



## I 主要内容

- n 内存管理的功能
- n 物理内存管理
  - u 分区存储管理
  - u 覆盖技术
  - u 对换技术
- n 虚拟内存管理
  - u 页式存储管理
  - u 段式存储管理
- n LINUX存储管理

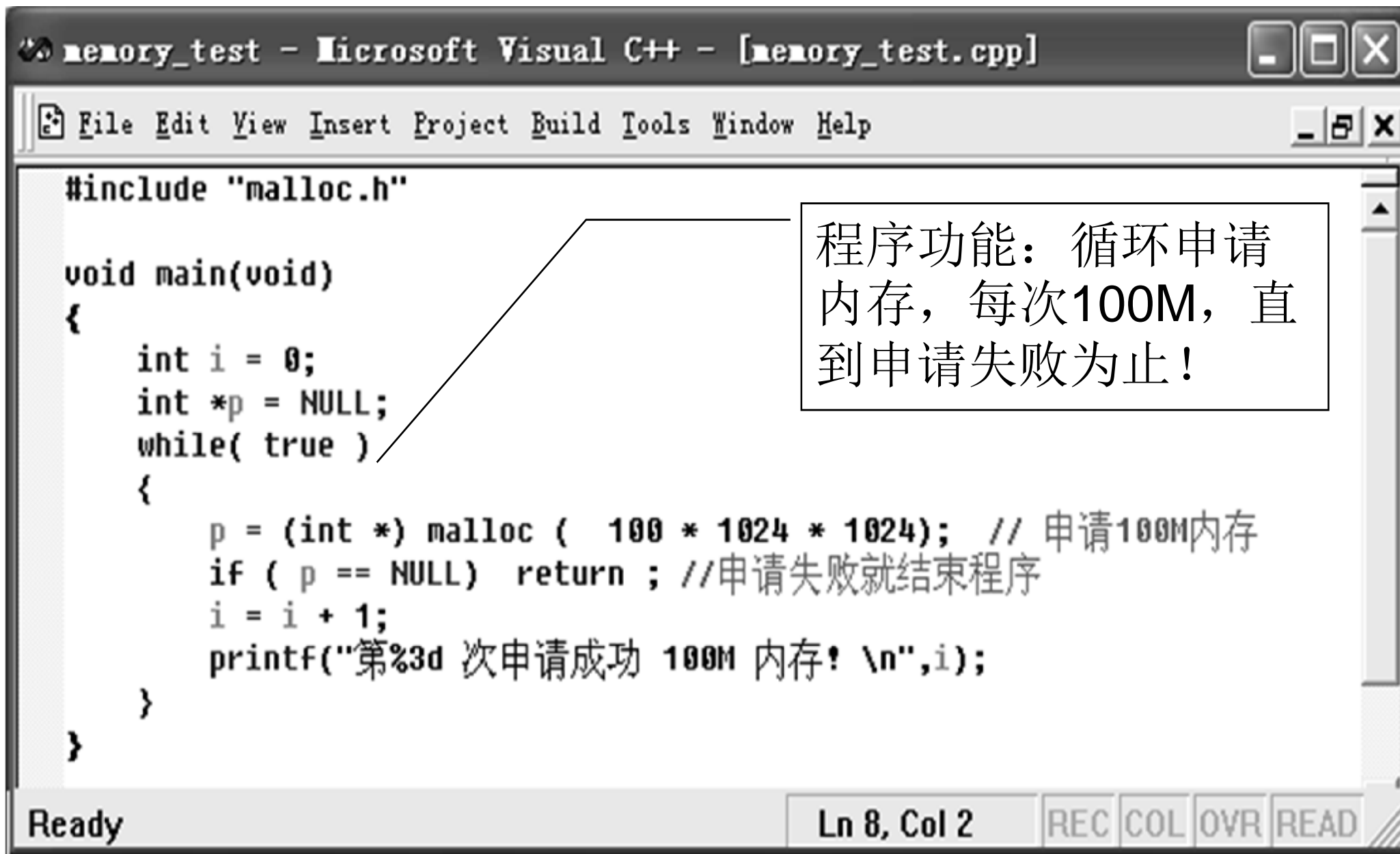
## I 重点

- n 地址映射概念
- n 虚拟存储概念
- n 页式存储管理原理



# 1. 存储管理的功能

# 试图“耗尽”内存的程序



```
memory_test - Microsoft Visual C++ - [memory_test.cpp]
File Edit View Insert Project Build Tools Window Help

#include "malloc.h"

void main(void)
{
    int i = 0;
    int *p = NULL;
    while( true )
    {
        p = (int *) malloc ( 100 * 1024 * 1024); // 申请100M内存
        if ( p == NULL) return ; //申请失败就结束程序
        i = i + 1;
        printf("第%3d 次申请成功 100M 内存! \n",i);
    }
}
```

程序功能：循环申请内存，每次100M，直到申请失败为止！

Ready Ln 8, Col 2 REC COL OVR READ

# 试图“耗尽”内存的程序-运行结果

## I 测试环境

n Win XP

n 1G内存

n 80G硬盘

## I 测试方法

n 运行1个实例

n 运行5个实例

## I 测试结论

n 内存足够“大”

n > 实际内存



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The prompt is at "E:\《操作系统原理》>". The user has entered "memory\_test.exe". The program outputs a list of 19 lines, each showing a successful memory allocation of 100M. The output is as follows:

```
E:\《操作系统原理》>memory_test.exe
第1次申请成功 100M 内存!
第2次申请成功 100M 内存!
第3次申请成功 100M 内存!
第4次申请成功 100M 内存!
第5次申请成功 100M 内存!
第6次申请成功 100M 内存!
第7次申请成功 100M 内存!
第8次申请成功 100M 内存!
第9次申请成功 100M 内存!
第10次申请成功 100M 内存!
第11次申请成功 100M 内存!
第12次申请成功 100M 内存!
第13次申请成功 100M 内存!
第14次申请成功 100M 内存!
第15次申请成功 100M 内存!
第16次申请成功 100M 内存!
第17次申请成功 100M 内存!
第18次申请成功 100M 内存!
第19次申请成功 100M 内存!
E:\《操作系统原理》>
```

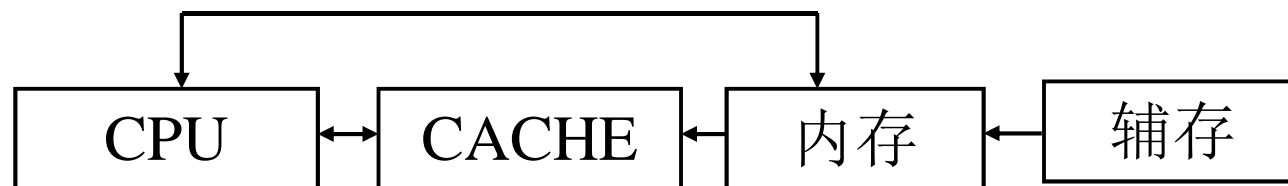
1个实例:  
申请内存: 1.85G  
( = 19 X 100 M )

5个实例:  
申请内存: 9.2G  
( = 1.85 G X 5 )

## I 存储器功能需求

- n 容量足够大
- n 速度足够快
- n 信息永久保存

## I 实际存储器体系



- n 三级存储体系

n Cache(快,小,贵) + 内存 (适中) + 辅存 (慢,大,廉)

- n 基本原理:

- u 当内存太小不够用时，用“辅存”来支援内存。
- u 暂时不运行的模块换出到辅存上，必要时再换入回内存。

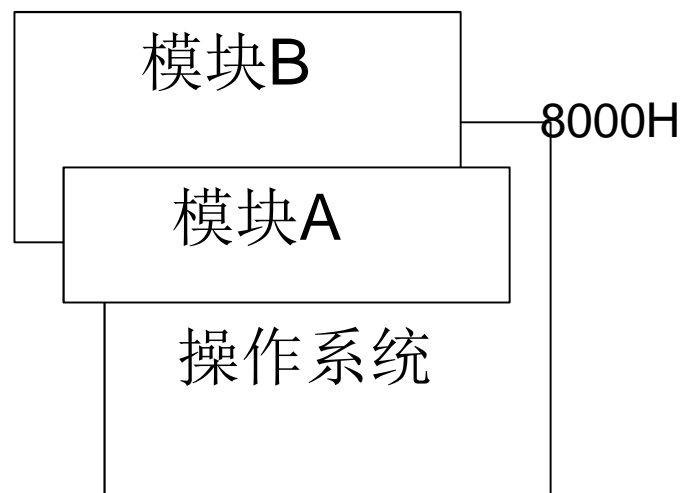
# 换出与换入的讨论

## I 模块A换入到硬盘

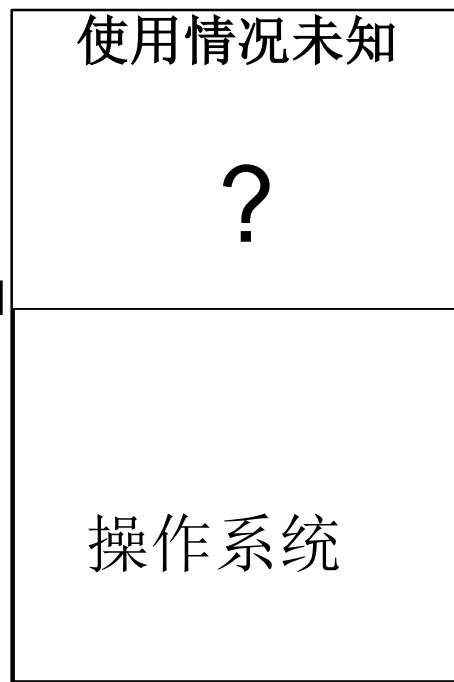
n 放到什么位置

u 原来位置?

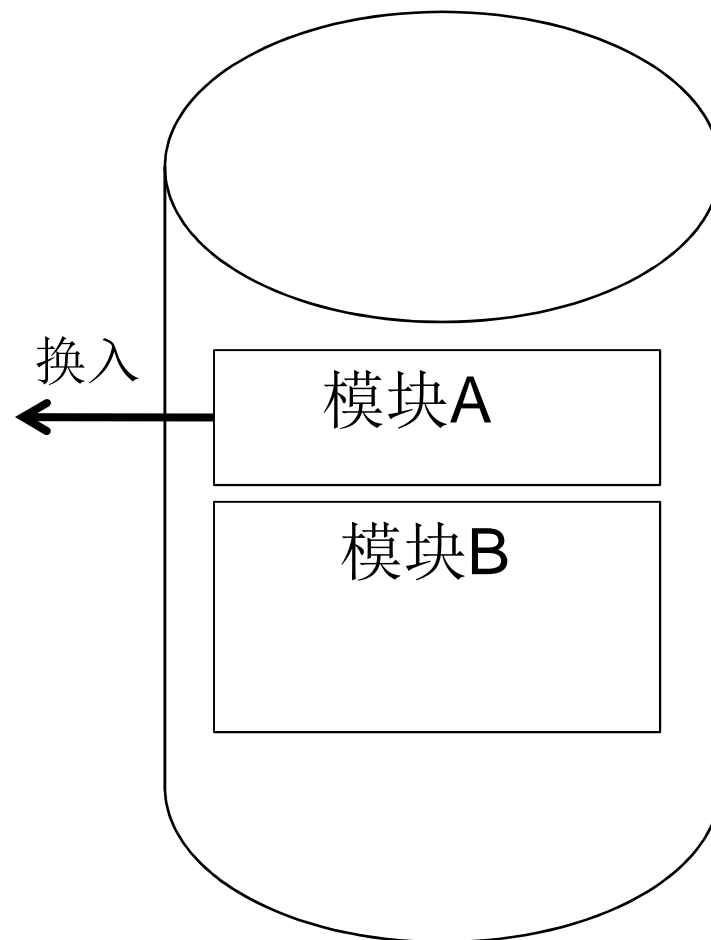
u 任一位置?



进程/模块



内存



硬盘

l 进程或模块换出/换入需要解决的问题：访问地址有变  
n地址重定位（地址重映射）

u 辅存上的映像**换入**到内存时，需要重新确定放置地点，且把程序中访问的地址更新为**新地址**。



# I 存储器功能需求

- n 容量足够大

- n 速度足够快

- n 信息永久保存

- n 多道程序并行

# I 多道程序并行带来的问题

n 共享

u 代码和数据共享，节省内存

n 保护

u 不允许内存中的程序相互间非法访问

# I 存储管理的功能

**n1)** 地址映射

**n2)** 虚拟存储

**n3)** 内存分配

**n4)** 存储保护

# 存储管理的功能：1) 地址映射

## I 定义

**n** 把程序中的地址（虚拟地址,虚地址,逻辑地址）变换成真实的内存地址（实地址,物理地址）的过程。

**n** 地址重定位，地址重映射

**n** 源程序—逻辑地址—物理地址

## I 方式

**n** 固定地址映射

**n** 静态地址映射

**n** 动态地址映射

# 固定地址映射

## I 定义

**n**编程或编译时确定逻辑地址和物理地址映射关系。

## I 特点

**n**程序加载时必须放在指定的内存区域。

**n**容易产生地址冲突，运行失败。

# 静态地址映射

## I 定义

**n** 程序装入时由操作系统完成逻辑地址到物理地址的映射。

## I 静态地址映射

**n**逻辑地址: VA(Virtual Addr. Register )

**n**装入基址: BA(Base Addr. Register)

**n**物理地址: MA(Memory Addr. Register)

$$\mathbf{n} \text{MA} = \text{BA} + \text{VA}$$

## I 静态地址映射

n 逻辑地址: VA(Virtual Addr. Register )

n 装入基址: BA(Base Addr. Register)

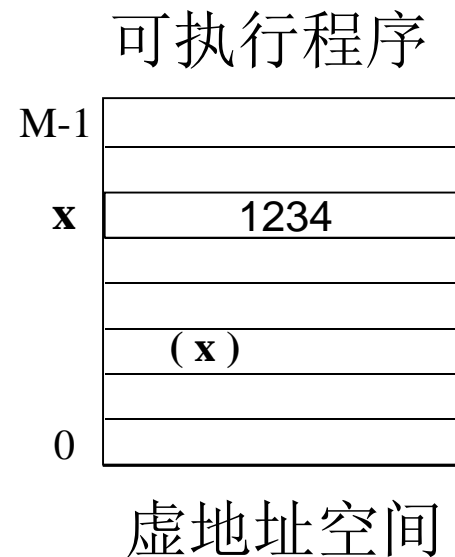
n 物理地址: MA(Memory Addr. Register)

$$n MA = BA + VA$$

## I 例子

n 逻辑地址:  $VA = X$

$$(X) = 1234$$





## 静态地址映射

**n**逻辑地址: VA(Virtual Addr. Register )

**n**装入基址: BA(Base Addr. Register)

**n物理地址: MA(Memory Addr. Register)**

$$\mathbf{n}MA = BA + VA$$

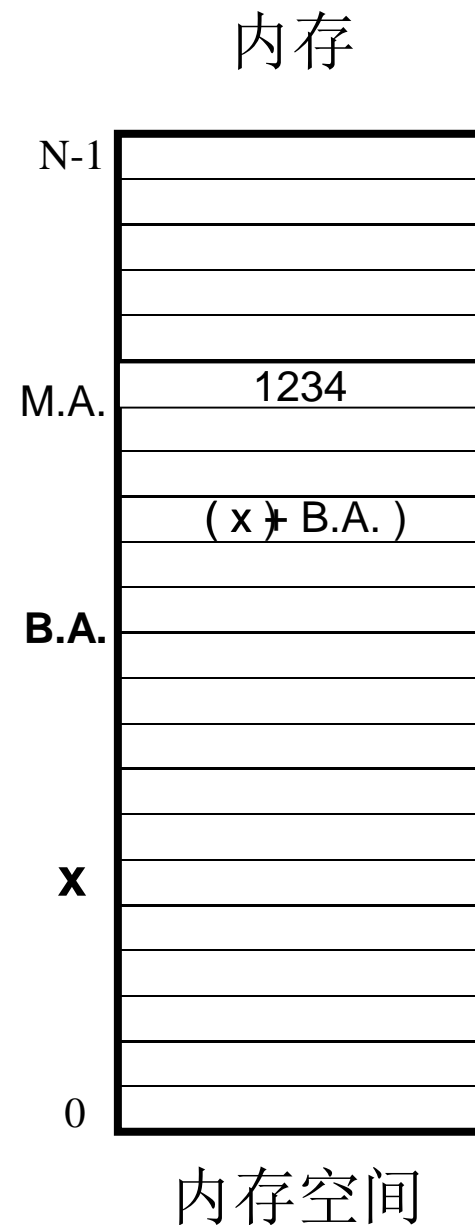
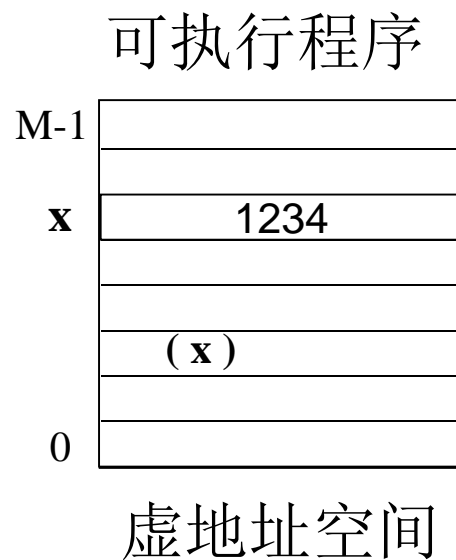
## ■ 例子

**n**逻辑地址:  $VA = X$

$$(X) = 1234$$

**n**装入基址: BA

$$\mathbf{n}MA = X + BA$$



## I 特点

- n 程序运行之前确定映射关系

- n 程序装入后不能移动

  - u 如果移动必须放回原来位置

- n 程序占用连续的内存空间

# 动态地址映射

## I 定义

**n**在程序执行过程中把逻辑地址转换为物理地址。

**u**例如：MOV AX, [500]；访问500单元时执行地址转换

## I 映射过程

**n**逻辑地址：VA(Virtual Addr. Register )

**n**装入基址：BA(Base Addr. Register)

**n**物理地址：MA(Memory Addr. Register)

**n** $MA = BA + VA$

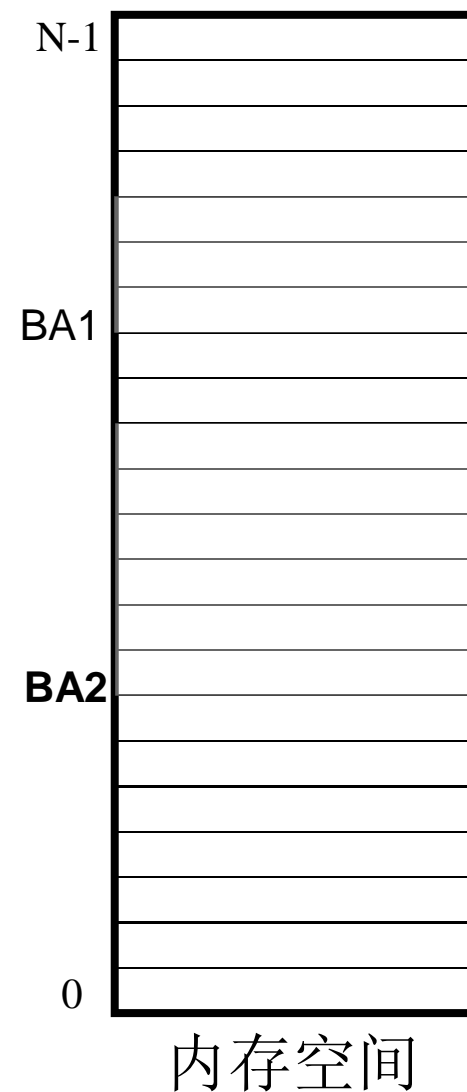
**u**注意：如果程序有移动，BA可能会有改变，自动计算新的MA。

## I 特点

- n 程序占用的内存空间可动态变化
  - u 要求及时更新基址
- n 程序不要求占用连续的内存空间
  - u 每段放置的基址系统应该知道
- n 便于多个进程共享代码
  - u 共享代码作为独立的一段存放
- n 硬件改变后不需要重新编译程序

## I 缺点

- n 硬件支持（MMU：内存管理单元）
- n 硬件时延
- n 软件复杂



## 存储管理的功能：2) 虚拟存储

### I 解决的问题

- n 1) 程序过大或过多时，内存不够，不能运行；

- n 2) 多个程序并发时地址冲突，不能运行；

### I 问题1的解决方法（虚拟存储的基本原理）

- n 借助辅存在逻辑上扩充内存，解决内存不足

- n 把进程当前正在运行的部分装入内存（迁入），把当前不运行的部分暂时存放在辅存上（迁出），尽量腾出足够的内存供进程正常运行。

  - u 辅存存放的部分当需要运行时才临时按需调入内存。

  - u 进程不运行的部分往往占大部分，尤其是大进程。

# I 程序局部性原理

## n 时间局部性

- u 一条指令或数据，会在较短时间内被重复访问

- p 例如：循环语句

## n 空间局部性

- u 任一内存单元及其邻近单元会在短时间内被集中访问

- u 短时间内，CPU对内存的访问往往集中在一个较小区域内

- u 例如：表，数组的操作

## n 结论：

- u 程序在一个有限的时间段内访问的代码和数据往往集中在有限的地址范围内。因此，一般情况下，把程序的一部分装入内存在较大概率上也足够让其运行。

# 存储管理的功能：2) 虚拟存储——

## I 解决的问题

- n 1) 程序过大或过多时，内存不够，不能运行；
- n 2) 多个程序并发时地址冲突，不能运行；

## I 问题2的解决方法（虚拟存储的定义）

- n 编程时不受内存容量和结构限制，认为内存是理想存储器（虚拟存储器）

### u 特点

- p 线性地址
- p 封闭空间
- p 容量足够大： $2^{32}$ BYTE
- p 虚拟地址和虚拟地址空间

### u 工作原理

- p 运行时把虚拟地址转化为具体的物理地址：地址映射功能
- p 不同程序中的同一虚拟地址转化为不同的物理地址。
- p 内存管理单元：MMU: Memory Management Unit

### u 作用：实现虚拟地址和物理地址分离

## I 实现虚拟存储的前提

- n 足够的辅存
- n 适当容量的内存
- n 地址变换机构

## I 虚拟存储的应用

- n 页式虚拟存储
- n 段式虚拟存储



## 存储管理的功能：3) 内存分配功能

- I 为程序运行分配足够的内存空间

- I 需要解决的问题

  - n 放置策略

    - u 程序调入内存时将其放置在哪个/哪些内存区

  - n 调入策略

    - u 何时把要运行的代码和要访问的数据调入内存？

  - n 淘汰策略

    - u 内存空间不够时，迁出（/淘汰）哪些代码或数据以腾出内存空间。

## 存储管理的功能：4) 存储保护功能

I 保证在内存中的多道程序只能在给定的存储区域内活动并互不干扰。

- n 防止访问越界

- n 防止访问越权

I 方法：界址寄存器

- n 在CPU中设置一对下限寄存器和上限寄存器存放程序在内存中的下限地址和上限地址

- u 程序访问内存时硬件自动将目的地址与下限寄存器和上限寄存器中存放的地址界限比较，判断是否越界。

- n 基址寄存器和限长寄存器



## 2. 物理内存管理

# I 物理内存管理方法

- n 单一区存储管理（不分区存储管理）

- n 分区存储管理

- n 内存覆盖技术

- n 内存交换技术

# 单一区存储管理（不分区存储管理）

## I 定义

**n**用户区不分区，完全被一个程序占用。

**u**例如：DOS

## I 优点

**n**简单，不需复杂硬件支持，适于单用户单任务OS

## I 缺点

**n**程序运行占用整个内存，即使小程序也是如此

**u**内存浪费，利用率低

# 分区存储管理

## I 定义

- n 把用户区内存划分为若干大小不等的分区，供不同程序使用。
- n 最简单的存储管理, 适合单用户单任务系统。

## I 分类

- n 固定分区
- n 动态分区

# 固定分区

## I 定义

**n**把内存固定地划分为若干个大小不等的分区供各个程序使用。每个分区的大小和位置都固定，系统运行期间不再重新划分。

**n**分区表

**u**记录分区的位置、大小和使用标志

# 固定分区的例子

## I 4个分区的例子

n3个程序在占用

n分区表

分区表

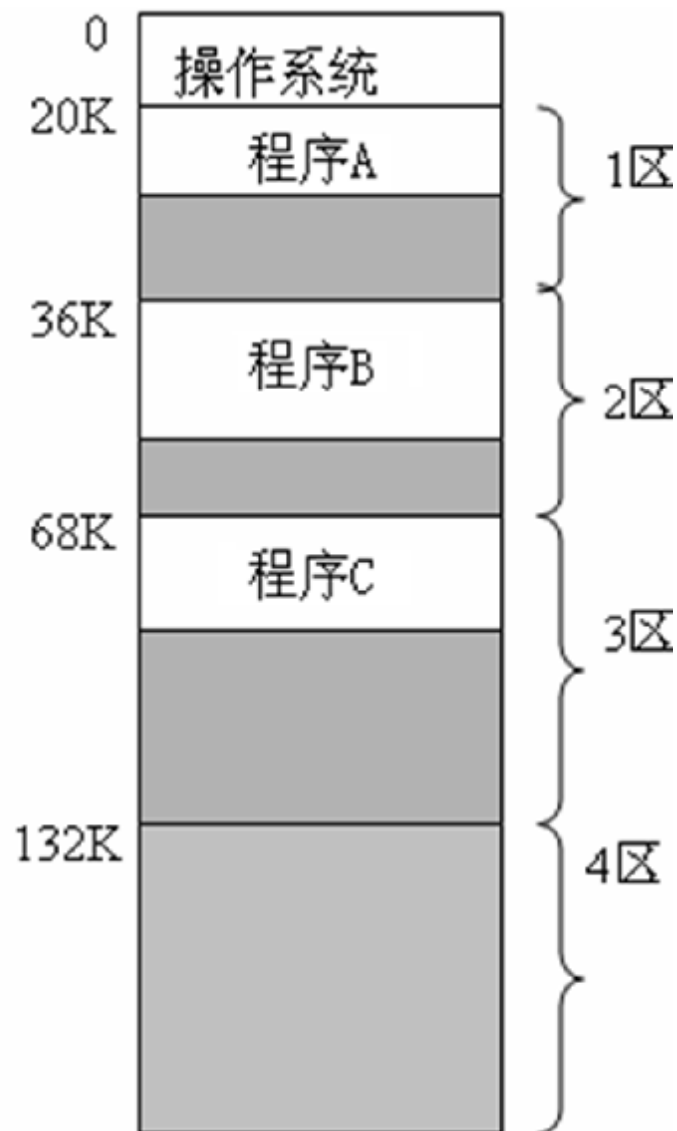
区号	大小	起址	标志
1	16K	20K	已分配
2	32K	36K	已分配
3	64K	68K	已分配
4	124K	132K	未分配



被占用的空间



未占用的空间





# 固定分区

## I 使用特点

**n**在程序装入前，内存已被分区，不再改变。

**n**每个分区大小可能不同，以适应不同大小的程序。

**n**系统维护分区表，说明分区大小、地址和使用标志

**p**例如：IBM的OS/360采用了固定分区方法。

**p**具有固定任务数的多道程序系统

# 固定分区

## I 固定分区的性能

- n 当程序比所在分区小时，浪费内存

- n 当一个程序比最大分区大时，无法装入运行

## I 建议措施

- u 根据分区表安排程序的装入顺序，使得每个程序都能找到合适的分区运行。

- u 当程序的大小、个数、装入顺序等都固定时，内存使用效率很高。

# 动态分区

## I 定义

**n**在程序装入时创建分区，使分区的大小刚好与程序的大小相等。

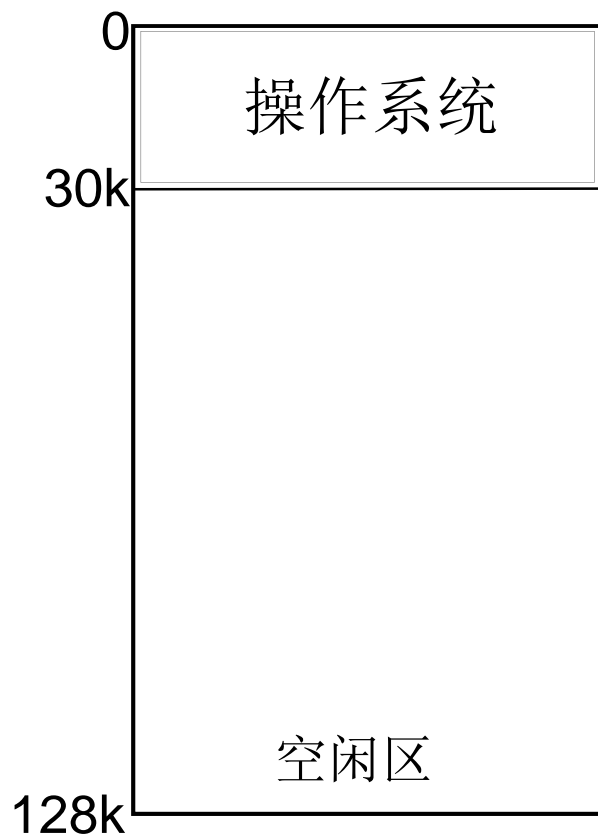
**p**解决固定分区浪费内存和大程序不能运行的问题。

## **p**特点

**p**分区动态建立

# 动态分区例子

I 程序1 (20K) ; 程序2 (16k) ; 程序3 (24k) ; 程序4 (30K)



