第6周　虚拟存储器

第1讲　分页存储管理的基本概念
第2讲　虚拟存储器及虚拟地址空间
第3讲　分页存储管理的实现
第4讲　存储器层次结构及其访问过程
第5讲　段式和段页式虚拟存储管理
第6讲 存储保护

早期分页方式的概念

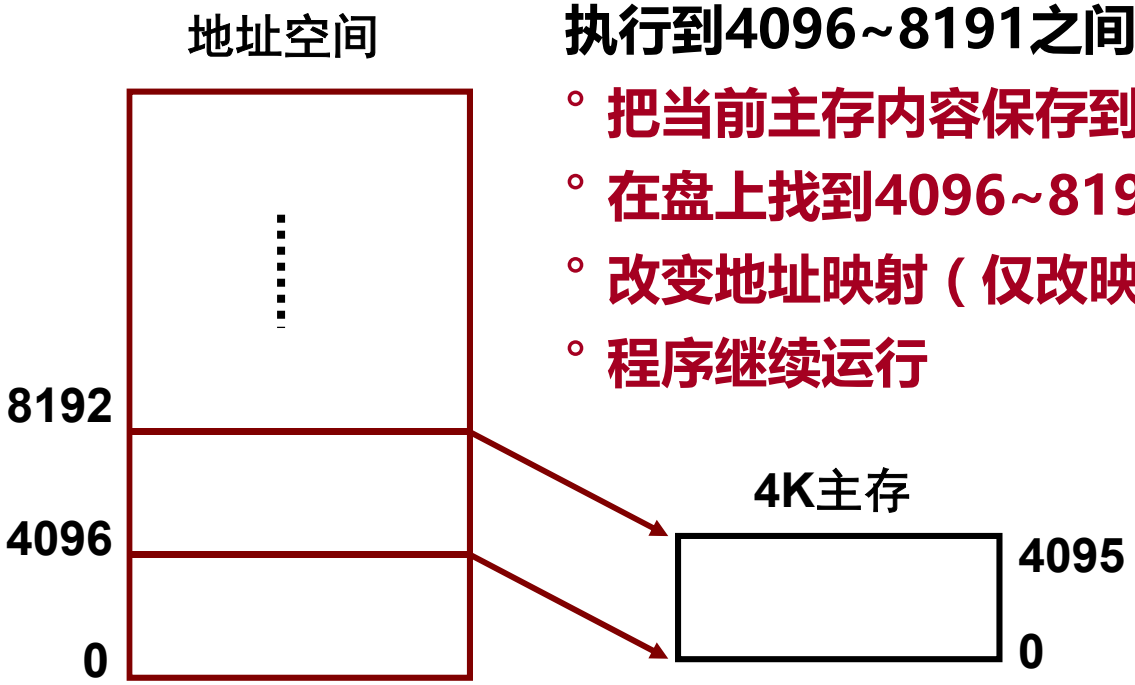
早期：程序员自己管理主存，通过分解程序并覆盖主存的方式执行程序

- 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，则指令可寻址范围是多少？

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（仅改映射区间号（页号））**
- **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**
- **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：
  - 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）
  - 程序块可装到存储器中可用的存储块中
  - 无需用连续页框来存放一个进程
  - 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（Address Mapping ）
- 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内存地址，也称为实地址、主存地址。

