The Hardware/Software Interface

CSE351 Winter 2013

Virtual Memory I

Roadmap

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

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();
```

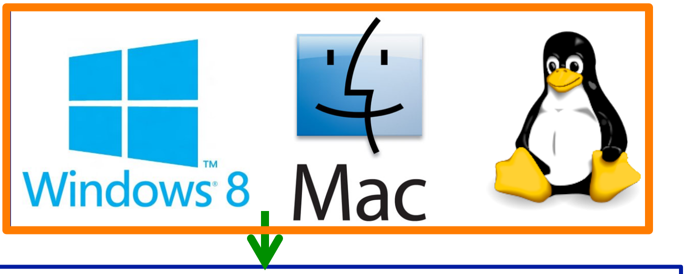
Assembly language:

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

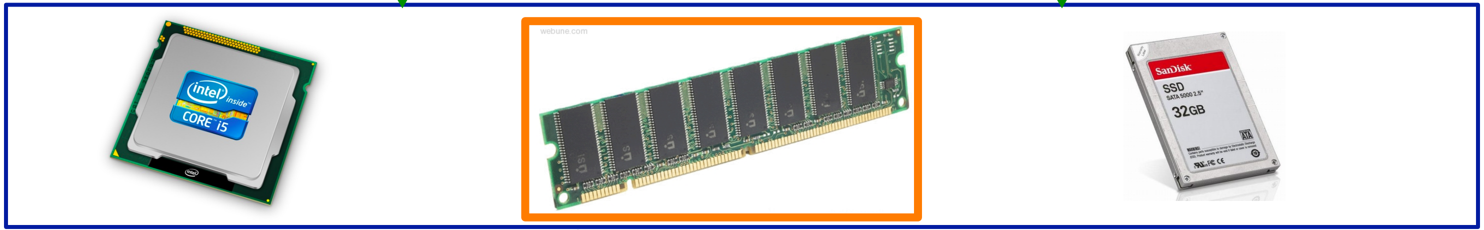
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



Virtual Memory (VM)

- Overview and motivation
- VM as tool for caching
- Address translation
- VM as tool for memory management
- VM as tool for memory protection

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”

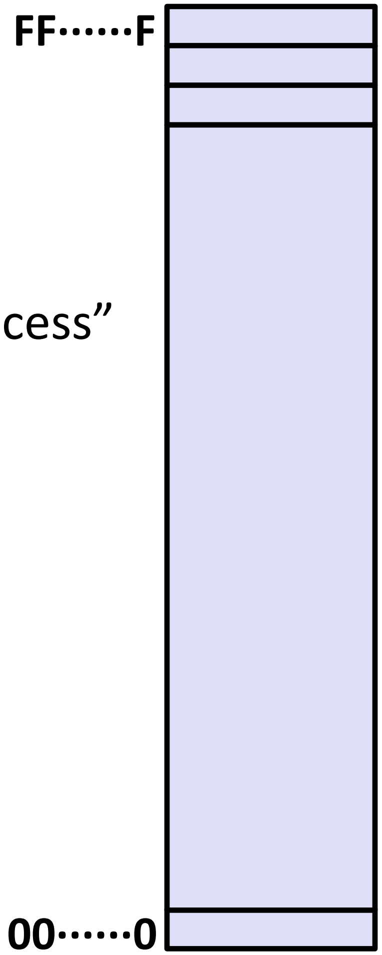
- **Process provides each program with two key abstractions:**
  - Logical control flow
    - Each process seems to have exclusive use of the CPU
  - Private virtual address space
    - Each process seems to have exclusive use of main memory

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

Virtual Memory (Previous Lectures)

- **Programs refer to virtual memory addresses**
  - `movl (%ecx),%eax`
  - Conceptually memory is just a very large array of bytes
  - Each byte has its own address
  - 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

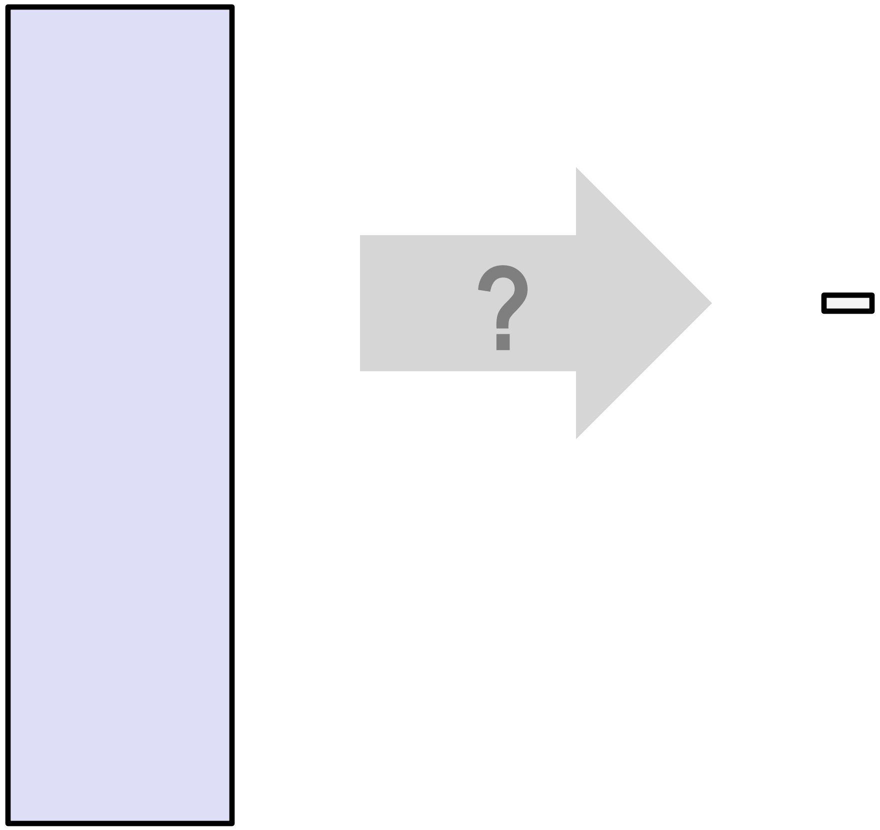
- ***What problems does virtual memory solve?***



Problem 1: How Does Everything Fit?

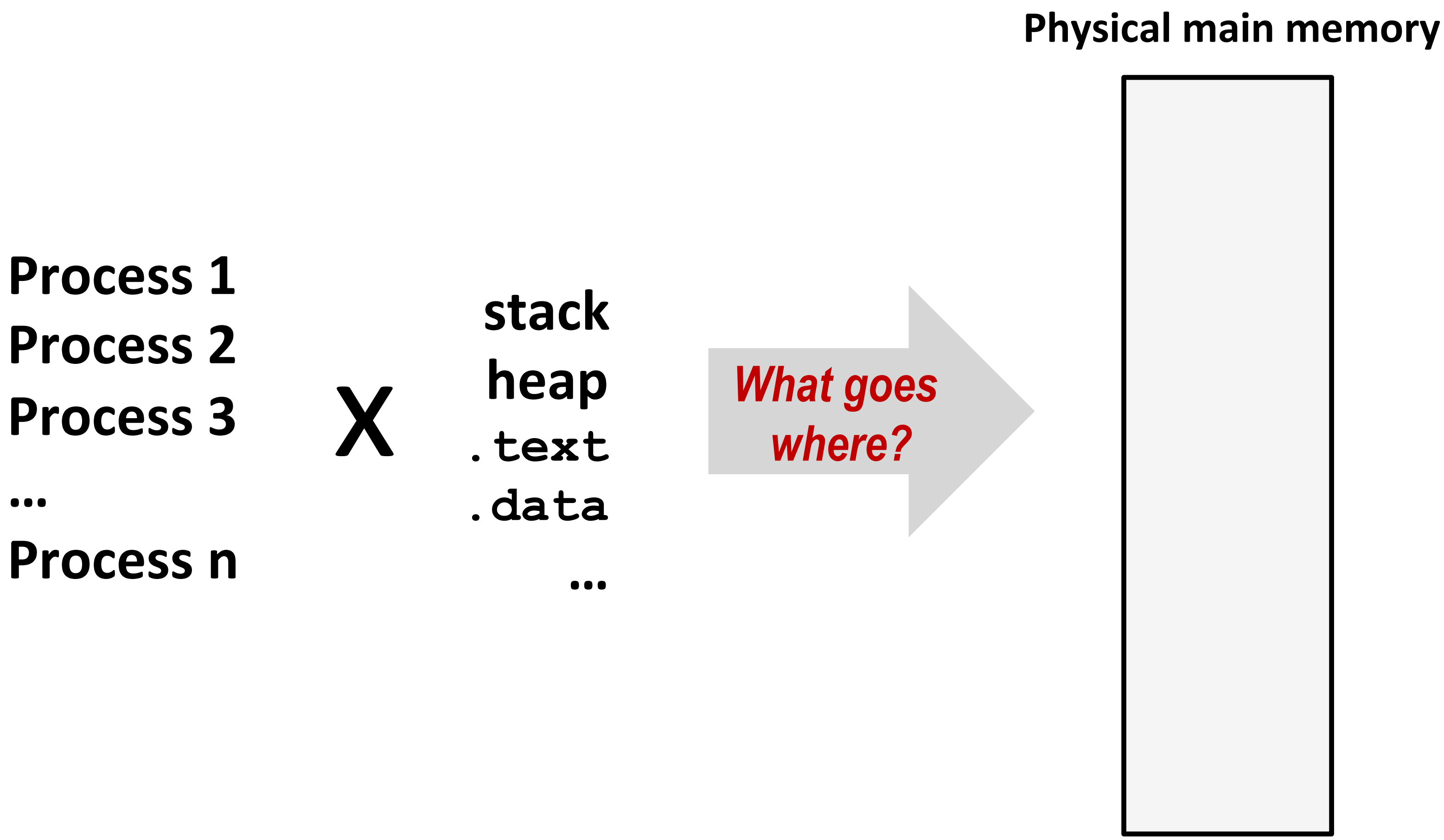
64-bit addresses:
16 Exabyte

Physical main memory:
Few Gigabytes



And there are many processes ….

