哈爾濱工業大學（深圳）

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

操作系统 (Operating System)

第七章：虚拟内存

陈芳林　副教授

哈尔滨工业大学（深圳）

2024年秋

Email：chenfanglin@hit.edu.cn

主要内容

- 7.1 背景

（1）内存不够用怎么办（2）内存管理视图（3）用户眼中的内存（4）虚拟内存的优点

- 7.2 虚拟内存实现--按需调页（请求调页）

（1）交换与调页（2）页表的改造（3）请求调页过程

- 7.3 页面置换

（1）FIFO页面置换（2）OPT（最优）页面置换

（3）LRU页面置换（准确实现：计数器法、页码栈法）

（4）近似LRU页面置换（附加引用位法、时钟法）

- 7.4 其他相关问题

（1）写时复制（2）交换空间（交换区）与工作集（3）页置换策略:全局置换和局部置换

（4）系统颠簸现象和Belady异常现象 （5）虚拟内存中程序优化

- 7.5 CSAPP第9章<虚拟内存>串讲

CSAPP<虚拟内存>串讲

- 虚拟地址的翻译
- 全相连/组相连/直接映射
- 案例研究: Core i7/Linux 内存系统
- 一个小内存系统示例
- 内存映射

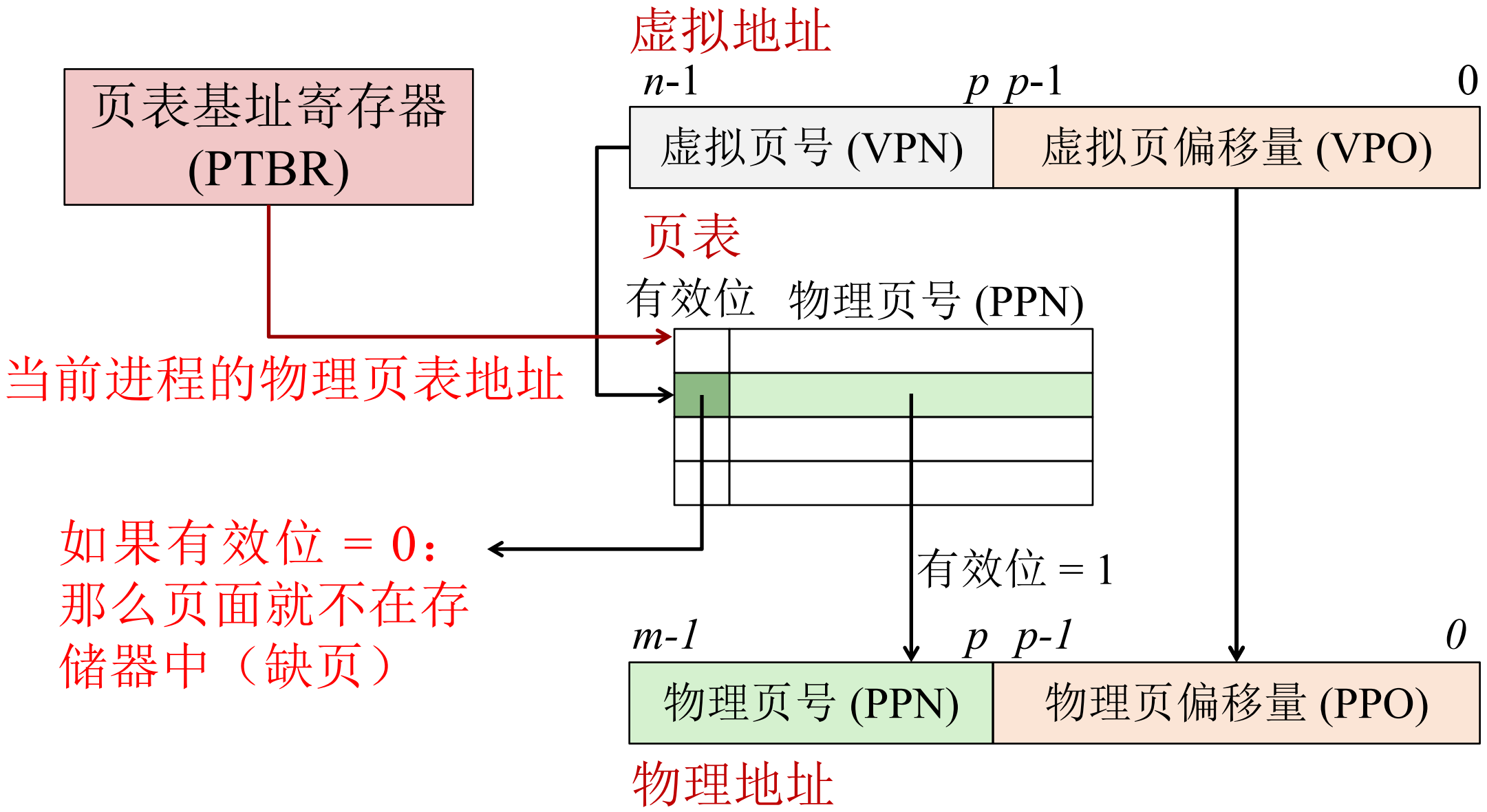
VM Address Translation 地址翻译

- Virtual Address Space 虚拟地址空间
  - V = {0, 1, …, N–1}
- Physical Address Space 物理地址空间
  - P = {0, 1, …, M–1}
- Address Translation 地址翻译
  - 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p的物理地址 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

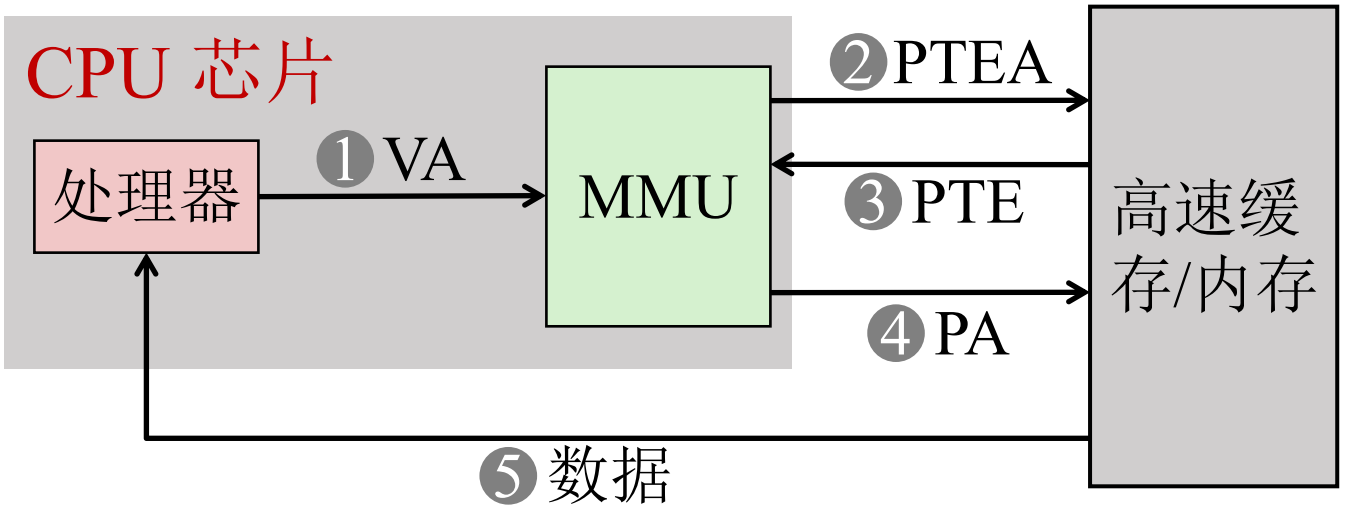
地址翻译使用到的所有符号

- 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

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
有效位
物理页号 (PPN)
页表基址寄存器
(PTBR)
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



VA：virtual address虚拟地址

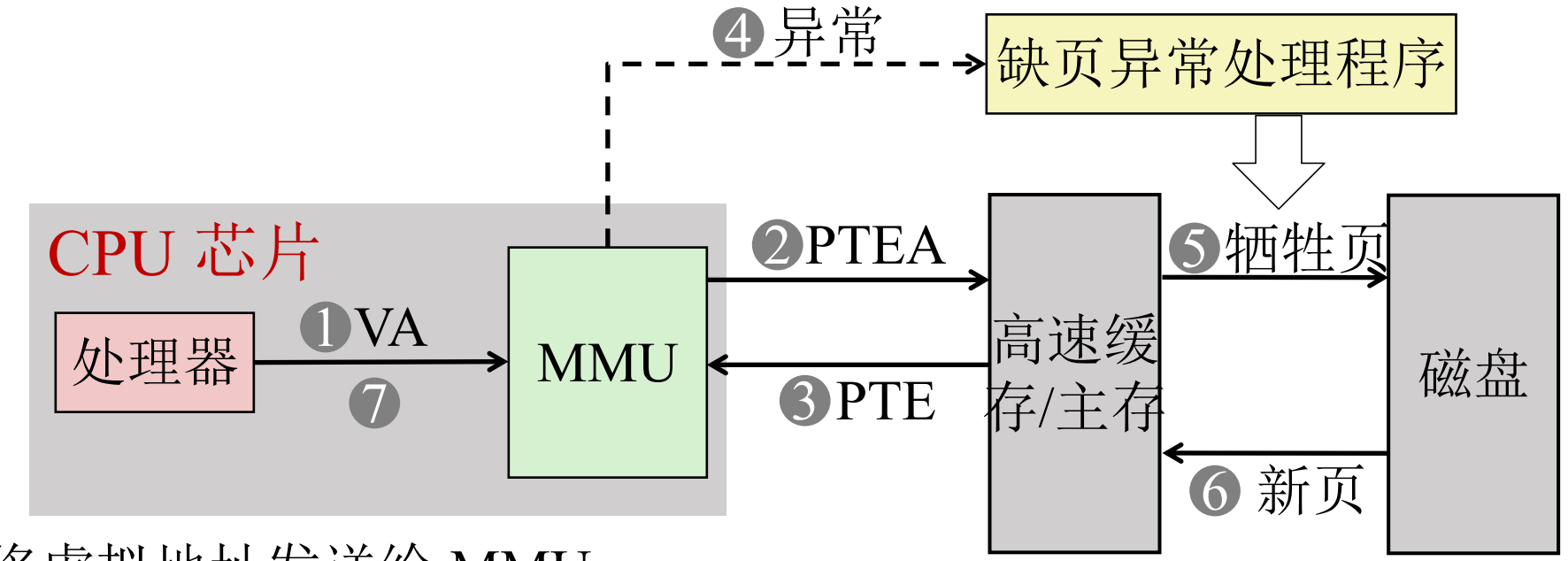
PA：physical address物理地址

PTE：page table entry页表条目

PTEA：PTE address页表条目地址

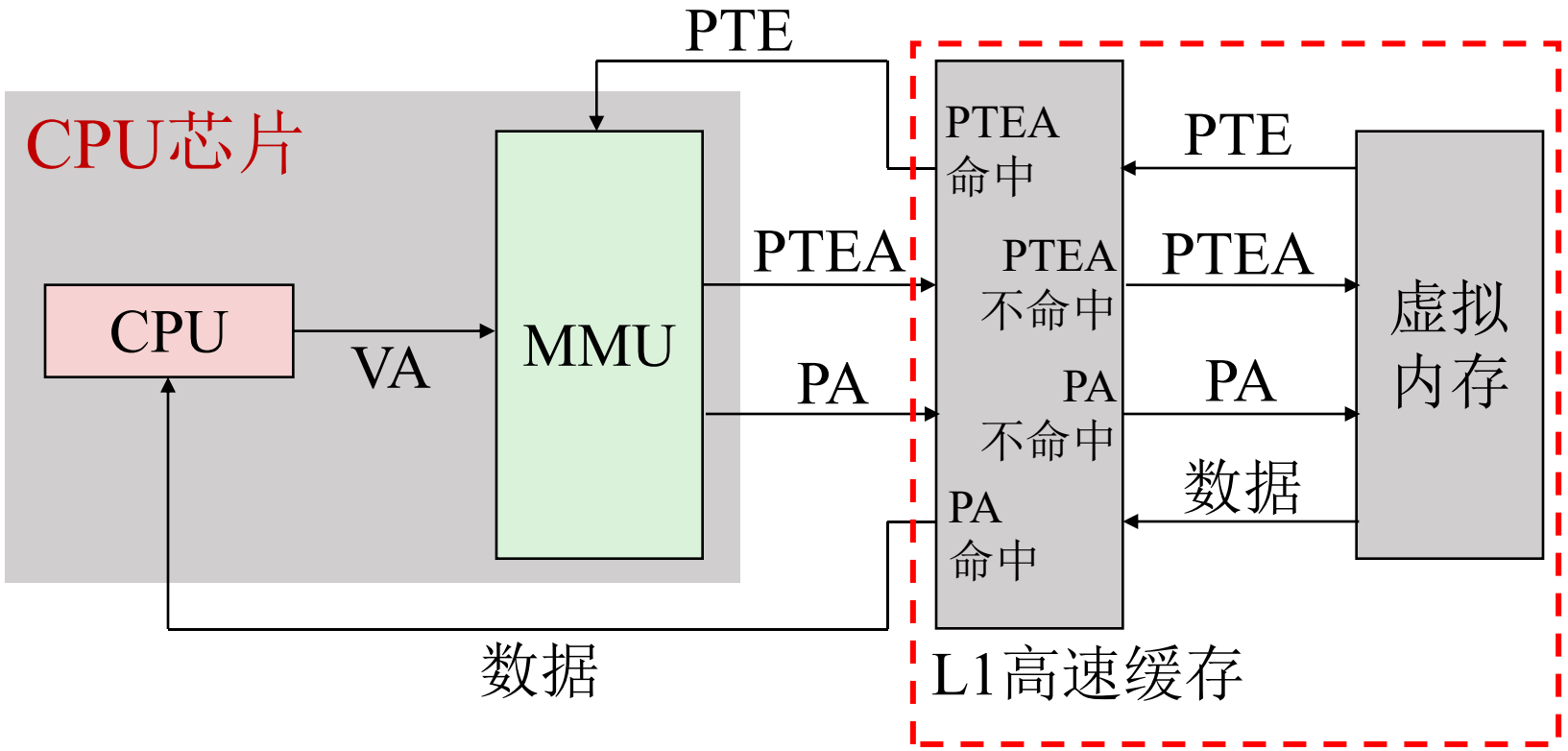
- 1) 处理器生成一个虚拟地址，并将其传送给MMU
- 2-3) MMU 使用内存中的页表生成PTE地址
- 4) MMU 将物理地址传送给高速缓存/主存
- 5) 高速缓存/主存返回所请求的数据字给处理器

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替换页（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，再次执行引起缺页的指令

