



# 第4章 内存管理(Memory Management)

---

## 4.1 概述

## 4.2 程序的连接与装入

## 4.3 实存储器管理

## 4.4 虚拟存储管理

## 4.5 小结



## 4.1 概述

---

### 一、存储器的层次结构

**存储系统的设计目标可归纳成3个问题：**

容量，速度，成本

- ✓ 容量：需求无止境
- ✓ 速度：能匹配处理器的速度
- ✓ 成本：成本和其他部件相比应在合适范围之内



# 存储器的层次结构

---

## 容量、速度和成本之间的矛盾：

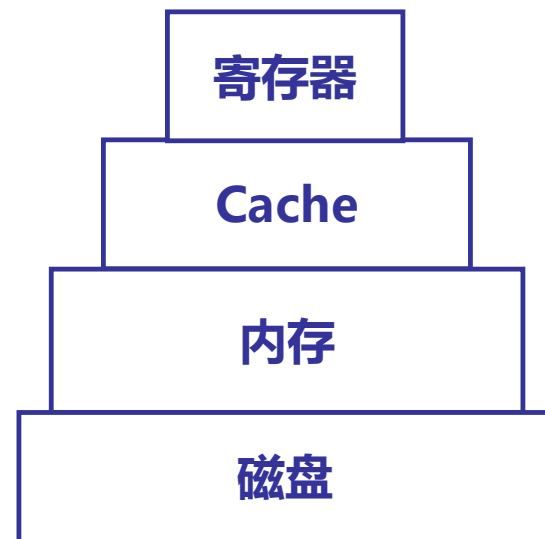
- ✓ 三个目标不可能同时达到最优，要作权衡
- ✓ 存取速度越快，每一个位（Bit）的价格越高
- ✓ 追求大容量，就要降低存取速度
- ✓ 追求高速度，就要降低容量

