西安科技大学
XI' AN UNIVERSITY OF SCIENCE TECHNOLOGY

第五章 内存管理

刘晓建

2018年10月23日

本章概要

进程的内存管理问题是研究：如何把进程的结构加载到物理内存？为此需要回答这样几个问题：

- When and Who 执行内存管理？When=程序运行时？
- 一次把程序的全部代码载入内存还是部分代码？
- 程序在内存中是连续分布还是离散分布？
- 程序在内存中的位置是一成不变的，还是可以移动的？
- 进程间如何共享公共代码？
- 进程的生命周期=？进程在内存中的周期？

4.1 并发问题

内存管理的基本需求：

1、重定位的需求：一个程序在不同的运行实例中，操作系统为它分配的物理内存空间可以不同，甚至在一个程序运行期间，它所占据的物理内存空间也可以上下浮动。

2、共享的需求：一是指在多道程序设计中，多个进程共享同一物理内存，为此需要对内存空间进行合理有效的划分，使得每个进程独占一定的内存区域，而且这些区域不相互冲突。二是指允许多个进程访问同一内存区域（共享库/共享代码）。

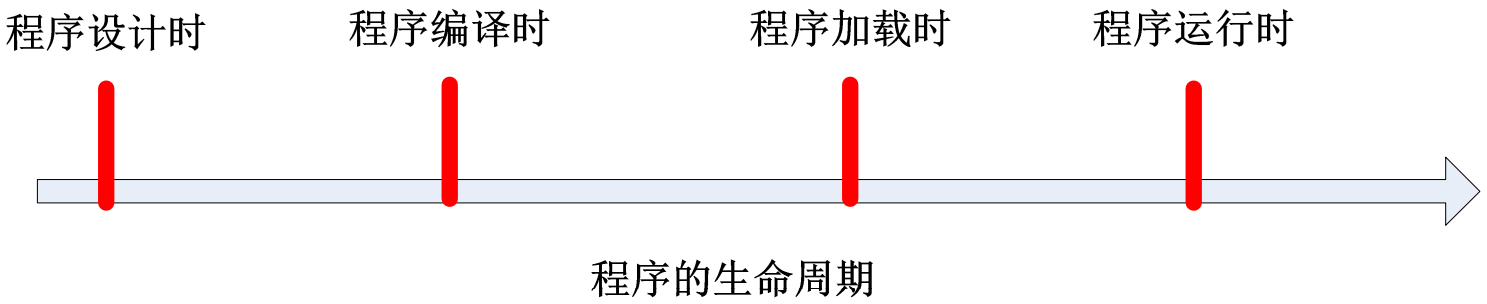
3、保护的需求：由于多个进程驻留在内存中，因此必须保护一个进程的内存空间不受其它进程有意或无意的干扰。一个进程不能未经授权的访问另一个进程的内存单元。

4、存储器扩充的需求：用有限的物理内存运行相对更多的程序。

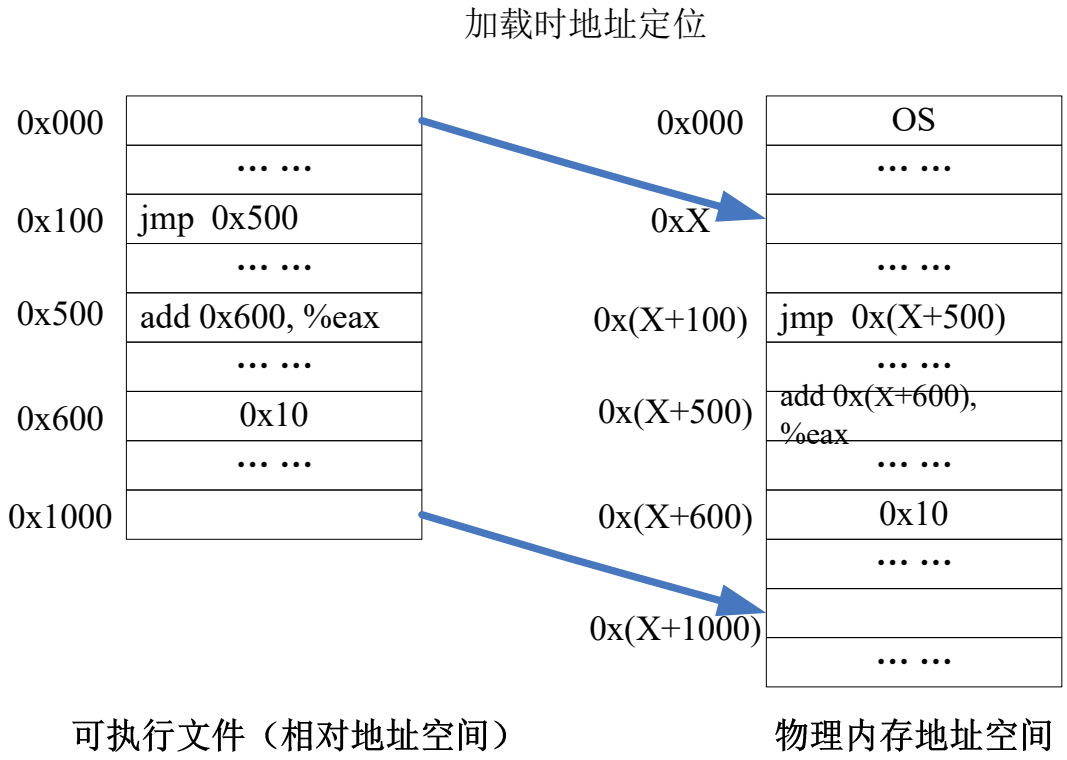
地址定位

所谓**地址定位**（或称为**地址绑定**，Address binding）是指：为了执行一个程序，必须确定程序指令或数据所在物理内存地址的过程。

在程序还没有加载之前，程序以文件的形式存在于磁盘空间中，程序中的指令和数据使用相对地址（或逻辑地址）来编址；当准备运行程序时，程序被加载到内存中，这时需要定位程序指令和数据的物理内存地址，即在物理内存中确定指令和数据的位置。因此，地址定位过程就是逻辑地址到物理地址的变换过程。



程序加载时地址定位，也称为**静态地址定位**，是由加载器（Loader）在加载程序过程中将程序指令和数据的物理地址确定下来。



程序中指令和数据的逻辑地址（用LA表示）以及指令所引用的逻辑地址都在加载时被定位到物理地址（用PA表示），地址定位的算法为：

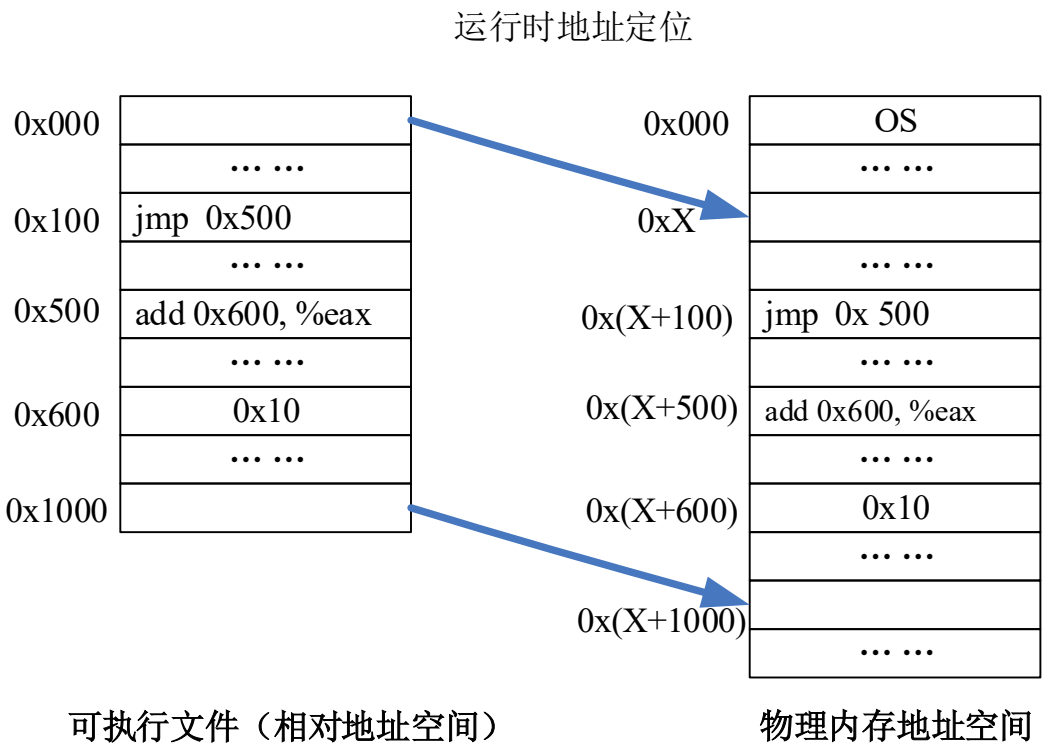
$$PA = LA + X$$

其中X是加载的起始内存地址。

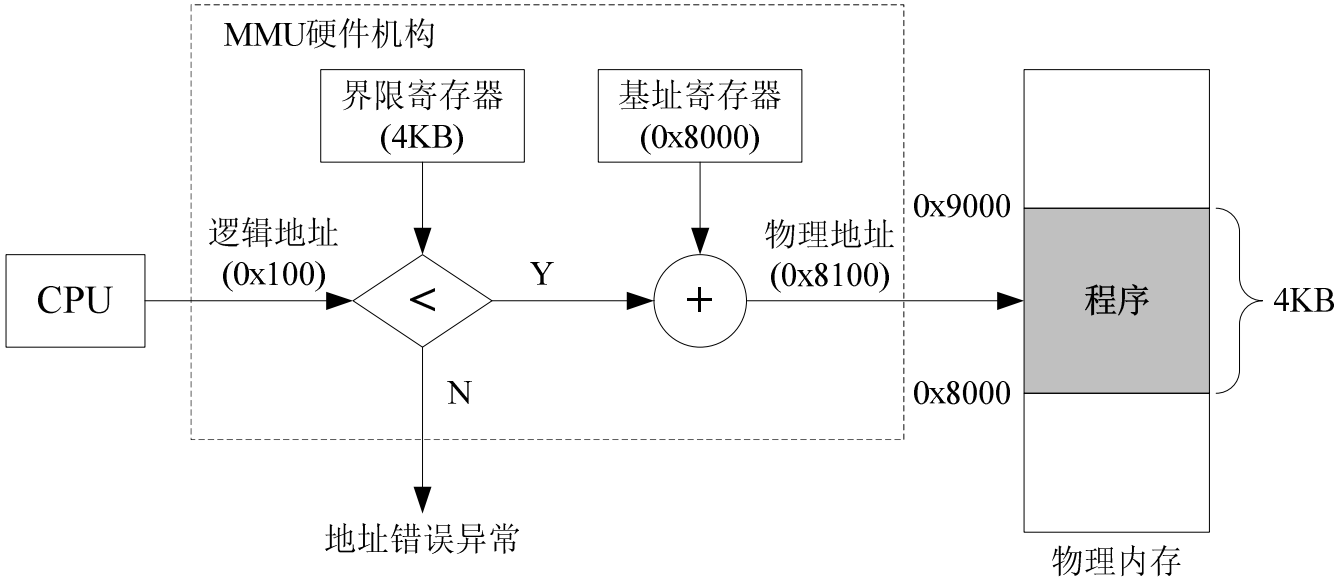
静态地址定位的优缺点:

- 优点是容易实现，无需硬件支持，它只要求程序本身是可重定位的，即对那些要修改的地址部分具有某种标识。早期的操作系统大多采用这种方法进行地址定位。
- 主要缺点是:
  - •程序经地址定位之后，在程序运行过程中就不能再移动了，因而不能重新分配内存，不利于内存的有效利用;
  - •程序在内存空间中只能连续分配，不能离散分布在内存的不同区域;
  - •多个用户很难共享内存中的同一程序，如果多个进程要共享同一程序，则必须使用自己的副本。

程序运行时地址定位，也称为**动态地址重定位**，把地址定位从加载时推迟到运行时。在加载时，仍然保持程序的相对地址不变，只有在访问一条指令或数据时，才计算它们的物理地址。



运行时地址定位使用一个硬件机构MMU（内存管理单元）来实现。基址寄存器记录程序在内存中的起始地址，界限寄存器记录程序的最大逻辑地址（即程序的字节大小）。当程序被加载入内存或当该进程被换入时，必须设置这两个寄存器。



运行时地址定位的算法为：

PA = LA + R[基址寄存器] 并且 LA< R[限界寄存器]

与静态地址定位相比，运行时地址定位的程序的起始地址在加载后并不是一成不变的，而是可以上下浮动。动态地址定位的优点是：

- 用户程序在内存中的位置可以上下浮动，这有利于实现进程地址空间的换入和换出策略，也有利于内存的充分利用；
- 程序不必连续存放在内存中，可以分散在内存中的不同区域，这只需增加几对基址-界限寄存器，每对寄存器对应一个区域；
- 多个用户可以共享同一程序。

需要注意的是，动态地址定位发生在运行时，因此不可能由操作系统来完成地址定位，只能由CPU的MMU硬件机构来完成地址变换。由于在运行时使用附加的硬件进行地址映射，因此处理器指令循环的一部分周期花费在了地址映射上，在一定程度上影响了程序的执行效率。

5.2.1 固定分区管理

a)大小相等的分区

b)大小不等的分区

当一个进程请求内存区时，有以下三种分配方法：

•**最小适配**：在所有满足进程内存需求的分区中选择一个最小的分区来分配，即

$\min\{x \mid x \geqslant R\}$，其中$x$是分区的大小，$R$是进程的内存需求大小。

•**首次适配**：即按照某种顺序，找到的第一个满足进程内存需求的分区分配给该进程。

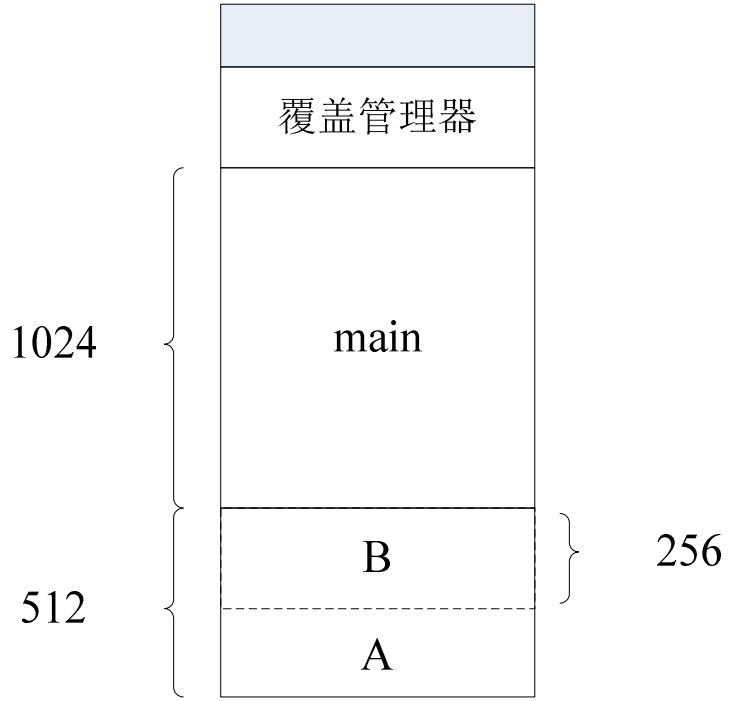
•**最大适配**：在所有满足进程内存需求的分区中选择一个最大的分区来分配，即

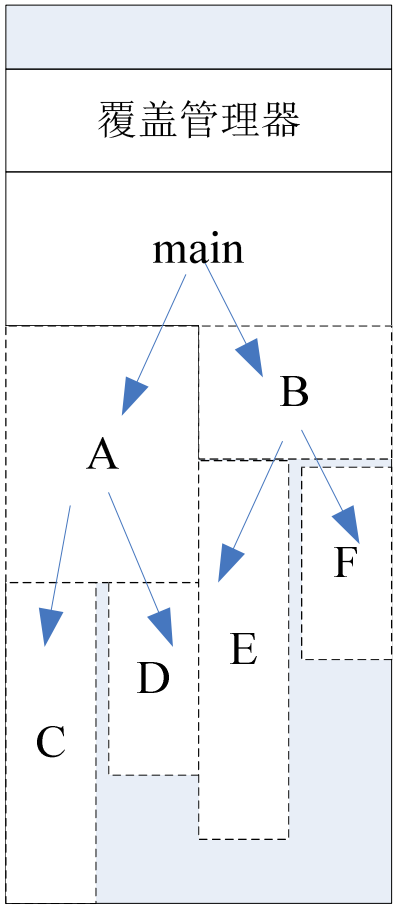
$\max\{x \mid x \geqslant R\}$。

5.2.2 覆盖技术

程序大小超过最大分区时，不能放到一个分区中。这种情况下，程序员必须使用覆盖技术（Overlaying）设计程序，使得任何时候只有一部分程序驻留内存中，而且不影响程序的运行。

一个程序有主模块main，它分别会调用模块A和模块B，但是A和B之间不会相互调用。假定这三个模块的大小分别是1024字节、512字节和256字节，在理论上它们需要占用1792个字节的内存。由于模块A和B相互不调用，因此它们可以相互覆盖，共享同一块内存区域，使得实际内存占用减小到1536字节，





覆盖技术对于程序员是不透明的，也就是覆盖是通过程序员编写代码来实现的。程序员在编写程序时，必须手工将程序分成若干块，然后编写一个小的辅助代码来管理这些模块何时应该驻留在内存、何时应该被替换掉。这个小的辅助代码就是所谓的**覆盖管理器**（Overlay Manager）。

优点：变完全加载为部分加载

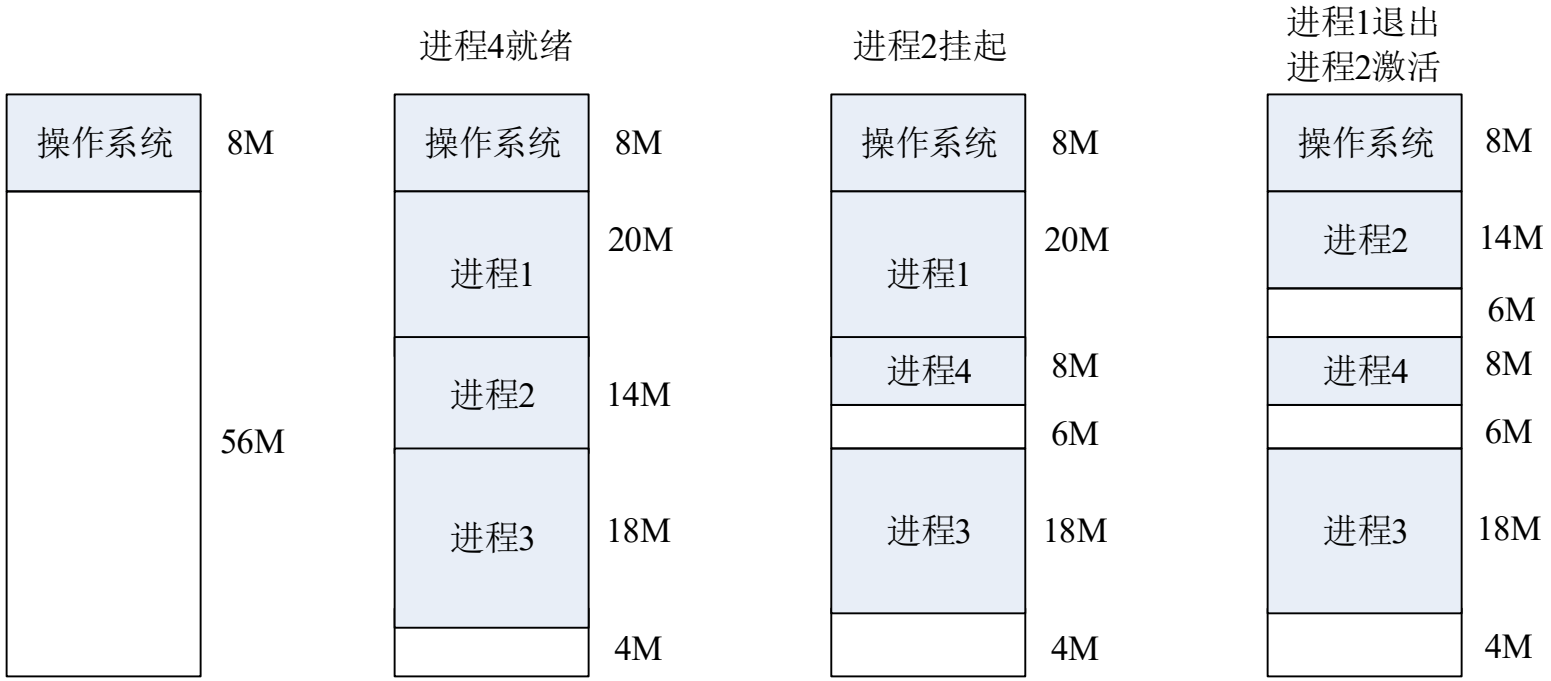
缺点：（1）内存管理对程序员是透明的，因此增加了编写程序的复杂性；

(2)如果一个模块在内存中，那么从该模块开始的整个调用路径上的模块都必须在内存中；

（3）分支树上的模块之间不能存在相互调用。有时，模块之间的调用关系并非树状结构，因此覆盖技术就失去了前提。

5.2.3 可变分区管理

可变分区管理的分区长度和分区数目都是不固定的。当进程被载入内存时，系统会给它分配一块和它所需内存大小完全相等的内存空间。



容易造成内存碎片

克服内存碎片的一个办法是将进程占据的内存上下浮动，使得若干细小、零散的碎片被整合成一整块大的内存，以便用于其它进程的内存分配，但是要这样做需要两个机制的支持：

- 一是动态地址重定位，
- 二是内存拷贝。

在进程的生命周期内，其内存空间要上下浮动，如果采用静态地址定位，当程序和数据部分浮动之后，内存引用就会出现错位。另外，内存拷贝很浪费处理器时间，效率不高。

分区管理具有下面几个特点：

•利于共享。分区管理能够实现多个进程共享同一内存空间，而且便于实现进程的保护。

•全部加载。一个进程的地址空间要么被全部交换到内存空间，要么被全部交换出内存空间。使用分区管理无法实现真正意义上的存储器扩充。

•连续分配。一个进程需要被映射到连续的内存区域。无论是静态还是动态地址定位，都以连续内存分配为前提。如果进程的不同部分被映射到离散的内存区域，那么逻辑地址变换和进程保护就将变得非常复杂。

