第5章 存储器管理

一、存储管理概述

1.存储器类型

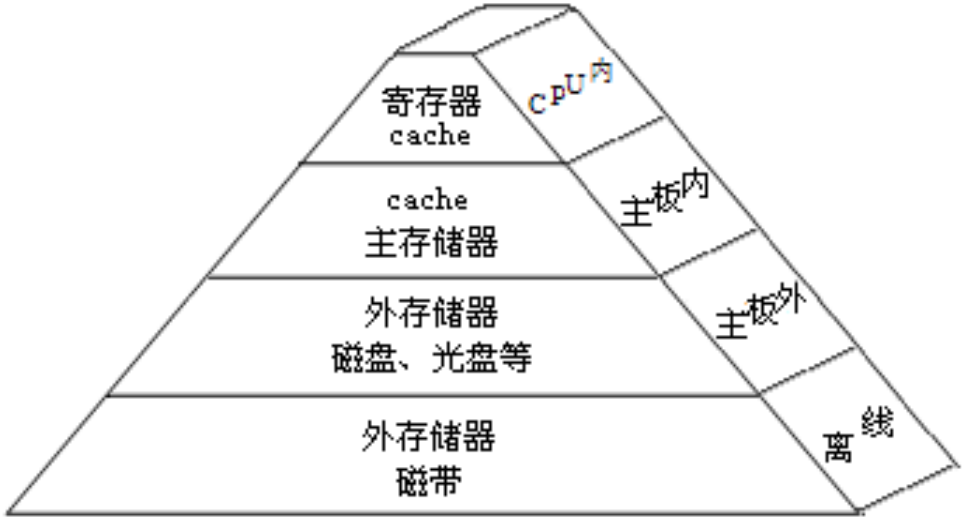

图5-1 存储器类型

2.虚拟地址(Virtual Address)和物理地址(Physical Address)

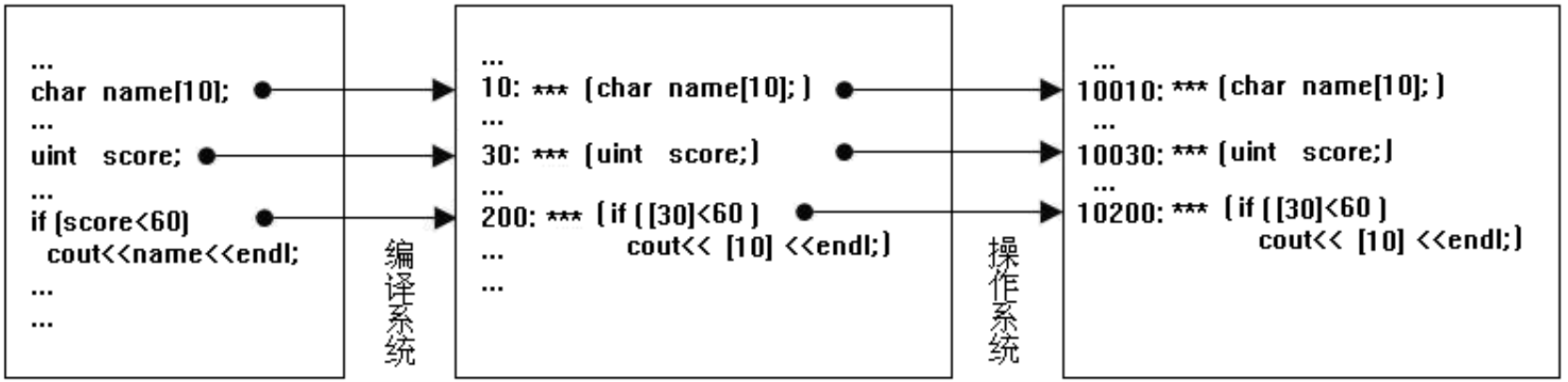

图5-2 虚拟地址、物理地址的例子

虚拟地址空间

作业大小是指作业中各程序虚拟地址空间大小的总和。

3.重定位(Relocation)/地址转换

- 程序装入(Programming Loading)
- 重定位(Relocation)
  - 静态重定位(Static Relocations)
  - 动态重定位(Dynamic Relocation)

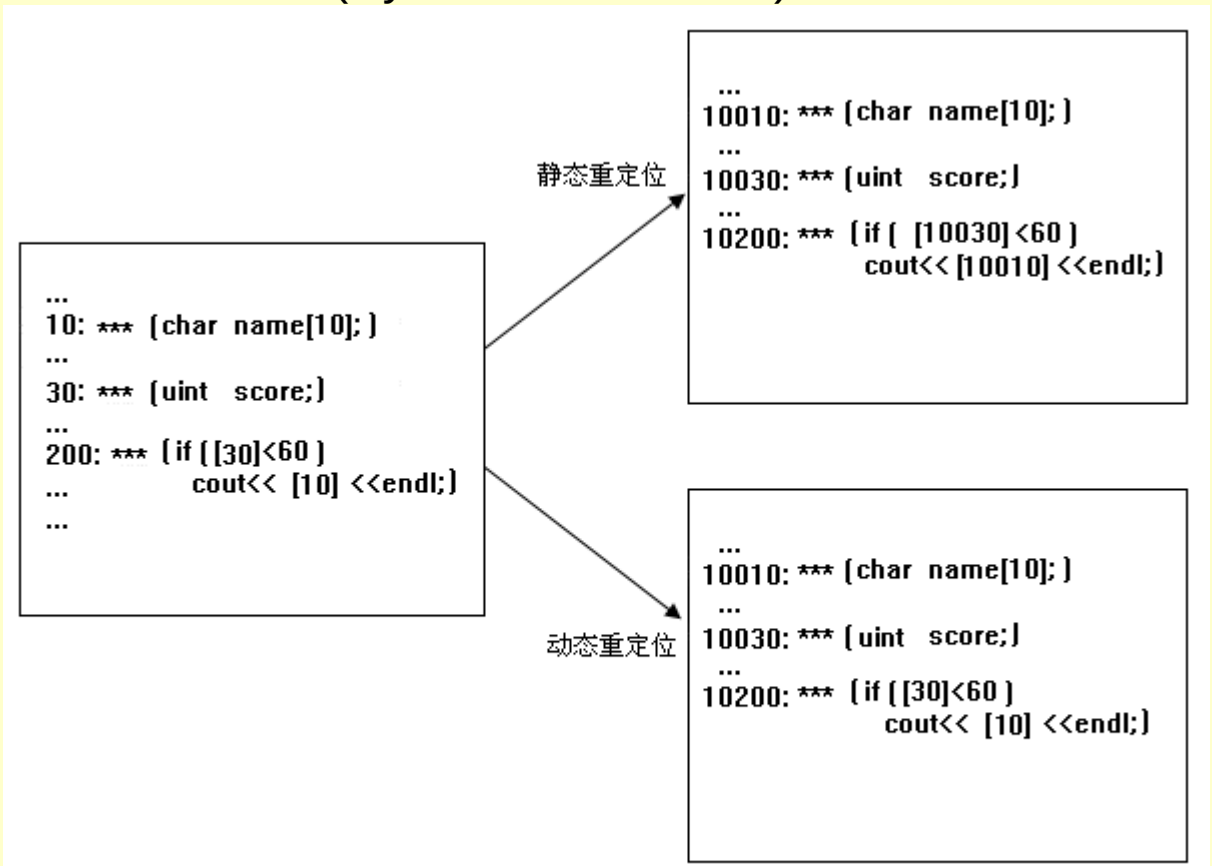

图5-3 两种重定位方式的例子

4.存储管理的目的

- 系统空间(System Space)和用户空间(User Space)
- 存储管理目的
  - 提高主存储器的利用率
  - 方便用户对主存储空间的使用

5.存储管理的主要功能

- 存储空间的分配和回收
  - 设计合理适的数据结构，登记存储单元的使用情况
  - 设计分配算法
  - 存储空间回收
- 重定位
- 存储空间的共享与保护
  - 界限寄存器法
  - 保护键法
  - 界限寄存器和CPU工作模式
- 虚拟存储器

