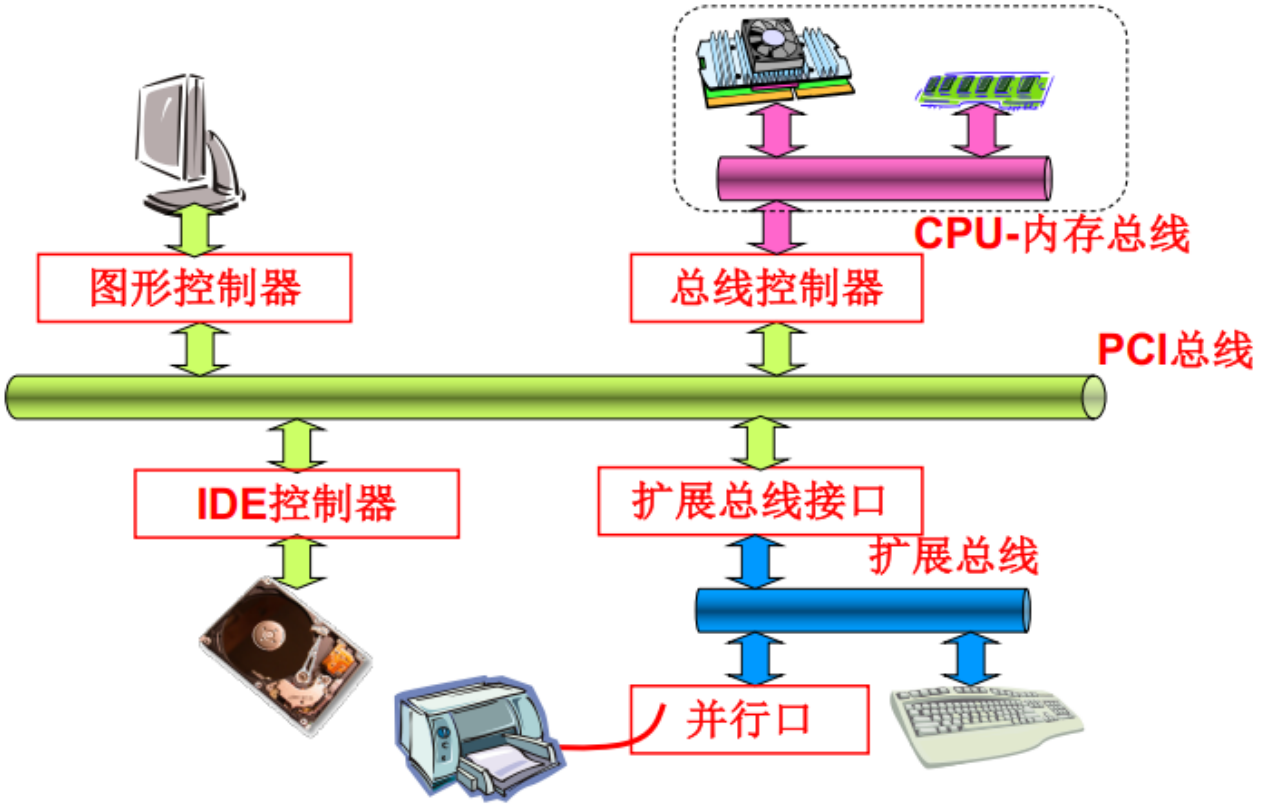
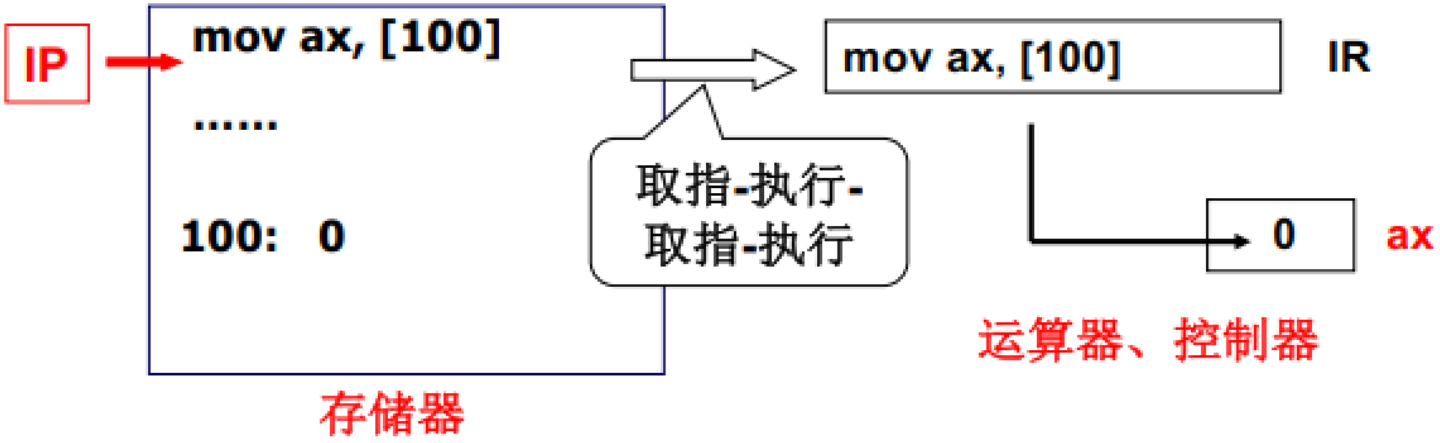
# 第四章　存储器管理

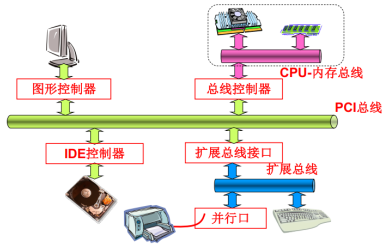



如何将程序放到内存中，如何执行？？

# 第四章　存储器管理




如何将程序放到内存中，如何执行？？



# 第四章　存储器管理

C 程序

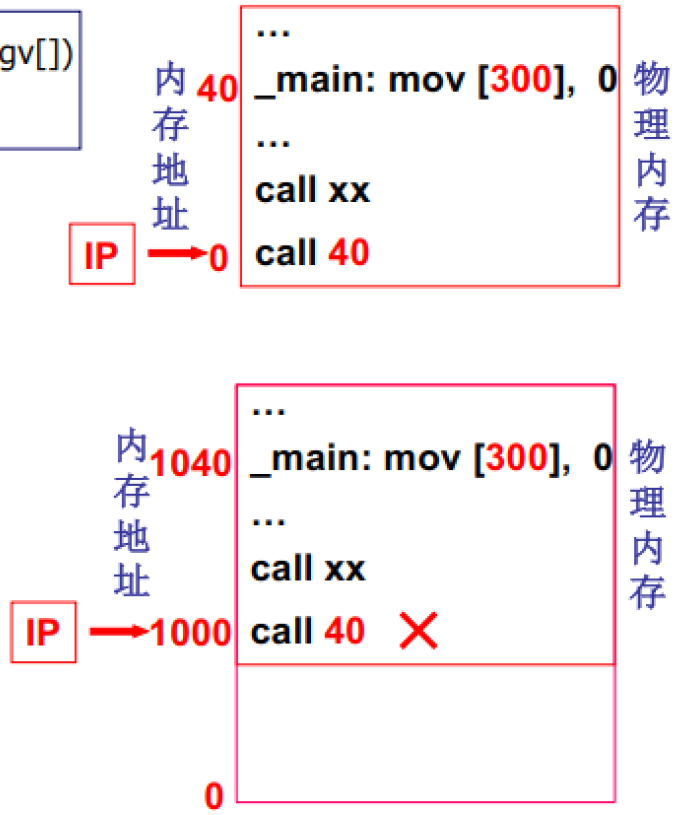
```
int main(int argc, char* argv[])
{ ...
```

汇编程序

```
.text
_entry: //入口地址
    call _main
    call _exit
_main:
    ...
    ret
```

_main地址是40

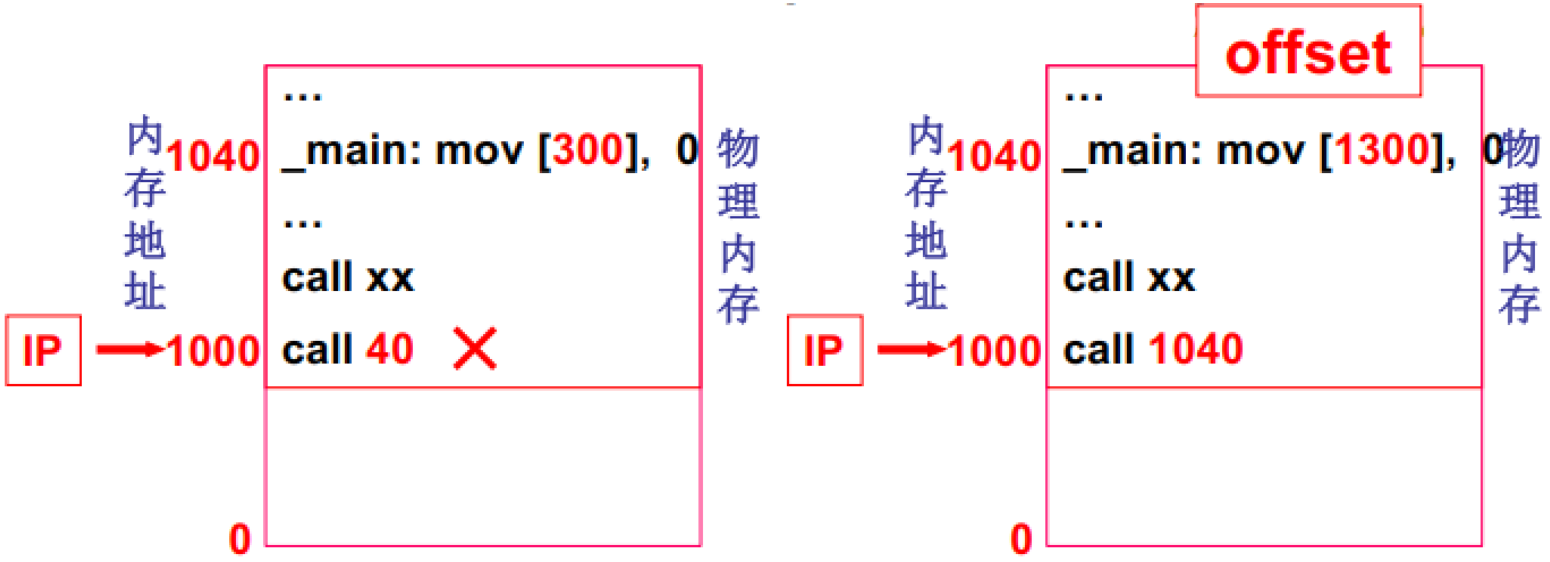
```
_entry: //入口地址
    call 40
    call xx
_main: //偏移是40
    ...
```




指令从0地址开始，_main的偏移必须是40？所有的程序地址都是从0开始？

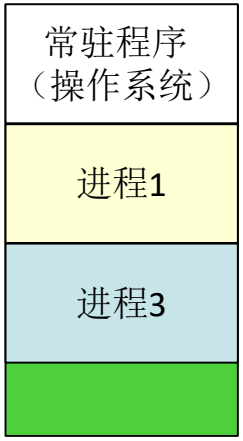
程序存放到空闲地址

地址偏移

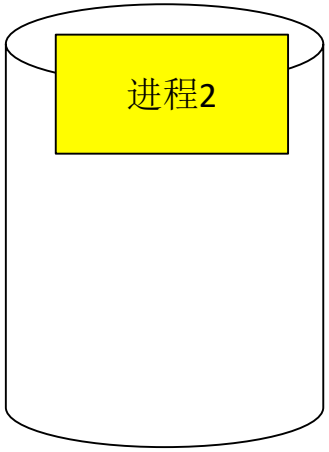



- **地址的重定位**：相对地址 → 绝对地址
- **编译时重定位**：只能放在内存的固定地址，效率高，不灵活
- **载入时重定位**：一旦载入内存就不能动了，相对灵活

# 第四章　存储器管理




内存




磁盘

进程1睡眠 换出

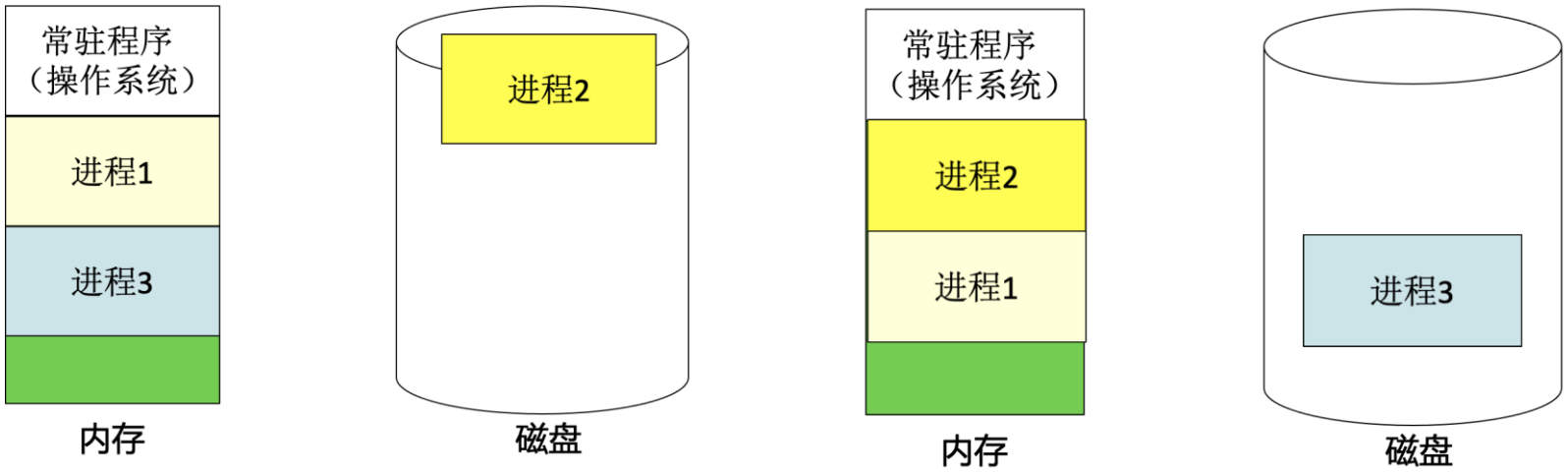
进程2　　换入

进程3睡眠 换出

进程1　　换入

**对换（swapping），内存的换出，内存换入**

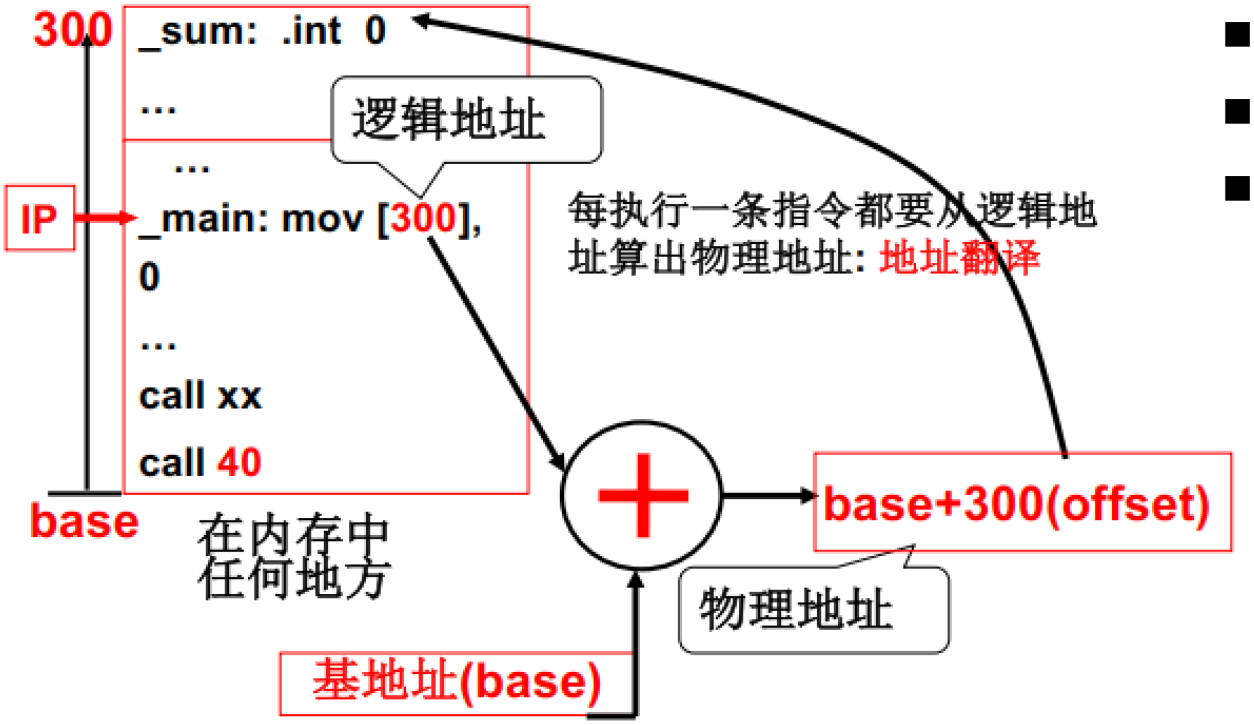
# 第四章　存储器管理




**进程1载入后又发生了移动**

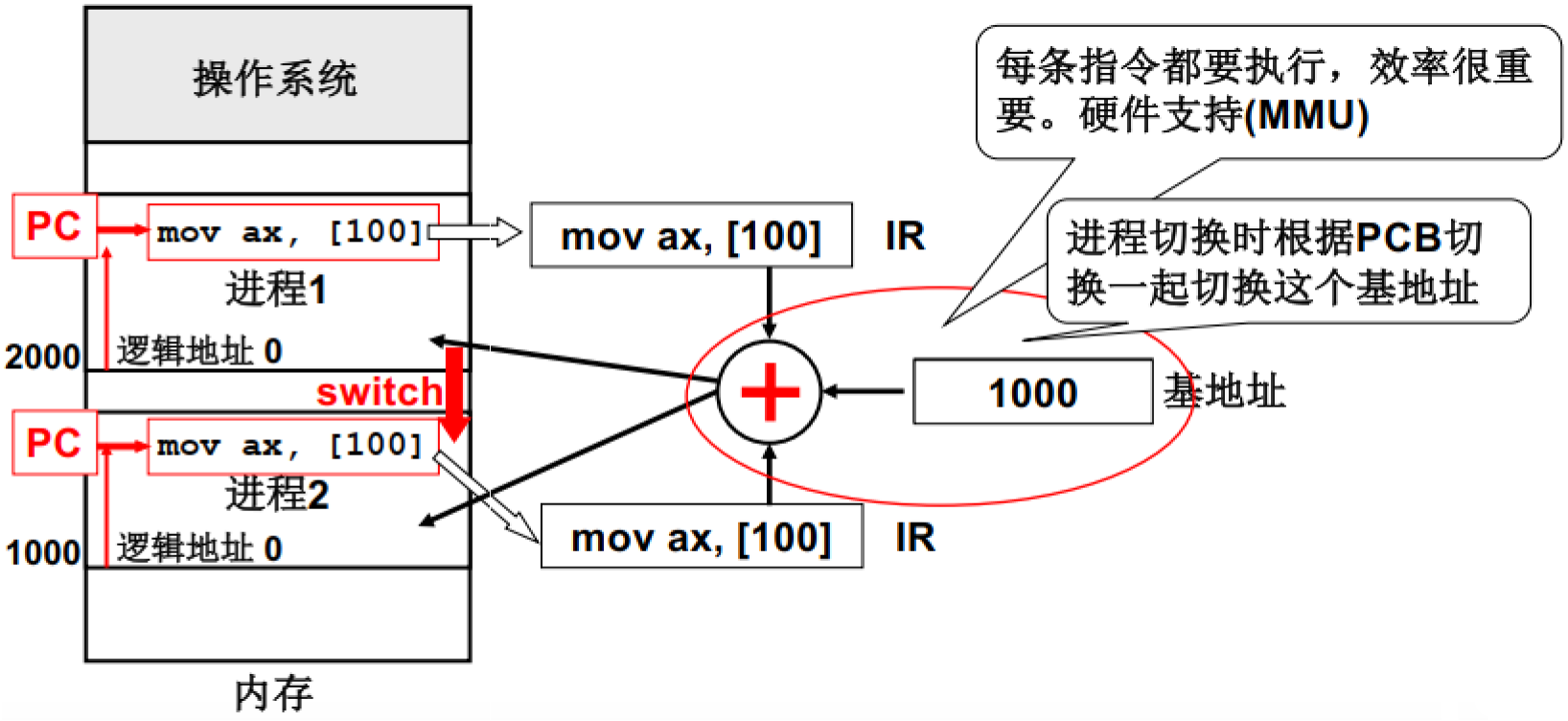
**装载后应仍允许重定位**

**运行时重定位：地址翻译**

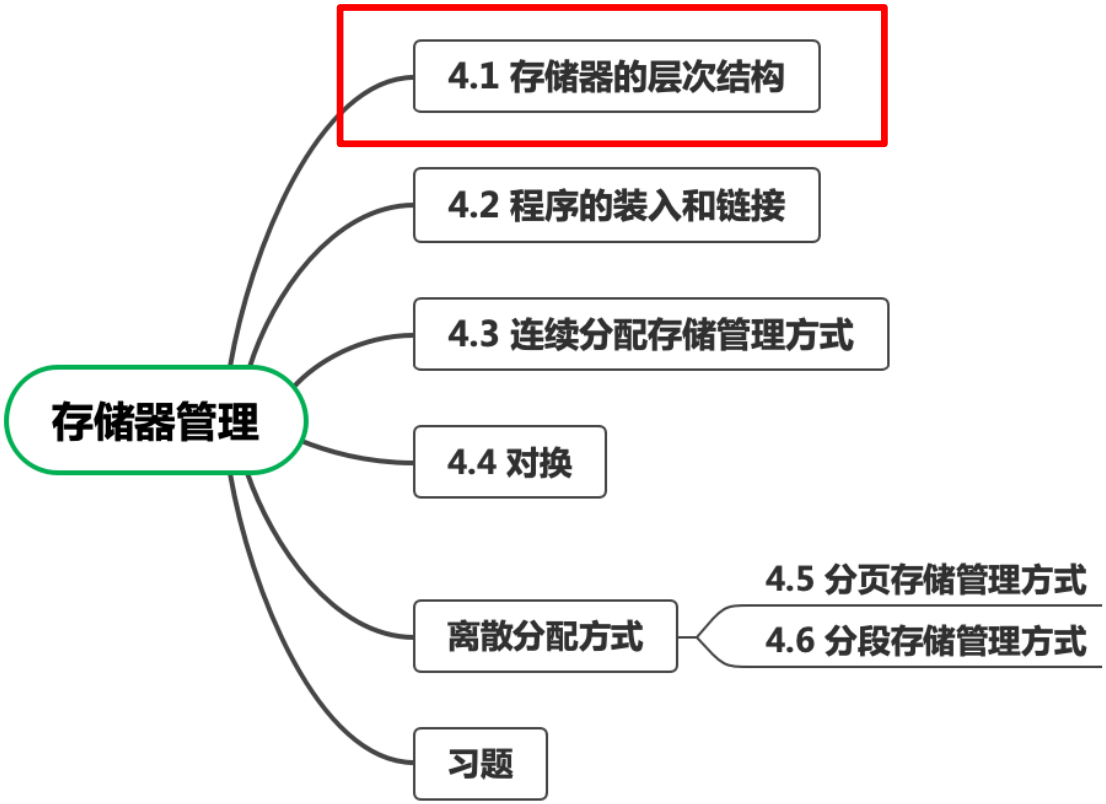



- 每个进程有各自的基地址
- 基地址存放于 PCB 中
- 执行指令时第一步先从PCB中取出这个基地址

# 第四章　存储器管理


操作系统
PC
mov ax, [100]
进程1
2000
逻辑地址 0
switch
PC
mov ax, [100]
进程2
1000
逻辑地址 0
内存
mov ax, [100]
IR
mov ax, [100]
IR
1000
基地址
每条指令都要执行，效率很重要。硬件支持(MMU)
进程切换时根据PCB切换一起切换这个基地址

# 第四章　存储器管理

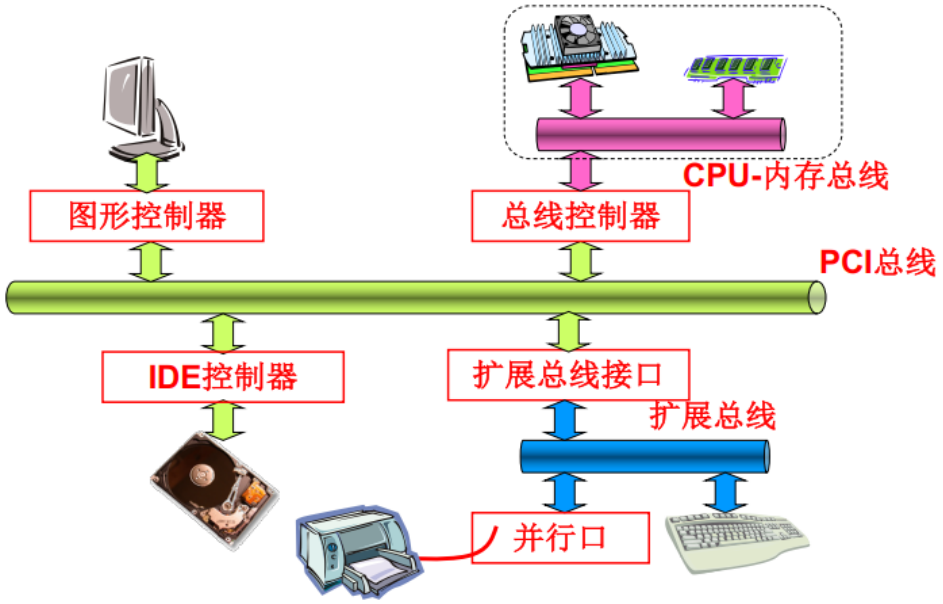

存储器管理
4.1 存储器的层次结构
4.2 程序的装入和链接
4.3 连续分配存储管理方式
4.4 对换
离散分配方式
4.5 分页存储管理方式
4.6 分段存储管理方式
习题

# 4.1 存储器的层次结构

在计算机执行时，几乎每一条指令都涉及对存储器的访问，因此要求对存储器的访问速度能跟得上处理机的运行速度。或者说，①存储器的速度必须非常快，能与处理机的速度相匹配，否则会明显地影响到处理机的运行。此外还要求②存储器具有非常大的容量，而且③存储器的价格还应很便宜。

条件严苛，难以满足！

解决方法：多层结构存储器系统

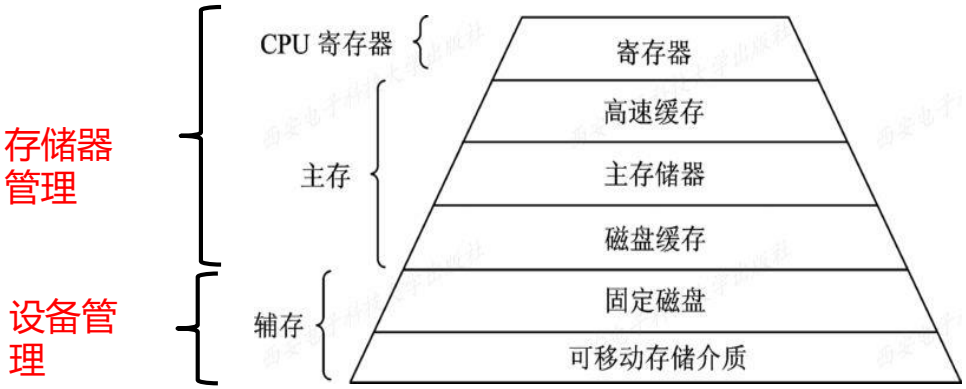


# 4.1 存储器的层次结构

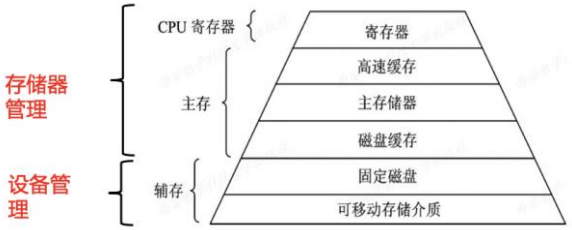
## 4.1.1 多层结构的存储器系统

### 1. 存储器的多层结构

对于通用计算机而言，存储层次至少应具有三级：最高层为CPU寄存器，中间为主存，最底层是辅存。在较高档的计算机中，还可以根据具体的功能细分为寄存器、高速缓存、主存储器、磁盘缓存、固定磁盘、可移动存储介质等6层。如图4-1所示。



