第9章 虚拟内存: 系统

主要内容

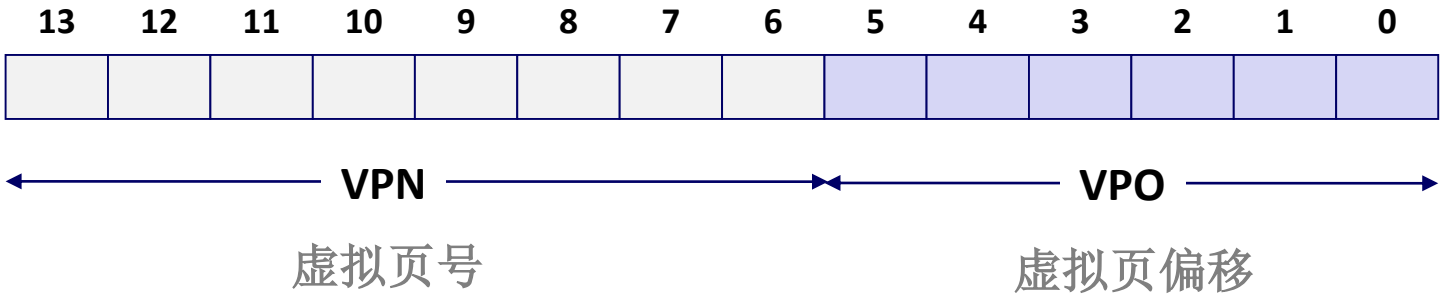
- 一个小内存系统示例
- 案例研究: Core i7/Linux 内存系统
- 内存映射

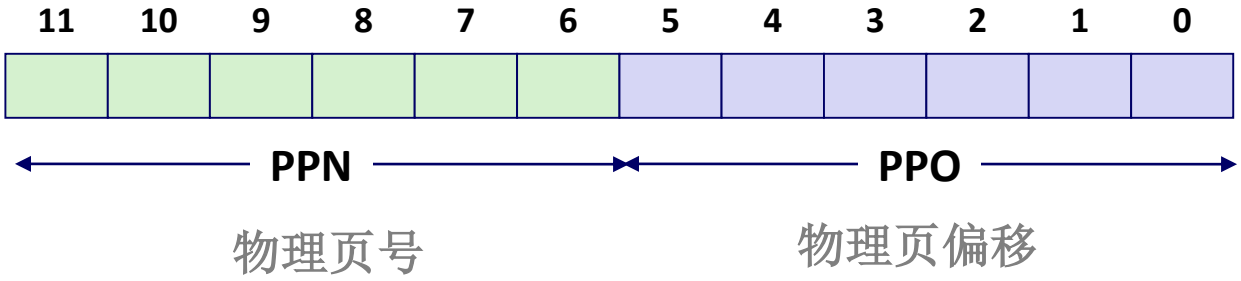
Review of Symbols符号回顾

- 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

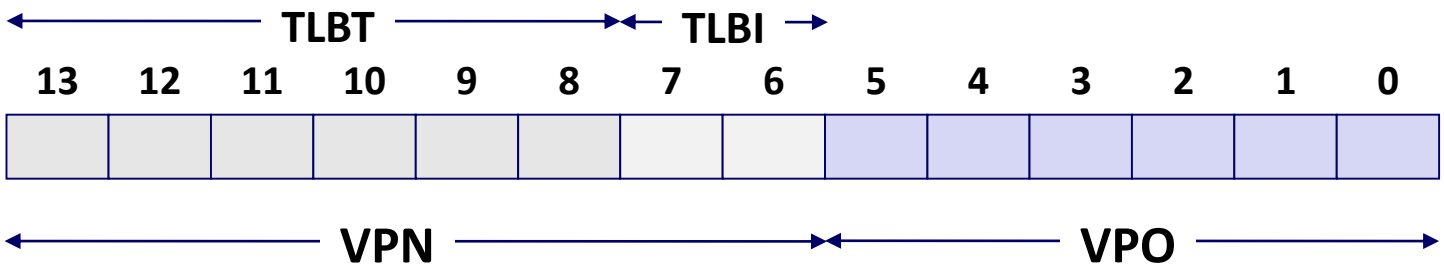
- 地址假设
  - 14位虚拟地址（n=14）
  - 12位物理地址（m = 12）
  - 页面大小64字节（P = 64）





1. 小内存系统的 TLB

- 16 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

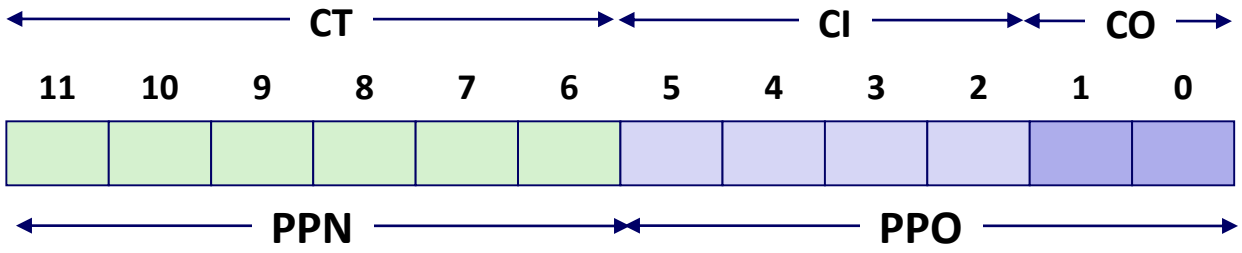
只展示了前16个PTE (out of 256)

