Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

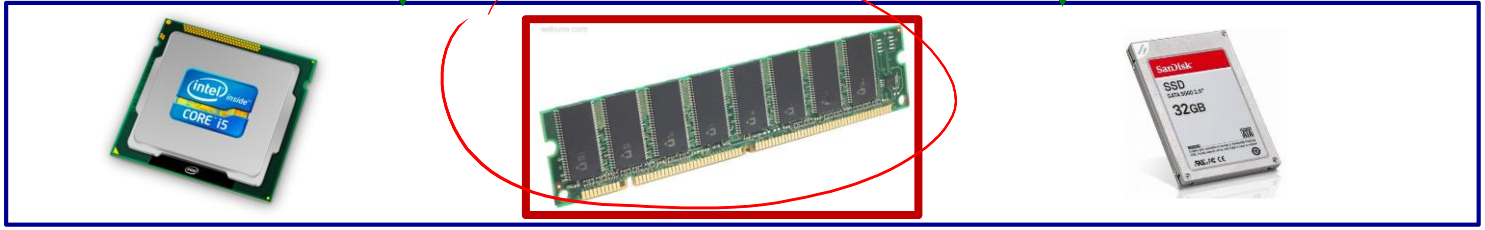
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Computer system:



Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Virtual Memory (VM*)

- **Overview and motivation**
  - *Fair warning*: it's pretty complex, but crucial for understanding how processes work and for debugging performance.
- **VM as tool for caching**
- **Address translation**
- **VM as tool for memory management**
- **VM as tool for memory protection**

**Not to be confused with "Virtual Machine" which is a whole other thing.*

Again: Processes

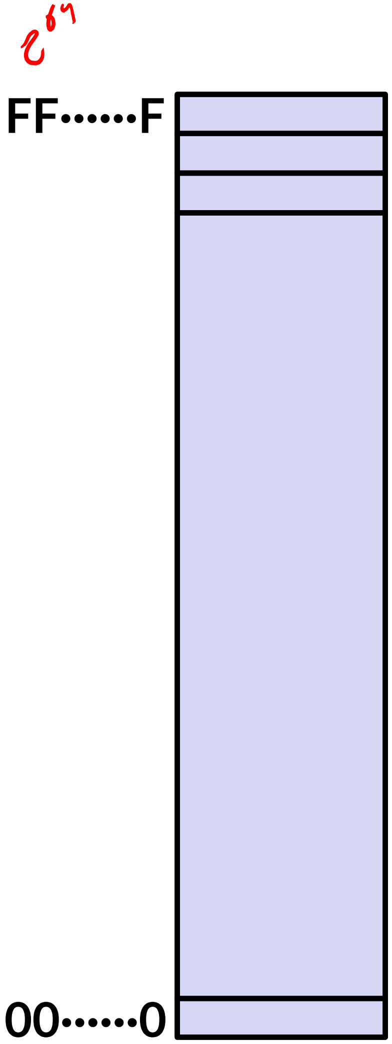
- **Definition: A *process* is an instance of a running program**
  - One of the most important ideas in computer science
  - Not the same as “program” or “processor”
  - Necessary for allowing programs to be developed *independently of each other* (another form of *encapsulation*)

- **Process provides each program with two key abstractions:**
  - Logical control flow
    - Each process seems to have exclusive use of the CPU
  - Private virtual address space
    - Each process seems to have exclusive use of memory (all 2^{64} bytes of it!)

- **How are these illusions maintained?**
  - Process executions interleaved (multi-tasking) – done…
  - Address spaces managed by virtual memory system – **now!**

Memory as we know it so far… is *virtual!*

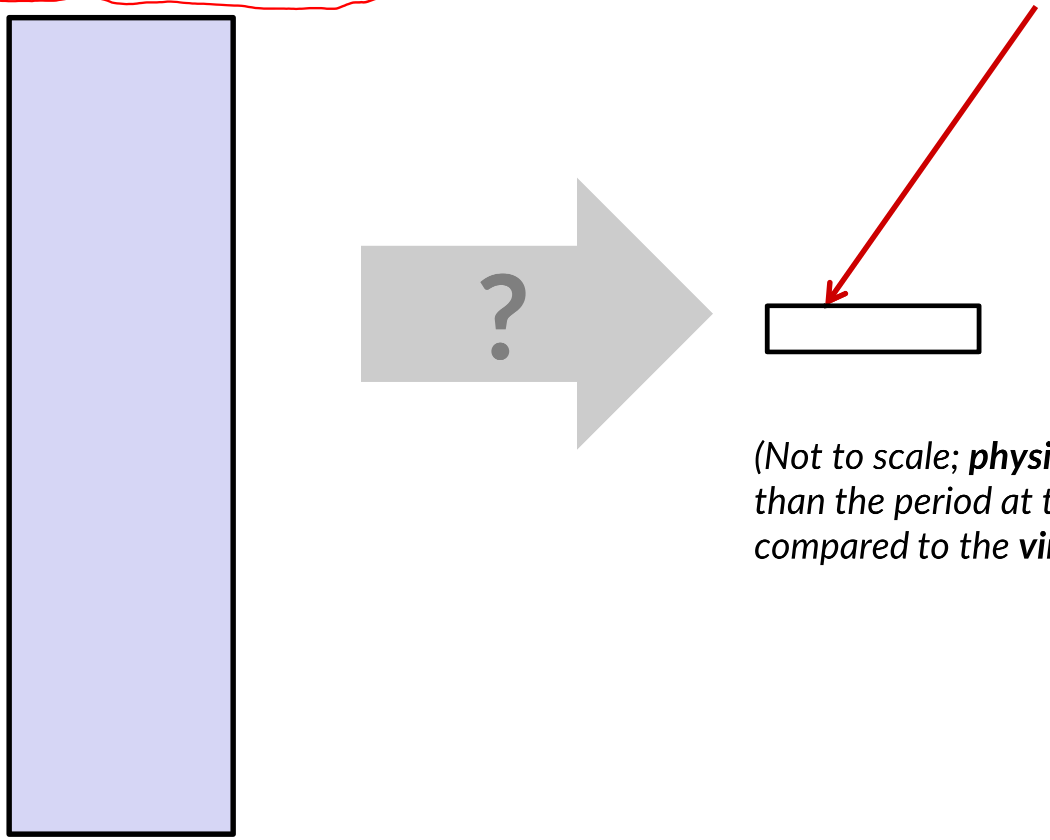
- **Programs refer to virtual memory addresses**
  - `movq (%rdi),%rax`
  - Conceptually memory is just a very large array of bytes
  - Each byte has its own address
  - System provides private address space to each process
- **Allocation: Compiler and run-time system**
  - Where different program objects should be stored
  - All allocation within single virtual address space
- **But…**
  - We *probably* don't have 2w bytes of physical memory (definitely not if w = 64!)
  - We *certainly* don't have 2w bytes of physical memory ***for every process***.
  - Processes should not interfere with one another
    - Except in certain cases where they want to share code or data



Problem 1: How Does Everything Fit?

64-bit virtual addresses can address several exabytes (18,446,744,073,709,551,616 bytes)

Physical main memory offers a few gigabytes (e.g. 8,589,934,592 bytes)



*(Not to scale; **physical** memory would be smaller than the period at the end of this sentence compared to the **virtual** address space.)*

1 virtual address space per process, with many processes…

Problem 2: Memory Management

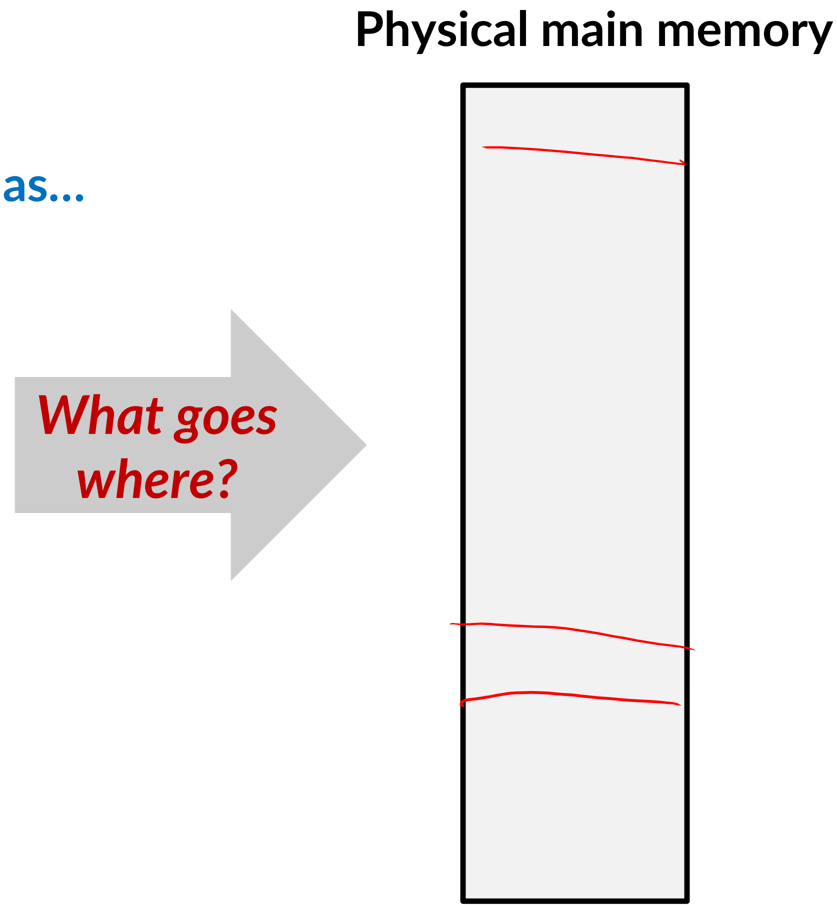
We have multiple processes:

Process 1
Process 2
Process 3
…
Process n

X

Each process has…

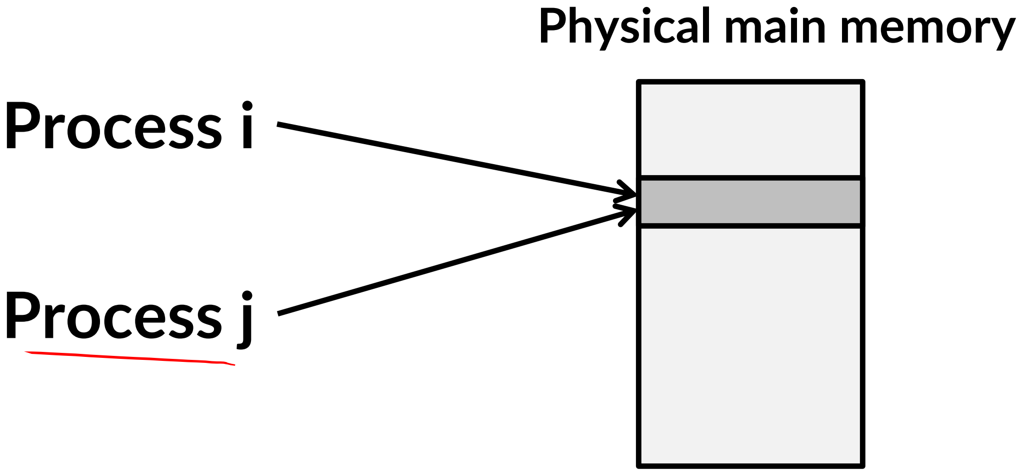
stack
heap
`.text`
`.data`
…



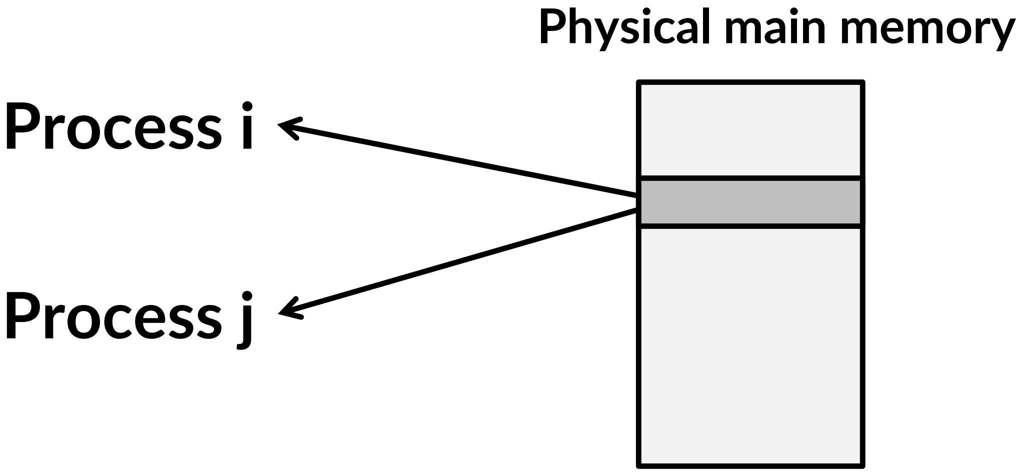
What goes where?

Physical main memory

Problem 3: How To Protect



Problem 4: How To Share?



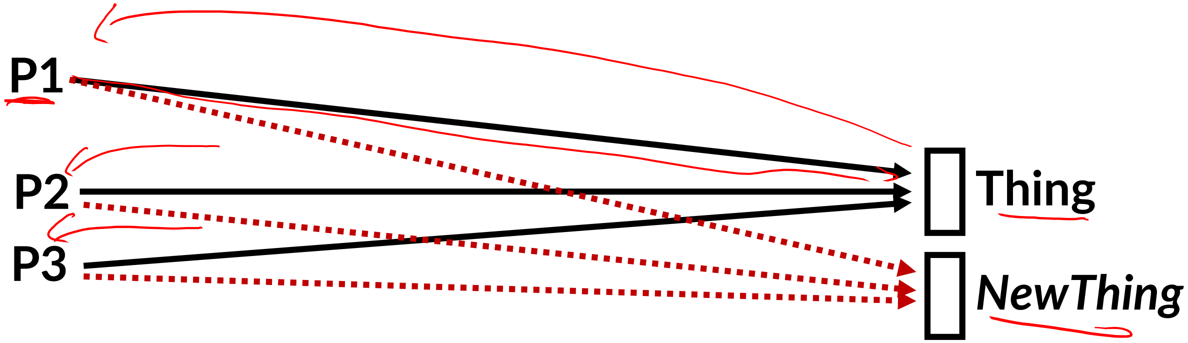
How can we solve these problems?

- Fitting a huge address space into a tiny physical memory
- Managing the address spaces of multiple processes
- Protecting processes from stepping on each other's memory
- Allowing processes to share common parts of memory

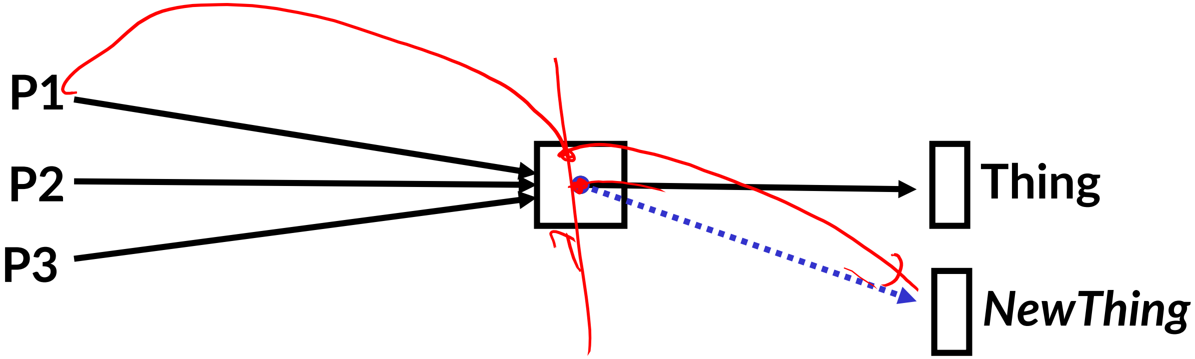
Indirection

- "Any problem in computer science can be solved by adding another level of indirection." *–David Wheeler, inventor of the subroutine (a.k.a. procedure)*

- **Without Indirection**



- **With Indirection**

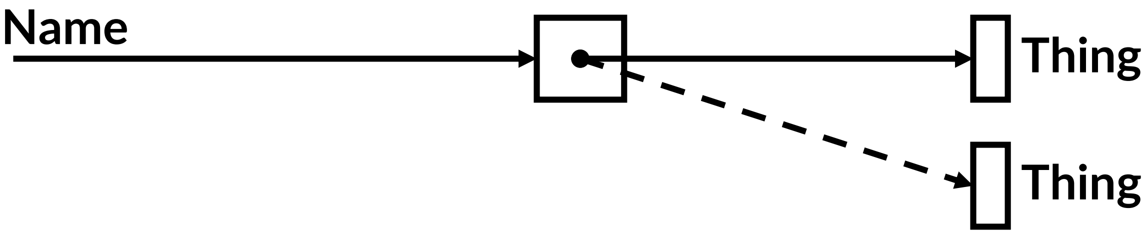


What if I want to move Thing?

