Virtual Memory

CSCI 224 / ECE 317: Computer Architecture

Instructor:
Prof. Jason Fritts

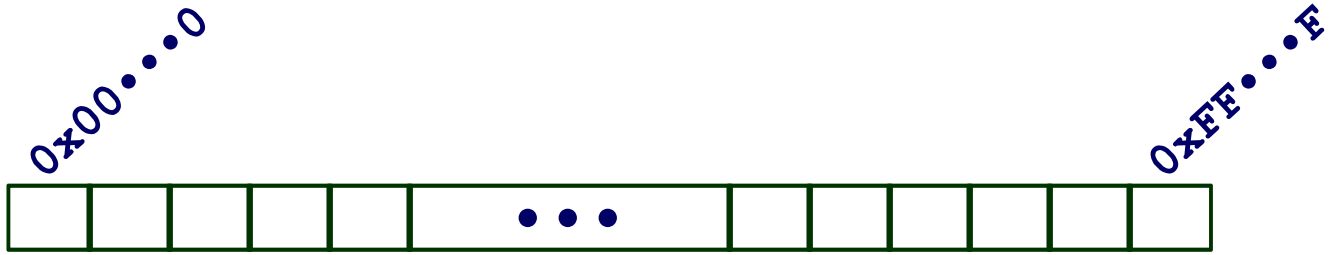
Slides adapted from Bryant & O'Hallaron's slides

Data Representation in Memory

- **Memory organization within a process**

- **Virtual vs. Physical memory**
  - Fundamental Idea and Purpose
  - Page Mapping
  - Address Translation
  - Per-Process Mapping and Protection

Recall: Basic Memory Organization

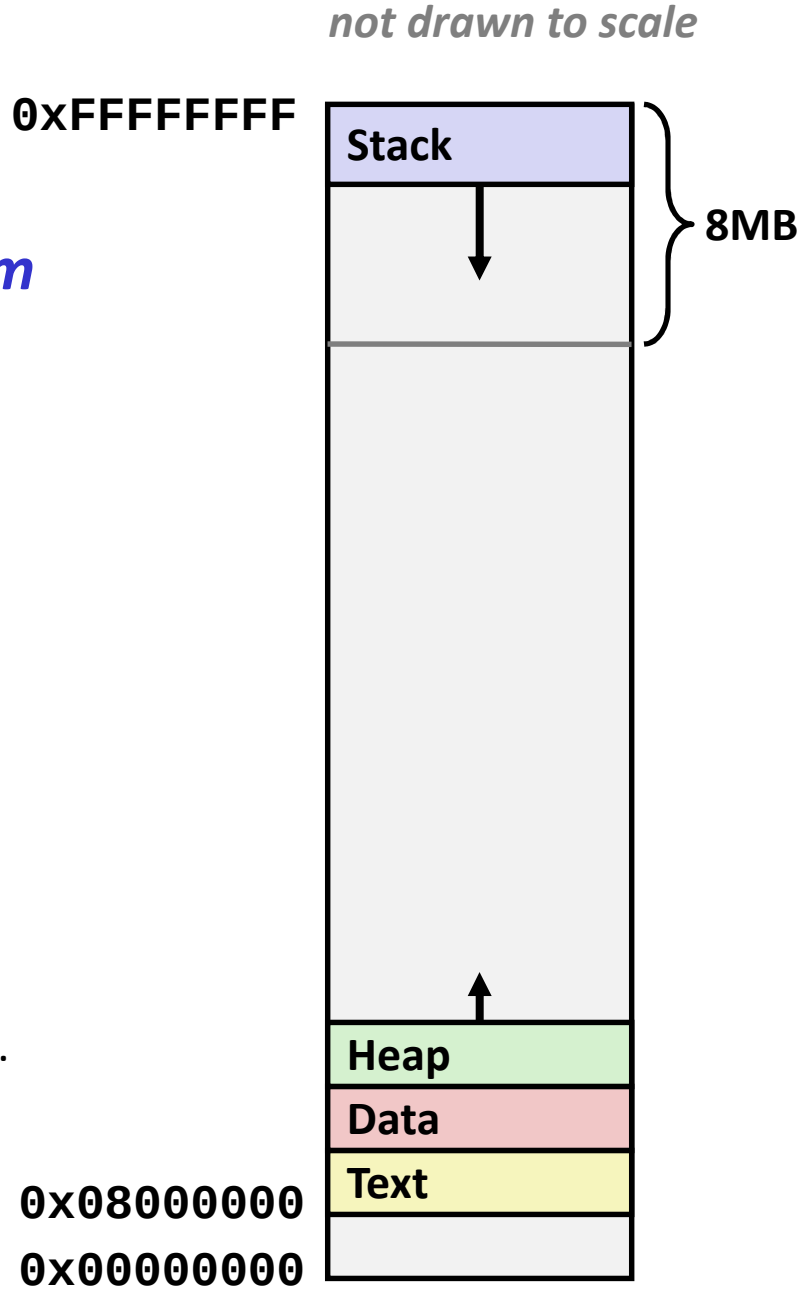


- **Byte-Addressable Memory**
  - Conceptually a very large array, with a unique address for each byte
  - Processor width determines address range:
    - 32-bit processor has 2^{32} unique addresses
    - 64-bit processor has 2^{64} unique addresses

- ***Where does a given process reside in memory?***
  - depends upon the perspective…
    - virtual memory: process can use most any virtual address
    - physical memory: location controlled by OS

Virtual Address Space for IA32 (x86) Linux

- ***All processes have the same uniform view of memory***
- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables
- **Heap**
  - Dynamically allocated storage
  - When call *malloc(), calloc(), new()*
- **Data**
  - Statically allocated data
  - E.g., global variables, arrays, structures, etc.
- **Text**
  - Executable machine instructions
  - Read-only data

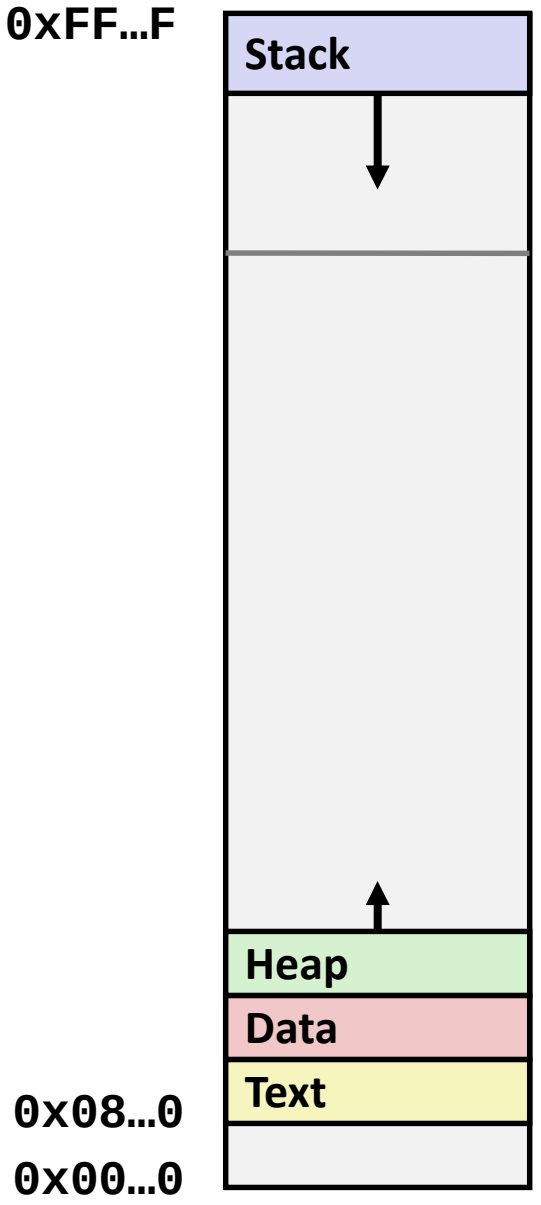


Memory Allocation Example

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Where does everything go?