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

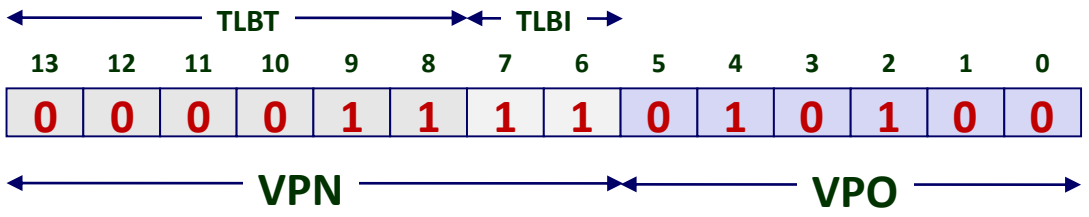


索引	标记位	有效位	块0	块1	块2	块3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

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	–	–	–	–

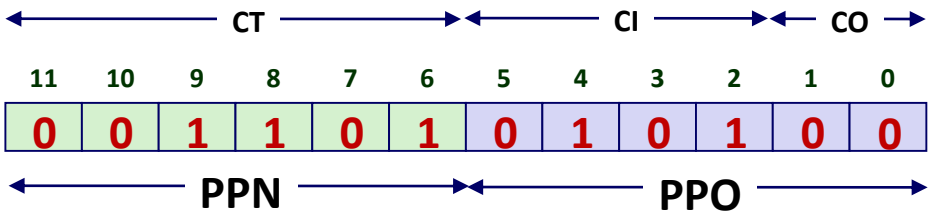
地址翻译 Example #1

虚拟地址: 0x03D4



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

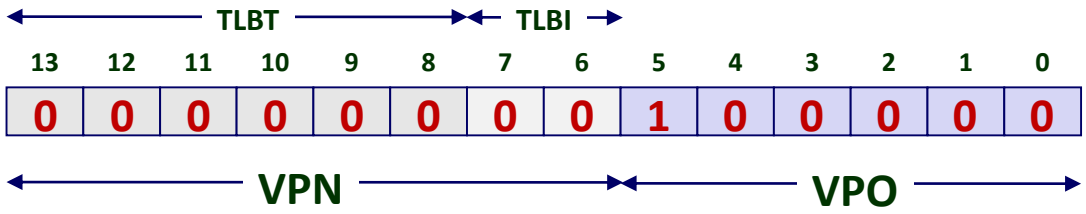
物理地址



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

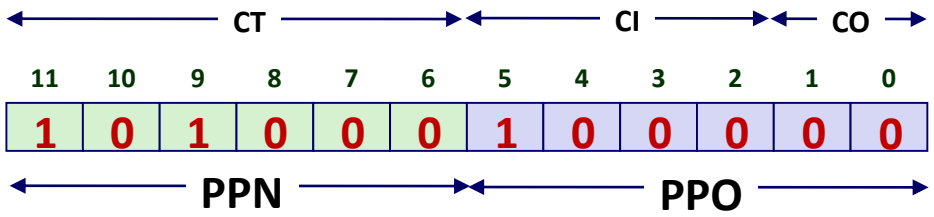
地址翻译 Example #2

虚拟地址: 0x0020