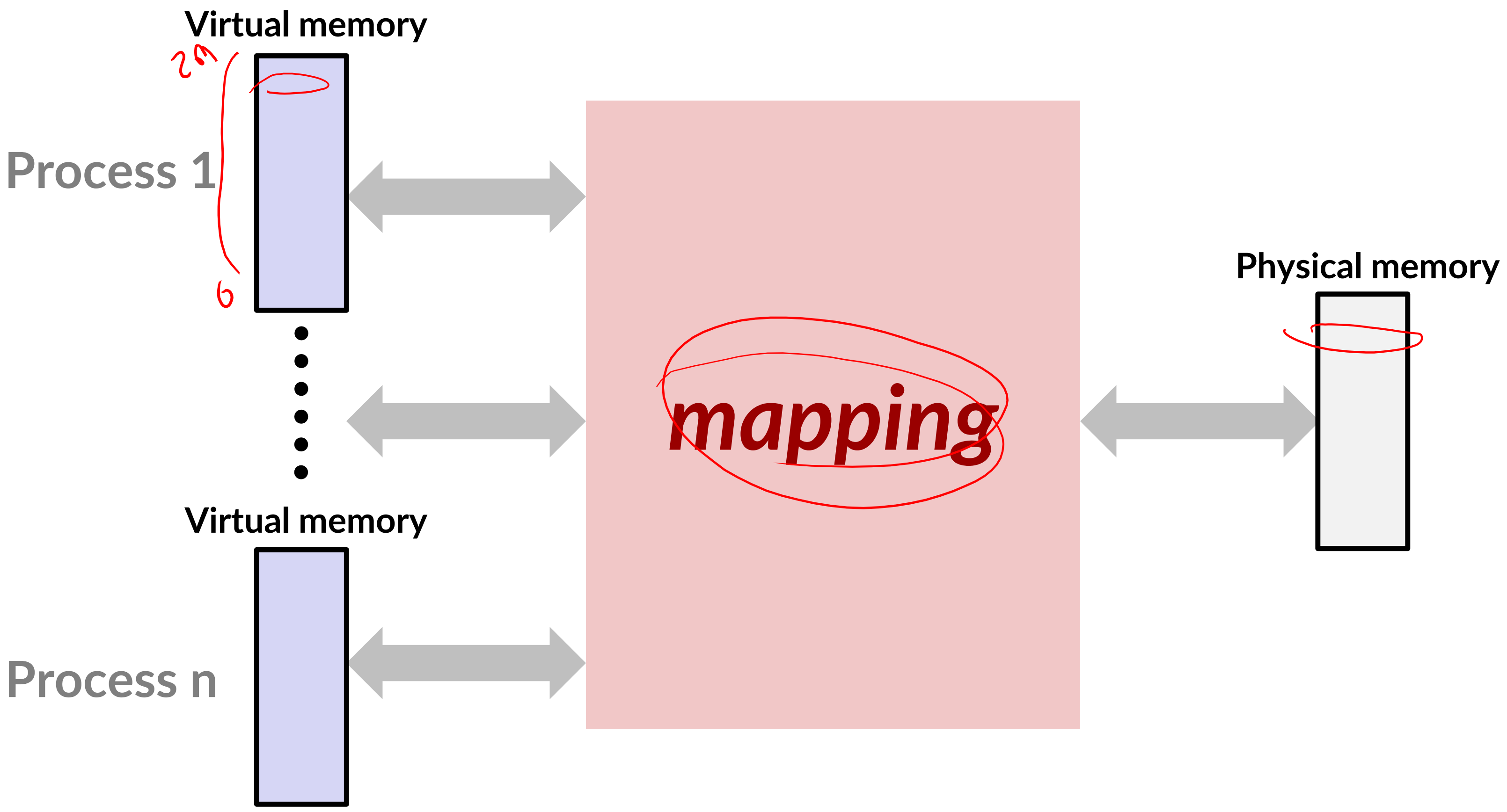
Indirection

- ***Indirection*:** the ability to reference something using a name, reference, or container instead the value itself. A flexible mapping between a name and a thing allows changing the thing without notifying holders of the name.
  - Adds some work ("*overhead*"; now have to look up 2 things instead of 1)
  - But don't have to track everyone that uses the name/address

- **Examples of indirection:**
  - **911:** routed to local office
  - **Call centers:** route calls to available operators, etc.
  - **Phone system:** cell phone number portability
  - **Snail mail:** mail forwarding
  - **Domain Name Service (DNS):** translation from name to IP address
  - **Dynamic Host Configuration Protocol (DHCP):** local network address assignment



Indirection in Virtual Memory



- Each process gets its own private virtual address space
- Solves the previous problems

Address Spaces

2^{64}

- **Virtual address space:** Set of N = 2^n virtual addresses
  {0, 1, 2, 3, …, N-1}

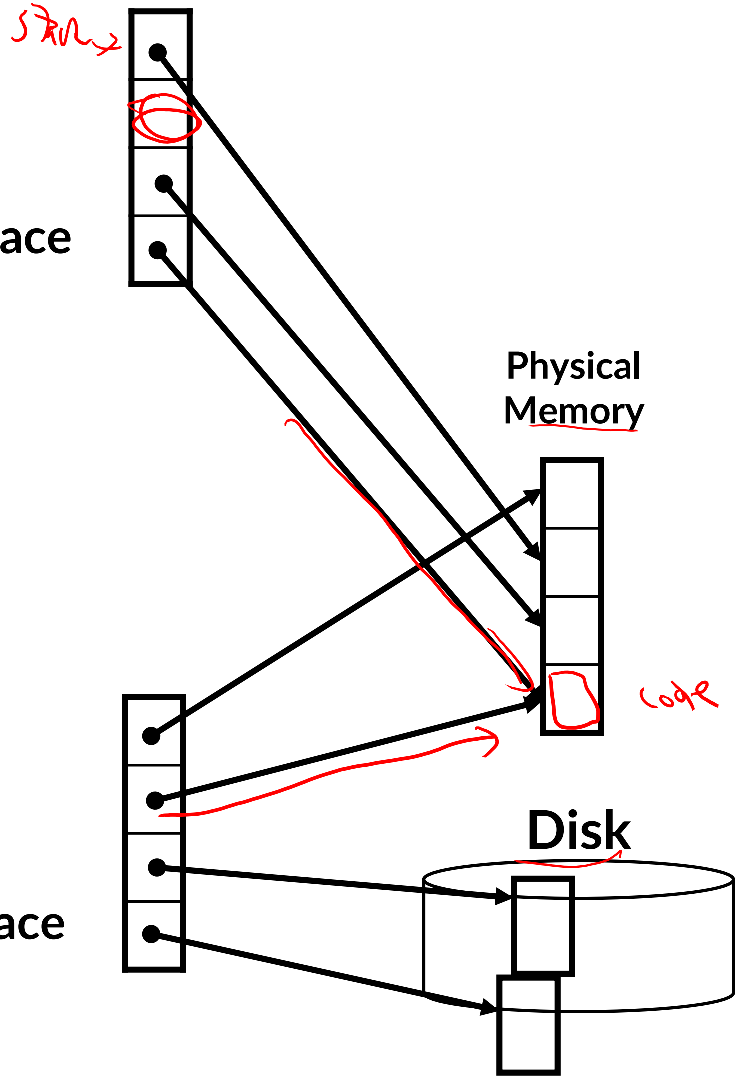
- **Physical address space:** Set of M = 2^m physical addresses (n >= m)
  {0, 1, 2, 3, …, M-1}

38 48

1 GB 2^8 256

- **Every byte in main memory has:**
  - one physical address
  - zero, one, *or more* virtual addresses

Mapping



P1's Virtual Address Space

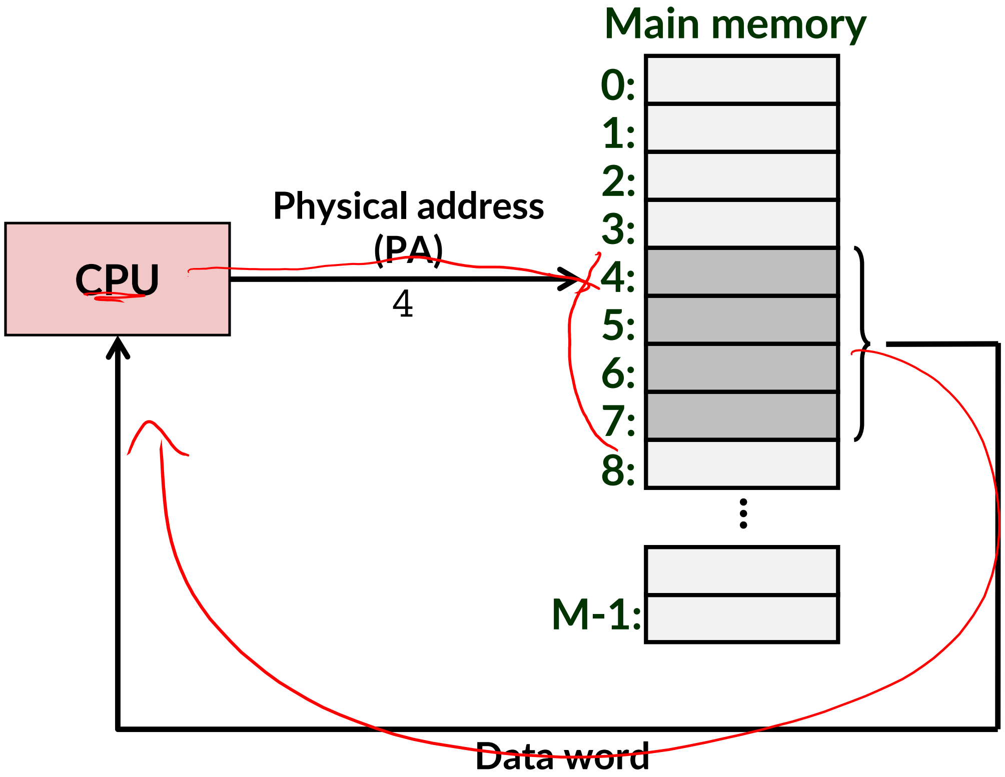
Physical Memory

Disk

P2's Virtual Address Space

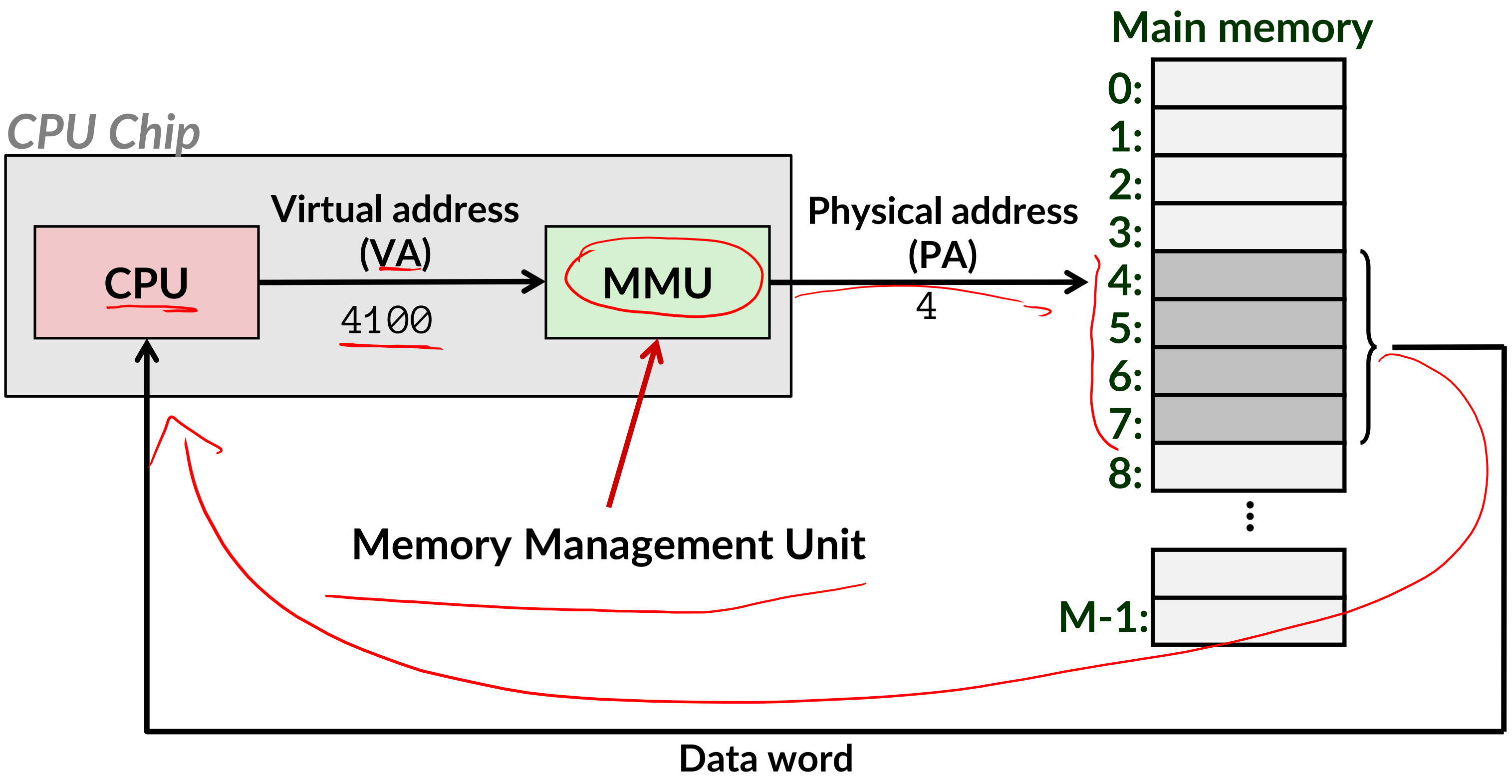
A virtual address can be mapped to either physical memory or disk

A System Using Physical Addressing



- Used in "simple" systems with (usually) just one process:
  - embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



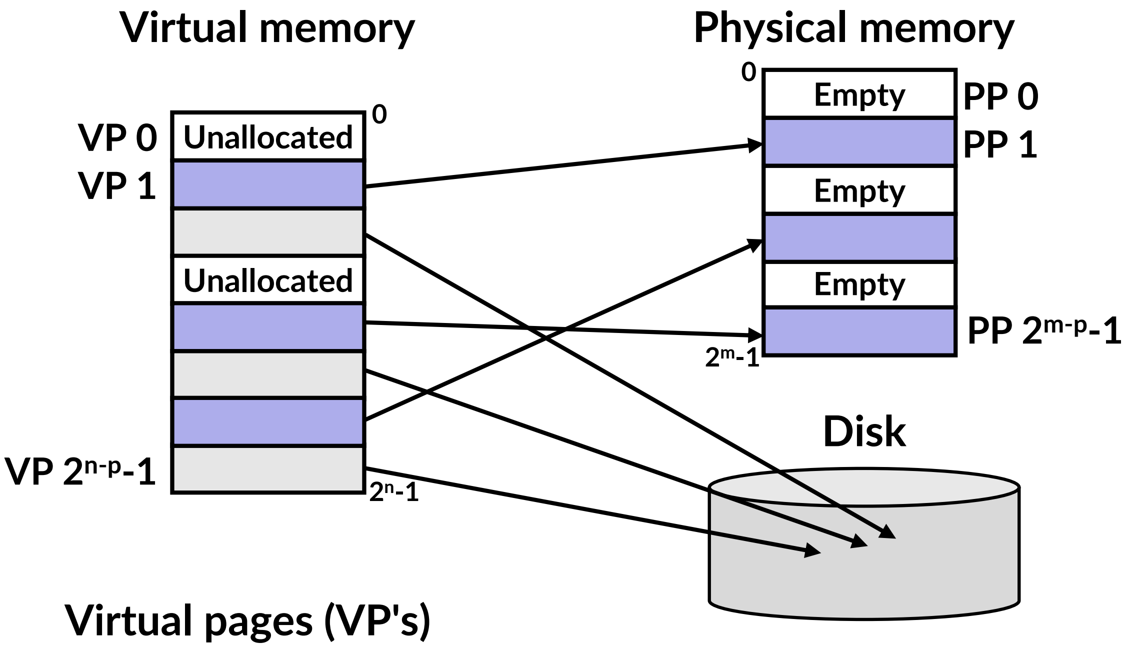
- **Physical addresses are *completely invisible to programs***
  - Used in all modern desktops, laptops, servers, smartphones…
  - One of the great ideas in computer science

Why Virtual Memory (VM)?

- **Efficient use of limited main memory (RAM)**
  - Use RAM as a cache for the parts of a virtual address space
    - some non-cached parts stored on disk
    - some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - transfer data back and forth as needed
- **Simplifies memory management for programmers**
  - Each process gets the same full, private linear address space
- **Isolates address spaces**
  - One process can't interfere with another's memory
    - because they operate in *different address spaces*
  - User process cannot access privileged information
    - different sections of address spaces have different permissions

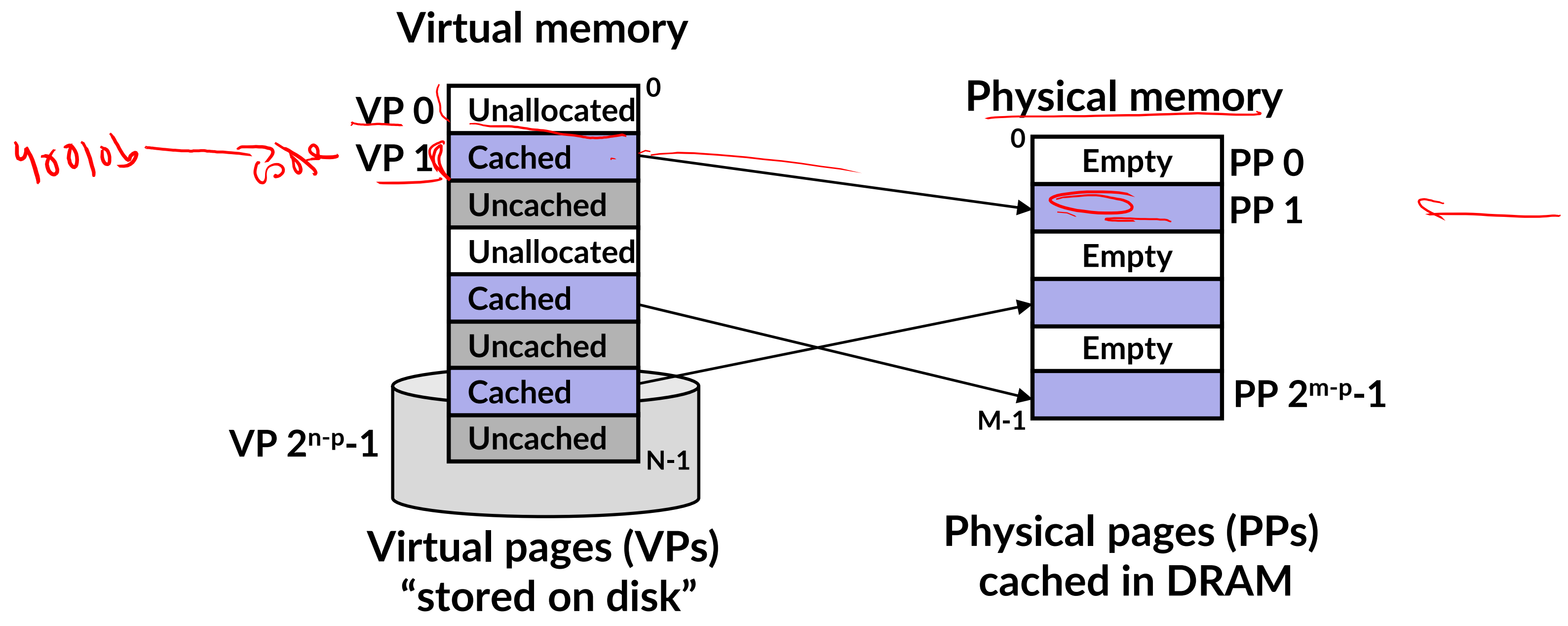
VM and the Memory Hierarchy

- Think of virtual memory as array of $N = 2^n$ contiguous bytes.
- *Pages* of virtual memory are usually stored in physical memory, but sometimes spill to disk.
  - Pages are another unit of aligned memory (size is $P = 2^p$ bytes)
  - Each virtual page can be stored in *any* physical page



or: Virtual Memory as DRAM Cache for Disk

- Think of virtual memory as an array of N = 2^n contiguous bytes stored *on a disk.*
- Then physical main memory is used as a *cache* for the virtual memory array
  - These "cache blocks" are called *pages* (size is P = 2^p bytes)

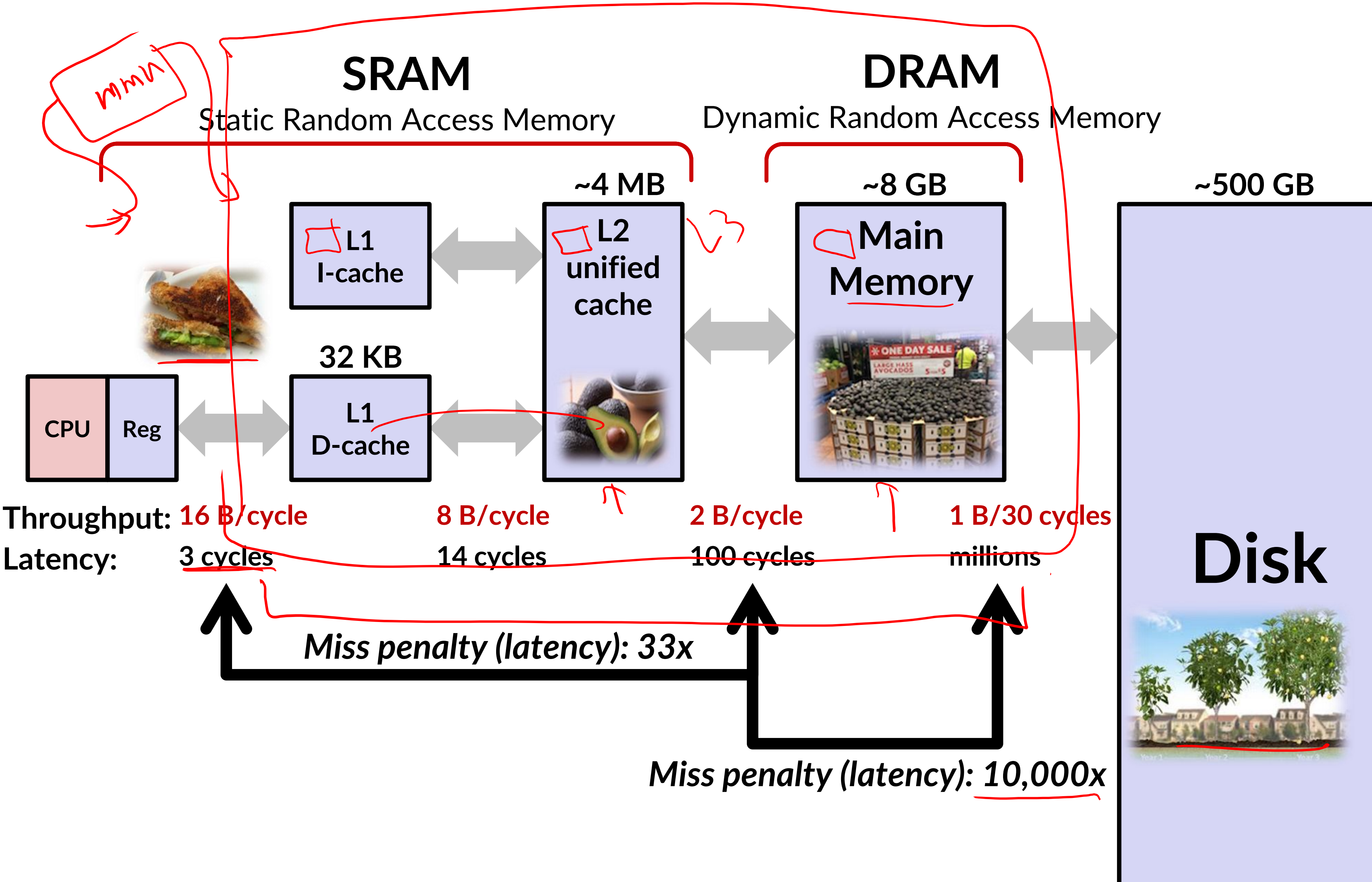


Virtual pages (VPs) "stored on disk"