# 4.1 存储器的层次结构



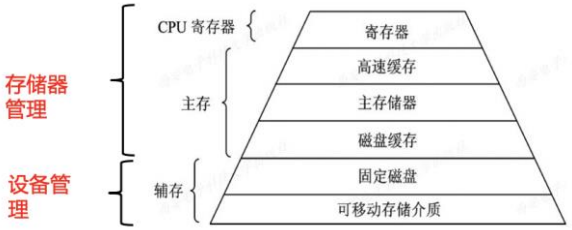

## 4.1.1 多层结构的存储器系统

### 2. 可执行存储器

在计算机系统的存储层次中，寄存器和主存储器又被称为可执行存储器。对于存放于其中的信息，与存放于辅存中的信息相比较而言，计算机所采用的访问机制是不同的，所需耗费的时间也是不同的。

进程可以在很少的时钟周期内使用一条load或store指令对可执行存储器进行访问。但对辅存的访问则需要通过I/O设备实现，因此，在访问中将涉及到中断、设备驱动程序以及物理设备的运行，所需耗费的时间远远高于访问可执行存储器的时间，一般相差3个数量级甚至更多。

# 4.1 存储器的层次结构




## 4.1.2 主存储器与寄存器

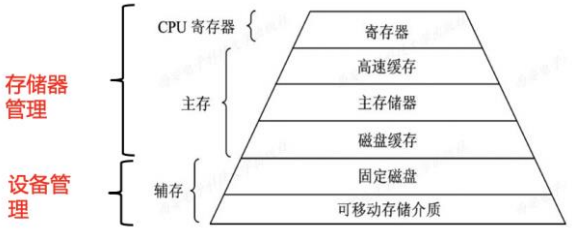
### 1. 主存储器

主存储器简称内存或主存，是计算机系统中的主要部件，用于保存进程运行时的程序和数据，也称可执行存储器。

### 2. 寄存器

寄存器具有与处理机相同的速度，故对寄存器的访问速度最快，完全能与CPU协调工作，但价格却十分昂贵，因此容量不可能做得很大。

# 4.1 存储器的层次结构




## 4.1.3 高速缓存和磁盘缓存

### 1. 高速缓存

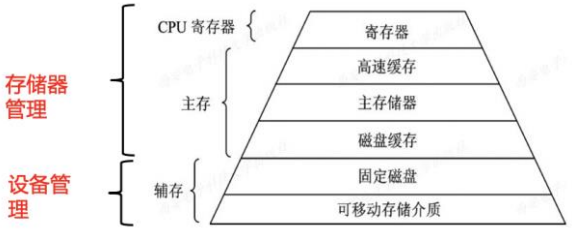
高速缓存是现代计算机结构中的一个重要部件，它是介于寄存器和存储器之间的存储器，主要用于备份主存中较常用的数据，以减少处理机对主存储器的访问次数，这样可大幅度地提高程序执行速度。

高速缓存容量远大于寄存器，而比内存约小两到三个数量级左右，从几十KB到几MB，访问速度快于主存储器。

进程的程序和数据存放于主存储器中，每当要访问时，才被临时复制到一个速度较快的高速缓存中。

CPU 取指执行过程中先检查缓存。若存在则使用；若不存在，则访问主存。

# 4.1 存储器的层次结构




## 4.1.3 高速缓存和磁盘缓存

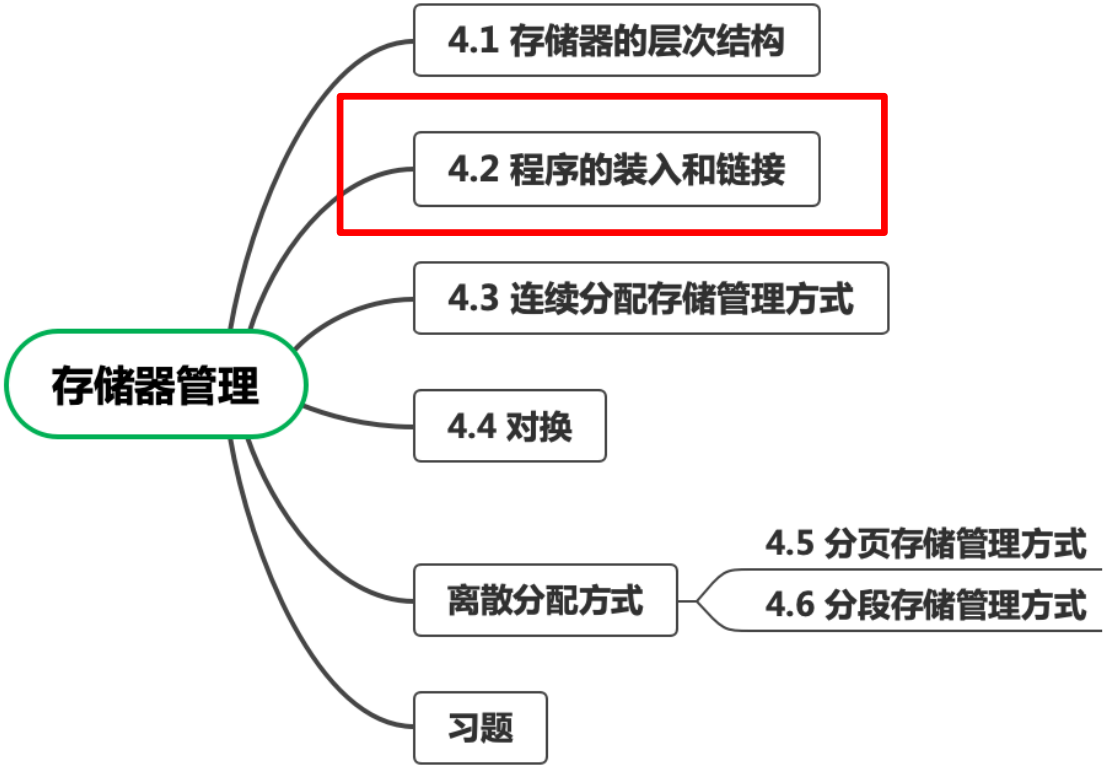
### 2. 磁盘缓存

由于目前磁盘的I/O速度远低于对主存的访问速度，为了缓和两者之间在速度上的不匹配，而设置了磁盘缓存，主要用于暂时存放频繁使用的一部分磁盘数据和信息，以减少访问磁盘的次数。

但磁盘缓存与高速缓存不同，它本身并不是一种实际存在的存储器，而是利用主存中的部分存储空间暂时存放从磁盘中读出(或写入)的信息。

主存也可以看作是辅存的高速缓存，因为，辅存中的数据必须复制到主存方能使用，反之，数据也必须先存在主存中，才能输出到辅存。

# 第四章　存储器管理


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 连续分配存储管理方式
4.4 对换
离散分配方式
4.5 分页存储管理方式
4.6 分段存储管理方式
习题

# 4.2 程序的装入和链接

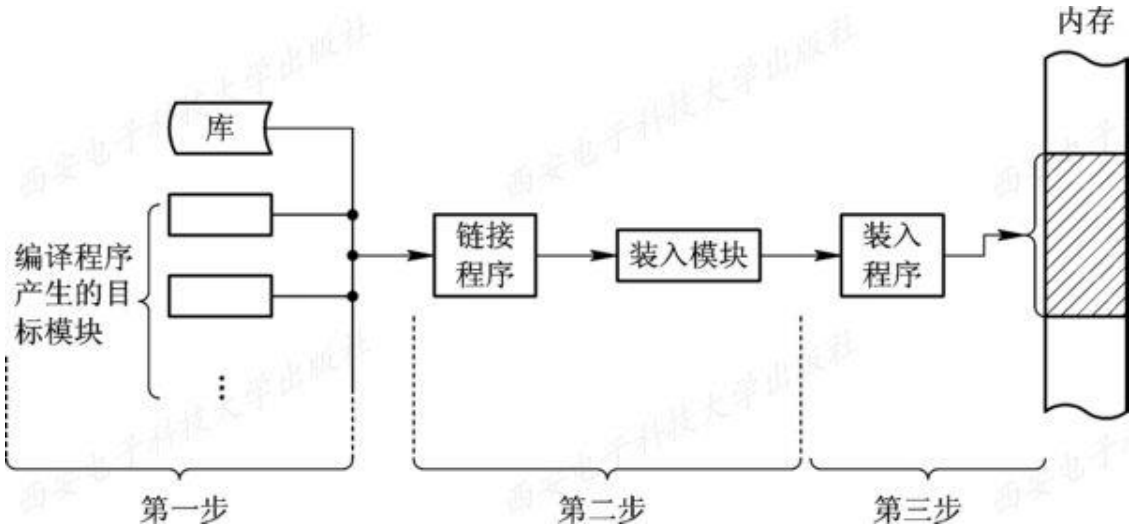
用户程序要在系统中运行，必须先将它装入内存，然后再将其转变为一个可以执行的程序，通常都要经过以下几个步骤：

(1) 编译

由编译程序(Compiler)对用户源程序进行编译，形成若干个目标模块(Object Module)；

(2) 链接

(3) 装入



# 4.2 程序的装入和链接

用户程序要在系统中运行，必须先将它装入内存，然后再将其转变为一个可以执行的程序，通常都要经过以下几个步骤：

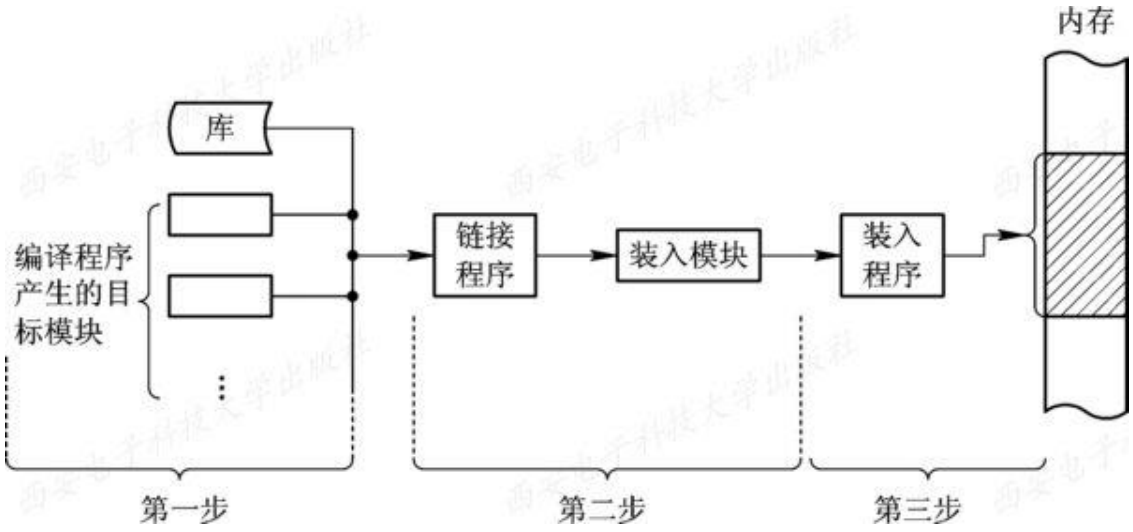
(1) 编译

(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)；

(3) 装入

由装入程序(Loader)将装入模块装入内存。



# 4.2 程序的装入和链接

## 4.2.1 程序的装入

为了阐述上的方便，我们先介绍一个无需进行链接的单个目标模块的装入过程。该目标模块也就是装入模块。在将一个装入模块装入内存时，可以有如下三种装入方式：

**1. 绝对装入方式(Absolute Loading Mode)**

当计算机系统很小，且仅能运行单道程序时，完全有可能知道程序将驻留在内存的什么位置。此时可以采用绝对装入方式。用户程序经编译后，将产生绝对地址(即物理地址)的目标代码。

**2. 可重定位装入方式(Relocation Loading Mode)**

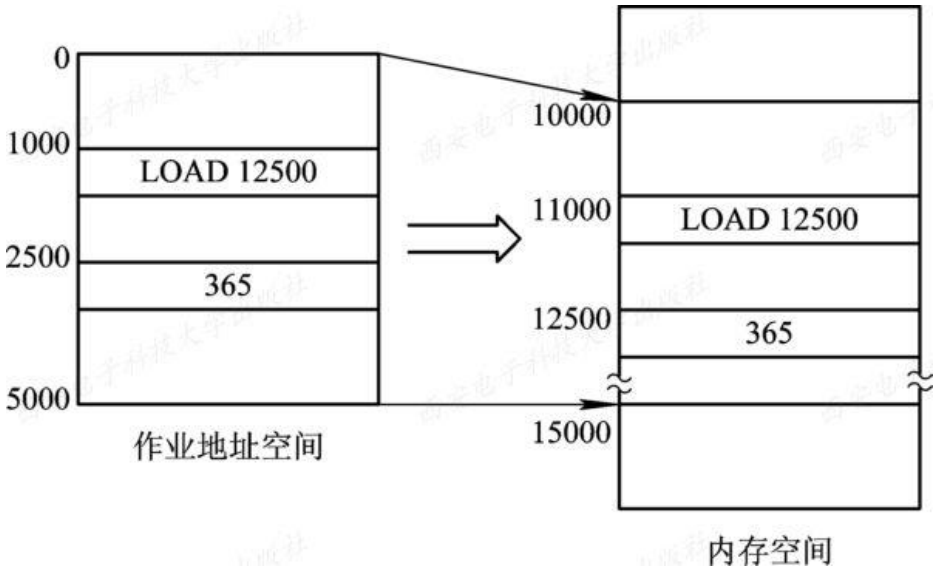
绝对装入方式只能将目标模块装入到内存中事先指定的位置，这只适用于单道程序环境。而在多道程序环境下，编译程序不可能预知经编译后所得到的目标模块应放在内存的何处。因此，对于用户程序编译所形成的若干个目标模块，它们的起始地址通常都是从0开始的，程序中的其它地址也都是相对于起始地址计算的。

# 4.2 程序的装入和链接

**1. 绝对装入方式**
**(Absolute Loading Mode)**

**2. 可重定位装入方式**
**(Relocation Loading Mode)**

地址变换通常是在进程装入时一次完成，以后不再改变，故称为静态重定位。



# 4.2 程序的装入和链接

## 4.2.1 程序的装入

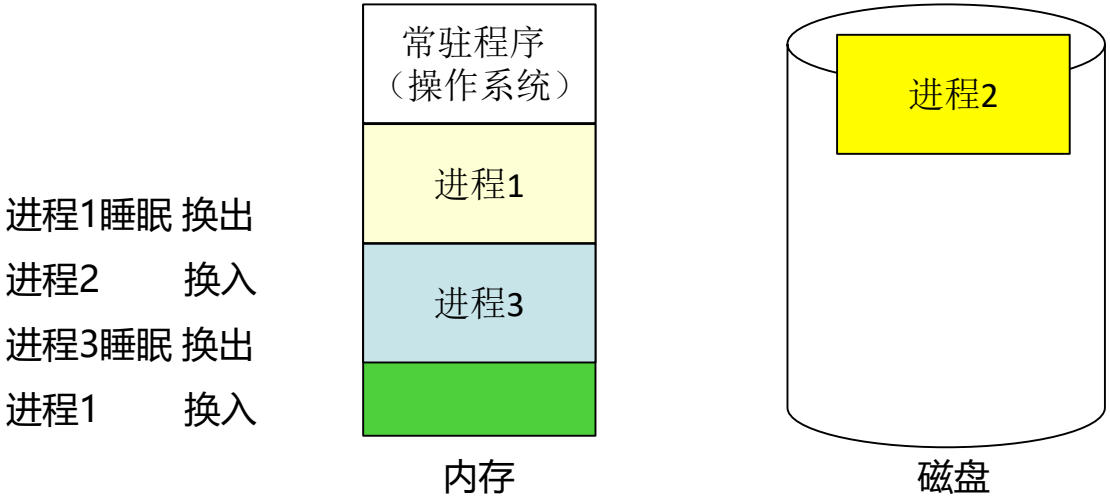
### 3. 动态运行时的装入方式(Dynamic Run-time Loading)

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。但该方式并不允许程序运行时在内存中移动位置。

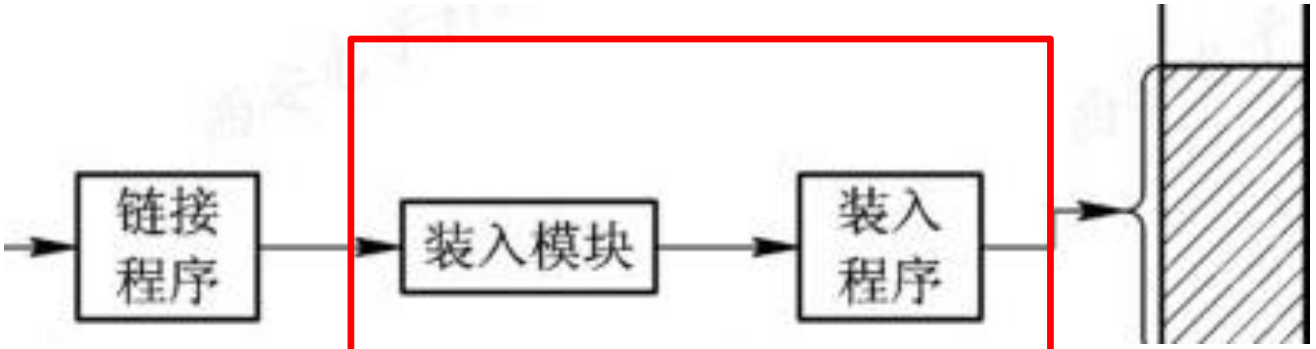
一个进程可能被多次换入换出，每次换入地址不同。

地址转换推迟到程序真正执行时才进行。

需要重定位寄存器的支持。



# 4.2 程序的装入和链接

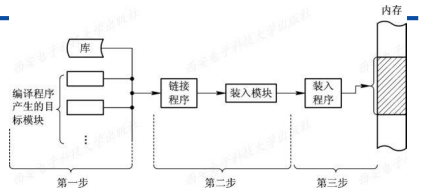



1. 绝对装入方式
   (Absolute Loading Mode)
2. 可重定位装入方式（静态重定位）
   (Relocation Loading Mode)
3. 动态运行时的装入方式
   (Dynamic Run-time Loading)

# 4.2 程序的装入和链接

## 4.2.2 程序的链接

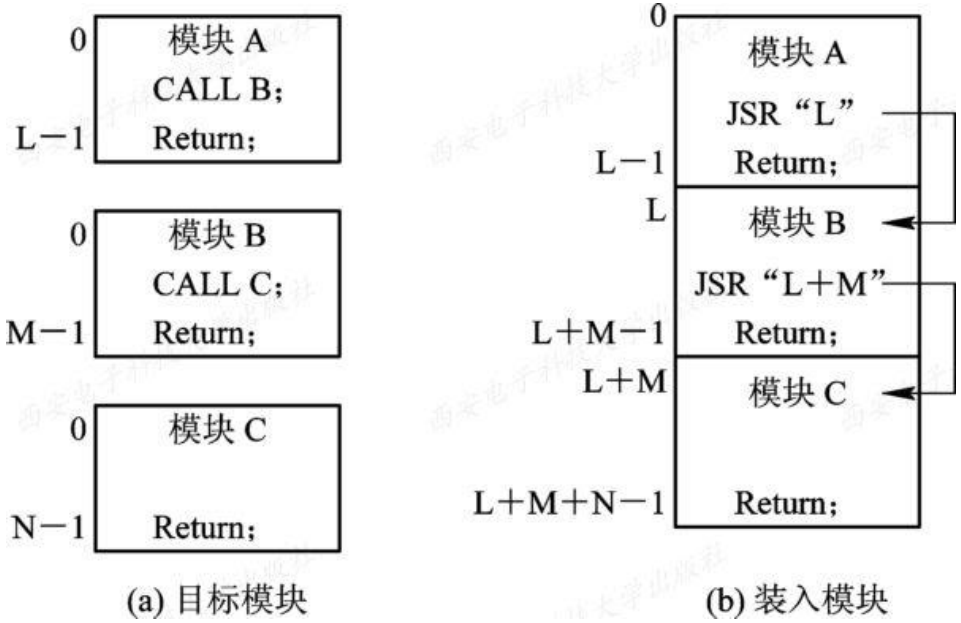
### 1. 静态链接(Static Linking)方式




在程序运行之前，先将各目标模块及它们所需的库函数链接成一个完整的装配模块，以后不再拆开。B和C都属于外部调用符号，在将这几个目标模块装配成一个装入模块时，须解决以下两个问题：

(1) 对相对地址进行修改。

(2) 变换外部调用符号。



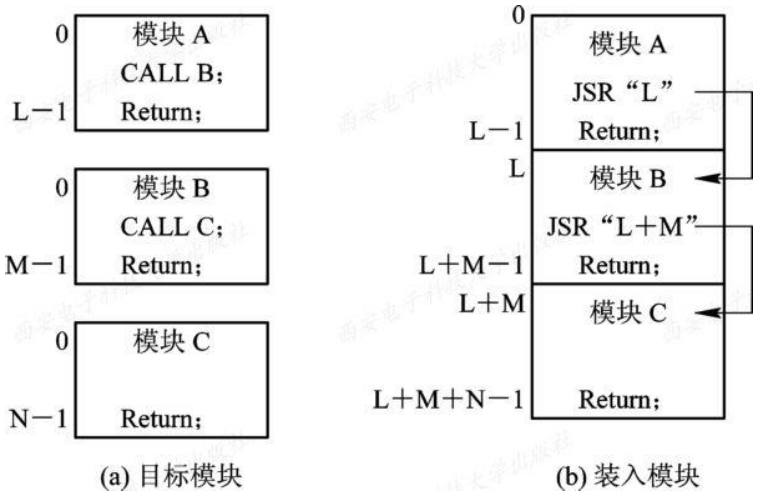
### 4.2.2 程序的链接

#### 2. 装入时动态链接(Load-time Dynamic Linking)

这是指将用户源程序编译后所得到的一组目标模块，在装入内存时，采用边装入边链接的链接方式。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，还要按照图所示的方式修改目标模块中的相对地址。装入时动态链接方式有以下优点：

(1) 便于修改和更新。

(2) 便于实现对目标模块的共享。



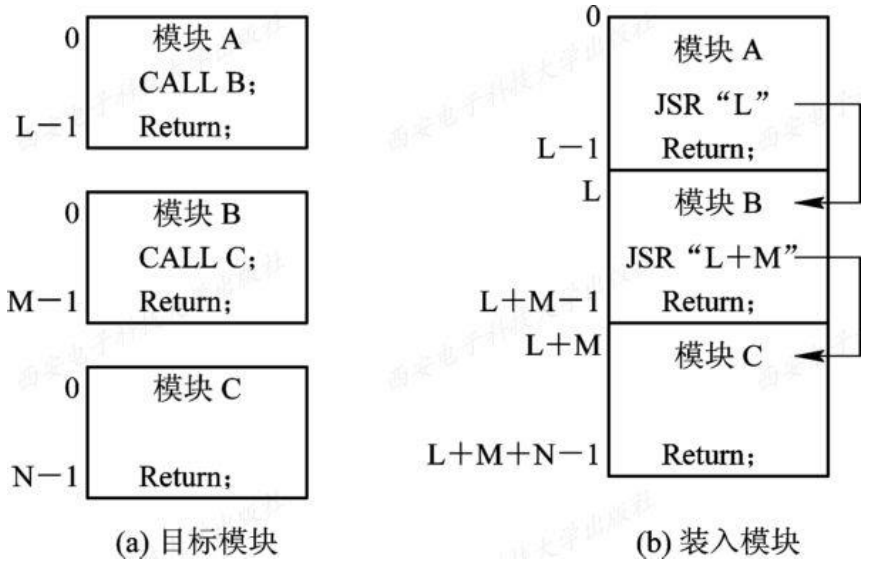
# 4.2 程序的装入和链接

## 4.2.2 程序的链接

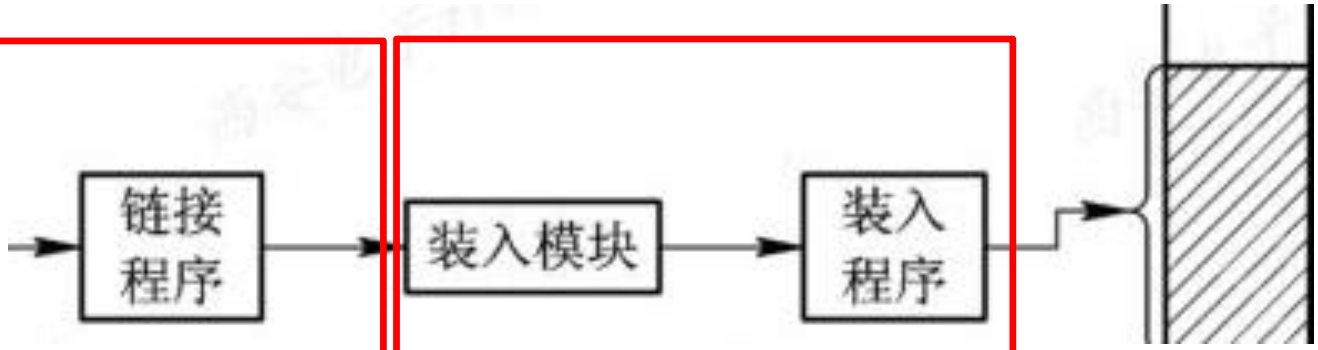
### 3. 运行时动态链接(Run-time Dynamic Linking)

在许多情况下，应用程序在运行时，每次要运行的模块可能是不相同的。但由于事先无法知道本次要运行哪些模块，故只能是将所有可能要运行到的模块全部都装入内存，并在装入时全部链接在一起。显然这是低效的，因为往往会有部分目标模块根本就不运行。

