

# 第4章 存储器管理

# 存储器管理的任务

- 记录哪些内存正在使用，哪些内存是空闲的，在进程需要时为其分配存储器，在进程使用完后释放存储器，而在主存太小无法装入所有的进程时，管理主存和磁盘之间的交换。
- 存储器是现代计算机系统的操作中心，为CPU提供执行的指令和数据。

# 存储器层次结构

1. 少量快速昂贵易失的高速缓冲存储器；
2. 几十GB的中速价格适中的易失的主存(RAM)；
3. 几百GB至几十TB慢速的便宜的非易失的磁盘存储器。

协调使用这些存储器是OS的工作之一。

# 本章内容

- 4. 1 概述
- 4. 2 单用户单道程序的存储器分配
- 4. 3 多用户多道程序的存储器分配——分区分配
- 4. 4 覆盖与交换技术
- 4. 5 页式存储器管理
- 4. 6 段式存储器管理
- 4. 7 虚拟存储器管理

各种方案的实现原理、采用的数据结构、分配和回收算法。

## 4.1 概述

- 20世纪80年代，很多大学在**4MB** VAX机上运行拥有数十个用户的分时系统。
- 单用户版Windows 2000系统微软都建议用户至少有**64MB**的内存。
- 帕金森定律：(Parkinson's law) 存储器有多大，程序就会有多大。

# 存储器管理的功能

- (1) 存储器分配：解决多道程序或多进程共享主存的问题。
- (2) 地址转换或重定位：研究各种地址变换方法及相应的地址变换机构。
- (3) 存储器保护：防止故障程序破坏OS和其它信息
- (4) 存储器扩充：采用多级存储技术实现虚拟存储器及所用的各种管理算法。
- (5) 存储器共享：并发执行的进程如何共享主存中的程序和数据。

# 1. 地址空间

- 符号名字空间：源程序中的各种符号名的集合所限定的空间。如源程序中的数据和子程序通常是由符号名进行访问的。
- 用户程序和非常驻的系统程序随机且动态地进入系统，编译和连接程序无法预先确定其内存存储位置，所以只能用逻辑地址编址。编译时，程序中各个地址总是以“0”作为起始地址顺序编码。
- 逻辑地址空间：经编译连接后的目标代码所限定的空间。用地址码替换符号地址。
- 相对地址，逻辑地址，虚地址。

## 2. 存储空间

- 物理存储器中全部物理存储单元的集合所限定的空间。
- 存储空间是由字或字节组成的一个大的阵列，每一个字或字节都有它自己的编号地址。
- 绝对地址，物理地址，实地址。
- 一个程序只有从地址空间装入到存储空间后才能运行。

### 3. 地址重定位

- 把程序地址空间的逻辑地址转换为存储空间的物理地址。地址映射，或地址变换。
- 地址重定位的原因：
  - ◆ 地址空间的逻辑地址往往与分配到的存储空间的物理地址不一致；
  - ◆ 处理机执行用户程序时，所要访问的指令和数据地址必须是实际的物理地址。

## (1) 程序的链接

- 静态链接：在构造可执行文件时，就将各目标模块与库例程链接在一起。其映像形成的是个一维的地址域。
- 动态链接：
  - ◆ 装入时动态链接：边装入边链接。
  - ◆ 运行时动态链接：程序运行时才进行链接。便于程序模块的共享，减少系统空间的开销。

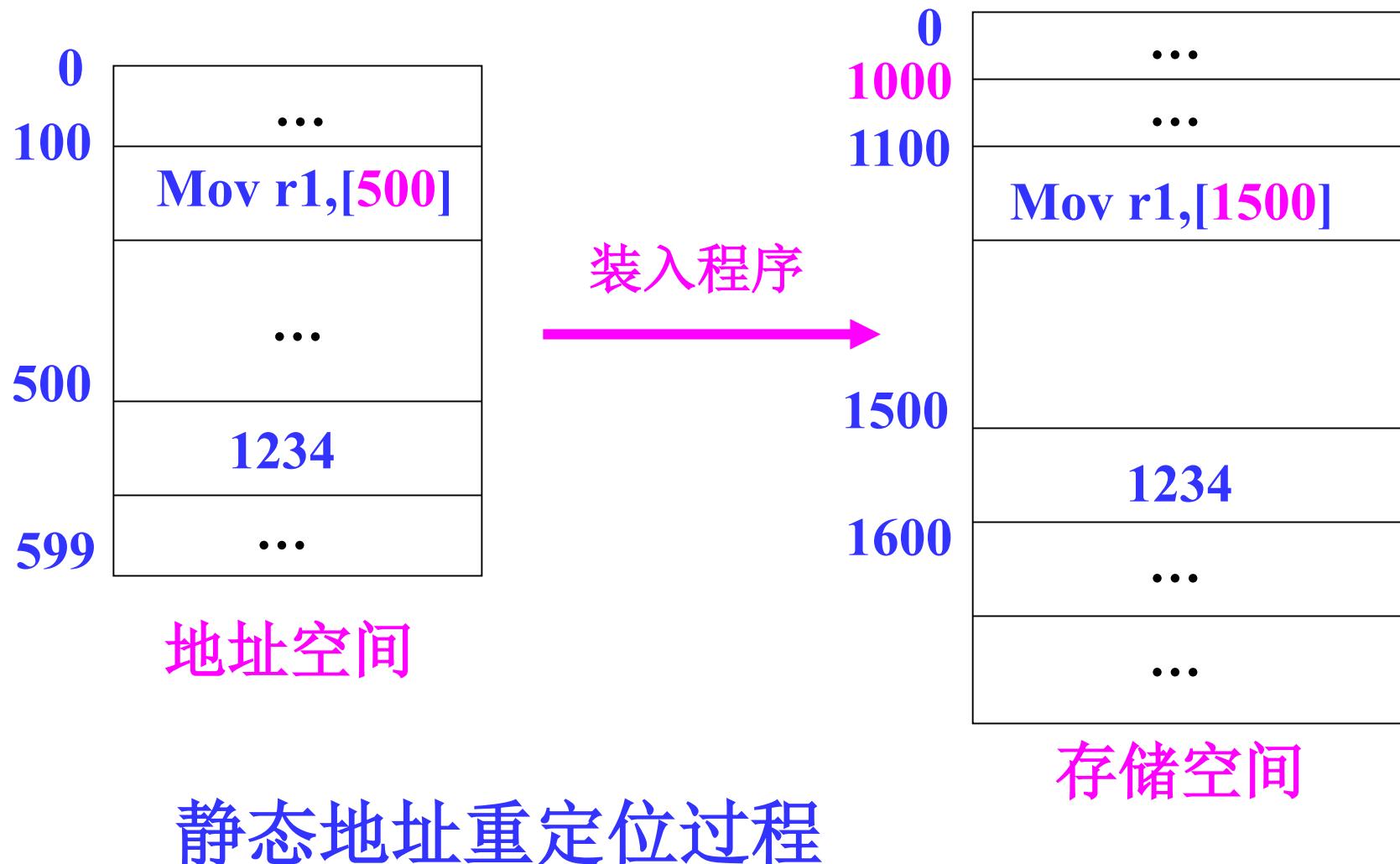
## (2) 静态重定位

在进程执行前，由装入程序把用户程序中的指令和数据的逻辑地址全部转换成存储空间的物理地址。

特点：

- 1) 无硬件变换机构；
- 2) 为每个程序分配一个连续的存储区；
- 3) 在程序执行期间不能移动，主存利用率低；
- 4) 难以做到程序和数据的共享；
- 5) 用于单道批处理系统。

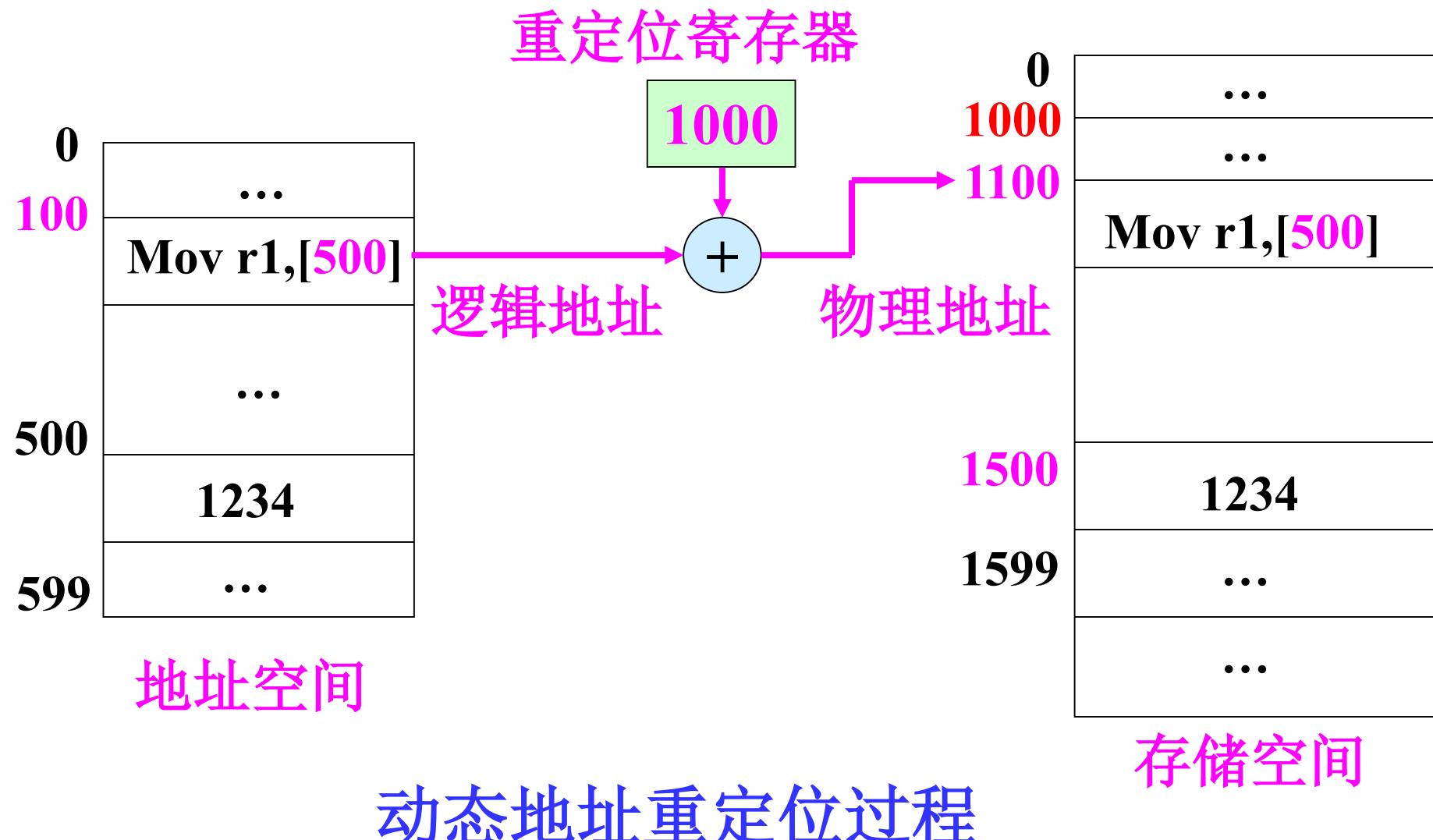
# 把程序装入起始地址为1000的内存区



### (3) 动态重定位

- 装入程序把程序和数据原样装入到已分配的存储区中。程序运行时，把该存储区的起始地址送入重定位寄存器。需硬件地址转换机构。
- 多道批处理系统、分时系统
- 优点：
  - ◆ 主存利用充分。可移动用户程序。移动后，只需修改重定位寄存器。
  - ◆ 程序不必占有连续的存储空间。
  - ◆ 便于多用户共享存储器中的同一程序和数据。

# 把程序装入起始地址为1000的内存区



# 4. 存储器保护

- 防止地址越界：进程运行时产生的所有存储器访问地址都要进行检查，确保只访问为该进程分配的存储区域。
- 正确地进行存取：对所访问的存储空间的操作方式（读、写、执行）进行检查，以防止由于误操作，使其数据的完整性受到破坏。

# 5. 存储器共享

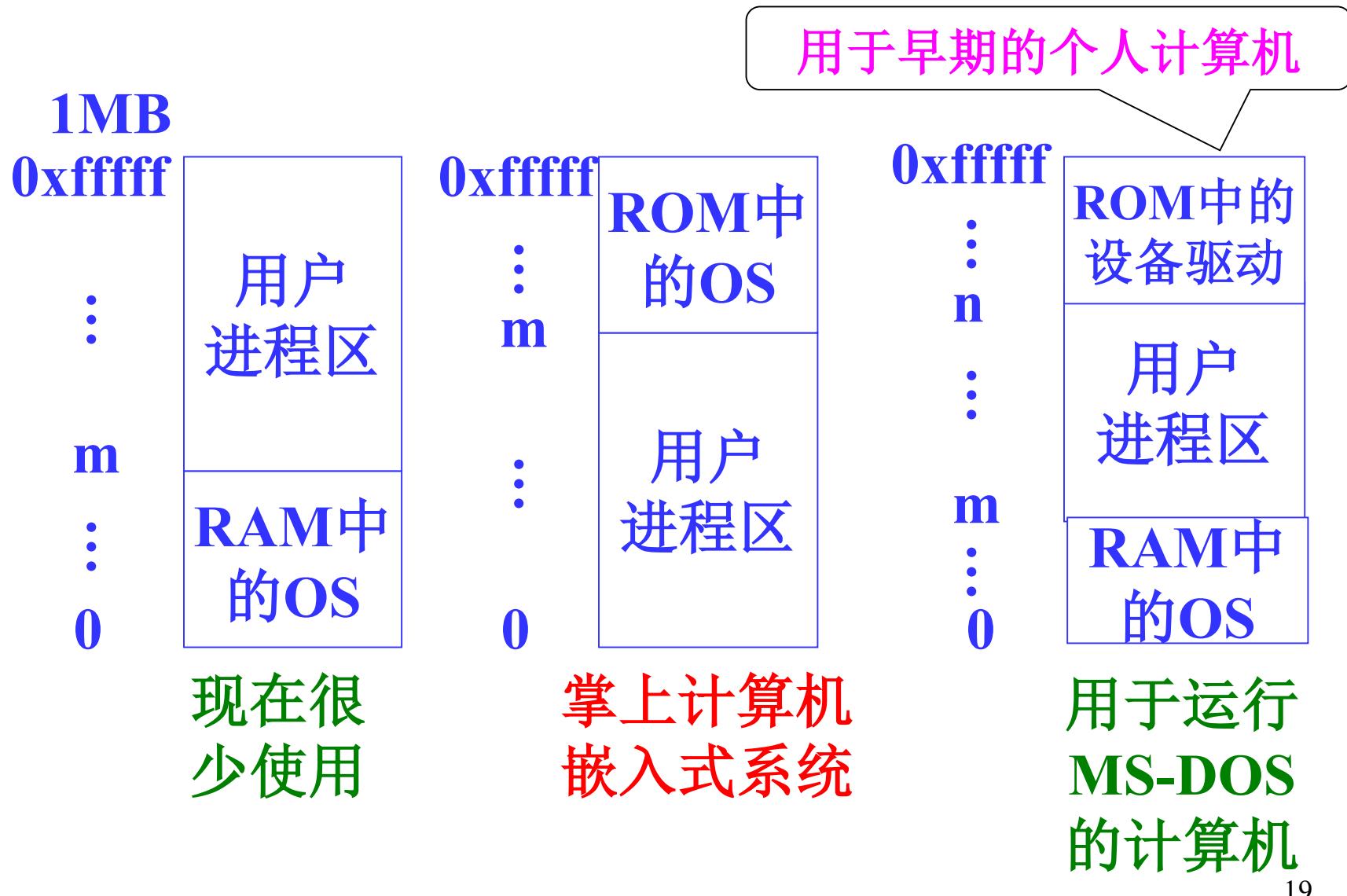
- 为了提高存储器的利用率，允许多个进程共享同一个主存区。
- 既可以是数据区，也可以是程序区。
- 被共享的程序叫可重入程序，其代码无论执行多少遍，都保持不变。具有这种性质的程序又叫纯代码。

- 通常存储器划分为两部分：一部分是操作系统占用区，另一部分是用户进程占用区（或用户区）。
- 存储器管理是指对用户区的管理。

## 4.2 单用户单道程序的 存储器分配

- 是一种最简单的存储管理方式。
- 用于单道批处理、单用户单任务系统。
- 单一连续区分配，主存只有一个用户作业。
- 作业一旦进入内存，就要等到它执行结束后才能释放内存。
- 存储保护容易实现，易判断地址是否越界。

# 单一连续区的存储空间的组织结构

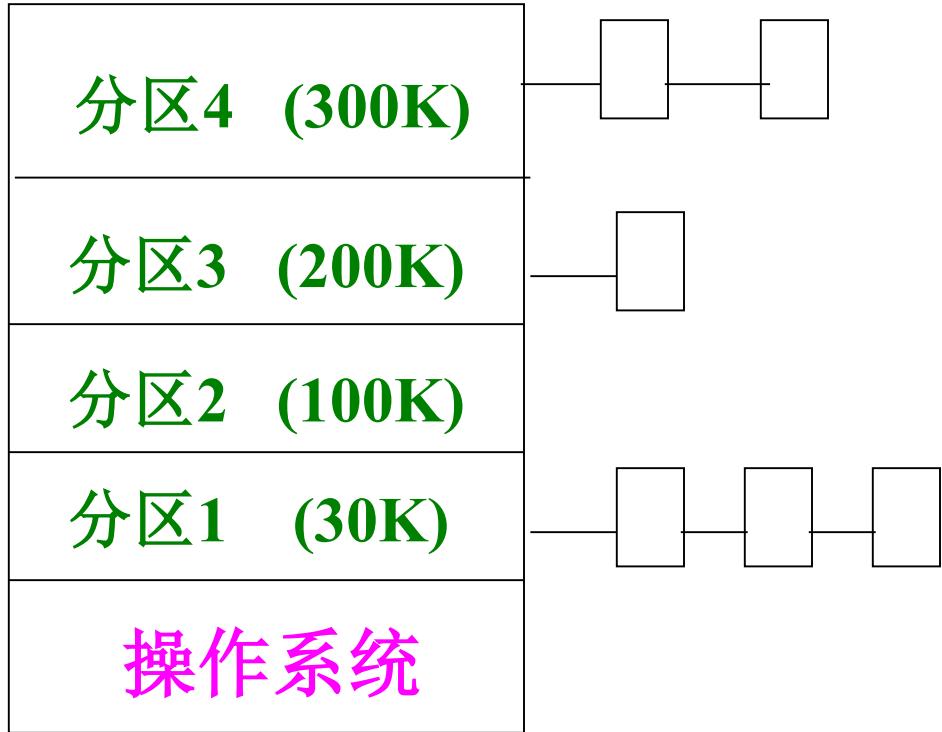


## 4.3 多用户多道程序的存储分配 —— 分区分配

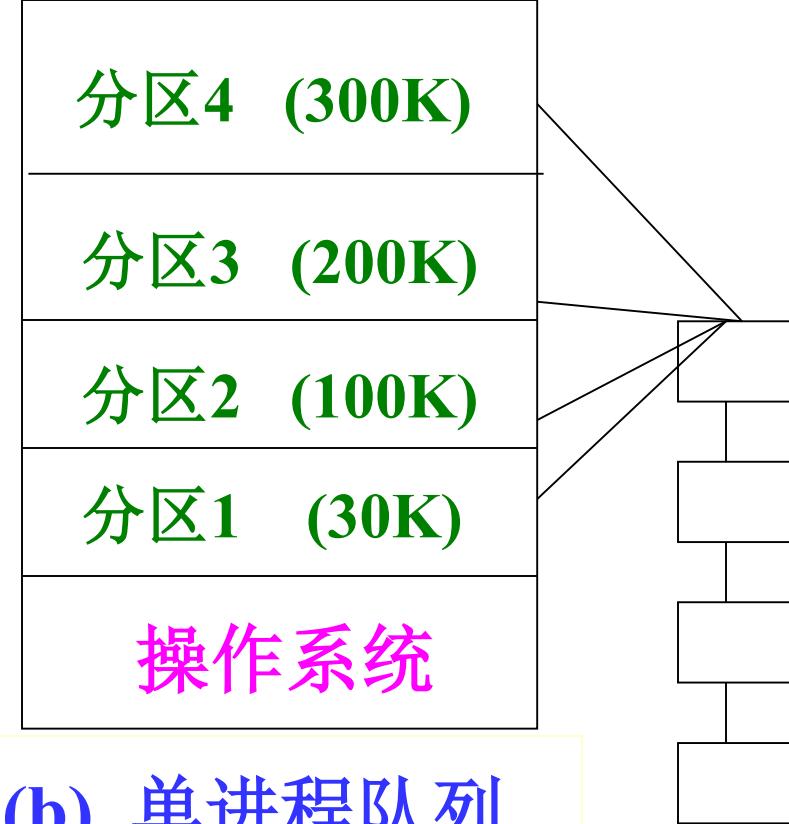
- 把主存划分成若干个连续的区域，每个进程占用一个。
- 系统中必须配置相应的数据结构，用来记录内存的使用情况，为分配提供信息。
- 固定式分区；可变式分区