结合高速缓存和虚拟内存



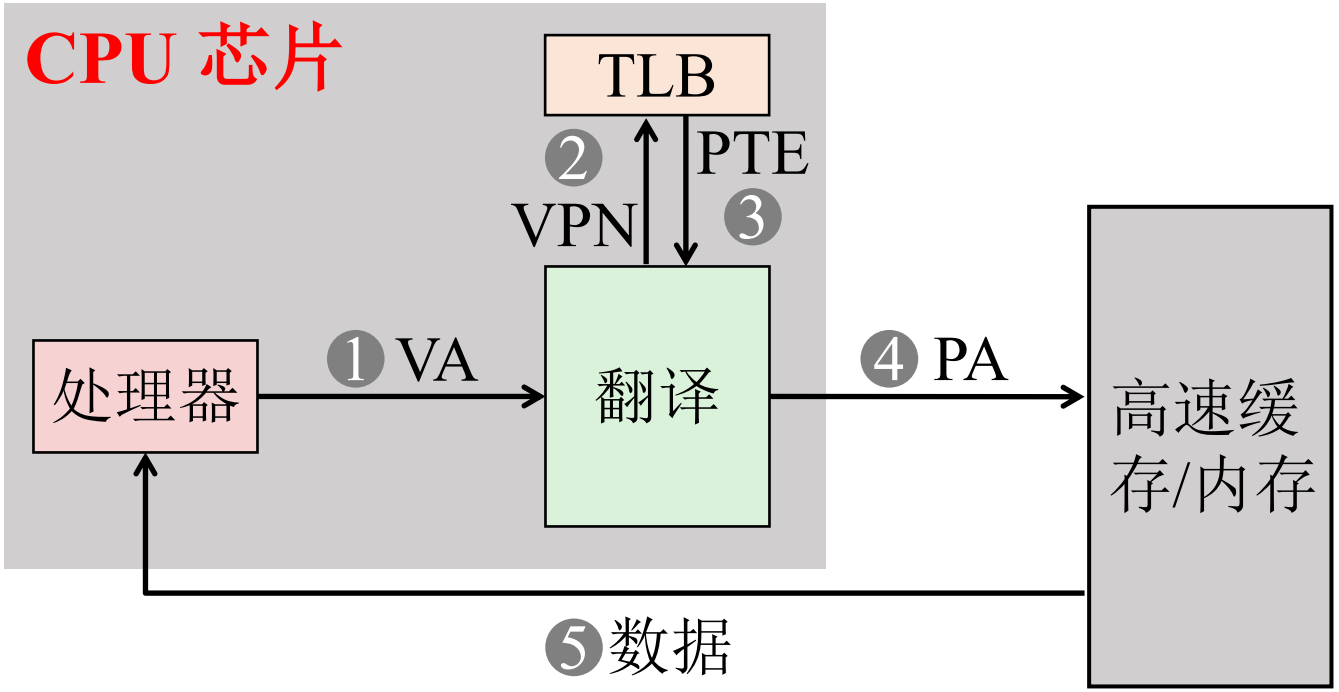
VA：virtual address虚拟地址

PA：physical address物理地址

PTE：page table entry页表条目

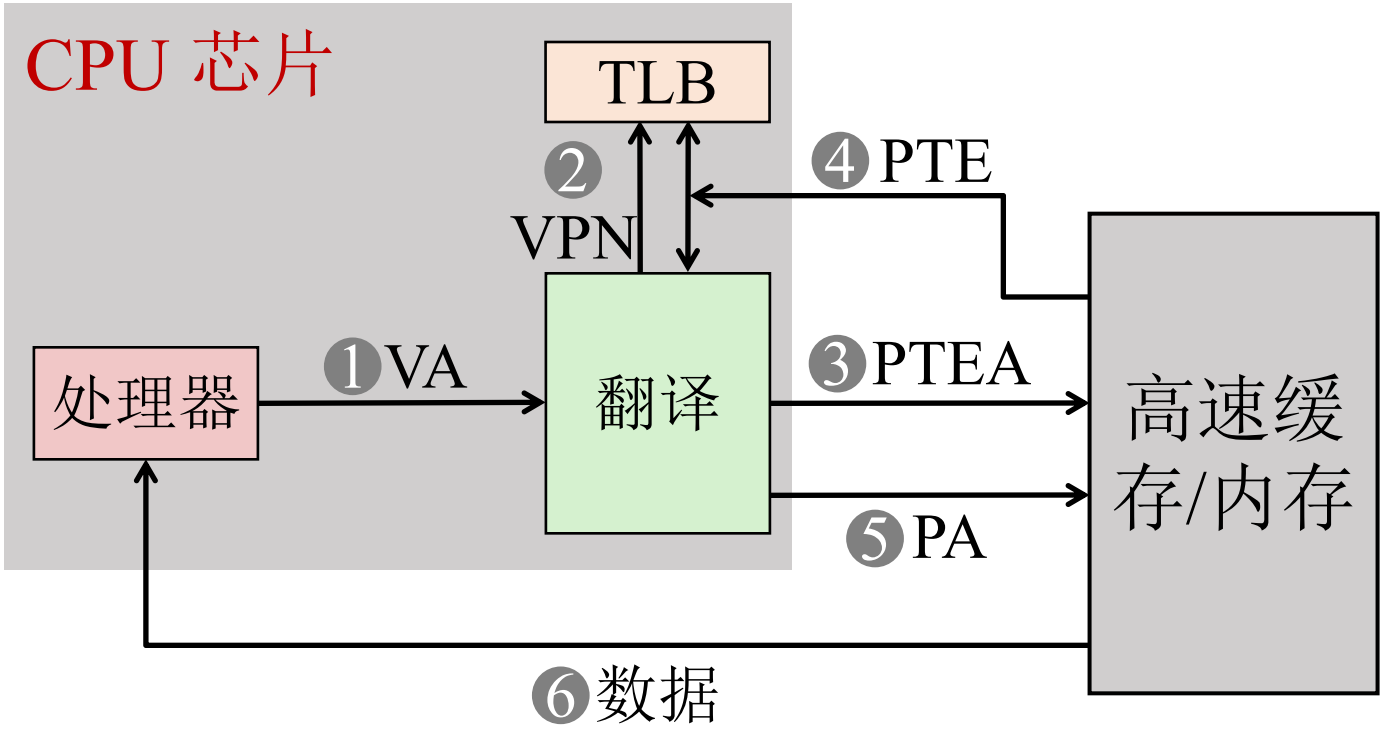
PTEA：PTE address页表条目地址

TLB命中



- TLB 命中减少内存访问

TLB 不命中



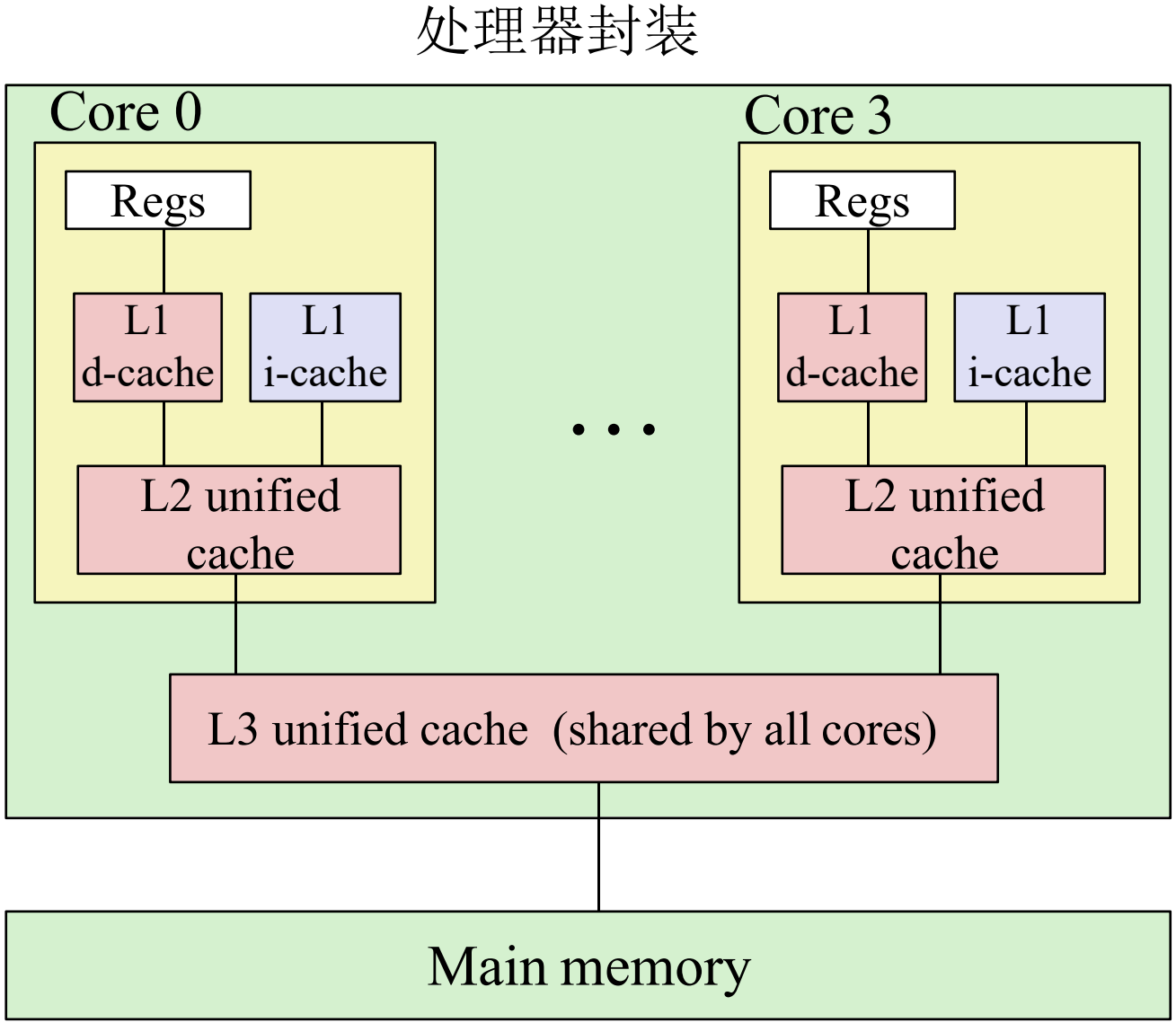
- TLB 不命中引发了额外的内存访问
- 万幸的是, TLB 不命中很少发生。这是为什么呢？--局部性

CSAPP<虚拟内存>串讲

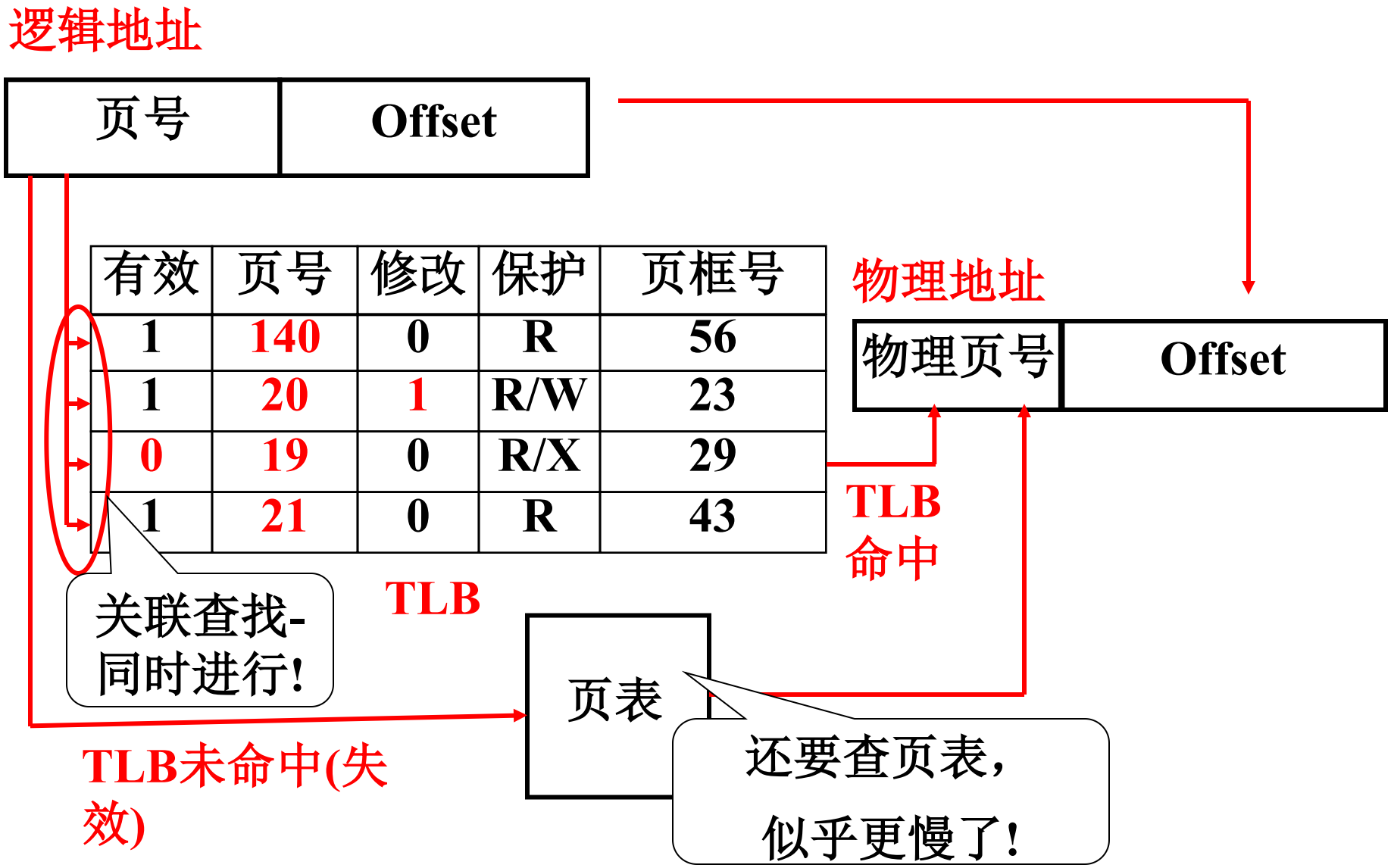
- 虚拟地址的翻译
- 全相连/组相连/直接映射
  - 主存地址与缓存地址的映象及转换
  - 按一定原则对Cache的内容进行替换
- 案例研究: Core i7/Linux 内存系统
- 一个小内存系统示例
- 内存映射

Intel Core i7高速缓存层次结构

- L1 指令高速缓存和数据高速缓存:
  - 32 KB, 8-way,
  - 访问时间: 4周期
- L2 统一的高速缓存:
  - 256 KB, 8-way,
  - 访问时间: 10 周期
- L3 统一的高速缓存:
  - 8 MB, 16-way,
  - 访问时间: 40-75 周期
- 块大小: 所有缓存都是64 字节



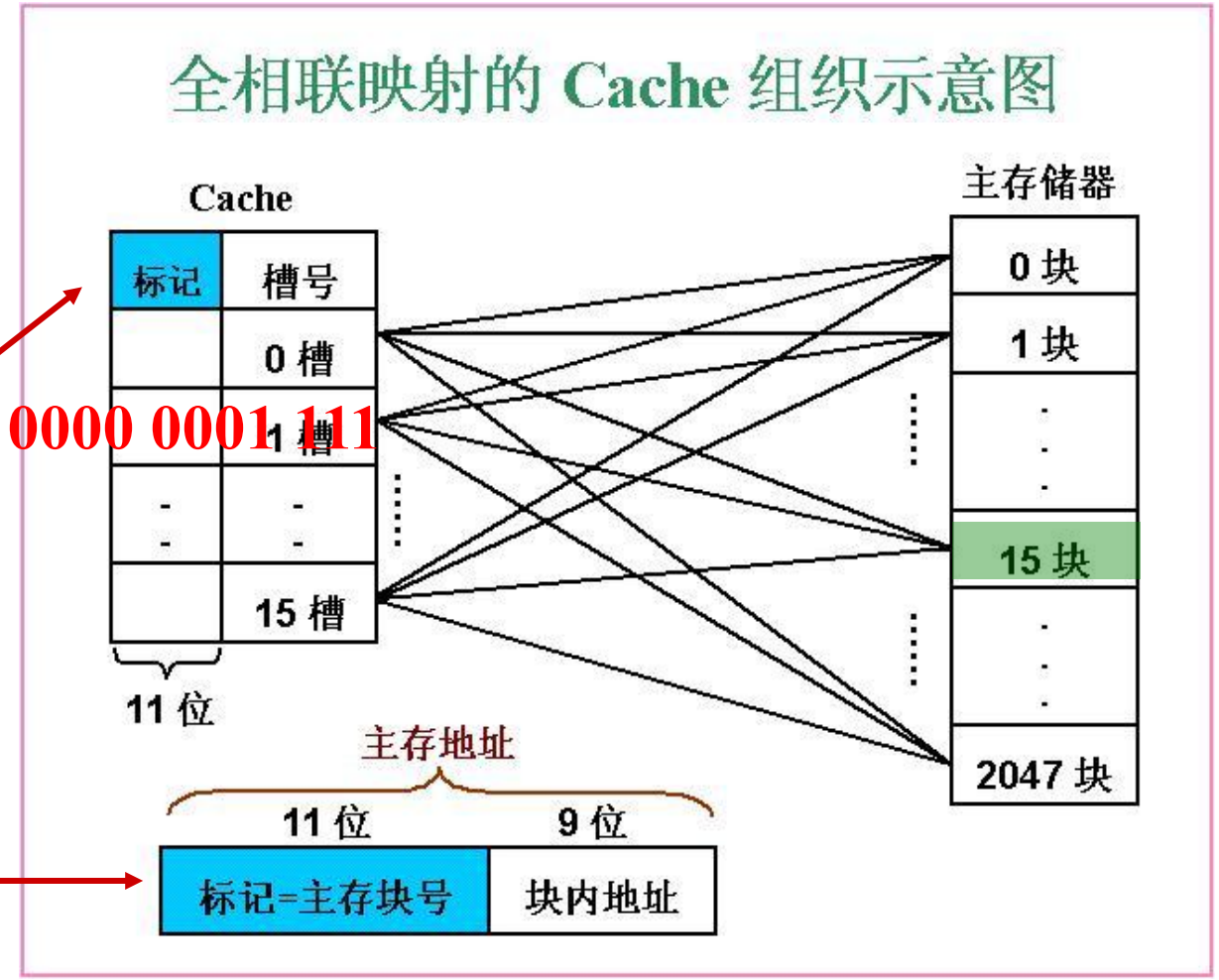
回顾：采用TLB后的地址翻译



全相联映射Cache组织示意图

- 假定数据在主存和Cache间的传送单位为512字。
- Cache大小：2^{13}字=8K字=16行 × 512字/行
- 主存大小：2^{20}字=1024K字=2048块 × 512字/块
- Cache标记（tag）指出对应行取自哪个主存块
- 主存tag指出对应地址位于哪个主存块

每个主存块可装到Cache任一行中



如何对0x01E0C单元进行访问？

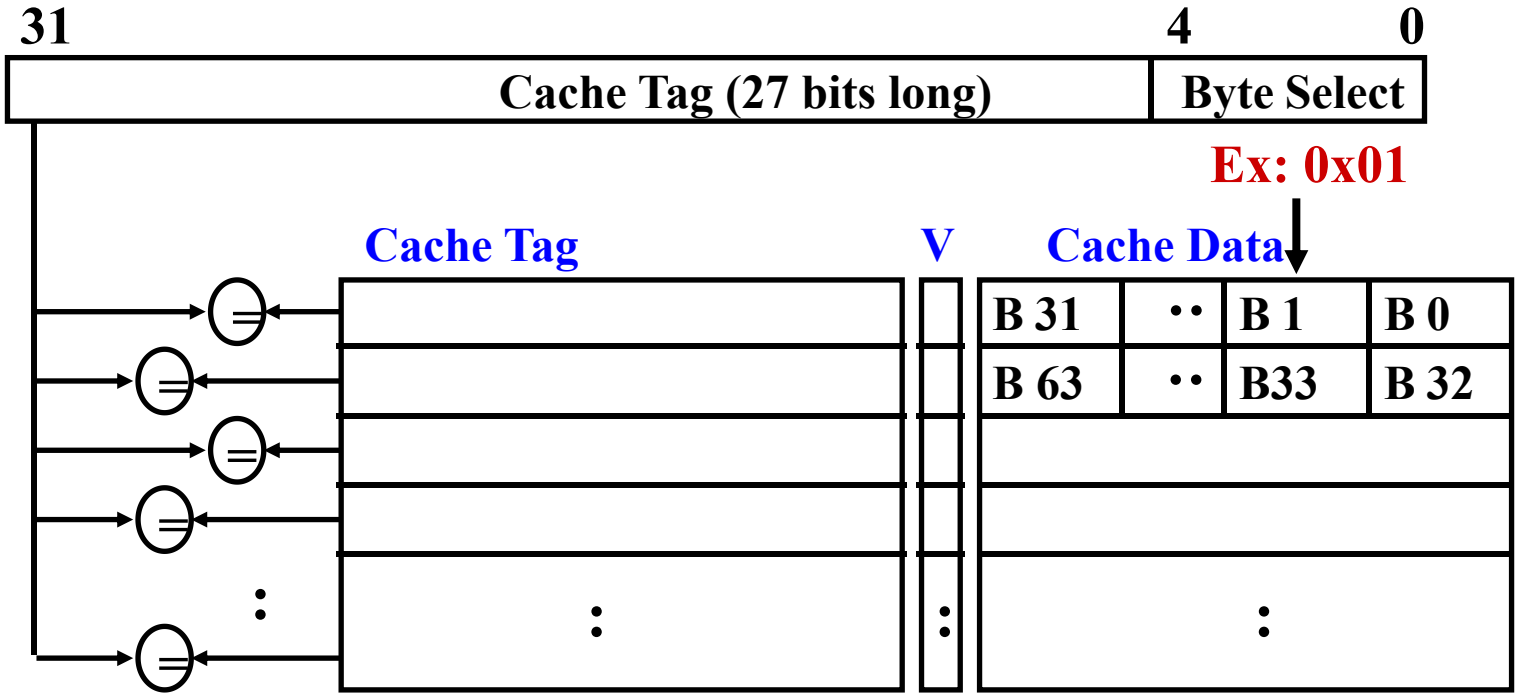
0000 0001 1110 0000 1100 是第15块中的第12个单元！

如何实现按内容访问？ **直接比较！**

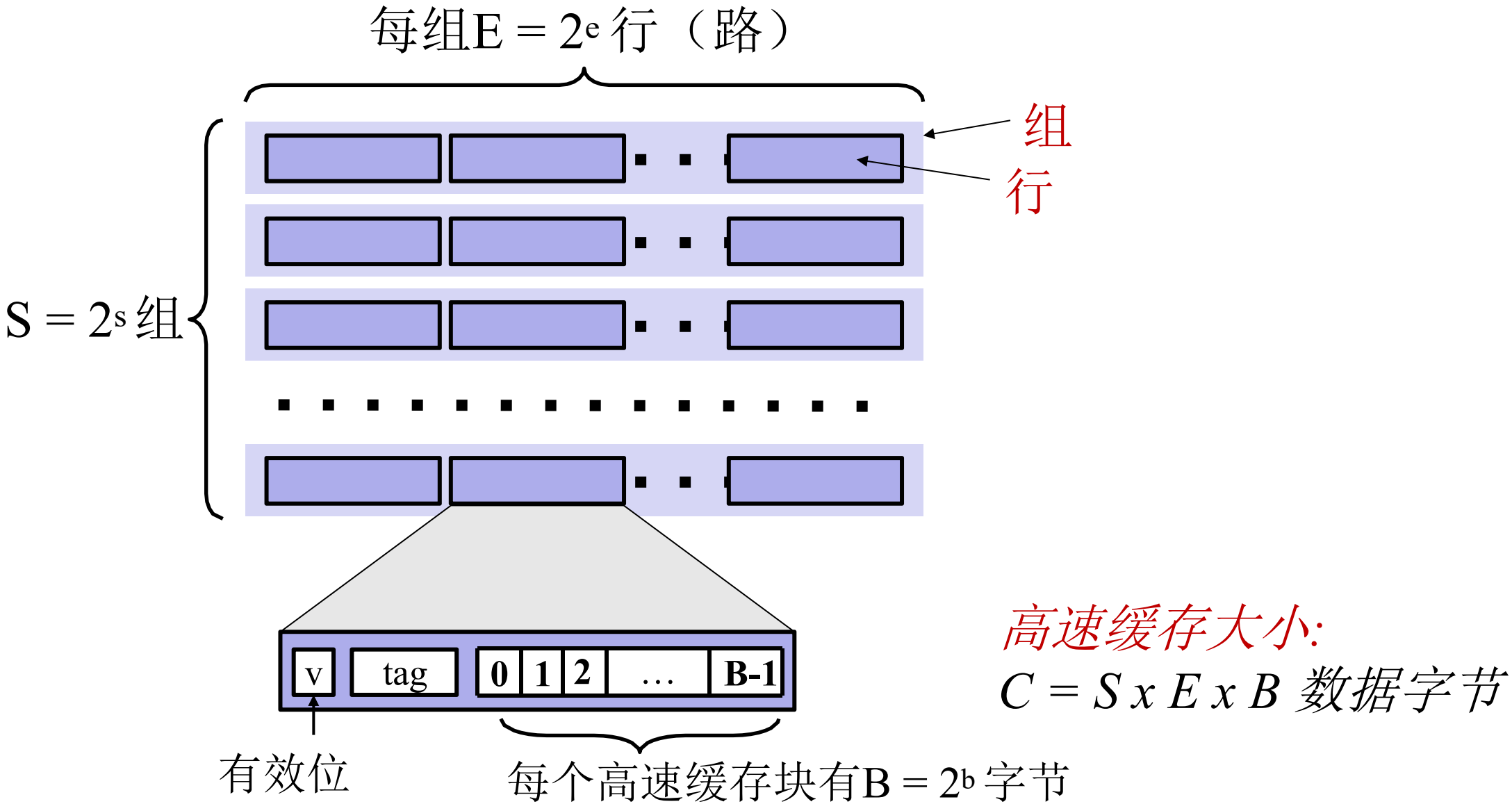
举例：全相联

- 全相联 Cache
  - 同时比较所有Cache项的标志

- Example： 32bits 内存地址, 32 B 块大小。
  - 比较器位数多长？
    需要 27-bit 大小的比较器
  - 需要多少个比较器？
    1024个比较器！



