Today

- **Virtual memory (VM)**
  - Overview and motivation
  - VM as tool for caching
  - VM as tool for memory management
  - VM as tool for memory protection
  - Address translation

Processes

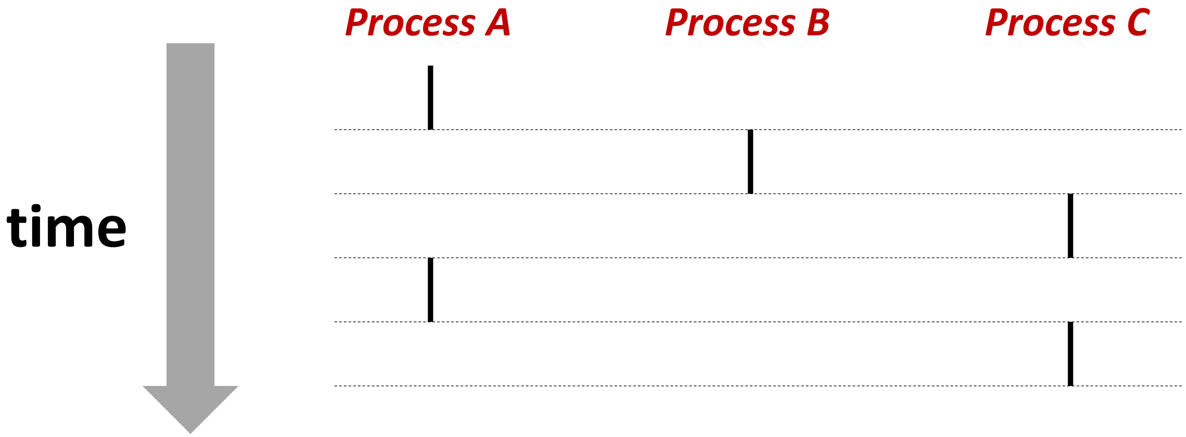
- **Definition: A *process* is an instance of a running program**
  - One of the most important ideas in computer science
  - Not the same as “program” or “processor”

- **Process provides each program with two key abstractions:**
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
  - Private virtual address space
    - Each program seems to have exclusive use of main memory

- **How are these Illusions maintained?**
  - Process executions interleaved (multi-tasking)
  - Address spaces managed by virtual memory system

Concurrent Processes

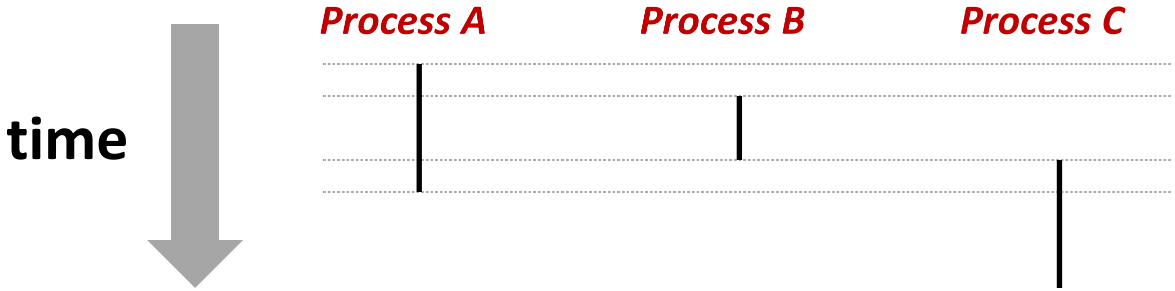
- **Two processes *run concurrently* (are concurrent) if their instruction executions (flows) overlap in time**
- **Otherwise, they are *sequential***
- **Examples:**
  - Concurrent: A & B, A & C
  - Sequential: B & C



User View of Concurrent Processes

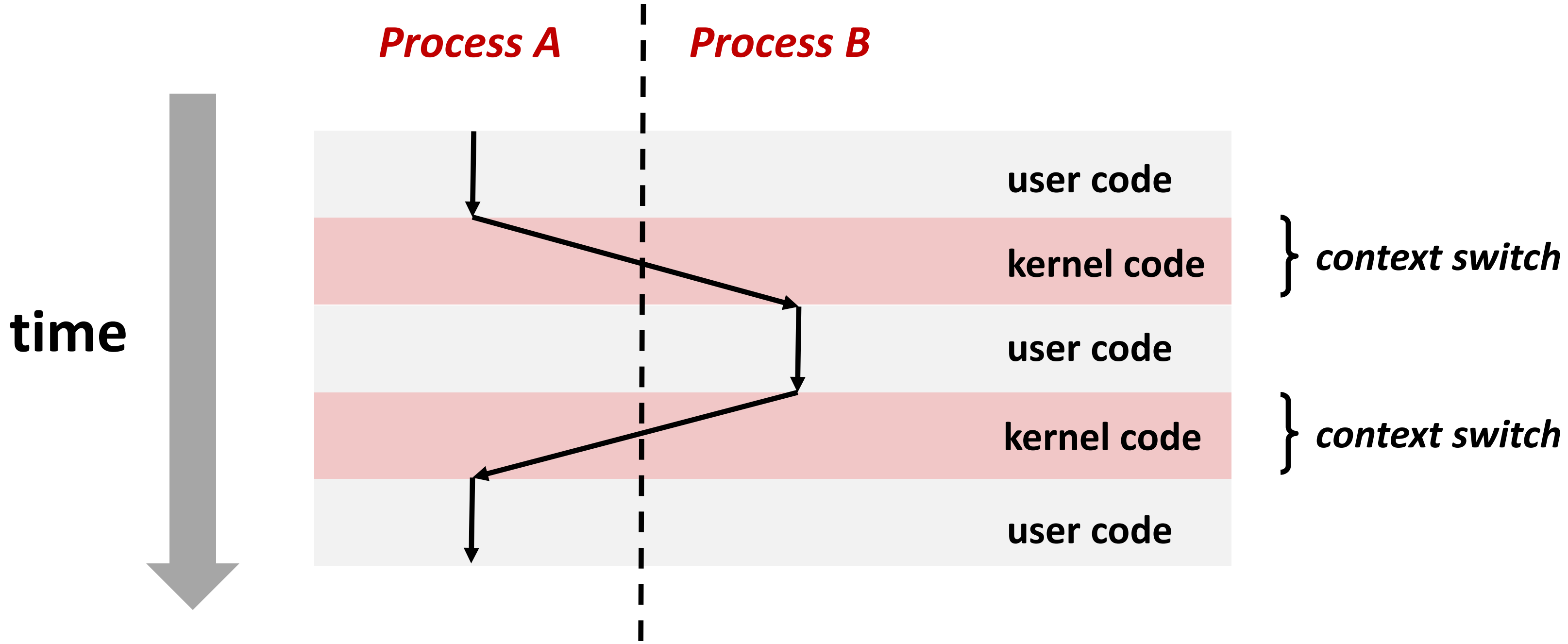
- Control flows for concurrent processes are physically disjoint in time

- However, we can think of concurrent processes as executing in parallel (only an illusion?)



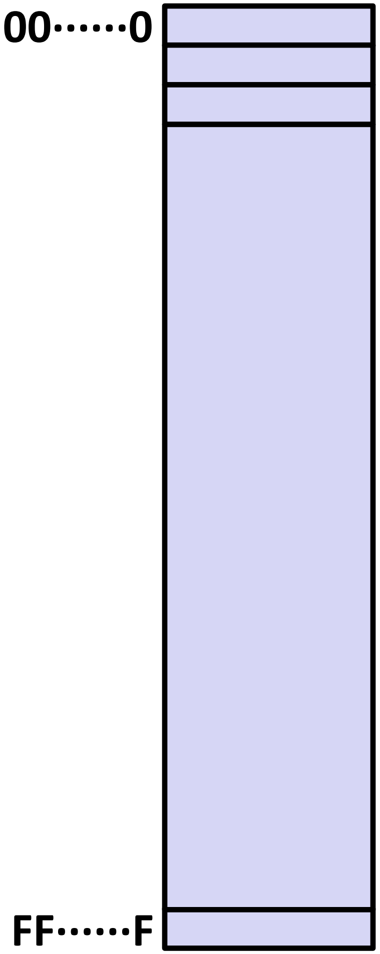
Context Switching

- **Processes are managed by a shared chunk of OS code called the *kernel***
  - Important: the kernel is not a separate process, but rather runs as part of a user process
- **Control flow passes from one process to another via a *context switch*…** *(how?)*



Virtual Memory (Previous Lectures)

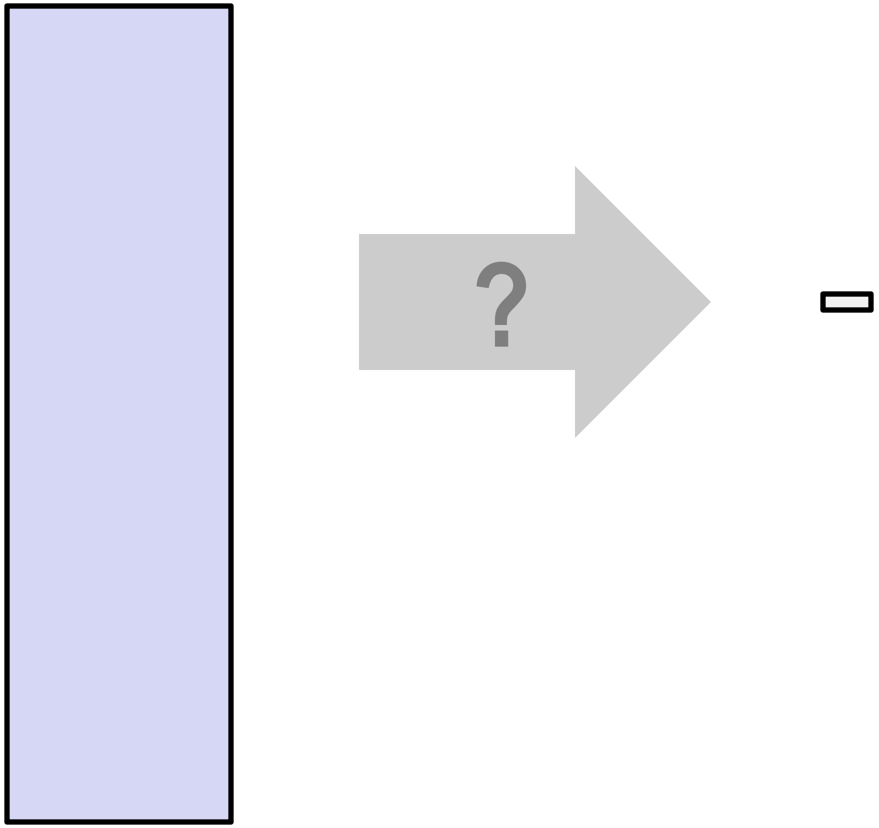
- **Programs refer to virtual memory addresses**
  - `movl (%ecx),%eax`
  - Conceptually very large array of bytes
  - Each byte has its own address
  - Actually implemented with hierarchy of different memory types
  - System provides address space private to particular “process”
- **Allocation: Compiler and run-time system**
  - Where different program objects should be stored
  - All allocation within single virtual address space
- ***But why virtual memory?***
- ***Why not physical memory?***



Problem 1: How Does Everything Fit?

64-bit addresses:
16 Exabyte

Physical main memory:
Few Gigabytes



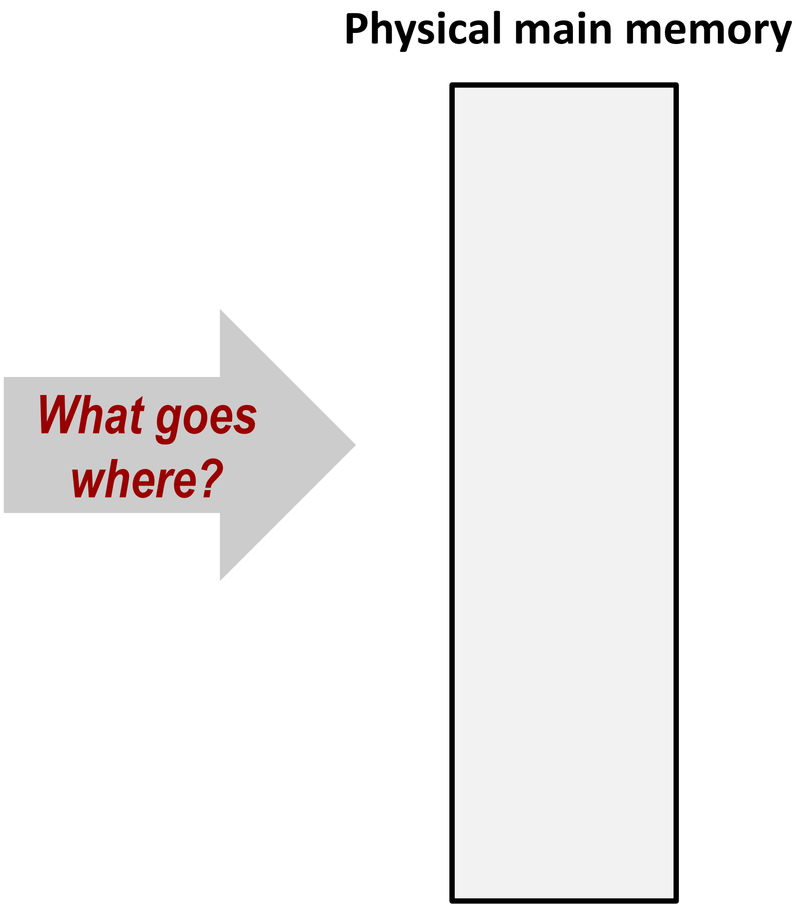
And there are many processes ….

