第五章 存储管理

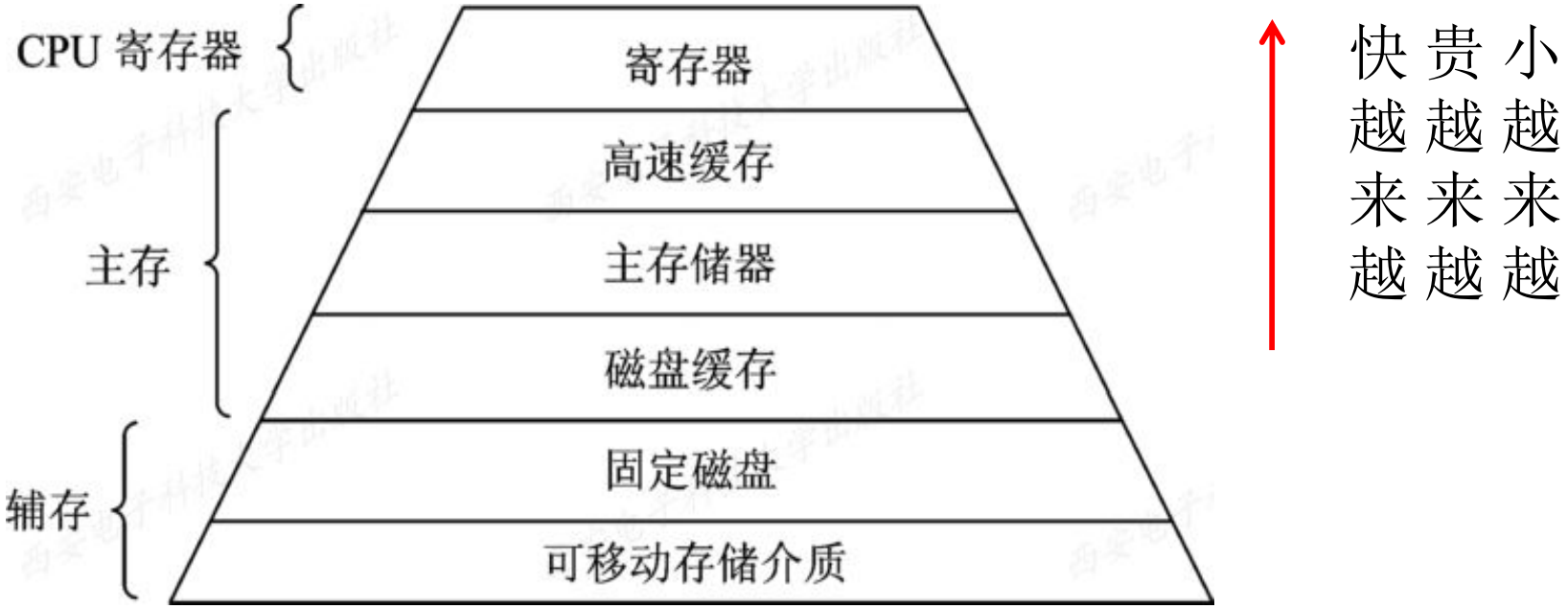
软件称为计算机的灵魂，存储器则是灵魂展现活力的舞台，栖身的场所，也是与人交互信息的通道。

“64K内存空间即可满足所有人的内存空间。”__比尔·盖茨

内存是CPU直接存取指令和数据的存储器。任何一个程序（包括应用程序和OS本身）必须被装入内存，才可能被执行。尽管RAM芯片集成度越来越高，价格不断降低，由于其需求量大，整体价格仍较昂贵，而且受CPU寻址能力的限制，内存容量仍有限。

因此，对主存的管理和有效利用仍然是当含今操作系统十分重要的内容。内存区域被分为两大区域：系统空间，用户进程空间。本章主要讲述用户区域的管理方法和基本技术。

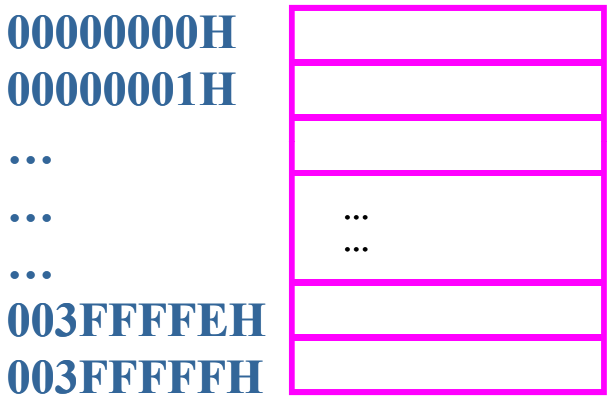
5.1 存储管理基本概念



5.1.1　物理内存和虚拟存储空间

物理地址

1. 物理内存(主存储器)



5-1　5-1 4MB物理地址空间

2．虚拟存储空间

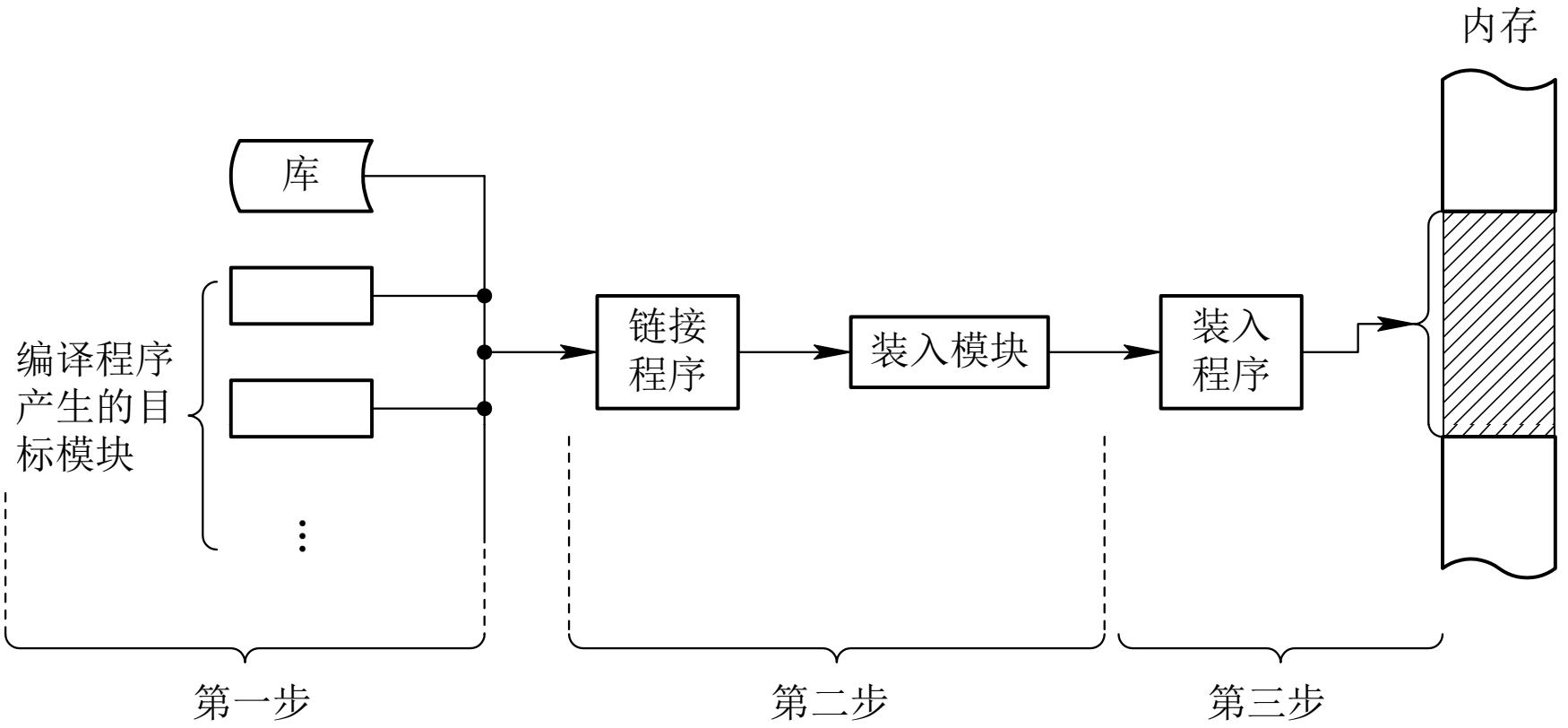
在多道程序环境下，要使程序运行，必须先为之创建进程。而创建进程的第一件事，便是将程序和数据装入内存。

如何将一个用户源程序变为一个可在内存中执行的程序?通常都要经过以下几个步骤：

(1)编译，由编译程序(Compiler)将用户源代码编译成若干个目标模块(Object Module)；

(2)链接，由链接程序(Linker)将编译后形成的一组目标模块，以及它们所需要的库函数链接在一起，形成一个完整的装入模块(Load Module)；链接既可以是在程序执行前由链接程序完成（静态链接），也可以是在程序执行过程中由于需要而进行的（动态链接）；

(3)装入，由装入程序(Loader)将装入模块装入内存。下图示出了这样的三步过程。


内存
库
编译程序
产生的目
标模块
链接
程序
装入模块
装入
程序
第一步
第二步
第三步

程序在运行中要访问的内存地址应该怎样给出？

（1）程序员在程序中直接给出要访问的数据或指令的物理地址。

（2）编程时源程序使用的都是符号地址，（GOTO A等），用户不必关心符号地址在内存中的物理位置。源程序经过编译链接后形成一个以0地址为起始地址的虚拟地址空间。每条指令或数据单元都在这个虚拟地址空间中拥有确定的地址，我们把这个地址称为虚拟地址（virtual address）或相对地址。程序运行时访问的内存物理地址由地址装入模块的地址变换机构完成。

我们将进程中的目标代码、数据等的虚拟地址组成的虚拟空间称为虚拟存储空间。（virtual memory）

虚拟地址空间不考虑物理内存的大小和信息存放的实际地址，它只考虑相互关连的信息之间的相对位置，其容量只受计算机的寻址方式和地址结构限制。每个进程拥有自己的虚拟空间。

采用虚拟存储空间技术使得用户程序空间和内存空间分离开，从而使用户程序不受内存空间大小的限制，为用户提供一个比实际内存更大的虚拟空间。可以为用户提供比实际内存更大的空间(虚拟存储器）。

5.1.2　存储管理的主要任务

1. 内存的分配与回收

2．地址变换

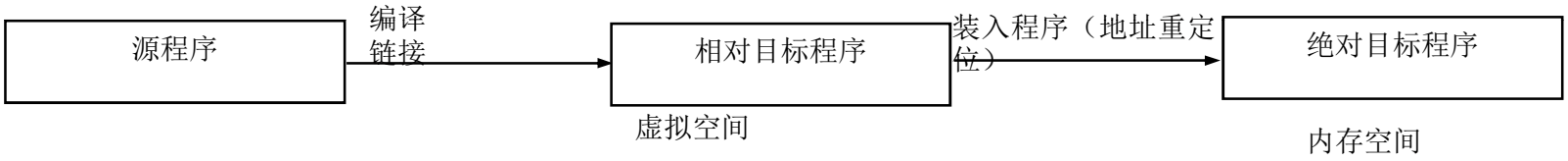
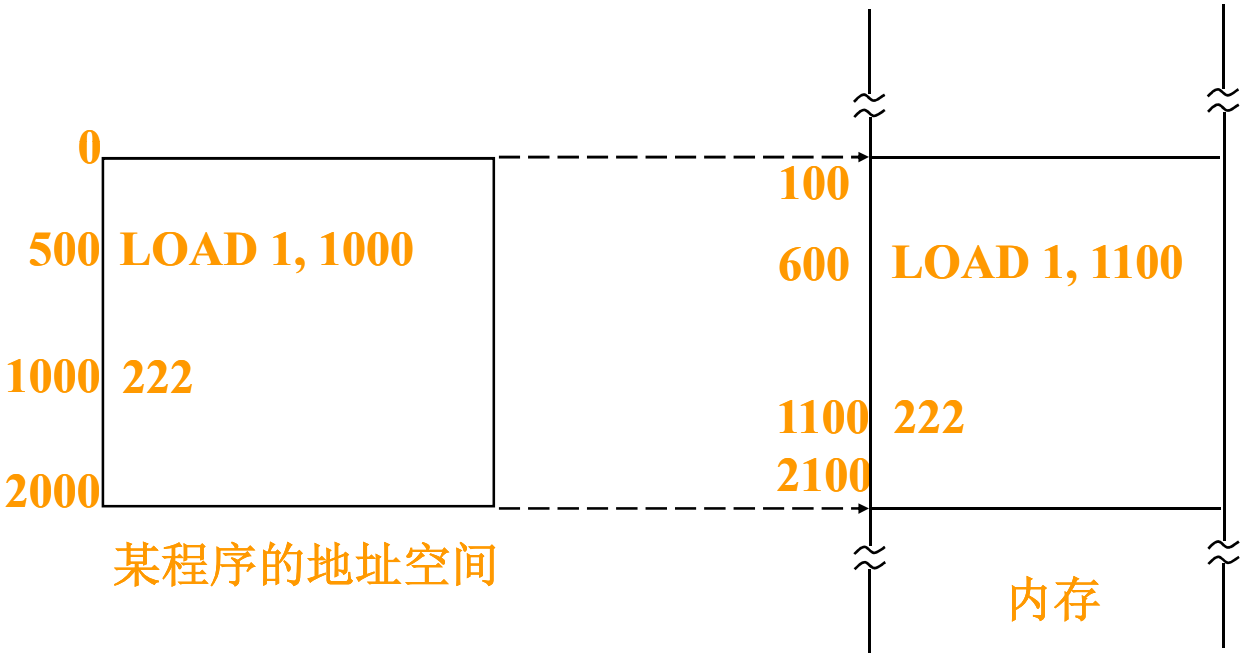
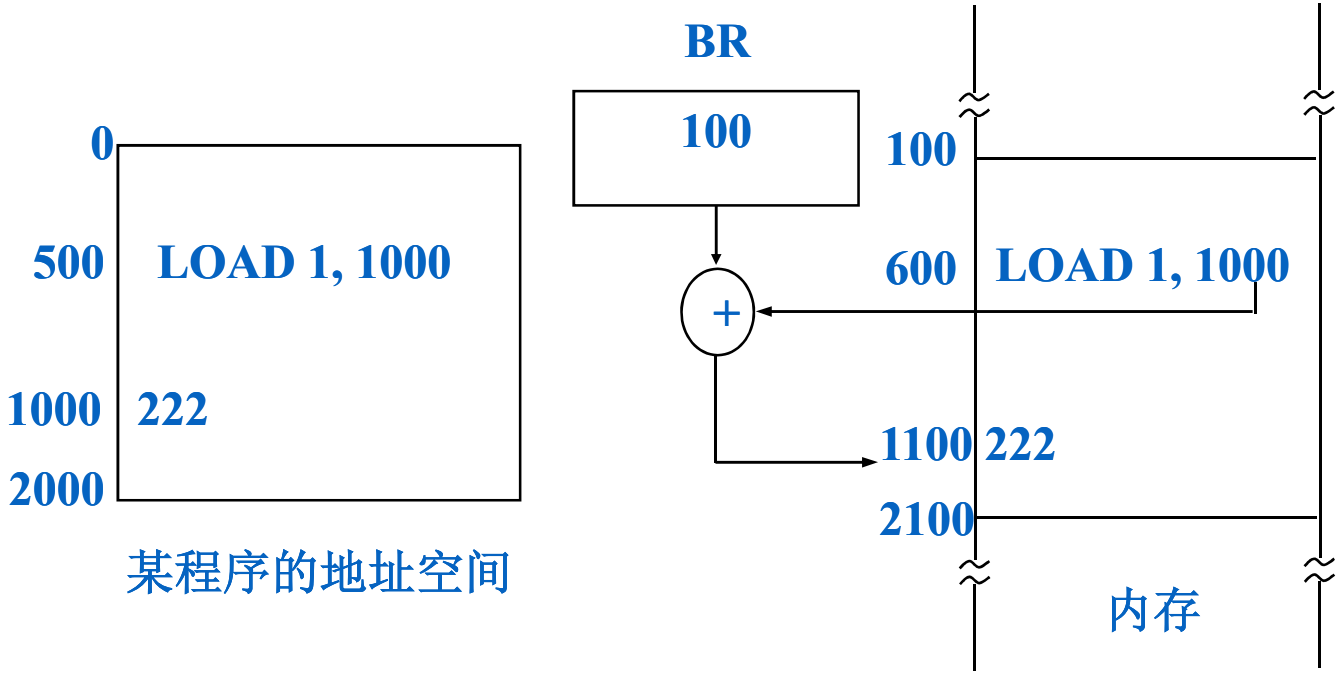


图　　地址变换与存储空间

（1）静态地址重定位：在目标程序装入指定的区域时由装配程序完成地址的变换。例：假设目标程序分配的内存起始地址为BA，目标程序中的某条指令或数据的虚拟地址为VA，则其对应的物理地址是MA＝BA＋VA。