not drawn to scale



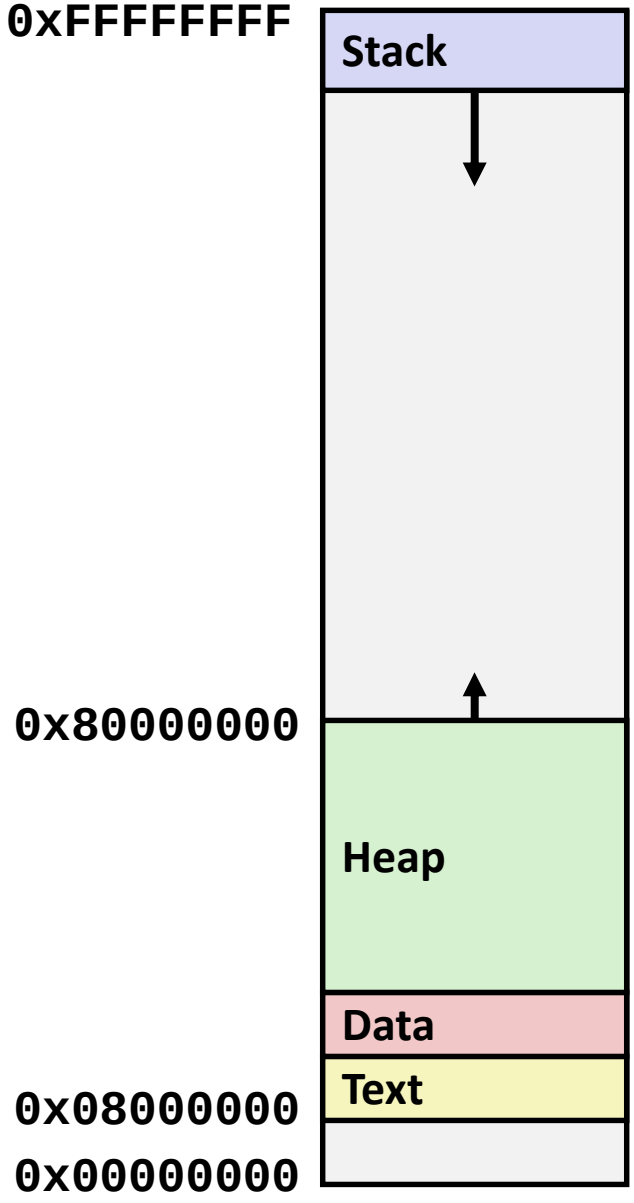
Addresses in IA32 (x86)

address range ~2^{32}

`$esp`	`0xffffbcd0`
`p3`	`0x65586008`
`p1`	`0x55585008`
`p4`	`0x1904a110`
`p2`	`0x1904a008`
`&p2`	`0x18049760`
`&beyond`	`0x08049744`
`big_array`	`0x18049780`
`huge_array`	`0x08049760`
`main()`	`0x080483c6`
`useless()`	`0x08049744`

`malloc()` is dynamically linked
address determined at runtime

not drawn to scale



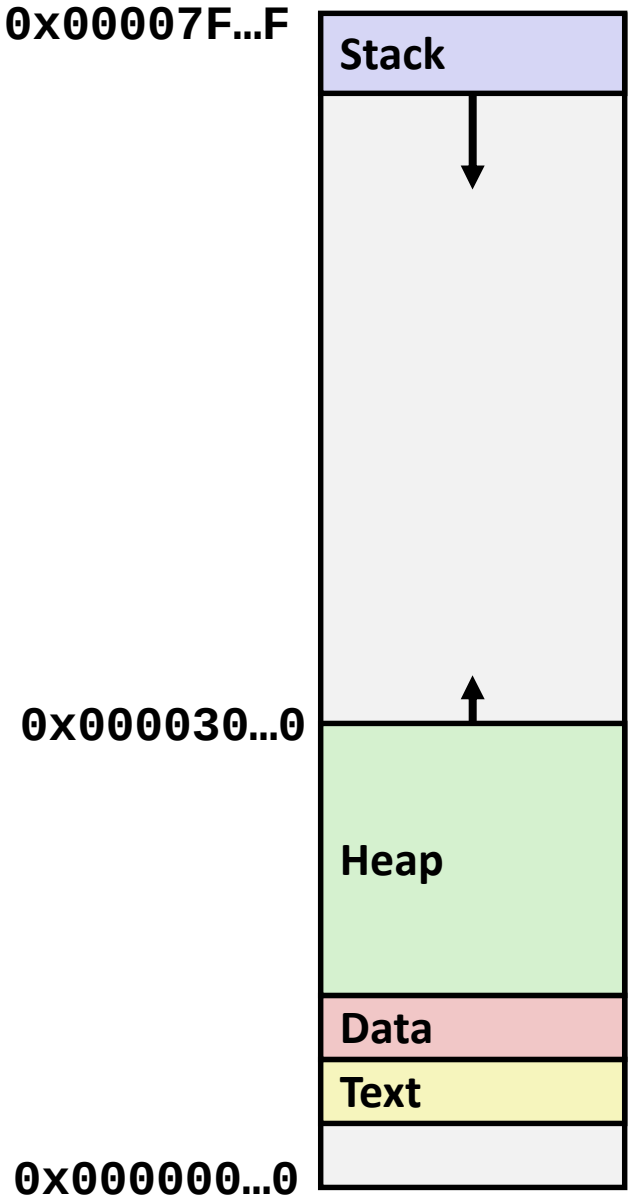
Addresses in x86-64

address range ~2^{47}

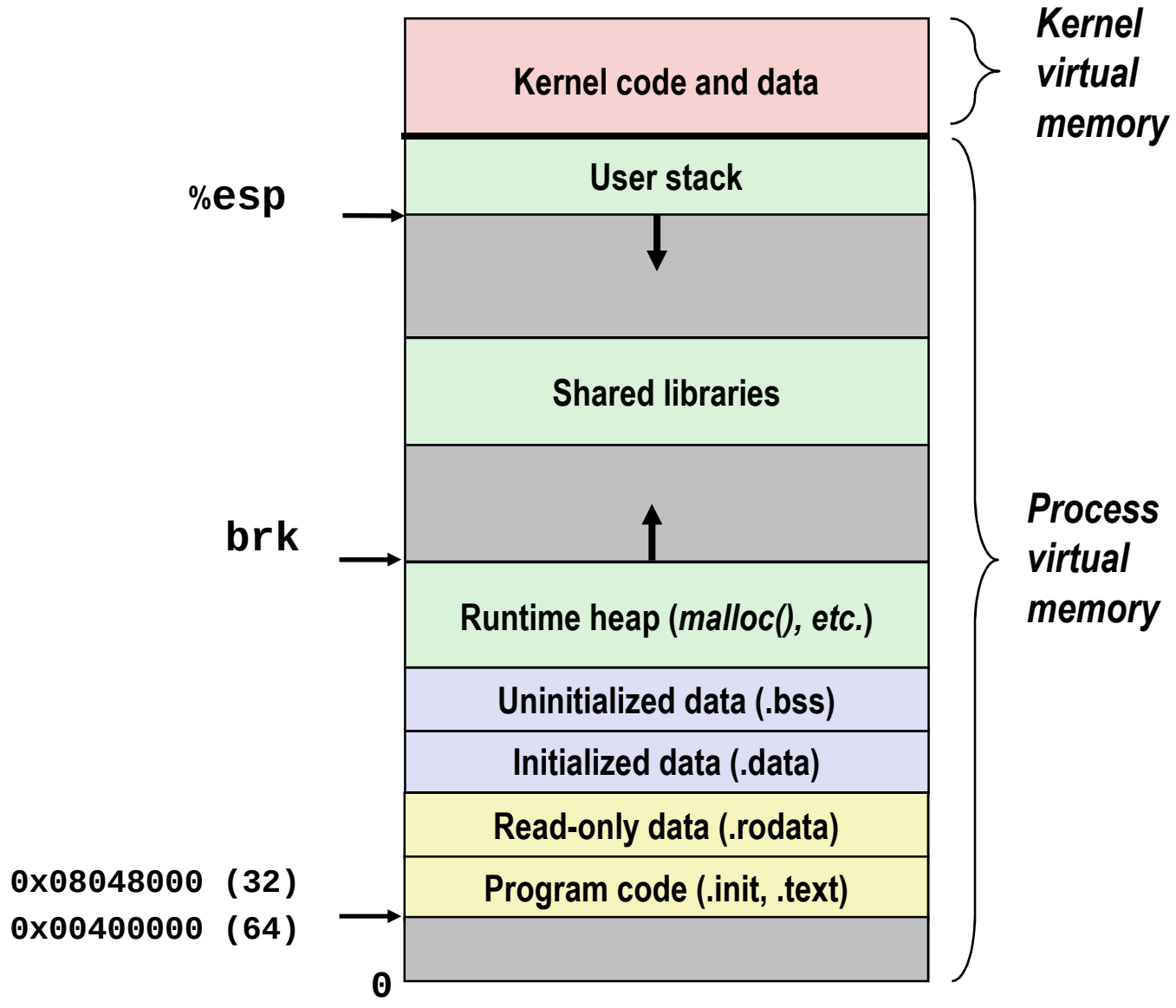
`$rsp`	`0x00007fffffff8d1f8`
`p3`	`0x00002aaabaadd010`
`p1`	`0x00002aaaaaadc010`
`p4`	`0x0000000011501120`
`p2`	`0x0000000011501010`
`&p2`	`0x0000000010500a60`
`&beyond`	`0x0000000000500a44`
`big_array`	`0x0000000010500a80`
`huge_array`	`0x0000000000500a50`
`main()`	`0x0000000000400510`
`useless()`	`0x0000000000400500`

