

# 第四章 存储器管理



# 第四章 存储器管理

存储器是计算机系统的重要组成部分之一。随着计算机技术的发展，存储器容量一直在扩充，但仍不能满足现代软件和用户的需要，因此存储器仍是一种宝贵、紧俏的资源。对存储器加以有效管理，不仅直接影响存储器的利用率，而且对系统性能有重大影响。内存管理的主要对象是内存，对外存的管理在文件管理中。



# 第四章 存储器管理

4.1 存储器的层次结构

4.2 程序的装入和链接

4.3 连续分配存储器管理方式

4.4 对换 (Swapping)

4.5 分页存储管理方式

4.6 分段存储管理方式



# 第四章 存储器管理

4.1 存储器的层次结构

4.2 程序的装入和链接

4.3 连续分配存储器管理方式

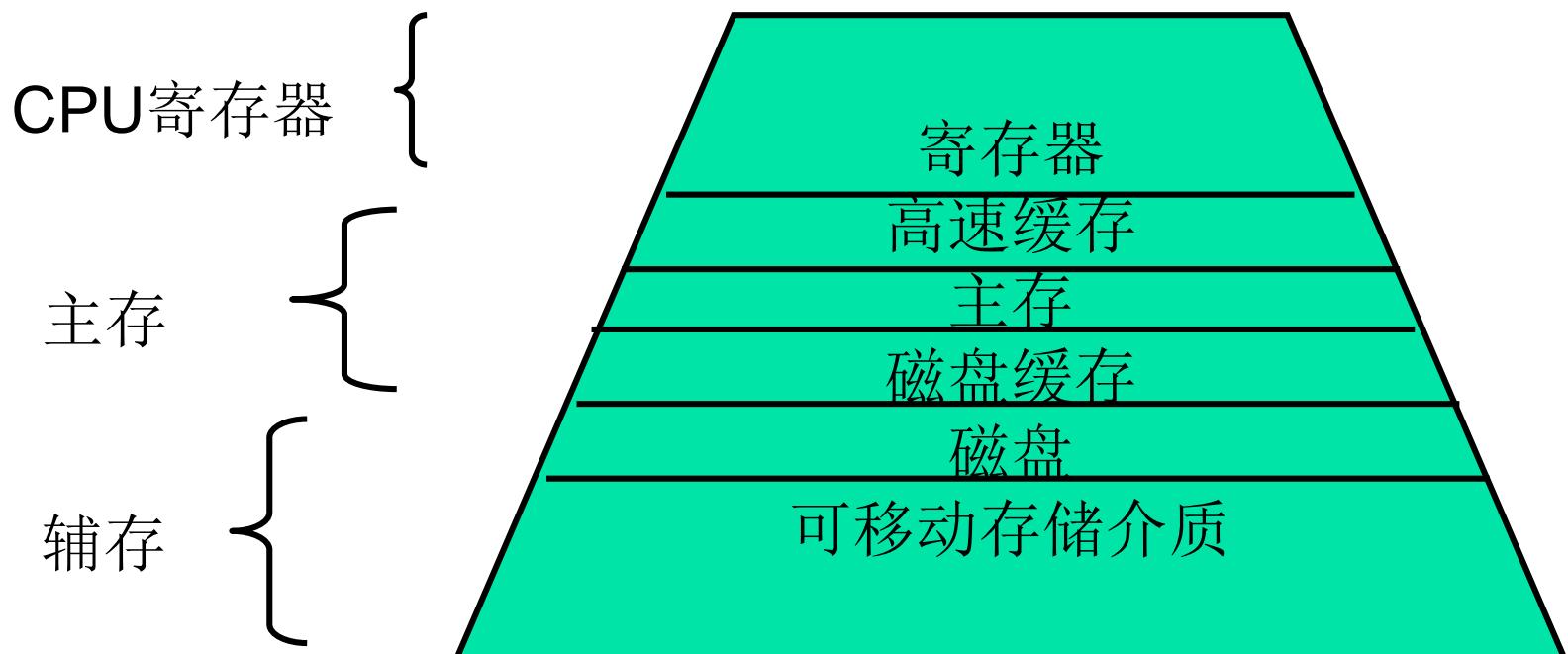
4.4 对换 (Swapping)

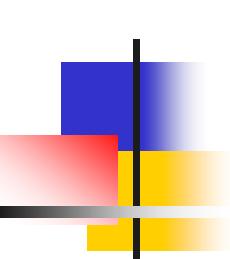
4.5 分页存储管理方式

4.6 分段存储管理方式

# 存储器的层次结构

- 多级存储器结构





# 存储器的层次结构

- 主存储器与寄存器

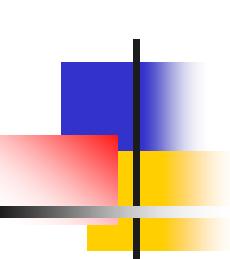
- 主存储器：可执行存储器

- 寄存器：访问速度最快。

- 高速缓存和磁盘缓存

- 高速缓存：访问速度快于主存储器

- 磁盘缓存：利用主存中的存储空间。



# 第四章 存储器管理

4.1 存储器的层次结构

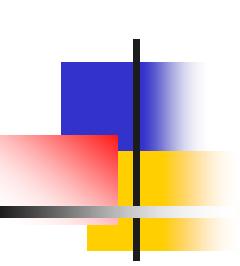
4.2 程序的装入和链接

4.3 连续分配存储器管理方式

4.4 对换 (Swapping)

4.5 分页存储管理方式

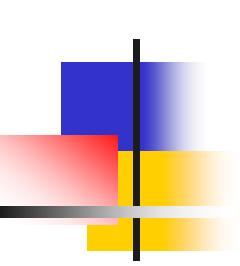
4.6 分段存储管理方式



## 4.2 程序的装入和链接

### 4.2.1 程序的装入

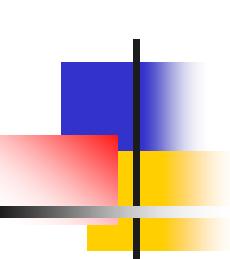
### 4.2.2 程序的链接



## 4.2 程序的装入和链接

### 4.2.1 程序的装入

### 4.2.2 程序的链接



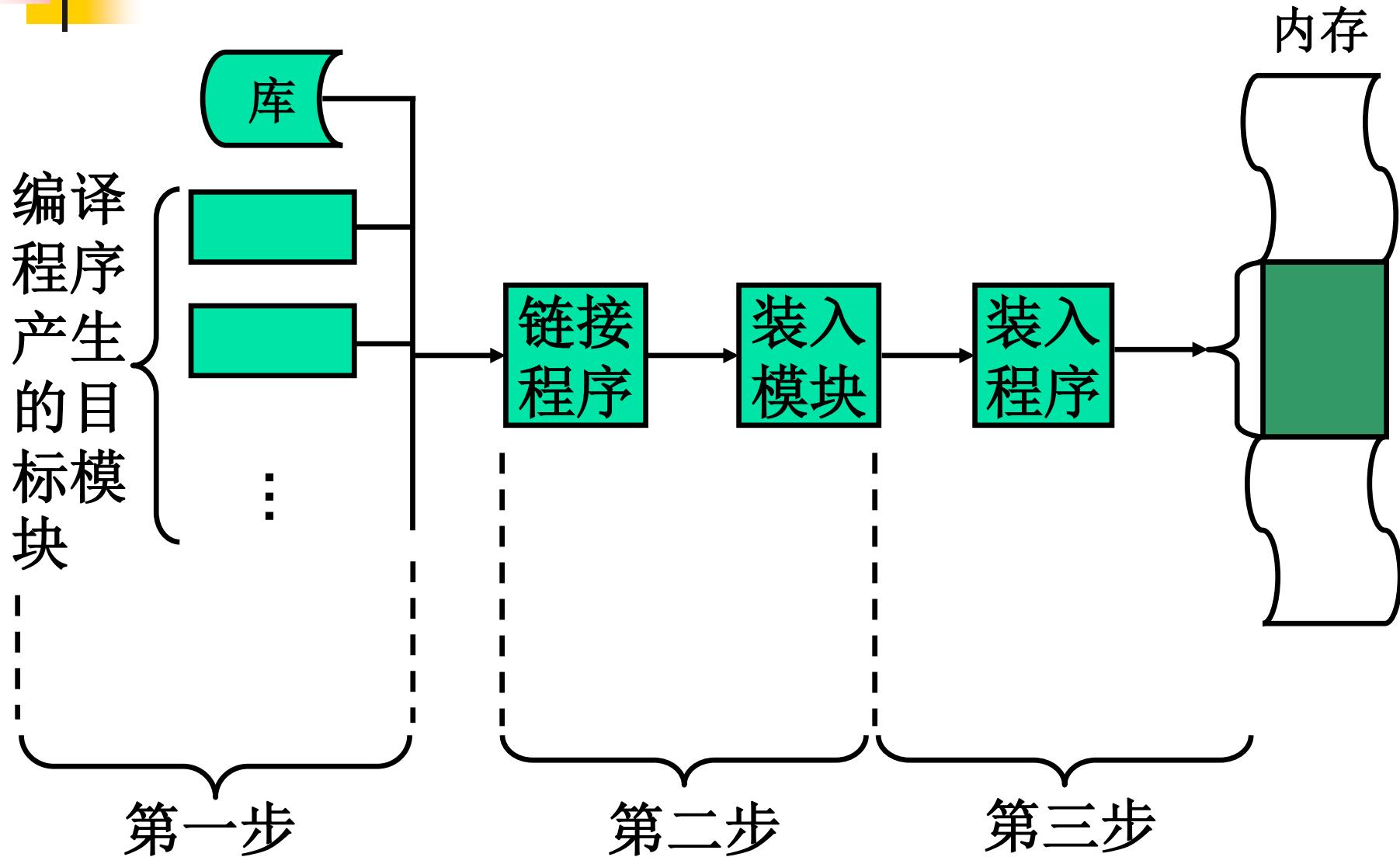
## 4.2 程序的装入和链接

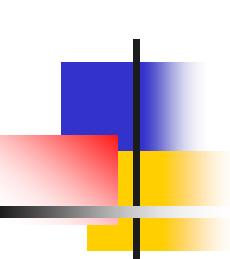
在多道程序环境下，要使程序运行，必须为之先建立进创建进程的第一件事是将程序和数据装入内存。

将用户源程序变为可在内存中执行的程序的步骤：

- **编译**：由编译程序将用户源代码编译成若干个目标模块
- **链接**：由链接程序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
- **装入**：由装入程序将装入模块装入内存

## 4.2 程序的装入和链接



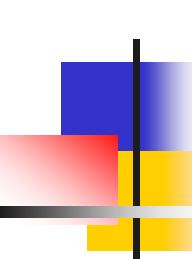


## 4.2 程序的装入和链接

### 4.2.1 程序的装入

将一个装入模块装入内存时，有三种方式：

- 绝对装入方式
- 可重定位装入方式
- 动态运行时装入方式



## 4.2 程序的装入和链接

### 4.2.1 程序的装入

#### 1. 绝对装入方式

在编译时，如果知道程序驻留在内存的什么位置，那么编译程序将产生绝对地址的目标代码。

装入模块装入内存后，程序中的逻辑地址与实际内存地址完全相同，不须对程序和数据的地址进行修改。

程序中所使用的绝对地址，可在编译或汇编时给出，也可由程序员赋予。通常在程序中采用符号地址，然后在编译或汇编时，再将这些符号地址转换为绝对地址。

# 4.2 程序的装入和链接

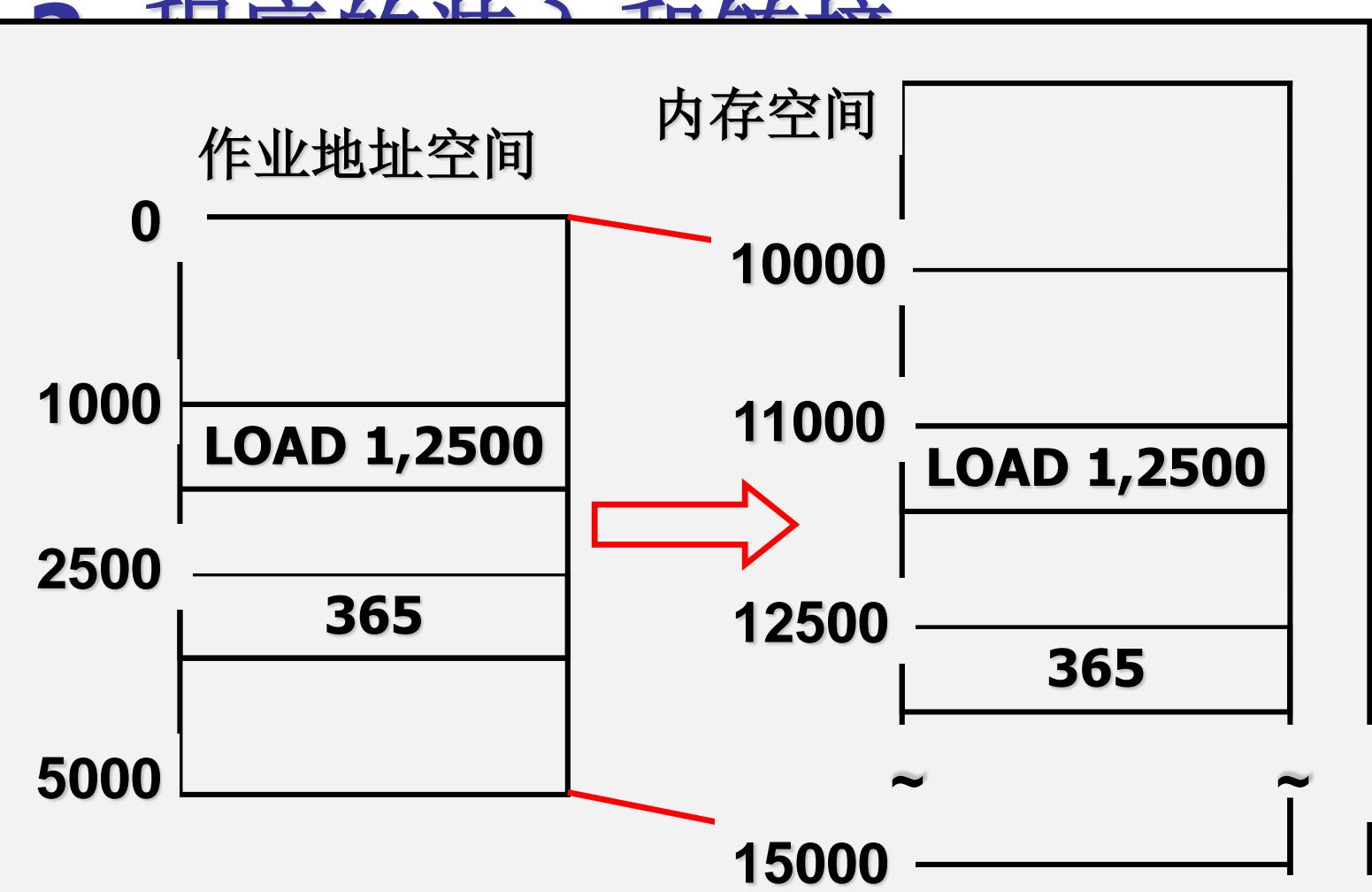
## 4.2.1 程序的装入

1. 绝对装入方式
2. 可重定位装入方式

绝对装入方式只能将目标模块装入到内存中事先指定的位置。在多道程序环境下，编译程序不能预知所编译的目标模块应放在内存何处，因此绝对装入方式只适用于单道程序环境下。

在多道程序环境下，目标模块的起始地址通常从0开始，程序中的其他地址都是相对于起始地址计算的。因此应采用可重定位装入方式，根据内存的当前情况，将装入模块装入到内存的适当位置。