## 4.3.1 固定式分区

- 适合多道程序的**最简单**的存储器管理方式
- 把主存预先划分成**几个大小不等**的分区
- 进程到达时，选择一个适合进程要求的**最小空闲区分给进程**；无适合的空闲分区时，**让其等待**。
- 曾经在**IBM OS/360**大型机中运行了许多年



(a) 多进程队列



(b) 单进程队列

按大小排入各队列等  
待, 易造成某个队列  
拥挤

得到一个空闲分区,  
就装入一个作业  
可搜索作业队列

# 分区说明表

用以描述各分区的分配情况。

[例]

分区起始地址	分区大小	占用标志
50KB	30KB	J1
80KB	100KB	0
180KB	200KB	J2
380KB	300KB	0

## 内存管理过程：

- ✓ 用户程序要装入内存，内存分配程序检索分区说明表，从表中找出一个空闲分区分给该程序，并置占用标志。
- ✓ 若找不到空闲分区，就不分配内存。
- ✓ 当进程执行完毕释放内存时，内存管理程序将对应分区的状态置为未分配。

**缺点：**主存利用不充分。因为作业的大小不可能刚好等于某个分区的大小，绝大多数已分配的分区中，都有一部分存储空间被浪费掉了。

**优点：**简单。

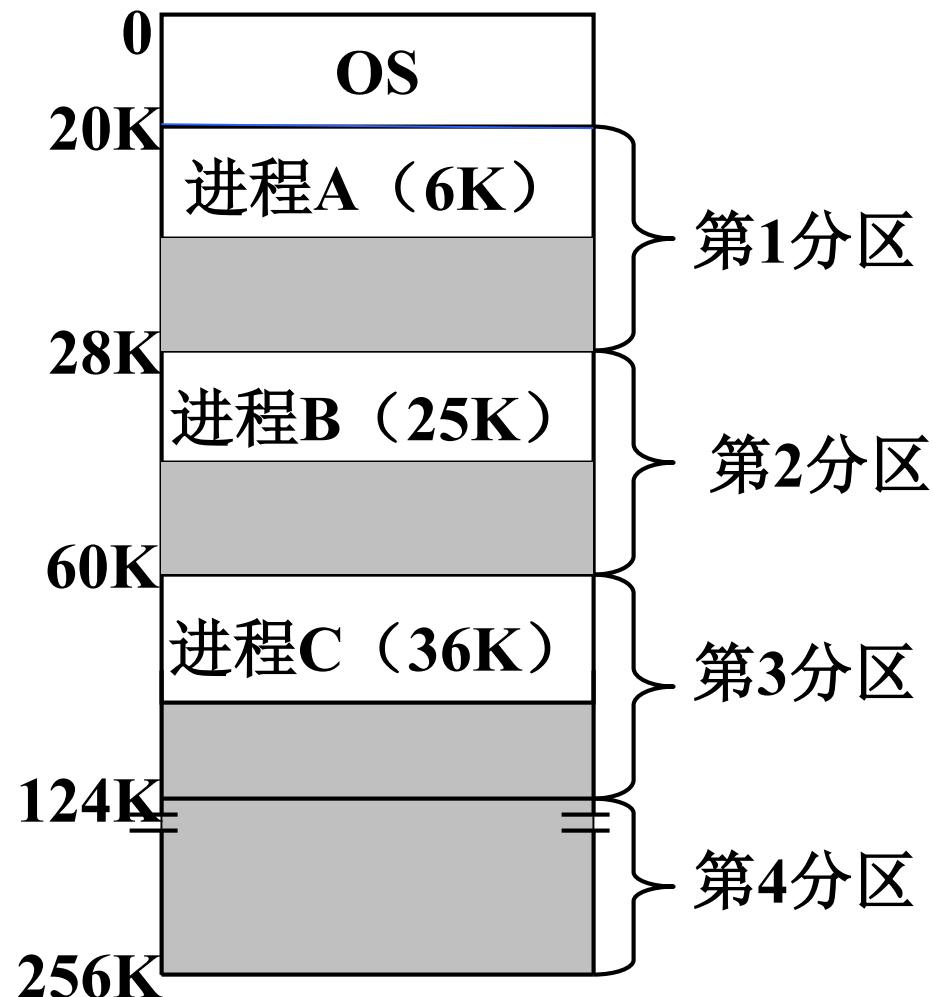
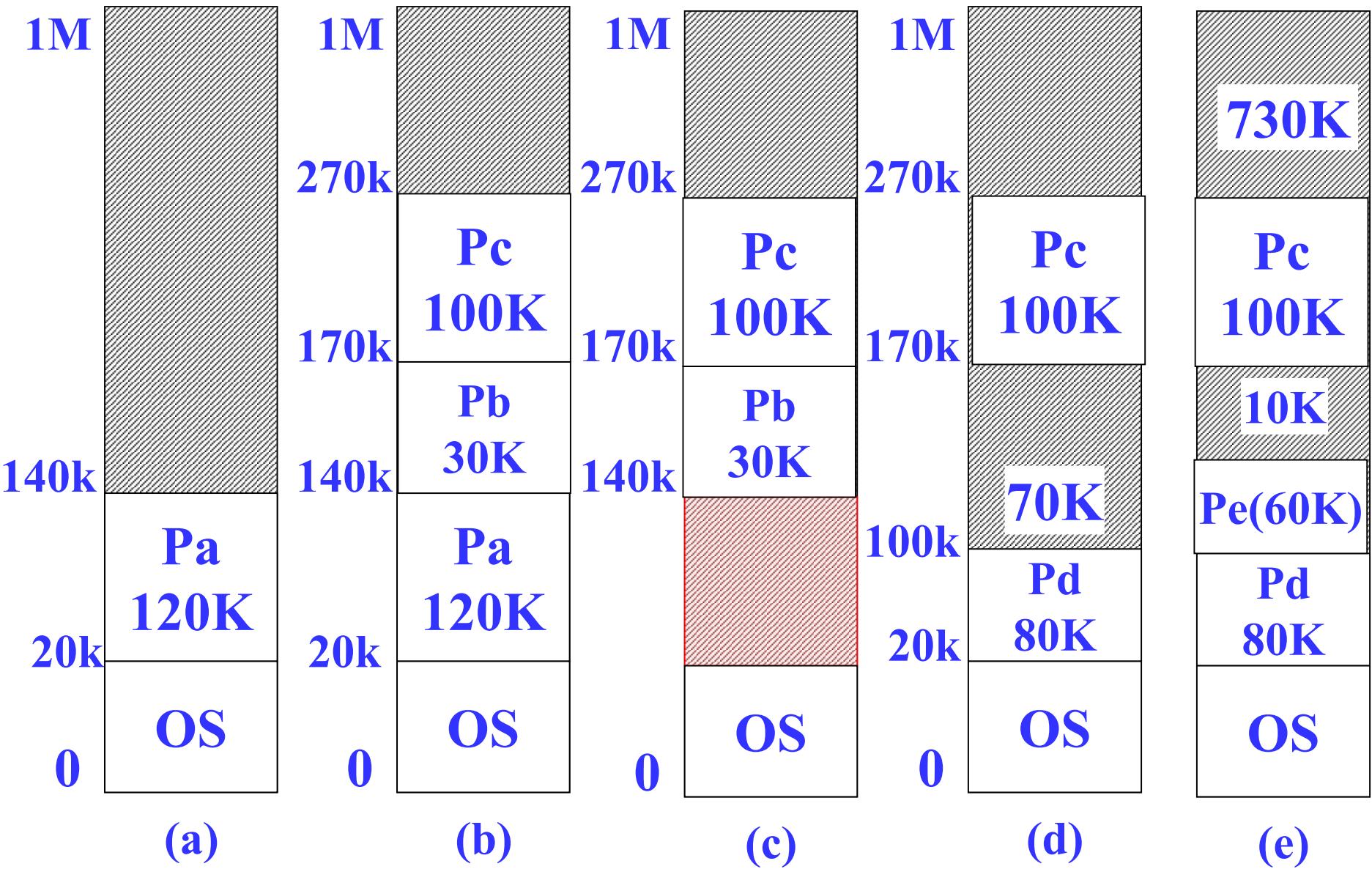


图4.5内存状态

## 4.3.2 可变式分区

- 根据作业的大小动态地划分分区，使分区的大小正好等于作业大小。
  - ◆ 各分区的大小是不定的；
  - ◆ 内存中分区的数目也是不定的。

[例] 进程A在主存；进程B和C从盘上装入；  
进程A运行完；进程D装入，进程B运行完；  
进程E装入主存。



# 管理分区使用的数据结构

- 分区说明表、空闲区链表。
- 分区说明表：已分配区表，未分配区表。
- 分配算法：
  - ◆ 首次适应法(first fit),
  - ◆ 最佳适应法(best fit),
  - ◆ 最坏适应法(worst fit)

# (1) 分区说明表

- 分配主存：
  - ◆ 从未分配区表中找一个空闲区。将其一分为二，一部分分给作业，另一部分留在表中。
  - ◆ 在已分配区表中进行记录。
- 回收主存：
  - ◆ 将回收的分区登记在未分配区表中。
  - ◆ 将该作业占用的已分配区表项置为空。

始址	长度	占用标志
20KB	80KB	Pd
100KB	60KB	Pe
170KB	100KB	Pc
		空
		⋮

(a) 已分配区表

始址	长度	占用标志
160KB	10KB	有效
270KB	730KB	有效
		空
		空
		⋮

(b) 未分配区表

**优点：**比较直观、简单。

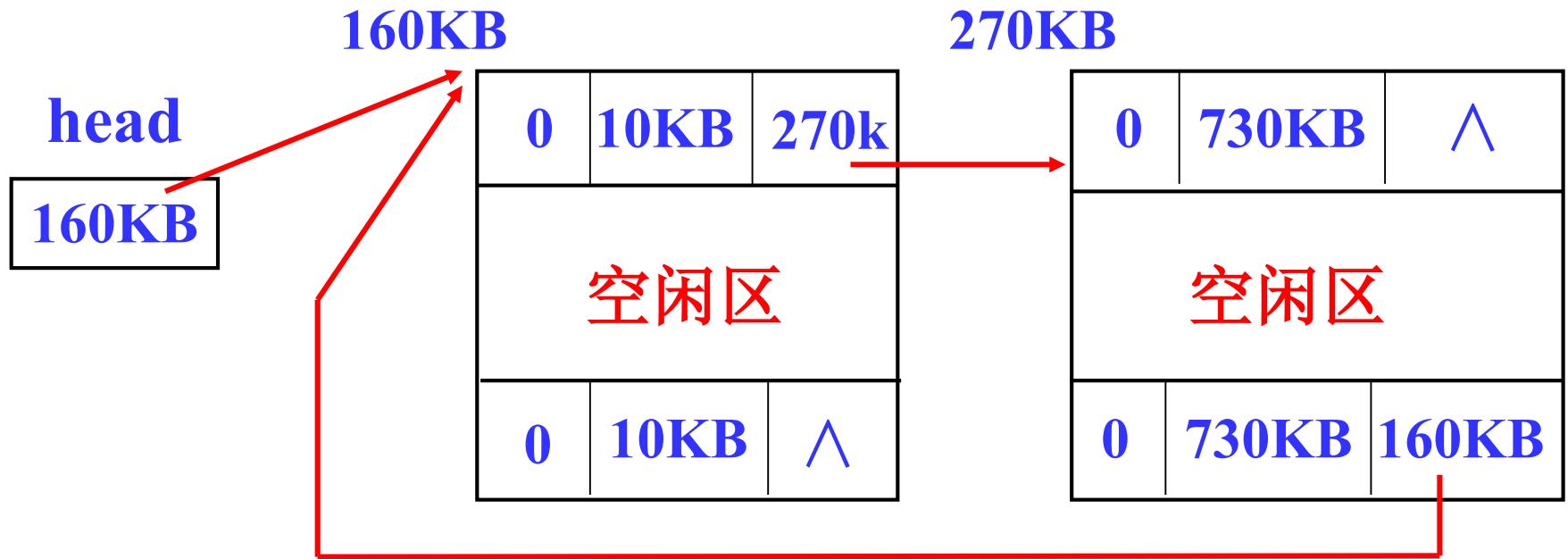
**缺点：**由于主存分区个数不定，所以表格的长度不好确定。要么表格溢出，要么表格浪费。

## (2) 空闲区链

- 是记录内存**空闲区**的一种较好方法。
- 已分配区，在进程的**PCB**中有记录。
- 在每个分区的**首字**和**尾字**中，有该区信息的记录。

状态位	分区大小 (N+2)	向前指针	字
大小为N 的已分配区或空闲区			
状态位	分区大小 (N+2)	向后指针	字

- ✓ 状态位：“0” 表示空闲，“1” 已分配。
- ✓ 分区大小(以字节或字为单位)。
- ✓ 指针：通常采用双向链表。

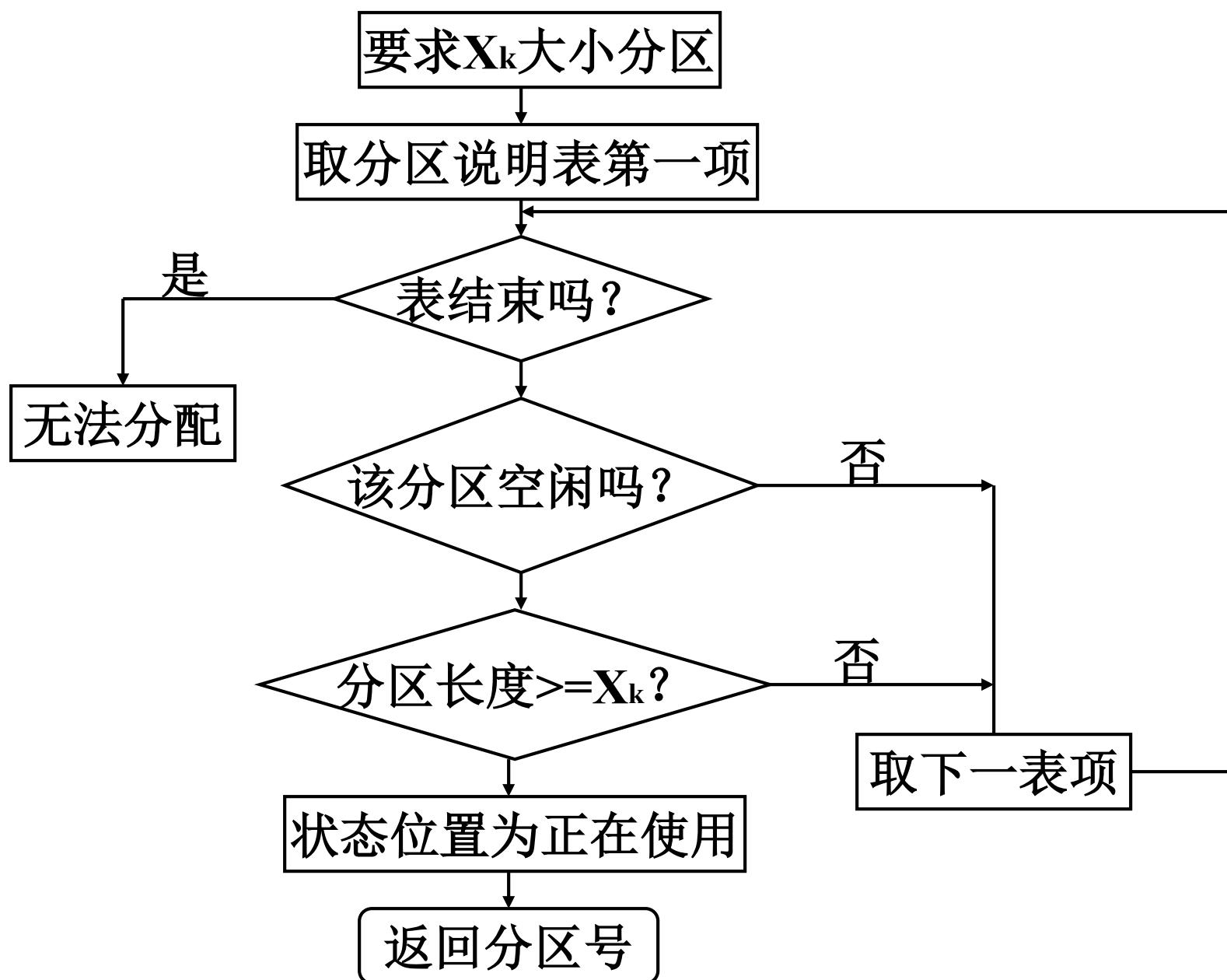


系统设置一个链表头指针head

# 分区的分配与回收

- 固定分区时的分配与回收

该方法较为简单，当用户程序要装入执行时，通过请求表提出内存分配要求和所要求的内存空间大小。存储管理程序根据请求表查询分区说明表，从中找出一个满足要求的空闲分区，并将其分配给申请者。



固定分区分配算法

## ● 可变式分区的分配与回收

它主要起解决以下三个问题：

- a.对于请求表中要求的内存长度，从可用表或自由链中寻找合适的空闲区分配给程序；
- b.分配空闲区后，更新可用表或自由链；
- c.进程或作业释放内存资源时，和相邻的空闲区进行链接合并，更新可用表或自由链。

该方法从可用表或自由链中寻找空闲区的常用算法有三种：首次适应法(最先适应法)、最佳适应算法、最坏适应算法。

动态分区的回收更加简单。当进程执行完毕，不再需要内存资源时，管理程序将对应的分区状态置为未使用即可。

# 可变式分区--分配算法

## (1) 首次适应(first fit)法:

- ◆ 要求空闲区表或空闲区链中的空闲区按地址从小到大排列。
- ◆ 分配内存时，从起始地址最小的空闲区开始扫描，直到找到一个能满足其大小要求的空闲区为止。分一块给请求者，余下部分仍留在其中。

