

Lecture 16

Virtual Memory

CS213 – Intro to Computer Systems
Branden Ghen a – Winter 2026

Slides adapted from:

St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Administrivia

- Homework 3
 - Due Tuesday next week (reminder, no late submissions)
- SETI Lab
 - Due next week Thursday
 - See pinned Piazza posts for a video and the steps to this lab
 - Test with just one thread before testing with many!!
 - If it doesn't work for one thread, it'll never work with more than one

Today's Goals

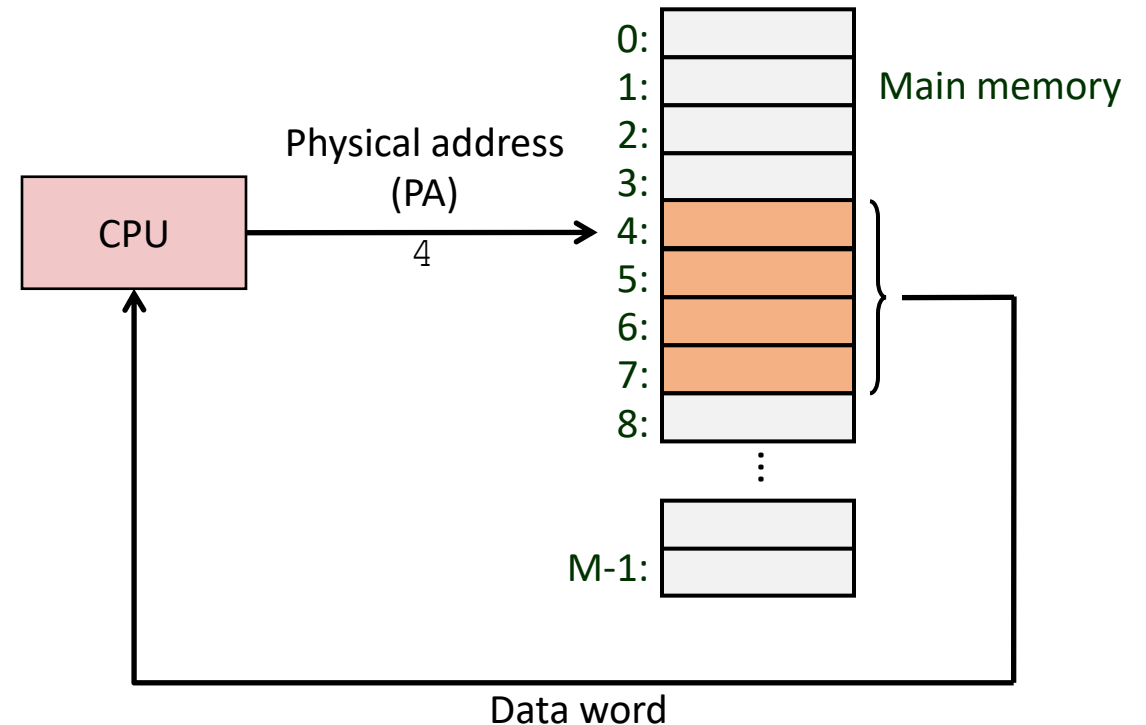
- Understand goals and application of virtual memory
- Explore how virtual memory resolves memory problems
- Practice translating virtual addresses to physical addresses
- Bonus: Practice problems at the end
 - Also some bonus details on caching page table entries and on multi-level page tables

Outline

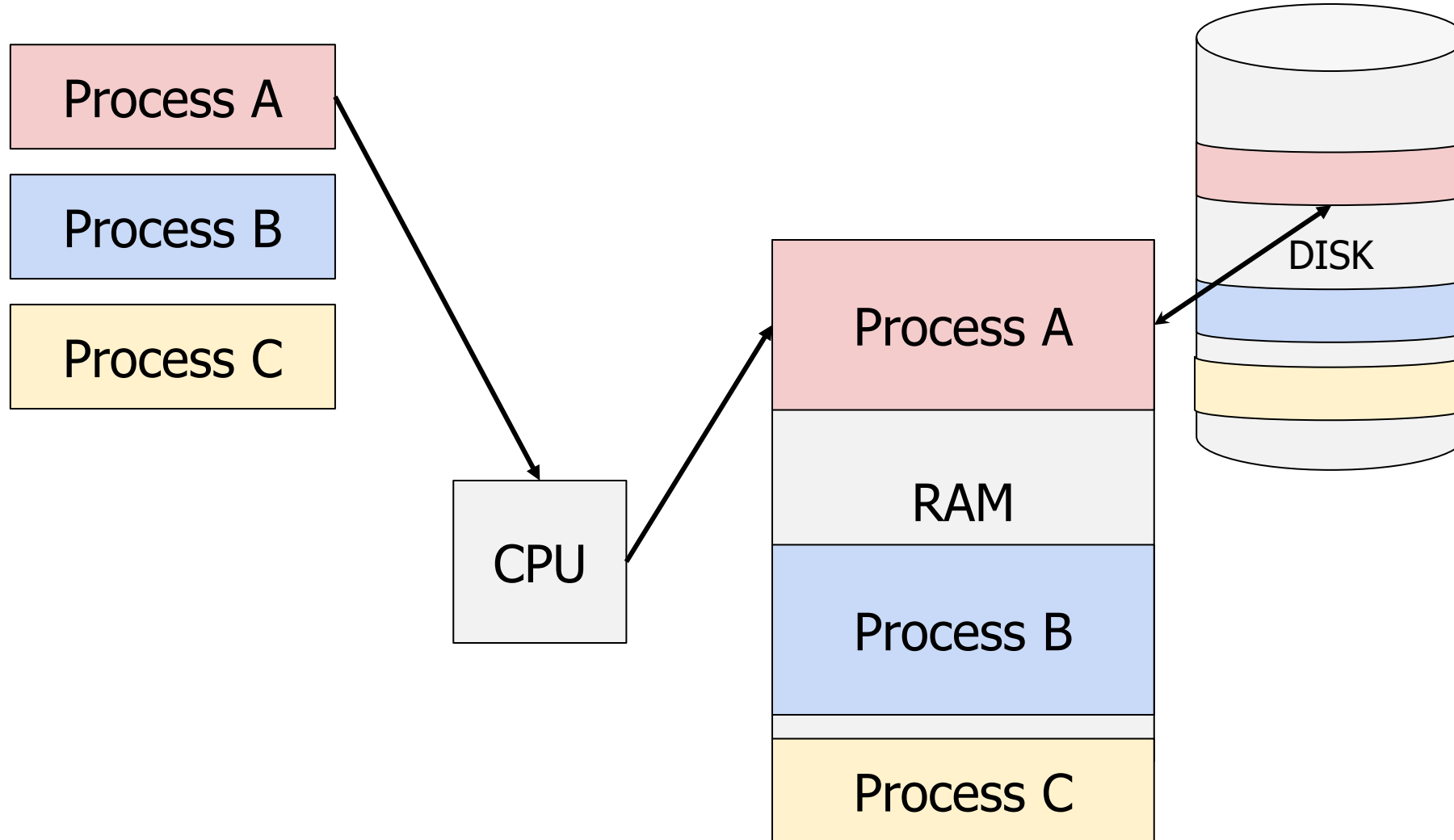
- **Virtual Memory Concept**
- Virtual Memory Process
- Solving Memory Problems with Virtual Memory
- Address Translation
- Virtual Memory Summary

A system using physical addresses

- Main memory - An array of M contiguous byte-sized cells, each with a unique physical address
- Physical addressing
 - Most natural way to access it
 - Addresses used by the CPU correspond to bytes in memory
 - Used in simple systems like early PCs and embedded microcontrollers

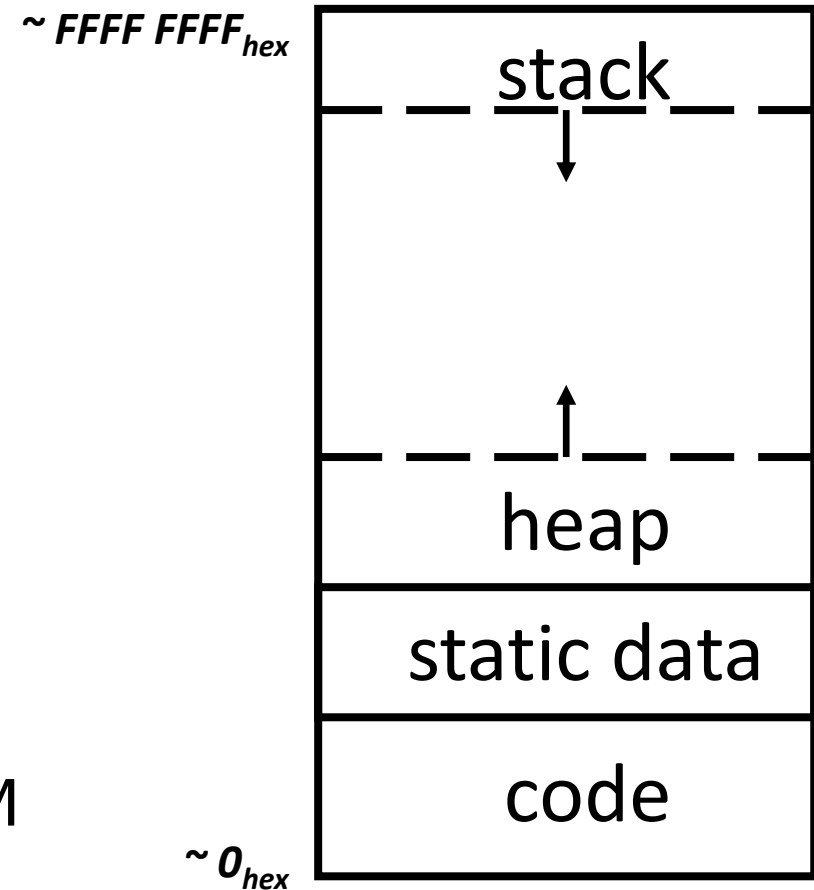


The reality of memory in a computer



A process's view of the memory

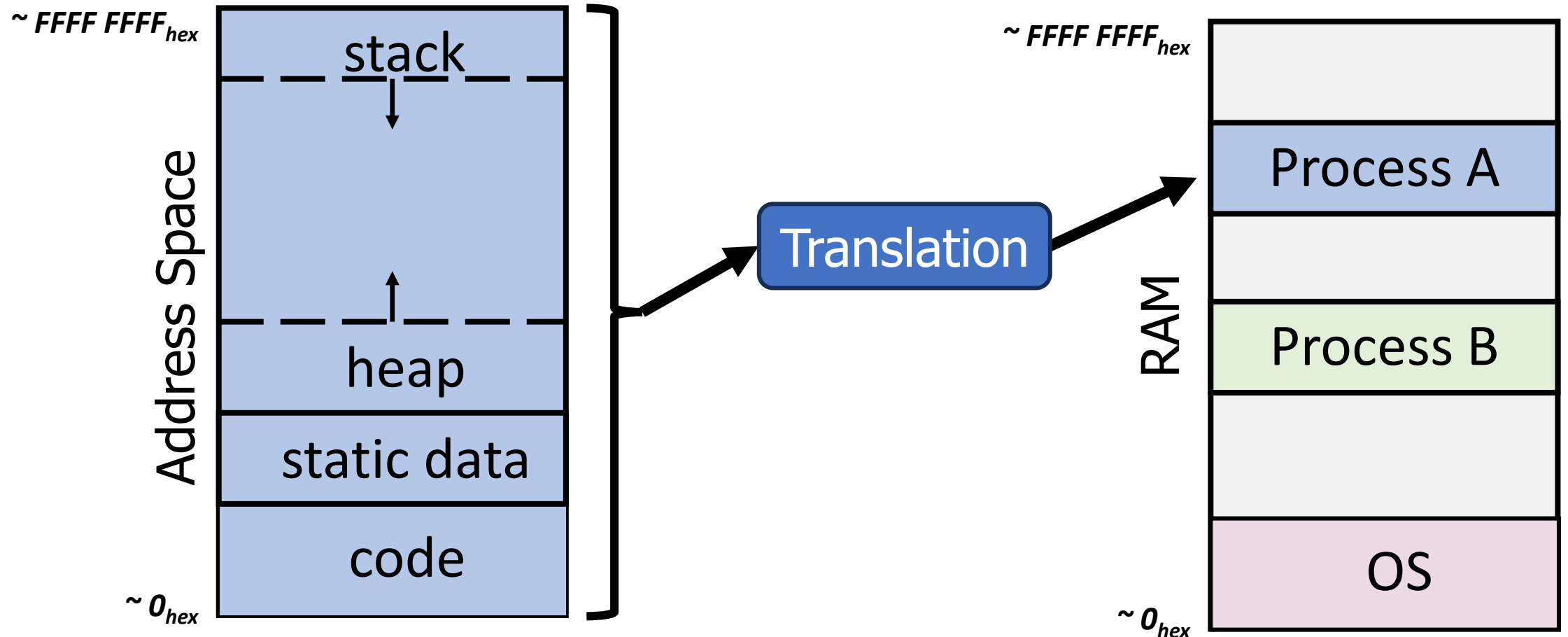
- The **Address Space** of the process
- The illusion:
 - Processes run alone on the computer
 - They have full access to every memory address
 - 2^{64} bytes of memory available to them
- The reality:
 - There are many processes which share RAM
 - There is only so much RAM available



Virtual memory enables this illusion

Virtual Addresses

Physical Addresses

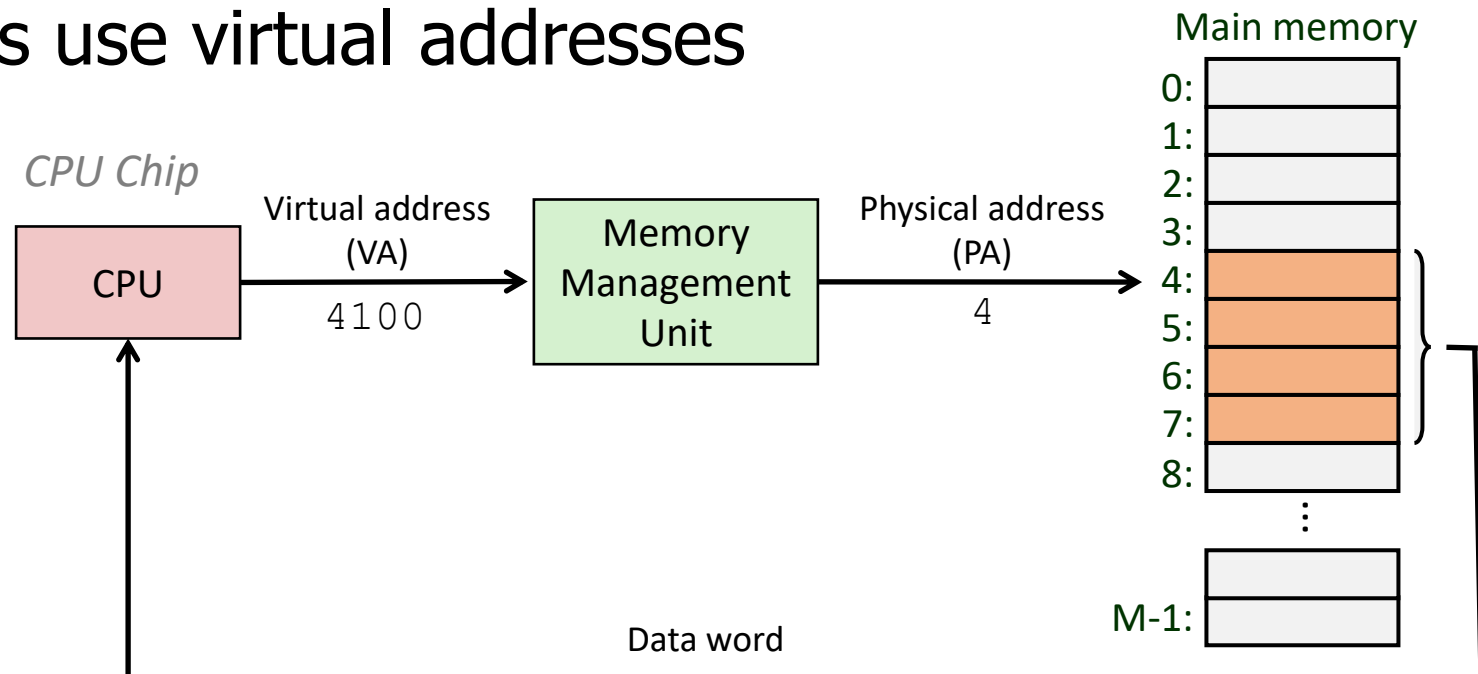


Virtual memory concept

- Disconnect reality of RAM from illusion of main memory
- Processes work with the illusion
 - They use **virtual addresses** to reference where their memory is
- Computer (and OS) work with the reality
 - They use **physical addresses** that are real locations in RAM
- The hardware/OS *translates* virtual addresses into physical addresses

A system using virtual addresses

- The CPU generates virtual address
 - Address translation is done by dedicated hardware (memory management unit) via OS-managed lookup table (called a **Page Table**)
 - Resulting physical address is used to access memory hierarchy
- Modern processors use virtual addresses
 - All addresses your programs work with are virtual!



Your experiences with Virtual Memory

- In Attack Lab, what was the address of touch2?
 - 0x40000-ish, right?
 - The same each time you run it too
- But multiple of you were running separate ctargget processes at the same time on Moore
 - 0x40000-ish was a **Virtual Address**
- Really, each process's code was at a totally different **Physical Address** in Moore's actual RAM

Virtual Memory

- From here on out, we'll be working with two different memory spaces:
 - **Virtual Memory (VM):** A large ($\sim$ infinite) space that a process believes it, and only it, has access to
 - **Physical Memory (PM):** The limited RAM space your computer must share among all processors
- This idea is independent of physical caches
 - There are still multiple layers of memory caches in the CPU
 - They might use virtual or physical addresses
 - We'll usually assume caches use physical addresses for this class

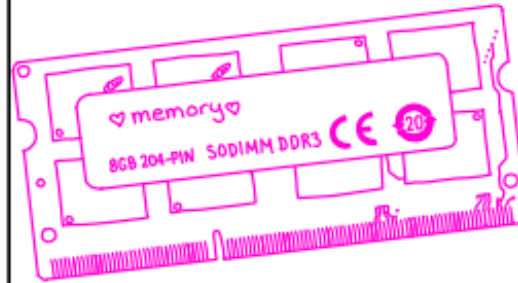
Break + Review

JULIA EVANS
@b0rk

virtual memory

17

your computer has
physical memory



physical memory has
addresses, like

0-8GB

but when your program
references an address
like 0x5c69a2a2,
that's not a physical
memory address!
It's a **virtual** address.

every program has its
own virtual address space

program 1
0x129520 → "puppies"

program 2
0x129520 → "bananas"

Linux keeps a mapping from
virtual memory pages to
physical memory pages called
the **page table**

a "page" is a 4kb chunk of memory
or sometimes bigger

PID	virtual addr	physical addr
1971	0x20000	0x192000
2310	0x20000	0x228000
2310	0x21000	0x9788000

when your program
accesses a virtual address

CPU
I'm accessing
0x21000

MMU
"memory
management
unit"
hardware
I'll look that up in
the **page table** and
then access the right
physical address

every time you switch
which process is running,
Linux needs to switch
the page table

Linux
here's the address of
process 2950's page table

thanks, I'll use
that now!
MMU

Outline

- Virtual Memory Concept
- **Virtual Memory Process**
- Solving Memory Problems with Virtual Memory
- Address Translation
- Virtual Memory Summary

The Operating System manages the computer

- The OS is in charge of a computer
 - Manages hardware
 - Allocates resources to processes
 - Enforces restrictions and security
- One resource the OS manages is memory
 - For example: a “SEGFAULT” is a message from the OS when a process tries to access memory it wasn’t allocated

We translate between entire pages of memory

- If we want to translate memory from virtual to physical, the OS is going to need some kind of table with each mapping
- Mapping every virtual byte to some physical byte would require our mapping to contain one address per byte
 - 8 bytes (one address) of data per byte of data...
 - That's not going to work
- Instead, we organize memory into **Pages**: contiguous chunks of memory
 - Each virtual page will map to a physical page
 - Page size is usually 4 kB or so, occasionally larger (can be 2 MB or 1 GB)

Page Tables list Virtual-to-Physical Translations

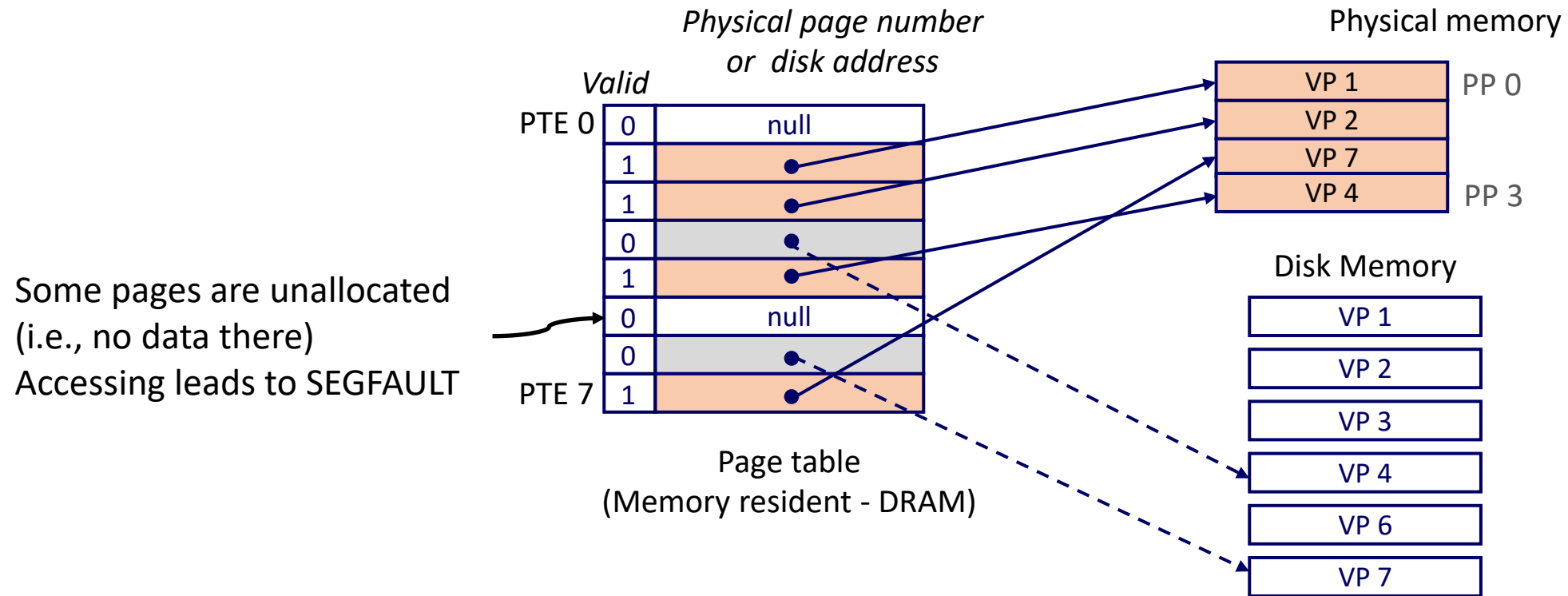
- A **page table** maps virtual pages to physical pages for a process
 - One page table entry (PTE) per page of virtual memory
- A separate Page Table exists for each running process
 - Each process has its own mappings from virtual to physical
- Each Page Table Entry could have one of three possible values:
 1. Invalid (no actual data exists at this address, **SEGFault**)
 2. An address for the page in physical memory
 3. An address for the page on disk

Why did disk get involved here?

- Physical Memory size: usually a number of GBs these days
 - RAM size is usually tens of GBs (8 or 16 GB is common), more on servers
- Users have a lot more data than that though!
 - Data and programs are stored on the disk (measured in thousands of GBs)
 - When needed we'll load them into RAM and then work with them
- We can also *partially* load things into RAM
 - Focus on the important parts of data: whatever we're using right now
 - Even programs can be partially loaded into RAM
 - Essentially: use RAM as a cache for the disk!

