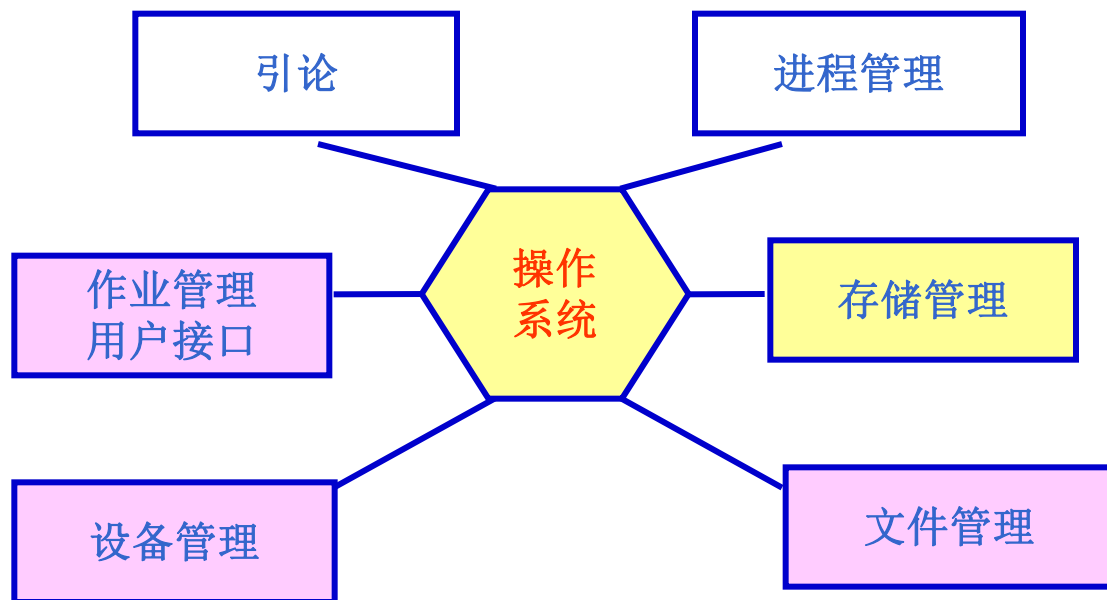
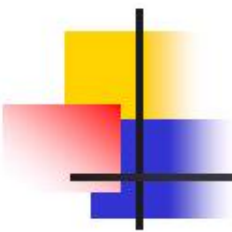




---

# 操作系统课程复习





# 主要知识点

- 存储管理基本概念
- 连续分区存储
- 分页存储管理
- 分段存储管理
- 虚拟存储器
- 请求分页存储管理
- 请求分段存储管理



# 主要知识点

- 存储管理基本概念
- 连续分区存储
- 分页存储管理
- 分段存储管理
- 虚拟存储器
- 请求分页存储管理
- 请求分段存储管理



## 主要知识点

- 逻辑地址与物理地址

- 编译程序在对一个源程序进行编译时，总是从0号单元开始为其分配地址，其他所有地址都是从这个开始地址顺序排下来的，这样的地址称为逻辑地址。地址空间是逻辑地址的集合。

- 一个编译好的程序采用的是逻辑地址，当他在计算机上运行时，要将“虚”的逻辑地址转换为“实”的物理地址，这个转换叫地址转换，也叫地址映射。物理地址的集合叫做存储空间。

- 地址空间是逻辑地址的集合

- 存储空间是物理地址的集合



# 主要知识点

- 存储管理基本概念
- 连续分区存储
- 分页存储管理
- 分段存储管理
- 虚拟存储器
- 请求分页存储管理
- 请求分段存储管理



# 主要知识点

- 单一连续分配方式
- 分区分配方式
  - *固定分区分配方式*
  - *动态分区分配方式*
    - ✓ 分区分配算法
    - ✓ 内存分配与回收
- 存储保护



# 主要知识点

- 单一连续分配方式
- 分区分配方式
  - 固定分区分配方式
  - 动态分区分配方式
    - ✓ 分区分配算法
    - ✓ 内存分配与回收
- 存储保护

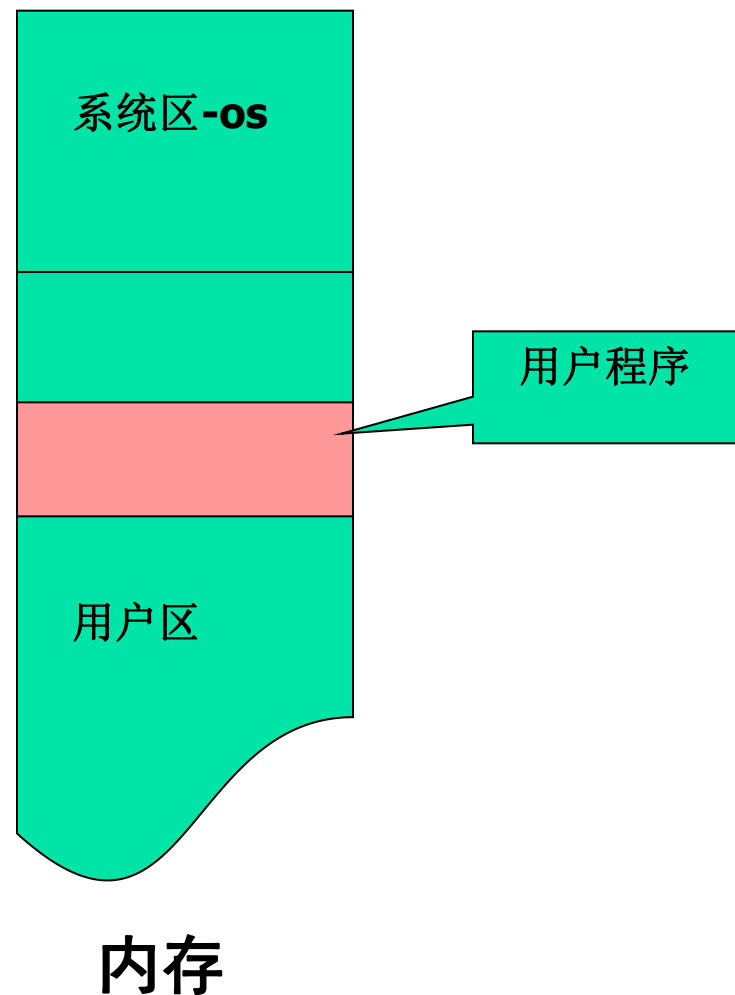


# 单一连续分配方式

最简单的一种存储管理方式，但只能用于单用户、单任务的OS中。

- **存储管理方法**：将内存分为系统区（内存低端，分配给OS用）和用户区（内存高端，分配给用户用）。采用静态分配方式，即作业一旦进入内存，就要等待它运行结束后才能释放内存。

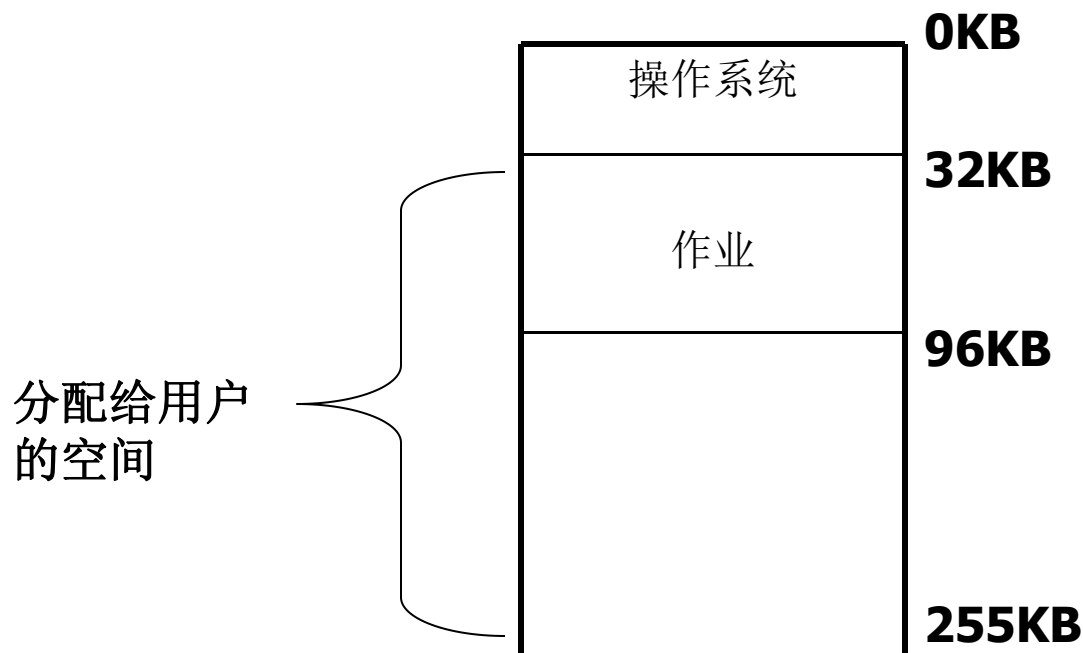
- **主要特点**：管理简单，只需少量的软件和硬件支持，便于用户了解和使用。但因内存中只装入一道作业运行，内存空间浪费大，各类资源的利用率也不高。





例：

- 一个容量为256KB的内存，操作系统占用32KB，剩下224KB全部分配给用户作业，如果一个作业仅需64KB，那么就有160KB的存储空间被浪费。





# 主要知识点

- 单一连续分配方式
- 分区分配方式
  - 固定分区分配方式
  - 动态分区分配方式
    - ✓ 分区分配算法
    - ✓ 内存分配与回收
- 覆盖与交换



# 分区分配方式存储管理

分区分配方式是满足多道程序设计需要的一种最简单的存储管理方法。

- 存储管理方法

将内存分成若干个分区（大小相等/不相等），除OS占用一个分区外，其余的每一个分区容纳一个用户程序。按分区的变化情况，可将分区存储管理进一步分为：

- 固定分区存储管理
- 动态分区存储管理



# 主要知识点

- 单一连续分配方式
- 分区分配方式
- **固定分区分配方式**
- **动态分区分配方式**
  - ✓ 分区分配算法
  - ✓ 内存分配与回收
- 存储保护



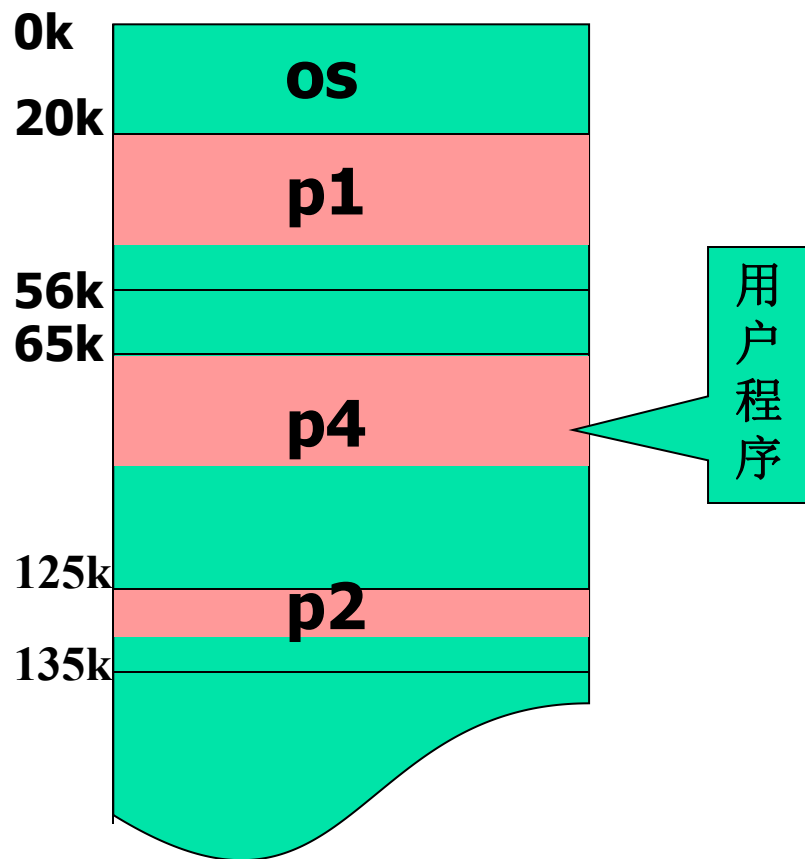
## 二、固定分区分配方式（固定分区存储管理）

是最早使用的一种可运行多道程序的存储管理方法。

- 存储管理方法

- **内存空间的划分**：将内存空间划分为若干个固定大小的分区，除OS占一分区外，其余的每一个分区装入一道程序。分区的大小可以相等，也可以不等，但事先必须确定，在运行时不能改变。即分区大小及边界在运行时不能改变。
- 系统需**建立一张分区说明表或使用表**，以记录分区号、分区大小、分区的起始地址及状态（已分配或未分配）。

# 固定分区分配方式示意图



区号	大小	起址	状态
1	36k	20k	已分配
2	9k	56k	未分配
3	60k	65k	已分配
4	10k	125k	已分配

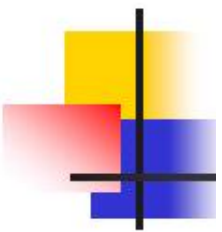
分区说明表



## ■ 内存分配

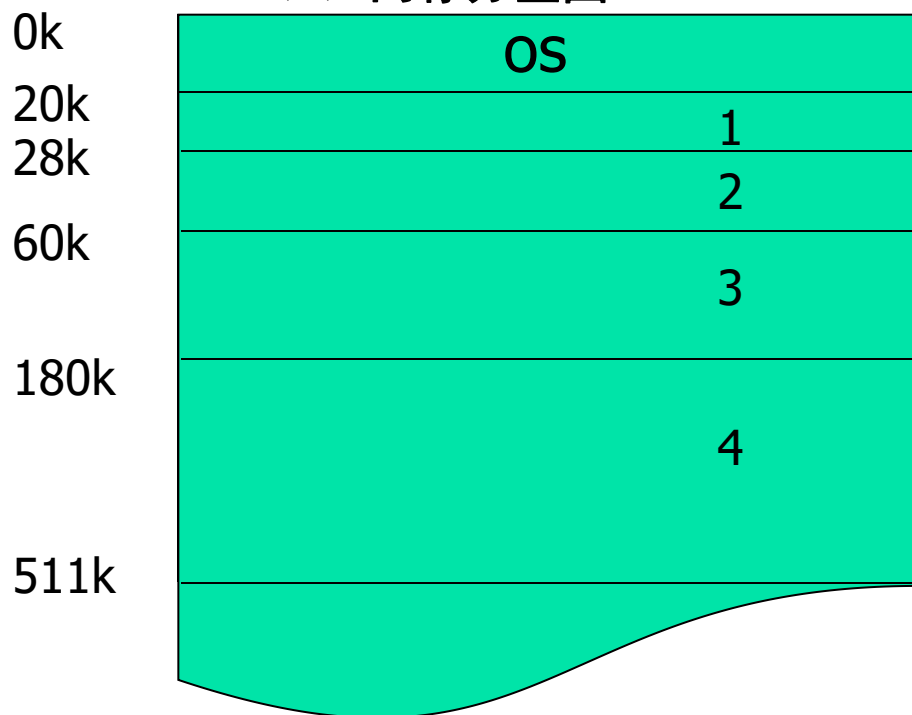
- 当某个用户程序要装入内存时，由内存分配程序检索分区说明表，从表中找出一个满足要求的尚未分配的分区分配给该程序，同时修改说明表中相应分区的状态；若找不到大小足够的分区，则拒绝为该程序分配内存。
- 当程序执行完毕，释放占用的分区，管理程序将修改说明表中相应分区的状态为未分配，实现内存资源的回收。
- 主要特点：管理简单，但因作业的大小并不一定与某个分区大小相等，从而使一部分存储空间被浪费。所以主存的利用率不高。
- 例 题





例：在某系统中，采用固定分区分配管理方式，内存分区（单位：字节）情况如图所示，现有大小为1K、9K、33K、121K的多个作业要求进入内存，试画出它们进入内存后的空间分配情况，并说明主存浪费多大？

(1) 内存分区图

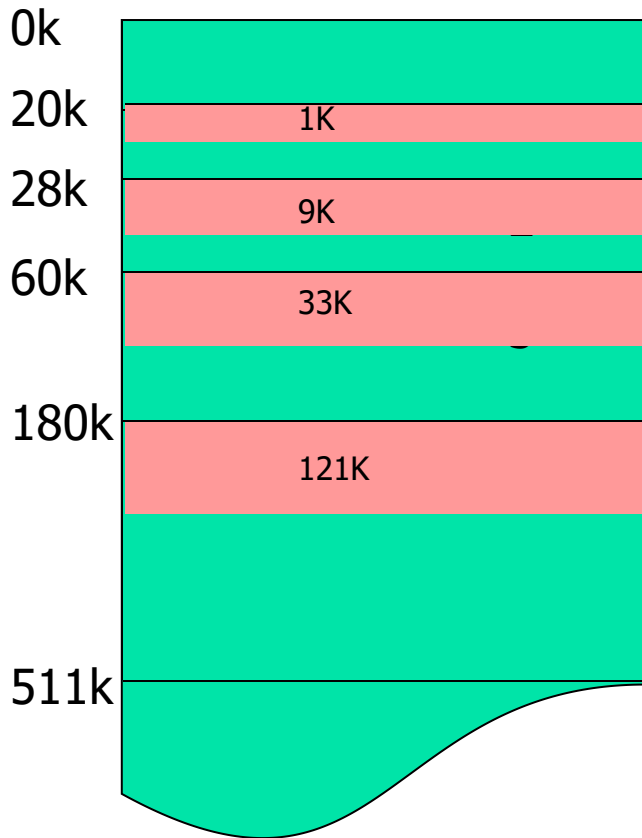


(2) 分区说明表

区号	大小	起址	状态
1	8k	20k	未分配
2	32k	28k	未分配
3	120k	60k	未分配
4	331k	180k	未分配

**解：**根据分区说明表，将4个分区依次分配给4个作业，同时修改分区说明表，其内存分配和分区说明表如下所示：

(1) 内存分配图



(2) 分区说明表

区号	大小	起址	状态
1	8k	20k	已分配
2	32k	28k	已分配
3	120k	60k	已分配
4	331k	180k	已分配

$$\begin{aligned} (3) \text{ 主存浪费空间} &= (8-1) + (32-9) + (120-33) + (331-121) \\ &= 7+23+87+210=327 (k) \end{aligned}$$



# 主要知识点

- 单一连续分配方式
- 分区分配方式
- 固定分区分配方式
- 动态分区分配方式
  - ✓ 分区分配算法
  - ✓ 内存分配与回收
- 存储保护



# 动态分区分配方式

动态分区分配又称为可变式分区分配，是一种动态划分存储器的分区方法。

## ■ 存储管理方法

不事先将内存划分成一块块的分区，而是在作业进入内存时，根据作业的大小动态地建立分区，并使分区的大小正好适应作业的需要。因此系统中分区的大小是可变的，分区的数目也是可变的。

## ■ 主要特点

管理简单，只需少量的软件和硬件支持，便于用户了解和使用。进程的大小与某个分区大小相等，从而主存的利用率有所提高。



# 1、分区分配中的数据结构（1）

## ■ 空闲分区表

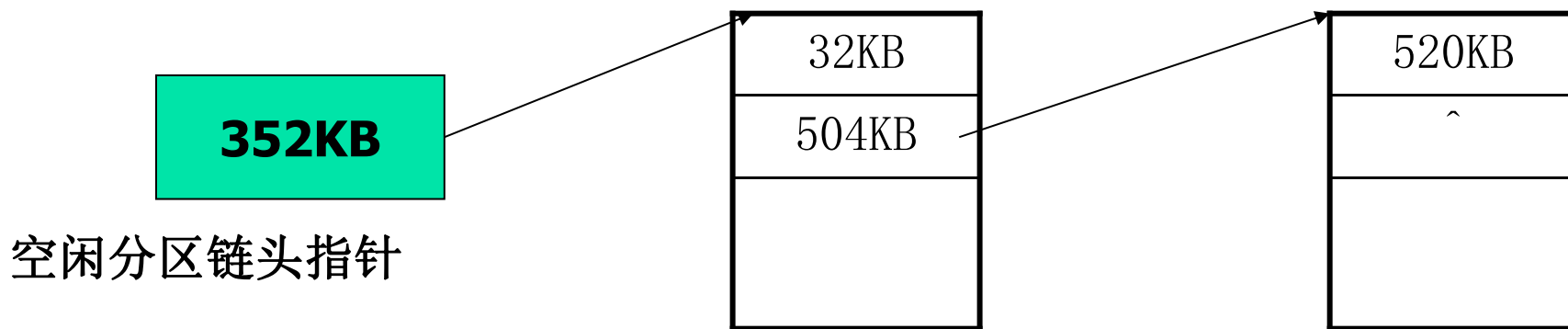
用来登记系统中的空闲分区（分区号、分区起始地址、分区大小及状态）。

分区号	大小KB	起始地址KB	状态
1	32	352	空闲
2	...	...	空表目
3	520	504	空闲
4	...	...	空表目
5	...	...	...