## 特点：

- ◆ 优先利用低地址部分的空闲分区，保留了高地址部分的大空闲区。

其好处是，当需要一个较大的分区时，有较大希望找到足够大的空闲区以满足要求。

- ◆ 低地址端可能留下许多很少的空闲分区，而每次查找是从低地址部分开始，会增加查找开销。

## (2) 最佳适应(best fit)法:

- ◆ 存储分配程序要扫描所有空闲区，直到找到能满足进程需求且为最小的空闲区为止。
- ◆ 缺点：
  - 因为要查找所有的分区，所以比首次适应算法效率低。
  - 可能把主存划分得更小，出现很多无用的碎片。
- ◆ 改进：从小到大对空闲区排序。

## 特点：

- ◆ 因为分配分区要查找整个链表，所以比首次适应算法效率低。若存在与作业大小一致的空间分区，则它必然选中；若不存在与作业大小一致的空闲分区，则只划分比作业稍大的空闲分区，从而保留了大的空闲区。
- ◆ 因为它可能把主存划分得更小，成为无用的碎片，所以它比首次适应要浪费更多的空贮空间。

### (3) 最坏适应(worst fit)法:

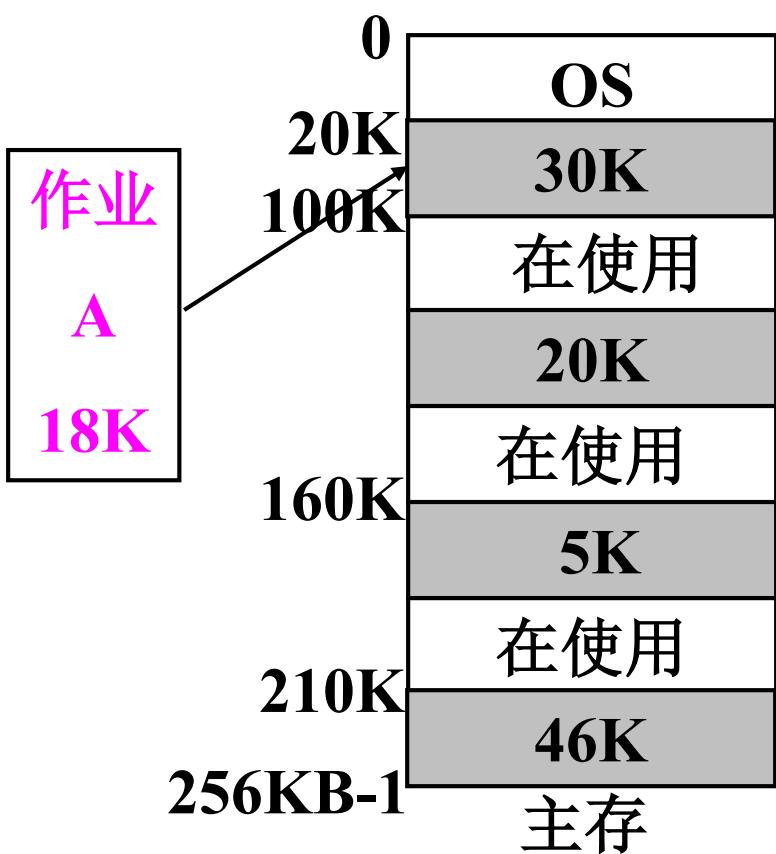
- ◆要扫描所有的空闲区，直到找到满足进程要求且为最大的空闲区为止。一分为二，一部分分给进程，另一部分仍留在链表中。
- ◆目的：使剩下的空闲区可用。
- ◆缺点：要扫描所有的空闲区；大空闲区的不断分割，可能满足不了大进程的要求。
- ◆改进：从大到小对空闲区排序，以提高查找速度。

## 优点

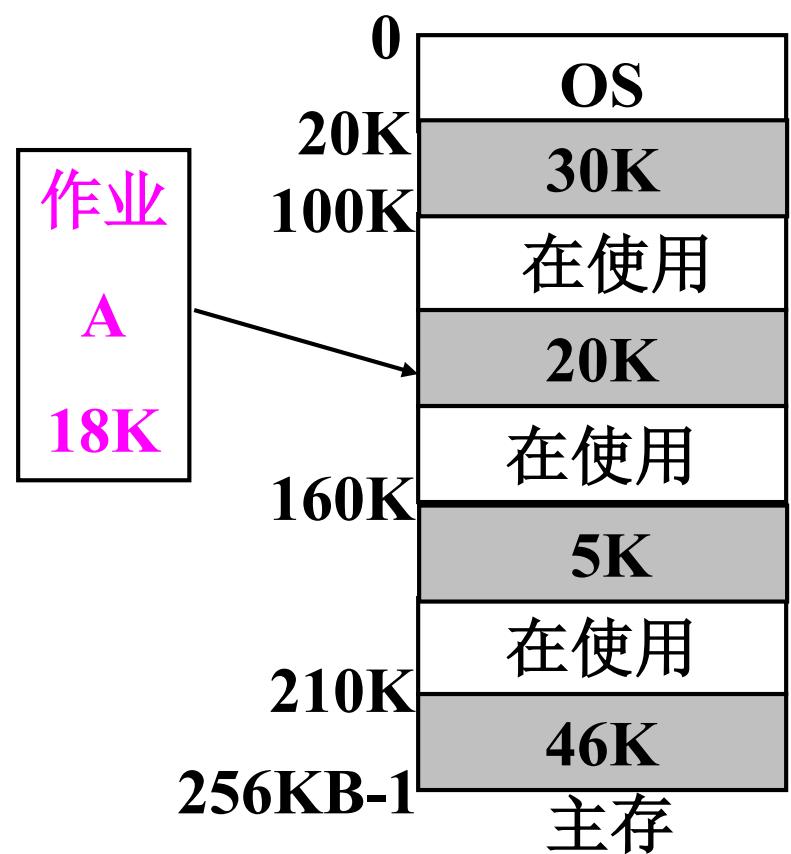
总是挑选满足作业要求的最大的分区分配给作业，这样使分给作业后剩下的空闲区也比较大，于是也能装下其他作业。

## 缺点

由于最大的空闲区总是首先被分配而进行划分，当有大作业到来时，其存储空间的申请往往得不到满足。

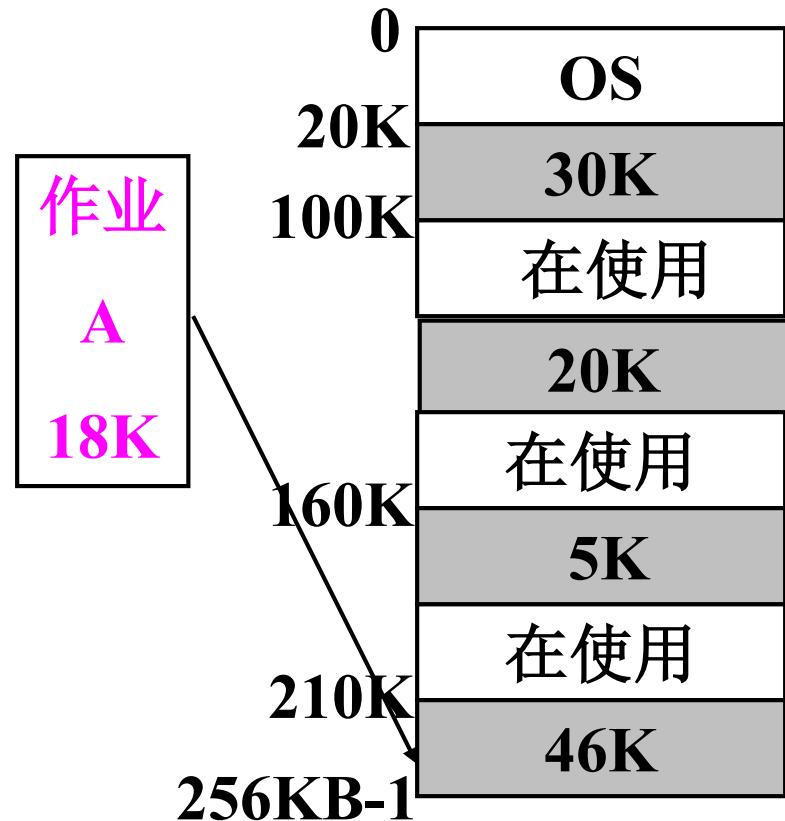


(a) 最先适应算法



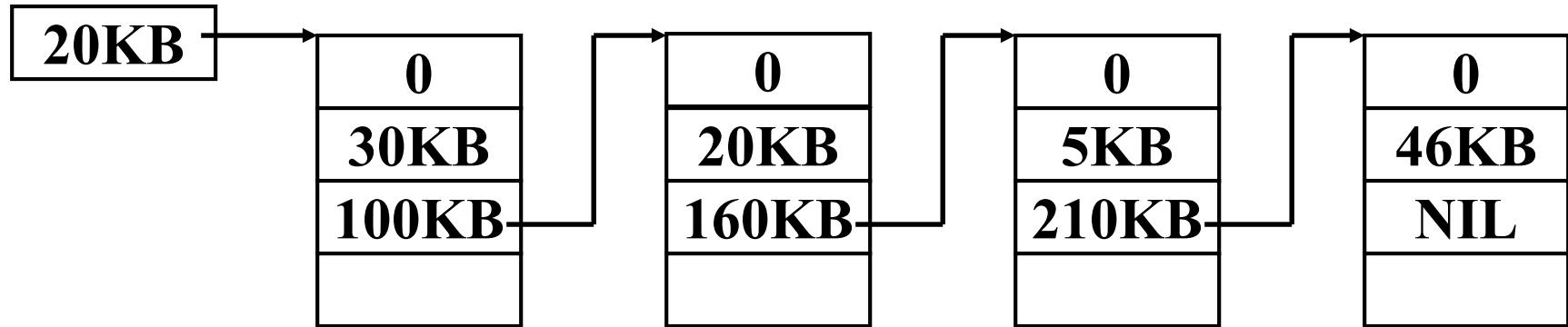
(b) 最佳适应算法

图 三种算法的说明

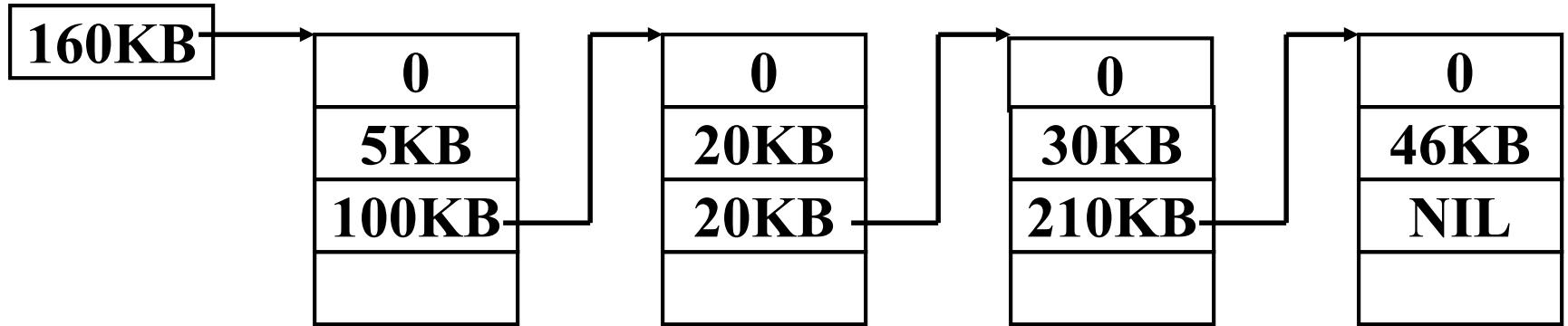


(c) 最坏适应算法

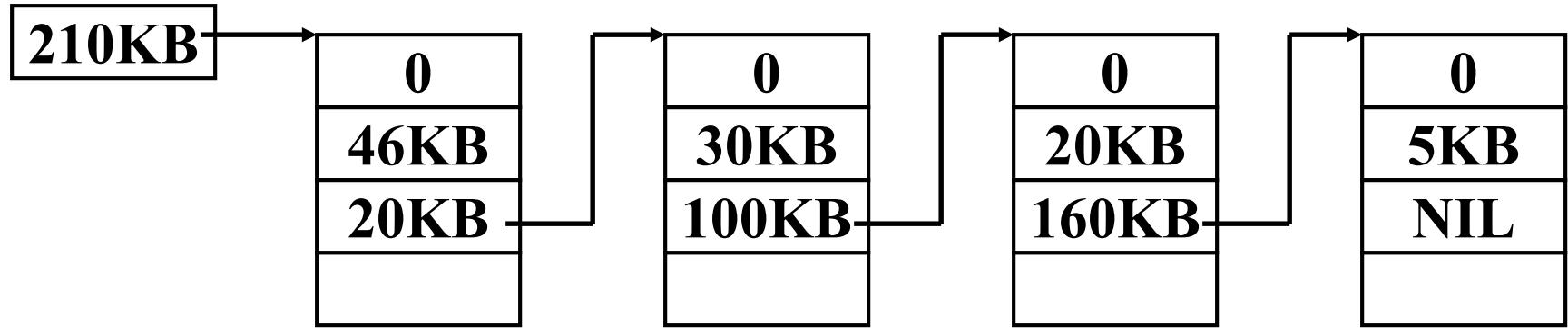
# 三种算法下的自由主存队列



(a) 最先适应算法中的自由主存队列



(b) 最佳适应算法中的自由主存队列



(c) 最坏适应算法中的自由主存队列

(3) 这三种算法到底哪一种好，不能一概而论，而应针对具体的作业序列来分析。

对于某一作业序列来说，若某种算法能将该作业序列中的所有作业安置完毕，那我们就说该算法对这一作业序列是合适的。

对于某一算法而言，如它不能立即满足某一要求（即在某个被分配的分区回收之前无法进行分配），而其他算法却可以满足此要求，则这一算法对该作业序列是不合适的。

例1.表4.1给出了某系统中的空闲分区表，系统采用动态分区存储管理策略。现有以下作业序列：96K、20K、200K。若用最先适应算法和最佳适应算法来处理这些作业序列，试问哪一种算法可以满足该作业序列的请求，为什么？

表4.1 空闲分区表

分区号	大小	起始地址
1	32K	100K
2	10K	150K
3	5K	200K
4	218K	220K
5	96K	530K

解：若采用最佳适应算法：

在申请96K存储区时，选中的是5号分区，5号分区的大小与申请空间大小一致，应从空闲分区表中删去该表项；

接着申请20K时，选中1号分区，分配后剩下12K；

最后申请200K，选中4号分区，分配后剩下18K。显然，采用最佳适应算法进行内存分配，可以满足该作业序列的需求。为作业序列分配内存后，空闲分区表如下4.2所示。

表4.2 空闲分区表

分区号	大小	起始地址
1	12K	120K
2	10K	150K
3	5K	200K
4	18K	420K

若采用最先适应算法：

在申请96K存储区时，选中的是4号分区，进行分配后4号分区还剩下122K；

接着申请20K时，选中1号分区，分配后剩下12K；

最后申请200K，现有的五个分区都无法满足要求，该作业等待。

显然，采用最先适应算法进行内存分配，无法满足该作业序列的需求。这时的空闲分区表如表4.3所示。

表4.3 空闲分区表

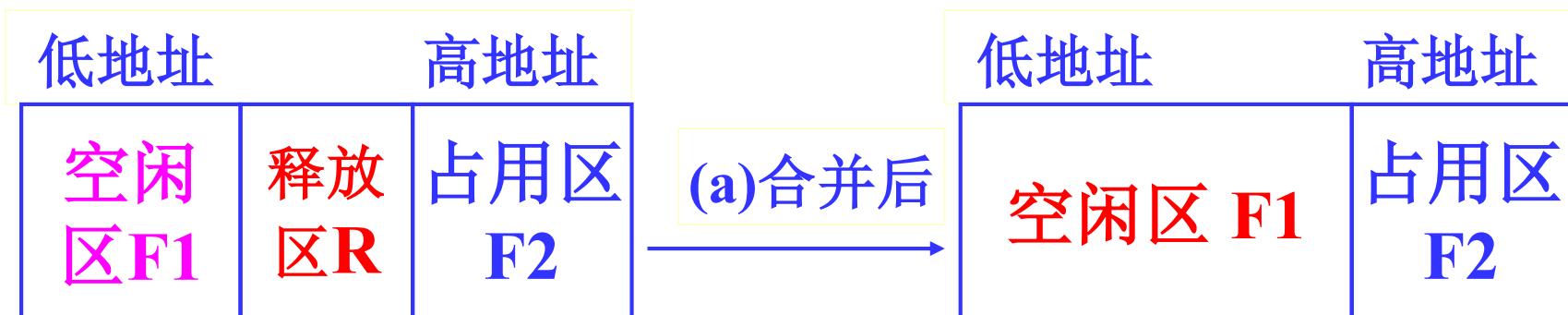
分区号	大小	起始地址
1	12K	120K
2	10K	150K
3	5K	200K
4	122K	316K
5	96K	530K

# 回收一个释放区

若释放区与空闲区相邻接，要进行合并。

[例] 以首次适应法说明系统回收进程释放区存在的四种可能情况。

(a) R与F1相邻接。合并后，F1的大小为两者之和。



(b) R与F2相邻接。F2的首地址，大小为两者之和



(c) 先将R与F2合并，再将F 2与F1合并。



(d) 若释放区R不邻接空闲区，则将其插入空闲区链的适当位置即可。

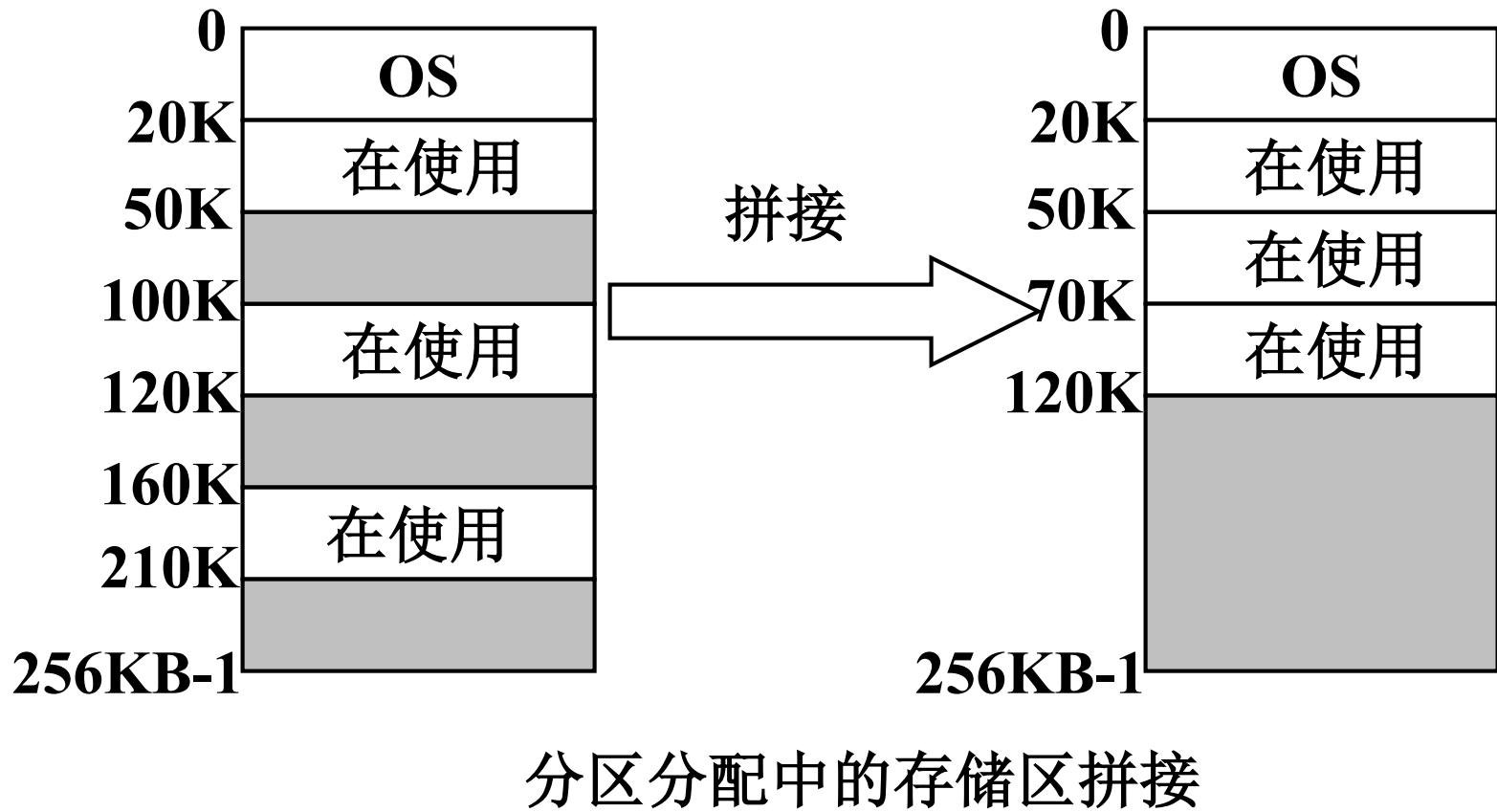
# 碎片问题与拼接技术

(1) 分区存储管理技术能满足多道程序设计的需要，但也存在着一个非常严重的问题，即**碎片**（也称**零头**）是指内存中无法被利用的小的空闲区。

在分区存储管理方式下，系统运行一段时间后，内存中的碎片会占据相当数量的空间，有时甚至会出现这样的情况，当一个作业申请一定数量的内存时，虽然此时空闲区的总和大于作业所申请的内存容量，但却没有单个空闲区大到足以装下这个作业。