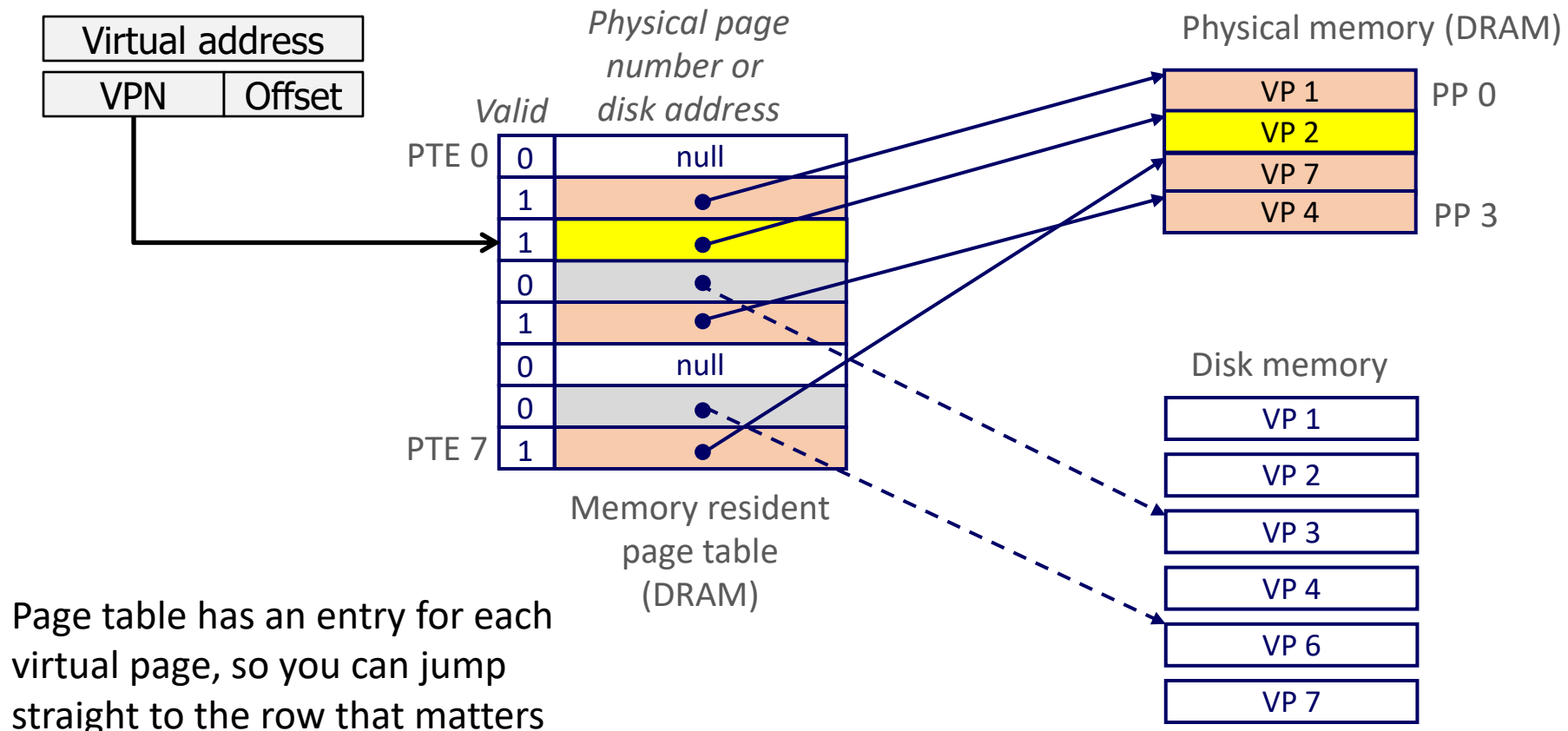
Example Page Table

- One Page Table Entry (PTE) for **every** virtual page
 - Valid entries point to memory
 - Invalid entries point to disk, or to nowhere at all



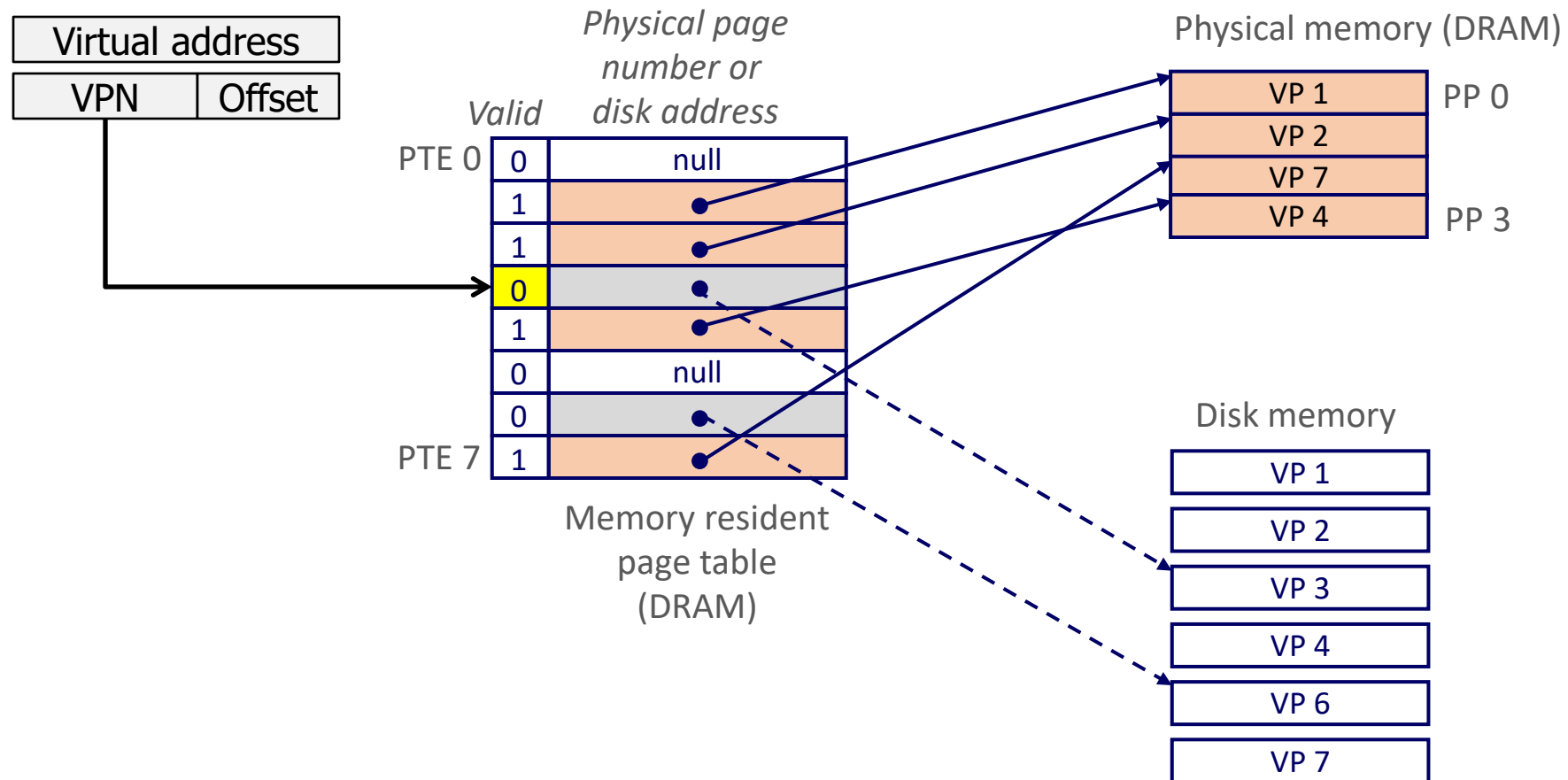
Page Hit

- *Page hit*: reference to a VM word that is in physical memory



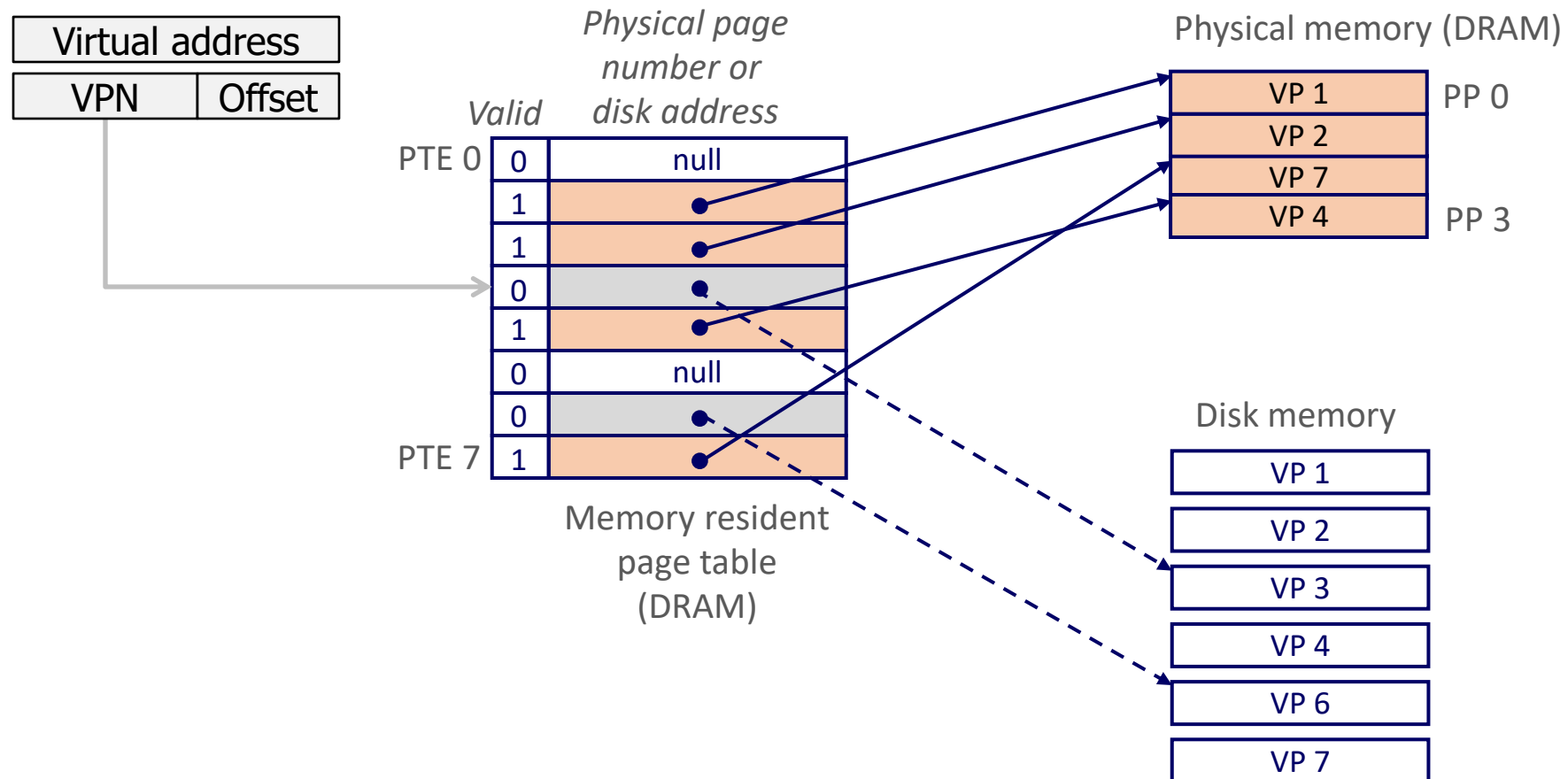
Page Fault

- *Page fault*: reference to VM word that is not in physical memory



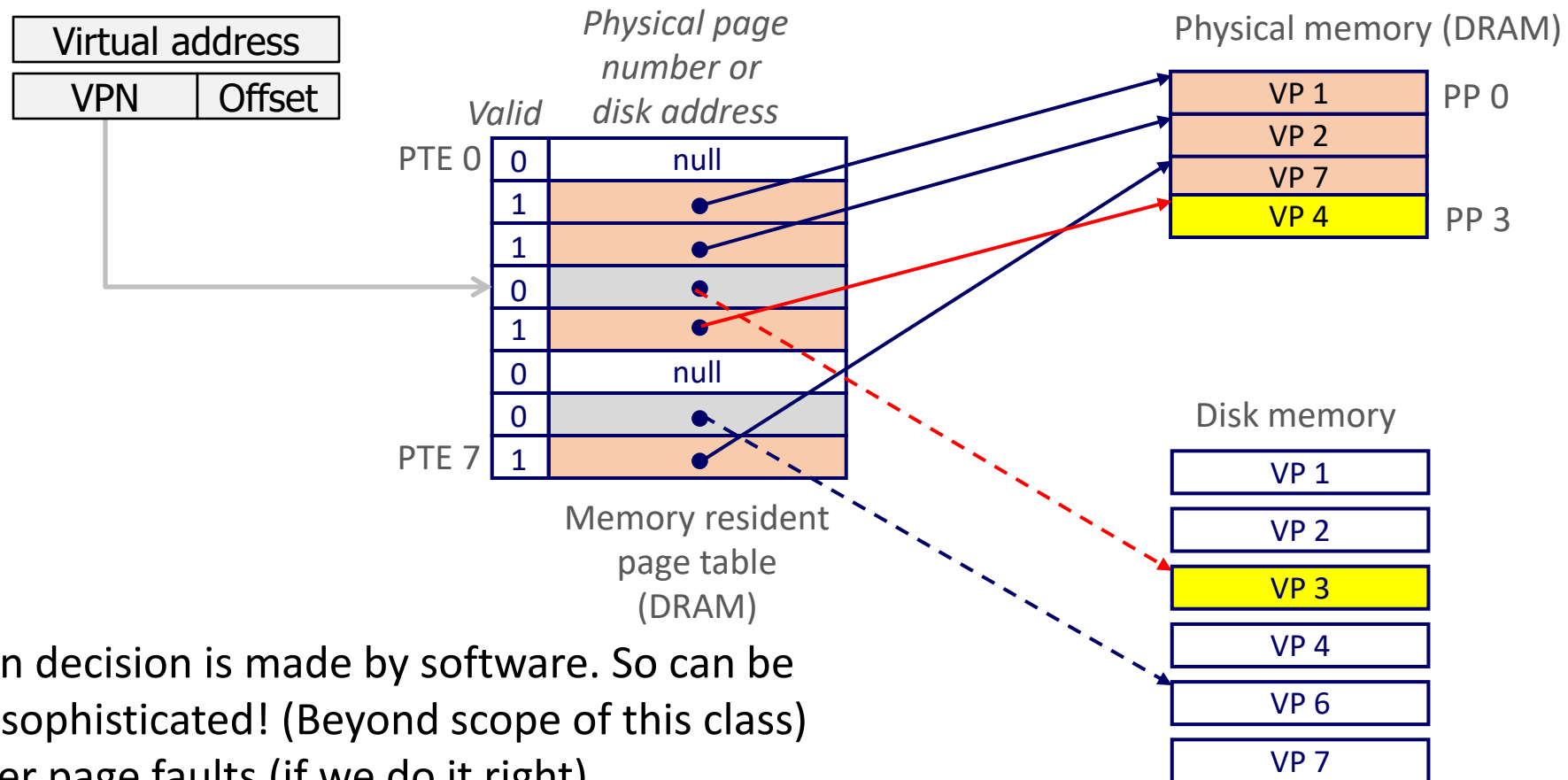
Handling Page Fault

- Page miss causes page fault (a HW exception, OS code kicks in to handle)



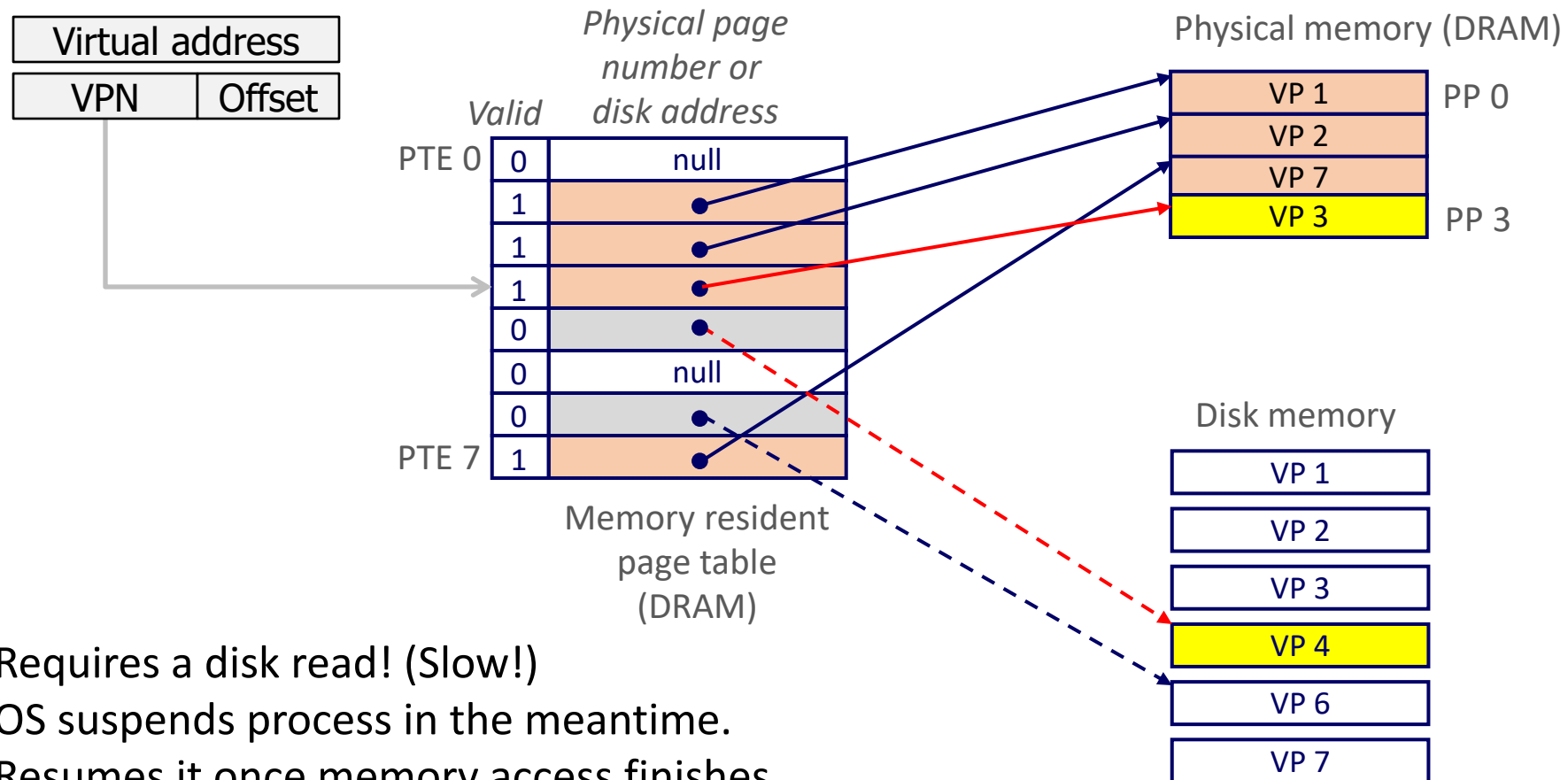
Handling Page Fault

- Page miss causes page fault (a HW exception, OS code kicks in to handle)
- Page fault handler selects a victim to be evicted (here VP 4)



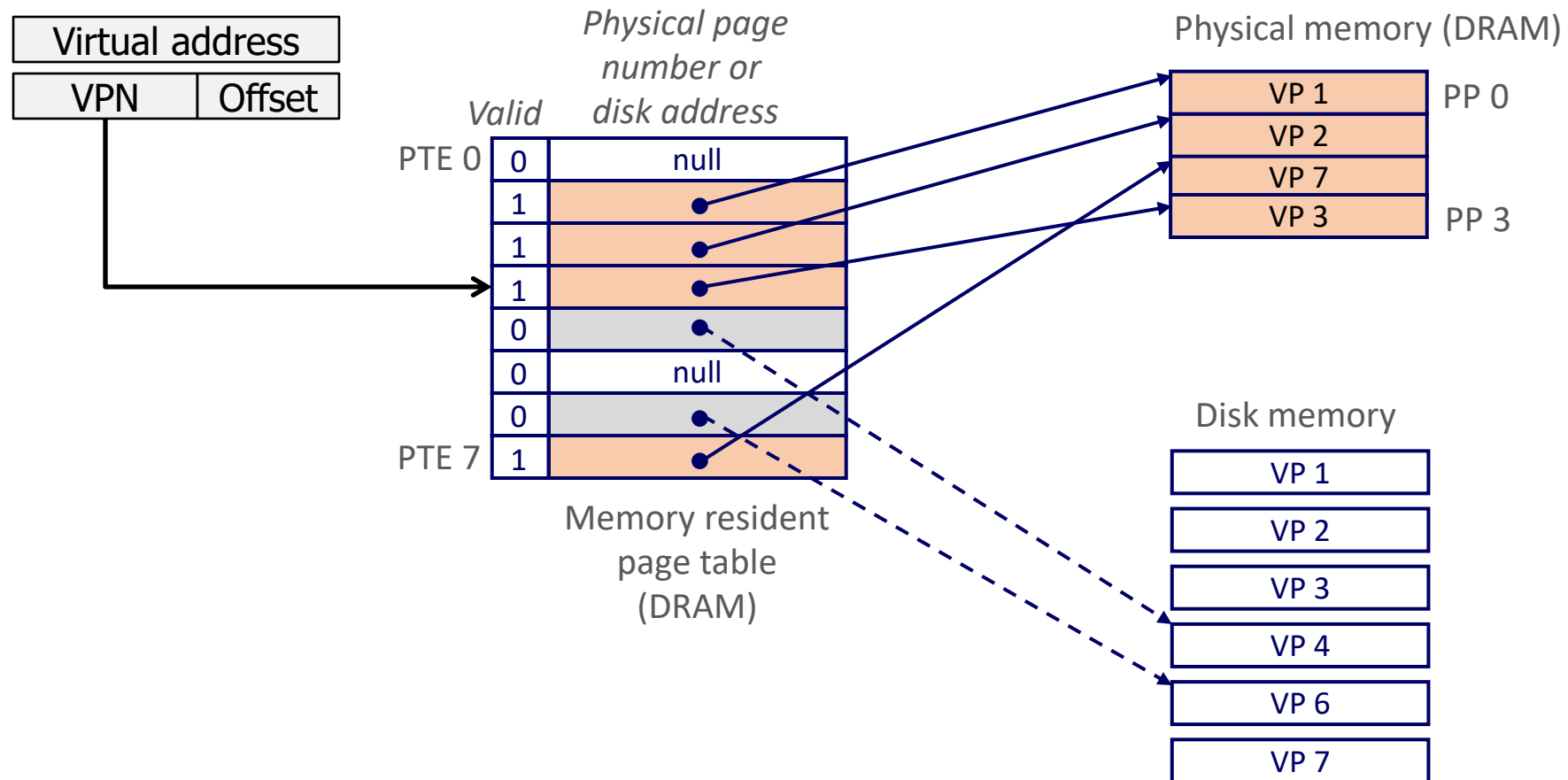
Handling Page Fault

- Page miss causes page fault (a HW exception, OS code kicks in to handle)
- Page fault handler selects a victim to be evicted (here VP 4)
- The victim page is swapped with the disk block of the requested address



Handling Page Fault

- Offending instruction is restarted: page hit this time!



Break + Question

- Computer has:
 - 8 pages of Virtual Memory
 - 4 pages of Physical Memory
- How many entries (rows) does a page table have?
- How many entries can be valid at any time?

Break + Question

- Computer has:
 - 8 pages of Virtual Memory
 - 4 pages of Physical Memory
- How many entries (rows) does a page table have? **8 entries**
- How many entries can be valid at any time? **4 valid**
- Page Table translates Virtual to Physical
 - It needs an entry for each virtual page, so 8 entries
 - Rows are valid if they point at physical memory
 - So only four entries can be valid (unless they share a physical page)

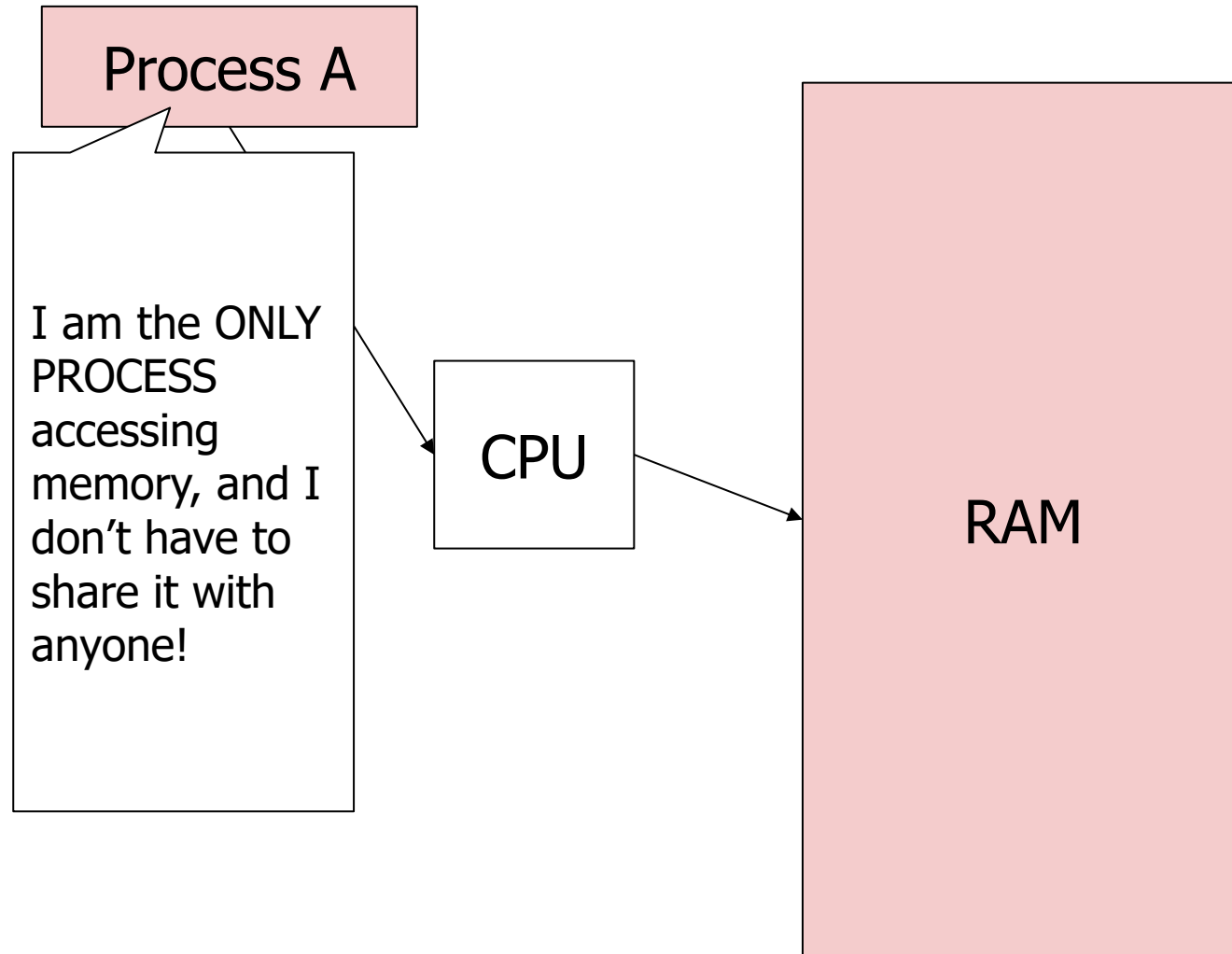
Outline

- Virtual Memory Concept
- Virtual Memory Process
- **Solving Memory Problems with Virtual Memory**
- Address Translation
- Virtual Memory Summary

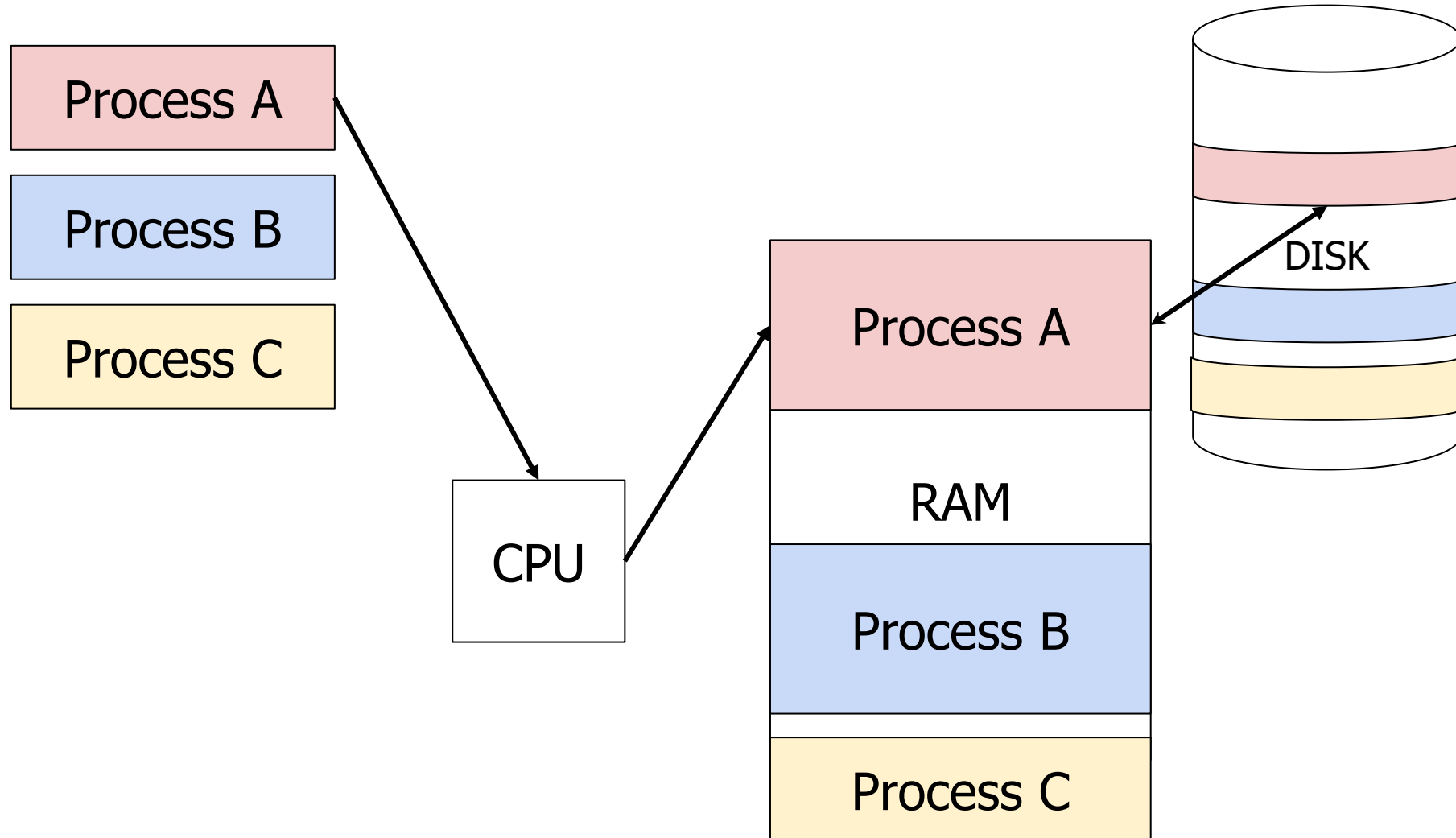
Problems virtual memory solves

- Virtual memory grew more complicated over time to solve several different problems
 - Started off simple, the current design is rather complex
- What problems does virtual memory solve?
 1. Which addresses does each process get?
 2. How do we deal with fragmentation?
 3. How do we support processes bigger than RAM?
 4. How do we protect processes from each other?
 5. How do we deal with how incredibly slow disk is?