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

物理地址

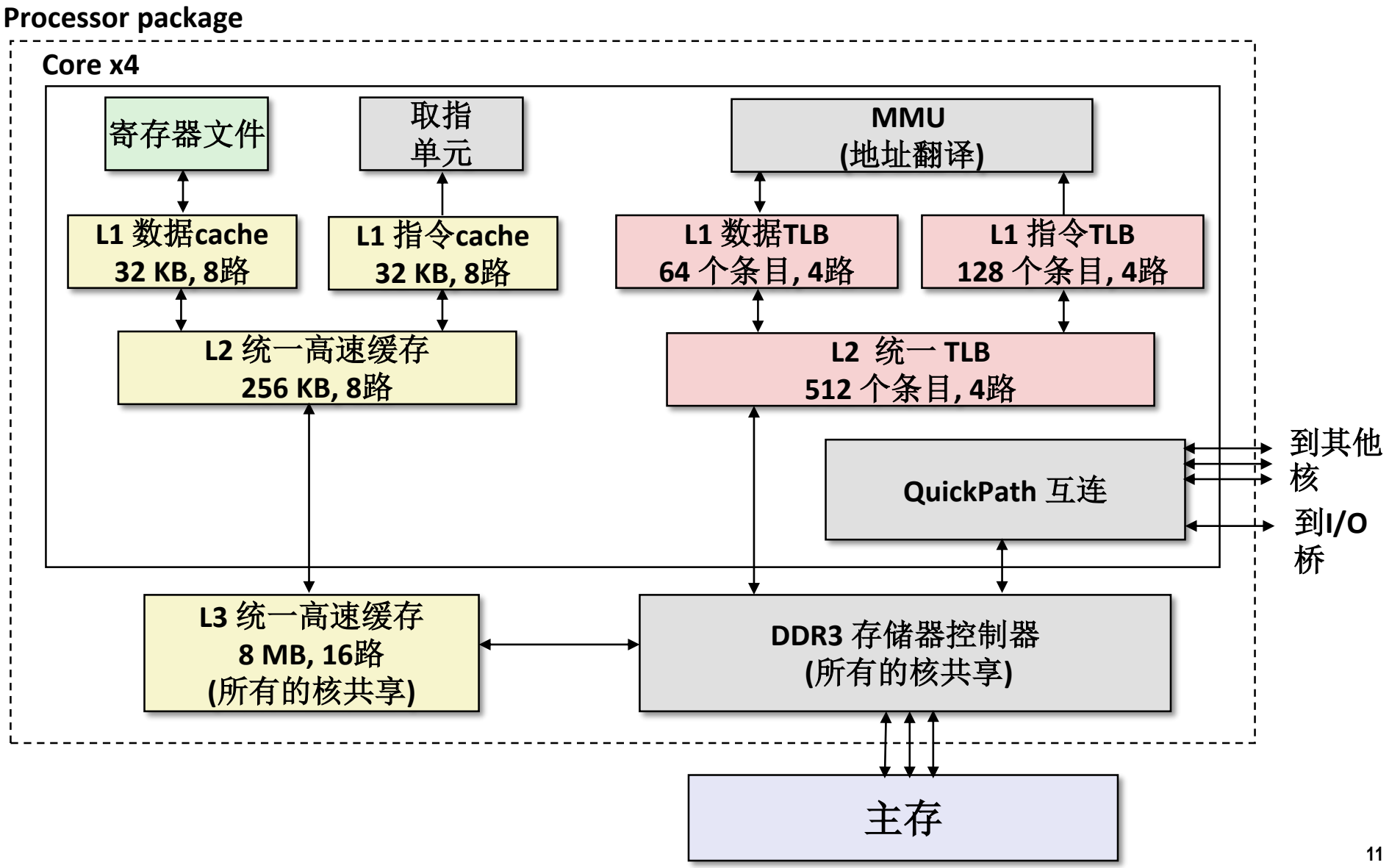


CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem

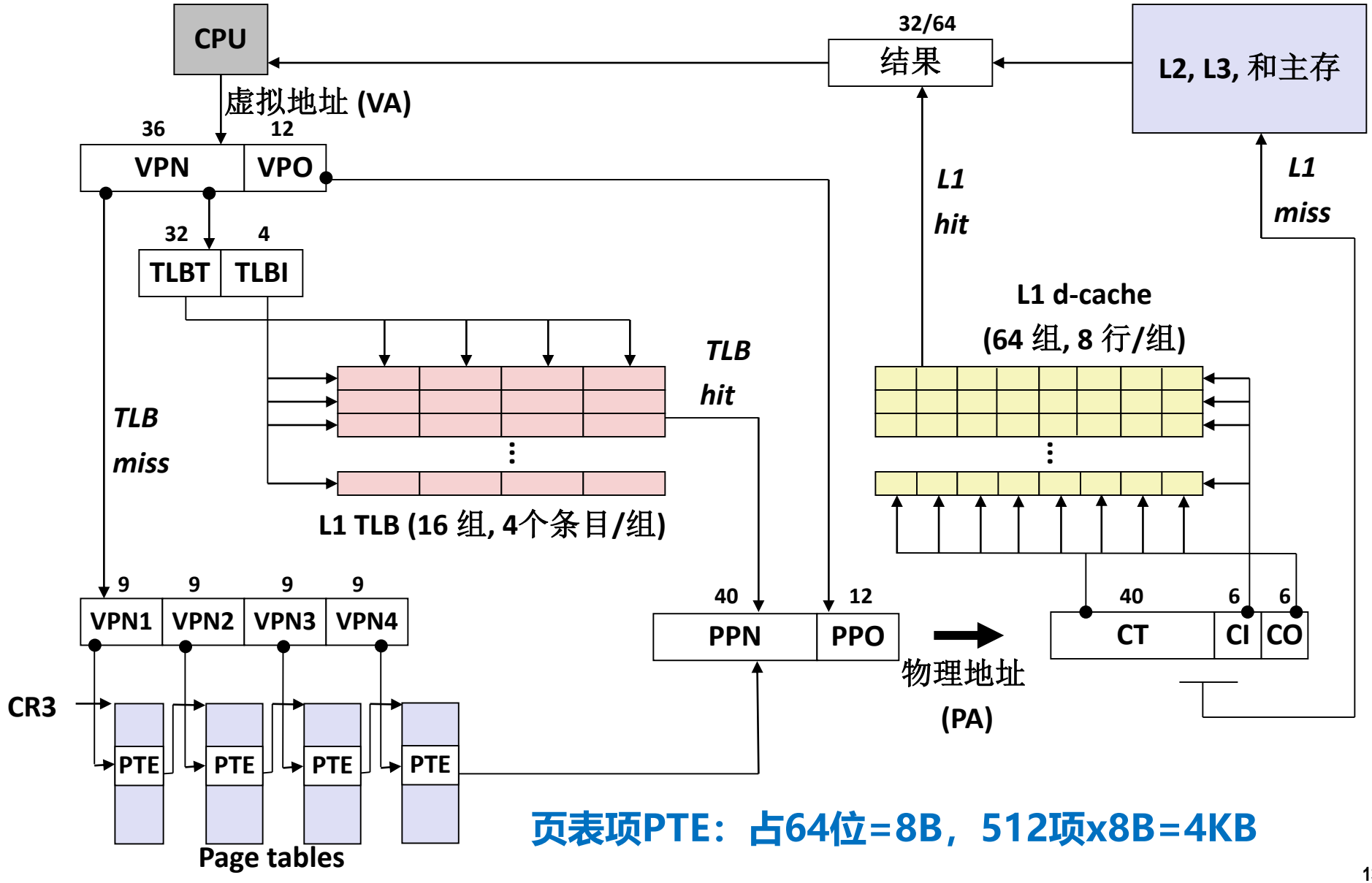
主要内容

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

Intel Core i7 内存系统

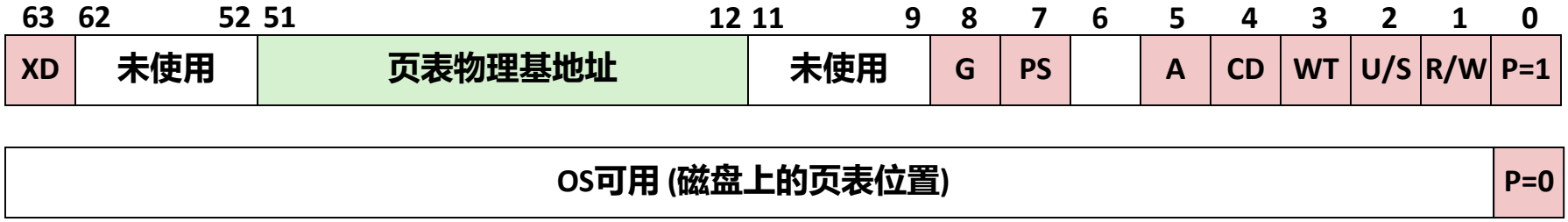


Core i7 地址翻译（VA48位PA52位）



页表项PTE：占64位=8B，512项x8B=4KB

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).

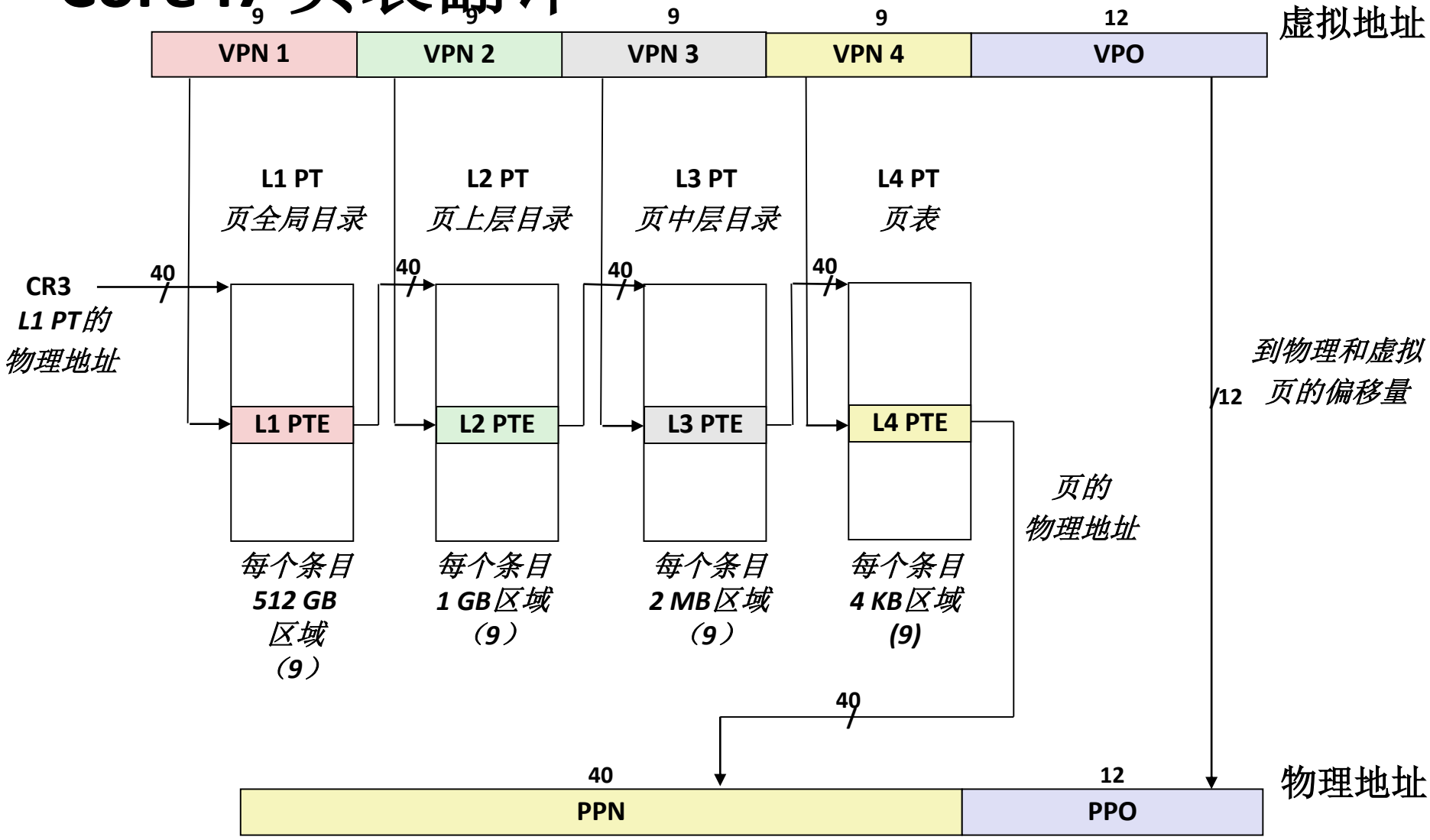
D: 修改位 (由MMU 在读和写时设置，由软件清除)

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.