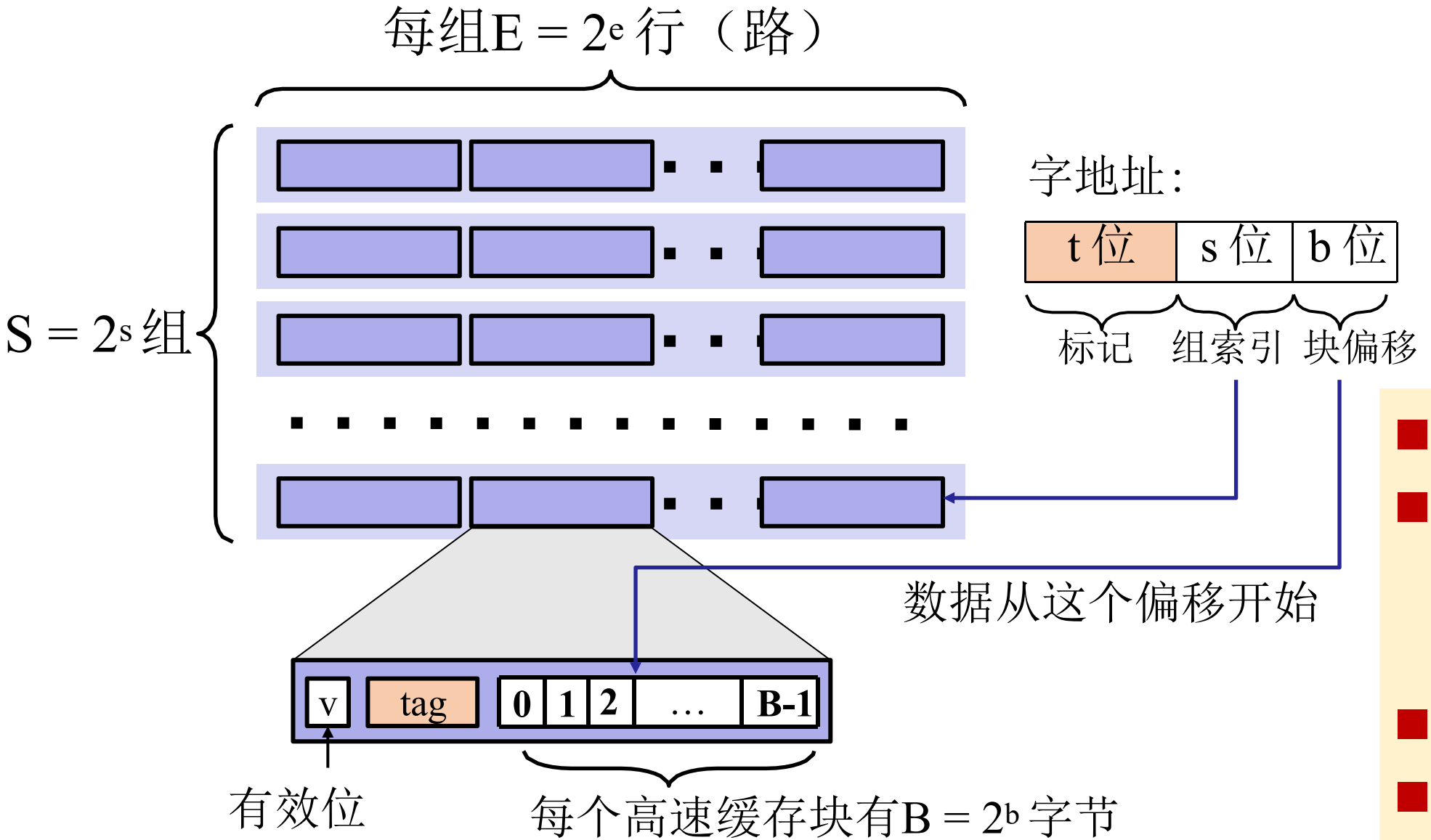
高速缓存通用组织(S, E, B)



高速缓存大小:

$C = S \times E \times B$ *数据字节*

高速缓存通用组织(S, E, B)



- 定位组
- 检查集合中的任何行是否有匹配的标记
- 是 + 行有效: 命中
- 定位从偏移开始的数据

例子: Core i7 L1 数据缓存

32 kB 的8路组相联

64 字节/块

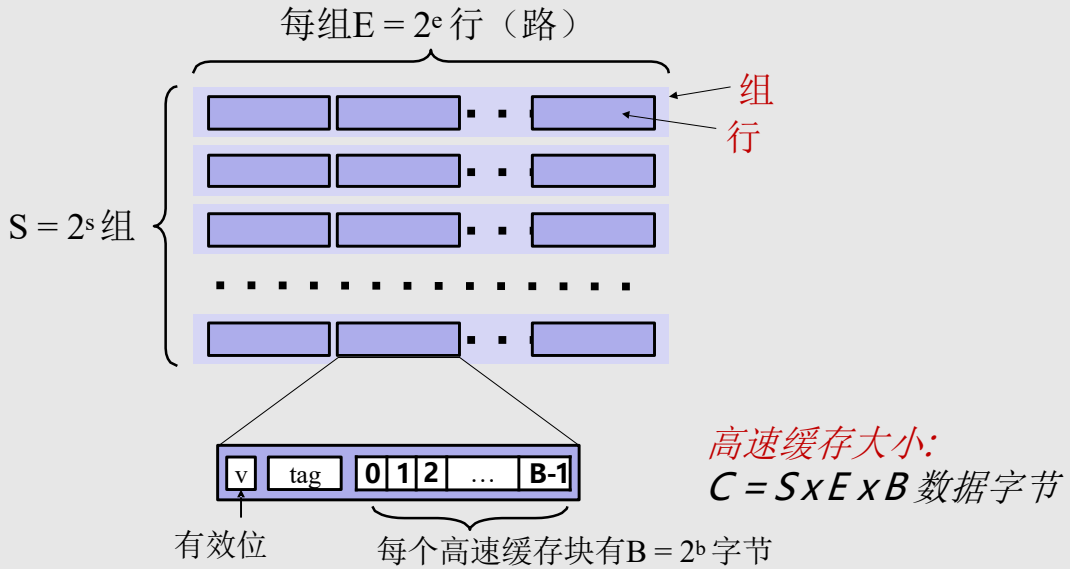
47 位地址范围

B =

S =, s =

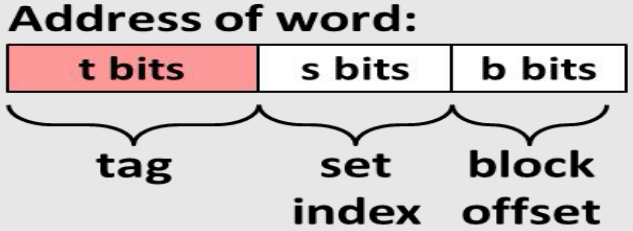
E =, e =

C =



高速缓存大小:
C = S x E x B 数据字节

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111



块偏移: 位

组索引: 位

标记: 位

栈地址:

0x00007f7262a1e010

块偏移: 0x??

组索引: 0x??

标记: 0x??

例子: Core i7 L1 数据缓存

32 kB 的8路组相联

64 字节/块

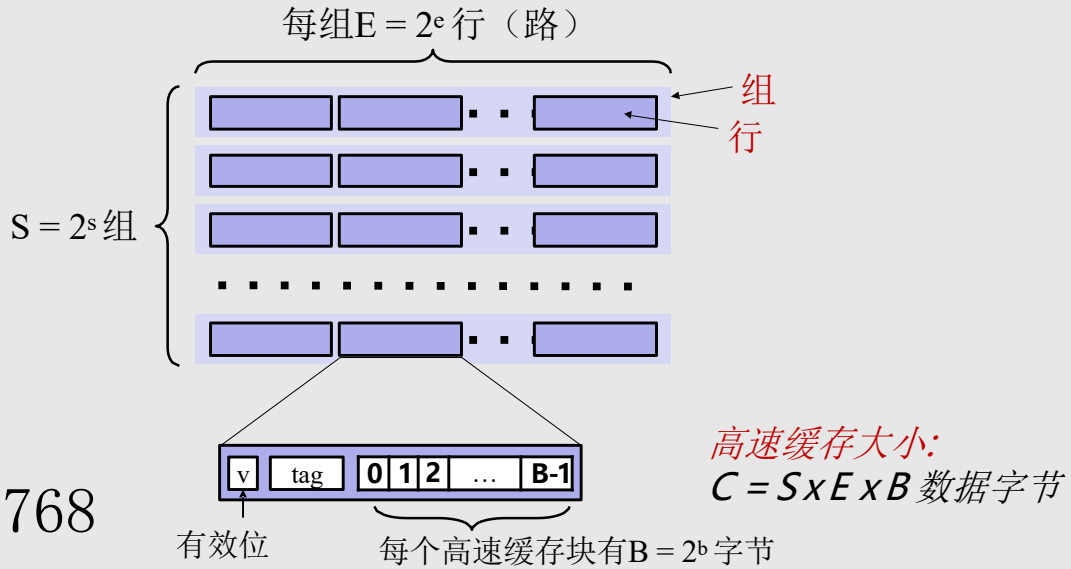
47 位地址范围

B = 64

S = 64, s = 6

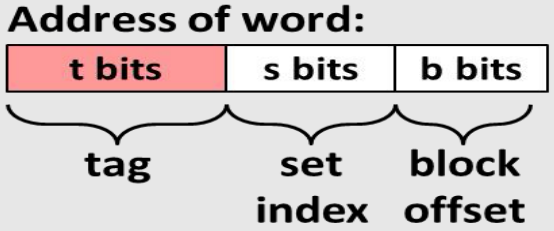
E = 8, e = 3

C = 64 x 64 x 8 = 32,768



高速缓存大小:
C = S x E x B 数据字节

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111



块偏移: 6 bits

组索引: 6 bits

标记: 35 bits

栈地址:

0x00007f7262a1e010

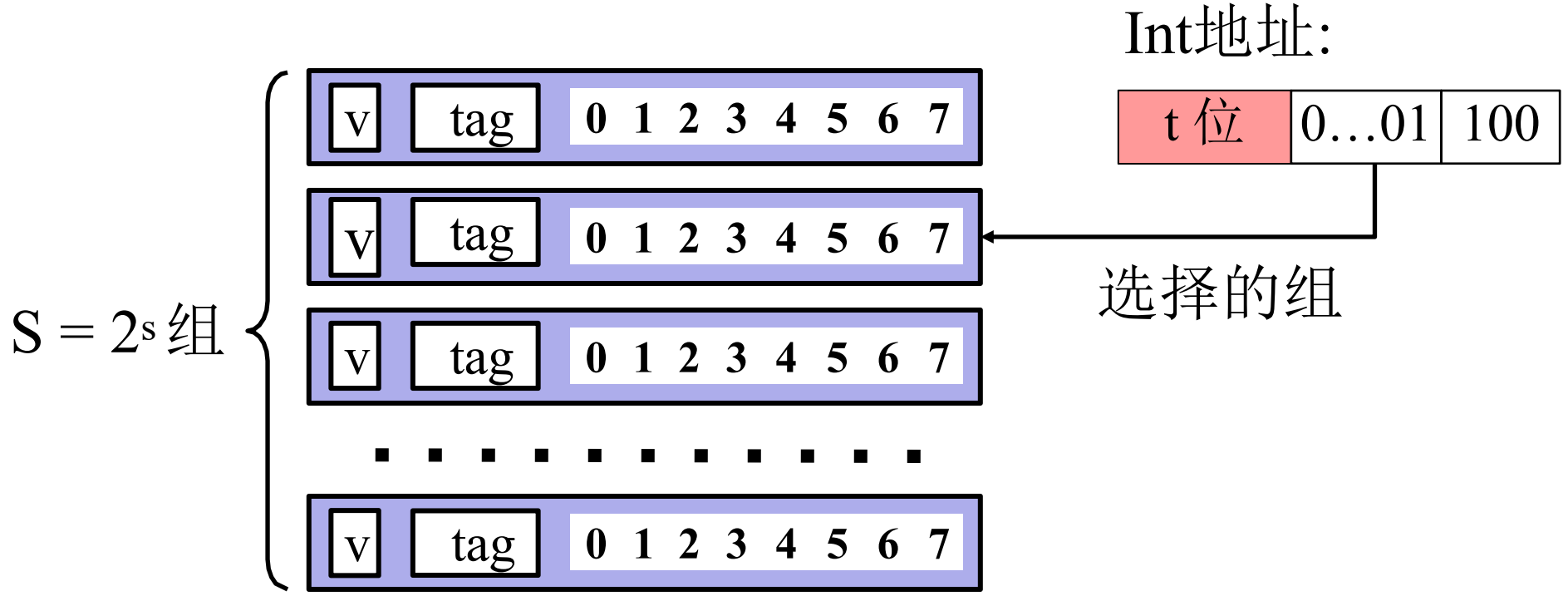
块偏移: 0x10

组索引: 0x0

标记: 0x7f7262a1e

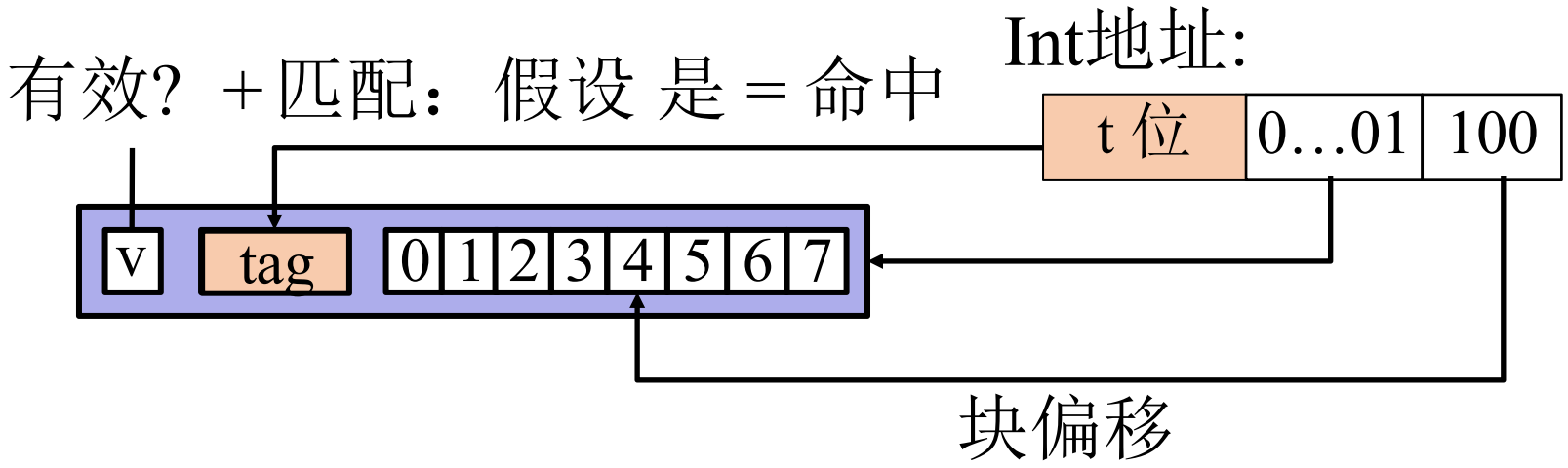
示例：直接映射高速缓存 (E = 1)

- 直接映射: 每一组只有一行
- 假设: 缓存块大小为8字节



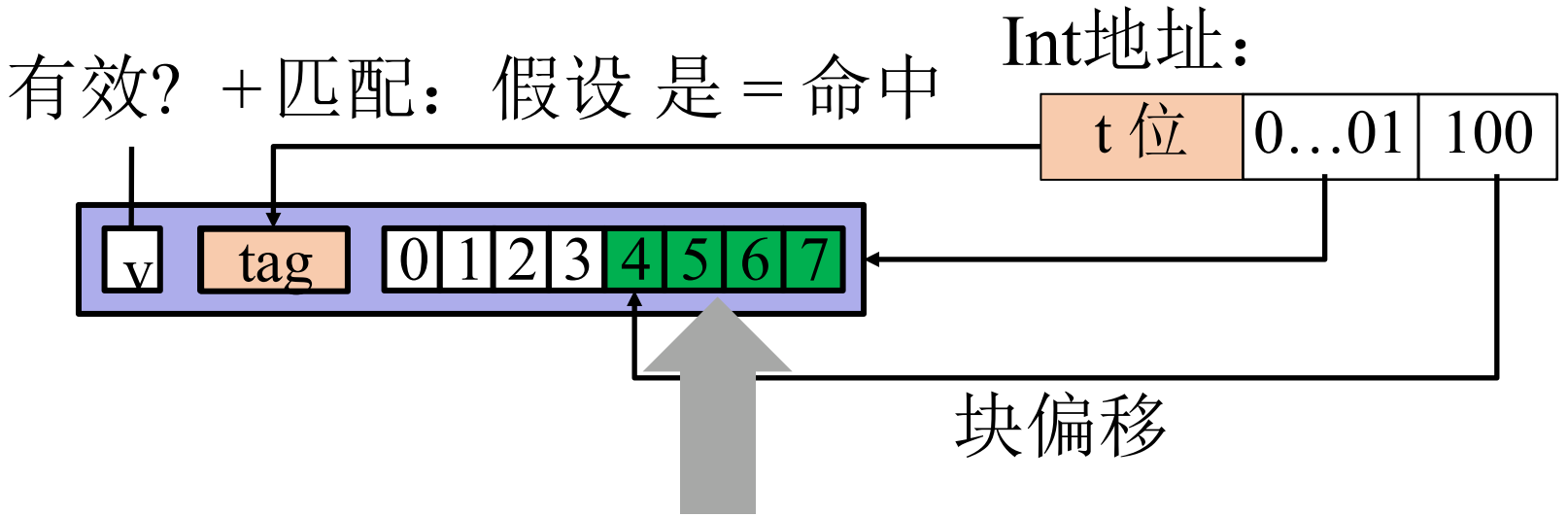
示例：直接映射高速缓存 (E = 1)

- 直接映射: 每一组只有一行
- 假设: 缓存块大小为8字节



示例：直接映射高速缓存 (E = 1)

- 直接映射: 每一组只有一行
- 假设: 缓存块大小为8字节



如果标记不匹配: 旧的行被驱逐和替换

直接映射高速缓存模拟

- 抖动：频繁间距访问的数据映射到同一个Cache 组。
- 优化：组相联。

M=16 字节 (4-位 地址), B=2 字节/块, S=4 组, E=1 块/组

t=1	s=2	b=1
x	xx	x

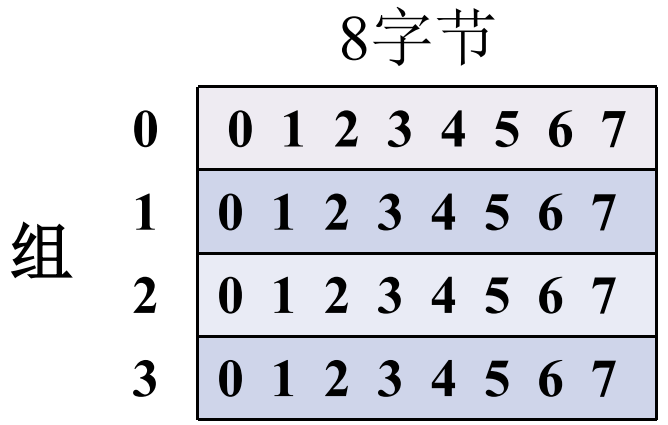
地址跟踪(读, 每读一个字节):

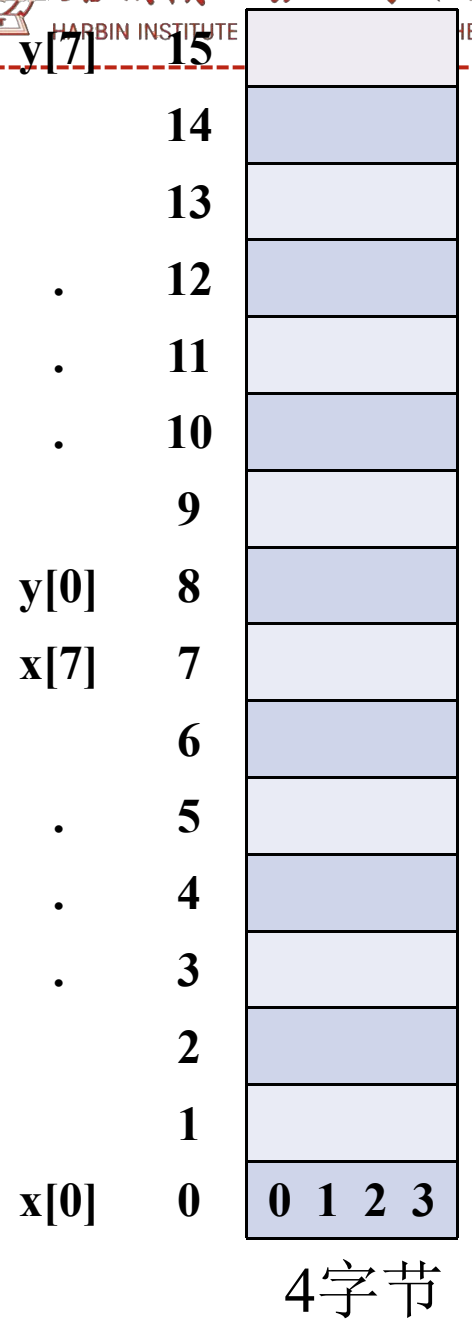
0 [0000_2], 无冲突不命中
1 [0001_2], 命中
7 [0111_2], 无冲突不命中
8 [1000_2], 冲突不命中
0 [0000_2] 冲突不命中

	v	Tag	Block
组 0	1	0	M[0-1]
组 1			
组 2			
组 3	1	0	M[6-7]

