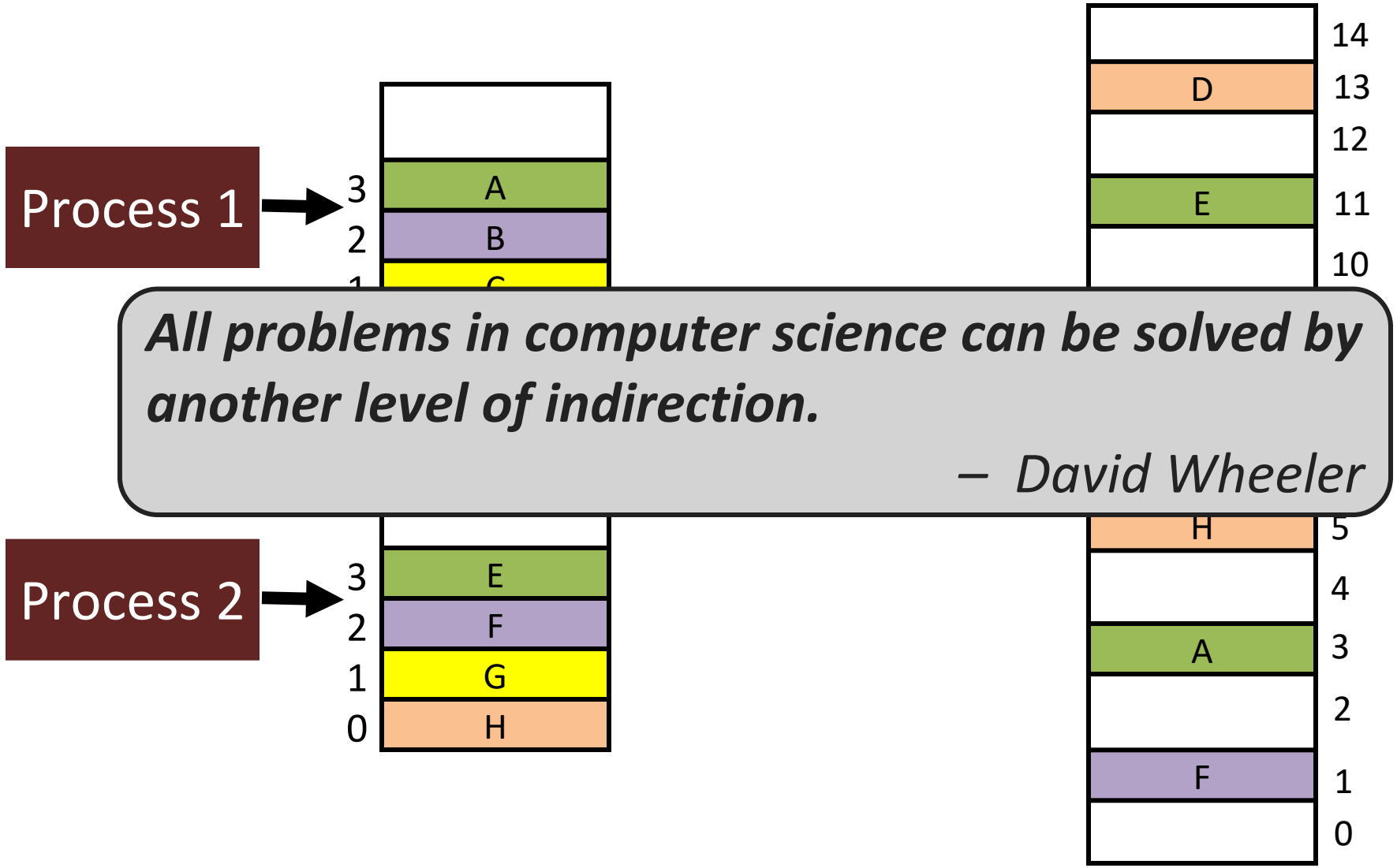
How do we create the illusion?





Physical Memory

How do we create the illusion?

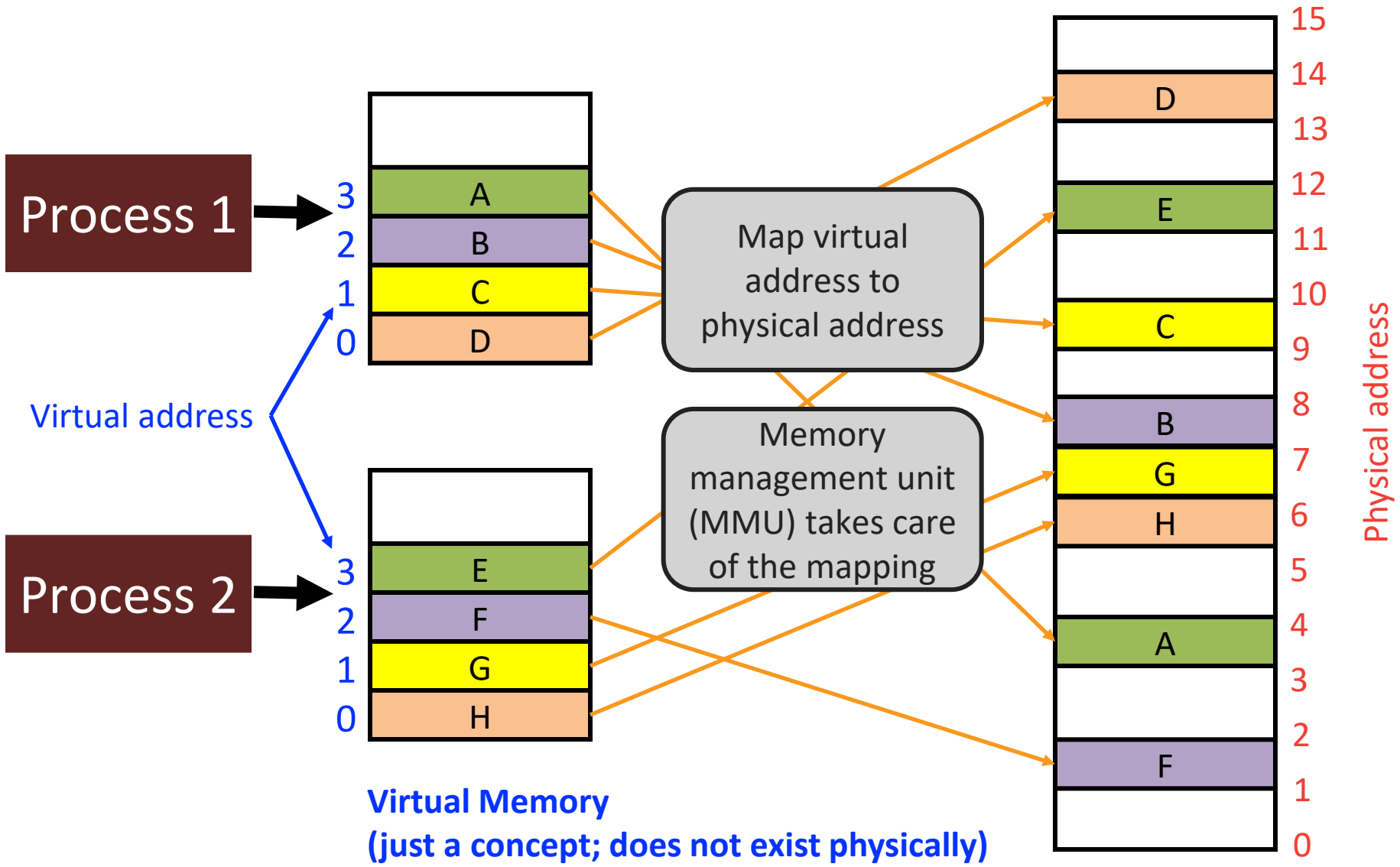


All problems in computer science can be solved by another level of indirection.