后，会  
地址不



在装入时对目标程序中指令  
和数据的修改过程称为重定位。  
地址变换在装入时一次完成，以  
后不再改变，称为静态重定位。

# 4.2 程序的装入和链接

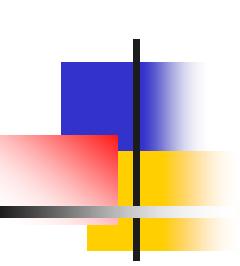
## 4.2.1 程序的装入

1. 绝对装入方式
2. 可重定位装入方式
3. 动态运行时装入方式

可重定位装入方式，可将装入模块装入到内存中任何允许的位置，故可用于多道程序环境；但并不允许程序运行时在内存中移动位置。

实际上，在运行过程中程序在内存中的位置可能经常要改变。

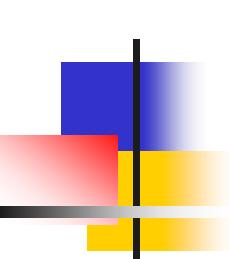
动态运行时的装入程序，在把装入模块装入内存后，并不立即把装入模块中的相对地址转换为绝对地址，而是把这种地址转换推迟到程序真正要执行时才进行。因此，装入内存后的所有地址都仍是相对地址。为使地址转换不影响指令的执行速度，应设置一个重定位寄存器。



## 4.2 程序的装入和链接

### 4.2.1 程序的装入

### 4.2.2 程序的链接



## 4.2 程序的装入和链接

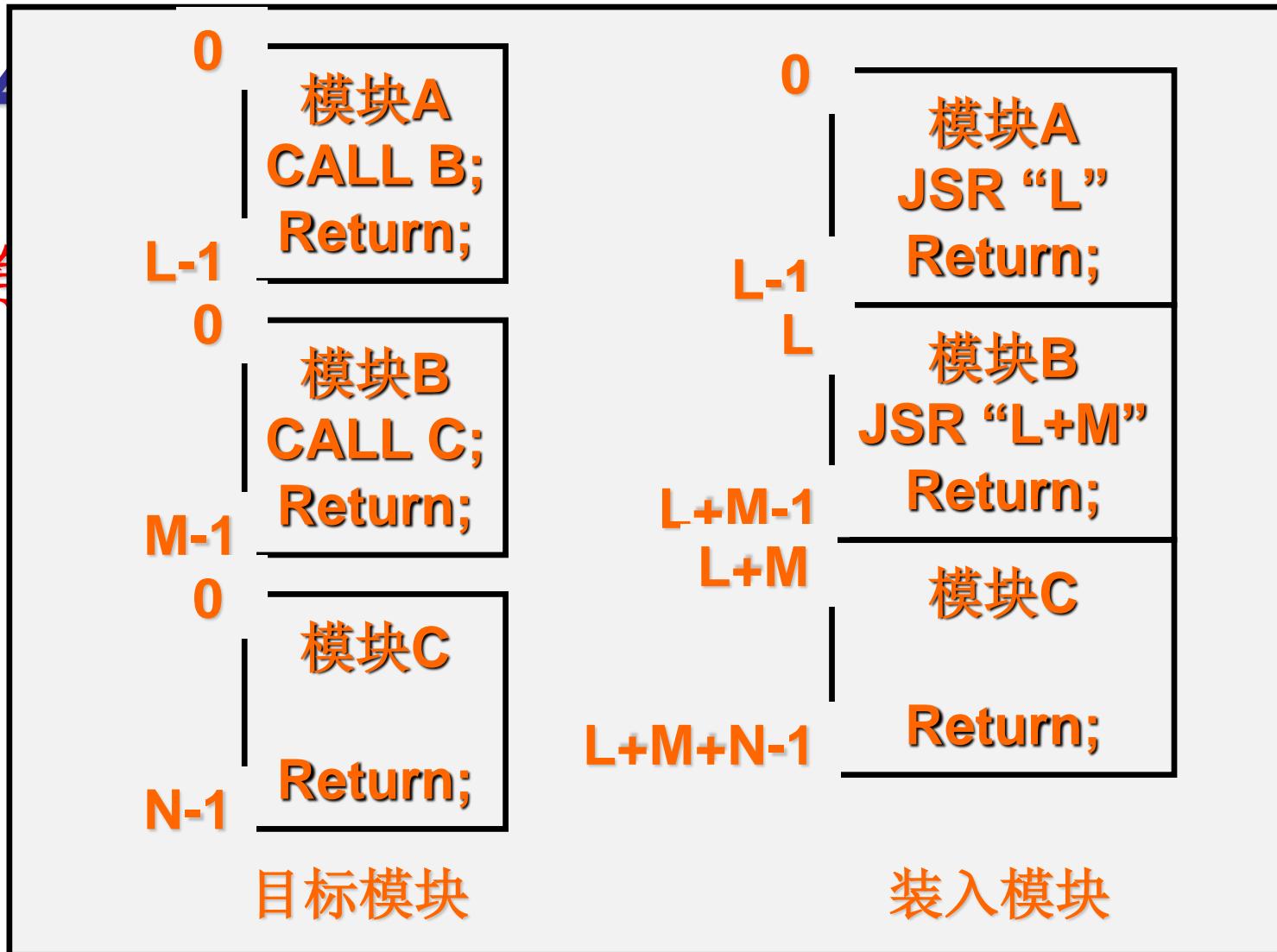
### 4.2.2 程序的链接

程序经过编译后得到一组目标模块，再利用链接程序将目标模块链接，形成装入模块。

根据链接时间的不同，把链接分成三种：

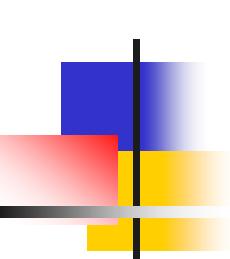
- **静态链接**：在程序运行前，将目标模块及所需的库函数链接成一个完整的装配模块，以后不再拆开。
- **装入时动态链接**：指将用户源程序编译后所得的一组目标模块，在装入内存时，采用边装入边链接的链接方式。
- **运行时动态链接**：指对某些目标模块的链接，是在程序执行中需要该目标模块时，才对它进行链接。

## 1. 链



将目标模块装配成装入模块时需解决的两个问题：

- (1) 对相对地址进行修改
- (2) 变换外部调用符号



## 4.2.2 程序的链接

1. 静态链接

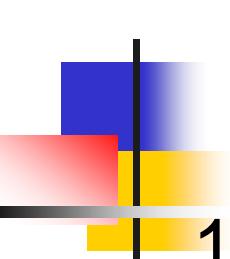
2. 装入时动态链接

用户源程序经编译后所得的目标模块，是在装入内存时，边装入边链接的，即在装入一个目标模块时，若发生一个外部模块调用事件，将引起装入程序去找出相应的外部目标模块，并将它装入内存，还要修改目标模块中的相对地址。

优点：

便于修改和更新

便于实现对目标模块的共享

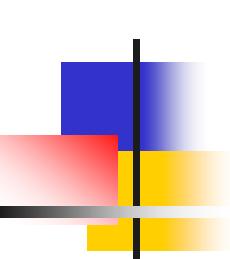


## 4.2.2 程序的链接

1. 静态链接
2. 装入时动态链接
3. 运行时动态链接

在许多情况下，应用程序要运行的模块可能不同，事先不知道要运行哪些模块，只能全部装入，装入时全部链接在一起，效率低。

运行时动态链接是将对某些模块的链接推迟到执行时才执行，即在执行过程中，当发现一个被调用模块尚未装入内存时，立即由OS去找到该模块并将之装入内存，把它链接到调用者模块上。凡执行过程中未被用到的目标模块，不会调入内存和链接，这样不仅加快程序的装入过程，而且节省大量的内存空间。



# 第四章 存储器管理

4.1 存储器的层次结构

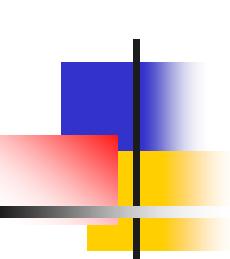
4.2 程序的装入和链接

4.3 连续分配存储器管理方式

4.4 对换 (Swapping)

4.5 分页存储管理方式

4.6 分段存储管理方式

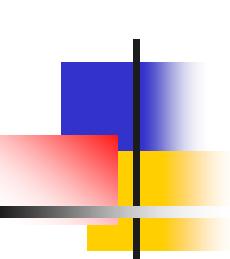


## 4.3 连续分配方式

连续分配方式，是指为一个用户程序分配一个连续的内存空间。

### 分类：

- 单一连续分配
- 固定分区分配
- 动态分区分配
- 可重定位分区分配



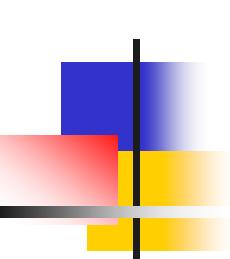
## 4.3 连续分配方式

4.3.1 单一连续分配

4.3.2 固定分区分配

4.3.3 动态分区分配

4.3.6 可重定位分区分配



## 4.3 连续分配方式

4.3.1 单一连续分配

4.3.2 固定分区分配

4.3.3 动态分区分配

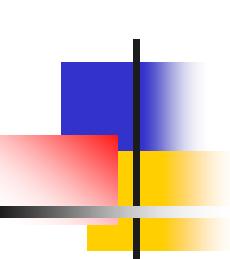
4.3.6 可重定位分区分配

## 4.3.1 单一连续分配方式

最简单的一种存储管理方式，但只能用于单用户、单任务的操作系统中。

采用这种存储管理方式时，可把内存分为系统区和用户区两部分，系统区仅提供给OS使用，通常放在内存低址部分，用户区是指除系统区以外的全部内存空间，提供给用户使用。





## 4.3 连续分配方式

4.3.1 单一连续分配

4.3.2 固定分区分配

4.3.3 动态分区分配

4.3.6 可重定位分区分配

## 4.3.2 固定分区分配

### 1. 原理

将内存用户空间划分为若干个固定大小的区域，在每个分区中只装入一道作业，这样把用户空间划分为几个分区，便允许有几道作业并发执行。当有一空闲分区时，便可以再从外存的后备作业队列中，选择一个适当大小的作业装入该分区，当该作业结束时，可再从后备作业队列中找出另一作业调入该分区。

### 2. 划分分区的方法

可用两种方法将内存的用户空间划分为若干个固定大小的分区：

- (1) **分区大小相等**：缺乏灵活性，用于一台计算机控制多个相同对象的场合
- (2) **分区大小不等**：把内存区划分成含有多个较小的分区、适量的中等分区及少量的大分区，可根据程序的大小为之分配适当的分区。

## 4.3.2 固定分区分配

### 3. 实现

为便于内存分配，通常将分区按大小进行排队，并为之建立一张分区使用表，其中各表项包括每个分区的起始地址、大

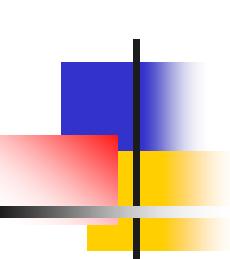
小  
中  
序  
够

分区号	大小(K)	起址(K)	状态
1	12	20	已分配
2	32	32	已分配
3	64	64	已分配
4	128	128	未分配

分区说明表



存储空间分配情况



## 4.3 连续分配方式

4.3.1 单一连续分配

4.3.2 固定分区分配

4.3.3 动态分区分配

4.3.6 可重定位分区分配

## 4.3.3 动态分区分配

### 1. 原理

动态分区分配是根据进程的实际需要，动态地为之分配内存空间。作业装入内存时，把可用内存分出一个连续区域给作业，且分区的大小正好适合作业大小的需要。分区的大小和个数依装入作业的需要而定。

### 2. 实现

在实现过程中涉及如下问题：

- 分区分配中的数据结构
- 分区分配算法
- 分区分配及回收操作

## 4.3.3 动态分区分配

### 2. 实现

#### 1) 分区分配中的数据结构

##### (1) 空闲分区表示

- 空闲分区表：记录每个空闲分区的情况。每个空闲分区占一个表目。  
表目中包括：分区序号、分区始址、分区的大小等。
- 空闲分区链：在每个分区的起始部分，设置一些用于控制分区分配的信息，以及用于链接各分区所用的前向指针；在分区尾部则设置一后向指针，在分区末尾重复设置状态位和分区大小表目。

##### (2) 已占分区说明表

结构：作业号；起始地址；大小

序号	大小	起址	状态
1	48k	116k	可用
2	252k	260k	可用

(a) 空闲分区表



或者



(b) 空闲链结构

作业号	大小	起址
1	32k	20k
2	64k	52k
4	96K	164K

(C) 已占分区表

### 4.3.3 动态分区分配 之例

0k:  
20k:

操作系统
作业1 (32k)

52k:

作业2 (64k)
-----------

116k:

空闲区 (48k)
-----------

164k:

作业4 (96k)
-----------

260k:

空闲区 (252k)
------------

512k:

(d) 内存分配图

## 4.3.3 动态分区分配

### 2. 实现

#### 2) 分区分配算法

为把一个新作业装入内存，需按照一定的分配算法，从空闲分区表或空闲分区链中选出一分区分配给该作业。

常用的分配算法：

- (1) 首次适应算法FF
- (2) 循环首次适应算法
- (3) 最佳适应算法

## 4.3.3 动态分区分配

### 2. 实现

#### 2) 分区分配算法

##### (1) 首次适应算法FF

FF算法要求空闲分区表以地址递增的次序排列。在分配内存时，从表首开始顺序查找，直至找到一个大小能满足要求的空闲分区为止；然后按照作业的大小，从该分区中划出一块内存空间分配给请求者，余下的空闲分区仍留在空闲分区表中。若从头到尾不存在满足要求的分区，则分配失败。

