Lecture 16
Virtual Memory

CS213 – Intro to Computer Systems

Branden Ghena – Fall 2023

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

Administrivia

- Homework 4
  - Due Tuesday after Thanksgiving
  - Caches and Cache Performance and Virtual Memory
    - Today's lecture is relevant

- SETI Lab
  - Due Thursday after Thanksgiving
  - See pinned Piazza posts on Getting Started and on Testing
  - Make sure you do tests of your code on Amdahl!!
    - Running with many cores on Moore slows it down for everyone

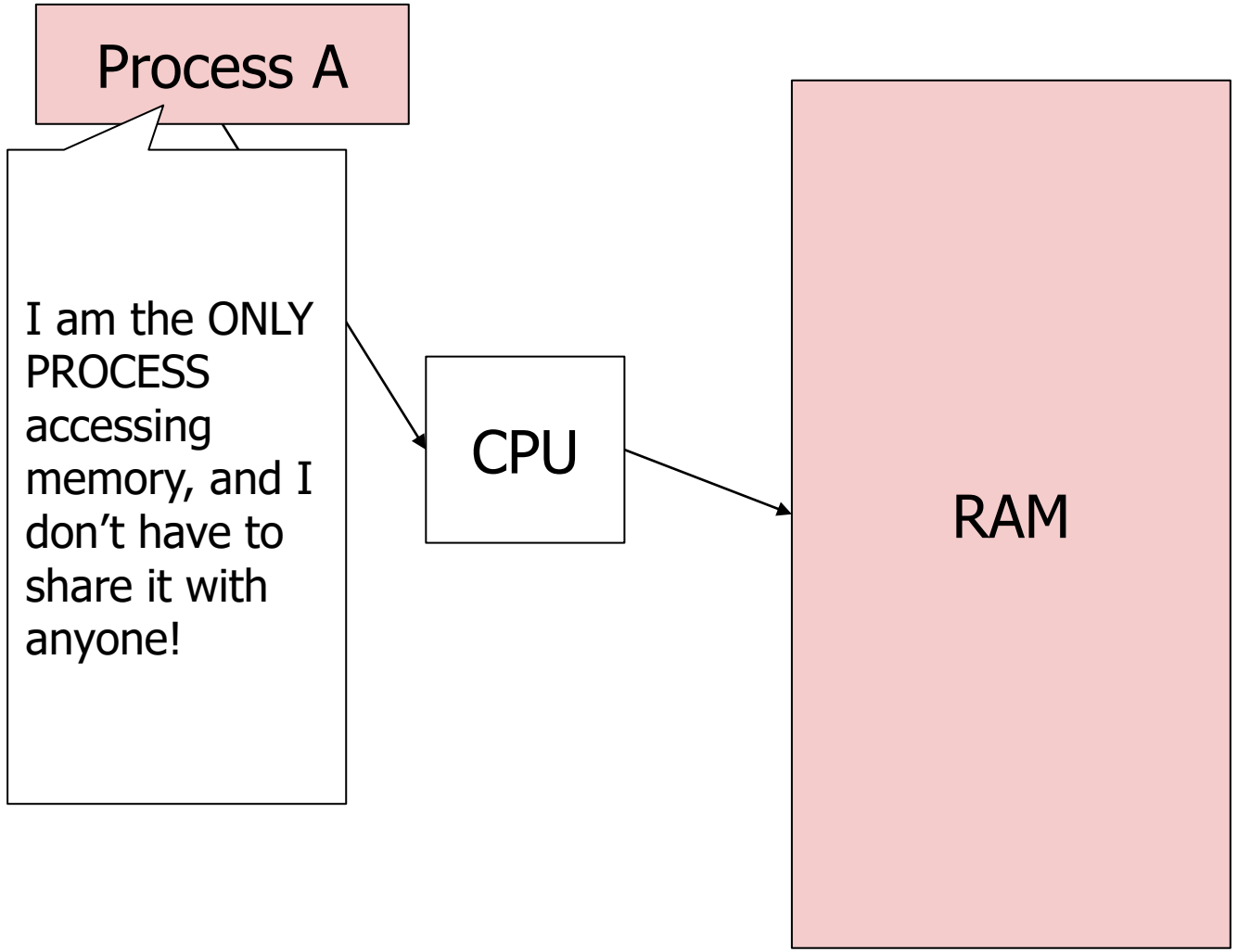
Today's Goals

- 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 that we won't test you on

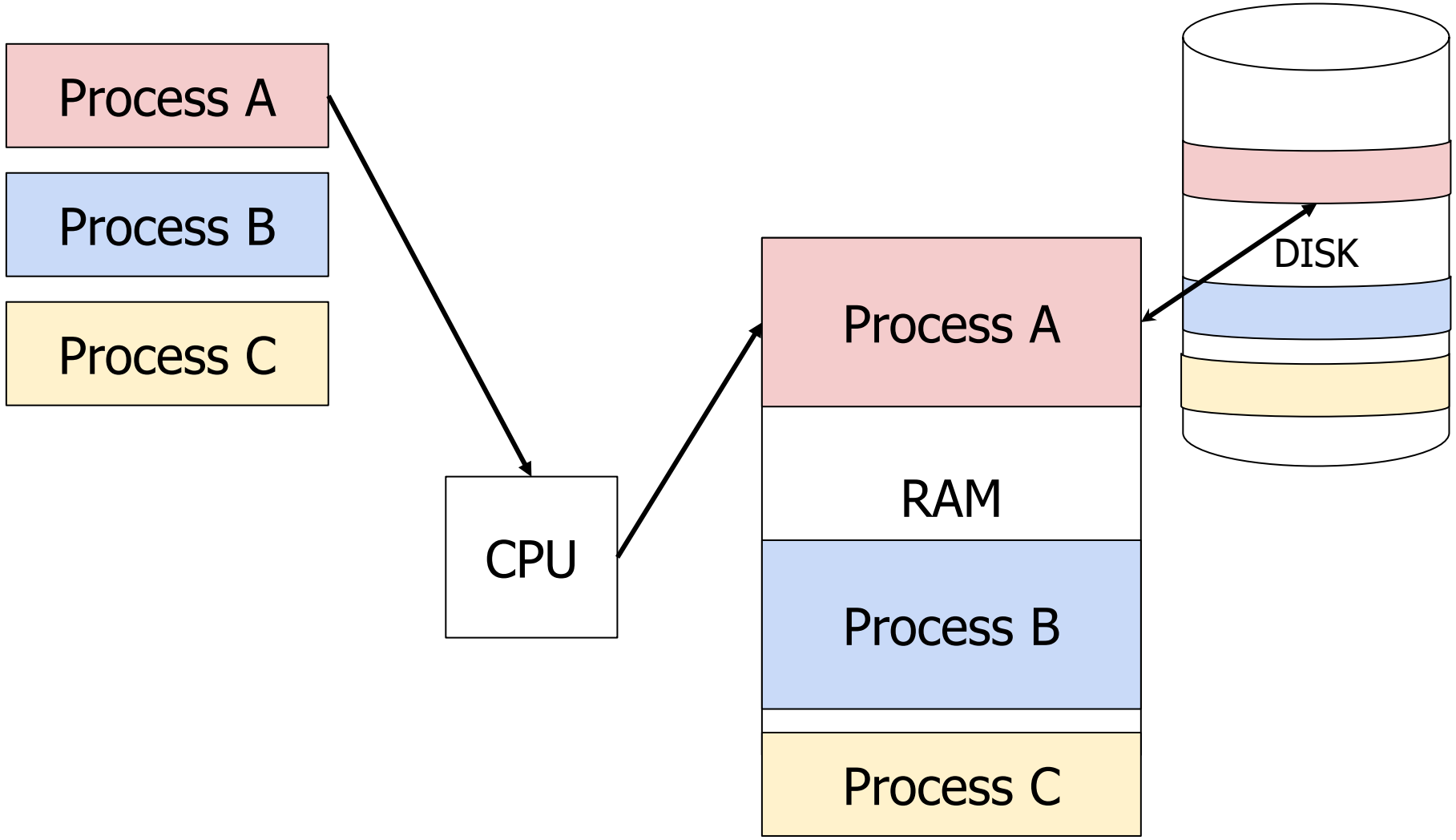
Outline

- **Memory Problems**
- Virtual Memory Concept
- Virtual Memory Process
- Memory Problems Solved
- Address Translation
- Virtual Memory Summary

The Illusion!



The Reality!

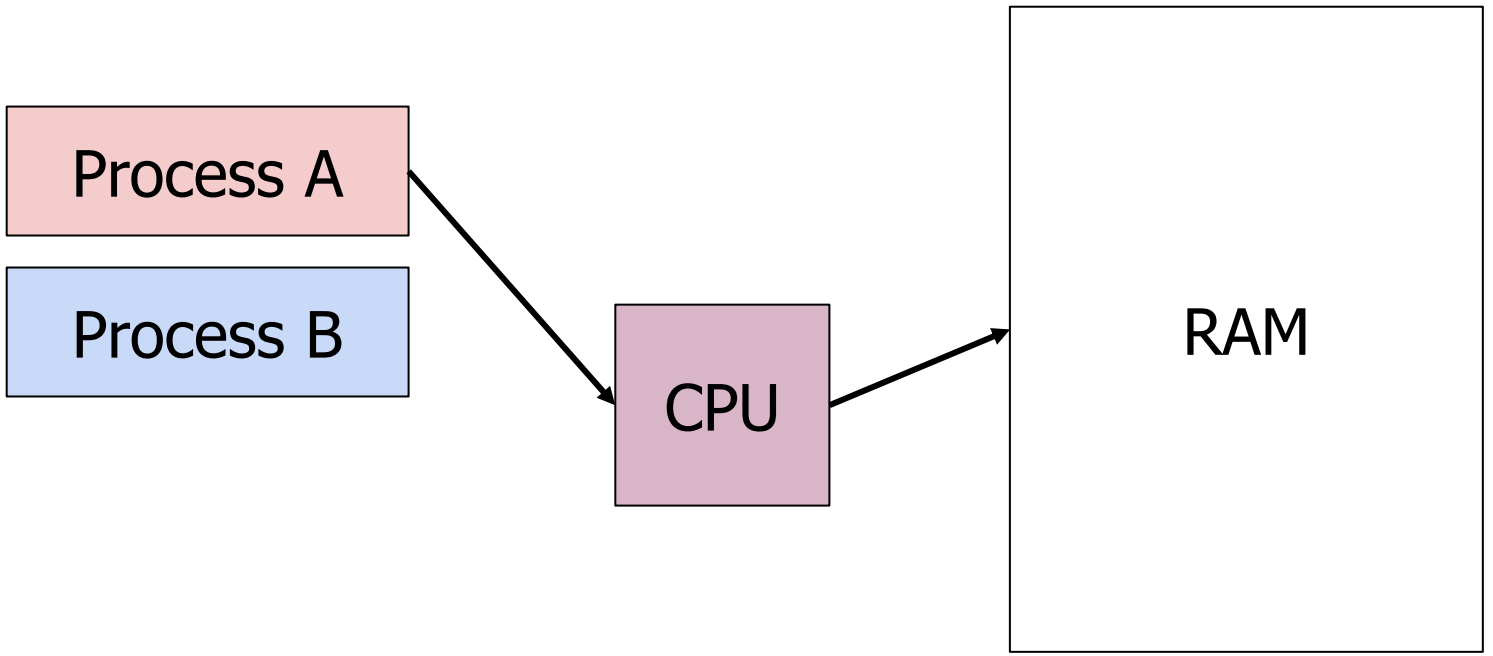

Process A
Process B
Process C
CPU
Process A
RAM
Process B
Process C
DISK

Memory problems

- What are the challenges to supporting this reality?

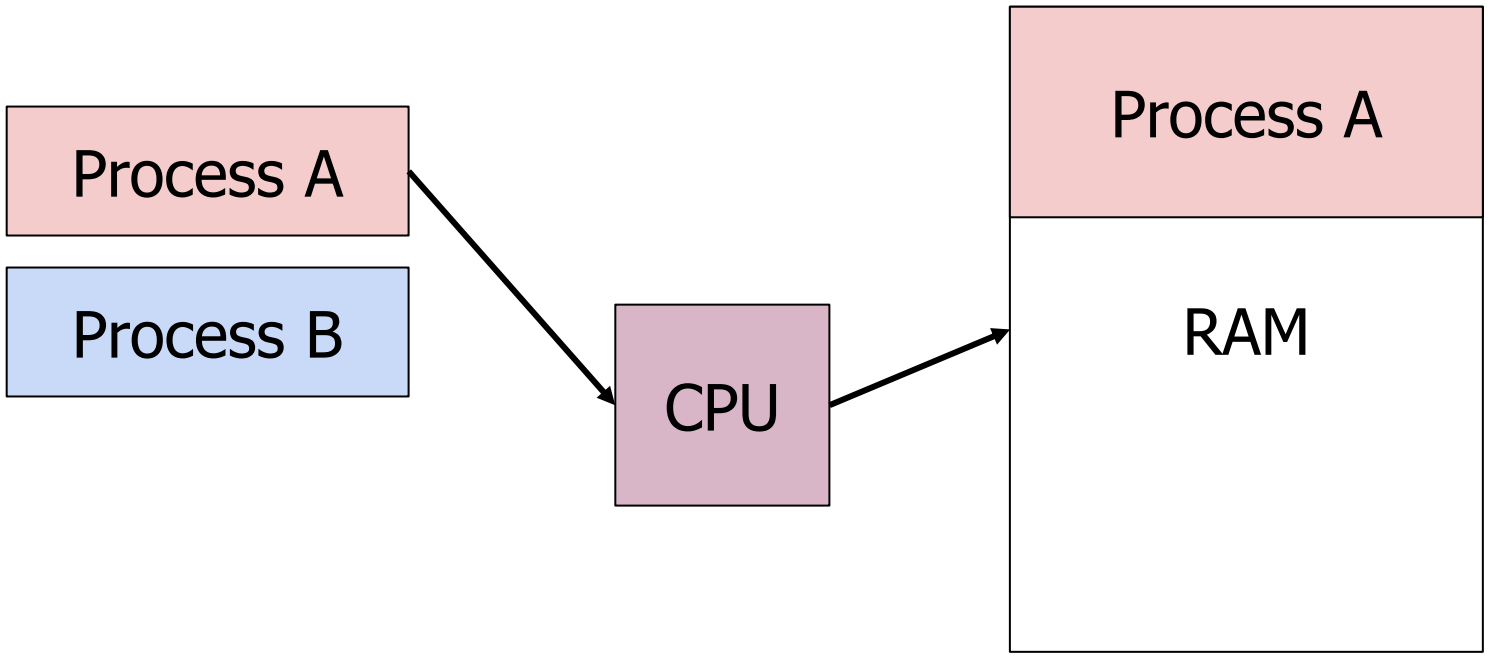
    1. Which addresses does each process get?

Multiple applications share RAM

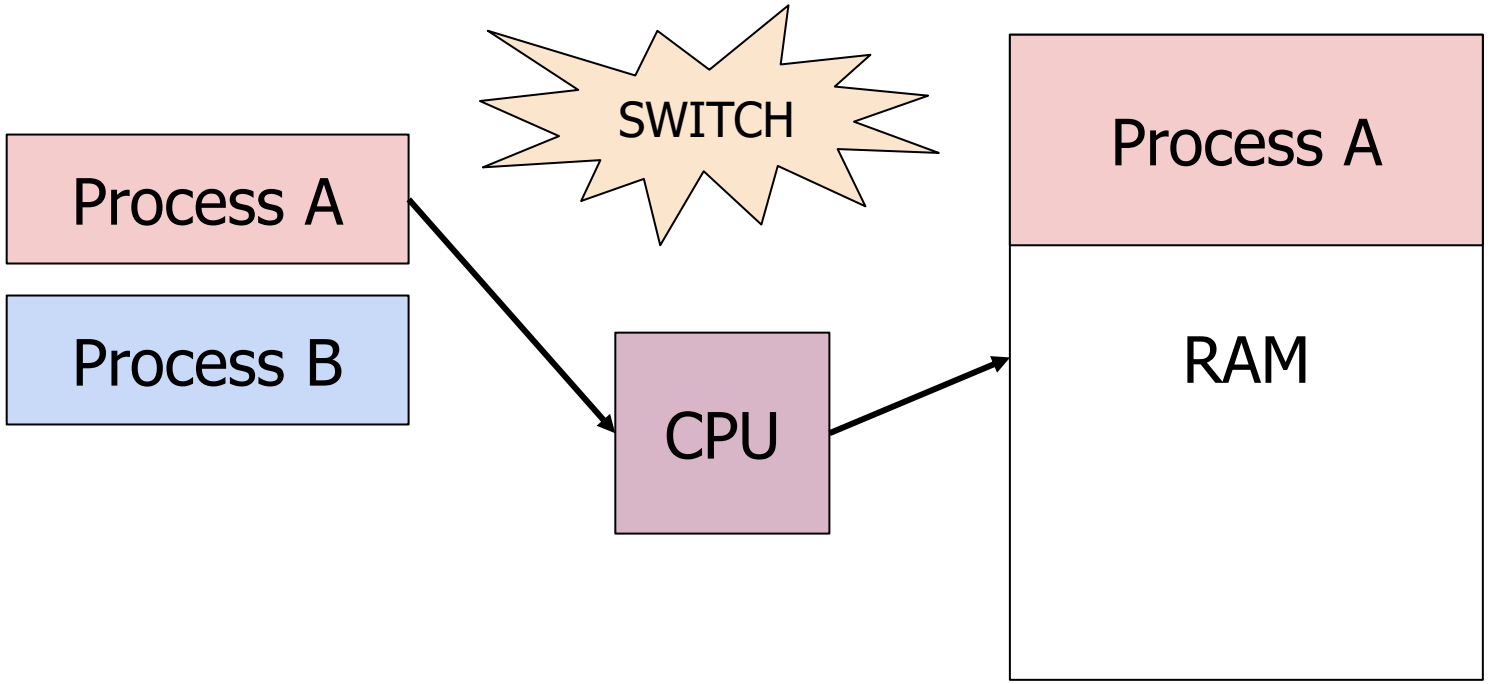


Both processes assume they start at the beginning of RAM and use as much as they need