二、单一连续区存储管理

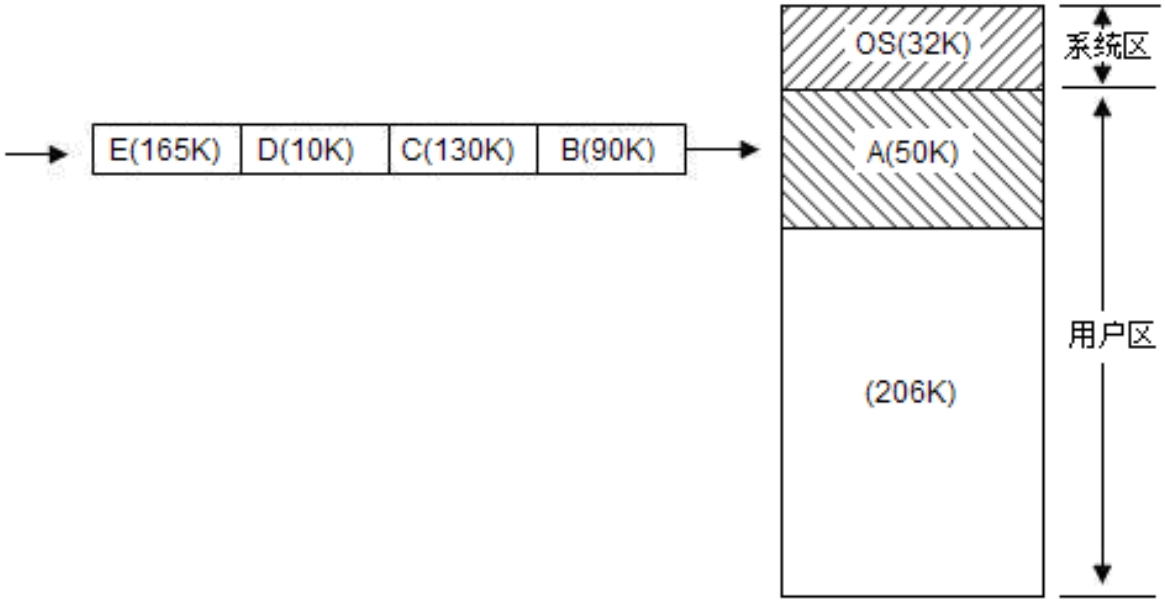

图5-4 单一连续区存储管理

三、固定分区存储管理

1.基本思想

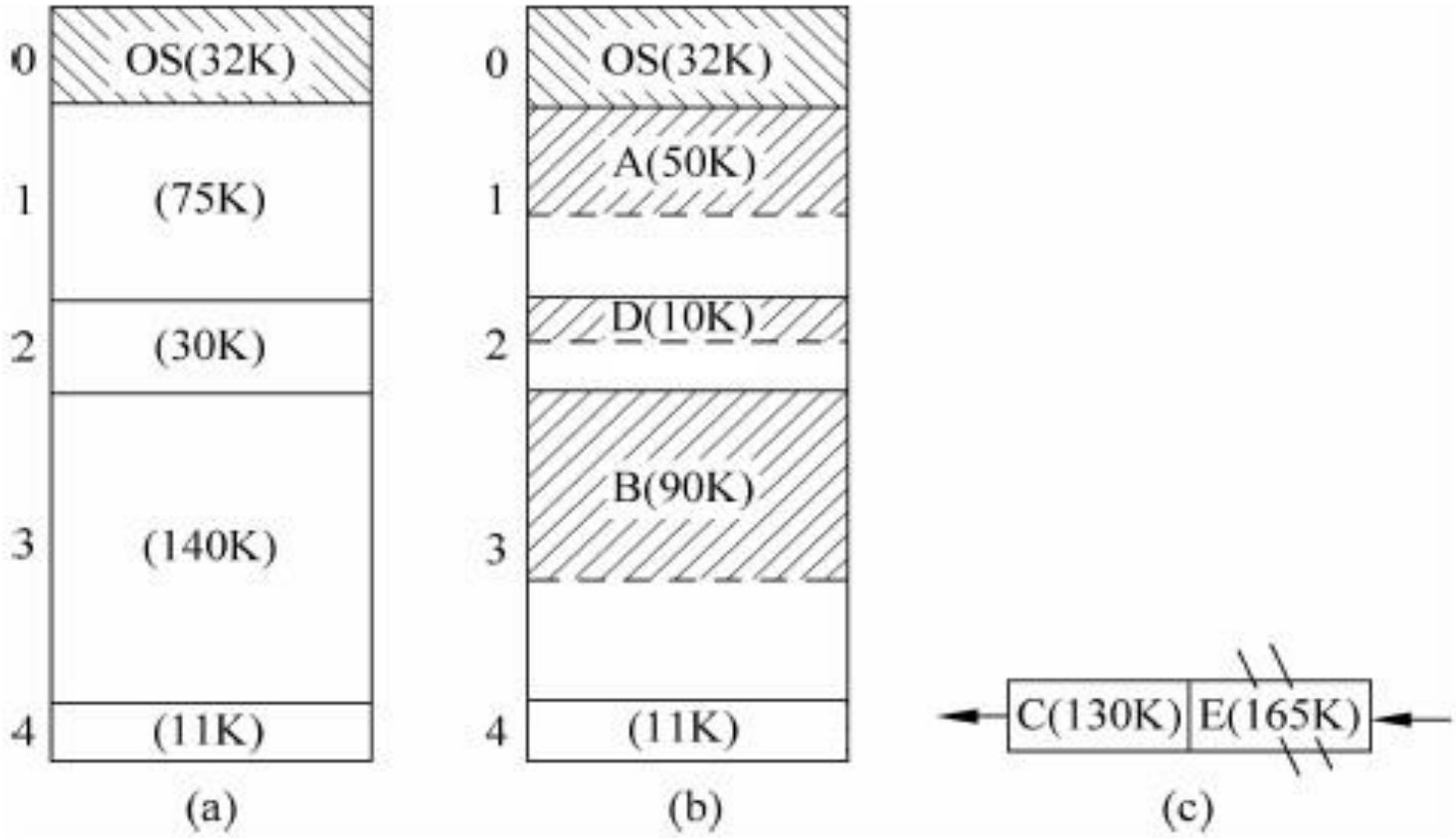

图 5-5　固定分区

2.实现关键

- 数据结构设计

分区说明表(DPT, Descriptive Partition‘s Table)由分区号、起始地址、分区长度和状态组成

表5-1 分区说明表结构及初始化

区号	长度	起始地址	状态
1	75K	32K	0
2	30K	107K	0
3	140K	137K	0
4	11K	277K	0

- 分配和回收

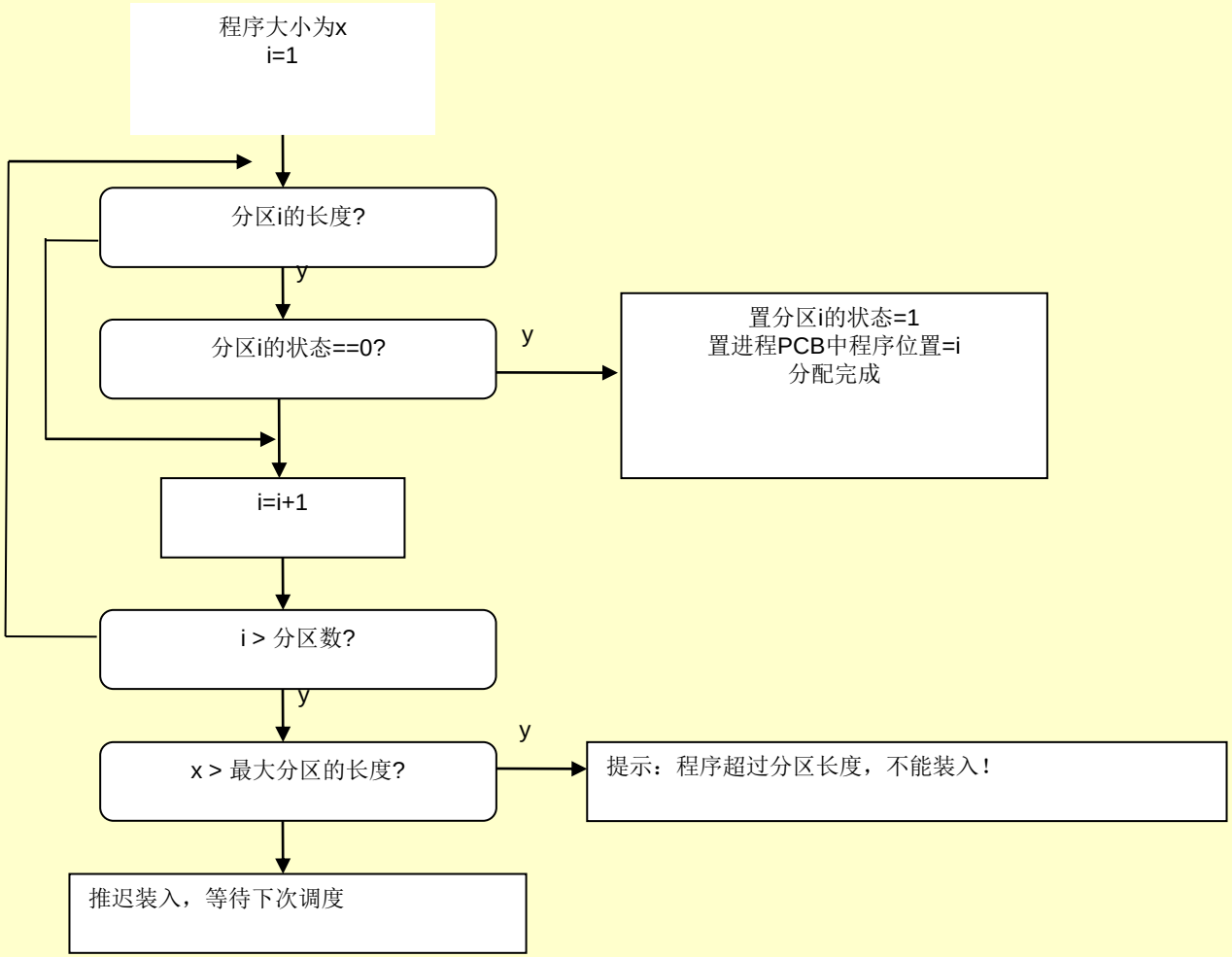

图5-6 固定分区的分配流程

- 重定位和存储保护
- 主要特点
  - 能够支持多道程序设计
  - 并发执行的进程数受分区个数的限制
  - 程序大小受分区长度的限制
  - 存在“碎片”