直接映射高速缓存中的冲突不命中

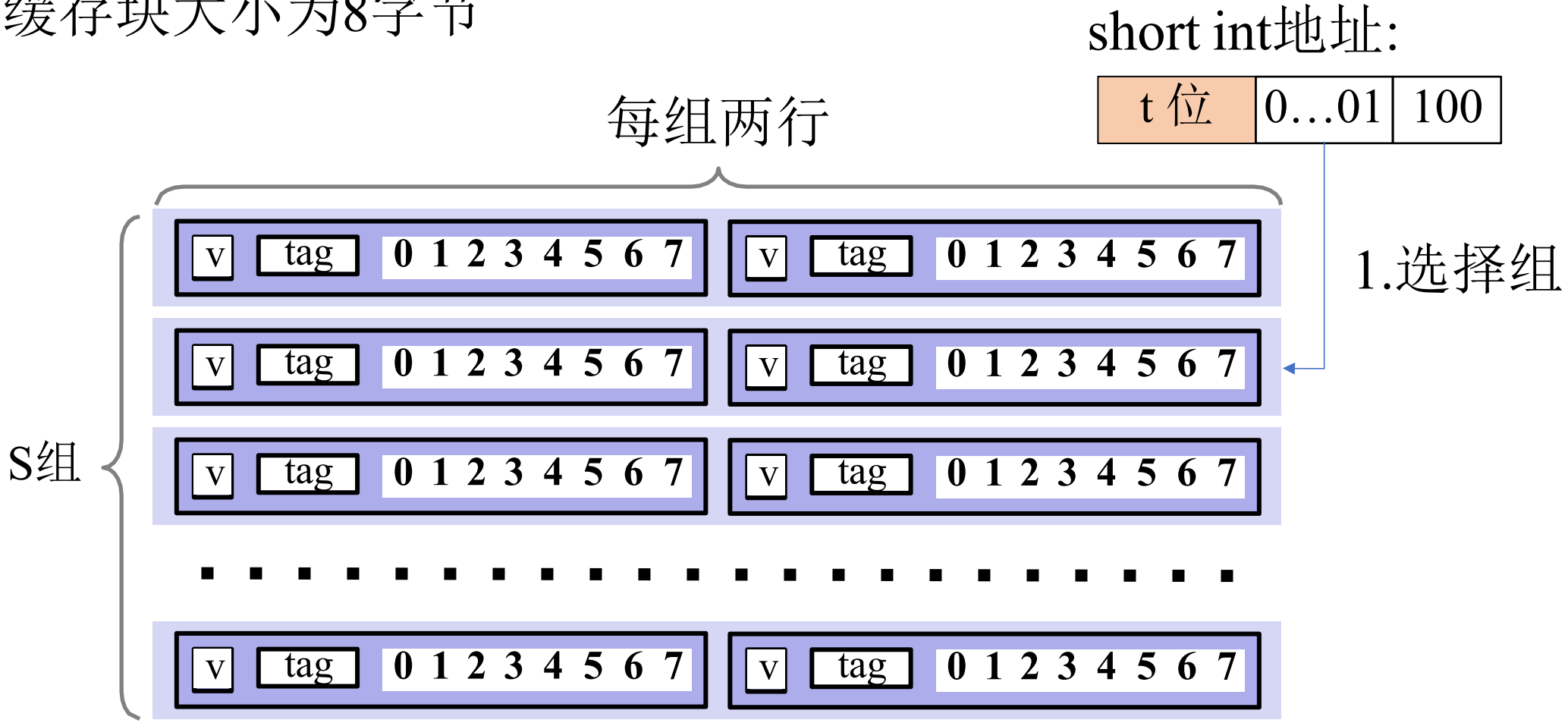

```
int dotpord(int x[8], int y[8]) //x与y向量的内积
{
    int sum = 0;
    int i;
    for(i = 0;i < 8;i++)
        sum += x[i] * y[i];
    return sum;
}
```





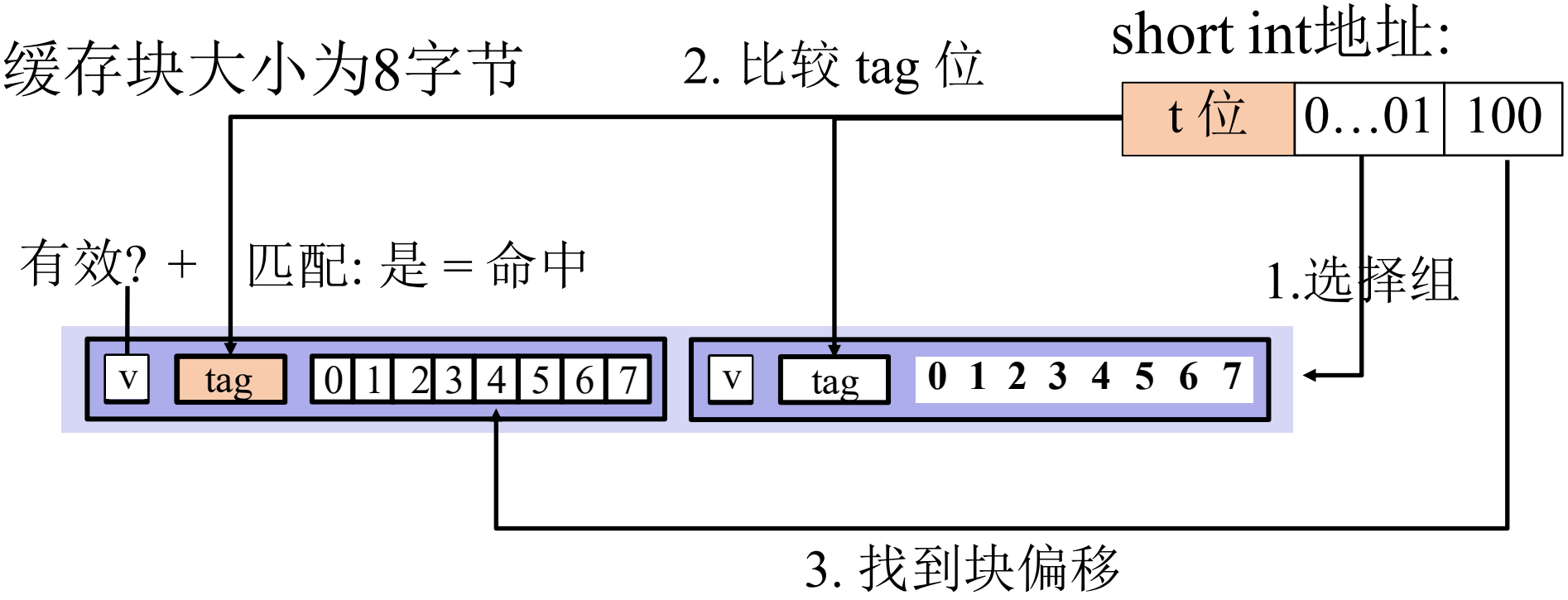
E-路组相联高速缓存 (E = 2)

- E = 2: 每组两行
- 假设:缓存块大小为8字节



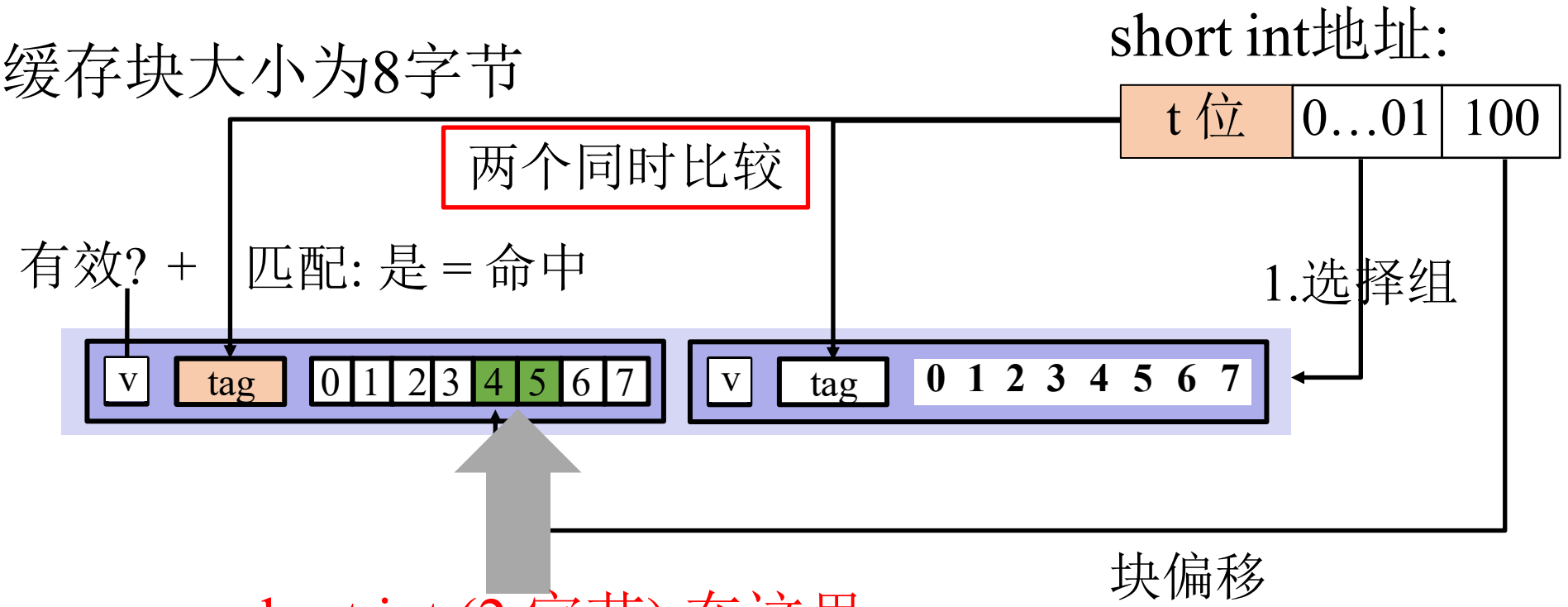
E-路组相联高速缓存 (E = 2)

- E = 2： 每组两行
- 假设： 缓存块大小为8字节



E-路组相联高速缓存 (E = 2)

- E = 2：每组两行
- 假设：缓存块大小为8字节



- 不匹配：
  - 在组中选择1行用于替换
  - 替换策略：随机、最不常使用LFU、最近最少使用(LRU)、 …

2-路 组相联缓存模拟

t=2　　s=1　b=1

xx	x	x

M=16 字节地址, B=2 字节/块, S=2 组, E=2 块/组

	v	Tag	Block
组0	1	00	M[0-1]
	1	10	M[8-9]
组1	1	01	M[6-7]
	0		

地址跟踪(读, 每读一个字节):

0　[0000$_2$],　无冲突不命中

1　[0001$_2$],　命中

7　[0111$_2$],　无冲突不命中

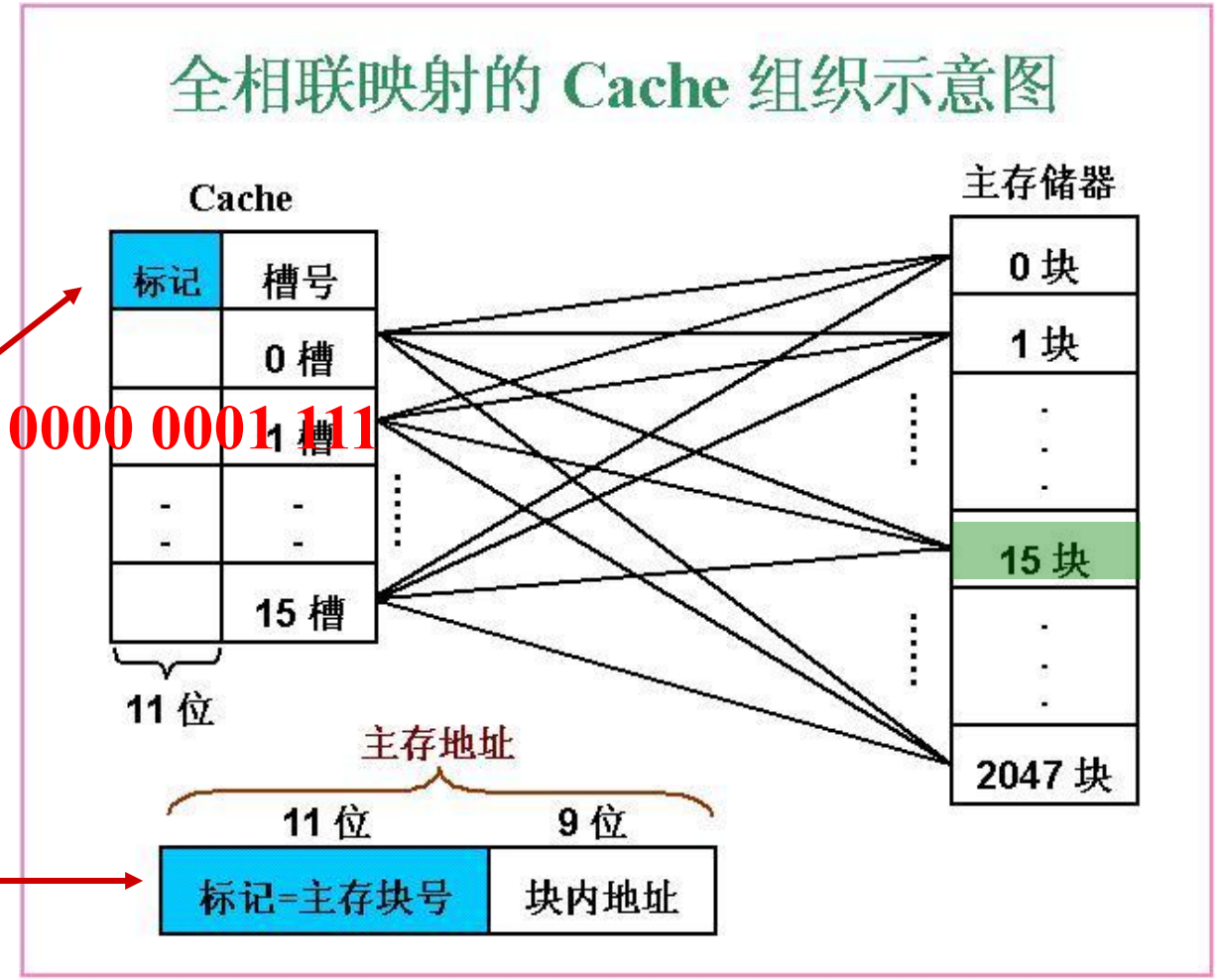
8　[1000$_2$],　冲突不命中

0　[0000$_2$]　命中

全相联映射Cache组织示意图

- 假定数据在主存和Cache间的传送单位为512字。
- Cache大小：2^{13}字=8K字=16行 × 512字/行
- 主存大小：2^{20}字=1024K字=2048块 × 512字/块
- Cache标记（tag）指出对应行取自哪个主存块
- 主存tag指出对应地址位于哪个主存块

每个主存块可装到Cache任一行中



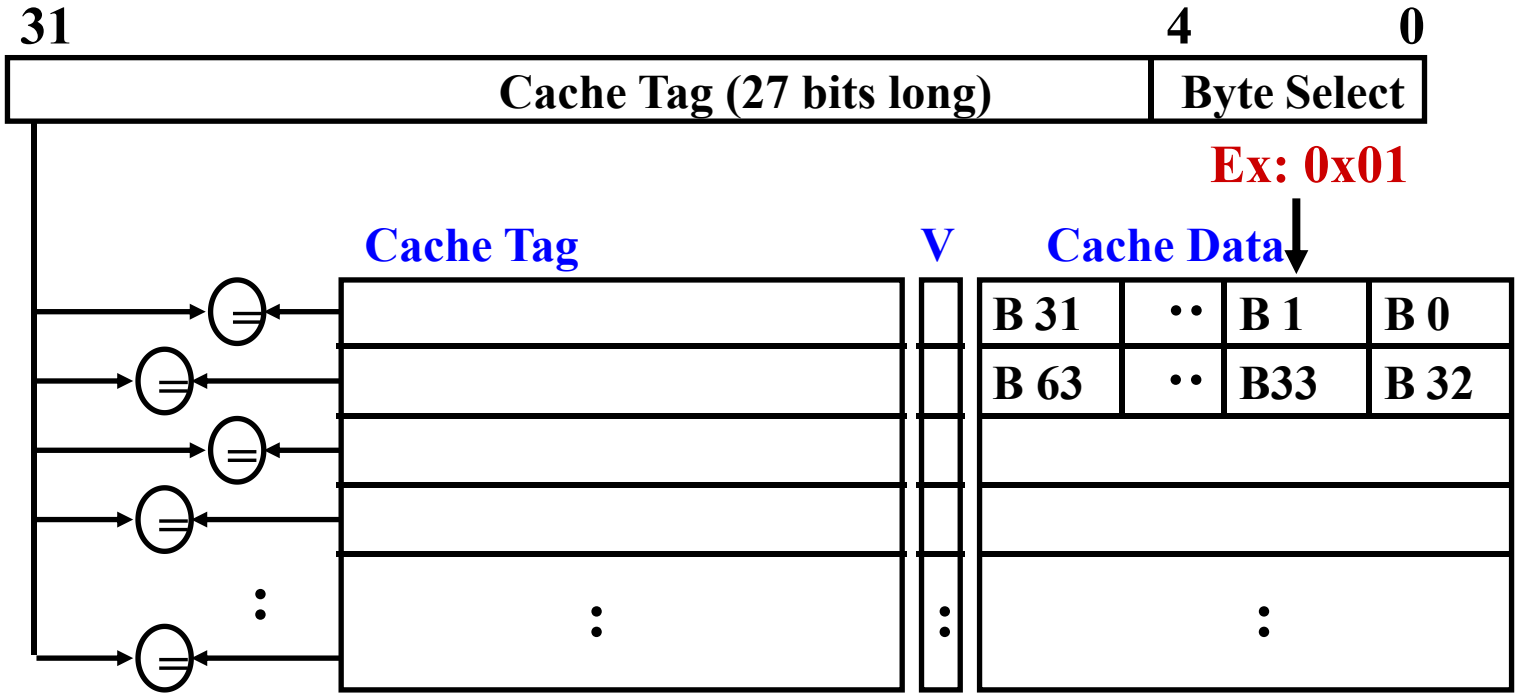
如何对0x01E0C单元进行访问？

0000 0001 1110 0000 1100 是第15块中的第12个单元！

如何实现按内容访问？ **直接比较！**

举例：全相联

- 全相联 Cache
  - 无需Cache索引，为什么？ 因为同时比较所有Cache项的标志
- 根据定义：冲突不命中率 = 0
  - 没有因tag冲突导致的缺失，因为只要有空闲Cache块，都不会发生冲突
- Example：32bits 内存地址, 32 B 块大小。
  - 比较器位数多长？
    需要 27-bit 大小的比较器
  - 需要多少个比较器？
    每行一个比较器！



Cache的组相联映射和直接映射的对比

- 直接映射
  - cache利用率很低
  - cache冲突率很高
  - 淘汰算法很简单
  - 应用场合：大容量cache

- 组相联映射
  - cache利用率较高
  - cache冲突率较低(减少了冲突不命中)
  - 淘汰算法较复杂
  - 应用场合：小容量cache

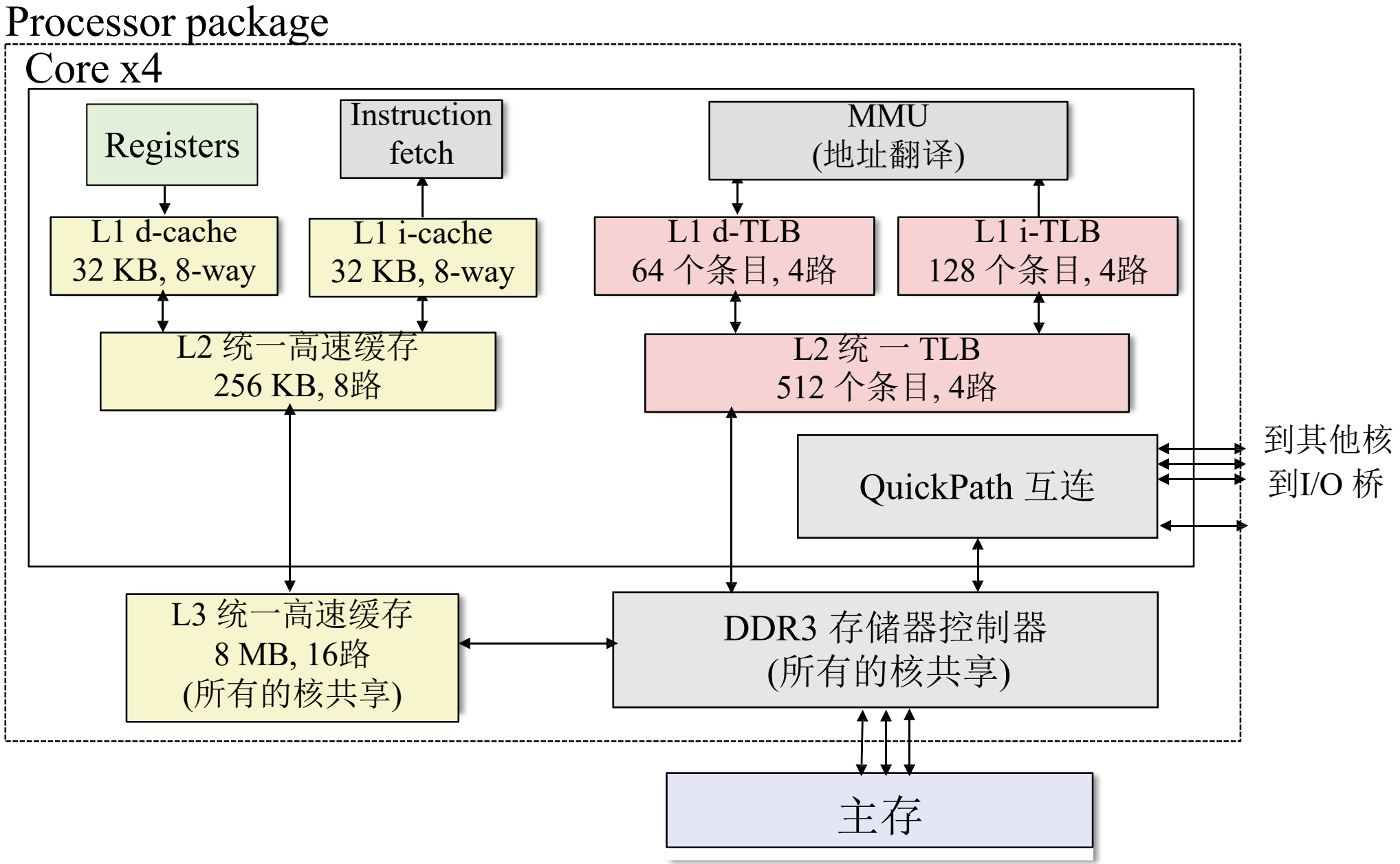
- 全相联映射(特殊的组相联映射)
  - cache利用率很高
  - cache冲突率为零(无冲突不命中)
  - 淘汰算法较复杂
  - 应用场合：小容量cache