Physical pages (PPs) cached in DRAM

Memory Hierarchy: Core 2 Duo

Not drawn to scale



SRAM

Static Random Access Memory

DRAM

Dynamic Random Access Memory

~4 MB

~8 GB

~500 GB

L1 I-cache

32 KB

L1 D-cache

L2 unified cache

Main Memory

CPU

Reg

Disk

Throughput: 16 B/cycle 8 B/cycle 2 B/cycle 1 B/30 cycles

Latency: 3 cycles 14 cycles 100 cycles millions

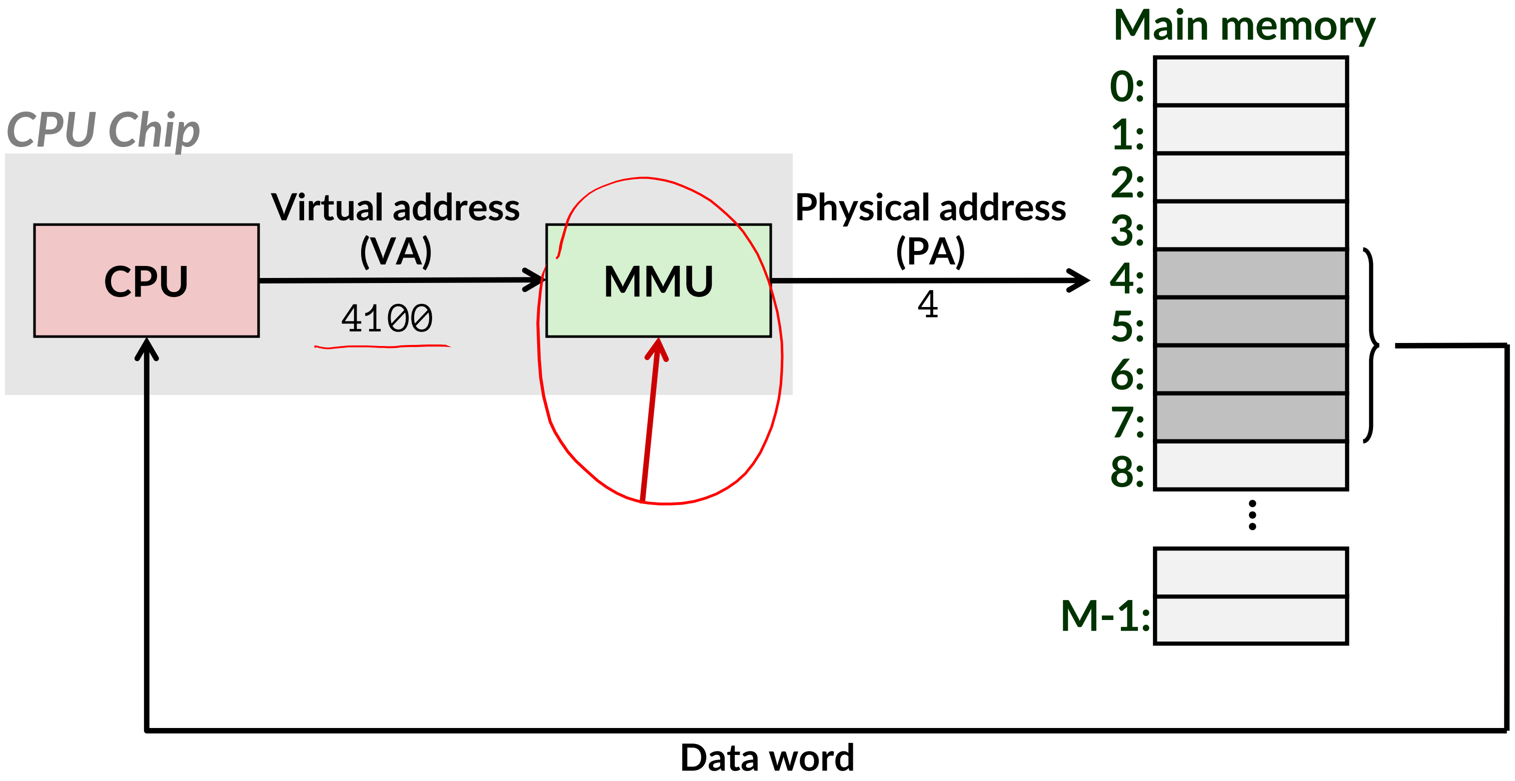
Miss penalty (latency): 33x

Miss penalty (latency): 10,000x

Virtual Memory Design Consequences

- **Large page size: typically 4-8 KB or 2-4 MB**
  - *Can* be up to 1 GB (for "Big Data" apps on big computers)
  - Compared with 64-byte cache blocks
- **Fully associative**
  - Any virtual page can be placed in any physical page
  - Requires a "large" mapping function – different from CPU caches
- **Highly sophisticated, expensive replacement algorithms in OS**
  - Too complicated and open-ended to be implemented in hardware
- ***Write-back* rather than *write-through***
  - *Really* don't want to write to disk every time we modify something in memory
  - Some things may never end up on disk (e.g. stack for short-lived process)

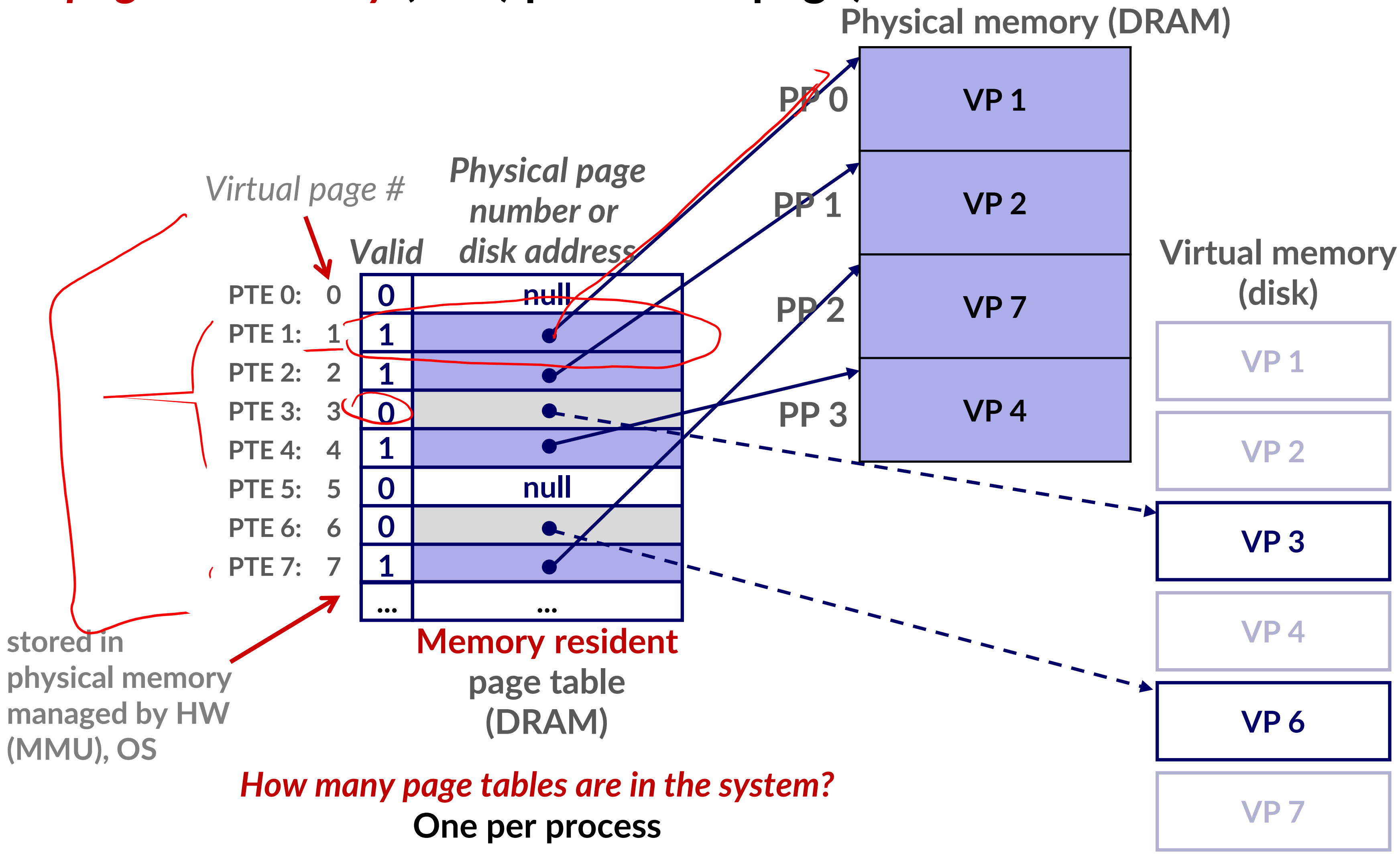
Address Translation



How do we perform the virtual → physical address translation?

Address Translation: Page Tables

- A *page table* is an array that maps virtual pages to physical pages (one *page table entry* (PTE) per virtual page)



	Virtual page #	Valid	Physical page number or disk address
PTE 0:	0	0	null
PTE 1:	1	1	●
PTE 2:	2	1	●
PTE 3:	3	0	●
PTE 4:	4	1	●
PTE 5:	5	0	null
PTE 6:	6	0	●
PTE 7:	7	1	●
		…	…

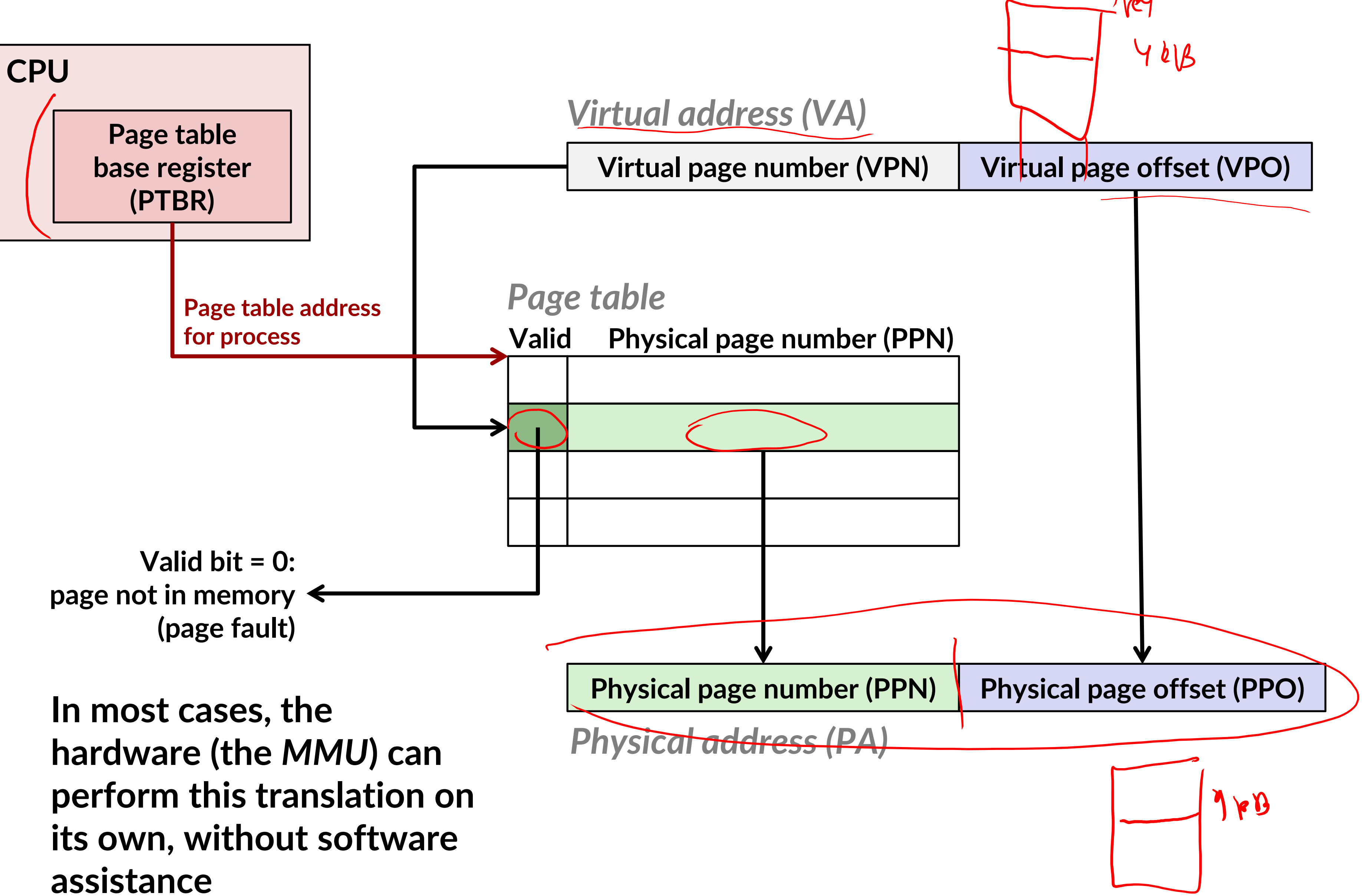
stored in physical memory managed by HW (MMU), OS

Memory resident page table (DRAM)

How many page tables are in the system?

One per process

Address Translation With a Page Table



CPU

Page table base register (PTBR)

Page table address for process

Virtual address (VA)

Virtual page number (VPN)

Virtual page offset (VPO)

Page table

Valid

Physical page number (PPN)

Valid bit = 0:
page not in memory
(page fault)

Physical page number (PPN)

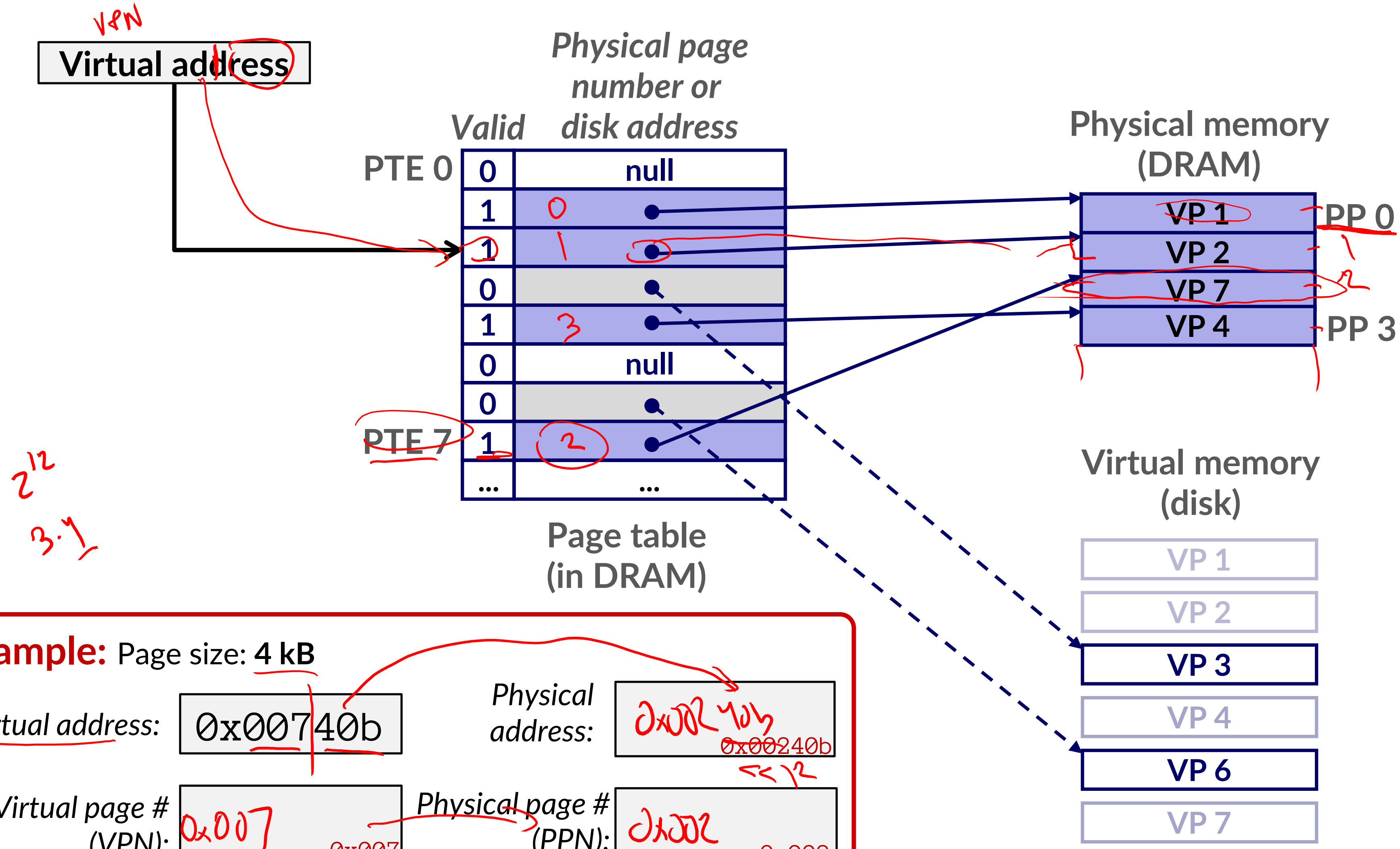
Physical page offset (PPO)

Physical address (PA)

In most cases, the hardware (the *MMU*) can perform this translation on its own, without software assistance