内存初始状态



动态划分分区



程序1和3运行完;  
程序撤出收回内存

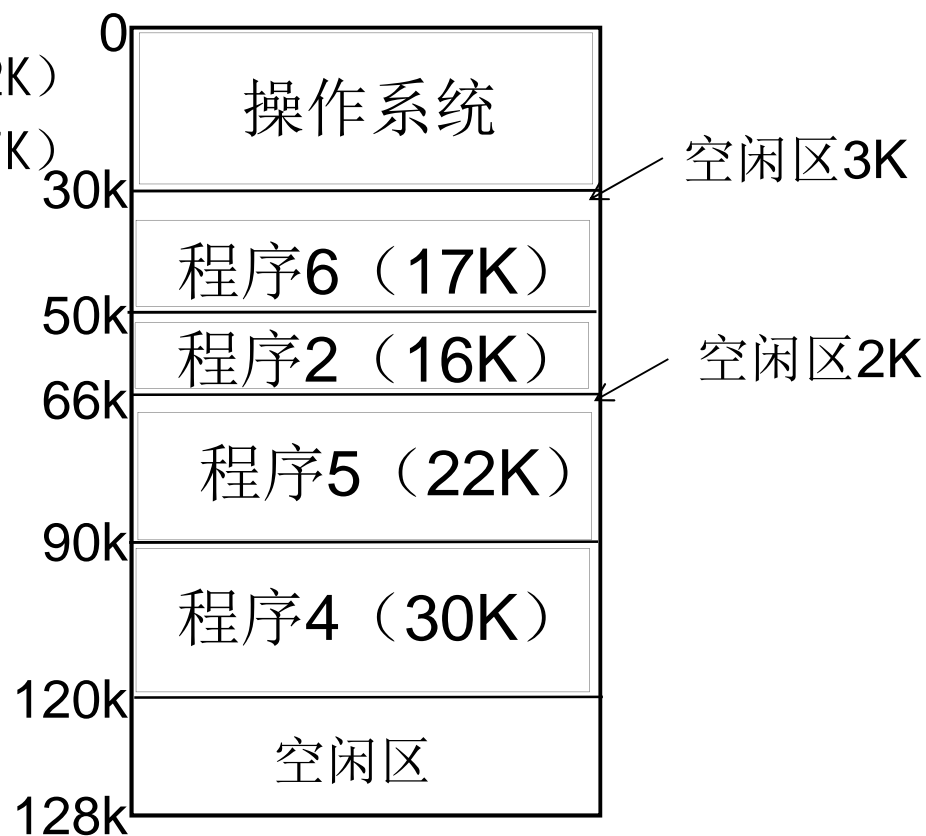
# 动态分区例子

I 程序1 (20K) ; 程序2 (16k) ; 程序3 (24k) ; 程序4 (30K)



程序1和3运行完;  
程序撤出收回内存

- 程序5 (22K)
- 程序6 (17K)



内存碎片: 过小的空闲区, 难实际利用

# 动态分区

## I 特点

- n分区的个数和大小均可变

- n存在内存碎片

## I 动态分区需要解决的问题

- n分区的分配？

- n分区的选择？

- n分区的回收？

- n解决内存碎片问题？

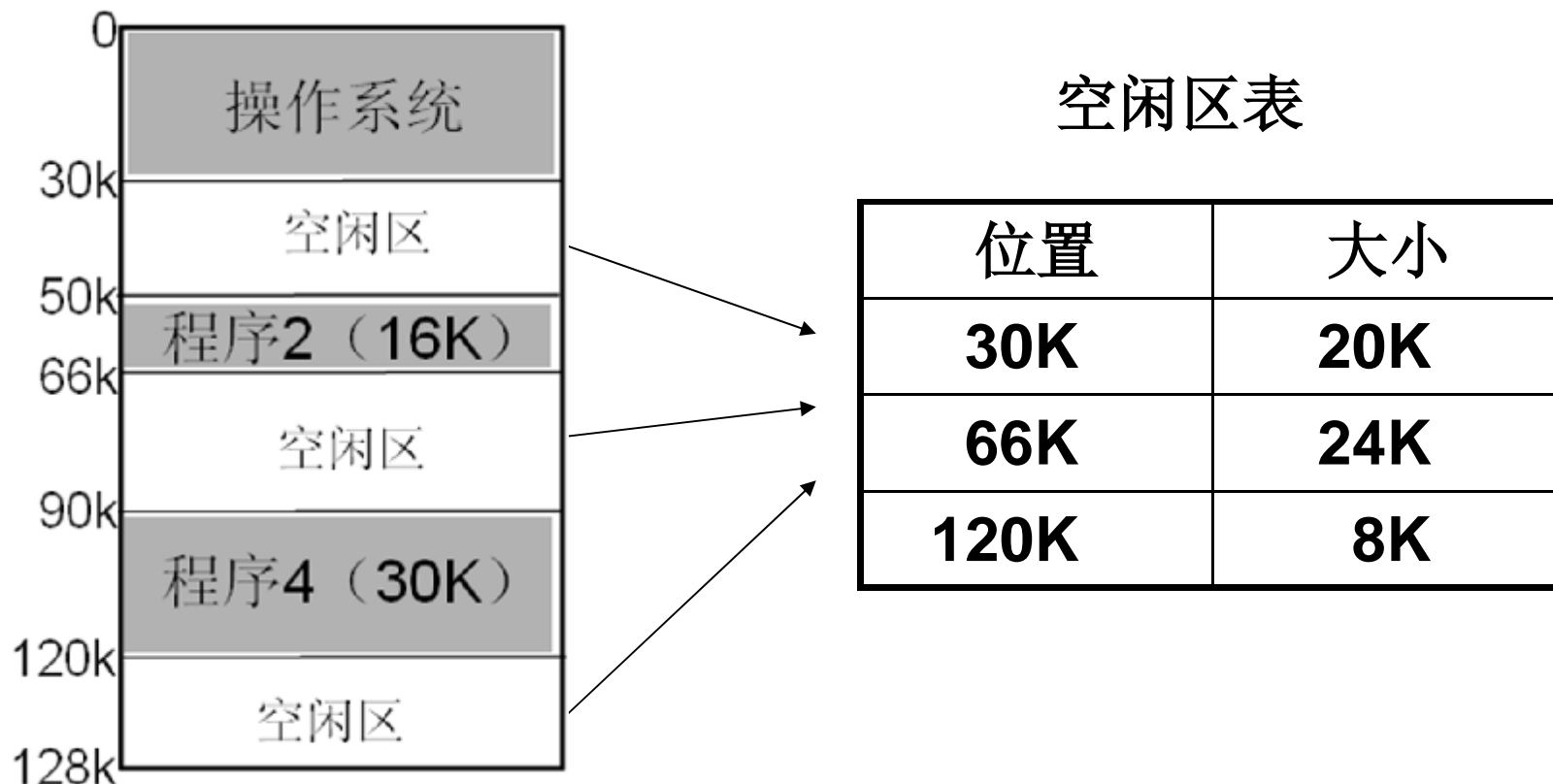
# 分区的分配

## I 功能

n 在所有空闲区中寻找一个空闲区，分配给用户使用。

n 基本要求：空闲区的大小应满足用户要求。

## I 空闲区表：描述内存空闲区的位置和大小的数据结构



# 分区的分配

## I 过程(假定用户要求的空间大小为SIZE)

- n (1) 从空闲区表的第1个区开始, 寻找 $\geq$ SIZE的空闲区
- n (2) 找到后从分区中分割出大小为SIZE的部分给用户使用。
- n (3) 分割后的剩余部分作为空闲区仍然登记在空闲区表中。
- n 注意: 分割空闲区时一般从底部分割。

空闲区表

位置	大小
30K	20K
66K	24K
120K	8K



# 分区的选择——放置策略

## I 放置策略

- n 选择空闲分区的策略

## I 空闲区表的排序原则

- n 按空闲区位置（首址）递增排序

- n 按空闲区位置（首址）递减排序

- n 按空闲区大小的递增排序

- n 按空闲区大小的递减排序

## I 常用的放置策略

- n 首次匹配（首次适应算法）

- n 最佳匹配（最佳适应算法）

- n 最坏匹配（最坏适应算法）

空闲区表

位置	大小
66K	24K
30K	20K
120K	8K

大小的递减

# 首次适应法

空闲区表

位置	大小
30K	20K
66K	24K
120K	8K

## I 前提

n 空闲区表按首址递增排序

## I 算法

n 从空闲区表的第1个分区开始查找，直到找到第一个满足用户要求的分区为止。

n 优点尽可能地先利用低地址空间，保证高地址空间有较大的空闲区，当大程序需要较大分区时，满足的可能性很大。

# 最佳适应法

## I 前提

n 空闲区表按大小递增排序

## I 算法

n 从空闲区表的第1个分区开始查找，直到找到第一个满足用户要求的分区为止。

## I 优点

n 选中分区是满足要求的最小的空闲区，尽量保留较大空闲区，当大程序需要较大分区时，满足的可能性很大。

## I 缺点

n 容易出现内存碎片

位置	大小
120K	8K
30K	20K
66K	24K

# 最坏适应法

## I 前提

**n**空闲区表按大小递减排序

## I 算法

**n**从空闲区表的第1个分区开始查找，直到找到第一个满足用户要求的分区为止。

## I 优点

**n**最大的空闲区分割后的剩下部分还可能相当大，还能装下较大的程序。

## I 特点

**n**仅作一次查找就可找到所要分区。

位置	大小
66K	24K
30K	20K
120K	8K

# 分区的回收

## I 功能

**n**回收程序结束后所释放的分区（**释放区**），将其**适当处理**后登记到空闲区表中，以便再分配。

**n**要考虑**释放区**与现有**空闲区**是否相邻？

## I 回收算法

**n**若**释放区**与**空闲区**不相邻，则把**释放区**直接插入**空闲区表**。

**n**若**释放区**与**空闲区**相邻，则把**释放区**和**空闲区**合并后作为新的更大的**空闲区**插入**空闲区表**。

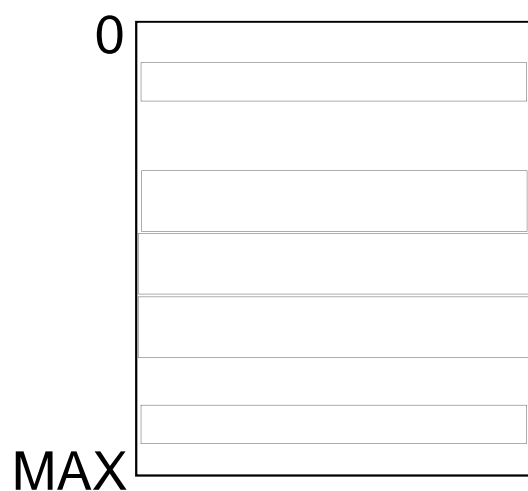
位置	大小
<b>30K</b>	<b>20K</b>
<b>66K</b>	<b>24K</b>
<b>120K</b>	<b>8K</b>

# I 具体的回收过程

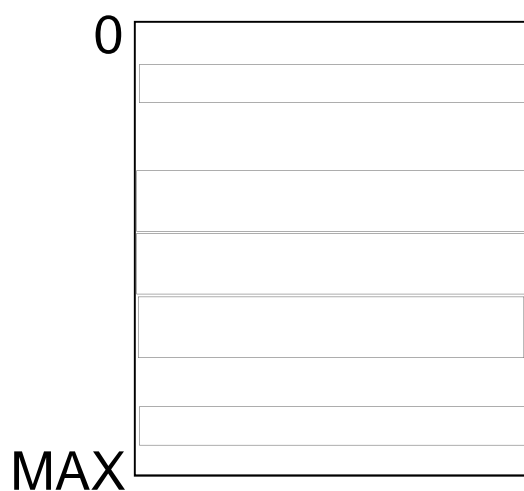
**n**  当前的空闲区

**n**  释放区

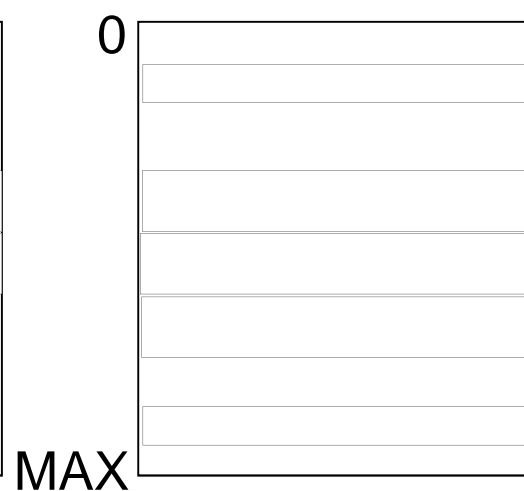
**n**  占用区



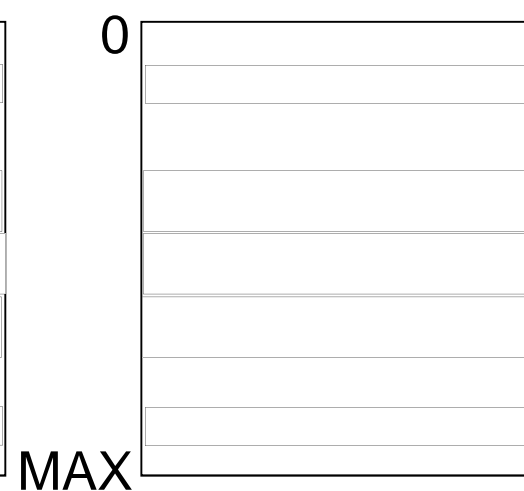
合并,  
更新大小



合并,  
更新位置和大小



合并,  
先删除后更新大小



插入

# 碎片问题

## I 解决碎片的办法

### n 规定门限值

- u 分割空闲区时，若剩余部分小于门限值，则此空闲区不进行分割，而是全部分配给用户。

### n 内存拼接技术

- u 定期清理存储空间，将所有空闲区集中一起。

- u 存储器紧缩技术

## I 拼接的时机

- n 分区回收的时候

  - u 拼接频率过大，系统开销大

- n 系统找不到足够大的分区时

  - u 空闲区的管理复杂

## I 拼接技术的缺点

- n 消耗系统资源；

- n 离线拼接；

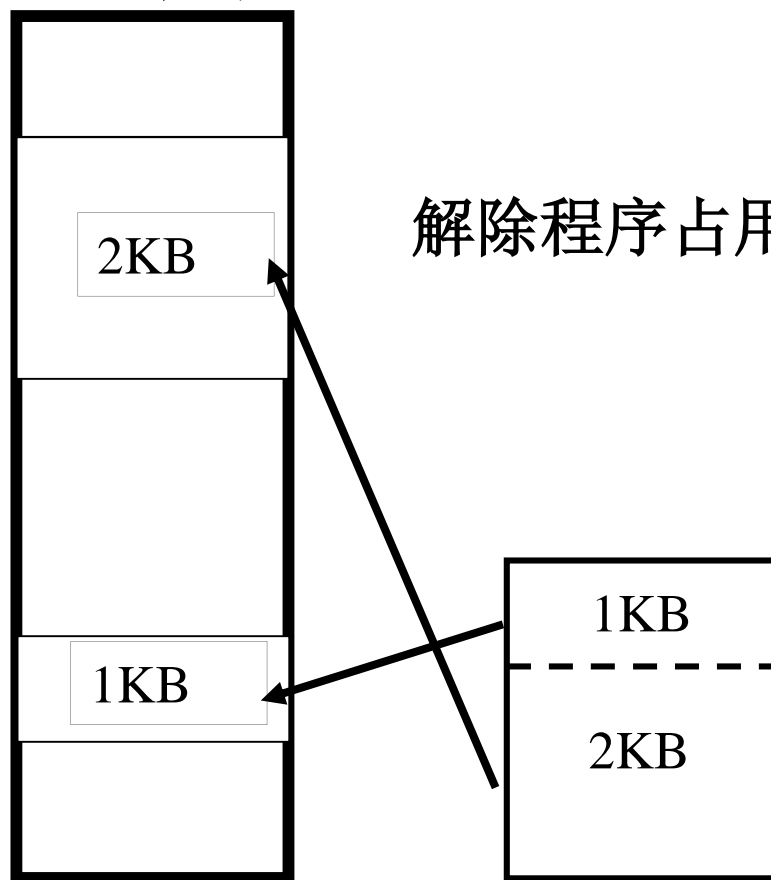
- n 重新定义作业



## I 解决碎片的办法（续）

- 把程序分成几个部分装入不同的分区中，化整为零，以便充分利用小的碎片。

内存空间有  
两块空闲区  
1KB + 2KB



解除程序占用连续内存的限制。

3KB的进程

# 覆盖——Overlay

## I 目的

**n** 在较小的内存空间中运行较大的程序

## I 内存分区

**n** 常驻区：被某段单独占用的区域，可划分多个

**n** 覆盖区：能被多段重复共用（覆盖）的区域，可划分多个

OS
覆盖区2 40K
覆盖区1 50K
常驻区 20K

用户内存（110K）

# 覆盖——Overlay

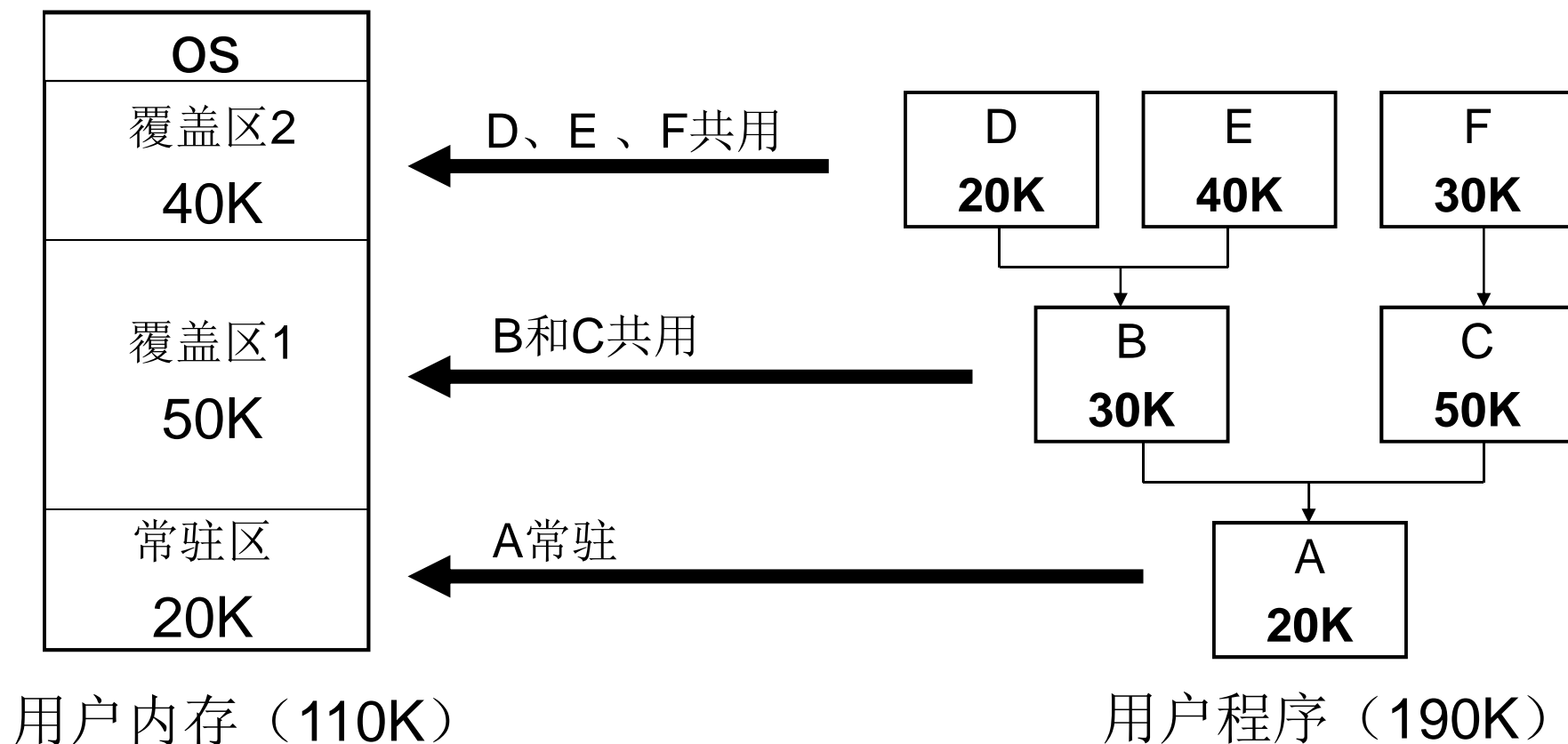
## I 工作原理

- n 程序分成若干代码段或数据段
- n 将程序常用的段装入常驻区的内存；（核心段）
- n 目前正运行的的段处于覆盖区中；
- n 将目前不用的段或用过的段放在硬盘的特殊区域中(覆盖文件)；
- n 临时要用的段从硬盘装入覆盖区的内存（覆盖不再用的内容）；
  - u 意义盖区，减少程序对内存的需求

## I 覆盖的例子

**n** 内存（110K）：一个常驻区，两个覆盖区

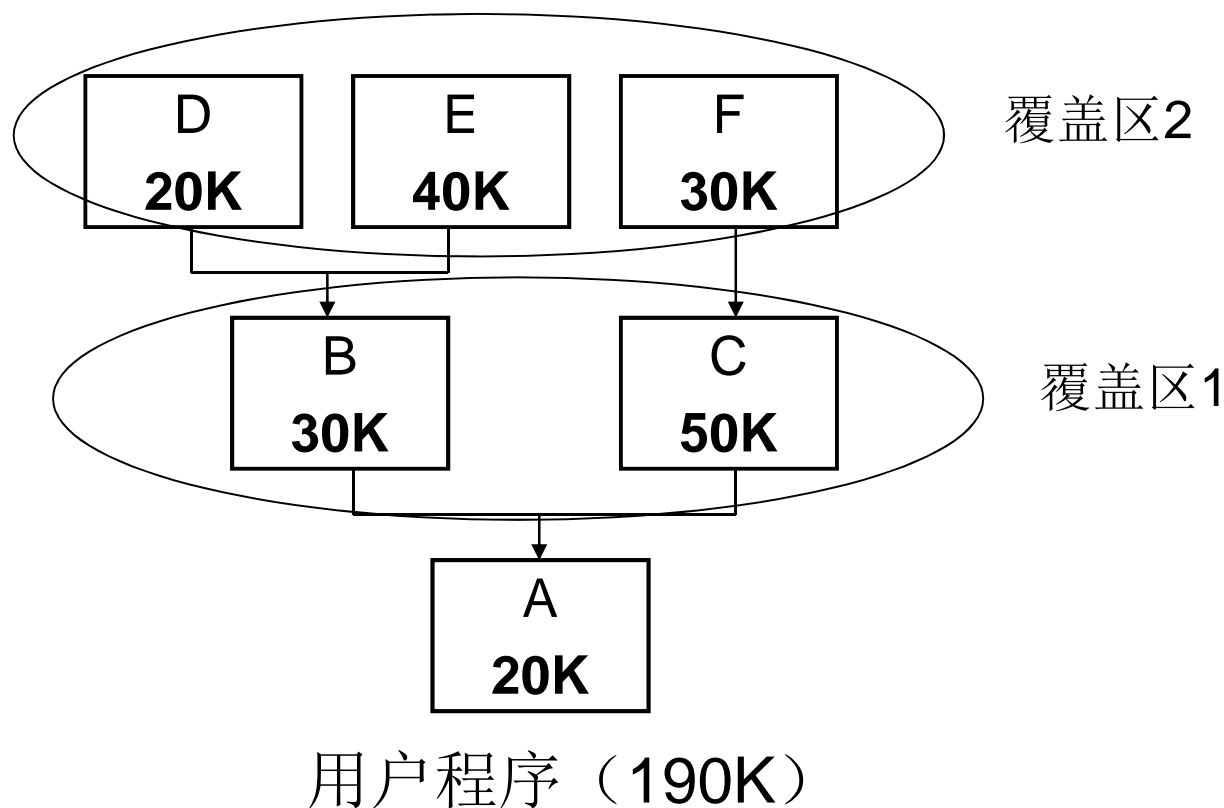
**n** 程序（190K）：多个模块（段）



## I 覆盖的缺点

**n**编程复杂：程序员划分程序模块并确定覆盖关系。

**n**程序执行时间长：从外存装入内存耗时



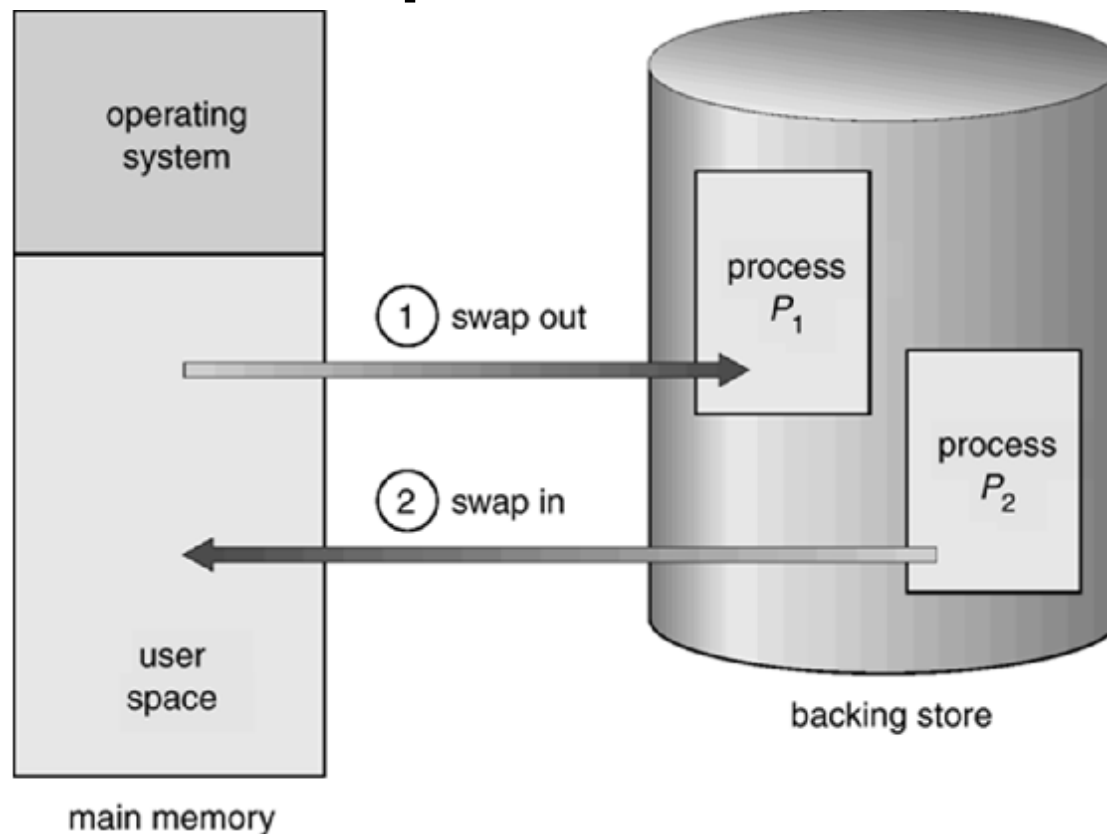
# 对换技术——Swapping

## I 原理

n 当内存不够时，把进程写到磁盘上（换出，**Swap Out**），以便腾空内存。当进程要运行时，再把它重新写回到内存（换入，**Swap In**）。

## I 优点

n 增加进程并发数；  
n 不考虑程序结构。



## I 对换技术的缺点

- n 换入和换出增加**CPU**开销；
- n 对换单位太大（整个进程）。

## I 需要考虑的问题

- n 程序换入时的地址重定位
- n 减少对换传送的信息量
- n 外存对换空间的管理方法
- n 采用交换技术的**OS**
  - n **UNIX, Linux, Windows 3.1**



### 3. 虚拟内存管理



# 虚拟内存的概念

I 虚拟内存是面向用户的虚拟封闭存储空间。

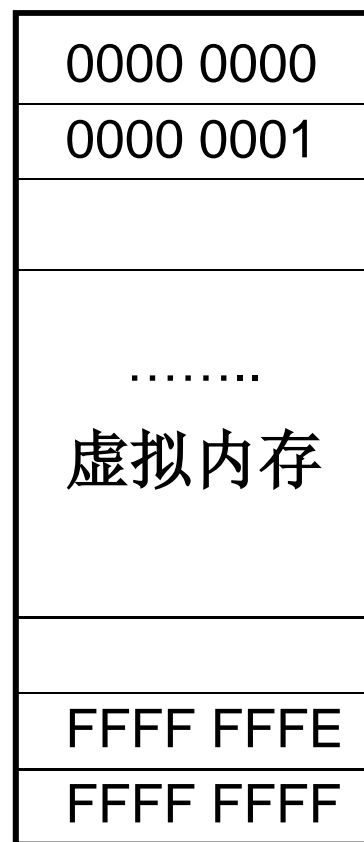
n 线性地址空间。

n 容量**4G =  $2^{32}$  Byte**

n 封闭空间（进程空间）

n 和物理地址分离（地址无冲突）

n 程序员编程时使用线性虚拟地址



## I 虚拟内存管理的目标

- n 使得大的程序能在较小的内存中运行；
- n 使得多个程序能在较小的内存中运行（/能容纳下）；
- n 使得多个程序并发运行时地址不冲突（/方便,高效）；
- n 使得内存利用效率高：无碎片,共享方便

## I 程序运行的局部性

- n 程序在一个有限的时间段内访问的代码和数据往往集中在有限的地址范围内。
- n 把程序一部分装入内存在较大概率上也足够让其运行一小段时间。

# 虚拟内存原理/技术的用处

- I 1、应用层程序：编写一些较底层或有特殊功能的应用程序时，往往需要使用虚拟内存相关技术；
  - n 例：（防）木马和病毒会使用虚拟内存技术进行代码注入操作。
- I 2、内核层程序：优化或裁减**OS**内核（尤其是开源嵌入式**OS**）
  - n 内存管理优化
  - n 实时性优化