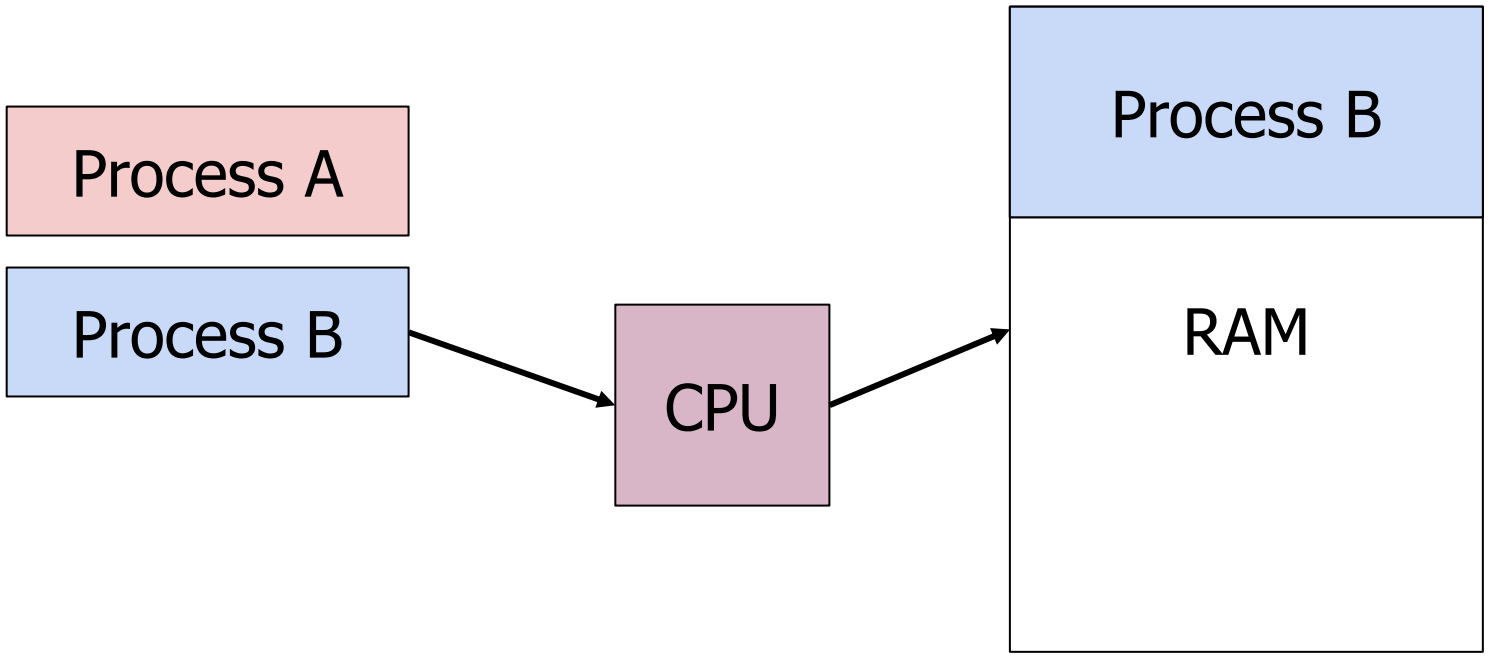
Multiple applications share RAM



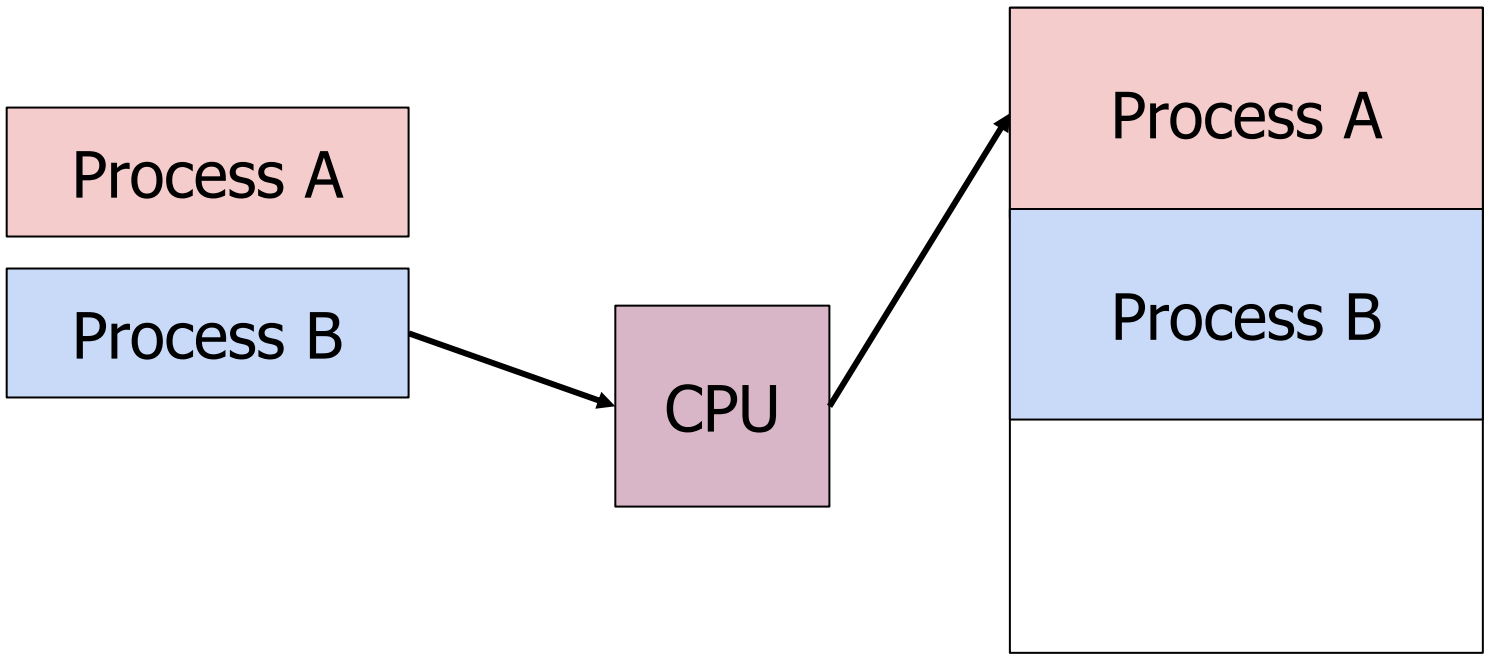
Multiple applications share RAM



Multiple applications share RAM



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…

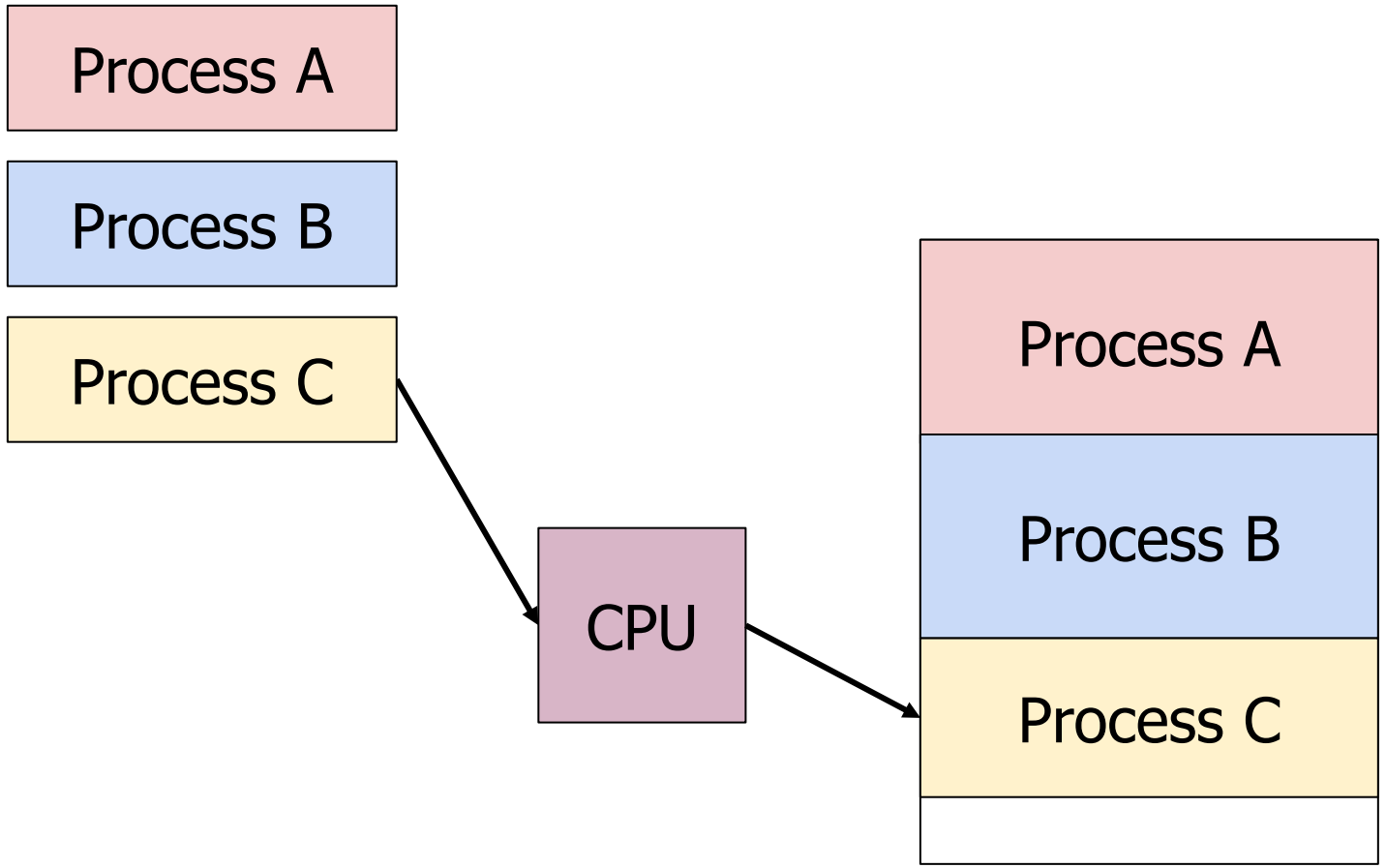
Memory problems

- What are the challenges to supporting this reality?

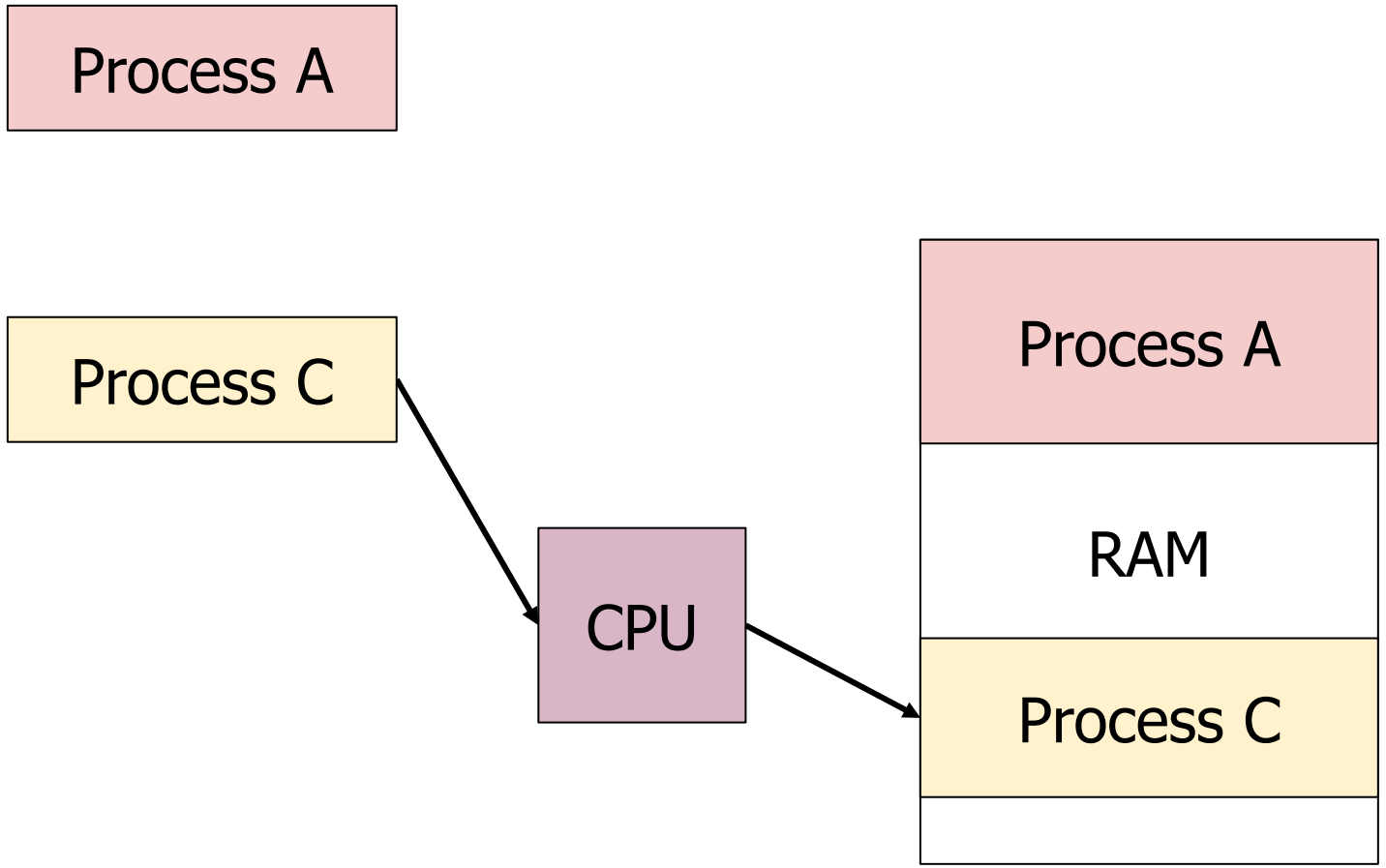
    1. Which addresses does each process get?

    2. How do we move memory around?

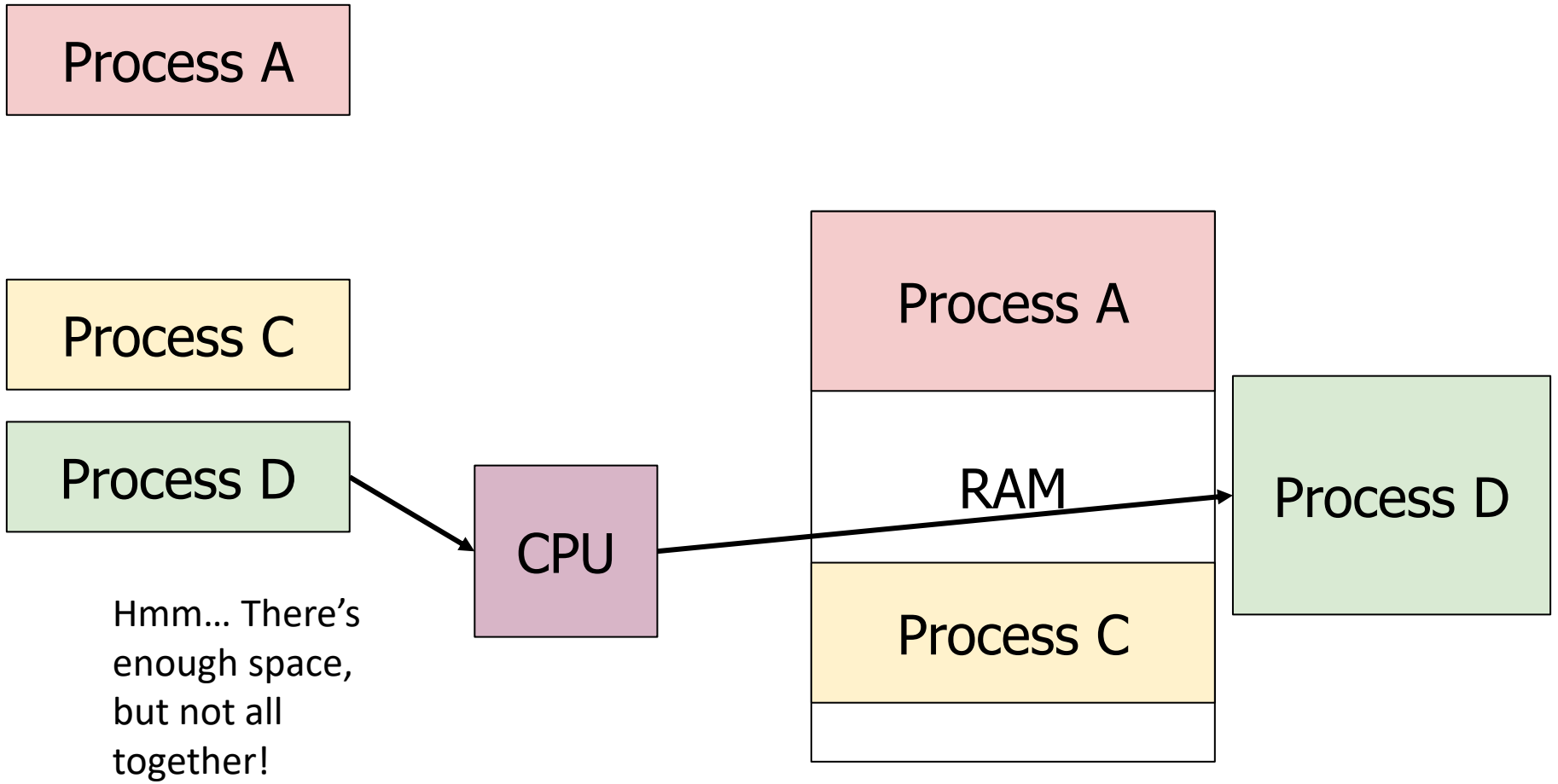
Memory fragmentation



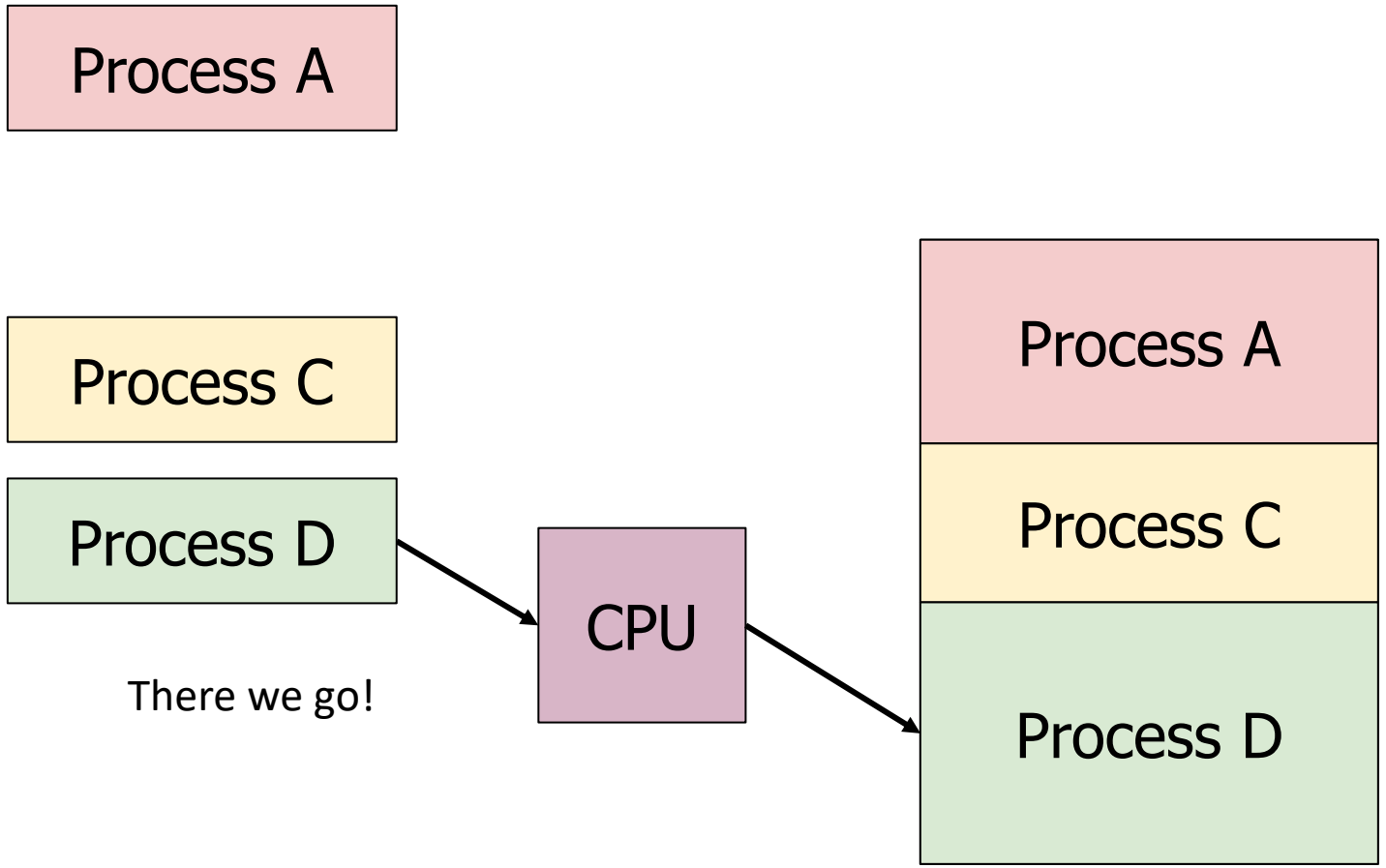
Memory fragmentation



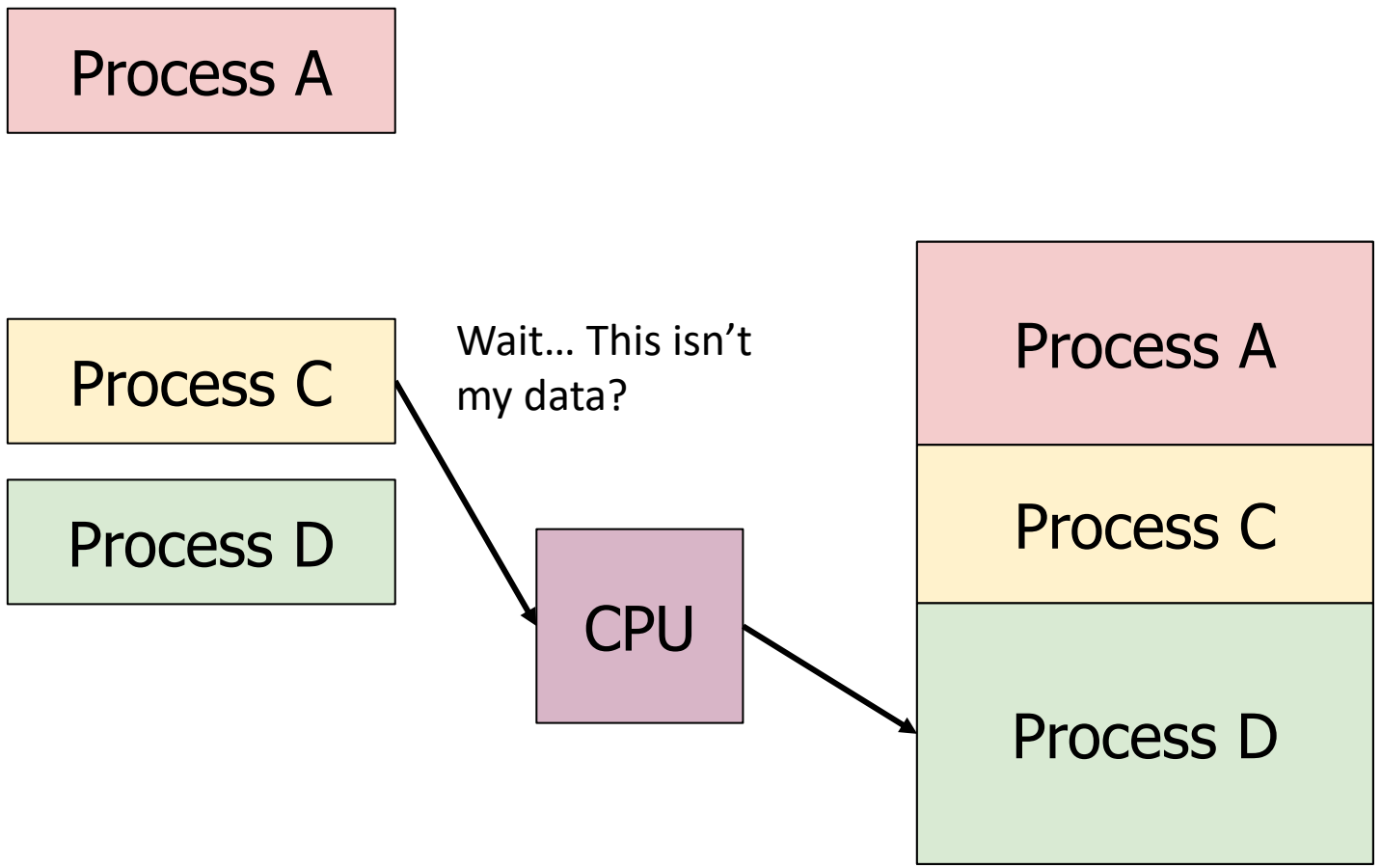
Memory fragmentation



Memory fragmentation



Memory fragmentation

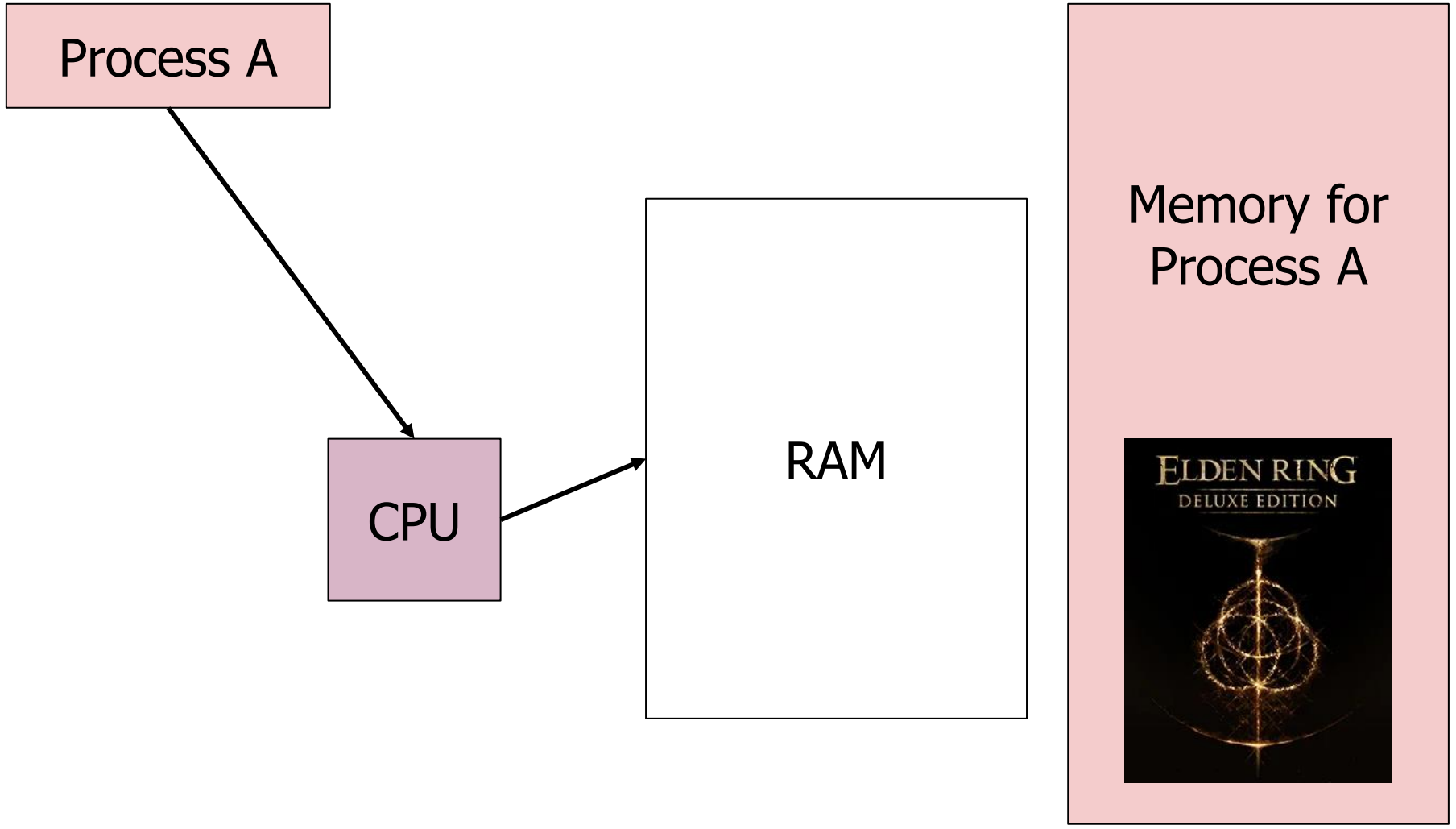


Memory problems

- What are the challenges to supporting this reality?

