第7章 虚拟存储器

虚拟存储器概述

页式虚拟存储器的实现

具有TLB和cache的存储系统

存储保护机制

实例： LoongArch架构虚拟存储系统

实例：Intel Core i7+Linux存储系统

Linux系统的存储器映射

虚拟存储器

- 主要教学目标
  - 了解虚拟存储管理的基本概念
  - 理解进程的虚拟地址空间的基本概念
  - 理解页式虚拟存储管理的实现原理
  - 理解访存操作完整过程以及所涉及到的部件之间的关联

    地址转换（查TLB、查页表）、访问Cache、访问主存、读写磁盘
  - 理解访存过程中硬件和操作系统之间的协调关系

    页表的构建和修改、访问违例、存储保护机制
  - 了解进程的虚拟存储器映射

虚拟存储器

- 分以下五个部分介绍
  - 第一讲：虚拟存储器概述
    - 进程的虚拟地址空间、虚拟存储器的基本类型
  - 第二讲：页式虚拟存储器的实现
    - 页表和页表项的结构、页式存储管理总体结构
    - 页式虚拟存储地址转换、快表
  - 第三讲：具有TLB和cache的存储系统
    - 层次化存储系统结构、CPU的访存过程
    - cache的4种查找方式
    - 存储保护机制
  - 第四讲：存储系统实例
    - LoongArch架构虚拟存储系统
    - Core i7+Linux存储系统
    - Linux系统的存储器映射

早期分页方式的概念

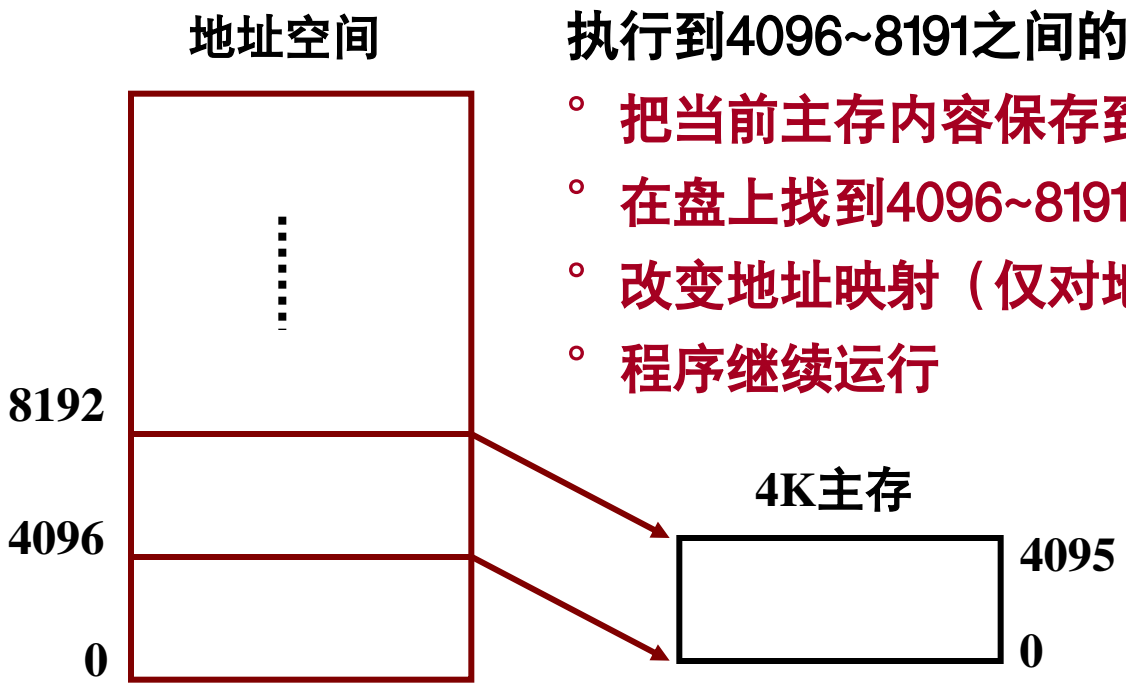
早期：程序员自己管理主存，通过分解程序并覆盖主存的方式执行程序

- 1961年，英国曼切斯特研究人员提出一种自动执行overlay的方式。
- 动机：把程序员从大量繁琐的存储管理工作中解放出来，使得程序员编程时不用管主存容量的大小。
- 基本思想：把地址空间和主存容量的概念区分开来。程序员在地址空间里编写程序，而程序则在真正的内存中运行。由一个专门的机制实现地址空间和实际主存之间的映射。
- 举例说明：

例如，当时的一种典型计算机，其指令中给出的主存地址为16位，而主存容量只有4K字，指令的可寻址范围是 …….？

地址空间为0、1、2…、65535组成的地址集合，即地址空间大小为2^{16}。程序员编写程序的空间（地址空间，可寻址空间）比执行程序的空间（主存容量）大得多，怎么自动执行程序呢?

早期分页方式的实现



执行到4096~8191之间的程序段时，自动做：

- 把当前主存内容保存到磁盘上；
- 在盘上找到4096~8191之间的程序段并读入主存
- 改变地址映射（仅对地址取模即可） 模为多少？
- 程序继续运行

后来把区间称为页(page)，主存中存放页的区域称为页框(page frame)。
早期主存只有一个页框！

- 将地址空间划分成4K大小的区间，装入内存的总是其中的一个区间
- 执行到某个区间时，把该区间的地址自动映射到0~4095之间，例如：
  - 4096→0，4097 →1，……，8191 →4095 如何映射？
- 程序员在0~65535范围内写程序，完全不用管在多大的主存空间上执行，所以，这种方式对程序员来说，是透明的！
- 可寻址的地址空间是一种虚拟内存！

分页（Paging）

- 基本思想：

  如：页大小=页框大小=4KB

  - 内存被分成固定长且比较小的存储块（页框、实页、物理页）
  - 每个进程也被划分成固定长的程序块（页、虚页、逻辑页）
  - 程序块可装入主存页框中
  - 无需用连续页框存放一个进程
  - 操作系统为每个进程生成一个页表
  - 通过页表(page table)实现逻辑地址向物理地址转换

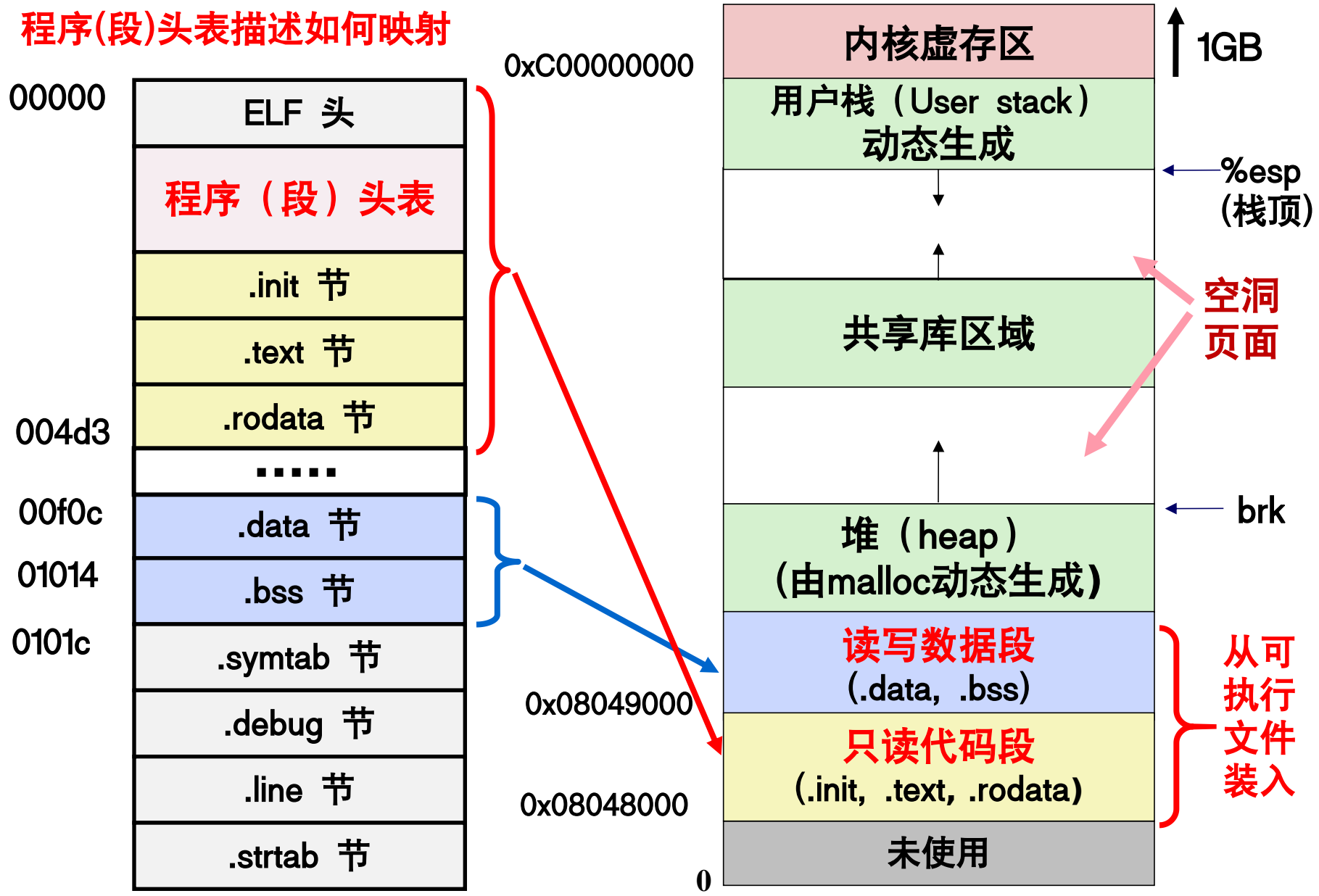
  程序块装入主存前存放在哪里？

  每页的起始地址是什么形式？

  磁盘和虚拟地址空间如何关联？

- 逻辑地址（Logical Address）：
  - 程序中指令所用地址(进程所在地址空间)，也称为虚拟地址（Virtual Address，简称VA）
- 物理地址（Physical Address，简称PA）：
  - 存放指令或数据的实际主存地址，也称为实地址、主存地址。

回顾： 可执行文件的存储器映像

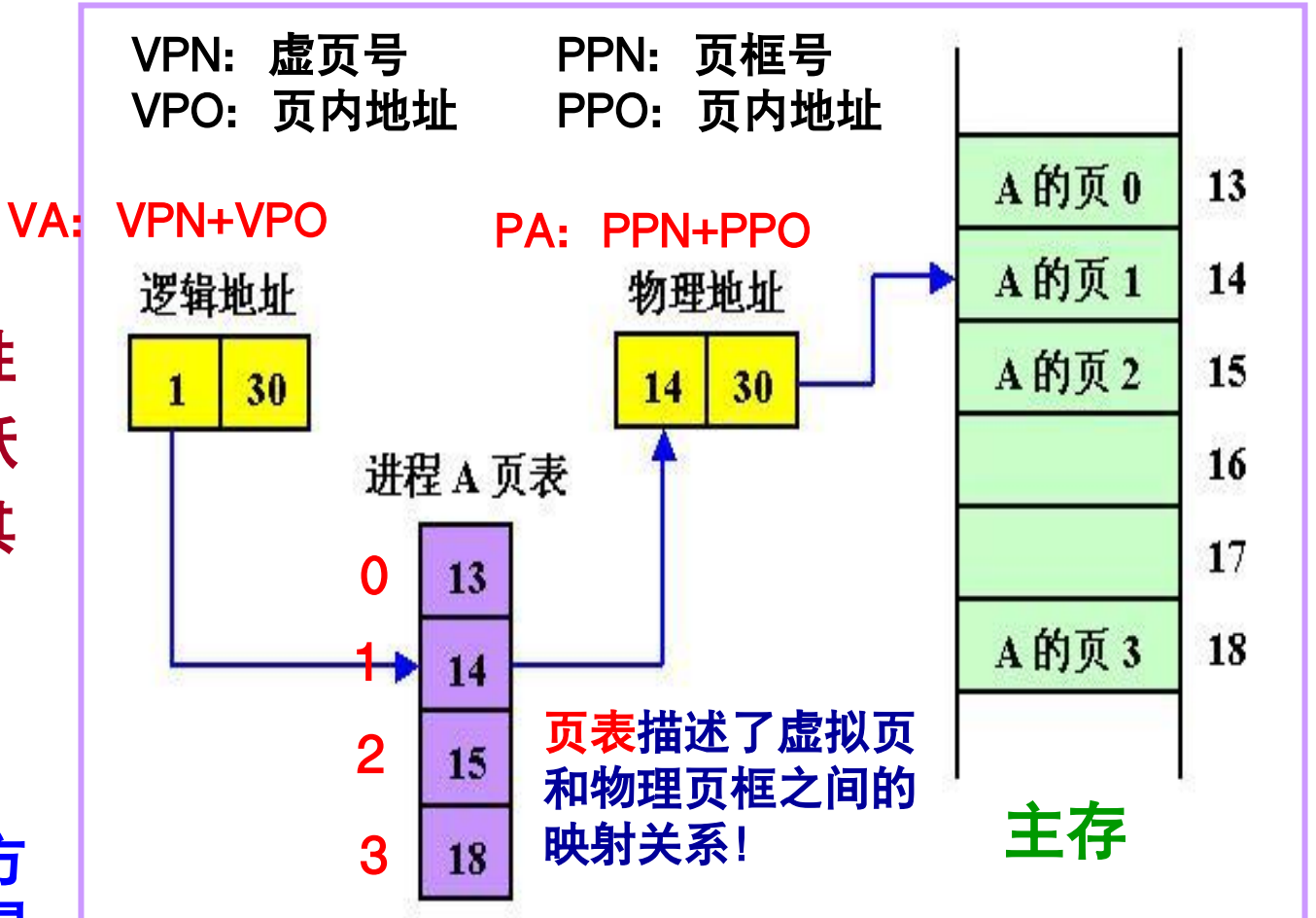


分页（Paging）

问题：是否需要将一个进程的全部都装入内存？为什么？

根据程序访问局部性可知：可把当前活跃的页面调入主存，其余留在磁盘上！

采用“按需调页 Demand Paging”方式分配主存！这就是虚拟存储管理概念



页大小有什么特点？和页内地址位数有什么关系？

每个页和每个页框的起始地址一定是固定的吗？

地址转换由硬件还是软件实现？为什么？

虚拟存储系统的基本概念

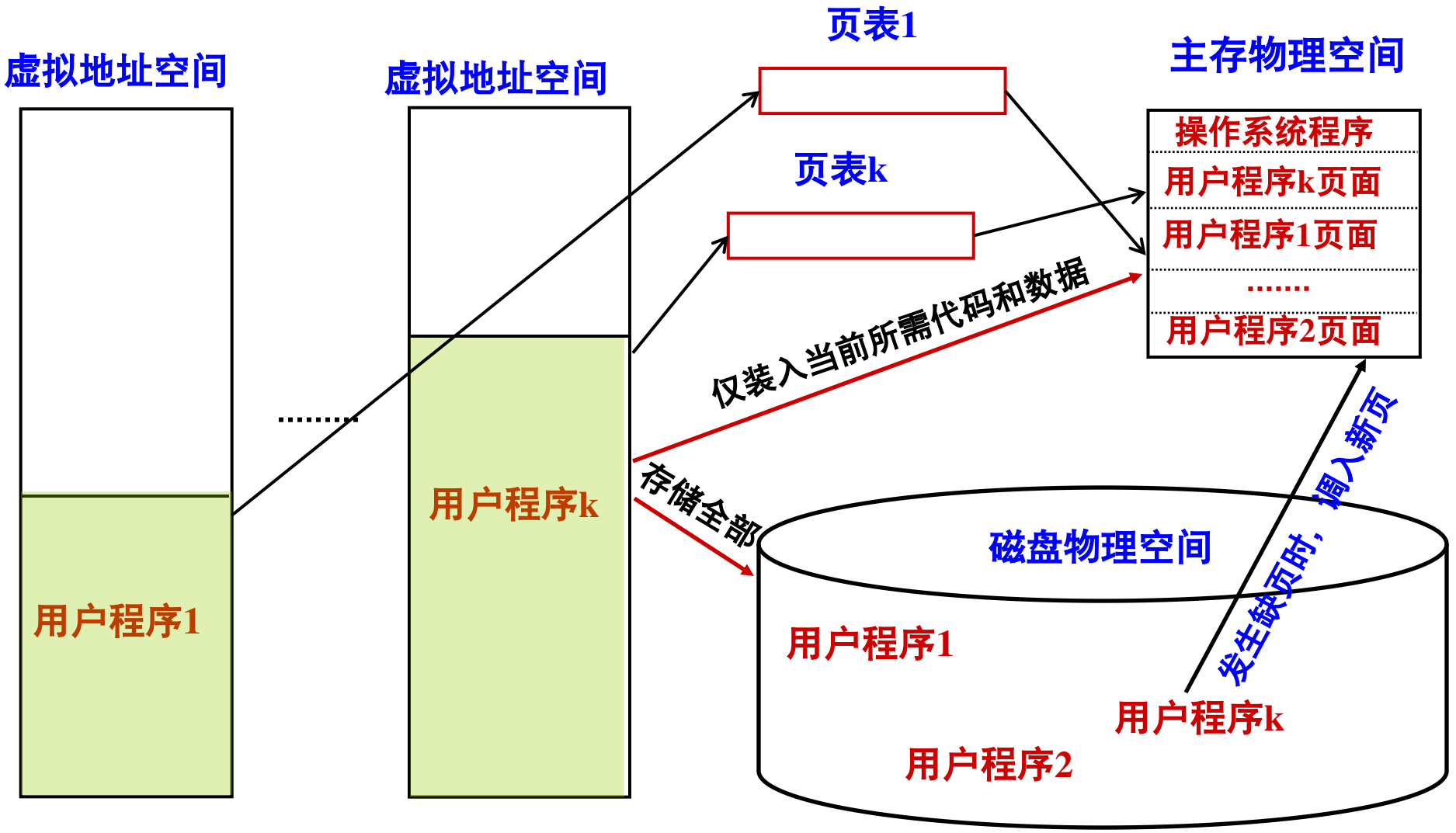
- 虚拟存储技术的最初引入用来解决一对矛盾
  - 一方面，由于技术和成本等原因，主存容量受到限制
  - 另一方面，系统程序和应用程序要求主存容量越来越大
- 虚拟存储技术的实质
  - 程序员在比实际主存空间大得多的逻辑地址空间中编写程序
  - 程序执行时，把当前需要的程序段和相应的数据块调入主存，其他暂不用的部分存放在磁盘上
  - 指令执行时，通过硬件将逻辑地址（也称虚拟地址或虚地址）转化为物理地址（也称主存地址或实地址）
  - 在发生程序或数据访问失效(缺页)时，由操作系统进行主存和磁盘之间的信息交换
- 虚拟存储器机制由硬件与操作系统共同协作实现，涉及到操作系统中的许多概念，如进程、进程的上下文切换、存储器分配、虚拟地址空间、缺页处理等。

SKIP

虚拟存储技术的实质

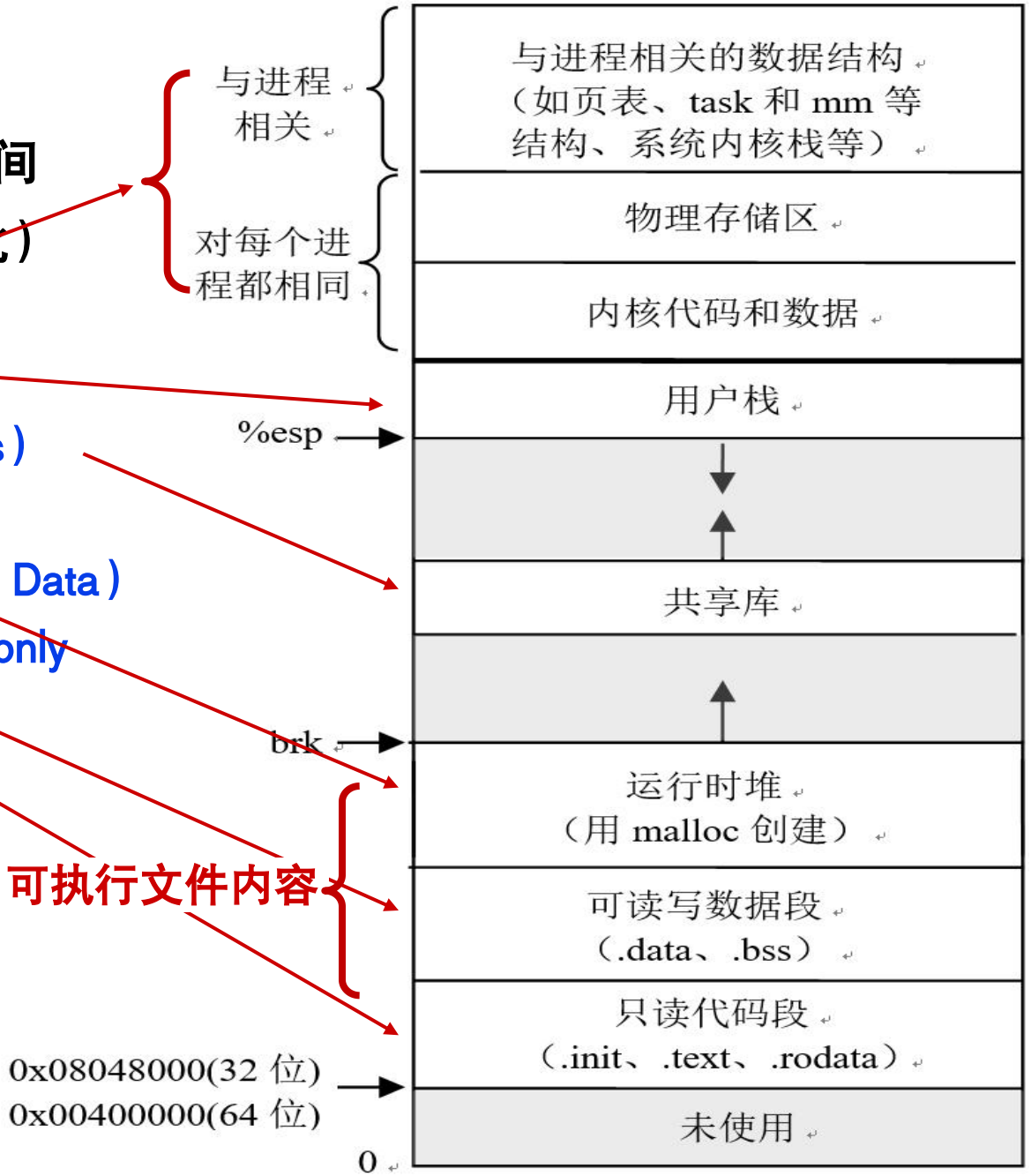
通过页表建立虚拟空间和物理空间之间的映射!

BACK



虚拟地址空间

- Linux在X86上的虚拟地址空间
  （其他Unix系统的设计类此）
  - 内核空间（Kernel）
  - 用户栈（User Stack）
  - 共享库（Shared Libraries）
  - 堆（heap）
  - 可读写数据（Read/Write Data）
  - 只读数据和代码（Read-only Data/Code）



问题：加载时是否真正从磁盘调入信息到主存?