Problem 2: Memory Management



Physical main memory

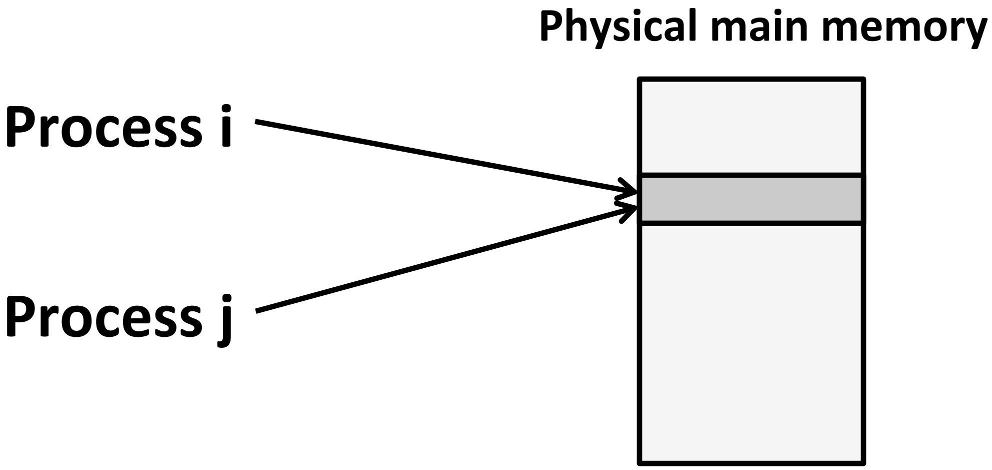
Process 1
Process 2
Process 3
…
Process n

X

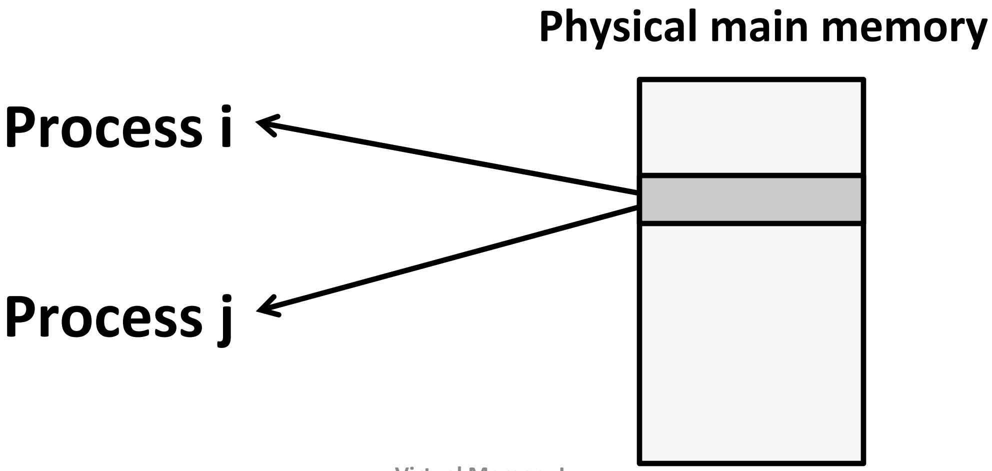
stack
heap
`.text`
`.data`
…

What goes where?

Problem 3: How To Protect



Problem 4: How To Share?

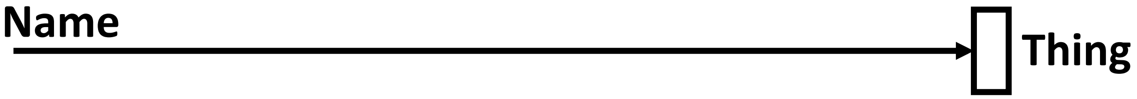


How would you solve those problems?

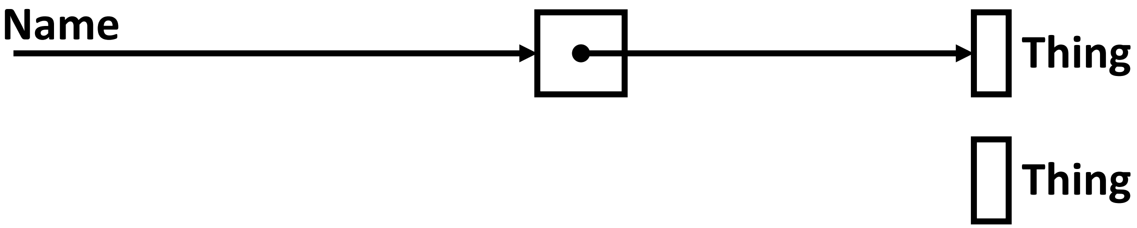
Indirection

- "Any problem in computer science can be solved by adding another level of indirection"

- Without Indirection



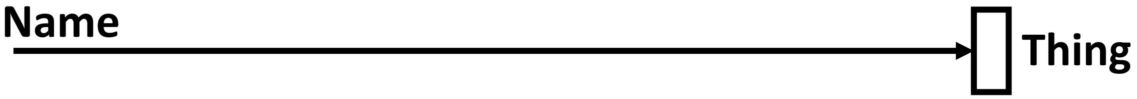
- With Indirection



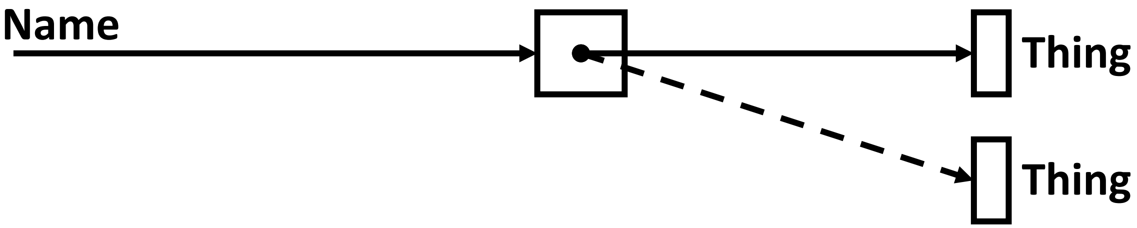
Indirection

- 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.

