

第9章 虚拟内存: 基本概念

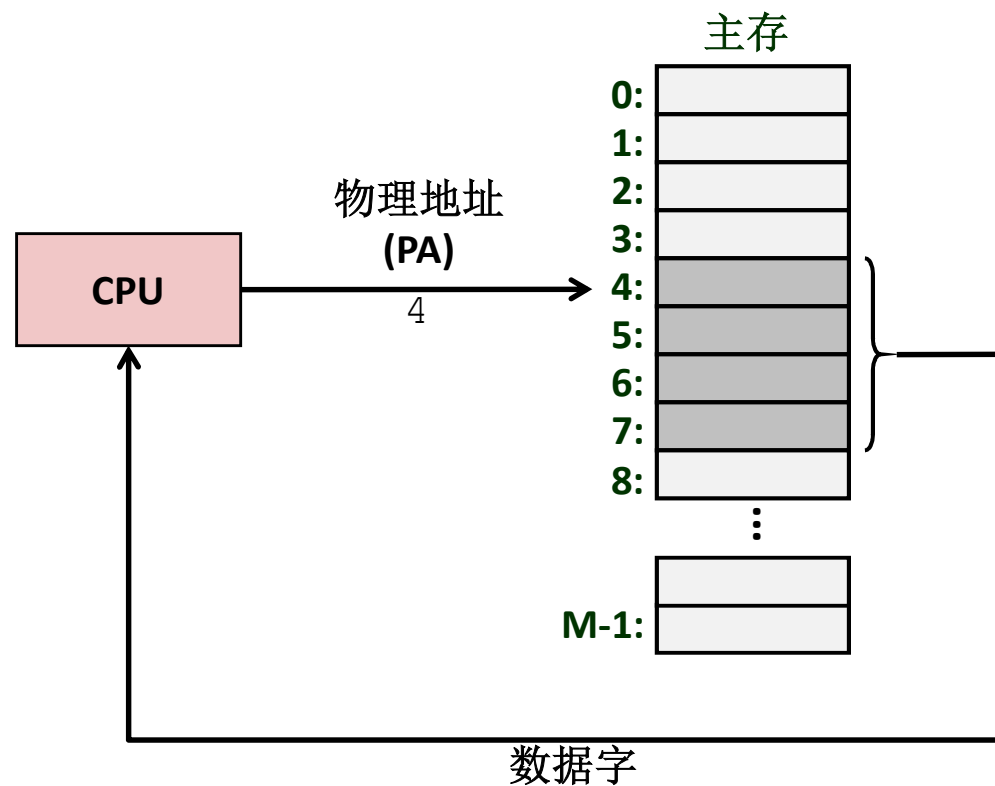
教 师: 史先俊
计算机科学与技术学院
哈尔滨工业大学

主要内容

- **Address spaces** 地址空间
- **VM as a tool for caching** 虚拟内存作为缓存的工具
- **VM as a tool for memory management**
虚拟内存作为内存管理的工具
- **VM as a tool for memory protection**
虚拟内存作为内存保护的工具有
- **Address translation**地址翻译

A System Using Physical Addressing

一个使用物理寻址的系统



- 诸如汽车、电梯、数字图像帧（digital picture frame）等“简单”系统中作为嵌入式微控制器使用

Address Spaces 地址空间

■ 逻辑地址空间：段地址： 偏移地址

23: 8048000 段寄存器（CS等16位）： 偏移地址（16/32/64）

实模式下： 逻辑地址CS: EA \Rightarrow 物理地址CS*16+EA

保护模式下： 以段描述符作为下标，到GDT/LDT表查表获得段地址，
段地址+偏移地址=线性地址。

■ 线性地址空间: 非负整数地址的有序集合:

$\{0, 1, 2, 3 \dots\}$

■ 虚拟地址空间: $N = 2^n$ 个虚拟地址的集合 === 线性地址空间

$\{0, 1, 2, 3, \dots, N-1\}$

■ 物理地址空间: $M = 2^m$ 个物理地址的集合

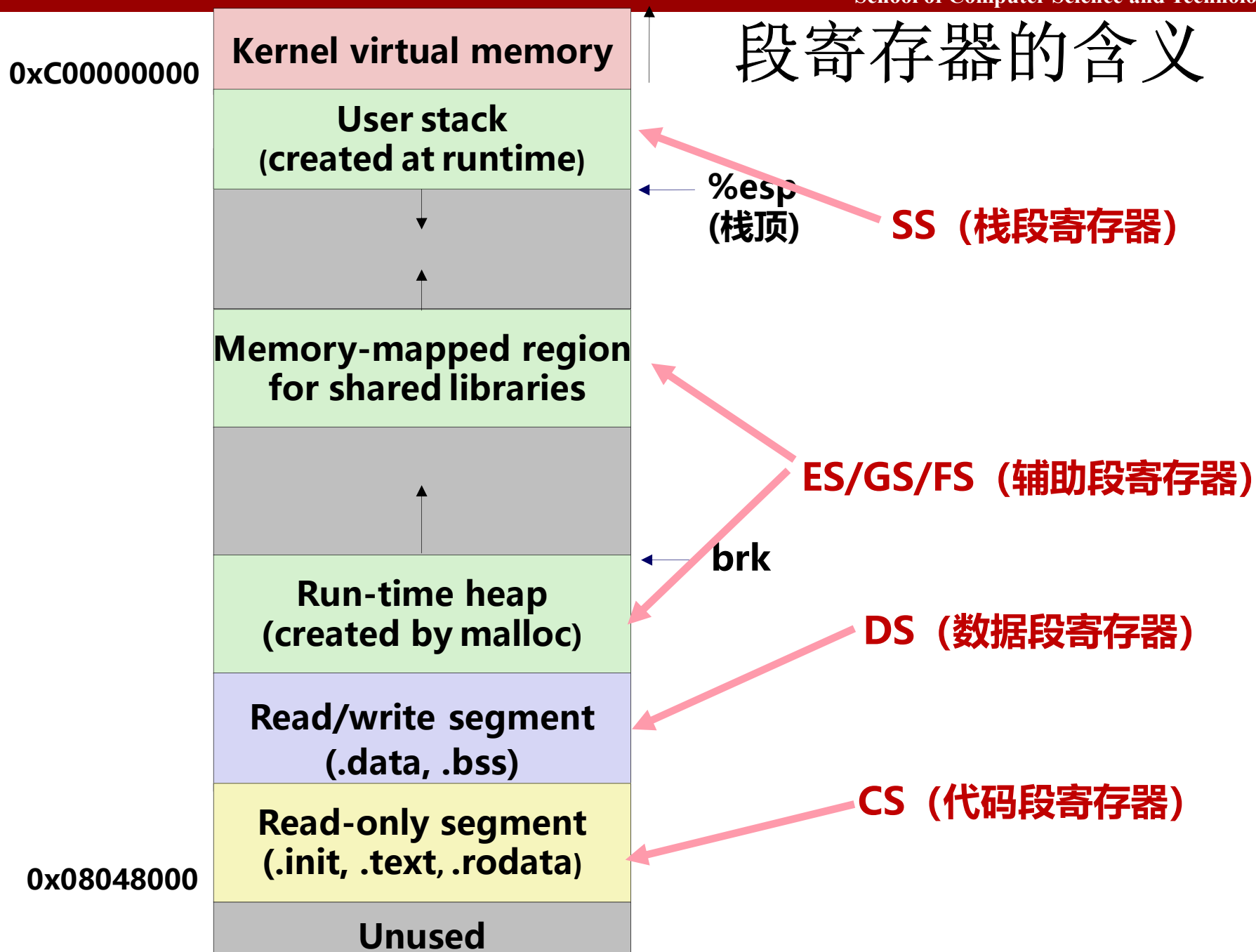
$\{0, 1, 2, 3, \dots, M-1\}$

■ Intel采用段页式存储管理（MMU实现）

■ 段式管理： 逻辑地址 \rightarrow 线性地址==虚拟地址

■ 页式管理： 虚拟地址 \rightarrow 物理地址

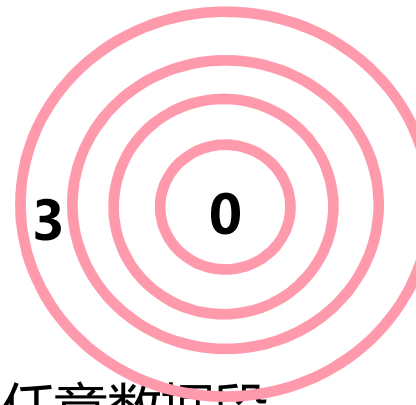
段寄存器的含义



段选择符和段寄存器

■ 段寄存器 (16位) , 用于存放段选择符

- CS(代码段): 程序代码所在段
- SS(栈段): 栈区所在段
- DS(数据段): 全局静态数据区所在段
- 其他3个段寄存器ES、GS和FS可指向任意数据段



环保护: 内核工作在0环, 用户工作在3环, 中间环留给中间软件用。
Linux仅用第0和第3环。

°段选择符各字段含义:

15	14	3	2	1	0
索引				TI	RPL	

CS寄存器中的RPL字段表示CPU的**当前特权级** (Current Privilege Level, **CPL**)

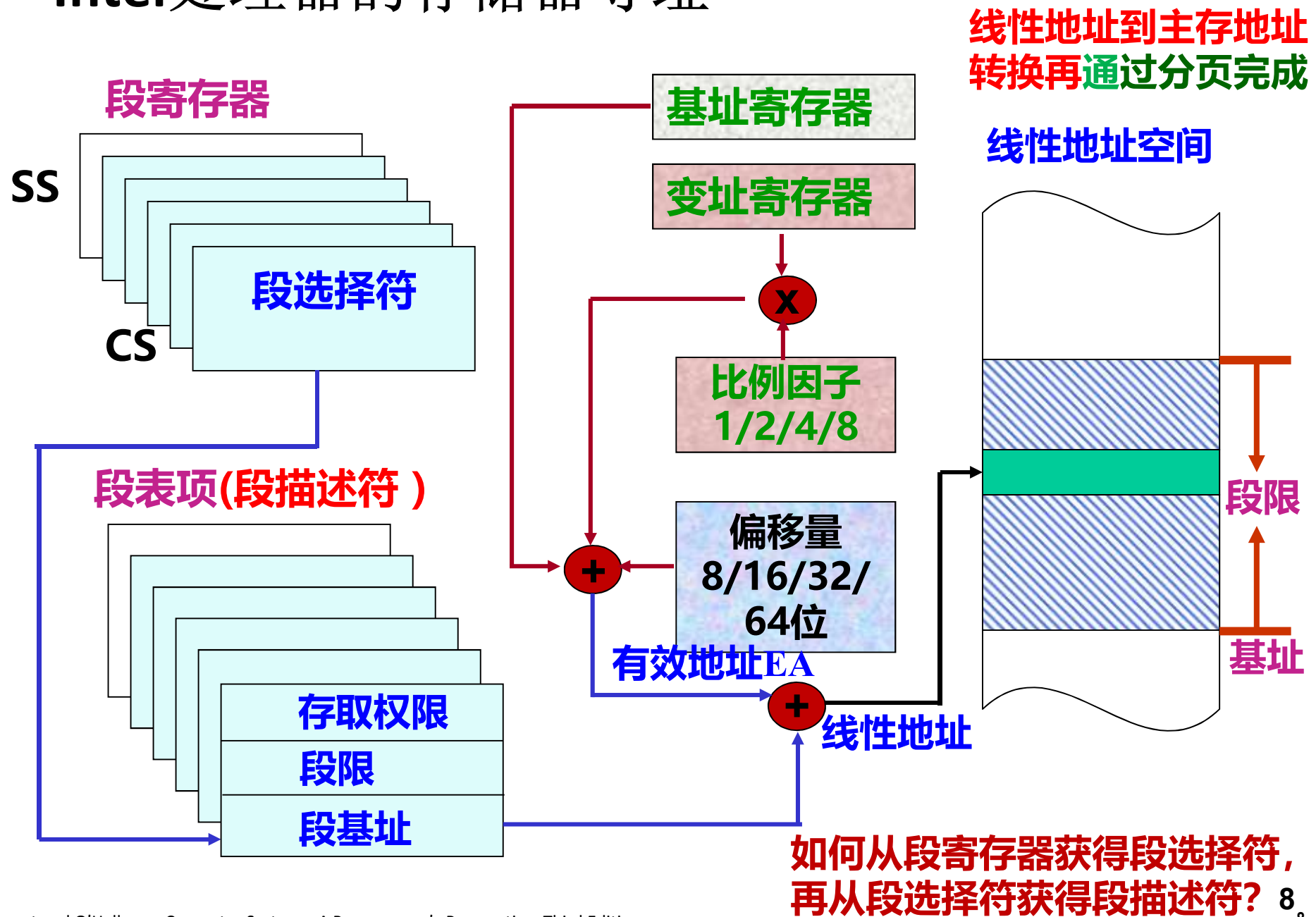
- TI=0, 选择**全局描述符表(GDT)**, TI=1, 选择**局部描述符表(LDT)**
- RPL=00, 为第0级, 位于最高级的内核态, RPL=11, 为第3级, 位于最低级的用户态, **第0级高于第3级**
- 高13位-8K个索引用来确定当前使用的**段描述符**在描述符表中的位置



段描述符和段描述符表

- **段描述符**是一种数据结构，实际上就是**段表项**，分两类：
 - 用户的代码段和数据段描述符
 - 系统控制段描述符，又分两种：
 - 特殊系统控制段描述符，包括：局部描述符表（LDT）描述符和任务状态段（TSS）描述符
 - 控制转移类描述符，包括：调用门描述符、任务门描述符、中断门描述符和陷阱门描述符
- **描述符表**实际上就是**段表**，由段描述符**(段表项)**组成。有三种类型
 - 全局描述符表GDT：只有一个，用来存放系统内每个任务都可能访问的描述符，例如，内核代码段、内核数据段、用户代码段、用户数据段以及TSS（任务状态段）等都属于GDT中描述的段
 - 局部描述符表LDT：存放某任务（即用户进程）专用的描述符
 - 中断描述符表IDT：包含256个中断门、陷阱门和任务门描述符

