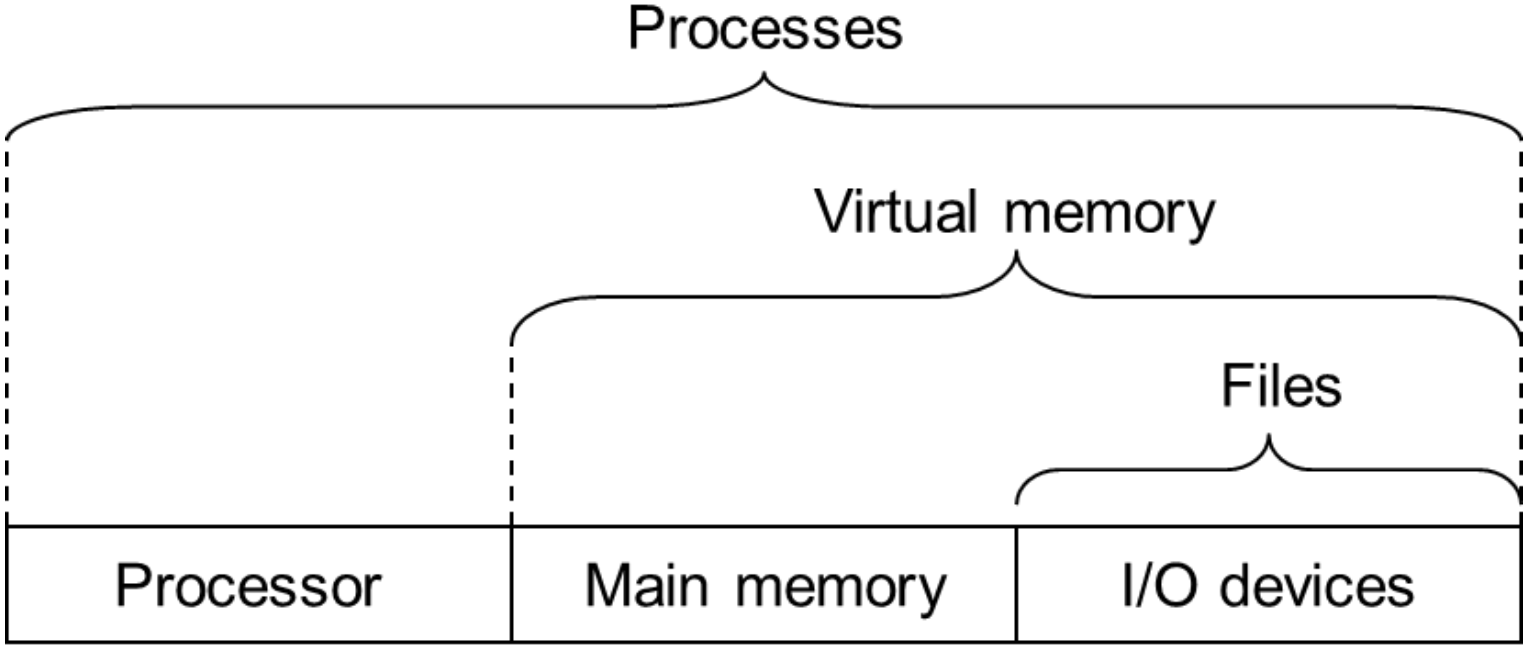
Virtual Memory (I)

Introduction

OS: Three Easy Pieces

- 虚拟化
- 持久化
- 并发



*

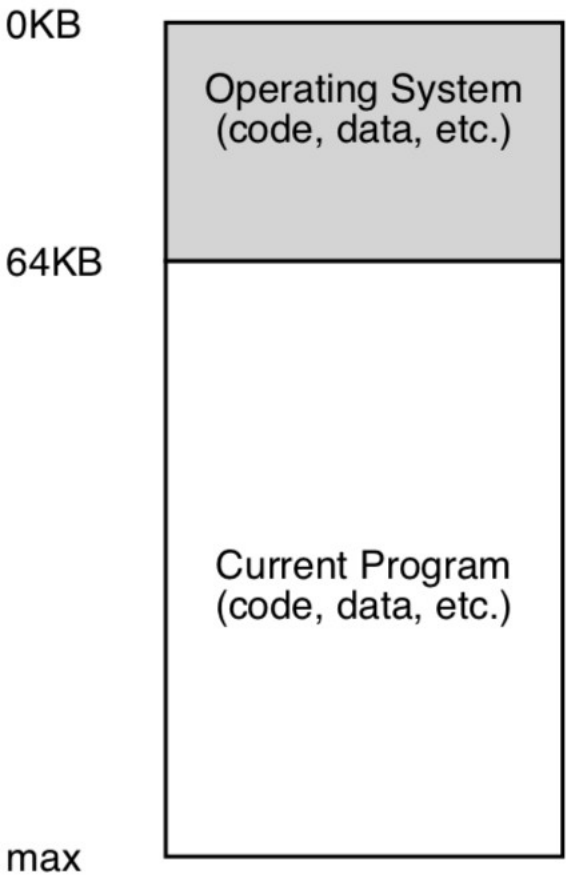
Outline

- VM History
- VM Overview
- Memory API
- VM Discussions

Early Systems

- 直接使用物理地址，没有抽象
- OS 还只是一些库，放在内存地址 0 开始的一段地址空间
- 用户程序从某个地址开始（如 64K），使用剩余的全部内存

- 用户程序都要自己搞定，对 OS 没有多少期待

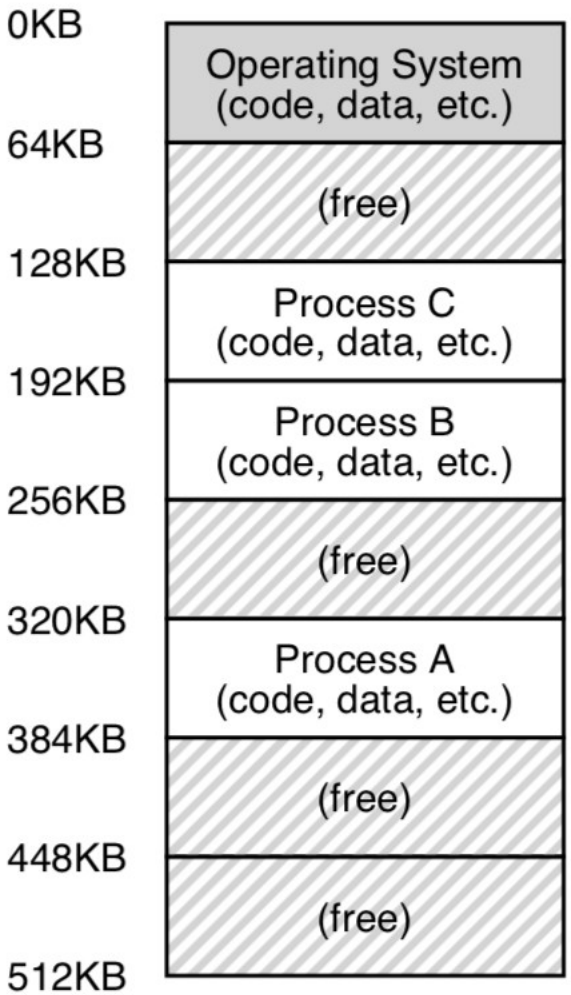


Multiprogramming and Time Sharing

- Multiprogramming
  - 多个程序并发，当有程序在进行 IO 操作时，其他进程可以利用 CPU ，提高 CPU 利用率
- Time Sharing
  - Batch processing 的反应太慢，用户感受不好
    - 例如 debug 过程反馈时间很长
  - 让用户感觉自己独享整个计算机
    - 用户的 interactivity 开始重要

Time Sharing Implementation

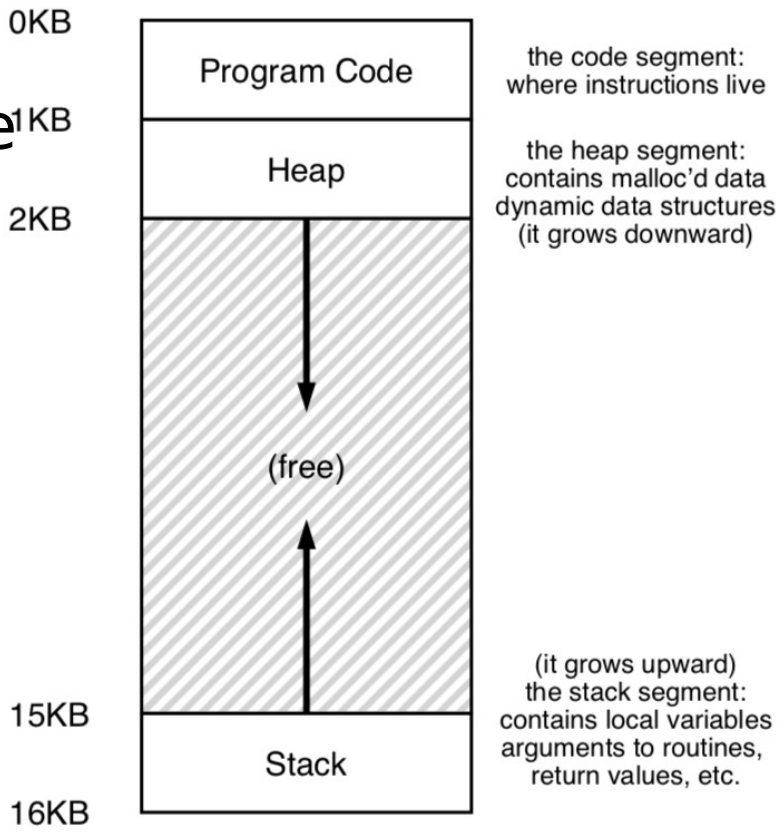
- Method 1. 每次一个程序独占全部内存
  - 好处：内存更大，程序执行更快
  - 缺点：切换任务时，需要有大量的 I/O ，因此性能很差
- Method 2. 每个程序都留在内存中
  - 各占用一部分内存



三个进程共享内存

Multi-program -> Protection

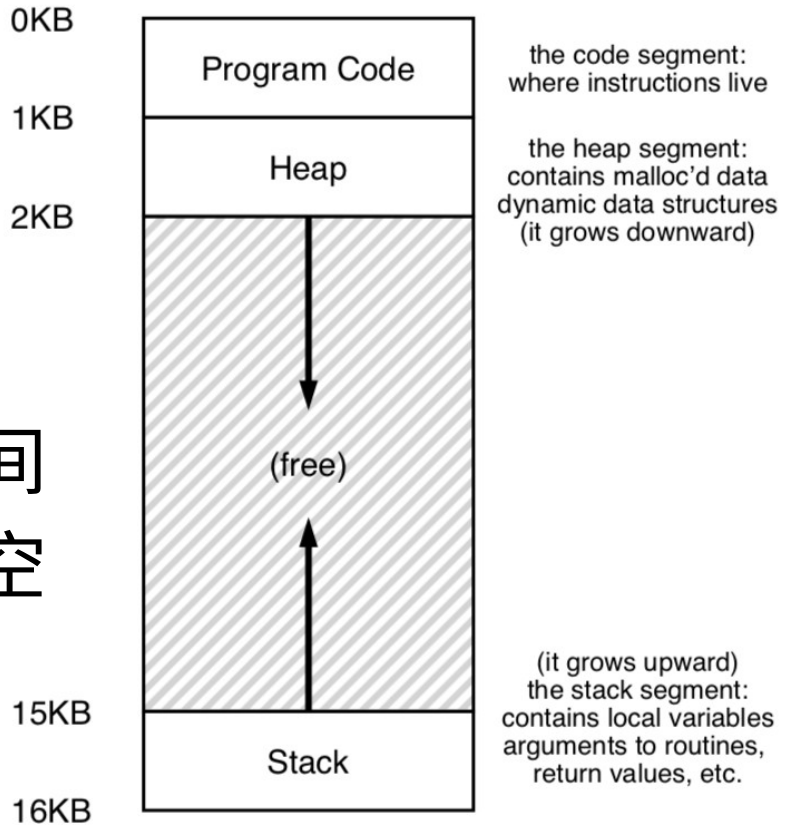
- 为了不让进程之间互相访 (po) 问 (huai) 对方的内存
  - Abstraction: address space
    - 每个进程应该只能看到属于自己的内存空间
    - 那么就不能把物理地址直接给用户
    - 建立虚拟地址空间，以及虚拟地址和物理地址间的映射



Multi-program -> Protection

- Address space
  - Code 区域
  - Stack 区域
  - Heap 区域

  - 看不到其他进程的内存空间
  - 也看不到内核程序的内存空间



VM Overview

Virtual Memory

- Abstraction
  - 不是真实内存，但是方便使用
- Transparent
  - 用户程序感知不到，以为是真的
  - 在物理地址和虚拟地址上都能跑
- Efficient
  - 性能是前提，否则用户不太能接受 VM 机制
- Protection / Isolation
  - 进程无法访问其他进程、内核的地址空间
  - 任意一条指令执行时，不能访问到自己空间以外的地方