不会，只是将虚拟页和磁盘上的数据/代码建立对应关系，称为“存储器映射”，mmap系统调用！

回顾：可执行文件中的程序头表

用“readelf -l test”命令显示LA64中可执行文件test的程序头表部分信息

Elf file type is EXEC (Executable file)

Entry point 0x1200004e0

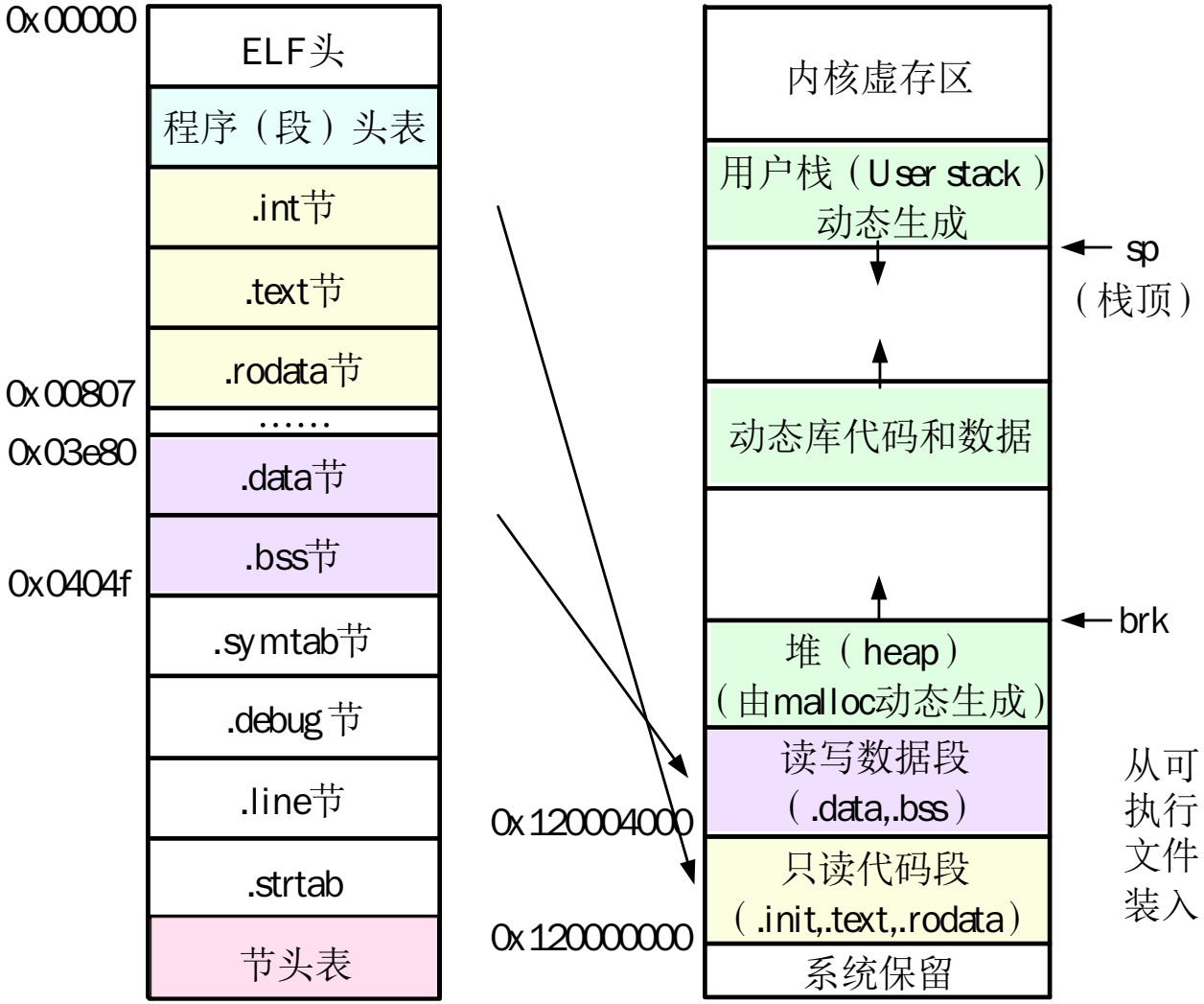
There are 9 program headers, starting at offset 64

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flags	Align
PHDR	0x00040	0x0120000040	0x0120000040	0x001f8	0x01f8	R	0x8
INTERP	0x00238	0x0120000238	0x0120000238	0x0000f	0x0000f	R	0x1
[Requesting program interpreter: /lib64/ld.so.1]							
LOAD	0x00000	0x120000000	0x120000000	0x0808	0x0808	R E	0x4000
LOAD	0x03e80	0x120007e80	0x120007e80	0x01d0	0x01e0	RW	0x4000
DYNAMIC	0x03e90	0x120007e90	0x120007e90	0x0170	0x0170	RW	0x8
NOTE	0x00248	0x120000248	0x120000248	0x0044	0x0044	R	0x4
GNU_EH_FRAME	0x00780	0x120000780	0x120000780	0x001c	0x001c	R	0x4
GNU_STACK	0x00000	0x000000000	0x000000000	0x0000	0x0000	RW	0x10
GNU_RELRO	0x03e80	0x120007e80	0x120007e80	0x0180	0x0180	R	0x1

回顾：可执行文件的存储器映像

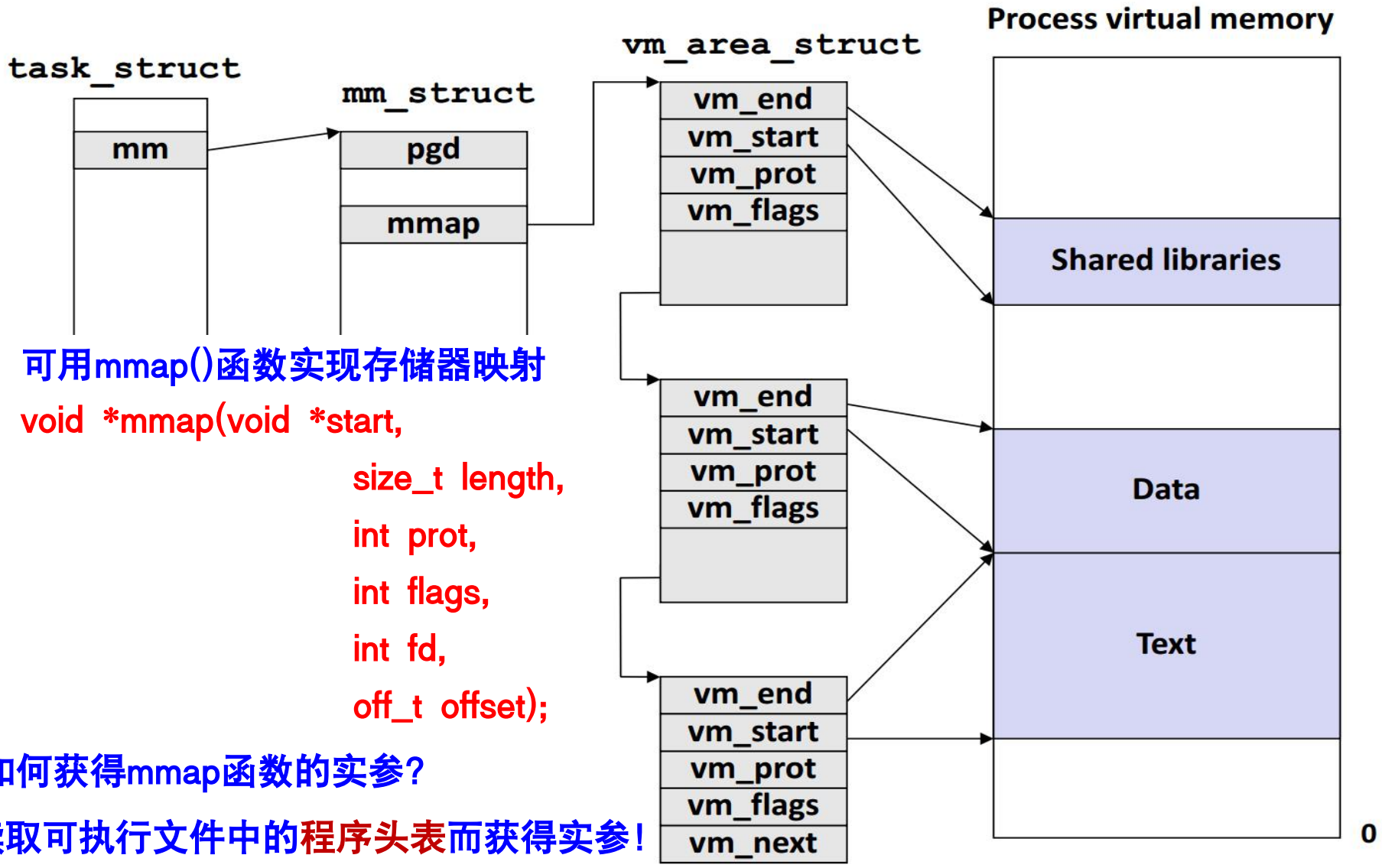
根据上页可执行文件test的程序头表信息画出其存储器映像如下



test中最开始的0x00808字节映射到虚拟地址0x1 2000 0000开始的**只读代码段**，按0x4000=16KB对齐，只读代码段只有0x808B，因此，**读写数据段**从0x1 2000 4000开始。为提升存储访问性能，test中0x03e80到0x0404f之间的可装入段映射到虚拟地址0x1 2000 4000+0x03e80=0x12000 7e80开始的位置，该装入段包含.data节和.bss节，在虚拟地址空间中需给.bss节中定义的变量分配空间，且初始值为0。.bss节的起始虚拟地址为0x12000 4000 + 0x03e80+0x1d0 = 0x1 2000 8050。

进程的存储器映射

存储器映射（memory mapping）是指将进程虚拟地址空间中的一个区域与硬盘上的一个对象建立关联（生成页表项），并初始化一个vm_area_struct结构信息



可用mmap()函数实现存储器映射

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

如何获得mmap函数的实参？

读取可执行文件中的程序头表而获得实参！

回顾：可执行文件中的程序头表

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004d4 0x004d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

第一可装入段：第0x00000~0x004d3字节（包括ELF头、程序头表、.init、.text和.rodata节），映射到虚拟地址0x8048000开始长度为0x4d4字节的区域，按0x1000=4KB对齐，具有只读/执行权限（Flg=RE），是只读代码段。

第二可装入段：第0x000f0c开始长度为0x108字节的.data节，映射到虚拟地址0x8049f0c开始长度为0x110字节的存储区域，在0x110=272B存储区中，前0x108=264B用.data节内容初始化，后面272-264=8B对应.bss节，初始化为0，按0x1000=4KB对齐，具有可读可写权限（Flg=RW），是可读写数据段。

虚拟存储器管理

实现虚拟存储器管理，需考虑:

块大小（在虚拟存储器中“块”被称为“页 / Page”）应多大?

主存与虚存的空间如何分区管理?

程序块（页） / 存储块（页框）之间如何映射?

逻辑地址和物理地址如何转换，转换速度如何提高?

主存与辅存之间如何进行替换（与Cache所用策略相似）?

页表如何实现，页表项中要记录哪些信息?

如何加快访问页表的速度?

如果要找的内容不在主存，怎么办?

如何保护进程各自的存储区不被其他进程访问?

有三种虚拟存储器实现方式:

分页式、分段式、段页式

这些问题是由硬件和OS共同协调解决的!

“主存--磁盘”层次

与“Cache--主存”层次相比:

页大小（2KB~64KB）比Cache中的Block大得多！ Why?

采用全相联映射！Why?

缺页的开销比Cache缺失开销大的多！缺页时需要访问磁盘（约几百万个时钟周期），而cache缺失时，访问主存仅需几十到几百个时钟周期！因此，页命中率比cache命中率更重要！“大页面”和“全相联”可提高页命中率。

通过软件来处理“缺页”！Why?

缺页时需要访问磁盘（约几百万个时钟周期），慢！不能用硬件实现。

采用Write Back写策略！ Why?

避免频繁的慢速磁盘访问操作。

地址转换用硬件实现！Why?

加快指令执行

层次结构存储系统

- 分以下五个部分介绍
  - 第一讲：虚拟存储器概述
    - 进程的虚拟地址空间、虚拟存储器的基本类型
  - 第二讲：页式虚拟存储器的实现
    - 页表和页表项的结构、页式存储管理总体结构
    - 页式虚拟存储地址转换、快表
  - 第三讲：具有TLB和cache的存储系统
    - 层次化存储系统结构、CPU的访存过程
    - cache的4种查找方式
    - 存储保护机制
  - 第四讲：存储系统实例
    - LoongArch架构虚拟存储系统
    - Core i7+Linux存储系统
    - Linux系统的存储器映射

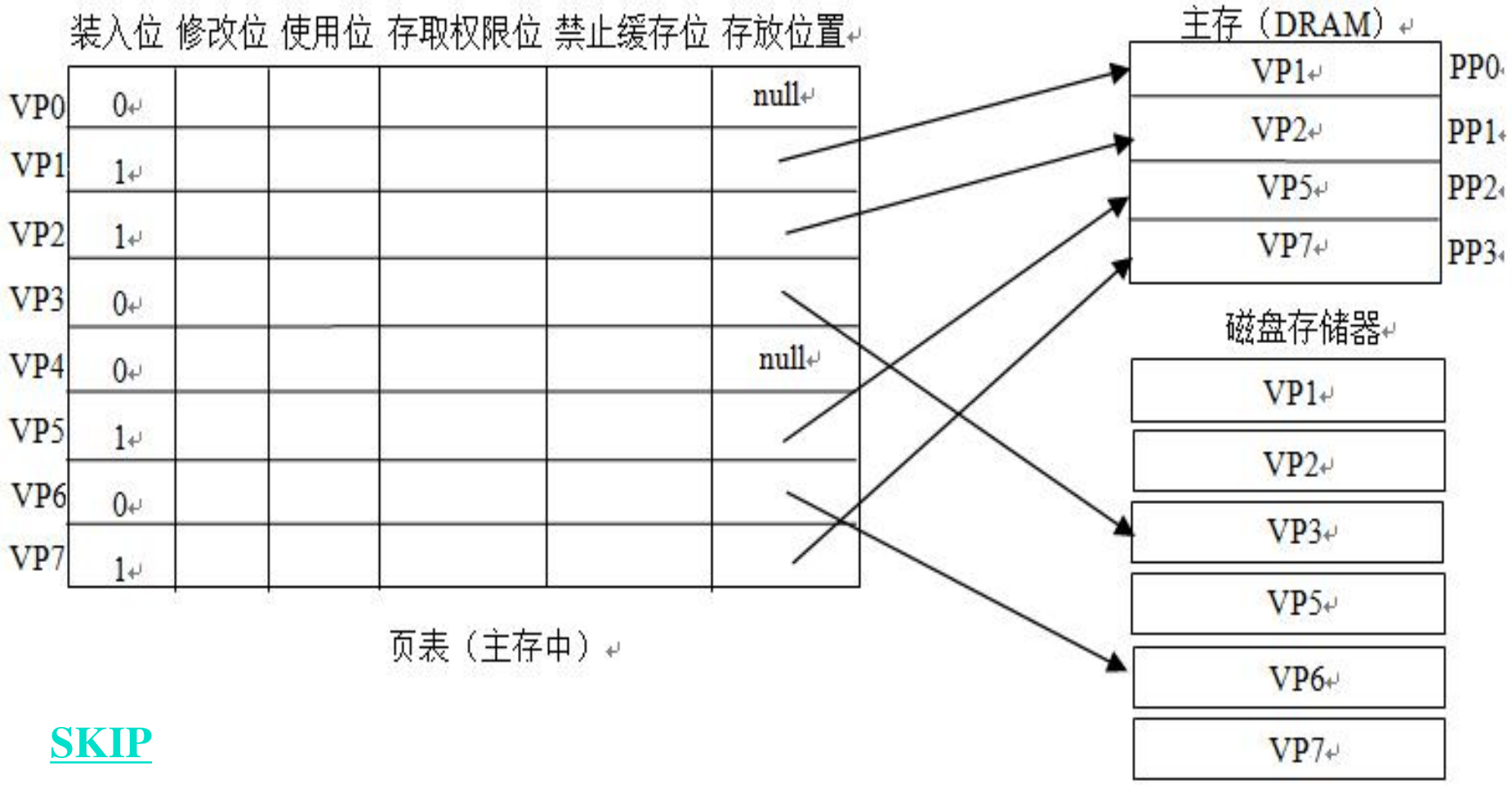
页表结构

u 页表首址记录在页表基址寄存器中

	装入位	修改位	替换控制位	其他	实页号（8进制）
0 虚页	1				11
1 虚页	1				13
2 虚页	1				16
3 虚页	1				10
4 虚页	1				14

- 每个进程有一个页表，其中有装入位、修改（dirty）位、替换控制位、访问权限位、禁止缓存位、实页号。
- 一个页表的项数由什么决定？ 理论上由虚拟地址空间大小决定。
- 每个进程的页表大小一样吗？ 各进程有相同虚拟空间，故理论上一样。实际大小看具体实现方式，如“空洞”页面如何处理等

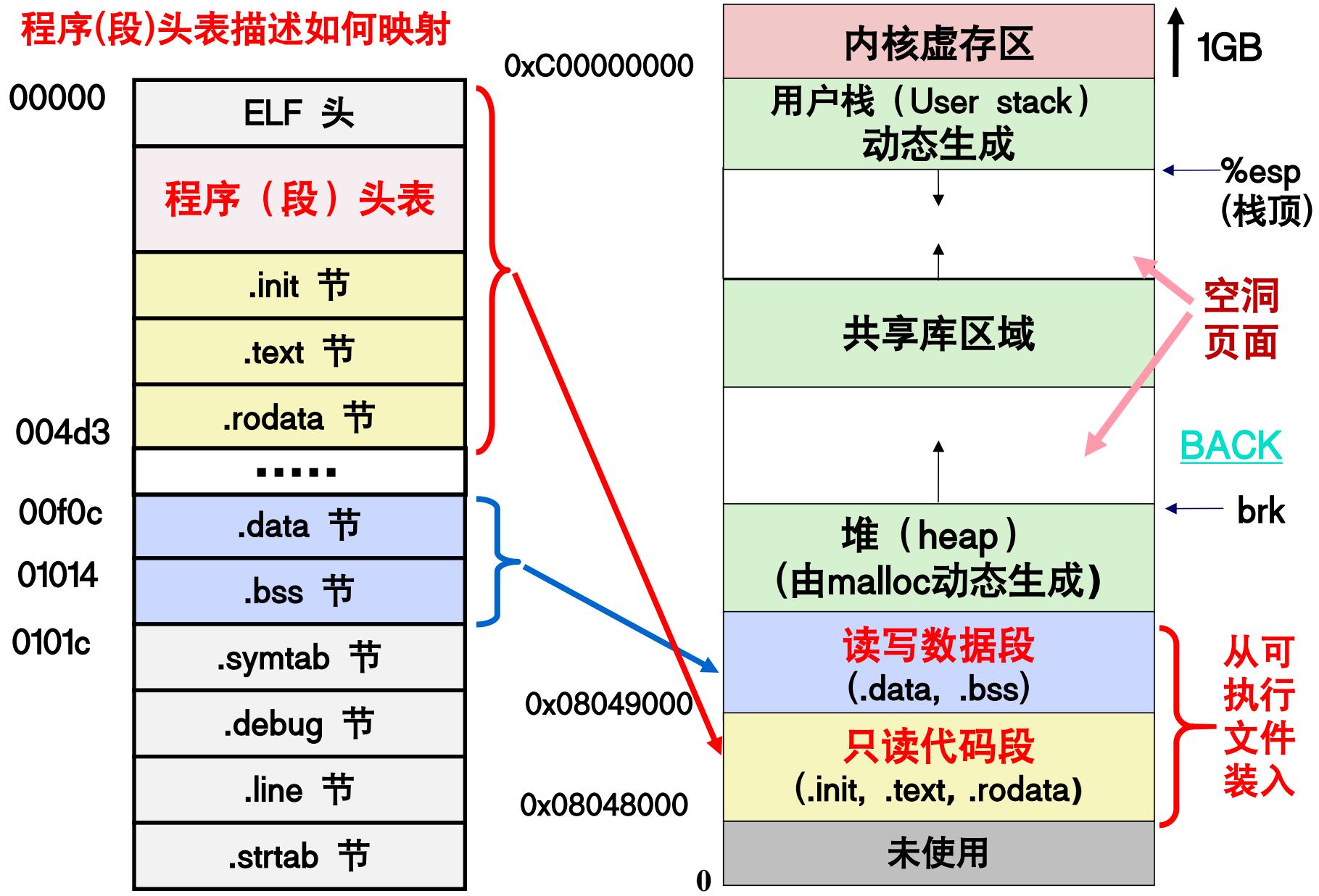
主存中的页表示例



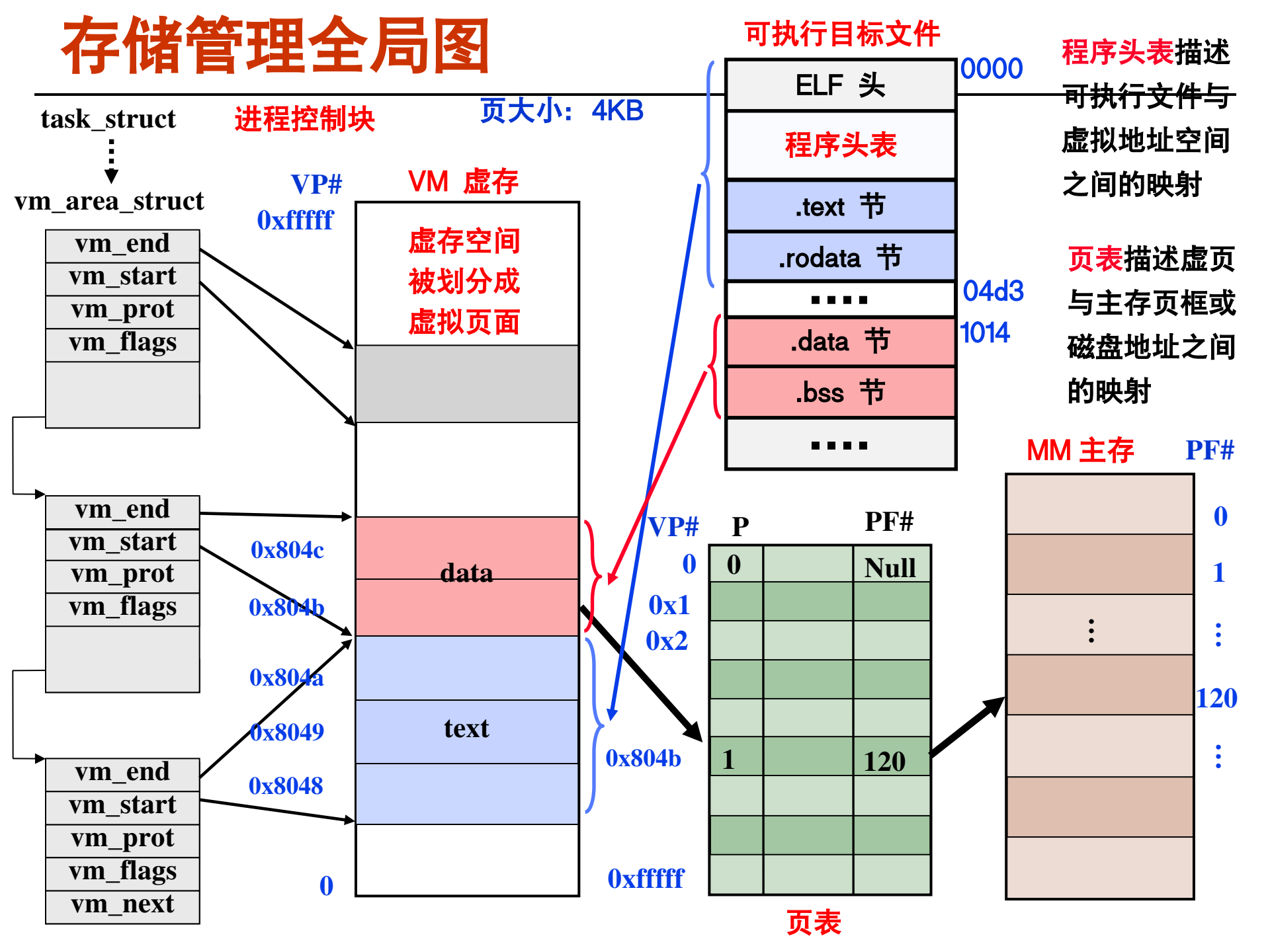
SKIP

- 未分配页：进程的虚拟地址空间中“空洞”对应的页（如VP0、VP4）
- 已分配的缓存页：有内容对应的已装入主存的页（如VP1、VP2、VP5等）
- 已分配的未缓存页：有内容对应但未装入主存的页（如VP3、VP6）