Intel处理器的存储器寻址



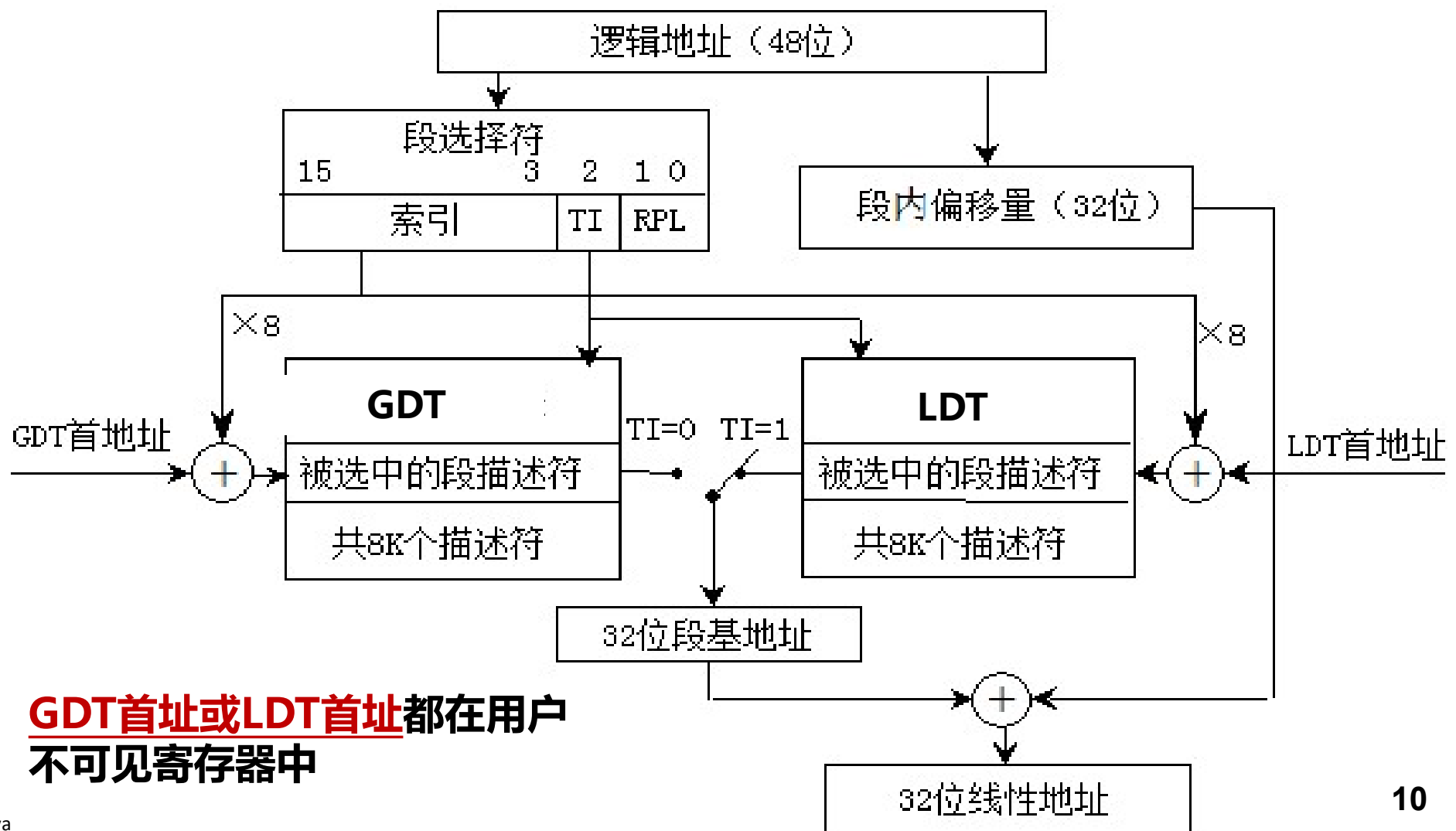
段描述符的定义 (IA32)

	15					8	7	6	5	4	3	2	1	0	
7	段基址 (B31-B24)						G	D	0	AVL	限界 (L19-L16)				6
5	P	DPL	S	TYPE		A	段基址 (B23-B16)								4
3	当CPL>DPL时, 说明当前特权级比所要求的最低等级更低, 故访问越级						段基址 (B15-B0)				因为CPL和DPL的数值越大, 等级越低				2
1							限界 (L15-L0)								0

- B31~B0: 32位基地址; L19~L0: 20位限界, 表示段中最大页号
- G: 粒度。G=1以页 (4KB) 为单位; G=0以字节为单位。因为界限为20位, 故当G=0时最大的段为1MB; 当G=1时, 最大段为 $4KB \times 2^{20} = 4GB$
- D: D=1表示段内偏移量为32位宽, D=0表示段内偏移量为16位宽
- P: P=1表示存在, P=0表示不存在。Linux总把P置1, 不会以段为单位淘汰
- DPL: 访问段时对当前特权级的最低等级要求。因此, 只有CPL为0 (内核态) 时才可访问DPL为0的段, 任何进程都可访问DPL为3的段 (**0最高、3最低**)
- S: S=0系统控制描述符, S=1普通的代码段或数据段描述符
- TYPE: 段的访问权限或系统控制描述符类型
- A: A=1已被访问过, A=0未被访问过。 (通常A包含在TYPE字段中)

逻辑地址向线性地址转换

- 被选中的段描述符先被送至描述符cache，每次从描述符cache中取32位段基址，与32位段内偏移量（有效地址）相加得到线性地址



IA-32/Linux中的分段机制

- 为使能移植到绝大多数流行处理器平台，Linux简化了分段机制
- RISC对分段支持非常有限，因此Linux仅使用IA-32的分页机制，而对于分段，则通过在初始化时将所有段描述符的基址设为0来简化
- 若把运行在用户态的所有Linux进程使用的代码段和数据段分别称为**用户代码段**和**用户数据段**；把运行在内核态的所有Linux进程使用的代码段和数据段分别称为**内核代码段**和**内核数据段**，则Linux初始化时，将上述4个段的段描述符中各字段设置成下表中的信息：

段	基地址	G	限界	S	TYPE	DPL	D	P
用户代码段	0x0000 0000	1	0xFFFFF	1	10	3	1	1
用户数据段	0x0000 0000	1	0xFFFFF	1	2	3	1	1
内核代码段	0x0000 0000	1	0xFFFFF	1	10	0	1	1
内核数据段	0x0000 0000	1	0xFFFFF	1	2	0	1	1

每个段都被初始化在
0~4GB的线性地址空间中

初始化时，上述4个**段描述符**被存放在GDT中

Linux的全局描述符表（GDT）

[BACK](#)

Linux 全局描述符表 段选择符

0000000001100000

说明内核代码段处于第0环，其描述符在GDT中，索引值为0x000C

0000000001101000

说明内核数据段处于第0环，其描述符在GDT中，索引值为0x000D

0000000001110011

说明用户代码段处于第3环，其描述符在GDT中，索引值为0x000E

0000000001111011

说明用户数据段处于第3环，其描述符在GDT中，索引值为0x000F

reserved

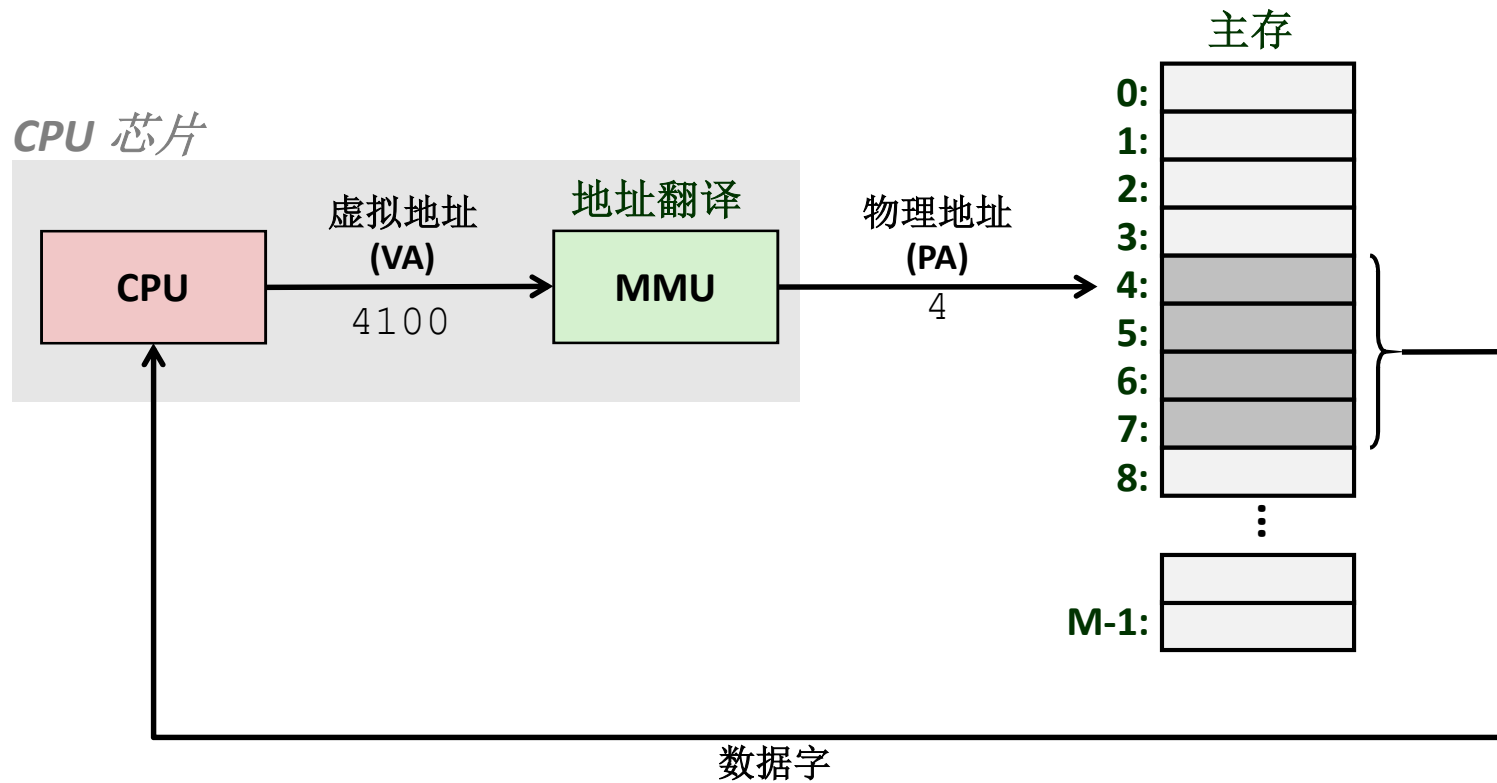
kernel code	0x60 (__KERNEL_CS)
kernel data	0x68 (__KERNEL_DS)
user code	0x73 (__USER_CS)
user data	0x7b (__USER_DS)

Linux 全局描述符表 段选择符

TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
not used	
not used	
not used	
not used	
double fault TSS	0xf8

A System Using Physical Addressing

一个使用物理寻址的系统



- 现在处理器使用, 比如笔记本、智能电话等
- 计算机科学的伟大思想之一

Why Virtual Memory (VM)?

为什么要使用虚拟内存?

- 有效使用主存
 - 使用DRAM作为部分虚拟地址空间的缓存
- 简化内存管理
 - 每个进程都使用统一的线性地址空间
- 独立地址空间
 - 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

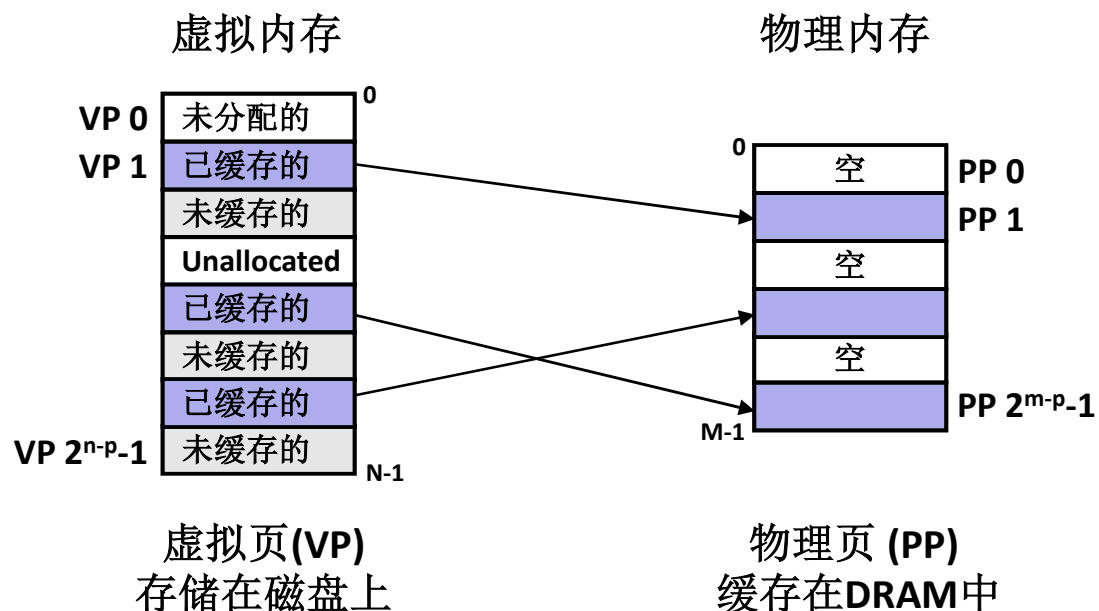
主要内容

- **Address spaces** 地址空间
- **VM as a tool for caching** 虚拟内存作为缓存的工具
- **VM as a tool for memory management**
虚拟内存作为内存管理的工具
- **VM as a tool for memory protection**
虚拟内存作为内存保护的工具有
- **Address translation**地址翻译

VM as a tool for caching

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** 个连续的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)