    1. Which addresses does each process get?
    2. How do we move memory around?
    3. How do we support processes bigger than RAM?

Processes might be bigger than RAM

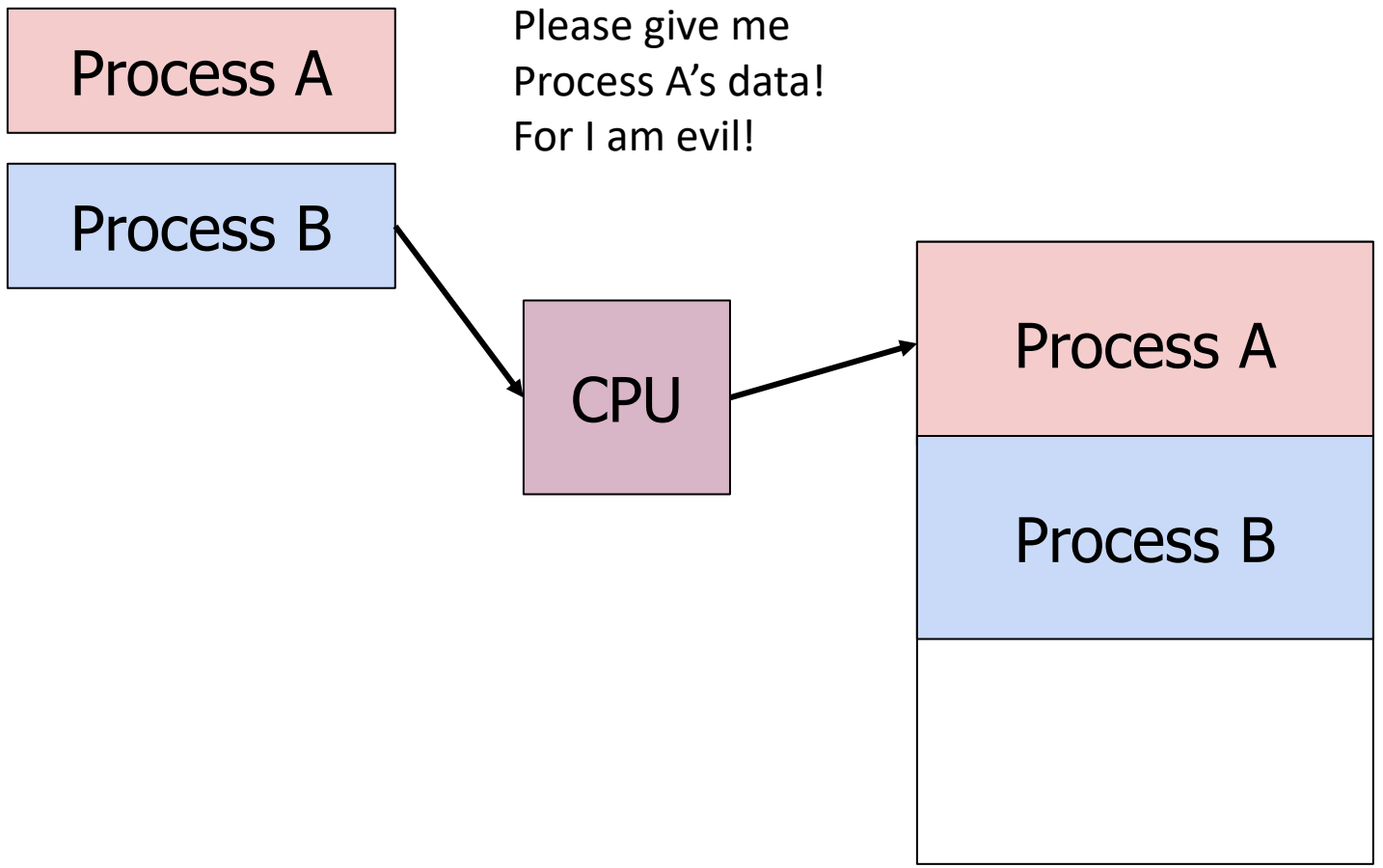


Memory problems

- What are the challenges to supporting this reality?

1. Which addresses does each process get?
2. How do we move memory around?
3. How do we support processes bigger than RAM?
4. How do we protect processes from each other?

Processes can’t be trusted



Memory problems

- What are the challenges to supporting this reality?

1. Which addresses does each process get?
2. How do we move memory around?
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

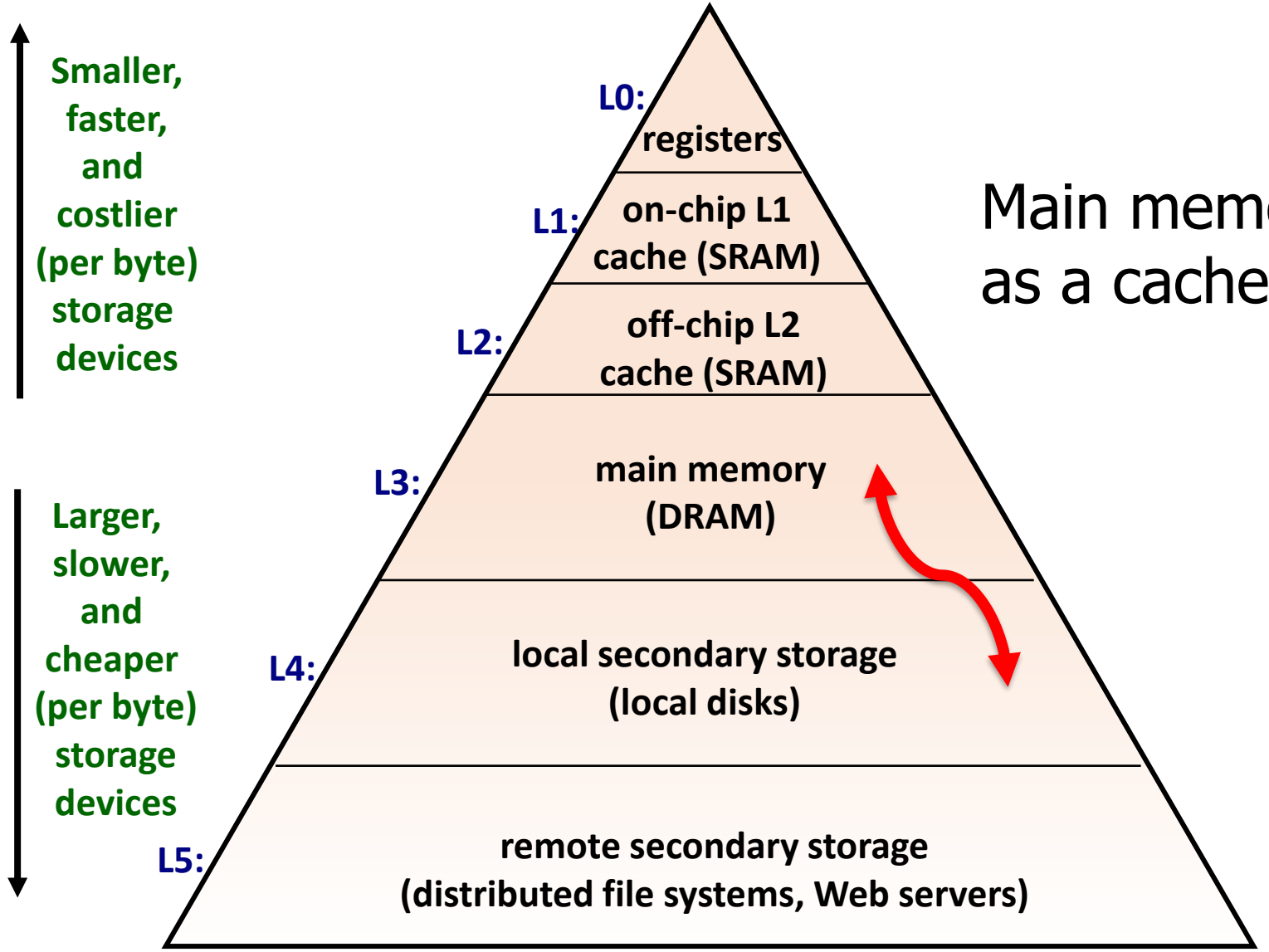
Jeff Dean (Google AI): "Numbers Everyone Should Know"

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

Jim Gray's analogy:

- Registers are in your apartment
- Disk is on Mars

Caching disks



Main memory should act as a cache for disk!

Memory problems

- What are the challenges to supporting this reality?
    1. Which addresses does each process get?
    2. How do we move memory around?
    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!

Outline

- Memory Problems
- **Virtual Memory Concept**
- Virtual Memory Process
- Memory Problems Solved
- Address Translation
- Virtual Memory Summary

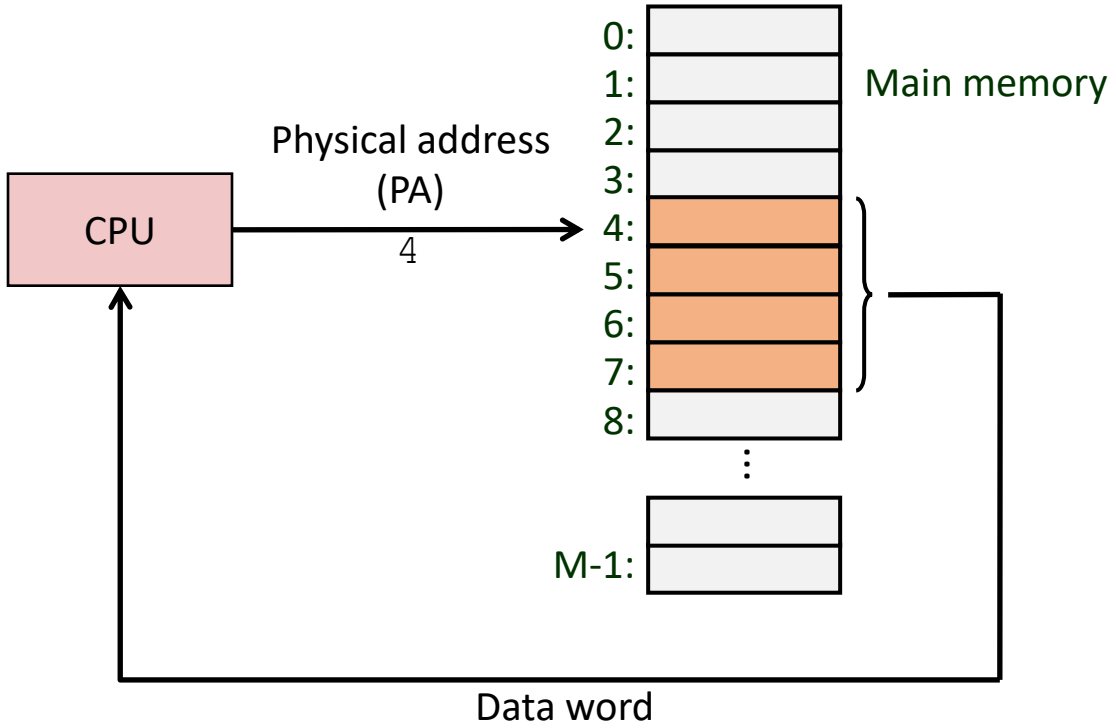
Virtual memory concept

- 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 physical addresses

- Main memory - An array of M contiguous byte-sized cells, each with a unique physical address

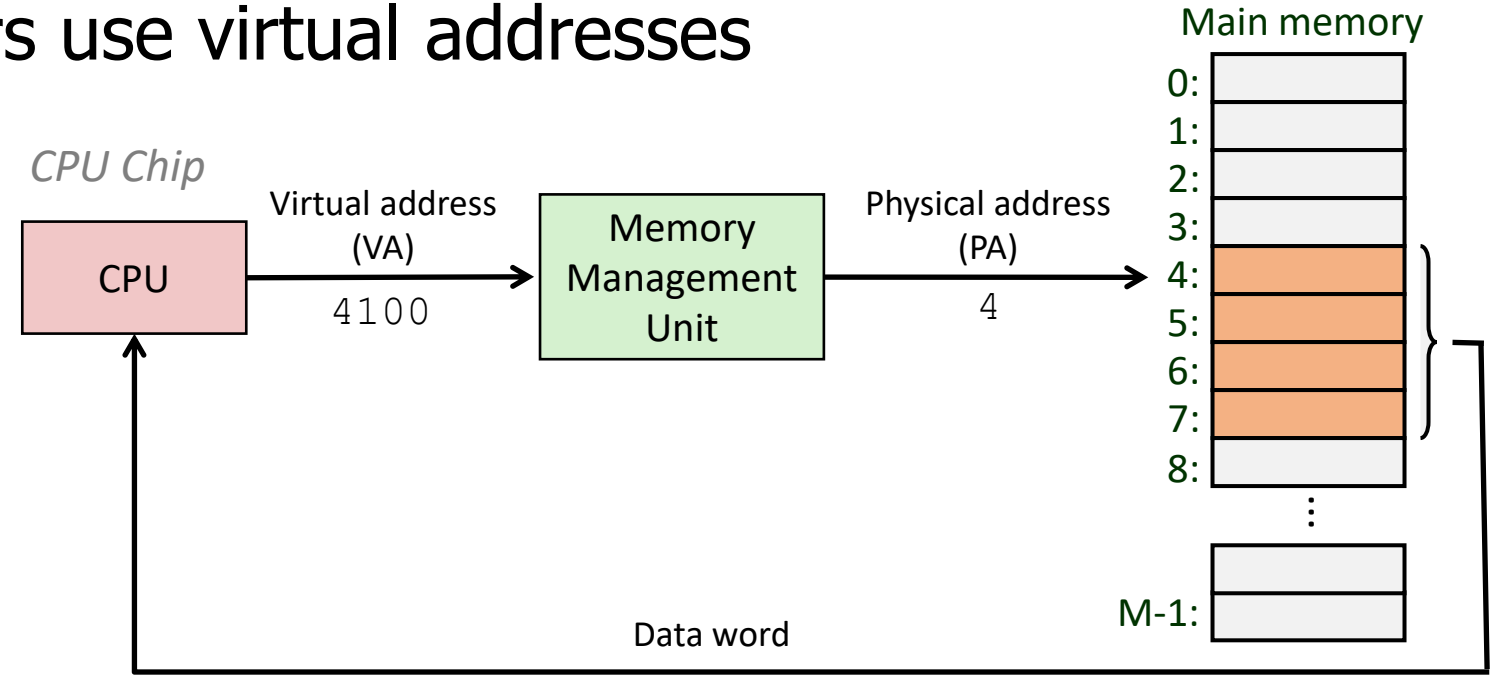
- Physical addressing
  - Most natural way to access it
    - Addresses used by the CPU correspond to bytes in memory
  - Used in simple systems like early PCs and embedded microcontrollers



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 (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

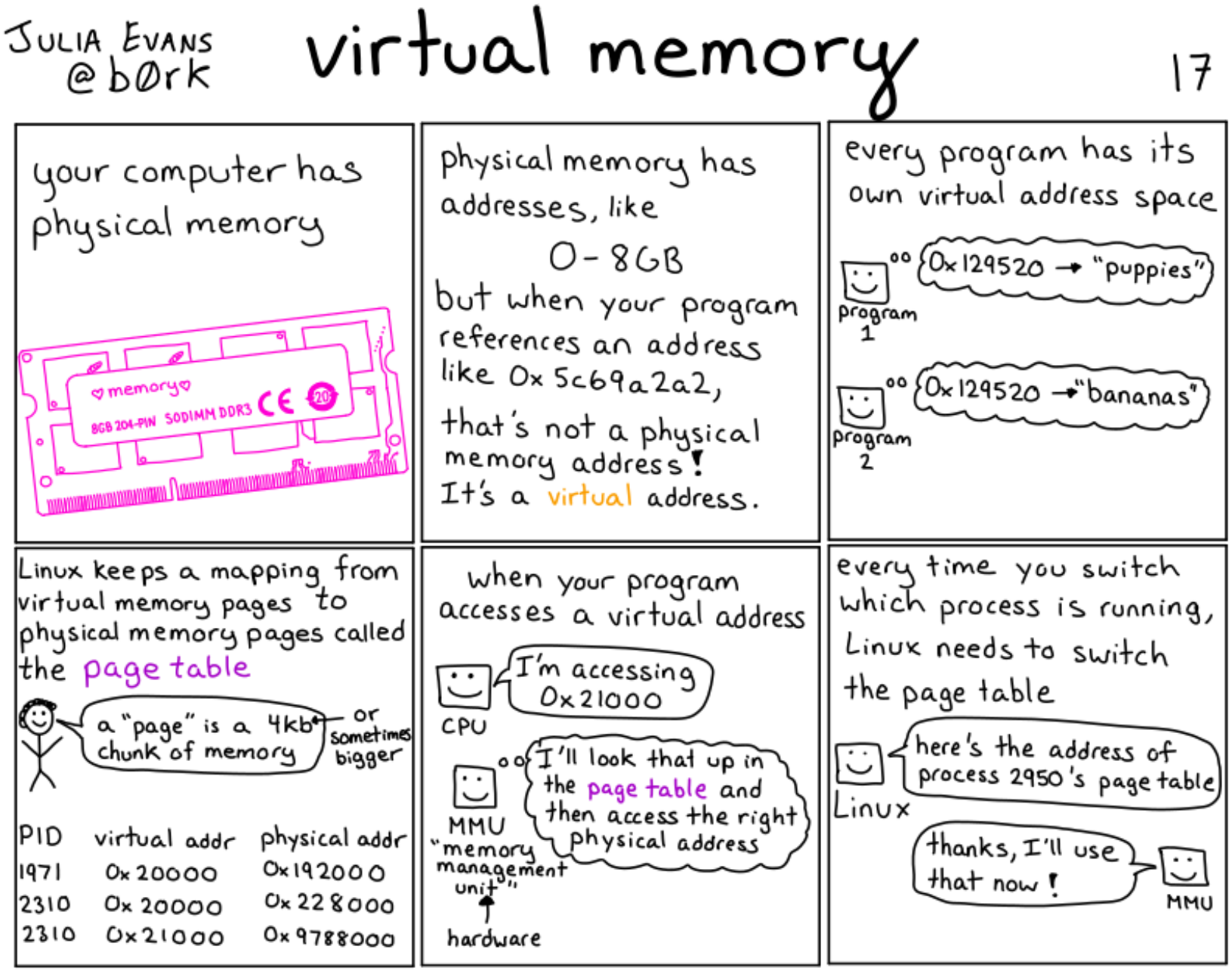
- But multiple of you were running separate ctarget processes at the same time on Moore
  - 0x40000-ish was a **Virtual Address**

- 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 (~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



https://wizardzines.com/comics/virtual-memory/

Outline

- Memory Problems
- Virtual Memory Concept
- **Virtual Memory Process**
- Memory Problems Solved
- Address Translation
- Virtual Memory Summary

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 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 (virtual or physical)
  - Each virtual page will map to a physical page
  - Page size is usually 4 kB or so, occasionally larger (2 MB or 1 GB on x86-64)

Page Tables list Virtual-to-Physical Translations

- A **page table** maps virtual pages to physical pages
  - One page table entry (PTE) per page of virtual memory

- A separate Page Table exists for each running process
  - Each has its own mappings

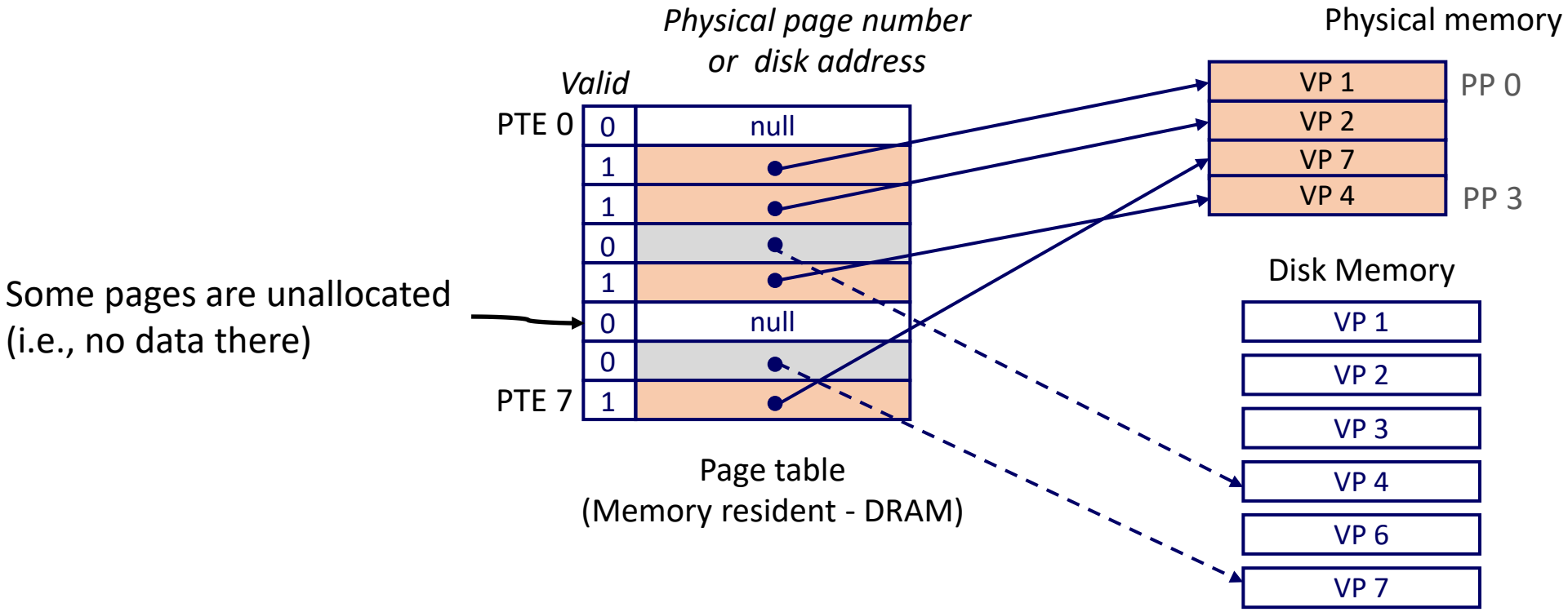
- Page Table Entries could have three possible values
  1. An address for the page in physical memory
  2. An address for the page on disk
  3. Invalid (no actual data exists at this address, **SEGFAULT**)

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!

