

第9章 虚拟内存: 系统

教 师: 郑贵滨
计算机科学与技术学院
哈尔滨工业大学

主要内容

- 一个小内存系统示例
- 案例研究: Core i7/Linux 内存系统
- 内存映射

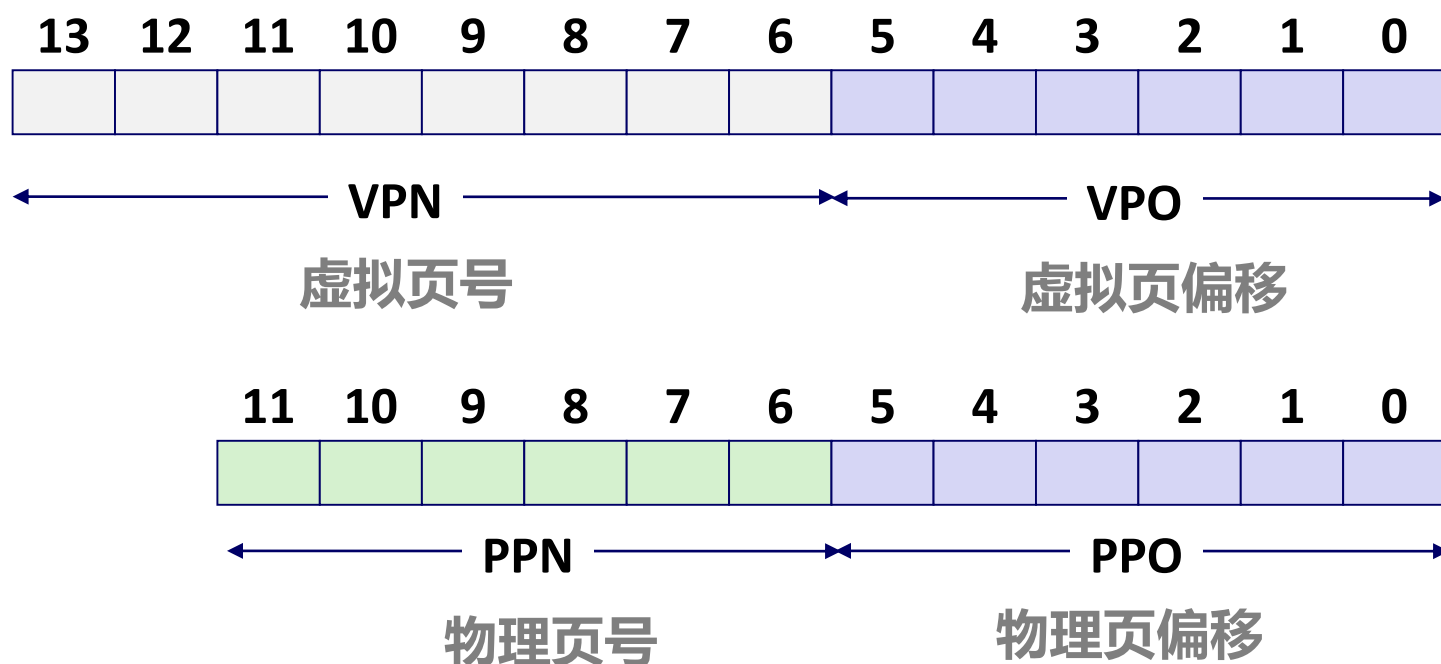
符号回顾Review of Symbols

- 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)
- 虚拟地址组成部分
 - TLBI: TLB(组)索引
 - TLBT: TLB 标记
 - VPO: 虚拟页面偏移量 (字节)
 - VPN: 虚拟页号
- 物理地址组成部分
 - PPO: 物理页面偏移量 (same as VPO)
 - PPN: 物理页号
 - CO: 缓冲块内的字节偏移量
 - CI: Cache 索引
 - CT: Cache 标记

一个小内存系统示例

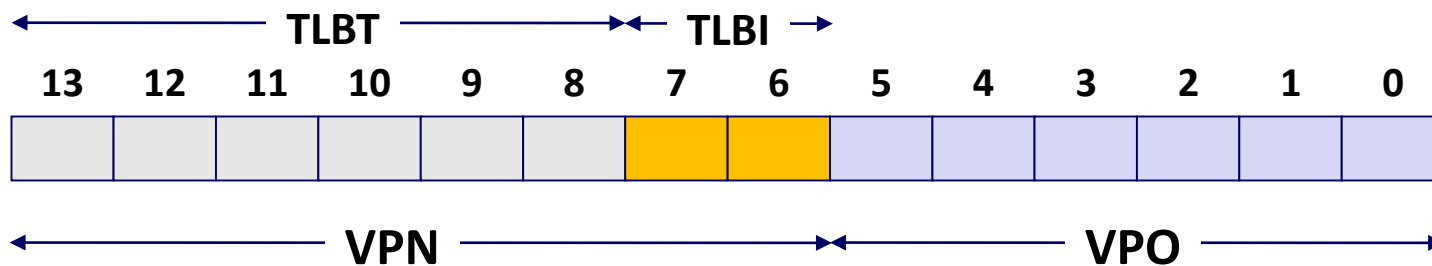
■ 地址假设

- 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个条目(16 entries)
- 4路组相联(4-way associative)
 - 4组，组索引2位



组Set	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	<i>Tag</i>	<i>PPN</i>	<i>Valid</i>	
	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

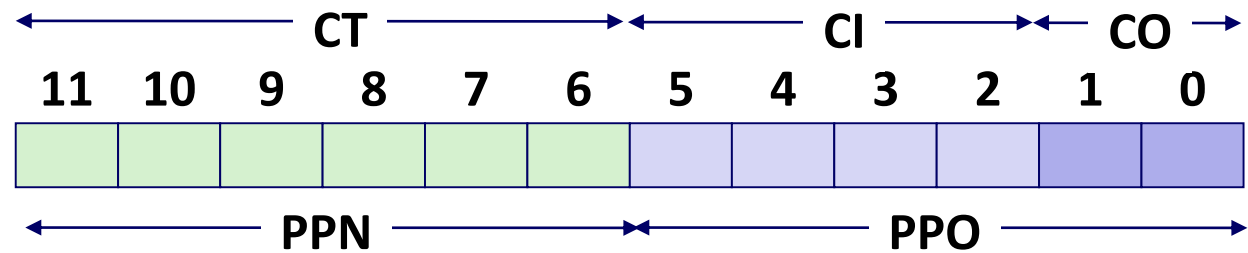
- 256个PTE中的前16个PTE：

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
00	28	1
01	—	0
02	33	1
03	02	1
04	—	0
05	16	1
06	—	0
07	—	0

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
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

- 共64字节:直接映射(每组1行/块),每行4字节,因此块内偏移量2位; 有16个组,因此组索引4位。
- 用物理地址寻址

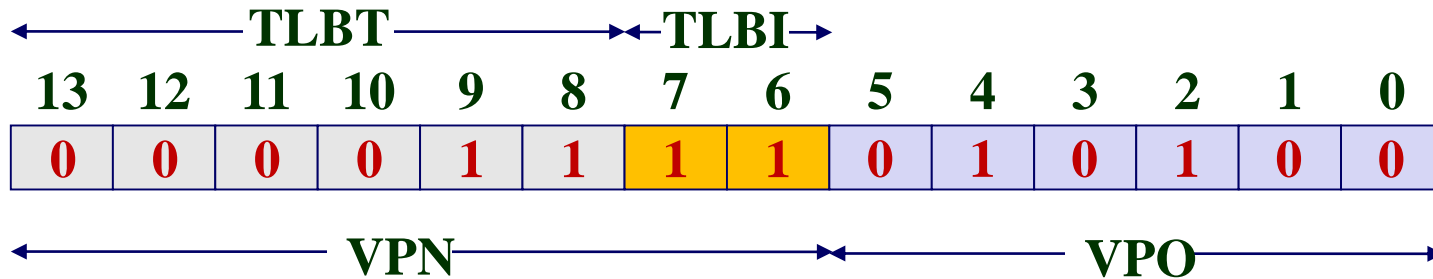


Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	—	—	—	—
2	1B	1	00	02	04	08
3	36	0	—	—	—	—
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	—	—	—	—
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
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	—	—	—	—