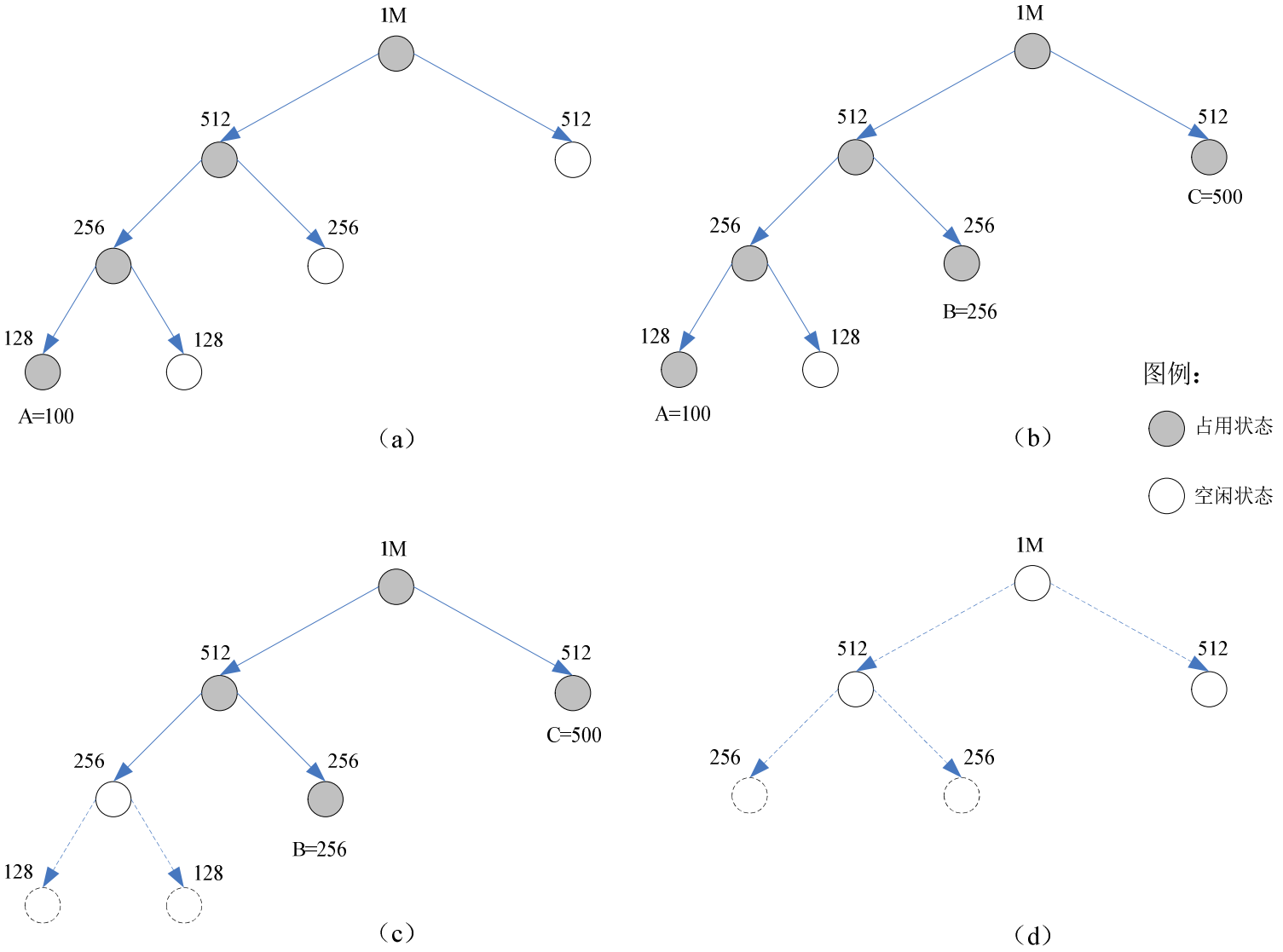
5.2.4 伙伴系统

伙伴系统是固定分区和可变分区的一种折衷方案。

在伙伴系统中，可用内存块的大小为2^K字节，$L \leqslant K \leqslant U$，其中，$2^L$表示最小内存块的大小，$2^U$表示最大内存块的大小。通常$2^U$是可供分配的整个内存的大小。

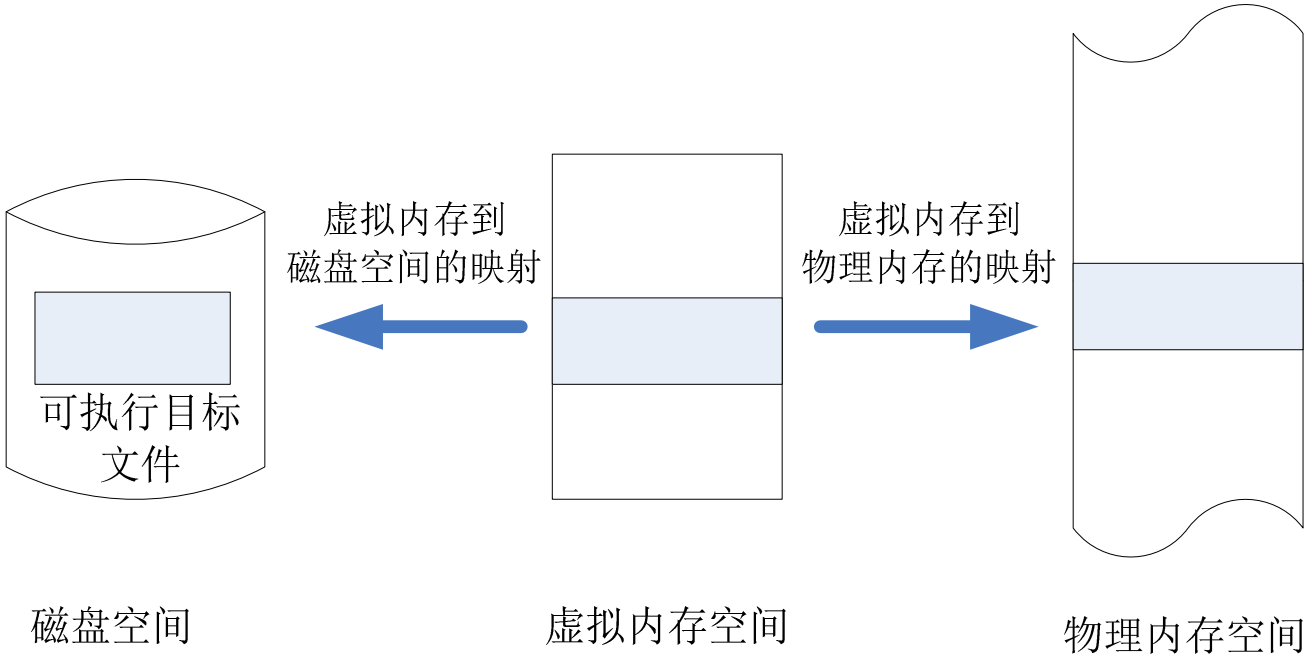
开始时，可用于分配的整个空间被看作一个大小为2^U的块。如果请求的大小s满足$2^{U-1} < s \leqslant 2^U$，则分配整个空间。否则该块被分成两个大小相等的伙伴，大小均为2^{U-1}。如果$2^{U-2} < s \leqslant 2^{U-1}$，则分配两个伙伴中的任何一个；否则，其中的一个伙伴又被分成大小相等的两半。这个过程一直继续下去，直到产生大于或等于s的最小块，并分配给该请求。

【例5-1】设内存大小为1M，初始时未分配。第一个进程A需要100KB，第二个进程B需要256KB，第三个进程C需要500KB。这些进程按照A、B、C的顺序依次进入并退出系统。伙伴系统管理内存的过程如下：



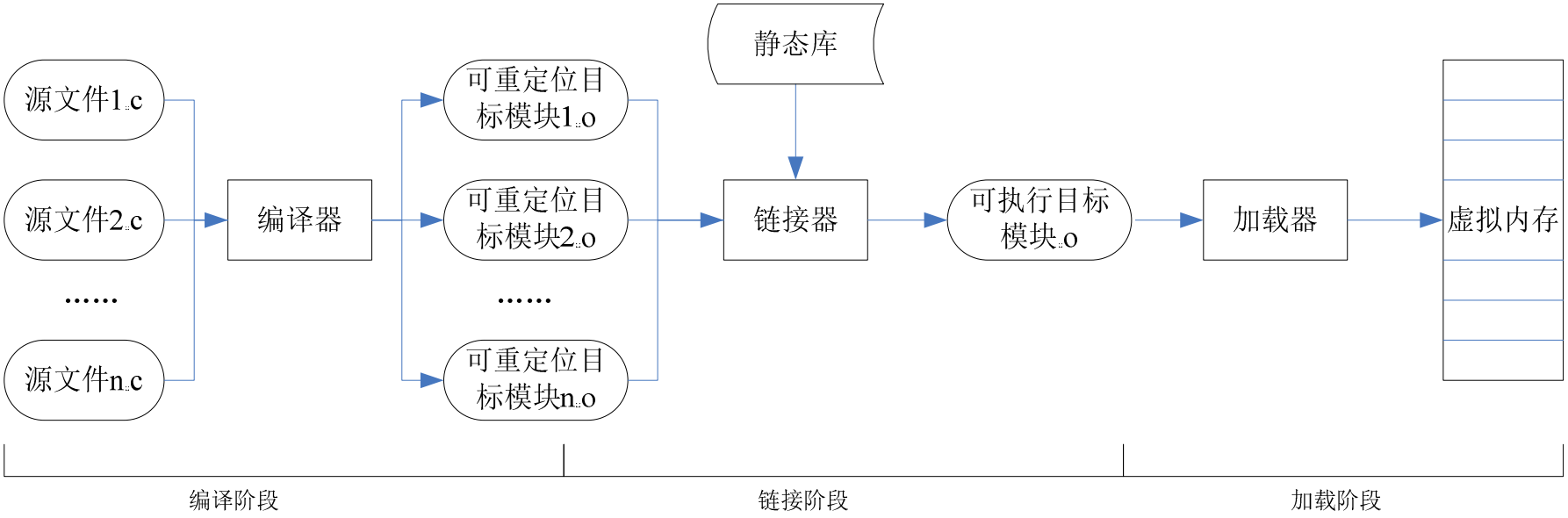
5.3 虚拟内存

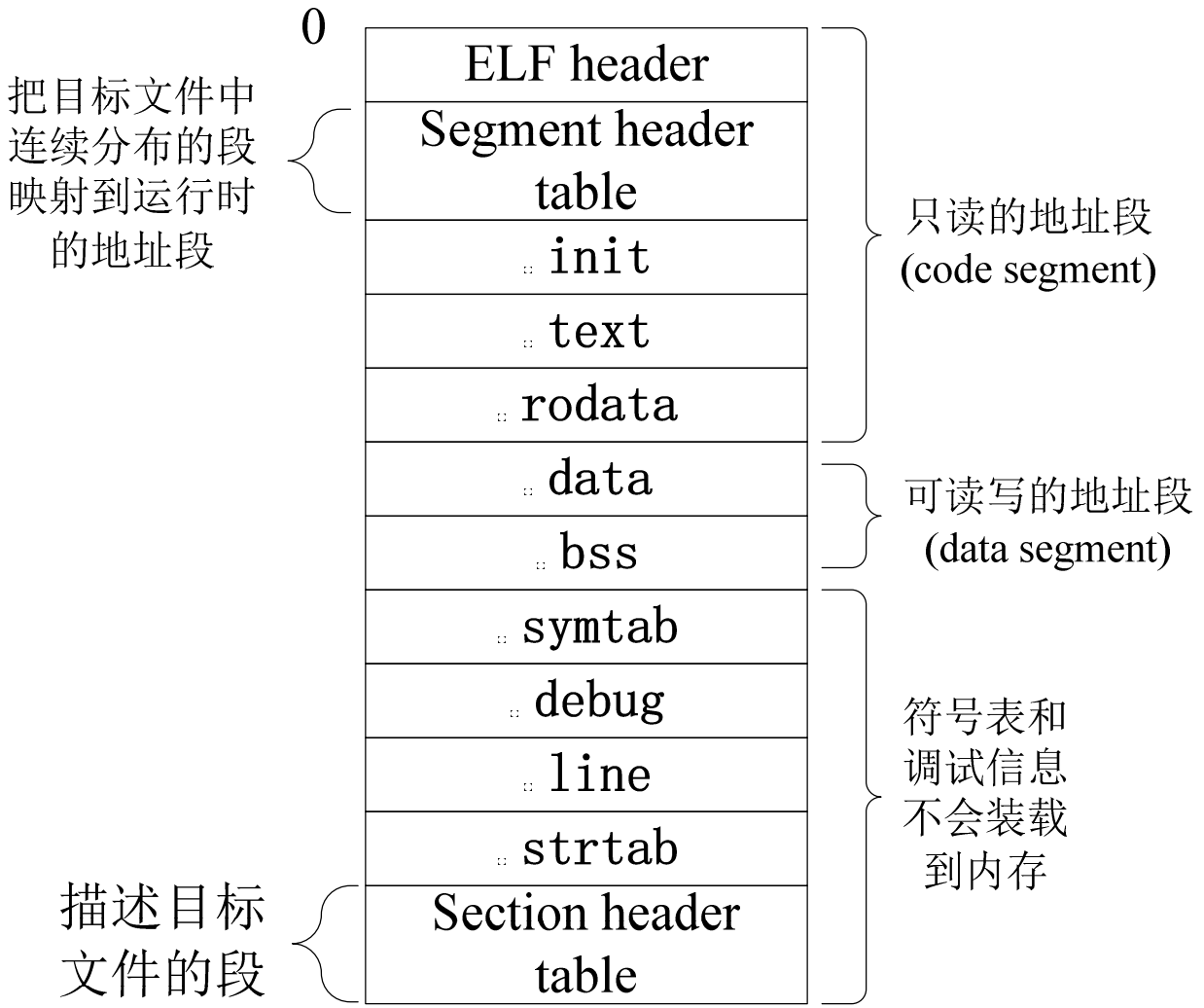
虚拟内存是对进程结构进行内存管理的一个中间数据结构。基于虚拟内存的内存管理并不是直接将可执行目标文件映射到物理内存中，而是通过虚拟内存间接建立可执行目标文件与物理内存的映射关系。



5.3.1 可执行目标文件

使用一种程序设计语言编写完程序之后，通常需要进行编译、链接两个阶段，最终产生可执行目标文件。




0
ELF header
Segment header table
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
Section header table
把目标文件中连续分布的段映射到运行时的地址段
描述目标文件的段
只读的地址段 (code segment)
可读写的地址段 (data segment)
符号表和调试信息不会装载到内存

5.3.2 虚拟地址空间

【定义1】（**地址空间**）地址空间是一个非负整数地址的有序集合：

$$\{0, 1, 2, \cdots, N-1\}$$

如果地址空间中的元素是连续的，那么我们称该地址空间为**线性地址空间**。在线性地址空间{0,1,2,…,N-1}中，N称为地址空间的大小。

地址空间是一个抽象的数学概念，使用它可以编码任何字节内容，比如物理内存、处理器寻址范围、文件内容以及进程结构等。我们把编码M个字节的物理内存的地址空间称为**物理地址空间**（PAS，Physical Address Space）：

$$\{0, 1, 2, \cdots, M-1\}$$

把编码处理器寻址范围的地址空间称为**虚拟地址空间**（VAS，Virtual Address Space）：

$$\{0, 1, 2, \cdots, N-1\}$$

其中$N=2^n$，n表示处理器所支持的最大地址的位数，通常取32或64。

5.3.2 虚拟地址空间

虚拟地址空间中的每一个地址称为**虚拟地址**。虚拟地址空间的大小取决于**处理器以及计算机硬件体系结构的设计**，表示了**处理器能够寻址的字节范围**，与**物理内存的大小无关**。

地址空间概念的引入使得**数据对象**本身或数据对象的内容，与它们的**地址**分离开来，地址可以看作是数据对象的一个属性。意识到这一点，就可以让**一个数据对象具有多个独立的地址，每个地址来自不同的地址空间**。一个数据对象在不同地址空间中的地址可能不同，但是这些地址都指向同一个数据对象。

三个空间：

- **磁盘地址空间**：保存数据对象的磁盘存储介质的地址空间
- **虚拟地址空间**：处理器能够寻址/编址的地址空间
- **物理地址空间**：物理内存中所有字节物理地址构成的地址空间

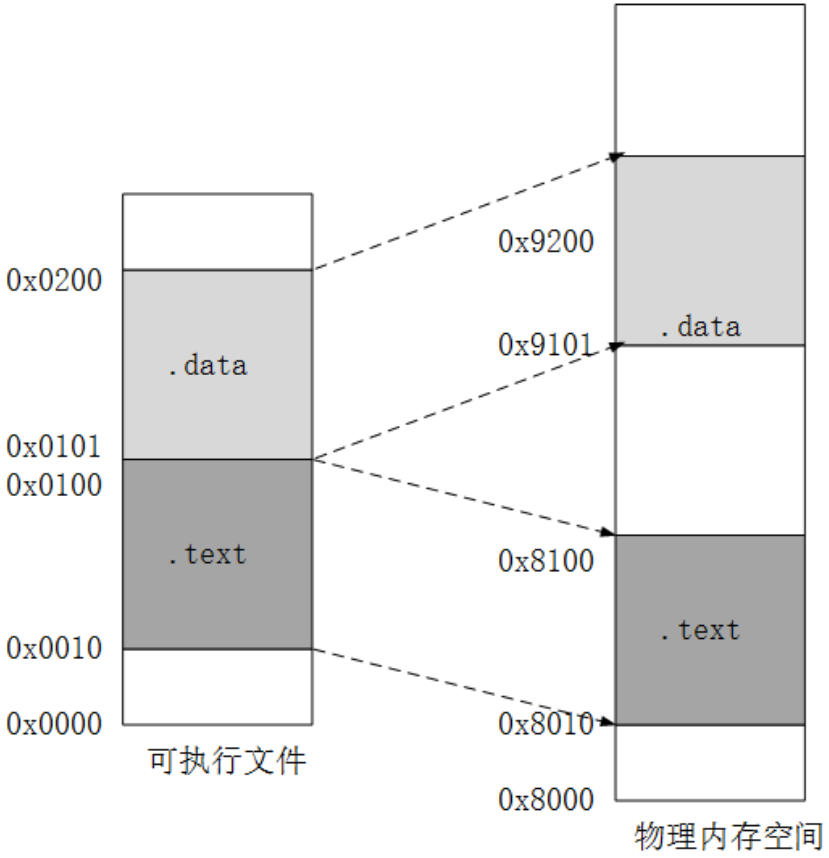
【例5-2】设一个可执行目标文件中

- 代码段：.text的地址范围是0x10~0x100，
- 数据段：.data的地址范围是

0x101~0x200。

把它们分别加载到地址从0x8010到0x8100和0x9101~0x9200的物理内存中。

把.text和.data从文件加载到内存的过程，可以看作是把数据对象从文件地址空间映射到物理地址空间的过程。

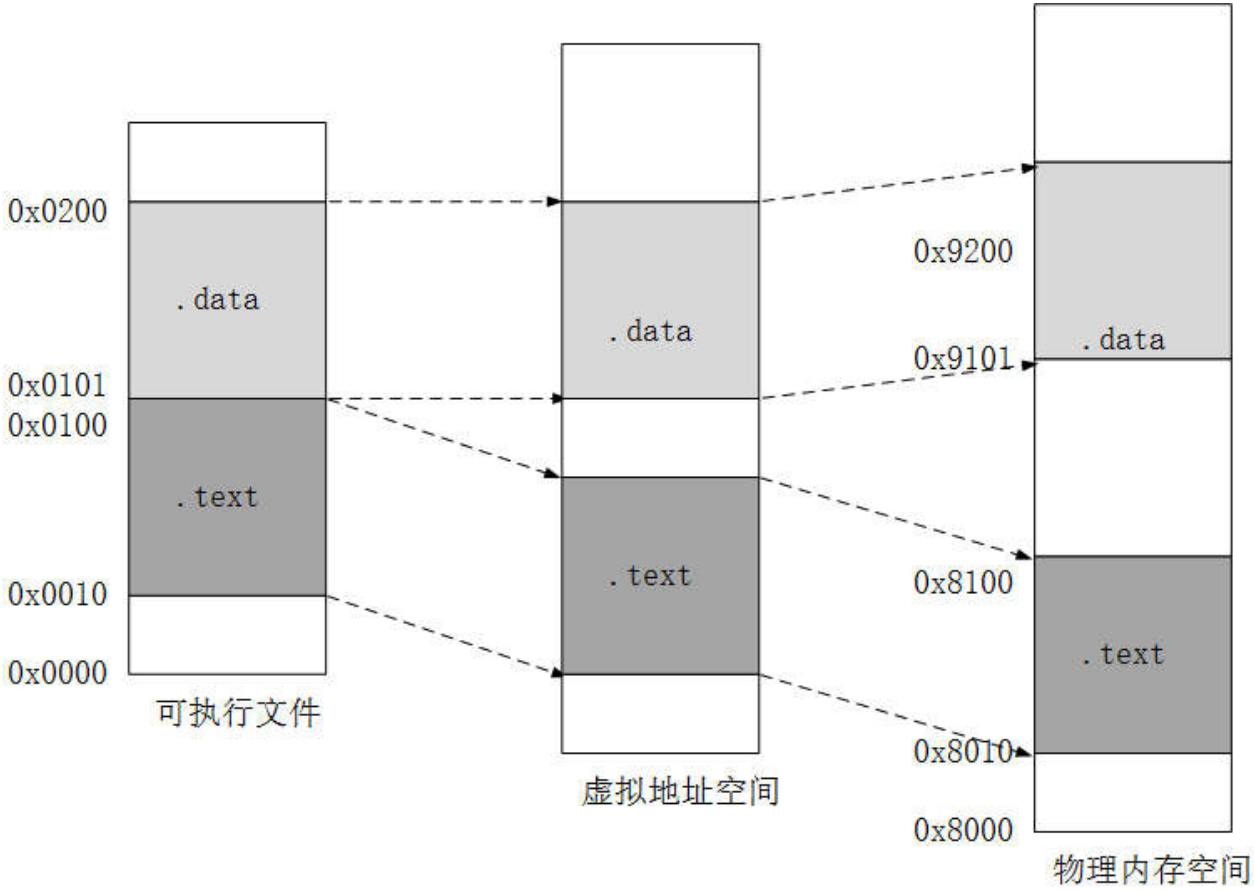


5.3.3 虚拟内存

【定义2】（**虚拟内存**）（VM，Virtual Memory）虚拟内存是由N个连续的字节存储单元构成的一个数组，每个字节具有一个唯一的虚拟地址作为其索引，N是虚拟内存的大小。