`malloc()` is dynamically linked
address determined at runtime

not drawn to scale



Detailed Virtual Address Space for a Linux Process

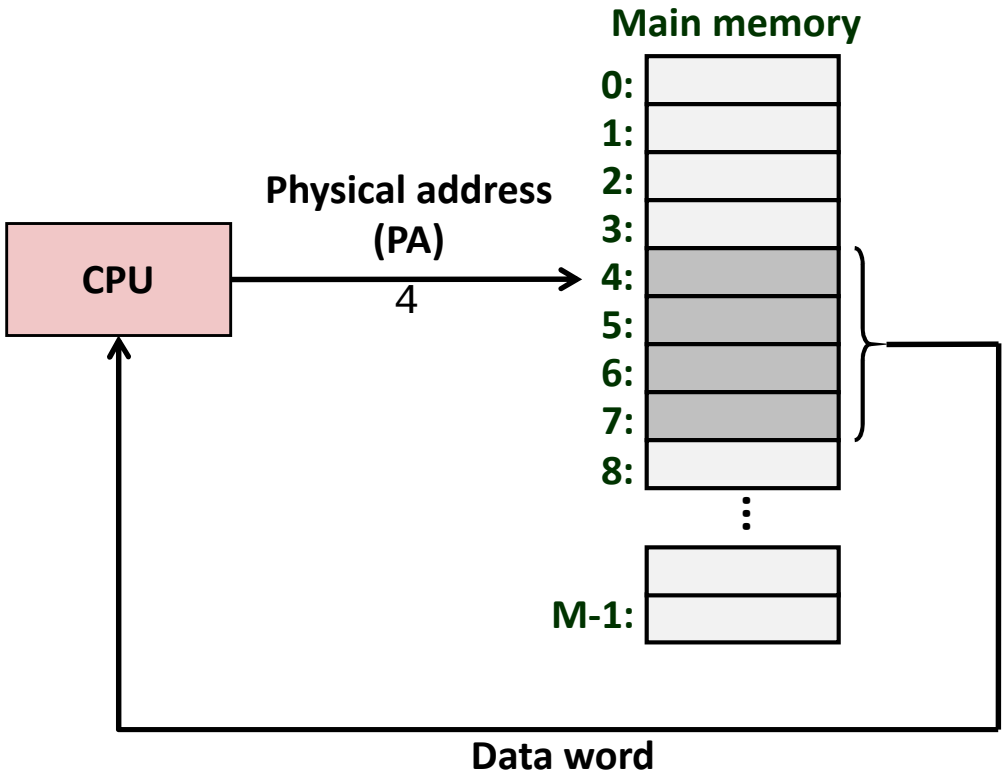


Data Representation in Memory

- Memory organization within a process

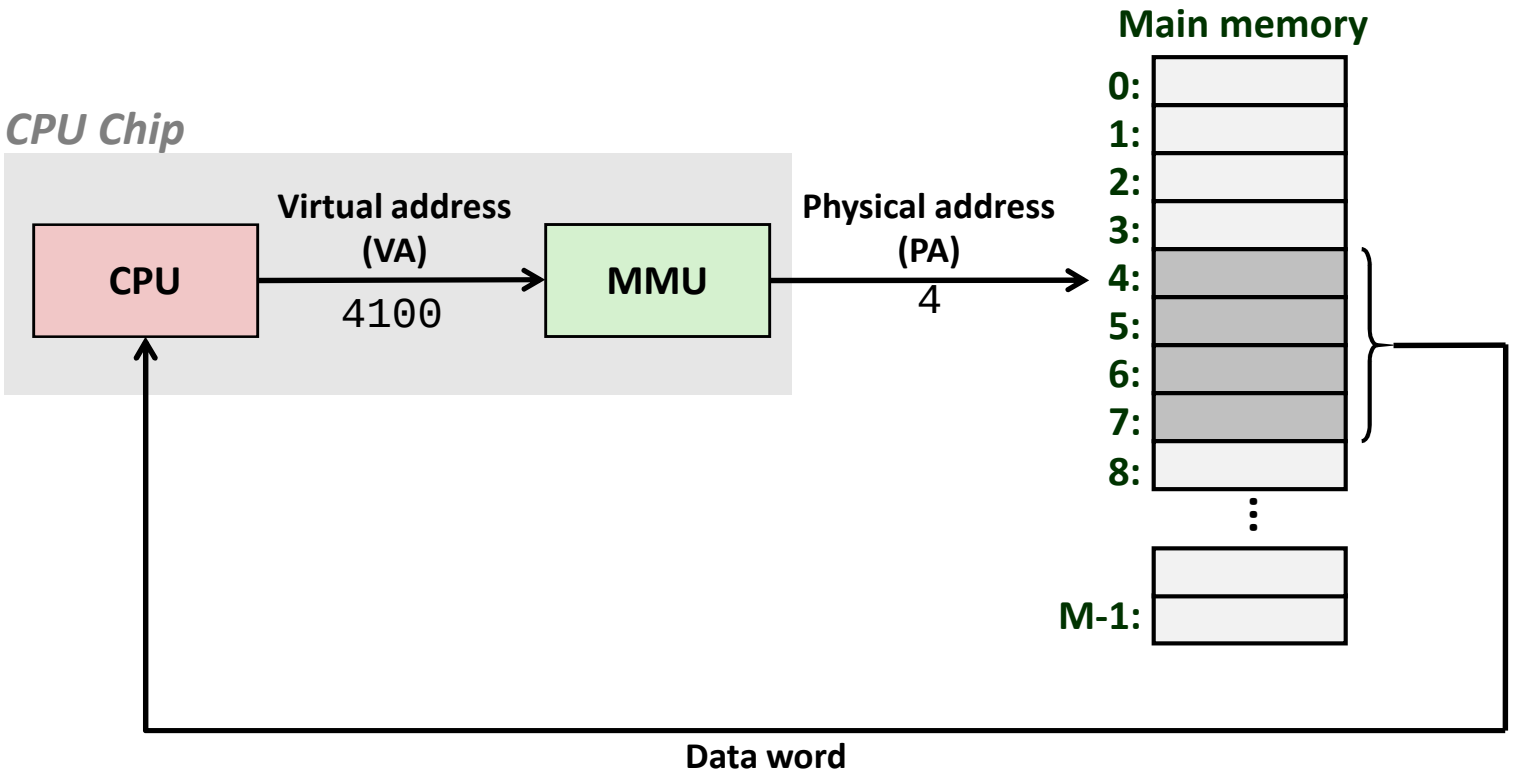
- **Virtual vs. Physical memory**
  - Fundamental Idea and Purpose
  - Page Mapping
  - Address Translation
  - Per-Process Mapping and Protection

Contrast: System Using Physical Addressing



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

Contrast: System Using Virtual Addressing



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

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
  {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

Why Virtual Memory (VM)?