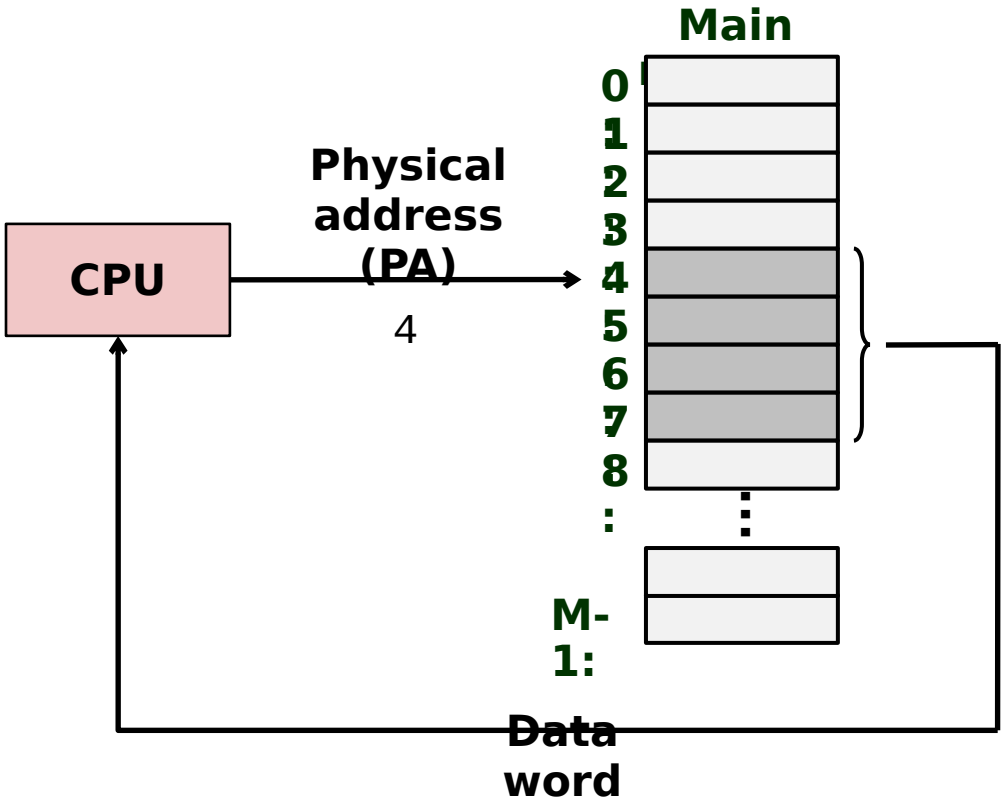
Virtual Memory

- Addresses
  - Virtual address
  - Physical address
  - Address Translation

Physical Addressing

- Attributes of the main memory
  - Organized as an array of M contiguous byte-sized cells
  - Each byte has a unique physical address (PA) started from 0
- physical addressing
  - A CPU uses physical addresses to access memory
- Examples
  - Early PCs, DSP, embedded microcontrollers, and Cray supercomputers

A System Using Physical Addressing



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

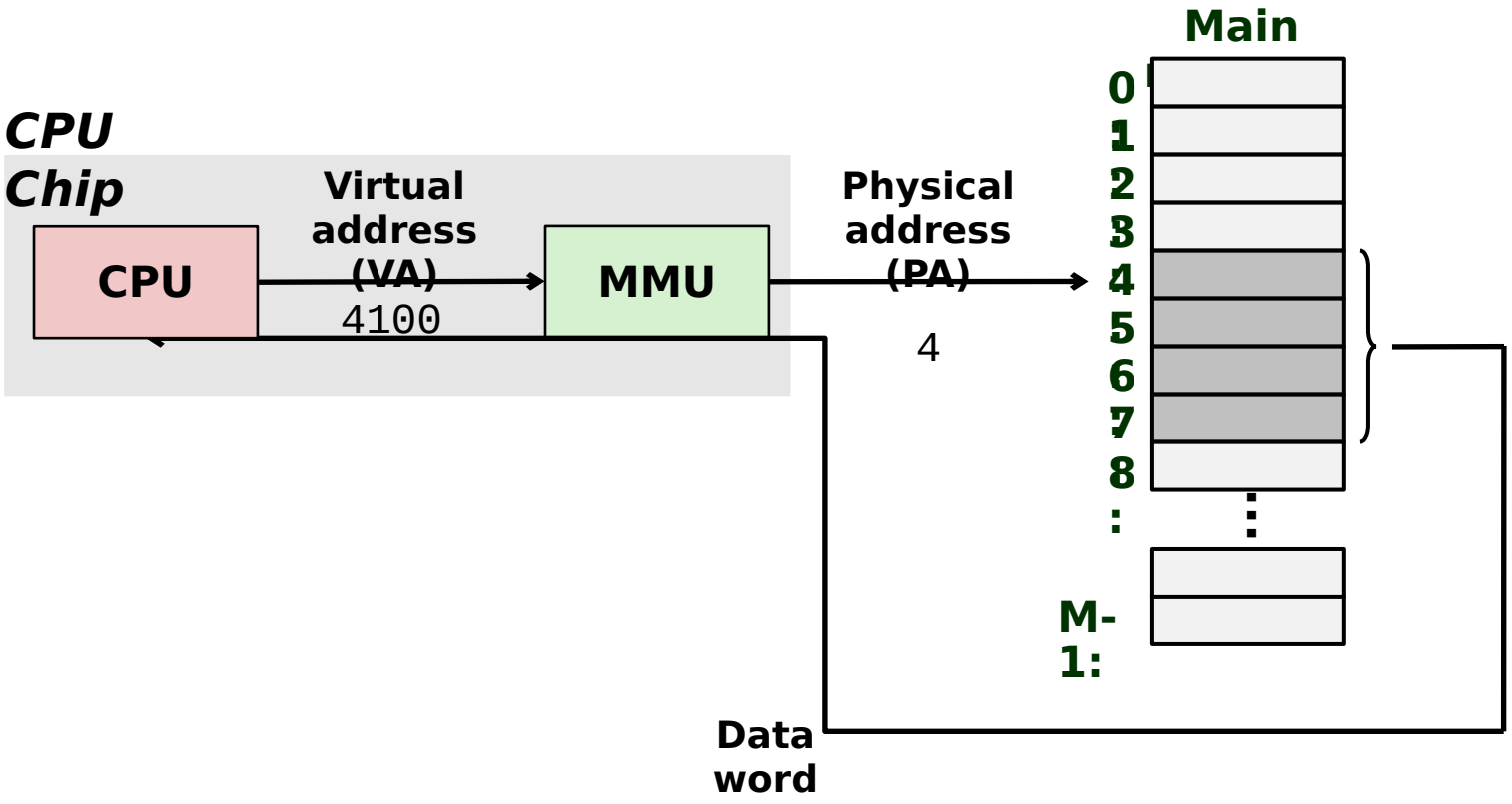
Virtual Addressing

- Virtual addressing
  - the Program accesses main memory by a virtual address (VA)
    - The virtual address is converted to the appropriate physical address by hardware

Virtual Addressing

- Address translation
  - Converting a virtual address to a physical address
  - Requires close cooperation between the CPU hardware and the operating system
    - HW: the memory management unit (MMU)
      - Dedicated hardware on the CPU chip to translate virtual addresses on the fly
    - SW: A look-up table (MMU 可以访问)
      - Stored in main memory
      - Contents are managed by the operating system

A System Using Virtual Addressing



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

Address Space

- N-bit Address Space
  - 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}**
- Each object can now have multiple addresses
  - one physical address, one (or more) virtual addresses

Virtual Memory

- 打印程序中各部分地址
  - 打印出来的都是 virtual address

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(int argc, char *argv[]) {
4      printf("location of code  : %p\n", (void *) main);
5      printf("location of heap  : %p\n", (void *) malloc(1));
6      int x = 3;
7      printf("location of stack : %p\n", (void *) &x);
8      return x;
9  }
```

- 64-bit Mac OS：
  - 先是 code，然后是 heap，再是 stack

```
location of code  : 0x1095afe50
location of heap  : 0x1096008c0
location of stack : 0x7fff691aea64
```

*

课堂练习

- 填写下表中的空格
- K=2^{10}(Kilo), M=2^{20}(Mega), G=2^{30}(Giga), T=2^{40}(Tera), P=2^{50}(Peta), E=2^{60}(Exa)

#virtual address bits (n)	#virtual address (N)	Largest possible virtual address
8		
	64K	
		2^{32}-1=?
	$2^{?}$=256T	G-1
64		*

课堂练习答案

- K=2^{10}(Kilo), M=2^{20}(Mega), G=2^{30}(Giga), T=2^{40}(Tera), P=2^{50}(Peta), E=2^{60}(Exa)

#virtual address bits (n)	#virtual address (N)	Largest possible virtual address
8	256	255
16	64K	64K-1
32	4G	4G-1
48	256T	256T-1
64	16E	16E-1

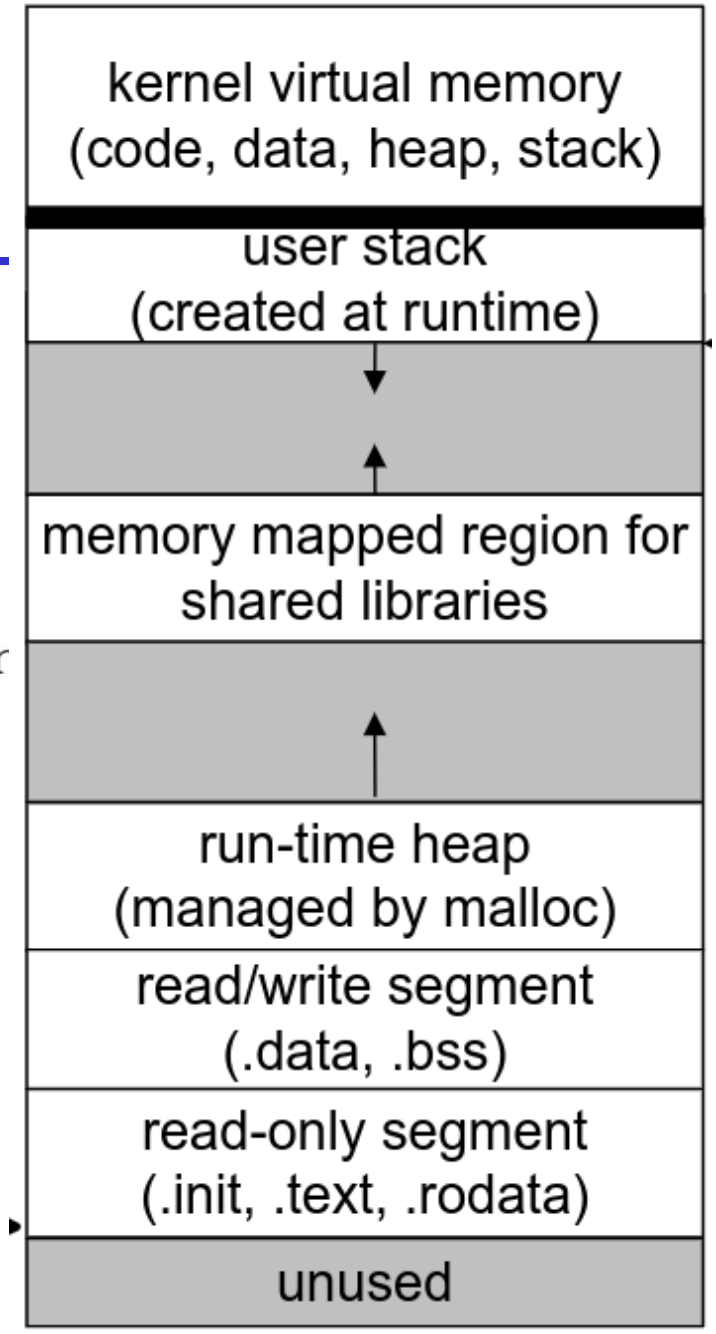
Memory API

Memory API

- Stack:
  - 申请局部变量