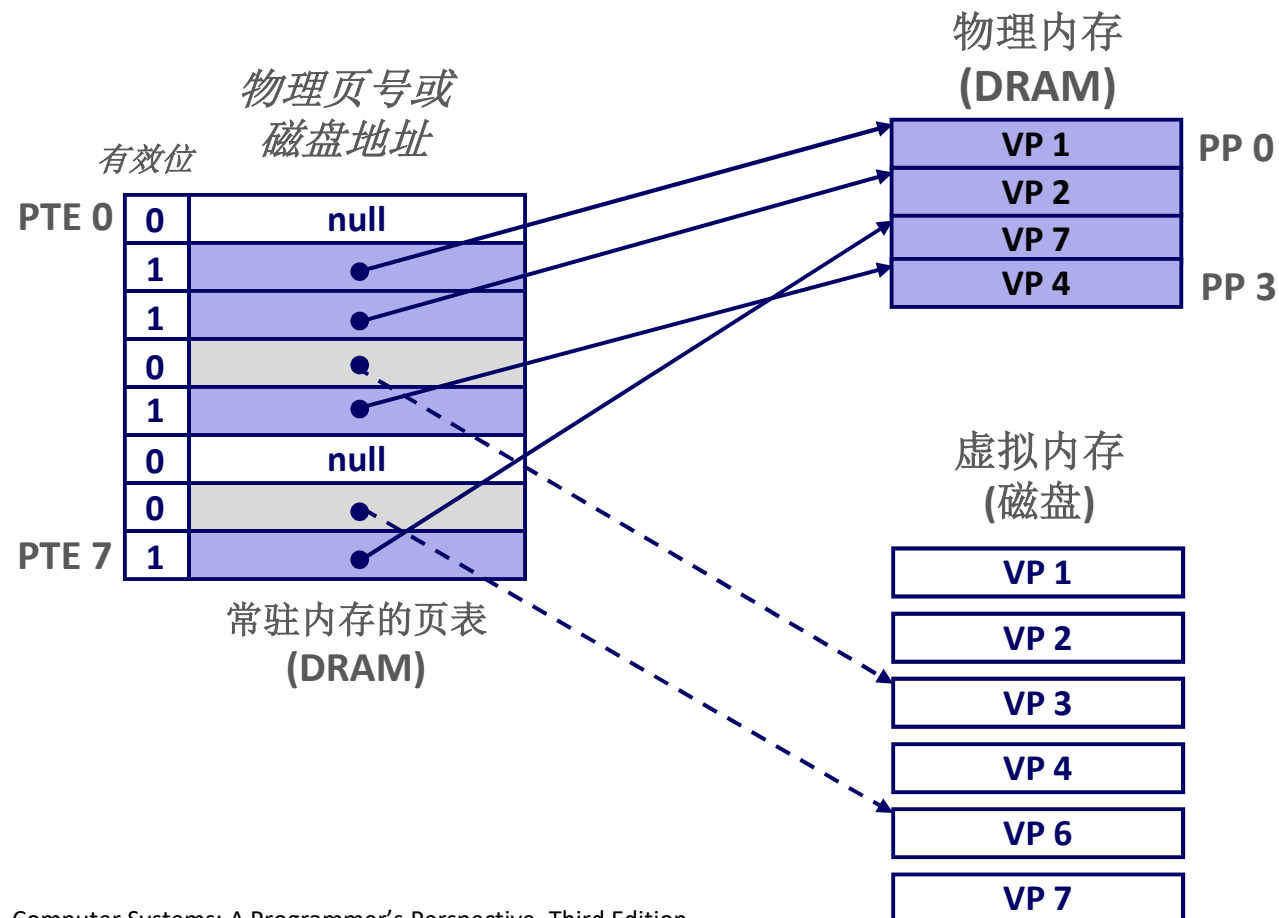
DRAM Cache Organization

DRAM缓存的组织结构

- **DRAM** 缓存的组织结构完全是由巨大的不命中开销驱动的
 - DRAM 比 SRAM 慢大约 **10倍**
 - 磁盘比 DRAM 慢大约 **10,000倍**
- 因此
 - 虚拟页尺寸: 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

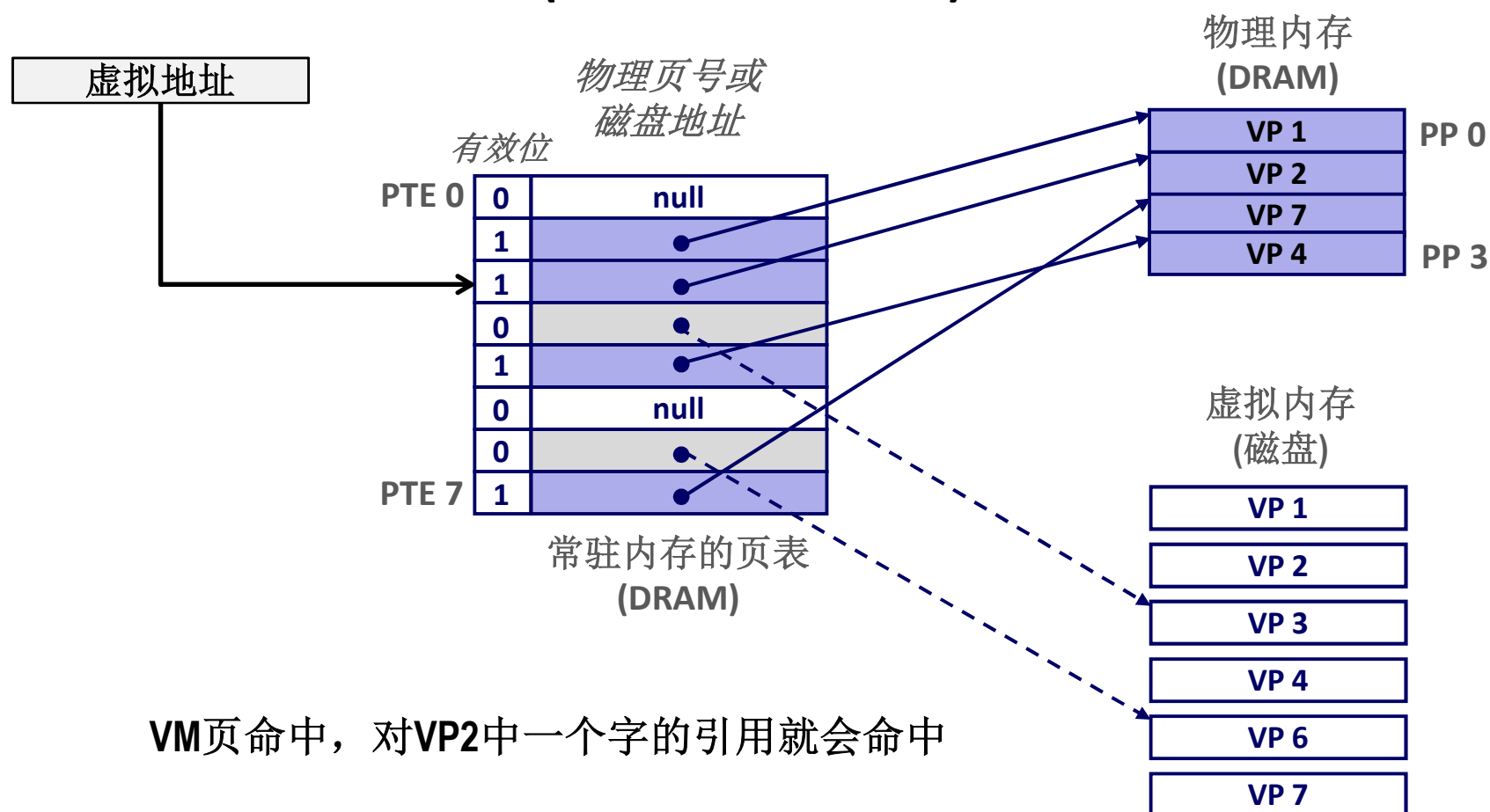
Enabling Data Structure: Page Table 页表

- **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。
 - DRAM 中的每个进程都使用的核心数据结构



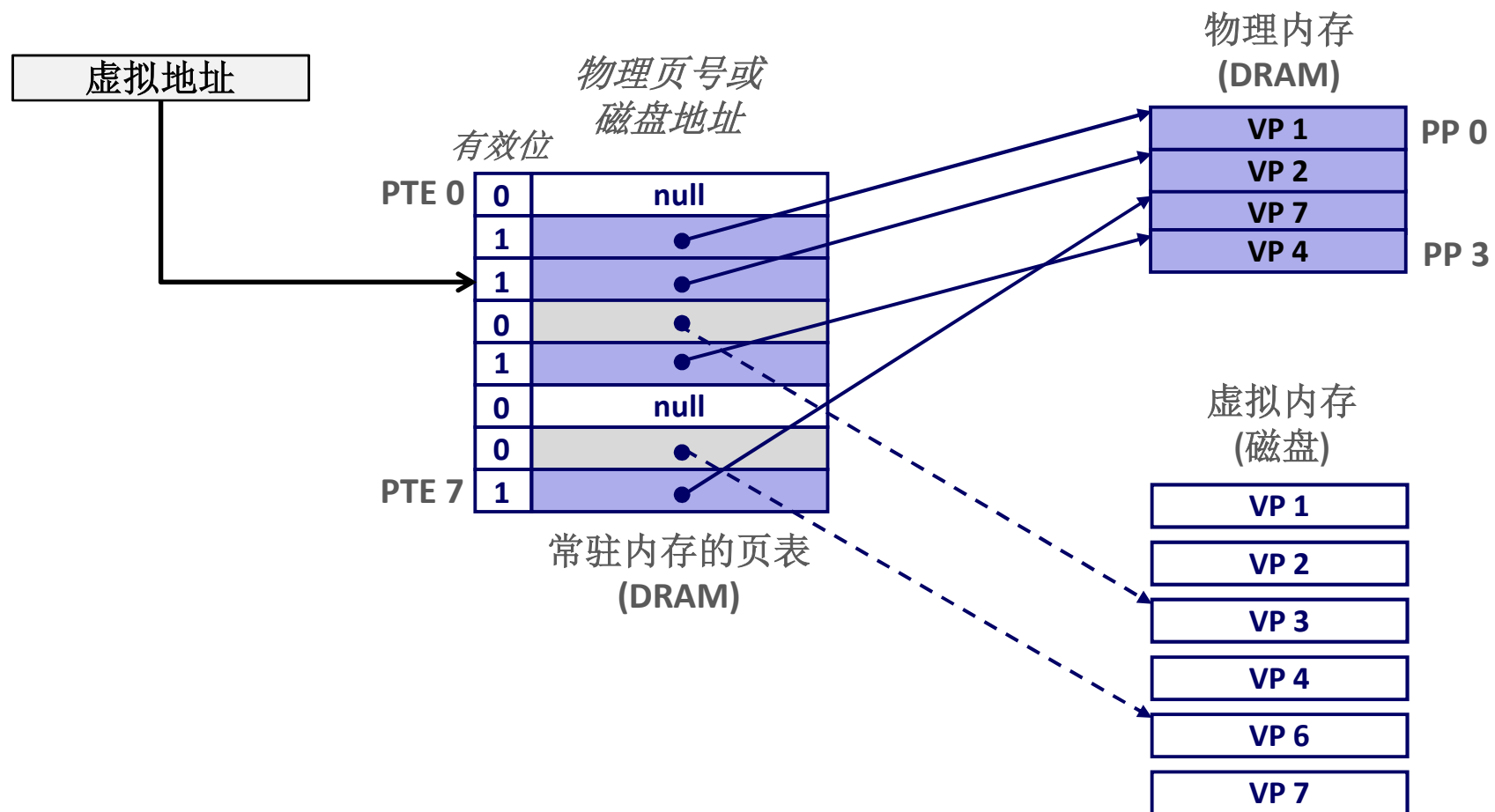
Page Hit 页命中

- **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)