- **Uses main memory efficiently**
  - Use DRAM as a cache for the parts of a virtual address space
- **Simplifies memory management**
  - Each process gets the same uniform linear address space
- **Isolates address spaces**
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information

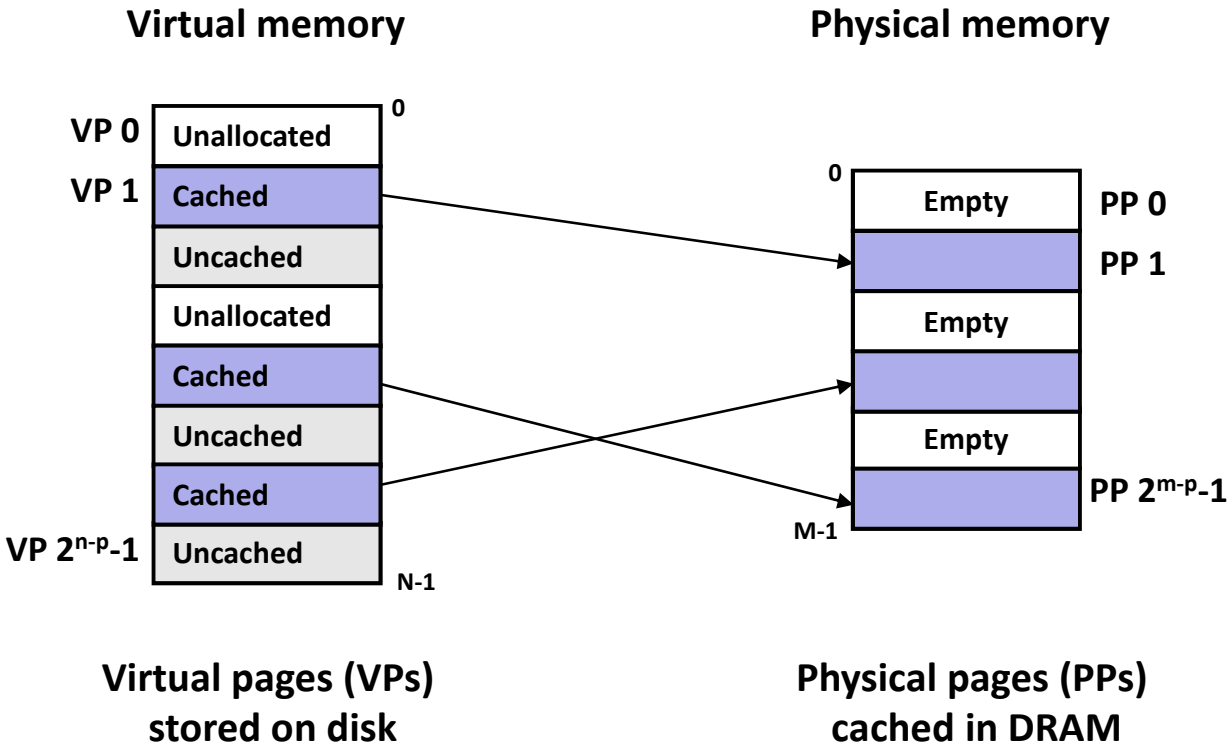
Data Representation in Memory

- Memory organization within a process

- **Virtual vs. Physical memory**
  - Fundamental Idea and Purpose
  - Page Mapping
  - Address Translation
  - Per-Process Mapping and Protection

VM as a Tool for Caching

- ***Virtual memory*** **– array of N contiguous bytes stored on disk.**
- **The contents of the array on disk are cached in *physical memory* (*DRAM cache*)**
  - These cache blocks are called ***pages*** (size is $P = 2^p$ bytes)



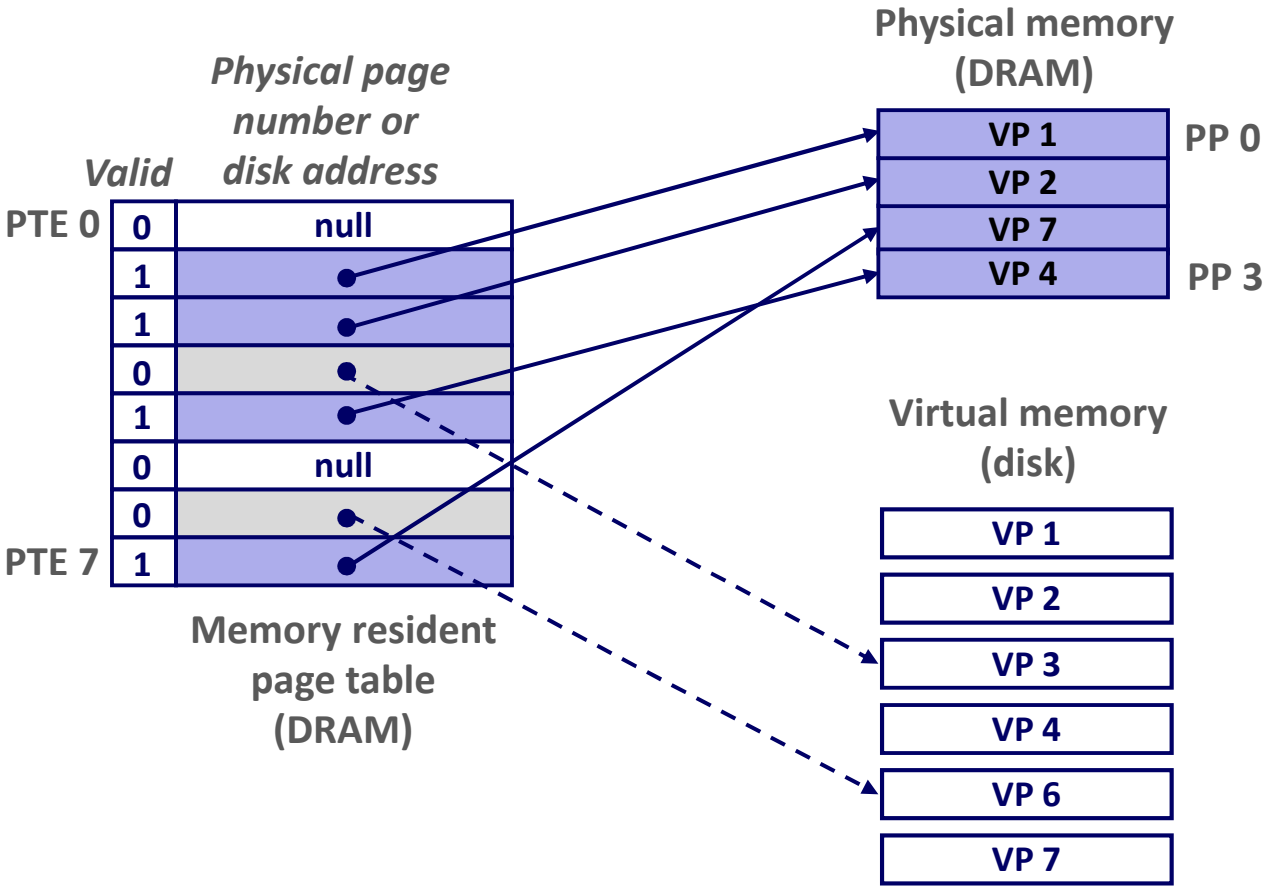
DRAM as a Cache for Disk

- **Disk has enormous miss penalty**
  - DRAM is only about ***10x*** slower than SRAM
  - Disk is much slower than DRAM: about ***10,000x*** slower

- **Consequences**
  - Highly sophisticated algorithms used for organizing DRAM effectively
    - cache memory has relatively simple mechanisms
  - Large page (block) size: typically 4-8 KB, sometimes 4 MB
  - Fully associative
    - any VP can be placed in any PP

