


《操作系统原理》

第7章 内存管理

教师：苏曙光

华中科技大学网安学院

2023年10月-2024年01月



内存管理

● 主要内容

- 内存管理的功能
- 物理内存管理
 - ◆ 分区存储管理
 - ◆ 覆盖技术
 - ◆ 对换技术
- 虚拟内存管理
 - ◆ 页式存储管理
 - ◆ 段式存储管理
- LINUX存储管理
- 保护模式
 - ◆ Intel CPU保护模式
 - ◆ 段和页机制

● 重点

- 地址映射
- 虚拟内存
- 页式管理
- 段式管理
- 保护模式内存管理机制



7.1 存储管理的功能

存储管理的功能

- 存储器功能需求

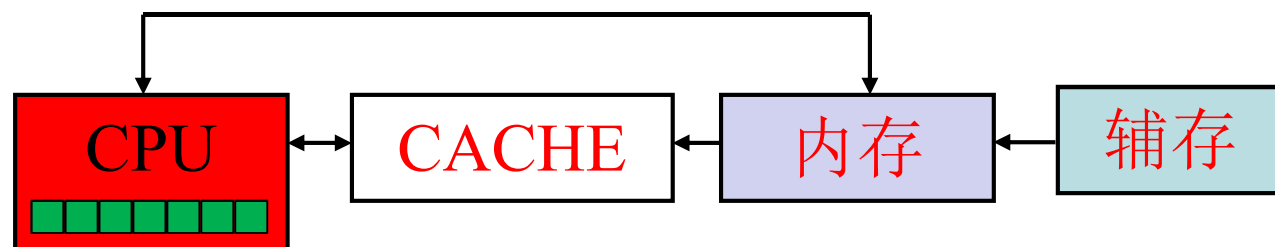
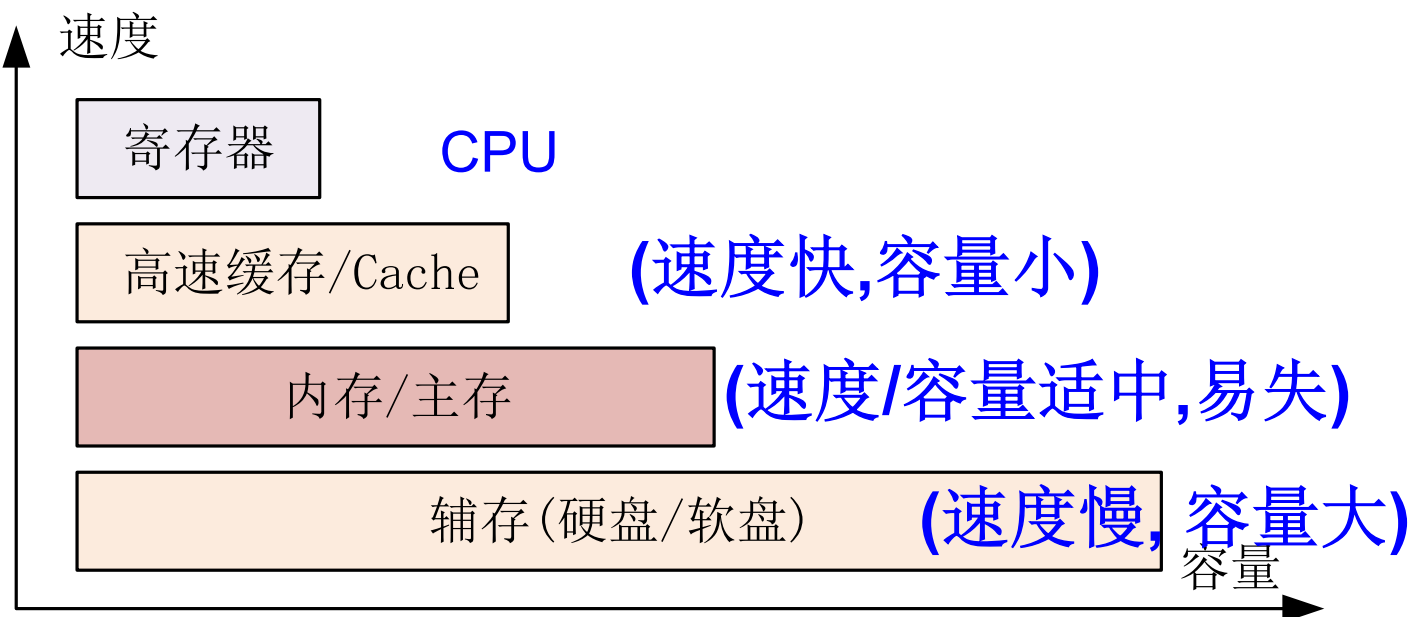
- 容量足够大
- 速度足够快
- 信息永久保存

