

虚拟内存

2200017853 李长烨

动机：

保护进程的地址空间的要求、物理内存的稀有、共享主存资源的需求，使得我们需要更高效地管理和利用主存。

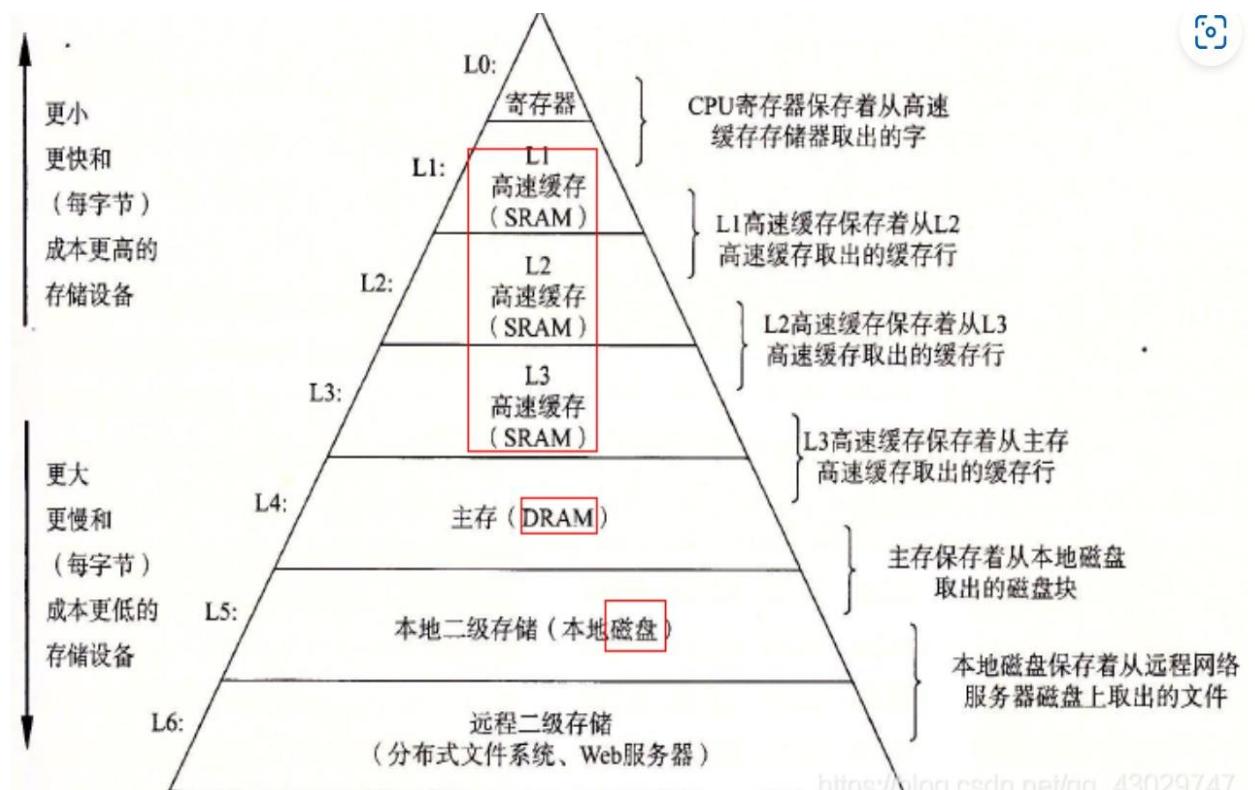
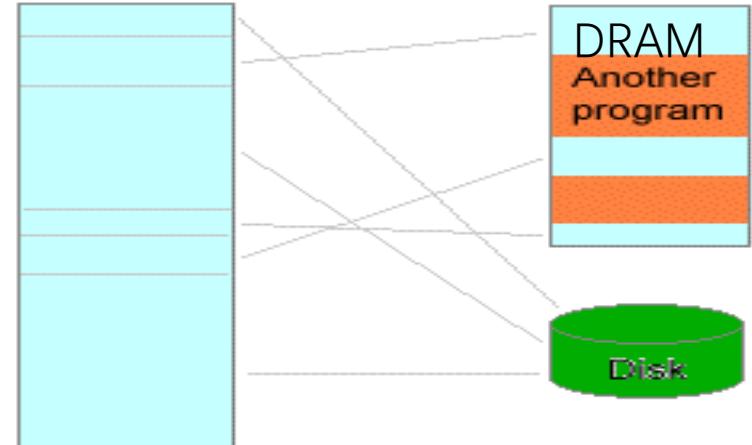
如何保护地址空间/物理内存？虚拟化，不直接修改物理内存。（地址翻译）

如何更高效地利用物理内存？合理替换机制保证最常用的地址/数据能够最快地被取到。（内存替换/页替换）

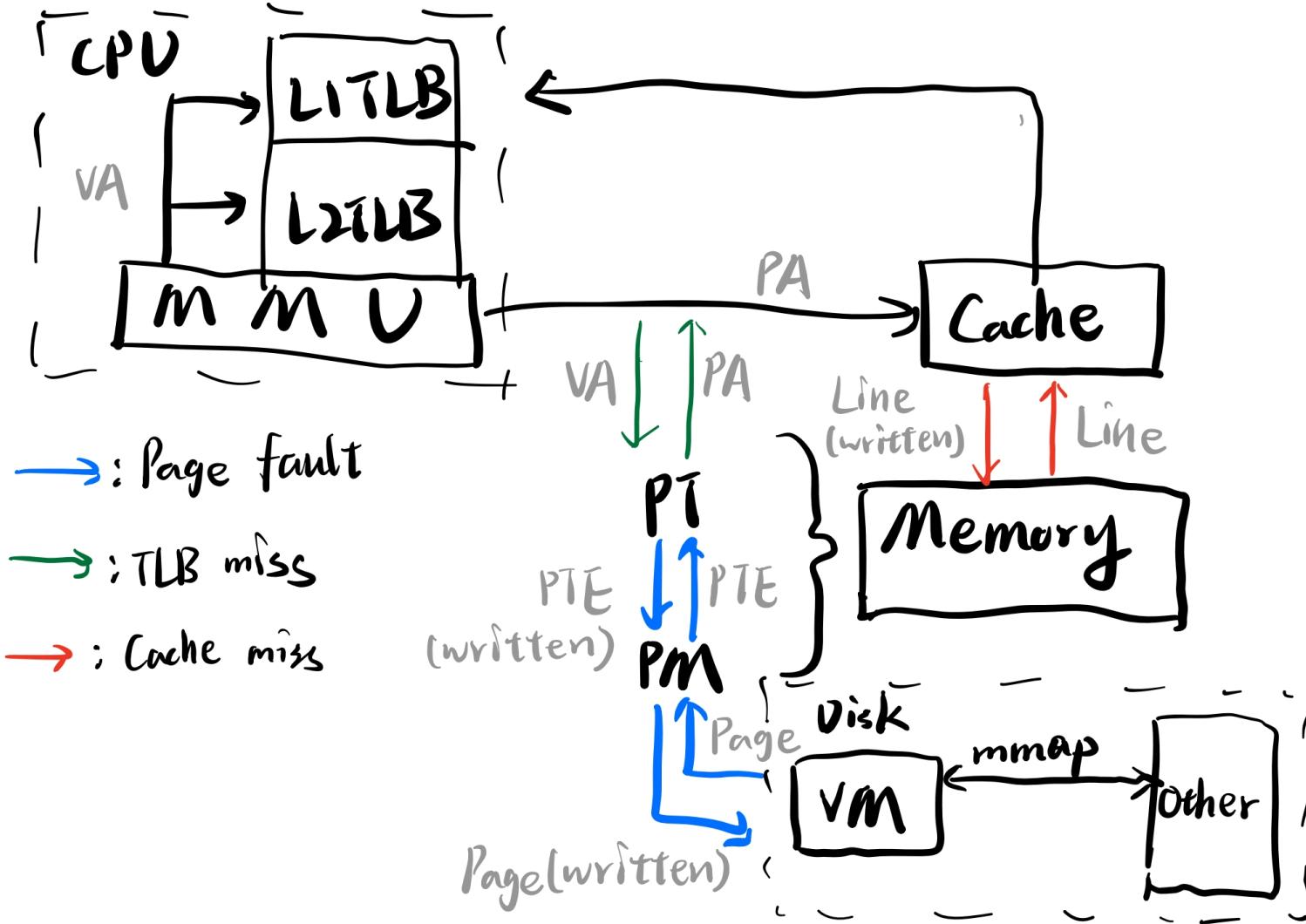
如何节省共享资源的空间？尽可能减少复制的次数。（写时替换）

Recall：第六章的存储器层次结构中运用的机制，程设中指针、多态与继承。

Application sees: But in reality:



总览：



目录：

1. 地址空间
2. 页表
3. 页
4. 通过虚拟内存技术实现的缓存
5. 地址翻译
6. 缺页
7. Linux虚拟内存系统
8. 内存映射

地址空间

我们这里假设虚拟地址空间是线性的且物理地址空间字节数为2的幂，实际上当我们之后讨论到Linux地址空间的时候，这些假设就不再必要了

在讨论内存时：

$$K = 2^{10}, M = 2^{20}, G = 2^{30}, T = 2^{40}, P = 2^{50}, E = 2^{60},$$

虚拟地址位数 (n)	虚拟地址数 (N)	最大可能的虚拟地址
8		
	$2^8 = 64K$	
		$2^{32} - 1 = ?G - 1$
	$2^7 = 256T$	
64		

页表

是页表条目(PTE)的数组,作用是将虚拟地址(VA)转化为物理地址(PA)

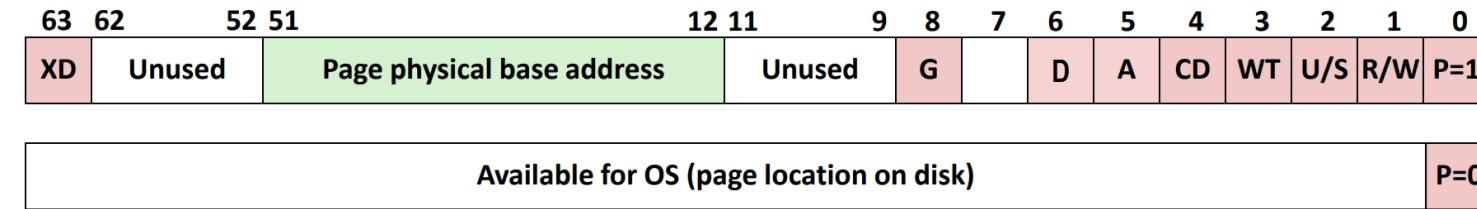
PTE:记录了地址以及该地址中数据的一些特征,各个位的含意参考课本P578

一些比较重要的字段:

- 52~12: 页表物理基址(PPBA)或物理页号(PPN)
- P:有效位
- A:引用位, 通过A和D实现了页表的替换机制
- D:修改位, 判定是否写回

页表优化:

- 多级页表: 减少内存要求, 只有一级页表总是在主存中, 只有最经常使用的二级页表缓存在主存中
- TLB: 即翻译后备缓冲器(P570), 位于CPU中, 类比于Cache



一个(最低级)PTE的结构

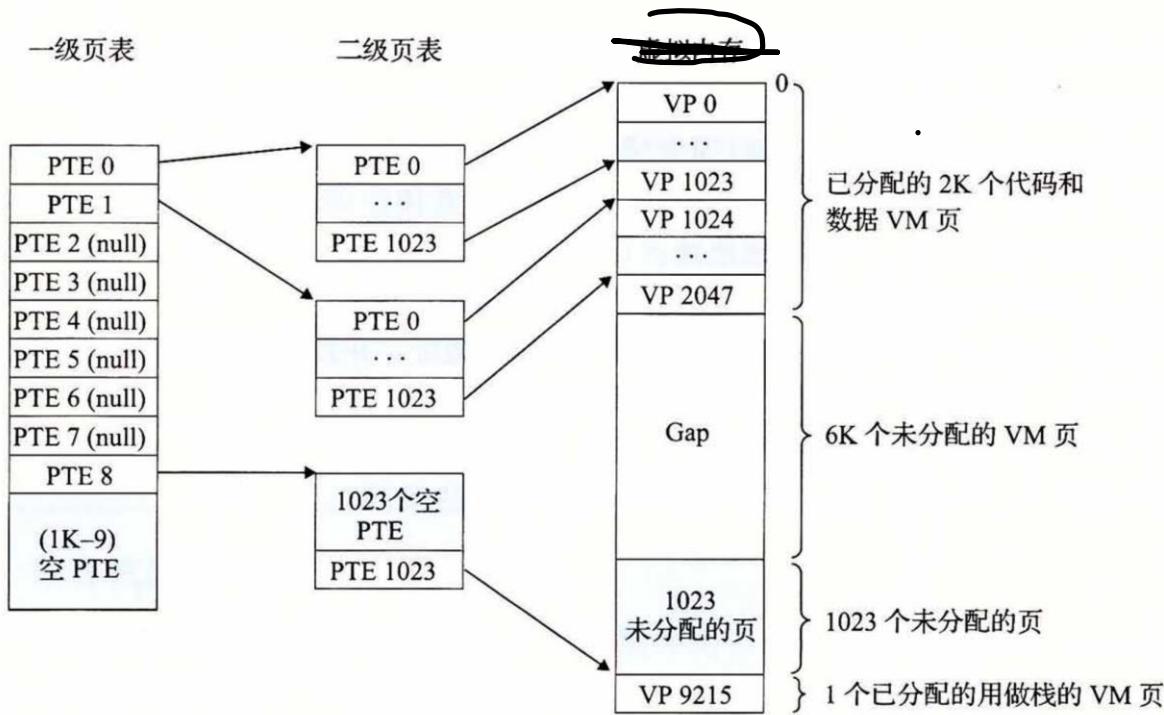


图 9-17 一个两级页表层次结构。注意地址是从上往下增加的

17. 在页表条目中，以下哪个条目是由 MMU 在读和写时进行设置，而由软件负责清除：
- A. P 位，子页或子页表是否在物理内存中
 - B. G 位，是否为全局页（在任务切换时，不从 TLB 中驱逐出去）
 - C. CD 位，能/不能缓存子页或子页表
 - D. D 位，修改位，是否对子页进行了修改操作

