# Virtual Memory: Concepts
# 虚拟内存:概念

100076202：  计算机系统导论

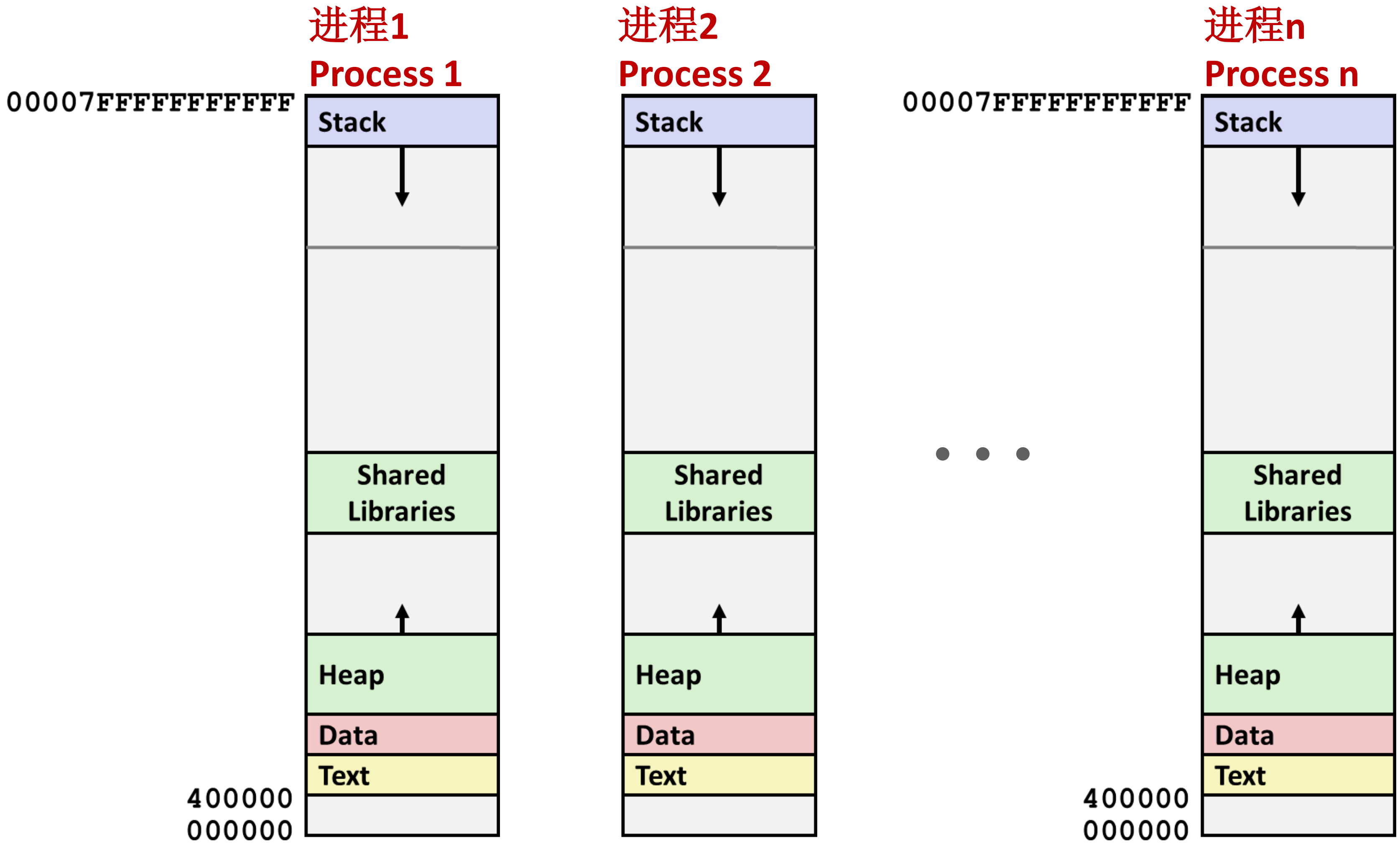

**任课教师：**

**宿红毅  张艳  黎有琦  李秀星**

**原作者：**

Randal E. Bryant and David R. O'Hallaron

# 嗯，这是怎么工作的？！
# Hmmm, How Does This Work?!



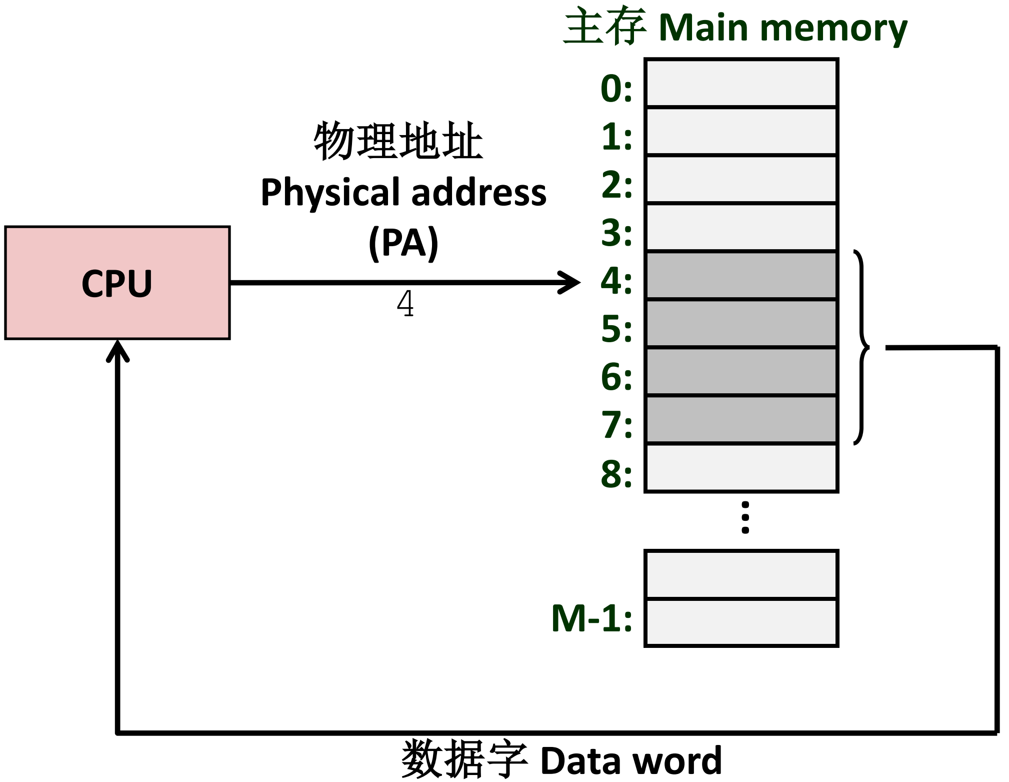

*解决方案：虚拟内存（本次和下次课）*

***Solution: Virtual Memory (today and next lecture)***

# 内容提纲 Today

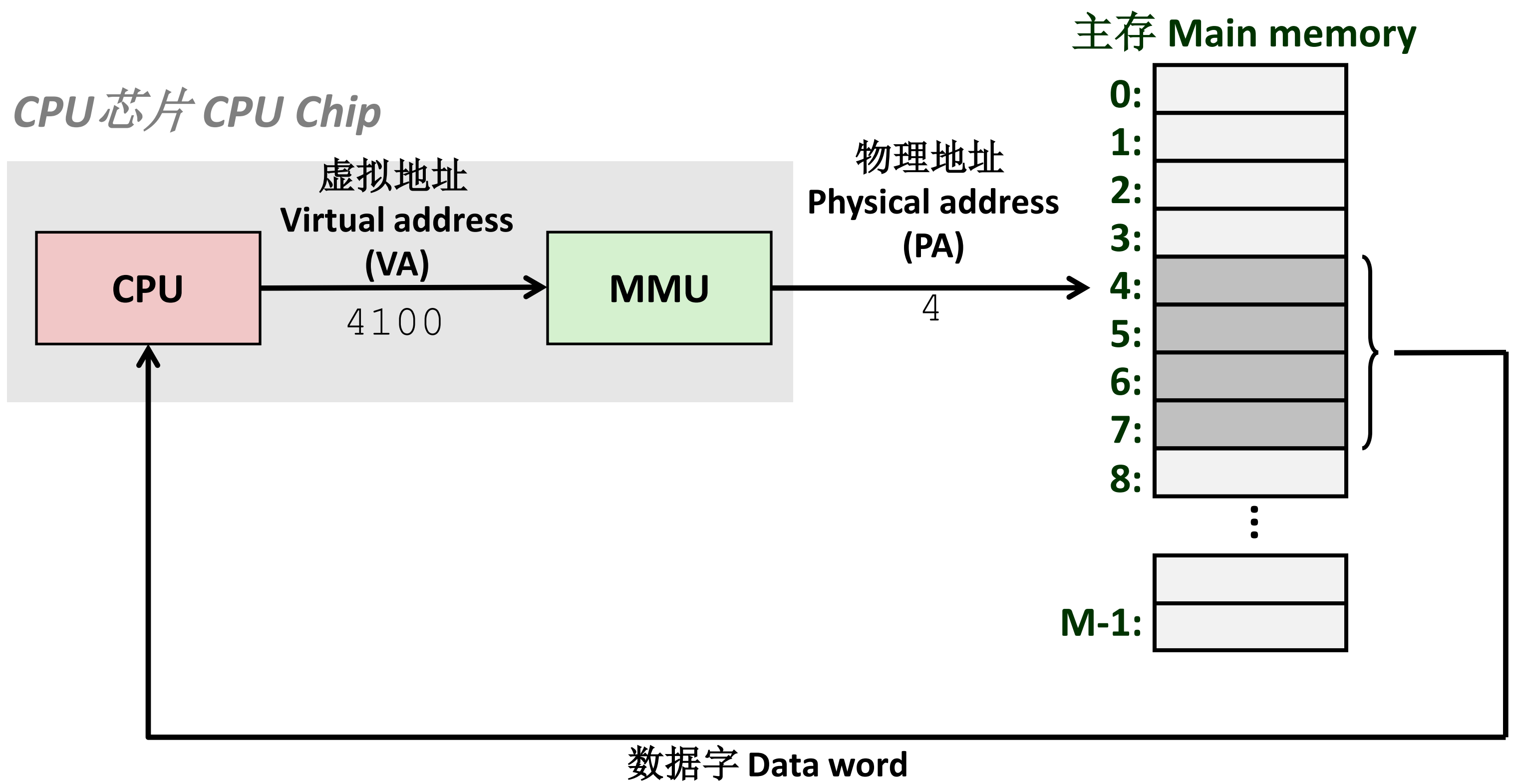
- **地址空间 Address spaces** CSAPP 9.1-9.2
- 基于虚拟内存的缓存机制 VM as a tool for caching CSAPP 9.3
- 基于虚拟内存的内存管理机制 VM as a tool for memory management CSAPP 9.4
- 基于虚拟内存的内存保护机制 VM as a tool for memory protection CSAPP 9.5
- 地址翻译 Address translation CSAPP 9.6

# 使用物理寻址的系统
# A System Using Physical Addressing




- 通常在车、电梯、数字相框等设备中简单系统的嵌入式微控制器使用 **Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames**

# 使用虚拟寻址的系统
# A System Using Virtual Addressing




- 在所有现代服务器、笔记本和智能手机中使用 **Used in all modern servers, laptops, and smart phones**
- 计算机科学的伟大思想之一 **One of the great ideas in computer science**

# 地址空间 Address Spaces

- **线性地址空间：**连续非负整型地址的有序集合 **Linear address space:** Ordered set of contiguous non-negative integer addresses:

  {0, 1, 2, 3 … }

- **虚拟地址空间：** $N = 2^n$ 虚拟地址集合 **Virtual address space:** Set of $N = 2^n$ virtual addresses

  {0, 1, 2, 3, …, N-1}

- **物理地址空间：** $M = 2^m$ 物理地址集合 **Physical address space:** Set of $M = 2^m$ physical addresses

  {0, 1, 2, 3, …, M-1}

# 为什么需要虚拟内存(VM)?
# Why Virtual Memory (VM)?

- 更高效地使用主存 **Uses main memory efficiently**
  - 使用DRAM作为一部分虚拟地址空间的缓存 Use DRAM as a cache for parts of a virtual address space

- 简化内存管理 **Simplifies memory management**
  - 每个进程都用同样的统一线性地址空间 Each process gets the same uniform linear address space

- 隔离的地址空间 **Isolates address spaces**
  - 一个进程不会干扰另一个进程的内存 One process can't interfere with another's memory
  - 用户程序不能访问特权内核信息和代码 User program cannot access privileged kernel information and code

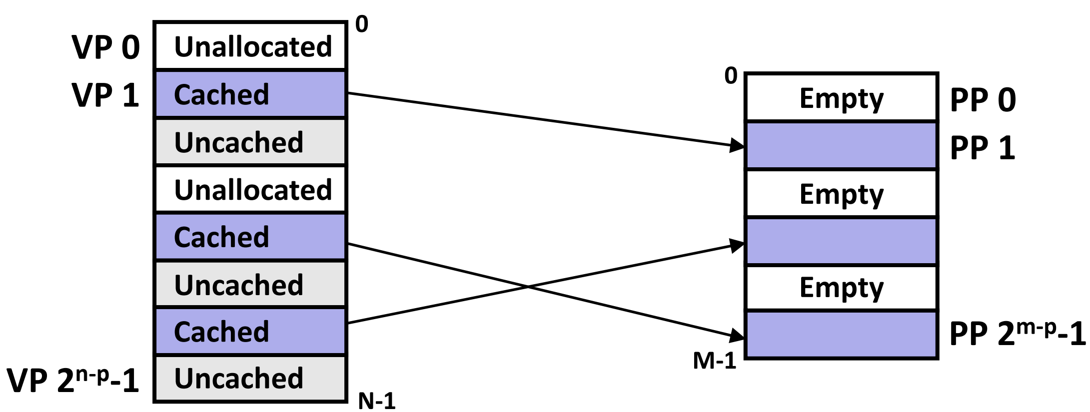
# 内容提纲/Today

- 地址空间 Address spaces
- **基于虚拟内存的缓存机制 VM as a tool for caching**
- 基于虚拟内存的内存管理机制 VM as a tool for memory management
- 基于虚拟内存的内存保护机制 VM as a tool for memory protection
- 地址翻译 Address translation