The Illusion!



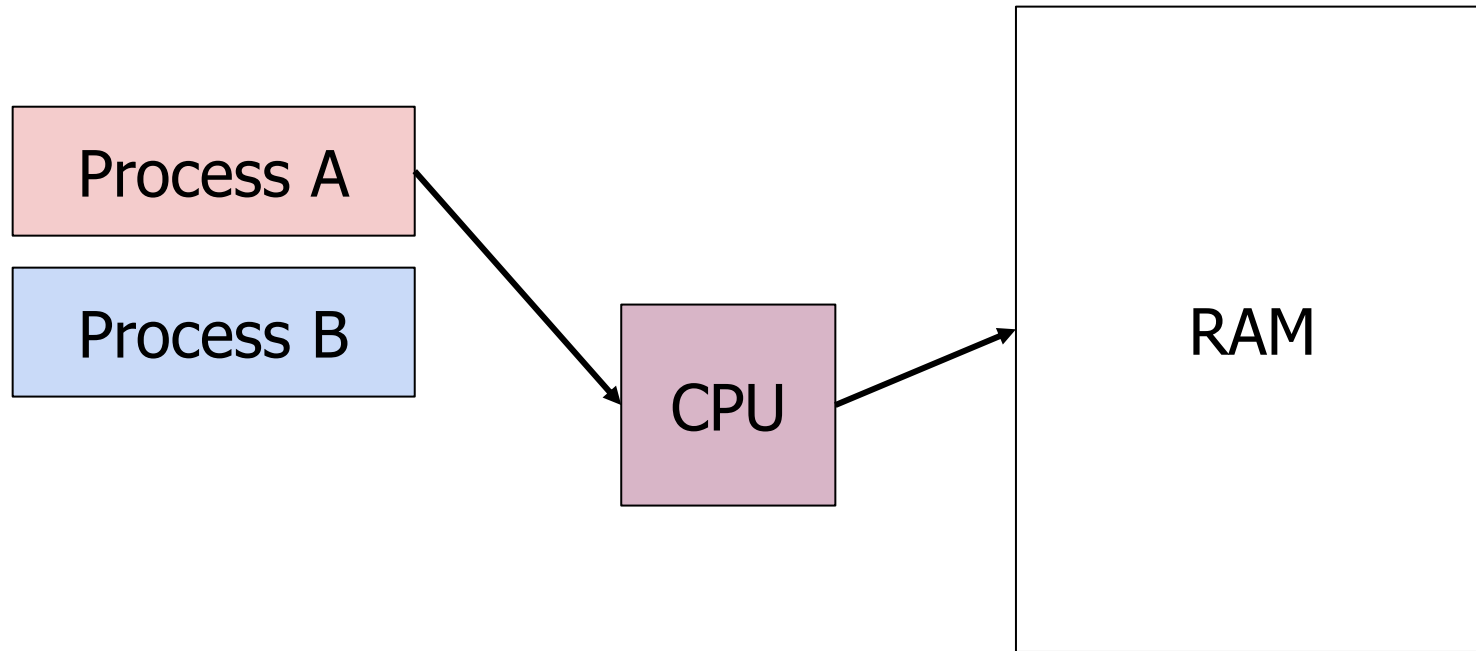
The Reality!



Memory problems

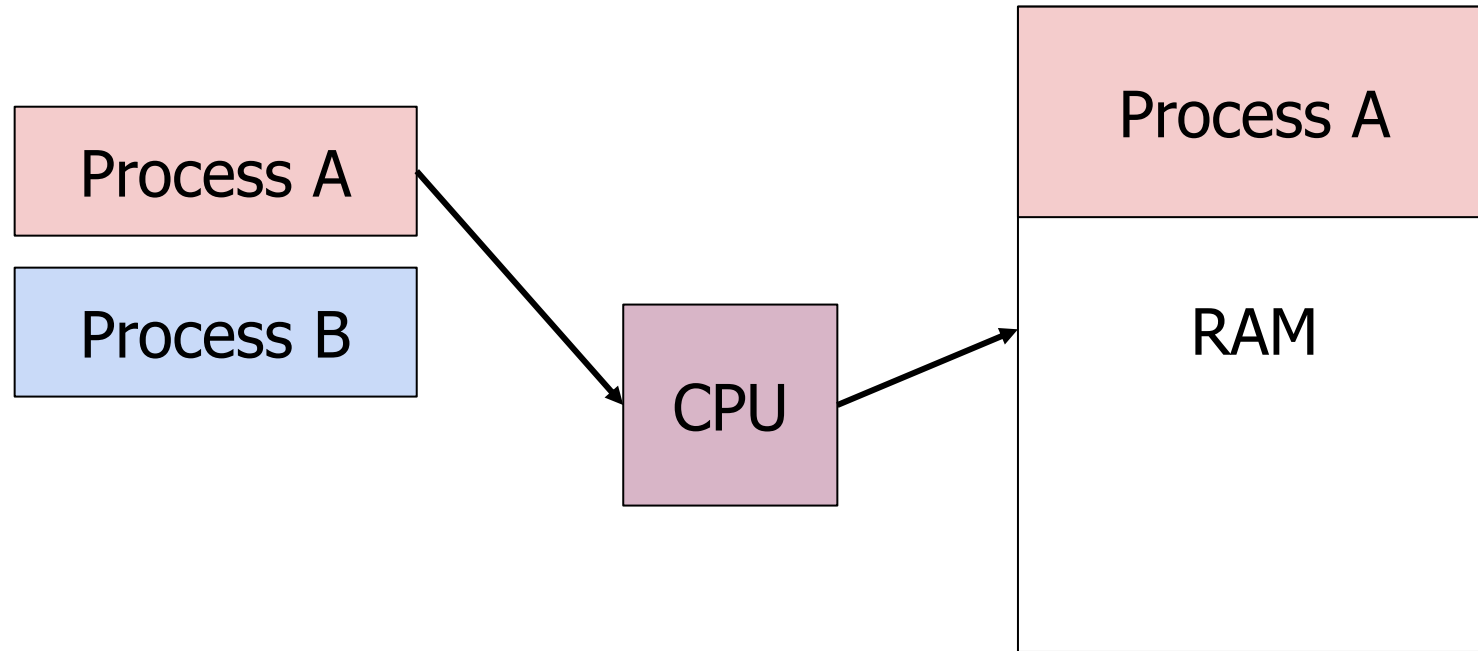
- What problems does virtual memory solve?
 - 1. Which addresses does each process get?**

Problem: How do multiple applications share RAM?

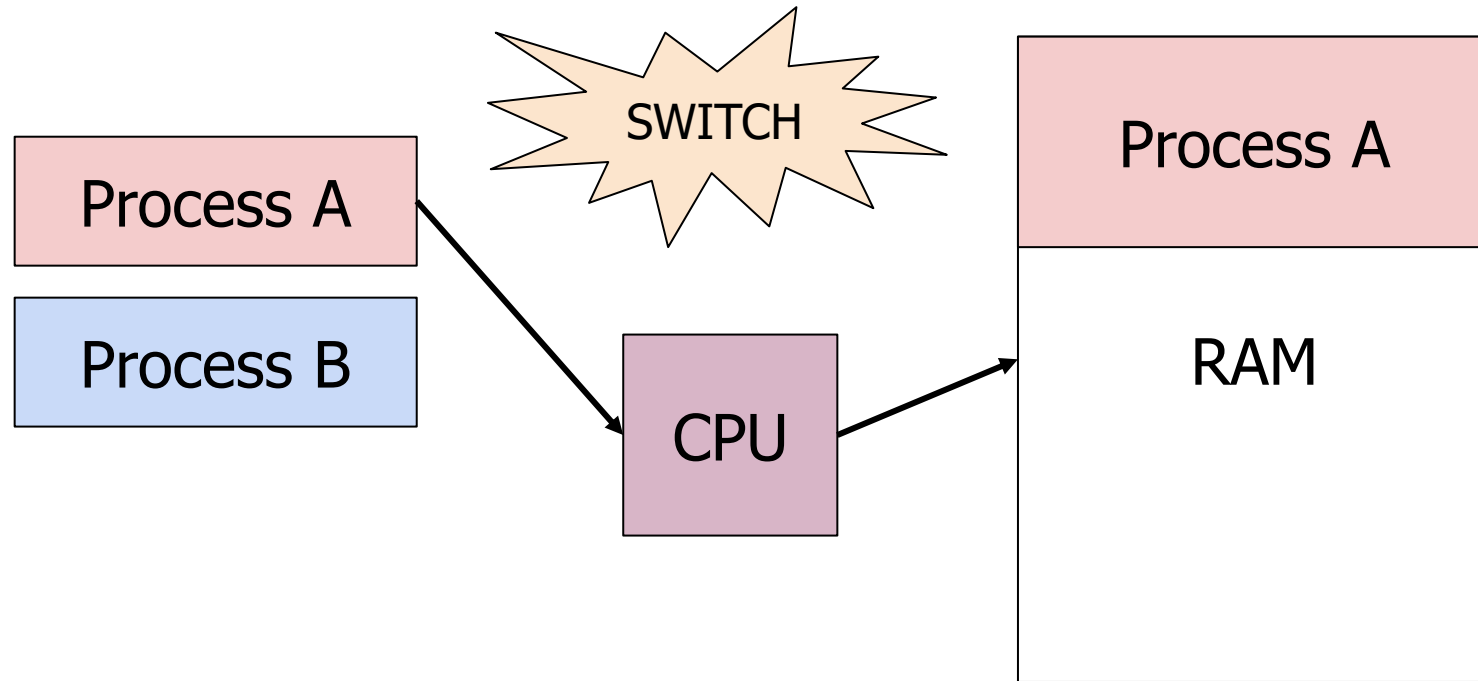


Both processes assume they start at the beginning of RAM and use as much as they need

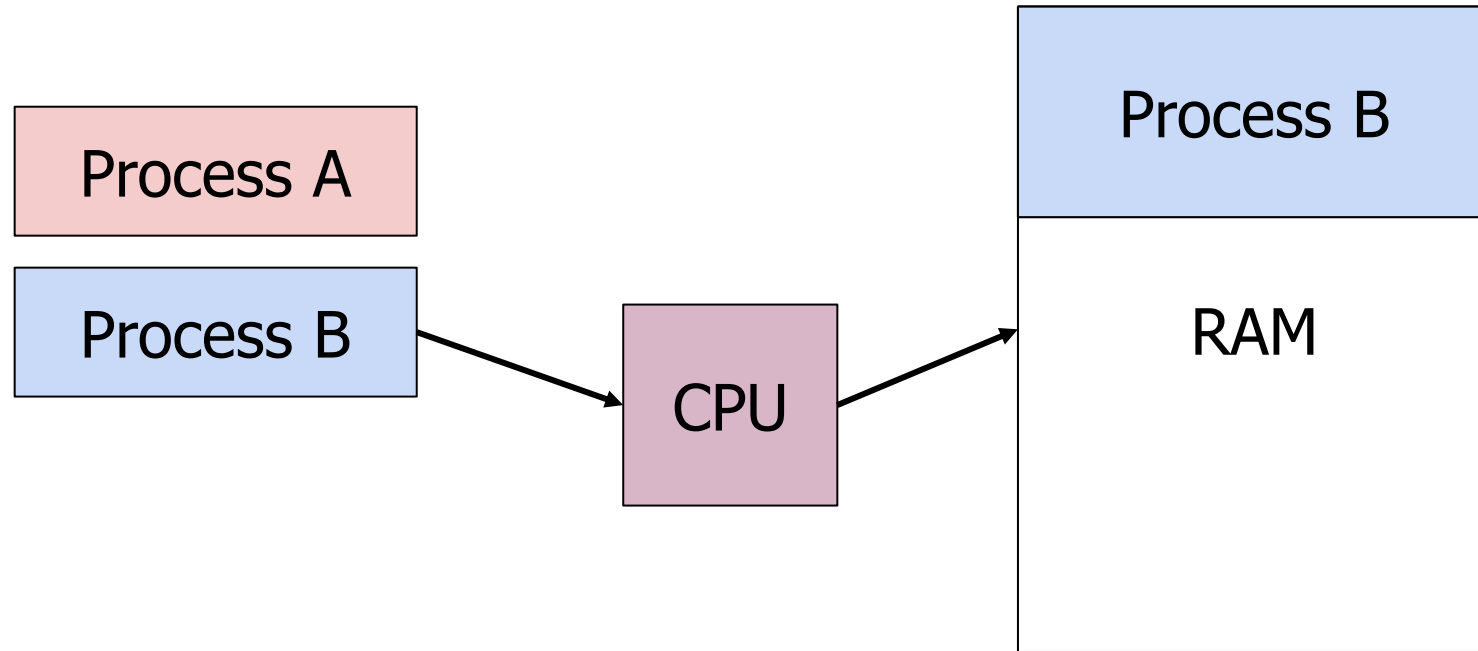
Problem: How do multiple applications share RAM?



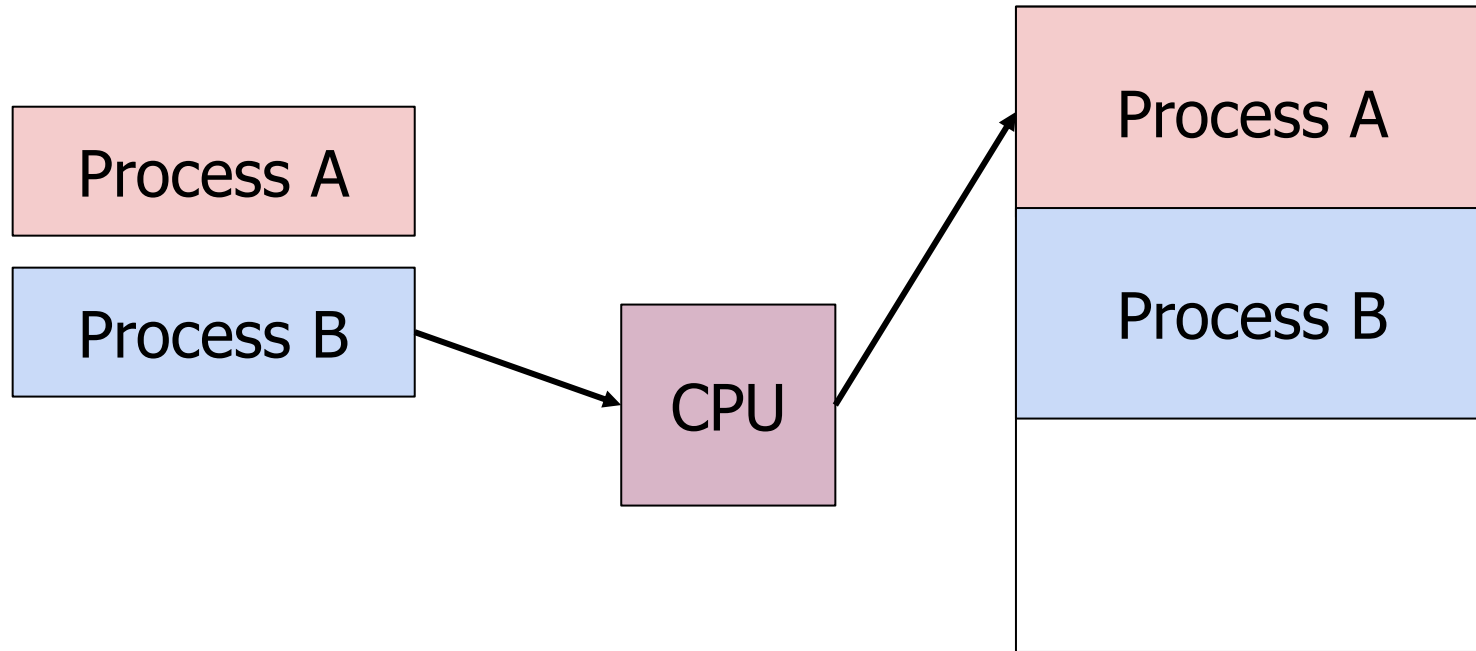
Problem: How do multiple applications share RAM?



Problem: How do multiple applications share RAM?



Problem: How do multiple applications share RAM?

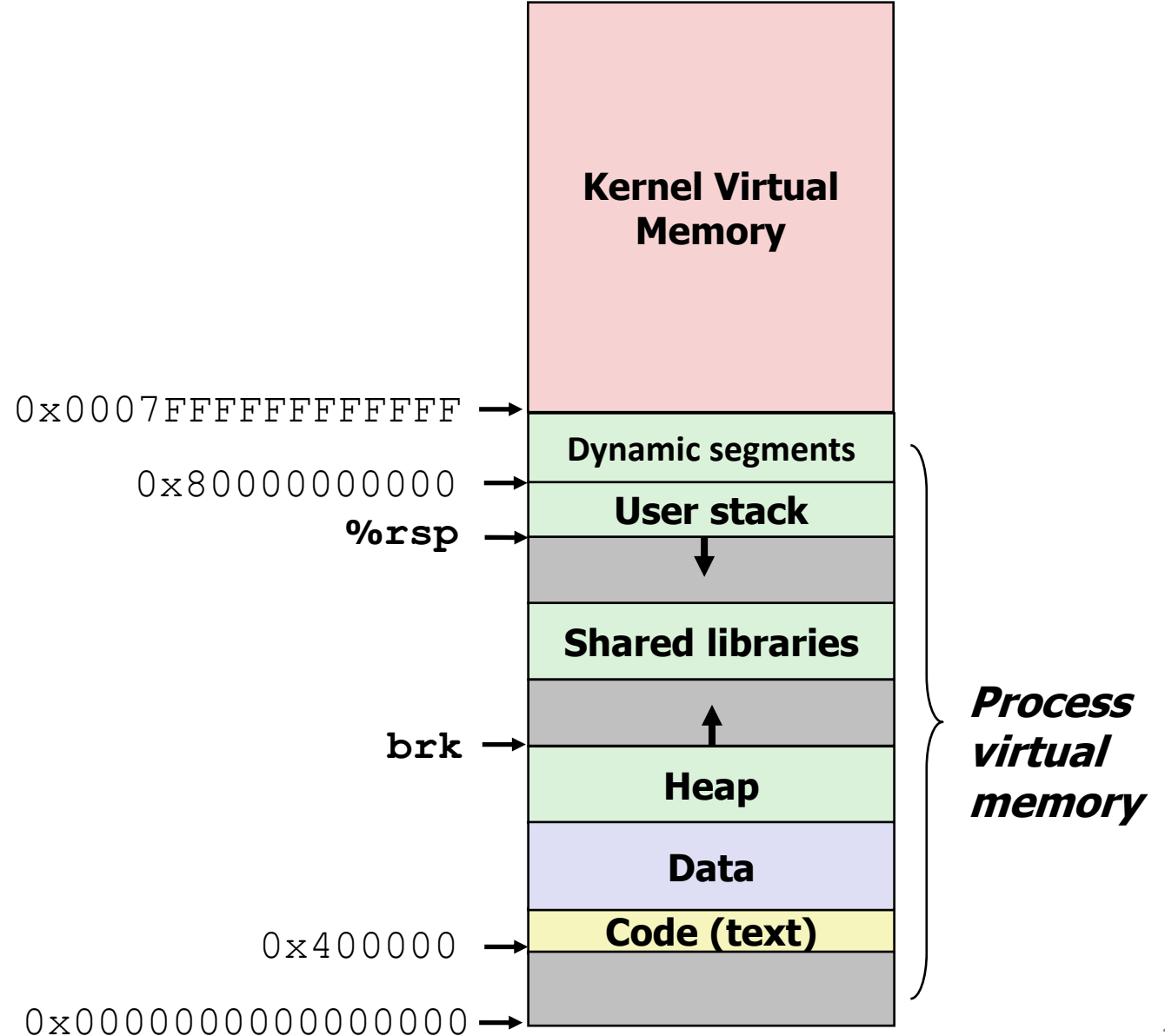


There's enough RAM for both. Why should we have to swap?

Challenge here is that programs are compiled with specific addresses...

Solution: virtual addresses allow RAM sharing

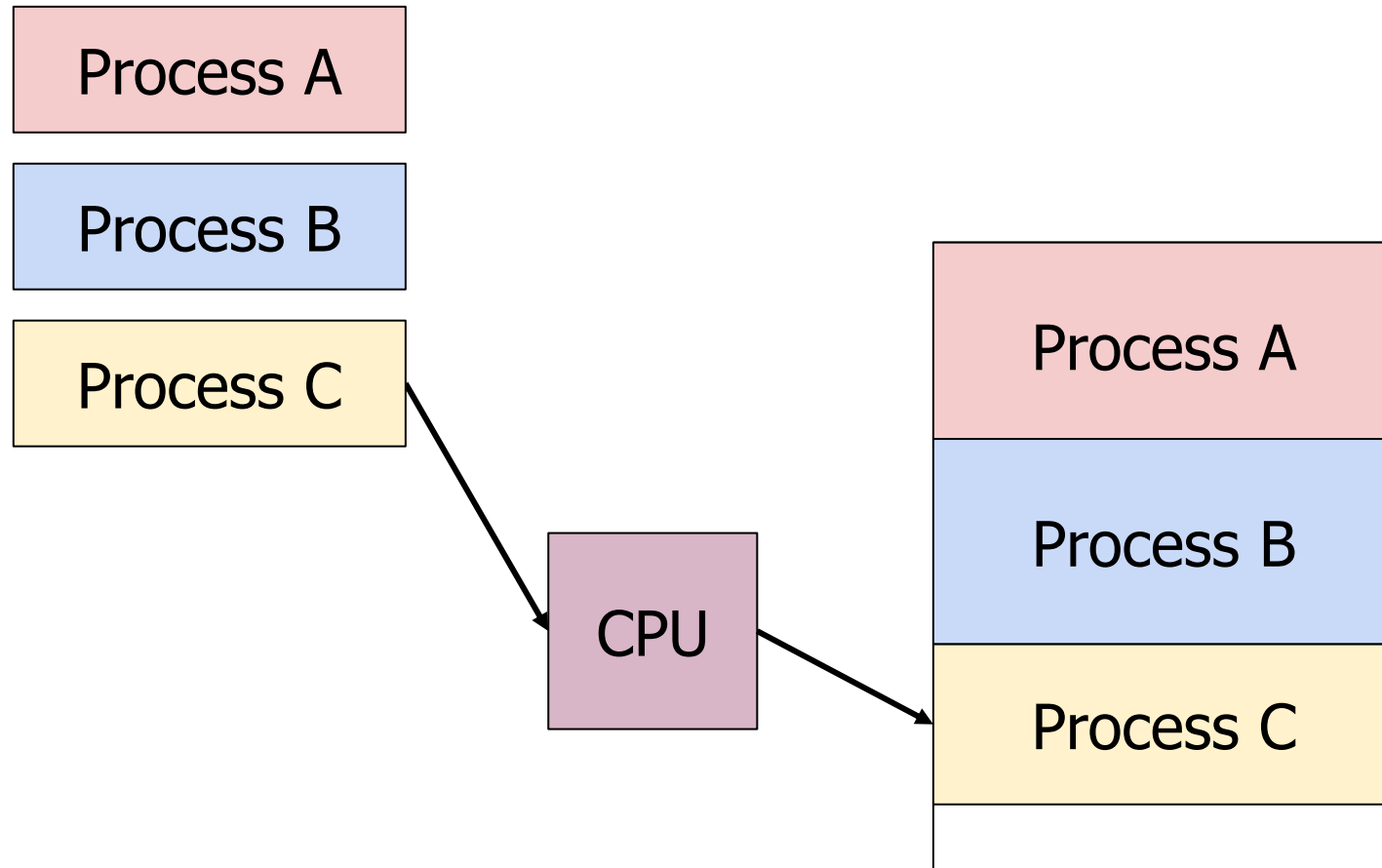
- Programs can use whatever virtual addresses they want
 - Usually, there's a fixed mapping for a given OS
- OS controls physical addresses
 - Decides which parts of RAM are used for which things