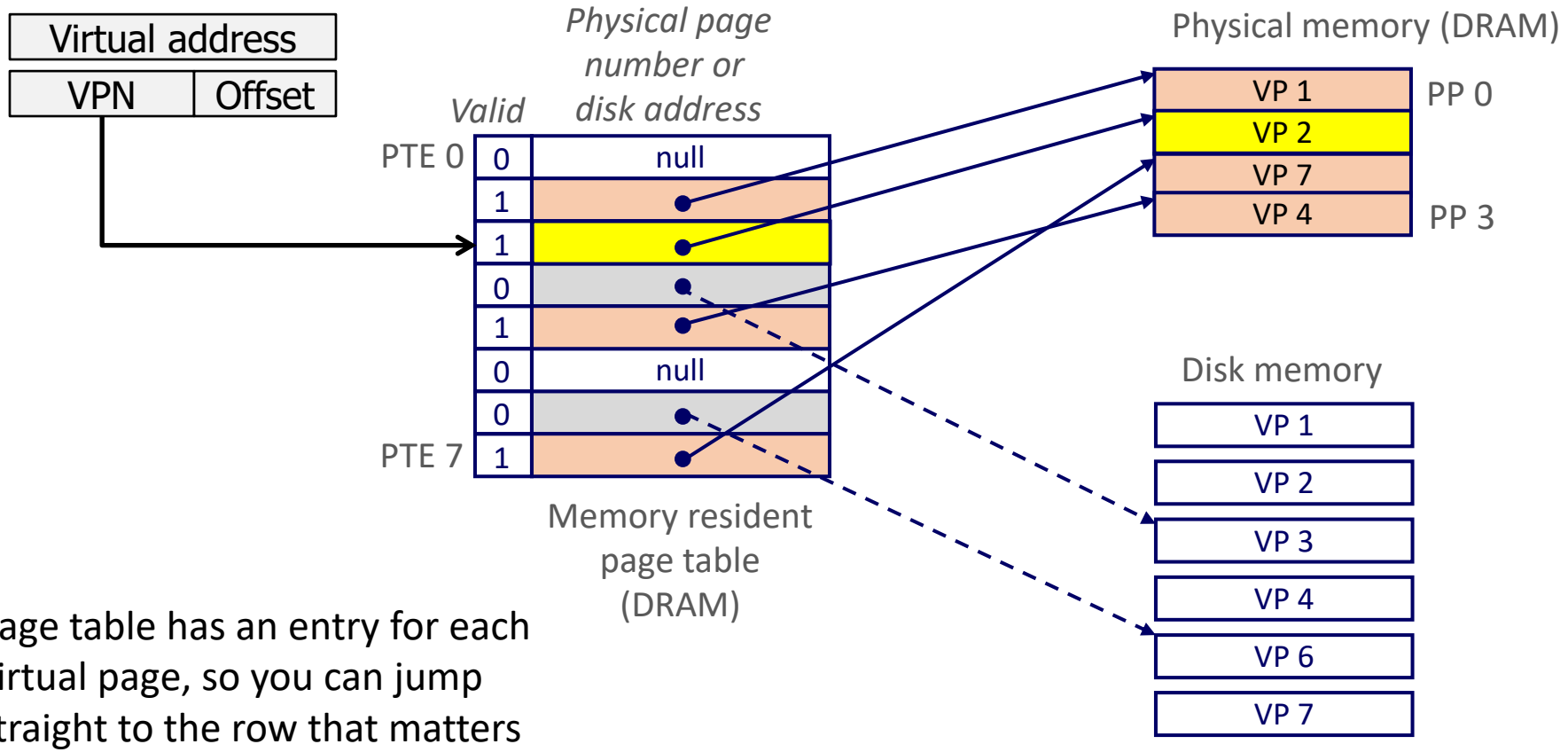
Example Page Table

- The Page Table has an entry for **every** virtual page
  - Valid entries point to memory
  - Invalid entries point to disk, or to nowhere at all



Page Hit

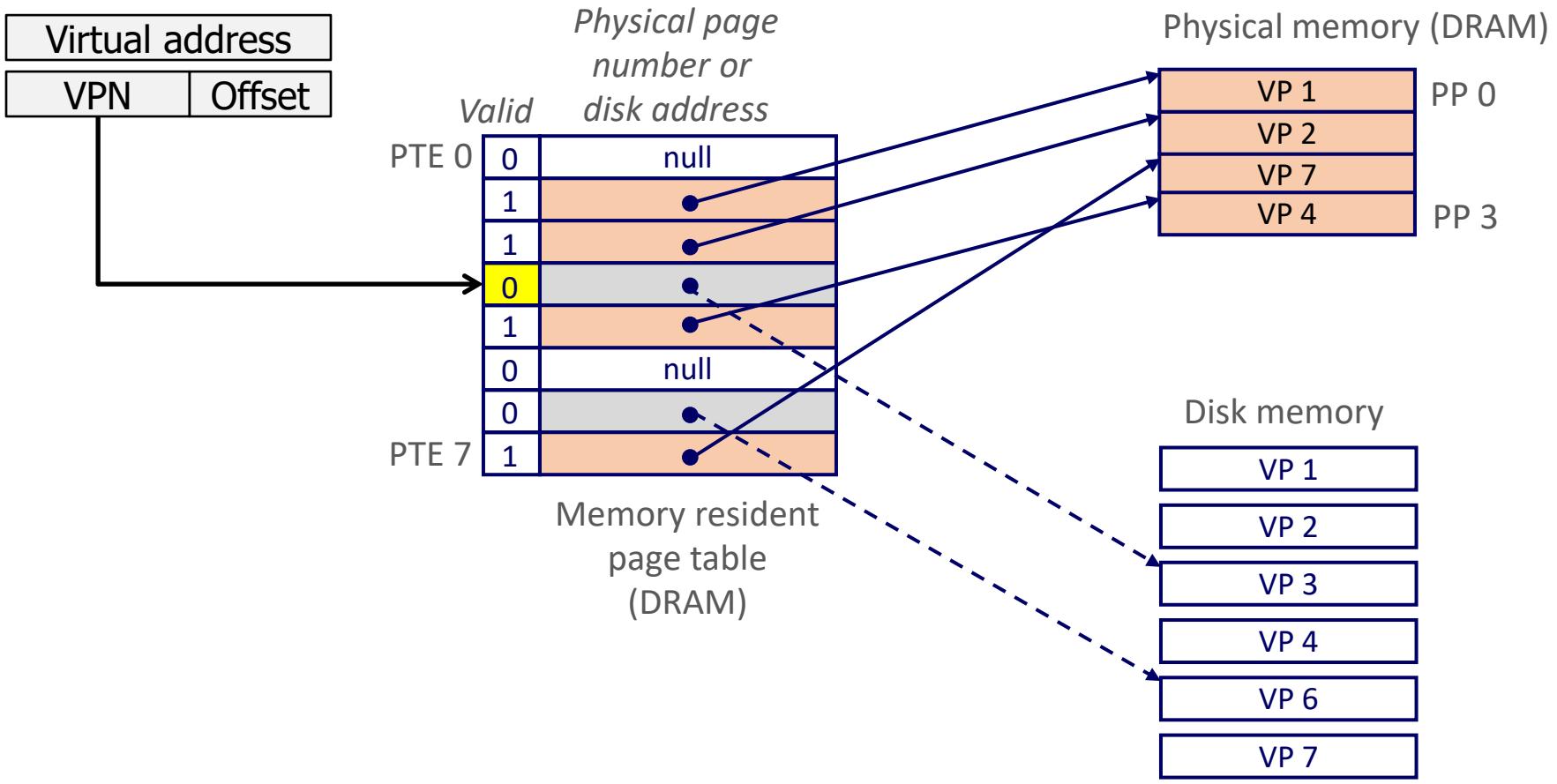
- *Page hit:* reference to a VM word that is in physical memory



Page table has an entry for each virtual page, so you can jump straight to the row that matters

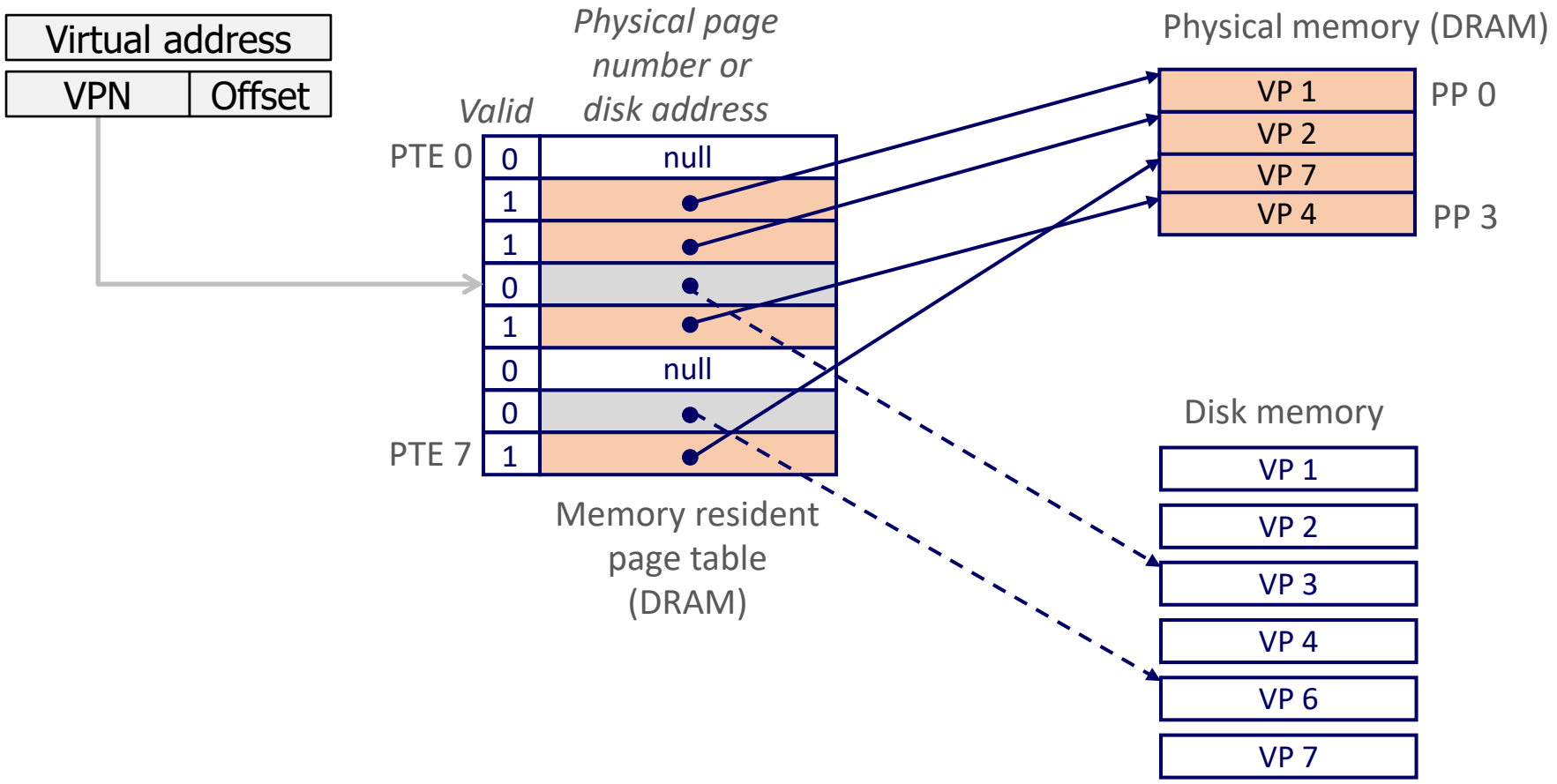
Page Fault

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



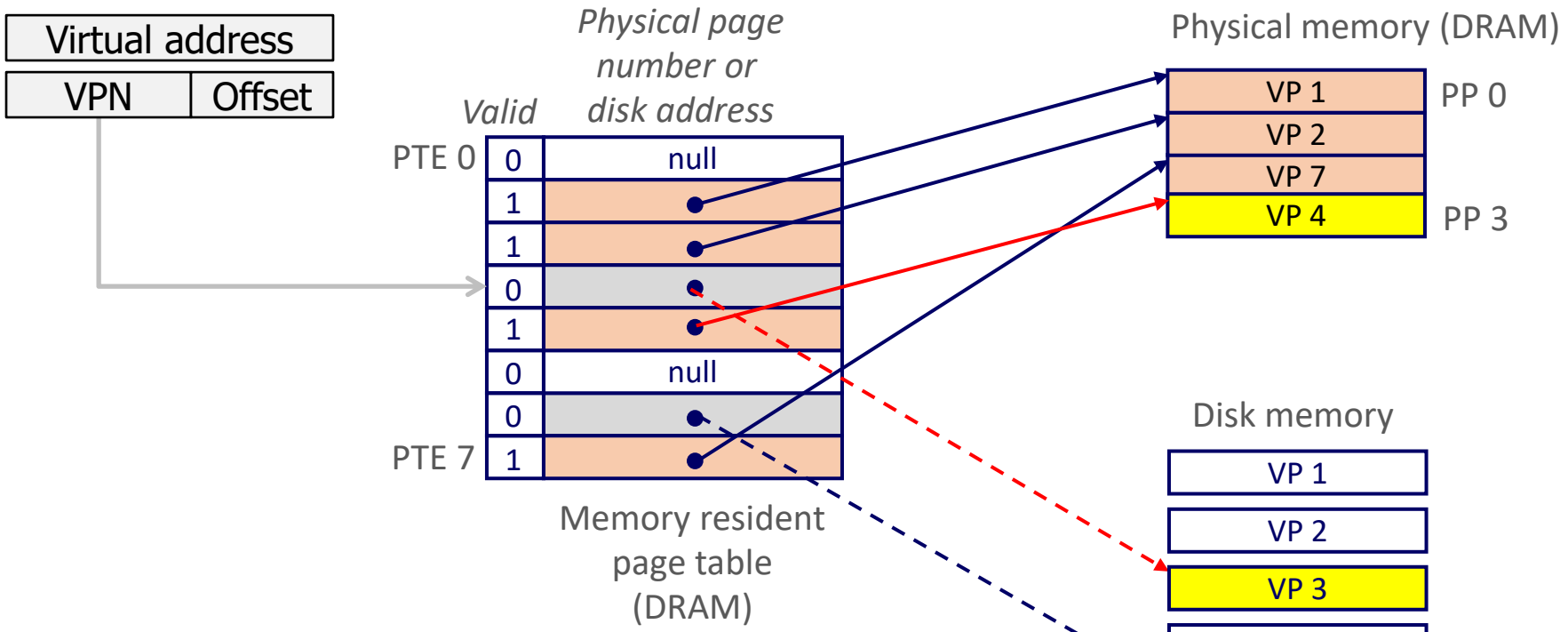
Handling Page Fault

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



Handling Page Fault

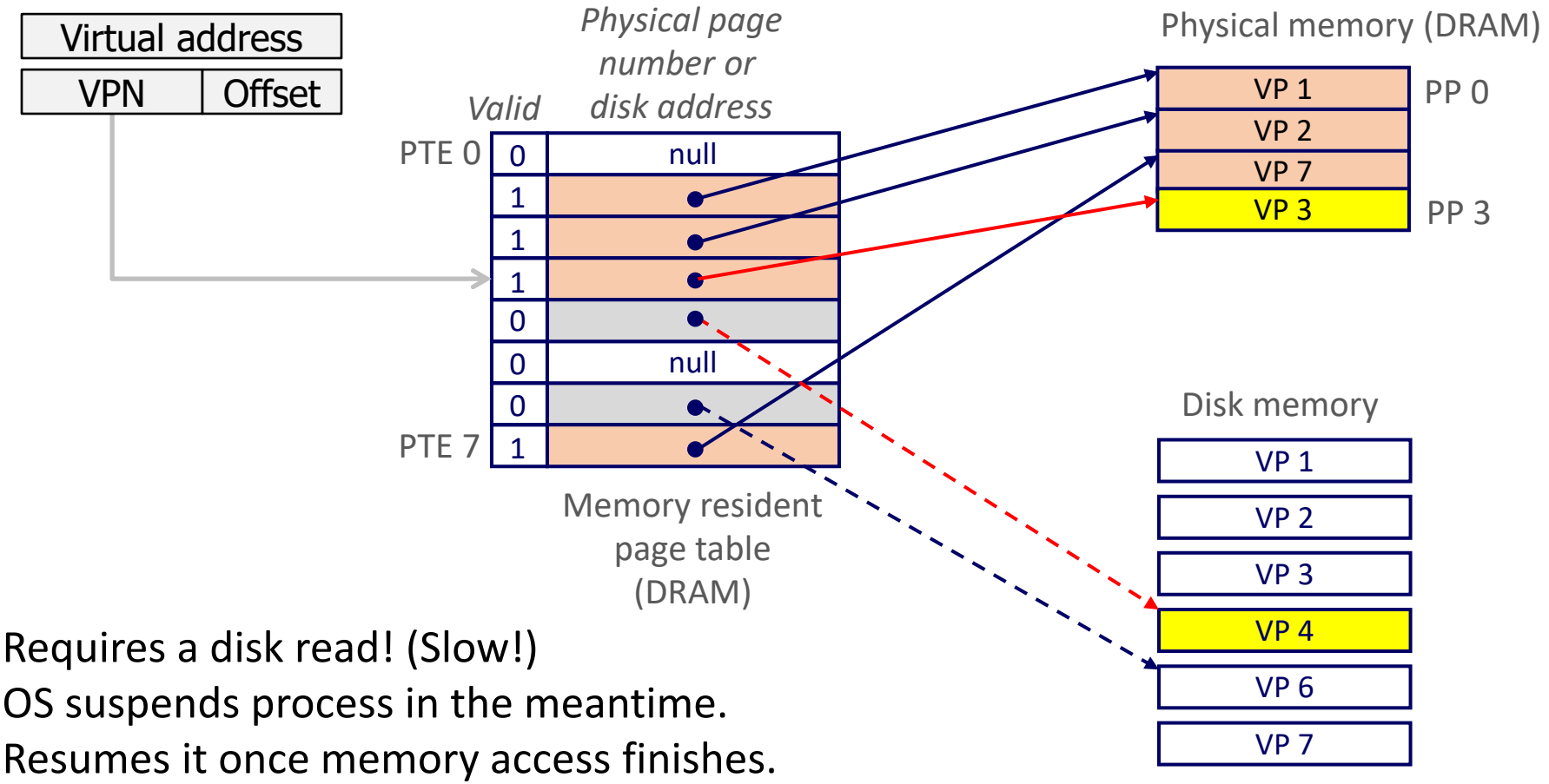
- Page miss causes page fault (a HW exception, OS code kicks in to handle)
- Page fault handler selects a victim to be evicted (here VP 4)



Eviction decision is made by software. So can be pretty sophisticated! (Beyond scope of this class)
→ fewer page faults (if we do it right)

Handling Page Fault

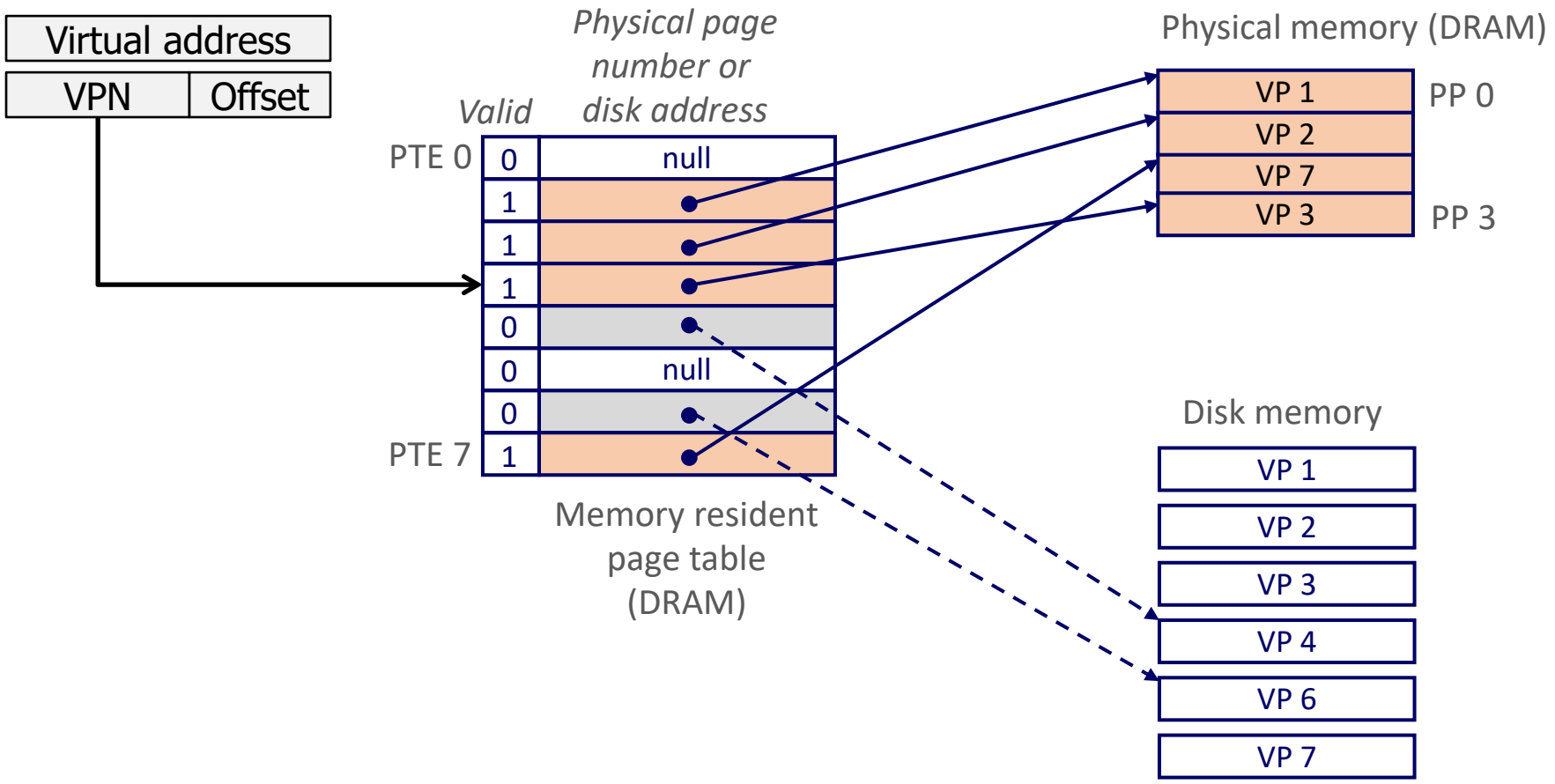
- Page miss causes page fault (a HW exception, OS code kicks in to handle)
- Page fault handler selects a victim to be evicted (here VP 4)
- The victim page is swapped with the disk block of the requested address



Requires a disk read! (Slow!)
OS suspends process in the meantime.
Resumes it once memory access finishes.