# 基于虚拟内存的缓存机制
# VM as a Tool for Caching

- 概念上来讲，**虚拟内存**就是N个连续地存储在磁盘上的字节数组 **Conceptually, *virtual memory* is an array of N contiguous bytes stored on disk.**
- 磁盘上的数组的内容是缓存在**物理内存**中的（**DRAM缓存**）**The contents of the array on disk are cached in *physical memory* (*DRAM cache*)**
  - 这些cache块称为页（大小为P=$2^p$字节）These cache blocks are called *pages* (size is P = $2^p$ bytes)

**虚拟内存 Virtual memory** **物理内存 Physical memory**




**虚拟页(VP)存储在磁盘上 Virtual pages (VPs) stored on disk**

**物理页(PP)缓存在DRAM中 Physical pages (PPs) cached in DRAM**

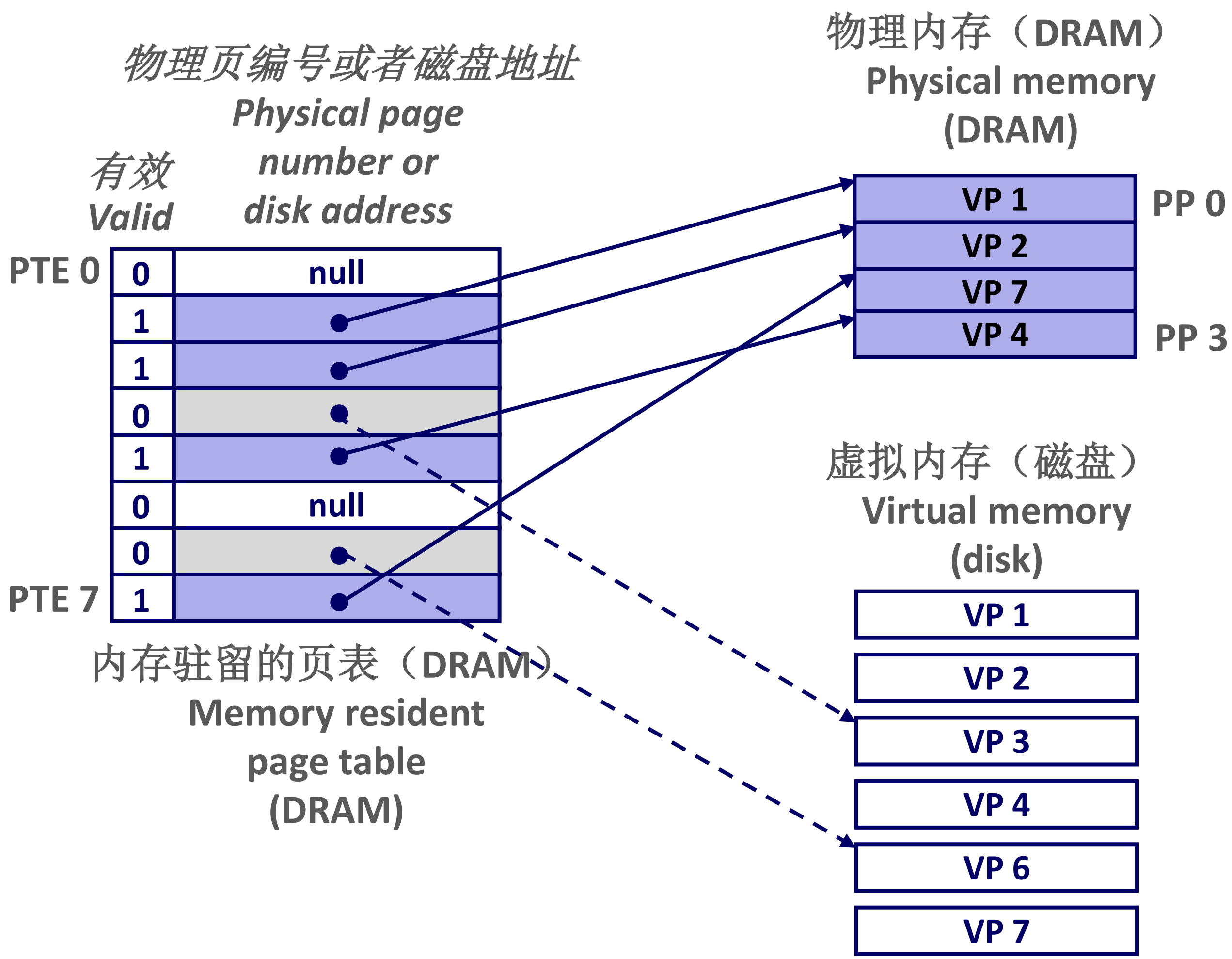
# DRAM缓存组织 DRAM Cache Organization

- **DRAM缓存组织是受不命中后惩罚会很高这一因素影响的 DRAM cache organization driven by the enormous miss penalty**
  - DRAM大概比SRAM慢*10*倍左右 DRAM is about ***10x*** slower than SRAM
  - 磁盘大概比DRAM慢*10000*倍 Disk is about ***10,000x*** slower than DRAM
  - 从磁盘装入块的时间大于1ms（超过一百万个时钟周期）Time to load block from disk > 1ms (> 1 million clock cycles)
    - 在此期间CPU能够做很多计算 CPU can do a lot of computation during that time

- 因此 **Consequences**
  - 比较大的页（块）：通常4 KB  Large page (block) size: typically 4 KB
    - Linux的"巨大页"可以2MB（默认）到1GB  Linux "huge pages" are 2 MB (default) to 1 GB
  - 全相联 Fully associative
    - 任意的虚拟页可以放在任意的物理页中 Any VP can be placed in any PP
    - 与Cache内存不同，需要一个更灵活的映射函数 Requires a "large" mapping function – different from cache memories
  - 高度复杂，替换算法开销比较大 Highly sophisticated, expensive replacement algorithms
    - 由于过于复杂和不确定性，无法在硬件中实现 Too complicated and open-ended to be implemented in hardware
  - 采用写回机制而不是写直达机制 Write-back rather than write-through

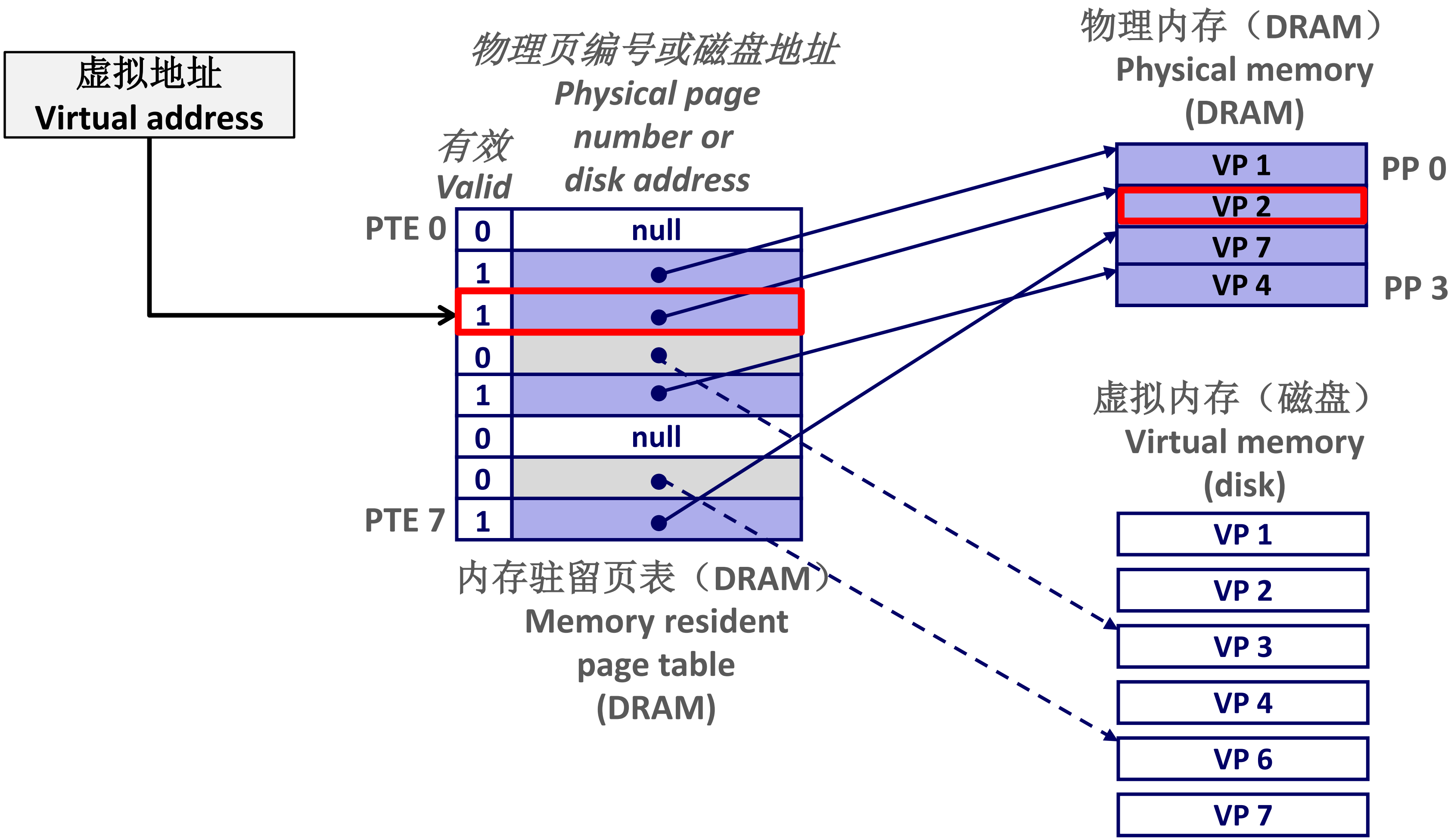
# 使能数据结构：页表
# Enabling Data Structure: Page Table

- 一个页表实际上是将虚拟页映射物理页的页表条目（PTE）构成的数组 A ***page table*** is an array of page table entries (PTEs) that maps virtual pages to physical pages.
  - 每个进程在DRAM中的核心数据结构 Per-process kernel data structure in DRAM



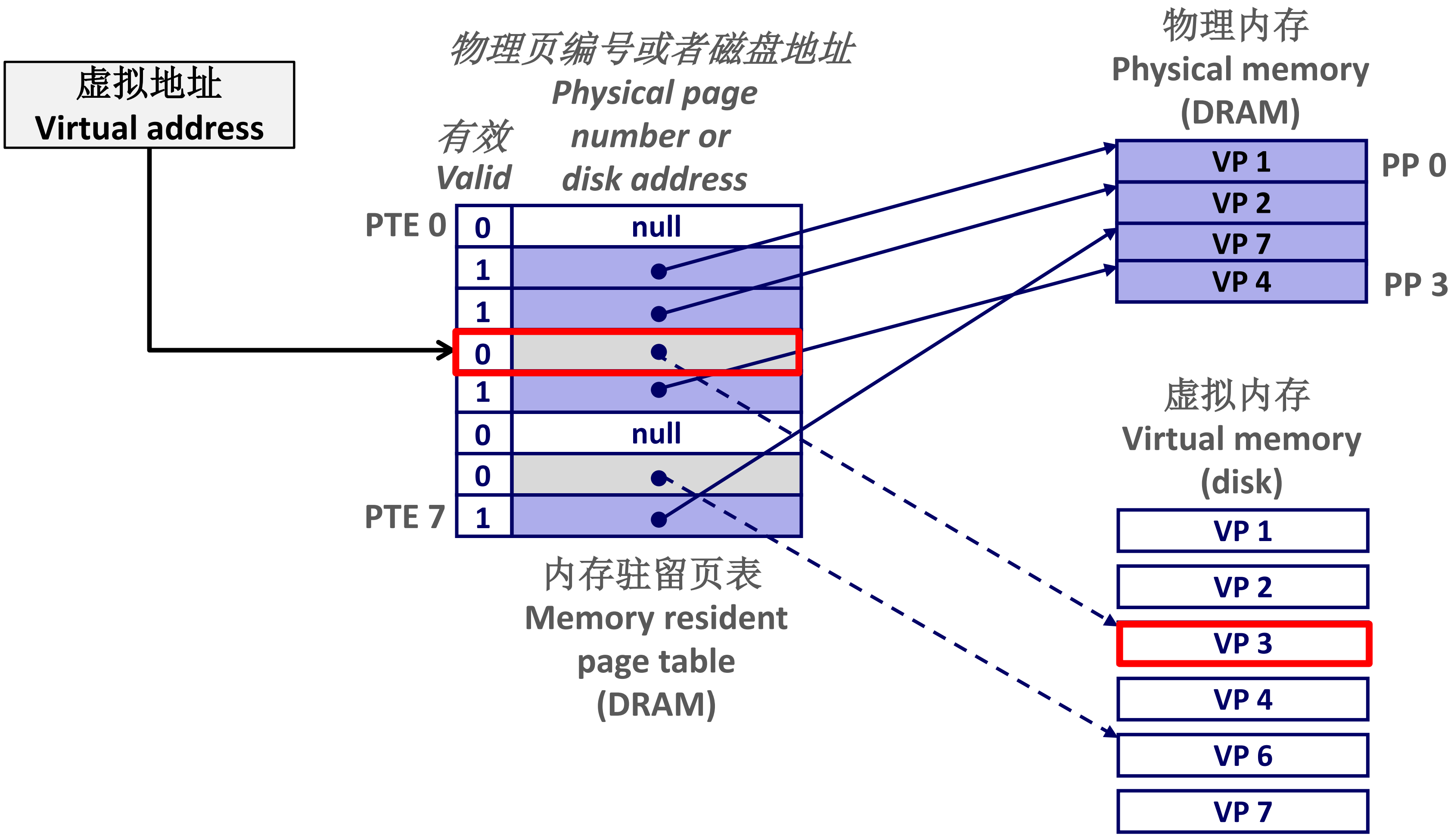
# 页命中 Page Hit

- *页命中：*引用的虚拟内存字在物理内存中（DRAM命中） ***Page hit:*** reference to VM word that is in physical memory (DRAM cache hit)