# 1、分区分配中的数据结构（2）

## ❖ 空闲分区链

用链头指针将系统中的空闲分区链接起来，构成空闲分区链。每个空闲分区的起始部分存放相应的控制信息（如大小，指向下一空闲分区的指针等）。





# 主要知识点

- 单一连续分配方式
- 分区分配方式
  - 固定分区分配方式
  - 动态分区分配方式
    - ✓ 分区分配算法
    - ✓ 内存分配与回收
- 存储保护



## 2、分区分配算法

为了将一个作业装入内存，应按照一定的分配算法从空闲分区表（链）中选出一个满足作业需求的分区分配给作业，如果这个空闲分区的容量比作业申请的空间要大，则将该分区一分为二，一部分分配给作业，剩下的部分仍然留在空闲分区表（链）中，同时修改空闲分区表（链）中相应的信息。目前常用分配算法有：

- ❑ 首次适应算法 (First Fit)
- ❑ 循环首次适应算法 (Next Fit)
- ❑ 最佳适应算法 (Best Fit)
- ❑ 最坏适应算法 (Worst Fit)





## 2、分区分配算法

- 首次适应算法 (First Fit)
- 循环首次适应算法 (Next Fit)
- 最佳适应算法 (Best Fit)
- 最坏适应算法 (Worst Fit)



# 首次适应算法（最先适应算法FF）

## ■ 算法

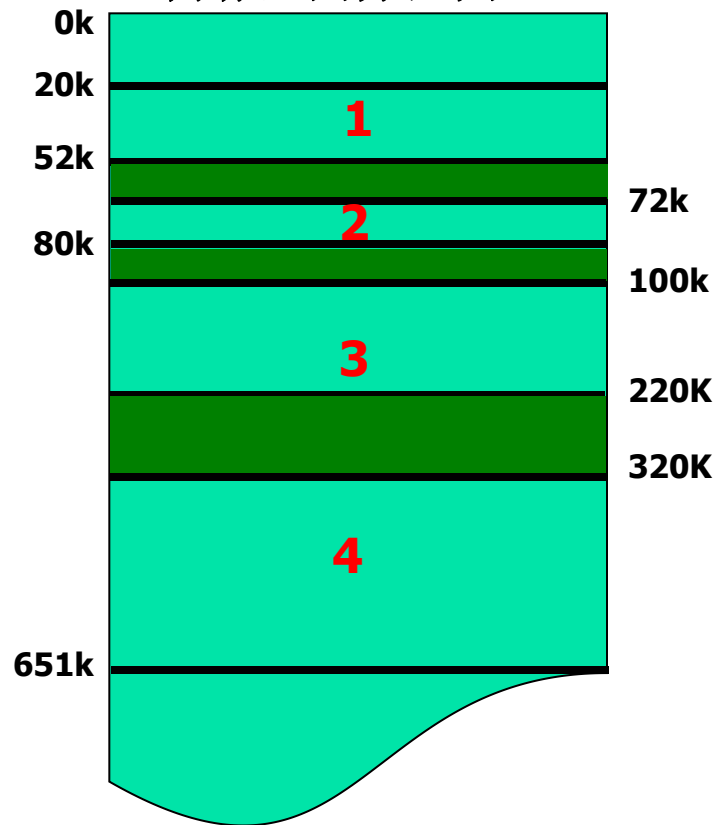
- 空分区（链）按地址递增的次序排列。
- 在进行内存分配时，从空闲分区表/链首开始顺序查找，直到找到第一个满足其大小要求的空闲分区为止。
- 然后按照作业大小，从该分区中划出一块内存空间分配给请求者，余下的空闲分区仍按地址递增的次序保留在空闲分区表（链）中。

**例：**系统中的空闲分区表如下，现有三个作业申请分配内存空间100KB、30KB及7KB。给出按首次适应算法的内存分配情况及分配后空闲分区表。

空闲分区表

区号	大小	起址
1	32k	20k
2	8k	72k
3	120k	100k
4	331k	320k

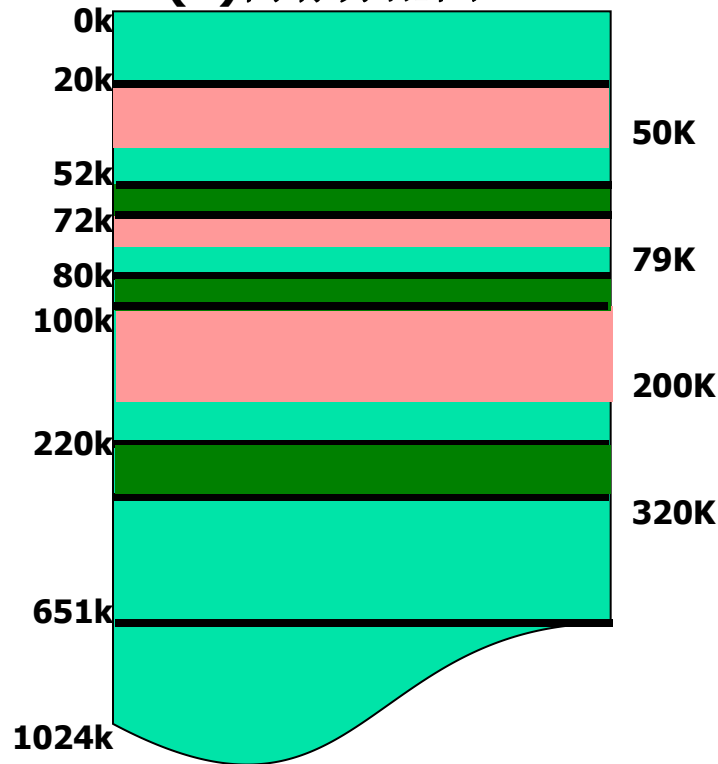
内存空闲分区图



**解：**按首次适应算法，  
申请作业100k，分配3号分区，剩下分区为20k，起始地址200K ；  
申请作业30k，分配1号分区，剩下分区为2k，起始地址50K ；  
申请作业7k，分配2号分区，剩下分区为1k，起始地址79K ；  
其内存分配图及分配后空闲分区表如下

区号	大小	起址
1	32k	20k
2	8k	72k
3	120k	100k
4	331k	320k

(1)内存分配图



(2)该算法分配后的空闲分区表

区号	大小	起址
1	2k	50k
2	1k	79k
3	20k	200k
4	331k	320k



## ❖ 首次适应算法的特点

优先利用内存低地址部分的空闲分区, 从而保留了高地址部分的大空闲区。

但由于低地址部分不断被划分, 致使低地址端留下许多难以利用的很小的空闲分区(碎片或零头), 而每次查找又都是从低地址部分开始, 这增加了查找可用空闲分区的开销。



## 2、分区分配算法

---

- ❑ 首次适应算法 (First Fit)
- ❑ 循环首次适应算法 (Next Fit)
- ❑ 最佳适应算法 (Best Fit)
- ❑ 最坏适应算法 (Worst Fit)



## 循环首次适应算法（NF）

### ■ 算法要求

又称为下次适应算法，由首次适应算法演变而来。在为作业分配内存空间时，不再每次从空闲分区表/链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，直到找到第一个能满足其大小要求的空闲分区为止。

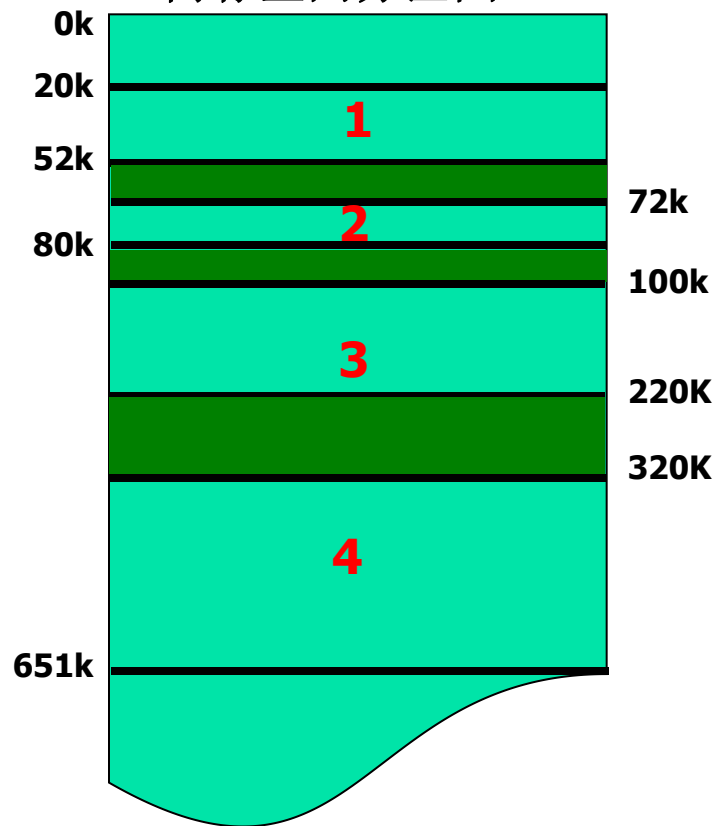
然后，再按照作业大小，从该分区中划出一块内存空间分配给请求者，余下的空闲分区仍按地址递增的次序保留在空闲分区表/链中。

**例：**系统中的空闲分区表如下，现有三个作业申请分配内存空间100KB、30KB及7KB。给出按循环首次适应算法的内存分配情况及分配后空闲分区表。

空闲分区表

区号	大小	起址
1	32k	20k
2	8k	72k
3	120k	100k
4	331k	320k

内存空闲分区图



**解：**按循环首次适应算法，

申请作业100k，分配3号分区，剩下分区为20k，起始地址200K；

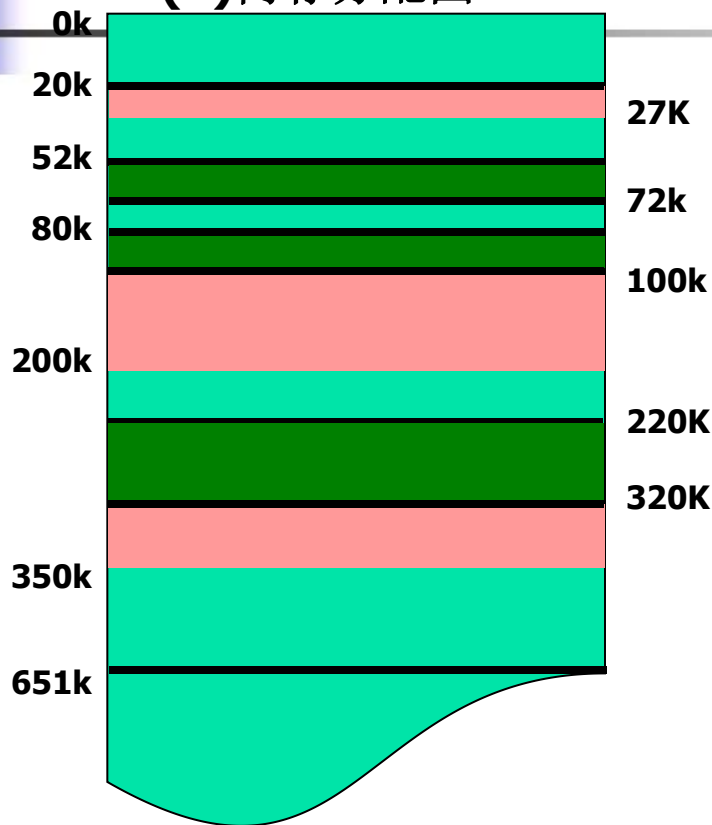
申请作业30k，分配4号分区，剩下分区为301k，起始地址350K；

申请作业7k，分配1号分区，剩下分区为25k，起始地址27K；

其内存分配图及分配后空闲分区表如下



(1)内存分配图



(2) 该算法分配后的空闲分区表

区号	大小	起址
1	25k	27k
2	8k	72k
3	20k	200k
4	301k	350k

## ❖ 算法特点

使存储空间的利用更加均衡，不致使小的空闲区集中在存储区的一端，但这会导致缺乏大的空闲分区。



## 2、分区分配算法

- 首次适应算法 (First Fit)
- 循环首次适应算法 (Next Fit)
- 最佳适应算法 (Best Fit)
- 最坏适应算法 (Worst Fit)



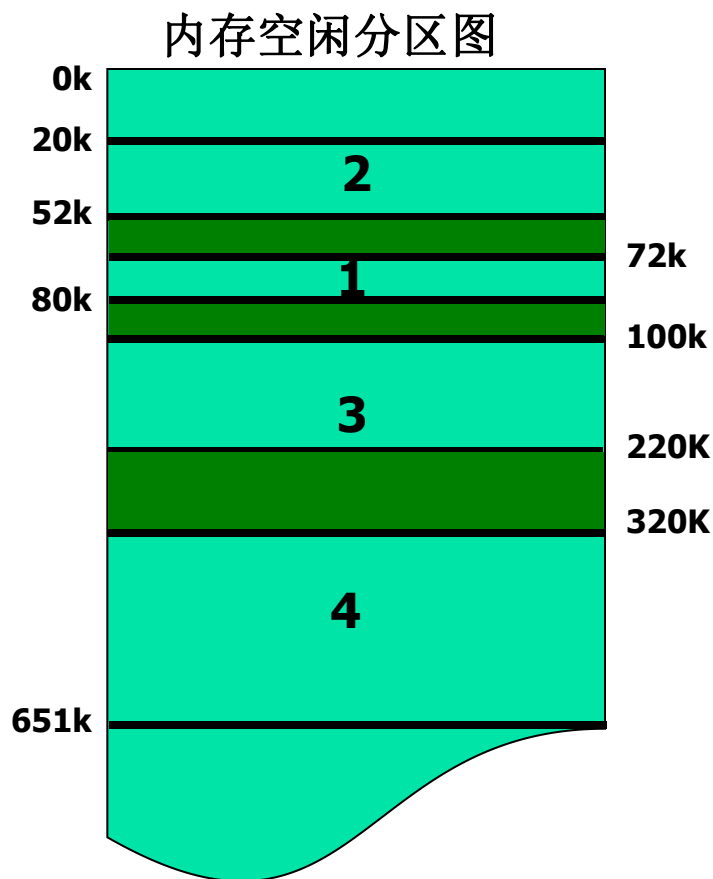
## 最佳适应算法 (BF)

### ■ 算法要求：

空闲分区表/链按容量大小递增的次序排列。在进行内存分配时，从空闲分区表/链首开始顺序查找，直到找到第一个满足其大小要求的空闲分区为止。

按这种方式为作业分配内存，就能把既满足作业要求又与作业大小最接近的空闲分区分配给作业。如果该空闲分区大于作业的大小，则与首次适应算法相同，将剩余空闲分区仍按容量大小递增的次序保留在空闲分区表/链中。

**例：**系统中的空闲分区表如下，现有三个作业申请分配内存空间100KB、30KB及7KB。给出按最佳适应算法的内存分配情况及分配后空闲分区表。



分配前的空闲分区表

区号	大小	起址
1	8k	72k
2	32k	20k
3	120k	100k
4	331k	320k



**解：**按最佳适应算法，分配前的空闲分区表如上表。

申请作业100k，分配3号分区，剩下分区为20k，起始地址200K；

申请作业30k，分配2号分区，剩下分区为2k，起始地址50K；

申请作业7k，分配1号分区，剩下分区为1k，起始地址79K；

其内存分配图及分配后空闲分区表如下

作业**100K**分配后的空闲分区表

区号	大小	起址
1	8k	72k
3	20k	200k
2	32k	20k
4	331k	320k

作业**30K**分配后的空闲分区表

区号	大小	起址
2	2k	50k
1	8k	72k
3	20k	200k
4	331k	320k

作业**7K**分配后的空闲分区表

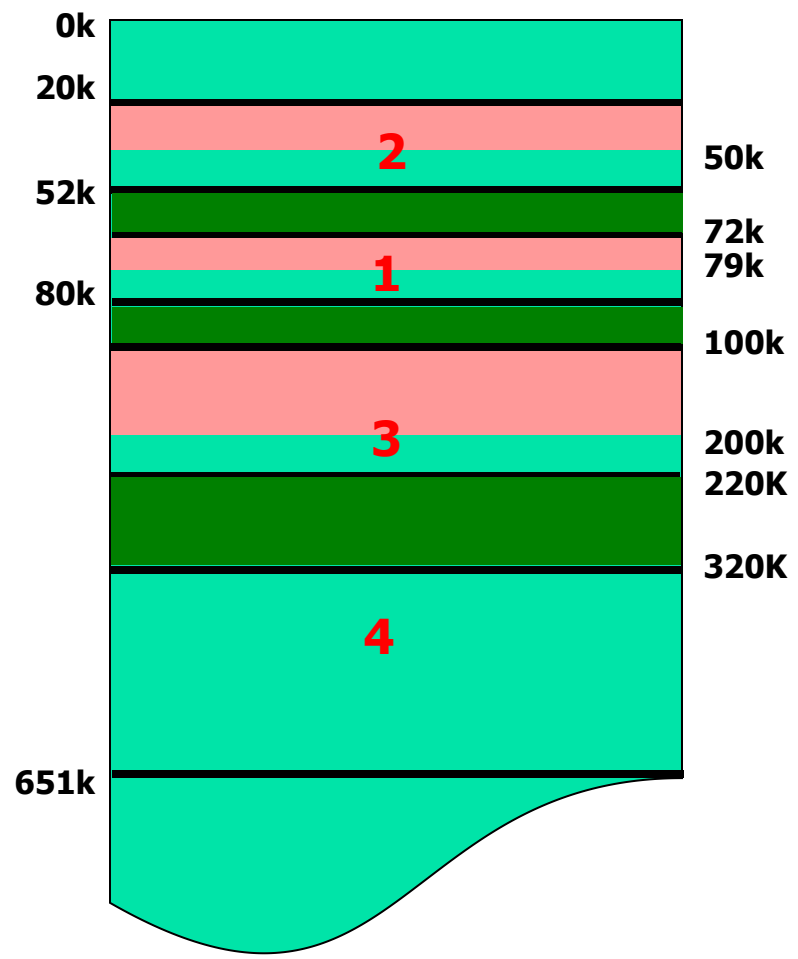
区号	大小	起址
1	1k	79k
2	2k	50k
3	20k	200k
4	331k	320k

区号	大小	起址
1	8k	72k
2	32k	20k
3	120k	100k
4	331k	320k



(1) 内存分配示意图

(2) 该算法分配后的空闲分区表



区号	大小	起址
1	1k	79k
2	2k	50k
3	20k	200k
4	331k	320k



## ❖ 算法特点

若存在与作业大小一致的空闲分区，则它必然被选中，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分区，但空闲区一般不可能正好和它申请的内存空间大小一样，因而将其分割成两部分时，往往使剩下的空闲区非常小，从而在存储器中留下许多难以利用的小空闲区（外碎片或外零头）。



## 2、分区分配算法

- ❑ 首次适应算法 (First Fit)
- ❑ 循环首次适应算法 (Next Fit)
- ❑ 最佳适应算法 (Best Fit)
- ❑ 最坏适应算法 (Worst Fit)





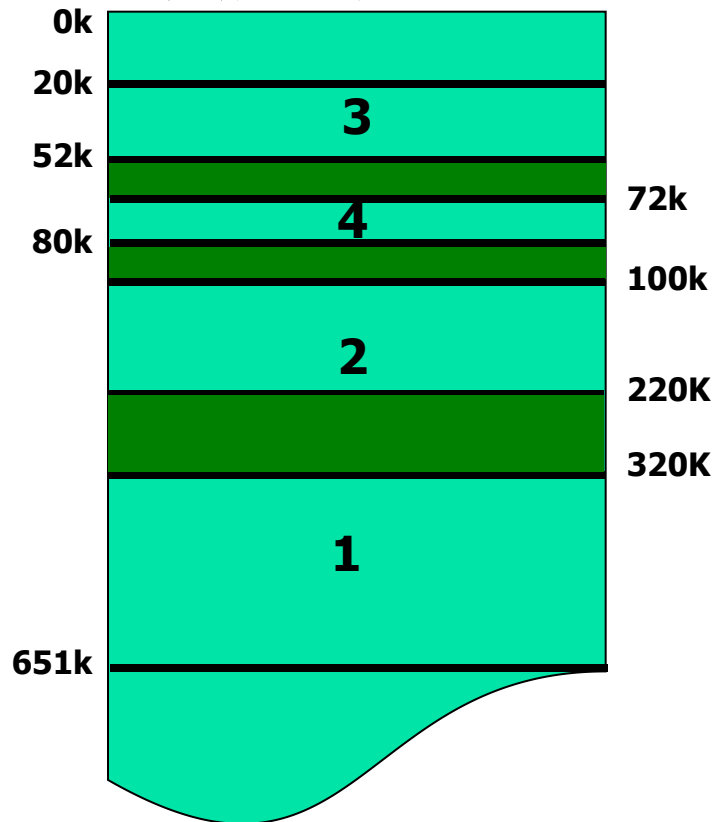
## 最坏适应算法 (WF)

### ■ 算法要求

空闲分区表/链按容量大小递减的次序排列。在进行内存分配时，从空闲分区表/链首开始顺序查找，找到的第一个能满足作业要求的空闲分区，一定是个最大的空闲区。这样可保证每次分割后的剩下的空闲分区不至于太小（还可被分配使用，以减少“外碎片”），仍把它按从大到小的次序保留在空闲分区表/链中。

**例：**系统中的空闲分区表如下，现有三个作业申请分配内存空间100KB、30KB及7KB。给出按最坏适应算法的内存分配情况及分配后空闲分区表。

内存空闲分区图



空闲分区表

区号	大小	起址
1	331k	320k
2	120k	100k
3	32k	20k
4	8k	72k

**解：**按最坏适应算法，分配前的空闲分区表如上表。

申请作业100k，分配1号分区，剩下分区为231k，起始地址420K；

申请作业30k，分配1号分区，剩下分区为201k，起始地址450K；

申请作业7k，分配1号分区，剩下分区为194k，起始地址457K；

其内存分配图及分配后空闲分区表如下

作业**100K**分配后的空闲分区表

区号	大小	起址
1	231k	420k
2	120k	100k
3	32k	20k
4	8k	72k

作业**30K**分配后的空闲分区表

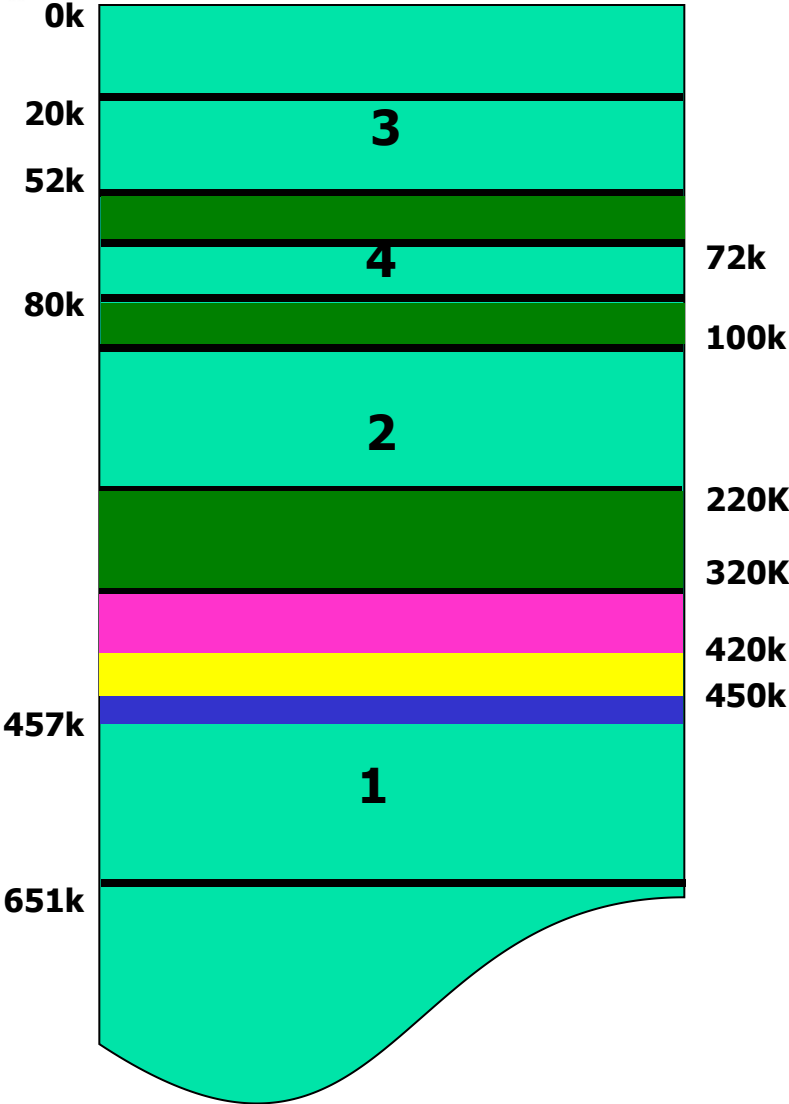
区号	大小	起址
1	201k	450k
2	120k	100k
3	32k	20k
4	8k	72k