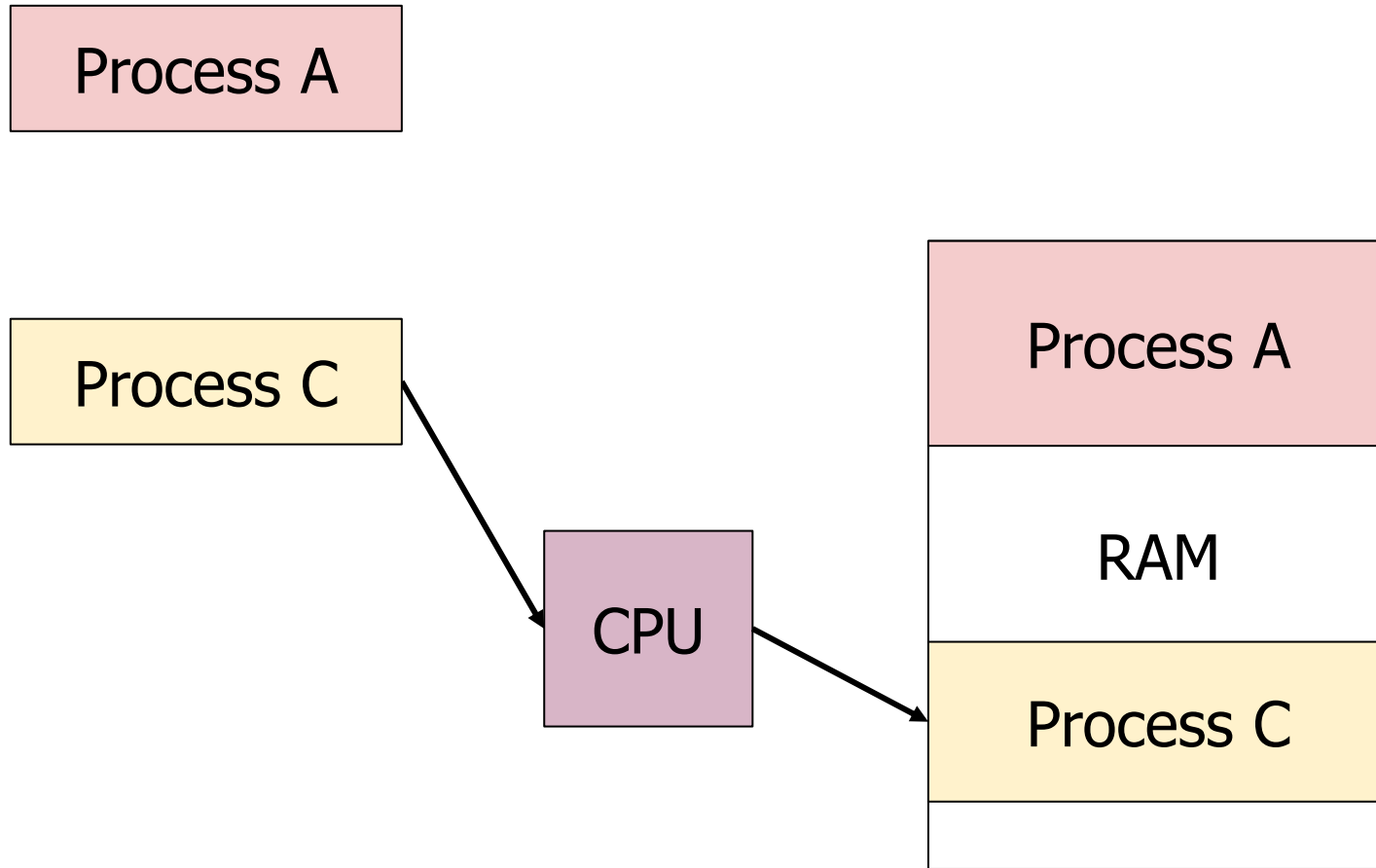
Memory problems

- What problems does virtual memory solve?
 1. Which addresses does each process get?
 - 2. How do we deal with fragmentation?**

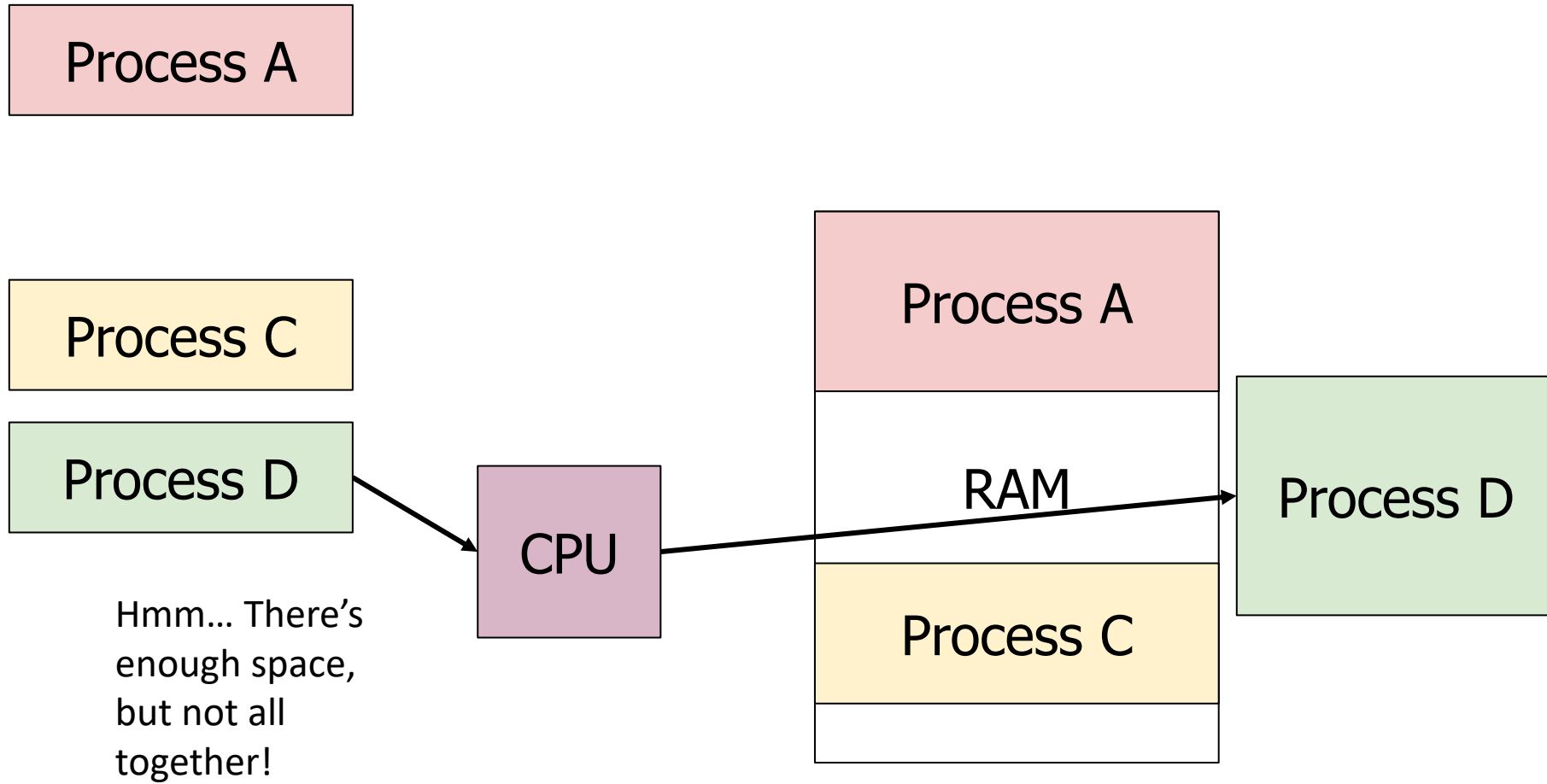
Problem: memory fragmentation



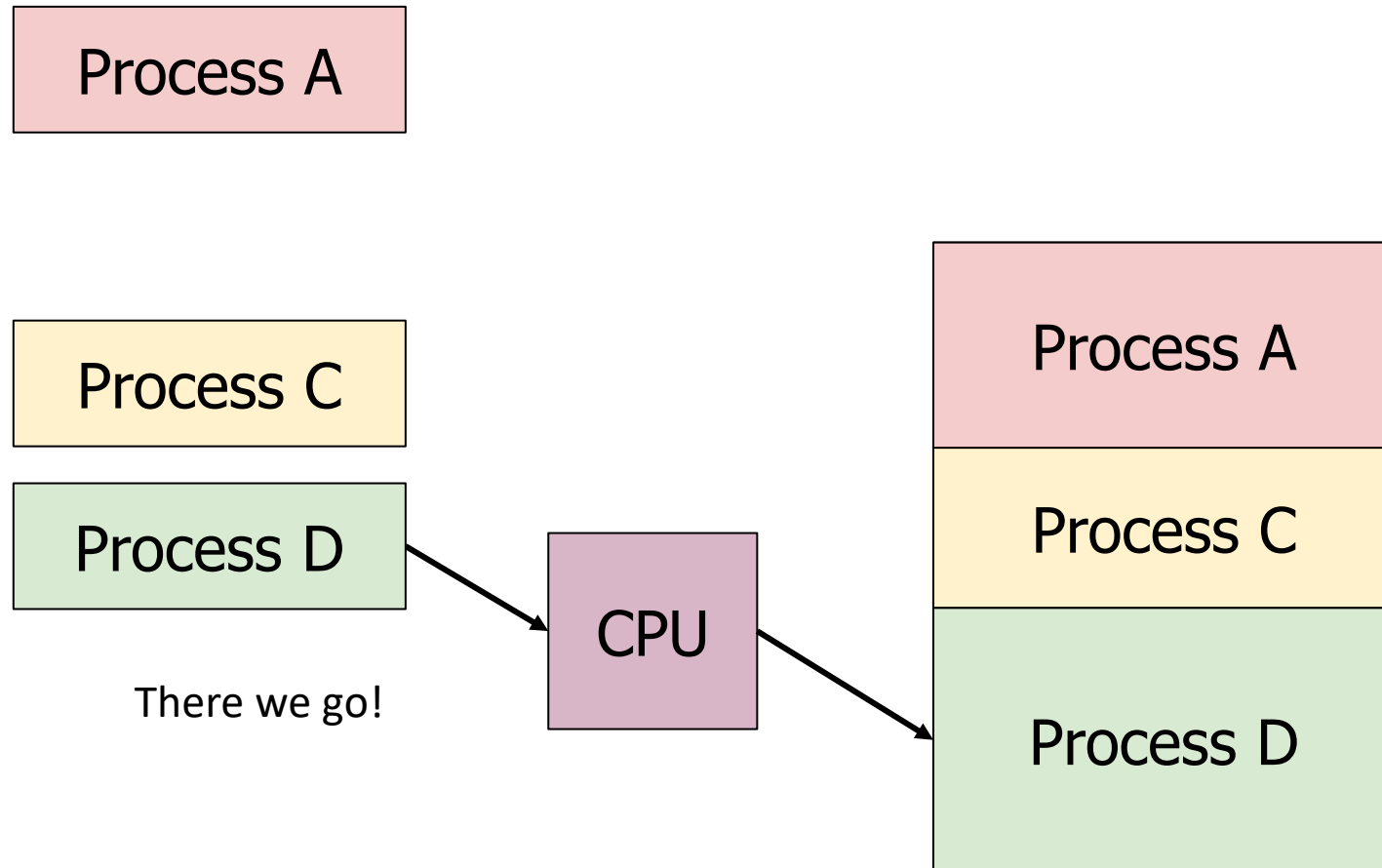
Problem: memory fragmentation



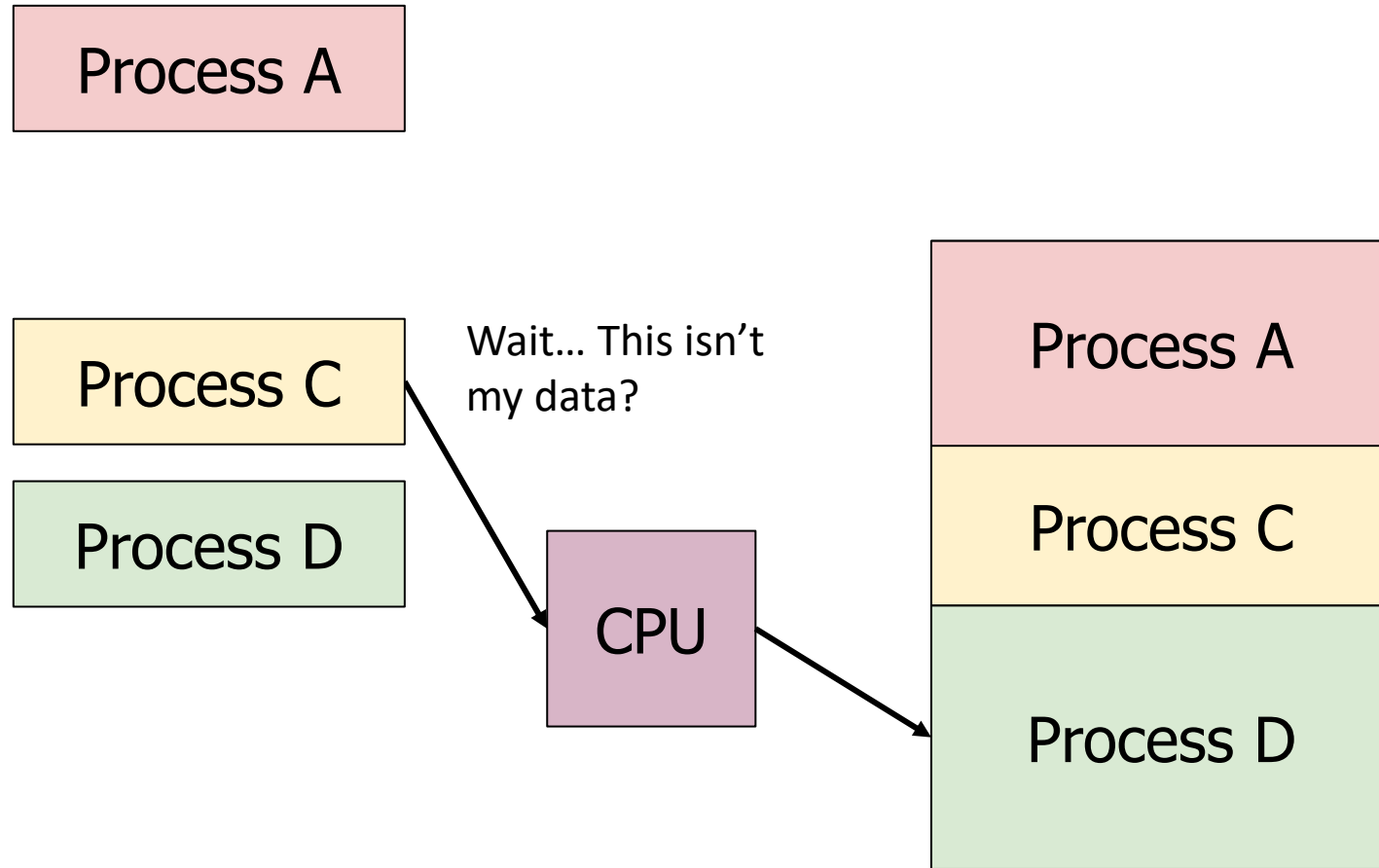
Problem: memory fragmentation



Problem: memory fragmentation

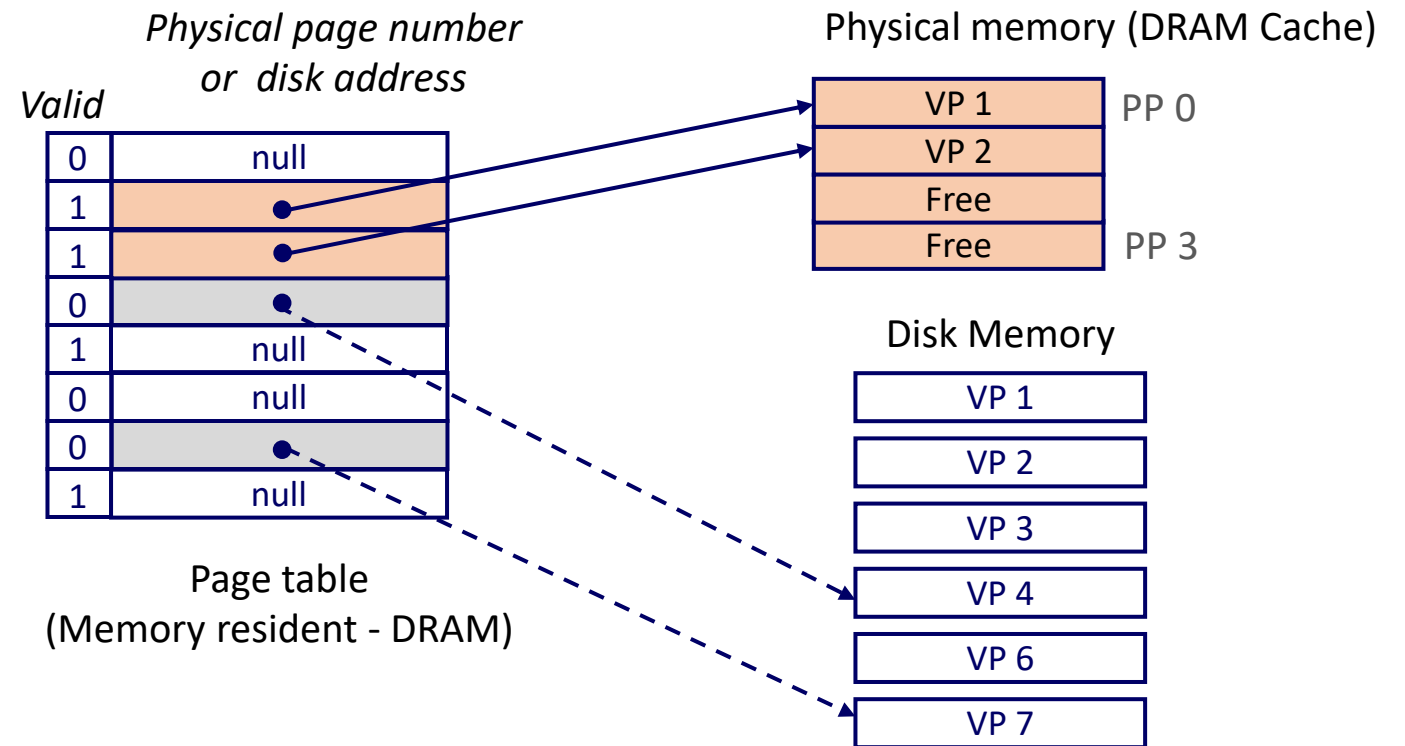


Problem: memory fragmentation



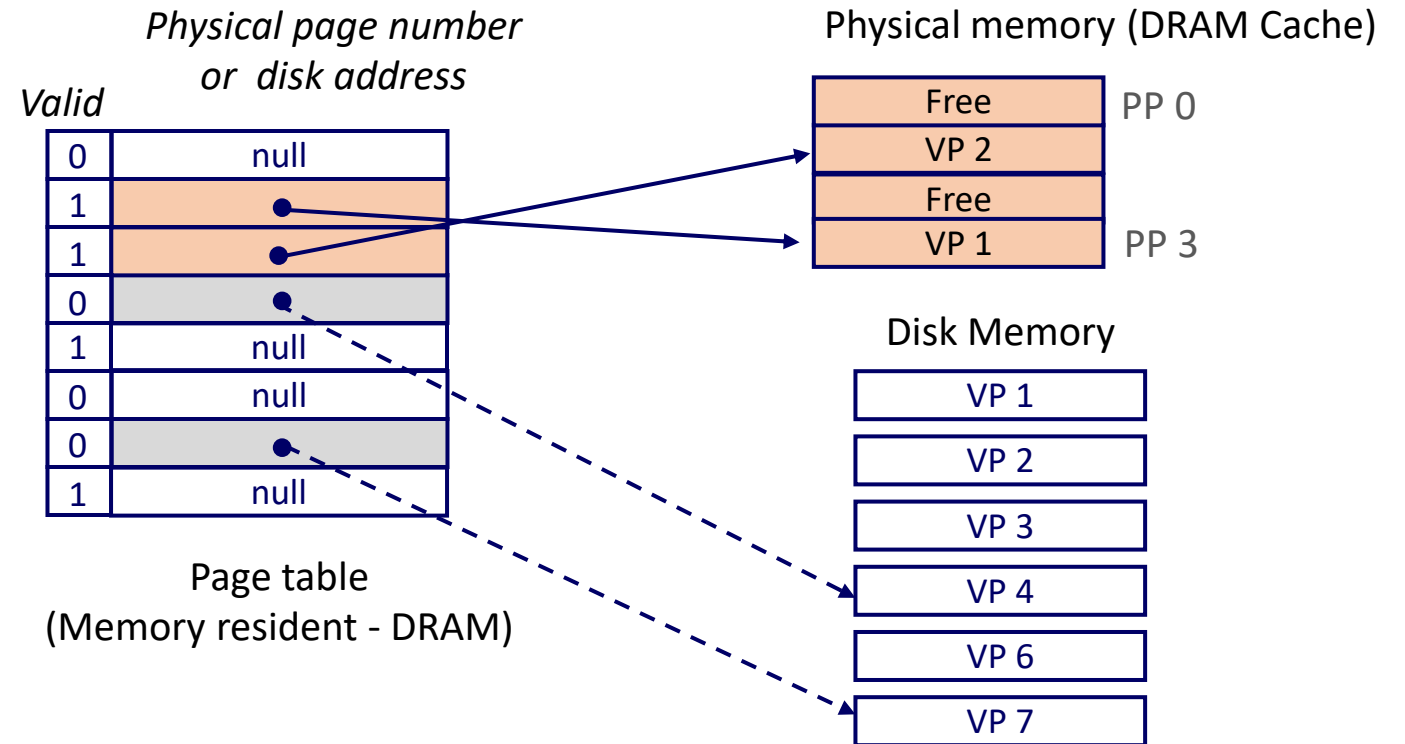
Solution: page tables allow for memory to be noncontiguous

- Entries for virtual pages *can* use contiguous pages of physical memory in order
 - And often do by default



Solution: page tables allow for memory to be noncontiguous

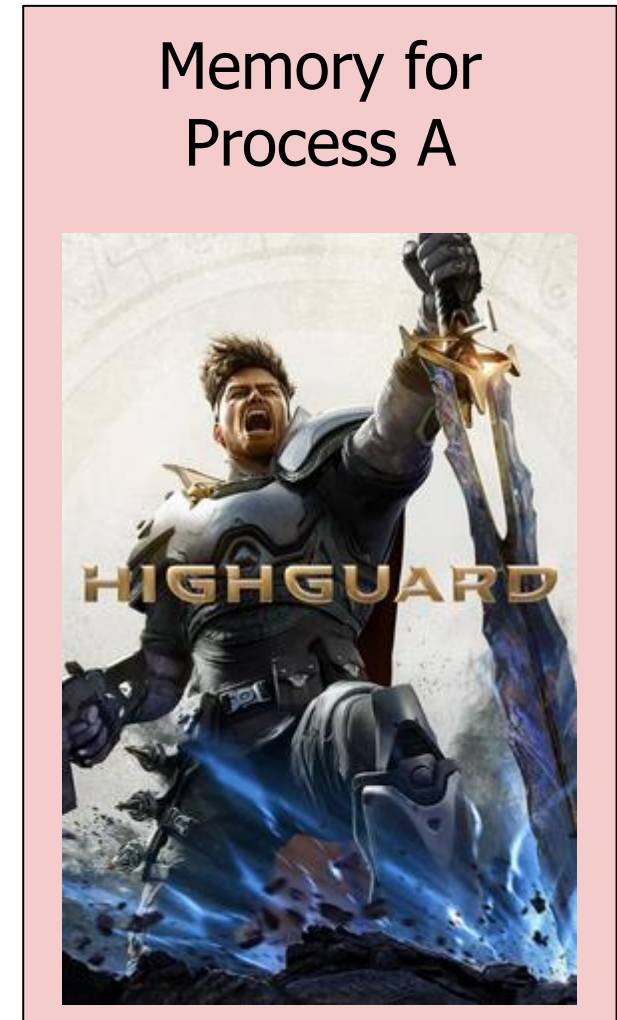
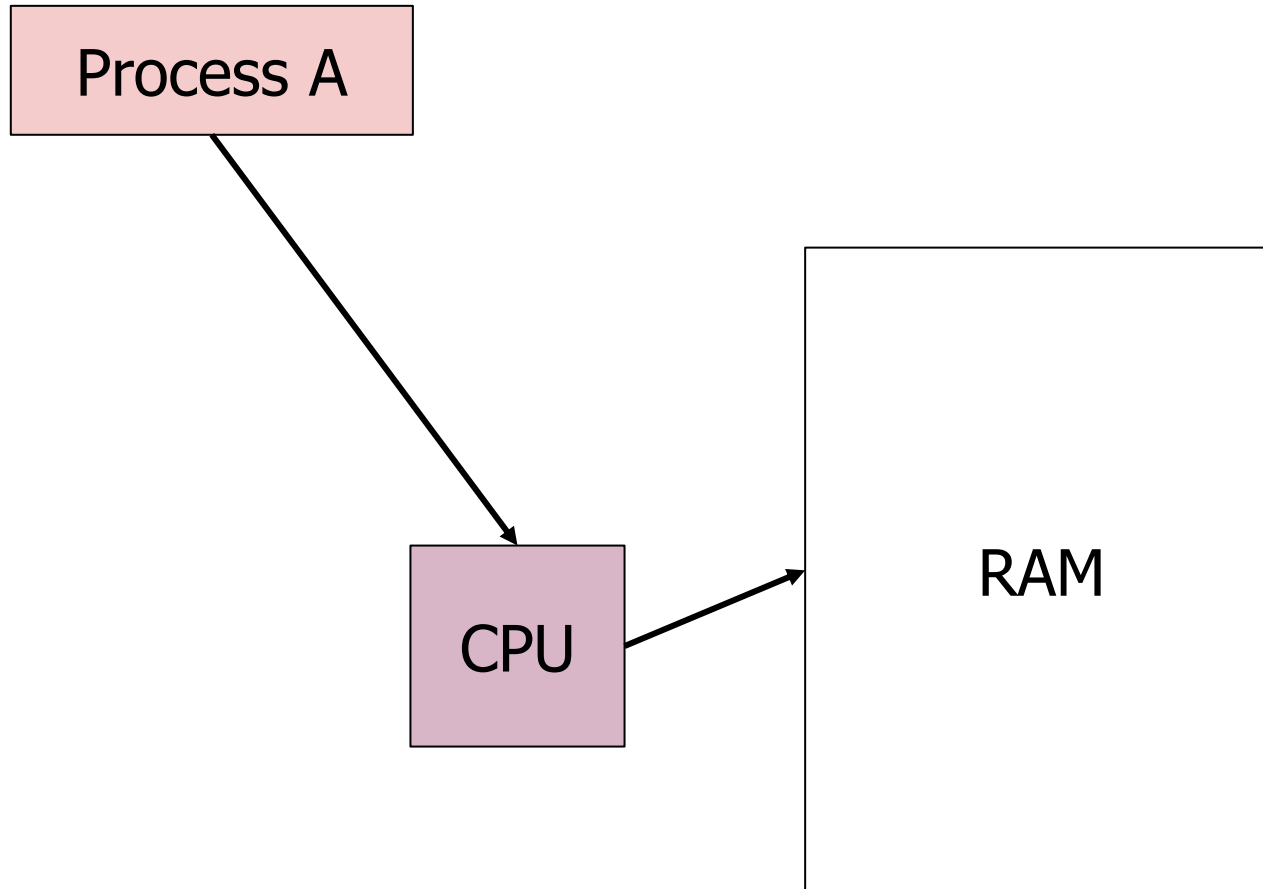
- But virtual pages can be mapped to any physical pages
 - Noncontiguous
 - Out-of-order
- Mappings can also be changed later if memory moves locations
 - Happens rarely