Problem 2: Memory Management

Process 1
Process 2
Process 3
…
Process n

X

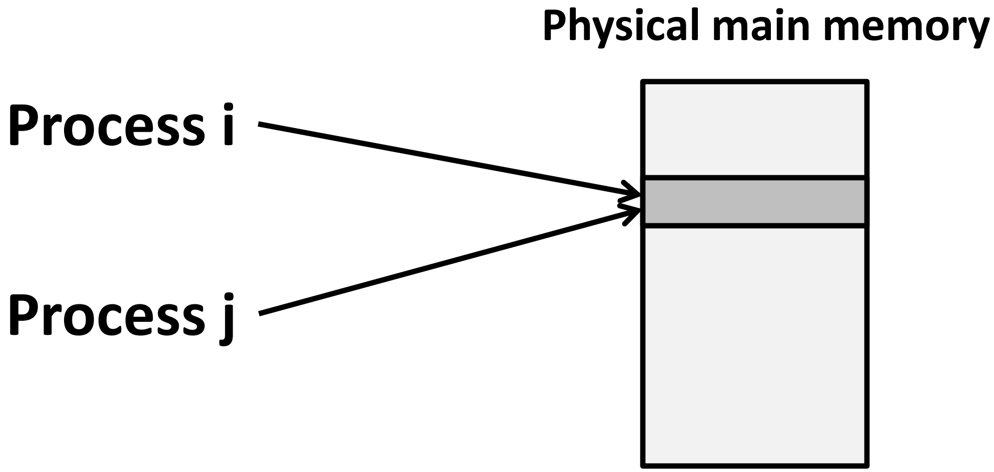
stack
heap
`.text`
`.data`
…



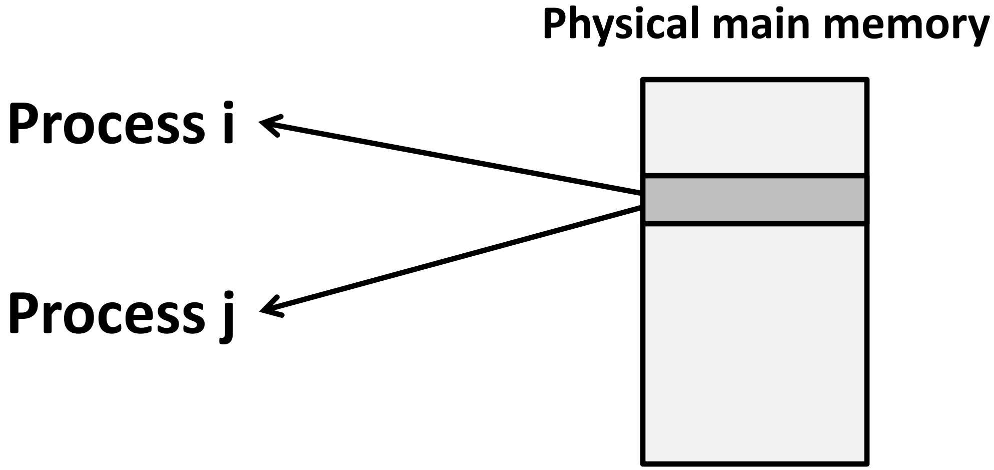
What goes where?

Physical main memory

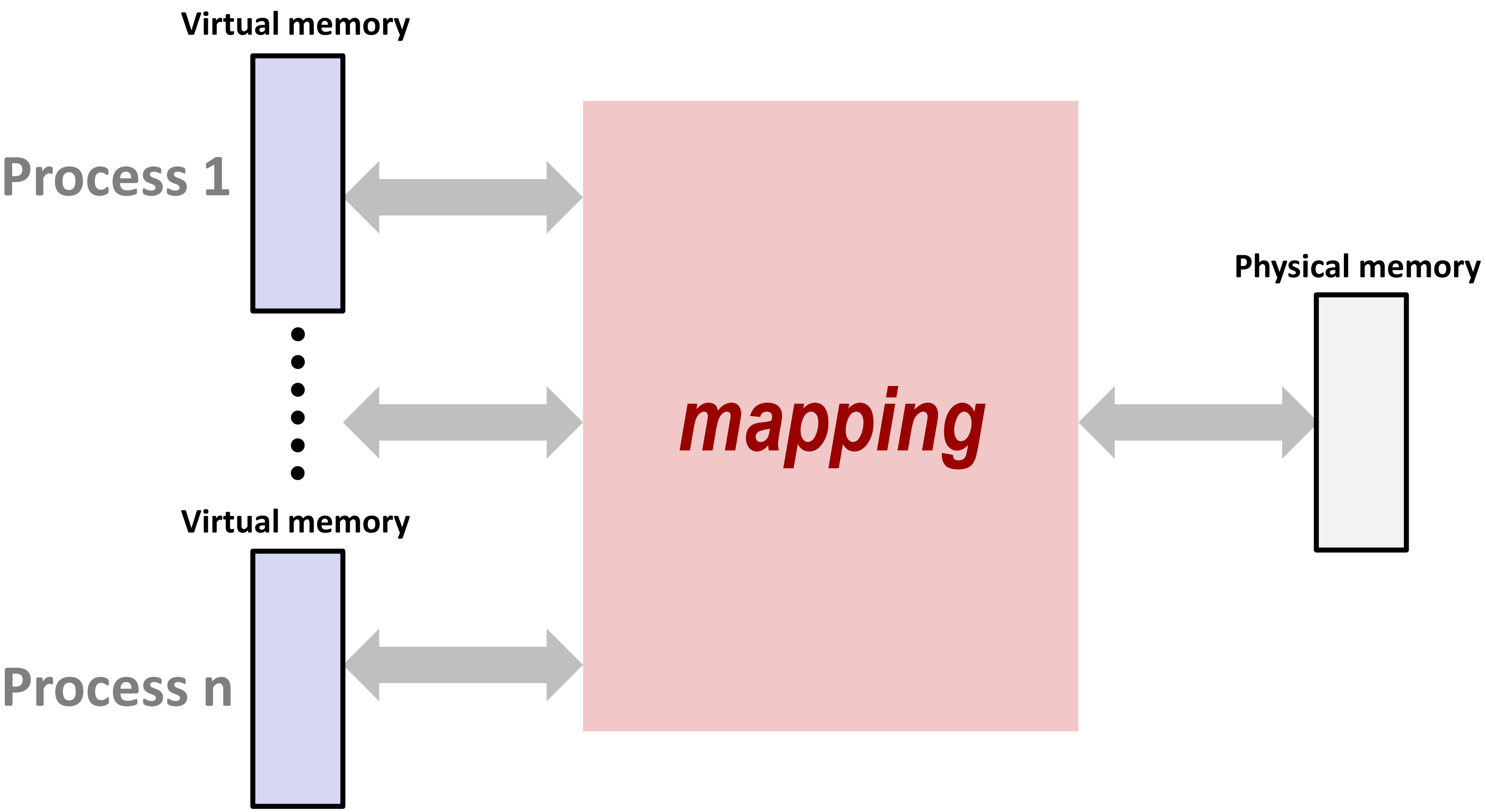
Problem 3: How To Protect



Problem 4: How To Share?



Solution: Level Of Indirection



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

Address Spaces

- **Linear address space:** Ordered set of contiguous non-negative integer addresses:

  {0, 1, 2, 3 … }

- **Virtual address space:** Set of N = 2^n virtual addresses

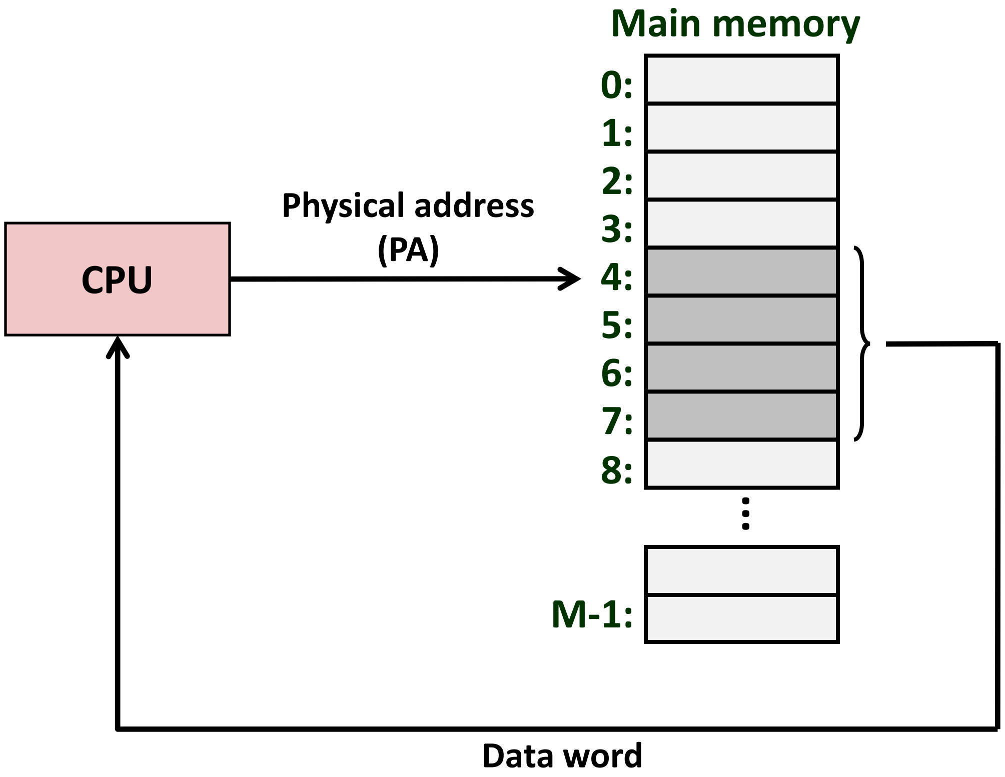
  {0, 1, 2, 3, …, N-1}

- **Physical address space:** Set of M = 2^m physical addresses (n >> m)

  {0, 1, 2, 3, …, M-1}

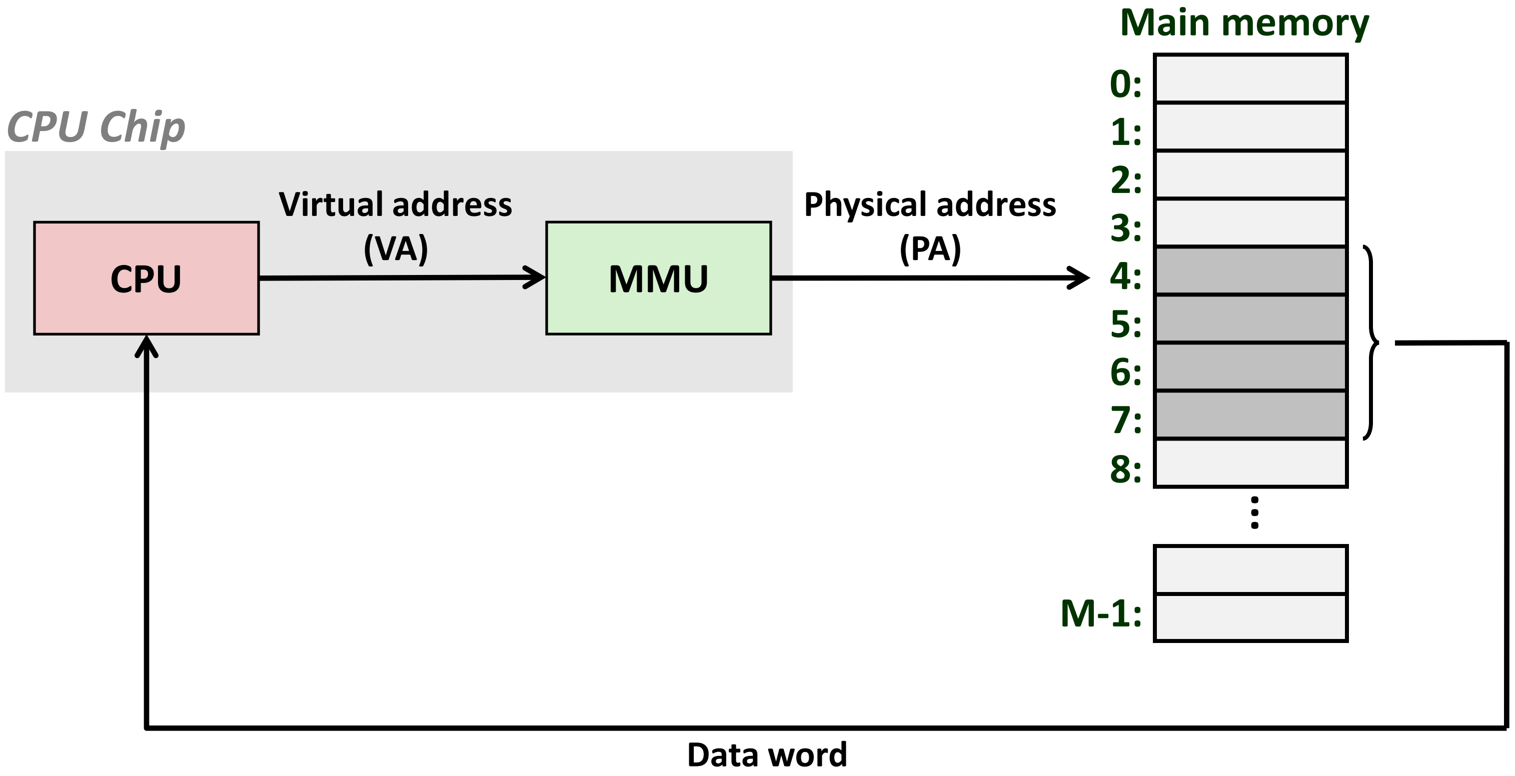
- **Clean distinction between data (bytes) and their attributes (addresses)**
- **Each object can now have multiple addresses**
- **Every byte in main memory:**
  one physical address, one (or more) virtual addresses

A System Using Physical Addressing



- Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



- Used in all modern desktops, laptops, workstations
- One of the great ideas in computer science
- *MMU checks the cache*

Why Virtual Memory (VM)?

