# 第五章 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
- 实例：Intel Core i7+Linux存储系统

# 虚拟存储器

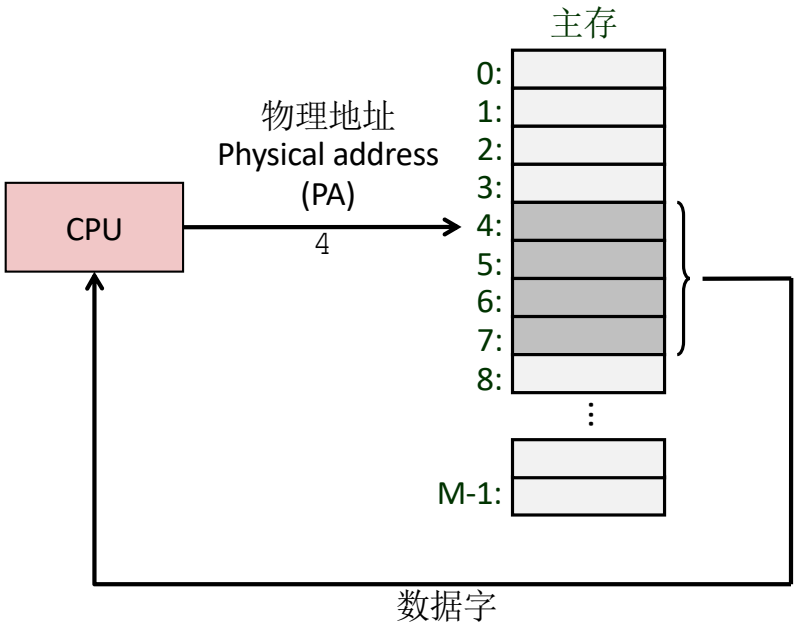
- 学习目标
  - 了解虚拟存储管理的基本概念
  - 理解进程的虚拟地址空间的基本概念
  - 理解页式虚拟存储管理的实现原理
  - 理解访存操作完整过程以及所涉及到的部件之间的关联
    - 地址转换（查TLB、查页表）、访问Cache、访问主存、读写磁盘
  - 理解访存过程中硬件和操作系统之间的协调关系
    - 页表的构建和修改、访问违例、存储保护机制

# 虚拟存储器

- 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
- 第三讲：具有TLB和cache的存储系统
  - 快表、层次化存储系统结构
  - CPU的访存过程
- 第四讲：存储系统实例
  - Core i7+Linux存储系统

# 寻址形式

- 物理寻址
  - 主存被组织成M个连续的字节大小的单元组成的数组
  - 每字节有唯一的物理地址：0, 1, …, M-1
  - CPU使用物理地址访问主存
  - 使用机器：早期PC、数字信号处理器、嵌入式微控制器、Cray超级计算机



# 早期分页方式的实现

- 1961年，英国曼切斯特的研究人员提出一种自动执行内存覆盖（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考虑主存容量的大小
- 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

当时的一种典型计算机

- 指令中的主存地址为16位
- 主存容量只有4K字
- 寻址范围0, 1, …, 65536（16位地址空间）比执行程序的空间0, …, 4095（主存容量）大得多，程序该如何执行呢？

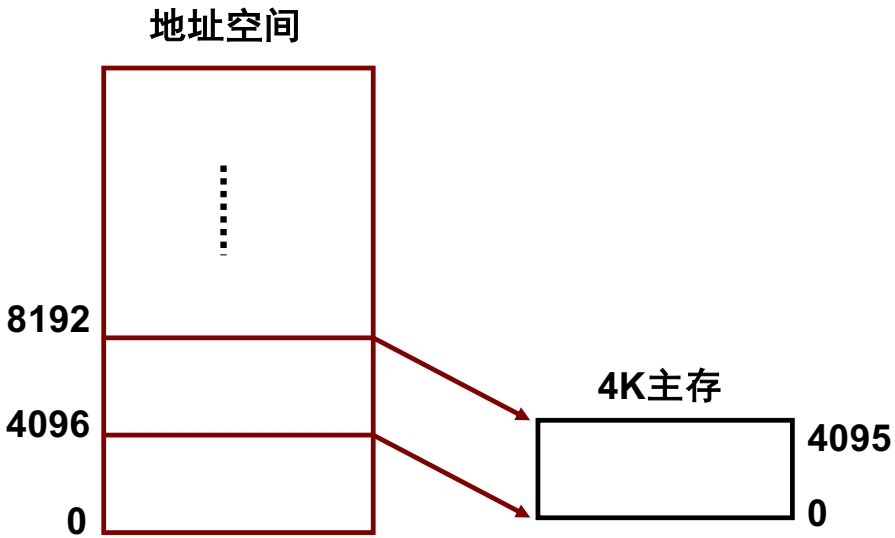


# 早期分页方式的实现

- 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？
- 超出主存大小的完整程序在磁盘上，不断换入换出主存
- 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，即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

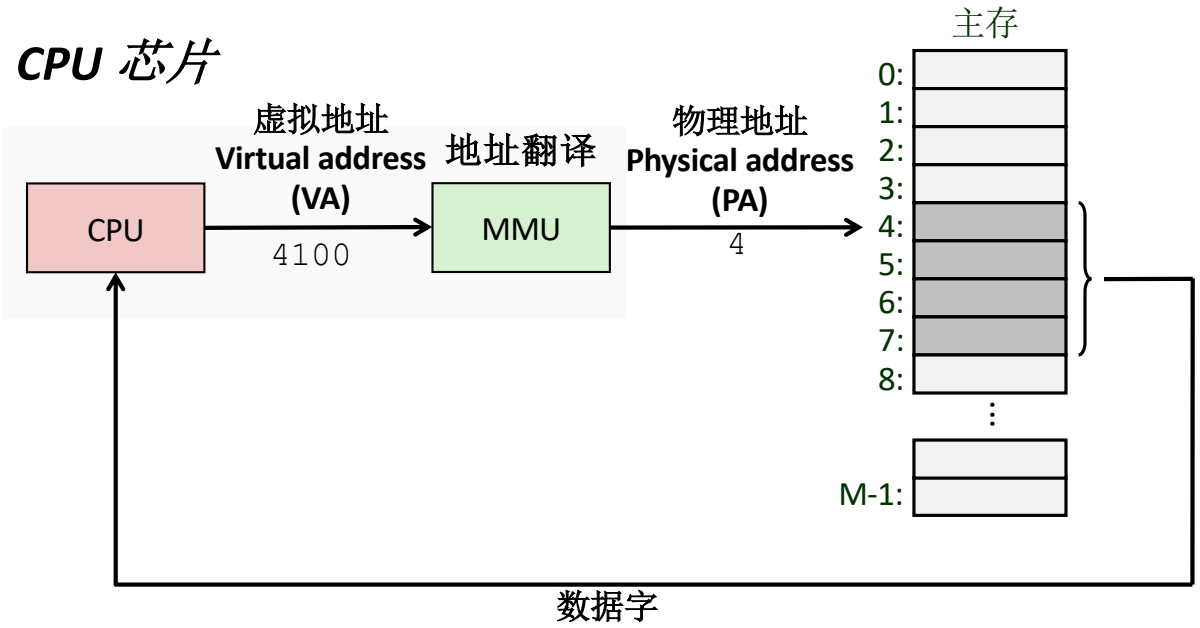
后来把区间称为页(page)
主存中存放页的区域称为页帧(page frame)
早期主存只有一个页帧

发明在SRAM及高速缓存Cache出现之前



# 寻址形式

- 虚拟寻址
  - CPU通过一个虚拟地址访问主存
  - 虚拟地址经过地址翻译转化成适当的物理地址
  - 地址翻译由CPU中的专用硬件（内存管理单元，MMU）和操作系统合作完成
  - 使用机器：多数现代机器（服务器、台式机、笔记本、智能手机）

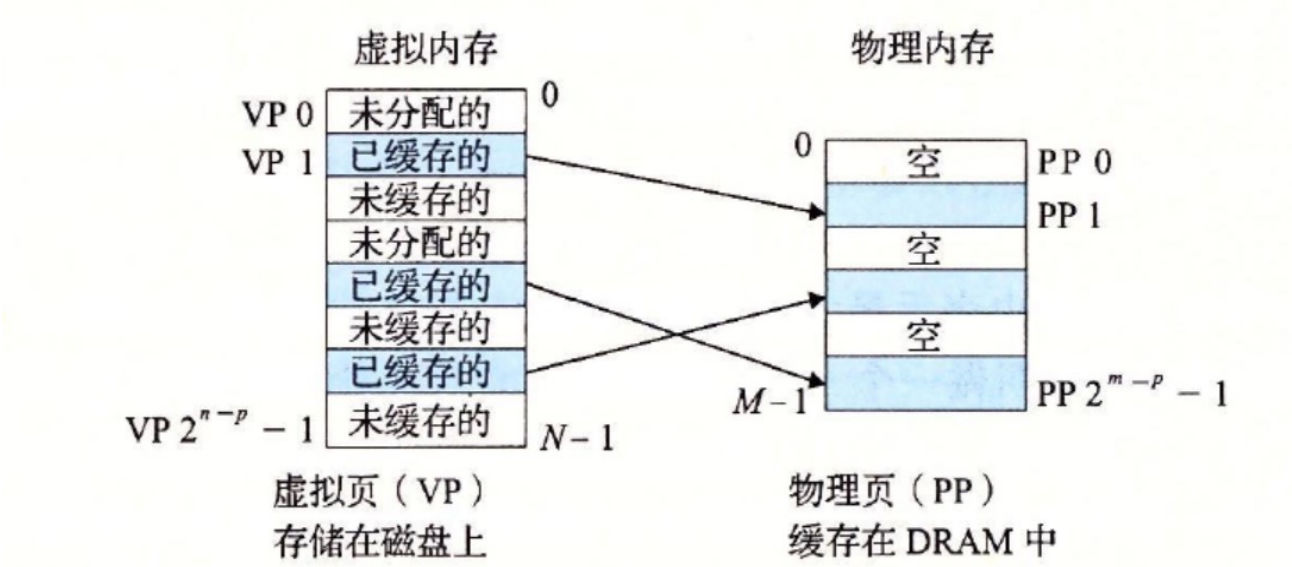


# 逻辑地址与物理地址

- 逻辑地址
  - 程序中指令所用地址（进程所在地址空间）
  - 也称为虚拟地址
  - 在一个使用虚拟内存的系统中，CPU从一个有$N=2^n$个地址的地址空间中生成虚拟地址，这个地址空间称为n位逻辑地址空间：{0, 1, 2, ···, N-1}
  - 现代系统通常支持32位或64位虚拟地址空间
- 物理地址
  - 存放指令或数据的实际主存地址
  - 也称为实地址、主存地址
  - 对应物理地址空间：{0, 1, 2, ···, M-1}
- 主存中每字节都有一个来自虚拟地址空间的虚拟地址和一个来自物理地址空间的物理地址

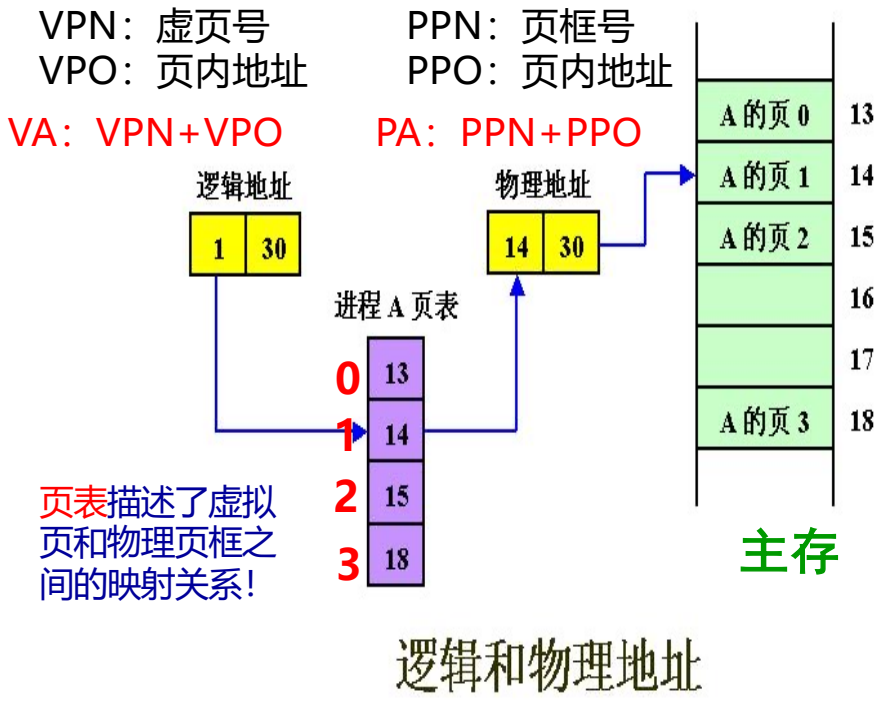
# 分页（paging）

- 虚拟内存系统将虚拟内存（VM）分割成虚拟页（VP，虚页/逻辑页）的大小固定的块来处理这个问题。每个虚页的大小为P=$2^p$字节。
- 物理内存被分割成物理页（PP），大小也是P字节，物理页也被称为页帧（page frame，页框/实页）
- 概念上，VM被组织为一个由存放在磁盘上的N个连续的字节大小的单元组成的数组，每字节有唯一的虚拟地址，作为到数组的索引。磁盘上数组的部分内容被缓存在主存中。



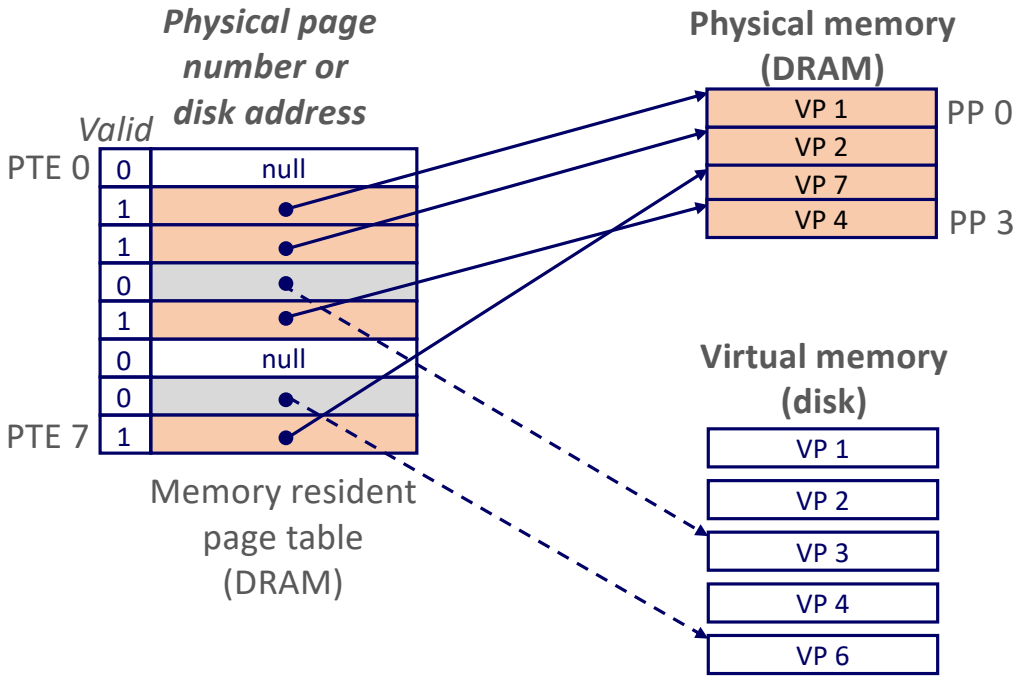
# 分页（paging）

- 按需调页（demand paging）
  - 局部性：对于一个进程，仅将当前活跃的页面调入主存



# 分页（paging）

- 页表
  - 虚拟页映射到物理页
    - 判定一个虚拟页是否缓存
    - 是1，确定虚拟页存放在哪个物理页
    - 否0，虚拟页在磁盘哪个位置，虚拟页装入物理页（替换）
  - 每次地址翻译硬件将一个虚拟地址转换为物理地址时，都会读取页表
  - 操作系统负责维护页表的内容
  - 是一个页表条目（Page Table Entry，PTE）的数组
    - 虚拟地址空间每个页在页表中固定偏移处有一个PTE
      - 未分配、已缓存、未缓存
  - 页表常驻在DRAM中
    - 任意虚拟页可以装入任意物理页（全相联）