- Without Indirection

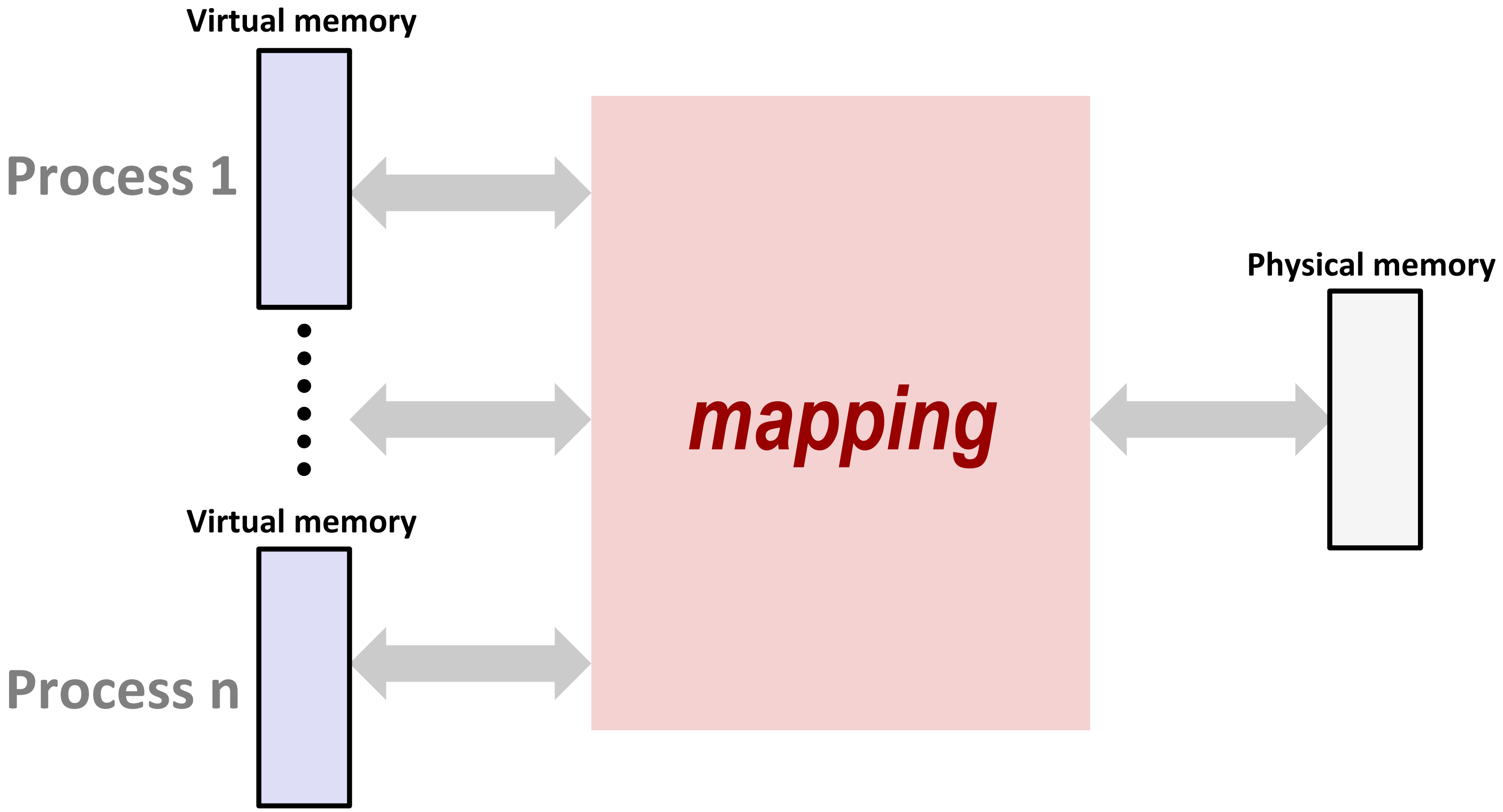


- With Indirection



- Examples:
  Domain Name Service (DNS) name->IP address, phone system (e.g., cell phone number portability), snail mail (e.g., mail forwarding), 911 (routed to local office), DHCP, call centers that route calls to available operators, etc.

Solution: Level Of Indirection



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

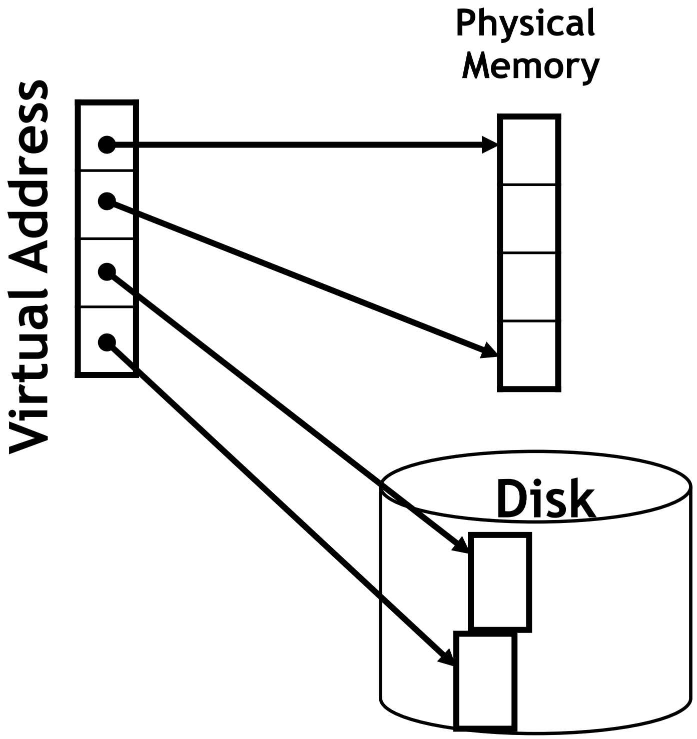
Address Spaces

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

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

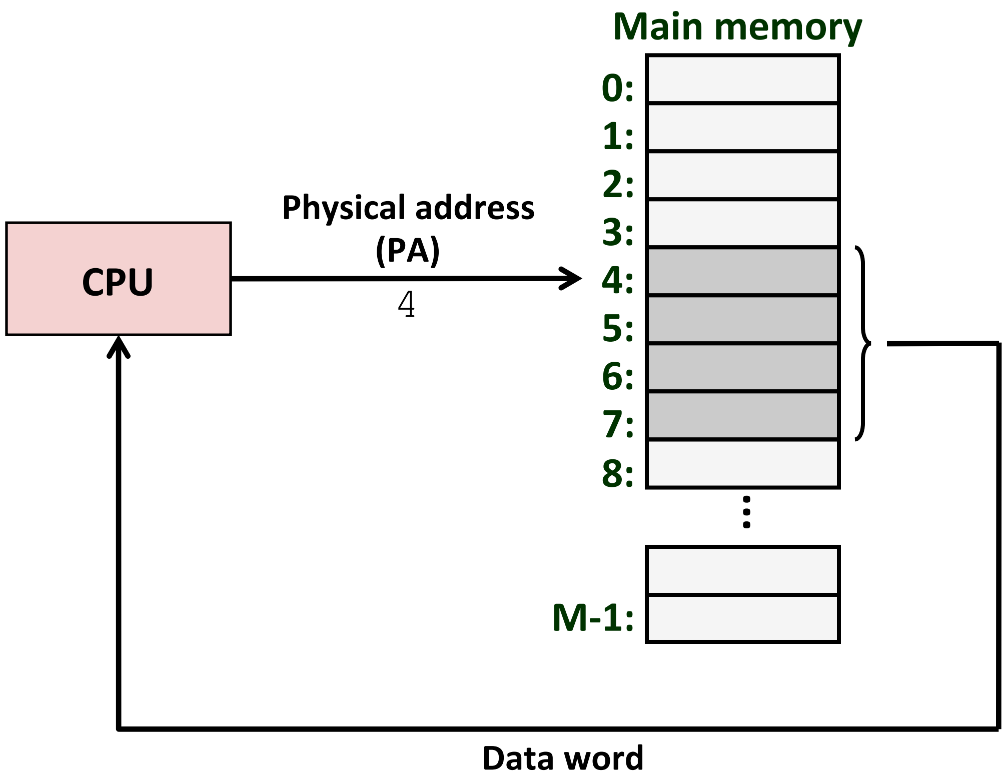
- **Every byte in main memory:**
  one physical address; zero, one, or more virtual addresses

Mapping



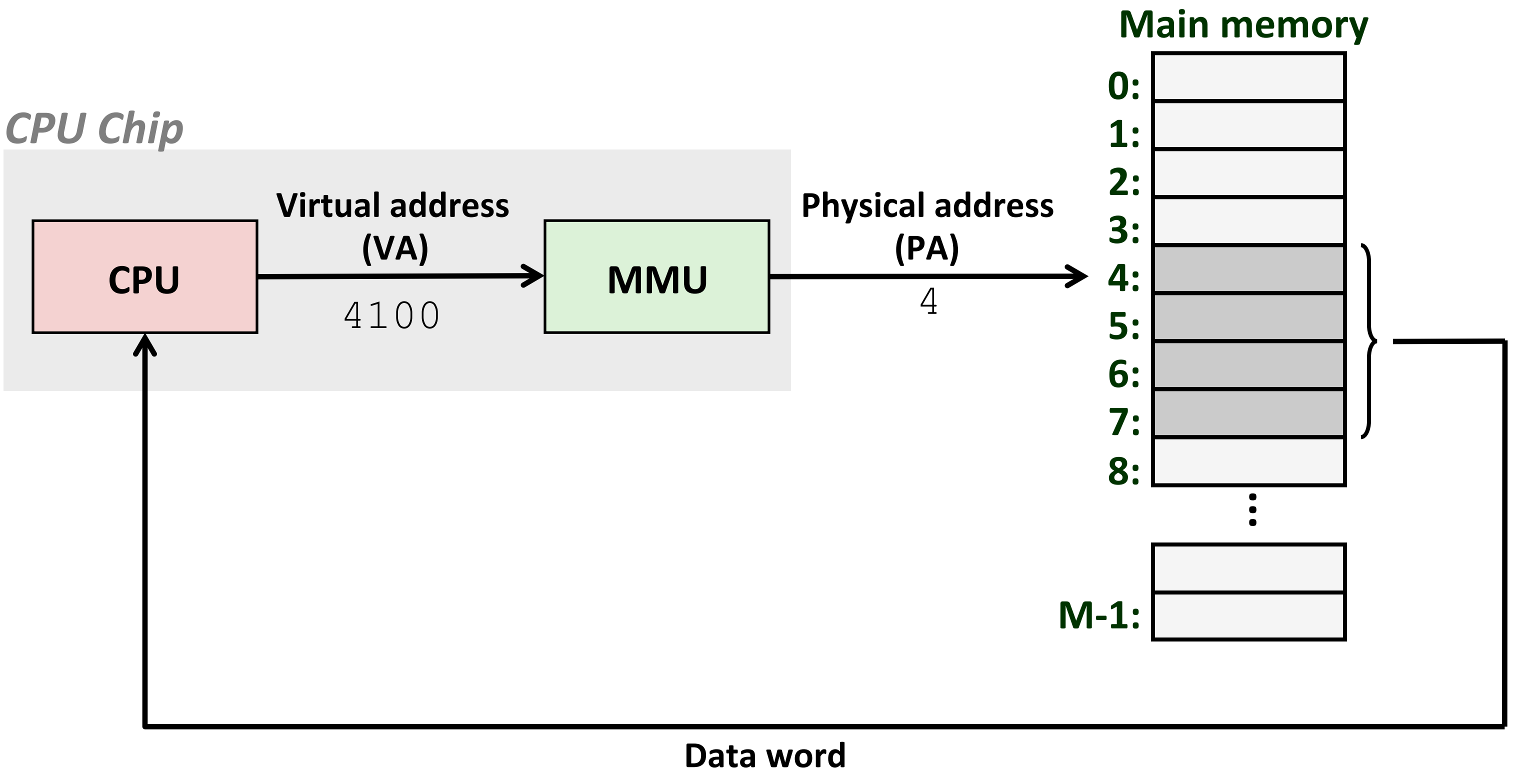
A virtual address can be mapped to either physical memory or disk.