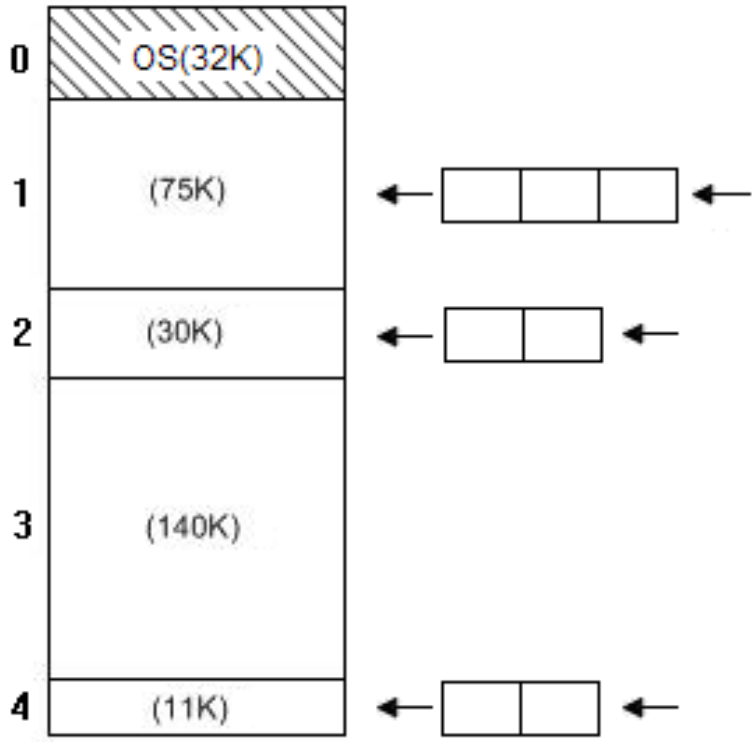

图5-7 多输入队列的固定分区

四、可变分区存储管理

1.基本思想

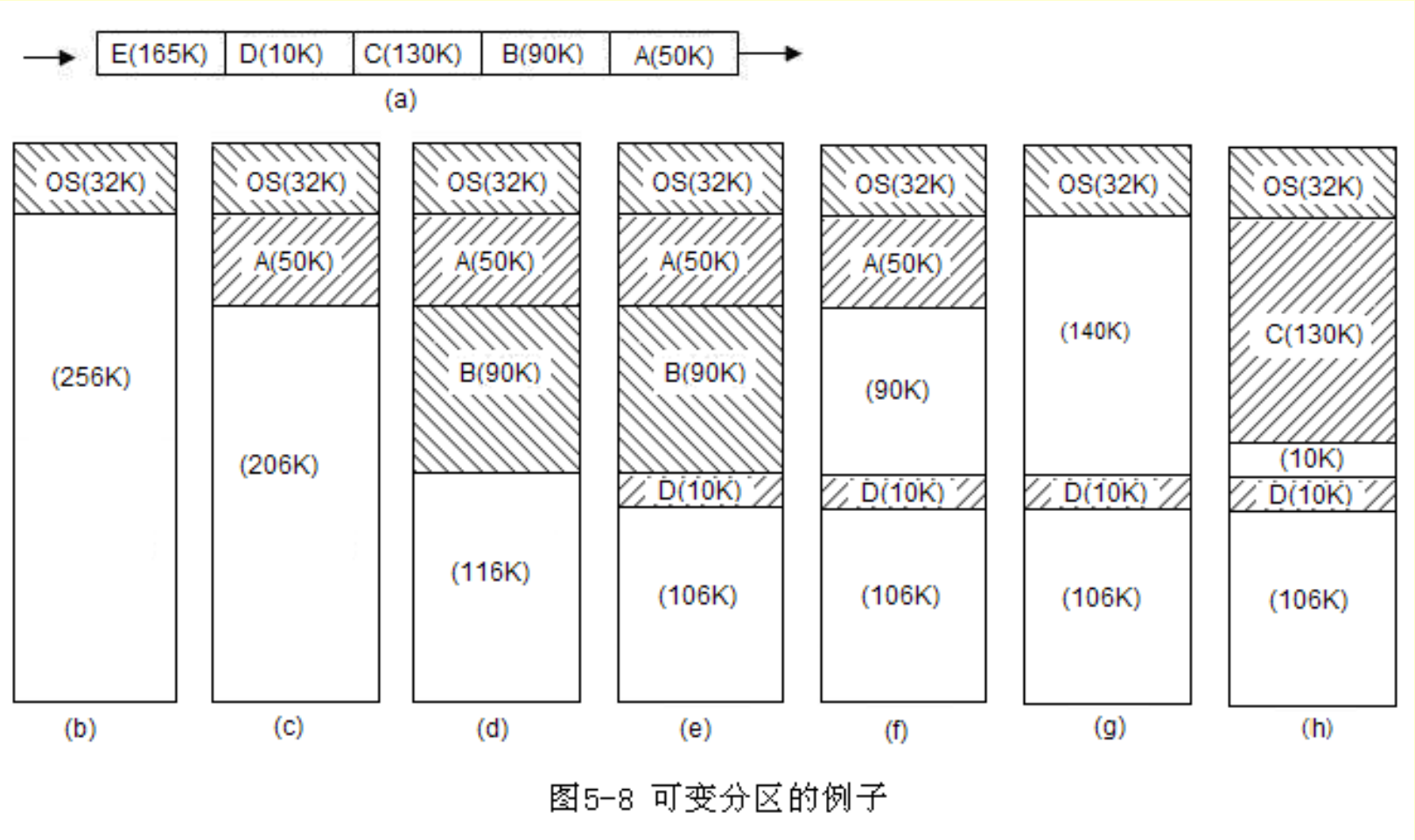

图5-8 可变分区的例子

2.实现关键

- 数据结构设计
  - 可用表
  - 空闲区链表

```
struct FreeNode {
        long  start;                          //分区的起始地址
        long  length;                         //分区的长度
        struct FreeNode *next;                //向下指针
        }*freePartitionsList;                 //空闲区链表头指针
```

  - 请求表

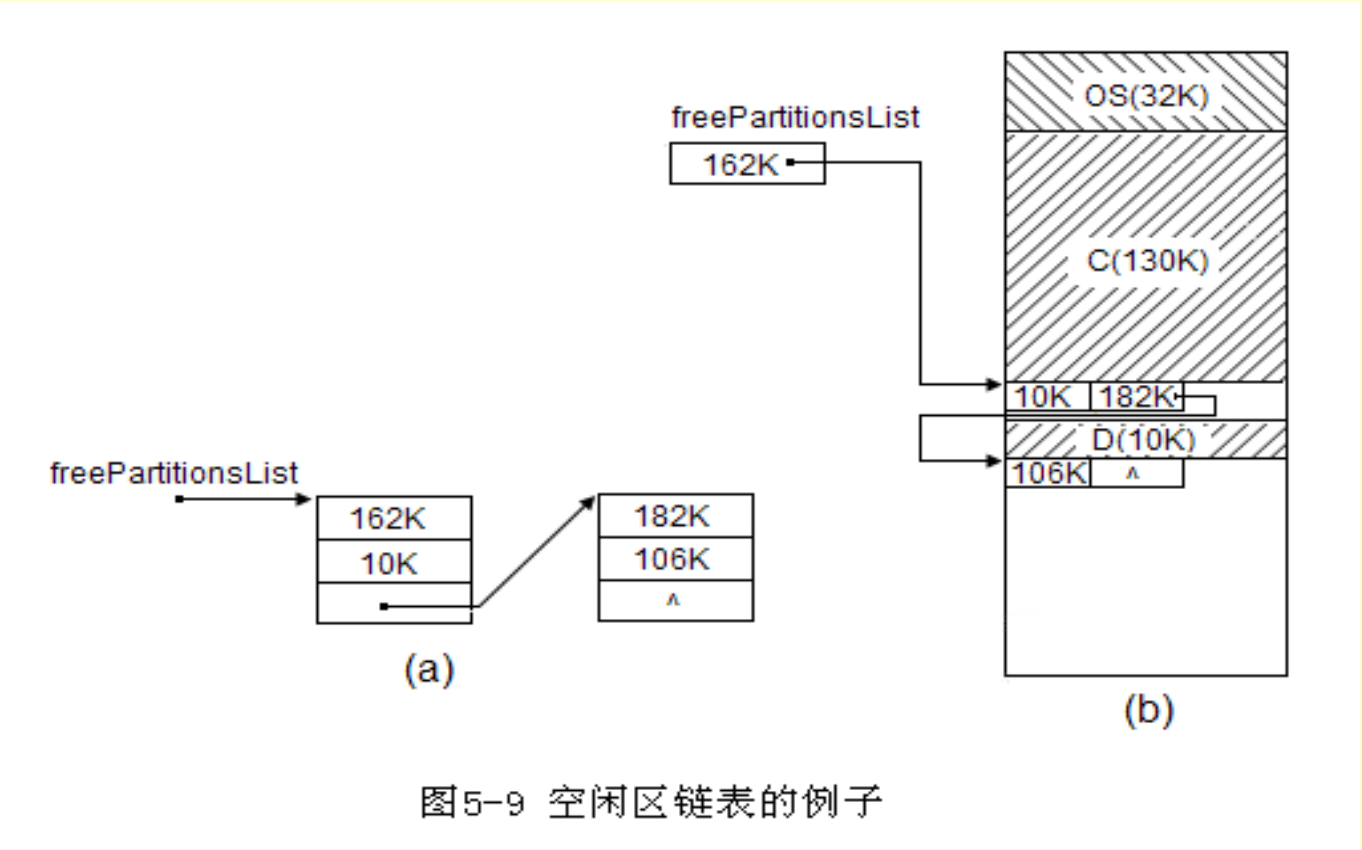

图5-9 空闲区链表的例子

- 分配
  - 动态分区
  - 存储空间分配的基本策略
    - 最先适应法(FF，First Fit)
    - 最佳适应法(BF，Best Fit)
    - 最坏适应法(WF，Worst Best Fit)