# 页表

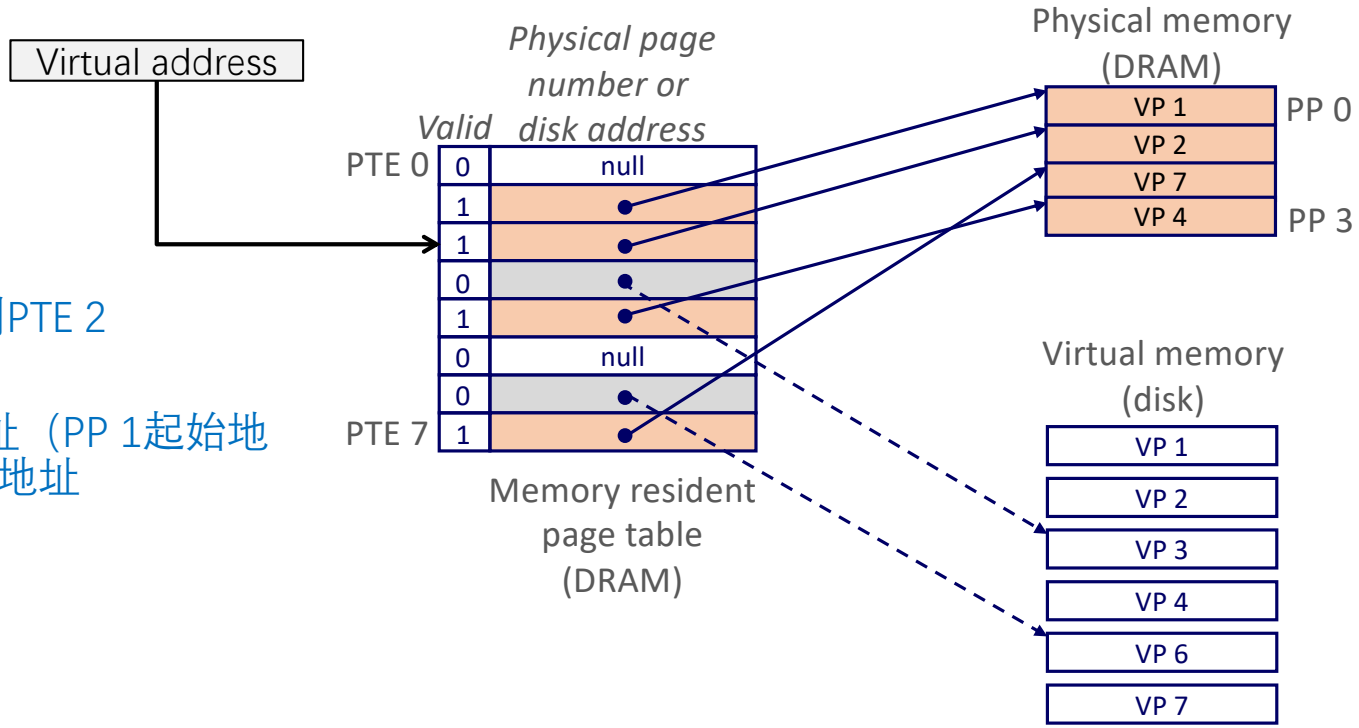
练习题9.2

确定下列虚拟地址大小(n)和页大小(P)的组合所需要的PTE数量

| n | **P=$2^p$** | **PTE数量** |
|---|---|---|
| 16 | 4K | 16 |
| 16 | 8K | 8 |
| 32 | 4K | $2^{20}$ |
| 32 | 8K | $2^{19}$ |

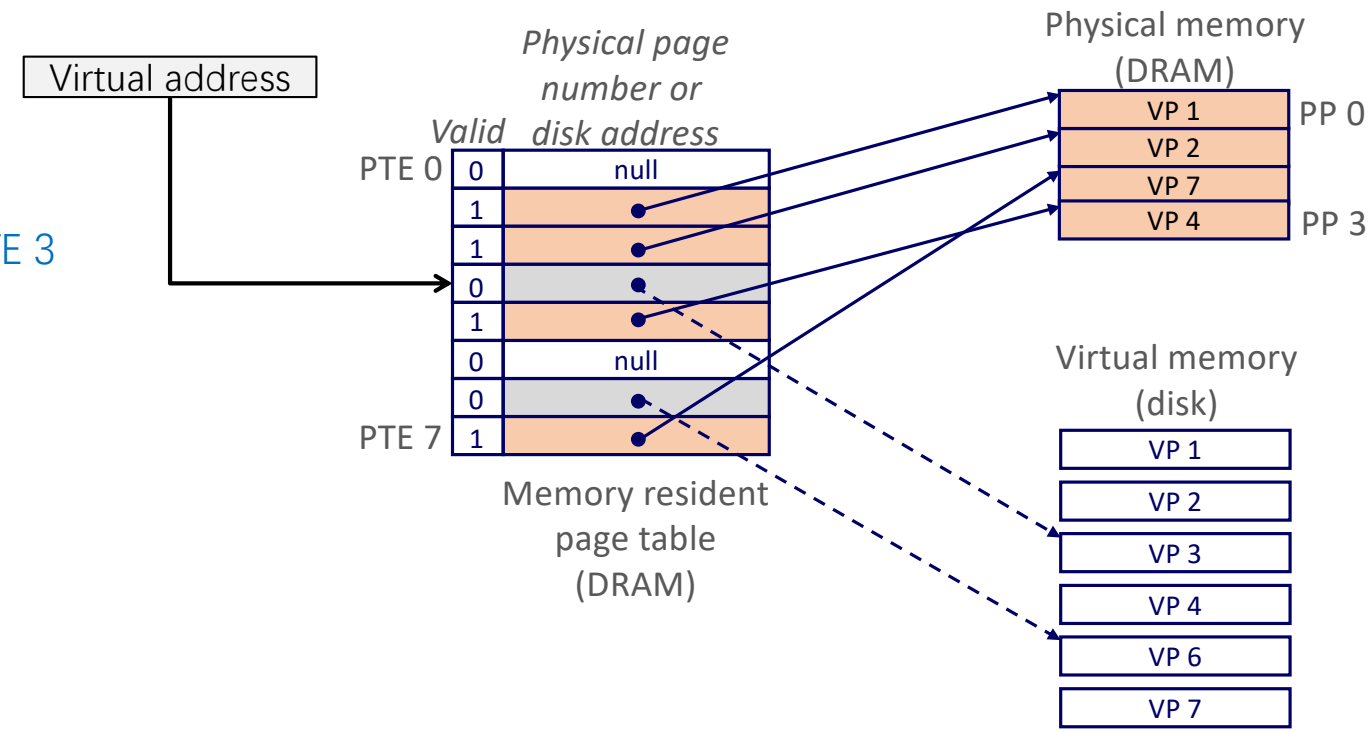
# 按需调页

- 页命中
  - 引用VP 2中某个字
  - 根据VA地址翻译定位到PTE 2
  - 有效，即VP 2已缓存
  - 根据PTE中物理内存地址（PP 1起始地址）构造访问字的物理地址



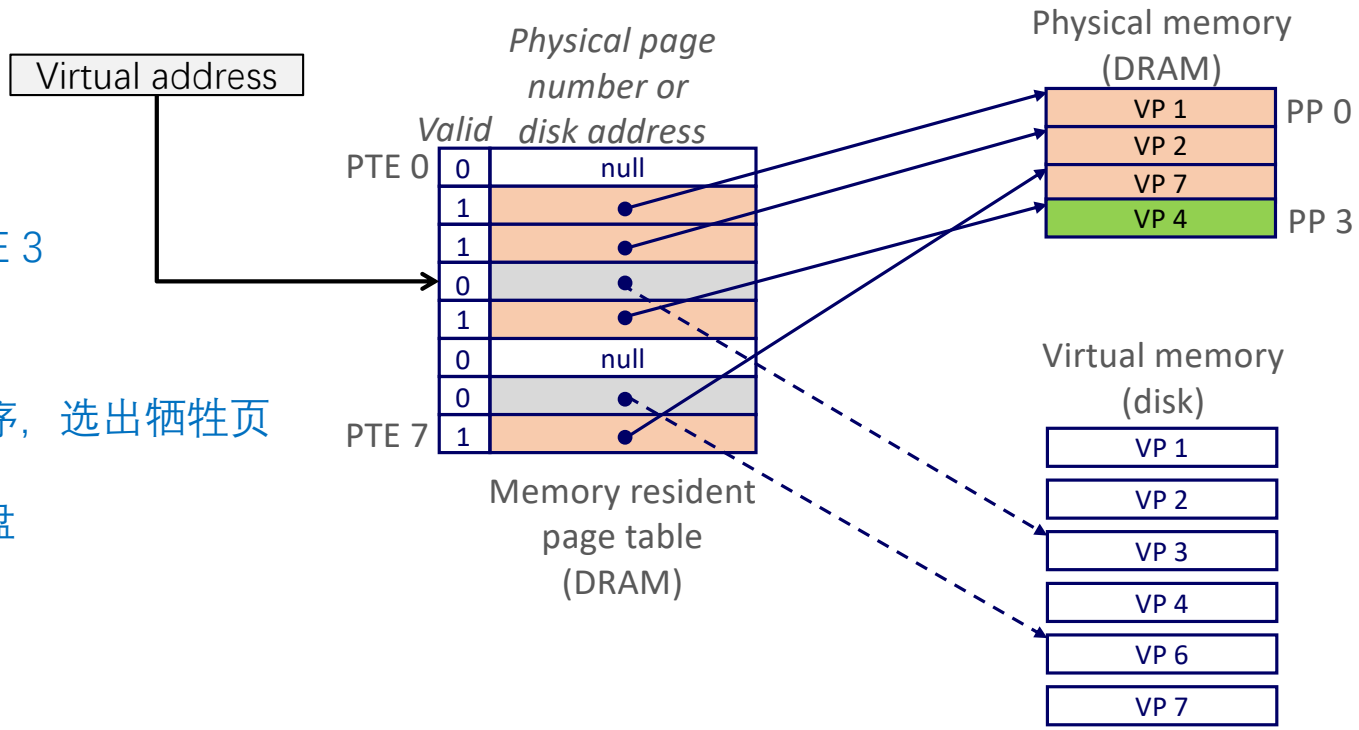
# 按需调页

- 缺页
  - 引用VP 3中某个字
  - 根据VA地址翻译定位到PTE 3
  - 无效，即VP 3未缓存



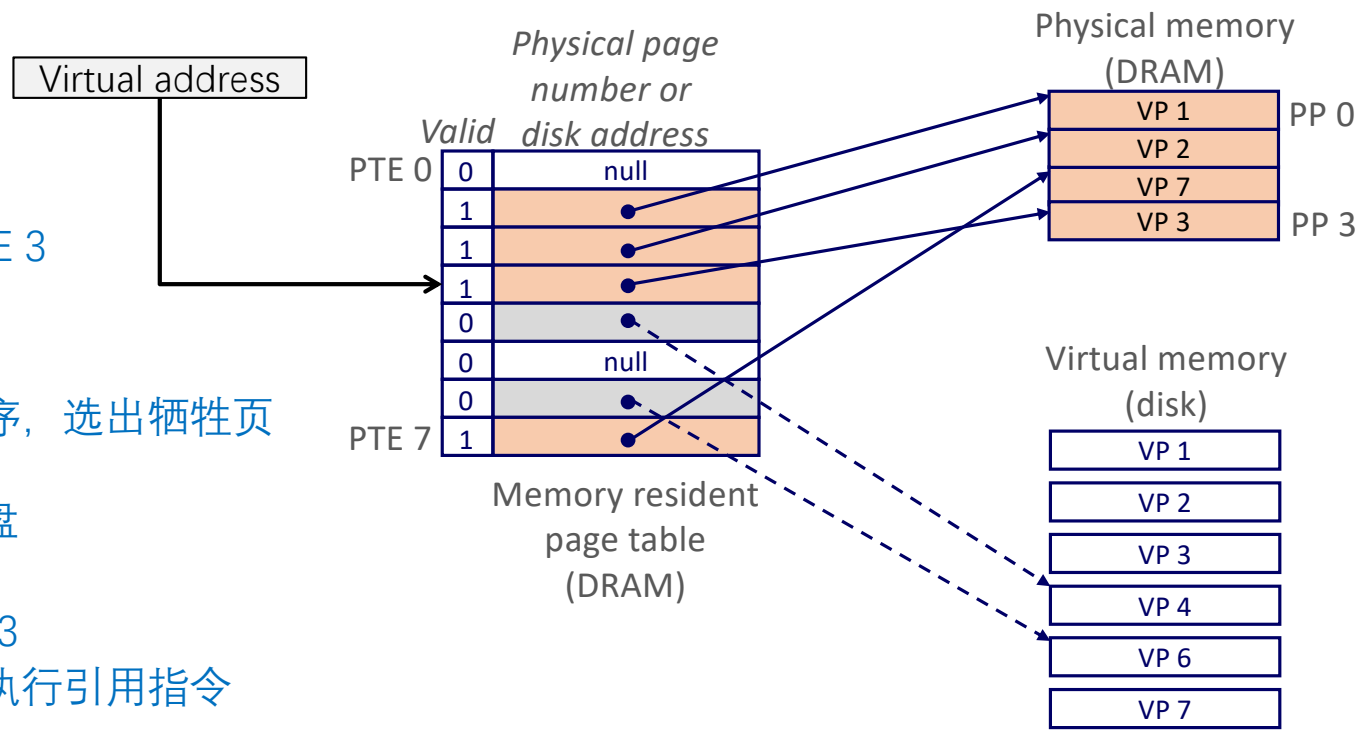
# 按需调页

- 缺页
  - 引用VP 3中某个字
  - 根据VA地址翻译定位到PTE 3
  - 无效，即VP 3未缓存
  - 触发缺页异常
  - 内核启动缺页异常处理程序，选出牺牲页（PP 3中的VP 4）
  - 若VP 4有修改，复制回磁盘



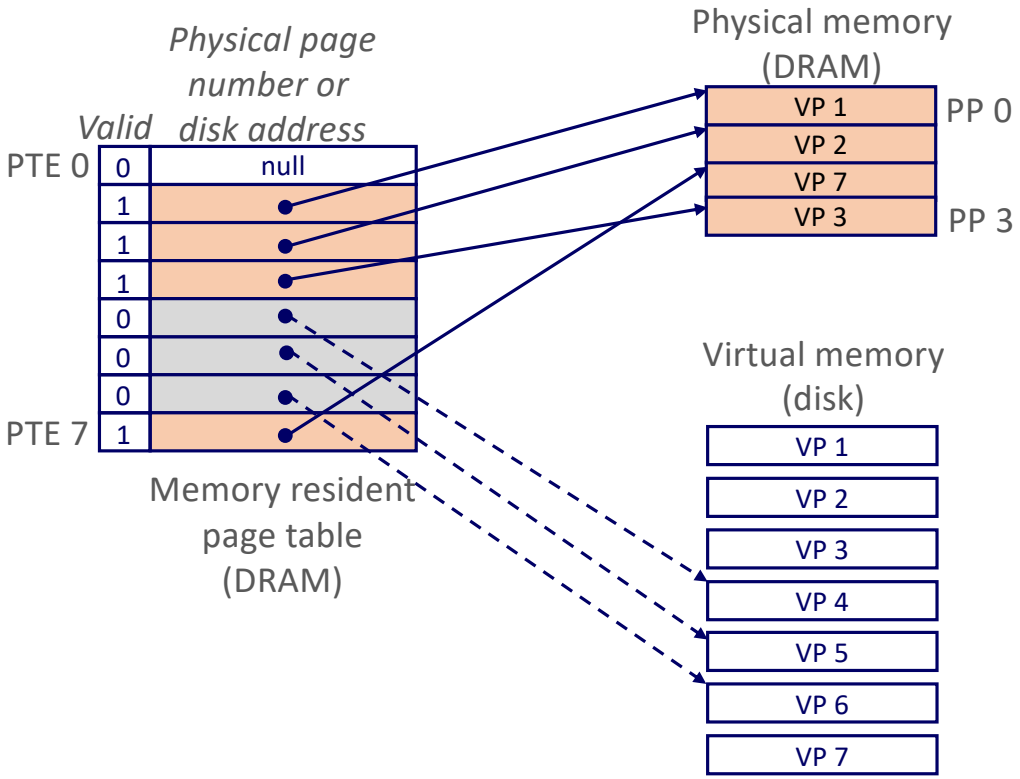
# 按需调页

- 缺页
  - 引用VP 3中某个字
  - 根据VA地址翻译定位到PTE 3
  - 无效，即VP 3未缓存
  - 触发缺页异常
  - 内核启动缺页异常处理程序，选出牺牲页（PP 3中的VP 4）
  - 若VP 4有修改，复制回磁盘
  - 修改VP 4的PTE 4
  - VP 3复制到PP3，修改PTE 3
  - 异常处理程序返回，重新执行引用指令



# 按需调页

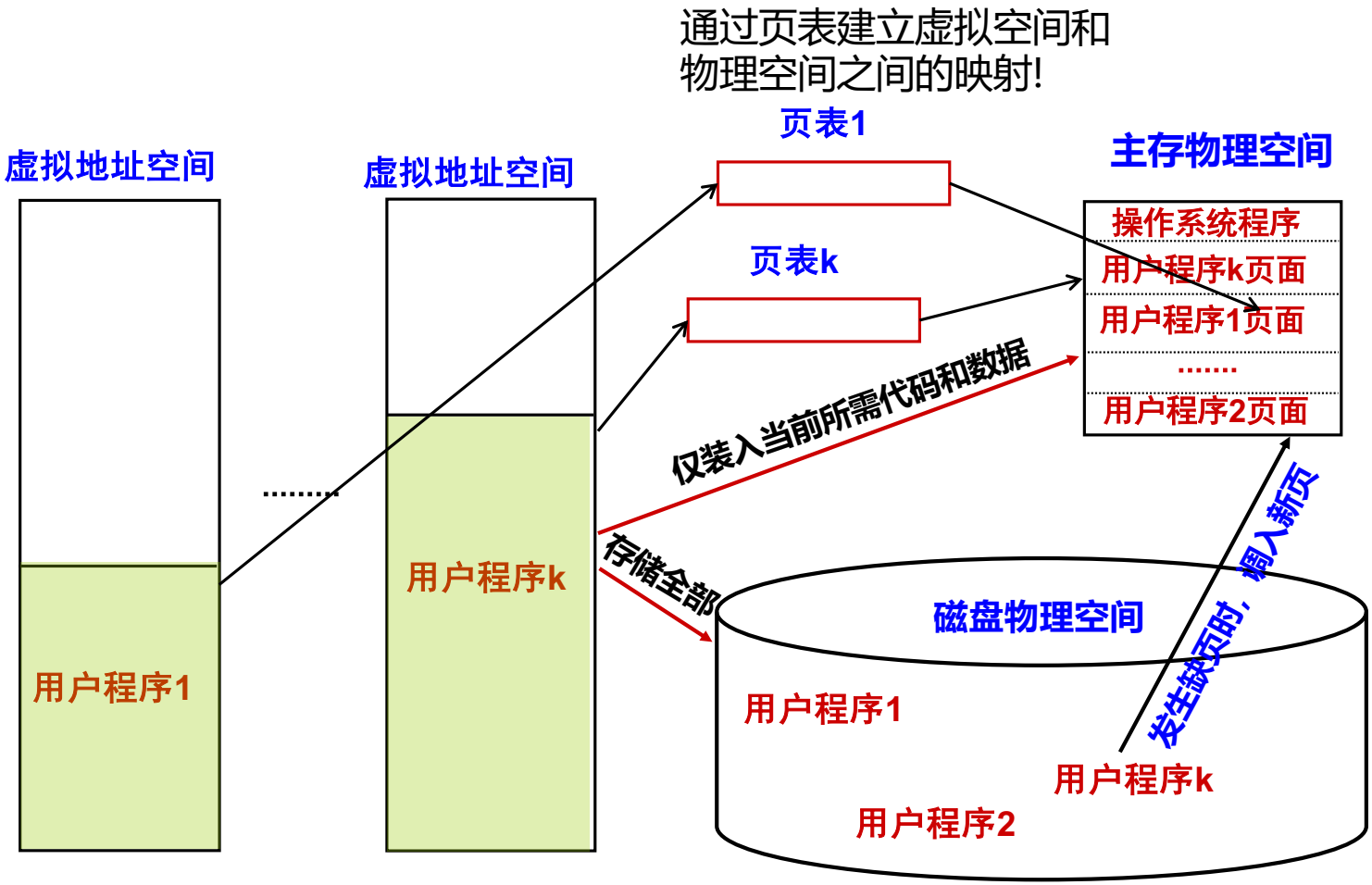
- 页分配
  - 操作系统分配一个新的VP 5（malloc）
  - 更新PTE 5指向VP 5



# 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调度
- 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。

# 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!
虚拟地址空间
虚拟地址空间
页表1
页表k
主存物理空间
操作系统程序
用户程序k页面
用户程序1页面
.....
用户程序2页面
仅装入当前所需代码和数据
存储全部
用户程序k
用户程序1
磁盘物理空间
用户程序1
用户程序k
用户程序2
发生缺页时，调入新页

# 虚拟存储管理实现

- 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?