回顾： 可执行文件的存储器映像



存储管理全局图

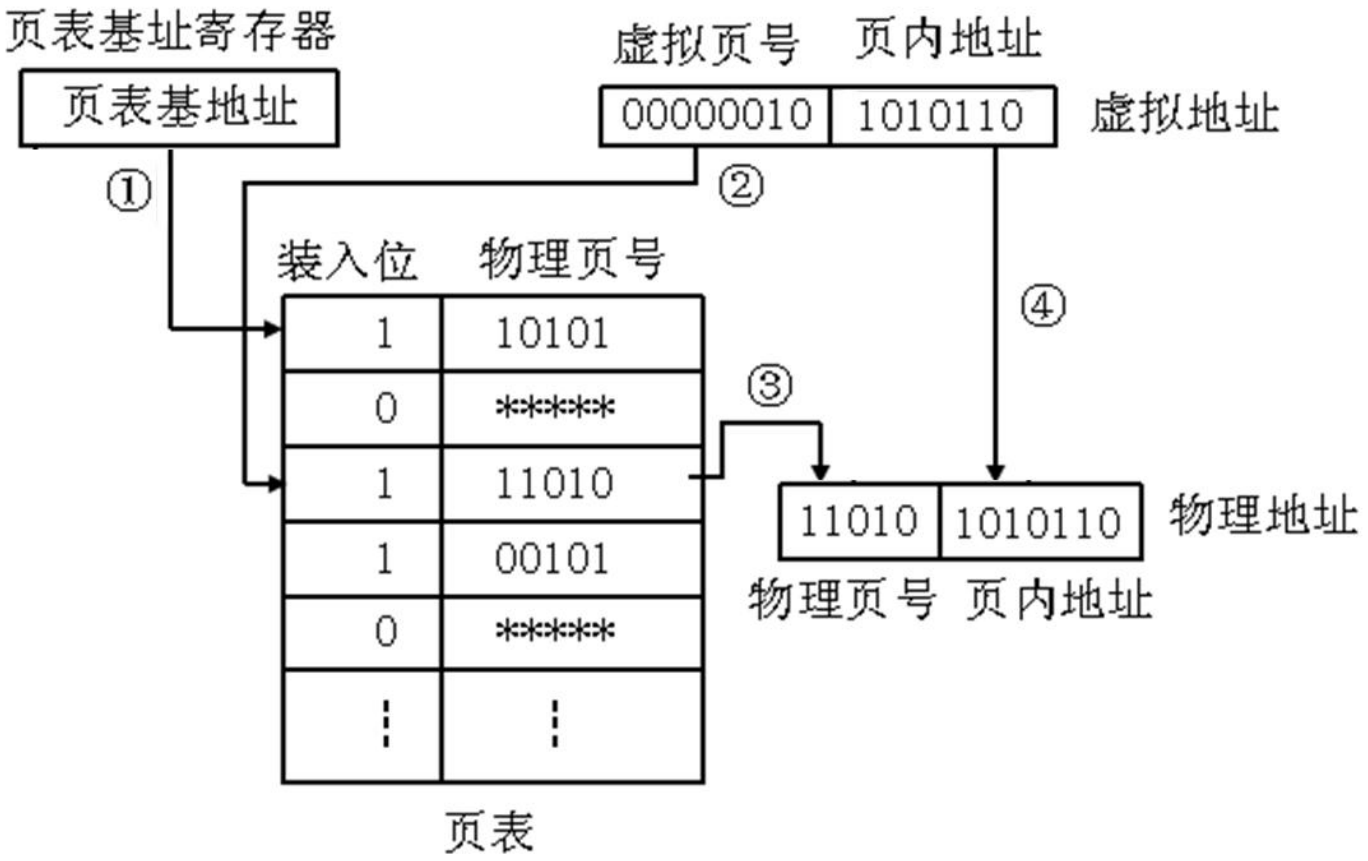

可执行目标文件
程序头表描述可执行文件与虚拟地址空间之间的映射
页表描述虚页与主存页框或磁盘地址之间的映射
task_struct
进程控制块
页大小：4KB
vm_area_struct
VP#
VM 虚存
0xfffff
虚存空间被划分成虚拟页面
vm_end
vm_start
vm_prot
vm_flags
vm_next
data
text
0x804c
0x804b
0x804a
0x8049
0x8048
0
ELF 头
程序头表
.text 节
.rodata 节
.data 节
.bss 节
0000
04d3
1014
VP#
P
PF#
Null
120
0x1
0x2
0x804b
0xfffff
MM 主存
PF#
页表

逻辑地址转换为物理地址的过程

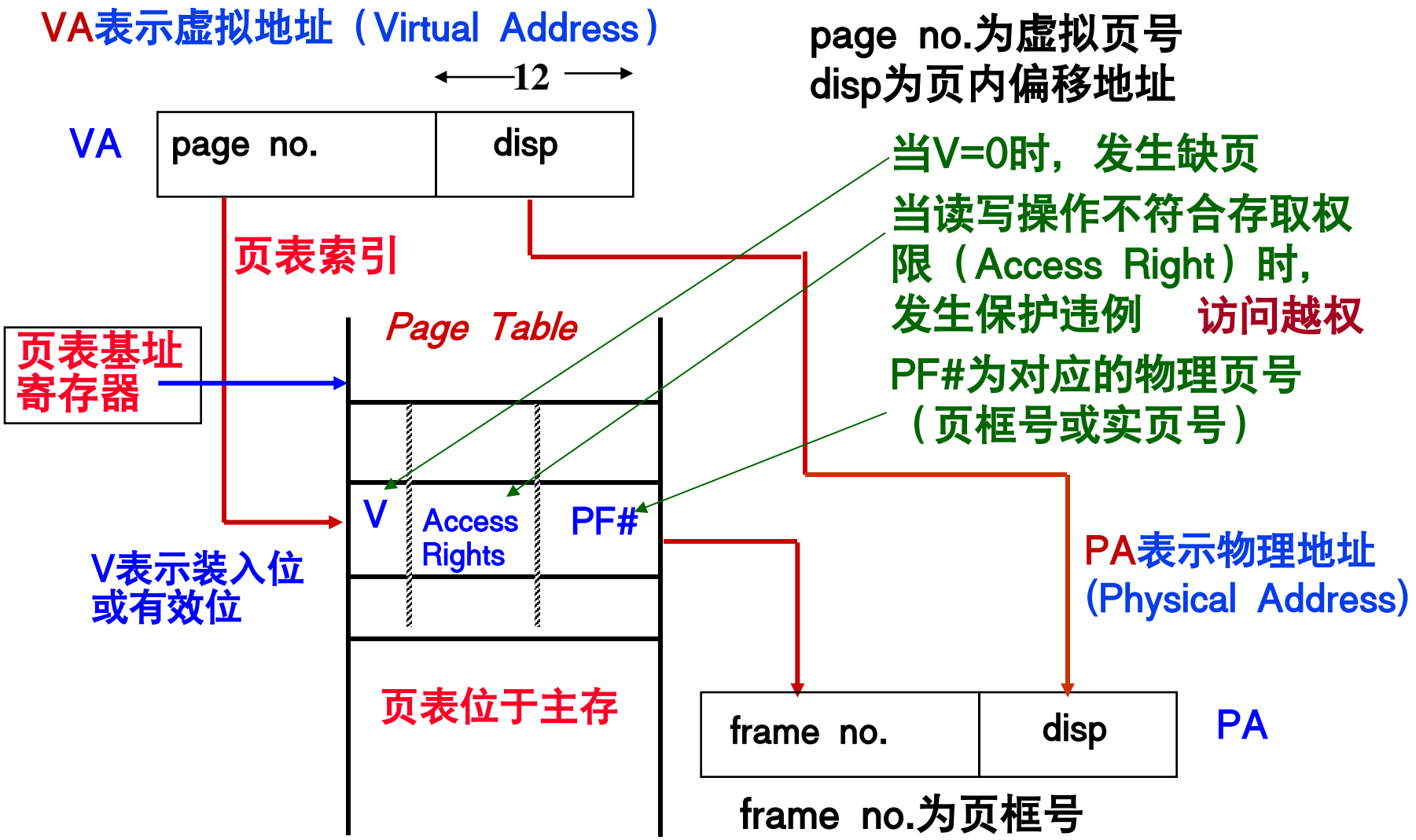
虚拟地址分两个字段：高位字段为虚拟页号，低位字段为页内偏移地址，简称页内地址

主存物理地址分两个字段：高位字段为物理页号，低位字段为页内偏移地址

由于虚拟页和物理页的大小一样，所以两者的页内偏移地址相等



逻辑地址转换为物理地址的过程



问题1：什么情况下，不能成功进行地址转换呢?

问题2：如何知道页表中的存取权限应该设置成什么（R/W、R、X）？

回顾：可执行文件中的程序头表

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00100	0x00100	R E	0x4
INTERP	0x000134	0x08048134	0x08048134	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x004d4	0x004d4	R E	0x1000
LOAD	0x000f0c	0x08049f0c	0x08049f0c	0x00108	0x00110	RW	0x1000
DYNAMIC	0x000f20	0x08049f20	0x08049f20	0x000d0	0x000d0	RW	0x4
NOTE	0x000148	0x08048148	0x08048148	0x00044	0x00044	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4
GNU_RELRO	0x000f0c	0x08049f0c	0x08049f0c	0x000f4	0x000f4	R	0x1

第一可装入段：第0x00000~0x004d3字节（包括ELF头、程序头表、.init、.text和.rodata节），映射到虚拟地址0x8048000开始长度为0x4d4字节的区域，按0x1000=4KB对齐，**具有只读/执行权限（Flg=RE）**，是只读代码段。

第二可装入段：第0x000f0c开始长度为0x108字节的.data节，映射到虚拟地址0x8049f0c开始长度为0x110字节的存储区域，在0x110=272B存储区中，前0x108=264B用.data节内容初始化，后面272-264=8B对应.bss节，初始化为0，按0x1000=4KB对齐，**具有可读可写权限（Flg=RW）**，是可读写数据段。

信息访问中可能出现的异常情况

可能有两种异常（exception）情况:

1）缺页（ page fault）

产生条件: 当Valid（有效位 / 装入位）为 0 时

相应处理: 从磁盘读页面到主存，若主存没有空间，则从主存选择一页替

换到磁盘上，替换算法类似于Cache，采用回写法，淘汰时，根据“dirty”

位确定是否要写磁盘

当前指令执行被阻塞，当前进程被挂起，处理结束回到原指令继续执行

2）保护违例（ protection_violation_fault ）或访问违例

产生条件: 当Access Rights (存取权限)与所指定的具体操作不相符时

相应处理: 在屏幕上显示“内存保护错”或“访问违例”信息

当前指令的执行被阻塞，当前进程被终止

Access Rights (存取权限)可能的取值有哪些?

R = Read-only, R/W = read/write, X = execute only

回顾：用“系统思维”分析问题

```
int sum(int a[ ], unsigned len)
{
  int    i, sum = 0;
  for    (i = 0; i <= len - 1; i++)
         sum += a[i];
  return sum;
}
```

当参数len为0时，返回值应该是0，但是在机器上执行时，却发生访存异常。但当len为int型时则正常Why?

TLBs --- Making Address Translation Fast

问题：一次存储器引用要访问几次主存？

页表在主存，
至少 1 次？

把经常要查的页表项放到Cache中，这种在Cache中的页表项组成的页表称为Translation Lookaside Buffer or TLB（快表）

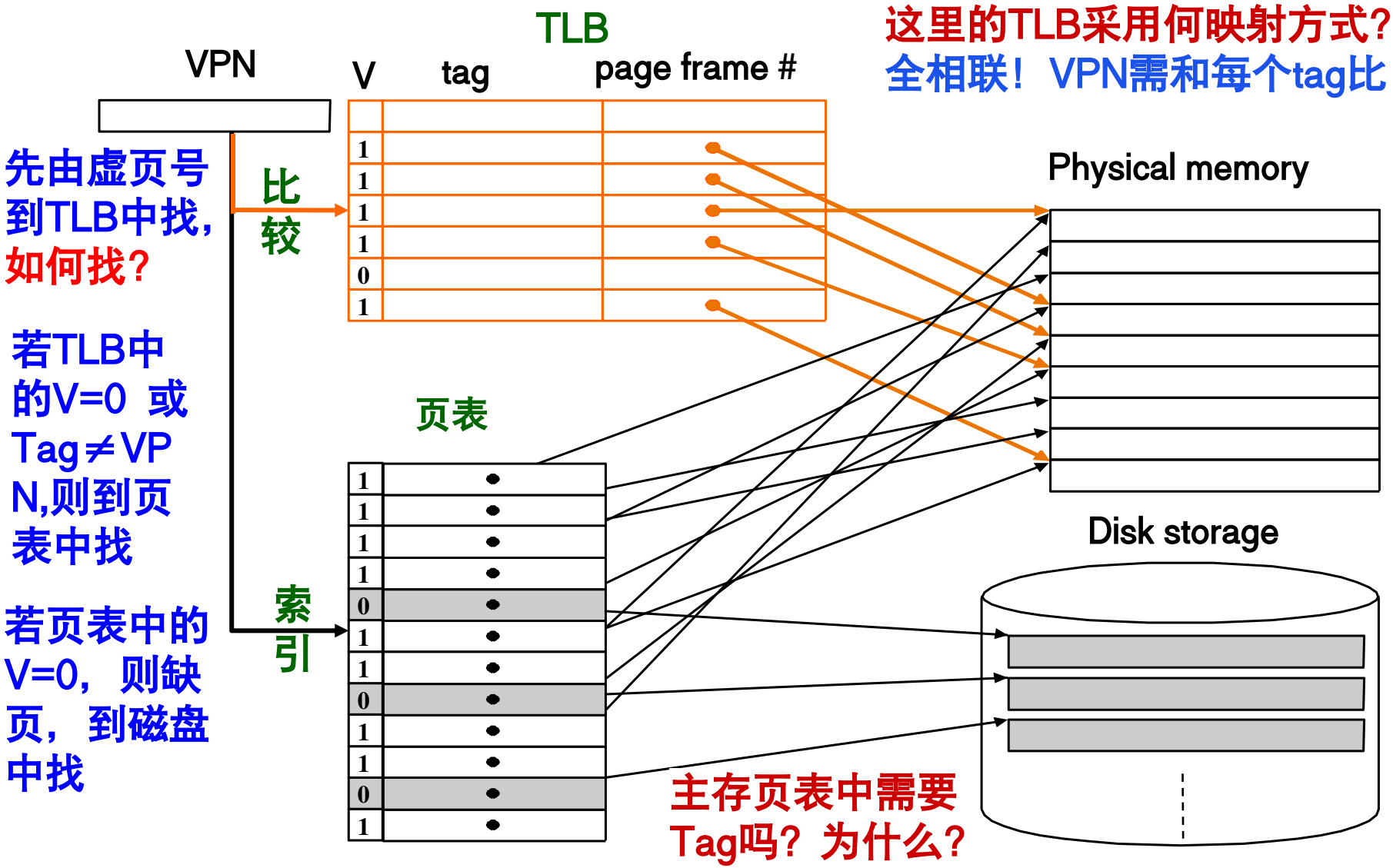
TLB中的页表项：tag+主存页表项

Virtual page num (tag)	Physical Address	Dirty	Ref	Valid	Access
	对应物理页框号				

CPU访存时，地址中虚页号被分成tag+index，tag用于和TLB页表项中的tag比较，index用于定位需比较的表项

TLB全相联时，没有index，只有Tag，虚页号需与每个Tag比较；
TLB组相联时，则虚页号高位为Tag，低位为index，用作组索引。

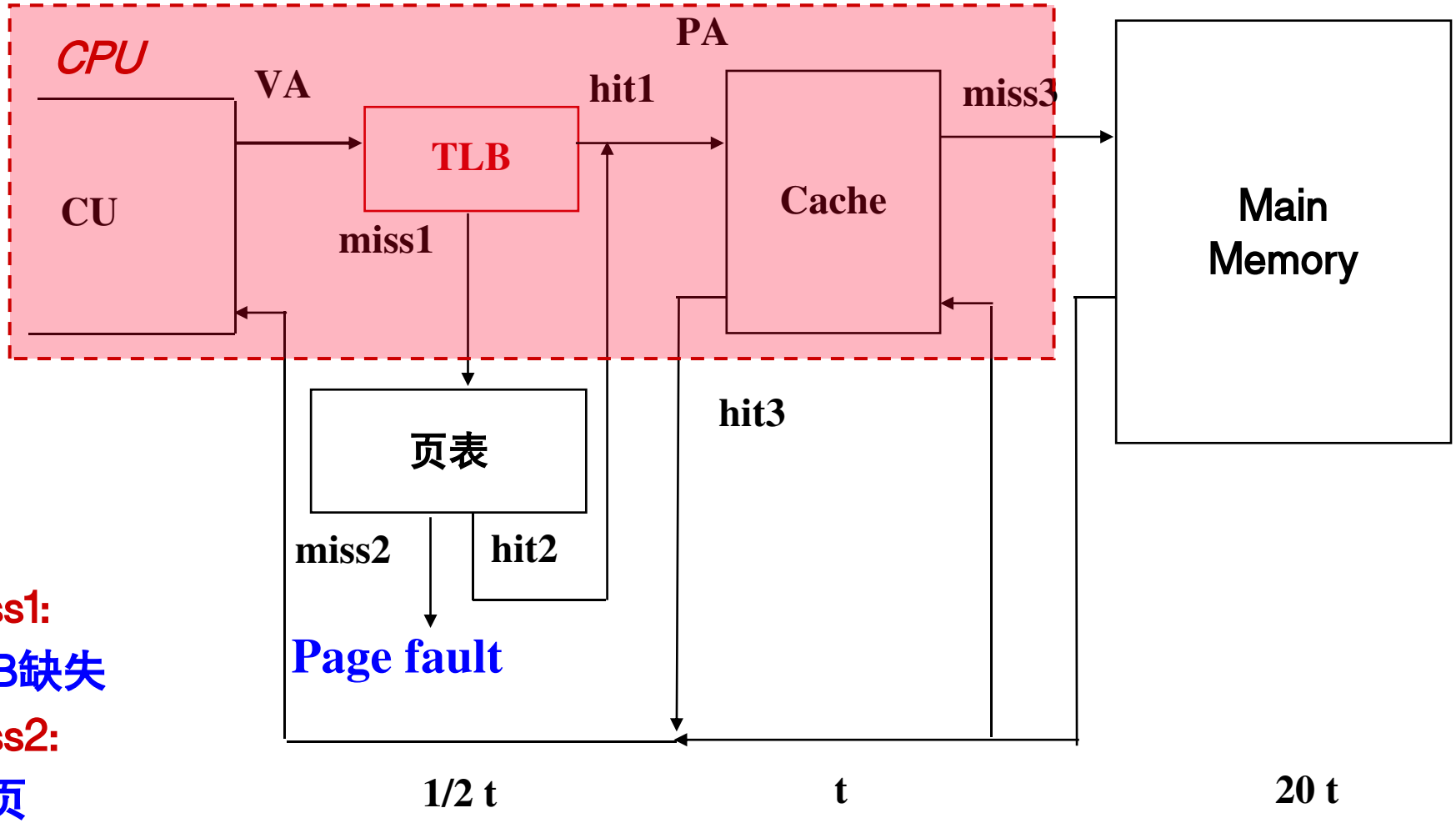
TLBs --- Making Address Translation Fast



层次结构存储系统

- 分以下五个部分介绍
  - 第一讲：虚拟存储器概述
    - 进程的虚拟地址空间、虚拟存储器的基本类型
  - 第二讲：页式虚拟存储器的实现
    - 页表和页表项的结构、页式存储管理总体结构
    - 页式虚拟存储地址转换、快表
  - 第三讲：具有TLB和cache的存储系统
    - 层次化存储系统结构、CPU的访存过程
    - cache的4种查找方式
    - 存储保护机制
  - 第四讲：存储系统实例
    - LoongArch架构虚拟存储系统
    - Core i7+Linux存储系统
    - Linux系统的存储器映射

Translation Look-Aside Buffers



Miss1:

TLB缺失

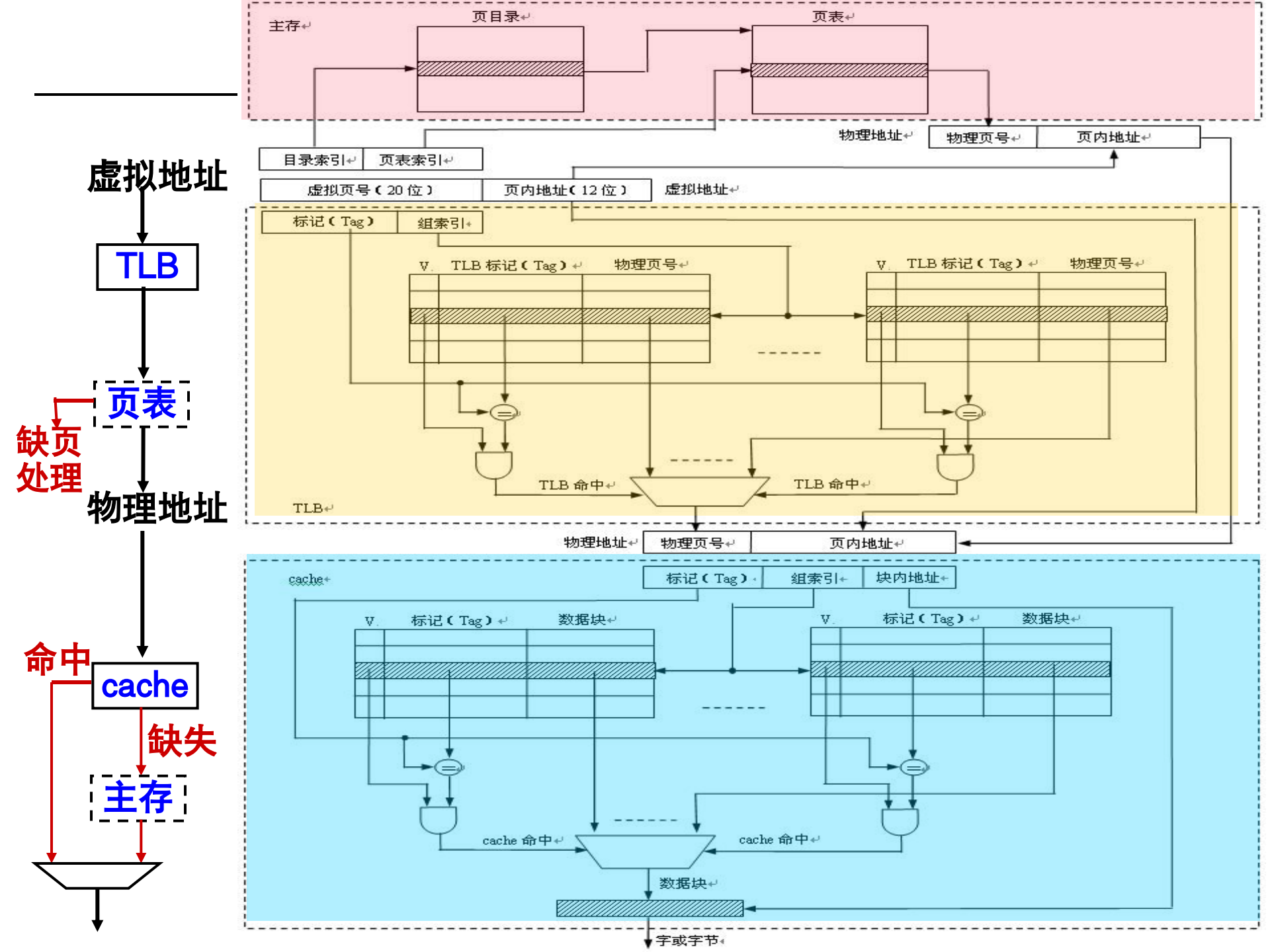
Miss2:

缺页

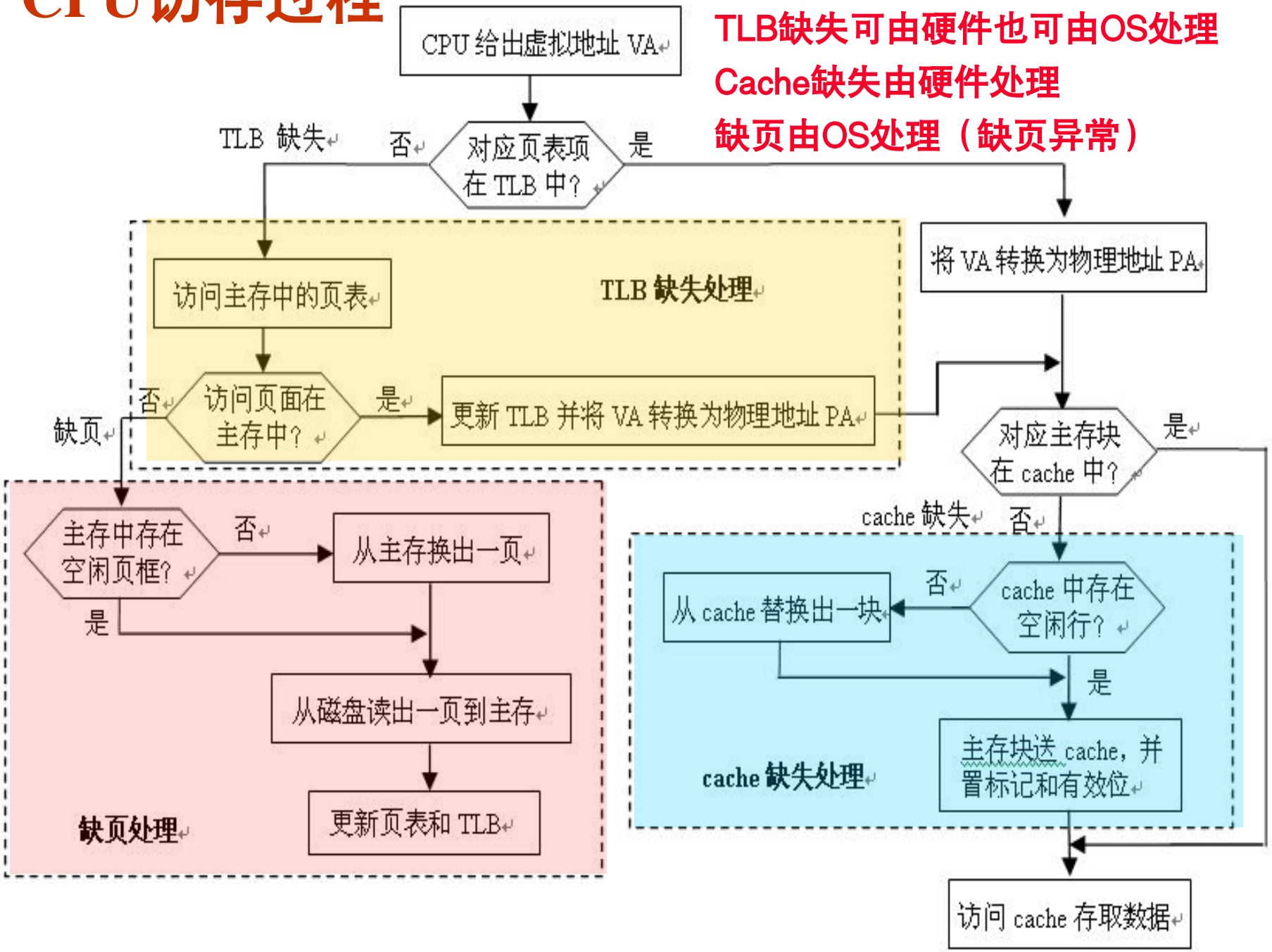
Miss3:

PA 在主存中，但不在Cache中

TLB冲刷指令和Cache冲刷指令都是操作系统使用的特权指令


虚拟地址
TLB
页表
缺页处理
物理地址
命中
cache
缺失
主存
主存
页目录
页表
物理地址
物理页号
页内地址
目录索引
页表索引
虚拟页号（20 位）
页内地址（12 位）
虚拟地址
标记（Tag）
组索引
V
TLB 标记（Tag）
物理页号
TLB 命中
TLB
物理地址
物理页号
页内地址
cache
标记（Tag）
组索引
块内地址
V
标记（Tag）
数据块
cache 命中
数据块
字或字节

CPU访存过程



TLB缺失可由硬件也可由OS处理

Cache缺失由硬件处理

缺页由OS处理（缺页异常）

举例：三种不同缺失的组合

TLB	Page table	Cache	Possible? If so, under what circumstance?
hit	hit	miss	可能，TLB命中则页表一定命中，但实际上不会查页表
miss	hit	hit	可能，TLB缺失但页表命中，信息在主存，就可能在Cache
miss	hit	miss	可能，TLB缺失但页表命中，信息在主存，但可能不在Cache
miss	miss	miss	可能，TLB缺失页表缺失，信息不在主存，一定也不在Cache
hit	miss	miss	不可能，页表缺失，信息不在主存，TLB中一定没有该页表项
hit	miss	hit	同上
miss	miss	hit	不可能，页表缺失，信息不在主存，Cache中一定也无该信息

最好的情况是hit、hit、hit，此时，访问主存几次？ 不需要访问主存！

以上组合中，最好的情况是？ hit、hit、miss和miss、hit、hit 访存1次

以上组合中，最坏的情况是？ miss、miss、miss 需访问磁盘、并访存至少2次

介于最坏和最好之间的是？ miss、hit、miss 不需访问磁盘、但访存至少2次

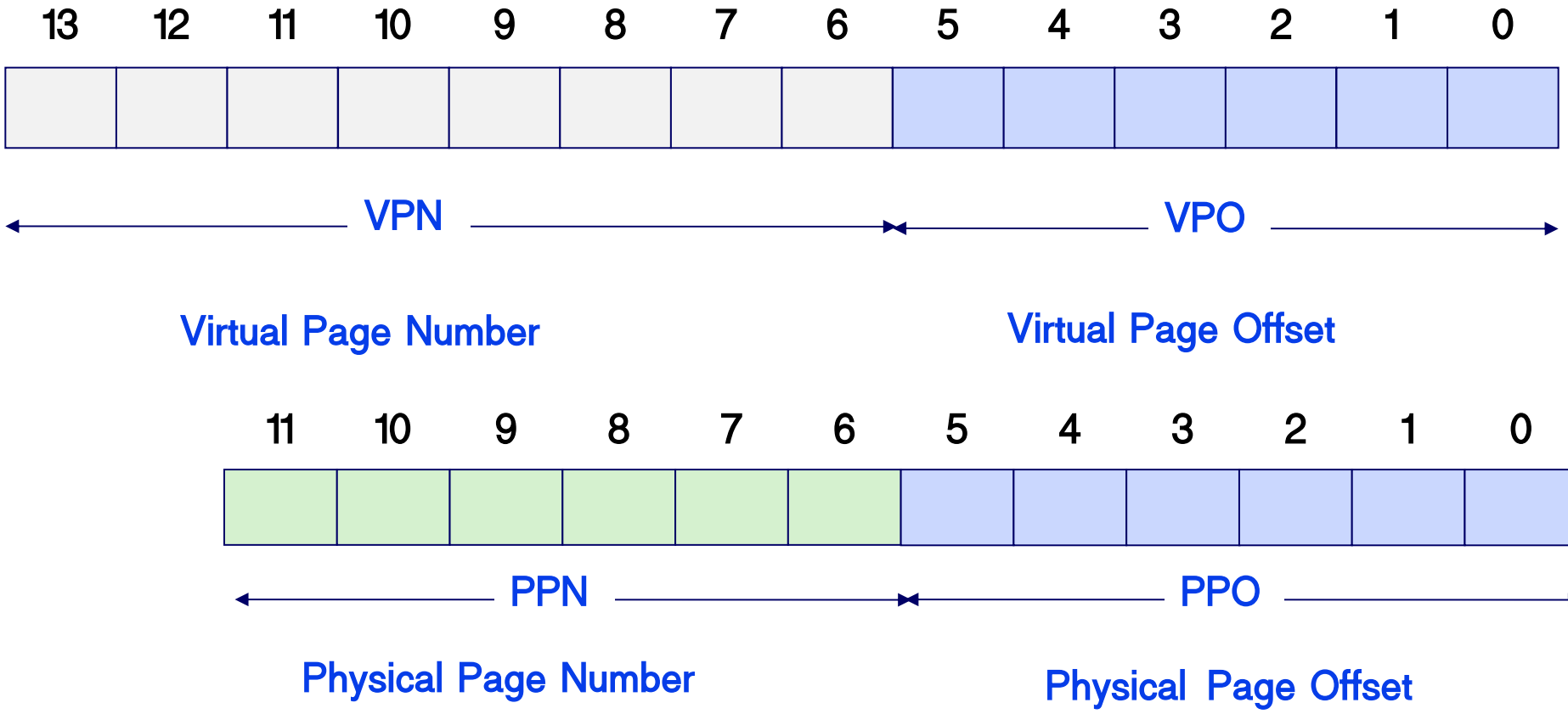
缩写的含义

- 基本参数（按字节编址）
  - $N = 2^n$：虚拟地址空间大小
  - $M = 2^m$：物理地址空间大小
  - $P = 2^p$ ：页大小
- 虚拟地址（VA)中的各字段
  - TLBI: TLB index（TLB索引）
  - TLBT: TLB tag（TLB标记）
  - VPO: Virtual page offset （页内偏移地址）
  - VPN: Virtual page number （虚拟页号）
- 物理地址(PA)中的各字段
  - PPO: Physical page offset（页内偏移地址）
  - PPN: Physical page number（物理页号）
  - CO: Byte offset within cache line（块内偏移地址）
  - CI: Cache index（cache索引）
  - CT: Cache tag（cache标记）

一个简化的存储系统举例

- 假定以下参数，则虚拟地址和物理地址如何划分？共多少页表项？
  - 14-bit virtual addresses（虚拟地址14位）
  - 12-bit physical address（物理地址12位）
  - Page size = 64 bytes（页大小64B）

页表项数应为：
$2^{14-6}=256$



一个简化的存储系统举例（续）

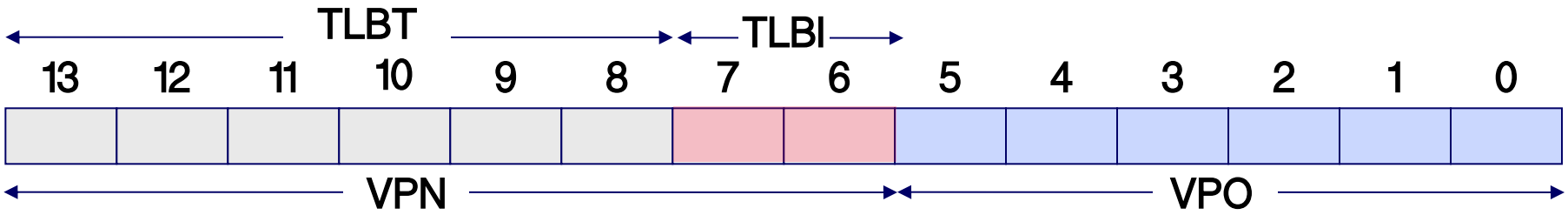
假定部分页表项内容（十六进制表示）如右：

红字处存在什么问题？

假定TLB如下：16 个TLB项，4路组相联，则TLBT和TLBI各占几位？

VPN	PPN	Valid
000	28	1
001	–	0
002	33	1
003	02	1
004	–	0
005	16	1
006	–	0
007	–	0

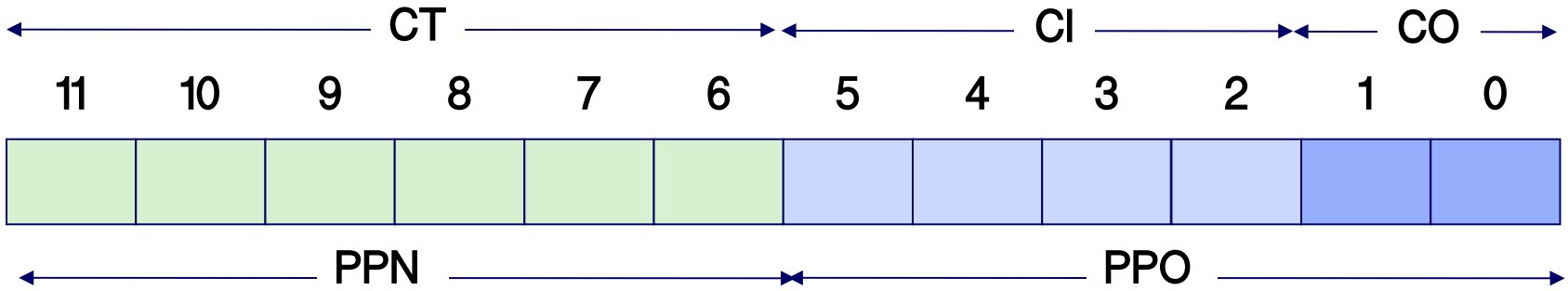
VPN	PPN	Valid
028	13	1
029	17	1
02A	09	1
02B	–	0
02C	–	0
02D	2D	1
02E	11	1
02F	0D	1



Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	08	0	00	–	0	07	02	1
1	0B	2D	1	0A	17	1	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	0B	0D	1	0A	34	1	02	–	0

一个简化的存储系统举例（续）

假定Cache的参数和内容（十六进制）如下：16 行，主存块大小为4B，直接映射，则主存地址如何划分?

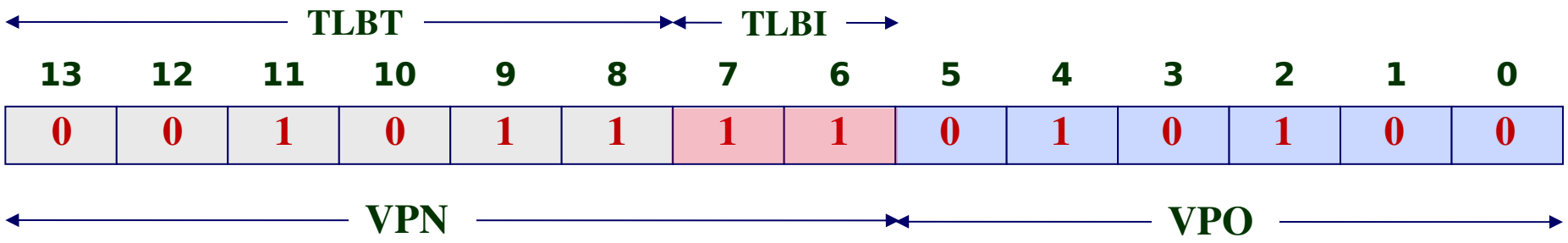


Idx	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	V	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	–	–	–	–

一个简化的存储系统举例（续）

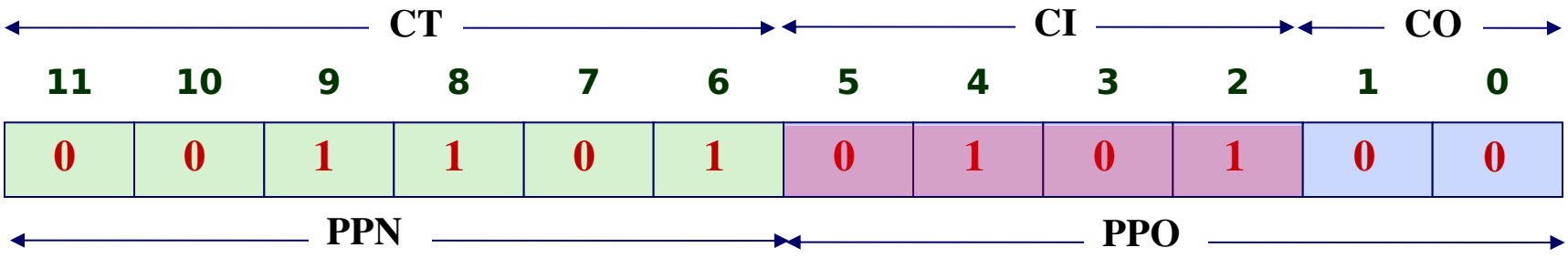
假设该存储系统所在计算机采用小端方式， CPU执行某指令过程中要求访问一个16位数据，给出的逻辑地址为0x0BD4，说明访存过程。



VPN 0x2F TLBI 0x3 TLBT 0x0B TLB Hit? Y Page Fault? N PPN: 0x0D

物理地址为

问题：逻辑地址为0x0A7A、0x0507时的访存过程如何？



CO 0 CI 0x5 CT 0x0D cache Hit? Y 数据: 0x7236

分段式虚拟存储器

- 分页方式：不同信息跨页 或 浪费空间
- 分段系统的实现
  - 程序有多个代码段和数据段构成，按照程序的逻辑结构划分，而形成多个相对独立的部分。

    （例如，代码段、只读数据段、可读写数据段等，按实际大小划分）
  - 段带有段名或基地址，便于程序编写、编译器优化和操作系统调度管理
  - 分段系统将主存空间按实际程序中的段来划分，每个段在主存中的位置记录在段表中，并附以“段长”项
  - 段表由段表项组成，段表本身也是主存中一个可再定位段

段式虚拟存储器的地址映像

段页式存储器

- 分页、分段方式各自的优缺点
  - 分页：浪费空间（按页边界划分段） 或
    无法设置权限（不按页边界划分，只读代码和可读写数据在同一页）
  - 分段：不同段分开易管理，但内存会形成大量碎片
- 段页式系统基本思想　段页式的缺点是什么？　开销大！慢!
  - 段、页式结合：
    - 程序的虚拟地址空间按模块分段、段内再分页，进入主存后仍以页为基本单位管理
  - 逻辑地址由段地址、页地址和偏移量三个字段构成
  - 用段表和页表（每段一个）进行两级定位管理
    - 先分段：根据段地址到段表中找到该段对应的页表首地址
    - 再分页：根据页号从页表中找到对应页框地址，形成物理地址

存储保护的基本概念

- 什么是存储保护?
  - 为避免多道程序相互干扰，防止某程序出错而破坏其他程序的正确性或不合法地访问其他程序或数据区，应对每个程序进行存储保护
- 操作系统程序和用户程序都需要保护
- 以下情况发生存储保护错
  - 地址越界（转换得到的物理地址不属于可访问范围）
  - 访问越权（访问操作与所拥有的访问权限不符）
    - 页表中设定访问（存取）权限
- 访问属性（权限）的设定
  - 数据段可指定R/W或RO；程序段可指定R/E或RO
- 最基本的保护措施:

  规定各道程序只能访问属于自己所在的存储区和共享区
  - 对于属自己存储区的信息：可读可写，只读/只可执行
  - 对共享区或已获授权的其他用户信息：可读不可写
  - 对未获授权的信息（如OS内核、页表等）：不可访问

内存访问时的异常信息


_360DesktopSwitcherControlWnd_MsgOnly_Wnd: 360DesktopSwitch64.e...
0x772440e8 指令引用的 0x191c17d8 内存。该内存不能为 read。
要终止程序，请单击“确定”。
确定


Microsoft Visual C++
Unhandled exception in Test.exe: 0xC0000005: Access Violation.
0xc0000005为出错ID
确定

存储保护的硬件支持

- 为了对操作系统的存储保护提供支持，ISA必须提供以下三种基本机制：
  - 支持至少两种运行模式：
    - 管理模式(Supervisor Mode)

      执行操作系统内核程序时处理器所处的模式称为管理（监管）模式(Supervisor Mode)，或称管理程序状态，简称管态、管理态、核心态、内核态
    - 用户模式(User Mode)

      CPU执行非内核的用户程序时，处理器所处的模式就是用户模式，或称用户状态、目标程序状态，简称为目态或用户态
  - 使一部分CPU状态只能由内核程序读写而不能由用户程序读写：这部分信息包括：页表、页表首地址、TLB等。OS内核可以用特殊的指令（一般称为管态指令或特权指令）来读写这些信息
  - 提供让CPU在管理模式（内核态）和用户模式（用户态）相互转换的机制：“异常”和“中断”使CPU从用户态转到内核态；异常/中断处理中的“返回”指令使CPU从内核态转到用户态
- 通过上述三个功能并把页表保存在OS的地址空间，OS就可以更新页表，并防止用户程序改变页表，确保用户程序只能访问由OS分配给的存储空间

层次结构存储系统

- 分以下五个部分介绍
  - 第一讲：虚拟存储器概述
    - 进程的虚拟地址空间、虚拟存储器的基本类型
  - 第二讲：页式虚拟存储器的实现
    - 页表和页表项的结构、页式存储管理总体结构
    - 页式虚拟存储地址转换、快表
  - 第三讲：具有TLB和cache的存储系统
    - 层次化存储系统结构、CPU的访存过程
    - cache的4种查找方式
    - 存储保护机制
  - 第四讲：存储系统实例
    - LoongArch架构虚拟存储系统
    - Core i7+Linux存储系统
    - Linux系统的存储器映射

实例：LoongArch架构虚拟存储系统

- LoongArch具有RISC的典型特征，存储访问除取指令外，还有Load/Store指令中的取数和存数操作。
- LoongArch定义了一系列控制状态寄存器（CSR），本书用CSR.%%.##的形式表示控制状态寄存器%%中的字段##。如，CSR.CRMD.PLV表示CRMD中的PLV字段。
- LoongArch的CSR具有独立地址空间，地址从0开始，占14位。
- 与虚拟存储管理相关的几个控制状态寄存器