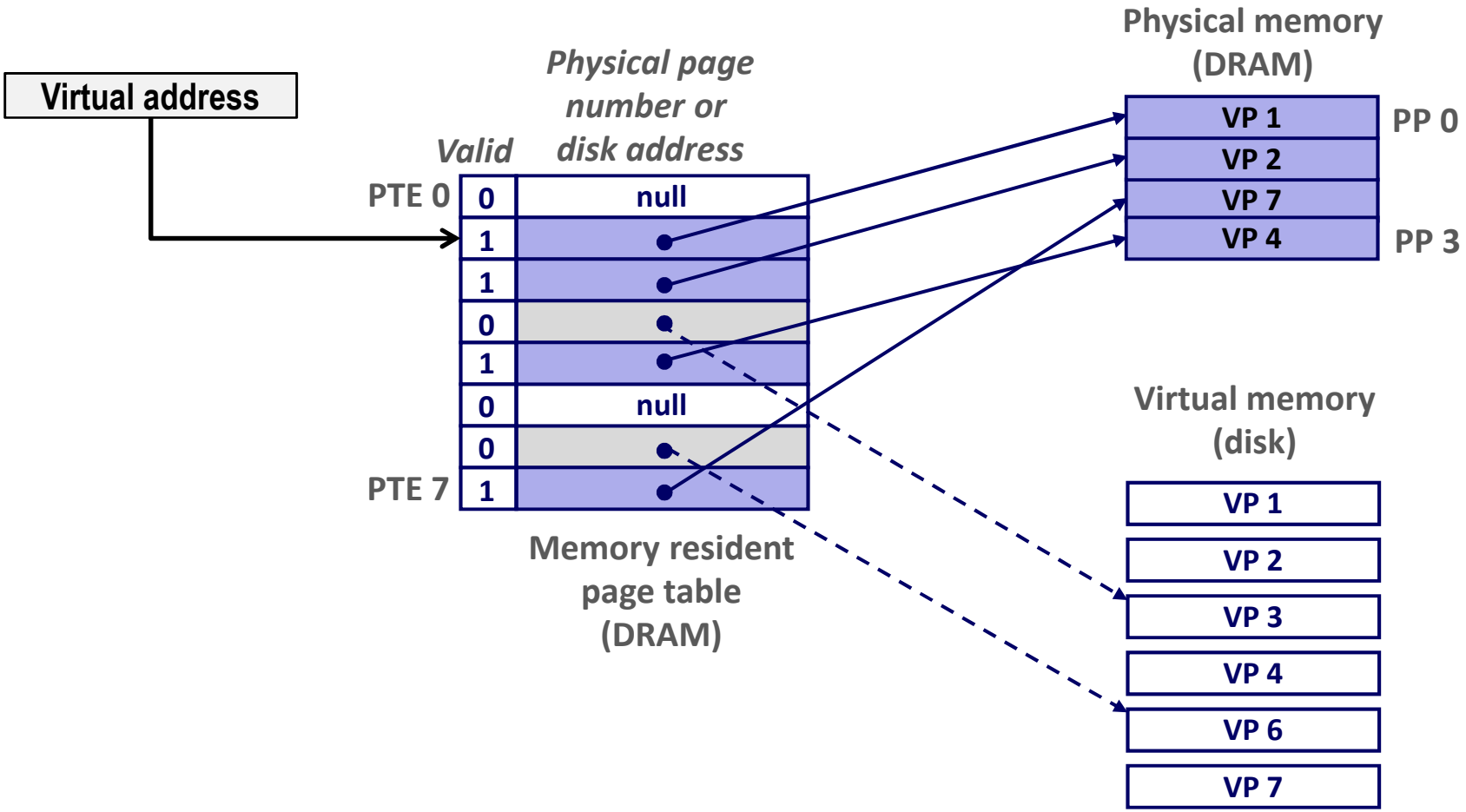
Page Tables

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



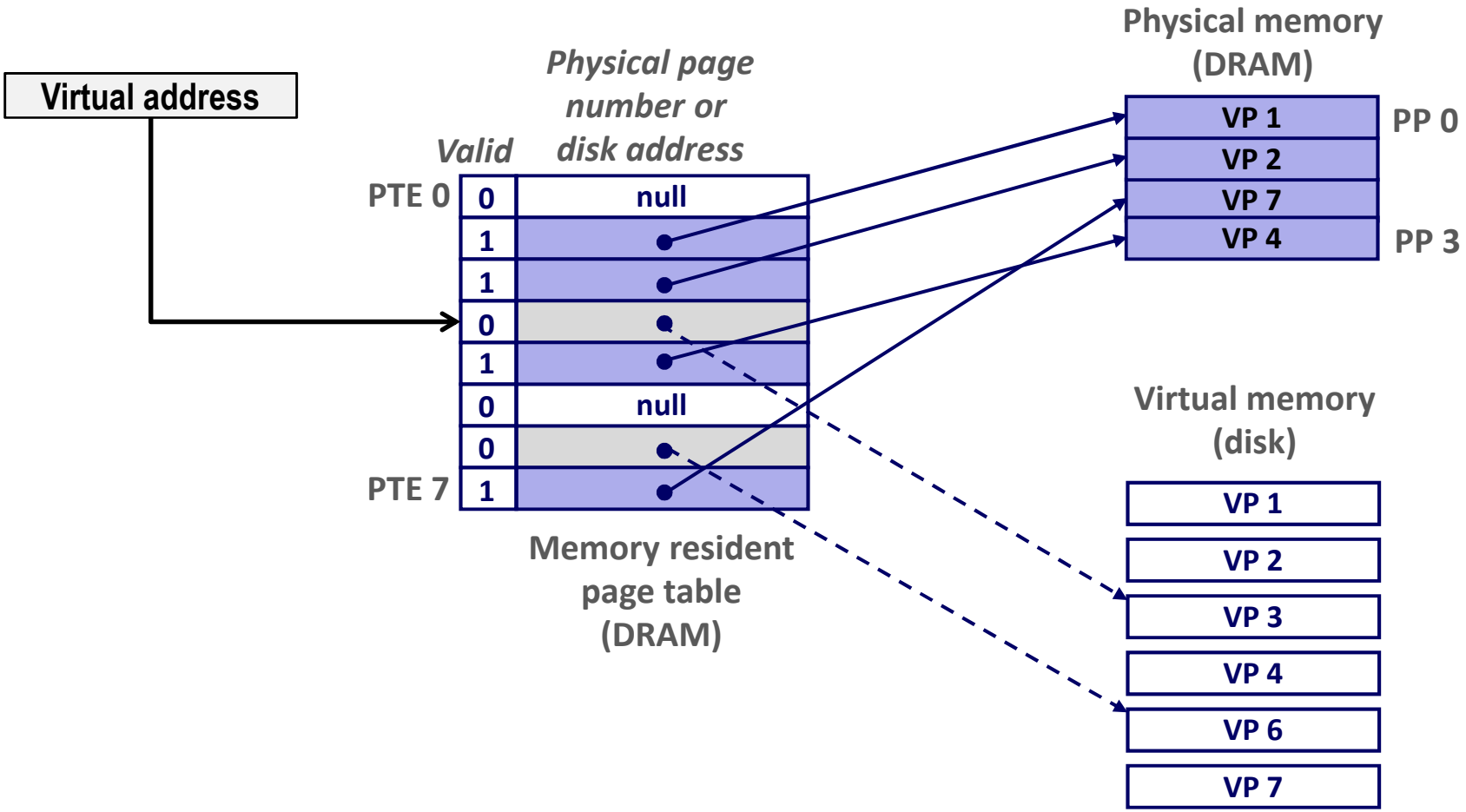
Page Hit

- ***Page hit:***
  - reference to VM word that is in physical memory (DRAM cache hit)