这些问题由硬件和OS协同解决

三种虚拟存储器实现方式

- 分页式：固定页长
- 分段式：段长因程序而异
- 段页式：段内分页

# 主存-磁盘 VS Cache-主存

- 页（2KB-64KB）比 块（B数量级，32B）大得多　　Why?
- 全相联映射 VS 直接相联/组相联　　Why?

- 缺页开销 比 Cache不命中的开销 大得多
  - 缺页时需要访问磁盘（约几百万个时钟周期）
  - Cache不命中时，访问主存仅需几十到几百个时钟周期
  - 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

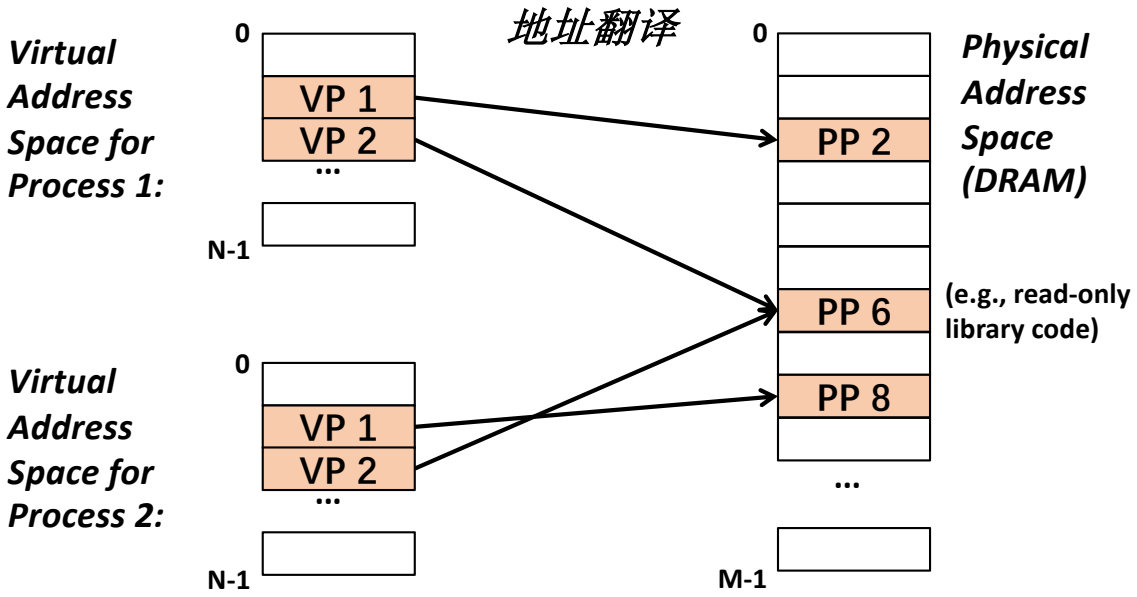
- 地址转换用硬件实现，缺页处理用软件实现　　Why?
  - 加快速度
- 采用写回的写策略　　Why?
  - 避免频繁访问磁盘

# 虚拟存储器

- 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
- 第三讲：具有TLB和cache的存储系统
  - 快表、层次化存储系统结构
  - CPU的访存过程
- 第四讲：存储系统实例
  - Core i7+Linux存储系统

# 虚拟内存作为内存管理工具

- 操作系统为每个进程提供一个独立的页表，即一个独立的虚拟地址空间
  - 多个虚拟页面可以映射到同一个物理页面

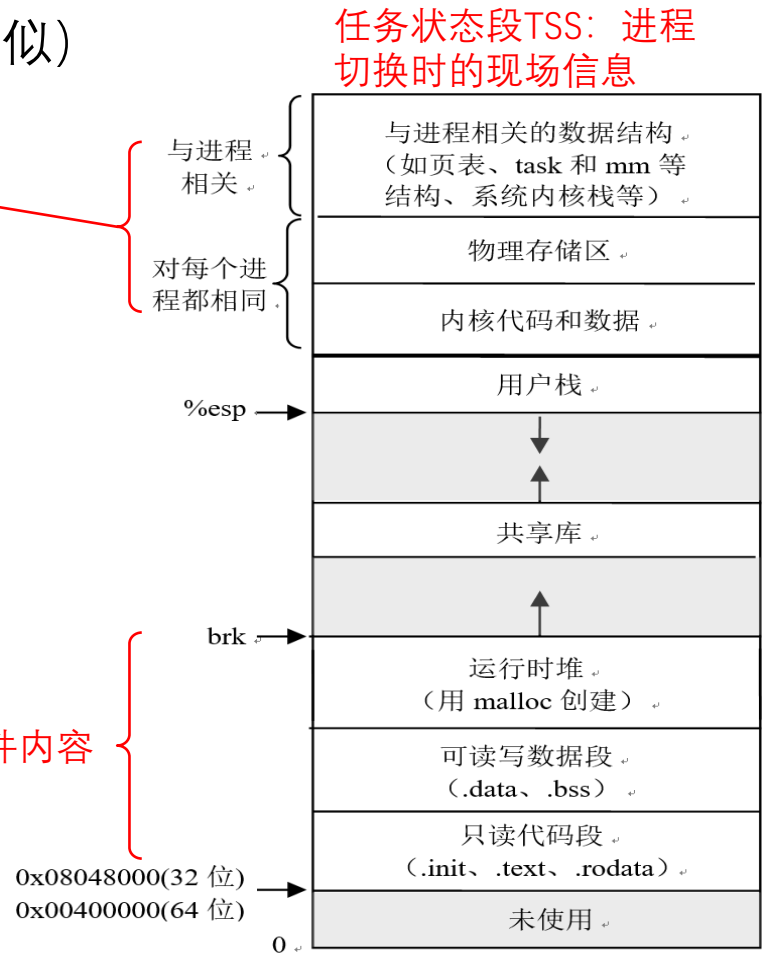


# 虚拟内存作为内存管理工具

- Linux在x86上的虚拟地址空间（其他Unix系统类似）
  - 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）

Linux平台下，每个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）

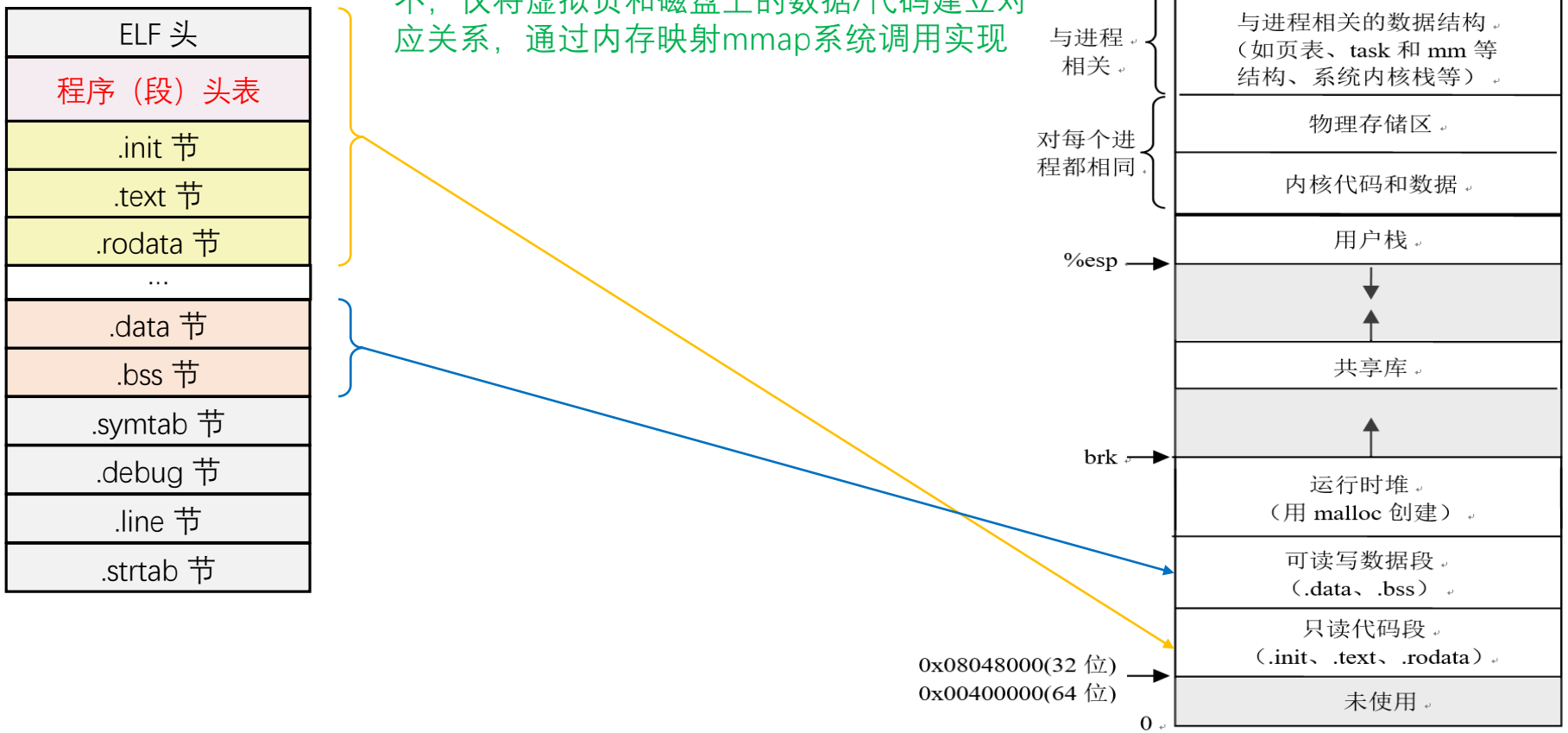


# 回顾：可执行文件的存储器映像

- 程序（段）头表

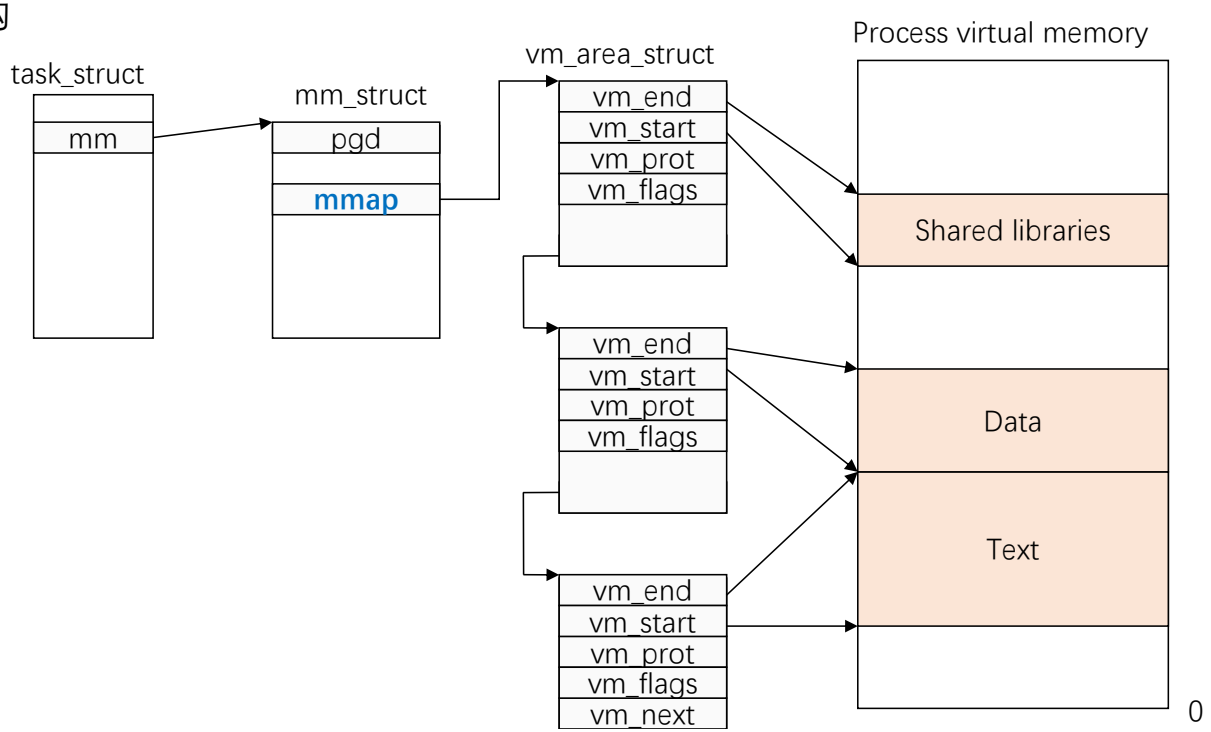
加载可执行文件时是否从磁盘复制到内存?

不，仅将虚拟页和磁盘上的数据/代码建立对应关系，通过内存映射mmap系统调用实现



# 虚拟内存作为内存管理工具

- 内存映射（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
- 进程描述符 task_struct
  - 某个进程所有相关信息的描述结构
  - 其中mm是其虚拟空间的描述结构
    - pgd: 全局页目录地址
    - vm_prot: 访问权限
    - vm_flags: 映射对象的类型