- 三级存储体系

- 内存
- 辅存
- cache
- 基本思想

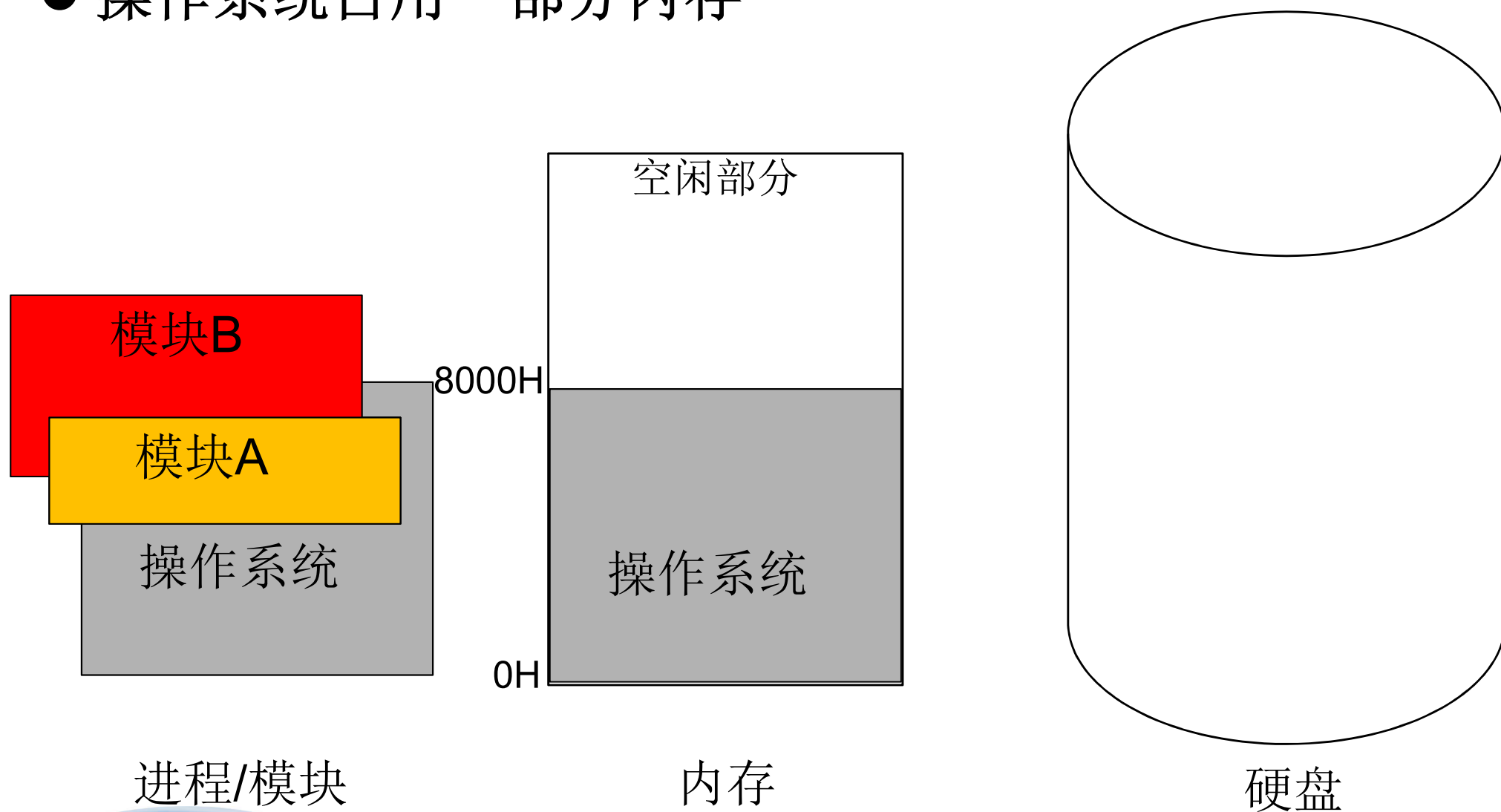
- ◆ 用辅存支援内存，提高容量

- ◆ 用cache支援内存，提升效率



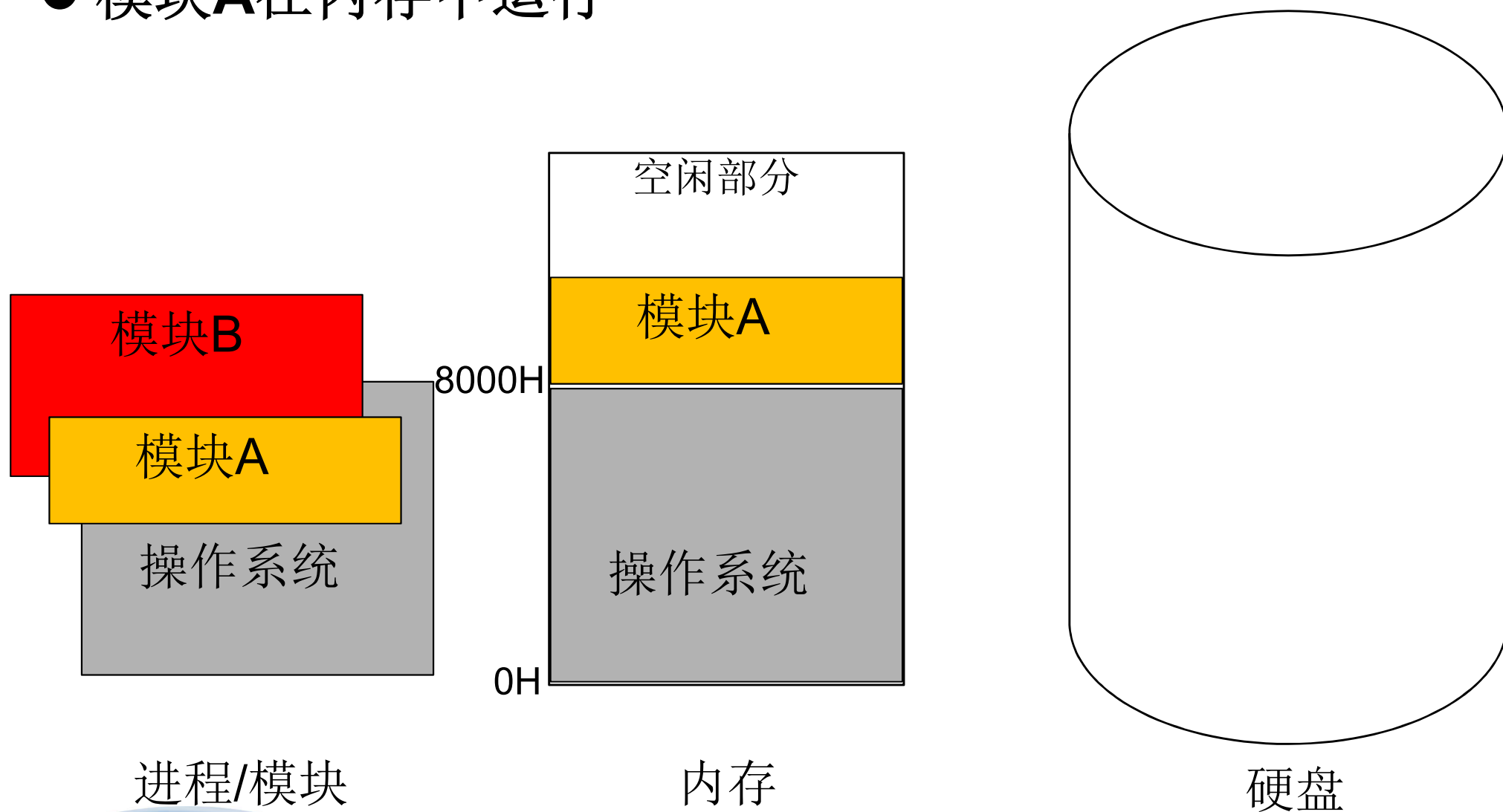
换出与换入的讨论（辅存支援内存）

- 操作系统占用一部分内存



换出与换入的讨论（辅存支援内存）

- 模块A在内存中运行



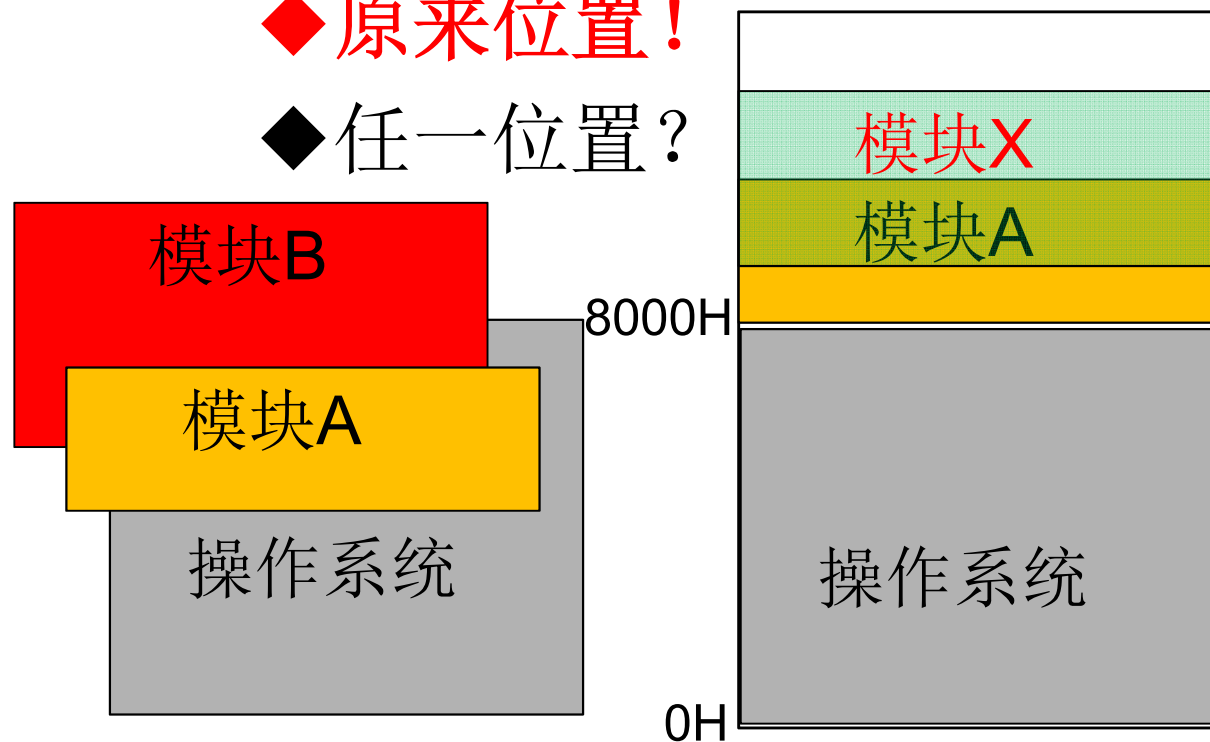
换出与换入的讨论（辅存支援内存）

● 模块A换入到内存

■ 放到什么位置

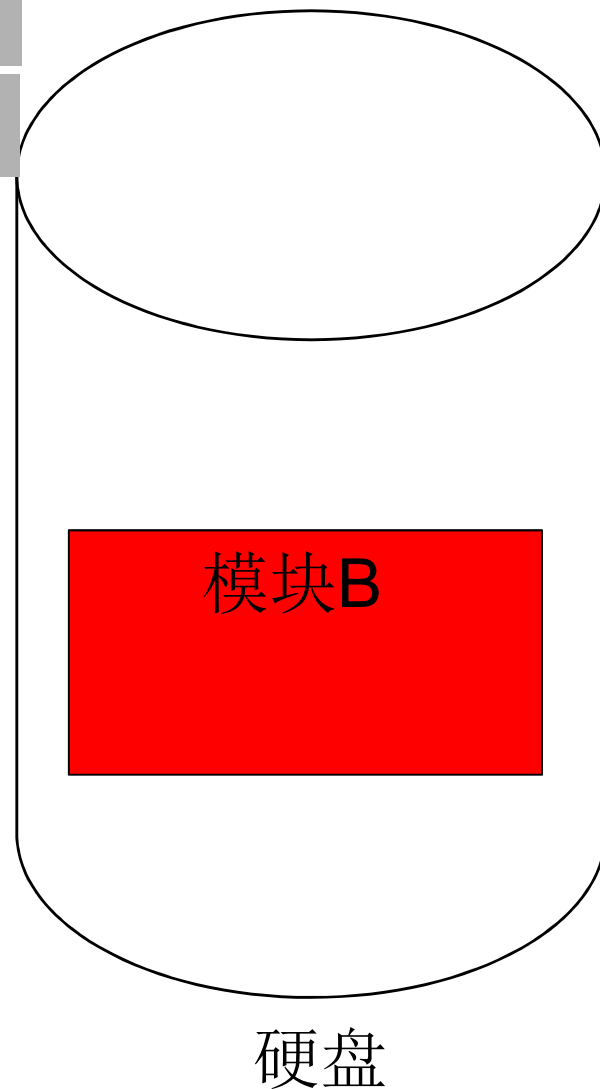
◆ 原来位置！

◆ 任一位置？



优点：程序简单！

缺点：地址冲突？



进程/模块

内存

硬盘

换出与换入的讨论（辅存支援内存）

● 模块A换入到内存

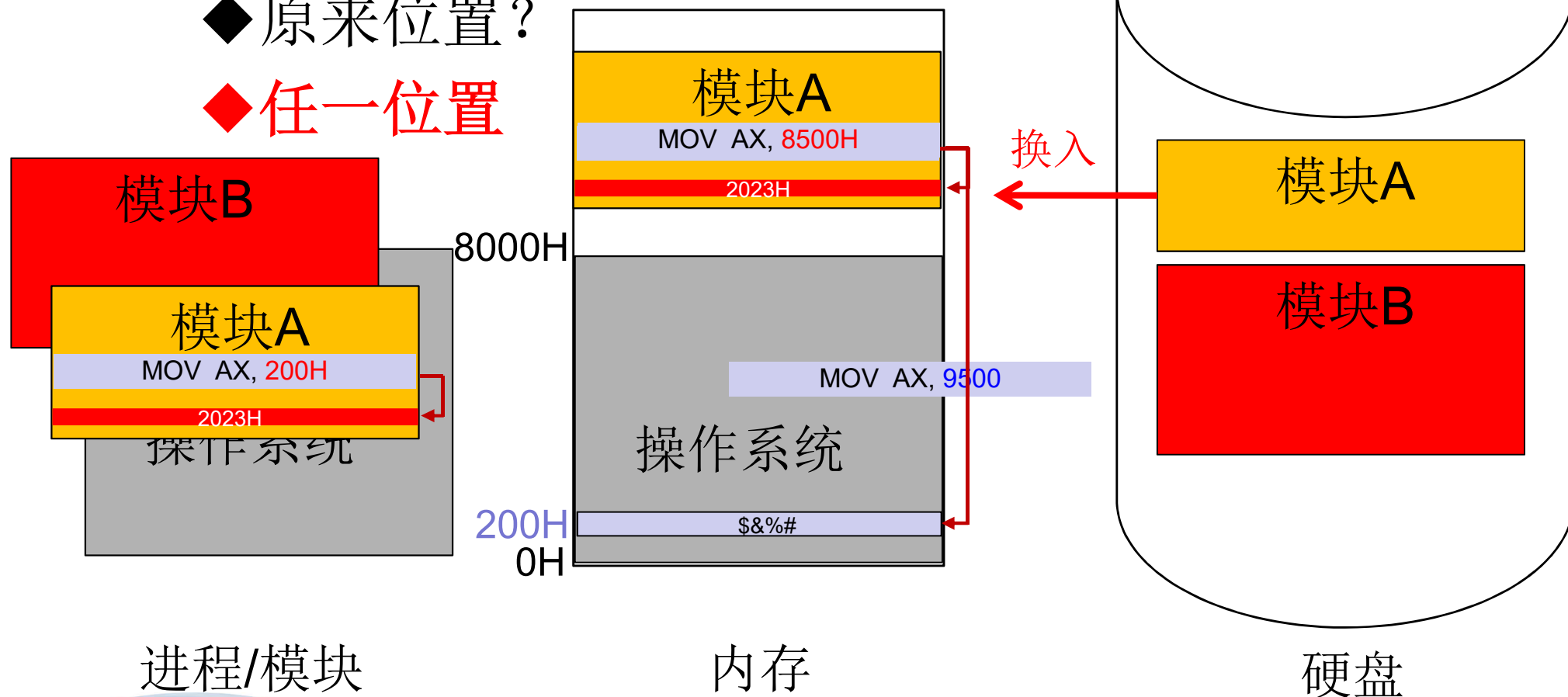
■ 放到什么位置

◆ 原来位置？

◆ 任一位置

优点：利用内存灵活

问题：地址要重定位



换出与换入的讨论（辅存支援内存）

- 地址重定位（地址重映射）

- 重新确定指令中目标数据的正确地址（更新地址）。
- 新的值与目标程序块的实际放置位置有关

MOV AX, 200H



MOV AX, 8500H

存储管理的功能

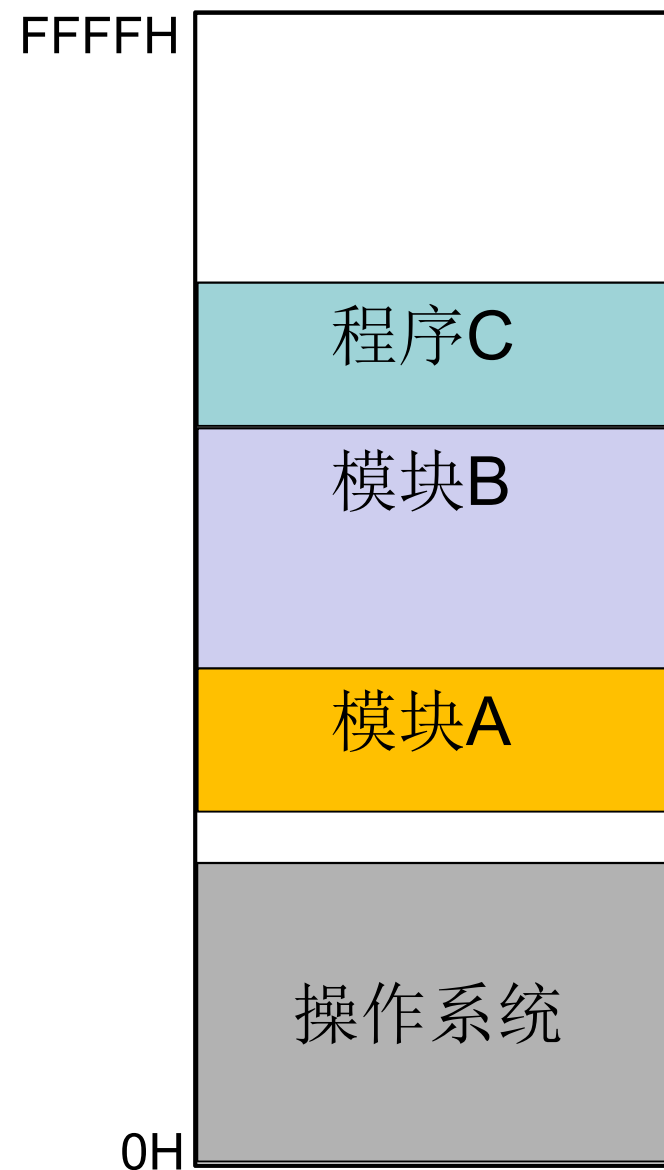
● 存储器功能需求

■ 容量足够大

■ 速度足够快

■ 信息永久保存

■ 多道程序并行



存储管理的功能

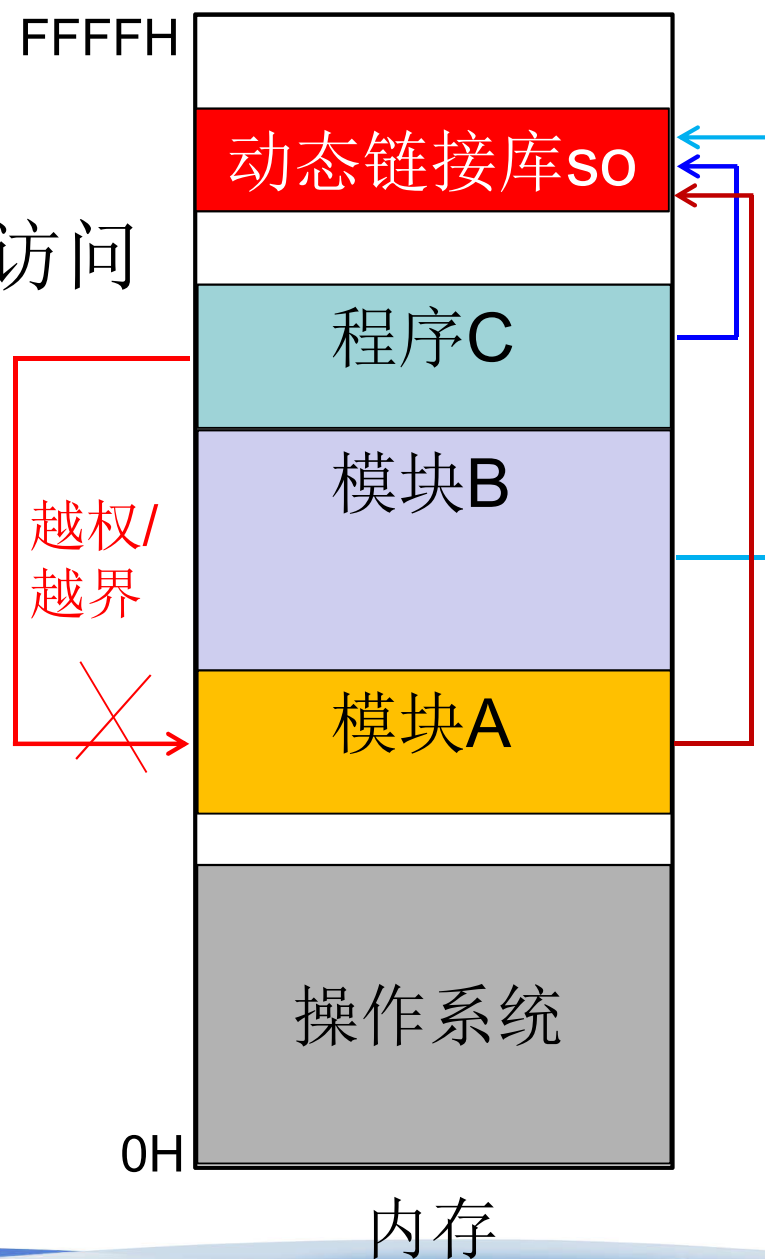
- 多道程序并行带来的问题

- 保护

- ◆ 禁止程序间的越权访问 | 越界访问

- 共享

- ◆ 代码和数据共享，节省内存



存储管理的功能

- 存储管理的功能

- 1) 地址映射
- 2) 存储扩充/虚拟存储
- 3) 内存分配
- 4) 存储保护

存储管理的功能：1) 地址映射

● 定义

■ 地址重定位，地址重映射

■ 把程序中的地址（虚拟地址,虚地址,逻辑地址,相对地址）转换成真实的内存地址（实地址,物理地址,绝对地址）的过程。

◆ 虚拟地址/源程序地址：地址，变量，标号，函数名

```
31 char m_sLabelFile[1024];  
32 string m_sImageFileName; //当前正处理的图像文件全名  
33  
34 int m_nImageCount;  
35 int m_CurrentFrameNo;  
36  
37  
38 void readPath(CString src_path, std::vector<CString>  
39 void ShowMatImageInControl(cv::Mat& matImage, UINT n  
40 void DrawRectangle(HWND m_HWnd, int x1, int y1, int
```