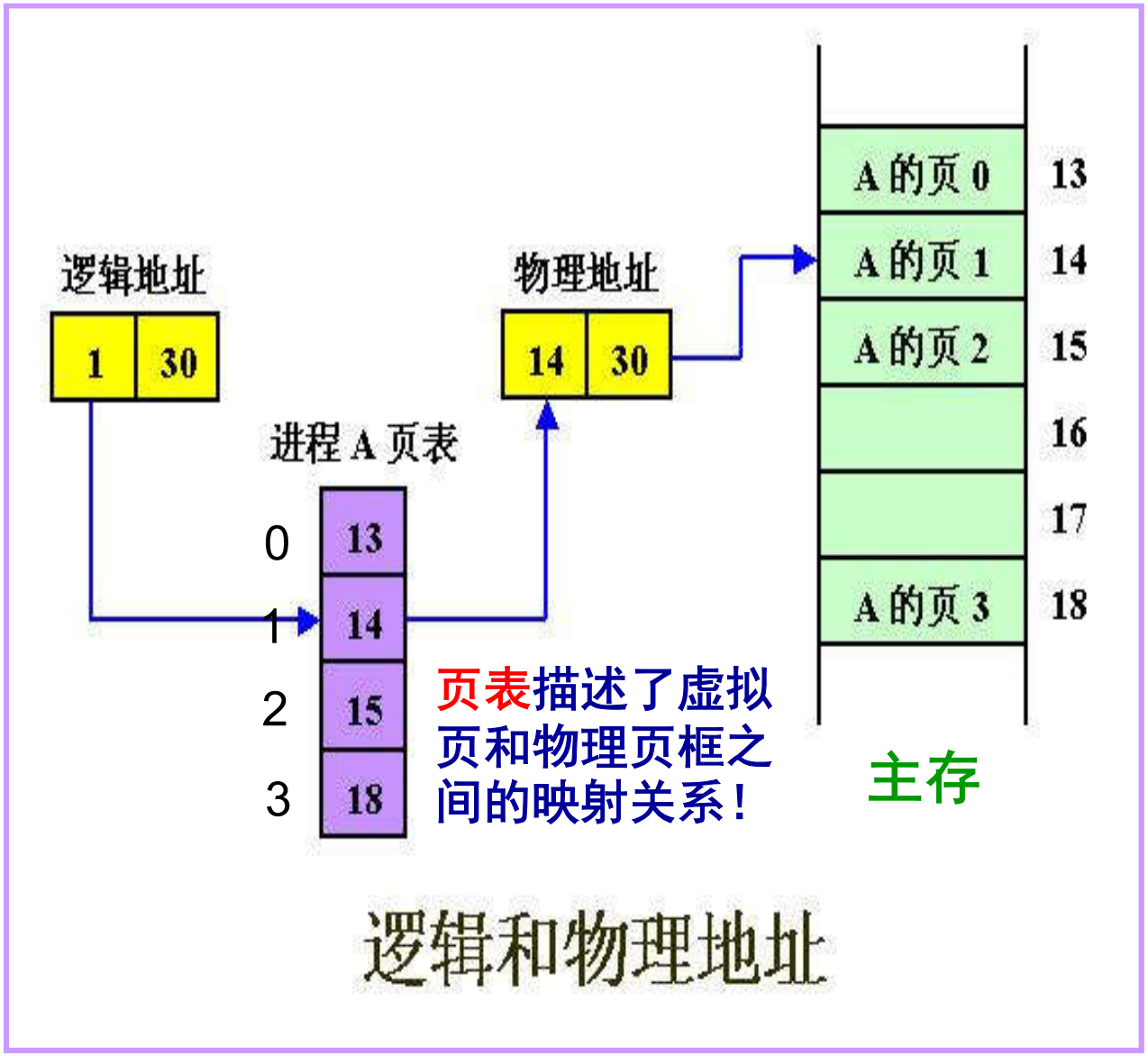
分页（Paging）

问题：是否需要将一个进程的全部都装入内存？

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

优点：浪费的空间最多是最后一页的部分！



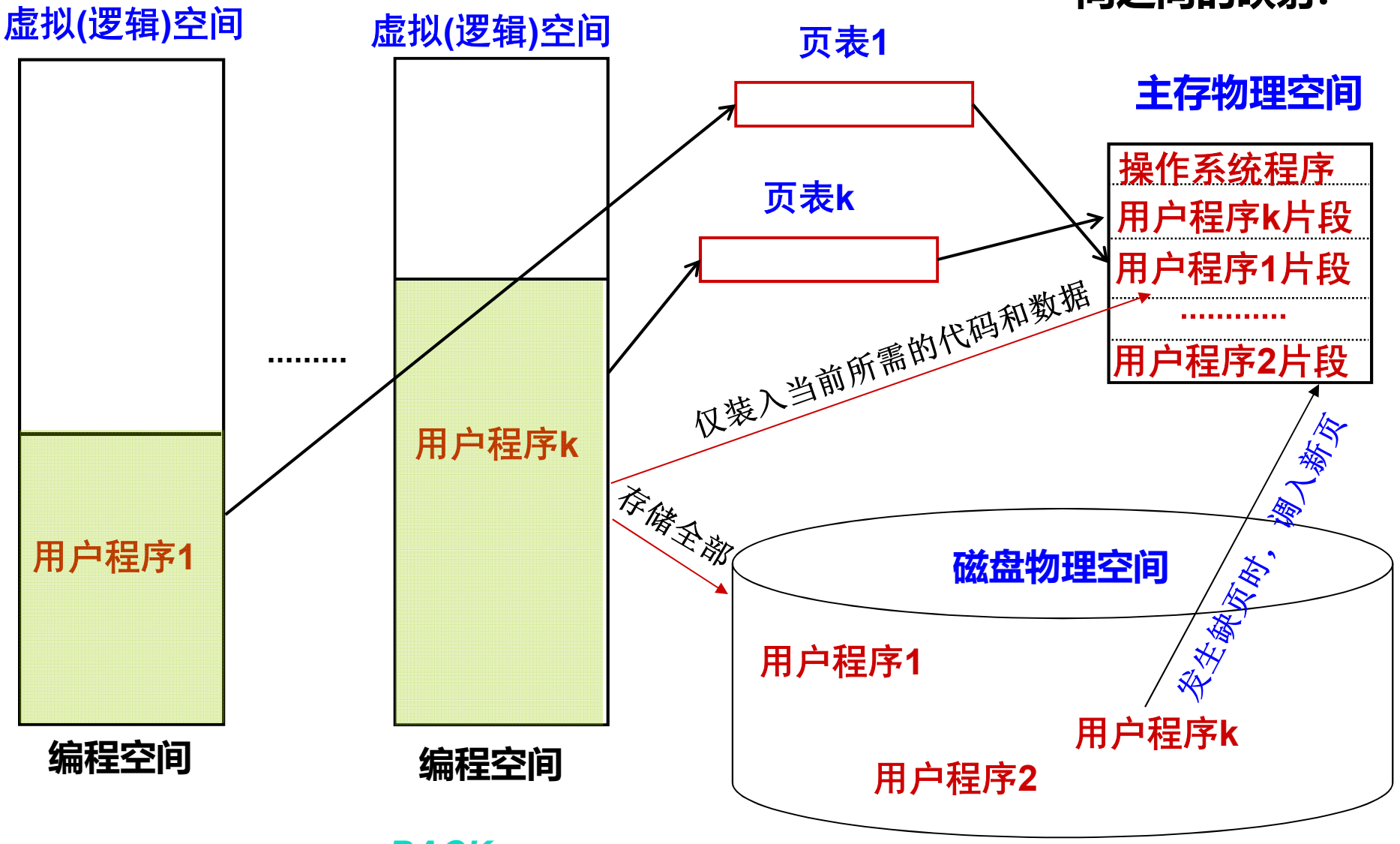
虚拟存储系统的基本概念

- 虚拟存储技术的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!



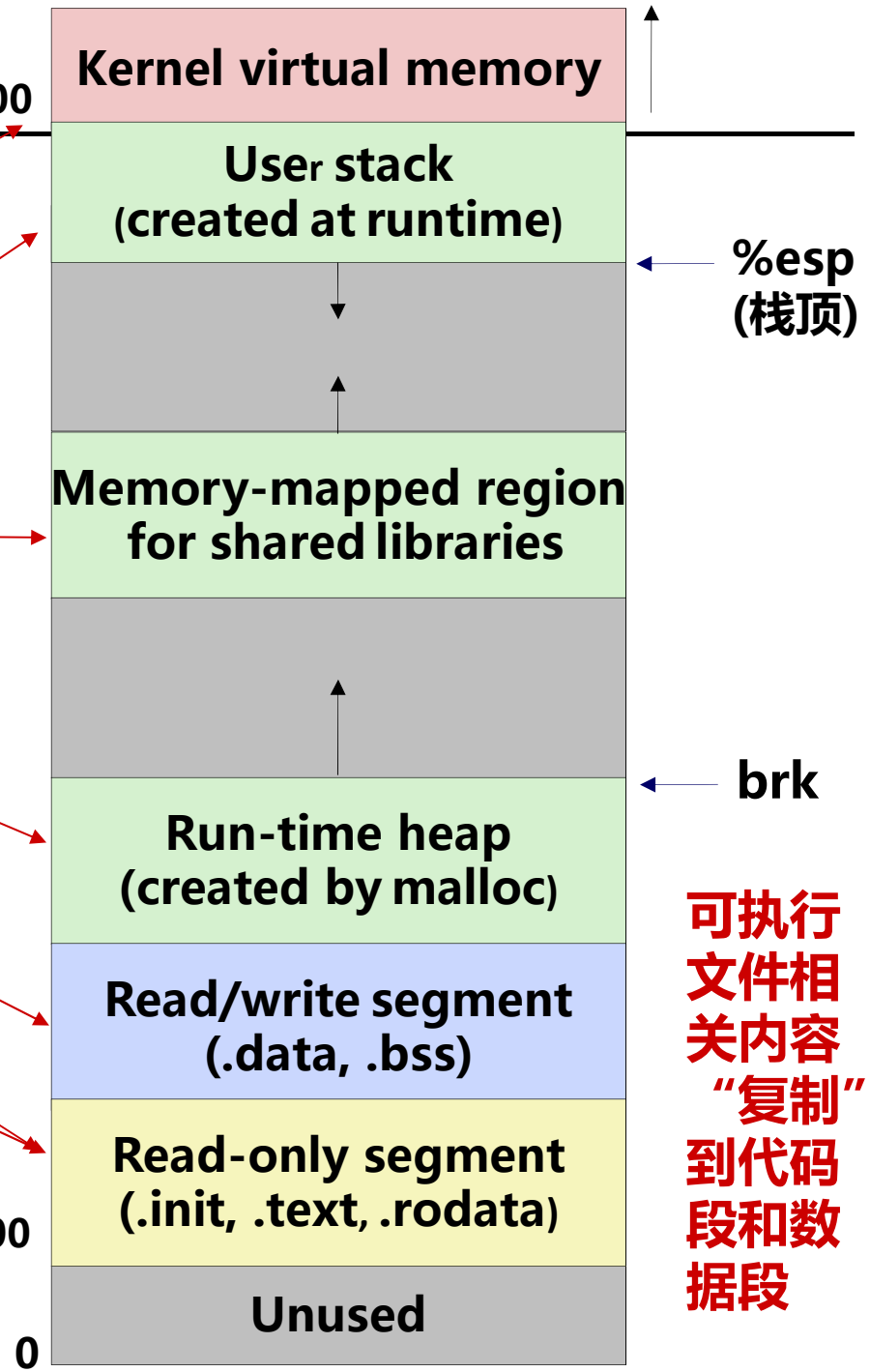
BACK

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（Read-only Data）
  - 代码（Code）