Page Hit

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



Example: Page size: **4 kB**

Virtual address: 0x00740b

Physical address: 0x002 40b 0x00240b

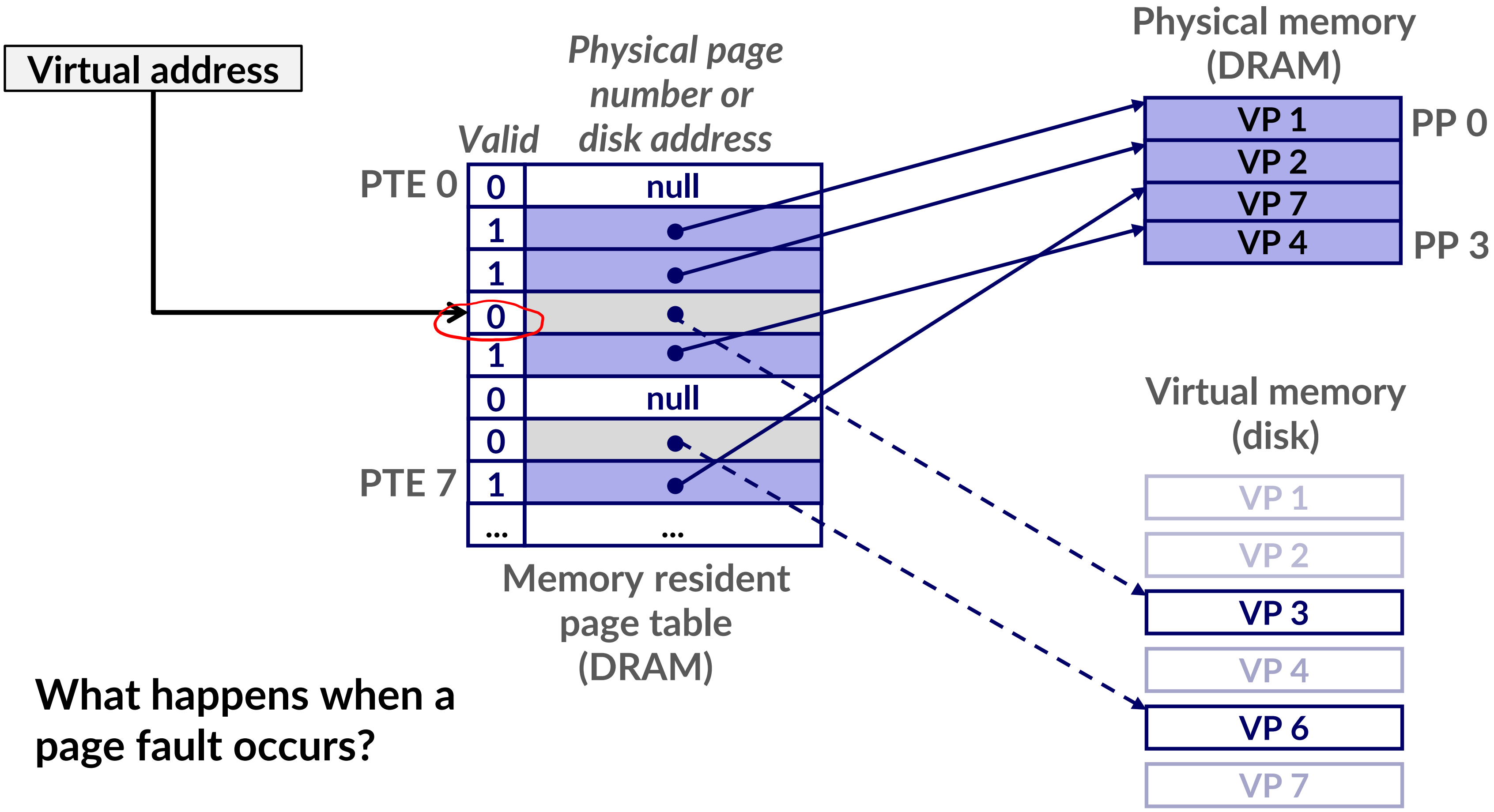
<<12

Virtual page # (VPN): 0x007 0x007

Physical page # (PPN): 0x002 0x002

Page Fault

- *Page fault:* reference to VM byte that is NOT in physical memory



What happens when a page fault occurs?

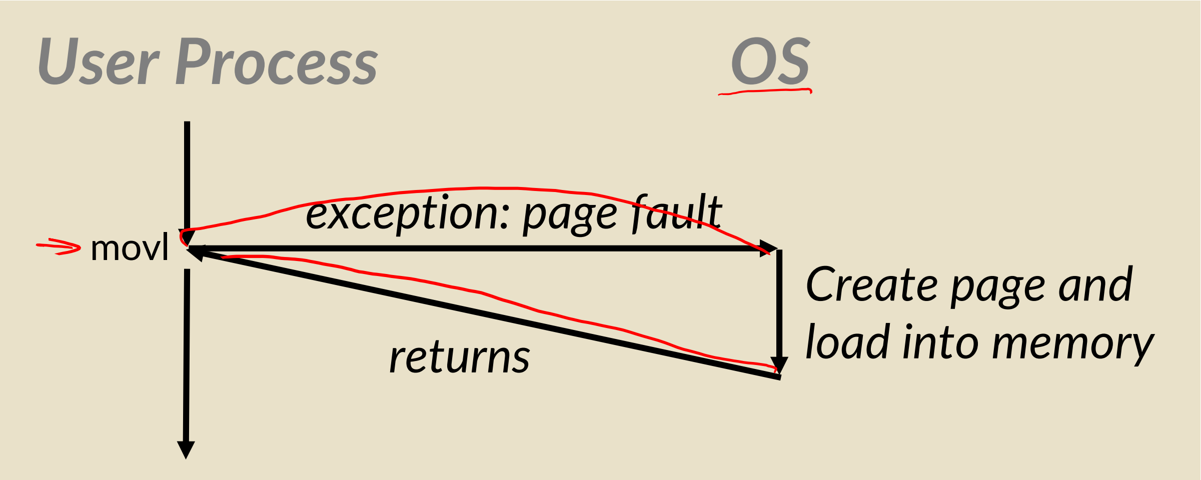
Fault Example: Page Fault

```
int a[1000];

int main()
{
    a[500] = 13;
}
```

- User writes to memory location
- That portion (page) of user's memory is currently on disk

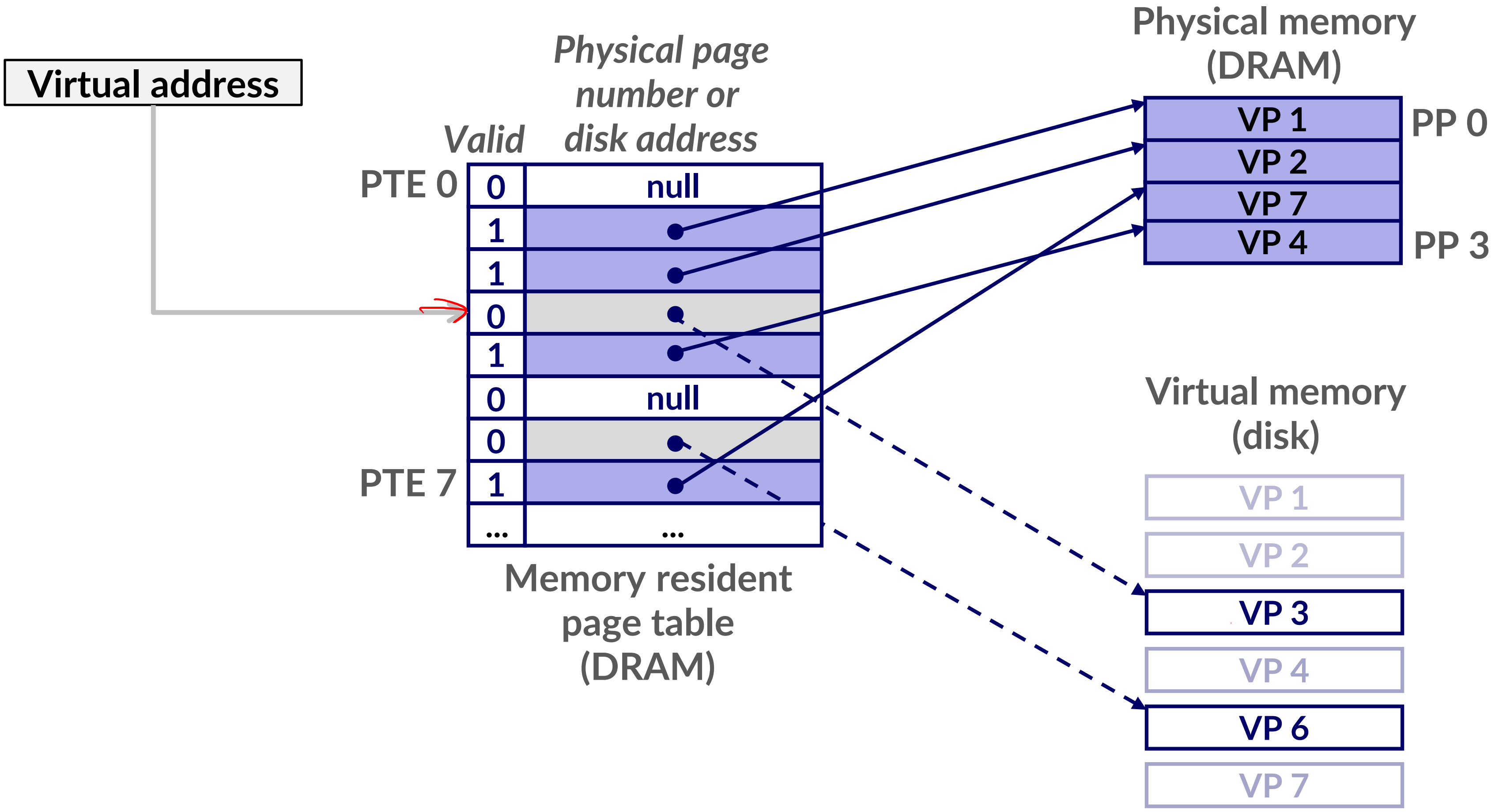
```
80483b7:        c7 05 10 9d 04 08 0d     movl    $0xd,0x8049d10
```



- Page fault handler must load page into physical memory
- Returns to faulting instruction: **mov** is executed *again*!
- Successful on second try

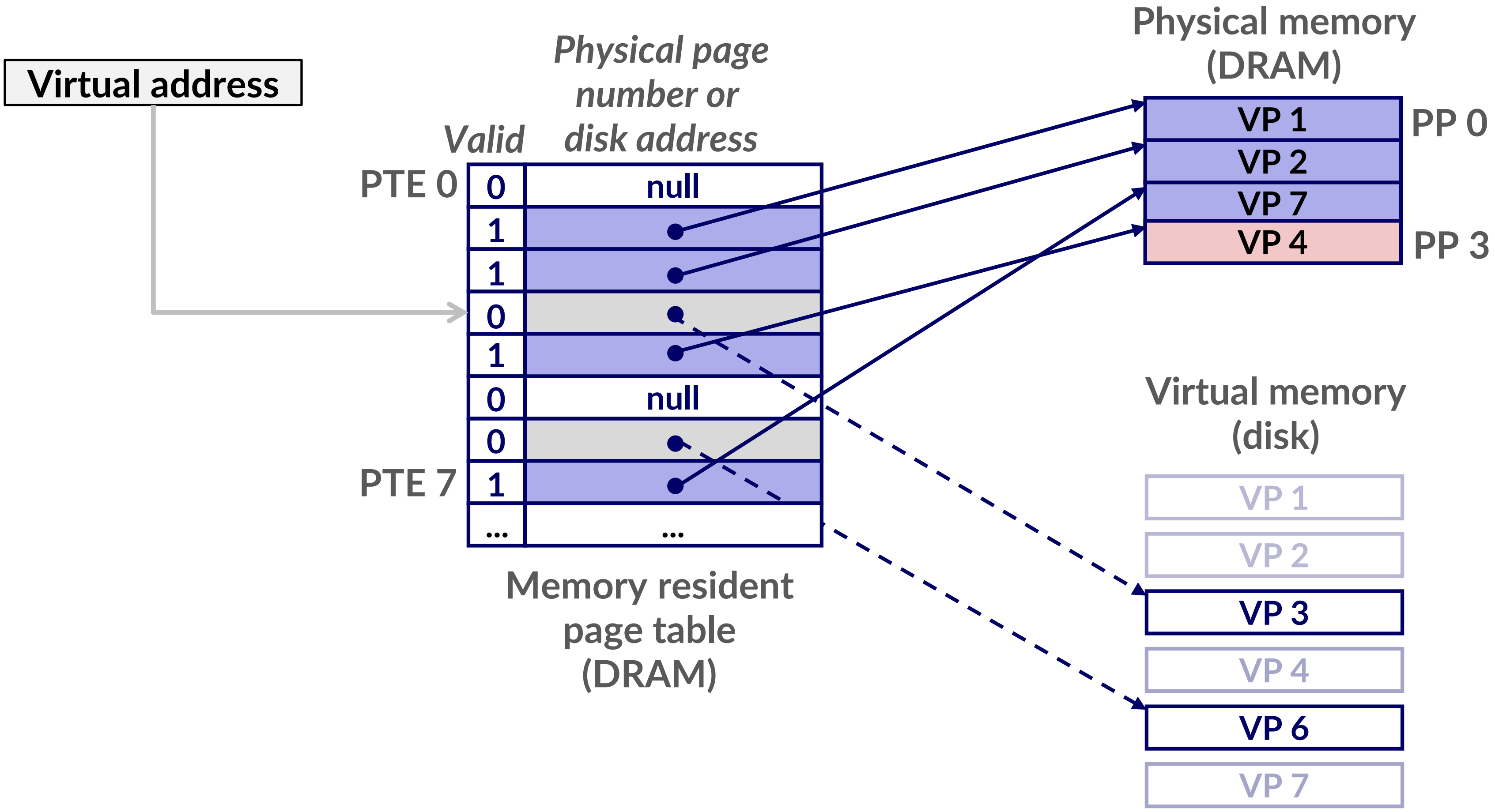
Handling Page Fault

- Page miss causes page fault (an exception)



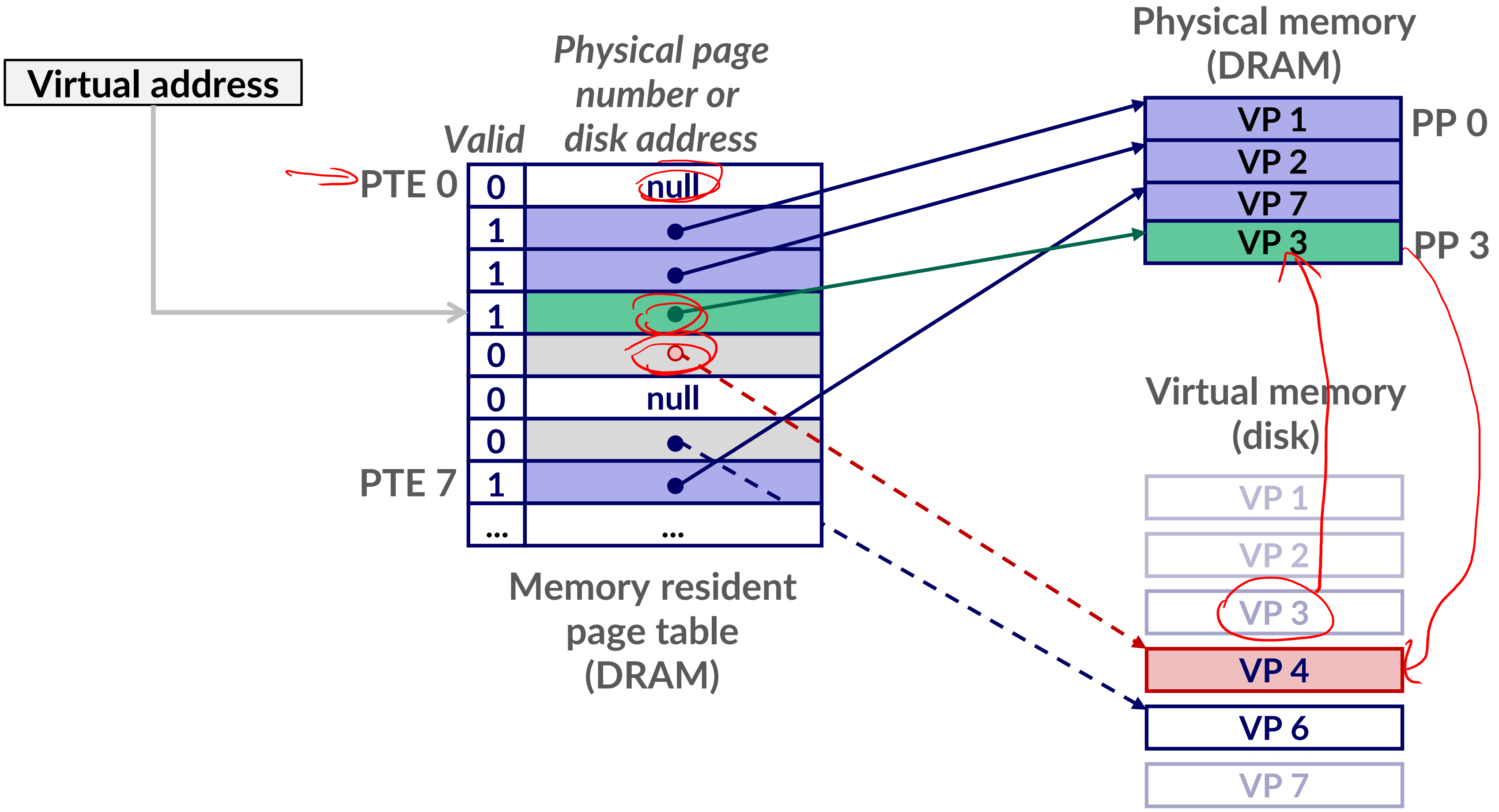
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)



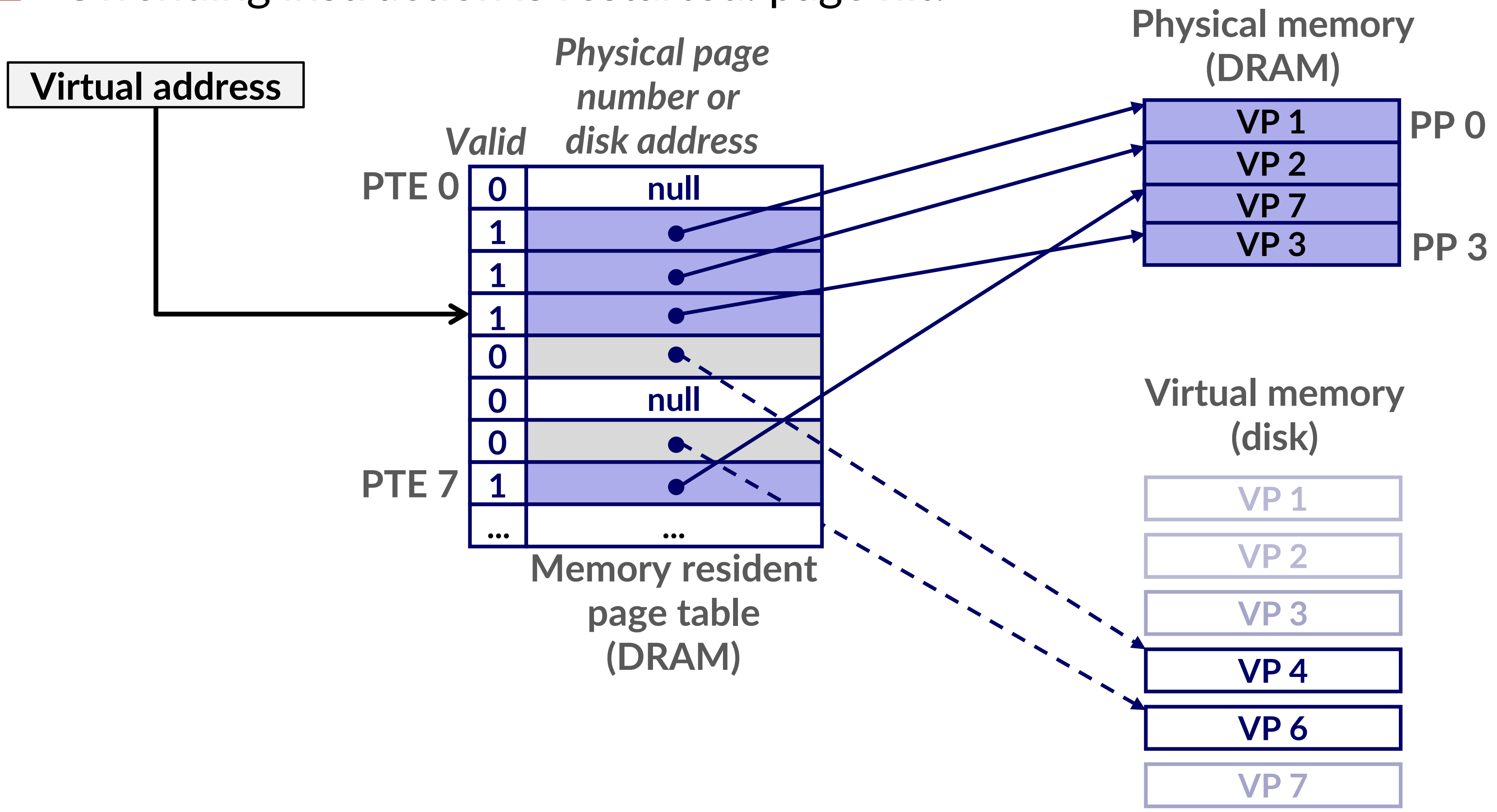
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)



Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



Why does it work?