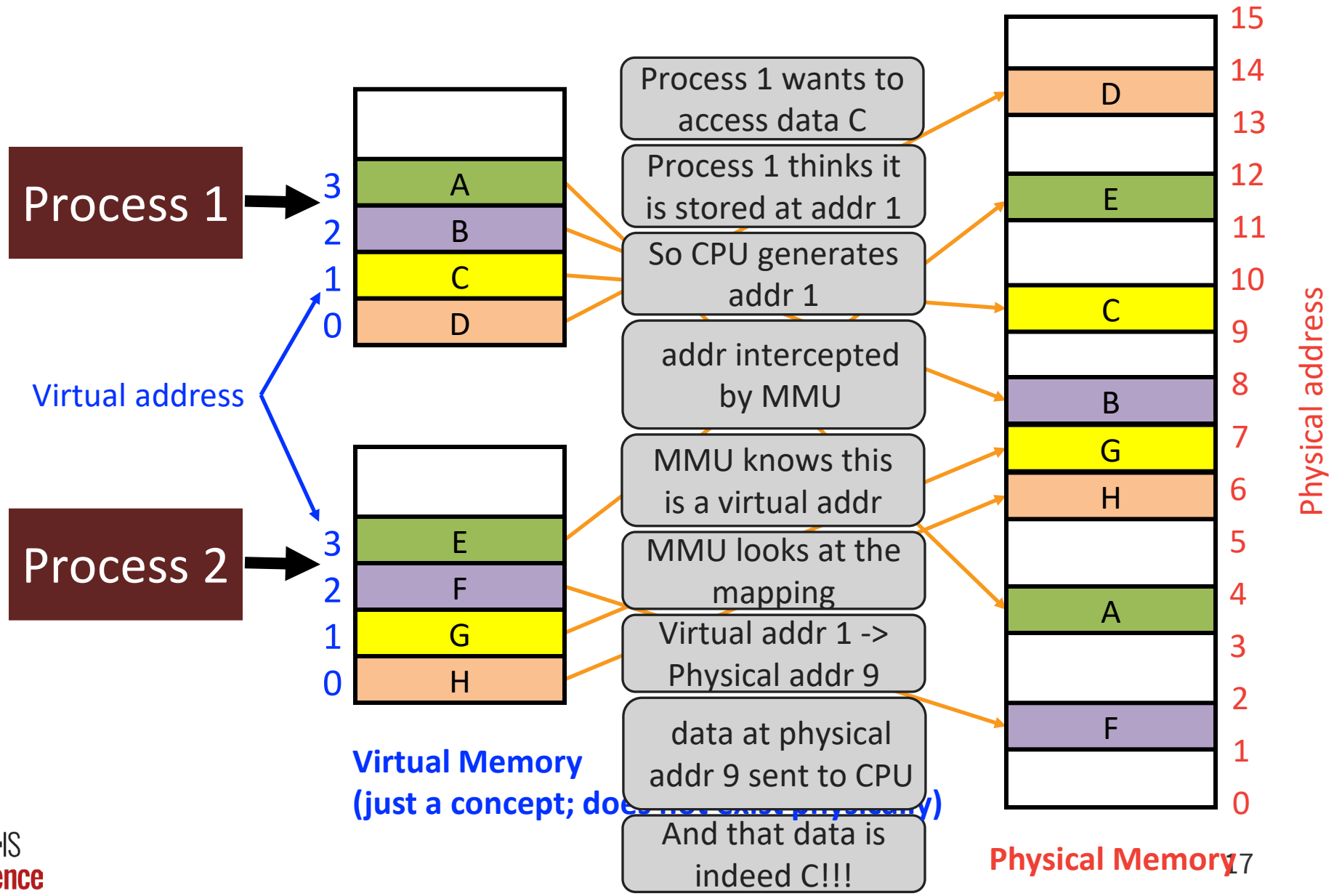
– David Wheeler

Physical Memory

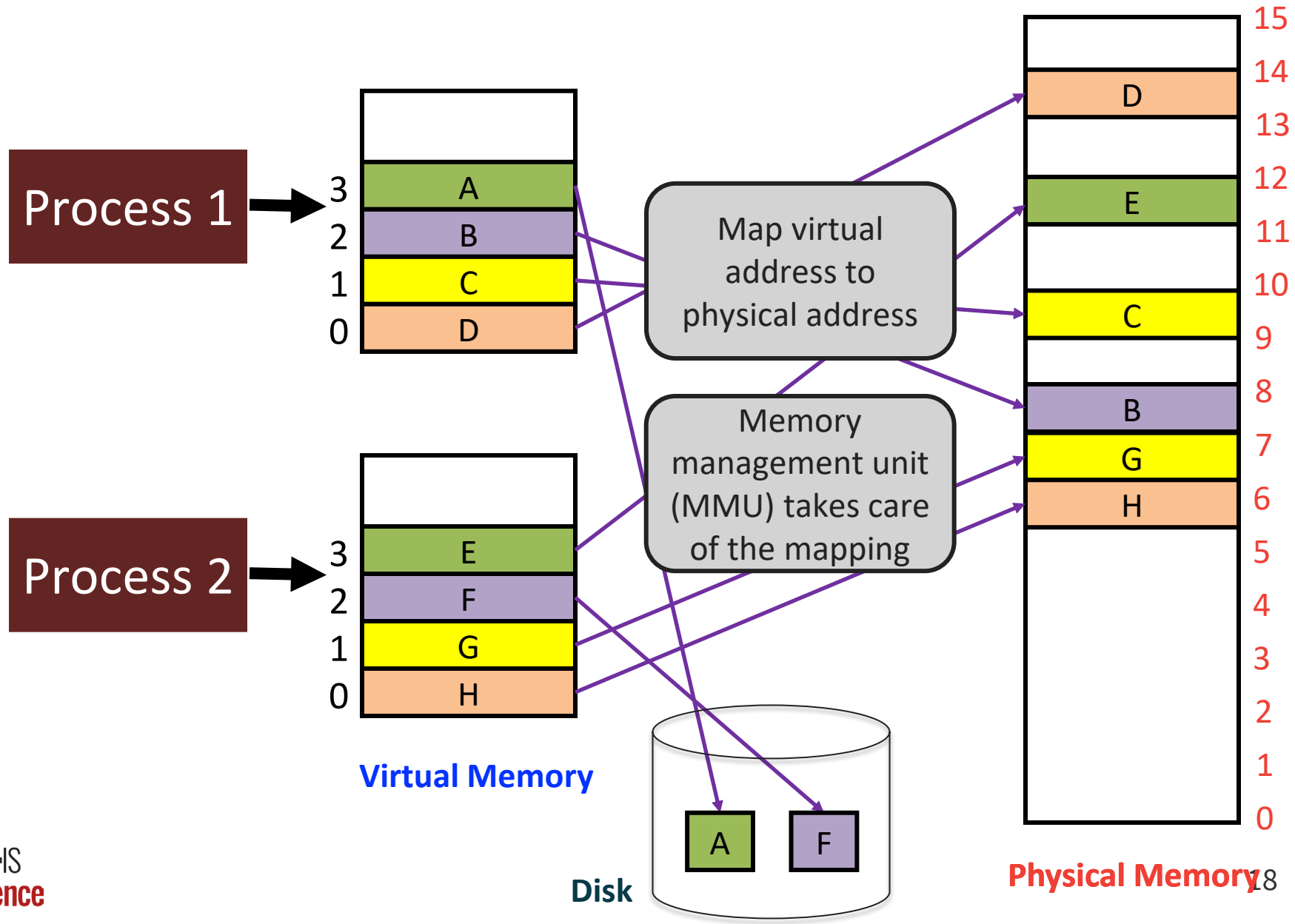
How do we create the illusion?



How do we create the illusion?