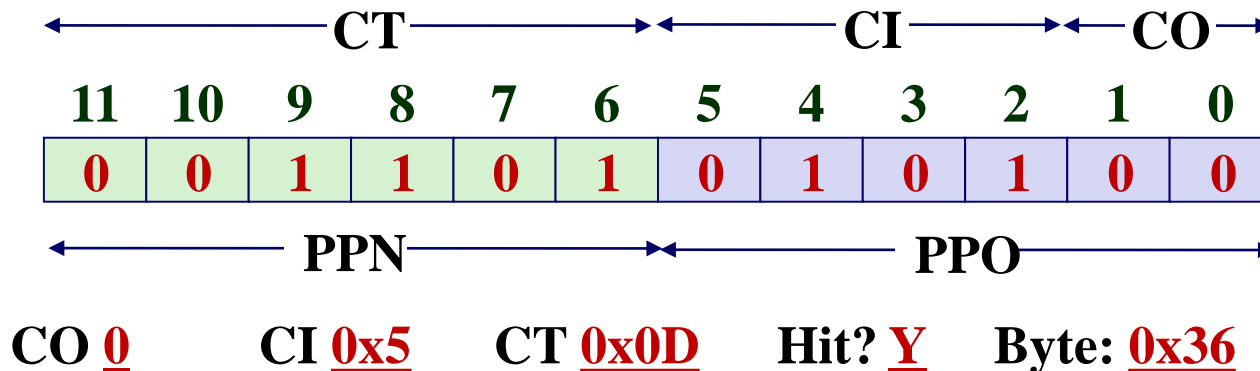
地址翻译 Example #1

虚拟地址: 0x03D4, 读取1字节数据



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

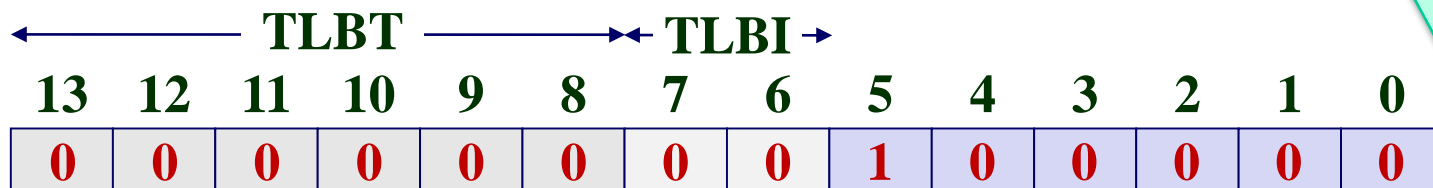
物理地址



地址翻译 Example #2

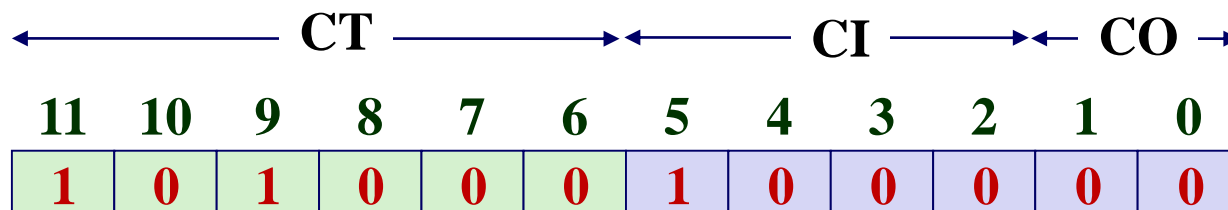
查页表：MMU根据页表首地址、VPN(PTE大小)获得PTEA，到缓存读取PTE，获得PPN，假设是0x28