A System Using Physical Addressing



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

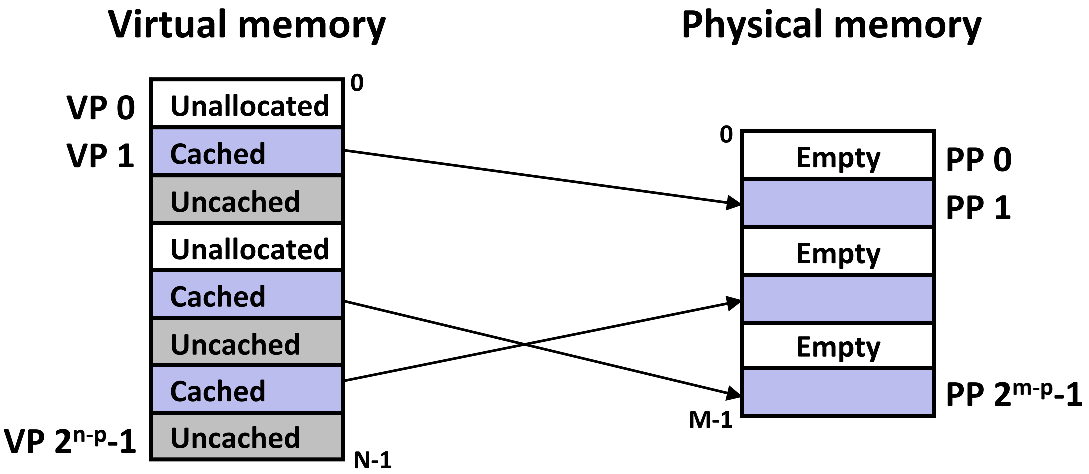
A System Using Virtual Addressing



- Used in all modern desktops, laptops, servers
- One of the great ideas in computer science

VM and the Memory Hierarchy

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



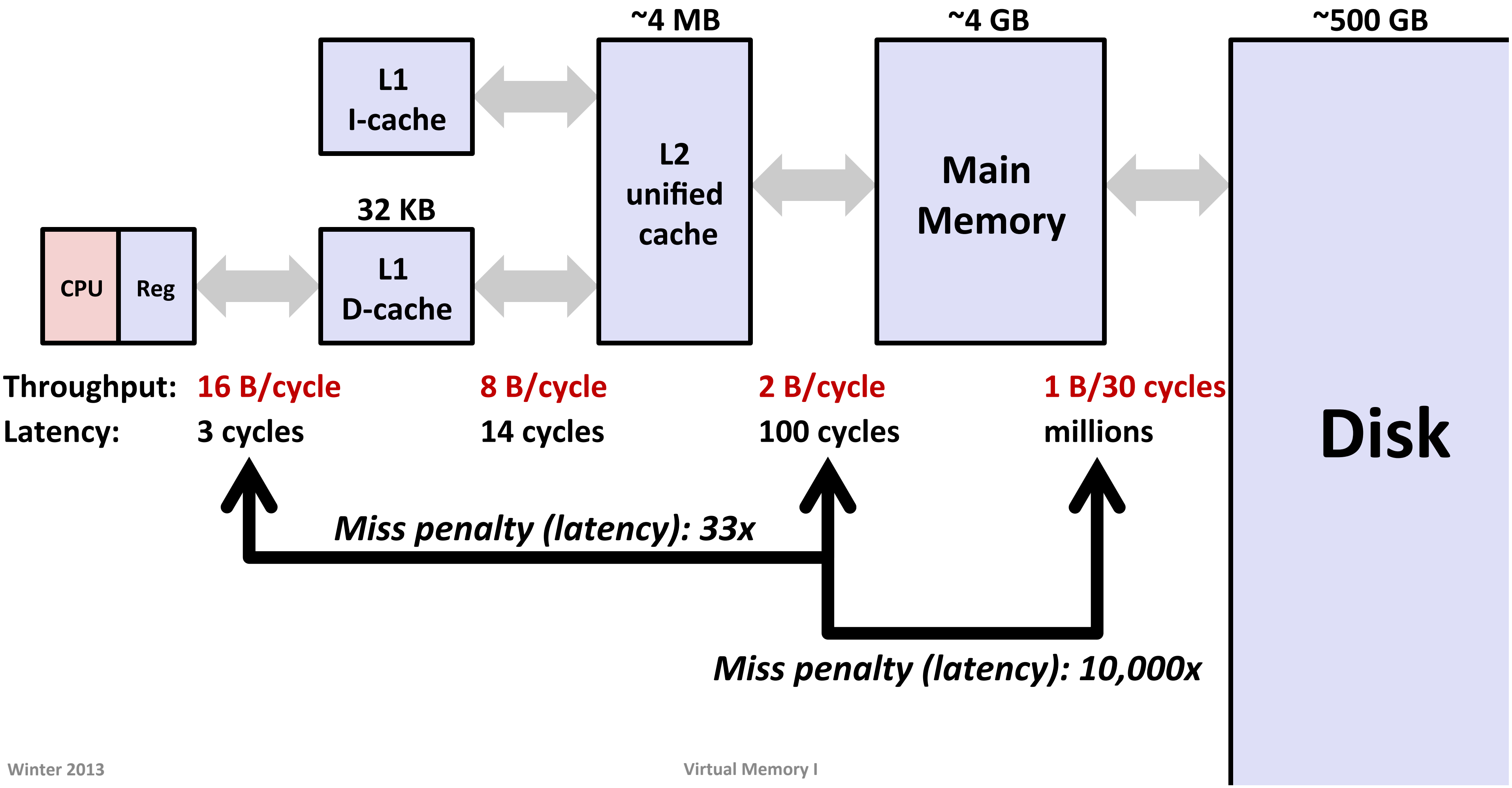
Virtual pages (VPs) stored on disk

Physical pages (PPs) cached in DRAM

Memory Hierarchy: Core 2 Duo

Not drawn to scale

L1/L2 cache: 64 B blocks



~4 MB

~4 GB

~500 GB

L1 I-cache

32 KB

L1 D-cache

CPU

Reg

L2 unified cache

Main Memory

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

DRAM Cache Organization

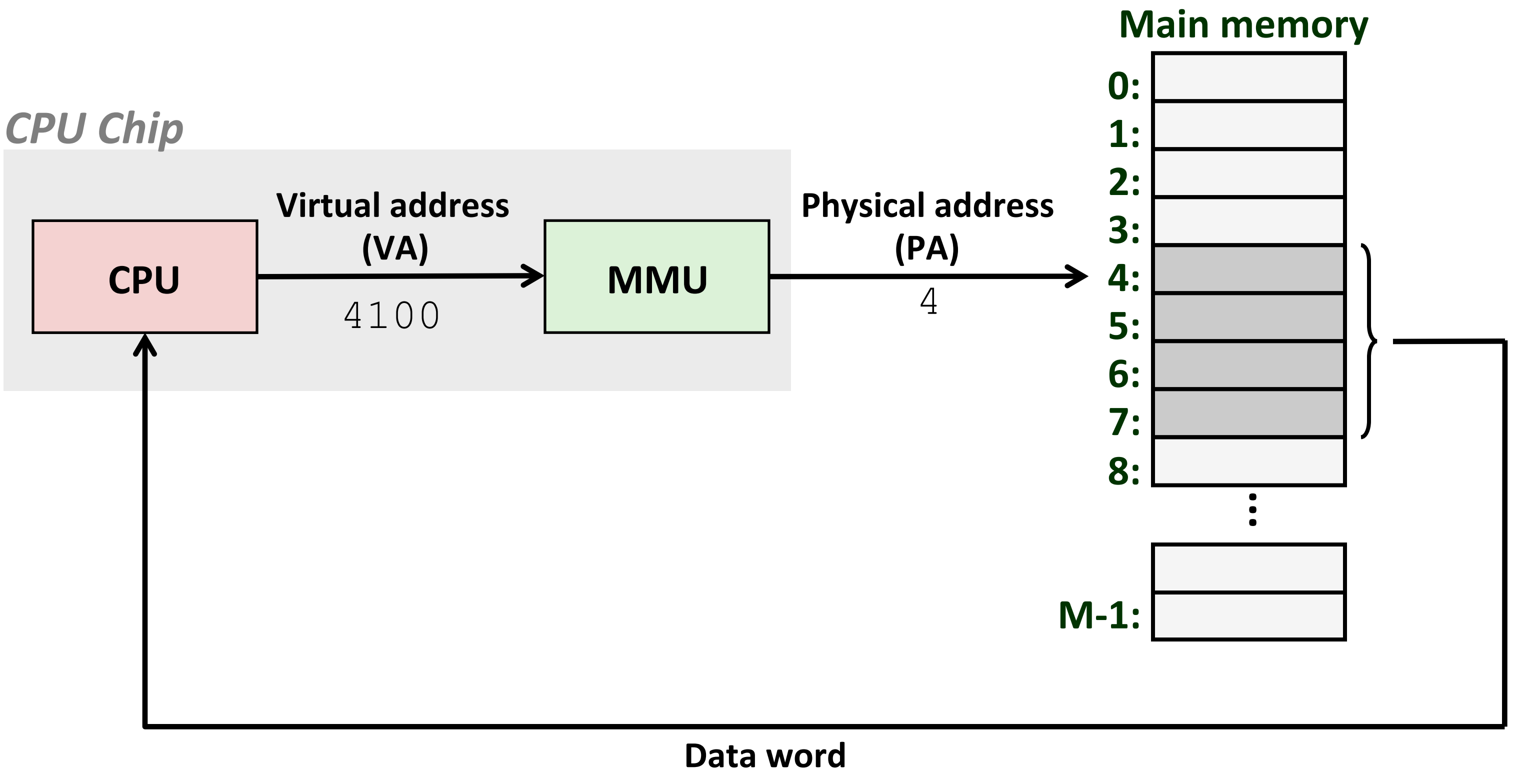
- **DRAM cache organization driven by the enormous miss penalty**
  - DRAM is about 10x slower than SRAM
  - Disk is about 10,000x slower than DRAM
    - (for first byte; faster for next byte)
- **Consequences?**
  - Block size?