(2) 解决这个问题的办法之一是采用拼接技术

**拼接：**是指移动存储器中所有已分配区到主存的一端，使本来分散的空闲区连成一个大的空闲区。如图所示。



### (3) 拼接的时机问题

这个问题有两种解决方案。

**第一种方案：在某个分区回收时，立即进行拼接，  
这样在主存中总是只有一个连续的空闲区。**

**由于拼接很费时间，拼接频率过高会使系统开销加大。**

**第二种方案：当找不到足够大的空闲区，且空闲区  
的存储总量可以满足作业要求时，进行拼接。**

**这样，拼接的频率比上一方案要小得多，但空闲区的管  
理稍为复杂一些。**

### 4.3.3 地址重定位和存储器保护

- 固定分区
  - ◆ 静态重定位，进程运行时使用主存物理地址。
  - ◆ 设置上、下界寄存器来实现存储器保护
- 可变式分区
  - ◆ 动态重定位，进程运行时CPU给出的是程序的逻辑地址。
  - ◆ 基址+限长寄存器。世界上第一台超级计算机CDC 6600使用该方案。

- **上、下界寄存器**: 分别存放进程在主存区的最高地址和起始地址。进程运行时，下界寄存器 $\leq$ CPU访存地址 $\leq$ 上界寄存器。否则产生地址越界错误，停止运行。
- **基址和限长寄存器**: 进程运行时，将其分区的首地址装入基址（或重定位）寄存器，将其程序的大小装入限长寄存器。每个逻辑地址应小于限长寄存器内容。

## 4.3.4 分区管理的优缺点

- 主要优点：
  - ◆ 实现了多道程序共享主存。
  - ◆ 系统设计相对简单，不需要更多的软硬件开销。
  - ◆ 实现存储器保护的手段也比较简单。
- 缺点：
  - ◆ 主存利用不够充分。系统中总有一部分存储空间得不到利用，这部分被浪费的空间叫碎片。
  - ◆ 没有实现主存的扩充。进程的地址空间受实际存储空间的限制。

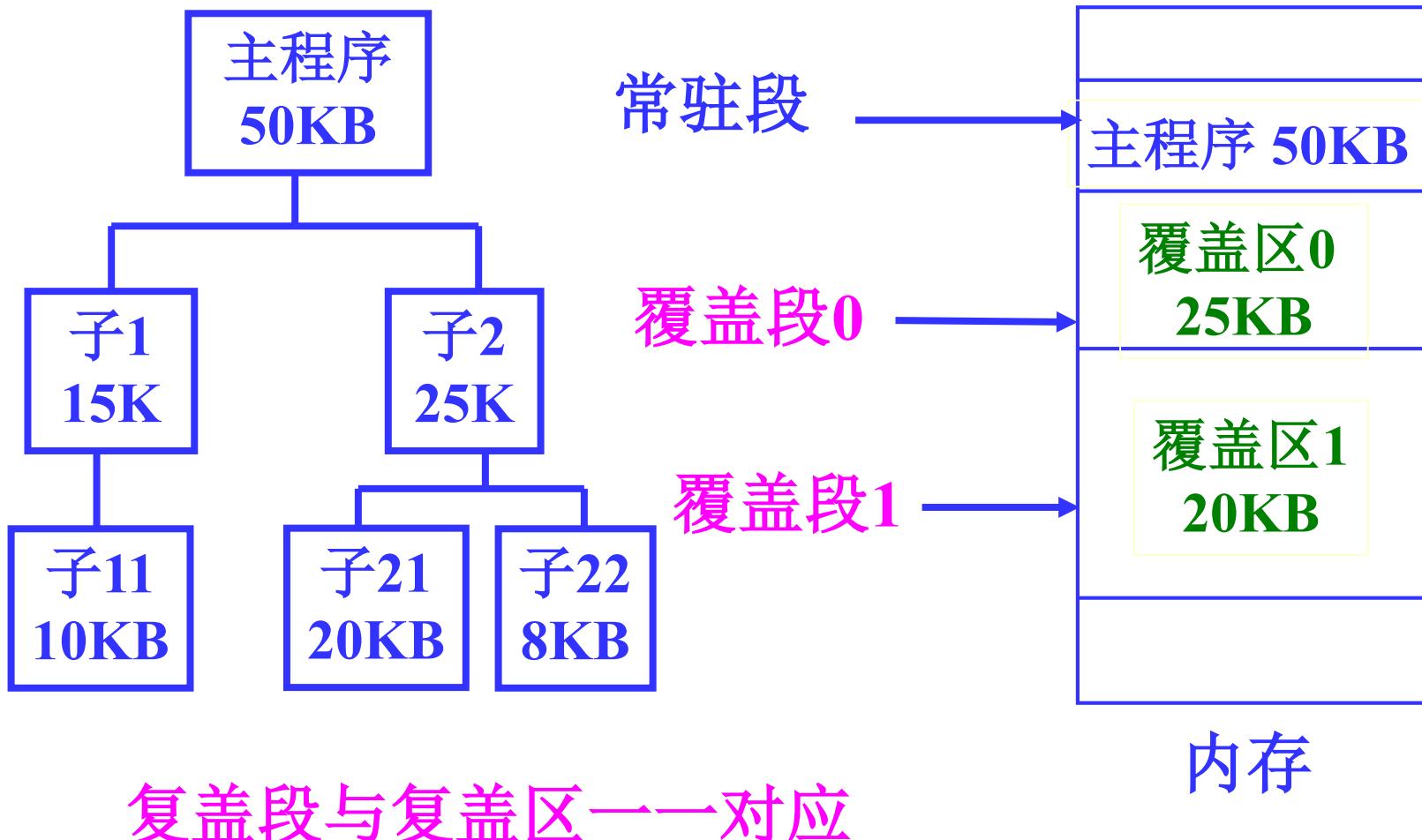
## 4.4 覆盖与交换技术

- 覆盖与交换技术是解决大进程和小主存矛盾的两种存储器管理技术
- 其实质是对主存进行逻辑扩充
- 覆盖技术主要用在早期的OS中
- 交换技术则在现代OS中仍具有较强的生命力

# 1. 覆盖(overlay)

- **覆盖**: 是指同一主存区可以被不同的程序段重复使用
- 通常一个进程由若干个功能上相互独立的程序段组成，进程在一次运行时，也只用到其中的几段。让那些不会同时执行的程序段共用同一个主存区。
- **覆盖段**: 可以相互覆盖的程序段。
- **覆盖区**: 可共享的主存区。

## [例] 一个用户程序的覆盖结构。



- 主程序是一个独立的段，它调用子1和2，且子1与子2是互斥被调用的两个段，在子1执行过程中，它调用子11，子2执行过程中它又调用子21和子22，显然子21和子22也是互斥被调用的。
- 因此我们可以为作业J建立覆盖结构：主程序段是作业J的常驻主存段，其余部分组成复盖段。

- 子1和子2组成**覆盖段0**; 子11、子21和子22组成**覆盖段1**,
- 相应的**覆盖区大小**应为每个覆盖段中最大覆盖的大小。

- 系统必须提供相应的**覆盖管理控制程序**。  
用户提供**覆盖结构**。通常难以分析和建立  
程序的覆盖结构。
- 覆盖技术主要用于系统内部程序的主存管  
理上。**[例]** 操作系统分常驻主存部分和不常  
驻主存部分。
- **特点：**打破了必须将一个进程的全部信息  
装入主存后才能运行的限制。在逻辑上扩  
充了主存。**小主存可运行大进程。**

## 2. 交换(Swapping)

- 系统根据需要把主存中暂时不运行的进程中的部分或全部信息移到外存，而把外存中的进程移到主存，并使其投入运行。
- 目的：
  - ◆ 解决主存容量不够大的问题；
  - ◆ 保证分时用户的合理响应时间。

- 交换的时机：
  - ◆ 进程用完时间片或等待输入输出时；
  - ◆ 进程要求扩充其占用的存储区而得不到满足时。
- 具有交换功能的操作系统，通常把外存分为文件区和交换区。
- 文件区存放文件； 交换区存放从内存换出的进程。
- 对交换区的分配是采用连续分配方式，目的是提高进程换入、换出速度。

- 交换技术的**关键**: 设法减少每次交换的信息量。
  - ◆ 常将进程的**副本**保留在外存，每次换出时，仅换出那些**修改过**的信息即可。
- 交换主要是在**进程之间**进行，而覆盖则主要在同一个**进程内**进行。
- **主要特点**: 打破了一个程序一旦进入主存，便一直运行到结束的限制。

**实质：用辅存作缓冲，让用户程序在较小的存储空间中通过不断地换出作业而运行较大的作业。**

- 交换技术不要求程序员给出程序段之间的**覆盖结构**。

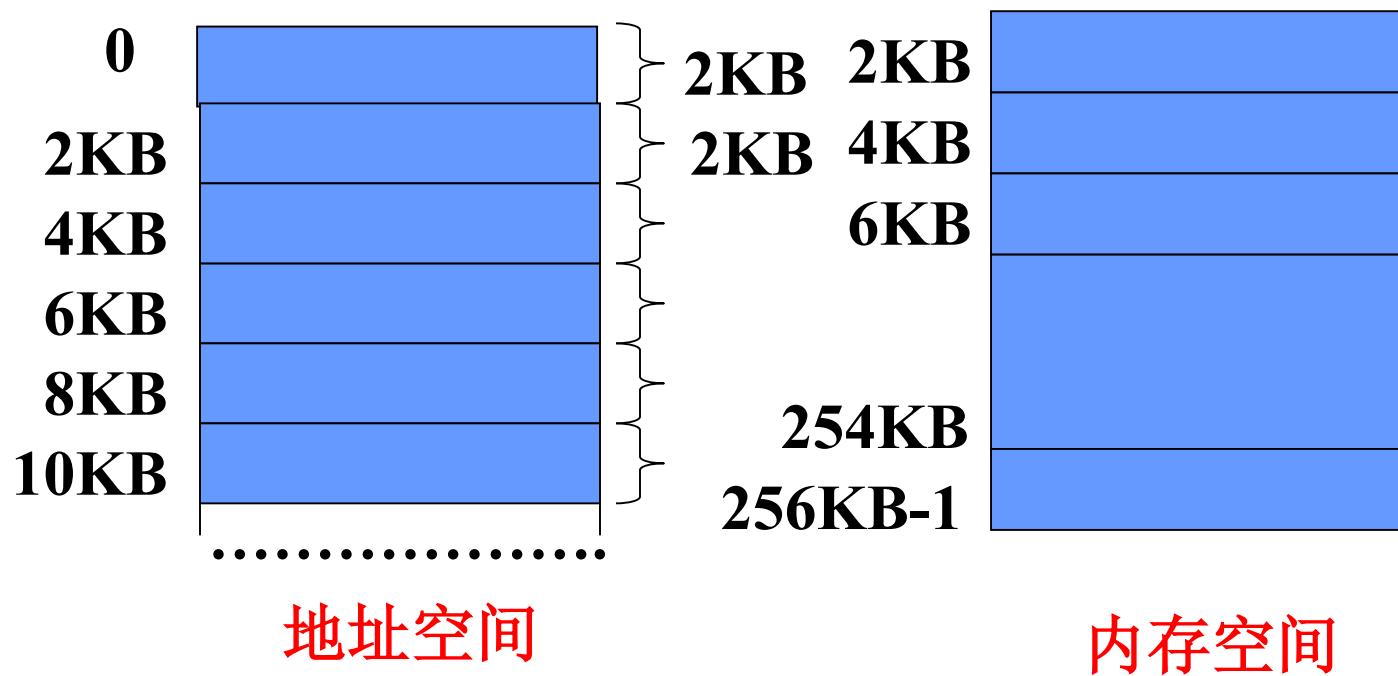
## 4.5 页式存储器管理

- 在分区存储管理中，每道程序要求占用主存的一个连续的存储区域，因而会产生许多碎片。
- 页式存储管理允许一个进程占用不连续的存储空间，从而克服了碎片。

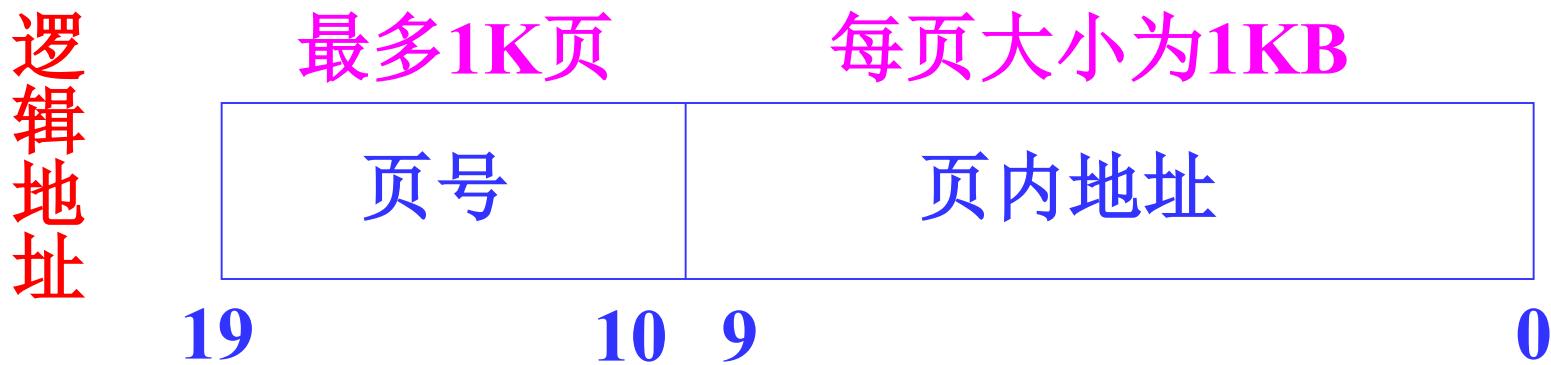
# 1. 页式管理的实现原理

- **页框（主存块）**：把主存分成大小相等的若干块。块的大小一般为1024或4096字节等2的整次幂。
- **页**：进程的地址空间被划分成与主存块大小相等的页。
- 可以将进程中的任意一页放到主存的任意一块中，实现了离散分配。

例：如果页的大小定义为2K，那么进程的地址空间和内存空间的示意图如下图所示。



- **页表**: 系统为每个进程建立一张页面映像表，记录逻辑页与主存块的映射关系。
- CPU产生的的地址:



例如，一个进程申请6150B的存储区，当页的大小为1024B时，它共有 $7(6150/1024)$ 个页，需7个内存块。

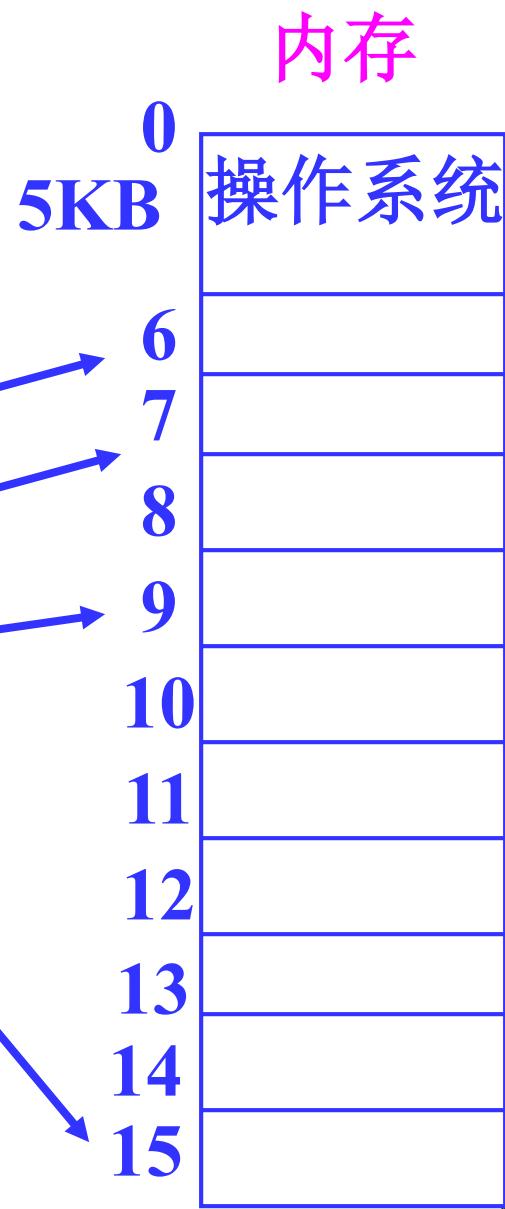
## [例] 页式管理的逻辑图

进程的  
地址空间

页号	地址空间
0	0-1023
1	1024-2047
2	2048-3071
3	3072-4095
:	:

页表

6
7
9
15
:



- 页表存放在主存。
- **控制寄存器**: 在页式管理中，系统为每个处理机设立一个控制寄存器，用以记录现运行进程的页表始址和页表长度。
- 硬件动态地址转换机构**MMU**负责将逻辑地址转换为物理地址

## ● 页式管理的特征

页式管理时，用户进程在内存空间内，除了在每个页面内地址连续之外，每个页面之间不再连续。这样进行页式管理，第一是实现了内存中碎片的减少，因为任一碎片都会小于一个页面。第二是实现了由连续存储到非连续存储这个飞跃，为在内存中局部的、动态地存储那些反复执行或即将执行的程序和数据段打下了基础。