对照书
p578

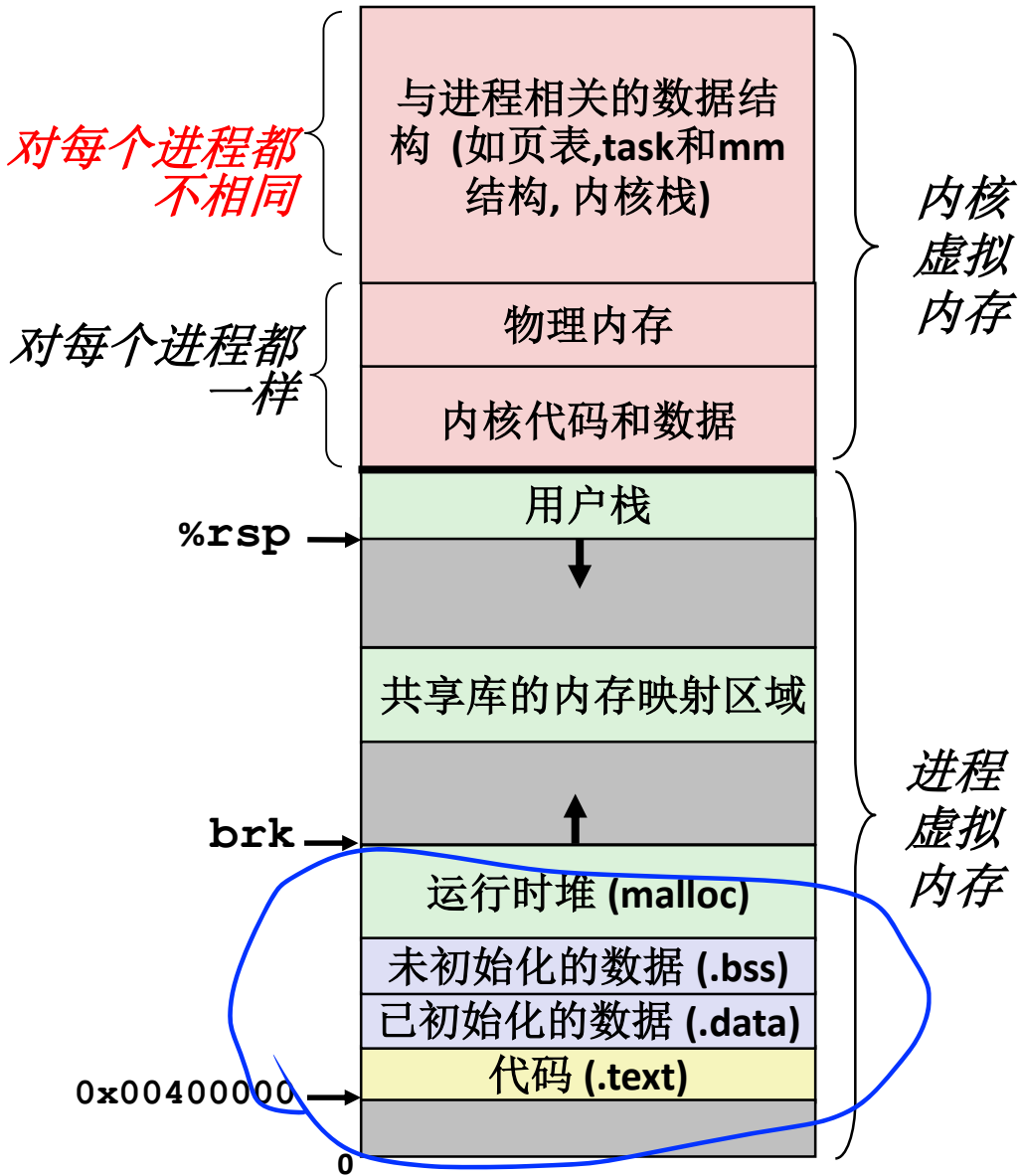
Core i7 页表翻译



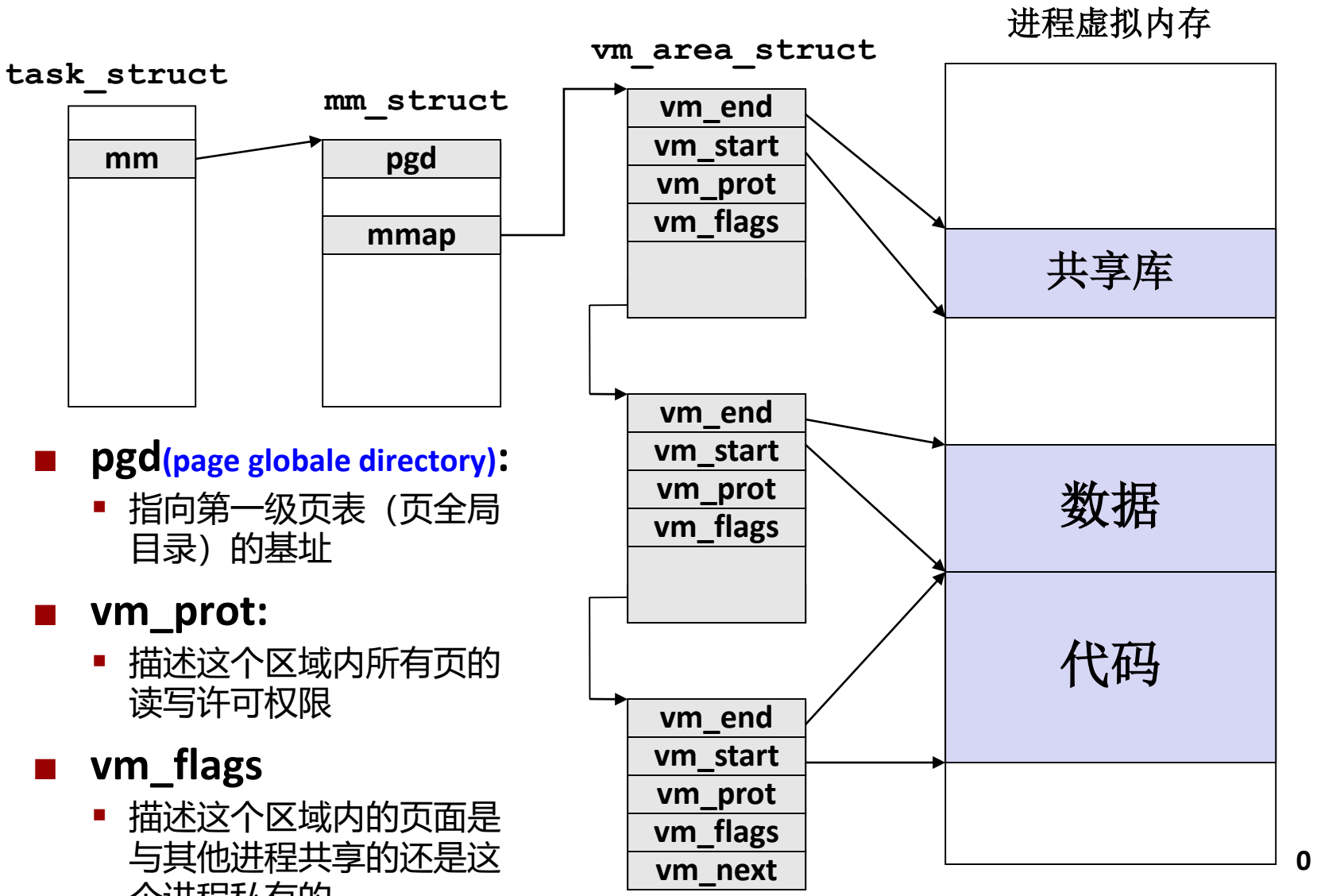
前面都是40位的页框地址，每一级的表大小也是4KB，因为包含2^9个8B表项

注意：每个PTE（8B）都藏着关键的40位页框地址指向下一级页框

一个Linux 进程的虚拟地址空间

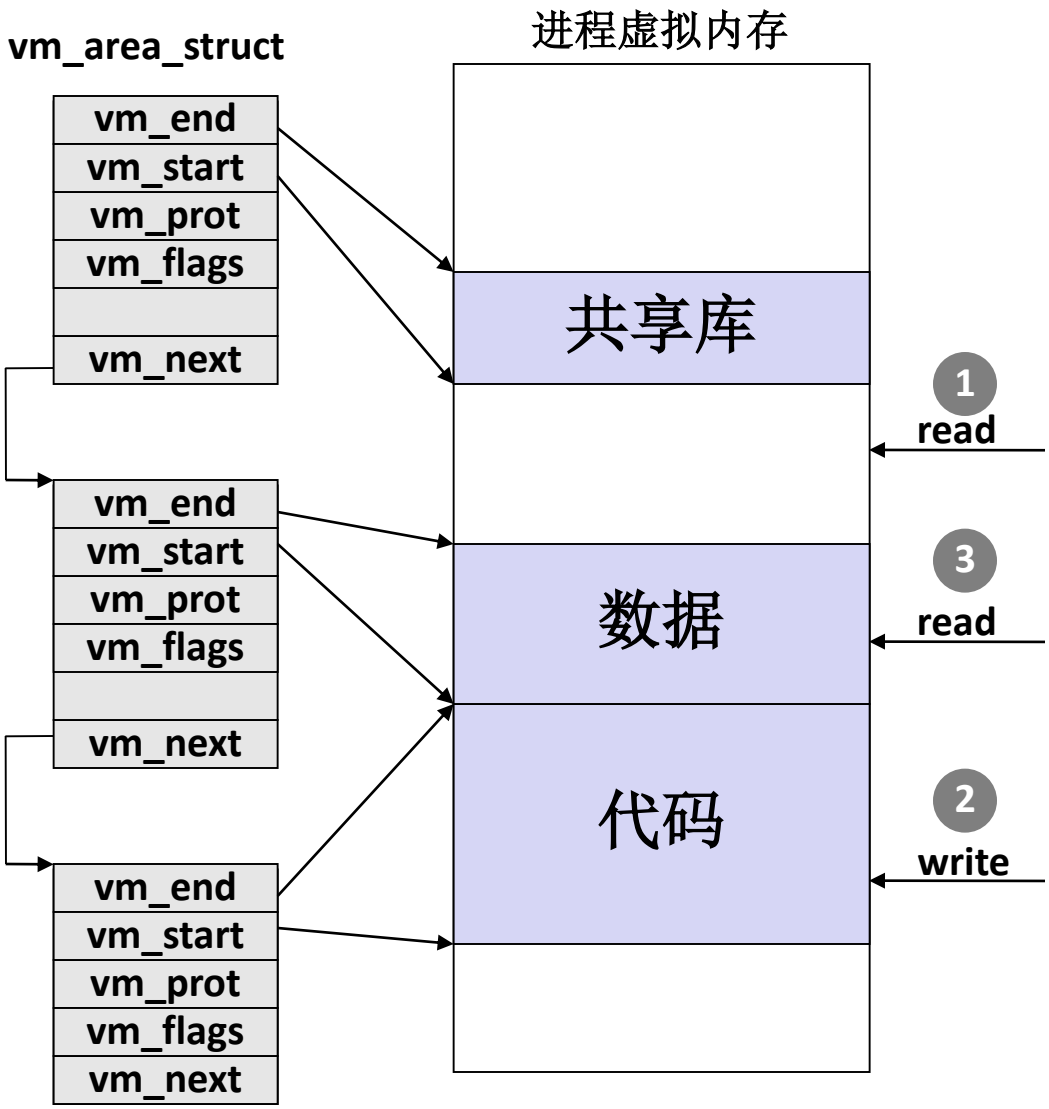


Linux将虚拟内存组织成一些区域的集合



- **pgd(page globale directory):**
  - 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



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)