# 存储器的层次结构

**解决方案：**采用**层次化的存储体系结构**

当沿着层次向下时

- ✓ 每位的价格下降
- ✓ 容量增大
- ✓ 速度变慢



**策略：**

- ✓ 较小、较贵的存储器由较大、较便宜的慢速存储器作为后援

**虚拟存储器**

- ✓ 降低较大、较便宜的慢速存储器的访问频率

**高速缓存**

# 存储器的层次结构

- ✓ **寄存器 ( Register )** : 在CPU内部，用来暂存数据、地址以及指令信息。在计算机的存储系统中它具有最快的访问速度。

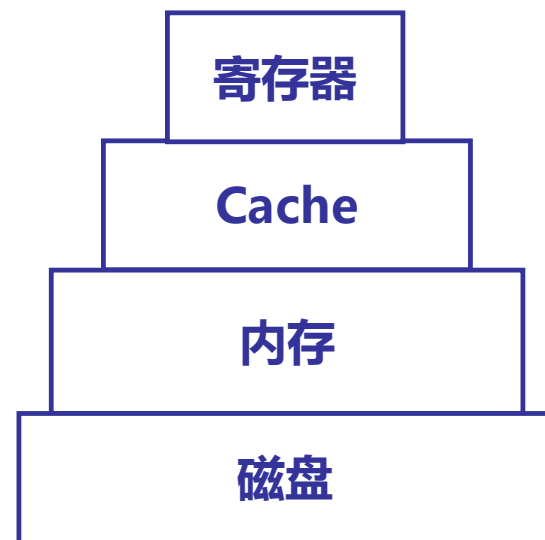
速度比主存快得多

造价高，容量一般都很小

- ✓ **内存** : 速度尽量快到与CPU取指速度相匹配，大到能装下当前运行的程序与数据，否则CPU执行速度就会受到内存速度和容量的影响而得不到充分发挥

- ✓ **高速缓存 ( cache )**

处于CPU和物理内存之间，访问速度快于内存，但低于寄存器





## 4.1 概述

---

### 二、内存管理的目的

- ✓ 有效利用内存空间

**帕金森 ( parkinson ) 定律**：内存有多大，程序就有多大

- ✓ 考虑管理的开销：时间，空间
- ✓ 应用程序不必特别考虑内存的大小



## 4.1 概述

---

### 三、内存管理的功能

- ✓ 记录内存的使用情况（是否空闲）
- ✓ 进程所占内存空间的分配与回收
- ✓ 当内存不足时，采取相应措施
- ✓ 内存空间的共享与保护



## 4.2 程序的连接与装入

---

### 一、程序连接的功能

多个目标文件及库文件连接成1个完整的可执行文件。

- ✓ 定位目标文件可能存在的一些外部符号
- ✓ 浮动地址的重定位

### 二、程序连接的时机

- ✓ 静态连接：静态的，只连接1次，多次运行
- ✓ 装入时连接：装入后是静态的
- ✓ 实际运行时连接：调用时动态连接





## 4.1 程序的连接与装入

---

### 三、程序的装入方式

程序能否直接在外存中执行？

不行！每个程序在运行前，必须装入内存。

不一定一次全部装入。

**装入方式：**

- ( 1 ) 完全静态装入
- ( 2 ) 静态重定位装入
- ( 3 ) 动态重定位装入



# 程序的装入方式

---

## 1. 完全静态装入

程序装入时不作任何修改。

即装入内存的每个字节与其可执行文件完全相同。

例如，早期DOS操作系统下的.com文件。



# 程序的装入方式

---

## 2. 静态重定位装入

程序**装入时进行**一次地址重定位，运行时不变。

**重定位：**

程序中的相对地址（从0开始）  绝对地址

**逻辑地址（相对地址）：**

用户的程序经过汇编或编译连接后形成可执行代码，代码通常采用相对地址的形式，其首地址为0，指令中的地址都采用相对于首地址的偏移量。

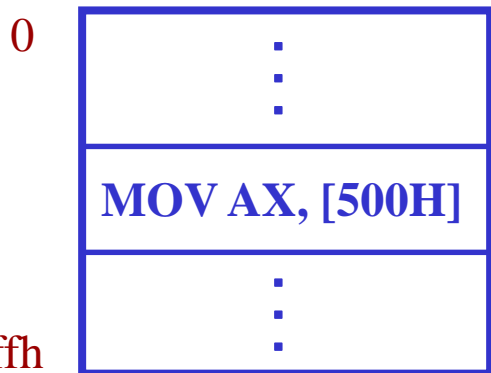
机器是不能用逻辑地址在内存中读取信息的。

**物理地址（绝对地址）：**

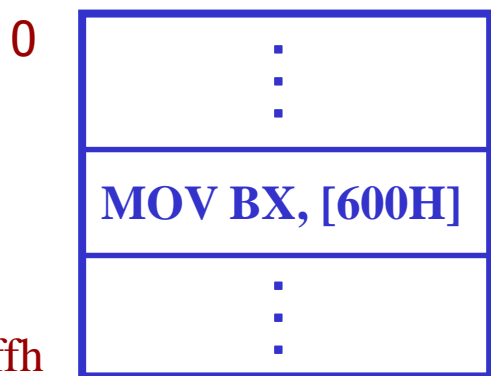
内存中存储单元的实际地址

# 静态重定位装入

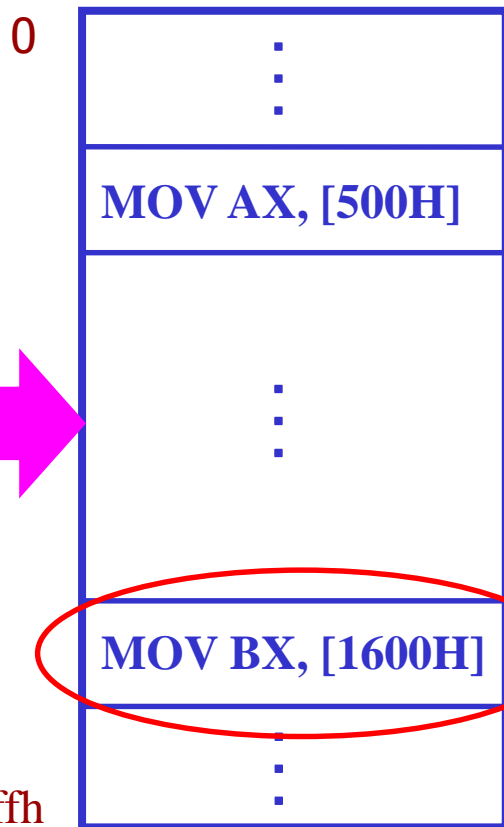
模块A



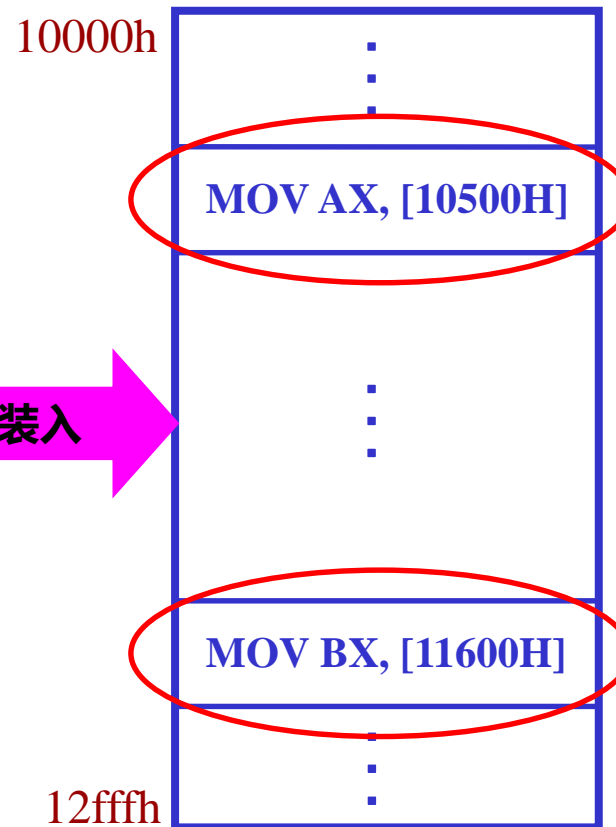
模块B



连接



装入





# 程序的装入方式

---

## 3. 动态重定位装入

真正**执行到一条指令要访问某个内存地址时**，才进行地址重定位。

好处：程序可以在内存中移动。

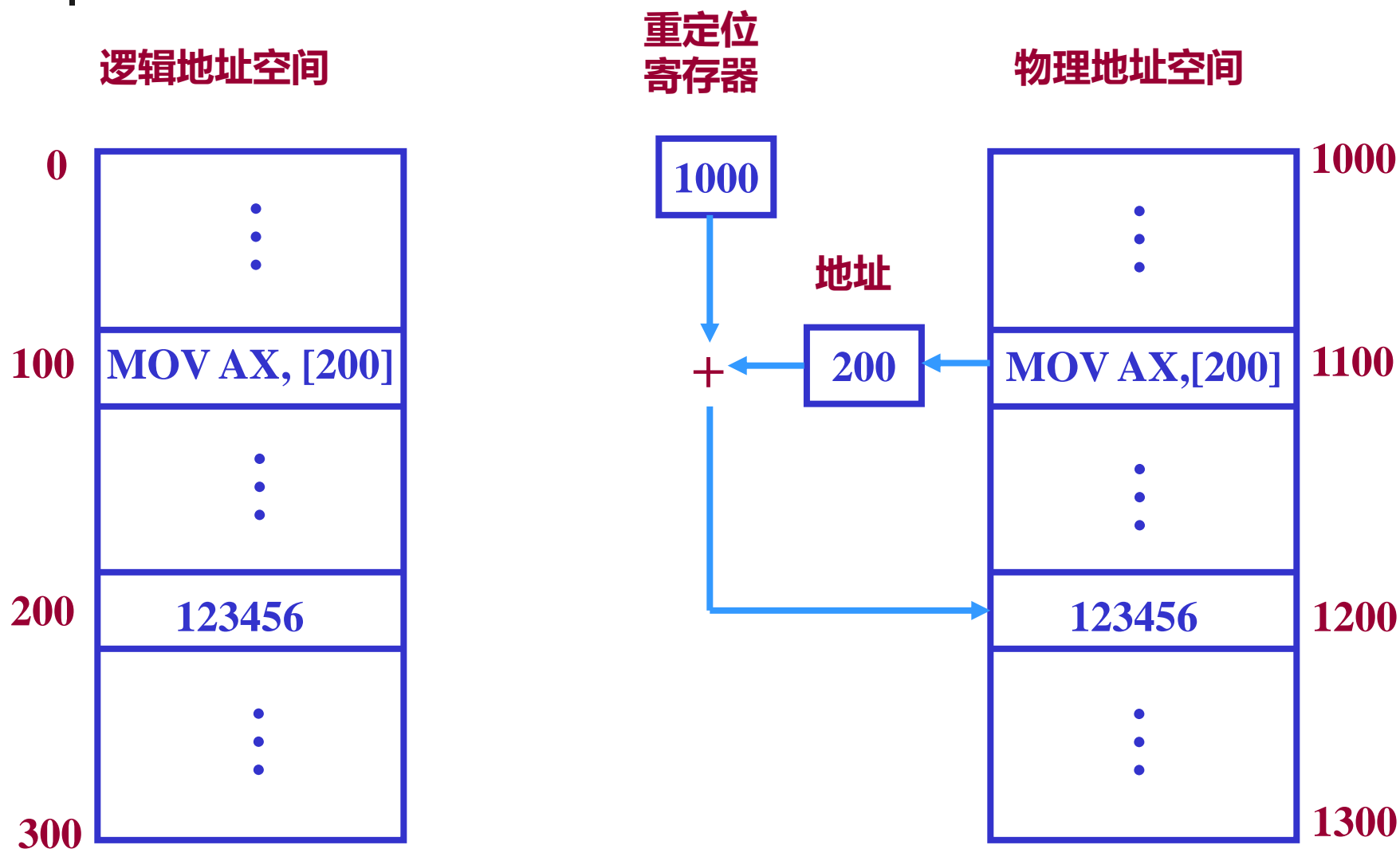
**如何实现动态重定位？**

一般设置1个重定位寄存器，

存放当前进程在内存的起始地址

**绝对地址 = 相对地址 + 重定位寄存器的值**

# 动态重定位装入





## 4.3 实存储器管理

---

程序的大小不能超过可用内存空间的大小。

### 一、连续分配

为每个进程分配连续的内存空间。

#### 1. 单一连续区分配

内存中只存在1个用户程序。

整个用户区为该程序独占。

**一般将内存划分为2个区：**

- ✓ 系统区：存放OS程序和数据
- ✓ 用户区：存放用户程序和数据

只能用于单用户、单任务OS。



# 连续分配

---

## 2. 固定分区

将内存的用户区预先划分为若干区域（分区）

分区个数和每个分区的大小是固定的

每个分区存放1个进程

**管理所需的数据结构：**

1个分区使用表（内存分配表），记录分区状态。



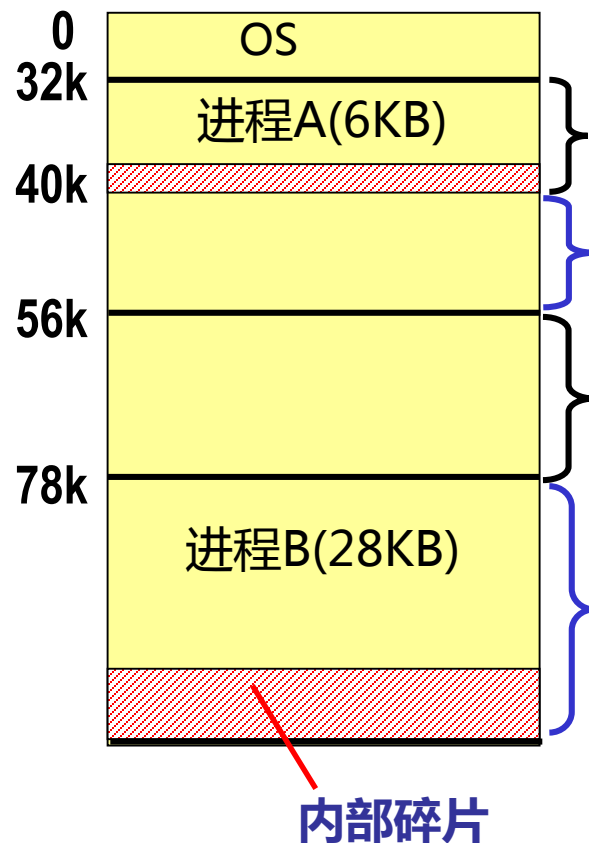
# 固定分区

分区使用表

| 区号 | 分区大小 | 起始地址 | 状态   |
|----|------|------|------|
| 1  | 8kB  | 32k  | Used |
| 2  | 16kB | 40k  | Free |
| 3  | 22kB | 56k  | Free |
| 4  | 34kB | 78k  | Used |

## 存在的问题：

- (1) 超过最大分区的程序无法装入；
- (2) 存在内部碎片(Internal fragmentation)：  
由于程序长度小于分区，使得分区内部有空间浪费。





# 连续分配

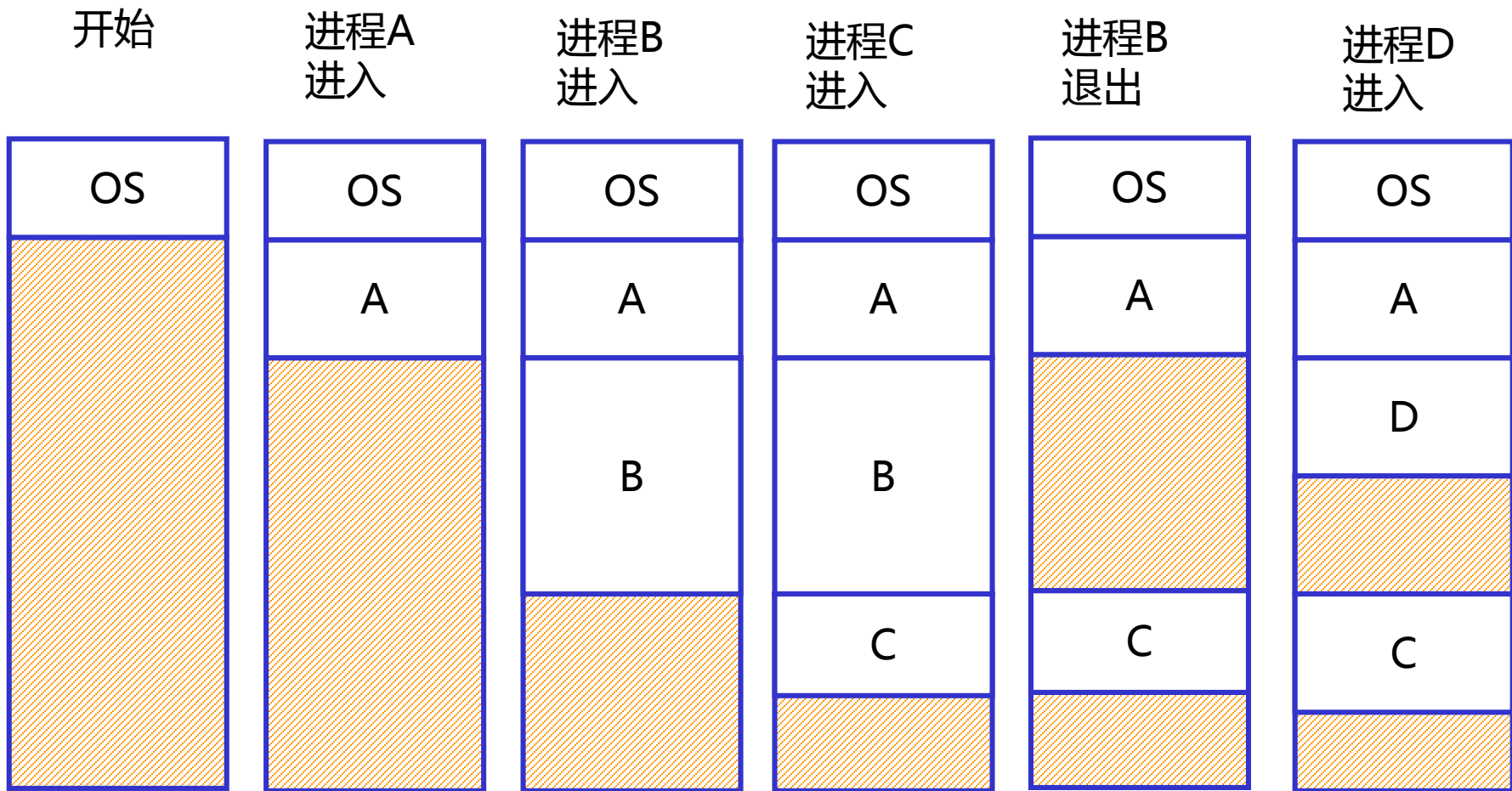
---

## 3. 可变分区 ( Variable Partition )

或称动态分区。

开始时只有1个空闲分区，随着进程的装入和退出，分区的个数和每个分区的大小、位置会动态变化。

# 可变分区



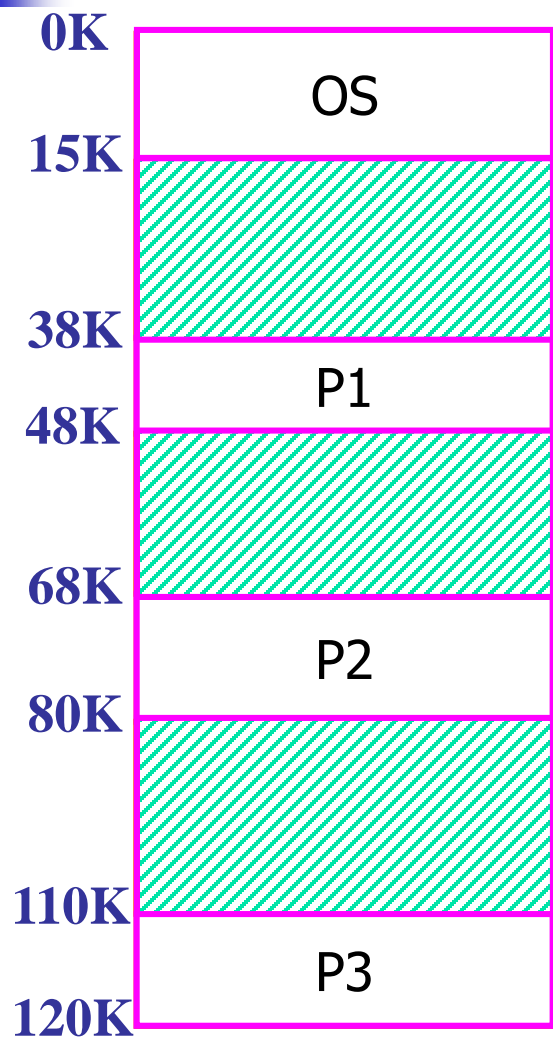


# 可变分区

---

## (1) 分区管理所需的数据结构

- ✓ 空闲分区表/空闲分区链  
记录空闲分区的起始地址和长度
- ✓ 已分配分区表

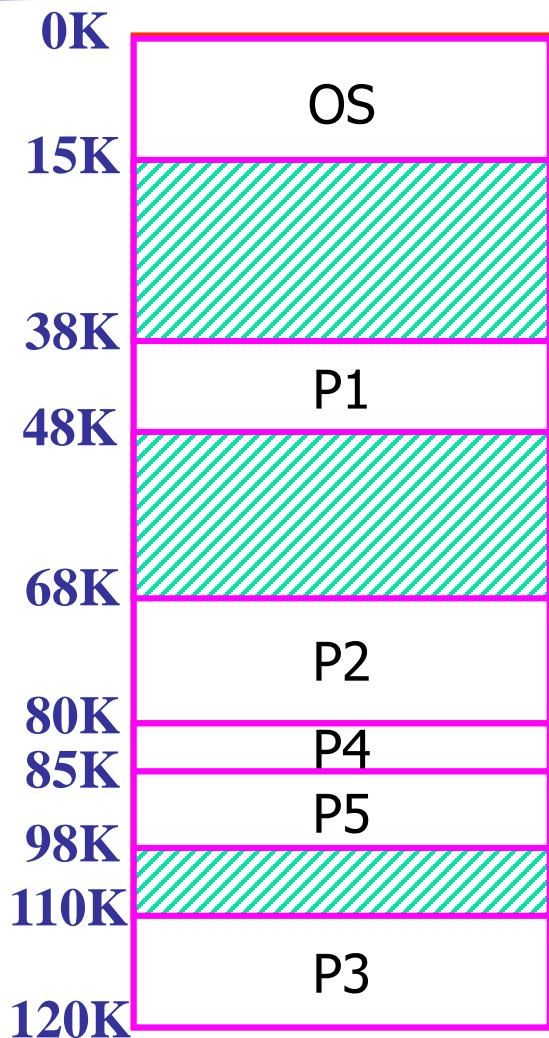
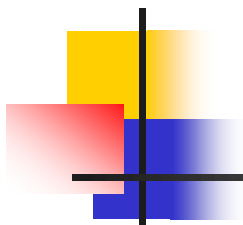