虚拟地址: 0x0020，读取1字节数据



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

物理地址



CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem

主要内容

- 一个小内存系统示例
- 案例研究: Core i7/Linux 内存系统
- 内存映射

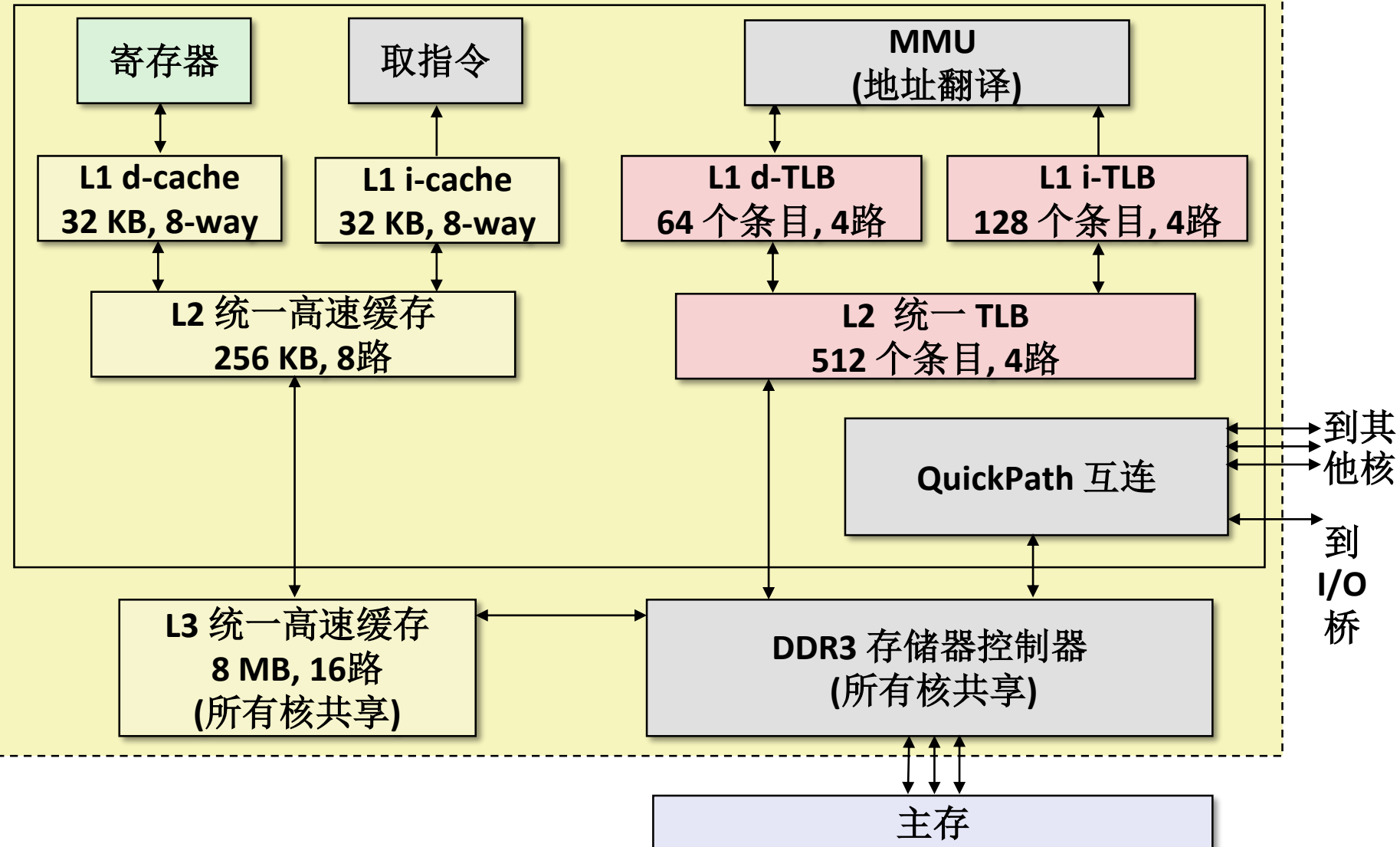
符号回顾Review of Symbols

- 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)
- 虚拟地址组成部分
 - TLBI: TLB(组)索引
 - TLBT: TLB 标记
 - VPO: 虚拟页面偏移量 (字节)
 - VPN: 虚拟页号
- 物理地址组成部分
 - PPO: 物理页面偏移量 (same as VPO)
 - PPN: 物理页号
 - CO: 缓冲块内的字节偏移量
 - CI: Cache 索引
 - CT: Cache 标记

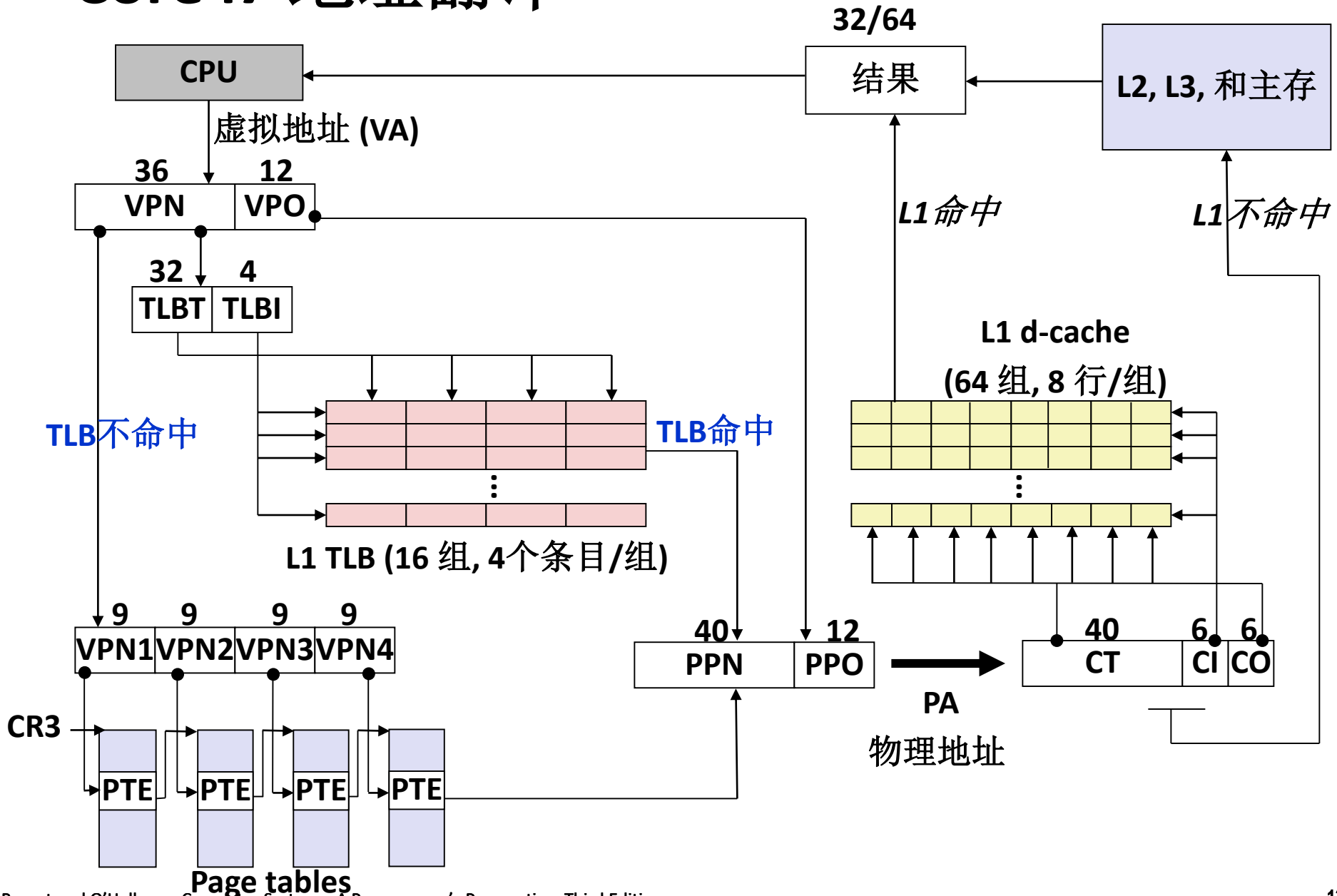
Intel Core i7 内存系统

寄存器封装

Core x4



Core i7 地址翻译



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user或超级用户supervisor(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)

页表物理基地址: 子页表物理基地址的最高40位 (强制物理页表 4KB 对齐)

XD: 能/不能从这个PTE可访问的所有页中取指令

Core i7 第 4 级页表条目格式

63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0
XD	未使用	页的物理基地址				未使用	G		D	A	CD	WT	U/SR/W	P=1	

OS可用 (页在磁盘上的位置page location on disk)

P=0

每个条目引用一个4KB的页:

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

SD: 能/不能缓存(Cache disabled or enabled)

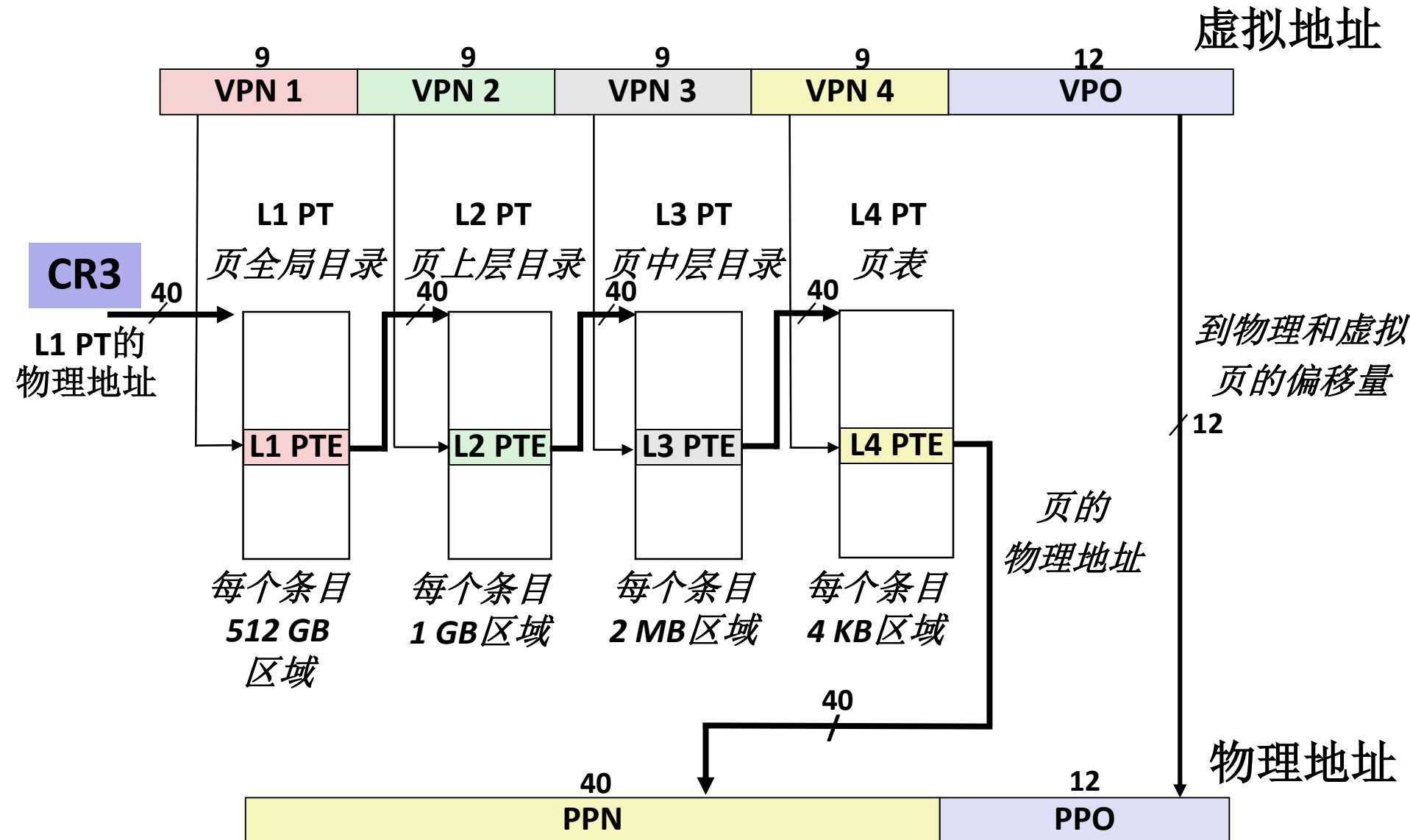
A: 引用位 (由MMU 在读或写时设置, 由软件清除)