D

修改位（由MMU在读和写时设置，由软件清除）

页是内存大小的一个单位，分页是地址空间的一种实现

并非所有内存都可以分页的，操作系统内核以及驱动程序使用的一部分内存是不能分页的（例如处理分页错误有关的代码），只能存储在主存(DRAM)中

虚拟内存分割形成的块称为虚拟页，物理内存分割形成的块称为物理页

虚拟页总在三个状态之一：

- 未分配的
- 缓存的
- 未缓存的

三个状态的转换：分配通过内存映射实现，缓存通过缺页异常实现

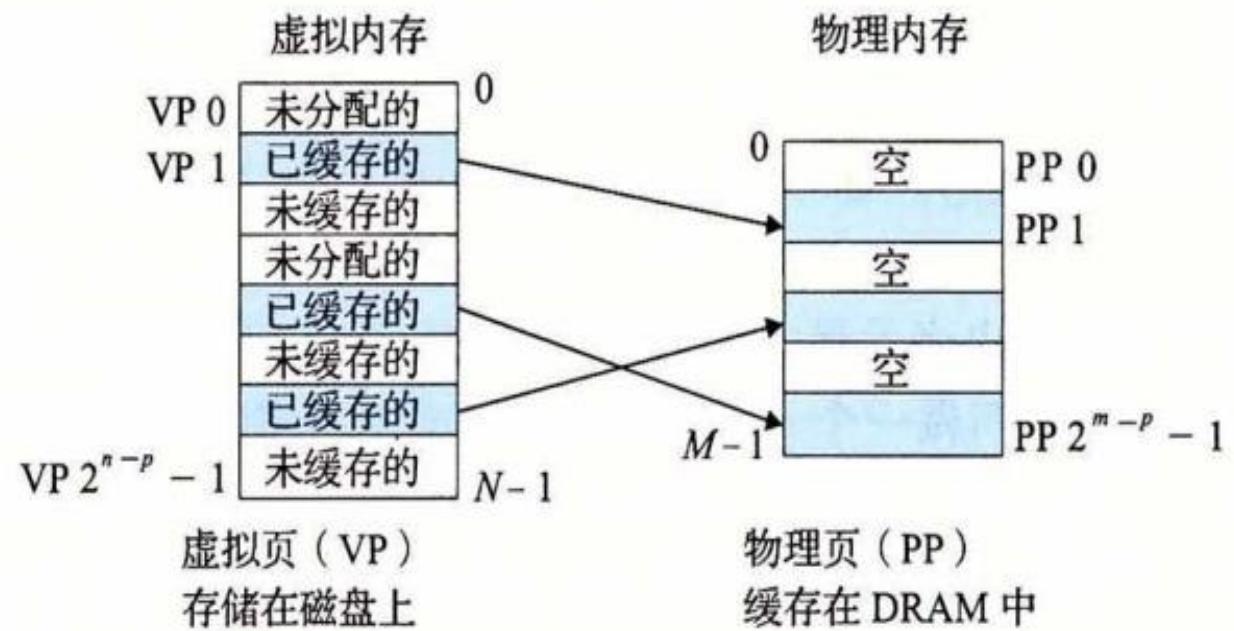


图 9-3 一个 VM 系统是如何使用主存作为缓存的

有时候放图只是因为不想留白

通过虚拟内存技术实现的缓存

- 回忆到缓存的概念：对于每个k，位于k层的更快更小的存储设备作为第k+1层的更大更慢的存储设备的缓存。(P422)

页表实现了将主存(物理内存\\Memory)作为磁盘(Disk)的缓存

易混淆的两种虚拟内存概念：

- 将磁盘的一部分作为一个存储空间，**物理内存是这个空间的缓存**，这个空间称为虚拟内存。
- 将磁盘的一部分当作DRAM来使用，**是对物理内存的一种扩充**，这一部分称为虚拟内存。

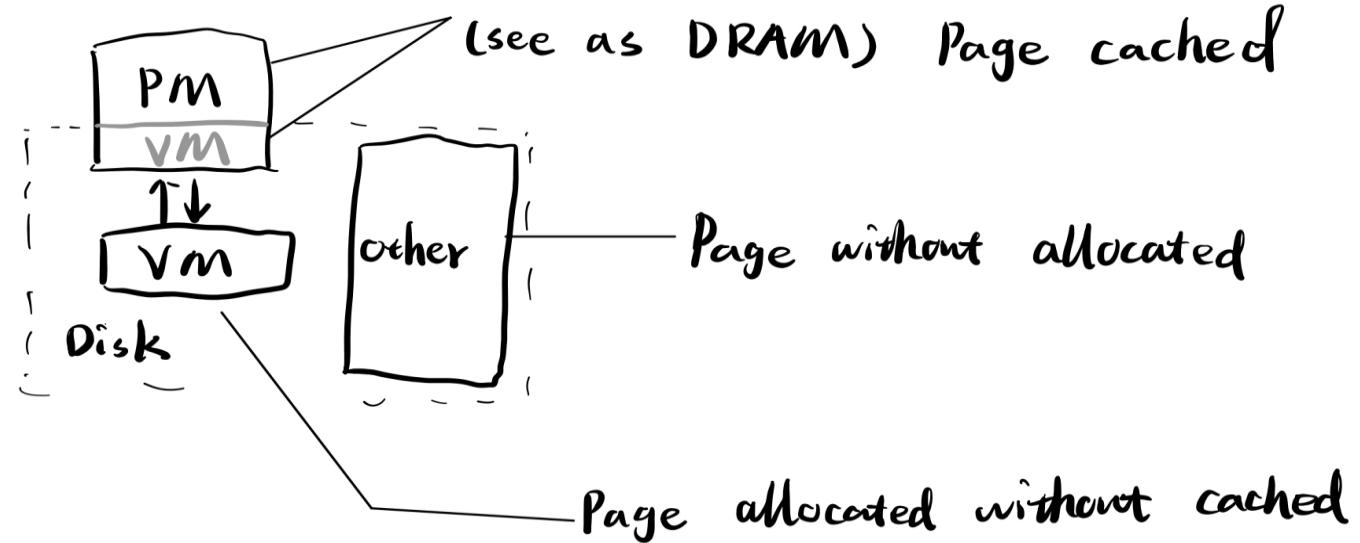
虚拟内存

分页文件是硬盘上的一块区域，Windows 当作 RAM 使用。

所有驱动器总分页文件大小：

3092 MB

更改(C)...



本章所有提到的虚拟内存技术相关概念都是对于第一种理解而言的。

这一意义下，所有物理内存中的内容都在虚拟内存中，所有物理页都存在与之内容相同的虚拟页。

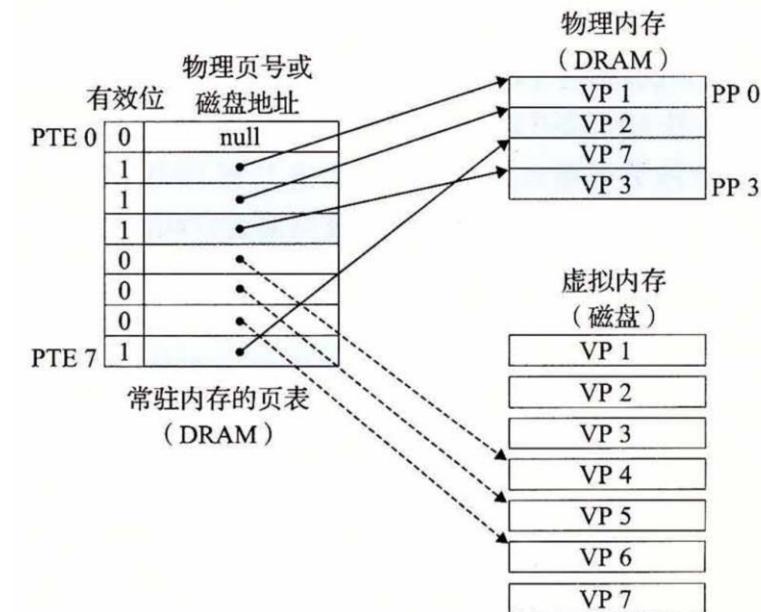


图 9-8 分配一个新的虚拟页面。内核在磁盘上分配 VP 5，并且将 PTE 5 指向这个新的位置

地址翻译

1. 处理器生成虚拟地址，并传送给MMU
2. MMU生成PTE，并向cache发出请求
3. MMU构造物理地址，传送到cache
4. Cache将数据返回处理器