虚拟内存与**虚拟地址空间**的关系是：

虚拟内存在概念上是一个由连续字节存储单元构成的数组，而虚拟地址空间是虚拟内存中每一个字节的虚拟地址的集合；虚拟内存的大小和虚拟地址空间的大小是相同的。比如，对于一个32位计算机系统，其虚拟内存和虚拟地址空间的大小都是4G。由于虚拟内存和虚拟地址空间具有一一对应关系，因此，在以后的讨论中，我们通常**不区分虚拟地址空间和虚拟内存，可以相互指代**。

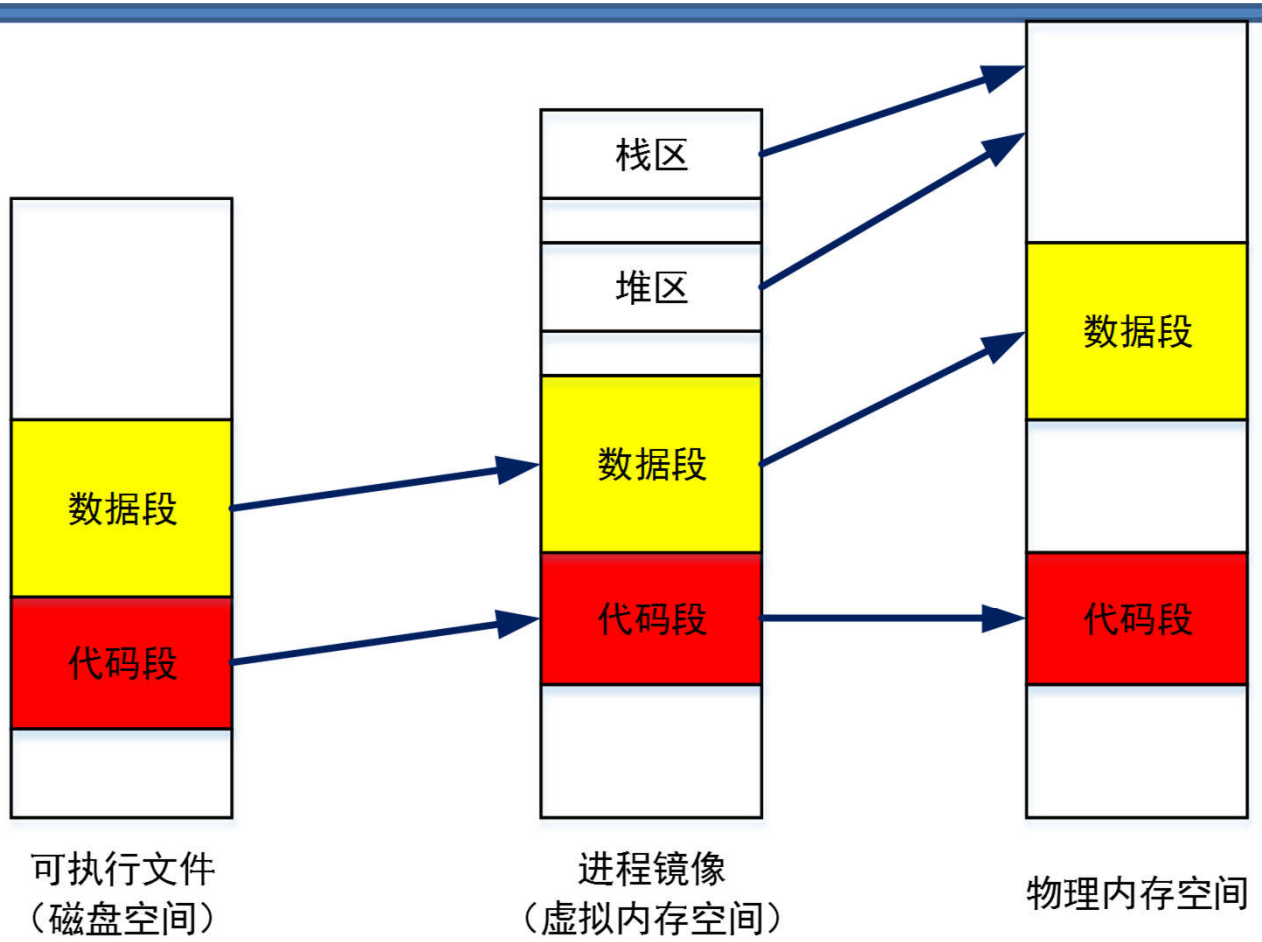

0x0200
.data
0x0101
0x0100
.text
0x0010
0x0000
可执行文件
.data
.text
虚拟地址空间
0x9200
0x9101
.data
0x8100
.text
0x8010
0x8000
物理内存空间

秋曼

2018-10-25

校园秋色 2018-10-25
X-T2

回顾



1. 三个空间都以字节为基本存储单位；
2. 每个字节有唯一地址作为标识。通过地址来引用字节和字节中的数据内容。
3. 可以用地址段来引用一个数据对象。数据对象在不同空间中的映射关系，就是地址映射关系。

5.3.3 虚拟内存

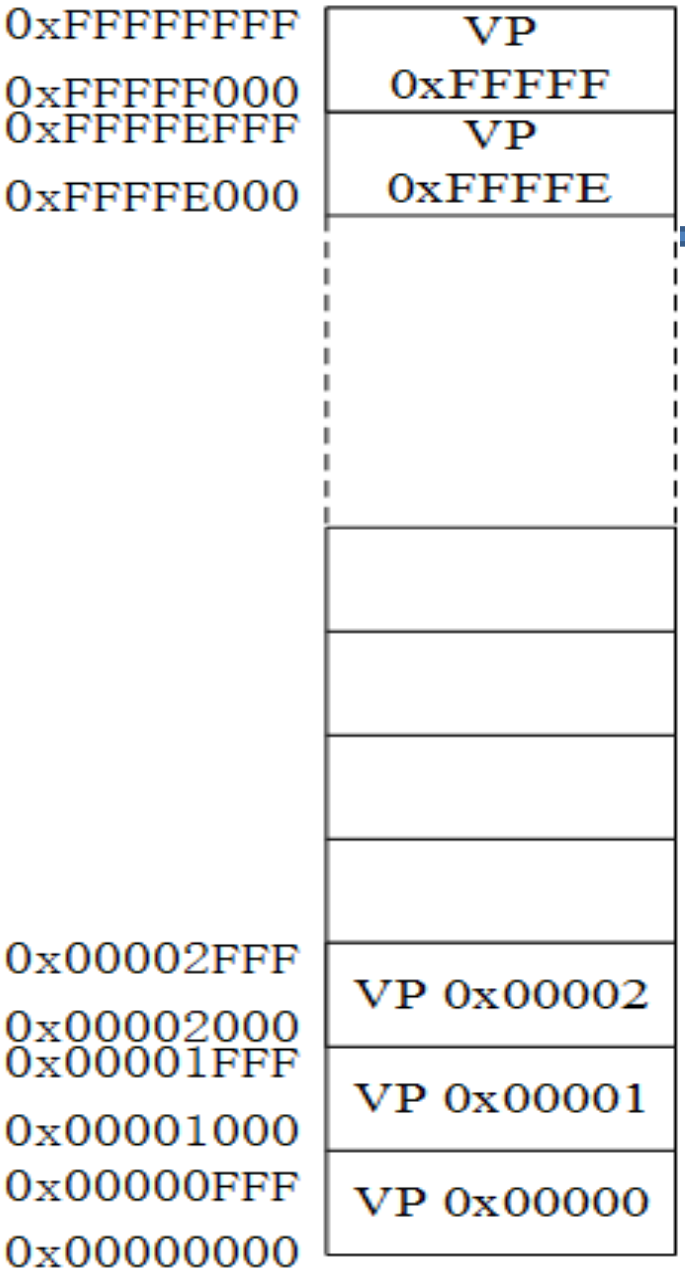
管理虚拟页面和物理页面（Page）——以页面（块）为单位进行管理

为了便于把虚拟内存映射到磁盘块和物理内存块，虚拟内存通常也被划分为固定大小的块，每一个这样的块被称为**虚拟页面**（VPs）（简称**虚页**），页面的大小为$P=2^p$字节。类似的，物理内存也被划分为P个字节大小的**物理页面**（PPs），或简称**实页**或**页框/页帧**（Page frame）。通常虚拟页面的大小为4K，即$p=12$。

计算：

对于大小为4GB的虚拟内存，如果以4KB为一个页面大小。

- 可以划分出多少个虚页?
- 如果为每个虚页从0x0开始连续编址，可以编出多少个虚页地址?
- 地址为0x12345的虚页对应的字节地址范围?
- 虚拟地址为0x12345678的字节所在的虚拟页面地址?



虚拟地址=<虚拟页面号，偏移量offset>

有两个事实：

1、同一页面内的字节地址的高20位是相同的

2、同一页面内的字节地址的低12位总是从000~FFF，这段地址区域的大小恰好是4KB，即页面大小

3、页面号恰好是该页面内字节地址的高20位

基于此，

1、我们用页面号（20位）或页面中首字节地址（32位）来表示页面地址。二者的关系是：

页面中首字节地址=<虚拟页面号（高20位），000>

字节地址与页面号（地址）的关系

- 地址空间的大小为2^n
- 页面大小为2^p

若已知字节地址（二进制）为：

若已知页面地址（二进制）为：$(0|1)^{n-p}$，

则该页面内字节地址的范围是

$(0|1)^{n-p}\wedge 0\cdots 0$，　$(0|1)^{n-p}\wedge 0\cdots 1$，　…，　$(0|1)^{n-p}\wedge 1\cdots 1$

Allocate映射和Cache映射

Allocate映射：虚拟内存空间（页面集合） ----> 磁盘空间（磁盘块集合）。

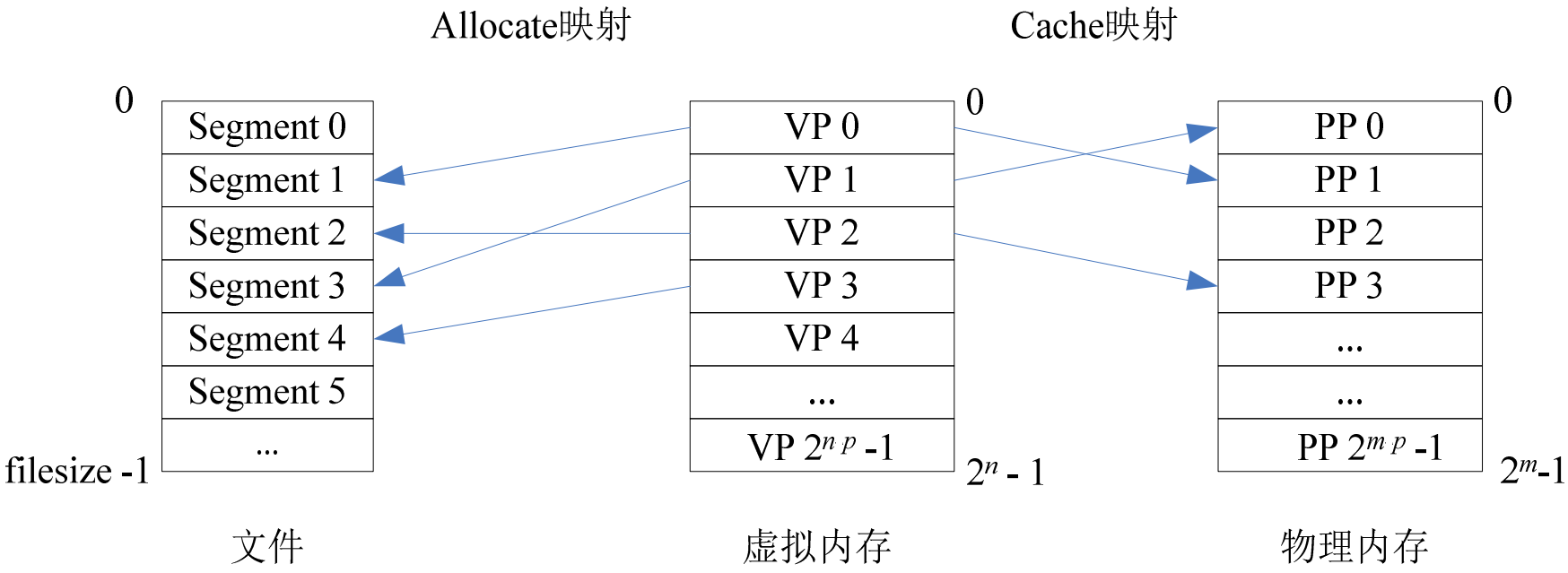
该映射是一个偏函数，将虚拟内存中的一个页面映射到磁盘中的一个数据对象，该数据对象是一段连续字节的区段，这个区段可以是一个普通文件中的一个片段，也可以是磁盘中连续的磁盘块（或称为匿名文件，或Paging file）。如果已经建立了Allocate映射，说明虚拟内存中的一些（或全部）虚页已经分配给了磁盘中的数据对象，这时我们也可以把对应的磁盘空间称为虚拟内存的备份区（Backing store）。

Cache映射：把虚拟内存 ----> 物理内存。

它也是一个偏函数，将一个虚拟页面映射到唯一的一个物理页面。

通过虚拟内存以及这两个映射，间接建立了磁盘空间与物理内存空间之间的关系。当Cache映射发生变化时，说明一个数据对象从原来的物理页面移动到了另一个物理页面，从而实现了一个数据对象在内存中的上下浮动。

【例5-3】图5-14给出了一个虚拟内存与磁盘中的文件以及物理内存的映射关系。通过Allocate映射和Cache映射，间接建立磁盘空间与物理内存的关系。



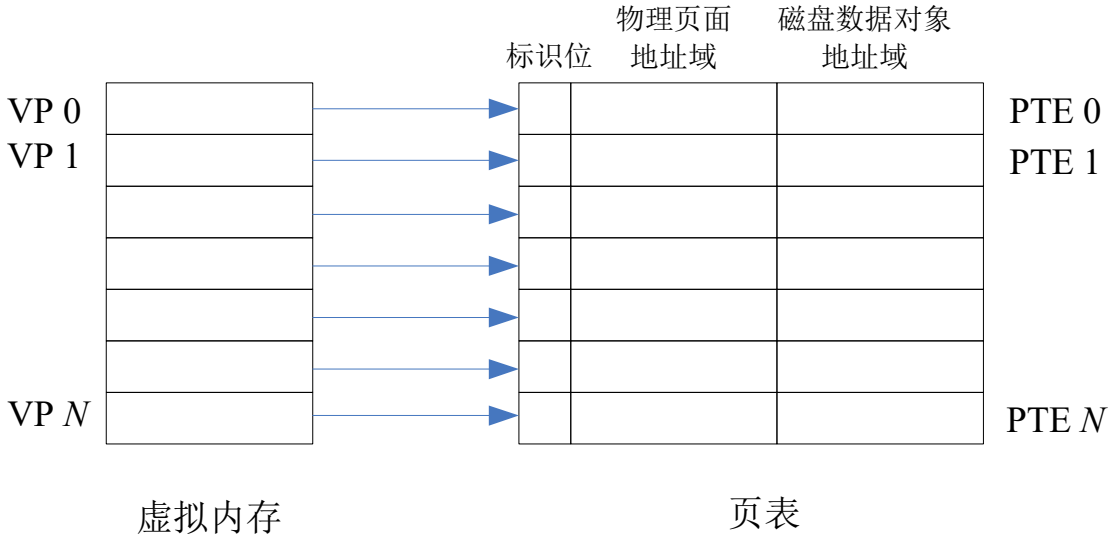
虚拟页面的状态

页面状态	Allocate映射	Cache映射	说明
Unallocated（未分配）	未建立	未建立	该虚页还没有分配给一个磁盘数据对象，因此不占用磁盘上的任何空间。如果程序访问该虚页中的指令或数据，那么就会出现非法（invalid）访问异常。
Cached（已缓存状态）	已建立	已建立	该虚页已经分配给一个磁盘数据对象，而且已被缓存（或已加载）到一个的物理页面中。
Uncached（未缓存状态）	已建立	未建立	该虚页已经分配给一个磁盘数据对象，但该数据对象还没有被换入（swap in）到一个物理页面中，仍然位于磁盘中。
--	未建立	已建立	该状态不存在

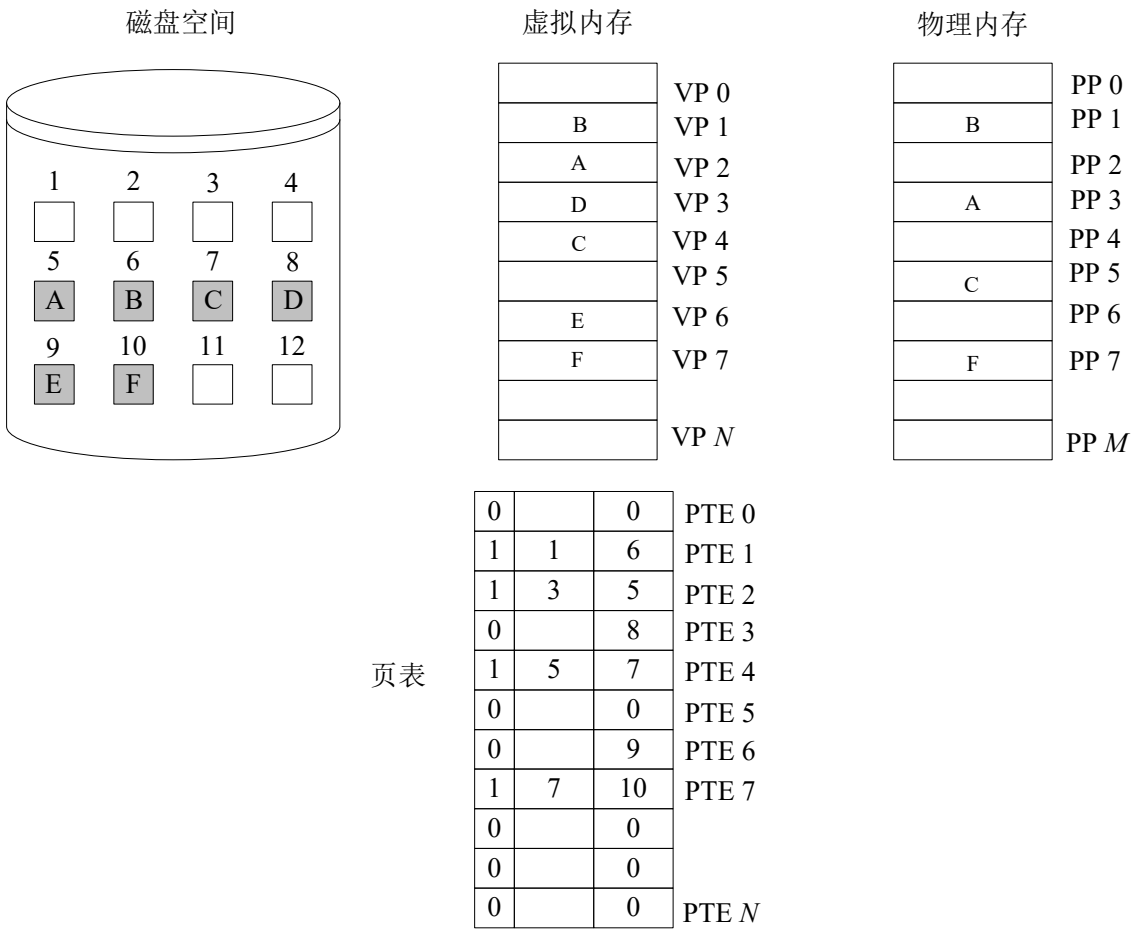
5.3.4 页表

为了方便的描述Allocate和Cache映射，需要一个称为“**页表**”（PT，Page Table）的数据结构。页表是一个由**页表项**（PTEs，Page Table Entries）组成的数组。每个虚拟页面在页表中都对应一个页表项。

就目前而言（以后根据应用需求，还会对页表项进行扩充），每个页表项PTE包含一个1位标识位（valid bit）、一个物理页面地址域和一个磁盘数据对象地址域。标识位和地址域一起指示虚拟页面当前的状态。



【例5-4】图5-16用一个页表描述虚拟内存的映射状态。在N+1个虚拟页面中，6个虚拟页面分别被分配给6个磁盘块（5号块~9号块），其中4个页面已经缓存在物理页面中。其余虚拟页面均未分配。



关于页表还需要注意以下几点：

•页表存储在物理内存中，占有一定存储空间。对于一个n位虚拟地址空间，虚拟内存中虚拟页面的个数为2^{n-p}个，那么页表中的页表项就有2^{n-p}个。如果一个页表项占用m个字节，那么页表总共占用$m\times 2^{n-p}$个字节。