D: 修改位(Dirty bit, 由MMU 在写时设置, 由软件清除)

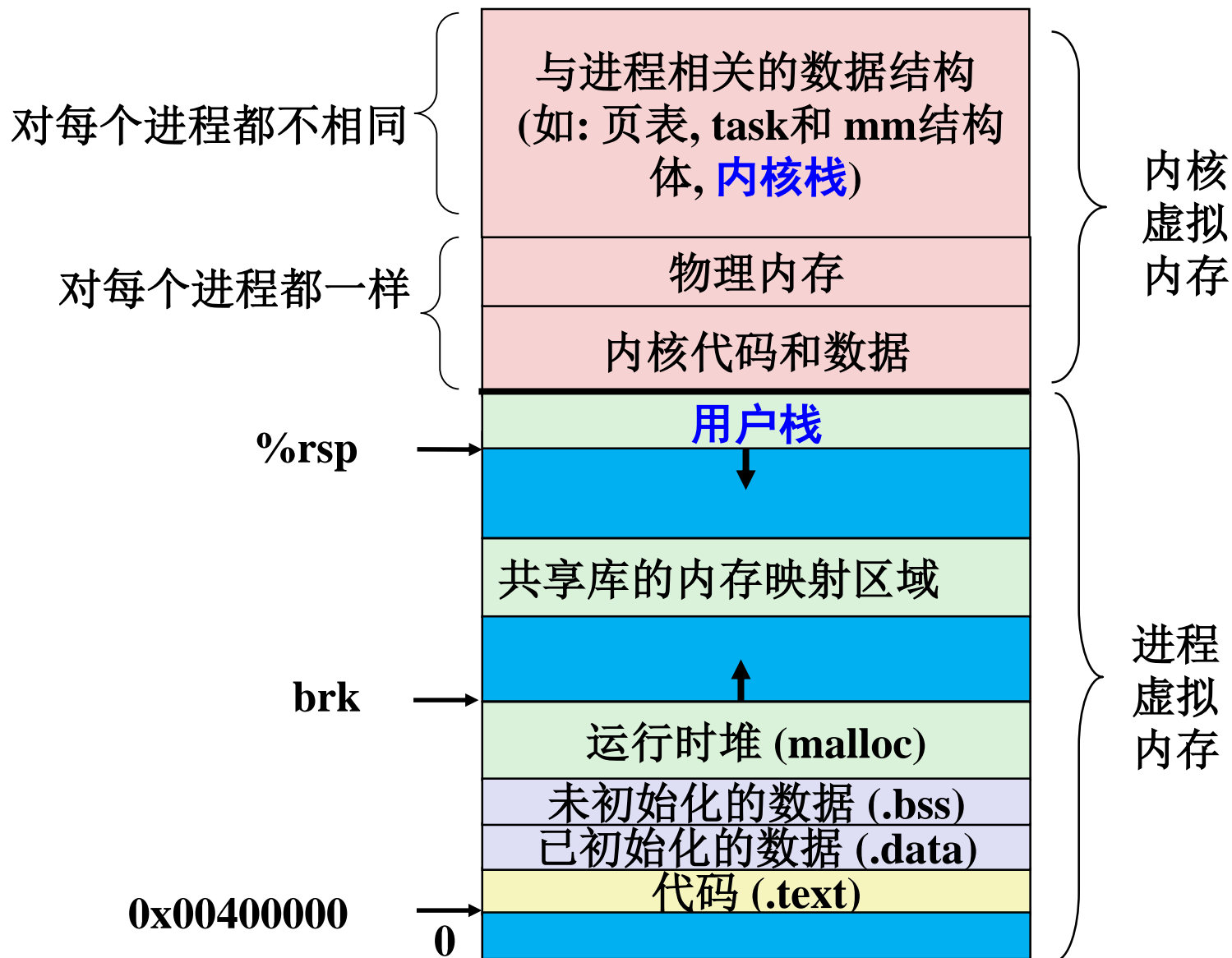
页表物理基地址: 物理页基地址的最高40位(强制物理页 4KB 对齐)