**优点：**优先利用内存低址部分的内存空间

**缺点：**低址部分不断划分，产生小碎片（内存碎块、内存碎片、零头）；每次查找从低址部分开始，增加了查找的开销

## 4.3.3 动态分区分配

### 2. 实现

#### 2) 分区分配算法

##### (2) 循环首次适应算法

在分配内存空间时，从上次找到的空闲分区的下一个空闲分区开始查找，直到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作业。

为实现算法，需要：

- 设置一起始查寻指针
- 采用循环查找方式

**优点：**使内存空闲分区分布均匀，减少查找的开销

**缺点：**缺乏大的空闲分区

## 4.3.3 动态分区分配

### 2. 实现

2) 分区分配算法

(3) 最佳适应算法

所谓“最佳”是指每次为作业分配内存时，总是把能满足要求、又是最小的空闲分区分配给作业，避免“大材小用”。

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。

缺点：产生许多难以利用的小空闲区

## 4.3.3 动态分区分配

### 2. 实现

#### 2) 分区分配算法

为把一个新作业装入内存，需按照一定的分配算法，从空闲分区表或空闲分区链中选出一分区分配给该作业。

常用的分配算法：

- (1) 首次适应算法FF
- (2) 循环首次适应算法
- (3) 最佳适应算法

## 4.3.3 动态分区分配

### 2. 实现

#### 3) 分区分配及回收操作

##### ■ 分配内存

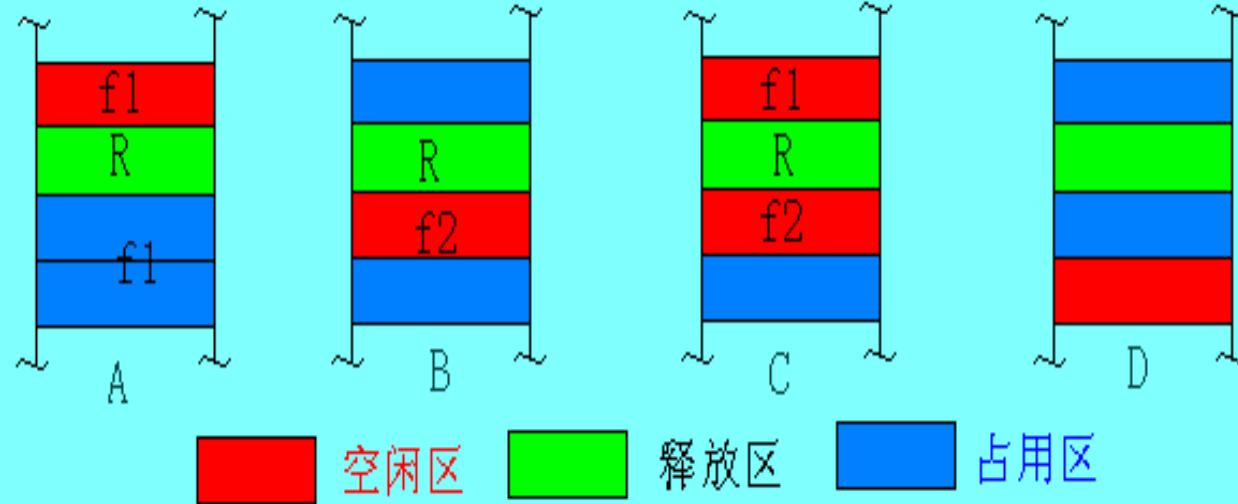
利用某种分配算法，从空闲分区链(表)中找到所需大小的分区。设请求的分区大小为 $u.size$ ，表中每个空闲分区的大小表示为 $m.size$ ，若 $m.size - u.size \leq size$ (规定的不再切割的分区大小)，将整个分区分配给请求者，否则从分区中按请求的大小划出一块内存空间分配出去，余下部分留在空闲链中，将分配区首址返回给调用者。

## 2. 实现

### 3) 内存回收

#### 回收内存

#### 闲分区



- (1) 回收区与插入点的前一个分区F1邻接：将回收区与F1合并，修改F1的表项的分区大小
- (2) 回收区与插入点的后一个分区F2邻接：将回收区与F2合并，修改F2的表项的首址、分区大小
- (3) 回收区与插入点的前后两个分区F1、F2邻接：将三个分区合并，使用F1的表项和F1的首址，取消F2的表项，大小为三者之和
- (4) 回收区既不与F1邻接，又不与F2邻接：为回收区单独建立新表项，填写回收区的首址与大小，根据其首址插到空闲链中的适当位置

## 4.3.3 动态分区分配

### 1. 原理：

动态分区分配是根据进程的实际需要，动态地为之分配内存空间。作业装入内存时，把可用内存分出一个连续区域给作业，且分区的大小正好适合作业大小的需要。分区的大小和个数依装入作业的需要而定。

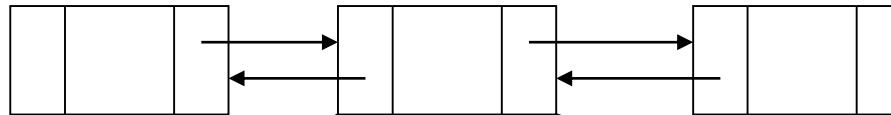
### 2. 实现

在实现过程中涉及如下问题：

- 分区分配中的数据结构
- 分区分配算法
- 分区分配及回收操作

序号	大小	起址	状态
1	48k	116k	可用
2	252k	260k	可用

(a) 空闲分区表



或者



(b) 空闲链结构

作业号	大小	起址
1	32k	20k
2	64k	52k
4	96K	164K

(C) 已占分区表

### 4.3.3 动态分区分配 之例

0k:  
20k:

操作系统
作业1 (32k)

52k:

作业2 (64k)

116k:

空闲区 (48k)

164k:

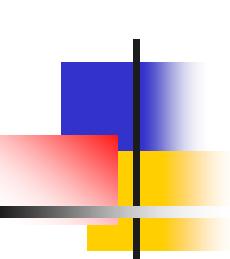
作业4 (96k)

260k:

空闲区 (252k)

512k:

(d) 内存分配图



## 4.3 连续分配方式

4.3.1 单一连续分配

4.3.2 固定分区分配

4.3.3 动态分区分配

4.3.6 可重定位分区分配

## 4.3.6 可重定位分区分配

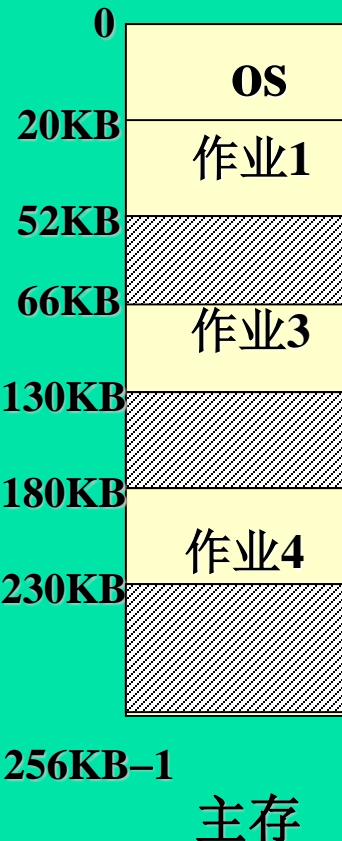
### 1. 动态重定位

在连续分区分配中，作业的物理地址是不变的。如果作业的逻辑地址空间大于要装入的物理地址空间，则不能装入内存。

解决方法：

这样可把原来“邻接”或“分离”的作业“拼接”。

缺点：用

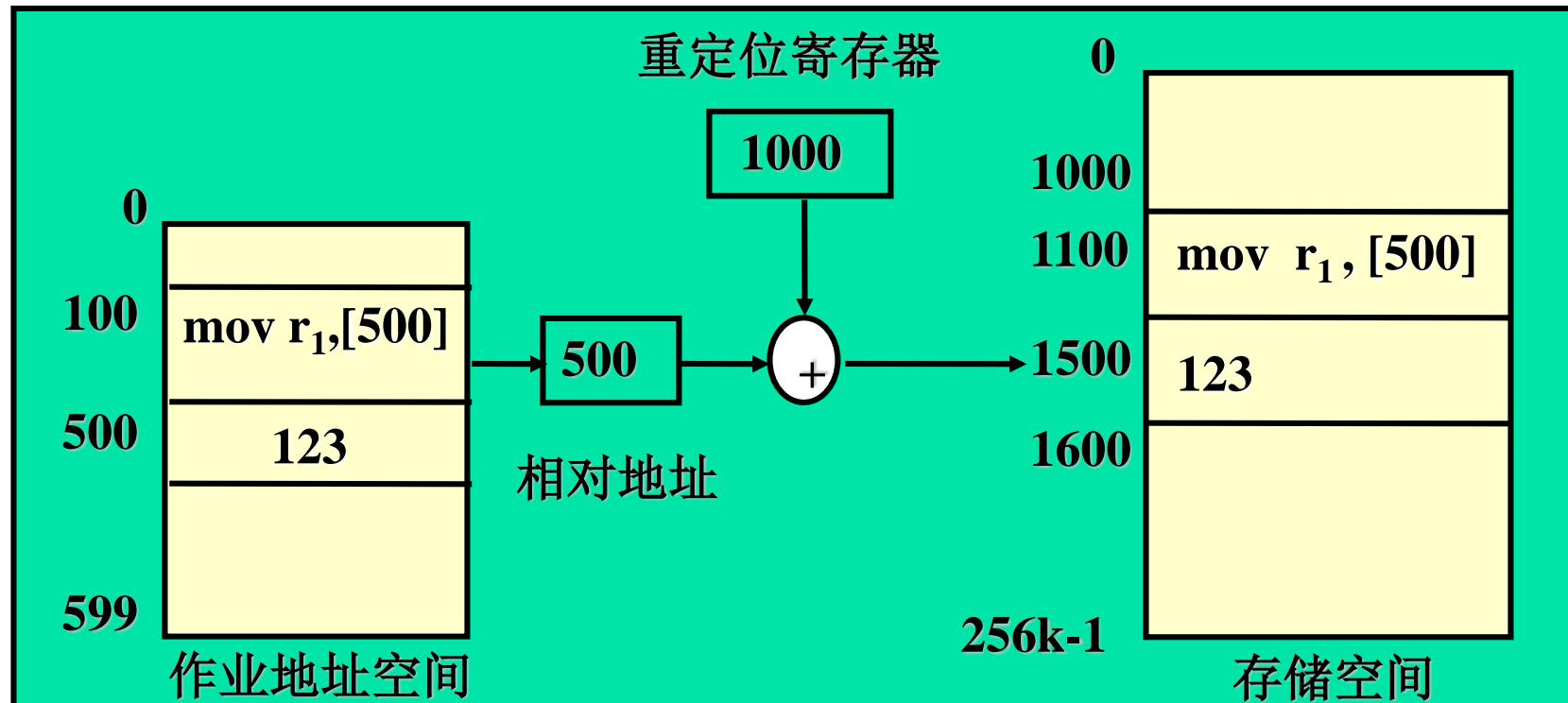


连续的内存空间总和大于程序装入内存。  
全部邻接，“拼接”  
重定位。

## 4.3.6 可重定位分区分配

### 2. 动态重定位的实现

在动态运行时装入的方式时，将相对地址转换为物理地址的工作在程序指令真正要执行时才进行。地址转换需要重定位寄存器的支持。程序执行时访问的内存地址是相对地址与重定位寄存器中的地址相加而成。

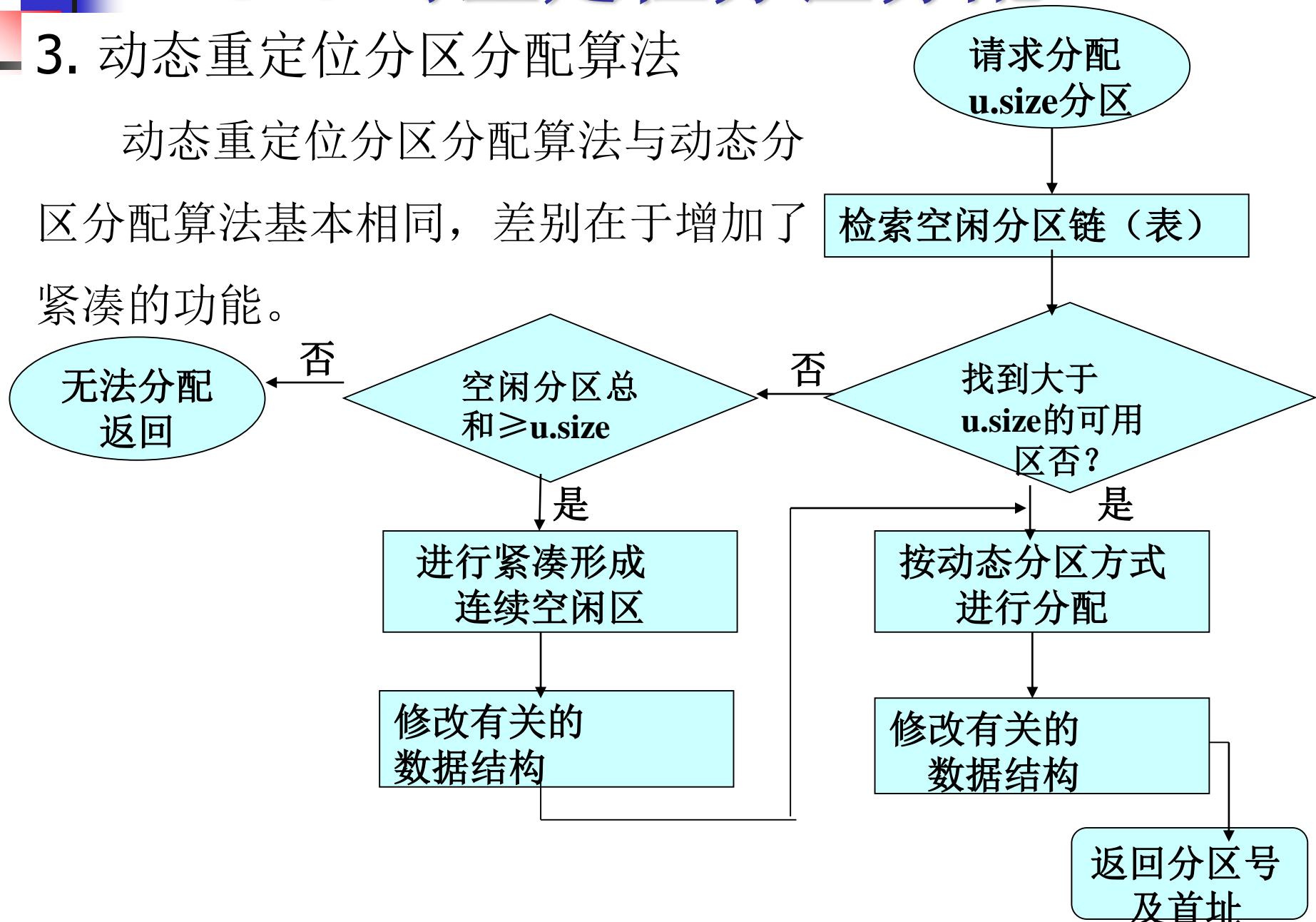


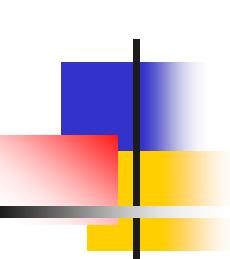
地址变换过程是在程序执行过程期间，随着对每条指令的访问自动进行的，称为**动态重定位**。