空闲分区表

| 始址  | 长度   | 状态  |
|-----|------|-----|
| 15K | 23KB | 未分配 |
| 48K | 20KB | 未分配 |
| 80K | 30KB | 未分配 |
|     |      | 空   |
|     |      | 空   |

已分配分区表

| 始址   | 长度   | 进程 |
|------|------|----|
| 38K  | 10KB | P1 |
| 68K  | 12KB | P2 |
| 110K | 10KB | P3 |
|      |      | 空  |
|      |      | 空  |



空闲分区表

| 始址  | 长度   | 状态  |
|-----|------|-----|
| 15K | 23KB | 未分配 |
| 48K | 20KB | 未分配 |
| 98K | 12KB | 未分配 |
|     |      | 空   |
|     |      | 空   |

已分配分区表

| 始址   | 长度   | 进程 |
|------|------|----|
| 38K  | 10KB | P1 |
| 68K  | 12KB | P2 |
| 110K | 10KB | P3 |
| 80K  | 5KB  | P4 |
| 85K  | 13KB | P5 |



# 可变分区

---

## (2) 分区分配算法

### ① 最先适配法 ( first fit )

空闲分区链 ( 表 ) 按地址递增的次序排列

从头开始, 选择第1个大小足够的分区

### ② 下次适配法 ( next fit )

从上次分配的分区的一个开始, 选择第1个大小足够的分区

### ③ 最佳适配法 ( best fit )

空闲分区链 ( 表 ) 按大小递增的次序排列

从头开始, 选择第1个大小足够的分区

### ④ 最差适配法 ( worst fit )

空闲分区链 ( 表 ) 按大小递减的次序排列

从头开始, 选择第1个分区 ( 如果足够大 )



# 可变分区

---

## (3) 分区的回收

- ✓ 将回收的分区插入到空闲分区链（表）的合适位置
- ✓ 合并相邻的多个空闲分区

**考虑：上邻、下邻、上下相邻、上下不相邻**

存在一个问题：**外部碎片**（External fragmentation）

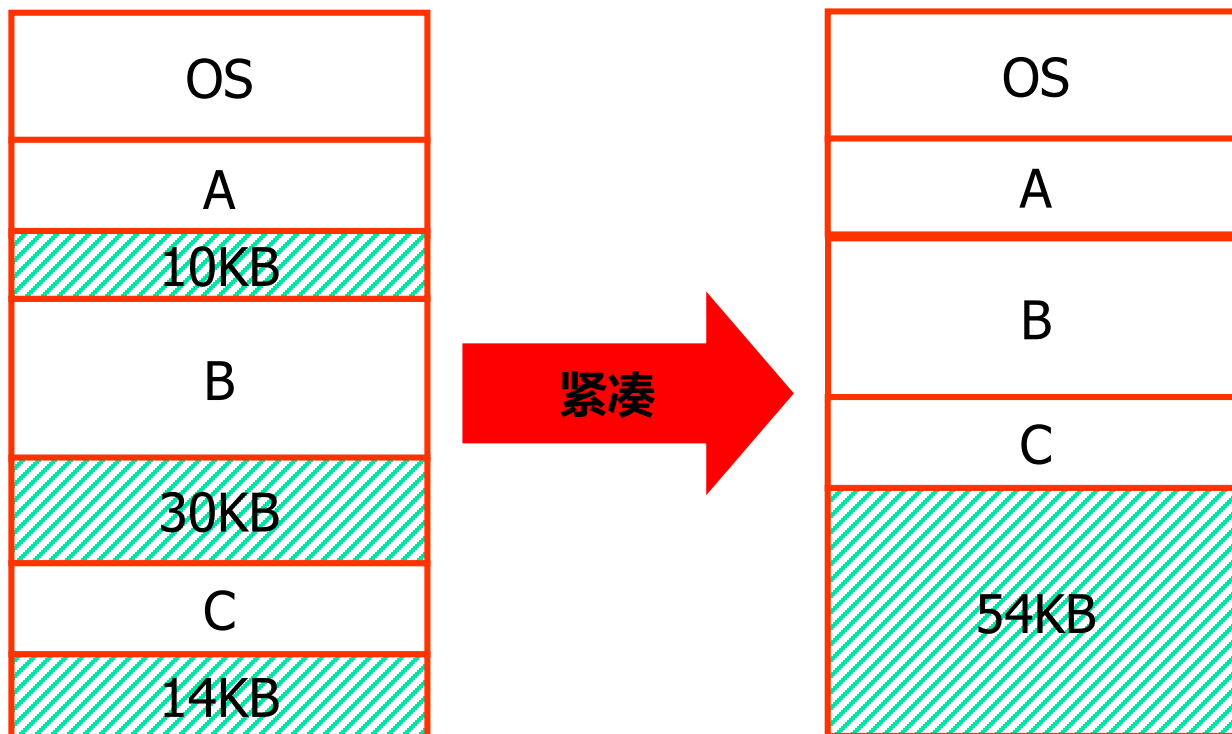
经过一段时间的分配和回收后，内存中存在很多很小的空闲分区。它们每一个都很小，不足以满足分配要求，但其总和满足分配要求。这些空闲块被称为碎片，造成存储资源的浪费。



# 可变分区

## (4) 如何解决碎片问题？

**内存紧凑** ( compaction ) : 集中小碎片为大分区



涉及到程序在内存中的移动，开销很大。



# 可变分区

---

为什么会产生大量碎片而难以利用呢？

**根本原因是：连续分配。**

如果把程序分成几部分装入不同分区呢？

**引入离散分配：**

- ✓ 分页
- ✓ 分段
- ✓ 段页式（分段 + 分页）



## 4.3 实存储器管理

---

### 二、分页（Paging，静态页式管理）

#### 1. 基本原理

（1）等分内存为物理块（或称页面，页框，page frame）

物理块编号为0，1，2，...。

（2）进程的逻辑地址空间分页（page）

**页大小 = 物理块大小**

页编号为0，1，2，...。

（3）内存分配原则：

**进程的1页可装入任一物理块**



# 分页

---

进程的最后1页可能装不满，产生“页内碎片”。

**分页类似于固定分区，但不同之处在于：**

- 1) 页比较小，且大小相等；
- 2) 一个进程可占据多页，且不要求连续。

进程的逻辑地址如何构成？

为实现分页管理，OS需要记录什么信息？

如何实现逻辑地址到物理地址的转换？



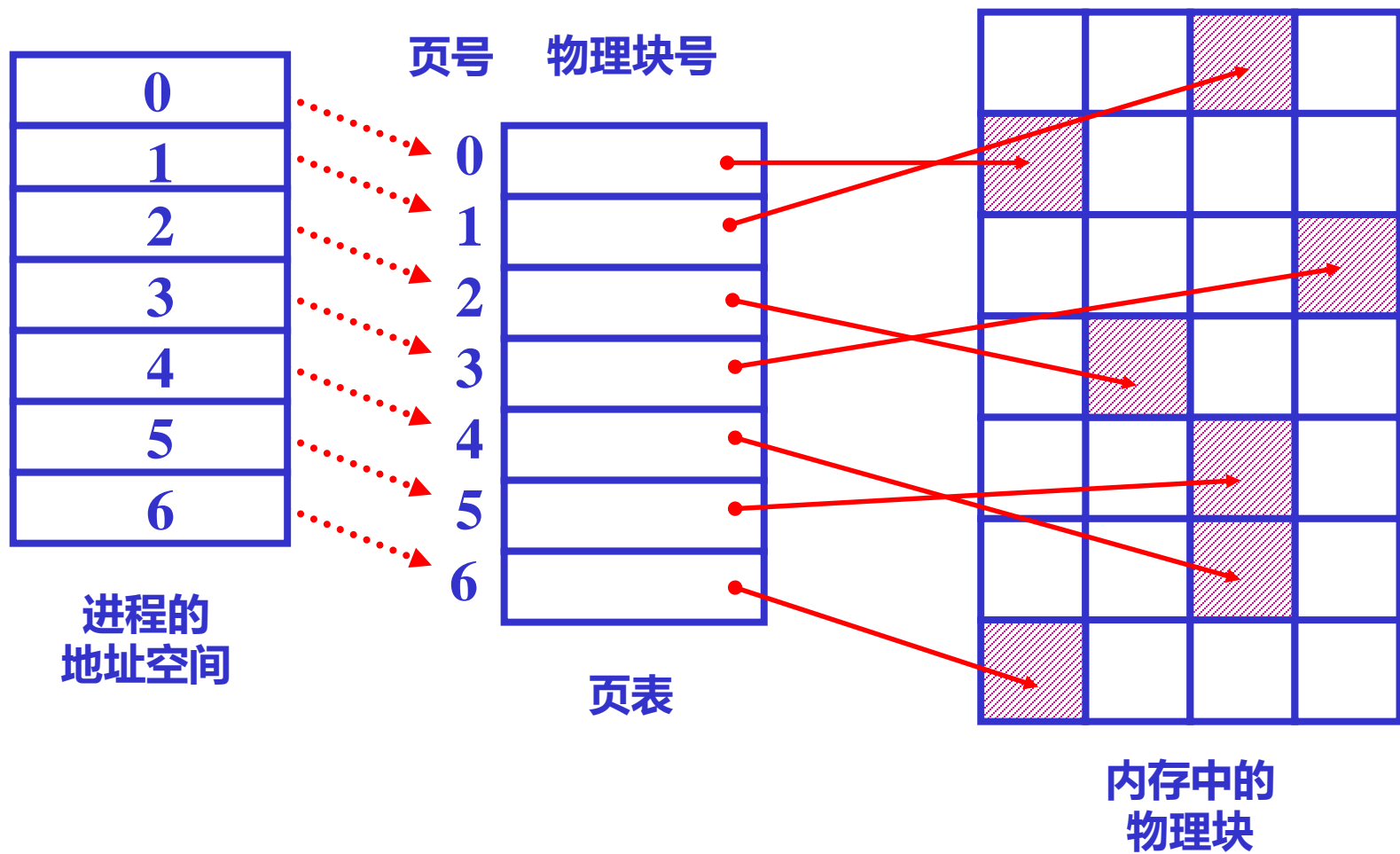
# 分页

---

## 2. 管理需要的数据结构

OS为每个进程建立1个**页表** ( Page Table )

记录进程的页号和物理块号的对应关系





# 分页

---

## 3. 地址变换

逻辑地址 → 物理地址

### ( 1 ) 页表寄存器 ( Page Table Register, PTR )

系统设置1个页表寄存器，存放当前进程的页表起始地址和长度

每个进程的页表起始地址和长度平时放在其PCB中，调度时由OS放入PTR

# 分页

## (2) 页大小的选择

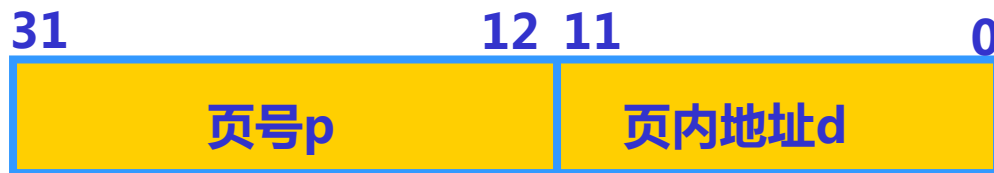
页小：碎片小，但页表占用空间大

页大：页表小，但页内碎片大

通常，页的大小为2的整数次幂

## (3) 进程的逻辑地址结构

地址的高位部分为页号，低位部分为页内地址



## (4) 基本的地址变换



# 分页

页表寄存器

逻辑地址

页表起始地址b

页表长度L

页号p

页内地址d

+

N

$p \geq L$

Y

$b[p]$

物理块号

地址越界

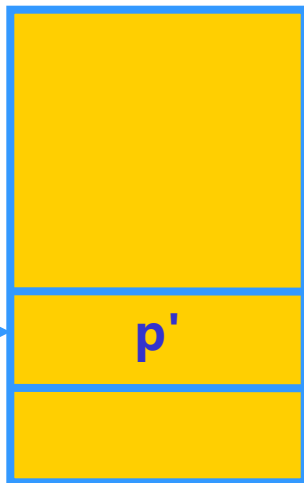
$p'$

页表

$p'$

d

物理地址



# 分页

【例】设逻辑地址是16位，  
页大小为 1KB = 1024B，  
即第0-9位是页内地址，第  
10-15位是页号。

逻辑地址空间

|       |
|-------|
| page0 |
| page1 |
| page2 |
| page3 |

页号 页表

|   |   |
|---|---|
| 0 | 1 |
| 1 | 4 |
| 2 | 3 |
| 3 | 7 |

块号 物理地址空间

|   |       |
|---|-------|
| 0 |       |
| 1 | page0 |
| 2 |       |
| 3 | page2 |
| 4 | page1 |
| 5 |       |
| 6 |       |
| 7 | page3 |
| 8 |       |