# Linux中进程的存储器映射

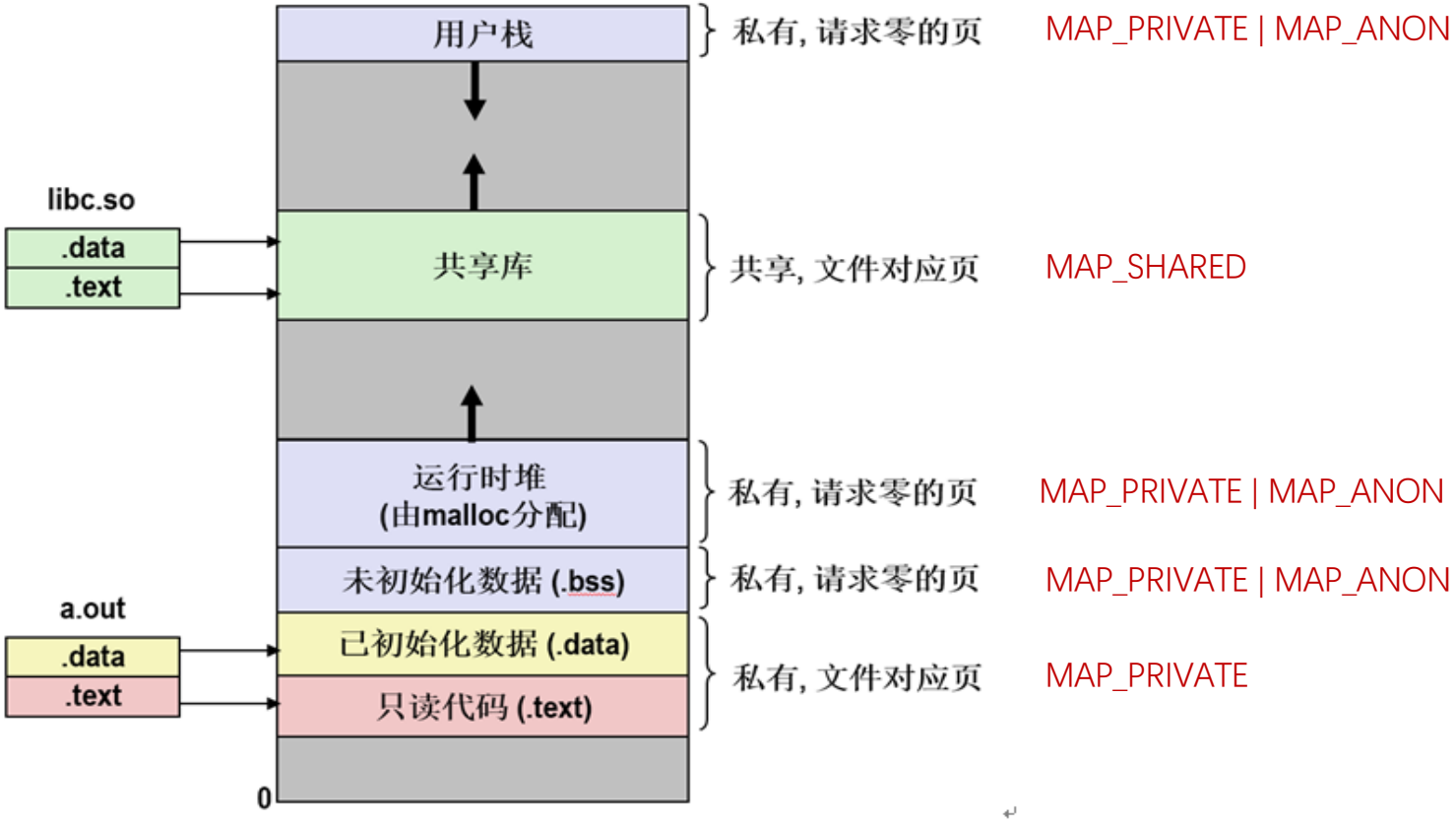
```
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

- 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，以后调进调出都是在主存页框和硬盘交换文件（swap file）间进行。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）。

如何获得mmap的实参？

读取可执行文件的程序头表获得

# Linux中进程的存储器映射



# 回顾：可执行文件中的程序头表

VirtAddr=PhysAddr：使用虚拟内存

FileSiz<MemSiz：.bss（未初始化全局变量和静态变量），不在磁盘中初始化，加载到内存时填充默认值

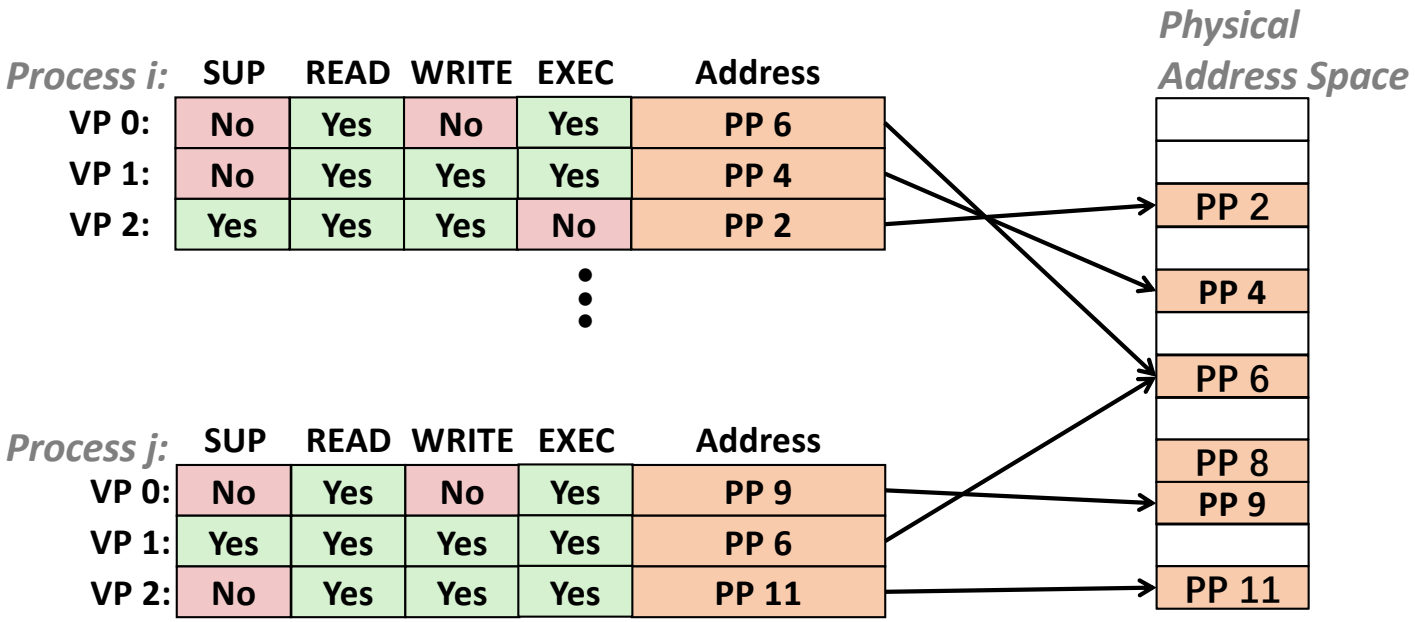
```
Type      Offset             VirtAddr           PhysAddr           FileSiz            MemSiz             Flags    Align
PHDR      0x0000000000000040 0x0000000000400040 0x0000000000400040 0x00000000000002d8 0x00000000000002d8 R        0x8
INTERP    0x0000000000000318 0x0000000000400318 0x0000000000400318 0x000000000000001c 0x000000000000001c R        0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD      0x0000000000000000 0x0000000000400000 0x0000000000400000 0x00000000000004a8 0x00000000000004a8 R        0x1000
LOAD      0x0000000000001000 0x0000000000401000 0x0000000000401000 0x0000000000000159 0x0000000000000159 RE       0x1000
LOAD      0x0000000000002000 0x0000000000402000 0x0000000000402000 0x00000000000000a0 0x00000000000000a0 R        0x1000
LOAD      0x0000000000002e50 0x0000000000403e50 0x0000000000403e50 0x00000000000001e8 0x00000000000001f8 RW       0x1000
```

- PHDR：程序头表本身，告诉加载器，在文件哪个位置（Offset）
- INTERP：加载指定的动态链接器ld-linux-x86-64.so.2
- LOAD(1)：从文件开头开始（Offset=0），映射到虚拟地址VirtAddr（64位ELF基址），只读（R），按4KB（Align）对齐，ELF文件头、程序头表
- LOAD(2)：读+执行（RE），代码节
- LOAD(3)：只读数据节
- LOAD(4)：读写（RW），数据节

# 虚拟内存作为内存保护工具

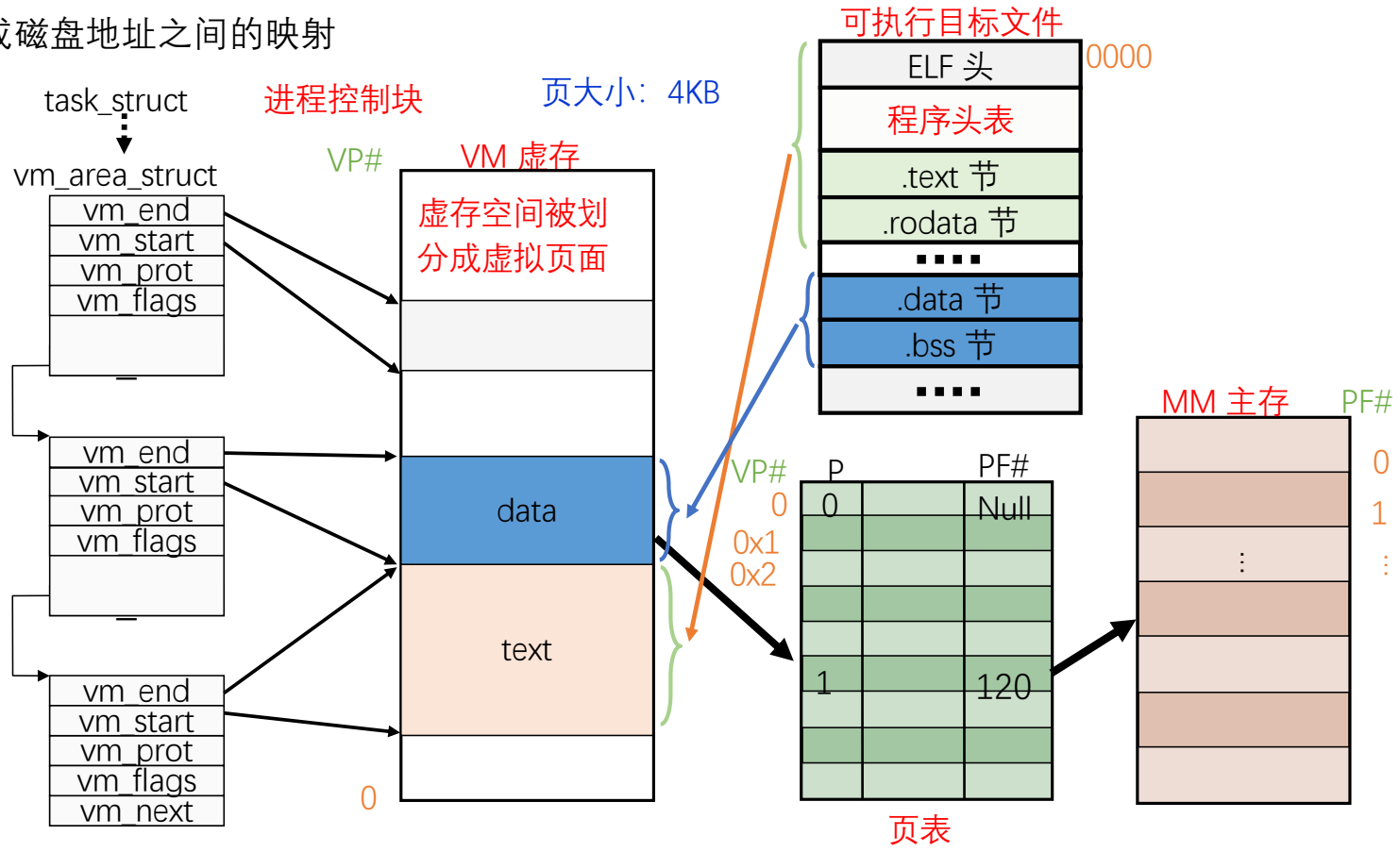
- PTE上添加一些额外的许可位来控制对一个虚拟页面内容的访问
  - SUP：进程只有在内核模式（超级用户）下运行才能访问对应页
  - READ/WRITE/EXEC

  - 违反触发保护故障，内核处理异常，报错“段错误（segmentation fault）”



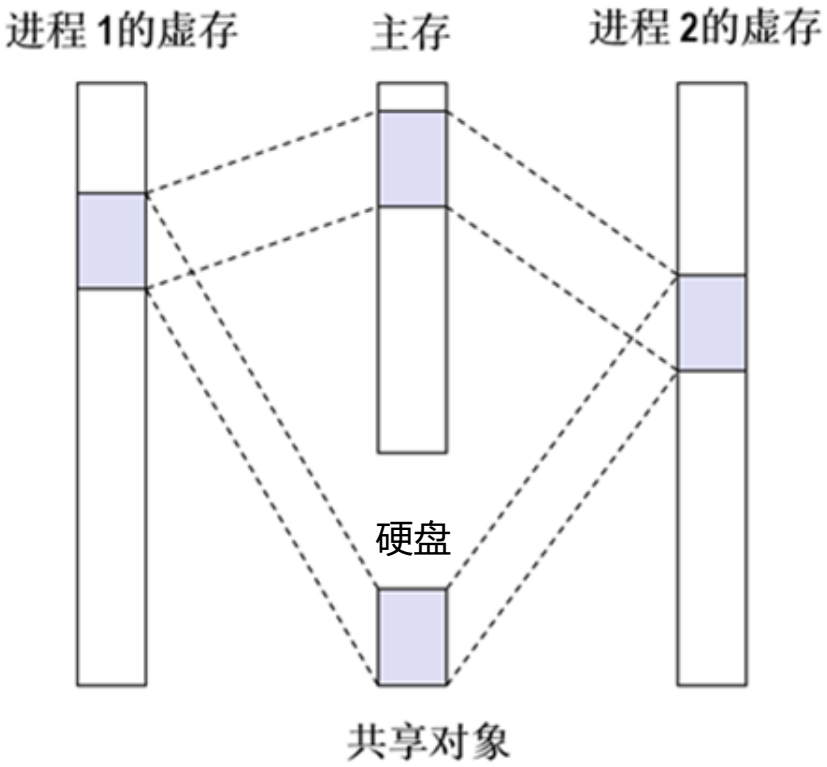