页表方式实质上是动态重定位技术的一种延伸。

## ● 页式管理需解决的几个问题

一个进程在运行时，只是将它的部分页面装入内存，在进程的执行过程中，系统将根据需要从外存调入所需的页面。为此，需解决以下几个问题：

(1) 地址映射：由于进程的页面被装入到内存的若干块中，由于这些块可能是不连续的，因此，为了保证程序的正确执行，必须进行地址映射。

(2) 调入策略：当装入部分页面时，需判断当前访问的页是否在内存。当确认所访问的页面不在内存时，系统必须从外存调入请求的页面。

(3) 淘汰策略：当需要调入一页，而内存已被其他页面所调用，需要确定哪些页面应从内存中淘汰。

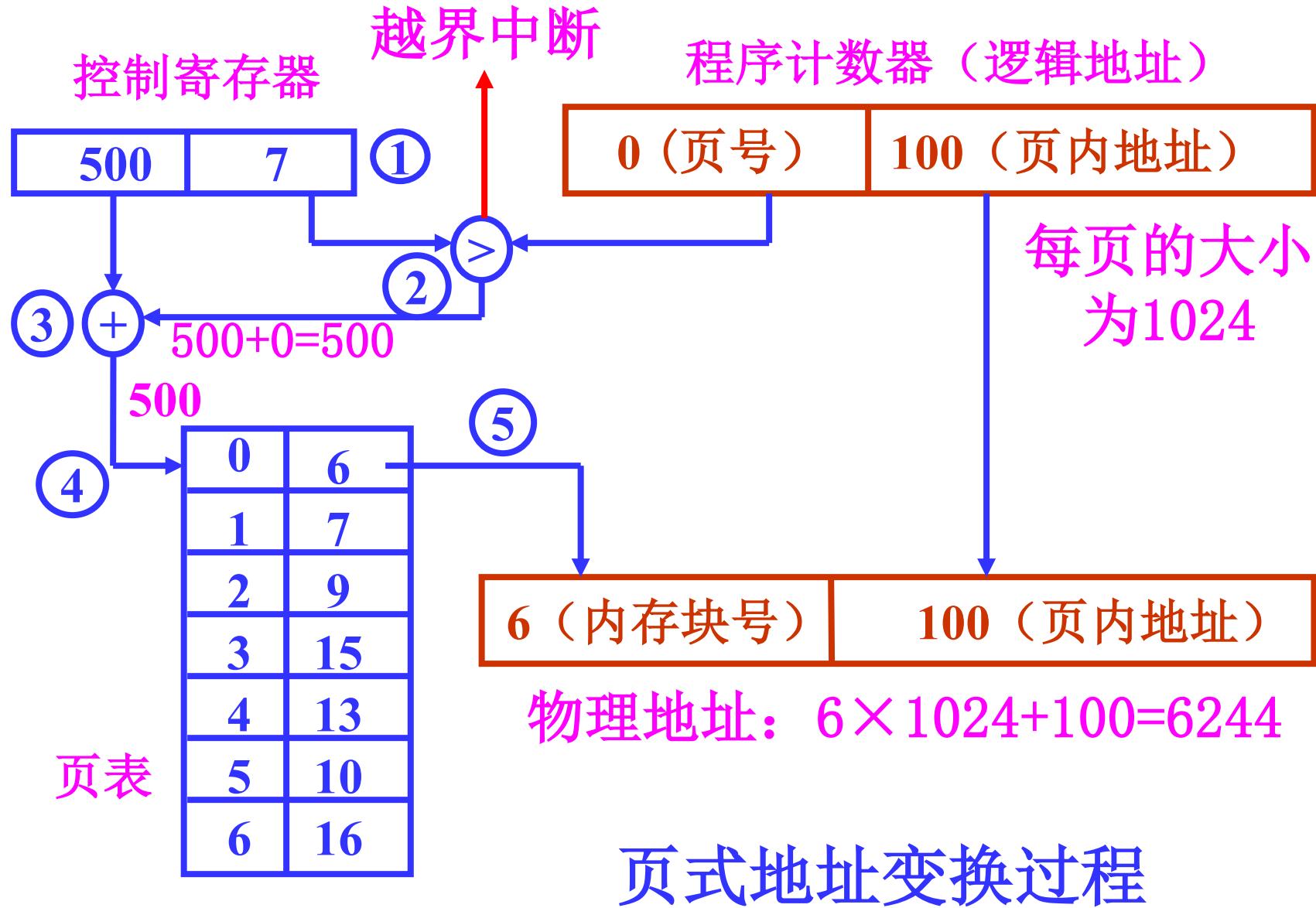
(4) 放置策略：就是确定进程的各个页分配到内存的哪些页面中，以及用什么原则挑选内存中的页面。

## 2. 页式动态地址变换

[例] 进程地址空间共有7个页，每页的大小为1024B。其对应的主存块在页表中已列出。假定页表在主存始址为500。若该程序从第0页开始运行。某时刻，程序计数器内容为：

程序计数器：

0	100
---	-----



- **注意：**在将虚拟地址分为页号和页内地址两部分时，要将页号与页表长度进行比较。
- 该地址变换过程是由**硬件地址变换机构**自动完成的。

### 3. 快表和联想存储器

- 在上述地址转换过程中，执行一次访内操作至少要访问主存两次。降低了程序的执行速度。
  - ◆ 一次访问页表；
  - ◆ 一次实现指定操作

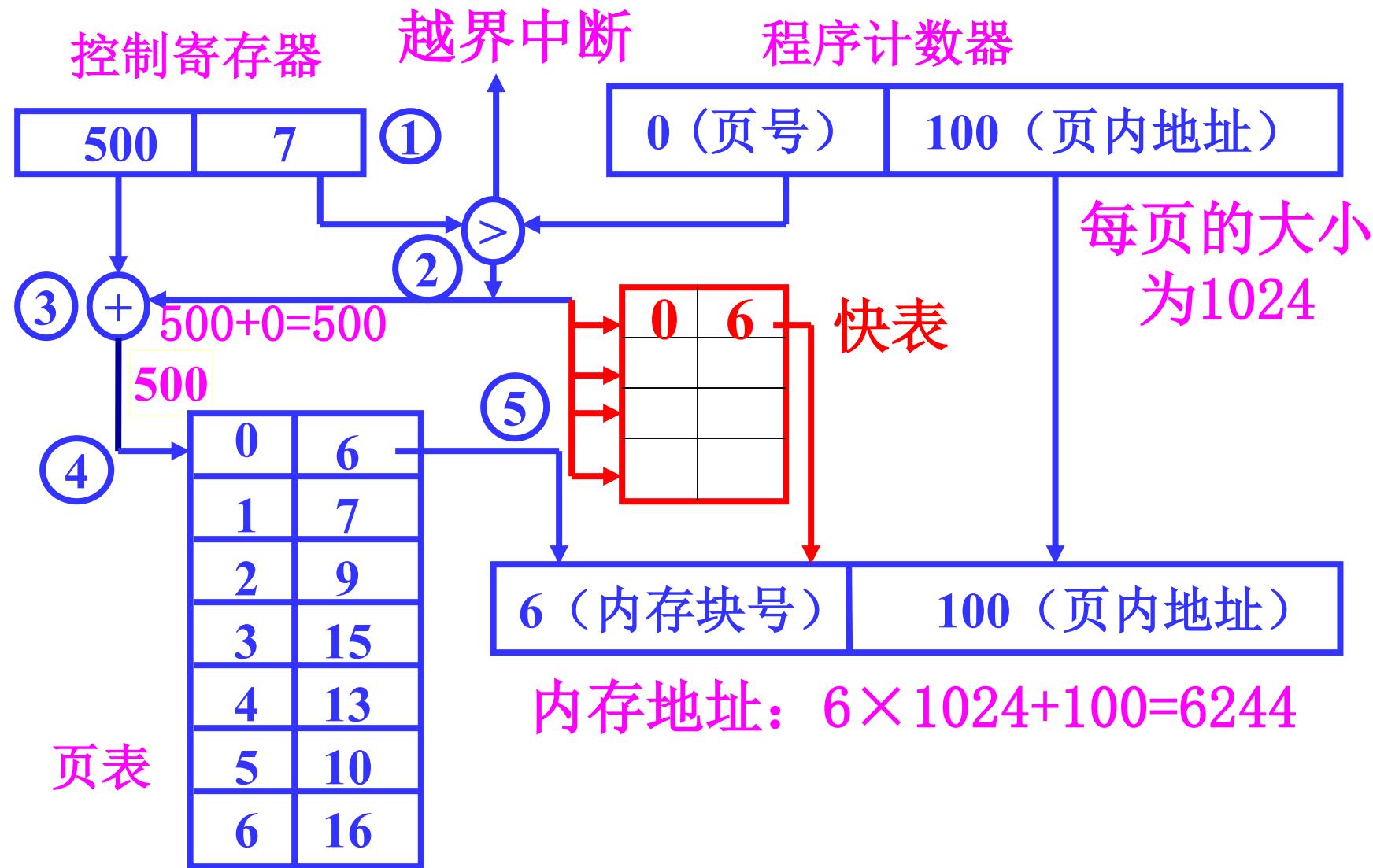
# 联想存储器

- 联想存储器: (TLB, Translation Lookaside Buffer)。为了提高存取速度，在地址变换机构MMU中增设一个具有并行查找能力的硬件高速缓冲寄存器组。用来存放最近访问过的页表项。
- 存取速度比主存快，造价高。用8~16个寄存器，就可提高程序执行速度。

# 快表

- 快表：存放在高速缓冲寄存器中的页表。
- 快表的格式
  - ◆ 状态位：指示该寄存器是否被占用。0表示空闲，1表示占用。
  - ◆ 访问位：指示该页最近是否被访问过。0表示没有访问过，1表示访问过。

页 号	块 号	访 问 位	状 态 位



使用快表后的地址变换过程

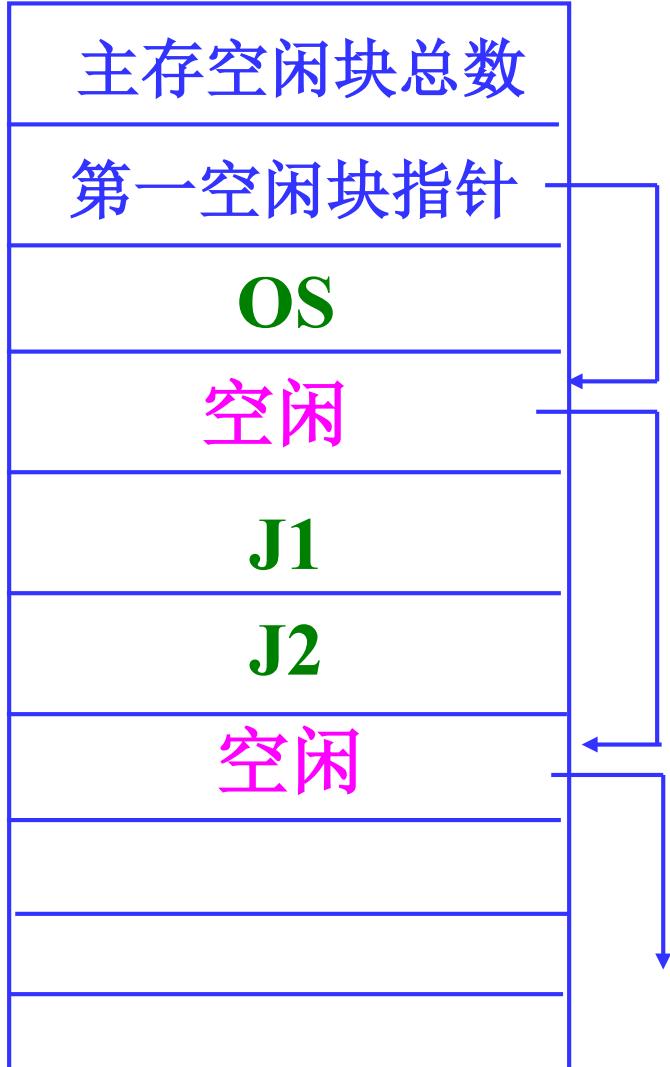
# 引入快表的地址变换过程

- 同时开始两个变换：利用主存`页表`进行的正常变换；利用`快表`进行的快速变换。
  - ◆ 快表中有待查找的页号时，立即停止正常访问主存页表的过程。
  - ◆ 快表中没有要查询的页。则继续正常的转换过程。还要把从主存页表中取出的块号和CPU给出的页号一起写入快表中。（找空闲快表项；无时再找最近没被访问的快表项）

# 4. 页式管理的主存分配与保护

## (1) 页式主存分配

- ① **页表**: 每个进程一个，在主存，用来实现将进程的虚页转换成主存的物理块。
- ② **进程控制块**：存有页表在主存的始址和页表长度。
- ③ 存储空间使用情况表
  - a. 存储分块表
  - b. 位示图



a. **存储分块表**: 记录存储器中的块的占用情况。表的第一项指出当前空闲块总数，第二项为指向第一个空闲块的指针，各空闲块通过单向链链接在一起。

- ◆ 分配主存时，查存储分块表。先查空闲块总数是否够用，再分配。同时为进程建立页表。
- ◆ 进程完成时，回收主存块。

b. 位示图：每个主存块对应位示图中的一位。0表示空闲，1表示被占用。

- ◆ 主存块号 $\leftrightarrow$ 位示图中的字节、位
- ◆ 块越小，位示图越大；块越大，位示图越小。

1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0
0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

## (2) 页式管理的存储器共享与保护

- ◆ 在系统中，很多程序代码是可共享的。
- ◆ 保护：可在页表中增加对该页的操作方式位，以表示是可读写/只读/只执行等。

# 页式管理的分类

## ● 静态页式管理

该管理方法在作业或进程开始执行之前，把该作业或进程的程序段和数据全部装入内存的各个页面（或物理块）中，并通过页表和硬件地址变换机构来实现虚拟地址到内存物理地址的地址映射。

## (1) 静态页式管理的特点

它解决了内存中的碎片问题，但它要求将作业或进程的所有页面一次调入内存。当内存可用空间不足或进程太大时，就会限制一些进程进入内存运行。为此，提出了动态页式存储管理方法。

## ● 动态页式管理

### (1) 动态页式管理的二种调入方式

它分为请求页式管理和预调入页式管理，这两种方式在作业或进程开始执行之前，都不是将作业或进程的程序段和数据段一次性地全部装入内存，而是只装入经常反复执行和调用的工作区部分。其他部分则在执行过程中动态装入。

## (2) 动态页式管理的二种调入方式的主要区别

主要区别在调入方式上：

**请求页式管理的调入方式**是：当需要执行某条指令而又发现它不在内存时，或当执行某条指令需要访问其他的数据或指令时，这些指令和数据不在内存中，从而发生缺页中断，系统将外存中相应的页面调入内存。

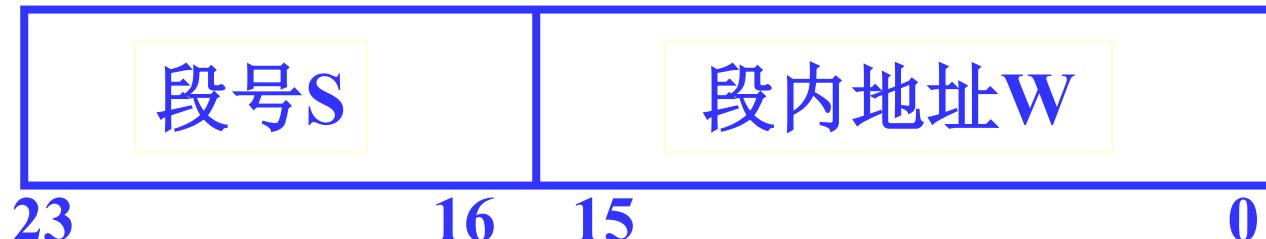
**预调入方式**是：系统对那些在外存中的页进行调入顺序计算，估计出这些页中指令和数据的执行和被访问的数据，并按此顺序将它们依次调入和调出内存。

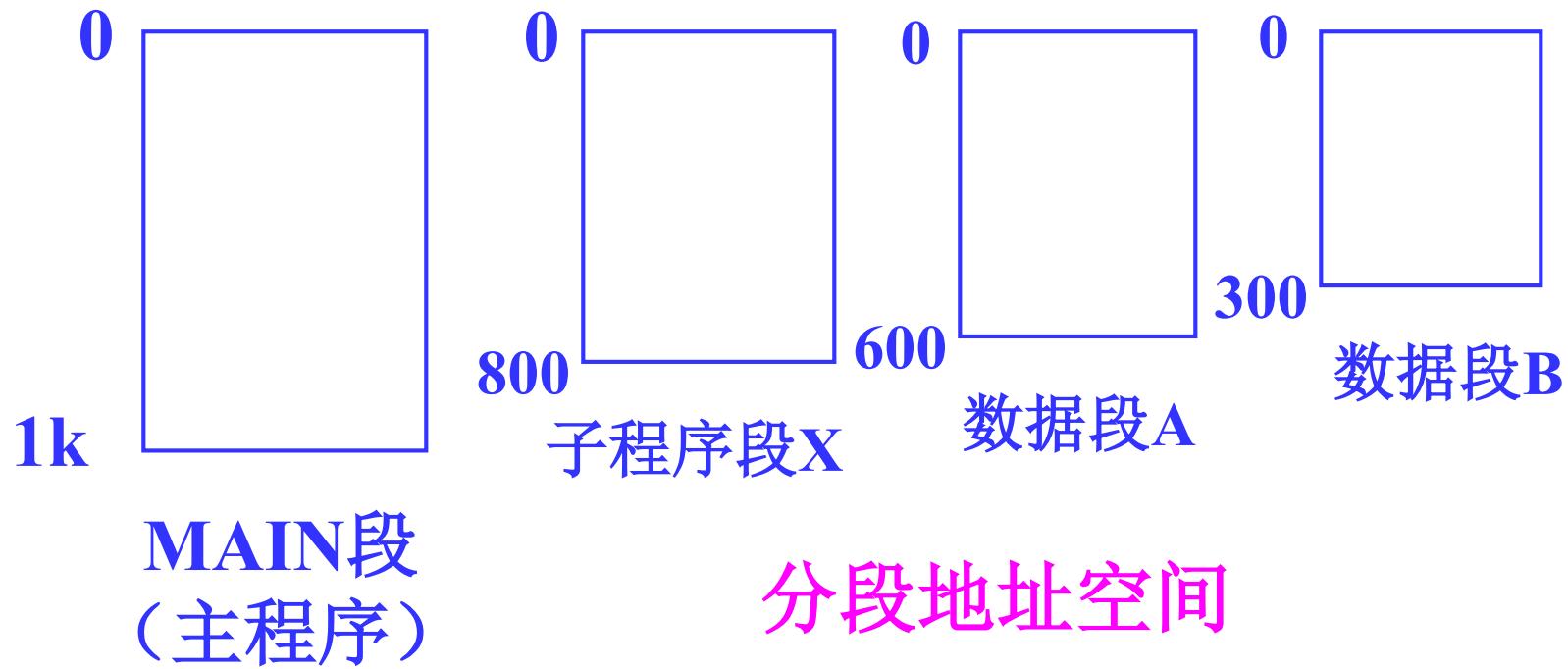
## 4.6 段式存储器管理

- 前面介绍的各种存储管理技术中，用户的地址空间已被连接成一个一维的空间。
- 通常，一个进程由若干个程序段和数据段组成。**共享**用户编写的某些程序段和数据段是现代操作系统必须解决的问题。