已知逻辑地址05DEH，即0000010111011110B，

物理地址为：

11DEH，即0001000111011110B

因为页号 = 1，对应的物理块号 = 4

物理地址 = 块号 × 页大小 + 页内地址



# 基本的地址变换

---

CPU要存取一个数据时，需要访问内存几次？

2次。

- 第1次：访问页表，找到该页对应的物理块号，将此块号与页内地址拼接形成物理地址；
- 第2次：访问该物理地址，存取其中的指令或数据。



# 分页

---

## (5) 引入快表的地址变换

快表，又称联想存储器(Associative Memory)：

具有并行查找能力的特殊高速缓冲存储器 ( cache )。

在x86系统中叫做TLB ( Translation lookaside buffers )

用途：保存当前进程的页表的子集 ( 部分表项 ) ，比如最近访问过的页表项。

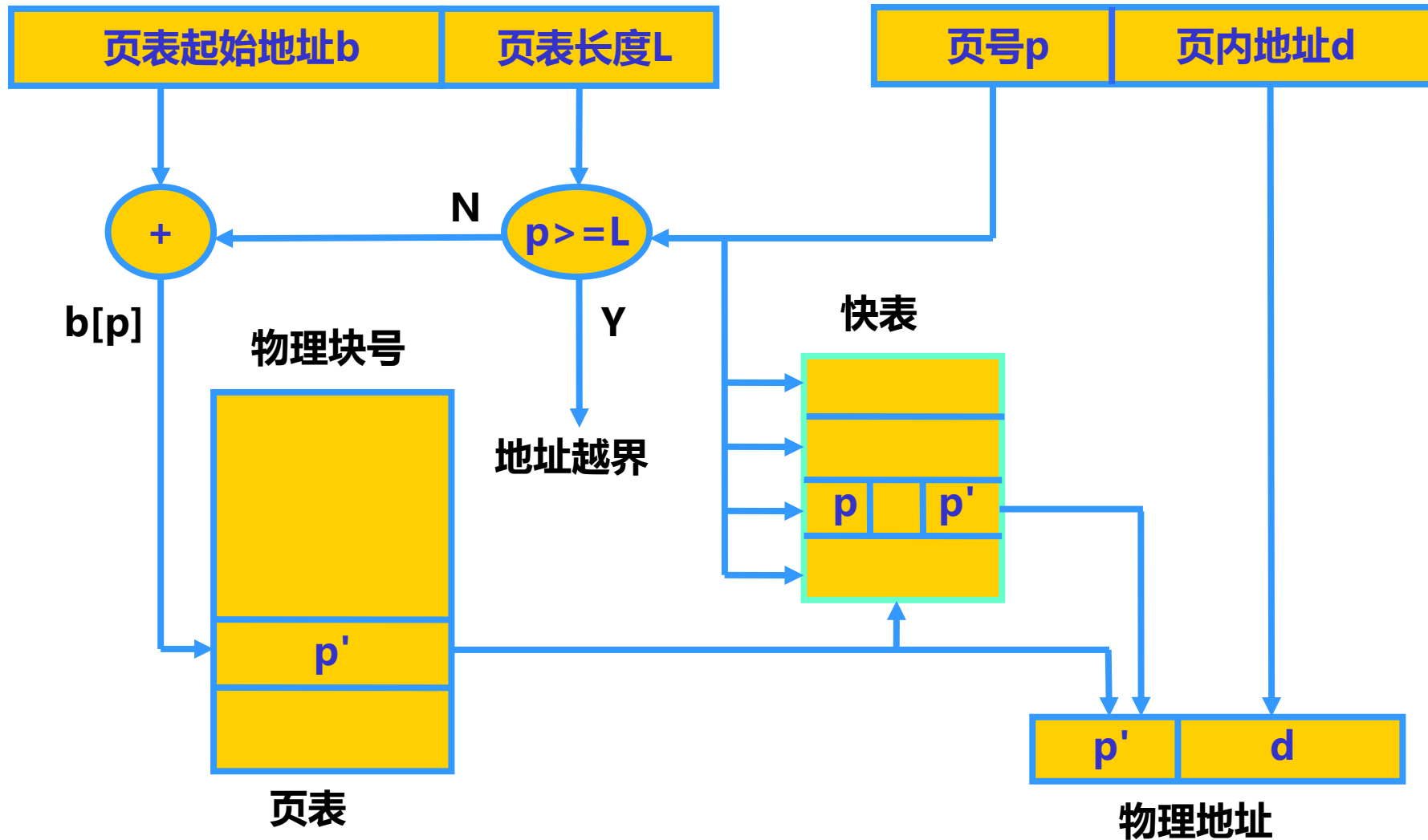
当切换到新进程时，快表要刷新。

目的：提高地址变换速度

# 分页

页表寄存器

逻辑地址





# 分页

---

## 快表表项：

- ✓ 页号：进程访问过的地址空间的页号
- ✓ 块号：该页所对应的物理块号
- ✓ 访问位：指示该页最近是否被访问过（0：没有被访问，1：访问过）
- ✓ 状态位：该快表项是否被占用（0：空闲，1：占用）

为了保证快表中的内容为现正运行进程的页表内容，**在每个进程被选中时，由恢复现场程序把快表的所有状态位清0，或恢复已保存的快表内容。**



# 分页

---

## 具有快表的地址变换过程：

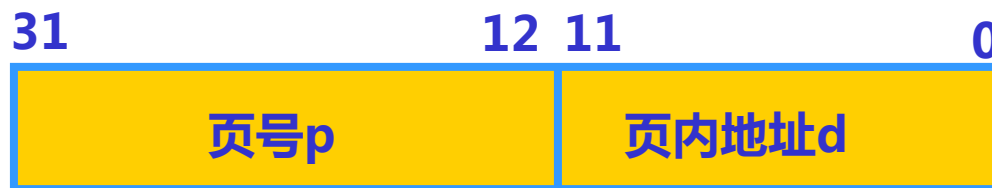
- ①当进程访问一页时，系统将页号与快表中的所有项进行并行比较。若访问的页在快表中，即可取得块号，生成物理地址。
- ②当被访问的页不在快表中时，就要根据页号查内存中的页表，找到对应的物理块号，生成物理地址。同时将页号与块号填入快表中，若快表已满，则置换其中访问位为0的一项。