```
void func() {
    int x; // declares an ir
    ...
}
```

- 全局变量区
  - 全局变量，静态变量



Memory API

- Heap :
  - Malloc(), free()

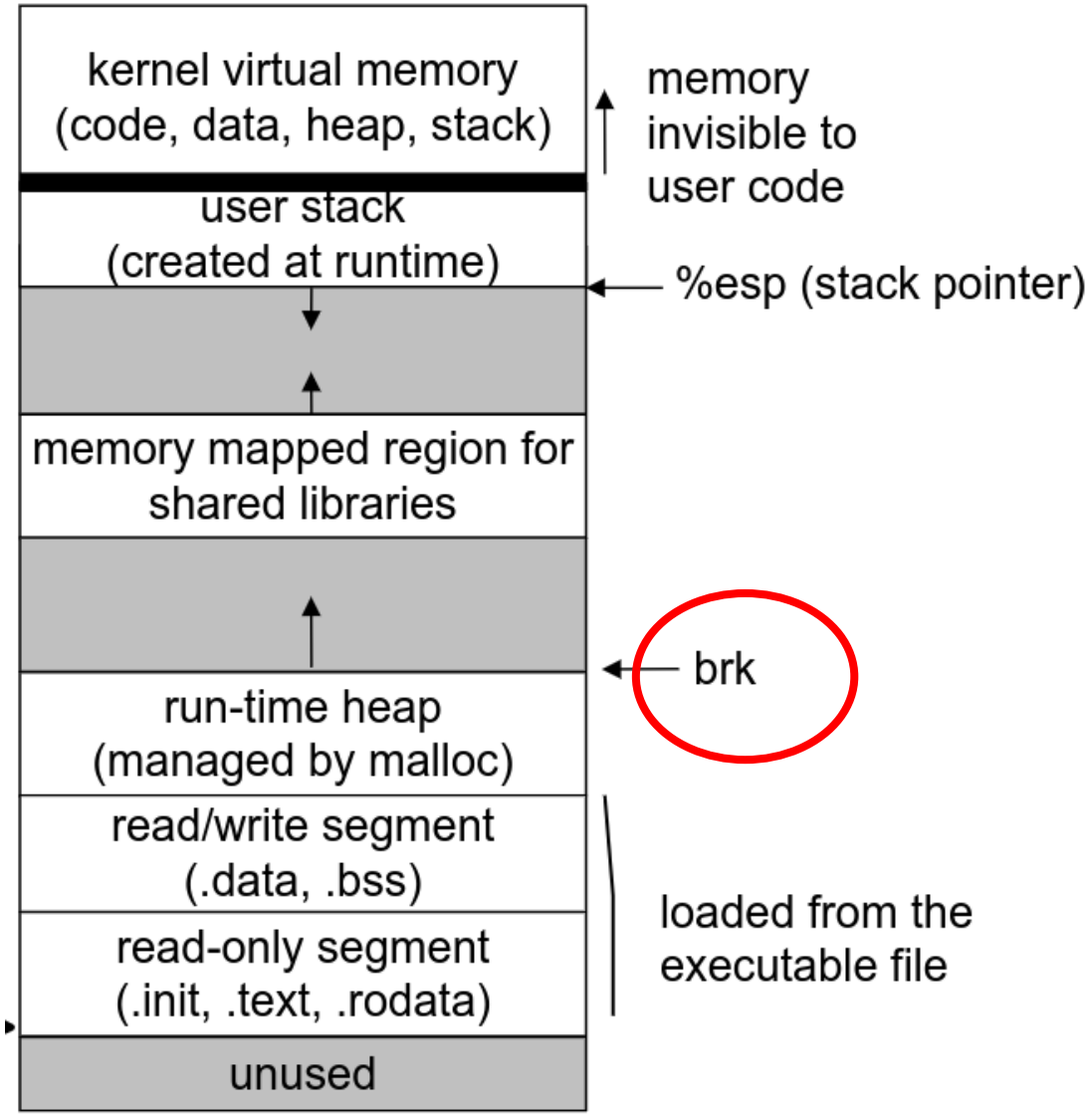
```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

```
#include <stdlib.h>
...
void *malloc(size_t size);
```

Memory API

- Malloc(), free() 不是 system call
  - 只是 library call， C 函数库内部通过一定的结构来保存当前有多少可用内存
  - 如果程序 malloc 的大小超出了库里所留存的空间，那么将首先调用 brk 系统调用来增加可用空间，然后再分配空间
  - free 时，释放的内存并不立即返回给 os，而是保留在内部结构中
- 系统调用通常提供一种最小功能，而库函数通常提供比较复杂的功能。

Memory API


kernel virtual memory
(code, data, heap, stack)
memory
invisible to
user code
user stack
(created at runtime)
%esp (stack pointer)
memory mapped region for
shared libraries
brk
run-time heap
(managed by malloc)
read/write segment
(.data, .bss)
read-only segment
(.init, .text, .rodata)
loaded from the
executable file
unused

Memory API

- brk
  - 改变 heap 区大小的 system call，修改 heap end 指针
  - 不要直接调用，一般都是内存分配的库函数调用
- 此外 malloc 还用到了 mmap 系统调用
  - 后面详细介绍

Memory API

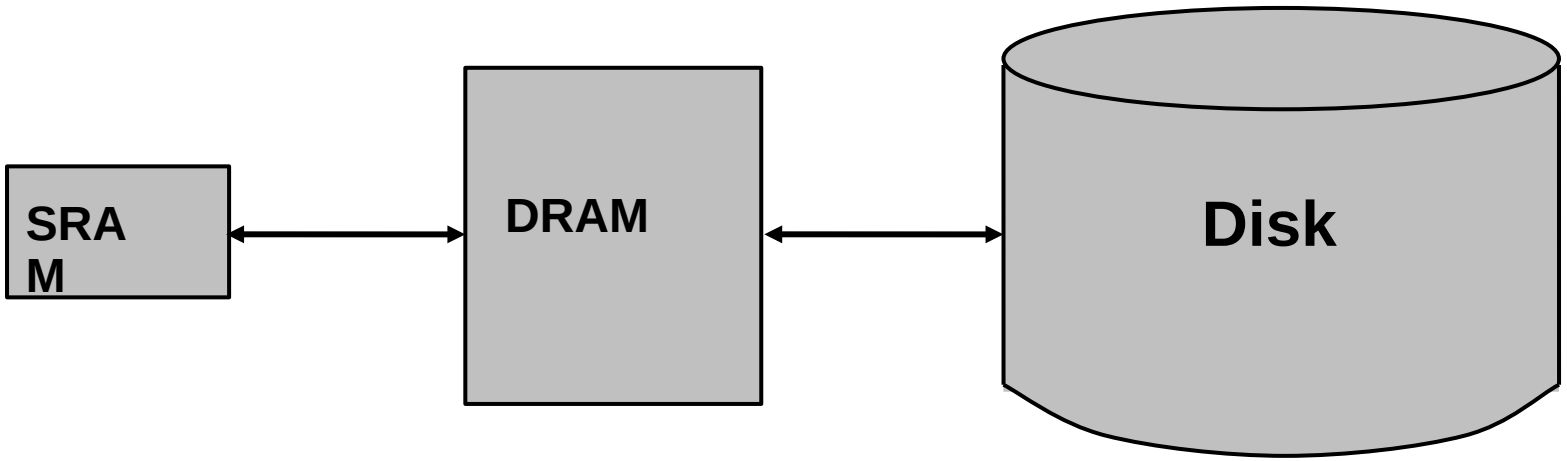
- Calloc()
  - Malloc() + 初始化为 0
- Realloc()
  - 重新分配空间，扩充分配空间
  - 具体操作是找到一个更大的空间，然后把旧的数据复制过去，返回新空间的指针

VM Discussions

Why Virtual Memory (VM)?

- 高效使用 DRAM
  - 实现将 DRAM 用作 address space 上的一个 cache
- 简化内存管理
  - 每个进程都可以看到一个线性的、统一的地址空间
- 隔离地址空间，提供内存保护
  - 进程无法访问其他进程的内存空间
  - 用户进程无法访问内核的内存空间

Using Main Memory as a Cache



Using Main Memory as a Cache

- DRAM vs. disk is more extreme than SRAM vs. DRAM
  - Access latencies:
    - DRAM ~10X slower than SRAM
    - Disk ~100,000X slower than DRAM
  - 一旦 DRAM Cache 未命中，代价巨大

Using Main Memory as a Cache

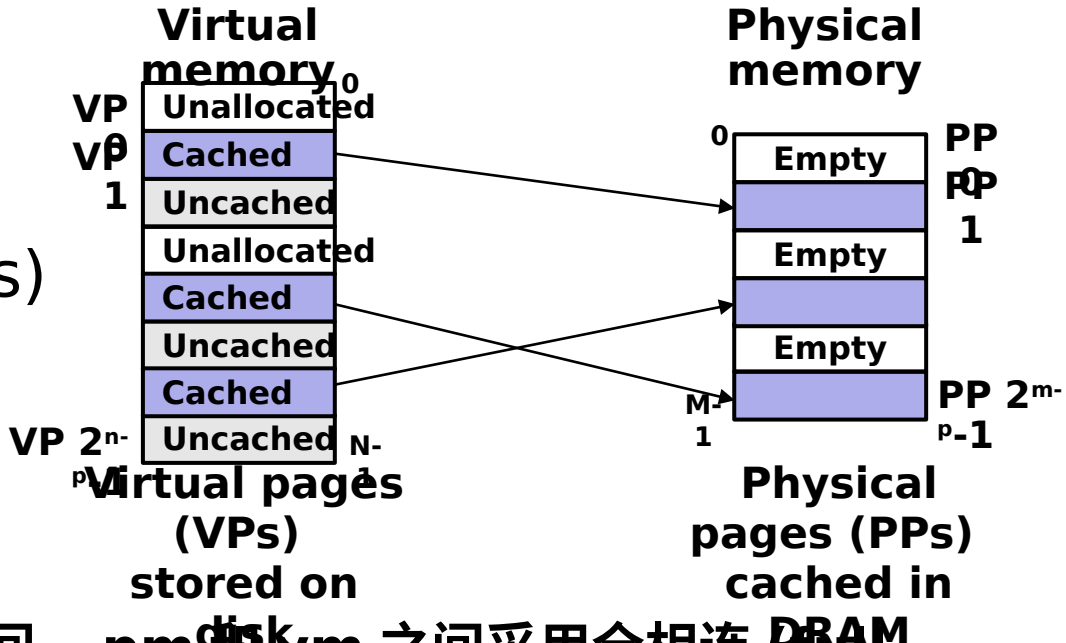
- Virtual memory
  - Organized as an array of contiguous byte-sized cells stored on disk conceptually
  - Each byte has a unique virtual address that serves as an index into the array
  - The contents of the array on disk are cached in main memory

Design Considerations

- 缓存单元大小 (line size)? (Large vs. Small)
  - Large, since disk better at transferring large blocks

Page

- The data on disk is partitioned into pages
  - Serve as the transfer units between the disk and the main memory
  - virtual pages (VPs)
  - physical pages (PPs)
    - or page frames



page 通常在 4KB-2MB 之间，pm 和 vm 之间采用全相连 (full associative)

Page Attributes

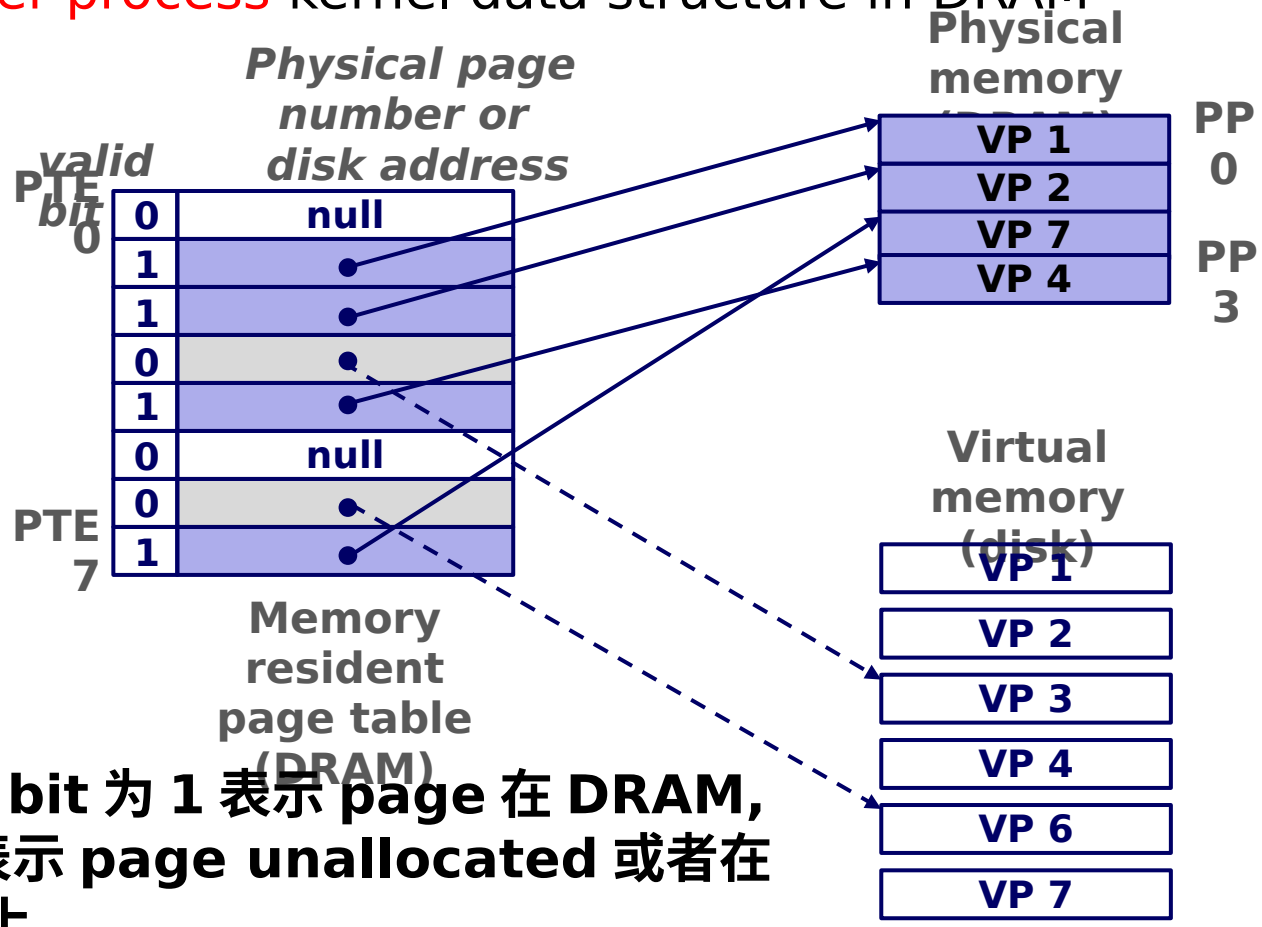
- Unallocated:
  - Pages that have not yet been allocated (or created) by the VM system
  - Do not have any data associated with them
  - Do not occupy any space on disk or physical memory