作业**7K**分配后的空闲分区表

区号	大小	起址
1	194k	457k
2	120k	100k
3	32k	20k
4	8k	72k

区号	大小	起址
1	331k	320k
2	120k	100k
3	32k	20k
4	8k	72k

(1)内存分配图



(2) 该算法分配后的空闲分区表

区号	大小	起址
1	194k	457k
2	120k	100k
3	32k	20k
4	8k	72k



## ❖ 算法特点

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分区也较大，可装下其它作业。

但由于最大的空闲分区总是因首先分配而划分，当有大作业到来时，其存储空间的申请往往会得不到满足。



## 2、分区分配算法

- ❑ 首次适应算法 (First Fit)
- ❑ 循环首次适应算法 (Next Fit)
- ❑ 最佳适应算法 (Best Fit)
- ❑ 最坏适应算法 (Worst Fit)



# 主要知识点

- 单一连续分配方式
- 分区分配方式
- 固定分区分配方式
- 动态分区分配方式
  - ✓ 分区分配算法
  - ✓ 内存分配与回收
- 存储保护



## 分区分配操作\_分配内存和回收内存

### (1) 分配内存

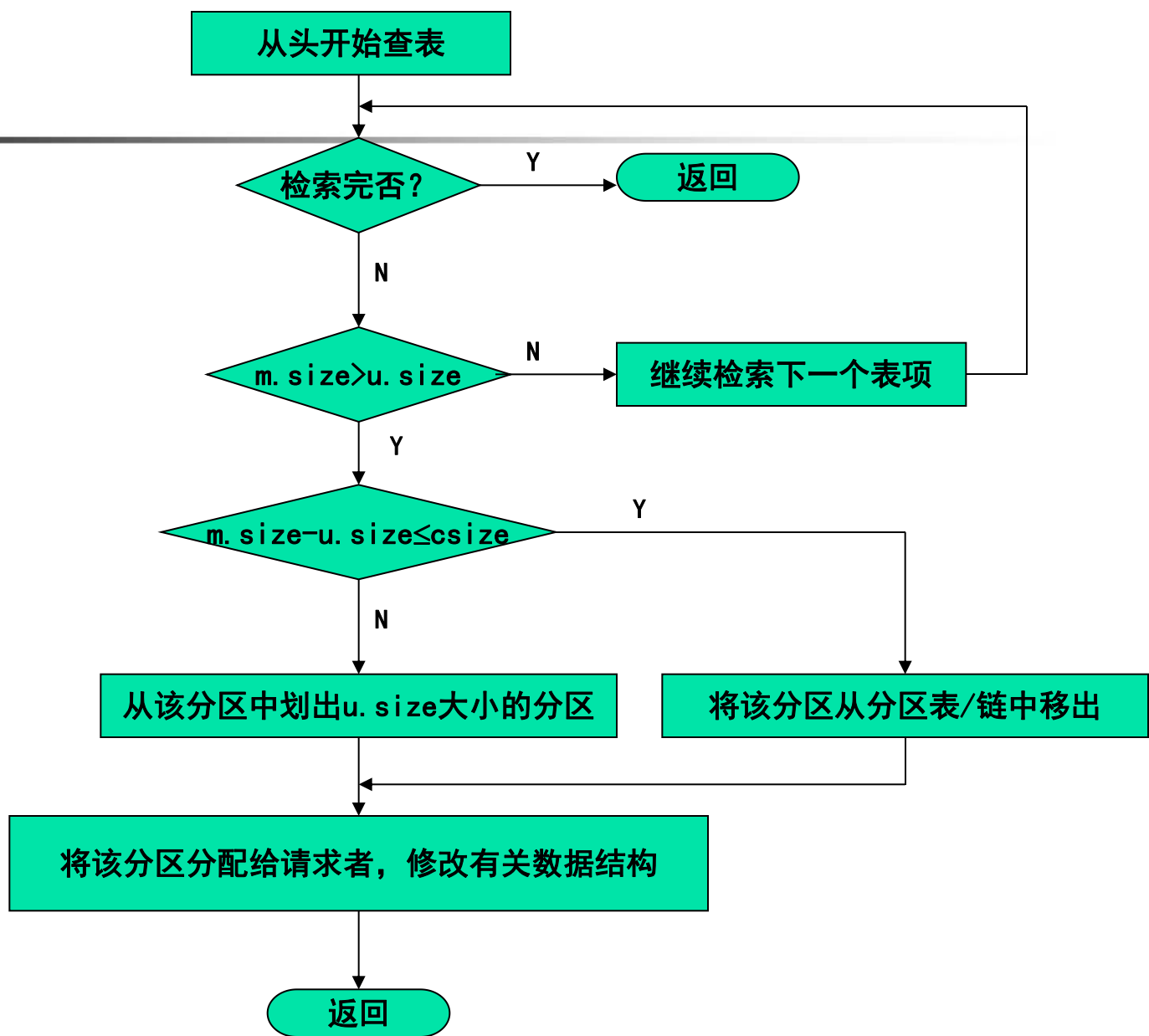
系统利用某种分配算法，从空闲分区表/链中找到所需大小的分区。

#### 分区的切割：

设请求的分区大小为 $u.size$ ，空闲分区的大小为 $m.size$ ，若 $m.size - u.size \leq csize$ （ $csize$ 是事先规定的不再切割的剩余分区的大小），说明多余部分太小，可不再切割，将整个分区分配给请求者；否则，从该分区中按请求的大小划分出一块内存空间分配出去，余下的部分仍留在空闲分区表/链中，然后，将分配区的首址返回给调用者。

分配流程图如下





内存分配流程图



## (2) 回收内存

当作业执行结束时，释放所占有的内存空间，OS应回收已使用完毕的内存分区。

系统根据回收分区的大小及首地址，在空闲分区表中检查是否有邻接的空闲分区，如有，则合成为一个大的空闲分区，然后修改有关的分区状态信息。

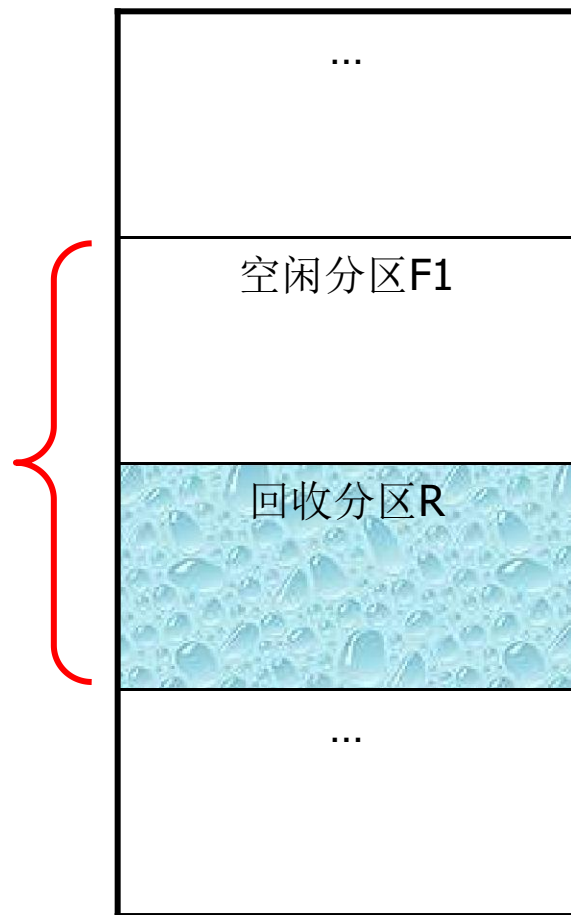
回收分区与已有空闲分区的相邻情况有以下四种：

## (2) 回收内存

- 1) 回收分区R上面邻接一个空闲分区F1，合并后首地址为空闲分区F1的首地址，大小为F1和R二者大小之和。

这种情况下，回收后空闲分区表中表项数不变。

内存回收情况

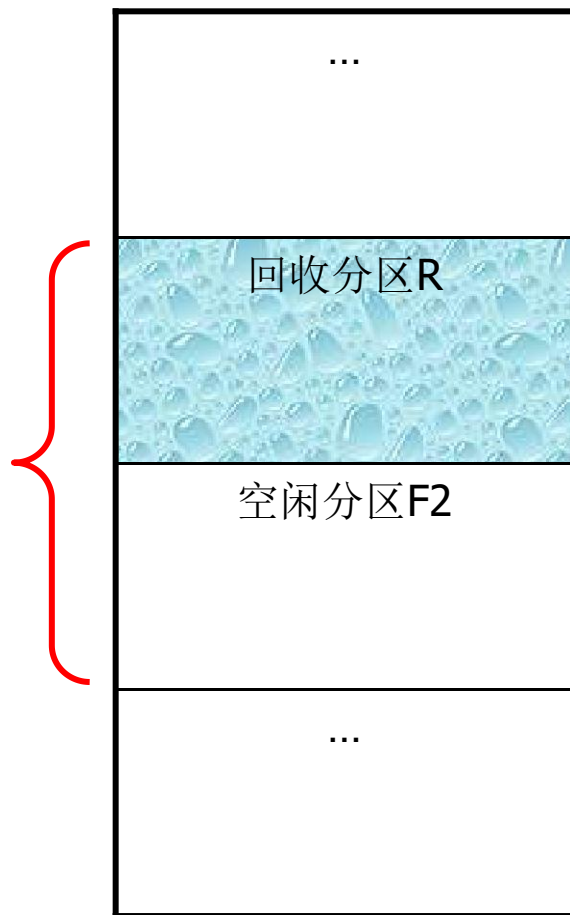


(a)

## (2) 回收内存

2) 回收分区R下面邻接一个空闲分区F2，合并后首地址为回收分区R的首地址，大小为R和F2二者大小之和。

这种情况下，回收后空闲分区表中表项数不变。

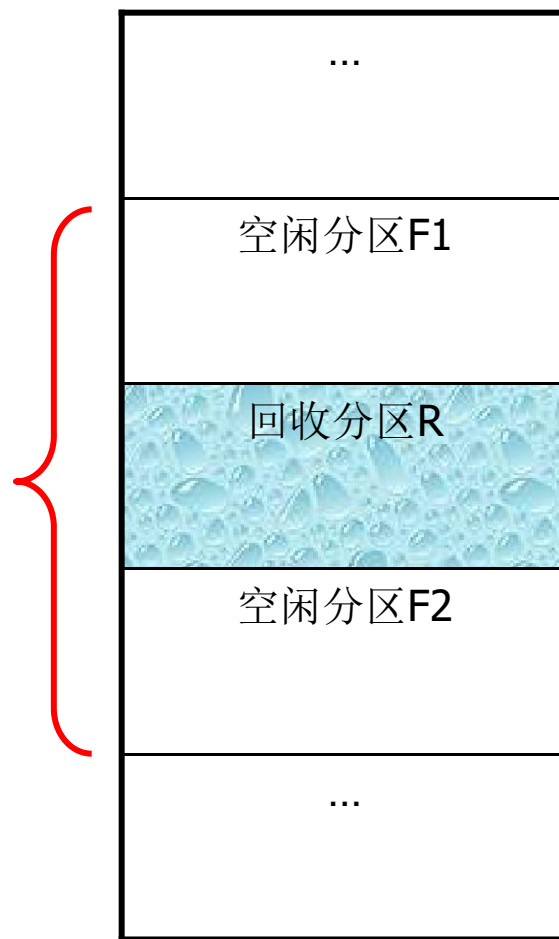


内存回收情况

(b)

## (2) 回收内存

3) 回收分区R上下邻接空闲分区F1和F2，合并后首地址为上空闲分区F1的首地址，大小为F1、R和F2三者大小之和。这种情况下，回收后空闲分区表中表项数不但没有增加，反而减少一项。



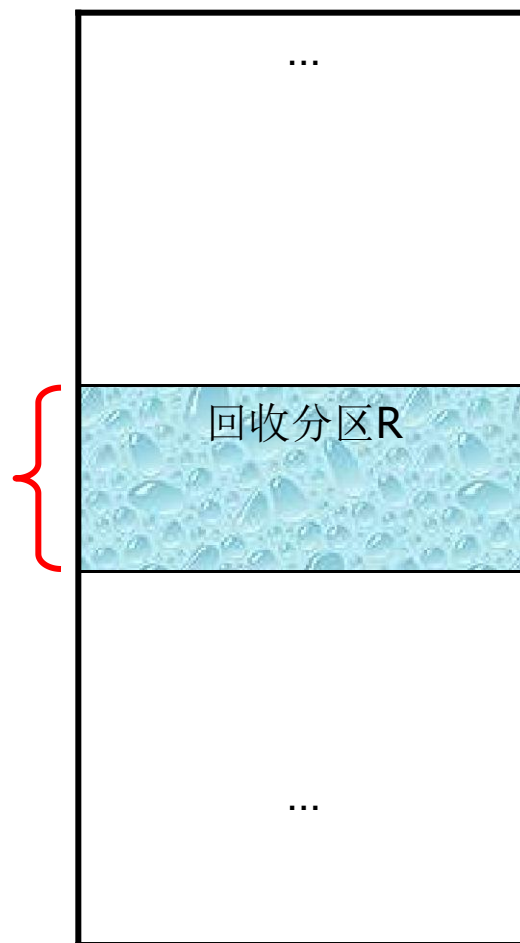
内存回收情况

(c)

## (2) 回收内存

4) 回收分区R不邻接空闲分区，这时在空闲分区表中新建一表项，并填写分区首地址、大小等信息。

这种情况下，回收后空闲分区表中表项数增加一项。



内存回收情况

(d)



# 主要知识点

- 单一连续分配方式
- 分区分配方式
  - *固定分区分配方式*
  - *动态分区分配方式*
    - ✓ 分区分配算法
    - ✓ 内存分配与回收
- 存储保护



# 分区的存储保护

存储保护是为了防止一个作业有意或无意地破坏操作系统或其它作业，常用的存储保护方法有：

## 1、界限寄存器方法

- **上下界寄存器方法**：用这两个寄存器分别存放作业的起始地址和结束地址。在作业运行过程中，将每一个访问内存的地址都同这两个寄存器的内容比较，如超出这个范围便产生保护性中断。





# 分区的存储保护

## 1、界限寄存器方法

- **基址、限长寄存器方法**：用这两个寄存器分别存放作业的起始地址和作业的地址空间长度。当作业执行时，将每一访问内存的相对地址和限长寄存器比较，如果超过了限长寄存器的值，则发出越界中断信号，并停止作业的运行。



# 分区的存储保护

## 2、存储保护键方法

给每个存储块（大小相同，一个分区为存储块的整数倍）分配一个单独的保护键，它相当于一把锁。

进入系统的每个作业也赋予一个保护键，它相当于一把钥匙。

当作业运行时，检查钥匙和锁是否匹配，如果不匹配，则系统发出保护性中断信号，停止作业运行。



# 主要知识点

- 存储管理基本概念
- 连续分区存储
- 分页存储管理
- 分段存储管理
- 虚拟存储器
- 请求分页存储管理
- 请求分段存储管理



# 主要知识点

- 分页管理基本概念
- 地址变换
- 快表处理
- 多级页表



# 主要知识点

- 分页管理基本概念
- 地址变换
- 快表处理
- 多级页表



# 基本分页存储管理方式

## ❑ 连续分配存储管理方式产生的问题

1. 会产生碎片
2. 把进程放在一个存储区中，要求一段较大并且连续的空间。

## ❑ 解决方法

(1) 碎片问题：拼接/紧凑技术——代价较高。

(2) 较大空间问题： 类比京东等电商的仓储问题

(2) 离散分配方式——允许将作业/进程离散放到多个不相邻接的分区中，就可以避免拼接。基于这一思想产生了以下的离散分配方式：

- ❖ 分页式存储管理：离散分配的基本单位是页
- ❖ 分段式存储管理：离散分配的基本单位是段

# 一、基本思想（1）

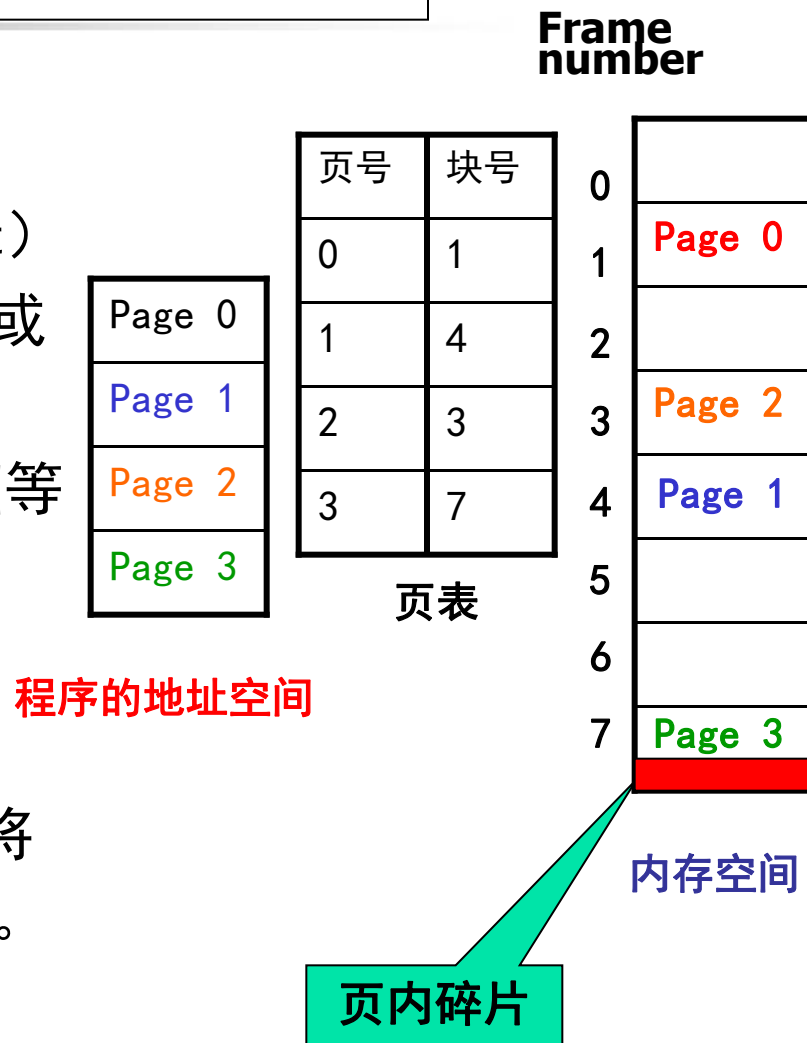
## ■ 空间划分

（1）将一个用户进程的地址空间（逻辑）划分成若干个大小相等的区域，称为页或页面，并为各页从0开始编号。

（2）内存空间也分成若干个与页大小相等的区域，称为（存储、物理）块或页框（frame），同样从0开始编号。

## ■ 内存分配

在为进程分配内存时，以块为单位，将进程中若干页装入到多个不相邻的块中。



注：需要CPU的硬件支持（地址变换机构）



## 一、基本思想（2）

页号	位移量（页内地址）
----	-----------

### ■ 页面大小——由地址结构决定

若页面较小：

- 减少页内碎片和内存碎片的总空间, 有利于提高内存利用率。
- 每个进程页面数增多, 从而使页表长度增加, 占用内存较大。
- 页面换进换出速度将降低。

若页面较大：

- 每个进程页面数减少, 页表长度减少, 占用内存就较小。
- 页面换进换出速度将提高。
- 会增加页内碎片, 不利于提高内存利用率。

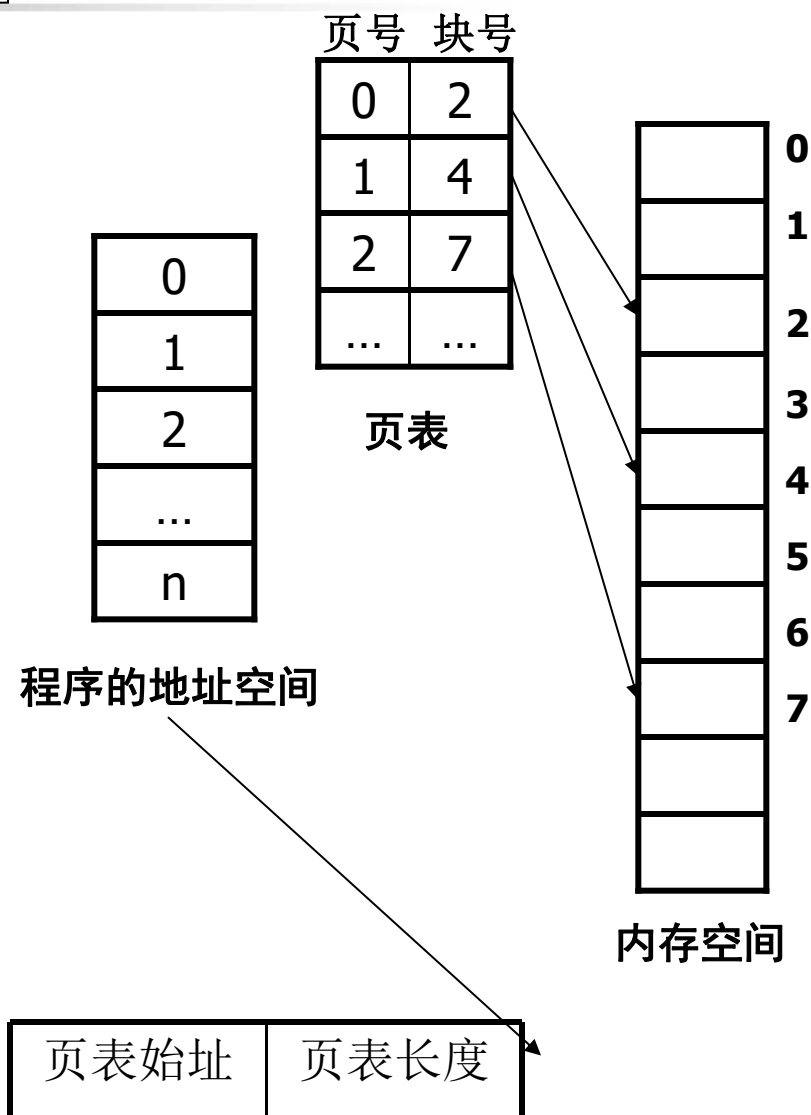
页面大小——选择适中, 通常为2的幂, 一般在512B-8KB之间。



## 二、页表

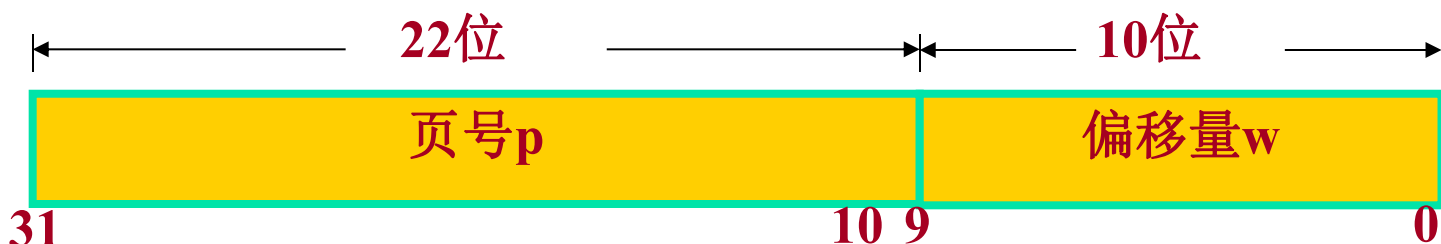
为了便于在内存找到进程的每个页面所对应的块，系统为每个进程建立一张页面映象，简称页表，如图。