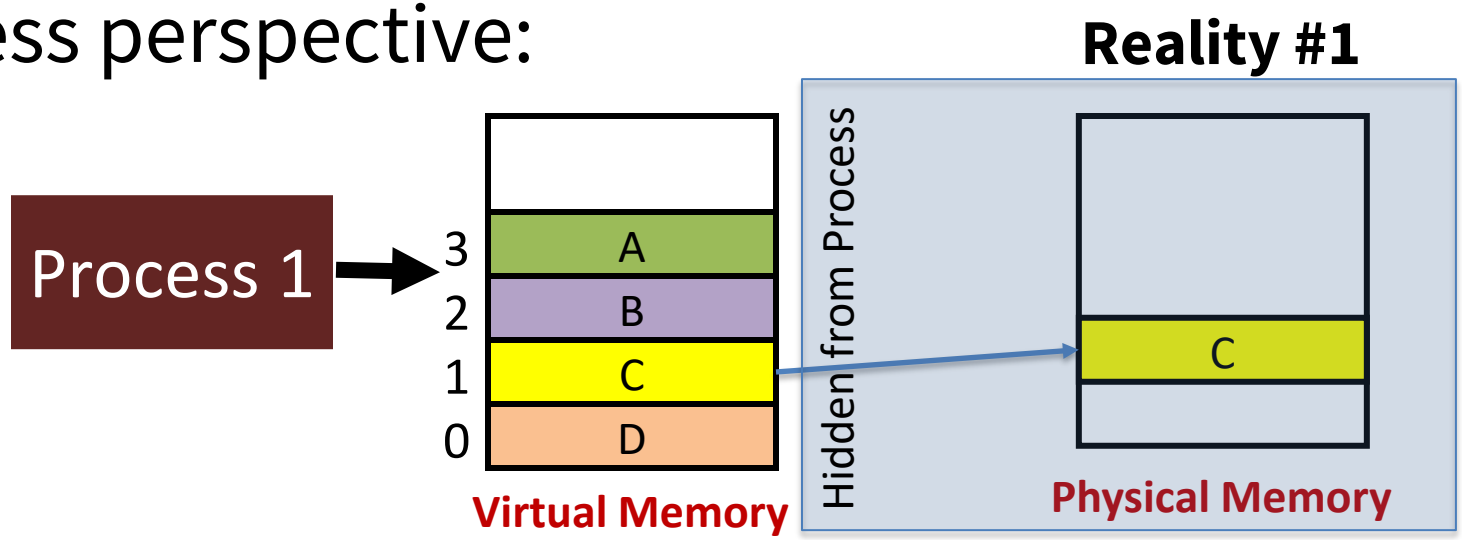


How do we create the illusion?



Big Picture: (Virtual) Memory

Process perspective:

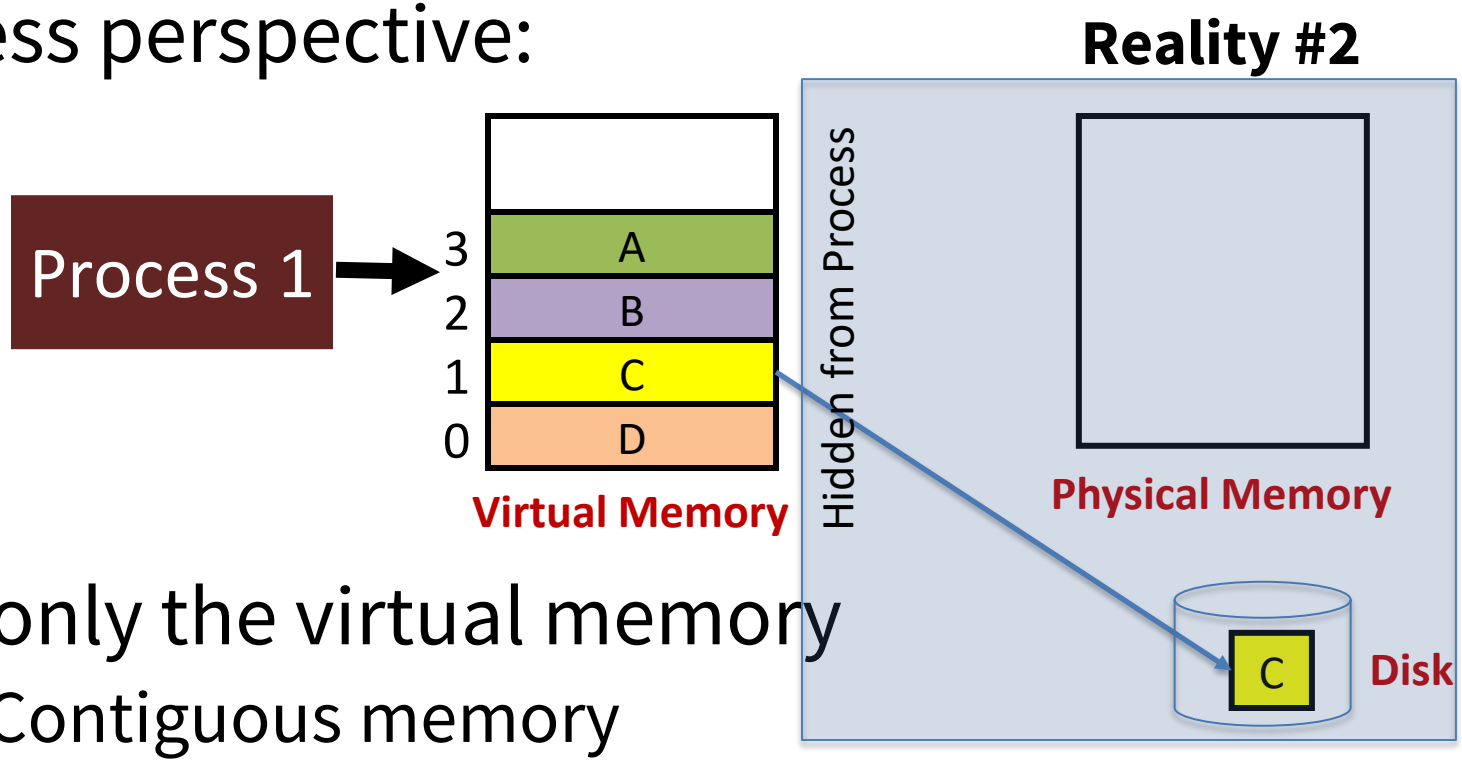


sees only the virtual memory

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

Big Picture: (Virtual) Memory

Process perspective:



sees only the virtual memory

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

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

map

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

And more to come…

Next Goal

- 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?

Virtual Memory Agenda

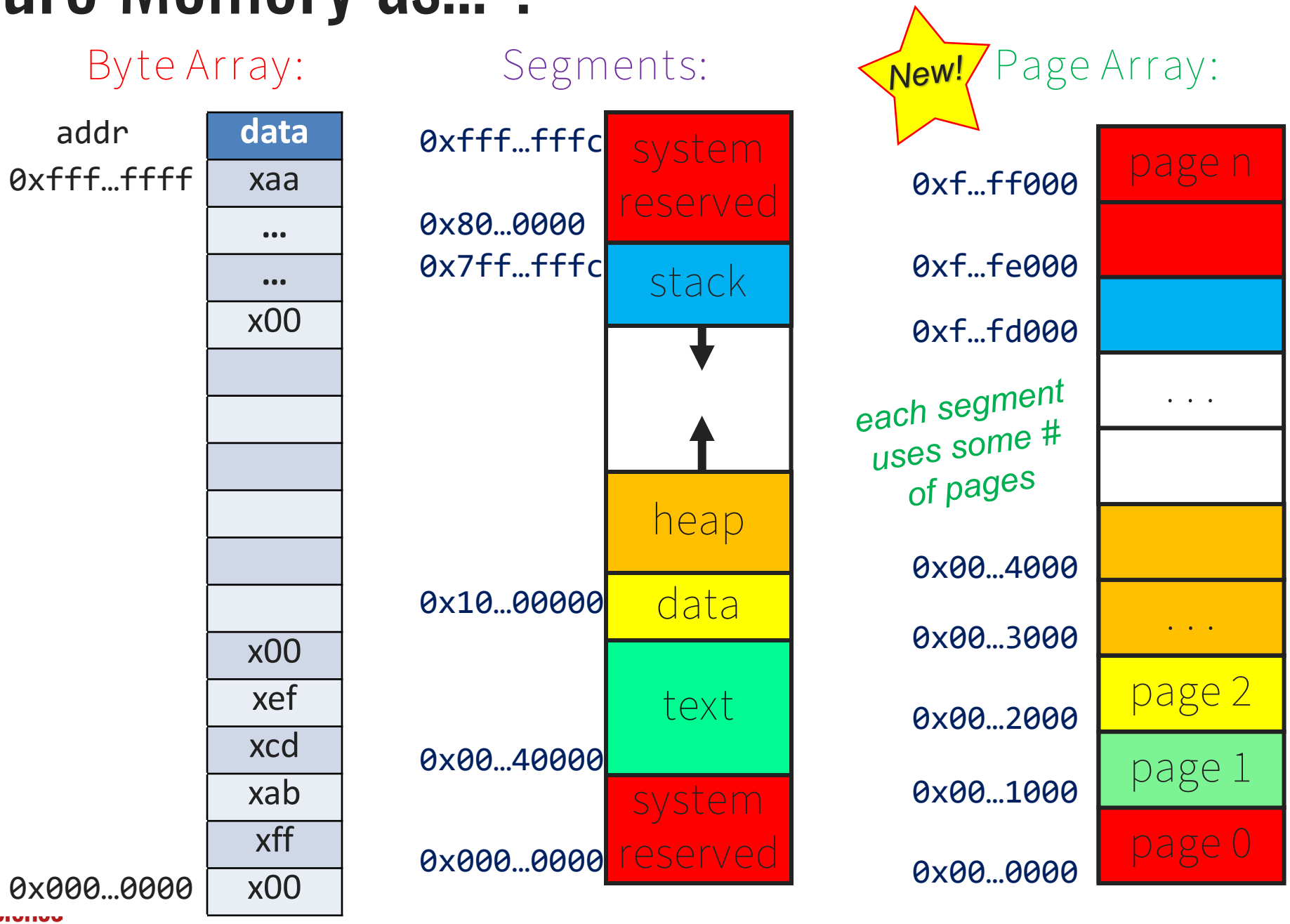
What is Virtual Memory?