## 4.3.6 可重定位分区分配

### 3. 动态重定位分区分配算法

动态重定位分区分配算法与动态分区分配算法基本相同，差别在于增加了紧凑的功能。





# 第四章 存储器管理

4.1 存储器的层次结构

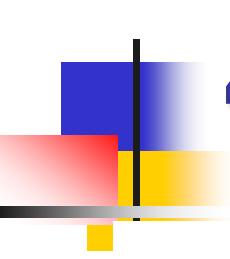
4.2 程序的装入和链接

4.3 连续分配存储器管理方式

4.4 对换（Swapping）

4.5 分页存储管理方式

4.6 分段存储管理方式



## 4. 4 对换

1. 对换的引入
2. 对换空间的管理
3. 进程的换出与换入

# 4.4 对换

## 1. 对换的引入

多道程序环境下存在的问题：

- 阻塞进程占据大量内存空间
- 许多作业在外存而不能进入内存运行

**对换**：把内存中暂时不能运行的进程或者暂时不用的程序和数据，调到外存上，以便腾出足够的内存空间，再把已具备运行条件的进程和进程所需要的程序和数据，调入内存。

对换的分类：

- 整体对换(或进程对换)：以整个进程为单位
- 页面对换或分段对换：以页或段为单位

实现进程对换，系统必须具备的功能：

- 对换空间的管理
- 进程的换出
- 进程的换入

## 4.4 对换

### 2. 对换空间的管理

一般从磁盘上划出一块空间作为对换区使用

	存储内容	驻留时间	主要目标	分配方式
文件区	文件	较长久	提高文件 存储空间 的利用率	离散
对换区	从内存换出 的进程	短暂	提高进程 换入和换 出的速度	连续

在系统中设置相应的数据结构以记录外存的使用情况

对换空间的分配与回收，与动态分区方式时的内存分配与回收雷同。

## 4.4 对换

### 3. 进程的换出与换入

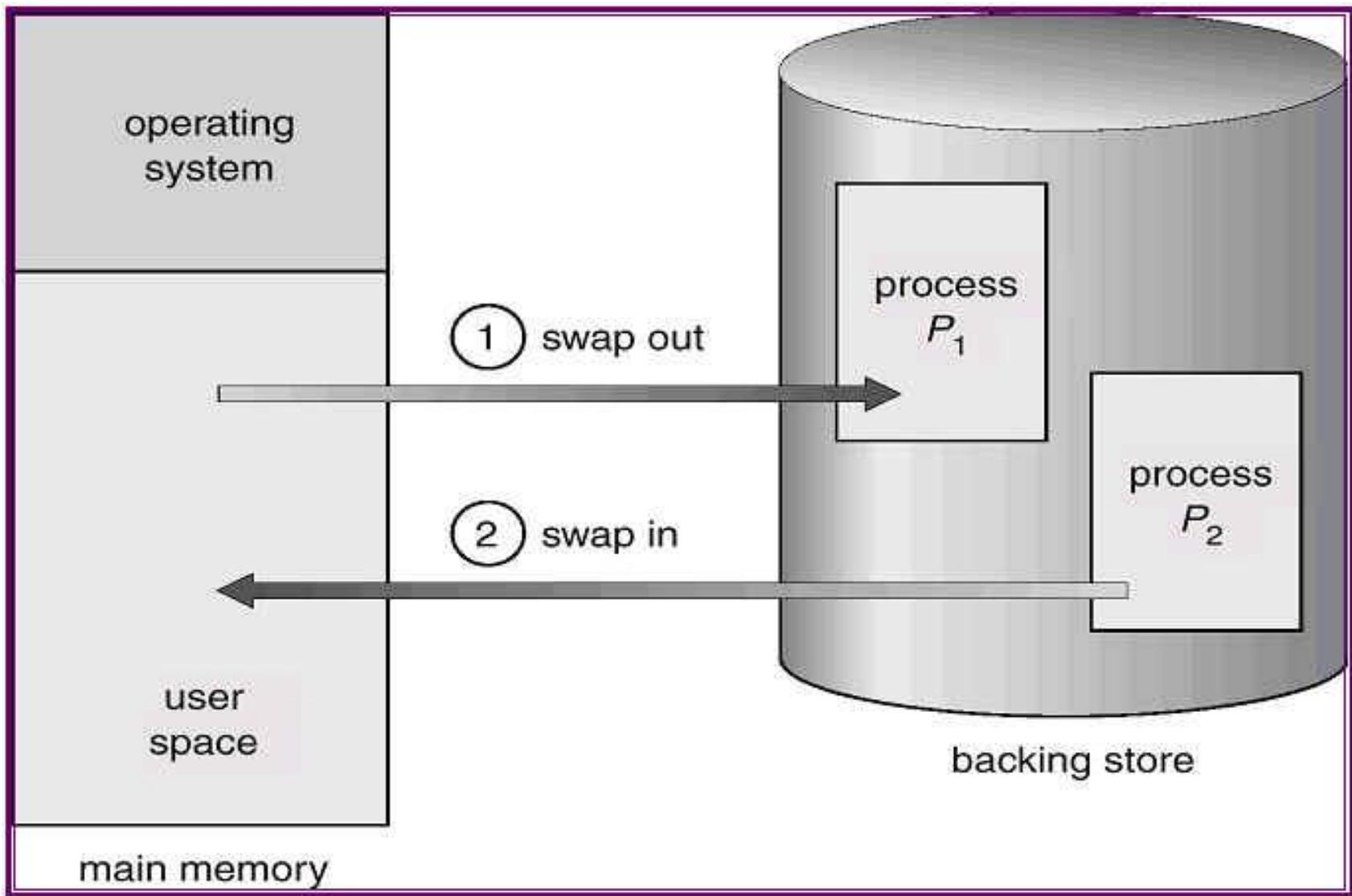
#### ■ 进程的换出

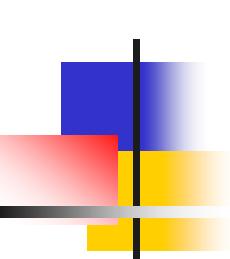
**换出过程：**系统首先选择处于阻塞状态且优先级最低的进程作为换出进程，然后启动盘块，将该进程的程序和数据传送到磁盘的对换区上。

#### ■ 进程的换入

**换入过程：**系统应定时查看所有进程的状态，从中找出“就绪”状态但已换出的进程，将换出进程最久的进程作为换入进程，将之换入，直至已无可换入的进程或无可换出的进程为止。

## 4.4 对换





# 第四章 存储器管理

4.1 存储器的层次结构

4.2 程序的装入和链接

4.3 连续分配存储器管理方式

4.4 对换 (Swapping)

4.5 分页存储管理方式

4.6 分段存储管理方式

# 4.5 分页存储管理方式

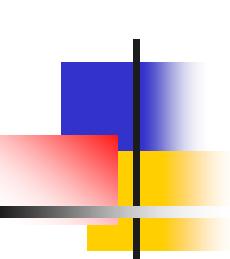
连续分配方式会形成“碎片”，虽然可以通过“紧凑”解决，但开销大。如果允许将一个进程直接分散地装入许多不相邻的分区中，则无需“紧凑”，由此产生离散分配方式。

**分类：**

**分页存储管理方式：**离散分配的基本单位是页

**分段存储管理方式：**离散分配的基本单位是段

基本的分页存储管理方式(或纯分页存储管理方式)：不具备页面对换功能，不具有支持实现虚拟存储器的功能，要求把每个作业全部装入内存后方能运行。

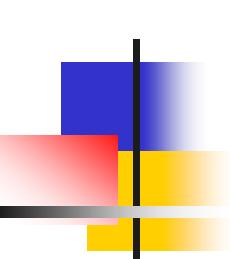


## 4.5 分页存储管理方式

4.5.1 页面与页表

4.5.2 地址变换机构

4.5.3 两级和多级页表



## 4.5 分页存储管理方式

4.5.1 页面与页表

4.5.2 地址变换机构

4.5.3 两级和多级页表

# 4.5.1 页面与页表

## 1. 页面

### ➤ 分页式存储管理的原理

分页存储管理是将一个进程的逻辑地址空间分成若干个大小相等的片称为页面或页，并为各页加以编号，从0开始。同时把内存空间分成与页面相同大小的若干个存储块，称为块或页框。在为进程分配内存时，以块为单位将进程的若干个页分别装入到多个可以不相邻的物理块中。

进程的最后一页经常装不满而形成“页内碎片”。

### ➤ 基本分页式存储管理（简单页式存储管理）的原理

系统若能满足一个作业所要求的全部块数，此作业才能被装入内存，否则不为它分配任何内存。

### ➤ 请求分页式存储管理的原理

运行一个作业时，并不要求把该作业的全部程序和数据都装入内存，可以只把目前要执行的几页调入内存的空闲块中，其余的仍保存在外存中，以后根据作业运行的需要再调入内存。

## 4.5.1 页面与页表

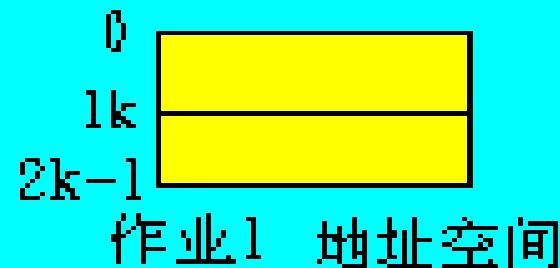
### ➤ 页面大小的选择

- 由机器的地址结构所决定的，即由硬件所决定。
- 某一种机器只能采用一种大小的页面。
- 通常是：几KB到几十KB。
  - 小：内碎片小，内存利用率高，但页面数目多，使页表过长，占大量内存，管理开销大；
  - 大：页表短，管理开销小，内碎片大，内存利用率低

### ➤ 页面大小

页面大小应选地适中，应是2的幂，通常是512B~8KB。

## 4.5.1 页面与页表



# 用户程序

0
1
2
3

# 页表

页号	块号
0	2
1	4
2	5
3	7

# 内存

0
1
2
3
4
5
6
7

0	0
1	1
2	2
3	3

0	---
1	---
2	---

ProcessA

0	7
1	8
2	9
3	10

ProcessC

0	4
1	5
2	6
3	11
4	12

ProcessD

13
14

Free Frame List

## 4.5.1 页面与页表

### 2. 地址结构



地址长度32位:

0~11位为位移量(页内地址), 即每页的大小为4KB

12~31位为页号, 地址空间最多允许有1M页

- 若给定一个逻辑地址空间中的地址为A, 页面大小为L, 则

$$\text{页号 } P = \text{INT}[A/L]$$

$$\text{页内地址 } d = [A] \bmod L$$

例如: 系统页面大小为1KB, 设A=2170D, 则

$$P=2, \quad d=122$$

数值的表示:

二进制,十进制,八进制,十六进制,分别在其后加上B,D,Q,H.

如:十进制的1,用二进制表示为1B,八进制为1Q,十六进制为1H.

## 4.5.1 页面与页表

3. 基

任  
映  
射。  
加以

页表的作用：

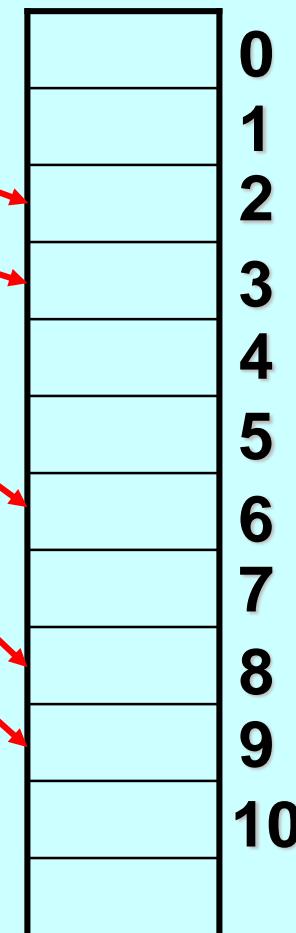
用户程序

0页
1页
2页
3页
4页
5页
...
n页

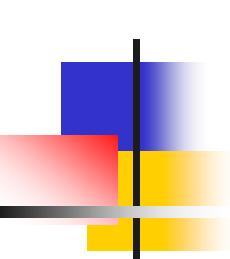
页表  
页号 块号

0	2
1	3
2	6
3	8
4	9
5	
...	...