引入TLB(翻译后备缓存器)的情形：对PTE加一层

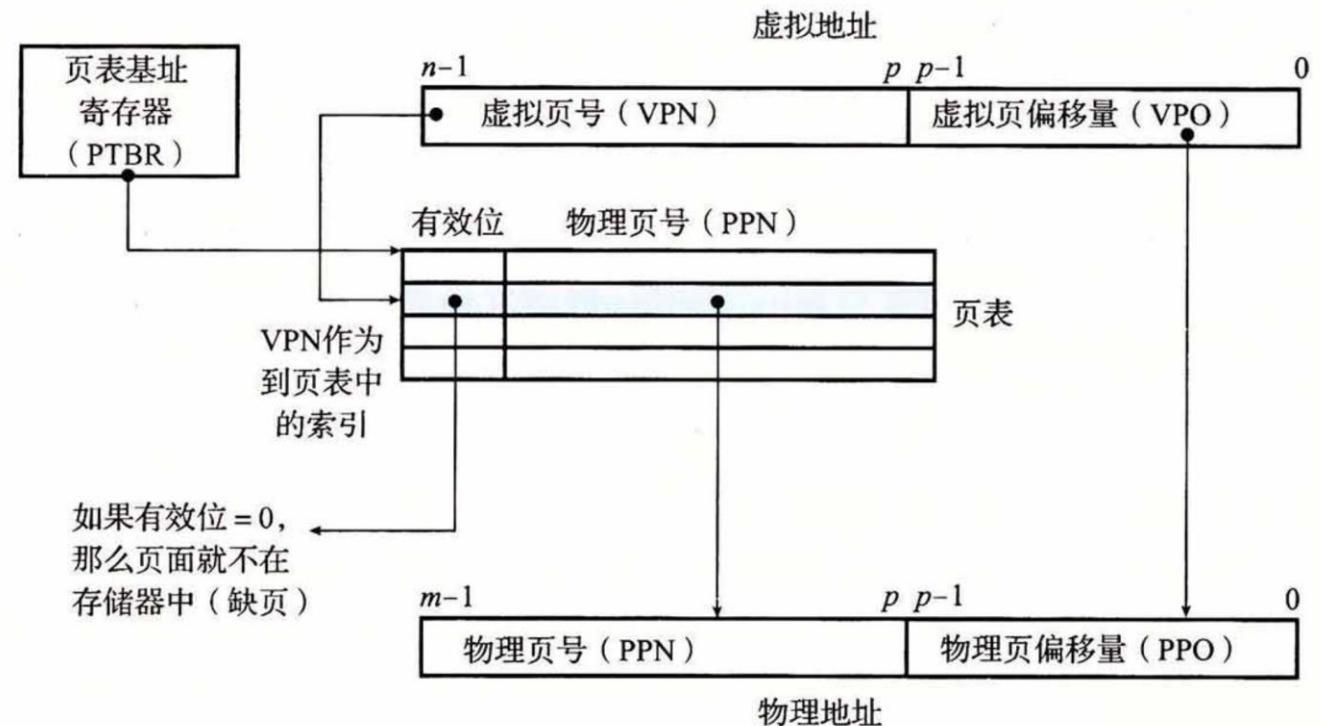
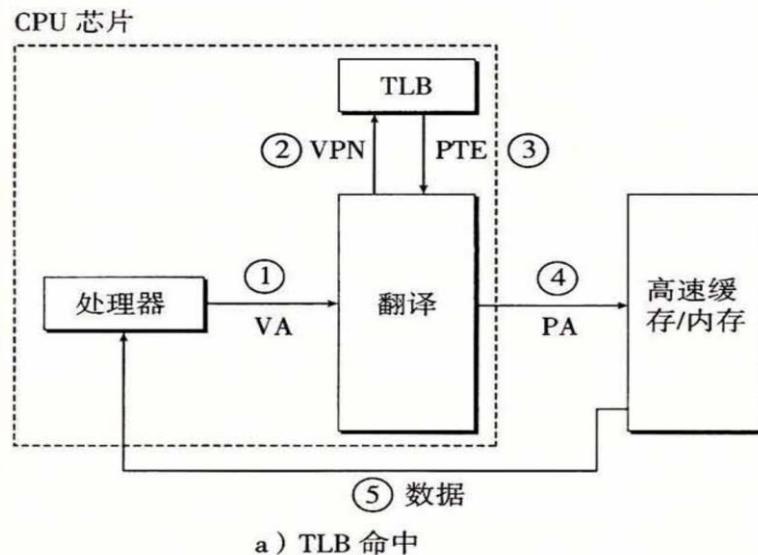
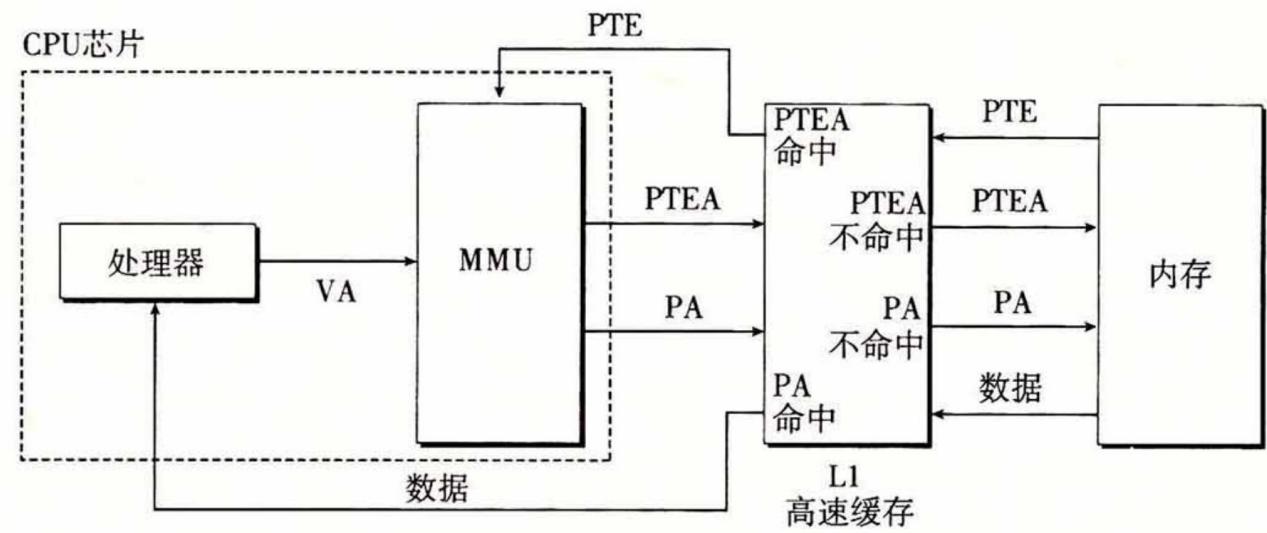


图 9-12 使用页表的地址翻译

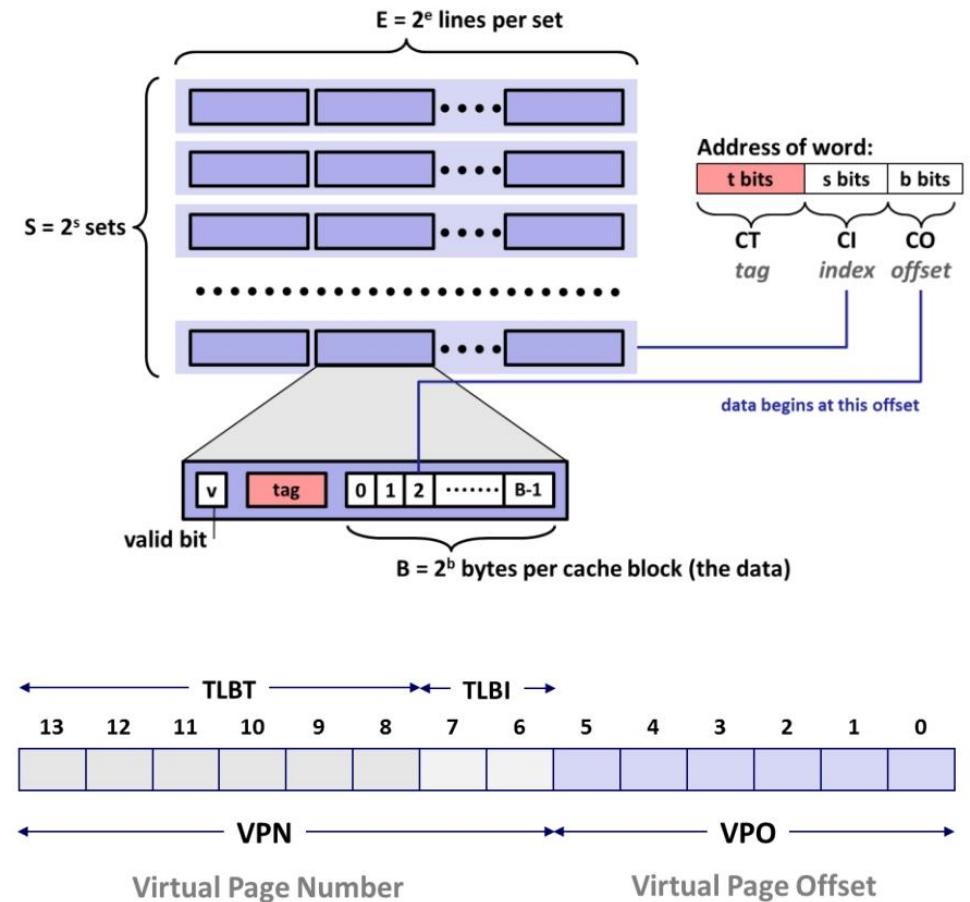


地址翻译

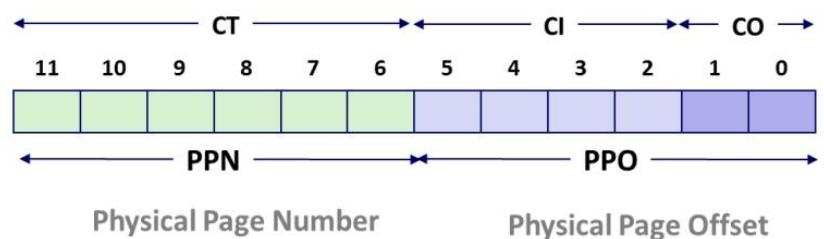
基本参数	
符 号	描 述
$N = 2^n$	虚拟地址空间中的地址数量
$M = 2^m$	物理地址空间中的地址数量
$P = 2^p$	页的大小 (字节)

虚拟地址 (VA) 的组成部分	
符 号	描 述
VPO	虚拟页面偏移量 (字节)
VPN	虚拟页号
TLBI	TLB 索引
TLBT	TLB 标记

物理地址 (PA) 的组成部分	
符 号	描 述
PPO	物理页面偏移量 (字节)
PPN	物理页号
CO	缓冲块内的字节偏移量
CI	高速缓存索引
CT	高速缓存标记



1) (bits per field for our simple example)



全相联高速缓存

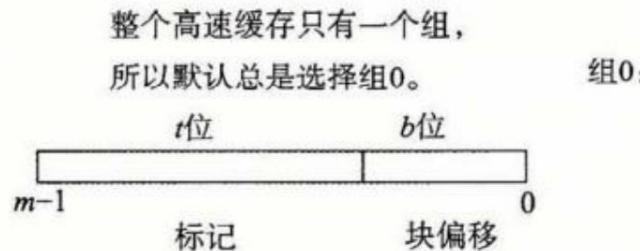


图 6-36 全相联高速缓存中的组选择。注意没有组索引位

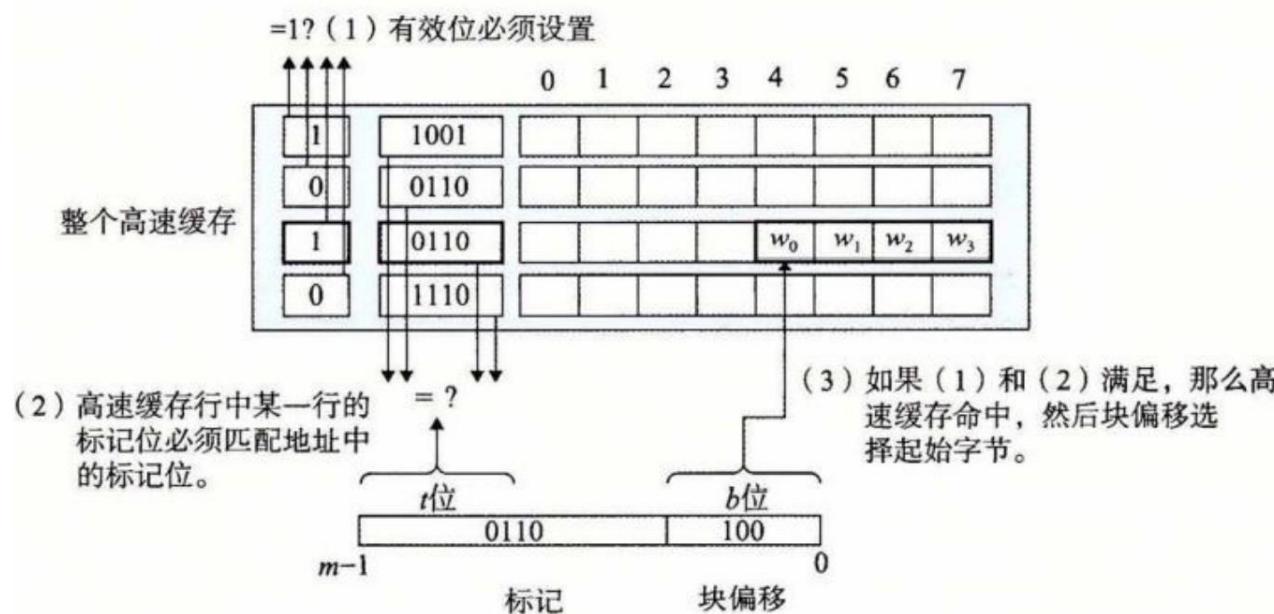


图 6-37 全相联高速缓存中的行匹配和字选择

- 一个包含所有高速缓存行的组
- 不需要组索引位，默认为0组
- 因为高速缓存电路必须并行搜索许多相匹配的标记，构造一个又大又快的相联高速缓存很困难并且昂贵。因此，全相联高速缓存只适合做小的高速缓存，例如虚拟内存系统中的快表TLB

地址翻译

0000 0011
1101 0111

0xf

0x3

0x3

T

F

0xd

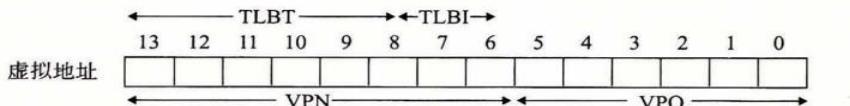
0x3

0x5

0xd

T

0x1d



	位标记位	PPN	有效位	标记位	PPN	有效位	标记位	PPN	有效位	标记位	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

a) TLB: 四组, 16 个条目, 四路组相联

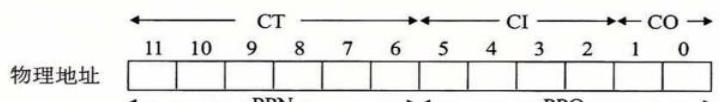
VPN PPN 有效位

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

b) 页表: 只展示了前 16 个 PTE



索引 标记位 有效位 块 0 块 1 块 2 块 3

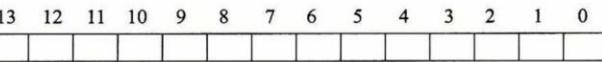
0	19	1	99	11	23	11
1	15	0	-	-	-	-
2	1B	1	00	02	04	08
3	36	0	-	-	-	-
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	-	-	-	-
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	-	-	-	-
A	2D	1	93	15	DA	3B
B	0B	0	-	-	-	-
C	12	0	-	-	-	-
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	-	-	-	-

c) 高速缓存: 16 个组, 4 字节的块, 直接映射

练习题 9.4 说明 9.6.4 节中的示例内存系统是如何将一个虚拟地址翻译成一个物理地址和访问缓存的。对于给定的虚拟地址, 指明访问的 TLB 条目、物理地址和返回的缓存字节值。指出是否发生了 TLB 不命中, 是否发生了缺页, 以及是否发生了缓存不命中。如果是缓存不命中, 在“返回的缓存字节”栏中输入“—”。如果有缺页, 则在“PPN”一栏中输入“—”, 并且将 C 部分和 D 部分空着。