# 1. 段式管理的实现原理

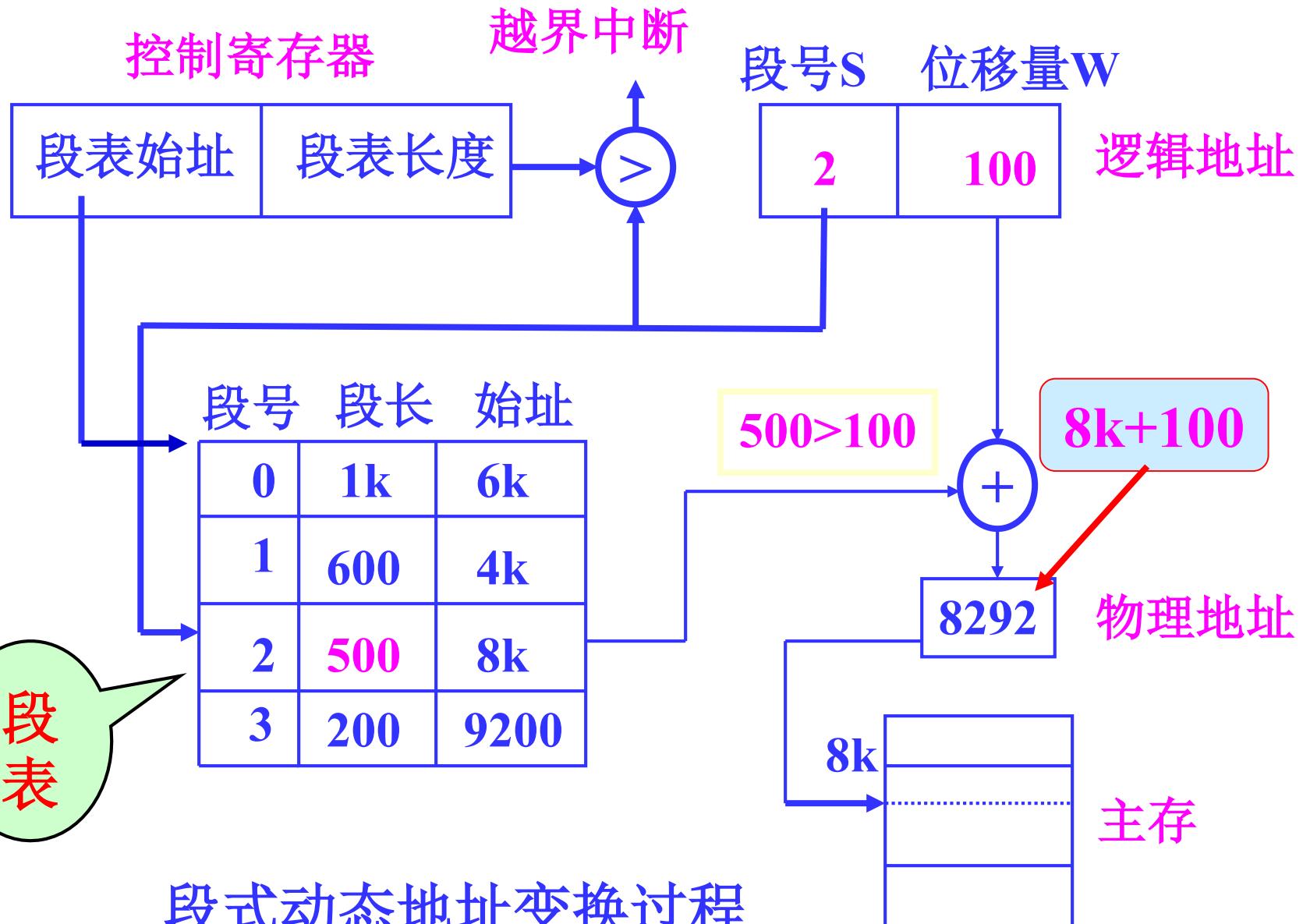
- ◆ 把每个进程的地址空间按照程序自身的逻辑关系划分成若干段，每个段都有自己的段名。
- ◆ 每个段都有从“0”开始编址的一维地址空间。
- ◆ 进程的地址空间是二维地址空间。
- ◆ 逻辑地址：段号S和段内地址W。段名→段号





## 2. 段式动态地址变换

- 与页式管理基本相同，
- 段表：实现从逻辑地址到物理地址的变换。  
系统为每个进程建立一个段表。
- 由系统将该进程的段表始址和段表长度送入控制寄存器中。



### 3. 段式管理的存储保护和共享

#### (1) 段式管理的存储器保护

- 第一级保护：控制寄存器的段表长度>段号。
- 第二级保护：段表中的段长>段内地址。
- 对段的信息进行保护：在段表中增加相应的操作方式字段，对相应的段规定读、写、执行是否许可的操作权限。

#### (2) 段的共享

易实现信息的共享。是通过使各进程的段表项指向共享段的物理地址来实现的。

## 4. 段的存储器分配

- 类似于可变式分区。分配策略同样可采用首次适应法、最佳适应法、最坏适应法。
- 可变式分区管理是以进程为单位分配一个连续的分区；段式管理以段为单位分配分区，各段之间可以占有不连续的分区。
- 同样不可避免碎片问题。

## 5. 段式与页式管理的比较

- (1) 段是由用户划分的；页是为了方便管理由硬件划分的，对用户是透明的。
- (2) 页的大小固定；段的大小不固定。
- (3) 段式用二维地址空间；页式用一维地址空间。
- (4) 段允许动态扩充，便于存储保护和信息共享。
- (5) 段可能产生主存碎片；页消除了碎片。
- (6) 段式管理便于实现**动态链接**，页式管理只能进行**静态链接**。
- (7) 段与页一样，实现地址变换开销大，表格多。

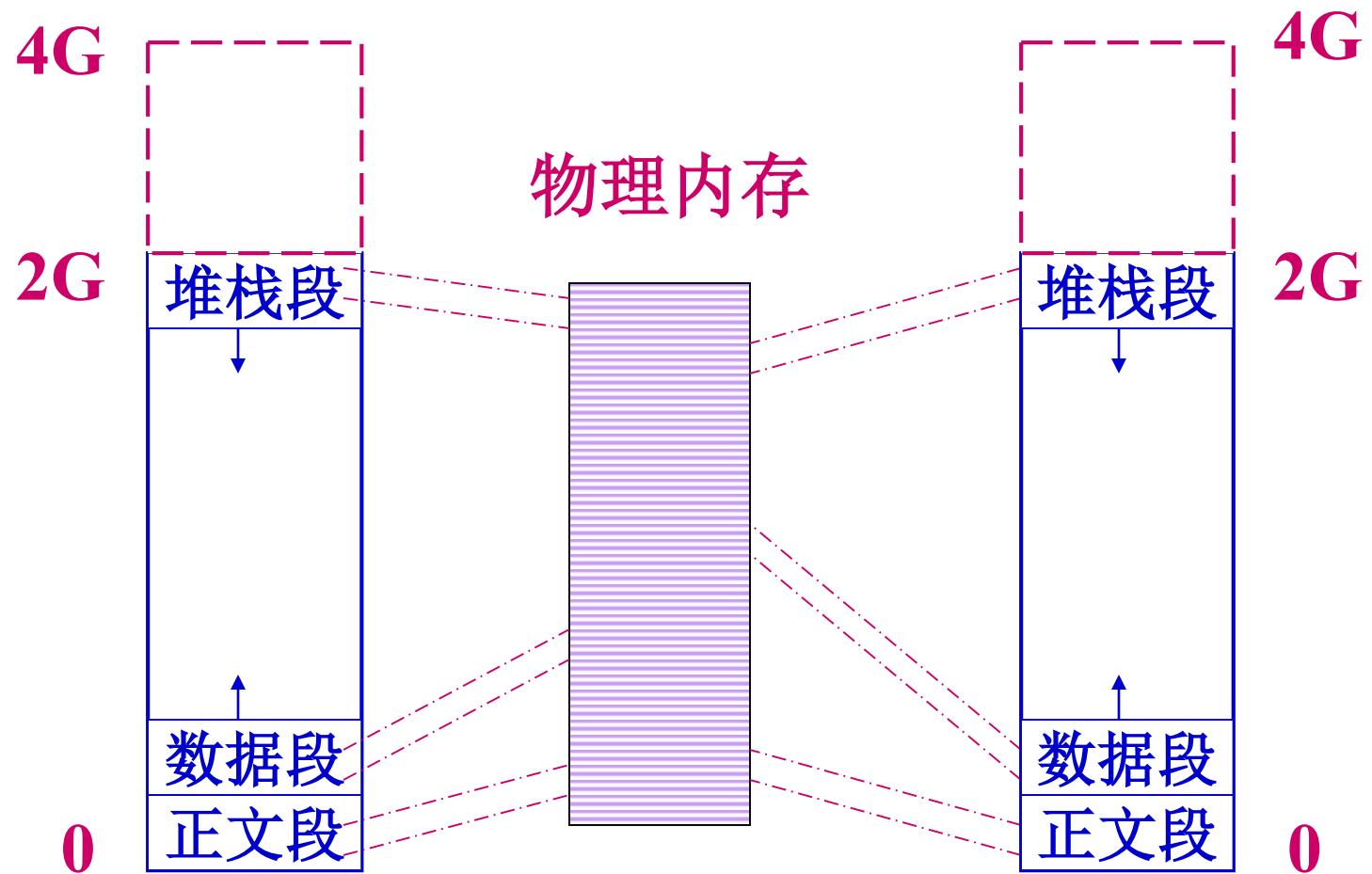
## 4.7 虚拟存储器管理

- 实存管理技术：进程运行时，整个进程的地址空间必须全部装入主存。
- 虚拟存储技术：允许进程的执行实体不必完全在内存中。程序可以比物理内存大。
- 现在，功能较强的计算机均采用了虚拟存储器管理技术。

**1. 虚拟存储器：**是系统为了满足应用对存储器容量的巨大需求而构造的一个非常大的地址空间。

- 其容量由计算机的地址结构确定。系统的指令地址部分能覆盖的地址域大于实际主存的容量。

**[例]** 某机器的主存容量为**1M**（**20位**），而机器的指令地址部分能覆盖的地址空间为**4G**（**32位**）。地址空间大于实际存储器容量。



进程A的  
虚拟地址空间

进程B的  
虚拟地址空间

## 2. 程序访问的局部性原理

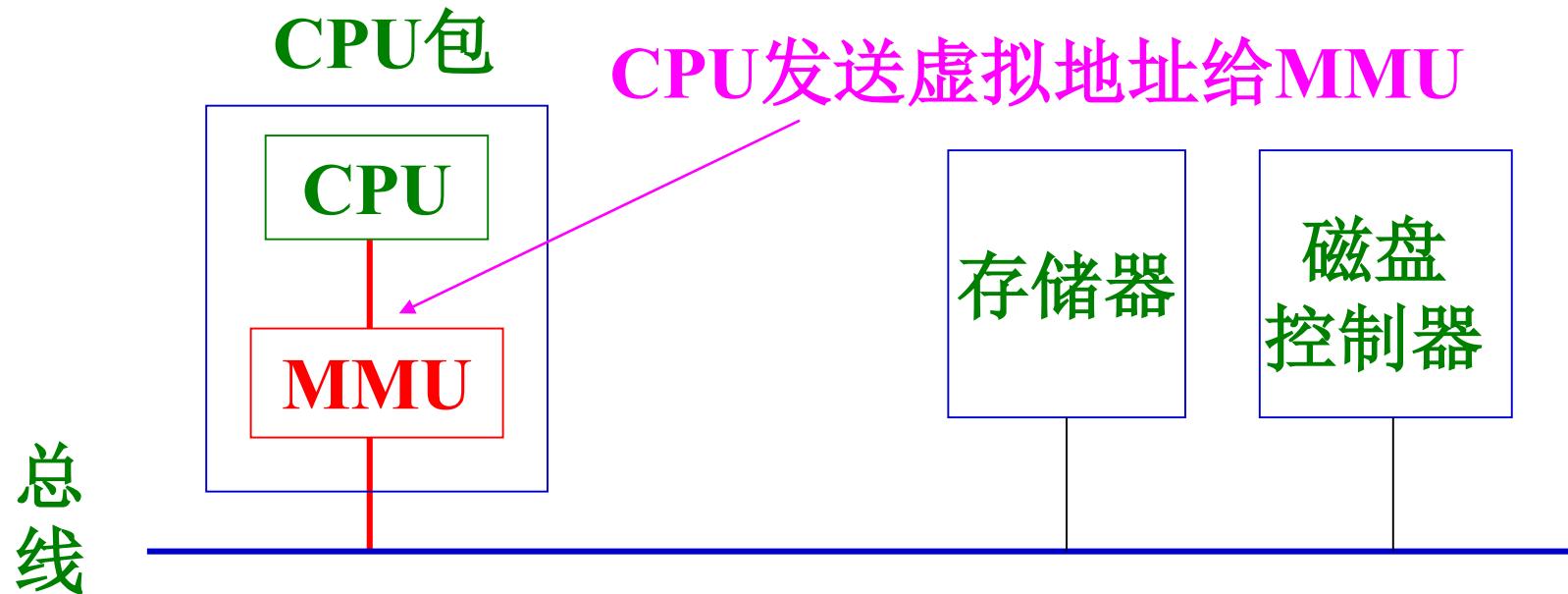
- 1) **时间局部性**: 程序中往往含有许多循环，在一段时间内会重复执行该部分。
- 2) **空间局部性**: 程序中含有许多分支，在一次执行中，只有满足条件的代码运行，不满足条件的代码不运行。即使顺序执行程序，程序的地址域在短时间内变化不大。
- 3) 在进程运行过程中，用到哪部分程序或数据再由系统自动装入。

### 3. 支持虚拟存储器的物质基础

- 有一个大的CPU地址结构。
- 采用多级存储结构（最流行的为二级）：要有大容量的外存，足以存放多用户的程序；要有一定容量的内存。
- 地址转换机构(MMU)，以动态实现虚地址到实地址的地址变换。



外存的容量  
主存的存取速度



CPU发送虚拟地址给内存管理单元  
(Memory Management Unit, MMU) ,  
MMU把虚拟地址映射为物理内存地址。

## 4.7.2 页式虚拟存储器管理

### 1. 实现原理

- 页式虚拟存储器管理：页式管理 + 交换技术
- 请求页式管理与页式管理的主要区别：是将进程信息的副本存放在磁盘辅助存储器中，并为其建立一个外页表，指出各页对应的辅存地址。当进程被调度运行时，先将进程中的较少页装入主存，在执行过程中，访问不在主存页时，再将其调入。

# 页面调度策略

## ① 取页

- a. 请求调页
- b. 预调页：需要调入某页时，一次调入该页以及相邻的几个页。能减少I/O次数。

② 置页：把调入的页放在内存何处

③ 置换：固定分配局部置换。

进程地址空间的页有的在主存，有的在辅存，为此要修改页表。

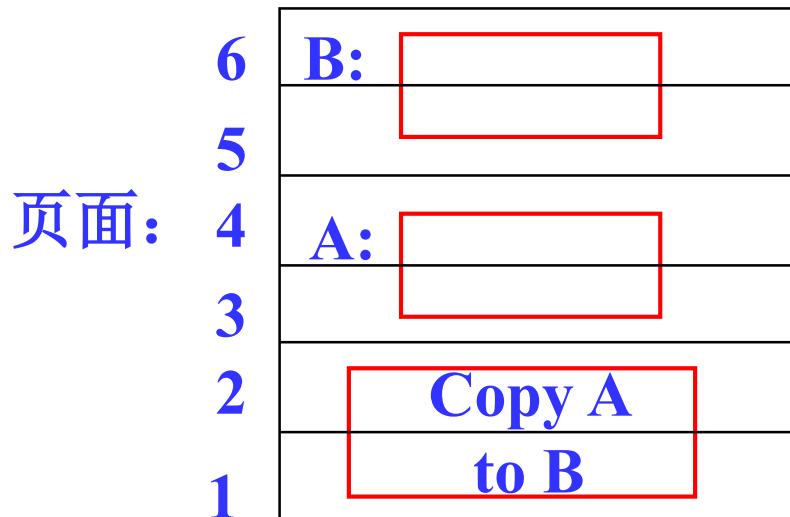
### 页表项：

页号	块号	有效位	修改位	访问位	外存地址
					保护码

保护码：对该页的读/写/执行操作的许可。

- (1) 有效位（状态位）：用来指示某页是否在主存。
  - 为1表示该页在主存，完成正常的地址变换；
  - 为0表示该页不在主存，由硬件发出一个缺页中断，转操作系统进行缺页处理。
- (2) 修改位：指示该页调入主存后是否被修改过。“1”表示修改过，“0”表示未修改过。
- (3) 访问位（引用位）：指示该页最近是否被访问过，“1”表示最近访问过，“0”表示最近未访问。

- 通常，CPU都是在一条指令执行完后去检查是否有中断请求到达的。
- 产生缺页中断时，一条指令没执行完，OS进行缺页中断处理后，应重新执行被中断的指令。
- 在一条指令执行期间可能产生多次缺页中断。



数据块B 跨了两个页面

数据块A 跨了两个页面

可能产生6 次缺页中断

# 缺页处理过程简述

1. 根据当前执行指令中的逻辑地址查**页表**的状态位。
2. 状态位为0， 缺页中断。
3. 操作系统处理**缺页中断**， 寻找一个空闲的内存页。
4. 若**有**空闲页，则把从磁盘读入信息装入该页面。
5. 若**无**空闲页，则按某种算法选择一个已在内存的页面，暂时调出内存。若修改过还要写磁盘。调入需要的页。之后要修改相应的**页表**和**内存分配表**。
6. 恢复现场，重新执行被中断的指令。

**阻塞状态：**由于从外存向主存调入一页需要的时间较长，故在调页过程中应将请求调页的进程置为阻塞状态。

**唤醒：**直到该页装入主存再将其唤醒。

## 2. 页面淘汰算法

**页面淘汰：**当主存空间装满各运行程序页时，若再产生缺页中断，操作系统必须按一定的算法把已在主存的某页淘汰出去。

**抖动 (thrashing) 现象：**刚被淘汰的页面马上又要用，因而又要把它调入。调入不久再被淘汰，淘汰不久再次装入。如此频繁地调入调出，降低了系统的处理效率。

进程在运行过程中的缺页率:  $f=F/A$

其中:  $A=S+F$ ,

$A$ 为进程执行过程中总的访问次数;

$S$ 为成功的访问次数;