内存



现  
的  
面  
映  
系



## 4.5 分页存储管理方式

4.5.1 页面与页表

4.5.2 地址变换机构

4.5.3 两级和多级页表

## 4.5.2 地址变换机构

地址变换机构实现从逻辑地址到物理地址的转换，其任务是借助于页表，将逻辑地址中的页号转换为内存中的物理块号。

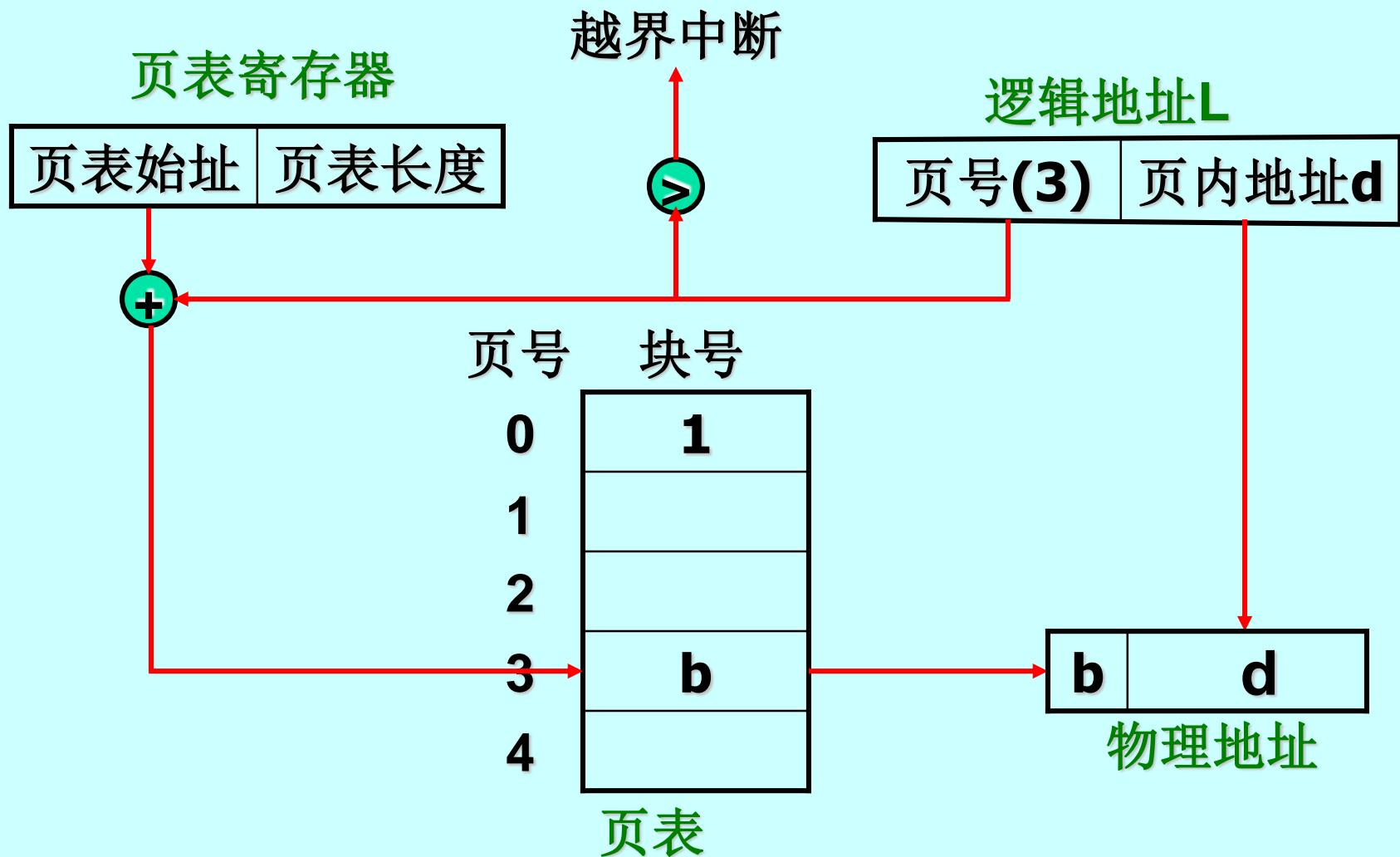
### 1. 基本的地址变换机构

页表的功能可以由一组专门的寄存器来实现，一个页表项用一个寄存器。但寄存器成本高，系统页表可能很大，所以页表大多常驻内存。

在系统中只设置一个页表寄存器PTR，在其中存放页表在内存中的始址和页表的长度。

## 4.5.2 地址变换机构

分页系统的地址变换机构：



## 4.5.2 地址变换机构

### 2. 具有快表的地址变换机构

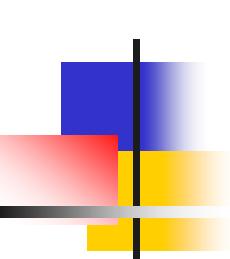
CPU在每存取一个数据时，需要**两次**访问内存：

**第一次：**访问页表，找到指定页的物理块号，将块号与页内偏移量拼接形成物理地址。

**第二次：**从第一次所得地址中获得所需数据，或向此地址中写入数据。

存储器利用率提高，处理器处理速度降低。

**解决方法：**在地址变换机构中，增设一个具有并行查寻能力的特殊高速缓冲寄存器，称为“联想存储器”或“快表”。



## 4.5 分页存储管理方式

4.5.1 页面与页表

4.5.2 地址变换机构

4.5.3 两级和多级页表

## 4.5.3 两级和多级页表

现代计算机系统都支持非常大的逻辑地址空间( $2^{32} \sim 2^{64}$ )，  
页表就非常大，需占用较大的地址空间。

**例如：**一个具有32位逻辑地址空间的分页系统，规定页面大小  
为4KB即 $2^{12}$ B，则每个进程页表的页表项可达1M个，若每个页表  
项占用一个字节，则每个进程的页表就要占据1MB的内存空间，  
而且要求连续存放。

**解决方法：**

- 采用离散方式
- 只将当前所需页表项调入内存

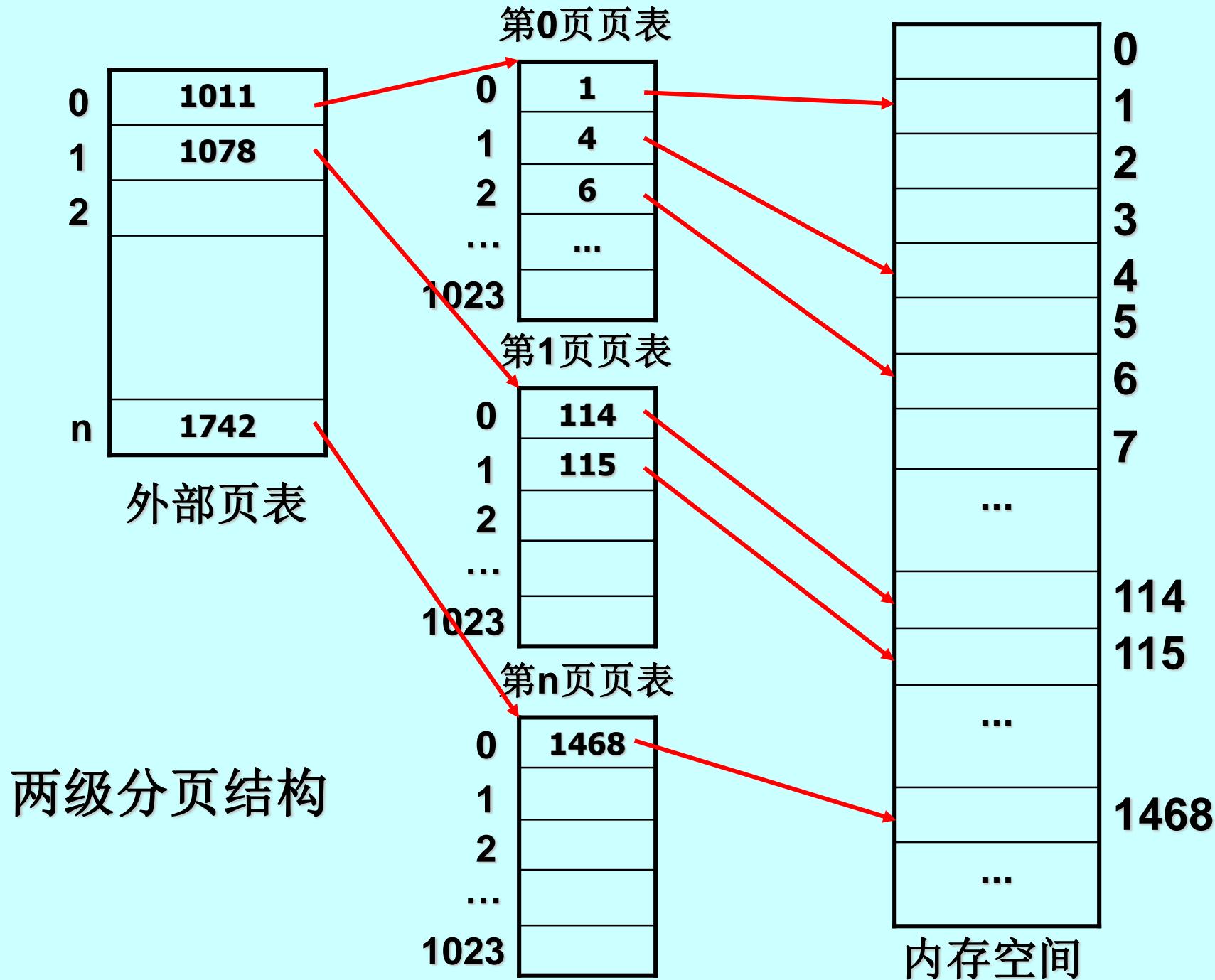
## 4.5.3 两级和多级页表

### 1. 两级页表

将页表分页，并离散地将各个页面分别存放在不同的物理块中，同时为离散分配的页表在建立一张页表，称为外层页表，其每个页表项记录了页表页面的物理块号。

**例如：**32位逻辑地址空间，页面大小为4KB(即12位)，若采用一级页表机构，应有20位页号，即页表项应有1M个；在采用两级页表机构时，再对页表进行分页，使每页包含 $2^{10}$ (即1024)个页表项，最多允许有 $2^{10}$ 个页表分页。即

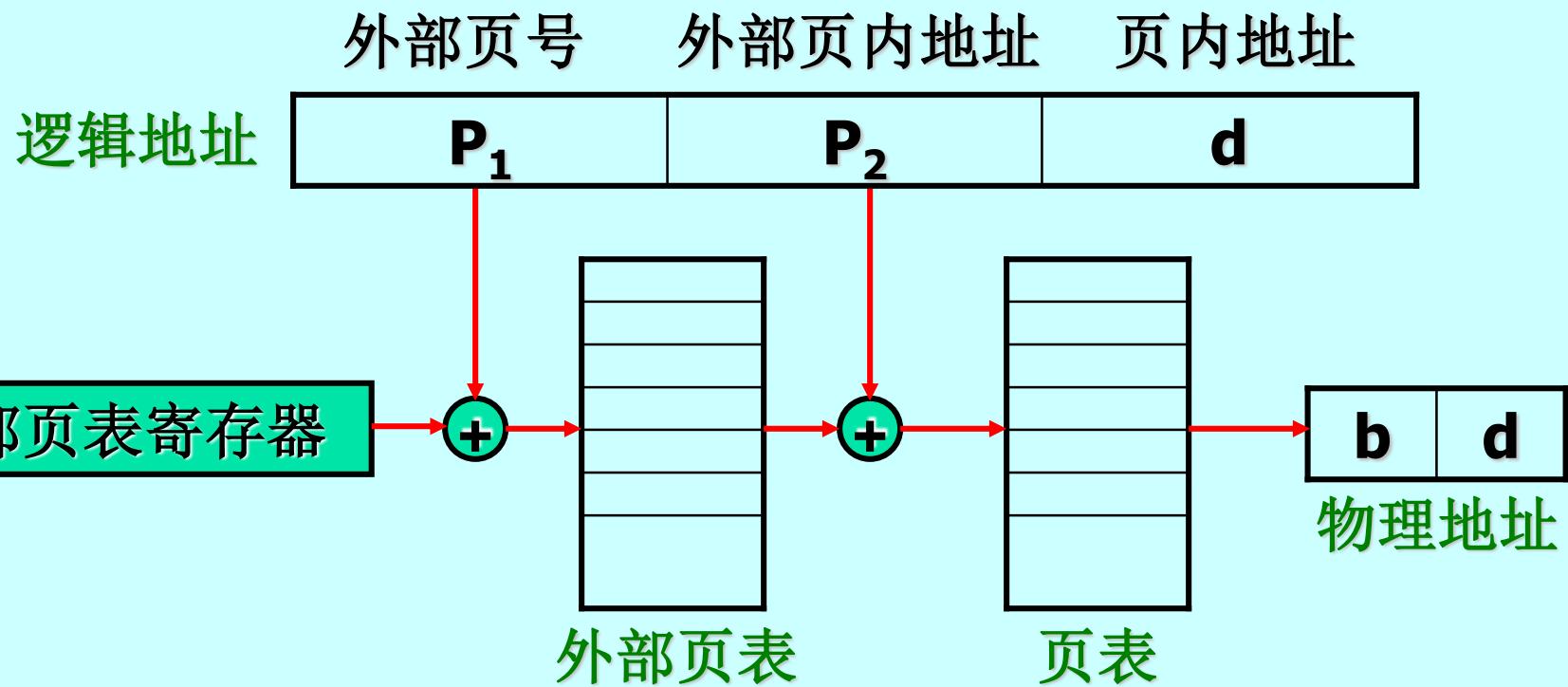




## 4.5.3 两级和多级页表

### 1. 两级页表

具有两级页表的地址变换机构：



## 4.5.3 两级和多级页表

### 1. 两级页表

上述方法用离散分配空间解决了大页表无需大片存储空间的问题，但并未减少页表所占的内存空间。

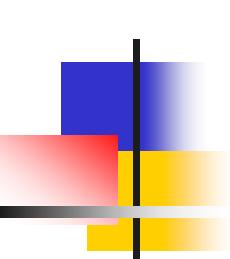
解决方法是把当前需要的一批页表项调入内存，以后再根据需要陆续调入。

### 2. 多级页表

两级页表对32位机器适用，64位呢？

页面大小为4KB即 $2^{12}$ B，还剩52位，按物理块大小 $2^{12}$ 位来划分页表，则剩余40位用于外层页号，此时外层页表可能有1024G个页表项，要占用4096GB的连续存储空间

解决方法：采用多级页表，将外层页表再进行分页。



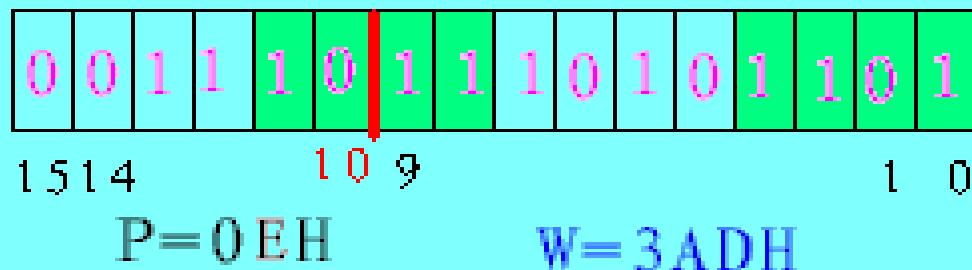
# 关于分页的地址变换计算---虚地址结构

**例1.** 设页面大小为1KB，将逻辑地址3BADH划分为页号和页内偏移量两部分。用16进制表示。

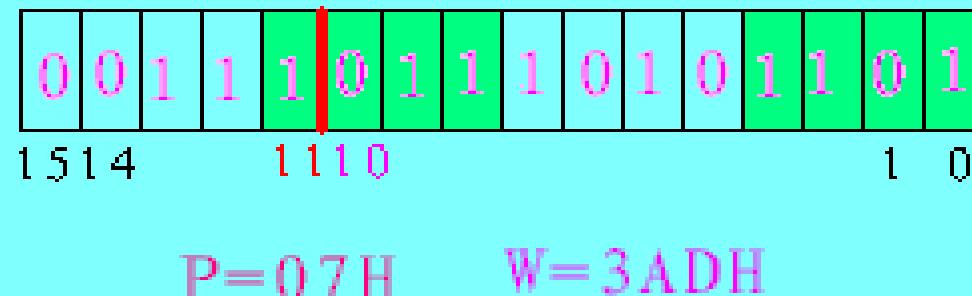
**例2.** 设页面大小为2KB，将逻辑地址3BADH划分为页号和页内偏移量两部分。用16进制表示。