MOV AX, 200H



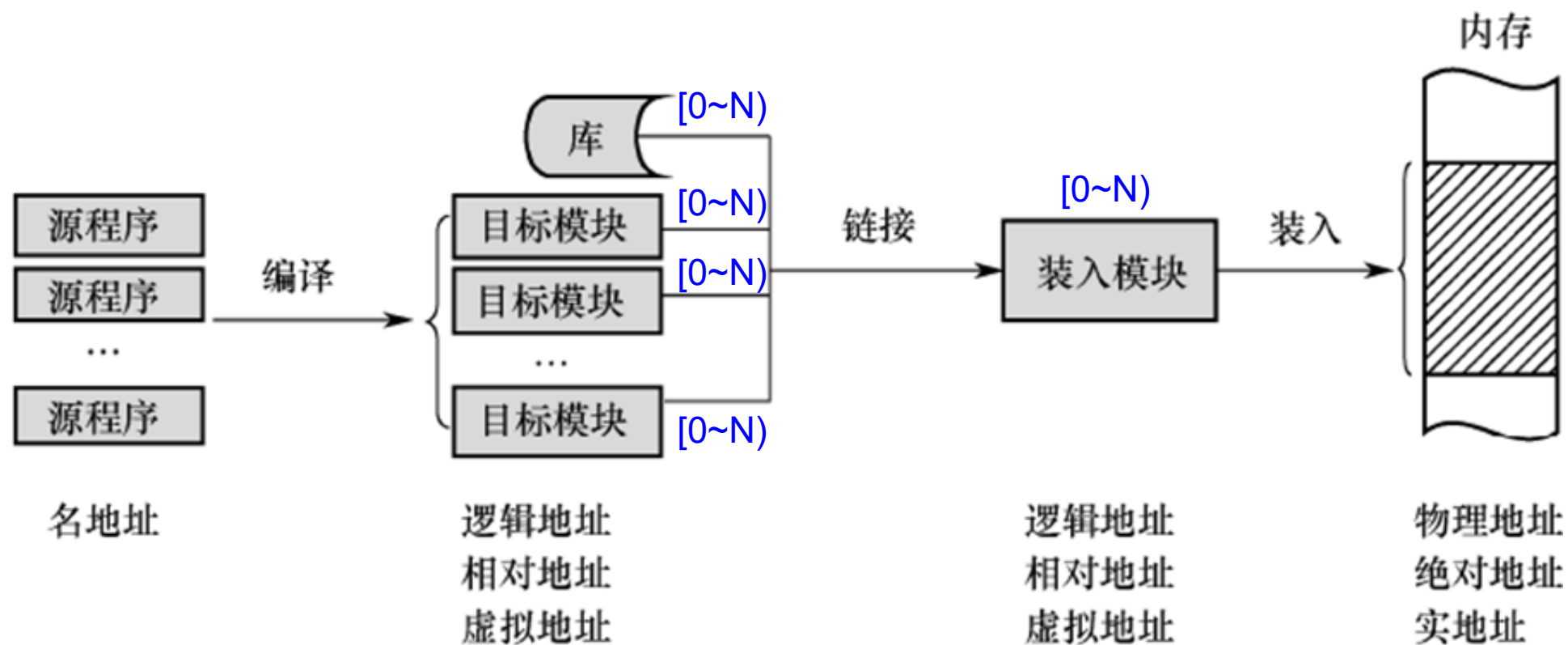
MOV AX, 8500H

物理地址/逻辑地址

● 逻辑地址

■ 多道程序环境

■ 目标模块/装入模块：使用内部的线性地址： $[0 \sim N)$



物理地址/逻辑地址

- 物理地址

- 内存单元的绝对地址

- ◆ 数据实际存放位置

- 单道程序环境中

- ◆ 程序装入的起始地址可以预知。

- ◆ 程序中直接指明数据/指令的物理地址。

- 多道程序环境中

- ◆ 程序不能预知装入的地址

- ◆ 程序中无法明确地使用物理地址。

存储管理的功能：1) 地址映射

● 定义

■ 地址重定位，地址重映射

■ 把程序中的地址（虚拟地址,虚地址,逻辑地址,相对地址）转换成真实的内存地址（实地址,物理地址,绝对地址）的过程。

◆ 虚拟地址/源程序地址：地址，变量，标号，函数名

```
31 char m_sLabelFile[1024];
32 string m_sImageFileName; //当前正处理的图像文件全名
33
34 int m_nIamgeCount;
35 int m_CurrentFrameNo;
36
37
38 void readPath(CString
39 void ShowMatImageInCo
40 void DrawRectangle(HW
```

```
31 char m_sLabelFile[1024];
32 string m_sImageFileName; //当前正处理的图像文件全名
33
34 int m_nIamgeCount;
35 int m_CurrentFrameNo;
36
37
38 void readPath(CString src_path, std::vector<CString>
39 void ShowMatImageInControl(cv::Mat& matImage, UINT n
40 void DrawRectangle(HWND m_HWnd, int x1, int y1, int
```

MOV AX, 200H



MOV AX, 8500H

固定地址映射

- 定义

- 编程或编译时确定逻辑地址和物理地址映射关系。

- 特点

- 程序加载时必须加载到指定的内存区域。

- ◆ 容易产生地址冲突，运行失败。

- 不能适应多道程序环境

```
31 char m_sLabelFile[1024];
32 string m_sImageFileName; //当前正处理的图像文件全名
33
34 int m_nImageCount;
35 int m_CurrentFrameNo;
36
37
38 void readPath(CString src_path, std::vector<CString>
39 void ShowMatImageInControl(cv::Mat& matImage, UINT n
40 void DrawRectangle(HWND m_HWnd, int x1, int y1, int
```

MOV AX, 200H



MOV AX, 8500H

静态地址映射

● 定义

- 程序**装入时**由操作系统完成逻辑地址到物理地址的映射。
- 保证程序在**运行之前**所有地址都绑定到主存
- 映射方式

◆ 物理地址 $MA = \text{装入基址}BA + \text{虚拟地址}VA$

```
31 char m_sLabelFile[1024];
32 string m_sImageFileName; //当前正处理的图像文件全名
33
34 int m_nImageCount;
35 int m_CurrentFrameNo;
36
37
38 void readPath(CString src_path, std::vector<CString>
39 void ShowMatImageInControl(cv::Mat& matImage, UINT n
40 void DrawRectangle(HWND m_HWnd, int x1, int y1, int
```

MOV AX, 200H

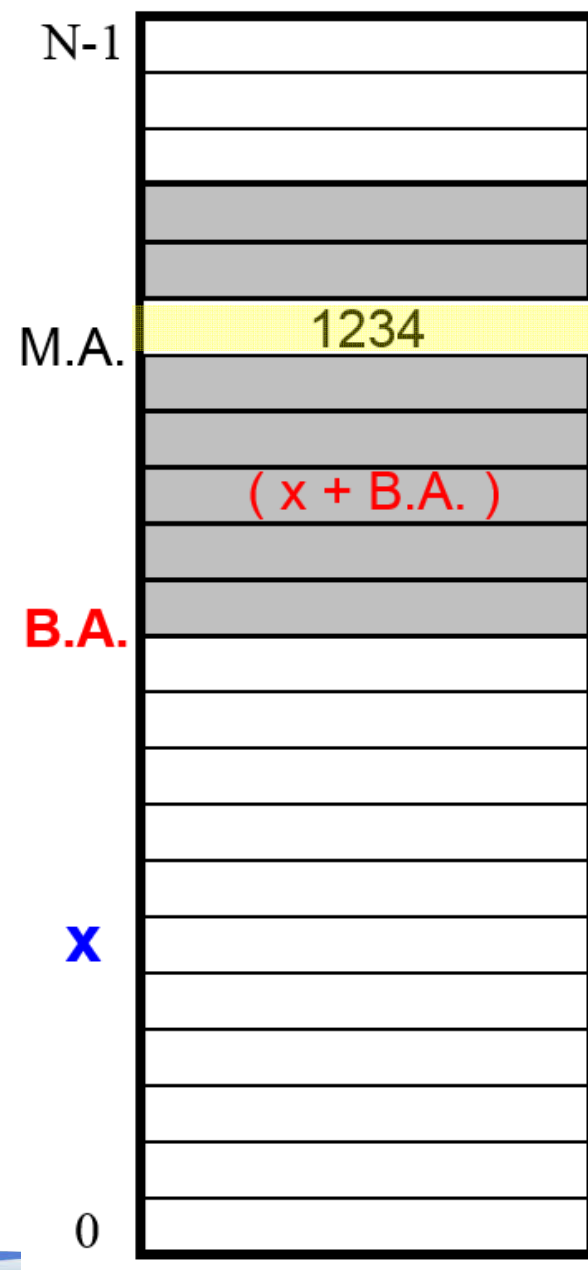


MOV AX, 8500H

静态地址映射

● 特点

- 程序运行之前确定映射关系
- 程序占用连续的内存空间
- 程序装入后不能移动
 - ◆ 如果移动必须放回原来位置



动态地址映射

● 定义

■ 在程序**执行过程中**把逻辑地址转换为物理地址。

■ 映射方式

◆ 物理地址 $MA = \text{装入基址} BA + \text{虚拟地址} VA$

◆ 装入基址：基址寄存器**BAR**

```
31 char m_sLabelFile[1024];
32 string m_sImageFileName; //当前正处理的图像文件全名
33
34 int m_nImageCount;
35 int m_CurrentFrameNo;
36
37
38 void readPath(CString src_path, std::vector<CString>
39 void ShowMatImageInControl(cv::Mat& matImage, UINT n
40 void DrawRectangle(HWND m_HWnd, int x1, int y1, int
```

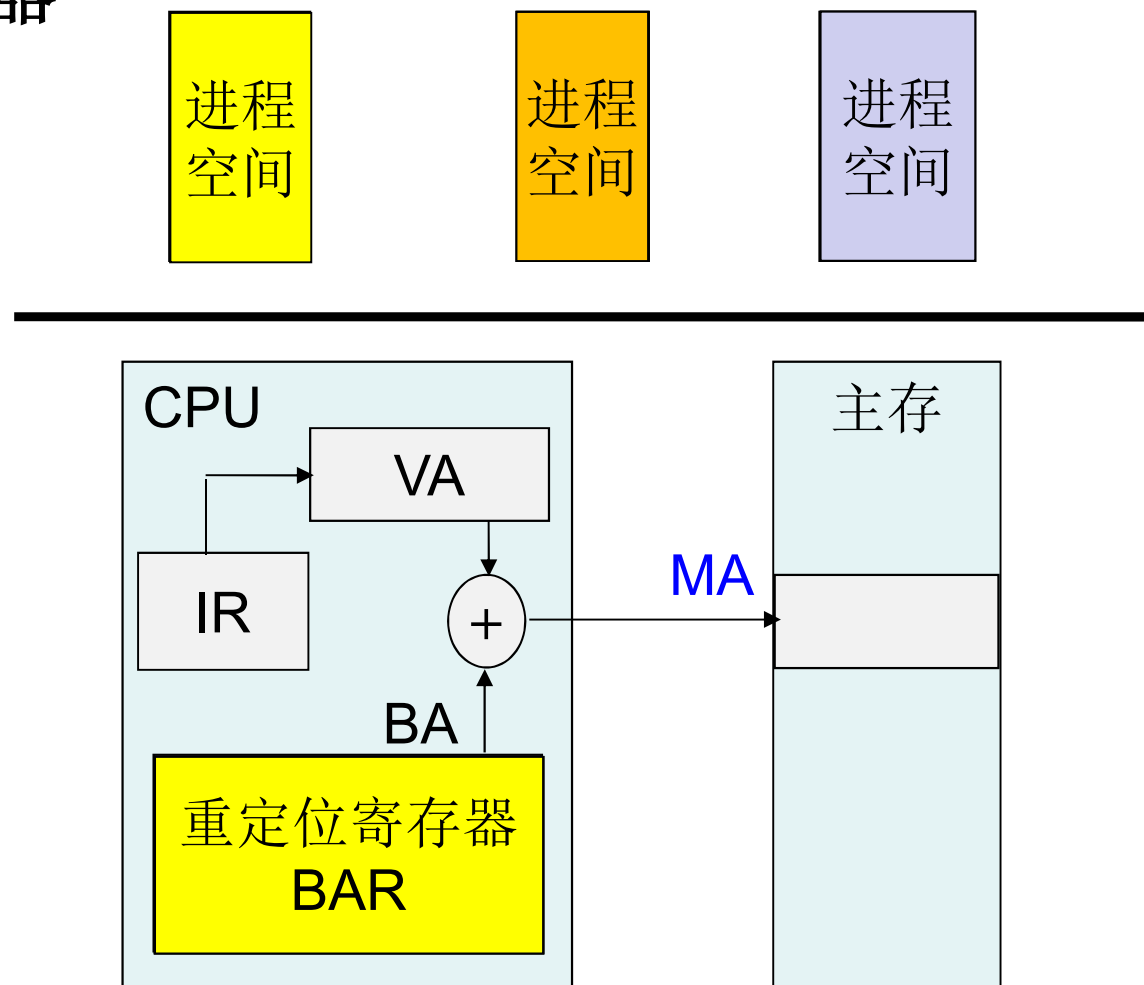
MOV AX, 200H



MOV AX, 8500H

实现动态地址映射的思路

- 切换进程的同时切换基址寄存器**BAR**
- **IR = 指令寄存器**
 - 提供VA



实现动态地址映射的思路

- 程序可分配到不连续的多块内存中存放

- 例：程序 = 段A + 段B + 段C

- 按段编译

- ◆ 段内地址：线性地址[0~N)

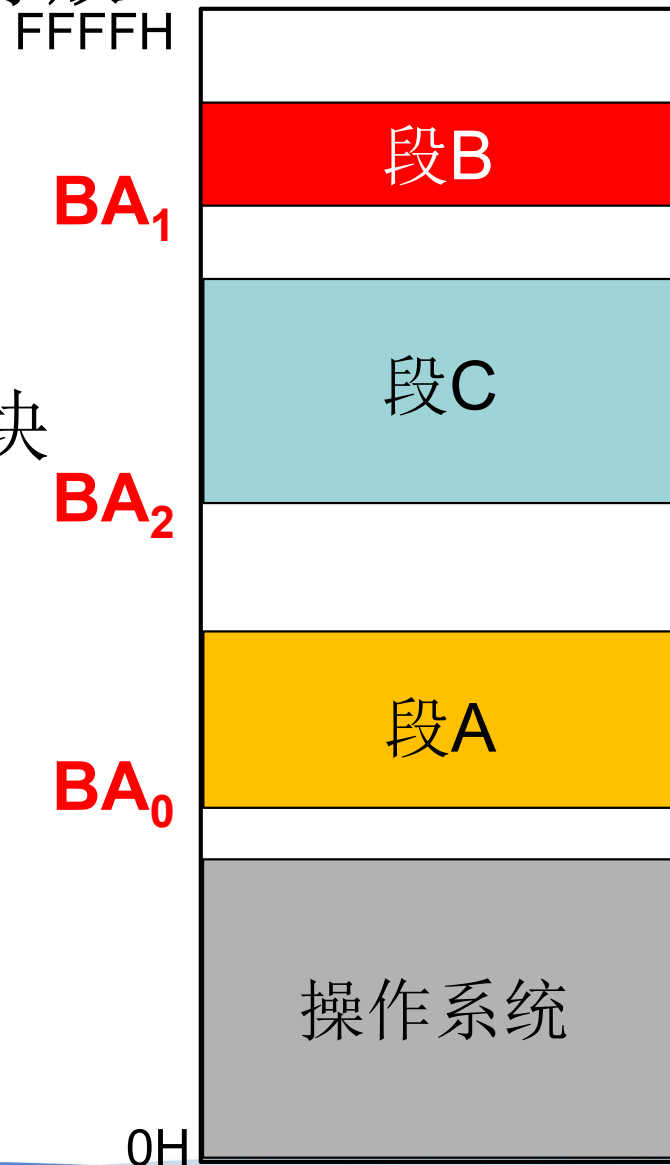
- 按段装入，不同段放入不同内存块

- ◆ 每段维护一个段寄存器

- 段的重定位寄存器

- 段式存储管理

- 段的切换



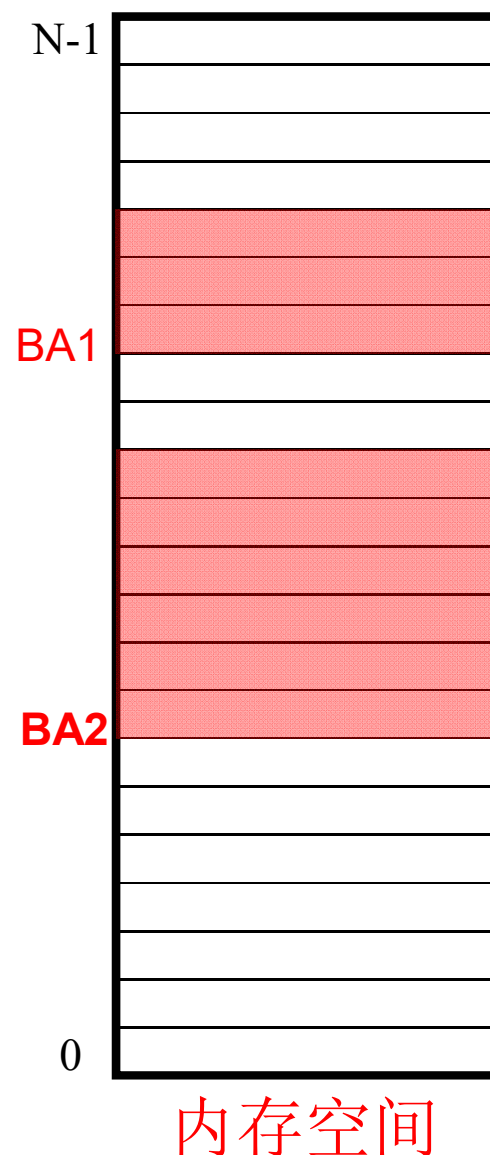
动态地址映射

● 特点

- 程序占用的内存空间可动态变化
 - ◆ 若程序移动及时更新基址BA
- 程序不要求占用连续的内存空间
 - ◆ 需要记录每段放置的基址BA
- 便于多个进程共享代码
 - ◆ 共享代码作为独立的一段存放