  1）当前模式CSR

  2）与映射地址翻译相关的CSR，包括以下几类:

  直接映射地址翻译模式

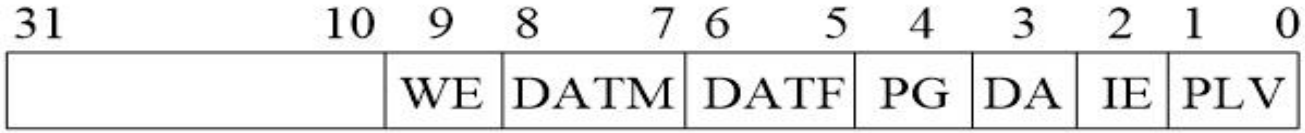
  常规信息

  非TLB重填异常

  TLB重填异常

  页表遍历

当前模式CSR：CRMD



PLV（Privilege LeVel）：当前特权级（特权模式）。00表示内核态（PLV0级）。CPU在异常/中断响应过程中，先将该位保存到相关控制状态寄存器中，然后将其设为0，以确保异常/中断响应后进入内核态（PLV0级）。执行ERTN指令从异常/中断处理返回时，再将相关控制状态寄存器中保存的PLV值恢复到CSR.CRMD.PLV中。

IE（Interrupt Enable）：当前全局中断使能位。该位为1，表示允许中断。CPU在异常/中断响应过程中，将该位设为0，以禁止响应新的中断（即关中断）。

DA：1表示当前为直接地址翻译模式。

PG：1表示当前为映射地址翻译模式。

DA和PG只能组合为0、1或1、0，即两种模式互斥

DATF：直接地址翻译模式下取指令操作的访存类型。

DATM：直接地址翻译模式下Load/Store操作的访存类型。

访存类型：00--强序非缓存(SUC)，01---一致可缓存(CC)，10--弱序非缓存(WUC)

WE：所有指令和数据监视点的全局使能位，为1时有效。CPU在异常/中断响应过程中，将该位设为0。

加电或复位时处于直接地址翻译模式，类似x86实地址模式，此时物理地址直接等于虚拟地址的PALEN-1:0位（不足补0）。初始化后转入映射地址翻译模式。映射地址翻译模式分为直接映射地址翻译（直接映射）模式和页表映射地址翻译（页表映射）模式两种。

PALEN为系统支持的物理地址位数，VALEN为系统支持的虚拟地址位数

LA的虚拟地址空间

LA64最大虚拟空间为2^{64}字节，范围为0~$2^{VALEN-1}$。理论上VALEN小于等于64，具体通过配置设定，常见在[40，48]范围内

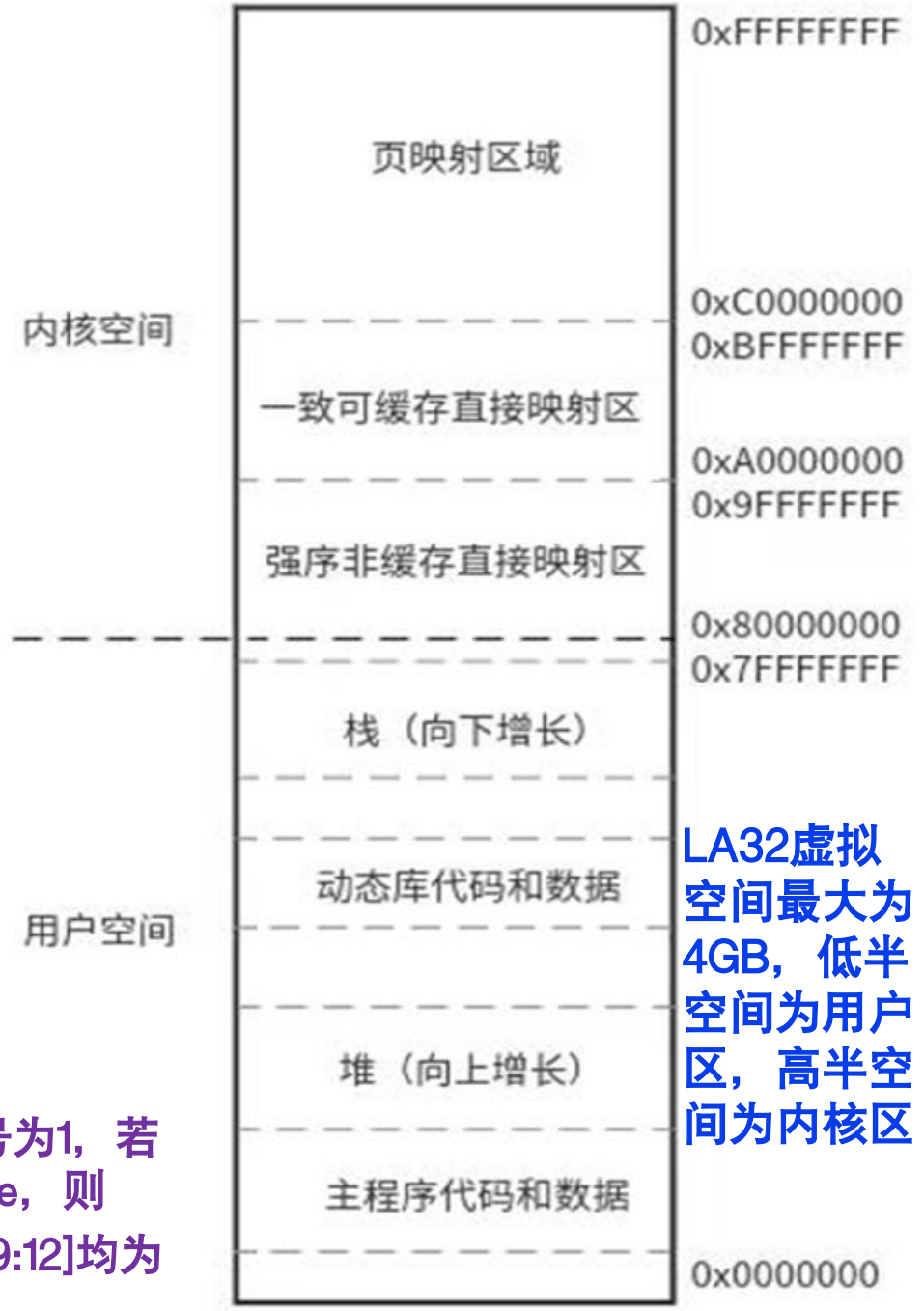
LA64下，采用**页表映射模式**时，合法虚拟地址[63:VALEN]中每位须与VALEN−1位相同，即符号扩展，否则发生地址错异常；采用**直接映射模式**时，无须合法性判断

VALEN和PALEN等**指令系统实现的功能特性**记录在一系列**配置信息字**中，可通过执行**CPUCFG指令**读取一个配置信息字

每个配置信息字中包含若干配置信息字段，其表示形式为CPUCFG.<配置字号>.<配置信息助记名>[位下标]。例如，CPUCFG.1.PALEN[11:4]和CPUCFG.1.VALEN[19:12]中的配置信息分别为PALEN−1和VALEN−1的值。

```
addi.w   $r13, $r0, 1      说明访问的配置字号为1，若
cpucfg   $r12, $r13        r12中内容为0x3f2f2fe，则
```

CPUCFG.1.PALEN[11:4]和CPUCFG.1.VALEN[19:12]均为0010 1111B=47，即PALEN和VALEN值为48

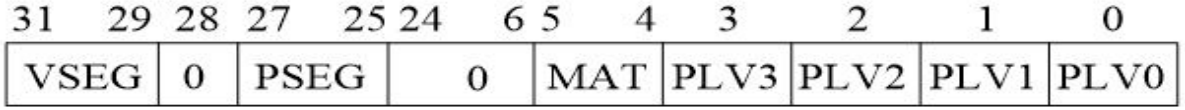


LA32虚拟空间最大为4GB，低半空间为用户区，高半空间为内核区

LA32中直接映射地址翻译模式

- 直接映射地址翻译模式常用于内核程序，例如，LA32内核空间中有一个一致可缓存直接映射区和一个强序非缓存直接映射区。
- 直接映射配置窗口寄存器CSR.DMW0～CSR.DMW3用于设定直接映射地址翻译时使用的参数。

前两个CSR用于取指令和Load/Store操作；后两个仅用于Load/Store操作

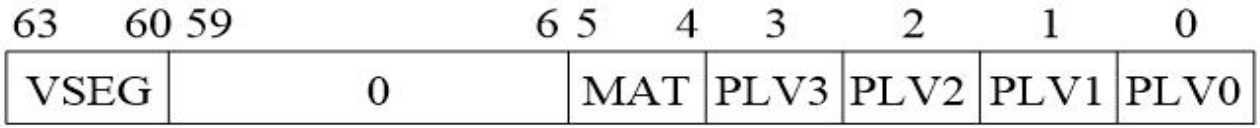


(a) LA32 架构下 DMW0~DMW3 寄存器定义

PLV0～PLV3：可用特权级，为1表示在特权级PLVn下可使用对应窗口的配置信息进行直接映射地址翻译；MAT：访存类型；PSEG：在LA32中对应物理地址的[31:29]位；VSEG：在LA32中对应虚拟地址的[31:29]位。

LA32中，若访存虚拟地址高3位与某配置窗口中设定的VSEG字段相等，且在该配置窗口中当前特权级对应的PLVn为1，则其物理地址等于虚拟地址[28:0]位与该配置窗口所设定的PSEG进行拼接。例如，若CSR.DMW0配置为0x8000 0001，则当前特权级为PLV0时，执行某取指令操作或Load/Store操作，只要其虚拟地址在0x8000 0000～0x9FFF FFFF之间，就将其直接映射到物理地址空间0x0～0x1FFF FFFF，其访存类型为00（强序非缓存），这段区域正好对应LA32虚拟地址空间中内核空间的强序非缓存直接映射区

LA64中直接映射地址翻译模式



(b) LA64 架构下 DMW0~DMW3 寄存器定义

PLV0～PLV3：可用特权级，为1表示在特权级PLVn下可使用对应窗口的配置信息进行直接映射地址翻译；MAT：访存类型；VSEG：在LA64中对应虚拟地址的[63:60]位。

LA64下，每个配置窗口可设置一个2^{PALEN}字节的虚拟地址空间

当访存虚拟地址高4位与配置窗口设定的VSEG相等，且当前特权级对应的PLVn为1，则物理地址等于虚拟地址的[PALEN−1:0]位

例如，若PALEN=48且CSR.DMW0为0x9000 0000 0000 0011，则当前特权级为PLV0时执行某取指令操作或Load/Store操作，只要其虚拟地址在0x9000 0000 0000 0000～0x9000 FFFF FFFF FFFF之间，就将其直接映射到物理地址空间0x0～0xFFFF FFFF FFFF，其访存类型为一致可缓存。

页表映射模式下的TLB访问

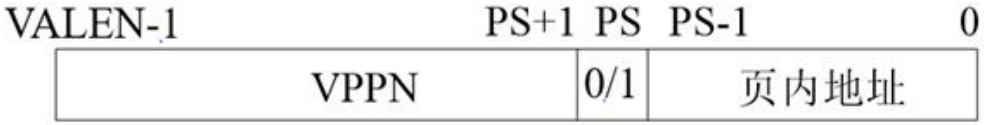
- 映射地址翻译模式下，所有在直接映射配置窗口中设置范围之外的虚拟地址都必须通过页表映射完成虚实地址转换。
- 使用TLB可加速页表映射地址翻译模式下的虚实地址转换过程

1）基于TLB的虚拟地址划分

LA中有采用单一固定页大小的STLB和支持不同页大小的MTLB ， MTLB采用全相联方式，STLB采用组相联方式

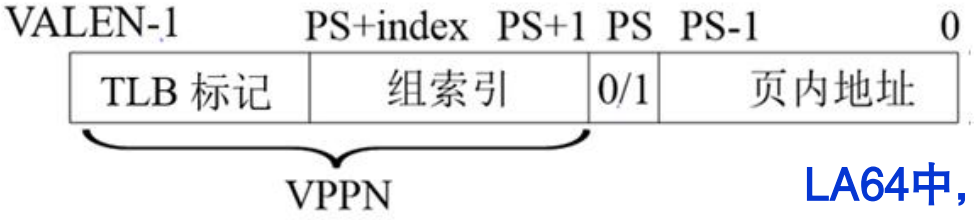
两种TLB表项中都存放一对相邻页表的页表项，即每个TLB表项中总是包含相邻偶数页（页号最低位为0）和奇数页（页号最低位为1）的两个页表项

相邻奇–偶虚拟号除最低位外的其余高位称为虚双页号（记为VPPN）



(a) 基于 MTLB 的虚拟地址划分

虚页号部分含VPPN字段（即TLB标记）和虚页号的最低位



(b) 基于 STLB 的虚拟地址划分

对于4路组相联STLB，一个TLB组中有4个TLB表项，共含8个页表项

LA64中，若VALEN为48，页大小为16KB，则PS =14，虚页号34位。若STLB有256组，4路组相联，则先据VA[22:15]位的组索引定位TLB组，再根据第14位0或1选择偶数或奇数页对应的页表项进行处理

页表映射模式下的TLB访问

2）TLB表项的结构

STLB和MTLB的表项结构基本一致，区别仅在于MTLB每个表项需包含页大小信息，而STLB中的页大小固定，由系统软件配置在CSR.STLBPS.PS字段中

VPPN	PS	G	ASID	E

PPN0	RPLV0	PLV0	MAT0	NX0	NR0	D0	V0

PPN1	RPLV1	PLV1	MAT1	NX1	NR1	D1	V1

TLB表项高位部分用于查找匹配，以确定是否TLB命中，各字段具体含义说明如下：

VPPN：虚双页号；PS：指示页大小，仅在MTLB中使用；G：全局标志位，当OS需在所有进程间共享同一虚拟页时，G=1，此时无须对ASID匹配检测；ASID：地址空间标识，占10位。OS为每个进程分配唯一的ASID，TLB表项查找时需比较ASID，以减少进程切换而清空整个TLB所带来的性能损失；E：1表示所在TLB表项非空，可进行查找匹配

TLB表项低位部分存放偶-奇相邻页的页表项，各字段的具体含义说明如下：

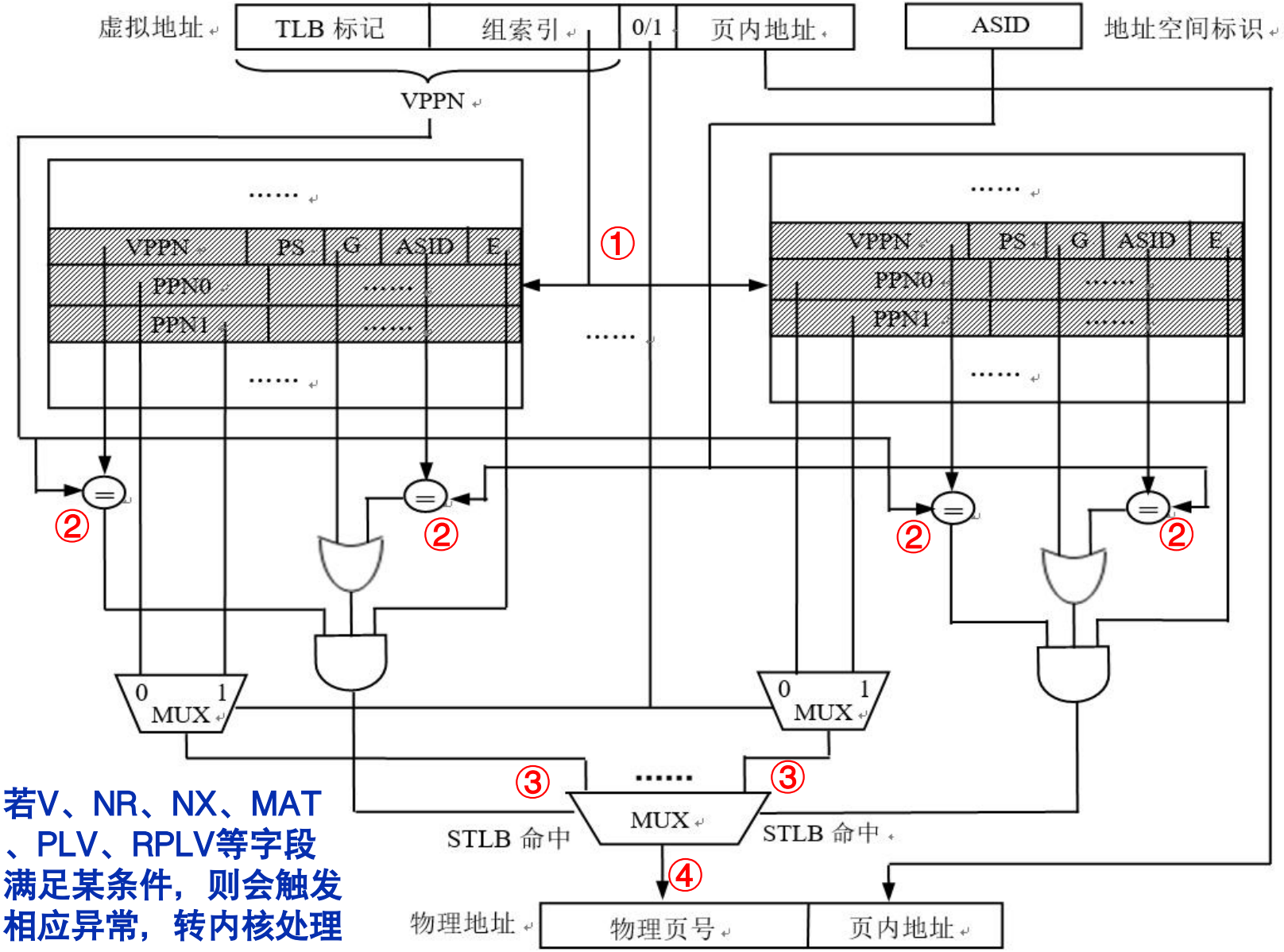
PPN：物理页号；RPLV：受限特权级使能位，0表示该页表项可被任何特权级不低于PLV的程序访问；1表示仅可被特权级等于PLV的程序访问，仅在LA64中有效；PLV：该页表项对应的特权级；MAT：在该页表项对应地址空间中进行访存操作时的访存类型；NX：不可执行位，1表示不可在该页表项所在地址空间中执行取指令操作，仅在LA64中有效；NR：不可读位，1表示不可在该页表项所在地址空间中执行Load操作，仅在LA64中有效；D：脏位，1表示该页表项所对应地址范围内已有脏数据；V：有效位，1表示有效且被访问过