- 全相联映射看似最好，但是全相联映射也有缺点：硬件结构很复杂，实现难度和价格都很高，因此实际应用中**往往采取组相联映射**。

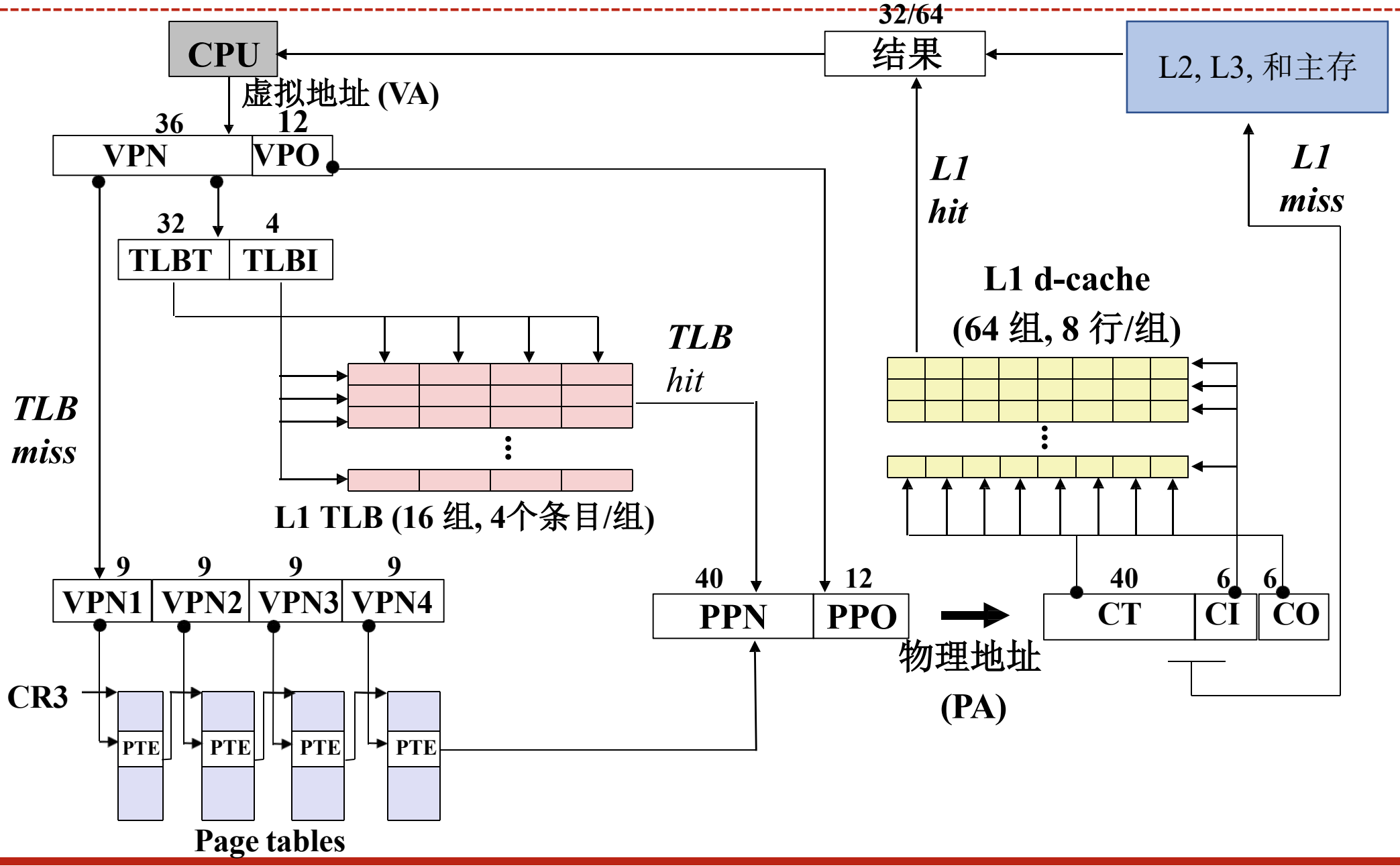
CSAPP<虚拟内存>串讲

- 虚拟地址的翻译
- 全相连/组相连/直接映射
- 案例研究: Core i7/Linux 内存系统
- 一个小内存系统示例
- 内存映射

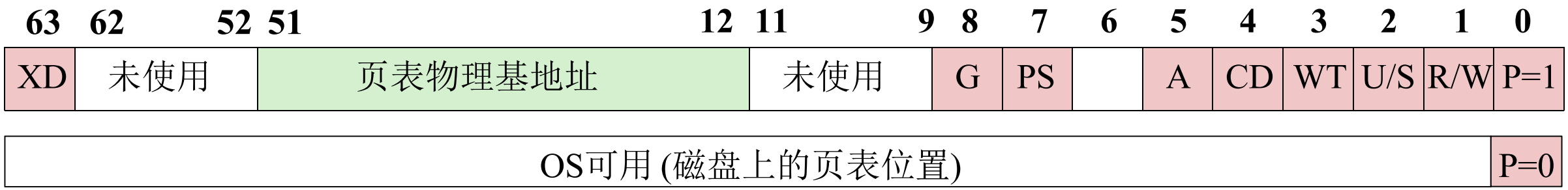
Intel Core i7 内存系统



Core i7 地址翻译


CPU
32/64
结果
L2, L3, 和主存
虚拟地址 (VA)
36
12
VPN
VPO
L1
hit
L1
miss
32
4
TLBT
TLBI
L1 d-cache
(64 组, 8 行/组)
TLB
hit
TLB
miss
L1 TLB (16 组, 4个条目/组)
9
9
9
9
VPN1
VPN2
VPN3
VPN4
40
12
PPN
PPO
40
6
6
CT
CI
CO
物理地址
(PA)
CR3
PTE
PTE
PTE
PTE
Page tables

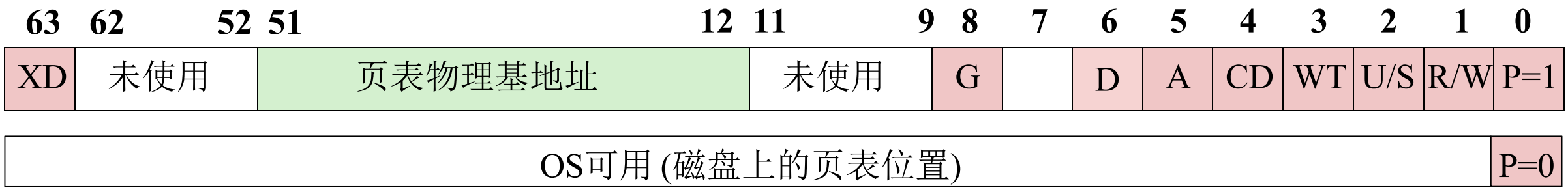
Core i7 1-3级页表条目格式



- 每个条目引用一个 4KB子页表:
  - P: 子页表在物理内存中 (1)不在 (0).
  - R/W: 对于所有可访问页，只读或者读写访问权限.
  - U/S: 对于所有可访问页，用户或超级用户 (内核)模式访问权限.
  - WT: 子页表的直写或写回缓存策略.
  - A: 引用位 (由MMU 在读或写时设置，由软件清除).
  - PS: 页大小为4 KB 或 4 MB (只对第一层PTE定义).
  - Page table physical base address: 子页表的物理基地址的最高40位 (强制页表 4KB 对齐)
  - XD: 能/不能从这个PTE可访问的所有页中取指令.

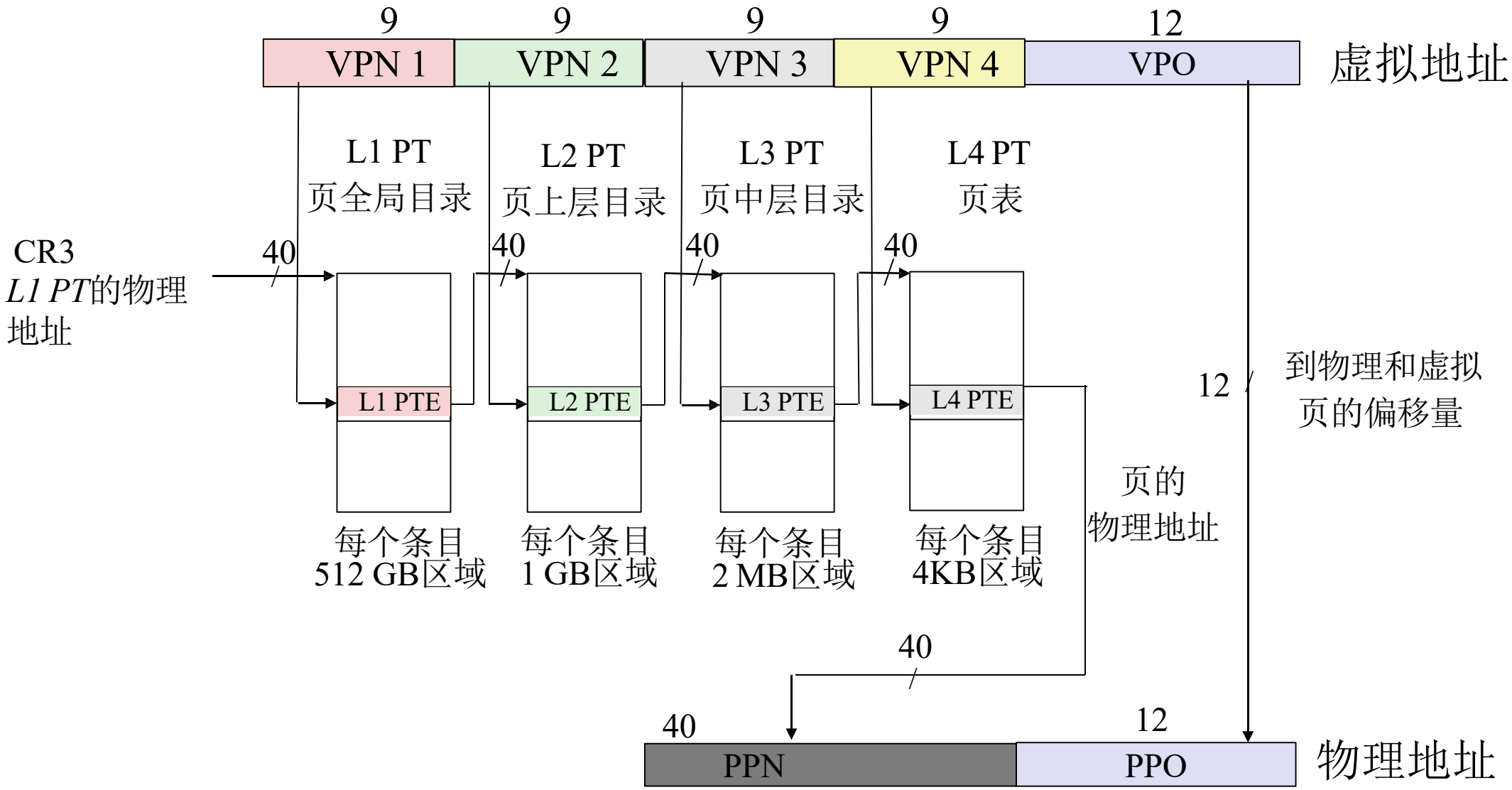
对照csapp
参考书p578

Core i7 第 4 级页表条目格式



- 每个条目引用一个 4KB子页表:
  - P: 子页表在物理内存中 (1)不在 (0).
  - R/W: 对于所有可访问页，只读或者读写访问权限.
  - U/S: 对于所有可访问页，用户或超级用户 (内核)模式访问权限.
  - WT: 子页表的直写或写回缓存策略.
  - A: 引用位 (由MMU 在读或写时设置，由软件清除).
  - D: 修改位 (由MMU 在读和写时设置，由软件清除)
  - Page table physical base address: 子页表的物理基地址的最高40位 (强制页表 4KB 对齐)
  - XD: 能/不能从这个PTE可访问的所有页中取指令.

Core i7 页表翻译

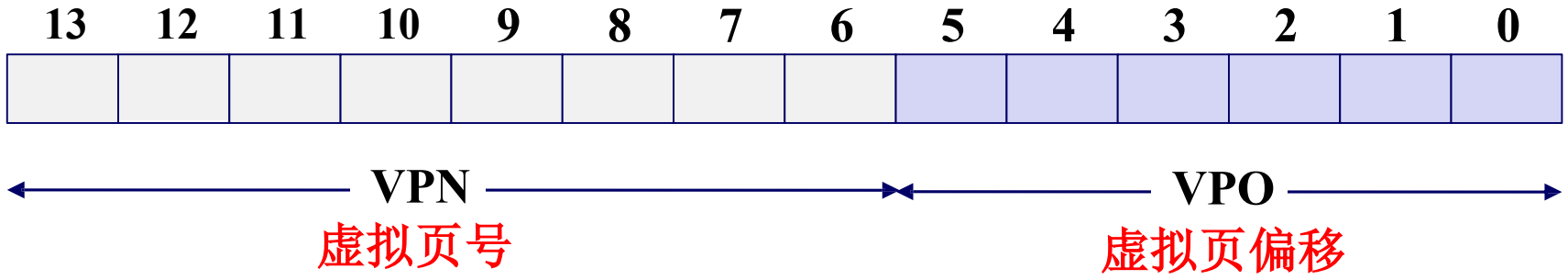

9
9
9
9
12
VPN 1
VPN 2
VPN 3
VPN 4
VPO
虚拟地址
L1 PT
页全局目录
L2 PT
页上层目录
L3 PT
页中层目录
L4 PT
页表
CR3
L1 PT的物理
地址
40
40
40
40
L1 PTE
L2 PTE
L3 PTE
L4 PTE
每个条目
512 GB区域
每个条目
1 GB区域
每个条目
2 MB区域
每个条目
4KB区域
12
到物理和虚拟
页的偏移量
页的
物理地址
40
40
12
PPN
PPO
物理地址

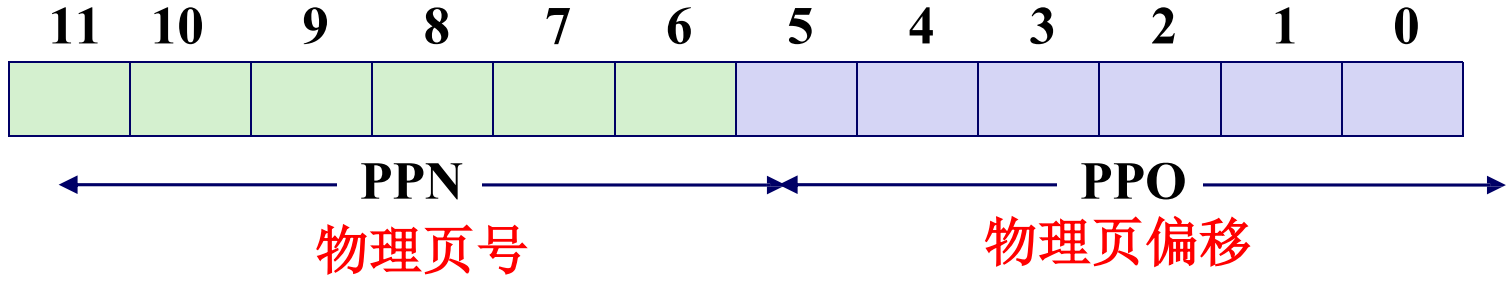
CSAPP<虚拟内存>串讲

- 虚拟地址的翻译
- 全相连/组相连/直接映射
- 案例研究: Core i7/Linux 内存系统
- 一个小内存系统示例
- 内存映射

一个小内存系统示例

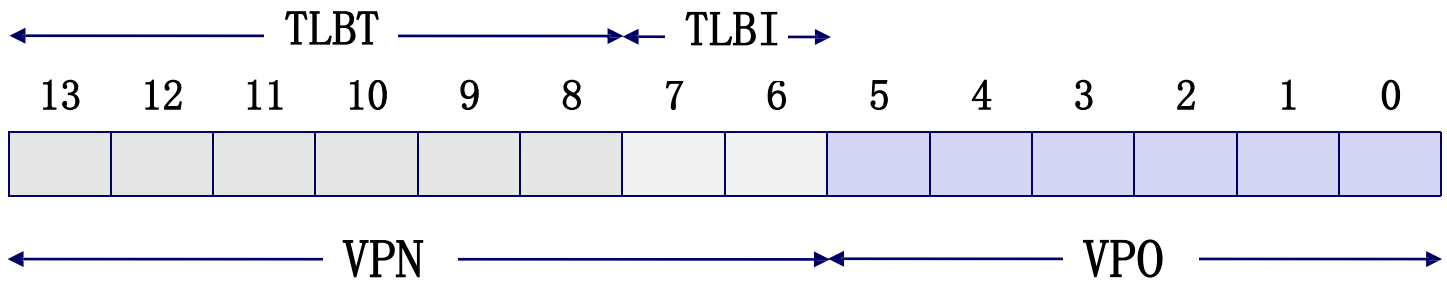
- 地址假设
  - 14位虚拟地址（n=14）
  - 12位物理地址（m = 12）
  - 页面大小64字节（P = 64）





1. 小内存系统的 TLB

- 16 entries 16个条目
- 4-way associative 4路组相联



组	标记	*PPN*	有效位	标记	*PPN*	有效位	标记	*PPN*	有效位	标记	*PPN*	有效位
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

2. 小内存系统的页表

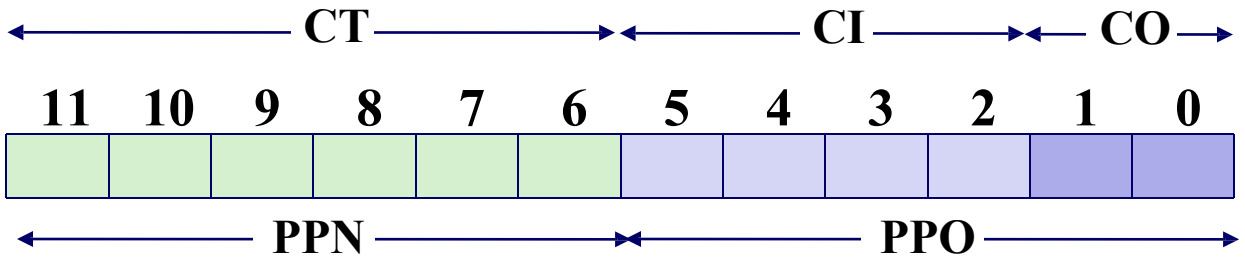
- 只展示了前16个PTE (out of 256)

VPN	*PPN*	有效位
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	*PPN*	有效位
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

3. 小内存系统的 Cache

- 16个组，每块为4字节
- 通过物理地址中的字段寻址
- 直接映射



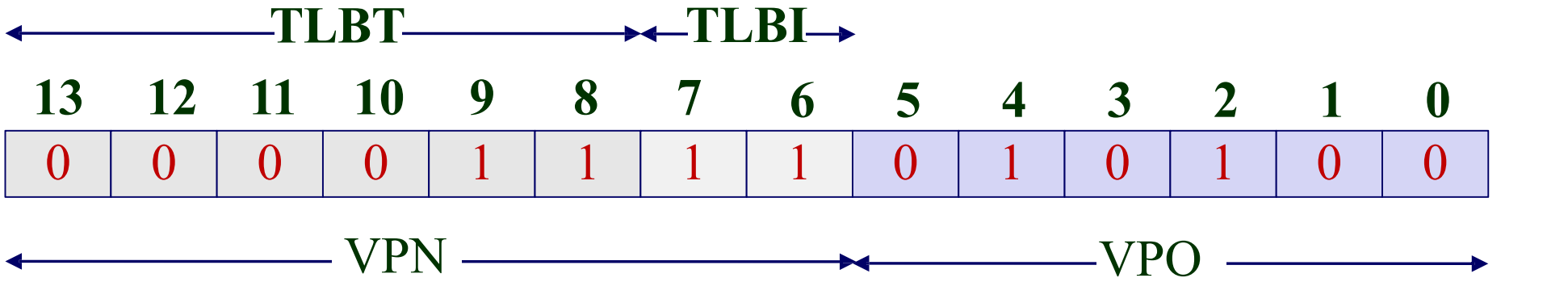
B

索引	标记位	有效位	块[0]	块[1]	块[2]	块[3]
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

索引	标记位	有效位	块[0]	块[1]	块[2]	块[3]
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