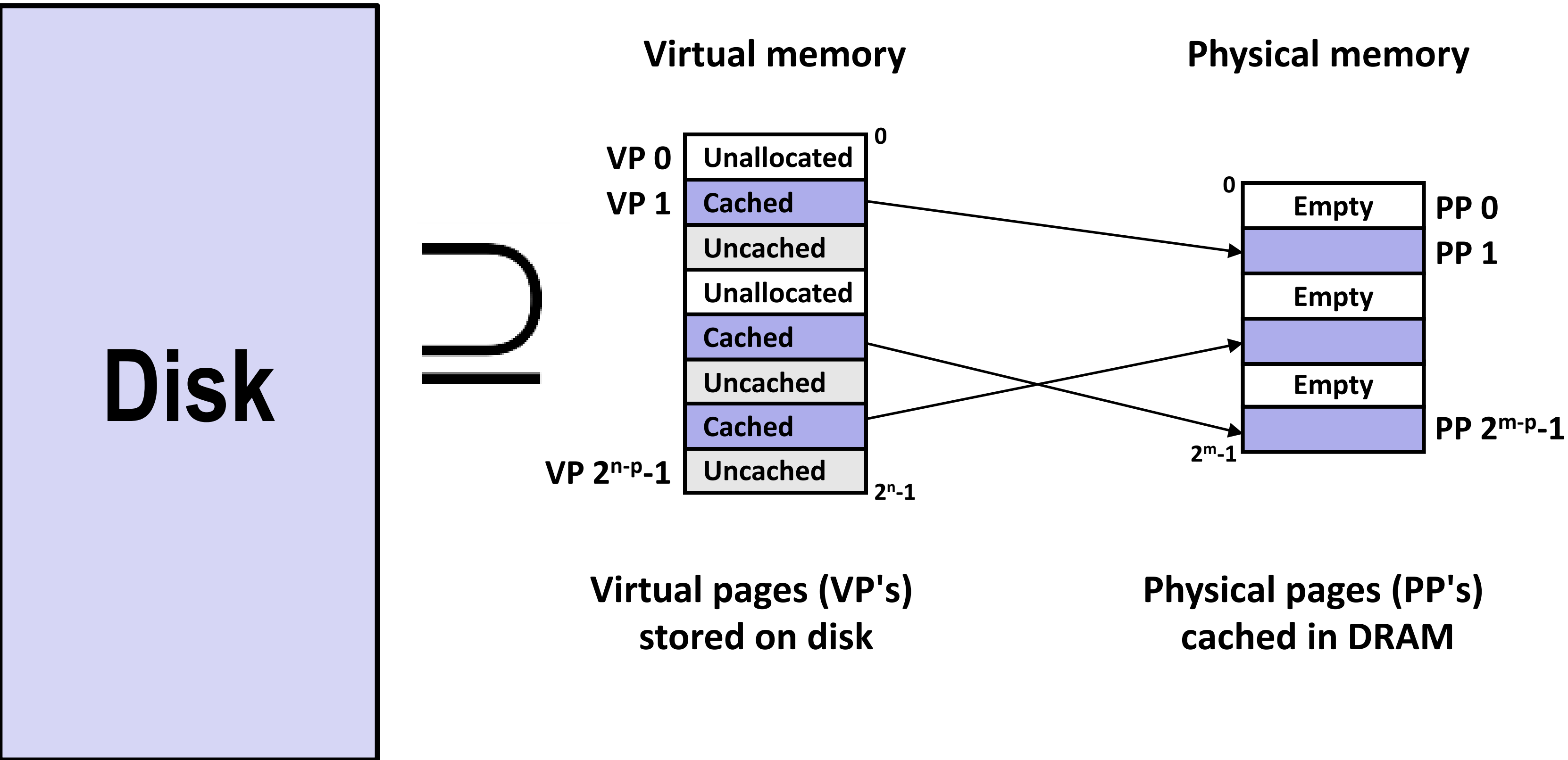
- **Efficient use of limited main memory (RAM)**
  - Use RAM as a cache for the parts of a virtual address space
    - some non-cached parts stored on disk
    - some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - transfer data back and 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 as a Tool for Caching

- *Virtual memory:* **array of N = 2^n contiguous bytes**
  - think of the array (allocated part) as being stored on disk
- **Physical main memory (DRAM) = cache for allocated virtual memory**
- **Blocks are called pages; size = 2^p**



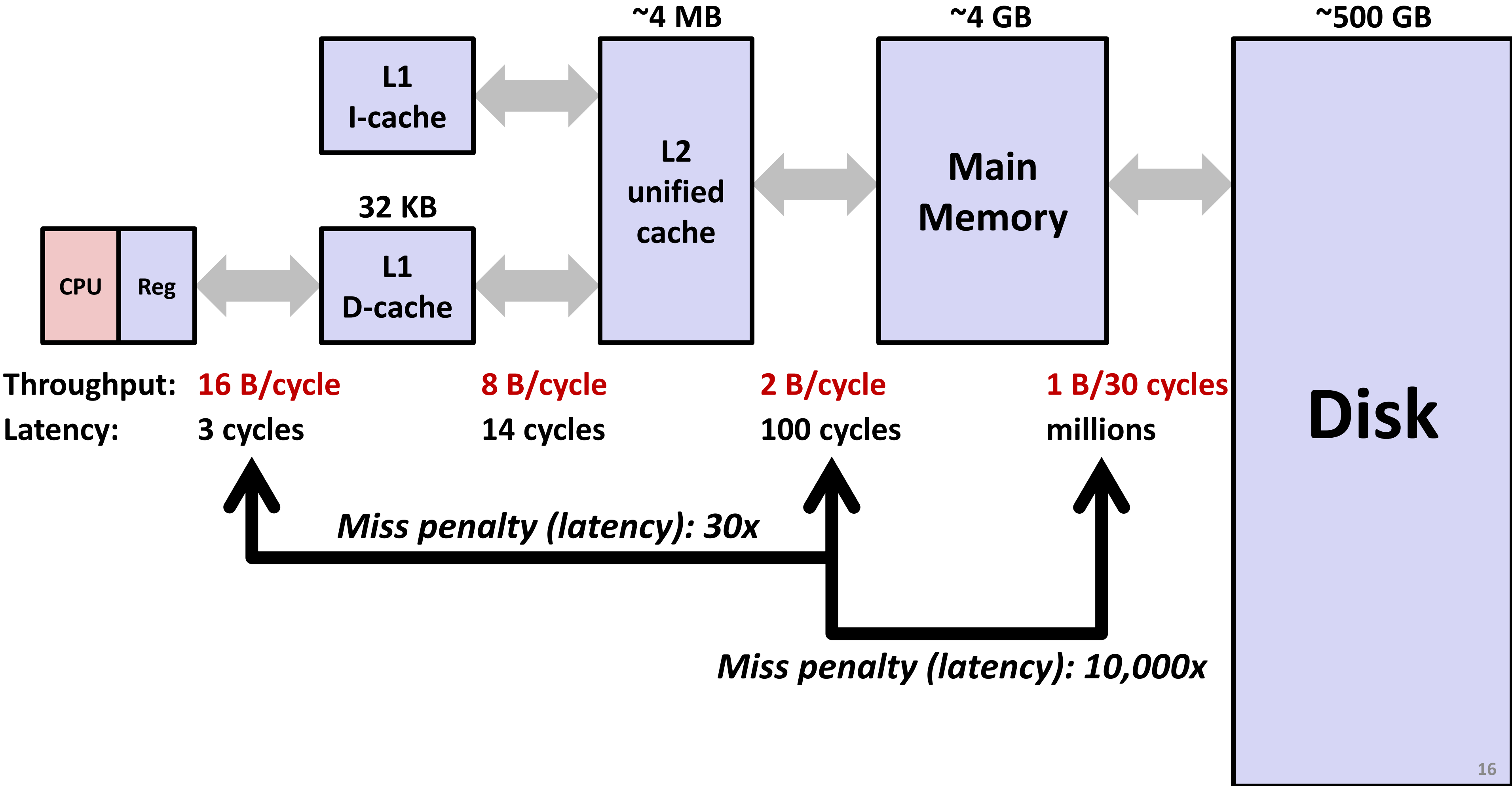
Virtual pages (VP's) stored on disk

Physical pages (PP's) cached in DRAM

Memory Hierarchy: Core 2 Duo

Not drawn to scale

L1/L2 cache: 64 B blocks



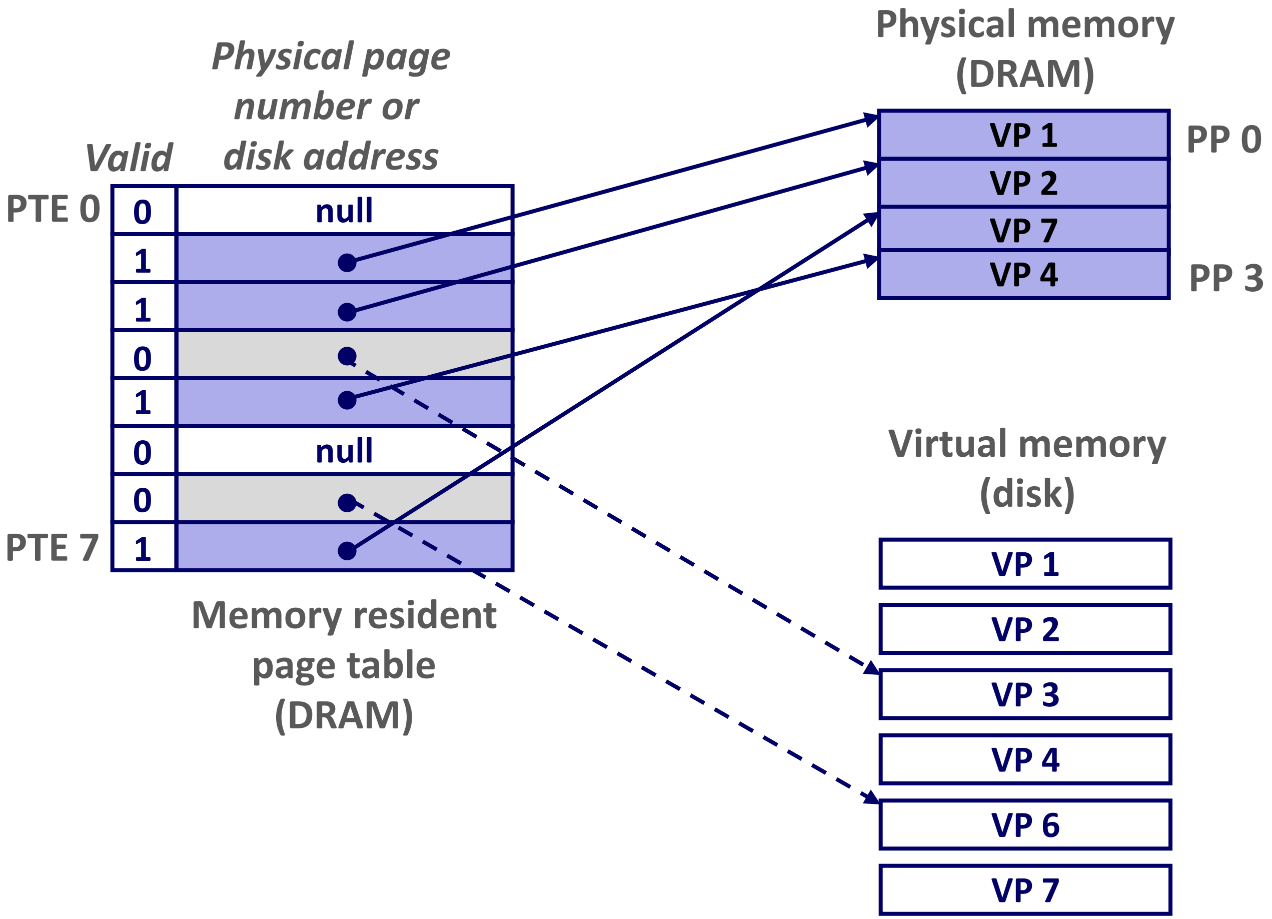
DRAM Cache Organization

- **DRAM cache organization driven by the enormous miss penalty**
  - DRAM is about ***10x*** slower than SRAM
  - Disk is about ***10,000x*** slower than DRAM
    - For first byte, faster for next byte