  - Associativity?
  - Write-through or write-back?

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

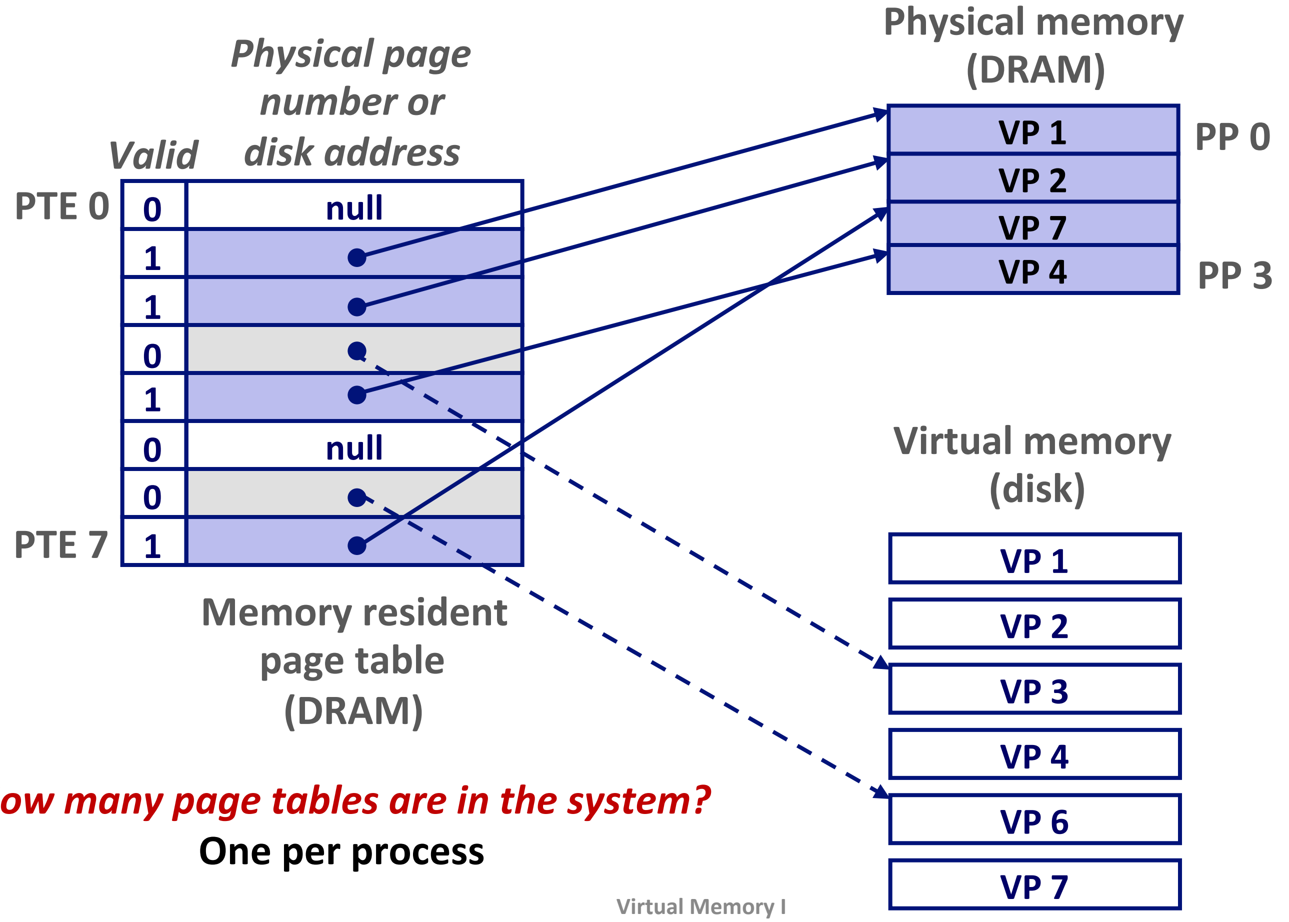
Indexing into the “DRAM Cache”



How do we perform the VA -> PA translation?

Address Translation: Page Tables

- A *page table* (PT) is an array of *page table entries* (PTEs) that maps virtual pages to physical pages.



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

Memory resident page table (DRAM)

Physical memory (DRAM)

VP 1	PP 0
VP 2	
VP 7	
VP 4	PP 3

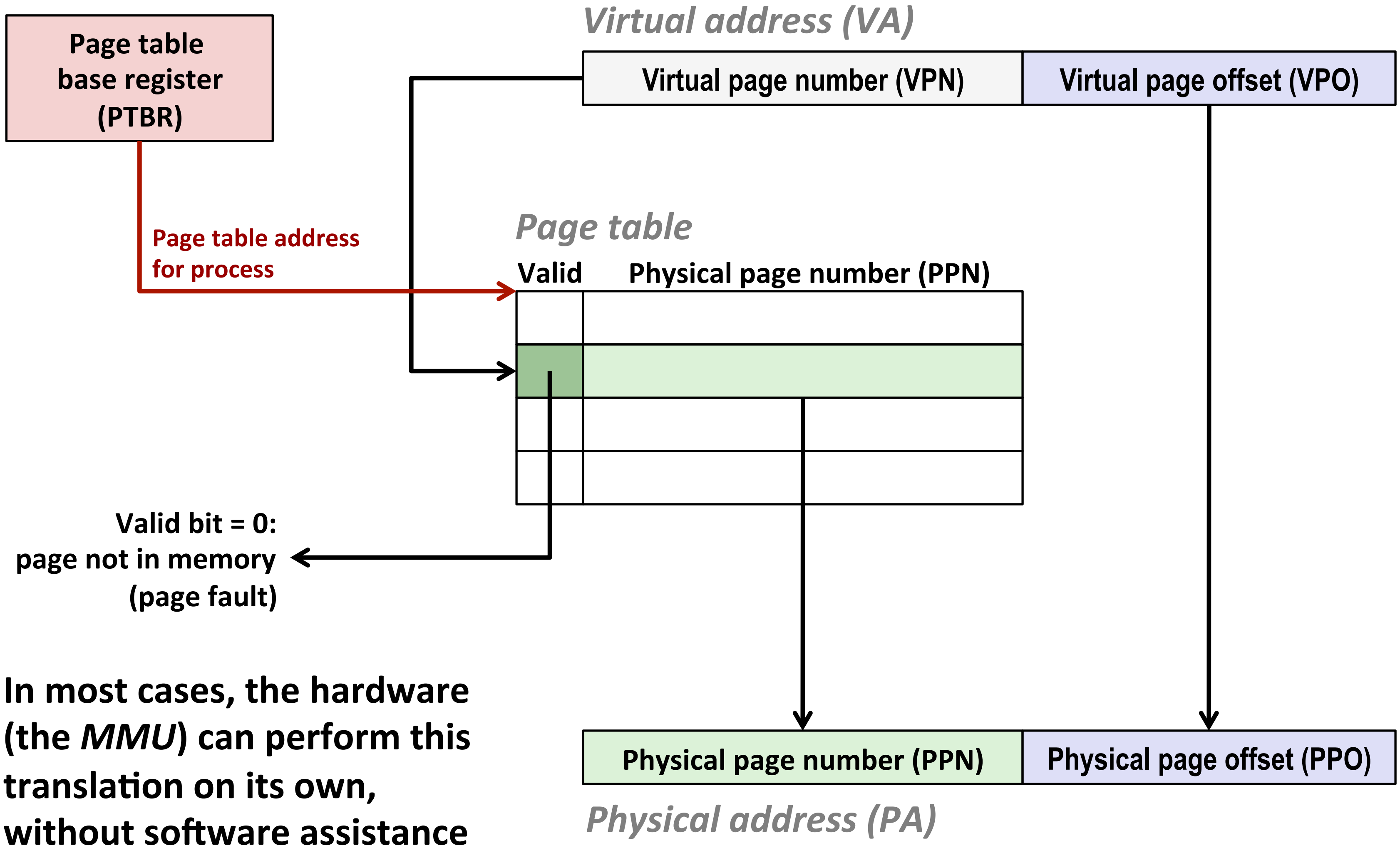
Virtual memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

How many page tables are in the system?