# 缺页中断 Page Fault

- *缺页中断*：引用的虚拟字不在物理内存中(DRAM缓存不命中) ***Page fault:*** reference to VM word that is not in physical memory (DRAM cache miss)

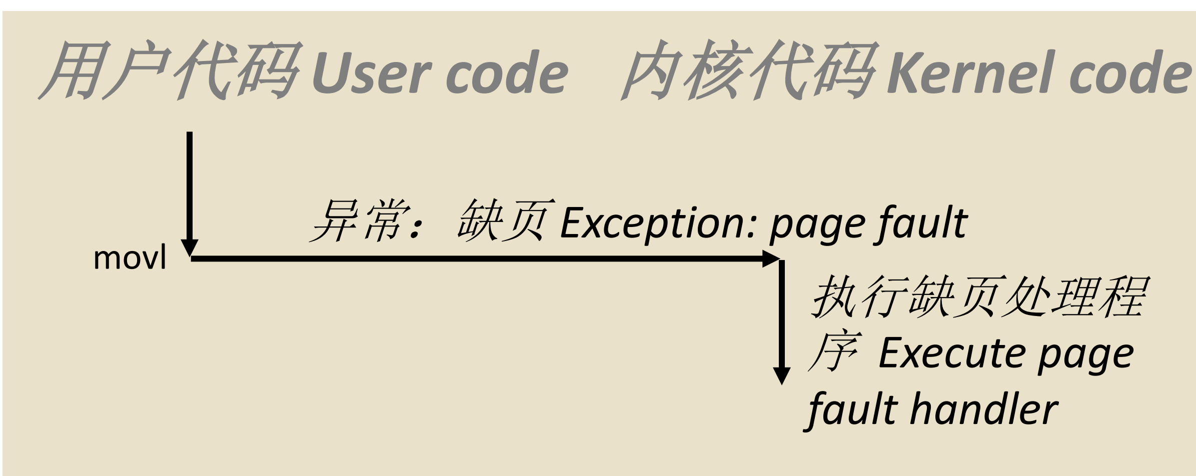


# 触发缺页中断 Triggering a Page Fault

- 用户对内存位置写入　User writes to memory location

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

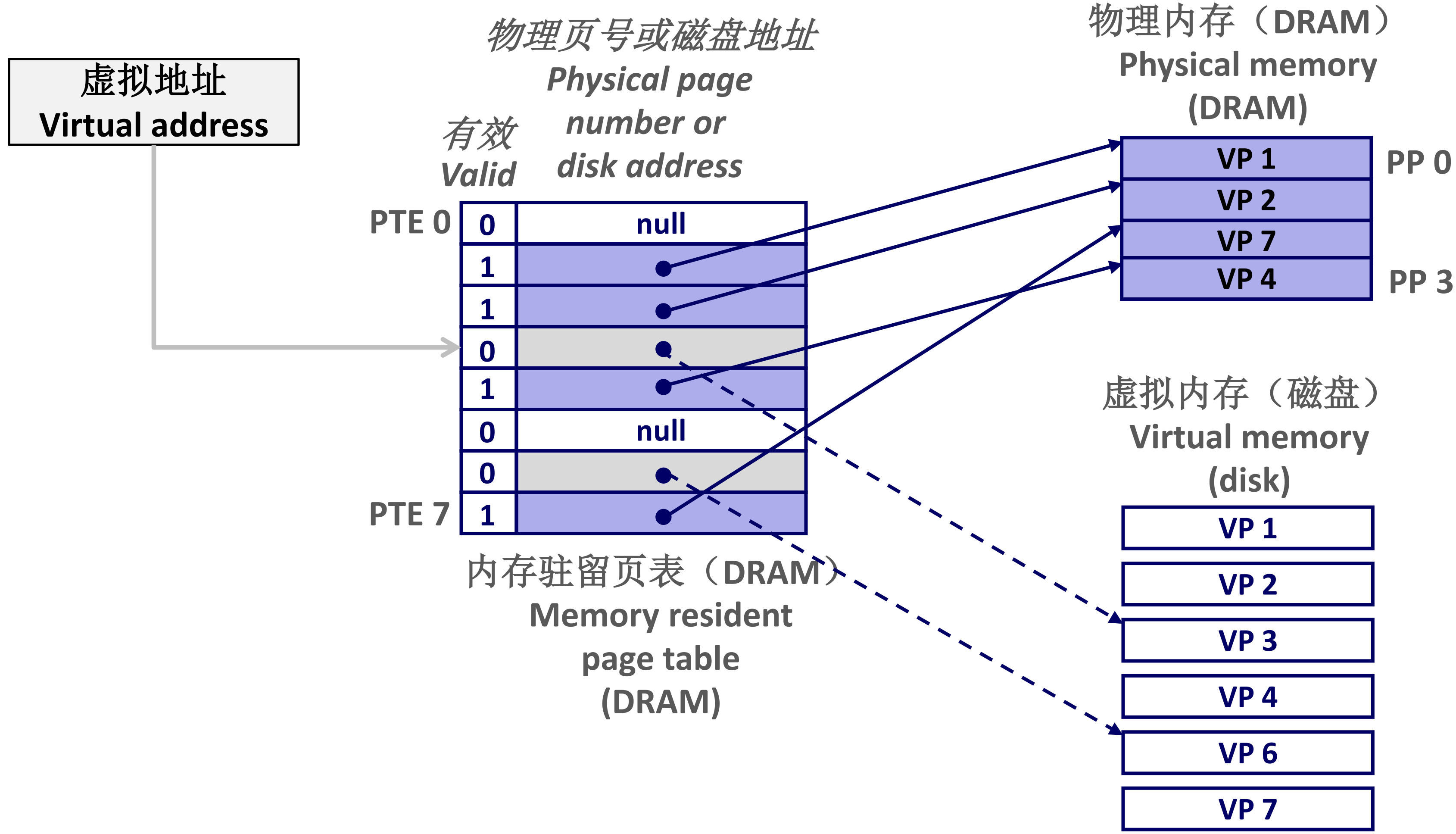
- 用户内存的这部分（页）当前在磁盘上　That portion (page) of user's memory is currently on disk
- **MMU触发缺页异常**　MMU triggers page fault exception
  - (更多细节下次课讲 More details in later lecture)
  - 提升优先级到监督态　Raise privilege level to supervisor mode
  - 引起对软件缺页中断处理程序的过程调用 Causes procedure call to software page fault handler




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

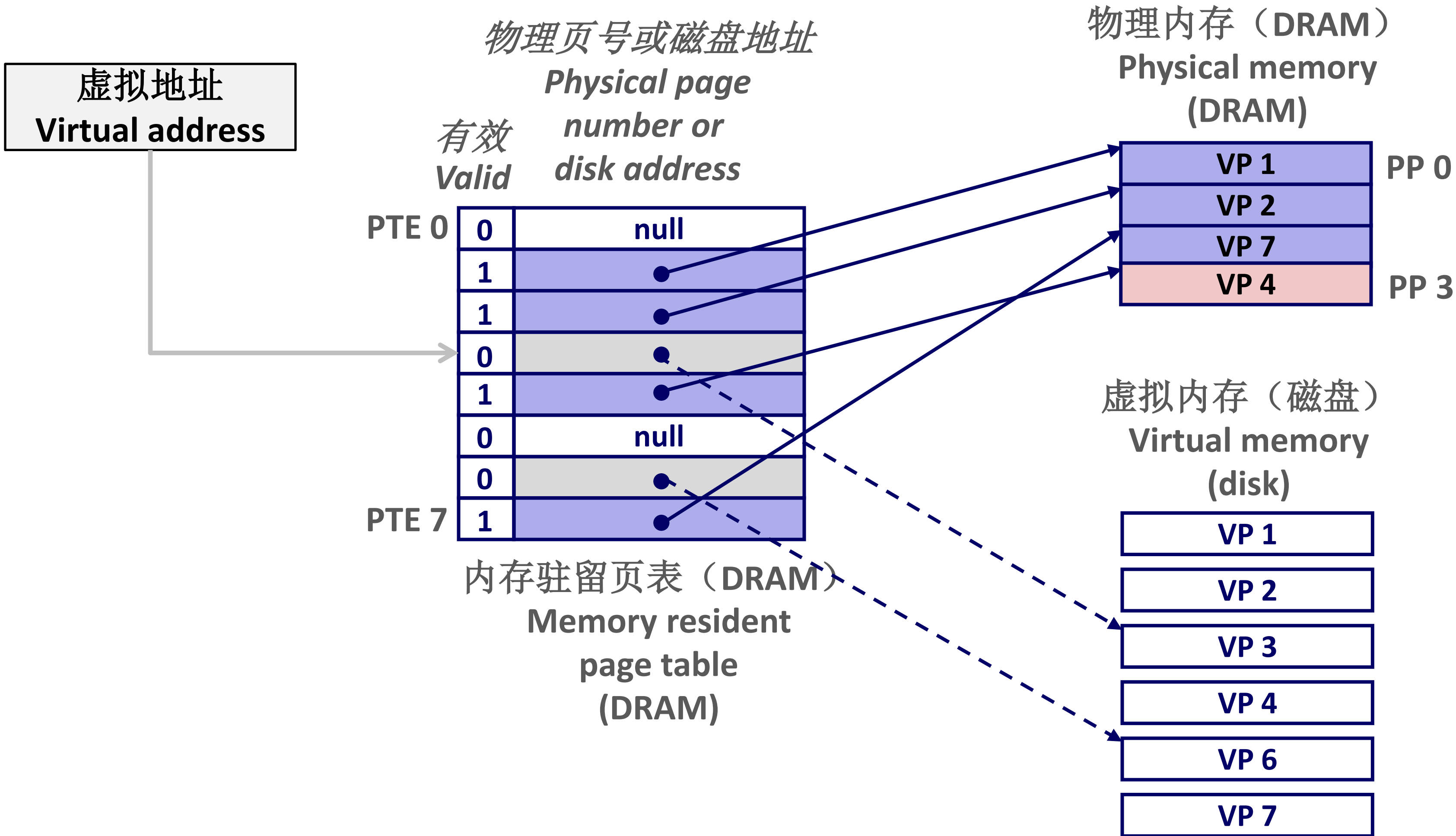
# 缺页中断处理 Handling Page Fault

- 页不命中导致缺页中断（异常的一种） Page miss causes page fault (an exception)



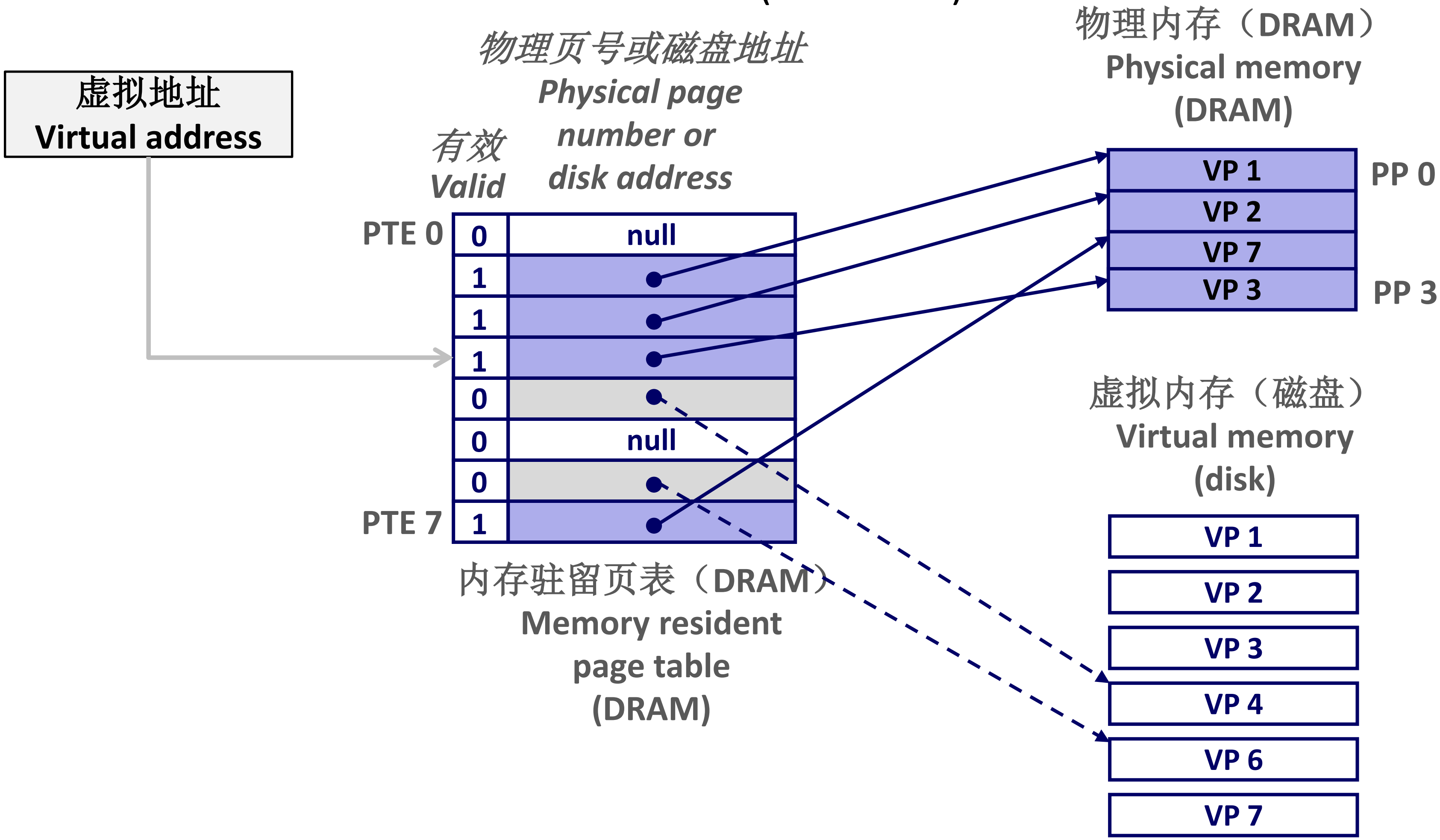
# 缺页中断处理 Handling Page Fault

- 页不命中导致缺页中断（异常的一种） Page miss causes page fault (an exception)
- 缺页中断处理程序选择一个牺牲页换出（以VP 4为例） Page fault handler selects a victim to be evicted (here VP 4)