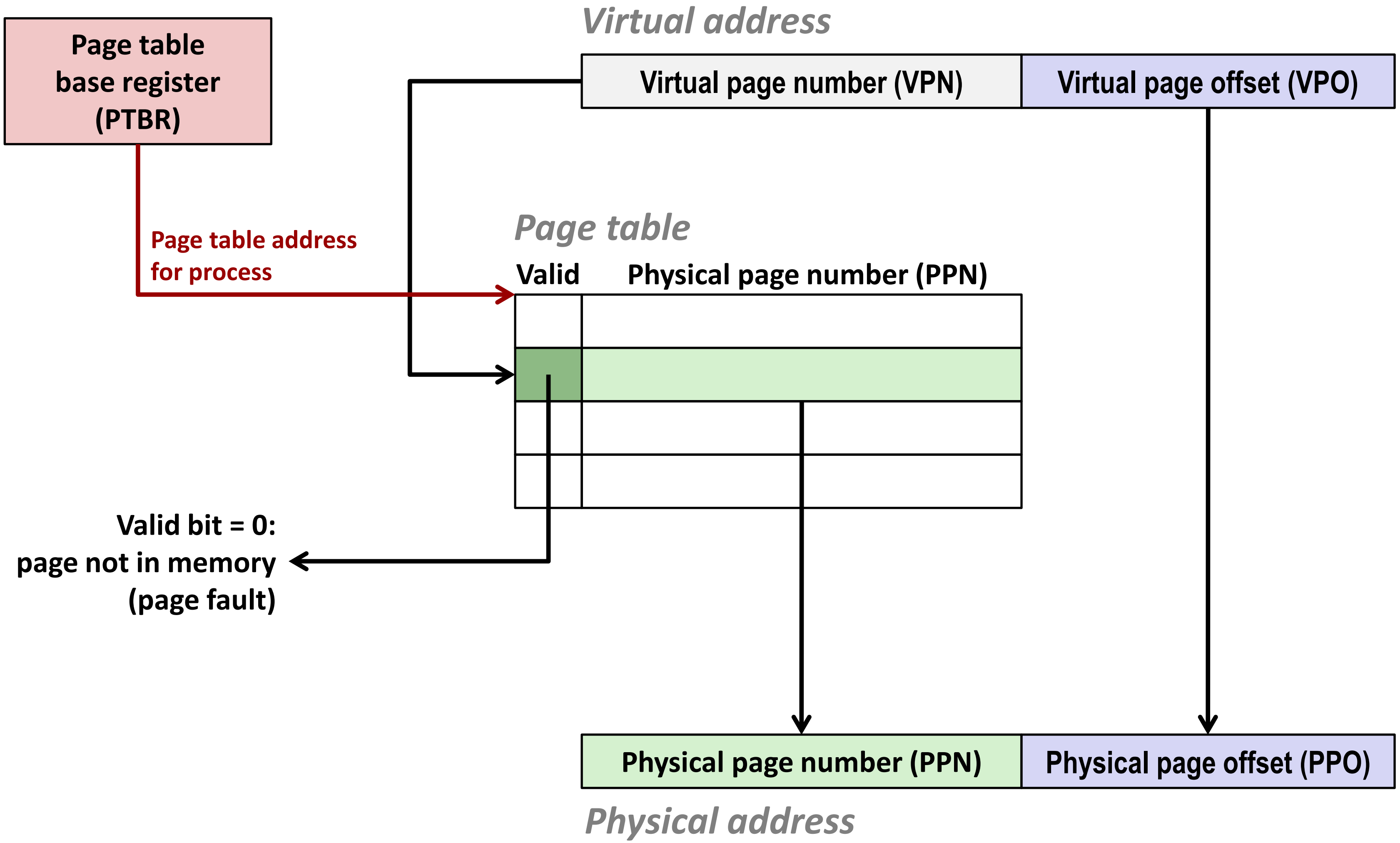
- **Consequences**
  - Large page (block) size: typically 4-8 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a "large" mapping function – different from CPU caches
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

Address Translation: Page Tables

- A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages. Here: 8 VPs
  - Per-process kernel data structure in DRAM

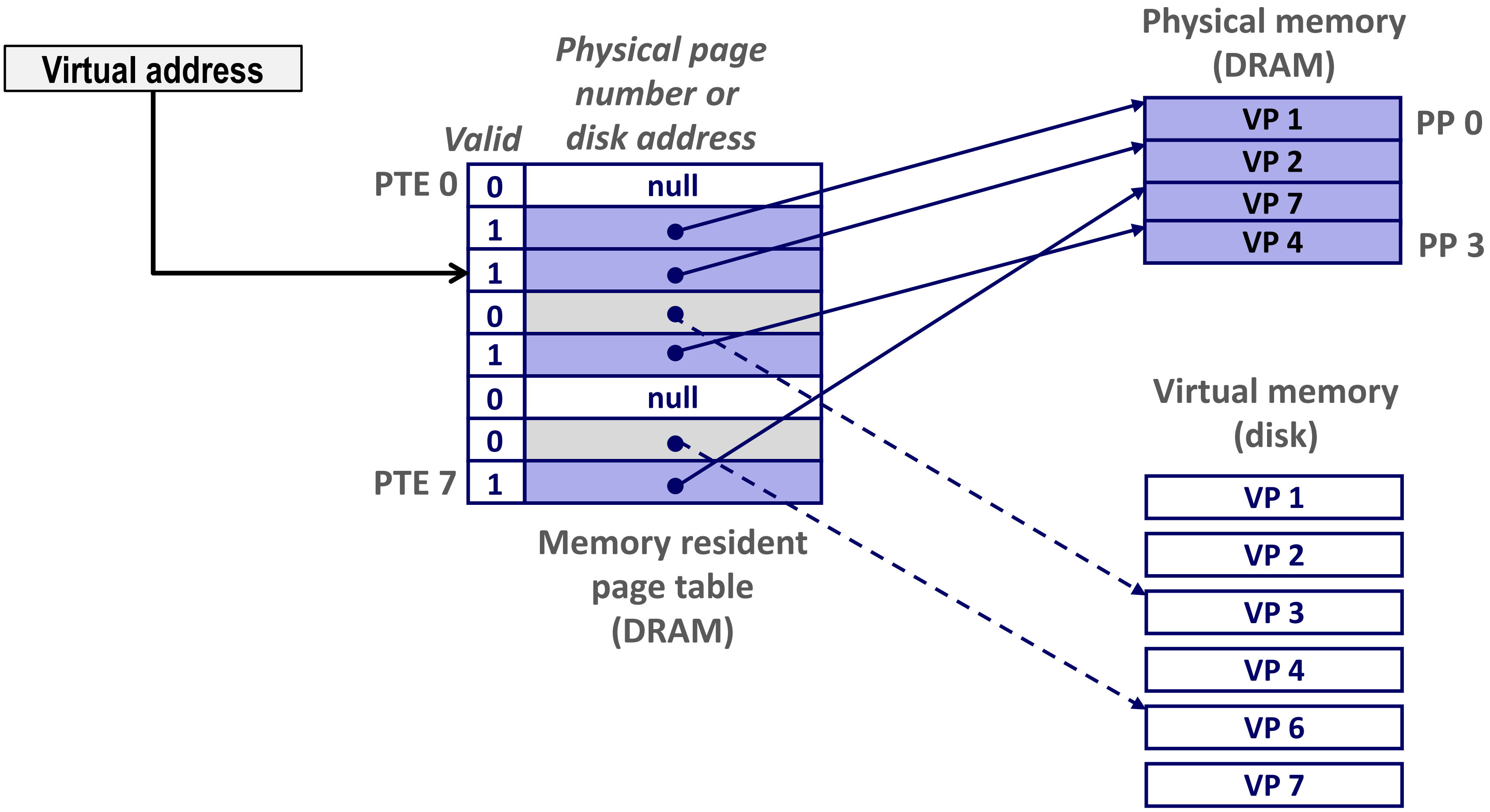


Address Translation With a Page Table


Page table base register (PTBR)
Virtual address
Virtual page number (VPN)
Virtual page offset (VPO)
Page table address for process
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