Why does Virtual Memory work on RAM/disk?

- **Works well for avoiding disk accesses because of *locality*.**
  - Same reason that L1 / L2 / L3 caches work

- **The set of virtual pages that a program is "actively" accessing at any point in time is called its *working set***

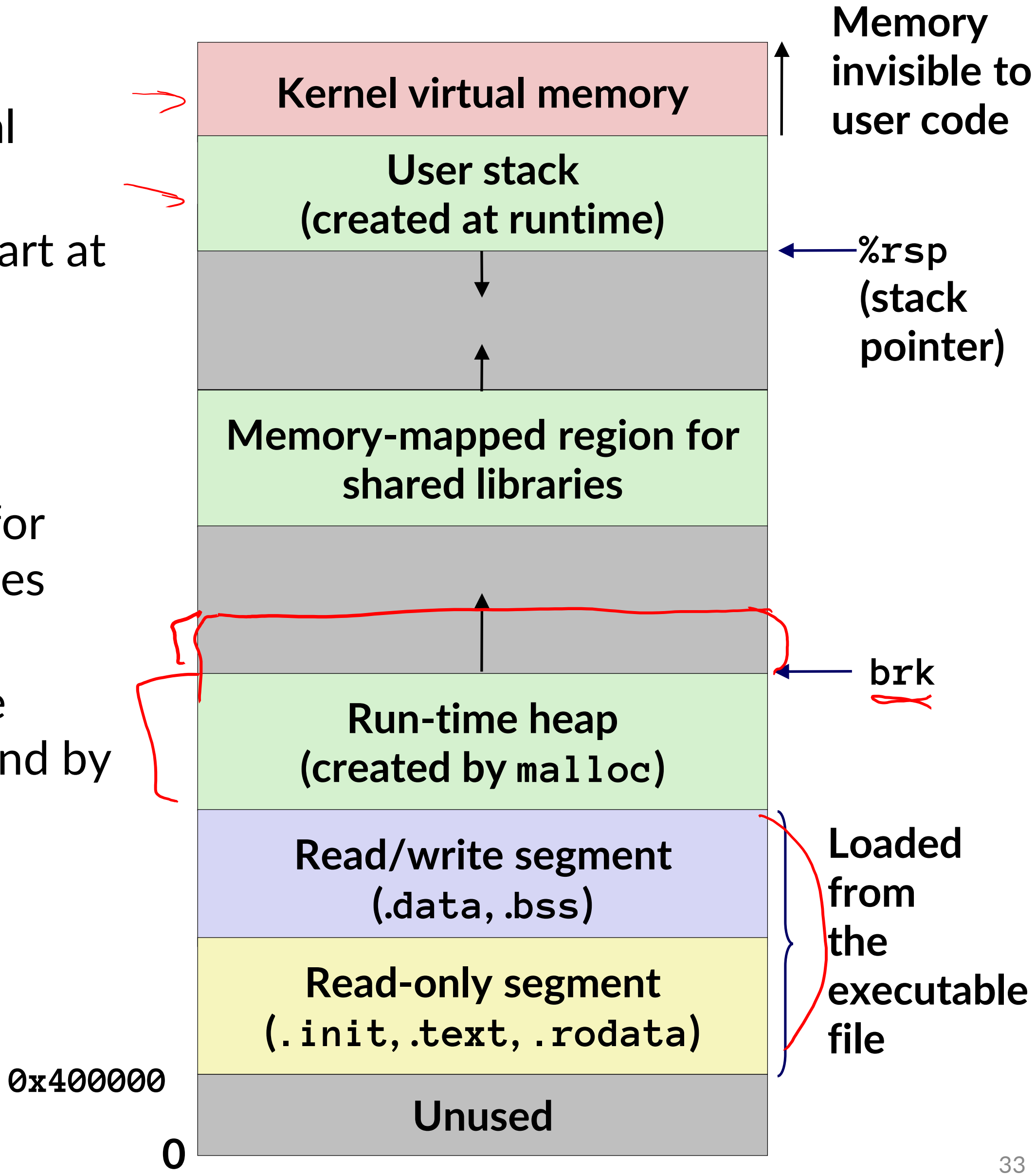
- **if (*working set size of one process* < *main memory size*):**
  - Good performance for one process (after compulsory misses)

- **But…**

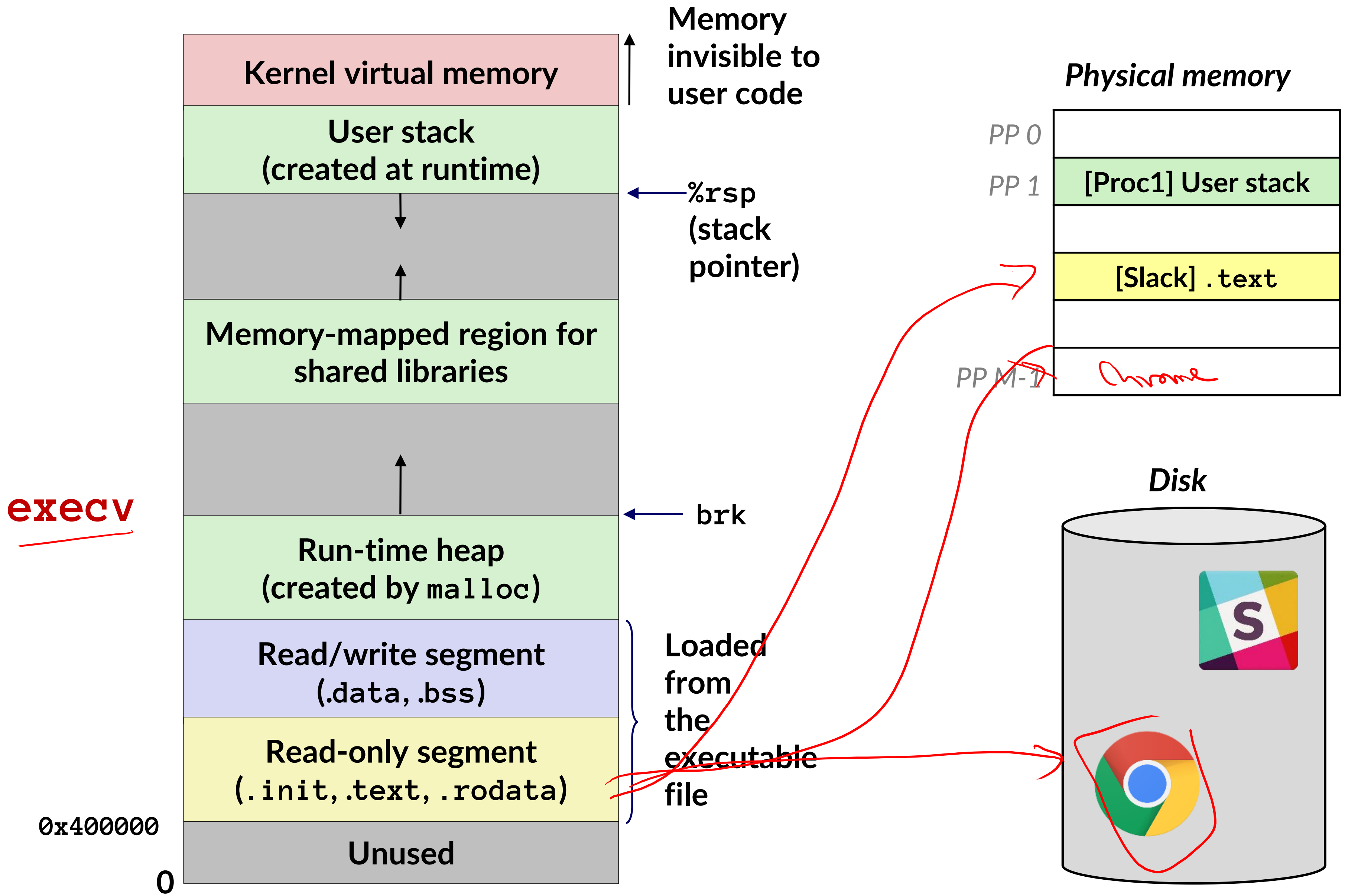
  if sum(*working set sizes of all processes*) > *main memory size*:
  - *Thrashing:* Performance meltdown where pages are swapped (copied) between memory and disk continuously. CPU always waiting or paging.
  - This is why your computer can feel faster when you add RAM.

Simplifying Linking and Loading

- **Linking**
  - Each program has similar virtual address space
  - Code, data, and heap always start at the same addresses.
- **Loading**
  - execve allocates virtual pages for .text and .data sections & creates PTEs marked as invalid
  - The .text and .data sections are copied, page by page, on demand by the virtual memory system

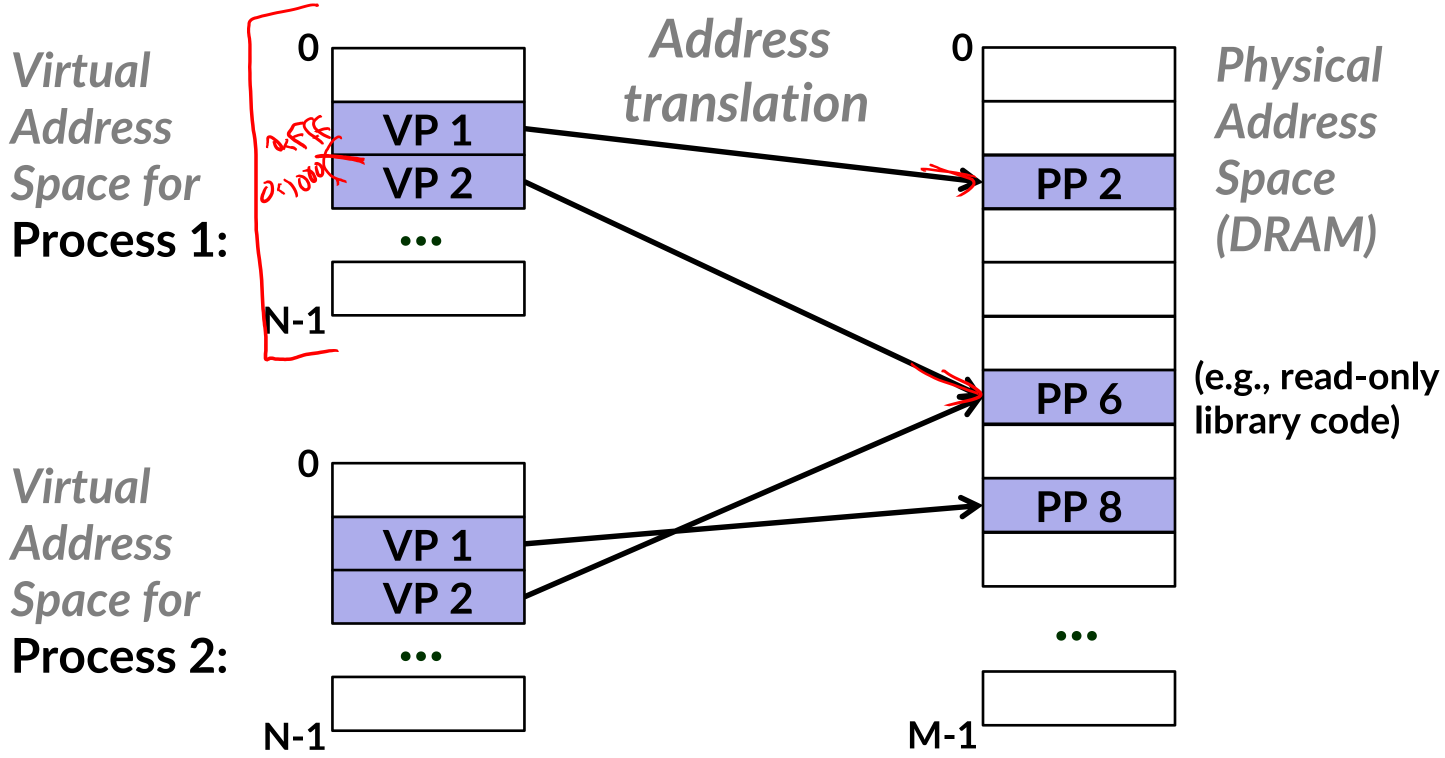

Kernel virtual memory
Memory invisible to user code
User stack (created at runtime)
%rsp (stack pointer)
Memory-mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
0x400000
Unused
0

Simplifying Linking and Loading


Kernel virtual memory
Memory invisible to user code
User stack (created at runtime)
%rsp (stack pointer)
Memory-mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
Unused
0x400000
0
execv
Physical memory
PP 0
PP 1
[Proc1] User stack
[Slack] .text
PP M-1
Chrome
Disk

VM for Managing Multiple Processes

- **Key abstraction: each process has its own virtual address space**
  - It can view memory as *a simple linear array*
- **With virtual memory, this simple linear virtual address space need not be contiguous in physical memory**
  - Process needs to store data in another VP? Just map it to *any* PP!



Virtual Address Space for **Process 1:**

0
0xFFF
0x1000
VP 1
VP 2
...
N-1

Address translation

Virtual Address Space for **Process 2:**

0
VP 1
VP 2
...
N-1

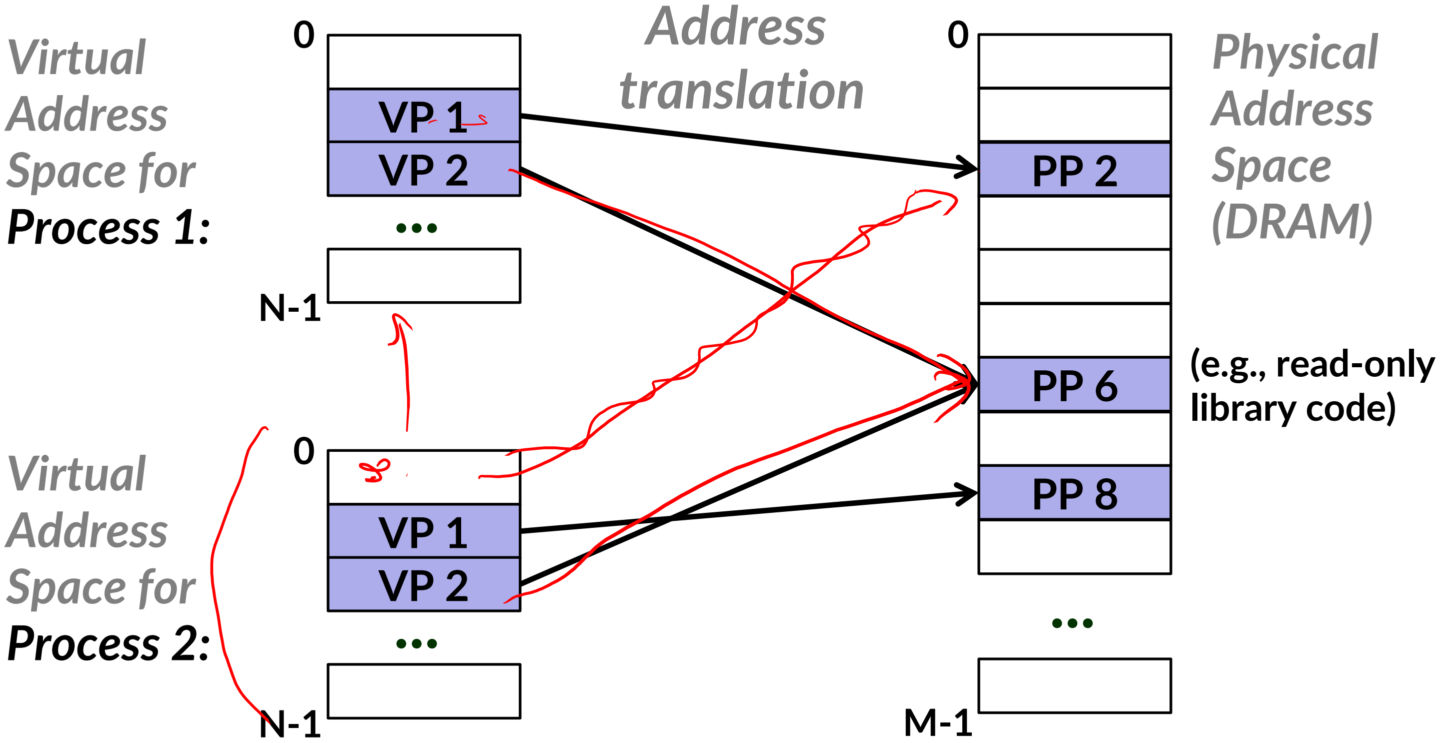
0
PP 2
PP 6
PP 8
...
M-1

Physical Address Space (DRAM)

(e.g., read-only library code)

VM for Protection and Sharing

- **The mapping of VPs to PPs provides a simple mechanism to *protect* memory and to *share* memory between processes**
  - Sharing: just map virtual pages in separate address spaces to the same physical page (here: PP 6)
  - Protection: process simply can't access physical pages to which none of its virtual pages are mapped (here: Process 2 can't access PP 2)



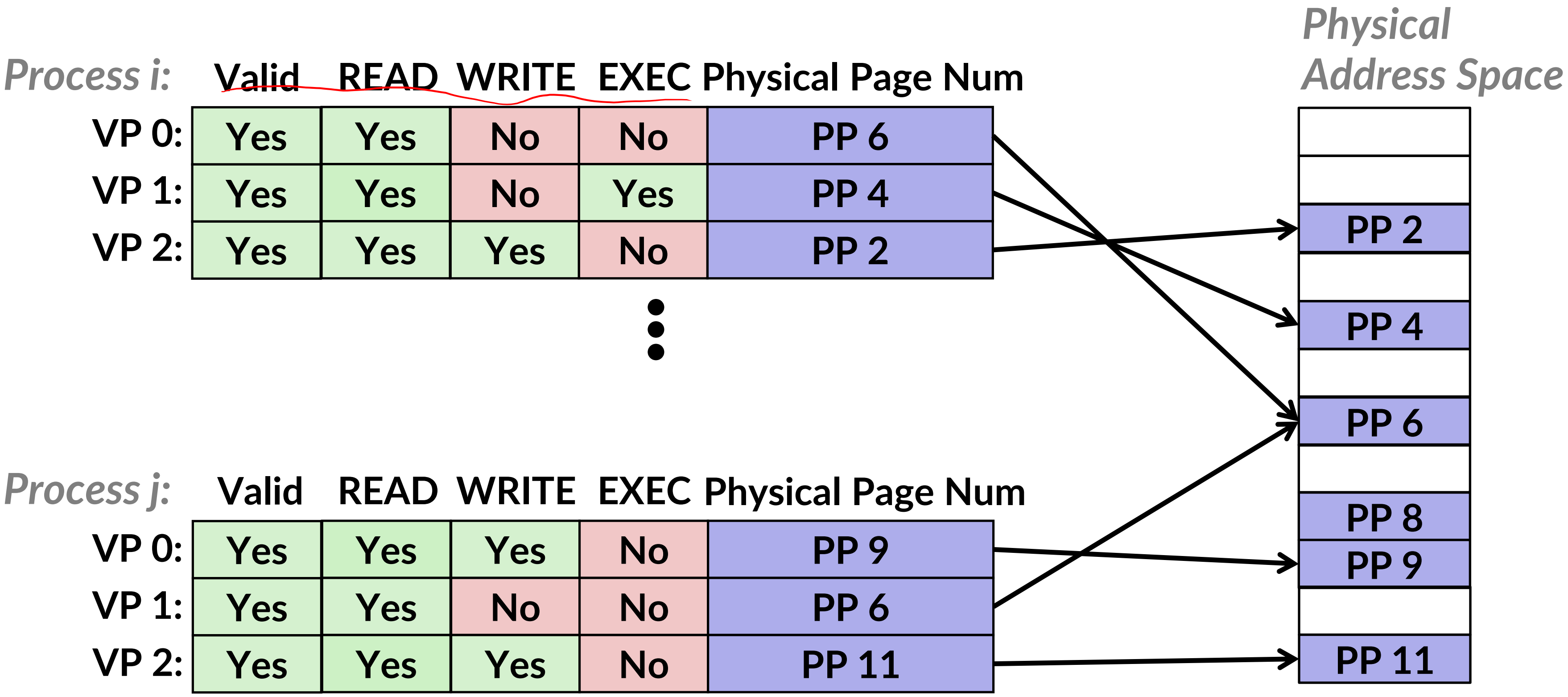
Virtual Address Space for Process 1:
0
VP 1
VP 2
…
N-1
Address translation
0
PP 2
PP 6
PP 8
…
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
…
N-1

Memory Protection Within a Single Process

- Can we use virtual memory to control read/write/execute permissions? How?