虚拟地址: 0x03d7

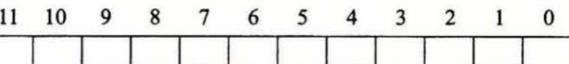
A. 虚拟地址格式



B. 地址翻译

参数	值
VPN	
TLB 索引	
TLB 标记	
TLB 命中? (是/否)	
缺页? (是/否)	
PPN	

C. 物理地址格式



D. 物理内存引用

参数	值
字节偏移	
缓存索引	
缓存标记	
缓存命中? (是/否)	
返回的缓存字节	

地址翻译

在Intel core i7中VA地址空间大小
 $2^{48}B$ 其中36位作为VPN， 12位作为
VPO

PPN为40位，与PTE中的物理页号对
应

优化地址翻译：提前将VPO发送到
L1cache中查找对应的组，设计
VPO=PPO，提高并行性从而提高效
率

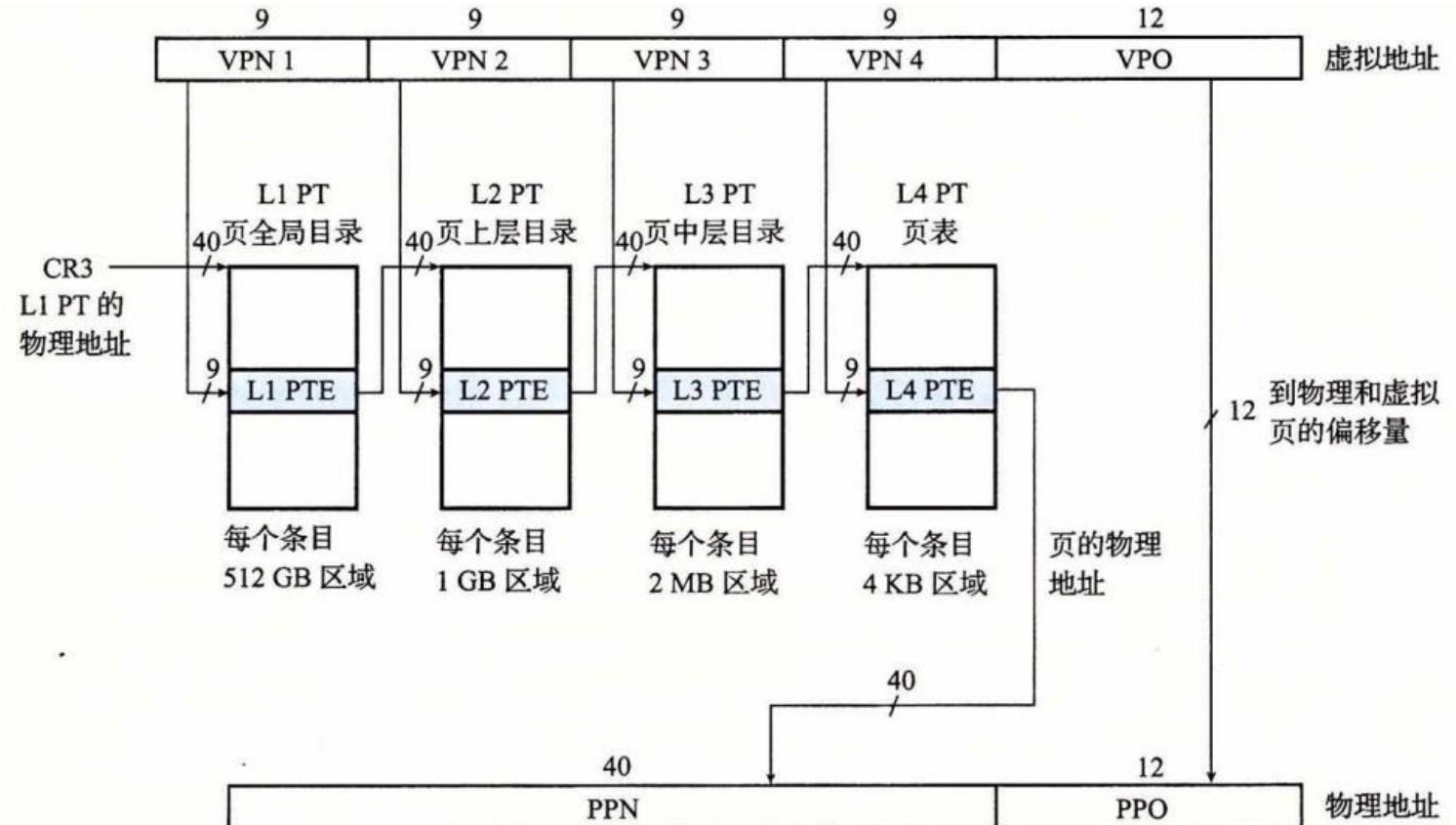


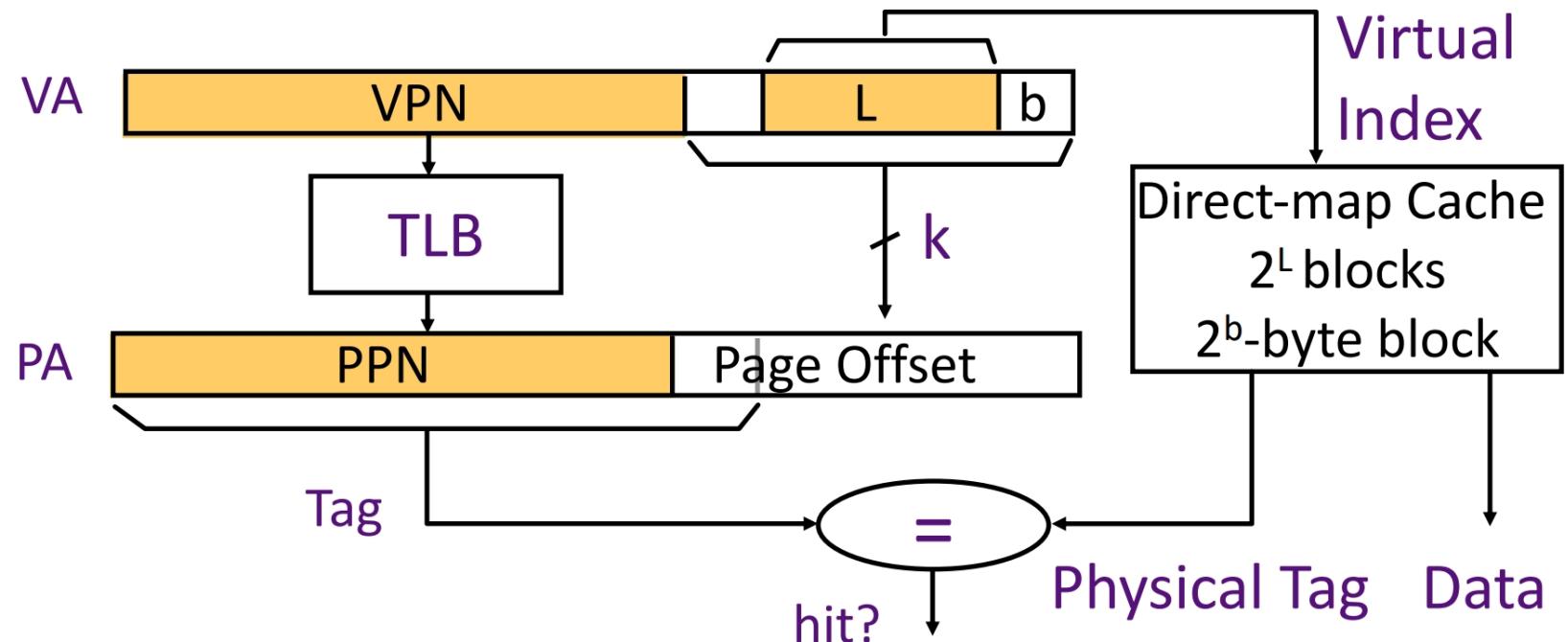
图 9-25 Core i7 页表翻译(PT：页表，PTE：页表条目，VPN：虚拟页号，VPO：虚拟页偏移，
PPN：物理页号，PPO：物理页偏移量。图中还给出了这四级页表的 Linux 名字)

地址翻译

一个问题是这样的机制使得cache不能超过页的大小,

然而这是不现实的，回忆到page 4KB, L1 cache 32KB

Concurrent Access to TLB & Cache (Virtual Index/Physical Tag)



Index L is available without consulting the TLB

→ *cache and TLB accesses can begin simultaneously!*

Tag comparison is made after both accesses are completed

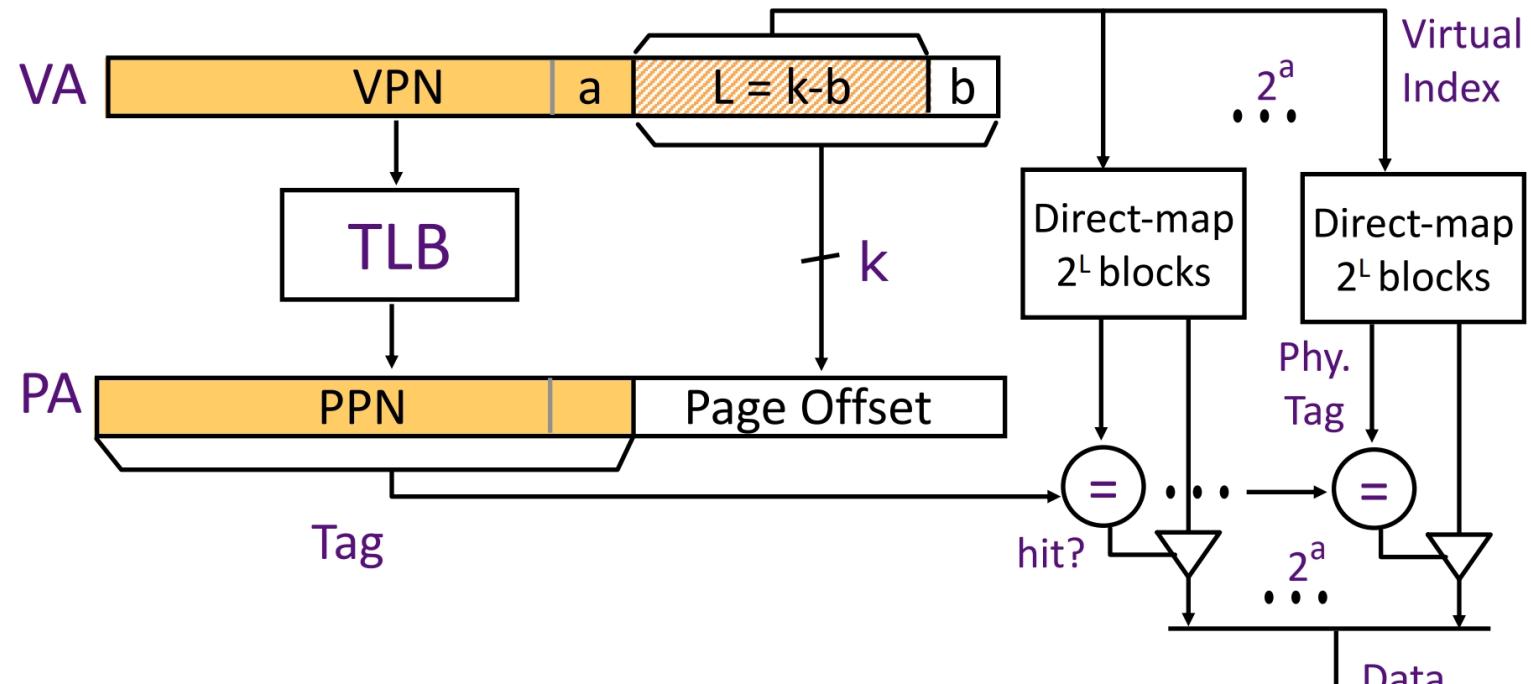
Cases: only works when $L + b \leq k$

- *cache can only be as big as page size!*

地址翻译

实际上，PPN一部分也会用到行匹配和组选择的过程中，比如说这里 $a=3$ ，就是有3位运用到了这里。

Virtual-Index Physical-Tag Caches: Use associativity to further increase cache capacity