比较典型的例子是作为错误处理用的目标模块，如果程序在整个运行过程中都不出现错误，则显然就不会用到该模块。



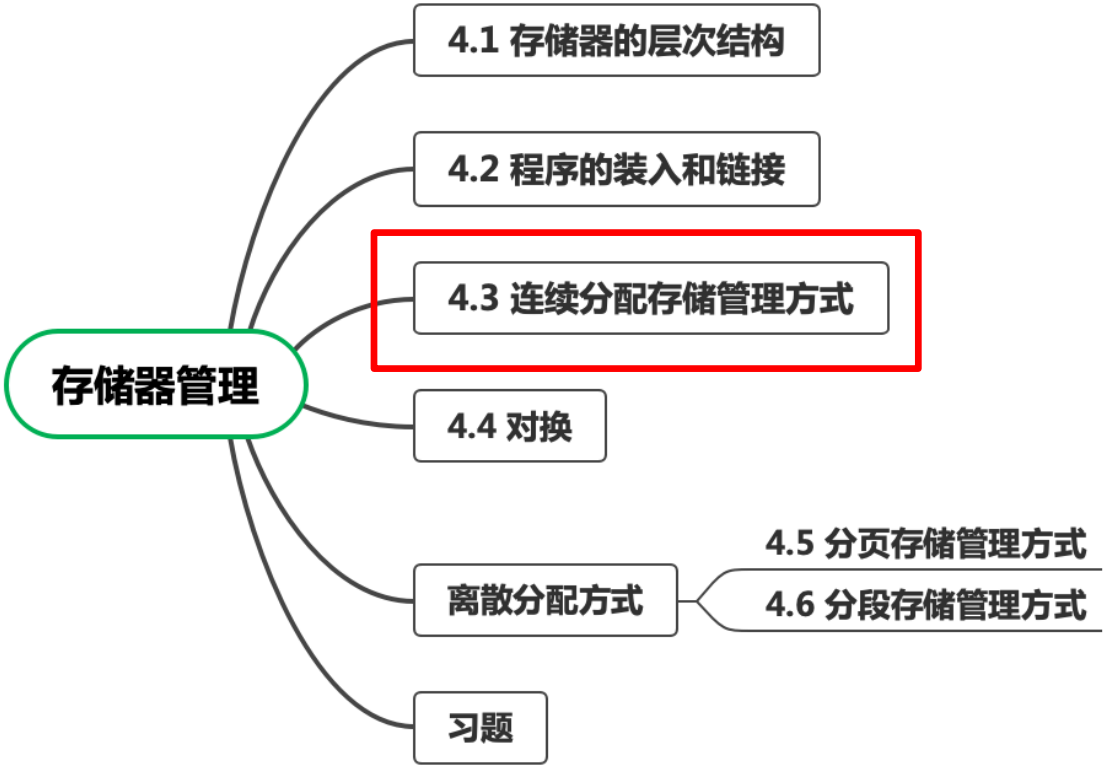
# 4.2 程序的装入和链接




1. 静态链接方式
   (Static Linking)
2. 装入时动态链接
   (Load-time Dynamic Linking)
3. 运行时动态链接
   (Run-time Dynamic Linking)

1. 绝对装入方式
   (Absolute Loading Mode)
2. 可重定位装入方式（静态重定位）
   (Relocation Loading Mode)
3. 动态运行时的装入方式
   (Dynamic Run-time Loading)

# 第四章　存储器管理

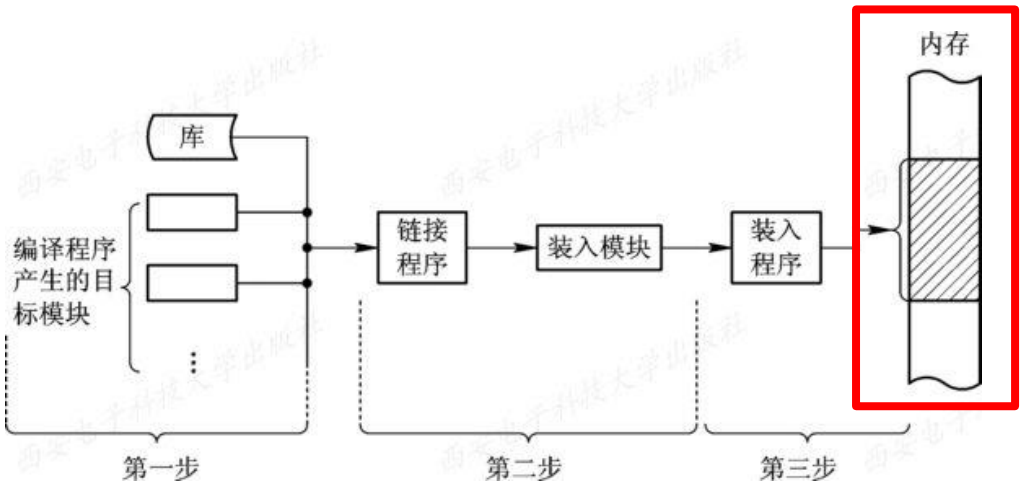

存储器管理
4.1 存储器的层次结构
4.2 程序的装入和链接
4.3 连续分配存储管理方式
4.4 对换
离散分配方式
4.5 分页存储管理方式
4.6 分段存储管理方式
习题

# 4.3 连续分配存储管理方式

该分配方式为一个用户程序分配一个连续的内存空间，即程序中的代码或者数据的逻辑地址相邻。

可分为四类：

- 单一连续分配
- 固定分区分配
- 动态分区分配
- 动态可重定位分区分配



# 4.3 连续分配存储管理方式

### 4.3.1 单一连续分配

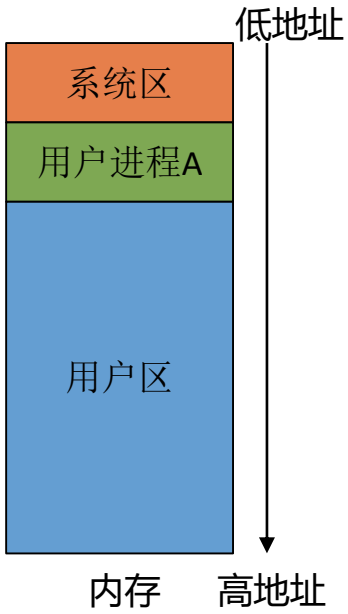
在单道程序环境下，当时的存储器管理方式是把内存分为系统区和用户区两部分，系统区仅提供给OS使用，它通常是放在内存的低址部分。

而在用户区内存中，仅装有一道用户程序，即整个内存的用户空间由该程序独占。

这样的存储器分配方式被称为单一连续分配方式。

如：单道批处理系统。

此时，仅有一道用户程序，不存在干扰，出错后重启容易恢复系统。



# 4.3 连续分配存储管理方式


### 4.3.2 固定分区分配

**面向多道程序系统。如：将用户空间划分为若干个固定大小的区域，在每个分区内只装入一道作业。**

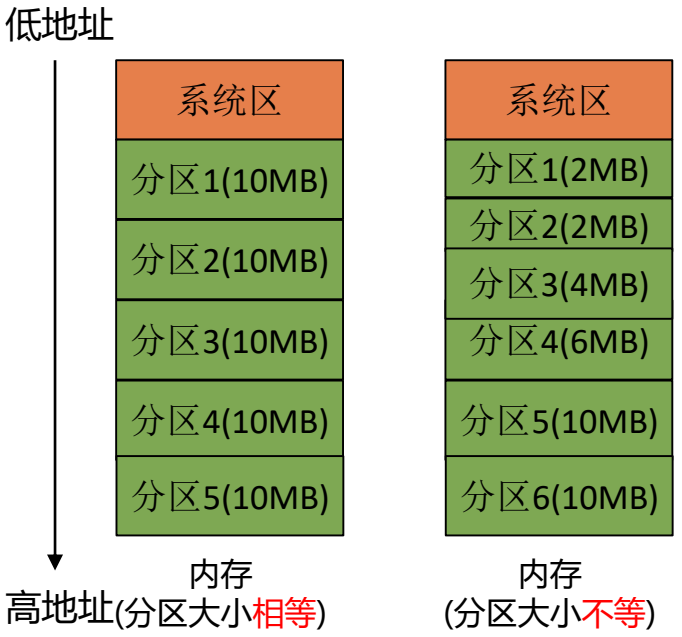
**1. 划分分区的方法**

可用下述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：

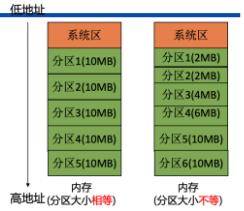
(1) 分区大小相等(指所有的内存分区大小相等)。

大作业不够用，小作业浪费。

(2) 分区大小不等。



# 4.3 连续分配存储管理方式




## 4.3.2 固定分区分配

面向多道程序系统。

由于每个分区大小固定，必然存在浪费的情况。

| 分区号 | 大小（MB） | 起址(M) | 状态 |
|---|---|---|---|
| 1 | 2 | 20 | 已分配 |
| 2 | 2 | 22 | 已分配 |
| 3 | 4 | 24 | 未分配 |
| 4 | 6 | 28 | 已分配 |

分区说明表

| 空间 | 操作系统 |
|---|---|
| 2MB | 作业A |
| 2MB | 作业B |
| 4MB | 无 |
| 6MB | 作业D |
| … | … |

存储空间分配情况

记录方式


内存
(分区大小不等)

# 4.3 连续分配存储管理方式

## 4.3.3 动态分区分配

**又叫：可变分区分配。**

**涉及：数据结构、分区分配算法、分区的分配与回收**


内存
(分区大小不等)

问题1：系统用什么样的数据结构记录内存的使用情况?

问题2：当很多个空闲分区都能满足需求时，应该选择哪个分区进行分配?

问题3：如何分配与回收?

# 4.3 连续分配存储管理方式


## 4.3.3 动态分区分配

**又叫：可变分区分配。**

**涉及：数据结构、分区分配算法、分区的分配与回收**

问题1：系统用什么样的数据结构记录内存的使用情况？

**1. 动态分区分配中的数据结构**

常用的数据结构有以下两种形式：

① 空闲分区表。

在系统中设置一张空闲分区表，用于记录每个空闲分区的情况。每个空闲分区占一个表目，表目中包括分区号、分区大小和分区始址等数据项。

| 分区号 | 分区大小(KB) | 分区始址(K) | 状态 |
|---|---|---|---|
| 1 | 50 | 85 | 空闲 |
| 2 | 32 | 155 | 空闲 |
| 3 | 70 | 275 | 空闲 |
| 4 | 60 | 532 | 空闲 |
| 5 | … | … | … |

# 4.3 连续分配存储管理方式

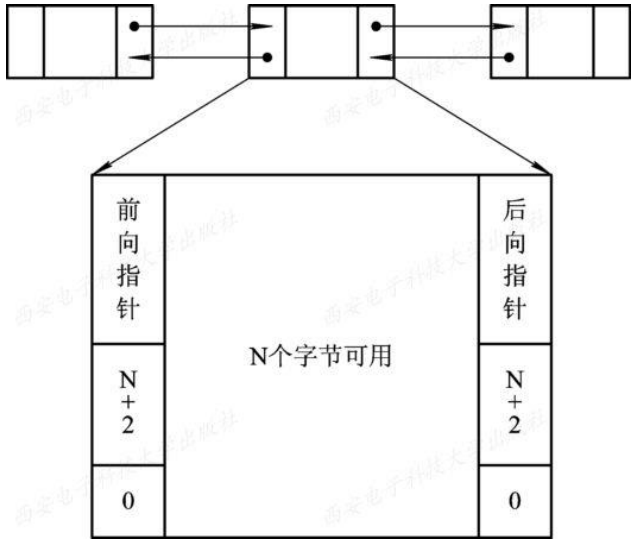

## 4.3.3 动态分区分配

**又叫：可变分区分配。**

**涉及：数据结构、分区分配算法、分区的分配与回收**

**1. 动态分区分配中的数据结构**

问题1：系统用什么样的数据结构记录内存的使用情况?




② 空闲分区链。

为了实现对空闲分区的分配和链接，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。通过前、后向链接指针，可将所有的空闲分区链接成一个双向链。

# 4.3 连续分配存储管理方式

问题2：当很多个空闲分区都能满足需求时，应该选择哪个分区进行分配？

## 4.3.3 动态分区分配

### 2. 动态分区分配算法

为把一个新作业装入内存，须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。由于内存分配算法对系统性能有很大的影响，故人们对它进行了较为广泛而深入的研究，于是产生了许多动态分区分配算法。

基于顺序搜索的动态分区分配算法

基于索引搜索的动态分区分配算法

动态可重定位分区分配

。。。。。。

# 4.3 连续分配存储管理方式

### 4.3.3 动态分区分配

**3. 分区分配操作**

分配内存+回收内存

问题3：如何分配与回收？

# 4.3 连续分配存储管理方式

问题3：如何分配与回收?

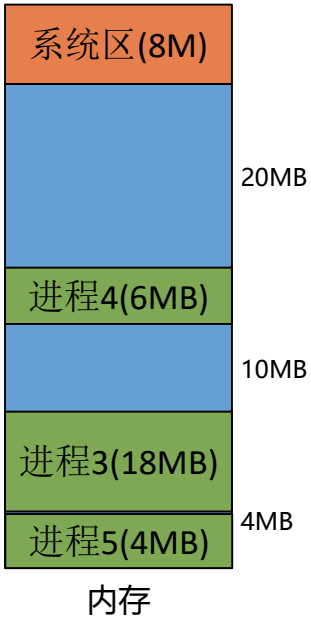
## 4.3.3 动态分区分配

### 3. 分区分配操作

分配内存+回收内存

1) 分配内存

系统应利用某种分配算法，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可表示为m.size。




| 分区号 | 分区大小(MB) | 起始地址（M） | 状态 |
|---|---|---|---|
| 1 | 20 | 8 | 空闲 |
| 2 | 10 | 34 | 空闲 |
| 3 | 4 | 62 | 空闲 |

| 分区号 | 分区大小(MB) | 起始地址（M） | 状态 |
|---|---|---|---|
| 1 | 20 | 8 | 空闲 |
| 2 | 10 | 34 | 空闲 |

# 4.3 连续分配存储管理方式

问题3：如何分配与回收？

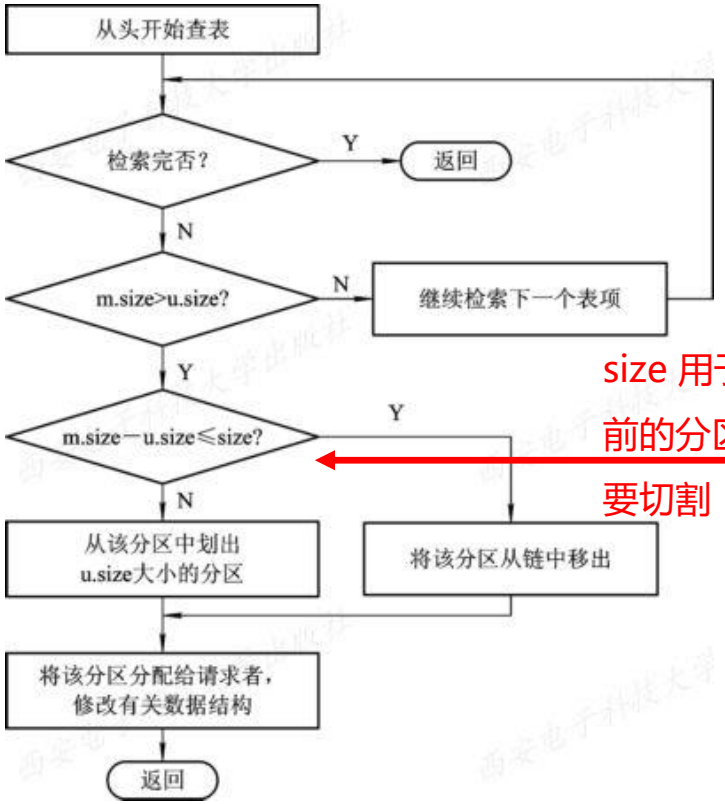
## 4.3.3 动态分区分配

### 3. 分区分配操作

分配内存+回收内存

1) 分配内存

系统应利用某种分配算法，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可表示为m.size。




size 用于判定当前的分区是否需要切割

# 4.3 连续分配存储管理方式

问题3：如何分配与回收？

2) 回收内存

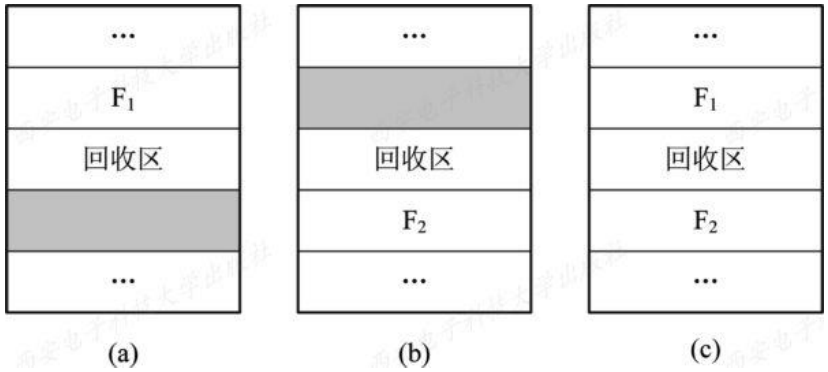
当进程运行完毕释放内存时，系统根据回收区的首址，从空闲区链(表)中找到相应的插入点，此时可能出现以下四种情况之一：

(1) 回收区与插入点的前一个空闲分区$F_1$相邻接。

(2) 回收分区与插入点的后一空闲分区$F_2$相邻接。

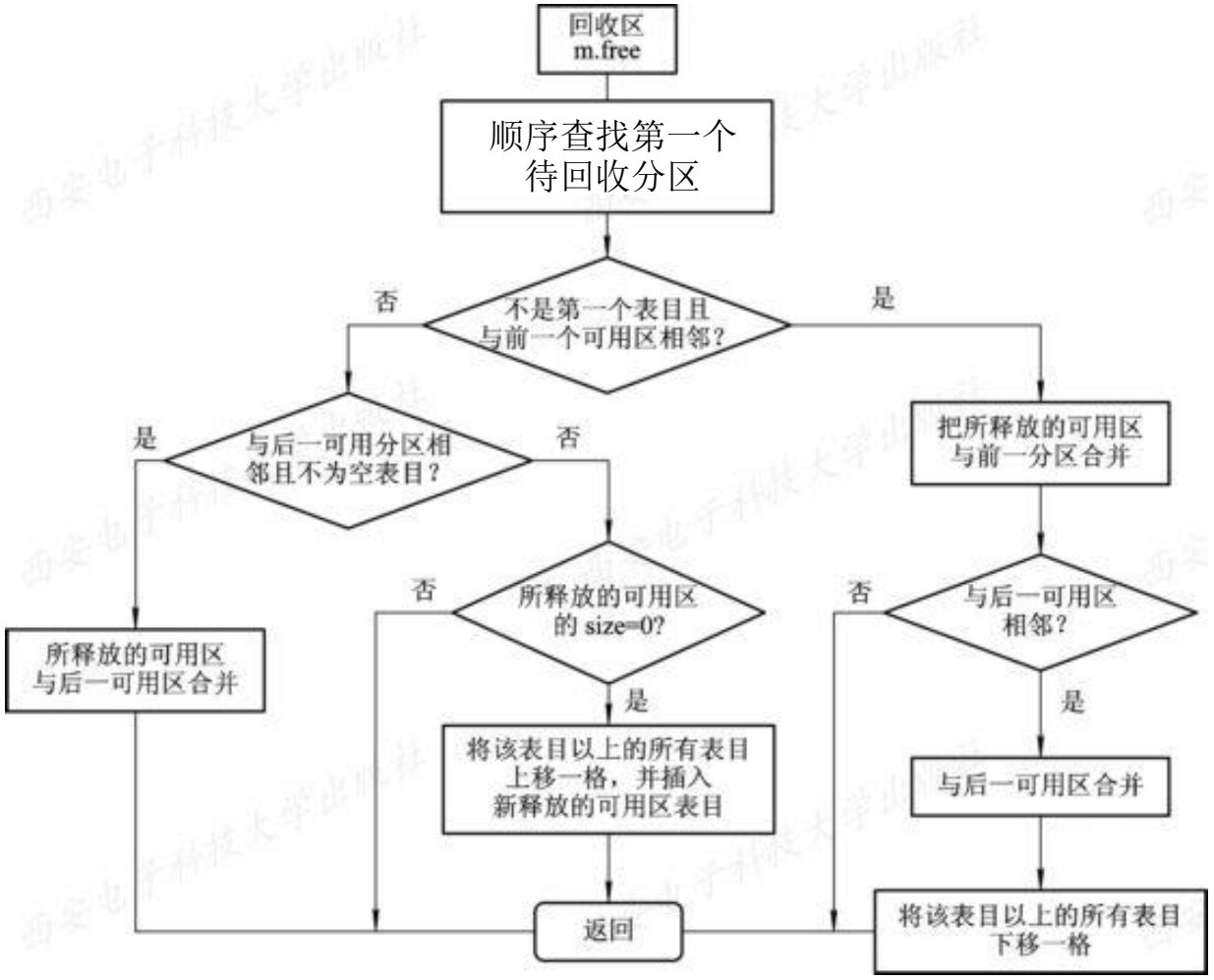
(3) 回收区同时与插入点的前、后两个分区邻接。

(4) 回收区既不与$F_1$邻接，又不与$F_2$邻接。

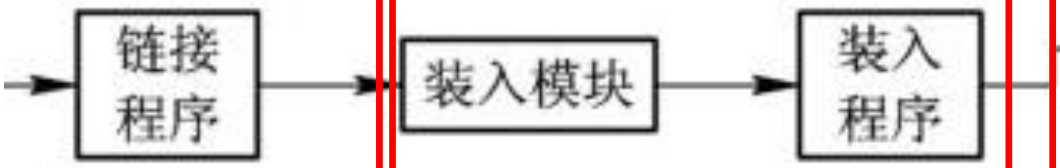


# 4.3 连续分配存储管理方式

2) 回收内存流程

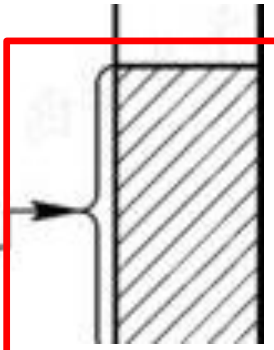


# 4.3 连续分配存储管理方式




1. 静态链接方式
   (Static Linking)
2. 装入时动态链接
   (Load-time Dynamic Linking)
3. 运行时动态链接
   (Run-time Dynamic Linking)

1. 绝对装入方式
   (Absolute Loading Mode)
2. 可重定位装入方式（静态重定位）
   (Relocation Loading Mode)
3. 动态运行时的装入方式
   (Dynamic Run-time Loading)



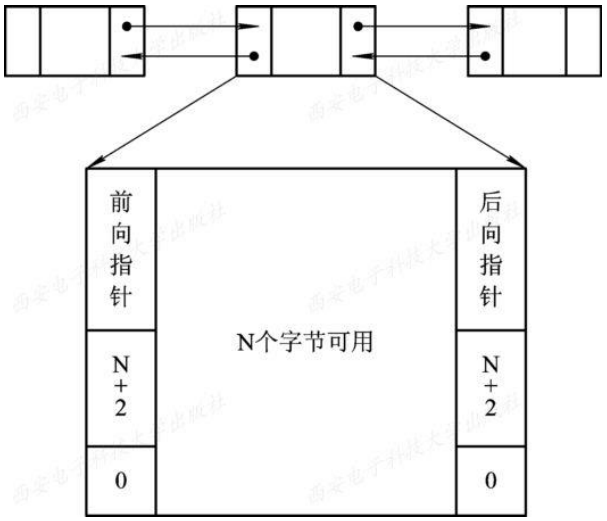
1. 单一连续分配
2. 固定分区分配
3. 动态分区分配

# 4.3 连续分配存储管理方式

## 4.3.4 基于顺序搜索的动态分区分配算法

**将空闲分区链成链表，顺序搜索链表。**

| 分区号 | 分区大小(KB) | 分区始址(K) | 状态 |
|---|---|---|---|
| 1 | 50 | 85 | 空闲 |
| 2 | 32 | 155 | 空闲 |
| 3 | 70 | 275 | 空闲 |
| 4 | 60 | 532 | 空闲 |
| 5 | … | … | … |




| 1. 首次适应算法 | 2.循环首次适应算法 | 3.最佳适应算法 | 4.最坏适用算法 |
|---|---|---|---|
| • First Fit, FF | • Next Fit, NF | • Best Fit, BF | • Worst Fit, WF |

# 4.3 连续分配存储管理方式

优点：小空间在低地址部分，大空间在高地址部分

缺点：低址部分被多次分割，导致有碎片。

### 4.3.4 基于顺序搜索的动态分区分配算法

**1. 首次适应(first fit，FF)算法**

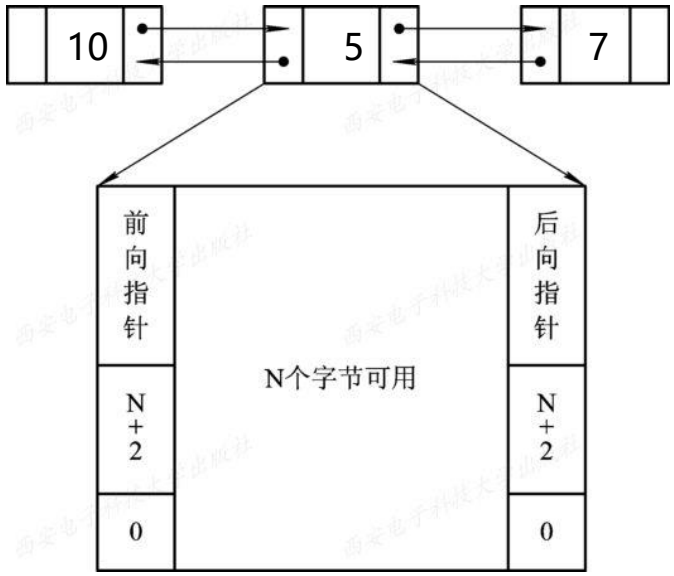
我们以空闲分区链为例来说明采用FF算法时的分配情况。

① FF算法要求空闲分区链以**地址递增**的次序链接。

② 在分配内存时，从链首开始顺序查找，直至找到一个大小能满足要求的空闲分区为止。

③ 然后再按照作业的大小，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链中。

④若从链首直至链尾都不能找到一个能满足要求的分区，则表明系统中已没有足够大的内存分配给该进程，内存分配失败，返回。



# 4.3 连续分配存储管理方式

优点：小空间在低地址部分，大空间在高地址部分

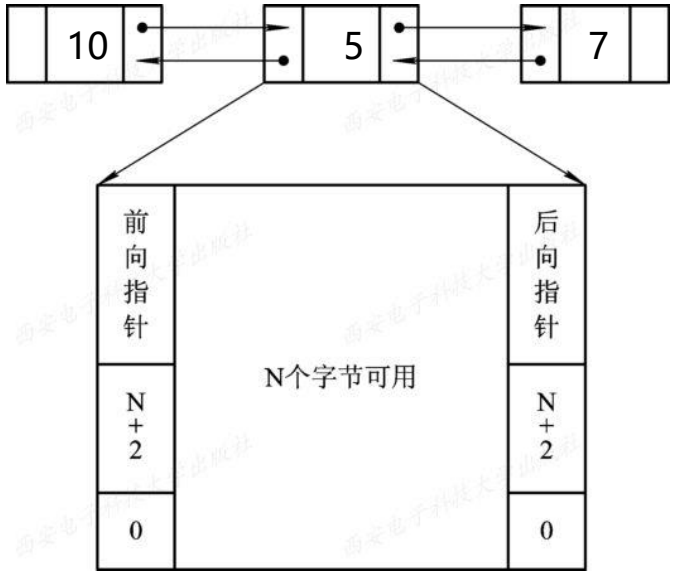
缺点：低址部分被多次分割，导致有碎片。

## 4.3.4 基于顺序搜索的动态分区分配算法

### 1. 首次适应(first fit，FF)算法

空间需求：6　5　4

分配结果：10　5　7　→　0　0　7



# 4.3 连续分配存储管理方式

### 4.3.4 基于顺序搜索的动态分区分配算法

**2. 循环首次适应(next fit，NF)算法**

① NF算法要求空闲分区链以**地址递增**的次序链接。

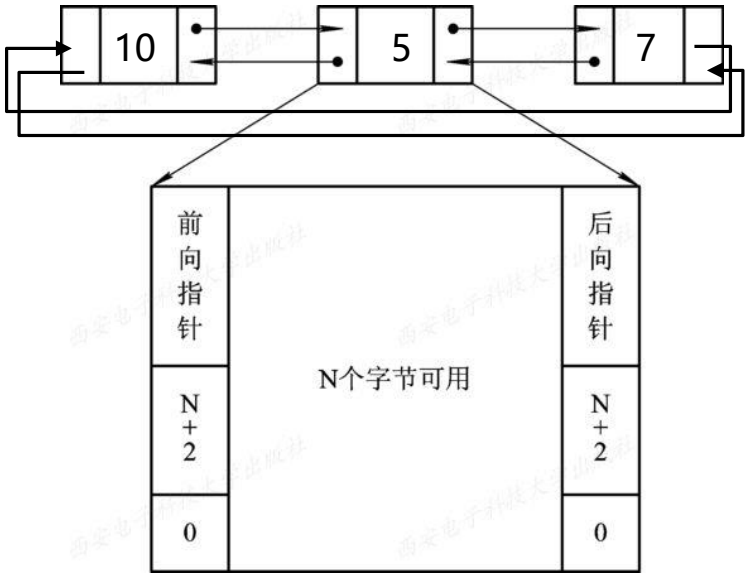
② 在分配内存时，从**上次找到的空闲分区的下一个空闲分区**开始顺序查找，直至找到一个大小能满足要求的空闲分区为止。