Page Attributes

- Cached:
  - Allocated pages that are currently cached in physical memory
- Uncached:
  - Allocated pages that are not cached in physical memory

Page Table

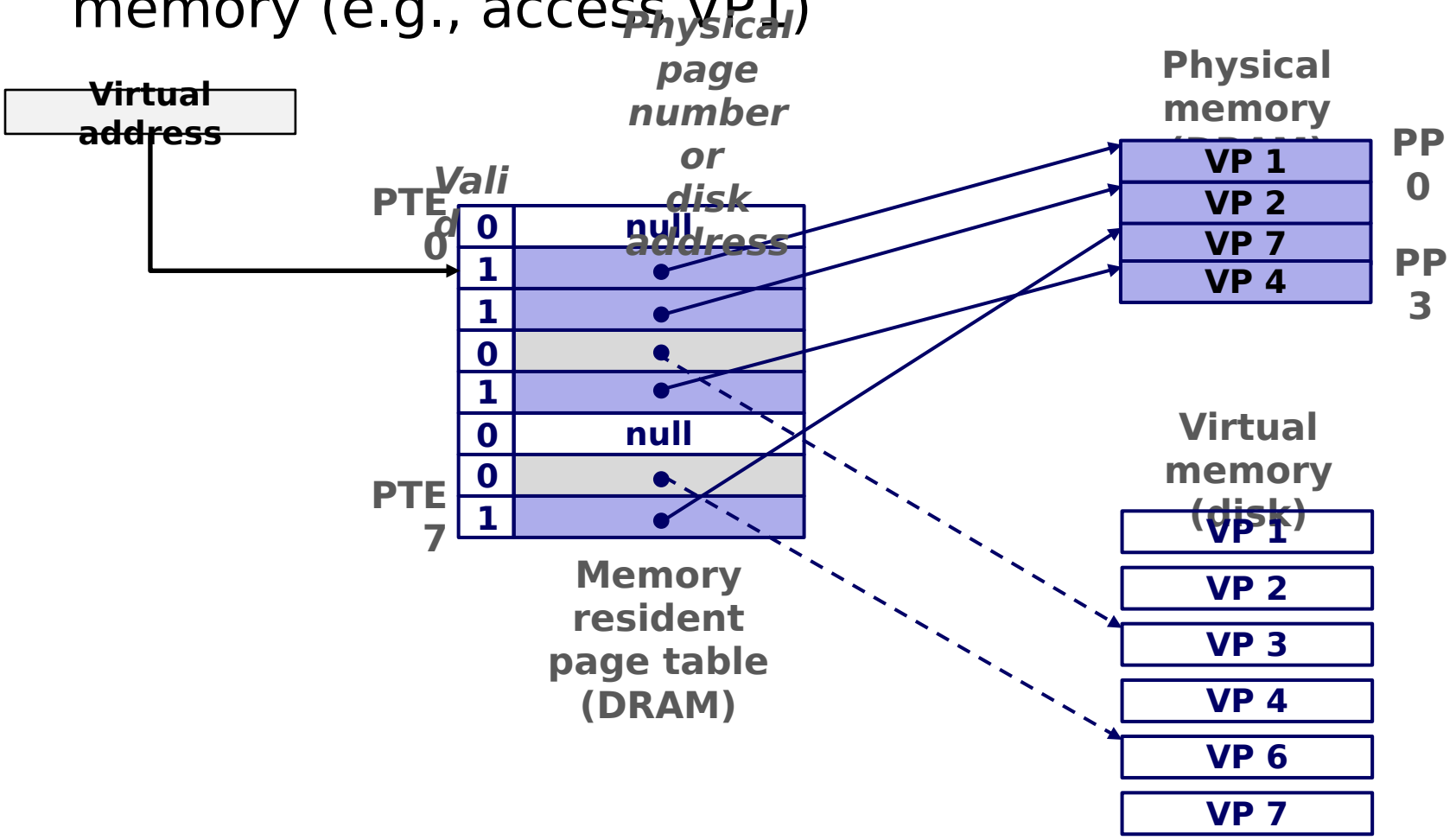
- A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages
  - Per-process kernel data structure in DRAM



valid bit 为 1 表示 page 在 DRAM，为 0 表示 page unallocated 或者在 disk 上

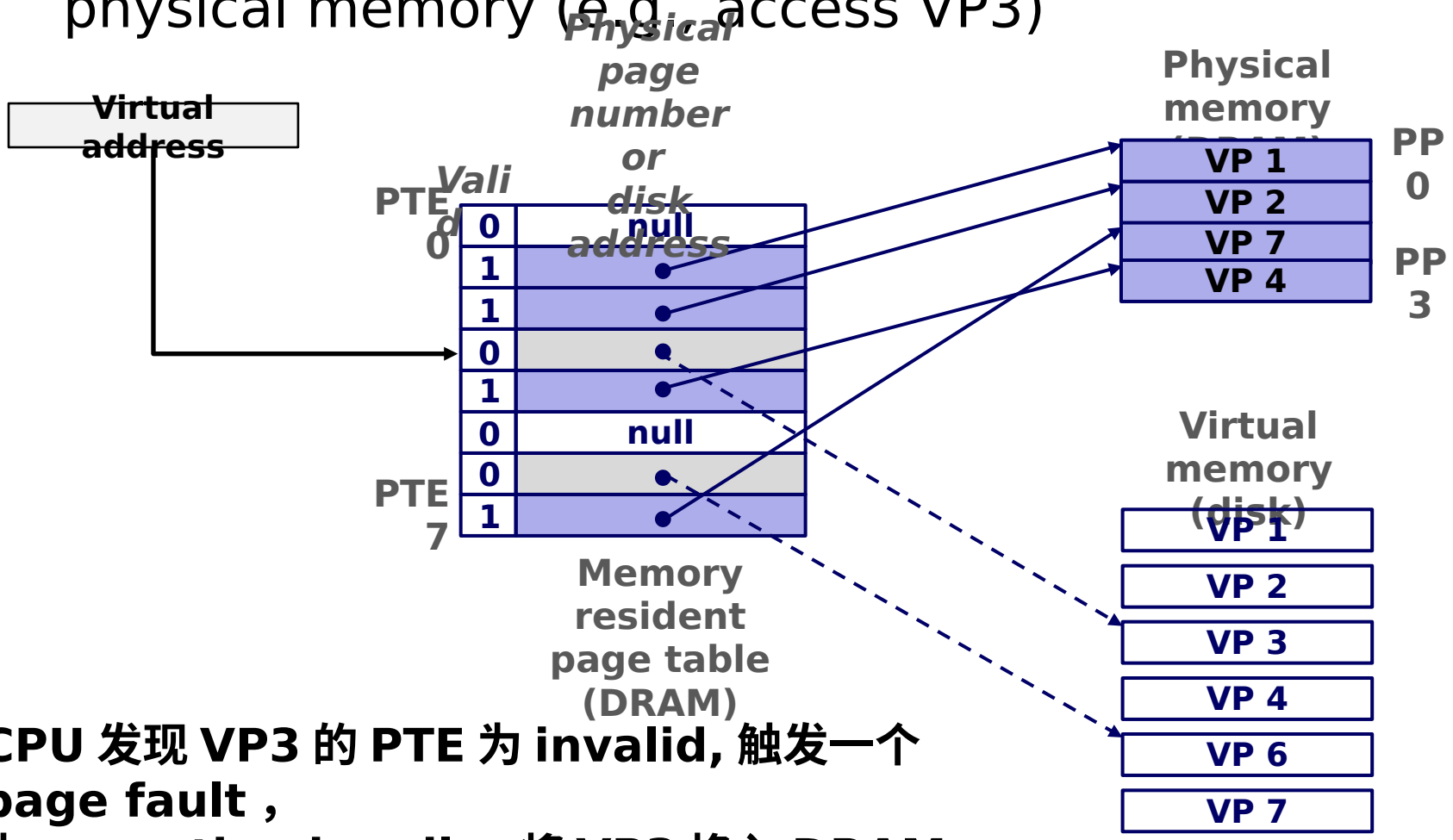
Page Hit

- *Page hit:* reference to a VM word that is in physical memory (e.g., access VP1)



Page Fault

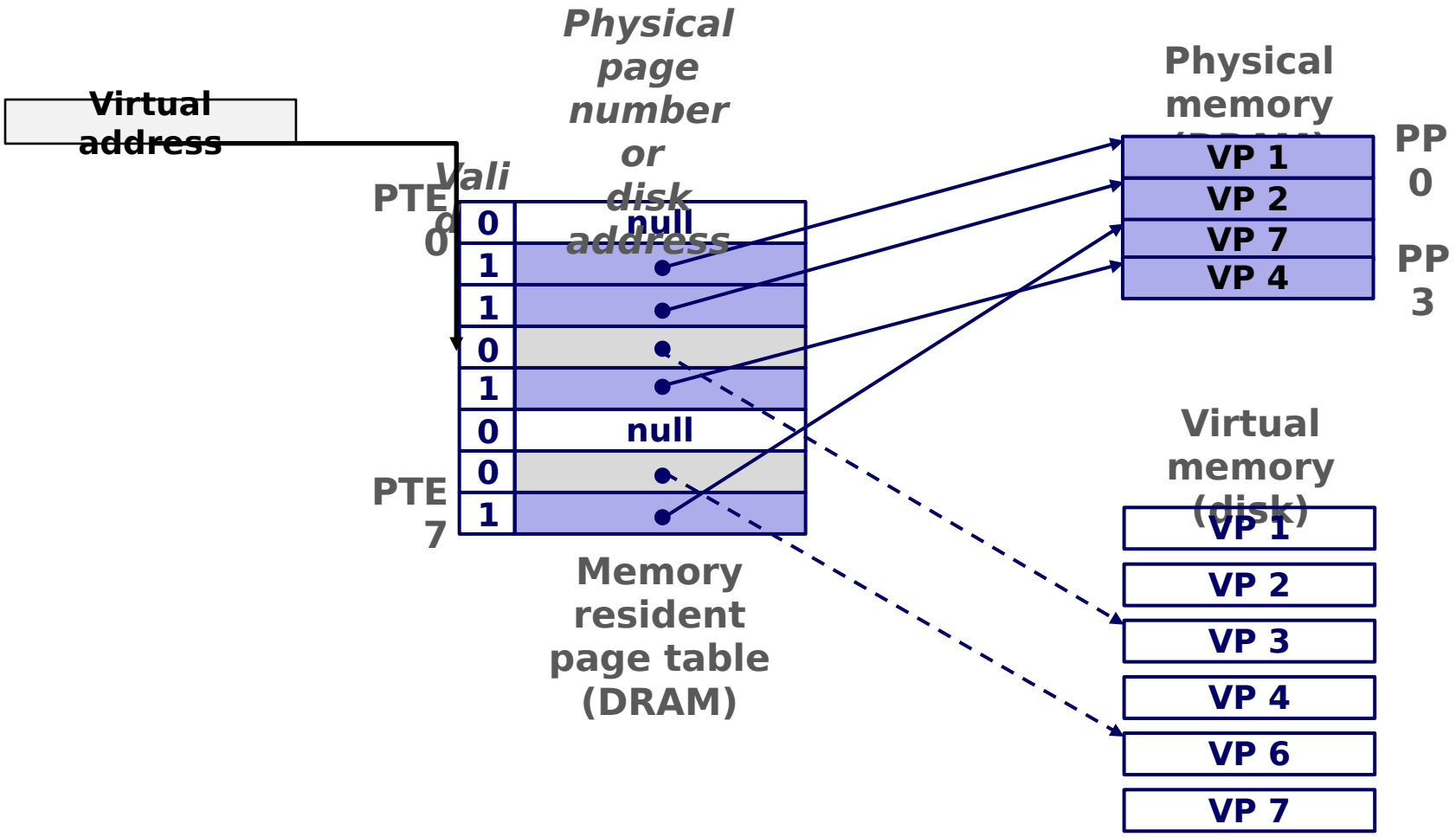
- *Page fault:* reference to VM word that is not in physical memory (e.g., access VP3)