After the **PPN** is known, 2^a physical tags are compared

$$4KB \text{ page size} * 8\text{-way } (2^3) \text{ associative} = 32KB \text{ cache}$$

缺页

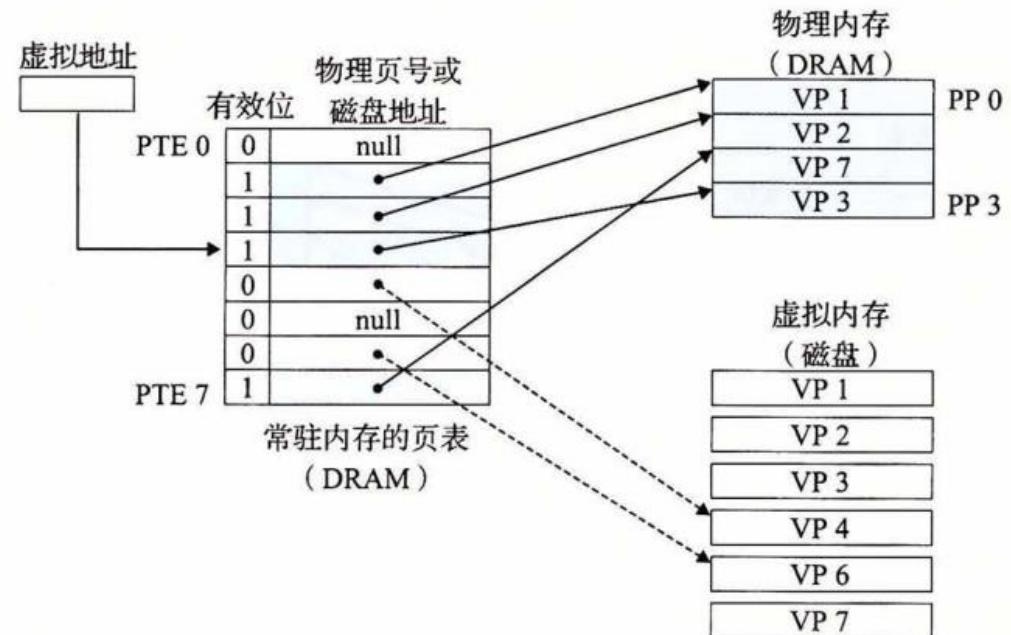
在给出完整的地址翻译流程前，我们先专门讨论一下缺页。

缺页中断分为三类：

- 硬(缺页)中断：我们第八章讨论过的情形，物理内存中没有对应的页帧，需要CPU打开磁盘设备读取到物理内存中，再让MMU建立VA和PA的映射。
- 软缺页中断：物理内存中是存在对应页帧的，只不过可能是其他进程调入的，发出缺页异常的进程不知道而已，此时MMU只需要建立映射即可，无需从磁盘读取写入内存，一般出现在多进程共享内存区域。
- 无效缺页中断：比如进程访问的内存地址越界访问，又比如对空指针解引用导致内核产生segment fault错误

Linux分配物理页的原则：按需页面调度

页面置换策略：现实中采用工作集时钟页面置换算法，主要思路类似于LRU



缺页的发生：VA索引PTE发现P=0，引发缺页，内核通过页面置换策略替换一个已缓存的页面(如果该页面被修改还要写回磁盘)，更新物理内存，之后内核更新PTE，最后重新运行引发缺页的指令

缺页

缺页处理：

1. 判断地址是不是存在的
和每个段的起始位置作比较，如果没有找到在某个段中，则终止
2. 判断访问是不是合法的
查看读/写/执行位
3. 前两个都不是，则页故障是缺页导致的
将页面调入

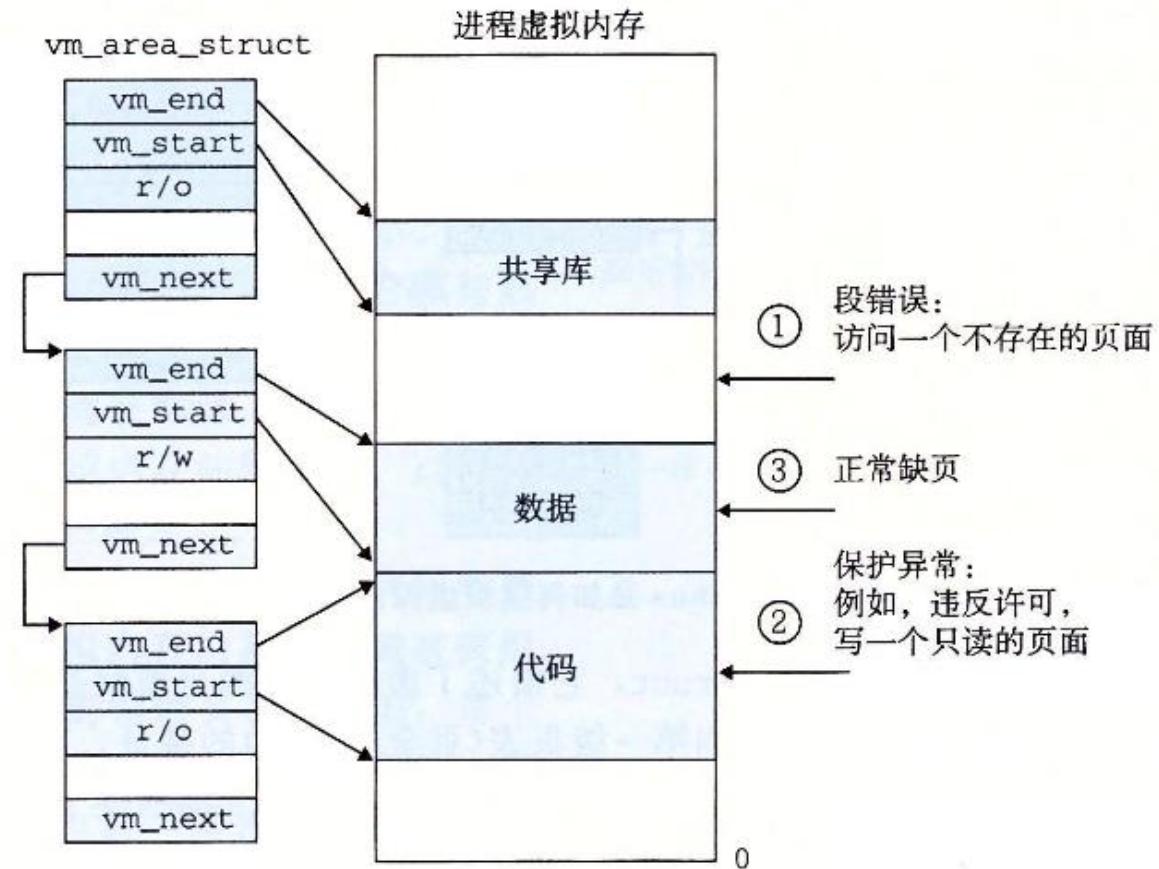


图 9-28 Linux 缺页处理

缺页

缺页在内核空间和用户空间均会发生，但总体而言缺页在正常情形下是极少发生的

局部性原则保证程序在一个较小的工作集上运行，保证了虚拟内存的高效

名称	PID	硬中断/秒	提交(KB)	工作集(KB)	可共享(KB)	专用(KB)
MsMpEng.exe	6160	3	310,200	272,424	38,376	234,048
POWERPNT.EXE	16636	0	352,336	394,188	161,592	232,596
dwm.exe	1296	0	270,212	291,920	60,828	231,092
msedge.exe	14556	0	235,132	246,208	65,048	181,160
msedge.exe	13372	0	147,336	303,356	182,188	121,168
msedge.exe	17228	0	127,100	203,932	90,744	113,188
SearchHost.exe	11184	0	121,948	199,160	98,732	100,428
explorer.exe	10244	0	157,628	224,584	132,484	92,100

正常工作状态下缺页很少发生

名称	PID	硬中断/秒	提交(KB)	工作集(KB)	可共享(KB)	专用(KB)
netconvert.exe	16656	135	21,902,236	10,060,484	828	10,059,656
Memory Compression	3776	167	9,460	1,462,312	0	1,462,312
MsMpEng.exe	6528	7	406,548	166,668	31,164	135,504
msedge.exe	11772	3	206,652	142,988	84,804	58,184
dwm.exe	1072	0	880,488	127,788	81,820	45,968
explorer.exe	3312	1	431,412	91,480	49,276	42,204
TextInputHost.exe	10880	0	57,236	76,708	73,648	3,060
msedge.exe	4360	0	39,700	63,644	60,072	3,572

工作集大小接近或超过了物理内存的大小($16\text{GB}=16,777,216\text{KB}$)，发生抖动(thrashing)，缺页增加

(U/S): 1 表示内核模式，0 表示用户模式 ←

4. Double fault: Intel 处理器中有一种特殊的异常，被称为 double fault。此异常发生表明调用某个故障 (**fault**) A 的处理程序后又触发了另一个故障 B。正常情况下，故障 B 会有相应异常处理程序来处理，因此两个故障 B 和 A 可以被顺序解决。但是如果处理器无法正常处理故障 B，或是处理了之后依然无法处理故障 A，就会产生 double fault，并终止 (abort)。假设除了缺页异常处理程序外，其他异常处理程序都不会产生新的故障。如果在某次缺页故障时产生了 double fault，其原因可能是 _____ (不定项选择，都选对才得分，1 分) ←

- ① 运行缺页故障处理程序时，CPU 上的权限位是内核态，但所执行代码段 U/S=0 ←
- ② 运行缺页故障处理程序时，CPU 接收到了键盘发送的 Ctrl + C 信号 ←
- ③ 缺页故障处理程序没有加载到主存中 ←

4. 本题考查对缺页故障、异常处理的综合理解，属于难题。←

- ① 如果 CPU 权限位是内核态，它自然可以运行用户级代码，不会触发异常。←
- ② 如果缺页故障处理程序运行时，收到外部中断，这不属于故障，因此不会触发。←
- ③ 如果缺页故障处理程序不在主存，那么调用它会触发缺页故障，于是这个故障又会导致调用缺页故障处理程序，但它仍然不在主存，故这一故障无法被解决，于是触发 double fault。←

相关内容参见课本第 8.1 节(P502-507)、9.7.2 节(P581-582)和 Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3: System Programming Guide 的第 6.15 节。←