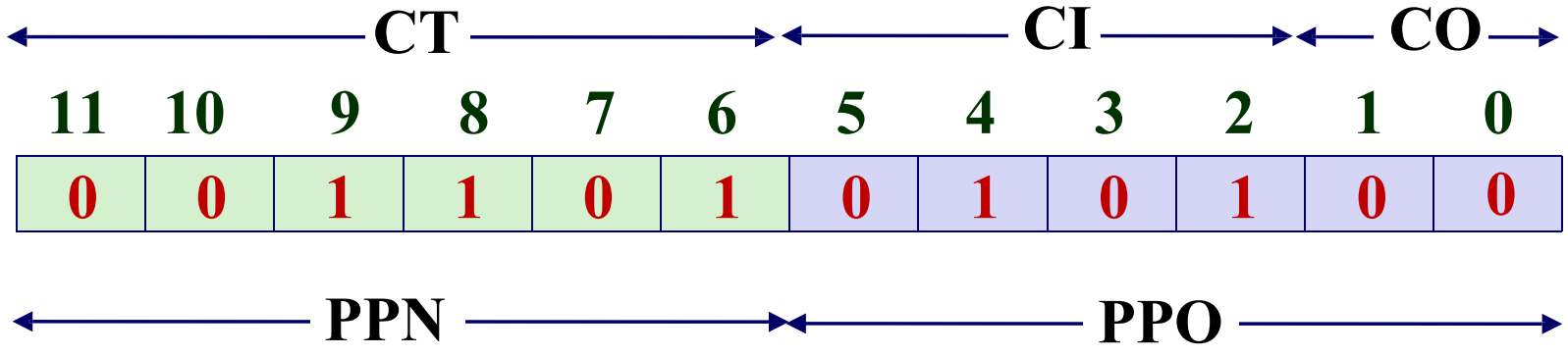
地址翻译 Example #1

- 虚拟地址: 0x03D4



VPN **0x0F** TLBI **0x3** TLBT **0x03** TLB Hit? **Y** Page Fault? **N** PPN **0x0D**

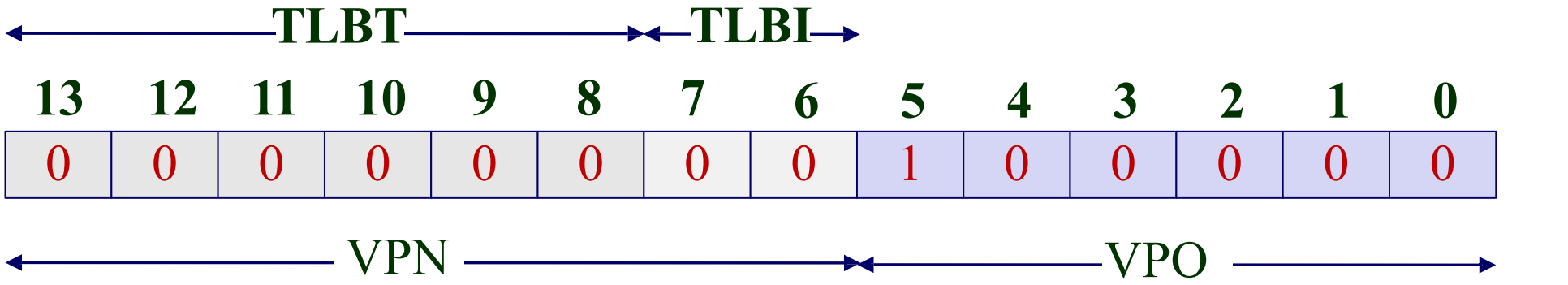
- 物理地址



CO **0** CI **0x5** CT **0x0D** Hit? **Y** Byte **0x36**

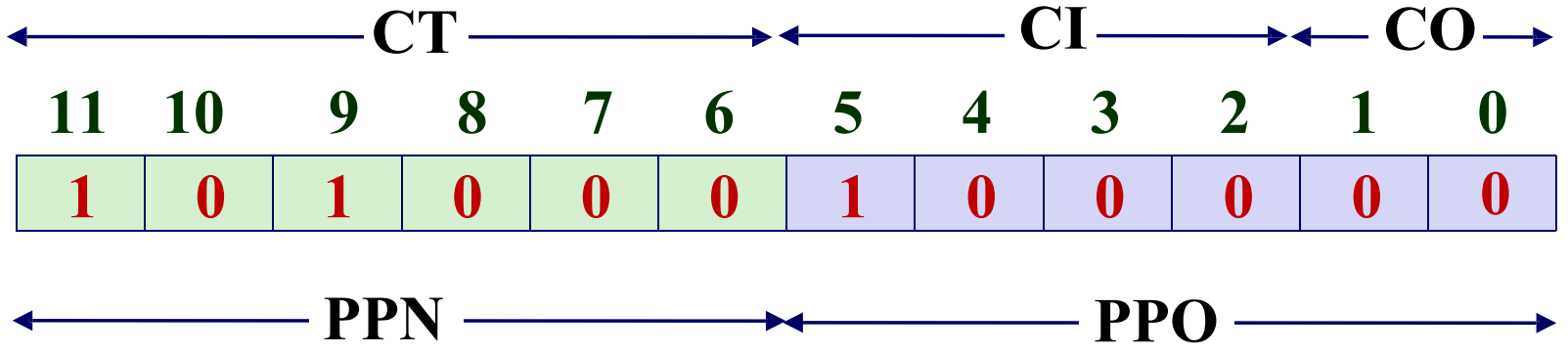
地址翻译 Example #2

- 虚拟地址: 0x0020



VPN **0x00** TLBI **0x0** TLBT **0x00** TLB Hit? **N** Page Fault? **N** PPN: **0x28**

- 物理地址

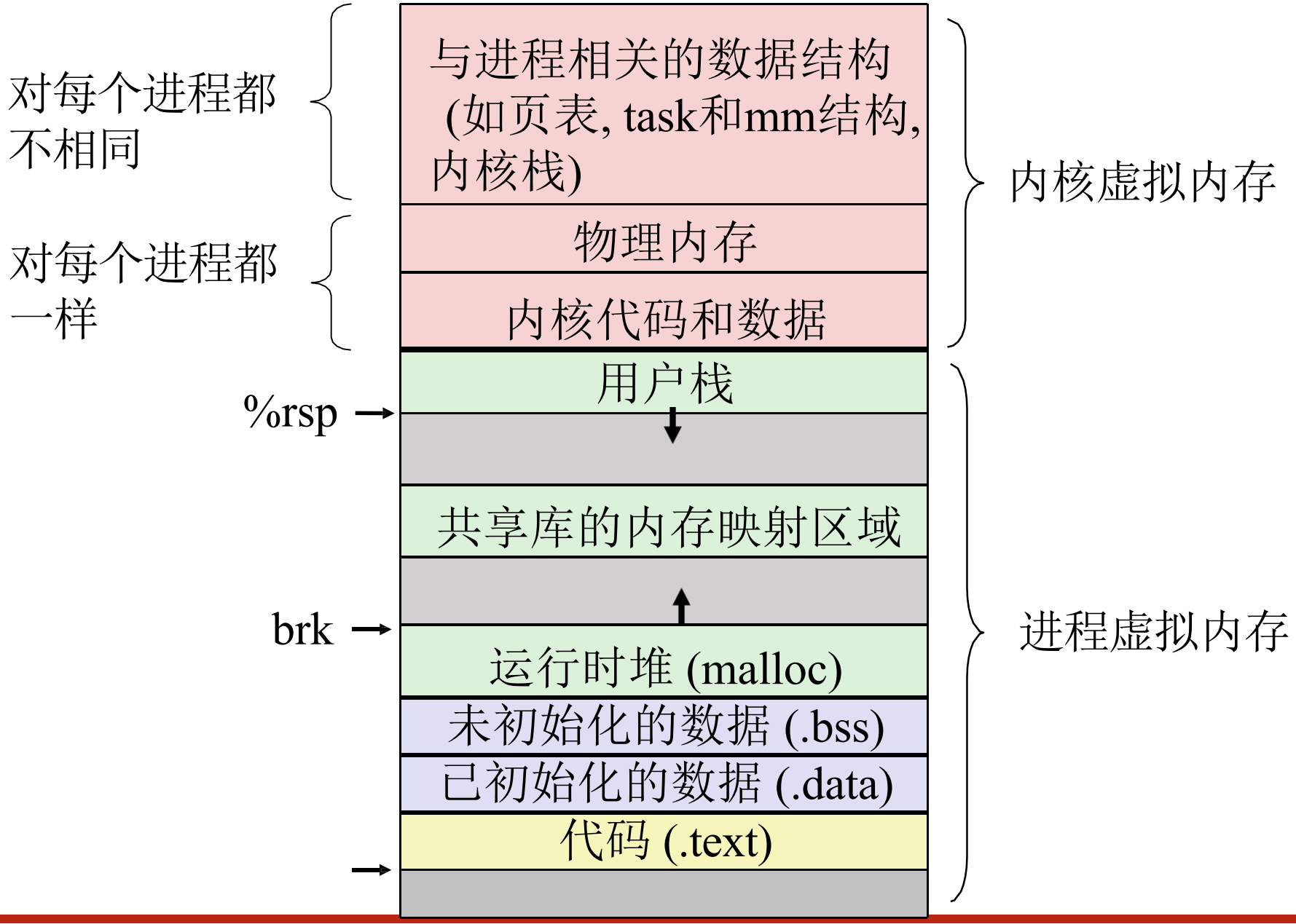


CO **0** CI **0x8** CT **0x28** Hit? **N** Byte: **0xMem**

CSAPP<虚拟内存>串讲

- 虚拟地址的翻译
- 全相连/组相连/直接映射
- 案例研究: Core i7/Linux 内存系统
- 一个小内存系统示例
- 内存映射

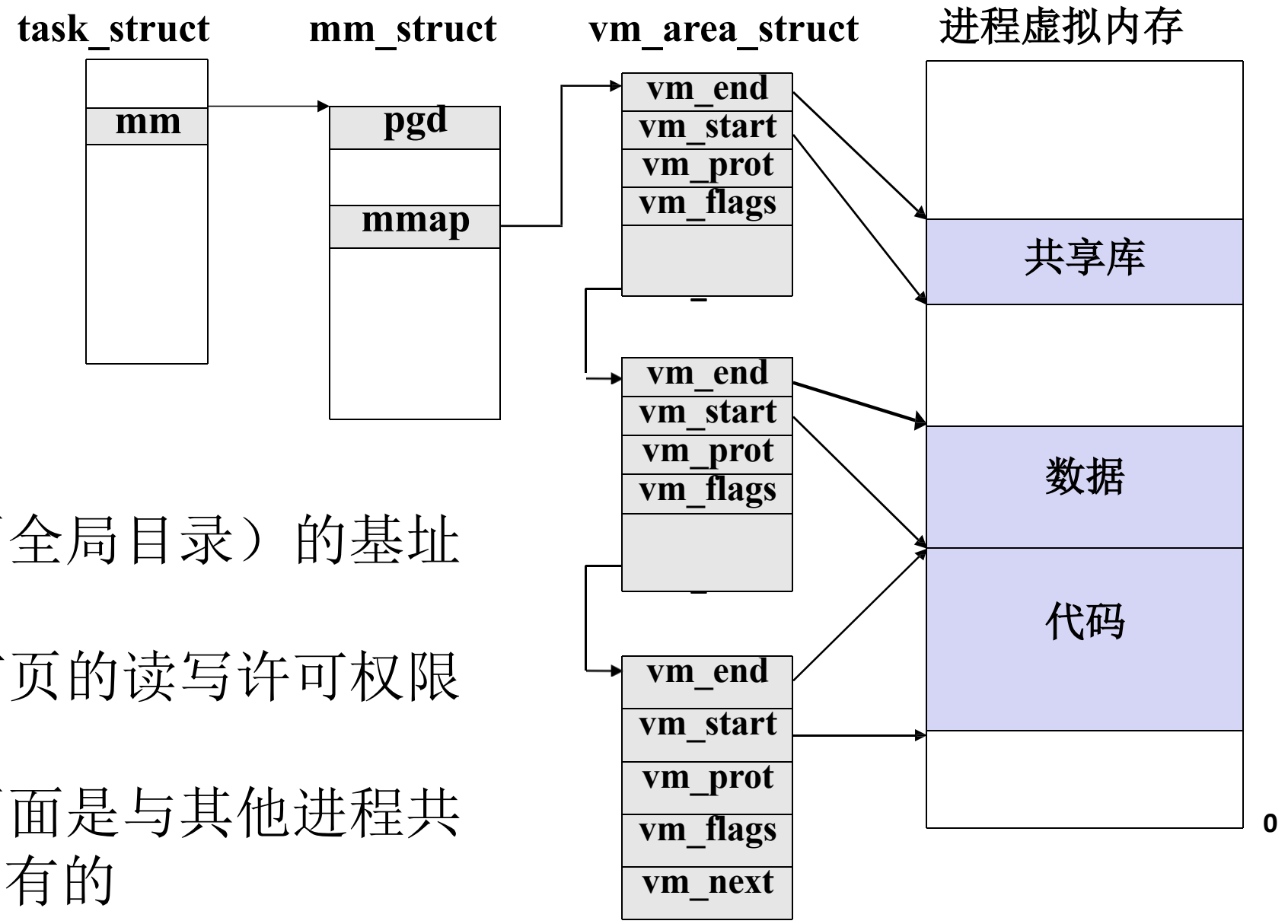
一个Linux 进程的虚拟地址空间



内存映射

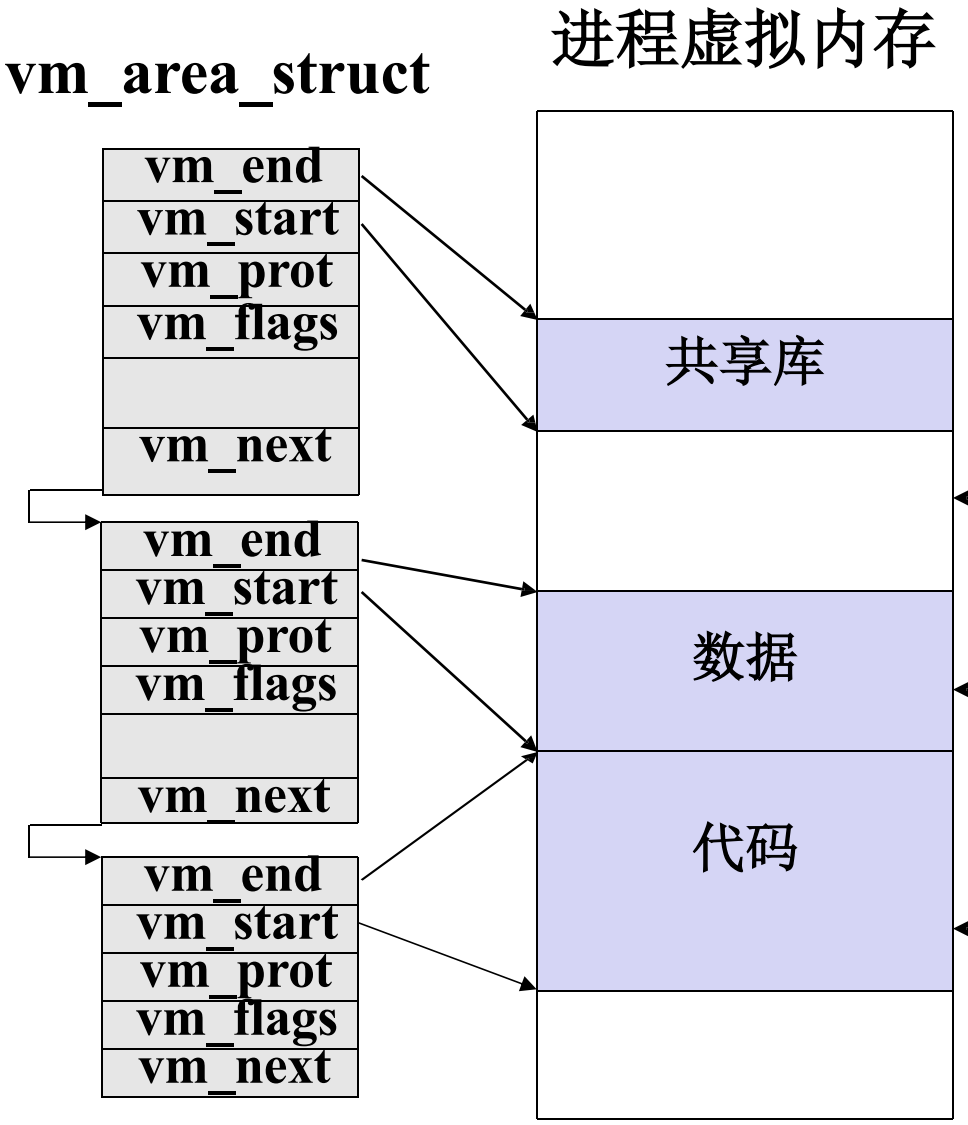
- Linux通过将虚拟内存区域与磁盘上的对象关联起来以初始化这个虚拟内存区域的内容
  - 这个过程称为内存映射（memory mapping）。
- 虚拟内存区域可以映射的对象（根据初始值的不同来源分）：
  - 磁盘上的普通文件（e.g.,一个可执行目标文件的.data，.text 段）
    - ✓文件区被分成页大小的片，对虚拟页面初始化（执行按需页面调度）
  - 匿名文件（内核创建，包含的全是二进制零，堆、栈、.bss段）
    - ✓CPU第一次引用该区域内的虚拟页面时会分配一个全是零的物理页 （demand-zero page请求二进制零的页）
    - ✓一旦该页面被修改，即和其他页面一样
- 初始化后的页面在内存和交换文件（ swap file）之间换来换去

Linux将虚拟内存组织成一些区域的集合



- pgd:
  - 指向第一级页表（页全局目录）的基址
- vm_prot:
  - 描述这个区域内所有页的读写许可权限
- vm_flags
  - 描述这个区域内的页面是与其他进程共享的还是这个进程私有的

Linux缺页处理




```
Segmentation fault
/bin/sh: error while loading shared libraries: ▪ F8▪$▪ET▪▪
cannot open shared object file: No such file or directory
Segmentation fault
Segmentation fault
```

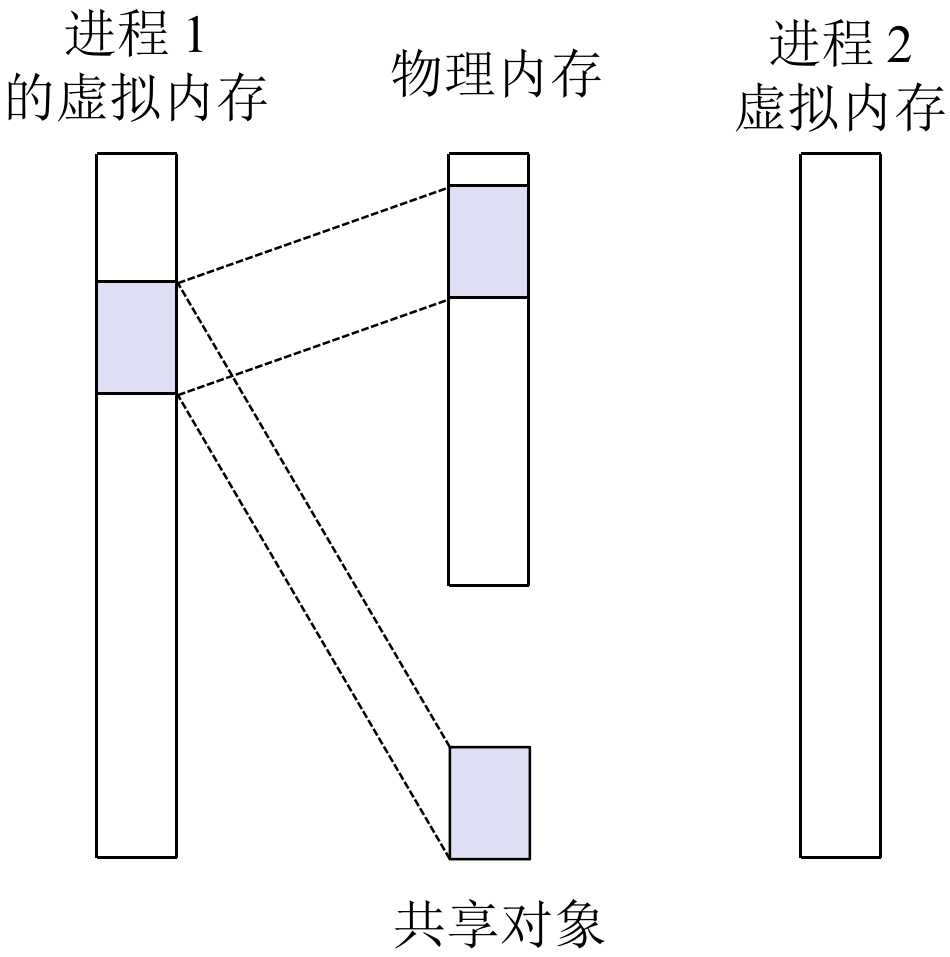
段错误:
访问一个不存在的页面

正常缺页

保护异常:
例如,违反许可，写一个只读的页面
(Linux 报告 Segmentation fault)