CPU 发现 VP3 的 PTE 为 invalid, 触发一个 page fault ,
由 exception handler 将 VP3 换入 DRAM

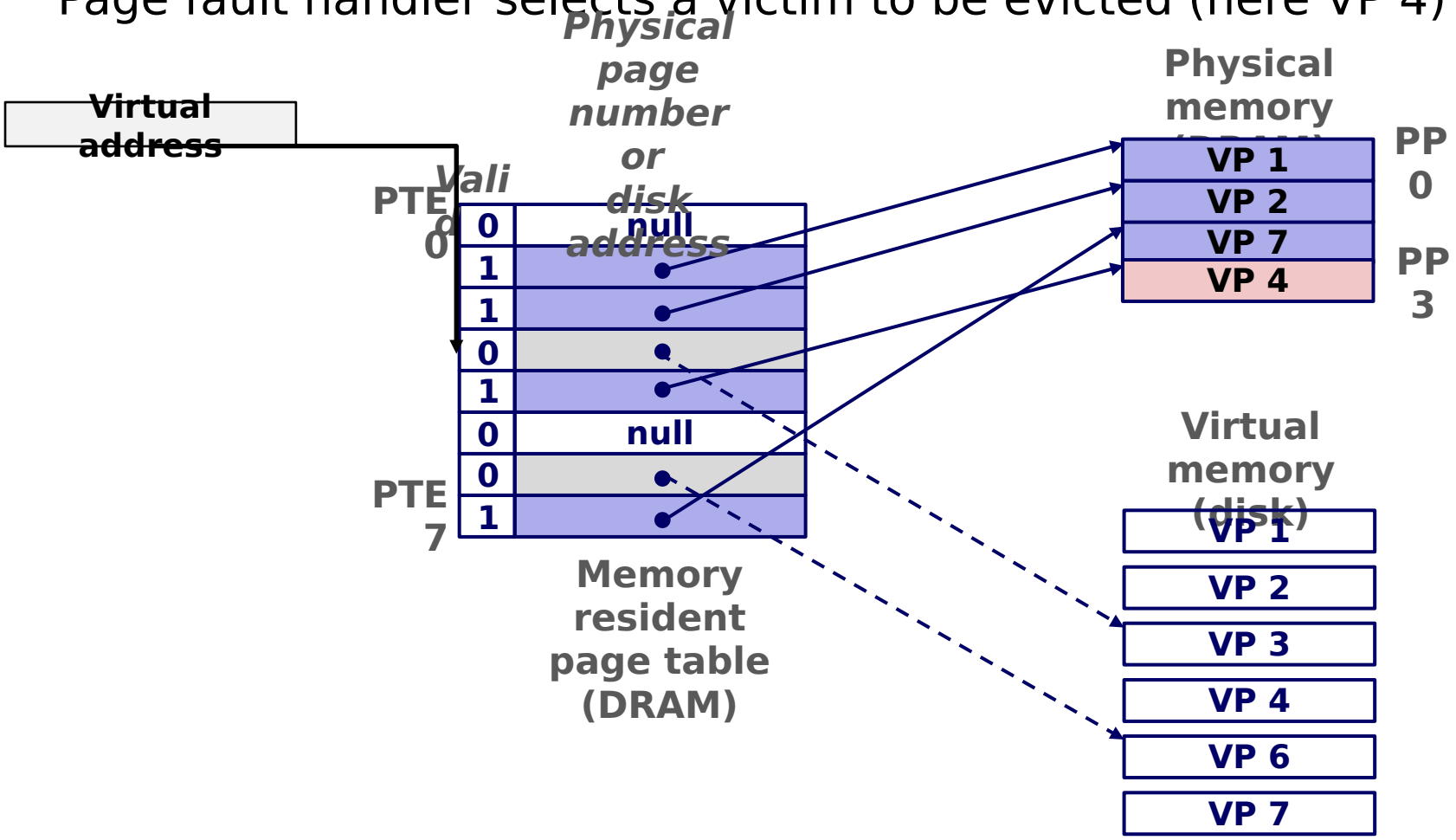
Handling Page Fault

- Page miss causes page fault (an exception)



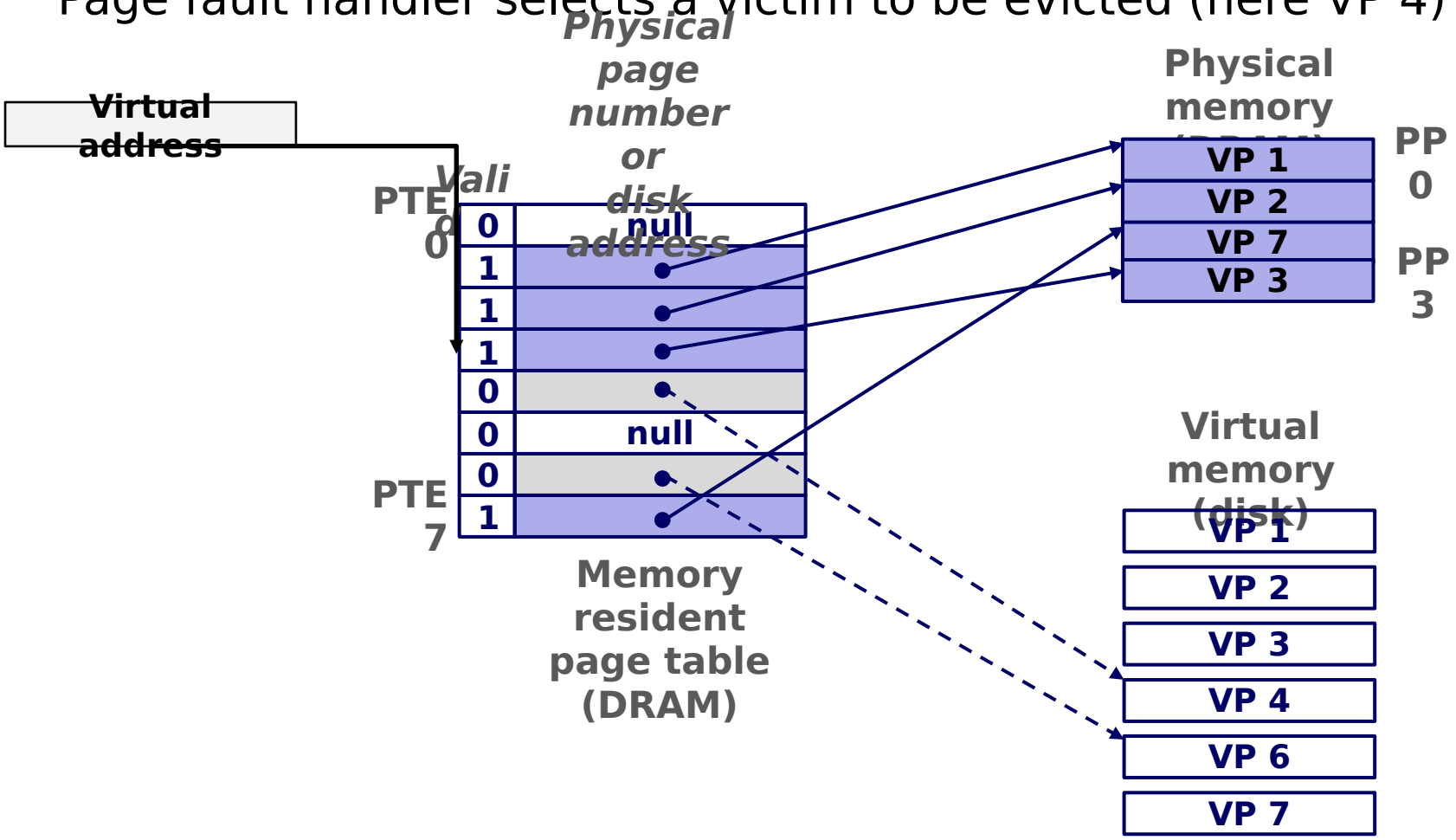
Handling Page Fault

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



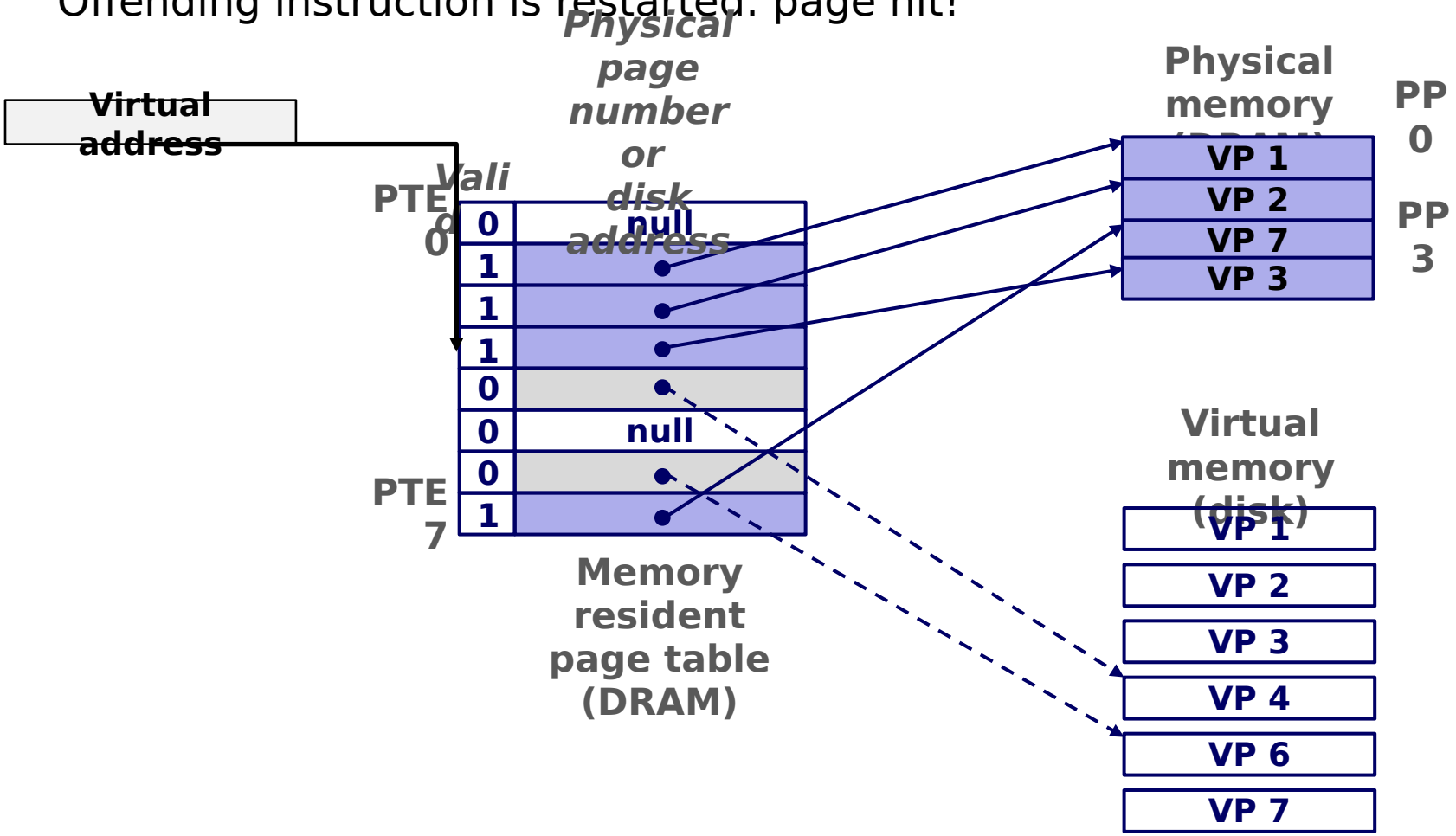
Handling Page Fault

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



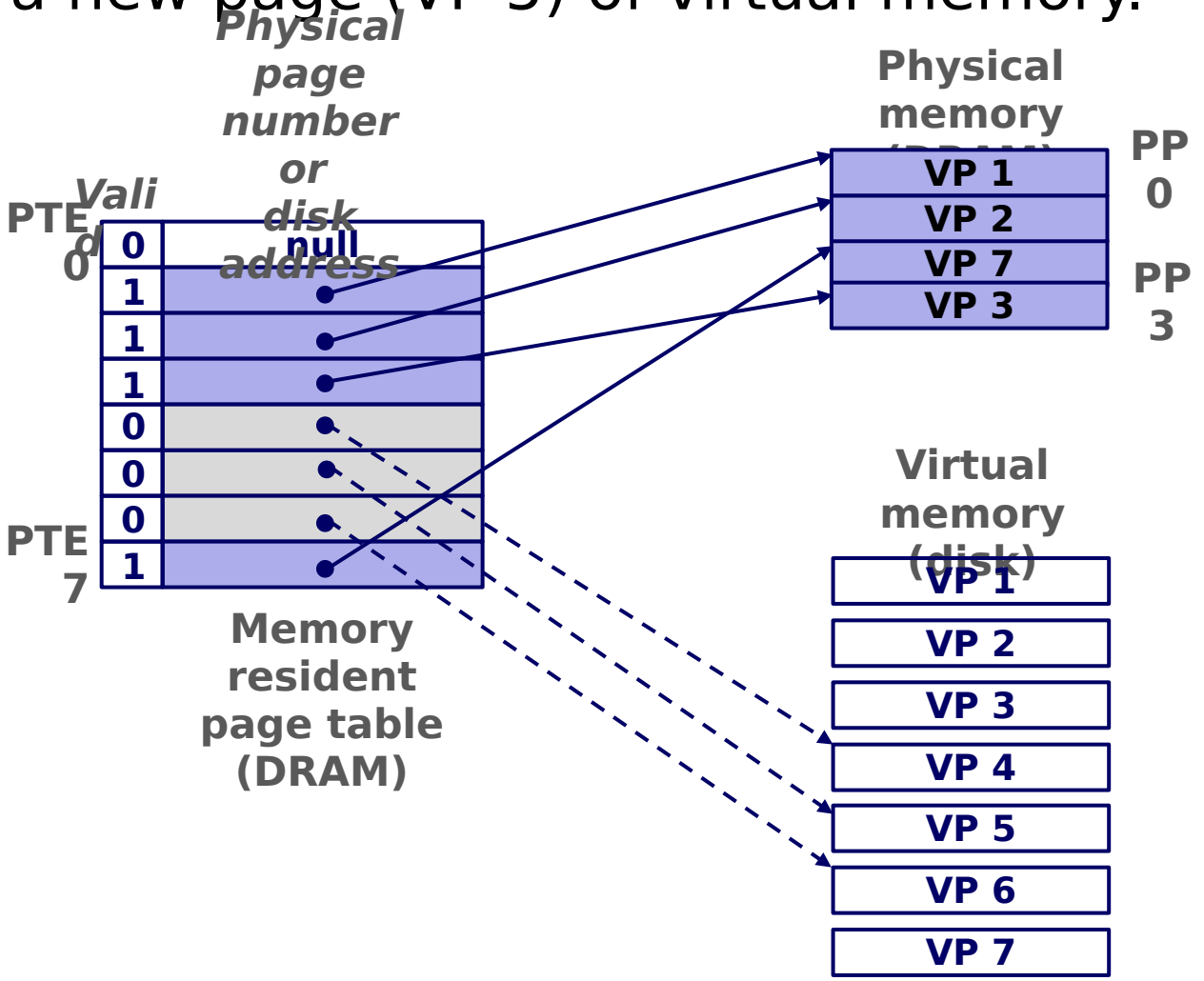
Handling Page Fault

- 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!