XD: 能/不能从这个PTE可访问的所有页中取指令

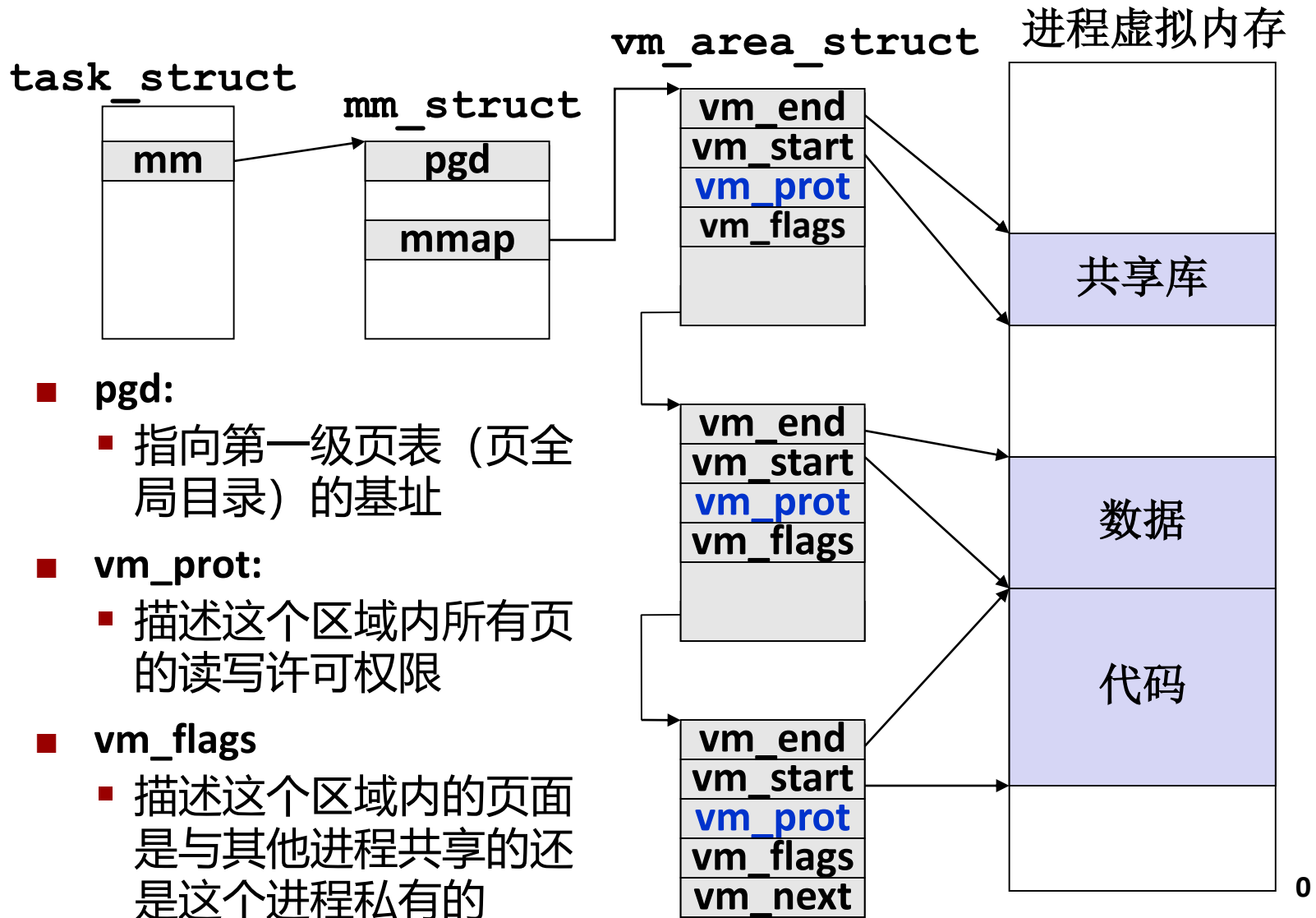
Core i7 页表翻译



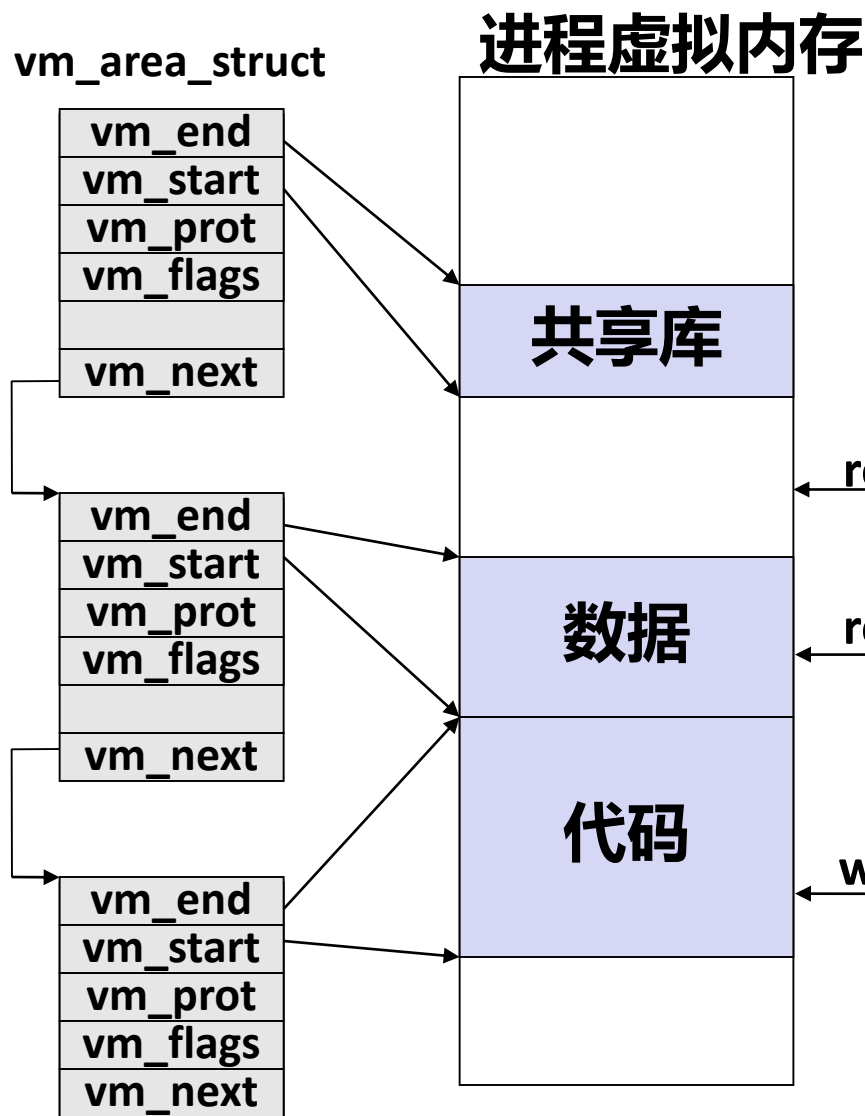
一个Linux 进程的虚拟地址空间



Linux将虚拟内存组织成一些区域的集合



Linux缺页处理



缺页——访问一个不存在的页面，缺页处理程序检查：

(1) 地址是否合法？

搜索区域链表，确认地址在(合法的)某个区域内？

否则，非法→段错误

(2) 访问是否合法？

有读、写或执行区域内页面的权限。否则，违反许可，触发保护异常→段错误

段错误： 访问不存在的页面
地址非法

正常缺页

保护异常：
写一个只读的页面

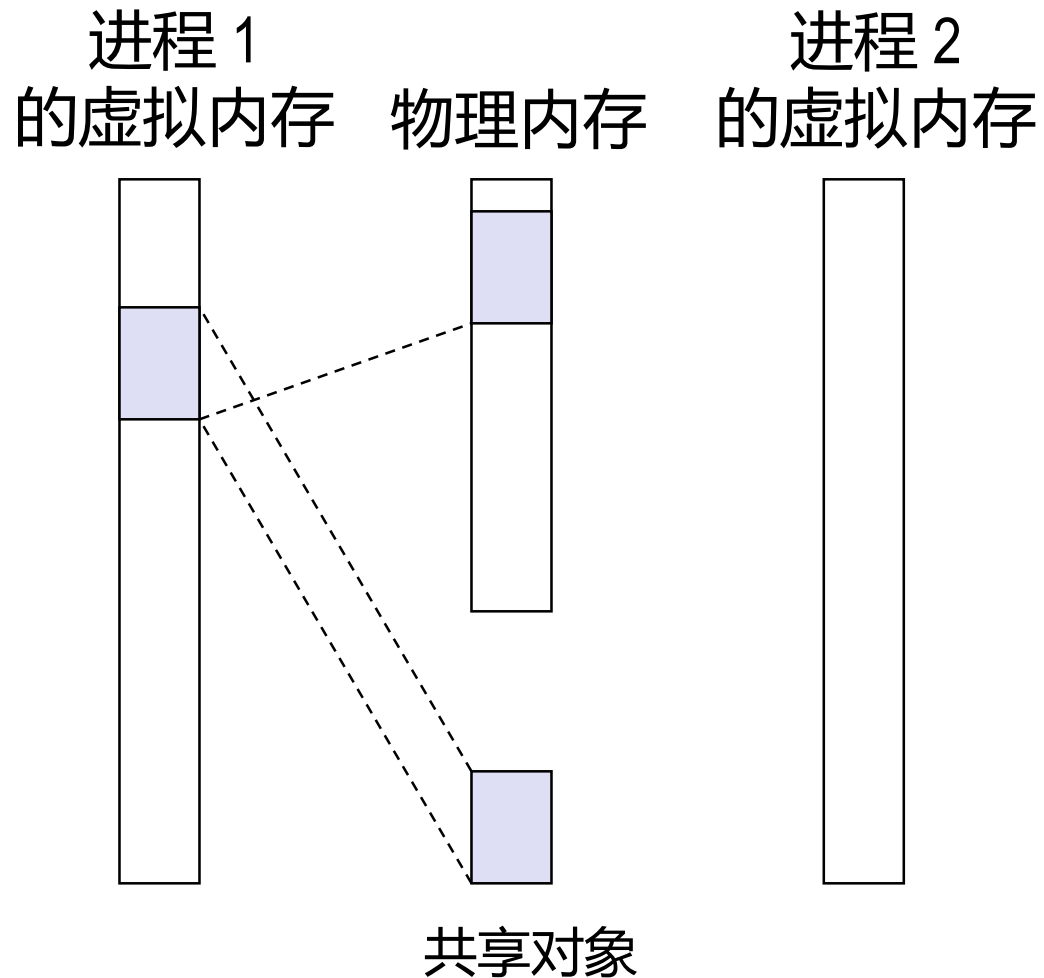
主要内容