问题：加载时是否真正从磁盘调入信息到主存？

实际上不会从磁盘调入，只是将虚拟页和磁盘上的数据/代码建立对应关系，称为“映射”。



虚拟存储器管理

实现虚拟存储器管理，需考虑：

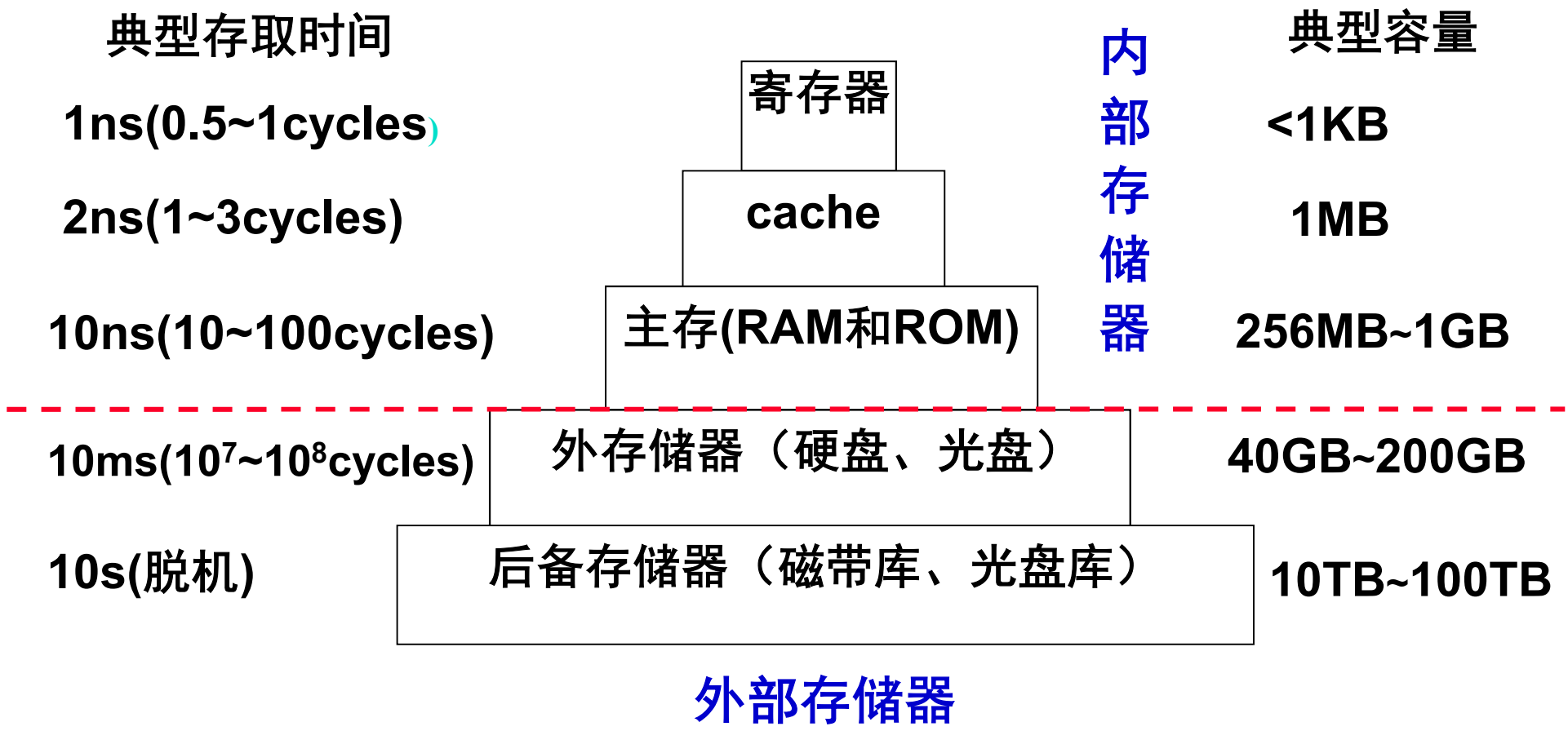
- **块大小（在虚拟存储器中“块”被称为“页 / Page”）应多大？**
- **主存与辅存的空间如何分区管理?**
- **程序块 / 存储块之间如何映像?**
- **逻辑地址和物理地址如何转换，转换速度如何提高?**
- **主存与辅存之间如何进行替换（与Cache所用策略相似）?**
- **页表如何实现，页表项中要记录哪些信息?**
- **如何加快访问页表的速度?**
- **如果要找的内容不在主存，怎么办?**
- **如何保护进程各自的存储区不被其他进程访问?**

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！

存储器的层次结构



列出的时间和容量会随时间变化，但数量级相对关系不变。

“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?

因为缺页的开销比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

页表结构

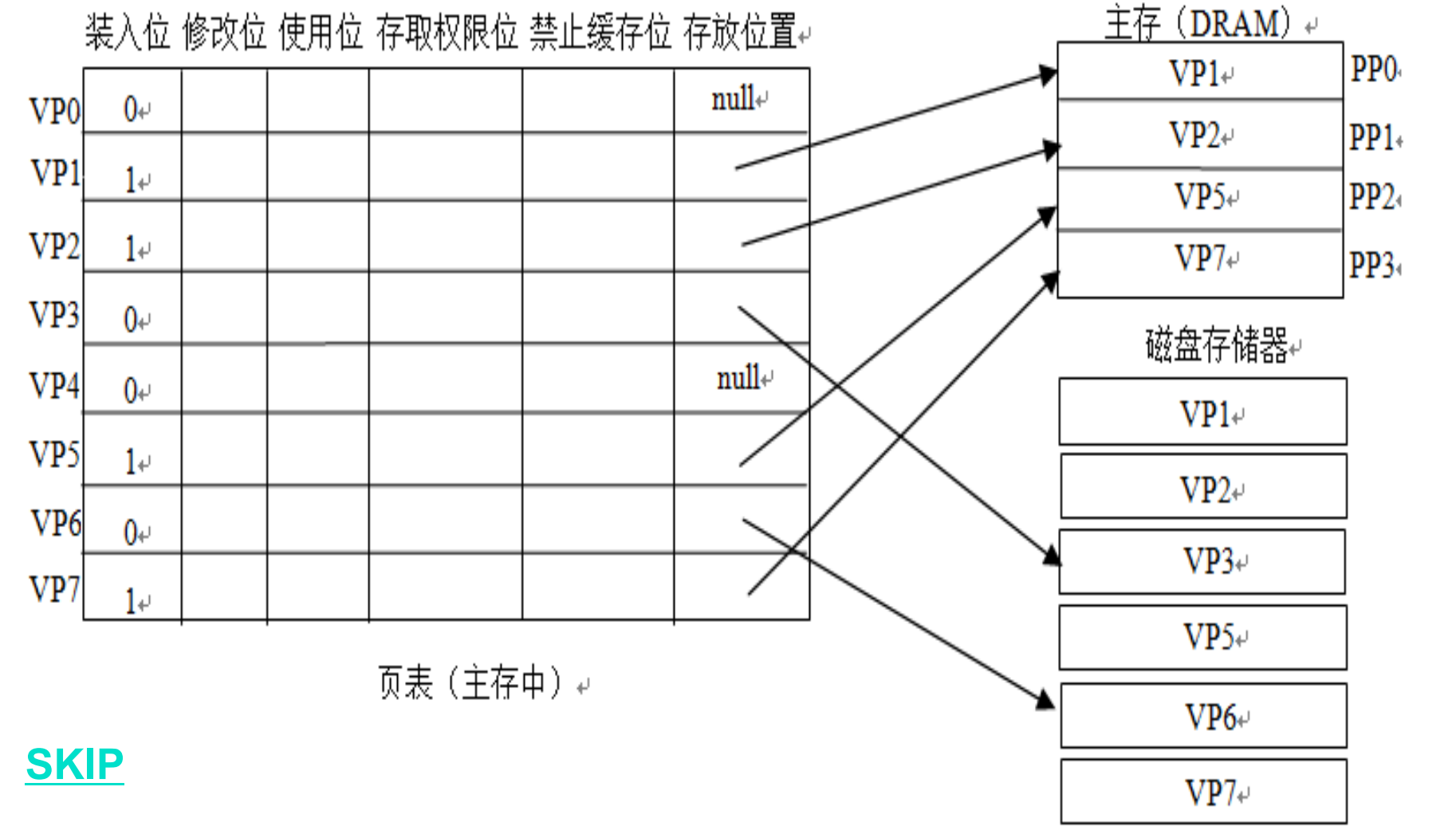
◆ 页表首址记录在页表基址寄存器中

页表首地址

	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（Dirt）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