● 缺点

- 硬件支持（MMU：内存管理单元）
- 软件复杂



存储管理的功能

- 存储管理的功能

- 1) 地址映射

- 2) 存储扩充/虚拟存储

- 3) 内存分配

- 4) 存储保护

存储管理的功能：2) 存储扩充（虚拟存储）

● 解决的问题

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

● 问题1的解决方法（覆盖|换出换入|虚拟存储的原理）

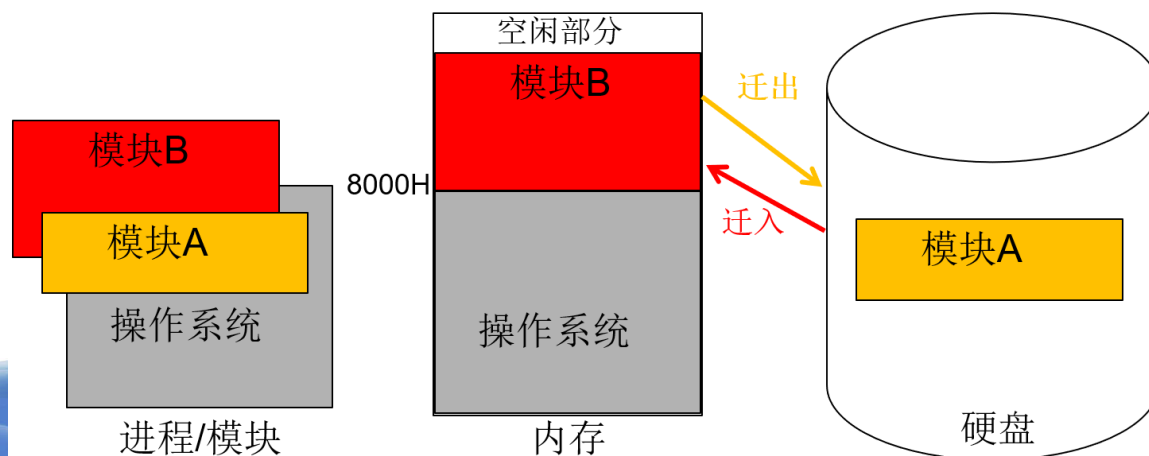
- (回顾)借助辅存在逻辑上扩充内存，解决内存不足

◆ 迁入：装入将要运行的部分到内存

□ 从辅存临时调入内存

◆ 迁出：把不运行部分暂存到辅存上

□ 尽量腾空内存



| | | | |
|----|--------------|---------|-----------|
| 1 | LF | EQU | 0Ah |
| 2 | CR | EQU | 0Dh |
| 3 | ESC | EQU | 01Bh |
| 4 | COM1 | EQU | 03F8h |
| 5 | COM2 | EQU | 02F8h |
| 6 | COMPORT | EQU | COM1 |
| 7 | THRE | EQU | 020h |
| 8 | TEMT | EQU | 040h |
| 9 | DR | EQU | 001h |
| 10 | THR | EQU | COMPORT |
| 11 | RBR | EQU | COMPORT |
| 12 | LSR | EQU | COMPORT+3 |
| 13 | MCR | EQU | COMPORT+4 |
| 14 | DLL | EQU | COMPORT+0 |
| 15 | DLM | EQU | COMPORT+1 |
| 16 | LCR | EQU | COMPORT+3 |
| 17 | FCR | EQU | COMPORT+2 |
| 18 | IIR | EQU | COMPORT+1 |
| 19 | IER | EQU | COMPORT+1 |
| 20 | EOF_REC | EQU | 01 |
| 21 | DATA_REC | EQU | 00 |
| 22 | EAD_REC | EQU | 02 |
| 23 | SSA_REC | EQU | 03 |
| 24 | PIO | EQU | 0398h |
| 25 | RTC_BASE | EQU | 0070h |
| 26 | RTC_DATA | EQU | 0071h |
| 27 | BASE_SEGMENT | EQU | 0380h |
| 28 | WRSPEC | MACRO | |
| 29 | MOV | AL, ' ' | |
| 30 | CALL | TXCHAR | |
| 31 | #EM | | |
| 32 | WREQUAL | MACRO | |
| 33 | MOV | AL, '=' | |
| 34 | CALL | TXCHAR | |
| 35 | #EM | | |
| 36 | | ORG | 0400h |
| 37 | INITMON: | CLI | |
| 38 | | MOV | AX, CS |
| 39 | | MOV | DS, AX |
| 40 | | MOV | SS, AX |

存储管理的功能： 2) 存储扩充（虚拟存储）

● 实现存储扩充（虚拟存储）的前提

- 有适当容量的内存
- 有足够大的辅存
- 有地址变换机构

● 虚拟存储的应用

- 页式虚拟存储
- 段式虚拟存储

| | | | |
|----|--------------|---------|-----------|
| 1 | LF | EQU | 0Ah |
| 2 | CR | EQU | 0Dh |
| 3 | ESC | EQU | 01Bh |
| 4 | COM1 | EQU | 03F8h |
| 5 | COM2 | EQU | 02F8h |
| 6 | COMPORT | EQU | COM1 |
| 7 | THRE | EQU | 020h |
| 8 | TEMT | EQU | 040h |
| 9 | DR | EQU | 001h |
| 10 | THR | EQU | COMPORT |
| 11 | RBR | EQU | COMPORT |
| 12 | LSR | EQU | COMPORT+3 |
| 13 | MCR | EQU | COMPORT+4 |
| 14 | DLL | EQU | COMPORT+0 |
| 15 | DLM | EQU | COMPORT+1 |
| 16 | LCR | EQU | COMPORT+3 |
| 17 | FCR | EQU | COMPORT+2 |
| 18 | IIR | EQU | COMPORT+1 |
| 19 | IER | EQU | COMPORT+1 |
| 20 | EOF_REC | EQU | 01 |
| 21 | DATA_REC | EQU | 00 |
| 22 | EAD_REC | EQU | 02 |
| 23 | SSA_REC | EQU | 03 |
| 24 | PIO | EQU | 0398h |
| 25 | RTC_BASE | EQU | 0070h |
| 26 | RTC_DATA | EQU | 0071h |
| 27 | BASE_SEGMENT | EQU | 0380h |
| 28 | WRSPEC | MACRO | |
| 29 | MOV | AL, ' ' | |
| 30 | CALL | TXCHAR | |
| 31 | #EM | | |
| 32 | WREQUAL | MACRO | |
| 33 | MOV | AL, '=' | |
| 34 | CALL | TXCHAR | |
| 35 | #EM | | |
| 36 | | ORG | 0400h |
| 37 | INITMON: | CLI | |
| 38 | | MOV | AX, CS |
| 39 | | MOV | DS, AX |
| 40 | | MOV | SS, AX |