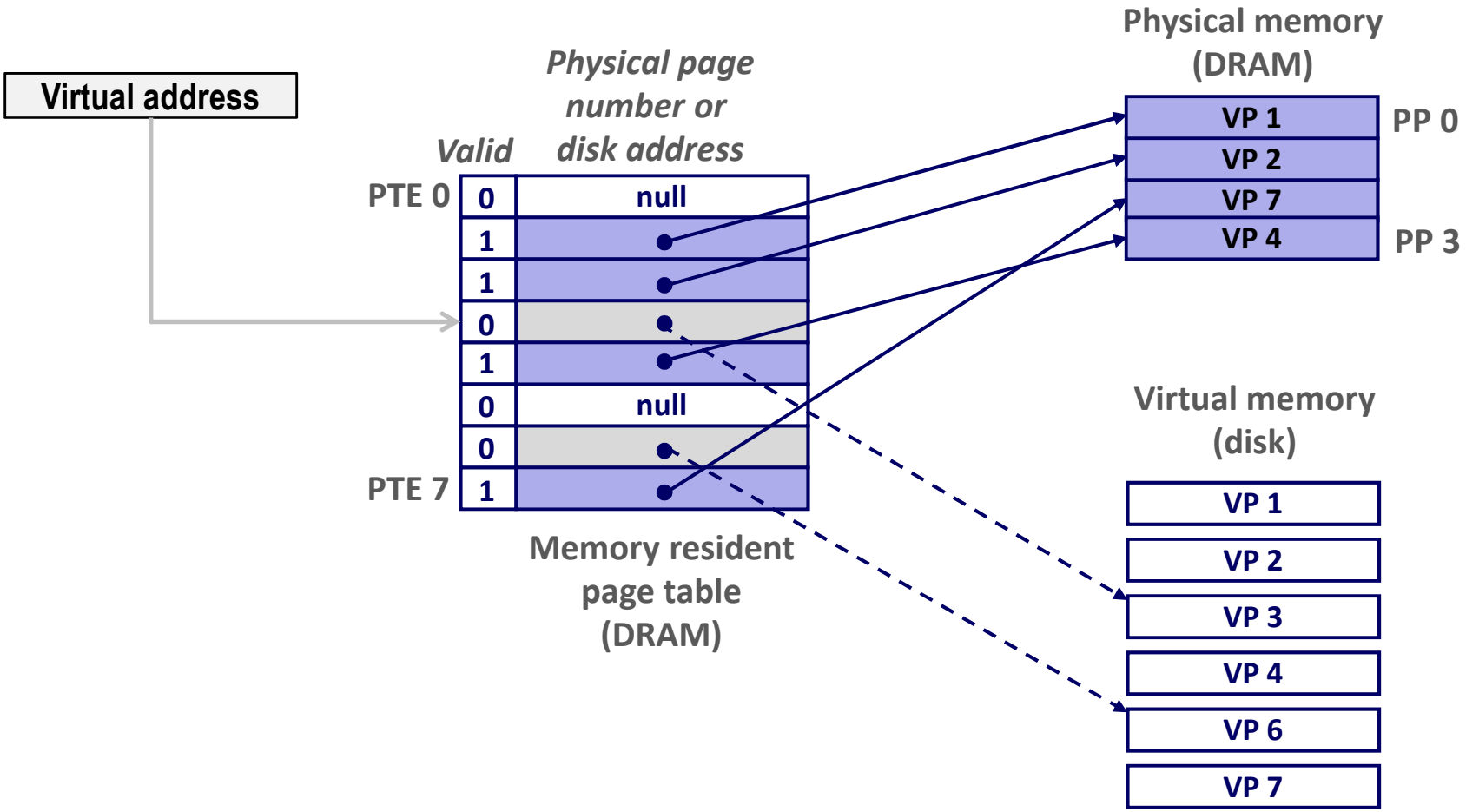
Page Fault

- ***Page fault:***
  - reference to VM word that is not in physical memory (DRAM miss)



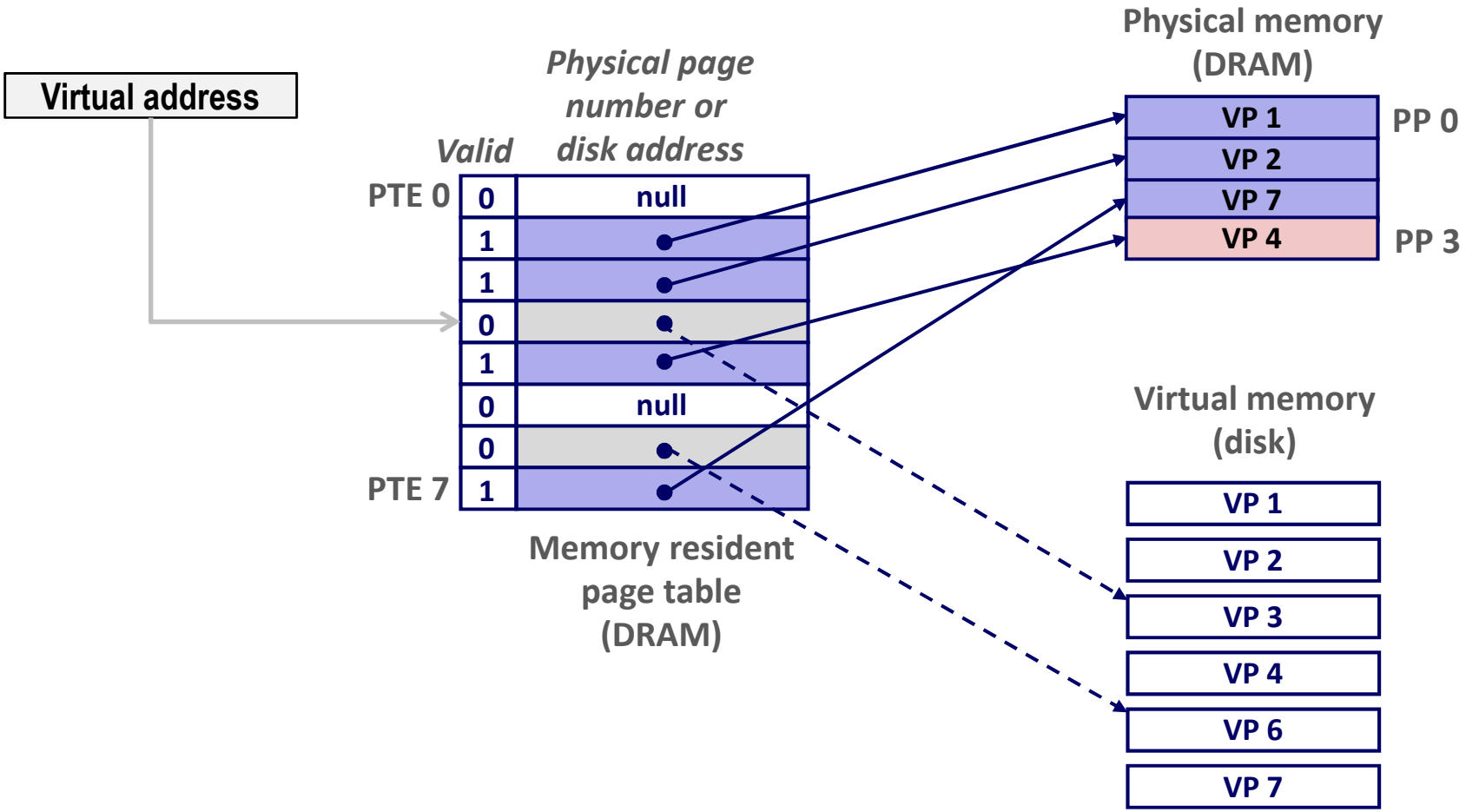
Handling Page Fault

- Page miss causes page fault (an *exception*)