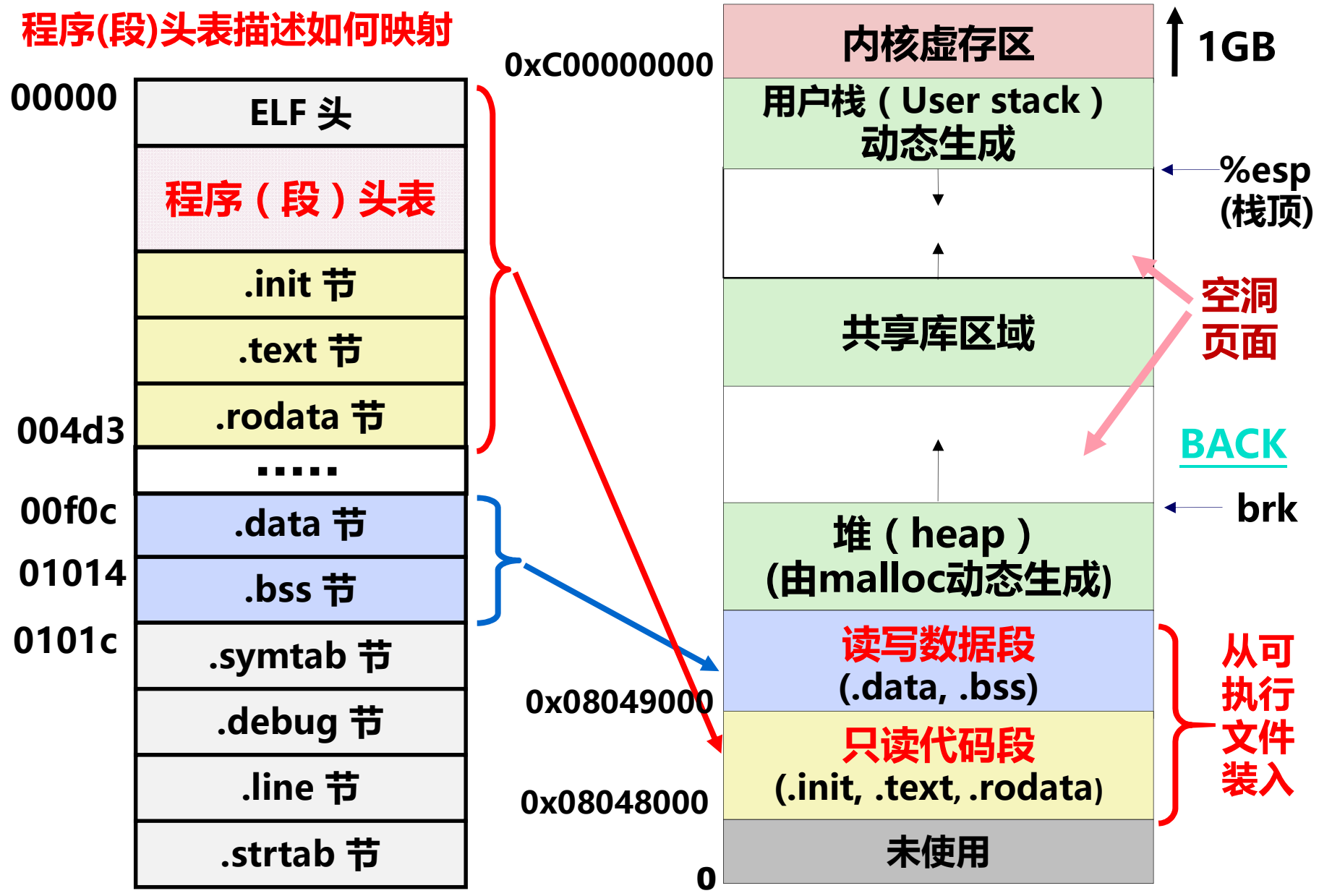
主存中的页表示例



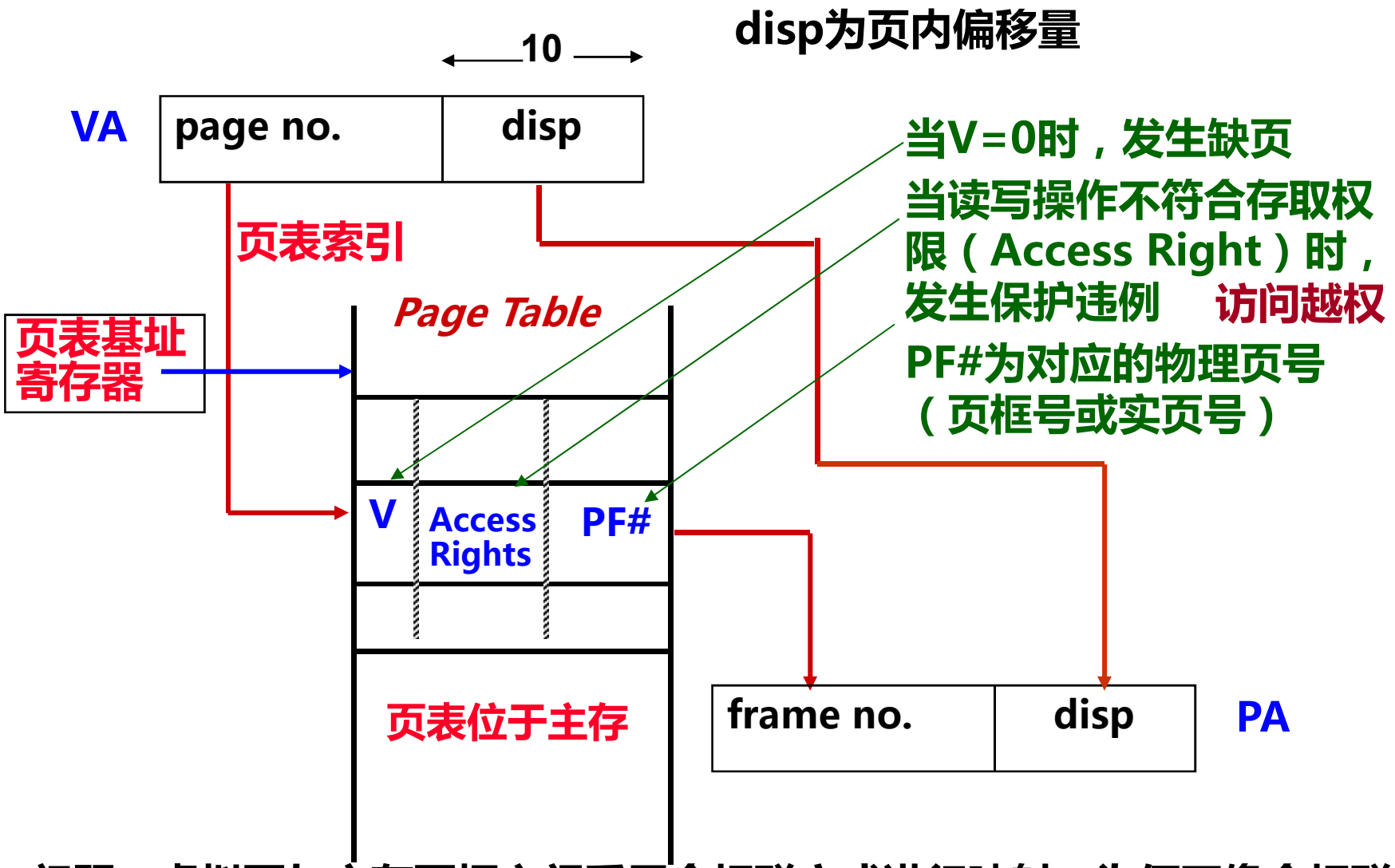
SKIP

- **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）

可执行文件的存储器映像



逻辑地址转换为物理地址的过程



问题：虚拟页与主存页框之间采用全相联方式进行映射，为何不像全相联Caceh那样（高位地址是Tag），而高位地址是索引呢？

信息访问中可能出现的异常情况

可能有两种异常情况：

1）缺页（ page fault）

产生条件：当Valid（有效位 / 装入位）为 0 时

相应处理：从磁盘读到内存，若内存没有空间，则还要从内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？

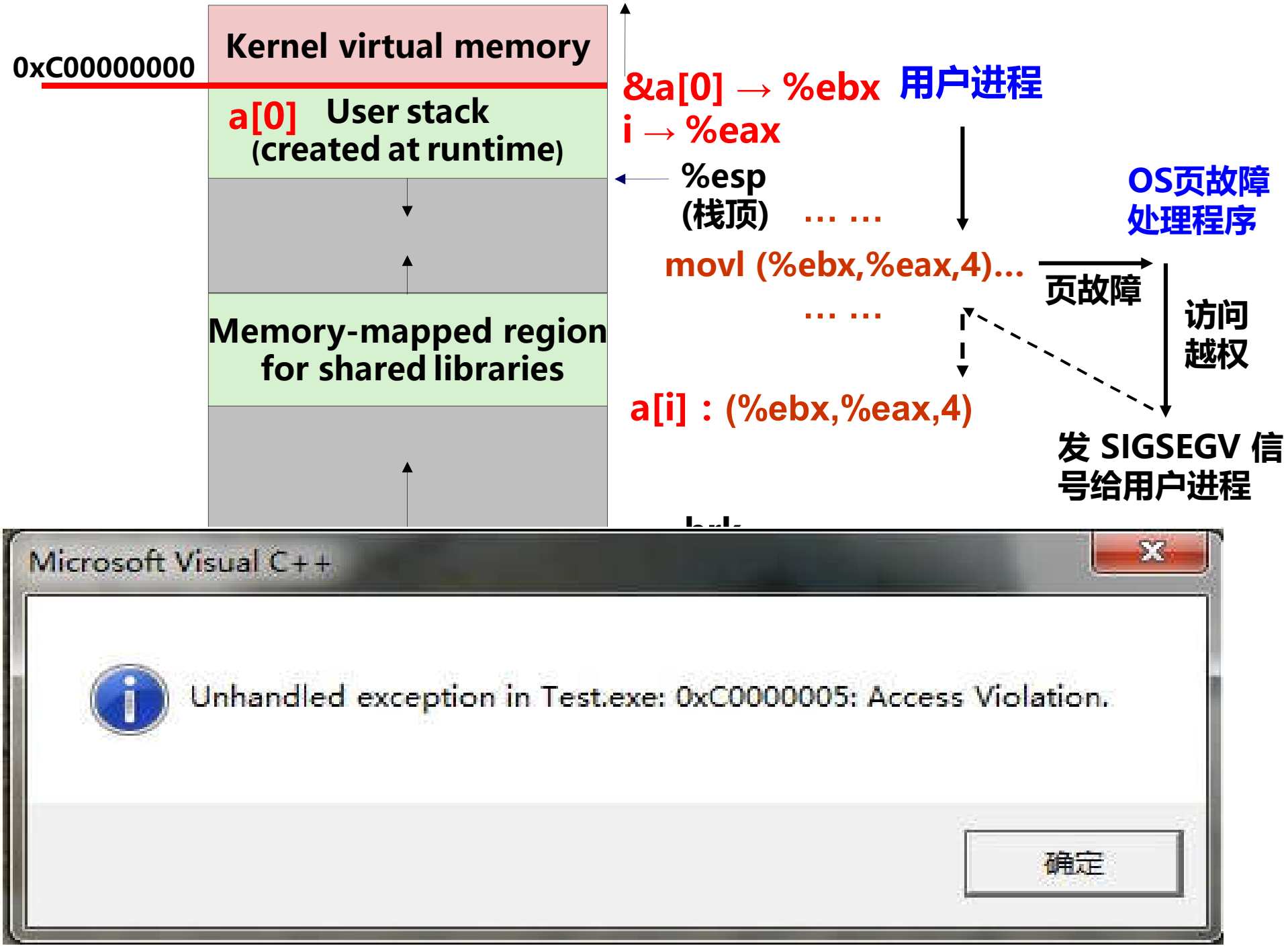
R = Read-only, R/W = read/write, X = execute only

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?


0xC00000000