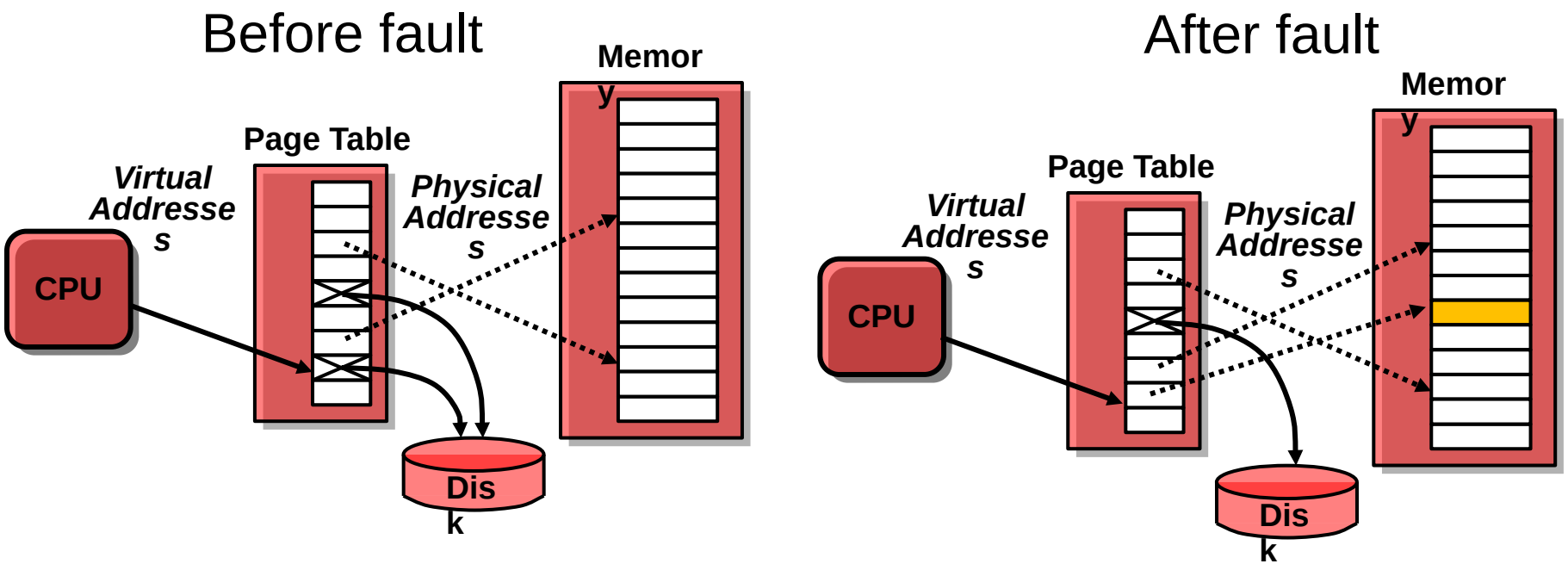


Allocating Pages

- Allocating a new page (VP 5) of virtual memory.



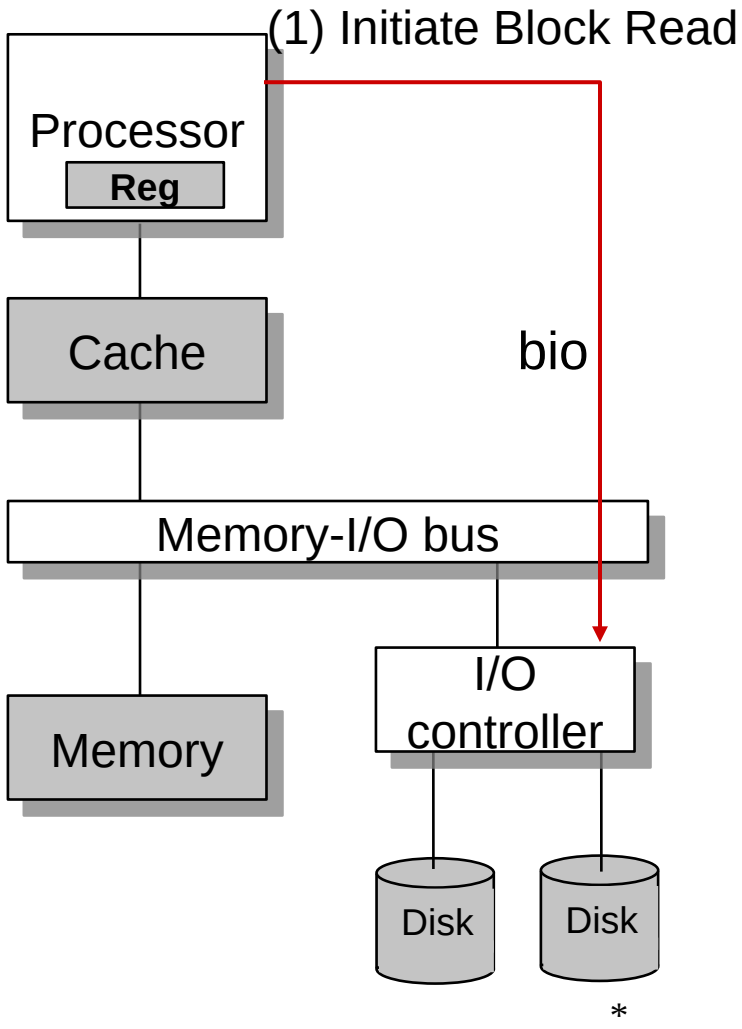
Page Faults



在 page fault 的处理过程中发生了什么?

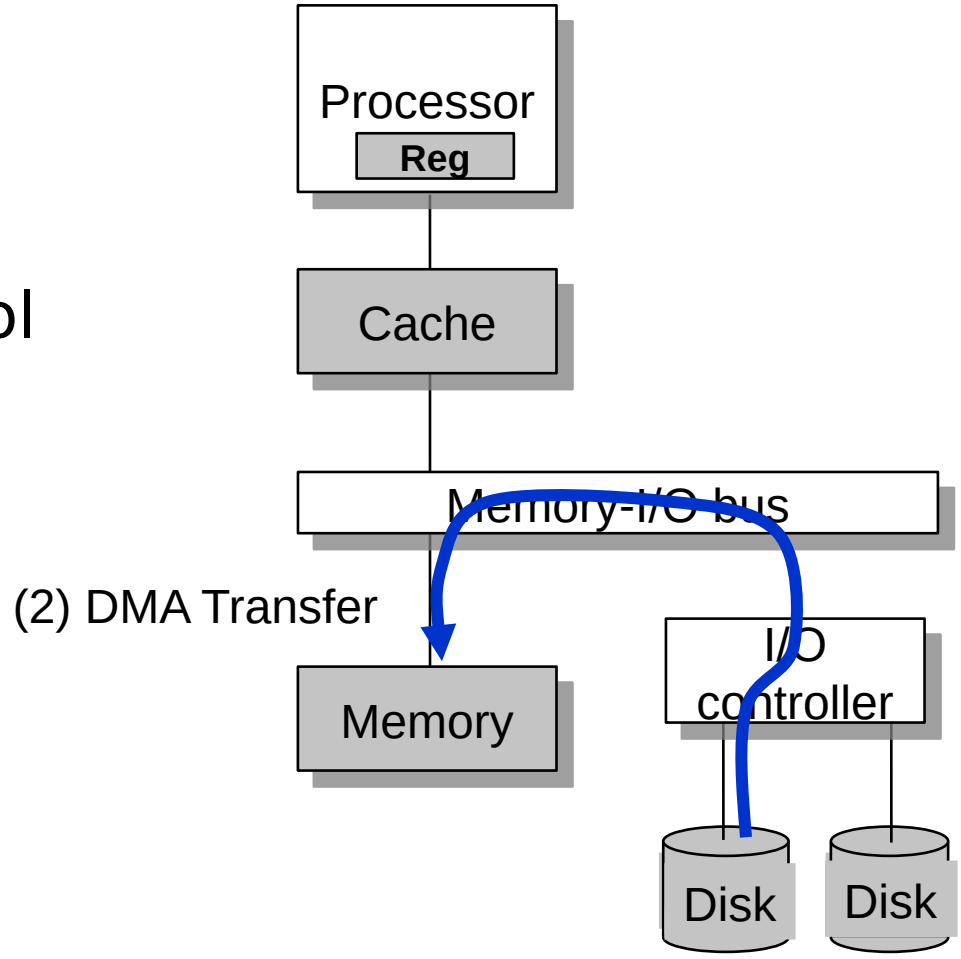
Servicing a Page Fault

- Processor Signals I/O Controller
  - Read block (page) of length P starting at disk address X and store starting at memory address Y



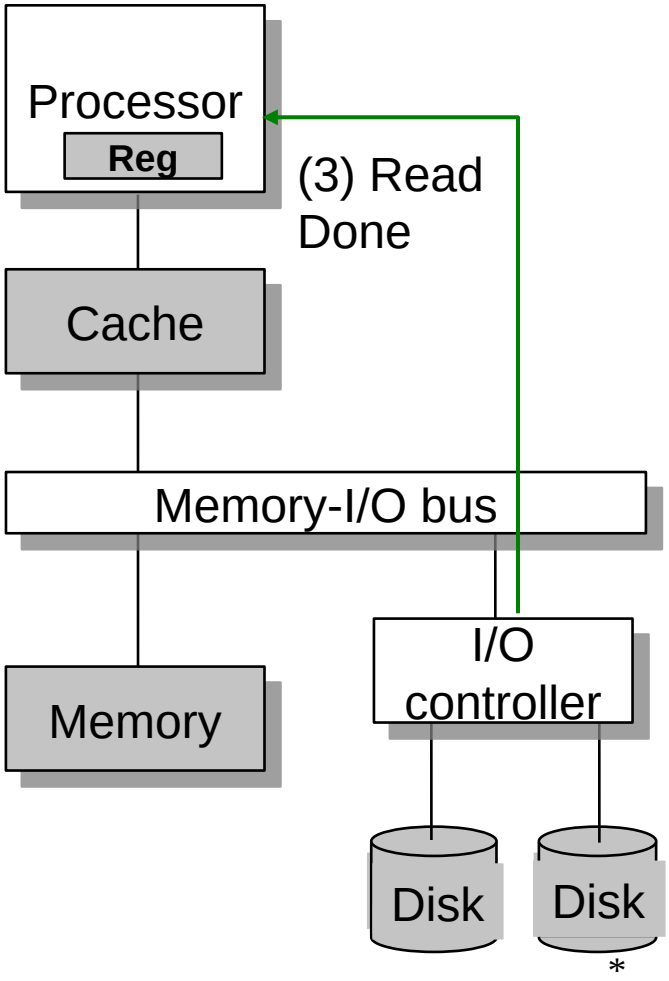
Servicing a Page Fault

- Read Occurs
  - Direct Memory Access (DMA) under the control of I/O controller



Servicing a Page Fault

- I / O Controller Signals Completion to CPU
  - Interrupt processor
  - OS resumes suspended process



Locality to the Rescue Again!

- Virtual memory works because of data locality
- Temporal locality:
  - 一个刚刚被访问的 page 在不久的将来还会被访问
- Spatial locality:
  - 一个 page 刚刚被访问，则它的邻居也将被访问

DRAM cache 利用了哪种 locality?

Locality to the Rescue Again!