One per process

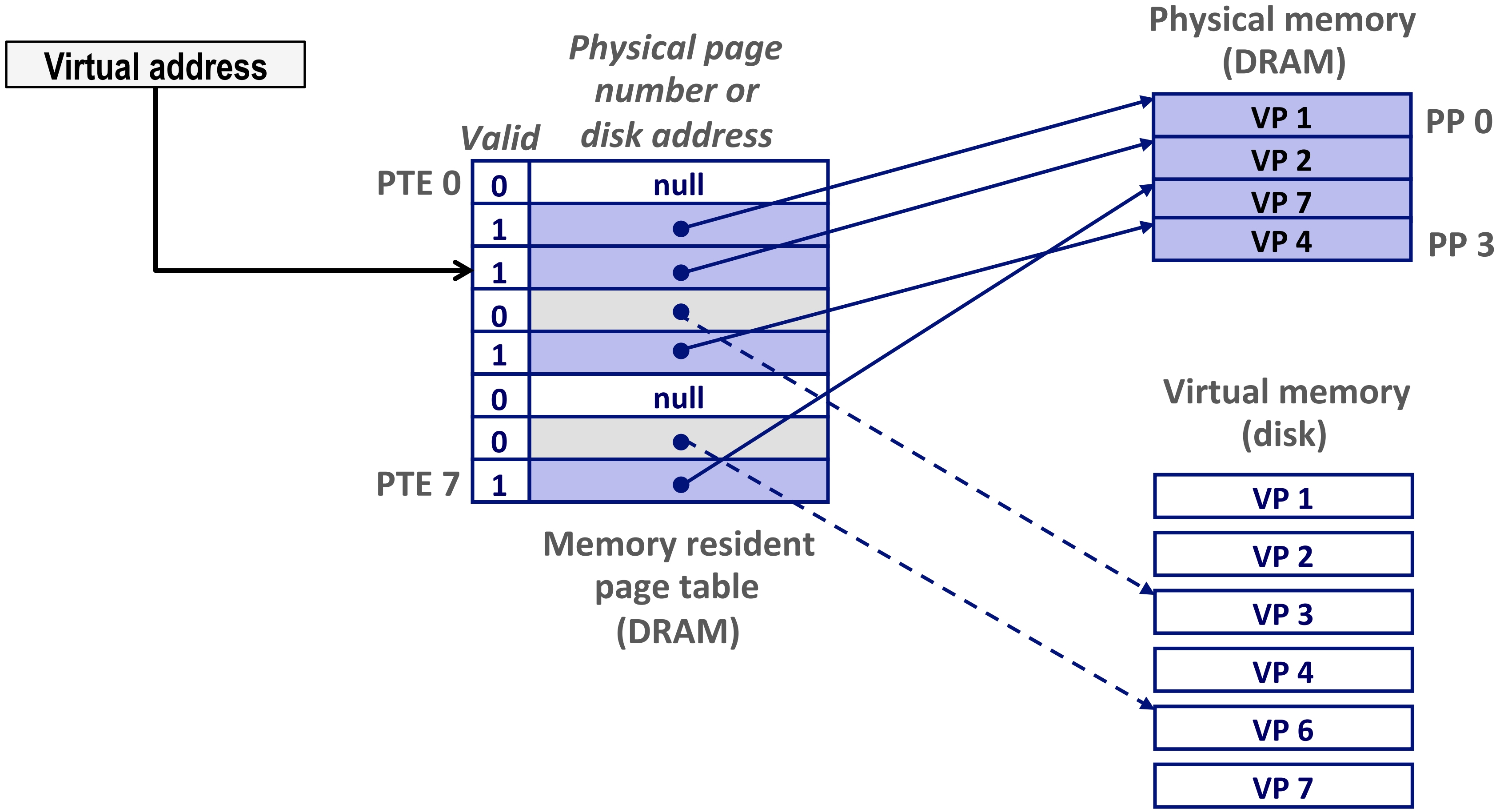
Address Translation With a Page Table


Page table base register (PTBR)
Virtual address (VA)
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 (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**



Virtual address

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

Memory resident page table (DRAM)

Physical memory (DRAM)

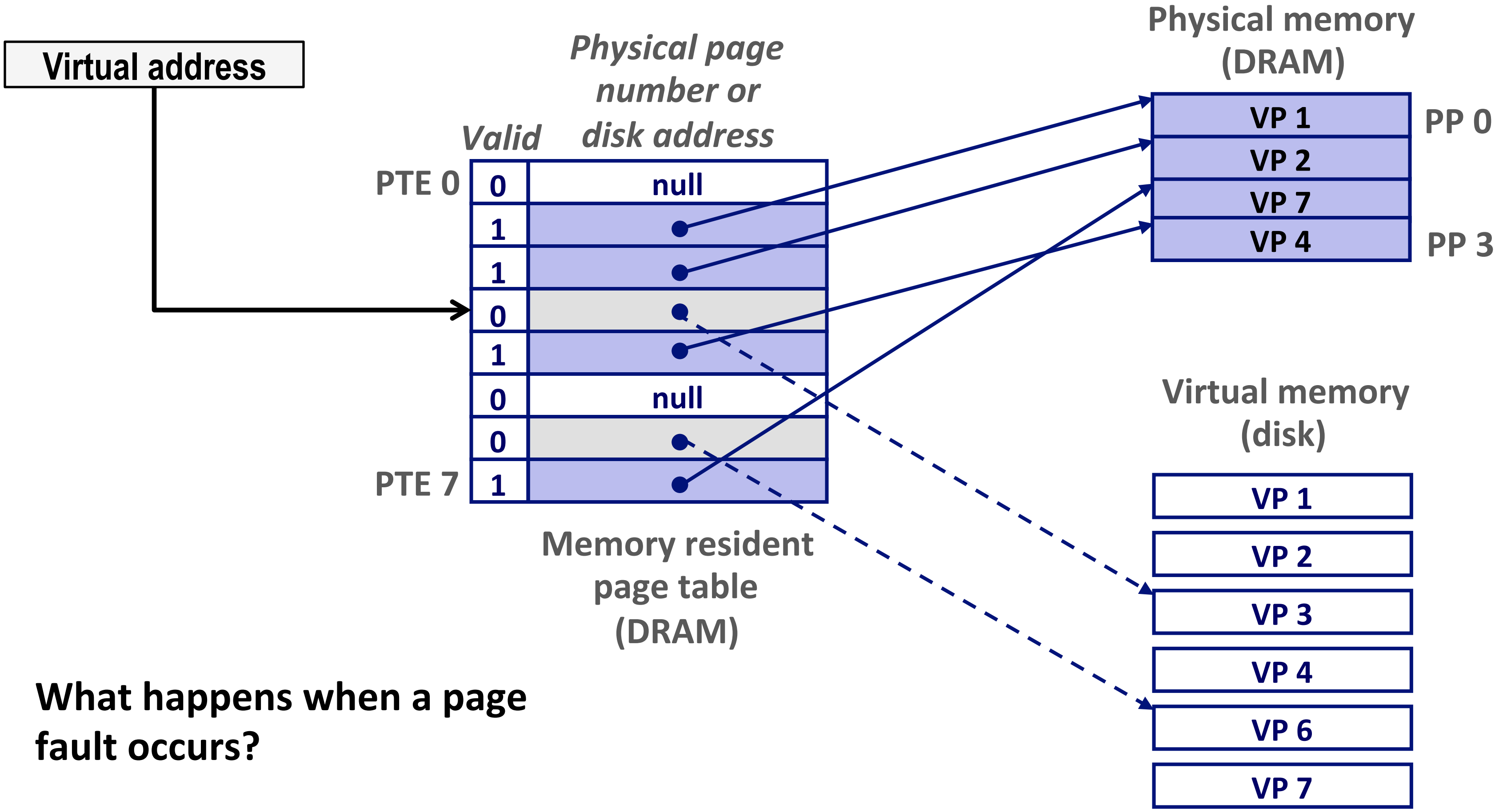
VP 1	PP 0
VP 2	
VP 7	
VP 4	PP 3

Virtual memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

Page Fault

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



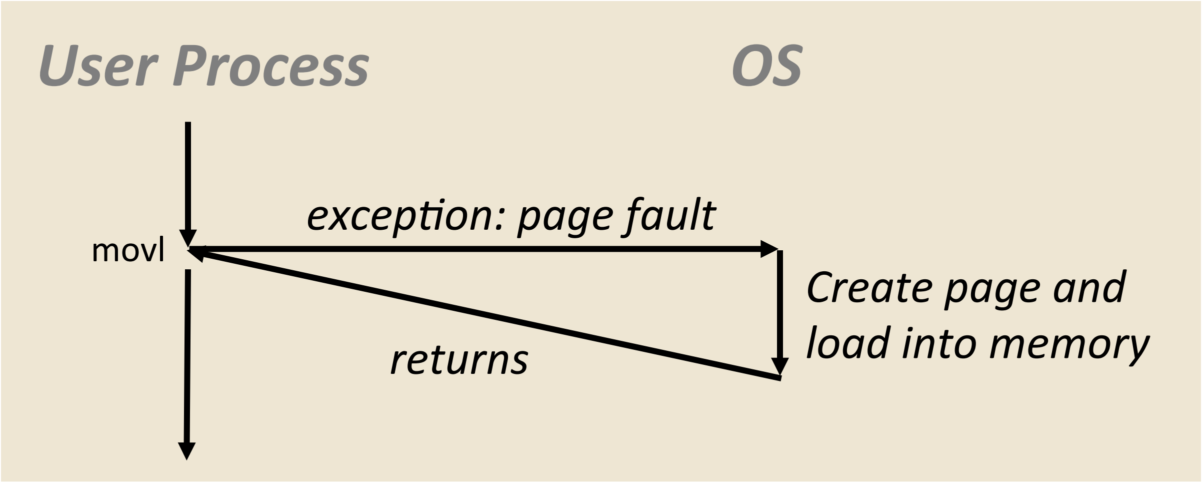
What happens when a page fault occurs?