•页表的状态不是一成不变的。如果操作系统把一个虚拟页面对应的磁盘数据对象换入（swap in）到了某个物理页面中，那么相应页表项的状态变化为

0		磁盘数据对象的地址

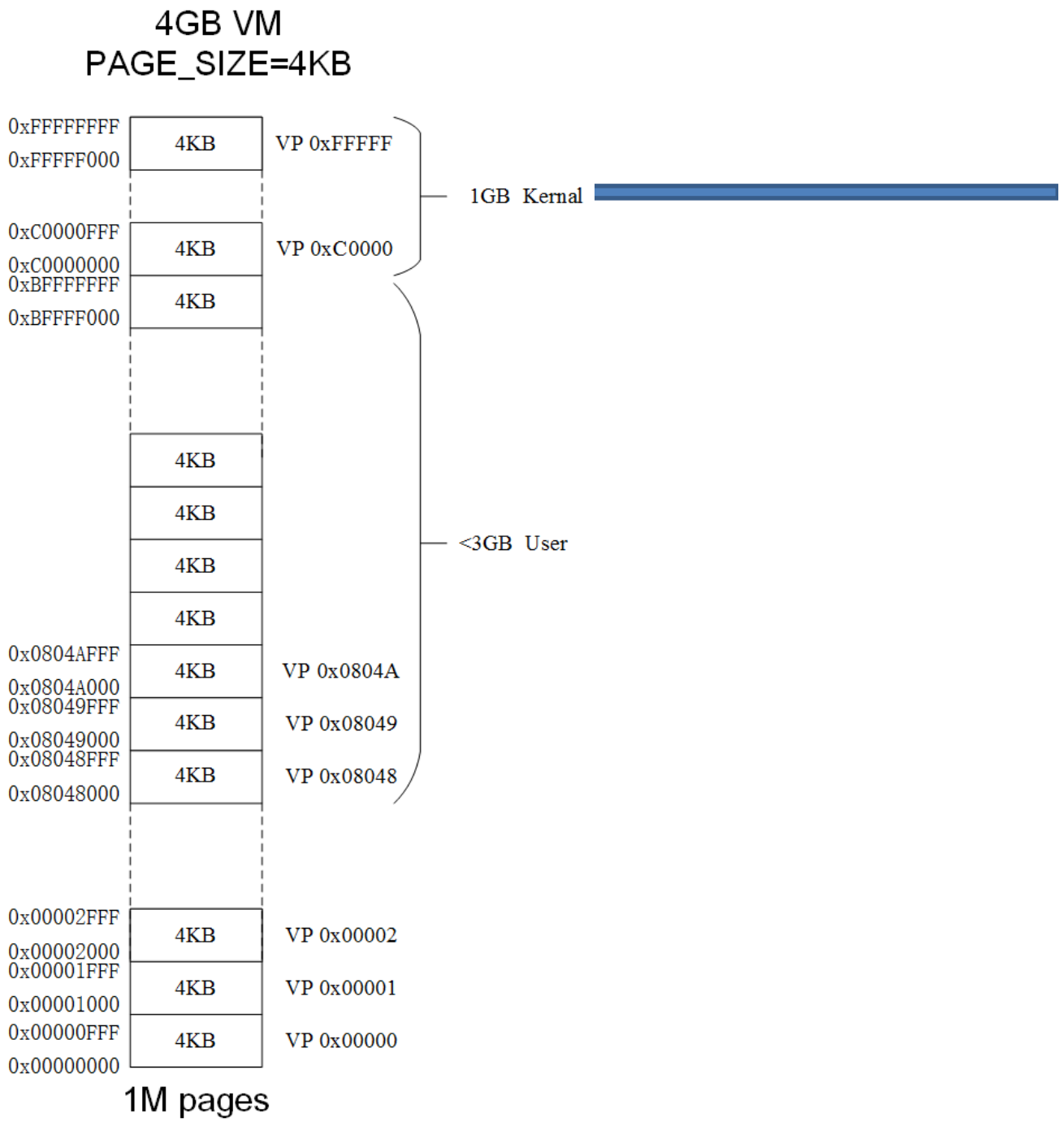
1	物理页面起始地址	磁盘数据对象的地址

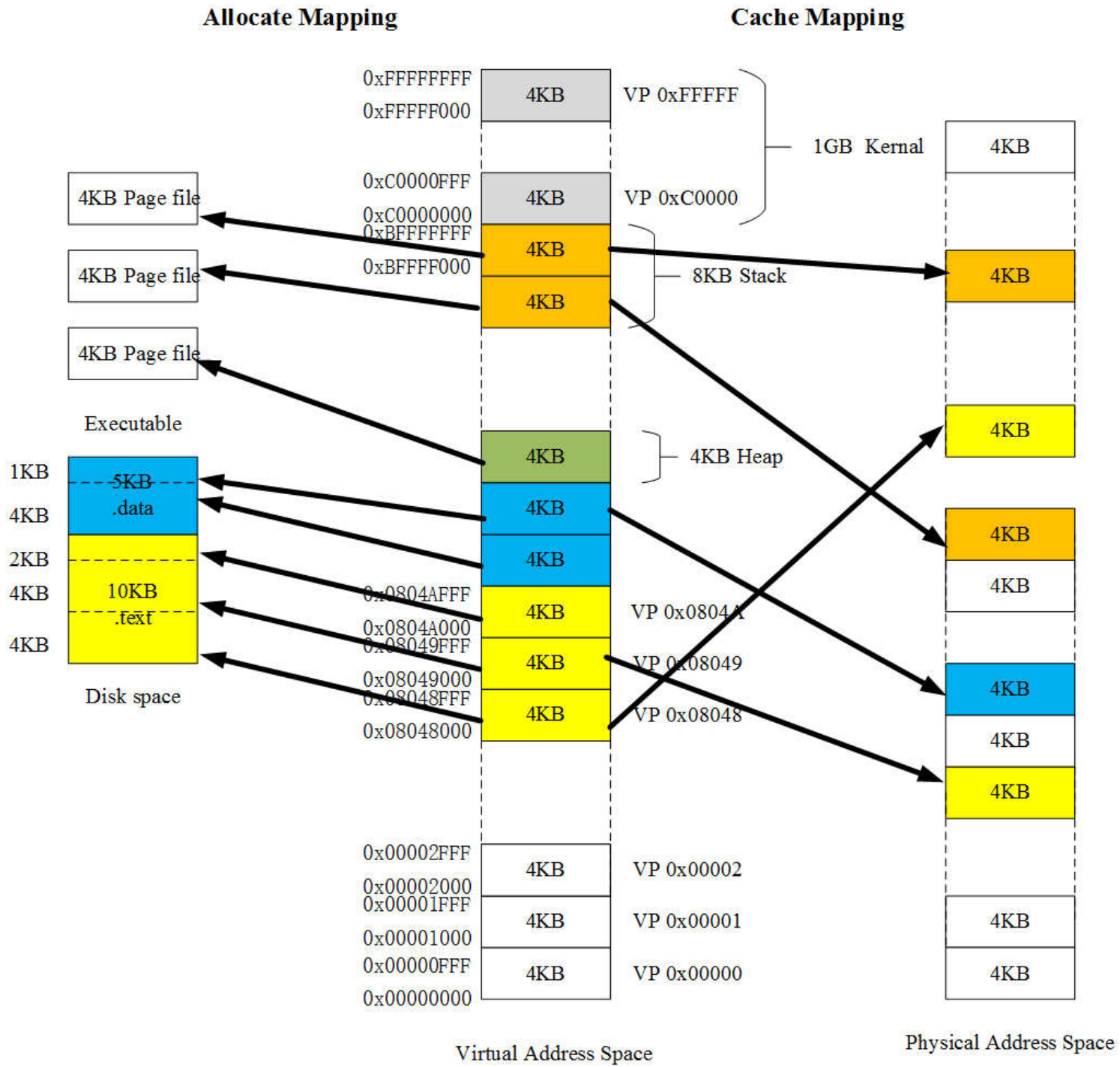
如果操作系统把某个虚拟页面从物理页面中换出（swap out）到磁盘空间，那么对应的页表项状态变化为

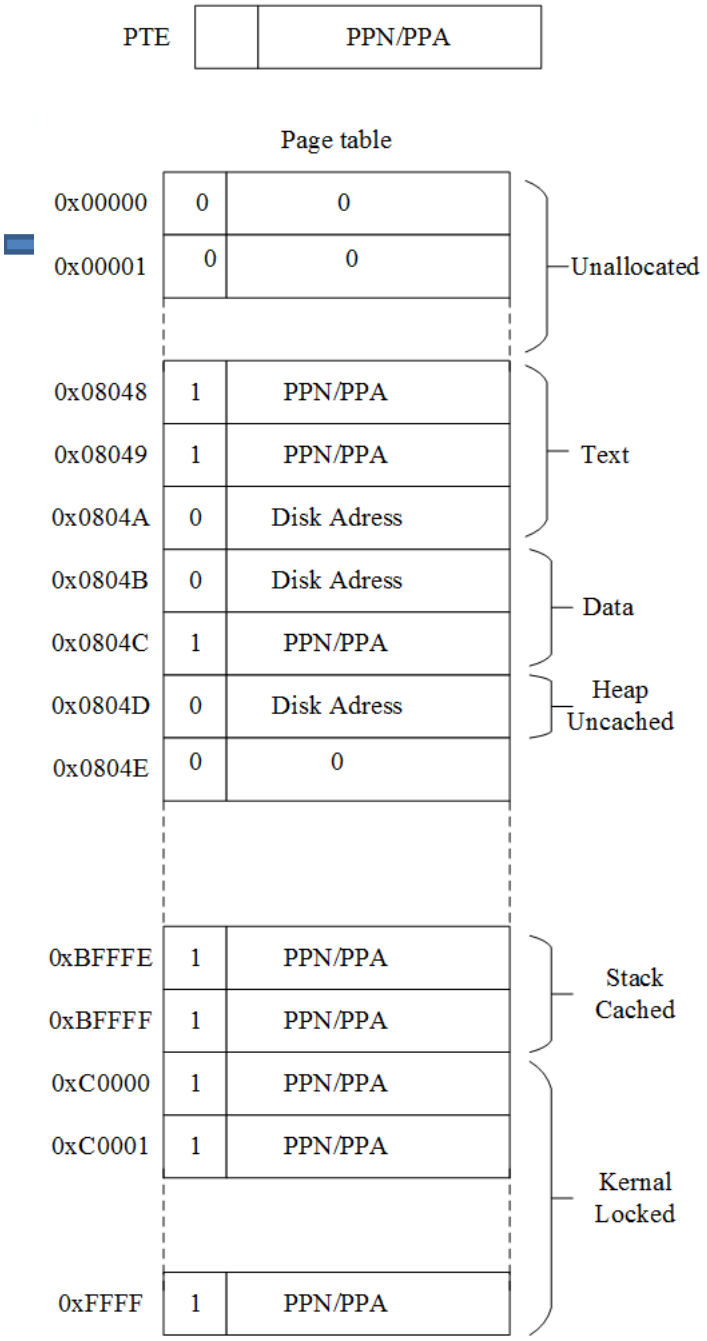
1	物理页面起始地址	磁盘数据对象的地址

0		磁盘数据对象的地址

- 回顾


4GB VM
PAGE_SIZE=4KB
0xFFFFFFFF
0xFFFFF000
4KB
VP 0xFFFFF
1GB Kernal
0xC0000FFF
0xC0000000
4KB
VP 0xC0000
0xBFFFFFFF
0xBFFFF000
4KB
4KB
4KB
4KB
4KB
<3GB User
0x0804AFFF
0x0804A000
4KB
VP 0x0804A
0x08049FFF
0x08049000
4KB
VP 0x08049
0x08048FFF
0x08048000
4KB
VP 0x08048
0x00002FFF
0x00002000
4KB
VP 0x00002
0x00001FFF
0x00001000
4KB
VP 0x00001
0x00000FFF
0x00000000
4KB
VP 0x00000
1M pages


Allocate Mapping
Cache Mapping
0xFFFFFFFF
0xFFFFF000
4KB
VP 0xFFFFF
1GB Kernal
0xC0000FFF
0xC0000000
4KB
VP 0xC0000
0xBFFFFFFF
0xBFFFF000
4KB
4KB
8KB Stack
4KB Page file
4KB Page file
4KB Page file
Executable
4KB
4KB Heap
1KB
5KB
.data
4KB
2KB
10KB
.text
4KB
4KB
4KB
4KB
4KB
4KB
0x0804AFFF
0x0804A000
4KB
VP 0x0804A
0x08049FFF
0x08049000
4KB
VP 0x08049
0x08048FFF
0x08048000
4KB
VP 0x08048
0x00002FFF
0x00002000
4KB
VP 0x00002
0x00001FFF
0x00001000
4KB
VP 0x00001
0x00000FFF
0x00000000
4KB
VP 0x00000
4KB
4KB
4KB
4KB
4KB
4KB
4KB
4KB
4KB
Disk space
Virtual Address Space
Physical Address Space



虚拟VPA地址转换：

1、求出vaddr对应的虚页号VPN和页内偏移VPO

- VPN= vaddr /PAGE_SIZE

  VPO=vaddr % PAGE_SIZE

- vaddr = < (n-p)位, p位>=<VPN, VPO>

2、以VPN为索引，查表中找到对应的页表项PTE

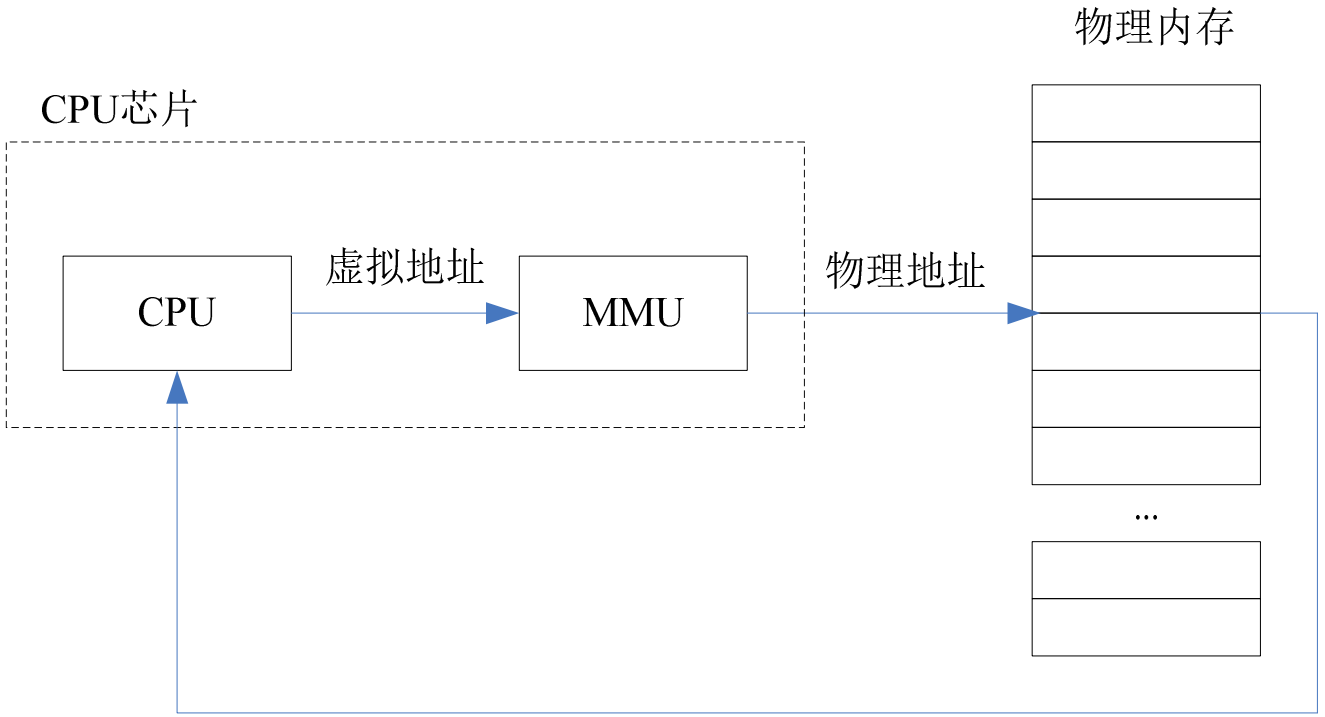
- if valid-bit(PTE)=1，说明vaddr所在的虚页已经加载到物理页面中，那么：

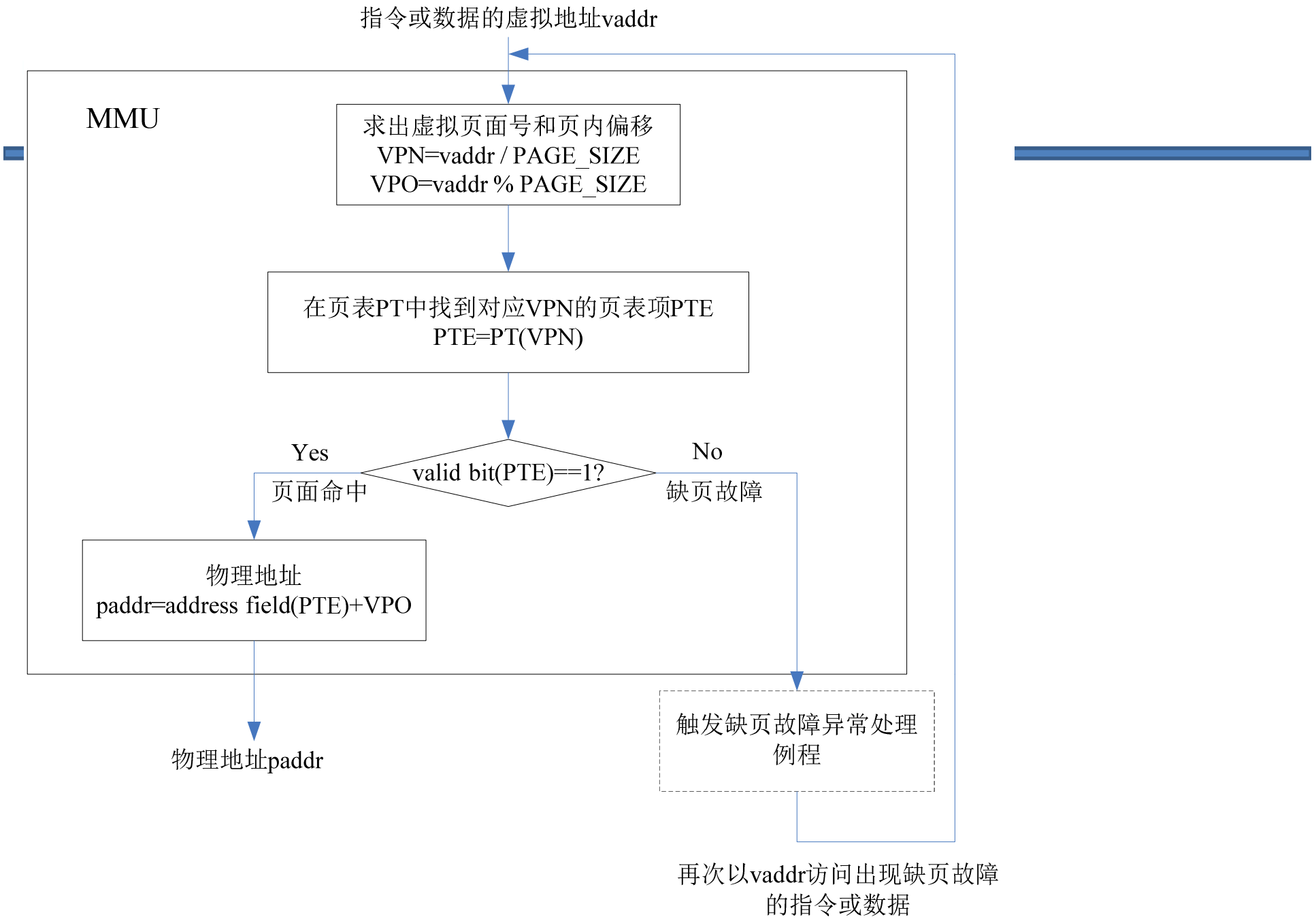
  paddr = PPA+VPO，或=PPN*SIZE+VPO

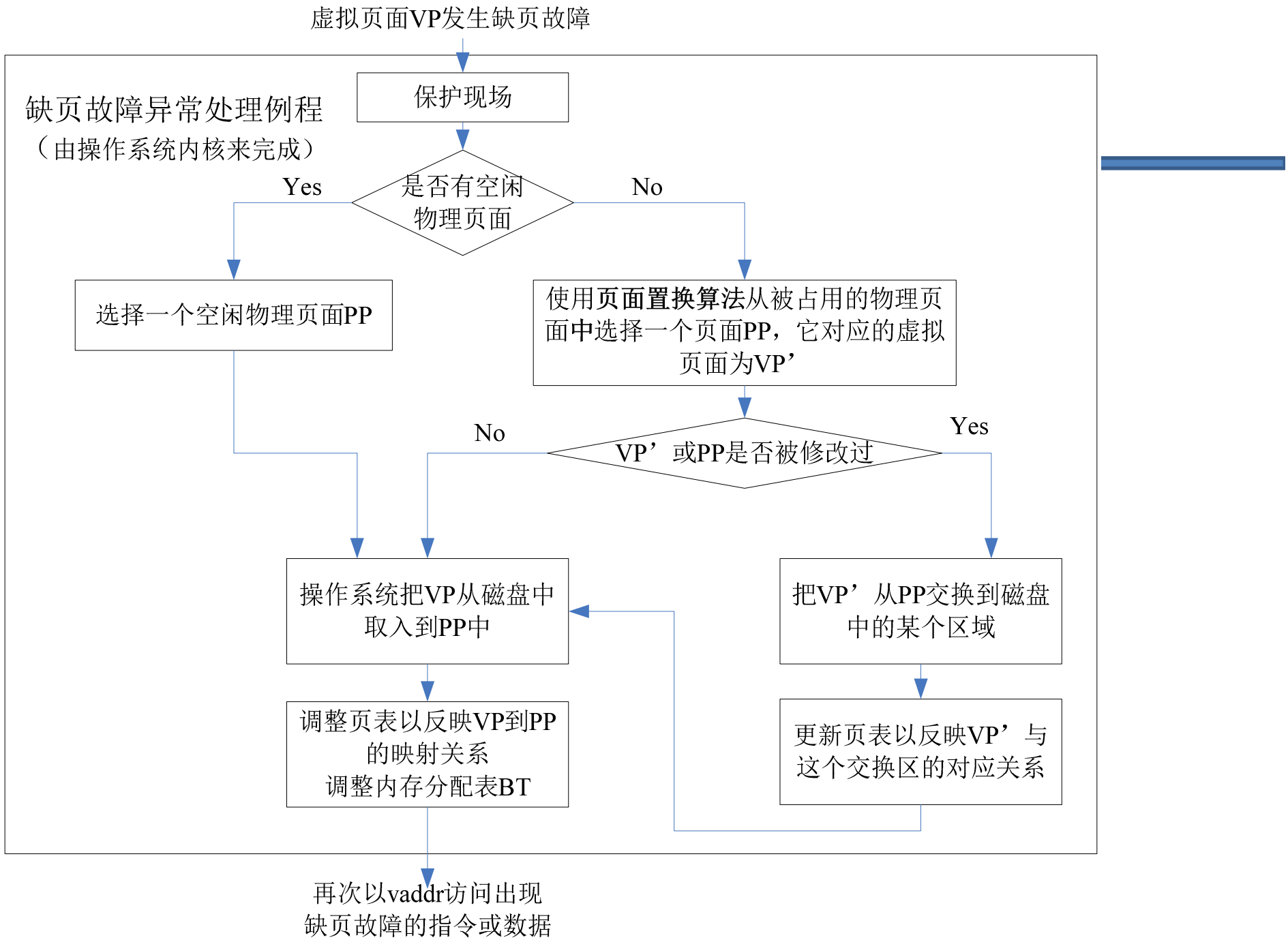
- if valid-bit(PTE)=0，说明vaddr所在的虚页还没有加载到物理页面中，那么中断当前正在执行的程序，把控制流转移到操作系统的缺页故障处理程序，该程序把虚页对应的磁盘数据加载到一个分配的物理页面中。