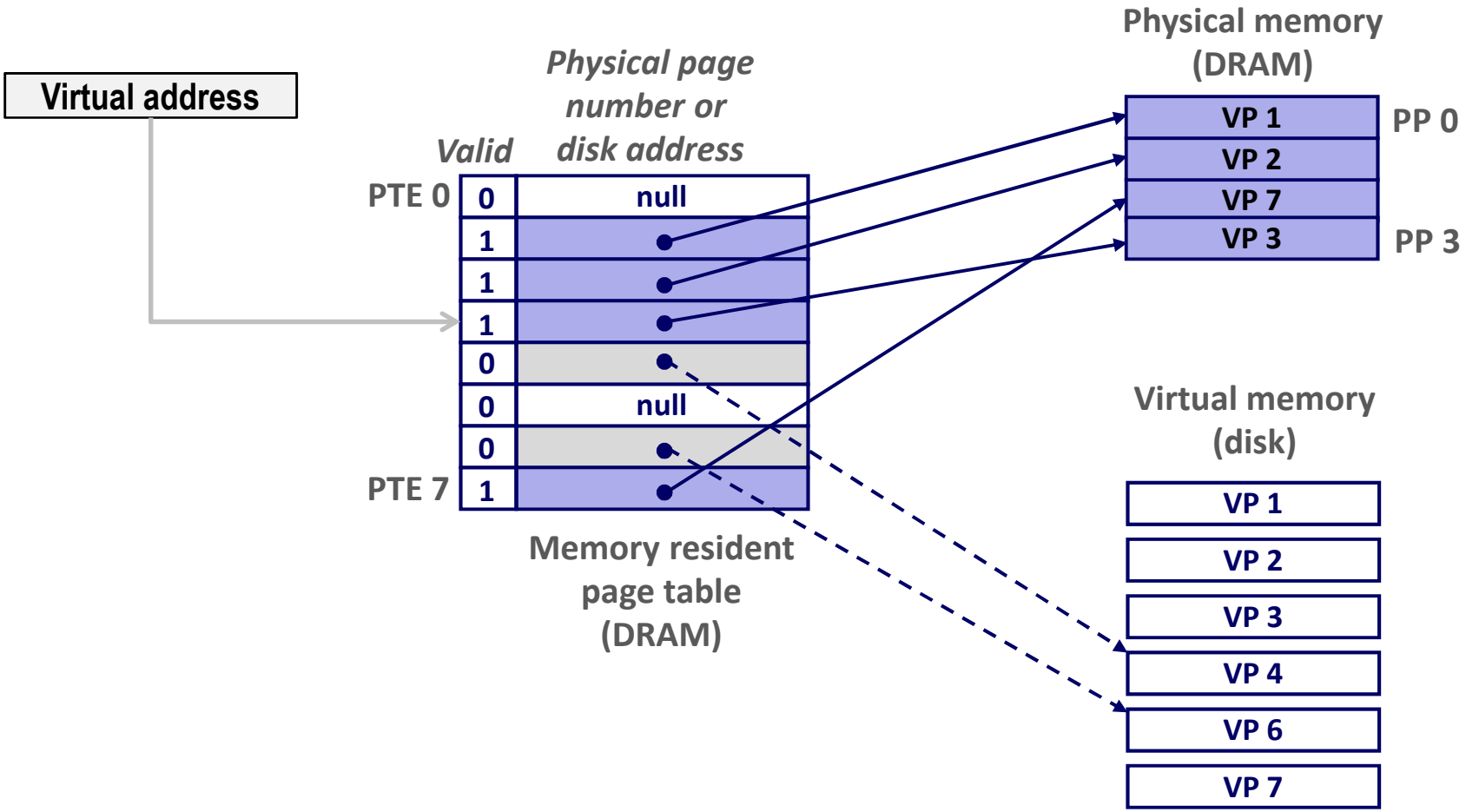
Handling Page Fault

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



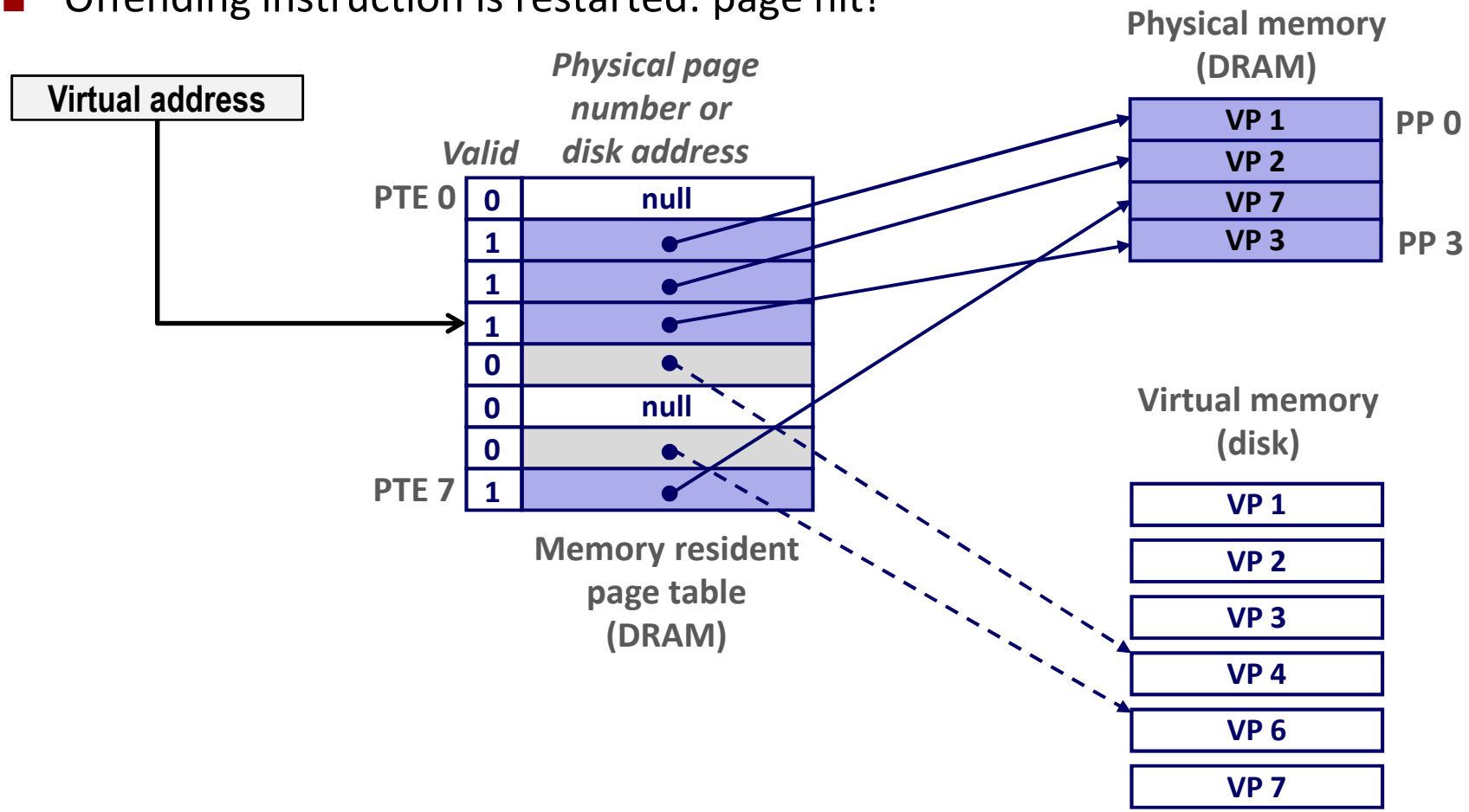
Handling Page Fault

- 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!



Data Representation in Memory

- Memory organization within a process

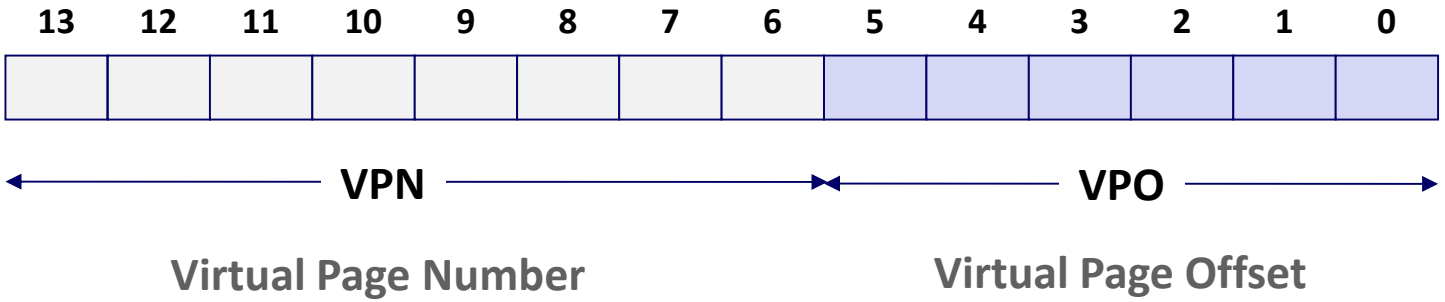
- **Virtual vs. Physical memory**
  - Fundamental Idea and Purpose
  - Page Mapping
  - Address Translation
  - Per-Process Mapping and Protection

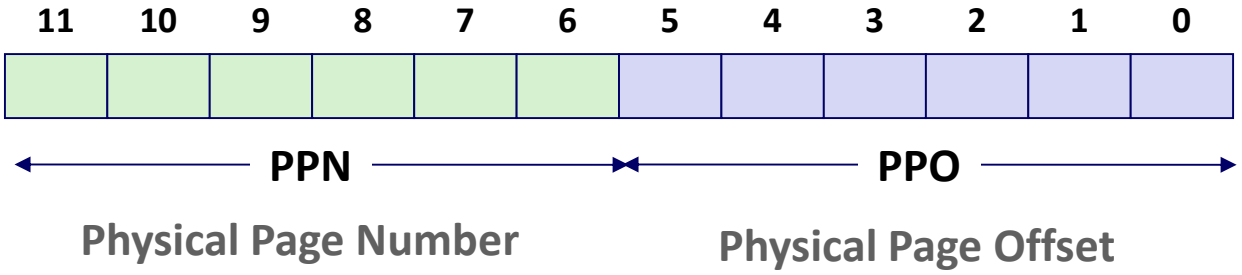
VM Address Translation

- **Virtual Address Space**
  - ***V*** = *{0, 1, …, N–1}*
- **Physical Address Space**
  - ***P*** = *{0, 1, …, M–1}*
- **Address Translation**
  - ***map: V → P U {∅}***
  - For virtual address ***a***:
    - ***map(A) = A'*** if virtual address ***A*** stored in physical address ***A'*** of ***P***
    - ***map(A) = ∅*** if data at virtual address ***A*** is not in physical memory
      - Either invalid or stored on disk