# 关于分页的地址变换计算---虚地址结构

例 1：页面大小是 1KB，虚地址是 3BADH

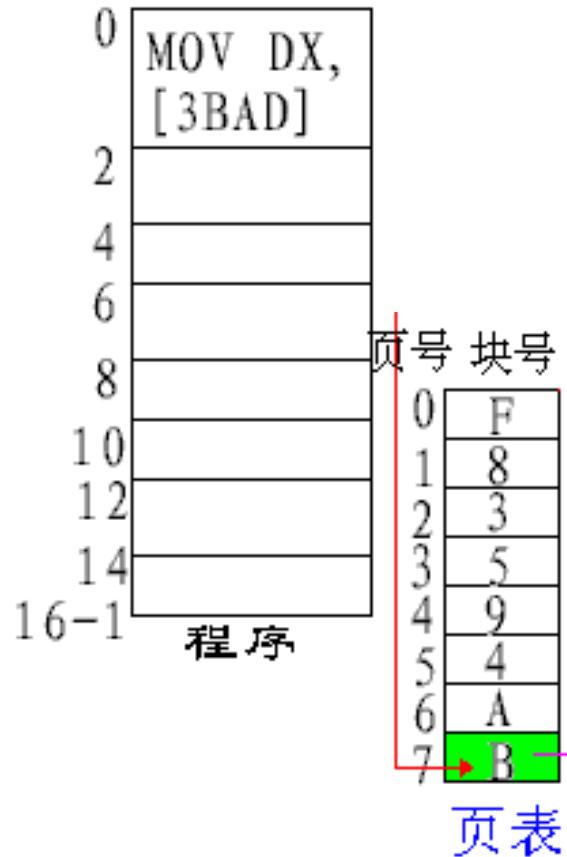


例 2：页面大小是 2KB，虚地址是 3BADH

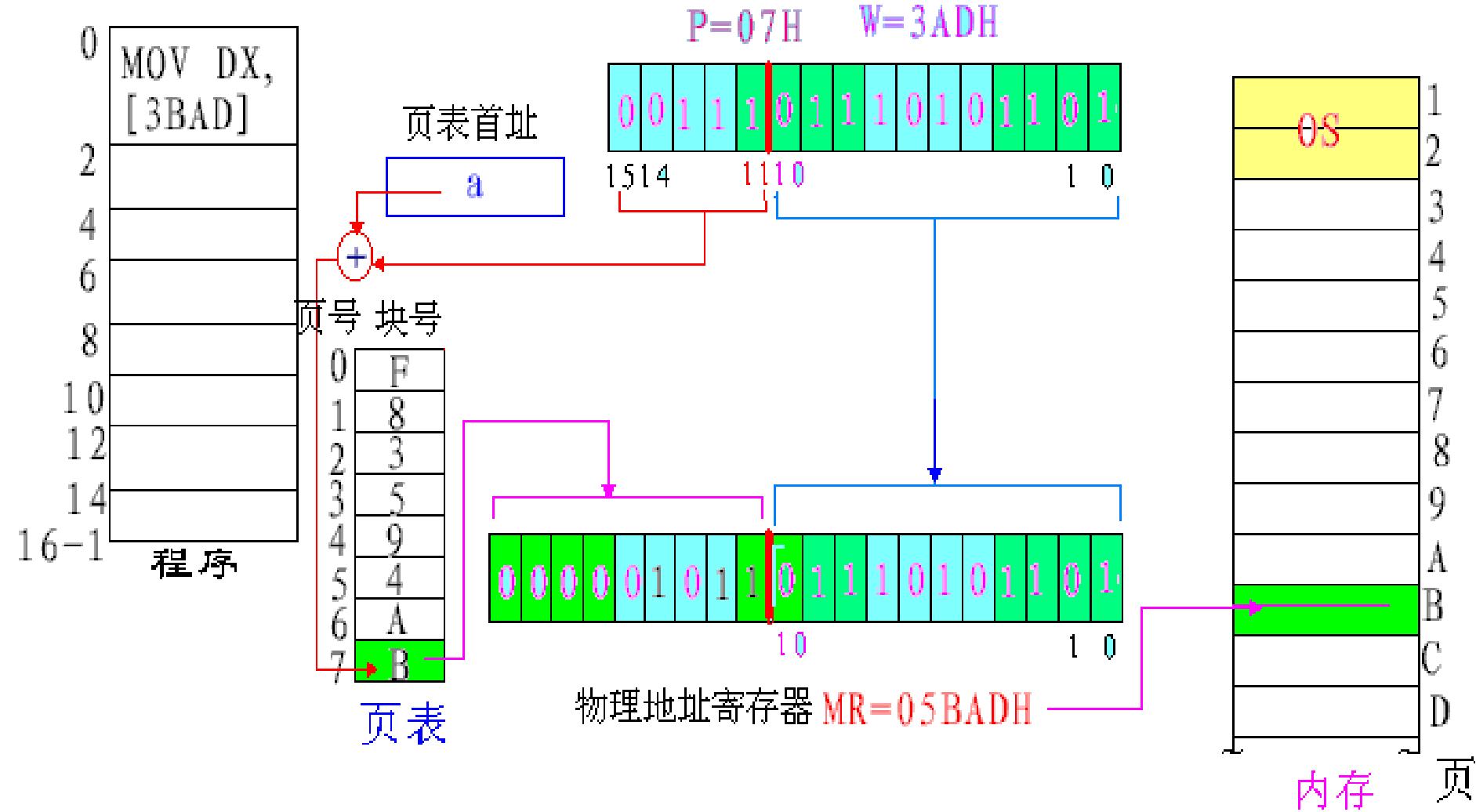


# 关于分页的地址变换计算---虚地址结构

例3. 设页面大小为2KB，作业的页表如下图。求逻辑地址3BADH的物理地址。用16进制表示。



# 关于分页的地址变换计算--- 页式地址映射



# 关于分页的地址变换计算---页式地址映射

**例4：**有一系统采用页式存储管理，有一作业大小是8KB，页大小为2KB，依次装入内存的第7、9、A、5块，试将虚地址0AFEH，1ADDH转换成内存地址。

虚地址0AFEH

0000 1010 1111 1110

P=1 d=010 1111 1110

MR=0100 1010 1111 1110

=4AFEH

虚地址1ADDH

0001 1010 1101 1101

P=3 d=010 1101 1101

MR=0010 1010 1101 1101=2ADDH

页号	块号
0	7
1	9
2	A
3	5

# 关于分页的地址变换计算---页式地址映射

例5：有一系统采用页式存储管理，有一作业大小是8KB，页大小为2KB，依次装入内存的第7、9、10、5块，试将虚地址7145，3412转换成内存地址。

虚地址 **3412**

$$P = 3412 \% 2048 = 1$$

$$d = 3412 \bmod 2048 = 1364$$

$$MR = 9 * 2048 + 1364 = 19796$$

虚地址**3412**的内存地址是：**19796**

虚地址 **7145**

$$P = 7145 \% 2048 = 3$$

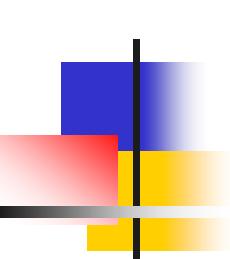
$$d = 7145 \bmod 2048 = 1001$$

$$MR = 5 * 2048 + 1001 = 11241$$

虚地址7145的内存地址是：**11241**

页表：

页号	块号
0	7
1	9
2	10
3	5



# 第四章 存储器管理

4.1 存储器的层次结构

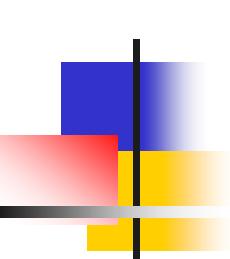
4.2 程序的装入和链接

4.3 连续分配存储器管理方式

4.4 对换 (Swapping)

4.5 分页存储管理方式

4.6 分段存储管理方式



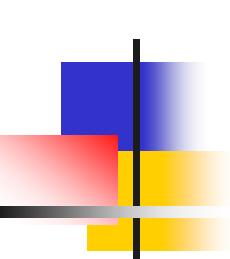
# 4.6 基本分段存储管理方式

4.6.1 分段存储管理方式的引入

4.6.2 分段系统的基本原理

4.6.3 信息共享

4.6.4 段页式存储管理方式



# 4.6 基本分段存储管理方式

4.6.1 分段存储管理方式的引入

4.6.2 分段系统的基本原理

4.6.3 信息共享

4.6.4 段页式存储管理方式

## 4.6.1 分段存储管理方式的引入

分页从根本上克服了外零头（地址空间、物理空间都分割）。

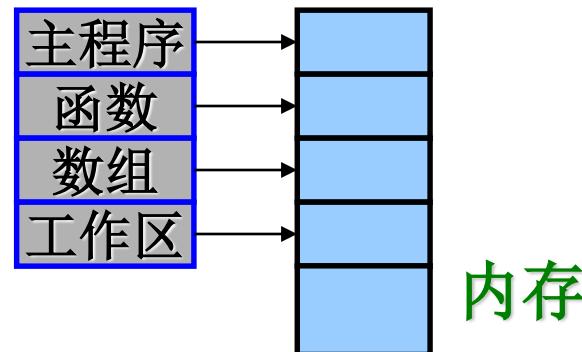


最后一个单元可能未占满，还有内零头，  
平均浪费PL/2大小的空间

内存利用率提高。

**缺点：**无论信息内容如何，按页长分割，分割后装入内存，有可能主程序未能全部进入内存，----执行速度降低。

所以考虑以逻辑单位分配内存。如：



## 4.6.1 分段存储管理方式的引入

- 便于编程:

用户常把自己的作业按逻辑关系划分成若干个段，每段都有自己的名字，且都从零开始编址，这样，用户程序在执行中可用段名和段内地址进行访问。例如：LOAD 1, [A] | <D>。

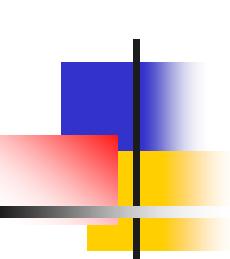
- 分段共享:

在实现程序和数据的共享时，常常以信息的逻辑单位为基础，而分页系统中的每一页只是存放信息的物理单位，其本身没有完整的意义，因而不便于实现信息的共享，而段却是信息的逻辑单位，有利于信息的共享。

- 分段保护: 信息保护是对相对完整意义的逻辑单位(段)进行保护。
- 动态链接:当运行过程中又需要调用某段时，再将该段（目标程序）调入内存并链接起来。所以，动态链接是以段为基础的。

- 动态增长:

在实际系统中，有些数据段会不断地增长，而事先却无法知道数据段会增长到多大，分段存储管理方式能较好地解决这个问题。



# 4.6 基本分段存储管理方式

4.6.1 分段存储管理方式的引入

4.6.2 分段系统的基本原理

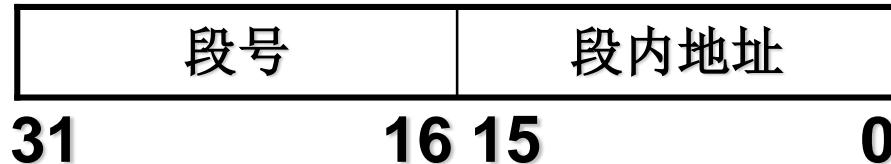
4.6.3 信息共享

4.6.4 段页式存储管理方式

## 4.6.2 分段系统的基本原理

### 1. 分段

在分段存储管理方式中，作业地址空间被划分为若干个段，每个段定义了一组逻辑信息，都有自己的名字。通常用段号代替段名，每段从0开始编址，并采用一段连续地址空间。段长由逻辑信息组的长度决定。整个作业的地址空间分成多个段，逻辑地址由段号(段名)和段内地址所组成。



该地址结构允许一个作业最长有64K个段，每段的最大长度为64KB。

## 4.6.2 分段系统的基本原理

### 1. 分段

#### ■ 分段式存储管理的原理

作业分为若干个段。每段分配一个连续的内存区，由于各段的长度不等，这些区域也就大小不一。作业各段间不要求连续。

#### ■ 基本分段式存储管理的原理

在段式存储管理原理的基础上，要求将整个作业的全部段装入内存。

#### ■ 请求分段式存储管理的原理

在段式存储管理原理的基础上，不要求将整个作业的全部段装入内存。只装入作业的几段即可运行，其余段可根据运行的需要再装入内存。

## 4.6.2 分段系统的基本原理

### 2. 基本分段式存储管理的实现

#### 1) 段表

在分段式存储管理系统中，系统为每个分段分配一个连续的分区，而进程中的各个段可以离散地移入内存中不同的分区中。

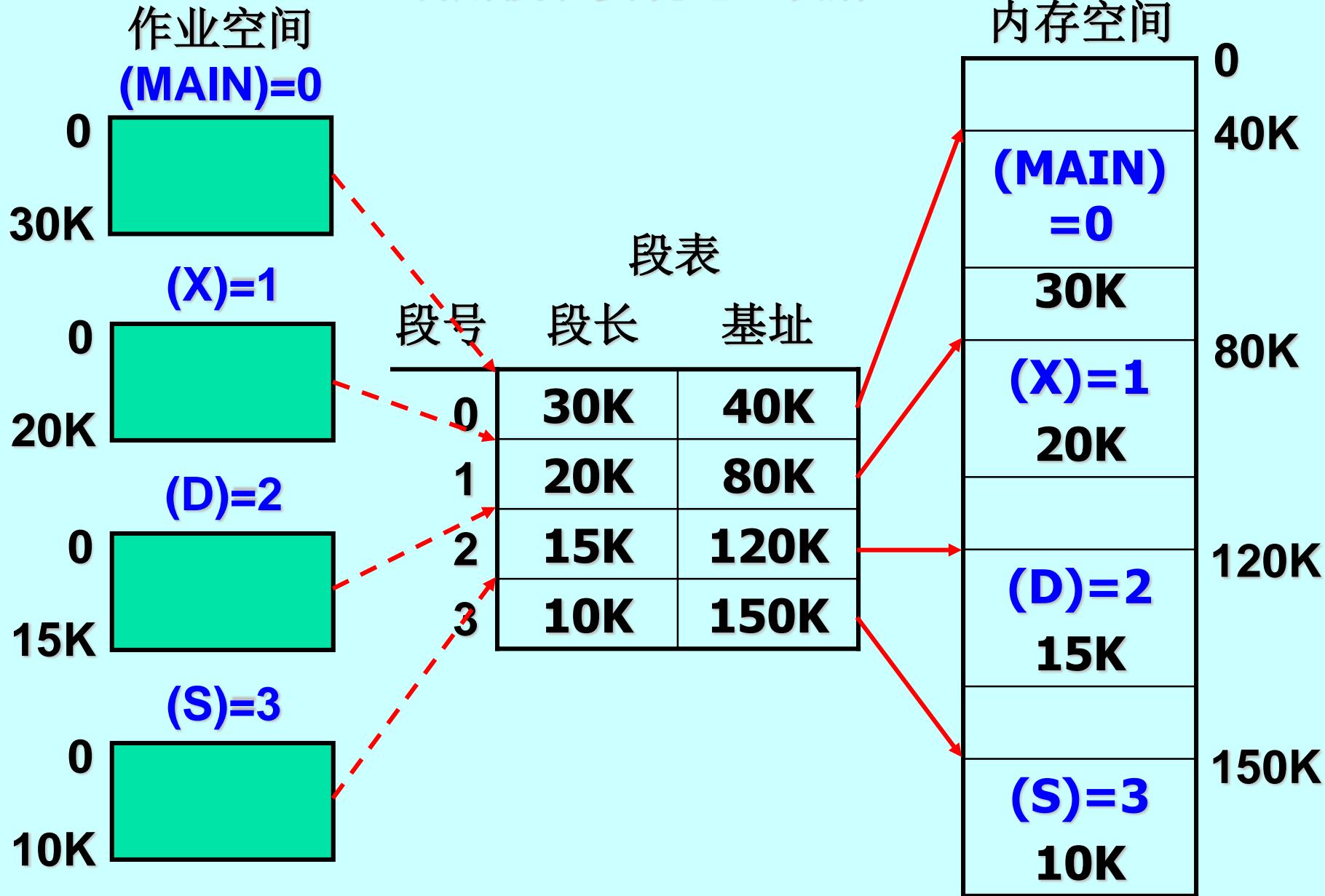
为使程序正常运行，须在系统中为每个进程建立一张段映射表，简称“**段表**”。每个段在表中占有一个表项。