Kernel virtual memory
a[0] User stack
(created at runtime)
Memory-mapped region
for shared libraries
&a[0] → %ebx 用户进程
i → %eax
%esp
(栈顶)
… …
movl (%ebx,%eax,4)…
… …
a[i] : (%ebx,%eax,4)
OS页故障
处理程序
页故障
访问
越权
发 SIGSEGV 信
号给用户进程
Microsoft Visual C++
Unhandled exception in Test.exe: 0xC0000005: Access Violation.
确定

TLBs --- Making Address Translation Fast

问题：一次存储器引用要访问几次主存？ 0 / 1 / 2 / 3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）*

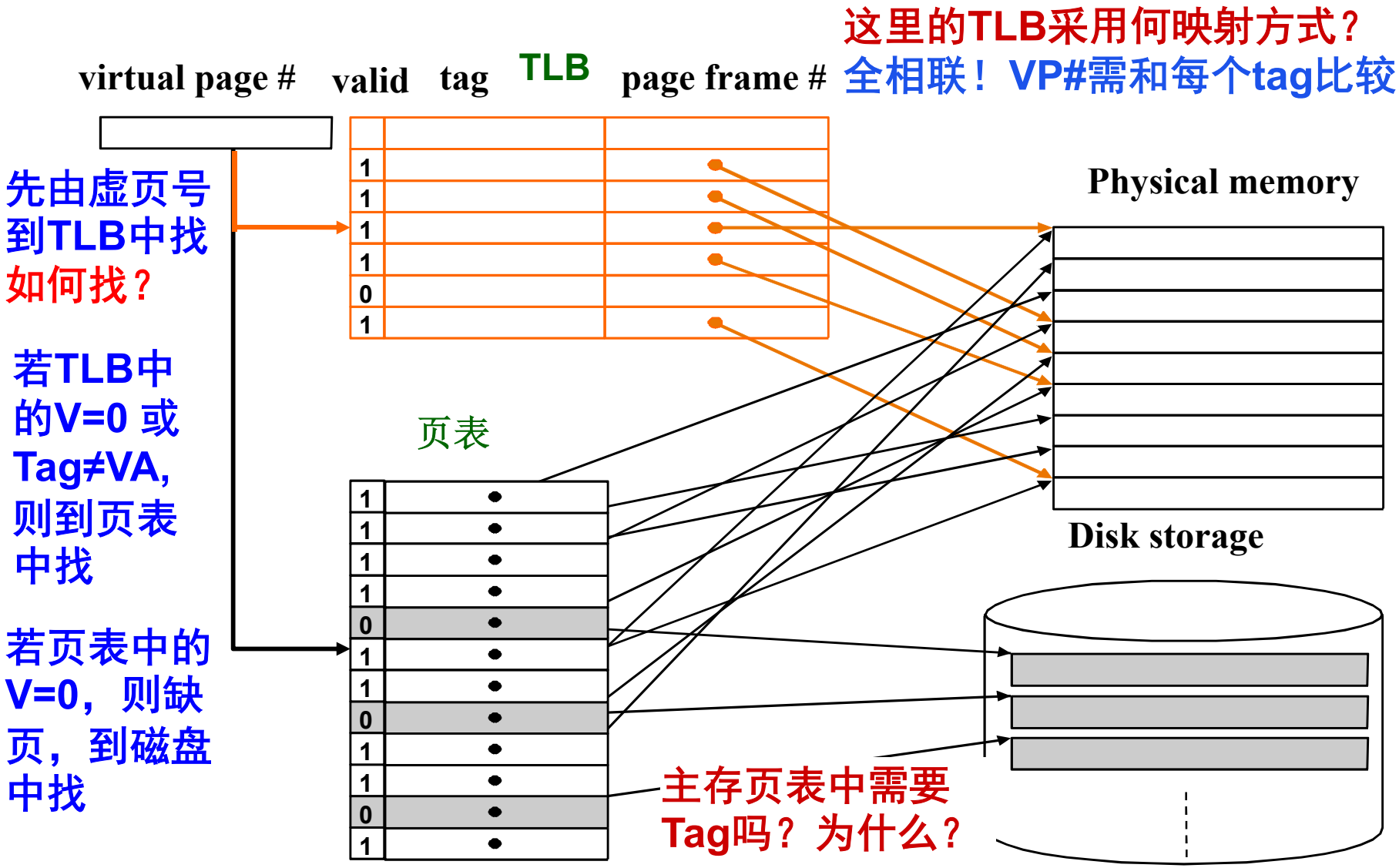
TLB中的页表项：tag+主存页表项

Virtual Address (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用于定位需要比较的表项