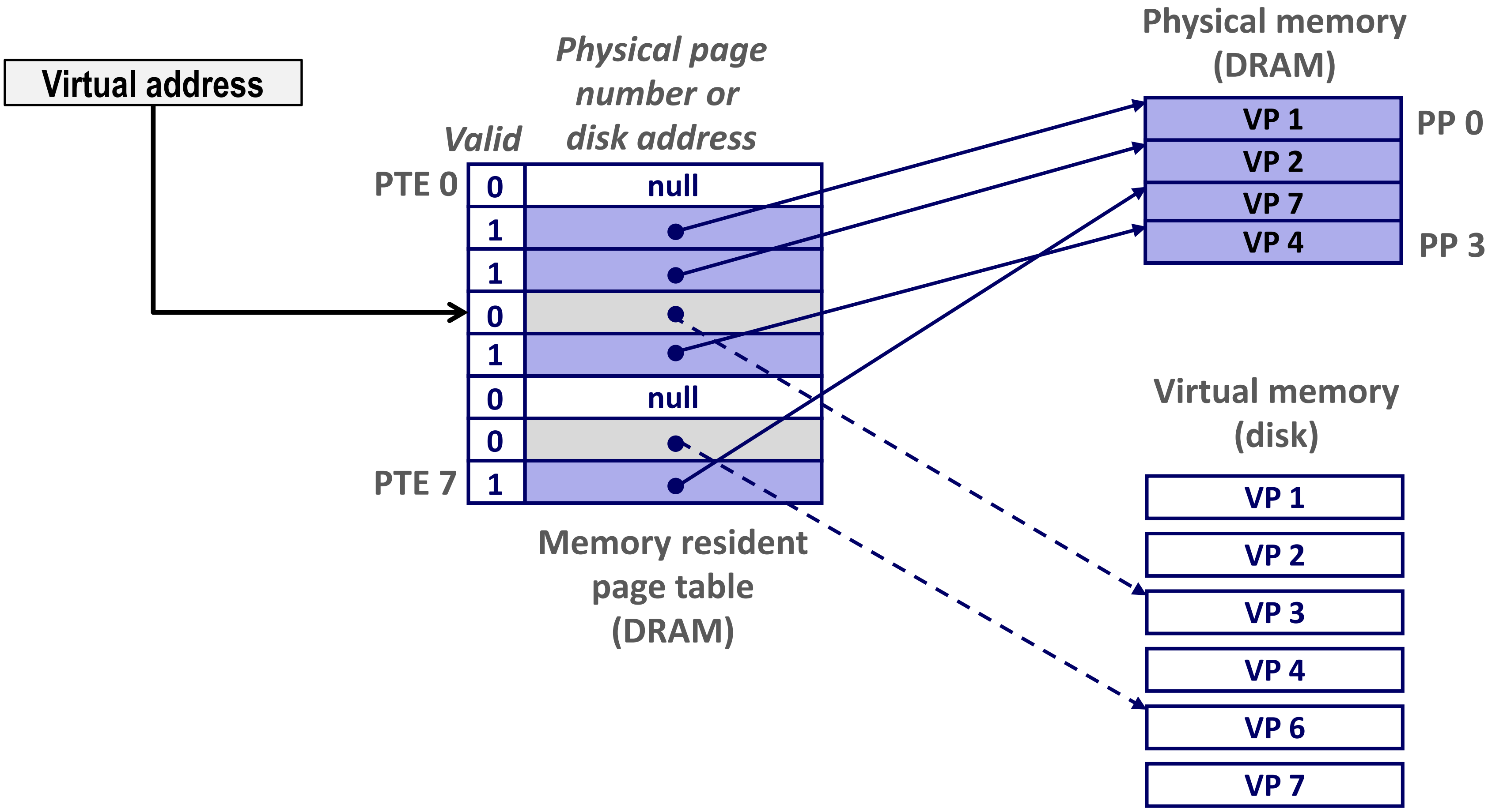
Page Hit

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



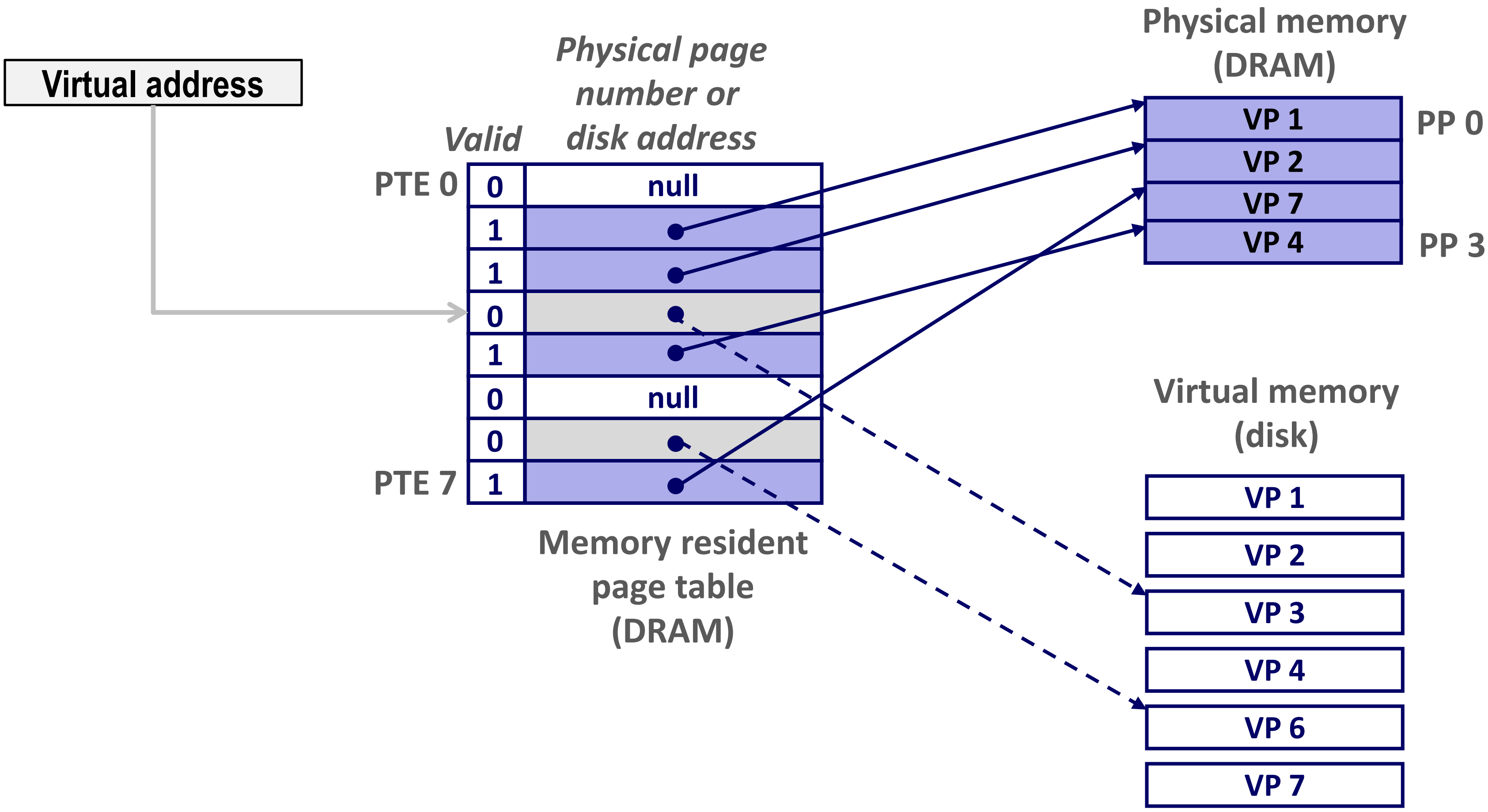
Page Miss

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



Handling Page Fault

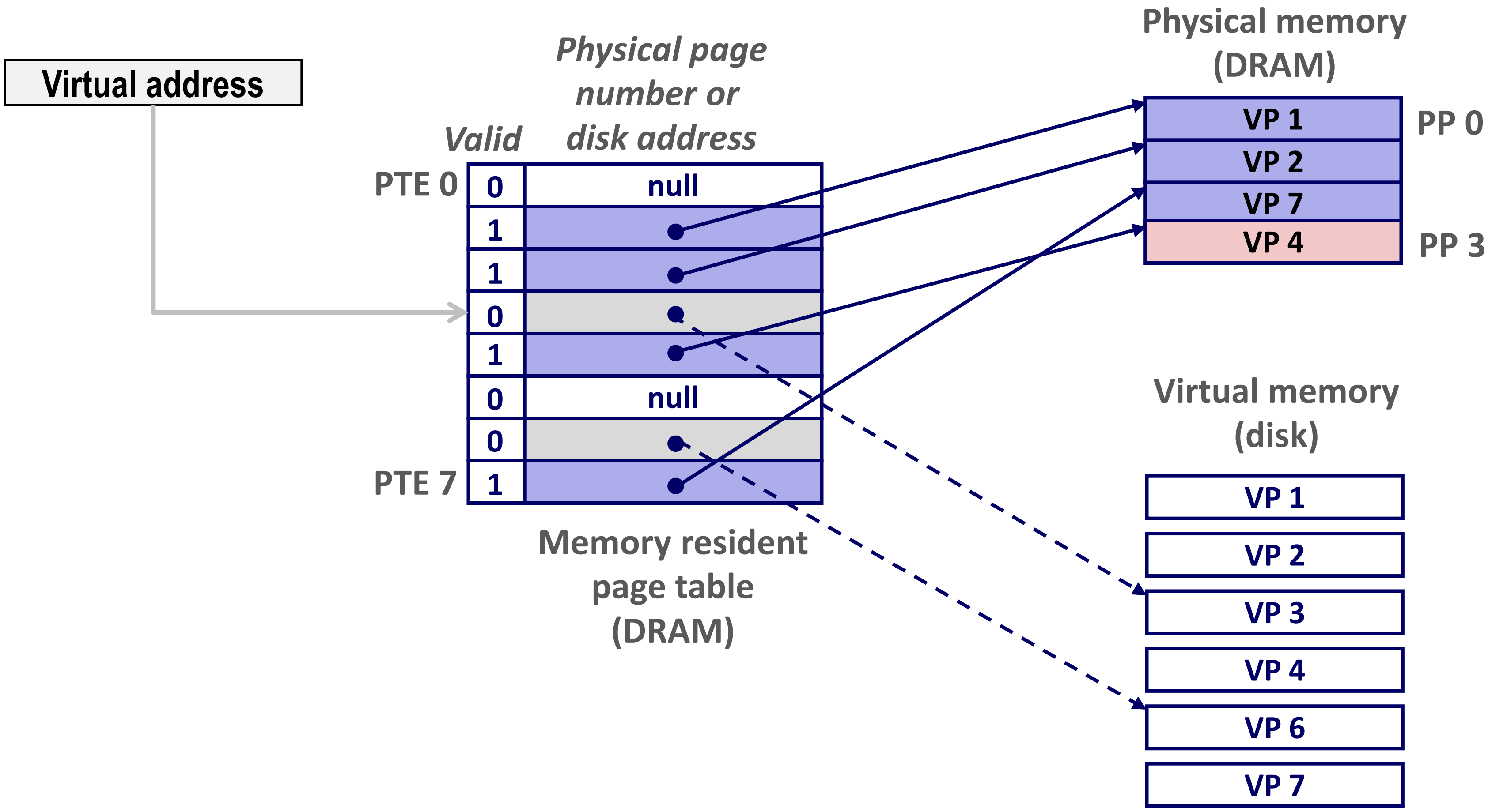
- Page miss causes page fault (an exception)



	Valid	Physical page number or disk address
PTE 0	0	null
	1	●
	1	●
	0	●
	1	●
	0	null
	0	●
PTE 7	1	●

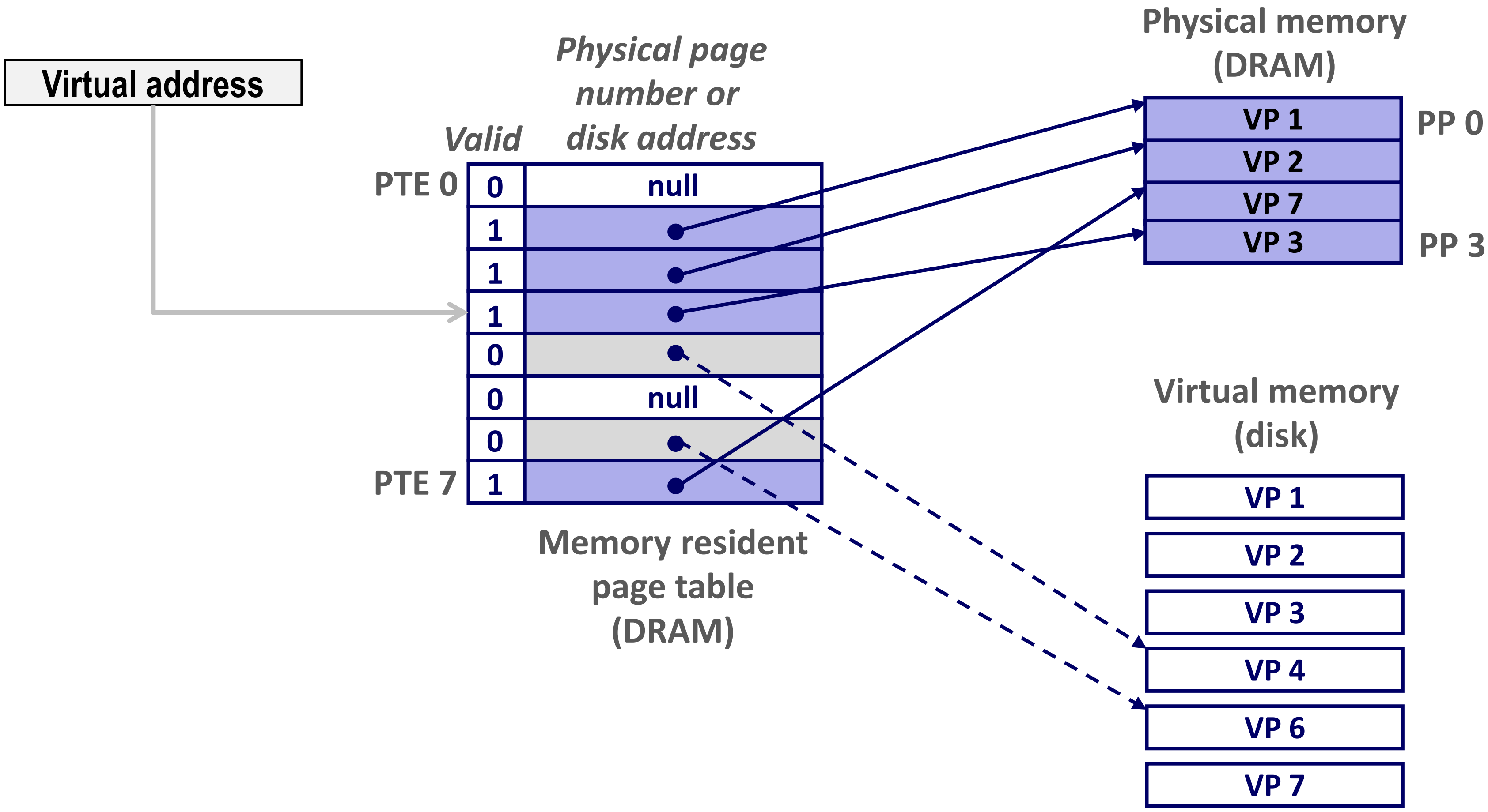
Handling Page Fault

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



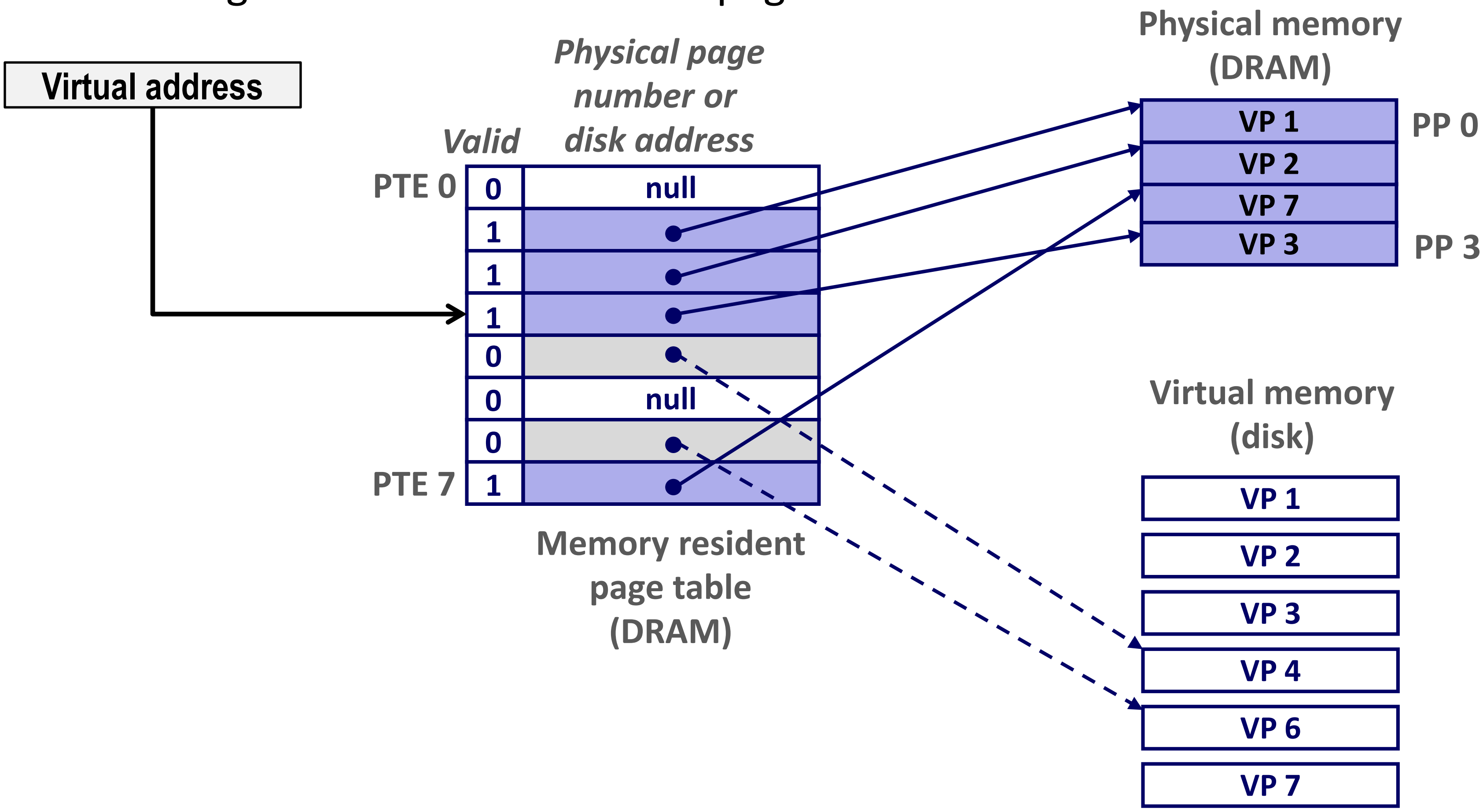
Handling Page Fault

- 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!