存储管理的功能： 3) 内存分配功能

- 为程序运行分配足够的内存空间

 - 代码 | 数据 | 堆栈

- 需要解决的问题

 - 放置策略

 - ◆ 程序调入到内存哪个/哪些区域

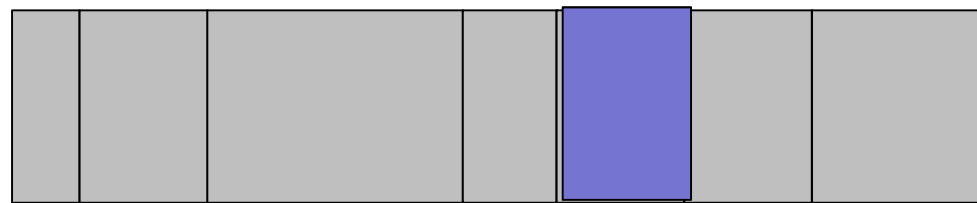
 - 调入策略

 - ◆ 何时把要运行的程序调入内存？

 - ◆ 预调策略 | 请调策略

 - 淘汰策略

 - ◆ 迁出（/淘汰）哪些程序以腾出内存空间。



内存

存储管理的功能：4) 存储保护功能

● 存储保护

■ 保证内存中的多道程序只能在给定区域活动，并且互不干扰。

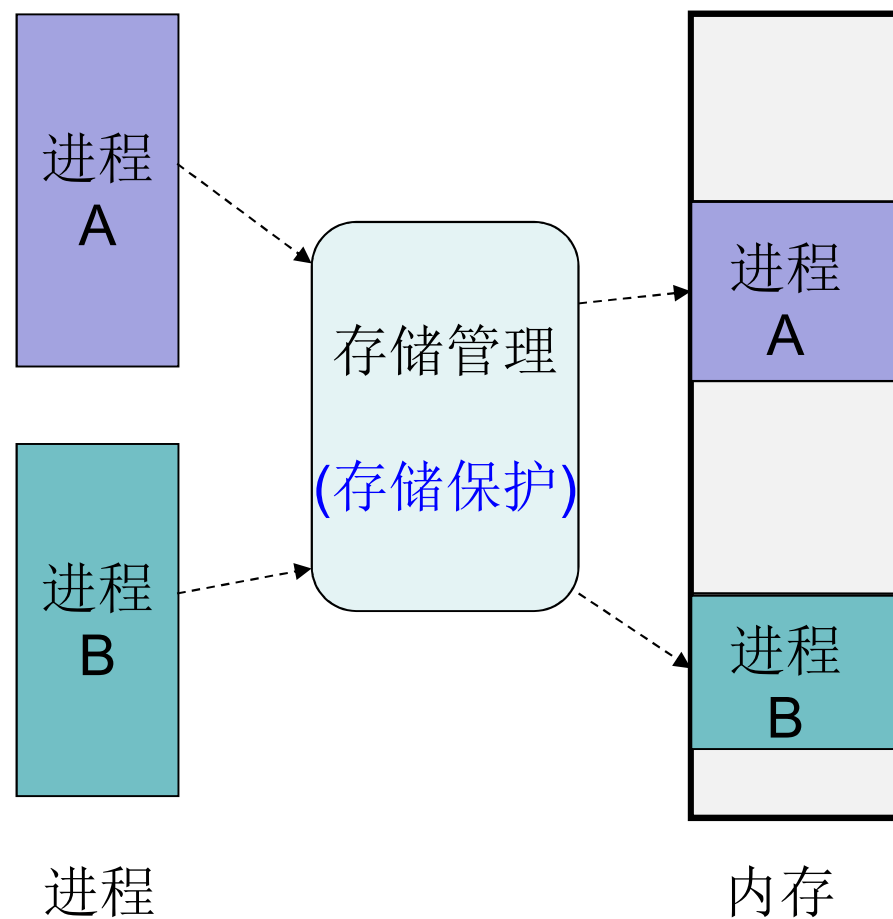
◆ 防止访问越界

◆ 防止访问越权

■ 方法

◆ 界址寄存器

◆ 存储键保护



存储管理的功能：4) 存储保护功能

● 保护方法

■ 界址寄存器

◆ 上限地址寄存器/下限地址寄存器

□ 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界限比较，判断是否越界？

◆ 基址寄存器和限长寄存器

◆ 适于连续物理分区中的情形

■ 存储键保护

◆ 适于不连续物理分块的情形，也可用于共享中的权限。

存储管理的功能：4) 存储保护功能

● 存储保护

■ 保证内存中的多道程序只能在给定区域活动，并且互不干扰。

◆ 防止访问越界

◆ 防止访问越权

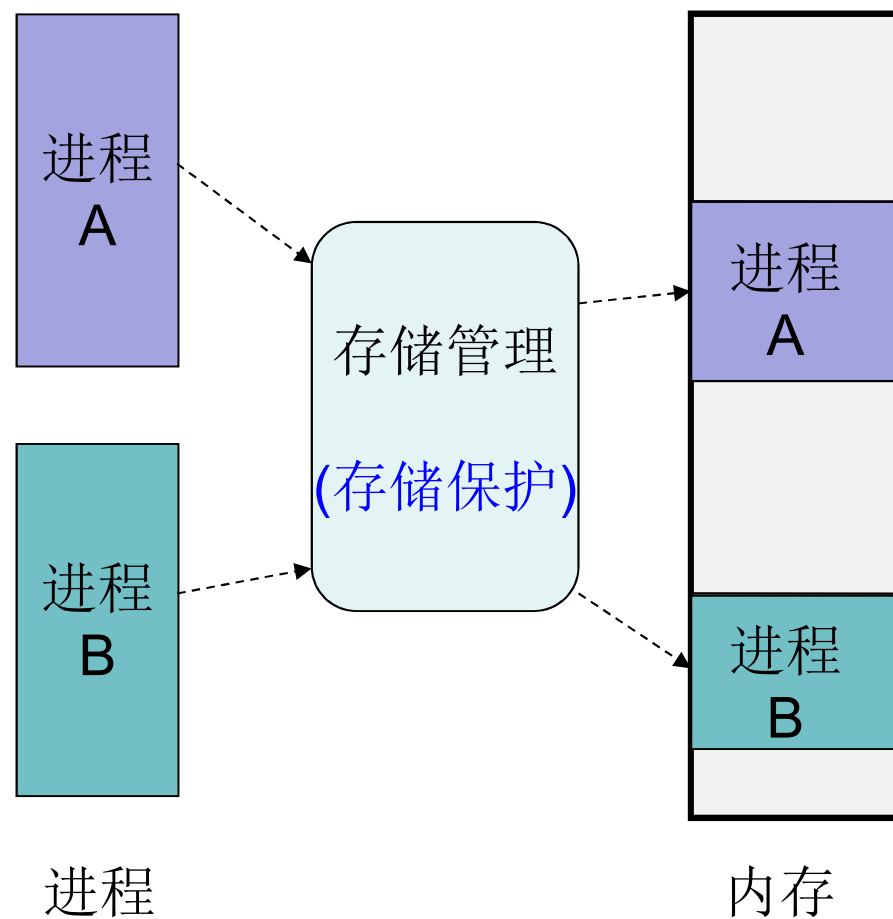
■ 方法

◆ 界址寄存器

◆ 存储键保护

◆ 段式虚拟内存管理

◆ 页式虚拟内存管理





2. 物理内存管理

物理内存管理

- 物理内存管理方法

- 单一区存储管理（不分区存储管理）

- 分区存储管理

- 内存覆盖技术

- 内存交换技术



内存

单一区存储管理（不分区存储管理）

- 定义

- 用户区**不分区**，完全被一个程序占用。

- 优点

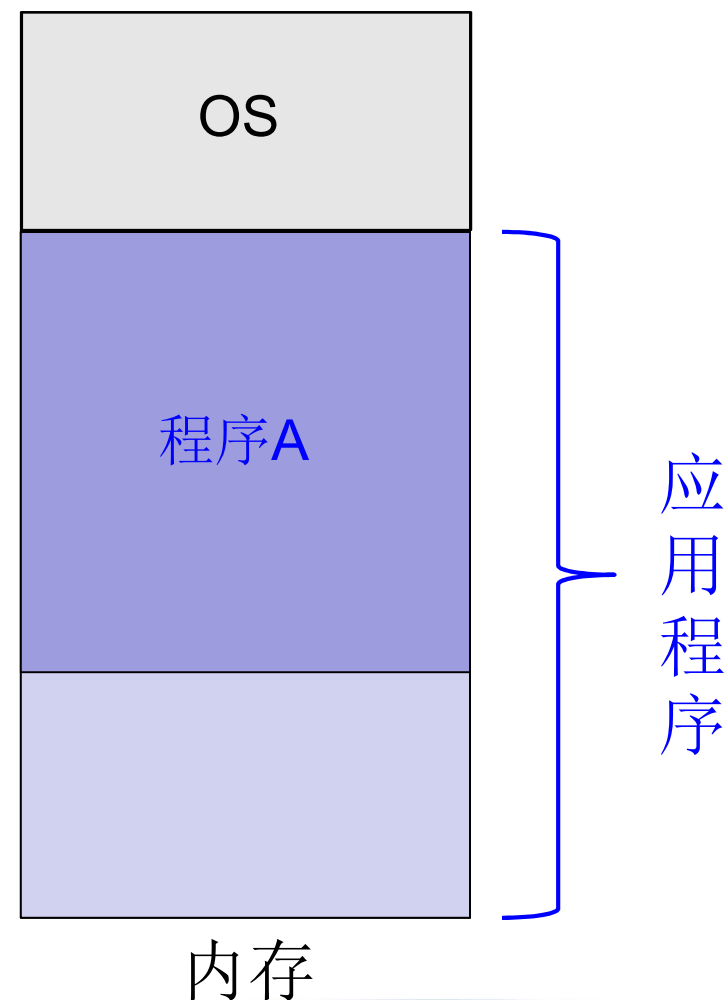
- 简单不需复杂硬件支持

- 缺点

- 程序占用整个内存
 - ◆ **内存浪费**，利用率低

- 应用场景

- 适于单用户单任务OS
 - ◆ 例：DOS、嵌入式系统



分区存储管理

- 定义

- 把用户区分为若干大小不等的分区，供不同程序使用。

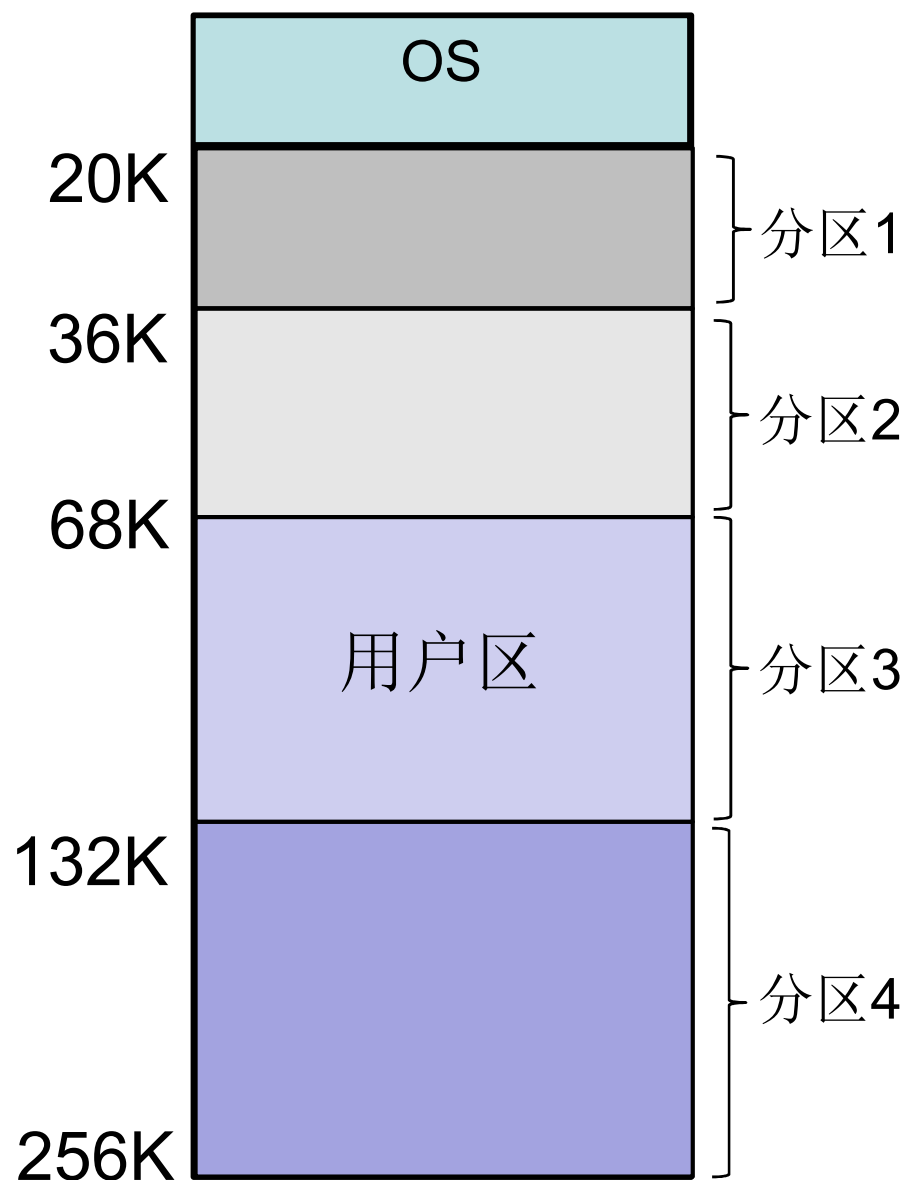
- 分类

- 固定分区

- ◆ 系统初始化时分区

- 动态分区

- ◆ 程序装入时临时分区



固定分区

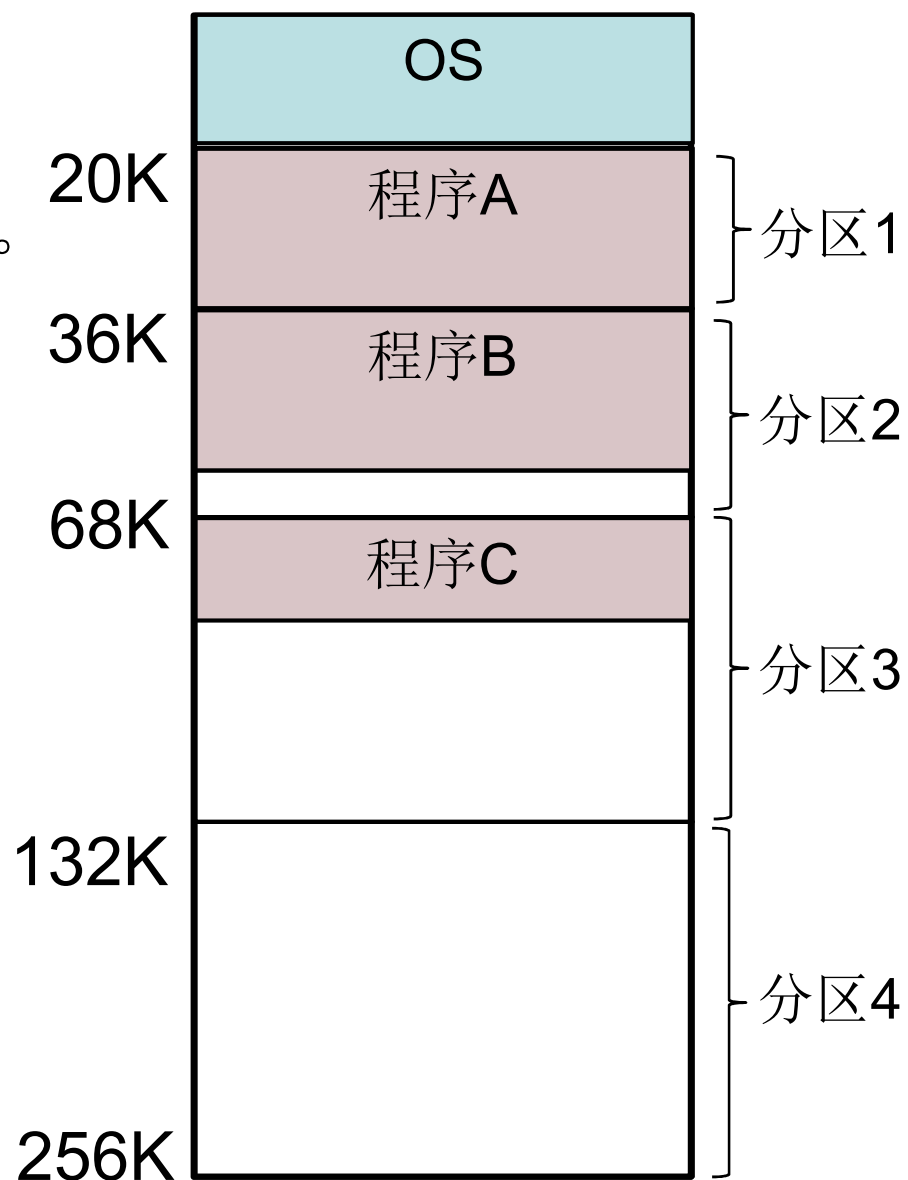
- 固定分区的特点

- 运行时分区的大小和位置不变
- 分区大小不同，适应不同程序需求。

- 分区表

- 记录分区的位置、大小和占用标志

| 区号 | 大小 | 起址 | 占用标志 |
|----|------|------|------|
| 1 | 16K | 20K | 1 |
| 2 | 32K | 36K | 1 |
| 3 | 64K | 68K | 1 |
| 4 | 124K | 132K | 0 |