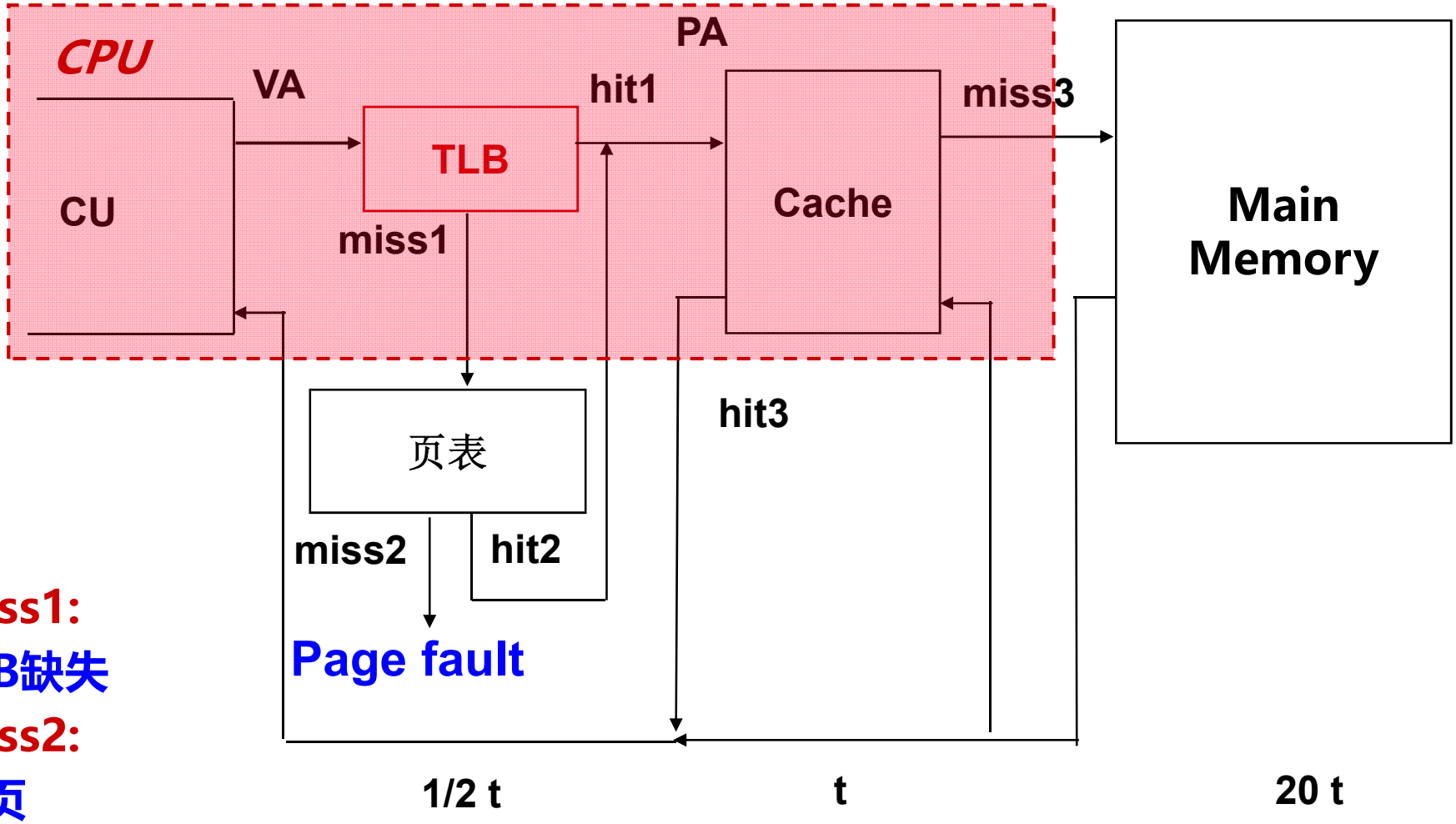
TLB全相联时，没有index，只有Tag，虚页号需与每个Tag比较；
TLB组相联时，则虚页号高位为Tag，低位为index，用作组索引。