程序A、B、C和D，它们的虚拟地址空间大小分别是80K、30K、130K和25K。

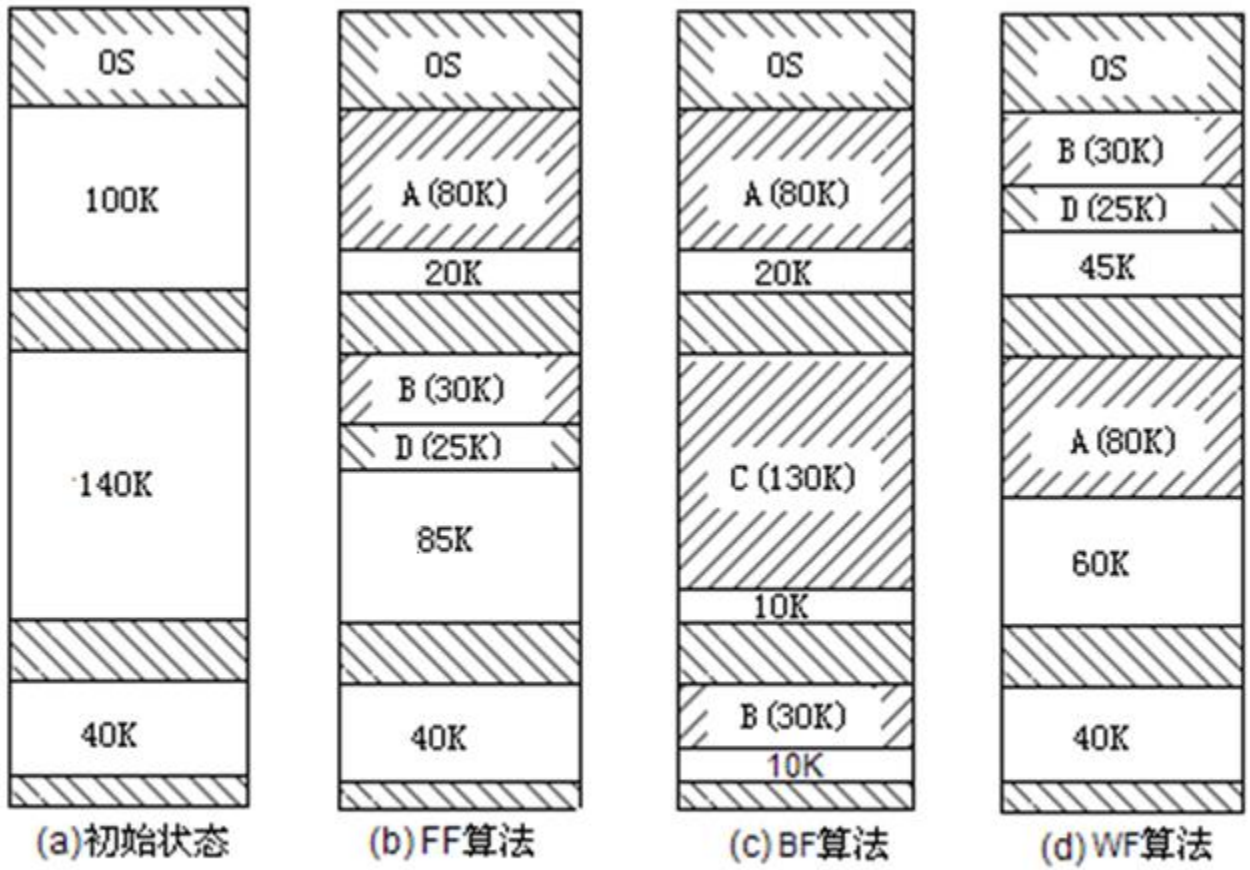

图5-10 三种基本分配策略的例子

- 回收
  - 合并判断

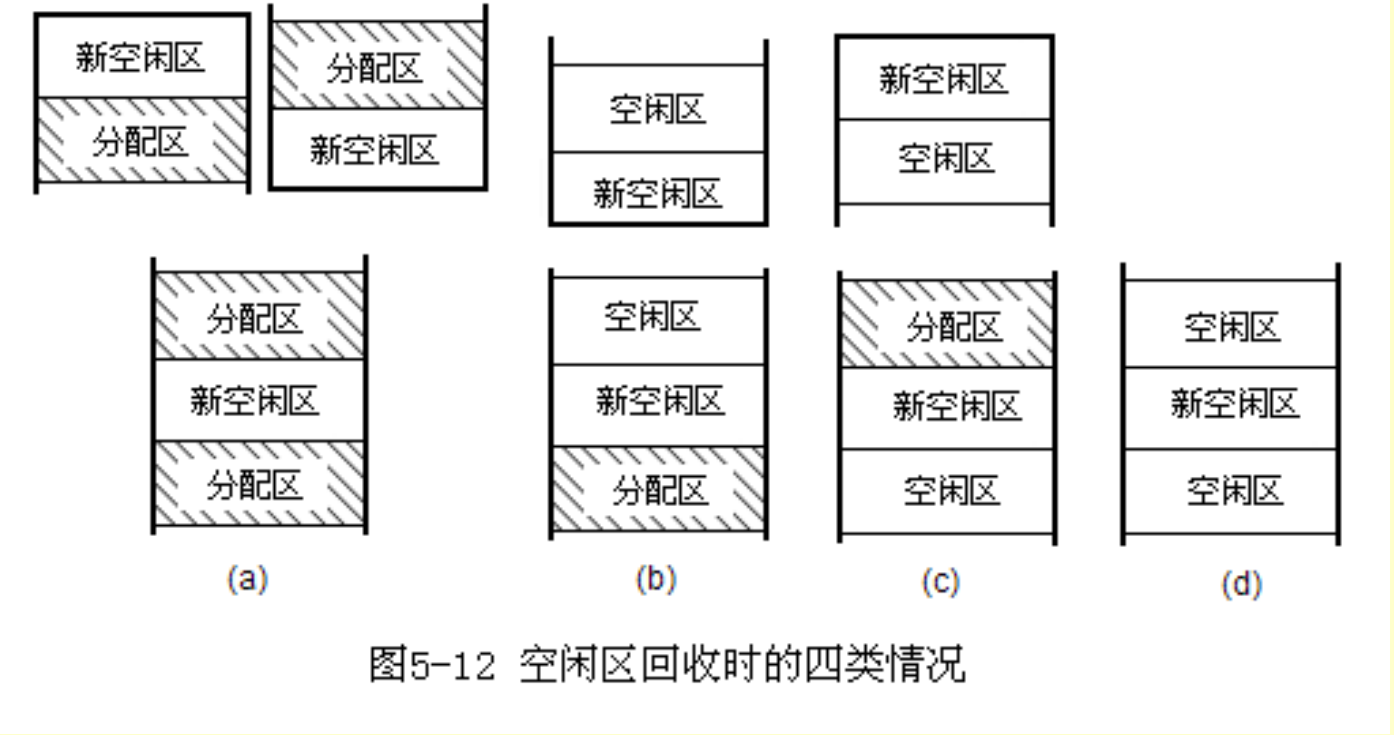

图5-12 空闲区回收时的四类情况

在回收一个分区时，系统中空闲区的个数变化情况是：满足图5-12(a)类型时，空闲区个数增加1个；满足图5-12(b)或(c)类型时空闲区个数不变；而满足图5-12(d)类型时，空闲区的个数反而减少1个。

- 重定位及存储保护
  - 动态重定位
  - 界限寄存器法

3.移动技术

4.主要特点

- 存在外碎片(External Fragmentation)，降低了存储空间的利用率
- 分区个数和每个分区的长度都在变化
- 为进程的动态扩充存储空间提供可能
- 需要相邻空间区的合并，增加系统的开销
- 基本分配算法FF、BF和WF，在存储空间利用率上没有很大差别

5.分区管理总结

- 存储空间连续分配，管理方法容易实现
- 存在碎片，存储空间利用率不高(**内碎片和外碎片的区别**)

  从存储单元的状态看，内碎片是分配状态，外碎片是空闲状态

  从长度看，内碎片的长度可能很大，但外碎片的长度往往比较小。
- 程序大小受分区的限制

5.对换(Swaping)和覆盖((Overlay))