说明：

- ✓ 由于成本的关系，快表不能太大；
- ✓ 引入快表的效果，取决于快表访问时的**命中率**(hit ratio)。

# 分页

**问题：**页表可能占用相当大的内存空间，而且是连续的。



设每个页表项占4B，页表最多可占用的内存空间是多少？

每个进程最多可达到 $2^{20}$  ( 1M ) 个页

页表占用的内存空间 = 4MB

如何解决？



# 二级页表

## 4. 二级页表

### (1) 基本原理

将页表进行分页，页大小 = 物理块大小

设置1个一级页表，多个二级页表

一级页表：第i项记录第i号二级页表所在的物理块号

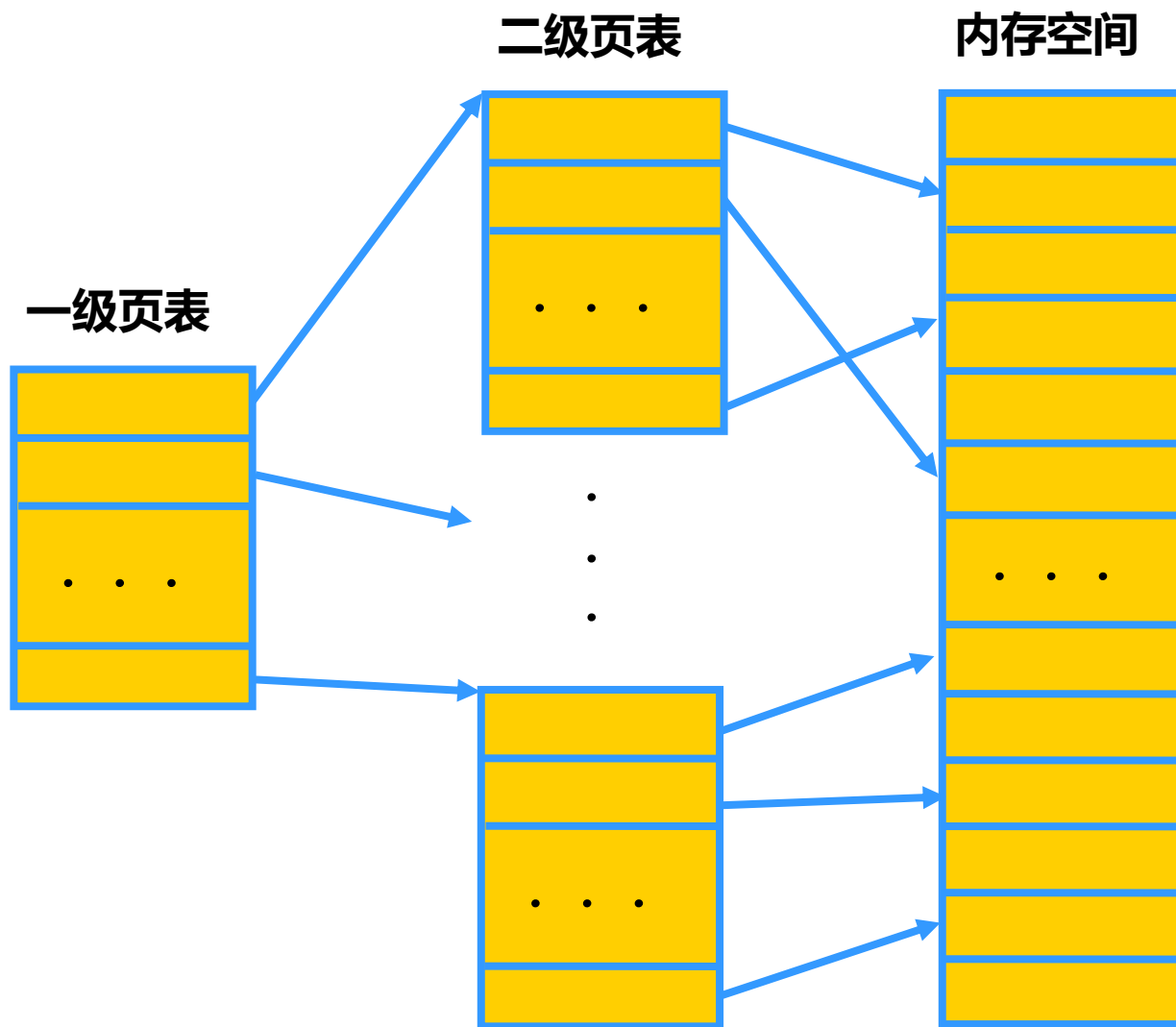
二级页表：第i项记录第i页对应的物理块号

系统设置1个页表寄存器，存放一级页表的起始地址和长度

### (2) 进程的逻辑地址结构：

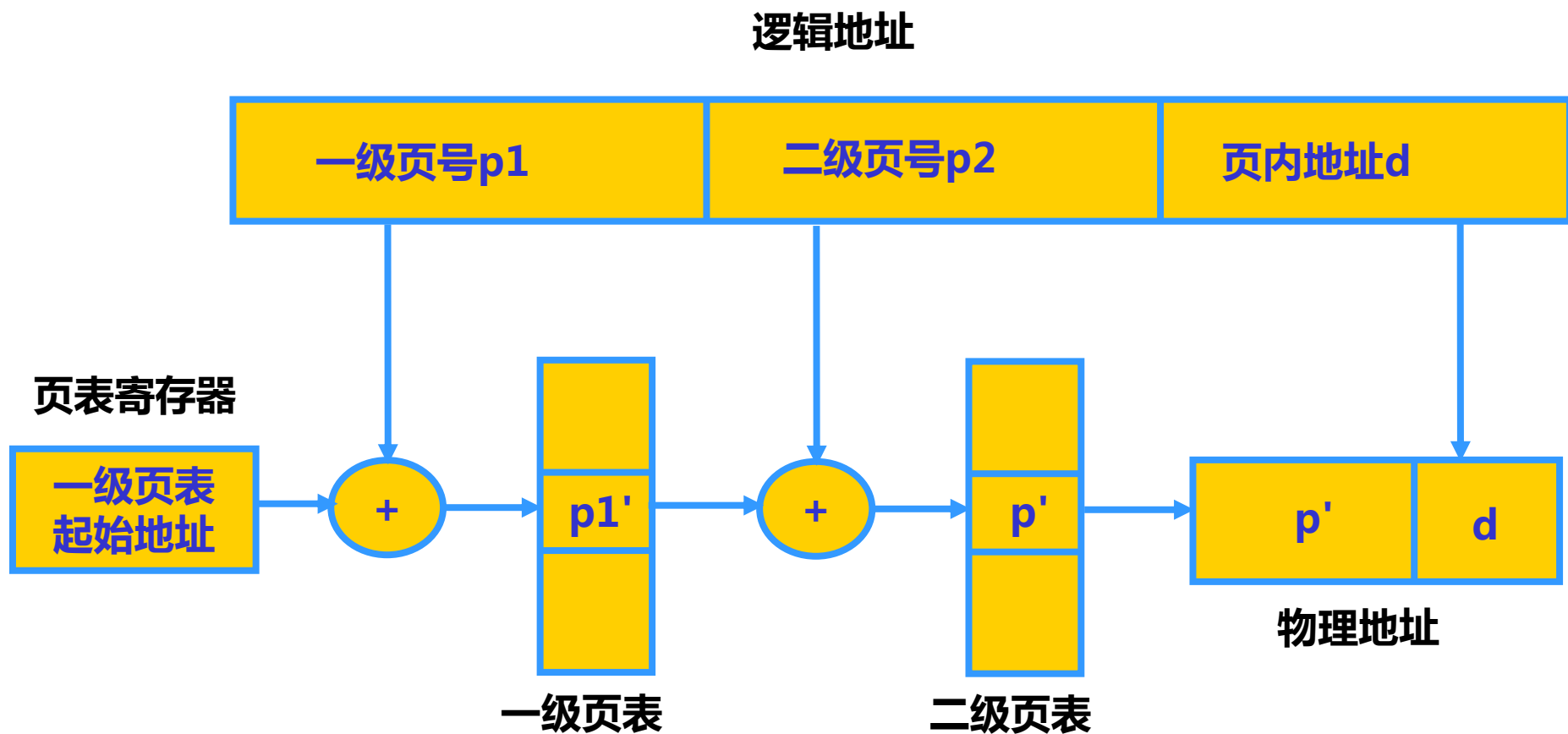


# 二级页表



# 二级页表

## (3) 地址变换





## 二级页表

---

### 说明：

二级页表并未减少页表所占的内存空间，但解决了页表的离散分配问题。

对于32位地址，采用二级页表是可以的。

如果是64位地址呢？

需要三级甚至更多级的页表。



## 4.3 实存储器管理

---

### 三、分段 ( Segmentation , 静态段式管理 )

#### 1. 基本原理

( 1 ) 进程的逻辑地址空间分段 ( segment ) :

按程序自身的逻辑关系划分为若干个程序段

每一个段是连续的

各个段的长度不要求相等

段号从0开始。

( 2 ) 内存分配原则 :

**以段为单位 , 各段不要求相邻**



# 分段

---

**分段类似于可变分区，但不同之处在于：**

一个进程可占据多个分区，而且分区之间不要求连续。

分段无内部碎片，但有外部碎片。

进程的逻辑地址如何构成？

为实现分段管理，OS需要记录什么信息？

如何实现逻辑地址到物理地址的转换？



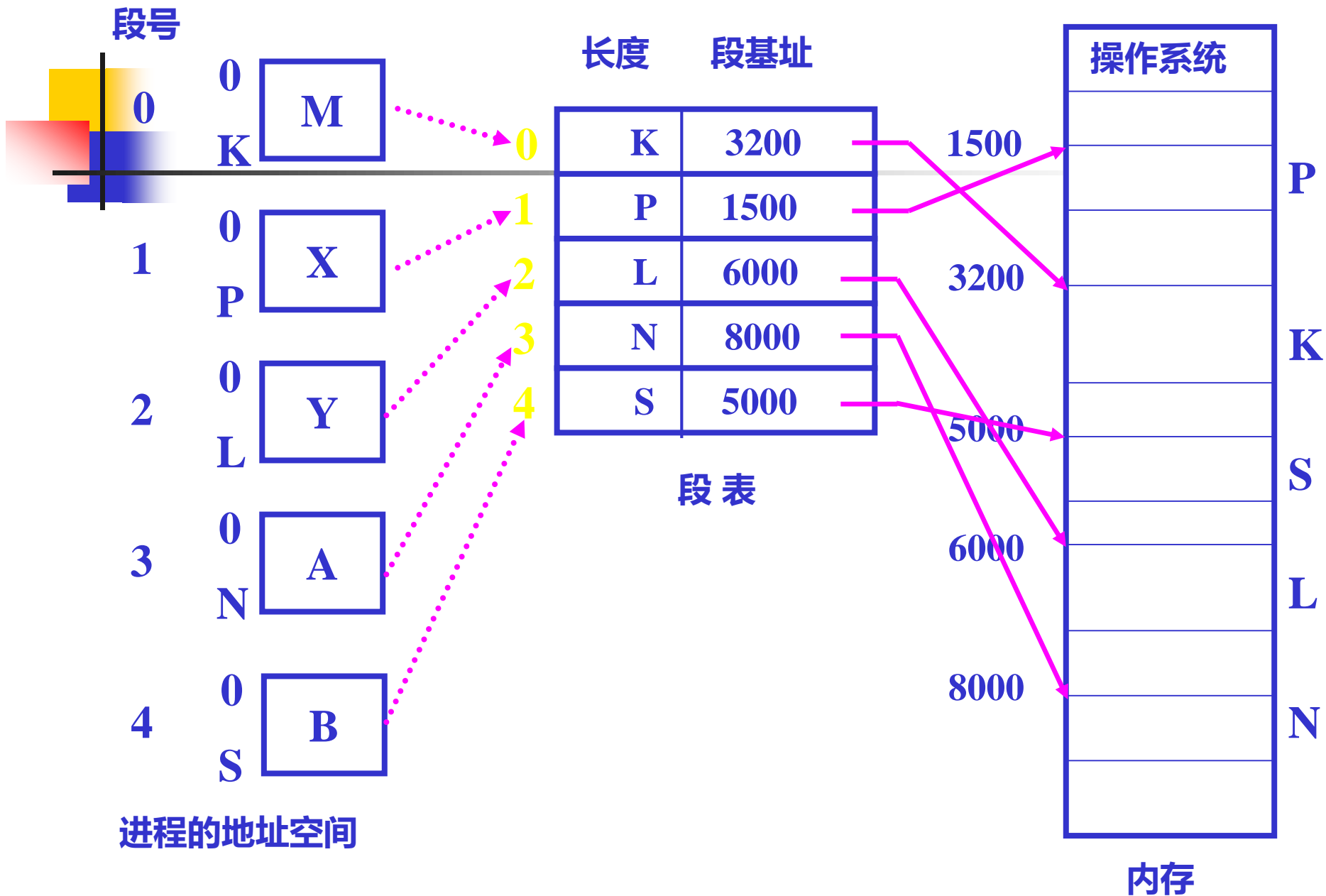
# 分段

## 2. 管理需要的数据结构

OS为每个进程建立1个**段表** ( Segment Table )

每个段在段表中有1项，记录该段在内存中的基址和长度

| 段号 | 段基址  | 段长度  |
|----|------|------|
| 0  | 58K  | 20K  |
| 1  | 100K | 110K |
| 2  | 260K | 140K |





# 分段

## 3. 地址变换

### (1) 段表寄存器

系统设置1个段表寄存器，存放**当前进程**的段表起始地址和长度  
每个进程的段表起始地址和长度平时放在其PCB中

### (2) 进程的逻辑地址结构

地址的高位部分为段号，低位部分为段内地址

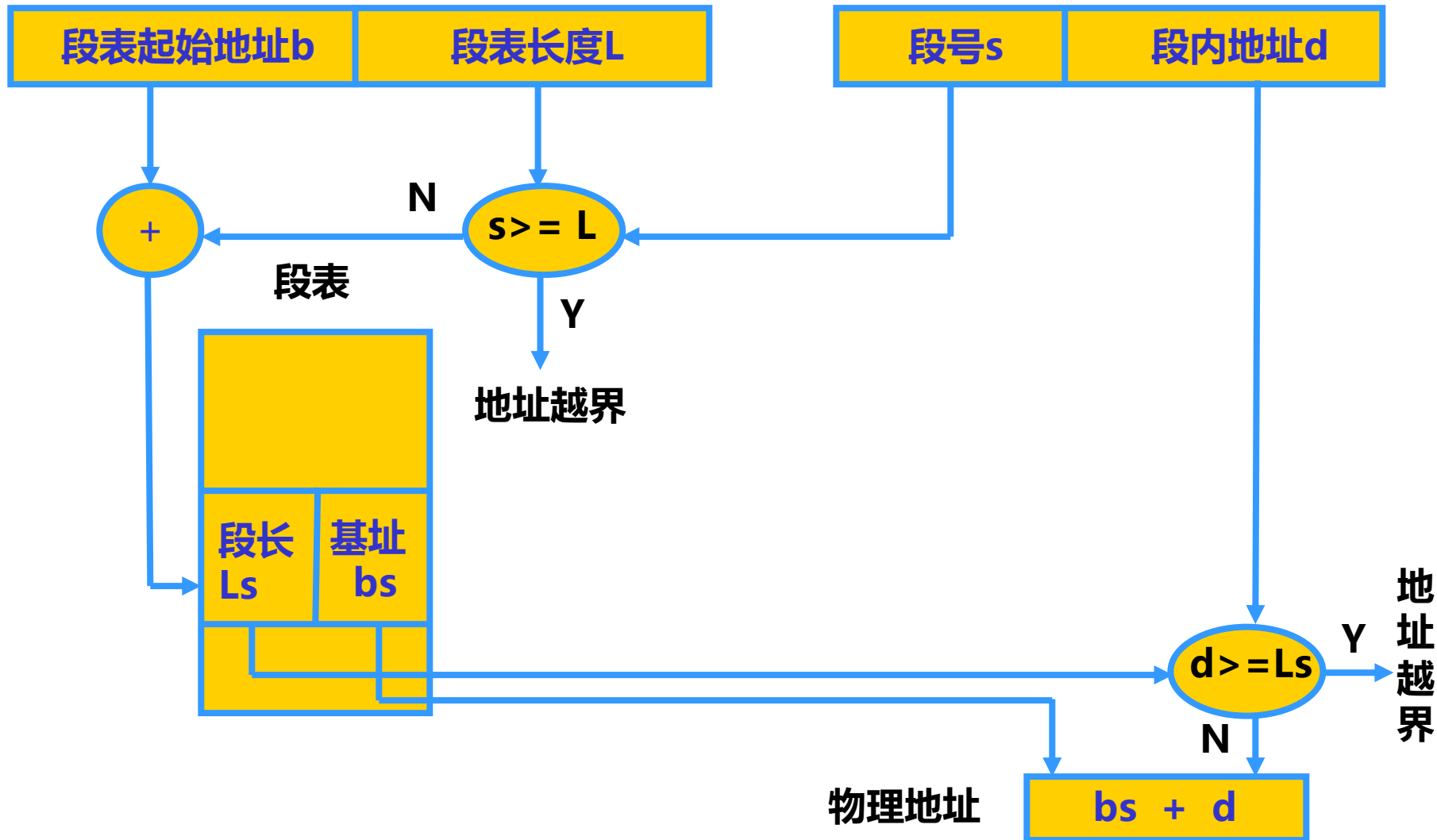


### (3) 基本的地址变换

# 分段

段表寄存器

逻辑地址





# 分段

【例】设逻辑地址是16位，段号占4位，段内地址占12位。

已知逻辑地址12f0h，即0001001011110000b，  
物理地址为：

2310h

段号 = 1，

段基址 = 2020h，段内地址 = 2f0h

| 段号 | 基址    | 长度 |
|----|-------|----|
| 0  | 1000h | 3k |
| 1  | 2020h | 4k |
| 2  | 4000h | 4k |
| 3  | 6300h | 2k |