TLBs --- Making Address Translation Fast



问题：引入TLB的目的是什么？ 减少到内存查页表的次数！

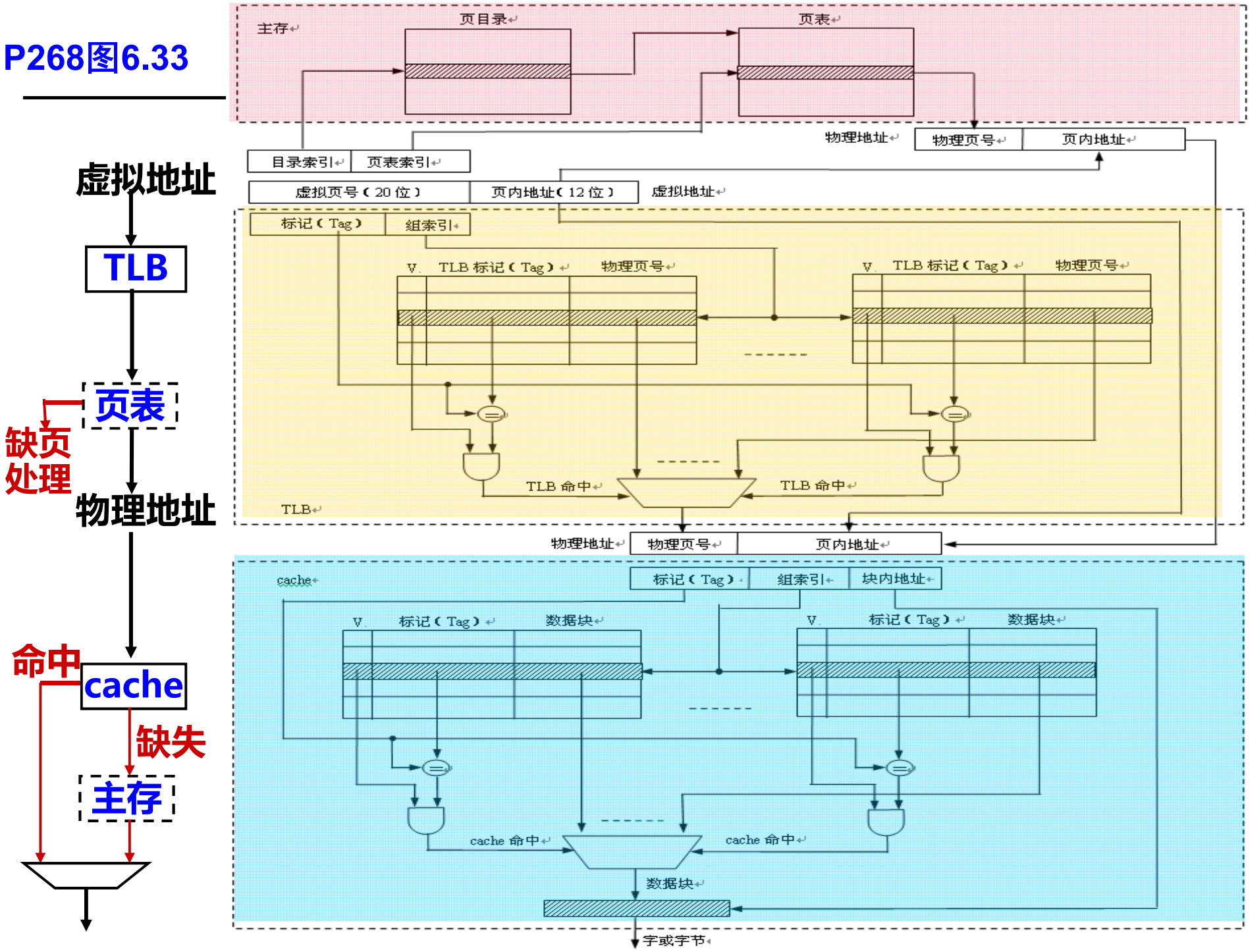
Translation Look-Aside Buffers



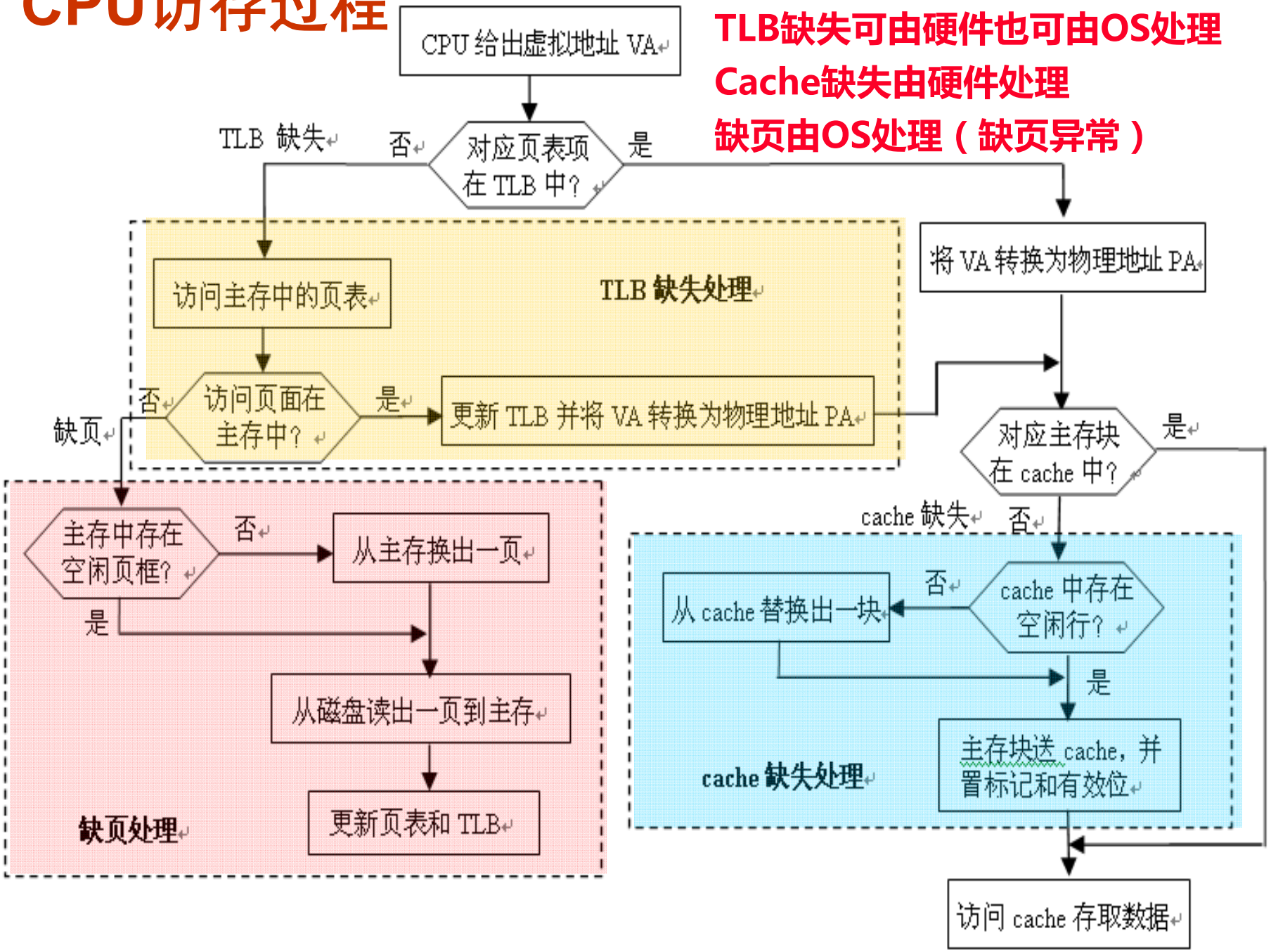
Miss1:
TLB缺失
Miss2:
缺页
Miss3:
PA 在主存中，但不在Cache中