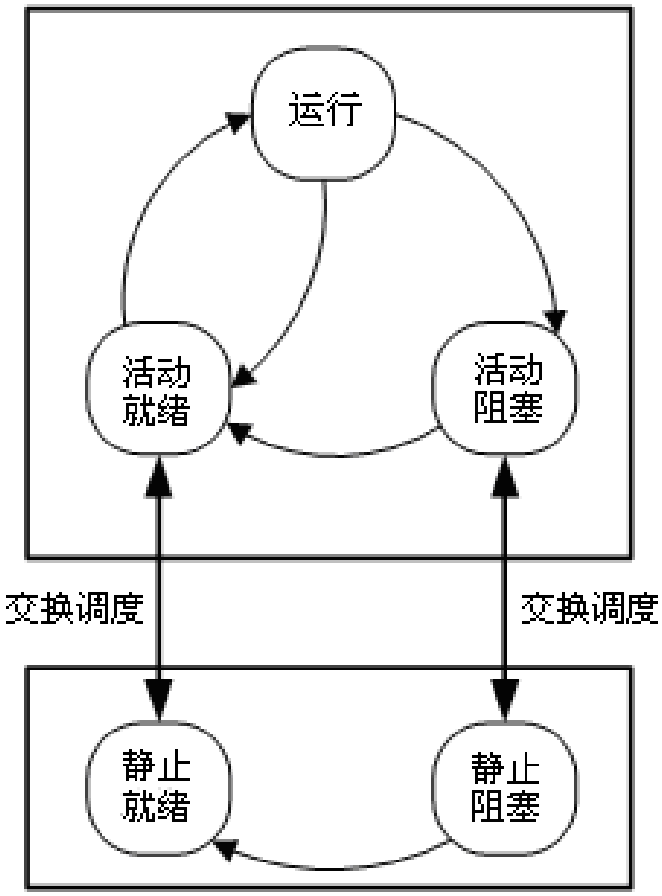

图5-13 具有对换技术的进程状态

对换技术是由操作系统实现，程序员或用户看不到进程的调出/调入过程

对换技术可以增加并发执行的进程数，或者使得当前运行的进程拥有更多的可用存储空间

覆盖技术(Overlay)是早期操作系统(DOS)采用的一种内存逻辑扩充技术

程序员：可覆盖结构设计

操作系统提供关于内存空间的分配、撤销和设置(Setblock)等存储管理的系统调用，以及程序装入或程序装入并运行等的进程管理的系统调用（也称为EXEC功能）

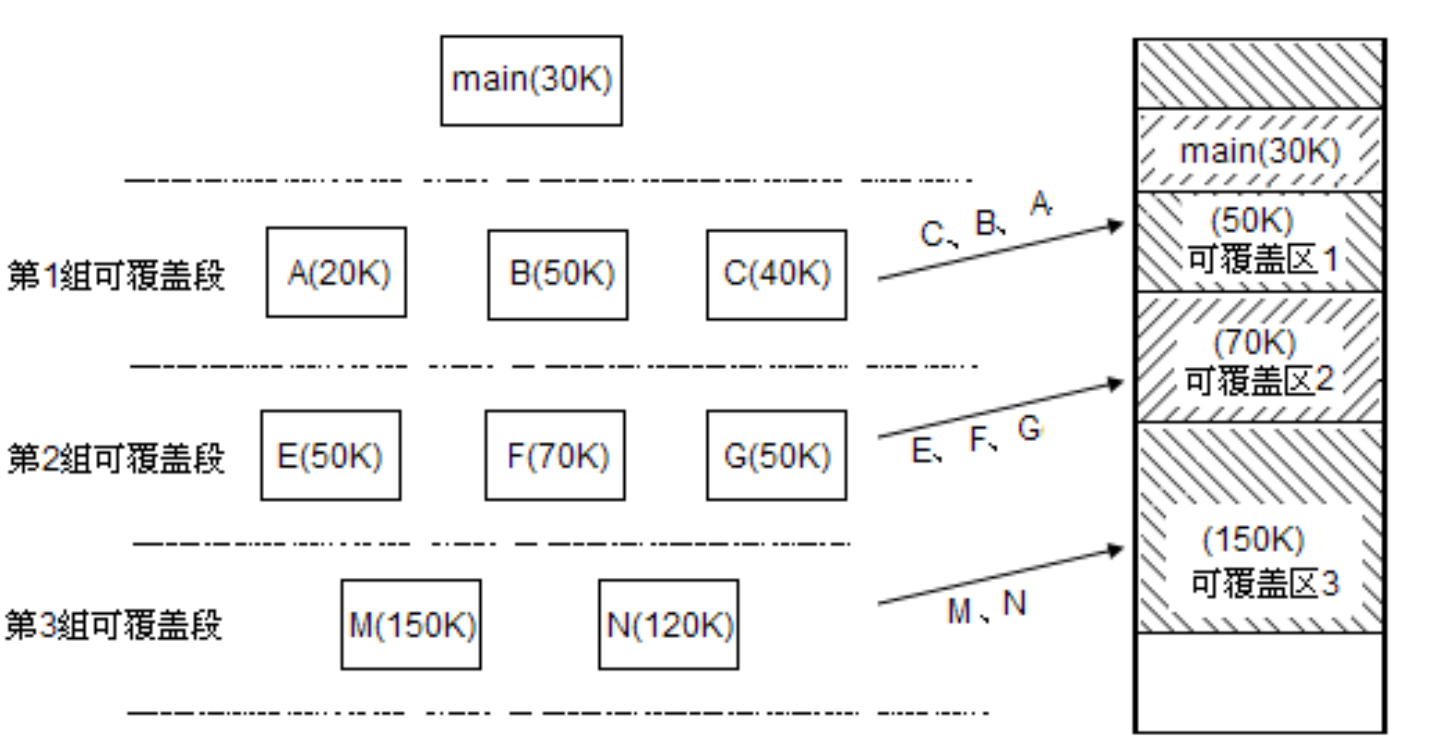

图5-14 可覆盖结构设计例子

五、分页存储管理

1.基本思想

- ✓ 内存分块
- ✓ 进程分页
- ✓ 非连续分配

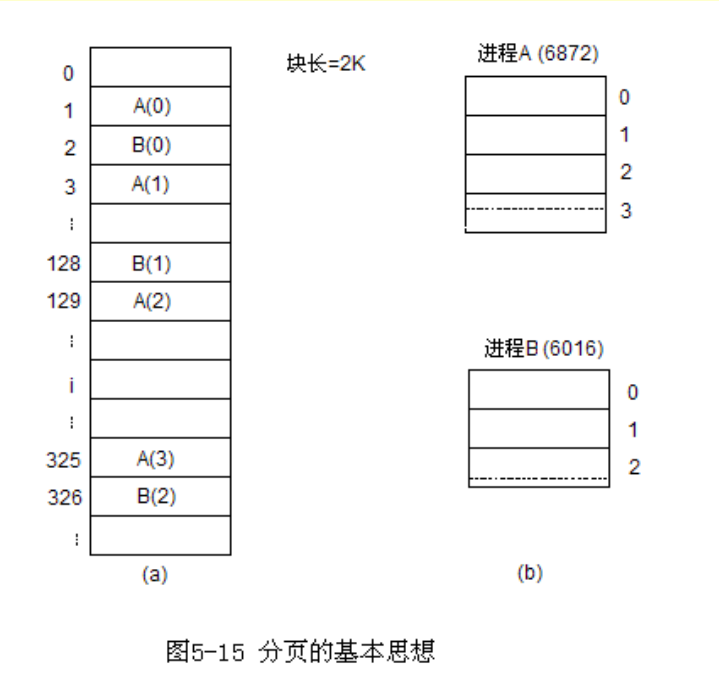

图5-15 分页的基本思想

分页存储管理由操作系统和硬件共同实现

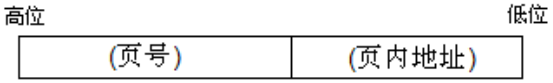

图5-16 虚拟地址结构

由虚拟地址a计算页号p和页内地址w的两种方法：

$p=a>>k,$

$w = a \,\&\, \underbrace{(11\ldots1)_2}_{k}$

$p=a/b$

$w=a\%b$

分页存储管理又分为静态分页和动态分页两种

2.静态分页的实现关键

✓ 位示图(bitmap)及其作用

假定某内存空间共256个块，机器字长为16位，那么，表示内存块使用状况的位示图，如图5-17所示。

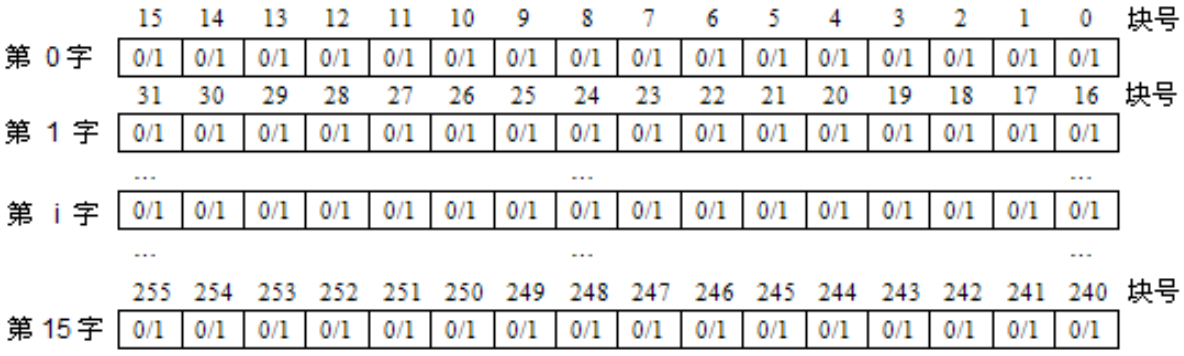

图5-17 位示图的例子

假定，在位示图中的一个位用bitmap[i,j]表示，其中i 称为**字号**，表示第i行即第i个字；j称为**位号**，表示在第i个字中的第j位，这里规定从低位开始计算。如果位示图中的第i个字记为bitmap[i]，那么

bitmap[i,j]=(bitmap[i] >>j)&1

位示图的一个位bitmap[i,j]表示的块号为b，可以计算得到

b=字长*i+j

相反地，如果已知一个块的块号，那么，这个块在位示图中的位bitmap[i,j]，则有

i=b/字长

j=b%字长

- ✓ 空闲块链表
- ✓ 页表(Page Table)及其作用

每一个进程都有一个页表，
页表描述进程页与块的对应关系。

页表的主要作用是重定位和存储保护

页表的建立和初始化过程-内存分配

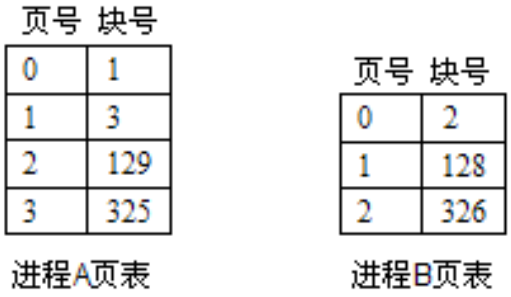

图5-18 页表的例子

✓ 重定位及存储保护

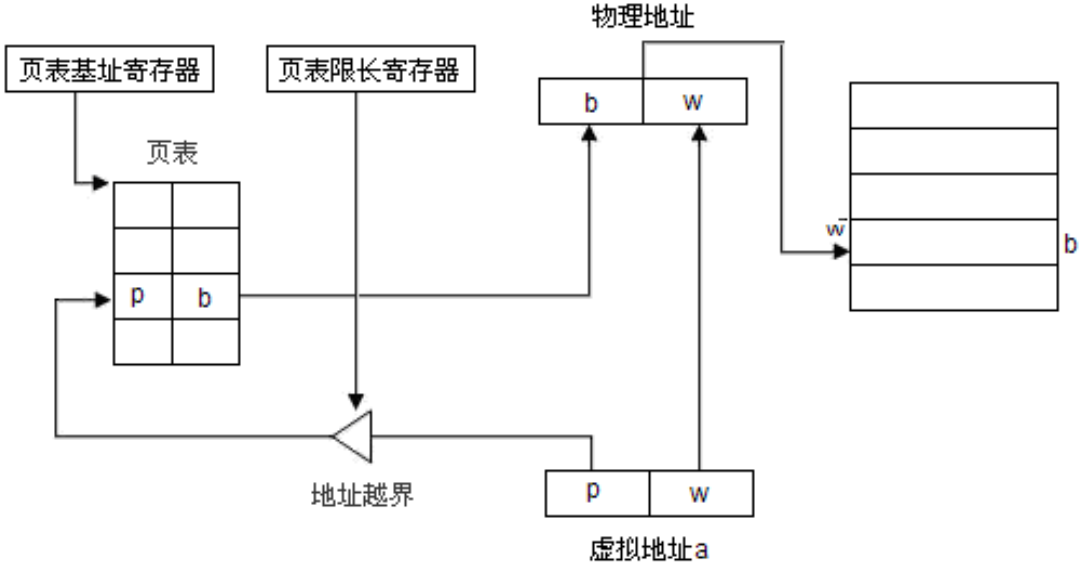

图5-21 分页的重定位

重定位过程，其步骤概括如下：

1)页号p和页内地址w

2)存储保护

3)利用页表得到块号

4)形成物理地址

例子 在某静态分页存储管理中，已知内存共的32块，块长度为4K，当前位示图如图5-22所示，进程P的虚拟地址空间大小为50000。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	1	1	0	1	1	1	1	1
0	0	0	0	0	1	1	0	0	0	1	0	0	1	1	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

图 5- 22 位示图的例子

问：
(1)进程P共有几页？
(2)根据图5-22的位示图，给出进程P的页表。
(3)给定进程P的虚拟地址： 8192(十进制)和0x5D8F(十六进制)，根据(2)的页表，分别计算对应的物理地址。

3.静态分页的特点及效率的改进

CPU每访问一条指令或一个数据，都要2次访问内存：

1)在MMU重定位过程中，根据页号查找内存中的页表得到块号

2)CPU根据重定位得到的物理地址访问内存中的指令或数据

TLB(Translation Lookaside Buffers) -快表

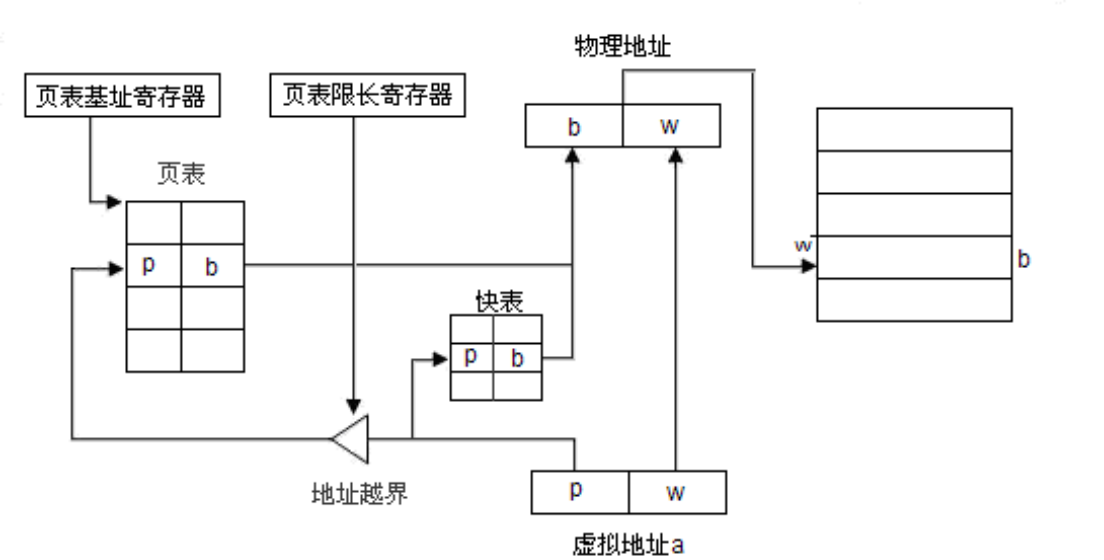

图5-24 具有快表的重定位

给定一个时间段，MMU在访问页对应的块时，如果有m次在快表中得到块号，有n次在页表得到块号，那么，这个时间段MMU的快表**命中率**h:

$$h=\frac{m}{m+n}*100\%$$

4.虚拟存储器思想

虚拟存储器要解决的主要技术有：理论基础、调入策略和置换算法

- 理论基础--**程序的局部性原理**

在程序运行过程的一个较小时间范围内，只需要一小部分的程序信息，其他部分暂时不需要；而且在程序的一次执行过程，程序的所有指令和数据并没有相同的访问概率，有一部分指令和数据经常被访问，有一部分指令和数据很少被访问，甚至存在部分指令和数据根本没有被访问。

程序的局部性原理又分为**时间局部性**和**空间局部性**

- 调入策略
  - 请求调入策略
  - 预调入策略
- 置换算法

5.请求分页的实现关键

- 扩充页表

扩充页表的基本结构主要由页号、块号、外存地址、中断位P、访问位A、修改位M等组成。**修改位M的作用？**

- 缺页中断及其处理
- 页面调度

操作系统的缺页中断处理过程，要为新读入的页分配一个空闲块，如果内存没有空闲块，必须按指定的策略，从内存中选择一页将其信息淘汰，空出的块分配给新的页，把这个过程称为**页面调度**。

页面调度分为**局部页面调度**和**全局页面调度**

置换算法(Page Replacement Algorithm)，也称淘汰算法

➢ 置换算法

置换算法的目标是，在内存中尽可能保留进程运行过程中经常访问的页，以减少缺页中断的次数。

1)先进先出算法(FIFO)

一个进程运行过程依次访问的页号(也称进程的引用序列)是：0、2、3、1、4、1、2、3、5、2、3、1、4、5、0、3、6、9、8、3、6、7、3、6、9、8、7。假定分配该进程4个块，按局部页面调度，采用FIFO置换算法时，如何计算缺页中断的次数？依次淘汰的页号是哪些？