（2）动态地址重定位：**指程序在执行的过程中，在CPU访问内存地址之前，由地址变换机构（硬件）来完成要访问的指令或数据的逻辑地址到物理地址的变换。**地址变换机构通常由一个公用的基址寄存器BR和一个（或多个）虚拟地址寄存器VR组成。只要改变BR的内容，就可以改变程序在内存的存放空间。



动态地址重定位的优点：

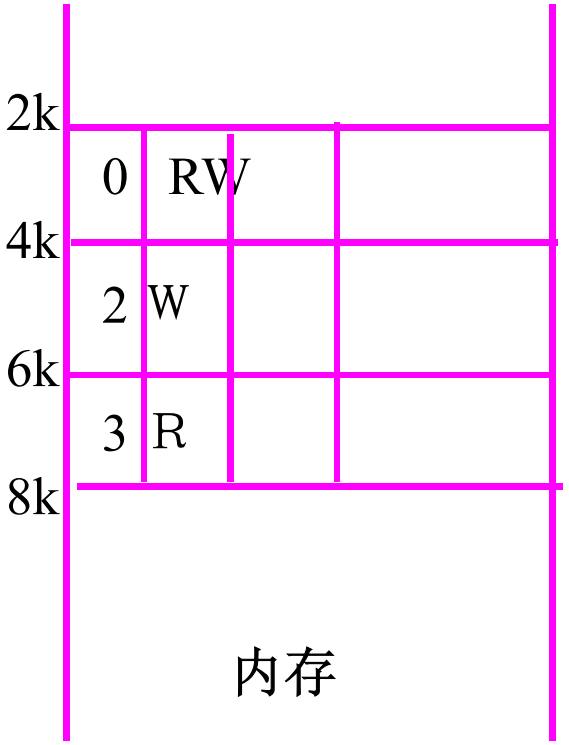
（1）目标模块装入内存时无需修改，且装入后可以方便的进行搬迁。

（2）一个程序由多个相对独立的模块组成是，每个目标模块可以各自装入一个不相邻的存储区域中。

3. 内存信息的共享和保护

常用的保护方法：上下界保护法；保护键法；

如：当前的程序状态字为2，R允许读，W允许写



合法的访问：
LOAD 1,5000
STORE 2,5200
LOAD 2,7000

非法的访问：
LOAD 1,2500 读写保护
STORE 1,7000 写保护

4. 内存扩充（虚拟存储技术）

5.2 分区式存储管理

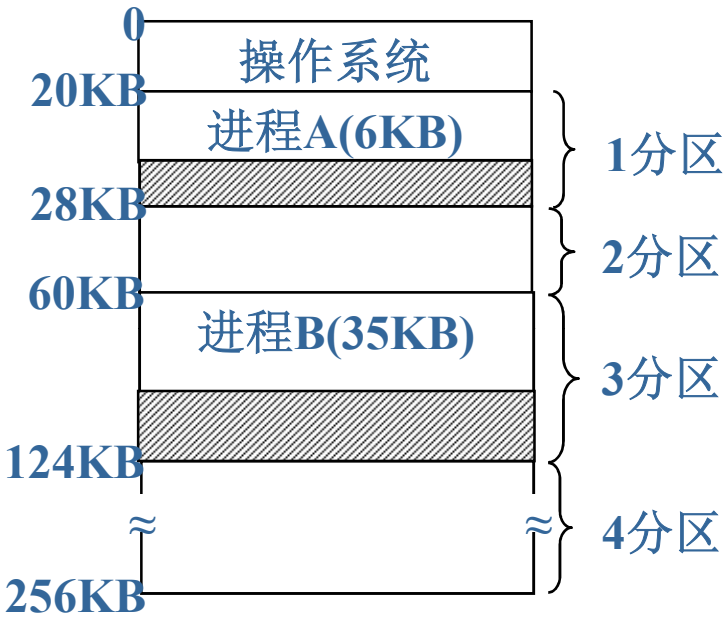
分区的基本思想是将内存区域划分成若干个大小不等的区域，每个区域称为一个分区，每个分区存放一道进程对应的程序和数据，使进程在内存中占用一个连续的区域，而且进程只能在所在分区内运行。

5.2.1 固定式分区（静态分区）

固定分区：个数、大小不变。

数据结构：分区说明表

区号	分区长度	起始地址	状态
1	8KB	20KB	1
2	32KB	28KB	0
3	64KB	60KB	1
4	132KB	124KB	0



优缺点：技术简单；但主存利用率不高，存在严重的内碎片。

5.2.2 可变式分区（动态分区）

(a) 某时刻状态　(b)加入进程D　(c) 撤销进程A和C

1. 分区大小、个数变化。
2. 数据结构：已分配分区表（大小、起址）和空闲分区表（链）（大小，起址，状态）。外碎片，拼接（内存紧凑）

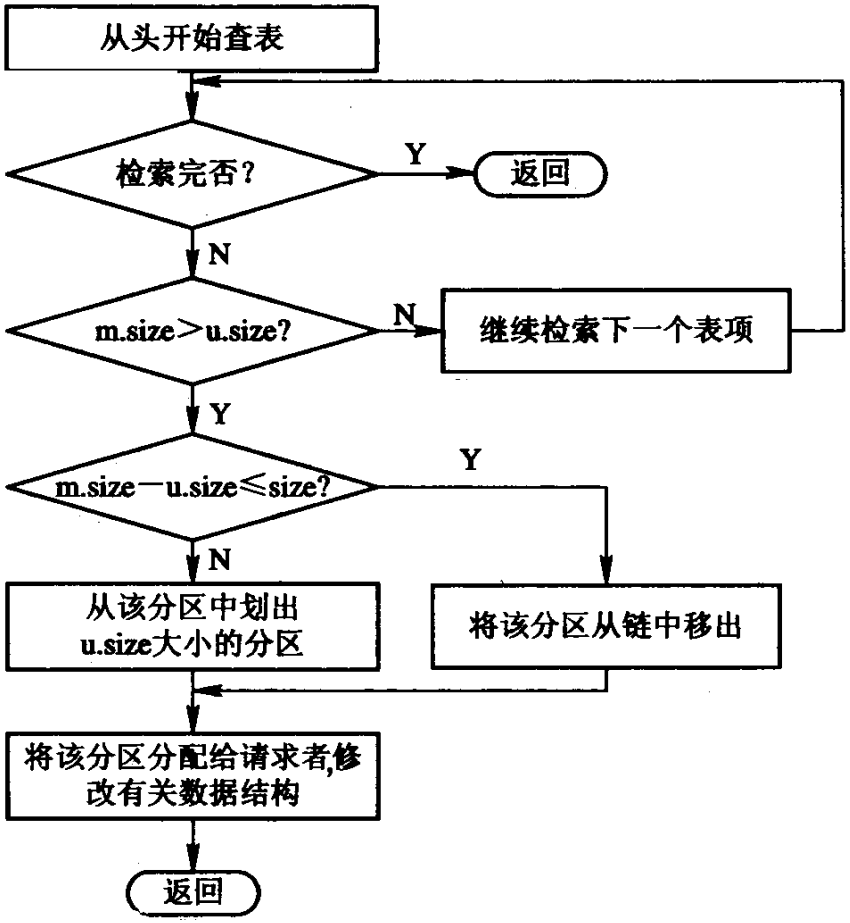

从头开始查表
检索完否？
Y
返回
N
m.size＞u.size?
N
继续检索下一个表项
Y
m.size－u.size≤size?
Y
N
从该分区中划出
u.size大小的分区
将该分区从链中移出
将该分区分配给请求者,修
改有关数据结构
返回

图5-8 内存分配流程

（1）首次适应算法。该算法要求空闲分区链以地址递增的次序链接，从链首顺序查找，找到能满足的空闲区，划分为分配区和空闲区。保留了高址部分的大空闲区。低地址部分不断被化分，会留下许多难以利用的、很小的空闲分区。

（2）循环首次适应算法。在为进程分配内存空间时，不再是每次都从链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，直至找到一个能满足要求的空闲分区，从中划出一块与请求大小相等的内存空间分配给进程。这样会缺乏大的空闲分区。

（3）最坏适应算法。最坏适应算法要求空闲分区按其容量以从大到小的顺序形成一空闲分区链。进程分配后剩下的空闲区也比较大，也能装下其它的进程。但是当有大的进程到来时，其申请的存储空间往往不能得到满足。

（4）最佳适应算法。所谓“最佳”是指每次为进程分配内存时，总是把能满足要求、又是最小的空闲分区分配给进程，避免“大材小用”。为了加速寻找，该算法要求将所有的空闲分区按其容量以从小到大的顺序形成一空闲分区链。因为每次分配后所切割下来的剩余部分总是最小的，这样，在存储器中会留下许多难以利用的小空闲区。

（5）快速适应算法。分类搜索法，每一类具有相同容量的空闲区设立一个空闲分区链表，每个链表的表头指针放在一个索引表中。优点是，分配时不会对分区分割，保留大的分区，不会产生碎片，查找效率高。缺点是，合并算法复杂。

（6）伙伴系统。把内存中的所有空闲分区按照$2^k(1\leq k\leq n)$的大小划分，划分后形成了大小不等的存储区，所有相同的空闲分区形成一个链。在某进程请求分配m 大小的内存时，首先计算i，使$2^{i-1}\leq m\leq 2^i$，然后在2^i 空闲分区链中搜索。若空闲分区链不为空，分配一个空闲分区给请求的进程。若没有，则查询2^{i+1}空闲分区链，若有2^{i+1}大小的空闲分区，则把该空闲分区分为两个相等的分区，这两个分区称为一对伙伴。伙伴必须是从同一个大分区中分离出来的，其中一个用于分配，另一个加入2^i 的空闲分区链中。若大小为2^{i+1}的空闲分区也为空，则需找大小为2^{i+2}的空闲分区，在找到可利用的空闲分区后，则进行两次分割，一个用于分配，一个加入2^{i+1}的空闲分区链中，一个加入2^i 的空闲分区链中，以此类推。最大限度的解决了内存分配引起的外部碎片问题。

（7）哈希算法：构建一张以空闲区大小为关键字的哈希表，该表的每一个表项记录了一个对应的空闲分区链表头指针。

4.内存回收

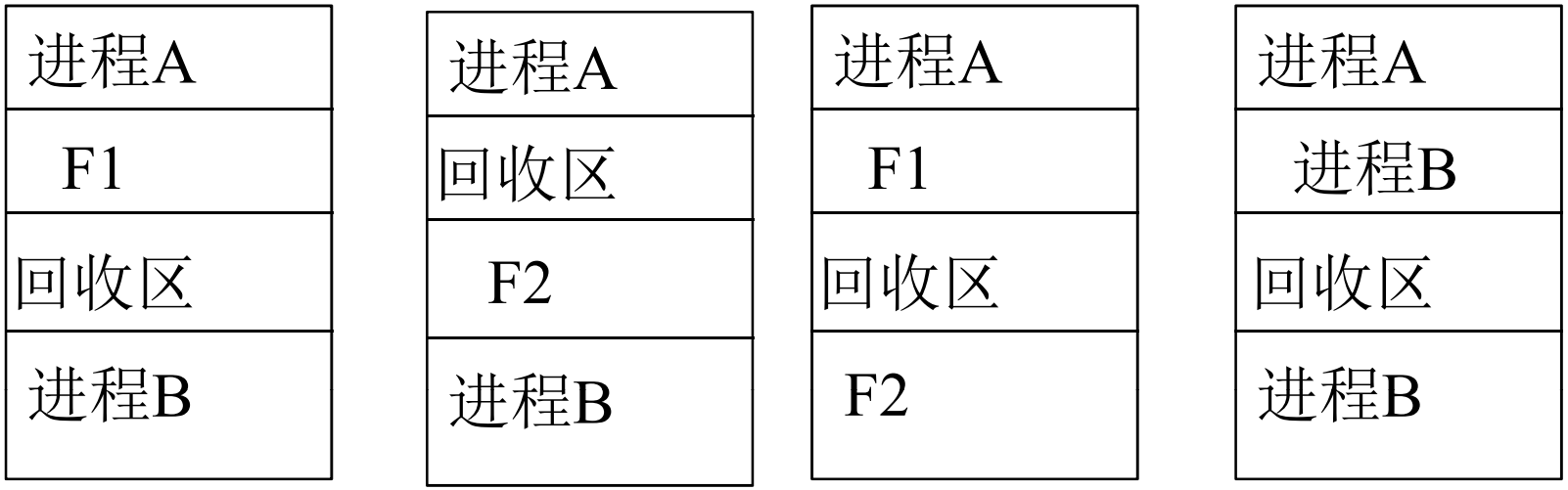


图5-9 内存回收情况

5.2.3 地址变换与内存保护

基址寄存器BR和限长寄存器LR。假设CPU要访问的逻辑地址LA，若LA＞LR，则说明地址越界，将产生保护性地址越界中断，系统转出错处理。若LA＜＝LR，则LA与BR中的基址形成有效的物理地址PA。

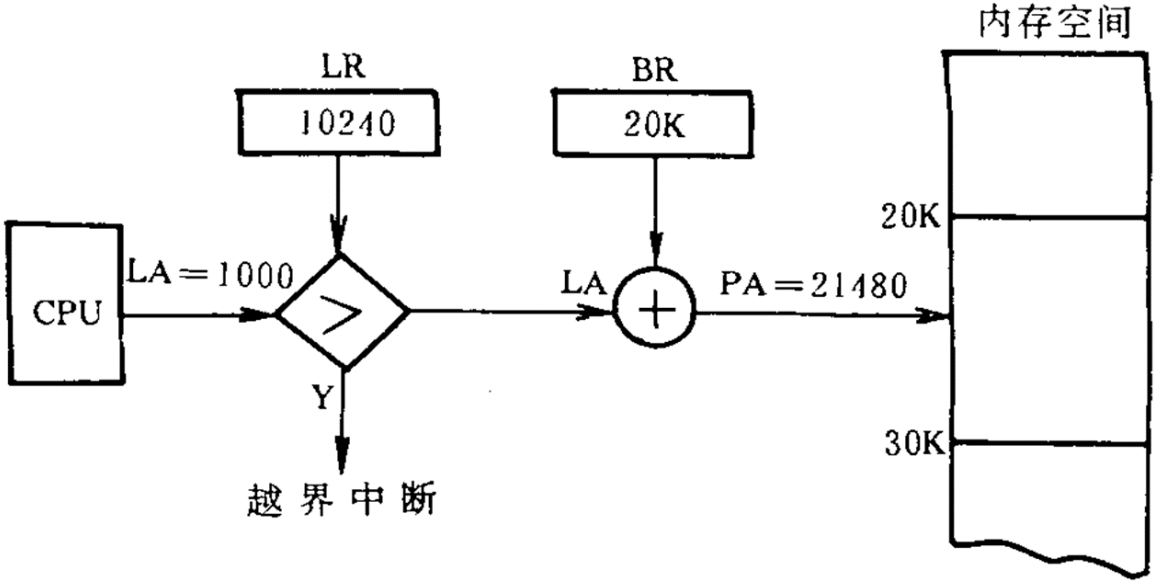


图5-10 分区管理的地址变换和保护

5.2.4 分区式管理的优缺点

主要优点：

（1）实现了多个作业或进程对内存的共享，有助于多道程序设计，从而提高了系统的资源利用率。

（2）该方法要求的硬件支持少，管理算法简单，因而实现容易。

主要缺点：

（1）内存利用率仍然不高。存在着严重的碎小空闲区（碎片）不能利用的问题。内存紧凑。

（2）作业或进程的大小受分区大小控制，除实现内存的扩充。非配合采用覆盖和交换技术来

（3）难以实现各分区间的信息共享。

5.3 页式存储管理

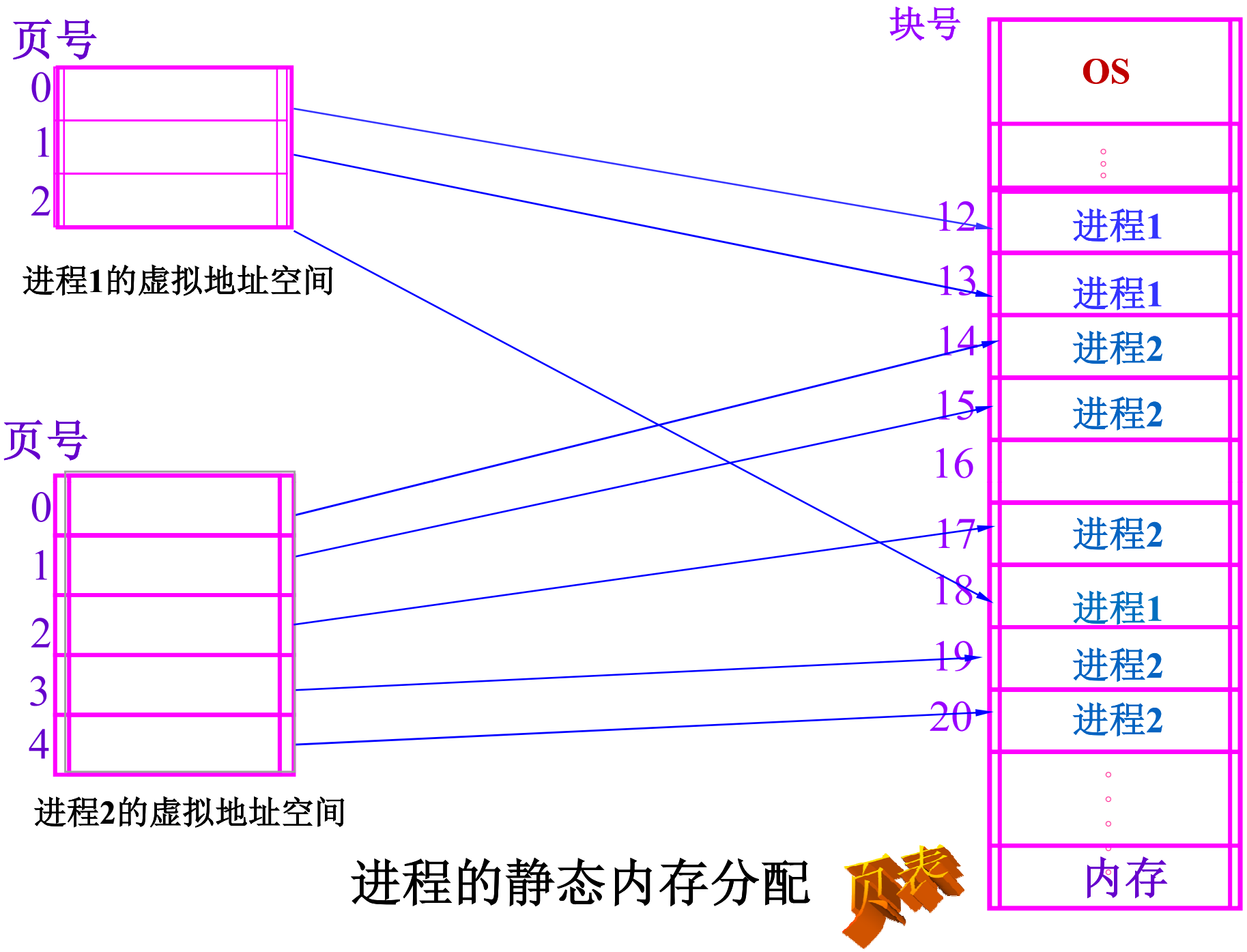
在分区式存储管理方案中，一个进程总占用一块连续的内存区域，因此产生了“零头”问题。虽然采用拼接技术可以使零散的“零头”连成一片，但要花费大量的处理机时间。为了更好地解决零头问题，人们提出了另一种内存存储管理方案，即分页式存储管理方案。该管理方案将一个进程存放在不连续的内存区域中。有**静态页式管理和动态页式管理。**