- 一个小内存系统示例
- 案例研究: Core i7/Linux 内存系统
- 内存映射

内存映射

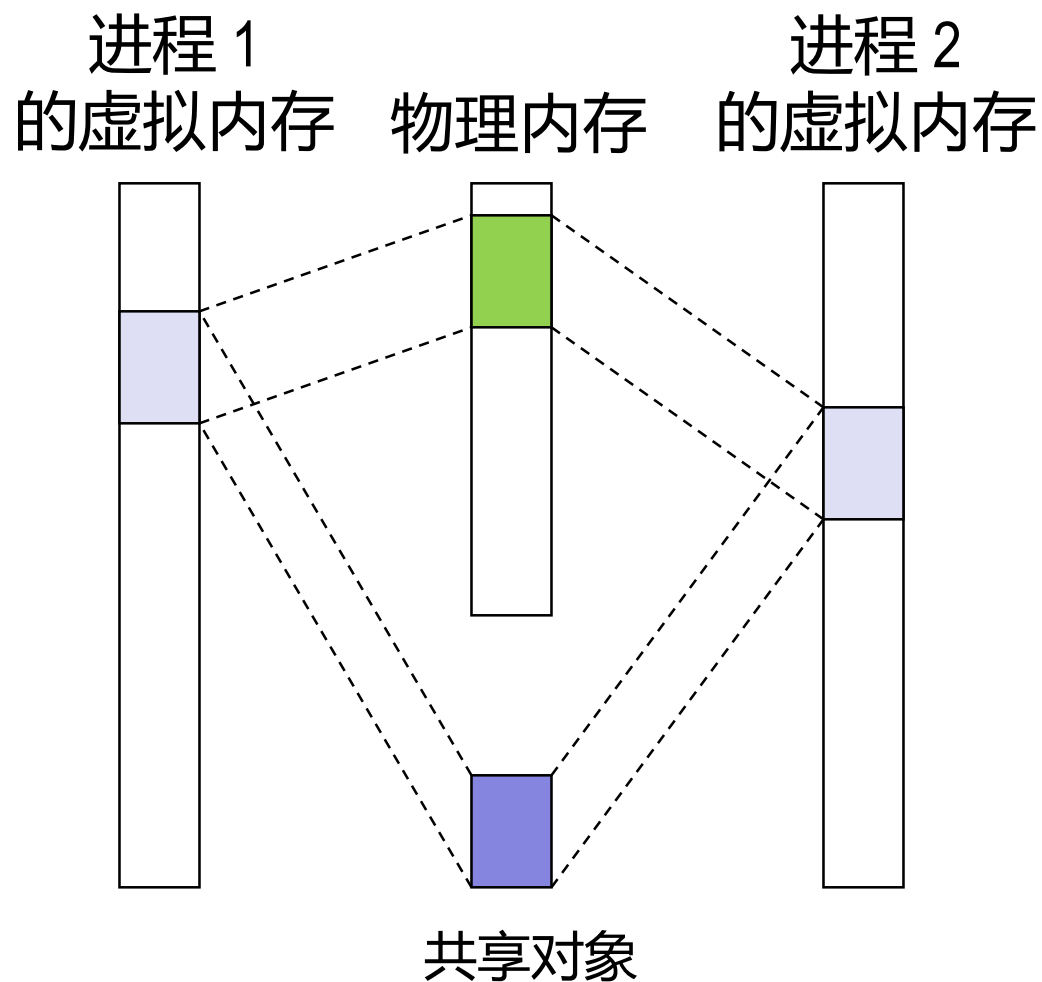
- 内存映射(*memory mapping*)
将虚拟内存区域与磁盘上的对象关联起来，用于初始化这个虚拟内存区域的内容。
- 虚拟内存区域可以映射的对象 (根据初始值的不同来源分):
 - 磁盘上的**普通文件** (e.g., 一个可执行目标文件)
 - 文件区被分成页大小的片，对虚拟页面初始化
 - **匿名文件** (内核创建，全是二进制零)
 - 首次访问该区域的虚拟页会引发缺页异常→分配一个全零的物理页 (**demand-zero page请求二进制零的页**)
 - 一旦该页面被修改，即和其他页面一样
- 初始化后的页面在内存和交换文件(*swap file*)之间换来换去