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

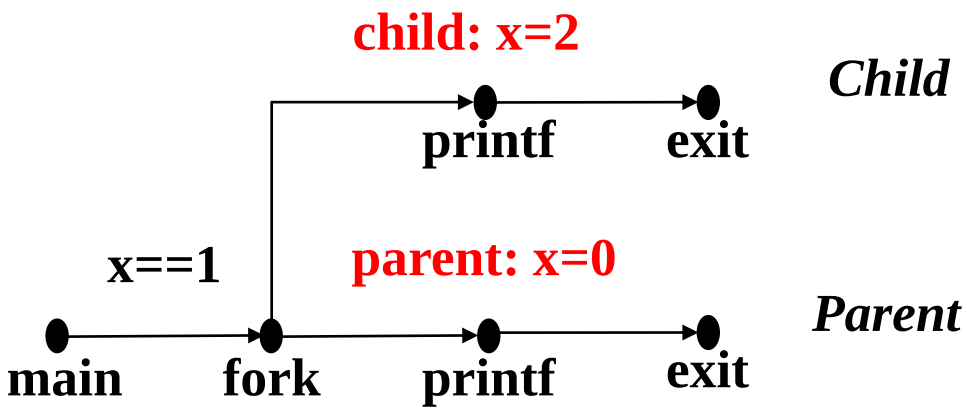
回顾进程图

fork.c

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
            exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```



提出问题：

进程fork：

父进程、子进程具有同样的存储区域，同时修改全局变量怎么办？

内存映射

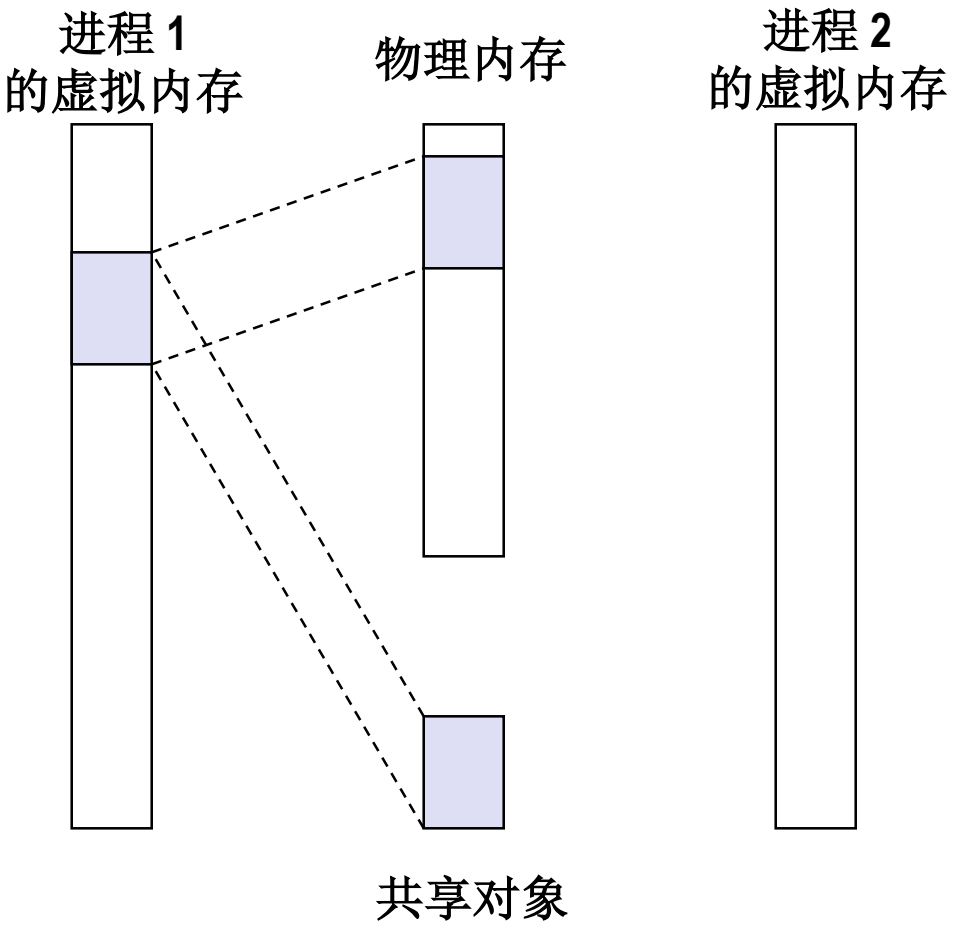
- Linux通过将虚拟内存区域与磁盘上的对象关联起来以初始化这个虚拟内存区域的内容.
  - 这个过程称为内存映射（*memory mapping*）.