How does Virtual memory Work?

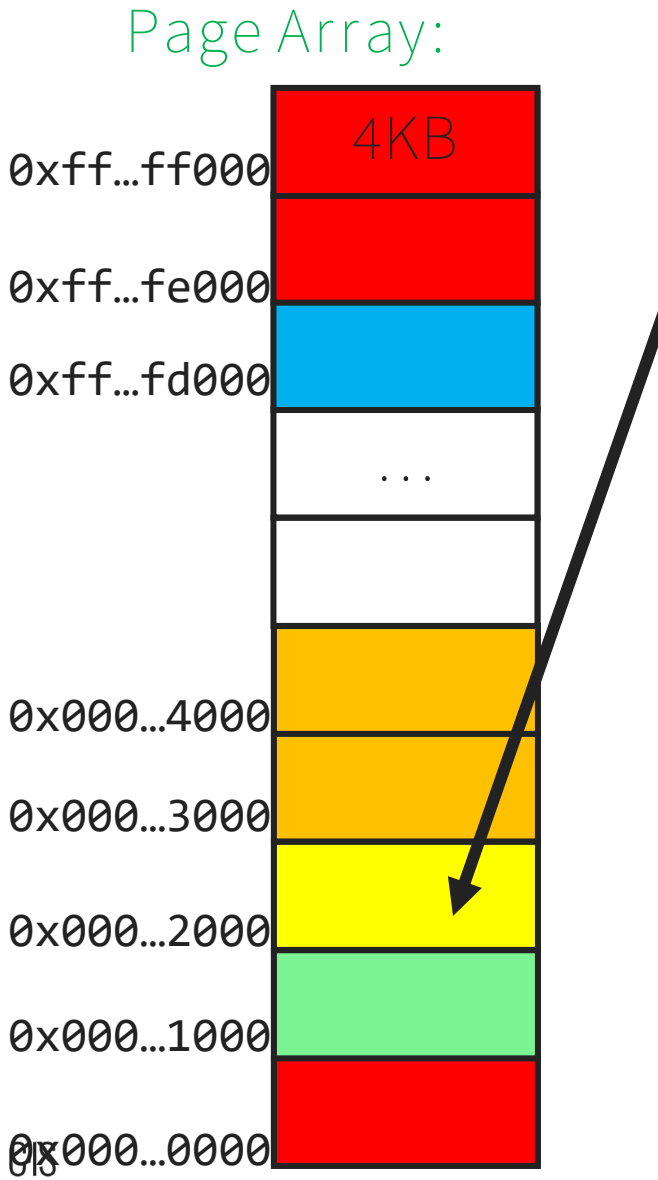
- Address Translation
- Page Table
- Paging
- Overhead
- Performance
- Virtual Memory & Caches


Futurism
IBM
Data Processing
Division
San Jose, California
A 5MB hard drive being
shipped by IBM, 1956

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, LD (ISA)
Registers: 64 bits (ISA)
Caches: cache line/block (µarch)

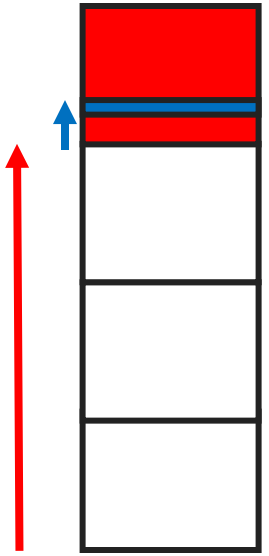
Address bits divided into:

tag: sanity check for address match
index: which entry in the cache
offset: which byte in the line

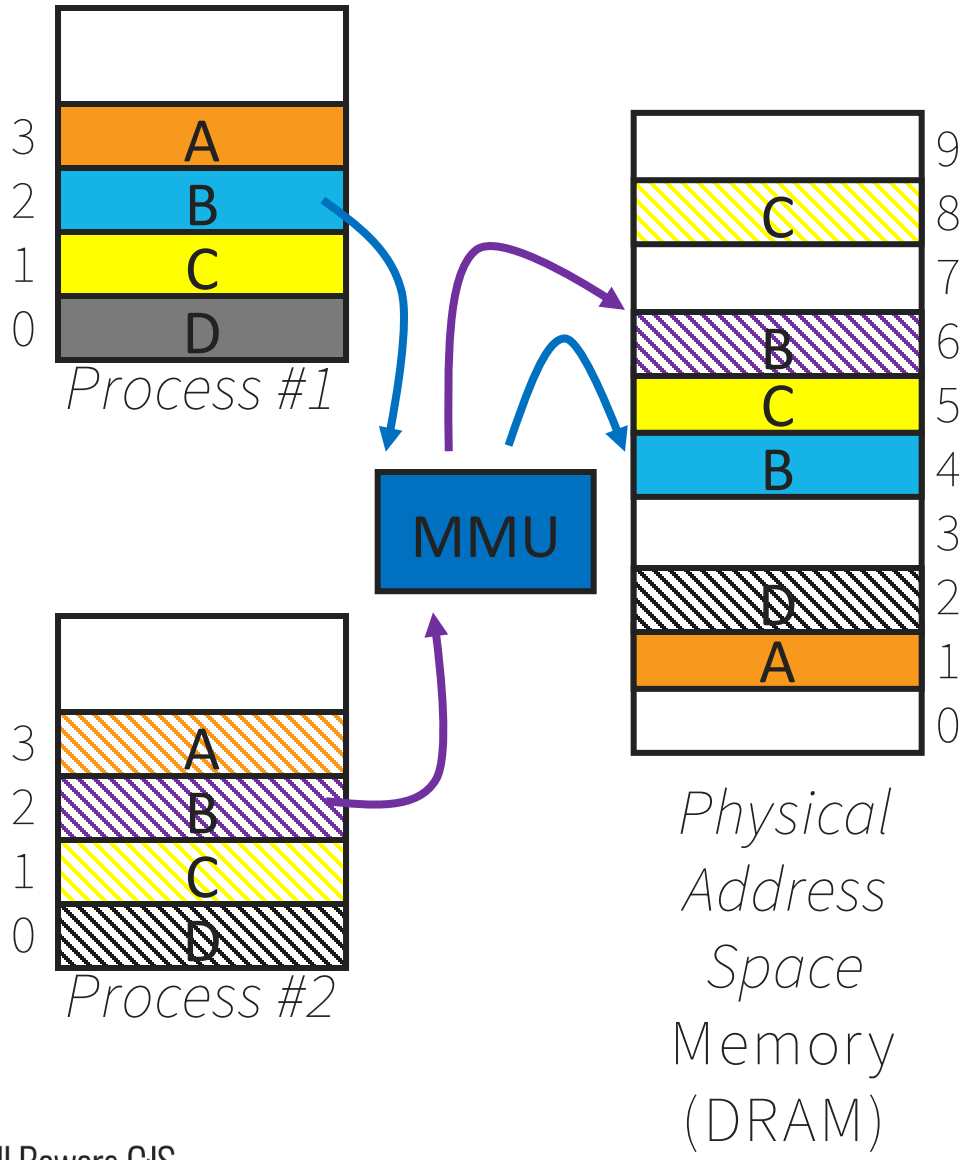
Memory: page

Address bits divided into:

page number: which page in memory
offset: which byte in the page



Address Translator: MMU



- Processes use virtual addresses
- DRAM 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];
```

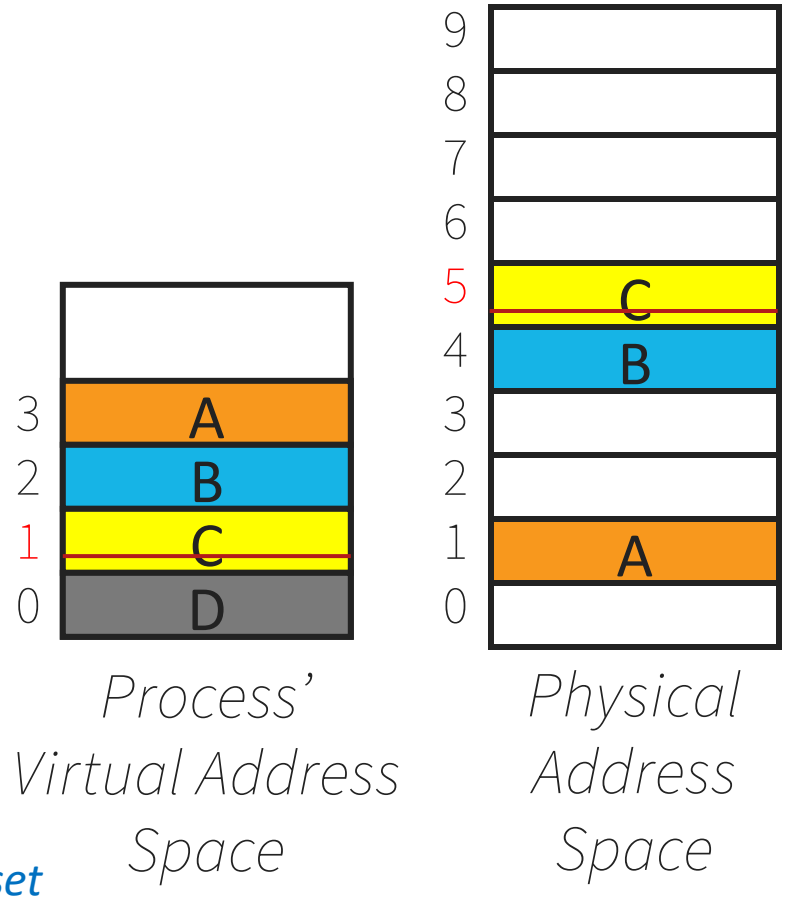
Remember:

any address 0x00001234

is x234 bytes into Page C

both virtual & physical

VP 1 → PP 5



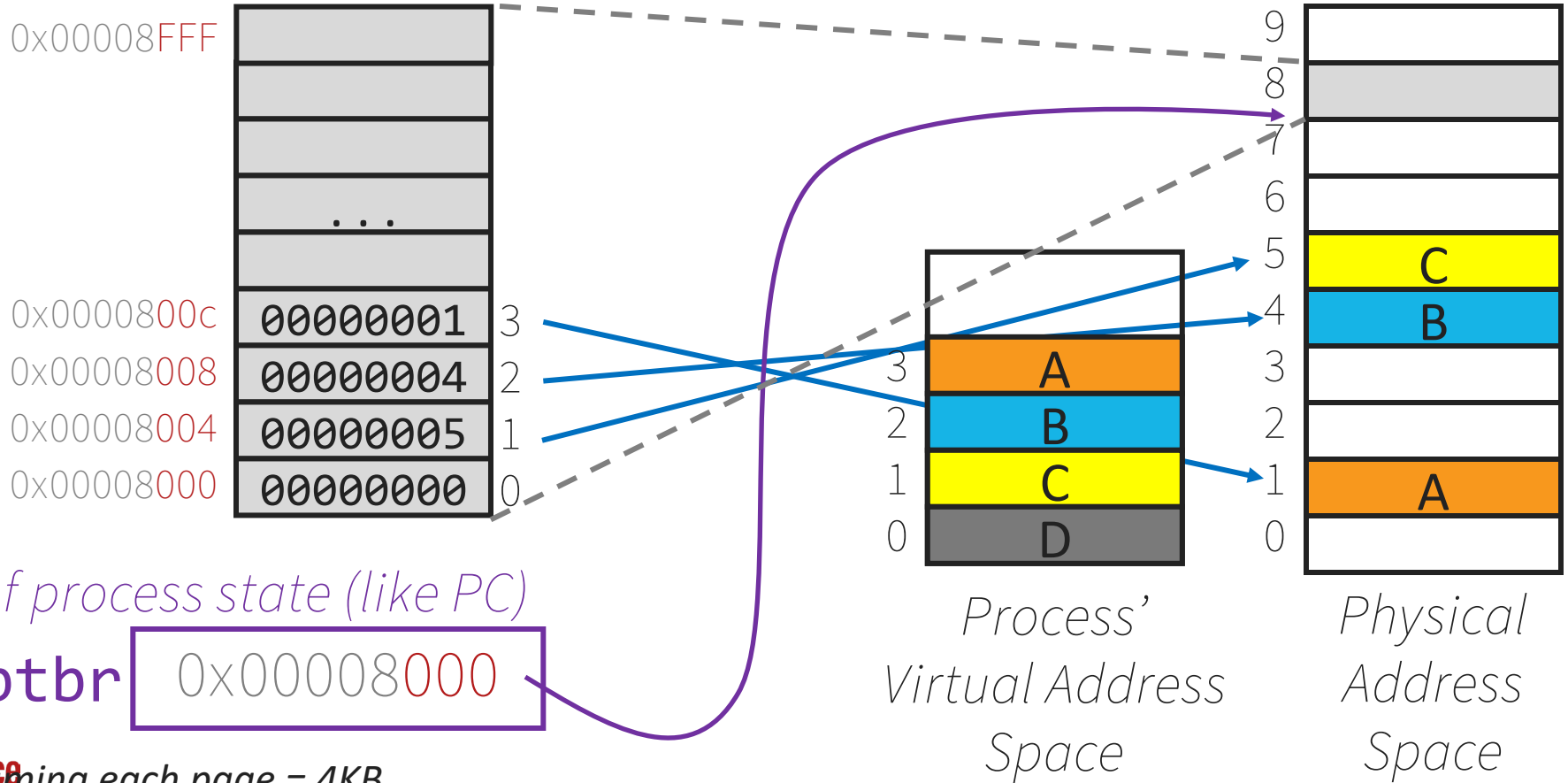
Assuming each page = 4KB, lower 12 bits → offset

Page Table Basics

1 Page Table *per process*

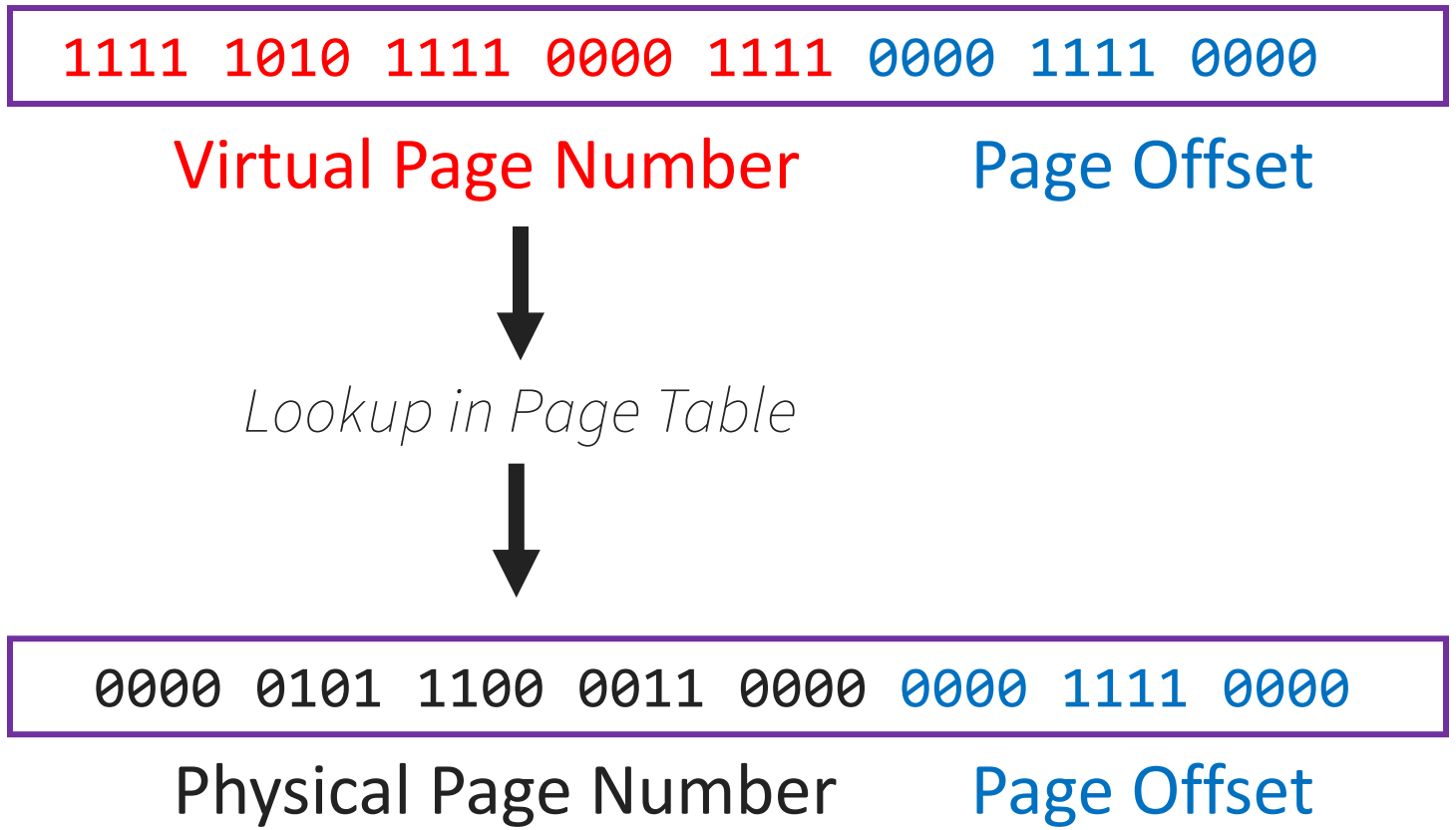
Lives in Memory, *i.e., in a page (or more…)*

Location stored in **Supervisor Page-Table Base Register**



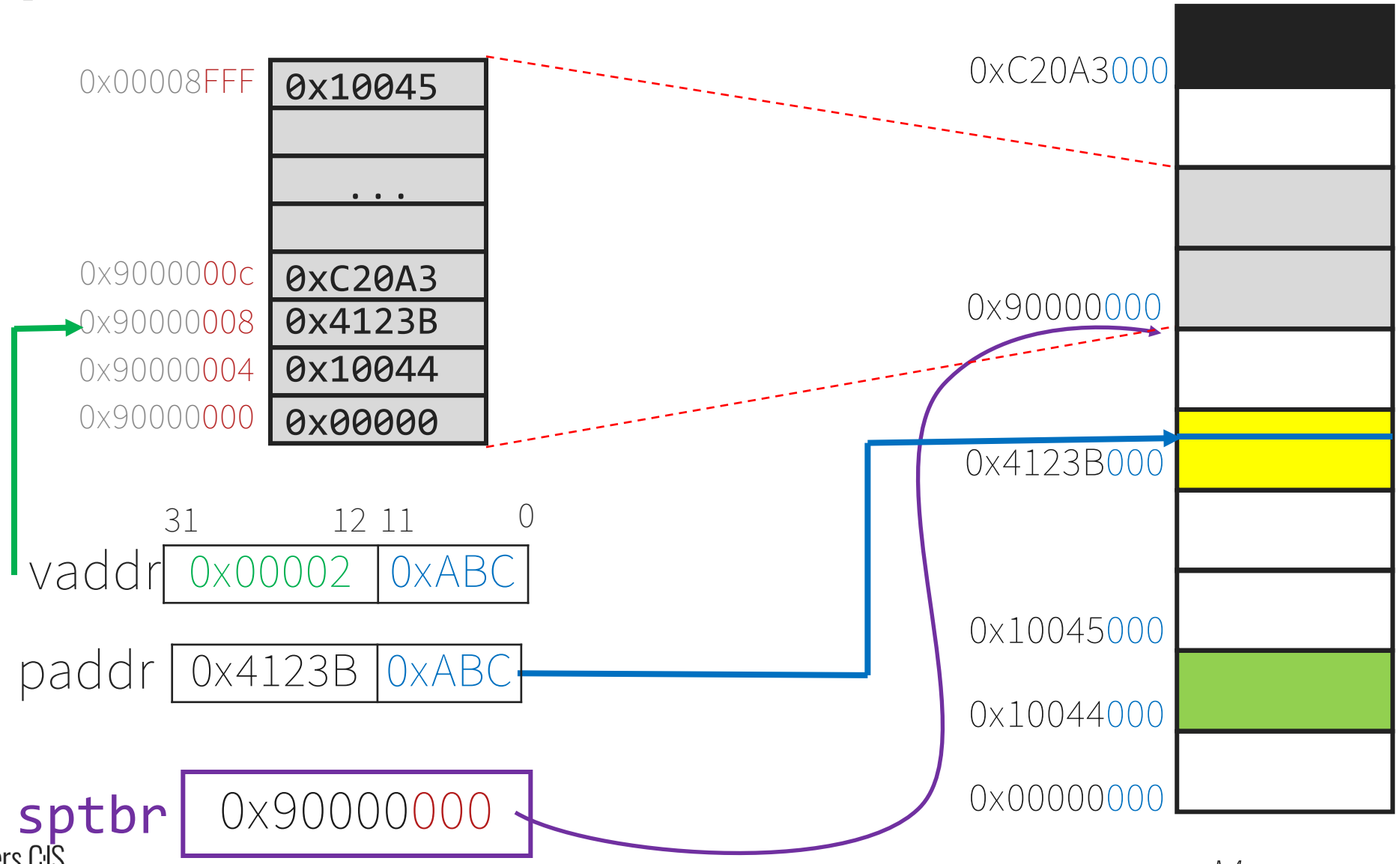
Assuming each page = 4KB

Simple Address Translation



Assuming each page = 4KB

Simple Page Table Translation



Assuming each page = 4KB

Takeaway

- All problems in computer science can be solved by another level of indirection.

- Need a map to translate a "fake" virtual address (generated by CPU) to a "real" physical Address (in memory)

- Virtual memory is implemented via a "Map", a PageTable, that maps a vaddr (a virtual address) to a paddr (physical address):

  paddr Ṗ PageTable[vaddr]

Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

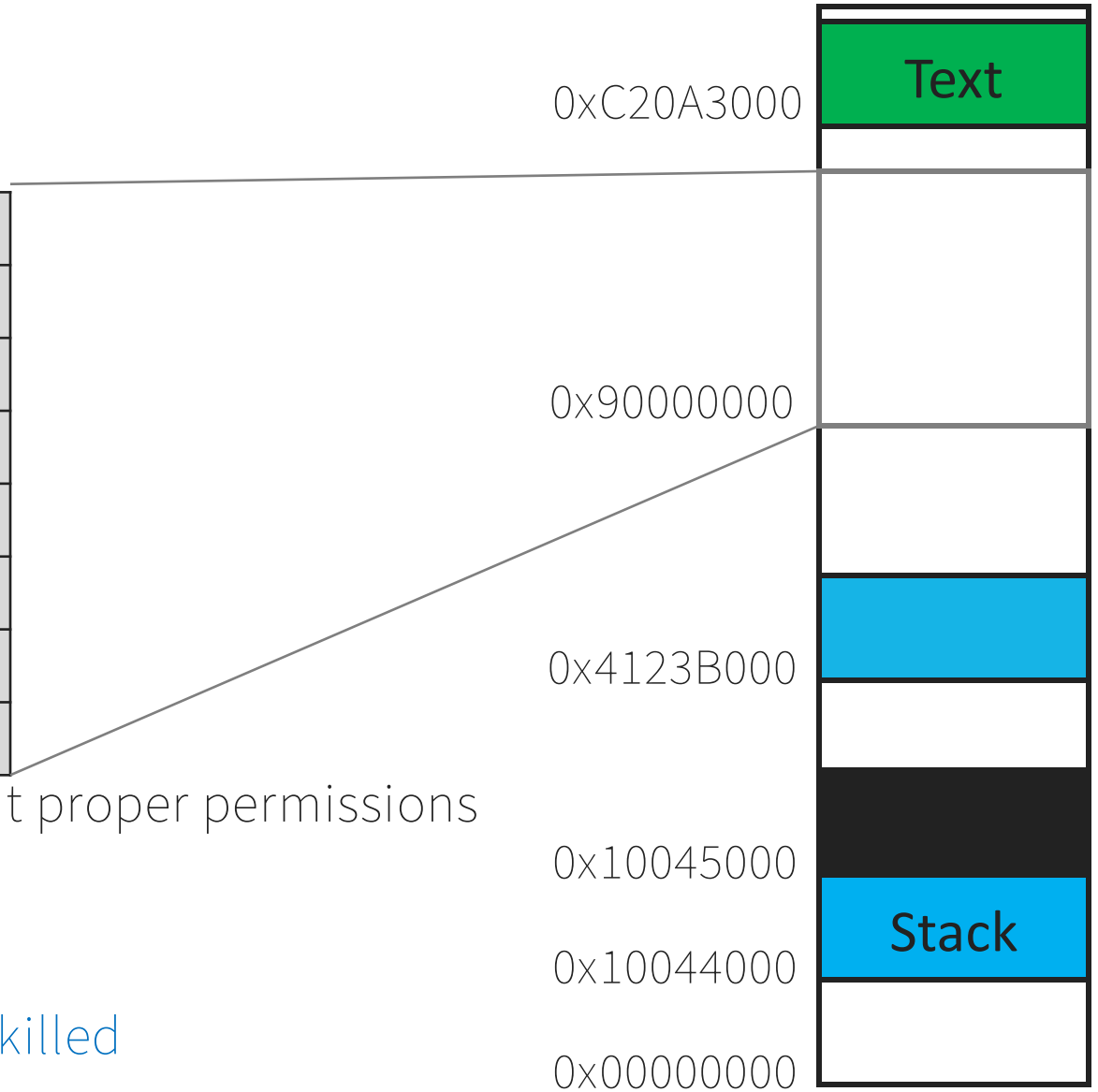
- Address Translation
- Page Table
- Paging
- Overhead
- Performance
- Virtual Memory & Caches

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	Physical Page Number
0				
1	1	0	1	0xC20A3
0				
0	1	1	0	0x10045
0				
1	1	1	1	0x4123B
1	1	1	0	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

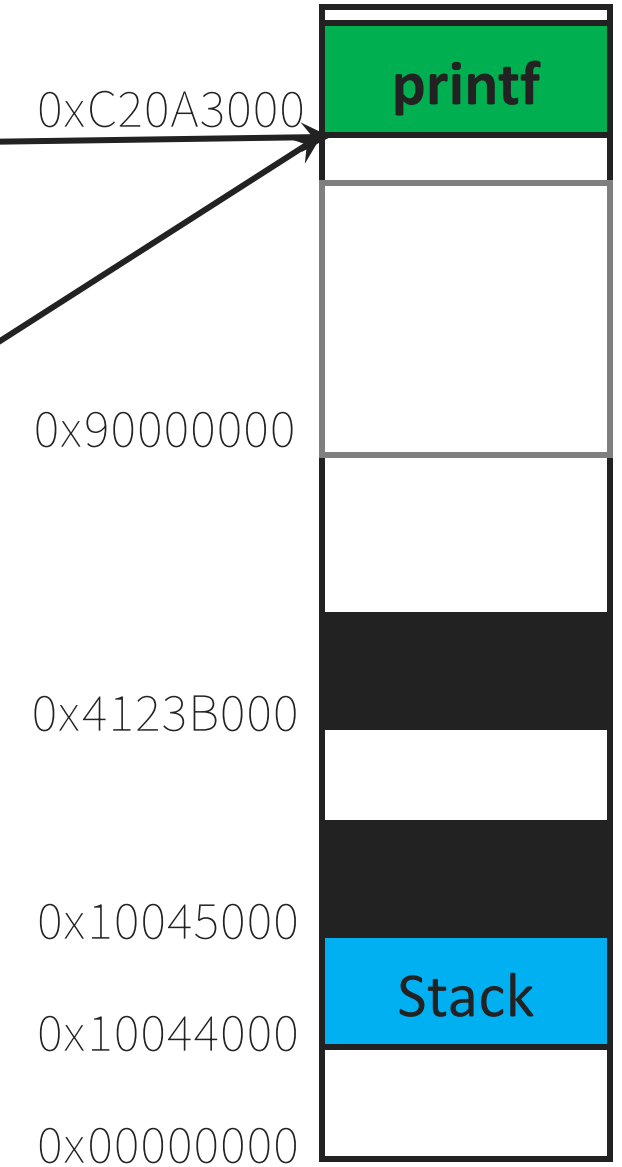
Aliasing:

Proc 1

V	R	W	X	Physical Page Number