页表映射模式下的TLB访问

3）基于TLB的虚拟地址到物理地址的转换

转换过程由MMU完成，可同时实现对STLB和MTLB的匹配查找，软件需保证不会使MTLB和STLB同时命中，否则处理器行为将不可知

右边是STLB命中时虚拟地址到物理地址转换过程示意图



若V、NR、NX、MAT、PLV、RPLV等字段满足某条件，则会触发相应异常，转内核处理

页表映射模式下的TLB访问

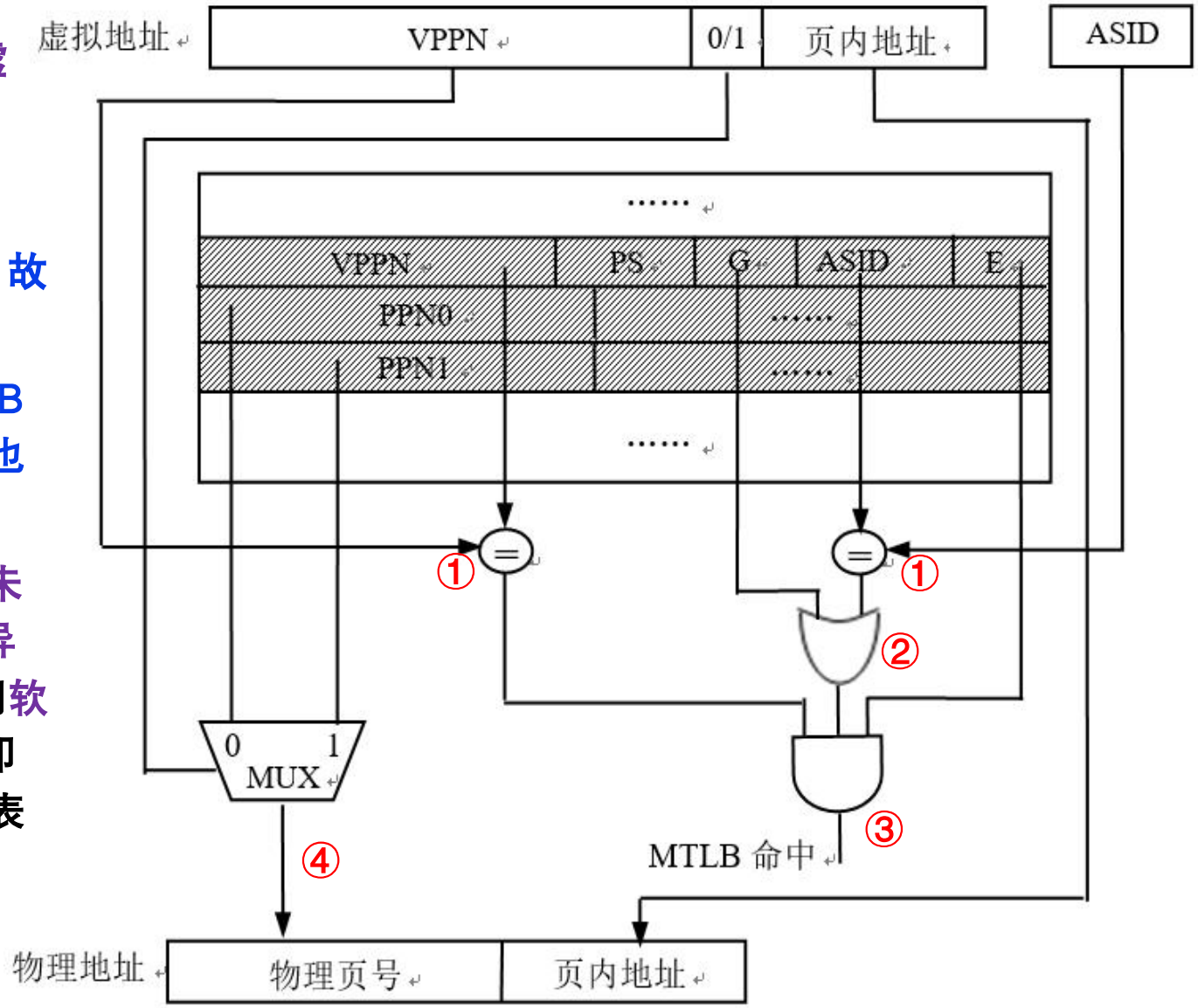
3）基于TLB的虚拟地址到物理地址的转换

右边是MTLB命中时，虚拟地址到物理地址转换过程示意图

MTLB采用全相联方式，故需比对所有TLB表项

MTLB命中的条件与STLB相同，触发的异常类型也与STLB一样

若在STLB和MTLB中都未命中，则触发TLB重填异常。LoongArch默认采用软件完成TLB重填异常，即由异常处理程序进行页表遍历并进行TLB填入



页表映射模式下的TLB访问

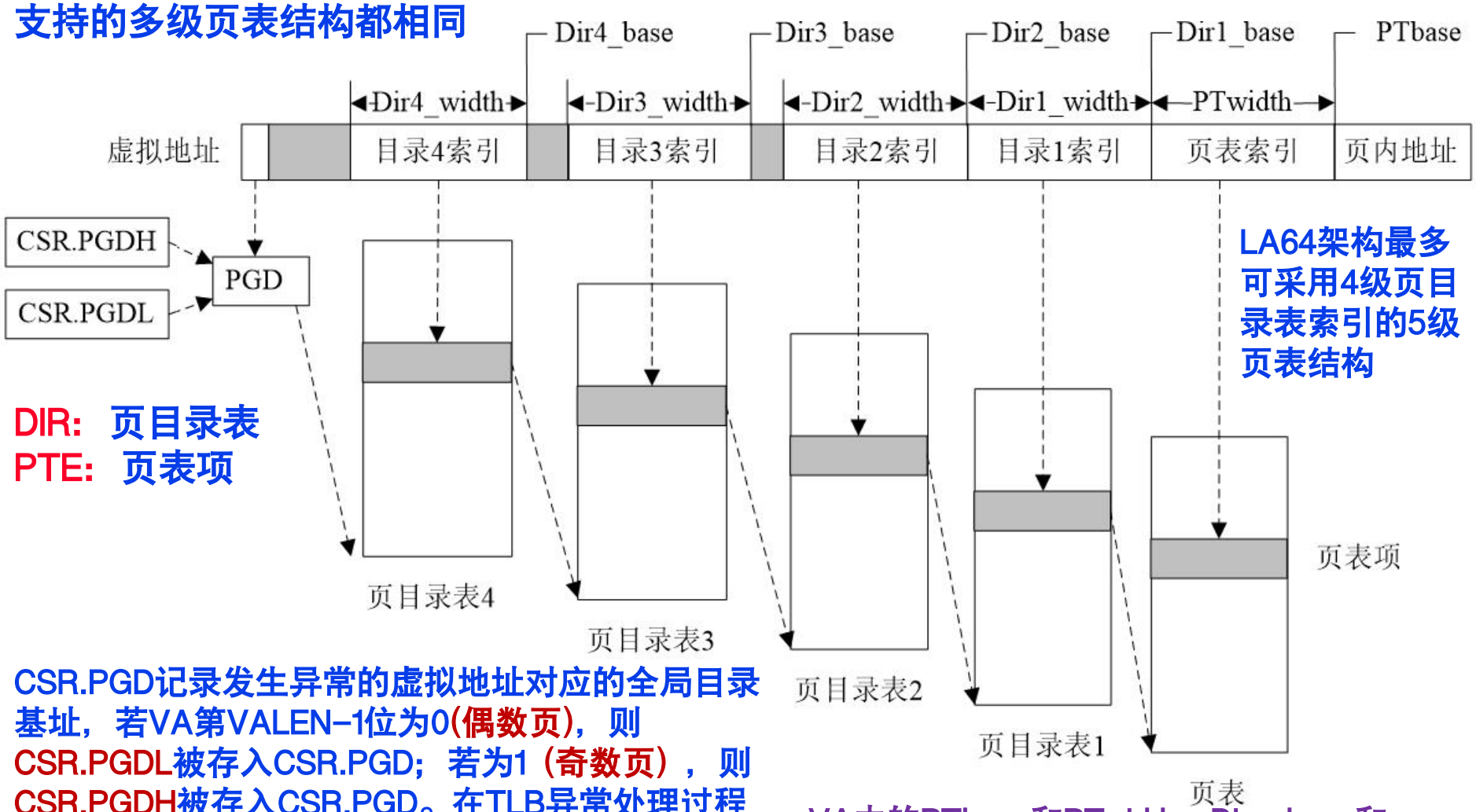
4）基于TLB的虚实地址转换过程中的异常事件

MMU通过TLB进行虚实地址转换过程中，若没找到匹配的TLB表项，或者尽管有匹配的TLB表项，但其中的页表项无效（V=0）或访问权限和特权级等不相符，则会触发异常，从而转到操作系统内核或其它监管程序进行异常处理

- MTLB和STLB都miss时，触发TLB重填（TLBR）异常
- 若取指令、Load取数、Store存数的地址转换过程发生TLB命中但V=0，则分别触发取指令页无效（PIF）、Load页无效（PIL）和Store页无效（PIS）三种异常
- 若TLB命中且对应V=1，但其特权级不合规，则触发页特权级不合规（PPI）异常页特权级不合规条件：页表项RPLV=0且当前特权级（CSR.CRMD.PLV）大于页表项PLV值，或者 页表项RPLV=1且当前特权级不等于页表项PLV值
- 若Store操作时TLB命中且对应V=1、特权级合规，但在当前特权级为PLV3或当前特权级不是PLV3但对应特权级的CSR.MISC.DWPL=0（未开启禁止写允许检查功能）的前提条件下，页表项D=0，则触发页修改（PME）异常
- 当Load操作时TLB命中且对应V=1、特权级合规，但该页表项中NR=1，将触发页不可读（PNR）异常
- 当取指令操作时TLB命中且对应V=1、特权级合规，但该页表项中NX=1，将触发页不可执行（PNX）异常

页表映射模式下的多级页表结构

LoongArch中，使用LDDIR和LDPTE指令实现的软件页表遍历和硬件页表遍历所支持的多级页表结构都相同



LA64架构最多可采用4级页目录表索引的5级页表结构

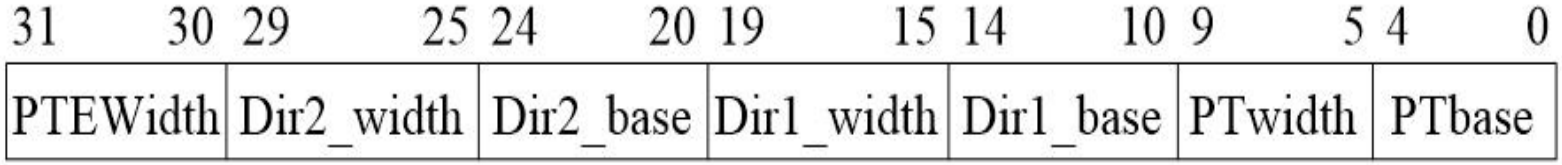
DIR：页目录表
PTE：页表项

CSR.PGD记录发生异常的虚拟地址对应的全局目录基址，若VA第VALEN-1位为0(偶数页)，则CSR.PGDL被存入CSR.PGD；若为1（奇数页），则CSR.PGDH被存入CSR.PGD。在TLB异常处理过程中需进行页表遍历，被遍历页表最顶层目录就是全局目录，其基地址PGD来自CSR.PGD寄存器

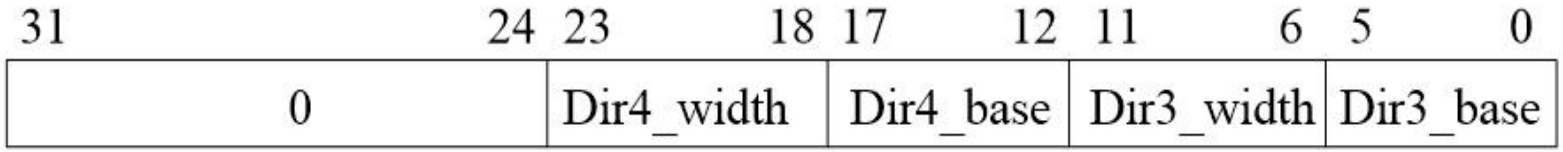
VA中的PTbase和PTwidth、Dir*n*_base和Dir*n*_width（*n*=1、2、3、4），都由OS在CSR.PWCL和CSR.PWCH中进行配置

页表映射模式下的多级页表结构

控制状态寄存器CSR.PWCL和CSR.PWCH的结构如下



(a) CSR.PWCL 寄存器的结构



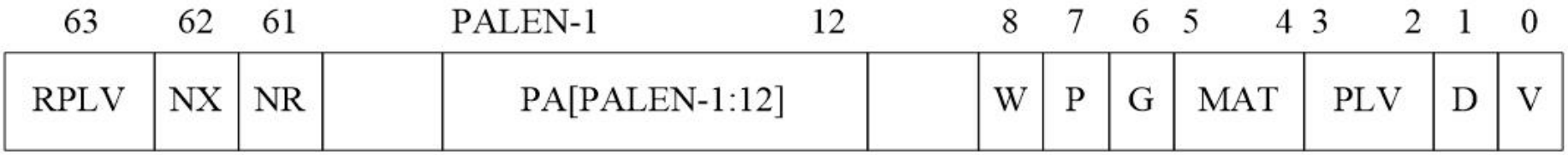
(b) CSR.PWCH 寄存器的结构

PTEWidth表示页表项位宽，由操作系统在CSR.PWCL寄存器中配置，占2位，为00、01、10、11时，分别表示页表项位宽为64、128、256、512

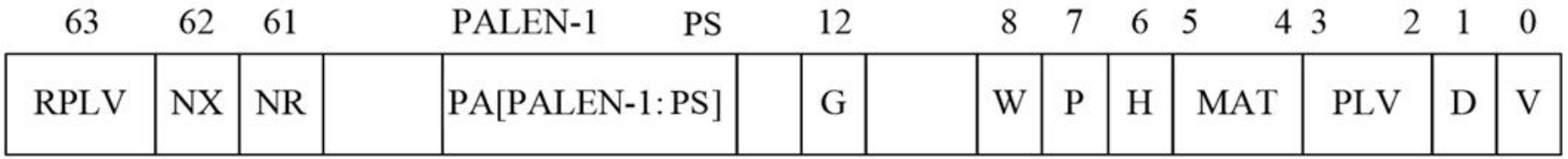
VA中各级索引位之间不一定连续，因此需定义各级索引的起始位，但实现中各级索引位之间通常是连续的

页表映射模式下的多级页表结构

页表项位宽为64位时，分基本页页表项格式、大页页表项格式



(a) 基本页页表项格式



(b) 大页页表项格式

页表项中P（存在位）、W（可写位）分别表示是否已装入主存、是否可写，这两位不填入TLB表项中，仅用于页表遍历。其他字段含义与TLB表项中相同，其中，PA[PALEN–1:12]和PA[PALEN–1:PS]就是TLB表项中的PPN（物理页号），因此基本页的物理页号从物理地址第12位开始，即基本页的页大小为4KB

两种格式区别：大页页表项第6位是H，为1表示大页；基本页页表项第6位是G，说明基本页页表项G一定为0，即基本页页表项格式肯定不用于共享的全局页，而大页页表项格式可用于全局页也可用于非全局页。

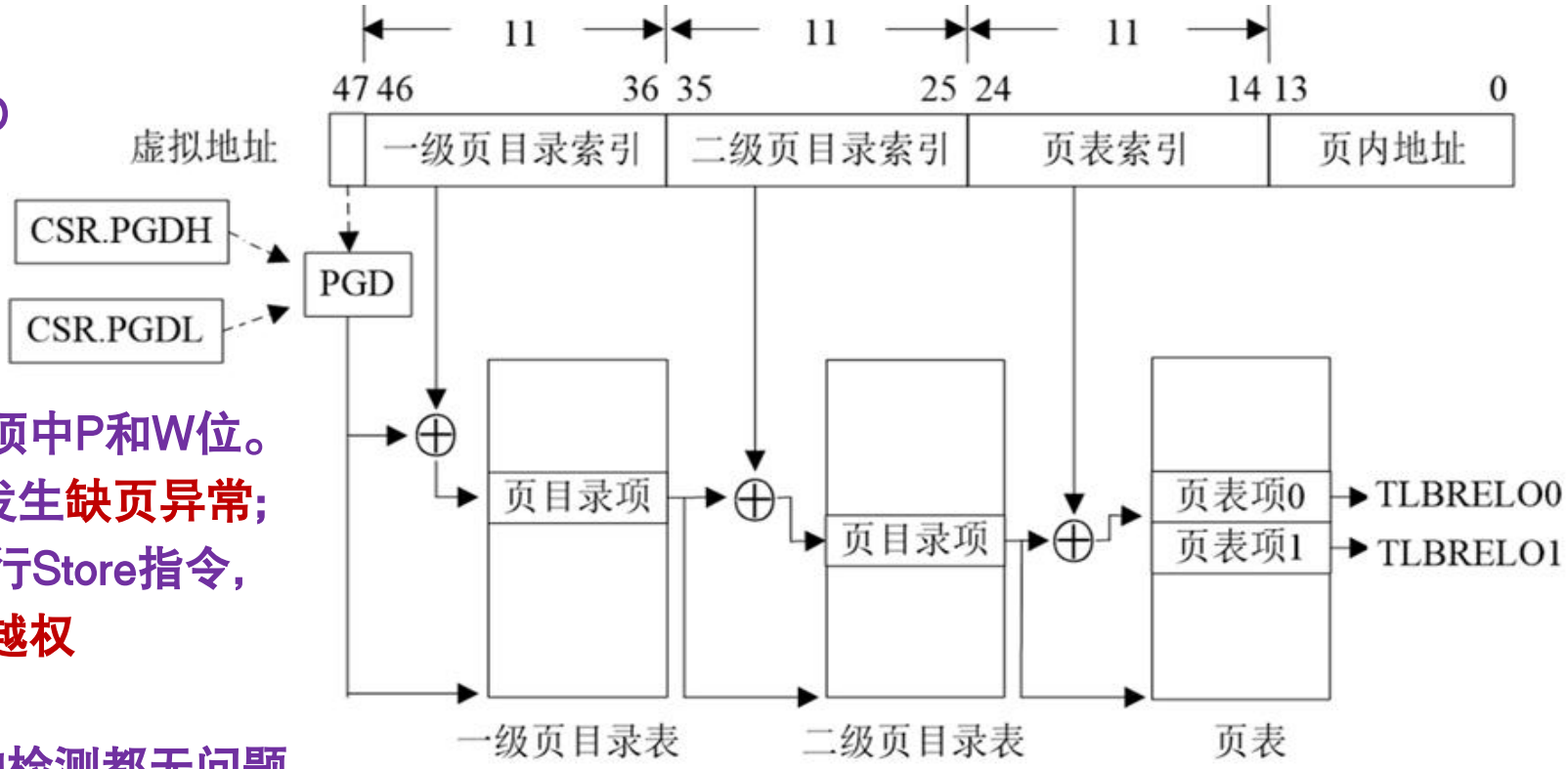
页表映射模式下的多级页表遍历

在TLB重填异常处理中，需基于CSR.TLBRBADV中的虚拟地址进行多级页表的遍历

① 根据PGD和各级索引找到对应页表项

②检测页表项中P和W位。若P=0，则发生缺页异常；若W=0且执行Store指令，则发生访问越权

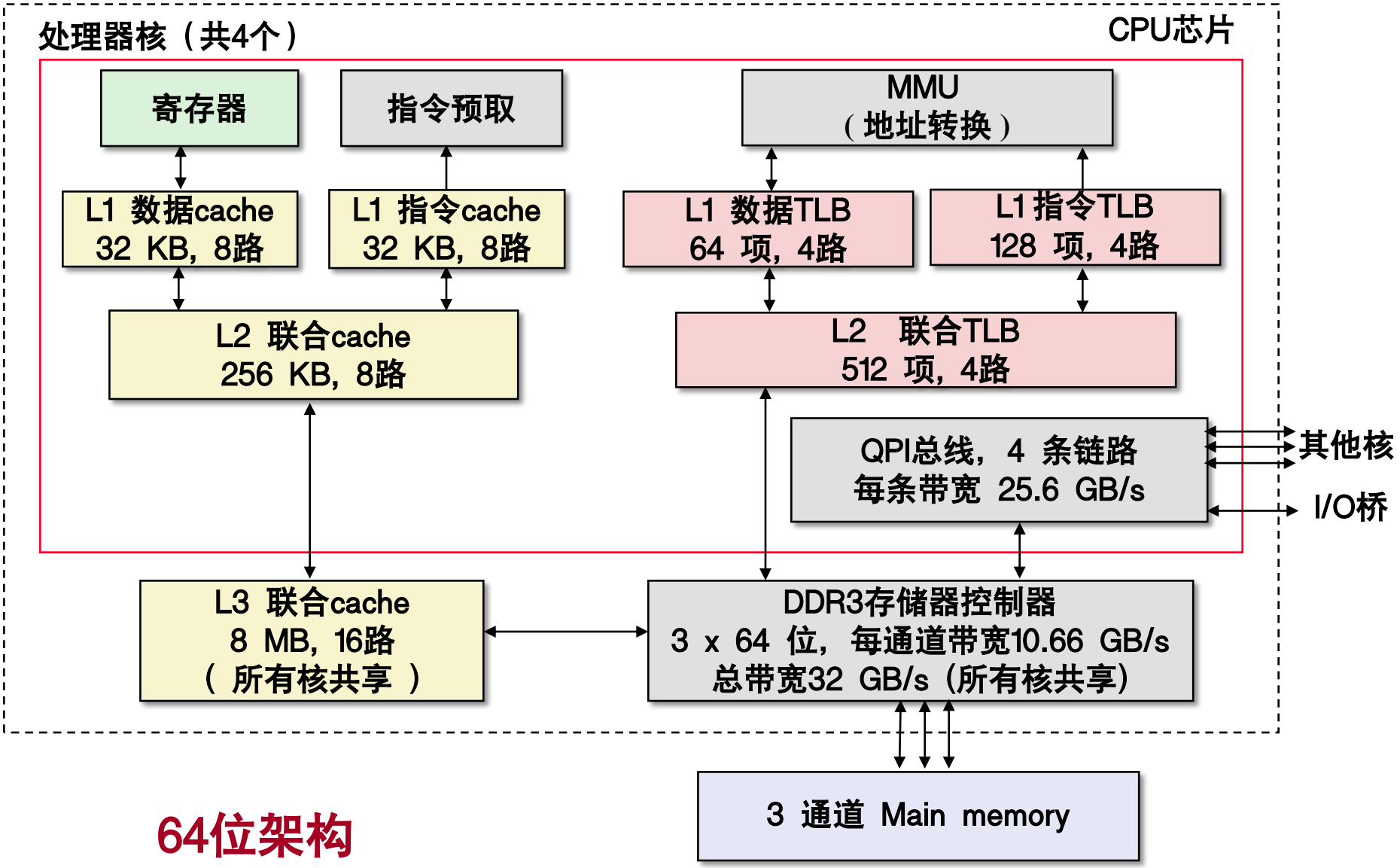
③对P和W的检测都无问题时，将包含当前页表项在内的相邻偶数–奇数页的页表项内容分别写入寄存器CSR.TLBRELO0和CSR.TLBRELO1



TLBRELO0	RPLV	NX	NR	0	PPN	0	G	MAT	PLV	D	V
TLBRELO1	RPLV	NX	NR	0	PPN	0	G	MAT	PLV	D	V

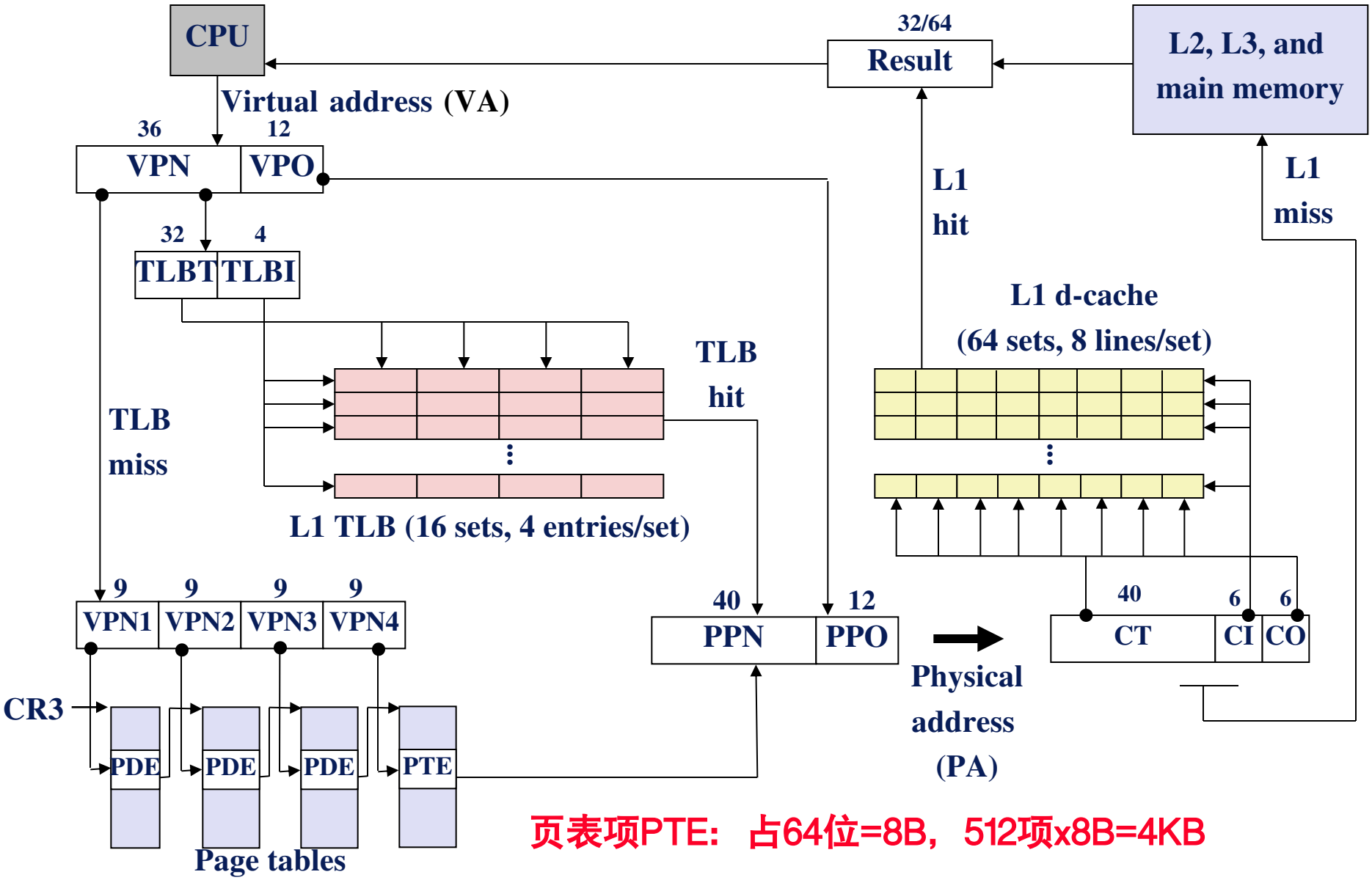
多级页表的遍历可通过LDDIR和LDPTE指令实现软件页表遍历，也可采用硬件页表遍历方式