# 存储管理全局图

- 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



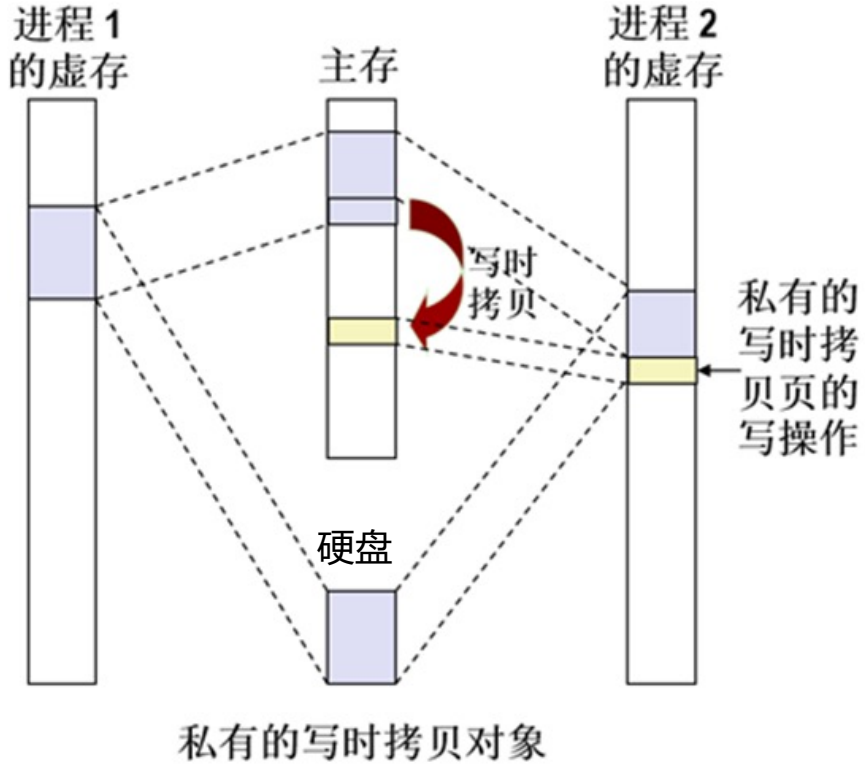
# 共享库中的共享对象

- 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



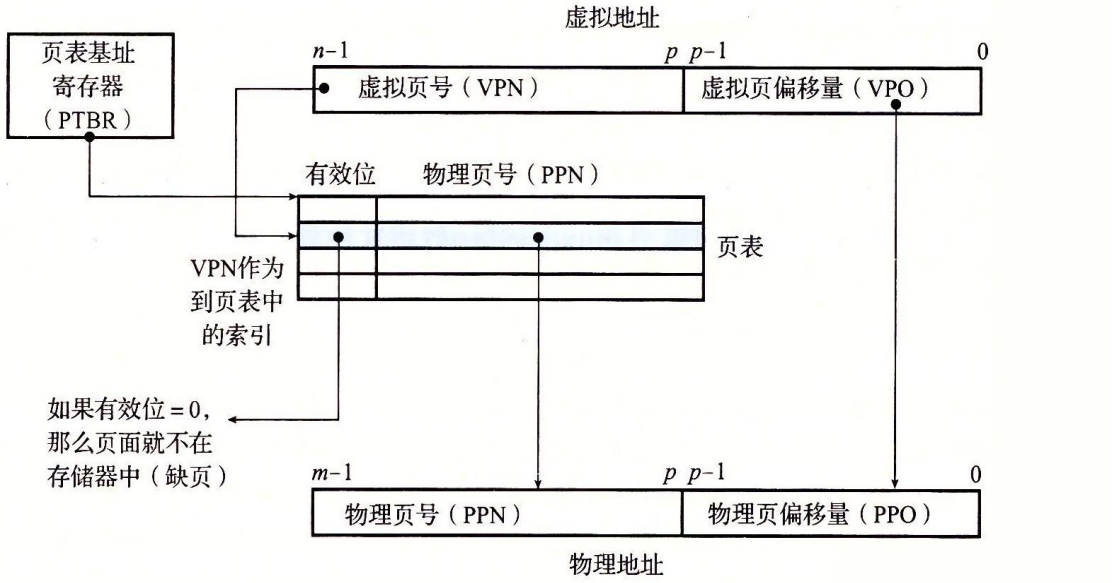
# 共享库中的共享对象

- 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，则触发保护故障，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



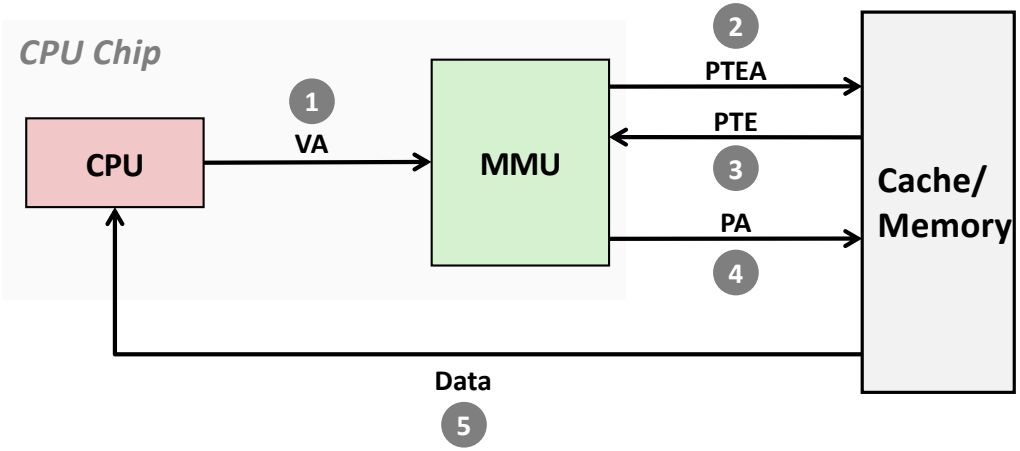
# 地址转换

- 虚拟地址分两个字段
  - 高位字段为虚拟页号（VPN，Virtual page number）
  - 低位字段为页内偏移地址（VPO，Virtual page offset），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（PPN，Physical page number）
  - 低位字段为页内偏移地址（PPO，Physical page offset）
- 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
  - VPO=PPO
- MMU利用页表实现地址转换
  - PTBR指向页表
  - 利用VPN选PTE



# 地址转换

- 页命中
  1. 处理器生成一个虚拟地址，并把它传送给MMU
  2. MMU生成PTE地址，并从高速缓存/主存请求得到它
  3. 高速缓存/主存向MMU返回PTE
  4. MMU构造物理地址，并把它传送给高速缓存/主存
  5. 高速缓存/主存返回所请求的数据字给处理器

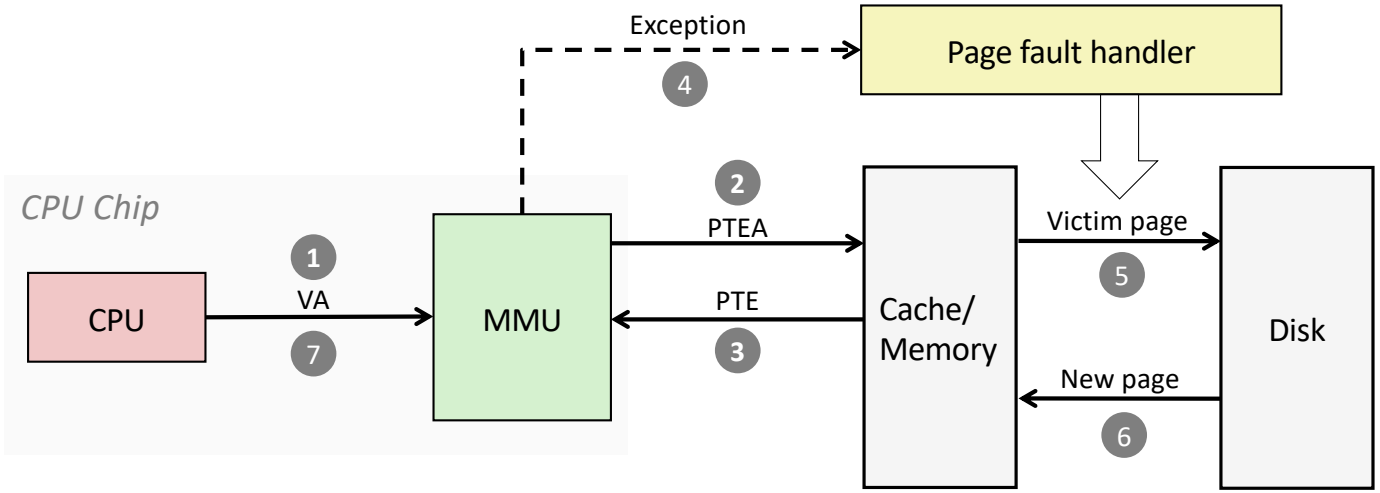


# 地址转换

- 缺页

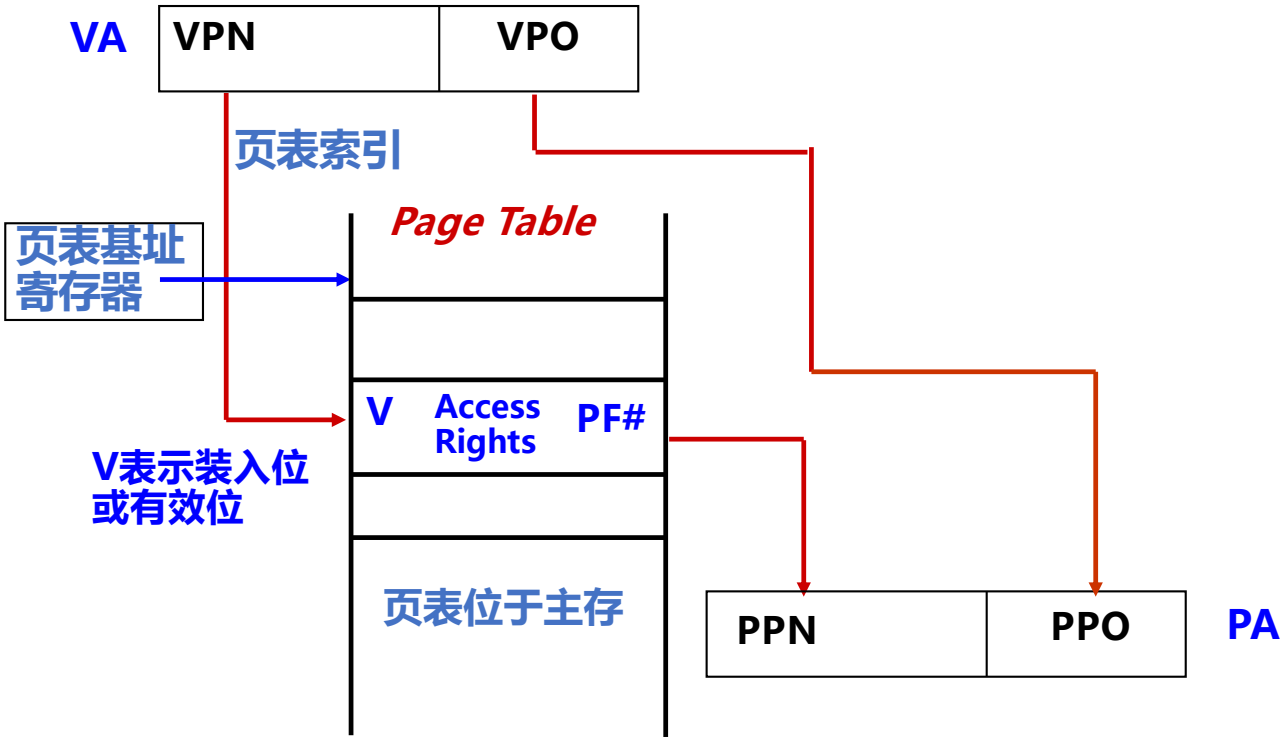
  1-3. 同上，MMU得到PTE

  4. PTE 中的有效位是0，所以MMU触发了一次异常，传递CPU中的控制到操作系统内核中的缺页异常处理程序
  5. 缺页处理程序确定出物理内存中的牺牲页，如果这个页面已经被修改了，则把它换出到磁盘
  6. 缺页处理程序页面调入新的页面，并更新内存中的PTE
  7. 缺页处理程序返回到原来的进程，再次执行导致缺页的指令

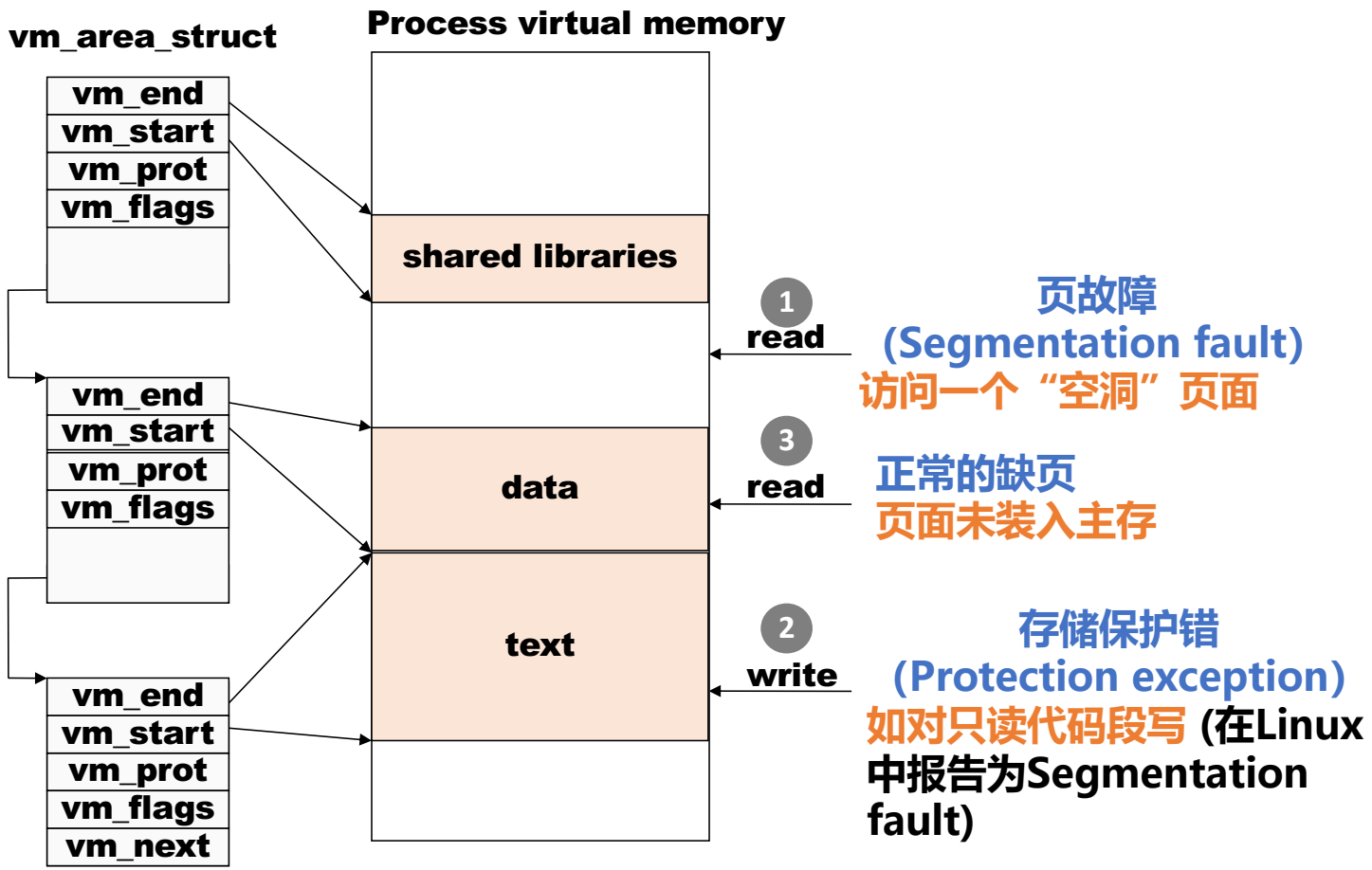


# 地址转换

- 当V=0时，发生缺页
- 当读写操作不符合存取权限（Access Right）时，发生保护违例（访问越权）
- PF#为对应的物理页号（页框号或实页号）



# Linux存储保护机制



# 回顾

- 当参数len为0时，预期返回值是0
- 但是在机器上执行时，却发生访存异常
- 当len为int型时则正常

Why?

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



# 虚拟存储器

- 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
- 第三讲：具有TLB和cache的存储系统
  - 快表、层次化存储系统结构
  - CPU的访存过程
- 第四讲：存储系统实例
  - Core i7+Linux存储系统

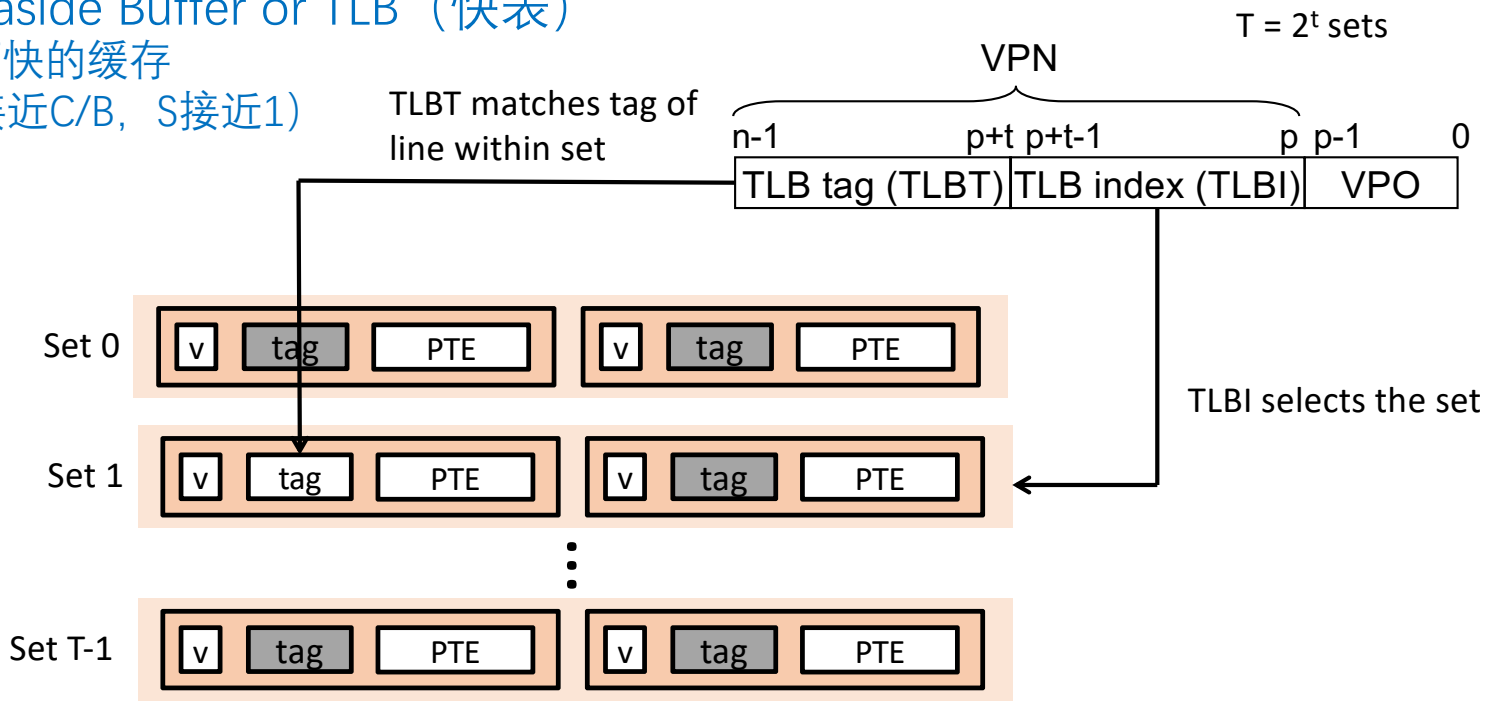
# TLB

一次存储器引用要访问几次主存?

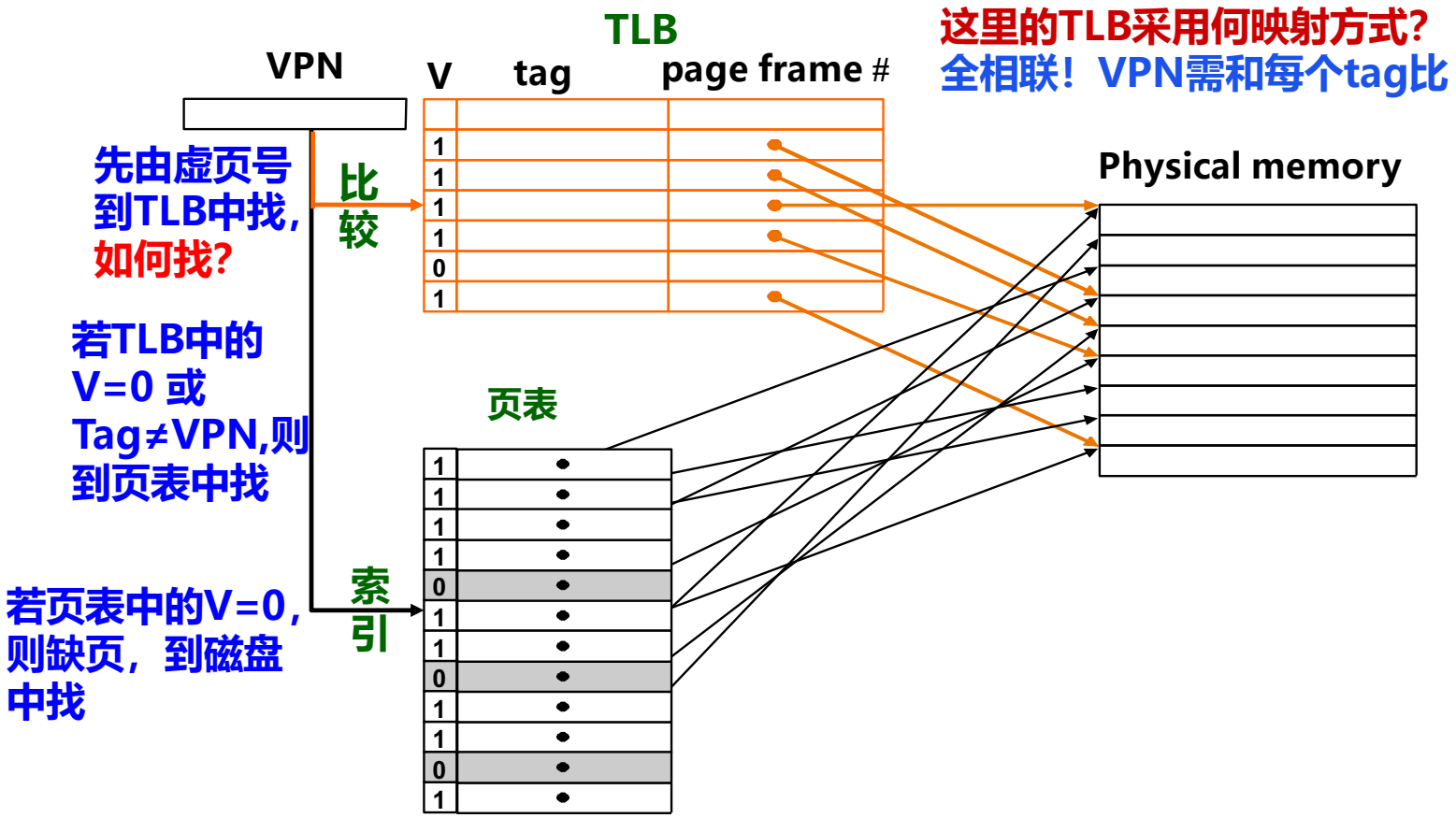
页表在主存，至少 1 次

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）

- MMU中一个小而快的缓存
- 相联度很高（E接近C/B，S接近1）