段表



# 基本的地址变换

---

CPU要存取一个数据时，需要访问2次内存。

- 第1次：访问段表，找到该段的基址，将基址与段内地址相加形成物理地址；
- 第2次：访问该物理地址，存取其中的指令或数据。



# 分段

---

## (4) 引入快表的地址变换

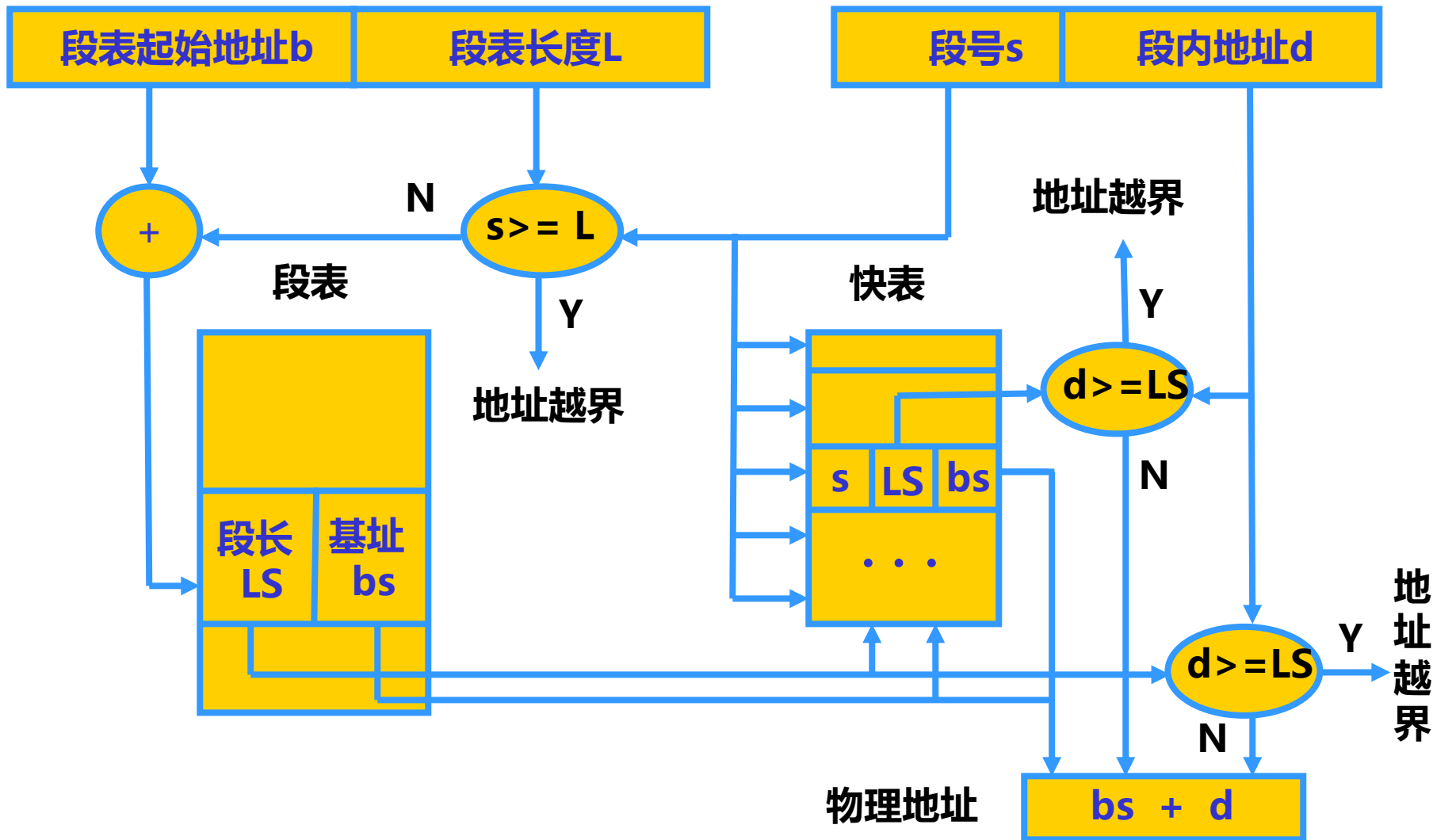
**快表表项：**

- ✓ 段号
- ✓ 段基址
- ✓ 段长度
- ✓ 访问位
- ✓ 状态位

# 分段

段表寄存器

逻辑地址





# 分段

---

## 4. 分段与分页的比较

- ( 1 ) **分页对程序员是不可见的；分段通常是可见的**，并作为组织程序和数据的手段提供给程序员；
- ( 2 ) **页的大小由系统确定，段的大小由用户程序确定**；
- ( 3 ) 分段更利于多个进程共享程序和数据；
- ( 4 ) 分段便于实现动态链接；
- ( 5 ) 分页可有效提高内存利用率，分段可更好地满足用户需要。

如果把分段和分页结合起来呢？



## 4.3 实存储器管理

---

### 四、段页式管理（分段 + 分页）

动机：结合分段和分页的优点，克服二者的缺点。

#### 1. 基本原理

（1）进程分段：同段式管理

每段分页，内存分块，内存以块为单位分配：同页式管理

（2）进程的逻辑地址结构

由段号、页号和页内地址构成。

|     |       |       |
|-----|-------|-------|
| 段号s | 段内页号p | 页内地址d |
|-----|-------|-------|





# 段页式管理

---

## 2. 管理需要的数据结构

### ✓ 每个进程1个段表

记录每个段对应的页表起始地址和长度

### ✓ 每个段有1个页表

记录该段所有页号与物理块号的对应关系



# 段页式管理

---

## 3. 地址变换

### (1) 段表寄存器

系统设置1个段表寄存器，存放**当前进程**的段表起始地址和长度  
每个进程的段表起始地址和长度平时放在其PCB中

### (2) 基本的地址变换

CPU要存取一个数据时，需要访问3次内存。

- 第1次：访问段表，获得该段的页表地址；
- 第2次：访问页表，取得物理块号，形成物理地址；
- 第3次：访问该物理地址，存取其中的指令或数据。



# 段页式管理

---

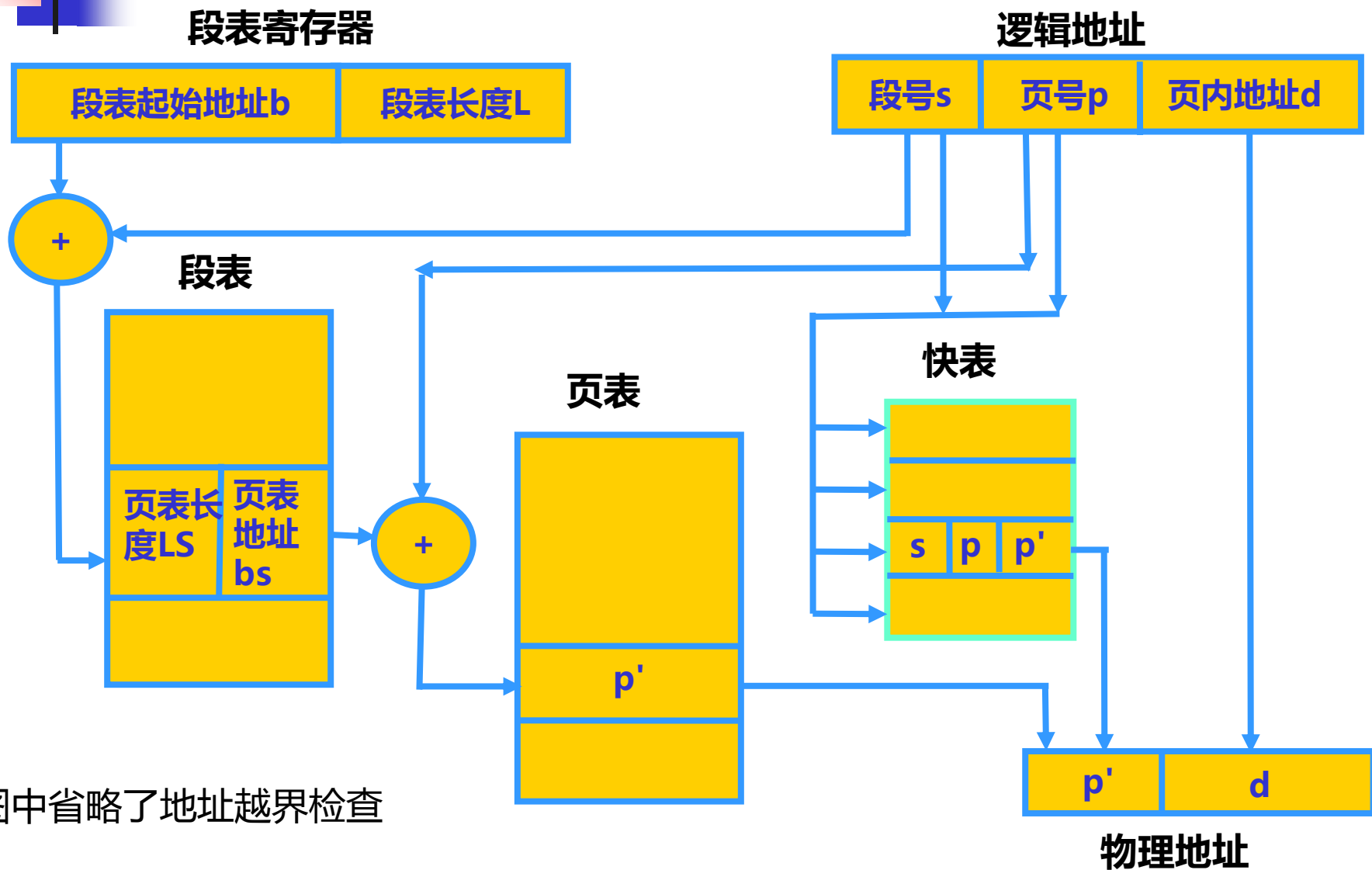
## (4) 引入快表的地址变换

**快表表项：**

- ✓ 段号
- ✓ 页号
- ✓ 物理块号
- ✓ 访问位
- ✓ 状态位

同时利用段号与页号查找相应的物理块号。

# 地址变换



图中省略了地址越界检查



## 4.3 实存储器管理

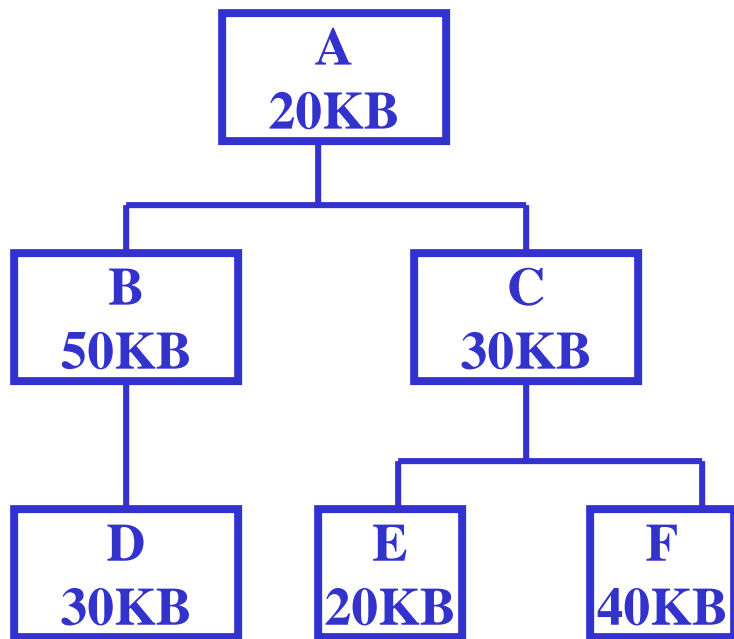
---

### 五、覆盖

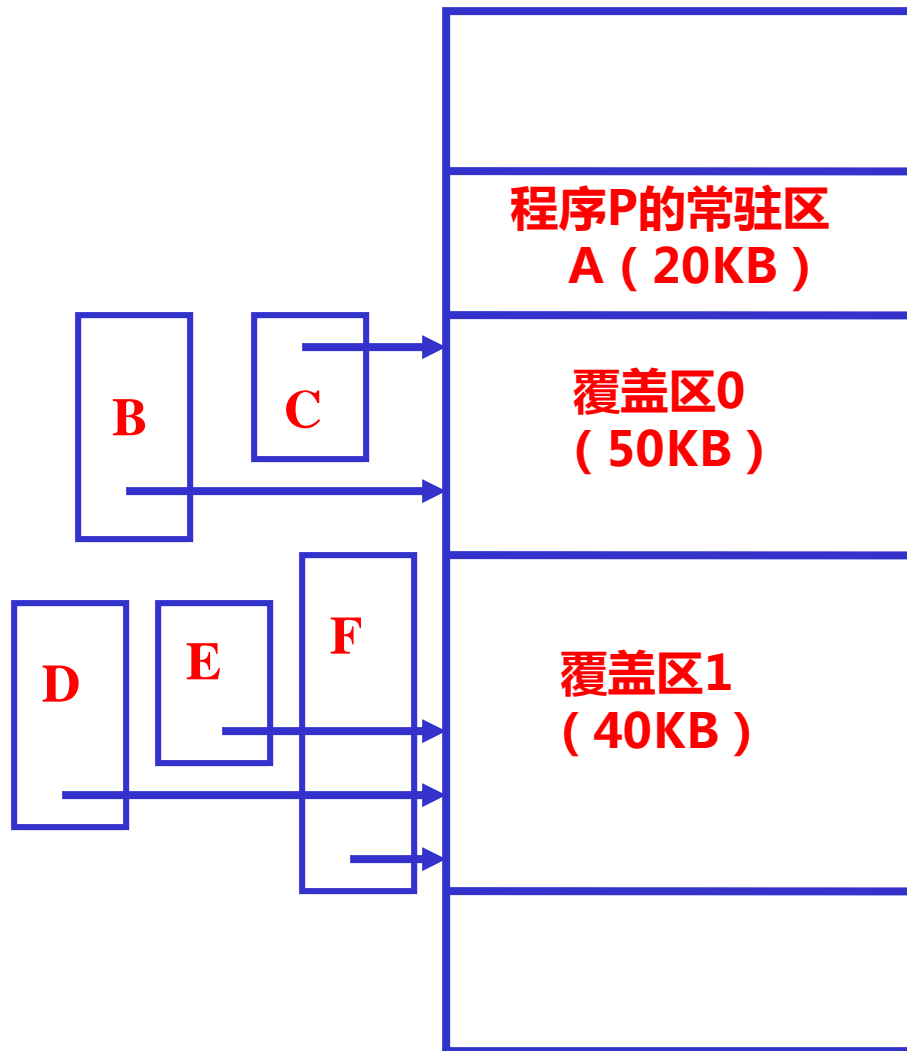
#### 基本原理：

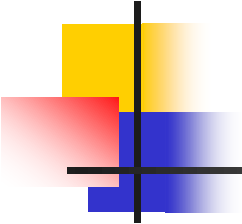
- (1) 把程序划分为若干个功能上相对独立的程序段（称为覆盖块），按照其自身的逻辑结构使那些不会同时执行的程序段共享同一块内存区域；
- (2) 覆盖块存放在磁盘上，当一个程序段执行结束，把后续程序段调入内存，覆盖前面的程序段（内存“扩大”了）。