TLB冲刷指令和Cache冲刷指令
都是操作系统使用的特权指令

P268图6.33


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

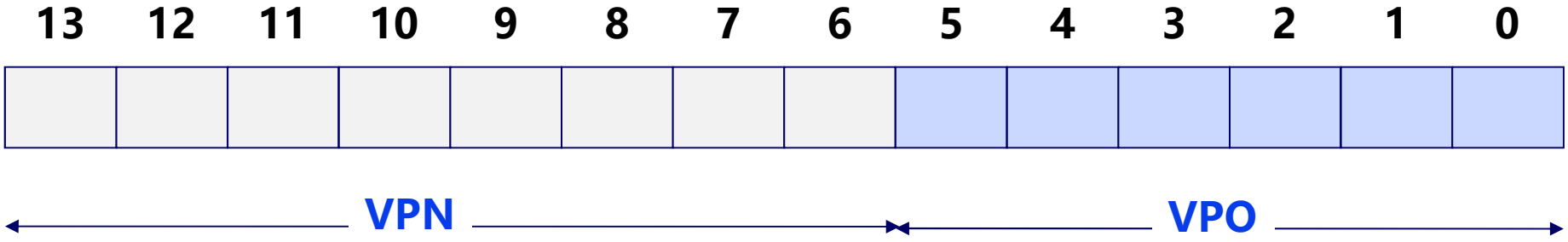
缩写的含义

- 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）

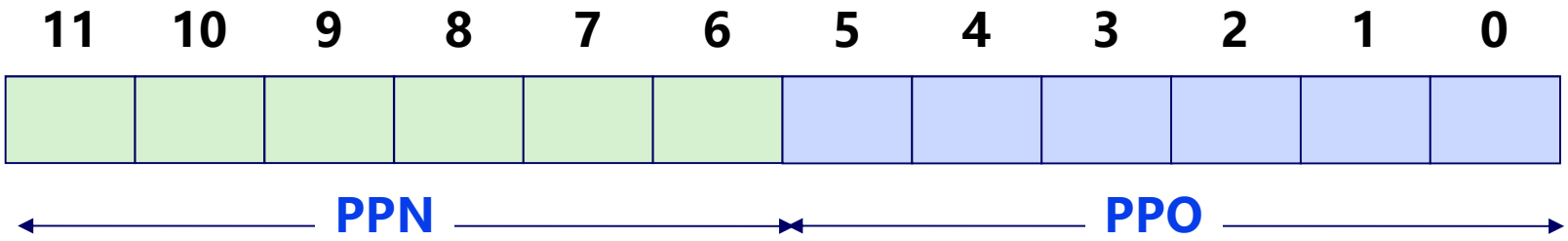
一个简化的存储系统举例

- 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：
$2^{14-6}=256$



Virtual Page Number　　Virtual Page Offset



Physical Page Number　　Physical Page Offset

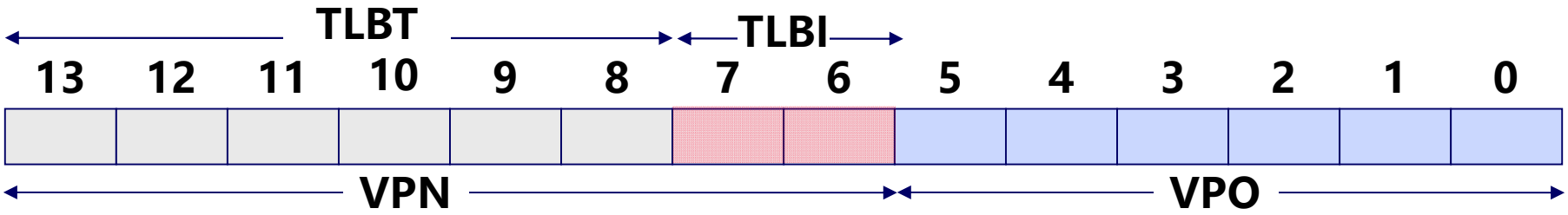
一个简化的存储系统举例（续）

假定部分页表项内容（十六进制表示）如右：

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

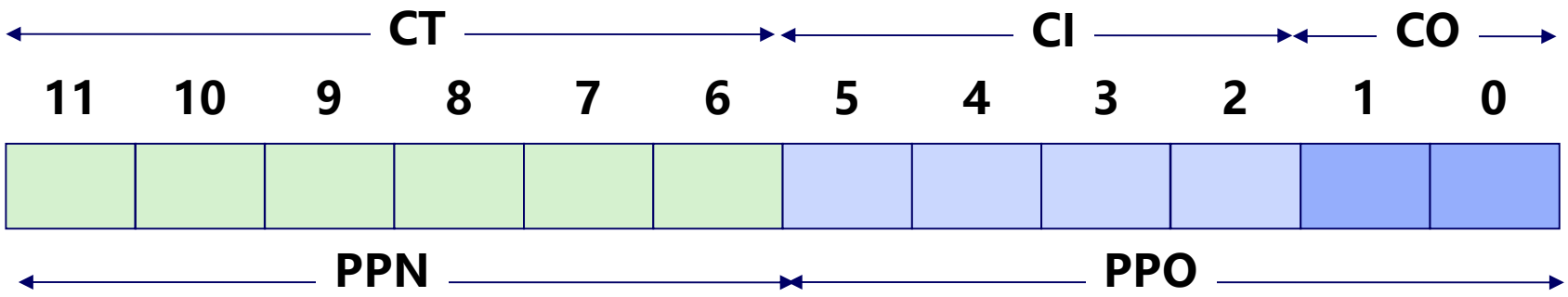
假定TLB如下：16 个TLB项，4路组相联，则TLBT和TLBI各占几位？



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

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？

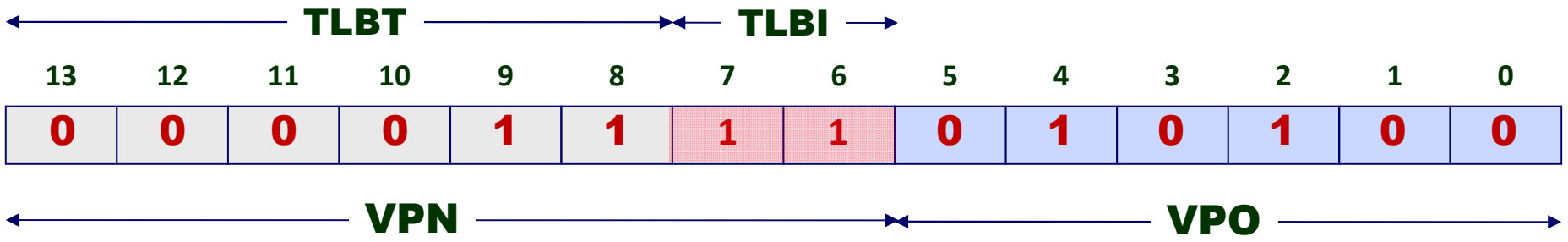


Idx	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

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）

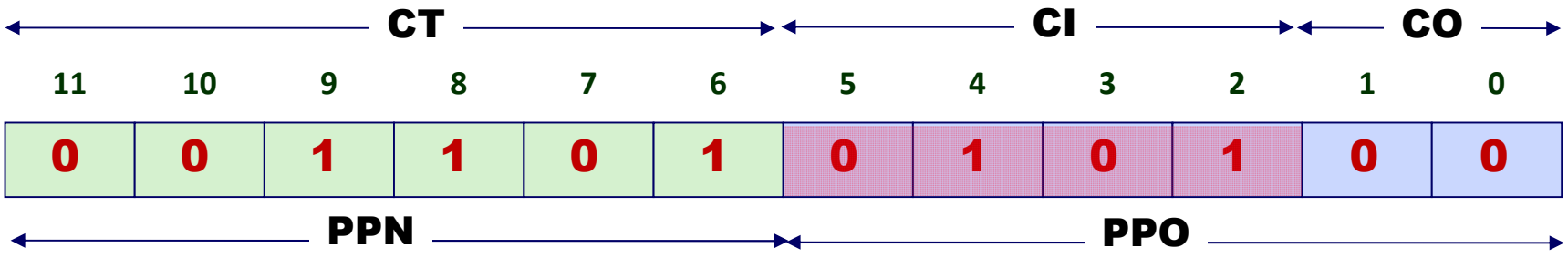
假设该存储系统所在计算机采用小端方式， CPU执行某指令过程中要求访问一个16位数据，给出的逻辑地址为0x03D4，说明访存过程。



VPN 0x0F TLBI 0x3 TLBT 0x03 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？
TLB缺失/cache缺失、TLB缺失/缺页



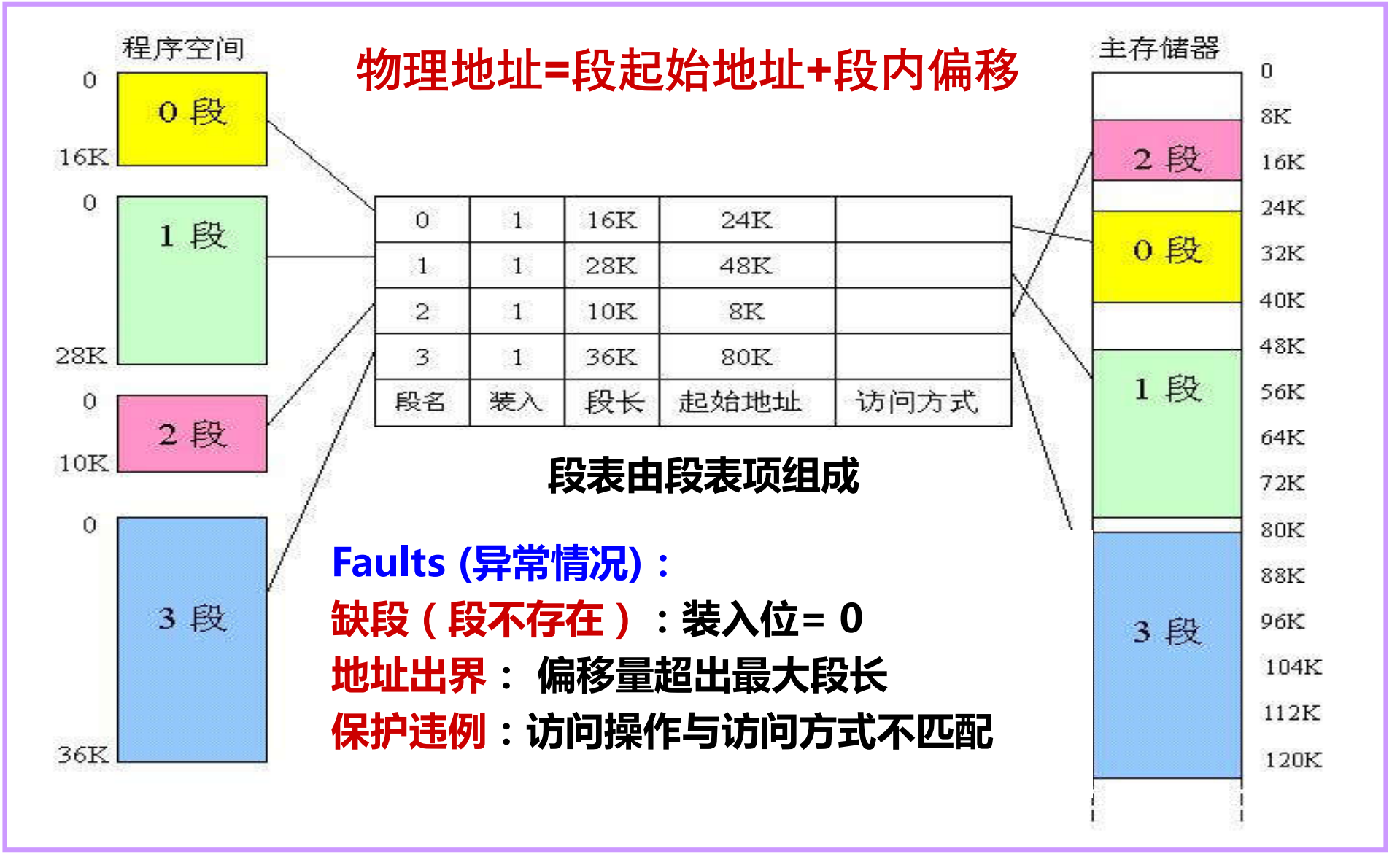
CO 0 CI 0x5 CT 0x0D cache Hit? Y 数据: 0x7236

分段式虚拟存储器

- 分段系统的实现
  - 程序员或OS将程序模块或数据模块分配给不同的主存段，一个大程序有多个代码段和多个数据段构成，是按照程序的逻辑结构划分而成的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）
  - 段通常带有段名或基地址，便于编写程序、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的一个可再定位段

段式虚拟存储器的地址映像



段页式存储器

- 段页式系统基本思想
  - 段、页式结合：
    - 程序的虚拟地址空间按模块分段、段内再分页，进入主存仍以页为基本单位
  - 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
  - 根据段地址到段表中查阅与该段相应的页表首地址，转向页表，然后根据页地址从页表中查到该页在主存中的页框地址，由此再访问到页内某数据

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