Fault Example: Page Fault

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

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

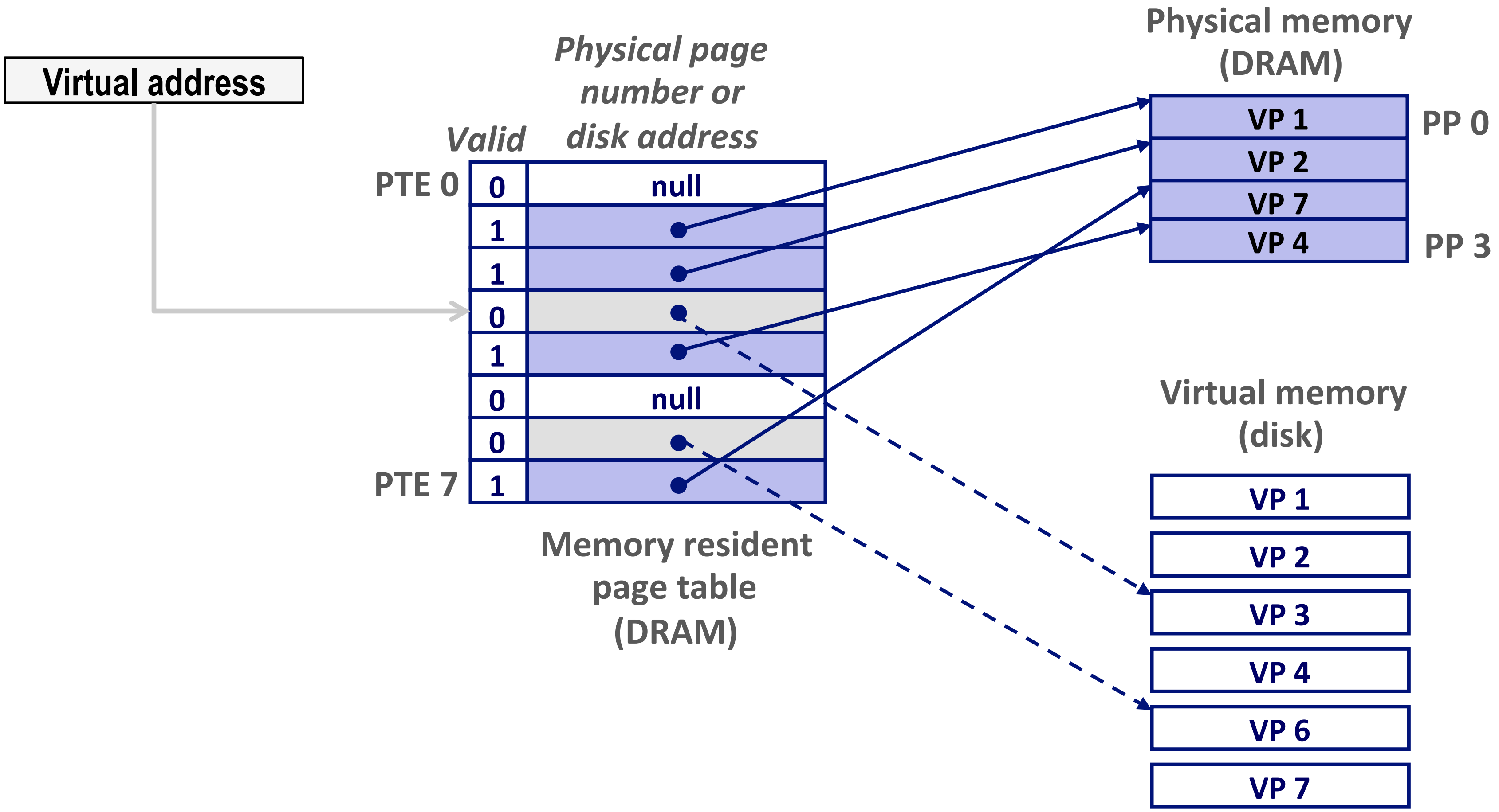
```
 80483b7:       c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```



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

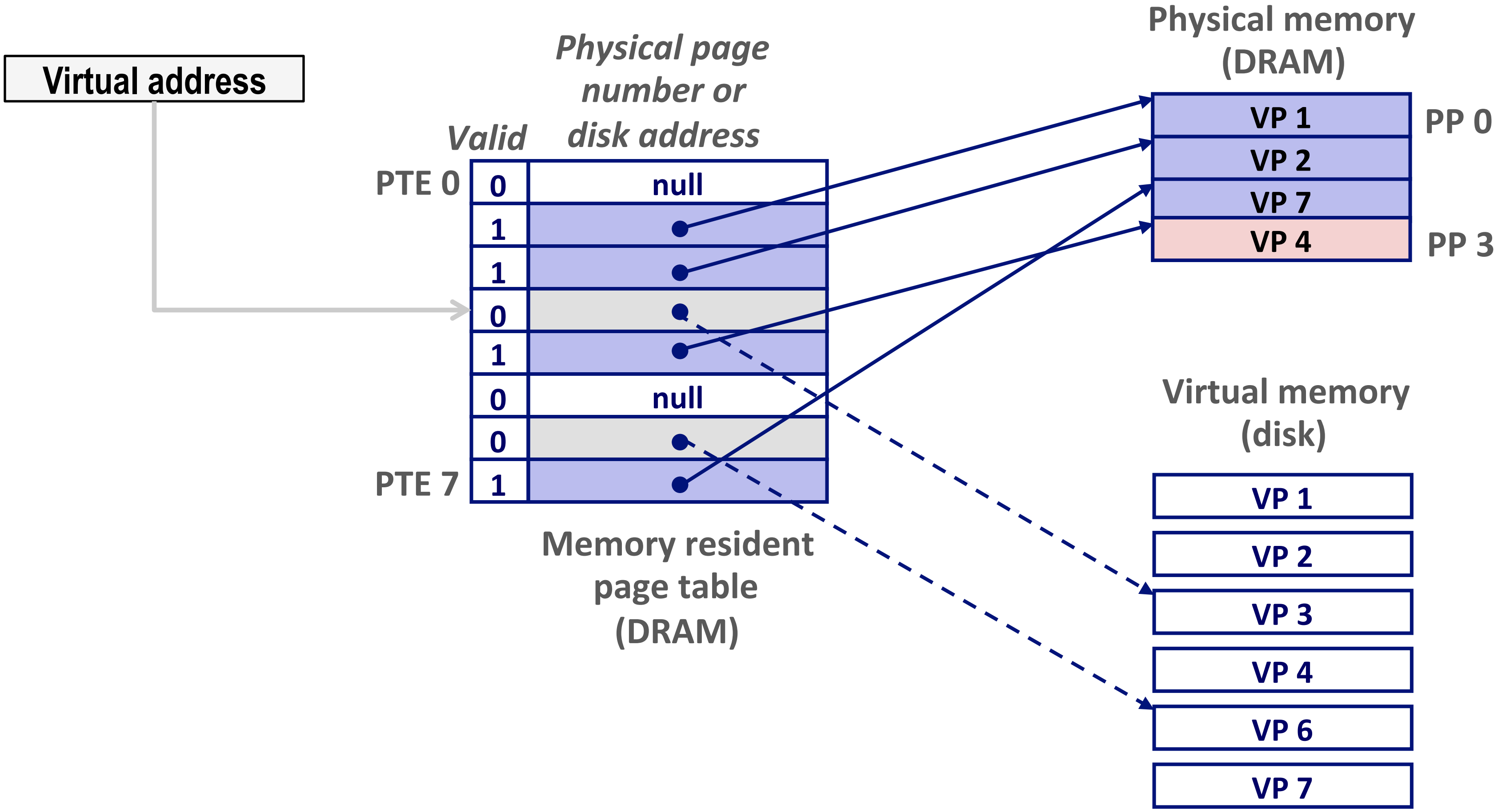
Handling Page Fault

- Page miss causes page fault (an exception)