③ 然后再按照作业的大小，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链中。

④若从链首直至链尾都不能找到一个能满足要求的分区，则表明系统中已没有足够大的内存分配给该进程，内存分配失败，返回。

优点：空闲分区分布均匀

缺点：缺乏大的空闲分区



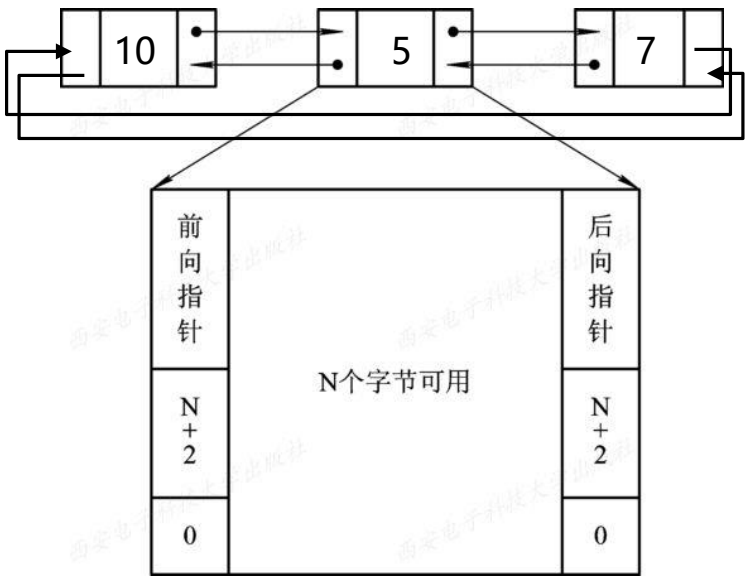
# 4.3 连续分配存储管理方式

优点：空闲分区分布均匀

缺点：缺乏大的空闲分区

### 4.3.4 基于顺序搜索的动态分区分配算法

#### 2. 循环首次适应(next fit，NF)算法




空间需求：6　　5　　4

FF分配结果：10　5　7　→　0　0　7

NF分配结果：10　5　7　→　4　0　3

# 4.3 连续分配存储管理方式

### 4.3.4 基于顺序搜索的动态分区分配算法

**3. 最佳适应(best fit，BF)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”。

① BF算法要求空闲分区链以**容量递增**的次序链接。

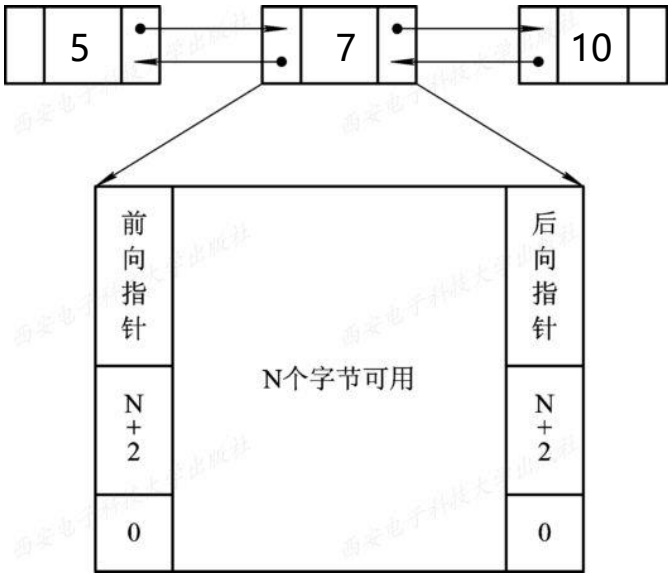
② 在分配内存时，从**链首**开始顺序查找，直至找到一个大小能满足要求的空闲分区为止。

③ 划分区域

④ 若无，内存分配失败，返回。

优点：避免大材小用

缺点：碎片



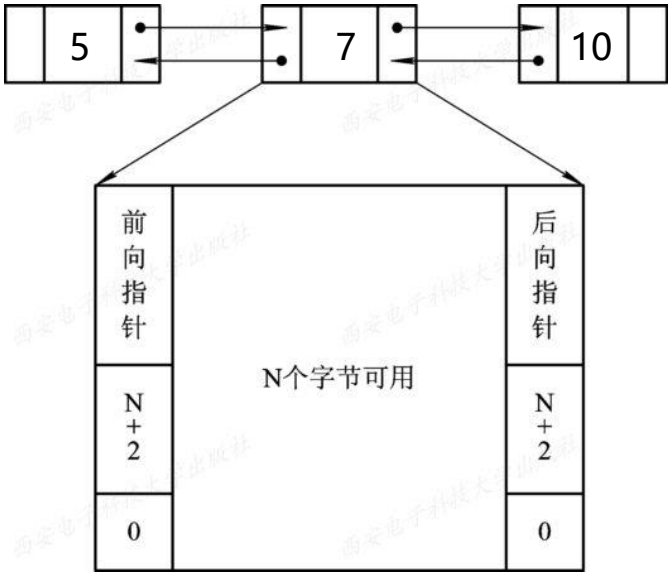
# 4.3 连续分配存储管理方式

### 4.3.4 基于顺序搜索的动态分区分配算法

#### 3. 最佳适应(best fit，BF)算法

优点：避免大材小用

缺点：碎片




空间需求：6　5　4

FF分配结果：10　5　7　→ 0　0　7

NF分配结果：10　5　7　→ 4　0　3

BF分配结果：　5　7　10 → 0　1　6

# 4.3 连续分配存储管理方式

优点：少碎片，快速查找到分区（第一次即可查找到相应分区）
缺点：缺乏大的空闲分区

### 4.3.4 基于顺序搜索的动态分区分配算法

#### 4. 最坏适应(worst fit，WF)算法

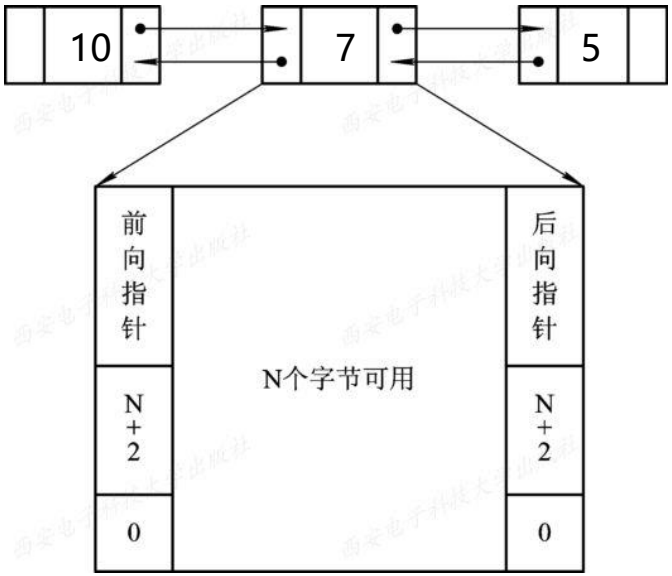
由于最坏适应分配算法选择空闲分区的策略正好与最佳适应算法相反：它在扫描整个空闲分区表或链表时，总是挑选一个最大的空闲区，从中分割一部分存储空间给作业使用，以至于存储器中缺乏大的空闲分区，故把它称为是最坏适应算法。

① WF算法要求空闲分区链以**容量递减**的次序链接。

② 在分配内存时，从**链首**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。

③ 划分区域

④ 若无，内存分配失败，返回。



# 4.3 连续分配存储管理方式

## 4.3.4 基于顺序搜索的动态分区分配算法

### 4. 最坏适应(worst fit，WF)算法

优点：少碎片，快速查找到分区（第一次即可查找到相应分区）

缺点：缺乏大的空闲分区

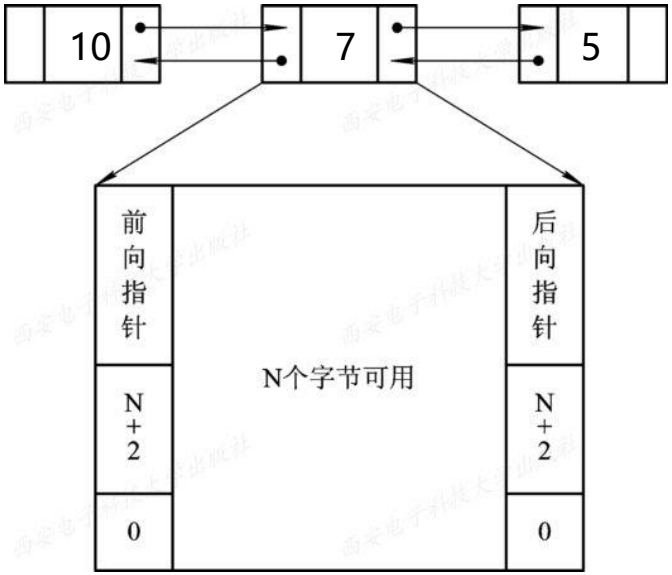
空间需求：6　5　4

FF分配结果：10　5　7　→　0　0　7

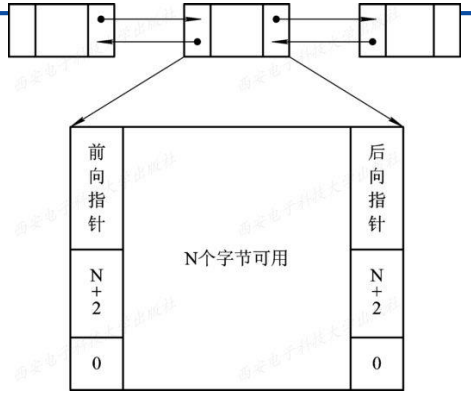
NF分配结果：10　5　7　→　4　0　3

BF分配结果：　5　7　10　→　0　1　6

WF分配结果：　10　7　5　→　4　2　1



4.3 连续分配存储管理方式




4.3.4 基于顺序搜索的动态分区分配算法

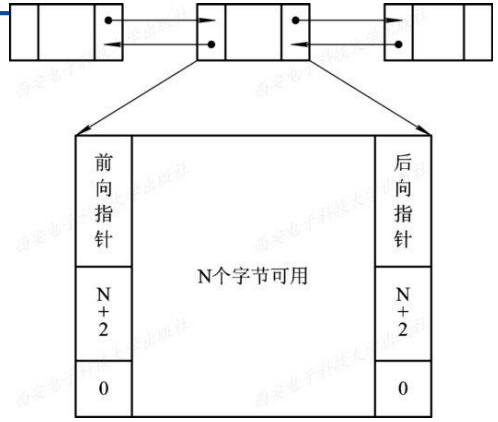
将空闲分区链成链表，顺序搜索链表。

找到则划分；找不到则失败返回。

| 1. 首次适应算法 | 2.循环首次适应算法 | 3.最佳适应算法 | 4.最坏适应算法 |
|---|---|---|---|
| • First Fit, FF<br>• 链表按地址顺序递增<br>• 从链首开始查找 | • Next Fit, NF<br>• 链表按地址顺序递增<br>• 上次找到的空闲分区的下一个空闲分区开始查找 | • Best Fit, BF<br>• 链表按容量大小递增<br>• 链首 | • Worst Fit, WF<br>• 链表按容量大小递减<br>• 链首 |

# 4.3 连续分配存储管理方式

## 4.3.4 基于顺序搜索的动态分区分配算法




### 1. 首次适应算法

- First Fit, FF
- 链表按地址顺序递增
- 从链首开始查找

优点：小空间在低地址部分，大空间在高地址部分
缺点：低址部分被多次分割，导致有碎片。

### 2.循环首次适应算法

- Next Fit, NF
- 链表按地址顺序递增
- 上次找到的空闲分区的下一个空闲分区开始查找

优点：空闲分区分布均匀
缺点：缺乏大的空闲分区

### 3.最佳适应算法

- Best Fit, BF
- 链表按容量大小递增
- 链首

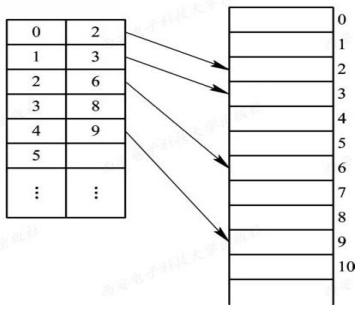
优点：避免大材小用
缺点：碎片

### 4.最坏适应算法

- Worst Fit, WF
- 链表按容量大小递减
- 链首

优点：少碎片，快速查找到分区（第一次即可查找到相应分区）
缺点：缺乏大的空闲分区

# 4.3 连续分配存储管理方式




## 4.3.4 基于索引搜索的动态分区分配算法：

**速度快，适用大中型操作系统**

| 1. 快速适应算法 | 2.伙伴系统 | 3.哈希算法 |
| --- | --- | --- |
| • Quick Fit，又称：分类搜索算法 | • Buddy System<br>• 多个空闲分区链（每个链中容量一致，$2^k$） | • 哈希表 |

# 4.3 连续分配存储管理方式

优点：查找效率高，少碎片

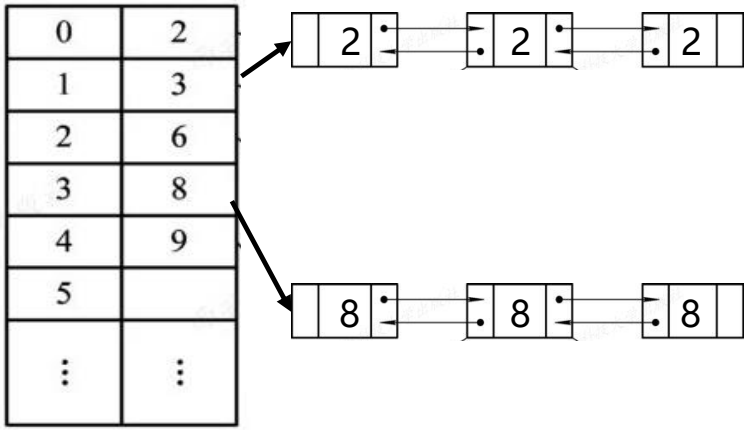
缺点：为有效的合并分区，产生额外系统开销

### 4.3.5 基于索引搜索的动态分区分配算法

#### 1. 快速适应(quick fit)算法

该算法又称为分类搜索法，是将空闲分区根据其容量大小进行分类，对于每一类具有相同容量的所有空闲分区，单独设立一个空闲分区链表，这样系统中存在多个空闲分区链表。

同时，在内存中设立一张管理索引表，其中的每一个索引表项对应了一种空闲分区类型，并记录了该类型空闲分区链表表头的指针。



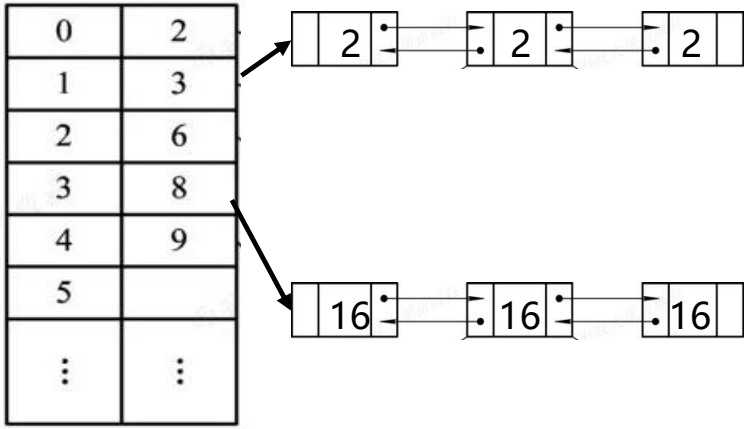
# 4.3 连续分配存储管理方式

### 4.3.5 基于索引搜索的动态分区分配算法

**2. 伙伴系统(buddy system)**

该算法规定，无论已分配分区或空闲分区，其大小均为2的k次幂(k为整数，l≤k≤m)。通常$2^m$是整个可分配内存的大小(也就是最大分区的大小)。

按照空闲分区容量建立双向链表。




① 进程请求n个容量空间时，计算 i，使$2^{i-1}<n\le 2^i$;

② 若找到则分配;

③ 若找不到，则$2^i$的空间分配完毕;

拆解$2^{i+1}$的空间; （伴随链表更改）

④ 依次类推。。。

在伙伴系统中，对于一个大小为$2^k$，地址为x的内存块，其伙伴块的地址则用$buddy_k(x)$表示，其通式为：

# 4.3 连续分配存储管理方式

优点：减少空闲分区

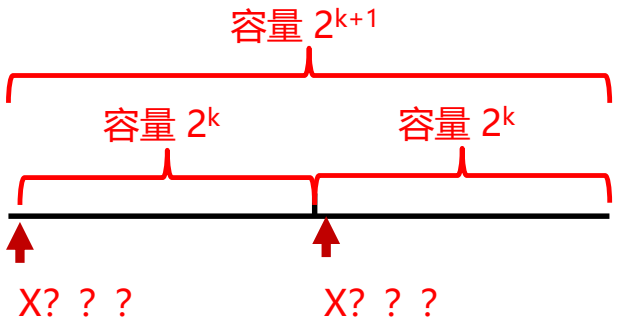
缺点：分配与回收均伴随链表的改变

### 4.3.5 基于索引搜索的动态分区分配算法

**2. 伙伴系统(buddy system)**

该算法规定，无论已分配分区或空闲分区，其大小均为2的k次幂(k为整数，l≤k≤m)。通常$2^m$是整个可分配内存的大小(也就是最大分区的大小)。

$$buddy_k(x)=\begin{cases} x+2^k\ (若\ xMOD\ \ 2^{k+1}=0) \\ x-2^k\ (若\ xMOD\ \ 2^{k+1}=2^k) \end{cases}$$



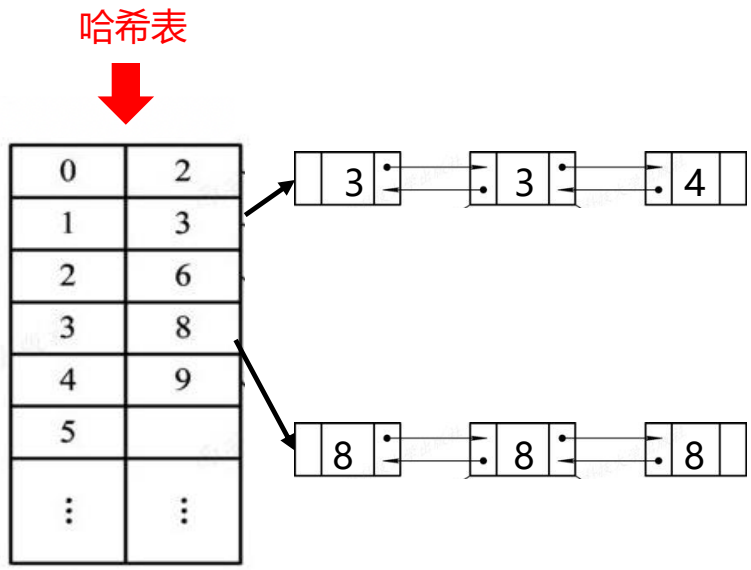

在伙伴系统中，对于一个大小为$2^k$，地址为x的内存块，其伙伴块的地址则用$buddy_k(x)$表示，其通式为：

# 4.3 连续分配存储管理方式

### 4.3.5 基于索引搜索的动态分区分配算法

**3. 哈希算法**

在上述的分类搜索算法和伙伴系统算法中，都是将空闲分区根据分区大小进行分类，对于每一类具有相同大小的空闲分区，单独设立一个空闲分区链表。如果对空闲分区分类较细，则相应索引表的表项也就较多，因此会显著地增加搜索索引表的表项的时间开销。

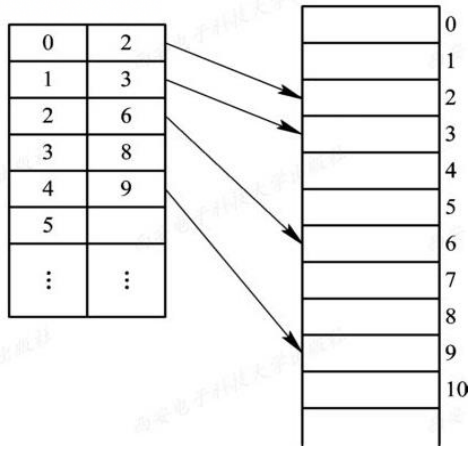



依据索引规律，构建一张以空闲分区大小为关键字的哈希表，每个表项指向链首指针。

# 4.3 连续分配存储管理方式

## 4.3.4 基于索引搜索的动态分区分配算法：

**速度快，适用大中型操作系统**




### 1. 快速适应算法

- Quick Fit，又称：分类搜索算法
- 空闲分区链表
- 管理索引表

优点：查找效率高，少碎片

缺点：为有效的合并分区，产生额外系统开销

### 2.伙伴系统

- Buddy System
- 多个空闲分区链（每个链中容量一致，$2^k$）

优点：减少空闲分区

缺点：分配与回收均伴随链表的改变

### 3.哈希算法

- 哈希表

优点：速度快

缺点：碎片

# 4.3 连续分配存储管理方式

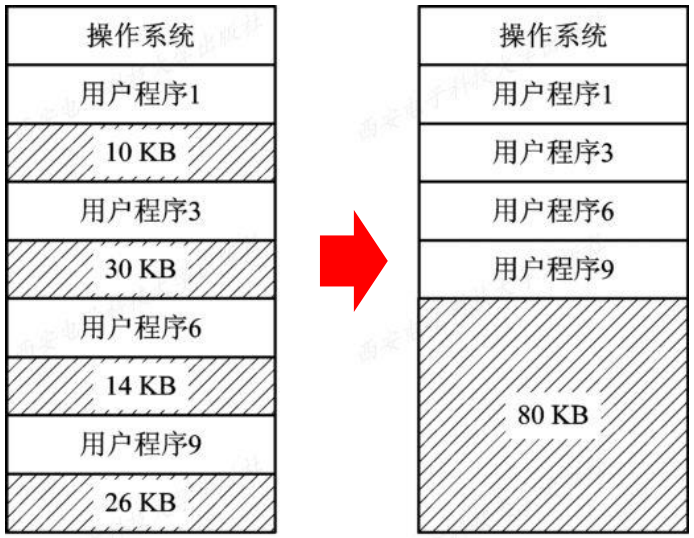
### 4.3.6 动态可重定位分区分配

**1. 紧凑**

连续分配方式的一个重要特点是，一个系统或用户程序必须被装入一片连续的内存空间中。

当一台计算机运行了一段时间后，它的内存空间将会被分割成许多小的分区，而缺乏大的空闲空间。

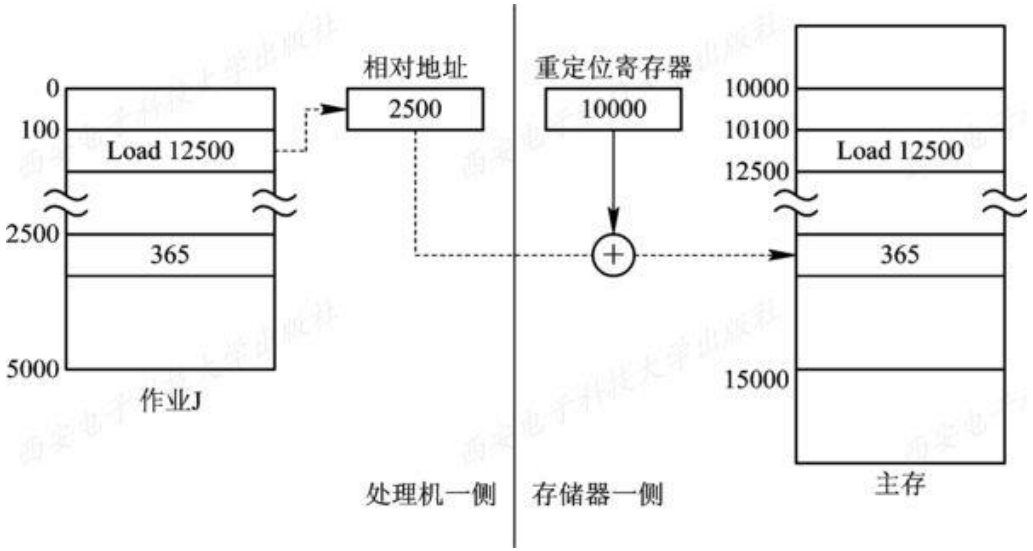
即使这些分散的许多小分区的容量总和大于要装入的程序，但由于这些分区不相邻接，也无法把该程序装入内存。




(a) 紧凑前　　(b) 紧凑后

# 4.3 连续分配存储管理方式

## 4.3.6 动态可重定位分区分配

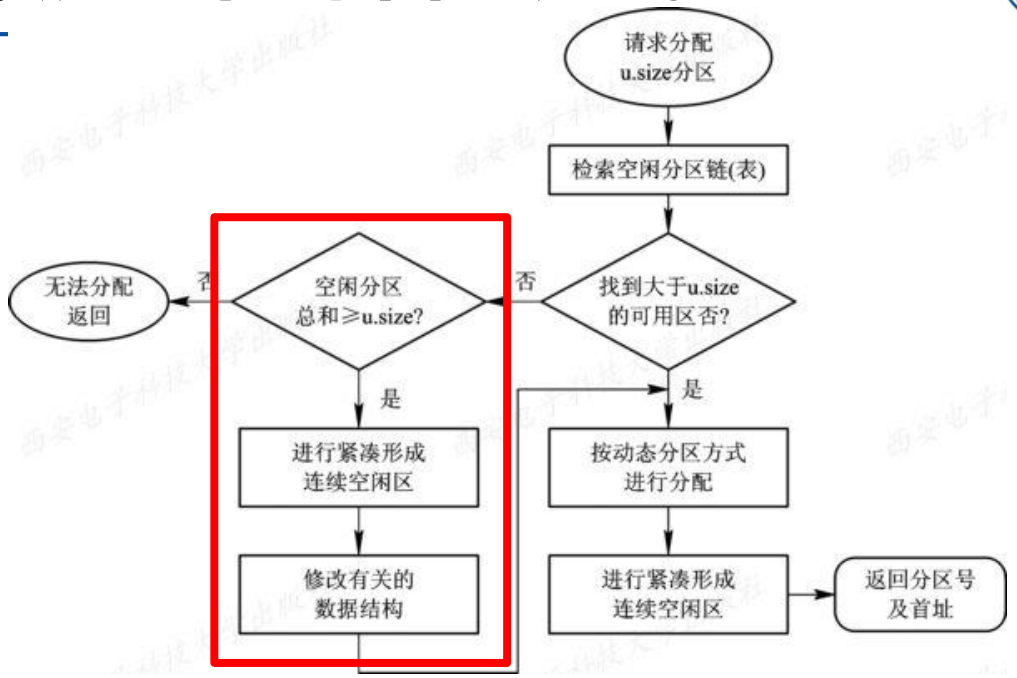



### 2. 动态重定位

在4.2.1节中所介绍的动态运行时装入的方式中，作业装入内存后的所有地址仍然都是相对(逻辑)地址。而将相对地址转换为绝对(物理)地址的工作被推迟到程序指令要真正执行时进行。为使地址的转换不会影响到指令的执行速度，必须有硬件地址变换机构的支持，即须在系统中增设一个重定位寄存器，用它来存放程序(数据)在内存中的起始地址。程序在执行时，真正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形成的。