$F$ 为不成功的访问次数。

# 页面置换算法

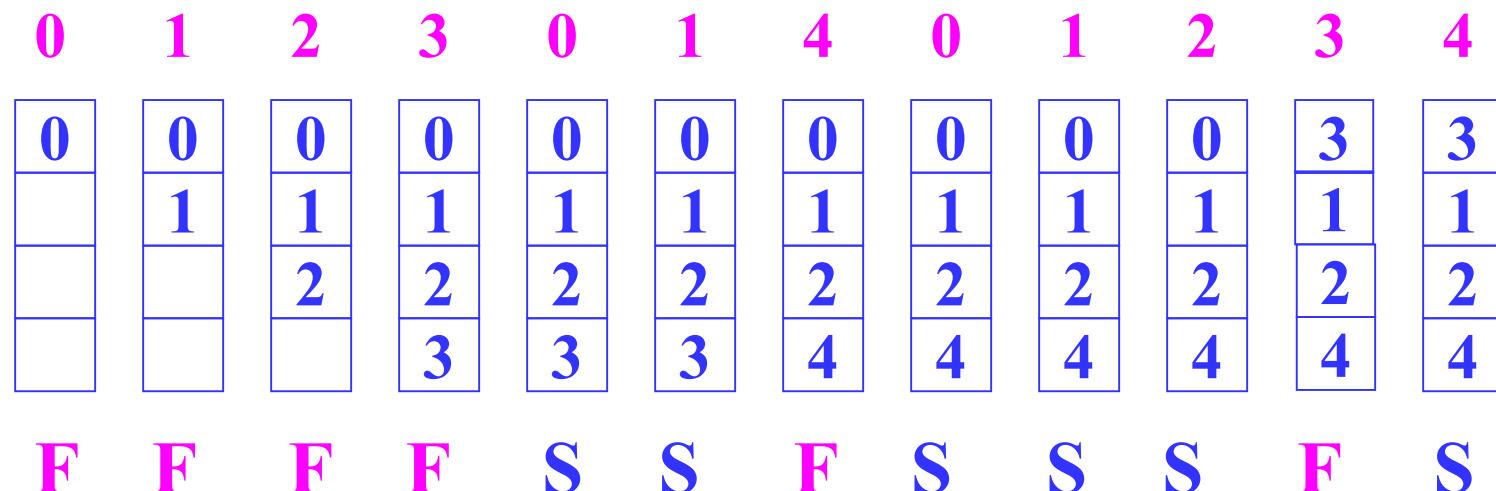
- 最佳置换算法 (OPT算法)
- 先进先出置换算法(FIFO)
- 最近最少使用的页面置换算法(LRU)
- 时钟页面置换算法

算法假设：一个进程分配的主存块数固定不变，且采用局部置换(在本进程内部置换)。

# (1) 最佳置换算法

简称OPT (optimal)算法。选择以后不再访问的页或经很长时间之后才可能访问的页进行淘汰

[例] 某进程有5个页面，执行时页面的引用序列为：0、1、2、3、0、1、4、0、1、2、3、4



分配四个内存块，缺页6次

## (2) 先进先出置换算法(FIFO)

- 当淘汰一页时，选择在主存驻留时间最长的那一页。
- 由操作系统维护一个所有当前在内存中的页面的链表（链队列），最老的页面在表头，最新的页面在表尾。当发生页面失效时，淘汰表头的页面，把新调入的页面加到表尾。

- 有可能出现抖动：因为在主存驻留时间最长的页未必是最长时间以后才被访问的页。频繁地调入调出。
- Belady异常：Belady在1969年发现，采用FIFO算法，当为进程分配的主存块多时，有时产生的缺页中断次数反而增多。

[例] 某进程有5个页面，执行时页面的引用序列为：0、1、2、3、0、1、4、0、1、2、3、4

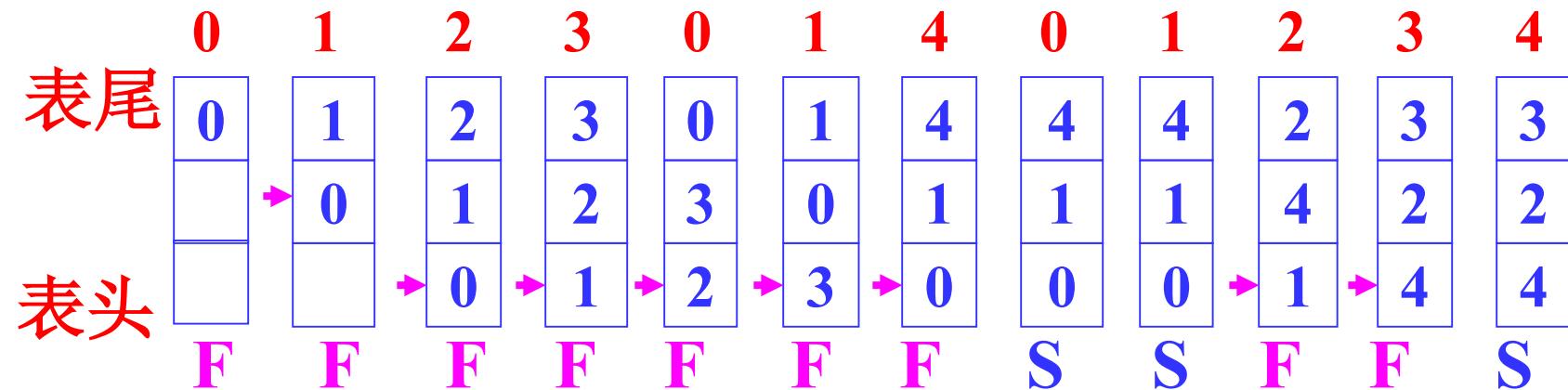
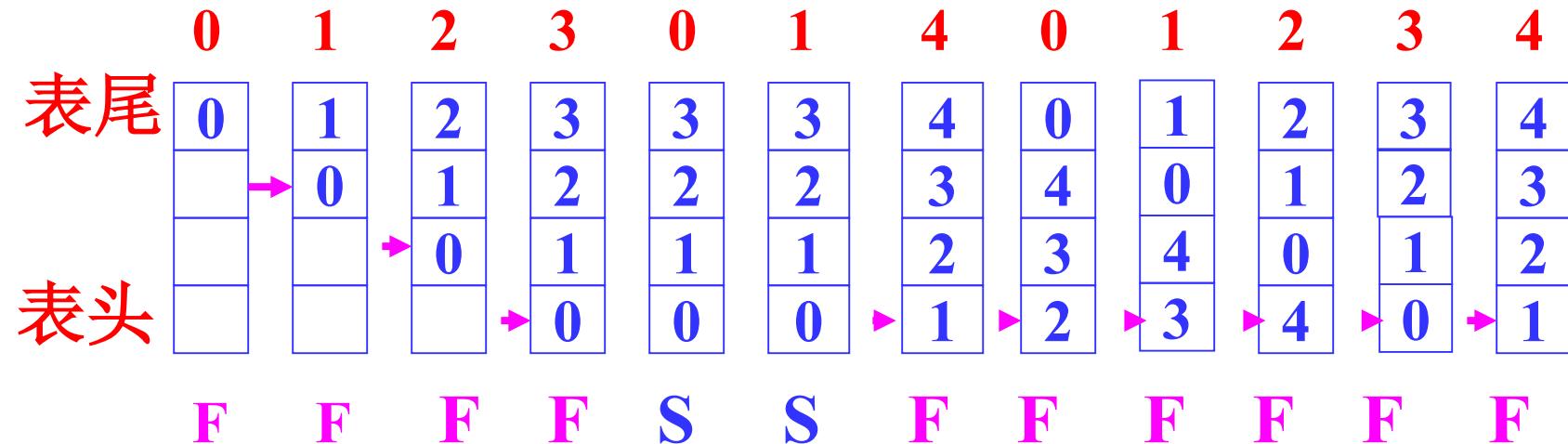


图4.21 FIFO (3个内存块时, 缺页9次)



FIFO的Belady异常 (4个内存块时, 缺页10次)

### (3) LRU页面置换算法

- 最近最少使用LRU, Least Recently Used。
- 根据局部性原理，淘汰那些在最近一段时间里最少使用的一页。
- 进程的工作集：在一段时间内，进程集中在一组子程序或循环中执行，导致所有的存储器访问局限于进程地址空间的一个固定的工作集。

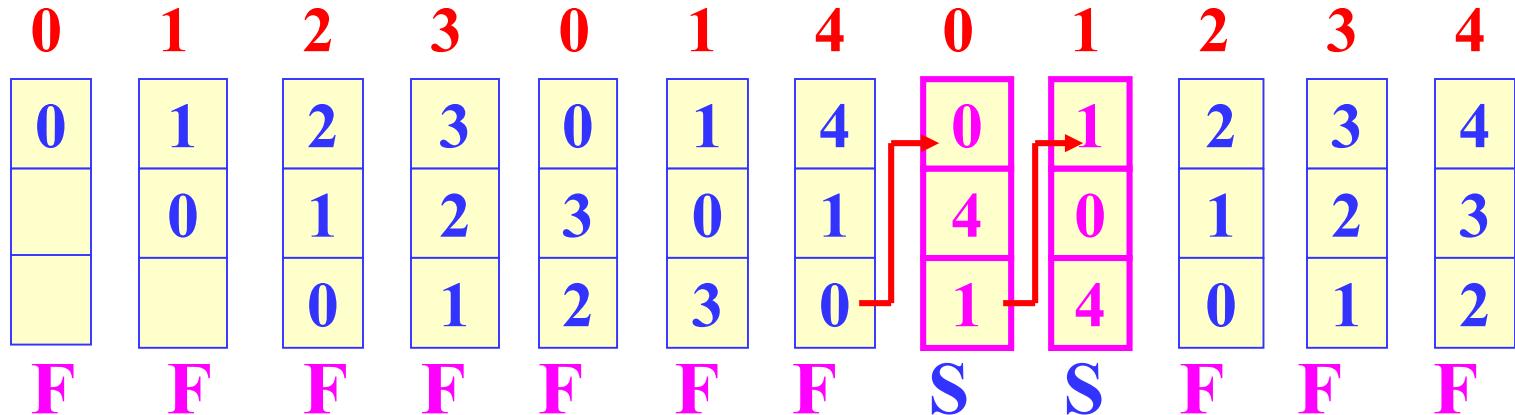
# LRU的第一种实现

- 要求硬件有一个64位计数器C，每当一条指令所在的页被引用后都自动加1。
- 每个页表项含有存放该计数器值的字段。
- 每次访问主存时，把C值存入页表项。
- 缺页中断时，系统查找页表，找出计数值最小的页作为最近最少使用的页，淘汰之。

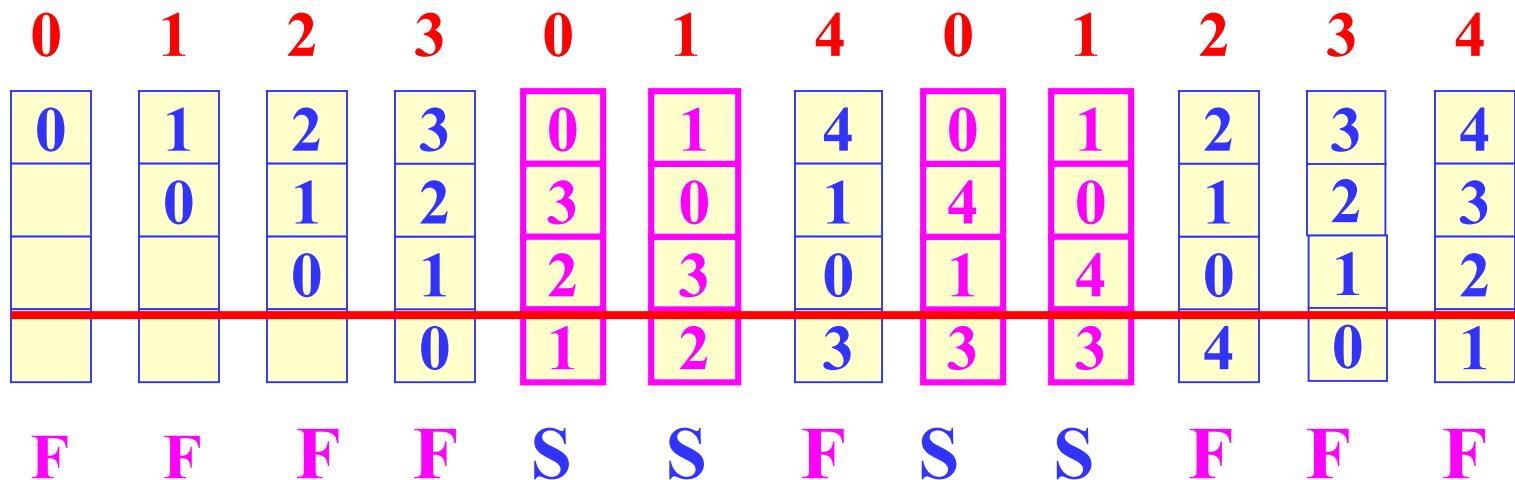
# LRU的第二种实现

- 利用**栈**记录页的使用。正在引用的页放在栈顶，最近最少使用的页放在栈底。
- 为了便于栈中元素移动，可采用**双向链**。

[例] 某进程有5个页面，执行时页面的引用序列为：0、1、2、3、0、1、4、0、1、2、3、4



LRU (3个内存块时，缺页10次)



LRU (4个内存块时，缺页8次) 无Belady异常

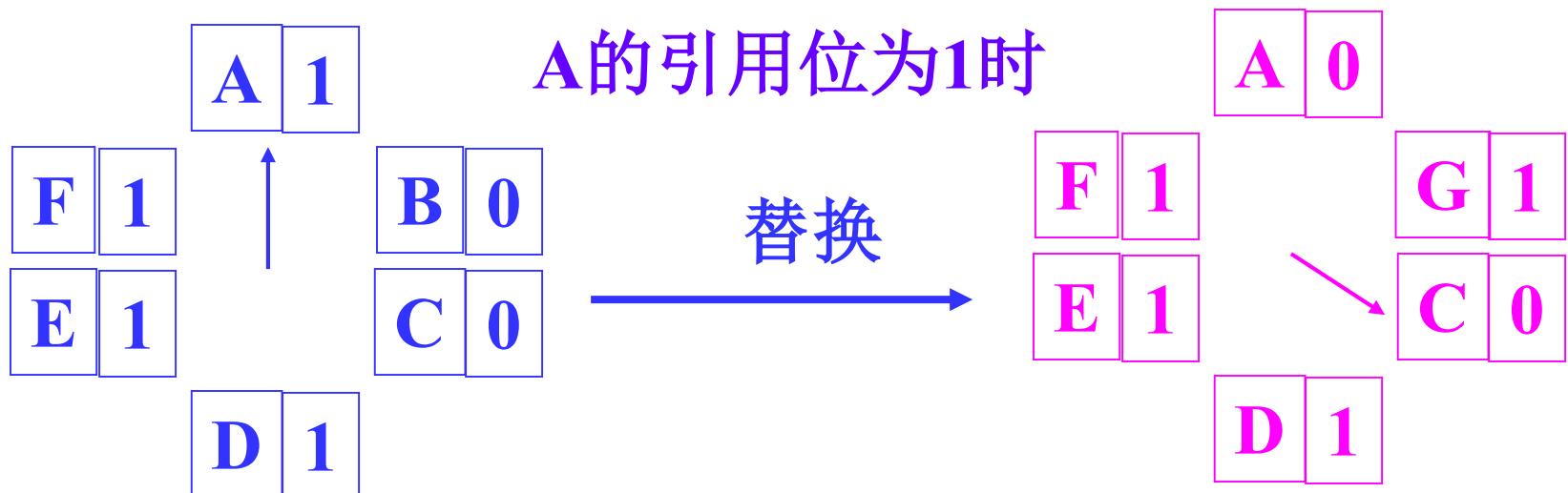
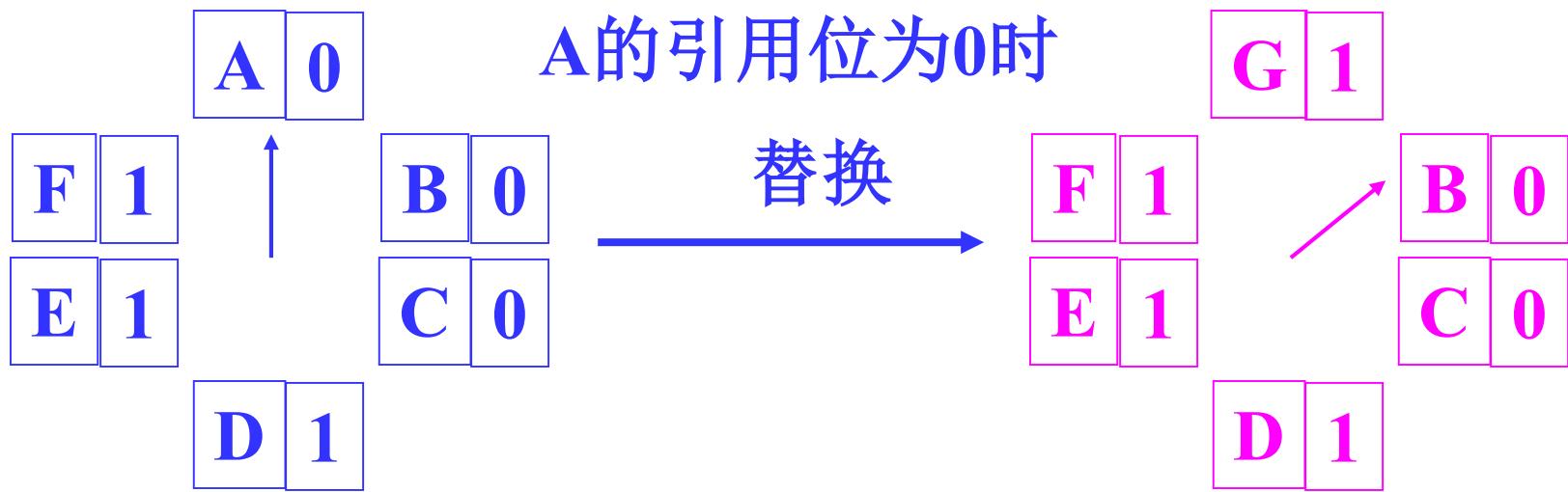
具有LRU算法特性的一类算法所具有的性质：

$$M(m, r) \subseteq M(m+1, r)$$

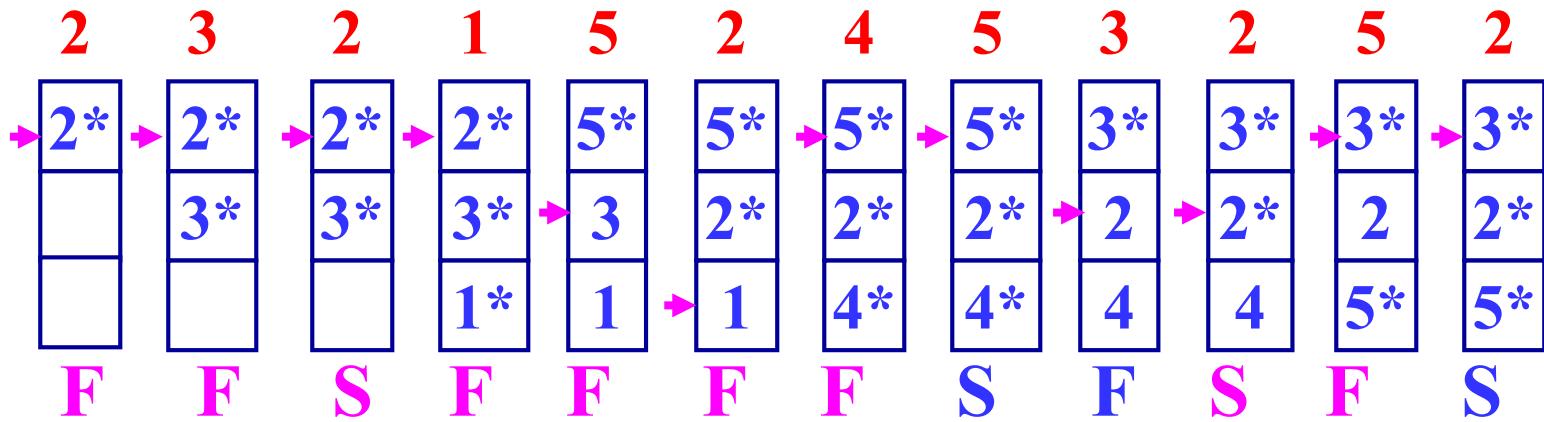
- $m$ 为分配的主存块数， $r$ 为页面引用序列。
- 公式表明：在任何时刻，主存块为 $m+1$ 时，存于主存中的一串页面中必然包含主存块为 $m$ 时存于主存中的各页。决不会出现Belady异常。