5.3.1 静态页式管理

1. 基本思想

系统首先把内存的存储空间等分成若干个大小相等的小区域，每个小区域称之为页面或内存块。页面按0、1、2、3 …依次编号。同样，进程的虚拟地址空间也被分成若个与页面大小相等的多个片段，称之为页，编号为0、1、2、3 … 。分配内存时，进程的每一个页装入内存的一个存储块。同一进程的多个页可以分配在页号不连续的存储块内。

2. 分配与回收


页号
0
1
2
进程1的虚拟地址空间
页号
0
1
2
3
4
进程2的虚拟地址空间
块号
OS
12
13
14
15
16
17
18
19
20
进程1
进程1
进程2
进程2
进程2
进程1
进程2
进程2
内存
页表

进程的静态内存分配

3. 数据结构

（1）页表“一个进程往往有多个页，而这些页在内存可以不连续存放。那么，系统是如何知道进程的每一页分别存放在内存的哪一个存储块呢？分散存放在内存中的进程又是如何正确运行的呢？ 为此，在分页式管理方案中，系统通过引入页表来完成地址变换，**对每一个进程，系统为它建立一个页表。**

（2）请求表：页表始址、请求块数、页表长度

（3）存储页面表

19	18	17	16	15	.....	4	3	2	1	0
0	1	1	1	1	.....	1	1	0	1	1
0	0	0	1	1	.....	0	0	1	1	0
0	0	1	1	1	.....	0	0	0	0	1

页表如下图所示：

页号	块号
0	12
1	13
2	18

进程1页表

页号	块号
0	14
1	15
2	17
3	19
4	20

进程 2页表

作用：1. 记录一个进程在内存中的分配情况；

2. 实现逻辑地址到物理地址的变换。

4. 地址结构及地址变换

（1）地址结构

在学习利用页表是怎样完成地址变换之前，必须了解地址结构。在页面管理中，分页系统自动把逻辑地址解释为两部分，高位部分为页号，低位部分为页内地址。因此，逻辑地址用一个数对（p, d）来表示，其中p表示所在的页号，d表示页内地址。

d、p所占的位数取决于页的大小。

为了简化地址变换过程和分页，通常页的大小为2的n次幂。

如：**1024(2^{10})=1k, 4096(2^{12}) =4k。**

若页的大小为**2^n**，则

d=n

p=有效地址长度-n

例：假设计算机CPU有效地址为16位，且页大小为1K。

分页系统的地变换机构自动把这个地址解释为页号（6位）和页内相对地址d（10位）。

地址构图

例：十进制地址：2500　　p=2　　d=452

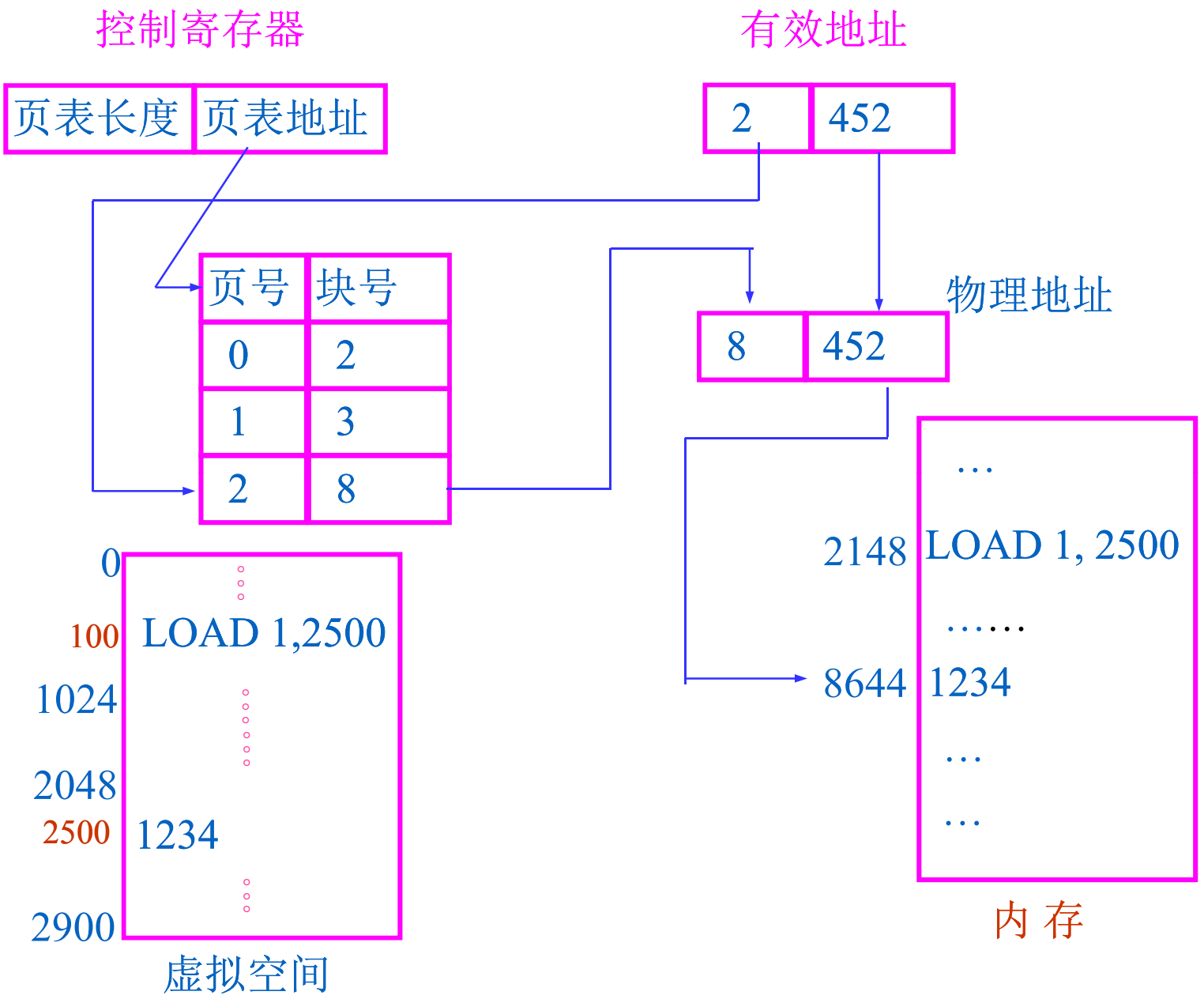
二进制地址：

计算方法：若给出一个逻辑地址为A，页面大小为L（字节），则：p＝int(A/L)　d=(A)mod(L)。

（2）地址变换

首先，当进程被调度运行，操作系统自动将该进程的页表起址和页表长度（该地址通常在PCB内），装入页表地址寄存器中，当CPU访问某个虚拟地址时，分页系统硬件机构自动完成逻辑地址到物理地址的转换。

例：设页的大小为1K，某进程有一条指令LOAD 1，2500，当CPU访问这条指令时，要进行地址变换。


控制寄存器
有效地址
页表长度
页表地址
2
452
页号
块号
0
2
1
3
2
8
物理地址
8
452
0
100
LOAD 1,2500
1024
2048
2500
1234
2900
虚拟空间
2148
LOAD 1, 2500
8644
1234
内 存

5.3.2 动态页式管理

1. 基本思想：

只需将当前要运行的那部分程序代码和数据所对应的页装入内存，便可启动运行。以后在进程运行的过程中，当需要访问某些页时，由系统自动地将需要的页从外存调入内存。如果内存没有足够的空闲页面，将暂不运行的进程页调出内存，以便装入新的进程页。内存中的页称为“实页”，把在外存中页称为“虚页”。

(1) 当进程要访问的某个虚页不在内存中时，怎样发现这种缺页情况，发现后又怎么办？

(2) 当需要把某一虚页调入内存时，若此时内存中没有空闲页面，应该怎么处理？

2．页表

页号	页面号	驻留位	访问位	修改位	外存地址

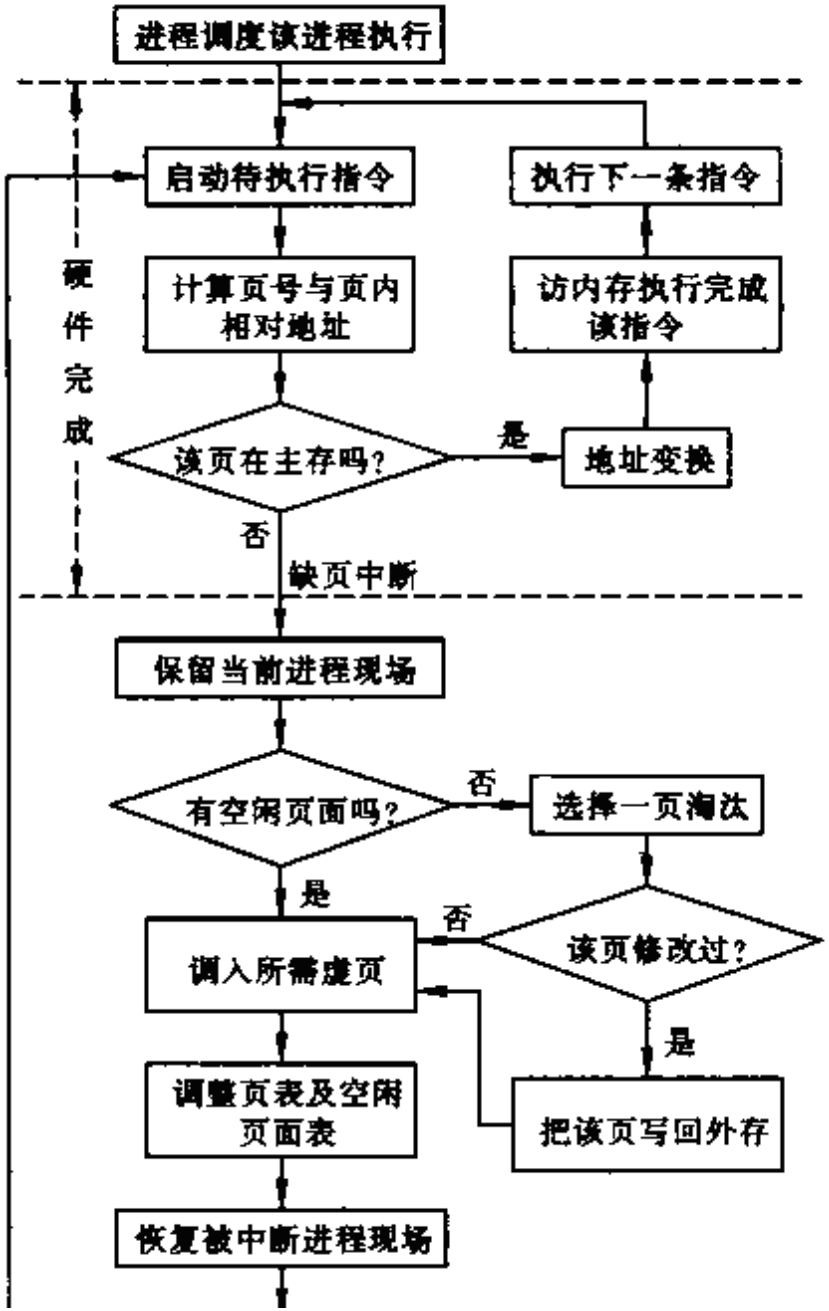
(1) 驻留位P：用于指示该页是否已调入内存。

(2) 访问位A：用于记录本页在一段时间内被访问的次数，或记录本页最近已有多长时间未被访问，供选择换出页面时参考。

(3) 修改位M：表示该页在调入内存后是否被修改过。若未被修改，在置换该页时就不需再将该页写回到外存上，若已被修改，则必须将该页重写到外存上，以保证外存中所保留的始终是最新副本。

(4) 外存地址：用于指出该页在外存上的地址，通常是物理块号，供调入该页时参考。

3．缺页中断机构

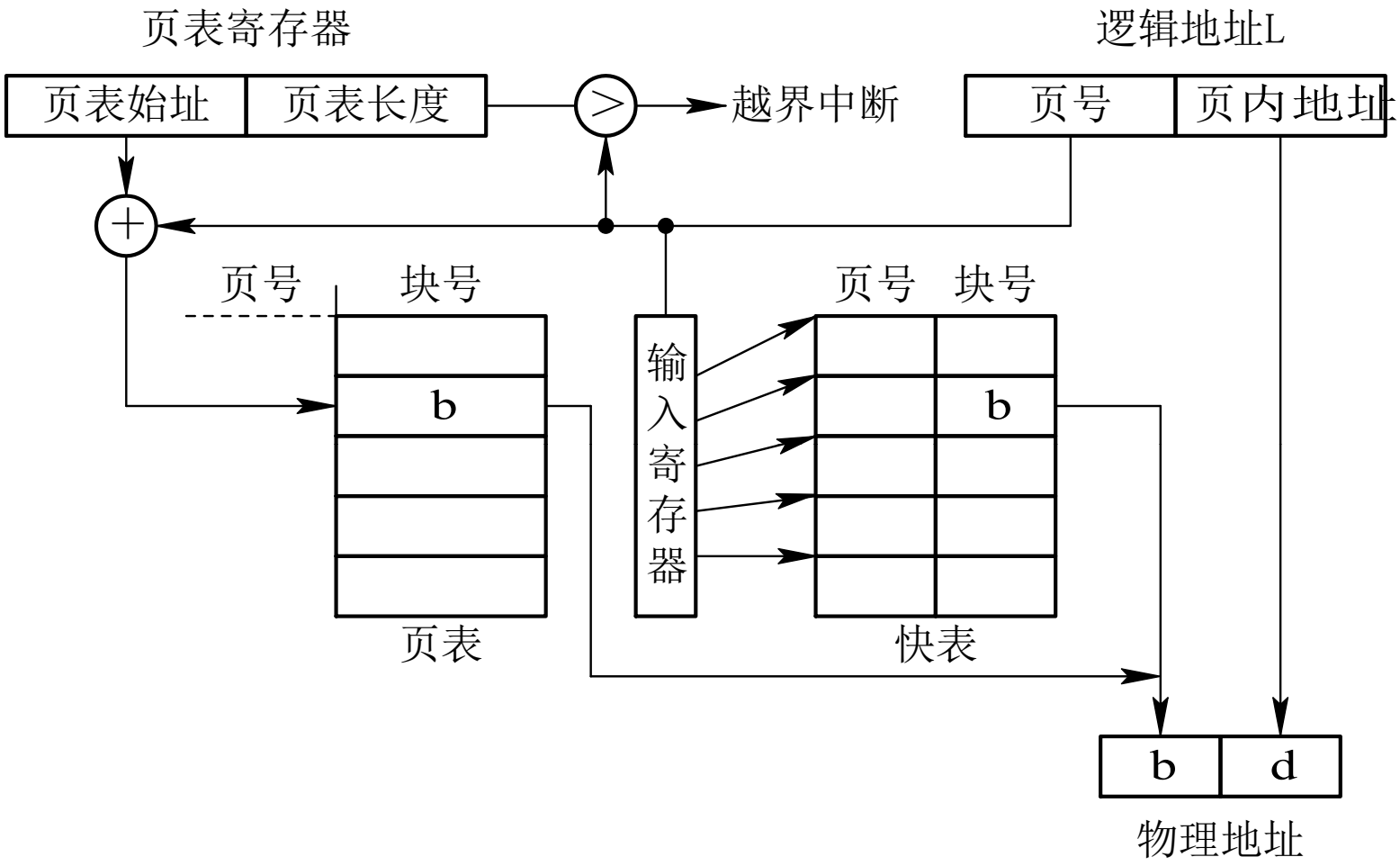

进程调度该进程执行
启动待执行指令
执行下一条指令
计算页号与页内相对地址
访内存执行完成该指令
硬件完成
该页在主存吗?
是
地址变换
否
缺页中断
保留当前进程现场
有空闲页面吗?
否
选择一页淘汰
是
否
调入所需虚页
该页修改过?
是
调整页表及空闲页面表
把该页写回外存
恢复被中断进程现场