Memory problems

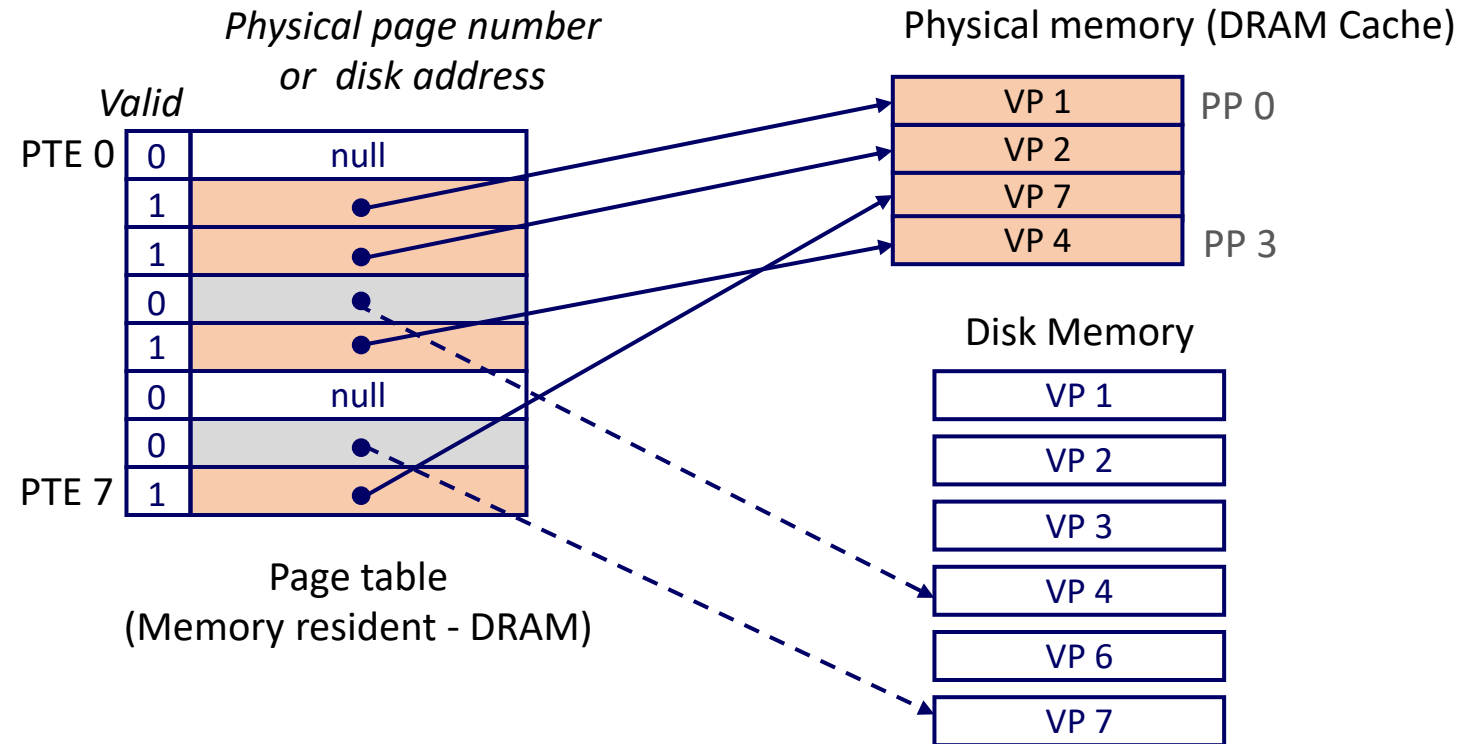
- What problems does virtual memory solve?
 1. Which addresses does each process get?
 2. How do we deal with fragmentation?
 - 3. How do we support processes bigger than RAM?**

Problem: processes might be bigger than RAM



Solution: some pages can be left on disk

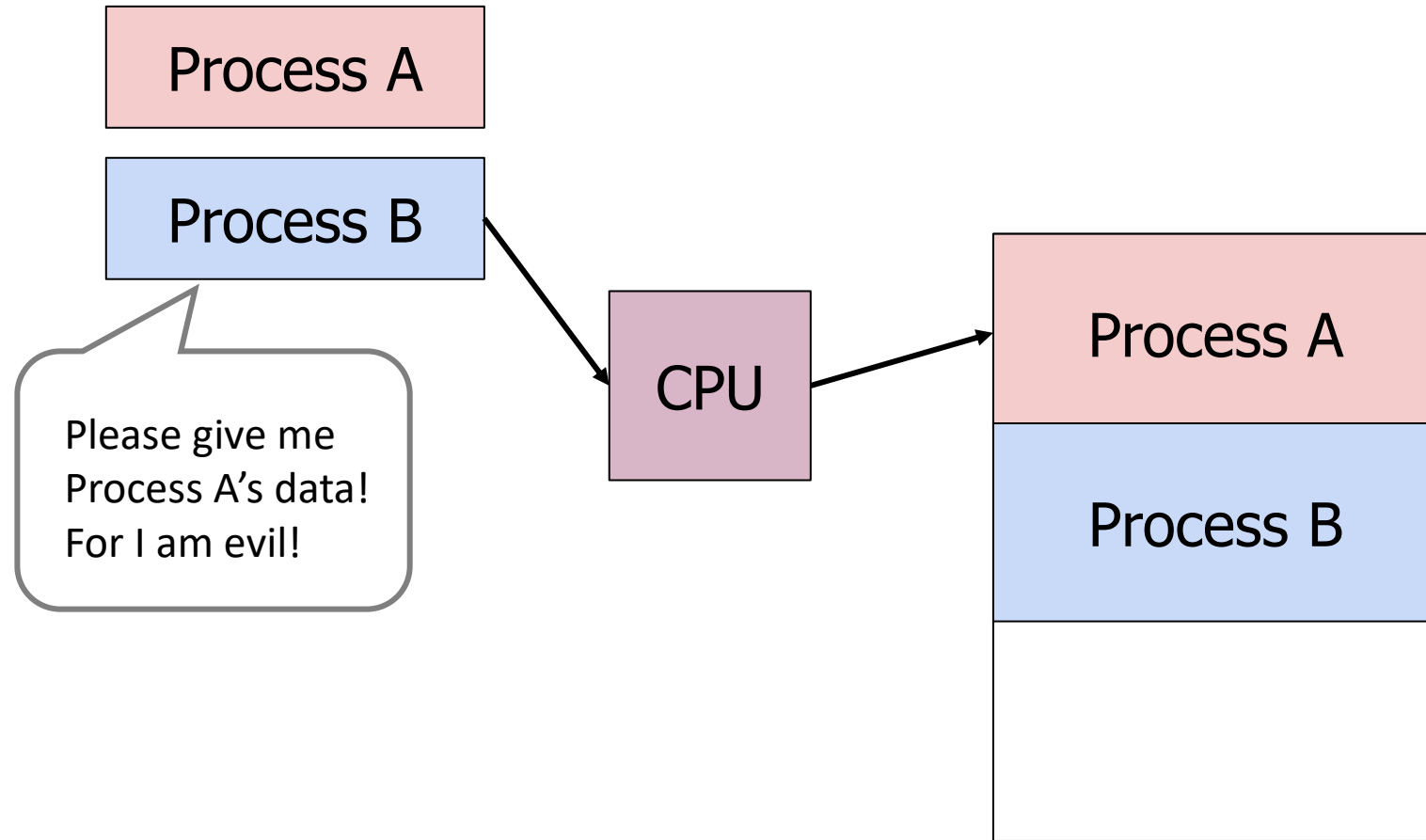
- Just leave some pages for that process on disk
- Page table entry still exists for each virtual page
- Hopefully working set is smaller than program memory



Memory problems

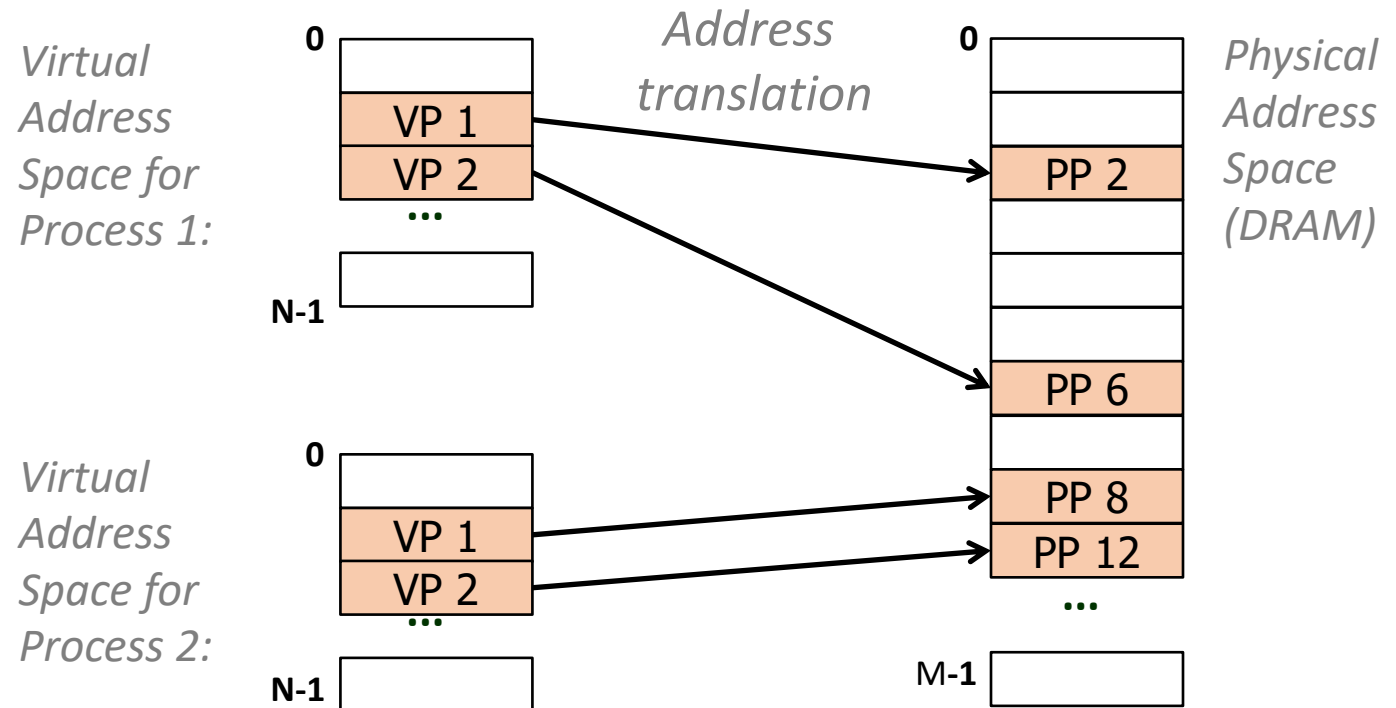
- What problems does virtual memory solve?
 1. Which addresses does each process get?
 2. How do we deal with fragmentation?
 3. How do we support processes bigger than RAM?
 - 4. How do we protect processes from each other?**

Problem: processes can't be trusted



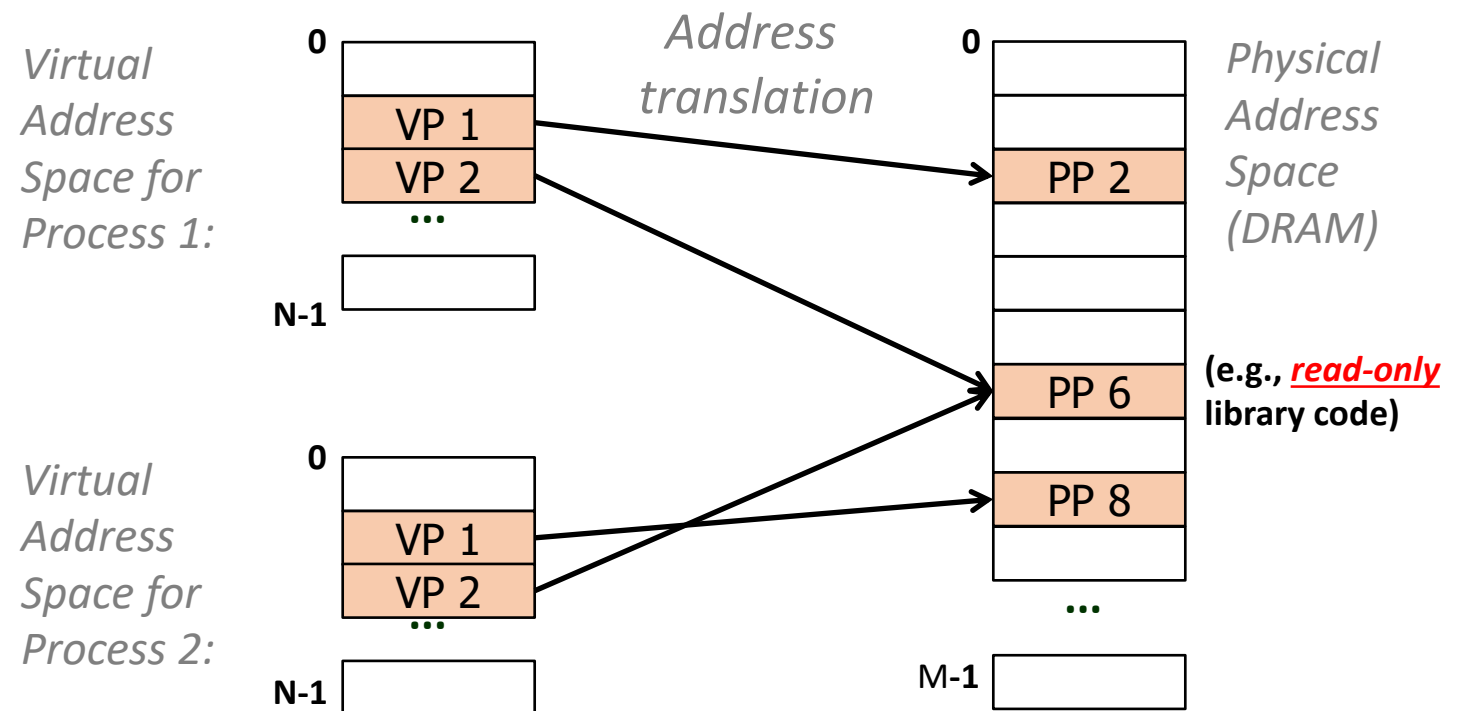
Solution: virtual memory isolates process memory

- Each process has separate virtual memory spaces
 - No way to access another process's physical memory unless it is mapped to one of your virtual addresses



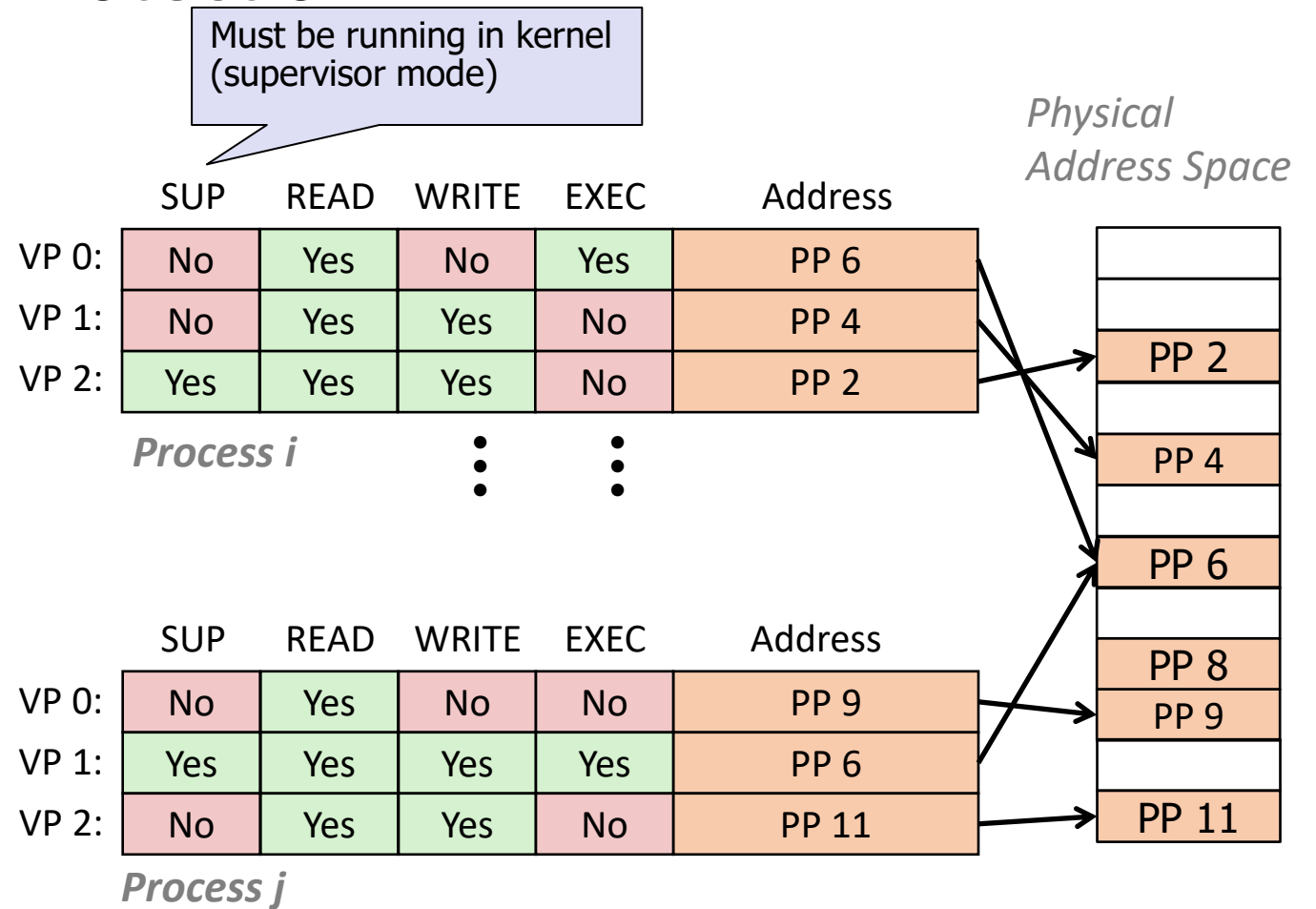
Virtual memory can still share memory if desired

- We could share some physical pages across processes to enable shared libraries or shared memory



VM as a Tool for Memory Protection

- What if we want better protection?
- Extend Page Table Entries with permission bits!
- HW enforces this protection
 - Faults process if a violation occurs



Memory problems

- What problems does virtual memory solve?
 1. Which addresses does each process get?
 2. How do we deal with fragmentation?
 3. How do we support processes bigger than RAM?
 4. How do we protect processes from each other?
 - 5. How do we deal with how incredibly slow disk is?**

Computing timescales

- Assuming 4 GHz processor, **Instruction (with registers): 0.25 ns**

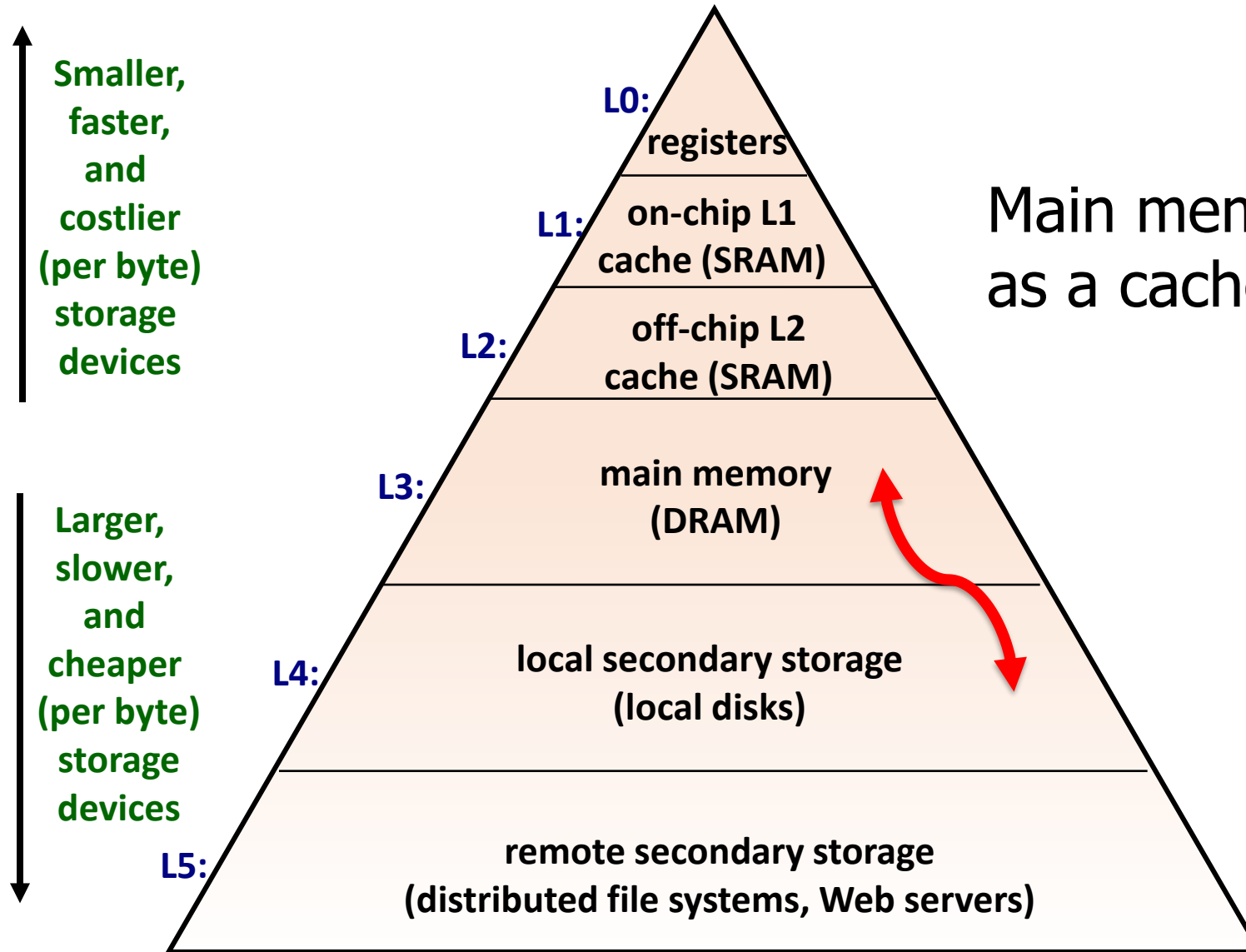
Jeff Dean (Google AI): “Numbers Everyone Should Know”

Jim Gray’s analogy:

- Registers are in your apartment
- Disk is on Mars

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Caching disks



Main memory should act as a cache for disk!

VM as a Tool for Caching

- We use physical memory as a sort of *cache!* (called: DRAM cache)
 - Store the bulk of your data on disk (very large, very cheap, but very slow)
 - And store the currently-used data in main memory (very fast by comparison)
 - Get the best of both worlds! Large capacity and fast access!
- DRAM cache organization driven by the *enormous* miss penalty
 - DRAM is about **100x** slower than SRAM
 - Disk is about **100,000x** slower than DRAM

Picking Cache Design Parameters

- Block size?
 - Disks are better at transferring large chunks of data (the first byte incurs a long delay, the rest come really fast afterwards)
 - Large block size: typically 4 KB -> these are memory “pages”
- Associativity?
 - DRAM cache misses incur enormous penalties; have to go to disk. Yikes!
 - Associativity is high to minimize miss rate
 - Fully associative (one huge set): any block can go anywhere in cache
 - Requires a “large” mapping function – but managed in software, so ok

Problem: most things are NOT in RAM

- Disk is MUCH larger than RAM is, so most data will not actually be in RAM
- But handling Page Faults takes a long time
 - Has to read a page of memory from disk
- So how is our system not incredibly slow?
 - Locality to the rescue!

Locality saves the day (as usual)

- At any point in time, programs tend to access a small set of active virtual pages called the **working set**
 - Programs with higher temporal locality will have smaller working sets
- If (working set size < main memory size)
 - High performance for one process after compulsory misses (i.e., the program is loaded)
 - Any page can go anywhere in RAM, so no conflicts. Only capacity matters.
 - Life is good!
- If (SUM(working set sizes) > main memory size)
 - **Thrashing**: Performance meltdown where pages are swapped to and from disk continuously
 - When cache memory is thrashing, CPU runs at the speed of memory. Ow.
 - When virtual memory is thrashing, CPU runs at the speed of disk. Yikes!
 - Hope you enjoy the commute to Mars. Because that's where your data is

Review

- What problems does virtual memory solve?
 1. Which addresses does each process get?
 2. How do we deal with fragmentation?
 3. How do we support processes bigger than RAM?
 4. How do we protect processes from each other?
 5. How do we deal with how incredibly slow disk is?
- Virtual memory addresses all of these problems!

Break + Conceptual Questions

1. When a process on an x86-64 platform accesses memory at an address, how do you know if that address is virtual or physical?
2. What changes for a process if a page is on disk rather than in memory?