Handling Page Fault

- Offending instruction is restarted: page hit this time!



VM as a Tool for Caching

- We're using physical memory as a *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

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

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

- Memory Problems
- Virtual Memory Concept
- Virtual Memory Process
- **Memory Problems Solved**
- Address Translation
- Virtual Memory Summary

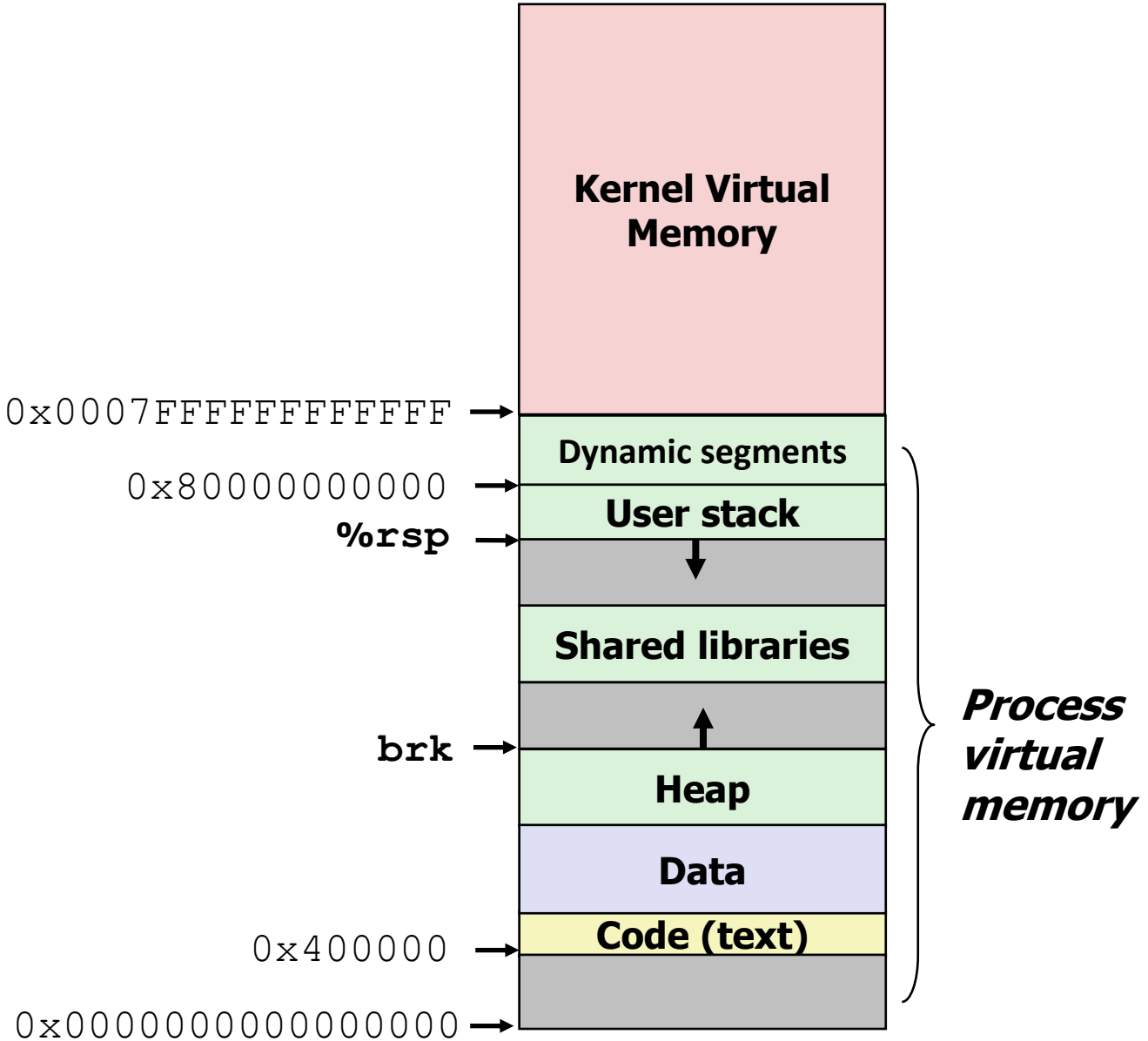
Memory problems

- What are the challenges to supporting this reality?

1. Which addresses does each process get?
2. How do we move memory around?
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? ✓
   Use RAM as a cache for disk

Which addresses do processes get?

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



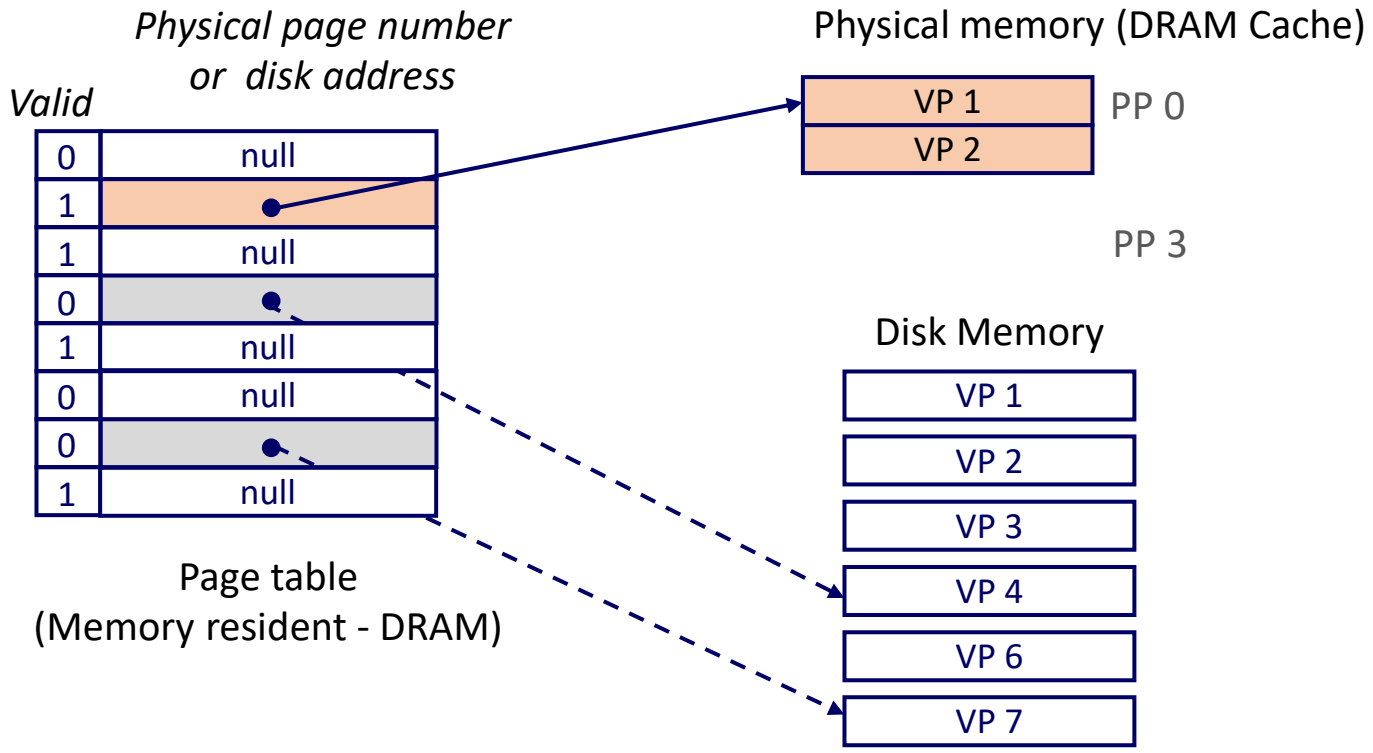
Memory problems

- What are the challenges to supporting this reality?

1. Which addresses does each process get? ✓
   Whatever virtual addresses they want
2. How do we move memory around?
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? ✓
   Use RAM as a cache for disk

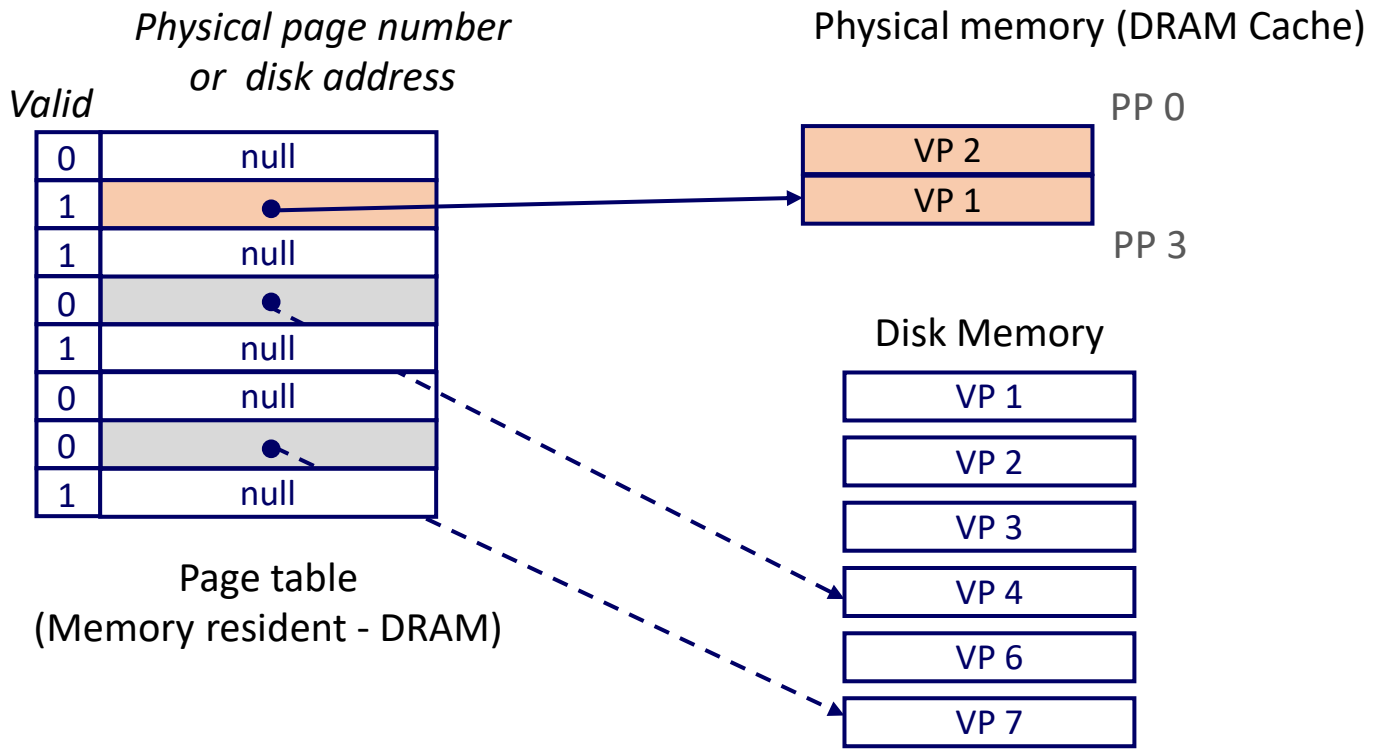
How do we move memory around?

- Just change the page table entry!



How do we move memory around?

- Just change the page table entry!
  - Same virtual address points at a different physical address
- Usually only happens when pages are swapped to disk and then later brought back



Memory problems

- What are the challenges to supporting this reality?

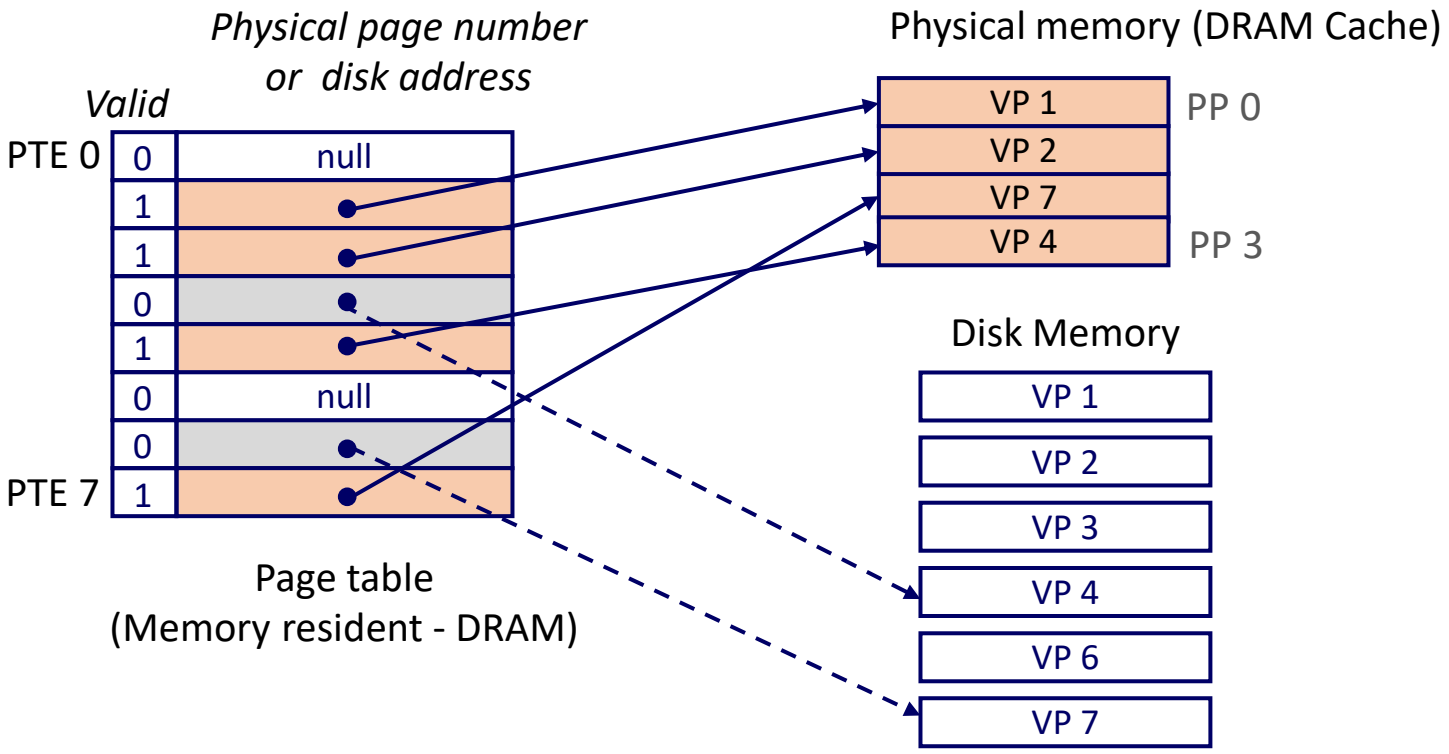
1. Which addresses does each process get? ✓
   Whatever virtual addresses they want
2. How do we move memory around? ✓
   Update page table entries
3. How do we support processes bigger than RAM?

4. How do we protect processes from each other?

5. How do we deal with how incredibly slow disk is? ✓
   Use RAM as a cache for disk

How do we support processes bigger than RAM?

- 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

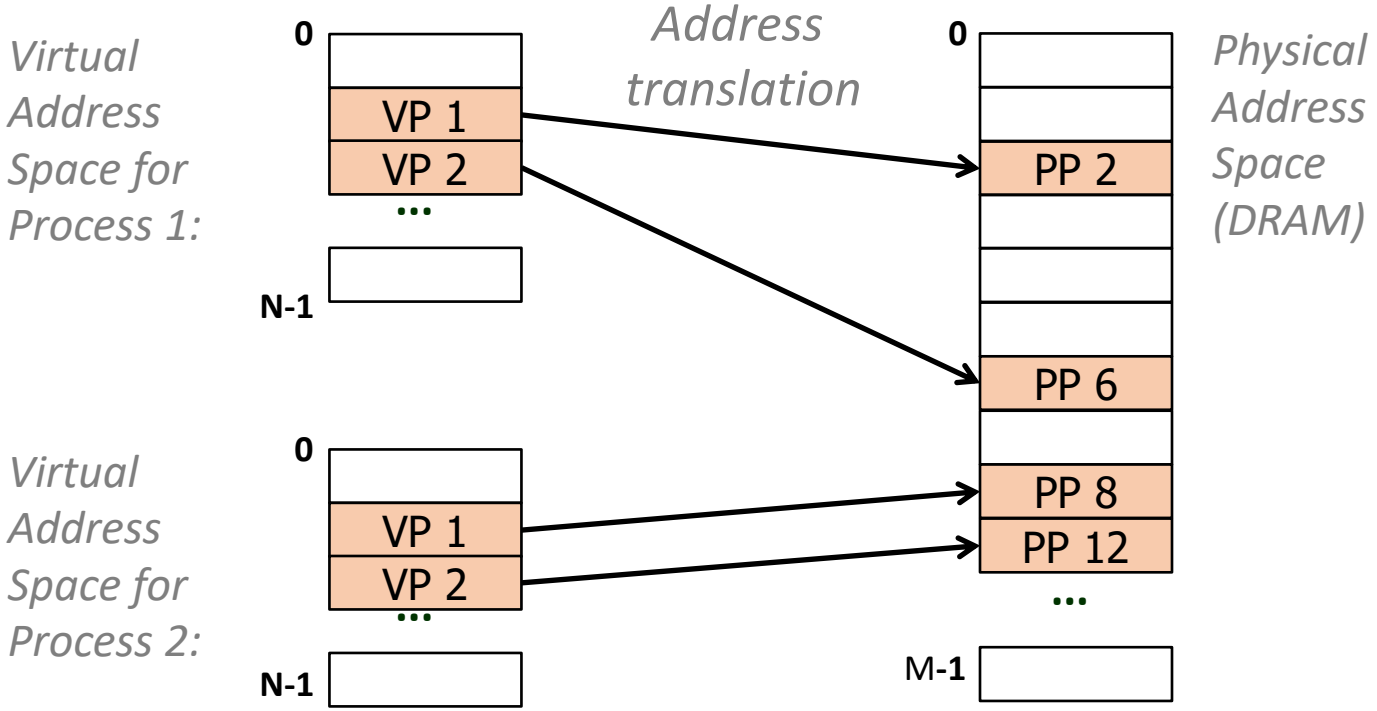
- What are the challenges to supporting this reality?

1. Which addresses does each process get? ✓
   Whatever virtual addresses they want
2. How do we move memory around? ✓
   Update page table entries
3. How do we support processes bigger than RAM? ✓
   Leave some pages on disk
4. How do we protect processes from each other?