0				
1	1	0	1	0xC20A3
0				
0	1	1	0	0x10045

Proc 2

V	R	W	X	Physical Page Number
0				
1	1	0	1	0xC20A3
0				
0	1	1	0	0x4123B



mapping several virtual addresses
→ same physical page

Takeaway

- All problems in computer science can be solved by another level of indirection.
- Need a map to translate a "fake" virtual address (generated by CPU) to a "real" physical Address (in memory)

- Virtual memory is implemented via a "Map", a PageTable, that maps a vaddr (a virtual address) to a paddr (physical address):
- paddr Ṗ PageTable[vaddr]

- A page is constant size block of virtual memory. Often, the page size will be around 4kB to reduce the number of entries in a PageTable.

- We can use the PageTable to set Read/Write/Execute permission on a per page basis. Can allocate memory on a per page basis. Need a valid bit, as well as Read/Write/Execute and other bits.
- But, overhead due to PageTable is significant.

Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

- Address Translation
- Page Table
- Paging
- Overhead
- 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!)

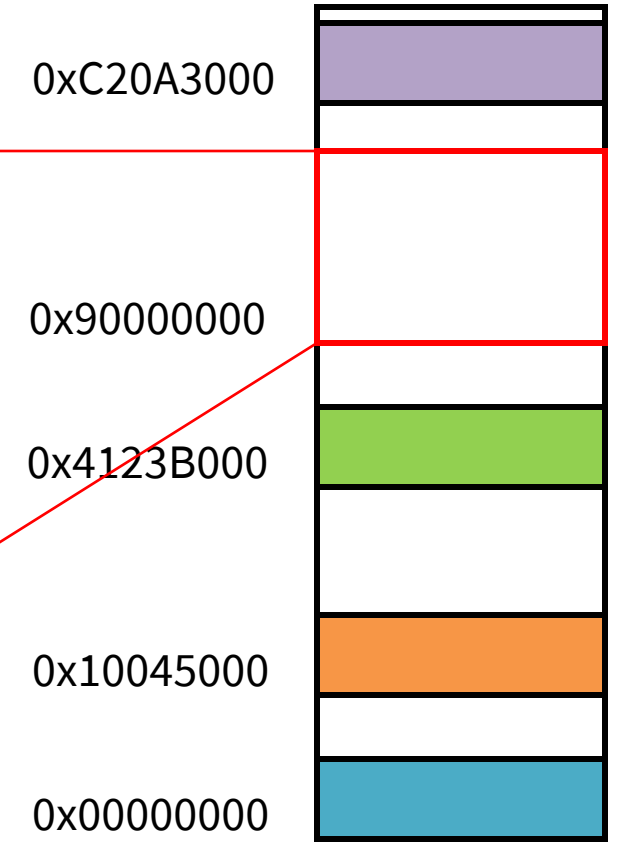
- Pages used recently mostly likely to be used again

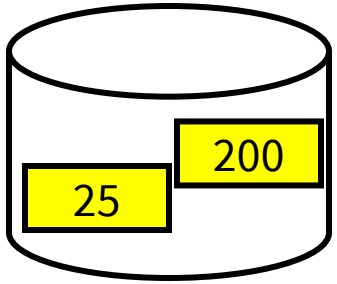
More Meta-Data:

- 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!

Page Fault

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

Why? Maybe the page…

- wasn't needed yet (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

- See next week's lectures + CS 4410 for details

Performance-wise page faults are really bad!

Takeaway

- All problems in computer science can be solved by another level of indirection.