5.3.3 指令存取速度与页面大小的问题

1. 快表

页式管理系统中，存取一个数据或访问一条指令要访问两次主存：页表、真正的内存地址。显然，这种方法比通常执行指令的速度慢了一倍。

为了提高查表速度，可在地址变换机构中增设一个具有并行查找能力的高速寄存器来存储当前常用的页号和对应的页面号。这个高速寄存器又称联想寄存器或快表。



具有快表的地址变换机构

2. 页面大小的选择

如果页面太大，页式管理就退化为分区管理，同时导致页内“碎片”过大。而页面太小，页表占用内存空间太多。一个系统的页表占用内存的空间大小与主存大小和页大小有关。

如：内存大小为16M，页大小为2K，则页面数为

16*1024/2=8192（个）

若一个页表的表目占2个字节，那么页表占用内存的空间大小

8192 * 2B=16KB

大部分计算机使用的页面大小为512B~64KB

3. 两级和多级页表

现代的大多数计算机系统，都支持非常大的逻辑地址空间(2^{32}～2^{64})。在这样的环境下，页表就变得非常大，要占用相当大的内存空间。

例如，对于一个具有32位逻辑地址空间的分页系统，规定页面大小为4 KB即2^{12} B，则在每个进程页表中的页表项可达1MB个之多。又因为每个页表项占用一个字节，故每个进程仅仅其页表就要占用1 MB的内存空间，而且还要求是连续的。显然这是不现实的，我们可以采用下述两个方法来解决这一问题：

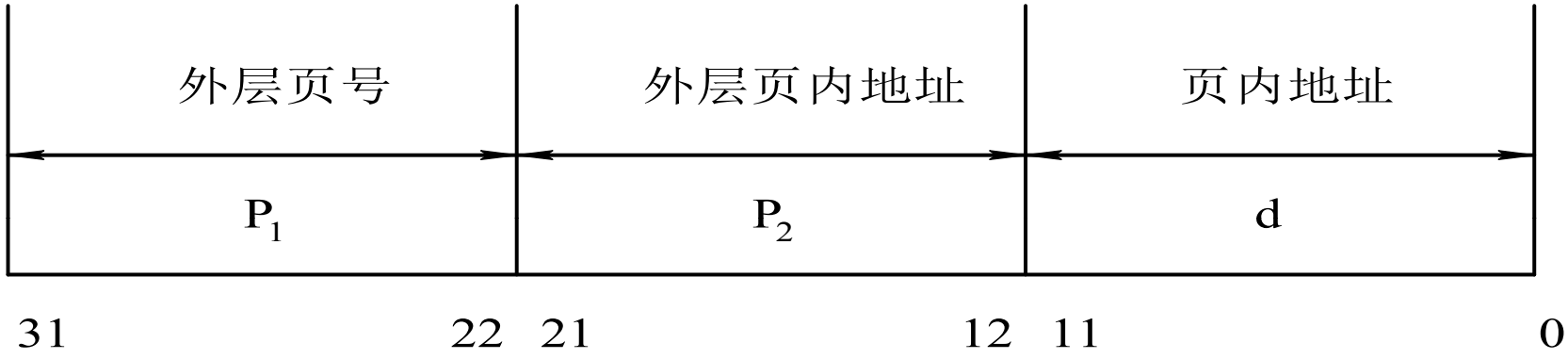
(1) 采用离散分配方式来解决难以找到一块连续的大内存空间的问题；

(2) 只将当前需要的部分页表项调入内存，其余的页表项仍驻留在磁盘上，需要时再调入。

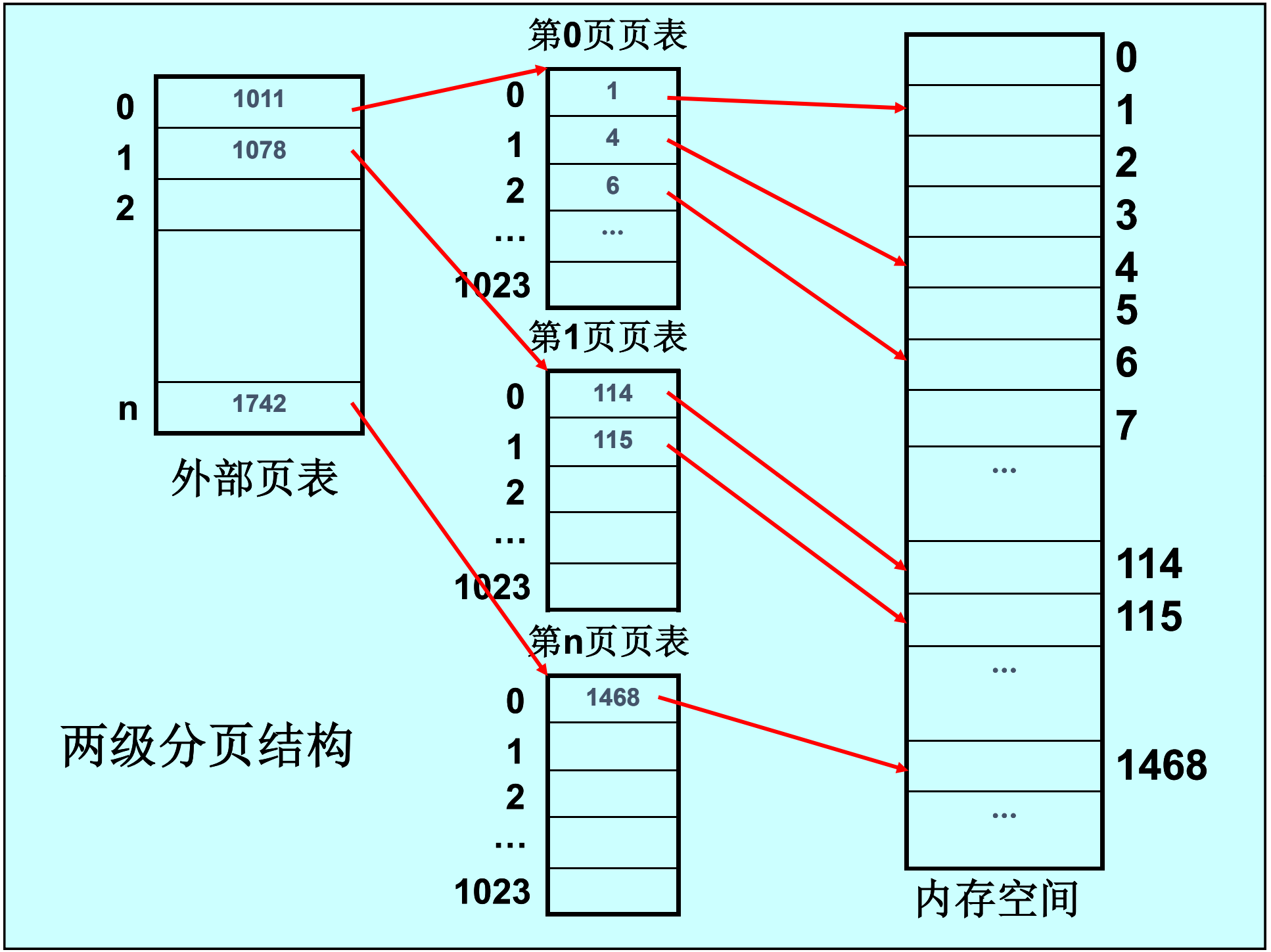
(1)两级页表(Two-Level Page Table)

对于要求连续的内存空间来存放页表的问题，可利用将页表进行分页，并离散地将各个页面分别存放在不同的物理块中的办法来加以解决，同样也要为离散分配的页表再建立一张页表，称为外层页表(Outer Page Table)，在每个页表项中记录了页表页面的物理块号。

当页面大小为 4 KB时(12位)，若采用一级页表结构，应具有20位的页号，即页表项应有1兆个；在采用两级页表结构时，再对页表进行分页，使每页中包含2^{10} (即1024)个页表项，最多允许有2^{10}个页表分页；或者说，外层页表中的外层页内地址P2为10位，外层页号P1也为10位。此时的逻辑地址结构可描述如下：


外层页号
外层页内地址
页内地址
P1
P2
d
31
22
21
12
11
0

由图可以看出，在页表的每个表项中存放的是进程的某页在内存中的物理块号，如第0#页存放在1#物理块中；1#页存放在4#物理块中。而在外层页表的每个页表项中，所存放的是某页表分页的首址，如第0#页表是存放在第1011#物理块中。我们可以利用外层页表和页表这两级页表，来实现从进程的逻辑地址到内存中物理地址间的变换。


第0页页表
第1页页表
第n页页表
外部页表
内存空间
两级分页结构
0
1
2
n
1011
1078
1742
0
1
2
…
1023
1
4
6
…
0
1
2
…
1023
114
115
0
1
2
…
1023
1468
0
1
2
3
4
5
6
7
…
114
115
…
1468
…

(2) 多级页表

对于32位的机器，采用两级页表结构是合适的；但对于64位的机器，如果页面大小仍采用4 KB即2^{12} B，那么还剩下52位，假定仍按物理块的大小(2^{12}位)来划分页表，则将余下的42位用于外层页号。此时在外层页表中可能有4096 G个页表项，要占用16 384 GB的连续内存空间。因此必须采用多级页表，将外层页表再进行分页，也就是将各分页离散地装入到不相邻接的物理块中，再利用第2级的外层页表来映射它们之间的关系。

对于64位的计算机，如果要求它能支持2^{64} B(= 1 844 744 TB)规模的物理存储空间，则即使是采用三级页表结构也是难以办到的；

在近两年推出的64位OS中，把可直接寻址的存储器空间减少为45位长度(即2^{45})左右，这样便可利用三级页表结构来实现分页存储管理。

5.3.4　存储保护

界限寄存器或保护键法

5.3.4　页式管理的优缺点

1. 优点

（1）有效地解决了内存的碎片问题，因此，可使内存得到有效的利用，有可能使更多的进程同时投入运行，进一步可提高处理机的利用率。

（2）动态页式存储管理只要求每个进程部分装入便可运行，实现了内存的扩充技术。可为用户提供比实际内存更大的虚拟存储空间，使用户可利用的存储空间大大增加，有利于多道程序的组织，提高内存的利用率。

2．缺点

（1）要求有相应的硬件支持。例如，地址变换机构，缺页中断机构和页面的淘汰等。这些增加了计算机的成本；

（2）增加了系统开销。如页面中断处理，表格的建立和管理这些都需花费处理机时间，且表格还要占用一定的存储空间；

（3）淘汰算法选择不当有可能会严重影响系统的使用效率。

（4）虽然消除了碎片，但同时还存在页内碎片问题。

5.4 淘汰算法与抖动现象

5.4.1 淘汰算法

通常把选择要换出页的算法称为淘汰算法。而一个好的淘汰算法，应使缺页率尽可能小。

1. 最佳（Optimal）淘汰算法

访页顺序	0	2	5	3	2	4	2	0	3	2	1	3	2	3	4	3
内存实页	0	0	0	0	0	0	0	0	3	3	3	3	3	3	3	3
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
			5	3	3	4	4	4	4	4	1	1	1	1	4	4
缺页中断	√	√	√	√		√			√		√				√	

8次缺页，5 次置换

缺页率：8/16，它可作为衡量其它算法优劣的一个标准。

2. 先进先出淘汰算法

访页顺序	0	2	5	3	2	4	2	0	3	2	1	3	2	3	4	3
内存实页	0	0	0	3	3	3	3	0	0	0	0	0	2	2	2	2
		2	2	2	2	4	4	4	3	3	3	3	3	3	4	4
			5	5	5	5	2	2	2	2	1	1	1	1	1	3
缺页中断	√	√	√	√		√	√	√	√		√		√		√	√

FIFO算法容易实现，但是它所依据的理由与普遍的进程运行规律不符。它只适用于CPU按线性顺序访问地址空间的进程。

3. 最近最少使用（LRU）淘汰算法

访页顺序	0	2	5	3	2	4	2	0	3	2	1	3	2	3	4	3
内存实页	0	0	0	3	3	3	3	0	0	0	1	1	1	1	4	4
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
			5	5	5	4	4	4	3	3	3	3	3	3	3	3
缺页中断	√	√	√	√		√		√	√		√				√	

最近最久没有被使用的页

LRU置换算法虽然是一种比较好的算法，但要求系统有较多的支持硬件。为了了解一个进程在内存中的各个页面各有多少时间未被进程访问，以及如何快速地知道哪一页是最近最久未使用的页面，须有一些硬件支持。

1）寄存器

为了记录某进程在内存中各页的使用情况，须为每个在内存中的页面配置一个移位寄存器，可表示为

$$R = R_{n-1}R_{n-2}R_{n-3}\cdots R_2R_1R_0$$

当进程访问某物理块时，要将相应寄存器的Rn－1位置成1。此时，定时信号将每隔一定时间(例如100 ms)将寄存器右移一位。如果我们把n位寄存器的数看做是一个整数，那么，具有最小数值的寄存器所对应的页面，就是最近最久未使用的页面。下图示出了某进程在内存中具有8个页面，为每个内存页面配置一个8位寄存器时的LRU访问情况。这里，把8个内存页面的序号分别定为1～8。由图可以看出，第3个内存页面的R值最小，当发生缺页时，首先将它置换出去。

实页 \ R	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0
1	0	1	0	1	0	0	1	0
2	1	0	1	0	1	1	0	0
3	0	0	0	0	0	1	0	0
4	0	1	1	0	1	0	1	1
5	1	1	0	1	0	1	1	0
6	0	0	1	0	1	0	1	1
7	0	0	0	0	0	1	1	1
8	0	1	1	0	1	1	0	1

LRU算法的寄存器实现

2）栈

可利用一个特殊的栈来保存当前使用的各个页面的页面号。每当进程访问某页面时，便将该页面的页面号从栈中移出，将它压入栈顶。因此，栈顶始终是最新被访问页面的编号，而栈底则是最近最久未使用页面的页面号。

访页顺序	0	2	5	3	2	4	2	0	3	2	1	3	2	3	4	3
栈顶	0	2	5	3	2	4	2	0	3	2	1	3	2	3	4	3
		0	2	5	3	2	4	2	0	3	2	1	3	2	3	4
栈底			0	2	5	3	3	4	2	0	3	2	1	1	2	2
置换				√		√		√	√		√				√	

3）计数器法

要求系统中有一个64位的硬件计数器C，它在每次执行完指令后自动加1，而进程的每个页表项必须有一个足以容纳这个计数器的值的域。在每次访问内存后，当前的C值存放到被访问的页的页表项中。当发生缺页时，操作系统检查页表中所有计数器的值，其中C值最小的对应页就是最近最少使用的页。