5.3.5 虚拟地址转换和缺页故障（Page faults）处理

在虚拟内存系统中，CPU不是直接用物理地址来访问程序指令和数据，而是先用虚拟地址访问虚拟内存，然后通过硬件地址转换机构MMU，把虚拟地址转换为物理地址，最后通过物理地址访问内存中的指令和数据。




指令或数据的虚拟地址vaddr
MMU
求出虚拟页面号和页内偏移
VPN=vaddr / PAGE_SIZE
VPO=vaddr % PAGE_SIZE
在页表PT中找到对应VPN的页表项PTE
PTE=PT(VPN)
valid bit(PTE)==1?
Yes
页面命中
No
缺页故障
物理地址
paddr=address field(PTE)+VPO
物理地址paddr
触发缺页故障异常处理例程
再次以vaddr访问出现缺页故障的指令或数据


虚拟页面VP发生缺页故障
缺页故障异常处理例程
（由操作系统内核来完成）
保护现场
是否有空闲物理页面
Yes
No
选择一个空闲物理页面PP
使用页面置换算法从被占用的物理页面中选择一个页面PP，它对应的虚拟页面为VP’
VP’或PP是否被修改过
No
Yes
操作系统把VP从磁盘中取入到PP中
把VP’从PP交换到磁盘中的某个区域
调整页表以反映VP到PP的映射关系
调整内存分配表BT
更新页表以反映VP’与这个交换区的对应关系
再次以vaddr访问出现缺页故障的指令或数据

为了判断一个虚拟页面（或物理页面）是否被修改过，通常需要在页表中增加一个**变更位**（Modify bit，或dirty bit），做了这样的扩充之后，页表项的结构变为：

PTE	标识位	变更位	物理页面起始地址域	磁盘数据对象地址域

在判断内存中是否存在空闲物理页面，以及从物理页面中选择一个将被换出的页面时，需要了解内存中物理页面的状态，我们用内存分配表BT（Block Table）（或称块表）来描述这个状态。与页表类似，BT也由若干表项BTE组成，每个表项BTE*i* 描述物理页面PP*i* 的空闲和占用状态，比如用1表示“占用”，0表示“空闲”。

- 注意：

1、虚拟地址转换是由MMU硬件来完成，而缺页故障处理过程是由MMU硬件和操作系统软件共同来完成的：MMU硬件判断对应页表项的标志位，如果标志位为0，那么产生一个缺页故障，对缺页故障的处理由操作系统内核中的缺页故障处理例程来完成。

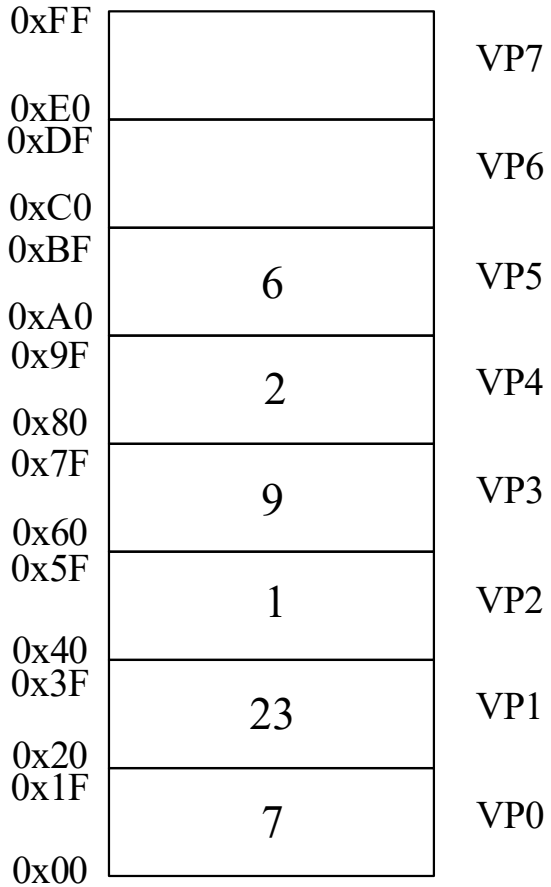
2、 MMU在进程虚拟地址转换时，只需要用到页表中的部分列，如标志位和物理页面起始地址域（或物理页面号），而缺页故障处理过程中需要用到页表中另外一些列，如变更位和磁盘数据对象地址域等。因此，在有些文献中，把缺页故障处理过程中用到的部分页表称为“软页表”（Software Page Table）。

【例5-4】考虑这样一个虚拟内存，虚拟地址用8位来表示，其中高3位表示虚拟页面号，低5位表示页面偏移。虚拟内存以字节为单位进行编址和访问。假设一个进程占用6个虚页，其页表如下：

虚拟页面号	有效位	物理页面号	变更位	磁盘块号
0	1	4	0	7
1	1	7	0	23
2	0	-	-	1
3	1	0	1	9
4	1	5	1	2
5	0	-	-	6
6	未分配			
7	未分配			

- 求虚拟内存的大小、页面大小和页面的个数？
- 写出每个页面的虚拟地址（用十六进制表示）范围？
- 如果程序计数器PC=0x25，在访问这条指令时是否会发生缺页故障？如果不会，那么MMU把该地址转换成的物理地址是多少？
- 执行指令mov %eax, 0xD1时，能否执行成功？

虚拟页面号	有效位	物理页面号	变更位	磁盘块号
0	1	4	0	7
1	1	7	0	23
2	0	-	-	1
3	1	0	1	9
4	1	5	1	2
5	0	-	-	6
6	未分配			
7	未分配			



5.3.6对内存管理需求的支持

本章一开始，我们就对内存管理提出了可重定位、共享、保护和可扩充等四个基本需求。虚拟内存的引入，可以使我们以一种非常一致、简洁和优雅的方式实现上面的需求。

（1）对重定位（上下浮动）的支持: VM + PT + 动态地址变换

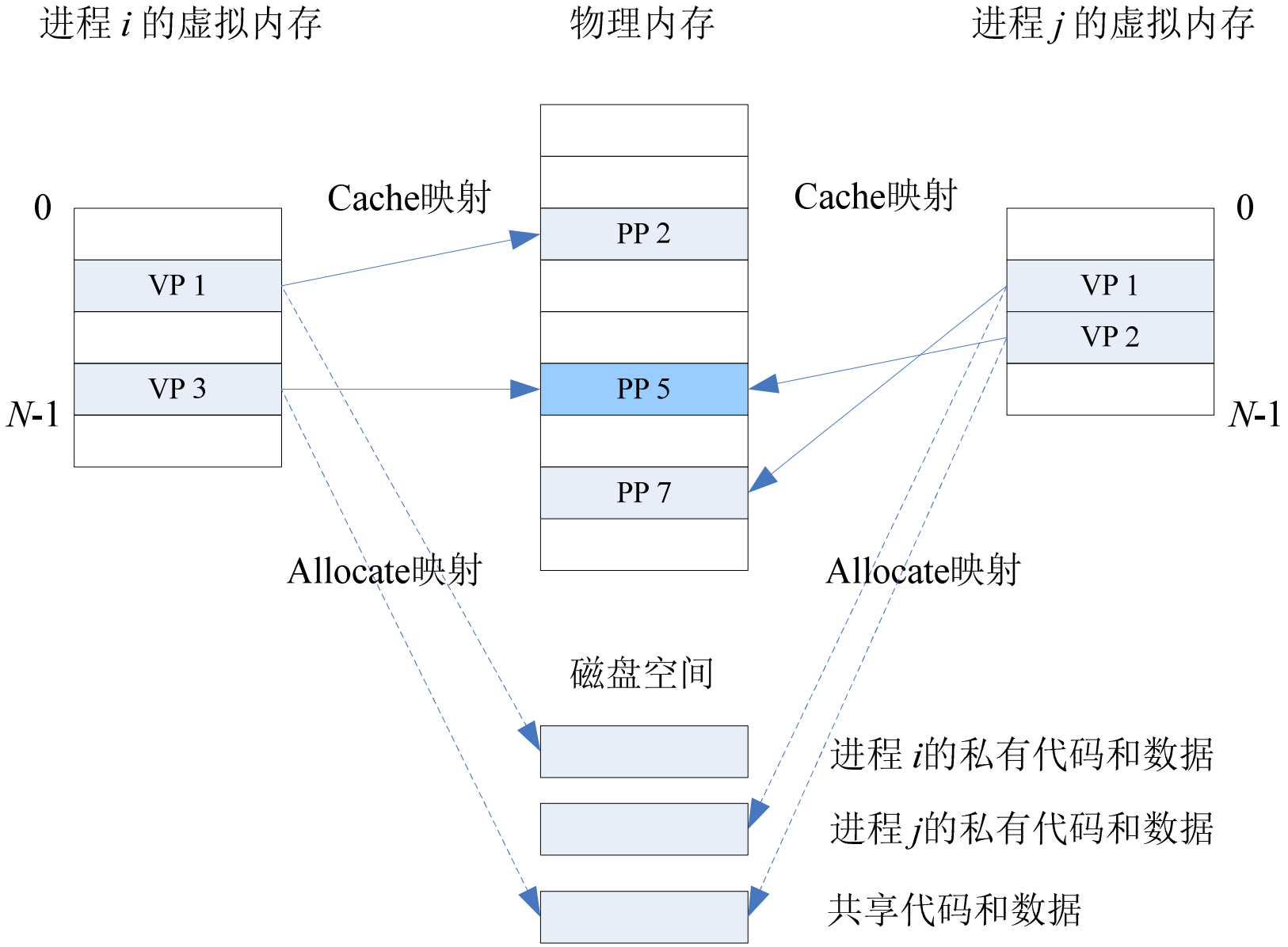
（2）对私有和共享的支持：每个进程都有自己的页表+通过页表把自己的虚拟地址空间映射到不同的物理内存空间，以此来实现私有性和隔离性。

同时，不同进程也可以把自己虚拟地址空间中的部分虚页映射到同一物理页面，以此来实现进程间的共享。（share memory的实现）

（3）对离散内存分配的支持：通过PT

（4）对内存保护的支持：内存访问控制作用在虚拟内存的页面上，而不是物理内存上，由于虚拟内存布局是稳定的、一致的，因此内存保护容易实施。

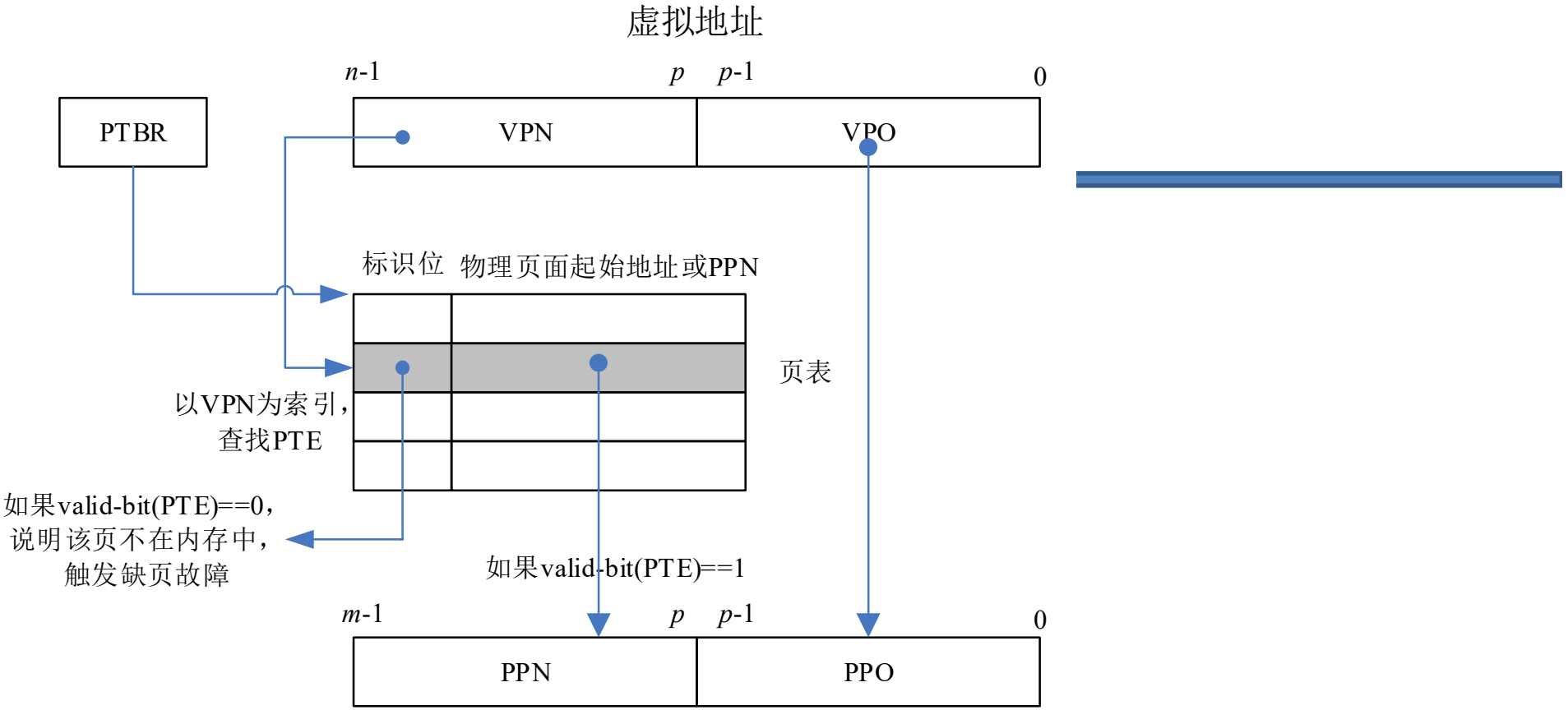
（5）对存储器扩充的支持：运行时加载+部分加载


进程 i 的虚拟内存
物理内存
进程 j 的虚拟内存
0
VP 1
VP 3
N-1
Cache映射
Cache映射
PP 2
PP 5
PP 7
VP 1
VP 2
0
N-1
Allocate映射
Allocate映射
磁盘空间
进程 i的私有代码和数据
进程 j的私有代码和数据
共享代码和数据

5.3.7 地址转换的硬件实现和加速

地址转换的硬件实现

符号	描述
$N=2^n$	n是虚拟地址的位数，N是虚拟地址空间的大小
$M=2^m$	m是物理地址的位数，M是物理地址空间的大小
$P=2^p$	P是页面大小（字节）
VPO	虚拟页面偏移量（字节）
VPN	虚拟页面号
PPO	物理页面偏移量（字节）
PPN	物理页面号



设n位虚拟地址vaddr可以简记为vaddr=<VPN，VPO>。在页面命中情况下，虚拟地址转换过程可以表示为公式：

paddr = <address-field (PT(VPN)),VPO>

其中，PT(VPN)表示以VPN为索引，在页表PT中找到对应的页表项PTE；

address-field(PT(VPN))表示页表项PT(VPN)的物理页面地址或页面号。

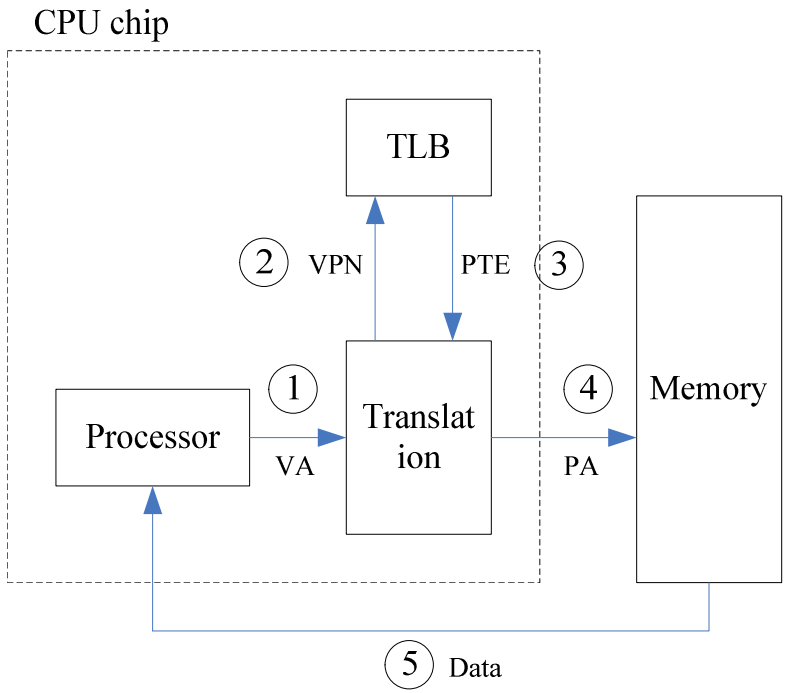
【例5-5】给定一个32位虚拟地址空间和24位物理地址空间，页面大小为P=4KB。

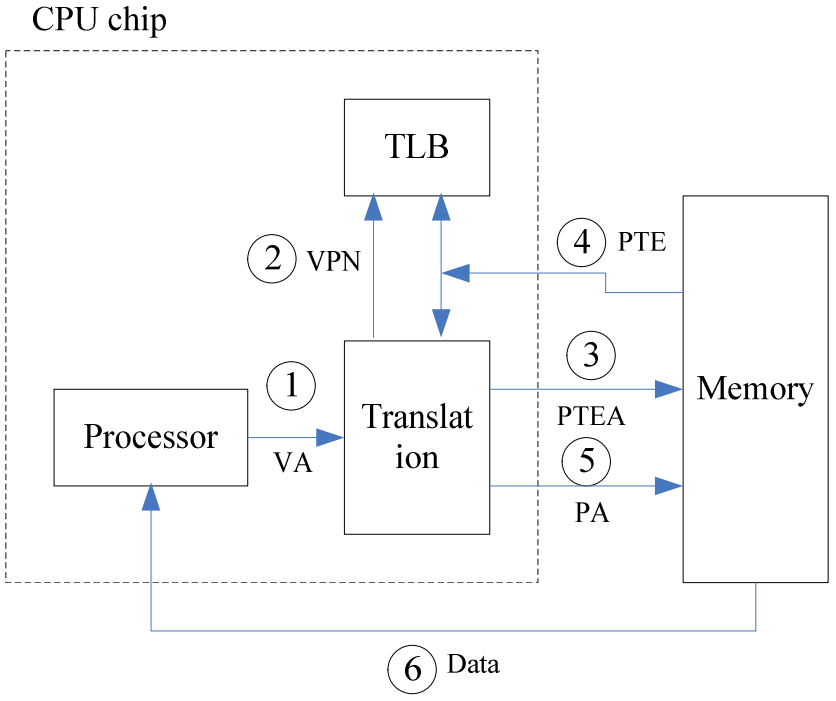
- 求VPN，VPO，PPN和PPO的位数?
- 求虚拟页面和物理页面的个数?
- 把虚拟地址0x08048010转化为物理地址，其中页表为

	标志位	物理页面号
0x00000	.	...
0x00001	.	...
0x00002	.	...
	……	
0x08047	.	...
0x08048	1	0x123
	……	
0xFFFFE	.	...