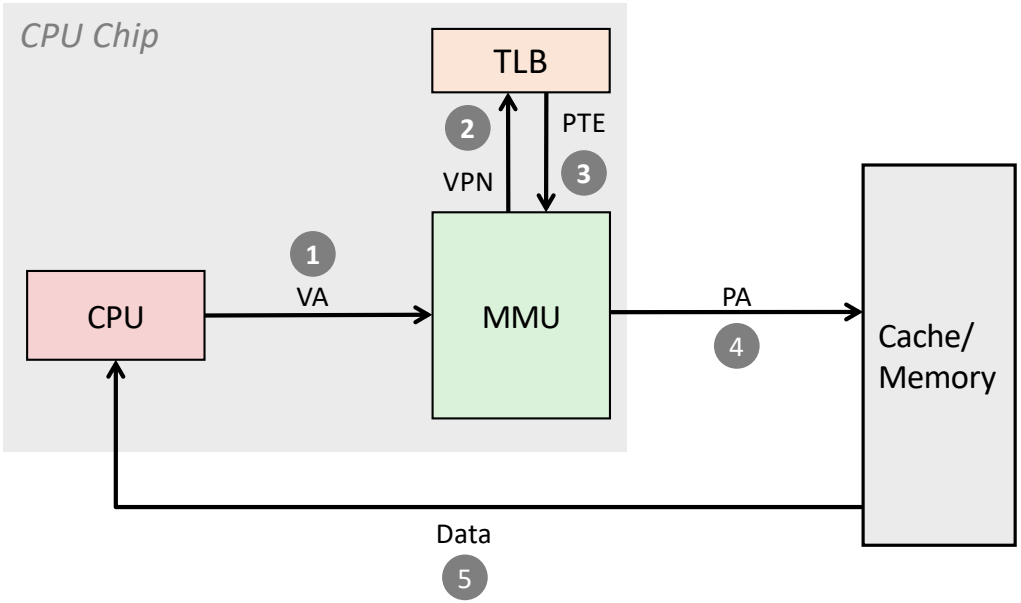


# TLB


TLB
VPN
V
tag
page frame #
这里的TLB采用何映射方式?
全相联！VPN需和每个tag比
先由虚页号到TLB中找，如何找?
比较
Physical memory
若TLB中的V=0 或Tag≠VPN,则到页表中找
页表
索引
若页表中的V=0，则缺页，到磁盘中找

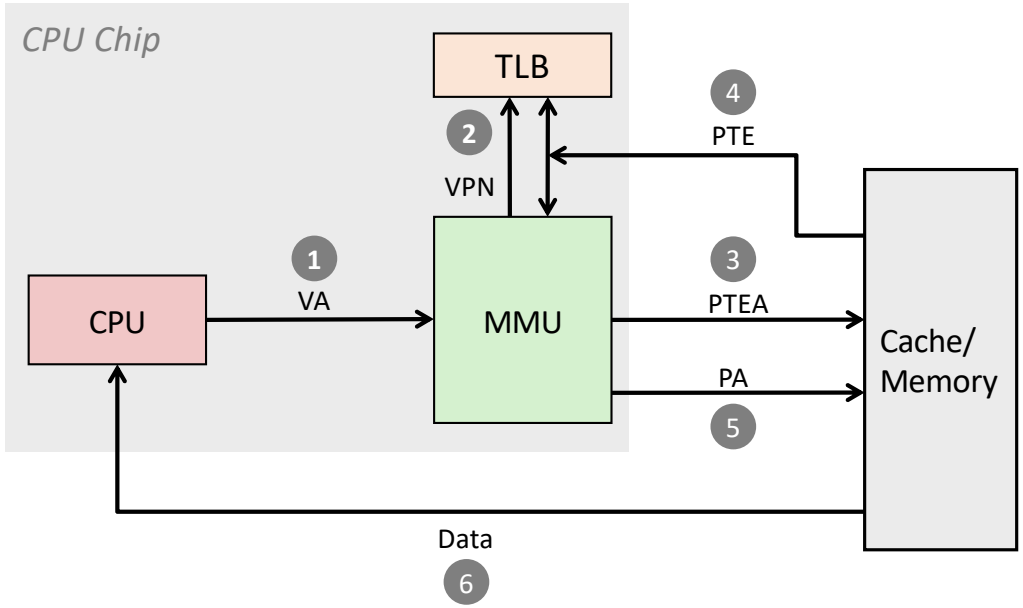
# TLB命中

1. 处理器生成一个虚拟地址，并把它传送给MMU
2-3. MMU从TLB取到PTE（减少一次访存）
4. MMU构造物理地址，并把它传送给高速缓存/主存
5. 高速缓存/主存返回所请求的数据字给处理器

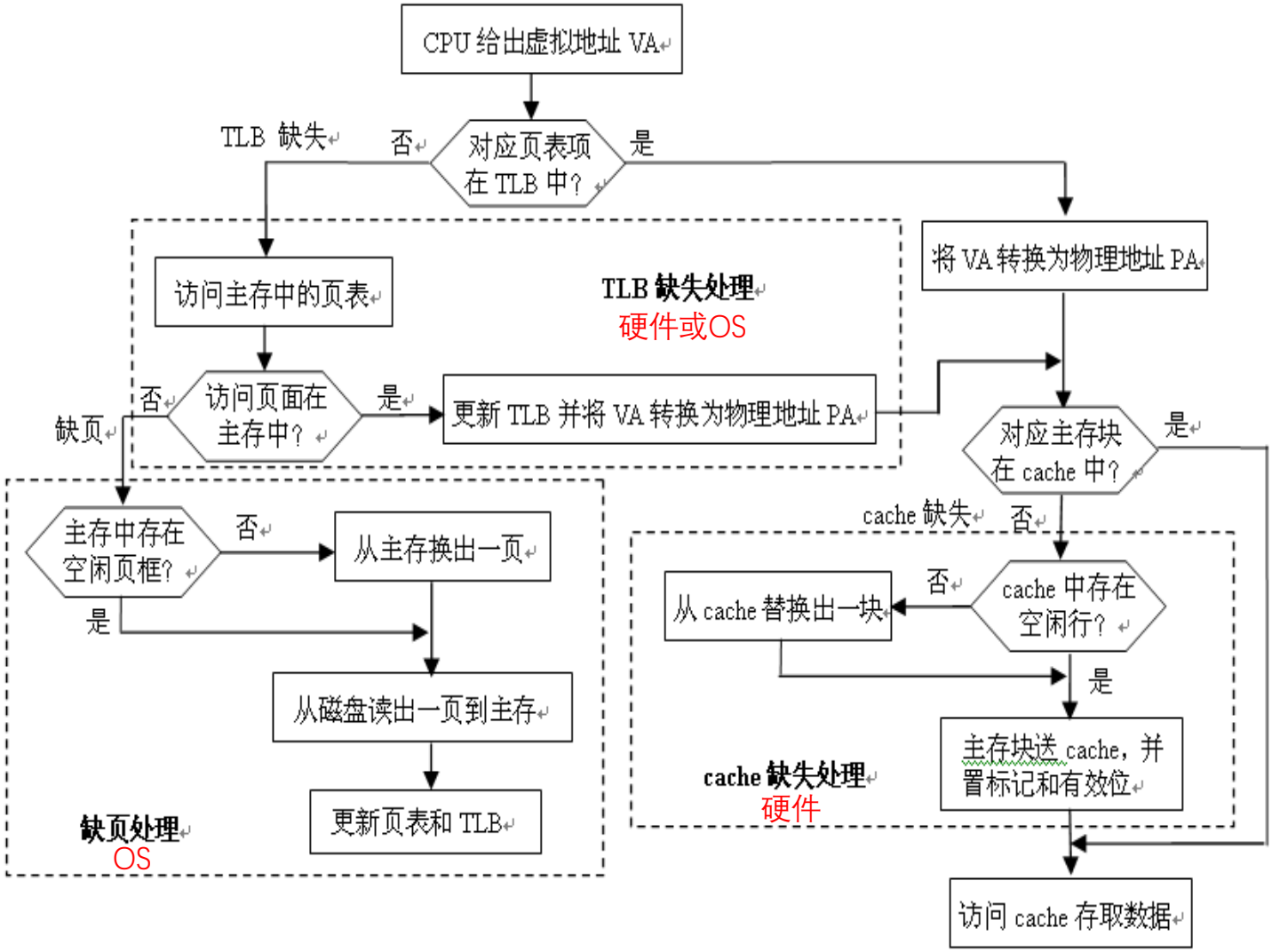


# TLB不命中

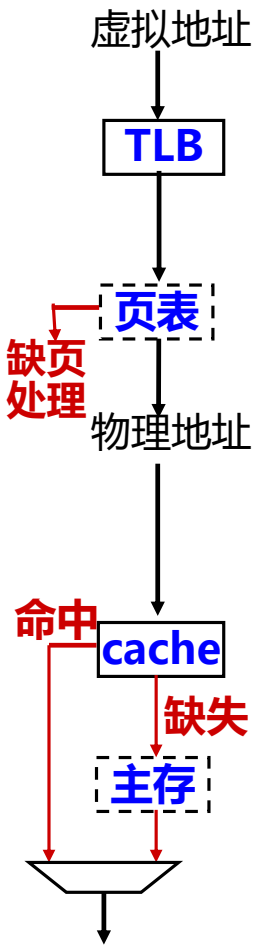
- 回到多一次PTE访存的情形
- TLB不命中的情况很少

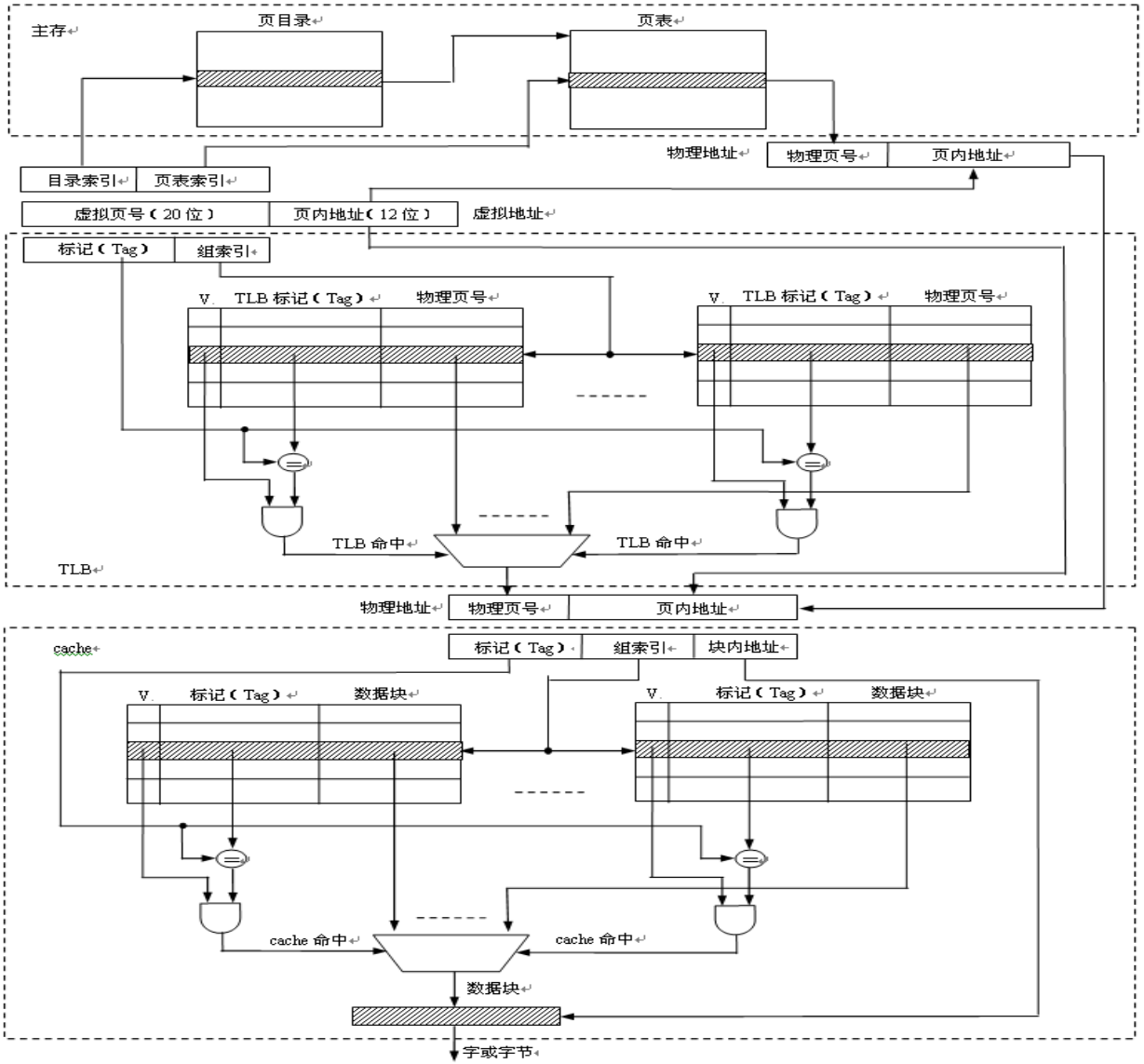


# CPU访存过程


CPU 给出虚拟地址 VA
对应页表项在 TLB 中?
TLB 缺失
否
是
将 VA 转换为物理地址 PA
访问主存中的页表
TLB 缺失处理
硬件或OS
访问页面在主存中?
否
是
缺页
更新 TLB 并将 VA 转换为物理地址 PA
对应主存块在 cache 中?
是
cache 缺失
否
主存中存在空闲页框?
否
从主存换出一页
是
从磁盘读出一页到主存
更新页表和 TLB
缺页处理
OS
从 cache 替换出一块
否
cache 中存在空闲行?
是
主存块送 cache，并置标记和有效位
cache 缺失处理
硬件
访问 cache 存取数据

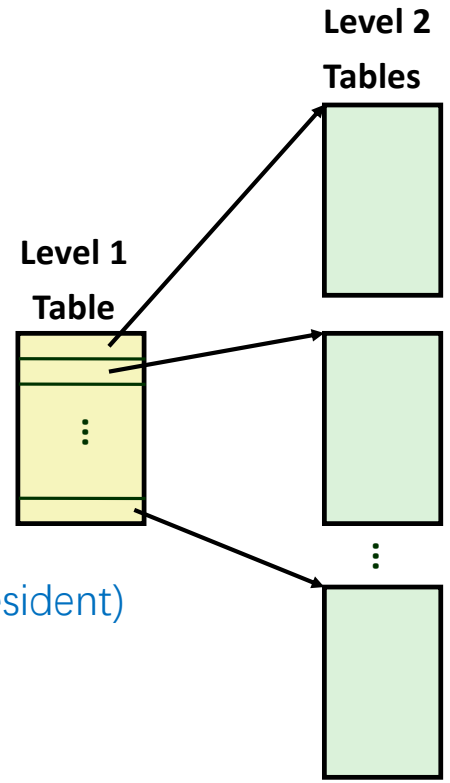
# CPU访存过程


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

# 多级页表

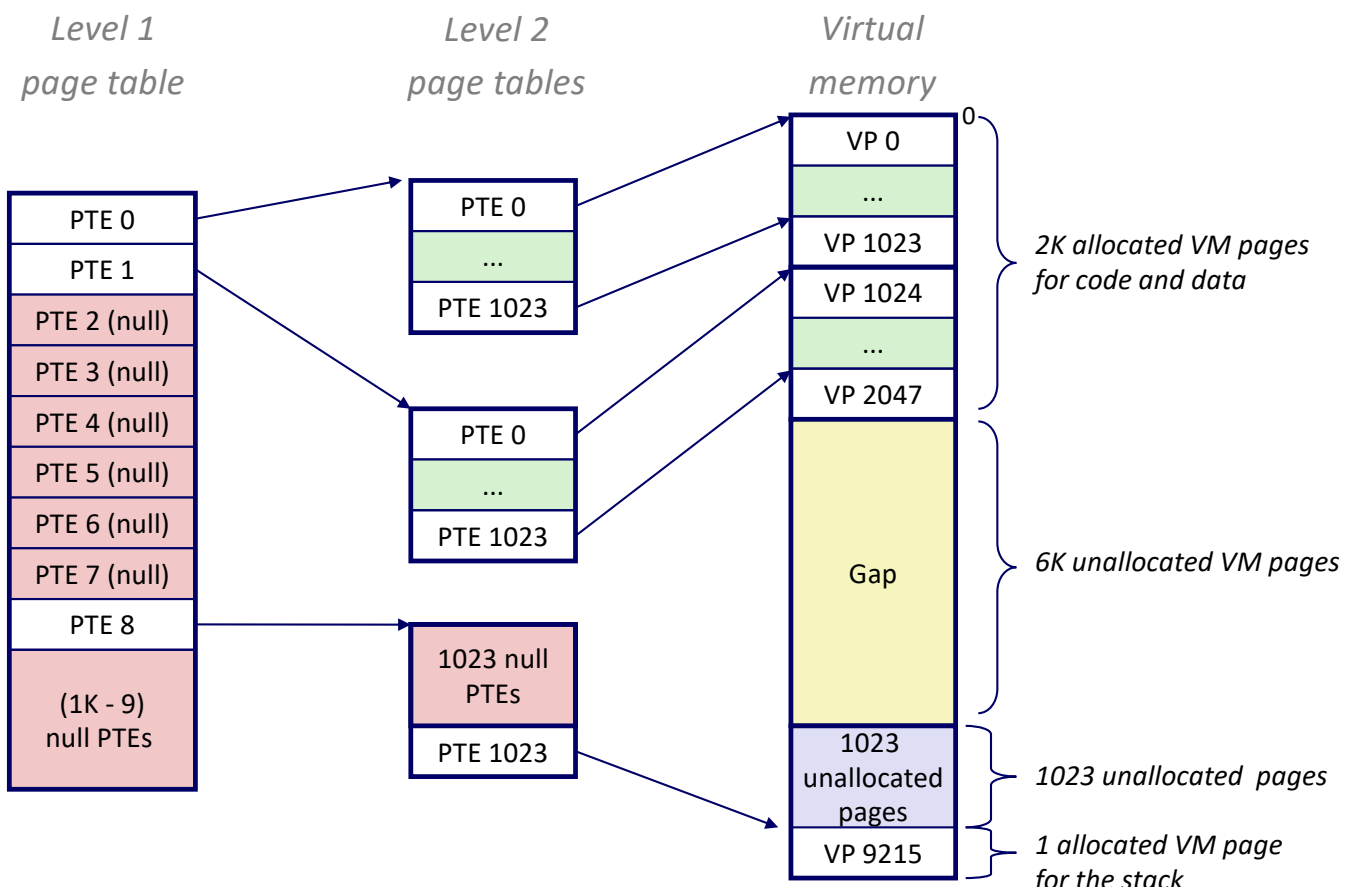
- Suppose:
  - 4KB ($2^{12}$) page size, 48-bit address space, 8-byte PTE
- Problem:
  - Would need a 512 GB page table!
    - $2^{48} / 2^{12} * 2^{3} = 2^{39}$ bytes
- Common solution: Multi-level page table
- Example: 2-level page table
  - Level 1 table: each PTE points to a page table (always memory resident)
  - Level 2 table: each PTE points to a page (paged in and out like any other data)



# 两级页表

- 内存片未使用，一级页表某PTE空
- 一级页表某PTE空，对应二级页表不存在

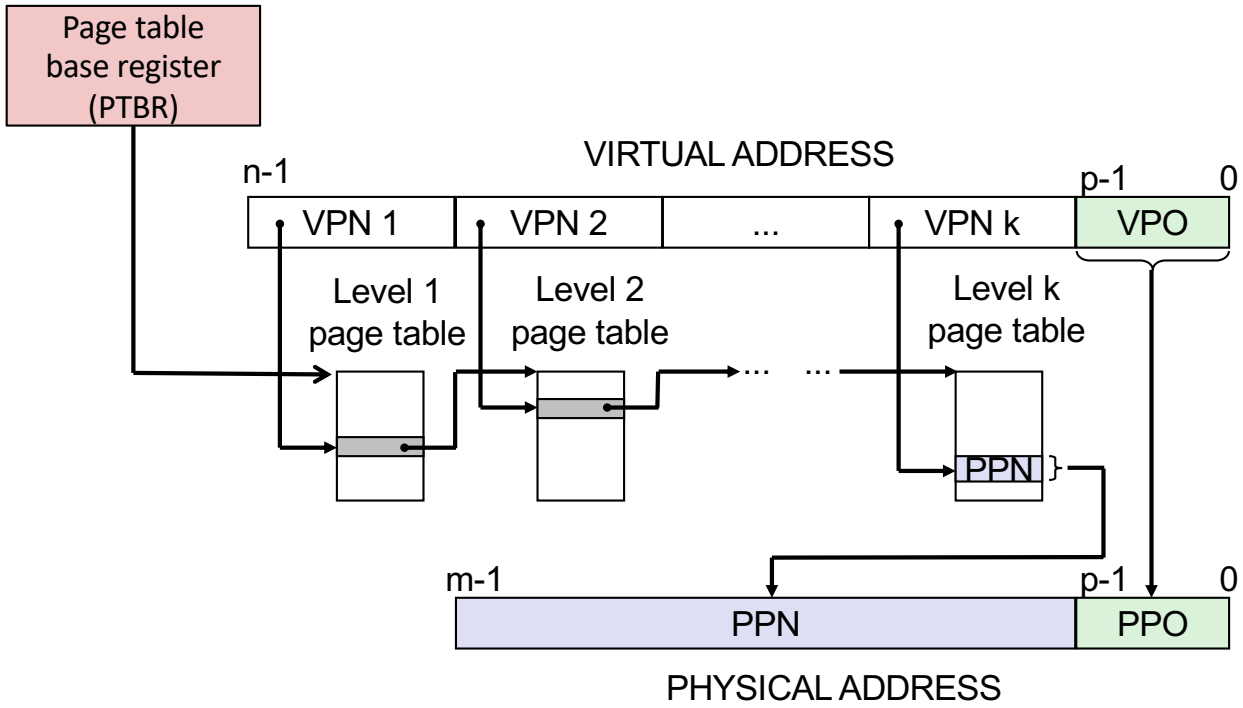
- 仅一级页表常驻主存
- 二级页表按需调入




*32bit address, 4KB page, 4-byte PTE*

# k级页表地址翻译

- 虚拟地址=k*VPN+1*VPO
- TLB缓存不同层次的PTE，访问k个PTE实际上速度接近单级页表



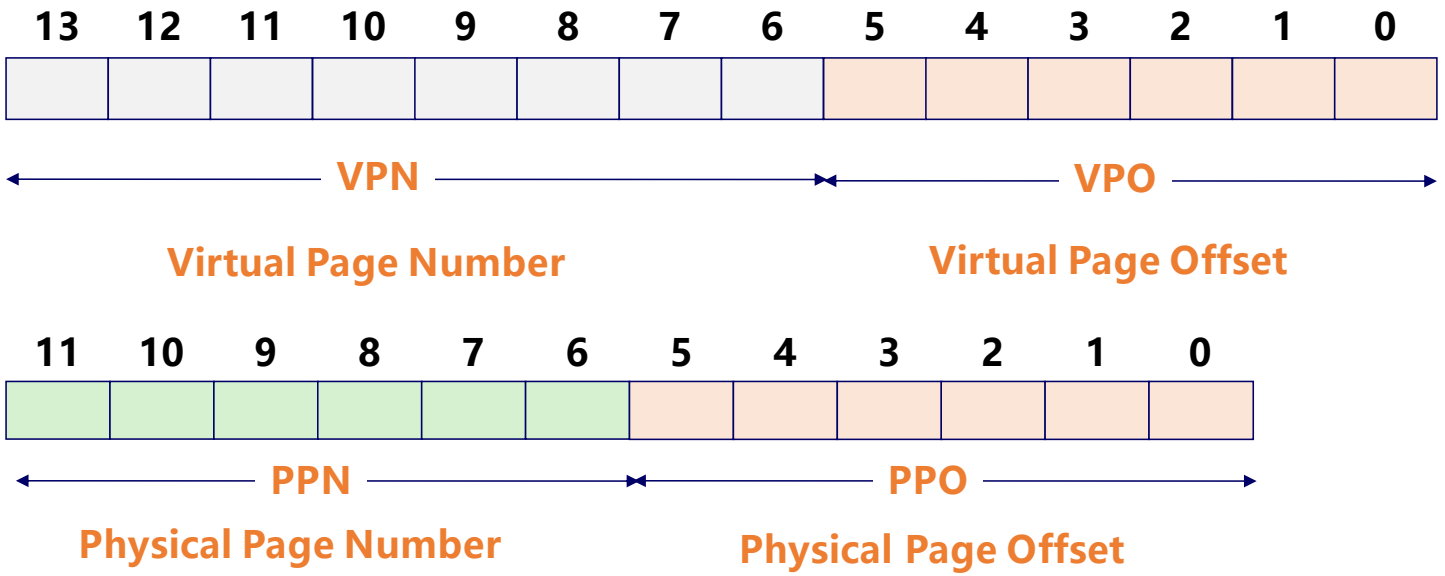
# 缩写

- 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）

# 存储系统例子

- 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）

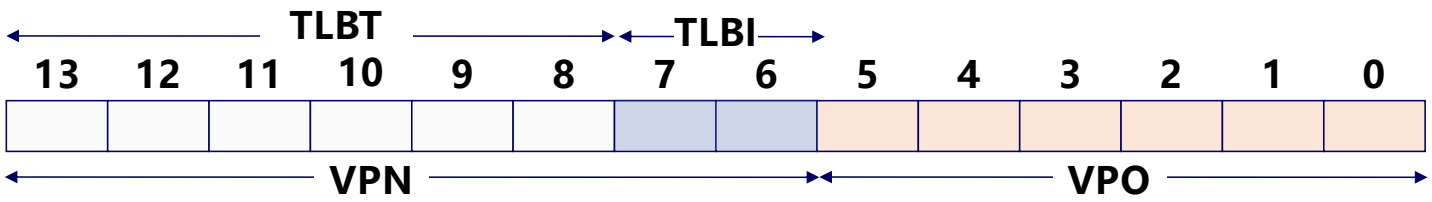