实例： Intel Core i7+Linux存储系统



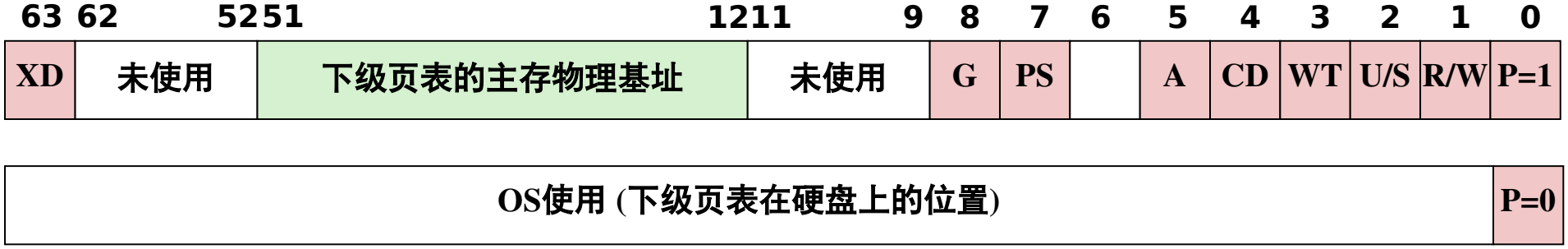
64位架构

End-to-end Core i7 Address Translation



页表项PTE：占64位=8B，512项x8B=4KB

Core i7 Level 1-3 Page Table Entries



P: 存在位，P=1表示对应的下级页表在主存中

R/W: 所表示范围内所有信息的读/写访问权限

U/S: 所表示范围内所有信息是否可被用户进程访问，为0表示用户进程不能访问；为1允许用户进程访问

WT: 指示下级页表对应的cache写策略是通写还是回写

CD: 指示下级页表能否缓存到cache中

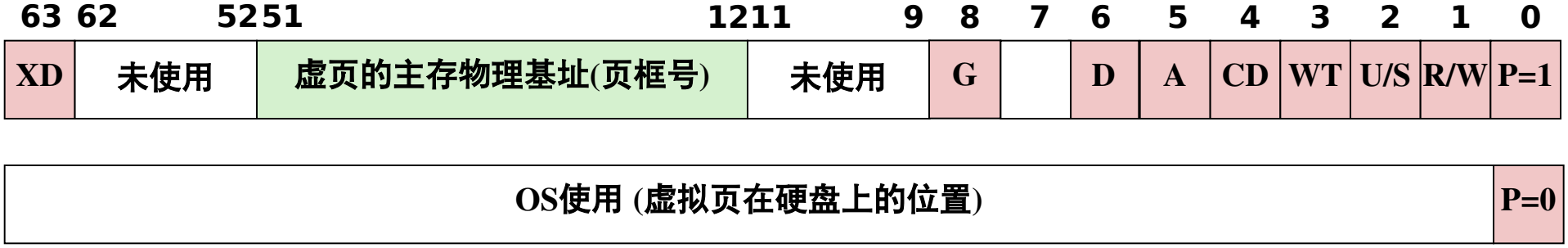
A: A=1表示下级页表被访问过，初始化时操作系统将其清0。由MMU在进行地址转换时将该位置1，由软件清0

PS: 设置页大小为4 KB、2MB或1GB，仅在二级页表或三级页表的表项中有定义

G: 设置是否为全局页面。全局页面在进程切换时不会从TLB中替换出去

下级页表物理基址: 用来表示下级页表在主存中的页框号，即主存地址的高40位（强迫各级页表在主存都按4KB对齐）

Core i7 Level 4 Page Table Entries



P：存在位，P=1表示对应的虚拟页在主存中

R/W：所表示范围内所有信息的读/写访问权限

U/S：所表示范围内所有信息是否可被用户进程访问，为0表示不能访问；为1允许访问。若用户进程欲访问操作系统页面，则会发生访问越级

WT：指定对应页的cache写策略是通写还是回写

CD：指定对应页能否缓存到cache中

A：A=1表示对应页面被访问过，初始化时操作系统将其清0。由MMU在进行地址转换时将该位置1，由软件清0

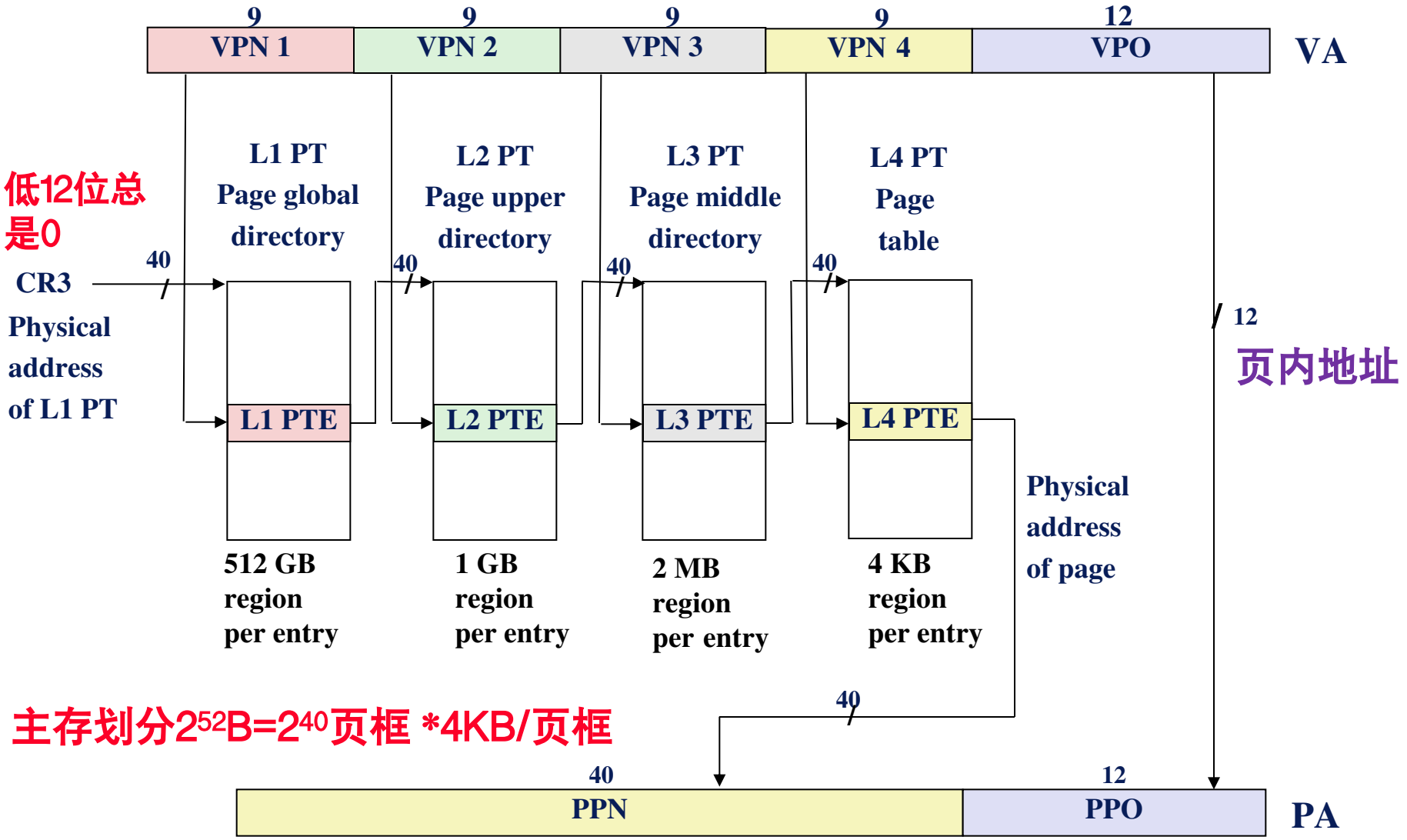
D：脏位（或称修改位），进行写操作时由MMU将该位置1，由软件清0

G：设置是否为全局页面。全局页面在进程切换时不会从TLB中替换出去

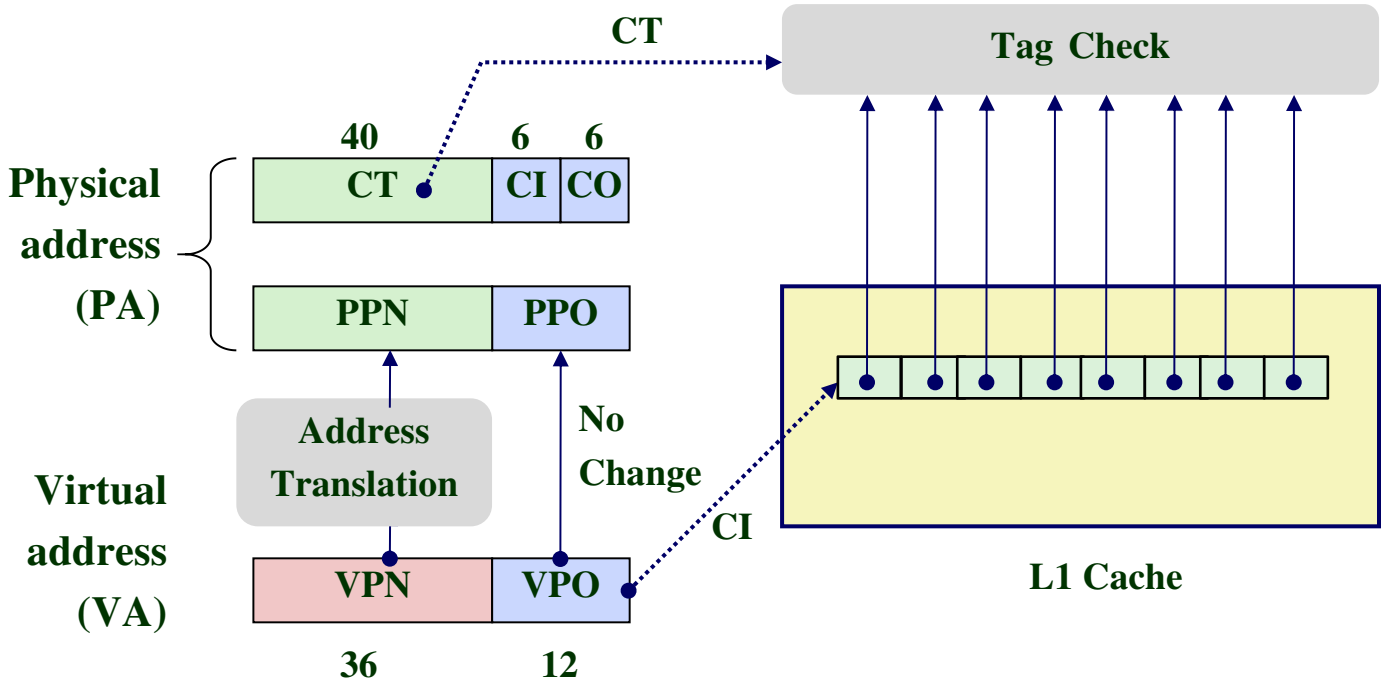
页表项第51~12位：用来表示对应页在主存中的页框号，即主存地址的高40位，因此，所有页面在主存中的起始地址低12位为全0，即所有页面在主存都按4KB对齐

Core i7 Page Table Translation

线性地址空间划分：2^{48}B=512* 512 *512 *512 * 4KB/页



Cute Trick for Speeding Up L1 Access



VIPT(Virtually indexed, physically tagged)：这里，VPO和PPO相同，Cache索引（CI）和块内地址（CO）合起来位数等于PPO，故CI直接使用VPO，不用等转换为物理地址后再进行Cache索引，但Cache标志（CT）由地址转换得到的PPN（物理地址）确定，VIPT只在L1 cache中采用

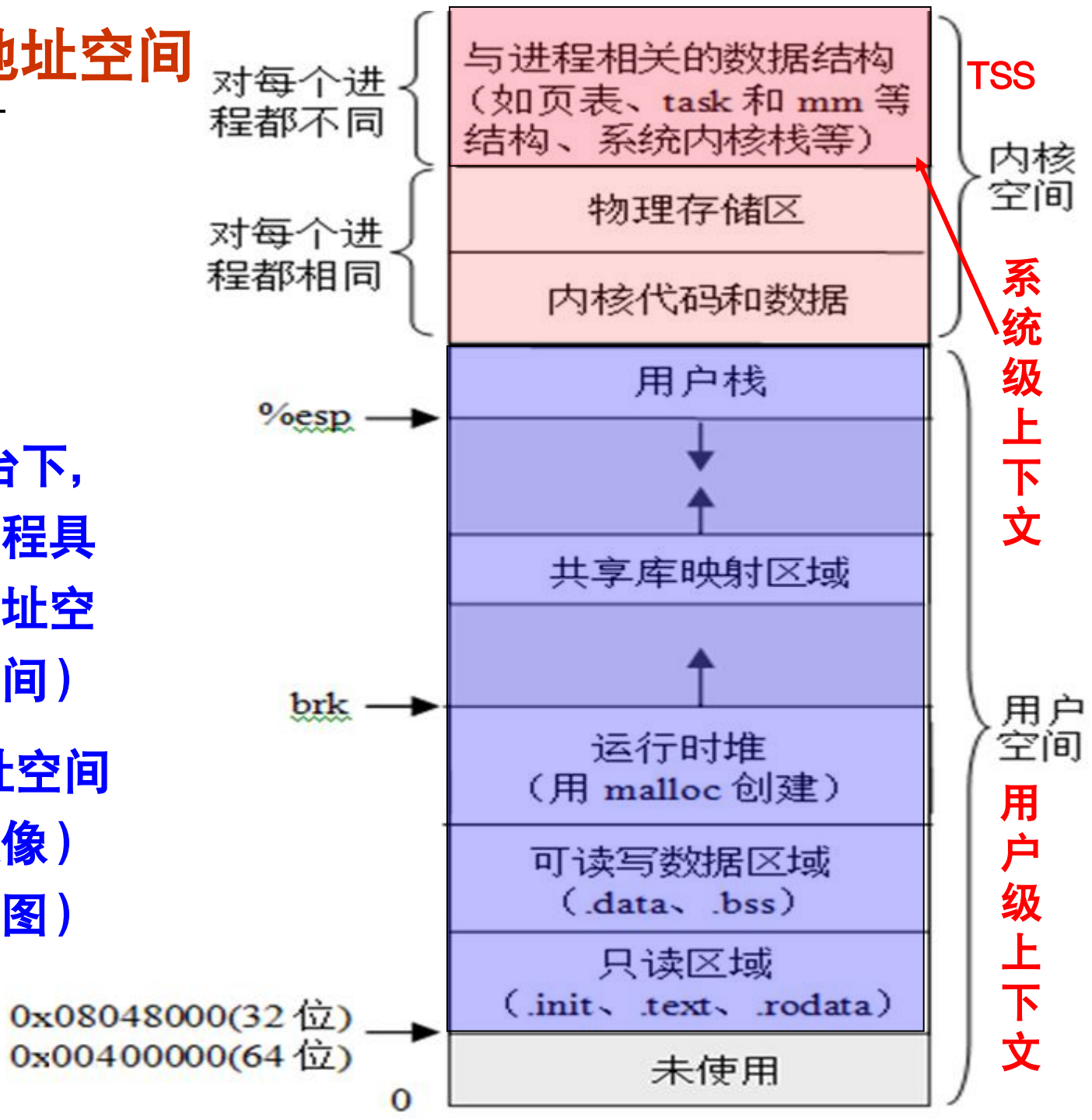
PIPT：索引和标记都使用物理地址。优点是易实现，但每次访问前都需先由MMU进行地址转换，TLB缺失时还需等待页表项装入TLB。通常L2 和L3 cache都采用PIPT

VIVT：索引和标记都使用虚拟地址。优点是查找速度快，无须地址转换即可访问，只有cache缺失时才需地址转换，但需解决别名问题、同名问题和页表项更新等问题。

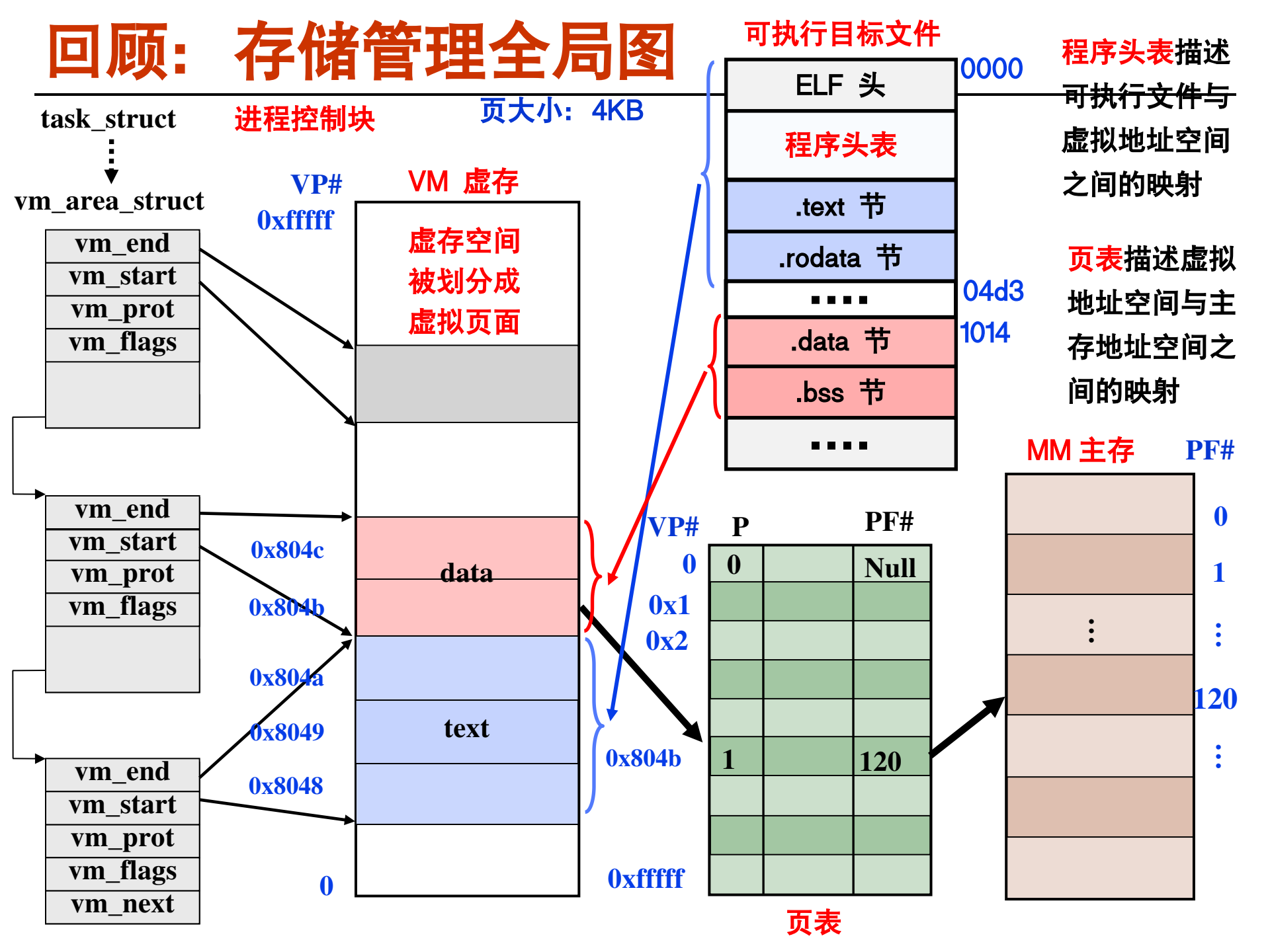
Linux进程的地址空间

任务状态段(TSS):
任务(进程)切换时的现场信息

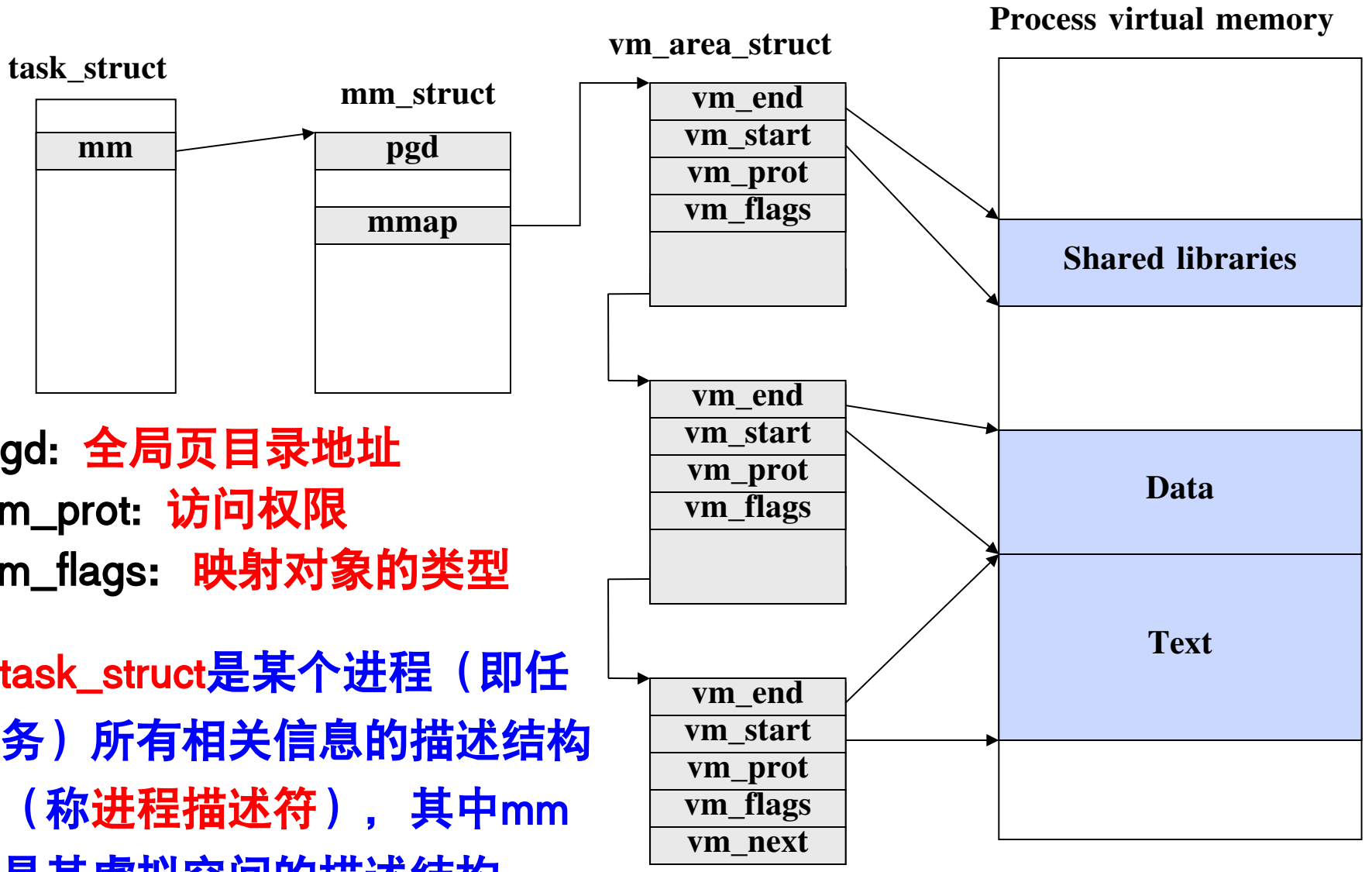
- IA-32/Linux平台下，每个（用户）进程具有独立的私有地址空间（虚拟地址空间）
- 每个进程的地址空间划分（即存储映像）布局相同（如右图）



回顾：存储管理全局图


可执行目标文件
程序头表描述可执行文件与虚拟地址空间之间的映射
页表描述虚拟地址空间与主存地址空间之间的映射
task_struct
进程控制块
页大小：4KB
vm_area_struct
vm_end
vm_start
vm_prot
vm_flags
vm_next
VP#
VM 虚存
0xfffff
虚存空间被划分成虚拟页面
0x804c
0x804b
0x804a
0x8049
0x8048
0
data
text
ELF 头
程序头表
.text 节
.rodata 节
.data 节
.bss 节
0000
04d3
1014
VP#
P
PF#
0
Null
1
120
0x804b
0xfffff
页表
MM 主存
PF#
0
1
120