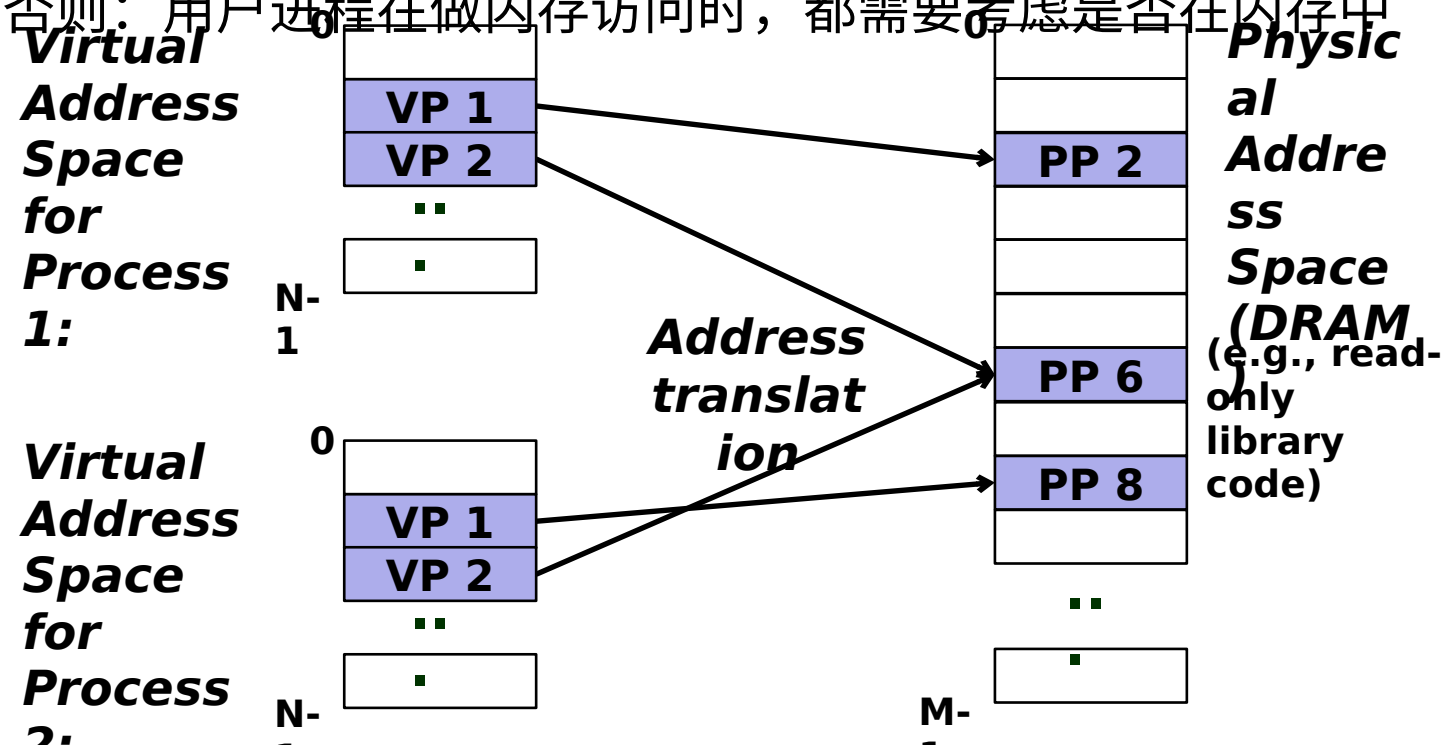
- Working set: the active pages of a process
- If (working set size < main memory size)
  - Good performance for one process after compulsory misses
- If (SUM(working set sizes) > main memory size)
  - Thrashing(颠簸): Performance meltdown where pages are swapped (copied) in and out continuously

Why Virtual Memory (VM)?

- 高效使用 DRAM
  - 实现将 DRAM 用作 address space 上的一个 cache
- 简化内存管理
  - 每个进程都可以看到一个线性的、统一的地址空间
- 隔离地址空间，提供内存保护
  - 进程无法访问其他进程的内存空间
  - 用户进程无法访问内核的内存空间

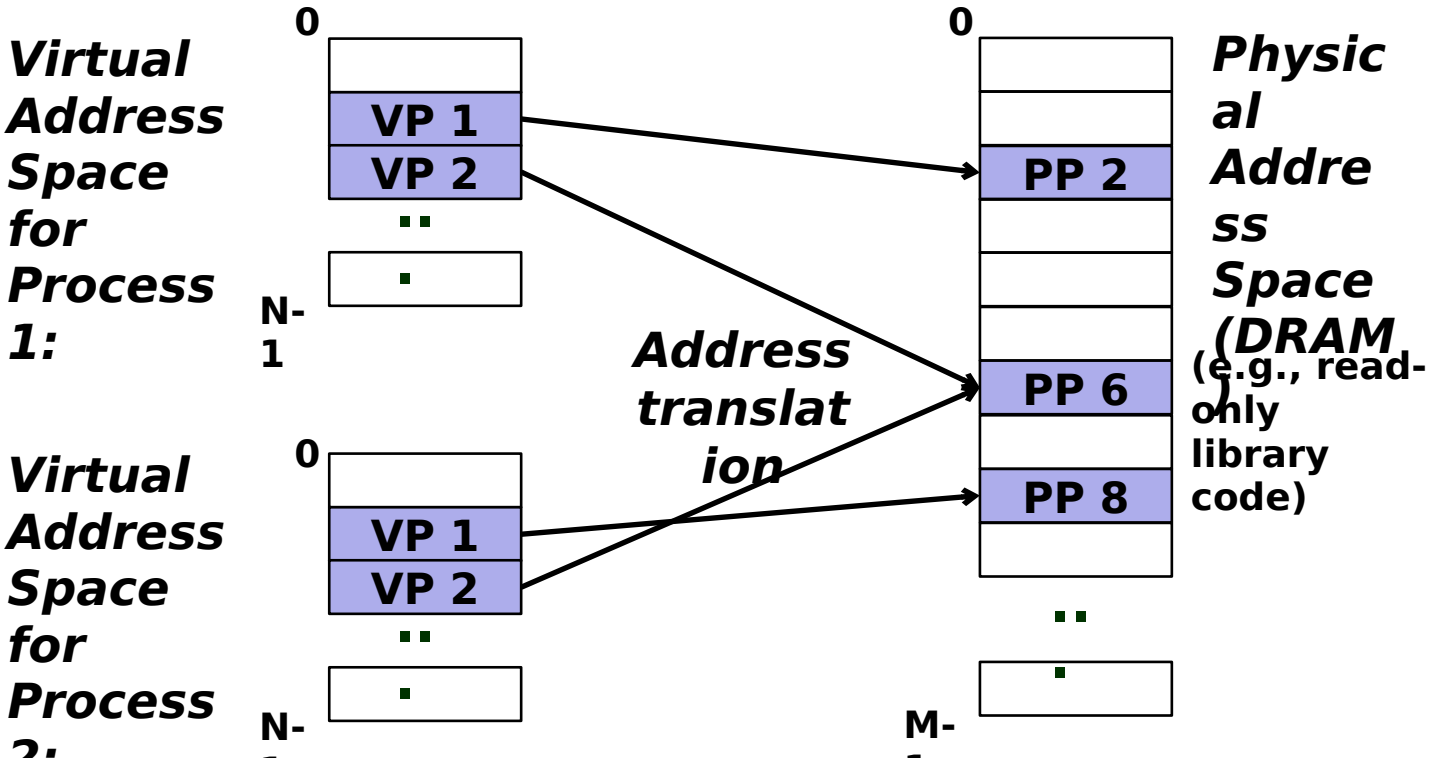
VM as a Tool for Memory Management

- Key idea: each process has its own virtual address space
  - 分层思想：链接、加载、共享、内存分配等很多操作和地址空间相关，于是每个进程的地址空间一样；逻辑页 -> 物理页单独管理，与以上这些操作无关
  - 否则：用户进程在做内存访问时，都需要考虑是否在内存中



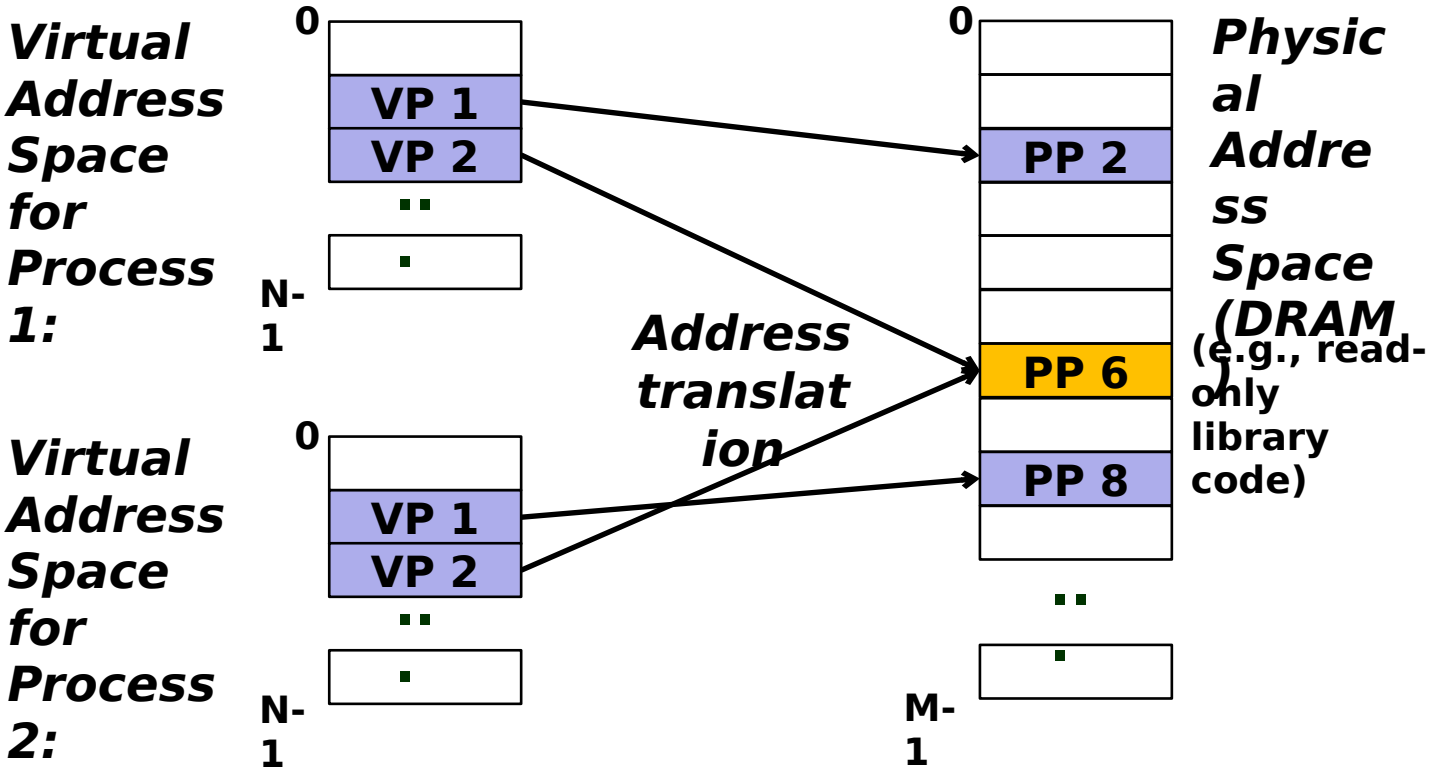
VM as a Tool for Memory Management

- 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
  - 由于有页表，分配时看起来连续的内存，物理内存页未必连续



VM as a Tool for Memory Management

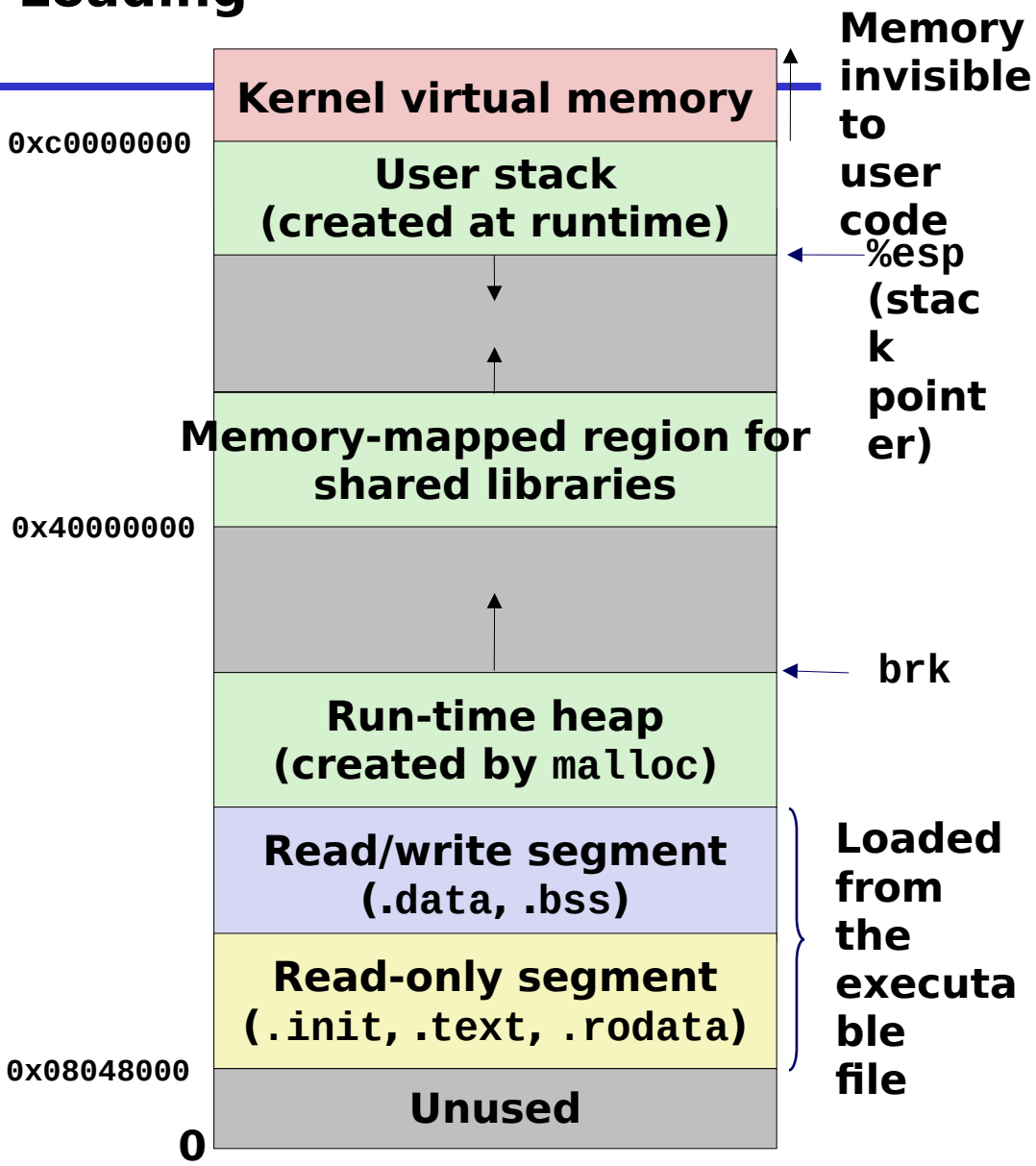
- Sharing code and data among processes
  - Map virtual pages to the same physical page