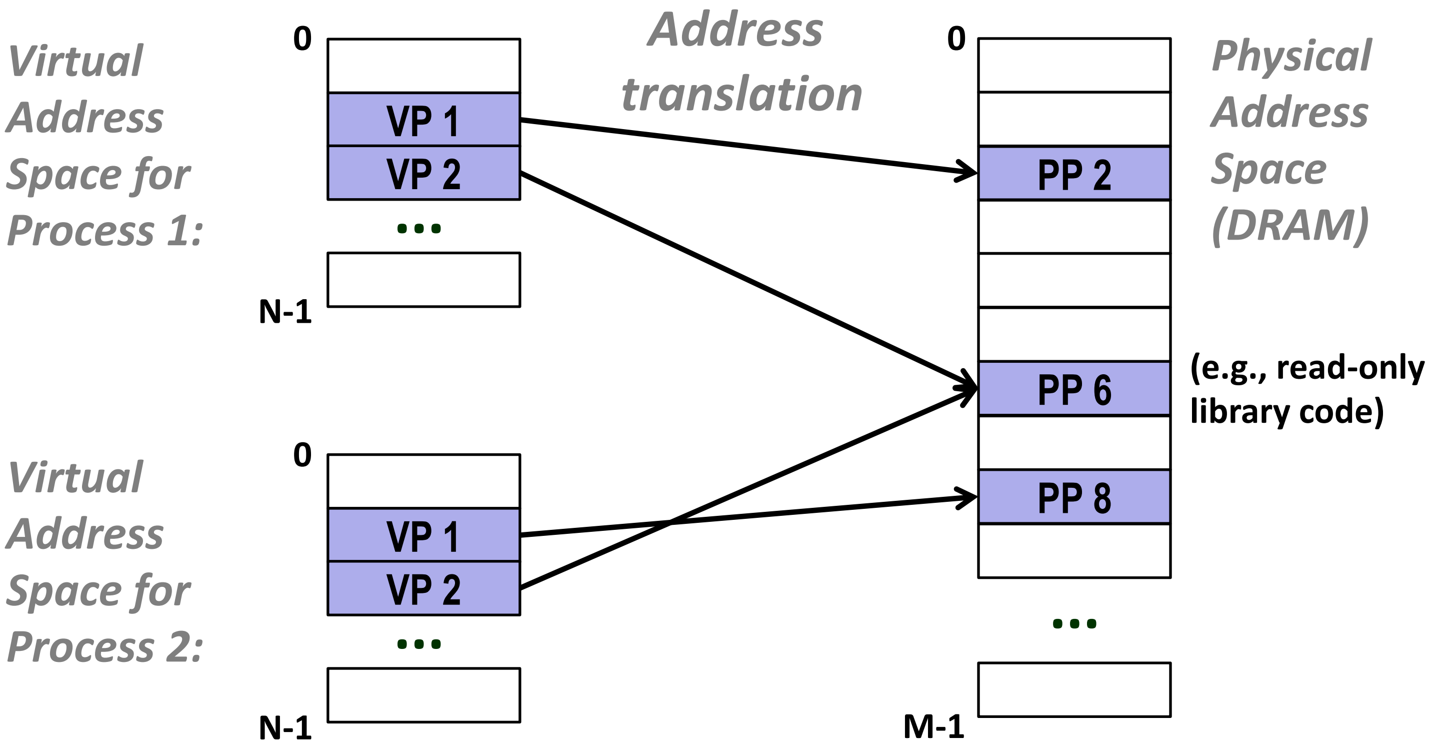


Why does it work? Locality

- **Virtual memory works because of locality**
- **At any point in time, programs tend to access a set of active virtual pages called the *working set***
  - Programs with better temporal locality will have smaller working sets
- **If (working set size < main memory size)**
  - Good performance for one process after compulsory misses
- **If (SUM(working set sizes) > main memory size)**
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

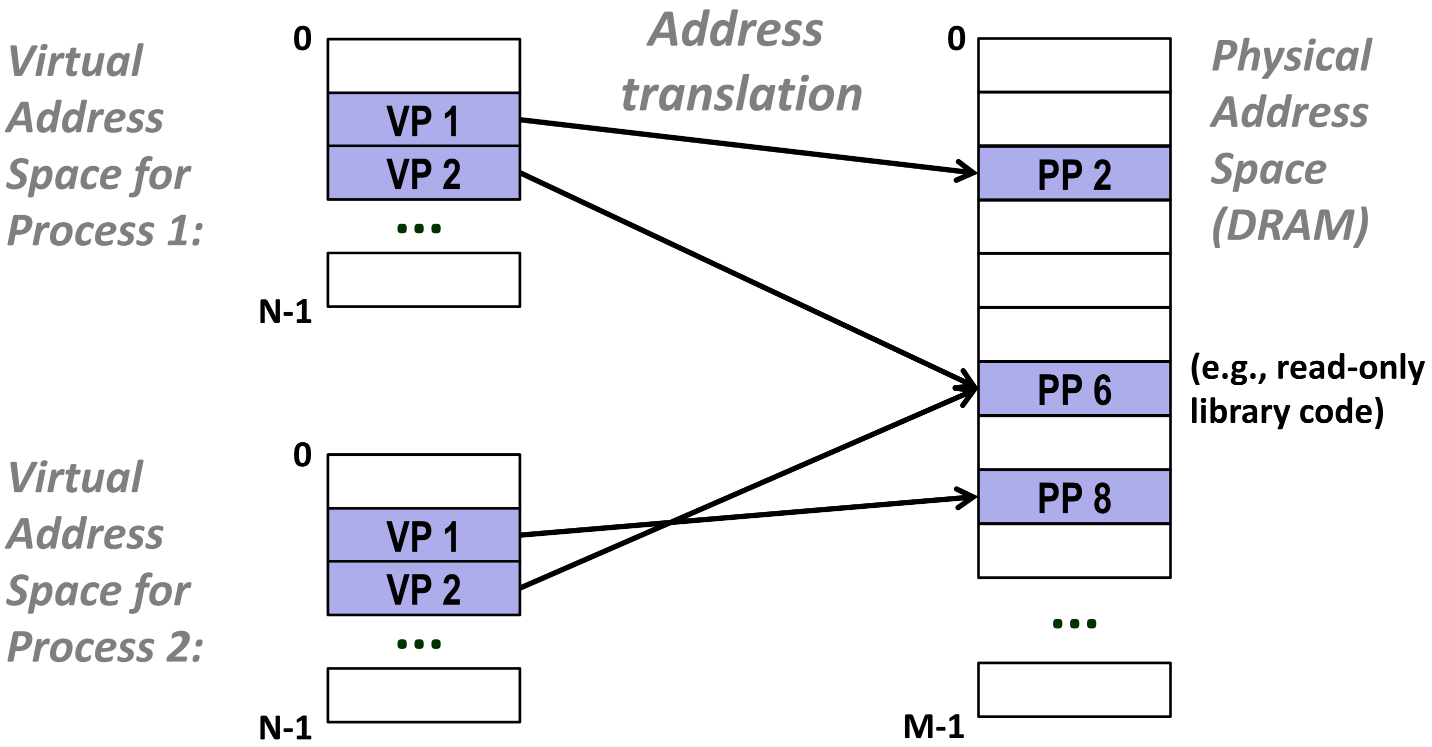
VM as a Tool for Memory Management

- **Key idea: each process has its own virtual address space**
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
    - Well chosen mappings simplify memory allocation and management


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

VM as a Tool for Memory Management

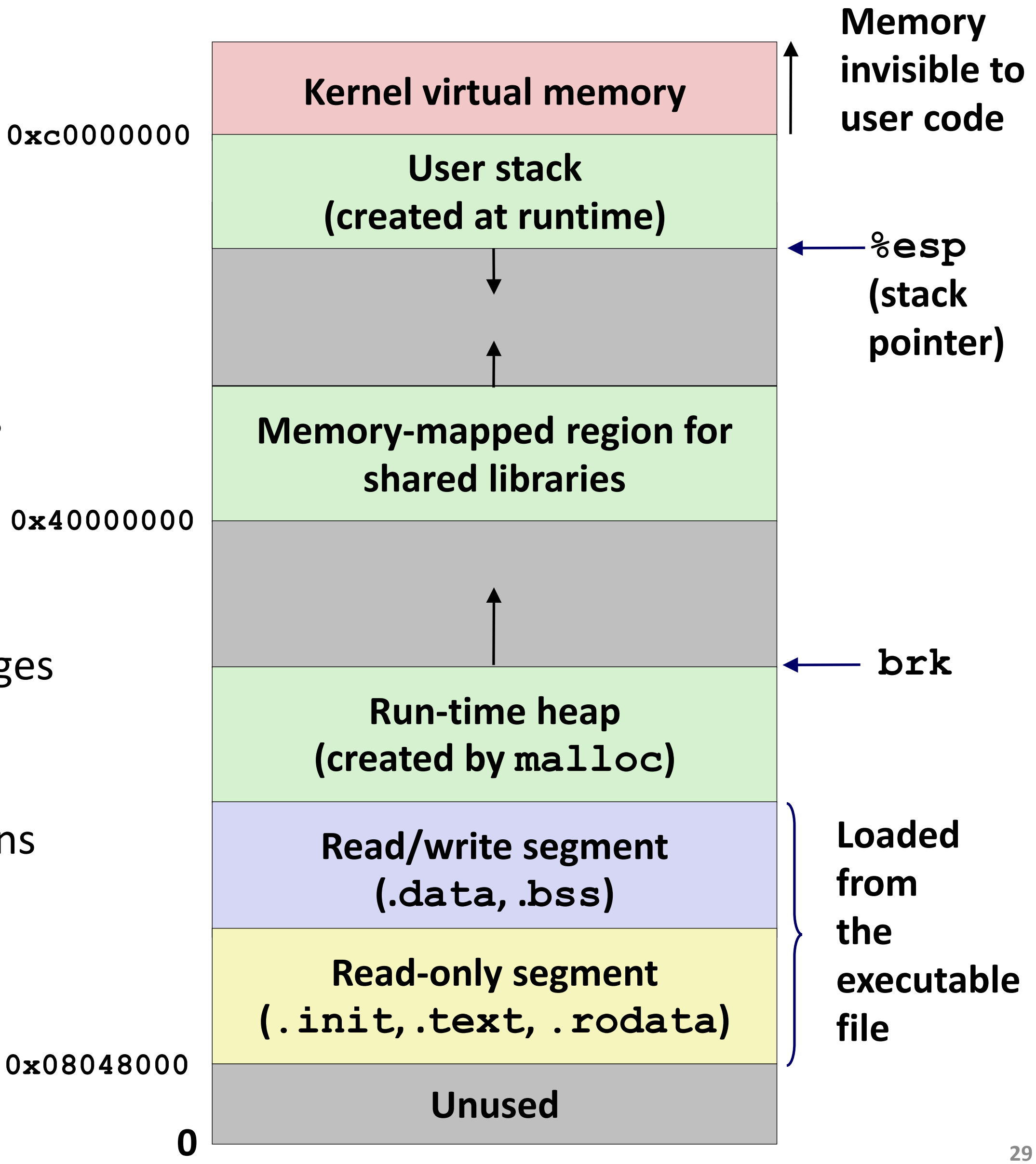
- **Memory allocation**
  - Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times
- **Sharing code and data among processes**
  - Map virtual pages to the same physical page (here: PP 6)


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

Simplifying Linking and Loading

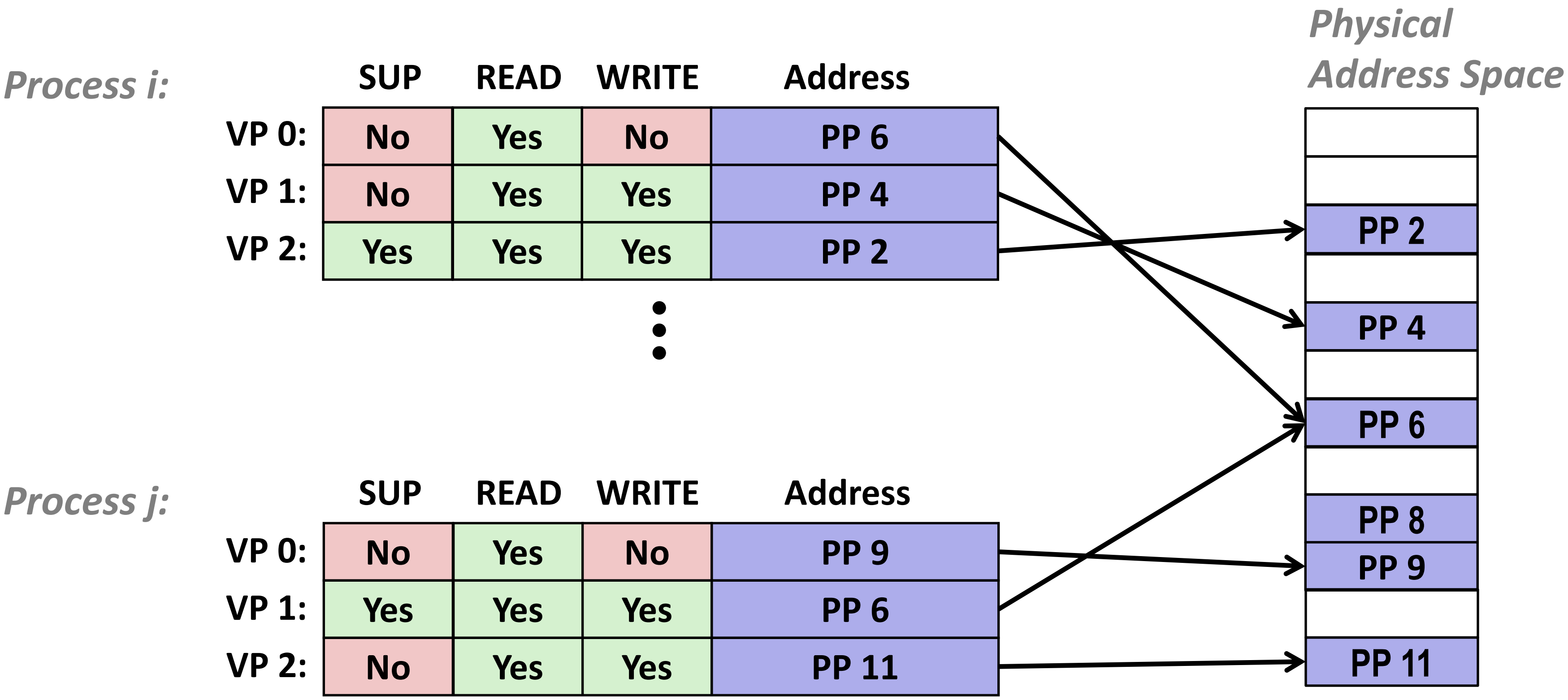
- **Linking**
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address

- **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



VM as a Tool for Memory Protection

- **Extend PTEs with permission bits**
- **Page fault handler checks these before remapping**
  - If violated, send process SIGSEGV signal (segmentation fault)



Process i:

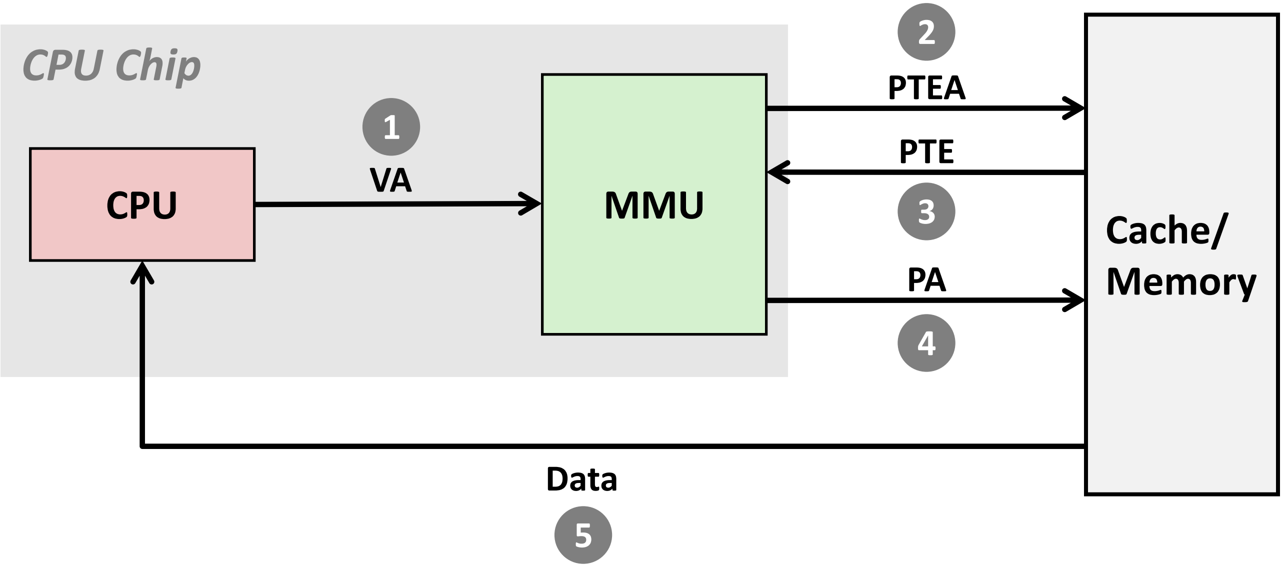
	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 6
VP 1:	No	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	PP 2

•
•
•

Process j:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 9
VP 1:	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	PP 11

Address Translation: Page Hit



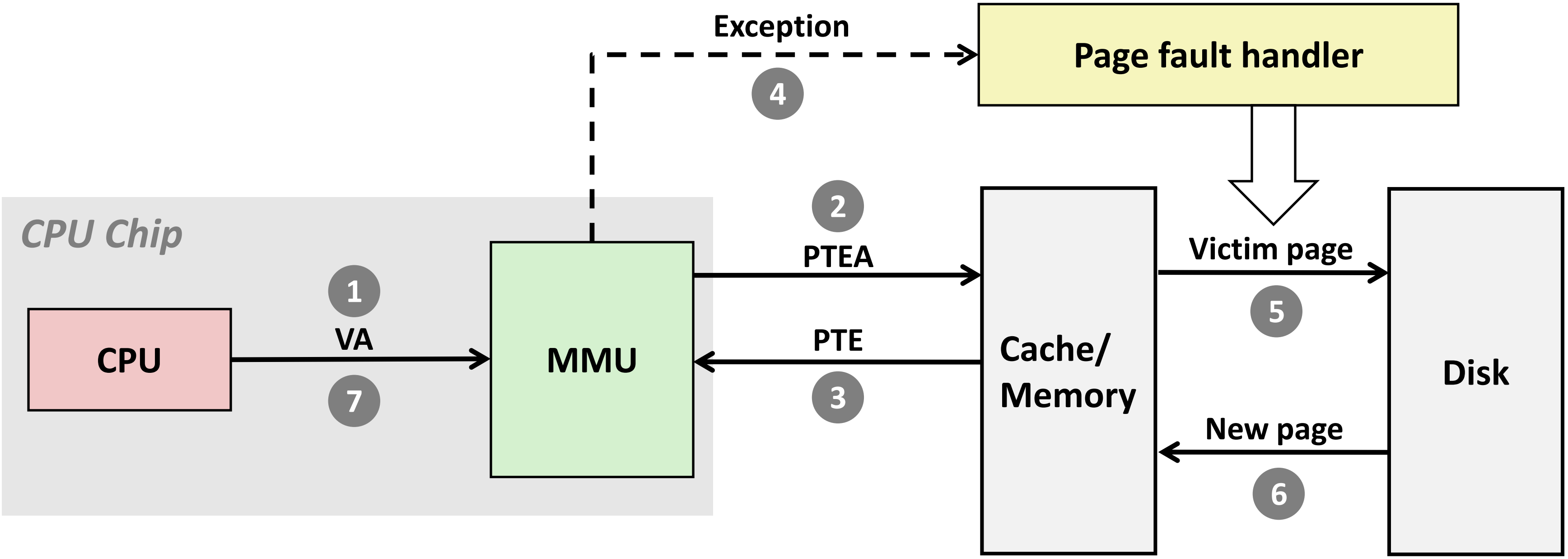
1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

Address Translation: Page Fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) Valid bit is zero, so MMU triggers page fault exception

5) Handler identifies victim (and, if dirty, pages it out to disk)

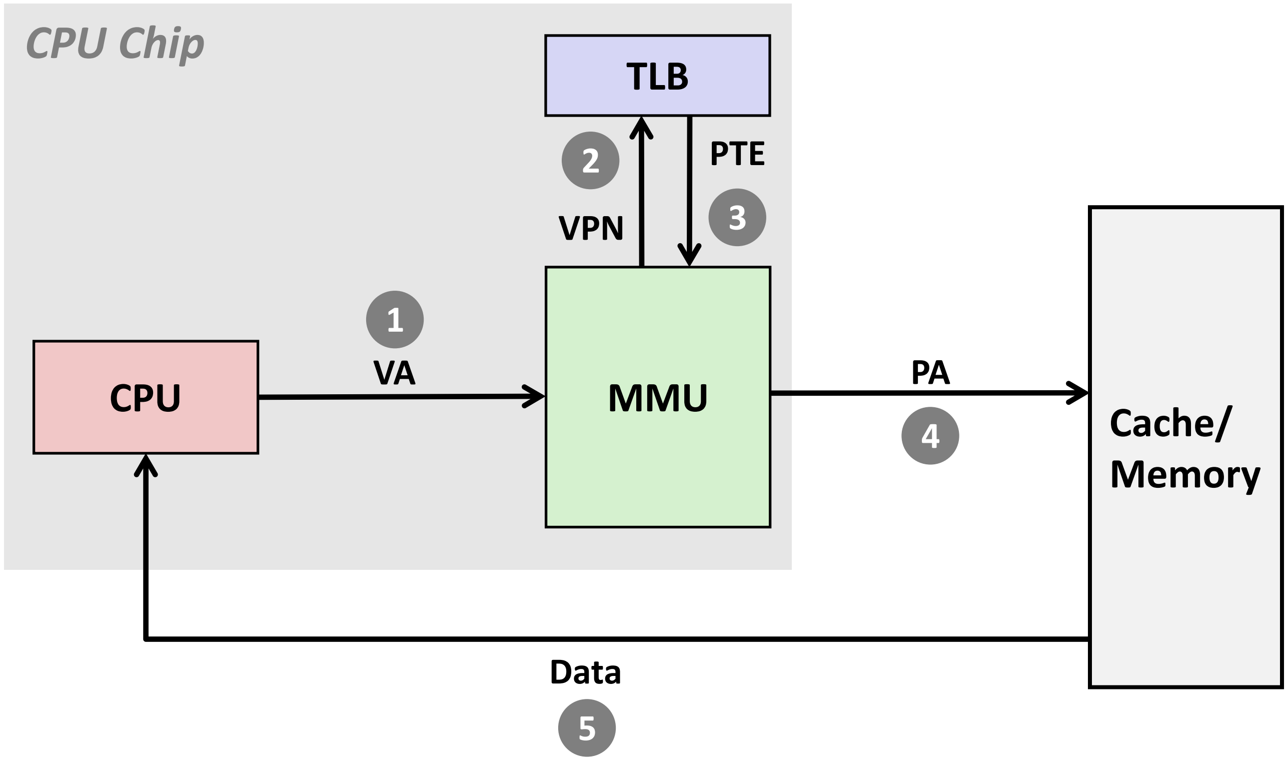
6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

Speeding up Translation with a TLB

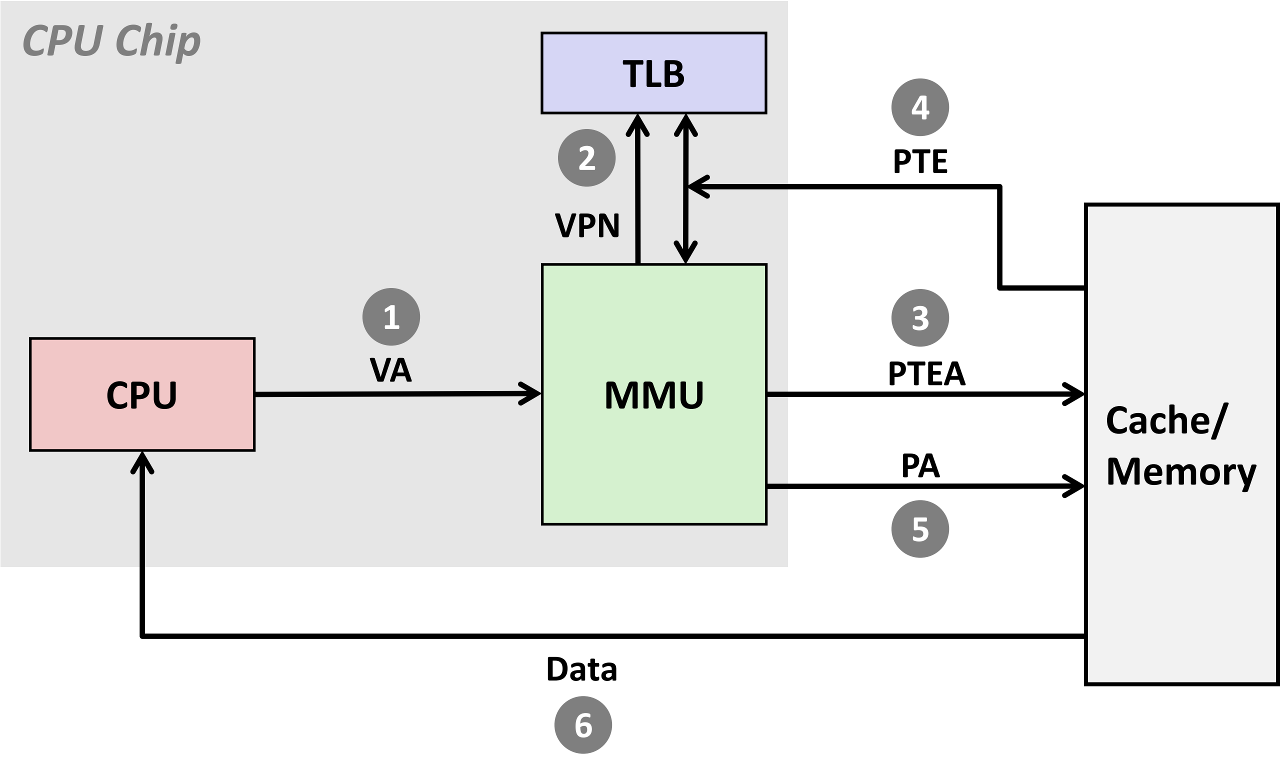
- **Page table entries (PTEs) are cached in L1 like any other memory word**
  - PTEs may be evicted by other data references
  - PTE hit still requires a 1-cycle delay
- **Solution: *Translation Lookaside Buffer* (TLB)**
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

TLB Hit



A TLB hit eliminates a memory access

TLB Miss

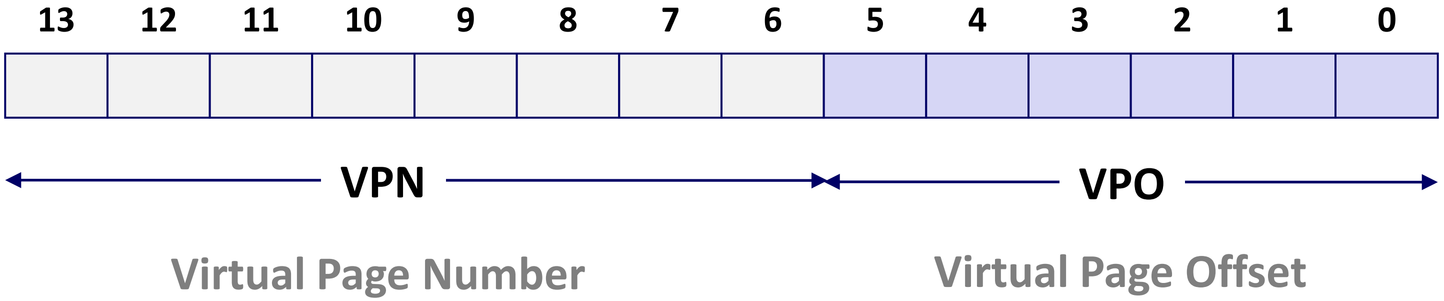


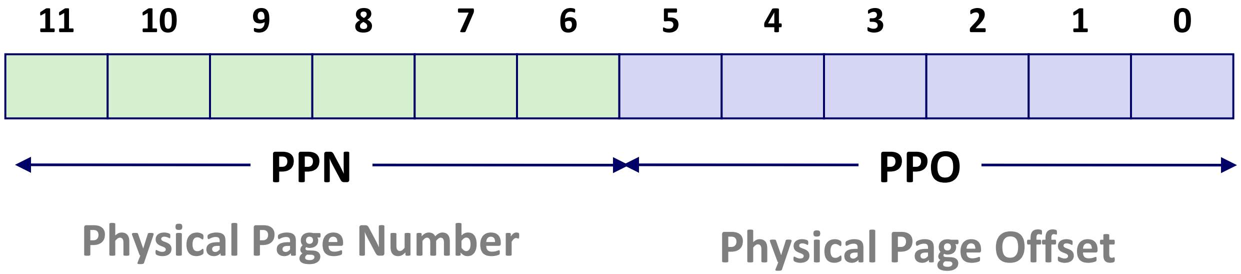
A TLB miss incurs an add'l memory access (the PTE)