缺点是浪费处理机时间和内存空间。

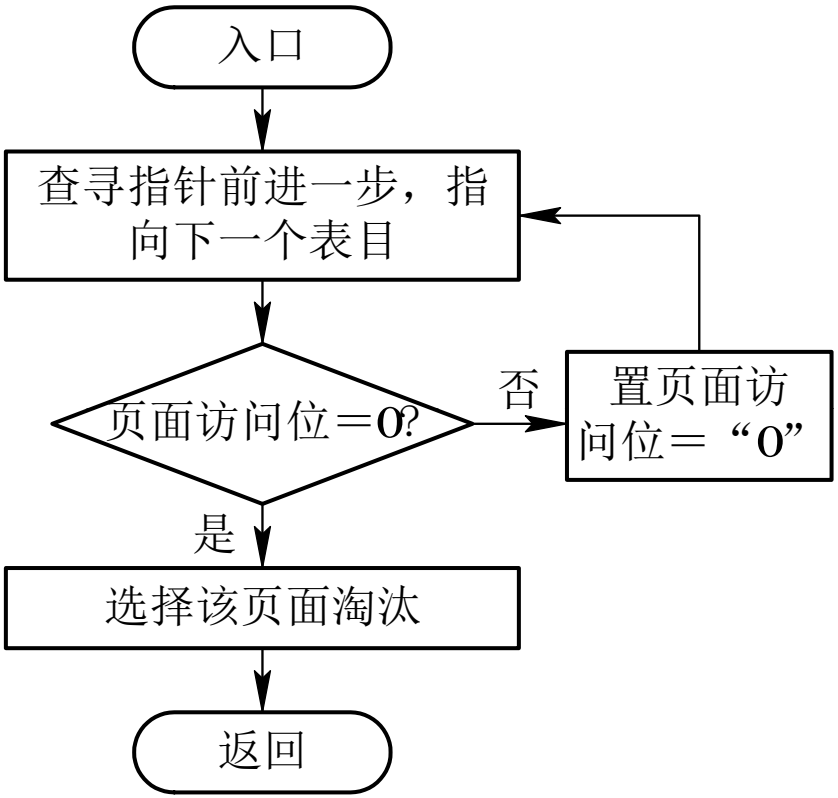
4. LRU 的近似算法

（1）最不经常使用(NotFrequentlyUsed,NFU)算法。该算法选择到当前时间为止被访问次数最少的那一页淘汰。这只需在页表中给每一页增设一个访问计数器即可实现。每当该页被访问时,访问计数器加1。而发生缺页时,则淘汰计数器值最小的那一页,并将所有的计数器清零。

（2） 最近未使用(NRU)算法（Clock算法）

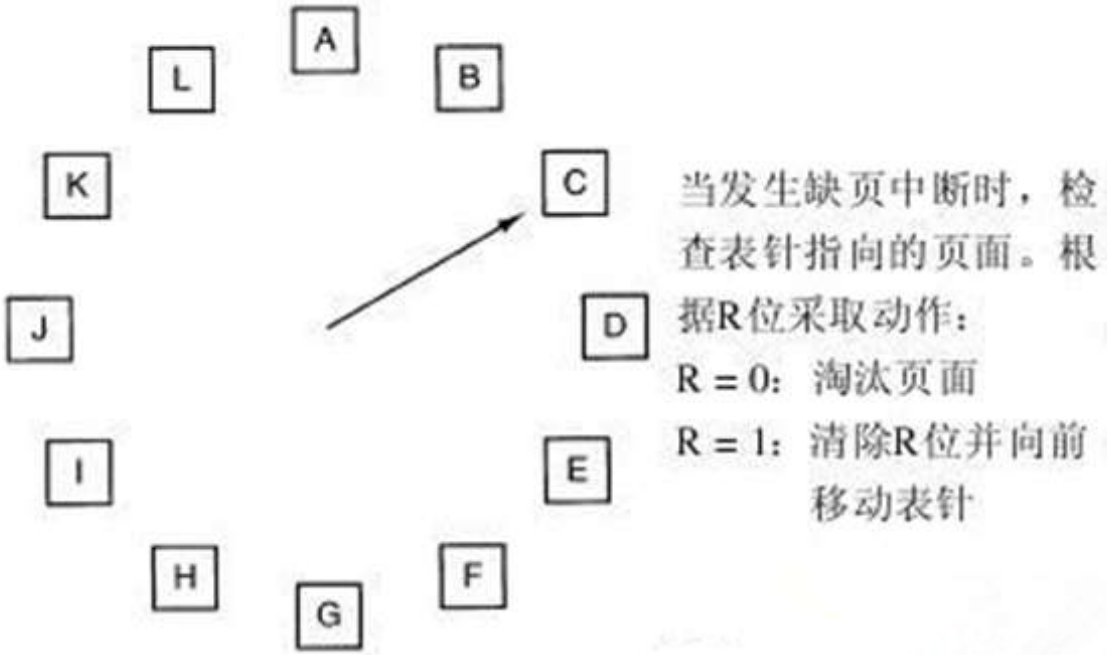
最近一段时间内没有被访问的页。

第二次机会算法：为每页设置一访问位R，表示该页是否已经被使用过，将内存中的所有页面都链接成一个循环队列。某页被访问时访问位置R=1。淘汰时，如果是R= 0 ，换出；如果为R= 1 ，置R= 0 ，不换出(第二次驻留内存的机会），按FIFO检查下一个页面。当队尾时，最后一个页面为1，则返回队首去检查第一个页面，必然为0 。



块号	页号	访问位	指针
0			
1			
2	4	0	
3			
4	2	1	
5			
6	5	0	
7	1	1	

替换指针



时钟算法：尽管第二次机会算法是一个比较合理的算法，但它经常要在链表中移动页面，既降低了效率又不是很有必要。一个更好的办法是把所有的页面都保存在一个类似钟面的环形链表中，一个表针指向最老的页面。当发生缺页中断时，算法首先检查表针指向的页面，如果它的R位是0就淘汰该页面，并把新的页面插入这个位置，然后把表针前移一个位置；如果R位是1就清除R位并把表针前移一个位置，重复这个过程直到找到了一个R位为0的页面为止。

（2）改进的NRU算法。

考虑到如果某一调入内存的页没有被修改过，则不必将它拷回到磁盘。

第1类：A＝0，M＝0，没有被访问过，没有被修改过，是最佳的淘汰页；

第2类：A＝0，M＝1，没有被访问过，被修改过；

第3类：A＝1，M＝0，被访问过，没有被修改过；

第4类：A＝1，M＝1，被访问过，被修改过；

其执行过程分一下三步：

第一步：从开始位置循环扫描队列，寻找=0、M=0的第一类面，找到立即置换。另外，第一次扫描期间不改变访问位A。

第二步：如果第一步失败，则开始第二轮扫描，寻找A=0且M=1的第二类页面，找到后立即置换，并将所有扫描过的A都置0。

第三步：如果第二步也失败，则返回指针开始位置，然后重复第一步，必要时再重复第二步，此时必能找到淘汰页。

5. 页面缓冲算法 BPA

（1）空闲页面链表，其中的每个物理块都是空闲的当需要读入一个页面时，便可利用空闲物理块链表中的第一个物理块来装入该页。当有一个未被修改的页要换出时，实际上并不将它换出内存，而是把该未被修改的页所在的物理块挂在自由页链表的末尾。

（2）修改页面链表。在置换一个已修改的页面时，将其所在的物理块挂在修改页面链表的末尾。利用这种方式可使已被修改的页面和未被修改的页面都仍然保留在内存中。当该进程以后再次访问这些页面时，只需花费较小的开销。当被修改的页面数目达到一定值时，例如64个页面，再将它们一起写回到磁盘上，从而显著地减少了磁盘I/O的操作次数。

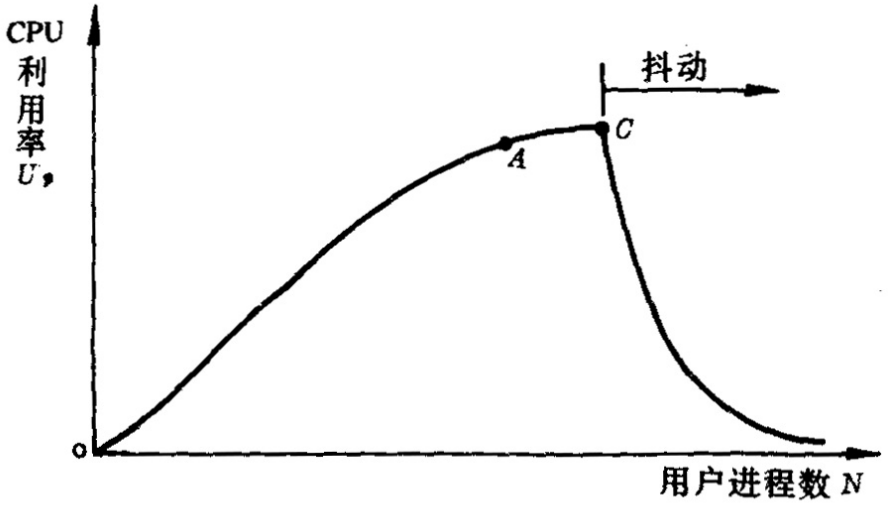
VAX/VMS操作系统（卫星控制类操作系统）便是使用页面缓冲算法。它的置换算法采用的是FIFO。该算法规定将一个被淘汰的页放入两个链表中的一个，即如果页面未被修改，就将它直接放入空闲链表中；否则，便放入已修改页面的链表中。须注意的是，这时页面在内存中并不做物理上的移动，而只是将页表中的表项移到上述两个链表之一中。

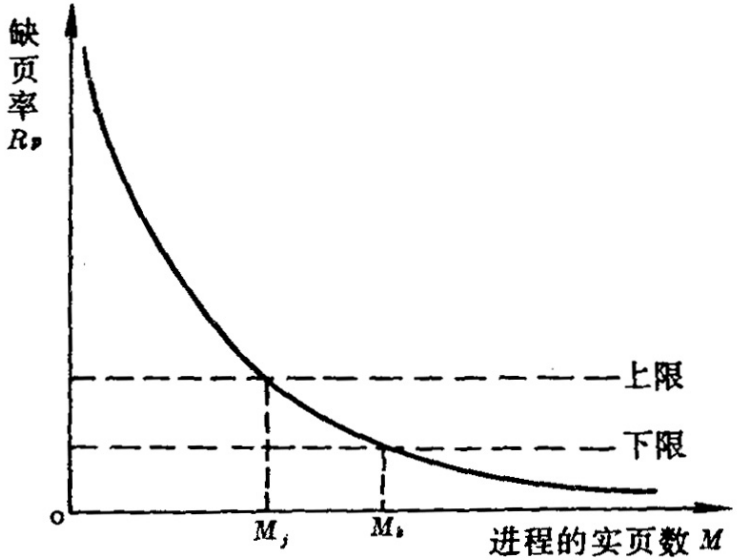
页面缓冲算法(PBA)的主要特点是：①显著地降低了页面换进、换出的频率，使磁盘I/O的操作次数大大减少，因而减少了页面换进、换出的开销；②由于换入换出的开销大大减少，又可采用一种较简单的置换策略如FIFO 算法，不需要硬件支持，实现起来非常简单。

5.4.2　抖动现象与工作集

不适当的算法可能会导致进程发生抖动（Thrashing）。即刚被换出的页很快又要被访问，需要将它重新调入，此时又需要再选一页调出；而此刚被调出的页很快又被访问，又需将它调入，如此频繁地更换页面，以致一个进程在运行中把大部分时间都花费在页面置换工作上，我们称该进程发生了抖动。

防止抖动现象方法：一种是选择好的淘汰算法，以减少缺页次数。一种是扩大工作集:是指进程在某个时间段里要访问的页的集合。如果能够预知进程在某段时间的工作集，并在此之前把该集合调入内存，至该段时间终了时，再将其在下一时间段时不需要访问的哪些页换出内存，这样就会可以减少页的交换。





5.5 段式存储管理

存储管理从单一连续分配发展到分页存储管理方式，其主要动力都是直接或间接提高内存利用率，引入分段存储管理方式的目的，则主要是为了满足用户（程序员）在编程和使用上多方面的要求，其中有些要求是其它集中存储管理方式难以满足的。因此，这种存储管理方式已成为当今所有存储管理方式的基础，许多高级语言和C语言的编译程序也都支持分段存储管理方式。

分段存储管理方式的引入

1）方便编程

通常，用户把自己的作业按照逻辑关系划分为若干个段，每个段都是从0开始编址，并有自己的名字和长度。因此，希望要访问的逻辑地址是由段名(段号)和段内偏移量(段内地址)决定的。例如，下述的两条指令便是使用段名和段内地址：

LOAD 1，[A]｜〈D〉；

STORE 1，[B]｜〈C〉；

其中，前一条指令的含义是将分段A中D单元内的值读入寄存器1；后一条指令的含义是将寄存器1的内容存入B分段的C单元中。

2）信息共享

在实现对程序和数据的共享时，是以信息的逻辑单位为基础的。比如，共享某个例程和函数。分页系统中的“页”只是存放信息的物理单位(块)，并无完整的意义，不便于实现共享；然而段却是信息的逻辑单位。由此可知，为了实现段的共享，希望存储管理能与用户程序分段的组织方式相适应。

3）信息保护

信息保护同样是对信息的逻辑单位进行保护，因此，分段管理方式能更有效和方便地实现信息保护功能。

4）动态增长

在实际应用中，往往有些段，特别是数据段，在使用过程中会不断地增长，而事先又无法确切地知道数据段会增长到多大。前述的其它几种存储管理方式，都难以应付这种动态增长的情况，而分段存储管理方式却能较好地解决这一问题。

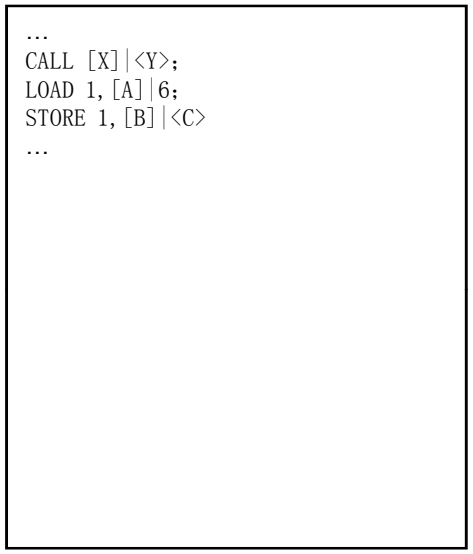
5）动态链接

动态链接是指在作业运行之前，并不把几个目标程序段链接起来。要运行时，先将主程序所对应的目标程序装入内存并启动运行，当运行过程中又需要调用某段时，才将该段(目标程序)调入内存并进行链接。可见，动态链接也要求以段作为管理的单位。

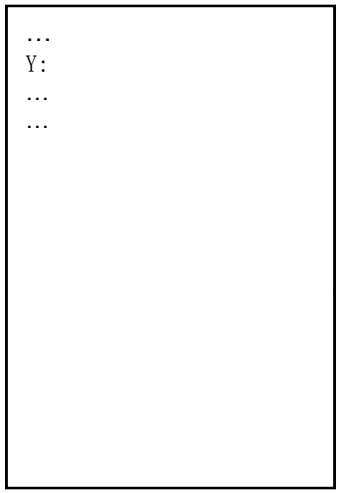
5.5.1 静态段式存储管理

1. 分段思想

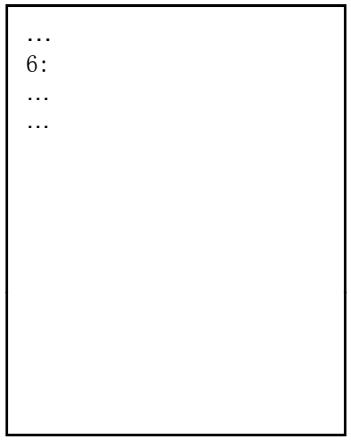
在分段存储管理方式中，作业的地址空间被划分为若干个段，每个段定义了一组逻辑信息。例如，有主程序段MAIN、子程序段X、数据段D及栈段S等。每个段都有自己的名字。为了实现简单起见，通常可用一个段号来代替段名，每个段都从0开始编址，并采用一段连续的地址空间。段的长度由相应的逻辑信息组的长度决定，因而各段长度不等。整个作业的地址空间由于是分成多个段，因而是二维的，亦即，其逻辑地址由段号(段名)和段内地址所组成。