# 覆盖



程序P的调用结构





# 覆盖

---

## 缺点：

要求程序员划分程序，提供一个明确的覆盖结构；

对用户不透明，增加了用户负担。

这种技术曾用在早期的操作系统中。



## 4.4 虚拟存储管理

---

### 一、虚拟存储器 ( Virtual Memory )

#### 1. 基本思想

进程的大小可以超过可用物理内存的大小，

由OS把当前用到的那部分留在内存，其余的放在外存中。

#### 2. 几个概念

- ✓ 虚拟地址：程序中使用的地址。进程的虚拟地址从0开始。
- ✓ 物理地址：可寻址的内存实际地址
- ✓ 虚拟地址空间：虚拟地址的集合
- ✓ 物理地址空间：实际的内存空间





# 虚拟存储器

---

## 3. 交换 ( Swapping ) 技术

借助于外存 ( 磁盘 ) , 将当前要使用的那部分程序或数据装入内存 , 将暂时不需要的放在磁盘上 , 待需要时再装入。

**交换** : 进程的整体或一部分的换入/换出。

**换入** : 从磁盘移入内存

**换出** : 从内存移出到磁盘

**交换是实现虚拟存储器的基础。**

最初的交换是针对整个进程的交换。



# 交换技术

---

## ( 1 ) 交换区 ( 交换空间 ) 的概念

磁盘上为虚拟内存保留的区域，称为交换区或交换空间。

## ( 2 ) 交换区的实现方式

- ✓ 交换分区，或称交换设备
- ✓ 交换文件，用于交换的有固定长度的文件

## ( 3 ) 交换区的分配方式

- ✓ 进程创建时分配：每次换出在同一个地方
- ✓ 换出时分配：首次换出时分配，以后换出在同一个地方；或者每次换出在不同地方。



# 虚拟存储器

---

## 4. 虚拟存储器的特征

逻辑上扩充了内存容量，对于用户程序来说，仿佛内存“无限大”。  
只不过有时慢一点而已。

## 5. 虚拟存储管理的实现方案

分页 + 虚拟存储技术

分段 + 虚拟存储技术

段页式 + 虚拟存储技术



## 4.4 虚拟存储管理

---

### 二、虚拟页式管理（动态页式管理）

#### 1. 基本思想

- ✓ 在进程开始运行之前，不是装入全部页，而是装入部分或0个页，之后根据进程运行的需要，动态装入其它页；
- ✓ 当内存空间已满，而又需要装入新的页时，则根据某种算法淘汰某个页，以便装入新的页。



# 虚拟页式管理

---

## 2. 页表项内容的扩充

除了页号和物理块号外，需要增加下列字段：

- ✓ 有效位（状态）：表示该页是否在内存中
- ✓ 访问位A（访问字段）：记录该页最近是否被访问过
- ✓ 修改位M：表示该页在装入内存后是否被修改过
- ✓ 外存地址：该页在磁盘上的地址



# 虚拟页式管理

---

## 3. 地址变换

虚拟地址 - > 物理地址

类似于静态页式管理。

由于访问的页p可能不在内存中，因而引出以下几个问题：

- ✓ 缺页中断处理
- ✓ 页的换入/换出



# 虚拟页式管理

---

## ( 1 ) 缺页中断 ( Page Fault ) 处理

在地址变换的过程中，当访问页表时，若根据状态位发现所访问的页不在内存，则产生缺页中断。

### 缺页中断处理：

- ① 保护当前进程现场；
- ② 根据页表中给出的外存地址，在外存中找到该页；
- ③ 若内存中无空闲物理块，则选择1页换出；
- ④ 分配一个空闲物理块，将新调入页装入内存；
- ⑤ 修改页表中相应表项的状态位及相应的物理块号，修改空闲物理块表（链）；
- ⑥ 恢复现场。



# 虚拟页式管理

---

## (2) 页的换入/换出

### 1) 页的分配策略：为每个进程分配多少个物理块

#### ✓ 固定分配

为每个进程分配的总物理块数固定，在整个运行期间不变。

#### ✓ 可变分配

先为每个进程分配一定数目的物理块，OS自身维持一个空闲物理块队列。

当发生缺页时，由系统分配一个空闲块，存入调入的页；

当无空闲块时，才会换出。





# 页的换入/换出

---

## 2) 页的置换策略：在什么范围内选择淘汰页

### ✓ 全局置换

从整个内存中选择淘汰页

### ✓ 局部置换

只从缺页进程自身选择淘汰页



# 页的换入/换出

---

## 3) 页的调入策略

### ① 何时调入

#### ✓ 请求调页 ( demand paging )

只有访问的页不在内存中时，才会调入该页。

#### ✓ 预调页 ( prepaging )

一次调入多个连续的页。为什么这样做？

### ② 从何处调入：文件区（可执行文件）、交换区

#### ✓ 全部从交换区调入

进程创建时，全部从文件区拷贝到交换区。

#### ✓ 首次调入从文件区，以后从交换区



# 虚拟页式管理

---

## 4. 页的置换算法 ( Page Replacement Algorithm )

作用：选择换出页

目的：减少缺页率

思路：以过去预测未来



# 页的置换算法

---

## (1) 最优置换算法 (Optimal, OPT)

淘汰以后永不使用的，或者过最长的时间后才会被访问的页。

这是Belady于1966年提出的一种理论上的算法。

显然，采用这种算法会保证最低的缺页率，但无法实现，因为它必须知道“将来”的访问情况。

因此，该算法只是作为衡量其他算法优劣的一个标准。



# 页的置换算法

---

## ( 2 ) 先进先出置换算法 ( First In First Out, FIFO )

淘汰最早进入内存的页。

实现方法：

只需把进程中已调入内存的页，按先后次序链成一个队列即可。

优点：开销较小，实现简单。

缺点：

① 它与进程访问内存的动态特性不相适应；

② 会产生belady现象。即：当分配给进程的物理块数增加时，有时缺页次数反而增加。



# 先进先出置换算法

---

**【例】** 设系统为某进程在内存中固定分配 $m$ 个物理块，初始为空，进程访问页的走向为1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5。采用FIFO算法，当 $m=3$ ， $m=4$ 时，缺页次数分别是多少？

$m=3$ 时，缺页次数：9次

$m=4$ 时，缺页次数：10次

产生Belady现象



# 页的置换算法

---

## ( 3 ) 最近最久未使用算法 ( Least Recently Used, LRU )

**淘汰最近一次访问距当前时间最长的页。**

即淘汰未使用时间最长的页

关键：如何快速地判断出哪一页是最近最久未使用的。

算法较好，但实现代价高。



# LRU 算法

---

## 实现方法：

### 1) 计时器

对于每一页增设一个访问时间计时器

每当某页被访问时，当时的绝对时钟内容被拷贝到对应的访问时间计时器中，这样系统记录了内存中所有页最后一次被访问的时间。

淘汰时，选取访问时间计时器值最小的页。

### 2) 移位寄存器

为内存中的每一页配置一个移位寄存器

当访问某页时，将相应移位寄存器的最高位置1

每隔一定时间，寄存器右移1位

淘汰寄存器值最小的页。





# LRU 算法实现方法

---

## 3) 栈

每次访问某页时，将其页号移到栈顶。使得栈顶始终是最近被访问的页，栈底是最近最久未用的。



# LRU 算法实现方法

4, 7, 0, 7, 1, 0, 1, 2, 1, 2, 6

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
|   |   |   |   |   |   |   | 2 | 1 | 2 | 6 |
|   |   |   |   | 1 | 0 | 1 | 1 | 2 | 1 | 2 |
|   |   | 0 | 7 | 7 | 1 | 0 | 0 | 0 | 0 | 1 |
|   | 7 | 7 | 0 | 0 | 7 | 7 | 7 | 7 | 7 | 0 |
| 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 7 |

当访问页6时，发生缺页，淘汰处于栈底的页4。



# 页的置换算法

---

**【例】** 设系统为进程P固定分配3个物理块，初始为空，进程访问页的顺序为4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5。分别采用OPT、FIFO、LRU置换算法的情况下，计算缺页次数。



# 页的置换算法

---

|     |   |   |   |   |   |   |   |   |   |   |   |   |
|-----|---|---|---|---|---|---|---|---|---|---|---|---|
| OPT | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 4 | 3 | 2 | 1 | 5 |
| 页①  | 4 | 3 | 2 | 1 | 1 | 1 | 5 | 5 | 5 | 2 | 1 | 1 |
| 页②  |   | 4 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 5 | 5 | 5 |
| 页③  |   |   | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
|     | x | x | x | x | √ | √ | x | √ | √ | x | x | √ |

缺页次数：7次



# 页的置换算法

---

|      |   |   |   |   |   |   |   |   |   |   |   |   |
|------|---|---|---|---|---|---|---|---|---|---|---|---|
| FIFO | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 4 | 3 | 2 | 1 | 5 |
| 页①   | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 5 | 5 | 2 | 1 | 1 |
| 页②   |   | 4 | 3 | 2 | 1 | 4 | 3 | 3 | 3 | 5 | 2 | 2 |
| 页③   |   |   | 4 | 3 | 2 | 1 | 4 | 4 | 4 | 3 | 5 | 5 |
|      | x | x | x | x | x | x | x | √ | √ | x | x | √ |

缺页次数：9次



# 页的置换算法

---

|     |   |   |   |   |   |   |   |   |   |   |   |   |
|-----|---|---|---|---|---|---|---|---|---|---|---|---|
| LRU | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 4 | 3 | 2 | 1 | 5 |
| 页①  | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 4 | 3 | 2 | 1 | 5 |
| 页②  |   | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 4 | 3 | 2 | 1 |
| 页③  |   |   | 4 | 3 | 2 | 1 | 4 | 3 | 5 | 4 | 3 | 2 |
|     | x | x | x | x | x | x | x | √ | √ | x | x | x |

缺页次数：10次



# 页的置换算法

---

## (4) 最近未使用算法 (Not Recently Used, NRU)

### 1) 简单的NRU算法

页表中设置一个访问位，当访问某页时，将访问位置1

将内存中的所有页链成一个循环队列

从上次换出页的下一个位置开始扫描

在扫描过程中，将访问位=1的页清0，直到遇到访问位=0的页，淘汰该页，并将指针指向下一页

NRU算法是用得较多的LRU的近似算法

由于该算法循环检查各页的访问位，故称Clock算法。

问题：如果考虑页是否已被修改，选择什么页换出好？



# 页的置换算法

---

## 2) 改进的NRU算法 (改进的Clock算法)

该算法的提出基于如下考虑：

- ① 对于已修改的页，换出时必须重新写回到磁盘上。因此，优先选择未访问过、未修改过的页换出；
- ② 淘汰一个最近未访问的已修改页要比淘汰一个被频繁访问的“干净”页好。