CPU每产生一个虚拟地址，MMU就必须访问一次PTE以进行地址转换。由于页表在内存中，地址转换过程会带来额外的内存访问，将花费数十到数百个周期。为了加速地址转换过程，减小访问页表的开销，许多计算机系统在MMU中增加了一个称为“**快表**”（TLB，Translation Lookaside Buffer）的存储部件，用来缓存最近访问的部分页表项。

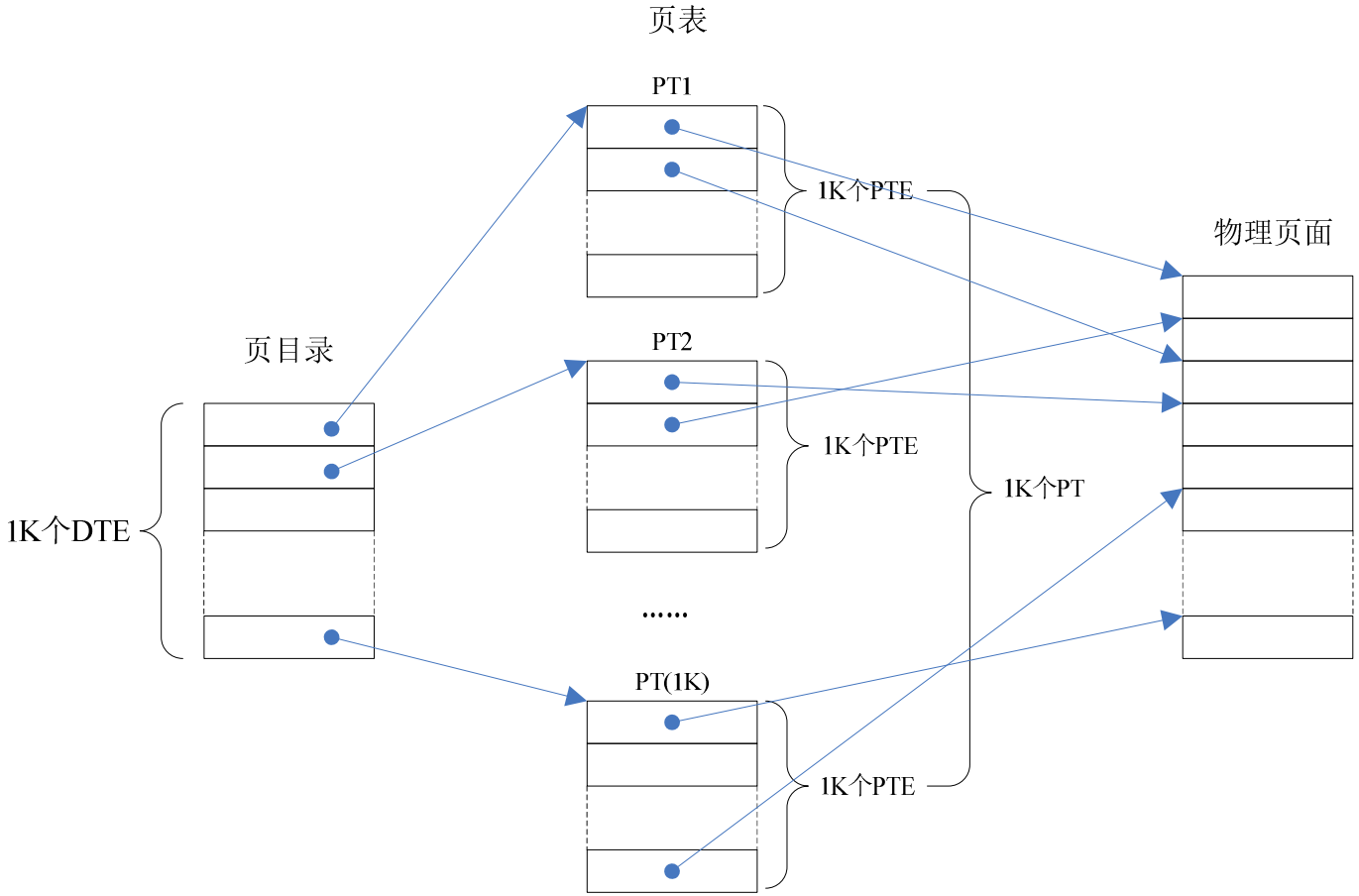
TLB的有效性基于程序局部性原理



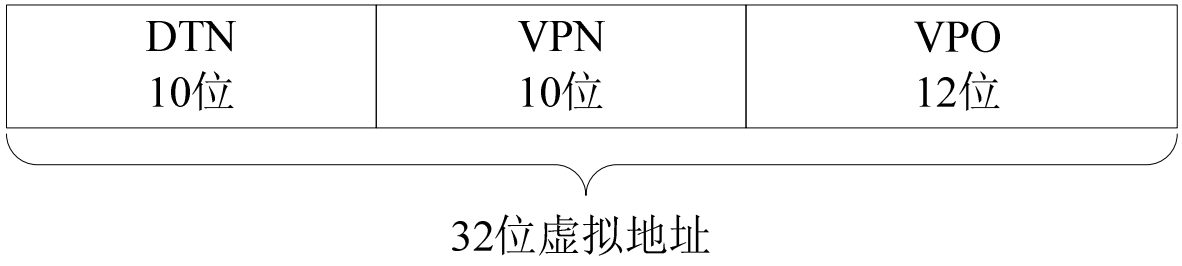


多级页表

通常，我们采用多级页表来管理虚拟页面与物理页面的映射关系。比如对于2级页表，第1级页表包括若干页表项，每个页表项的地址域指向第2级页表的基地址。每个第2级页表的页表项的地址域指向物理页面的基地址。通常我们把第1级页表称为“页目录”，对应的页表项称为“页目录表项”（DTE，Directory Table Entry），第2级页表仍然称为页表。



与1级页表类似，每一级页表的页表项数目与虚拟地址的分段划分是一一对应的。比如，对于图5-24所示的2级页表结构，它对应这样的虚拟地址划分：



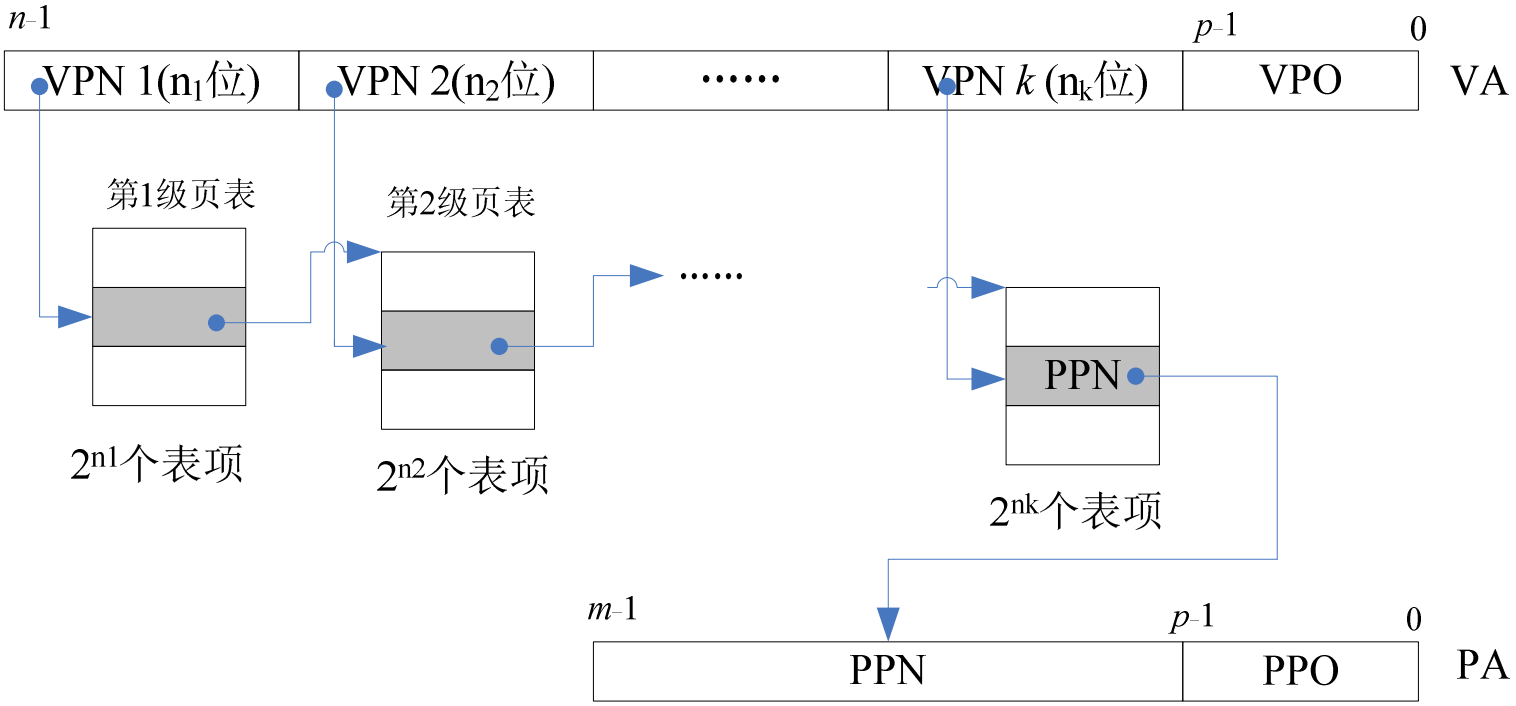
给定一个虚拟地址 vaddr = <DTN, VPN, VPO>，可以通过如下步骤进行地址转换：

1. 根据DTN，在目录表DT中找到第2级页表的基址：DT(DTN)
2. 根据VPN，在相应的第2级页表中找到对应的页表项：DT(DTN)(VPN)
3. 获取页表项DT(DTN)(VPN)的地址域，即物理页面的基址：

   address-field (DT(DTN)(VPN))

4. 然后将物理页面基址与VPO拼接，形成物理地址：

   paddr=<address-field(DT(DTN)(VPN)), VPO>


n-1
p-1
0
VPN 1(n1位)
VPN 2(n2位)
……
VPN k (nk位)
VPO
VA
第1级页表
第2级页表
……
PPN
2n1个表项
2n2个表项
2nk个表项
m-1
p-1
0
PPN
PPO
PA

- 分页管理回顾
  - 一个进程的映像（image）是什么时候被加载进物理内存的?
  - 一个进程的映像要被完全加载到物理内存吗?
  - 一个进程的映像要被连续加载物理内存吗?
  - 一个进程要被换出（swap out）时，是部分换出还是完全换出?
  - 当一个进程再次被换入时，一定要被换入到上一次被换出时的物理内存吗?
  - 页表

策略	说明
读取策略	用于决定何时将一个虚拟页面换入内存。常用的两种方法是: ● 请求分页（Demand paging）：只有当访问到某虚页中的一个数据单元时才将该页换入内存。 ● 预取分页（Prepaging）：预测以后将要访问到的虚拟页面，并提前将这些页面取入内存。
驻留策略	决定一个进程的虚拟地址空间驻留在（被缓存在）哪些物理页面。对于分页管理来说，这个问题通常无关紧要，因为无论进程驻留在哪些物理页面，地址转换硬件和内存访问硬件都会以相同的效率访问到进程的数据。
置换策略	当没有足够的空闲物理页面，需要把一个物理页面换出时，决定换出哪一个物理页面。
驻留集管理	驻留集管理涉及两个问题: ● 驻留集合大小：为每个活动进程分配多少物理页面才合理 ● 置换范围：当需要挑选一个物理页面被置换时，是在发生缺页故障的进程物理页面集合中挑选，还是在所有进程的物理页面集合中挑选，即局部置换还是全局置换。
换出策略	换出策略与读取策略相反，它用于确定何时将一个修改过的页面写回磁盘。通常有两种选择: ● 请求式：只有当一个物理页面被挑选用于置换时，才被写回磁盘; ● 预先式：将已修改的多个页面在需要用到它们所占的物理内存之前，成批的写回磁盘。
加载控制	决定驻留在内存中进程的数目，或称为“系统并发度”。如果某一时刻，驻留的进程太少，所有进程同时处于阻塞状态的概率较大，则会有许多时间花费在交换上；另一方面，如果驻留的进程太多，平均每个进程占用的内存较少，就会发生频繁缺页故障，从而导致抖动。

5.4.1 程序局部性原理

局部性原理有两种形式：

•**时间局部性**（Temporal locality）：如果一个内存单元刚被程序访问过一次，那么在不久的将来，该程序很可能多次重复访问该内存单元；

•**空间局部性**（Spatial locality）：如果一个内存单元刚被程序访问过一次，那么在不久的将来，该程序很可能会访问该内存单元附近的其它单元。

```
1    int sumvec(int v[N])
2    {
3       int i, sum=0;
4
5       for(i=0;i<N;i++)
6         sum+=v[i];
7       return sum;
8    }
```

■ 读取策略

读取策略用于决定何时将一个虚拟页面取入内存。该策略的两个选择是：

• 请求分页：请求分页是一种延迟换入策略（Lazy swapper），不是把整个进程虚拟内存换入物理内存，而是当访问到某虚拟页面中的一个单元时，才将该页取入内存，否则不会换入任何页面。

• 预取分页：采用特定的算法预测程序将要访问的虚拟页面，将这些页面预先取入内存，或者利用大多数辅存设备（如磁盘）的特性，一次从辅存设备中读取多个连续虚拟页面，并把它们换入内存，即使有些页面目前还没有被访问到。

请求分页策略，程序开始运行时，将产生大量缺页故障，之后会迅速减小。

请求分页的一个关键需求是，缺页故障处理完之后，要重新执行发生缺页故障的指令。缺页故障可以发生在指令循环内存访问的任何时刻：

•如果在读取指令时发生缺页故障，那么故障处理结束后，需要再次启动该指令的读取；

• 如果在访问一个指令操作数时发生缺页故障，仍然需要再次读取、解码和执行该指令。

ADD A, B, C

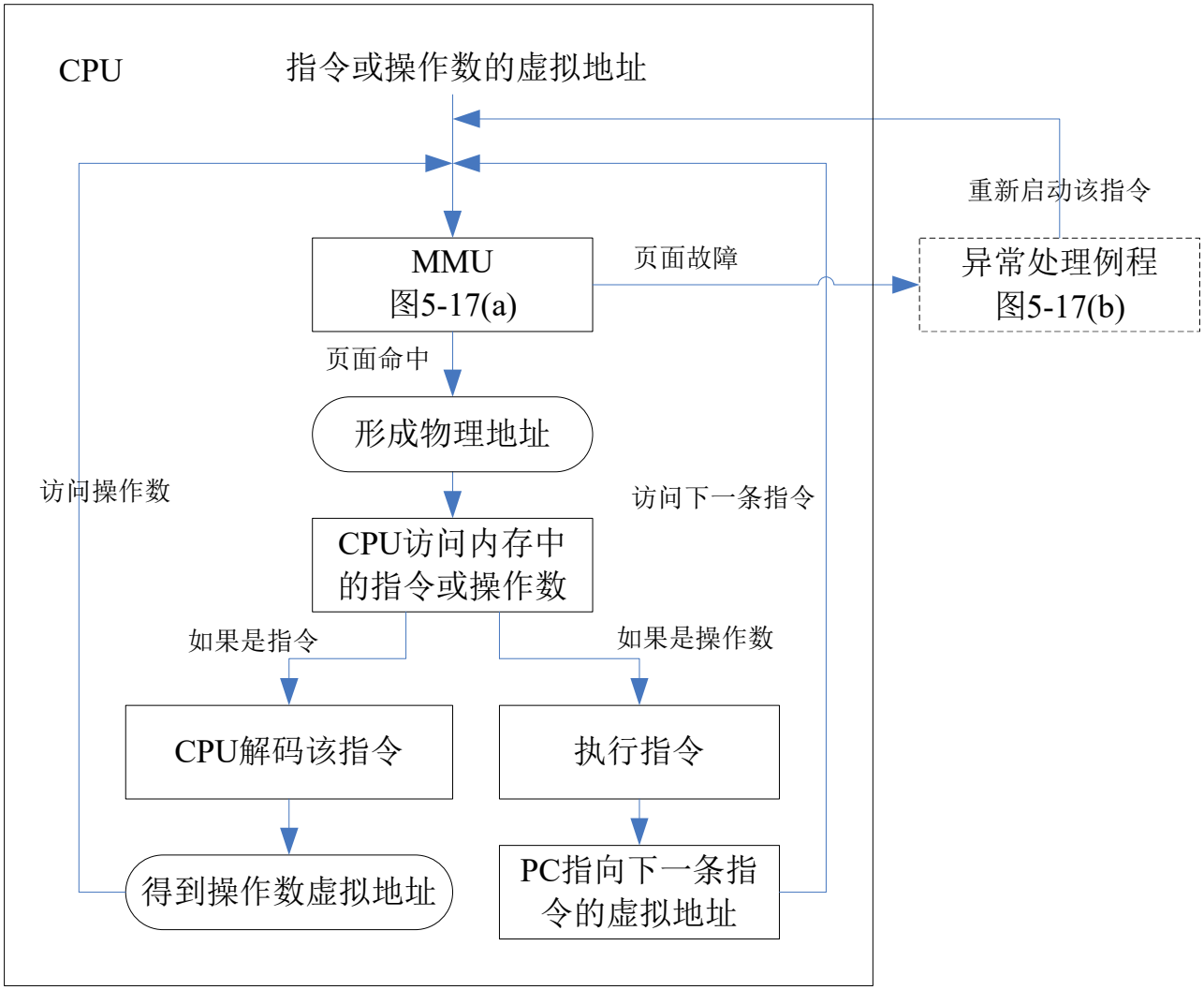
1．读取和解码ADD指令

2．读取A

3．读取B

4．把A和B相加

5．将相加结果存入C中


CPU
指令或操作数的虚拟地址
MMU
图5-17(a)
页面故障
异常处理例程
图5-17(b)
重新启动该指令
页面命中
形成物理地址
访问操作数
访问下一条指令
CPU访问内存中
的指令或操作数
如果是指令
如果是操作数
CPU解码该指令
执行指令
得到操作数虚拟地址
PC指向下一条指
令的虚拟地址

置换策略

置换策略是指，当内存中没有空闲物理页面，并且需要把一个新虚拟页面换入内存时，如何在候选物理页面集合中选择一个物理页面来置换。所有策略的目标都是移出最近最不可能访问的页。根据局部性原理，最近的访问历史和最近将要访问的模式之间具有很大相关性。因此大多数策略都是基于过去的行为来预测将来的行为。置换策略设计得越精细、越复杂，实现它的软硬件开销就越大，因此需要考虑策略的功能和性能之间的均衡。

置换策略有一个约束：某些虚拟页面或物理页面可能是被“锁定”（locked）在内存中的。如果一个页面被锁定，那么该页就不能被置换。

- 大部分操作系统内核和重要的控制结构就锁定的页面中。
- I/O缓冲区和其它对时间有严格要求的区域也可能锁定在物理页面中。

页面置换算法

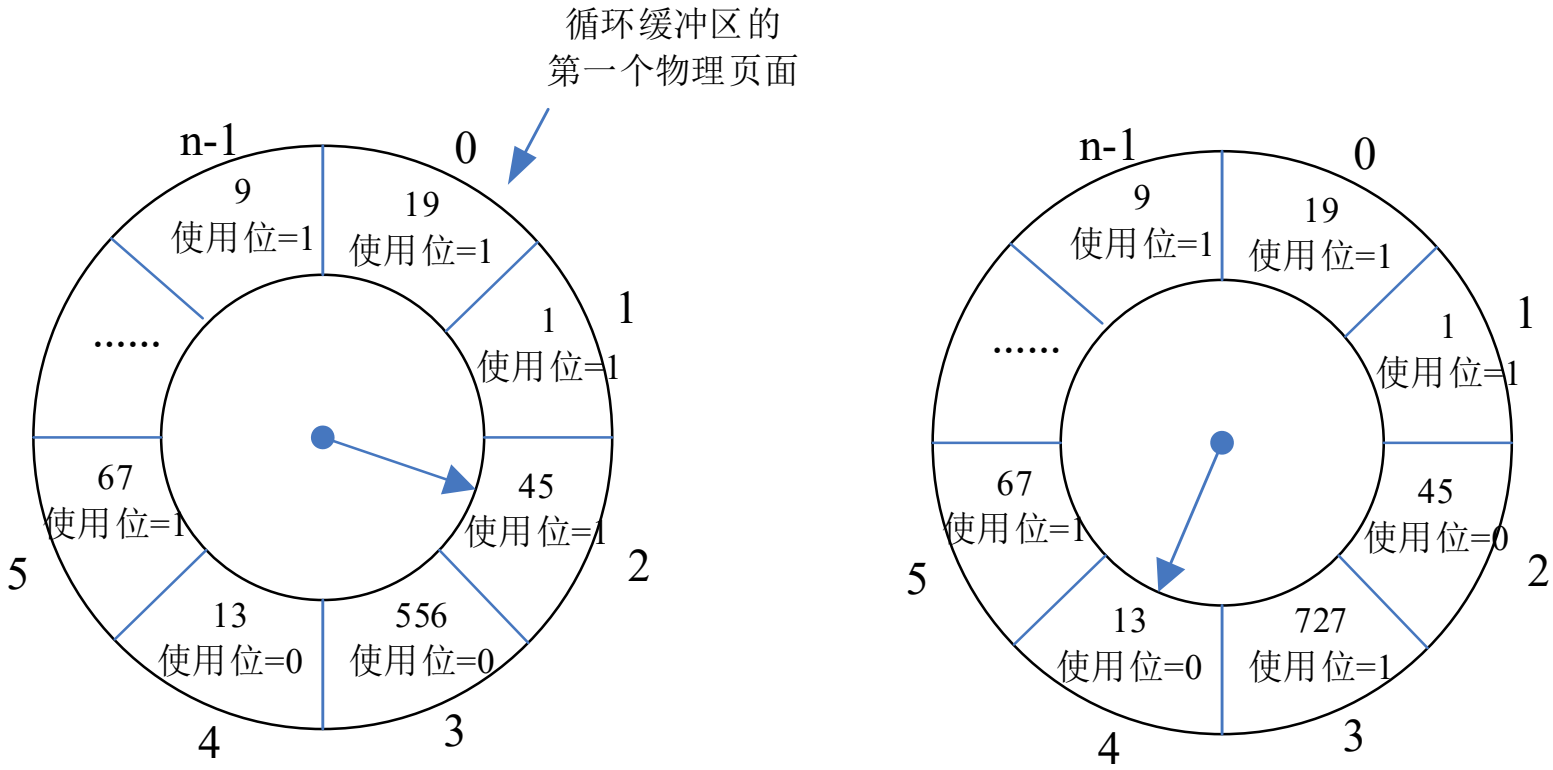
我们把分配给一个进程的物理页面的集合称为该进程的**驻留集**。

置换策略是在一个进程的驻留集中选择要被置换的页面。最基本的置换算法包括：

• LRU（Least Recently Used）：置换内存中最近一次使用距离当前最远的页。根据局部性原理，这也是最近最不可能访问到的页。实现LRU的一种方法是给每一页添加一个最后一次访问的时间戳，并且必须在每次访问时都更新这个时间戳。另一种方法是维护一个关于访问页的栈。

• FIFO：该策略把分配给进程的物理页面集合看做一个循环缓冲区，按先入先出的方式淘汰页面。该策略所隐含的逻辑是：置换驻留在内存中时间最长的页，即一个在很久以前取入内存的页，到现在可能已经不会再用到了。但是这个推断常常是错误的，因为一个很久前被取入内存的页，很可能最近被反复用到。比如，经常会出现一部分程序或数据在整个程序的生命周期中反复被使用的情况，如果使用FIFO算法，则这些页会反复的被换入和换出。

• CLOCK：时钟策略。


循环缓冲区的
第一个物理页面
n-1
0
1
2
3
4
5
9
使用位=1
19
使用位=1
1
使用位=1
45
使用位=1
556
使用位=0
13
使用位=0
67
使用位=1
.....
n-1
0
1
2
3
4
5
9
使用位=1
19
使用位=1
1
使用位=1
45
使用位=0
727
使用位=1
13
使用位=0
67
使用位=1
.....