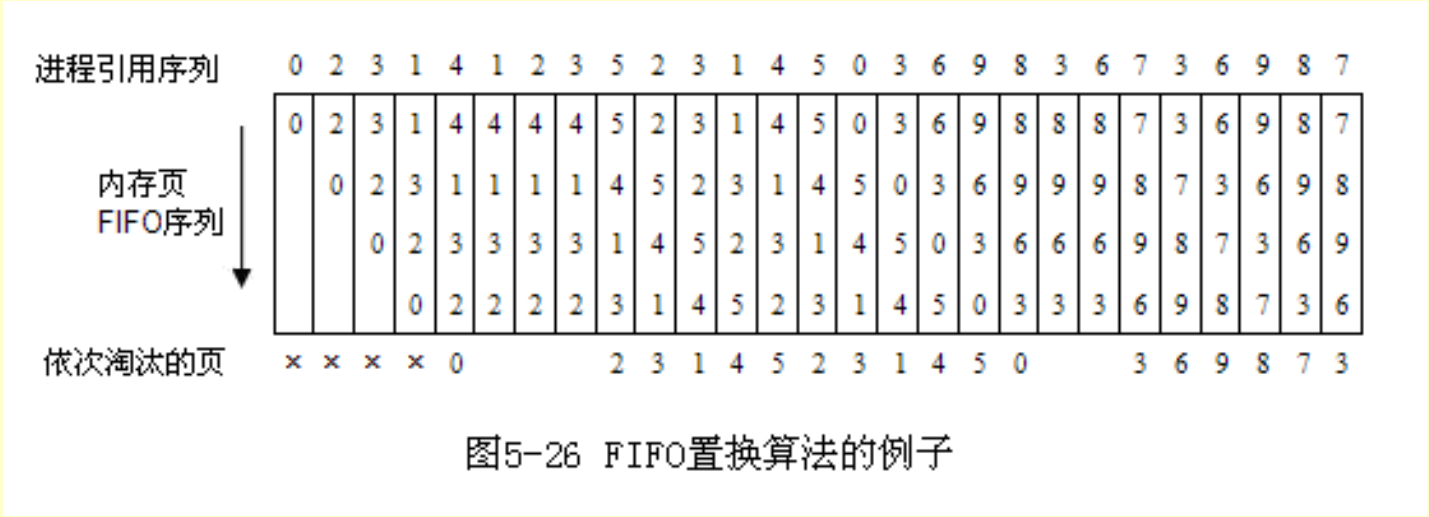

图5-26 FIFO置换算法的例子

2)最近最久未使用算法(LRU)

图5-27 LRU置换算法的例子

3)最近最不常用算法(LFU)

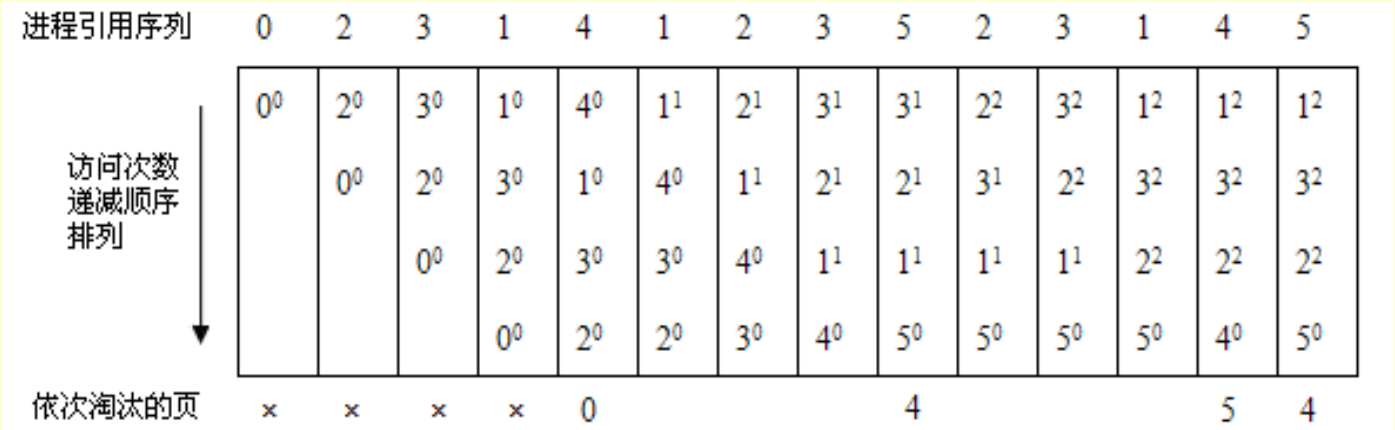

图5-28 LFU算法的例子

4)最近未使用算法(NRU)

5)二次机会算法(Second Chance)

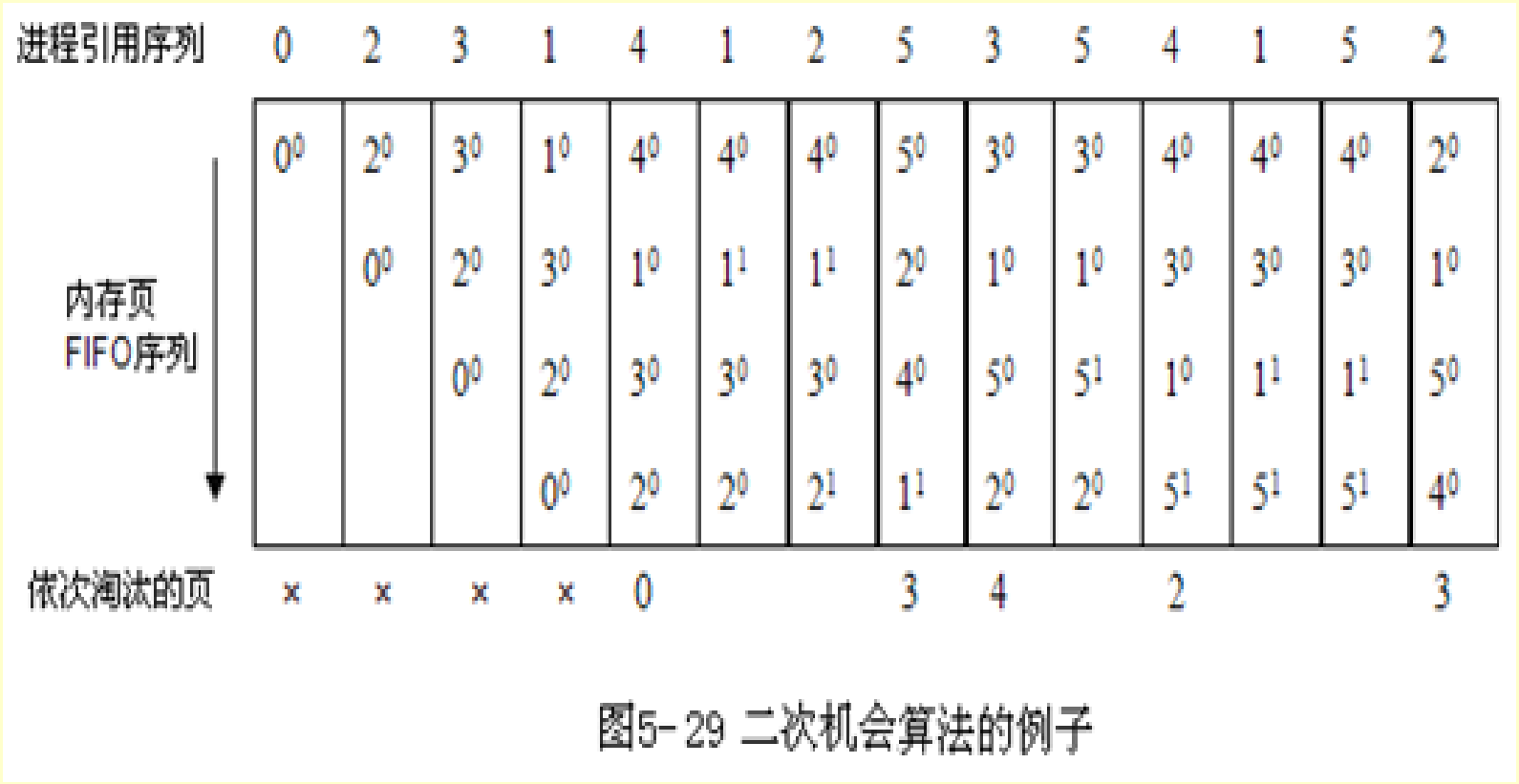

图5-29 二次机会算法的例子

二次机会算法也称为时钟算法(Clock)

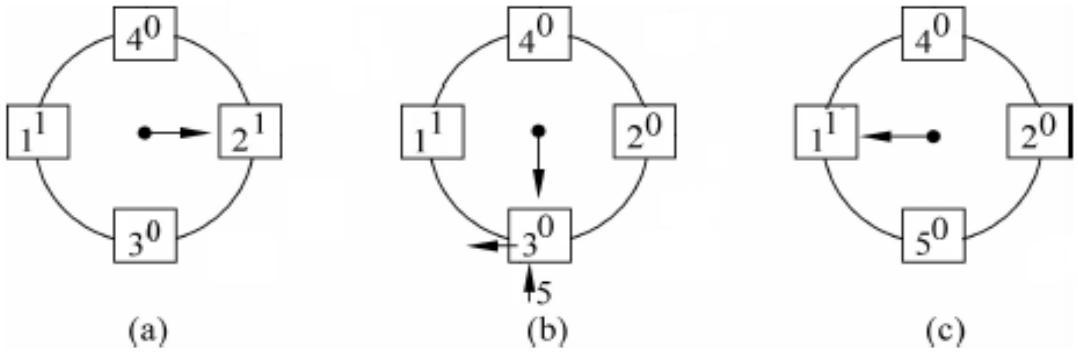

图 5-30　Clock 算法的例子 1(访问页号 5)

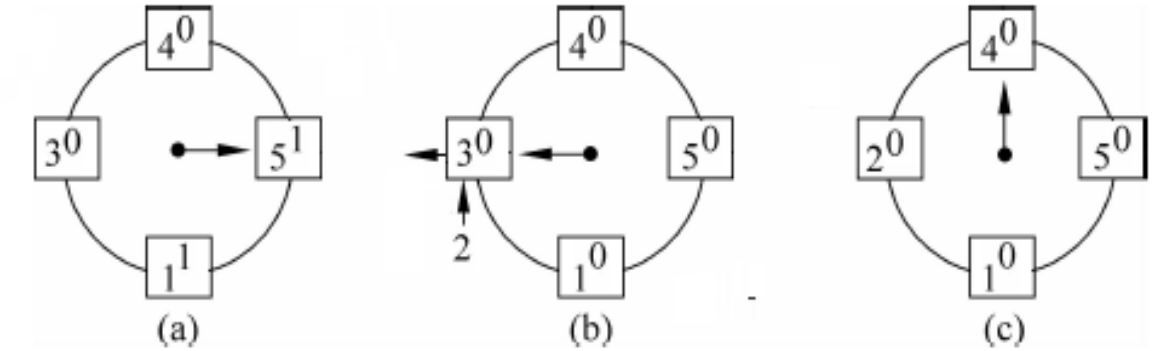

图 5-31　Clock 算法的例子 2(访问页号 2)

改进型二次机会算法思想如下：

按装入内存的时间排列，得到内存的页FIFO序列，再按访问位A和修改位M，内存的页分为如下4类：

①A=0，M=0。淘汰这类页可以减小I/O操作开销，是置换算法的首选页；

②A=0，M=1。淘汰这类页需要额外的写I/O操作，但可以减少抖动现象；

③A=1，M=0。淘汰这类页可以减小I/O操作开销，但可能产生抖动现象；

④A=1，M=1。这类页是在算法执行的最后，不得已的选择。

检查步骤：

(1)查找①类页的，如果找到则淘汰，算法完成；

(2)查找②类页的，如果找到则淘汰，算法完成；

在查找过程，置A=0.

