第9章 虚拟内存: 基本概念

本ppt为cmu csapp的中文翻译

主要内容

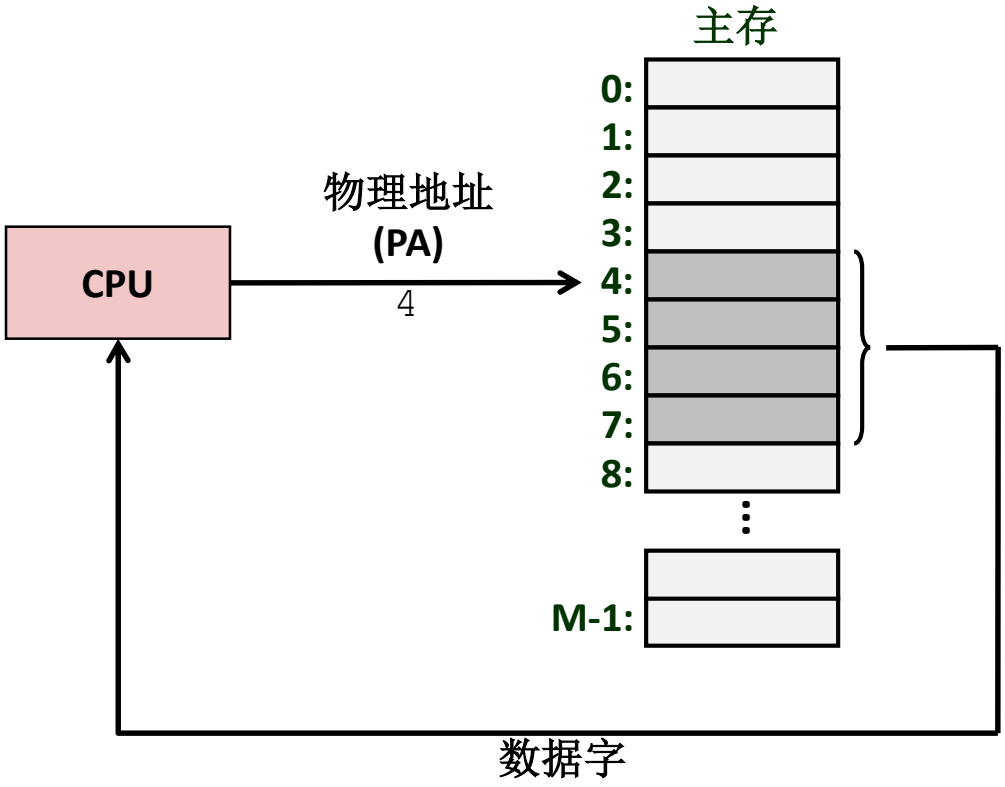
- 地址空间
- 虚拟内存作为缓存的工具
- 虚拟内存作为内存管理的工具
- 虚拟内存作为内存保护的工具
- 地址翻译

为什么要使用虚拟内存?

虚拟内存是对主存的抽象，提供了三个能力：

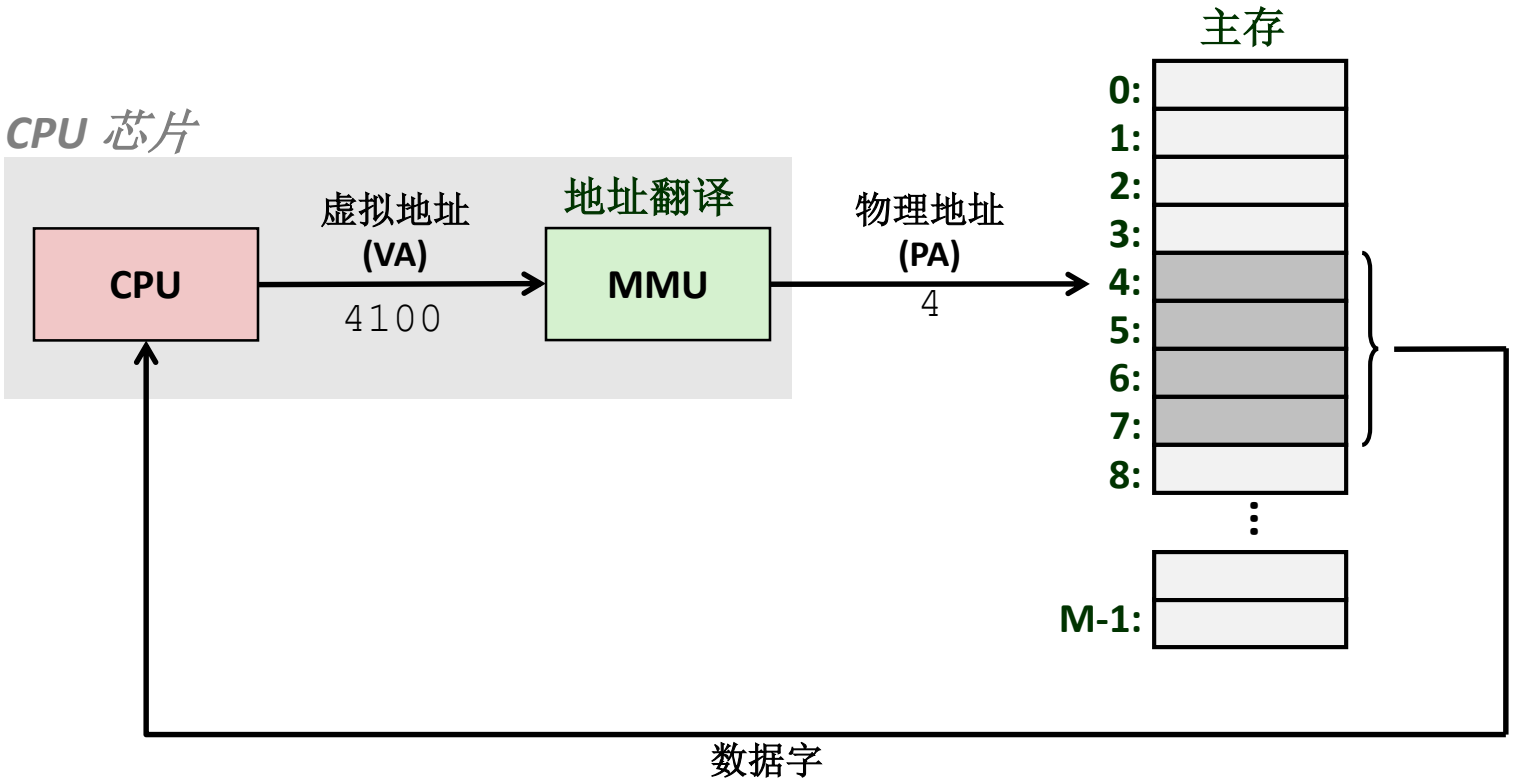
- **可以有效使用主存**
  - 将主存看作一个存储在磁盘上的地址空间的缓存（虚存建立在磁盘上，可以很大）
  - 主存中只保存活动区域
- **可以简化内存管理（一致性）**
  - 为每个进程提供统一的线性地址空间
- **提供独立地址空间（私有的）**
  - 一个进程不能影响其他进程的内存
  - 用户程序无法获取特权内核信息和代码

一个使用物理寻址的系统



- 在数字信号处理器、嵌入式微控制器中使用物理寻址

一个使用虚拟寻址的系统



- 现在处理器使用虚拟寻址, 比如笔记本、智能电话等应用
- 计算机科学的伟大思想之一

概念：地址空间

- **地址空间:** 非负整数地址的有序集合

  {0, 1, 2, 3 … }

  如果地址空间中的整数是连续的，称为***线性地址空间***

- **虚拟地址空间:** N = 2^n 个虚拟地址的集合

  {0, 1, 2, 3, …, N-1}

- **物理地址空间:** M = 2^m 个物理内存地址的集合

  {0, 1, 2, 3, …, M-1}

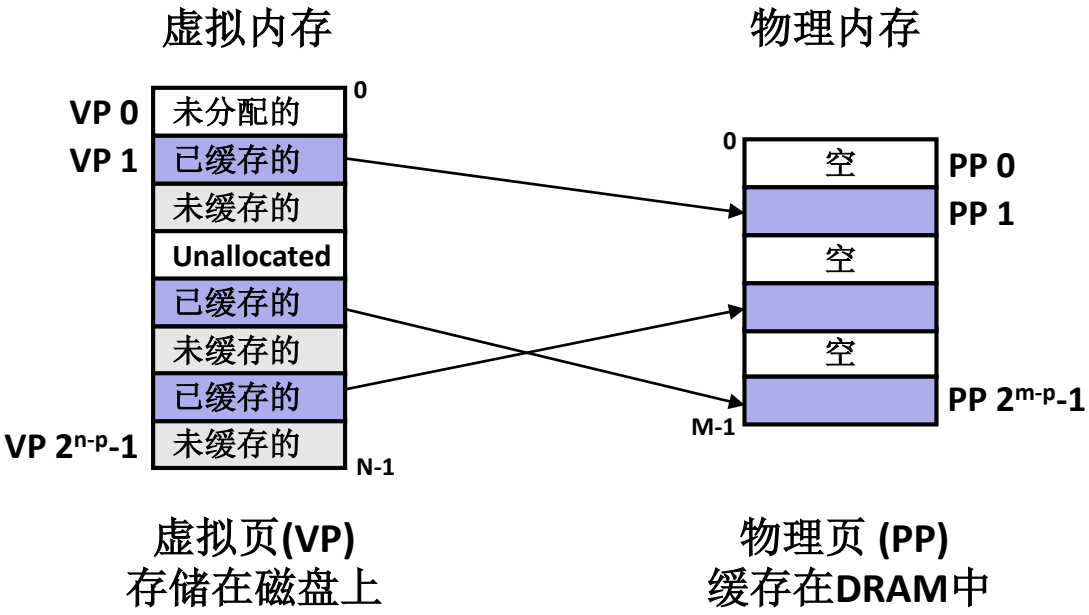
数据对象与地址是独立的，每个数据对象可以有多个独立的地址，这些地址来自不同地址空间

主要内容

- 地址空间
- **虚拟内存作为缓存的工具**
- 虚拟内存作为内存管理的工具
- 虚拟内存作为内存保护的工具
- 地址翻译

虚拟内存作为缓存的工具

- 概念上而言，虚拟内存被组织为一个由存放在磁盘上的N个连续字节大小的单元组成的数组.
- 磁盘上数组的内容被缓存在*物理内存中* (***DRAM cache***)
  - 这些内存块被称为页 (每个页面的大小为P = 2^p字节)

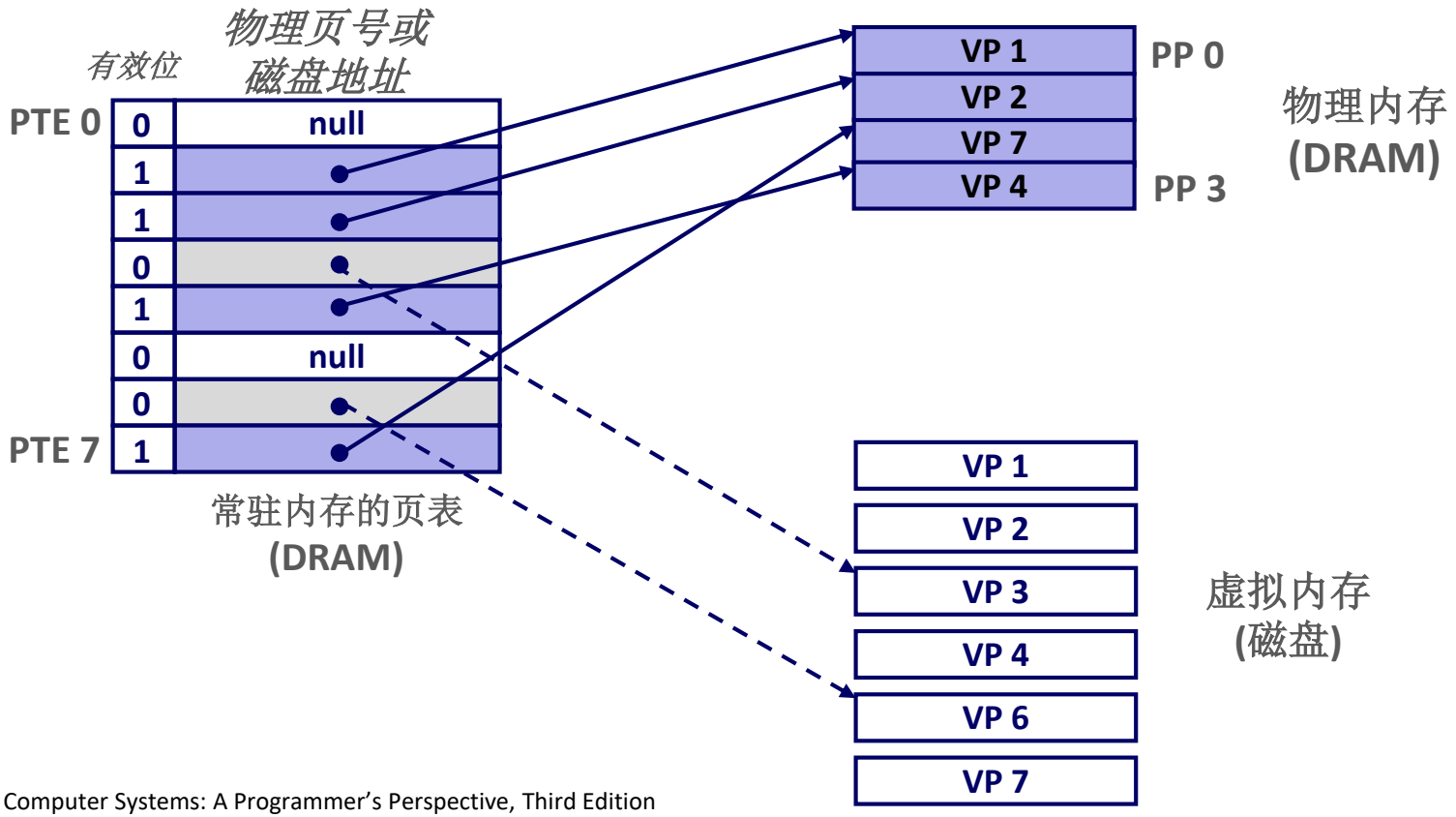


DRAM缓存的组织结构

- **DRAM 缓存的结构特点由巨大的不命中开销决定**
  - DRAM 比 SRAM 慢大约 ***10倍***
  - 磁盘比 DRAM 慢大约 ***100,000倍***
- **因此**
  - 虚拟页很大: 标准 4 KB, 有时可以达到 4 MB
  - DRAM缓存为全相联
    - 任何虚拟页都可以放置在任何物理页中
    - 需要一个更大的映射函数 – 不同于硬件对SRAM缓存
  - 更复杂精密的替换算法
    - 太复杂且无限制以致无法在硬件上实现
  - DRAM缓存总是使用写回，而不是直写