**页表项数=$2^{14}/64=256$**



# 存储系统例子（续）

- 假定TLB参数和内容如下
  - 16个TLB项，4路组相联，则TLBT和TLBI各占几位?

| Set | Tag | PPN | Valid | Tag | PPN | Valid | Tag | PPN | Valid | Tag | PPN | Valid |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

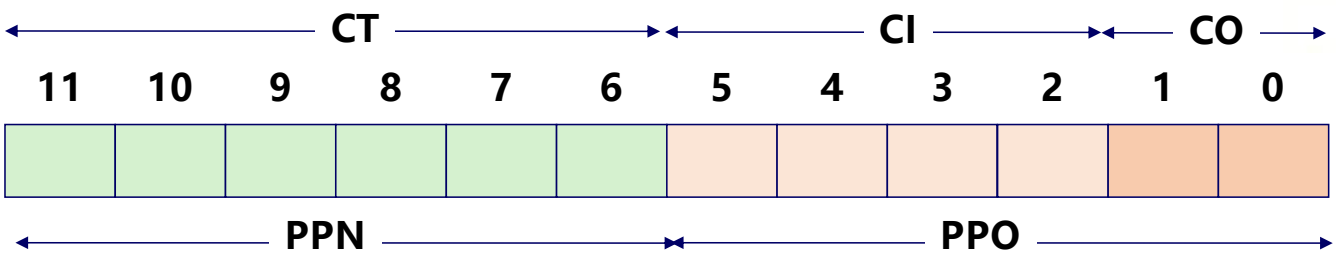


# 存储系统例子（续）

- 假定Cache参数和内容如下
  - 16组，4字节块，直接映射
  - 主存地址如何划分?

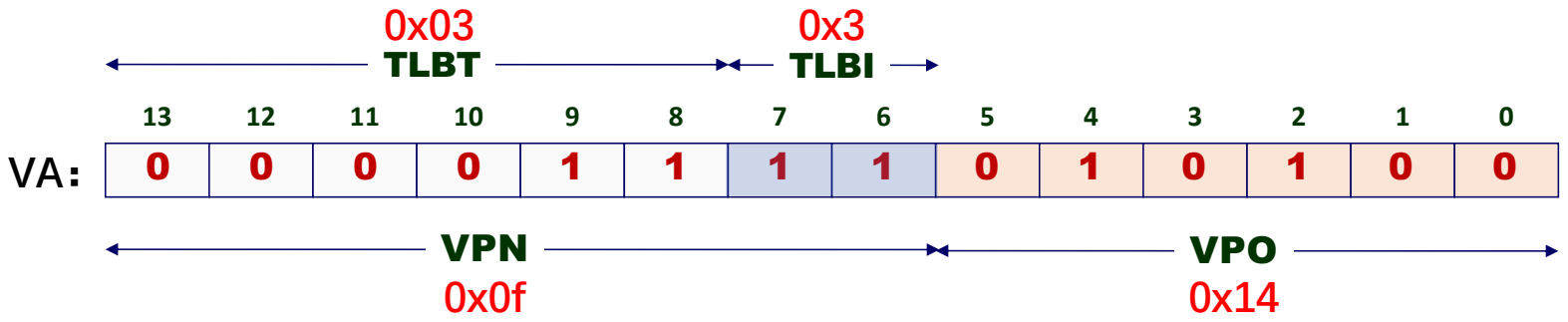
| 索引 | 标记位 | 有效位 | 块 0 | 块 1 | 块 2 | 块 3 |
|---|---|---|---|---|---|---|
| 0 | 19 | 1 | 99 | 11 | 23 | 11 |
| 1 | 15 | 0 | — | — | — | — |
| 2 | 1B | 1 | 00 | 02 | 04 | 08 |
| 3 | 36 | 0 | — | — | — | — |
| 4 | 32 | 1 | 43 | 6D | 8F | 09 |
| 5 | 0D | 1 | 36 | 72 | F0 | 1D |
| 6 | 31 | 0 | — | — | — | — |
| 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 | — | — | — | — |

**块偏移（CO）：2位**
**组索引（CI）：4位**



# 存储系统例子（续）

- 假定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逻辑地址为0x03D4的short类型数据，说明访存过程。

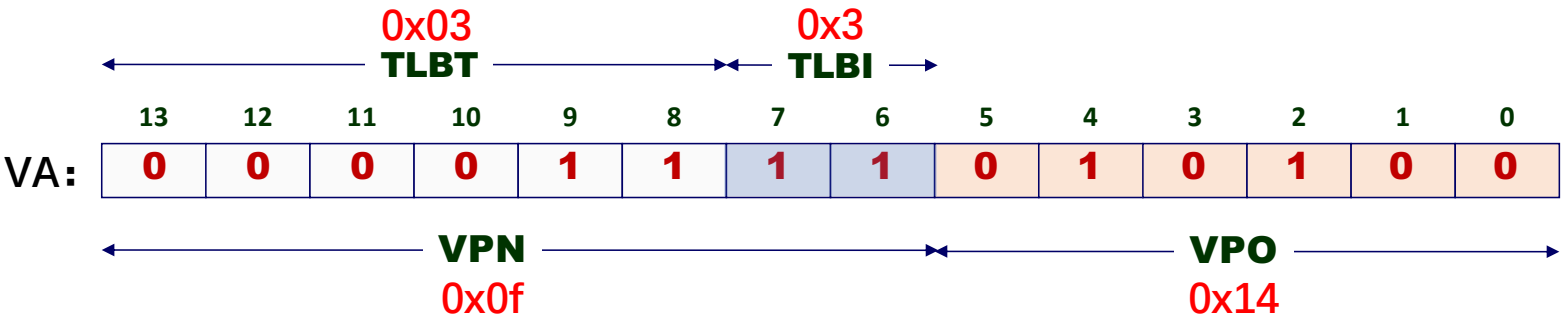



1. MMU从VA取出VPN，检查TLB看PTE是否缓存

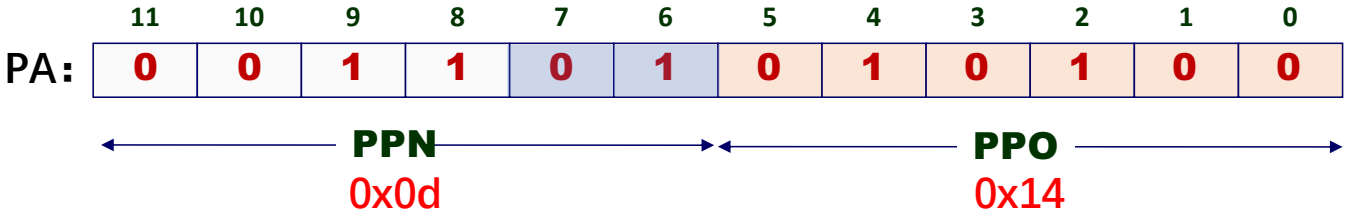
2. TLB从VPN取出TLB标记和索引，命中，取出缓存的PPN（0x0d）返回给MMU

| *Set* | *Tag* | *PPN* | *Valid* | *Tag* | *PPN* | *Valid* | *Tag* | *PPN* | *Valid* | *Tag* | *PPN* | *Valid* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

# 存储系统例子（续）

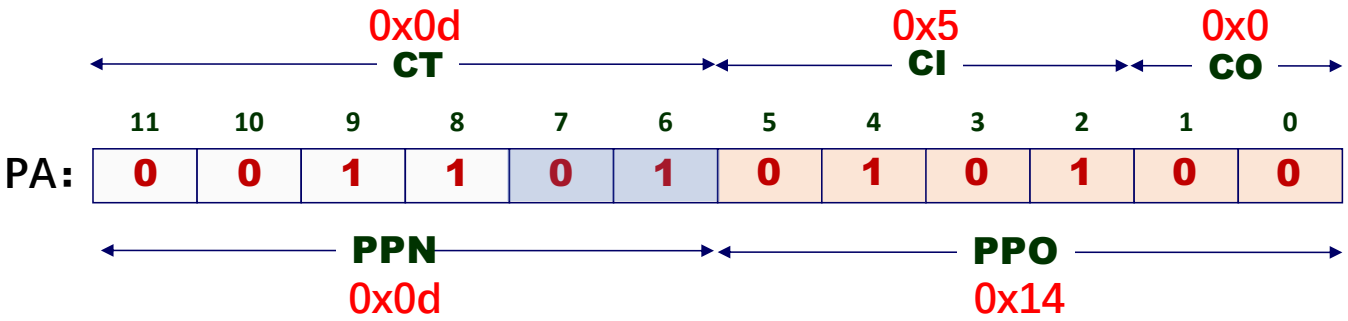



1. MMU从VA取出VPN，检查TLB看PTE是否缓存
2. TLB从VPN取出TLB标记和索引，命中，取出缓存的PPN（0x0d）返回给MMU
3. **MMU组合PPN和VPO得到物理地址（~~00~~001101 010100=0x0354）**



# 存储系统例子（续）

1. MMU从VA取出VPN，检查TLB看PTE是否缓存
2. TLB从VPN取出TLB标记和索引，命中，取出缓存的PPN（0x0d）返回给MMU
3. MMU组合PPN和VPO得到物理地址（~~00~~001101 010100=0x0354）
4. **MMU发送PA给Cache，Cache从PA提取缓存偏移CO（0x0）、组索引CI（0x5）、标记CT（0x0d）**
5. **Cache检测到命中，short 2字节，主存块4字节，小端，Cache返回数据字（0x7236）给MMU，MMU传回CPU**




| 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 | — | — | — | — |

# 存储系统例子（续）

1. MMU从VA取出VPN，检查TLB看PTE是否缓存