固定分区

- 固定分区的缺点

- 浪费内存

- ◆ 当程序比所占用的分区小时浪费内存

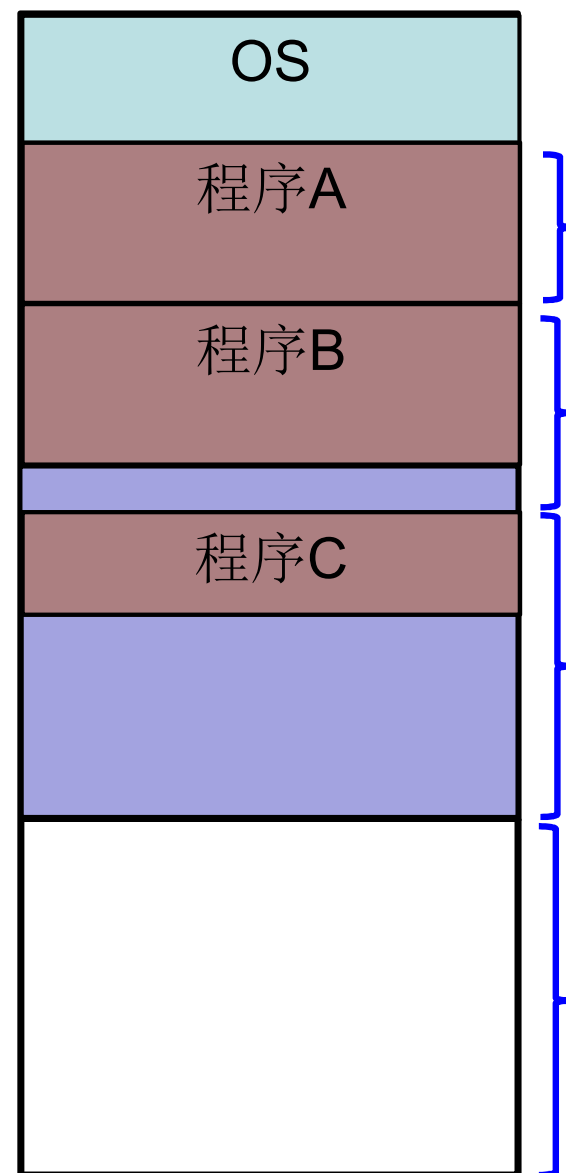
- 大程序可能无法运行

- ◆ 程序比最大分区大时

- 程序过多无法运行

- 应用建议

- 程序的装入数量和顺序要与分区的数量、大小顺序尽量保持一致。



固定分区

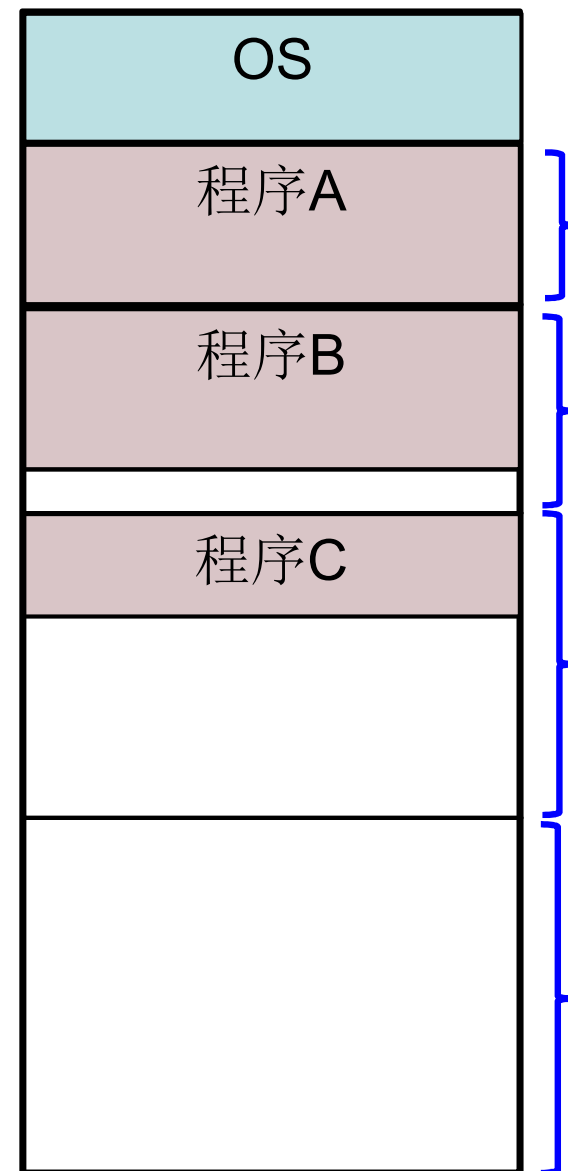
- 例子

- IBM OS/360

- ◆ 多道程序系统

- 15个分区

- ≤ 15 道程序



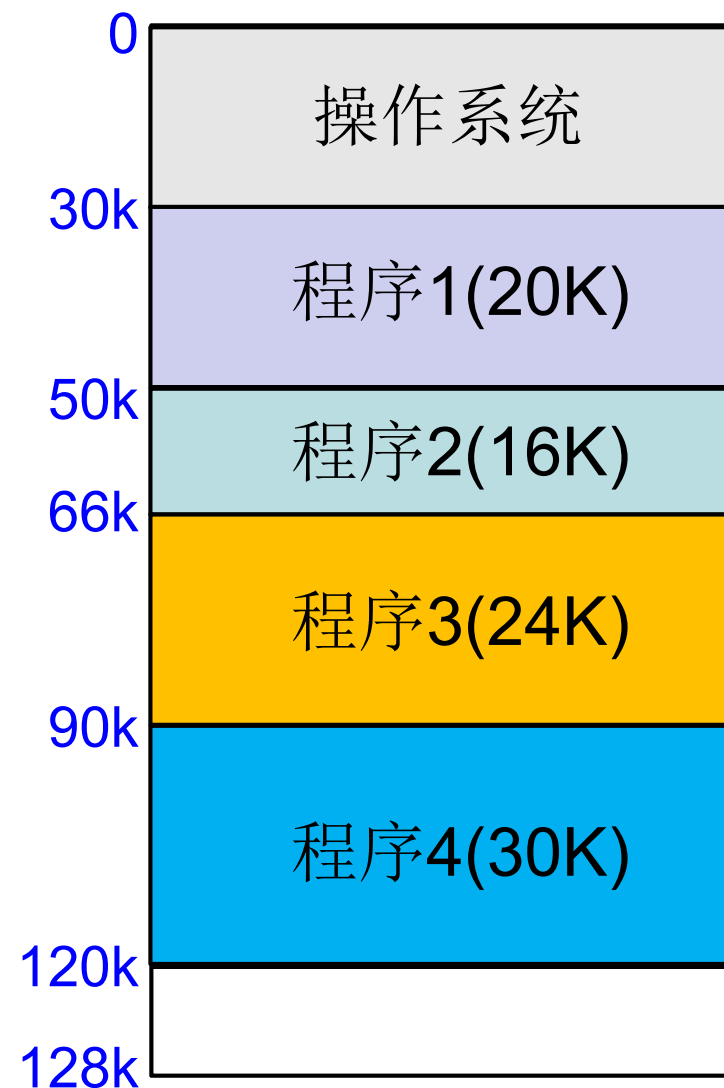
动态分区

- 定义

- 在程序装入时动态创建分区，分区与程序大小相等。

- 例：程序序列(先/后)

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



动态分区

● 分区回收（例）

■ 收回占用分区，以便重新分配

■ 例：

◆ 程序1（20K）：结束

◆ 程序3（24k）：结束

■ 回收时要考虑释放区和相邻区的合并

● 分区再分配（例）

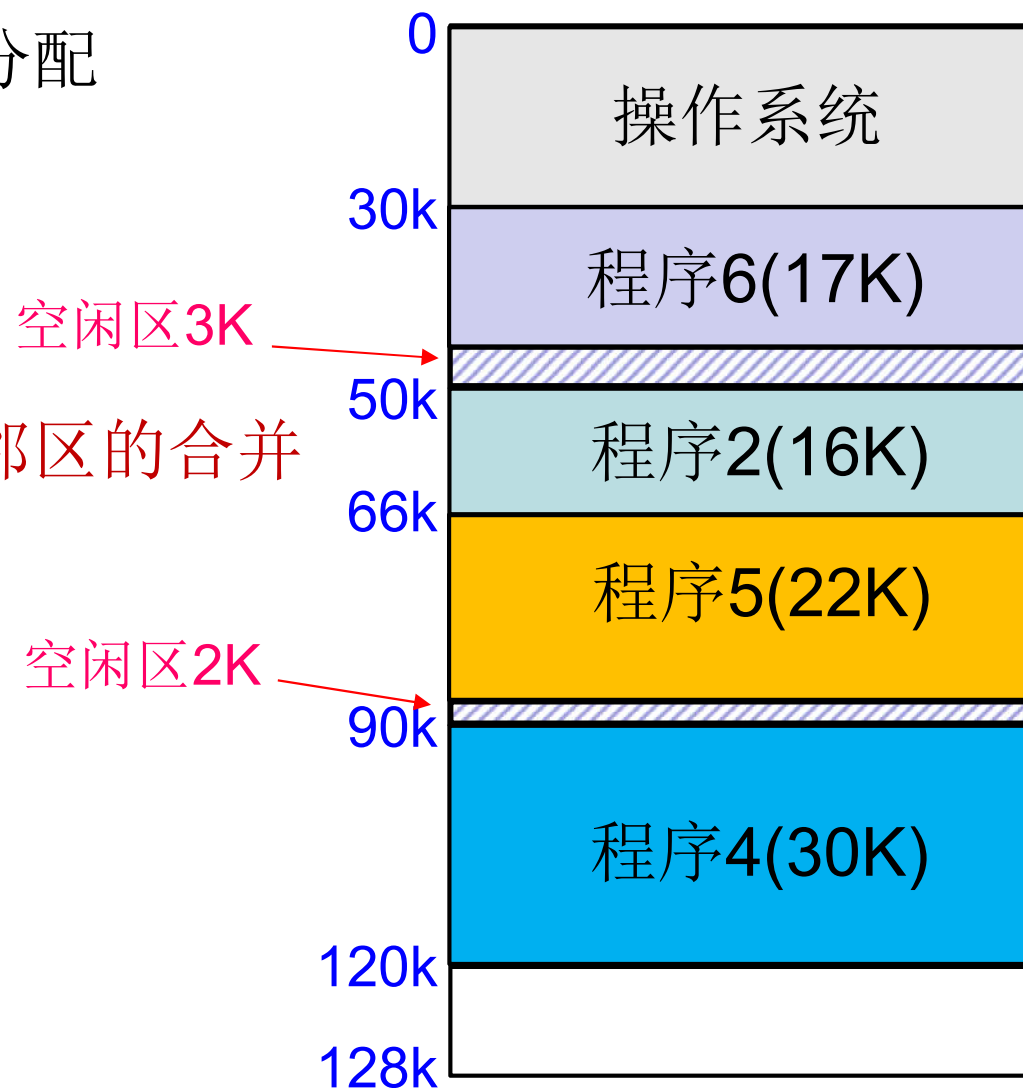
■ 程序5（22K）：装入

■ 程序6（17K）：装入

● 内存碎片

■ 过小的空闲区

■ 难以实际利用



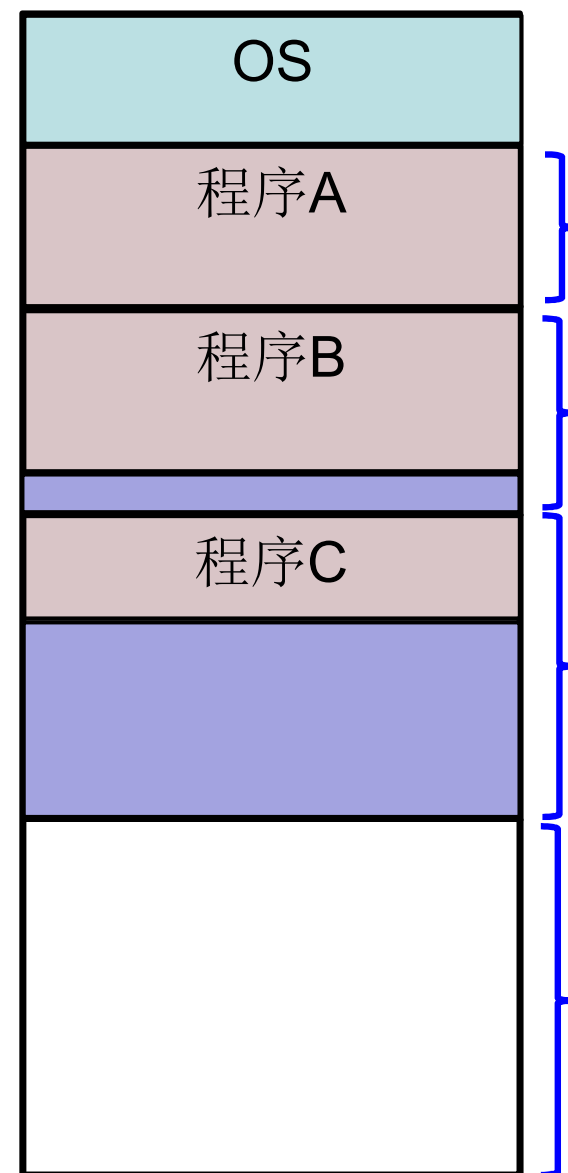
内存碎片

● 外部碎片

- 在所有分区之外的碎片（单独的分区）
- 例：动态分区的某分区分割后剩下的部分

● 内部碎片

- 分区内部出现的碎片。
 - ◆ 例：固定分区的某分区剩下部分。
 - ◆ 例：分页系统引起的页内碎片



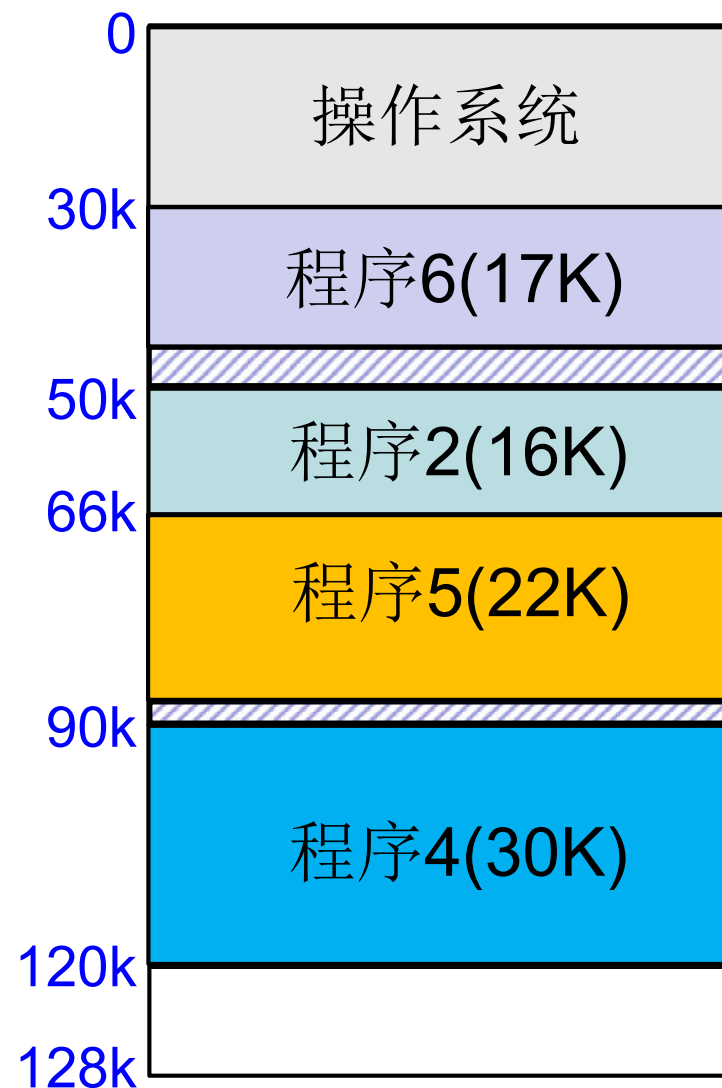
动态分区的特点

● 特点

- 分区的个数和大小均可变
- 存在内存碎片（外部碎片）

● 动态分区需要解决的问题

- 分区的选择？
- 分区的分配？
- 分区的回收？
- 内存碎片问题？



分区的选择(放置策略)

- 空闲区表

- 描述内存空闲区的位置和大小的数据结构

- 分区选择

例: 程序x (8K)

- 选择一个足够大的空闲区(给用户程序使用)

- 选择策略: 放置策略

- ◆ 地址最小的空闲区

- 空闲区表按地址递增排序(首次适应法)

- ◆ 地址最大的空闲区

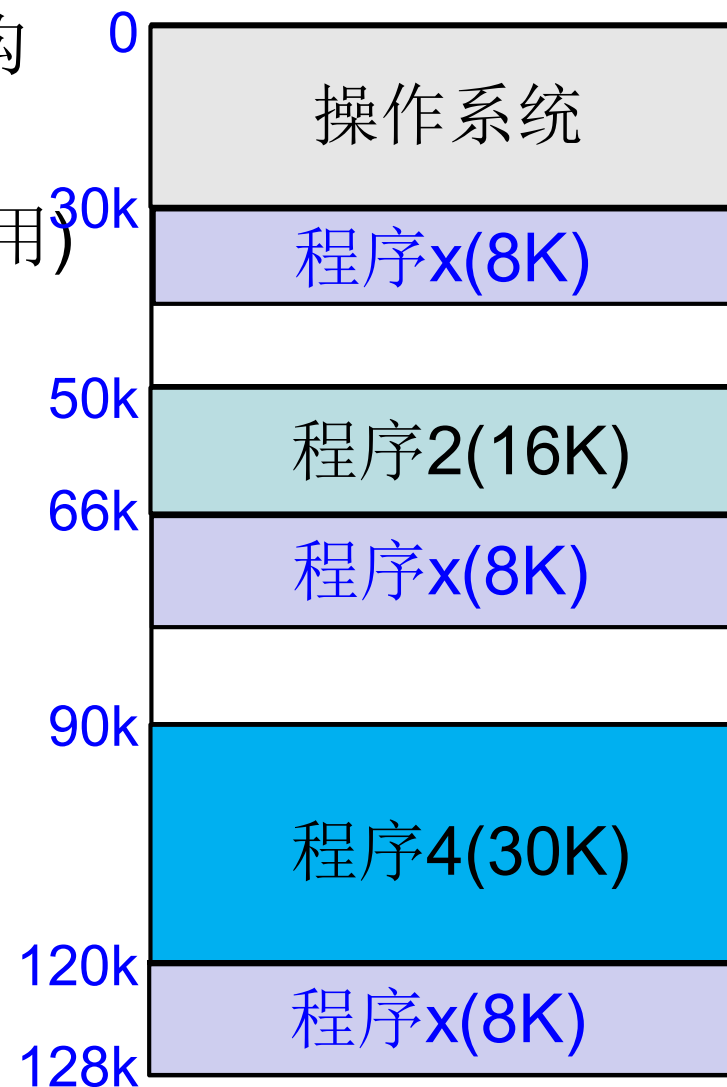
- 空闲区表按地址递减排序

- ◆ 最小的空闲区

- 空闲区表按大小递减排序(最佳适应法)