5. How do we deal with how incredibly slow disk is? ✓
   Use RAM as a cache for disk

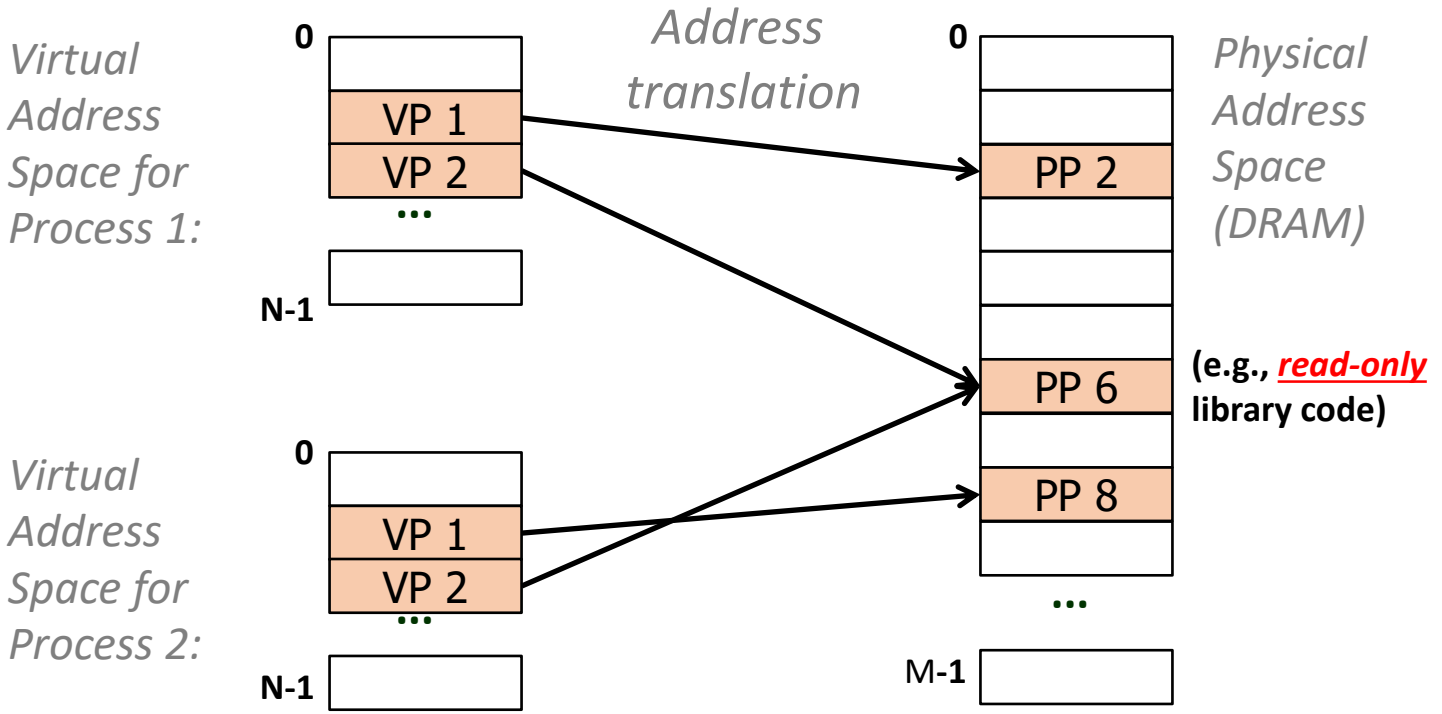
How do we protect processes from each other?

- 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



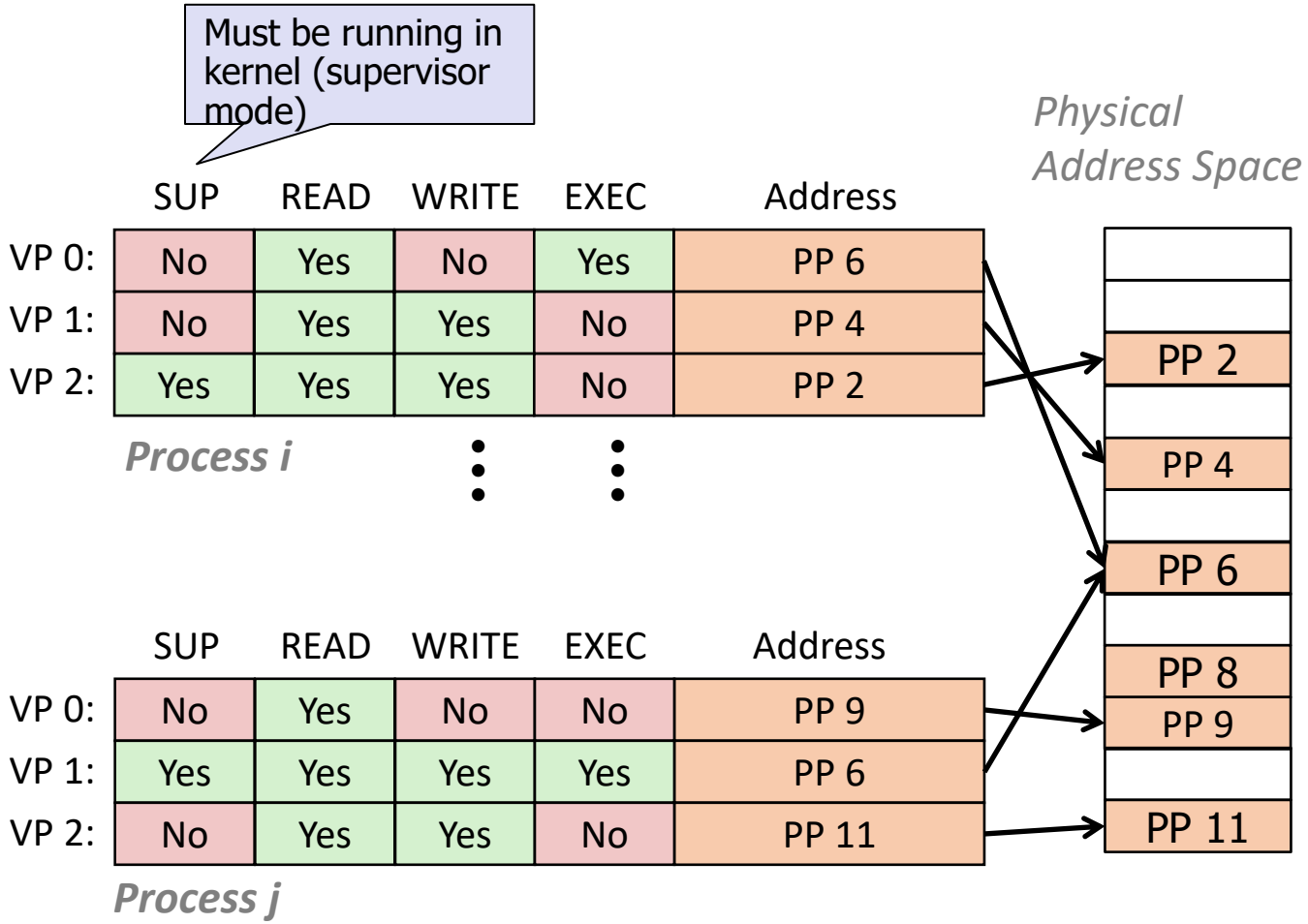
Enabling shared libraries

- 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?
  - Mark a page as read-only
  - Keep a page in memory, but only the OS can touch it
- Extend PTEs with permission bits!
  - Page fault handler checks these before remapping
- HW enforces this protection (trap into OS if violation occurs)



Memory problems

- What are the challenges to supporting this reality?

1. Which addresses does each process get? ✓
   Whatever virtual addresses they want
2. How do we move memory around? ✓
   Update page table entries
3. How do we support processes bigger than RAM? ✓
   Leave some pages on disk
4. How do we protect processes from each other? ✓
   Don’t overlap virtual address spaces + permission bits
5. How do we deal with how incredibly slow disk is? ✓
   Use RAM as a cache for disk

Outline

- Memory Problems
- Virtual Memory Concept
- Virtual Memory Process
- Memory Problems Solved
- **Address Translation**
- Virtual Memory Summary

Address Translation

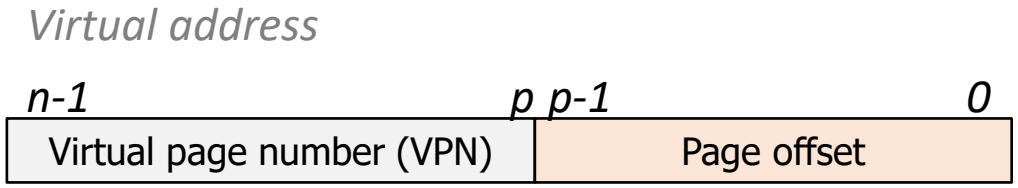
- Goal: Given virtual address, find corresponding physical address
  - (Or get a page fault if the page is not in memory)
  - Translation done by Memory Management Unit (hardware)
  - But mapping itself is maintained by OS (software)
    - Just a table in memory!

- 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)

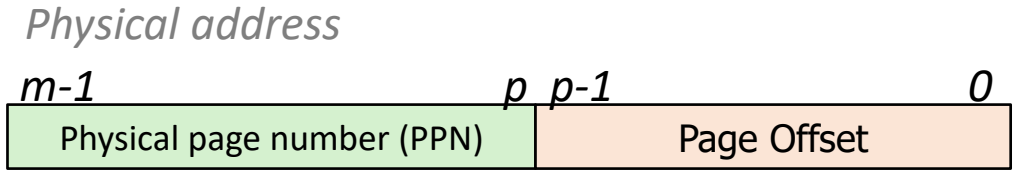
Breaking down virtual addresses

- Basic Parameters
  - **N = 2^n** : Number of addresses in virtual address space
  - **M = 2^m** : Number of addresses in physical address space. m ≤ n (usually much less)
  - **P = 2^p** : Page size (bytes)

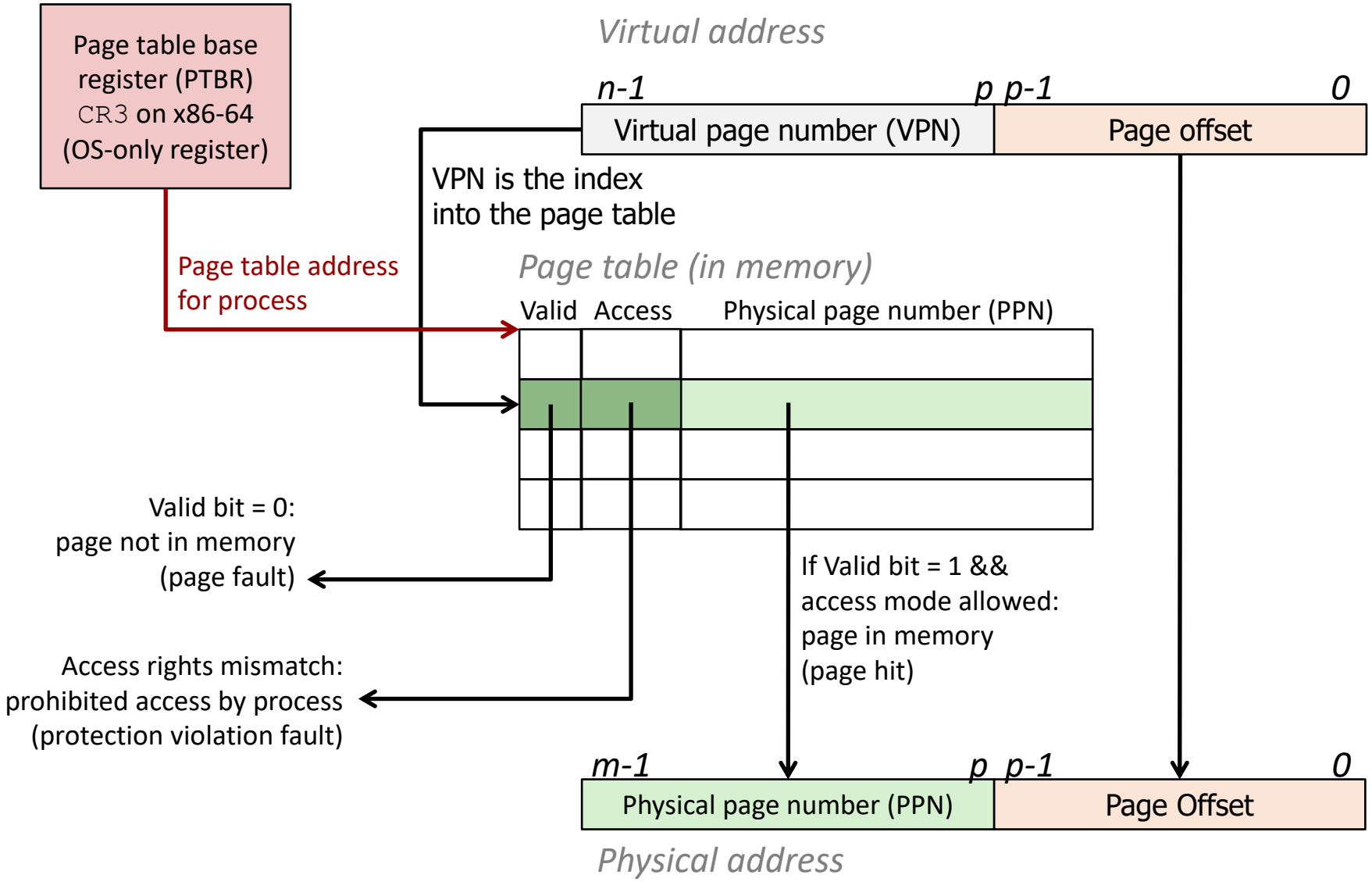
- Components of the virtual address (VA)
  - Virtual page number (VPN): **n-p** bits
  - Page Offset: **p** bits



- Components of the physical address (PA)
  - Physical page number (PPN): **m-p** bits
  - Page Offset (same offset as VA): **p** bits



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?

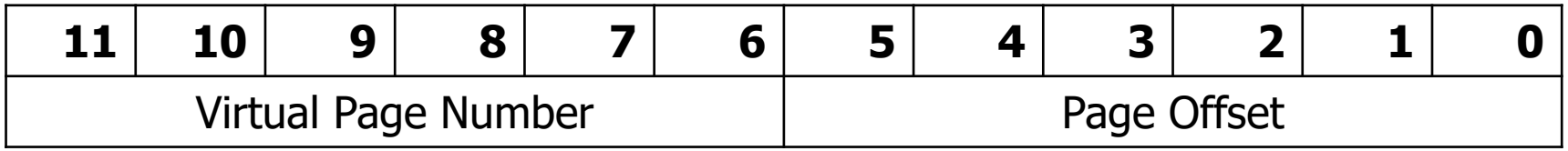
11	10	9	8	7	6	5	4	3	2	1	0

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!

1. How do we split Virtual Addresses into VPN and Offset?
   - Offset is based on page size: 64-bytes ⇨ 6 bits. All the rest are VPN



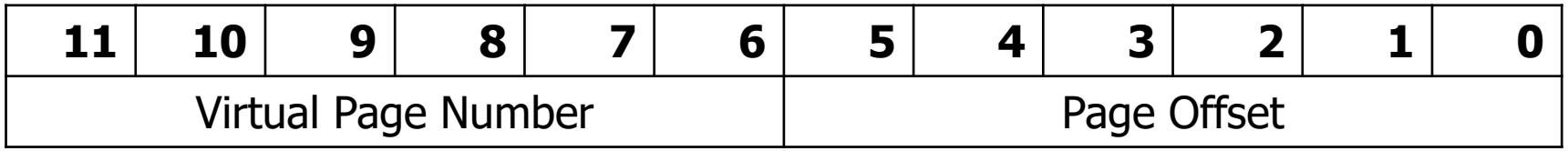
2. How big are Physical Page Numbers?

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!

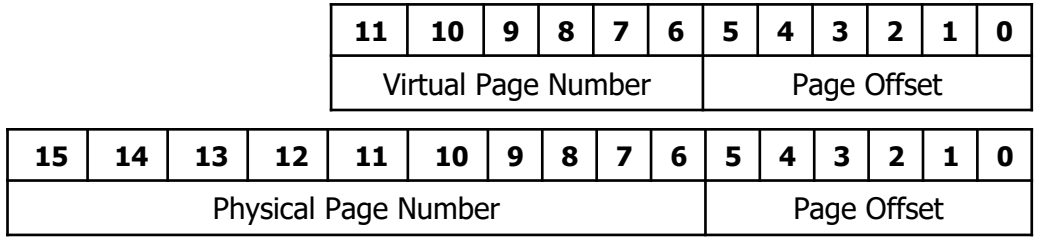
1. How do we split Virtual Addresses into VPN and Offset?
   - Offset is based on page size: 64-bytes ⇨ 6 bits. All the rest are VPN



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

Virtual memory example

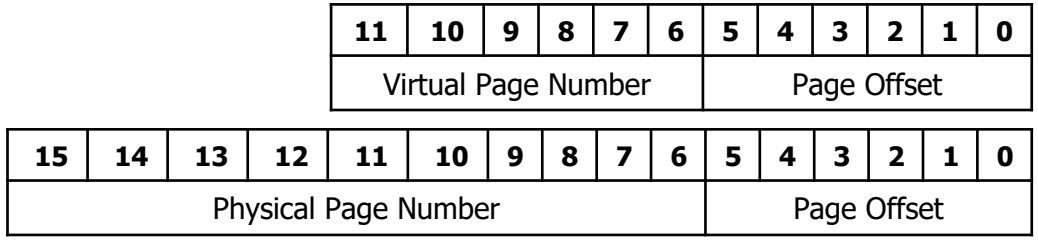
- Parameters
  - Virtual addresses are 12-bits
  - Physical addresses are 16-bits
  - Page size is 64 bytes



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

Virtual memory example

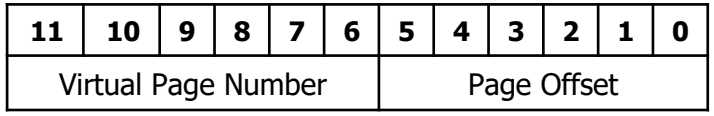
- Parameters
  - Virtual addresses are 12-bits
  - Physical addresses are 16-bits
  - Page size is 64 bytes

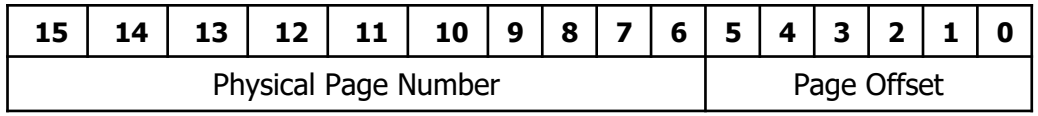


- Translate:
- Virtual address: 0x3F0
  - VPN: 0b001111
  - Offset: 0b110000

Virtual memory example

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





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

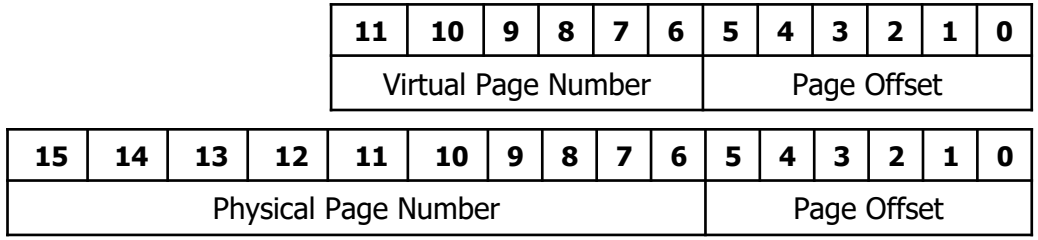
VPN	PPN	Valid
0x00	0x123	1
0x01	0x156	1
0x02	0x143	1
0x03	0x16F	1
0x04	0x1FF	0
0x05	0x107	0
0x06	0x100	0
0x07	0x1C0	0
0x08	0x1D8	0
0x09	0x1BF	0
0x0A	0x000	1
0x0B	0x3FF	1
0x0C	0x308	0
0x0D	0x3FD	0
0x0E	0x111	1
0x0F	0x1F0	1

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



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

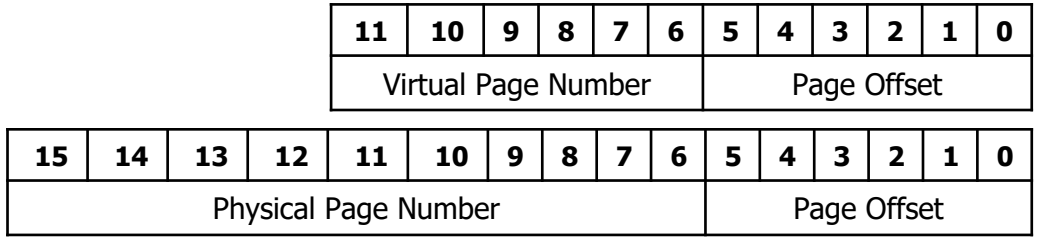
VPN	PPN	Valid
0x00	0x123	1
0x01	0x156	1
0x02	0x143	1
0x03	0x16F	1
0x04	0x1FF	0
0x05	0x107	0
0x06	0x100	0
0x07	0x1C0	0
0x08	0x1D8	0
0x09	0x1BF	0
0x0A	0x000	1
0x0B	0x3FF	1
0x0C	0x308	0
0x0D	0x3FD	0
0x0E	0x111	1
0x0F	0x1F0	1

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



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

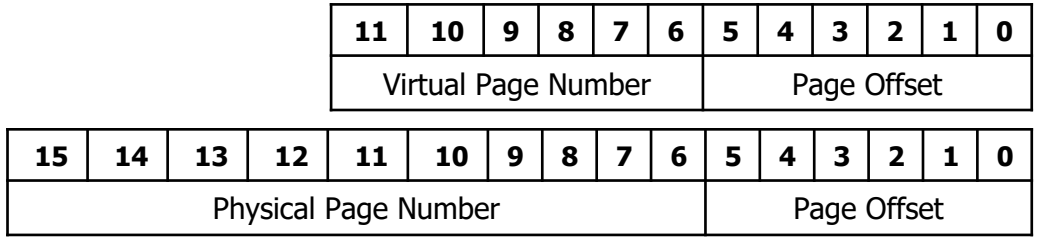
VPN	PPN	Valid
0x00	0x123	1
0x01	0x156	1
0x02	0x143	1
0x03	0x16F	1
0x04	0x1FF	0
0x05	0x107	0
0x06	0x100	0
0x07	0x1C0	0
0x08	0x1D8	0
0x09	0x1BF	0
0x0A	0x000	1
0x0B	0x3FF	1
0x0C	0x308	0
0x0D	0x3FD	0
0x0E	0x111	1
0x0F	0x1F0	1

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:
  - 0b011111000110000
  - 0x7C30

Break + Question

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



- Translate:
- Virtual address: 0x500
  - 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

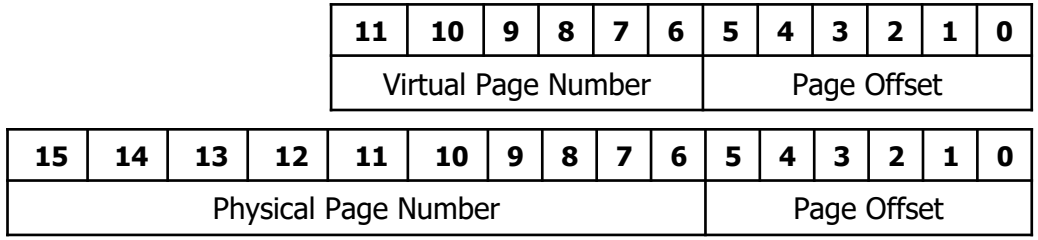
VPN	PPN	Valid
0x10	0x237	1
0x11	0x236	1
0x12	0x2B0	1
0x13	0x280	0
0x14	0x120	0
Continues on…		

- PPN:
- Offset:

- Physical address:

Break + Question

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



- Translate:
- Virtual address: 0x500
  - 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