再看共享对象



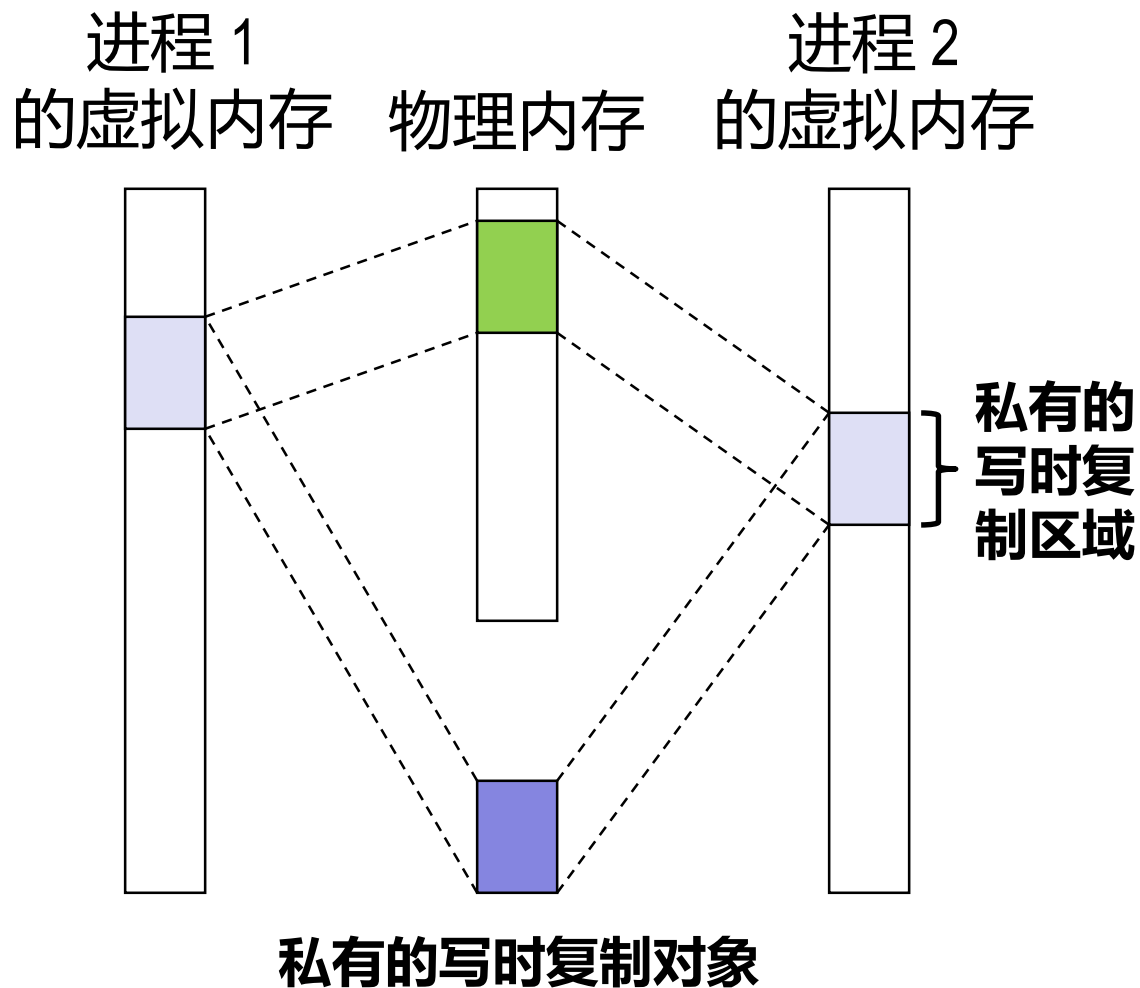
- 进程 1 映射了共享对象

再看共享对象



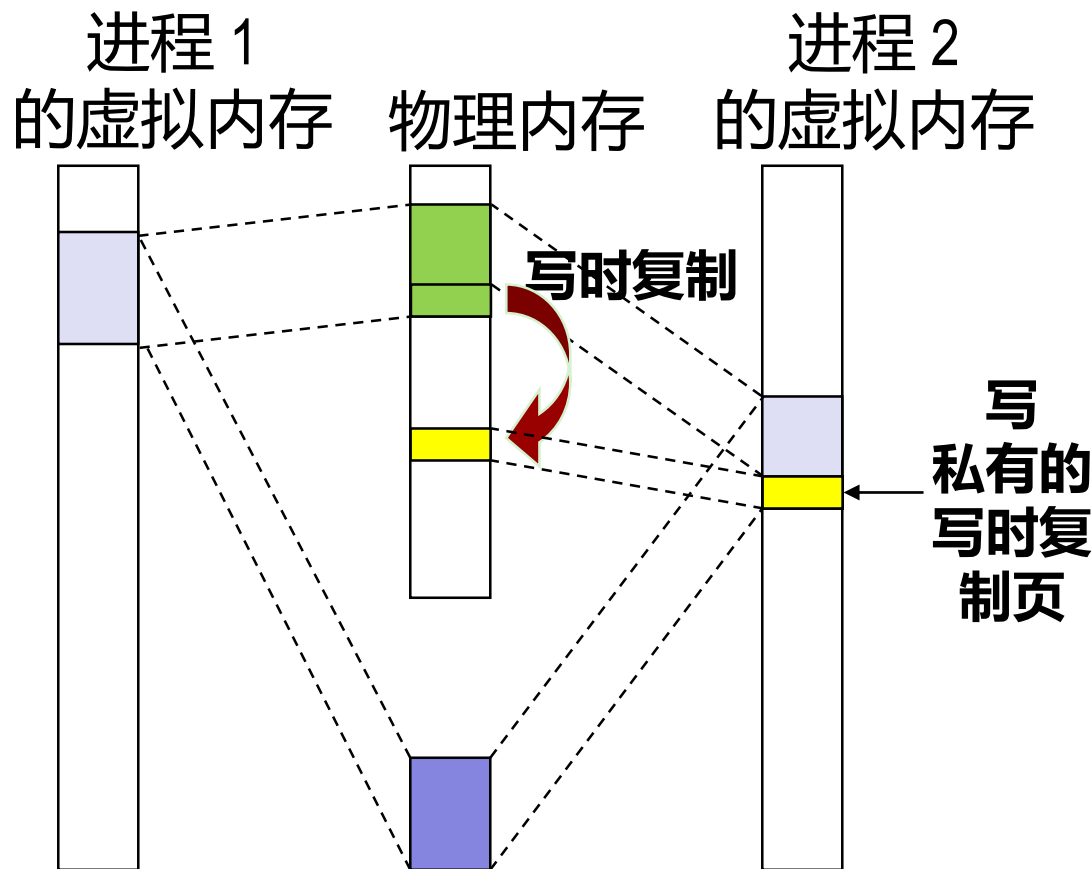
- 进程 2 映射了同一个共享对象。
- 两个进程的虚拟地址可以是不同的。

共享对象: 私有的写时复制对象



- 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

共享对象: 私有的写时复制对象



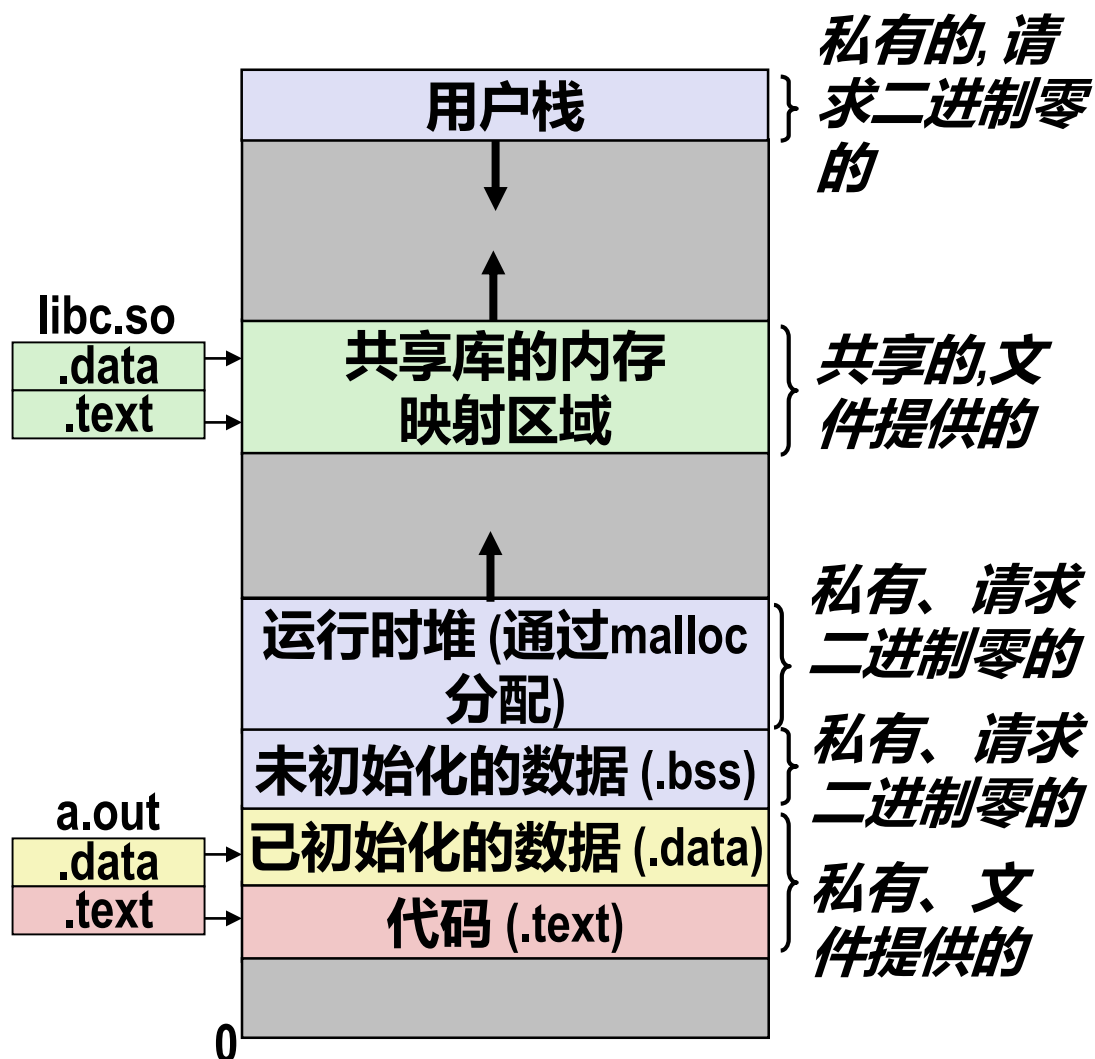
私有的写时复制对象

- 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
- 能尽可能地延迟拷贝（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mm_struct、vm_area_struct链表和页表的原样副本。
 - 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的父进程相同的虚拟内存
 - 随后的写操作会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:

- 删除已有页表和结构体 `vm_area_struct` 链表
- 创建新的页表和结构体 `vm_area_struct` 链表
 - 代码和初始化的数据映射到 `.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

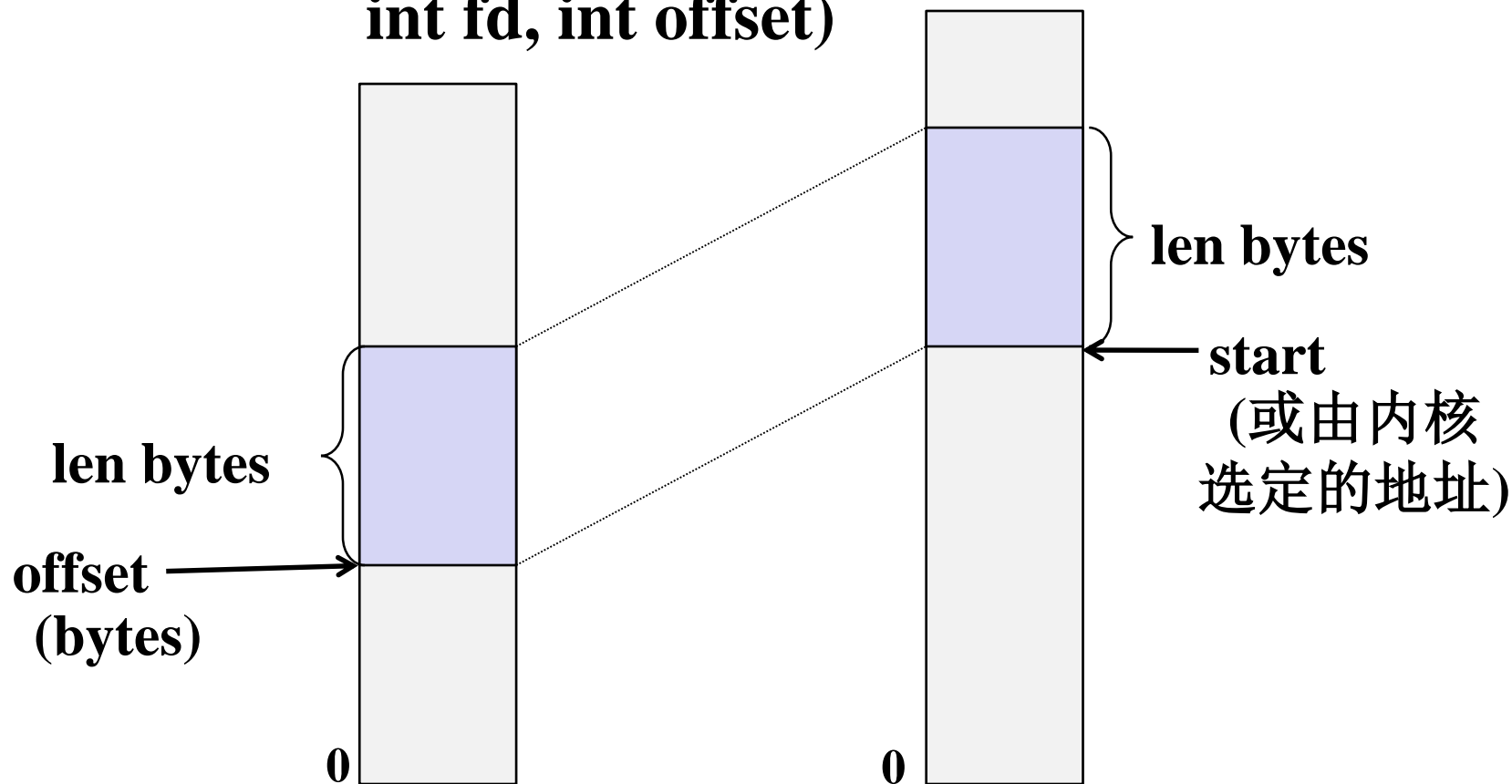
用户级内存映射

void *mmap(void *start, int len, int prot, int flags,
int fd, int offset)

- 从fd指定磁盘文件的offset处，映射len个字节到一个新创建的虚拟内存区域，该区域从地址start处开始
 - start: 虚拟内存的起始地址，通常定义为NULL
 - prot: 虚拟内存区域的访问权限，PROT_READ(可读)，PROT_WRITE(可写)，PROT_EXEC(可执行)，PROT_NONE (不能被访问)