# 4.3 连续分配存储管理方式

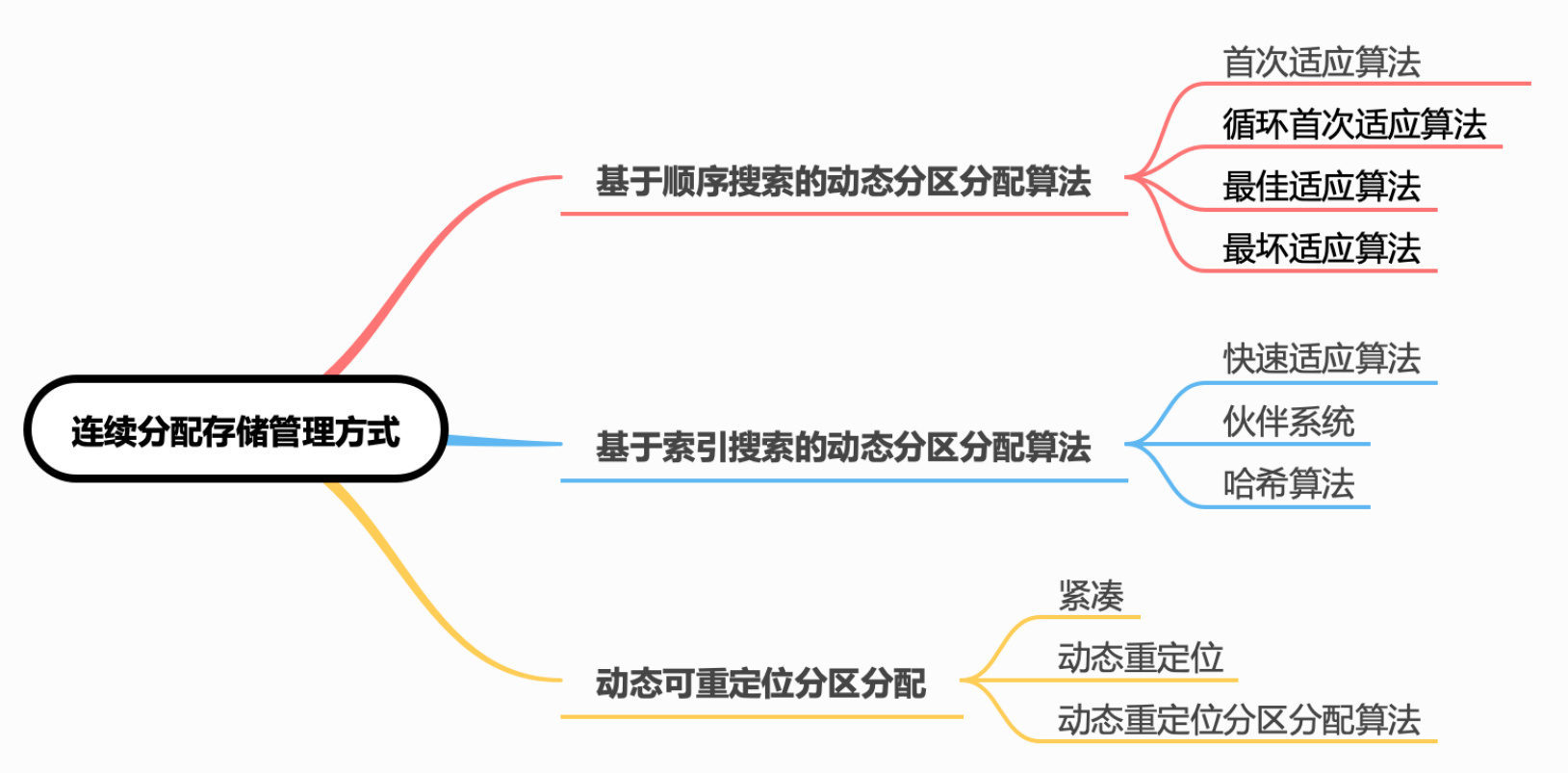
## 4.3.6 动态可重定位分区分配



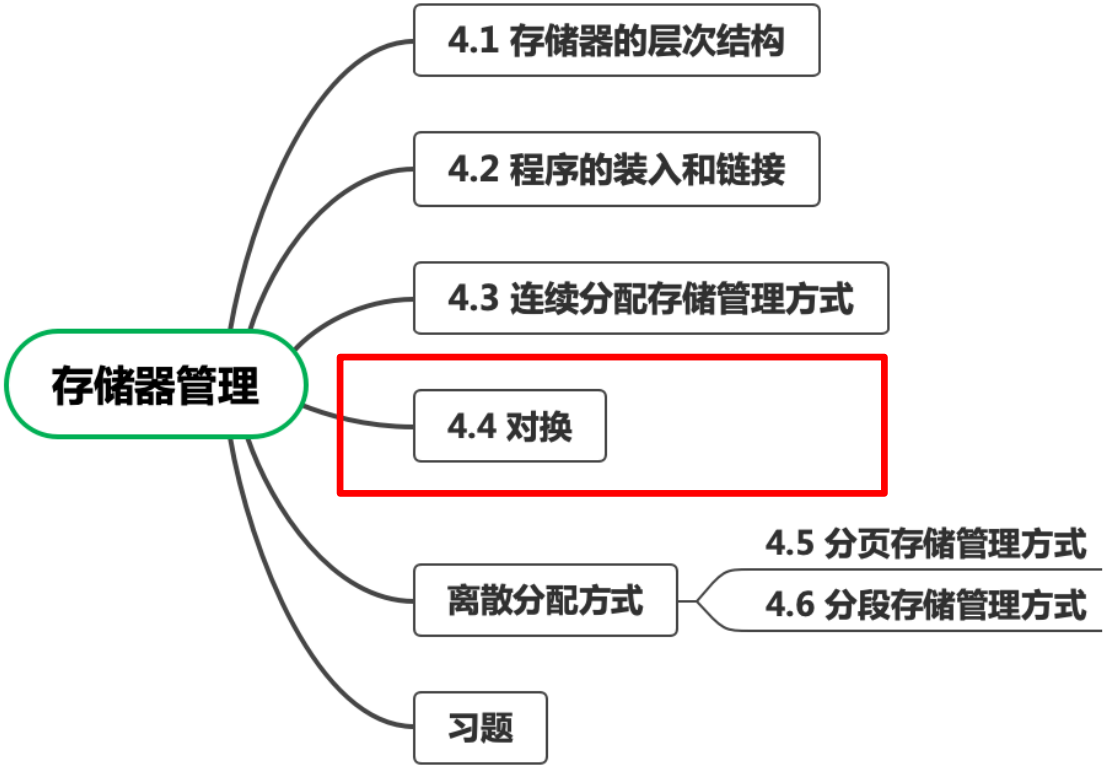

### 3. 动态重定位分区分配算法

动态重定位分区分配算法与动态分区分配算法基本上相同，差别仅在于：在这种分配算法中，增加了紧凑的功能。通常，当该算法不能找到一个足够大的空闲分区以满足用户需求时，如果所有的小的空闲分区的容量总和大于用户的要求，这时便须对内存进行“紧凑”，将经“紧凑”后所得到的大空闲分区分配给用户。如果所有的小的空闲分区的容量总和仍小于用户的要求，则返回分配失败信息。

# 4.3 连续分配存储管理方式



# 第四章　存储器管理


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 连续分配存储管理方式
4.4 对换
离散分配方式
4.5 分页存储管理方式
4.6 分段存储管理方式
习题

# 4.4 对换(Swapping)

对换技术也称为交换技术，最早用于麻省理工学院的单用户分时系统CTSS中。由于当时计算机的内存都非常小，为了使该系统能分时运行多个用户程序而引入了对换技术。

系统把所有的用户作业存放在磁盘上，每次只能调入一个作业进入内存，当该作业的一个时间片用完时，将它调至外存的后备队列上等待，再从后备队列上将另一个作业调入内存。

这就是最早出现的分时系统中所用的对换技术。现在已经很少使用。

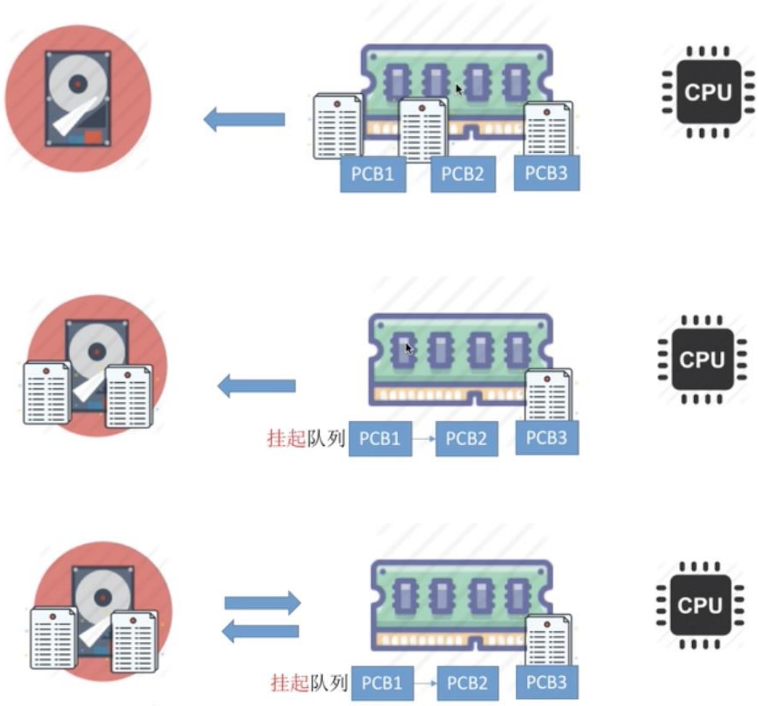
# 4.4 对换(Swapping)

### 4.4.1 多道程序环境下的对换技术

#### 1. 对换的引入

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换到外存上，以便腾出足够的内存空间，再把已具备运行条件的进程或者进程所需要的程序和数据换入内存。

Unix 与 Windows 操作系统均具有对换功能。



# 4.4 对换(Swapping)

### 4.4.1 多道程序环境下的对换技术

**2. 对换的类型**

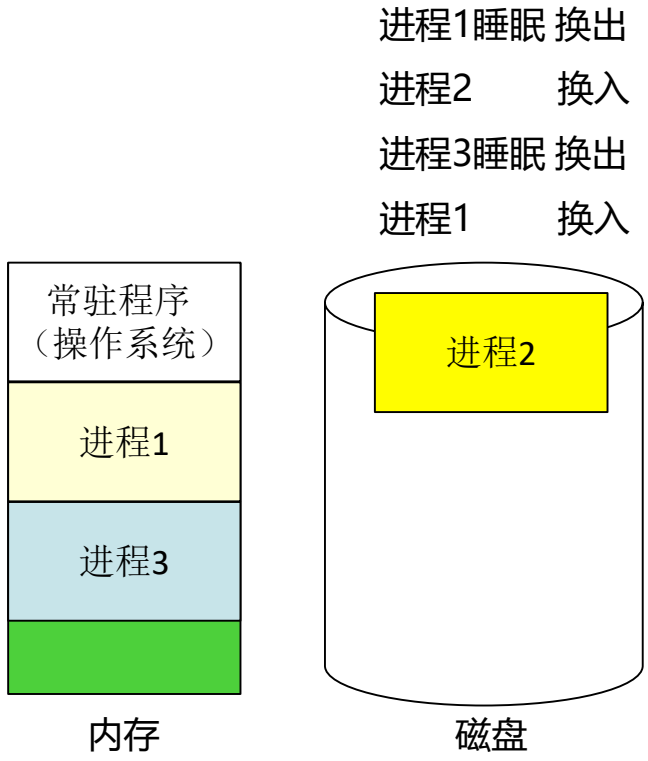
在每次对换时，都是将一定数量的程序或数据换入或换出内存。根据每次对换时所对换的数量，可将对换分为如下两类：

(1) 整体对换。

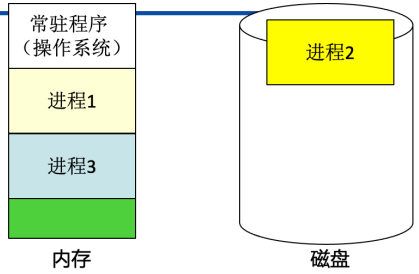
以进程为单位，又叫“进程对换”，中级调度。

(2) 页面(分段)对换。

对对换空间的管理、进程的换出、进程的换入



# 4.4 对换(Swapping)




## 4.4.2 对换空间的管理

### 1. 对换空间管理的主要目标

在具有对换功能的OS中，通常把磁盘空间分为文件区和对换区两部分。


1) 对文件区管理的主要目标。

占用磁盘空间的大部分

先：提高文件存储空间的利用率；

次：提高对文件的访问速度；

离散分配方式

2) 对对换空间管理的主要目标

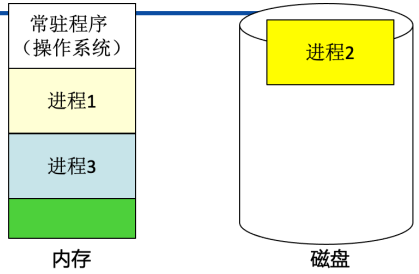
占用磁盘空间的小部分

先：提高对文件的访问速度；

次：提高文件存储空间的利用率；

连续分配方式

# 4.4 对换(Swapping)

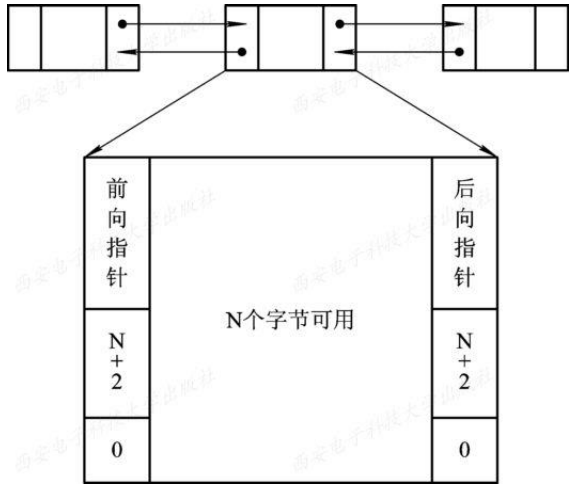



## 4.4.2 对换空间的管理

### 2. 对换区空闲盘块管理中的数据结构

为了实现对对换区中的空闲盘块的管理，在系统中应配置相应的数据结构，用于记录外存对换区中的空闲盘块的使用情况。其数据结构的形式与内存在动态分区分配方式中所用数据结构相似，即同样可以用空闲分区表或空闲分区链。在空闲分区表的每个表目中，应包含两项：对换区的首址及其大小，分别用盘块号和盘块数表示。

| 分区号 | 分区大小(KB) | 分区始址(K) | 状态 |
|---|---|---|---|
| 1 | 50 | 85 | 空闲 |
| 2 | 32 | 155 | 空闲 |
| 3 | 70 | 275 | 空闲 |
| 4 | 60 | 532 | 空闲 |
| 5 | … | … | … |



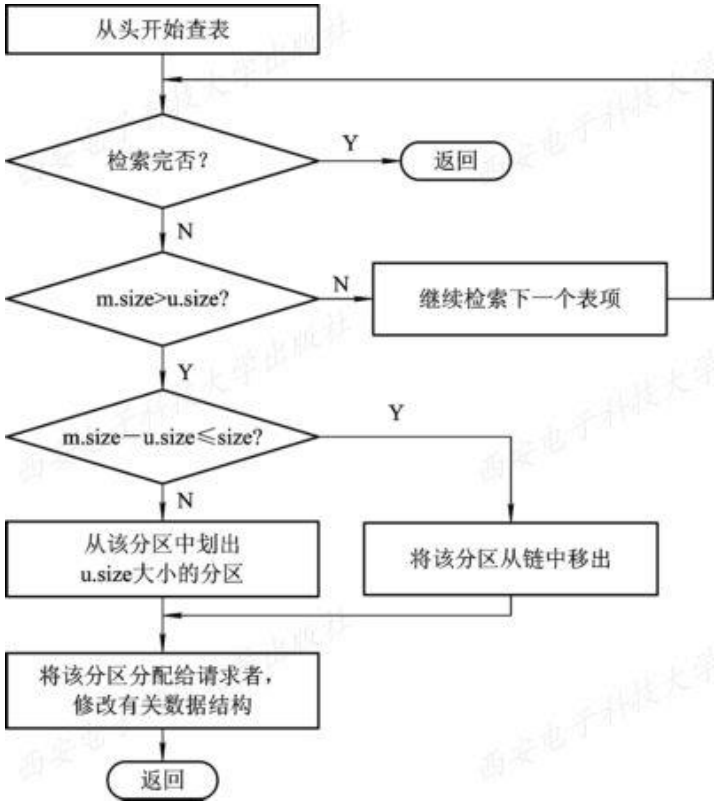
# 4.4 对换(Swapping)

## 4.4.2 对换空间的管理

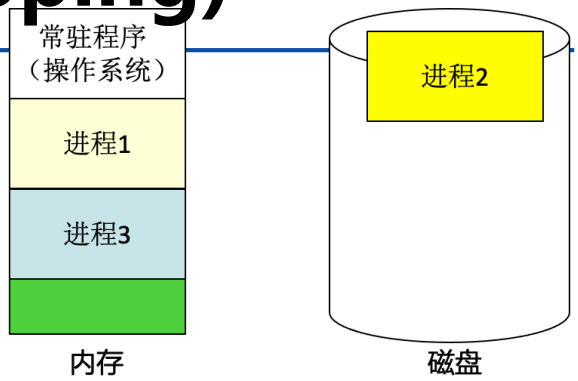
### 3. 对换空间的分配与回收

由于对换分区的分配采用的是连续分配方式，因而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方法雷同。

其分配算法可以是首次适应算法、循环首次适应算法或最佳适应算法等。具体的分配操作也与前述内存的分配过程相同。



# 4.4 对换(Swapping)




### 4.4.3 进程的换出与换入

**1. 进程的换出**

对换进程在实现进程换出时，是将内存中的某些进程调出至对换区，以便腾出内存空间。换出过程可分为以下两步：

(1) 选择被换出的进程。

情况①：堵塞状态或者睡眠状态的进程；

情况②：防止优先级低的进程换入后很快被换出，考虑进程在内存的驻留时间；

情况③：若已无堵塞进程，则选择优先级最低的进程换出。

(2) 进程换出过程。

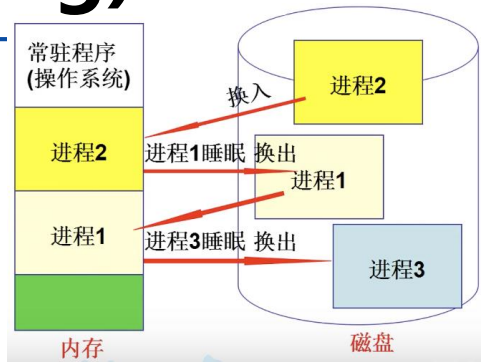
只能换出非共享的程序或数据

步骤①：申请空间

步骤②：启动磁盘，传输数据

步骤③：修改PCB、修改内存分配表

# 4.4 对换(Swapping)




## 4.4.3 进程的换出与换入

**2. 进程的换入**

步骤①：首先查看PCB集合中所有进程的状态，从中找出“就绪”但已换出的进程。当有许多这样的进程时，它将选择其中已换出到磁盘上时间最久(必须大于规定时间，如2s)的进程作为换入进程。

步骤②：申请内存。

步骤③：如果申请成功，可直接将进程从外存调入内存；如果失败，则需先将内存中的某些进程换出，腾出足够的内存空间后，再将进程调入。

# 第四章　存储器管理


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 连续分配存储管理方式
4.4 对换
离散分配方式
4.5 分页存储管理方式
4.6 分段存储管理方式
习题

# 4.5 分页存储管理方式

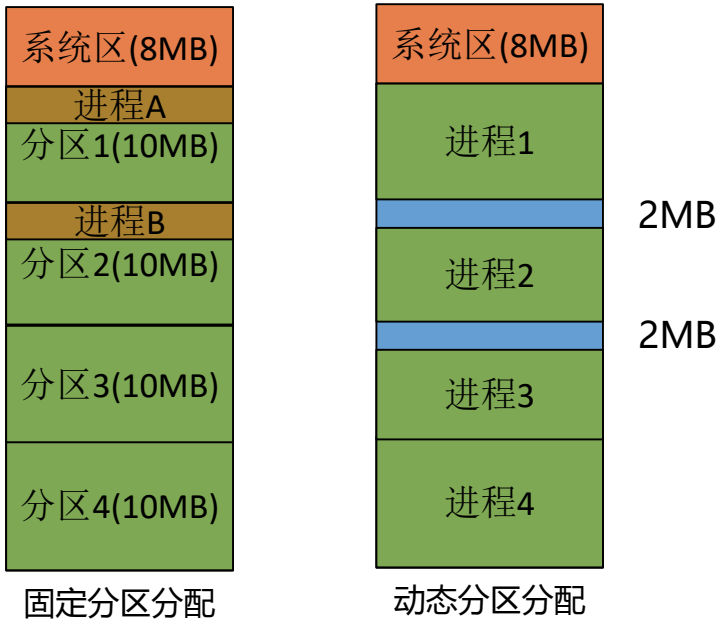
连续分配方式有“碎片”，通过“紧凑”消除碎片。

→ 离散分配方式

(1) 分页存储管理方式。

(2) 分段存储管理方式。

(3) 段页式存储管理方式。



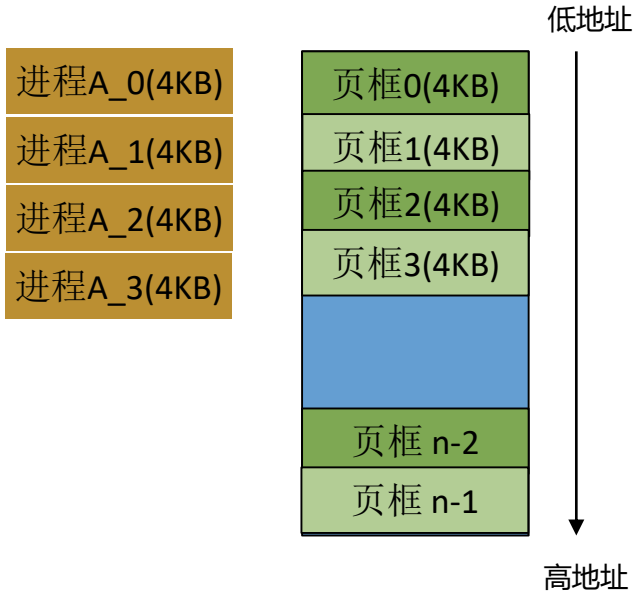
# 4.5 分页存储管理方式

连续分配方式有“碎片”，通过“紧凑”消除碎片。

→ 离散分配方式

(1) 分页存储管理方式。

| | | |
|---|---|---|
| | 页框 | 页框号 |
| 又称: | 页面 | 页面号 |
| 又称: | 页帧 | 页帧号 |
| 又称: | 内存块 | 内存块号 |
| 又称: | 物理块 | 物理块号 |



# 4.5 分页存储管理方式

## 4.5.1 分页存储管理的基本方法

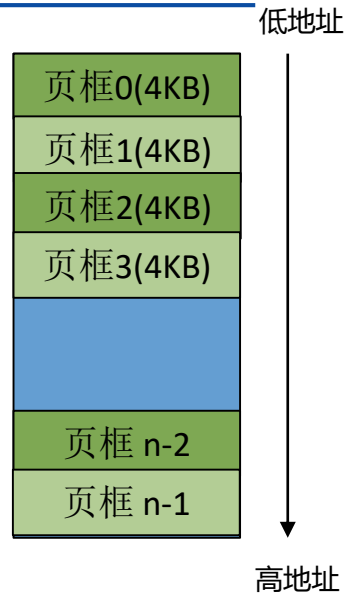
### 1. 页面和物理块

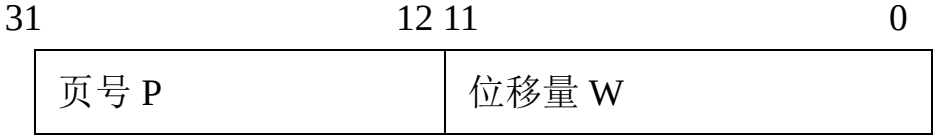
(1) 页面。

(2) 页面大小。

### 2. 地址结构

分页地址中的地址结构如下：




对某特定机器，其地址结构是一定的。若给定一个逻辑地址空间中的地址为A，页面的大小为L，则页号P和页内地址d可按下式求得：

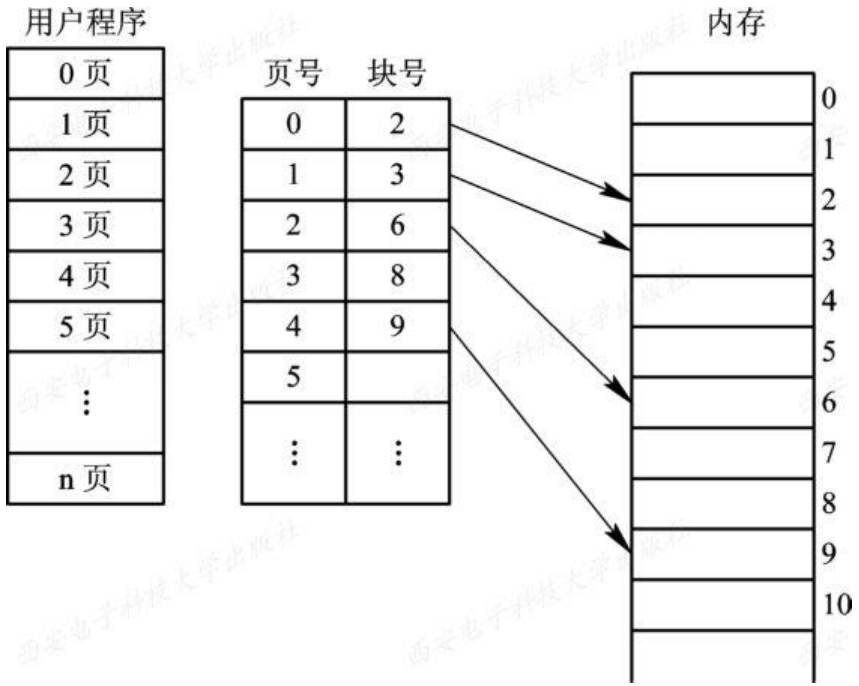
$$P = \mathrm{INT}\left[\frac{A}{L}\right], \qquad d = [A] \ \mathrm{MOD}\ L$$

# 4.5 分页存储管理方式

### 4.5.1 分页存储管理的基本方法

**3. 页表**

在分页系统中，允许将进程的各个页离散地存储在内存的任一物理块中，为保证进程仍然能够正确地运行，即能在内存中找到每个页面所对应的物理块，系统又为每个进程建立了一张页面映像表，简称页表。