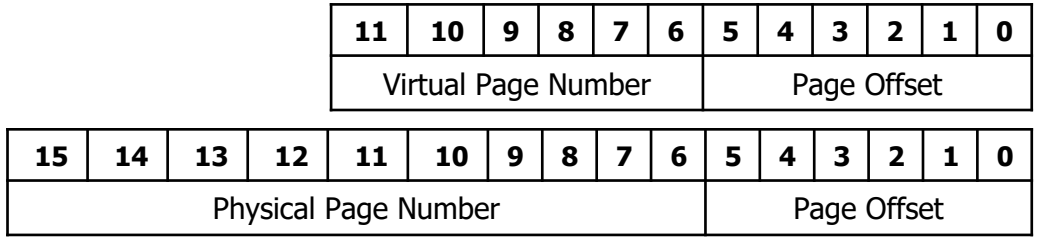
VPN	PPN	Valid
0x10	0x237	1
0x11	0x236	1
0x12	0x2B0	1
0x13	0x280	0
0x14	0x120	0
Continues on…		

- 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



- Translate:
- Virtual address: 0x0D6
  - 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

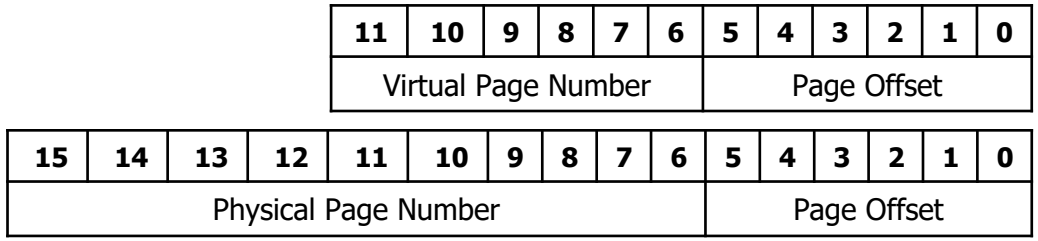
VPN	PPN	Valid
0x10	0x237	1
0x11	0x236	1
0x12	0x2B0	1
0x13	0x280	0
0x14	0x120	0
Continues on…		

- PPN:
- Offset:

- Physical address:

Break + Practice again

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



- Translate:
- Virtual address: 0x0D6
  - 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:
  - 0b010110111101011O
  - 0x5BD6

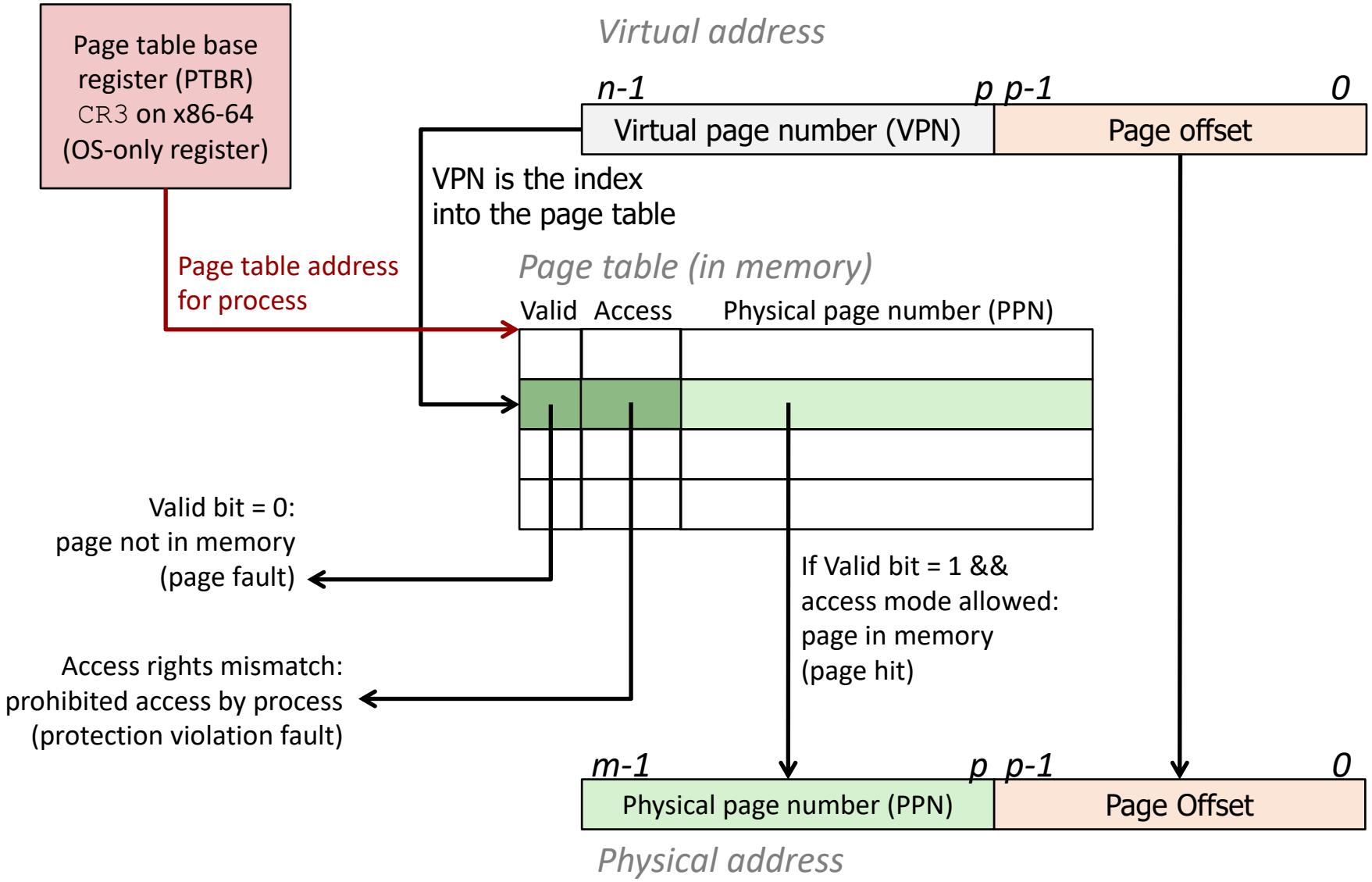
Outline

- Memory Problems
- Virtual Memory Concept
- Virtual Memory Process
- Memory Problems Solved
- Address Translation
- **Virtual Memory Summary**

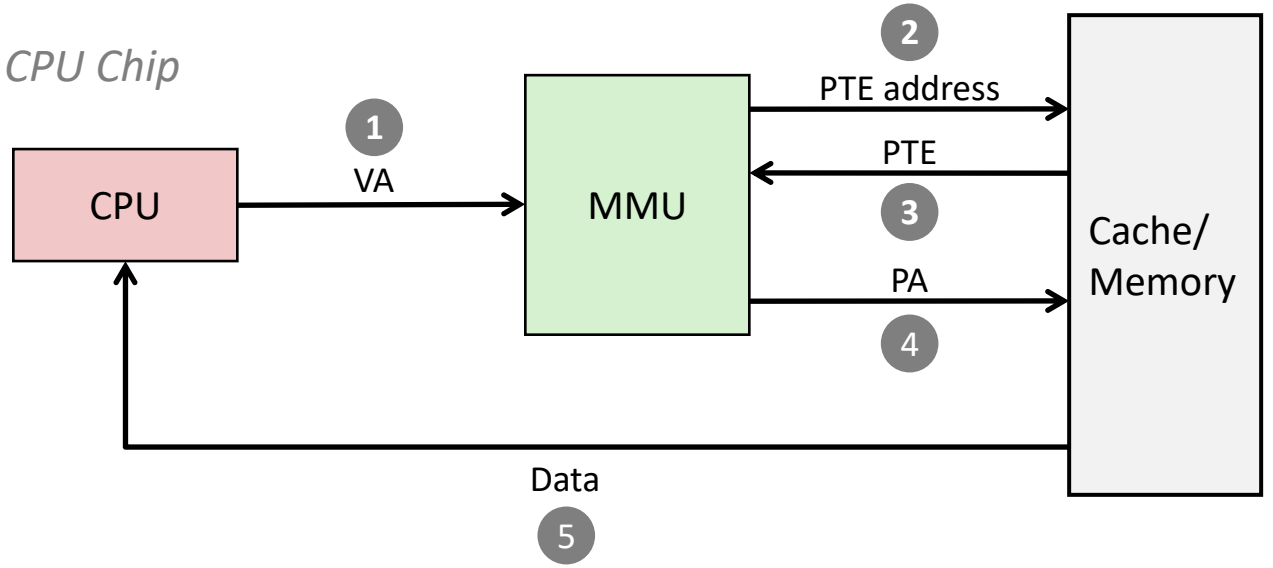
Virtual memory idea

- 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



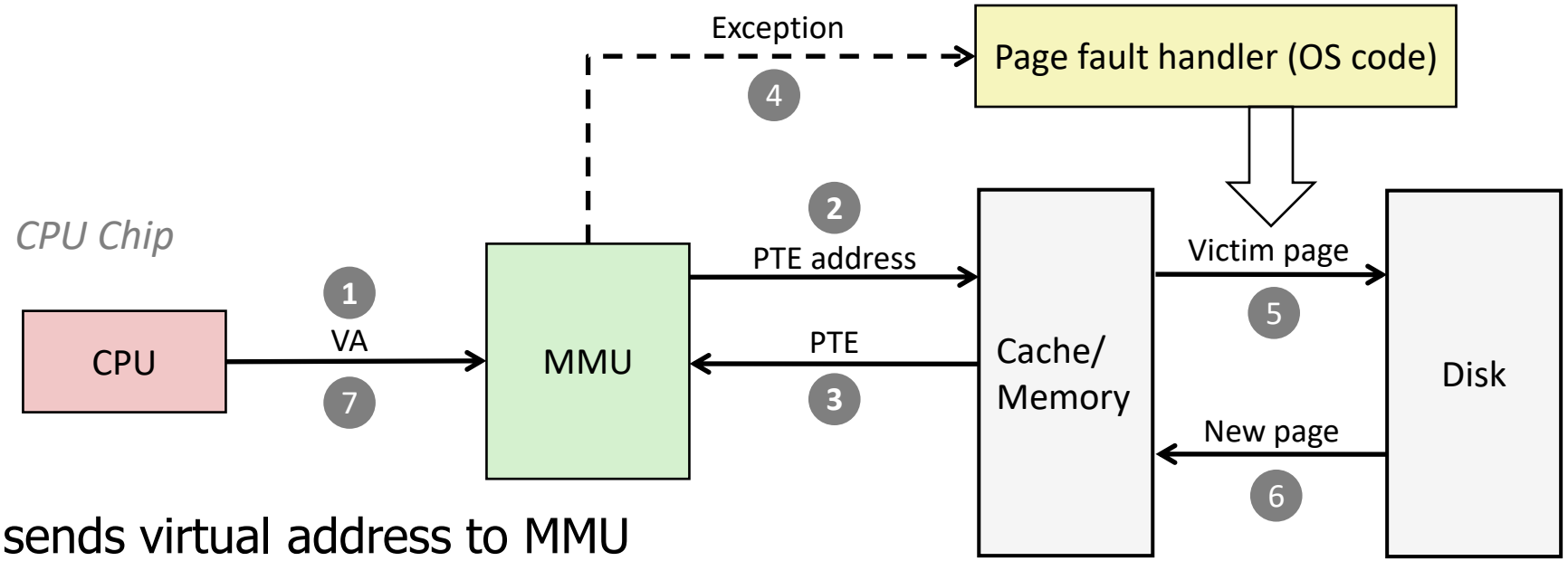
1) Processor sends virtual address to MMU

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

Outline

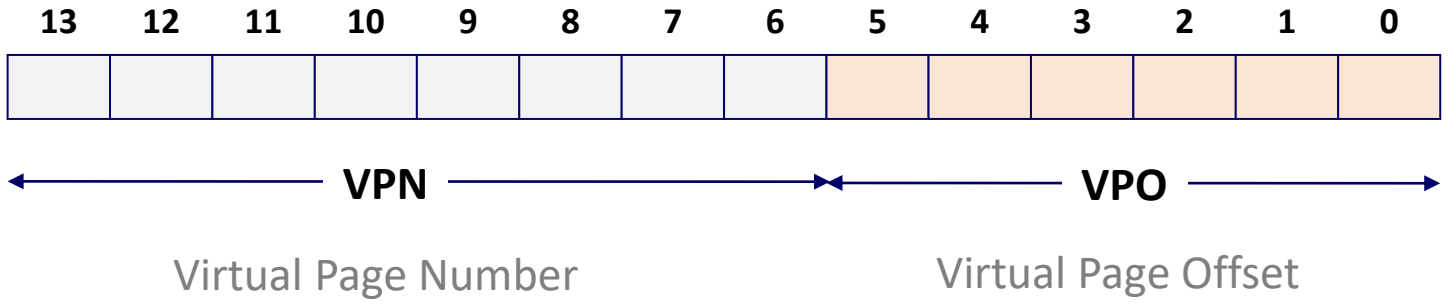
- Memory Problems
- Virtual Memory Concept
- Virtual Memory Process
- Memory Problems Solved
- Address Translation
- Virtual Memory Summary

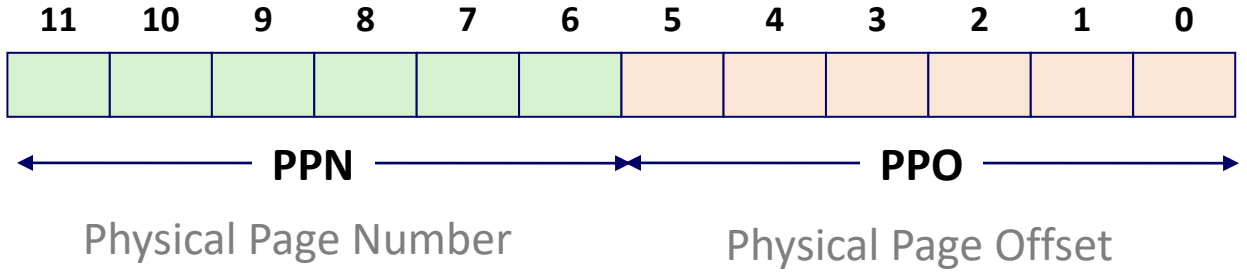
Outline

- 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)

VPN	*PPN*	*Valid*
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	*PPN*	*Valid*
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

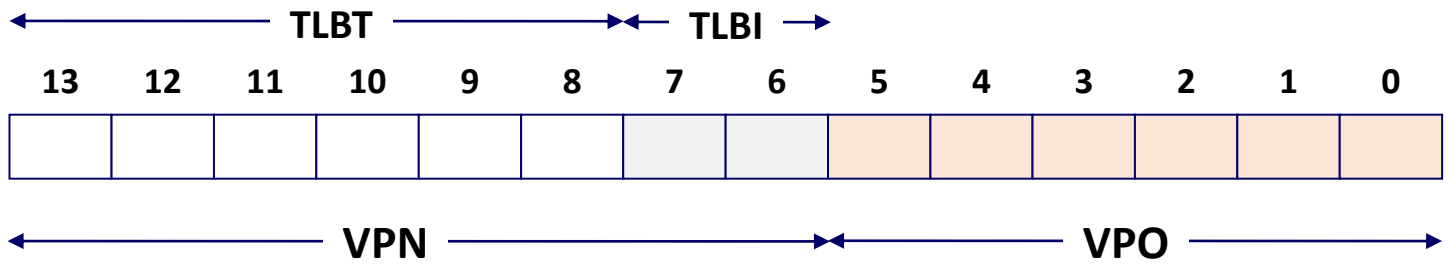
…

VPN	*PPN*	*Valid*
2E	-	0

⋮

Simple Memory System: TLB

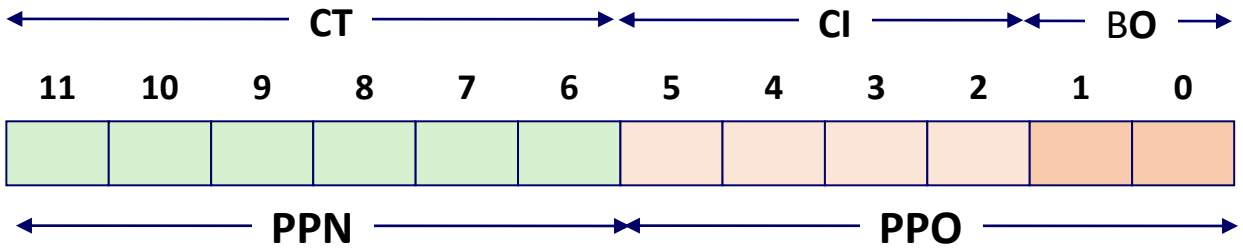
- 16 entries
- 4-way associative



Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
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



Idx	*Tag*	*Valid*	*B3*	*B2*	*B1*	*B0*
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

Idx	*Tag*	*Valid*	*B3*	*B2*	*B1*	*B0*
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	–	–	–	–

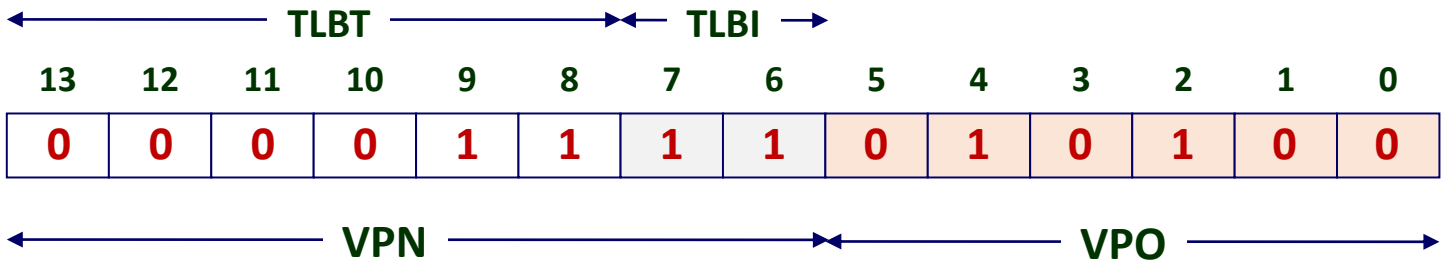
Address Translation Example #1

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

movb (%rcx), %al

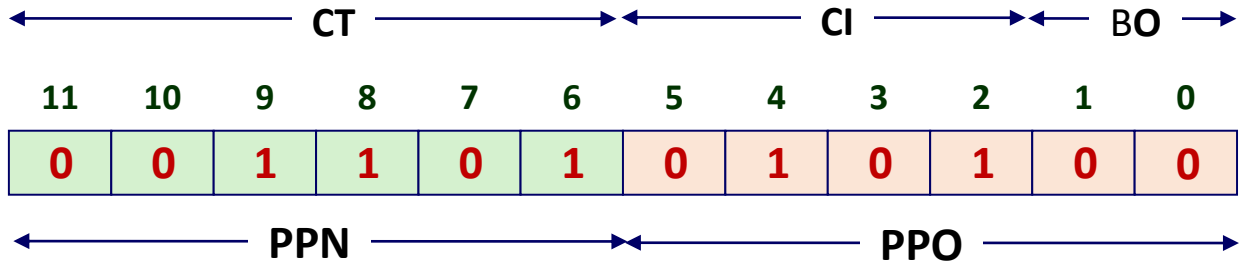
Virtual Address: 0x03D4

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

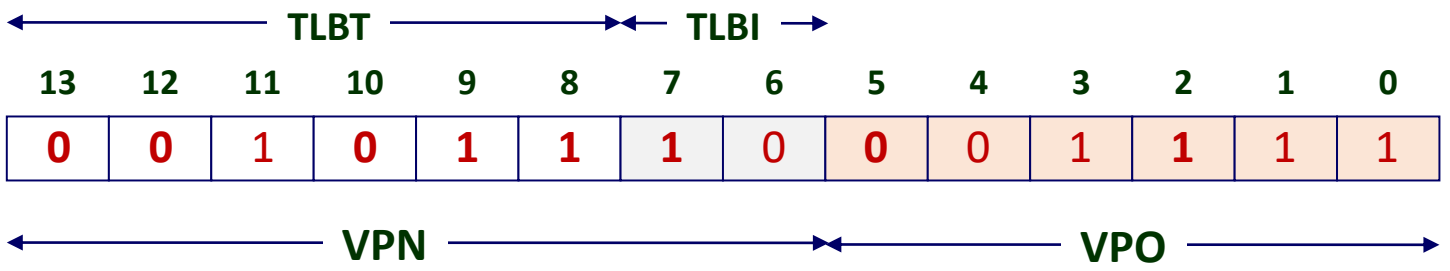


BO **0** CI **0x5** CT **0x0D** Hit? **Y** Byte: **0x36**

Address Translation Example #2

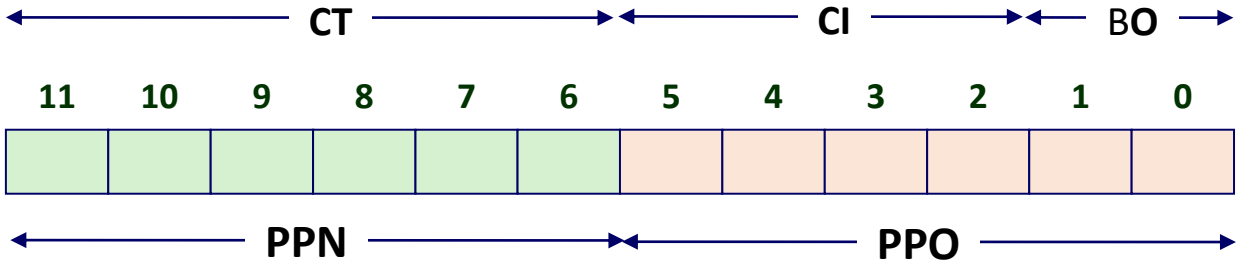
(using the Page Table, TLB, and L1 cache shown in the preceding slides)