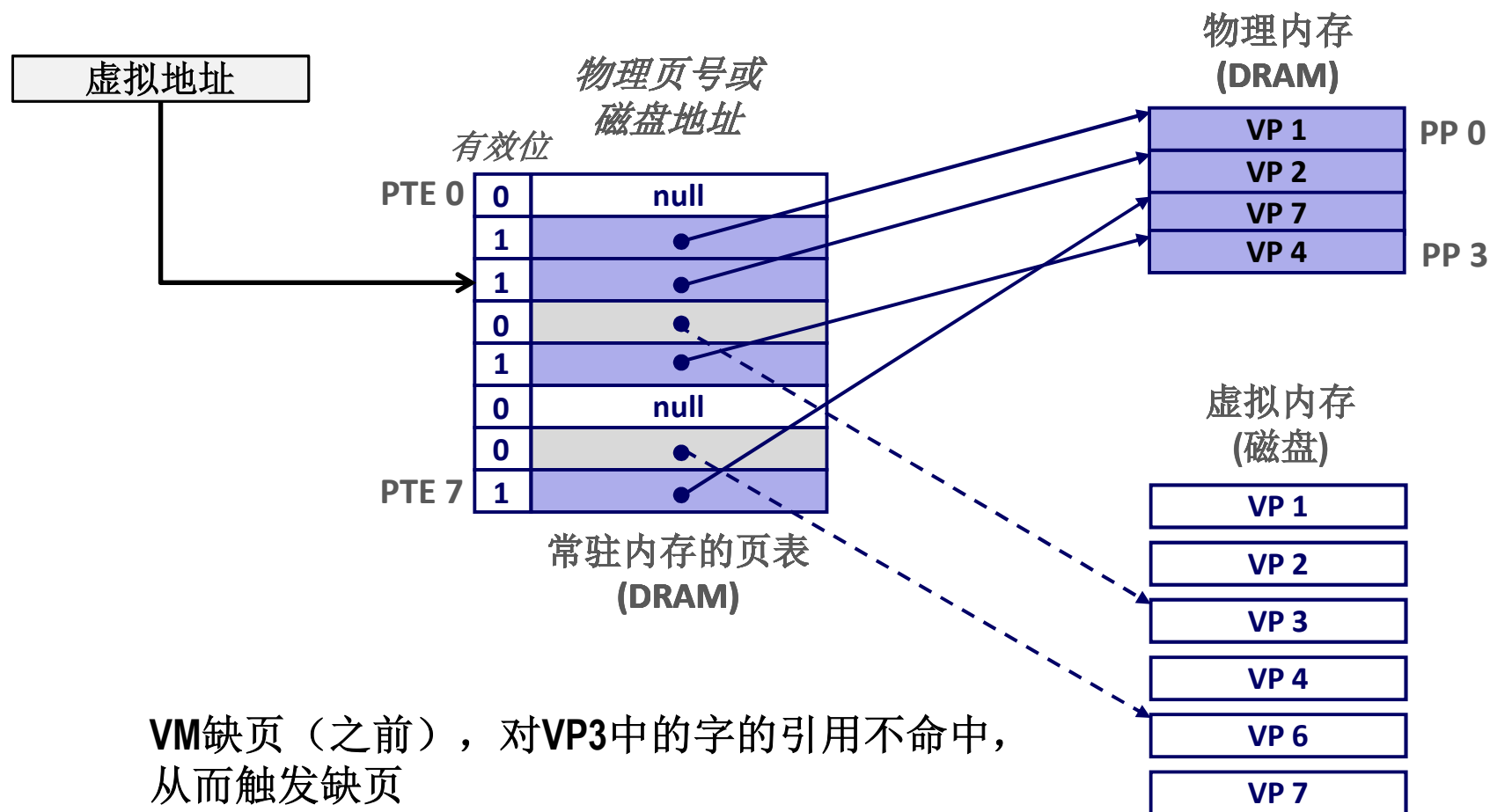
Page Fault 缺页

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



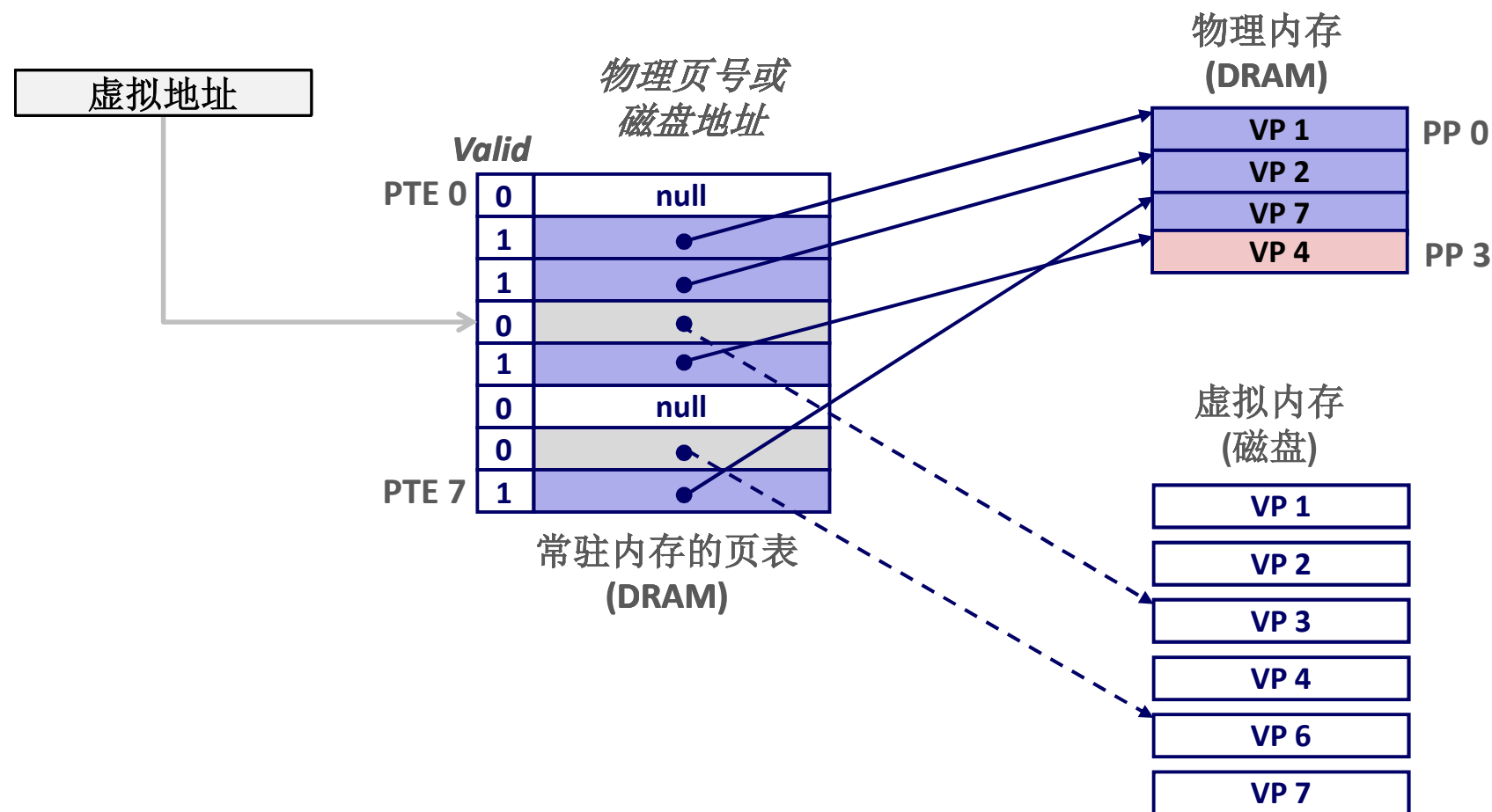
Handling Page Fault 缺页的处理

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



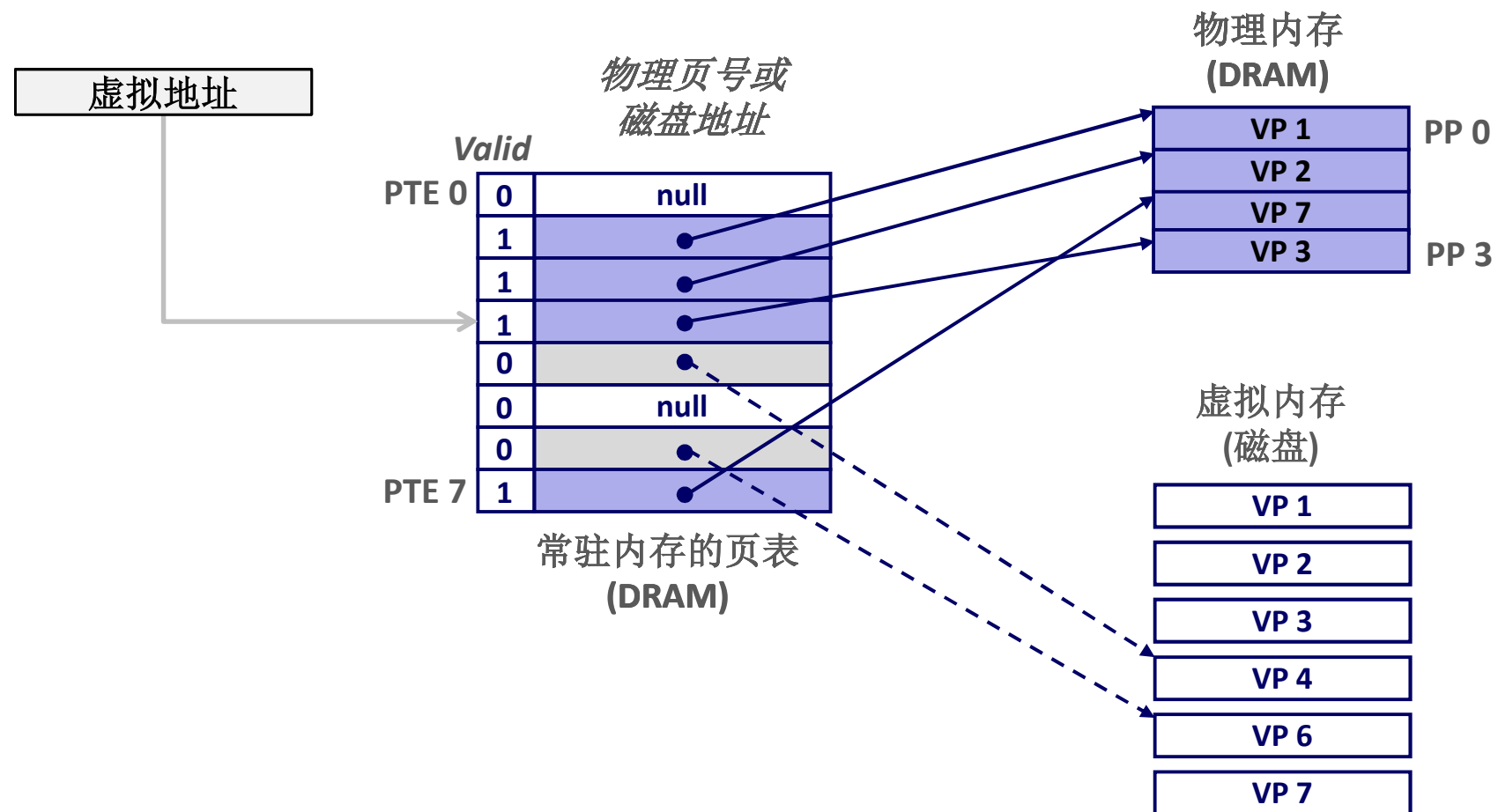
Handling Page Fault 缺页的处理

- 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)



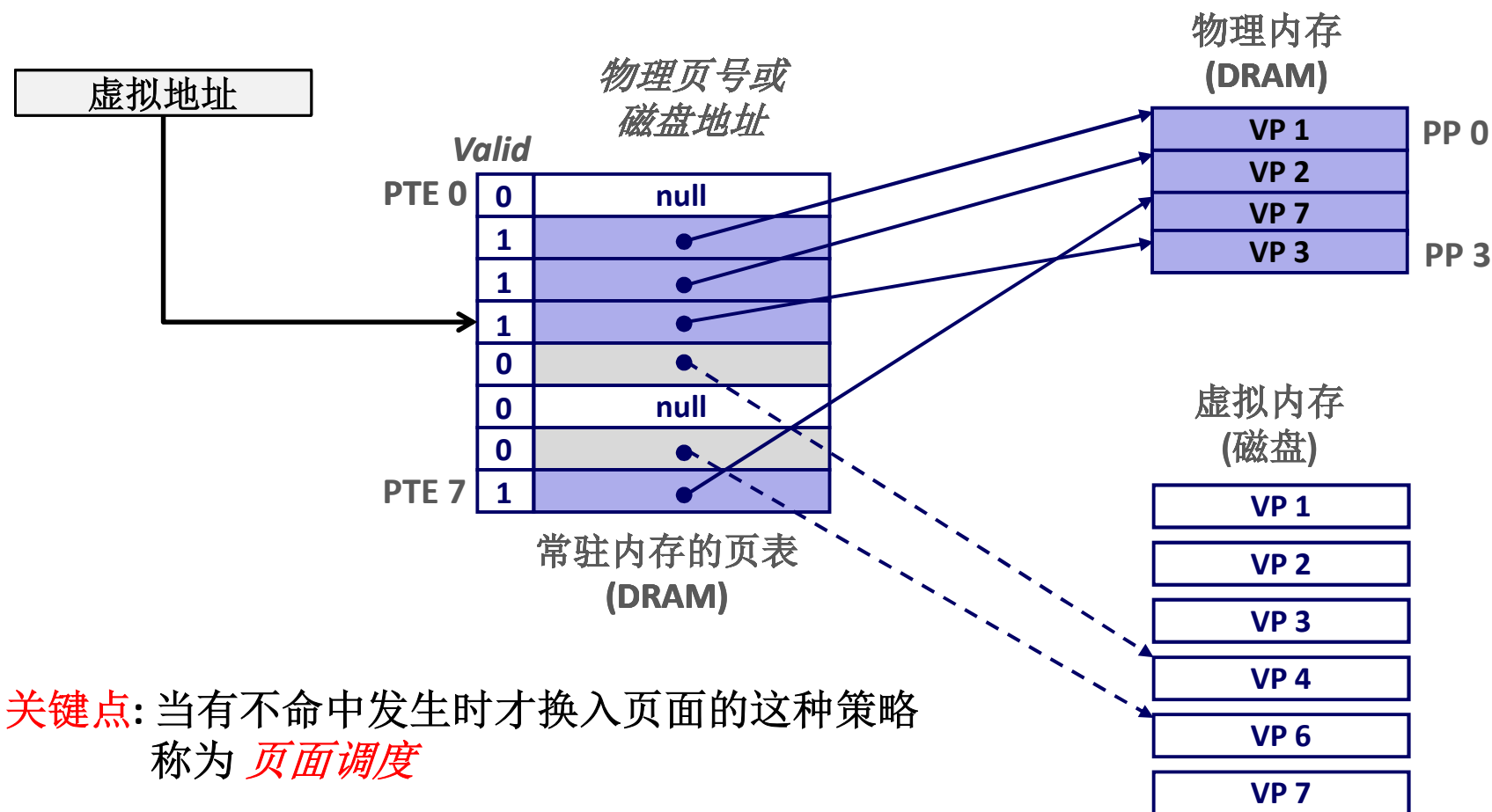
Handling Page Fault 缺页的处理

- 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)



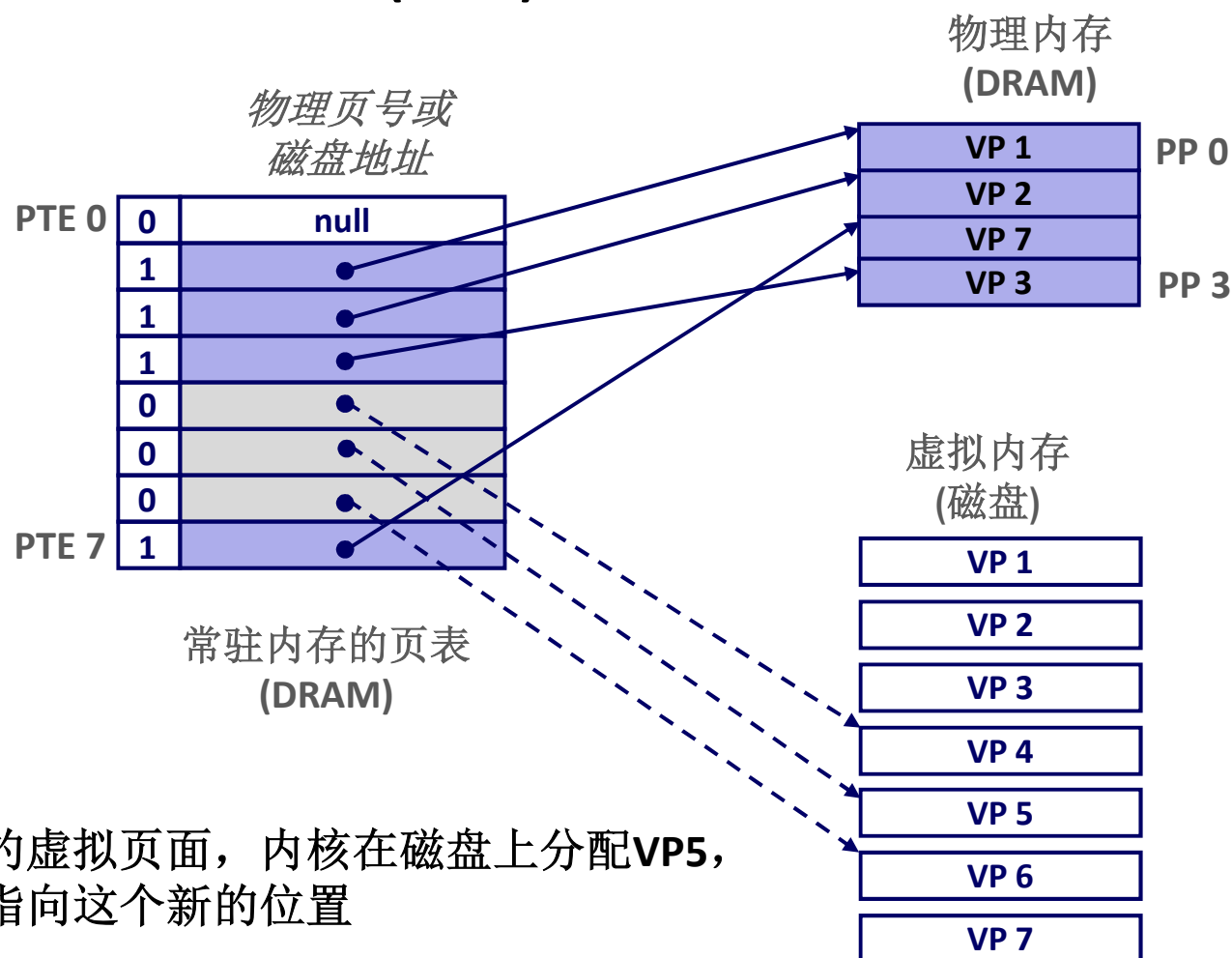
Handling Page Fault 缺页的处理

- 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!