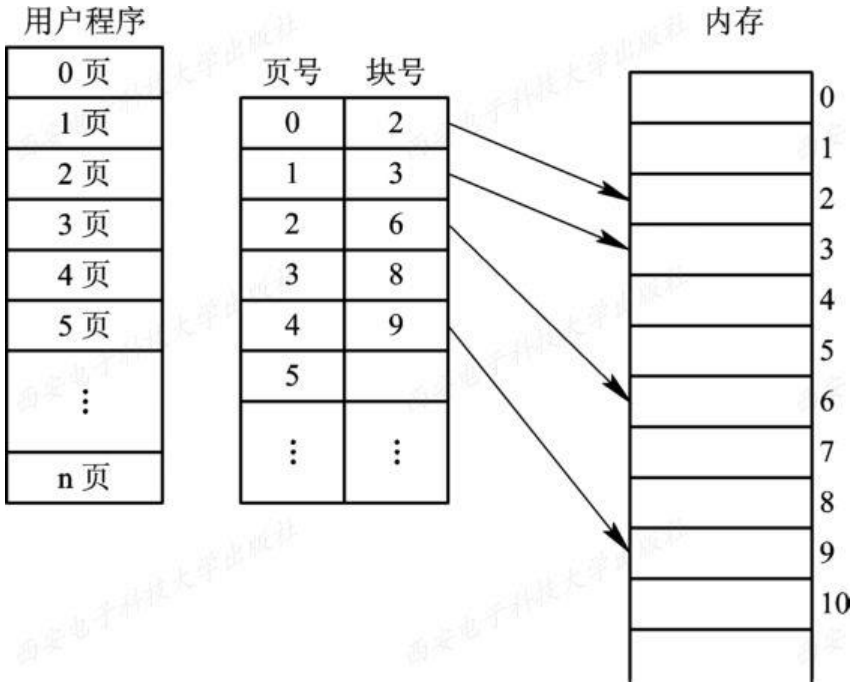

一个进程 ←→ 一个页表

# 4.5 分页存储管理方式

### 4.5.1 分页存储管理的基本方法

**3. 页表**

某计算机系统按字节寻址，支持32位的逻辑地址，采用分页存储管理，页面大小为4KB，页表项长度为4B。




页内地址需要：4KB=$2^{12}$B，需要12位

页　　号：32-12=20，20位表示页号

该系统中用户进程最多有：$2^{20}$ 页

一个进程的页表中最多有：$2^{20}$ 页

一个页表最大占用空间：$2^{20}$ 页 * 4B/页 = $2^{22}$ B

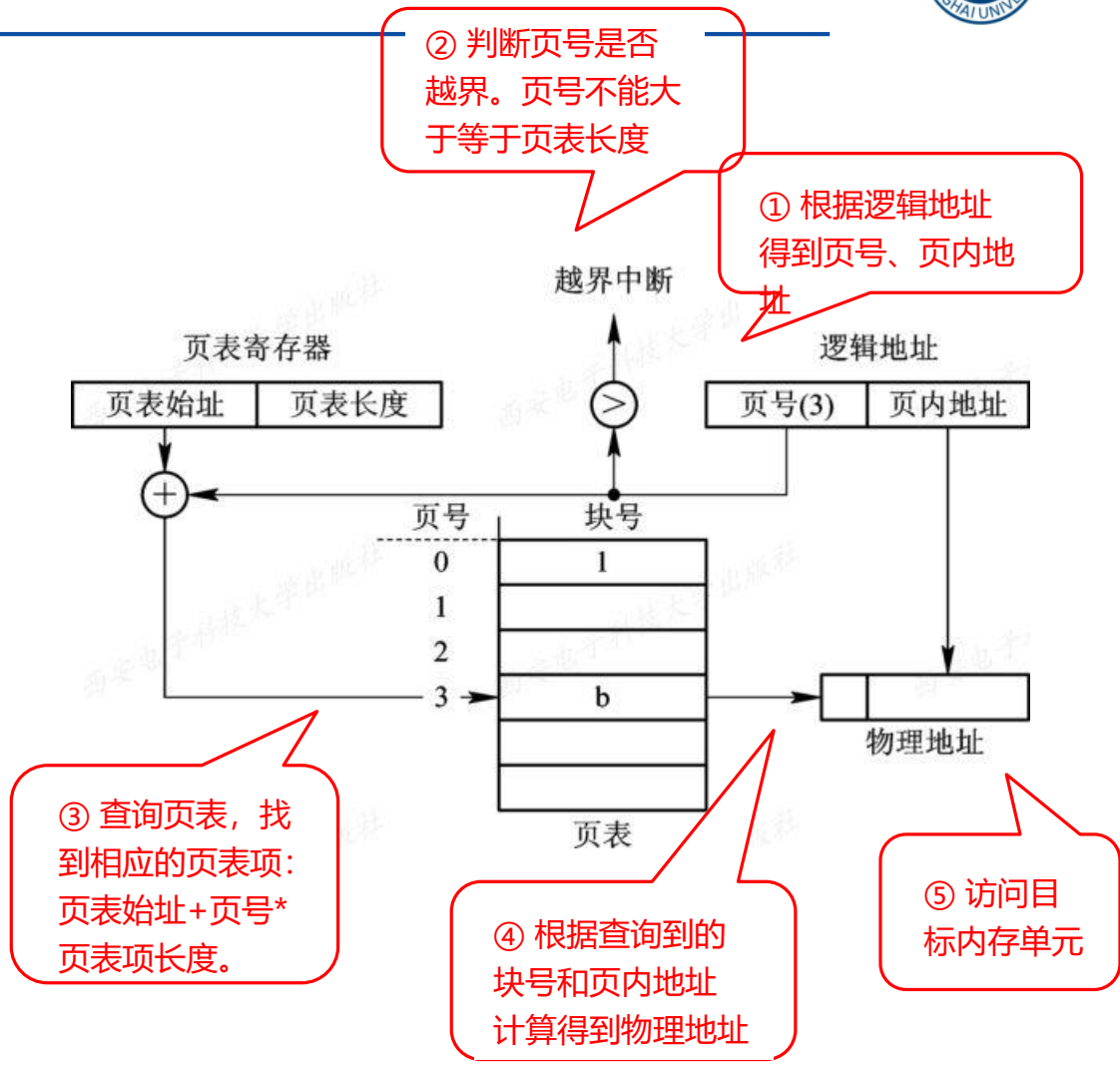
需要 $2^{22}/2^{12}=2^{10}$个页框存储该页表

# 4.5 分页存储管理方式

## 4.5.2 地址变换机构

### 1. 基本的地址变换机构

进程在运行期间，需要对程序和数据的地址进行变换，即将用户地址空间中的逻辑地址变换为内存空间中的物理地址，由于它执行的频率非常高，每条指令的地址都需要进行变换，因此需要采用硬件来实现。页表功能是由一组专门的寄存器来实现的。一个页表项用一个寄存器。



# 4.5 分页存储管理方式

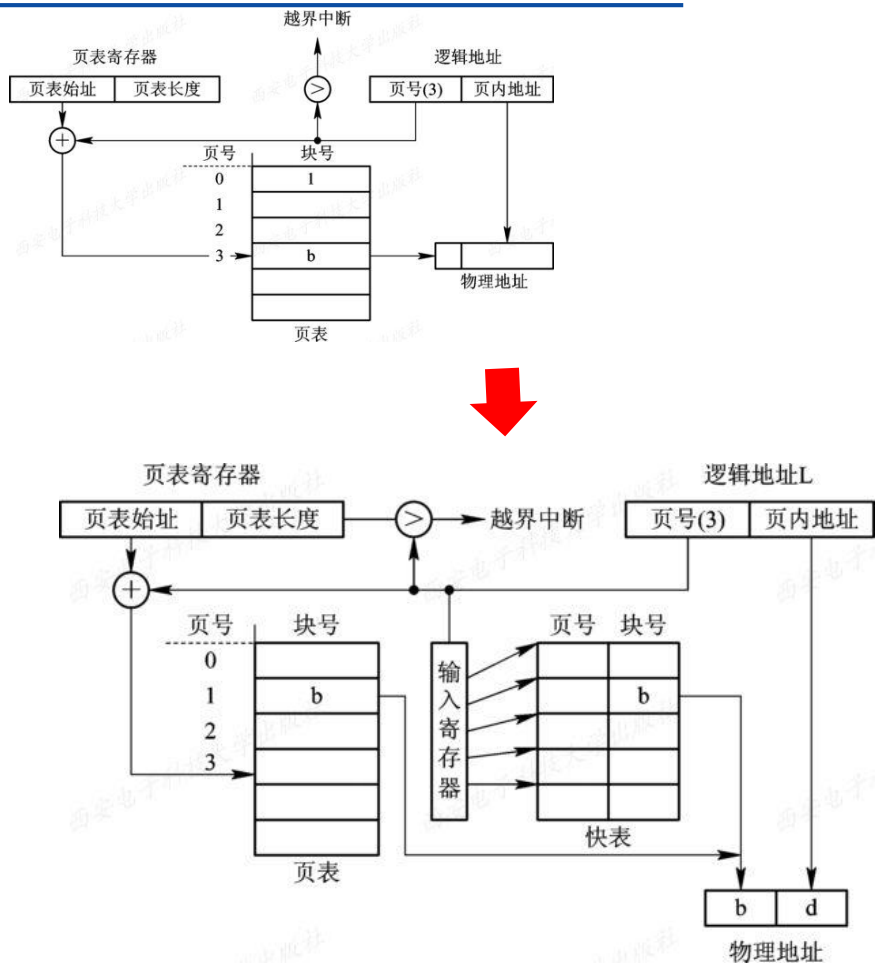
### 4.5.2 地址变换机构

**2. 具有快表的地址变换机构**

由于页表是存放在内存中的，这使CPU在每存取一个数据时，都要两次访问内存。

第一次是访问内存中的页表，从中找到指定页的物理块号，再将块号与页内偏移量W拼接，以形成物理地址。

第二次访问内存时，才是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)。因此，采用这种方式将使计算机的处理速度降低近1/2。可见，以此高昂代价来换取存储器空间利用率的提高，是得不偿失的。

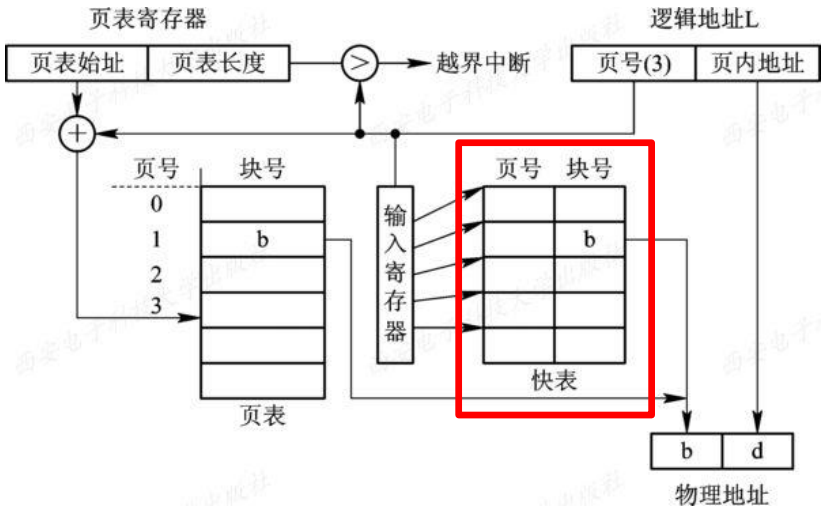


# 4.5 分页存储管理方式

## 4.5.2 地址变换机构

### 2. 具有快表的地址变换机构

快表：TLB（Translation Look Aside Buffer）



# 4.5 分页存储管理方式

### 4.5.3 访问内存的有效时间（无快表）

从进程发出指定逻辑地址的访问请求，经过地址变换，到在内存中找到对应的实际物理地址单元并取出数据，所需要花费的总时间，称为内存的有效访问时间(Effective Access Time，EAT)。

假设访问一次内存的时间为t，在基本分页存储管理方式中，有效访问时间分为第一次访问内存时间(即查找页表对应的页表项所耗费的时间t)与第二次访问内存时间(即将页表项中的物理块号与页内地址拼接成实际物理地址所耗费的时间t)之和：

$$EAT = t + t = 2t$$

# 4.5 分页存储管理方式

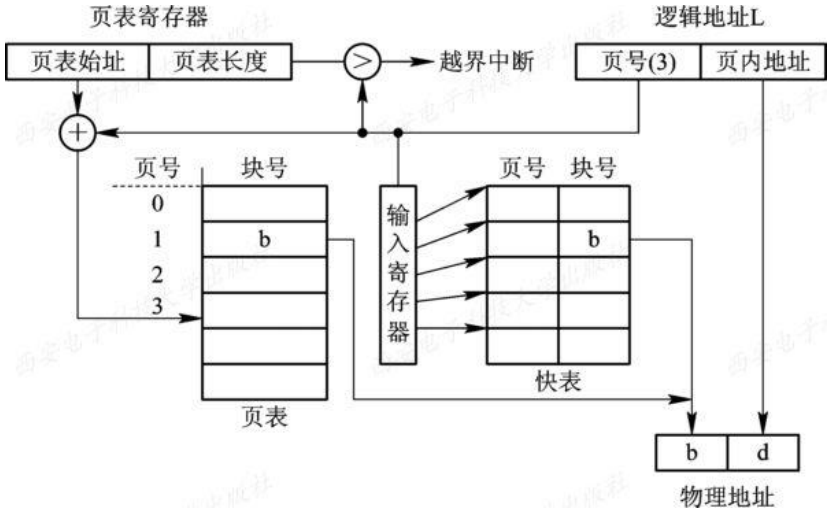
### 4.5.3 访问内存的有效时间（有快表）

在引入快表的分页存储管理方式中，有效访问时间的计算公式即为：

$$EAT=a\times\lambda+(t+\lambda)(1-a)+t=2t+\lambda-t\times a$$

或 $EAT=a(\lambda+t)+(1-a)(\lambda+t+t)=2t+\lambda-t\times a$

上式中，λ表示查找快表所需要的时间，a表示命中率，t表示访问一次内存所需要的时间。



# 4.5 分页存储管理方式

可见，引入快表后的内存有效访问时间分为查找到逻辑页对应的页表项的平均时间a × λ + (t + λ)(1 - a)，以及对应实际物理地址的内存访问时间t。假设对快表的访问时间λ为20ns(纳秒)，对内存的访问时间t为100 ns，则下表中列出了不同的命中率a与有效访问时间的关系：

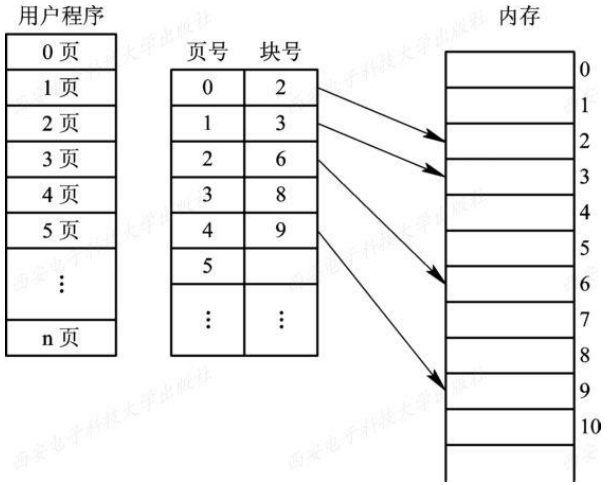
| 命中率 (%)<br>a | 有效访问时间<br>EAT |
|---|---|
| 0 | 220 |
| 50 | 170 |
| 80 | 140 |
| 90 | 130 |
| 98 | 122 |

# 4.5 分页存储管理方式

### 4.5.4 两级和多级页表

#### 1. 多级和多级页表

对于32位的机器，采用一级级页表结构是合适的，但对于64位的机器，采用一级页表是否仍然合适，须做以下简单分析。如果页面大小仍采用4 KB即$2^{12}$ B，一个页表项占用4B。




页内地址需要：4KB=$2^{12}$B，需要12位

页　　号：64-12=52，52位表示页号

该系统中用户进程最多有：　$2^{52}$ 页

一个进程的页表中最多有：　$2^{52}$ 页

一个页表最大占用空间：　$2^{52}$ 页 *　4B/页 = $2^{54}$ B

需要 $2^{54}/2^{12}=2^{42}$个页框存储该页表

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。

一级页表需要占用大量的连续空间，难以满足！

# 4.5 分页存储管理方式

### 4.5.4 两级和多级页表

#### 1. 两级页表(Two-Level Page Table)

针对难于找到大的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的方法，使每个页面的大小与内存物理块的大小相同，并为它们进行编号，即依次为0# 页、1# 页，…，n# 页，然后离散地将各个页面分别存放在不同的物理块中。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。

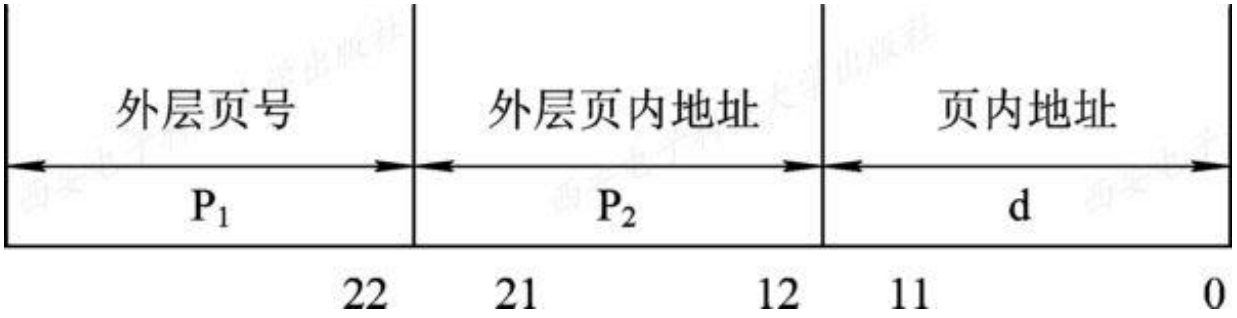
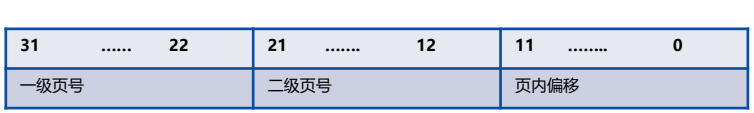



图4-17　两级页表结构

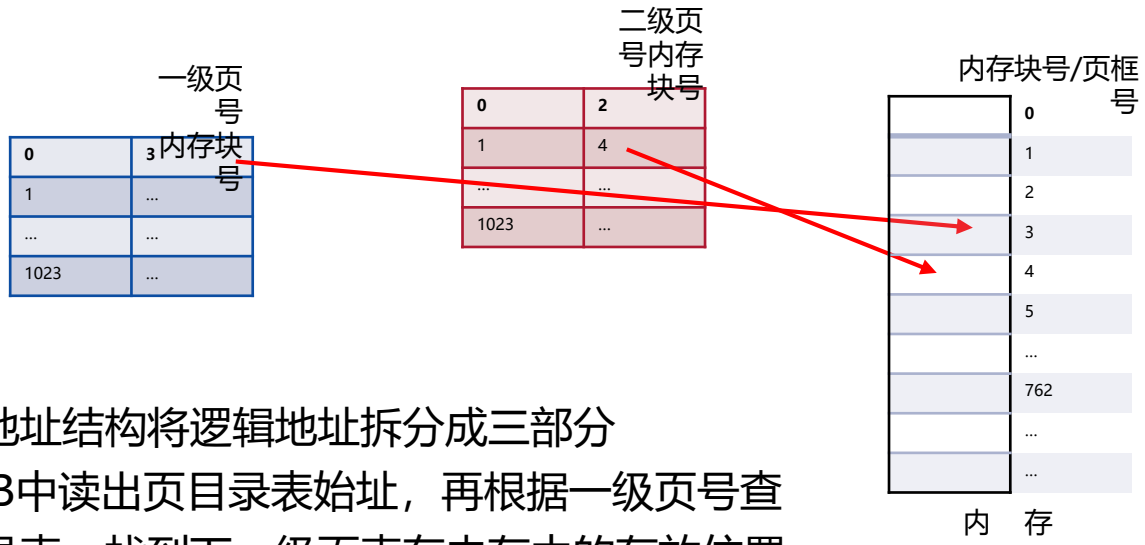
# 4.5 分页存储管理方式

## 4.5.4 两级和多级页表

### 2. 多级页表




两级页表结构的逻辑地址




① 按照地址结构将逻辑地址拆分成三部分

② 从PCB中读出页目录表始址，再根据一级页号查页目录表，找到下一级页表在内存中的存放位置

③ 根据二级页号查表，找到最终想访问的内存块号

④ 结合页内偏移量得到物理地址

页面大小为4KB

最终要访问的内存块号为4

该内存块的起始地址为

4*4096= 16384

假设页内偏移量为1023

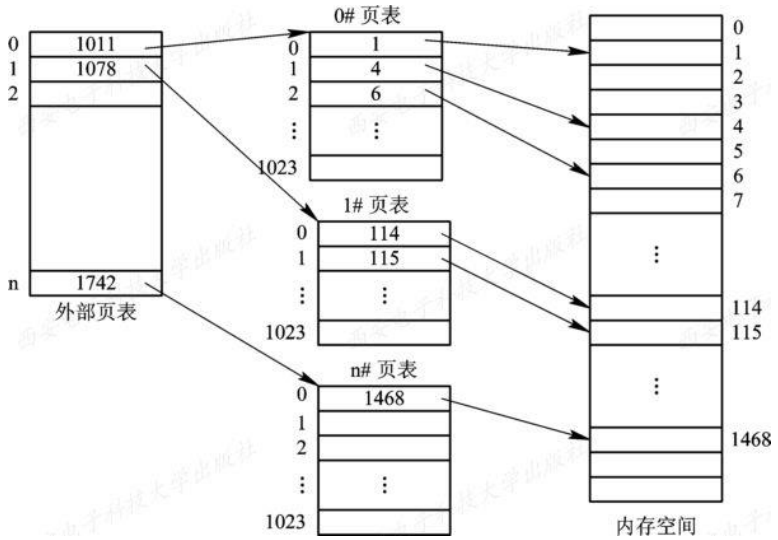
物理地址=起始地址+页内偏移

# 4.5 分页存储管理方式

### 4.5.4 两级和多级页表

#### 2. 多级页表

把多个页表项分成多组，每组空间连续。



# 4.5 分页存储管理方式

### 4.5.4 两级和多级页表

#### 2. 多级页表

若采用多级页表机制，则各级页表的大小不能超过一个页面

例:某系统按字节编址，采用40位逻辑地址，页面大小为4KB，页表项大小为4B，假设采用纯页式存储，则要采用（）级页表，页内偏移量为()位?

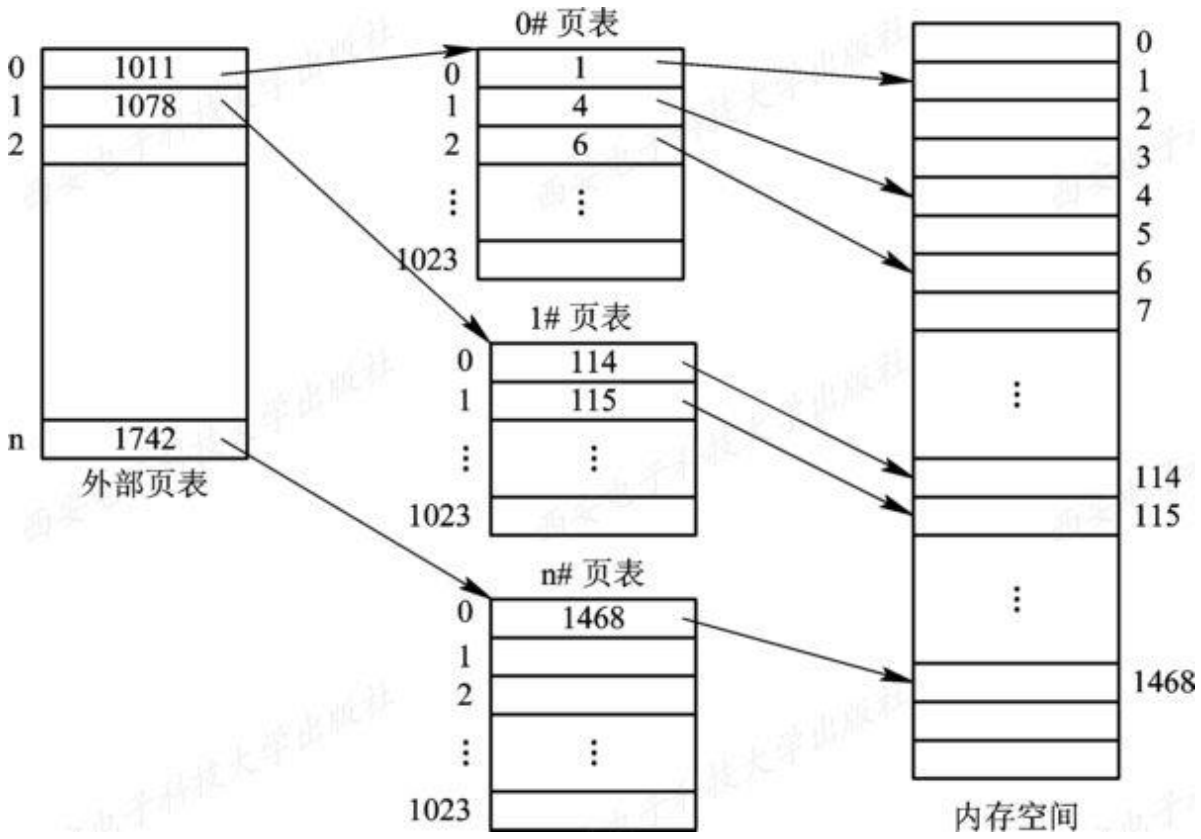
页面大小=4KB =$2^{12}$B，按字节编址，因此页内偏移量为12位

页号=40-12= 28位

页面大小= $2^{12}$B，页表项大小=4B，则每个页面可存放$2^{12}/4 = 2^{10}$个页表项。

因此各级页表最多包含$2^{10}$个页表项，需要10位二进制位才能映射到$2^{10}$个页表项，因此每一级的页表对应页号应为10位。总共28位的页号至少要分为三级。

# 4.5 分页存储管理方式



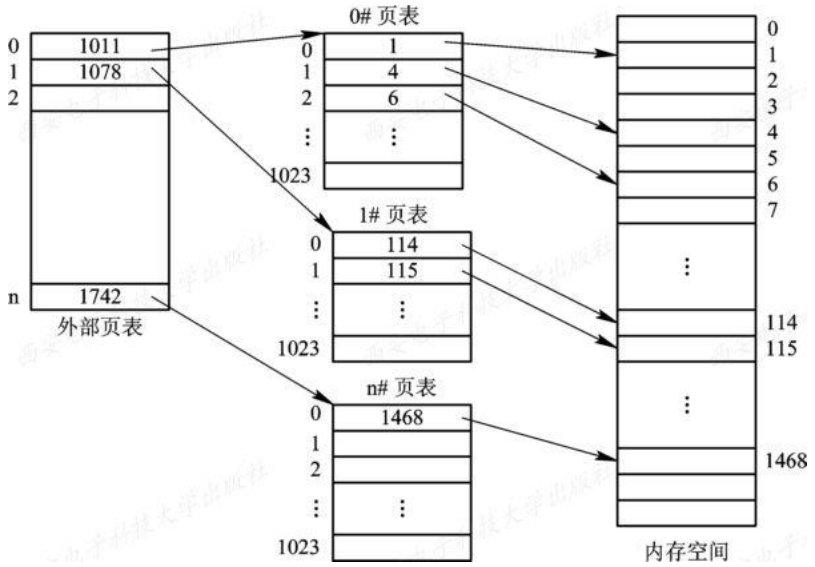

**为了提高内存空间利用率，页应该小，但页小了，页表就大了。**

# 4.5 分页存储管理方式

## 4.5.4 两级和多级页表

### 2. 多级页表

内存访问次数：页表级数+1（无快表的情况下）




**为了提高内存空间利用率，页应该小，但页小了，页表就大了，访问次数变多了。**

# 4.5 分页存储管理方式

### 4.5.5 反置页表(Inverted Page Table)

**1. 反置页表的引入**

在分页系统中，为每个进程配置了一张页表，进程逻辑地址空间中的每一页，在页表中都对应有一个页表项。在现代计算机系统中，通常允许一个进程的逻辑地址空间非常大，因此就需要有许多的页表项，而因此也会占用大量的内存空间。

**2. 地址变换**

在利用反置页表进行地址变换时，是根据进程标识符和页号，去检索反置页表。如果检索到与之匹配的页表项，则该页表项(中)的序号i便是该页所在的物理块号，可用该块号与页内地址一起构成物理地址送内存地址寄存器。若检索了整个反置页表仍未找到匹配的页表项，则表明此页尚未装入内存。对于不具有请求调页功能的存储器管理系统，此时则表示地址出错。对于具有请求调页功能的存储器管理系统，此时应产生请求调页中断，系统将把此页调入内存。