# I 典型虚拟内存管理方式

- n 页式虚拟存储管理

- n 段式虚拟存储管理

- n 段页式虚拟存储管理

# 页式虚拟存储管理

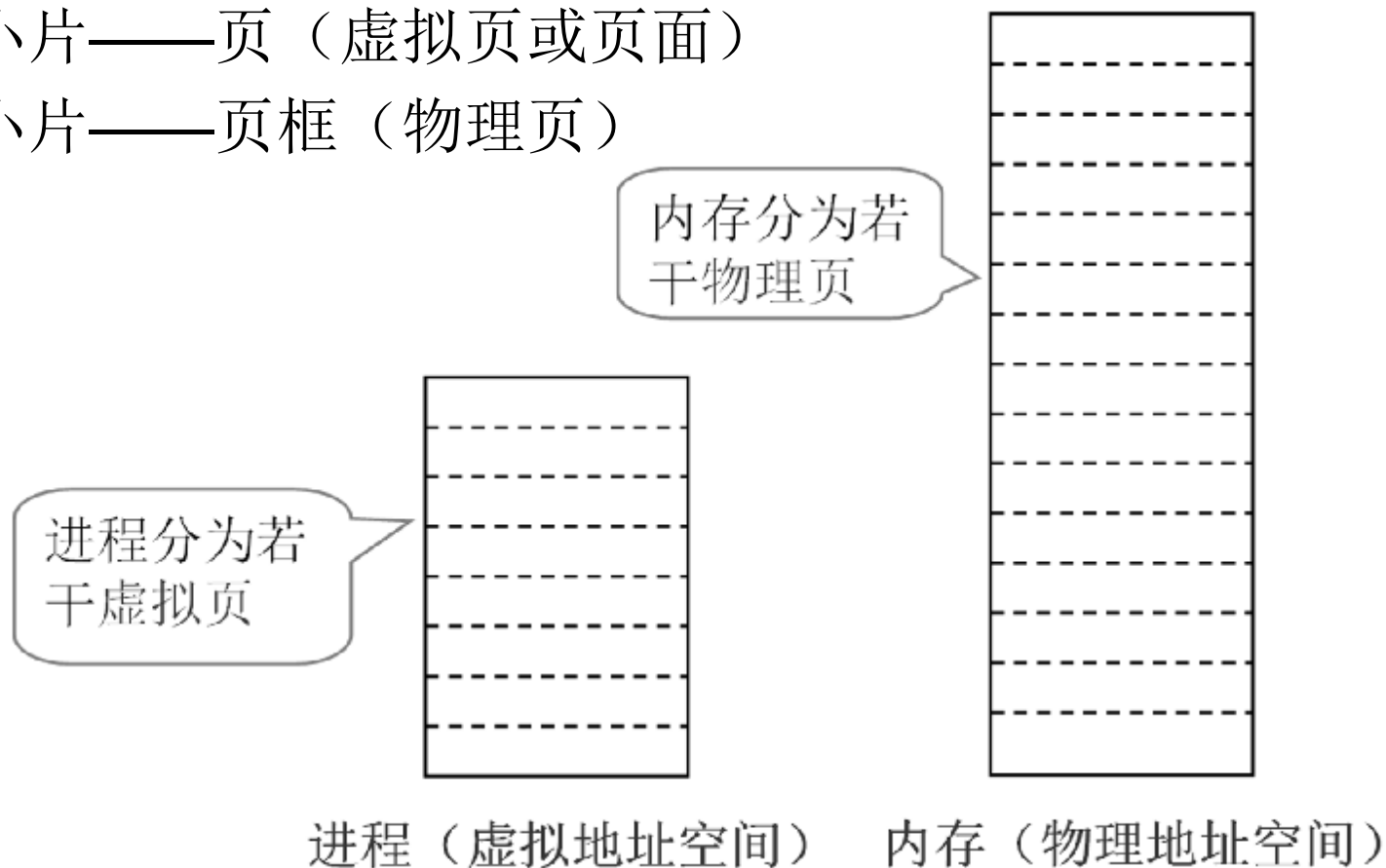
## I 概念

**n** 把进程空间（**虚拟**）和内存空间都划分成等大小的小片

**u** 小片的典型大小：1K，2K或4K...

**u** 进程的小片——页（虚拟页或页面）

**u** 内存的小片——页框（物理页）



# 页式虚拟存储管理

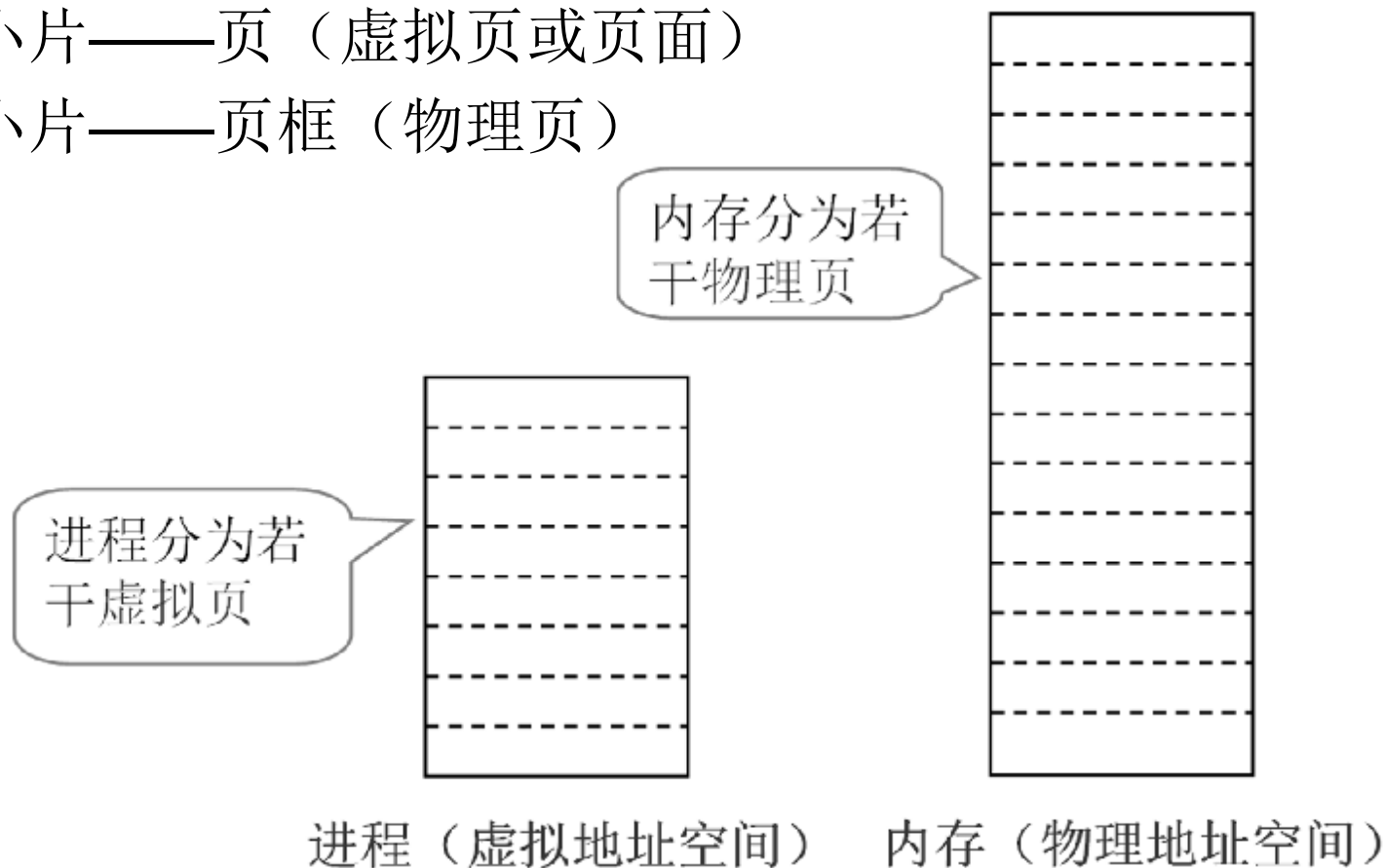
## I 概念

**n** 把进程空间（**虚拟**）和内存空间都划分成等大小的小片

**u** 小片的典型大小：1K，2K或4K...

**u** 进程的小片——页（虚拟页或页面）

**u** 内存的小片——页框（物理页）



# 虚拟内存管理的实现思路

在程序运行时，只把当前必要的很小一部分代码和数据装入内存中，以节省内存需求。其余代码和数据需要时再装入。不再运行的代码和数据及时从内存删除。

实际内存很容易就能满足上述的内存需求。

# I 进程装入和使用内存的原则

n 内存以页框为单位分配使用。

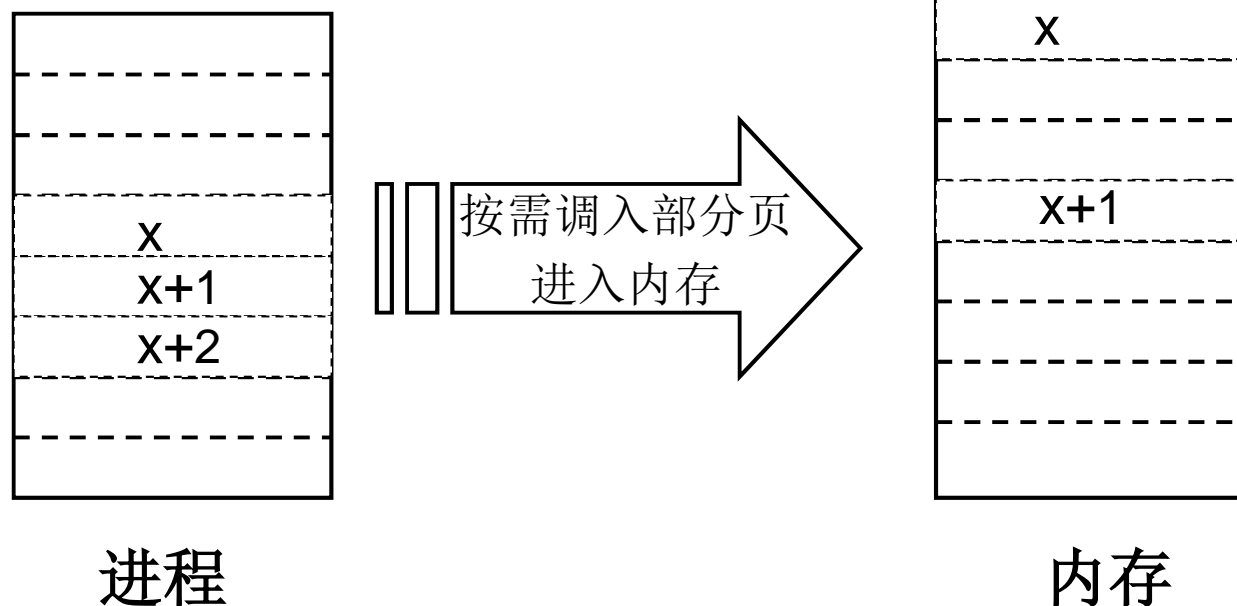
n 进程以页为单位装入内存

u 只把程序部分页装入内存便可运行。

u 页在内存中占用的页框不必相邻。

u 需要新页时，按需从硬盘调入内存。

u 不再运行的页及时删除，腾出空间





# I 实现页式虚拟内存管理需要解决的若干问题

n1. 虚拟地址如何组织或表达？

- u 页式地址

n2. 虚拟地址如何转化为物理地址（地址映射）？

- u 页表

- u 地址映射的过程

n3. 所需页面不在内存中怎么办？

- u 缺页，缺页中断，页面调入

n4. 内存中页框不够用怎么办？

- u 页面淘汰

n5. ....

# 页式系统中的地址

I 虚拟地址(VA) 可以分解成页号P和页内偏移W

n 页号 (P)

u VA所处页的编号 =  $VA / \text{页的大小}$

n 页内偏移(W)

u VA在所处页中的偏移 =  $VA \% \text{页的大小}$

I 例子

n  $VA = 2500$ ; 页面大小1K(1024)

$P = 2500 / 1024 = 2$

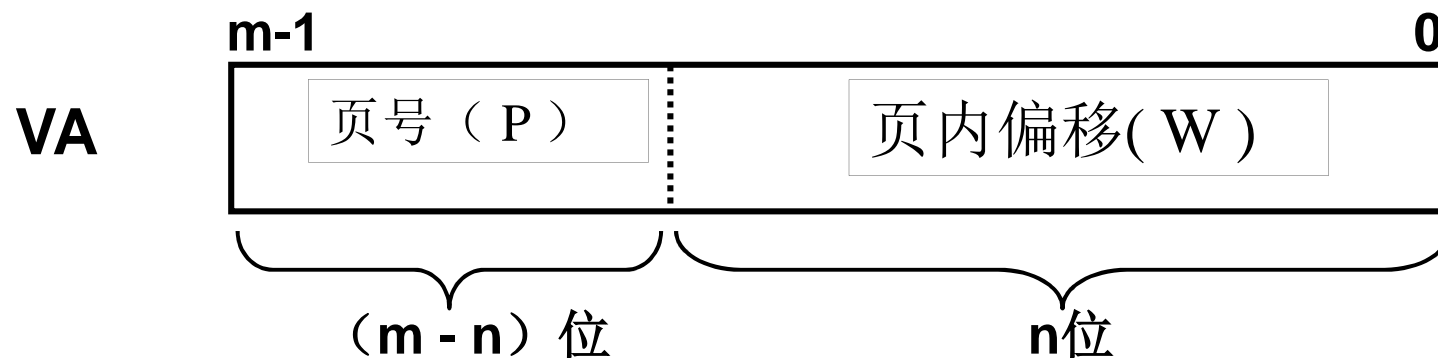
$W = 2500 \% 1024 = 452$

# P和W的另一种计算方法

## I 已知

**n** 虚拟地址的宽度:  $m$  位

**n** 页的大小:  $2^n$  单元 (显然  $m > n$ )



## I P和W计算

**n** 页号  $P =$  虚拟地址的高  $(m - n)$  位  $= VA \gg n$

**n** 页内偏移  $W =$  低  $n$  位  $= VA \&\& 2^n$

# 地址映射

- 丨 页面映射表
- 丨 地址映射过程

## I 页面映射表

**n**记录页与页框之间的对应关系。也叫页表。

页号	页框号	页面其它特性
<b>0</b>	<b>5</b>	...
<b>1</b>	<b>65</b>	...
<b>2</b>	<b>13</b>	...

## I 页表结构

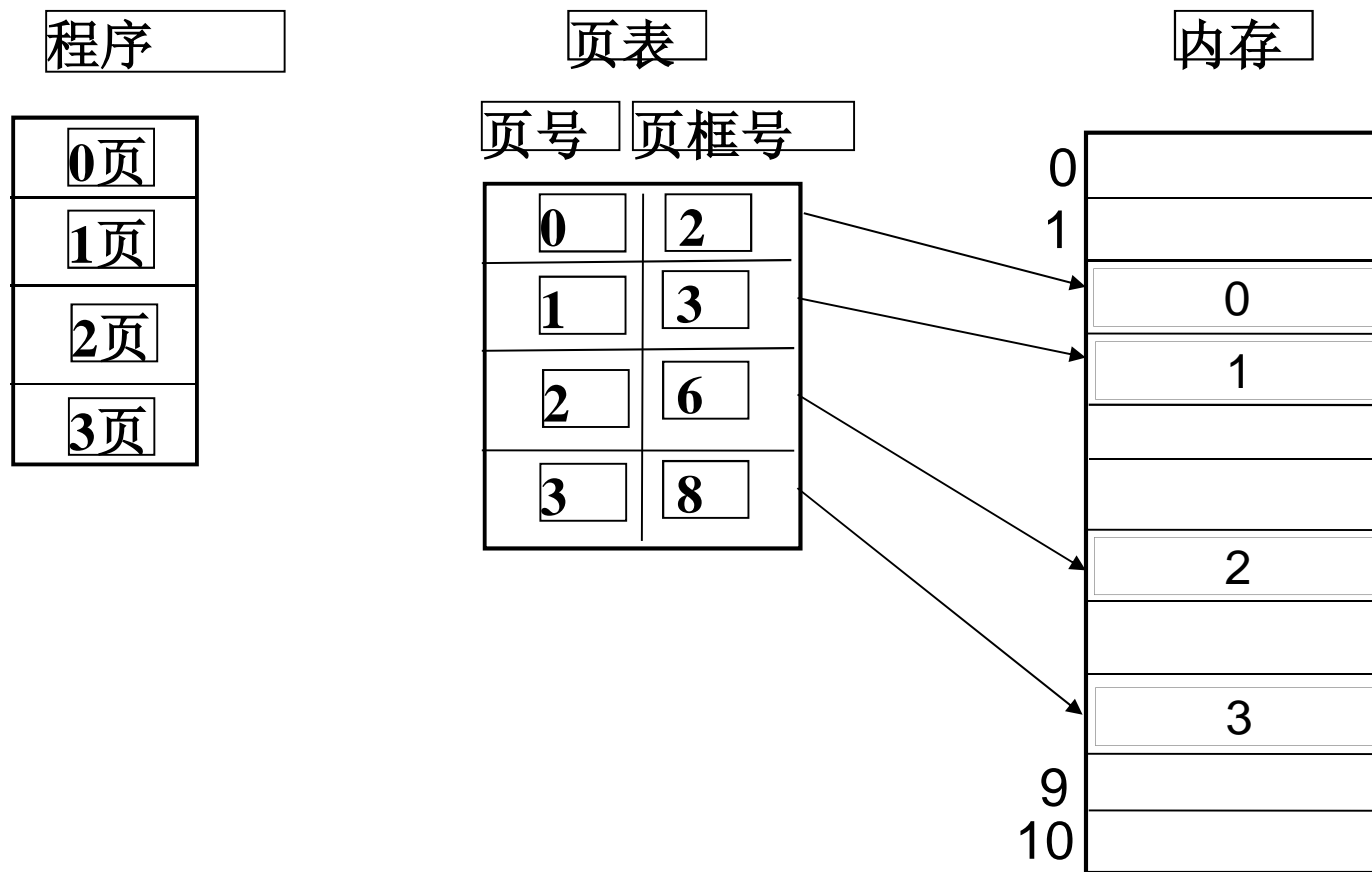
**n**页号：登记程序地址的页号。

**n**页框号：登记页所在的物理页号。

**n**页面其他特性：登记含存取权限在内的其他特性。

# I 页表例子

n一个进程：4页



# 页式地址映射

## I 功能

n 虚拟地址（页式地址）→物理地址

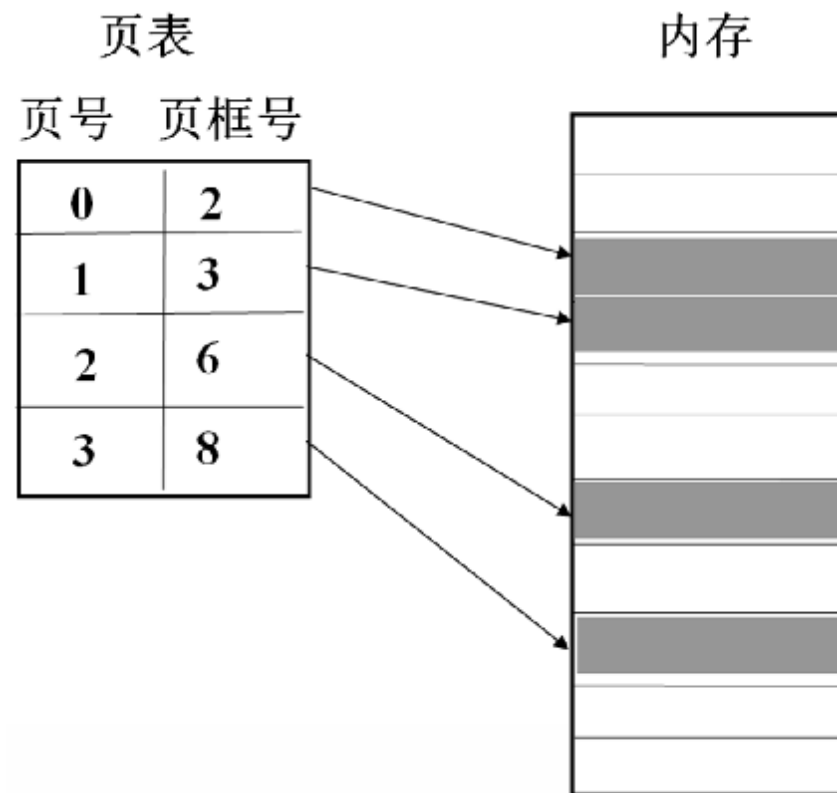
## I 过程【三步】

n 1.从VA分离页号P和页内偏移W;

n 2.查页表：以P为索引查页框号P';

n 3.计算物理地址MA

$$MA = P' \times \text{页大小} + W$$



I 页式地址映射例子  
nMOV R1, [2500]

I 解:

1. 分离P, W.

$$P = VA / \text{页面大小} = 2500 / 1024 = 2$$

$$W = VA \% \text{页面大小} = 2500 \% 1024 = 452$$

2. 查找页表

$$P = 2, P' = 7$$

3. 计算  $MA = P' \times \text{页面大小} + W$

$$MA = 7 * 1024 + 452 = 7620$$

页号	页框号	其它特性
0	4	...
1	2	...
2	7	...



# 防止越界访问页面

## I 防止越界访问其它进程

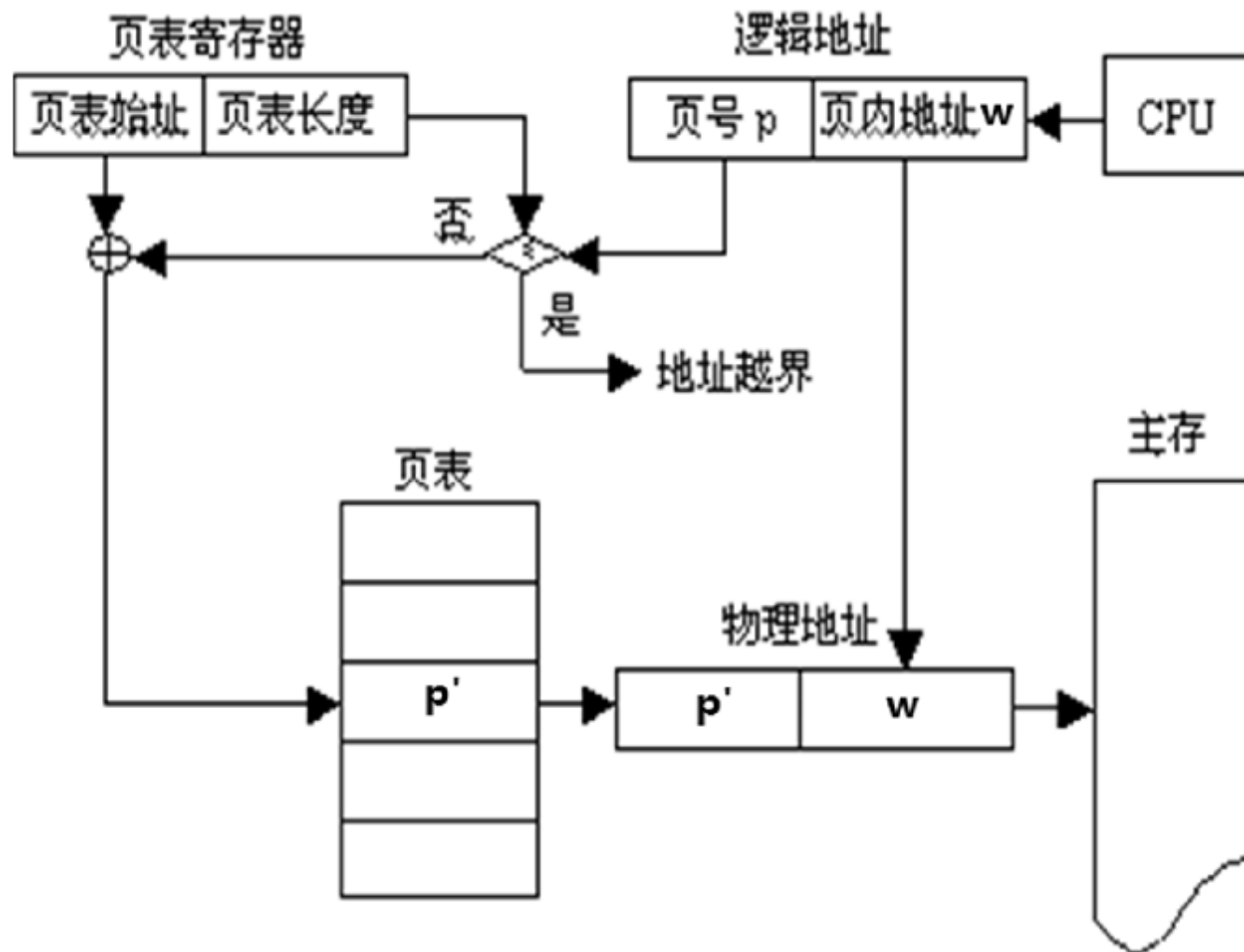
n 检查访问的目的页号 $X$ 是否在进程内？

u 检查标准： $0 \leq X < \text{虚拟页数}$

u 越界：产生越界中断

# 页式地址映射

- I 系统设置一个页表寄存器PTR（Page-Table Register），存放页表在内存的始址和页表的长度



# 页面的存取权限特性

## I 页面映射表

页号	页框号	其它特性
<b>0</b>	<b>5</b>	...
<b>1</b>	<b>65</b>	...
<b>2</b>	<b>13</b>	...