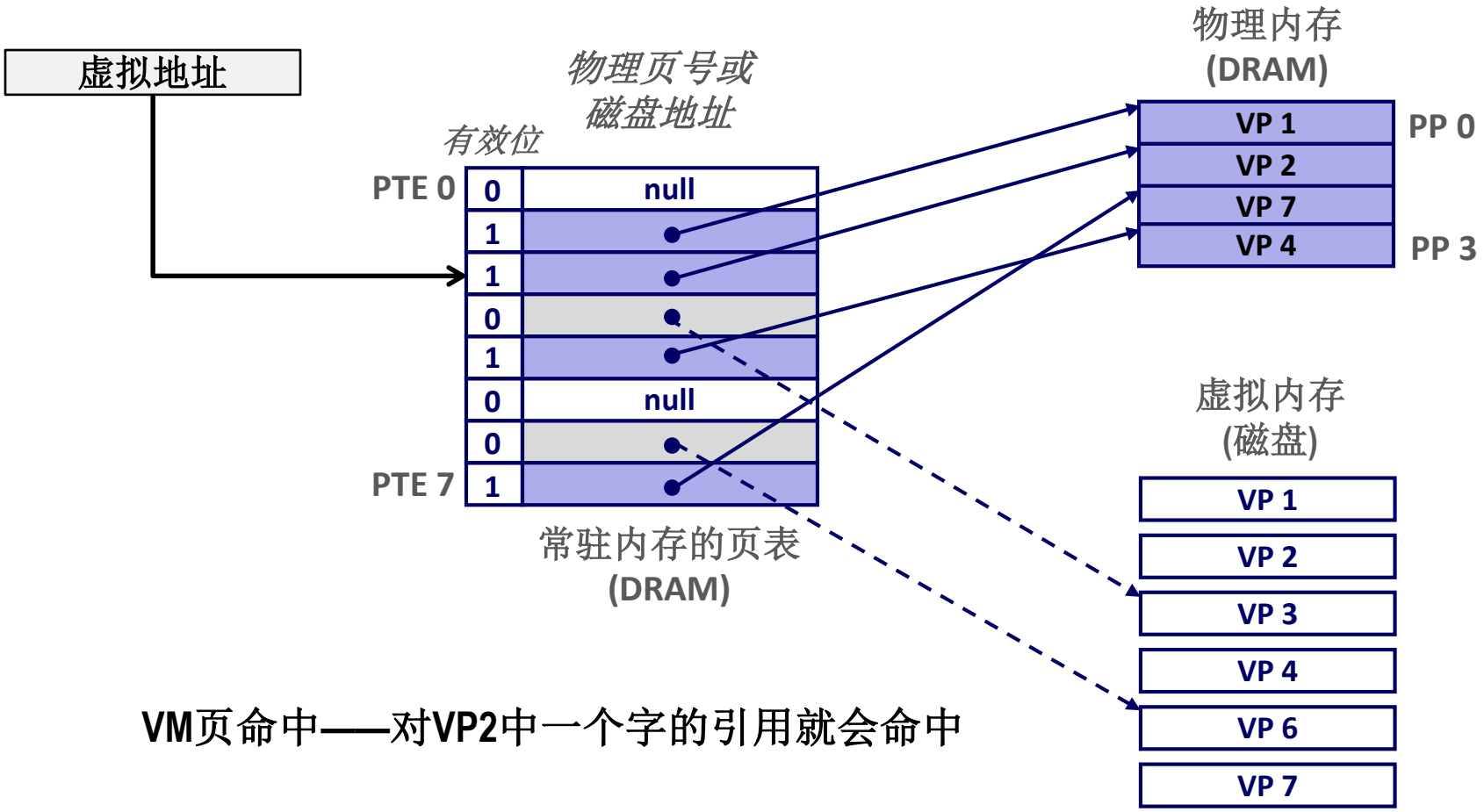
页表

- *页表*是一个页表条目 (Page Table Entry, PTE)的数组，将虚拟页地址映射到物理页地址。
  - 虚拟地址空间中的每个页（VP）在页表固定位置有一个PTE
  - PTE由一个有效位和一个n位地址字段组成，有效位表示虚页是否被缓存
  - 有效位为0，地址表示VP在磁盘位置或地址为空表示虚拟页还未分配



页命中

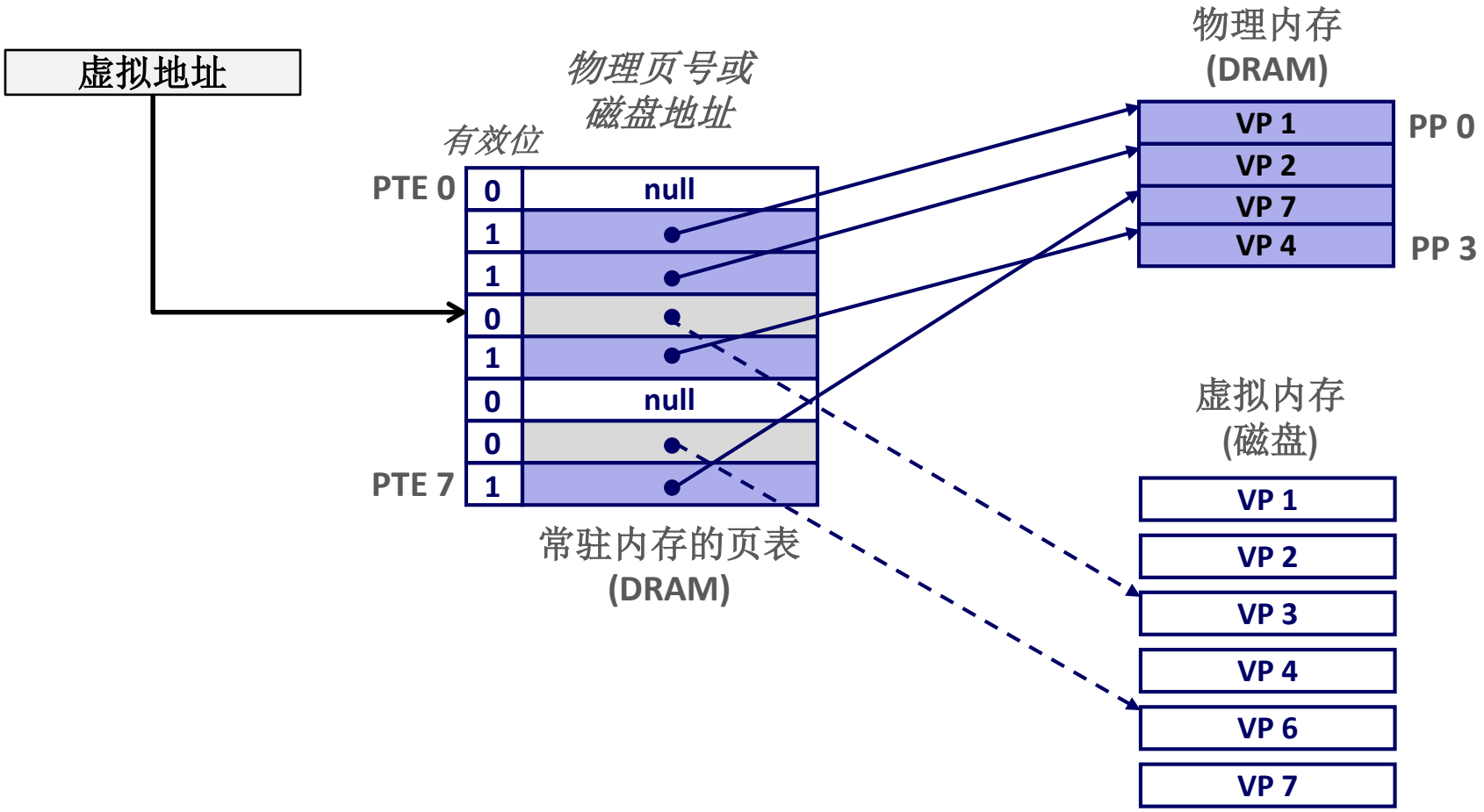
- ***Page hit页命中:*** 虚拟内存中 的 一个字存在于物理内存中，即(DRAM)缓存命中



VM页命中——对VP2中一个字的引用就会命中

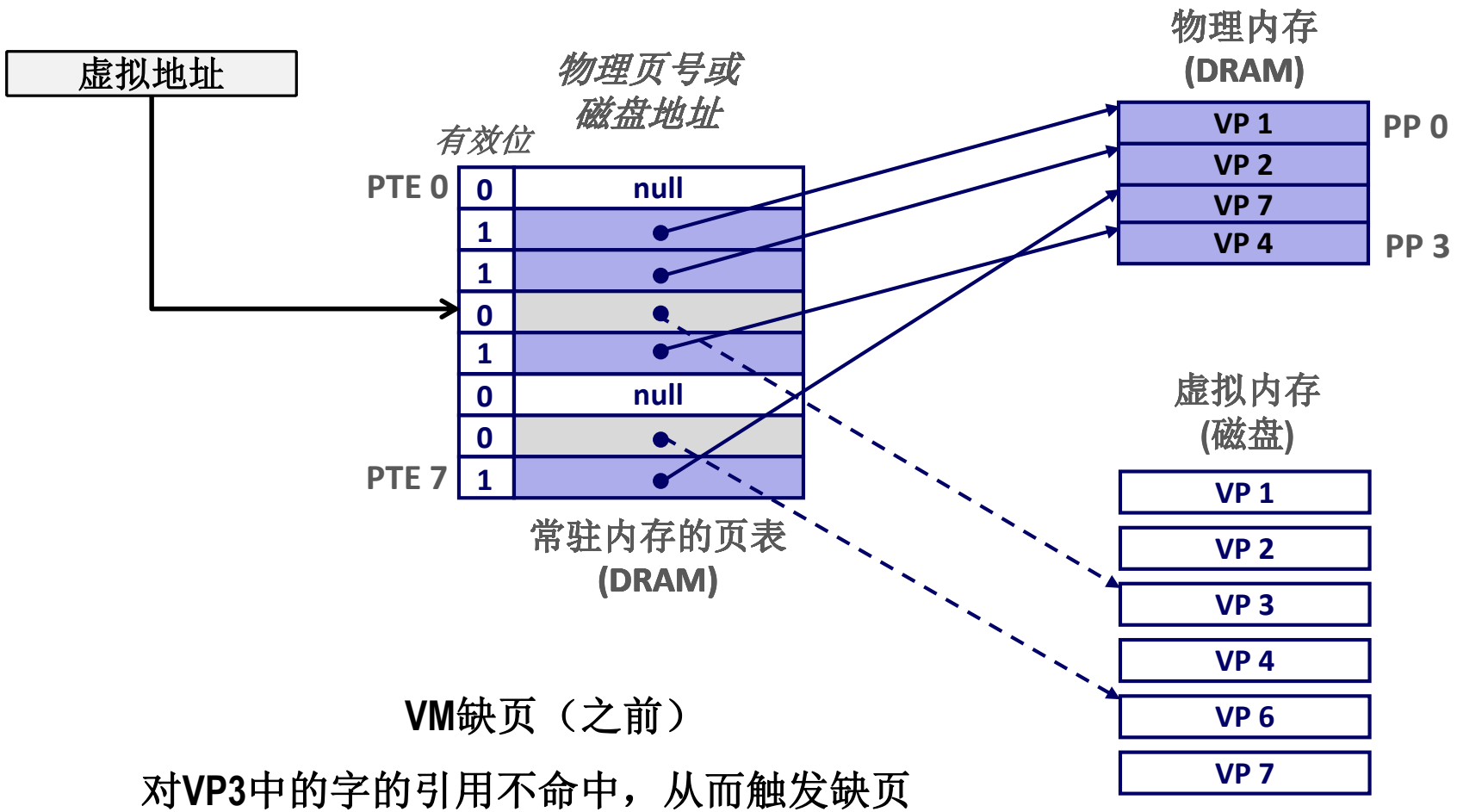
缺页

- ***Page fault缺页:*** 虚拟内存中的字不在物理内存中 (DRAM 缓存不命中)



缺页处理

- Page miss causes page fault (an exception)
  缺页导致页面出错 (缺页异常)