Allocating Pages 分配页面

- 分配一个新的虚拟内存页 (VP 5).



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

Locality to the Rescue Again!

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)
 - 在初始开销后,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。

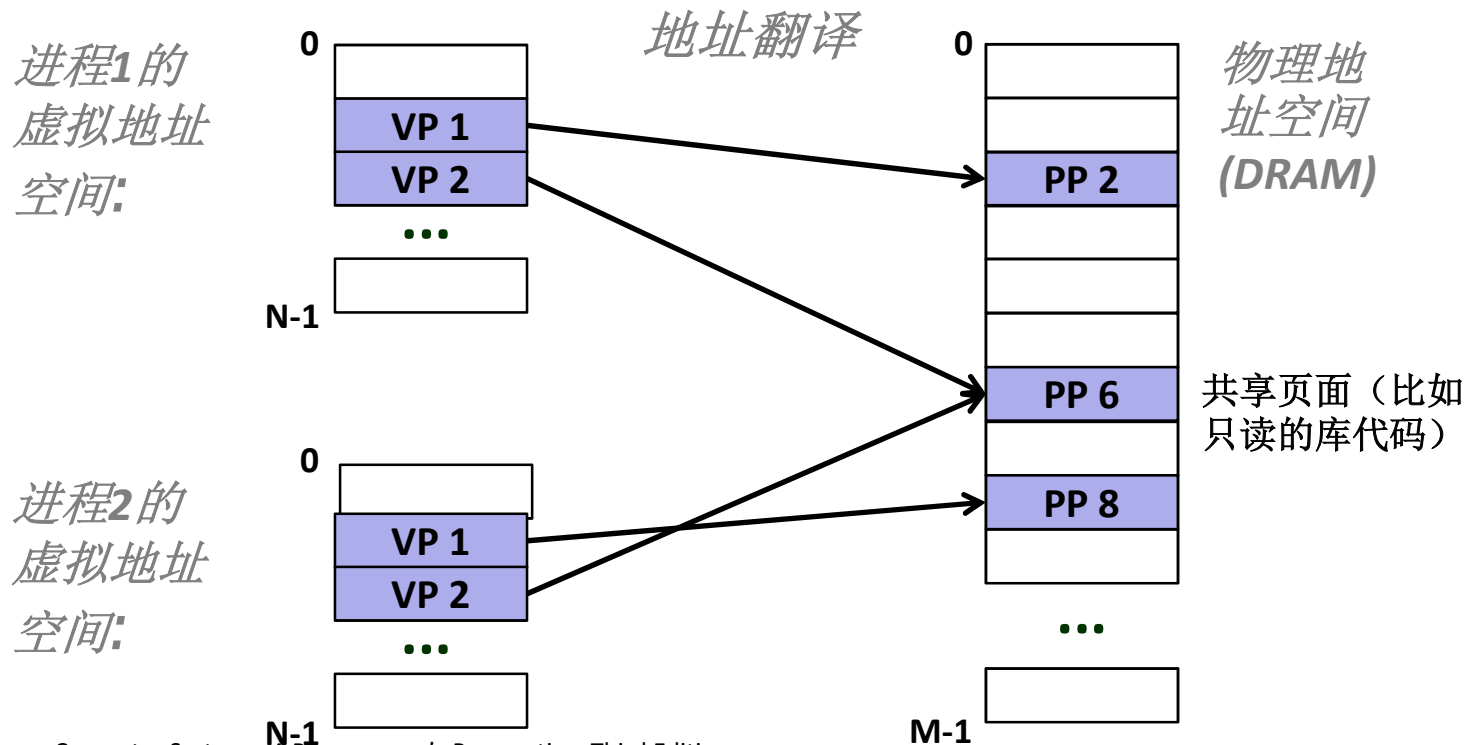
主要内容

- **Address spaces** 地址空间
- **VM as a tool for caching** 虚拟内存作为缓存的工具
- **VM as a tool for memory management**
虚拟内存作为内存管理的工具
- **VM as a tool for memory protection**
虚拟内存作为内存保护的工具有
- **Address translation**地址翻译

VM as a tool for memory management

虚拟内存作为内存管理的工具

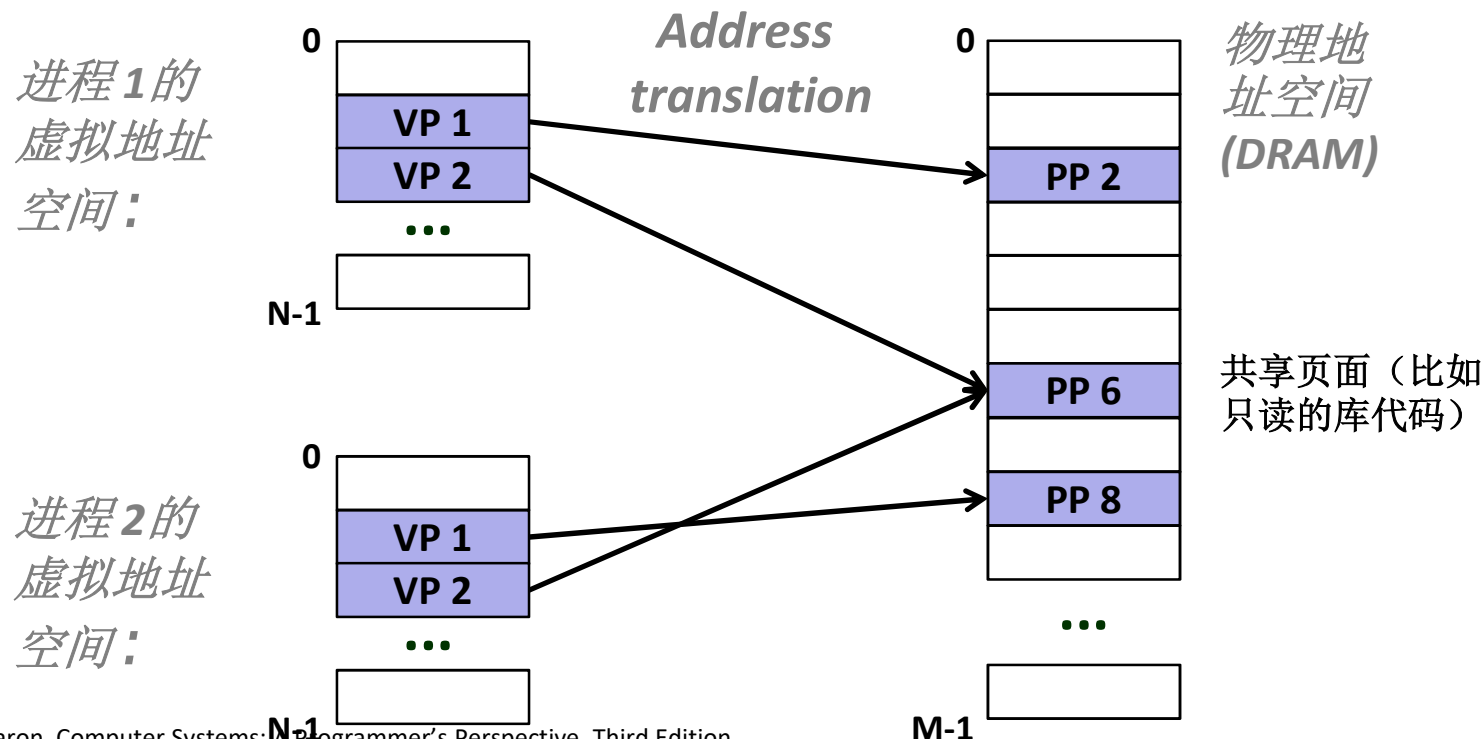
- **Key idea** 核心观点: 每个进程都拥有一个独立的虚拟地址空间
 - 把内存看作独立的简单线性数组
 - 映射函数通过物理内存来分散地址
 - 好的映射函数可以提高程序的局部性



VM as a tool for memory management

虚拟内存作为内存管理的工具

- **Simplifying memory allocation** 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
 - 一个虚拟内存页面每次可以被分配到不同的物理页面
- **Sharing code and data among processes** 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)



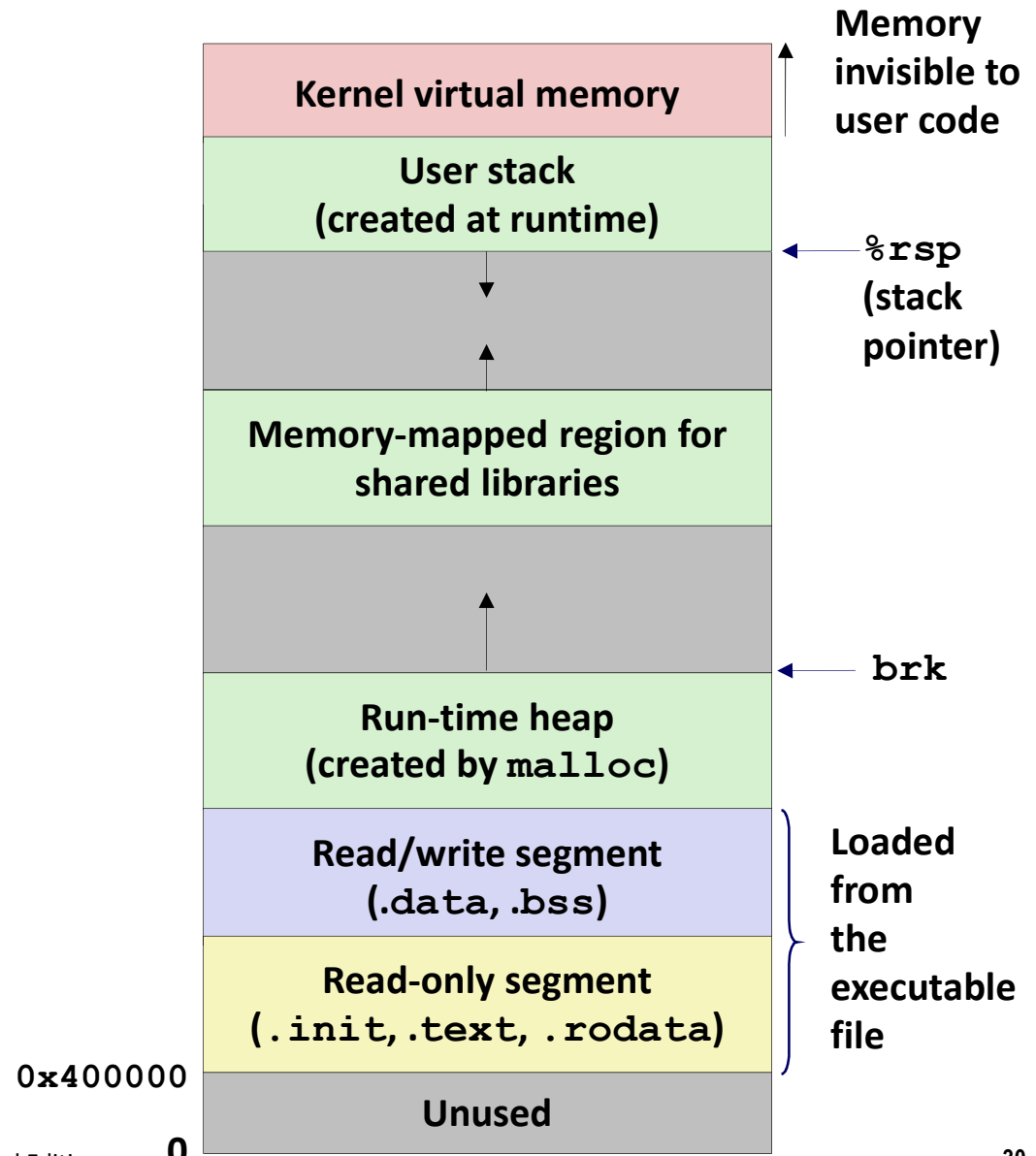
Simplifying Linking and Loading 简化链接和加载

■ Linking 链接

- 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。



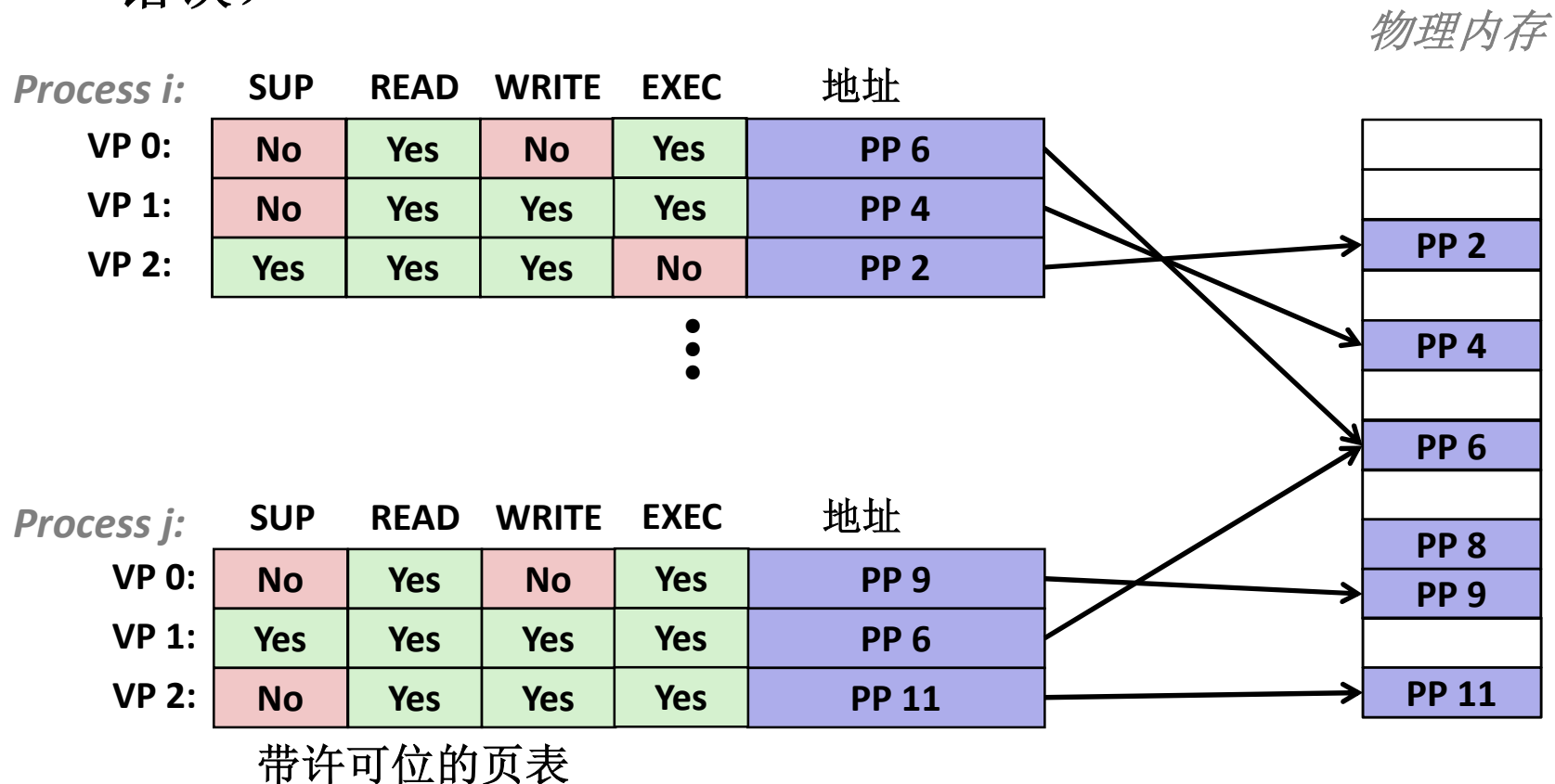
主要内容

- **Address spaces** 地址空间
- **VM as a tool for caching** 虚拟内存作为缓存的工具
- **VM as a tool for memory management**
虚拟内存作为内存管理的工具
- **VM as a tool for memory protection**
虚拟内存作为内存保护的工具有
- **Address translation**地址翻译