- ◆ 最大的空闲区

- 空闲区表按大小递增排序(最差适应法)



首次适应法

- 排序方式
 - 空闲区表按**首址递增**排序
- 目的
 - 尽可能先利用**低地址空间**

递增

| 位置 | 大小 |
|------|-----|
| 30K | 20K |
| 66K | 24K |
| 120K | 8K |

最佳适应法

- 排序方式
 - 空闲区表按**大小递增**排序
- 目的
 - 尽量先选中满足要求的**最小**空闲区

| 位置 | 大小 |
|------|-----|
| 120K | 8K |
| 30K | 20K |
| 66K | 24K |

递增

最坏适应法

- 排序方式

- 空闲区表按**大小递减**排序

- 目的

- 尽量先使用**最大的**空闲区

- 仅作**一次查找**就可找到所要分区。

| 位置 | 大小 |
|------|-----|
| 66K | 24K |
| 30K | 20K |
| 120K | 8K |

递减

分区的分配

- 功能

- 从用户选中的分区中分配/分割所需大小给用户
- 剩余部分（若有）依然作为空闲区登记

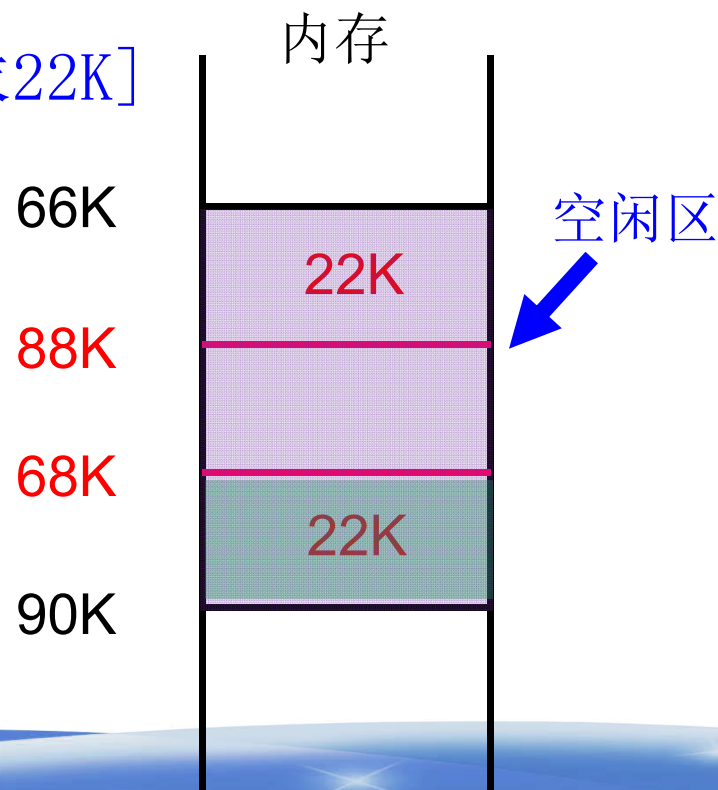
- 注意

- 分割空闲区时一般把（底部）分割给用户。

空闲区表

[例：用户要求22K]

| 位置 | 大小 |
|------|-----|
| 30K | 20K |
| 66K | 24K |
| 120K | 8K |



碎片问题

- 动态分区的缺点

- 容易产生内存碎片：内存反复分配和分割

- 首次适应法 | **最佳适应法** | 最坏适应法？

- 解决碎片的办法

- 规定门限值

- ◆ 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。

- 内存拼接技术

- ◆ 将所有空闲区**集中**一起构成一个大的空闲区。

- ◆ 拼接的时间和缺点？

碎片问题

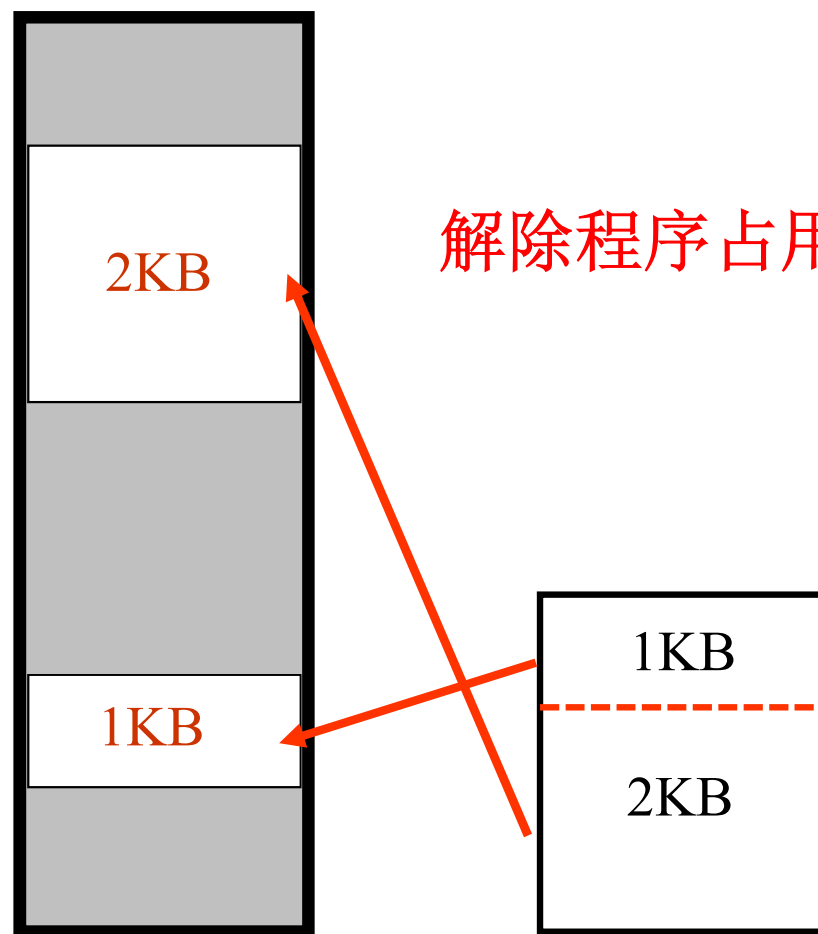
- 拼接的时机
 - 释放区回收的时候
 - ◆ 拼接频率过大，系统开销大
 - 系统找不到足够大的空闲区时
 - ◆ 空闲区的管理复杂
 - 定期
 - ◆ 空闲区的管理复杂
- 拼接技术的缺点
 - 消耗系统资源；
 - 离线拼接；
 - 重新定义作业

碎片问题

- 解决碎片的办法（续）

- 把程序分拆几个部分装入不同分区，充分利用碎片。

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

- 目的
 - 在较小的内存空间中运行较大的程序
- 内存分区
 - 常驻区：被某段单独且固定地占用的区域，可划分多个
 - 覆盖区：能被多段共用（覆盖）的区域，可划分多个

| OS |
|-------------|
| 覆盖区2 40K |
| 覆盖区1 30K |
| 常驻区2 20K |
| 常驻区1 20K |

用户内存（110K）

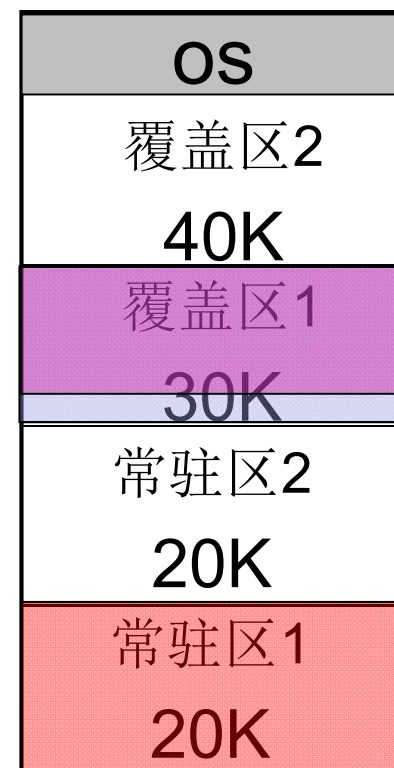
覆盖——Overlay

● 工作原理

- 程序分成**若干**代码段或数据段
- 将程序**常用的段**装入常驻区；（**核心段**）
- 将程序**不常用段**装入覆盖区；
 - ◆ **正运行的段**处于覆盖区；
 - ◆ **暂时不运行的段**放在**硬盘**中(**覆盖文件**)；
 - ◆ **即将运行的段**装入**覆盖区**（**覆盖**旧内容）；

● 意义

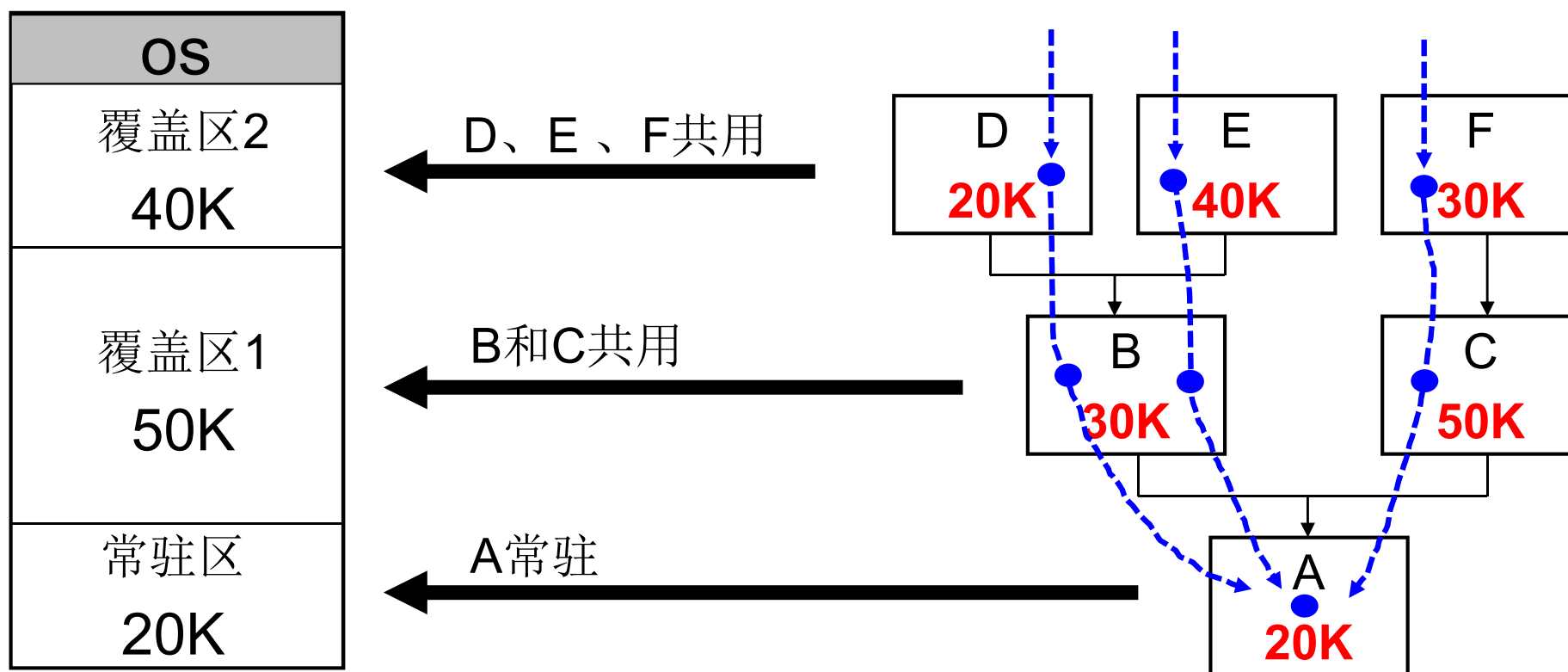
- 减少程序对内存需求



● 覆盖的例子

■ 内存（110K）：一个常驻区，两个覆盖区

■ 程序（190K）：多个模块（段）

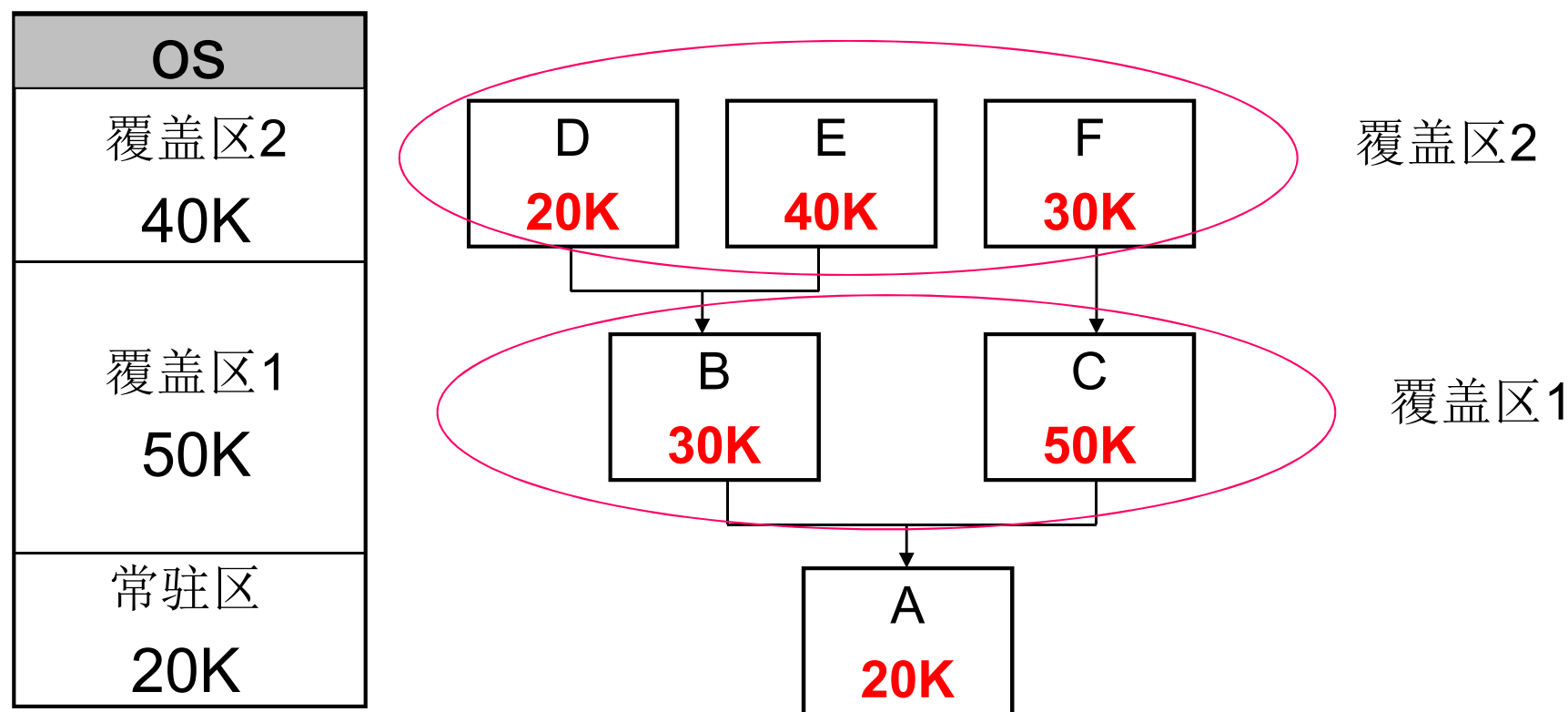


用户内存 (110K)