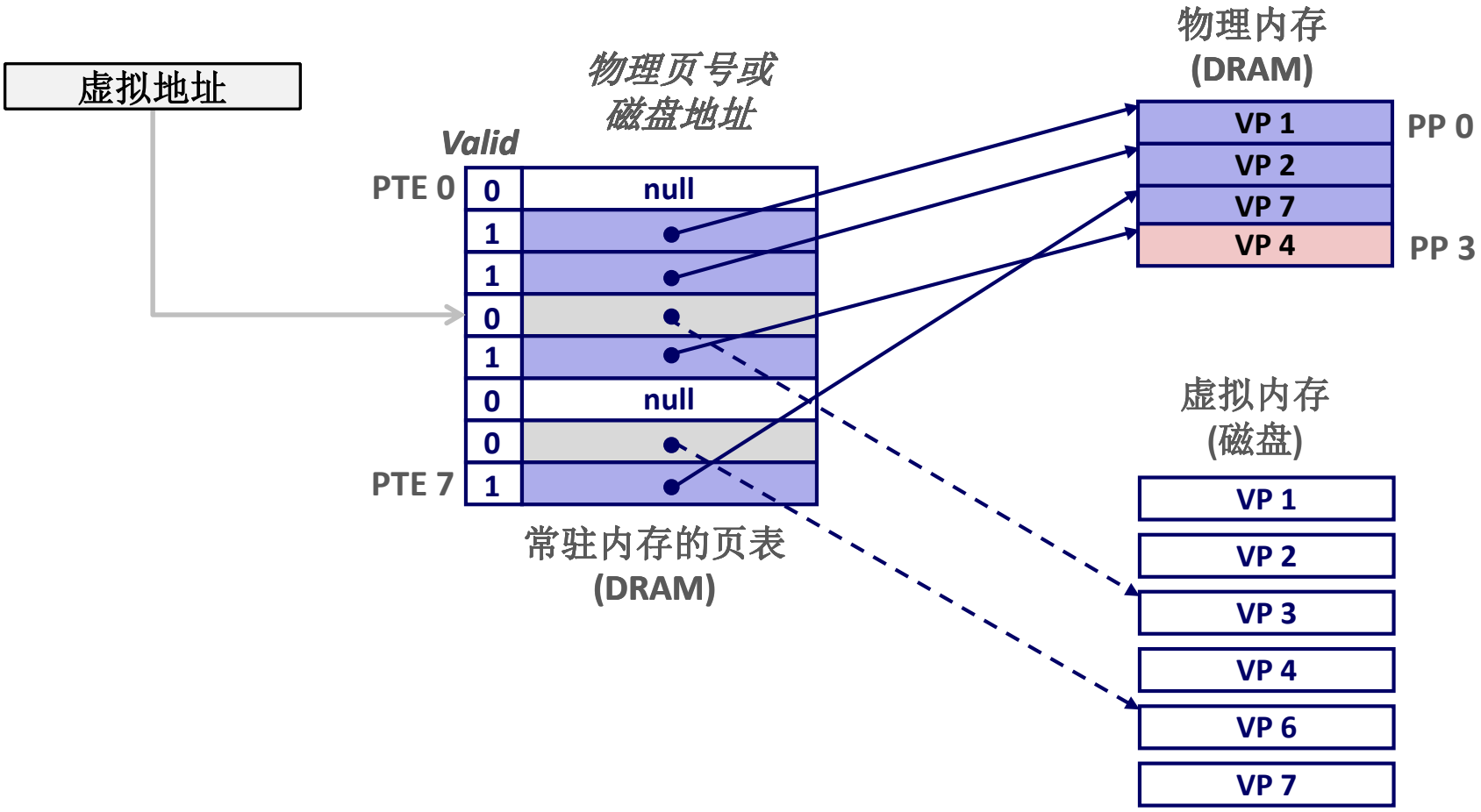


VM缺页（之前）

对VP3中的字的引用不命中，从而触发缺页

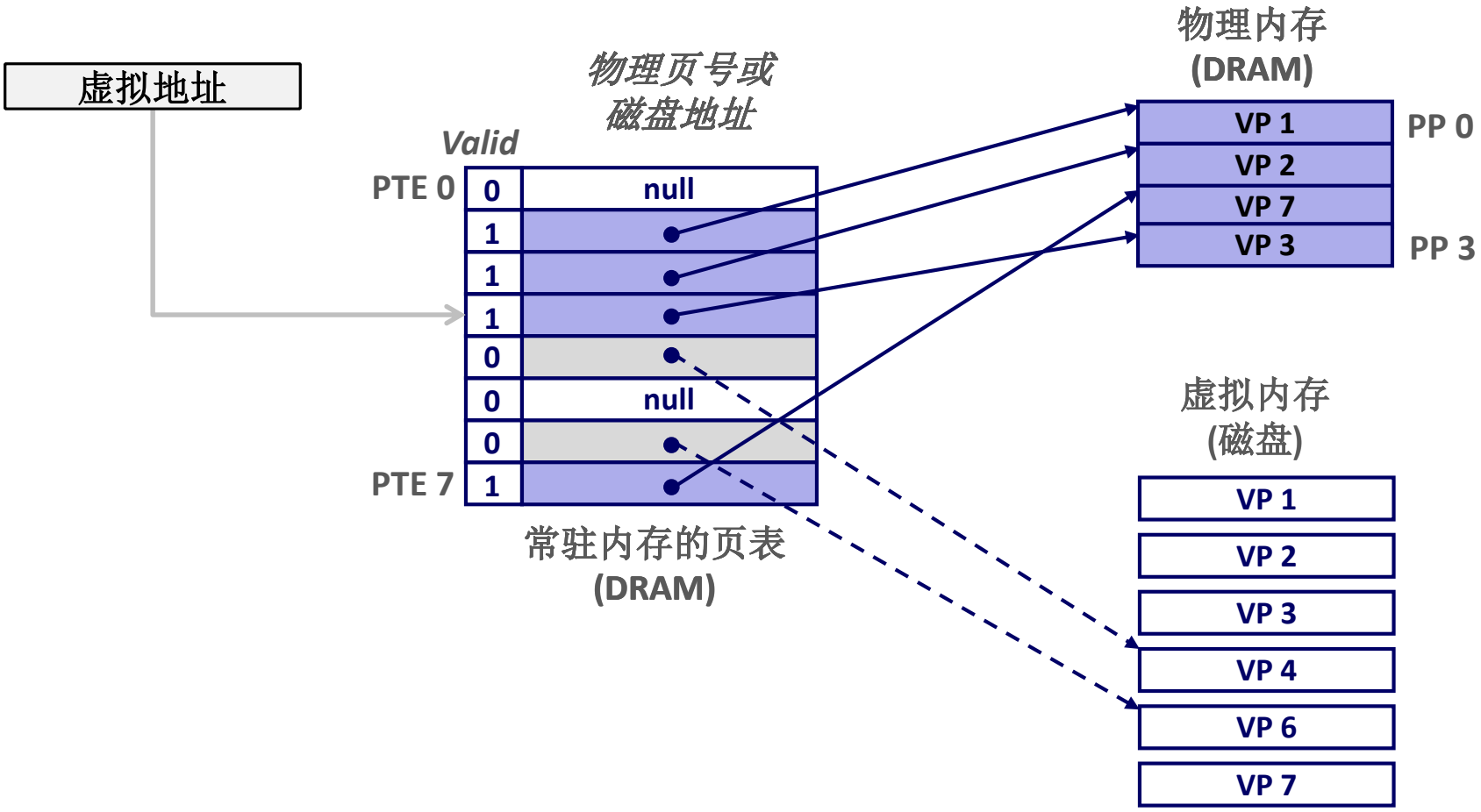
缺页处理

- 缺页导致页面出错 (缺页异常)
- 缺页异常处理程序选择一个牺牲页 (此例中就是 VP 4)



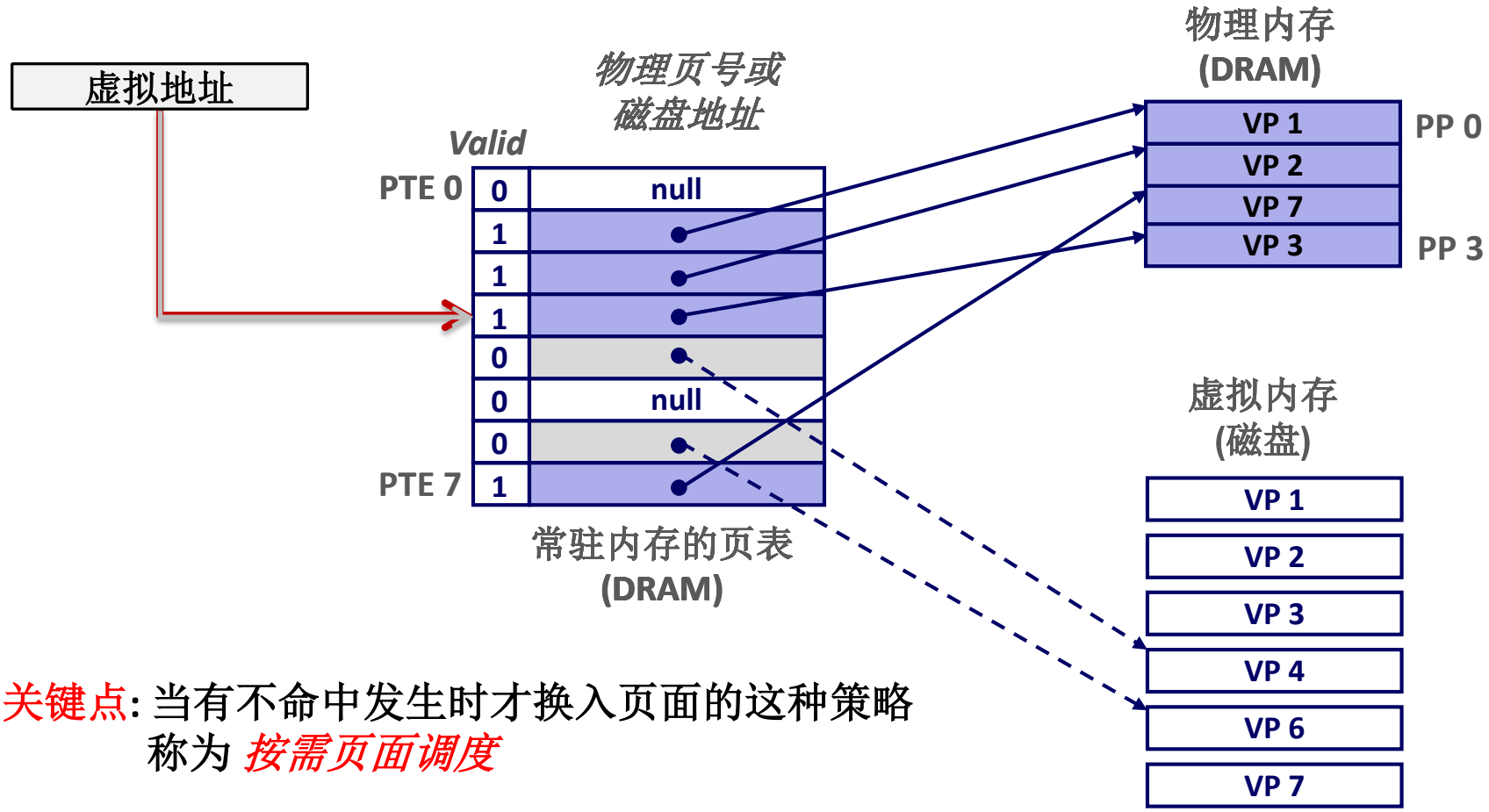
缺页处理

- 缺页导致页面出错 (缺页异常)
- 缺页异常处理程序选择一个牺牲页 (此例中就是 VP 4)



缺页处理

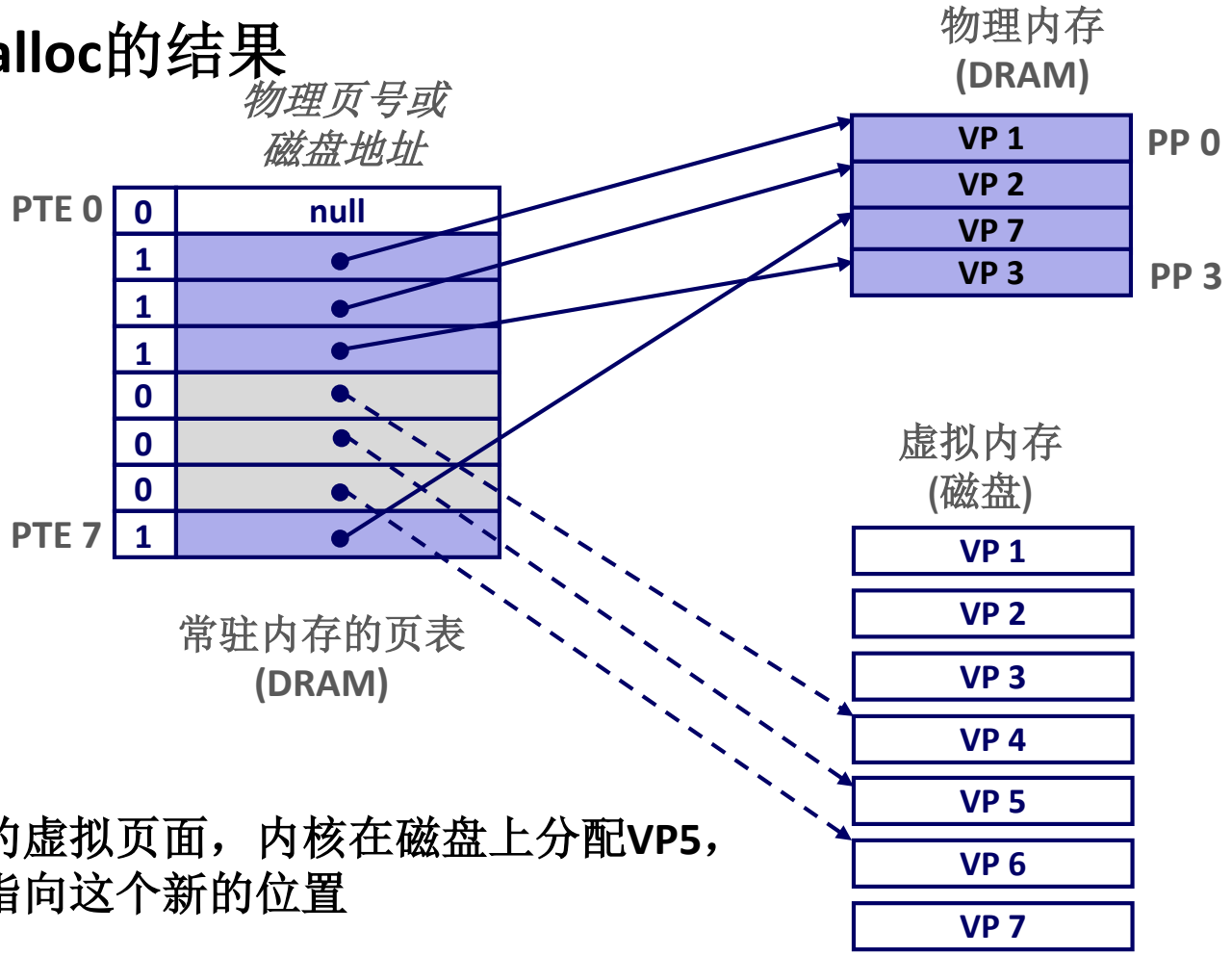
- 缺页导致页面出错 (缺页异常)
- 缺页异常处理程序选择一个牺牲页 (此例中就是 VP 4)
- 导致缺页的指令重新启动: 页面命中!