Memory Protection Within a Single Process

- **Extend page table entries with permission bits**
- **MMU checks these permission bits on every memory access**
  - If violated, raises exception and OS sends SIGSEGV signal to process (segmentation fault)



Process i:

	Valid	READ	WRITE	EXEC	Physical Page Num
VP 0:	Yes	Yes	No	No	PP 6
VP 1:	Yes	Yes	No	Yes	PP 4
VP 2:	Yes	Yes	Yes	No	PP 2

•
•
•

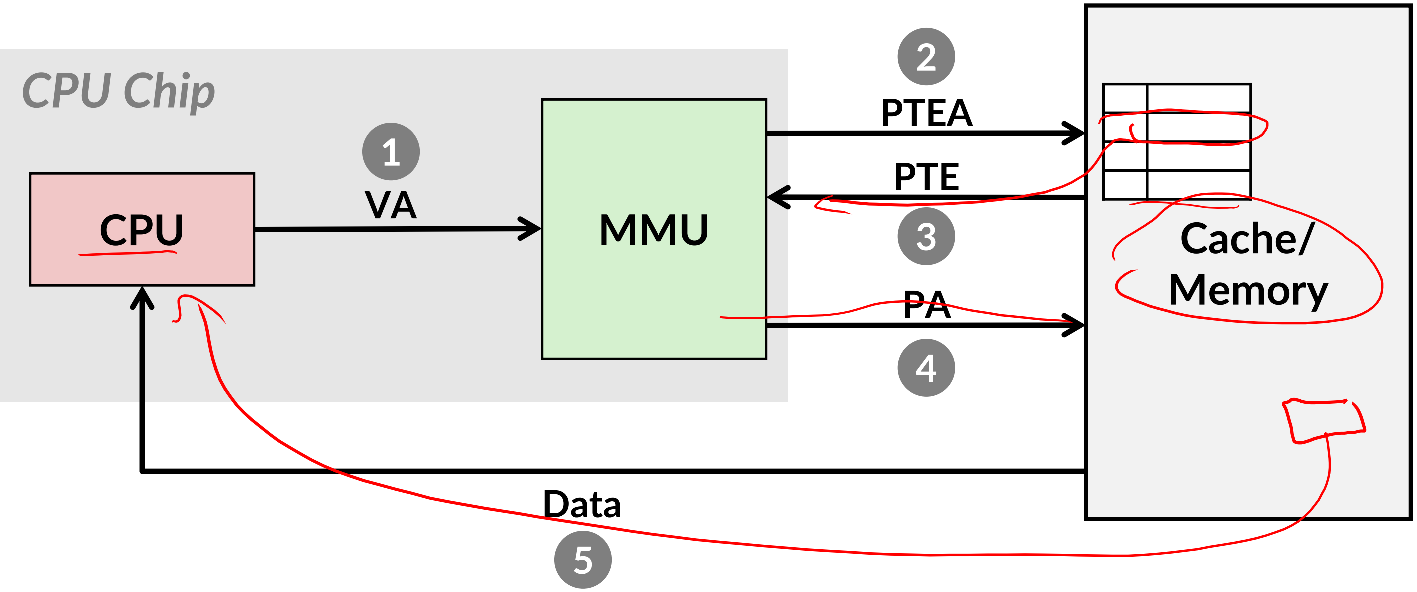
Process j:

	Valid	READ	WRITE	EXEC	Physical Page Num
VP 0:	Yes	Yes	Yes	No	PP 9
VP 1:	Yes	Yes	No	No	PP 6
VP 2:	Yes	Yes	Yes	No	PP 11

Terminology

- **context switch**
  - Switch between processes on the same CPU
- **page in**
  - Move pages of virtual memory from disk to physical memory
- **page out**
  - Move pages of virtual memory from physical memory to disk
- **thrash**
  - Total working set size of processes is larger than physical memory
  - Most time is spent paging in and out instead of doing useful computation

Address Translation: Page Hit



1) Processor sends *virtual* address to MMU (*memory management unit*)

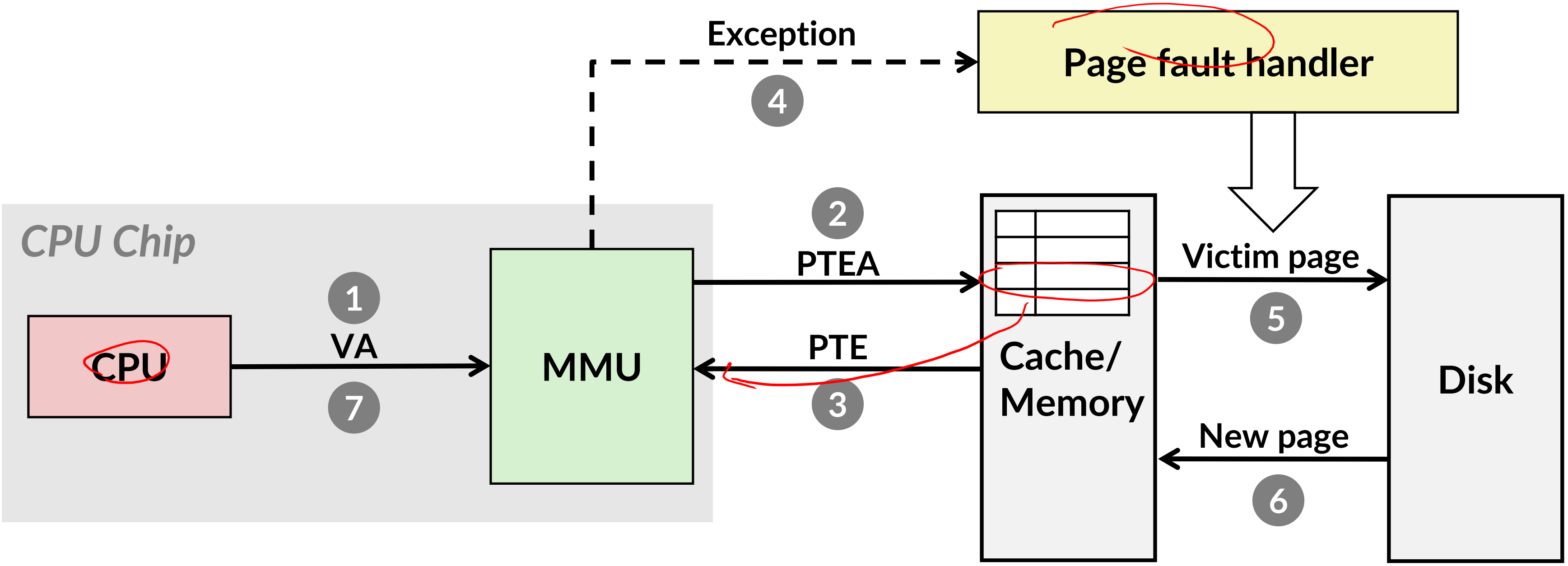
2-3) MMU fetches PTE from page table in cache/memory
(Uses PTBR to find beginning of page table for current process)

4) MMU sends *physical* address to cache/memory requesting data

5) Cache/memory sends data (~1 word) to processor

VA = Virtual Address
PA = Physical Address
PTEA = Page Table Entry Address
PTE= Page Table Entry
Data = Contents of memory stored at VA originally requested by CPU

Address Translation: 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

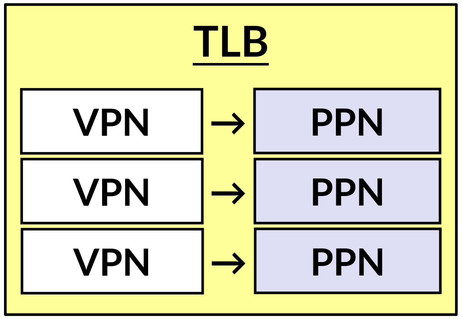
Hmm… Translation Sounds Slow!

- **The MMU accesses memory *twice*: once to get the PTE for translation, and then again for the actual memory request**
  - The PTEs *may* be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles

- *What can we do to make this faster?*

Speeding up Translation with a TLB

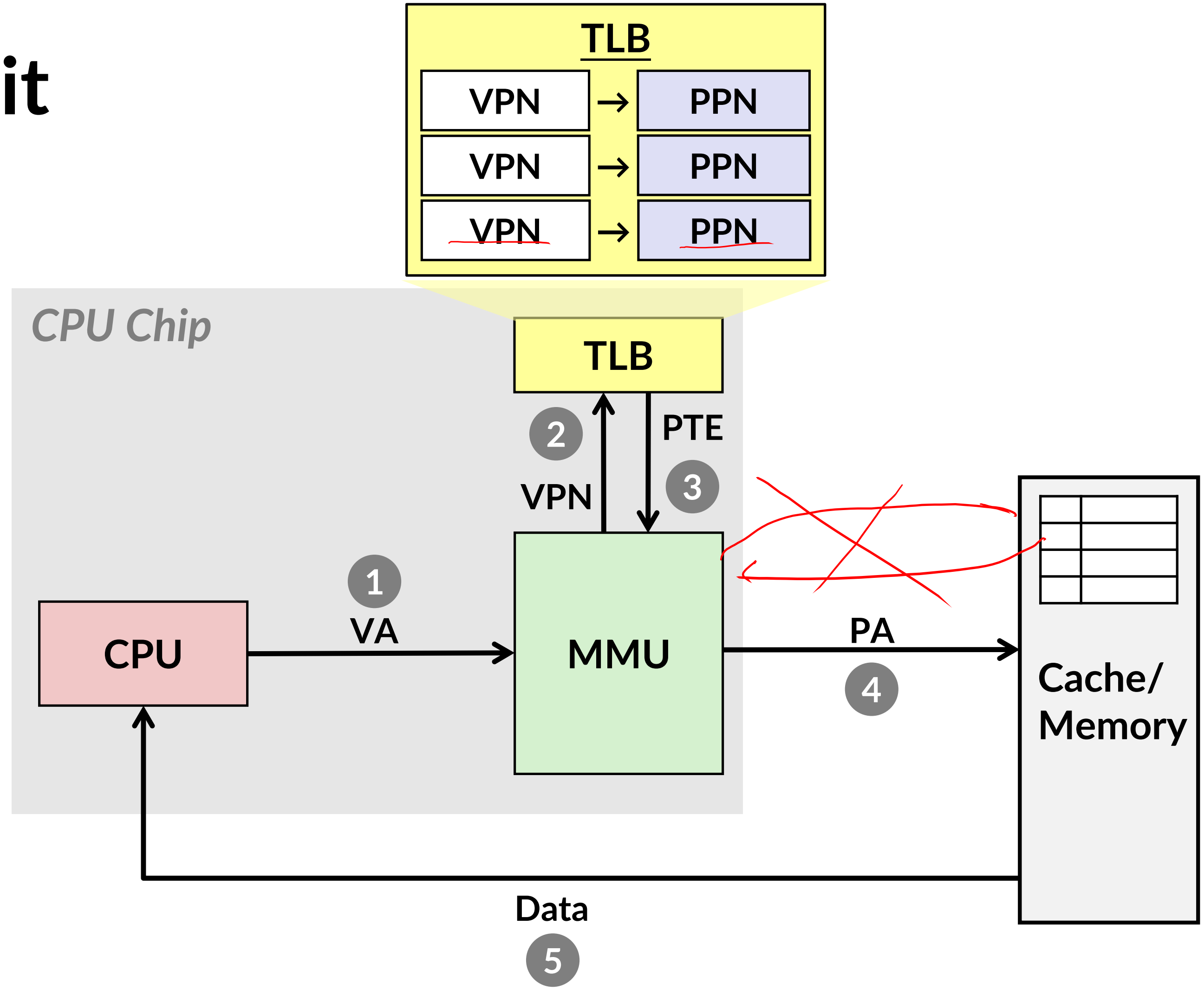
- **Solution: add another cache!** 🎉 ←
- ***Translation Lookaside Buffer*** **(TLB):**
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete *page table entries* for small number of pages
    - Modern Intel processors: 128 or 256 entries in TLB
  - Much faster than a page table lookup in cache/memory



TLB

VPN → PPN
VPN → PPN
VPN → PPN

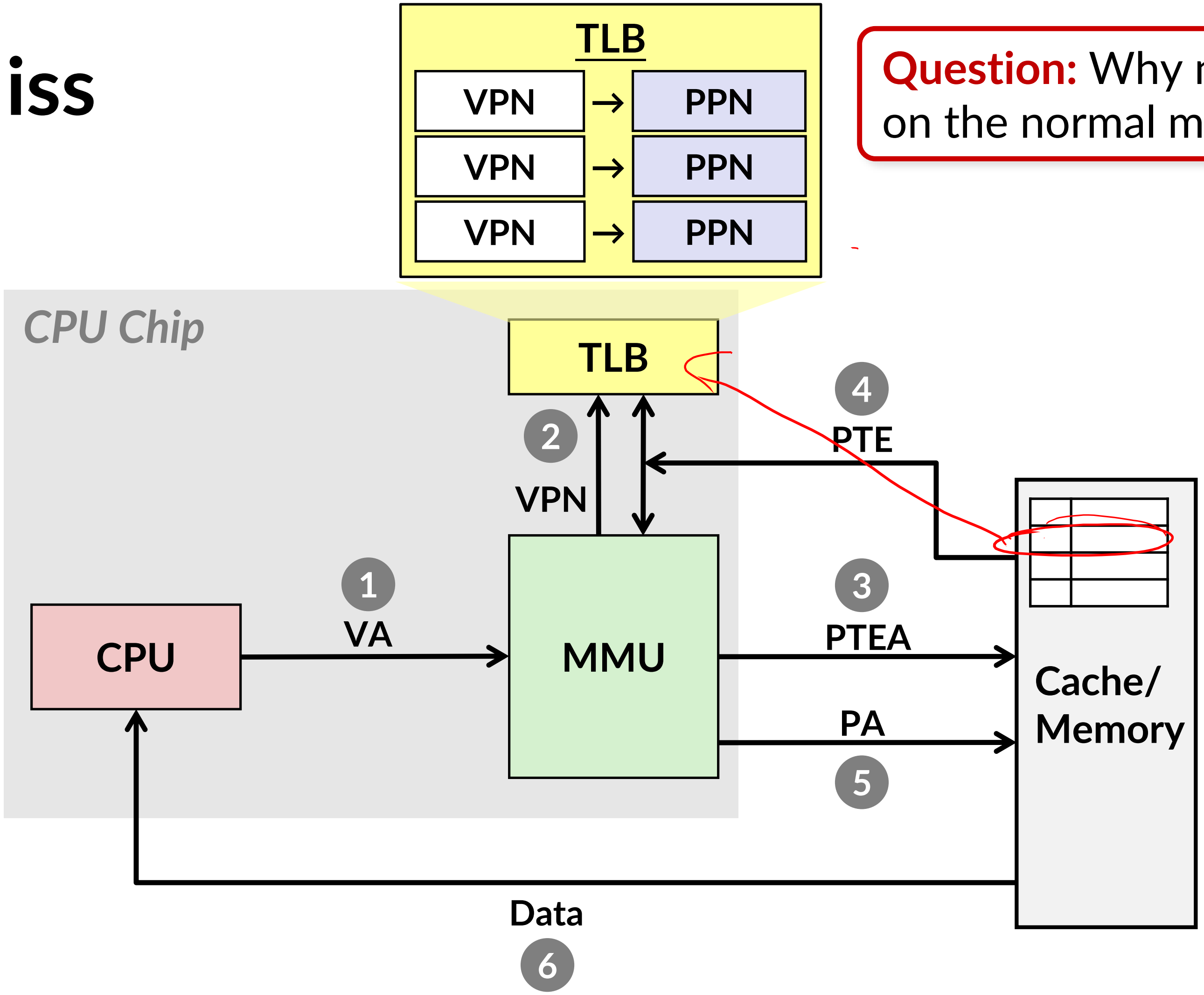
TLB Hit



A TLB hit eliminates a memory access

TLB Miss

Question: Why not just rely on the normal memory cache?



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

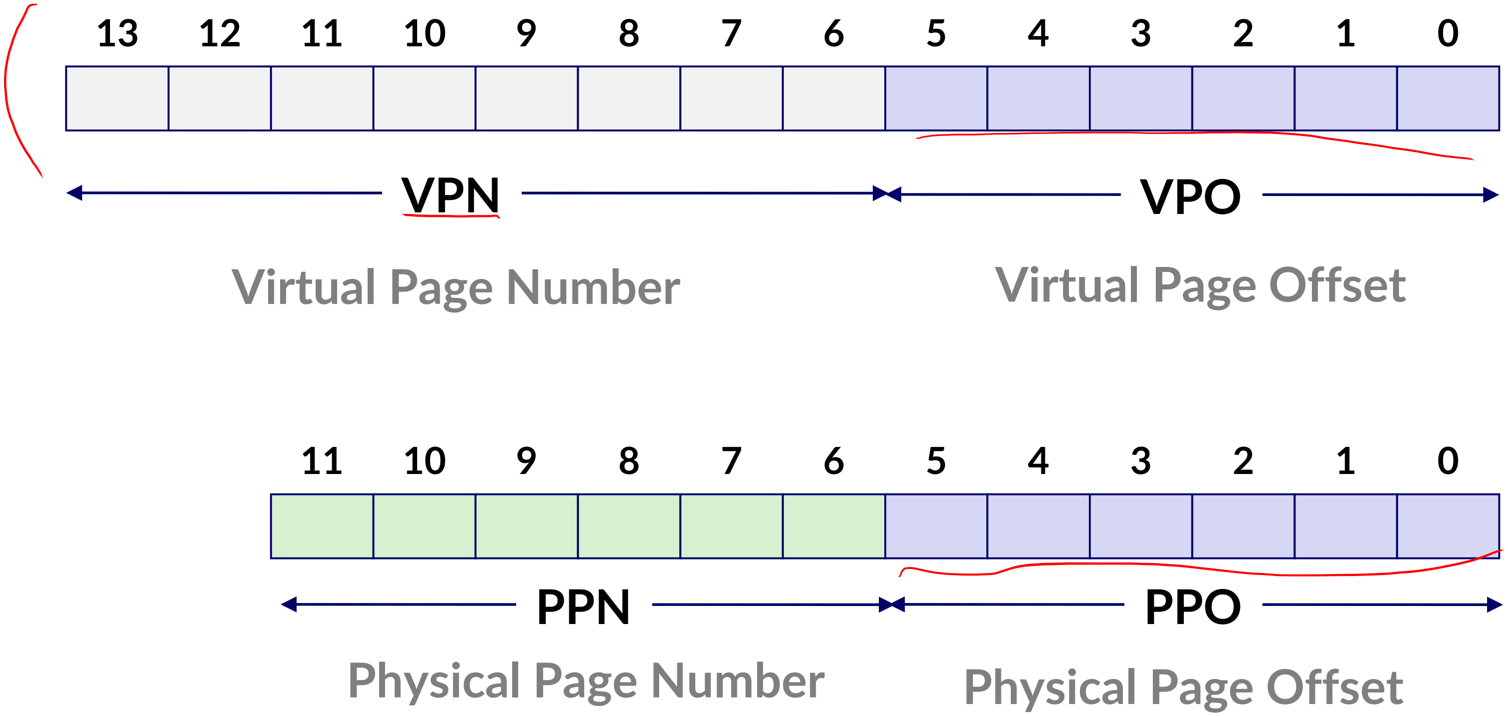
Fortunately, TLB misses are rare.