地址翻译

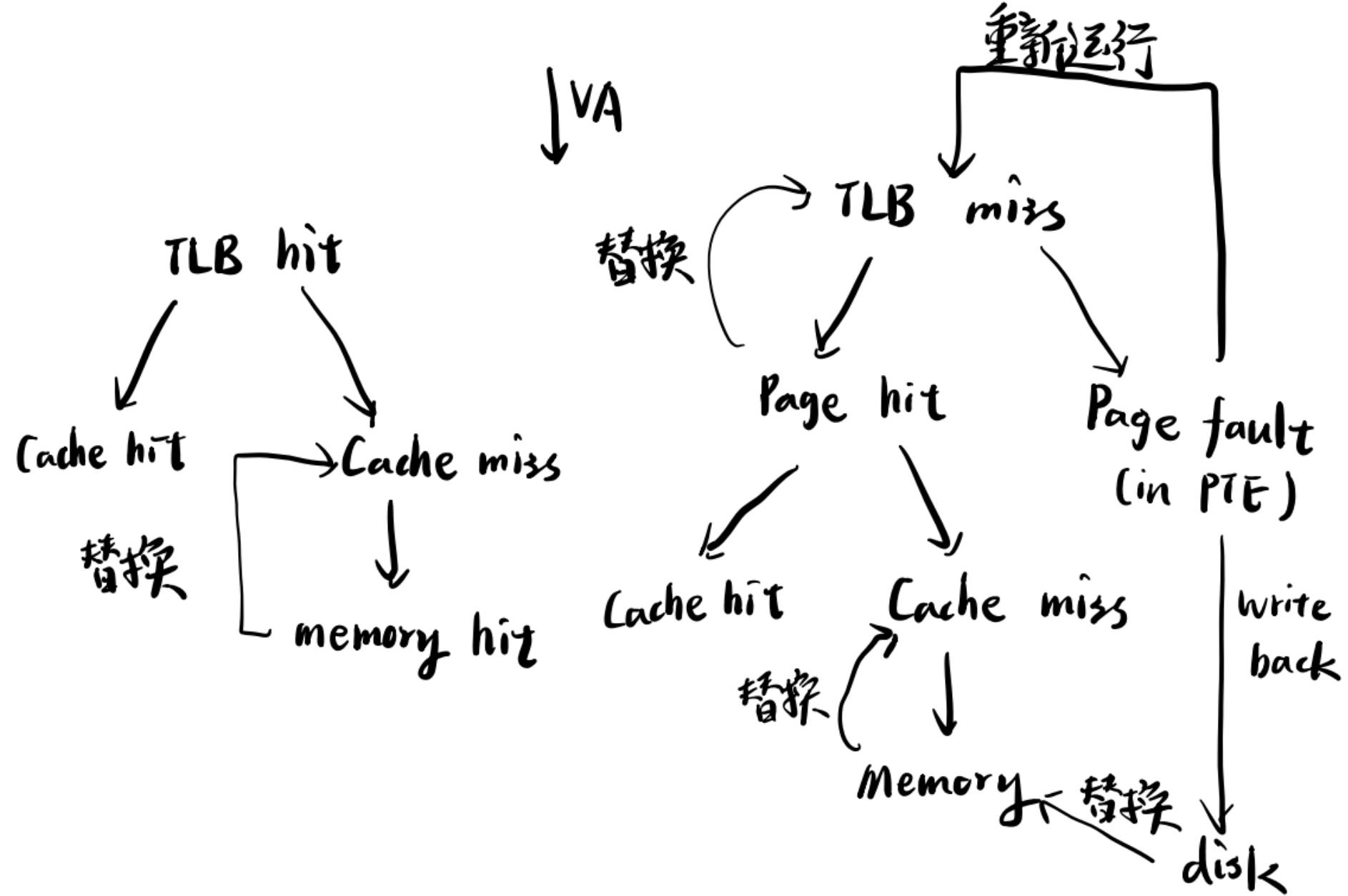
做一个总结性的讨论：

TLB hit/miss

Page hit/fault

Cache hit/miss

理论上有8种组合，但实
有些情形不会发生

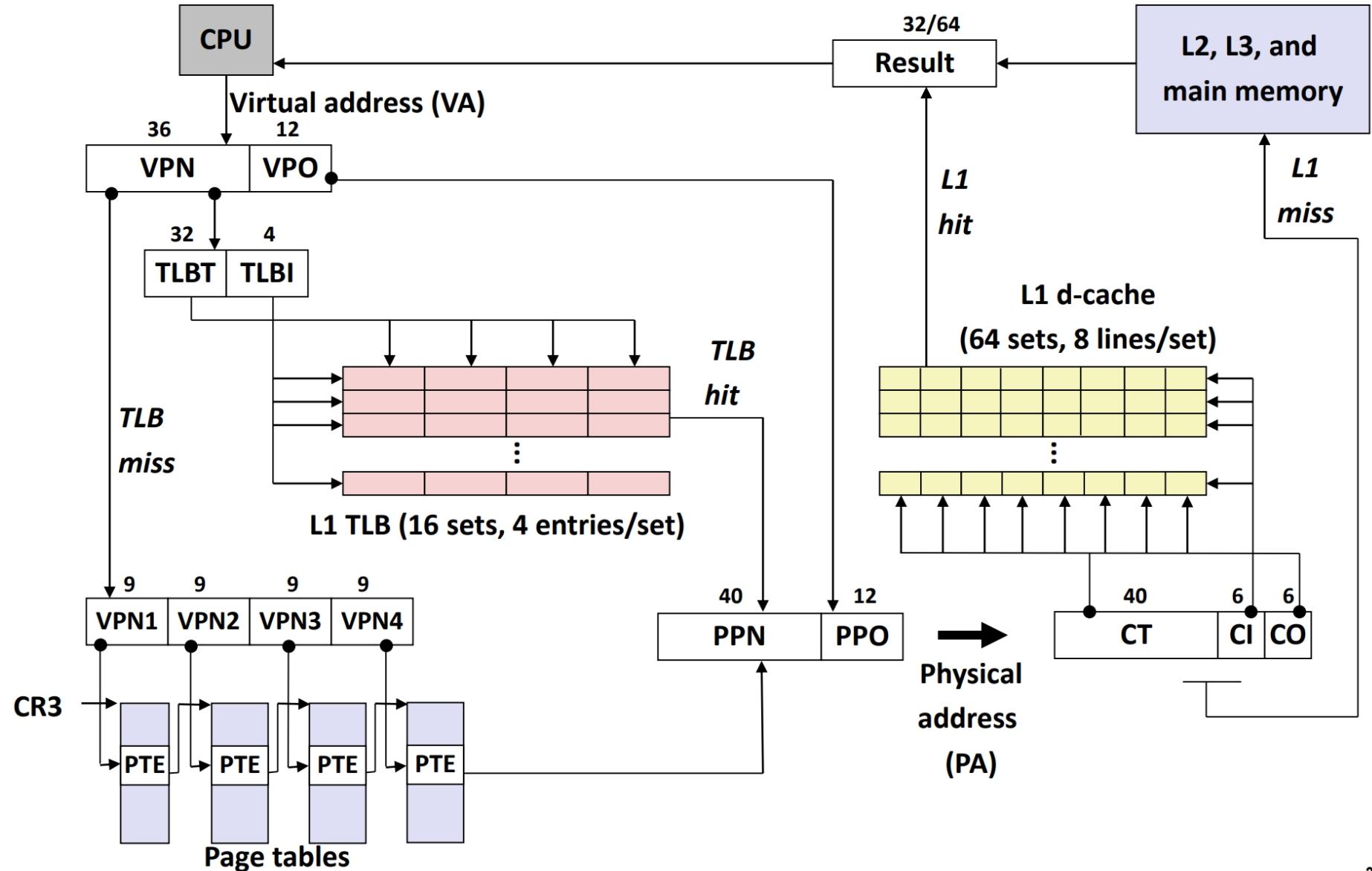


地址翻译

End-to-end Core i7 Address Translation

More detailed

注意每个事件发生在
硬件中的哪里



Linux 虚拟内存系统

- **进程虚拟内存**包括进程的代码和数据段、堆和共享库以及栈段。（从虚拟地址0x00000000到0xFFFFFFFF）
- **内核虚拟内存**包含内核中的代码和数据结构。内核虚拟内存的某些区域被映射到所有进程共享的物理页面，这一区域在虚拟内存和物理内存中都是连续的。
- **内核虚拟内存**的其他区域包含每个进程都不相同的数据，比如说页表和内核在进程的上下文中执行代码时使用的栈，以及记录虚拟地址空间当前组织的各种数据结构。
(P 580)

- Text:程序代码在内存中的映射，存放函数体的二进制代码。
- Data:在程序运行初已经对变量进行初始化的数据。
- BSS:在程序运行初未对变量进行初始化的数据。
- 栈: 位于用户空间顶端，耗尽栈内存区域后会触发page fault,大小低于RLIMIT_STACK（通常为8MB）时，由Linux内核函数分配空间，如果达到RLIMIT_STACK则会栈溢出，触发一个segmentation fault。
- 0x40000000:没有任何数据映射的虚拟地址空间，用于缓冲地址的溢出，历史原因选择了这个值。