- ❑ 记录了页面在内存中对应的块号
- ❑ 页表一般存放在内存中
- ❑ 页表的基址及长度由页表寄存器给出

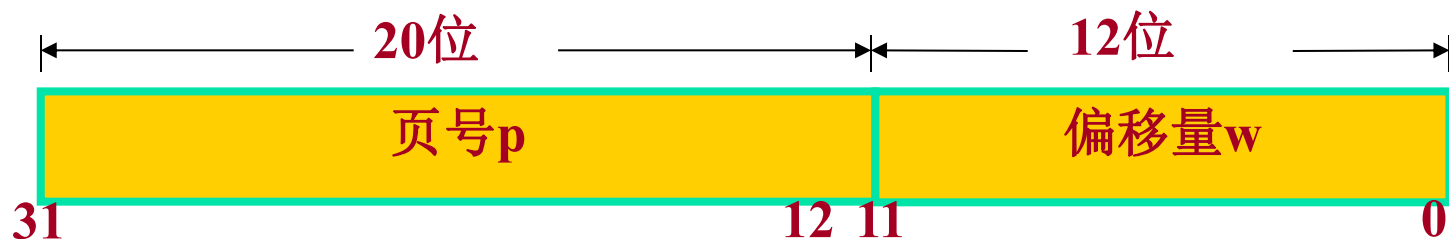


### 三、地址结构 (1)

逻辑地址：地址长为32位



(a) 页面大小为1K



(b) 页面大小为4K

### 三、地址结构（1）逻辑地址：

- 地址长为32位，其中0~11位为页内地址：



每页的大小为 $2^{12}=4\text{KB}$

12~31位为页号，地址空间最多允许有 $2^{20}=1\text{M}$ 页。

若给定一个逻辑地址空间中的地址为A，页面大小为L，则页号P和页内地址w可按下式求得：

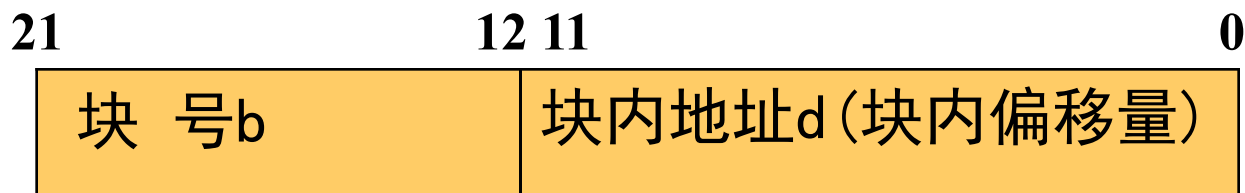
$$P = \text{INT}[A/L] \quad W = [A] \text{ MOD } L$$

其中，INT是整除函数，MOD是取余函数。

例：系统页面大小为 1 KB，设 $A=2170$ ，则 $P=2$ ,  $W=122$

### 三、地址结构（2）：物理地址

#### □ 物理地址：



地址长为22位，其中0~11位为块内地址，即每块的大小为 $2^{12}=4\text{KB}$ ，与页相等；

12~21位为块号，内存地址空间最多允许有 $2^{10}=1\text{K}$ 块。



# 主要知识点

- 分页管理基本概念
- 地址变换
- 快表处理
- 多级页表

# 地址结构例题

设有一页式存储管理系统，向用户提供的逻辑地址空间最大为16页，每页2048B，内存总共有8个存储块，试问逻辑地址和物理地址至少应为多少位？内存空间有多大？

解：（1）页式存储管理系统的逻辑地址为：

页号p

页内位移量w

其中页内地址表每页的大小即  $2048\text{B} = 2 \times 1024\text{B} = 2^{11}\text{B}$ ，所以页内地址为11位。

其中页号表最多允许的页数即  $16\text{页} = 2^4\text{页}$ ，所以页号为4位。

故逻辑地址至少应为15位。

（2）物理地址为：

块号b

块内位移d

其中块内地址表每块的大小与页大小相等，所以块内地址也为11位。

其中块号表内存空间最多允许的块数即  $8\text{块} = 2^3\text{块}$ ，所以块号为3位。

故内存空间至少应为14位，即  $2^{14} = 16\text{KB}$

## 地址变换例题

例1: 若在一分页存储管理系统中, 总页数为4, 总块数为8, 某作业的页表如表所示, 已知页面大小为1024B, 试将逻辑地址(十进制) 1011, 2148, 5012转化为相应的物理地址, 画出其地址转换图。

页号	块号
0	2
1	3
2	1
3	6

解: 由题知逻辑地址为:

页号p(2位)

位移量w(10位)

物理地址为:

块号b(3位)

块内位移d(10位)

(1) 逻辑地址1011的二进制表示为 00 1111110011

由此可知逻辑地址1011的页号0, 查页表知该页放在第2物理块中, 其物理地址的二进制表示为 010 1111110011

所以逻辑地址1011对应的物理地址为0BF3H. 其地址转换图如下页所示。

(2) 略

(3) 逻辑地址5012的二进制表示为: 100 1110010100

可知该逻辑地址的页号为4, 查页表知该页为不合法页, 则产生越界中断。



# 主要知识点

- 分页管理基本概念
- 地址变换
- 快表处理
- 多级页表





## 速度问题：TLB 快表

### ■分页方法存在的问题 1：速度问题

- 假设一条指令要把一个寄存器 中的数据复制到另一个寄存器。在不分页的情况下，这条指令只访问一次内存，即从内存中取指令。
- 有了分页后，则因为要访问页表而引起更多次的访问内存。由于执行速度通常被CPU从内存中取指令和数据的速度所限制，所以每次内存访问必须进行两次页表访问会降低一半的性能。在这种情况下，没人会采用分页机制。

### ■转换检测缓冲区(Translation Lookaside Buffer, TLB) 快表

- 这种解决方案的建立基于这样一种现象：大多数程序总是对少量的页面进行多次的访问，而不是相反的。因此，只有很少的页表项会被反复读取，而其他的页表项很少被访问。

# 地址变换机构

## -----具有快表的地址变换机构(1)

- 如果一个进程在虚拟地址19、20和21之间有一个循环，那么可能会生成下图中的TLB。
- 这三个表项中有可读和可执行的保护码。
- 当前主要使用的数据（假设是个数组）放在页面 129和页面130中。
- 页面140包含了用于数组计算的索引。
- 最后，堆栈位于页面861。
- 最终的TLB如图。

页号	块号
140	31
20	38
130	29
129	62
19	50
21	45
861	75

- 快表（联想寄存器、联想存储器、TLB）
  - 是一种特殊高速缓冲存储器。
  - 内容--为页表中的一部分或全部
  - CPU产生的逻辑地址的页首先在快表中寻找，若找到（命中），就找出其对应的物理块；若未找到（未命中），再到页表中找其对应的物理块，并将之复制到快表。
  - 若快表中内容满，则按某种算法淘汰某些页。



## TLB工作流程

- 将一个虚拟地址放入MMU (Memory Management Unit 内存管理单元) 中进行转换时，硬件首先通过将该虚拟页号与TLB中所有表项同时（即并行）进行匹配，判断虚拟页面是否在其中。如果发现了一个有效的匹配并且要进行的访问操作并不违反保护位，则将页框号直接从TLB中取出而不必再访问页表。
- 如果MMU检测到没有有效的匹配项时，就会进行正常的页表查询。接着从TLB中淘汰一个表项，然后用新找到的页表项代替它。这样，如果这一页面很快再被访问，第二次访问TLB时自然将会命中。

页号	块号
140	31
20	38
130	29
129	62
19	50
21	45
861	75



# 主要知识点

- 分页管理基本概念
- 地址变换
- 快表处理
- 多级页表 (理解)



# 主要知识点

- 存储管理基本概念
- 连续分区存储
- 分页存储管理
- 分段存储管理（了解）
- 虚拟存储器
- 请求分页存储管理
- 请求分段存储管理



# 主要知识点

- 存储管理基本概念
- 连续分区存储
- 分页存储管理
- 分段存储管理
- 虚拟存储器
- 请求分页存储管理
- 请求分段存储管理



# 虚拟存储器的基本概念

□ 常规存储管理方式的共同点：

要求一个作业全部装入内存后方能运行。

□ 问题：

(1) 有的作业很大, 所需内存空间大于内存总容量, 使作业无法运行。

(2) 有大量作业要求运行, 但内存容量不足以容纳下所有作业, 只能让一部分先运行, 其它在外存等待。

□ 解决方法 (1) 增加内存容量。

(2) 从逻辑上扩充内存容量

——虚拟存储器





# 一、虚拟存储器的引入(1)

- 常规存储器管理方式的特征

(1) 一次性:

作业在运行前需一次性地全部装入内存。将导致上述两问题。

(2) 驻留性:

作业装入内存后, 便一直驻留内存, 直至作业运行结束。



# 一、虚拟存储器的引入(2)

- 局部性原理

指程序在执行时呈现出局部性规律，即在一较短时间内，程序的执行仅限于某个部分，相应地，它所访问的存储空间也局限于某个区域。

局部性又表现为时间局部性（由于大量的循环操作，某指令或数据被访问后，则不久可能会被再次访问）和空间局部性（如顺序执行，指程序在一段时间内访问的地址，可能集中在一定的范围之内）。



# 虚拟存储器的概念（1）

- ◆ 基于局部性原理，程序在运行之前，没有必要全部装入内存，仅须将当前要运行的页（段）装入内存即可。
- ◆ 运行时，如访问的页（段）在内存中，则继续执行，如访问的页（段）未在内存中（缺页或缺段），则利用OS的请求调页（段）功能，将该页（段）调入内存。
- ◆ 如内存已满，则利用OS的页（段）置换功能，按某种置换算法将内存中的某页（段）调至外存，从而调入需访问的页。



## 虚拟存储器的概念（2）

虚拟存储器是指仅把作业的一部分装入内存便可运行作业的存储管理系统，它具有请求调页功能和页面置换功能，能从逻辑上对内存容量进行扩充，其逻辑容量由外存容量和内存容量之和决定，其最大容量由计算机的地址结构决定，其运行速度接近于内存，成本接近于外存。



## 二、虚拟存储器的实现方法

◆ 实现虚拟存储器必须解决好以下有关问题：

◆ 主存辅存统一管理问题

◆ 逻辑地址到物理地址的转换问题

◆ 部分装入和部分对换问题

◆ 虚拟存储管理主要采用以下技术实现：

◆ 请求分页存储管理

◆ 请求分段存储管理

◆ 请求段页式存储管理



# 主要知识点

- 存储管理基本概念
- 连续分区存储
- 分页存储管理
- 分段存储管理
- 虚拟存储器
- 请求分页存储管理
- 请求分段存储管理



# 主要知识点

- 请求分页存储管理概念
- 页面置换算法



# 请求分页存储管理方式

- **原理**——地址空间的划分与页式存储管理相同；装入页时，装入作业的一部分(即运行所需的)页即可运行。
  - 请求分页中的硬件支持
  - 请求分页中的页面调入策略





# 请求分页中的硬件支持

## 1、页表机制

页号	块号	状态位	访问位	修改位	外存地址
----	----	-----	-----	-----	------

(1) 状态位：指示该页是否已调入内存。是1，表示该表项是有效的，可以使用；是0，表示该表项对应的虚拟页面不在内存中，访问该页面会引起一个缺页中断。

(2) 访问位：不论是读还是写，系统都会在该页面被访问时设置访问位。它的值被用来帮助操作系统在发生缺页中断时选择要被淘汰的页面。

(3) 修改位：表示该页在调入内存后是否被修改过。若修改过，则换出时需重写至外存。

(4) 外存地址：指出该页在外存上的地址。



# 请求分页中的硬件支持


## 2、缺页中断机构

在请求分页系统中，当访问的页不在内存，便产生一缺页中断，请求OS将所缺页调入内存空闲块，若无空闲块，则需置换某一页，同时修改相应页表表目。

缺页中断与一般中断的区别：

- (1) 在指令执行期间产生和处理中断信号
- (2) 一条指令在执行期间，可能产生多次缺页中断





选择题：

在请求页式系统的页表中增加了若干项，其中修改位供（ D ）参考；状态位供（ C ）参考，访问位供（ B ）参考。

A.分配页面 B.置换算法 C.程序访问 D.换出页面 E.调入页面



## 虚拟存储器具有哪些基本特征？实现虚拟存储器的几个关键技术是什么？

### 基本特征：

- (1)多次性。作业只要部分装入内存便可启动执行，其余部分可待需要时再调入内存，即一个作业将分成多次装入内存。
- (2)对换性。在进程运行期间，允许将那些暂不使用的程序和数据从内存调至外存的对换区(换出)，待以后需要时再将它们从外存调入内存(换入)。
- (3)离散性。实现虚拟存储器必须采用离散的分配技术，而连续的分配技术无法实现虚拟存储器的功能。
- (4)虚拟性。虚拟存储器只是在逻辑上扩充内存容量，而实际的内存容量并没有真正扩大。

### 关键技术：

- (1)请求调页(段)技术：这是指及时将进程所要访问的、不在内存中的页(段)调入内存。该功能是由硬件(缺页(段)中断机构)发现缺页(段)和软件(将所需页(段)调入内存)配合实现的。
- (2)置换页(段)技术：当内存中已无足够空间用来装入即将调入的页(段)时，为了保证进程能继续运行，系统必须换出内存中的部分页(段)，以腾出足够的空间，将所需的页(段)调入内存。



# 主要知识点

- 请求分页存储管理概念
- 页面置换算法



# 请求分页中的页面置换算法

页面置换算法也称为页面淘汰算法，是用来选择换出页面的算法。页面置换算法的优劣直接影响到系统的效率，若选择不合适，可能会出现以下现象：

刚被淘汰出内存的页面，过后不久又要访问它，需要再次将其调入，而该页调入内存后不久又再次被淘汰出内存，然后又要访问它，如此反复，使得系统把大部分时间用在了页面的调进换出上，而几乎不能完成任何有效的工作，这种现象称为抖动（又称颠簸）。



# 请求分页中的页面置换算法

- 最佳置换算法
- 先进先出置换算法
- 最近最少使用置换算法
- Clock置换算法
- \*其它算法





# 请求分页中的页面置换算法

- 最佳置换算法
- 先进先出置换算法
- 最近最少使用置换算法
- Clock置换算法
- \*其它算法



# 最佳置换算法

- 很容易就可以描述出最好的页面置换算法，虽然此算法不可能实现。该算法是这样工作的：在缺页中断发生时，有些页面在内存中，其中有一个页面（包含紧接着的下一条指令的那个页面）将很快被访问，其他页面则可能要到10、100或1000条指令后才会被访问，每个页面都可以用在该页面首次被访问前所要执行的指令数作为标记。
- 最优页面置换算法规定应该置换标记最大的页面。如果一个页面在800万条指令内不会被使用，另外一个页面在600万条指令内不会被使用，则置换前一个页面，从而把因需要调入这个页面而发生的缺页中断推迟到将来，越久越好。计算机也像人一样，希望把不愉快的事情尽可能地往后拖延。
- 这个算法惟一的问题就是它是无法实现的。当缺页中断发生时，操作系统无法知道各个页面下一次将在什么时候被访问。
- 用这种方式，我们可以通过最优页面置换算法对其他可实现算法的性能进行比较。如果操作系统达到的页面置换性能只比最优算法差1%，那么即使花费大量的精力来寻找更好的算法最多也只能换来1% 的性能提高。

## 最佳置换算法例

假定系统为某进程分配了3个物理块，进程运行时的页面走向为 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5，开始时3个物理块均为空，计算采用最佳置换页面淘汰算法时的缺页率？

页面走向	1	2	3	4	1	2	5	1	2	3	4	5
物理块1	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>			<b>1</b>			<b>3</b>	<b>3</b>	
物理块2		<b>2</b>	<b>2</b>	<b>2</b>			<b>2</b>			<b>2</b>	<b>4</b>	
物理块3			<b>3</b>	<b>4</b>			<b>5</b>			<b>5</b>	<b>5</b>	
缺页	缺	缺	缺	缺			缺			缺	缺	

缺页率=7/12

**注：**实际上这种算法无法实现，因页面访问的未来顺序很难精确预测，但可用该算法评价其它算法的优劣。



# 请求分页中的页面置换算法

- 最佳置换算法
- 先进先出置换算法
- 最近最少使用置换算法
- Clock置换算法
- \*其它算法



# 先进先出置换算法

- 另一种开销较小的页面置换算法是FIFO（First-In First-Out，先进先出）算法。为了解释它是怎样工作的，我们设想有一个超级市场，它有足够的货架能展示 $k$ 种不同的商品。有一天，某家公司介绍了一种新的方便食品—即食的、冷冻干燥的、可以用微波炉加热的酸乳酪，这个产品非常成功，所以容量有限的超市必须撤掉一种旧的商品以便能够展示该新产品。
- 一种可能的解决方法就是找到该超级市场中库存时间最长的商品并将其替换掉（比如某种120年以前就开始卖的商品），理由是现在已经没有人喜欢它了。这实际上相当于超级市场有一个按照引进时间 排列的所有商品的链表。新的商品被加到链表的尾部，链表头上的商品则被撤掉。
- 同样的思想也可以应用在页面置换算法中。由操作系统维护一个所有当前在内存中的页面的链表，最新进入的页面放在表尾，最久进入的页面放在表头。当发生缺页中断时，淘汰表头的页面并把新调入的页面加到表尾。
- 当FIFO用在超级市场时，可能会淘汰剃须膏，但也可能淘汰面粉、盐或黄油这一类常用商品。因此，当它应用在计算机上时也会引起同样的问题，由于这一原因，很少使用纯粹的FIFO算法。

# 先进先出置换算法例题

1、假定系统为某进程分配了3个物理块，进程运行时的页面走向为 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5，开始时3个物理块均为空，计算采用先进先出页面淘汰算法时的缺页率？

页面走向	1	2	3	4	1	2	5	1	2	3	4	5
物理块1	1	1	1	4	4	4	5			5	5	
物理块2		2	2	2	1	1	1			3	3	
物理块3			3	3	3	2	2			2	4	
缺页	缺	缺	缺	缺	缺	缺	缺			缺	缺	

缺页率=9/12

## 先进先出置换算法例题

2、假定系统为某进程分配了4个物理块，进程运行时的页面走向为 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5，开始时4个物理块均为空，计算采用先进先出页面淘汰算法时的缺页率？

页面走向	1	2	3	4	1	2	5	1	2	3	4	5
物理块1	1	1	1	1			5	5	5	5	4	4
物理块2		2	2	2			2	1	1	1	1	5
物理块3			3	3			3	3	2	2	2	2
物理块4				4			4	4	4	3	3	3
缺页	缺	缺	缺	缺			缺	缺	缺	缺	缺	缺

缺页率=10/12

## 先进先出置换算法例题

3、假定系统为某进程分配了5个物理块，进程运行时的页面走向为 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5，开始时5个物理块均为空，计算采用先进先出页面淘汰算法时的缺页率？