Break + Conceptual Questions

1. When a process on an x86-64 platform accesses memory at an address, how do you know if that address is virtual or physical?
 - Processes always use virtual addresses
 - Only the OS can use physical addresses

2. What changes for a process if a page is on disk rather than in memory?
 - Performance is much worse
 - But otherwise the code works exactly the same
 - Desired page is loaded into memory and then the access resumes
 - This has certainly happened to your code and you haven't noticed

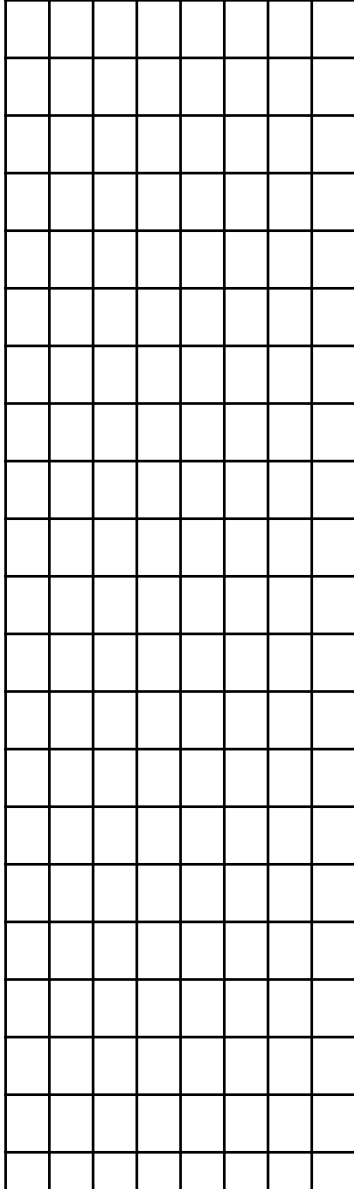
Outline

- Virtual Memory Concept
- Virtual Memory Process
- Solving Memory Problems with Virtual Memory
- **Address Translation**
- Virtual Memory Summary

Address translation visually

Virtual Memory

address 0x0

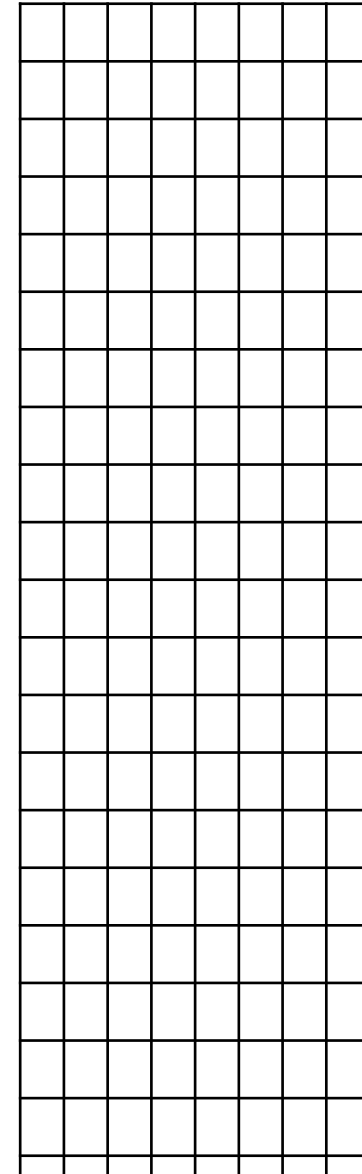


increasing
addresses ↓

Mapping every byte individually
would be too much bookkeeping

Physical Memory

address 0x0

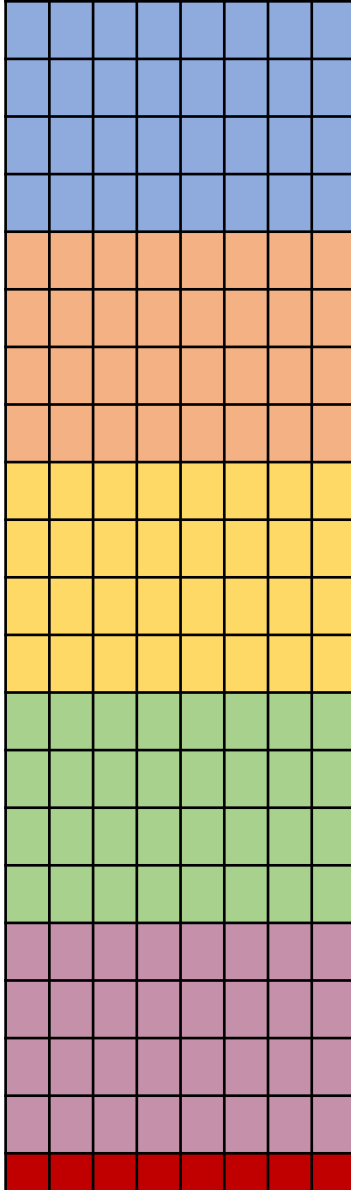


increasing
addresses ↓

Collecting memory into pages

Virtual Memory

address 0x0



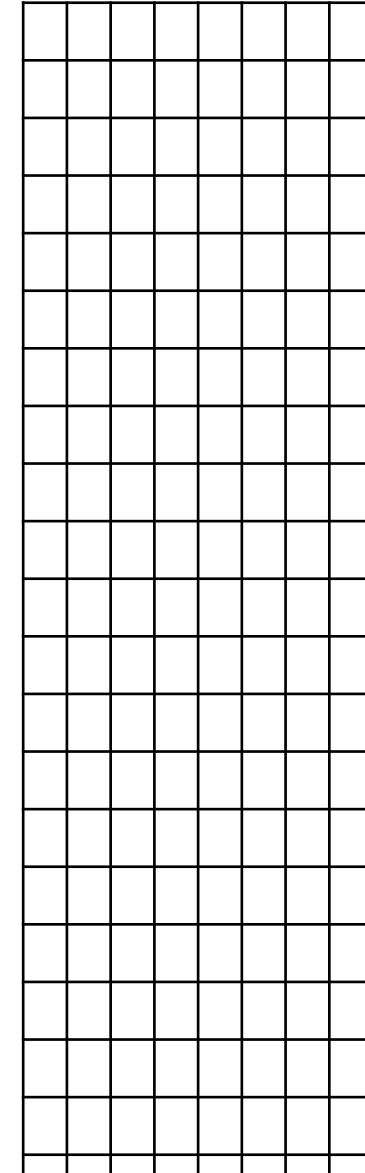
Instead, we collect memory into chunks called “pages”

Each page is exactly the same size (usually 4 kB = 4096 bytes in real systems)

Pages are aligned to a multiple of their size. So they slot in like lego bricks

Physical Memory

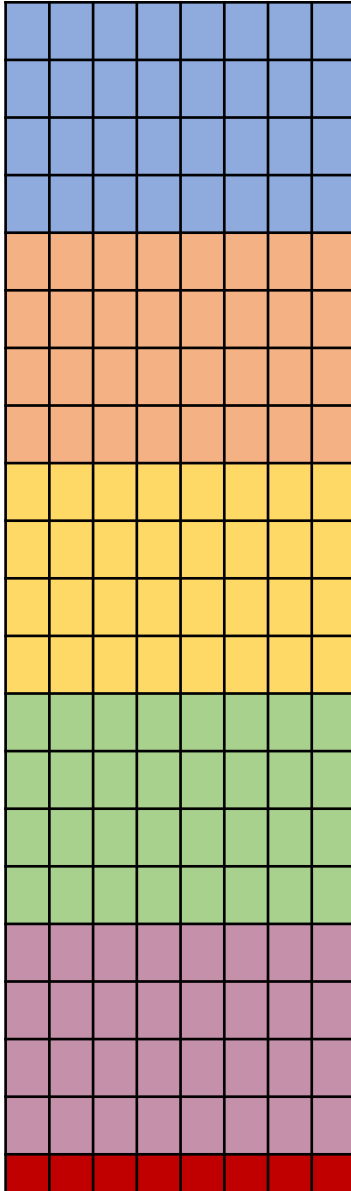
address 0x0



Mapping virtual pages to physical pages

Virtual Memory

address 0x0



increasing
addresses ↓

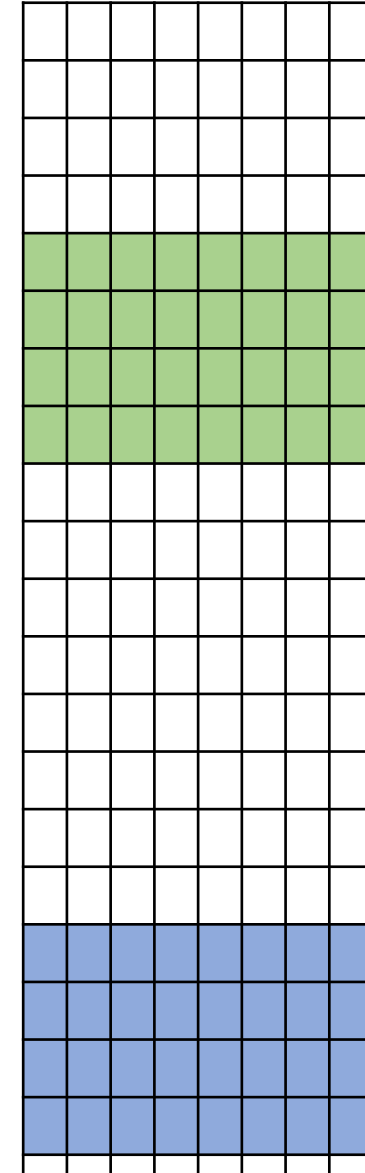
Each page of virtual memory will map to:

- A page of physical memory
- A page on disk
- Nowhere

In this example, the blue and green pages are in RAM

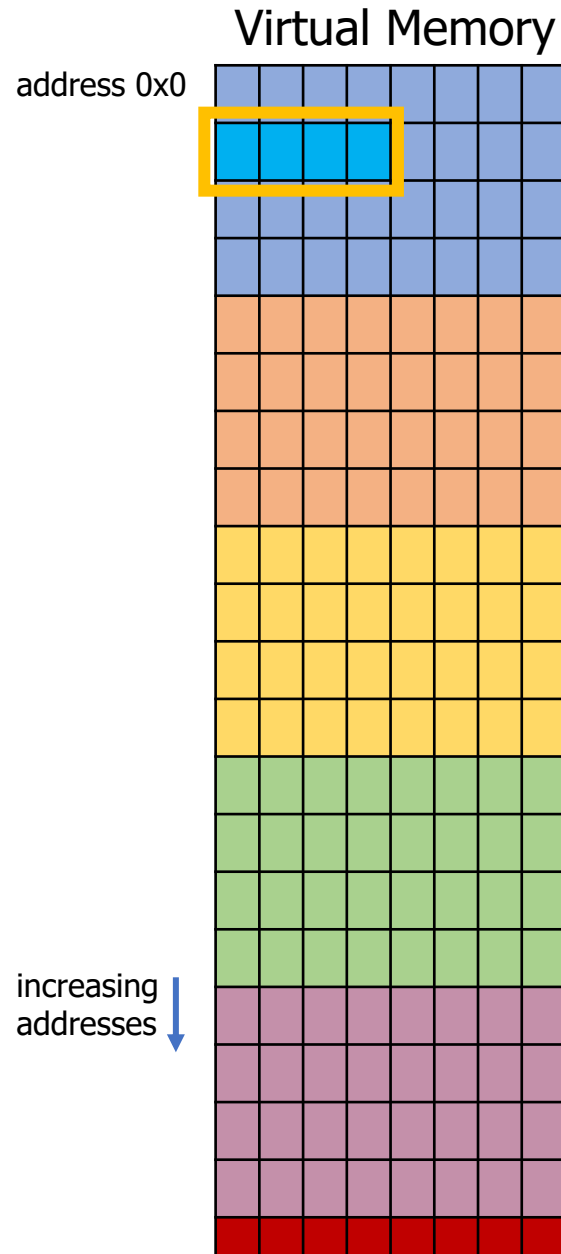
Physical Memory

address 0x0



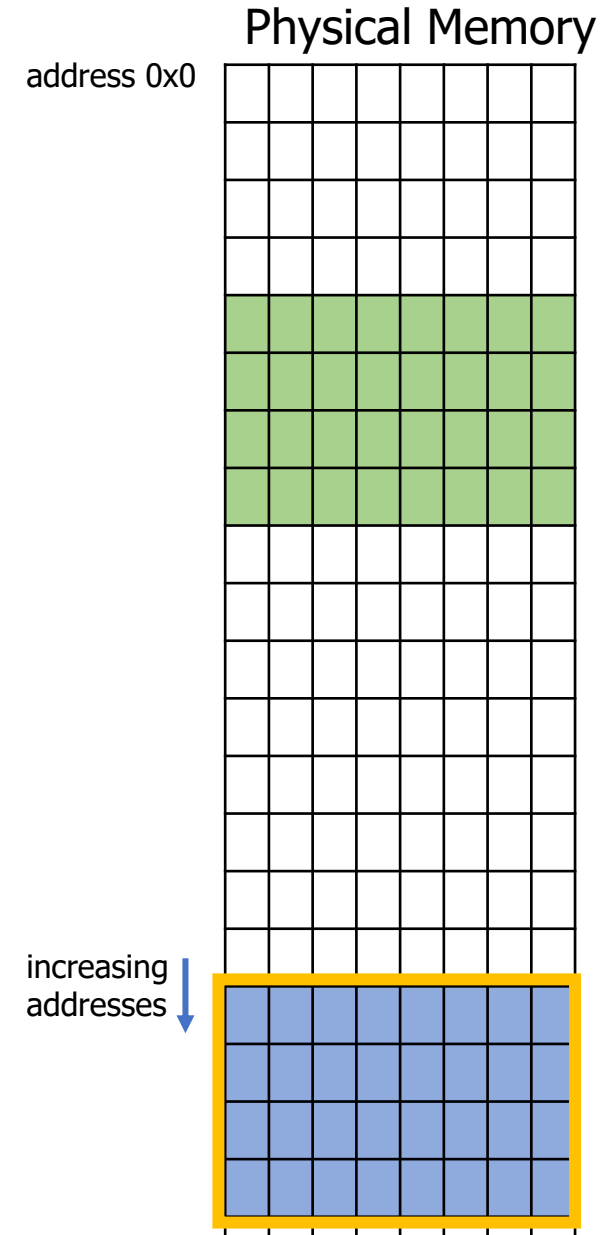
increasing
addresses ↓

Accessing individual bytes within a page

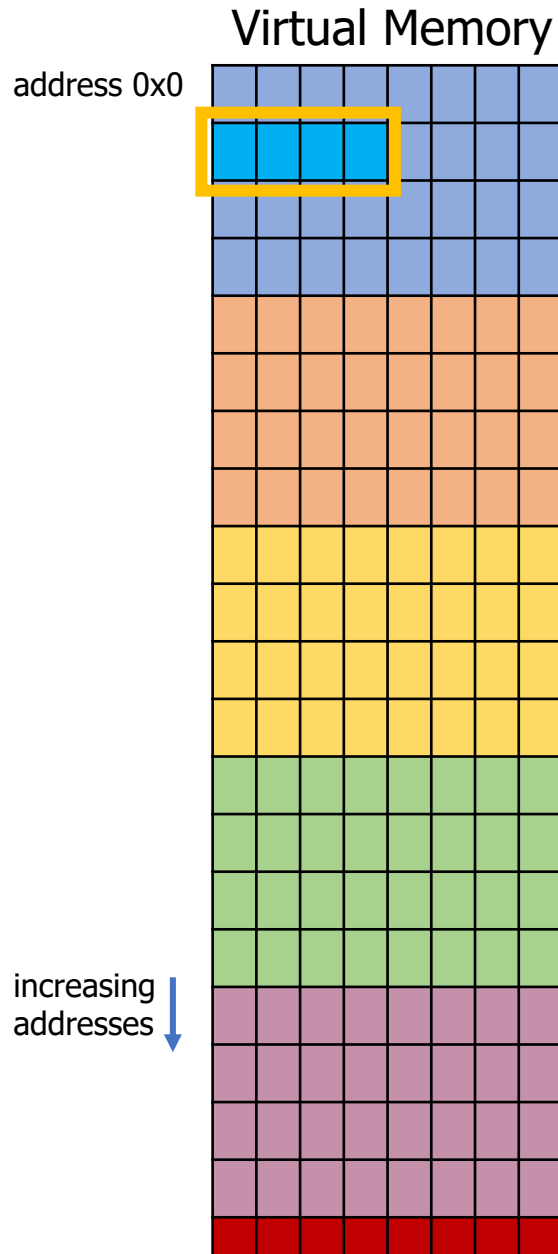


When accessing bytes within a page

First, you need to do the translation to figure out where that entire page is located



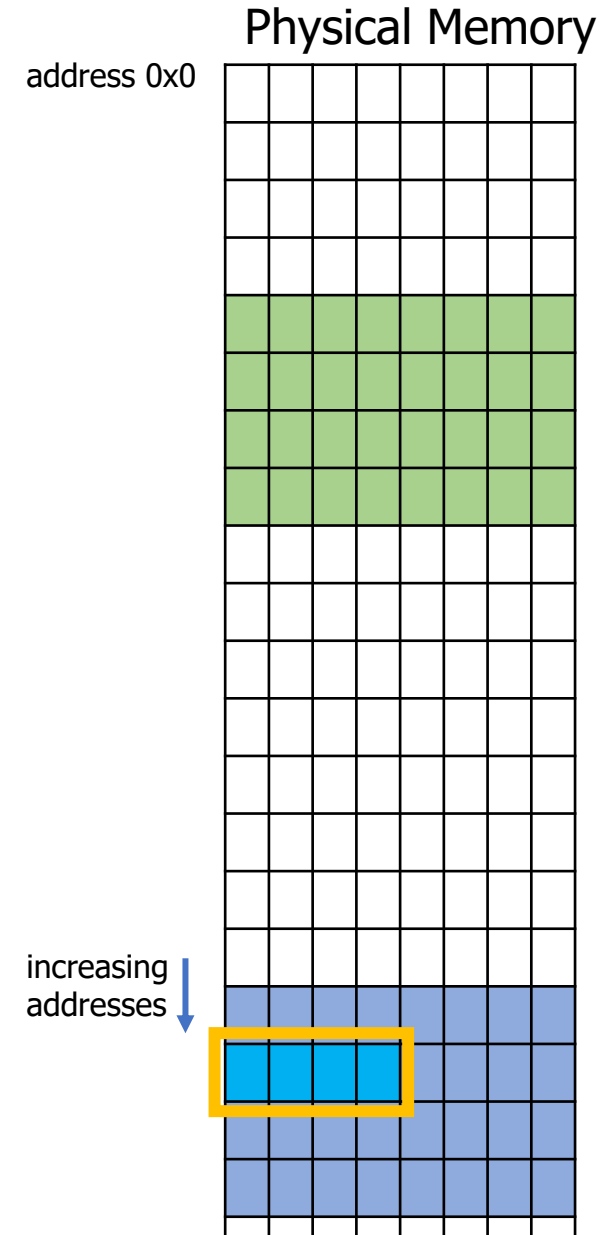
Accessing individual bytes within a page



When accessing bytes within a page

First, you need to do the translation to figure out where that entire page is located

Second, you access the data at the **same offset into the page** (if the mapping exists!)



Address Translation

- Goal: Given virtual address, find corresponding physical address
 - (Or get a page fault if the page is not in memory)
 - Translation done by Memory Management Unit (hardware)
 - But mapping itself is maintained by OS (software)
 - Just a table in memory!
- To do the actual translation, look at the address being accessed
 - Split it into parts, just like we did with Caches
 - Bottom bits of address: Page Offset (location of data within the page)
 - Top bits of address: Virtual Page Number (which page to access)

Breaking down virtual addresses

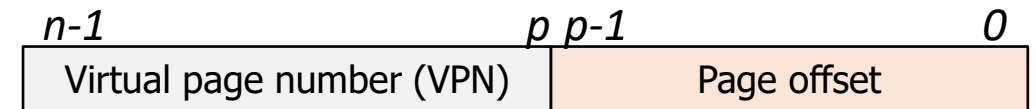
- Basic Parameters

- $\mathbf{N} = 2^n$: Number of addresses in virtual address space
- $\mathbf{M} = 2^m$: Number of addresses in physical address space. $m \leq n$ (usually much less)
- $\mathbf{P} = 2^p$: Page size (bytes)

- Components of the virtual address (VA)

- Virtual page number (VPN): $\mathbf{n-p}$ bits
- Page Offset: $\mathbf{p}$ bits

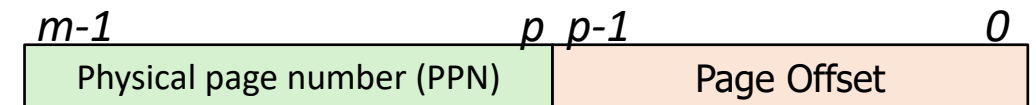
Virtual address (n bits total)



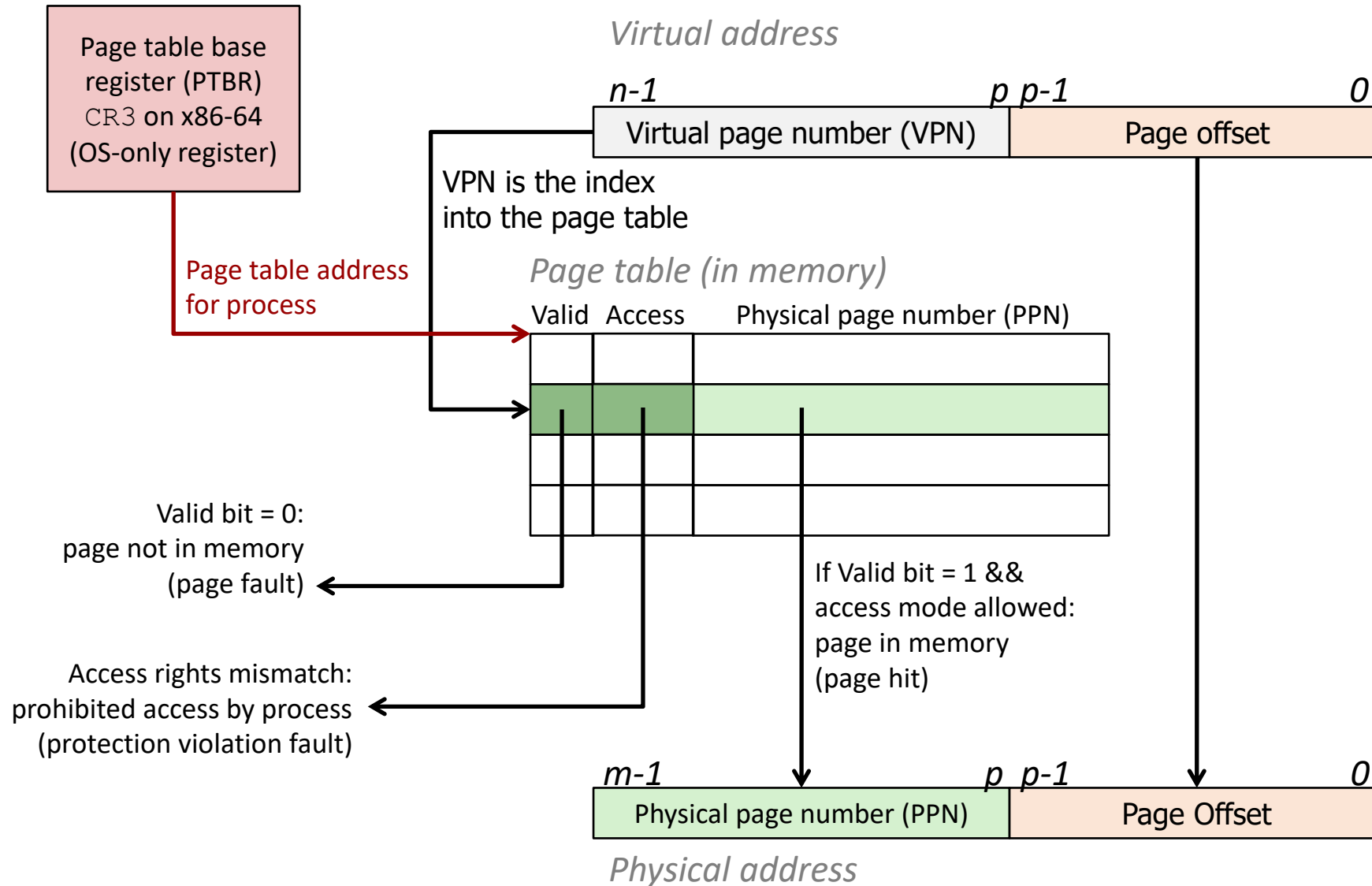
- Components of the physical address (PA)

- Physical page number (PPN): $\mathbf{m-p}$ bits
- Page Offset (same offset as VA): $\mathbf{p}$ bits

Physical address (m bits total)



Address Translation With a Page Table



Virtual memory example

- Parameters

- Virtual addresses are 12-bits
- Physical addresses are 16-bits
- Page size is 64 bytes

Mapping can be anything,
which is bigger doesn't really
matter!

1. How do we split Virtual Addresses into VPN and Offset?

11	10	9	8	7	6	5	4	3	2	1	0

Virtual memory example

- Parameters

- Virtual addresses are 12-bits
- Physical addresses are 16-bits
- Page size is 64 bytes

Mapping can be anything,
which is bigger doesn't really
matter!

- How do we split Virtual Addresses into VPN and Offset?

- Offset is based on page size: 64-bytes $\Rightarrow$ 6 bits. All the rest are VPN

11	10	9	8	7	6	5	4	3	2	1	0
Virtual Page Number						Page Offset					

- How big are Physical Page Numbers?

Virtual memory example

- Parameters

- Virtual addresses are 12-bits
- Physical addresses are 16-bits
- Page size is 64 bytes

Mapping can be anything,
which is bigger doesn't really
matter!

- How do we split Virtual Addresses into VPN and Offset?

- Offset is based on page size: 64-bytes $\Rightarrow$ 6 bits. All the rest are VPN

11	10	9	8	7	6	5	4	3	2	1	0
Virtual Page Number						Page Offset					

- How big are Physical Page Numbers? $16-6 = 10$ bits

Virtual memory example

- Parameters
 - Virtual addresses are 12-bits
 - Physical addresses are 16-bits
 - Page size is 64 bytes

11	10	9	8	7	6	5	4	3	2	1	0
Virtual Page Number						Page Offset					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Physical Page Number										Page Offset					

- Translate:
- Virtual address: 0x3F0
 - Binary:
 - VPN:
 - Offset:

Virtual memory example

- Parameters
 - Virtual addresses are 12-bits
 - Physical addresses are 16-bits
 - Page size is 64 bytes

11	10	9	8	7	6	5	4	3	2	1	0
Virtual Page Number						Page Offset					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Physical Page Number										Page Offset					

- Translate:
- Virtual address: 0x3F0
 - Binary: 0b001111110000
 - VPN: 0b001111
 - Offset: 0b110000

Virtual memory example

- Parameters
 - Virtual addresses are 12-bits
 - Physical addresses are 16-bits
 - Page size is 64 bytes

11	10	9	8	7	6	5	4	3	2	1	0
Virtual Page Number						Page Offset					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Physical Page Number										Page Offset					

- Translate:
- Virtual address: 0x3F0
 - Binary: 0b001111110000
 - VPN: 0b001111
 - Offset: 0b110000

VPN	PPN	Valid
0x00	0x123	1
0x01	0x156	1
0x02	0x143	1
0x03	0x16F	1
0x04	0x1FF	0
0x05	0x107	0
0x06	0x100	0
0x07	0x1C0	0
0x08	0x1D8	0
0x09	0x1BF	0
0x0A	0x000	1
0x0B	0x3FF	1
0x0C	0x308	0
0x0D	0x3FD	0
0x0E	0x111	1
0x0F	0x1F0	1

VPN	PPN	Valid
0x10	0x237	1
0x11	0x236	1
0x12	0x2B0	1
0x13	0x280	0
0x14	0x120	0
Continues on...		