- Need a map to translate a "fake" virtual address (generated by CPU) to a "real" physical Address (in memory)

- Virtual memory is implemented via a "Map", a PageTable, that maps a vaddr (a virtual address) to a paddr (physical address):
- paddr ṗ PageTable[vaddr]

- A page is constant size block of virtual memory. Often, the page size will be around 4kB to reduce the number of entries in a PageTable.

- We can use the PageTable to set Read/Write/Execute permission on a per page basis. Can allocate memory on a per page basis. Need a valid bit, as well as Read/Write/Execute and other bits.

Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

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

Page Table Overhead (32-bits)

- 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

  1. # entries in PageTable?
  2. size of PageTable? (in bytes)
- Physical address space:
  - total physical memory: 2^{29} bytes = 512MB

  1. overhead for 10 processes?

Now how big is this Page Table?

`struct pte_t page_table[2^20]`

`sizeof(struct pte_t) = 8 bytes`

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

Assuming each page = 4KB

Page Table Overhead (64-bits)

- How large is PageTable?
- Virtual address space (for each process):

  Given: total virtual memory: 2^{64} bytes = 16EB

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

  1. # entries in PageTable?
  2. size of PageTable? (in bytes)

Takeaway

- All problems in computer science can be solved by another level of indirection.
- Need a map to translate a "fake" virtual address (generated by CPU) to a "real" physical Address (in memory)