【例5-9】假设为某进程分配了固定的3个物理页面。进程执行需要访问5个不同的虚拟页面，该程序运行时需要访问的虚拟页面序列（即页面趋势或走向）为：

2,3,2,1,5,2,4,5,3,2,5,2

求：使用LRU、FIFO和CLOCK置换策略时，物理页面集合的状态变化过程。

时间戳	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11
页面序列	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2(t0)	2(t0)	2(t2)	2(t2)	2(t2)	2(t5)	2(t5)	2(t5)	3(t8)	3(t8)	3(t8)	3(t8)
		3(t1)	3(t1)	3(t1)	5(t4)	5(t4)	5(t4)	5(t7)	5(t7)	5(t7)	5(t10)	5(t10)
				1(t3)	1(t3)	1(t3)	4(t6)	4(t6)	4(t6)	2(t9)	2(t9)	2(t11)
					F		F		F	F		
FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
					F	F	F		F		F	F
CLOCK	2*	2*	2*	2*	5*	5*	5*	5*	3*	3*	3*	3*
		3*	3*	3*	3	2*	2*	2*	2	2*	2	2*
				1*	1	1	4*	4*	4	4	5*	5*
					F	F	F		F		F	

表5-4 驻留集管理策略

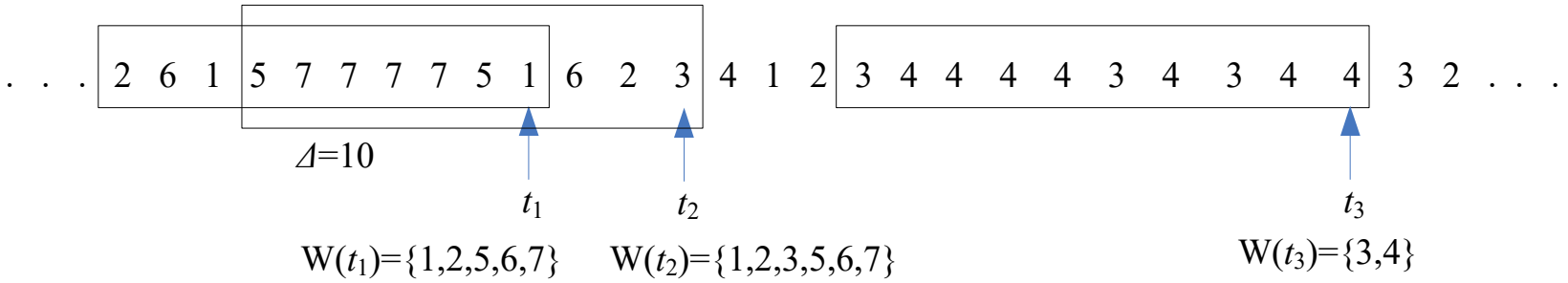
驻留集管理

所谓驻留集是指分配给一个进程的页框的集合。本节主要讨论两个问题：驻留集大小和置换范围。

	局部置换	全局置换
固定分配	● 进程的页框数固定 ● 从分配给该进程的页框中选择被置换的页	不存在
可变分配	● 分配给进程的页框数可以不断变化，用于保存该进程的工作集（Working set） ● 从分配给该进程的页框中选择被置换的页框	从内存中所有可用页框中选择被置换的页框；这导致进程驻留集的大小不断变化

无论是固定分配还是可变分配，都存在如何确定驻留集大小的问题。要解决这个问题，需要引入**工作集**（Working-set）的概念。

工作集$W(t, \Delta)$是一个页面的集合，表示在时间t时，最近Δ个页面访问所涉及的页面集合。如果一个页面最近是活动的，那么它将包含在工作集中；如果它最近不再被访问，那么从它最后一次访问后，经过Δ个时间单位，它将从工作集中消失。图5-30表示了一个页面访问序列以及Δ=10时的工作集。



工作集是程序局部性的一个近似。对于给定的工作集窗口长度Δ，如果工作集满足这样的性质：对于任意时间t，$|W(t, \Delta)| \leqslant R$，那么就意味着给该进程分配大小为$R$的驻留集就足够了。

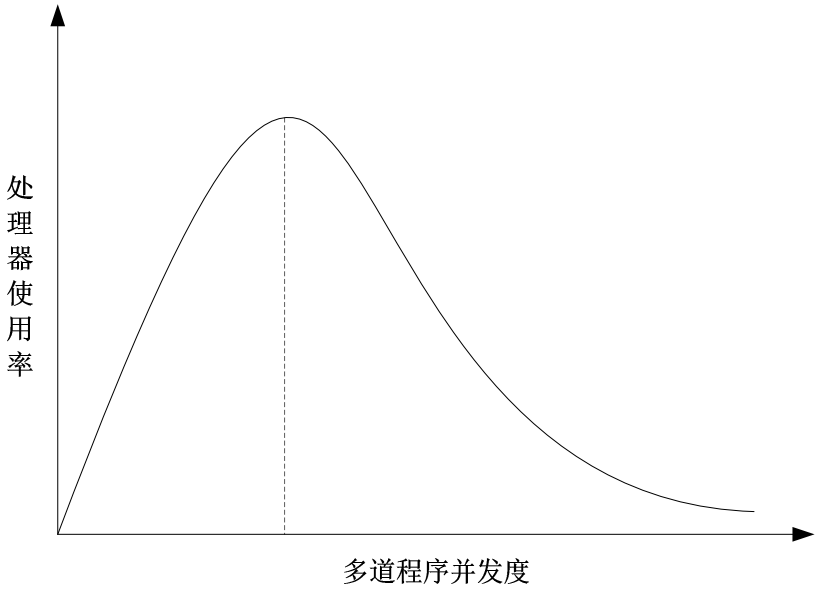
换出策略与读取策略正好相反，它用于确定何时将一个修改过的页面写回磁盘。通常有两种选择：

- **请求式**：只有当一个页面被选择用于置换时才被写回磁盘；
- **预先式**：将那些修改过的多个页在需要用到它们所占的页框之前成批的写回磁盘。

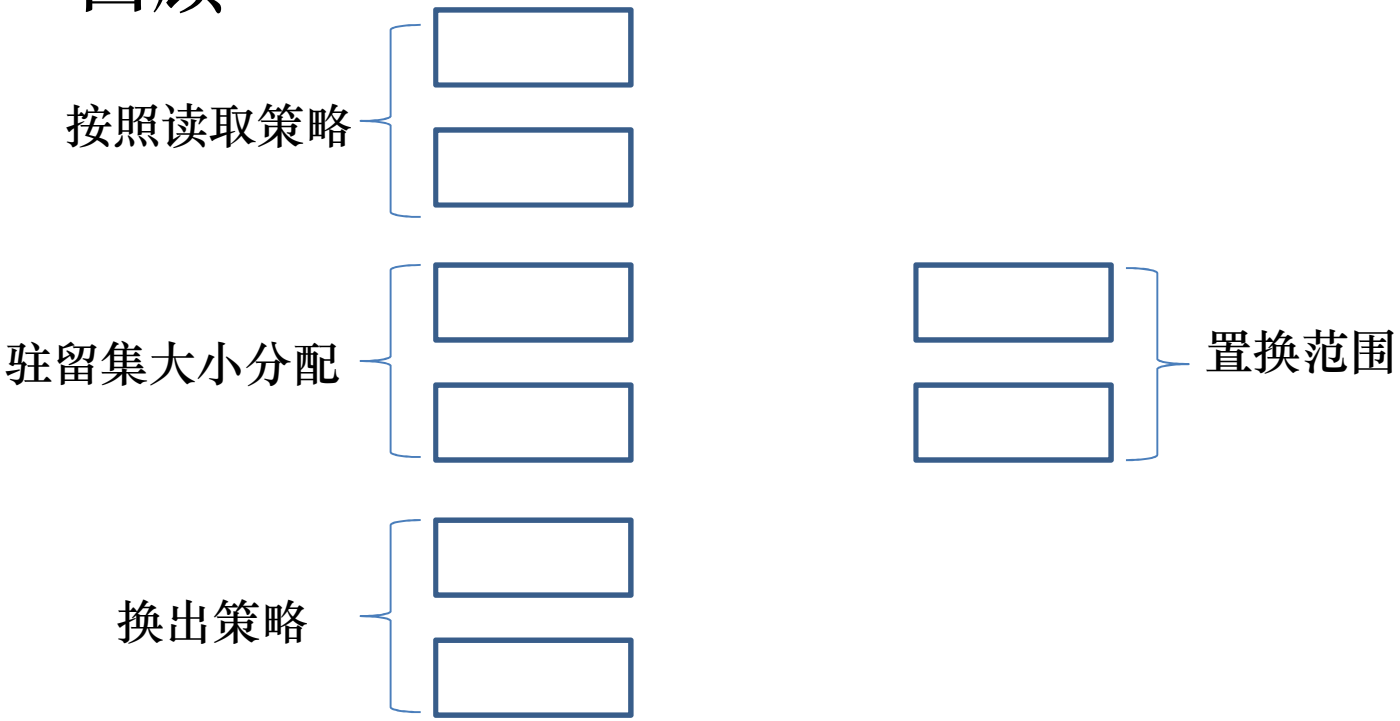
加载控制

加载控制决定驻留在内存中进程的数目，也就是系统的并发度。如果某一时刻，驻留的进程太少，所有进程同时处于阻塞状态的概率较大，因而会有较多时间花费在交换上；另一方面，如果驻留的进程太多，平均每个进程占用的内存较小，就会发生频繁缺页故障，从而导致抖动。

Denning等人提出了一种确定系统并发度的方法，称为*L*=*S*准则。他认为，当缺页故障之间的平均时间等于处理一次缺页故障的平均时间时，处理器的利用率达到最大。因此可以把缺页故障之间的平均时间和处理一次缺页故障的平均时间，作为动态调整并发度的依据。



回顾

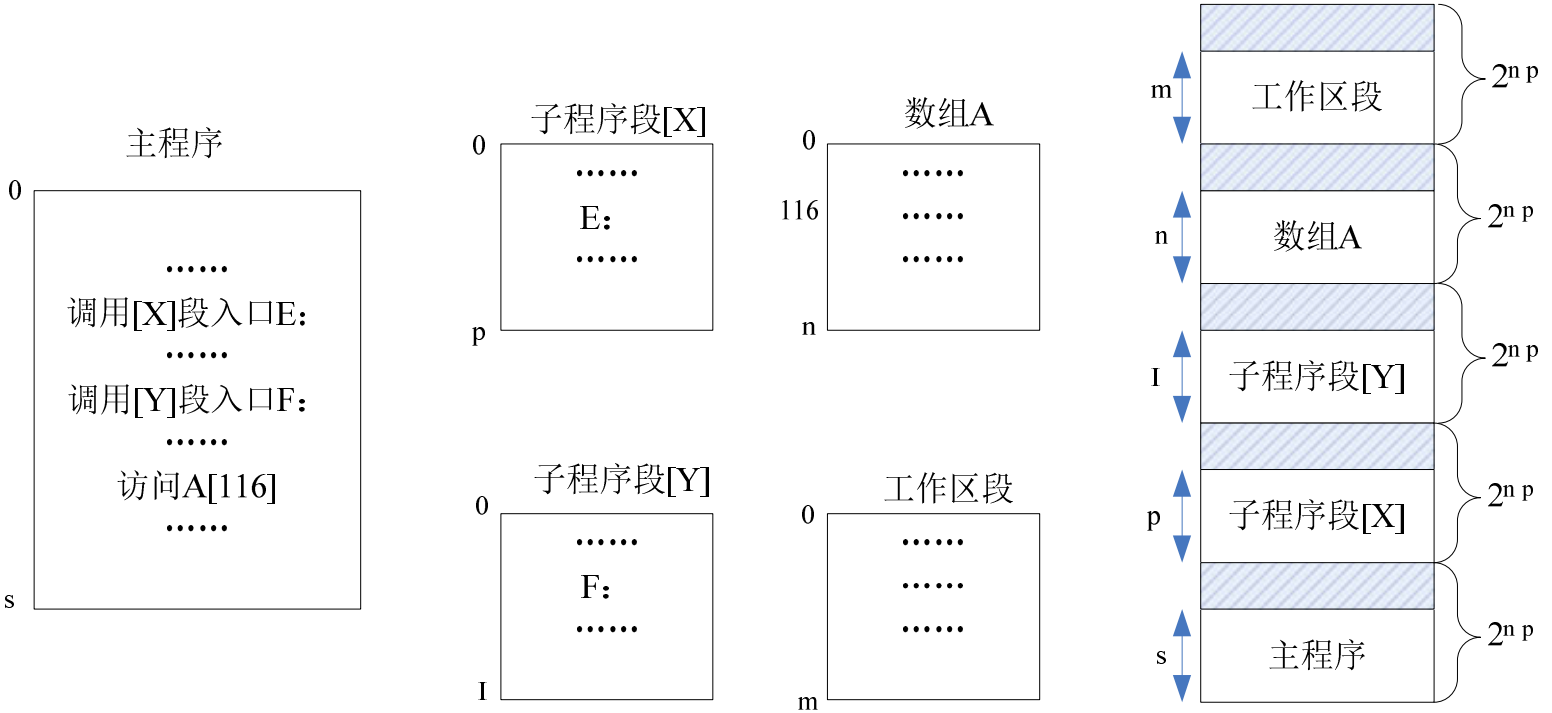


§5.5 分段式虚拟内存管理

早期的一些程序设计语言支持程序的分段，一个程序通常由若干段组成。分段式内存管理以段为单位进行存储分配，每一段都可从“0”开始编址，段内地址是连续的。通常使用

<段号，段内偏移>

的地址形式来访问段内的指令或数据 。



借助虚拟内存的概念，也可以把分段结构的程序统一用虚拟内存编排。与分页式虚拟内存不同，分段式虚拟内存以段为基本单位划分虚拟内存，每个段具有独立、完整的逻辑意义，并且对应程序中的一个模块。

虚拟地址结构确定以后，一个进程中允许的最大段数及每段的最大长度就确定了下来。

一般的，对于一个n位虚拟地址，如果段号占高p位，段内偏移占低$(n-p)$位，那么容许的最大段数为2^p，每段长度可达2^{n-p}字节。

与分页式管理类似，用段表来描述段在物理内存中的分配情况。每个段被缓存到物理内存中的一段连续区域，不同的段之间不要求连续存储。

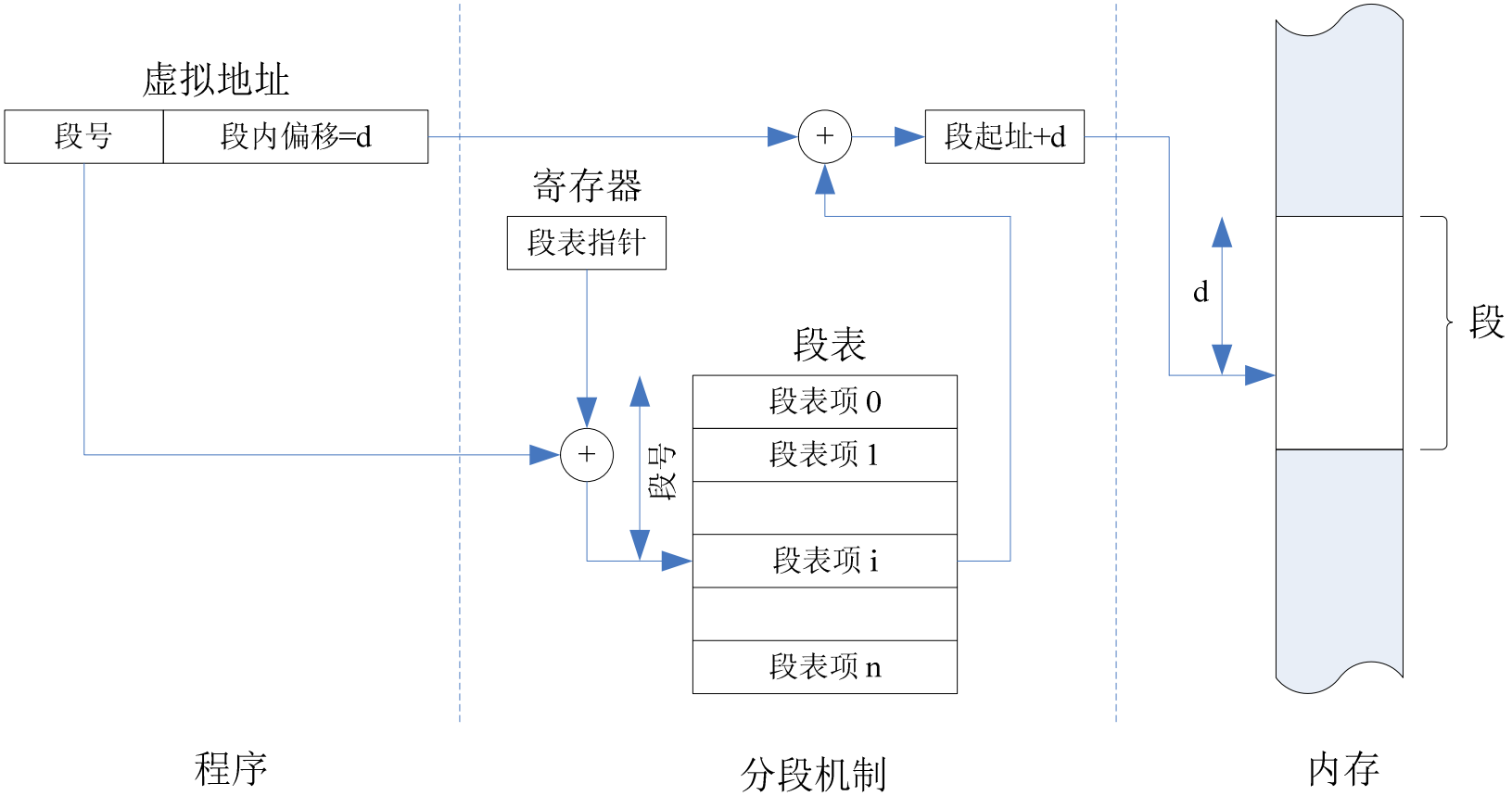
一个段表由若干段表项组成，每个段表项包括：标志位（标志一个段是否缓存在内存中）、段的内存起始地址、段长、段在磁盘等辅助存储器中的位置等，还可设置段是否被修改过、是否能移动、是否可扩充、是否共享等标志。段表格式如下：

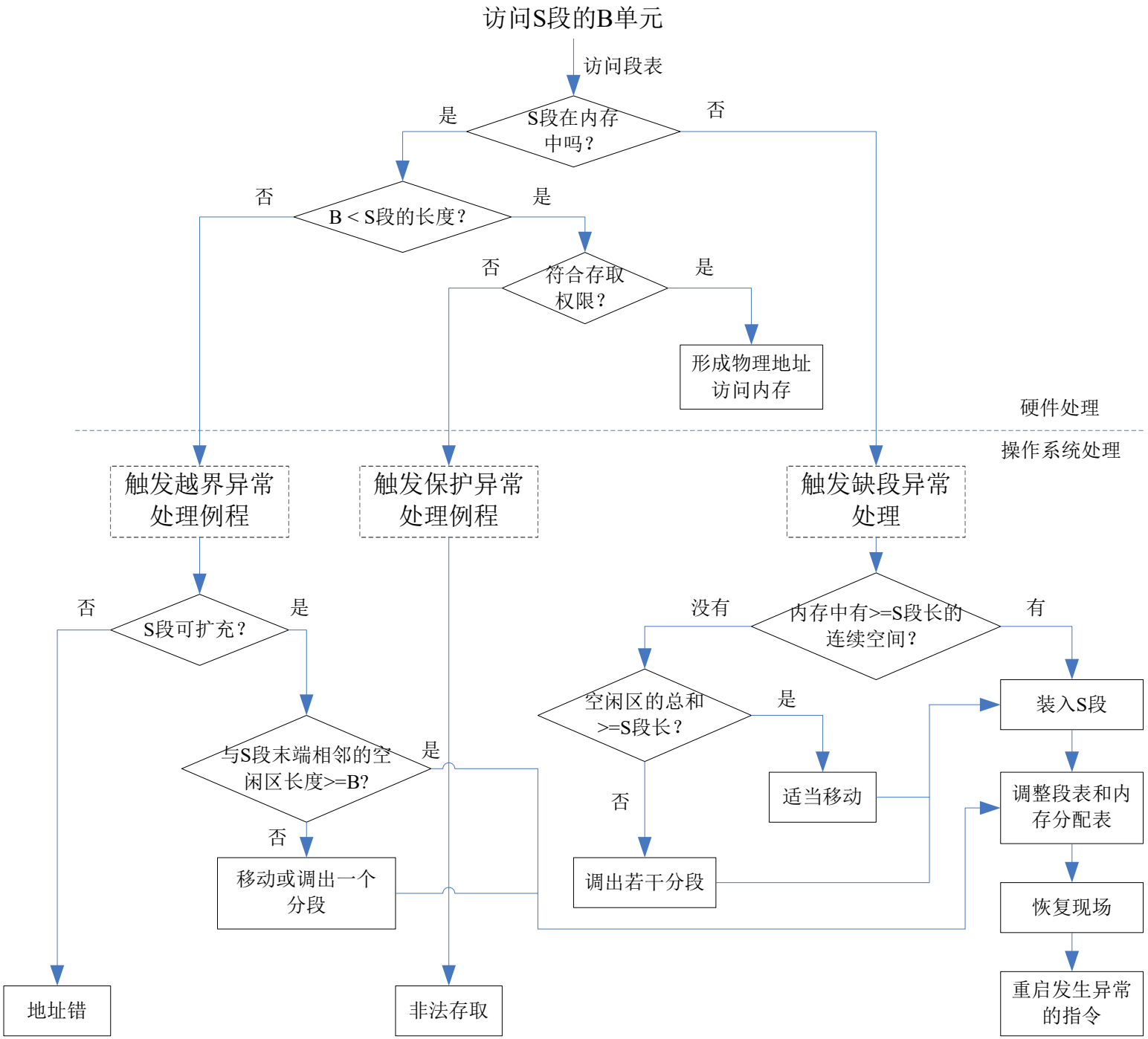
	标志位	存取权限	修改位	扩充位	内存起址	段长	辅存地址
0段							
1段							

与分页式内存管理类似，分段式内存管理也需要把一个虚拟地址转换为物理地址。对于一个形如

<段号，段内偏移>

的虚拟地址，硬件地址转换机构首先根据段号查找段表，若该段在内存中，则按照与分页管理类似的方法把虚拟地址转换为物理内存地址。




访问S段的B单元
访问段表
S段在内存中吗？
是
否
B < S段的长度？
否
是
符合存取权限？
否
是
形成物理地址访问内存
硬件处理
操作系统处理
触发越界异常处理例程
触发保护异常处理例程
触发缺段异常处理
S段可扩充？
否
是
与S段末端相邻的空闲区长度>=B?
是
否
移动或调出一个分段
地址错
非法存取
内存中有>=S段长的连续空间？
没有
有
空闲区的总和>=S段长？
是
否
适当移动
调出若干分段
装入S段
调整段表和内存分配表
恢复现场
重启发生异常的指令

分段式存储管理具有如下优点:

•简化对不断增长的数据结构的处理。如果程序员事先不知道一个特定数据结构的大小，除非允许使用动态的段大小，否则必须对其大小进行预估。而对于分段式虚拟内存，这个数据结构可以分配到它自己的段，需要时操作系统可以扩大或者缩小这个段的大小;

•允许程序段独立的改变或重新编译，而不要求整个程序集合重新链接和重新加载;

•有助于进程间的共享。程序员可以在段中放置一个实用工具程序或一个有用的数据表，供其他进程访问;