关键点: 当有不命中发生时才换入页面的这种策略
称为 *按需页面调度*

分配页面

- 分配一个新的虚拟内存页 (VP 5).
- 如：调用malloc的结果



分配一个新的虚拟页面，内核在磁盘上分配VP5，
并且将PTE5指向这个新的位置

又是局部性救了我们!

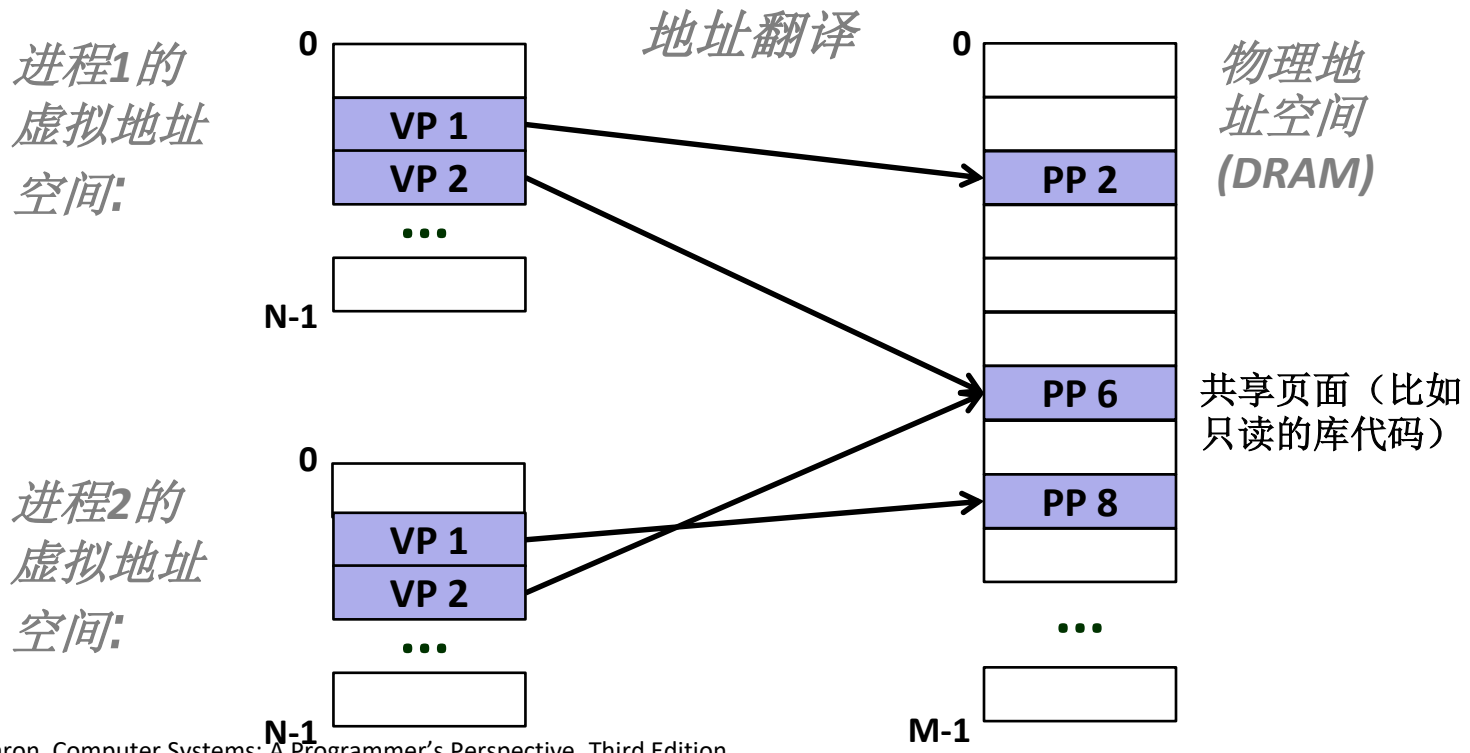
- **虚拟内存看上去效率非常低, 但它工作得相当好，这都要归功于“局部性”.**
- **在任意时间，程序将趋于在一个较小的活动页面集合上工作，这个集合叫做 *工作集 Working set***
  - 程序的时间局部性越好，工作集就会越小
- **如果 (工作集的大小 < 物理内存的大小)**
  - 在工作集页面调度到内存后（初始开销），对工作集的引用将导致命中。
- **如果 (工作集的大小 >物理内存的大小)**
  - *Thrashing**抖动:*** 页面不断地换进换出，导致系统性能崩溃。

主要内容

- 地址空间
- 虚拟内存作为缓存的工具
- **虚拟内存作为内存管理的工具**
- 虚拟内存作为内存保护的工具
- 地址翻译

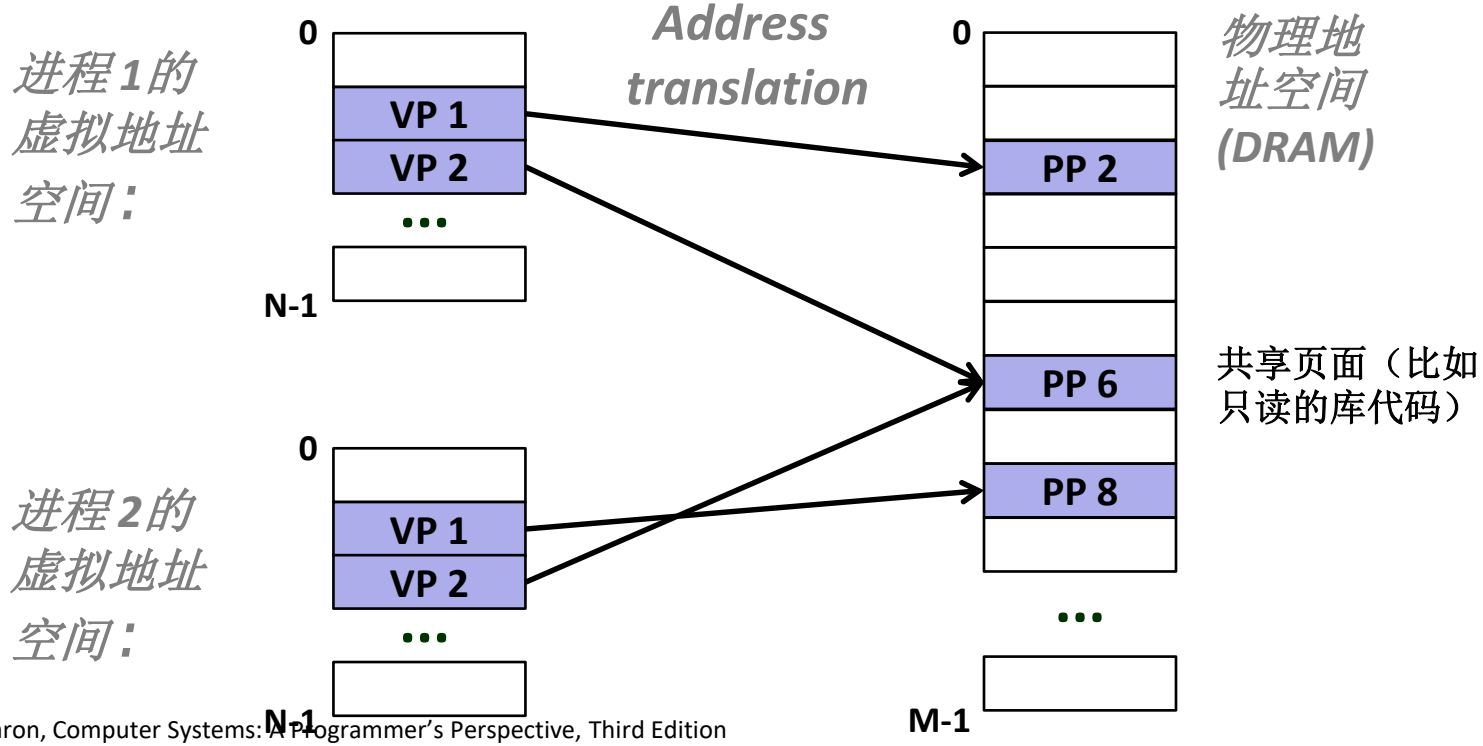
虚拟内存作为内存管理的工具

- **核心思想: 每个进程都拥有一个独立的虚拟地址空间**
  - 把内存看作独立的简单线性数组
  - 操作系统为系统中每个进程都维护一个单独的页表
    - 也就是一个独立的虚拟地址空间



虚拟内存作为内存管理的工具

- 简化内存分配
  - 每个虚拟内存页面都要被映射到一个物理页面
  - 一个虚拟内存页面可以被分配到任意一个物理页面（可以不连续）
- 简化代码和数据共享
  - 不同的虚拟内存页面被映射到相同的物理页面 (此例中的 PP 6)



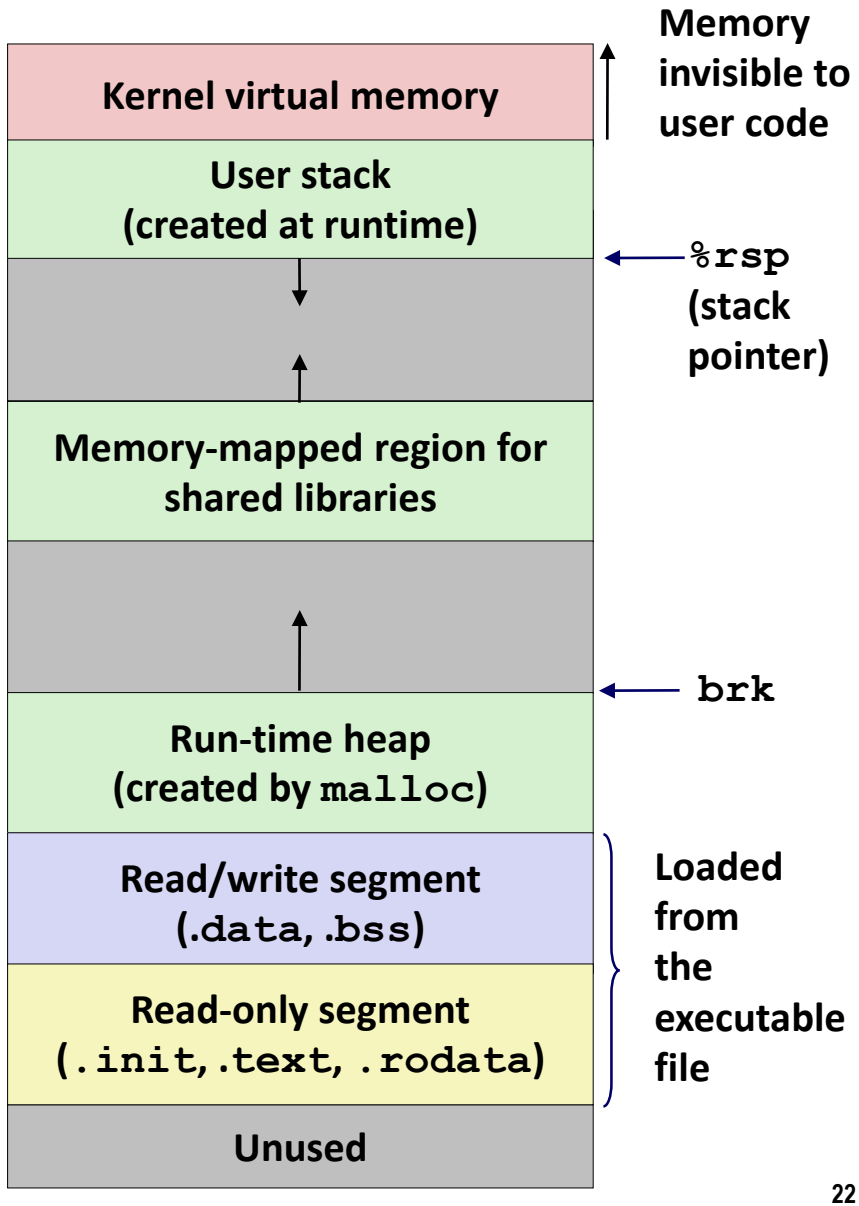
简化链接和加载

Linking 链接

- 独立的地址空间允许每个程序使用相似的虚拟地址空间
- 代码、数据和堆都使用相同的起始地址.