缺页率=5/12

页面走向	1	2	3	4	1	2	5	1	2	3	4	5
物理块1	1	1	1	1			1					
物理块2		2	2	2			2					
物理块3			3	3			3					
物理块4				4			4					
物理块5							5					
缺页	缺	缺	缺	缺			缺					



## 先进先出置换算法\_注（1）：

- 1、该算法的出发点是最早调入内存的页面不再被访问的可能性会大一些。
- 2、该算法实现比较简单，对具有线性顺序访问的程序比较合适，而对其他情况效率不高。因为经常被访问的页面，往往在内存中停留最久，结果这些常用的页面却因变老而被淘汰。

页面走向	1	2	3	4	1	2	5	1	2	3	4	5
物理块1	1	1	1	4	4	4	5			5	5	
物理块2		2	2	2	1	1	1			3	3	
物理块3			3	3	3	2	2			2	4	
缺页	缺	缺	缺	缺	缺	缺	缺			缺	缺	



## 先进先出置换算法\_注（2）：

**3、**先进先出算法存在一种异常现象，即在某些情况下会出现分配给的进程物理块数增多，缺页次数有时增加，有时减少的奇怪现象，这种现象称为 **Belady现象**。如上几例：

物理块数	3	4	5
缺页次数	9	10	5



# 请求分页中的页面置换算法

- 最佳置换算法
- 先进先出置换算法
- 最近最少使用置换算法
- Clock置换算法
- \*其它算法



# 最近最少使用算法

- 对最优算法的一个很好的近似是基于这样的观察：在前面几条指令中频繁使用的页面很可能在后面的几条指令中被使用。反过来说，已经很久没有使用的页面很有可能在未来较长的一段时间内仍然不会被使用。这个思想提示了一个可实现的算法：在缺页中断发生时，置换未使用时间最长的页面。这个策略称为LRU（Least Recently Used，最近最少使用）页面置换算法。
- 虽然LRU在理论上是可以实现的，但代价很高。为了完全实现LRU，需要在内存中维护一个所有页面的链表，最近最多使用的页面在表头，最近最少使用的页面在表尾。困难的是在每次访问内存时都必须更新整个链表。在链表中找到一个页面，删除它，然后把它移动到表头是一个非常费时的操作，即使使用硬件实现也一样费时。
- 最近最少使用置换算法的思想是当发生缺页时，系统会选择当前内存页面中没有被使用时间最久的那一页，即最少使用的那一页，并将它置换出去。

## 最近最少使用算法例

假定系统为某进程分配了3个物理块，进程运行时的页面走向为 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5，开始时3个物理块均为空，计算采用最近最少使用页面淘汰算法时的缺页率？

页面走向	1	2	3	4	1	2	5	1	2	3	4	5
物理块1	<b>1</b>	<b>1</b>	<b>1</b>	<b>4</b>	<b>4</b>	<b>4</b>	<b>5</b>			<b>3</b>	<b>3</b>	<b>3</b>
物理块2		<b>2</b>	<b>2</b>	<b>2</b>	<b>1</b>	<b>1</b>	<b>1</b>			<b>1</b>	<b>4</b>	<b>4</b>
物理块3			<b>3</b>	<b>3</b>	<b>3</b>	<b>2</b>	<b>2</b>			<b>2</b>	<b>2</b>	<b>5</b>
缺页	缺	缺	缺	缺	缺	缺	缺			缺	缺	缺

缺页率=10/12



# 最近最少使用算法\_注 (1)

---

❖ **该算法的出发点**：如果某个页面被访问了，则它可能马上还要被访问。

反之，如果很长时间未被访问，则它在最近一段时间也不会被访问。

❖ **该算法的性能接近于最佳算法，但实现起来**代价很高：

❖ 为了完全实现LRU，需要在内存中维护一个所有页面的链表，最近最多使用的页面在表头，最近最少使用的页面在表尾。困难的是在每次访问内存时都必须要更新整个链表。在链表中找到一个页面，删除它，然后把它移动到表头是一个非常费时的操作，即使使用硬件实现也一样费时。

❖ **所以在实际系统中往往使用该算法的近似算法。**

[illegible]

# 最近最少使用算法\_注 (2)

## ❖ 方法2 硬件方法:

- 每次访问页面时，先将对应行置1，再将对应列置0。
- 置换页面时，选取行值最小对应的页面。

页面访问顺序 0, 1, 2, 3, 2, 1, 0, 3, 2, 3

	Page			
	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

(a)

	Page			
	0	1	2	3
0	0	0	1	1
1	1	0	1	1
2	0	0	0	0
3	0	0	0	0

(b)

	Page			
	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	1	1	0	1
3	0	0	0	0

(c)

	Page			
	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0

(d)

	Page			
	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	1
3	1	1	0	0

(e)

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

(f)

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

(g)

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

(h)

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

(i)

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

(j)





## 最近最少使用算法\_注 (2)

---

❖ 该算法的近似算法实现：

方法3：软件模拟LRU算法，称为最不常用算法 (Not Frequently Used, NFU)，用频率近似时间。



## 单页此外编辑母版标题样式

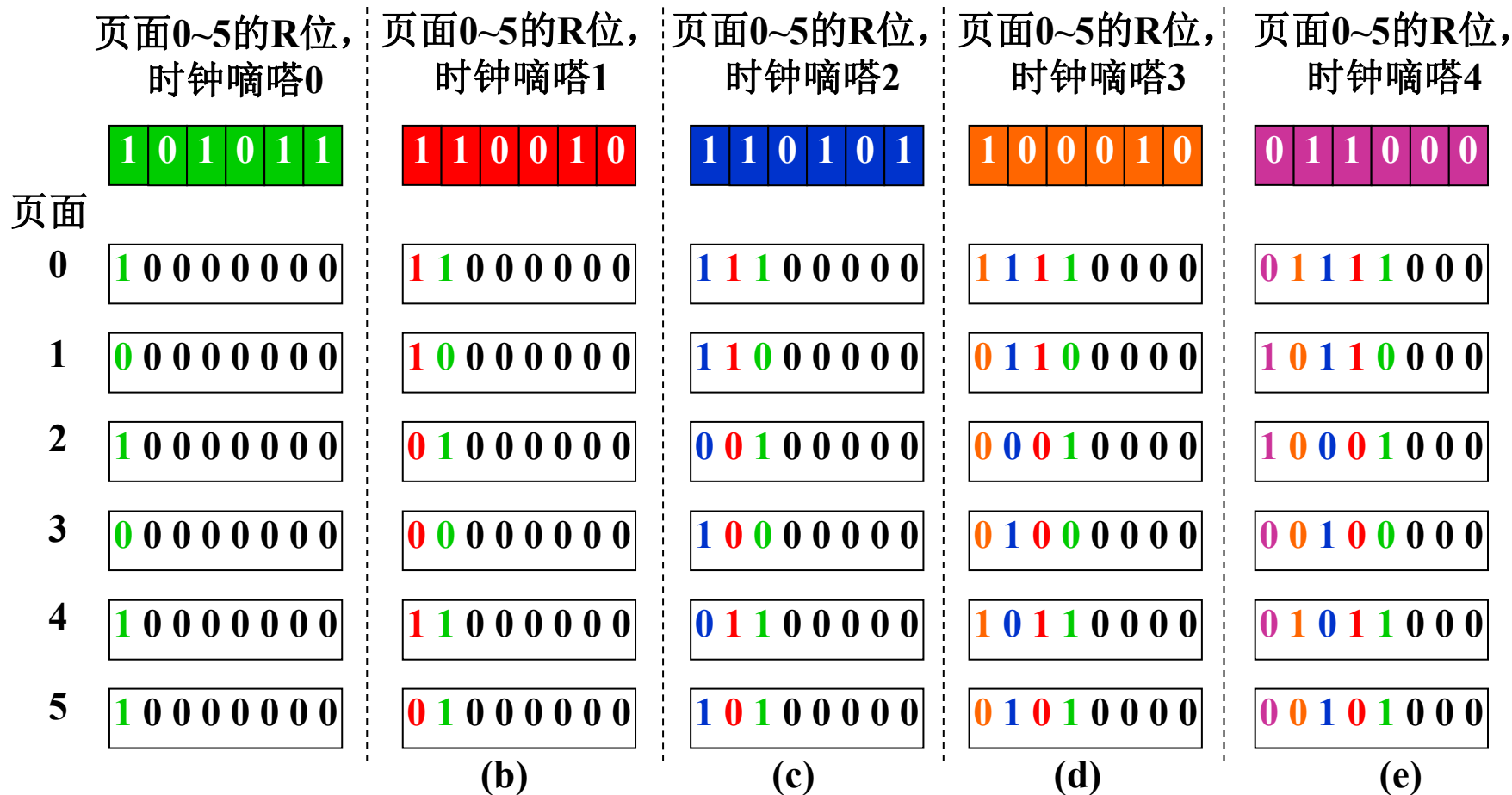
- **NFU算法**：将每个页面与一个软件计数器相联，计数器的初值为**0**。每次时钟中断时，由操作系统扫描内存中的所有的页面，将每个页面的**R**位（它的值是**0**或**1**）加到它的计数器上。这个计数器大体上跟踪了各个页面被访问的频繁程度。发生缺页中断时，则置换计数器值最小的页面。
- 存在问题：
  - **NFU**的主要问题是它从来不忘记任何事情。比如，在一个多次（扫描）编译器中，在第一次扫描中被频繁使用的页面在程序进入第二次扫描时，其计数器的值可能仍然很高。
  - 实际上，如果第一次扫描的执行时间恰好是各次扫描中最长的，含有以后各次扫描代码的页面的计数器可能总是比含有第一次扫描代码的页面小，结果是操作系统将置换有用的页面而不是不再使用的页面。



# 单击此外编辑母版标题样式

- 幸运的是只需对NFU做一个小小的修改就能使它很好地模拟LRU。
- 修改分为两部分：
  - 首先，在R位被加进之前先将计数器右移一位；
  - 其次，将R位加到计数器最左端的位而不是最右端的位。
- 修改以后的算法称为老化（aging）算法。
- 计数器记录了各个页面的访问频繁程度。
- 发生缺页中断时，将置换计数器值最小的页面。如果一个页面在前面4个时钟滴答中都没有访问过，那么它的计数器最前面应该有4个连续的0，因此它的值肯定要比在前面三个时钟滴答中都没有被访问过的页面的计数器值小。

# 单击此处编辑母版标题样式



用软件模拟LRU的老化算法, 图中所示是6个页面5个时钟嘀嗒的情况



# 请求分页中的页面置换算法

---

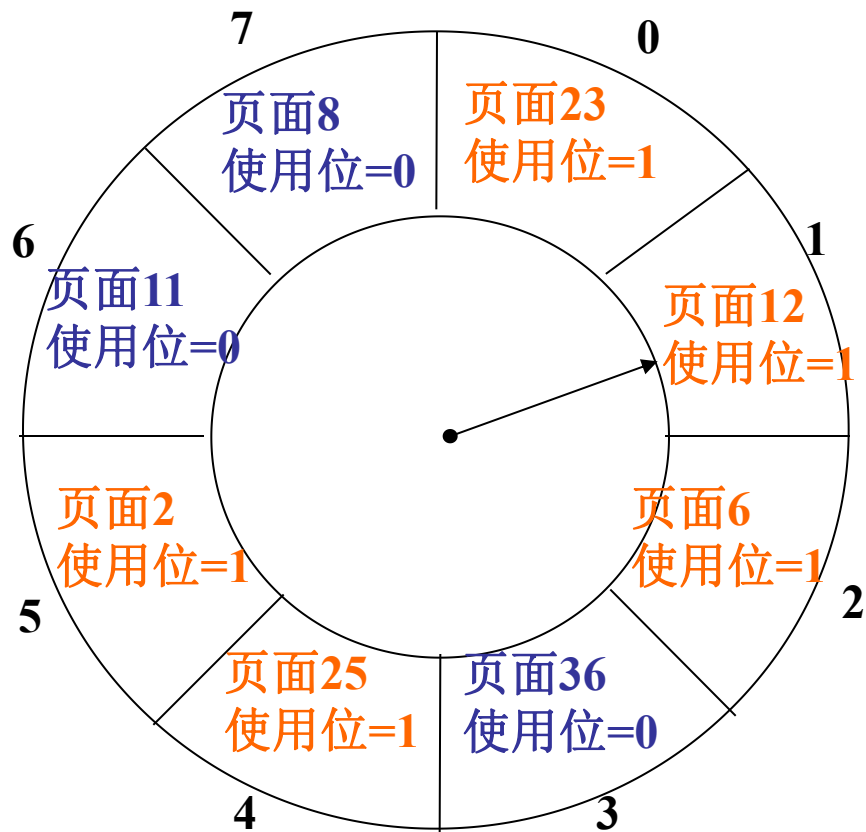
- 最佳置换算法
- 先进先出置换算法
- 最近最少使用置换算法
- 时钟置换算法
- \*其它算法



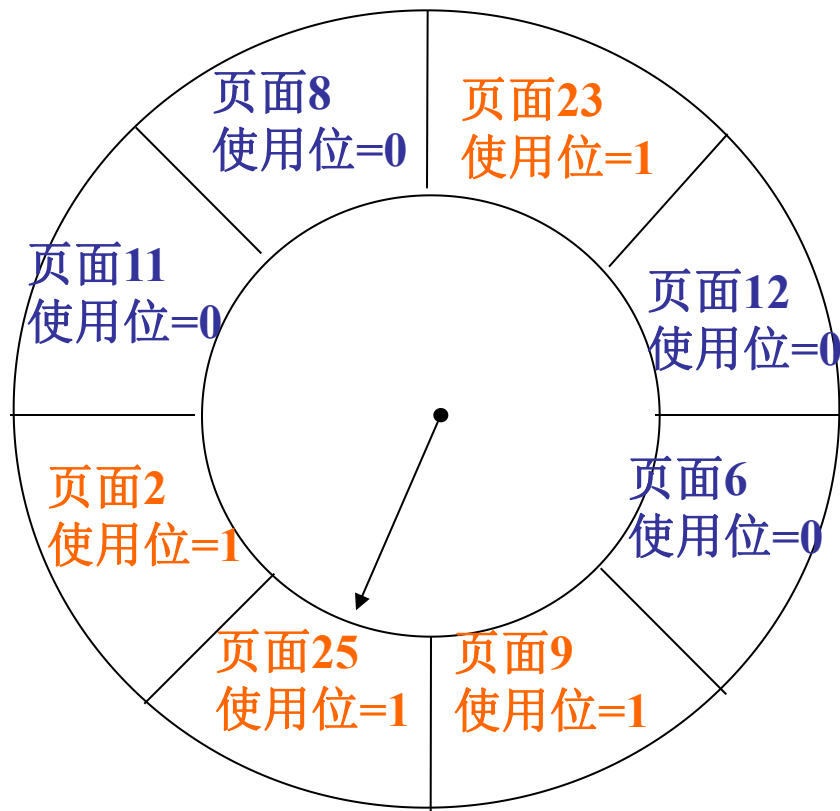
## 单击此处编辑母版标题样式

- 时钟算法需要给每个物理块增加一个附加位，称为使用位 $u$ 。当某一页装入内存，该物理块的使用位设为1；当该物理块被使用时，它的使用位也设为1。
- 对于页面置换算法，把用于替换的物理块集合看作是一个循环缓冲区，并且有一个指针与之关联。
- 当需要进行页面置换时，如果指针所在的页面 $u=0$ ，则将它置换，然后把指针指向下一个物理块。
- 否则，把该块的使用位置为0，然后跳过该块继续扫描，直到找到一个 $u=0$ 的物理块。


# 时钟算法



(a) 页面置换前状态



(b) 页面置换后状态



## 单击此处编辑母版标题样式

在Clock算法的基础上再增加一个附加

位修改位 $w$ 。具有2个附加位的物理块中的页具有4种情况：

- 最近未访问过，也未被修改过（ $u=0$ ， $w=0$ ）。
- 最近未访问过，但被修改过（ $u=0$ ， $w=1$ ）。
- 最近访问过，但没有被修改过（ $u=1$ ， $w=0$ ）。





## 改进Clock算法

改进型时钟算法如下：

- 从指针当前位置开始扫描，在这次扫描过程中对使用位的值不做任何修改，找到一个 $u=0$ ， $w=0$ 的物理块，进行置换。
- 如果第一步失败，则查找 $u=0$ ， $w=1$ 的块，遇到的第一个这样的物理块，则把该块中的页置换出去，同时把扫描过程中的遇到的 $u=1$ 的块设为 $u=0$ 。
- 如果前两步都失败，在重新执行第一步、第二步，这样一定会找到一个合适的页替换出去。



# 请求分页中的页面置换算法

---

- 最佳置换算法
- 先进先出置换算法
- 最近最少使用置换算法
- 时钟置换算法



# 置换算法比较

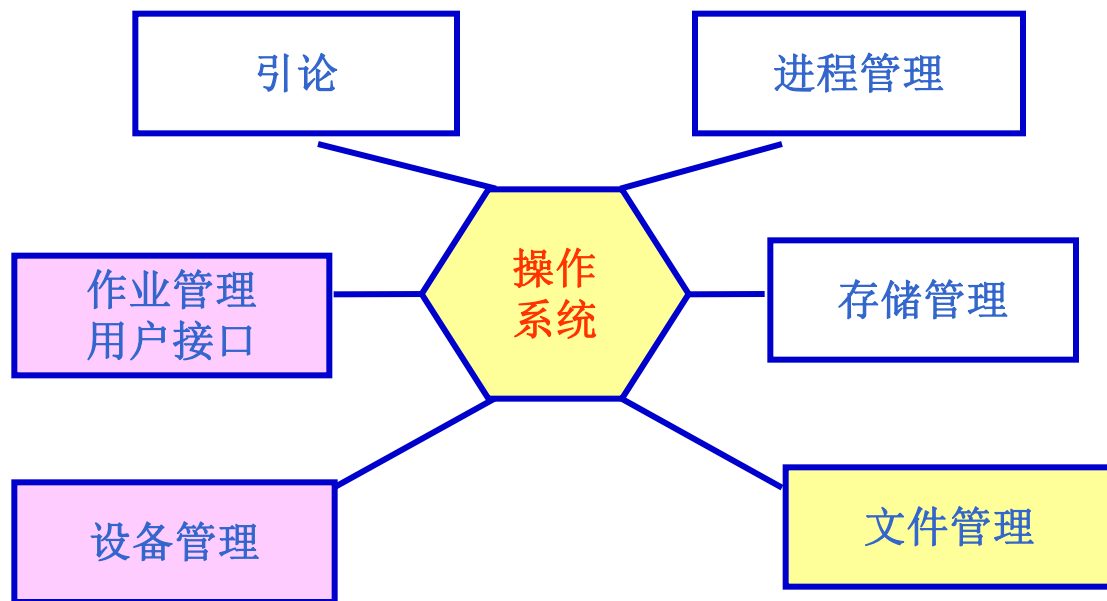
算法	注释
最优算法	不可实现，但可用作评价基准
<b>FIFO</b> （先进先出）算法	可能抛弃重要页面
<b>LRU</b> （最近最少使用）	很优秀，但很难实现
<b>NRU</b> （最近未使用）算法	<b>LRU</b> 的相对粗略的近似
老化算法	非常近似 <b>LRU</b> 的有效算法
时钟算法	现实的方法，速度较快



# 存储器管理主要内容

---

- ❖ 虚拟存储器的基本概念
- ❖ 请求分页存储管理方式
- ❖ 页面置换算法





# 主要知识点

- 文件系统概念
- 文件逻辑结构
- 外存分配方式
- 文件目录
- 文件路径



# 主要知识点

- 文件系统概念
- 文件逻辑结构
- 外存分配方式
- 文件目录
- 文件路径



# 主要知识点

- 文件系统是指 在操作系统中对文件进行管理的有关软件和数据集合，说白了，就是操作系统管理文件的方法。
- 文件系统主要有5个功能：
  - 1) 对文件进行按名存取
  - 2) 为用户提供统一和友好的接口
  - 3) 对文件和文件目录进行管理
  - 4) 对文件存储空间进行分配和管理
  - 5) 文件的共享与保护





# 主要知识点

- 文件系统概念
- 文件逻辑结构
- 外存分配方式
- 文件目录
- 文件路径



# 文件逻辑结构

**对文件的逻辑结构提出的基本要求：**

**提高检索速度；**

**便于修改；**

**降低文件存储费用**



# 主要内容

---

- 文件逻辑结构的类型
- 顺序文件
- 索引文件
- 索引顺序文件



# 一、文件逻辑结构的类型（1）

## 有结构的记录式文件

- ✓文件构成：由一个以上的记录构成。
- ✓记录长度：分为定长和变长。
- ✓分类（按记录的组织方式）：

**顺序文件**-定长（也可以是变长）记录按某种顺序排列形成

**索引文件**-针对变长记录

**索引顺序文件**-为文件建立一张索引表，为每组记录的第一个记录设置一个表项

# 一、文件逻辑结构的类型（2）

好处：更加方便、OS代码更加可靠、灵活，用户编程也更加方便

## 无结构的流式文件

- ✓ 文件构成：由字符流构成。大量的源程序、可执行文件、库函数等，所采用的就是无结构的文件形式，即流式文件。
- ✓ 长度：以字节为单位（通用计算机寻址的最小单位）
- ✓ 访问：对流式文件的访问，则是采用读写指针来指出下一个要访问的字符。
- ✓ 注：在UNIX系统中，所有的文件都被看作是流式文件；即使是有结构文件，也被视为流式文件；系统不对文件进行格式处理。
- ✓ 可以把流式文件看作是记录式文件的一个特例。



# 主要内容

---

- 文件逻辑结构的类型
- 顺序文件
- 索引文件



# 主要内容

---

- 顺序文件是最常用的文件组织形式。
- 顺序文件由一系列记录按照某种顺序排列形成。
- 其中的记录通常是定长记录（也可变长），因而能用较快的速度查找文件中的记录。