Simplifying Linking and Loading

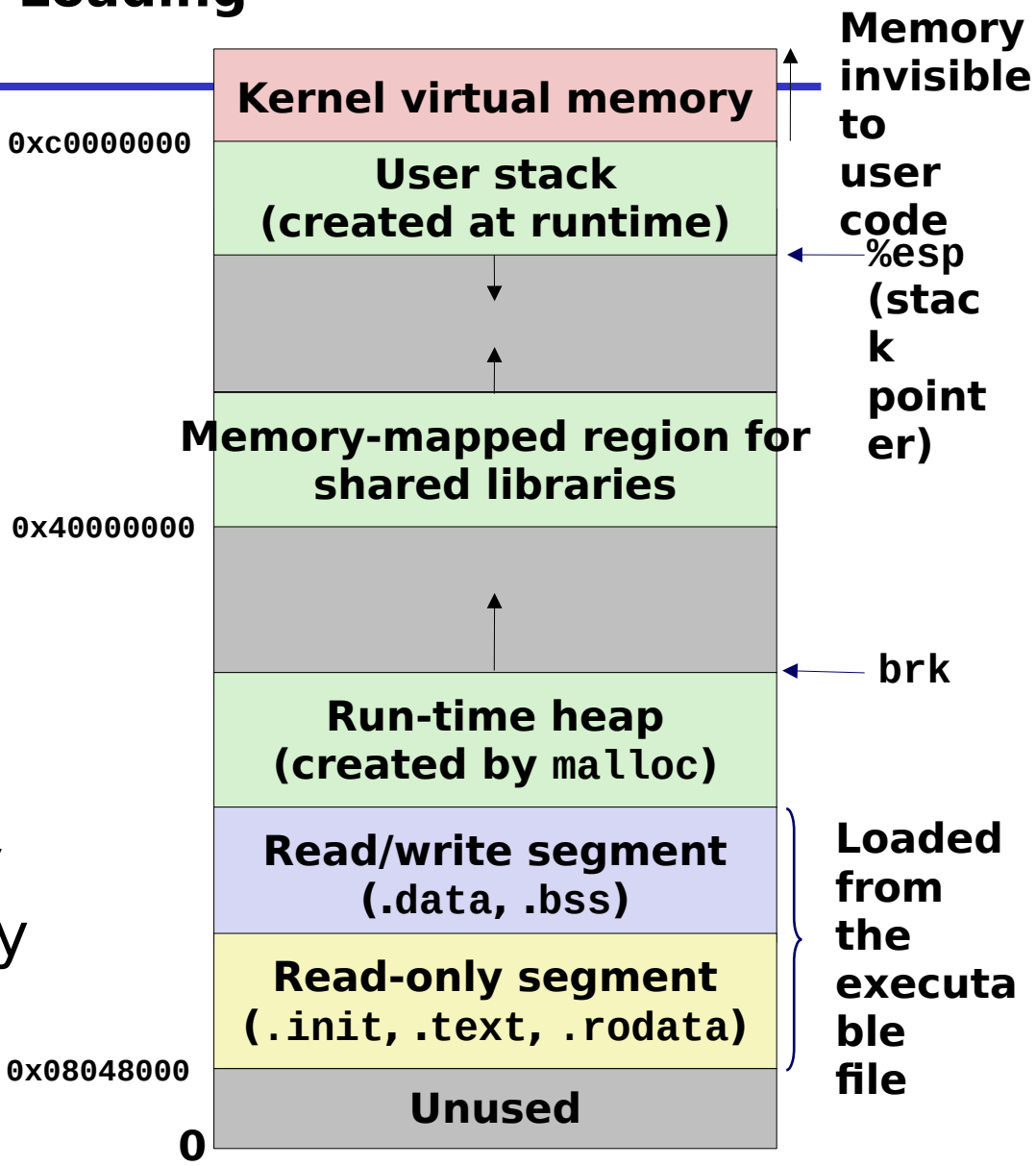
- Linking
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address

否则用户程序还要关心
自己程序所在的物理地址



Simplifying Linking and Loading

- 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

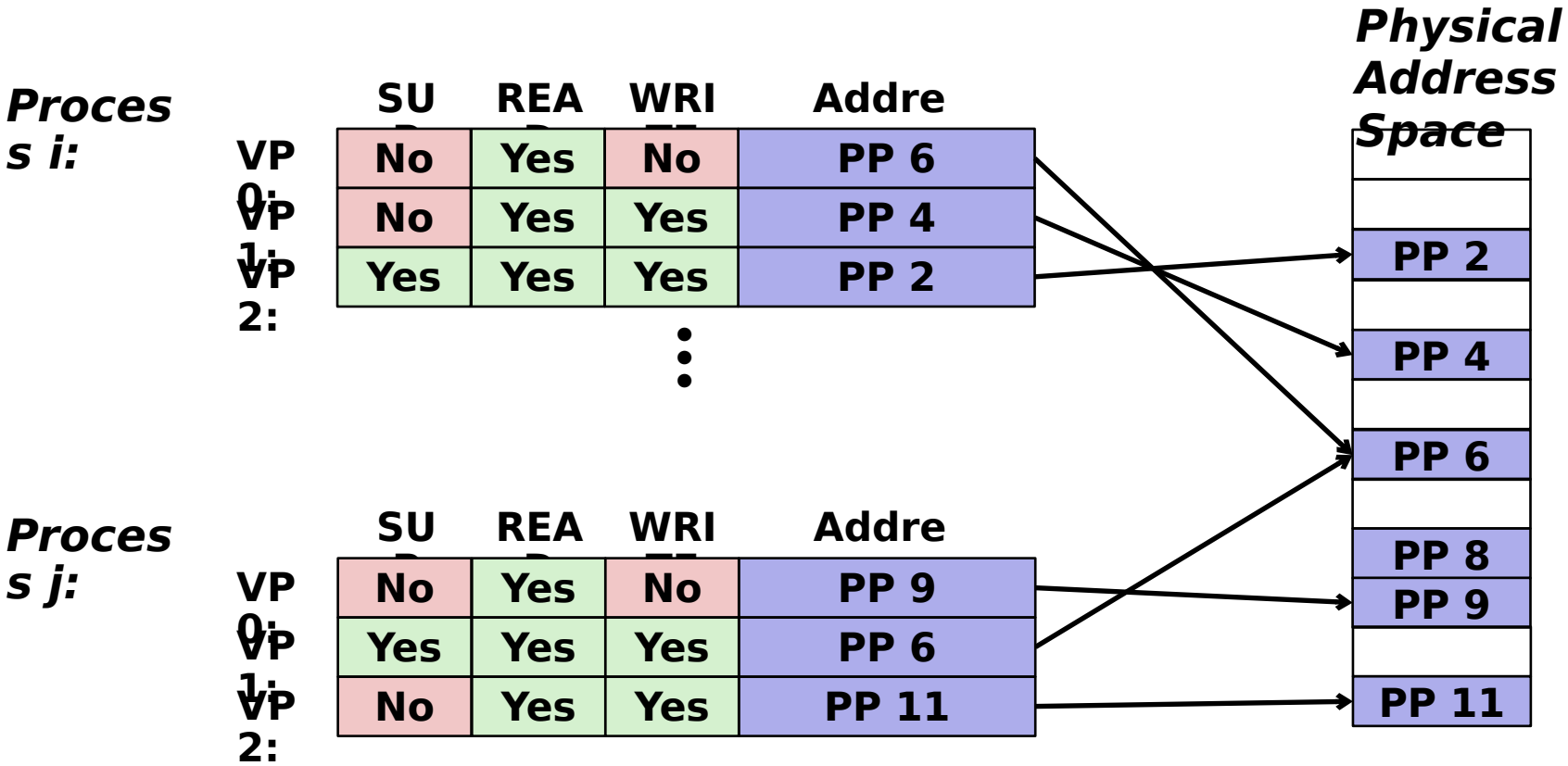


Why Virtual Memory (VM)?

- 高效使用 DRAM
  - 实现将 DRAM 用作 address space 上的一个 cache
- 简化内存管理
  - 每个进程都可以看到一个线性的、统一的地址空间
- 隔离地址空间，提供内存保护
  - 进程无法访问其他进程的内存空间
  - 用户进程无法访问内核的内存空间

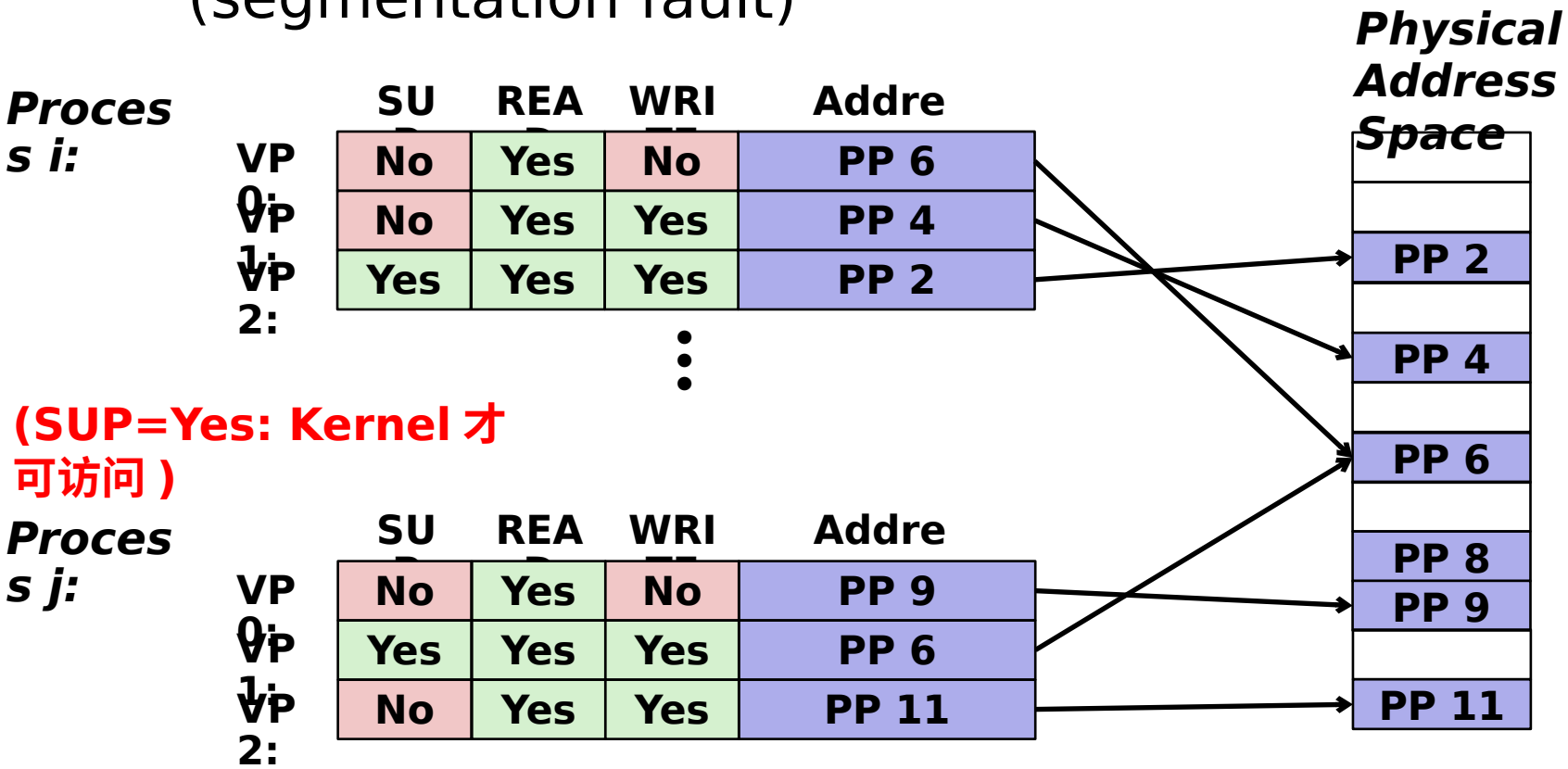
VM as a Tool for Memory Protection

- Extend PTEs with permission bits
  - The same physical page has different permission for different process



VM as a Tool for Memory Protection

- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)



Homework 4