## I 其它特性

**n**存取权限

**n**是否被访问过

**n**是否被修改过

**n**.....

# 页面的存取权限

## I 防止越权访问某些页面

- n 在页表中记录每个页面的读、写、执行等权限。

- n 当访问页面时首先检查该次访问是否越权。

## I 防止越界访问其它进程

- n 检查访问的目的页号 $X$ 是否在进程内？

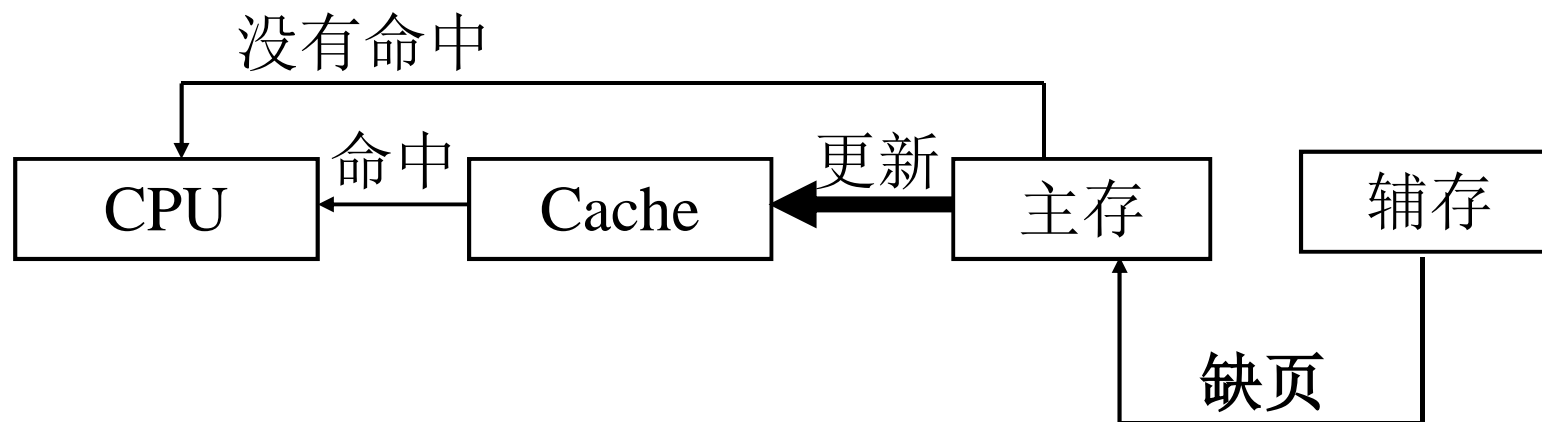
  - u 检查标准： $0 \leq X < \text{虚拟页数}$

  - u 越界：产生越界中断

# 快表机制 (Cache)

## I 分级存储体系

**n**CACHE + 内存 + 辅存



**n** 页表放在Cache中：快表

**n** 页表放在内存中：慢表

## I 快表的特点

- n 访问速度快，成本高，容量小, 具有并行查寻能力。

- n 快表是慢表的部分内容的复制。

- n 地址映射时优先访问快表

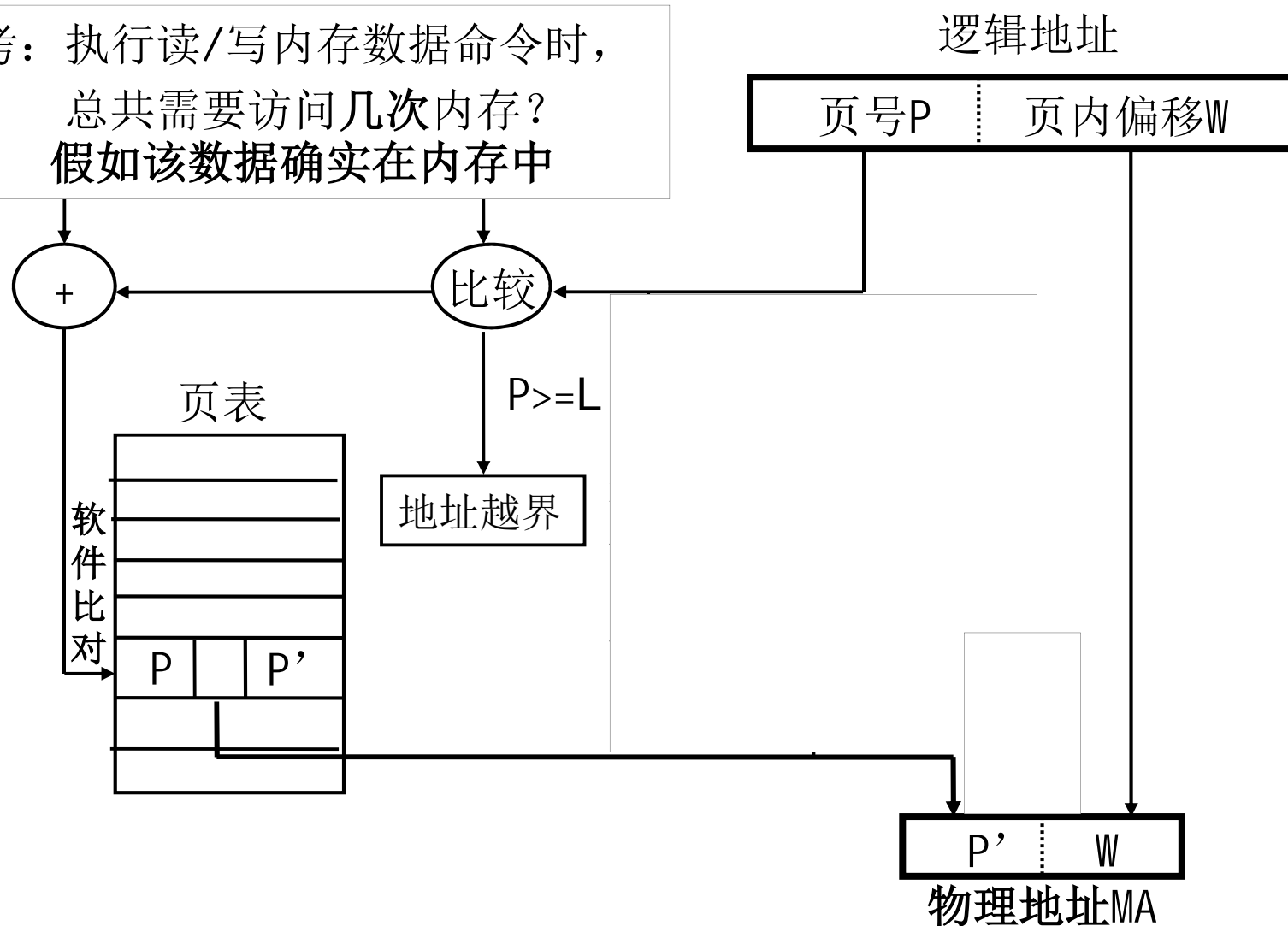
  - u 若在快表中找到所需数据，则称为“命中”

  - u 没有命中时，需要访问慢表，同时更新快表

- n 合理的页面调度策略能使快表具有较高命中率

# I 快表机制下地址映射过程

- 思考：执行读/写内存数据命令时，总共需要访问几次内存？假如该数据确实在内存中



## 例题

对于利用快表且页表存于内存的分页系统，假定CPU的一次访问内存时间为 $1\mu\text{s}$ ，访问快表时间忽略不计。如果快表命中率为85%。那么进程完成一次内存读写的平均有效时间是多少？

## ■ 分析:

**n** (1) 若直接通过快表完成，则只需1次访问内存。

**u1μs 概率 = 85%**

**n** (2) 若不能, 则需要2次访问内存。

**u2μs 概率 = 15%**

**n** (3) 平均时间=  $1\mu\text{s} * 85\% + 2\mu\text{s} * 15\%$   
=  $1.15\mu\text{s}$



# 页面的共享

## I 代码共享的例子——文本编辑器占用多少内存

n 文本编辑器：**150KB**代码段和50KB数据段

n 有10进程并发执行该文本编辑器。

n 占用内存 =  $10 \times (150 + 50) \text{ KB} = 2\text{M}$

n 如果采用代码段共享，代码段在内存只有一份真实存储

**u** 占用内存 =  $150 + 10 \times 50 = 650\text{KB}$

## I 页面共享

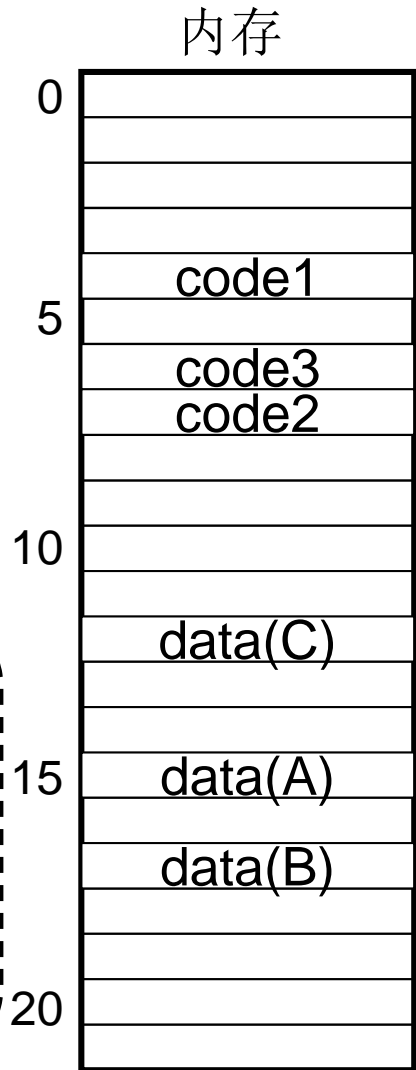
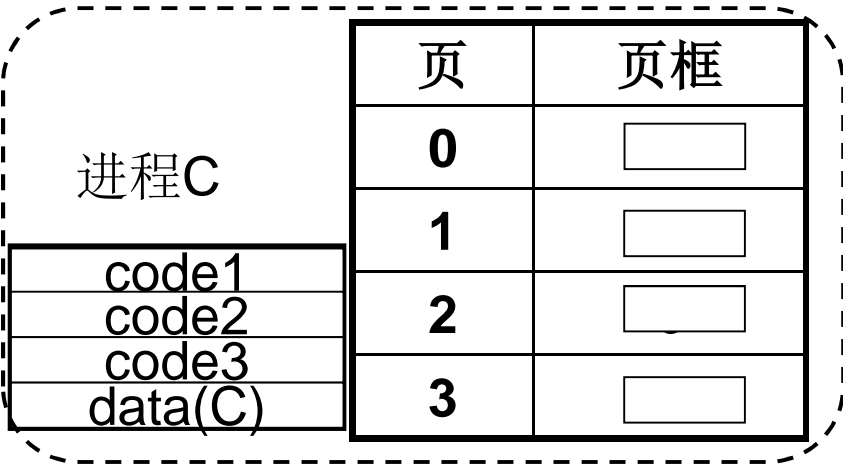
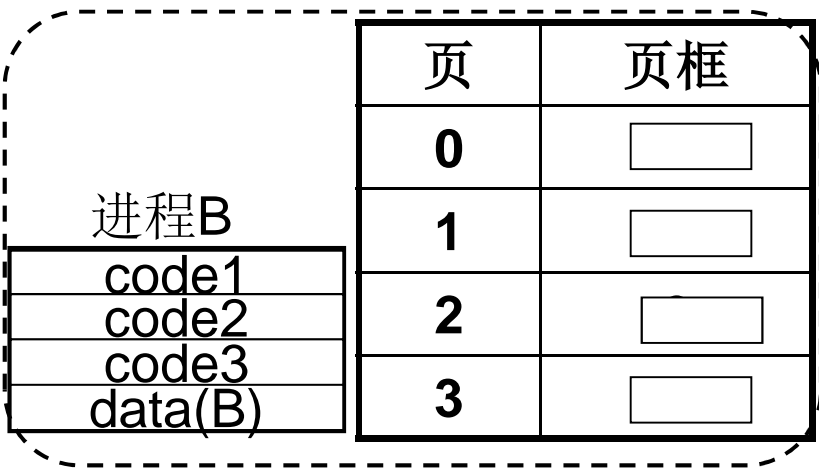
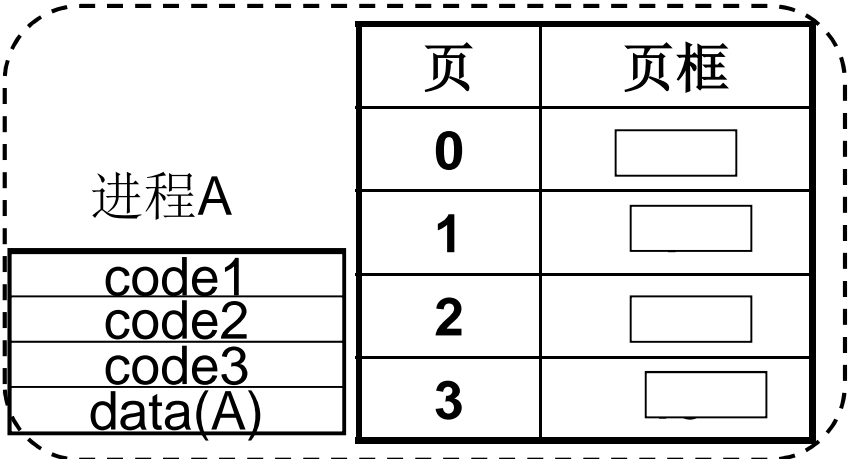
n 根据页式地址转换过程：如果在不同进程的页表中填上相同的页框号，就能访问相同的内存空间，从而实现页面共享。

n 共享页面在内存只有一份真实存储，节省内存。

# 页面的共享

## I 文本编辑器

**n** 代码段（code1~3），数据段（data）



## I 页表的建立

- n 操作系统为每个进程建立一个页表

- u 页表长度和首址存放在进程控制块中。

- n 当前运行进程的页表驻留在内存

- u 页表长度和首址由页表长度寄存器和页表首址寄存器指示。

## I 页表的形式

- n CACHE

- u 访问速度快，成本较高。

- n 内存

- u 成本较低，访问速度慢。

# I 页面的大小选择

## n 页面太大

- u 浪费内存：极限是分区存储。

## n 页面太小

- u 页面增多，页表长度增加，浪费内存；

- u 换页频繁，系统效率低

## n 页面的常见大小

- u 2的整数次幂：1KB, 2KB, 4KB

# 页表扩充——带中断位的页表

## I 扩充有中断位和辅存地址的页表

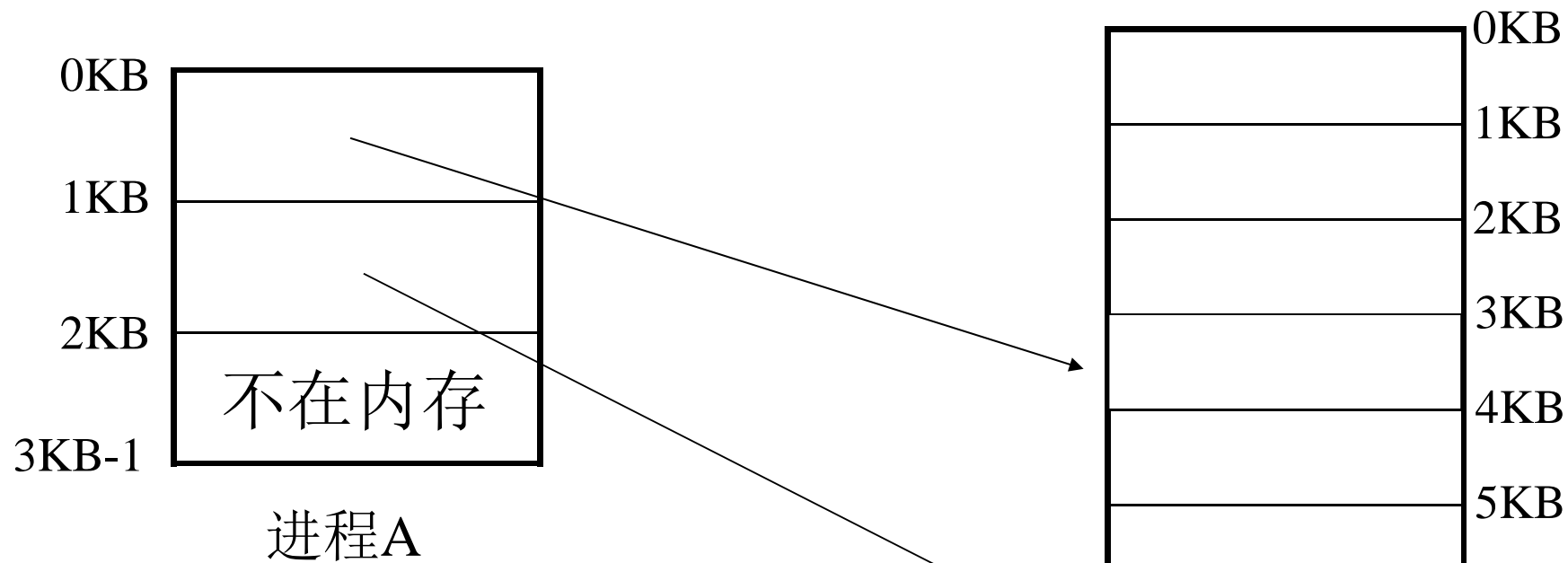
页号	页框号	中断位I	辅存地址
		1	
		0	

**n** 中断位I ——标识该页是否在内存？

**u** 若 $I = 1$ ，不在内存

**u** 若 $I = 0$ ，在内存

**n** 辅存地址——该页在辅存上的位置



页号	页框号	中断位I	辅存地址
0	3	0	1000
1	6	0	2000
2		1	6000

进程A的页表

# 页表扩充——带访问位和修改位的页表

## I 扩充有访问位和修改位的页表

页号	页框号	访问位	修改位
		<b>1</b>	<b>0</b>
		<b>0</b>	<b>1</b>

**n** 访问位——标识该页最近是否被访问？

**u** 0 ——最近没有被访问

**u** 1 ——最近已被访问

**n** 修改位——标识该页的数据是否已被修改？

**u** 0 ——该页未被修改

**u** 1 ——该页已被修改

# 缺页中断

## I 定义

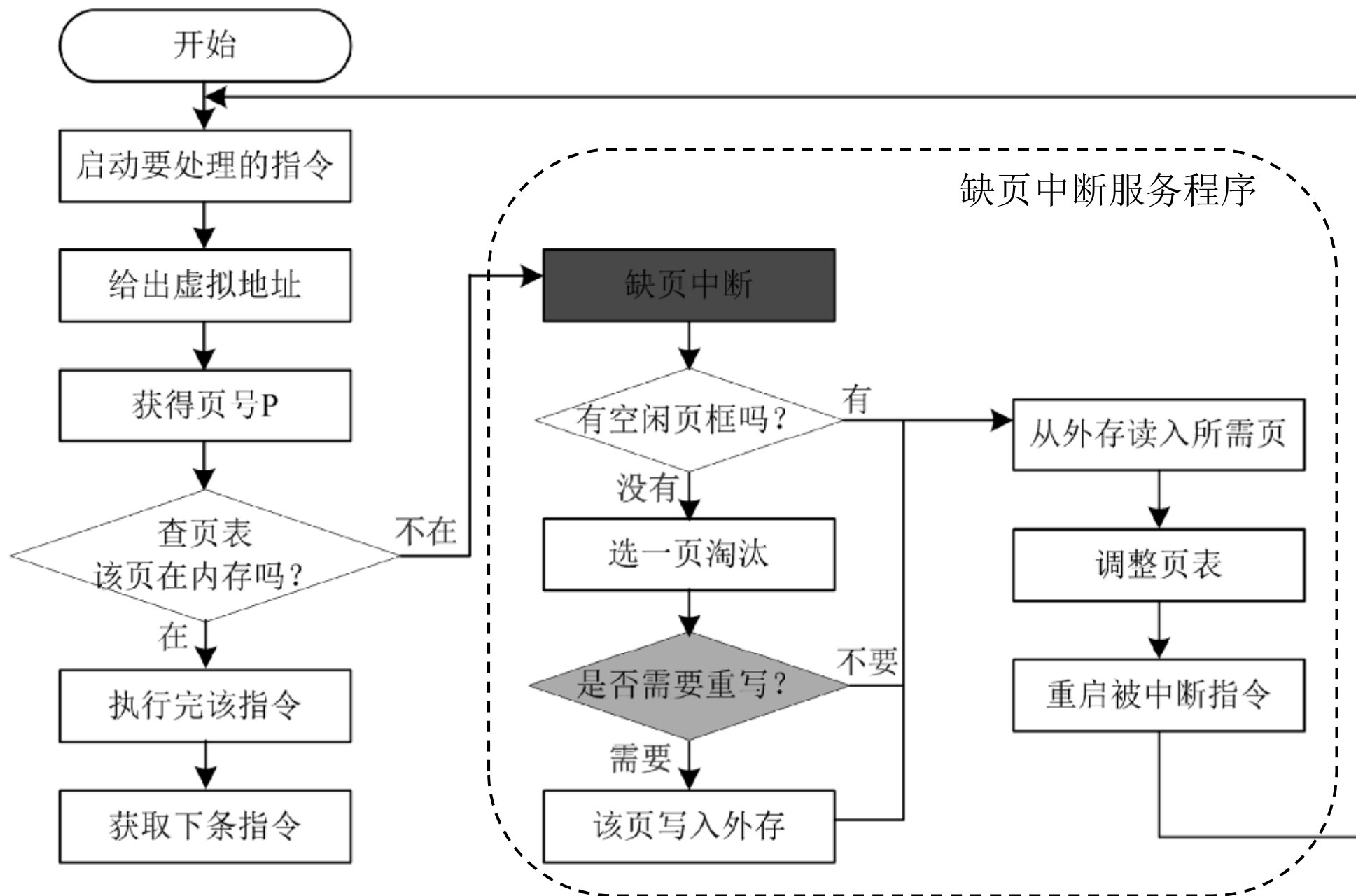
**n** 在地址映射过程中，当所要访问的目的页不在内存时，则系统产生异常中断——缺页中断。

## I 缺页中断处理程序

**n** 中断处理程序把所缺的页从页表指出的辅存地址调入内存的某个页框中，并更新页表中该页对应的页框号以及修改中断位I为0。



# 访存指令的执行过程（含缺页中断处理）



## I 缺页（中断）率

**n**缺页率  $f = \text{缺页次数} / \text{访问页面总次数}$

**n**命中率  $= 1 - f$

# 页式系统的实时性

## I 测量函数MuFunc( )花费的时间

```
1  #include <stdio.h>
2  #include <time.h>
3  int main(void)
4  { //说明: clock( )函数返回时钟滴答数, 单位: 毫秒
5      long start, end;
6      → start = clock( );
7      MyFunc( );
8      → end = clock( );
9      printf("执行MyFunc函数花费时间 %f 秒\n", end - start );
10     return 0;
11 }
```

## I 思考: 测量的时间准不准? 为什么?

# I 缺页中断 vs 普通中断

## n 相同点

- u 处理过程：保护现场、中断处理、恢复现场

## n 不同点

- u 响应时机

- p 普通中断在指令完成后响应

- p 缺页中断在指令执行过程中发生

- u 发生频率

- p 一条指令执行时可能产生多个缺页中断

# 二级页表

## I 页表实现时的问题

**n** 32位OS(4G空间)，每页4K，页表每个记录占4字节

**u** 进程的页数： $4G / 4K = 1M$ 个页

**p** 页表的记录数应有：1M条记录

▲ 页表所占内存： $1M * 4字节 = 4M$

▲ 页表占页框数： $4M / 4K = 1K$ 页框（连续）

**n** 问题：

**u1)** 难以找到连续1K个页框存放页表。

**u2)** 页表全部装入过度消耗内存（4M）。

**n** 解决办法

**u** ①将4M的超大页表存储到离散的1K个页框中；

**u** ②仅将页表的部分内容调入内存。

# 二级页表

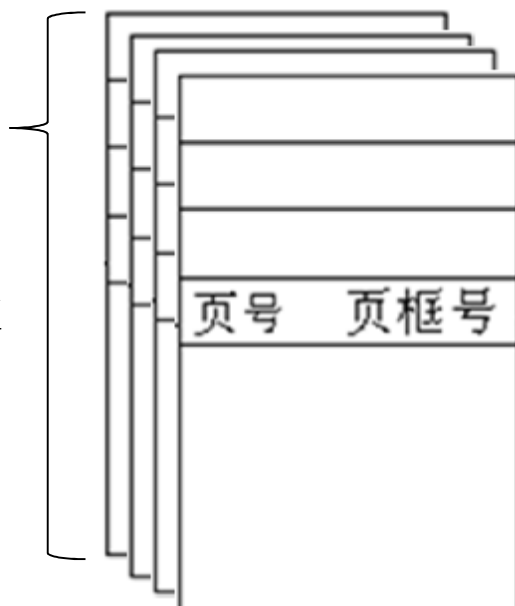
- I 把超大的页表（4M）以页为单位分成若干个小页表，存入离散的若干个页框中。

（含1M个记录）  
超大的页表

页号	页框号	中断位	外存地址	访问位	修改位
0	8	0	4000	1	0
1	24	0	8000	1	1
...	...	...	...	...	...
1M-1	5003	1	A3C4	0	1

小页表  
（含1K个记录）

分解成若干个小页表



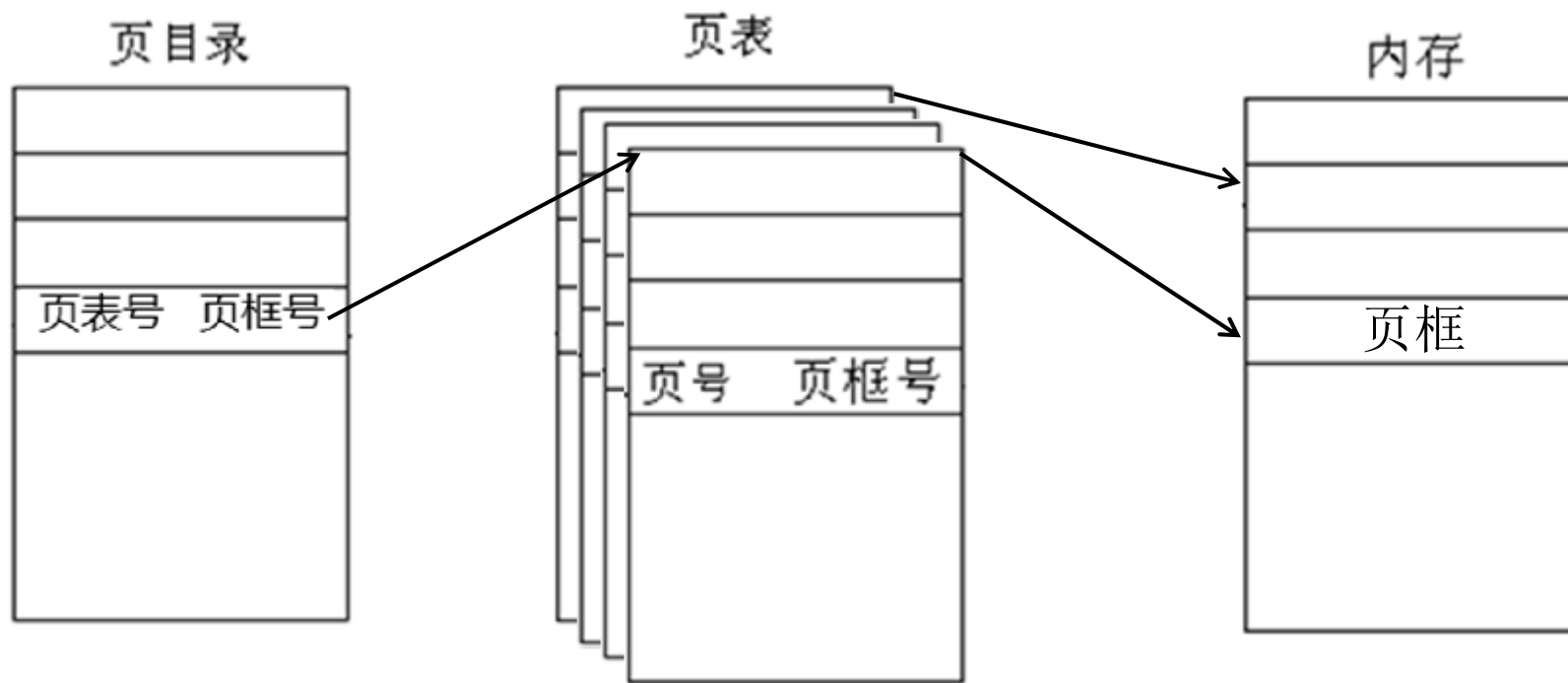
每个小页表含有1K个记录，大小是4K（=1Kx4），刚好占用一个页框。  
小页表有1K个（=1M / 1K）

## 二级页表

- Ⅰ 为了对小页表进行管理和查找，另设置一个叫页目录的表，记录每个小页表的存放位置（即页框号）。

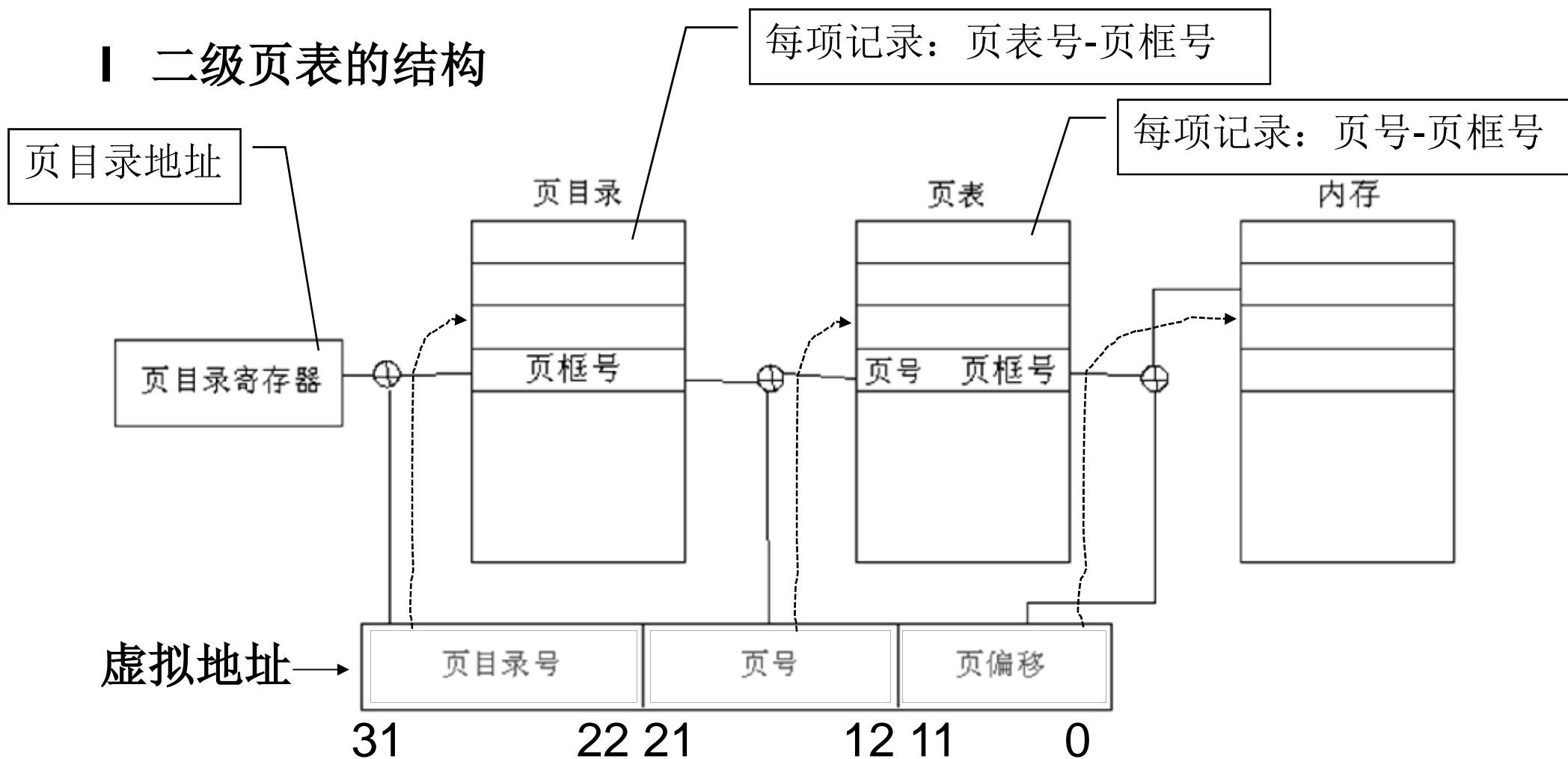
Ⅱ 页目录实际是一个特殊页表：每个记录存放的是小页表的编号和其所在的页框号之间的对应关系。

- Ⅲ 页目录：一级页表或外部页表； 小页表：二级页表



# WINDOWS NT 二级页表的结构

## I 二级页表的结构



页目录号: 页表的编号 (页目录的索引)  $2^{10} = 1\text{K}$  个页表

页号: 页面的编号 (页表的索引)  $2^{10} = 1\text{K}$  个页面

页偏移: 页偏移 页面大小 ( $2^{12} = 4\text{K}$ )