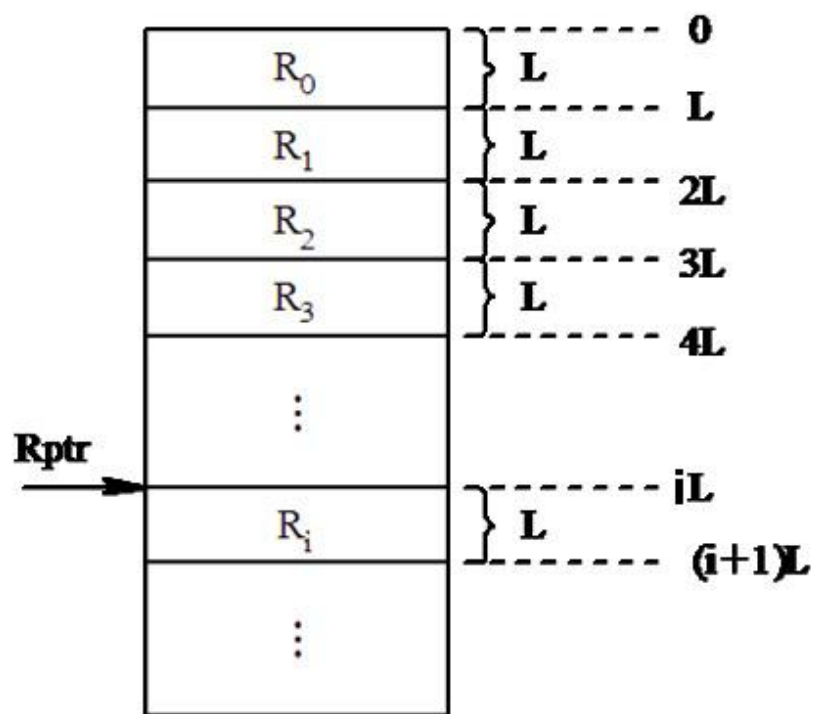
## 二、顺序文件（2）



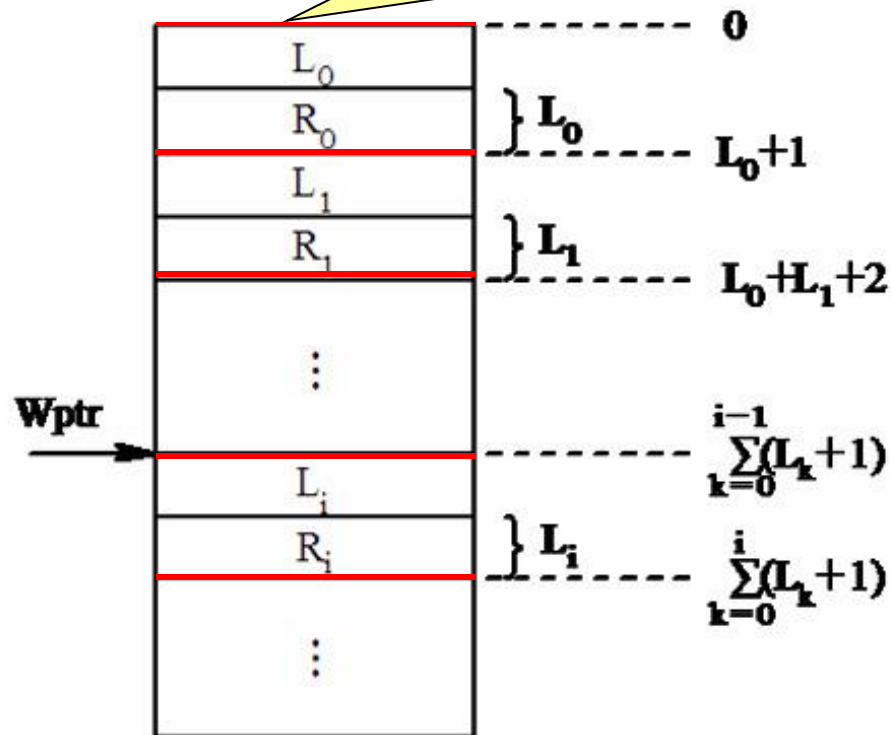
对顺序文件的读、写操作

- ✓ 记录为定长的顺序文件
- ✓ 记录为变长的顺序文件

其中  $L_i$  占用一个字节，用来指明记录  $i$  的长度



(a) 定长记录文件



(b) 变长记录文件

图 6-3 定长和变长记录文件



# 顺序文件的优缺点

文件的访问方式：顺序访问和随机访问（直接访问）

## 优点

- 顺序存取速度较快（批量存取）。
- 对定长记录，还可方便实现直接存取（随机存取）。
- 只有顺序文件才能存储在磁带上，并能有效地工作。

## 缺点

- ✓ 对变长记录，直接存取低效。
- ✓ 不利于文件的动态增长。



# 主要内容

---

- 文件逻辑结构的类型
- 顺序文件
- 索引文件

# 索引文件

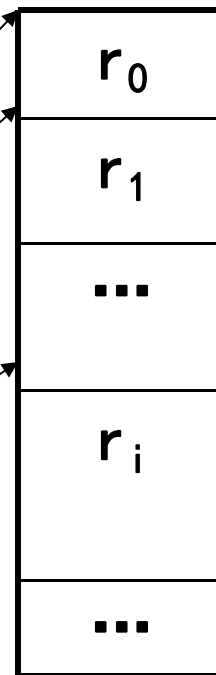
## 引入

为解决对变长记录文件难以进行直接存取的问题。

## 索引文件

为变长记录文件建立一张**索引表**。

索引号	长度 $m$	指针 ptr
0	$m_0$	
1	$m_1$	
...		
$i$	$m_i$	
...	索引表	



逻辑文件



# 索引文件的特点

## ■ 优点

- 通过索引表可方便地实现直接存取，具有较快的检索速度。
- 易于进行文件的增删。

## ■ 缺点

- 索引表的使用增加了存储费用；
- 索引表的查找策略对文件系统的效率影响很大。

■ 注：若索引表很大，可建立多级索引



# 主要知识点

- 文件系统概念
- 文件逻辑结构
- 外存分配方式
- 文件目录
- 文件路径



## 外存分配方式

- 文件的物理结构即文件的外存分配方式，是从系统的角度来看文件，从文件在物理介质上的存放方式来研究文件。
- 要考虑的主要问题：

如何有效地利用外存空间

如何提高对文件的访问速度



# 主要分配方式

- 连续分配（顺序分配）
- 链接分配
- 索引分配



# 主要分配方式

- 连续分配（顺序分配）
- 链接分配
- 索引分配

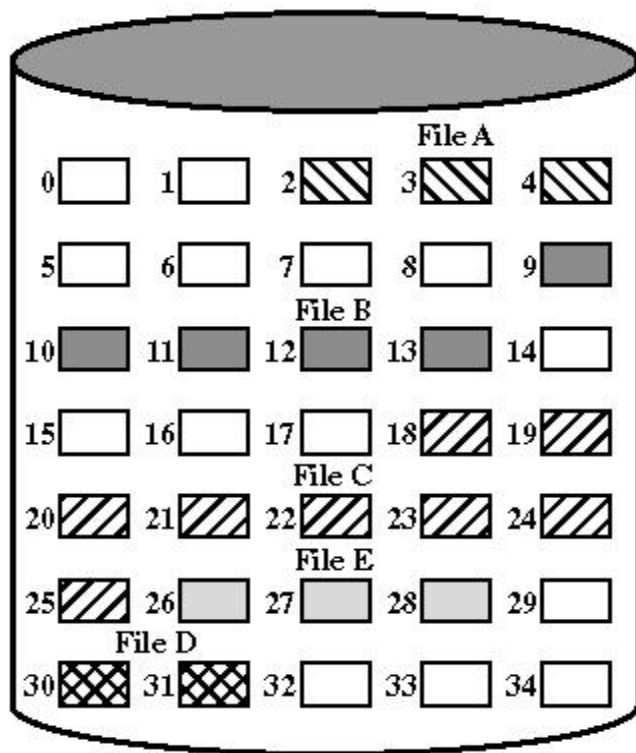




## 连续/顺序分配

- 每一个文件占用一个连续的磁盘块的集合, 从而成为一种物理上的**顺序文件**
- 即把逻辑文件中的记录**顺序地存储到**邻接的各物理块。邻接的物理块一般在同一条磁道上, 所以**不必移动磁头**
- 这样形成的文件结构称为**顺序文件结构**, 此时的物理文件称为**顺序文件**
- 在目录项的“文件物理地址”字段中记录该文件的**第一个记录所在盘块号和文件长度**

# 连续/顺序分配



File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

磁盘空间的连续分配



## 连续/顺序分配优缺点

- 优点：

- 顺序访问容易、速度快；
- 可以随机存取(Random access)

- 缺点：

- 要求连续的存储空间。浪费空间：几次动态存储分配后就出现零头问题；  
紧凑又要花费大量机器时间
- 必须事先知道文件长度，不能动态增长，不利于插入删除

- 适用：

- 简单应用环境，已知文件数量和大小
- CD-ROM, DVD



# 主要分配方式

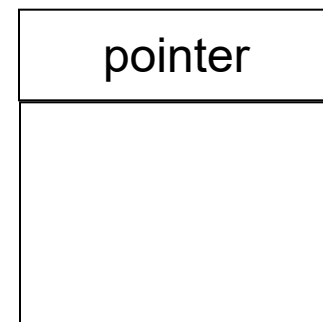
---

- 连续分配（顺序分配）
- 链接分配
- 索引分配

## (2) 外存分配方式-链接分配 (1)

- 将一个逻辑文件存储到外存时将文件装到多个**离散**的盘块中，通过每个盘块上的**指针**将同属于一个文件的盘块链成一个链表。这样形成的物理文件称为**链接文件**
- 每个文件是一个磁盘块的链接列表：块可以分散在磁盘各处
- 优点：采用离散分配消除了外部碎片，显著提高了外存利用率；当文件动态增长时，动态分配盘块，不需要预先知道文件的大小；对文件增删改方便。
- 有两种形式：**隐式链接和显式链接**

Block =

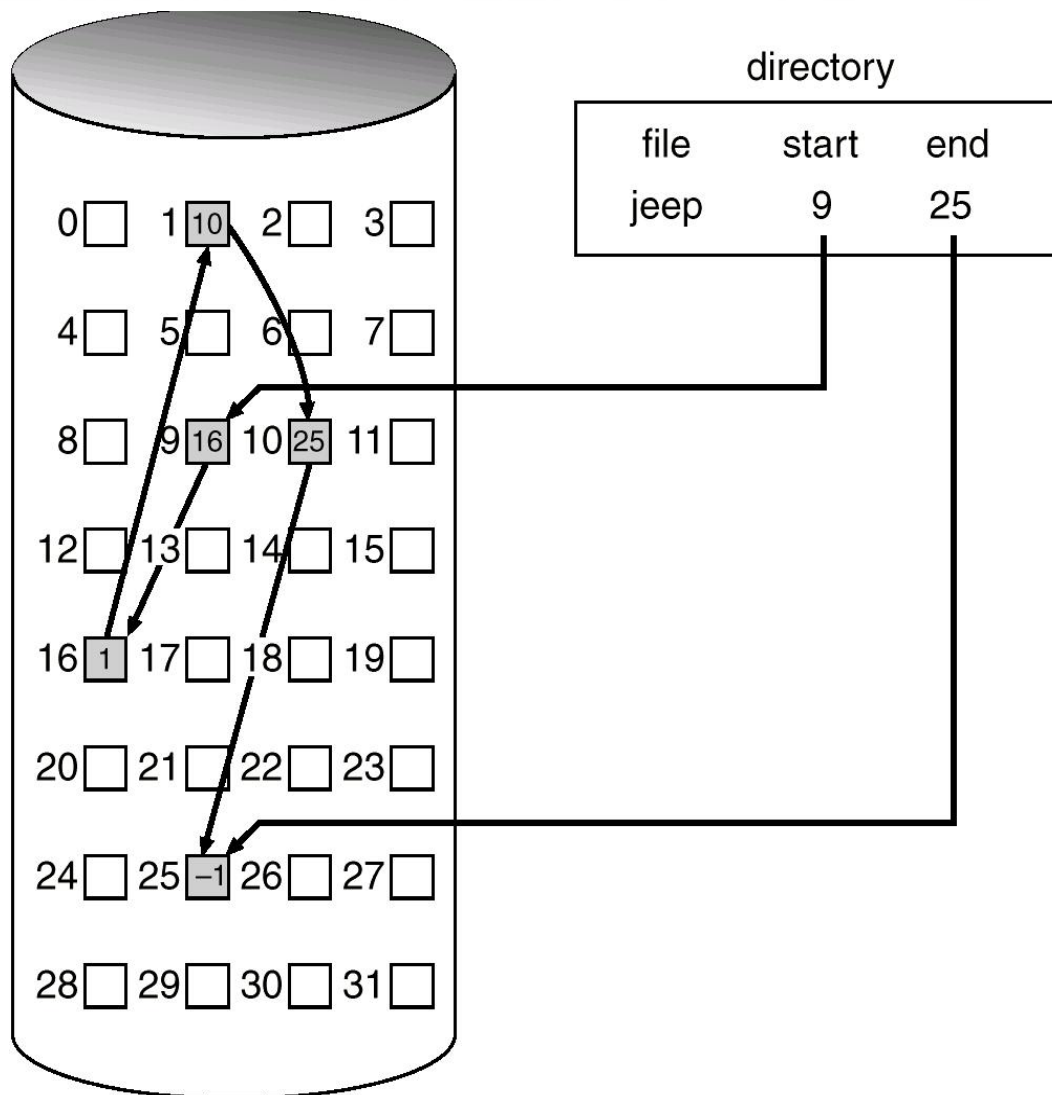




## 隐式链接

- 在文件目录的每一个目录项中含有指向链接文件第一个盘块和最后盘块的指针，每个盘块内含有指向下一盘块的指针。

# 隐式链接





# 隐式链接

---

- 隐式链接的缺点：
  - 只适合顺序访问，对随机访问低效
  - 可靠性差，只要盘块指针故障，便使整个链断开
  - 每个盘块设一指针，占用较大的空间





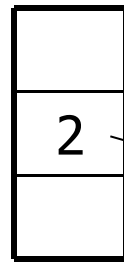
# 显式链接

- 把用于链接文件各物理块的指针，显式地存放在**内存**的一张链接表中。该表在整个磁盘仅设置一张
- 表的序号是物理块号，从0开始直到N-1，N为盘块总数。在每个表项中存放**链接指针，即下一个盘块号**
- 由于分配给文件的所有盘块号都放在链接表中，故把链接表称为**文件分配表FAT (File Allocation Table)**
- 在链接表中，每条链的链首指针对应的块号（属于某一个文件的第一个盘块号）作为**文件地址**填入相应文件的FCB (File Control Block, 文件控制块) 的“物理地址”字段中。

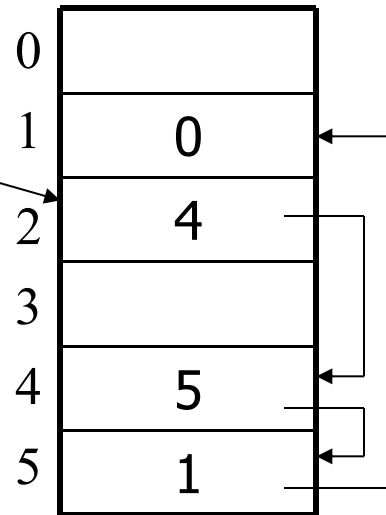
# 显式链接

FCB(File Control Block)

文件控制块



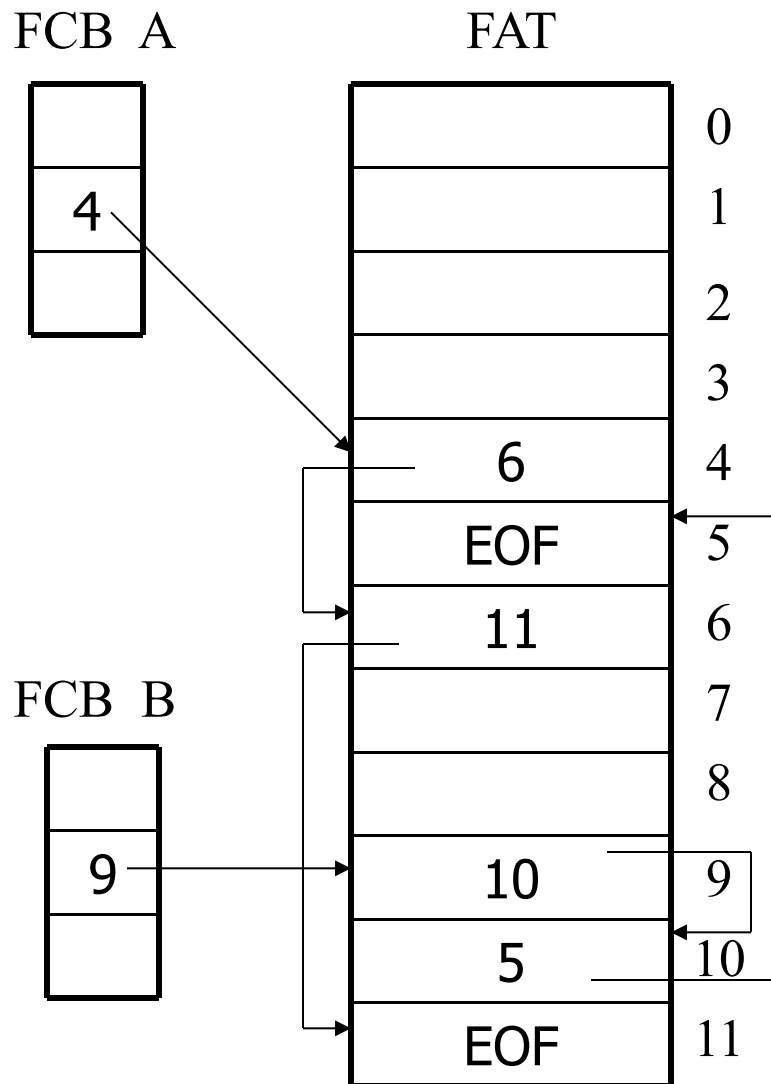
FAT



物理块号

显式链接结构

MS-DOS的文件物理结构。实例是两个文件：文件A占用三个盘块，文件B也占用三个盘块。





## 显式链接的优缺点

- **优点：**除了前述链接分配的一般优点外，还由于FAT表在**内存**，大大提高了检索速率，减少访问磁盘次数
- **缺点：**FAT表中每个盘块一项，占用大量内存



# 链接分配的优缺点

## ■ 优点

- 1、无外部碎片，没有磁盘空间浪费
- 2、无需事先知道文件大小。文件动态增长时，可动态分配空闲盘块。对文件的增、删、改十分方便。

## ■ 缺点

- 1、不能支持高效随机/直接访问，仅适合于顺序存取
- 2、需为指针分配空间。（隐式链接）
- 3、可靠性较低（指针丢失/损坏）
- 4、文件分配表FAT（显式链接）

FAT需占用较大的内存空间（如何计算FAT表所占的内存空间：）。



# 主要分配方式

- 连续分配（顺序分配）
- 链接分配
- 索引分配



### (3) 外存分配方式-索引分配 (1)

- 链接分配方式虽然解决了连续分配方式所存在的问题，但又出现了另外两个问题， 即：
  - (1) 不能支持高效的直接存取。要对一个较大的文件进行直接存取，必须首先在FAT中顺序地查找许多盘块号。
  - (2) FAT需占用较大的内存空间。
- 为每一个文件分配一个索引块（表），再把分配给该文件的所有块号，都记录在该索引块中。故索引块就是一个含有许多块号地址的数组。
- 该索引块的地址由该文件的目录项指出。



# 单级索引分配

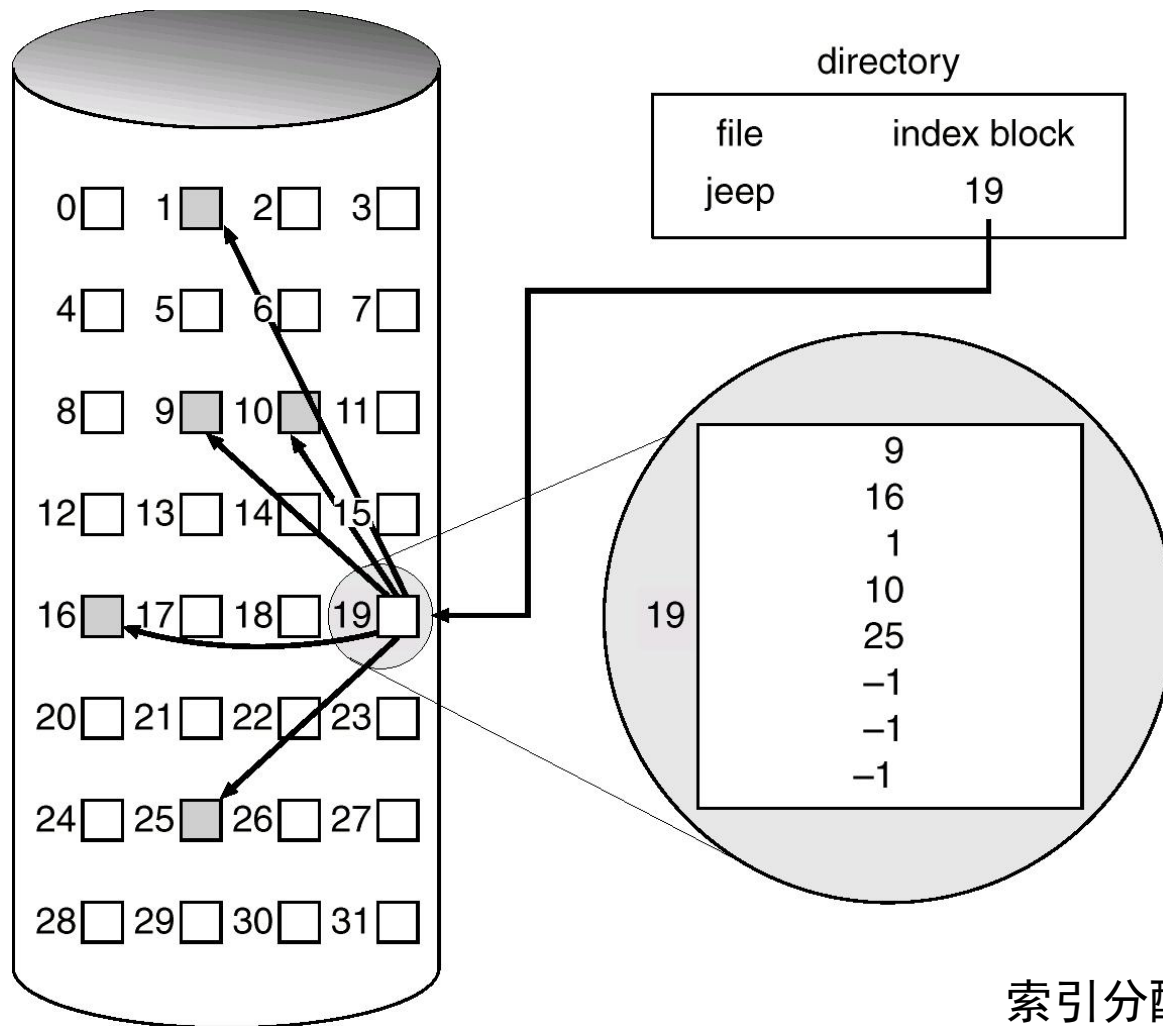
---

- 当打开一文件时不需要把整个FAT表调入内存，**只要把该文件占用的盘块的编号调入内存**，为此：
  - 为每一个文件建立一个索引块(表)，存放分配给该文件的所有盘块号，通常用一**专门的盘块作为索引表**。  
因此，索引块(表)就是一个含有许多盘块号的数组
  - 建立文件时，在文件目录的表项中填上指向该索引块的指针