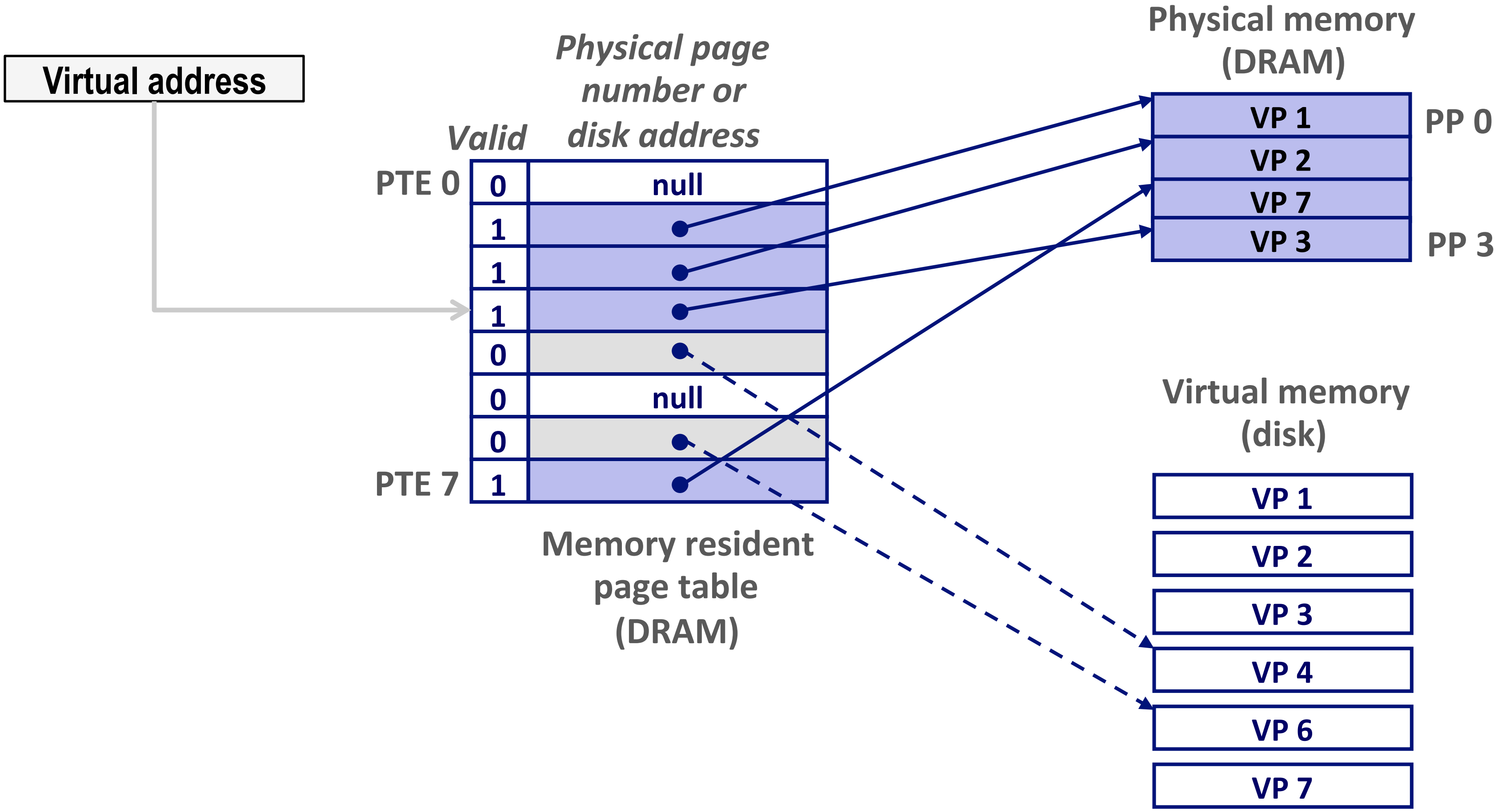
Handling Page Fault

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



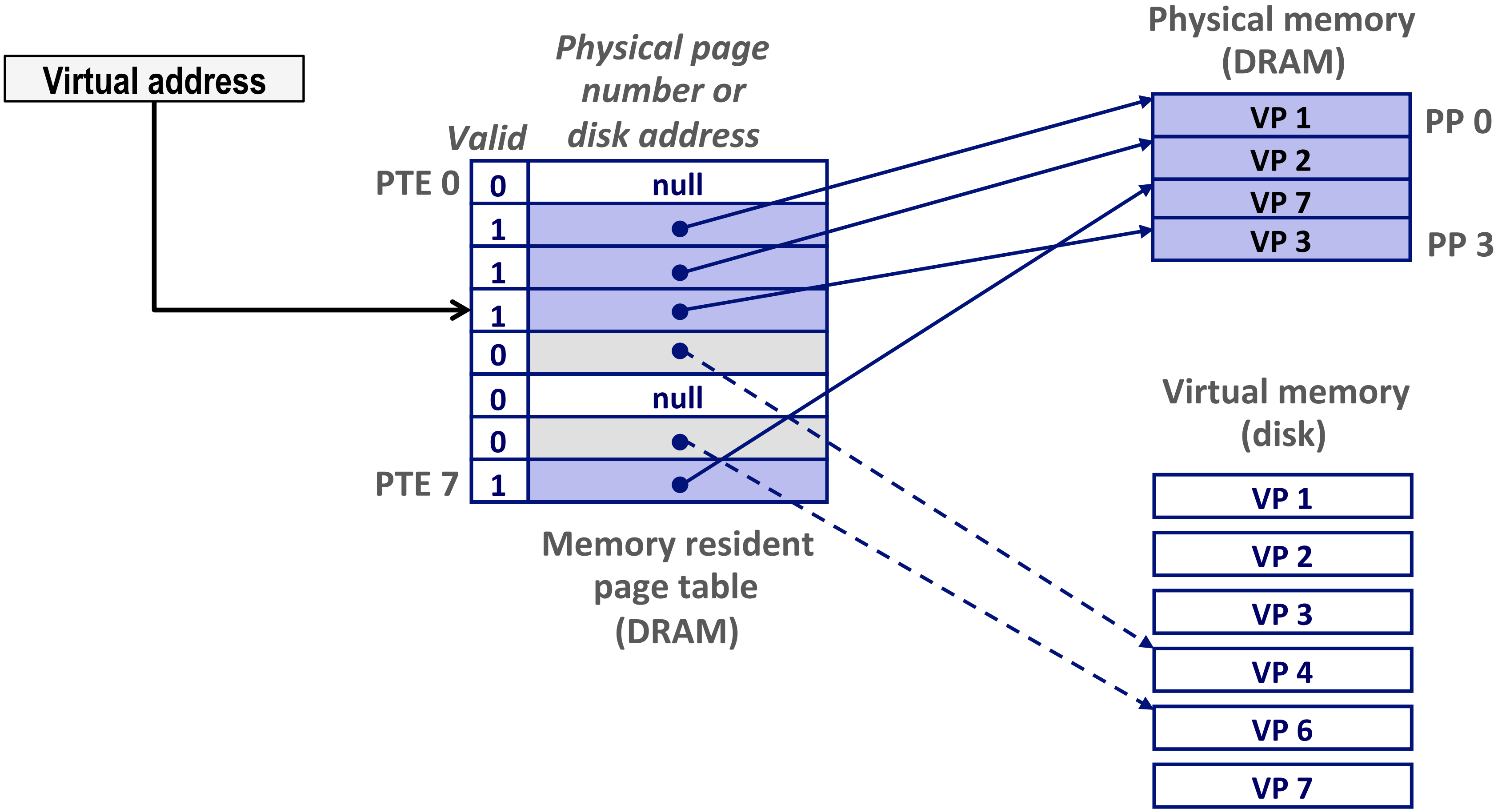
Handling Page Fault

- 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 it work? Locality

- **Virtual memory works well 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***
  - 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