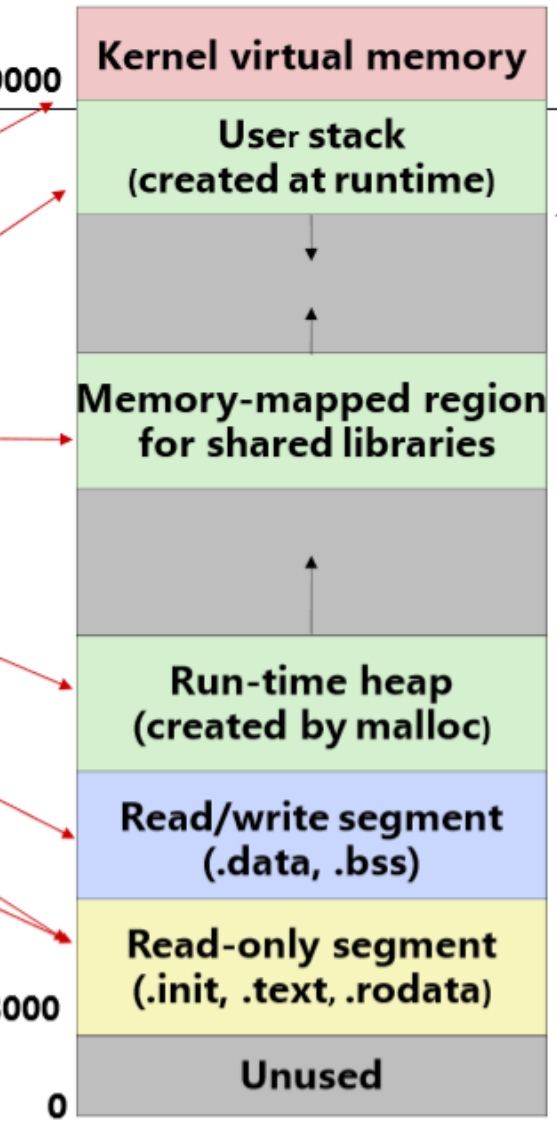
Loading 加载

- `execve` 为代码段和数据段分配虚拟页，并标记为无效（即未被缓存）
- 每个页面被初次引用时，虚拟内存系统会按照需要自动地调入数据页。



虚拟地址空间

- Linux在X86上的虚拟地址空间
  （其他Unix系统的设计类此）
  - 内核空间（Kernel）
  - 用户栈（User Stack）
  - 共享库（Shared Libraries）
  - 堆（heap）
  - 可读写数据（Read/Write Data）
  - 只读数据（Read-only Data）
  - 代码（Code）



问题：加载时是否真正从磁盘调入信息到主存？

实际上不会从磁盘调入，只是将虚拟页和磁盘上的数据/代码建立对应关系，称为“映射”。

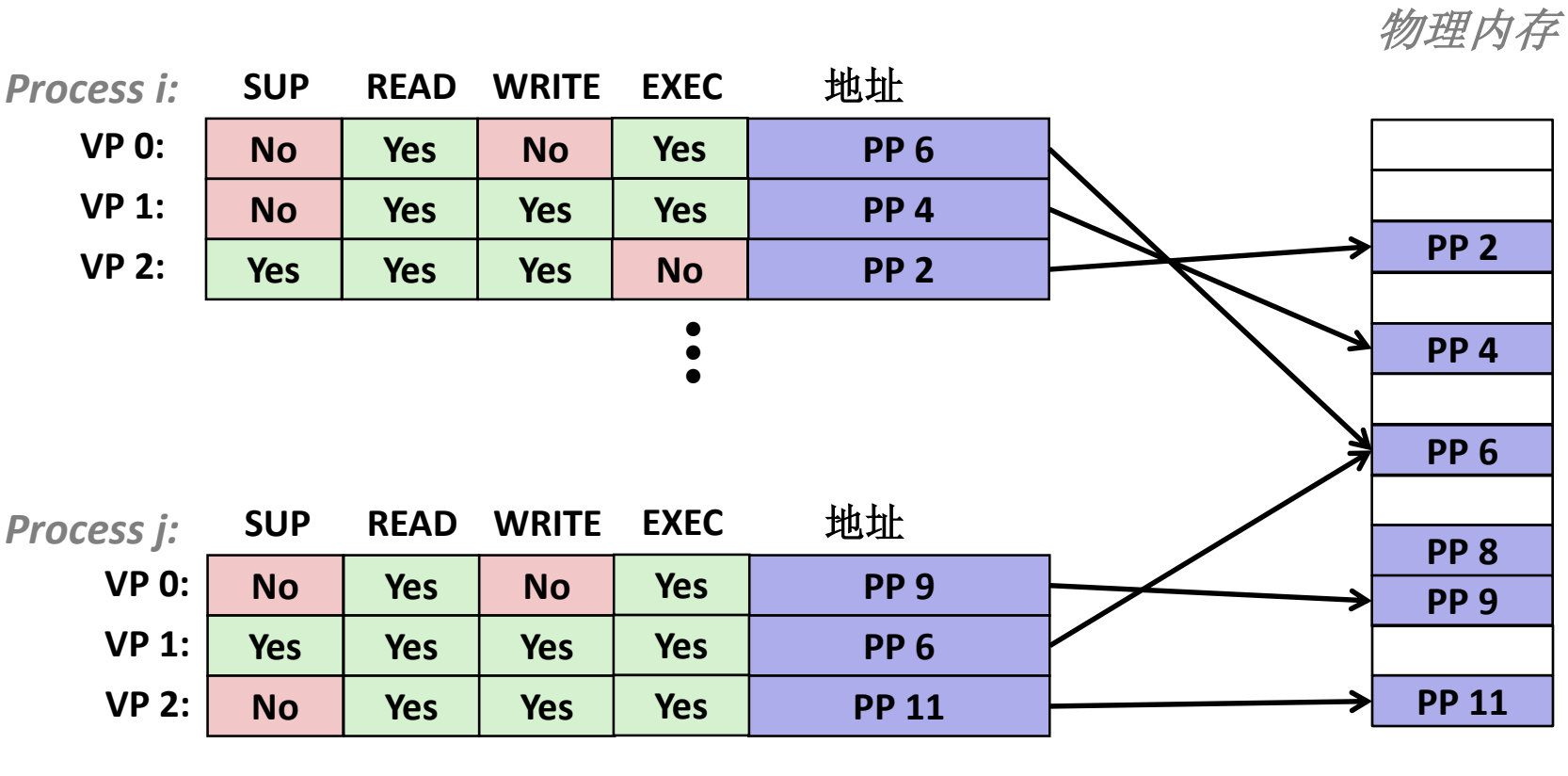
可执行文件相关内容“复制”到代码段和数据段

主要内容

- 地址空间
- 虚拟内存作为缓存的工具
- 虚拟内存作为内存管理的工具
- **虚拟内存作为内存保护的工具**
- 地址翻译

虚拟内存作为内存保护的工具

- 在 PTE 上扩展许可位以提供更好的访问控制
- 内存管理单元（MMU）每次访问数据都要检查许可位



带许可位的页表

信息访问中可能出现的异常情况

可能有两种异常情况：

1）缺页（ page fault ）

产生条件：当Valid（有效位 / 装入位）为 0 时

相应处理：从磁盘读到内存，若内存没有空间，则还要从内存选择一页替换到磁盘上，替换算法类似于Cache，采用回写法，淘汰时，根据“dirty”位确定是否要写磁盘

当前指令执行被阻塞，当前进程被挂起，处理结束回到原指令继续执行

2）保护违例（ protection_violation_fault ）或访问违例

产生条件： 当Access Rights (存取权限)与所指定的具体操作不相符时

相应处理：在屏幕上显示“内存保护错”或“访问违例”信息

当前指令的执行被阻塞，当前进程被终止

Access Rights (存取权限)可能的取值有哪些？

R = Read-only, R/W = read/write, X = execute only

主要内容

- 地址空间
- 虚拟内存作为缓存的工具
- 虚拟内存作为内存管理的工具
- 虚拟内存作为内存保护的工具
- **地址翻译**

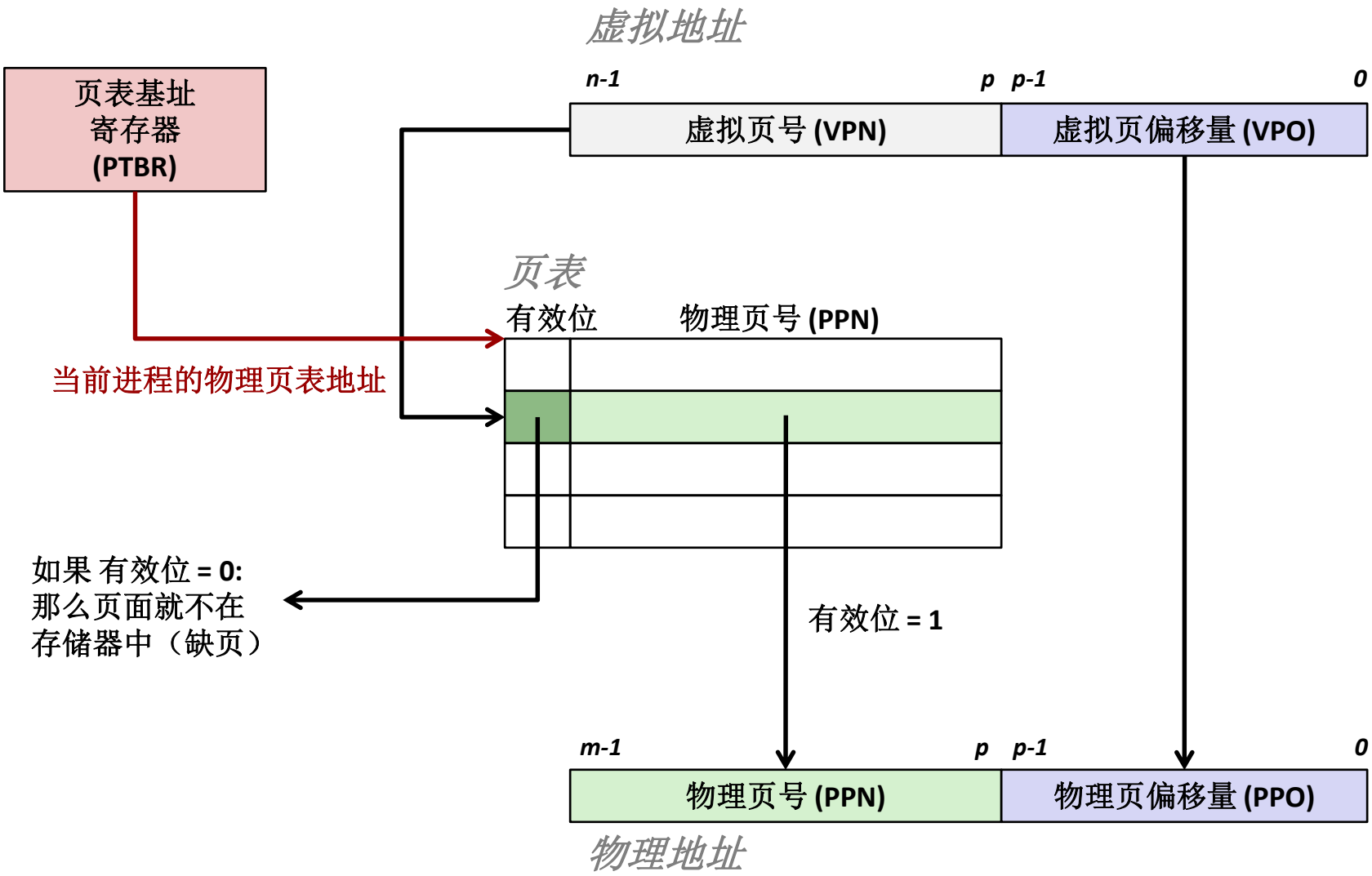
VM Address Translation 地址翻译

- **Virtual Address Space 虚拟地址空间**
  - $V = \{0, 1, \ldots, N-1\}$
- **Physical Address Space 物理地址空间**
  - $P = \{0, 1, \ldots, M-1\}$
- **Address Translation 地址翻译**
  - $\boldsymbol{MAP: V \rightarrow P \cup \varnothing}$
  - For virtual address ***a***:
    - ***MAP(a) = a'*** 如果虚拟地址 ***a***处的数据在物理地址空间的 ***a'*** 处
    - ***MAP(a) =*** $\varnothing$ 如果虚拟地址 ***a***处的数据不在物理内存中
      - 不论无效地址还是存储在磁盘上

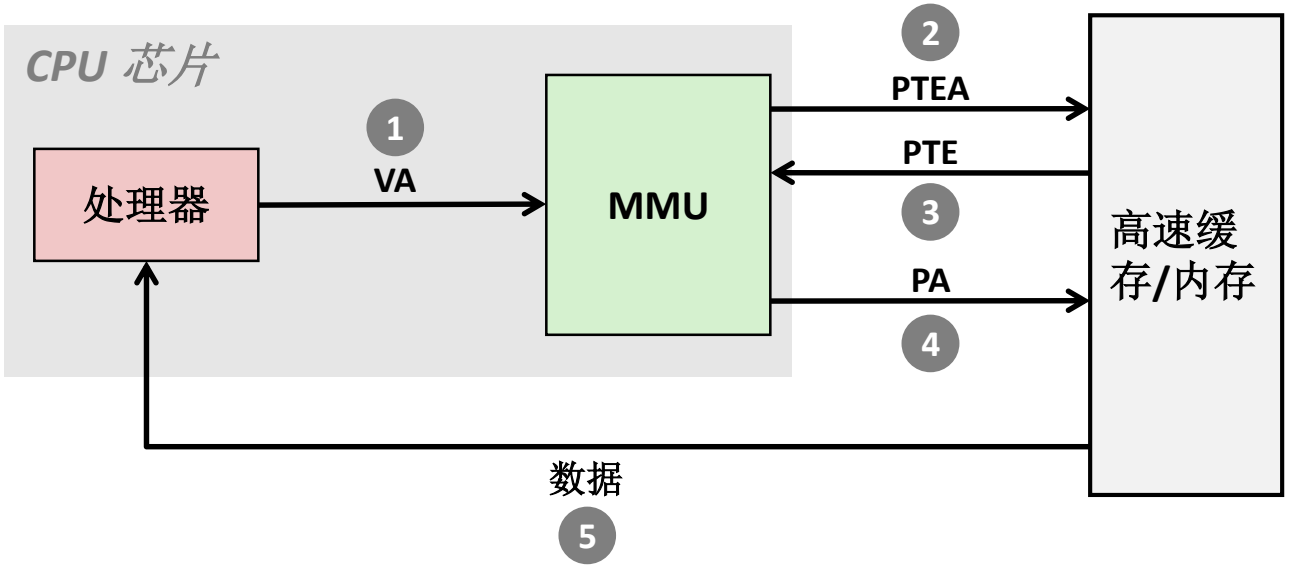
地址翻译使用到的所有符号

- **基本参数**
  - **N = 2^n** : 虚拟地址空间中的地址数量
  - **M = 2^m** : 物理地址空间中的地址数量
  - **P = 2^p** ：页的大小 (bytes)
- **虚拟地址(VA)的组成部分**
  - **TLBI**: ----TLB索引
  - **TLBT**: ----TLB标记
  - **VPO**: ----虚拟页面偏移量（字节）
  - **VPN**: ----虚拟页号
- **物理地址(PA)的组成部分**
  - **PPO**: ----物理页面偏移量（与VPO值相同）
  - **PPN:** ----物理页号