# 单级索引分配

## 1. 单级索引分配





## 单级索引分配特点

---

- 需要索引块，可能要花费较多的外存（一个盘块可放成百甚至上千个盘块号）
- 可**随机存取**（直接访问）
- 可以动态分配而无外部碎片
- 对于小文件采用索引分配方式，其索引块的利用率极低



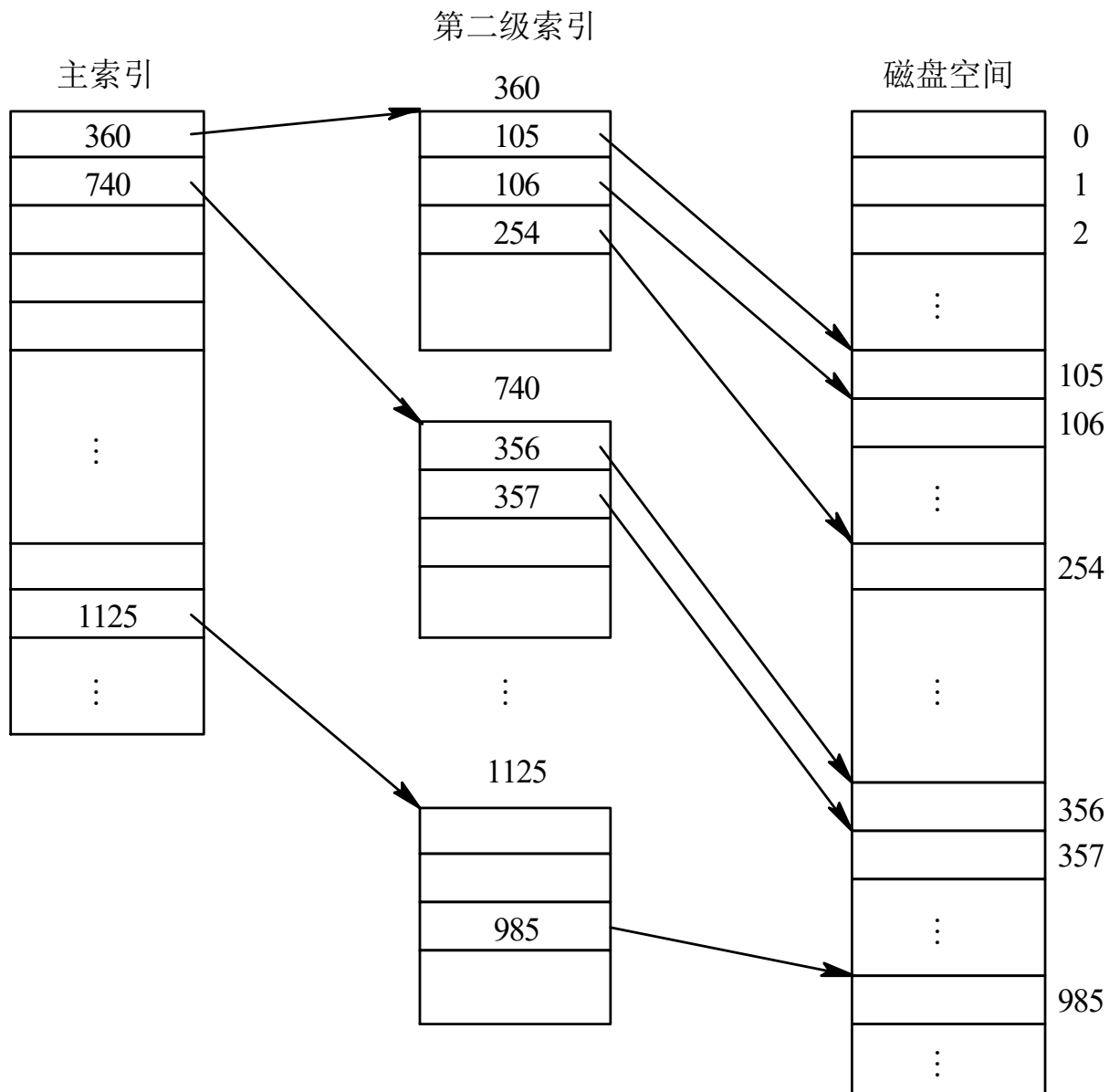
# 多级索引分配

---

- 对于大型文件，一个索引块可能容纳不下所有的盘块号，则可再分配一个索引块继续装入盘块号，....，直到装完所有的盘块号
- 通过链指针将各索引块链接起来
- 若文件太大，索引块很多，效率很低
- 为索引块再分配一个索引块，装填索引块的盘块号。于是形成二级索引分配方式

# 多级索引分配

## 两级索引分配





## 多级索引分配

---

假设每个盘块大小为1KB，每个盘块号占4个字节，则在一个索引块中可放256个文件物理块的盘块号。在两级索引时，最多可包含的存放文件的盘块的盘块号总数为 $256 * 256 = 64K$ 个盘块号。可以得出：采用两级索引时，所允许的文件最大长度为64MB。



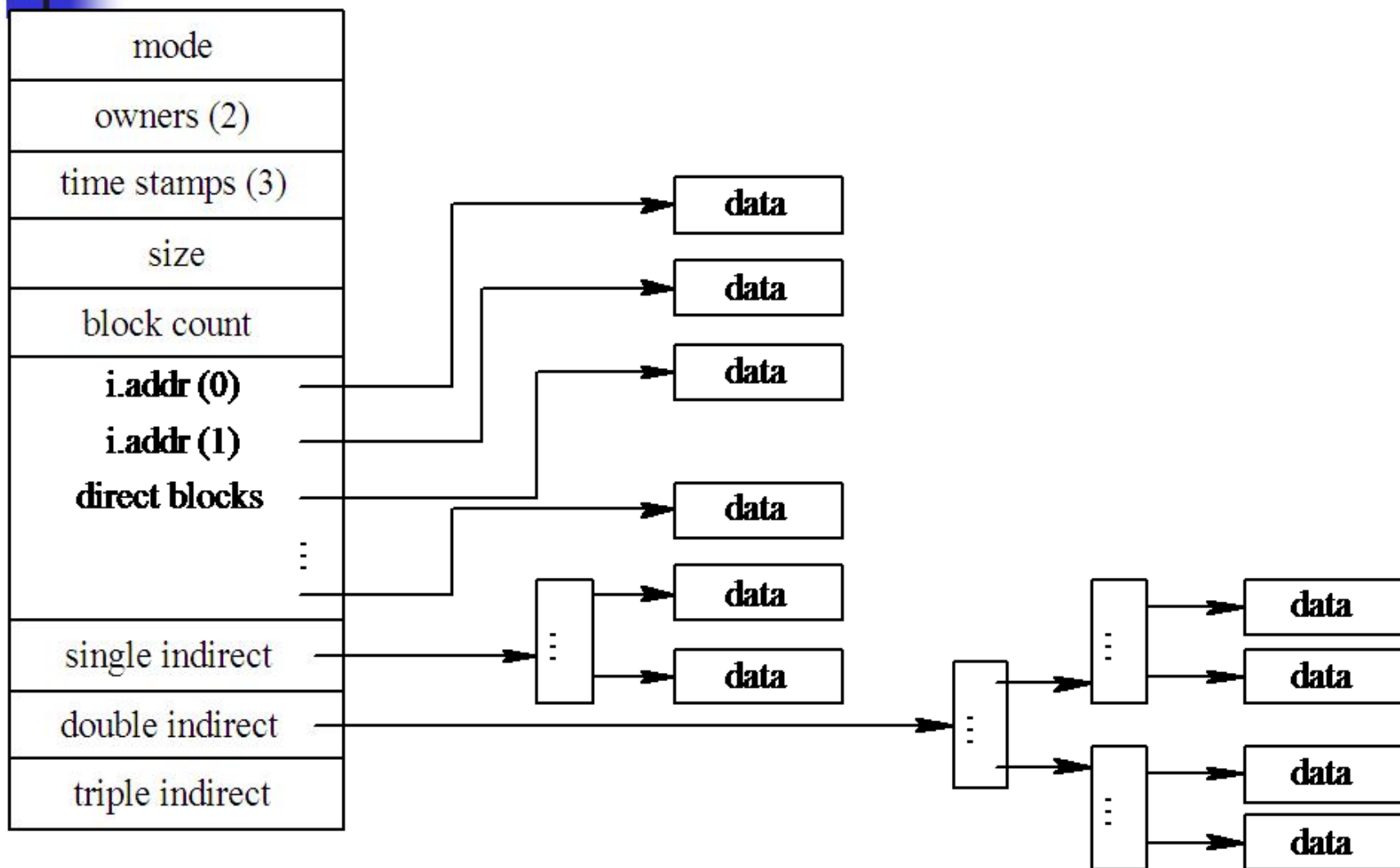
# 混合索引

## 3. 混合索引方式

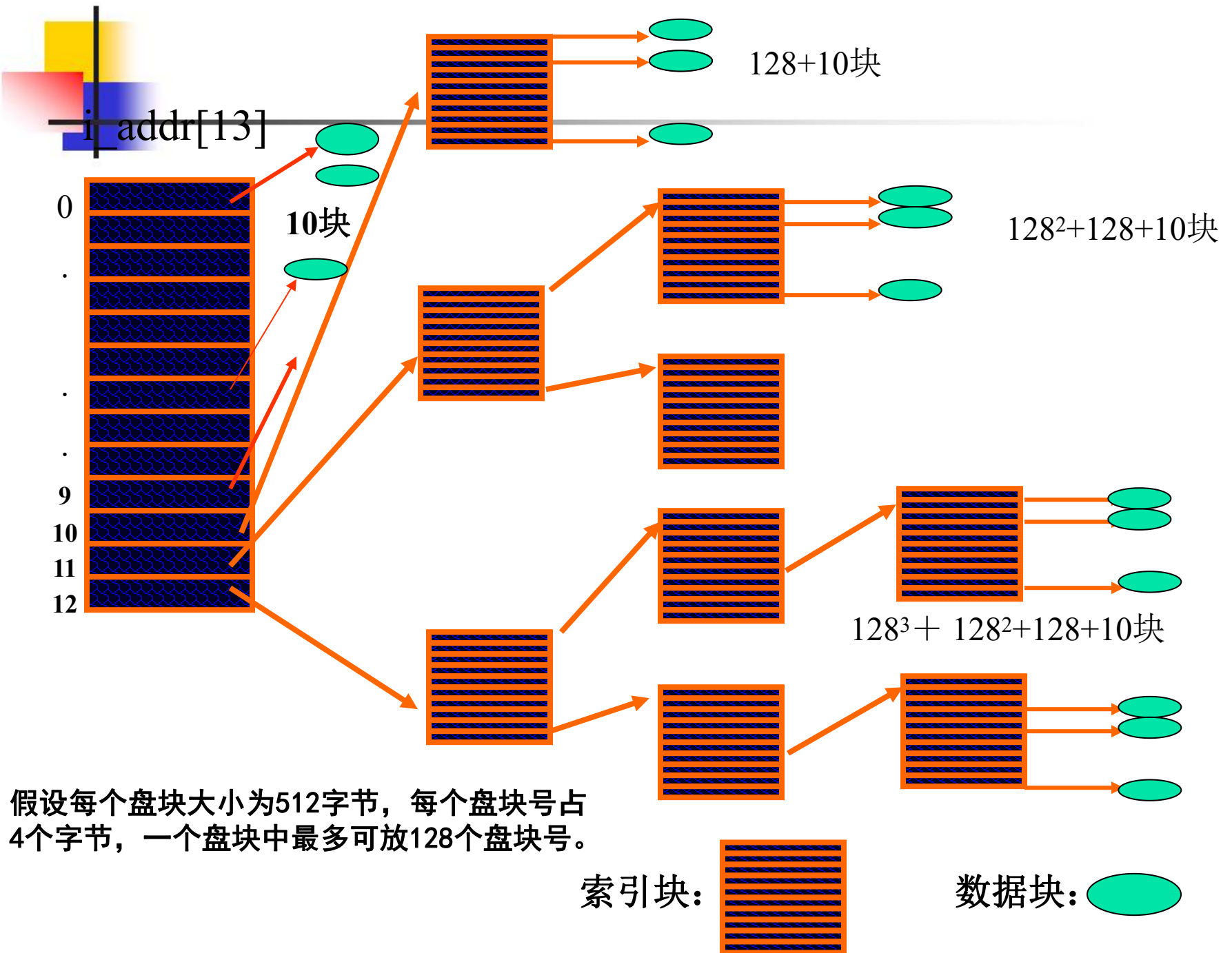
UNIX文件系统采用的是混合索引结构(综合模式)。每个文件的索引表为13个索引项。最前面10项直接登记存放文件信息的物理块号(直接寻址)

如果文件大于10块,则利用第11项指向一个物理块,假设每个盘块大小为512字节,每个盘块号占4个字节,该块中最多可放128个盘块号(一次间接寻址)。对于更大的文件还可利用第12和第13项作为二次和三次间接寻址。

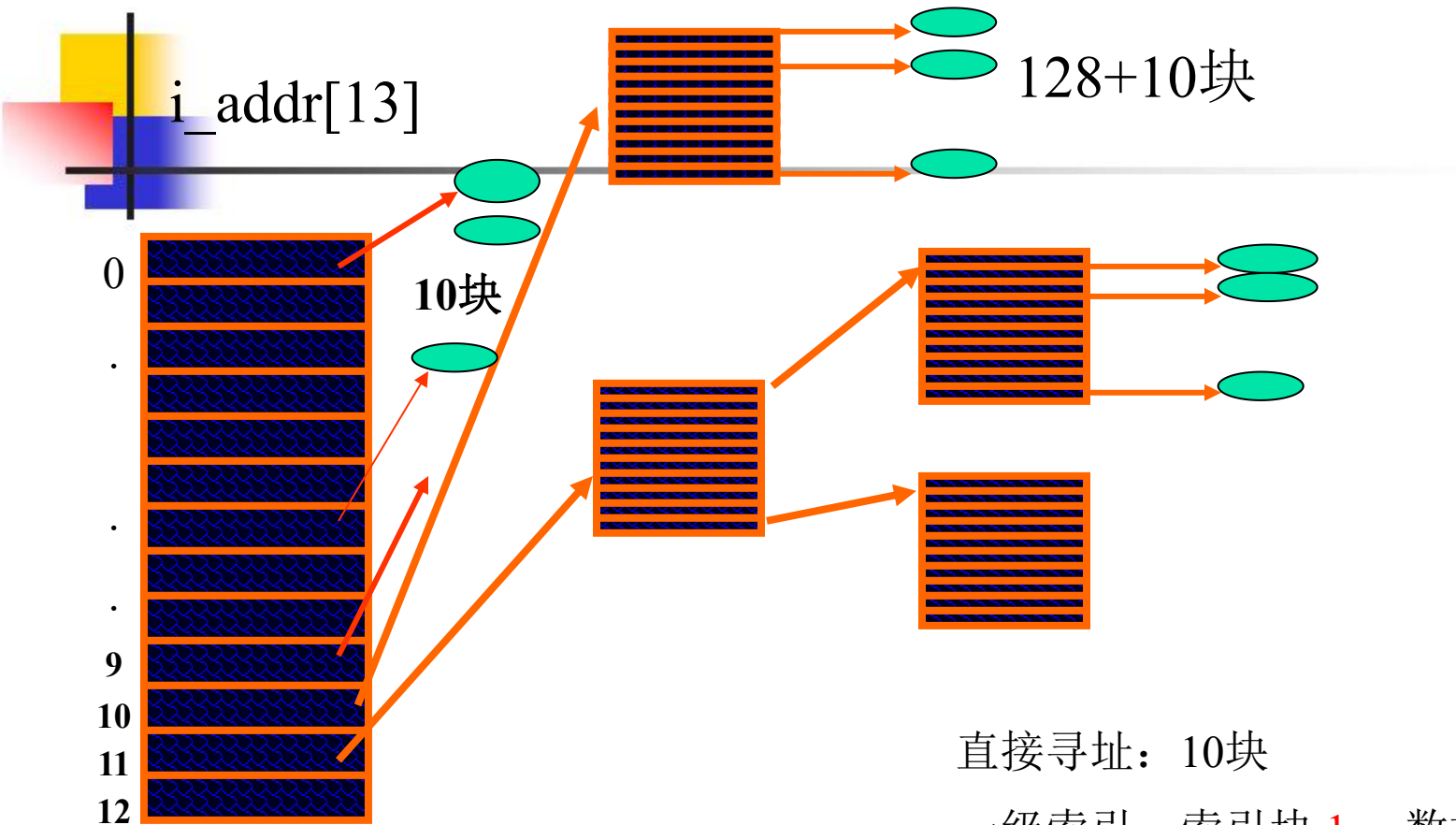
# 混合索引



混合索引方式







直接寻址：10块

一级索引：索引块 1，数据块128

二级索引：1级索引块 1，二级索引块  
 $(5000-10-128) / 128 = 37.98 \approx 38$

共占用磁盘块：索引块+数据库=5000+1+1+38=5040

大小为5000块的文件需占多少个磁盘块？



# 主要知识点

- 文件系统概念
- 文件逻辑结构
- 外存分配方式
- 文件目录
- 文件路径



## 文件目录

---

- 文件目录就是一个数据结构，在这个数据结构中，记录了文件与其对应的物理地址。
- 文件目录与文件的关系就相当于文件与其内部物理记录的关系，说白了，我们是对物理记录进行读写的，但在读写前，我们要通过文件目录到文件的映射**A**找到需要的文件，然后再通过文件到物理记录的映射**B**找到要操作的物理记录。



# 目录管理要求

## 对文件目录的管理要求

- 实现“按名存取”
- 提高对目录的检索速度
- 文件共享
- 允许文件重名



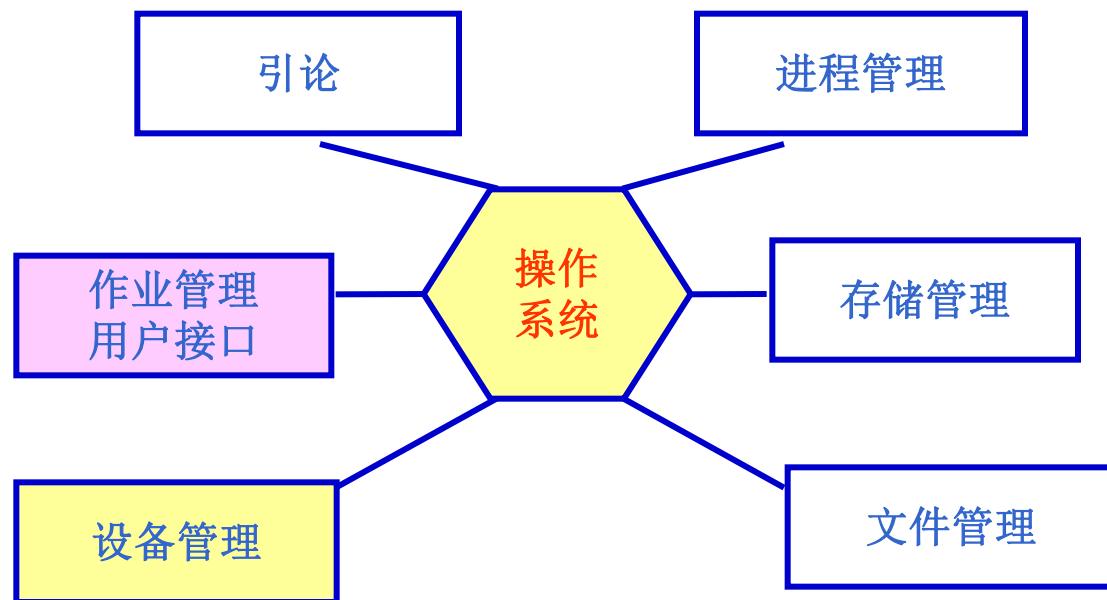
# 主要知识点

- 文件系统概念
- 文件逻辑结构
- 外存分配方式
- 文件目录
- **文件路径**



# 主要知识点

- 在树型目录中，同一目录中的各个文件不能同名，但不同目录中的文件可以同名。
- 文件路径名有两种表示形式：绝对路径名和相对路径名。
  - 1) 绝对路径名（全路径名）：是从根目录开始到达所要查找文件的路径。
  - 2) 相对路径名：系统为每个用户设置一个当前目录（又称工作目录），访问某个文件时，就从当前目录开始向下顺次检索。