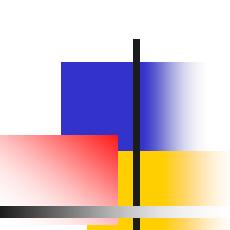
**段表结构：**段号；段在内存中的起始地址(基址)；段长。

段表可以存放在寄存器中，但更多的是存放在内存中。

段表用于实现从逻辑段到物理内存区的映射。

## 利用段表实现地址映射:





## 4.6.2 分段系统的基本原理

### 2. 基本分段式存储管理的实现

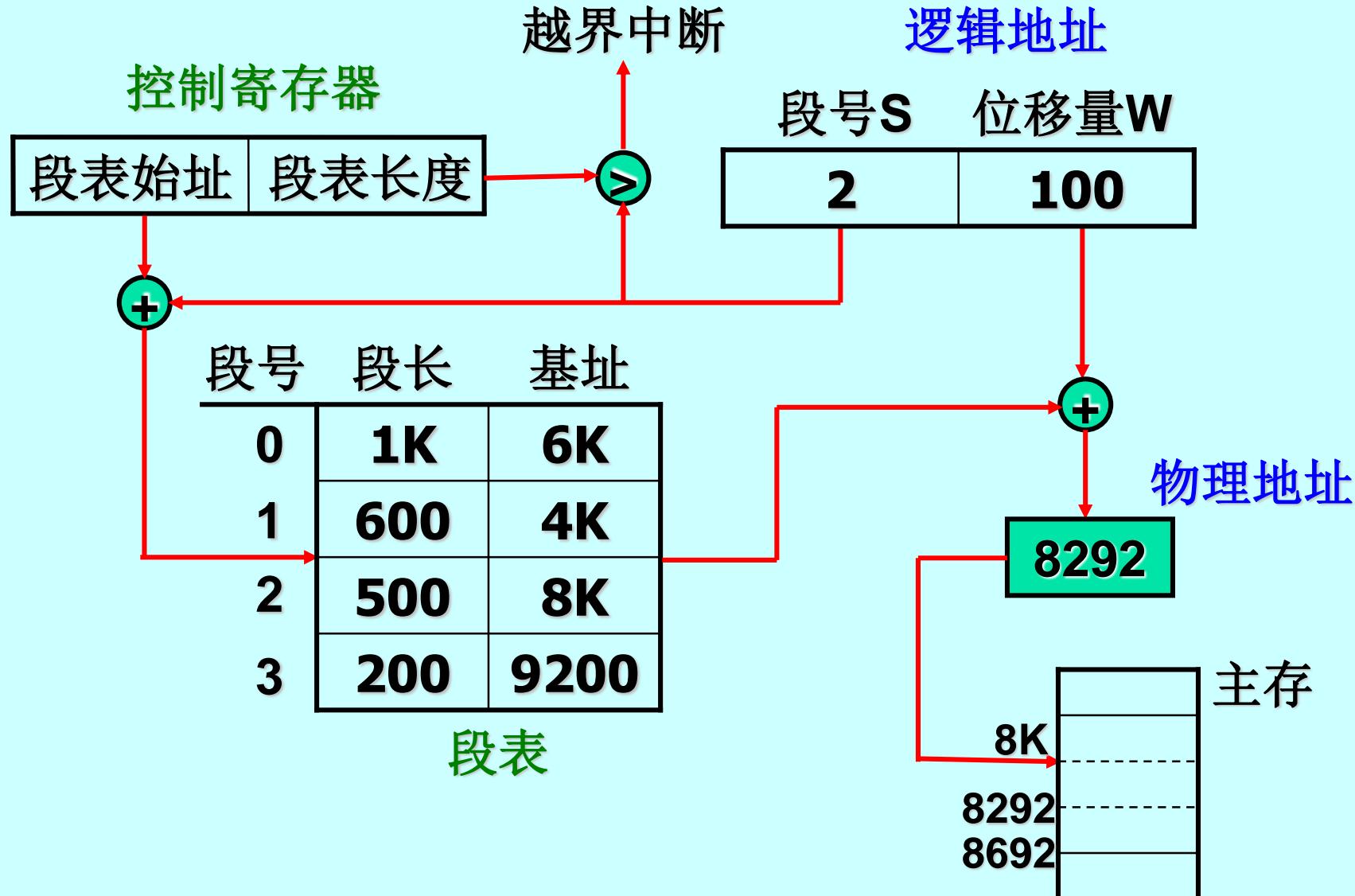
- 1) 段表
- 2) 地址变换机构

在系统中设置段表寄存器，用于存放段表始址和段表长度，以实现从进程的逻辑地址到物理地址的变换。

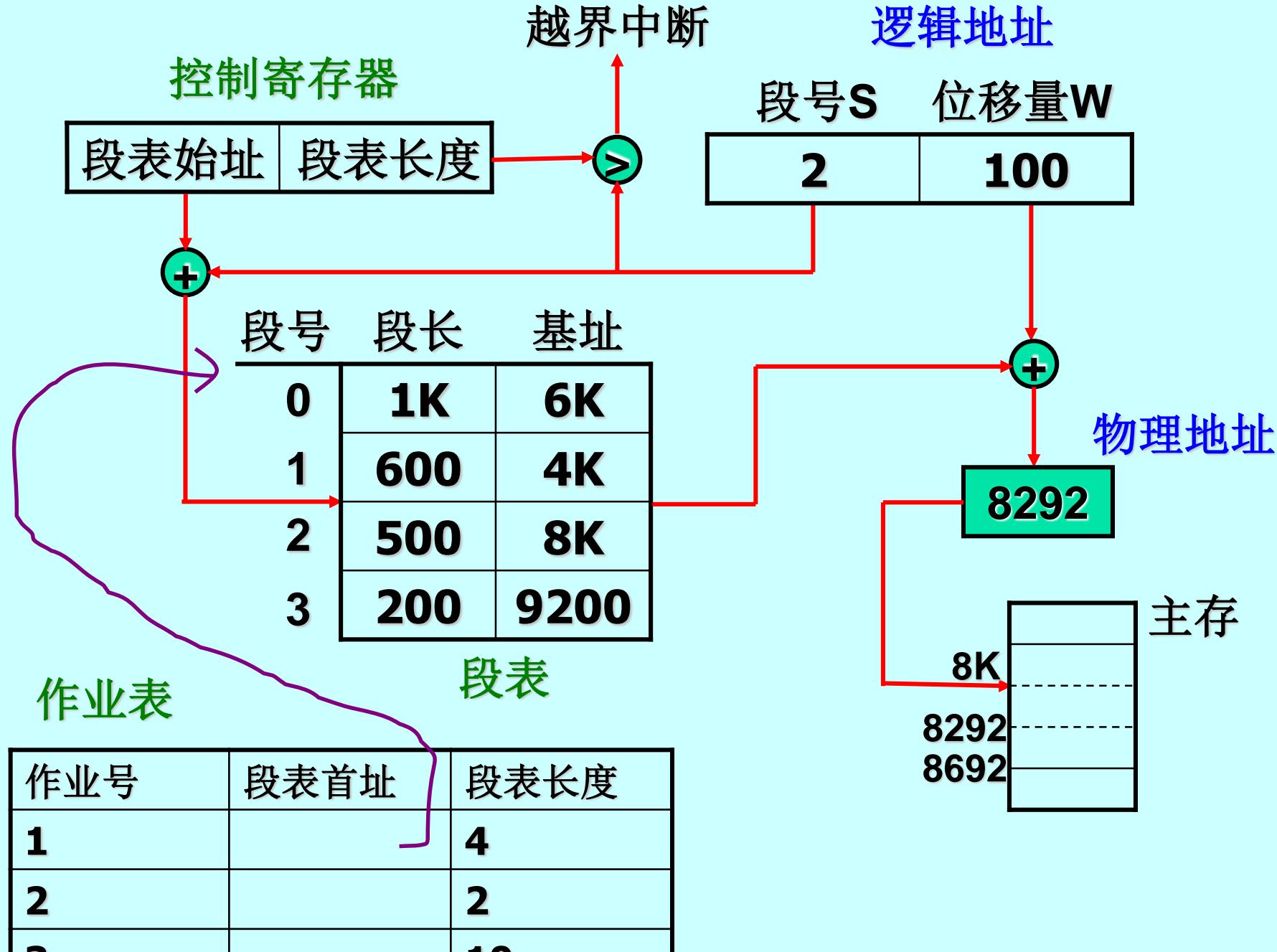
当段表存放在内存中时，每访问一个数据，都需访问两次内存，降低了计算机的速率。

解决方法：设置联想寄存器，用于保存最近常用的段表项。

# 分段系统的地址变换机构：



# 基本分段式存储管理之例



## 4.6.2 分段系统的基本原理

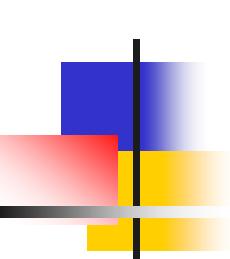
### 3. 分页和分段的主要区别

相似点：

采用离散分配方式，通过地址映射机构实现地址变换

不同点：

- 页是信息的物理单位，分页是为了满足系统的需要；段是信息的逻辑单位，含有一组意义相对完整的信息，分段是为了满足用户的需要。
- 页的大小固定且由系统确定，由系统把逻辑地址分为页号和页内地址，由机器硬件实现；段的长度不固定，取决于用户程序，编译程序对源程序编译时根据信息的性质划分。
- 分页的作业地址空间是一维的；分段的作业地址空间是二维的。