# 改进的NRU算法

---

## 基本方法：

设置1个访问位A，1个修改位M

按照下列次序选择淘汰页：

第1类： $A = 0$ ， $M = 0$ ：未访问，未修改；最佳淘汰页

第2类： $A = 0$ ， $M = 1$ ：未访问，已修改

第3类： $A = 1$ ， $M = 0$ ：已访问，未修改；有可能再次访问

第4类： $A = 1$ ， $M = 1$ ：已访问，已修改



# 改进的NRU算法

---

## 实现算法：

- ① 找第1类页，将遇到的第1个页作为淘汰页；
- ② 若查找1周后未找到第1类页，则寻找第2类页，并将扫描经过的页的访问位清0。将遇到的第1个页作为淘汰页；
- ③ 否则，转①、②，一定能找到淘汰页。



# 页的置换算法

---

## ( 5 ) 最少使用算法 ( Least Frequently Used, LFU )

选择最近访问次数最少的页淘汰。

实现方法：

通常不直接利用计数器来记录页的访问次数，而是采用移位寄存器R。

类似于LRU算法，每次访问某页时，将其移位寄存器的最高位置1。每隔一定时间将移位寄存器右移1位。

这样，在最近一段时间内使用次数最少的页就是移位寄存器各位之和最小的页。

当然，这并不能真正反映出页的访问次数。因为，在每一时间间隔内，只用1位来记录页的访问情况，访问1次和100次是等效的。



# 页的置换算法

---

## (6) 页缓冲算法 (Page Buffering Algorithm)

出发点：与上述算法配合使用，以提高效率。

### 基本方法：

设置2个链表：空闲页链表，已修改页链表。为什么？

- ① 若淘汰页未修改，则直接放入空闲页链表，否则放入已修改页链表；
- ② 当已修改页达到一定数量时，再将其一起写回磁盘，即**成簇写回**，以减少I/O操作的次数。



# 虚拟页式管理

---

## 5. 局部性原理

### (1) 程序的局部性特征

程序在执行过程中的一个较短时期，所执行的指令地址和操作数地址，分别局限于一定区域。表现在时间与空间两方面。

#### ✓ 时间局部性

一条指令被执行了，则在不久的将来它可能再被执行；数据也类似。例如循环。

#### ✓ 空间局部性

若某一存储单元被访问，则在一定时间内，与该存储单元相邻的单元可能被访问。如程序的顺序结构、数组的处理。



# 局部性原理

## (2) 程序结构对性能的影响

**【例】** 将二维整型数组a[256][256]的每个元素初始化为0的C语言程序：

程序1：

```
int a[256][256];  
for (i = 0; i < 256; i++)  
    for (j = 0; j < 256; j++)  
        a[i][j] = 0;
```

程序2：

```
int a[256][256];  
for (j = 0; j < 256; j++)  
    for (i = 0; i < 256; i++)  
        a[i][j] = 0;
```

假定在分页系统中，页的大小为1KB，int类型占4B，设分配给该程序1个物理块。这两个程序哪个好？忽略该程序代码所占的内存空间。

**程序1好。** 因为C数组在内存中以行主次序存放，程序1的缺页次数是256，程序2的缺页次数为 $256 \times 256 = 65536$ 。



# 局部性原理

---

缺页率与程序的行为有关。

因此，在设计程序时，要力求提高程序访问的局部性。

**局部性特征正是虚拟存储技术能有效发挥作用的基础。**



# 虚拟页式管理

---

## 6. 抖动（颠簸）

指页在内存与外存之间频繁换入/换出，以至于调度页所需时间比进程实际运行的时间还多，此时系统效率急剧下降，甚至导致系统崩溃。这种现象称为颠簸或抖动。

### 导致抖动的原因：

- ✓ 页置换算法不合理
- ✓ 分配给进程的物理块数太少





# 虚拟页式管理

---

## 7. 工作集(Working Set)

是由Denning提出并加以推广的，对于虚拟存储管理有着深远的影响。

**一个进程在时刻 $t$ 、参数为 $\Delta$ 的工作集 $W(t, \Delta)$ ，表示该进程在过去的 $\Delta$ 个时间单位中被访问到的页的集合。**

$\Delta$ 称为工作集的窗口大小。

**工作集的内容取决于三个因素：**

- ✓ 访页序列特性
- ✓ 时刻 $t$
- ✓ 观察该进程的时间窗口大小( $\Delta$ )



# 工作集

---

## 工作集的基本思想：

根据程序的局部性原理，一般情况下，进程在一段时间内总是集中访问一些页，这些页称为活动页。

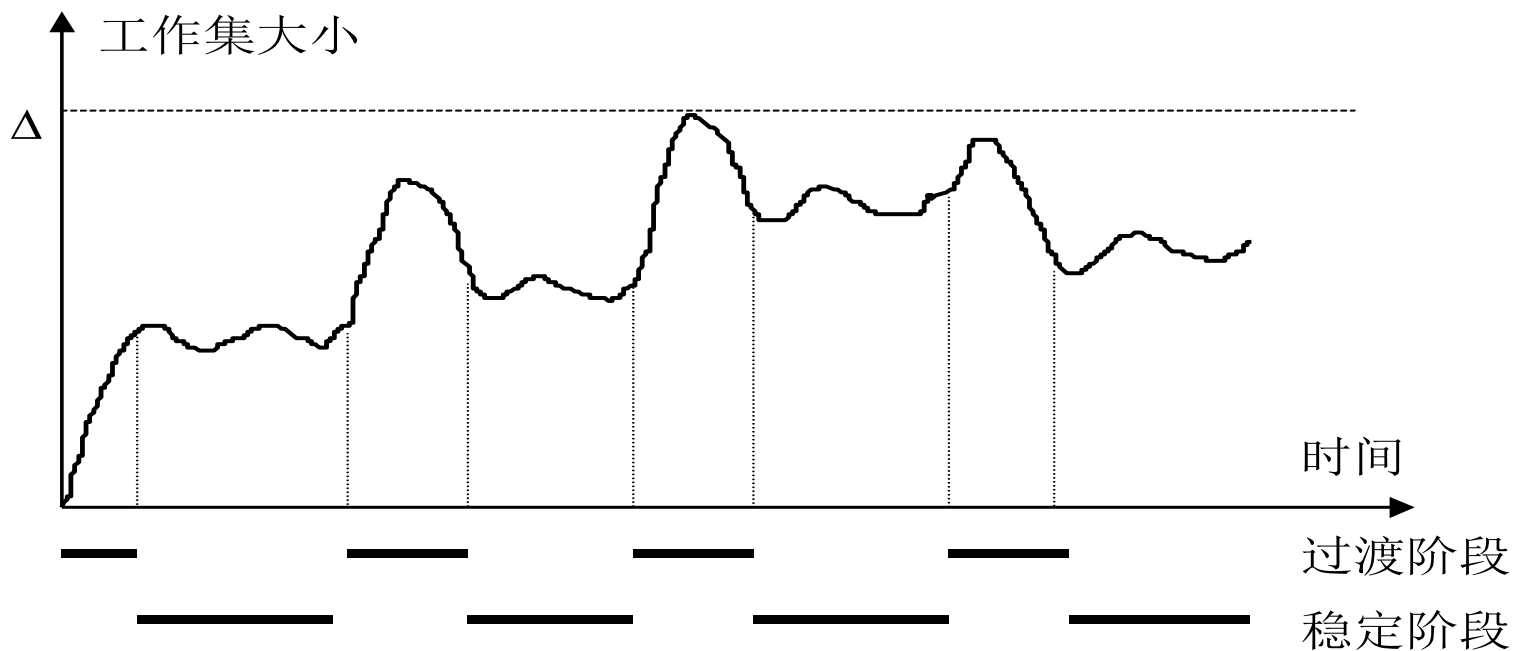
如果分配给一个进程的物理块数太少了，使该进程所需的活动页不能全部装入内存，则进程在运行过程中将频繁发生缺页中断。

如果能为进程提供与活动页数相等的物理块数，则可减少缺页中断次数。

# 工作集

## 工作集大小的变化：

进程开始执行后，随着访问新页逐步建立较稳定的工作集。当内存访问的局部性区域的位置大致稳定时，工作集大小也大致稳定；局部性区域的位置改变时，工作集快速扩张和收缩过渡到下一个稳定值。





# 工作集

---

## 引入工作集概念的意义：

- ✓ 可以指导分配给每个进程多少个物理块，以及如何动态调整各进程的物理块数
- ✓ 通过监视每个进程的工作集，周期性地从一个进程驻留在内存的页的集合中移去那些不在其工作集中的页



# 虚拟页式管理

---

## 8. 影响缺页次数的因素

### (1) 分配给进程的物理块数

一般来说，进程的缺页中断率与进程所占的内存块数成反比。

分配给进程的内存块数太少是导致抖动现象发生的最主要原因。

### (2) 页的大小

缺页中断率与页的大小成反比，但页的大小也不能一味地求大，它一般在0.5KB~4KB之间，是个实验统计值。因为页大时，页表较小，占空间少，查表速度快，缺页中断次数少，但页内碎片较大。页小时，恰恰相反。

### (3) 程序的编写方法

进程的缺页中断率与程序的局部性（包括时间和空间局部性）程度成反比。

用户程序编写的方法不合适可能导致程序运行的时空复杂度高，缺页次数多。

### (4) 页置换算法

算法不合理会引起抖动。



## 4.4 虚拟存储管理

---

### 三、虚拟段式管理

#### 1. 段表项的扩充

- ✓ 段号
- ✓ 段基址、长度
- ✓ 访问位（访问字段）
- ✓ 修改位
- ✓ 有效位（状态）
- ✓ 外存地址
- ✓ 存取方式：读/写/执行
- ✓ 扩充位（增长位）：固定长/可扩充



# 虚拟段式管理

---

## 2. 地址变换

类似于静态段式管理，但可能产生缺段中断。

## 3. 缺段中断处理

检查内存中是否有合适的空闲区

- ①若有，则装入该段，修改有关数据结构，中断返回；
- ②若没有，则检查内存中空闲区的总和是否满足要求，是则应采用紧缩技术，转①；否则，淘汰一（些）段，转①。



# 虚拟段式管理

---

## 4. 段的置换

可用页式管理的置换算法。但一次调入时所需淘汰的段数与段的大小有关。

## 5. 分段的保护措施

### (1) 越界检查

由段表长度、每段长度控制，保证每个进程只访问自己的地址空间。

然而，进程在执行过程中，有时需要扩大分段，如数据段。由于要访问的地址超出原有的段长，所以触发越界中断。操作系统处理越界中断时，首先判断该段的“扩充位”，如可扩充，则增加段的长度；否则按出错处理。

### (2) 访问控制检查

根据段表项的“存取方式”字段进行控制。





## 4.4 虚拟存储管理

---

### 四、虚拟段页式管理

涉及缺段中断、缺页中断。

注意的是，对于缺段中断处理，主要是在内存中建立该段的页表，而非调入完整的一段。



## 4.5 小结

---

### 一、存储器的层次结构及其思想

- ✓ 成本、容量、速度之间的权衡
- ✓ 较小、较贵的存储器由较大、较便宜的慢速存储器作为后援：**虚拟存储器**
- ✓ 降低较大、较便宜的慢速存储器的访问频率：**高速缓存**

### 二、程序的连接与装入

- ✓ 程序编译、连接的功能以及与操作系统（硬件）的关系
- ✓ 程序装入的方式、局限性、对硬件的要求（包括在虚拟存储管理下的装入）

### 三、内存管理有关的重要概念及其意义

- ✓ 局部性原理、页缓冲、交换区
- ✓ 抖动、工作集
- ✓ . . .



## 4.5 小结

---

### 四、典型内存管理机制（连续分配、离散分配；虚拟存储器）

- ✓ 基本原理、逻辑（虚拟）地址结构
- ✓ 数据结构
- ✓ 进程运行时的内存访问过程（包括快表、地址变换、缺页中断等）
- ✓ 对硬件的要求
- ✓ 局限性
- ✓ **注意整体理解**（连续分配能否实现虚拟存储器？能的话如何实现？不能的话为什么？）



## 4.5 小结

---

### 五、针对特定需求，能设计合理的内存管理方案

- ✓ 培养问题分析的意识
- ✓ 方案的基本思路
- ✓ 数据结构（逻辑结构，物理结构的表示）
- ✓ 内存分配与释放算法
- ✓ 进程运行时的内存访问过程
- ✓ 所设计方案的有效性分析

### 六、内存管理与进程管理（包括进程调度）的关系

- ✓ 内存管理是一种调度（资源分配）
- ✓ 与文件系统也有关系
- ✓ 进程从创建到结束需要OS完成的工作



# 本章作业

---

教材《计算机操作系统教程（第4版）》

p135:

5.9, 5.11