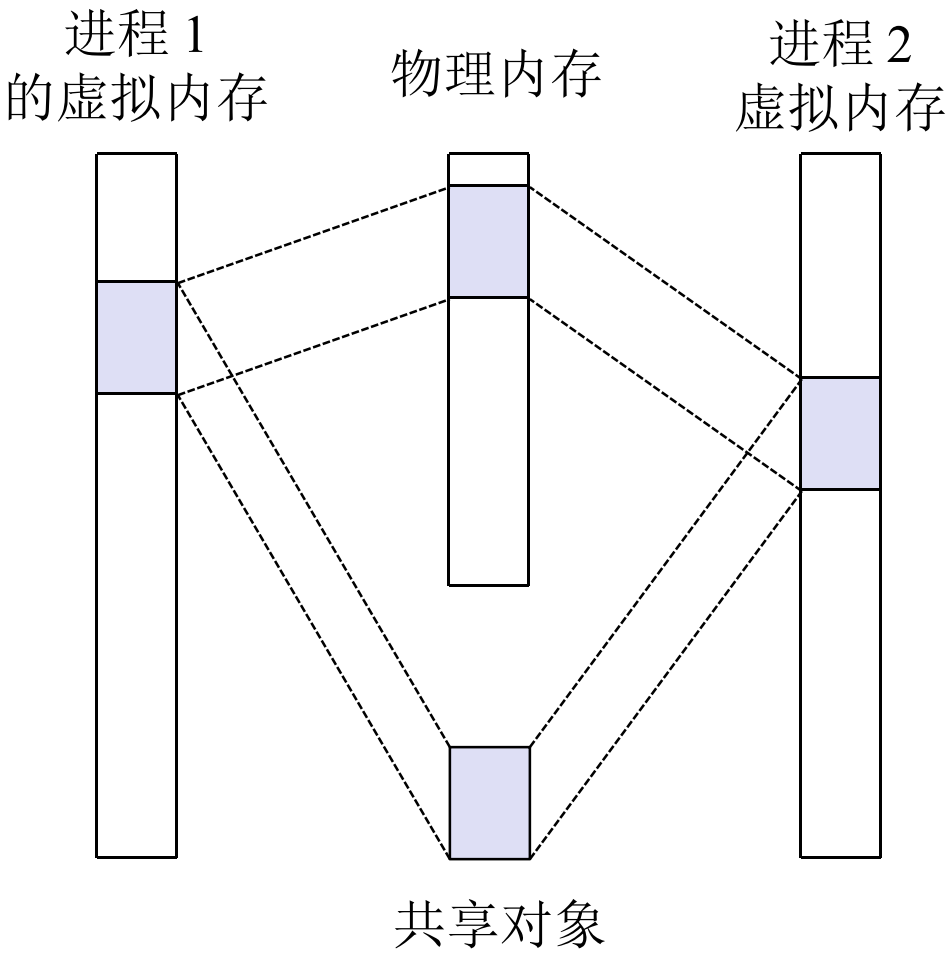
再看共享对象

- 一个对象被映射到虚拟内存的一个区域，要么作为共享对象，要么作为私有对象
  - 进程 1 映射了一个共享对象



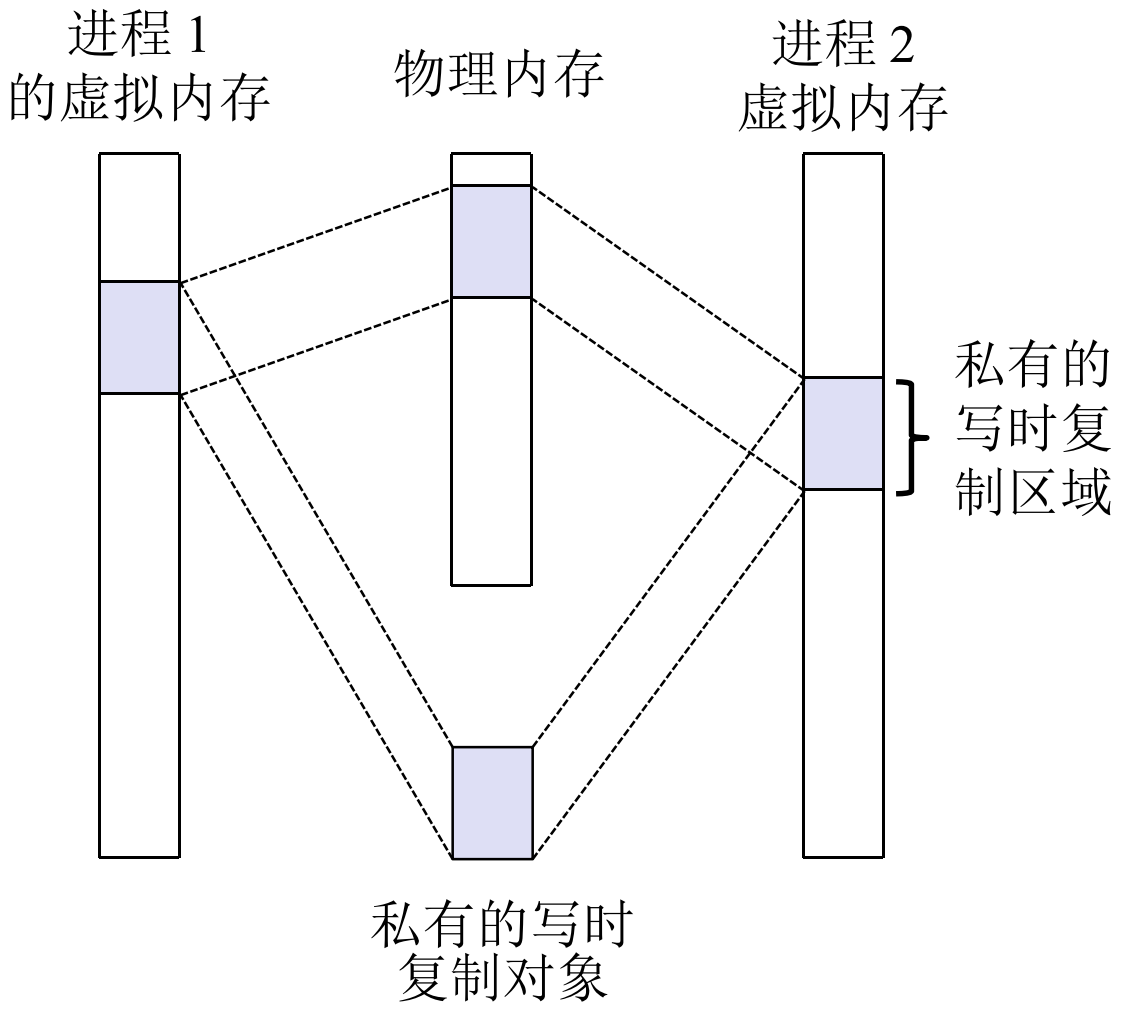
再看共享对象

- 进程 2 映射了同一个共享对象
- 两个进程的虚拟地址可以是不同的
- 物理内存中只有一个该共享对象的副本
- 进程1对共享对象的任何写操作（在进程1的虚拟内存区域中进行）对进程2都是可见的



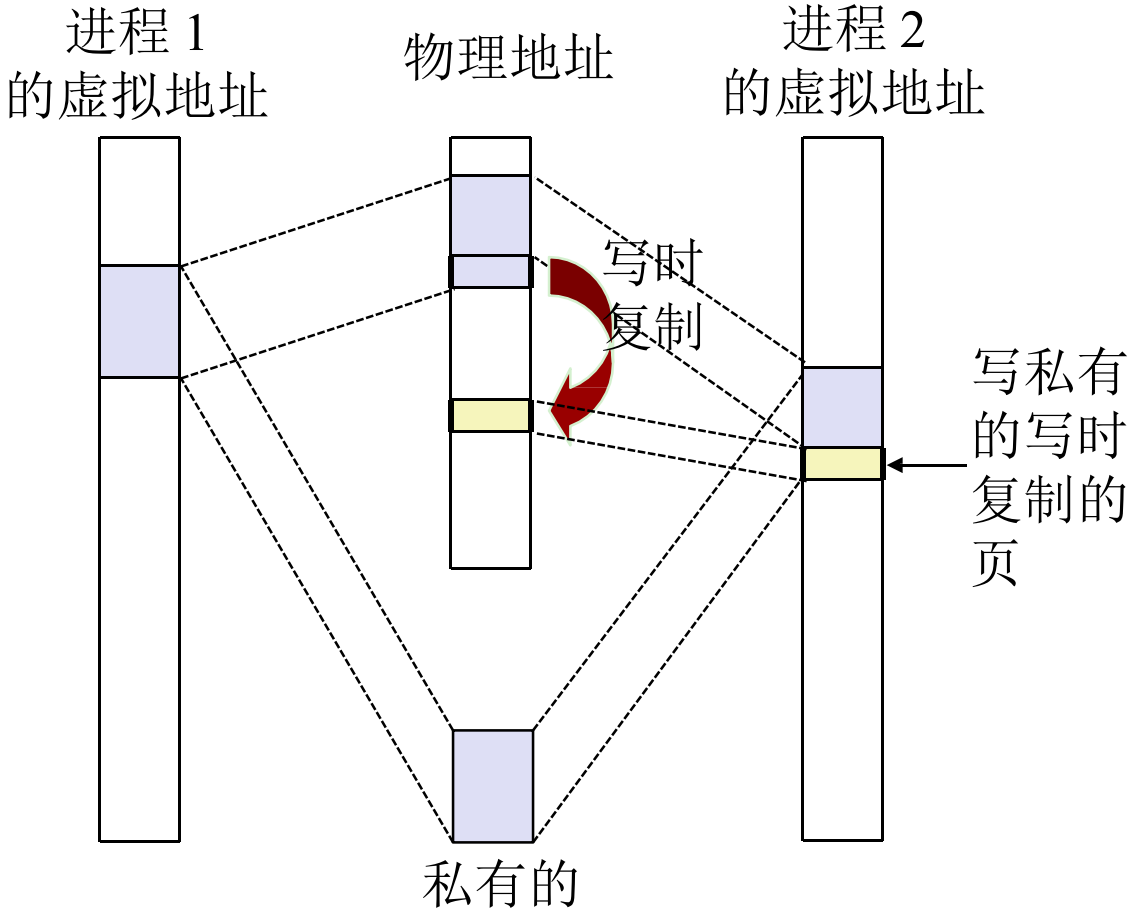
私有的写时复制（Copy-on-write）对象

- 两个进程都映射了私有的写时复制对象
- 区域结构被标记为私有的写时复制
- 私有区域的页表条目都被标记为只读



私有的写时复制（Copy-on-write）对象

- 写私有页的指令触发保护故障
- 故障处理程序创建这个页面的一个新副本
- 故障处理程序返回时重新执行写指令
- 尽可能地延迟拷贝（创建副本）

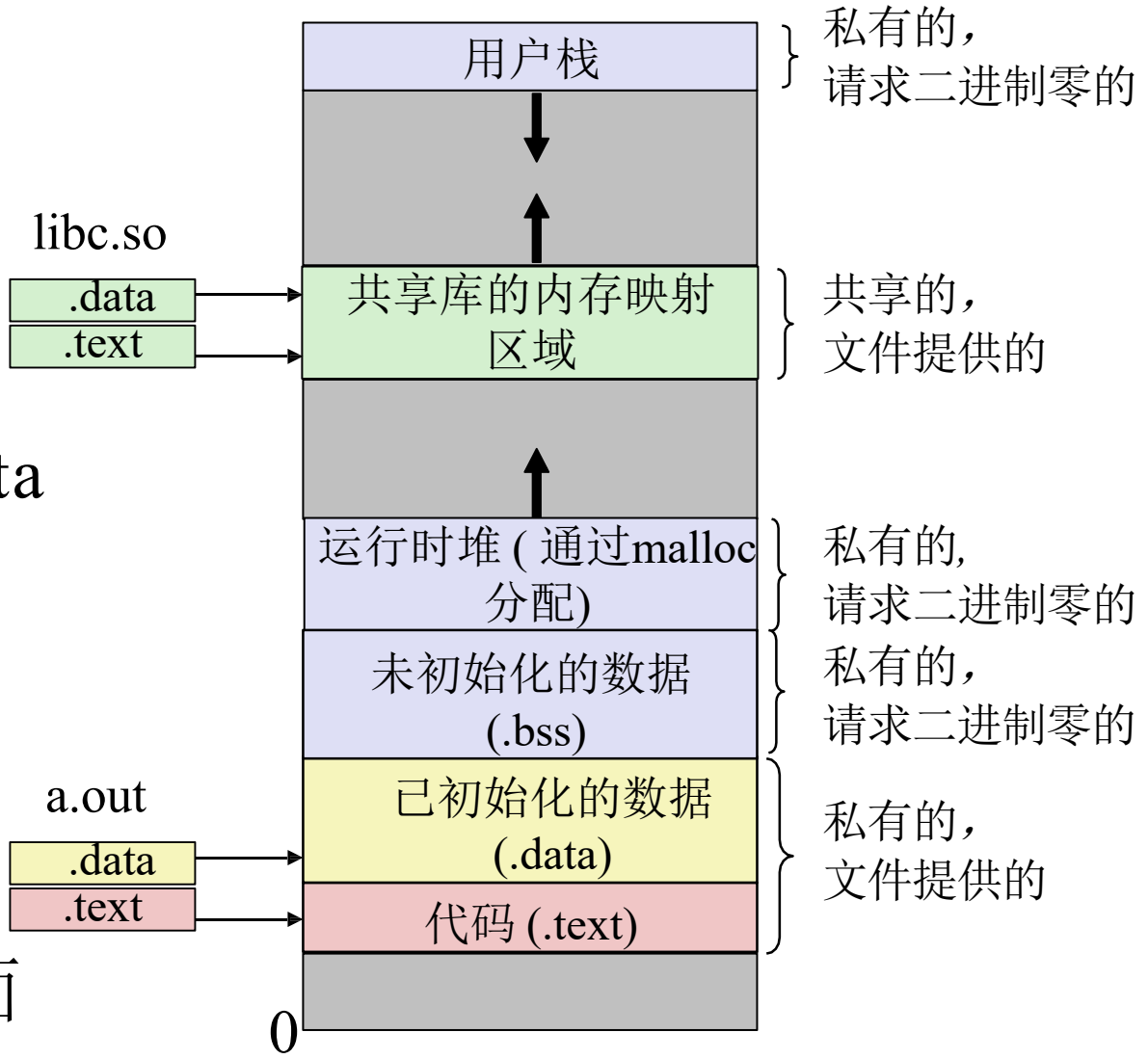


再看 fork 函数

- 虚拟内存和内存映射解释了fork函数如何为每个新进程提供私有的虚拟地址空间。
- 为新进程创建虚拟内存
  - 创建当前进程的的mm_struct, vm_area_struct和页表的原样副本
  - 两个进程中的每个页面都标记为只读
  - 两个进程中的每个区域结构（vm_area_struct）都标记为私有的
  - 写时复制（COW）
- 在新进程中返回时，新进程拥有与调用fork进程相同的虚拟内存
- 随后的写操作通过写时复制机制创建新页面

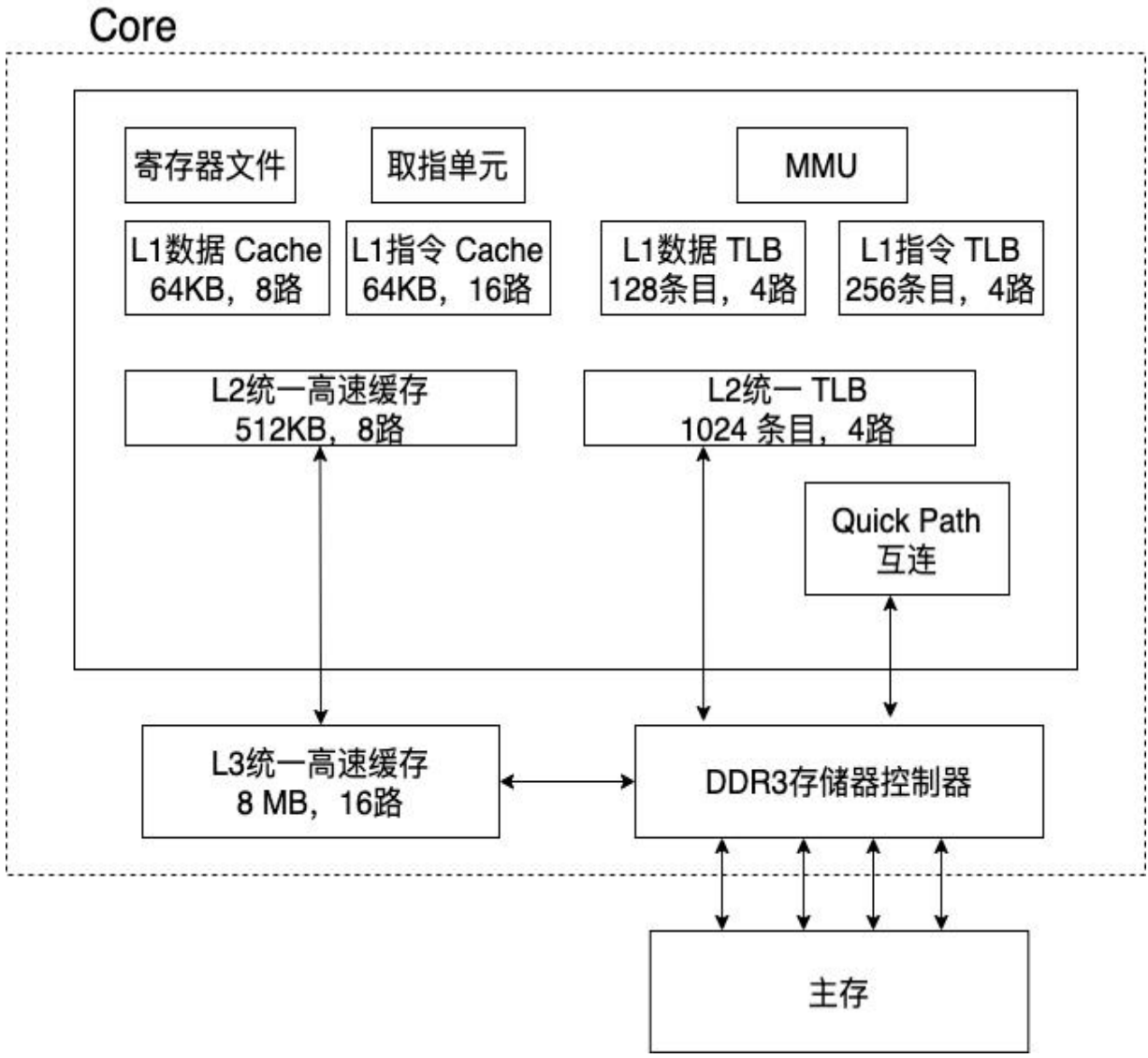
再看 execve 函数

- execve函数在当前进程中加载并运行新程序 a.out的步骤：
- 删除已存在的用户区域
- 创建新的区域结构
  - 代码和始化数据映射到.text和.data区（目标文件提供）
  - .bss和栈映射到匿名文件
- 设置PC，指向代码区域 的入口点
  - Linux根据需要换入代码和数据页面



随堂习题

- 某处理器的虚拟地址为 32 位。虚拟内存的页大小是 4KB，物理地址为 48 位，Cache 块大小为 32B。物理内存按照字节寻址。其内部结构如右图所示，依据这个结构，回答下面几个问题。



1. L1 数据 Cache 有多少组，相应的 Tag 位，组索引位和块内偏移位分别是多少？

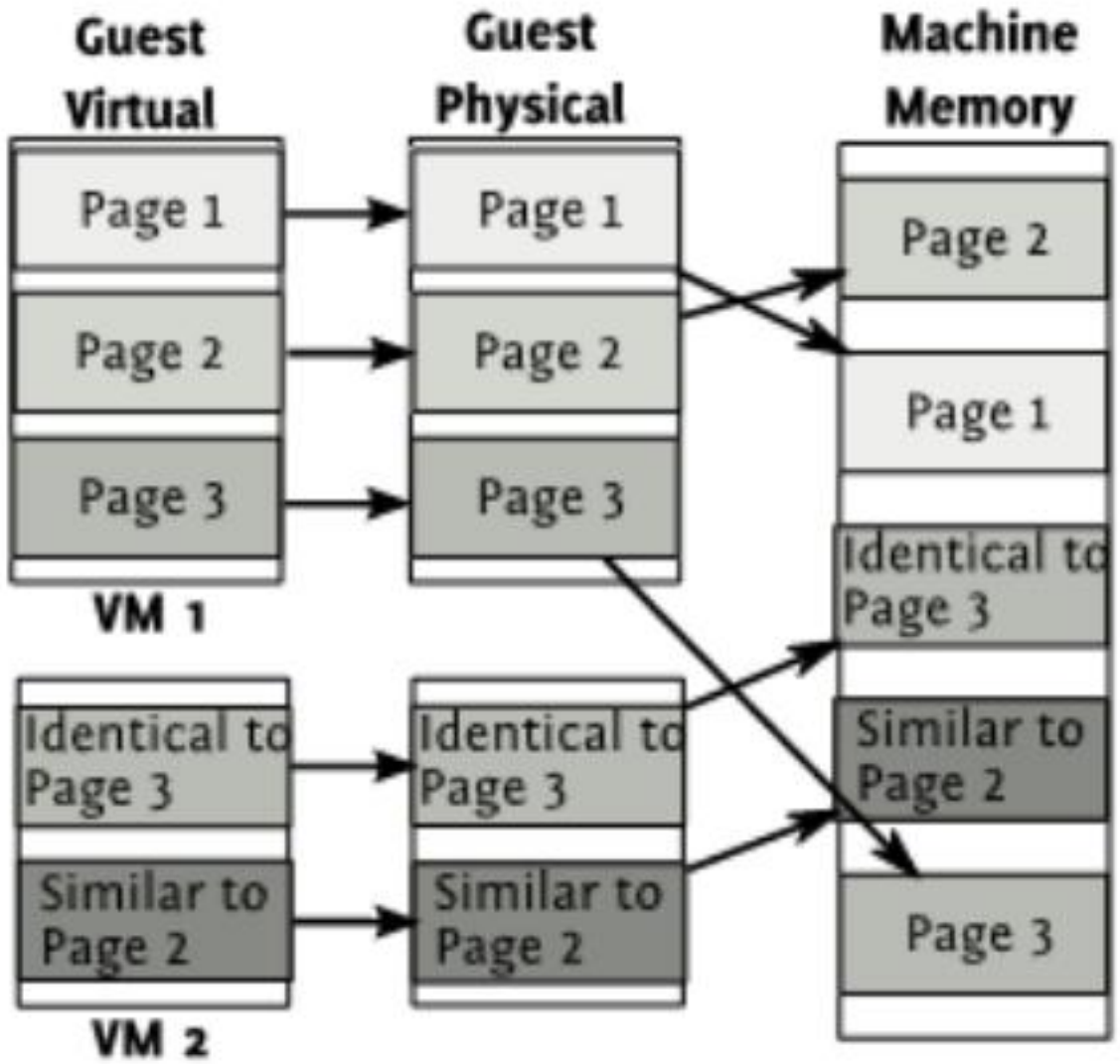
2. 对于某数据，其访问的虚拟地址为 0x829358B，则该地址对应的 VPO 为多少？对应的L1 TLBI 位为多少？（用 16进制表示）