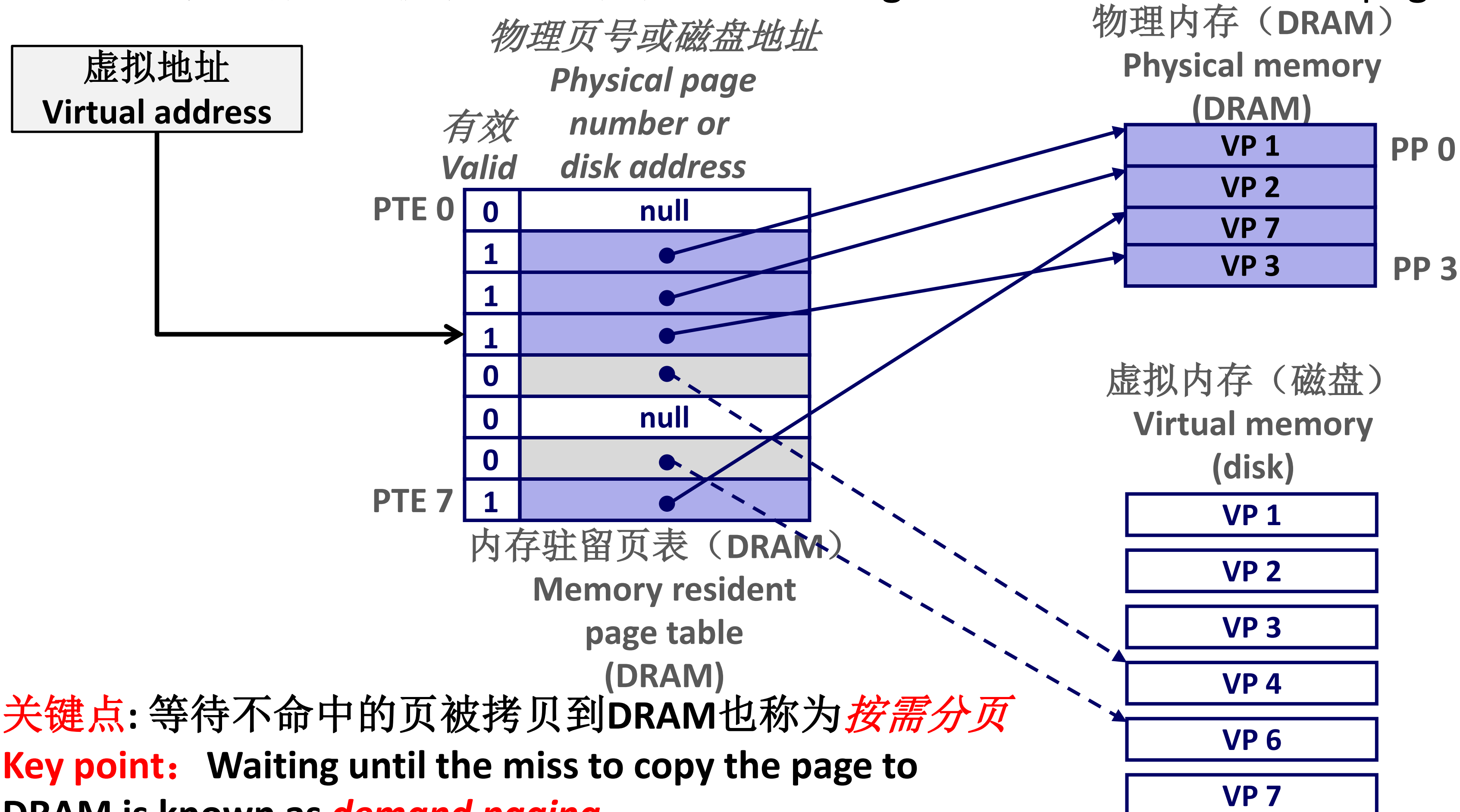
# 缺页中断处理 Handling Page Fault

- 页不命中导致缺页中断（异常的一种） Page miss causes page fault (an exception)
- 缺页中断处理程序选择一个牺牲页换出（以**VP 4**为例）Page fault handler selects a victim to be evicted (here VP 4)



# 缺页中断处理 Handling Page Fault

- 页不命中导致缺页中断(异常的一种) Page miss causes page fault (an exception)
- 缺页中断处理程序选择一个牺牲页换出（以VP 4为例） Page fault handler selects a victim to be evicted (here VP 4)
- 触发指令重新开始执行：页命中！ Offending instruction is restarted: page hit!




**关键点**: 等待不命中的页被拷贝到DRAM也称为*按需分页*

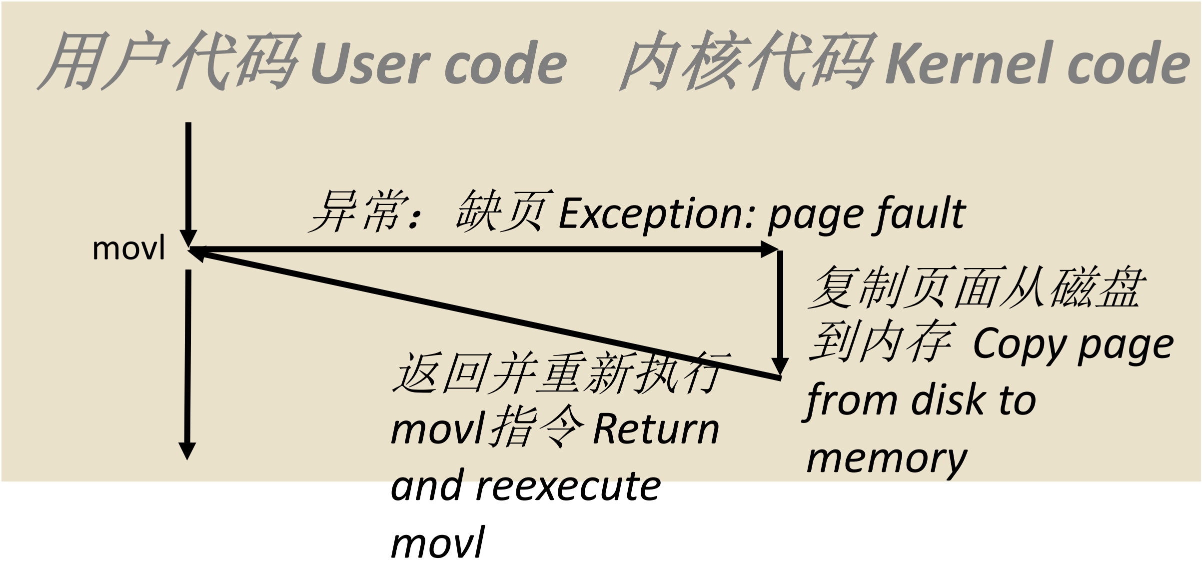
**Key point**： Waiting until the miss to copy the page to DRAM is known as *demand paging*

# 结束缺页中断 Completing page fault

- 缺页中断处理程序执行中断返回指令（iret） Page fault handler executes return from interrupt (**`iret`**) instruction
  - 类似于ret指令，但是还会恢复优先级 Like **`ret`** instruction, but also restores privilege level
  - 返回到引起故障的指令 Return to instruction that caused fault
  - 但是，这次不会产生缺页中断 But, this time there is no page fault

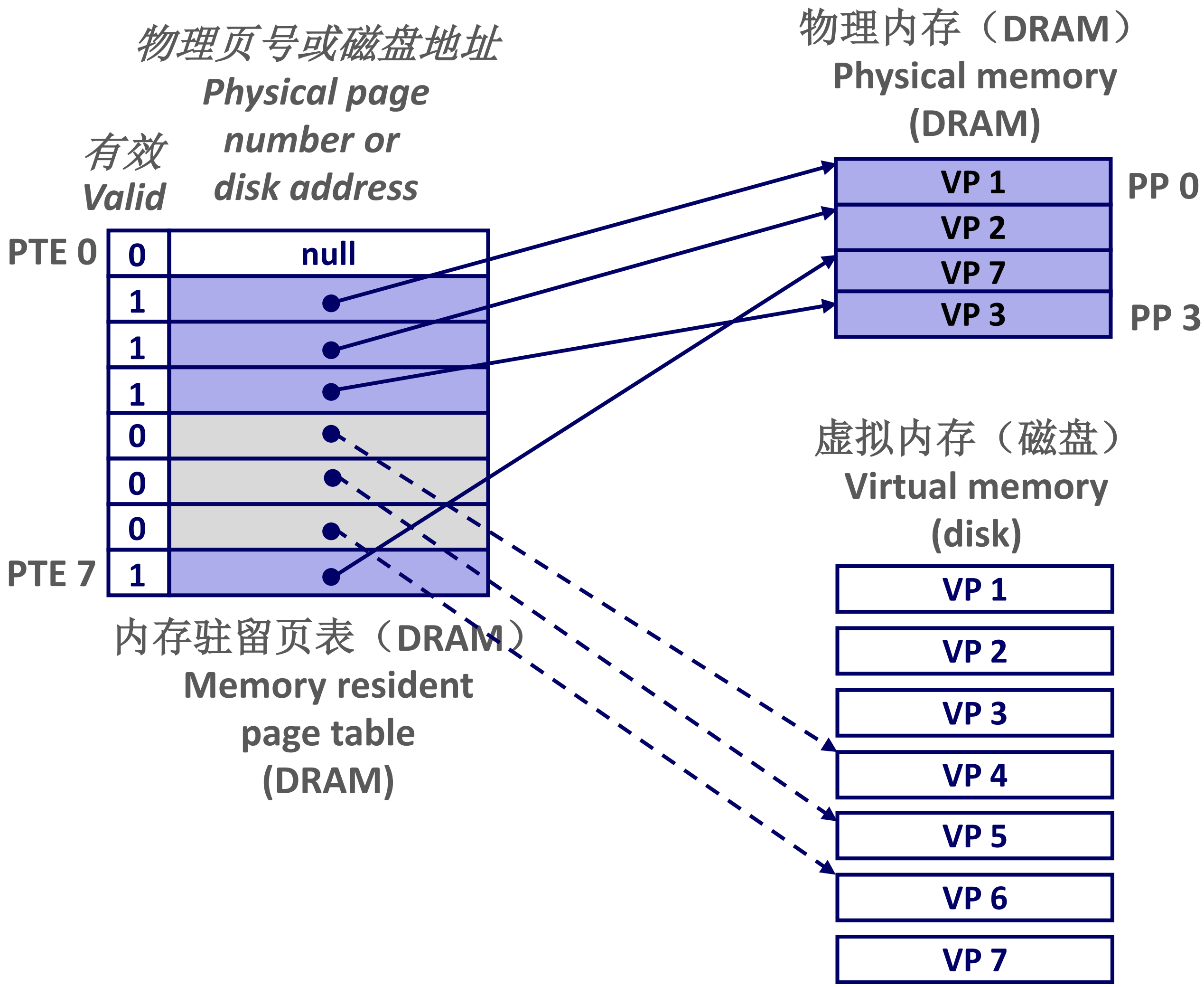
```
int a[1000];
main ()
{
    a[500] = 13;
}
```

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



# 页分配 Allocating Pages

- 分配虚拟内存的一个新页（VP 5）Allocating a new page (VP 5) of virtual memory.