- PPN:
- Offset:

Virtual memory example

- Parameters
 - Virtual addresses are 12-bits
 - Physical addresses are 16-bits
 - Page size is 64 bytes

11	10	9	8	7	6	5	4	3	2	1	0
Virtual Page Number						Page Offset					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Physical Page Number										Page Offset					

- Translate:
- Virtual address: 0x3F0
 - Binary: 0b001111110000
 - VPN: 0b001111
 - Offset: 0b110000

VPN	PPN	Valid
0x00	0x123	1
0x01	0x156	1
0x02	0x143	1
0x03	0x16F	1
0x04	0x1FF	0
0x05	0x107	0
0x06	0x100	0
0x07	0x1C0	0
0x08	0x1D8	0
0x09	0x1BF	0
0x0A	0x000	1
0x0B	0x3FF	1
0x0C	0x308	0
0x0D	0x3FD	0
0x0E	0x111	1
0x0F	0x1F0	1

VPN	PPN	Valid
0x10	0x237	1
0x11	0x236	1
0x12	0x2B0	1
0x13	0x280	0
0x14	0x120	0
Continues on...		

- PPN: 0b01 1111 0000
- Offset: 0b110000
- Physical address:

Virtual memory example

- Parameters
 - Virtual addresses are 12-bits
 - Physical addresses are 16-bits
 - Page size is 64 bytes

11	10	9	8	7	6	5	4	3	2	1	0
Virtual Page Number						Page Offset					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Physical Page Number										Page Offset					

- Translate:
- Virtual address: 0x3F0
 - Binary: 0b001111110000
 - VPN: 0b001111
 - Offset: 0b110000

VPN	PPN	Valid
0x00	0x123	1
0x01	0x156	1
0x02	0x143	1
0x03	0x16F	1
0x04	0x1FF	0
0x05	0x107	0
0x06	0x100	0
0x07	0x1C0	0
0x08	0x1D8	0
0x09	0x1BF	0
0x0A	0x000	1
0x0B	0x3FF	1
0x0C	0x308	0
0x0D	0x3FD	0
0x0E	0x111	1
0x0F	0x1F0	1

VPN	PPN	Valid
0x10	0x237	1
0x11	0x236	1
0x12	0x2B0	1
0x13	0x280	0
0x14	0x120	0
Continues on...		

- PPN: 0b0111110000
- Offset: 0b110000
- Physical address:
 - 0b0111110000110000
 - 0x7C30

Break + Question

- Parameters
 - Virtual addresses are 12-bits
 - Physical addresses are 16-bits
 - Page size is 64 bytes

11	10	9	8	7	6	5	4	3	2	1	0
Virtual Page Number						Page Offset					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Physical Page Number										Page Offset					

- Translate:
- Virtual address: 0x500
 - Binary:
 - VPN:
 - Offset:

VPN	PPN	Valid
0x00	0x123	1
0x01	0x156	1
0x02	0x143	1
0x03	0x16F	1
0x04	0x1FF	0
0x05	0x107	0
0x06	0x100	0
0x07	0x1C0	0
0x08	0x1D8	0
0x09	0x1BF	0
0x0A	0x000	1
0x0B	0x3FF	1
0x0C	0x308	0
0x0D	0x3FD	0
0x0E	0x111	1
0x0F	0x1F0	1

VPN	PPN	Valid
0x10	0x237	1
0x11	0x236	1
0x12	0x2B0	1
0x13	0x280	0
0x14	0x120	0
Continues on...		

- PPN:
- Offset:
- Physical address:

Break + Question

- Parameters
 - Virtual addresses are 12-bits
 - Physical addresses are 16-bits
 - Page size is 64 bytes

11	10	9	8	7	6	5	4	3	2	1	0
Virtual Page Number						Page Offset					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Physical Page Number										Page Offset					

- Translate:
- Virtual address: 0x500
 - Binary: 0b010100000000
 - VPN: 0b010100
 - Offset: 0b000000

VPN	PPN	Valid
0x00	0x123	1
0x01	0x156	1
0x02	0x143	1
0x03	0x16F	1
0x04	0x1FF	0
0x05	0x107	0
0x06	0x100	0
0x07	0x1C0	0
0x08	0x1D8	0
0x09	0x1BF	0
0x0A	0x000	1
0x0B	0x3FF	1
0x0C	0x308	0
0x0D	0x3FD	0
0x0E	0x111	1
0x0F	0x1F0	1

VPN	PPN	Valid
0x10	0x237	1
0x11	0x236	1
0x12	0x2B0	1
0x13	0x280	0
0x14	0x120	0
Continues on...		

- PPN: INVALID
- Offset:
- Physical address:
 - Page Fault**

Break + Practice again

- Parameters
 - Virtual addresses are 12-bits
 - Physical addresses are 16-bits
 - Page size is 64 bytes

11	10	9	8	7	6	5	4	3	2	1	0
Virtual Page Number						Page Offset					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Physical Page Number										Page Offset					

- Translate:
- Virtual address: 0x0D6
 - Binary:
 - VPN:
 - Offset:

VPN	PPN	Valid
0x00	0x123	1
0x01	0x156	1
0x02	0x143	1
0x03	0x16F	1
0x04	0x1FF	0
0x05	0x107	0
0x06	0x100	0
0x07	0x1C0	0
0x08	0x1D8	0
0x09	0x1BF	0
0x0A	0x000	1
0x0B	0x3FF	1
0x0C	0x308	0
0x0D	0x3FD	0
0x0E	0x111	1
0x0F	0x1F0	1

VPN	PPN	Valid
0x10	0x237	1
0x11	0x236	1
0x12	0x2B0	1
0x13	0x280	0
0x14	0x120	0
Continues on...		

- PPN:
- Offset:
- Physical address:

Break + Practice again

- Parameters
 - Virtual addresses are 12-bits
 - Physical addresses are 16-bits
 - Page size is 64 bytes

11	10	9	8	7	6	5	4	3	2	1	0
Virtual Page Number						Page Offset					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Physical Page Number										Page Offset					

- Translate:
- Virtual address: 0x0D6
 - Binary: 0b000011010110
 - VPN: 0b000011
 - Offset: 0b010110

VPN	PPN	Valid
0x00	0x123	1
0x01	0x156	1
0x02	0x143	1
0x03	0x16F	1
0x04	0x1FF	0
0x05	0x107	0
0x06	0x100	0
0x07	0x1C0	0
0x08	0x1D8	0
0x09	0x1BF	0
0x0A	0x000	1
0x0B	0x3FF	1
0x0C	0x308	0
0x0D	0x3FD	0
0x0E	0x111	1
0x0F	0x1F0	1

VPN	PPN	Valid
0x10	0x237	1
0x11	0x236	1
0x12	0x2B0	1
0x13	0x280	0
0x14	0x120	0
Continues on...		

- PPN: 0b010 110 1111
- Offset: 0b010110
- Physical address:
 - 0b0101101111010110
 - 0x5BD6

How do we get the actual memory value?

- Same as prior questions in class
 - We have a physical address, just read the data from memory
 - Pay attention to endianness
 - Always little-endian for x86-64 systems
 - Pay attention to the *size* of the memory access
 - 1, 2, 4, or 8 bytes
- Good news: this is the easy part of virtual memory questions

Short Break + Reading values from memory

Address	0	1	2	3	4	5	6	7
0x5BC0	2E	E2	BD	62	EF	A0	CD	93
0x5BC8	A4	75	61	2F	0F	DB	64	A4
0x5BD0	54	7A	F2	60	6E	47	B0	92
0x5BD8	DA	72	8F	A8	E5	15	18	CE
0x5BE0	86	BF	6A	6A	92	99	CF	6C

- What is the 2-byte value at 0x5BD6? (little-endian)

Short Break + Reading values from memory

Address	0	1	2	3	4	5	6	7
0x5BC0	2E	E2	BD	62	EF	A0	CD	93
0x5BC8	A4	75	61	2F	0F	DB	64	A4
0x5BD0	54	7A	F2	60	6E	47	B0	92
0x5BD8	DA	72	8F	A8	E5	15	18	CE
0x5BE0	86	BF	6A	6A	92	99	CF	6C

- What is the 2-byte value at 0x5BD6? (little-endian)
 - 0x92B0

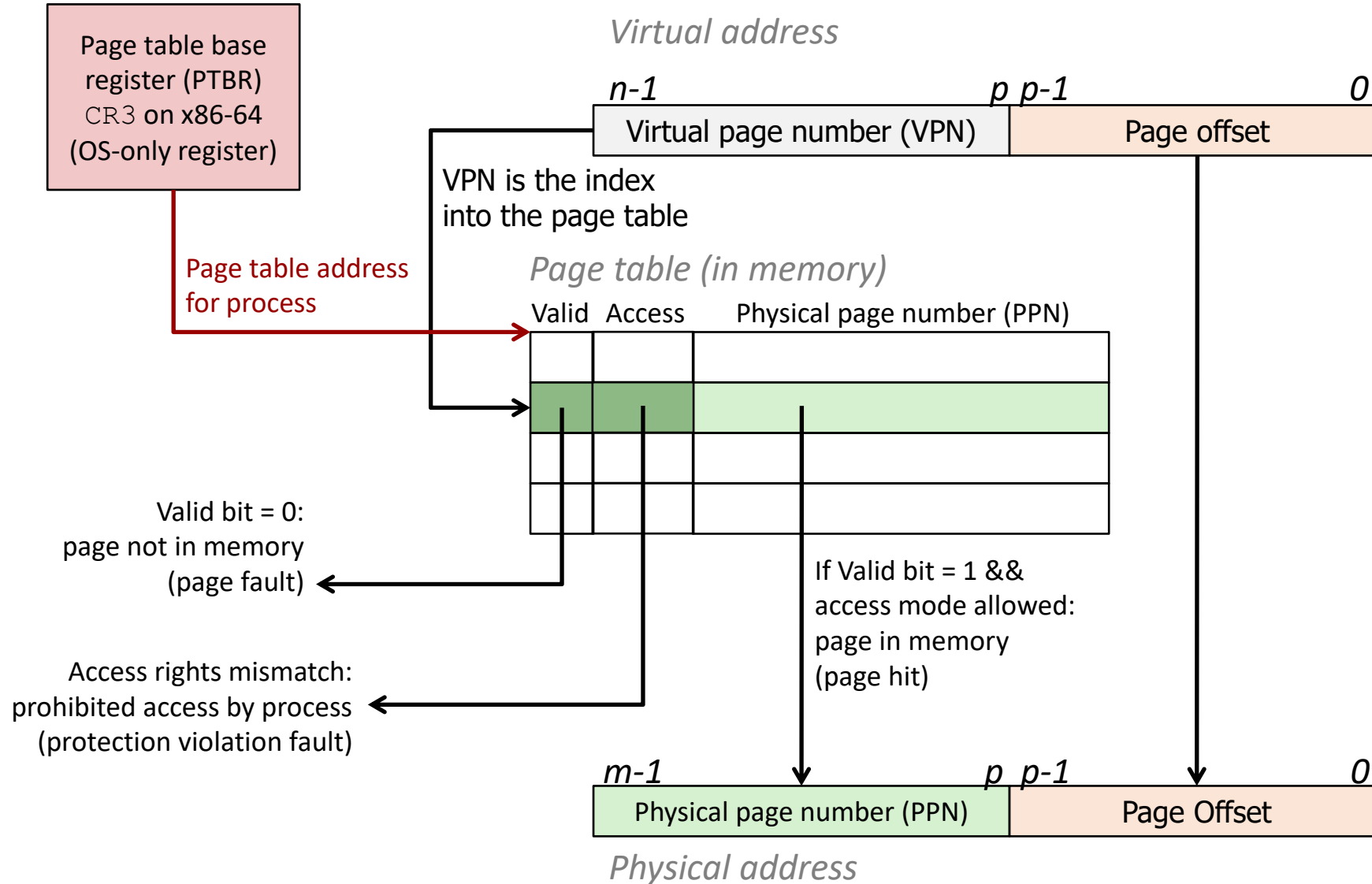
Outline

- Virtual Memory Concept
- Virtual Memory Process
- Solving Memory Problems with Virtual Memory
- Address Translation
- **Virtual Memory Summary**

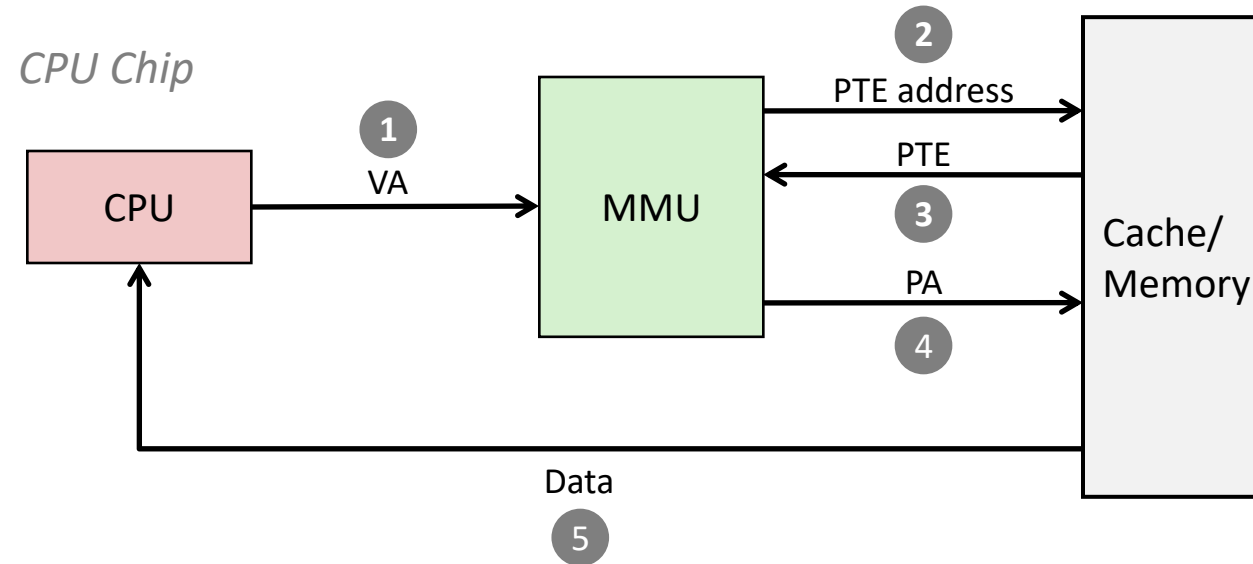
Virtual memory idea

- Processes see Virtual Addresses
 - Per-process representation of memory
- The OS and hardware see Physical Addresses
 - Real locations in RAM
- The OS keeps a Page Table for each process
 - Translates Virtual Pages (chunks of virtual memory) into Physical Pages (chunks of physical memory)

The MMU does address translation using a Page Table

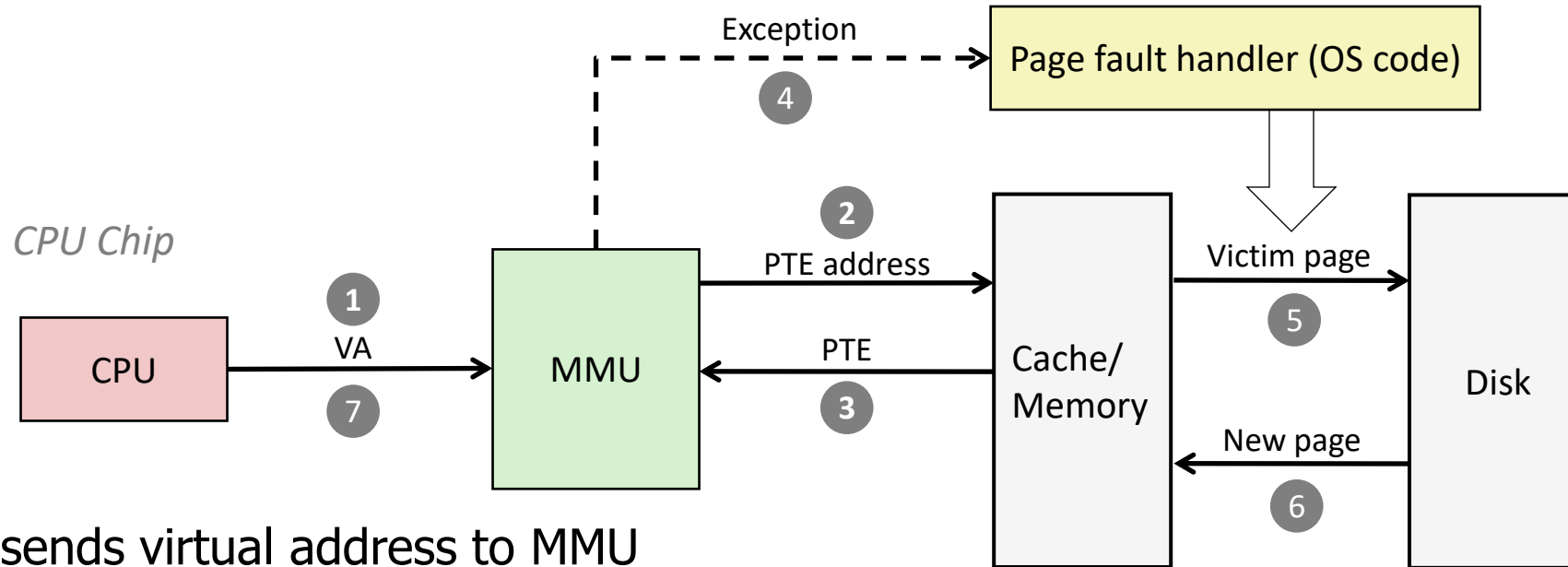


Memory Access: Page Hit



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in cache/memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

Memory Access: Page Fault



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in cache/memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction (which will now succeed)

Outline

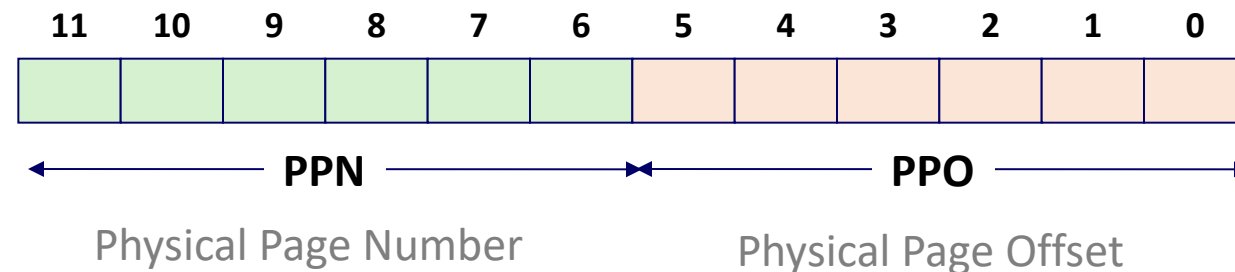
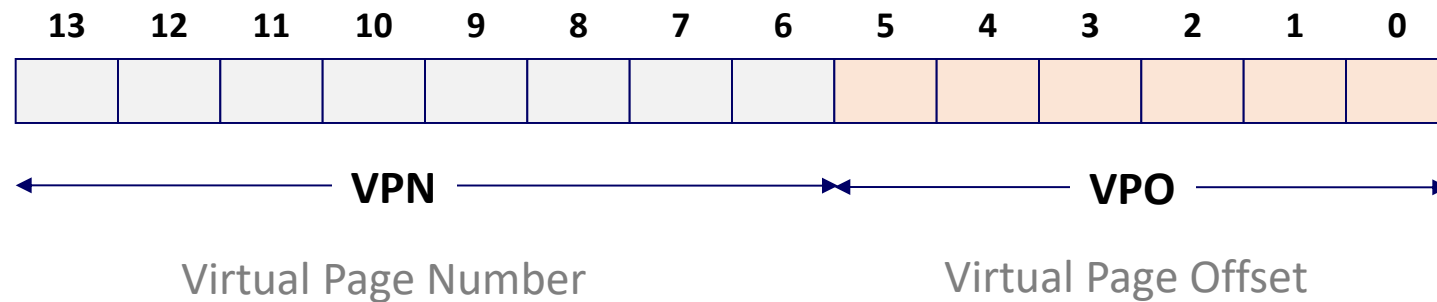
- Virtual Memory Concept
- Virtual Memory Process
- Solving Memory Problems with Virtual Memory
- Address Translation
- Virtual Memory Summary

Outline

- Bonus: Memory System Practice Problems

Simple Memory System Example

- Addressing
 - 14-bit virtual addresses
 - 12-bit physical address
 - Page size = 64 bytes



Simple Memory System: Page Table

We only show a few entries (out of 256)

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

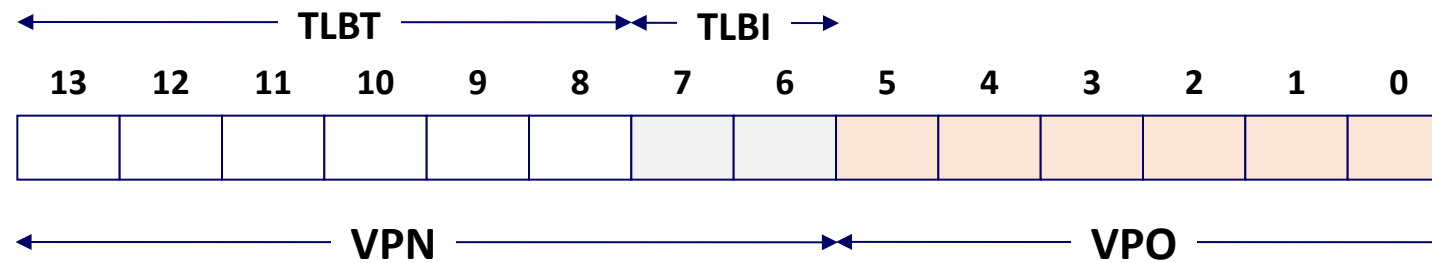
<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

...

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
2E	–	0
⋮		

Simple Memory System: TLB

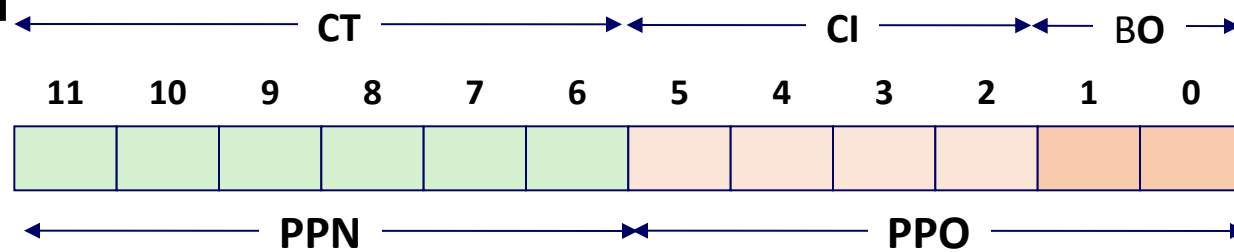
- 16 entries
- 4-way associative



<i>Set</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Simple Memory System: L1 Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped



<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B3</i>	<i>B2</i>	<i>B1</i>	<i>B0</i>
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	1D	72	F0	36
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B3</i>	<i>B2</i>	<i>B1</i>	<i>B0</i>
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Address Translation Example #1

(using the Page Table, TLB, and L1 cache shown in the preceding slides)

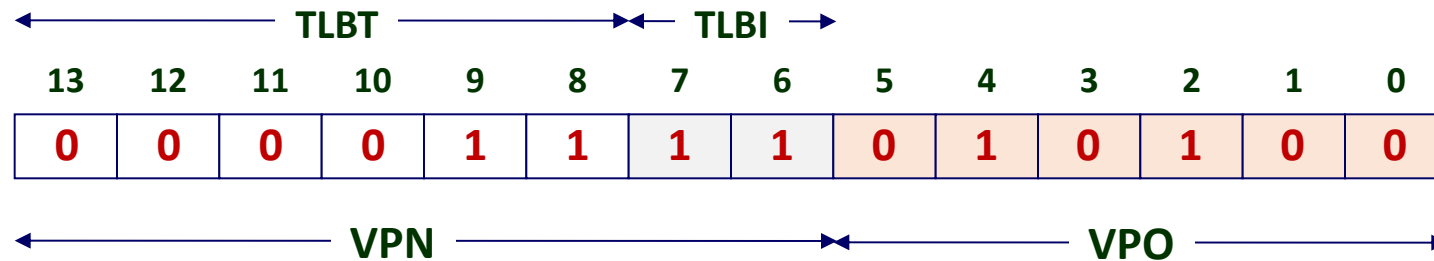
movb (%rcx), %al

Virtual Address: 0x03D4

Address space: 14-bit VAddr, 12-bit PAddr, 64-byte page

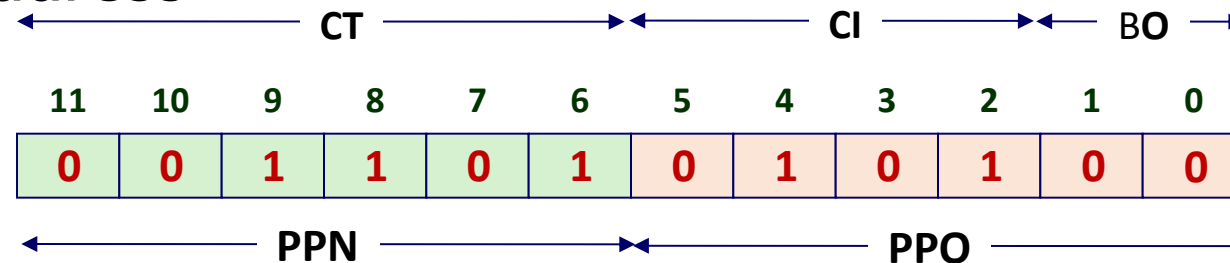
TLB: 16 entries, 4-way

L1 Cache: 16 lines, 4-byte block, direct mapped,
Physically addressed



VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

Physical Address

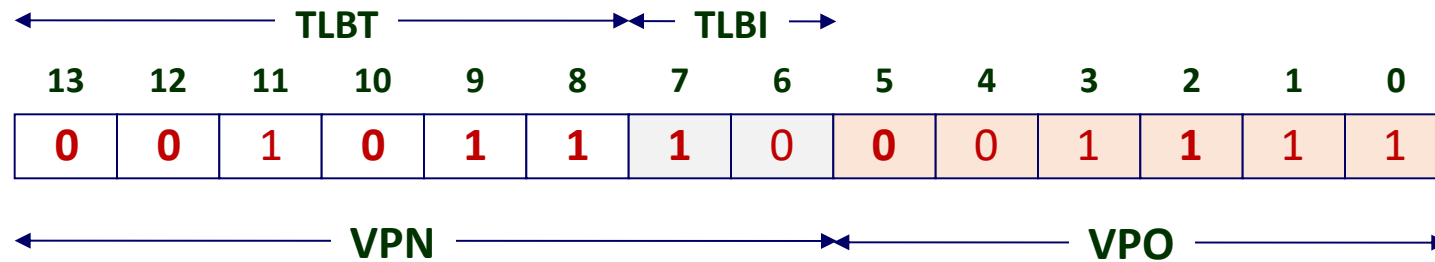


BO 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36

Address Translation Example #2

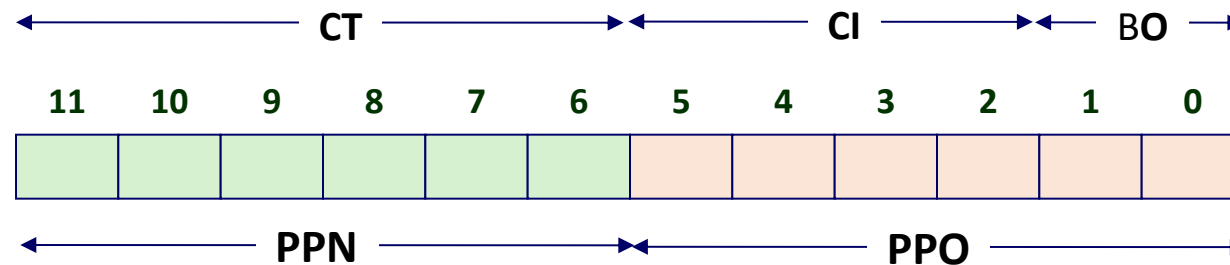
(using the Page Table, TLB, and L1 cache shown in the preceding slides)

Virtual Address: 0x0B8F



VPN 0x2E TLBI 2 TLBT 0x0B TLB Hit? N Page Fault? Y PPN: TBD

Physical Address



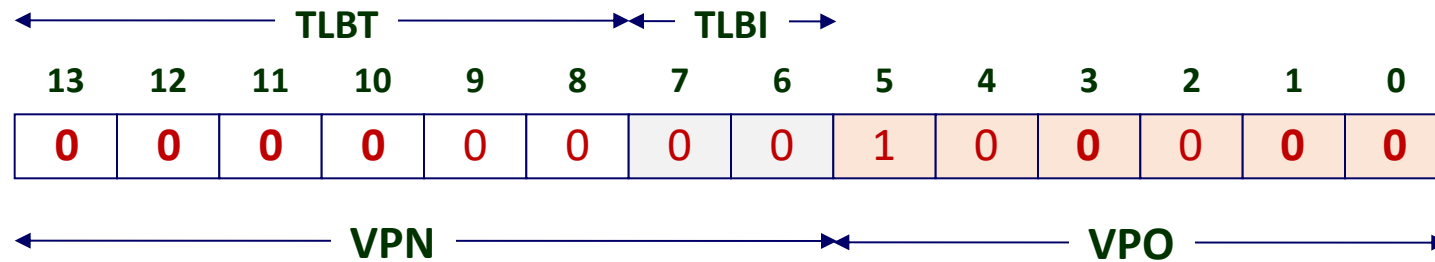
BO ____ CI ____ CT ____ Hit? ____ Byte: ____

Likely invalid page. Maybe needs to read from disk. Either way we don't know the PPN.