基于页表的地址翻译


虚拟地址
n-1
p
p-1
0
虚拟页号 (VPN)
虚拟页偏移量 (VPO)
页表基址
寄存器
(PTBR)
页表
有效位
物理页号 (PPN)
当前进程的物理页表地址
如果 有效位 = 0:
那么页面就不在
存储器中（缺页）
有效位 = 1
m-1
p
p-1
0
物理页号 (PPN)
物理页偏移量 (PPO)
物理地址

地址翻译：页面命中



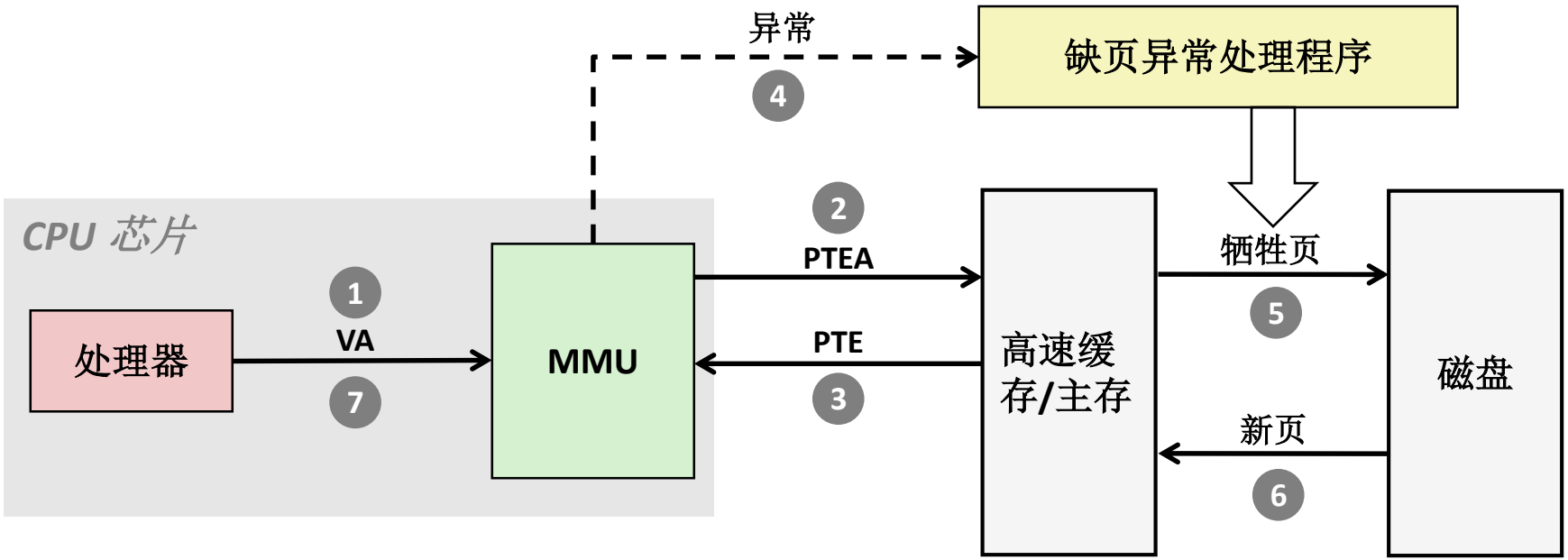
1) 处理器生成一个虚拟地址，并将其传送给MMU

2-3) MMU 使用内存中的页表生成PTE地址，高速缓存/主存向MMU返回PTE

4) MMU 将物理地址传送给高速缓存/主存

5) 高速缓存/主存返回所请求的数据字给处理器

地址翻译：缺页异常



1) 处理器将虚拟地址发送给 MMU

2-3) MMU 使用内存中的页表生成PTE地址

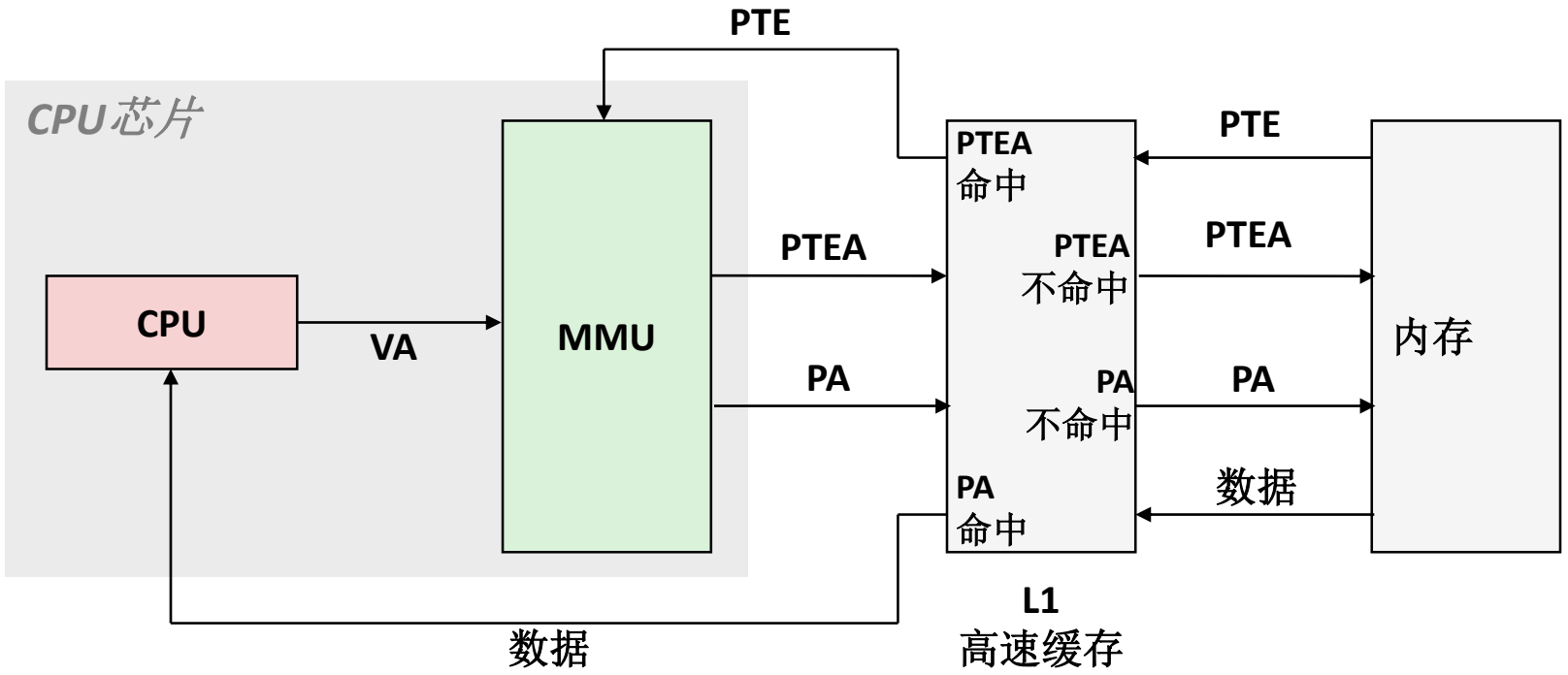
4) 有效位为零, 因此 MMU 触发缺页异常

5) 缺页处理程序确定物理内存中牺牲页 (若页面被修改，则换出到磁盘)