## I 二级页表地址映射特点

**n** 访问数据需要  次访问内存。

**n** 页目录调入内存

**n** 页表按需要调入主存

**n** 页面、页表，页目录的大小都刚好4K(占1个页框)。

# 淘汰策略

## I 淘汰策略

n 选择淘汰哪一页的规则称淘汰策略。

## I 页面抖动

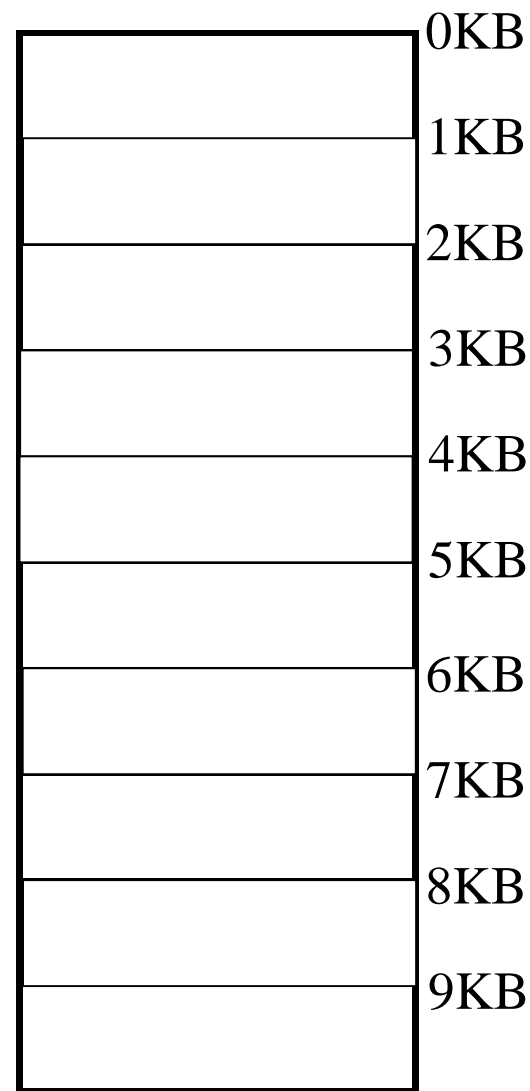
n 页面在内存和辅存间频繁交换的现象。

n “抖动”会导致系统效率下降。

## I 好的淘汰策略

n 具有较低的缺页率（高命中率）

n 页面抖动较少



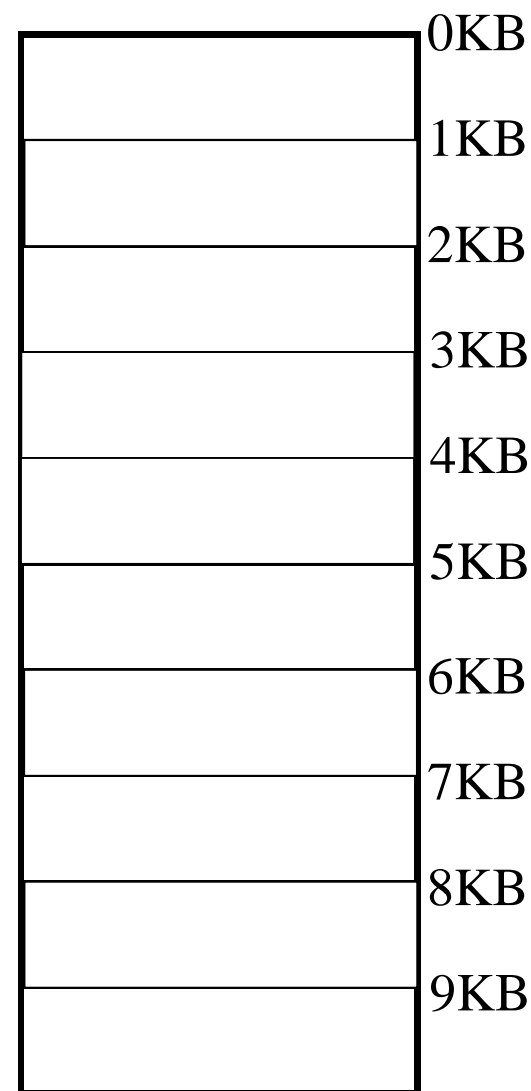
## I 常用的淘汰算法

n 最佳算法（OPT算法）

n 先进先出淘汰算法（FIFO算法）

n 最久未使用淘汰算法（LRU算法）

n 最不经常使用（LFU）算法



# 最佳算法（OPT算法, Optimal）

## I 思想

n 淘汰以后不再需要或最远的将来才会用到的页面。

## I 例子

n 分配3个页框。页面序列：A,B,C,D,A,B,E,A,B,C,D,E。分析其按照OPT算法淘汰页面的缺页情况。

序列	A	B	C	D	A	B	E	A	B	C	D	E
内存	A	A	A	A	A	A	A	A	A	C	C	C
		B	B	B	B	B	B	B	B	B	D	D
			C	D	D	D	E	E	E	E	E	E
缺页	X	X	X	X			X			X	X	

缺页次数 = 7      缺页率 =  $7 / 12 = 58\%$

# 最佳算法（**OPT**算法, **Optimal**）

## I 特点

**n**理论上最佳，实践中该算法无法实现。

# 先进先出淘汰算法（FIFO算法）

## I 思想

n 淘汰在内存中停留时间最长的页面

## I 例子

n 页框数为3。页面序列：A,B,C,D,A,B,E,A,B,C,D,E，分析其按照FIFO算法淘汰页面的缺页情况。

序列	A	B	C	D	A	B	E	A	B	C	D	E
内存	A	A	A	D	D	D	E	E	E	E	E	E
		B	B	B	A	A	A	A	A	C	C	C
			C	C	C	B	B	B	B	B	D	D
缺页	X	X	X	X	X	X	X			X	X	

缺页次数 = 9， 缺页率 =  $9 / 12 = 75\%$

## I 优点

- n 实现简单：页面按进入内存的时间排序，淘汰队头页面。

## I 缺点

- n 进程只有按顺序访问地址空间时页面命中率才最理想。

- n 异常现象：对于一些特定的访问序列，随分配的页框增多，缺页率反而增加！

## I FIFO例子

n分配4页框，页面序列：A,B,C,D,A,B,E,A,B,C,D,E。  
分析其按照FIFO算法进行页面淘汰时的缺页情况。

序列	A	B	C	D	A	B	E	A	B	C	D	E
内存	A	A	A	A	A	A	E	E	E	E	D	D
		B	B	B	B	B	B	A	A	A	A	E
			C	C	C	C	C	C	B	B	B	B
				D	D	D	D	D	D	C	C	C
缺页	X	X	X	X			X	X	X	X	X	X

缺页次数 = 10, 缺页率 =  $10 / 12 = 83\%$



# 最久未使用淘汰算法（LRU, Least Recently Used）

## I 思想

n 淘汰最长时间未被使用的页面。

## I 例子

n 3个页框，页面序列：A,B,C,D,A,B,E,A,B,C,D,E。分析其按照LRU算法进行页面淘汰时的缺页情况。

序列	A	B	C	D	A	B	E	A	B	C	D	E
内存	A	A	A	D	D	D	E	E	E	C	C	C
		B	B	B	A	A	A	A	A	A	D	D
			C	C	C	B	B	B	B	B	B	E
缺页	X	X	X	X	X	X	X			X	X	X

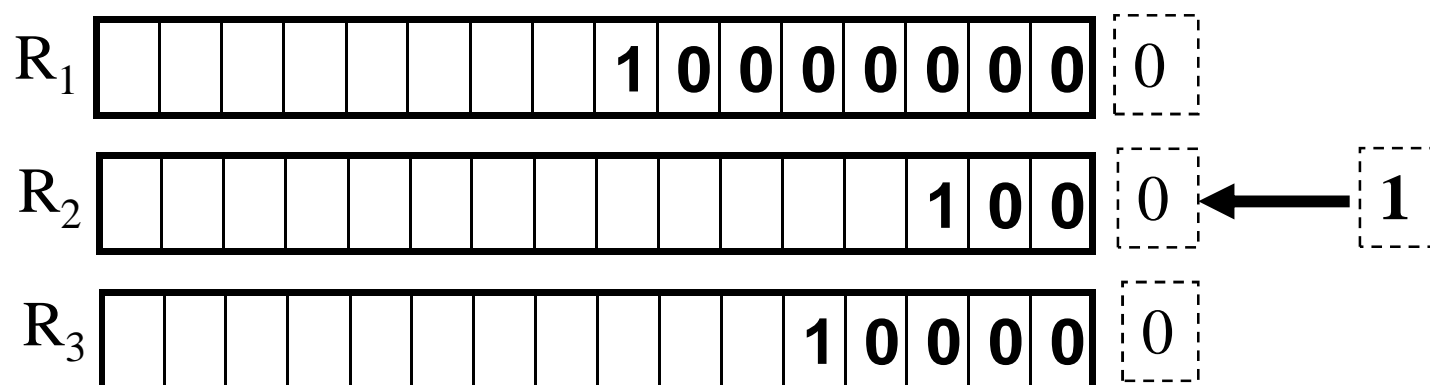
缺页次数 = 10, 缺页率 =  $10 / 12 = 83\%$



## I LRU的实现（硬件方法）

n 页面设置一个移位寄存器R。每当页面被访问则将其重置1。

n 周期性地(周期很短)将所有页面的R左移1位（右边补0）



n 当需要淘汰页面时选择R值最大的页。

uR值越大，对应页未被使用的时间越长。

n R的位数越多且移位周期越小就越精确，但硬件成本也越高。

n 若R的位数太少，可能同时出现多个位0页面情况，难以比较。

## I LRU近似算法

- n 利用页表访问位，页被访问时其值由硬件置1。
- n 软件周期性（ $T$ ）地将所有访问位置0。
- n 当淘汰页面时根据该页访问位来判断是否淘汰
  - u 访问位为1：在时间 $T$ 内，该页被访问过，保留该页。
  - u 访问位为0：在时间 $T$ 内，该页未被访问过，淘汰该页！

## I 缺点

- n 周期 $T$ 难定
  - u 太小，访问位为0的页过多，所选未必是最久未用。
  - u 太大，访问位全部为1，找不到合适的页。

# 最不经常使用（**LFU**）算法

## I **Least Frequently Used**

### I 算法原则

- n** 选择到当前时间为止被访问次数最少的页面
- n** 每页设置访问计数器，每当页面被访问时，该页面的访问计数器加1；
- n** 发生缺页中断时，淘汰计数值最小的页面，并将所有计数清零。

# I 影响缺页次数的因素

- n 淘汰算法

- n 分配给进程的页框数

  - u 页框越少，越容易缺页

- n 页本身的大小

  - u 页面越小容易缺页

- n 程序的编制方法

  - u .....

# I 影响缺页次数的因素

- n 淘汰算法

- n 分配给进程的页框数

  - u 页框越少，越容易缺页

- n 页本身的大小

  - u 页面越小容易缺页

- n 程序的编制方法

  - u 局部性越好，越不容易缺页

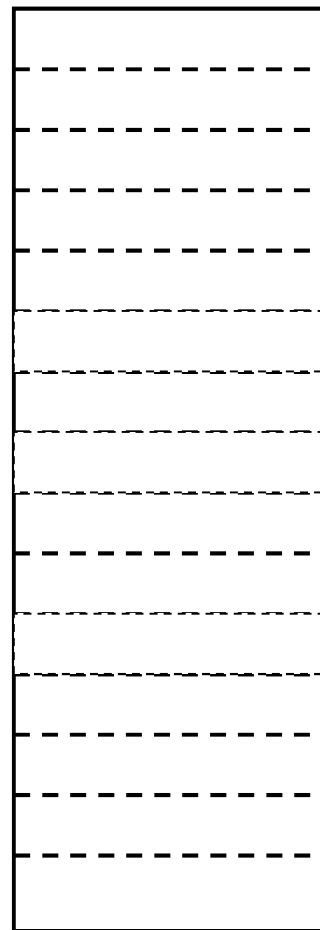
  - u 跳转或分支越多越容易缺页

## I 页式系统的不足

**n** 页面划分无逻辑含义

**n** 页的共享不灵活

**n** 页内碎片



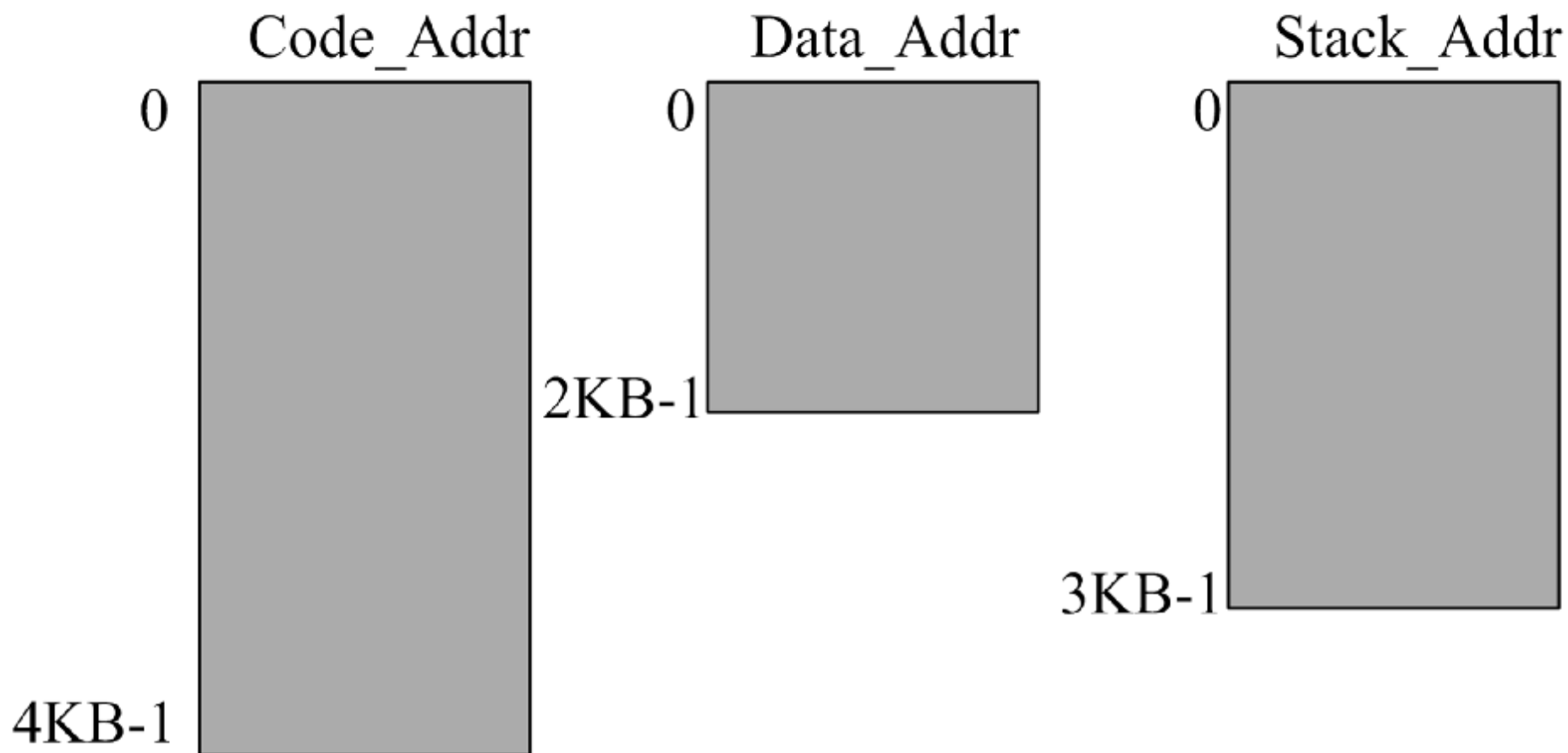
内存

# 段式存储管理

## I 进程分段

**n** 把进程按逻辑意义划分为多个段，每段有段名，长度不定。  
进程由多段组成，

**n** 例：一个具有代码段、数据段、堆栈段的进程





## I 段式内存管理系统的内存分配

n 以段为单位装入，每段分配连续的内存；

n 但是段和段不要求相邻。

## I 段式系统的虚拟地址

n 段式虚拟地址VA包含段号S和段内偏移W

n VA: (S, W)