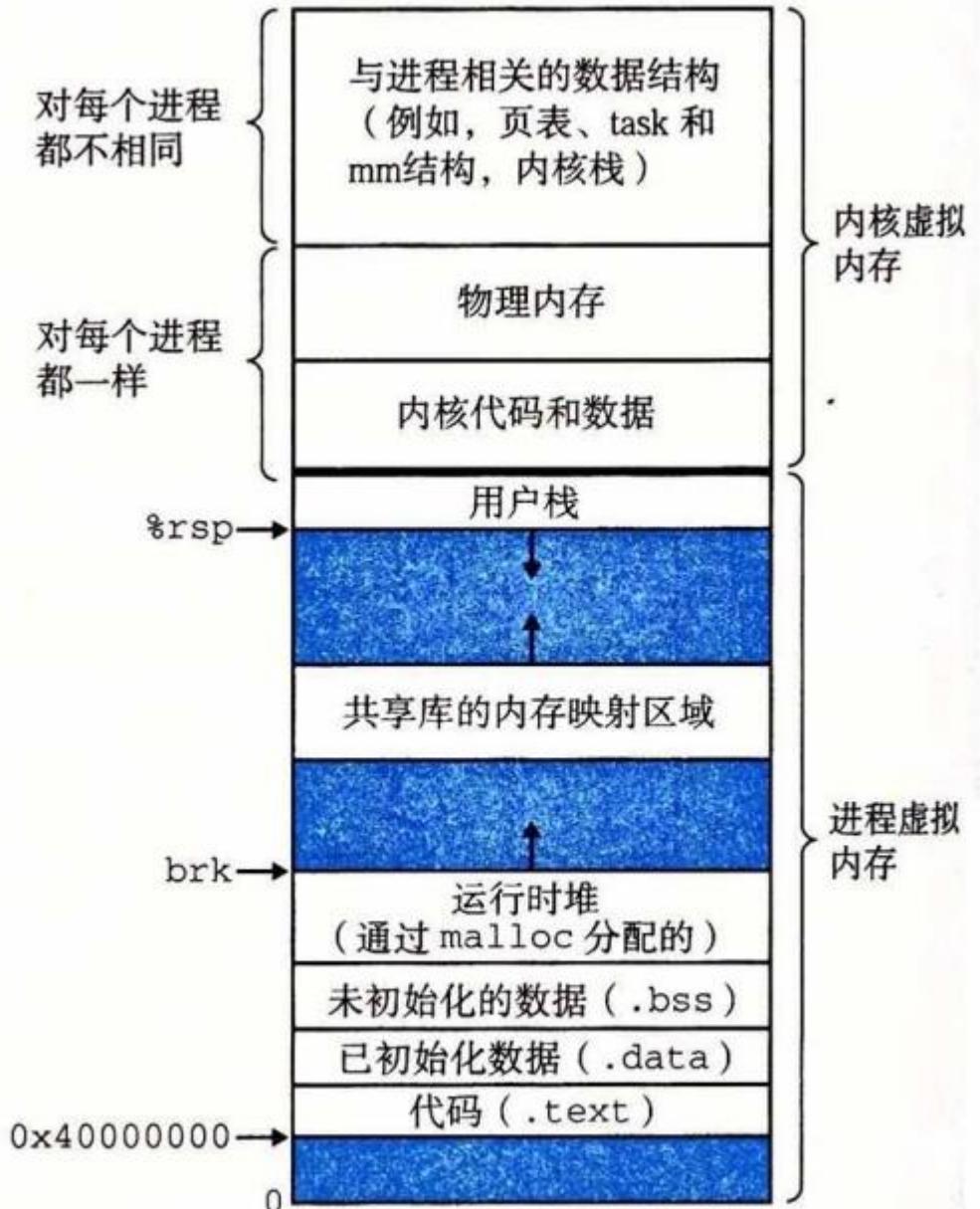


图 9-26 一个 Linux 进程的虚拟内存

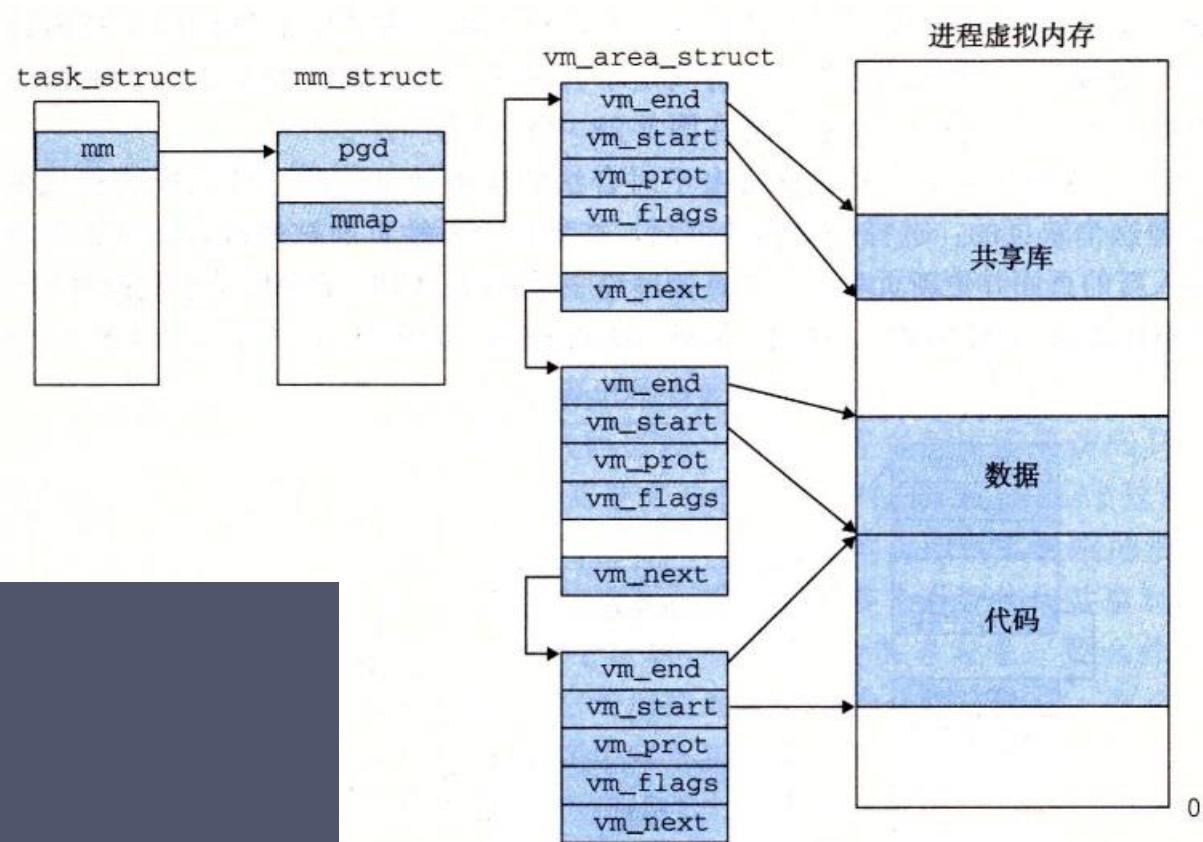
Linux 虚拟内存系统

每个进程有一个task_struct，包含PID、指向用户栈的指针、可执行目标文件的名字、以及程序计数器等。

pgd指向第一级页表(页全局目录)的基址。

mmap(与函数mmap区分)指向一个vm_area_structs(区域结构)的链表。

```
struct vm_area_struct {  
    struct mm_struct * vm_mm; /* 此vma所属的mm, 指向所属的内存描述符 */  
    unsigned long vm_start; /* 内存区域起始地址 */  
    unsigned long vm_end; /* 内存区域结束地址 */  
  
    /* 指向下一个vma */  
    struct vm_area_struct *vm_next;  
  
    pgprot_t vm_page_prot; /* vma访问权限 */  
    unsigned long vm_flags; /* vma属性比如: 只读, 读写, 可执行 */  
  
    struct rb_node vm_rb; /*vma结构组成的红黑树*/
```



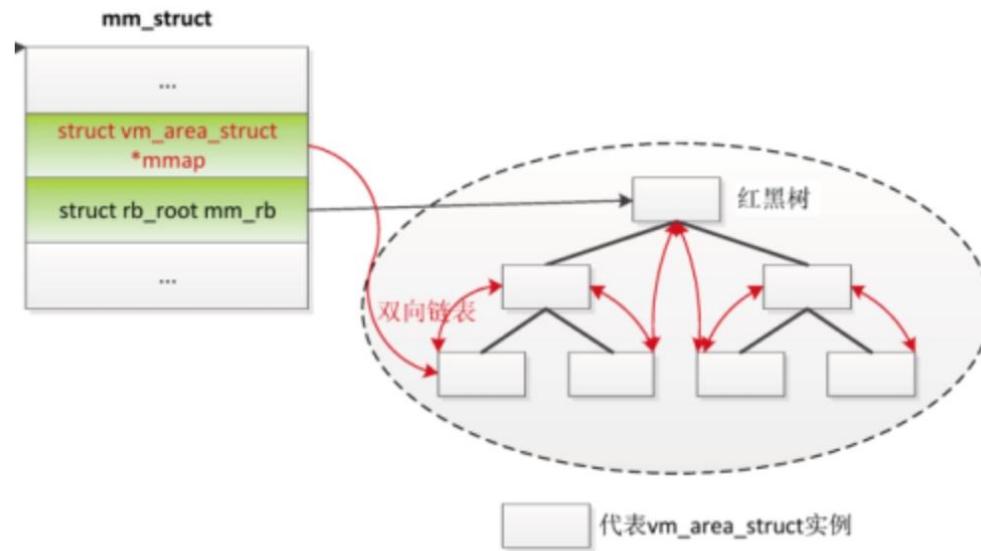
Linux 虚拟内存系统

More detailed

进程的虚拟内存空间会被分成不同的若干区域，每个区域都有其相关的属性和用途，一个合法的地址总是落在某个区域当中的，这些区域也不会重叠。在linux内核中，这样的区域被称之为虚拟内存区域(virtual memory areas),简称 **VMA**。

一个vma就是一块连续的线性地址空间的抽象，它拥有自身的权限(可读，可写，可执行等等)，每一个虚拟内存区域都由一个相关的
`struct vm_area_struct` 结构来描述。

Linux 通过双向链表在VMA实现了红黑树来管理`vm_area_struct`,大大提高了查找，插入，合并，删除的效率。



`vm_area_struct`结构体中主要是定义一个VMA的起始和结束的虚拟地址。

```
struct vm_area_struct {  
    unsigned long vm_start;      /* Our start address within vm_mm. */  
    unsigned long vm_end;        /* The first byte after our end address within vm_mm. */  
  
    /* linked list of VM areas per task, sorted by address */  
    struct vm_area_struct *vm_next, *vm_prev;  
  
    struct rb_node vm_rb;        /* 红黑树节点 */  
    struct mm_struct *vm_mm;     /* The address space we belong to. */ /* 此VMA属于的进程地址空间 */  
};
```

内存映射

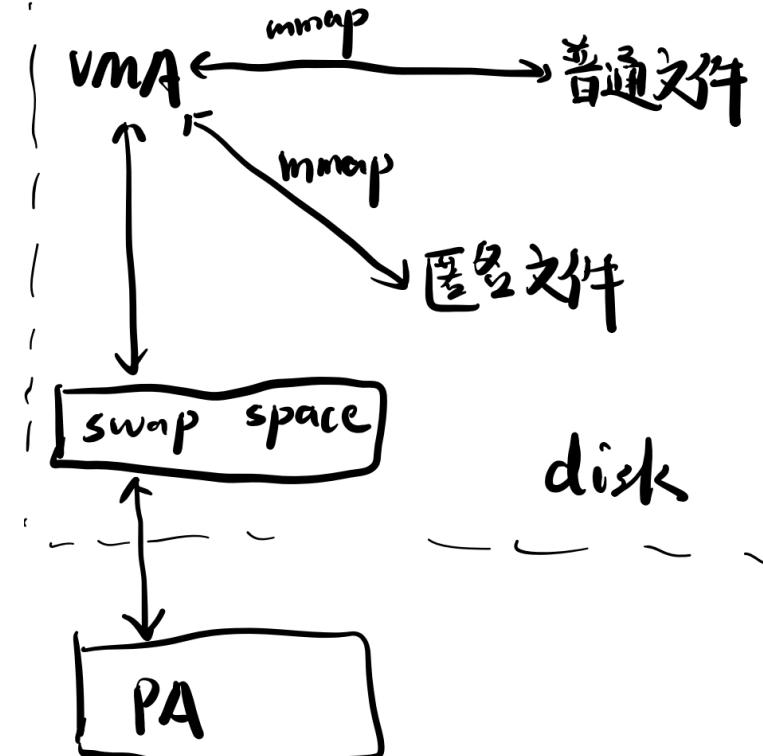
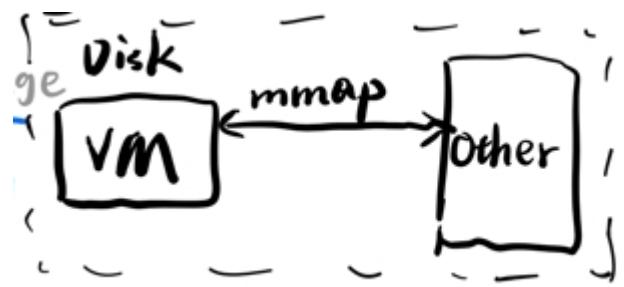
定义：将一个虚拟内存区域与一个磁盘上的对象关联起来，以初始化该虚拟内存区域的内容(磁盘->磁盘)

实现了将磁盘中未分配的页变为磁盘中已分配未缓存的页

普通文件和匿名文件都可以被映射到(或者说其实这是一个双射)VMA中，区别在于匿名文件映射后只是一个标记，不会占用额外磁盘空间。

只有在CPU第一次引用这个VMA的时候，它才会被交换入物理内存中，特别地，匿名文件VMA用二进制0覆盖物理内存中牺牲的页。

Swap space：类似于一个缓存(卫星发射中心？)，实际上只有在物理内存不足的时候它才会将物理内存中的页调出。



内存映射

——应用

execve

- 删除已存在的用户区域
- 映射私有区域
 - ✓ 代码、数据、.bss、栈
- 映射共享区域，
 - ✓ 共享对象（或目标）链接
- 设置程序计数器

BSS内存区域是匿名的，它不映射到任何文件（在程序运行时才会分配空间），数据段不是匿名的，它映射了源代码中指定了初始值的静态变量，它是一个私有内存映射，这意味着更改此处的内存不会影响被映射的文件。

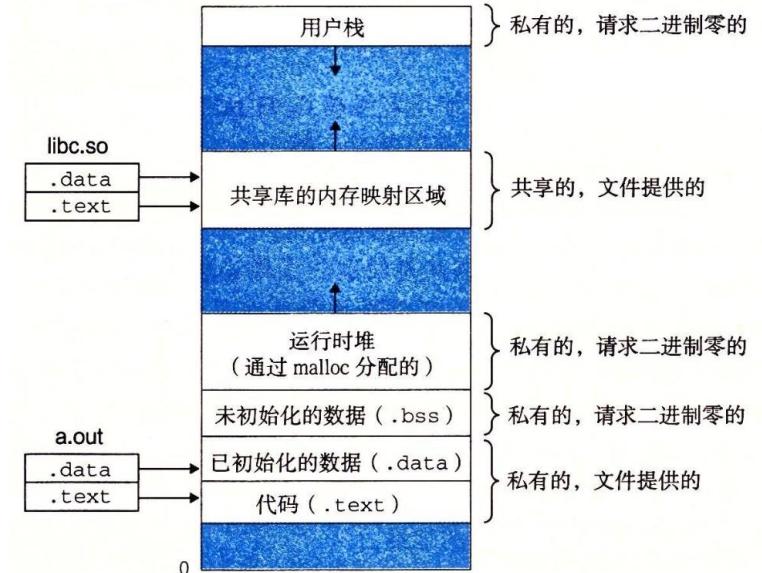


图 9-31 加载器是如何映射用户地址空间的区域的

内存映射

对于每个进程，对象有两类

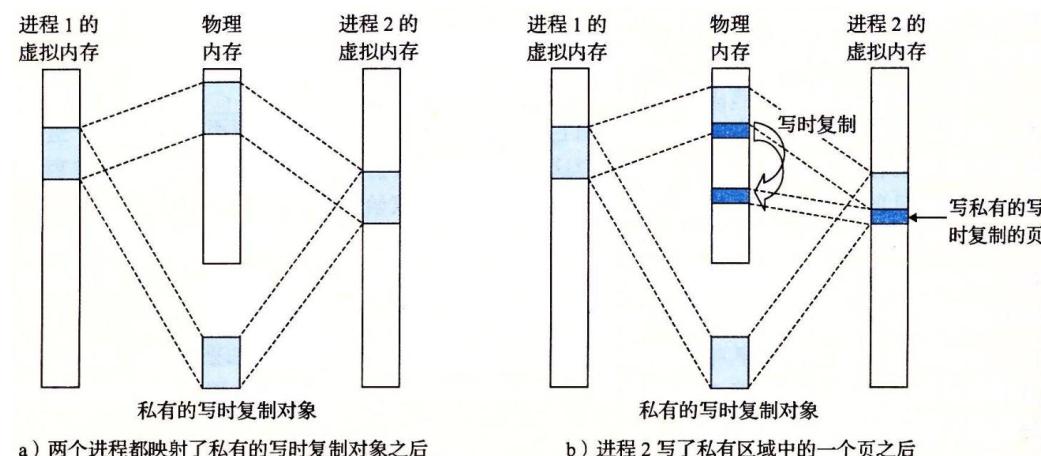
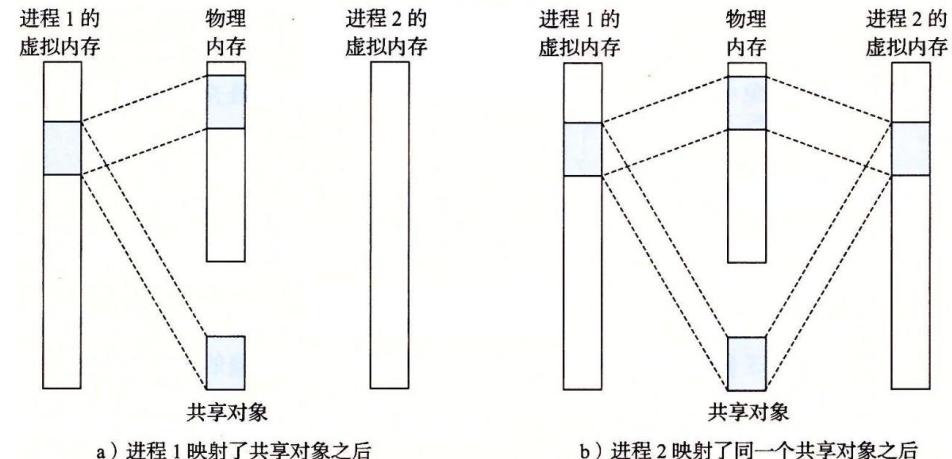
- 共享对象-共享区域：和其他共享这个对象的进程共享
- 私有对象-私有区域：对其他所有进程都不可见