(3)返回(1)，进行第二遍查找，因为此时所有页的A=0，所以，在第二遍查找时，最坏情况在(2)中必可找到淘汰的页。

6)页缓冲算法(Page Buffer)

设置剩余空闲块数量不足内存总块数的1/4和1/8两个界限，另外，再设置一个页缓冲区(Page Buffer)，页缓冲区用于保存2个页缓冲链表：

未修改页链表(M0链表)
修改页链表(M1链表)

在重定位发现访问页的中断位P=0时，则先在页缓冲区中M0和M1链表中查找，如果存在匹配的，则将其移出，并修改页表相关信息，如置p=1等，因为M0和M1链表中的页还没有被淘汰，所以不需要读I/O操作，可以直接从内存中得页的信息。如果在页缓冲区中M0和M1链表不存在与当前要访问的页，则产生缺页中断。

在为新读入的页分配内存时，如果没有空闲块，则可以从页缓冲区中的M0链表中移出一个页，将其对应的块分配给新的页；如果M0链表为空，则从M1链表中移出一个页淘汰，分配之前执行一个写I/O操作，或者一次性地把M1链表的所有页写入磁盘，全部淘汰，回收作为空闲区。

在产生缺页中断为新读入的页分配内存块时，如果剩余空闲块数量低于第1界限，比如不足内存总块数的1/4，则设置所有内存页的访问位A=0；如果剩余空闲块数量低于第2界限，比如不足内存总块数的1/8，将内存中访问位A= 0的页淘汰进入页缓冲区，具体作法是：置这些页的中断位P=0，再根据修改位M，把其中M=0的页加入页缓冲区的M0链表，把M=1的页加入M1修改页链表，这里，只须把能够标识页的归属的进程号和页号等信息加入链表中，这些页并没有被淘汰。

- 工作集模型(Working Set Model)

指系统设置一个跟踪程序，检查每一个进程的工作集，只有在一个进程的工作集在内存后，才允许它运行。

工作集模型可以用于内存的分配，假定系统有n个进程，当前第i个进程的工作集页数为wsi，那么，n个进程需要的内存块数

$$D=\sum_{i=1}^{n} wsi$$

当D大于内存块总数时，采用对换技术，选择一些进程从内存调出到交换区，以保证内存中剩余进程的工作集都可以装入内存，这样可以减少频繁的缺页中断可能产生的抖动现象。

缺页率

如果一个进程的引用序列是p1、p2、…、pn-1、pn，它在执行过程中，产生缺页中断的次数为m，那么，该进程执行这个引用序列的缺页率定义为：

$$f=\frac{m}{n}*100\%$$

对于一个进程，分配的内存块数越多，它在运行过程产生的缺页率越小。

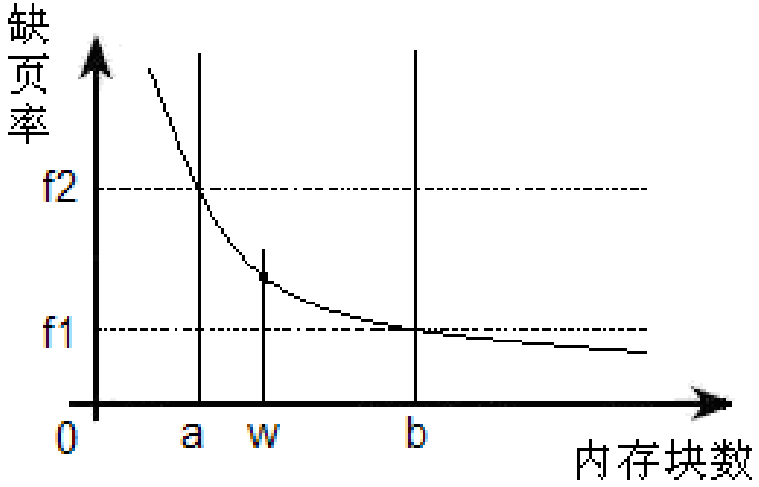

图5-32 缺页率与内存块的关系

对于一个进程，分配的内存块数越多，它在运行过程产生的缺页率越小。但是，对于FIFO置换算法，存在个别进程，分配给内存块数增加，缺页率反而也增加，这种的反常现象称为**Belady现象**。

一个进程运行过程依次访问的页号(也称进程的引用序列)是：0、2、3、1、4、1、2、3、5、2、3、1、4、5、0、3、6、9、8、3、6、7、3、6、9、8、7。 分别是2、3、4、5和6时，经验算，运行产生的缺页中断次数分别是26、20、22、11和11，缺页率分别是96%、74%、81%、41%和41%

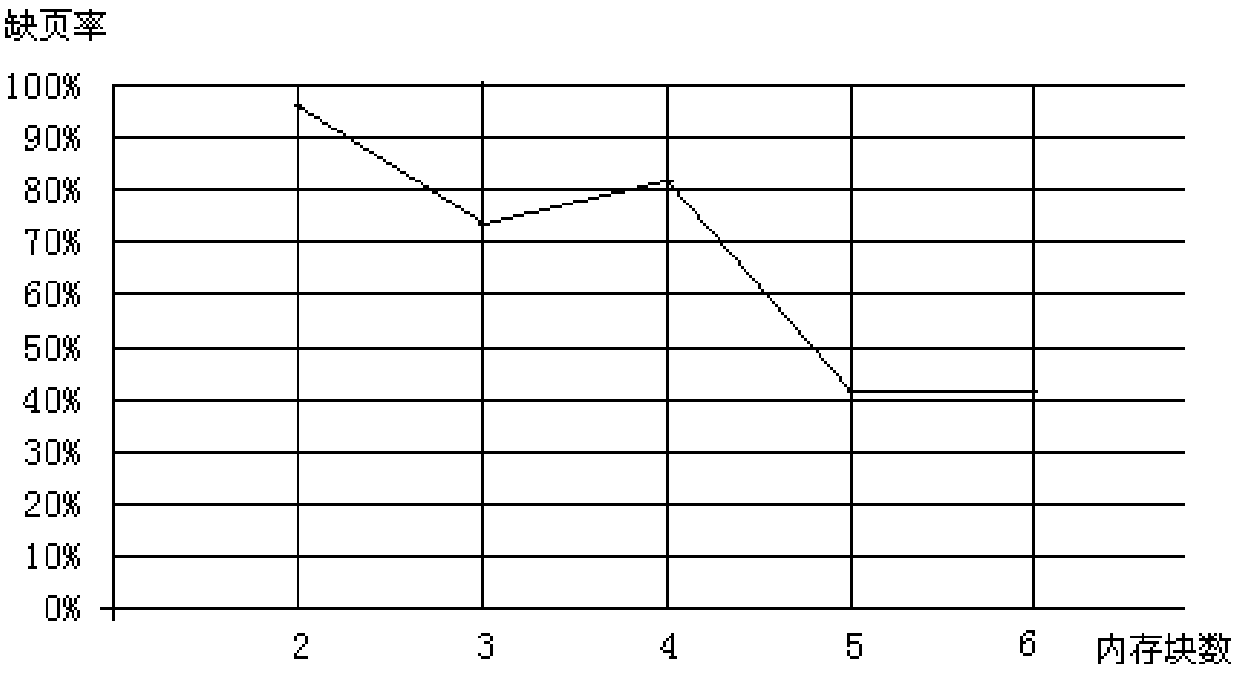

图5-33 Belady现象例子

- 分页存储管理的主要特点
  - 非连续的存储分配，提高了存储空间的利用率
  - 实现虚拟存储器
  - 页表占用额外的存储开销
  - 分页破坏了程序的完整性
  - 请求分页存在抖动现象，降低CPU的利用率