# 4.6 基本分段存储管理方式

4.6.1 分段存储管理方式的引入

4.6.2 分段系统的基本原理

4.6.3 信息共享

4.6.4 段页式存储管理方式

# 4.6

分段系

若干个进程

分页系

系统方便。

进程1

ed1
ed2
...
ed40
data1
...
data10

进程2

ed1
ed2
...
ed40
data1
...
data10

页表

21
22
...
60
61
...
70

页表

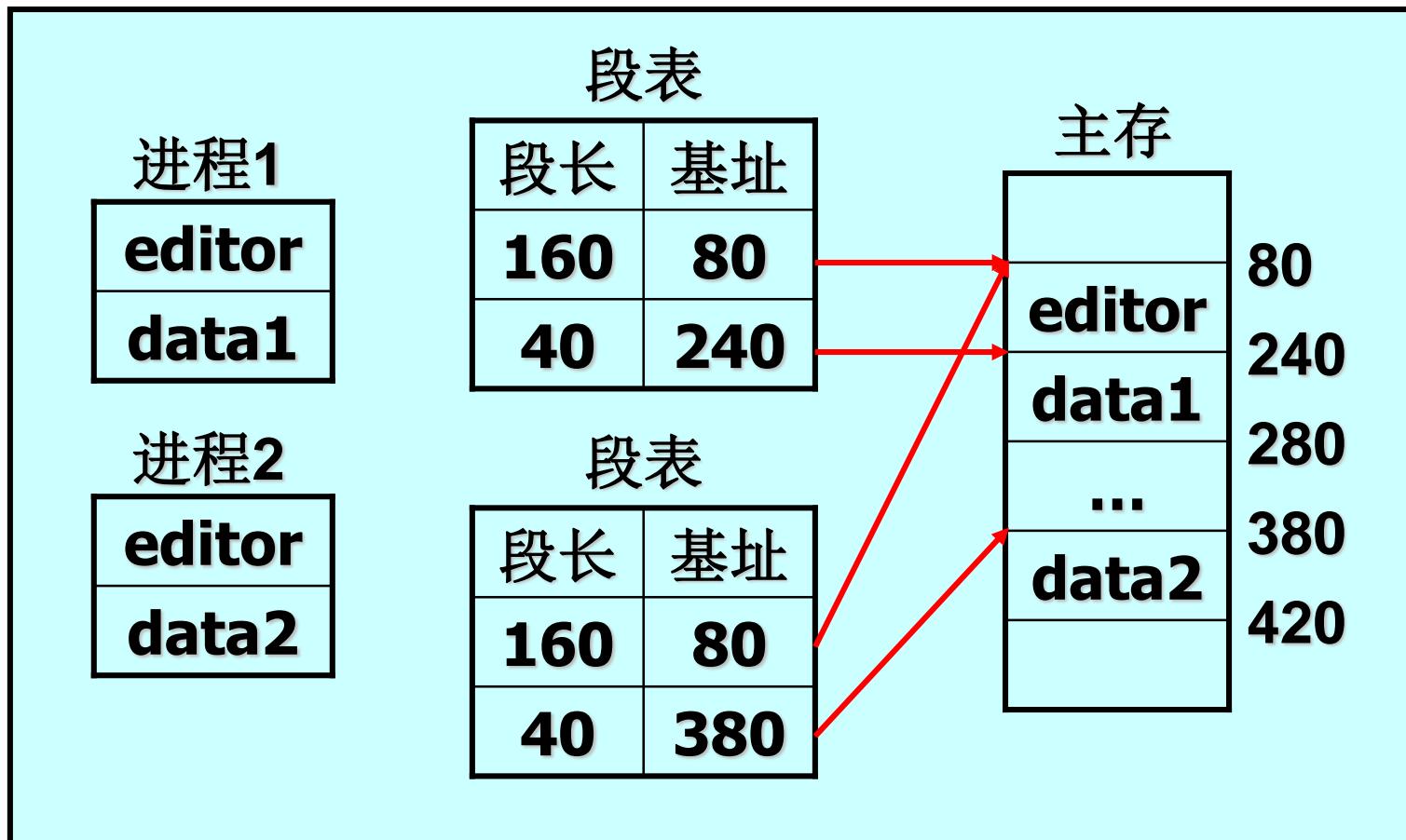
21
22
...
60
71
...
80

主存

0
...
ed1
ed2
...
ed40
data1
...
data10
70
71
...
data10
data1
...
data10
80

## 4.6.3 信息共享

在分段系统中，实现共享十分容易，只需在每个进程的段表中为共享程序设置一个段表项。



可重入代码又称为**纯代码**，是一种允许多个进程同时访问的代码，可重入代码不允许任何进程对它进行修改。

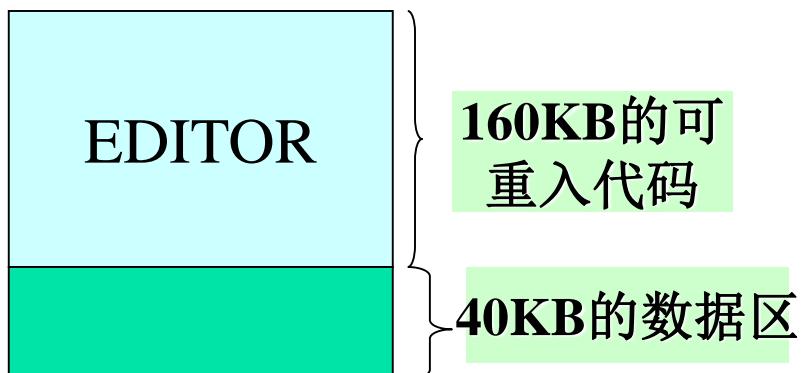
## 4.6.3 信息共享

### ■ 可重入代码(Reentrant Code):

又称为“纯代码”(Pure Code),在实现段共享时,需要用到可重入代码(Reentrant Code)。它是一种允许多个进程同时访问的代码,是一种不允许任何进程对其进行修改的代码。

在每个进程中,配以局部数据区,将在执行中可能改变的部分,拷贝到该数据区,这样,程序在执行时,只对该数据区(属于该进程私有)中的内容进行修改,而不去改变共享的代码,这时的可共享代码即成为可重入代码。

### ■ 例

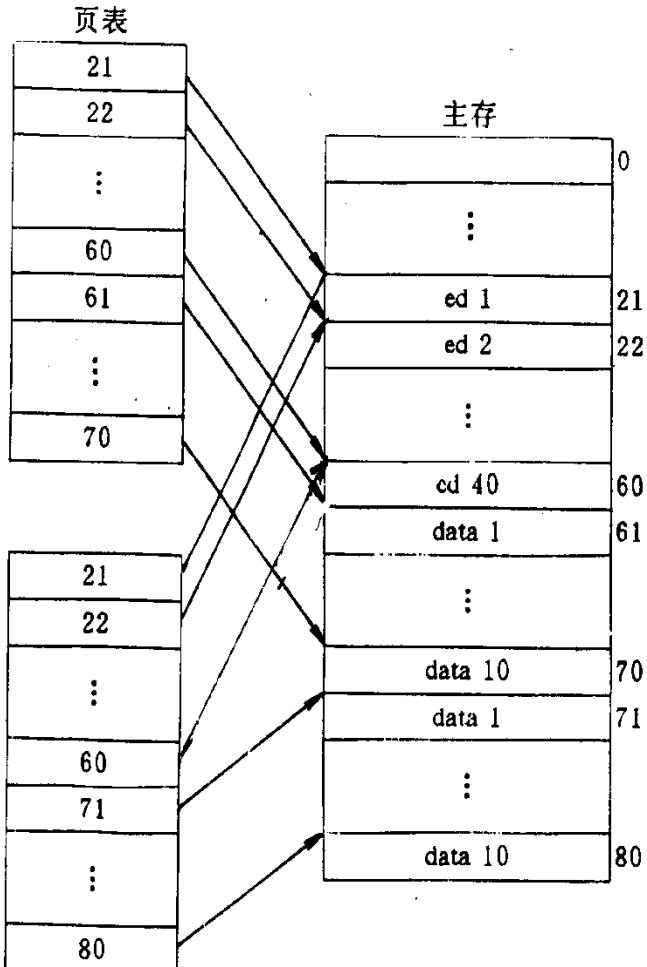


设同时有**40**个用户并发执行并使用该**EDITOR**,  $40 * (160 + 40) = 8\text{MB}$ 空间支持。如果代码可重入,内存只保留1份**COPY**即可,  $40 * 40 + 160 = 1760\text{KB}$ 。

# 下图是分段系统中共享 editor 编辑程序的示意图。

页表

进程 1
ed 1
ed 2
⋮
ed 40
data 1
⋮
data 10



进程 1

editor
data 1

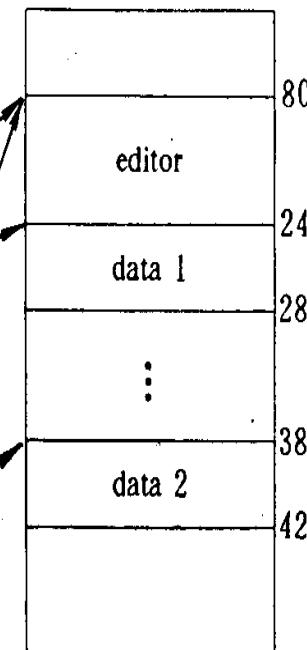
进程 2

editor
data 2

段表

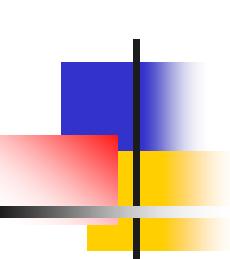
段长	基址
160	80
40	240

段长	基址
160	80
40	380



分段系统中共享 editor 的示意图

分页系统中共享 editor 的示意图



# 4.6 基本分段存储管理方式

4.6.1 分段存储管理方式的引入

4.6.2 分段系统的基本原理

4.6.3 信息共享

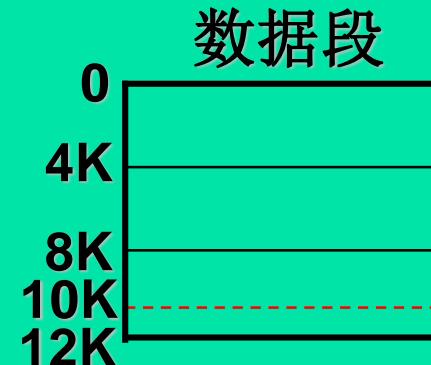
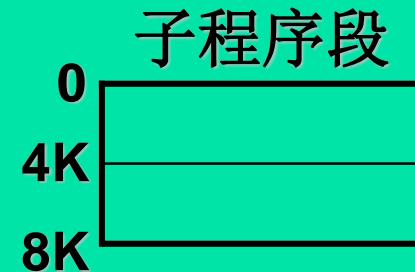
4.6.4 段页式存储管理方式

## 4.6.4 段页式存储管理方式

分段和分页存储管理方式各有优缺点。

把两者结合成一种新的存储管理方式——段页式存储管理方式，

作业地址空间：

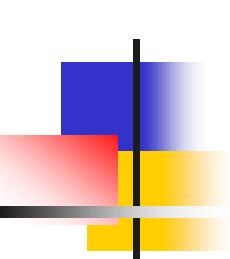


地址结构（逻辑地址）：

段号(S)

段内页号(P)

页内地址(W)



## 4.6.4 段页式存储管理方式

### 2. 实现

在段页式系统中，为了实现地址变换，增加一个段表寄存器，用来存放段表始址和段长。

## 段页式系统的地址变换机构：

### 作业表

作业号；段表首址；段表长度

段表寄存器

段表始址 | 段表长度

段超长

逻辑地址

段号 **S** | 页号 **P** | 页内地址

+

段表

0	
1	
2	
3	
	3

页表长度

页号

块号

0	1
1	
2	
b	

页表

块号 **b**

块内地址

物理地址

页表始址

段超长

逻辑地址

段号 **S** | 页号 **P** | 页内地址

0	1
1	
2	
b	

块号 **b**

块内地址

物理地址

# 段表寄存器

段表大小 段表始址

段号	状态	页表大小	页表始址
0	1		
1	1		
2	1		
3	0		
4	1		

段 表

页号	状态	存储块号
0	1	
1	1	
2	1	
3	0	
4	1	

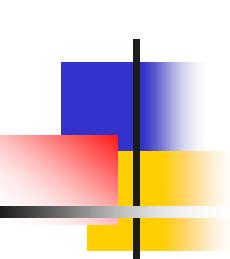
页 表

页号	状态	存储块号
0	1	
1	1	
2	1	
3	0	
4	1	

页 表

操作系统

主存



## 4.6.4 段页式存储管理方式

在段页式系统中，为了获得一条指令或数据，需访问三次内存：

第一次：访问内存中的段表，取得页表始址

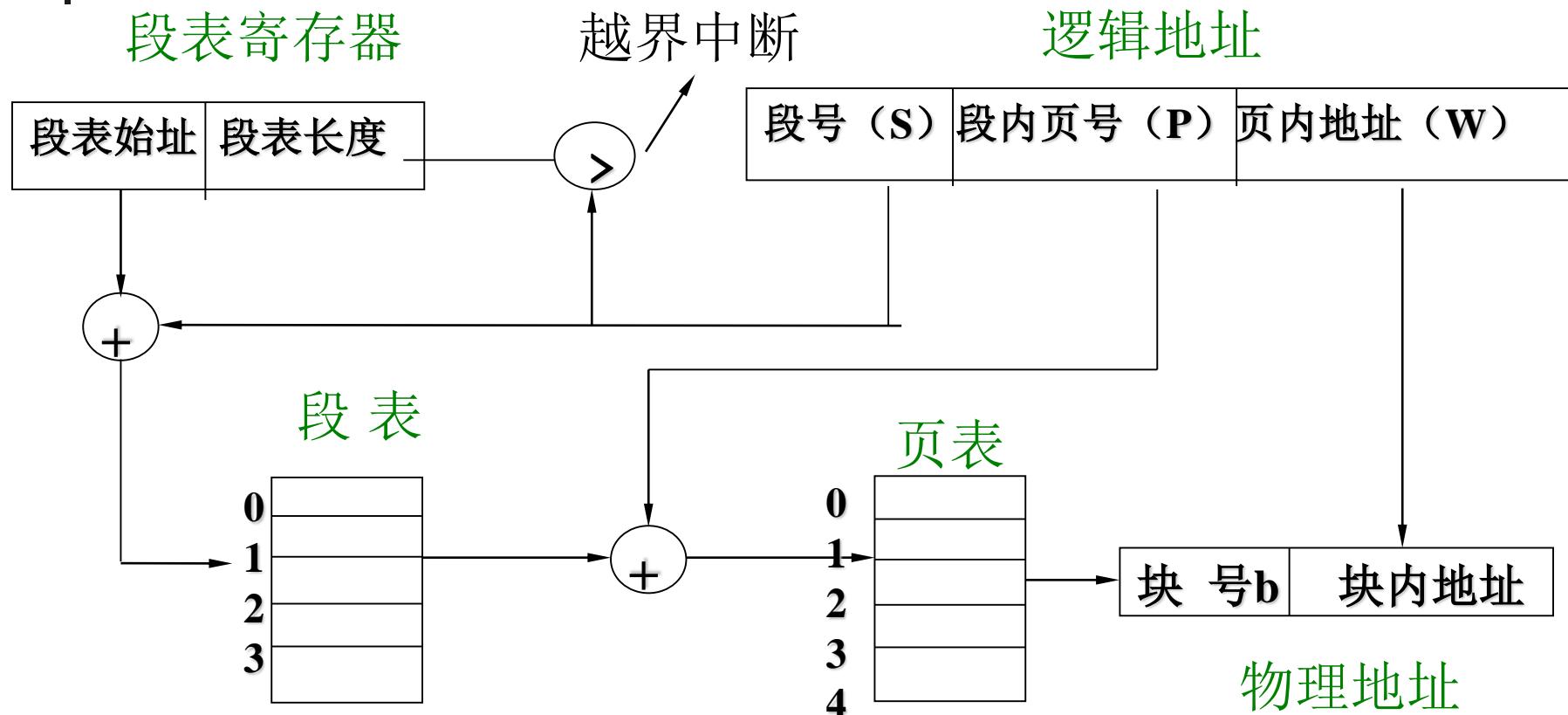
第二次：访问内存中的页表，取得该页所在的物理块号，将块号与页内地址形成物理地址

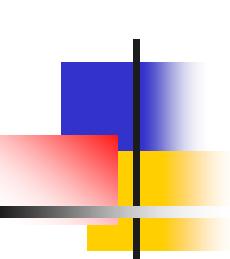
第三次：访问第二次所得的地址，取出指令或数据

缺点：访存次数增加两倍

解决方法：增设高速缓冲寄存器

## 4.6.4 段页式存储管理方式





# 作业

**补充题1：**某用户进程编程空间共4个页面，每页1KB，主存为64KB。假定该用户进程的页表如下。

页号	物理块号
0	5
1	10
2	4
3	7

求下面与虚拟地址相对应的物理地址（如果在主存中找不到，即为页失效）：

- (1) 0A5C(H)
- (2) 1A5C(H)

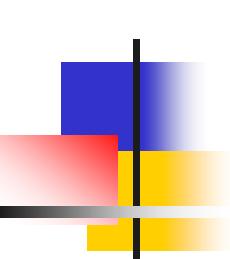
**答案：**

# 补充题2

- 每页1KB，对于下面的页表内容，逻辑地址37390，40462对应的物理地址分别是\_\_\_\_\_、\_\_\_\_\_。
- 每页1KB，8页逻辑空间，32块物理块，逻辑地址的有效位是\_\_\_\_位，物理地址至少\_\_\_\_位。

页号	页框号
36	84
37	85
38	95
39	96

答案：



## 补充题3：

- 逻辑地址 (2, 88) 对应的物理地址\_\_\_\_\_，  
(4, 100) 对应物理地址\_\_\_\_\_。

段号	起址	段长
0	<b>219</b>	<b>600</b>
1	<b>2300</b>	<b>14</b>
2	<b>90</b>	<b>100</b>
3	<b>1327</b>	<b>580</b>
4	<b>1952</b>	<b>96</b>

答案：