# 局部性再次发挥作用
# Locality to the Rescue Again!

- 虚拟内存看起来非常低效，能有效工作是因为局部性 **Virtual memory seems terribly inefficient, but it works because of locality.**

- 在任何时间点，程序更倾向于只访问一个活跃的虚拟页集合，也称为工作集 **At any point in time, programs tend to access a set of active virtual pages called the *working set***
  - 具有更好的时间局部性的程序会有更小的工作集 Programs with better temporal locality will have smaller working sets

- 如果工作集的大小小于主存大小 **If (working set size < main memory size)**
  - 每个进程在强制不命中后就会获得比较好的性能 Good performance for one process after compulsory misses

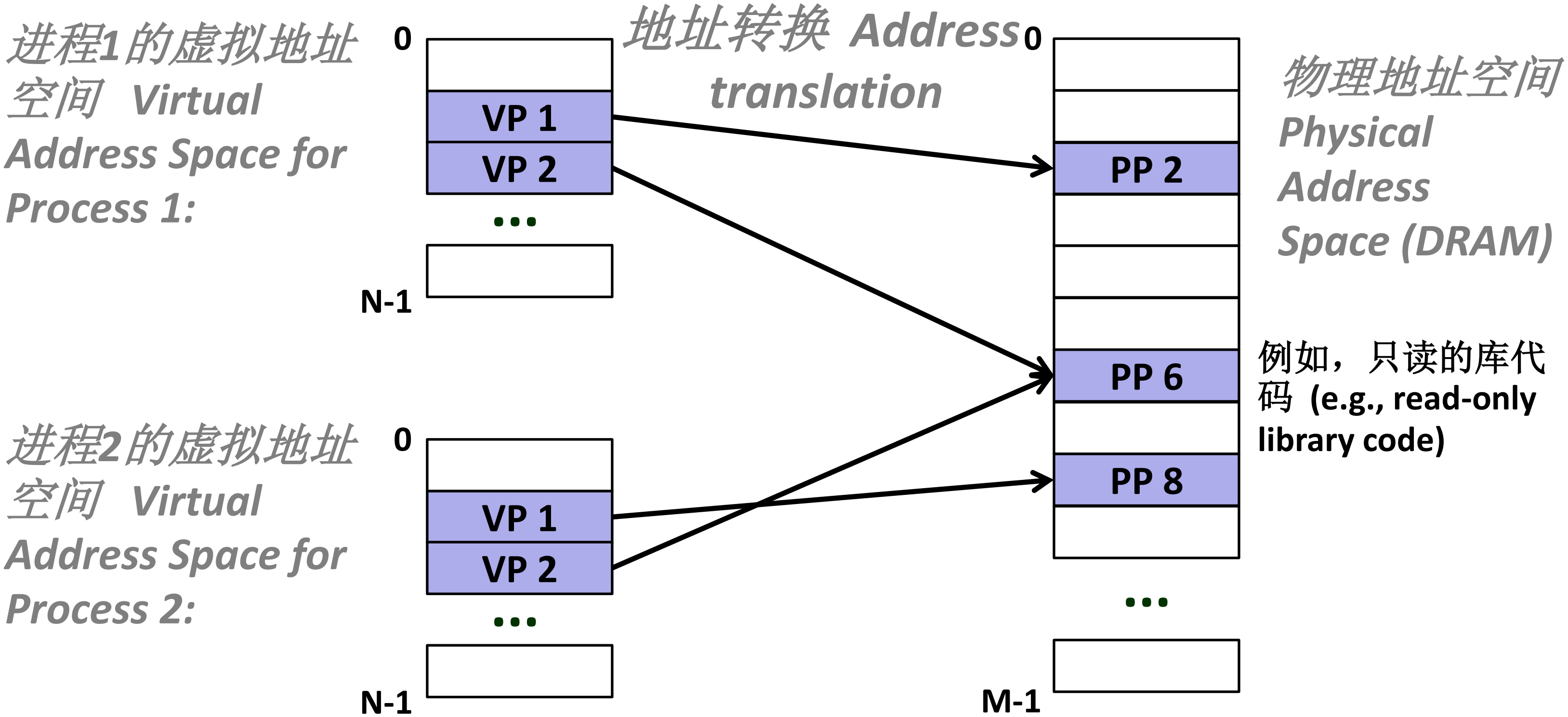
- 如果工作集的总大小大于主存大小 **If ( SUM(working set sizes) > main memory size )**
  - *抖动：性能会由于持续的页面换入换出而变差* *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously
  - 如果多个进程同时运行，在它们的总工作集大小大于主存大小时发生抖动 If multiple processes run at the same time, thrashing occurs if their total working set size > main memory size

# 议题 Today

- 地址空间 Address spaces
- 基于虚拟内存的缓存机制 VM as a tool for caching
- **基于虚拟内存的内存管理机制 VM as a tool for memory management**
- 基于虚拟内存的内存保护机制 VM as a tool for memory protection
- 地址翻译 Address translation

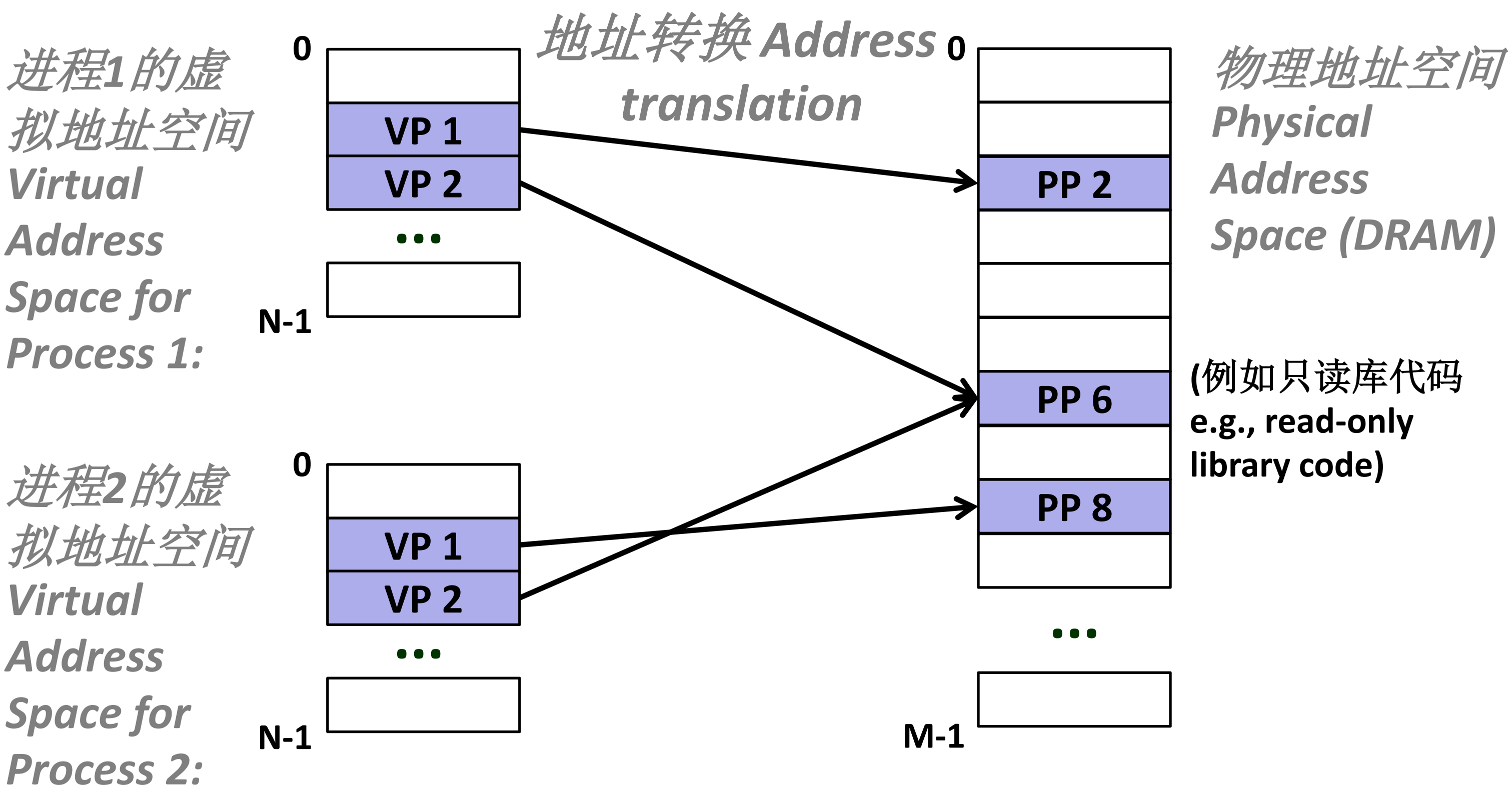
# 基于虚拟内存的内存管理机制
## VM as a Tool for Memory Management

- 关键点：每个进程有自己的虚拟地址空间 **Key idea: each process has its own virtual address space**
  - 将内存看做简单的线性数组 It can view memory as a simple linear array
  - 映射函数将地址分散到物理内存中 Mapping function scatters addresses through physical memory
    - 好的映射函数会提高局部性 Well-chosen mappings can improve locality



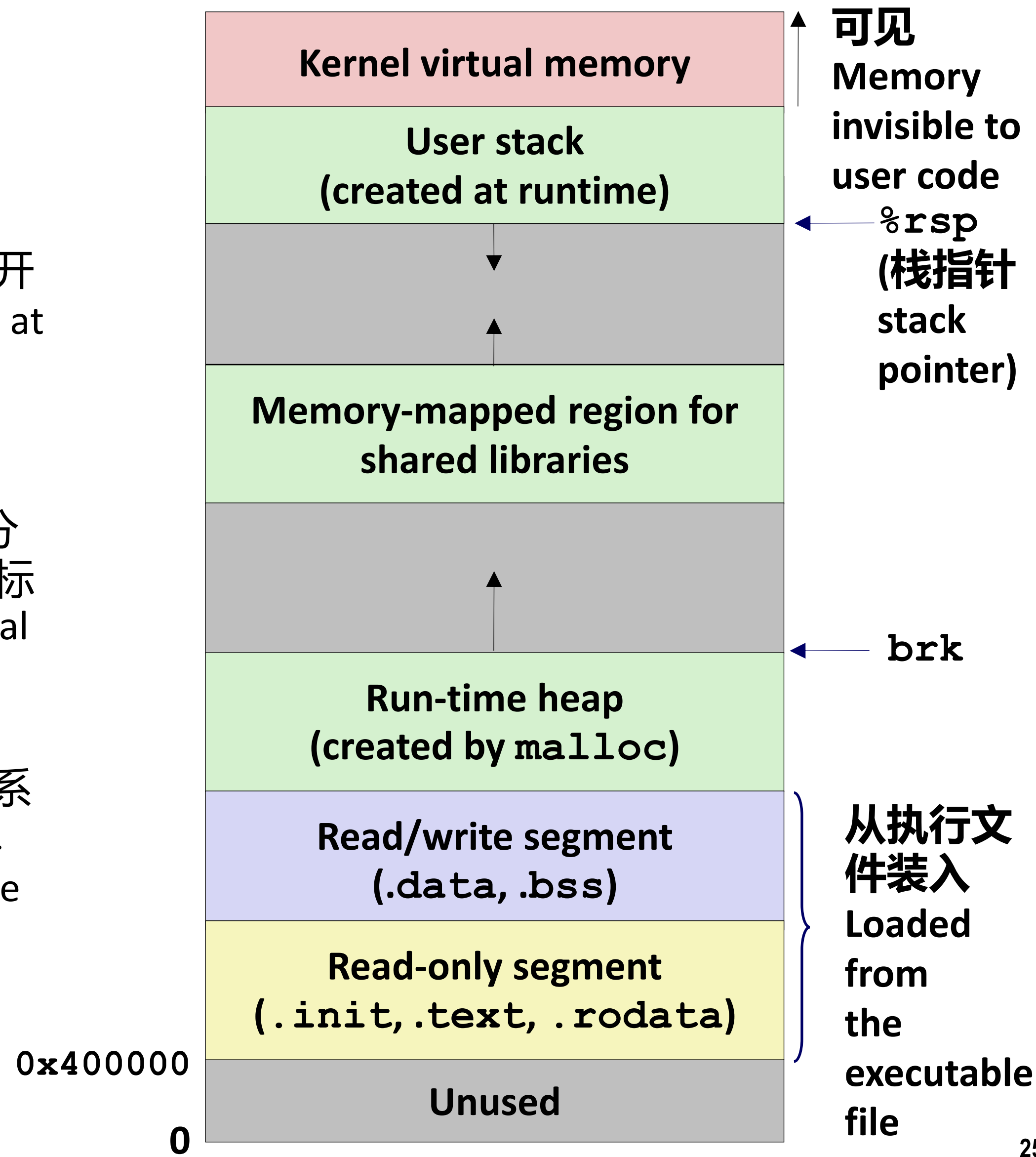
# 基于虚拟内存的内存管理机制
# VM as a Tool for Memory Management

- 简化内存分配 **Simplifying memory allocation**
  - 每个虚拟页可以被映射到任意物理页 Each virtual page can be mapped to any physical page
  - 一个虚拟页可以在不同的时间点存储在不同的物理页中 A virtual page can be stored in different physical pages at different times
- 在进程间共享代码和数据 **Sharing code and data among processes**
  - 将虚拟页映射到同一个物理页 Map virtual pages to the same physical page (here: PP 6)



# 简化链接和加载
# Simplifying Linking and Loading

- **链接 Linking**
  - 每个程序都有类似的虚拟地址空间 Each program has similar virtual address space
  - 代码、数据和堆总是从相同的地址开始 Code, data, and heap always start at the same addresses.
- **加载 Loading**
  - **execve**负责为`.text`和`.data`节分配虚拟页并创建页表条目，并将其标记为无效 **execve** allocates virtual pages for .text and .data sections & creates PTEs marked as invalid
  - .text和.data节中的页是由虚拟内存系统按需一页一页拷贝的 The **.text** and **.data** sections are copied, page by page, on demand by the virtual memory system



# 议题 Today

- 地址空间 Address spaces
- 基于虚拟内存的缓存机制 VM as a tool for caching
- 基于虚拟内存的内存管理机制 VM as a tool for memory management
- **基于虚拟内存的内存保护机制 VM as a tool for memory protection**
- 地址翻译 Address translation