- 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)
    - 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)
    - 第一次引用该区域内的虚拟页面时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

请查看https://blog.csdn.net/sinat_41619762/article/details/118972509

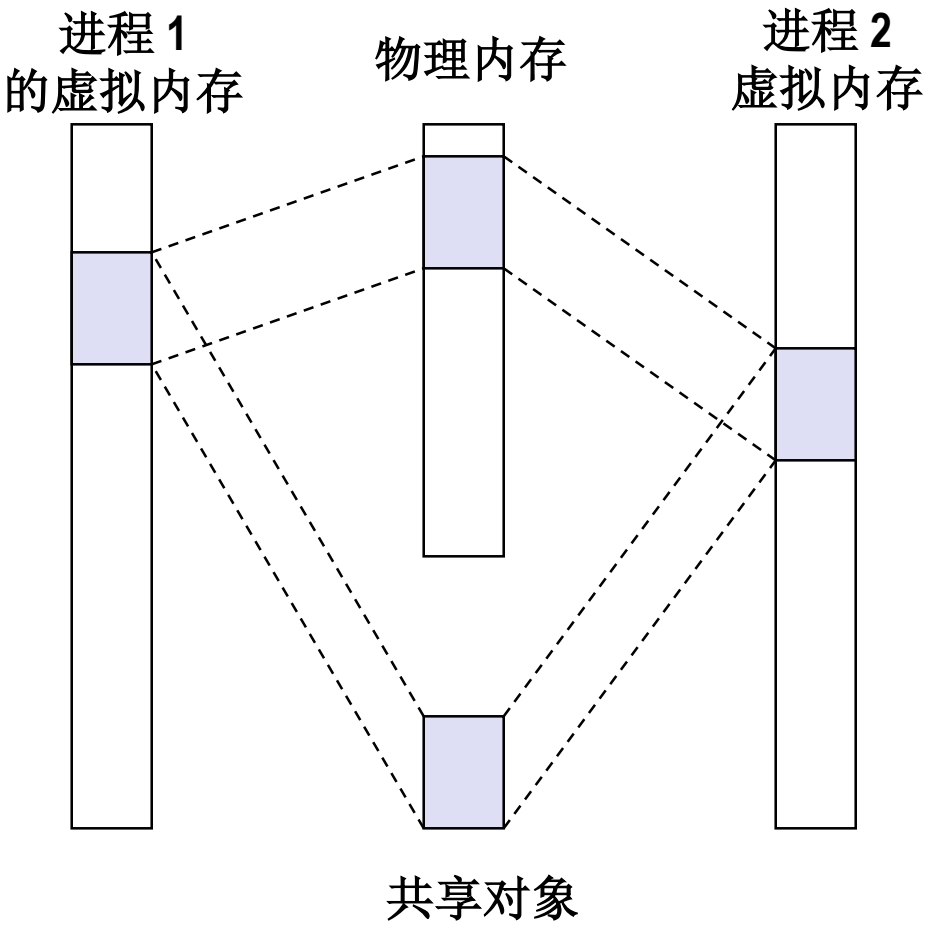
- 初始化后的页面在内存和交换文件（ *swap file*）之间换来换去

再看共享对象



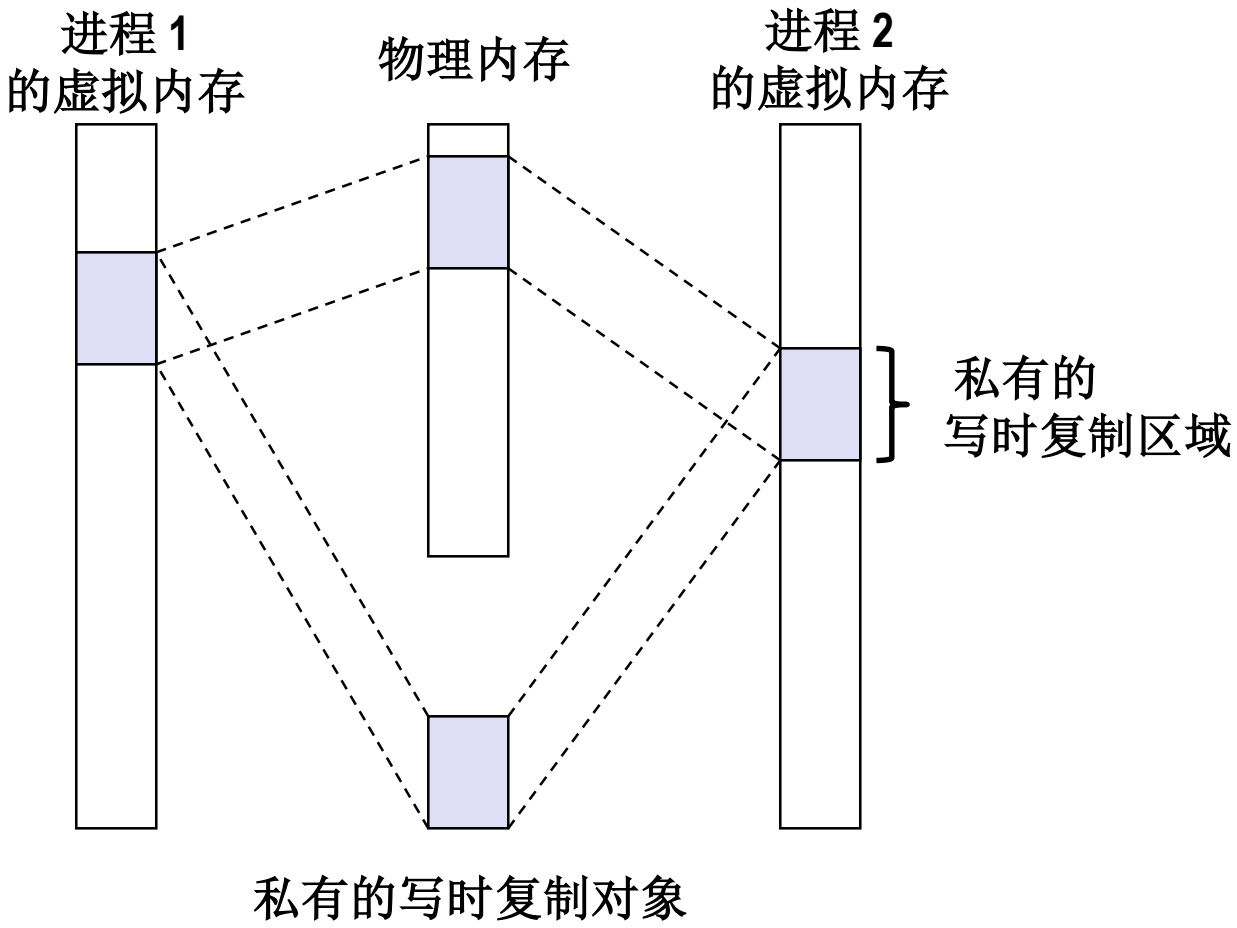