段号S	段内位移W
-----	-------

# 段式地址的映射机制

## I 段表（**SMT**, **Segment Memory Table**）

**n**记录每段在内存中映射的位置

段号	段长	基地址
S	L	B

## I 段表的字段

**n**段号**S**：段的编号（唯一的）

**n**段长**L**：该段的长度

**n**基地址**B**：该段在内存中的首地址

# 段式地址映射过程

## I 段式地址映射过程

**n1.**由逻辑地址VA分离出(S, W);

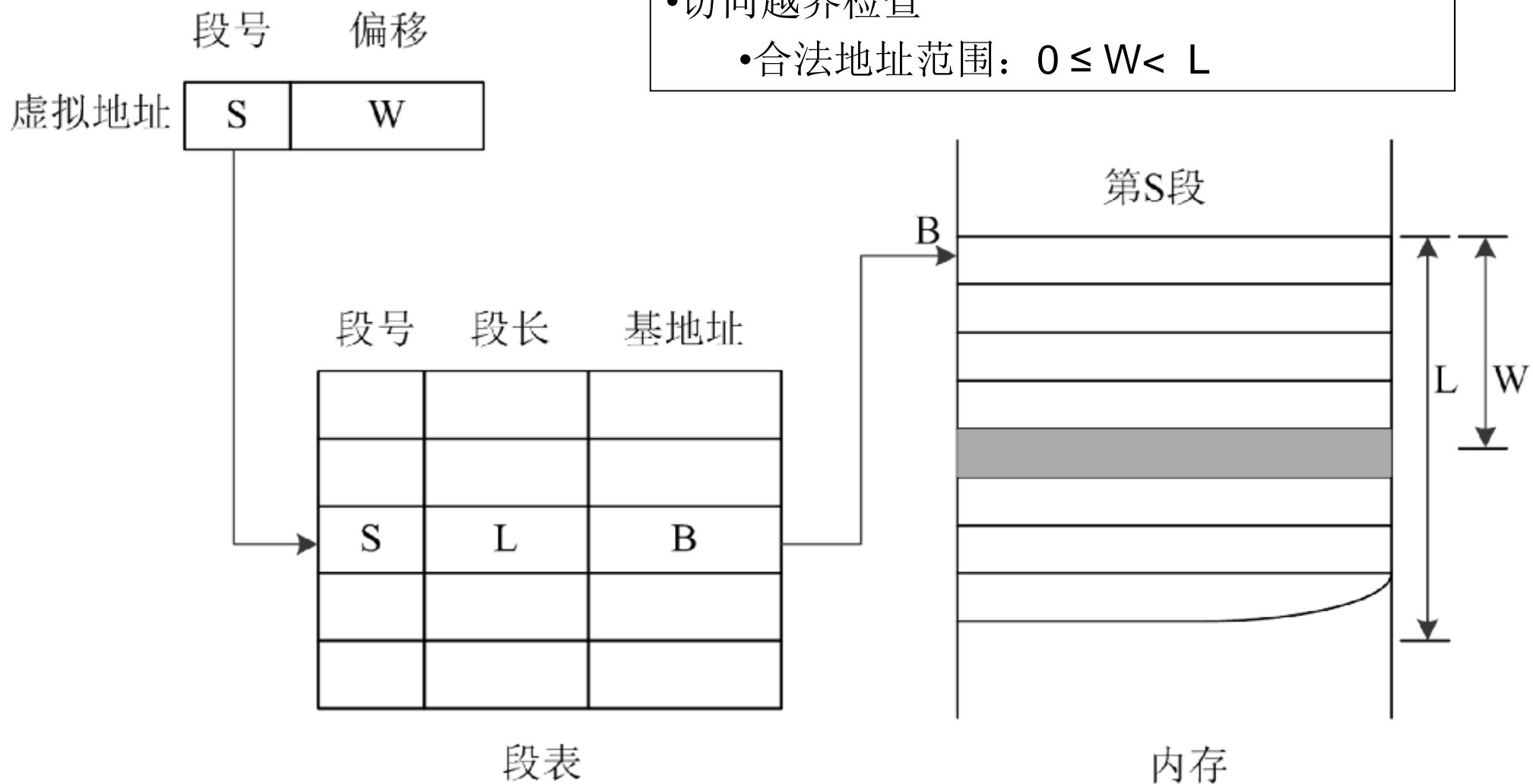
**n2.**查询段表

u检索段号S，查询该段基地址B和长度L。

**n3.**物理地址 $MA = B + W$

# 段式地址映射过程

- 访问越界检查
- 合法地址范围:  $0 \leq W < L$



## I 段表的扩充

**n**基本字段：段号，长度，基址

**n**扩展字段：中断位，访问位，修改位，R/W/X

段号	长度	基址	中断位	访问位	修改位	R	W	X

## I 段的共享

- n 共享段在内存中只有一份存储。
- n 共享段被进程映射到自己的空间（写入段表）
- n 需要共享的模块都可以设置为单独的段

## I 段式系统的缺点

- n 段需要连续的存储空间
- n 段的最大尺寸受到内存大小的限制；
- n 在辅存中管理可变尺寸的段比较困难；

# 段式系统 vs 页式系统

## I 地址空间的区别

**n**页式系统：一维地址空间

**n**段式系统：二维地址空间

## I 段与页的区别

**n**段长可变；页面大小固定

**n**段的划分有意义；页面无意义

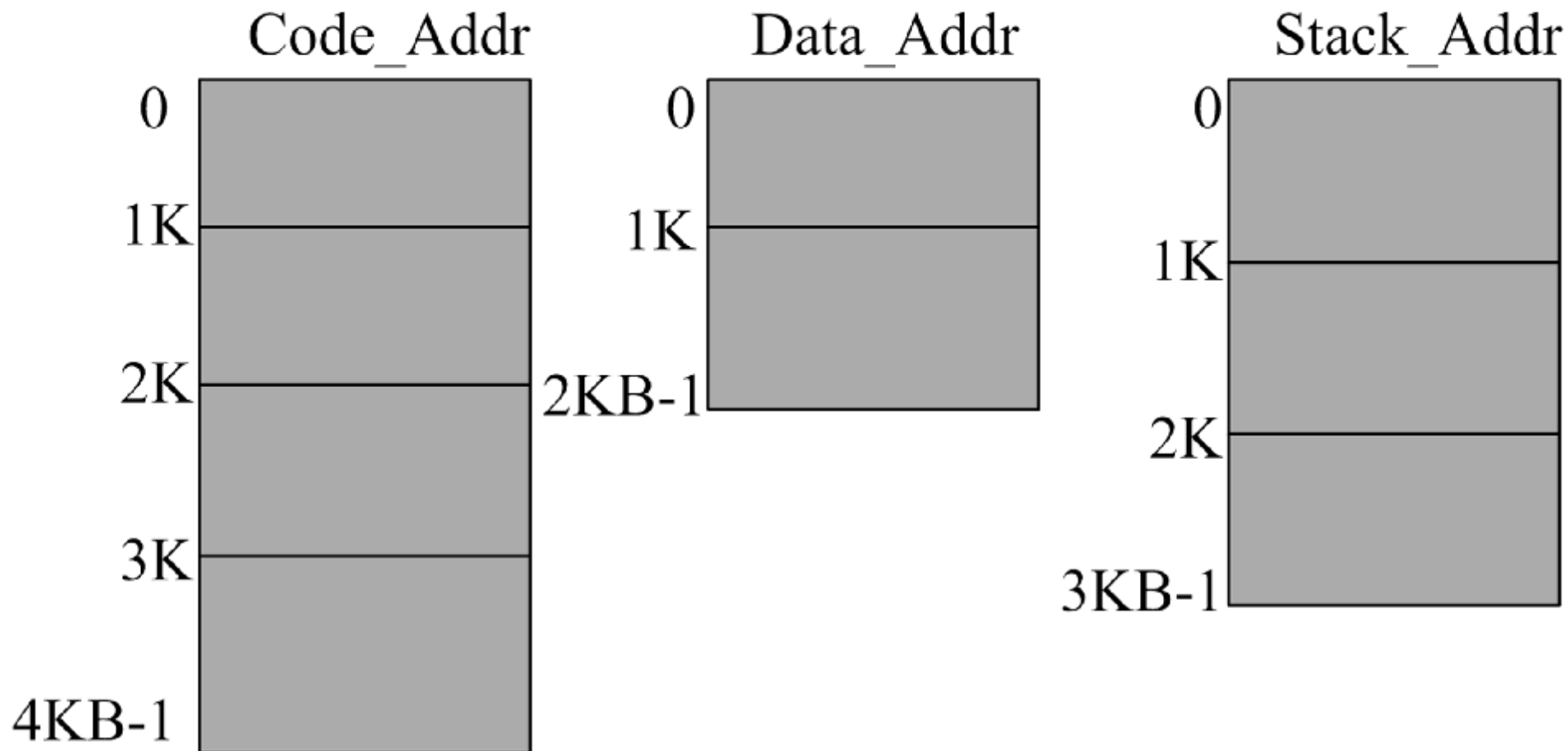
**n**段方便共享；页面不方便共享

**n**段用户可见；页面用户不可见

**n**段偏移有溢出；页面偏移无溢出

# 段页式存储管理

- 丨 在段式存储管理中结合页式存储管理技术
- 丨 在段中划分页面。





# I 段页式系统的地址构成：段号，页号，页内偏移



**n**逻辑地址：段号S、页号P和页内位移W。

**n**内存按页划分，按页装入。

## I 段页式地址的映射机构

n 同时采用段表和页表实现地址映射。

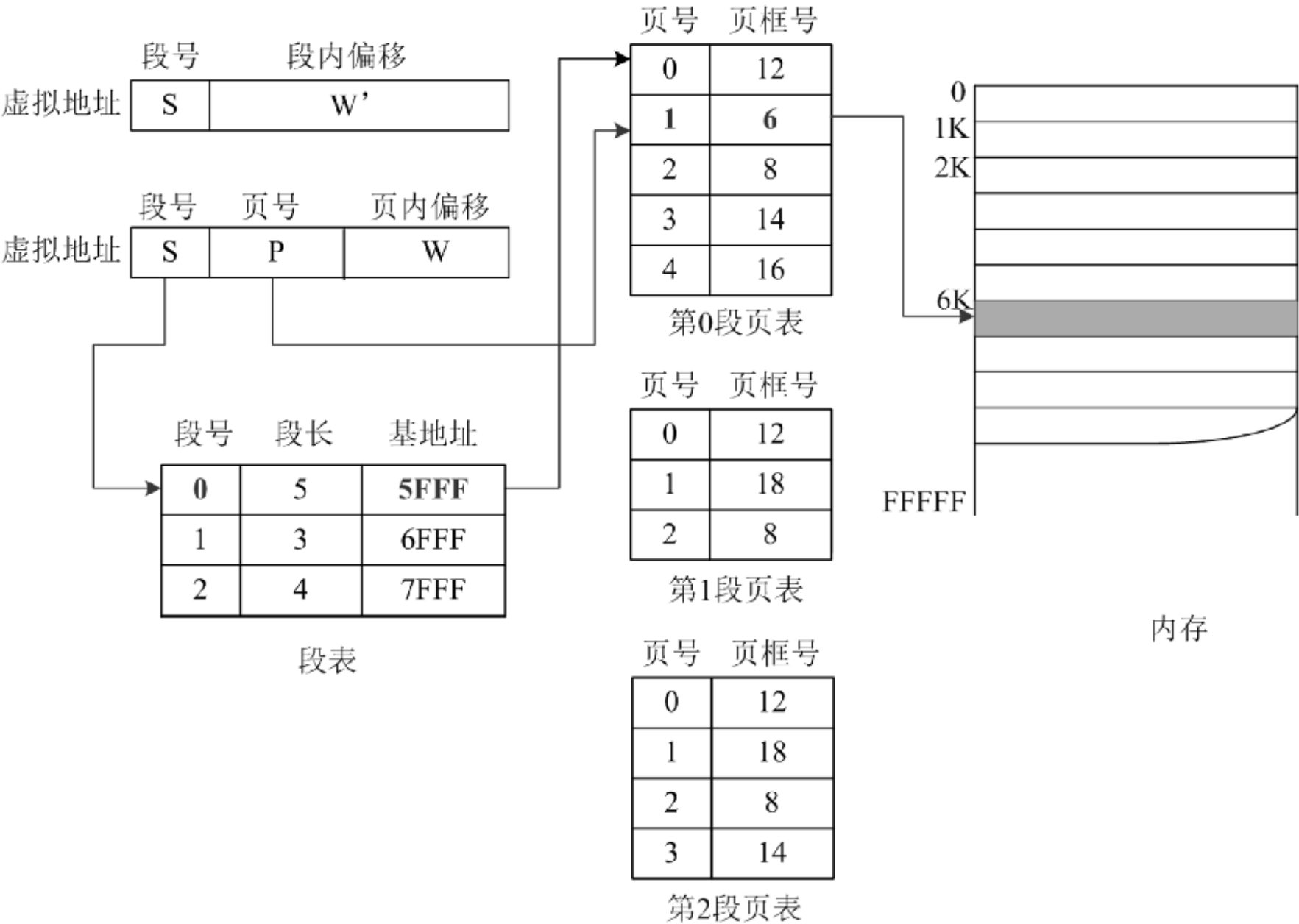
- u 系统为每个进程建立一个段表；

- u 系统为每个段建立一个页表；

- u 段表给出每段的页表基地址及页表长度（段长）。

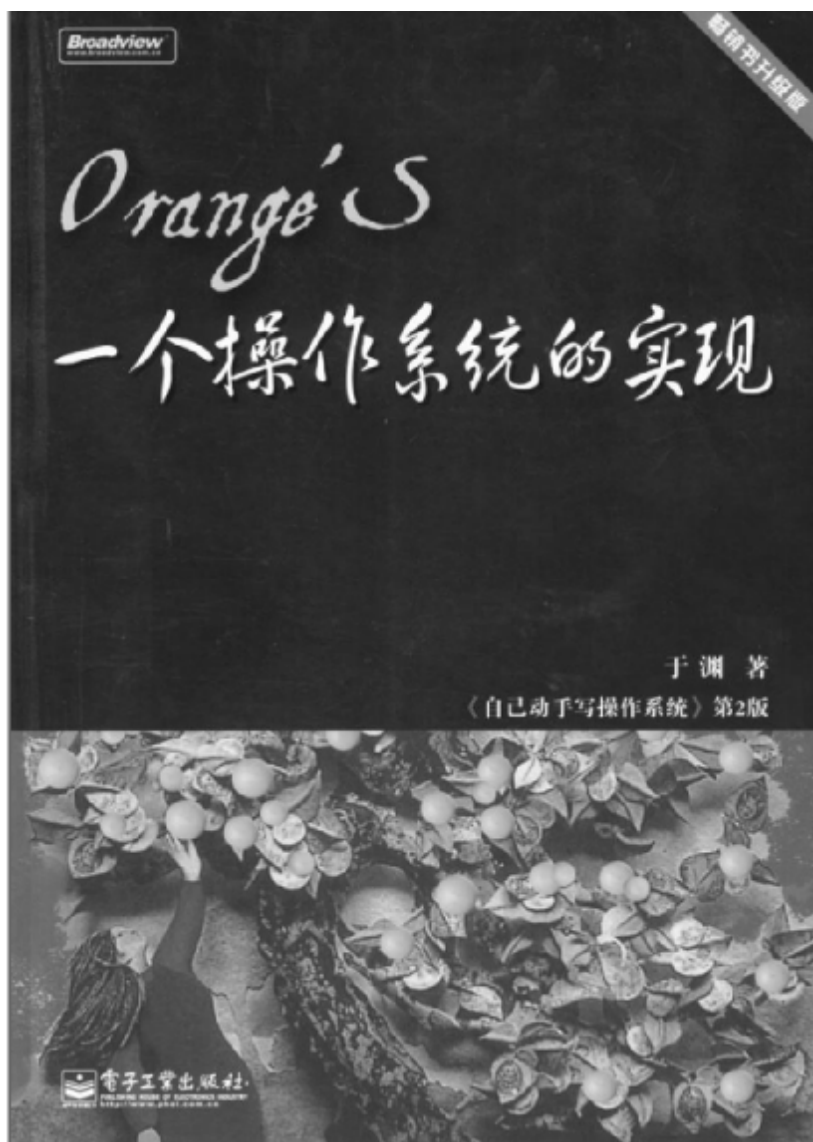
- u 页表给出每页对应的页框。

# 段页式地址映射: $VA(S, W') \rightarrow (S, P, W) \rightarrow MA$





## 4. i 386和Linux存储管理



# **i386和Linux的存储机制**

- I I386的段页硬件机制**
- I Linux的段页管理机制**

# I386的段页硬件机制

## I 实模式（Real Mode）

- n 20位地址

  - u 段地址（16位）：偏移地址（16位）

  - u 段地址16字节对齐

  - u 20位：1M内存空间

## I 保护模式（Protect Mode）

- n 32位地址空间：4G物理内存

- n 支持多任务，能够快速地进行任务切换和保护任务环境

- n 支持资源共享，能保证代码和数据的安全和任务隔离

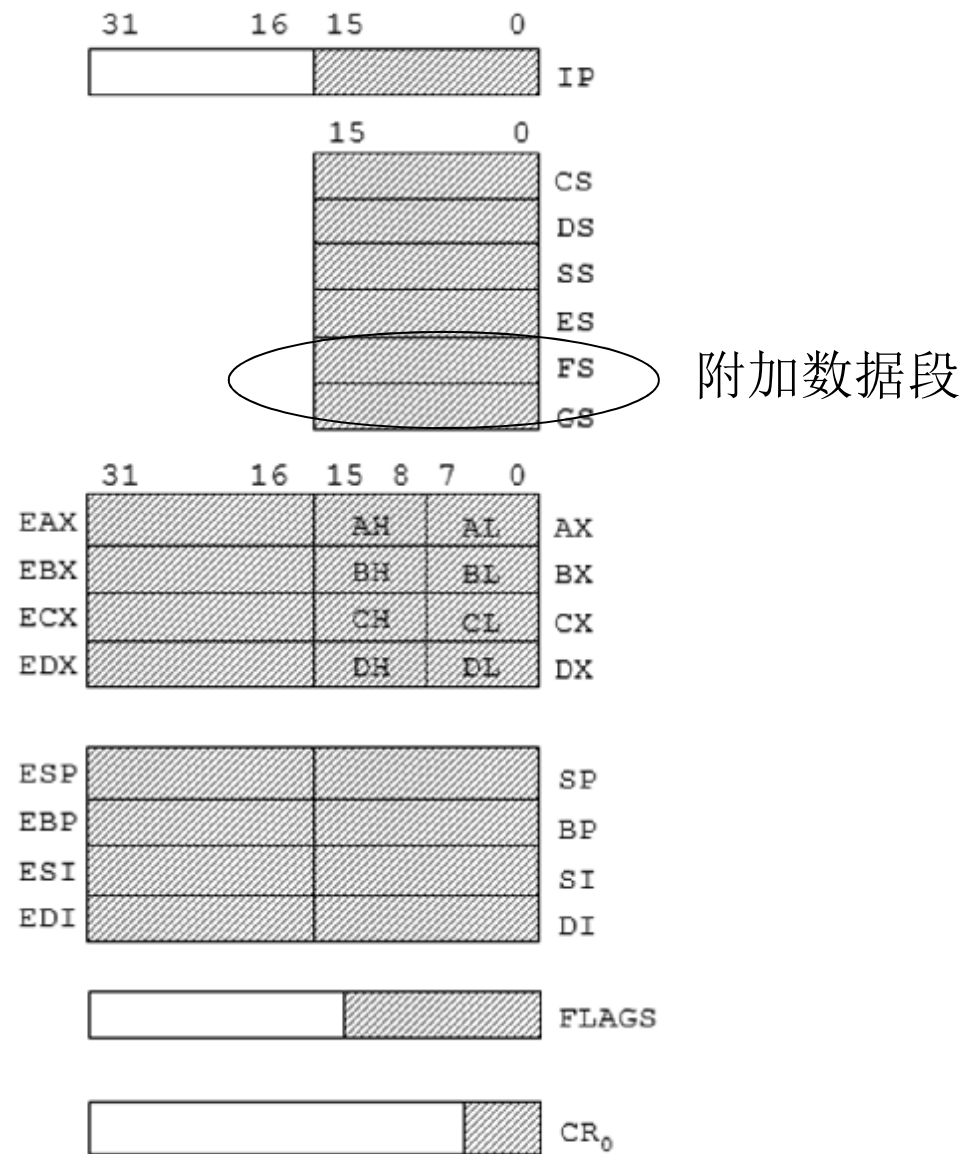
- n 支持分段机制+分页机制

- n 新的硬件

  - u EAX~EDX(32位扩展)

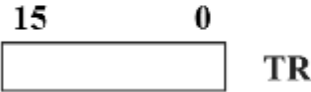
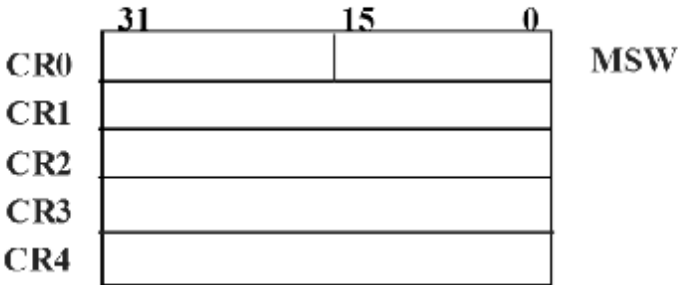
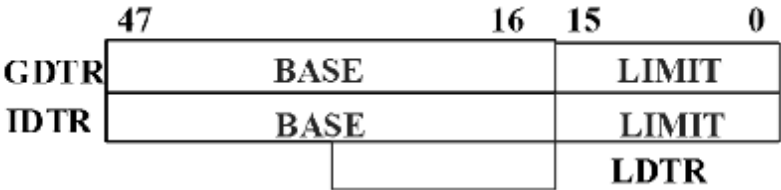
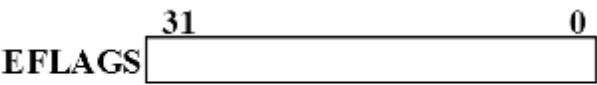
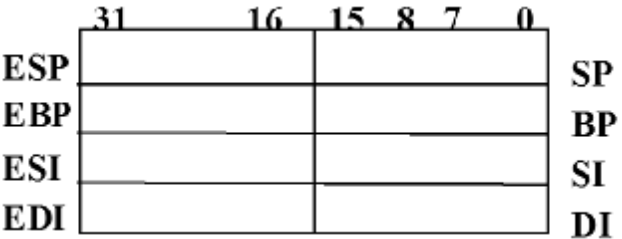
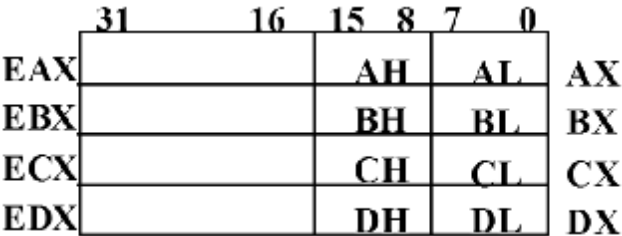
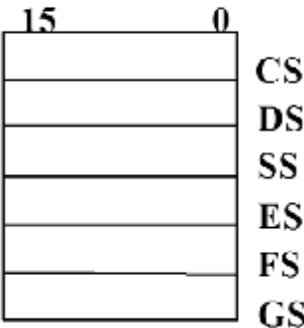
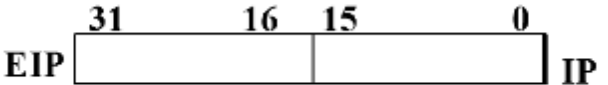
  - u CR0~CR3, GDTR, LDTR, IDTR, .....

# 实模式的寄存器模型

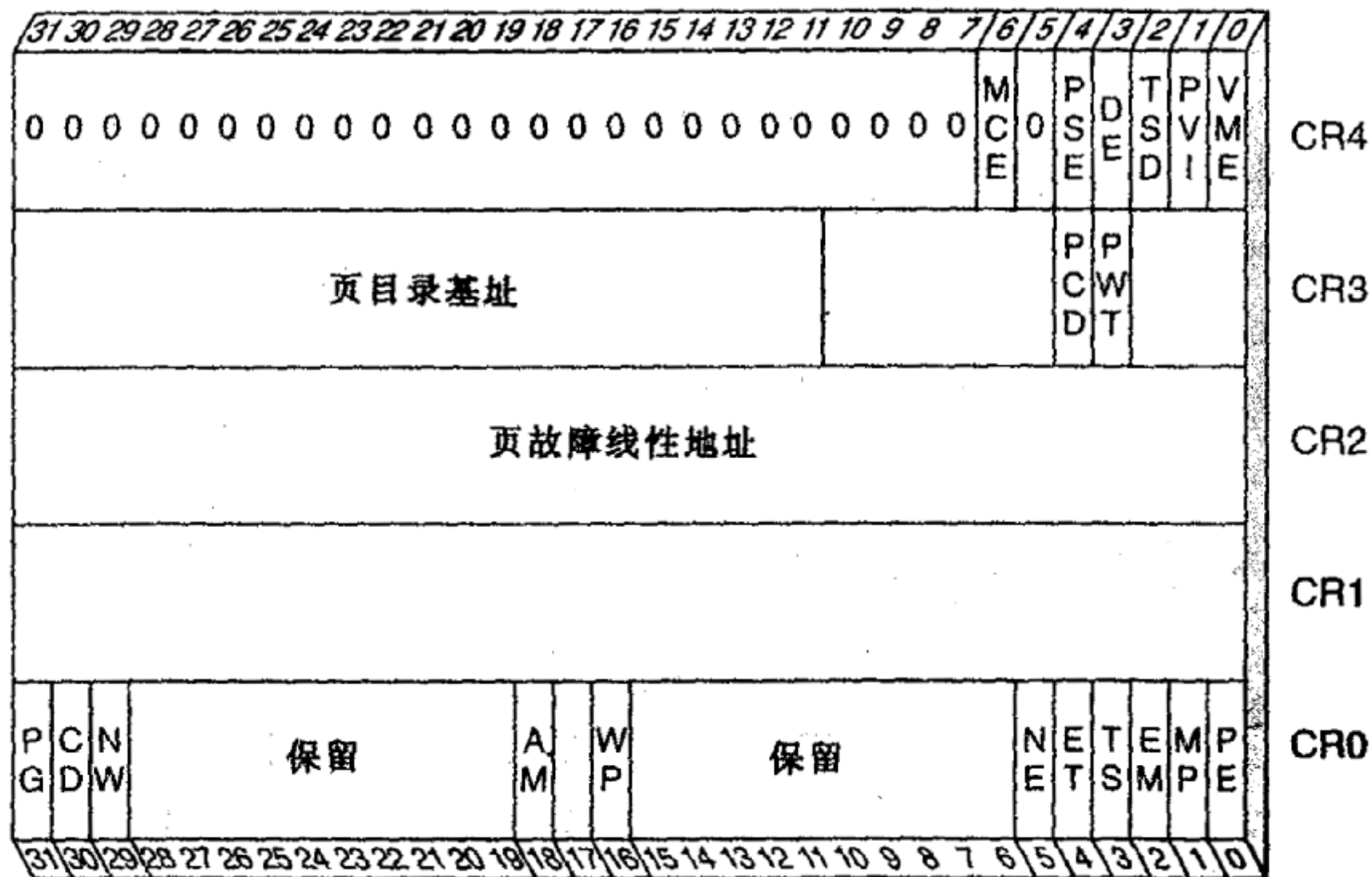




# 保护模式的寄存器模型



# 控制寄存器



# 控制寄存器CR0

## I CR0的低5位组成机器状态字（MSW）

**nPE:** 0——实模式；1——保护模式

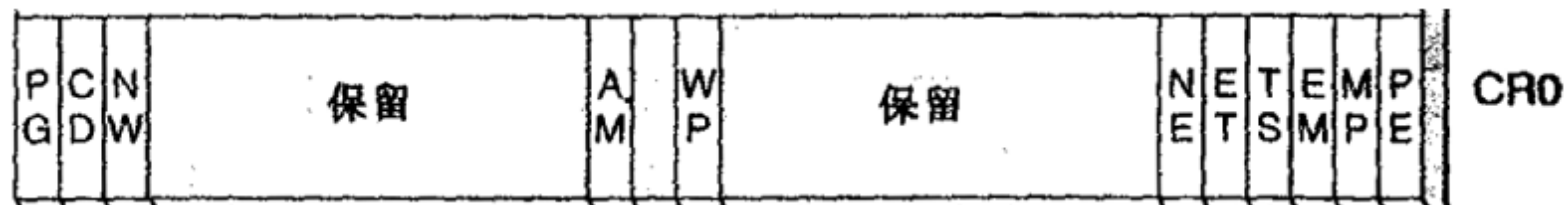
**nMP:** 1(系统有数学协处理器时)

**nEM:** 0（仿真协处理器）

**nTS:** 任务切换，切换任务时自动设置

**nET:** 1(协处理器的类型)

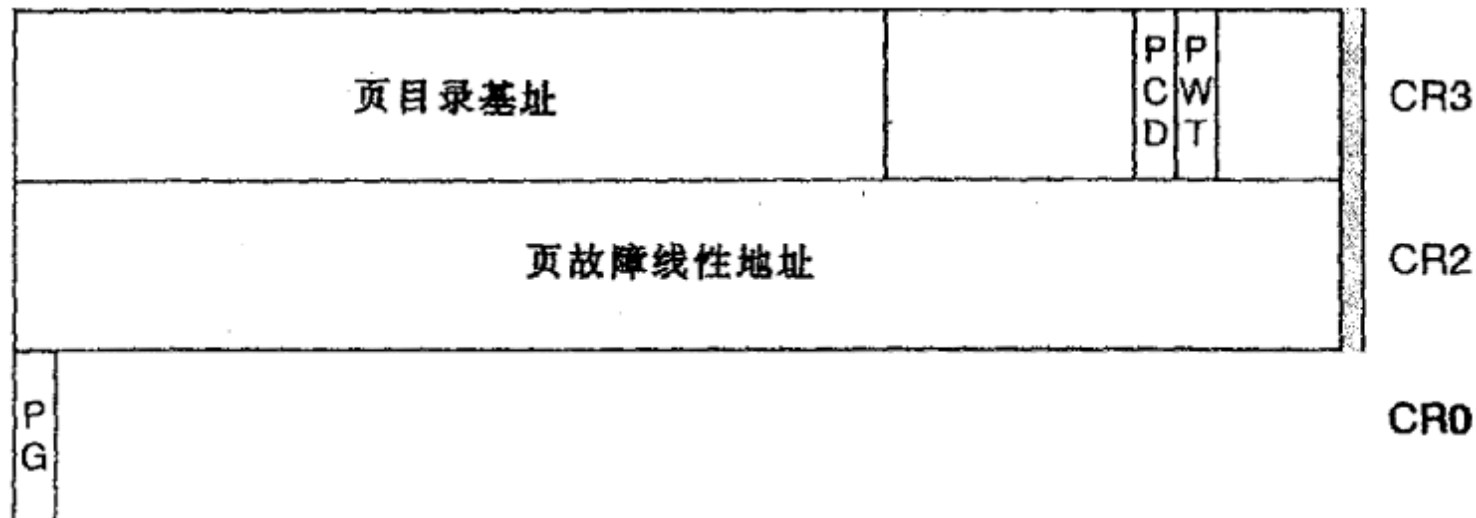
## I CR0中的PG位设置为1，表示允许分页



## I 控制寄存器CR3、CR2

**n**CR3包含页目录基址，指向页目录的开头

**n**如果发生缺页，则将发生缺页的地址保存在CR2中



# 权限管理

- I 管态

  - n0级（OS核心）、1级（OS）、2级(如组件)

- I 目态

  - n3级（应用程序）

- I RPL: 访问请求的权限

- I CPL: 当前执行任务的权限

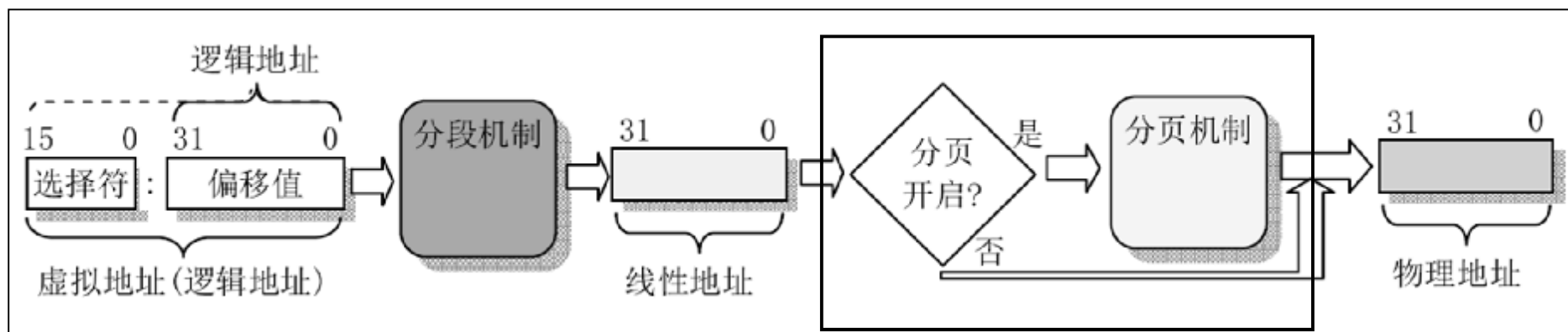
- I DPL: 被访问者权限

$$DPL \geq \text{MAX}(CPL, RPL)$$

## I i386段页式地址转换

n 第一级：段机制（逻辑地址到线性地址）

n 第二级：分页机制（线性地址到物理地址）



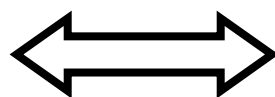
Intel x86 CPU 架构下的三种“地址”

n MMU：执行地址映射过程

# I Intel x86 CPU 架构下的三种“地址”

1、逻辑地址：汇编语言 (段:偏移)

```
mov BX, 1000H
mov DS, BX
mov AL, [10H]
```



1000H DS左移 4 位



10000H (实模式下)

+ 10H 加上段内偏移

2、线性地址：由逻辑地址转换得到

10010H

3、物理地址：未分页

线性地址 == 物理地址

分页（保护模式）线性地址 != 物理地址

保护模式下，DS不可能再直接放段的物理地址高16地址了！？

# MMU地址映射过程

## I MMU收到VA后

**n** 检查该段是否在内存中（“P”位）

**u** 在：段基址 + OFFSET  $\Rightarrow$  PA

**u** 否：产生“段不存在”异常，完成段调入

**n** 访问权限检查

**u** 如果  $RPL \geq DPL$ ，允许访问

**p** 否则，产生“权限违例”异常



# 段描述符（Descriptor）

## I 保护模式的段

**n**由段描述符（**Descriptor**）来描述， 8字节。

**u**段基址

**u**段界限

**u**段属性

**p**段类型

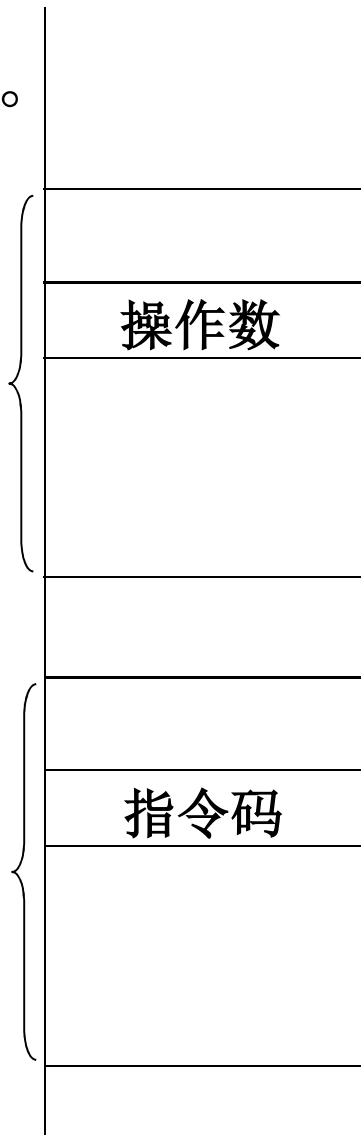
**p**访问该段所需的最小特权级

**p**是否在内存

**p**...

某个数据段

某个代码段



# I 段描述符 (Descriptor)

**n**段基址: 32位 (段基址1+段基址2)

**n**段界限: 20位 (段界限1+段界限2)

31..24 段基址1	属性	19..16 段界限2	23..0 段基址2	15..0 段界限1
----------------	----	----------------	---------------	---------------

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
G	D/B	0	AVL	段界限2				P	DPL		S	TYPE			

段的粒度: G=0, 段长以字节  
计量; G=1以页面4KB计量

描述符

类型: 描述符类型

等, 扩展,  
等及组合

## I 属性的描述

**nP: Present**, 是否在内存中【1在】

**nG: 段的粒度**（段长计量单位）

**uG=0**, 字节 (段最长1M)

**uG=1**, 页面4KB（段最长4G）

**nDPL: Descriptor Privilege Level** 描述符特权级别

**nS: 描述符的类型**

**u【数据段/代码段S=1】**

**u【系统描述符/门描述符S=0】**

**nTYPE: 描述符的存取类型**

**u【读，写，扩展，访问标志等及其组合】**

TYPE 值	数据段和代码段描述符 S=1	系统段和门描述符 S=0
0	只读	< 未定义 >
1	只读, 已访问	可用 286TSS
2	读 / 写	LDT
3	读 / 写, 已访问	忙的 286TSS
4	只读, 向下扩展	286 调用门
5	只读, 向下扩展, 已访问	任务门
6	读 / 写, 向下扩展	286 中断门
7	读 / 写, 向下扩展, 已访问	286 陷阱门
8	只执行	< 未定义 >
9	只执行、已访问	可用 386TSS
A	执行 / 读	< 未定义 >
B	执行 / 读、已访问	忙的 386TSS
C	只执行、一致码段	386 调用门
D	只执行、一致码段、已访问	< 未定义 >
E	执行 / 读、一致码段	386 中断门
F	执行 / 读、一致码段、已访问	386 陷阱门

# 段描述符表（Descriptor Table）

- I 描述符表（**Descriptor Table**）存放段描述符。
  - n 描述符表的字节数为8倍数。
- I 描述符表类型：
  - n 全局描述符表**GDT**: **Global Descriptor Table**
  - n 中断描述符表**IDT**: **Interrupt Descriptor Table**
  - n 局部描述符表**LDT**: **Local Descriptor Table**
  - n 任务状态描述符**TSS**: **Task State Stack**
- I 全局描述符表（**GDT**）：
  - n 包含所有进程可用的段描述符。系统仅1个**GDT**
- I 局部描述符表（**LDT**）
  - n 进程有关的描述符，每个进程都有各自**LDT**
- I **GDT/IDT**的基址存放在寄存器**GDTR/IDTR**中。

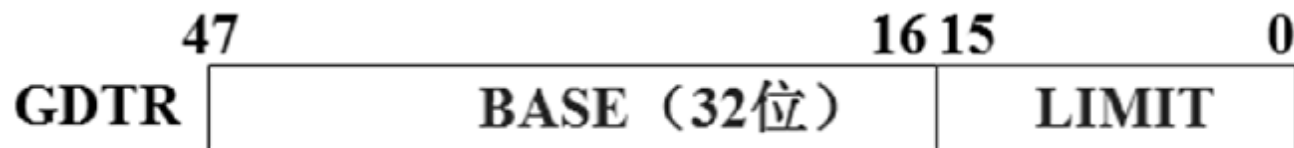
# GDTR/LDTR/IDTR/TR四个寄存器

## I GDTR/LDTR/IDTR/TR

**n**存放GDT/IDT基址和大小的寄存器

**n**存放LDT/TSS的选择子的寄存器

**n**GDTR（48位）



**u**BASE: 指示GDT在物理存储器中开始的位置

**u**LIMIT: 规定GDT的界限.

**p**LIMIT有16位，GDT最大65536个字节，最大的GDT能够容纳 $65536/8=8192$ 个描述符

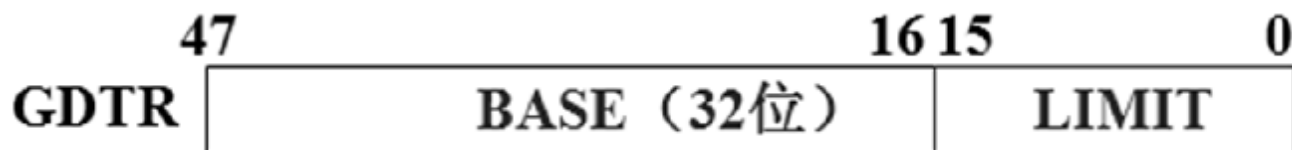
I 例: (GDTR)=001000000FFFH, 求GDT在物理存储器中的起始地址, 结束地址, 表的大小, 表中可以存放多少个描述符?

解: 起始地址: 0010 0000H

结束地址: 0010 0000H+0FFFH=0010 0FFFH

表的大小 = 0FFFH+1=4096字节

表中可以存放描述符数量 =  $4096 / 8 = 512$ 个



# 段描述符（Descriptor）的实现

```
typedef struct Descriptor
{
    unsigned int base24_31      :8    //地址的高8位
    unsigned int g              :1    //段长单位， 0:字节， 1:4K
    unsigned int d_b            :1
    unsigned int unused         :1
    unsigned int avl            :1
    unsigned int seg_limit_16_19 :4    //段界限高4位
    unsigned int p              :1
    unsigned int dpl            :1
    unsigned int s              :1
    unsigned int type           :4
    unsigned int base_0_23      :24    //地址的低24位
    unsigned int seg_limit_0_15 :16    //段界限低16位
}
```



# 中断描述符表寄存器IDTR（48位）

## I 在物理存储器地址空间中定义中断描述符表IDT



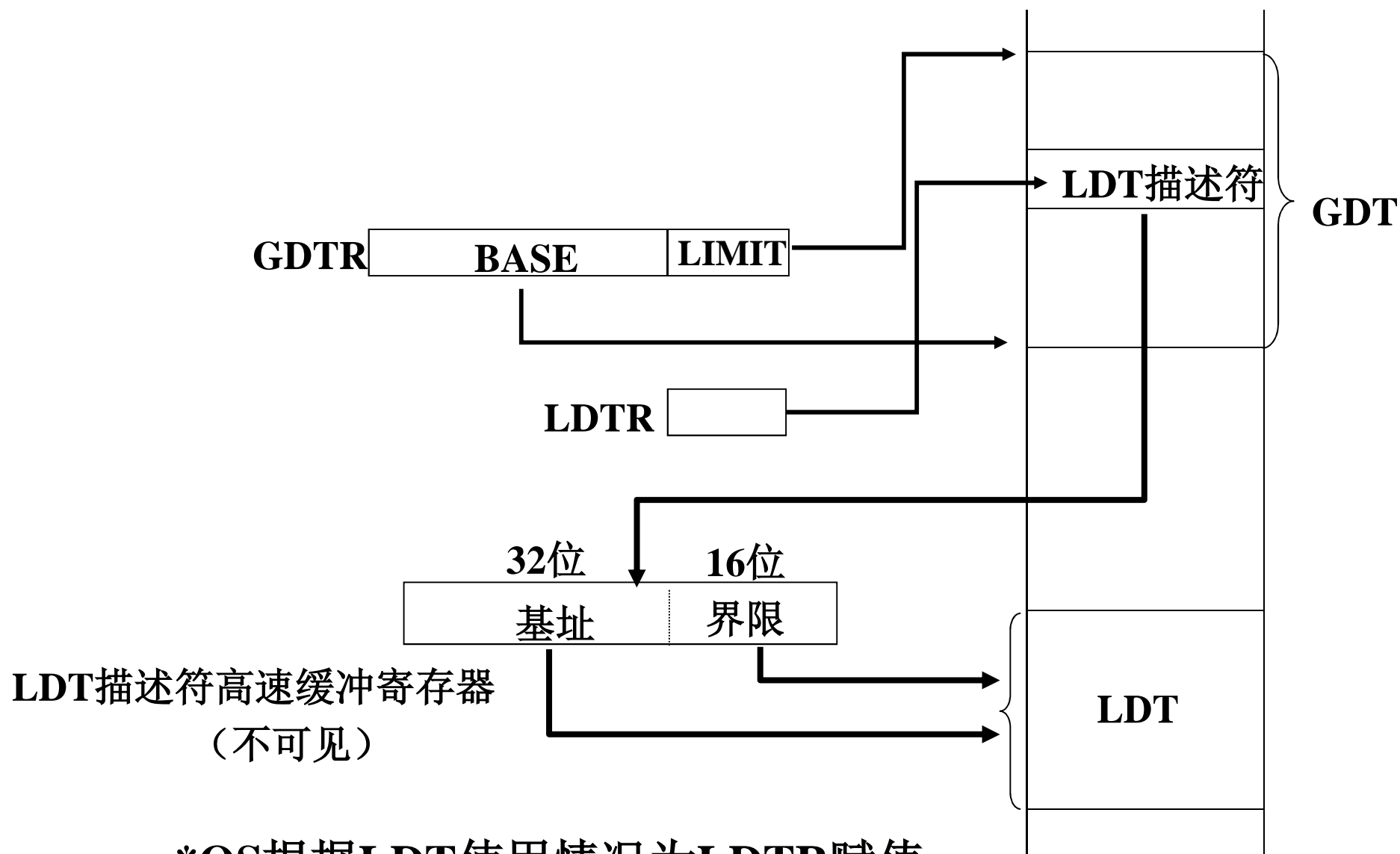
n 由于Pentium只能支持256个中断和异常，因此LIMIT最大为（）？

n IDT中的描述符类型为中断门

## 局部描述符表寄存器LDTR（16位）

- 丨 LDT定义任务用到的局部存储器地址空间
- 丨 16位的LDTR并不直接定义LDT的基址和长度，它只是一个指向GDT中LDT描述符的选择子。
- 丨 如果LDTR中装入了选择子，相应的描述符将从GDT中读出（仅读出基址和限长）并装入局部描述符表高速缓冲寄存器。将该描述符装入高速缓冲寄存器就为当前任务创建了一个LDT.