6) 缺页处理程序调入新的页面，并更新内存中的PTE

7) 缺页处理程序返回到原来进程，再次执行导致缺页的指令

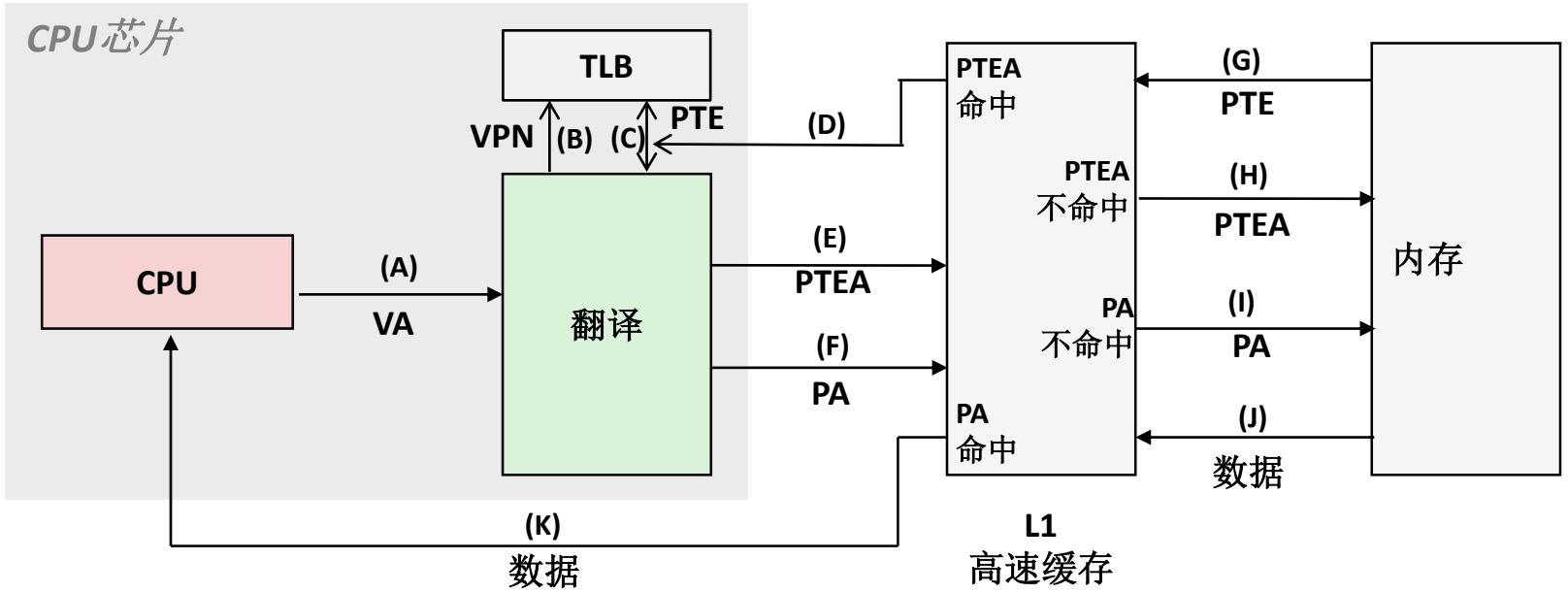
Integrating VM and Cache
结合高速缓存和虚拟内存



VA: virtual address虚拟地址, PA: physical address物理地址,

PTE:page table entry页表条目, PTEA = PTE address页表条目地址

Integrating VM and Cache
结合高速缓存和虚拟内存



***VA**: virtual address虚拟地址, **PA**: physical address物理地址,*

***PTE**:page table entry页表条目, **PTEA** = PTE address页表条目地址*

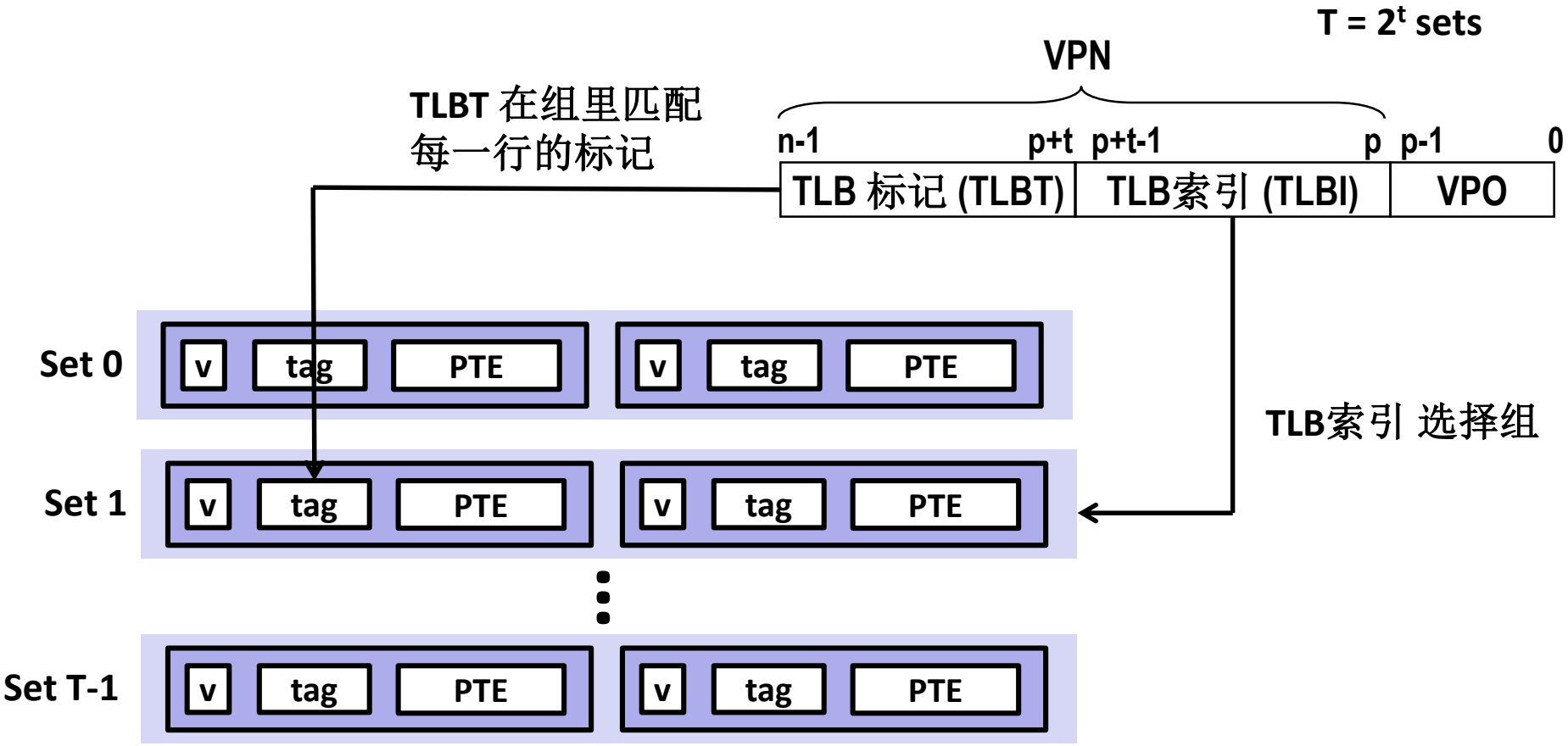
利用TLB加速地址翻译

每个虚拟地址都需要查PTE！

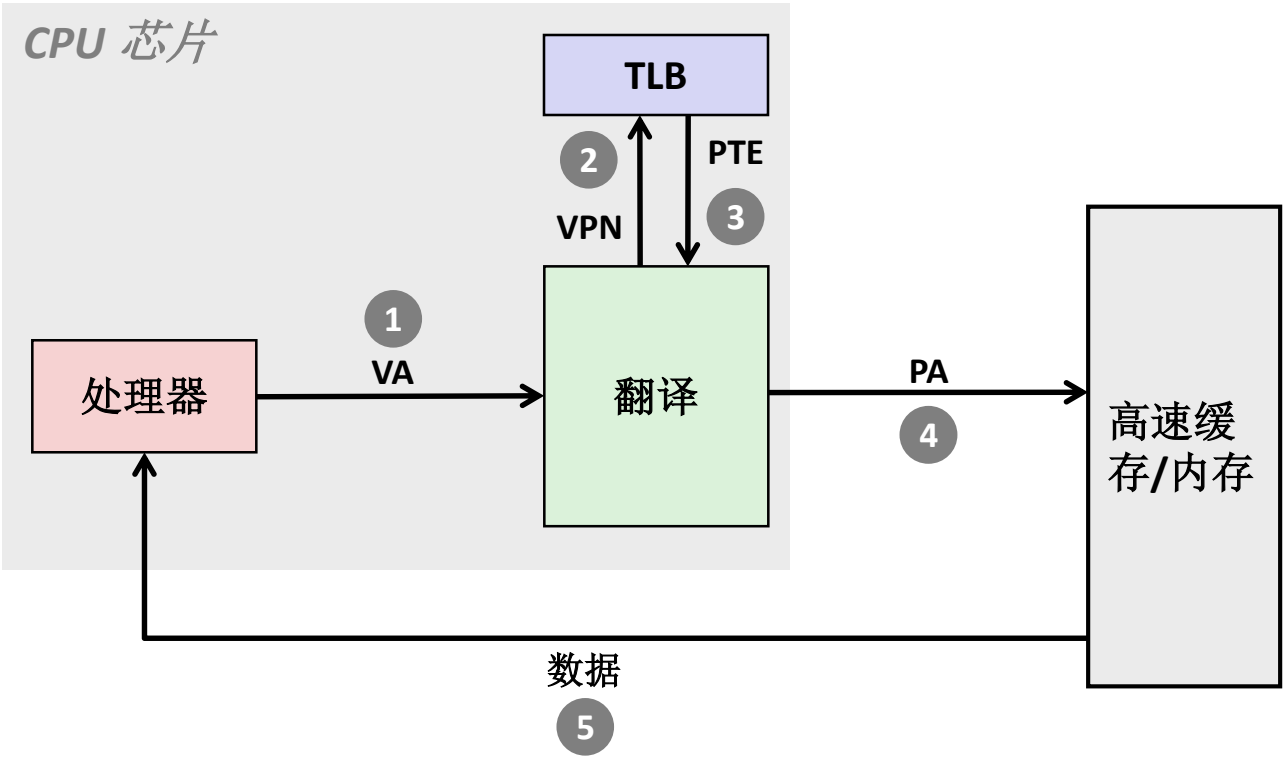
- **页表条目 (PTEs) 恰巧缓存在 L1**
  - PTE 可能被其他数据引用所驱逐
  - PTE 命中仍然需要1-2周期的延迟
- **解决办法: *Translation Lookaside Buffer* (TLB)翻译后备缓冲器——页表的缓存**
  - MMU中一个小的具有较高相联度的缓存
  - 实现虚拟页面向物理页面的映射
  - 对于页面数很少的页表可以完全包含在TLB中

访问TLB

- MMU 使用虚拟地址的 VPN 部分来访问TLB:

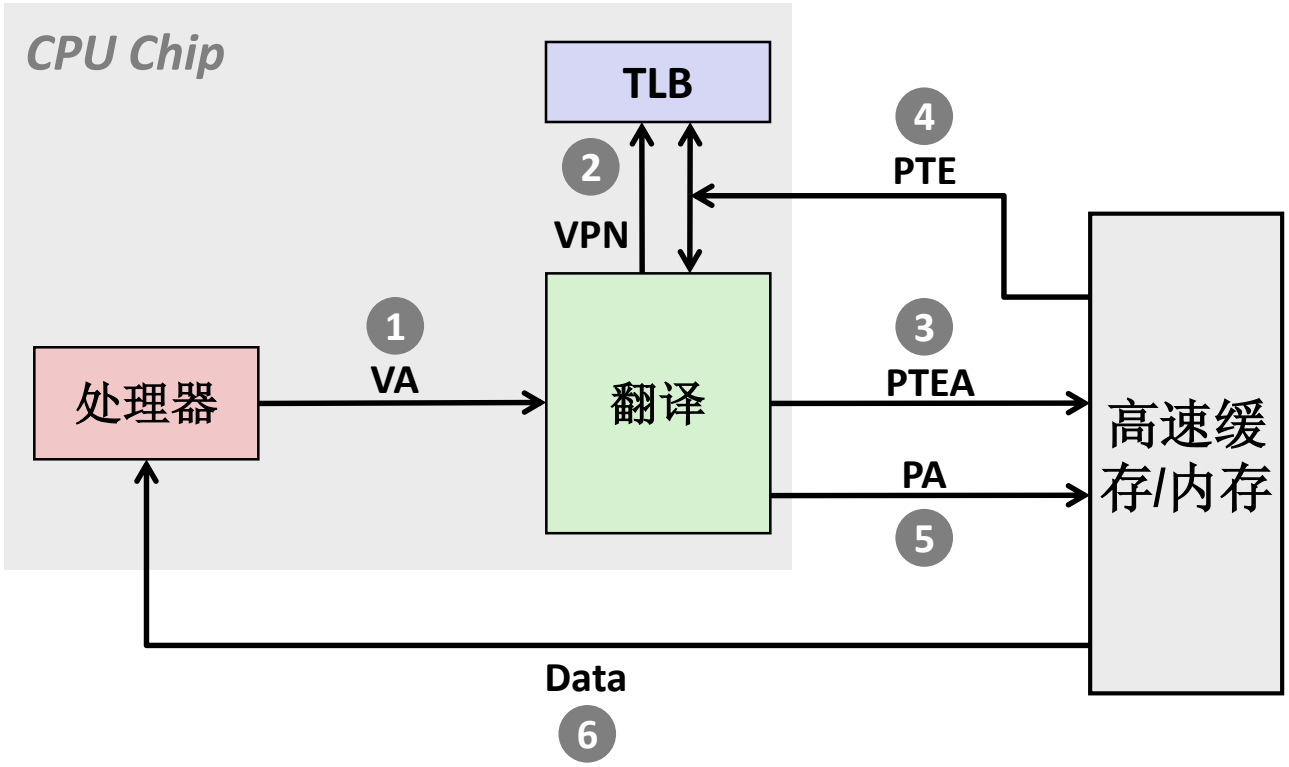


TLB Hit TLB命中



TLB 命中减少内存访问

TLB 不命中



TLB 不命中引发了额外的内存访问

万幸的是, TLB 不命中很少发生。

TLBs --- Making Address Translation Fast

问题：一次存储器引用要访问几次主存？ 0 / 1 / 2 / 3次？

把经常要查的页表项放到Cache中，这种在Cache中的页表项组成的页表称为 *Translation Lookaside Buffer* or *TLB（快表）*