- Virtual memory is implemented via a "Map", a PageTable, that maps a vaddr (a virtual address) to a paddr (physical address):
- paddr Ṗ PageTable[vaddr]

- A page is constant size block of virtual memory. Often, the page size will be around 4kB to reduce the number of entries in a PageTable.

- We can use the PageTable to set Read/Write/Execute permission on a per page basis. Can allocate memory on a per page basis. Need a valid bit, as well as Read/Write/Execute and other bits.
- But, overhead due to PageTable is significant.

Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

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

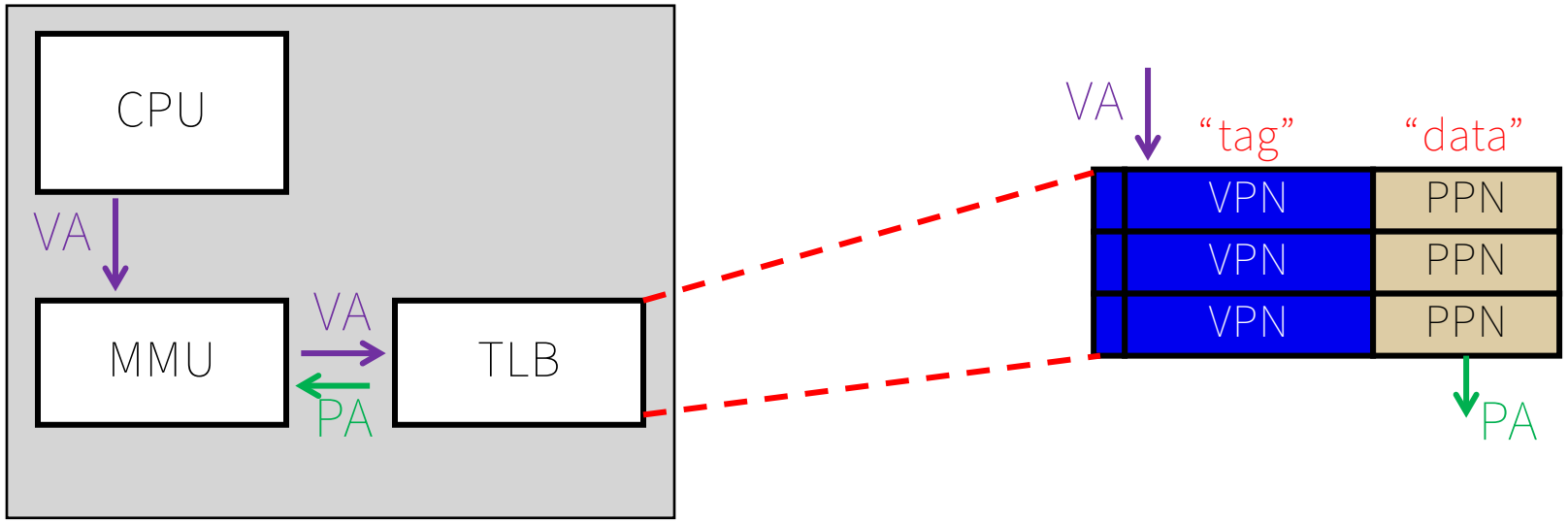
Watch Your Performance Tank!

For *every instruction*:

1. MMU translates address (virtual → physical)
   - Uses `sptbr` to find Page Table in memory
   - Looks up entry for that virtual page
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

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
- 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
  - If that fails, "walk" the Page Table
    - Use `sptbr` 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

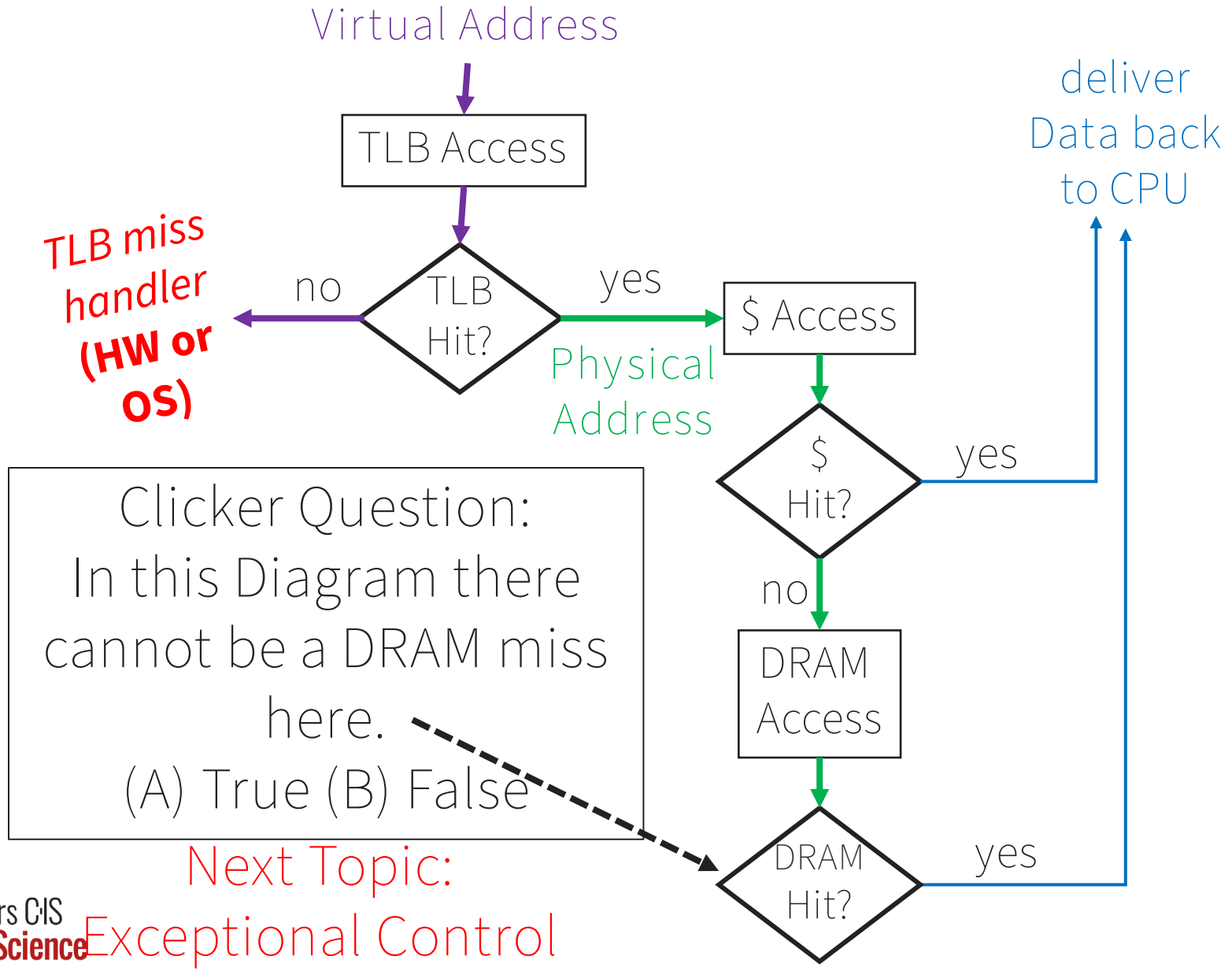
Virtual Memory Agenda

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



Virtual Memory

- Software use virtual addresses, that let every program imagine it has access to ALL of the physical memory
- Access to $ and DRAM use physical addresses
- OS sets up 1 page table per process.
  Page table specifies the virtual -> physical mapping
- *Every memory access* must now be translated via the sptbr and the page table
- TLB (translation lookaside buffer) caches translations for *more speed*