Address Translation Example #3

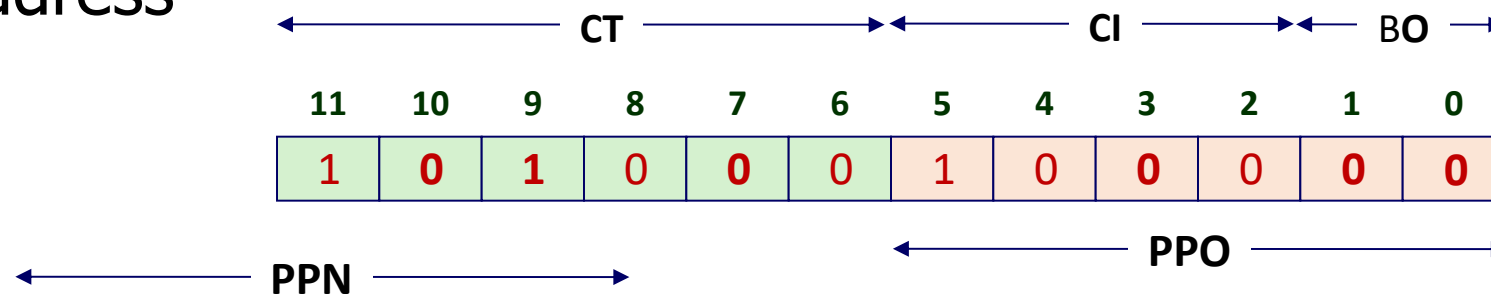
(using the Page Table, TLB, and L1 cache shown in the preceding slides)

Virtual Address: 0x0020



VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

Physical Address



BO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem Cache miss, so needs to read byte values from main memory

Outline

- Bonus: Optimizing Page Table accesses with a TLB

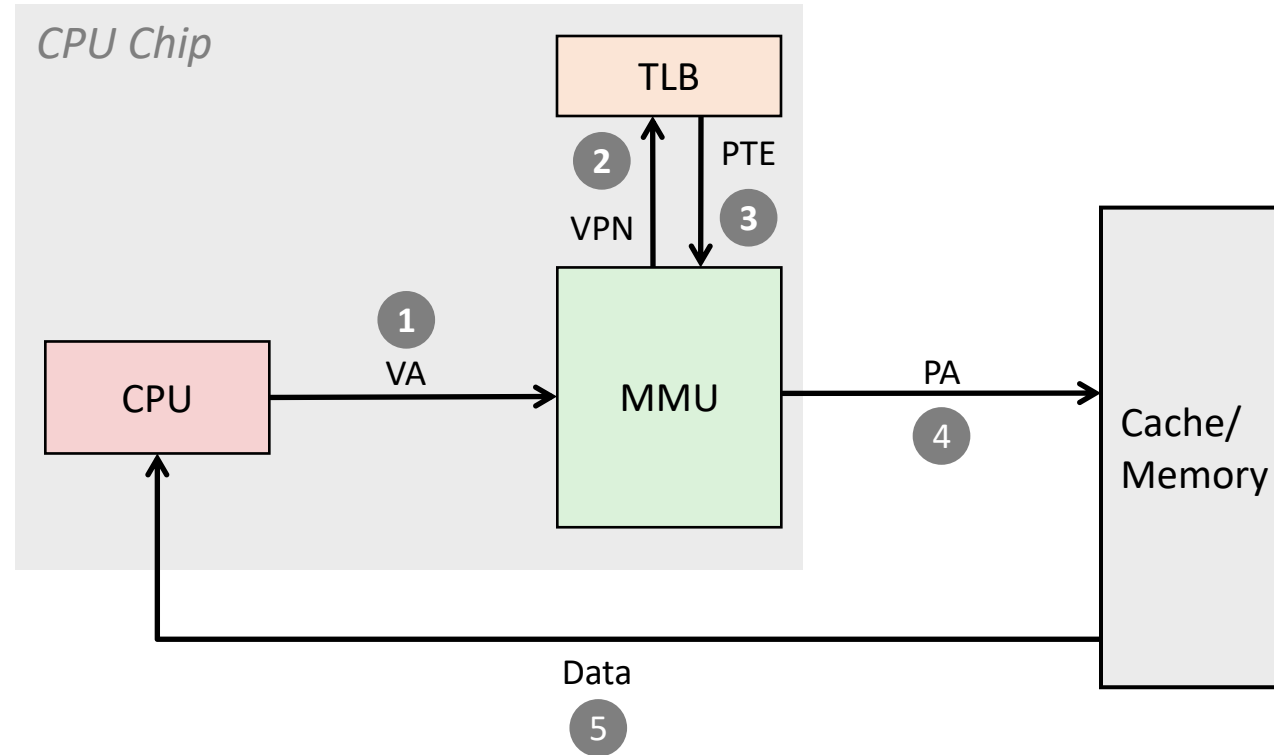
Accessing page tables is slow

- Problem: page tables are in memory
 - And we need to access them to find our address to access memory
 - Two memory accesses per access!!! 😱
- Page table entries (PTEs) are cached in L1, L2, etc, like any other data in memory
 - PTEs may be evicted by other data references. Oops.
 - PTE access still requires average effective memory access delay

Speeding up Translation with a TLB

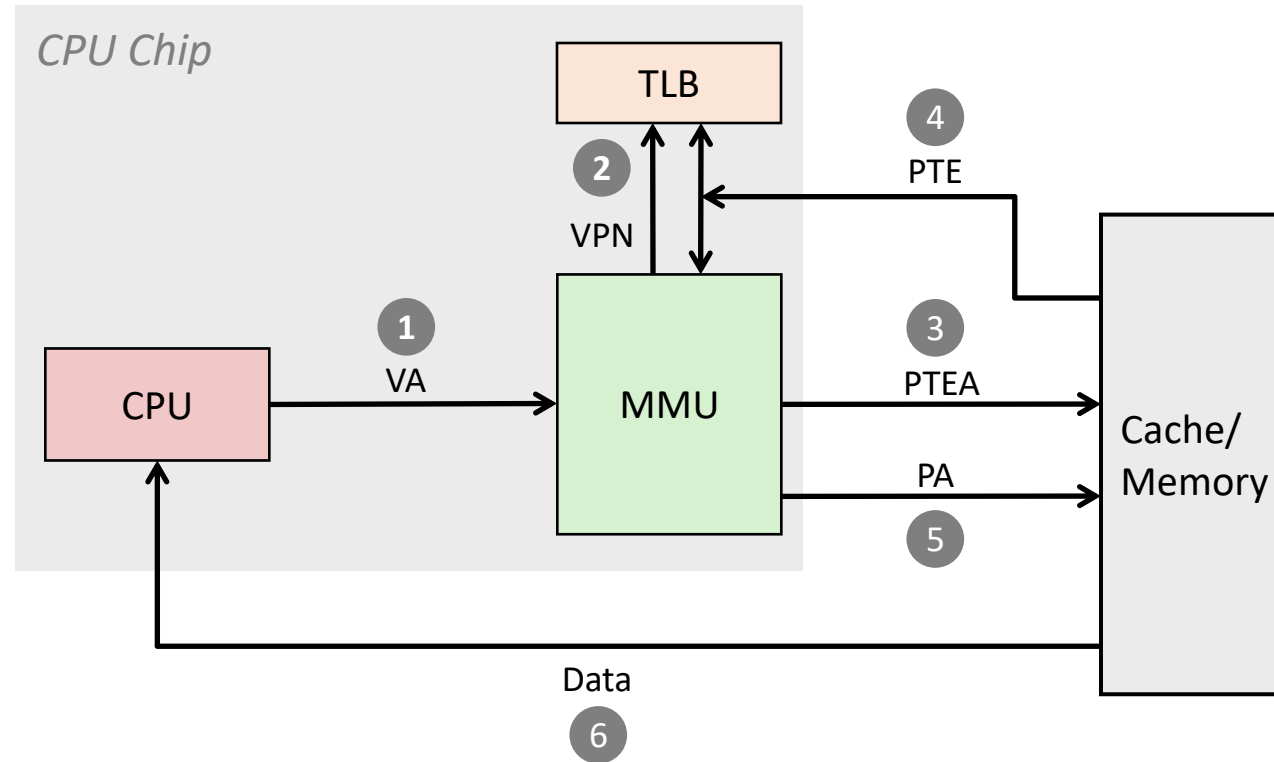
- Solution: *Translation Lookaside Buffer* (TLB)
 - Small hardware cache memory inside MMU
 - Contains page table entries for a small number of pages
 - Maps virtual page numbers to physical page numbers
 - Reduces issues with data kicking PTEs out of caches!
- Like cache memories, uses set indices, tags, and valid bits
 - VPN split into: TLB tag and TLB index (just like caches, because it is one!)
 - No need for a block offset equivalent (PTEs have a single value)

TLB Hit



A TLB hit eliminates a memory access

TLB Miss

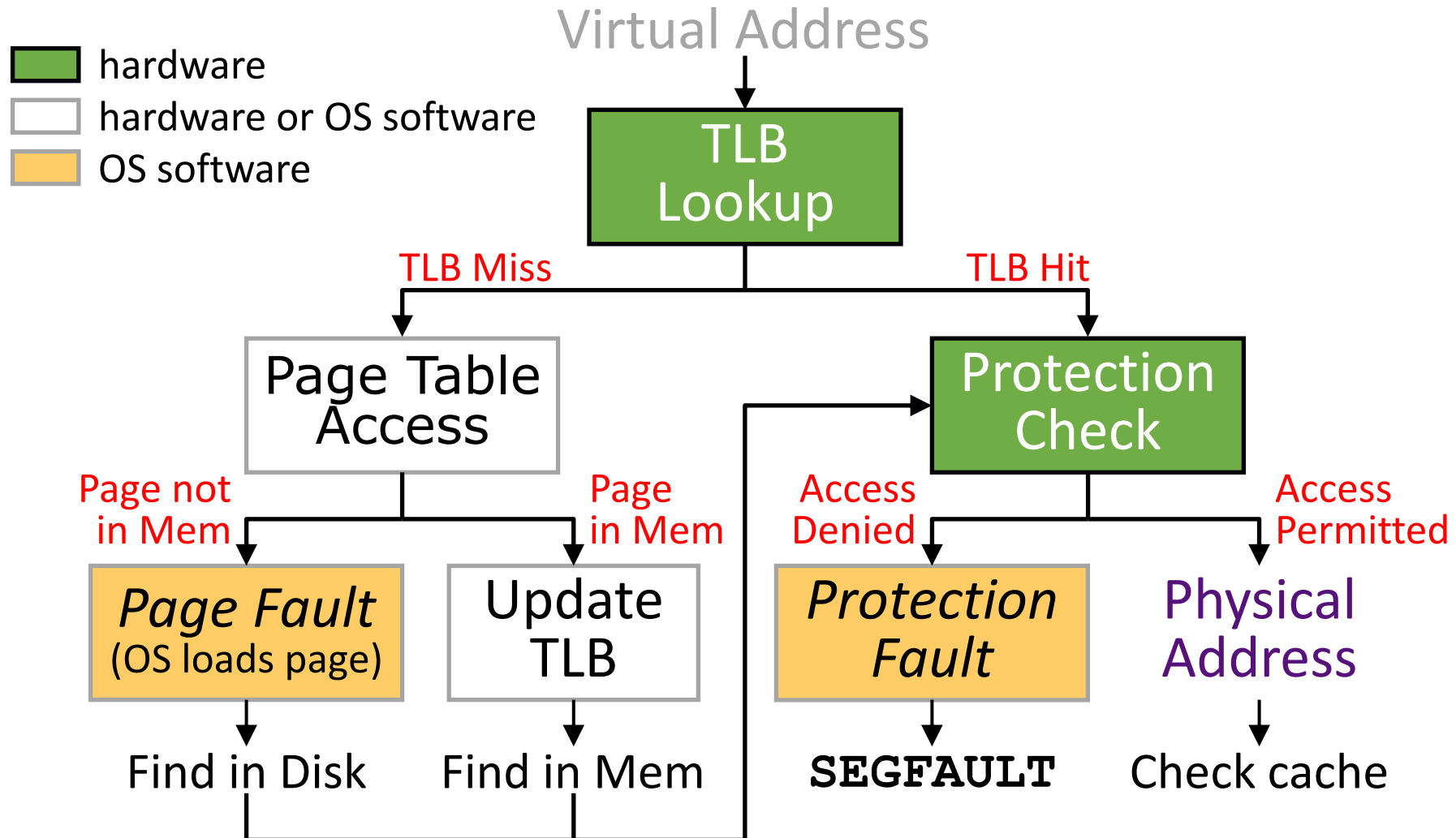


A TLB miss incurs an additional memory access (the PTE)

Fortunately, TLB misses are rare. Why?

Locality. It's always locality.

Address translation process

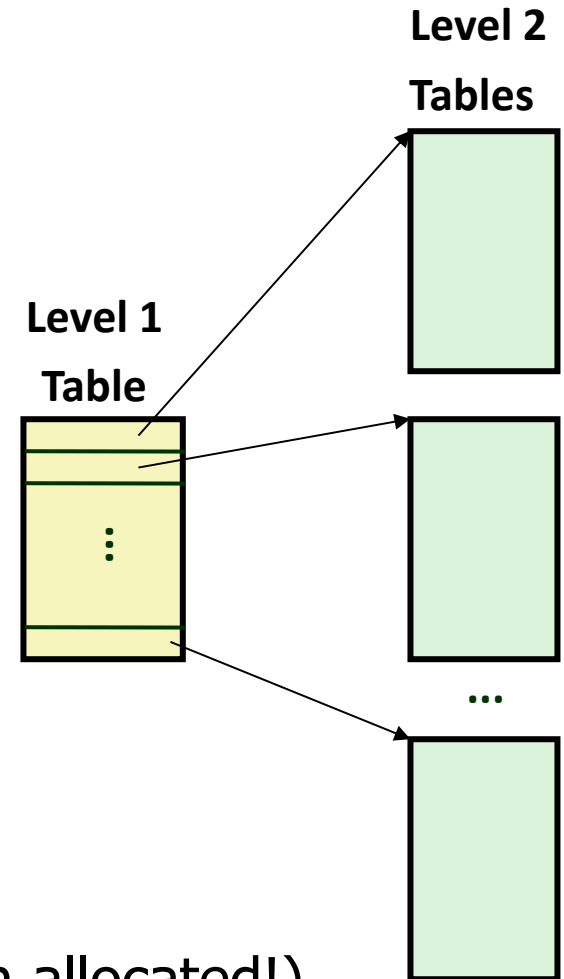


Outline

- Bonus: Multi-level Page Tables

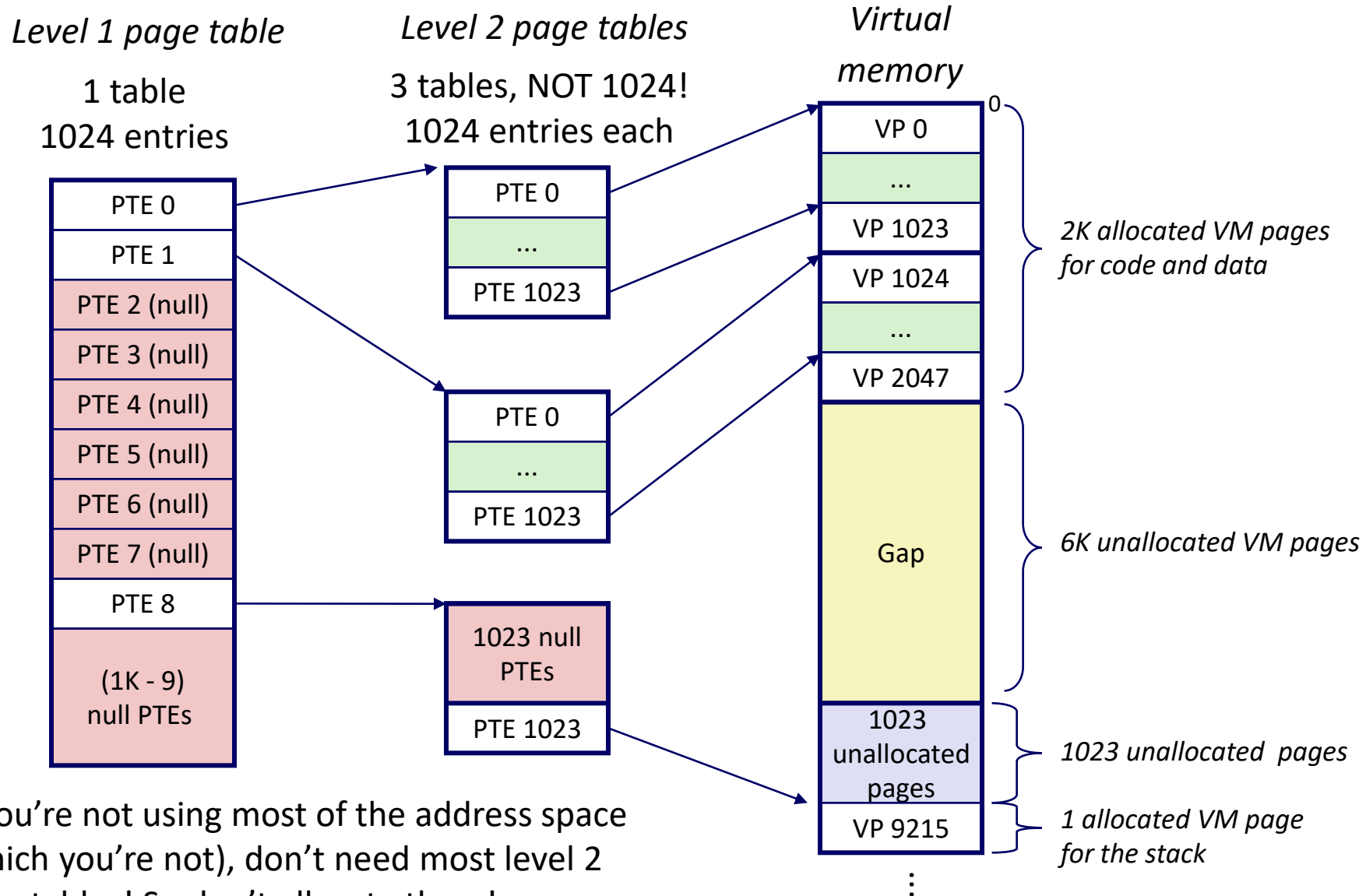
Multi-Level Page Tables

- Suppose:
 - 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE
- How big is the page table?
 - Would need a 512 GB page table!
 - $2^{48} * 2^{-12} * 2^3 = 2^{39}$ bytes
 - That's just meta-data!
Where does the data go?
- Common solution:
 - Multi-level page tables
 - Split the VPN into multiple pieces, 1 per level
 - Example: 2-level page table
 - Level 1 table: each PTE points to a level 2 page table (always memory resident)
 - Level 2 table: each PTE points to a page (paged in and out like any other data, maybe not even allocated!)

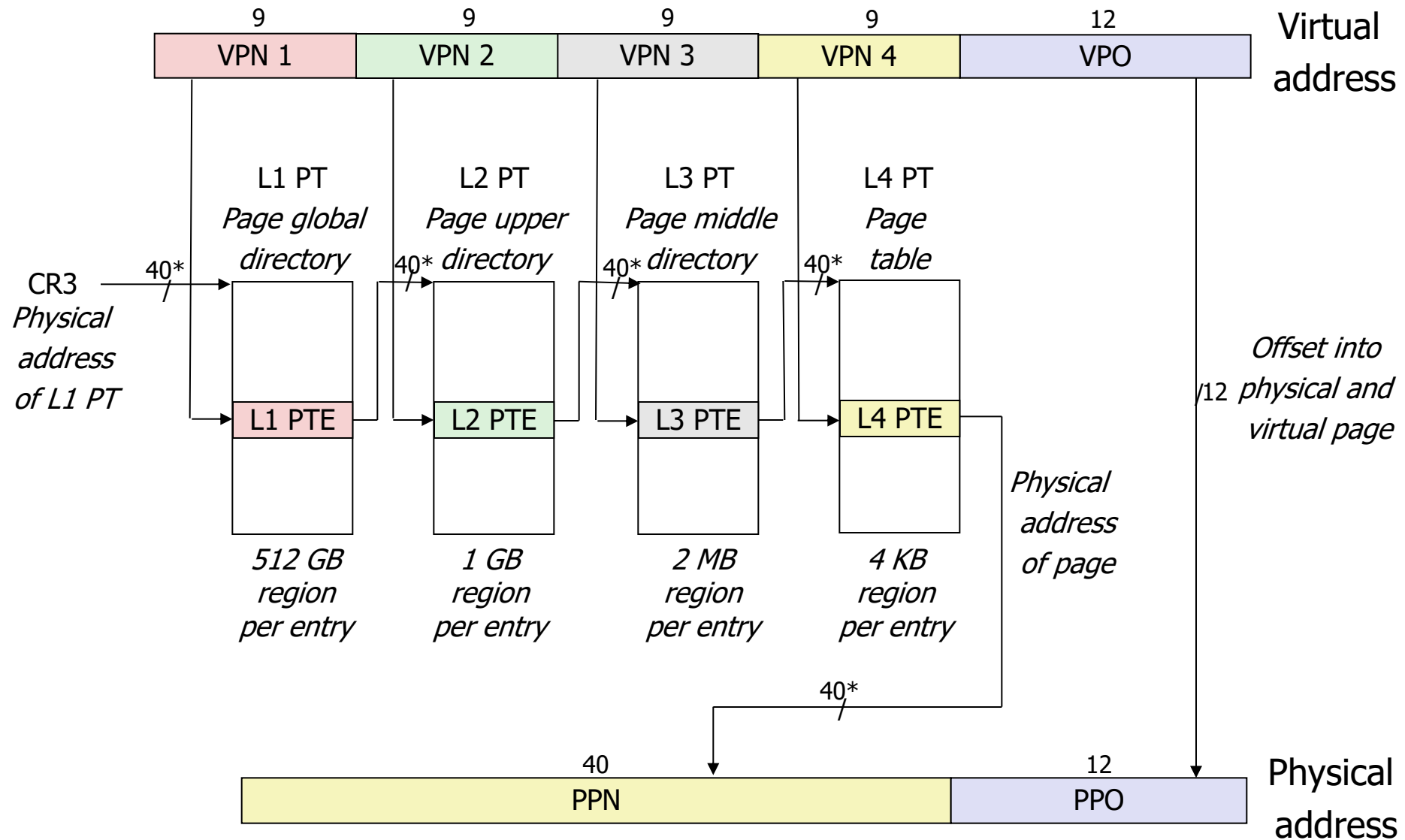


A Two-Level Page Table Hierarchy

32 bit addresses, 4KB pages, 4-byte PTEs



Multi-level page table: Core i7



**aligned to a 4K-boundary*

End-to-end Core i7 Data Address Translation