# 基于虚拟内存的内存保护机制
# VM as a Tool for Memory Protection

- 对页表记录进行扩展增加权限位 **Extend PTEs with permission bits**
- **MMU在每次内存访问时检查 MMU checks these bits on each access**

*进程i*
*Process i:*

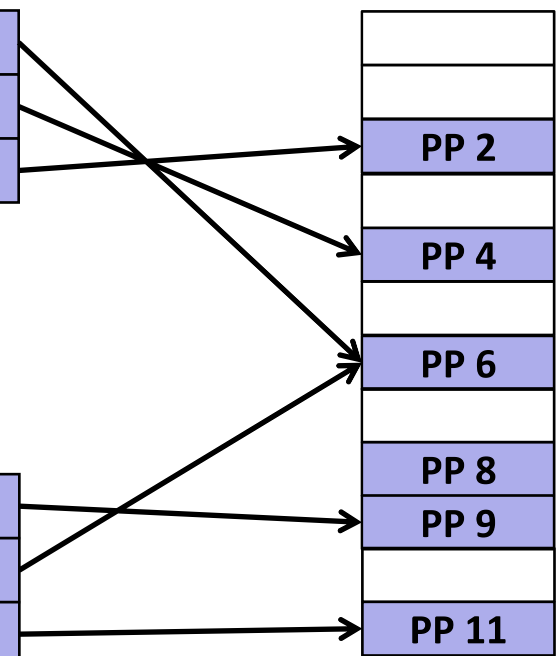
| | SUP | READ | WRITE | EXEC | Address |
|---|---|---|---|---|---|
| VP 0: | No | Yes | No | Yes | PP 6 |
| VP 1: | No | Yes | Yes | Yes | PP 4 |
| VP 2: | Yes | Yes | Yes | No | PP 2 |

⋮

*进程j*
*Process j:*

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

*物理地址空间*
*Physical Address Space*




**SUP：需要内核模式 SUP: requires kernel mode**

# 议题 Today

- 地址空间 Address spaces
- 基于虚拟内存的缓存机制 VM as a tool for caching
- 基于虚拟内存的内存管理机制 VM as a tool for memory management
- 基于虚拟内存的内存保护机制 VM as a tool for memory protection
- **地址翻译 Address translation**

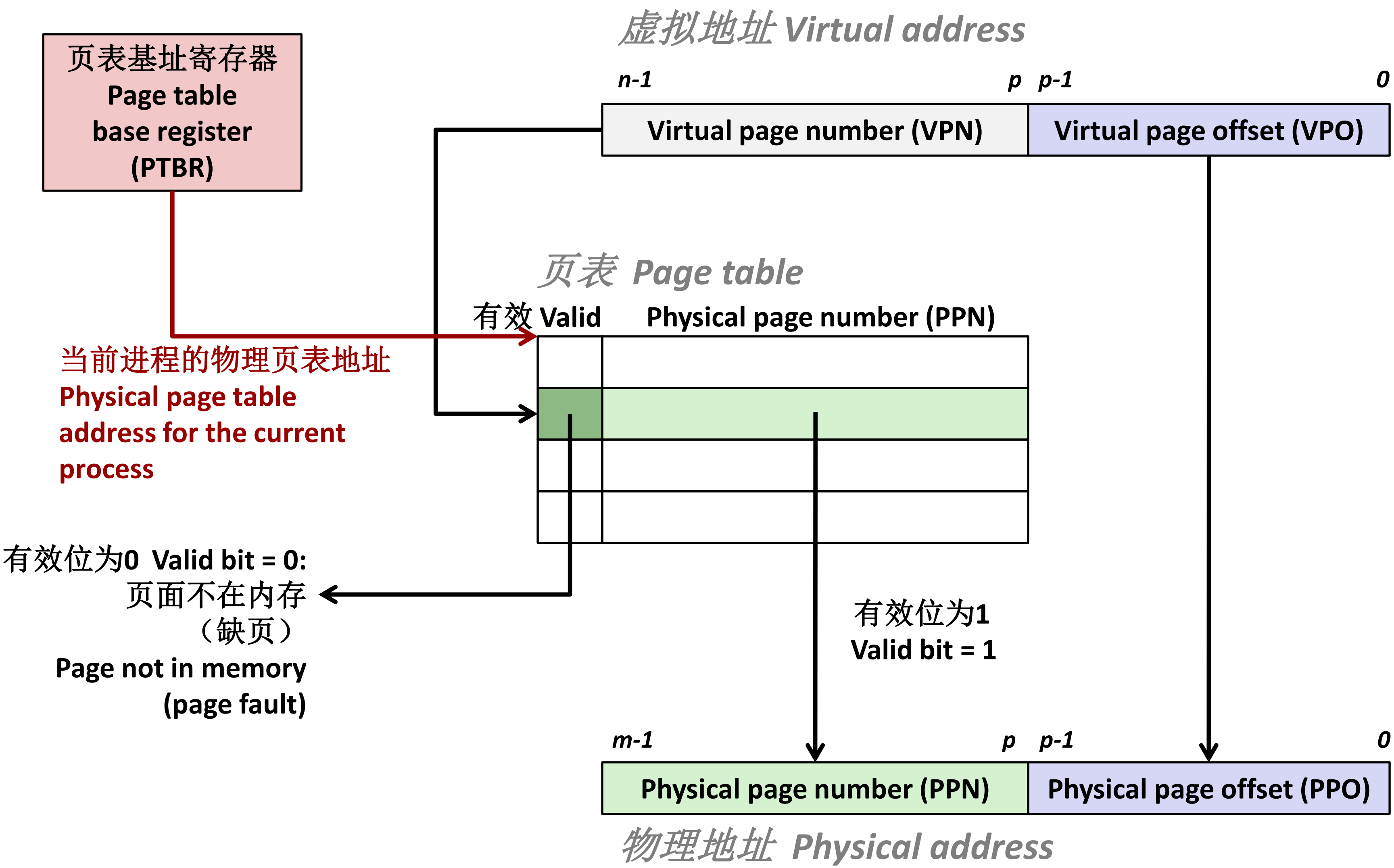
# 虚拟地址翻译 VM Address Translation

- 虚拟地址空间 **Virtual Address Space**
  - $V = \{0, 1, \ldots, N-1\}$
- 物理地址空间 **Physical Address Space**
  - $P = \{0, 1, \ldots, M-1\}$
- 地址翻译 **Address Translation**
  - ***映射*** ***MAP:*** $V \rightarrow P \cup \{\varnothing\}$
  - 对于虚拟地址a For virtual address ***a***:
    - ***MAP(a) = a'*** if data at virtual address ***a*** is at physical address ***a'*** in ***P***
      如果虚拟地址a中的数据在P的物理地址a'中
    - ***MAP(a) =*** $\varnothing$ if data at virtual address ***a*** is not in physical memory
      如果虚拟地址a中的数据不在物理内存中
      - 非法的或者存在磁盘上 Either invalid or stored on disk

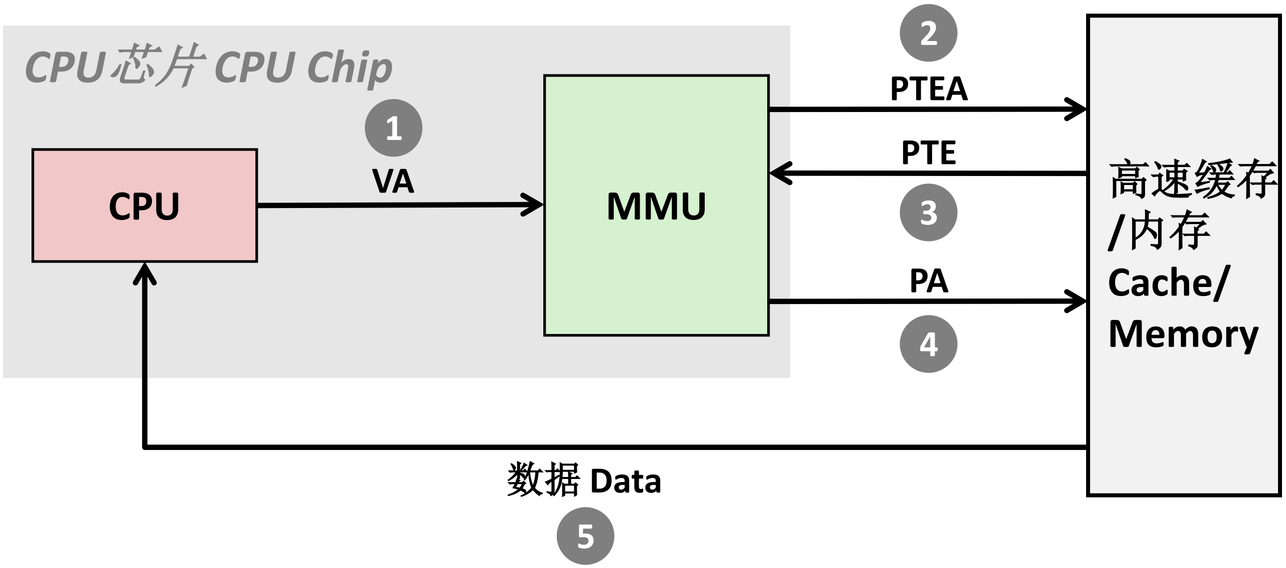
# 地址翻译符号总结
# Summary of Address Translation Symbols

- 基本参数 **Basic Parameters**
  - **$N = 2^n$** : Number of addresses in virtual address space 虚拟地址空间的地址个数
  - **$M = 2^m$** : Number of addresses in physical address space 物理地址空间的地址个数
  - **$P = 2^p$** : Page size (bytes) 页大小（字节）
- 虚拟地址VA划分 **Components of the virtual address (VA)**
  - **TLBI**: TLB index TLB索引
  - **TLBT**: TLB tag TLB标记
  - **VPO**: Virtual page offset 虚拟页内偏移
  - **VPN**: Virtual page number 虚拟页号
- 物理地址PA划分 **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO) 物理页内偏移（同VPO）
  - **PPN:** Physical page number 物理页号