# 4.5 分页存储管理方式

**反置页表自始至终只存在一个页表，无论多少个进程，都只有一个页表！**

## 4.5.5 反置页表(Inverted Page Table)

某计算机系统有64MB（$2^{26}$），采用分页存储管理，页面大小为4KB，页表项长度为4B。

页内地址需要：4KB=$2^{12}$B，需要12位

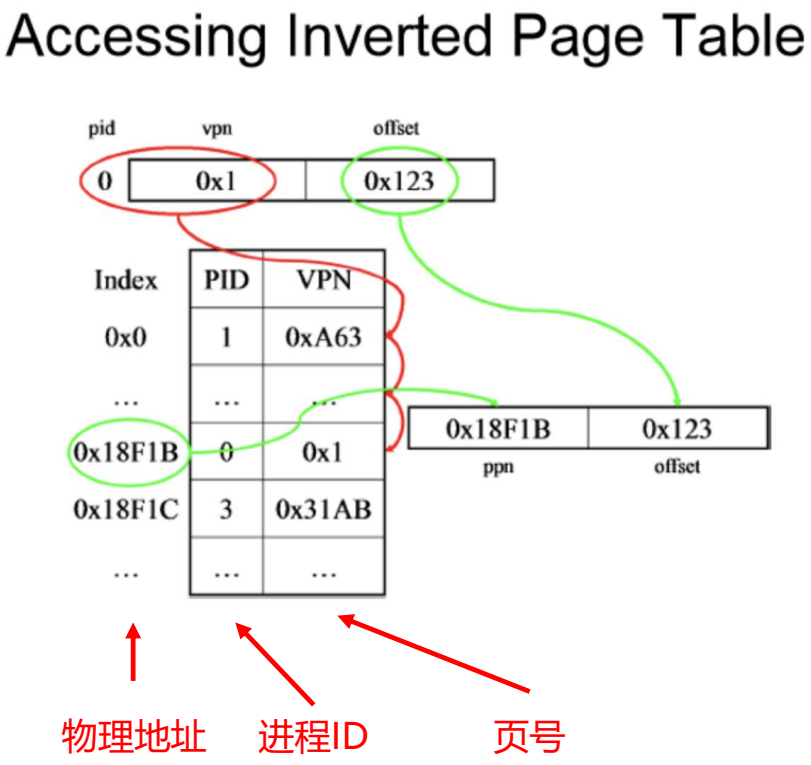
页　　号：26-12=14，14位表示页号

该系统中用户进程最多有：　$2^{14}$ 页

一个进程的页表中最多有：　$2^{14}$ 页

一个页表最大占用空间：

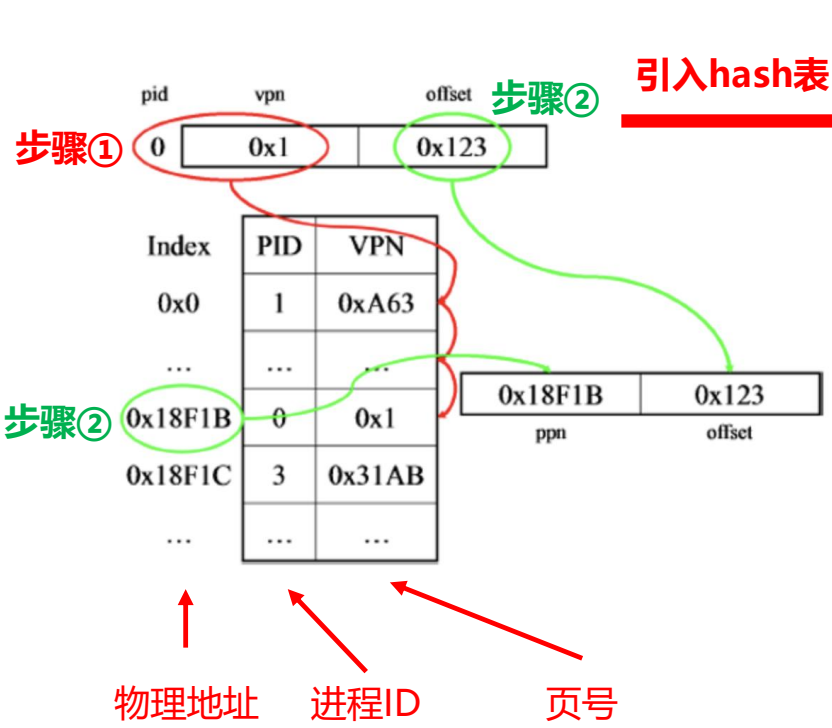
$2^{14}$ 页 * 4B/页 = $2^{16}$ B=64KB



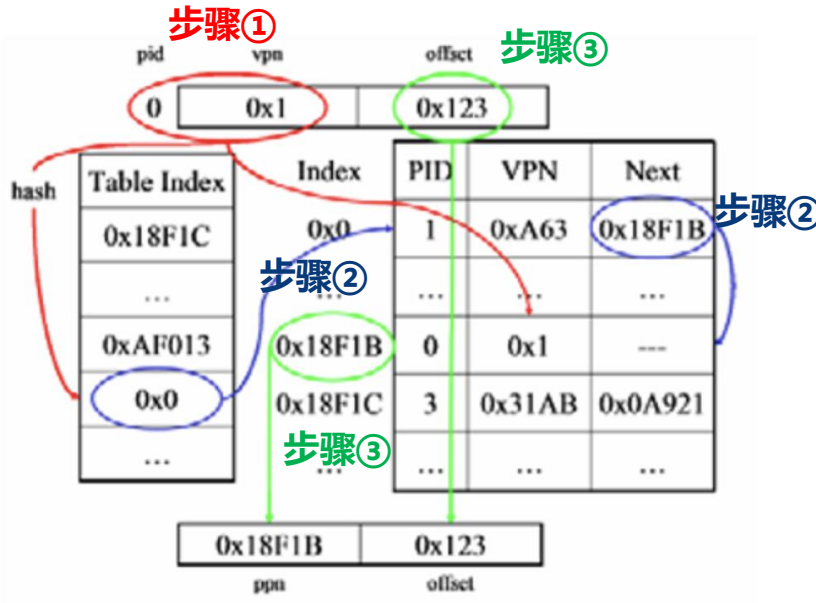
# 4.5 分页存储管理方式

反置页表自始至终只存在一个页表，无论多少个进程，都只有一个页表！

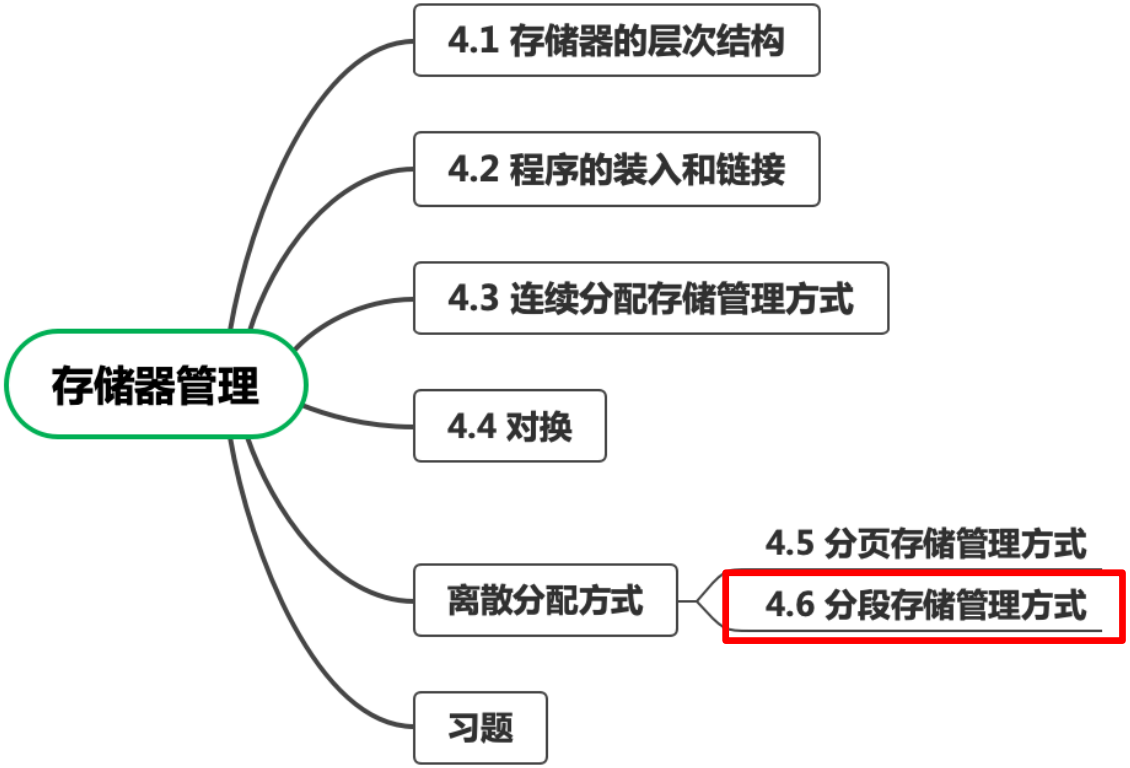
## 4.5.5 反置页表(Inverted Page Table)




引入hash表，加快速度！

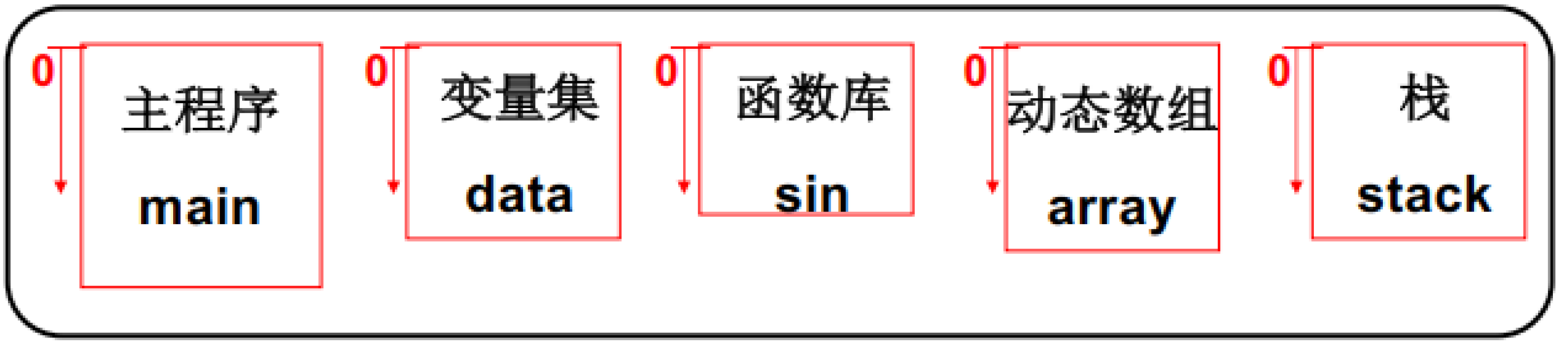


# 第四章　存储器管理


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 连续分配存储管理方式
4.4 对换
离散分配方式
4.5 分页存储管理方式
4.6 分段存储管理方式
习题

# 4.6 分段存储管理方式

存储管理方式随着OS的发展也在不断地发展。当OS由单道向多道发展时，存储管理方式便由单一连续分配发展为固定分区分配。




程序员眼中的一个程序

- 一个程序分为好几个段
- 每个段的长度不同
- 分治
- 段号，段内偏移

# 4.6 分段存储管理方式

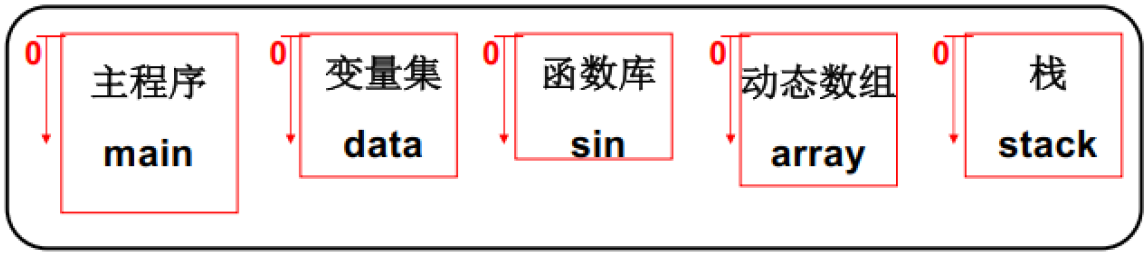
### 4.6.1 分段存储管理方式的引入

**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，每个段都从0开始编址，并有自己的名字和长度。因此，程序员们都迫切地需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，这不仅可以方便程序员编程，也可使程序非常直观，更具可读性。例如，下述的两条指令便使用段名和段内地址：

LOAD 1，[A]｜〈D〉；

STORE 1，[B]｜〈C〉；




程序员眼中的一个程序

# 4.6 分段存储管理方式

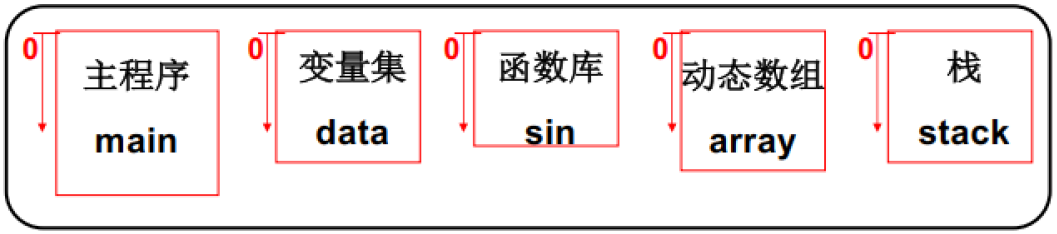
## 4.6.1 分段存储管理方式的引入

### 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，为了共享某个过程、函数或文件。分页系统中的“页”只是存放信息的物理单位(块)，并无完整的逻辑意义，这样，一个可被共享的过程往往可能需要占用数十个页面，这为实现共享增加了困难。

### 3. 信息保护

信息保护同样是以信息的逻辑单位为基础的，而且经常是以一个过程、函数或文件为基本单位进行保护的。



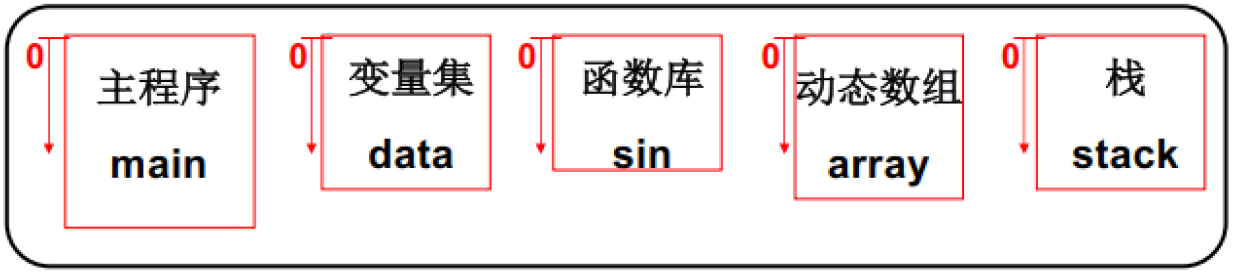

程序员眼中的一个程序

# 4.6 分段存储管理方式

### 4.6.1 分段存储管理方式的引入

**4. 动态增长**

在实际应用中，往往存在着一些段，尤其是数据段，在它们的使用过程中，由于数据量的不断增加，而使数据段动态增长，相应地它所需要的存储空间也会动态增加。然而，对于数据段究竟会增长到多大，事先又很难确切地知道。对此，很难采取预先多分配的方法进行解决。



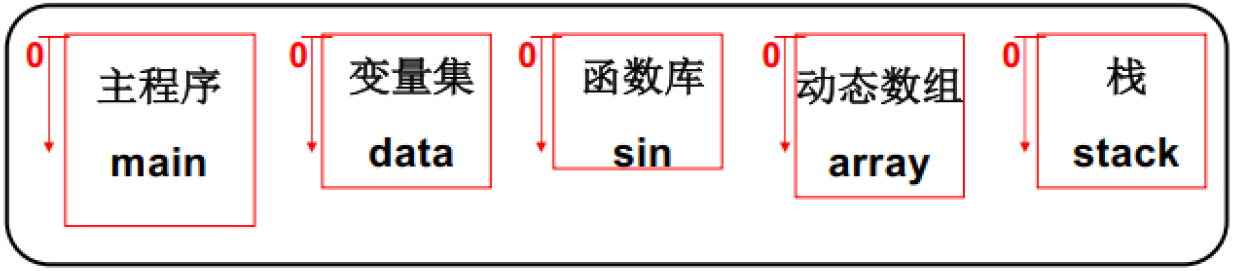

程序员眼中的一个程序

# 4.6 分段存储管理方式

## 4.6.1 分段存储管理方式的引入

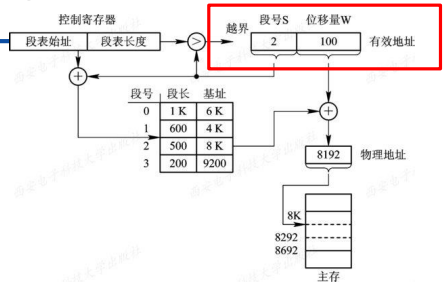
### 5. 动态链接

动态链接要求以目标程序（即段）作为链接的基本单位，因此，分段存储管理方式非常适合于动态链接。




程序员眼中的一个程序

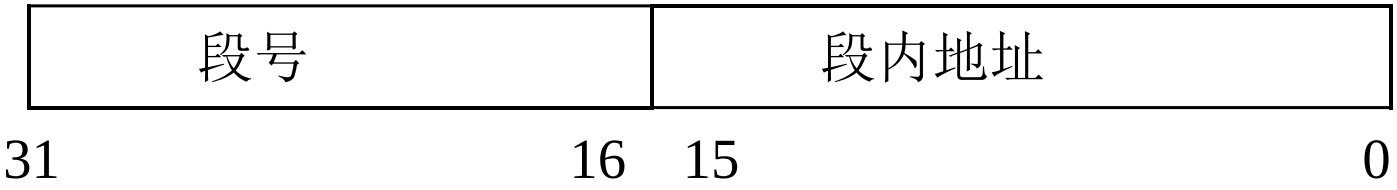
# 4.6 分段存储管理方式



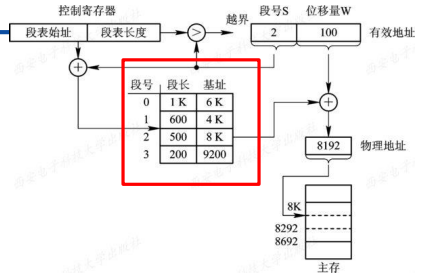

## 4.6.2 分段系统的基本原理

### 1. 分段

分段地址中的地址具有如下结构：



# 4.6 分段存储管理方式

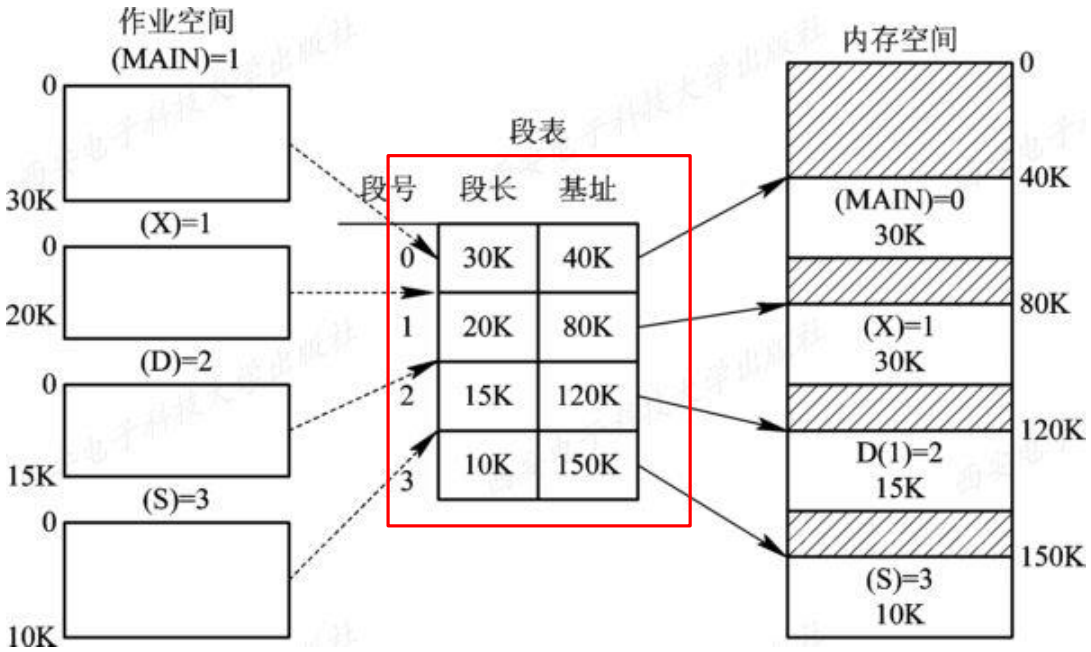



## 4.6.2 分段系统的基本原理

### 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。

段表（段号是隐藏项）：

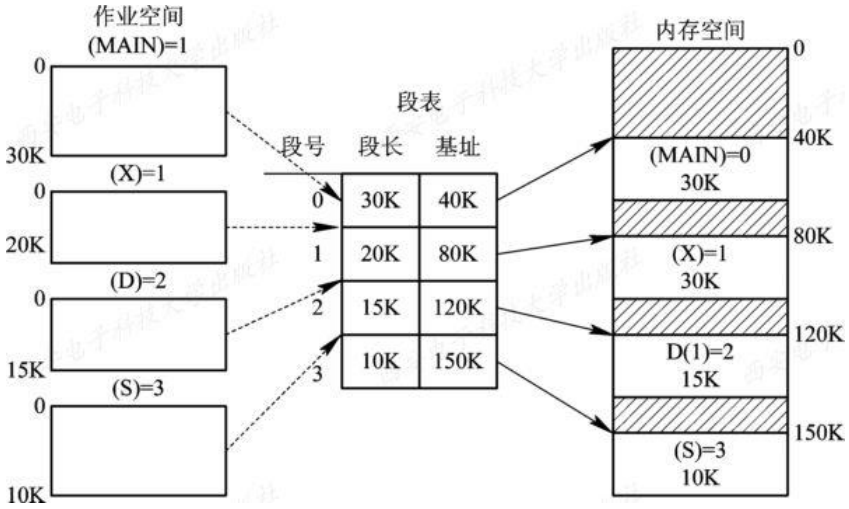


# 4.6 分段存储管理方式

## 4.6.2 分段系统的基本原理

### 2. 段表

在前面所介绍的动态分区分配方式中，系统为整个进程分配一个连续的内存空间。而在分段式存储管理系统中，则是为每个分段分配一个连续的分区。进程中的各个段，可以离散地装入内存中不同的分区中。为保证程序能正常运行，就必须能从物理内存中找出每个逻辑段所对应的位置。



# 4.6 分段存储管理方式

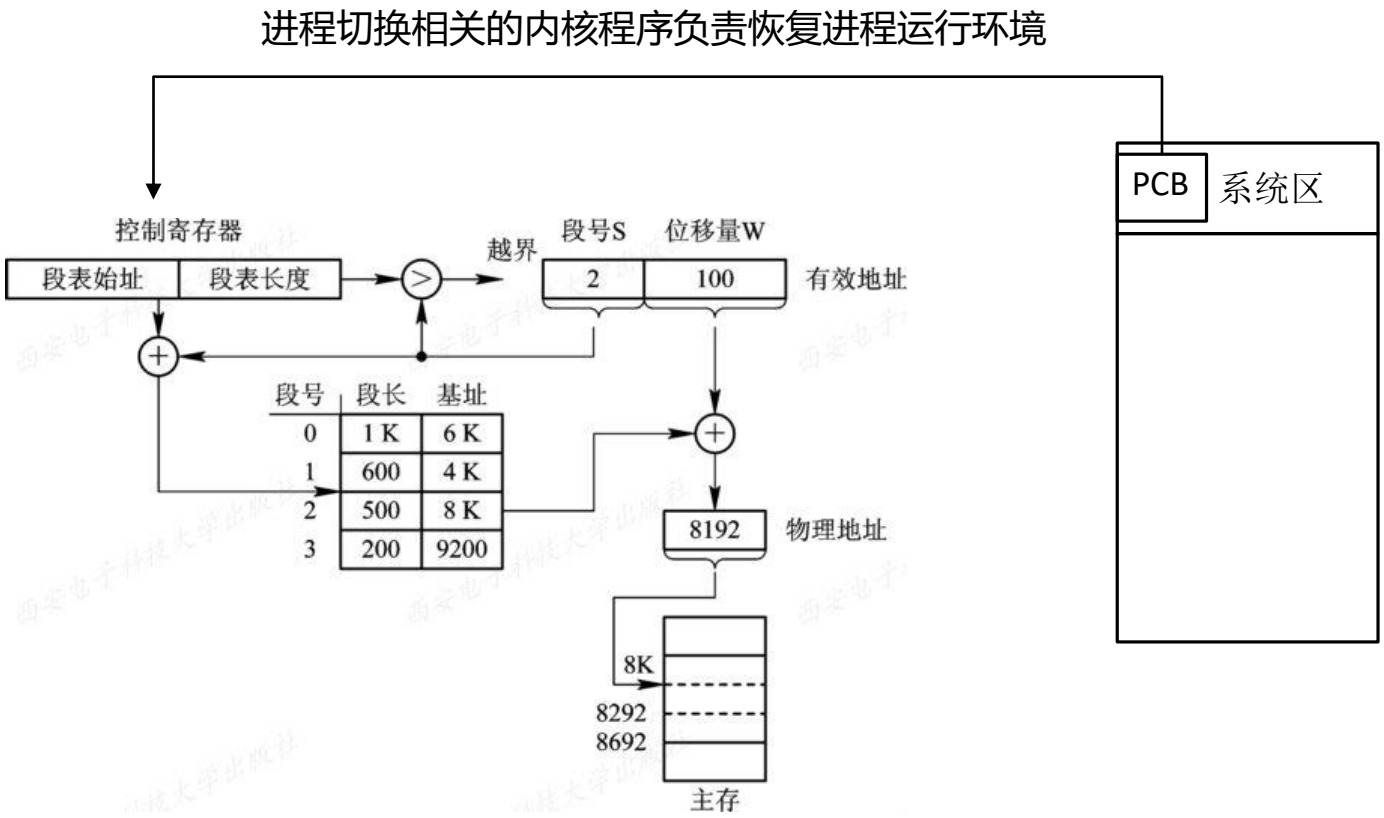
### 4.6.2 分段系统的基本原理

#### 3. 地址变换机构

为了实现进程从逻辑地址到物理地址的变换功能，在系统中设置了段表寄存器，用于存放段表始址和段表长度TL。

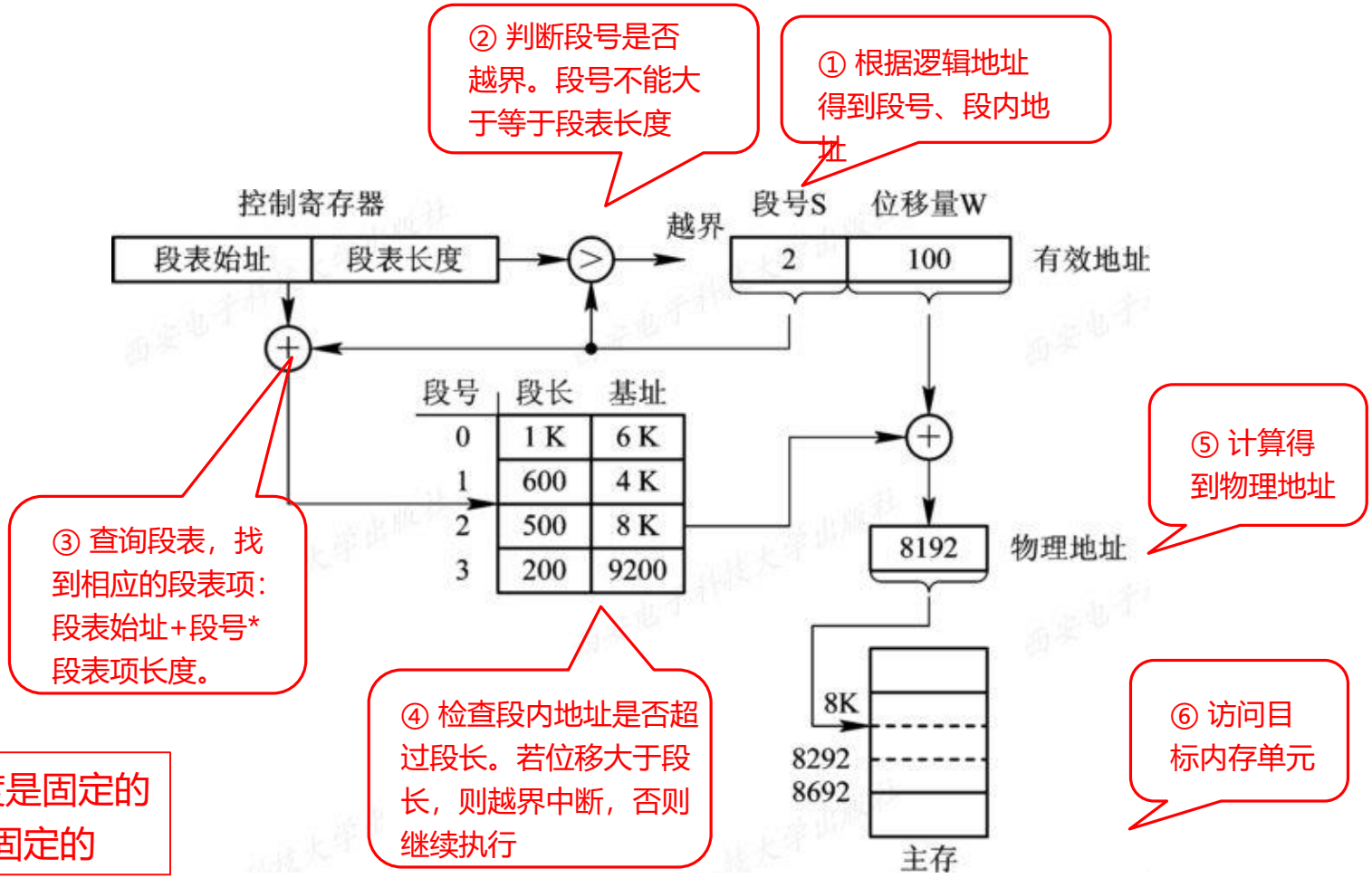
在进行地址变换时，系统将逻辑地址中的段号与段表长度TL进行比较。若S>TL，表示段号太大，是访问越界，于是产生越界中断信号。若未越界，则根据段表的始址和该段的段号，计算出该段对应段表项的位置，从中读出该段在内存的起始地址。然后，再检查段内地址d是否超过该段的段长SL。若超过，即d>SL，同样发出越界中断信号。若未越界，则将该段的基址d与段内地址相加，即可得到要访问的内存物理地址。图4-20示出了分段系统的地址变换过程。