VM as a Tool for Memory Protection

虚拟内存作为内存保护的工县

- 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 (段错误)



主要内容

- **Address spaces** 地址空间
- **VM as a tool for caching** 虚拟内存作为缓存的工具
- **VM as a tool for memory management**
虚拟内存作为内存管理的工具
- **VM as a tool for memory protection**
虚拟内存作为内存保护的工具有
- **Address translation**地址翻译

VM Address Translation 地址翻译

- **Virtual Address Space** 虚拟地址空间
 - $V = \{0, 1, \dots, N-1\}$
- **Physical Address Space** 物理地址空间
 - $P = \{0, 1, \dots, M-1\}$
- **Address Translation** 地址翻译
 - $MAP: V \rightarrow P \cup \{\emptyset\}$
 - For virtual address a :
 - $MAP(a) = a'$ 如果虚拟地址 a 处的数据在 p 的物理地址 a' 处
 - $MAP(a) = \emptyset$ 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

■ Basic Parameters 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)

■ Components of the virtual address (VA) 虚拟地址组成部分

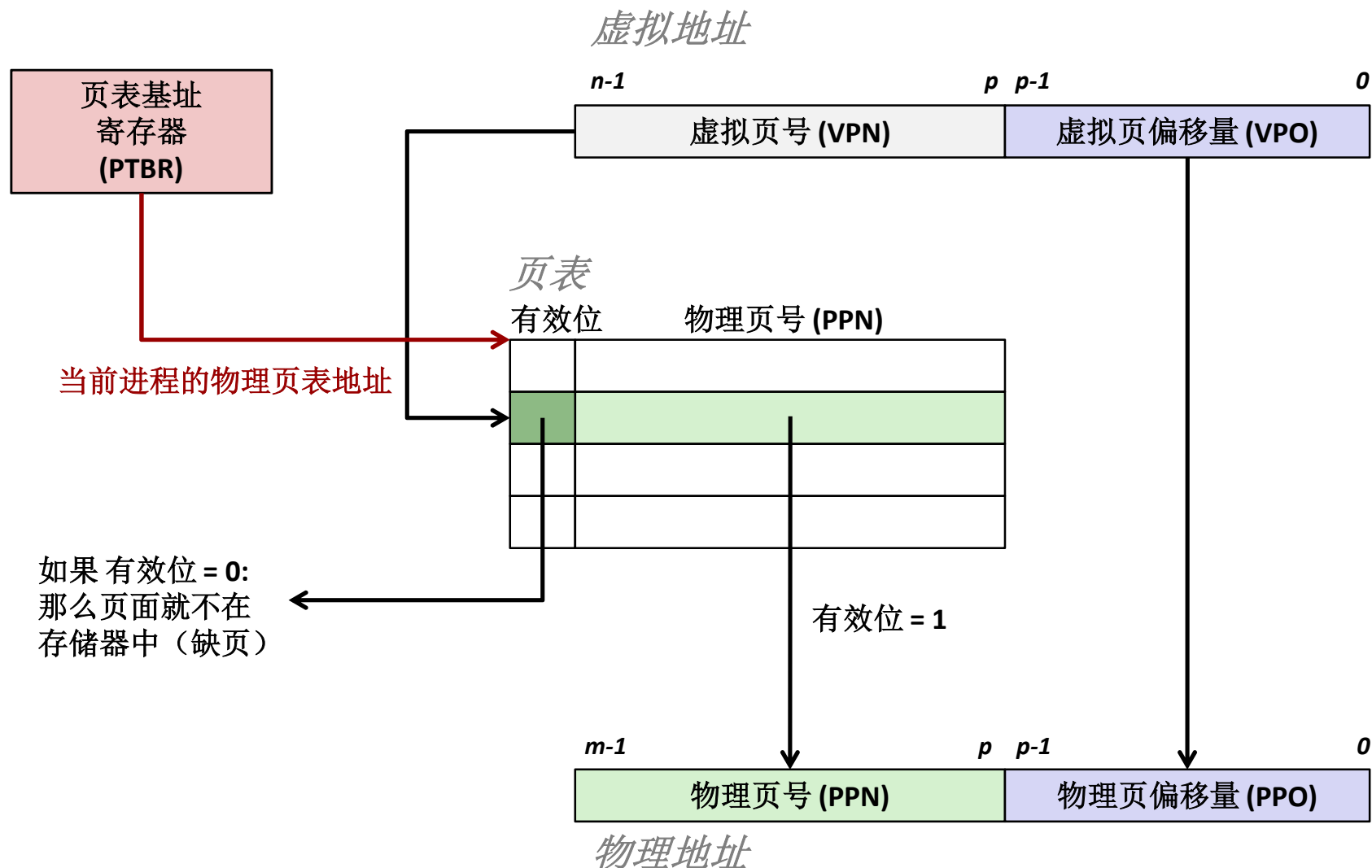
- TLBI: TLB index----TLB索引
- TLBT: TLB tag----TLB标记
- VPO: Virtual page offset----虚拟页面偏移量 (字节)
- VPN: Virtual page number----虚拟页号

■ Components of the physical address (PA) 物理地址组成部分

- PPO: Physical page offset (same as VPO)----物理页面偏移量
- PPN: Physical page number----物理页号

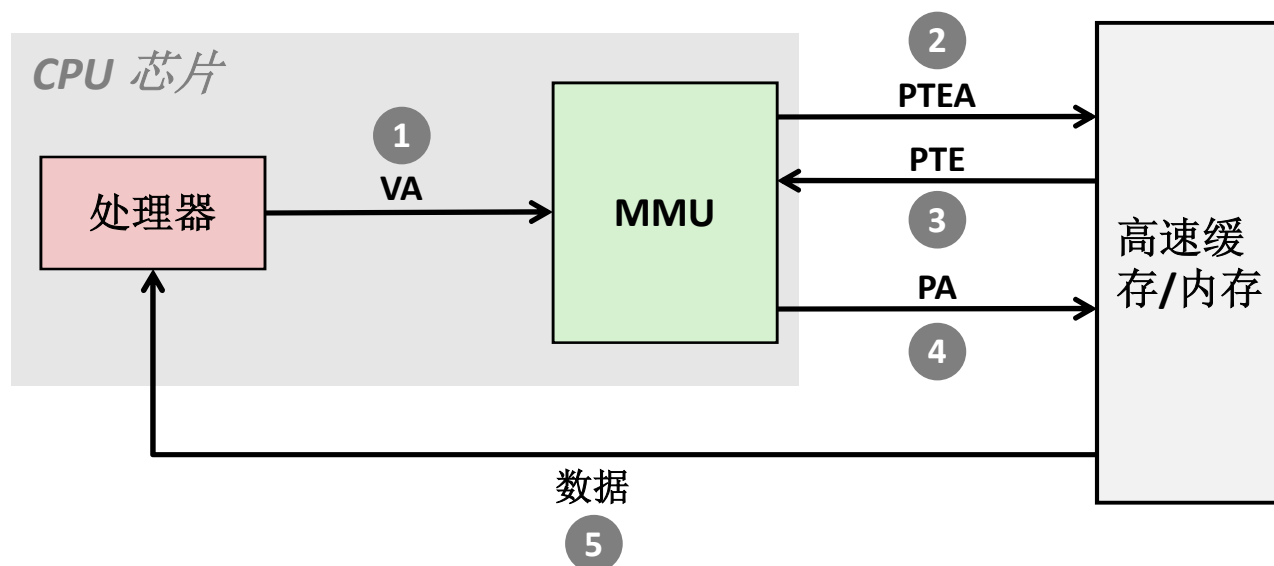
Address Translation With a Page Table

基于页表的地址翻译



Address Translation: Page Hit

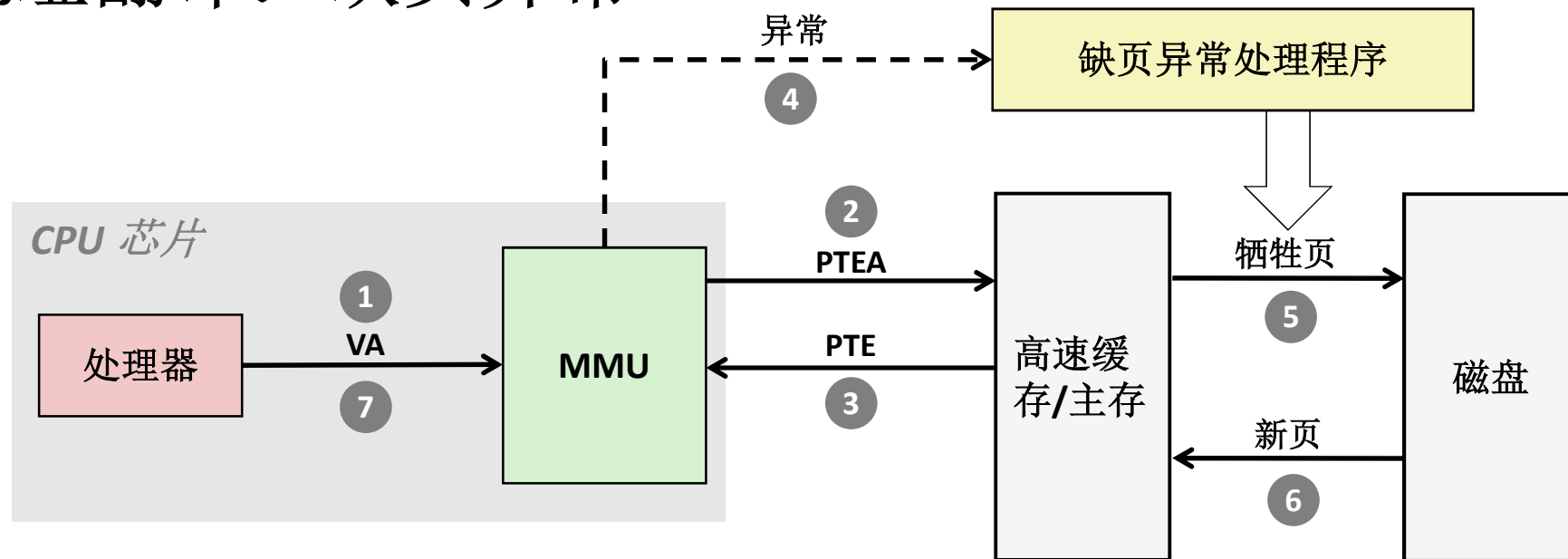
地址翻译：页面命中



- 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
- 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