Simple Memory System Example

- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes = 2^6 *(6-bit page offset addr)*





Simple Memory System Page Table

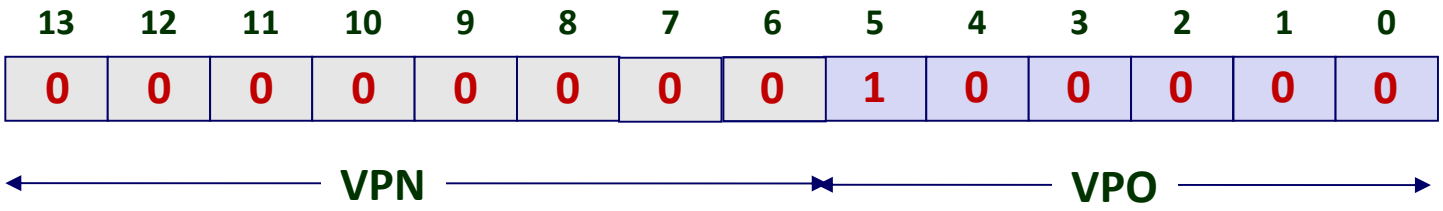
Only show first 16 entries (out of 256)

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

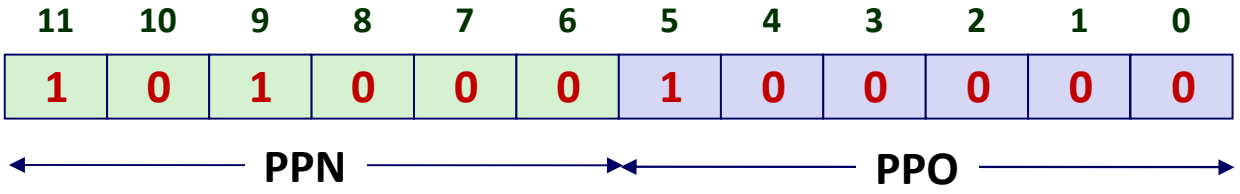
Address Translation Example #1

Virtual Address: 0x0020



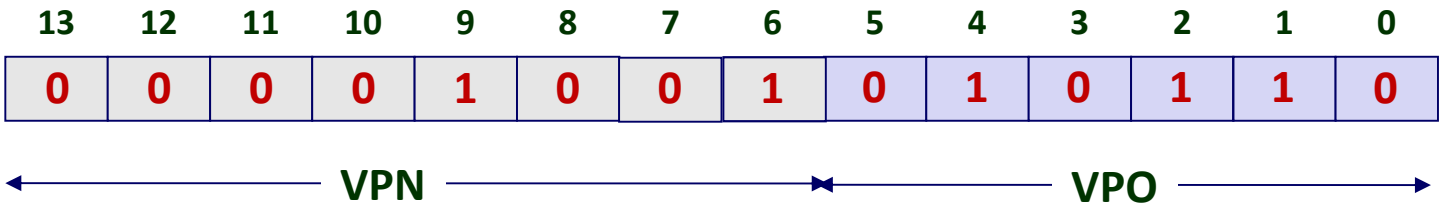
VPN 0x00 Page Table: Valid? Y PPN: 0x28 Location: Main Memory

Physical Address



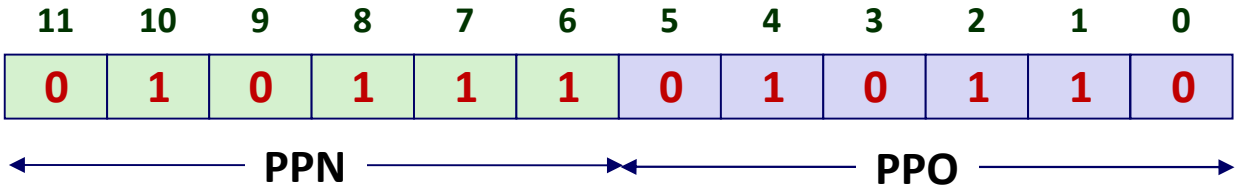
Address Translation Example #2

Virtual Address: 0x0256



VPN 0x09 Page Table: Valid? Y PPN: 0x17 Location: Main Memory

Physical Address



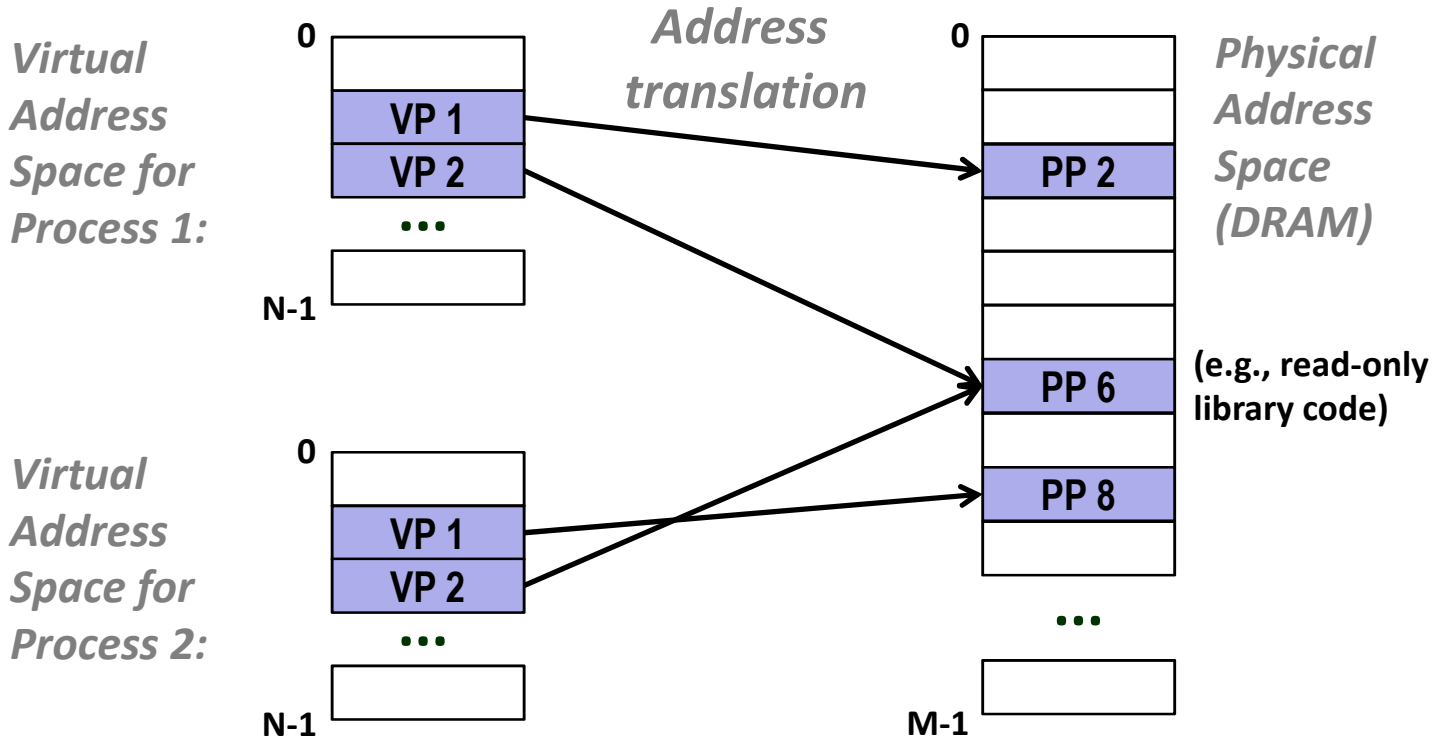
Data Representation in Memory

- Memory organization within a process

- **Virtual vs. Physical memory**
  - Fundamental Idea and Purpose
  - Page Mapping
  - Address Translation
  - Per-Process Mapping and Protection

VM as a Tool for Memory Management

- **Key idea: each process has its own virtual address space**
  - viewed as a simple linear array
  - mapping function scatters addresses through physical memory
- **Can share code and data among processes**
  - Map virtual pages to the same physical page (here: PP 6)



VM as a Tool for Memory Protection

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