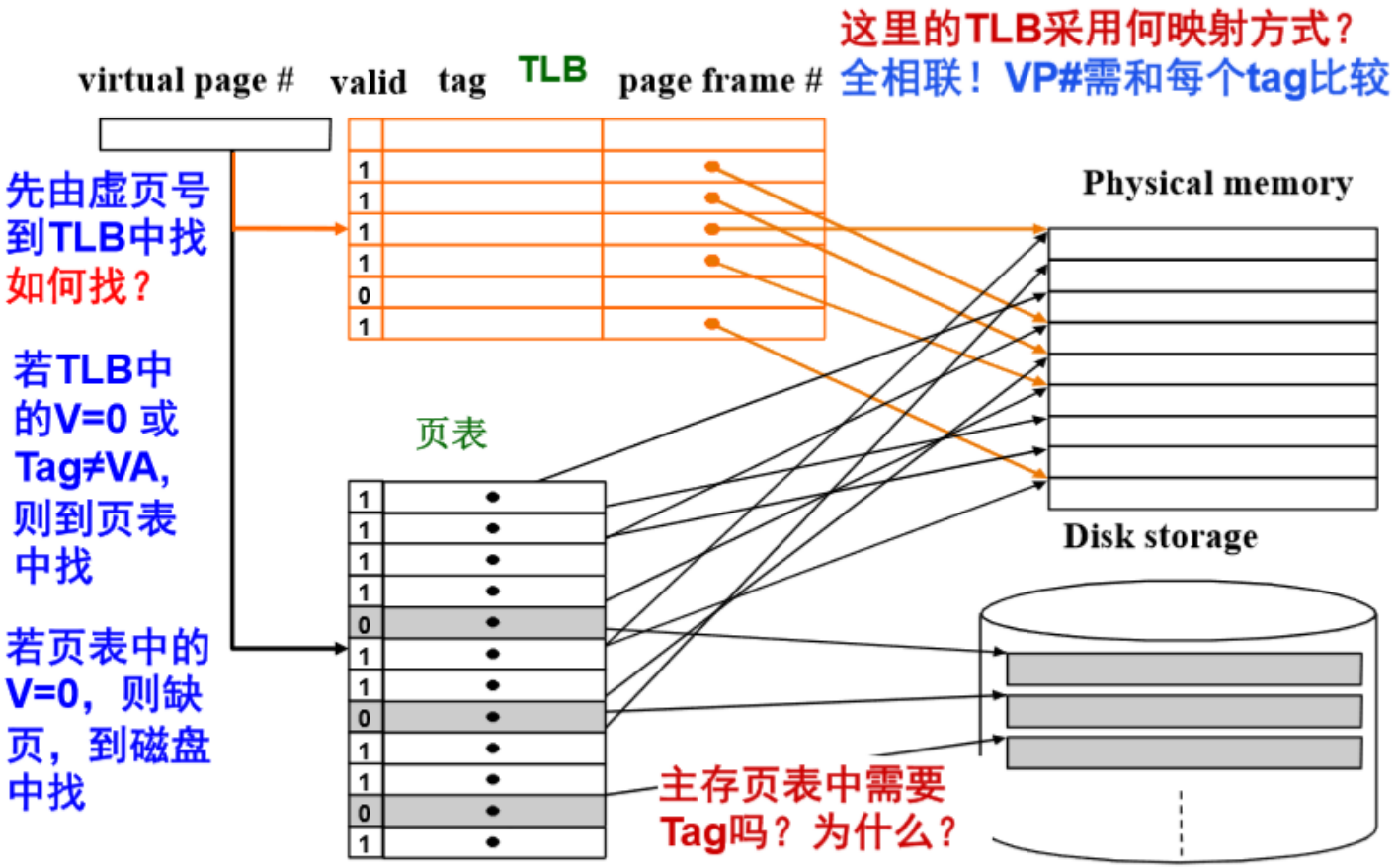
TLB中的页表项：tag+主存页表项

Virtual Address (tag)	Physical Address	Dirty	Ref	Valid	Access
	对应物理页框号				

CPU访存时，地址中虚页号被分成tag+index，tag用于和TLB页表项中的tag比较，index用于定位需要比较的表项

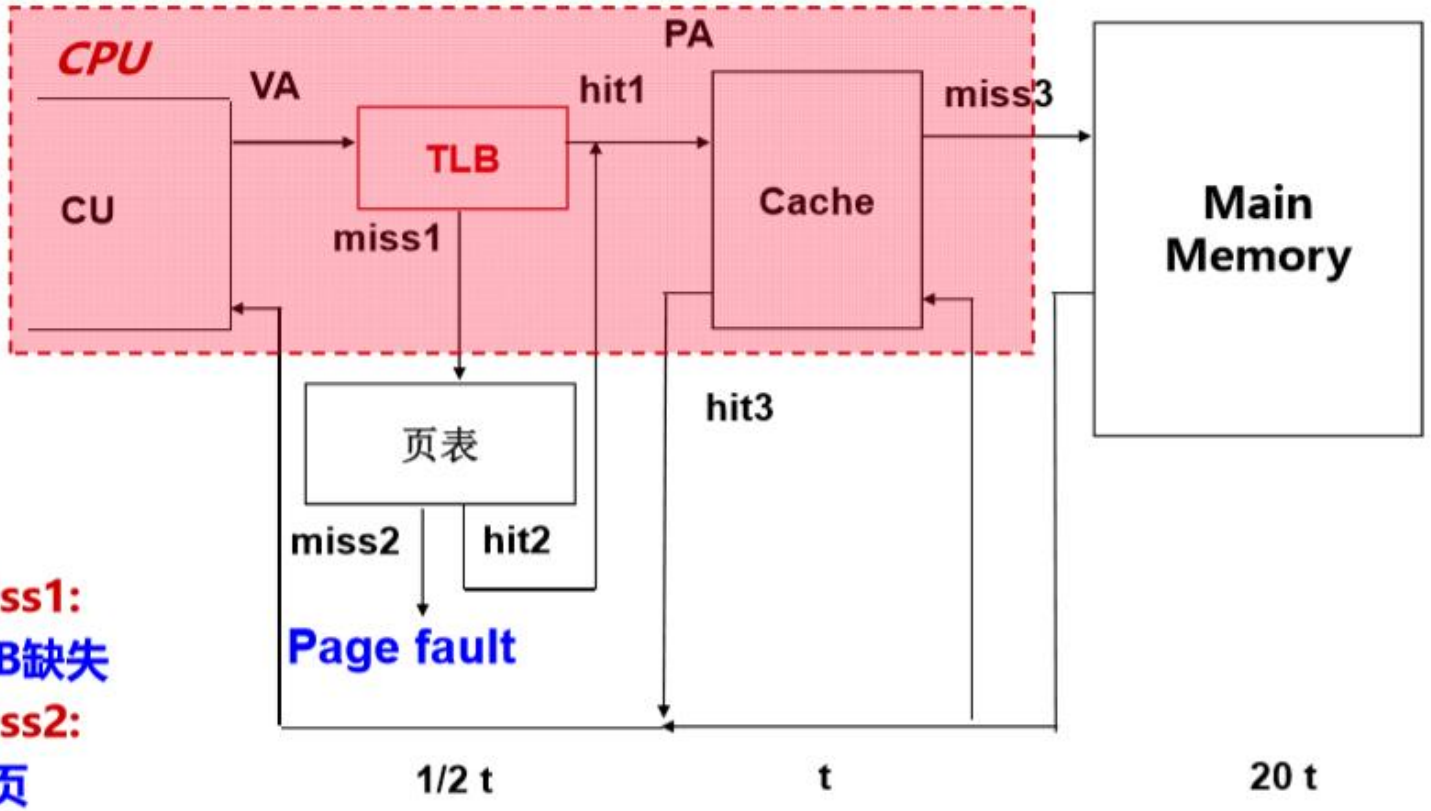
TLB全相联时，没有index，只有Tag，虚页号需与每个Tag比较；
TLB组相联时，则虚页号高位为Tag，低位为index，用作组索引。

TLBs --- Making Address Translation Fast



问题：引入TLB的目的是什么？ 减少到内存查页表的次数！

Translation Look-Aside Buffers



Miss1:
TLB缺失
Miss2:
缺页
Miss3:
PA 在主存中，但不在Cache中

TLB冲刷指令和Cache冲刷指令都是操作系统使用的特权指令

举例：三种不同缺失的组合

TLB	Page table	Cache	Possible? If so, under what circumstance?
hit	hit	miss	可能，TLB命中则页表一定命中，但实际上不会查页表
miss	hit	hit	可能，TLB缺失但页表命中，信息在主存，就可能在Cache
miss	hit	miss	可能，TLB缺失但页表命中，信息在主存，但可能不在Cache
miss	miss	miss	可能，TLB缺失页表缺失，信息不在主存，一定也不在Cache
hit	miss	miss	不可能，页表缺失，信息不在主存，TLB中一定没有该页表项
hit	miss	hit	同上
miss	miss	hit	不可能，页表缺失，信息不在主存，Cache中一定也无该信息

最好的情况是hit、hit、hit，此时，访问主存几次？ 不需要访问主存！

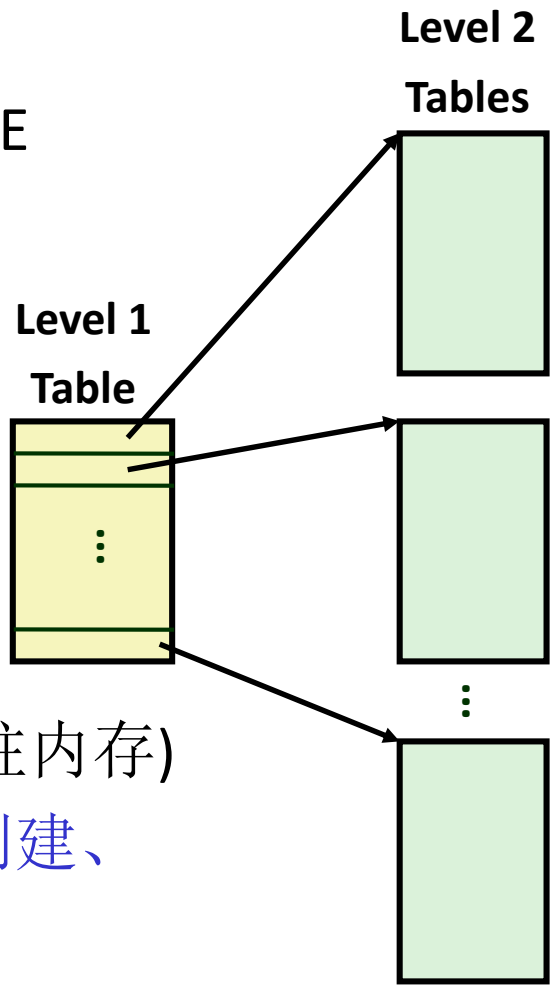
以上组合中，最好的情况是？ hit、hit、miss和miss、hit、hit 访存1次

以上组合中，最坏的情况是？ miss、miss、miss 需访问磁盘、并访存至少2次

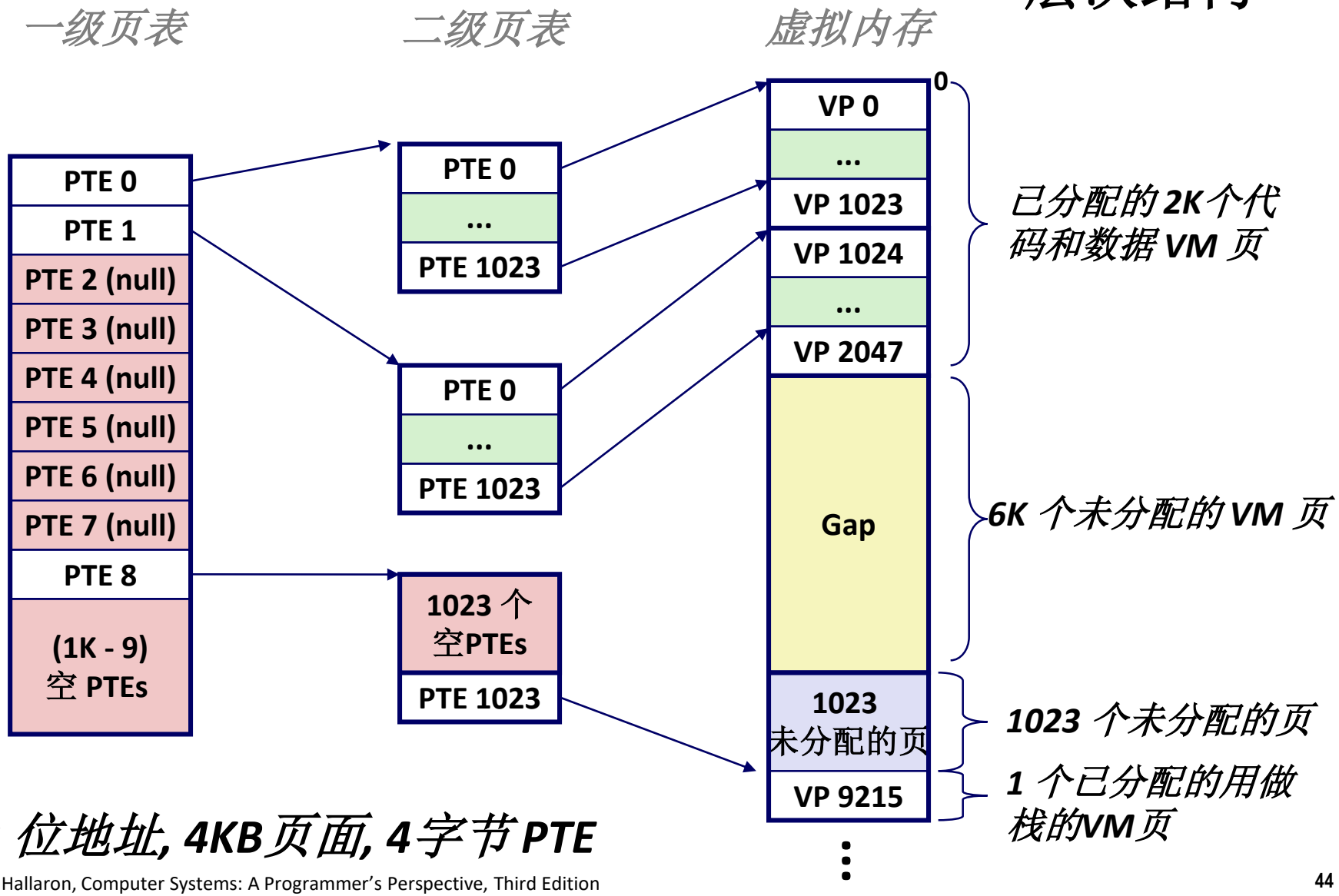
介于最坏和最好之间的是？ miss、hit、miss 不需访问磁盘、但访存至少2次

Multi-Level Page Tables 多级页表

- 假设:
  - 4KB (2^{12}) 页面, 48位地址空间, 8字节 PTE
- 问题:
  - 将需要一个大小为 512 GB 的页表!
    - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes
- 常用解决办法: 多级页表
- 以二级页表为例:
  - 一级页表: 每个 PTE 指向一个页表 (常驻内存)
  - 二级页表: 每个 PTE 指向一页(需要时创建、调入或调出二级页表)



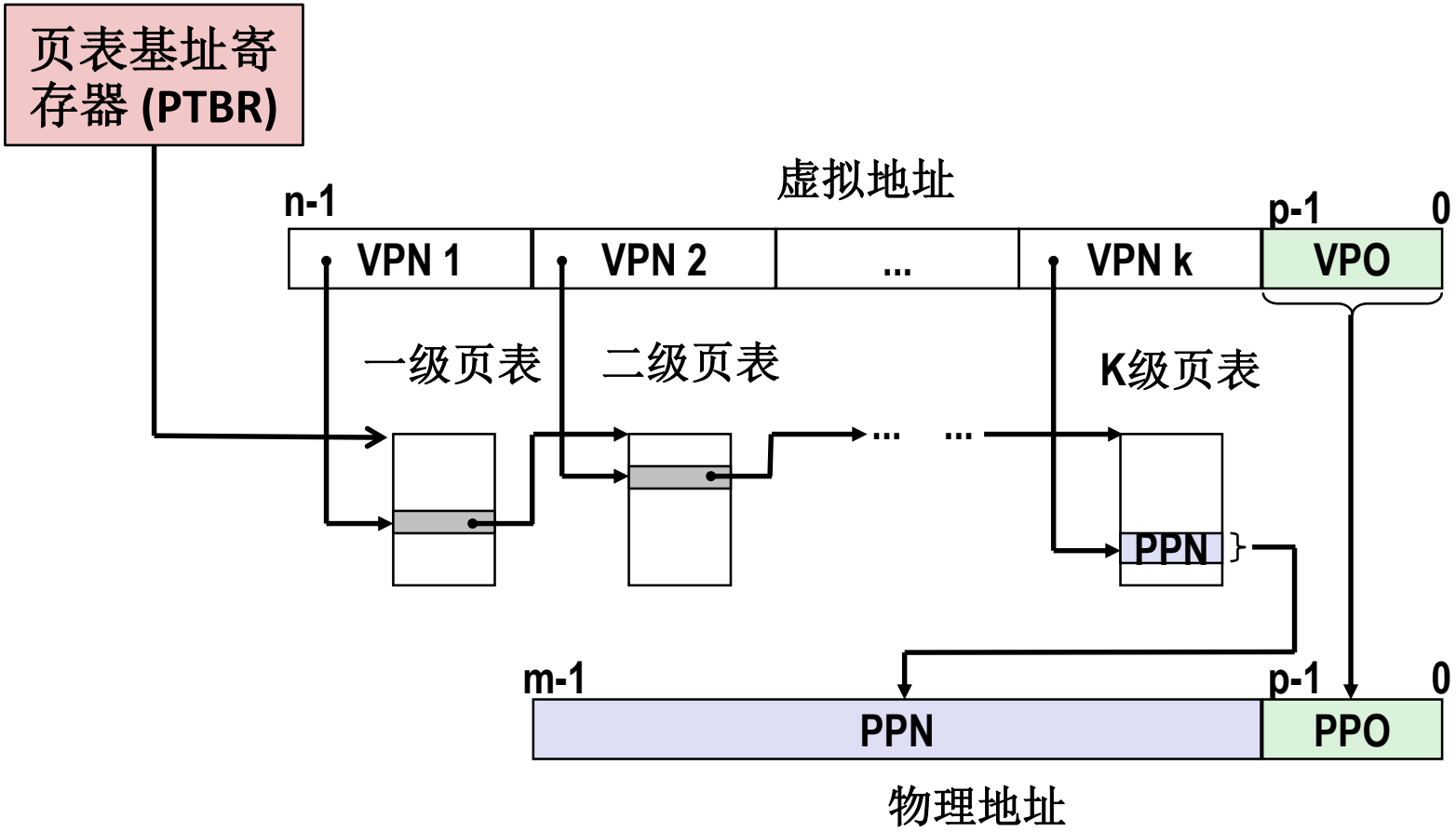
A Two-Level Page Table Hierarchy 二级页表的层次结构



32 位地址, 4KB页面, 4字节 PTE

Translating with a k-level Page Table

使用K级页表的地址翻译



总结

- **程序员的角度看待虚拟内存**
  - 每个进程拥有自己私有的线性地址空间
  - 不允许被其他进程干扰
- **系统的角度看待虚拟内存**
  - 通过获取虚拟内存页面来有效使用内存
    - 有效只因为“局部性”的原因
  - 简化编程和内存管理
  - 提供方便的标志位来检查权限以简化内存保护