2. TLB从VPN取出TLB标记和索引，命中，取出缓存的PPN（0x0d）返回给MMU
3. MMU组合PPN和VPO得到物理地址（~~00~~001101 010100=0x0354）
4. MMU发送PA给Cache，Cache从PA提取缓存偏移CO（0x0）、组索引CI（0x5）、标记CT（0x0d）
5. Cache检测到命中，short 2字节，主存块4字节，小端，Cache返回数据字（0x7236）给MMU，MMU传回CPU

**其他情况**

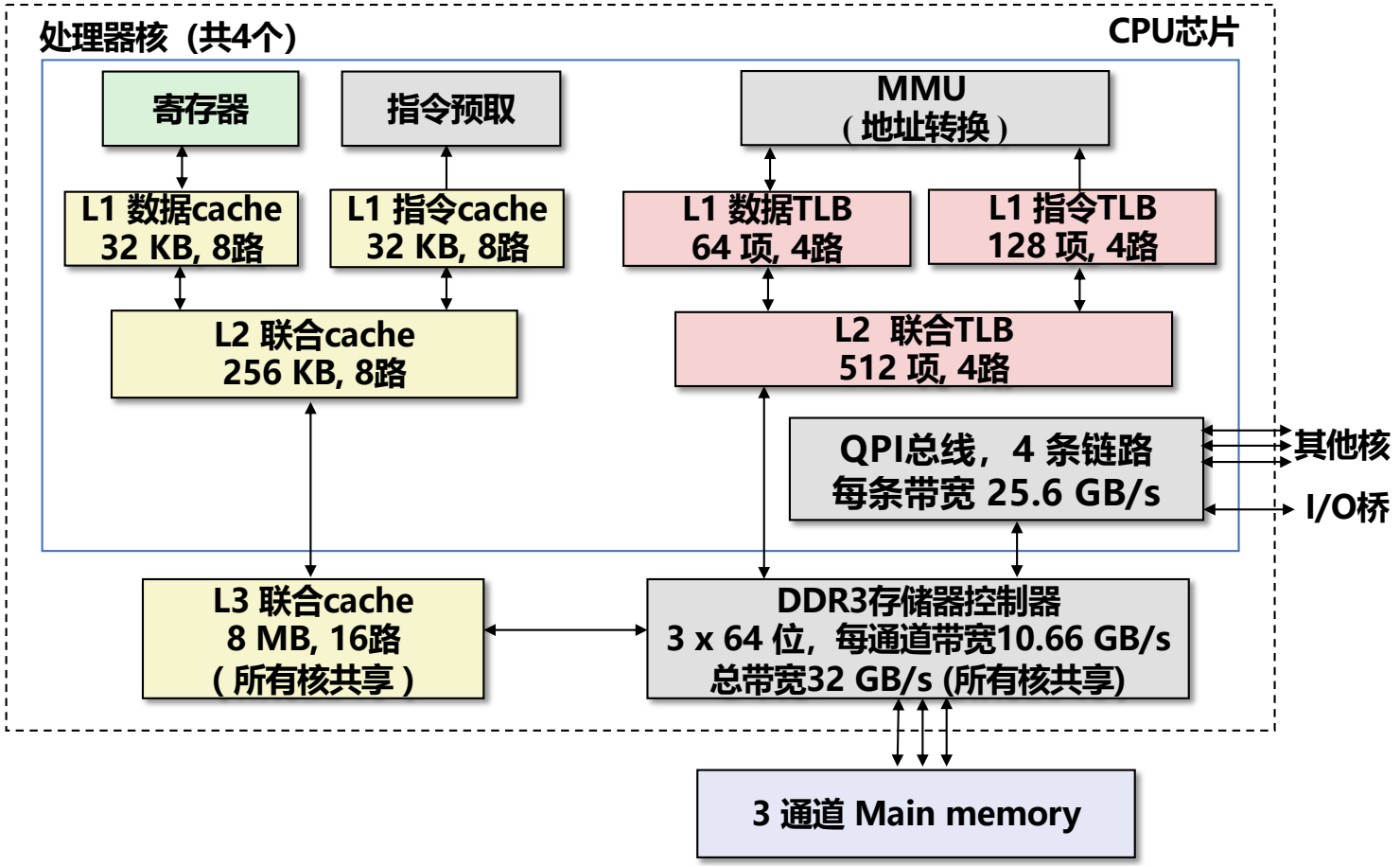
- TLB不命中，MMU访问页表取PTE
- 得到PTE无效，处理缺页
- 得到PTE有效，cache不命中

# 虚拟存储器

- 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
- 第三讲：具有TLB和cache的存储系统
  - 快表、层次化存储系统结构
  - CPU的访存过程
- 第四讲：存储系统实例
  - Core i7+Linux存储系统

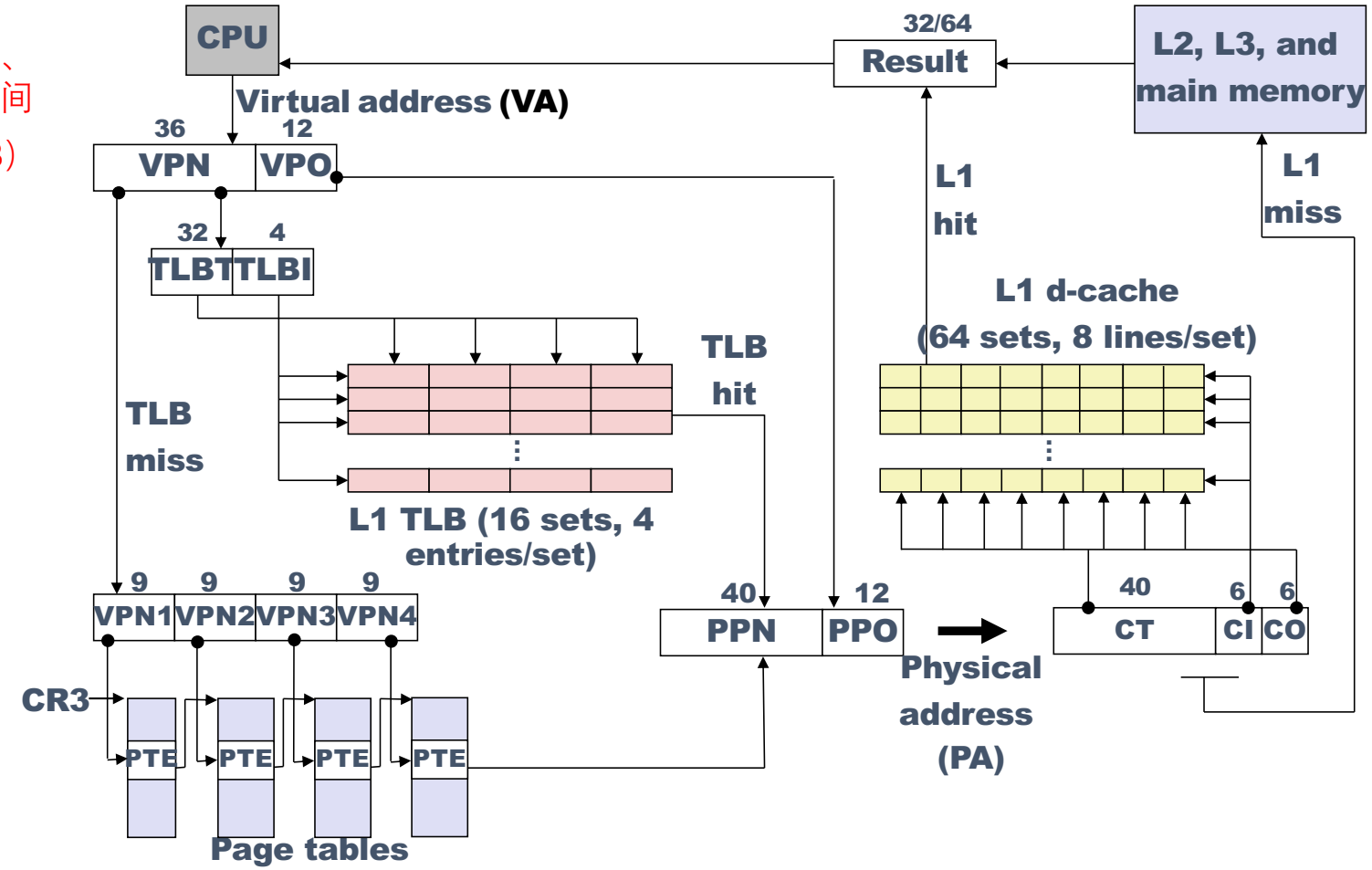
# Intel Core i7+Linux存储系统

- 现代的Intel处理器架构（IA）支持64位虚拟/物理地址空间

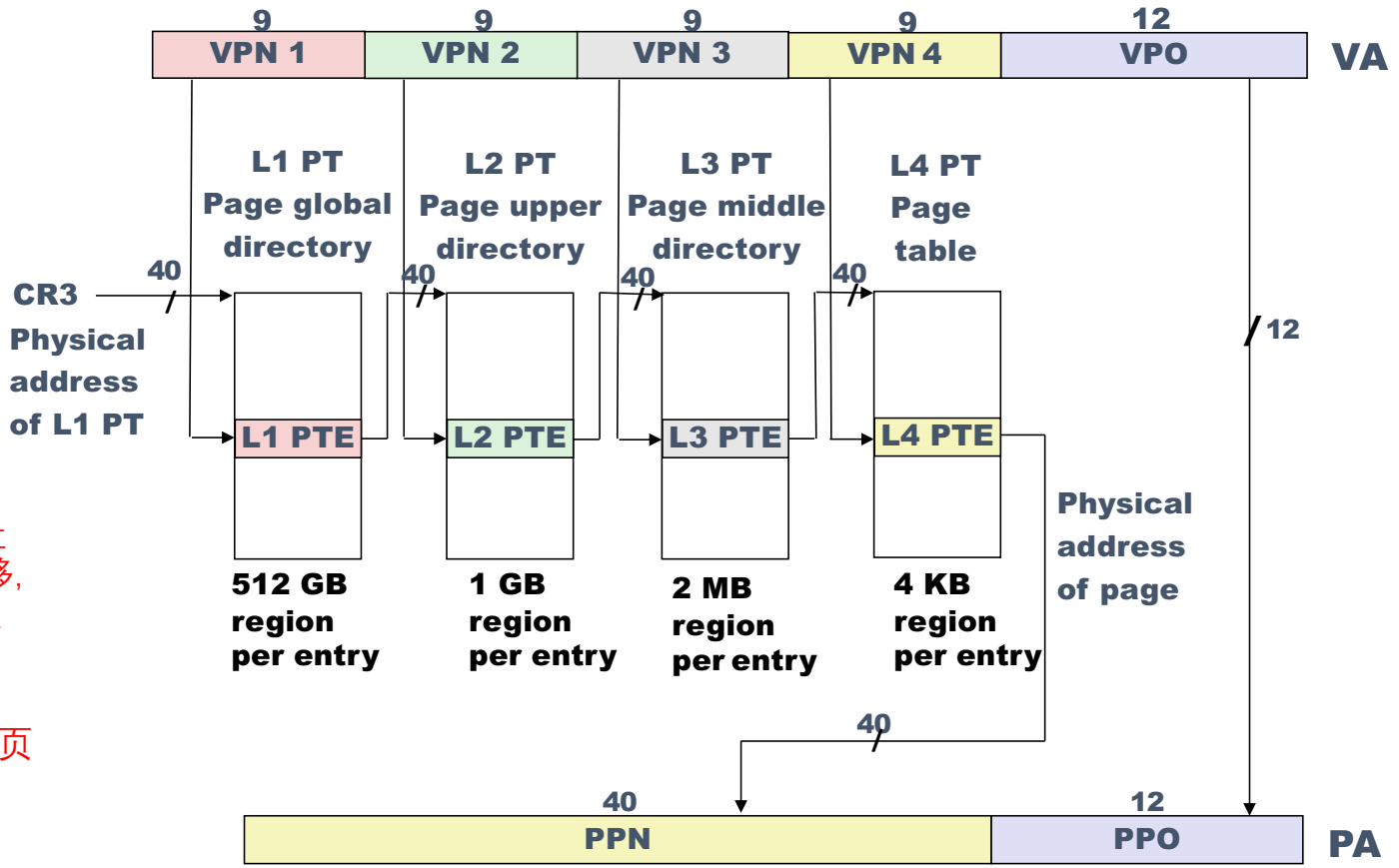


# Core i7地址翻译

- Core i7实现支持48位（256TB）虚拟地址空间、52位（4PB）物理地址空间
- 兼容模式支持32位（4GB）虚拟/物理地址空间
- Cache块64B
- Linux默认页4KB
- 4级页表



# Core i7地址翻译

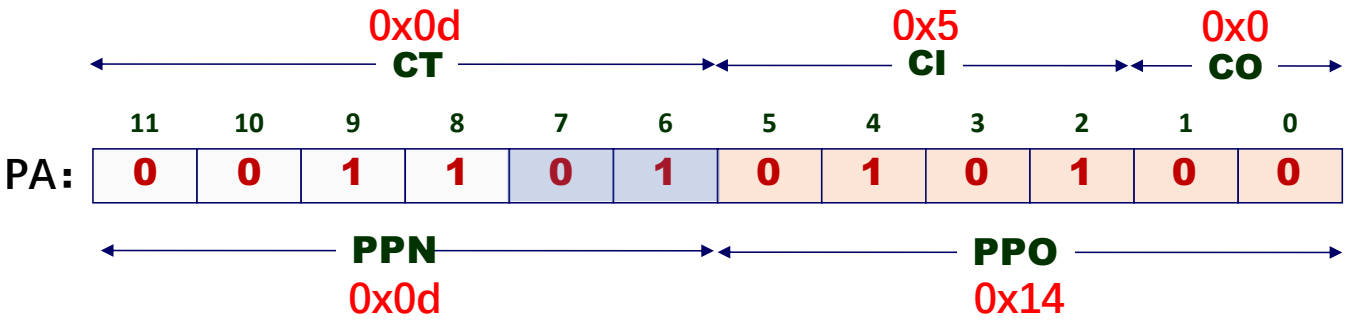



- 36位VPN分成4片
  - CR3寄存器：L1 PT物理地址
  - VPN1：到某个L1 PTE的偏移，该PTE记录某个L2 PT的基址
  - …
- 虚拟地址空间划分：
  $2^{48}$B=512*512*512*512*4KB/页
- 物理地址空间划分：
  $2^{52}$B=$2^{40}$页框*4KB/页框

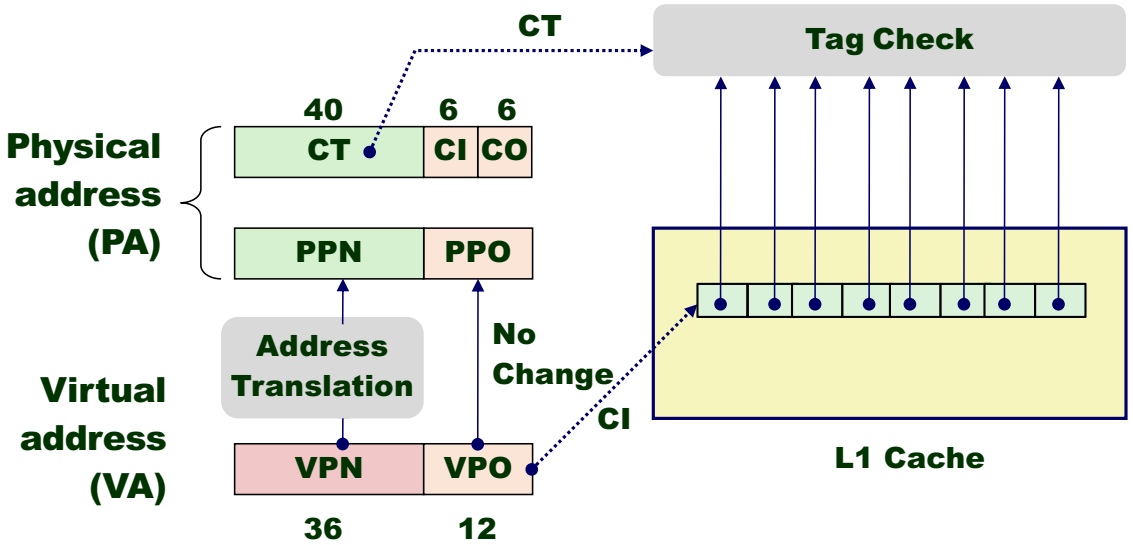
# Core i7地址翻译加速

1. MMU从VA取出VPN，检查TLB看PTE是否缓存
2. TLB从VPN取出TLB标记和索引，命中，取出缓存的PPN（0x0d）返回给MMU
3. MMU组合PPN和VPO得到物理地址（~~00~~001101 010100=0x0354）
4. MMU发送PA给Cache，Cache从PA提取缓存偏移CO（0x0）、组索引CI（0x5）、标记CT（0x0d）
5. Cache检测到命中，short 2字节，主存块4字节，小端，Cache返回数据字（0x7236）给MMU，MMU传回CPU

- **PPO=VPO**
- **CI在VPO中**
- **Cache行可以根据VA更早地取出，即省略4**
- **3得到PPN即知道CT，和已取出的cache行的tag比较即可完成5**



# Core i7地址翻译加速




- VIPT (Virtually indexed, physically tagged)
  - 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。

# 练习1

下列命中组合情况中，一次访存过程中不可能发生的是（ ）

A. TLB未命中、cache未命中、Page命中

B. TLB命中、cache未命中、Page命中

C. TLB未命中、cache命中、Page命中

D. TLB未命中、cache命中、Page未命中

正确答案：D

# 练习2

以下有关快表（TLB）的叙述中，错误的是（ ）

A. 快表中存放的是当前进程的常用页表项

B. 在快表中命中时，在L1 cache中一定命中

C. 引入快表的目的是为了加快地址转换速度

D. 快表是一种高速缓存，一定在CPU中

正确答案：B