Summary of Address Translation Symbols

- **Basic Parameters**
  - **N = 2^n** : Number of addresses in virtual address space
  - **M = 2^m** : Number of addresses in physical address space
  - **P = 2^p** : Page size (bytes)
- **Components of the virtual address (VA)**
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number

Simple Memory System Example (small)

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



Simple Memory System Page Table

- Only showing first 16 entries (out of 256 = 2^8)

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

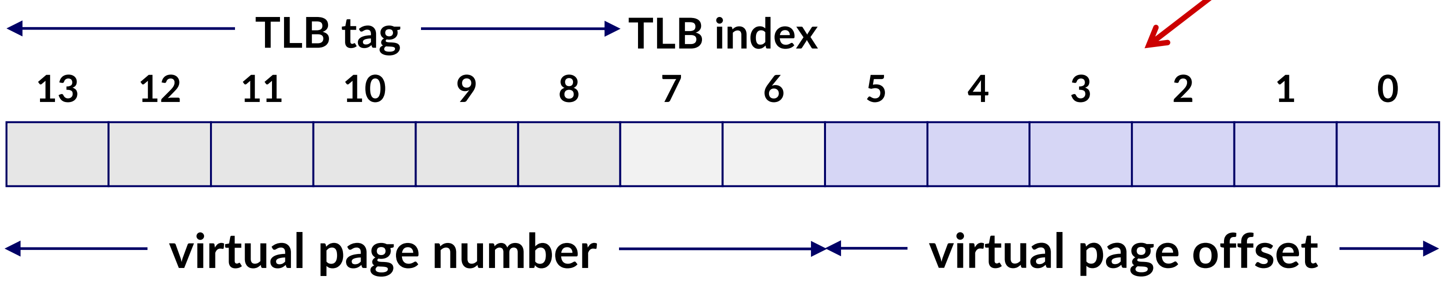
VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

- What about a real address space? Read more in the book…

Simple Memory System TLB

- 16 entries total
- 4 sets
- 4-way associative

TLB ignores page offset. Why?



TLB tag

TLB index

13 12 11 10 9 8 7 6 5 4 3 2 1 0

virtual page number

virtual page offset

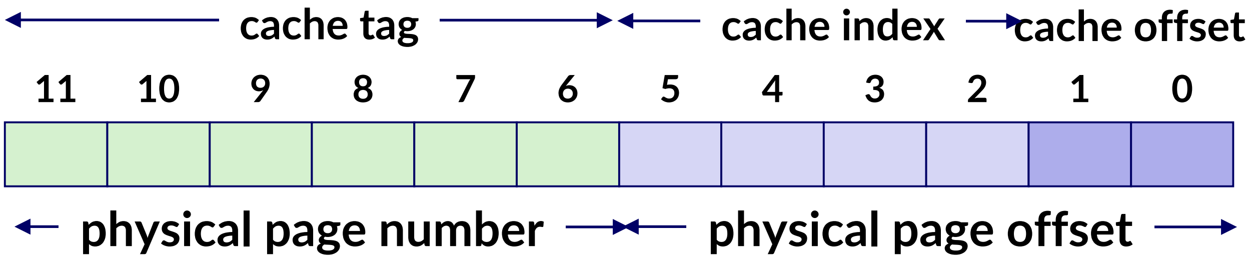
VPN

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 Cache

Note: It is a coincidence that the physical page number is the same bits as the cache tag

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



Index	Tag	Valid	B0	B1	B2	B3
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	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Index	Tag	Valid	B0	B1	B2	B3
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	–	–	–	–

So…

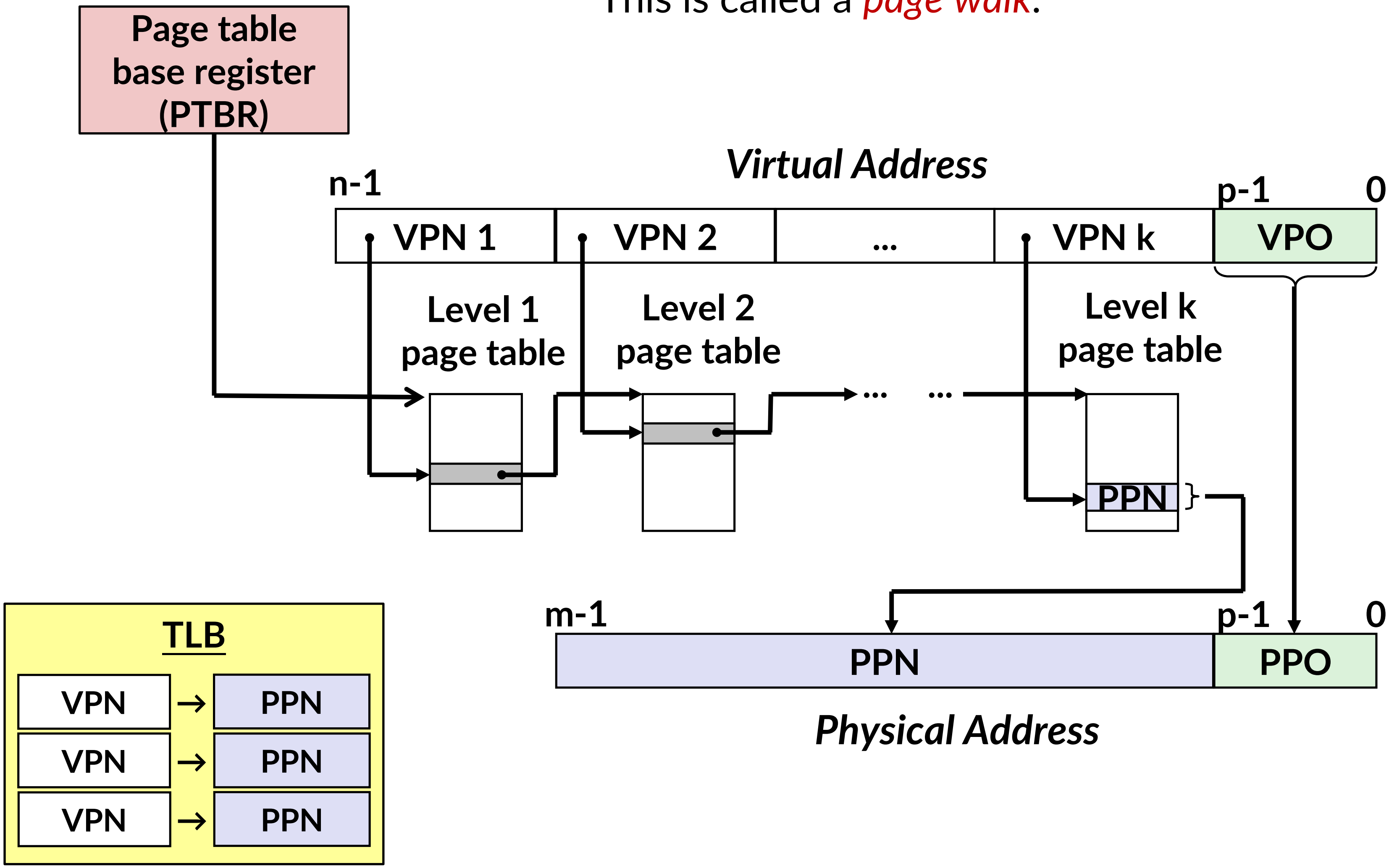
- **This seems complicated, but also elegant and effective**
  - Level of indirection to provide isolated memory, caching, etc.
  - TLB as a cache-of-a-page-table to avoid "two trips to memory for one load"

- **Just one issue… Numbers don't work out for the story so far!**

- **The problem is the page-table itself for each process…**
  - Suppose 64-bit addresses and 8KB pages
  - How many page-table-entries is that? (Also: Each PTE is > 1byte)
  - Moral: Cannot use this naïve implementation of the virtual→physical-page mapping: It's *way* too big.

A solution: Multi-level page tables

This is called a *page walk*.



Page table base register (PTBR)

Virtual Address

n-1

p-1

0

VPN 1

VPN 2

...

VPN k

VPO

Level 1 page table

Level 2 page table

... ...

Level k page table

PPN

m-1

p-1

0

PPN

PPO

Physical Address

TLB

VPN → PPN

VPN → PPN

VPN → PPN

This works!

- **Just a tree of depth k (e.g., 4) where each node at depth i has up to 2^k children if part i of the VPN has k bits**
- **Hardware for multi-level page tables inherently more complicated**
  - But it's a necessary complexity: 1-level does not fit
- **Why it works: Most subtrees are not used at all, so they are never created and definitely aren't in physical memory**
  - Even parts created can be evicted from cache/memory when not being used
  - Each node can have a size of ~1-100KB
- **But now for a k-level page table, a TLB miss requires k+1 cache/memory accesses**
  - Fine so long as TLB misses are rare: motivates larger TLBs

Summary

- **Programmer's view of virtual memory**
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- **System view of virtual memory**
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and sharing
  - Simplifies protection by providing a convenient interpositioning point to check permissions

Memory System Summary

- **L1/L2 Memory Cache**
  - Purely a speed-up technique
  - Behavior invisible to application programmer and (mostly) OS
  - Implemented totally in hardware
- **Virtual Memory**
  - Supports many OS-related functions
    - Process creation, task switching, protection
  - Operating System (software)
    - Allocates/shares physical memory among processes
    - Maintains high-level tables tracking memory type, source, sharing
    - Handles exceptions, fills in hardware-defined mapping tables
  - Hardware
    - Translates virtual addresses via mapping tables, enforcing permissions
    - Accelerates mapping via translation cache (TLB)

Memory System – Who controls what?

- **L1/L2 Memory Cache**
  - Controlled by hardware
  - Programmer cannot control it
  - Programmer ***can*** write code in a way that takes advantage of it
- **Virtual Memory**
  - Controlled by OS and hardware
  - Programmer cannot control mapping to physical memory
  - Programmer can control sharing and some protection
    - via OS functions (not in CSE 351)

Quick Review

- **What do Page Tables map?**

- **Where are Page Tables?**

- **How many Page Tables are there?**

- **Can your program tell if a page fault has occurred?**

- **What is thrashing?**

- **T/F: Virtual Addresses that are contiguous will always be contiguous in physical memory.**

- **TLB stands for ________________ and stores _____________**

Quick Review Answers

- **What do Page Tables map?**
  - Virtual pages to physical pages or location on disk
- **Where are Page Tables?**
  - In physical memory
- **How many Page Tables are there?**
  - One per process
- **Can your program tell if a page fault has occurred?**
  - Nope. But it has to wait a long time.
- **What is thrashing?**
  - Constantly paging out and paging in. The working set of all applications you are trying to run is bigger than physical memory.
- **T/F: Virtual Addresses that are contiguous will always be contiguous in physical memory. (could be on different physical pages)**
  - False; *pages* can be mapped anywhere (within a page they are contiguous)
- **TLB stands for Translation Lookaside Buffer, and stores page table entries.**

Virtual Memory Handout

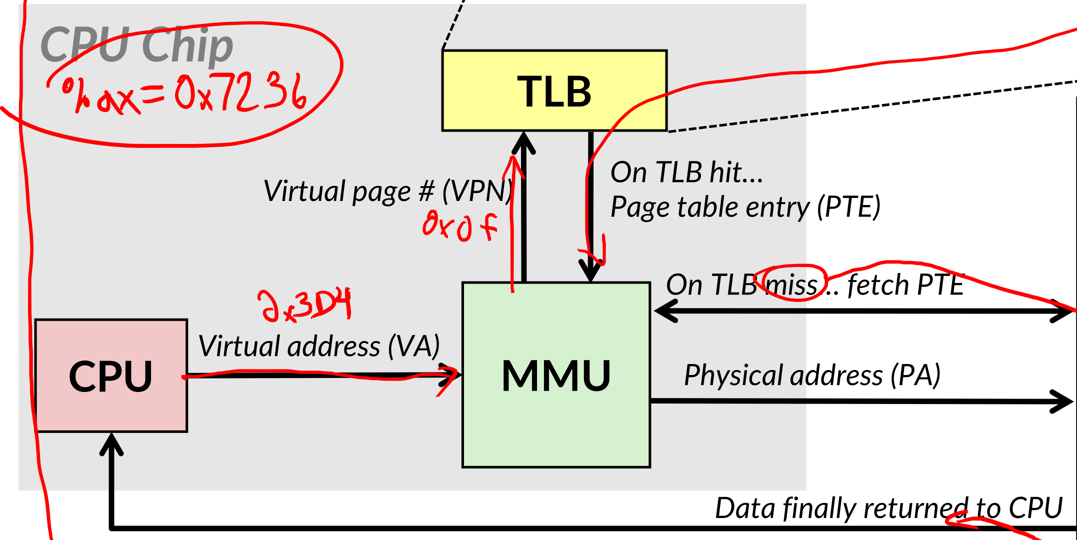
```
movw 0x3D4, ~~%rax~~
```

%ax

0011 1101 0100

TLB

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



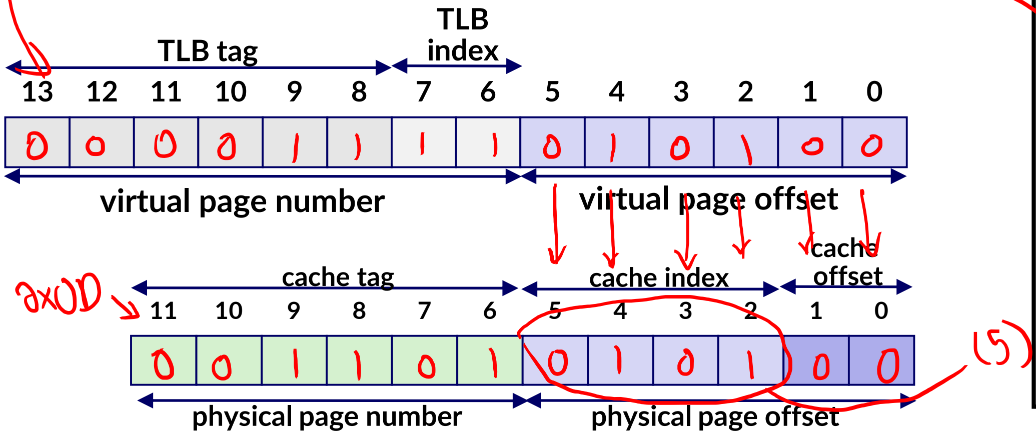
Cache/ Memory

Page Table

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Cache

Index	Tag	Valid	B0	B1	B2	B3
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	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

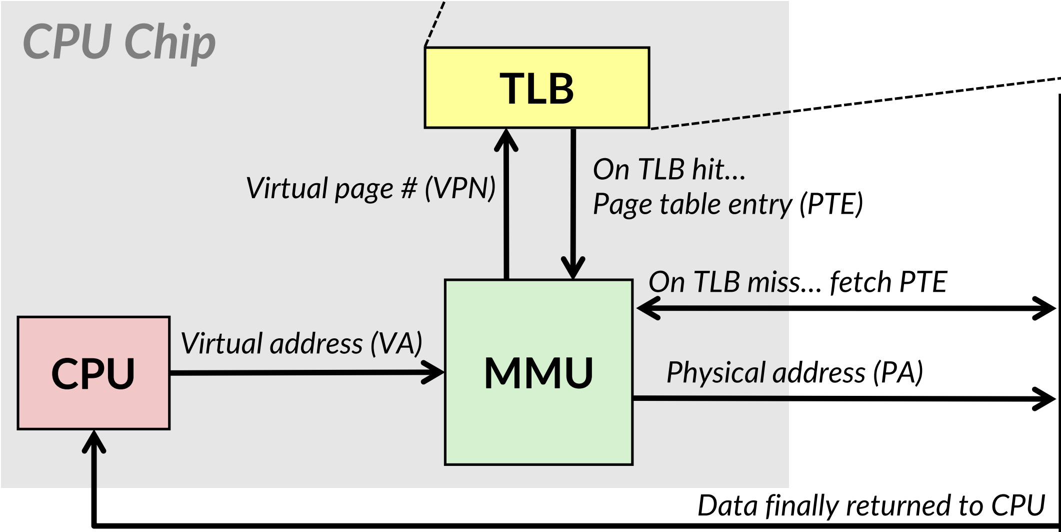


Virtual Memory Handout

```
movw 0x3D4, %rax
%rax = 0x7236
```

TLB

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



Cache/ Memory

Page Table

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	–	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	–	0
04	–	0	0C	–	0
05	16	1	0D	2D	1
06	–	0	0E	11	1
07	–	0	0F	0D	1

Cache

Index	Tag	Valid	B0	B1	B2	B3
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	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

TLB tag (bits 13–8) · TLB index (bits 7–6)

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

virtual page number (bits 13–6) · virtual page offset (bits 5–0)