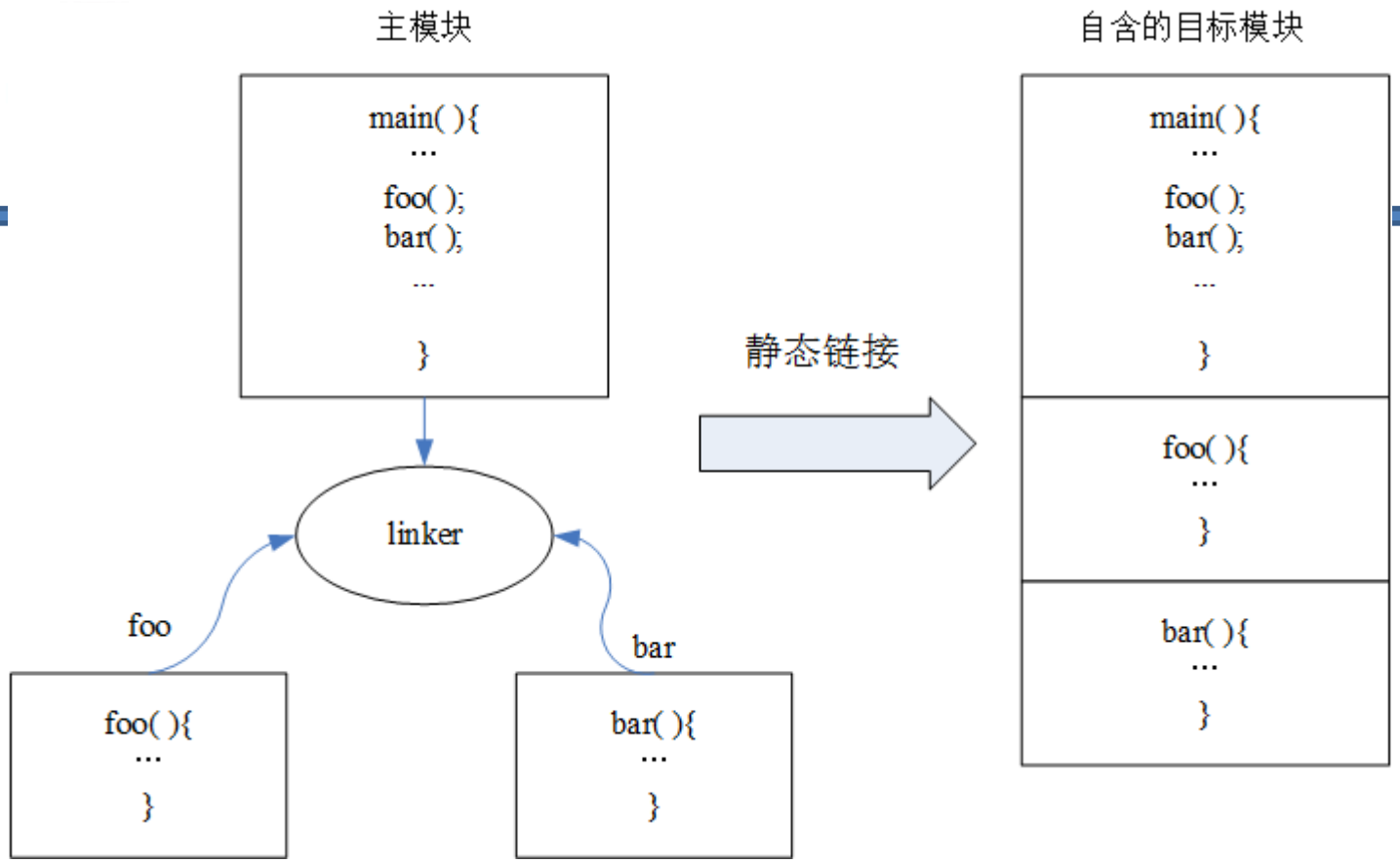
•有助于保护。由于一个段可以被构造成包含一个明确定义的程序或数据集，因而程序员或系统管理员可以更方便的指定访问权限。

5.5.2 段的动态链接

为了提高程序的结构化，一个应用程序通常由若干模块构成，每个模块独立编写和编译，形成可重定位目标文件。由于模块之间可能存在“定义-引用”关系，因此需要把一个目标文件中的符号引用与定义在另一个模块中的符号定义关联起来，这一过程就是程序的链接过程。

实现链接的方式有两种：

- 静态链接
- 动态链接

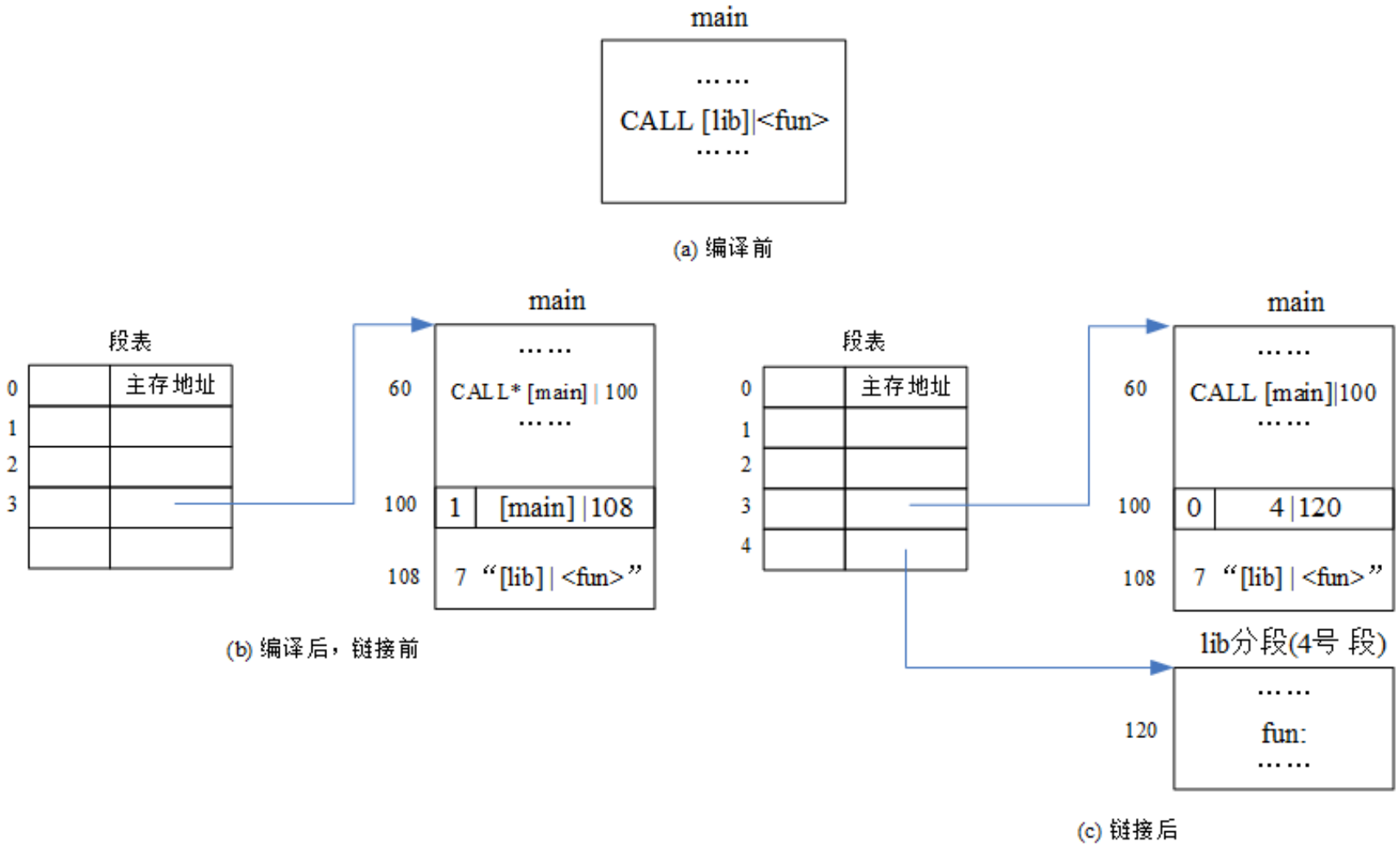


静态链接的优点：所有符号引用在汇编时已经得到解析

缺点：

- 公用库被链接进每一个目标模块，浪费了磁盘空间和内存空间
- 一个大型程序在运行时，并不一定要调用所有的子程序（如某些出错处理程序）或访问所有的数据，如果把它们静态链接进可执行程序，将会占用很大的内存空间。
- 模块一旦被静态链接后，之后的修改必须重新编译和链接

动态链接将链接过程推迟到程序加载时或程序运行时。如果在加载时已经知道程序要用到的动态库，那么就在程序加载时加载这些动态库并链接；如果只有在运行时才能知道需要用到的动态库，那么就只能在运行时加载并链接这些动态库。



大家回想一下程序中如何加载一个动态库？如何获取动态库中的一个函数地址？

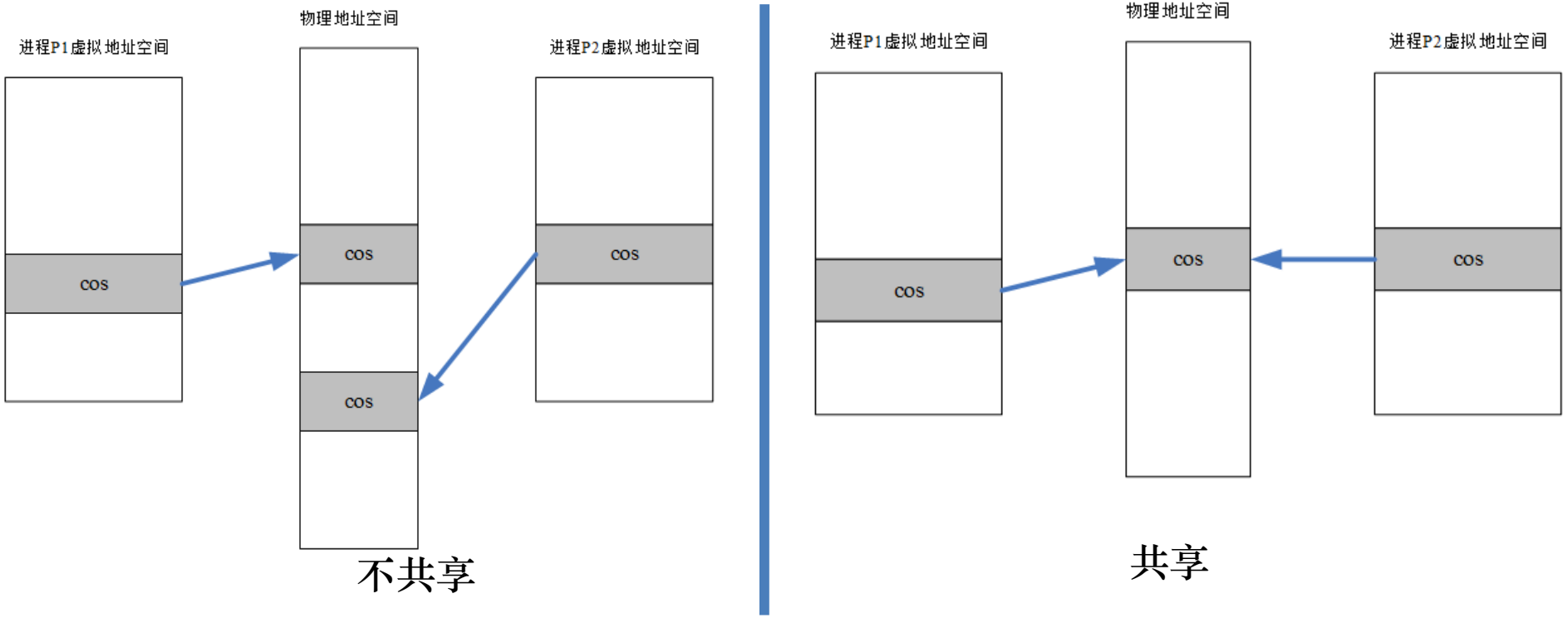
Handle h=LoadLibrary (“lib.dll”);

FunPtr fp=(FunPtr)GetProcAddress (h, “fun”);

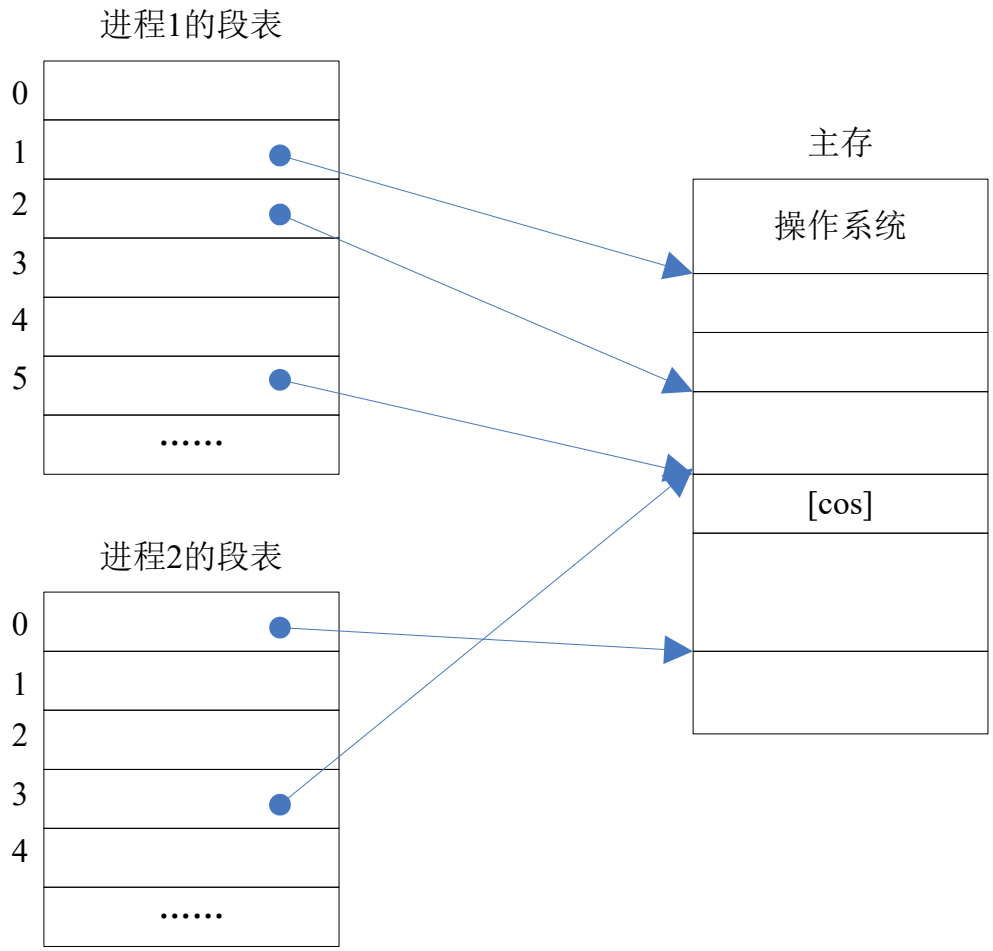
5.5.3 段的共享

所谓“段的共享”是指一个代码段或数据段在内存中只存在一个副本，而且可以被多个进程同时访问。

进程1和进程2都需要使用公共子程序cos。



- 如何实现段的共享？——使用段表

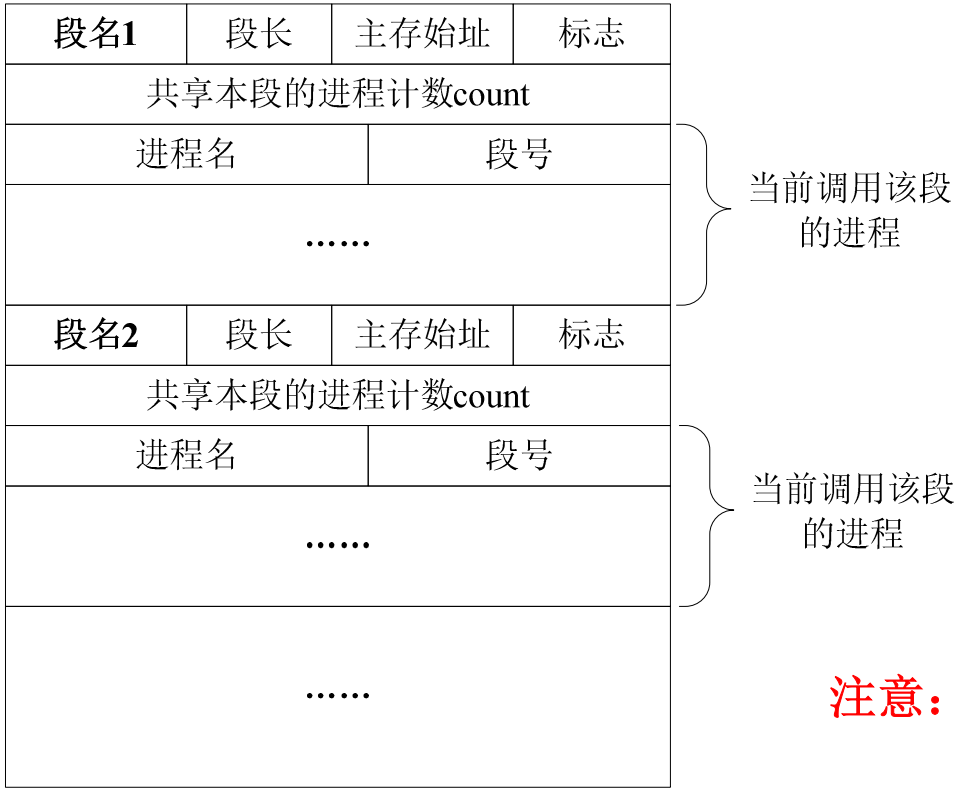


问题：

1、既然cos段被P1和P2共享，那么会不会出现并发控制问题？

2、P1和P2执行cos段时，会不会相互干扰对方的执行？

为了对共享段进行管理，除了段表之外，还需要设置一张“共享段表”来协助管理共享段。共享段表记录每个共享段的段名、在/不在主存、在主存时指出它在主存的起始地址和长度、调用该段的进程数、进程名和进程定义的段号等。



考虑下面几个过程：

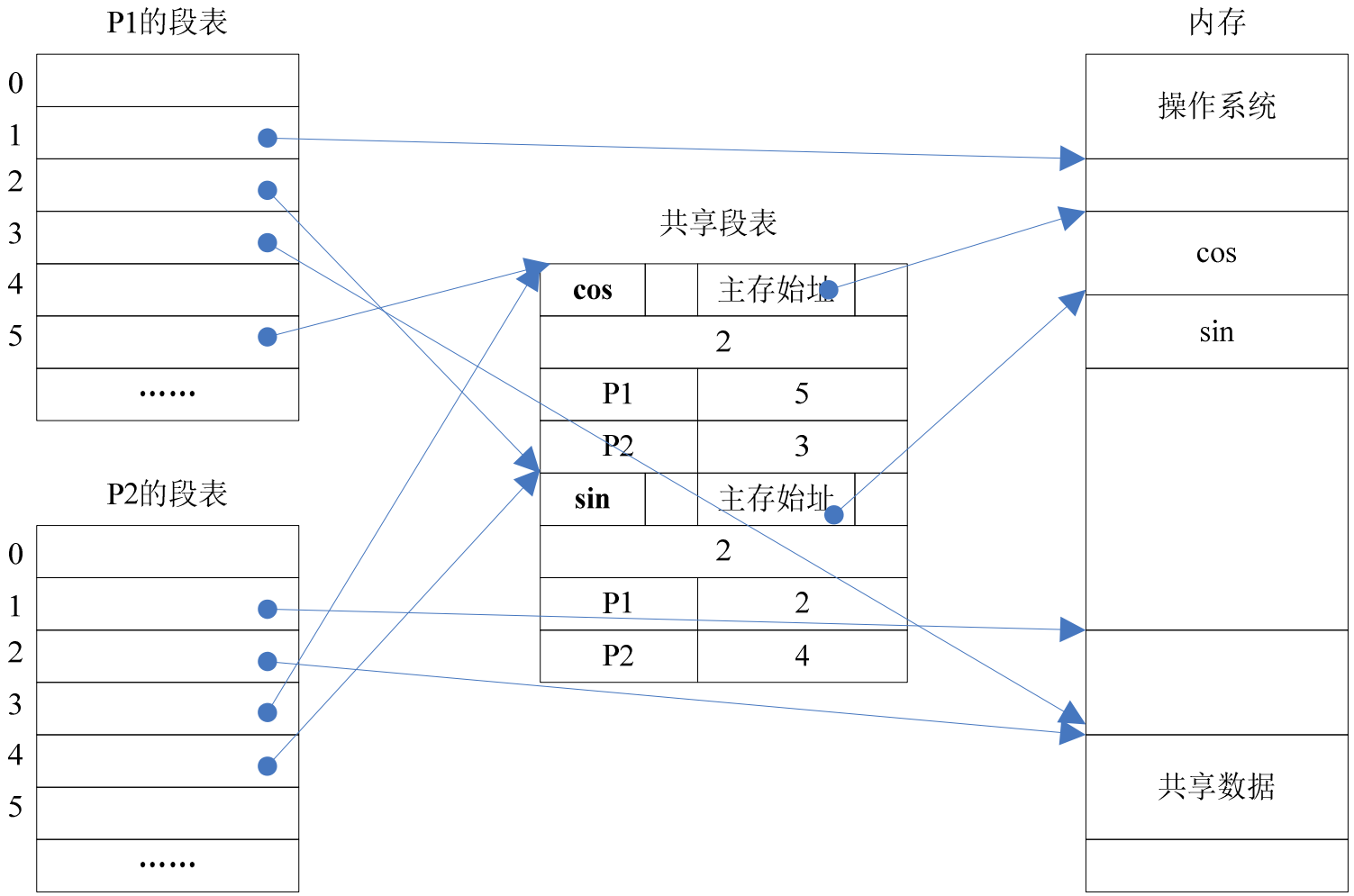
1、当第一次访问一个共享段时

2、当一个进程访问一个已加载的共享段

3、当一个进程卸载一个共享段时

4、当一个共享段的计数为0时

注意：段表和共享段表的状态要同时变化


P1的段表
0
1
2
3
4
5
……
P2的段表
0
1
2
3
4
5
……
共享段表
cos
主存始址
2
P1
5
P2
3
sin
主存始址
2
P1
2
P2
4
内存
操作系统
cos
sin
共享数据

- 习题

P191　　4、14、15、17