- 进程 1 映射了共享对象

再看共享对象



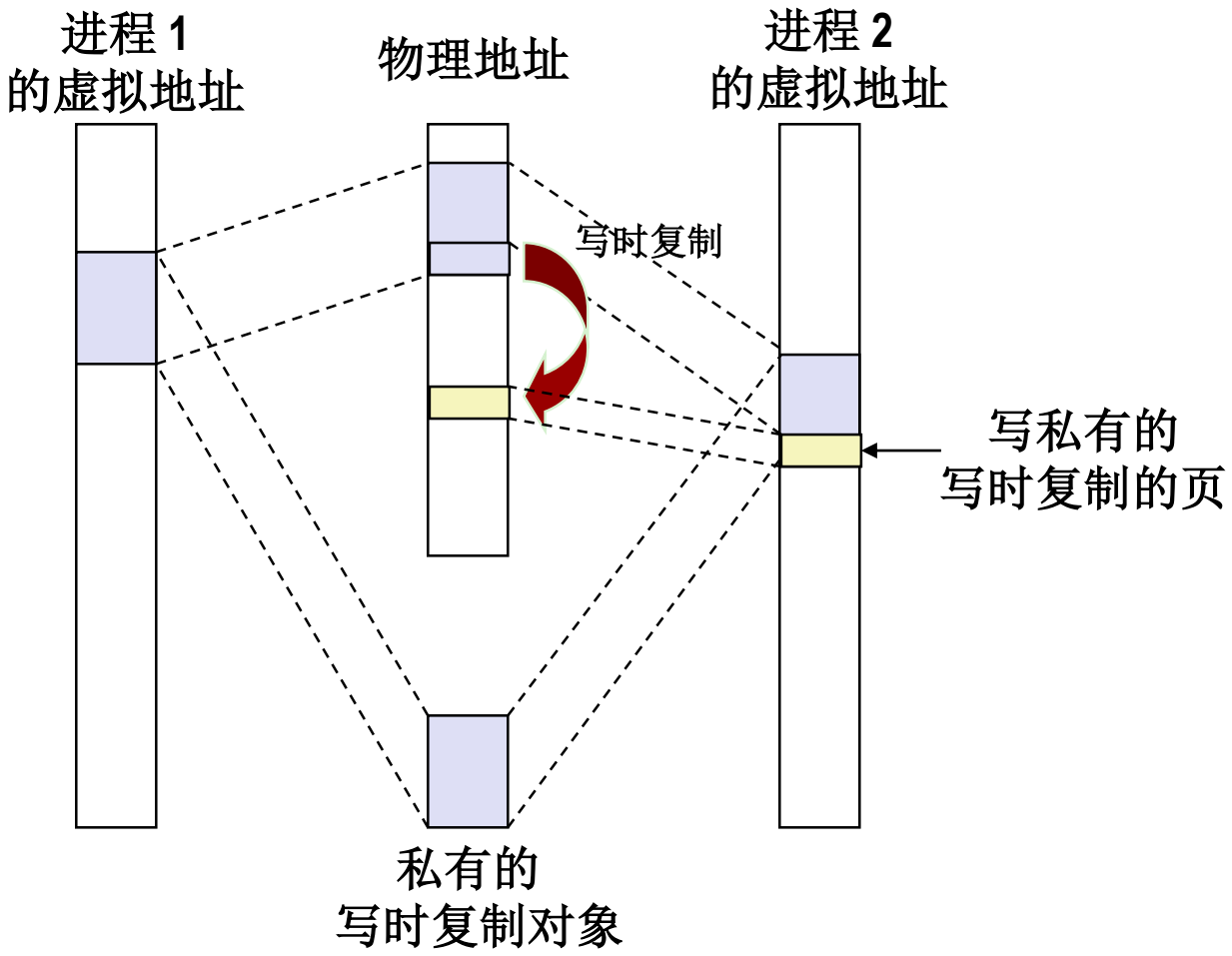
- 进程 2 映射了同一个共享对象.
- 两个进程的虚拟地址可以是不同的

共享对象： 堆栈、数据、堆区、寄存器
私有的写时复制（Copy-on-write）对象



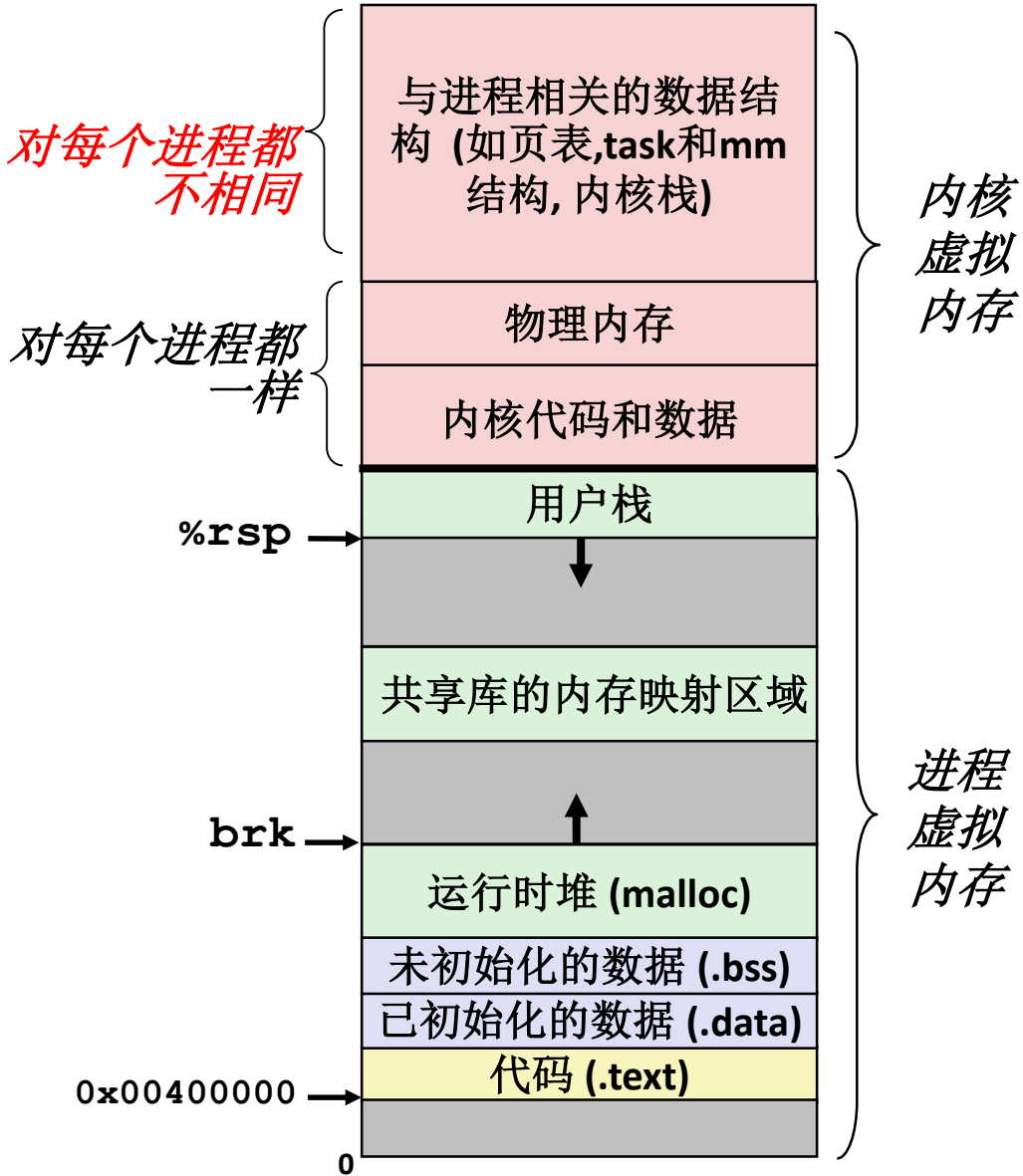
- 两个进程都映射了私有的写时复制对象
- 区域结构被标记为私有的写时复制
- 私有区域的页表条目都被标记为只读

共享对象:
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，更新PTE条目，且可写
- 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）充分利用物理内存