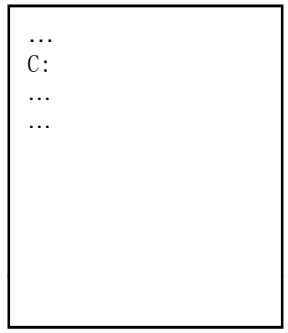
主程序段main



X段



D段



S段

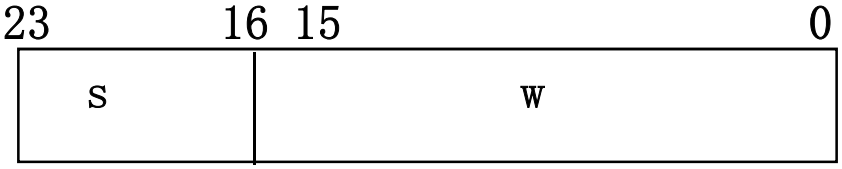
2．地址空间和地址结构

例：CALL [X]|<Y>；

LOAD 1,[A]|6；

STORE 1,[B]|<C>。

CALL 3,120



一个程序允许有256个段，一个段的最大长度为64KB。

3．数据结构

（1）段表

（2）内存空间表（或链）

（3）请求表 （记录每个进程的段表起始地址和长度）

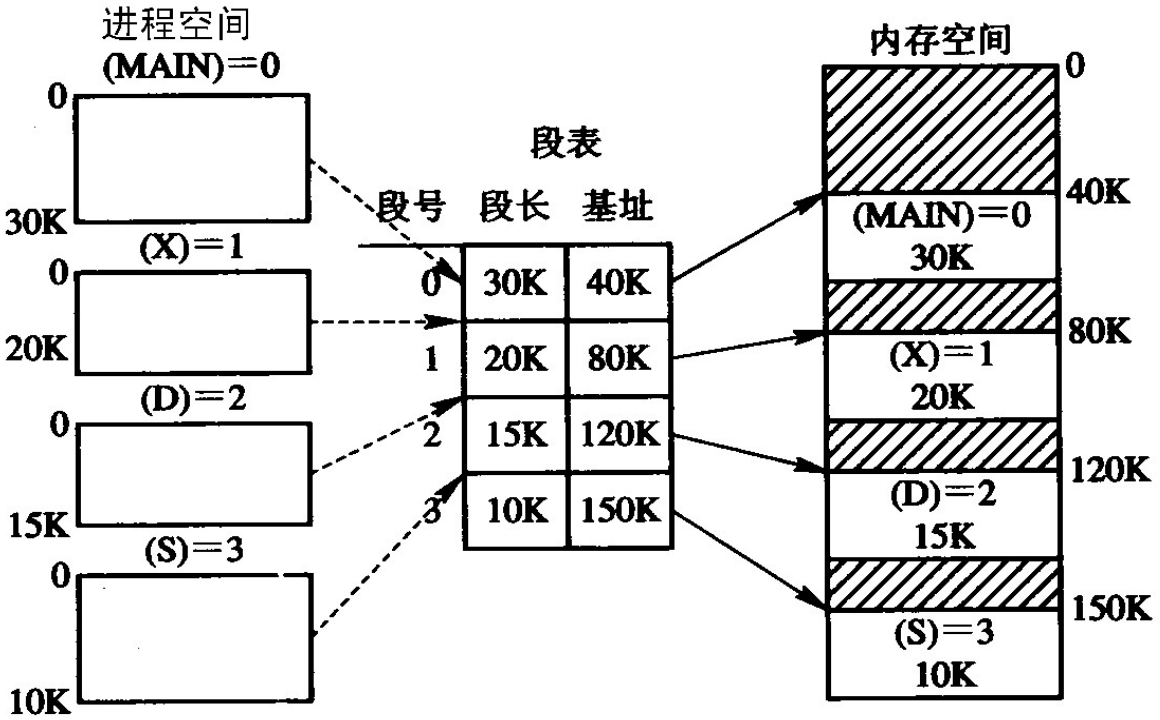


图5-31 段、段表及段在内存空间占用情况

4. 地址变换

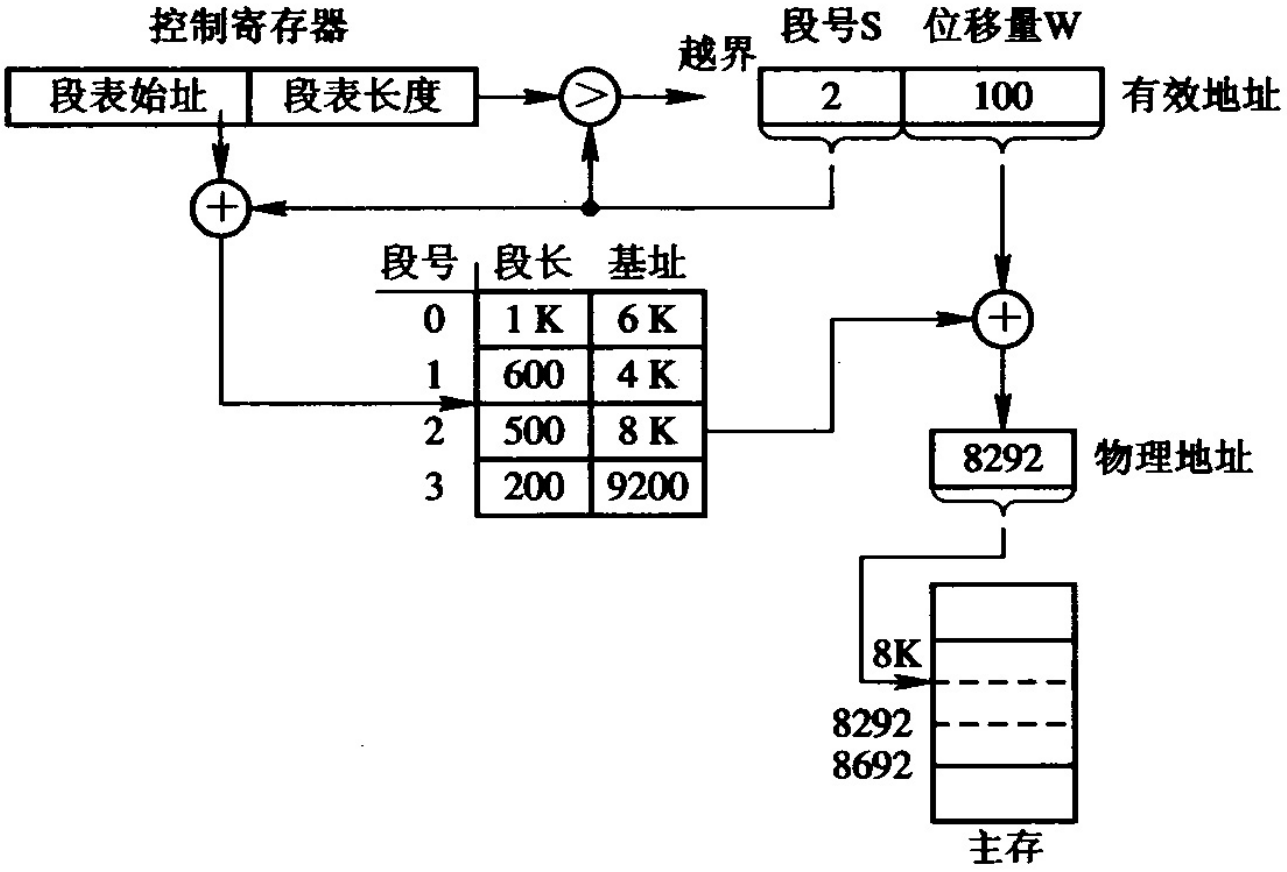


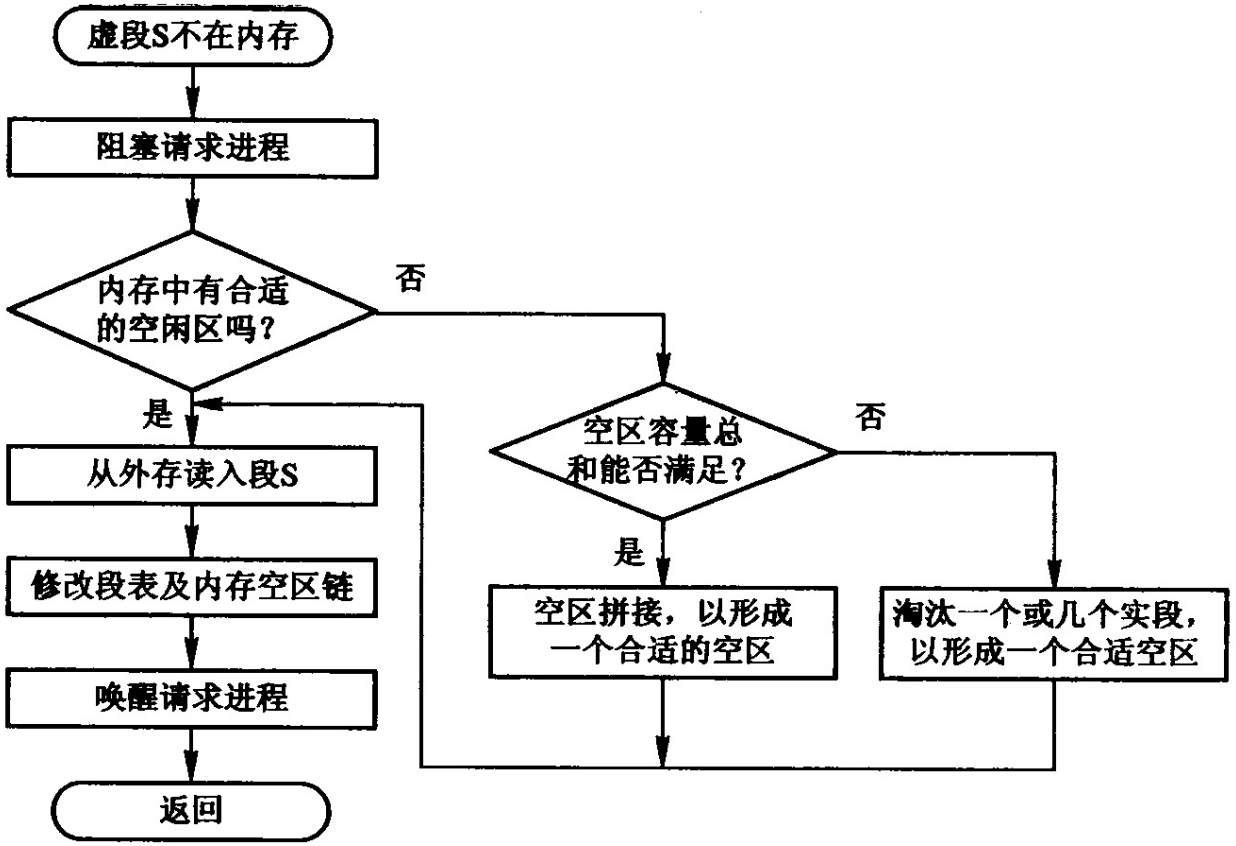
图5-32 段式存储管理地址变换过程

与可变分区的管理方法类似。

5.5.2　动态段式存储管理

段表

段号	内存始址	段长	驻留位	访问位	修改位	外存地址	存取方式	增补位



5-33　缺段中断处理

5.5.3 分段和分页的主要区别

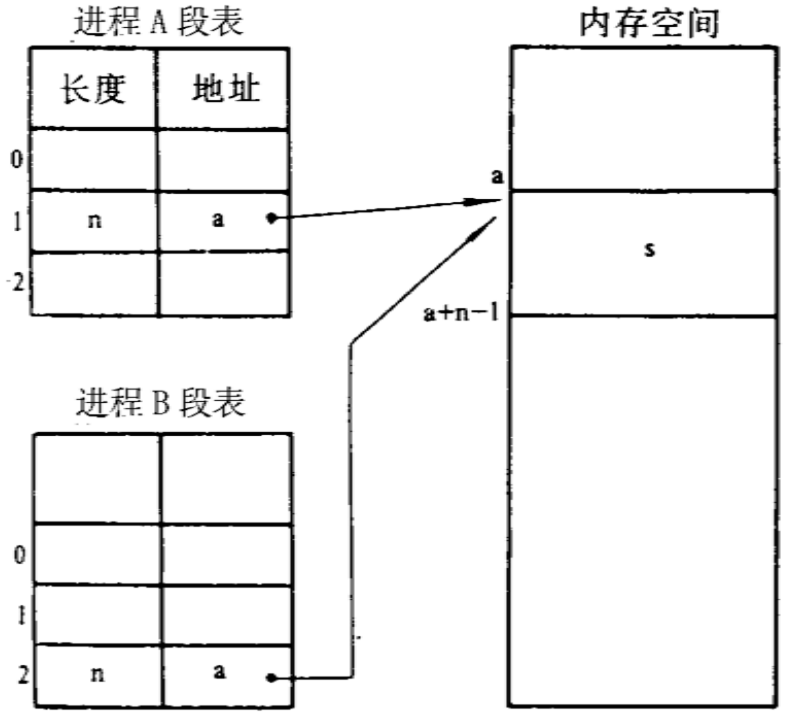
（1）段是面向用户的，页是面向系统的。

（2）页的大小是固定的，由系统决定；段的大小不固定，由用户决定。

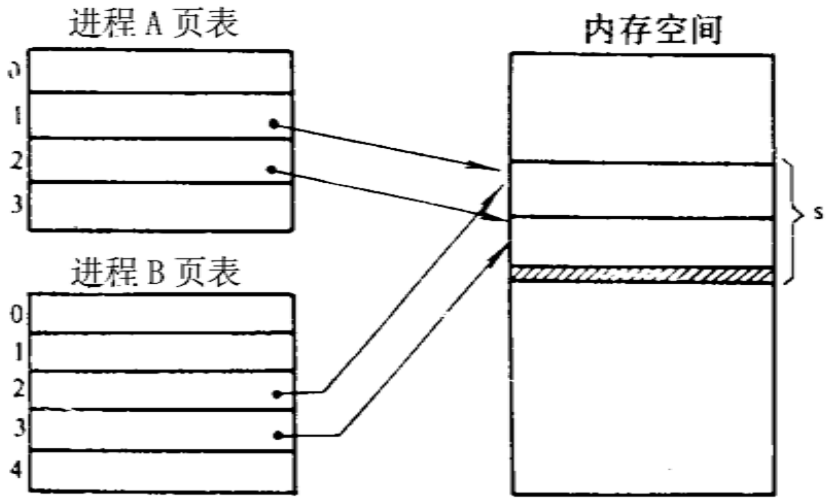
（3）从用户的角度看，分页系统的用户程序空间是一维连续的线性空间；段的地址空间是二维的，由段名和段内相对地址组成。

（4）从管理的角度看，分页系统的二维地址是在地址变换过程中由系统的硬件机构实现的，对用户是透明的。分段系统的在地址变换过程中的二维地址是由用户提供的。因而，页内没有地址越界问题，而段内的相对地址则存在地址越界问题。

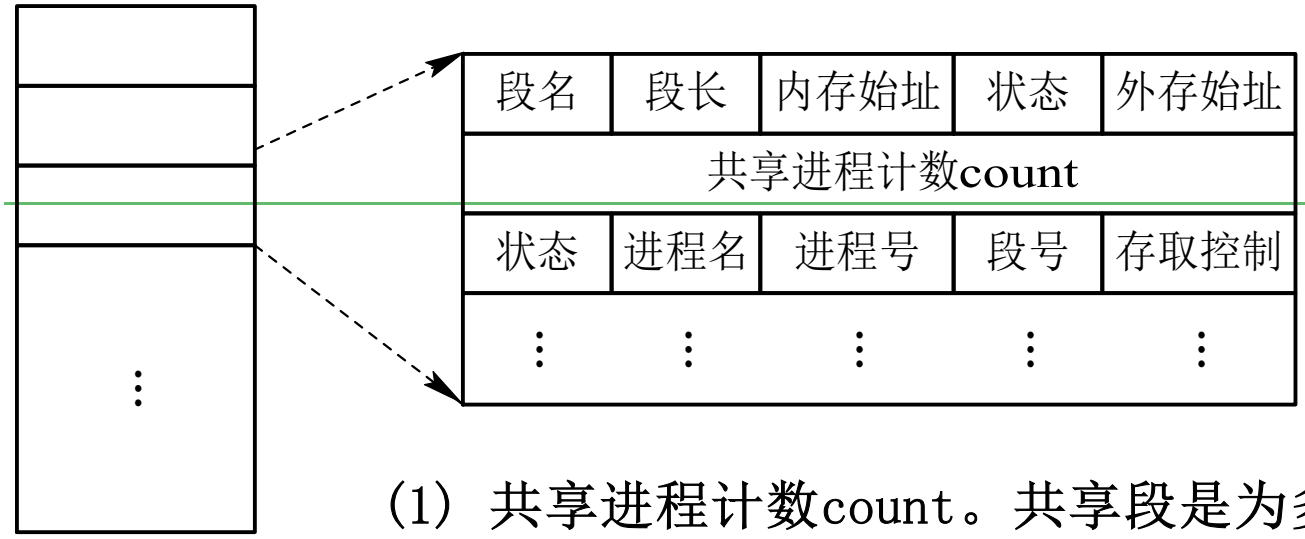
5.5.4 段的信息共享



（a）段的共享



（b）页的共享



（1）共享进程计数count。共享段是为多个进程所需要的，仅当所有共享该段的进程全都不再需要它时，才由系统回收该段所占内存区。为了记录有多少个进程需要共享该分段，特设置了一个整型变量count。