3. 对于某指令，其访问的物理地址为 0x829358B，则该地址访问 L1 Cache 时，CT 位为多少？CO 位为多少？（用16进制表示）

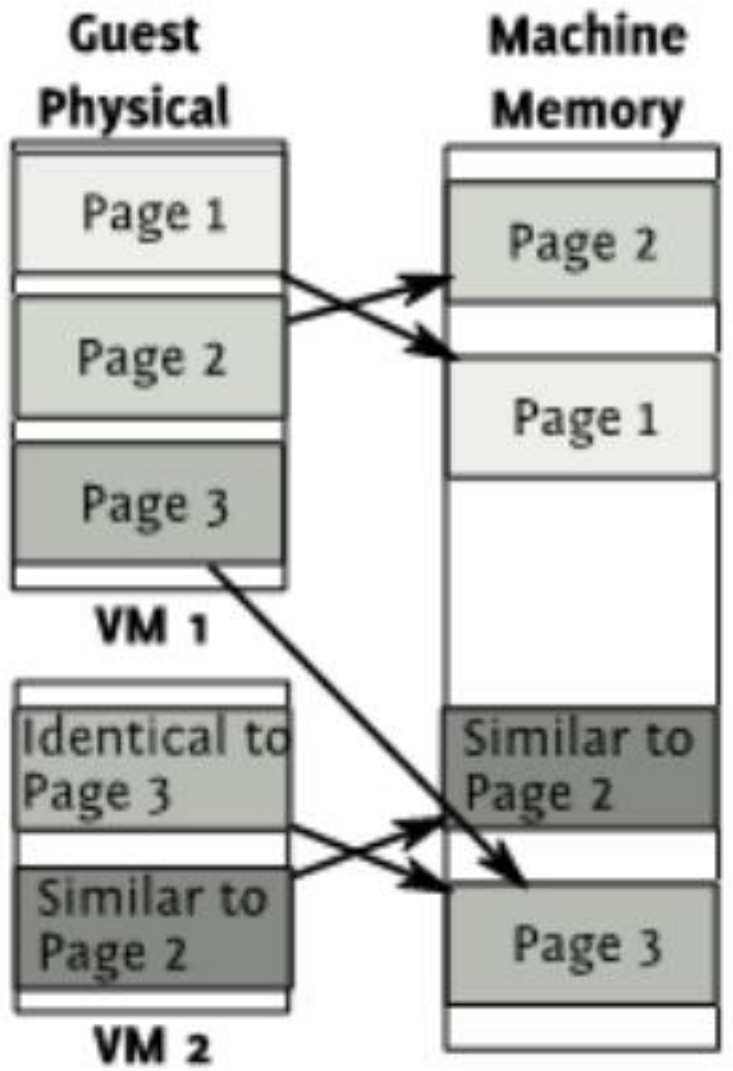
Difference Engine: Harnessing Memory Redundancy in Virtual Machines

----OSDI-08

内存仍然是稀缺资源

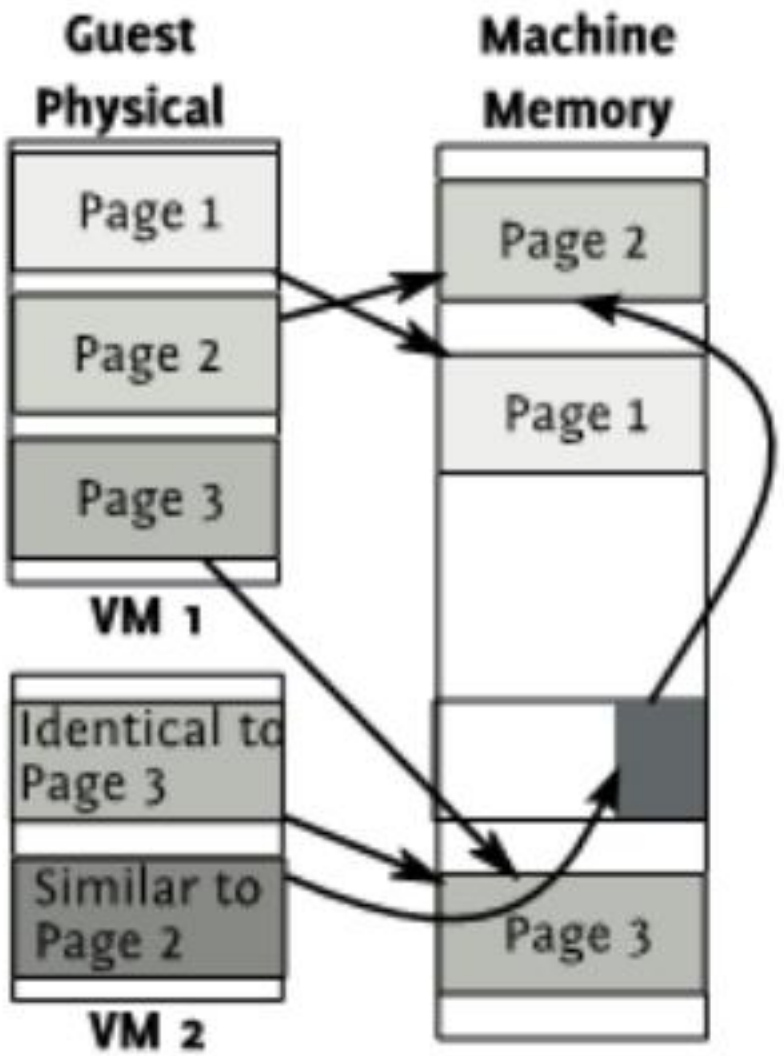


(a) Initial

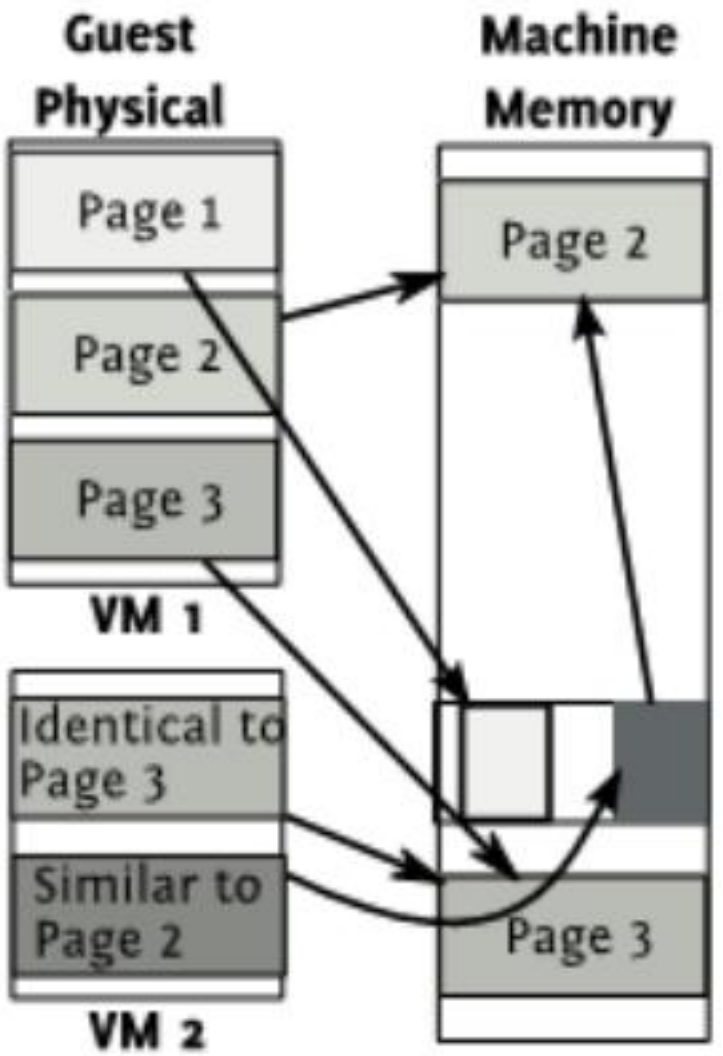


(b) Page sharing

改进方案



(c) Patching



(d) Compression

测试数据

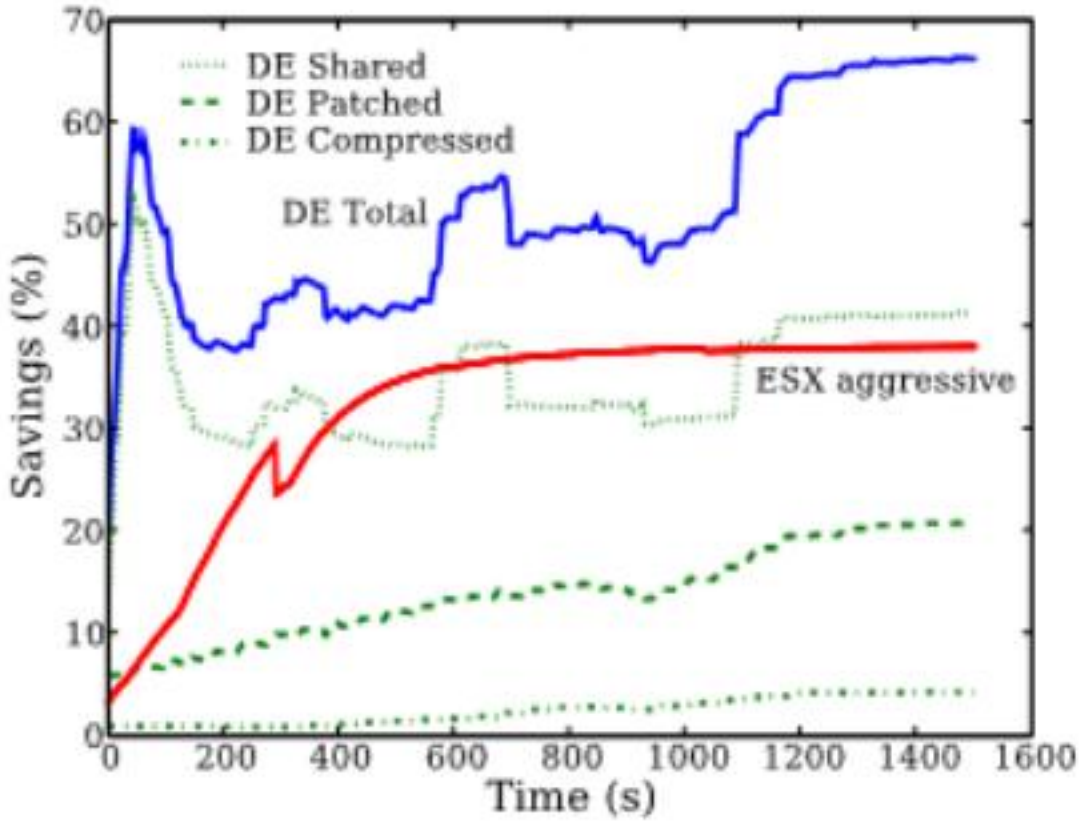


Figure 10: Memory savings for MIXED-1. Difference Engine saves up to 45% more memory than ESX.

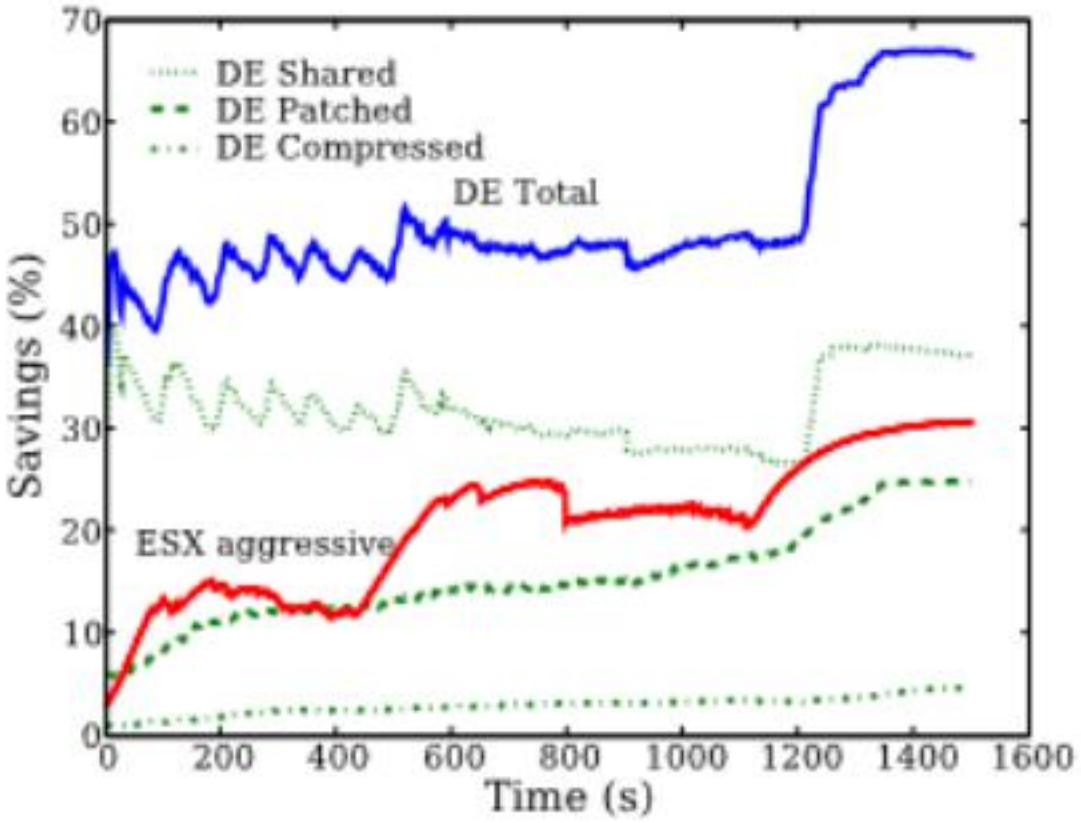
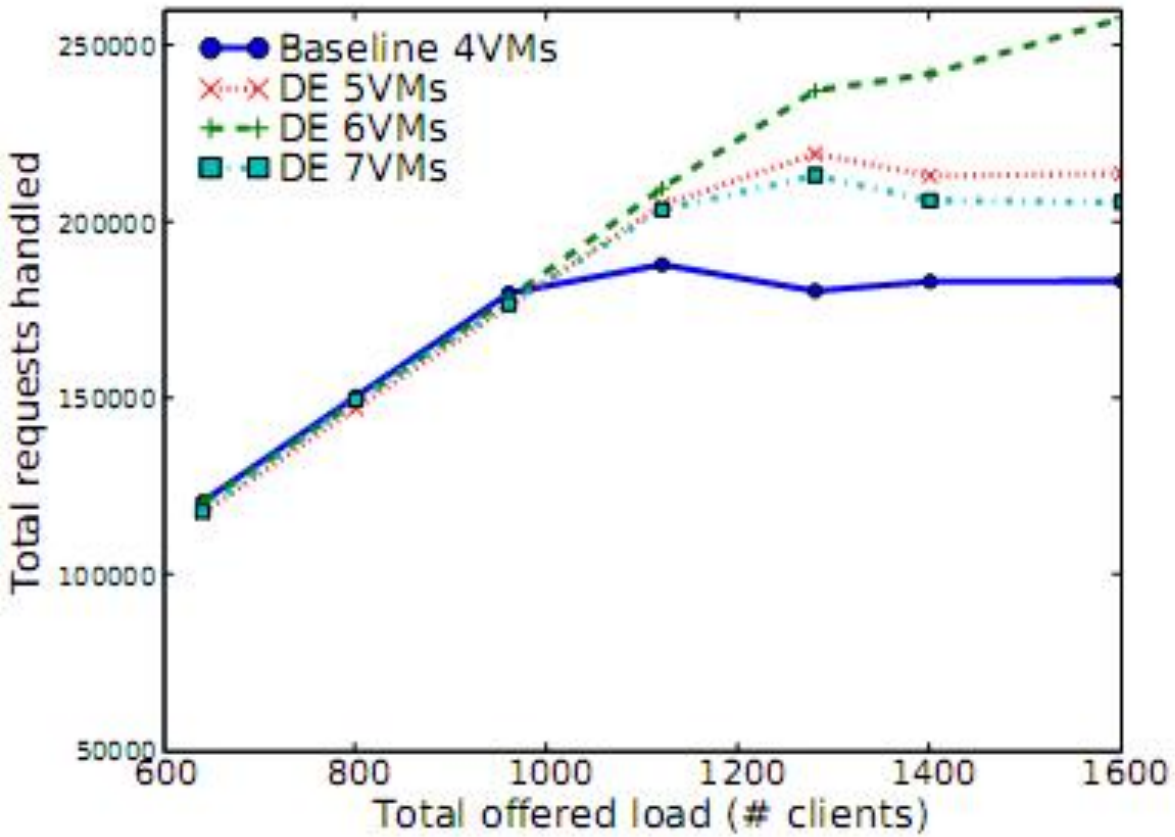
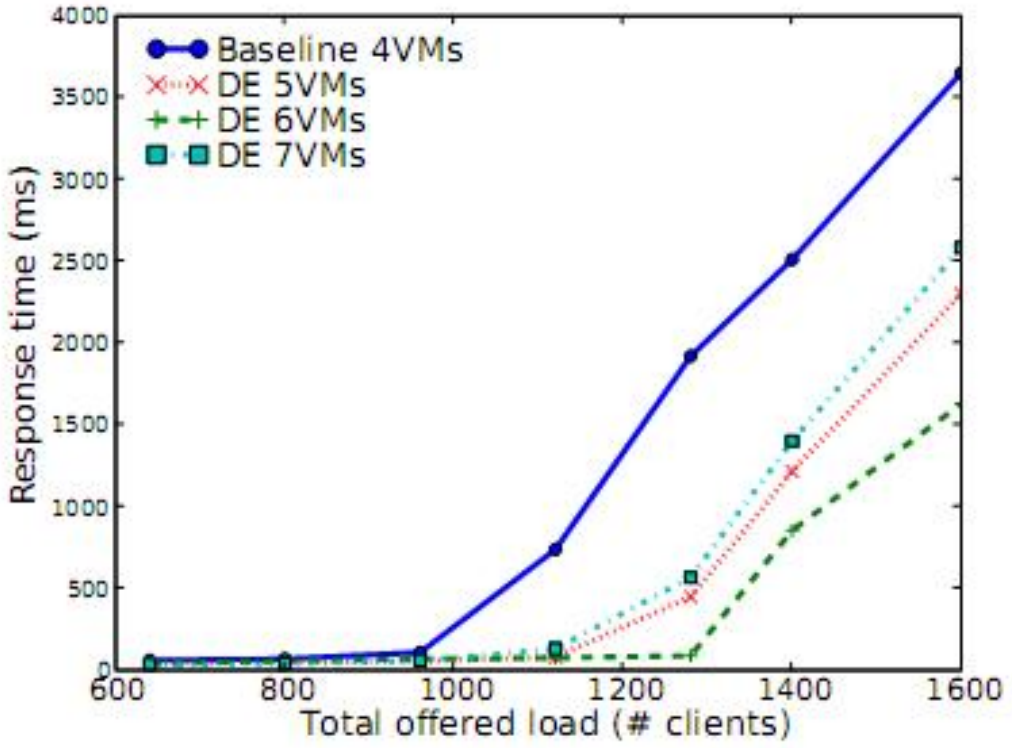


Figure 11: Memory savings for MIXED-2. Difference Engine saves almost twice as much memory as ESX.

测试数据



(a) Total requests handled



(b) Average response time

Figure 12: Up to a limit, Difference Engine can help increase aggregate system performance by spreading the load across extra VMs.

Hope you enjoyed the OS course!