# 基于页表的地址翻译
# Address Translation With a Page Table



# 地址翻译：页命中 Address Translation: Page Hit



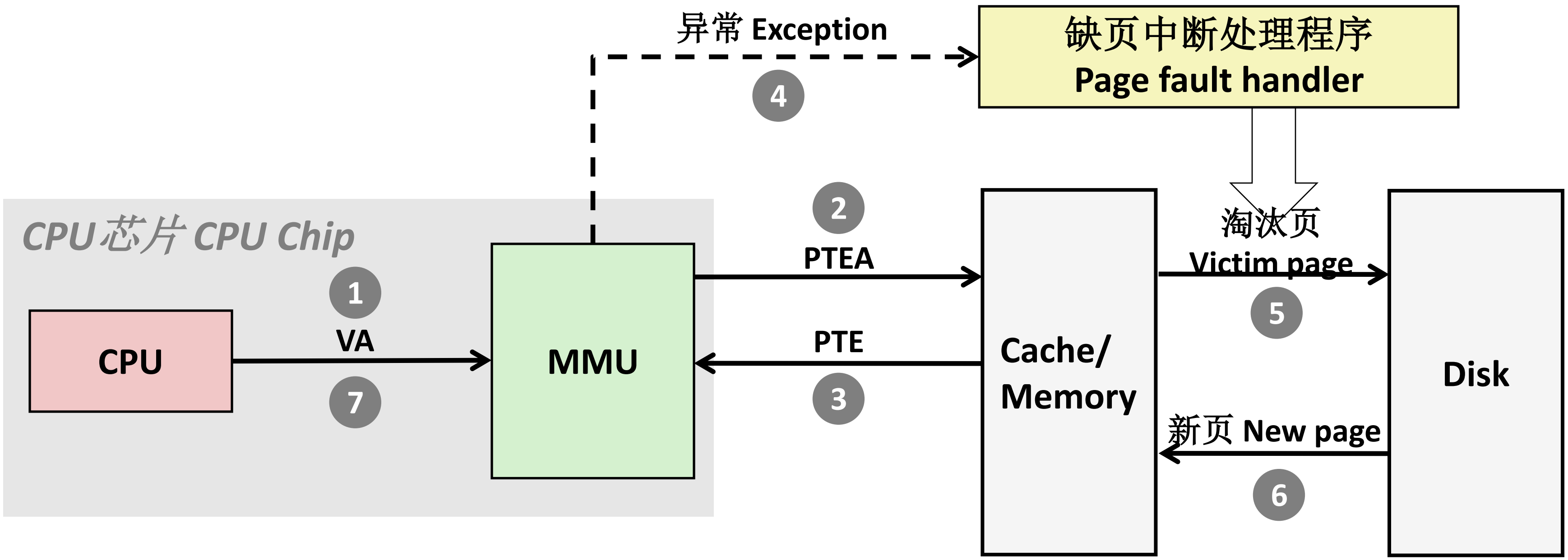

1) 处理器将虚拟地址发送给MMU  Processor sends virtual address to MMU

2-3) MMU从内存页表中获取页表条目  MMU fetches PTE from page table in memory

4) MMU将物理地址发给Cache或者主存  MMU sends physical address to cache/memory

5) Cache或者主存将数据字发送给处理器  Cache/memory sends data word to processor

# 地址翻译：缺页中断 Address Translation: Page Fault




1) 处理器将虚拟地址发给MMU  Processor sends virtual address to MMU

2-3)MMU从内存中的页表取出页表条目   MMU fetches PTE from page table in memory

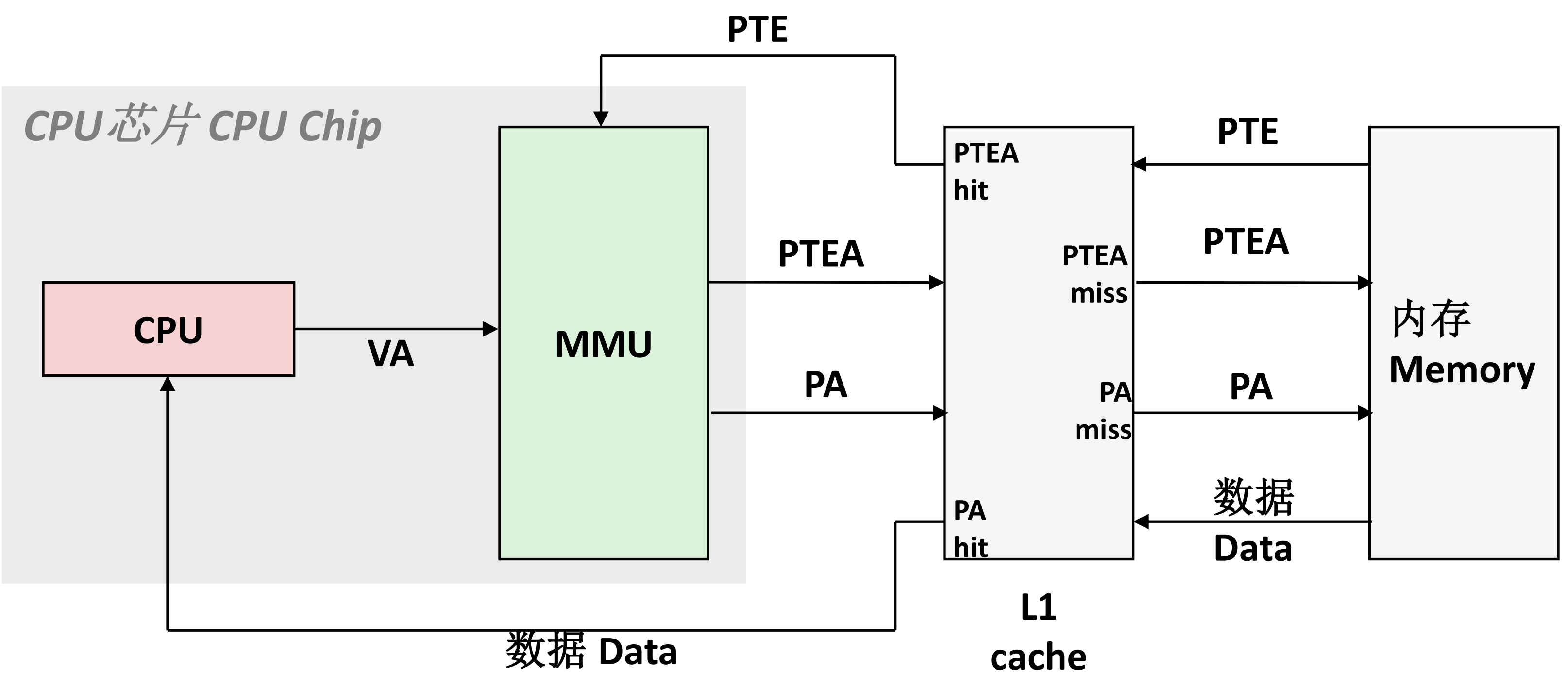
4) 当有效位为0时MMU触发缺页中断异常  Valid bit is zero, so MMU triggers page fault exception

5) 异常处理程序找到一个换出页（如果是脏页则要写回磁盘）  Handler identifies victim (and, if dirty, pages it out to disk)

6) 异常处理程序拷贝页并更新页表条目  Handler pages in new page and updates PTE in memory

7)异常处理程序返回原进程中断的指令重新执行   Handler returns to original process, restarting faulting instruction

# 整合虚拟内存和Cache Integrating VM and Cache




*VA：虚拟地址 VA: virtual address, PA：物理地址 PA: physical address,*
*PTE：页表条目 PTE: page table entry, PTEA是页表条目地址 PTEA = PTE address*

# 使用TLB加速地址翻译
# Speeding up Translation with a TLB

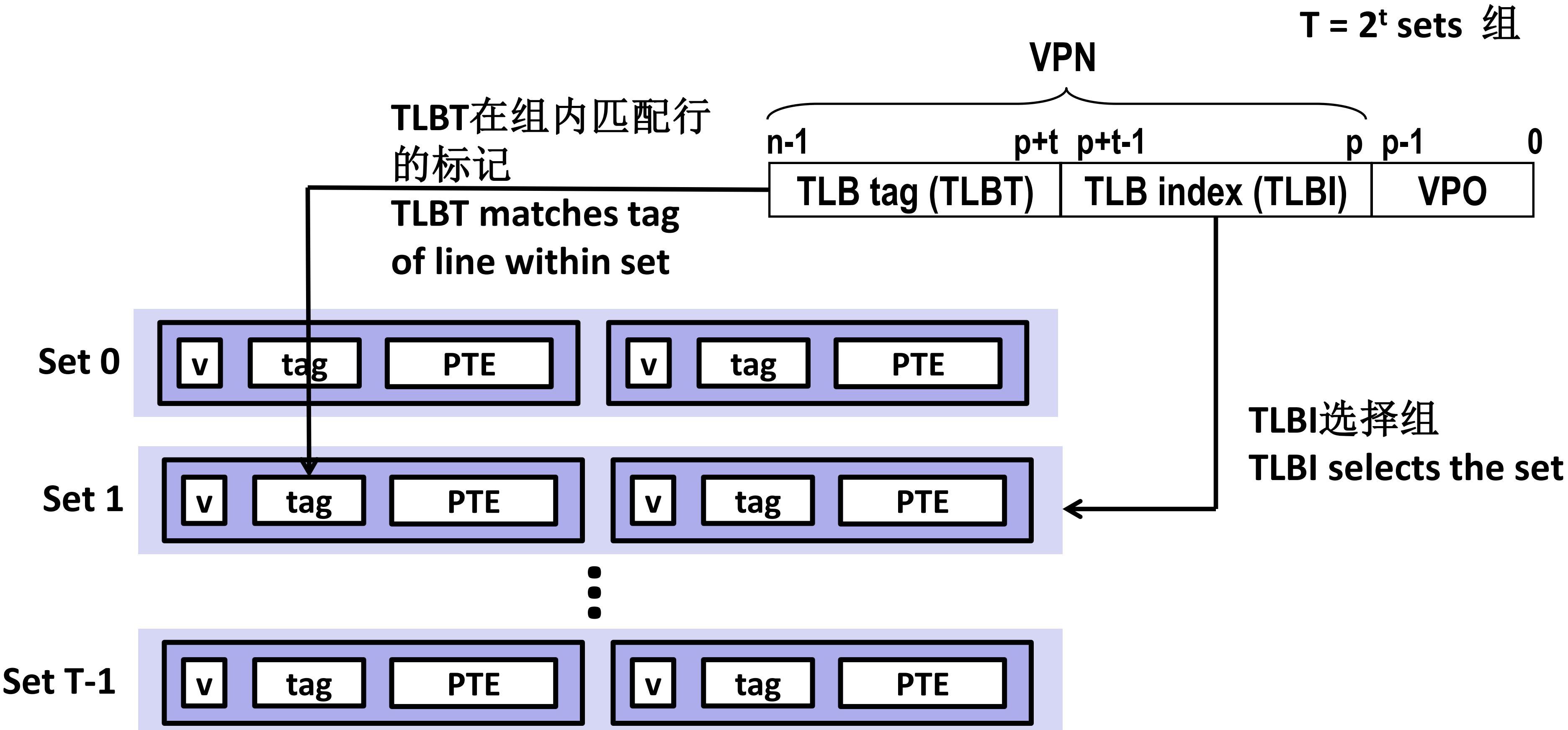
- 页表条目（**PTE**）像任何其他内存字一样缓存在**L1 cache**中 **Page table entries (PTEs) are cached in L1 like any other memory word**
  - 由于其他数据访问PTE可能会被驱逐出内存 PTEs may be evicted by other data references
  - PTE命中仍然需要较小的L1缓存延迟 PTE hit still requires a small L1 delay
- 解决方案：*翻译后备缓冲区*（**TLB**） **Solution: *Translation Lookaside Buffer* (TLB)**
  - 在MMU中的小型组相联硬件缓存 Small set-associative hardware cache in MMU
  - 将虚拟页号映射为物理页号 Maps virtual page numbers to physical page numbers
  - 包含了一少部分页面的完整页表条目 Contains complete page table entries for small number of pages

# 地址翻译符号总结
## Summary of Address Translation Symbols

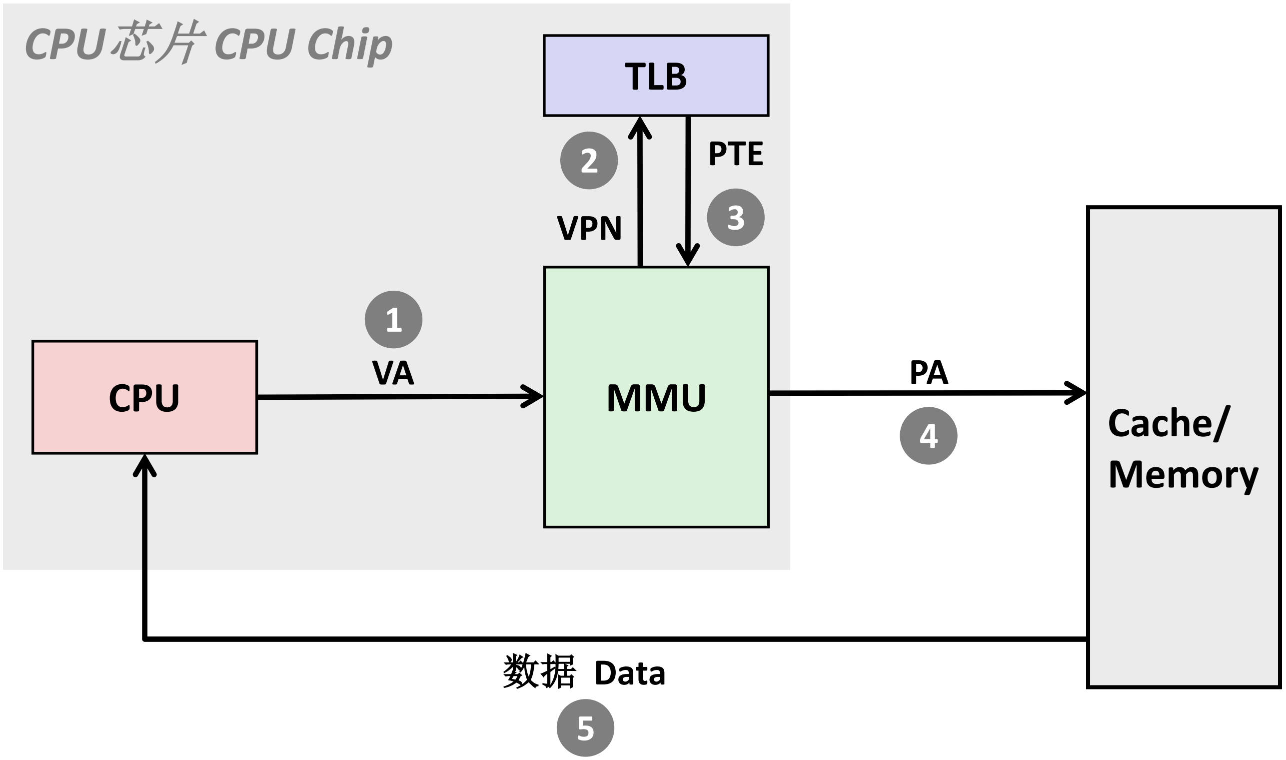
- 基本参数 **Basic Parameters**
  - **$N = 2^n$** : Number of addresses in virtual address space 虚拟地址空间的地址个数
  - **$M = 2^m$** : Number of addresses in physical address space 物理地址空间的地址个数
  - **$P = 2^p$** : Page size (bytes) 页大小（字节）
- 虚拟地址VA划分 **Components of the virtual address (VA)**
  - **TLBI**: TLB index TLB索引
  - **TLBT**: TLB tag TLB标记
  - **VPO**: Virtual page offset 虚拟页内偏移
  - **VPN**: Virtual page number 虚拟页号
- 物理地址PA划分 **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO) 物理页内偏移（同VPO）
  - **PPN:** Physical page number 物理页号

# 访问TLB   Accessing the TLB

- MMU使用虚拟地址的VPN部分访问TLB   MMU uses the VPN portion of the virtual address to access the TLB:

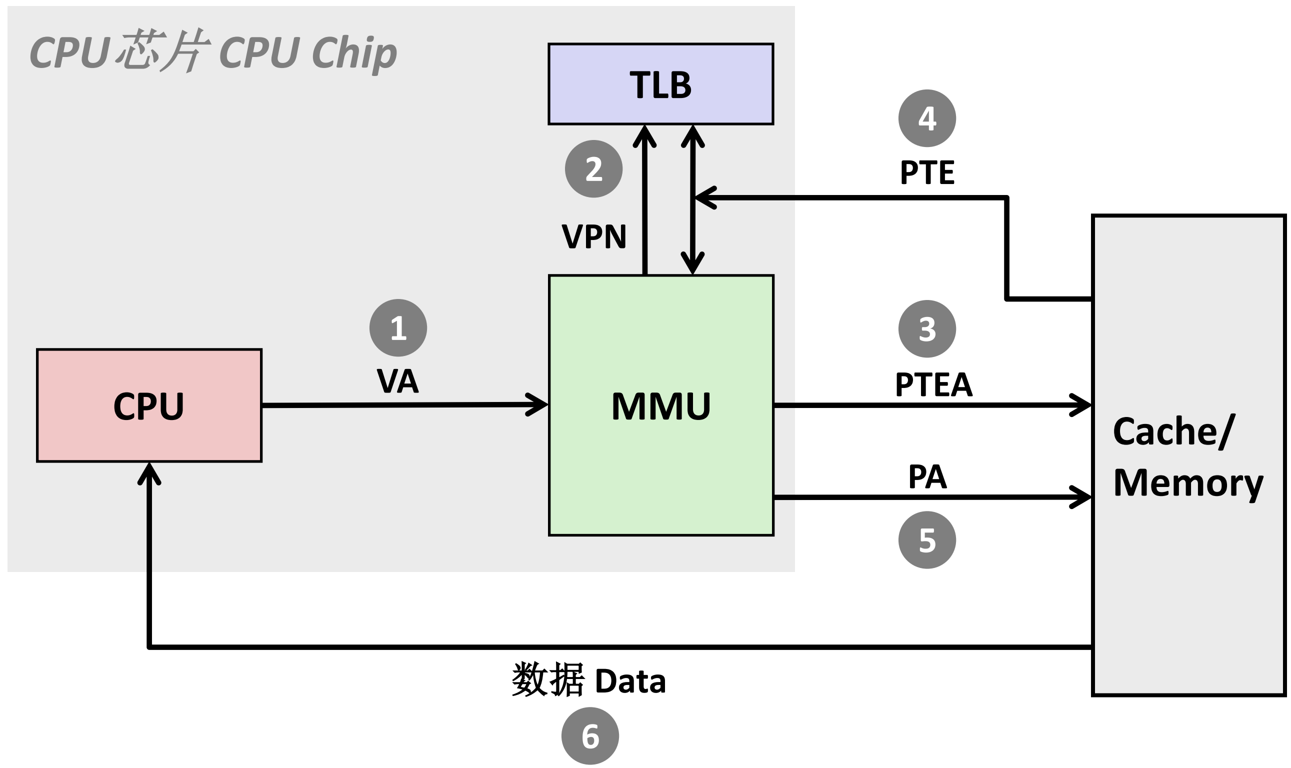


# TLB命中 TLB Hit




**TLB命中会减少一次内存访问 A TLB hit eliminates a memory access**

# TLB不命中 TLB Miss




**TLB不命中会导致一个额外的内存访问（页表条目）**

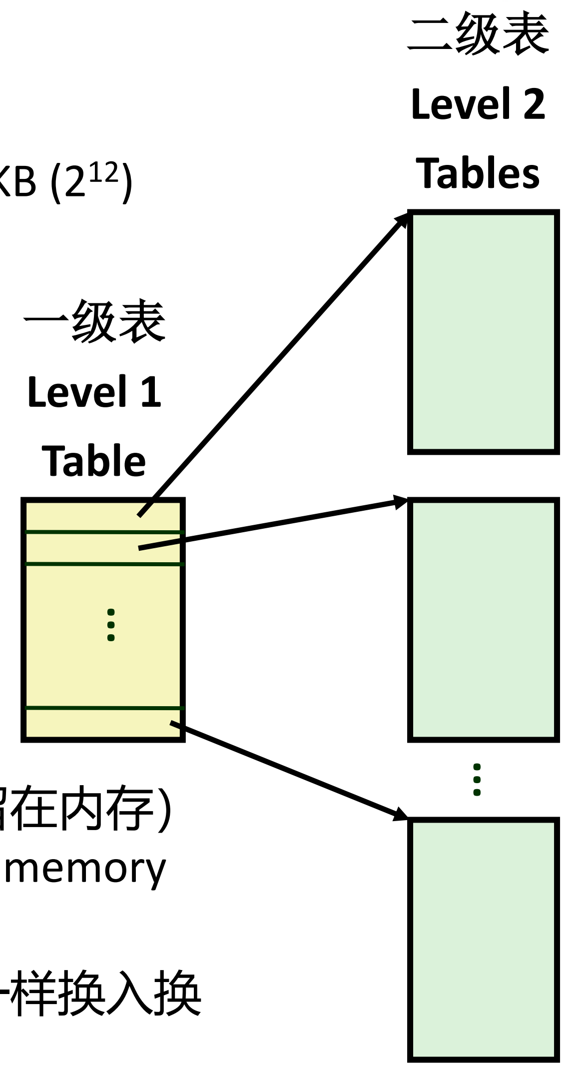
**幸运的是，TLB不命中很少发生。为何？**

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

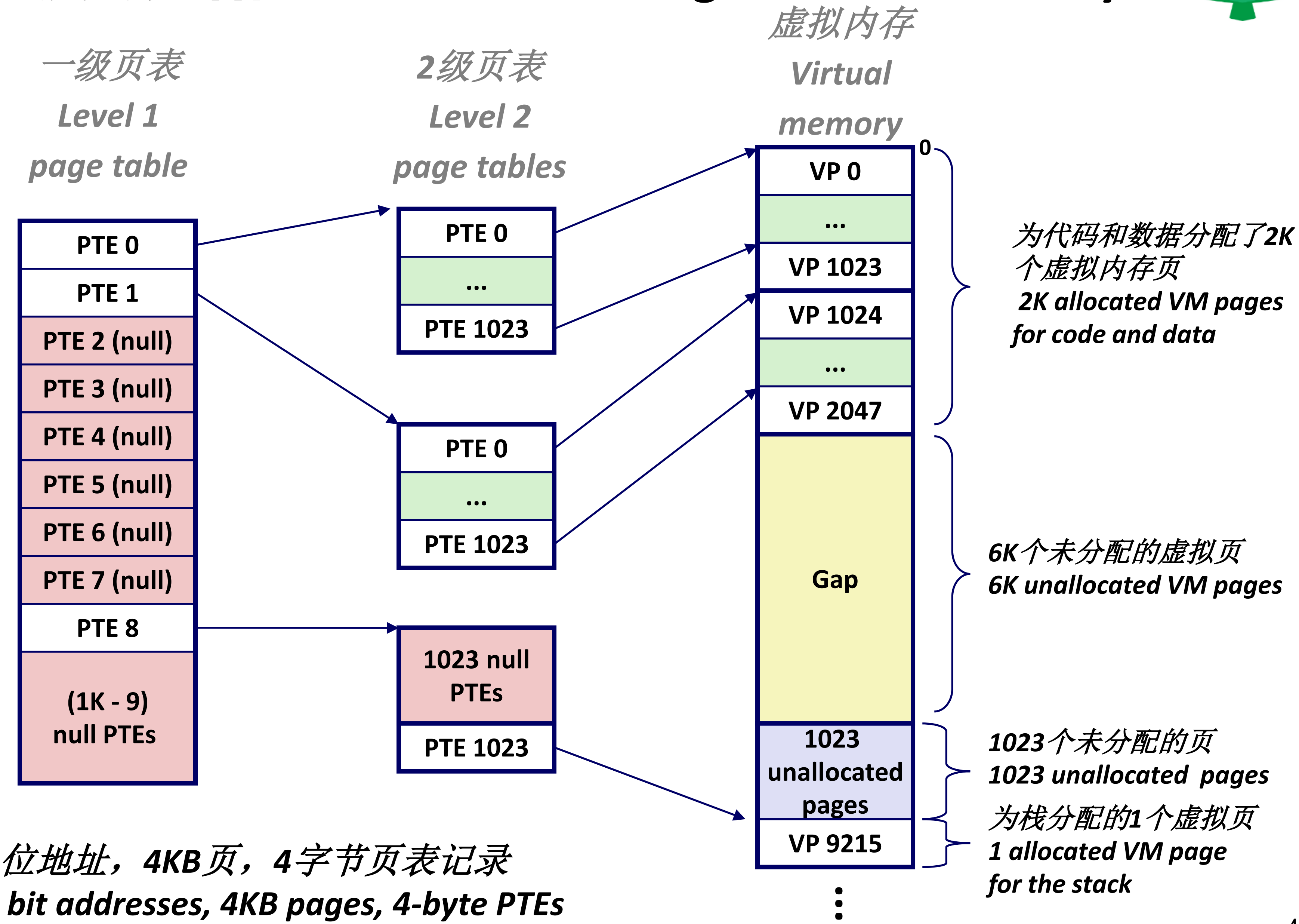
Fortunately, TLB misses are rare. Why?

# 多级页表 Multi-Level Page Tables

- **假设 Suppose:**
  - 4KB大小页表，48位地址空间，8字节页表记录 4KB ($2^{12}$) page size, 48-bit address space, 8-byte PTE
- **问题 Problem:**
  - 页表占用的空间将高达512GB
  - Would need a 512 GB page table!
    - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes
- **常见方法：多级页表 Common solution: Multi-level page table**
- **例如：2级页表 Example: 2-level page table**
  - 一级页表：每个页表记录指向一个页表（总是驻留在内存）Level 1 table: each PTE points to a page table (always memory resident)
  - 二级页表：每个页表记录指向一个页（像其他页一样换入换出） Level 2 table: each PTE points to a page (paged in and out like any other data)

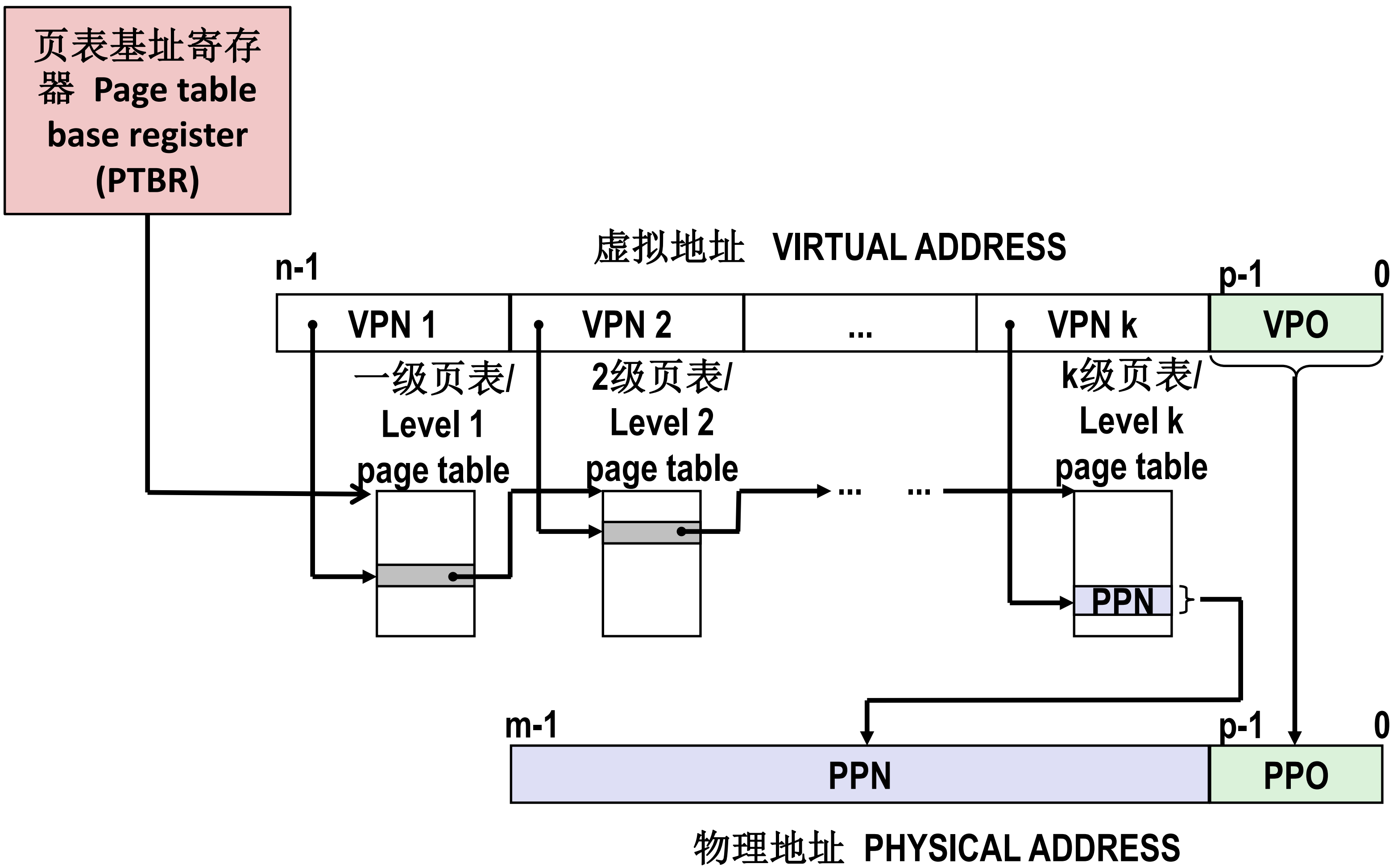


# 二级页表结构 A Two-Level Page Table Hierarchy




***32位地址，4KB页，4字节页表记录***
***32 bit addresses, 4KB pages, 4-byte PTEs***

# k级页表的地址翻译
# Translating with a k-level Page Table



# 总结 Summary

- 程序员眼中的虚拟内存 **Programmer's view of virtual memory**
  - 每个进程都有各自私有的线性地址空间 Each process has its own private linear address space
  - 不能被其他进程破坏 Cannot be corrupted by other processes
- 系统眼中的虚拟内存 **System view of virtual memory**
  - 通过缓存虚拟内存页高效地使用内存 Uses memory efficiently by caching virtual memory pages
    - 高效是因为局部性 Efficient only because of locality
  - 简化内存管理和编程 Simplifies memory management and programming
  - 通过提供方便的库打桩点来检查权限，简化了保护 Simplifies protection by providing a convenient interpositioning point to check permissions