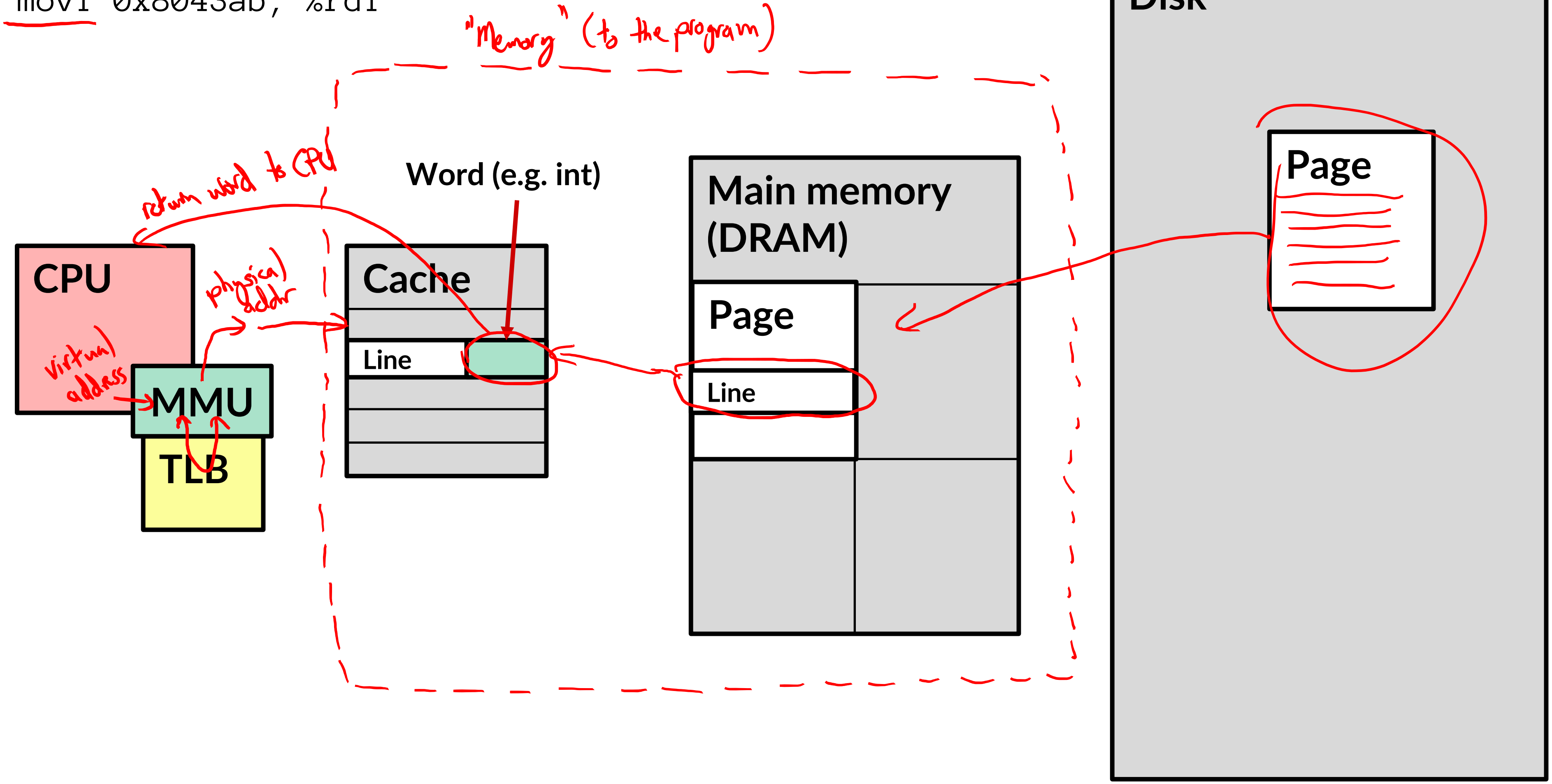
cache tag (bits 11–6) · cache index (bits 5–2) · cache offset (bits 1–0)

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

physical page number (bits 11–6) · physical page offset (bits 5–0)

Memory Overview

```
movl 0x8043ab, %rdi
```



Detailed Examples…

Current state of caches/tables

TLB

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

Page table (partial)

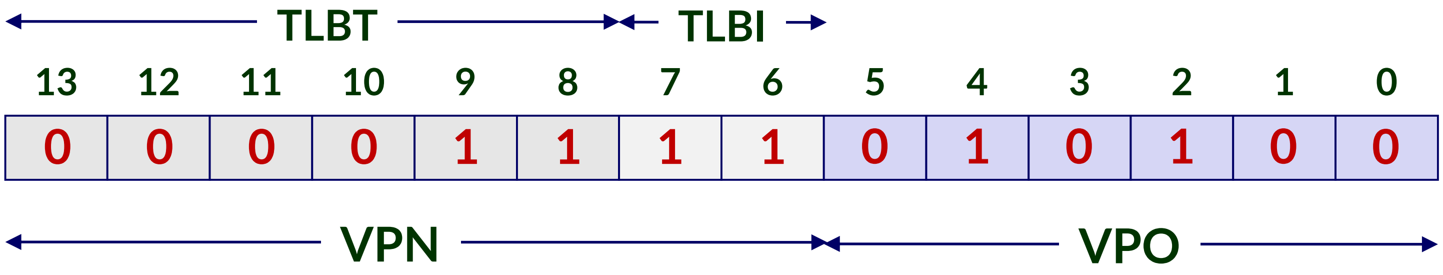
VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

Cache

Index	Tag	Valid	B0	B1	B2	B3
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	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03
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

Virtual Address: `0x03D4`



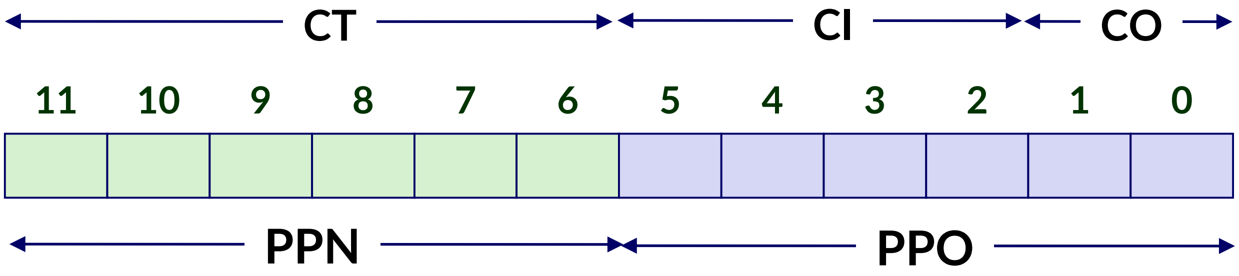
TLBT (bits 13–8) | TLBI (bits 7–6)

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN (bits 13–6) | VPO (bits 5–0)

VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

Physical Address



CT (bits 11–6) | CI (bits 5–2) | CO (bits 1–0)

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

PPN (bits 11–6) | PPO (bits 5–0)

CO ___ CI___ CT ____ Hit? __ Byte: ____

Address Translation Example #1

Virtual Address: `0x03D4`

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0
VPN								VPO					

VPN 0x0F TLBI 3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

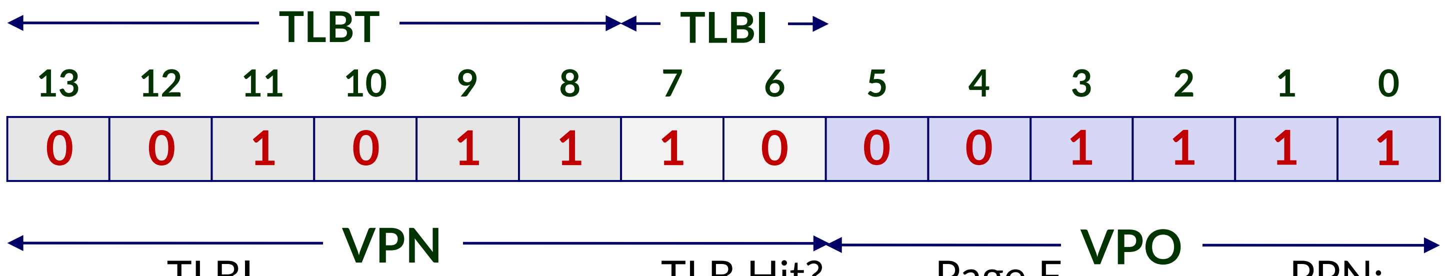
Physical Address

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0
PPN						PPO					

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

Address Translation Example #2

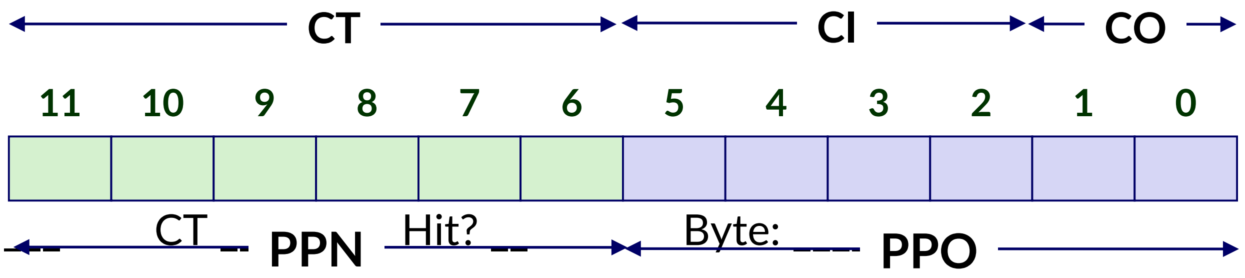
Virtual Address: `0x0B8F`



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	0	0	1	1	1	1

VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

Physical Address

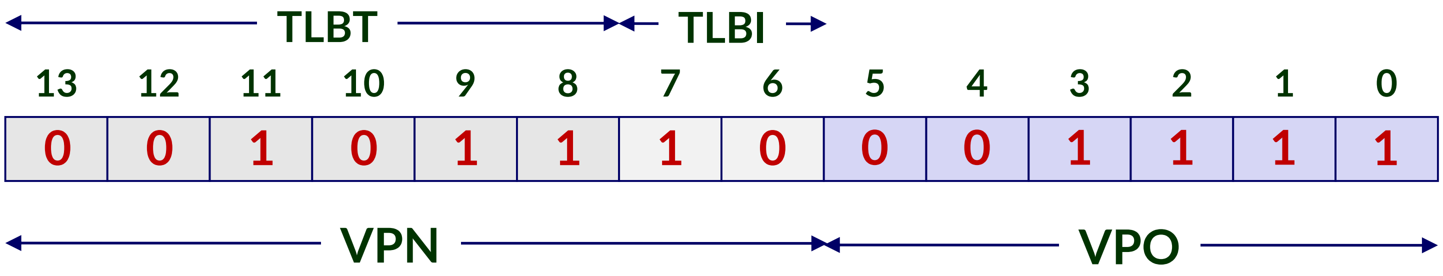


11	10	9	8	7	6	5	4	3	2	1	0

CO ___ CI ___ CT __ Hit? __ Byte: ____

Address Translation Example #2

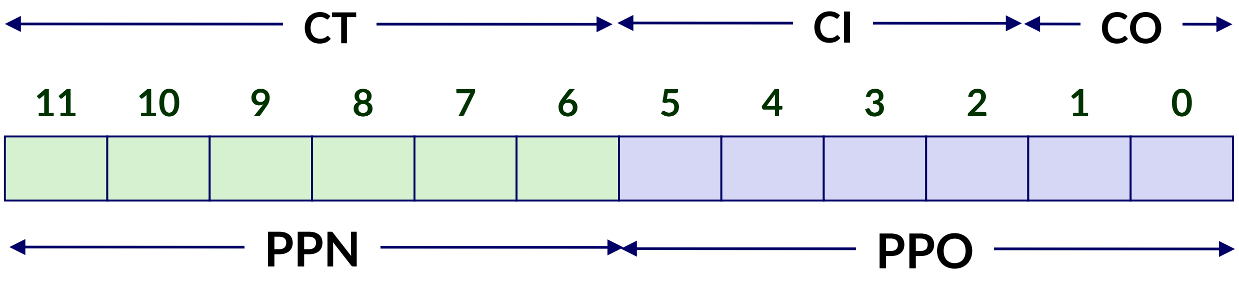
Virtual Address: `0x0B8F`



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	0	0	1	1	1	1

VPN 0x2E   TLBI _2   TLBT 0x0B   TLB Hit? N   Page Fault? ?   PPN: TBD

Physical Address

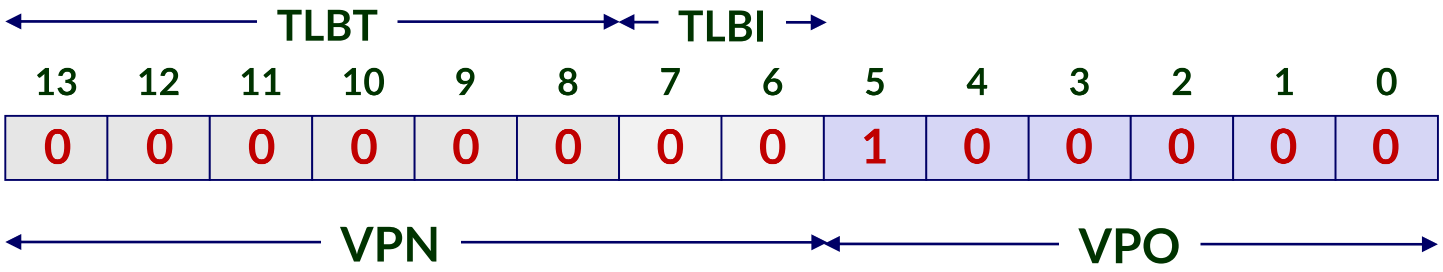


11	10	9	8	7	6	5	4	3	2	1	0

CO ___   CI___   CT ____   Hit? __   Byte: ____

Address Translation Example #3

Virtual Address: `0x0020`



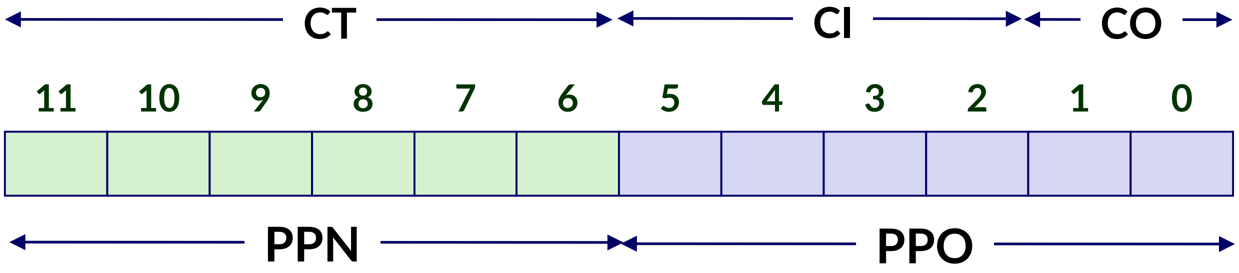
TLBT TLBI

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0

VPN VPO

VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

Physical Address



CT CI CO

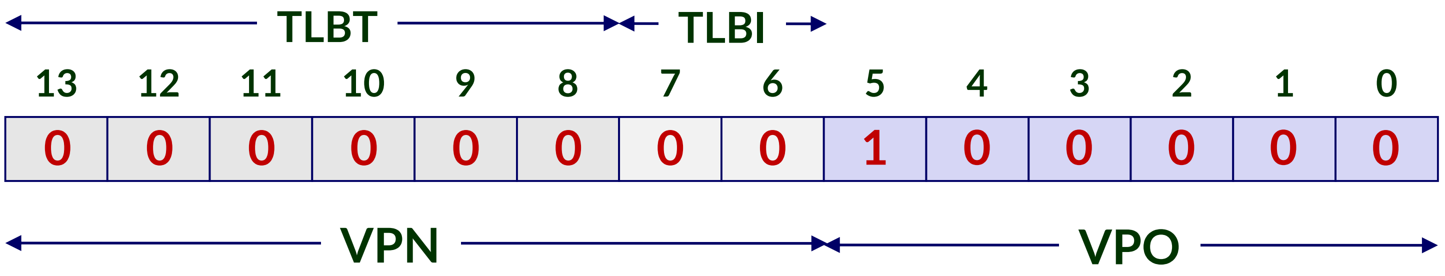
11	10	9	8	7	6	5	4	3	2	1	0

PPN PPO

CO___ CI___ CT ____ Hit? __ Byte: ____

Address Translation Example #3

Virtual Address: `0x0020`



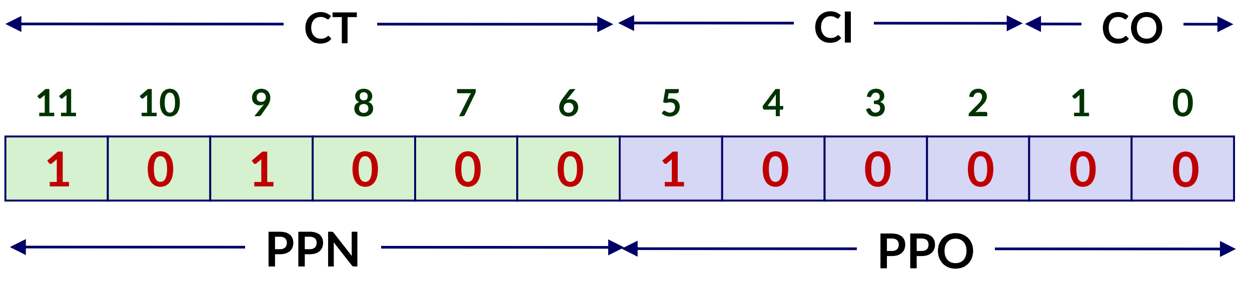
TLBT ← → TLBI

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0

VPN ← → VPO

VPN 0x00 TLBI _0_ TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

Physical Address



CT ← → CI ← → CO

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

PPN ← → PPO

CO_0_ CI 0x8 CT 0x28 Hit? N Byte: Mem

Address Translation Example #4

Virtual Address: `0x036B`

TLBT TLBI

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	0	1	0	1	1

VPN VPO

VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

Physical Address

CT CI CO

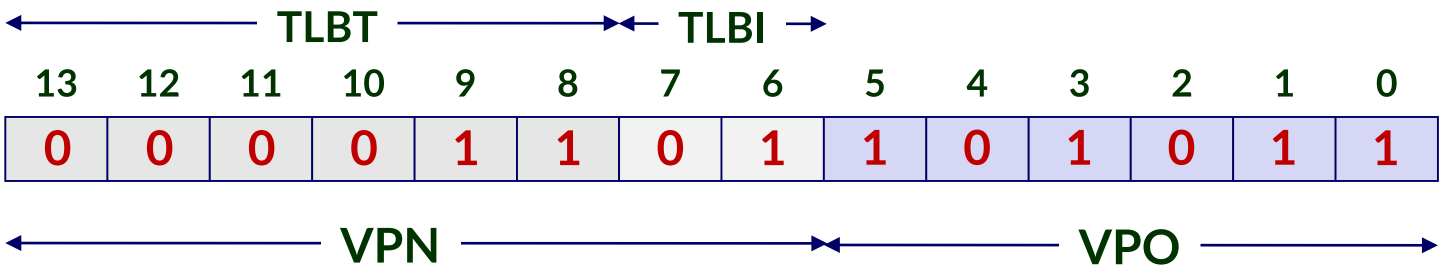
11	10	9	8	7	6	5	4	3	2	1	0

PPN PPO

CO ___ CI___ CT ____ Hit? __ Byte: ____

Address Translation Example #4

Virtual Address: `0x036B`



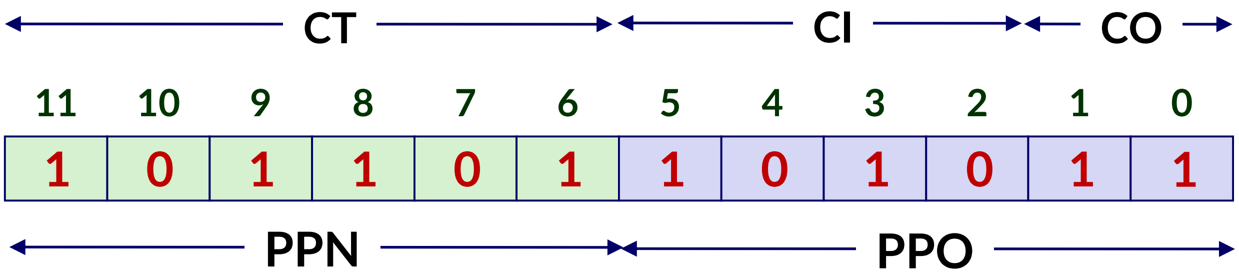
TLBT ← → TLBI

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	0	1	0	1	1

VPN ← → VPO

VPN 0x0D TLBI _1_ TLBT 0x03 TLB Hit? _Y_ Page Fault? _N_ PPN: 0x2D

Physical Address



CT ← → CI ← → CO

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

PPN ← → PPO

CO _3_ CI 0xA CT 0x2D Hit? _Y_ Byte: 0x3B