 - flags: 被映射对象的类型，MAP_ANON (匿名对象)，MAP_PRIVATE (私有的写时复制对象)，MAP_SHARED (共享对象)，...
- 返回值：指向映射区域开始处的指针
- **execve**用类似函数将磁盘文件映射到进程的虚拟地址空间

用户级内存映射

```
void *mmap(void *start, int len, int prot, int flags,  
           int fd, int offset)
```



文件描述符 `fd` 指定的磁盘文件

进程虚拟内存

Example: 使用 mmap 函数拷贝文件

■ 拷贝一个文件到 stdout (数据没有传输到用户空间)

```
#include "csapp.h"
```

```
void mmapcopy(int fd, int size)
{
```

```
/* Ptr to memory mapped area */
char *bufp;
```

```
bufp = Mmap(NULL, size,
            PROT_READ,
            MAP_PRIVATE,
            fd, 0);
```

```
Write(1, bufp, size);
return;
```

```
}
```

```
//mmapcopy.c
```

```
/* mmapcopy driver */
```

```
int main(int argc, char **argv){
    struct stat myfilestat;
    int fd;
```

```
/* Check for required cmd line arg */
if (argc != 2) {
    printf("usage: %s <filename>\n",
           argv[0]);
    exit(0);
}
```

```
/* Copy input file to stdout */
```

```
fd = Open(argv[1], O_RDONLY, 0);
Fstat(fd, & myfilestat);
mmapcopy(fd, myfilestat.st_size);
exit(0);
```

```
}
```

```
//mmapcopy.c
```