# 为当前任务创建LDT的过程



\*OS根据LDT使用情况为LDTR赋值

# 任务寄存器TR

- 丨 存放16位的选择子，指示全局描述符表中任务状态段（TSS）描述符的位置
- 丨 当选择子装入TR时，相应的TSS描述符自动从存储器中读出并装入任务描述符缓冲寄存器。该描述符定义了一个称为任务状态段（TSS）的存储块。每个任务都有TSS，TSS包含启动任务所必需的信息。
- 丨 TSS最大64K字节

# 段选择子 (Selector)

- 段/段描述符的选择，指向**GDT/LDT**中的某个段。
- 一个**13位**的整数形式的索引，存放在段寄存器中。



保护模式下，**DS**不可能再直接放段的物理地址高**16**地址了！？

## 构成

13位索引

**n**索引域 (INDEX) : 13位

**n**TI域 (Table Indicator) : 1位

**n**特权级别域 (Request Privilege Level) : 2位

# 段选择子（Selector）

## I TI 域（Table Indicator）

**n**TI = 1，从LDT中选择相应描述符，

**n**TI = 0，从GDT中选择描述符。

## I 索引域（INDEX）

**n**给出段描述符在GDT或LDT中的位置。

## I 特权级别域（Request Privilege Level）

**n**请求者最低特权级的限制。

**n**只有请求者特权级高于或等于DPL，对应段才能被存取，实现段的保护。

I 例：设LDT的基址为0012 0000H， GDT的基址为00100000H，  
(CS)=1007H， 那么：

n ①请求的特权级是多少

n ②该段描述符位于GDT中还是LDT中

n ③该段描述符的基地址是多少

解： (CS)=1007H=0001 0000 0000 0111B

① RPL=3， 申请的特权级为3

② TI=1， 描述符位于LDT中

③ 描述符相对于LDT基址的偏移量为

0001 0000 0000 0B ´ 8=512 ´ 8=4096=1000H

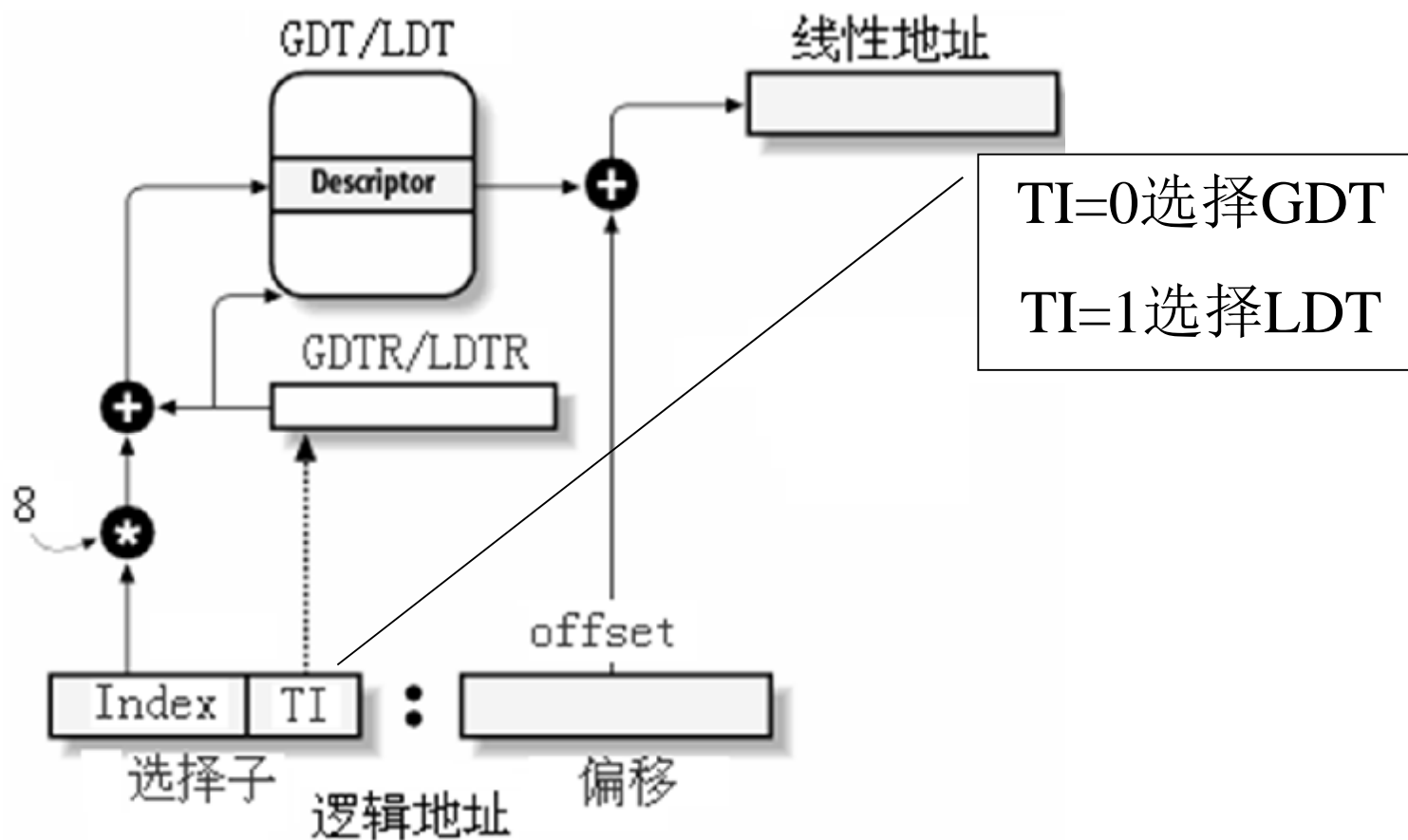
段描述符的地址为

0012 0000H+1000H=00121000H

15	3	2	1	0
选择子		TI	RPL	

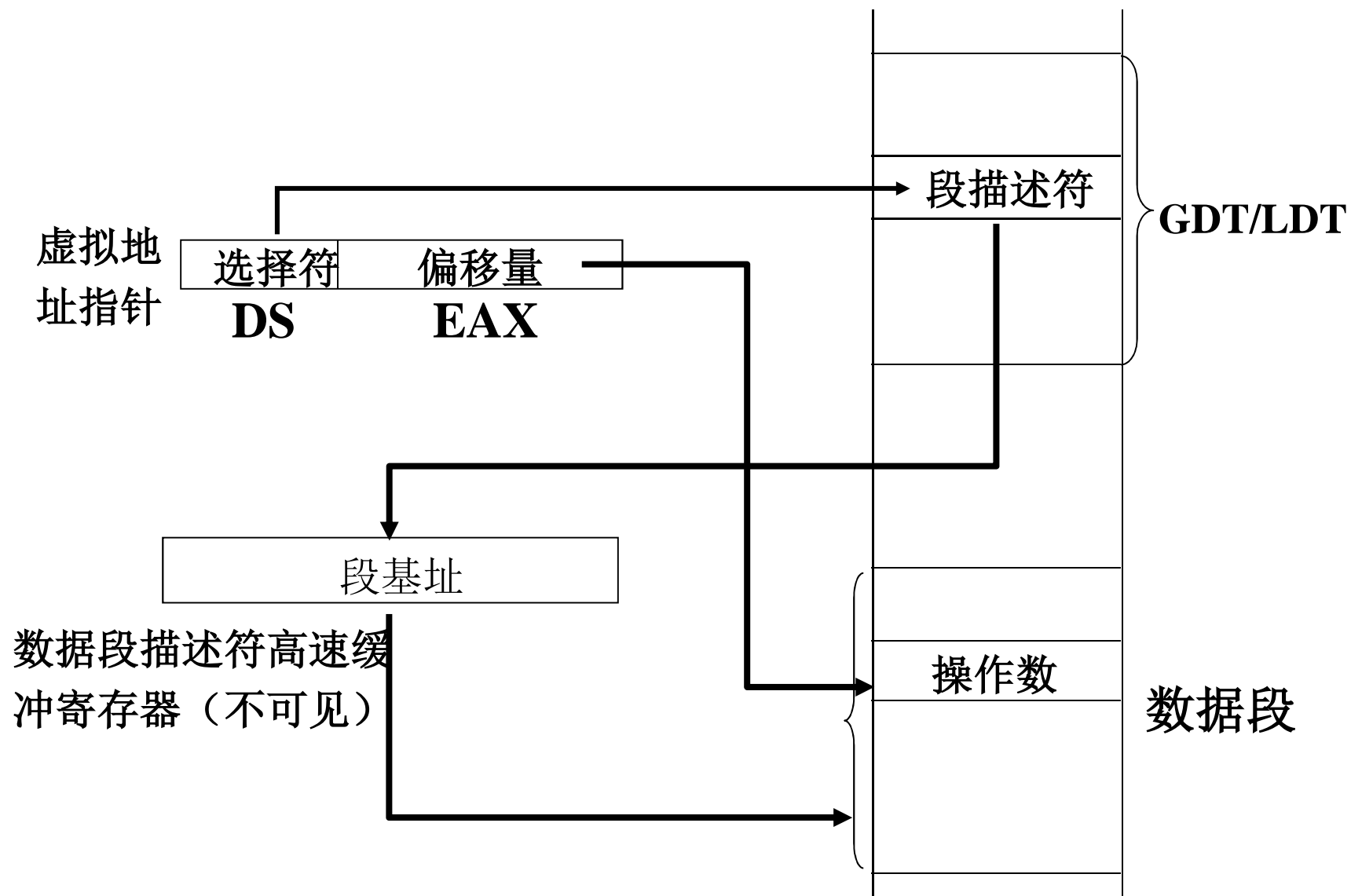
## I 段机制的功能

n 把逻辑地址转换到线性地址（32位，4G）。





# 段式地址转换



I 例：假设虚拟地址为**0100:0000 0200H**，禁止分页。  
如果描述符中读出的段基址为**0003 0000H**，那么操作数的物理地址是什么？

解：

将此虚拟地址转换成物理地址

= 基地址 + 偏移量

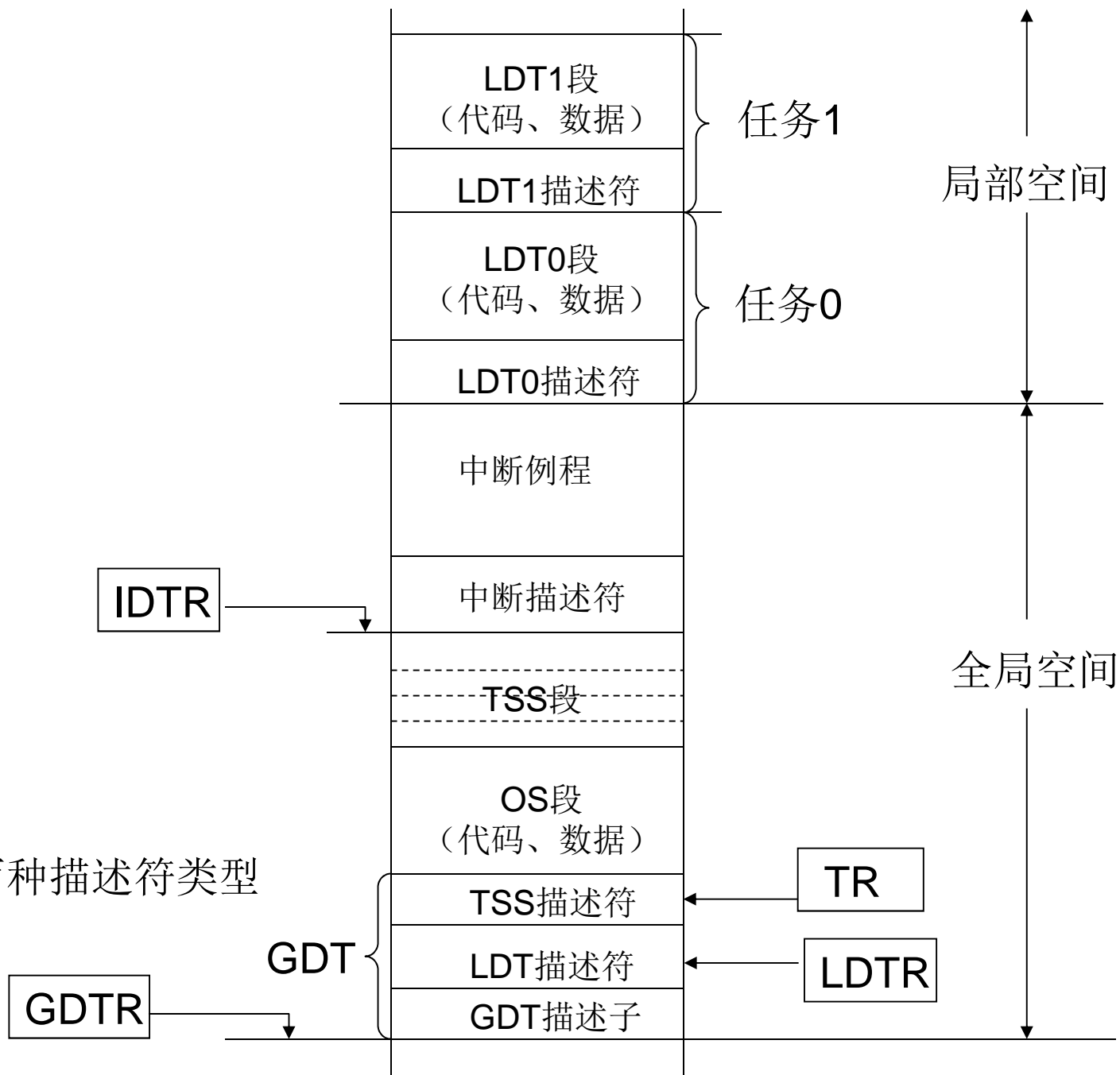
= 00030000H + 00002000H

= 0003 2000H

虚拟地址的冒号前面的**0100**起到了什么作用？

# 存储空间分配

注意：存在两种描述符类型



# 代码实例分析

## I 用来生成描述符的宏

**n** 这个宏表示的不是一段代码，而是一个数据结构，它的大小是8字节

```
251 ; 描述符
252 ; usage: Descriptor Base, Limit, Attr
253 ;      Base:  dd
254 ;      Limit: dd (low 20 bits available)
255 ;      Attr:  dw (lower 4 bits of higher byte are always 0)
256 %macro Descriptor 3
257     dw  %1 & 0FFFFh                ; 段基址1
258     dw  %2 & 0FFFFh                ; 段界限1
259     db  (%1>>16) & 0FFh            ; 段基址2
260     dw  ((%2>>8) & 0F00h) | (%3 & 0F0FFh) ; 属性1+段界限2+属性2
261     db  (%1>>24) & 0FFh            ; 段基址3
262 %endmacro; 共 8 字节
```

- I 数组的名字叫做GDT
- I GdtLen是GDT的长度
- I GdtPtr数据结构：有6字节，前2字节是GDT的界限， 后4字节是GDT的基地址。

```

11  [SECTION .gdt]
12  ; GDT
13  ;
14  LABEL_GDT:      Descriptor      0,          0, 0          ; 空描述符
15  LABEL_DESC_CODE32: Descriptor      0, SegCode32Len - 1, DA_C + DA_32; 非一致代码段
16  LABEL_DESC_VIDEO: Descriptor 0B8000h,          0ffffh, DA_DRW      ; 显存首地址
17  ; GDT 结束
18
19  GdtLen          equ             $ - LABEL_GDT          ; GDT长度
20  GdtPtr          dw             GdtLen - 1              ; GDT界限
21                dd             0                        ; GDT基地址
22
23  ; GDT 选择子
24  SelectorCode32  equ            LABEL_DESC_CODE32      - LABEL_GDT
25  SelectorVideo   equ            LABEL_DESC_VIDEO       - LABEL_GDT
26  ; END of [SECTION .gdt]

```

## I [BITS 16]指明是一个16位代码段

n 修改了一些GDT中的值

n 最后在第71行通过jmp指令跳转到保护模式

```
28  [SECTION .s16]
29  [BITS 16]
30  LABEL_BEGIN:
31      mov     ax, cs
32      mov     ds, ax
33      mov     es, ax
34      mov     ss, ax
35      mov     sp, 0100h
36
37      ; 初始化 32 位代码段描述符
38      xor     eax, eax
39      mov     ax, cs
40      shl     eax, 4
41      add     eax, LABEL_SEG_CODE32
42      mov     word [LABEL_DESC_CODE32 + 2], ax
43      shr     eax, 16
44      mov     byte [LABEL_DESC_CODE32 + 4], al
45      mov     byte [LABEL_DESC_CODE32 + 7], ah
```

```

47      ; 为加载GDTR 作准备
48      xor          eax, eax
49      mov          ax, ds
50      shl          eax, 4
51      add          eax, LABEL_GDT          ; eax <- gdt 基地址
52      mov          dword [GdtPtr + 2], eax ; [GdtPtr + 2] <- gdt 基地址
53
54      ; 加载  GDTR
55      lgdt         [GdtPtr]
56
57      ; 准备切换到保护模式
58      mov          eax, cr0
59      or           eax, 1
60      mov          cr0, eax
61
62      ; 真正进入保护模式
63      jmp          dword SelectorCode32:0 ; 执行这一句会把SelectorCode32 装入cs,
64                                          ; 并跳转到Code32Selector:0 处
65
66      ; END of [SECTION .s16]

```

```

76  [SECTION .s32];      32 位代码段. 由实模式跳入.
77  [BITS 32]
78
79  LABEL_SEG_CODE32:
80      mov     ax, SelectorVideo
81      mov     gs, ax          ; 视频段选择子(目的)
82
83      mov     edi, (80 * 11 + 79) * 2 ; 屏幕第11行, 第79列。
84      mov     ah, 0Ch          ; 0000:黑底    1100:红字
85      mov     al, 'P'
86      mov     [gs:edi], ax;
87
88      ; 到此停止
89      jmp     $
90
91  SegCode32Len     equ     $ - LABEL_SEG_CODE32
92  ; END of [SECTION .s32]

```



# I 描述符类型

201	; 存储段描述符类型			
202	DA_DR	<b>EQU</b>	90h	; 存在的只读数据段类型值
203	DA_DRW	<b>EQU</b>	92h	; 存在的可读写数据段属性值
204	DA_DRWA	<b>EQU</b>	93h	; 存在的已访问可读写数据段类型值
205	DA_C	<b>EQU</b>	98h	; 存在的只执行代码段属性值
206	DA_CR	<b>EQU</b>	9Ah	; 存在的可执行可读代码段属性值
207	DA_CCO	<b>EQU</b>	9Ch	; 存在的只执行一致代码段属性值
208	DA_CCOR	<b>EQU</b>	9Eh	; 存在的可执行可读一致代码段属性值
209				
210	; 系统段描述符类型			
211	DA_LDT	<b>EQU</b>	82h	; 局部描述符表段类型值
212	DA_TaskGate	<b>EQU</b>	85h	; 任务门类型值
213	DA_386TSS	<b>EQU</b>	89h	; 可用386任务状态段类型值
214	DA_386CGate	<b>EQU</b>	8Ch	; 386调用门类型值
215	DA_386IGate	<b>EQU</b>	8Eh	; 386中断门类型值
216	DA_386TGate	<b>EQU</b>	8Fh	; 386陷阱门类型值

# LDT

## I 一个局部的代码段

```
283 ; CodeA (LDT, 32 位代码段)
284 [SECTION .la]
285 ALIGN 32
286 [BITS 32]
287 LABEL_CODE_A:
288     mov ax, SelectorVideo
289     mov gs, ax ; 视频段选择子(目的)
290     mov edi, (80* 12 + 0) * 2 ; 屏幕第 10 行, 第 0 列。
291     mov ah, 0Ch ; 0000:黑底 1100: 红字
292     mov al, 'L'
293     mov [gs:edi], ax
294
295
296 ; 准备经由16位代码段跳回实模式
297     jmp SelectorCode16:0
298 CodeALen equ $ - LABEL_CODE_A
299 ; END of [SECTION .la]
```

# LDT

## I 填充LDT

```
269 ; LDT
270 [SECTION .ldt]
271 ALIGN 32
272 LABEL_LDT:
273 ;                段基址      段界限      属性
274 LABEL_LDT_DESC_CODEA: Descriptor 0, CodeALen - 1, DA_C + DA_32 ; Code, 32位
275
276 LDTLen          equ      $ - LABEL_LDT
277
278 ; LDT 选择子
279 SelectorLDTCodeA equ    LABEL_LDT_DESC_CODEA - LABEL_LDT + SA_TIL
280 ; END of [SECTION .ldt]
```

- I 在GDT中增加了一个LDT描述符，并增加与与对应的选择子
- I 增加对LDT描述符的初始化代码。

```
64  [SECTION .s16]
    ...
116      ; 初始化 LDT 在 GDT 中的描述符
117      xor     eax, eax
118      mov     ax, ds
119      shl     eax, 4
120      add     eax, LABEL_LDT
121      mov     word [LABEL_DESC_LDT+2], ax
122      shr     eax, 16
123      mov     byte [LABEL_DESC_LDT+4], al
124      mov     byte [LABEL_DESC_LDT+7], ah
125
126      ; 初始化 LDT 中的描述符
127      xor     eax, eax
128      mov     ax, ds
129      shl     eax, 4
130      add     eax, LABEL_CODE_A
131      mov     word[LABEL_LDT_DESC_CODEA+2], ax
132      shr     eax, 16
133      mov     byte[LABEL_LDT_DESC_CODEA+4], al
134      mov     byte[LABEL_LDT_DESC_CODEA+7], ah
```

- I 指令**lldt**，负责加载**ldtr**，它的操作数是一个选择子，这个选择子对应的就是用来描述**LDT**的描述符（标号**LABEL\_DESC\_LDT**）。

```
183  [SECTION .s32]; 32 位代码段. 由实模式跳入.  
    ...  
216      ; Load LDT  
217      mov     ax,SelectorLDT  
218      lldt     ax  
219  
220      jmp     SelectorLDTCodeA:0      ; 跳入局部任务
```

# 硬件分页

## I 页 vs. 页框

- n 页：虚拟页，可在主存也可在磁盘

- n 页框：物理页，RAM块

## I 分页

- n Intel CPU的页：4KB

- n 通过设置CR3的PG位开启分页功能

- n 分页发生在物理地址送地址线之前：线性地址→物理地址

- n 在MMU中进行分页

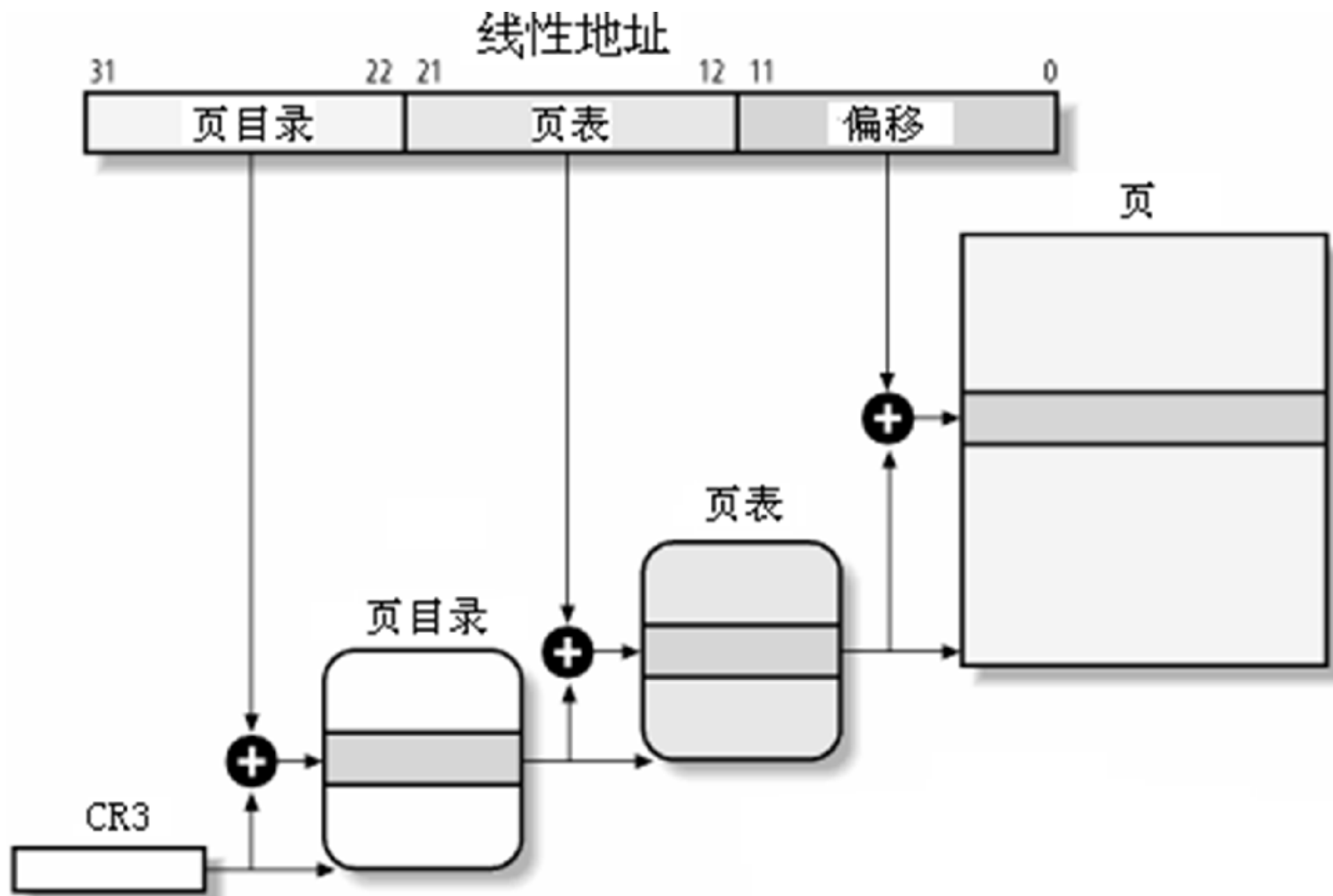
## I 数据结构

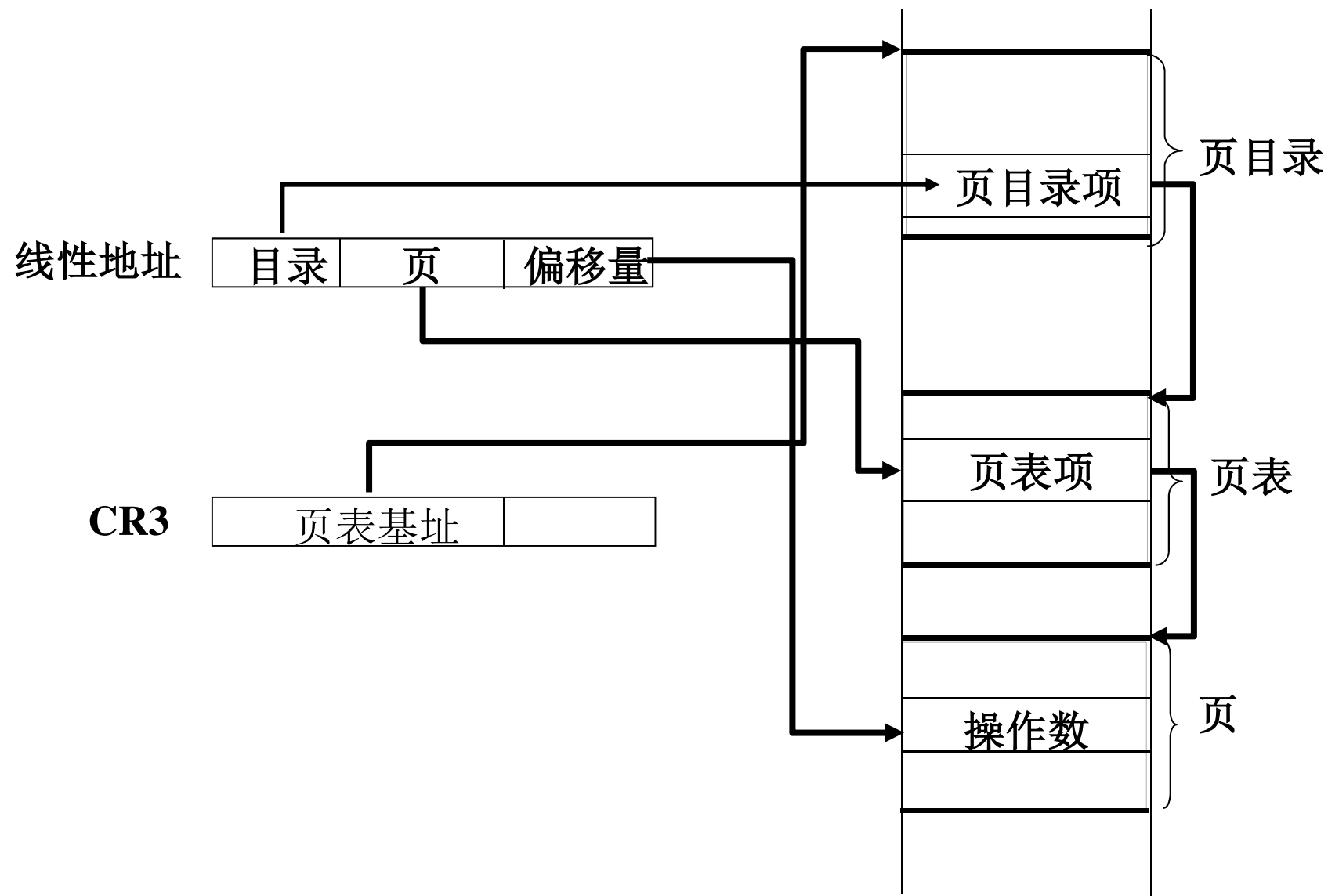
- n 页表

- n 页目录

- u 当前页目录的物理基地址在CR3中

## 二级页表结构 (Windows)







# 页目录表

- 页目录长**4KB**，包含最多**1024**个页目录项，每个页目录项**4**字节。
- 页表地址：页表的起始地址是**4K**的整数倍，因此**32**位地址的低**12**位总为**0**，用高**20**位表示即可，即页目录表中给出的是页基址的高**20**位。