## (4) 时钟页面置换算法

- 第二次机会算法，近似LRU算法。
- 页表中的引用位：0表示最近未被引用。
- 将进程所访问的页放在一个像时钟一样的循环链中。链中的节点数就是为之分配的主存块数。
- 系统设有一个指针指向最早进入主存的页。
- 淘汰时，检查指针所指页。若引用位为0，则用新页置换之，指针向前走一个位置。若引用位为1，清0，指针前进，直到找到引用位为0的页。



# Clock算法



### 4.7.3 页式管理设计中应考虑的问题

1. 交换区的管理
2. 页尺寸
3. 页的共享
4. 多级页表的结构
5. 写时复制技术（copy-on-Write）

# 1. 交换区的管理

- 交换区是OS利用磁盘扩充主存的主要方法。
- 系统使用交换区的方法：用交换区保存进程运行的整个映像；或存储分页系统可能被淘汰的页。
- 交换空间的设置：可以从文件系统中分割一部分空间作为一个大文件使用；或者占用一个独立的磁盘或磁盘分区。
- Windows 2000使用“**页文件**”作为主存补充的磁盘的那部分空间。当系统启动时，打开页文件。

## 2. 页尺寸

- 操作系统设计者可以选择页的大小。**4KB/4MB**
- 页尺寸大：**页的内部碎片大**（进程的代码段、数据段、堆栈段的最后一页都有内部碎片）。
- 页尺寸小：进程的**页表长**。
- 设进程为 $s$ 字节，页大小为 $p$ 字节，页表项占 $e$ 字节，  
页的内部碎片平均为 $p/2$ 。因此页表和内部碎片引起的系统的总开销为 $se/p + p/2$ ,
- 对 $p$ 求导，可得出页的最佳尺寸。

### 3. 页的共享

- 页的共享可以避免在主存中同时有多个相同页的副本。**节约主存。**
- 通常，只读页（如程序文本）可以共享，可**读/写**的数据页不可共享。
- 需要一个专门的**数据结构**来记录**共享页**。把共享页锁在内存，且在页表中增加引用计数项，仅当其引用计数为0时，才允许调出或释放盘空间。

## 4. 多级页表的结构

- 大页表：(地址空间4GB)/(页4KB)=1M个页表项。若每个页表项占4B，则最大的页表为4MB。
- 多级页表结构：页表在内存不必连续存放。
- 页表的建立不再是在进程装入主存时，而是推迟到要访问页时，才为包含该页的页表分配空间和建立页表项。
- Linux, Windows

31

22 21

12 11

0

页目录索引

页表索引

页内字节索引

处理机寄存器CR3

页目录表

页表地址

第*i*个页表

页框地址

物理主存

代码或数据页

## 5. 写时复制技术的利用

- **写时复制的页面保护：**若没有进程向共享主存页写时，两个进程就共享之。若有进程要写某页，系统就把此页复制到主存的另一个页框中，并更新该进程的页表，使之指向此复制的页框，且设置该页为可读/写。
- **父子进程之间的写时复制：**父子进程最初共享父进程的所有页，将这些页标记为写时复制。
- **Windows , Linux**

# 页式管理的优点

- (1) **有效地解决了碎片问题**: 是由于不要求作业或进程的程序段或数据在内存中连续存放。
- (2) 提供了**虚拟存储功能**, 从而提高了内存的利用率。

# 页式管理的缺点

- (1) 要求硬件的支持，从而增加了成本；
- (2) 增加了系统开销；
- (3) 页面置换算法如选择不当，有可能产生抖动问题；
- (4) 虽然消除了碎片，但每个作业或进程的最后一页总有一部分空间得不到利用。

## 4.7.4 段式虚拟存储器管理

### 1. 实现原理

又叫请求段式管理。把进程的所有段的副本存放在外存中。进程运行时，再把需要的段装入主存。

段表：

段始址	段长	操作方式	有效位	修改位	访问位	动态增长位

**有效位**: 为1表示在主存; 为0表示不在主存。

**访问位**: 含意同页式虚存管理。

**修改位**: 含意同页式虚存管理。

**动态增长标志位**: 是为动态数组和动态表设计的。

- ✓ 为1表示该段允许动态增长, 这样在段长越界中断时, 不做越界处理, 而由操作系统扩充其段长;
- ✓ 为0表示不允许动态增长。

- **进程执行**: 访问某段时，由硬件地址转换机构查**段表**。若该段在主存，则完成正常的地址转换。若该段不在主存，则硬件发一个缺段中断。由操作系统完成**缺段中断处理**。
- 缺段中断机构与缺页中断机构类似：
  - ✓ 在一条指令的执行期间产生和处理中断。
  - ✓ 在一条指令的执行期间可能产生多次缺段中断。
- 比缺页中断的处理**复杂**，因为段是不定长的。

## 2. 段的动态链接和装入

**静态连接：**是在生成可执行文件时进行的。

**动态链接：**是指在一个程序运行开始时，只需将主程序段装配好装入主存即可。在主程序段运行过程中，用到子程序或数据时，再将其装配好并与主程序段连接上。

需增加两个硬件功能：间接编址字、连接中断位。

- **间接编址字**：硬件指令的间接地址中包含的字。
- **连接中断位L**：表示是否需要动态链接的标志位。

间接字：



$L=0$ ：表示不要进行动态链接；

$L=1$ ：表示要进行动态链接，产生链接中断信号，转操作系统处理，完成段的动态链接。

[例]：以编译程序的工作为例。

编译程序所遵循的原则：

- 若该段的指令是访问本段地址，则编译成直接型指令；
- 若访问外段地址，则编译成间接型指令，并在直接地址中形成间接字：L=1和实际的直接地址。在直接地址中存放要链接段的段名和段内地址。

[例] 假设主程序段中有如下几条指令：

|  
10 CALL #0|100 调用0段的100地址，  
“#”代表间接编址字  
20 LOAD A #0|110 将0段110地址的内容  
取到寄存器A中

100 

0	[1]	150
1	[0]	<120>

 间接字 L=1, 因而产生链  
接中断, 转OS处理

110 

1	[0]	<130>
---	-----	-------

120 [X] | <Y>  
130 [C] | <D> ← 用符号名X和段内地址Y查符号表，  
得出X分配的段号为1, Y对应的  
段内地址为150。若在内存，不装入

- 主程序的第10条指令是一条子程序调用指令，“#”表示该指令是间接地址型，按“0|100”地址取出间接字，检查其L=1，因而产生链接中断，转OS处理。
- 操作系统根据间接字的地址“0|120”去取信息，得到欲访问段的符号名X和段内地址Y的。
- OS根据[X] | <Y>查符号表，查得为X分配的段号为1，Y对应的段内地址为150，再由段号1查段表，看第1段是否已在主存。若不在，再为该段分配主存，并装入。此时，再用1|150取代间接字中的地址0|120部分，并修改其L=0。至此，链接中断处理完成，返回到被中断指令重新执行。
- 当CPU再次执行“CALL # 0|100”这条指令时，检查间接字的L=0，所以按照间接字的地址部分“1|150”去实现子程序的调用功能。

### 3. 段的共享

利用段的动态链接很容易实现段的共享。由于各进程对共享段的使用情况不同，每个进程为其分配的段号也不同。

[例] 对于共享子程序SQRT(开平方函数)，进程1为其分配段号为1，进程2可能为其分配的段号为3。

为了便于多进程共享主存中的公用子程序，可在主存中设置一个共享段段表。其内容可包括：共享段名、在主存的始址、段长，调用该段的进程数及进程名。

# 段式管理的优点

- (1) 和动态页式管理一样，段式管理也提供了内外存统一管理的虚拟实现方案。不同的是，**段式虚拟存储每次交换的是一段有意义的程序段或数据段；**
- (2) 段长可根据需要**动态增长**；
- (3) 便于对具有完整逻辑功能的信息段进行**共享**；
- (4) 便于实现**动态链接**。

# 段式管理的缺点

它比其他几种方式要求更多的硬件支持，这就提高了机器的成本。另外，由于在内存空闲区管理方式上与分区管理相同，因而存在碎片问题。段式管理的另一个不足之处是：若淘汰算法选择不当，也有可能产生抖动现象。

## 4.7.5 段页式虚拟存储器管理

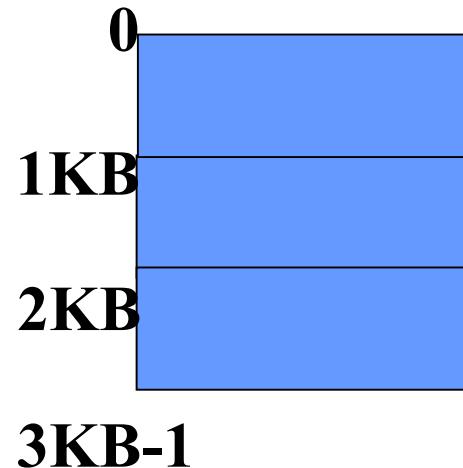
- 页式管理主存利用率高。
- 段式管理便于信息共享和存取保护。
- 段页式管理的基本思想：程序的逻辑结构按段划分，每段再按页划分。

作业的地址空间首先被分成若干个逻辑分段，每段都有自己的段号，然后再将每一段分成若干个大小固定的页。对于主存空间的管理仍然和页式管理一样，将其分成若干个和页面大小相同的存储块。



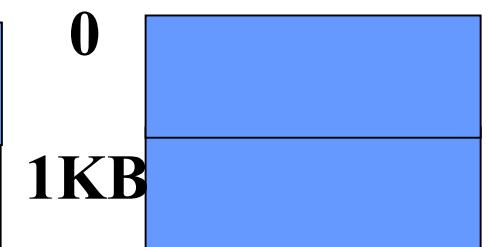
**4KB-1**

分段一程序



**2KB**  
**3KB-1**

分段二程序

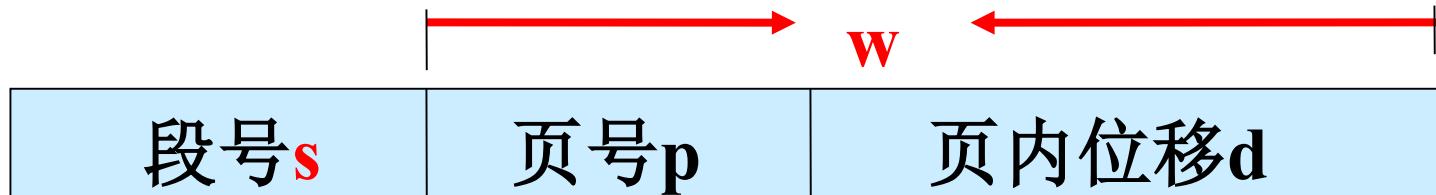


**1KB**  
**2KB-1**

数据分段

段页式地址空间

- 作业的地址结构包含三部分：段号、页号及页内位移。



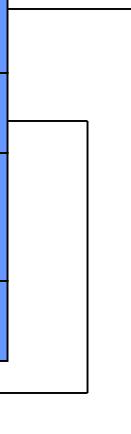
- 对于由这三部分组成的虚地址来说，程序员可见的仍然是段号s和段内位移w。地址变换机构将段内位移w的高几位解释为页号p，把剩下的低位解释为页内位移d。

- 系统为每个进程建立一张段表，而每个分段有一张页表。段表表目中至少包括段号、页表始址和页表长度，页表表目中至少包括页号和块号。系统中还有一个段表寄存器。
- 进行地址变换时，用段号s查段表，从中得到该段的页表始址，再利用逻辑地址中的页号p得到该页所在的物理块号，再与页内地址拼接成物理地址。
- 若段表和页表都在内存，则为了访问主存中的一条指令或数据，至少需访内三次。

# ➤ 段表、页表与内存的关系

段表

段号	页表长度	页表地址
0	2	
1	3	
...	...	
n		

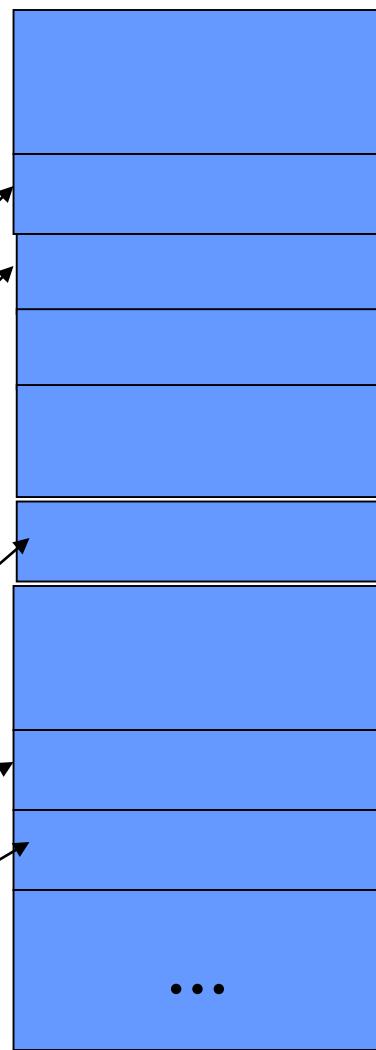


0段页表

页号	其他	页面号
0		
1		

1段页表

页号	其他	页面号
0		
1		
2		

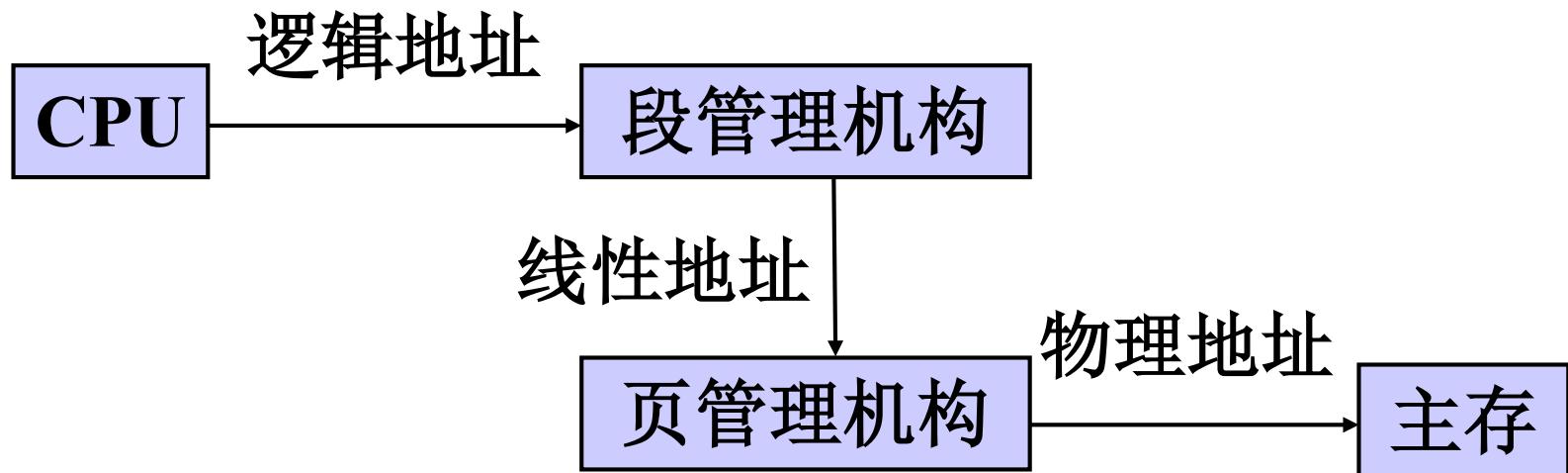


内存

- 可见，段页式存储管理中要得到内存地址须经过三次内存访问：
  - (1) 第一次访问段表，得到**页表起始地址**；
  - (2) 第二次访问页表，得到**内存页面号**；
  - (3) 第三次将内存页面号与页内地址组成，得到**内存地址**。
- 这样，虽然增加了硬件成本和系统开销，但在**方便用户**和**提高存储利用率**上很好的实现了存储管理的目标。

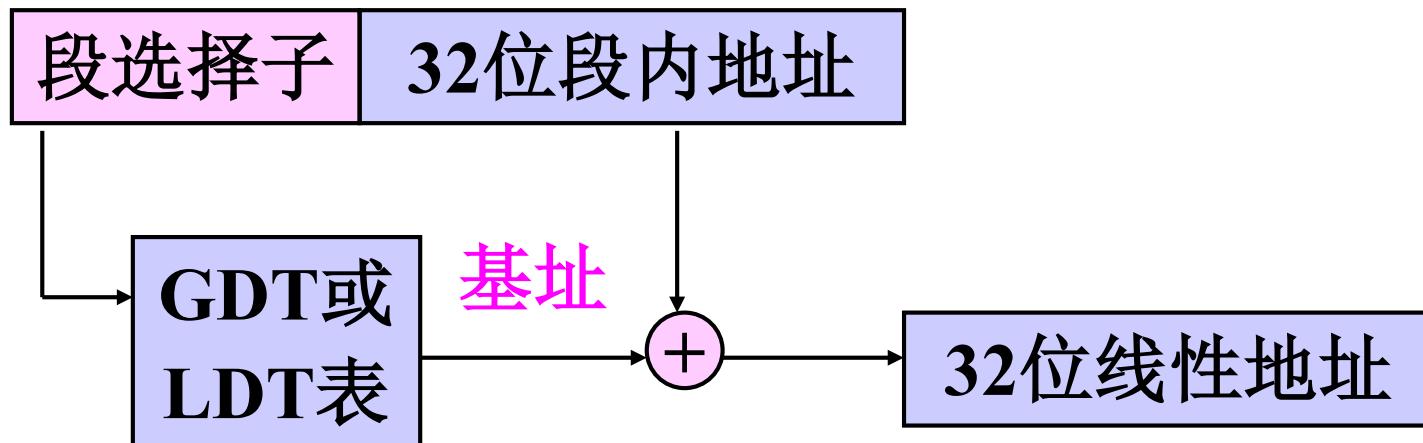
# Intel x86系统

- CPU产生的以段为单位的逻辑地址，经段硬件管理机构转换成一维的线性地址，再经页硬件管理机构转换成主存的物理地址。



# 1. x86系统的分段

- 以Linux系统为例。GDT进程共享的全局描述符表



# 段机制被屏蔽了

段	段基地址	段最大偏移
用户代码段	0x00000000	0xffff × 4KB
用户数据段	0x00000000	0xffff
内核代码段	0x00000000	0xffff
内核数据段	0x00000000	0xffff

## 2. x86系统的分页

- x86体系结构允许一个页的大小为4KB或4MB
- 4KB。 使用两级页表来索引所有的页。  
10位+10位+12位=32位
- 4MB。 只使用一级页表来索引所有的页。  
10位+22位=32位

## 4.8 小结

### □ 计算机系统常用的存储器管理方案：

- 从单道程序的单一连续区分配，到多道程序的分区分配；
- 从实存的页式管理、段式管理，到虚拟存储技术支持下的请求页式、段式和段页式管理。

### □ 几个主要概念：地址空间、存储空间、相对地址、物理地址、地址重定位，重定位类型，虚拟存储器。

## □ 各种存储器管理方案

- 动态地址变换机构，地址空间的相对地址变换为主存的物理地址的变换过程。
- 存储保护方法和优缺点。
- 所需要的各种数据结构。

## □ 页面置换算法，局部性原理，工作集的概念，抖动的概念和解决办法。

## □ 写时复制、多级页表。