# 主要知识点


- I/O 通道
- I/O 控制方式
- I/O设备的类型
- 缓冲技术
- SPPOOLING技术
- 磁盘调度算法





# 主要知识点

- I/O 通道
- I/O 控制方式
- I/O设备的类型
- 缓冲技术
- SPPOOLING技术
- 磁盘调度算法



## I/O 通道

---

通道：一种特殊的执行I/O指令的**处理机**，指令类型单一，没有自己的内存，与CPU共享内存。

引入目的：建立独立的I/O操作，解脱CPU对I/O的组织、管理。

CPU只需发送I/O命令给通道，通道通过调用内存中的相应通道程序完成任务。



# 主要知识点

- I/O 通道
- I/O 控制方式
- I/O设备的类型
- 缓冲技术
- SPOOLING技术
- 磁盘调度算法



# I/O 控制方式

- 查询控制方式
- 中断控制方式
- **DMA**控制方式
- 通道控制方式



# I/O 控制方式

- 查询控制方式
- 中断控制方式
- DMA控制方式
- 通道控制方式



# 1、查询控制方式

- 又称条件传送方式，CPU需要先了解（查询）外设的工作状态，然后在外设可以交换信息的情况下（就绪）实现数据输入或输出
- 对多个外设的情况，则CPU按一定顺序依次查询（轮询）。先查询的外设将优先进行数据交换
- 查询传送的特点是：工作可靠，适用面宽，但传送效率低



# 查询控制方式缺点

---

## 其主要缺点：

CPU必须循环等待，以检测外设状态，直到外设准备就绪以后才能传送数据。这样，为了传送一个数据，软件开销很大，CPU把绝大部分时间都花在循环等待上，而真正为外设服务的时间却很少，使CPU的使用效率低。

## 处理方法：

要想提高CPU使用效率，可在循环等待期间穿插一些其它运算处理，但这样势必影响I/O服务的响应速度，使I/O处理的实时性降低。



# I/O 控制方式

- 查询控制方式
- 中断控制方式
- DMA控制方式
- 通道控制方式





# 中断控制方式

---

## 中断控制方式的概念:


CPU不再被动循环查询，而是可以执行其它程序。一旦外设准备就绪**由外设主动**向CPU提出中断服务请求，CPU如果响应请求，便暂时停止当前程序的执行，转去执行与该请求相应的服务程序，服务程序执行完毕后再继续执行原来被中断的程序。



## 中断控制方式优点

---

不仅省去了**CPU**查询外设状态和等待外设准备就绪所花费的时间，提高了**CPU**的工作效率，而且还满足了外设的实时性要求。



外设主动



# 中断控制方式缺点

- 系统需要为每个I/O设备分配一个中断请求号和编写相应的中断服务程序，此外**还需要一个中断控制器**来管理I/O设备提出的中断请求。
- 每次数据传送都要进行一次中断，在中断服务程序中还需保留和恢复现场以便能继续原程序的执行，工作量较大。**如果需要大量数据交换，系统的效率较低。**

硬件复杂



# I/O 控制方式

- 查询控制方式
- 中断控制方式
- **DMA控制方式**
- 通道控制方式



# DMA的基本概念

- 采用程序控制方式以及中断方式进行数据传送时，都是靠CPU执行程序指令来实现数据的输入/输出的。
- 采用程序控制方式及中断方式时，数据的传输率不会很高。
- 对于高速外设，如高速磁盘装置或高速数据采集系统等，采用这样的传送方式，往往满足不了其数据传输率的要求。
- 例如，对于磁盘装置，其数据传输率通常在20万字节/秒以上，即传输一个字节的时间要小于 $5\mu\text{s}$ 。



# DMA的基本概念

---

- 对于通常的PC机来说，执行一条程序指令平均需要几 $\mu\text{s}$ 时间。显然，采用程序控制或中断方式不能满足这种高速外设的要求。
- 由此产生不需要CPU干预(不需CPU执行程序指令)，而在专门硬件控制电路控制之下进行的外设与存储器间直接数据传送的方式，称为直接存储器存取(Direct Memory Access)，简称DMA方式。
- 这一专门的硬件控制电路称为DMA控制器，简称DMAC。



# DMA特点

优点— CPU利用率进一步提高（并行度有所提高）。

缺点—数据传送方向、字节数、内存地址等需由CPU控制，且每一设备需一台DMA控制器，设备增多时，不经济。



# I/O 控制方式

- 查询控制方式
- 中断控制方式
- DMA控制方式
- 通道控制方式





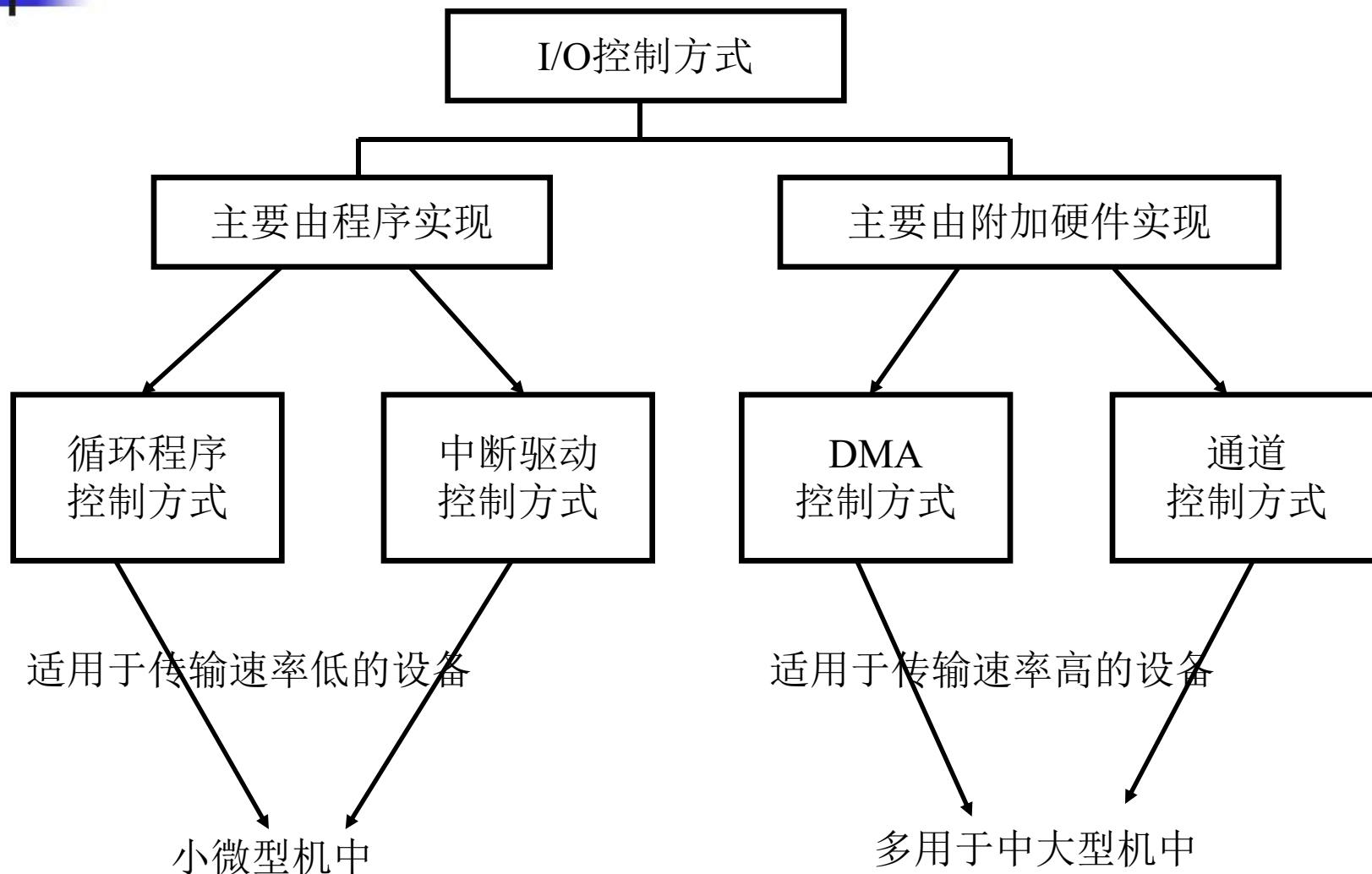
## 通道控制方式

---

为了获得**CPU**和外围设备间更高的并行工作能力，也为了让种类繁多，物理特性不同的外围设备能以标准的接口连接到系统中，计算机系统引入了自成独立体系的通道结构

由通道管理和控制**I/O**操作，减少了外围设备和**CPU**的逻辑联系，把**CPU**从琐碎的**I/O**操作中解放出来。

# 总结：I/O控制方式比较





# 主要知识点

- I/O 通道
- I/O 控制方式
- I/O设备的类型
- 缓冲技术
- SPPOOLING技术
- 磁盘调度算法



## 1) 按使用方式/共享属性分类

- 独享设备
- 共享设备
- 虚拟设备

**独享/独占设备：**在一段时间只允许一个用户进程访问的设备。多数低速设备属此类，打印机就是典型的独享设备。

**共享设备：**在一段时间允许多个用户进程同时访问的设备。磁盘就是典型的共享设备。

**虚拟设备：**指通过虚拟技术将一台独占设备变换为若干台逻辑设备，供若干个用户进程同时使用，通常把这种经过虚拟技术处理后的设备称为虚拟设备。



## 2) 按传输速率分类

- 低速设备
- 中速设备
- 高速设备

**低速设备**：传输速率仅为每秒钟几个字节至数百个字节的设备。典型的有：键盘、鼠标、语音的输入/输出等。

**中速设备**：传输速率为每秒钟数千个字节至数万个字节的设备。典型的有：打印机等。

**高速设备**：传输速率为每秒钟数百KB至数十MB的设备。典型的有：磁盘机、磁带机、光盘机等。



### 3) 按信息交换的单位分类

块设备

字符设备

**块设备**：信息交换的基本单位为**字符块**，属于有结构设备，块大小一般为512B---4KB，典型的有：磁盘、磁带等。块设备的**基本特征**：传输速率较高（几MB/s）、可寻址、I/O常采用DMA方式。

**字符设备**：信息交换的基本单位为**字符**，典型的有：键盘、打印机和显示器等。**基本特征**：传输速率较低，几字节~数千B/s、不可寻址、I/O常采用中断驱动方式。



# 主要知识点

- I/O 通道
- I/O 控制方式
- I/O设备的类型
- 缓冲技术
- SPPOOLING技术
- 磁盘调度算法



## 引入缓冲技术的目的

- 缓和CPU与I/O设备间速度不匹配的矛盾：
  - 比如，CPU的运算速率远远高于打印机I/O的速率，如果没有缓冲区，则在打印机输出数据时，必然会由于打印机的速度跟不上而使CPU停下来等待。
- 减少对CPU的中断频率，放宽对CPU中断响应时间的限制
  - 如果I/O操作每传送一个字节就要产生一次中断，那么设置了n个字节的缓冲区后，则可以等到缓冲区满才产生中断，这样中断次数就减少到 $1/n$ ，而且中断响应的时间也可以相应的放宽。
- 提高CPU和I/O设备之间的并行性
  - 例如，在CPU和打印机之间设置了缓冲区后，便可使CPU与打印机并行工作。





# 缓冲技术

- 单缓冲技术
- 双缓冲技术
- 循环缓冲技术
- 缓冲池技术



# 主要知识点

- I/O 通道
- I/O 控制方式
- I/O设备的类型
- 缓冲技术
- **SPOOLING技术**
- 磁盘调度算法



# 主要知识点

- SP00Ling技术是低速输入输出设备与主机交换的一种技术，通常也称为“假脱机真联机”，他的核心思想是以联机的方式得到脱机的效果。
- 低速设备经通道和外设在主机内存的缓冲存储器与高速设备相联，该高速设备通常是辅存。为了存放从低速设备上输入的信息，或者存放将要输出到低速设备上的信息（来自内存），在辅存分别开辟一固定区域，叫“输出井”（对输出），或者“输入井”（对输入）。
- 简单来说就是在内存中形成缓冲区，在高级设备形成输出井和输入井，传递的时候，从低速设备传入缓冲区，再传到高速设备的输入井，再从高速设备的输出井，传到缓冲区，再传到低速设备。



# 共享打印机

- 打印机是经常要用到的输出设备，属于独占设备。利用SPOOLing技术，可将之改造为一台可供多个用户共享的设备，从而提高设备的利用率，也方便了用户。
- 共享打印机技术已被广泛地用于多用户系统和局域网络中。当用户进程请求打印输出时，SPOOLing系统同意为它打印输出，但并不真正立即把打印机分配给该用户进程，而只为它做两件事：
  - ① 由输出进程在输出井中为之申请一个空闲磁盘块区，并将要打印的数据送入其中；
  - ② 输出进程再为用户进程申请一张空白的用户请求打印表，并将用户的打印要求填入其中，再将该表挂到请求打印队列上。
- 如果还有进程要求打印输出，系统仍可接受该请求，也同样为该进程做上述两件事。



# 主要知识点

- I/O 通道
- I/O 控制方式
- I/O设备的类型
- 缓冲技术
- SPPOOLING技术
- 磁盘调度算法



# 磁盘调度算法

- 磁盘调度算法
  - 早期的磁盘调度算法
    - 先来先服务FCFS
    - 最短寻道时间优先SSTF
  - 扫描算法
    - 扫描(SCAN) 算法
    - 循环扫描(CSCAN) 算法
    - N-STEP-SCAN调度算法
    - FSCAN调度算法



# 磁盘调度算法

---

- 先来先服务FCFS
- 最短寻道时间优先SSTF
- 扫描(SCAN) 算法
- 循环扫描(CSCAN) 算法
- N-STEP-SCAN调度算法
- FSCAN调度算法

## FCFS 先来先服务

例：假设一个请求序列：

55, 58, 39, 18, 90, 160, 150, 38, 184 磁头当前的位置在100。

按进程请求访问  
磁盘的先后次序  
进行调度。

**特点：**简单、较合理，但未对寻道进行优化。

FCFS算法（从100#磁道开始）

被访问的下一个磁道号	移动距离 (磁道数)
55	45
58	3
39	19
18	21
90	72
160	70
150	10
38	112
184	146
平均寻道长度：55.3	





# 磁盘调度算法

---

- 先来先服务FCFS
- 最短寻道时间优先SSTF
- 扫描(SCAN) 算法
- 循环扫描(CSCAN) 算法
- N-STEP-SCAN调度算法
- FSCAN调度算法

## 最短寻道时间优先 (SSTF-Shortest Seek Time First)

例：假设一个请求序列：

55, 58, 39, 18, 90, 160, 150, 38,  
184 磁头当前的位置在100。

选择从当前磁头位置所需寻道时间最短的请求。

**优点：**寻道性能比FCFS好，但不能保证平均寻道时间最短

**缺点：**有可能引起某些请求的饥饿。

### SSTF算法（从100#磁道开始）

被访问的下一个磁道号	移动距离 (磁道数)
90	10
58	32
55	3
39	16
38	1
18	20
150	132
160	10
184	24
平均寻道长度：27.5	



## 扫描算法 (SCAN) (1)

### 1) 进程“饥饿”现象

- SSTF算法虽然能获得较好的寻道性能，但却可能导致某个进程发生“饥饿”(Starvation)现象。
- 因为只要不断有新进程的请求到达，且其所要访问的磁道与磁头当前所在磁道的距离较近，这种新进程的I/O请求必须优先满足。
- 对SSTF算法略加修改后所形成的SCAN算法，即可防止老进程出现“饥饿”现象。



# 磁盘调度算法

---

- 先来先服务FCFS
- 最短寻道时间优先SSTF
- 扫描(SCAN) 算法
- 循环扫描(CSCAN) 算法
- N-STEP-SCAN调度算法
- FSCAN调度算法

## 扫描算法 (SCAN) (2)

例：假设一个请求序列：

55, 58, 39, 18, 90, 160, 150, 38,  
184 磁头当前的位置在100。

- 磁头从磁盘的一端开始向另一端移动，沿途响应访问请求，直到到达了磁盘的另一端，此时磁头反向移动并继续响应服务请求。有时也称为电梯算法。

**优点：**寻道性能较好，避免了饥饿。

**缺点：**不利于远离磁头一端的访问请求。

(从100#磁道开始，向  
磁道号增加的方向)

被访问的下一个磁道号	移动距离 (磁道数)
150	50
160	10
184	24
90	94
58	32
55	3
39	16
38	1
18	20

平均寻道长度：27.8

SCAN调度算法示例

# 循环扫描算法 (CSCAN)

例：假设一个请求序列：

55, 58, 39, 18, 90, 160, 150, 38,  
184 磁头当前的位置在100。

**规定磁头单向移动**

**特点：消除了对两端  
磁道请求的不公平。**

(从100#磁道开始，向  
磁道号增加的方向)

被访问的下一个磁道号	移动距离 (磁道数)
150	50
160	10
184	24
18	166
38	20
39	1
55	16
58	3
90	32
平均寻道长度： 35.8	
CSCAN调度算法示例	



# SSTF、SCAN及CSCAN存在的问题

## ■磁臂粘着

在**SSTF**、**SCAN**及**CSCAN**几种调度算法中，可能出现磁臂停留在某处的情况，即反复请求某一磁道，从而垄断了整个磁盘设备，这种现象称为磁臂粘着。

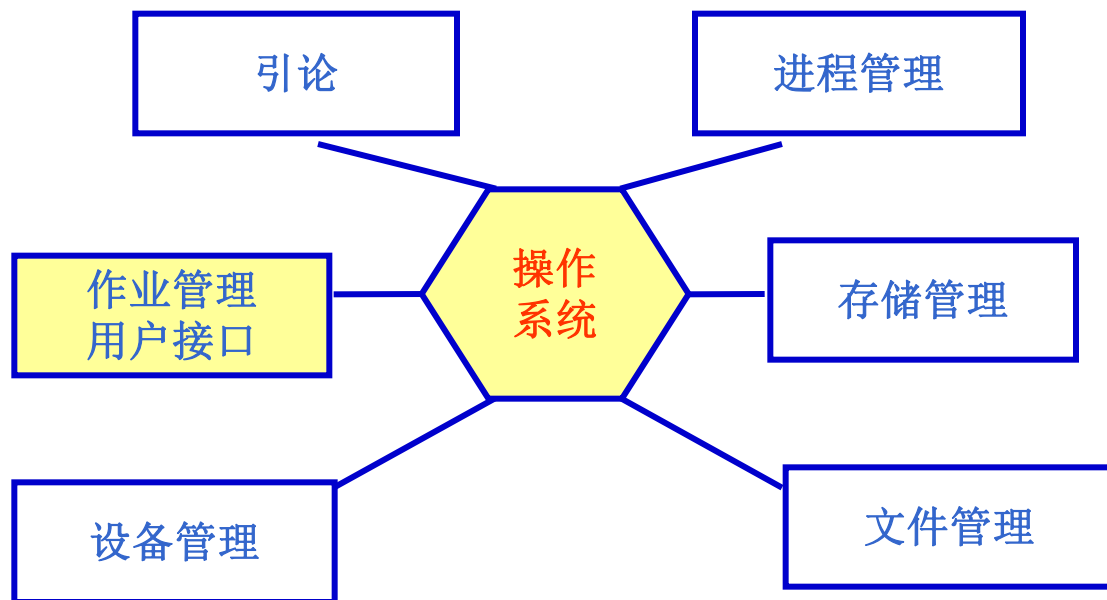


# 磁盘调度算法

---

- 先来先服务FCFS
- 最短寻道时间优先SSTF
- 扫描(SCAN) 算法
- 循环扫描(CSCAN) 算法
- N-STEP-SCAN调度算法 (理解)
- FSCAN调度算法(理解)



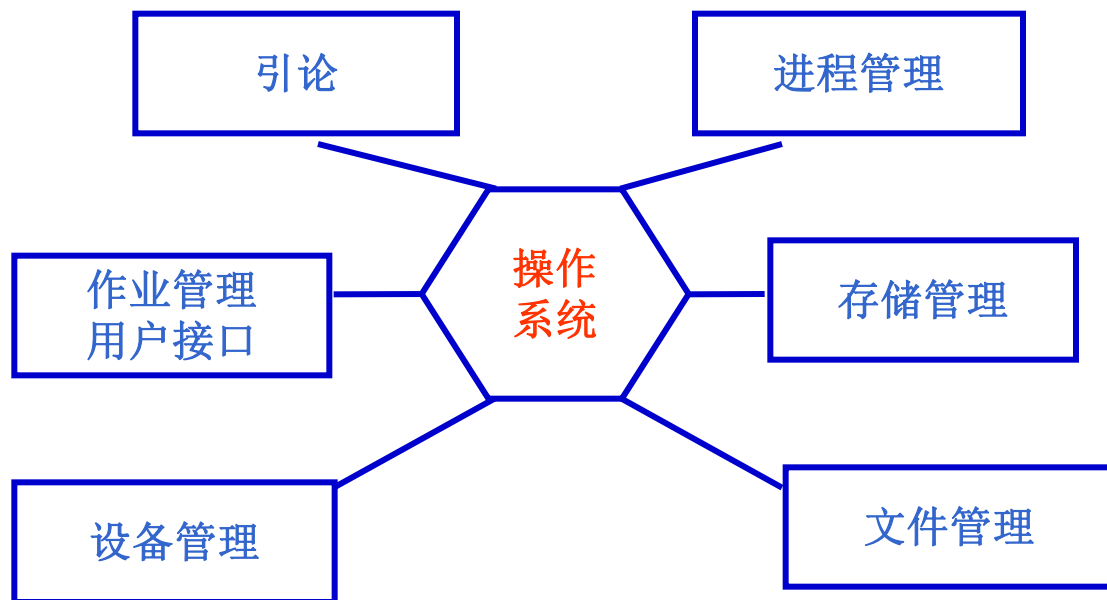




# 操作系统接口

---

- 作业接口（命令接口）
  - 脱机用户接口
  - 联机用户接口
- 图形用户接口。
  - 通过出现在屏幕上的对象直接进行操作，以控制和操纵程序的运行。
- 程序接口（系统调用）
  - 由一组系统调用组成，每一个系统调用都是一个完成特定功能的子程序。





# 操作系统里的程序思维

- 一切都是0和1
- 可执行文件的运行机理
  - 软件破解
- 进程控制
  - 创建，结束
  - 管道 -> 操控FFMPEG等控制台程序
- 多线程编程
  - 多线程文件下载
  - 线程同步
- 深入理解内存
  - 各种变量和指针
  - 游戏修改器
- 文件管理
  - 文件操作的方法：加工文件数据，加密解密文件数据
  - 文件在磁盘扇区中的二进制存在 (Winhex)
  - 恢复删除数据，彻底删除数据