回顾：Linux将虚存空间组织成“区域”的集合



pgd: 全局页目录地址

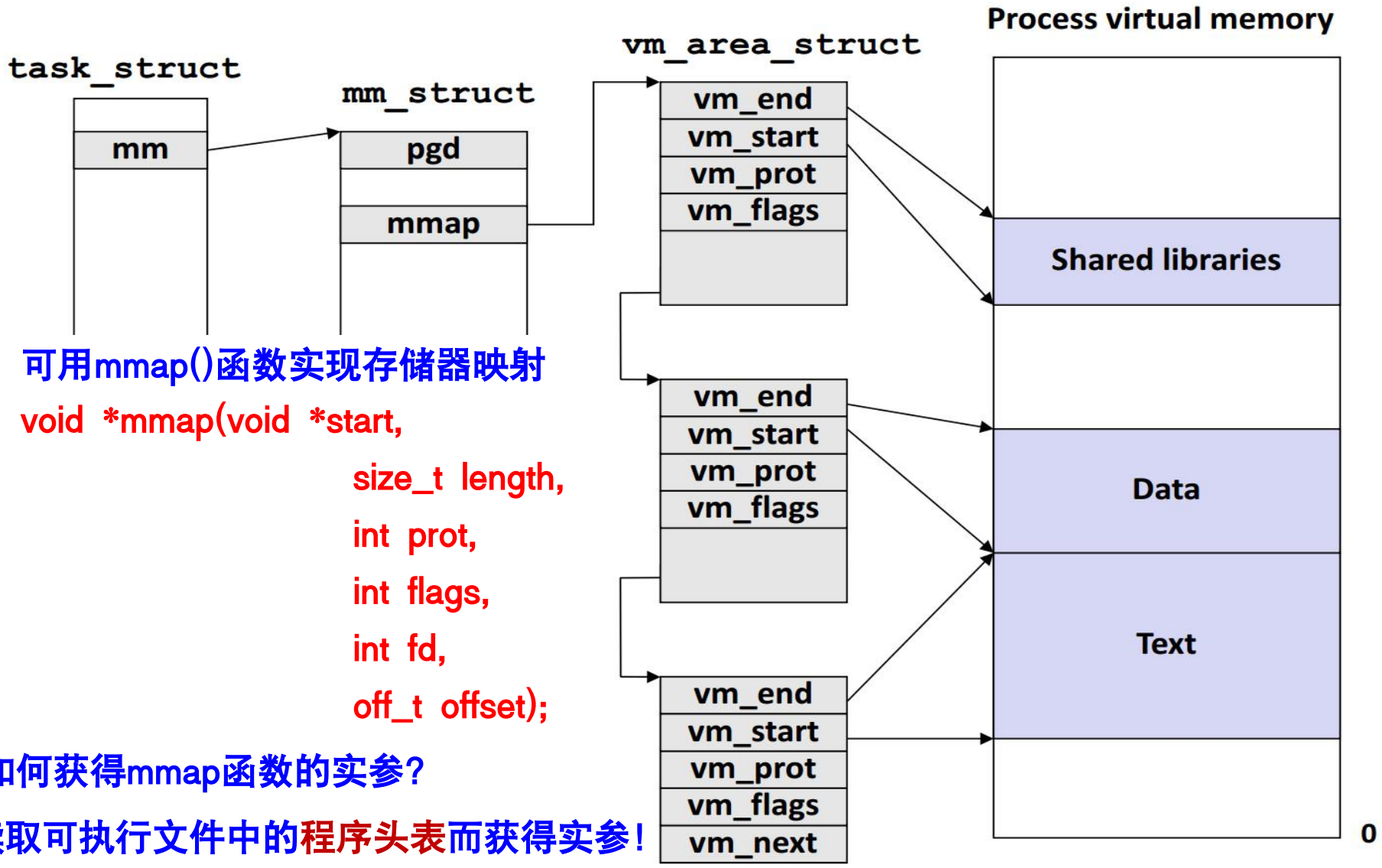
vm_prot: 访问权限

vm_flags: 映射对象的类型

task_struct是某个进程（即任务）所有相关信息的描述结构（称进程描述符），其中mm是其虚拟空间的描述结构

Linux中进程的存储器映射

存储器映射（memory mapping）是指将进程虚拟地址空间中的一个区域与硬盘上的一个对象建立关联（生成页表项），并初始化一个vm_area_struct结构信息



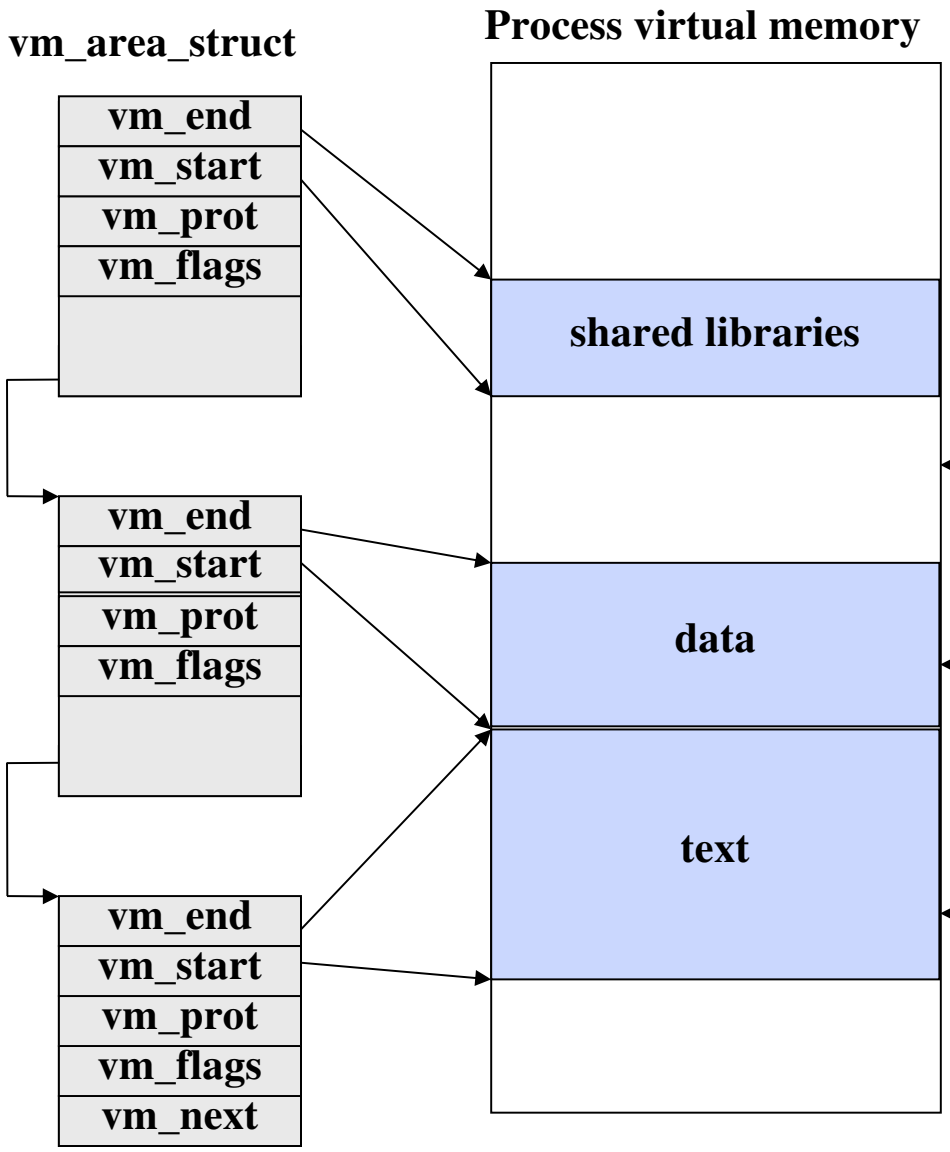
可用mmap()函数实现存储器映射

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

如何获得mmap函数的实参?

读取可执行文件中的程序头表而获得实参!

Linux 中的存储保护机制



Linux页故障类型

页故障
（Segmentation fault）
访问一个“空洞”页面

正常的缺页
页面未装入主存

存储保护错
（Protection exception）
如对只读代码段写（在Linux中报告为Segmentation fault）

Linux中进程的存储器映射

可用mmap()函数实现存储器映射

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

功能：将指定文件fd中偏移量offset开始的长度为length个字节的一块信息映射到虚拟空间中起始地址为start、长度为length个字节的一块区域，得到vm_area_struct结构的信息，并生成相应页表项，建立文件地址和区域之间的映射关系。

prot指定该区域内页面的访问权限位，对应vm_area_struct结构中的vm_prot字段

PROT_EXE：页面内容由指令组成

PROT_READ：区域内页面可读

PROT_WRITE：区域内页面可写

PROT_NONE：区域内页面不能被访问

flags指定所映射的对象的类型，对应vm_area_struct结构中的vm_flags字段

MAP_PRIVATE：私有的写时拷贝对象，对应可执行文件中只读代码区域（.init、.text .rodata）和已初始化数据区域（.data）

MAP_SHARED：共享对象，对应共享库文件中的信息

MAP_ANON：请求0的页，对应内核创建的匿名文件，相应页框用0覆盖并驻留内存

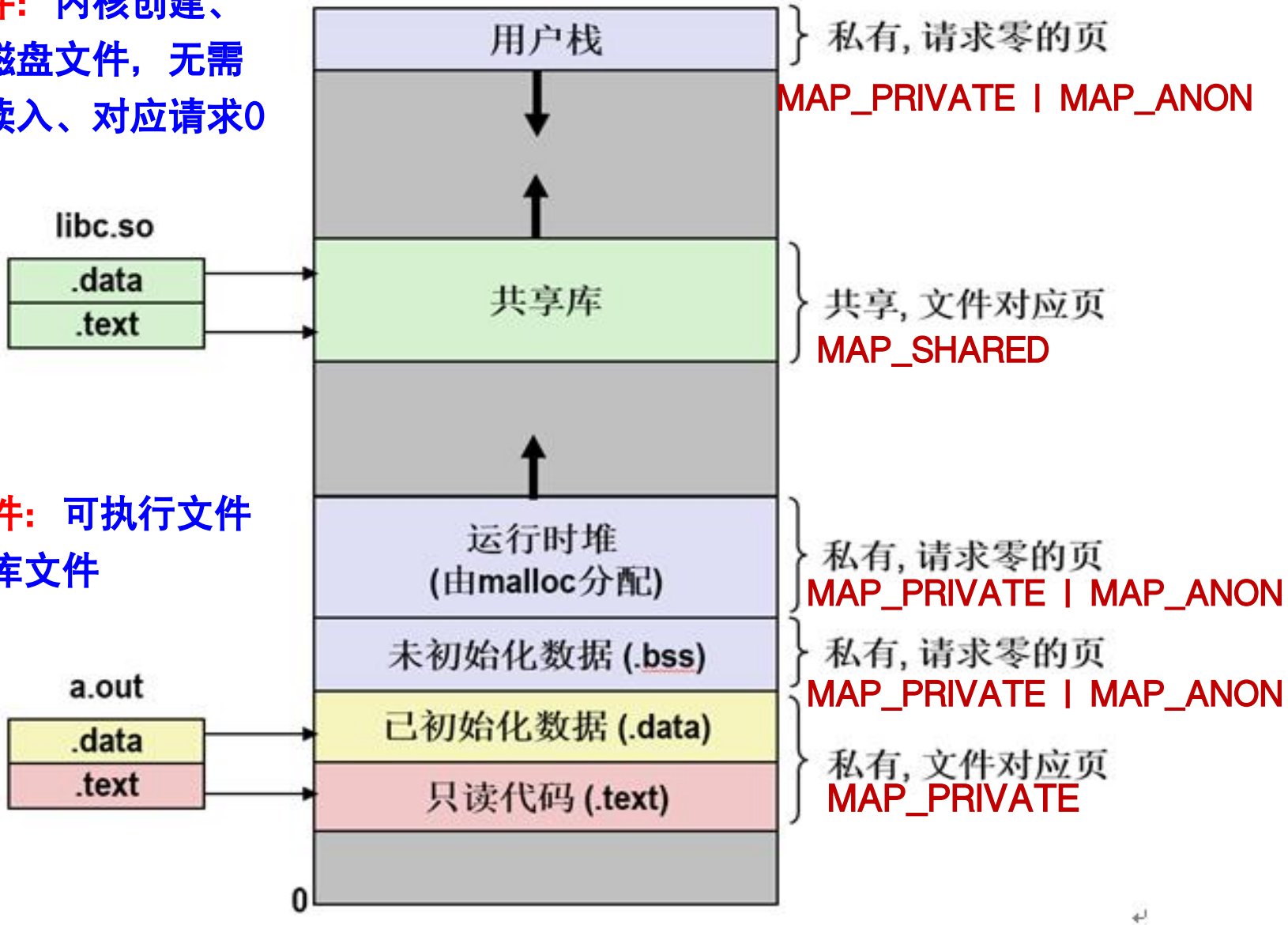
MAP_PRIVATE | MAP_ANON：未初始化数据（.bss）、堆和用户栈等对应区域

虚页第一次被装入内存后，不管是用普通文件还是匿名文件对其进行初始化，以后都是在主存页框和硬盘中交换文件（swap file）间进行调进调出。交换文件由内核管理和维护，称为交换分区（swap area）或交换空间（swap space）。

Linux中虚拟地址空间中的区域

匿名文件：内核创建、无实际磁盘文件，无需从磁盘读入、对应请求0的页面

普通文件：可执行文件和共享库文件



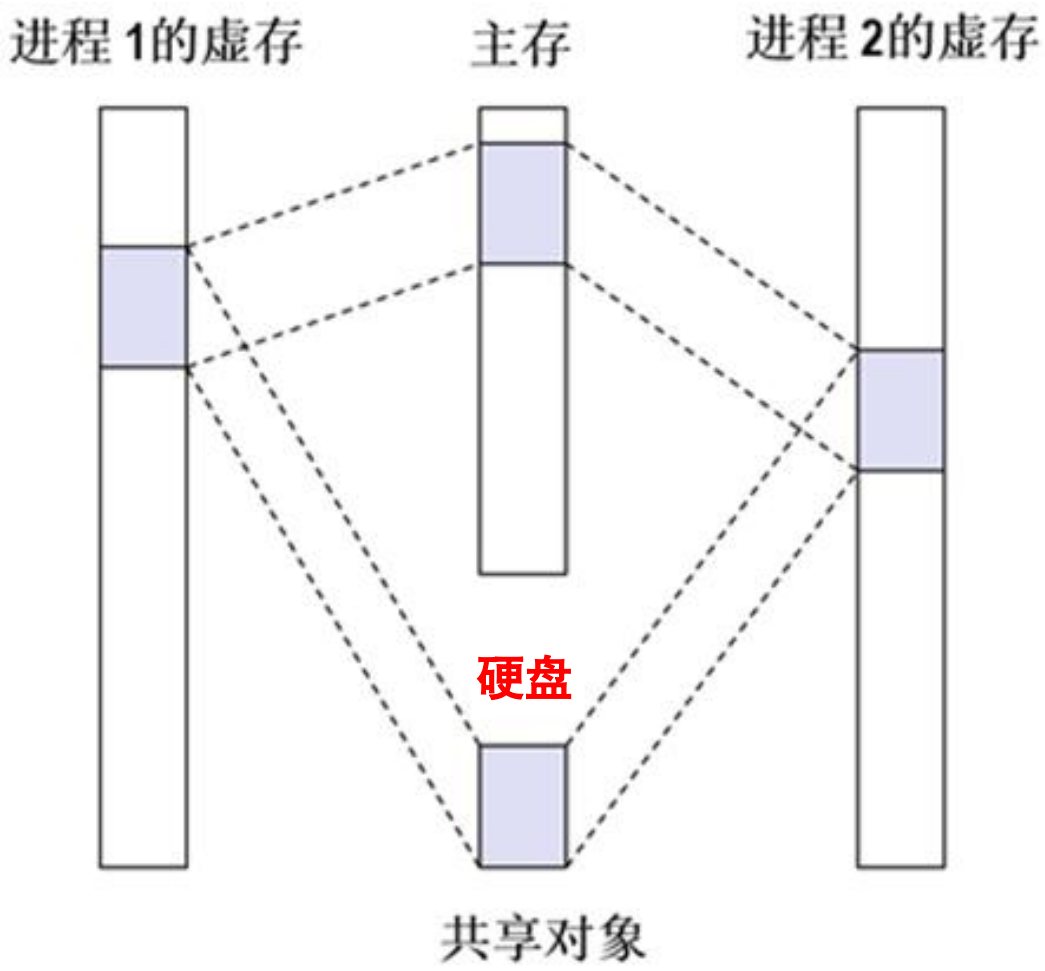
共享库文件中的共享对象

多个进程调用共享库文件中的代码，但共享库代码在内存和硬盘都只需要一个副本

进程1运行过程中，内核为共享对象分配若干页框

进程2运行过程中，内核只要将进程2对应区域内页表项中的页框号直接填上即可

一个进程对共享区域进行的写操作结果，对于所有共享同一个共享对象的进程都是可见的，而且结果也会反映在硬盘上对应的共享对象中



所分配的页框在主存不一定连续，为简化示意图，这里图中所示页框是连续的

私有的写时拷贝对象

同一个可执行文件对应不同进程时，只读代码区一样，可读可写数据区开始也一样，但属于私有对象

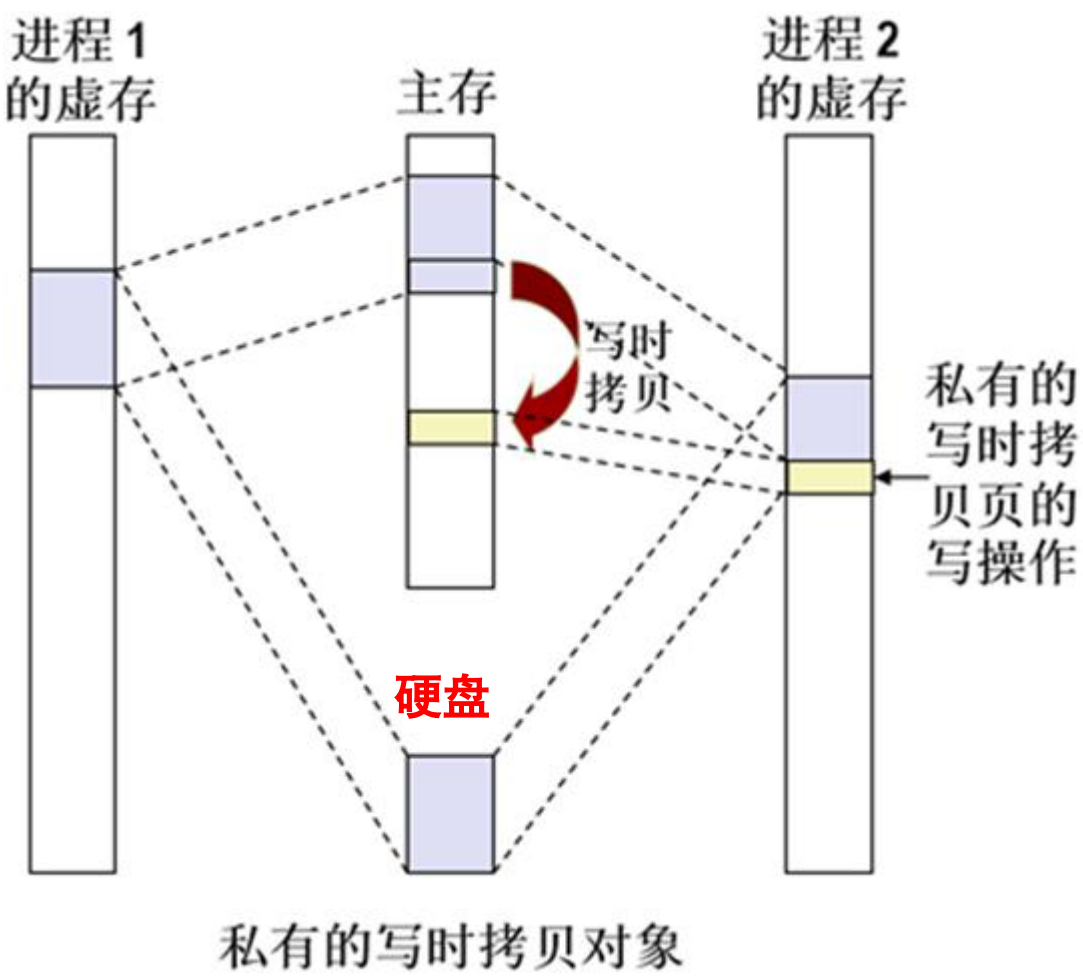
为节省主存，多采用写时拷贝技术

进程1运行过程中，内核为对象分配若干页框，并标记为只读

进程2运行过程中，内核只要将进程2对应区域内页表项中的页框号直接填上，并标记为只读

若两个进程都只是读或执行，则在内存只有一个副本，节省主存；

若进程2进行写操作，则发生访问违例，此时，内核判断异常原因是进程试图写私有的写时拷贝页，就会分配一个新页框，把内容拷贝到新页框，并修改进程2的页表项



私有的写时拷贝对象

所分配的页框在主存不一定连续，为简化示意图，这里图中所示页框是连续的

LA64+Linux中虚拟地址空间布局举例

第1列为VA范围；第2列为访问权限（r-读/w-写/x-执行/P-私有）；第3列为映射对象文件中起始位置；第4列为映射对象文件主、次设备号；第5列为映射对象文件节点号；最后一列是映射对象文件路径。每一行表示1个虚存区域（VMA），起始地址按16KB对齐

前3个VMA分别映射到可执行文件只读代码段中的代码部分（访问权限为r-xp）、只读代码段的只读数据部分（访问权限为r--p）和可读写数据段（访问权限为rw-p）

120000000-120004000	r-xp	00000000	fe:05	131896	/home/loongson/bao/7/pid
120004000-120008000	r--p	00000000	fe:05	131896	/home/loongson/bao/7/pid
120008000-12000c000	rw-p	00004000	fe:05	131896	/home/loongson/bao/7/pid
12053c000-120560000	rw-p	00000000	00:00	0	[heap]
fff5818000-fff5968000	r-xp	00000000	fe:04	1320855	/usr/lib64/libc-2.28.so
fff5968000-fff597c000	r--p	0014c000	fe:04	1320855	/usr/lib64/libc-2.28.so
fff597c000-fff5980000	rw-p	00160000	fe:04	1320855	/usr/lib64/libc-2.28.so
fff5980000-fff5984000	rw-p	00000000	00:00	0	
fff5994000-fff59b4000	r-xp	00000000	fe:04	1320714	/usr/lib64/ld-2.28.so
fff59b4000-fff59b8000	r--p	0001c000	fe:04	1320714	/usr/lib64/ld-2.28.so
fff59b8000-fff59bc000	rw-p	00020000	fe:04	1320714	/usr/lib64/ld-2.28.so
fffbcc4000-fffbce8000	rw-p	00000000	00:00	0	[stack]
fffbff4000-fffbff8000	r-xp	00000000	00:00	0	
fffe104000-fffe108000	r--p	00000000	00:00	0	[vvar]
fffe108000-fffe10c000	r-xp	00000000	00:00	0	[vdso]