Fortunately, TLB misses are rare

Simple Memory System Example

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





Simple Memory System Page Table

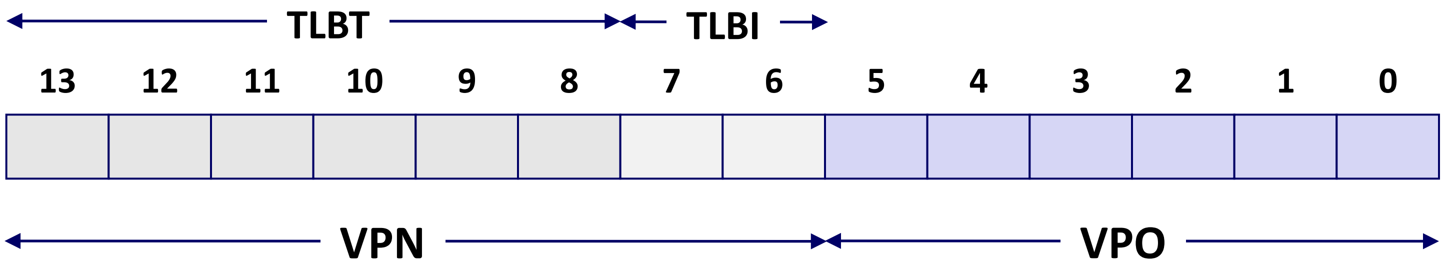
Only show first 16 entries (out of 256)

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

Simple Memory System TLB

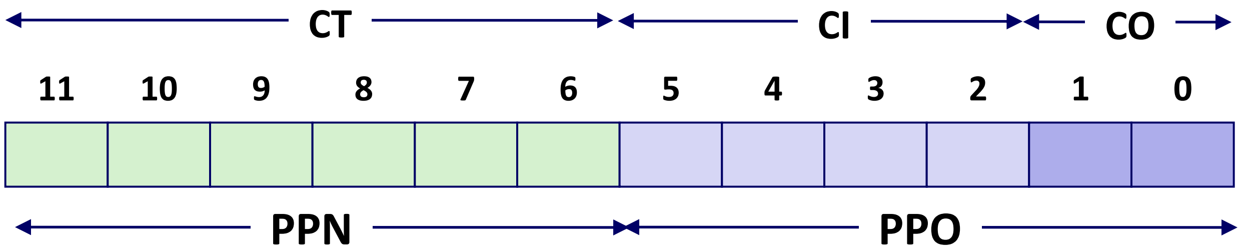
- 16 entries
- 4-way associative



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 Cache

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



Idx	*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

Idx	*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	–	–	–	–

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

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

Page table

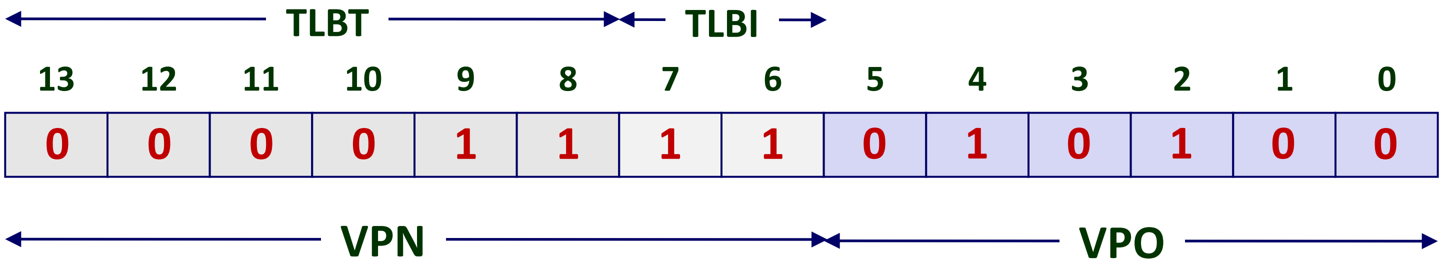
Cache

Idx	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

Idx	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	–	–	–	–

Address Translation Example #1

Virtual Address: 0x03D4

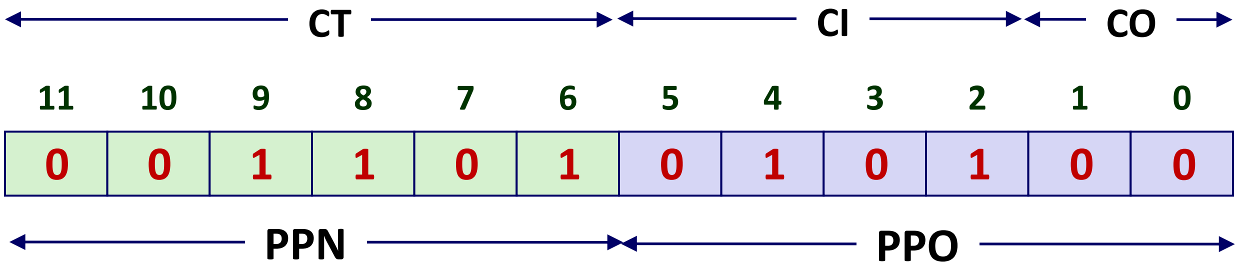

TLBT ← bits 13–8 → TLBI ← bits 7–6 →

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

Physical Address


CT ← bits 11–6 → CI ← bits 5–2 → CO ← bits 1–0 →

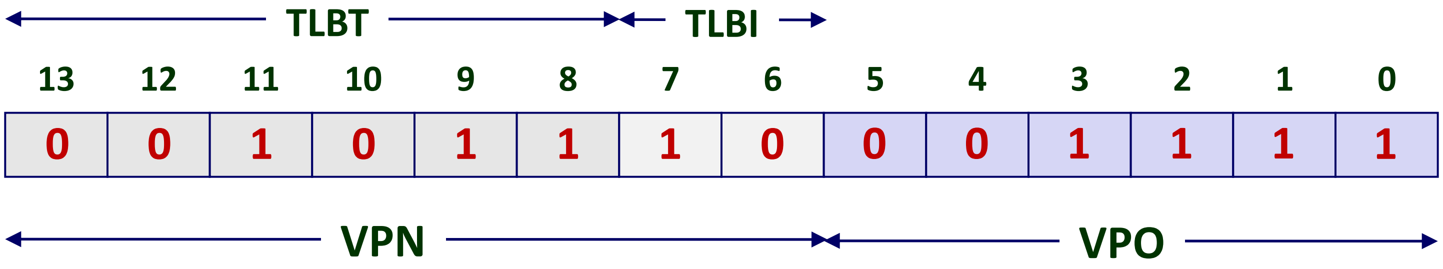
11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0

PPN ← bits 11–6 → PPO ← bits 5–0 →

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

Address Translation Example #2

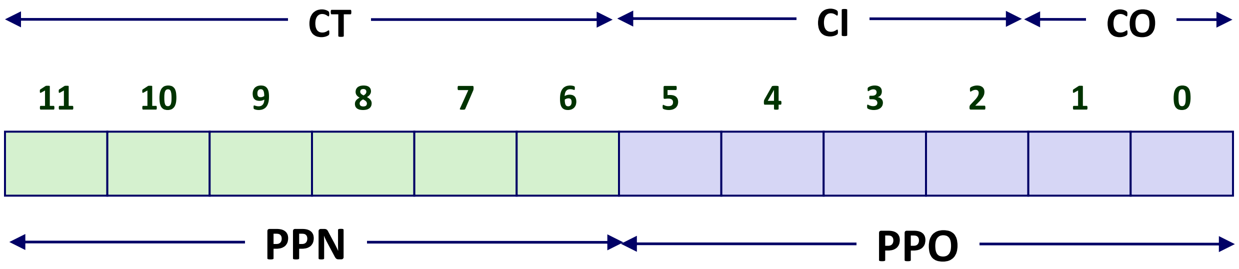
Virtual Address: 0x0B8F



TLBT						TLBI							
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								VPO					

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

Physical Address

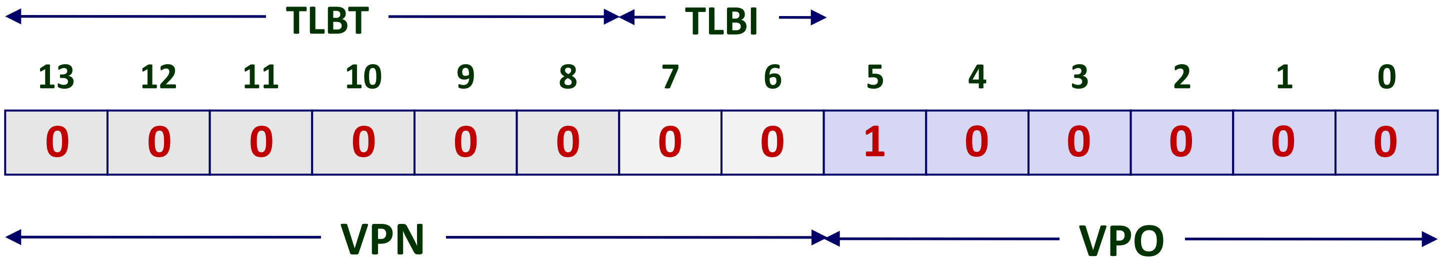


CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
PPN						PPO					

CO ___ CI___ CT _____ Hit? __ Byte: _____

Address Translation Example #3

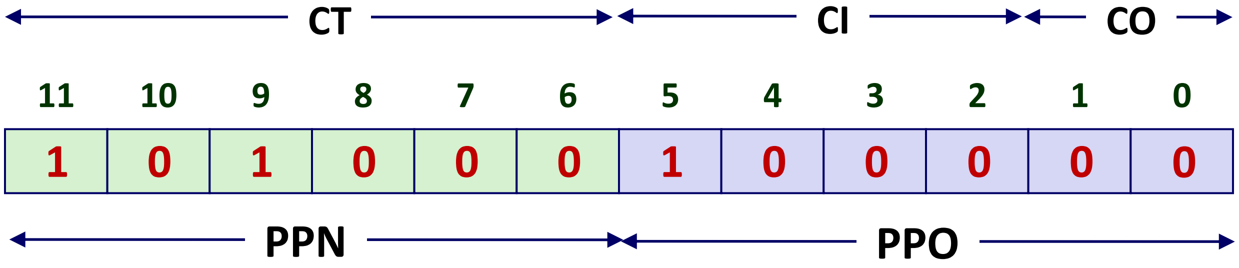
Virtual Address: 0x0020



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

Physical Address



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

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

Servicing a Page Fault

(1) Processor signals disk controller

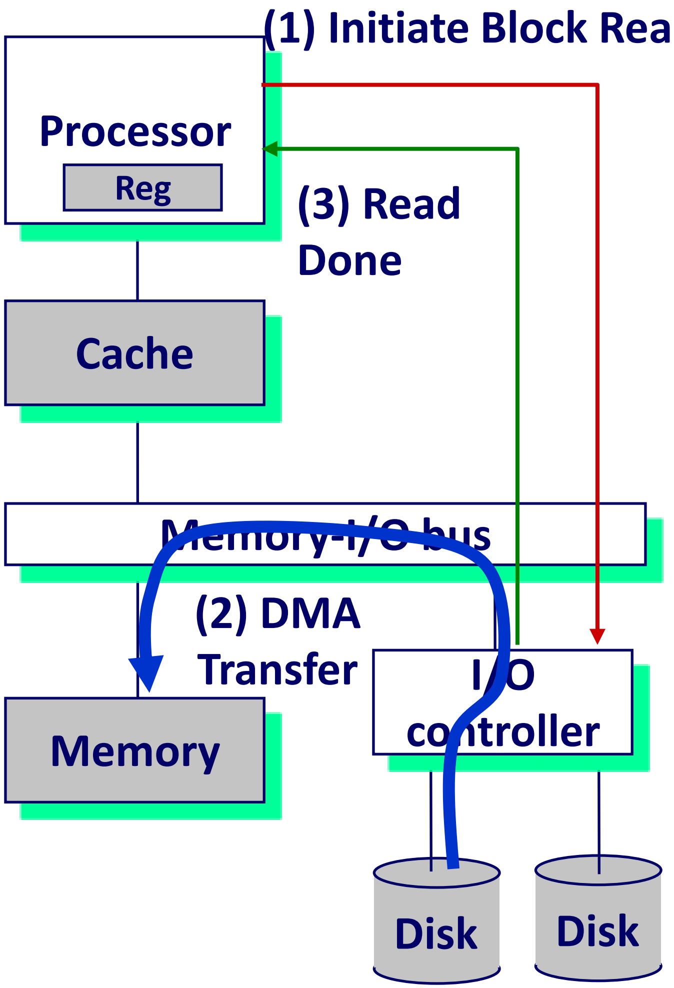
- Read block of length P starting at disk address X and store starting at memory address Y

(2) Read occurs

- Direct Memory Access (DMA)
- Under control of I/O controller

(3) Controller signals completion

- Interrupts processor
- OS resumes suspended process



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 programming
  - 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
  - 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)