(2) 存取控制字段。对于一个共享段，应给不同的进程以不同的存取权限。例如，对于文件主，通常允许他读和写；而对其它进程，则可能只允许读，甚至只允许执行。

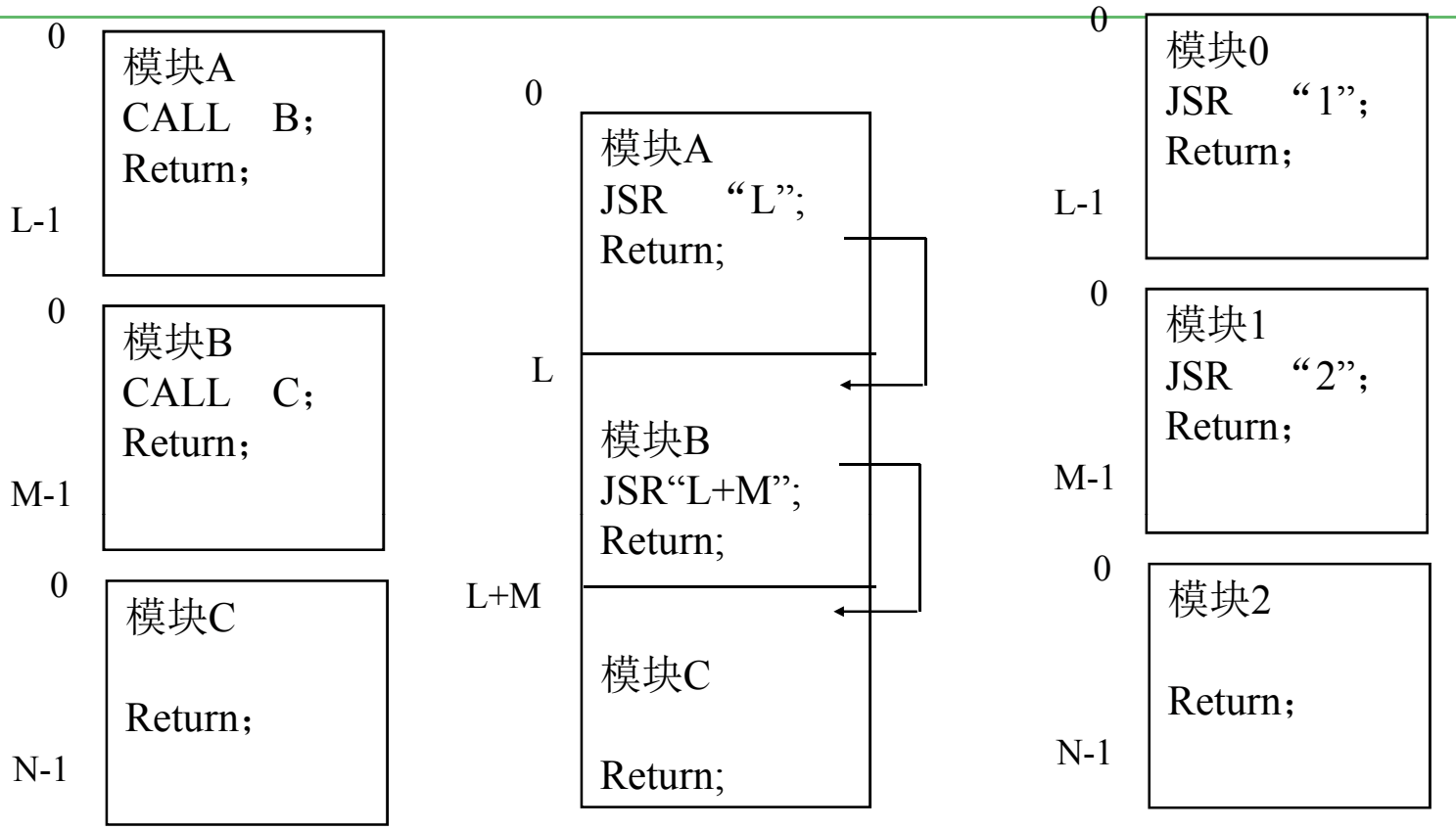
(3) 段号。对于一个共享段，不同的进程可以各用不同的段号去共享该段。

5.5.5 段的静态链接与动态链接

多个目标模块由链接程序链接形成装入模块，以便最后装入内存中执行。

1. 静态链接

指程序在执行前，由链接装配程序将该程序的所有目标模块进行链接和相对地址重定位，使之成为一个可运行的目标程序。



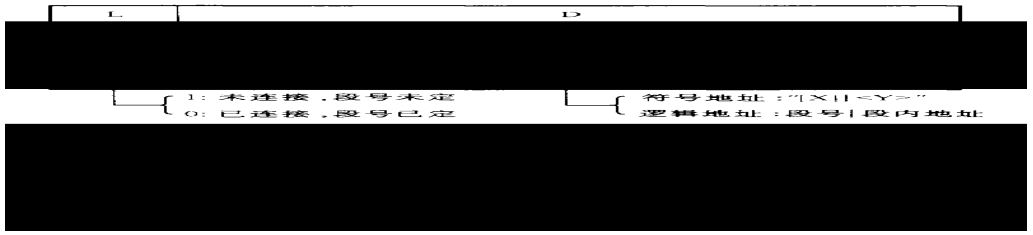
（a）目标模块　　（b）一维空间装入模块　　（c）二维空间装入模块

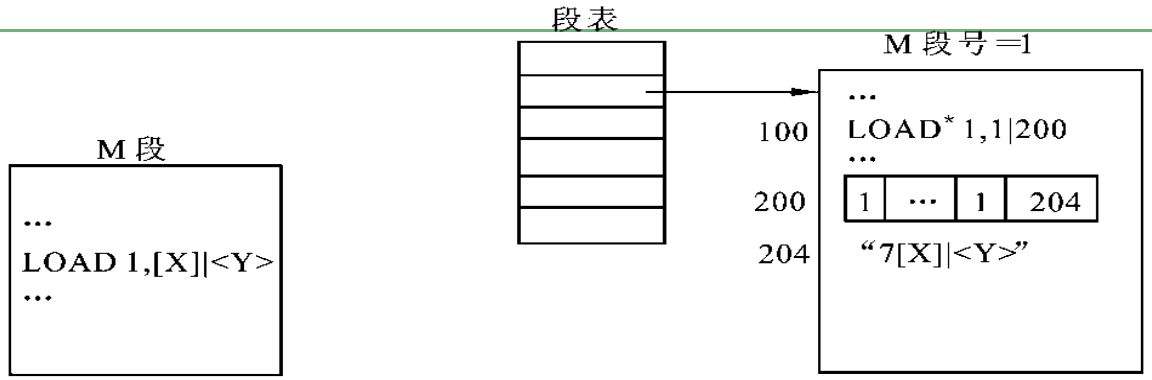
图　　静态链接示例

2. 动态链接

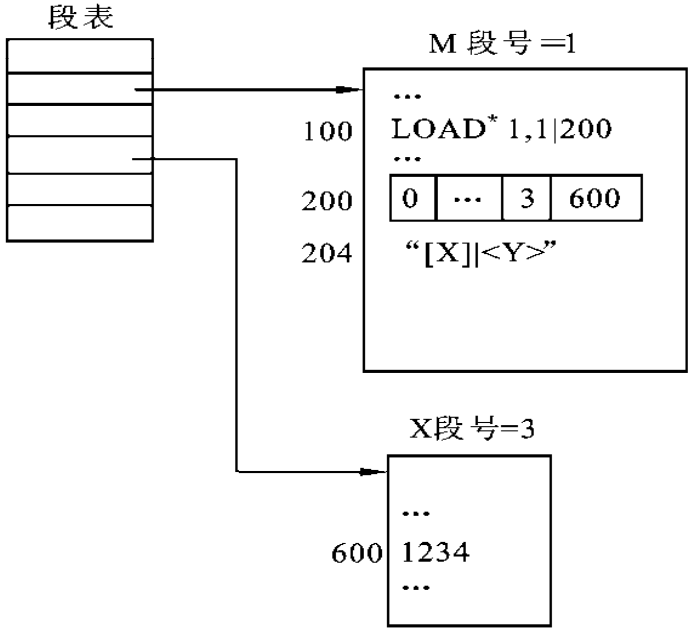
动态链接是指程序在执行过程中需要某一段时，再将该段从外存调入内存，把它与有关的段链接在一起。这样，凡是在程序执行过程中不会用到的段都不会调入内存，也不会链接到装入模块上，因而，不仅能加快程序的装入过程，也可节省大量的内存时间。

为了实现动态链接，在MULTICS系统中需要有进行链接中断的硬件机构和间接寻址功能。在程序汇编或编译时，当遇到访问外段的指令时，将其编译成一条间接寻址指令，即，将访问的地址指向一个间接地址，这个间接地址被称为间接字。





（a）编译前　　（b）编译后未链接



（c）链接后

5.5.6　段式存储管理的内存保护

地址越界保护法　　存取控制保护法

5.5.7　段式存储管理的优缺点

1．优点

（1）便于信息的共享和保护。

（2）实现了内存的扩充。

（3）便于信息的变化处理。

（4）便于实现动态链接。

2．缺点

（1）增加了计算机成本。

（2）存在碎片问题。

（3）段的长度受内存可用空间大小的限制。

（4）与页式存储管理类似，淘汰算法选择不当，可能产生抖动现象。

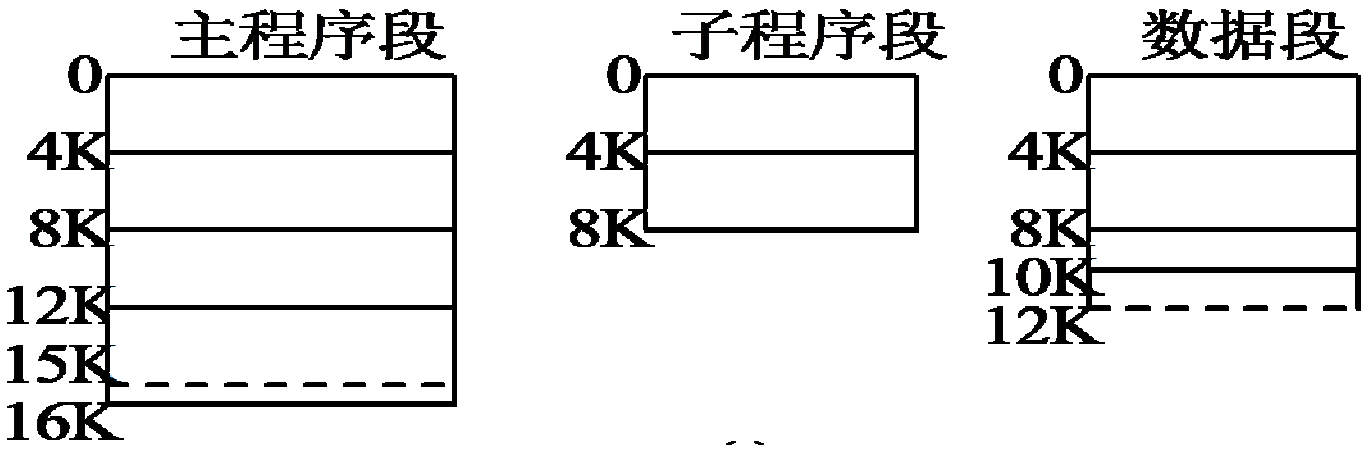
5.5 段页式存储管理

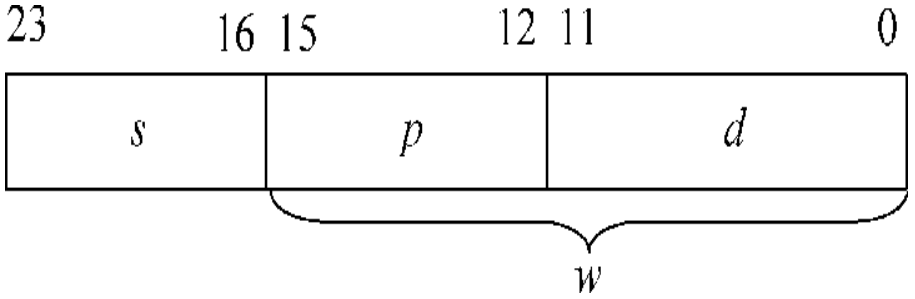
段式存储管理大大方便了用户，便于信息共享和信息的动态增加，但存在内存的碎片问题。页式存储管理则大大提高了主存的利用率，但不便于信息的共享。因此，结合二者的优点的一种新的存储管理方案被提了出来，这就是段页式存储管理。

1．基本原理

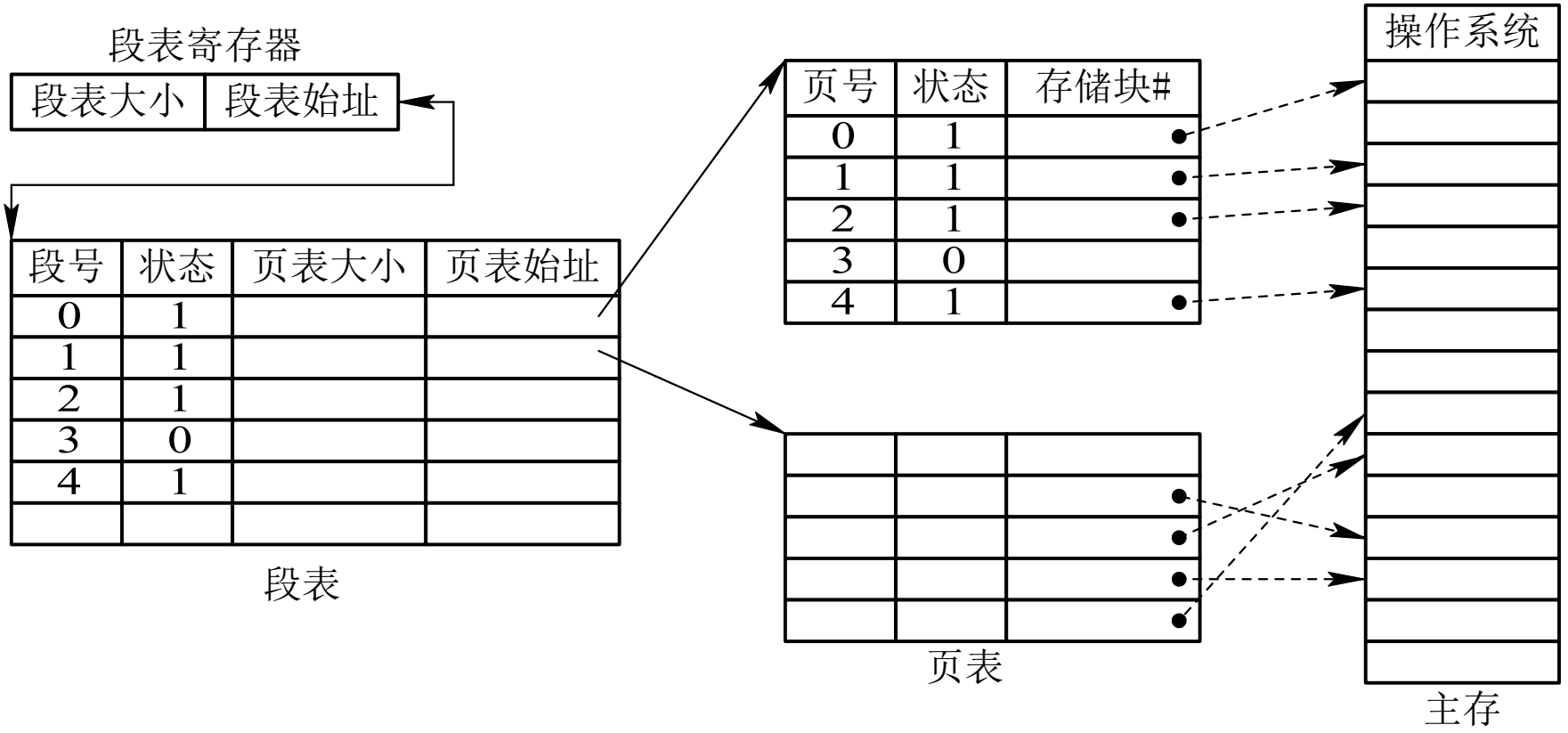
段页式系统的基本原理，是分段和分页原理的结合，即先将用户程序分成若干个段，再把每个段分成若干个页，并为每一个段赋予一个段名。