7	6	5	4	3	2	1	0	
0	0	A	0	0	U/S	R/W	P	0
页表地址0~3 位				OS 专用			0	1
页表地址4~11 位								2
页表地址12~19 位								3

7	6	5	4	3	2	1	0	
0	0	A	0	0	U/S	R/W	P	0
页表地址0~3 位				OS 专用			0	1
页表地址4~11 位								2
页表地址12~19 位								3

	U/S	R/W	特权级3	特权级0~2
user {	0	0	无	读/写
	0	1	无	读/写
supervisor {	1	0	只读	读/写
	1	1	读/写	读/写

## 二级页表结构（Windows）

┆ 页目录：10bit；页表：10bit；偏移：12bit。

┆ 线性地址结构

```
typed struct
```

```
{
```

```
    // 表示页目录的下标，该表项指向一个页表
```

```
    unsigned int dir :10;
```

```
    // 表示页表的下标，该表项指向一个页框
```

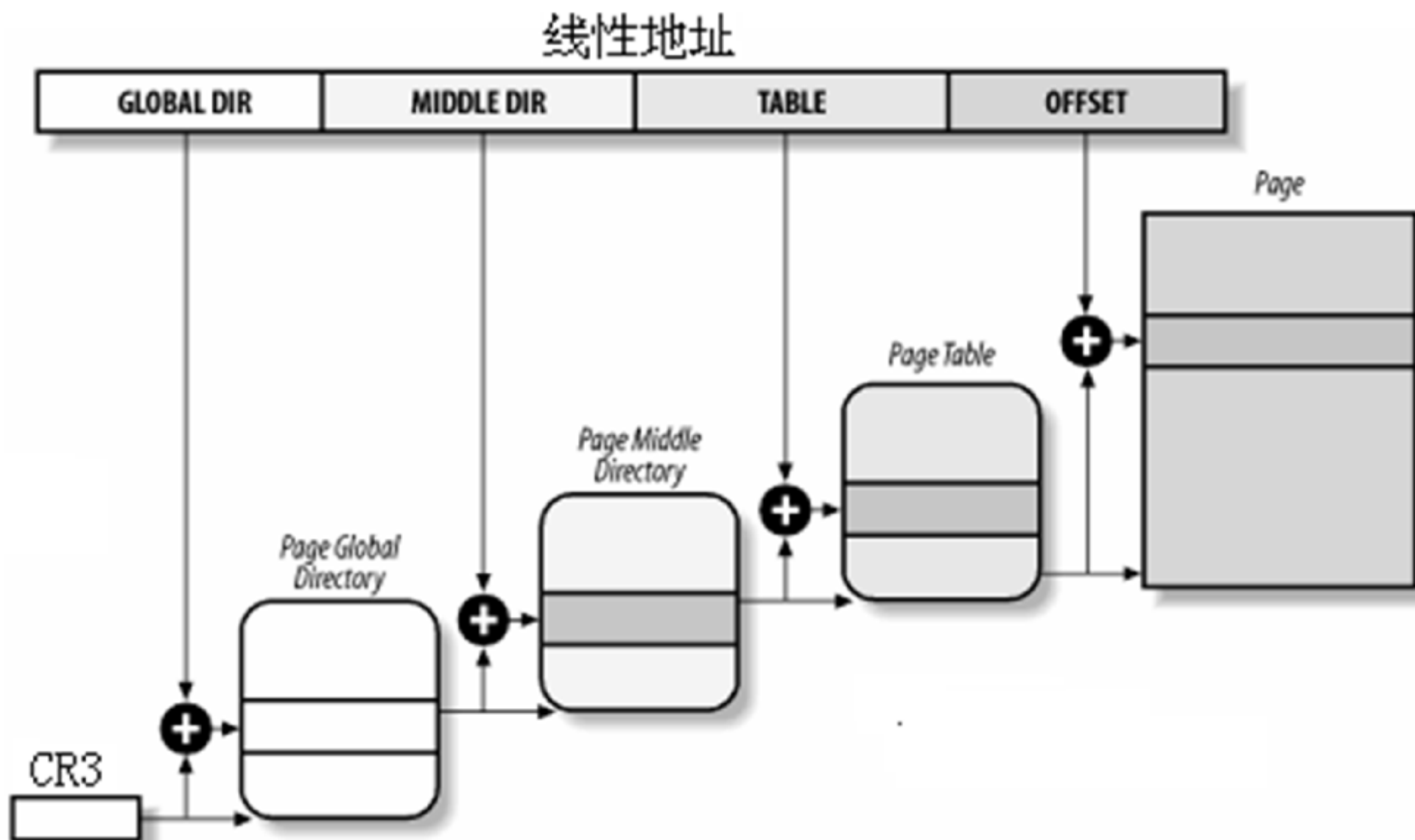
```
    unsigned int page: 10;
```

```
    // 在4K字节页框内的偏移量
```

```
    unsigned int offset :12;
```

```
}线性地址
```

# Linux的三级页表结构



# Linux的三级页表结构

## I 与体系结构无关的三级页表结构

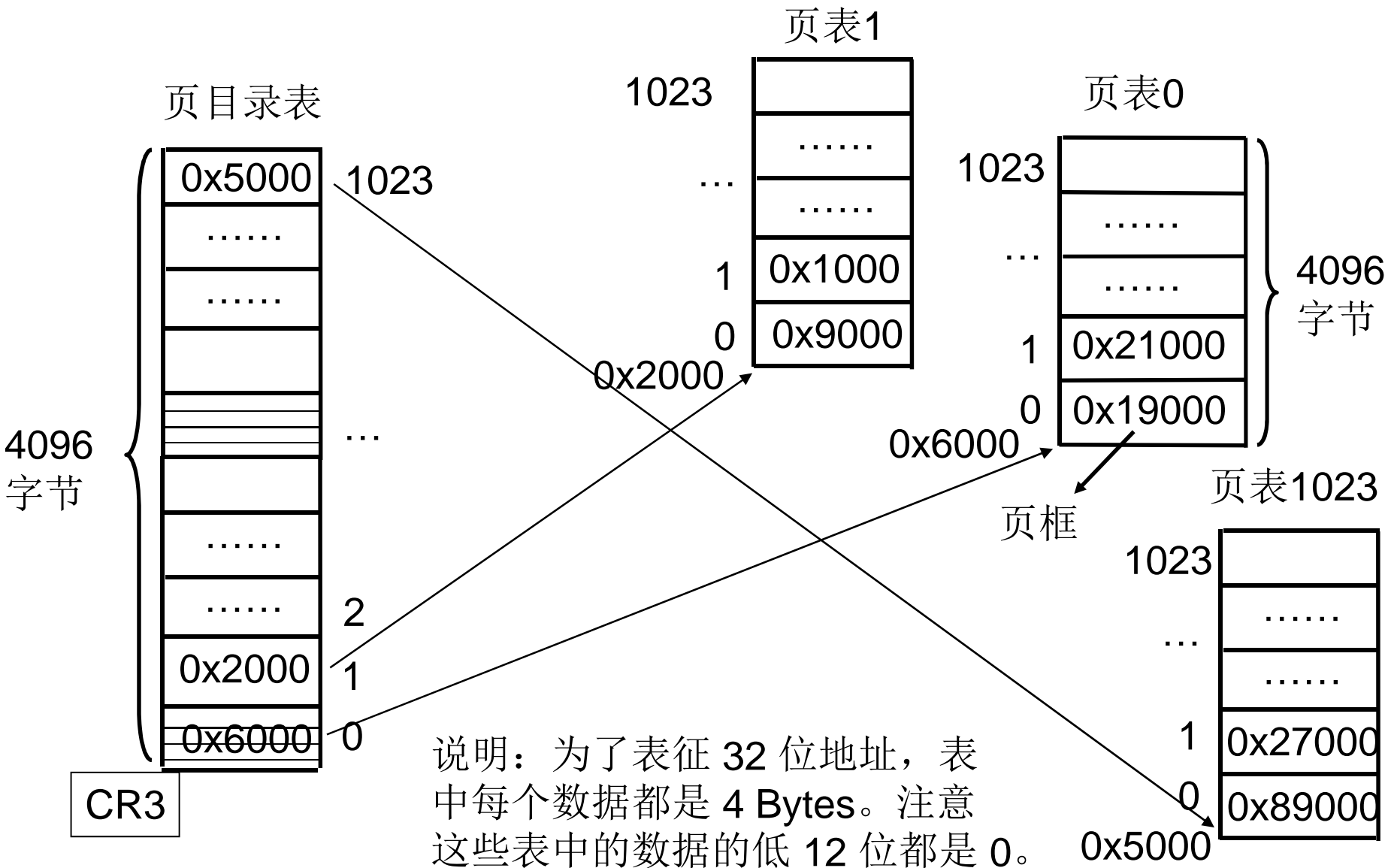
**npgd**, 页全局目录 (page global directory)

**npmd**, 页中级目录 (page middle directory)

**npte**, 页表项 (page table entry)

**noffset**, 偏移

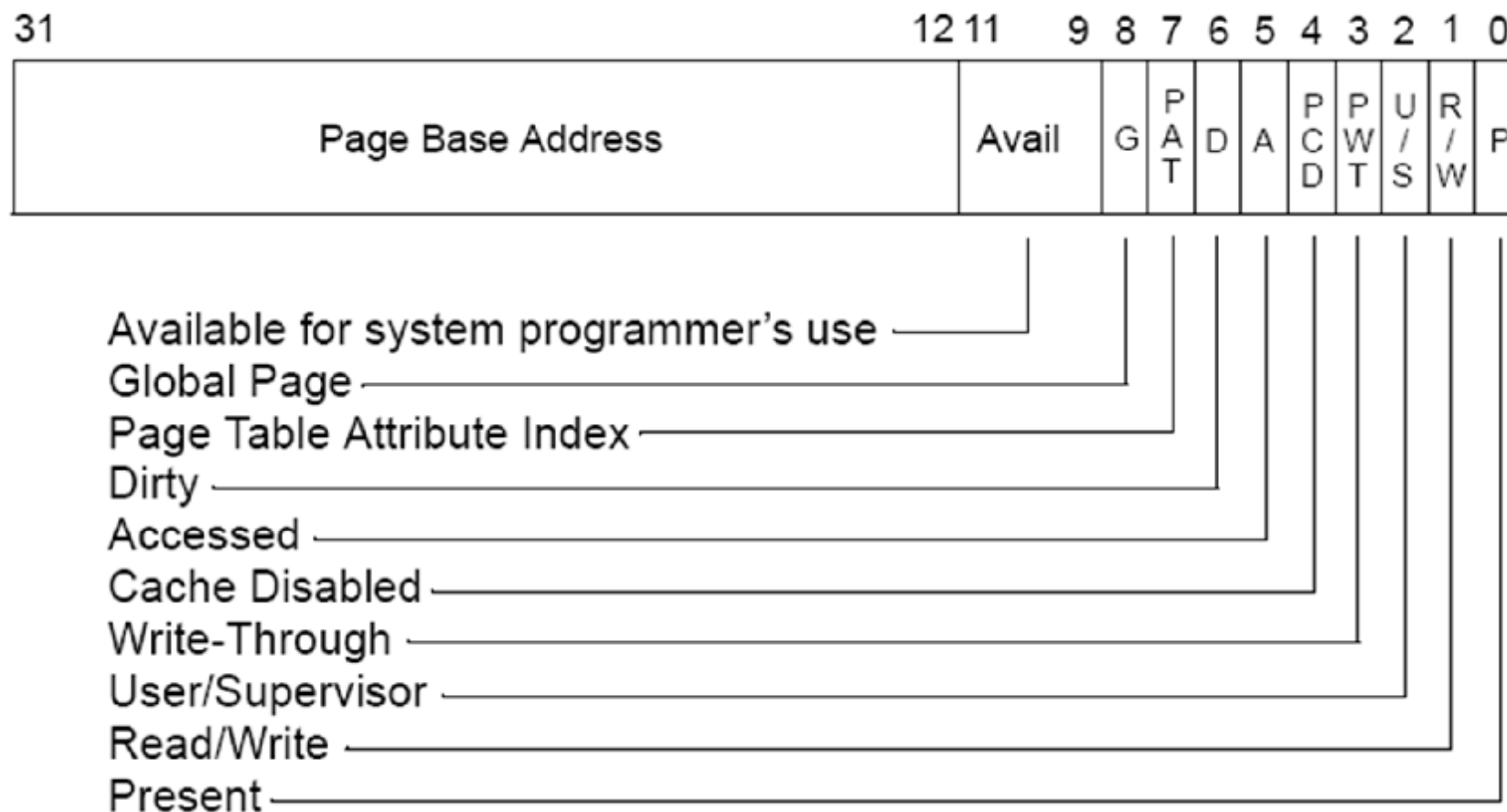
# 页目录表和页表的关系



# 页表中的数据项

## I 低12 位进行了重定义

Page-Table Entry (4-KByte Page)



P:1:表示该页在内存，0:表示在硬盘上

# Linux虚拟内存的组织

## I Linux将4G虚拟空间划分为两个部分

- n 用户空间与内核空间

- n 用户空间3G：从0到0xBFFFFFFF

- n 内核空间1G：从0xC0000000到4G。

- n 用户进程通常情况下只能访问用户空间的虚拟地址，不能访问内核空间。例外情况是用户进程通过系统调用访问内核空间。



# Linux的段机制

## I Linux四个范围一样的段：0 ~ 4 G

- n 内核数据段

- n 内核代码段

- n 用户数据段

- n 用户代码段

## I 各段属性不同

- n 内核段特权级为0

- n 用户段特权级为3

## I 作用

- n 利用段机制来隔离用户数据和系统数据

- n 简化（避免）逻辑地址到线性地址转换；

- n 保留段的等级保护机制

# Linux的段机制

I 进程建立时段机制对寄存器的初始化start\_thread

I **include/asm-i386/processor.h**

```
#define start_thread(regs, new_eip, new_esp) do{           \  
    __asm__ (“movl %0, %%fs; movl %0, %%gs”: :”r”(0));  
    set_fs(USER_DS);  
    regs->xds = __USER_DS;  
    regs->xes = __USER_DS;  
    regs->xss = __USER_DS;  
    regs->xcs = __USER_CS;  
    regs->eip = new_eip;  
    regs->esp = new_esp;  
} while(0)
```

# I include/asm-i386/segment.h

#define \_\_KERNEL\_CS 0x10

#define \_\_KERNEL\_DS 0x18

#define \_\_USER\_CS 0x23

#define \_\_USER\_DS 0x2B

15	3	2	1	0
选择子				TI
				RPL

宏	值	INDEX	TI	DPL
__KERNEL_CS	0x10	0000 0000 0001 0	0	00
__KERNEL_DS	0x18	0000 0000 0001 1	0	00
__USER_CS	0x23	0000 0000 0010 0	0	11
__USER_DS	0x2B	0000 0000 0010 1	0	11

I INDEX: 2,3,4,5; TI = 0; DPL: 0,3.

# I arch/i386/kernel/head.S

## nGDT定义

ENTRY(gdt\_table)

.quad 0x 0000 0000 0000 0000	//NULL descriptor
.quad 0x 0000 0000 0000 0000	// not used
.quad 0x 00cf 9a00 0000 ffff	// Index=2, kernel 4GB code at 0x0
.quad 0x 00cf 9200 0000 ffff	// Index=3, kernel 4GB data at 0x0
.quad 0x 00cf fa00 0000 ffff	// Index=4, user 4GB code at 0x0
.quad 0x 00cf f200 0000 ffff	// Index=5, user 4GB data at 0x0
.quad 0x 0000 0000 0000 0000	// not used
.quad 0x 0000 0000 0000 0000	// not used

## I 4个GD（全局描述符）

位置    xxxx xxxx G100 hhhh P010 1010    xxxx xxxx    xxxx    xxxx xxxx xxxx hhhh hhhh hhhh hhhh

---

K\_CS:0000 0000 1100 1111 1001 1010 0000 0000    0000 0000 0000 0000 1111 1111 1111 1111  
K\_DS:0000 0000 1100 1111 1001 0010 0000 0000    0000 0000 0000 0000 1111 1111 1111 1111  
U\_CS:0000 0000 1100 1111 1111 1010 0000 0000    0000 0000 0000 0000 1111 1111 1111 1111  
U\_DS:0000 0000 1100 1111 1111 0010 0000 0000    0000 0000 0000 0000 1111 1111 1111 1111

I XXXX:基地址； hhhh: 段界限

I **G**位都是**1**（段长单位**4KB**）； **P**位都是**1**（段在内存）

I K\_CS(Index=2), kernel 4GB code at 0x0

I K\_DS(Index=3), kernel 4GB data at 0x0

I U\_CS(Index=4), USER 4GB code at 0x0

I U\_DS(Index=5), USER 4GB data at 0x0

# Linux的段机制

## I 结论

- n** 每个段是从0地址开始的**4GB**的虚拟空间，线性地址和虚地址保持一致。
- n** 讨论和理解**LINUX**内核页式映射时，可以直接将虚拟地址当做线性地址，二者完全一致。
- n** 段仅仅提供了一种保护机制。

# Linux的物理内存管理

- I 减少存储空间中的空洞，减少碎片，增加利用率
- I 具体的方法
  - n 外部碎片——伙伴算法(buddy)
  - n 内部碎片——slab缓存
  - n 非连续内存区内存的分配——vmalloc( )

# 伙伴算法(buddy)

## I 伙伴算法(buddy)

**n**在linux中，频繁地请求和释放不同大小的一组连续页框，必然导致在已分配页框的块内分散了许多小块的空闲页框，即所谓的外部碎片。

**n**为此linux建立了一种健壮、高效的分配策略——伙伴算法，来解决外部碎片问题。



## I 伙伴算法

n 1、以页面为单位，按照  $2^n$  个页面为单位进行划分 ( $n=0\sim11$ )

u 进行11次划分，采用11个块链表分别包含大小为1, 2, 4, 8, 16, 32, 64, 128, 256, 512和1024 个连续的页框。

u 划分后形成大小不同的存储块，称为页面块（页块）。

u 包含1个页面的块称为1页块，包含1个页面的块称为2页块， ...

n 2、同名的页块按先后顺序两两结合成一对**buddy**伙伴

u 1页块中：0和1、2和3、4和5、...：1页块**buddy**伙伴

u 2页块中：0~1和2~3、4~5和6~7、...：2页块**buddy**伙伴

n 3、对空闲区域的管理按照页块大小分组进行管理。

u 系统设置一个静态数组**free\_area[ ]**来管理各个空闲页块组。

p 在/mm/page\_alloc.c中。

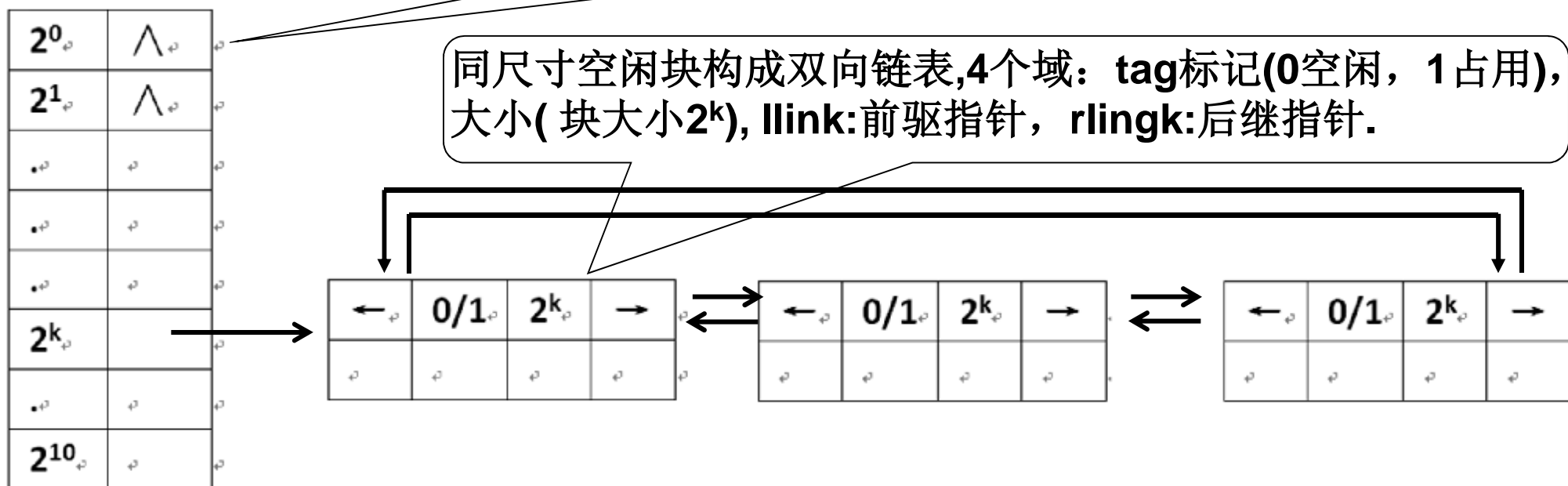
# I 伙伴算法特点:

n (1) 分配块的大小均是 $2^k$ ;

n (2) 分配和回收简单

空闲块按其大小链入各自的链表;  
该数组是各链表的表头接点

同尺寸空闲块构成双向链表,4个域: **tag**标记(0空闲, 1占用),  
大小(块大小 $2^k$ ), **llink**:前驱指针, **rlink**:后继指针.



# 伙伴算法的分配内存过程

- 1、当收到内存分配请求时，系统照buddy算法，根据请求的页面数在空闲块数组对应的空闲页面组中搜索。
- 2、若请求的页面不是2的整数次幂，则按照稍大于请求数的2的整数次幂的值搜索对应的页面块组。例如请求2个页面时，则搜索2页面块组，请求3个页面时，就搜索4页面块组。
- 3、当搜索对应的页块组，而没有可利用的空闲页块时，再搜索更大一些的页块组，例如当查找4页块组没有可利用的空闲页块时，就再搜索8页块组。
- 4、在找到可利用的空闲页块后，就按照请求的页面块数分配所需的页面。
- 5、当某一空闲页面块被分配后，若仍有剩余的空闲页面，则根据剩余页面的大小把它们加入相应的空闲页块组中。

# 伙伴算法的释放内存过程

- 丨 1、当释放内存页面时，系统将这些页面回收并作为空闲页面看待。
- 丨 2、然后检查是否存在与这些页面相邻的其它空闲页块，若存在，则把它们合并为一个连续的空闲区。
- 丨 3、再按照buddy算法划分不同的页面块，并加到相应的页面块组中。

# slab机制

## I 目标

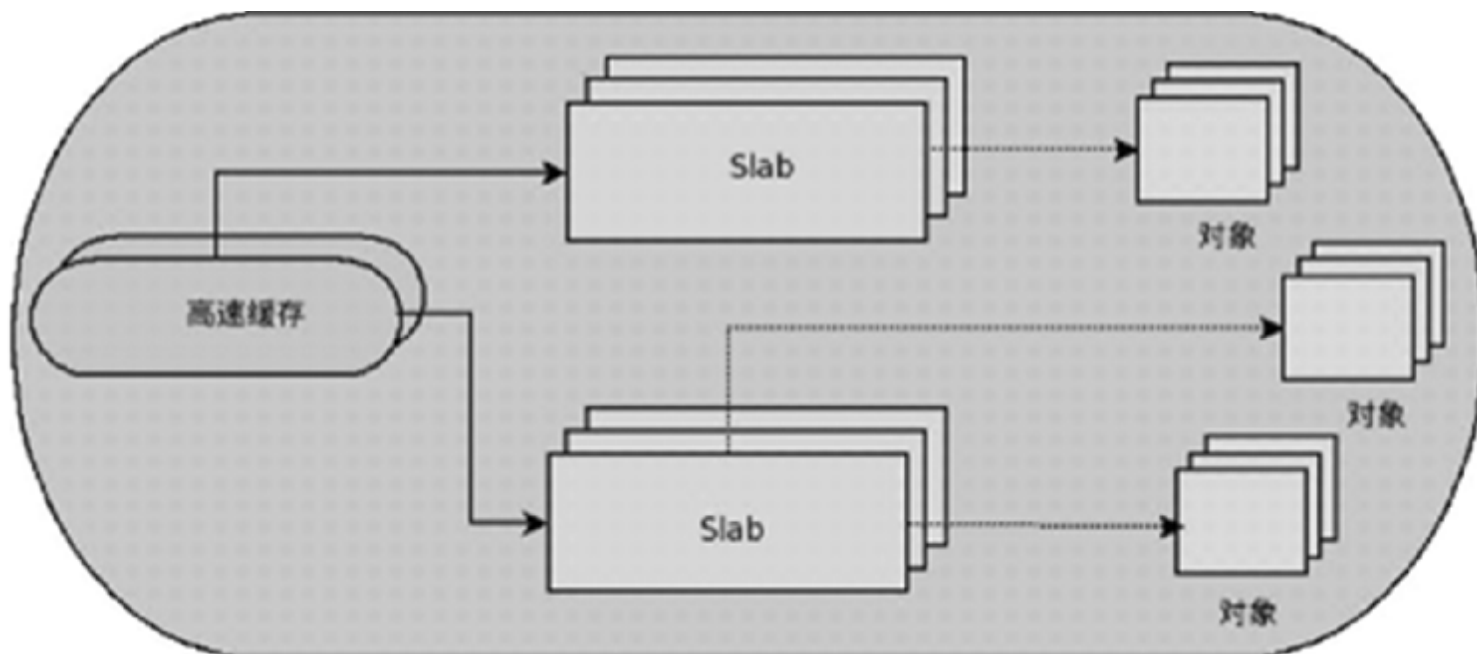
**nlinux**经常出现内存分配并释放的对象，如进程描述符等，这些对象的大小一般比较小，如果直接采用伙伴系统来进行分配和释放，不仅会造成大量的内存碎片，而且处理速度也太慢，从而引入**slab**机制。

## I **slab**原理

**ns**slab分配器是基于**对象**进行管理的，相同类型的对象归为一类(如进程描述符就是一类)，每当要申请这样一个对象，**slab**分配器就从一个**slab**列表中分配一个这样大小的单元出去，而当要释放时，将其重新保存在该列表中，而不是直接返回给伙伴系统，从而避免这些内碎片。

# slab分配器结构

- l **slab**分配器为每一种对象建立高速缓存。内核对该对象的分配和释放均是在这块高速缓存中操作。



- l 在**cache**和**object**中加入**slab**分配器，是在时间和空间上的折中方案。

# vmalloc

## I 目标

- n 前面已经分析了linux如何利用伙伴系统,slab分配器分配内存,用这些方法得到的内存在物理地址上都是连续的。
- n 然而，有些时候,每次请求内存时,系统都分配物理地址连续的内存块是不合适的，可以利用小块内存“连”成大块可使用的内存，这在操作系统设计中也被称为“内存拼接”



## I 特点

- n 在linux内核中用来管理内存拼接的接口是**vmalloc**.用**vmalloc**分配得到的内存在线性地址是平滑的,但是物理地址上是非连续的.
- n 显然,用**vmalloc**内存拼接在需要较大内存,而内存访问相比之下**不是很频繁**的情况下是比较有效的.

## I 实现

- n1.寻找一个新的连续线性地址空间;
- n2.依次分配一组非连续的页框;
- n3.为线性地址空间和非连续页框建立映射关系,即修改内核页表;