一个Linux 进程的虚拟地址空间（回顾）



再看 fork 函数

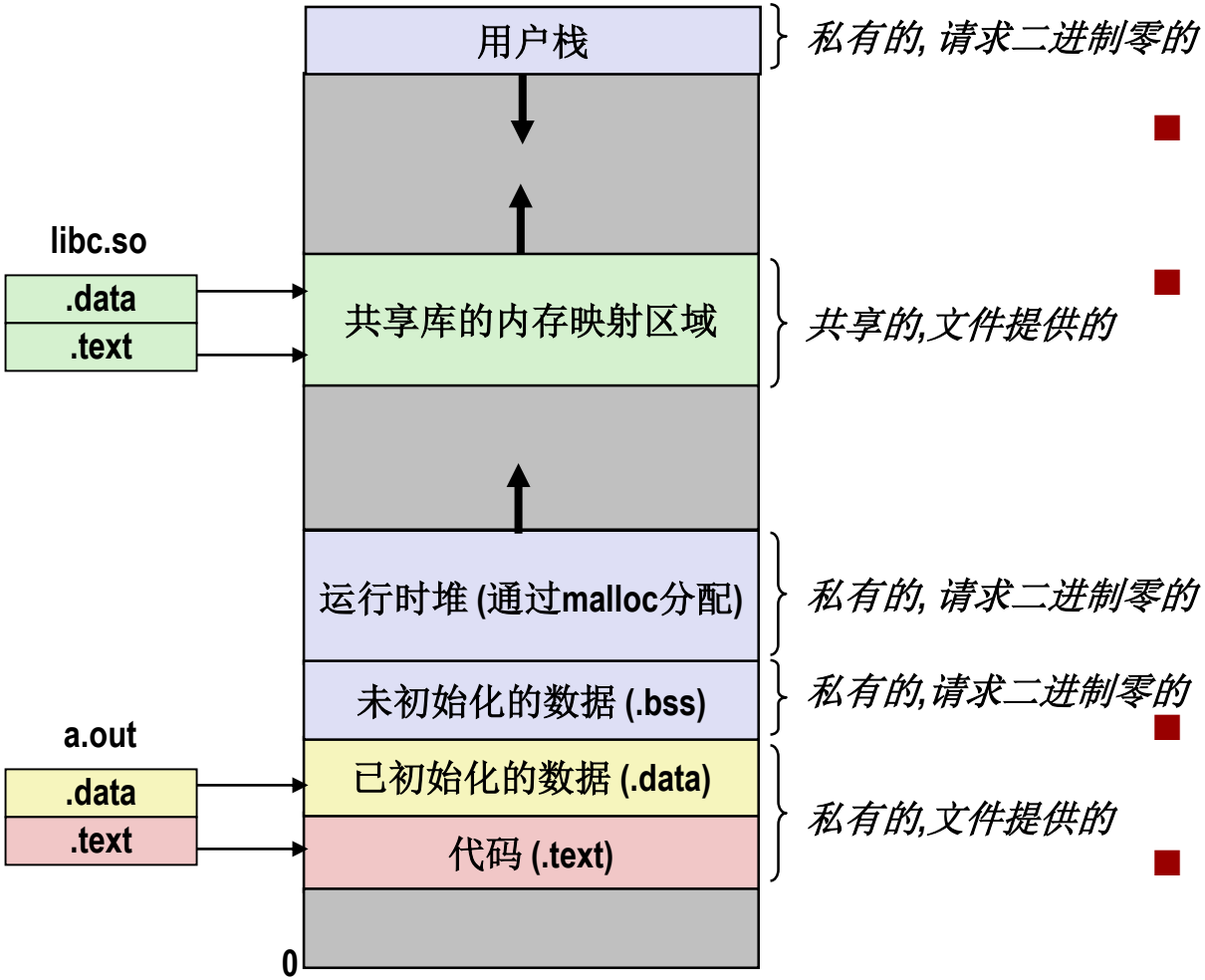
- 虚拟内存和内存映射解释了fork函数如何为每个新进程提供私有的虚拟地址空间.

- 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
  - 私有的、写时复制
  - 代码和初始化数据映射到.text和.data区（目标文件提供）
  - .bss和栈堆映射到匿名文件，栈堆的初始长度0
- 共享对象由动态链接映射到本进程共享区域
- 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

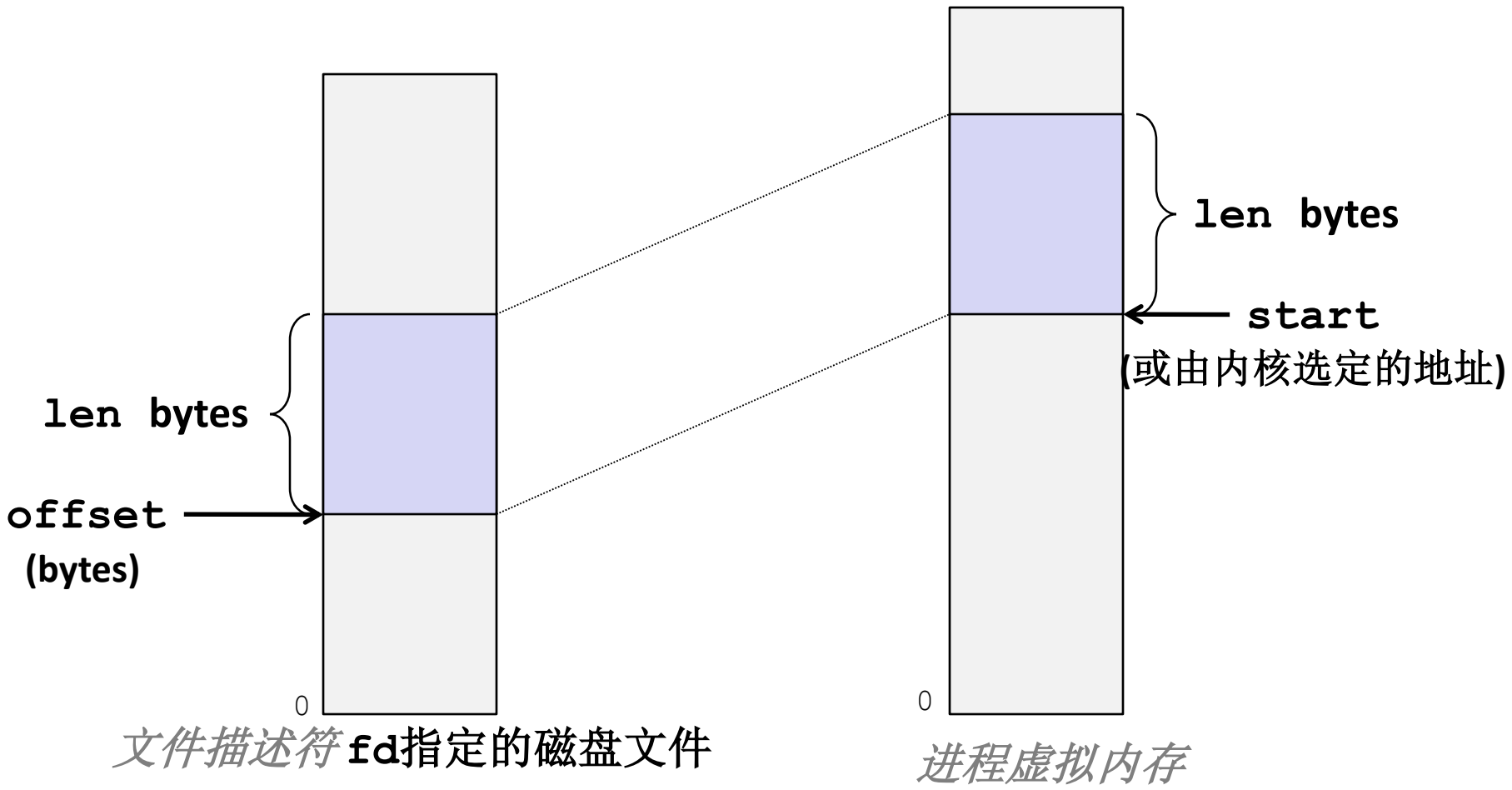
```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

创建新的虚拟内存区域，并将对象映射到这些区域

- 从**fd**指定的磁盘文件的**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, PROT_WRITE, ...
  - **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），...
- 返回一个指向映射区域开始处的指针

用户级内存映射

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



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）

```c
#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

```c
/* mmapcopy driver */
int main(int argc, char **argv)
{
   struct stat stat;
   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, &stat);
   mmapcopy(fd, stat.st_size);
   exit(0);
}
```

mmapcopy.c

这个程序的主要功能是将指定文件的内容复制到标准输出。它通过 mmap 系统调用来实现文件内容的内存映射，然后使用 write 函数将映射区域的内容写入标准输出。这种方法可以高效地处理大文件，因为 mmap 允许文件内容直接在内存中访问，而不需要频繁的读写操作。

需要注意的是，这个程序没有显式解除映射。在进程结束时，操作系统会自动解除映射并释放资源。此外，程序假设文件大小适合映射到进程的地址空间中。对于非常大的文件，可能需要分块处理以避免地址空间不足的问题。

Hope you enjoyed the CSAPP course!