共享对象的映射：

- 多个进程共享同一个对象
- 物理内存中只留一份，各个进程用自己的虚拟地址对应到它

私有对象的映射：

- COW: copy-on-write——保护故障
- 私有对象一开始是只读的当有进程尝试写的时候，被写的页面在物理内存中复制，新复制的页面标记为可写，并且与虚拟页面关联。既实现了私有性，又最大限度的保留了内存



内存映射

——应用

fork

- 为新进程创建数据结构，分配pid，创建当前区域结构的原样副本。
- 所有页都标记为只读、私有的写时复制
- 当任意一个进程进行写操作时，会创建新页面

Note:

在进程2结束之前，磁盘中的对象的都不会被修改

进程控制

- 创建进程fork()

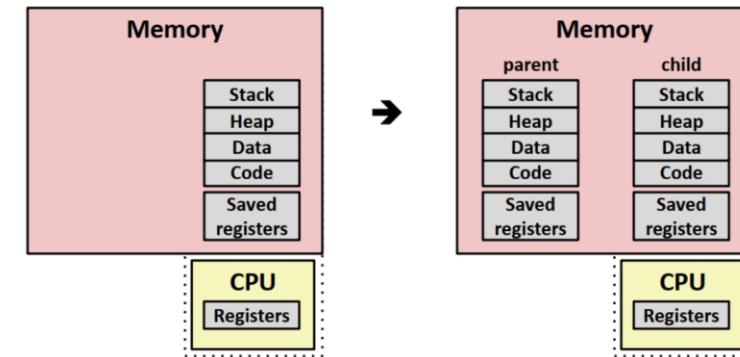
父进程调用fork函数创建一个新的运行的子进程。

- 子进程得到与父进程用户级虚拟地址空间相同且**独立**的一份副本
(代码、数据段、堆、共享库、用户栈)，并且**共享文件**，但是PID不相同。在父进程中，fork()返回子进程PID；在子进程中返回0。

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

返回：子进程返回 0，父进程返回子进程的 PID，如果出错，则为 -1。



内存映射

```
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

在进程的地址空间中创建映射区域

- start: 指定映射区域的首地址, 通常设为NULL, 让系统自动选择合适的地址。
- length: 映射区域的长度 (字节数)。
- prot: 保护标志, 指定对映射区域的访问权限, 如 PROT_READ、PROT_WRITE、PROT_EXEC等。
- flags: 控制映射区域的属性, 如 MAP_SHARED、MAP_PRIVATE等。
- fd: 文件描述符, 指定要映射的文件, 如果是匿名映射, 则传入 -1。
- offset: 文件映射的偏移量。

```
int munmap(void *addr, size_t length);
```

解除内存映射

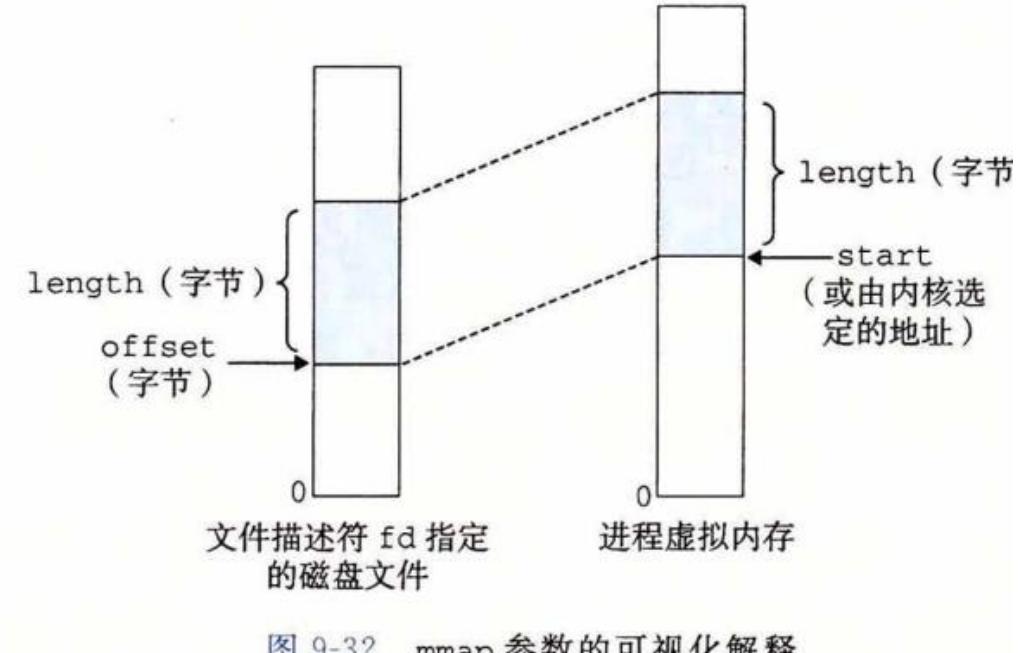


图 9-32 mmap 参数的可视化解释

内存映射

6分钟

A

注意程序运行的顺序
和内存映射COW的概念即可

程序打开文件描述符 `fd` 后，使用 `mmap` 函数创建了一块映射到文本 “`/input.txt`”所在区域的私有虚拟内存区域，其地址存储在指针 `bufp` 中。父进程创建完子进程后，首先等待子进程结束。子进程在试图对`*bufp`写入‘2’时，触发了一次 COW。它在物理内存中创建一个“`/input.txt`”文本所在页面的副本，并在新的页面里完成写入操作，写入完成后子进程 `bufp` 所在地址的第一个字节值为新写入的‘2’，第二个字节值为原始值‘2’。子进程结束后，其缓冲区内“21”的值正常输出。父进程在试图对`(bufp+1)`写入‘1’，同样触发了一次 COW，`bufp` 所在地址的第一个字节为原始值‘1’，第二个字节为新写入的‘1’，故父进程输出“11”。最后父进程重新创建一块映射到“`/input.txt`”所在页面的虚拟内存并输出。由于上述写入都是在新的物理页面下完成的，“`/input.txt`”所在页面没有发生修改，故最后一步父进程的输出为“12”。综上，最终在终端的输出为“221112”。 ↵

2. 阅读下列代码并回答选项。(已知文件“`input.txt`”中的内容为“12”，头文件没有列出) ↵

```
void *Mmap(void *addr, size_t length, int prot, int flags,  
           int fd, off_t offset);  
  
int main(){  
    int status;  
    int fd = Open("./input.txt", O_RDWR);  
    char* bufp = Mmap(NULL, 2, PROT_READ | PROT_WRITE,  
                      MAP_PRIVATE, fd, 0);  
  
    if (Fork()>0){  
        while(waitpid(-1, &status, 0)>0);  
        *(bufp+1) = '1';  
        Write(1, bufp, 2); // 1: stdout  
        bufp = Mmap(NULL, 2, PROT_READ, MAP_PRIVATE, fd, 0);  
        Write(1, bufp, 2);  
    }  
    else{  
        *bufp = '2';  
        Write(1, bufp, 2);  
    }  
}
```

在 shell 中运行该程序，正常运行时的终端输出应为 ↵

- A. 221112 B. 222121 C. 222112 D. 221111 ↵

Boomerang

虚拟内存技术的作用：

- 简化链接：独立的地址空间保证了每个进程的内存映像采用了相同的格式，这样的一致性极大简化了链接器的设计与实现
- 简化加载：只需要通过内存映射在VMA中构建一个进程，在CPU引用时在实际调入PM即可
- 简化共享，同样是应用内存映射实现

地址翻译

12分钟

流程: VA->TLE->page 1->page2->物理地址->内容

页大小4KB, 两级页表大小相同, 注意地址为20位,
推断出一个一级页表中包含4个二级页表

小端法

2. 考察虚拟内存地址翻译, 属于简单+中等题。 ↴

题目告知 TLB 未命中且无缺页, 因此需要访问页目录、二级页表和物理页, 共访问三次物理内存。 ↴

计算得 x 的 VPN1=1, VPN2=0, PPO=0x4d0。页目录地址在 0x00e66000, 因此相应页目录中所在条目地址为 $0x00e66000 + 1 * 4 = 0x00e66004$, 从表中按小端法读出 0x00615721, 因此二级页表存在, 首地址为 0x00615000, 这恰好也是相应二级页表中所在条目的地址, 从表中按小端法读出 0xc0ee2d21, 因此物理页存在, 首地址为 0xc0ee2000, 因此 x 的物理地址为 $0xc0ee2000 + 0x4d0 = 0xc0ee24d0$ 。从表中读出一个字节, 为 0x48。 ↴

参见课本第 9.6 和 9.7 节 (P567-582) ↴

31	12 [↓]	11	9 [↓]	8 [↓]	7 [↓]	6 [↓]	5 [↓]	4 [↓]	3 [↓]	2 [↓]	1 [↓]	0 [↓]
Address of 4KB page frame [↓]	Ignored [↓]	G [↓]	A [↓]	D [↓]	A [↓]	C [↓]	W [↓]	P [↓]	P [↓]	U [↓]	R [↓]	P [↓]
T [↓]	D [↓]	T [↓]	S [↓]	W [↓]	/ [↓]	P [↓]						

部分位的含义如下:

0 (P): 1 表示存在, 0 表示不存在 ↴

1 (R/W): 1 表示可写, 0 表示只读 ↴

2 (U/S): 1 表示内核模式, 0 表示用户模式 ↴

当系统运行到某一时刻时, TLB 有效位为 1 的条目如下 (未列出部分都是无效的): ↴

索引 (十进制) ↴	TLB 标记 [↓]	内容 [↓]
0 [↓]	0x0400 [↓]	0x0ec91313 [↓]
3 [↓]	0x02ff [↓]	0x5d2bac01 [↓]
5 [↓]	0xd551 [↓]	0x019fa42d [↓]
11 [↓]	0x55a6 [↓]	0xfd3c66b [↓]
13 [↓]	0x5515 [↓]	0xb591926b [↓]

一级页表的基址为 0x00e66000, 物理内存中的部分内容如下 (均为十六进制): ↴

地址 [↓]	内容 [↓]	地址 [↓]	内容 [↓]	地址 [↓]	内容 [↓]
00615000 [↓]	21 [↓]	00615001 [↓]	2d [↓]	00615002 [↓]	ee [↓]
00615003 [↓]	c0 [↓]	006154d0 [↓]	ff [↓]	006154d1 [↓]	a0 [↓]
00e66001 [↓]	a1 [↓]	00e66002 [↓]	a4 [↓]	00e66003 [↓]	67 [↓]
00e66004 [↓]	21 [↓]	00e66005 [↓]	57 [↓]	00e66006 [↓]	61 [↓]
00e66007 [↓]	00 [↓]	2167e000 [↓]	42 [↓]	2167e001 [↓]	67 [↓]
2167e002 [↓]	9a [↓]	2167e003 [↓]	7c [↓]	c0ee2000 [↓]	6f [↓]
c0ee2001 [↓]	d5 [↓]	c0ee2002 [↓]	7e [↓]	c0ee24d0 [↓]	48 [↓]
c0ee24d1 [↓]	83 [↓]	c0ee24d2 [↓]	ec [↓]	c0ee2d21 [↓]	11 [↓]
c0ee2d22 [↓]	6b [↓]	c0ee2d23 [↓]	82 [↓]	c0ee2d24 [↓]	8a [↓]

1. 将 cache 清空。访问一个在主存中的虚拟地址, TLB 命中, 没有触发缺页异常, 这一过程中, 需要访问物理内存 (主存) ____ 次 (3 分)。具体来说, 如果该虚拟地址为 $y = 0xd5515213$, y 地址所具有的实质权限是 _____ (多选题, 选对才得分 2 分)。 ↴

- ① 可写 ② 只读 ③ 用户模式权限 ④ 内核模式权限 ↴

2. 不考虑第一小问, 将 cache 清空。访问一个在主存中的虚拟地址, TLB 不命中, 没有触发缺页异常, 这一过程中, 需要访问物理内存 (主存) ____ 次 (3 分)。具体来说, 如果该虚拟地址为 $x = 0x004004d0$, 那么 x 对应的二级页表起始地址是 _____ (填写 16 进制, 例如 0x00123000, 2 分), x 地址上单字节的内容是 _____ (填写 16 进制, 例如 0x00, 1 分)。 ↴