Virtual Address: `0x0B8F`



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

Physical Address



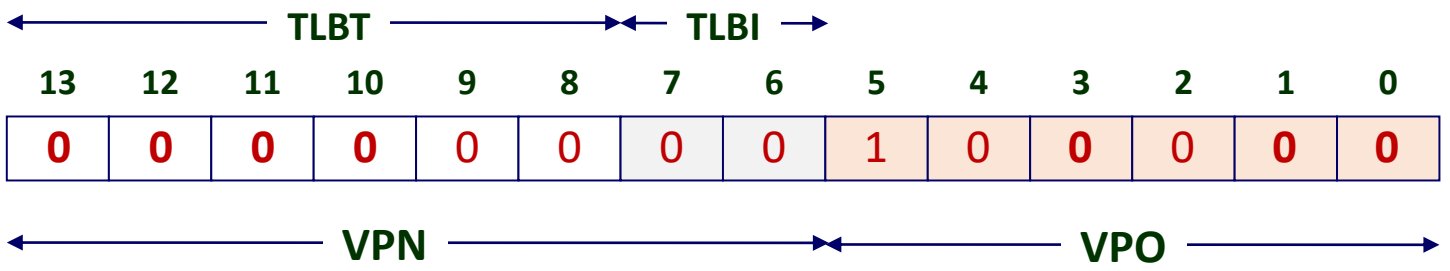
BO ___ CI___ CT ____ Hit? __ Byte: ____

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

Address Translation Example #3

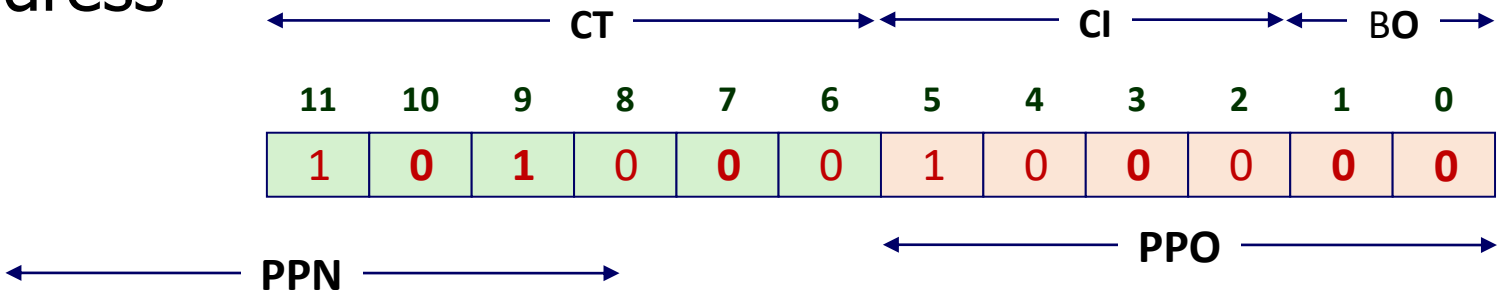
(using the Page Table, TLB, and L1 cache shown in the preceding slides)

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!!! 😱

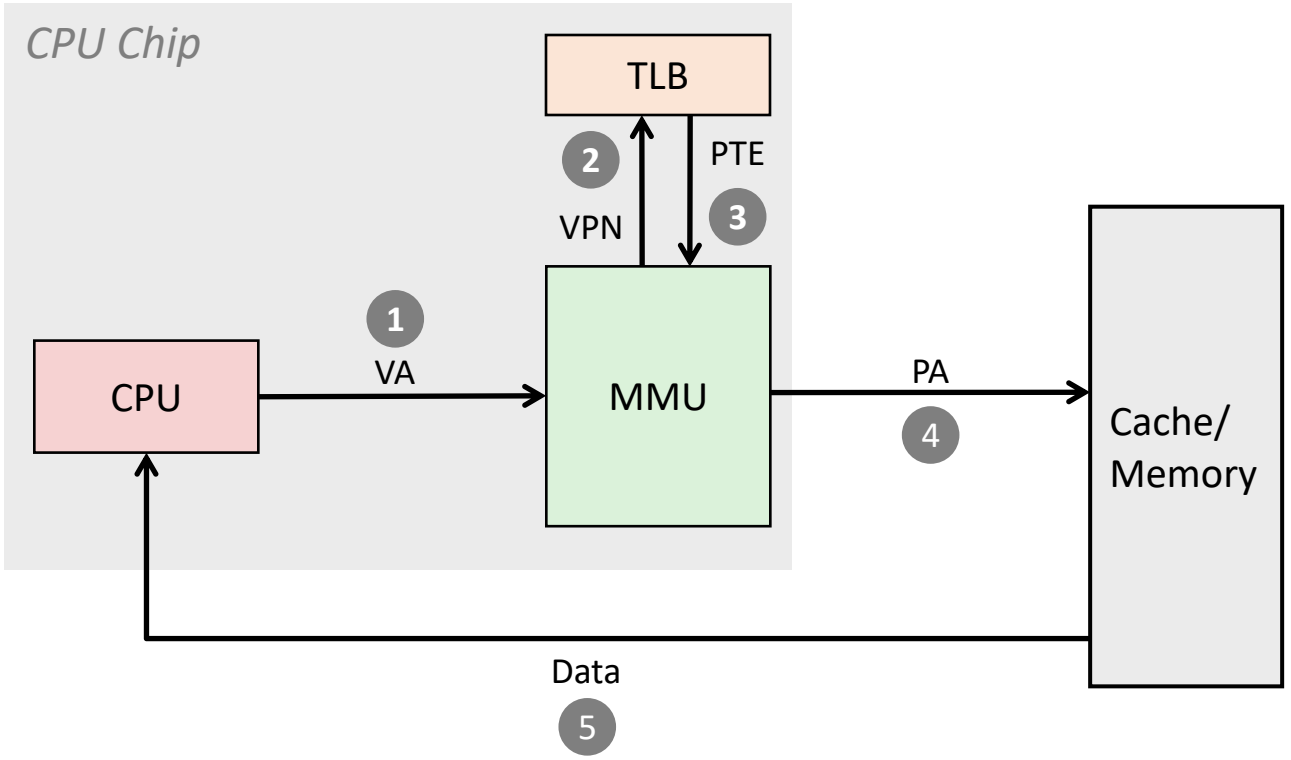
- Page table entries (PTEs) are cached in L1, L2, etc, like any other data in memory
  - PTEs may be evicted by other data references. Oops.
  - PTE access still requires average effective memory access delay

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!

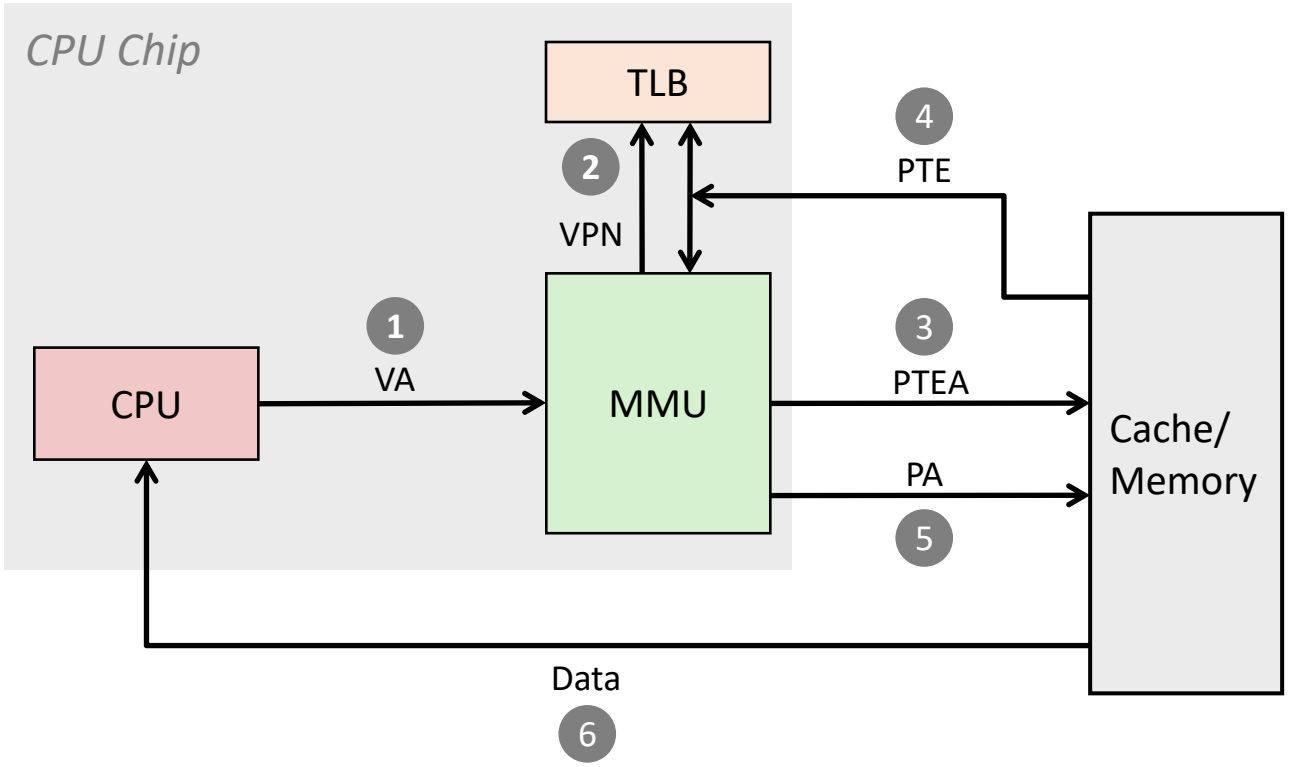
- Like cache memories, uses set indices, tags, and valid bits
  - VPN split into: TLB tag and TLB index (just like caches, because it is one!)
  - No need for a block offset equivalent (PTEs have a single value)

TLB Hit



A TLB hit eliminates a memory access

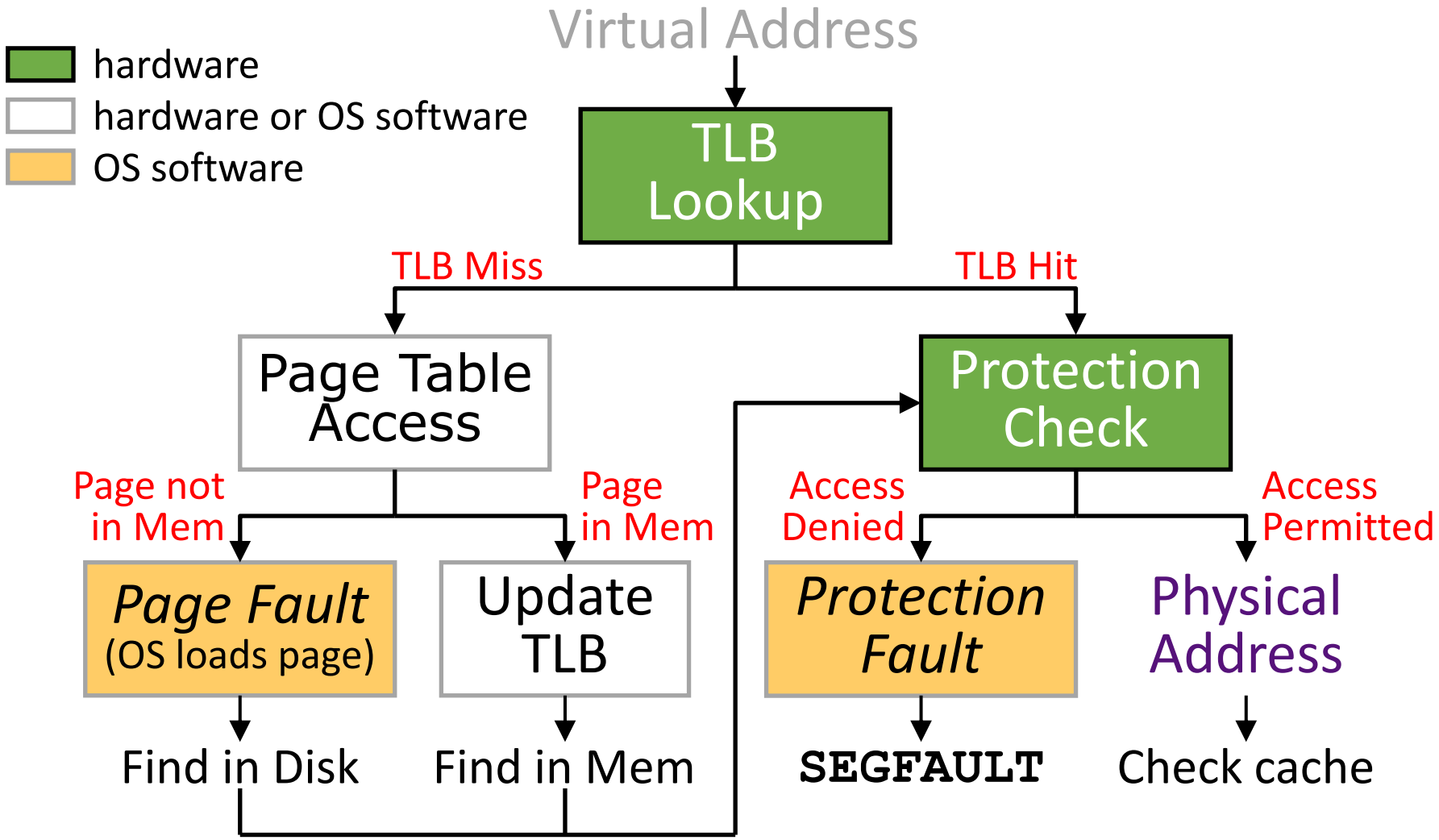
TLB Miss



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

Fortunately, TLB misses are rare. Why? Locality. It's always locality.

Address translation process

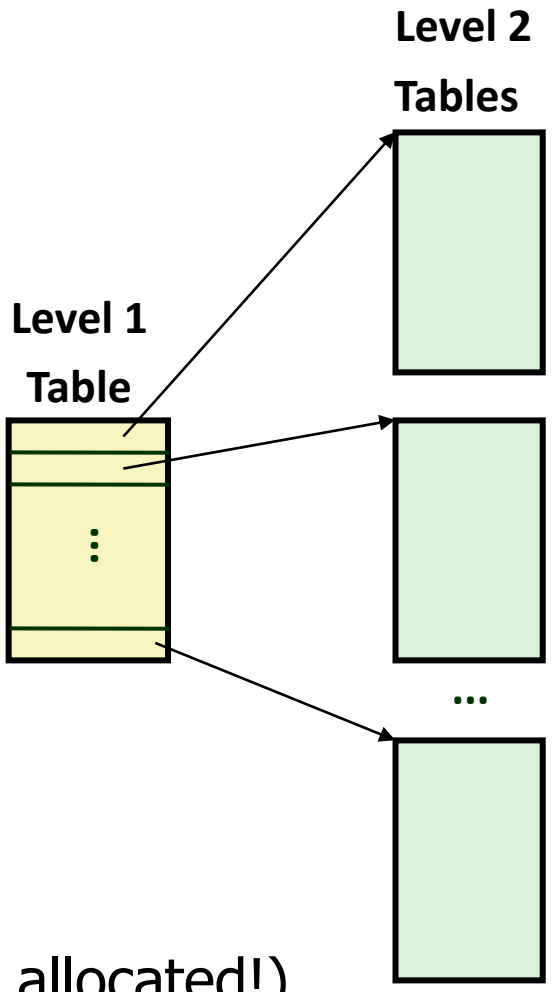


Outline

- Bonus: Multi-level Page Tables

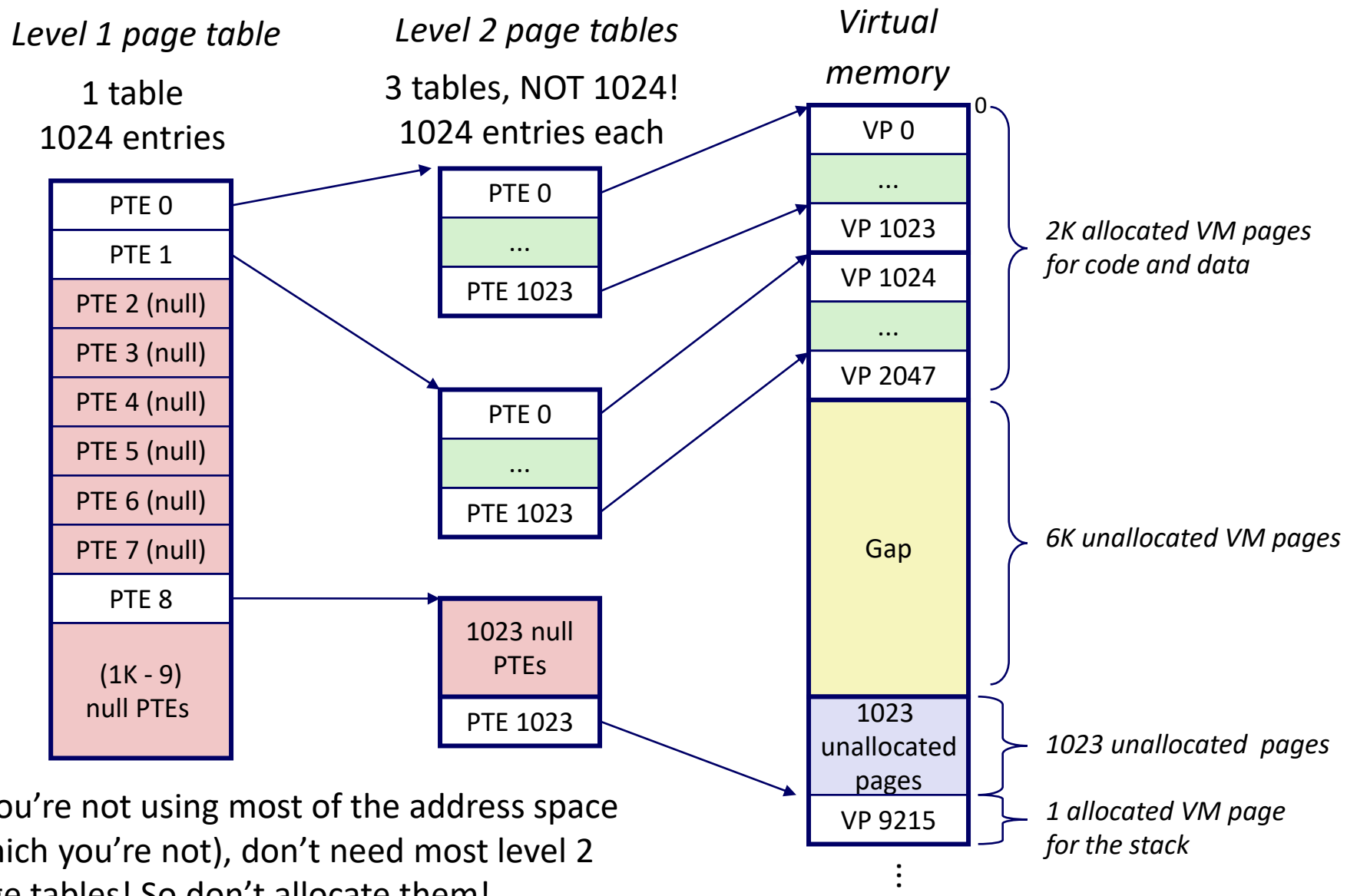
Multi-Level Page Tables

- 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!)



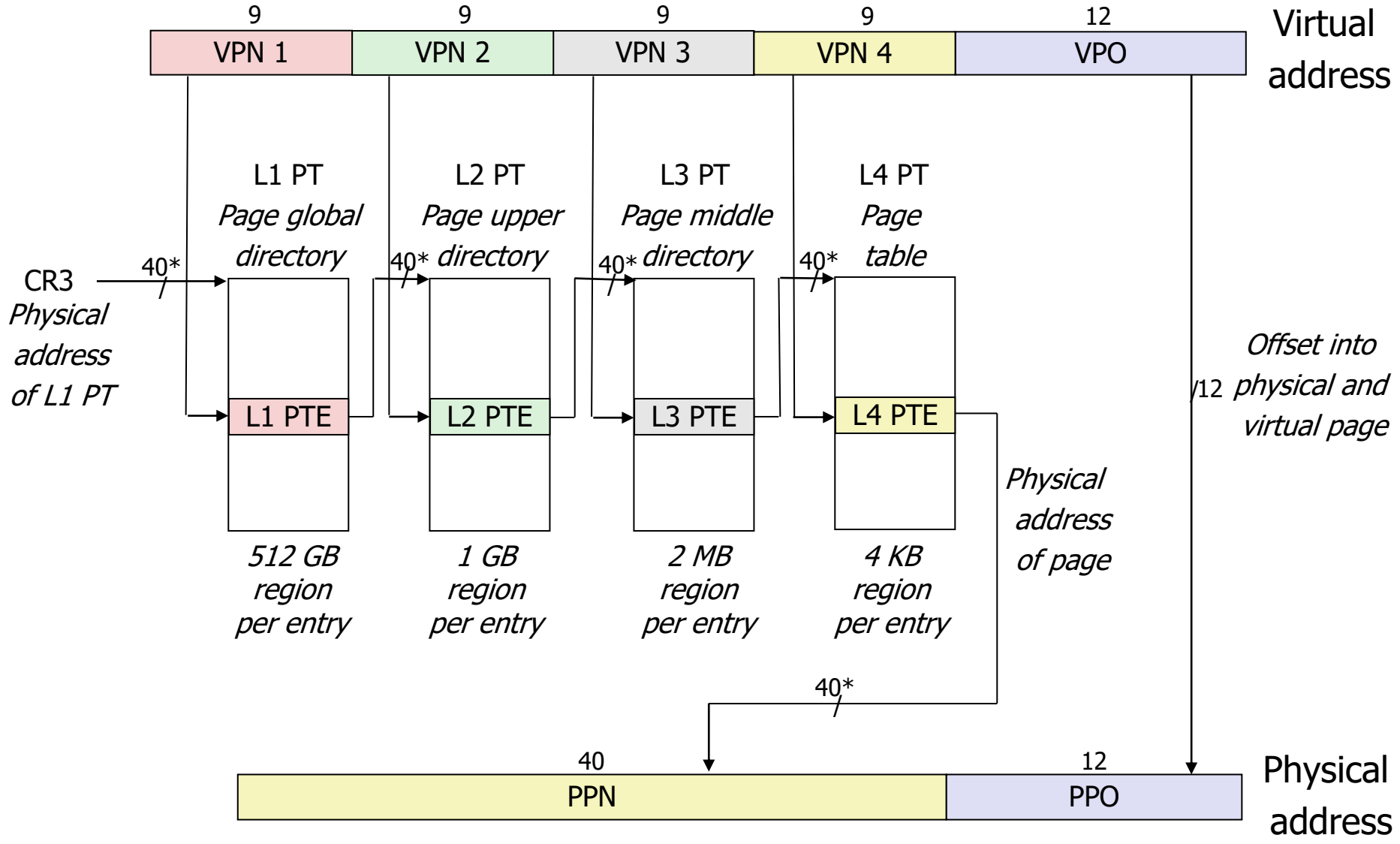
A Two-Level Page Table Hierarchy

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



If you're not using most of the address space (which you're not), don't need most level 2 page tables! So don't allocate them!

Multi-level page table: Core i7



*aligned to a 4K-boundary

End-to-end Core i7 Data Address Translation