# 4.6 分段存储管理方式




段表始址与段表长度从PCB转移到控制寄存器中。 段表始址：段表存放的初始地址；段表长度：段表项的数量。

# 4.6 分段存储管理方式



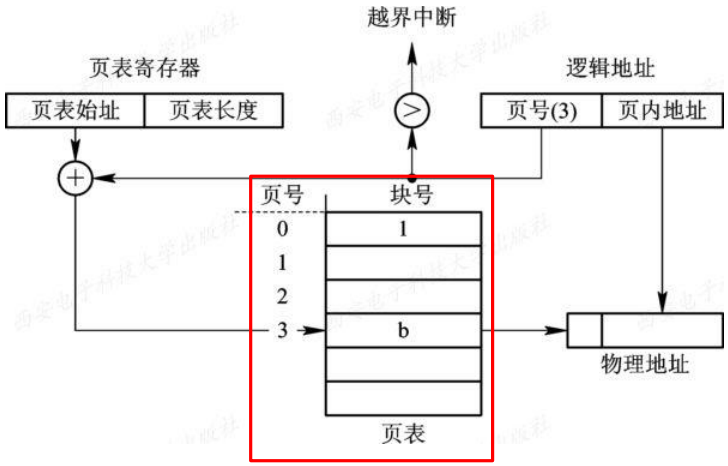

段表项的长度是固定的
段长是不固定的

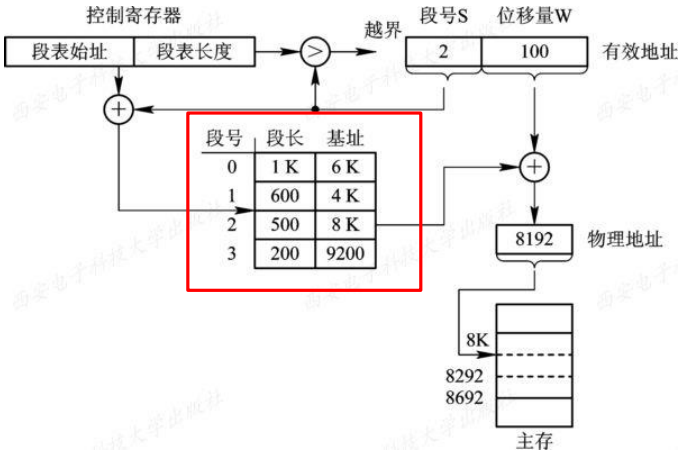
# 4.6 分段存储管理方式

## 4.6.2 分段系统的基本原理

### 4. 分页和分段的主要区别

| | |
|---|---|
| (1) 页是信息的物理单位。 | 段是信息的逻辑单位。 |
| (2) 页的大小固定且由系统决定。 | 段的长度不固定。 |
| (3) 分页的用户程序地址空间是一维的。 | 需要同时给出段名与段内地址。 |



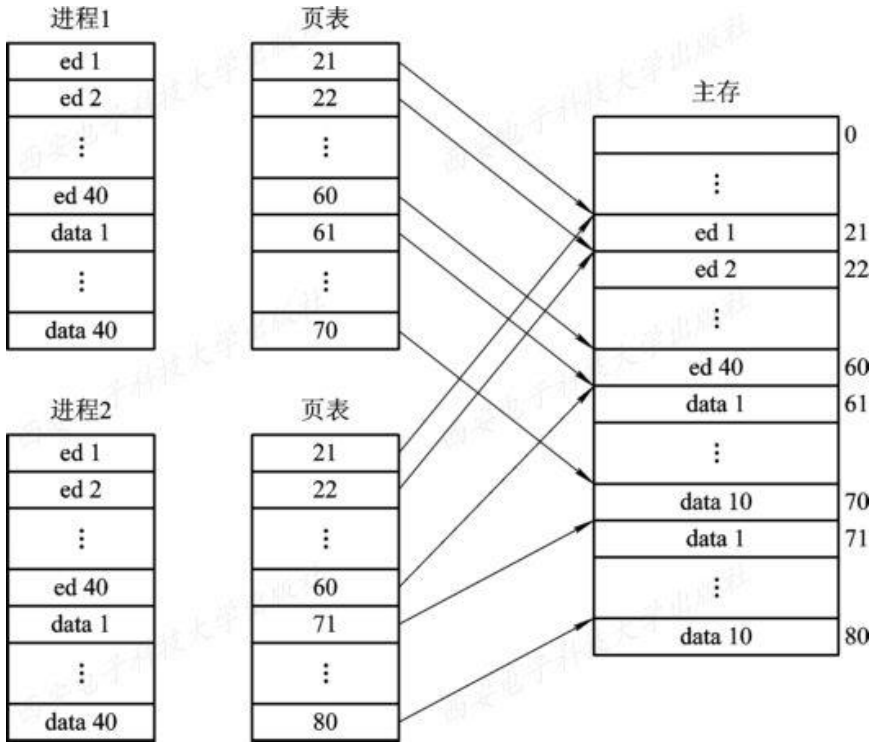



# 4.6 分段存储管理方式

### 4.6.3 信息共享

**1. 分页系统中对程序和数据的共享**

在分页系统中，虽然也能实现对程序和数据的共享，但远不如分段系统来得方便。我们通过一个例子来说明这个问题。

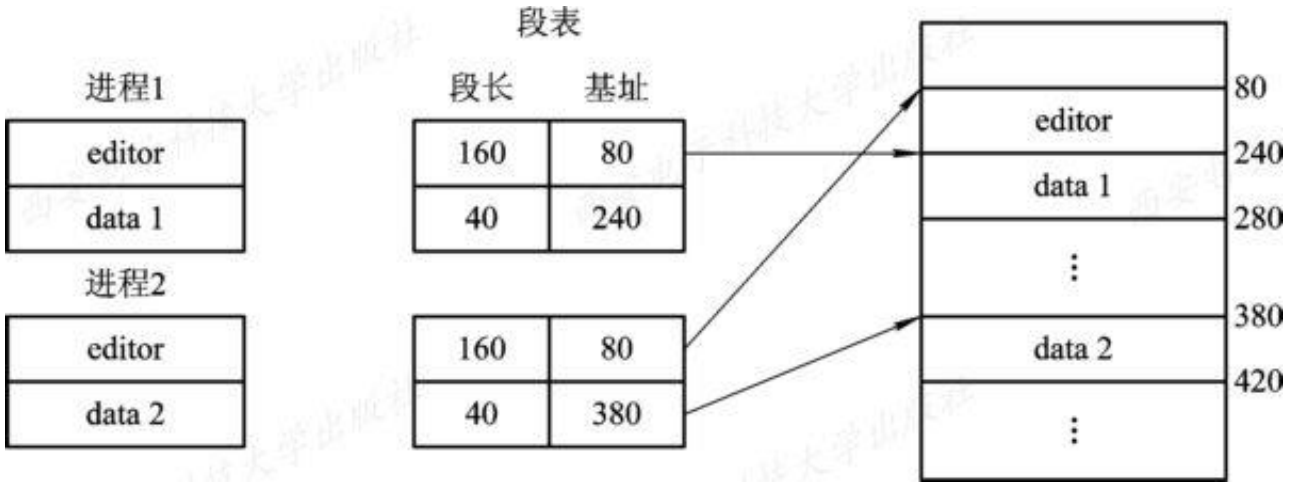


# 4.6 分段存储管理方式

### 4.6.3 信息共享

**2. 分段系统中程序和数据的共享**

在分段系统中，由于是以段为基本单位的，不管该段有多大，我们都只需为该段设置一个段表项，因此使实现共享变得非常容易。我们仍以共享editor为例，此时只需在(每个)进程1和进程2的段表中，为文本编辑程序设置一个段表项，让段表项中的基址(80)指向editor程序在内存的起始地址。图4-22是分段系统中共享editor的示意图。

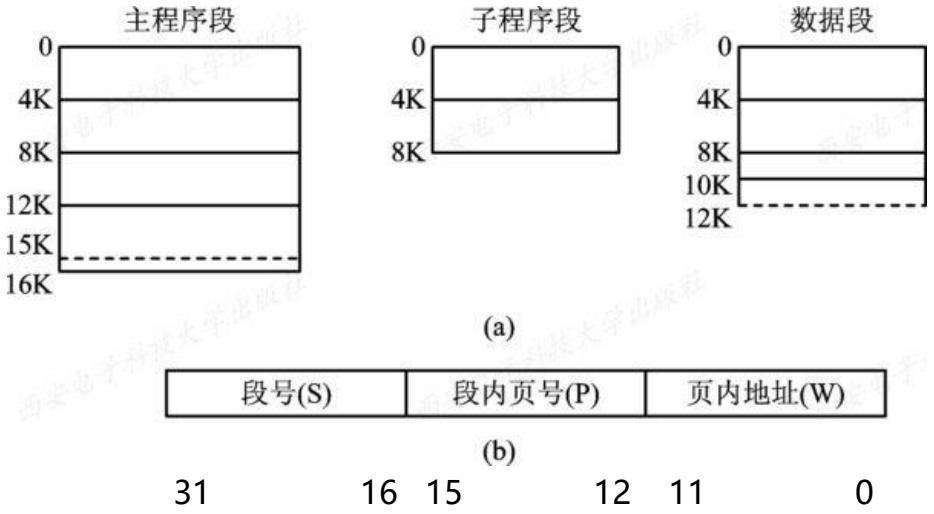


# 4.6 分段存储管理方式

## 4.6.4 段页式存储管理方式

### 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4-23(a)示出了一个作业地址空间的结构。该作业有三个段：主程序段、子程序段和数据段；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，如图4-23(b)所示。



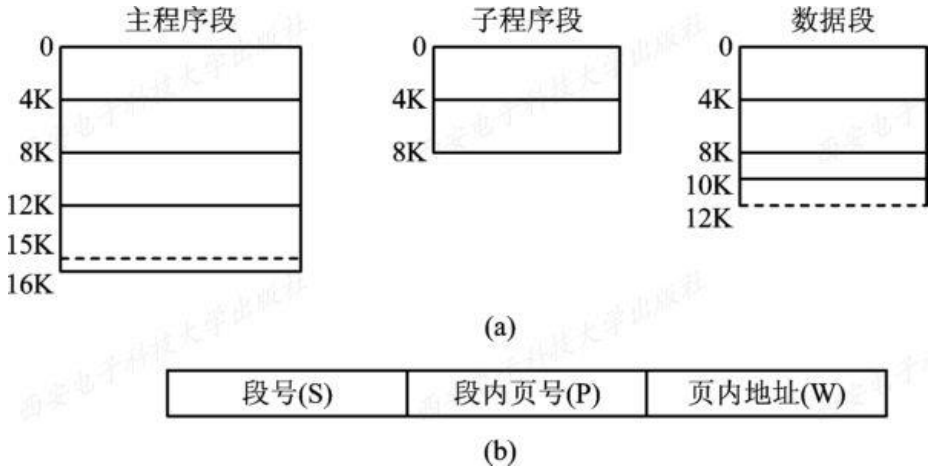
# 4.6 分段存储管理方式

### 4.6.4 段页式存储管理方式

#### 1. 基本原理

分段：用户给出，对用户可见

分页：系统给出，对用户不可见




段号的位数决定了每个进程最多可以分几个段

页号位数决定了每个段最大有多少页

页内偏移量决定了页面大小、内存块大小是多少

在上述例子中，若系统是按字节寻址的，则段号占16位，因此在该系统中，每个进程最多有$2^{16}$= 64K个段

页号占4位，因此每个段最多有$2^4$= 16页

页内偏移量占12位，因此每个页面\每个内存块大小为$2^{12}$= 4096= 4KB

# 4.6 分段存储管理方式

| 段号 | 段长 | 基址 |
|---|---|---|
| 0 | 1 K | 6 K |
| 1 | 600 | 4 K |
| 2 | 500 | 8 K |
| 3 | 200 | 9200 |

分段管理中的段表

段内有几个页表

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

段页管理中的段表

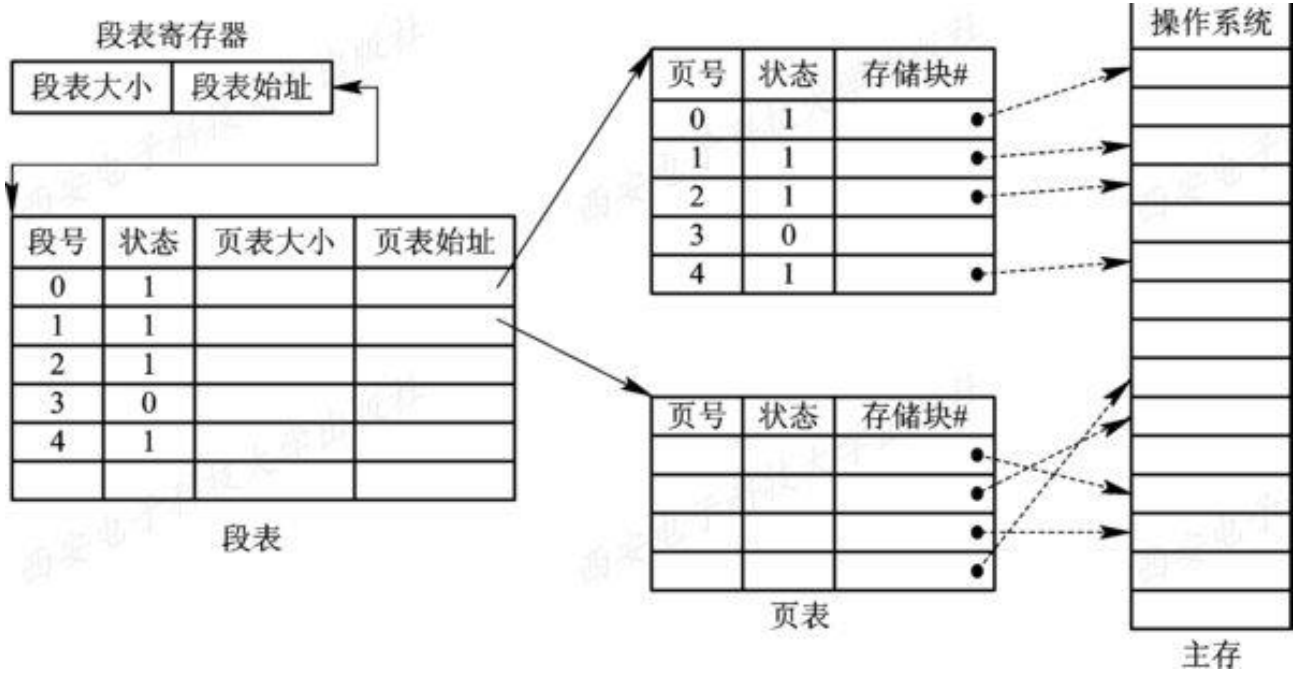
状态位表示当前页表是否在内存中

若不在内存中则引发缺页中断

教材上没有仔细介绍

# 4.6 分段存储管理方式

在段页式系统中，为了实现从逻辑地址到物理地址的变换，系统中需要同时配置段表和页表。段表的内容与分段系统略有不同，它不再是内存始址和段长，而是页表始址和页表长度。图展示出了利用段表和页表进行从用户地址空间到物理(内存)空间的映射。




一个进程可具有一个段表+多个页表

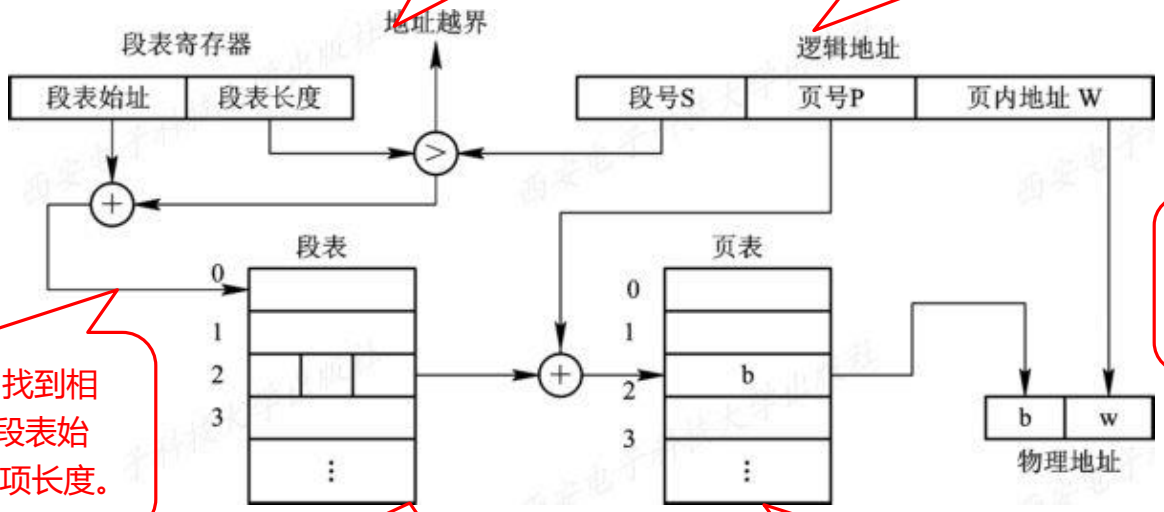
# 4.6 分段存储管理方式

### 4.6.4 段页式存储管理方式

**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长TL。进行地址变换时，首先利用段号S，将它与段长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。图4-25示出了段页式系统中的地址变换机构。

# 4.6 分段存储管理方式




② 判断段号是否越界。段号不能大于等于段表长度

① 根据逻辑地址得到段号、页号、页内偏移

⑥ 根据页内块号、页内偏移量找到最终的物理地址

③ 查询段表，找到相应的段表项：段表始址+段号*段表项长度。

④ 检查页号是否越界，若页号大于页表长度，则发生越界中断，否则继续执行

⑤ 根据页表存放块号、页号查询页表，找到对应页表项

⑦ 访问物理地址

# 习题

1. 为什么要配置层次式存储器?
2. 可采用哪几种方式将程序装入内存? 它们分别适用于何种场合?
3. 何谓静态链接? 静态链接时需要解决两个什么问题?
4. 何谓装入时动态链接? 装入时动态链接方式有何优点?
5. 何谓运行时动态链接? 运行时动态链接方式有何优点?
6. 在动态分区分配方式中，应如何将各空闲分区链接成空闲分区链?
7. 为什么要引入动态重定位? 如何实现?
8. 什么是基于顺序搜索的动态分区分配算法? 它可分为哪几种?
9. 在采用首次适应算法回收内存时，可能出现哪几种情况? 应怎样处理这些情况?
10. 什么是基于索引搜索的动态分区分配算法? 它可分为哪几种?

# 习题

11. 令buddyk(x)为大小为2k、地址为x的块的伙伴系统地址，试写出buddyk(x)的通用表达式。

12. 分区存储管理中常用哪些分配策略? 比较它们的优缺点。

13. 为什么要引入对换? 对换可分为哪几种类型?

14. 对文件区管理的目标和对对换空间管理的目标有何不同?

15. 为实现对换，系统应具备哪几方面的功能?

16. 在以进程为单位进行对换时，每次是否都将整个进程换出? 为什么?

17. 基于离散分配时所用的基本单位不同，可将离散分配分为哪几种?

18. 什么是页面? 什么是物理块? 页面的大小应如何确定?

19. 什么是页表? 页表的作用是什么?

20. 为实现分页存储管理，需要哪些硬件支持?

# 习题

21. 在分页系统中是如何实现地址变换的?
22. 具有快表时是如何实现地址变换的?
23. 较详细地说明引入分段存储管理是为了满足用户哪几方面的需要。
24. 在具有快表的段页式存储管理方式中，如何实现地址变换?
25. 为什么说分段系统比分页系统更易于实现信息的共享和保护?
26. 分页和分段存储管理有何区别?
27. 试全面比较连续分配和离散分配方式。

**3、静态重定位是在作业的（A）中进行的，动态重定位是在作业（B）中进行的。**

A：（1）编译过程　（2）装入过程　（3）修改过程　（4）执行过程

B：（1）编译过程　（2）装入过程　（3）修改过程　（4）执行过程

**5、静态链接是在（A）进行的；而动态链接是在（B）或（C）进行的，其中在（C）进行链接，可提高内存利用率。**

A：（1）编译某段程序时　（2）装入某段程序时　（3）调用某段程序时
（4）紧凑时　（5）装入程序之前

B：（1）编译某段程序时　（2）装入某段程序时　（3）调用某段程序时
（4）紧凑时　（5）装入程序之前

C：（1）编译某段程序时　（2）装入某段程序时　（3）调用某段程序时
（4）紧凑时　（5）装入程序之前

**6、要保证进程在主存中被改变了位置后仍能正确执行，则对主存空间应采用（A）。**

A：（1）静态重定位 （2）动态重定位 （3）动态链接 （4）静态链接

**8、在动态分区式内存管理中，倾向于优先使用低址部分空闲区的算法是（A）；能使内存空间中空闲区分布得较均匀的算法是（B）；每次分配时，把既能满足要求，又是最小的空闲区分配给进程的算法是（C）。**

A：（1）最佳适应算法　（2）最坏适应算法　（3）首次适应算法　（4）循环首次适应算法

B：（1）最佳适应算法　（2）最坏适应算法　（3）首次适应算法　（4）循环首次适应算法

C：（1）最佳适应算法　（2）最坏适应算法　（3）首次适应算法　（4）循环首次适应算法

**9、在首次适应算法中，要求空闲分区按（A）的顺序形成空闲分区链；在最佳适应算法中是按（B）的顺序形成空闲分区链；最坏适应算法是按（C）的顺序形成空闲链。**

A：（1）空闲区起始地址递增　　（2）空闲区起始地址递减

（3）空闲区大小递增　　（4）空闲区大小递减

B：（1）空闲区起始地址递增　　（2）空闲区起始地址递减

（3）空闲区大小递增　　（4）空闲区大小递减

C：（1）空闲区起始地址递增　　（2）空闲区起始地址递减

（3）空闲区大小递增　　（4）空闲区大小递减

**10、在动态分区式内存管理中，若某一时刻，系统内存的分配情况如图所示。当进程要申请一块20K的内存空间时，首次适应算法选中的是始址为（A）的空闲分区，最佳适应算法选中的是始址为（B）的空闲分区，最坏适应算法选中的是始址为（C）的空闲分区。**


A：（1）60K　（2）200K　（3）270K　（4）390K

B：（1）60K　（2）200K　（3）270K　（4）390K

C：（1）60K　（2）200K　（3）270K　（4）390K

**11、用动态分区存储管理系统中，主存总容量为55MB，初始状态全空，采用最佳适应算法，内存的分配和回收顺序为：分配15MB，分配30MB，回收15MB，分配8MB，分配6MB，此时主存中最大的空闲分区大小是（A）；若采用的是首次适应算法，则应该是（B）。**

A：（1）7MB　　（2）9MB　　（3）10MB　　（4）15MB

B：（1）7MB　　（2）9MB　　（3）10MB　　（4）15MB

**12、在伙伴系统中，一对空闲分区为伙伴是指（A）。**

A：（1）两个大小均为 $2^k$ B的相邻空闲分区 $2^{k+1}$

（2）两个大小可以相等或者不等，但均是2的幂的相邻空闲分区

（3）两个大小均为 $2^k$ B的相邻空闲分区，且前一个分区的起始地址是 $2^{k+1}$B的倍数

（4）两个大小均为 $2^k$ B的相邻空闲分区，且后一个分区的起始地址是 $2^{k+1}$B的倍数。

**13、在回收内存时可能出现下述四种情况：**

**释放区与插入点前一分区F1相邻接，此时应（A）；**

**释放区与插入点后一分区F2相邻接，此时应（B）；**

**释放区不与F1和F2相邻接，此时应（C）；**

**释放区既与F1相邻接，又与F2相邻接，此时应（D）。**

A，B，C，D:　（1）为回收区建立一分区表项，填上分区的大小和始址

（2）以F1分区的表项作为新表项且不做任何改变

（3）以F1分区的表项为新表项，但修改新表项的大小

（4）以F2分区的表项作为新表项，同时修改新表项的大小和始址

（5）以F1分区的表项为新表项，但修改新表项的大小且还要删除F2所对应的表项

**7、由连续分配方式发展为分页存储管理方式的主要推动力是（A）；由分页系统发展为分段系统，进而又发展为段页式系统的主要动力是（B）和（C）。**

A，B，C:（1）提高内存利用率

（2）提高系统吞吐量

（3）满足用户需要

（4）更好地满足多道程序运行的需要

（5）既满足用户要求，又提高内存利用率

**14、对外存对换区的管理应以（A）为主要目标，对外存文件区的管理应以（B）为主要目标。**

A：（1）提高系统吞吐量（2）提高存储空间的利用率（3）降低存储费用（4）提高换入换出速度

B：（1）提高系统吞吐量（2）提高存储空间的利用率（3）降低存储费用（4）提高换入换出速度