图4-21示出了一个作业地址空间的结构。该作业有三个段，页面大小为4 KB。在段页式系统中，其地址结构由段号、段内页号及页内地址三部分所组成，如下图。





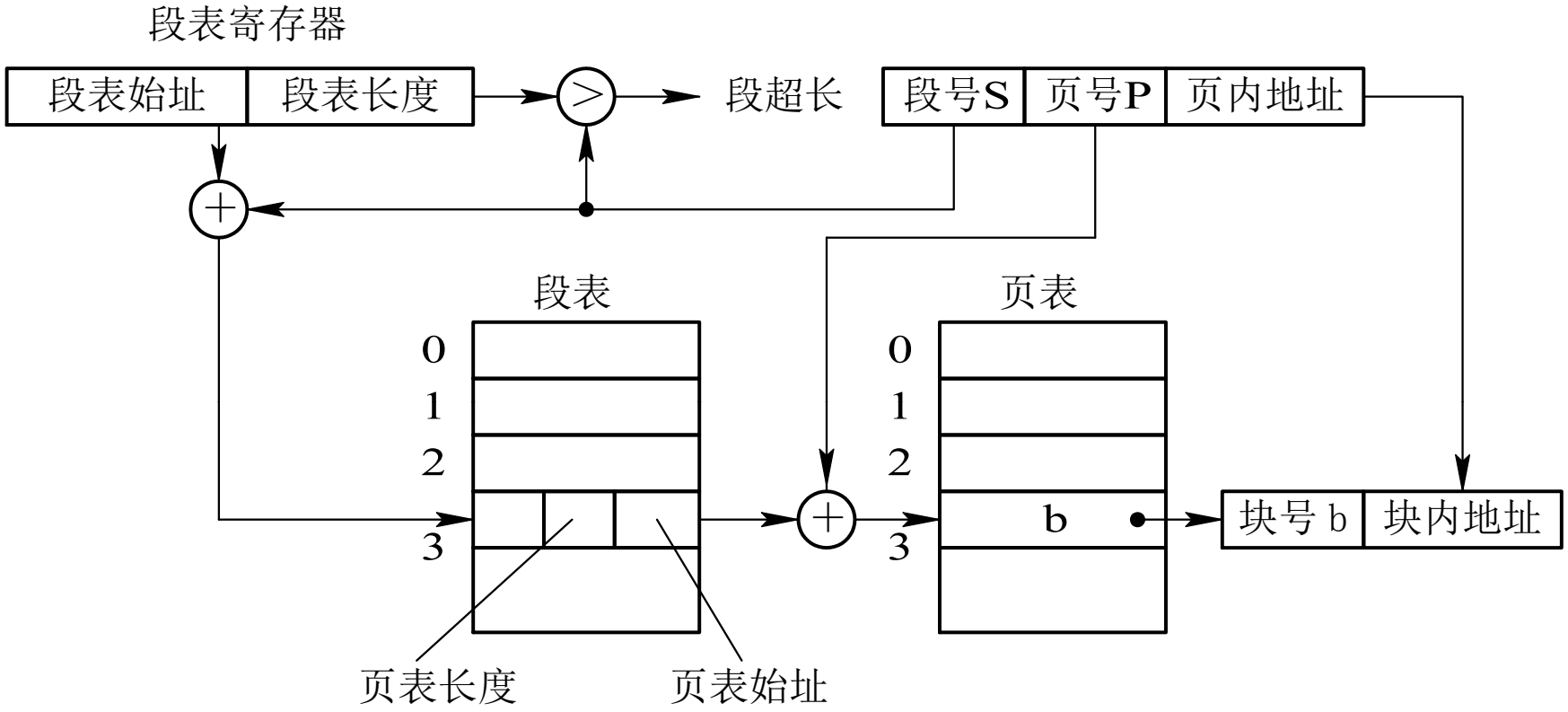
作业地址空间和地址结构



利用段表和页表实现地址映射

2．地址变换过程

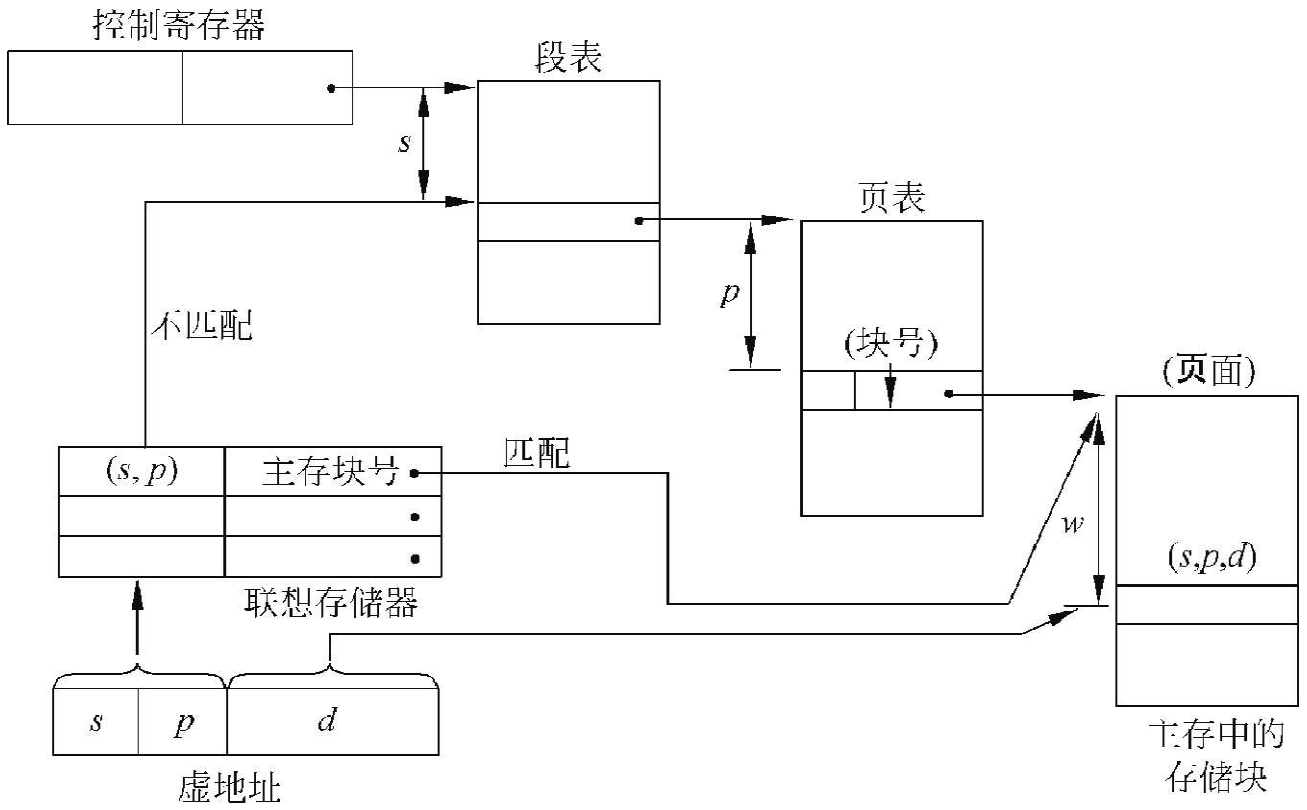
在段页式系统中，为了便于实现地址变换，须配置一个段表寄存器，其中存放段表始址和段表长TL。进行地址变换时，首先利用段号S，将它与段表长TL进行比较。若S<TL，表示未越界，于是利用段表始址和段号来求出该段所对应的段表项在段表中的位置，从中得到该段的页表始址，并利用逻辑地址中的段内页号P来获得对应页的页表项位置，从中读出该页所在的物理块号b，再利用块号b和页内地址来构成物理地址。下图示出了段页式系统中的地址变换机构。



段页式系统中的地址变换机构

在段页式系统中，为了获得一条指令或数据，须三次访问内存。第一次访问是访问内存中的段表，从中取得页表始址；第二次访问是访问内存中的页表，从中取出该页所在的物理块号，并将该块号与页内地址一起形成指令或数据的物理地址；第三次访问才是真正从第二次访问所得的地址中，取出指令或数据。

地址结构



5.6.2 段页式存储管理的其它问题：

增加系统管理开销；

增加硬件支持和内存占用；

页内碎片；

如果不采用联想寄存器，系统执行速度会大大降低。

5.7 Linux存储管理

Linux系统采用了虚拟的内存管理机制,即交换和请求分页存储管理技术

5.7.1 进程虚存空间的管理

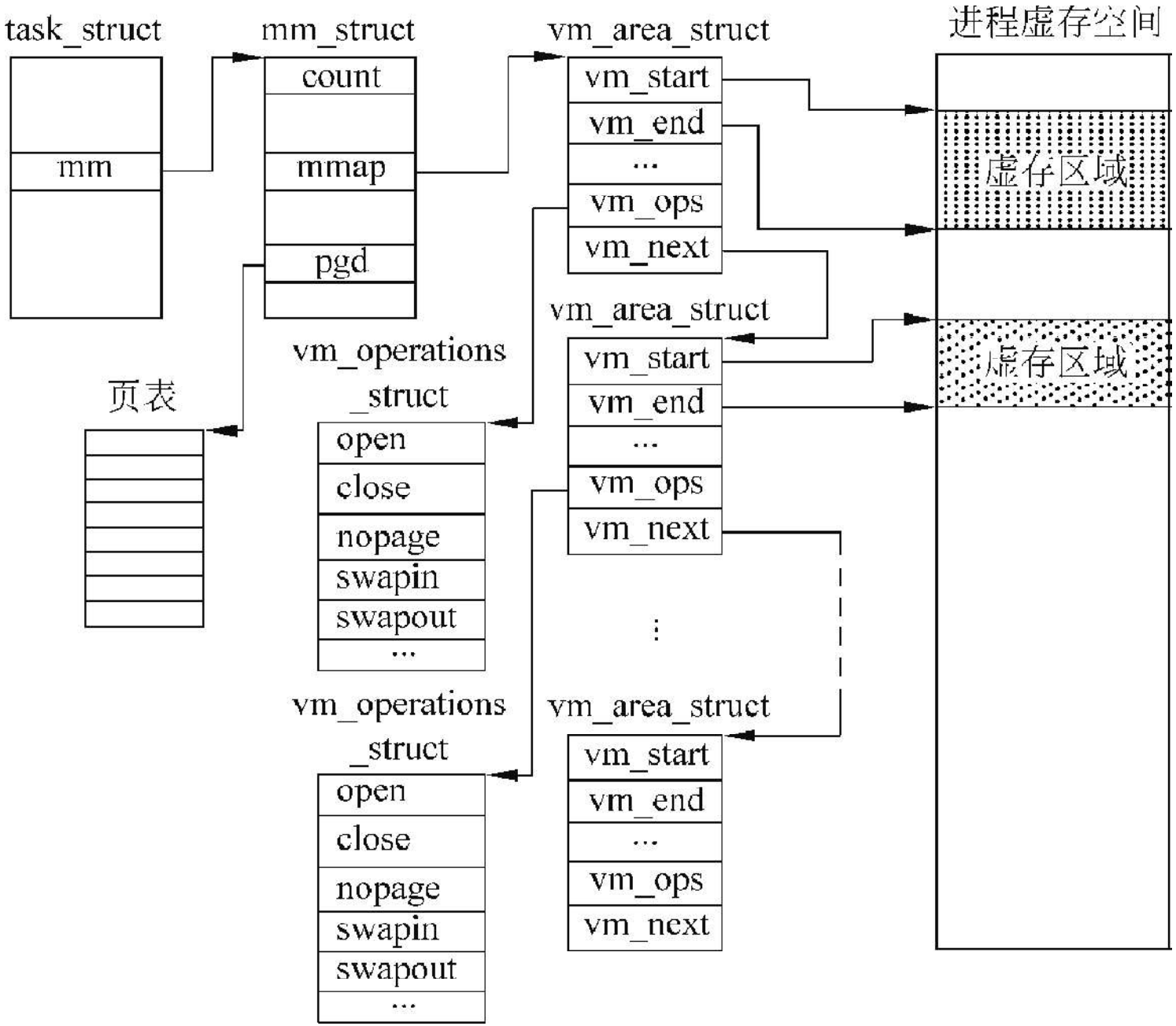
1. 地址空间

进程虚拟内存的用户区分成代码段、数据段、堆栈以及进程运行的环境变量、参数传递区域等。进程在运行中还必须得到操作系统的支持,进程的虚拟内存中还包含操作系统内核。

2. 进程的虚存区域

一个虚存区域是虚拟内存空间中一个连续的区域,在这个区域中的信息具有相同的操作和访问特性。每个虚存区域用一个vm_area_struct结构体进行描述:

```
        structmm_struct    *mm;
 structmm_struct{
        intcount;                                                    //虚存区域的个数
        pgd_t * pgd;        //为指向进程页表的指针
        unsignedlongcontext; //进程上下文的地址
        … //代码段、数据段、堆栈的首尾地址等
        structvm_area_struct * mmap;    //内存区域链表
        structvm_area_struct * mmap_avl;    //VM形成的红黑树
        structsemaphoremmap_sem;    //虚存区的信号量
 }
   structvm_area_struct
{
      structmm_struct * vm_mm; //指向进程的mm_struct结构体
      unsignedlongvm_start; //虚存区域的开始地址
      unsignedlongvm_end;    //虚存区域的终止地址
      pgprot_tvm_page_prot;    //虚存区域的页面的保护特性
      unsignedshortvm_flags; //虚存区域的操作特性
      … //AVL结构
      structvm_operations_struct * vm_ops;    //相关操作表
      unsignedlongvm_offset;    //文件起始位置的偏移量
      structinode * vm_inode;    //被映射的文件
};
```


task_struct
mm
mm_struct
count
mmap
pgd
vm_area_struct
vm_start
vm_end
...
vm_ops
vm_next
进程虚存空间
虚存区域
虚存区域
vm_area_struct
vm_start
vm_end
...
vm_ops
vm_next
页表
vm_operations
_struct
open
close
nopage
swapin
swapout
...
vm_operations
_struct
open
close
nopage
swapin
swapout
...
vm_area_struct
vm_start
vm_end
...
vm_ops
vm_next

3. 虚存空间的映射和虚存区域的建立

Linux使用do_mmap()函数完成可执行映像向虚存区域的映射,由它建立有关的虚存区域

5.7.2 Linux的分页式存储管理

1. 物理内存的页面管理

```
typedefstructpage{
        structpage *next;    //把page结构体链接成一个双向循环链表
        structpage *prev;
        structinode *inode; //有关文件的i节点
        unsignedlongoffset; //在文件中的偏移量
        structpage *next_hash; //page结构体连成一个哈希表
        atomic_tcount;    //页面的引用次数
        unsignedflags; //页面的状态
        unsigneddirty:16,age:8; //表示该页面是否被修改过
        structwait_queue *wait;    //等待该页面的进程队列
        …
        unsignedlongmap_nr;    //物理页号
}mem_map_t;
```

2. Linux的三级分页结构

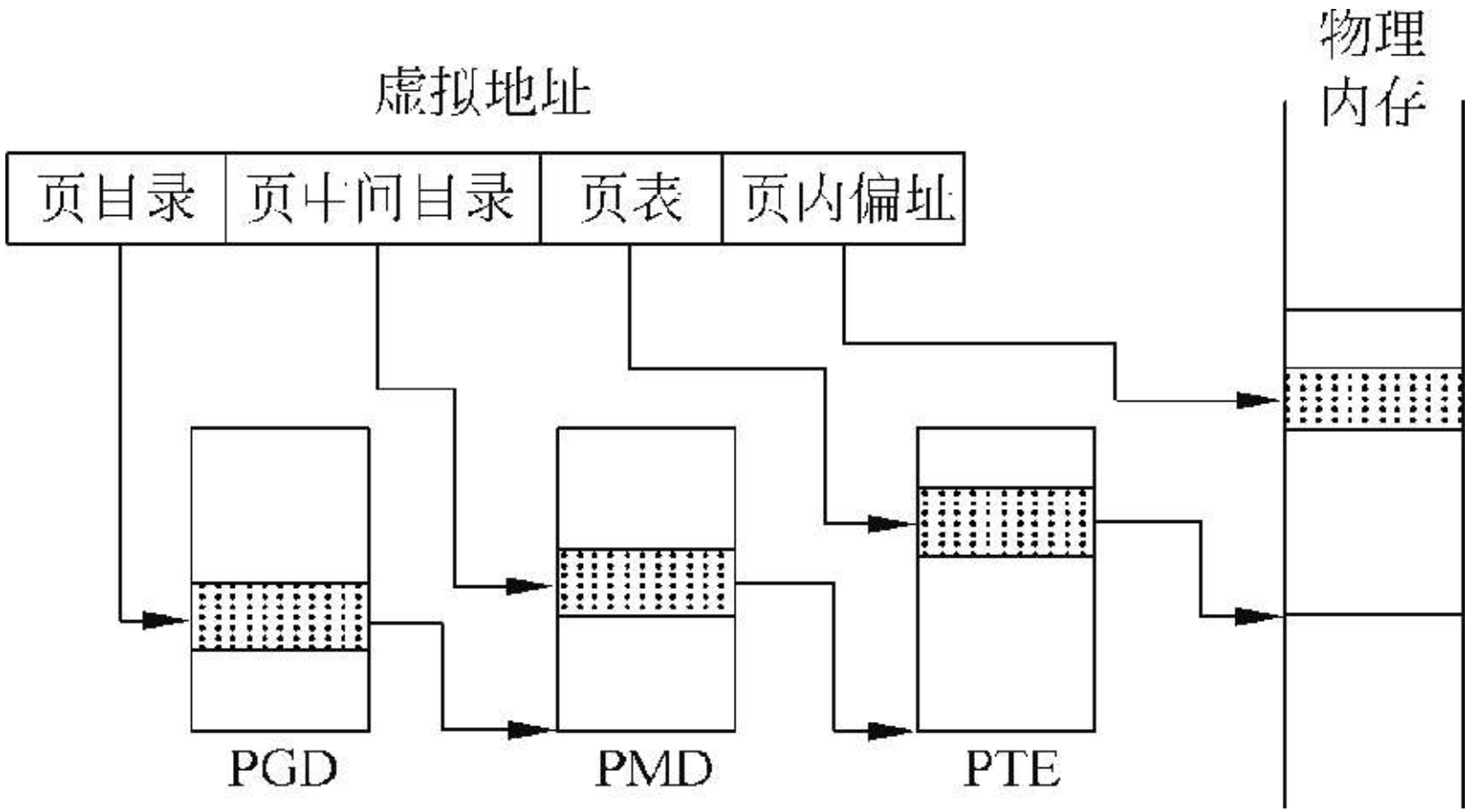


图5-43　Linux的三级分页管理

3. 内存的分配与释放

Linux中用于内存分配和释放的函数主要是kmalloc()和kfree(),它们用于分配和释放连续的内存空间。

在使用kmalloc()分配空闲块时以Buddy算法为基础。对kmalloc()分配的内存页面块中加上一个信息头,它处于该页面块的前部。页面块中信息头后的空间是可以分配的内存空间。

4. 虚拟内存的申请和释放

申请较大的内存空间时,使用vmalloc()。由vmalloc()申请的内存空间在虚拟内存中是连续的,它们映射到物理内存时,可以使用不连续的物理页面,而且仅把当前访问的部分放在物理页面中。free()用来释放由vmalloc()分配的虚存