Address Translation: Page Fault

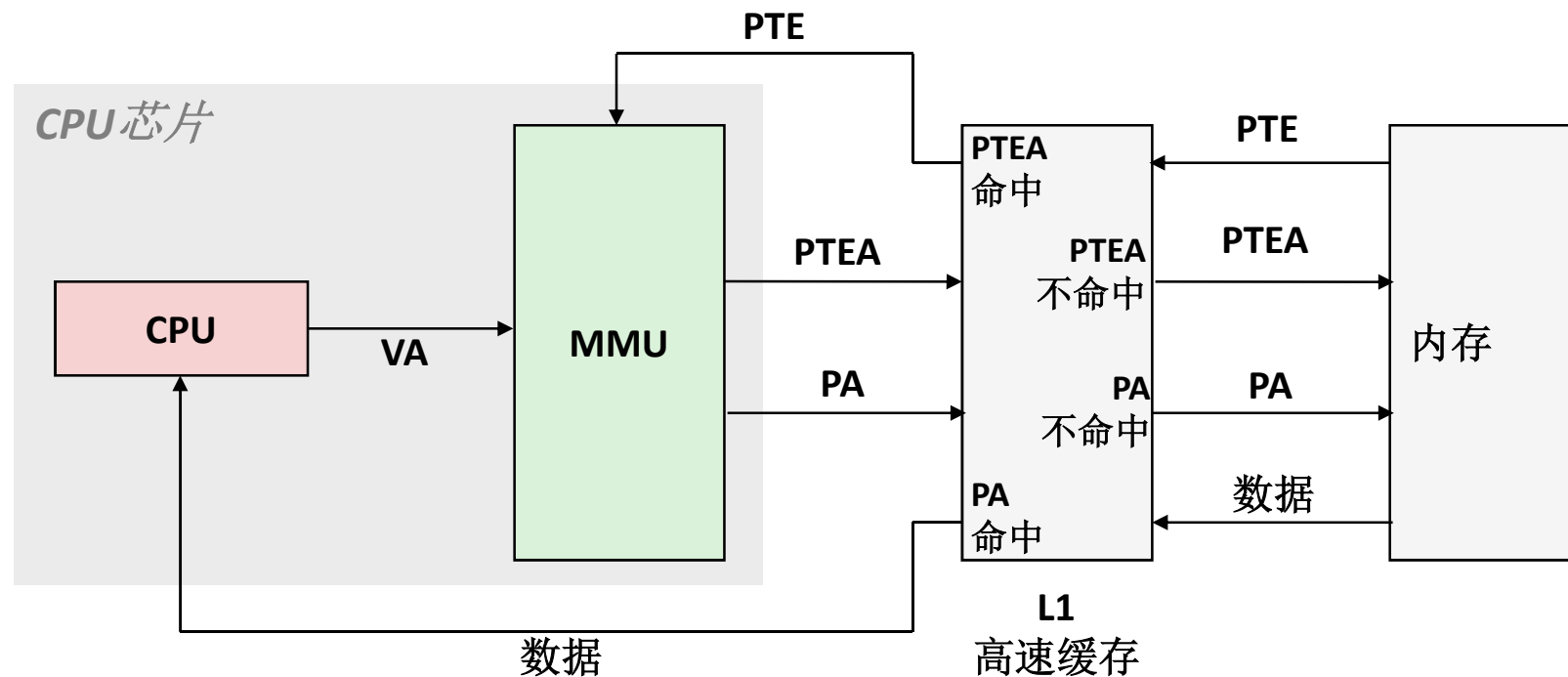
地址翻译：缺页异常



- 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

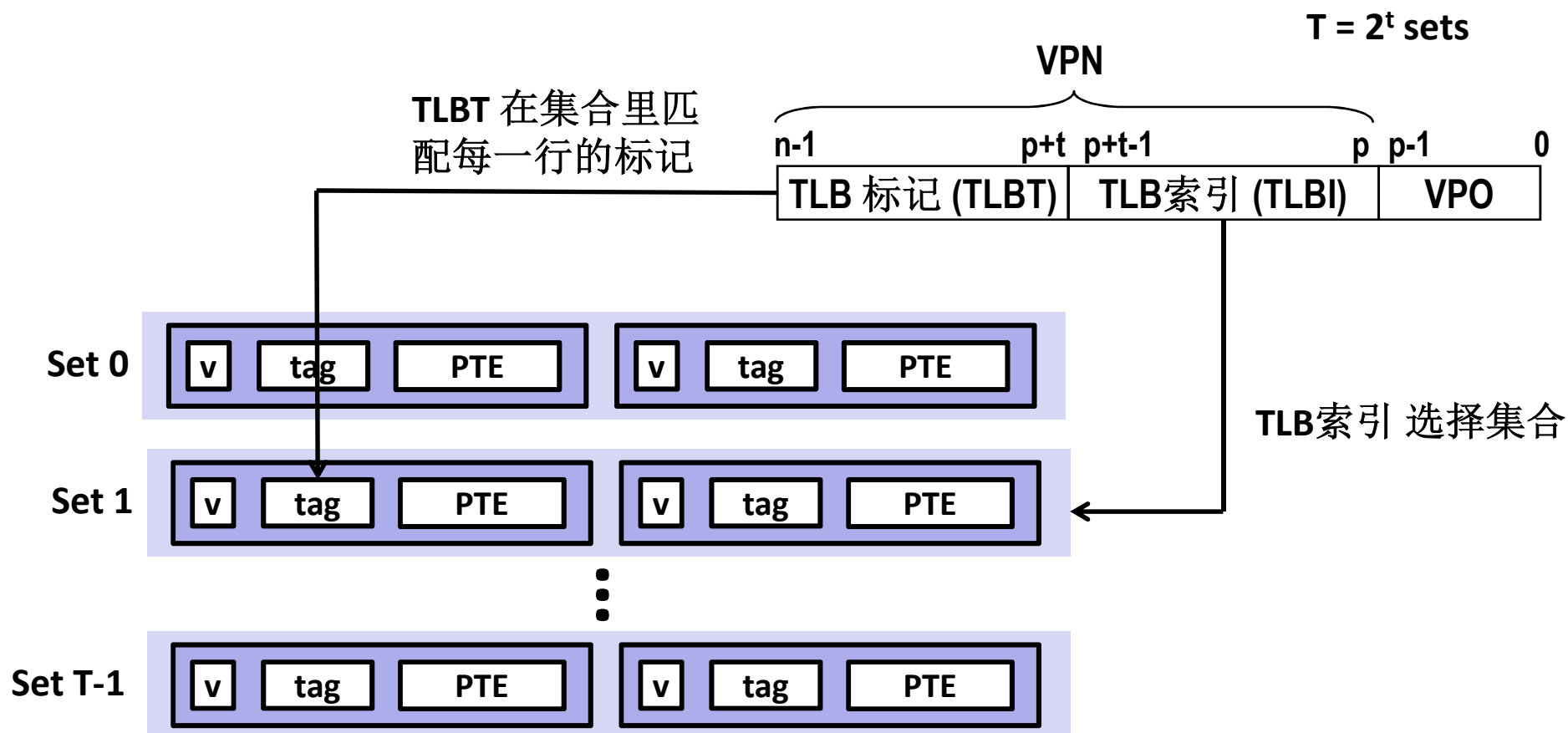
Speeding up Translation with a TLB

利用TLB加速地址翻译

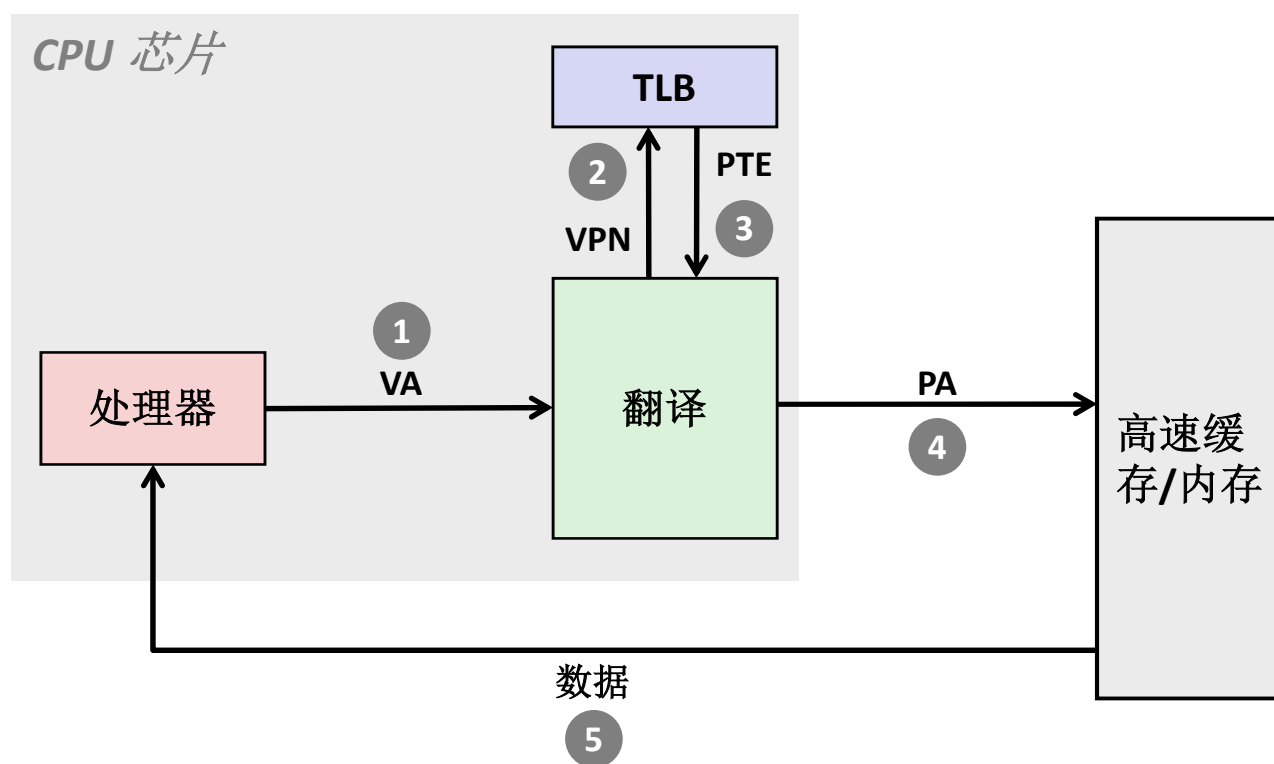
- 页表条目 (PTEs) 恰巧缓存在 L1
 - PTE 可能被其他数据引用所驱逐
 - PTE 命中仍然需要1-2周期的延迟
- 解决办法: *Translation Lookaside Buffer* (TLB)翻译后备缓冲器
 - MMU中一个小的具有高相联度的集合
 - 实现虚拟页码向物理页码的映射
 - 对于页码数很少的页表可以完全包含在TLB中

Accessing the TLB 访问TLB

- MMU 使用虚拟地址的 **VPN** 部分来访问TLB:

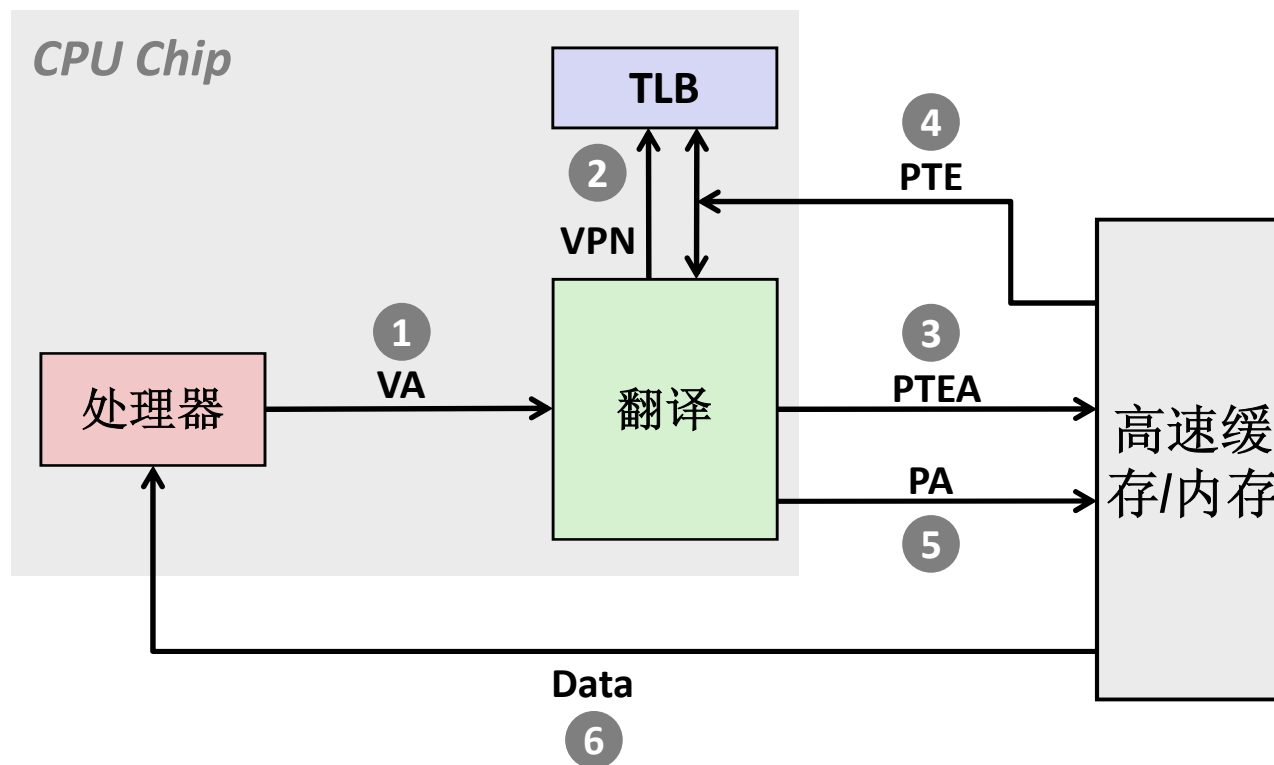


TLB Hit TLB命中



TLB 命中减少内存访问

TLB 不命中

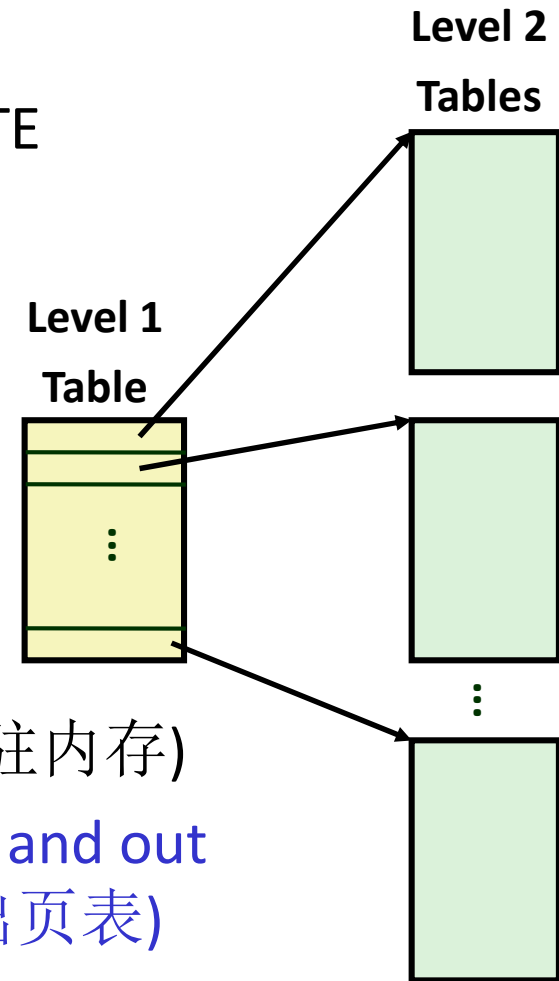


TLB 不命中引发了额外的内存访问

万幸的是, TLB 不命中很少发生。这是为什么呢? --局部性

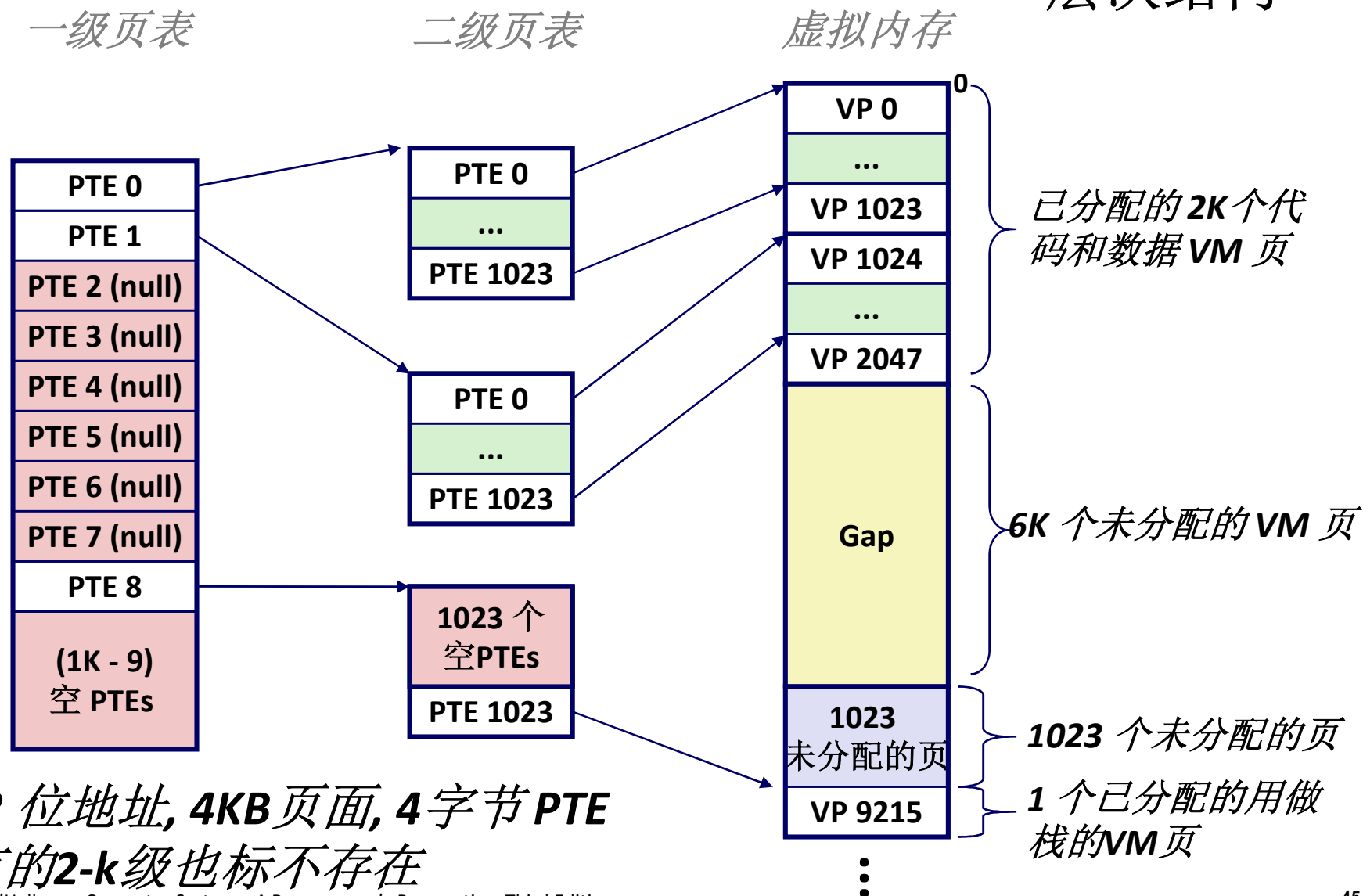
Multi-Level Page Tables 多级页表

- 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 (paged in and out like any other data 页面可以调入或调出页表)



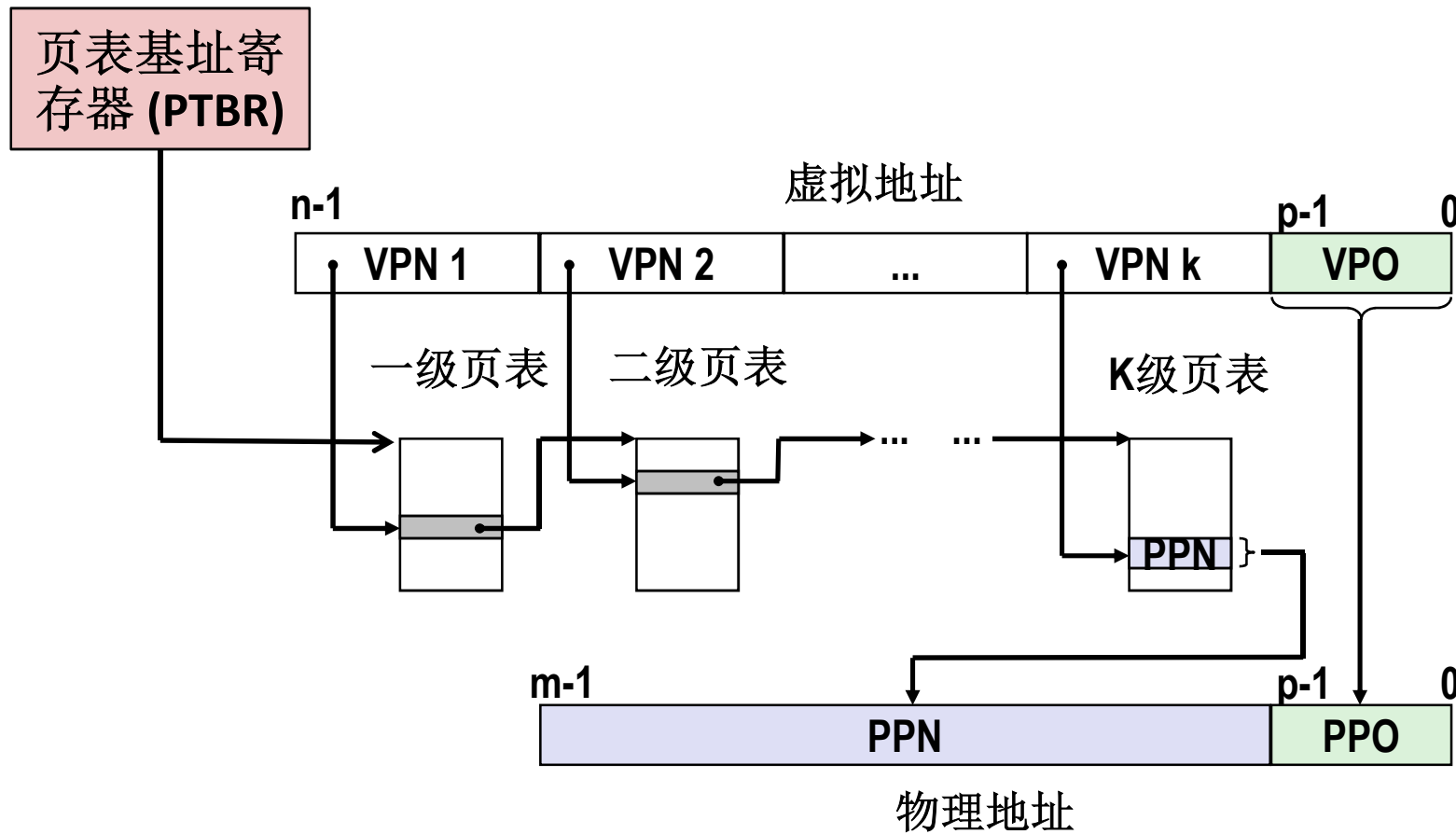
A Two-Level Page Table Hierarchy

二级页表的层次结构



Translating with a k-level Page Table

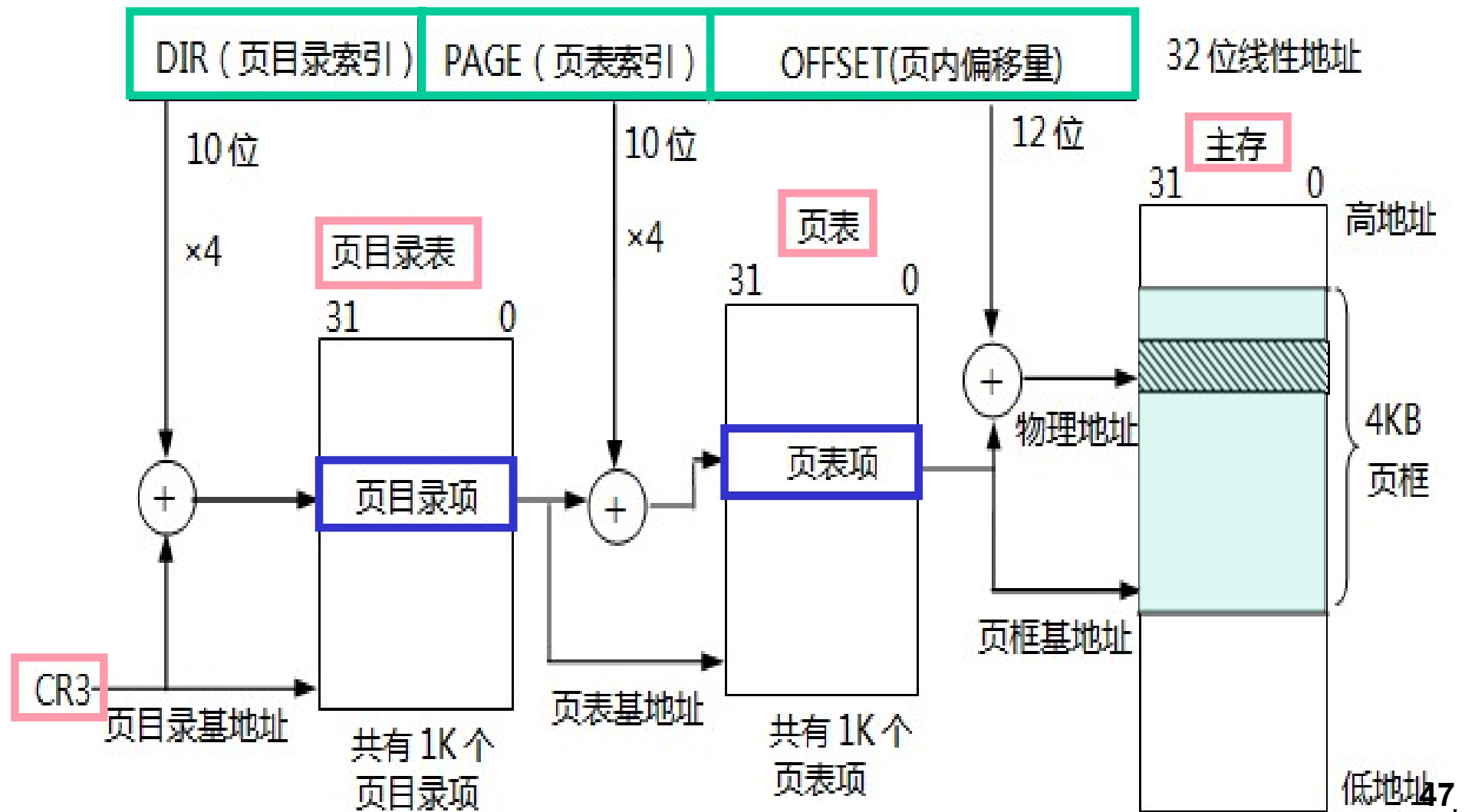
使用K级页表的地址翻译



IA32线性地址向物理地址转换

线性地址空间划分：4GB=1K个子空间 * 1K个页面/子空间 * 4KB/页

- 页目录项和页表项格式一样，有32位（4B）



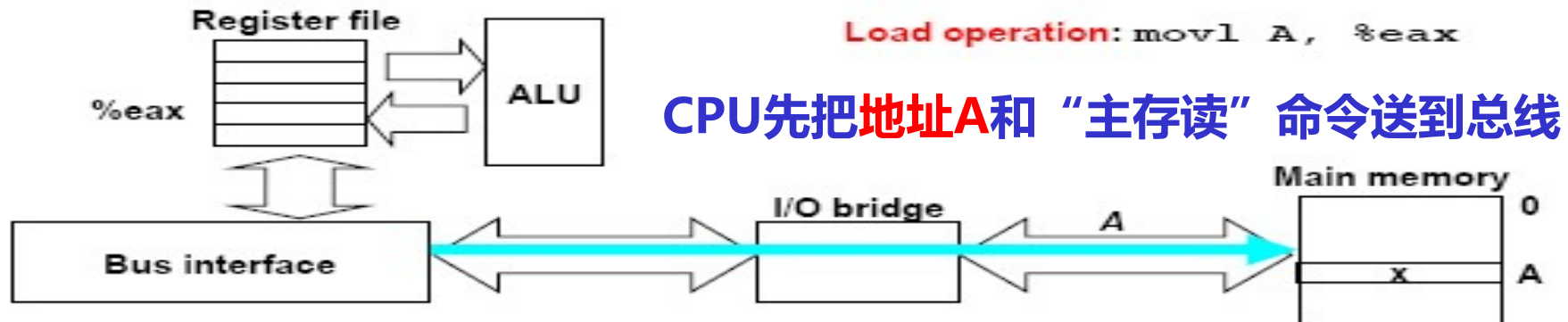
IA-32的页目录项和页表项

31	12	11	10	09	8	7	6	5	4	3	2	1	0
基地址			AVL		0	0	D	A	PCD	PWT	U/S	R/W	P

- **P**: 1表示页表或页在主存中; P=0表示页表或页不在主存, 即缺页, 此时需将页故障线性地址保存到CR2。
- **R/W**: 0表示页表或页只能读不能写; 1表示可读可写。
- **U/S**: 0表示用户进程不能访问; 1表示允许访问。
- **PWT**: 控制页表或页的cache写策略是全写还是回写 (Write Back) 。
- **PCD**: 控制页表或页能否被缓存到cache中。
- **A**: 1表示指定页表或页被访问过, 初始化时OS将其清0。利用该标志, OS可清楚了解哪些页表或页正在使用, 一般选择长期未用的页或近来最少使用的页调出主存。由MMU在进行地址转换时将该位置1。
- **D**: 修改位(脏位dirty bit)。页目录项中无意义, 只在页表项中有意义。初始化时OS将其清0, 由MMU在进行写操作的地址转换时将该位置1。
- 高20位是页表或页在主存中的首地址对应的页框号, 即首地址的高20位。
每个页表的起始位置都按4KB对齐。

回顾：指令“`movl 8(%ebp), %eax`”操作过程

由`8(%ebp)`得到主存地址A的过程较复杂，涉及MMU、TLB、页表等许多重要概念！



■ IA-32中，执行“`movl 8(%ebp), %eax`”中取数操作的大致过程如下：

- 若 $CPL > DPL$ 则越级，否则计算有效地址 $EA = R[ebp] + 0 \times 0 + 8$
- 通过段寄存器找到段描述符以获得段基址，线性地址 $LA = \text{段基址} + EA$
- 若“ $LA > \text{段限}$ ”则越界，否则将LA转换为主存地址A
 - 若访问TLB命中则地址转换得到A；否则处理TLB缺失（硬件/OS）
 - 若缺页或越权(R/W不符)则调出OS内核；否则地址转换得到A
 - 根据A先到Cache中找，若命中则取出A在Cache中的副本
 - 若Cache不命中，则再到主存取A所在主存块送对应Cache行

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
- Simplifies protection by providing a convenient interpositioning point to check permissions
提供方便的标志位来检查权限以简化内存保护