反向页表

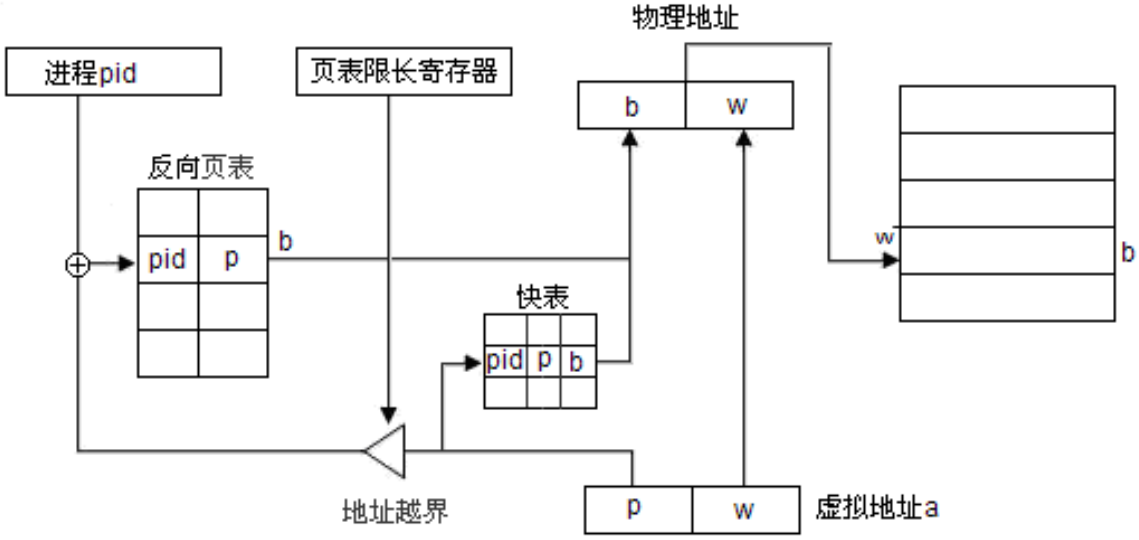

图5-34 反向页表的重定位

二级页表

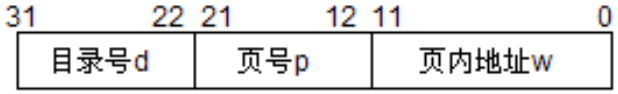

图5-35 32位CPU的虚拟地址结构

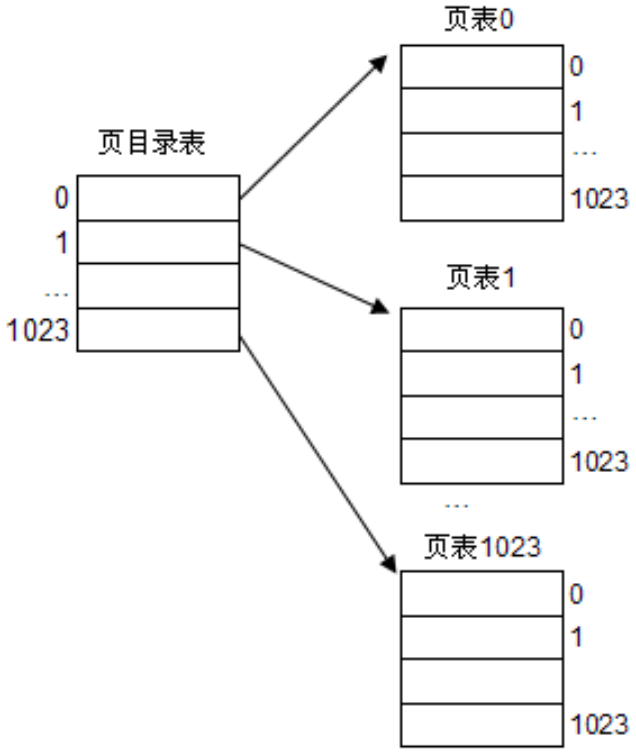

图5-36 一个进程的页目录表及其页表的关系

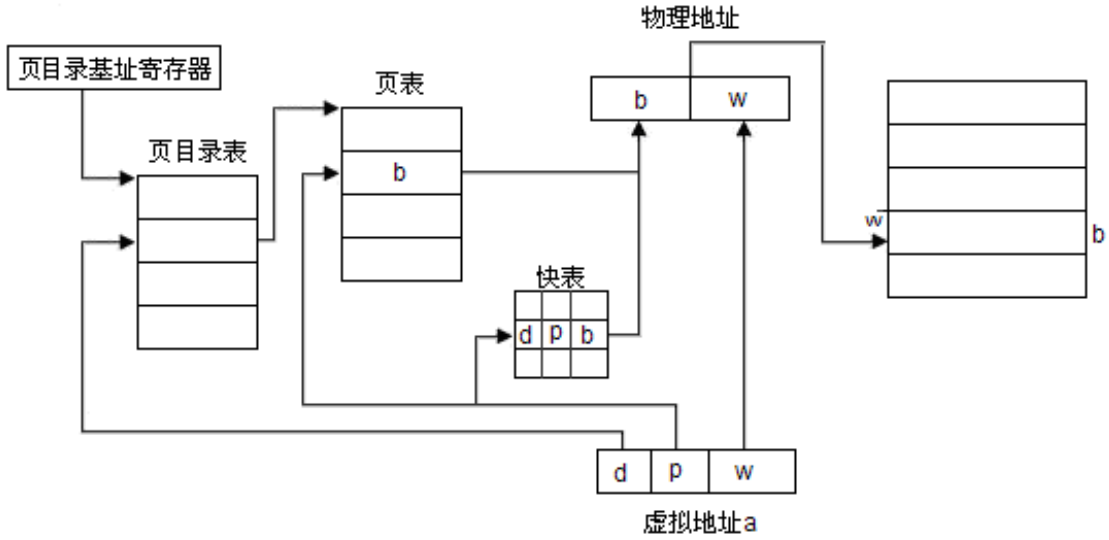

图5-37 二级页表的重定位

页长度与系统开销的关系

假设页长度为p，进程的大小为s，页表中每个表项的长度为e，那么，一个进程的内存的开销与页长度的关系为

$$f(p)=es/p+p/2$$

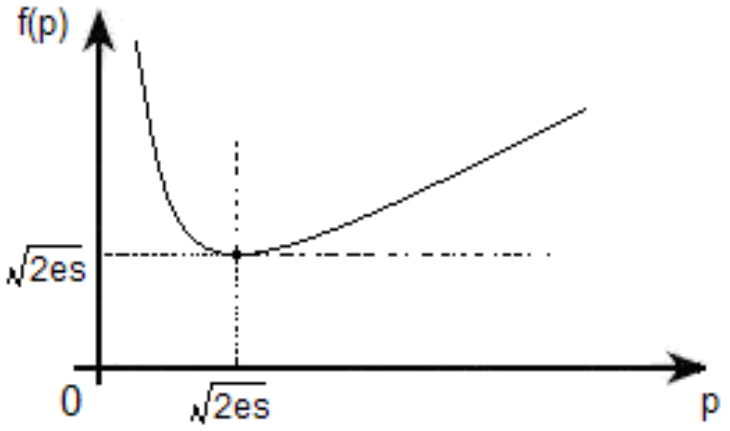

图5-38 内存开销与页长度的关系

六、分段存储管理

1.提出

2.基本思想

- ✓ 程序“分段”
- ✓ 内存动态分区
- ✓ 非连续存储分配
- ✓ 内、外存统一管理实现虚拟存储器

3.实现关键

- ➢ 数据结构设计

  段表的结构由段号、段长度、中断位P、分区起始地址、外存地址、存取控制信息、访问位A和修改位M等组成
- ➢ 存储空间的分配和回收
- ➢ 重定位和存储保护
- ➢ 段的共享

4.主要特点

5.分段与分页的区别

七、段页式存储管理

1.基本思想

3.实现关键

- 数据结构设计

  段表和段页表

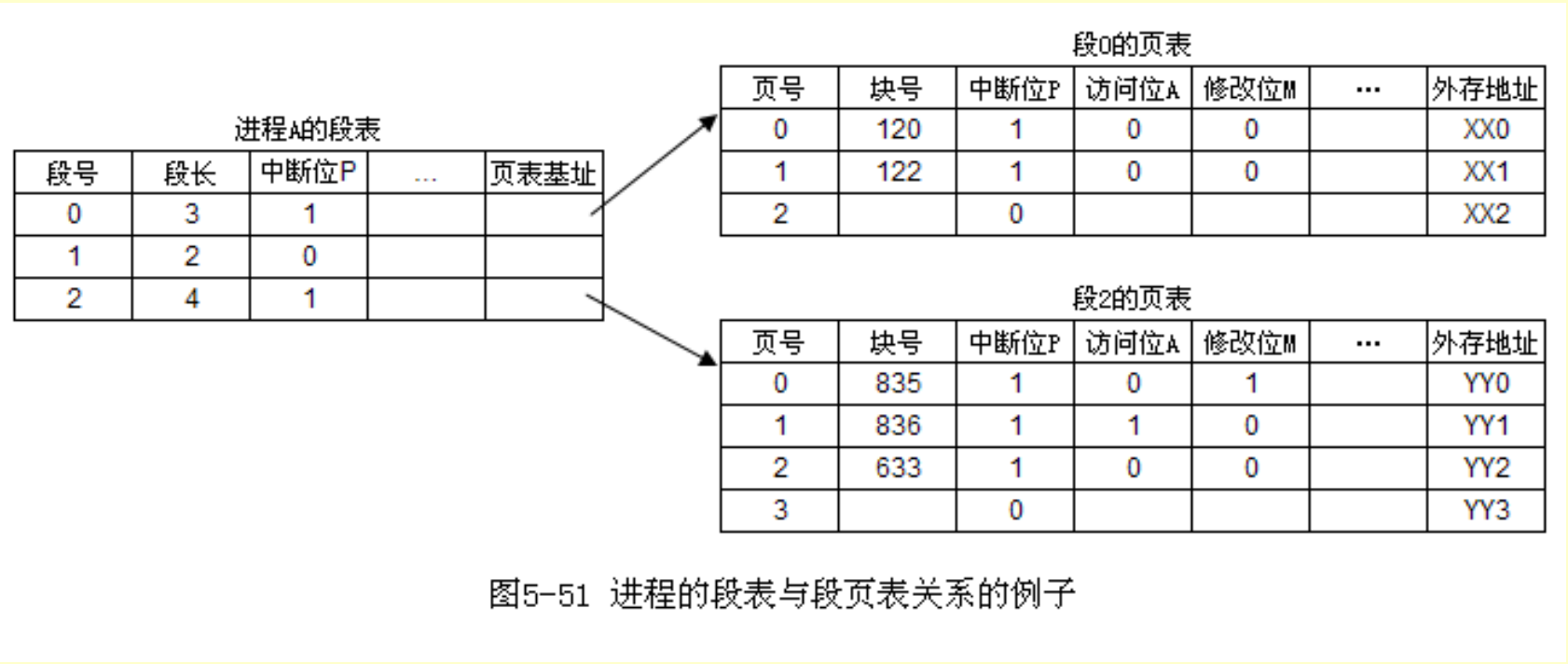

图5-51 进程的段表与段页表关系的例子

- 重定位

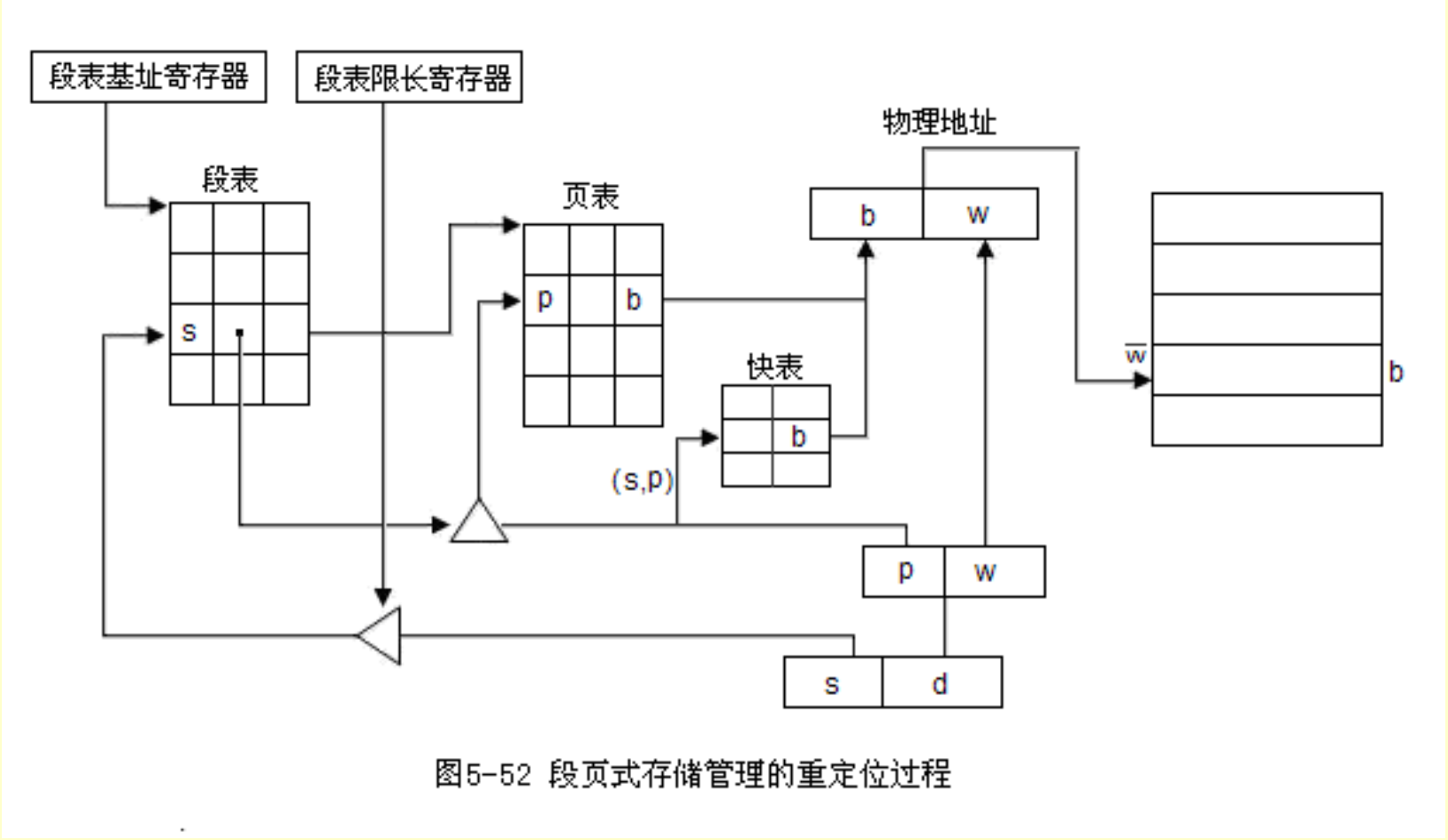

图5-52 段页式存储管理的重定位过程

- 主要特点