用户程序 (190K)

● 覆盖的缺点

- 编程复杂：程序员划分程序模块并确定覆盖关系。
- 程序执行时间长：从外存装入内存耗时



用户内存 (110K)

用户程序 (190K)

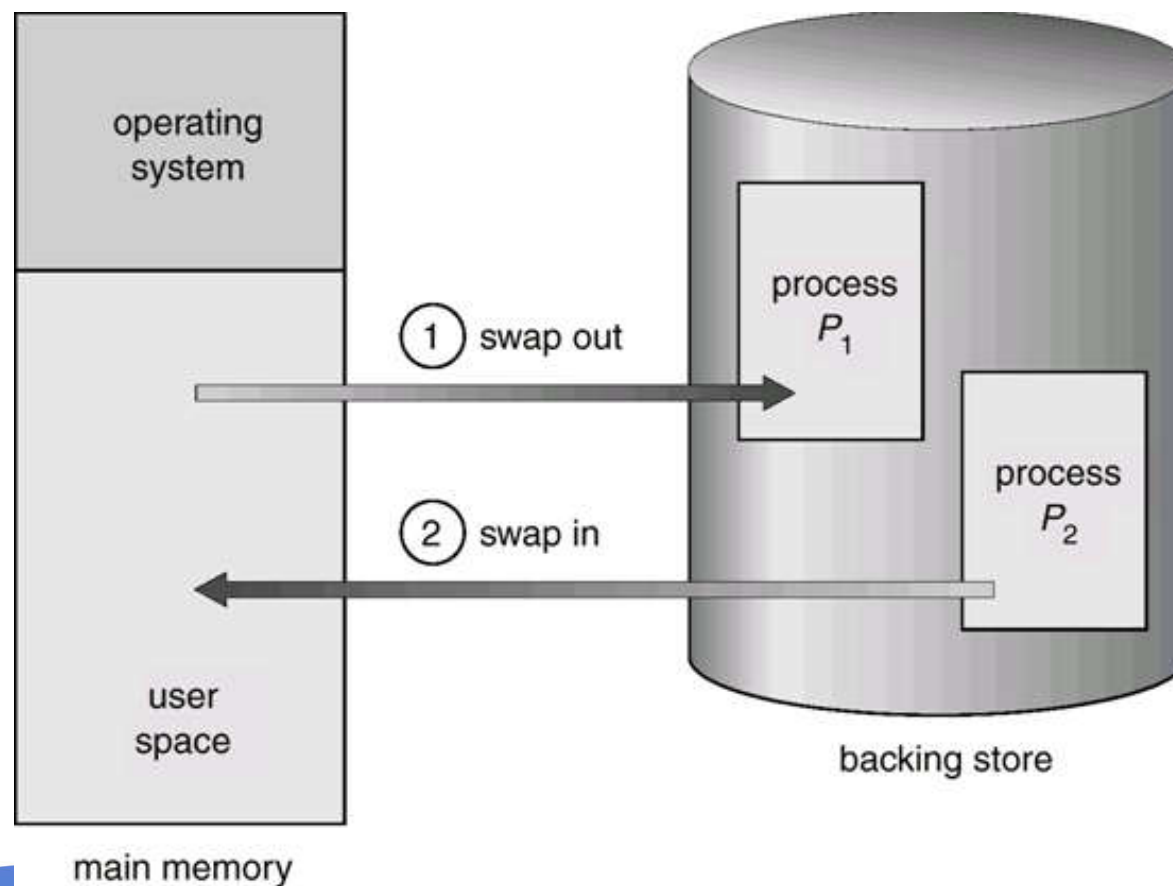
对换技术——Swapping

● 原理

- 内存不够时把进程写到磁盘（**换出/Swap Out**）。
- 当进程要运行时重新写回内存（**换入/Swap In**）。

● 优点

- 增加进程并发数；
- 不考虑程序结构。



- 对换技术的缺点

- 换入和换出增加CPU开销;
- 对换单位太大 (整个进程)

- 需要考虑的问题

- 程序换入时的地址重定位
- 减少对换传送的信息量
- 外存对换空间的管理方法

- 采用交换技术的OS

- **UNIX, Linux, Windows**



7.4 虚拟内存管理

页式内存管理的概念

- 页式内存管理的目的

- 小内存中运行大程序和多个程序

- ◆ 减少程序运行对内存的需求

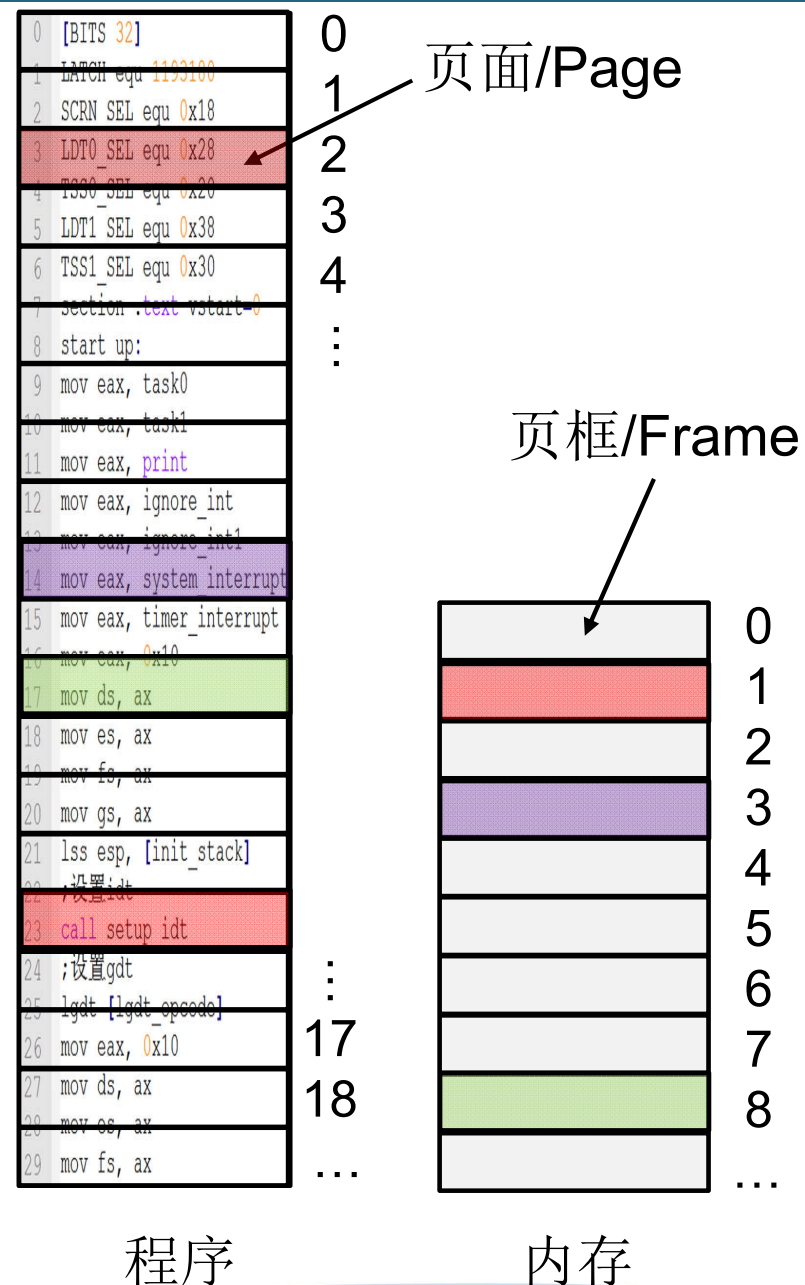
- 程序和内存的划分

- 程序和内存都划成等大小（例 4KB）的小片：页面和页框

- 程序的装入方式

- 程序以页面为单位装入页框

- 内存以页框为单位使用



页式内存管理的原理

● 程序运行的局部性

- 在任何**有限时段**内程序的运行活动**一般**局限在**有限范围**内。

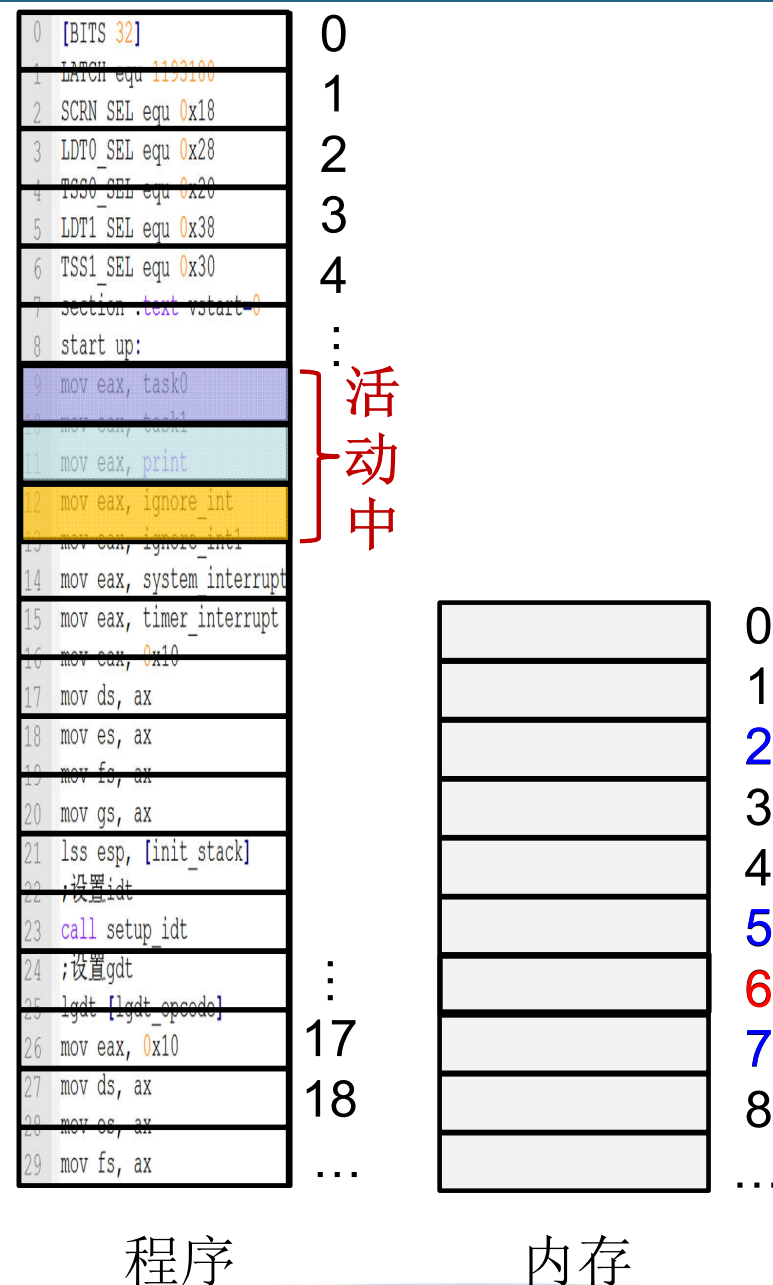
● 仅把程序的局部装入内存中

- 仅把**当前时段**涉及的**部分页面**装入内存(**可让程序短暂运行**)

- 运行过程逐步装入**新页面**

◆ 已运行过的**旧页面**可**删除**

- **意义**：确保有限时段内程序只占用了少量的内存。



页式虚拟存储管理

● 进程装入和使用内存的原则

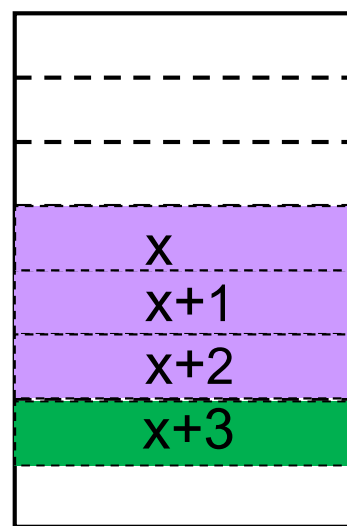
■ 原则：局部装入，不断更新

◆ 只把程序部分页装入内存便可运行。

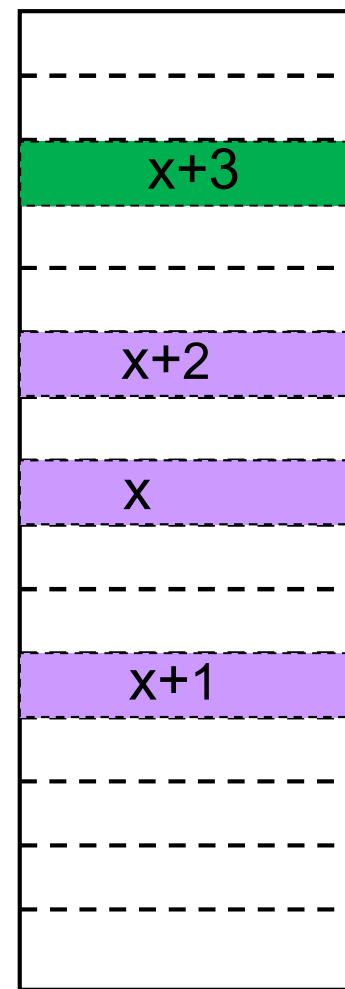
◆ 页在内存中占用的页框不必相邻。

◆ 需要新页时，按需从硬盘调入内存。

◆ 不再运行的页及时删除，腾出空间



进程



内存

页式虚拟存储管理

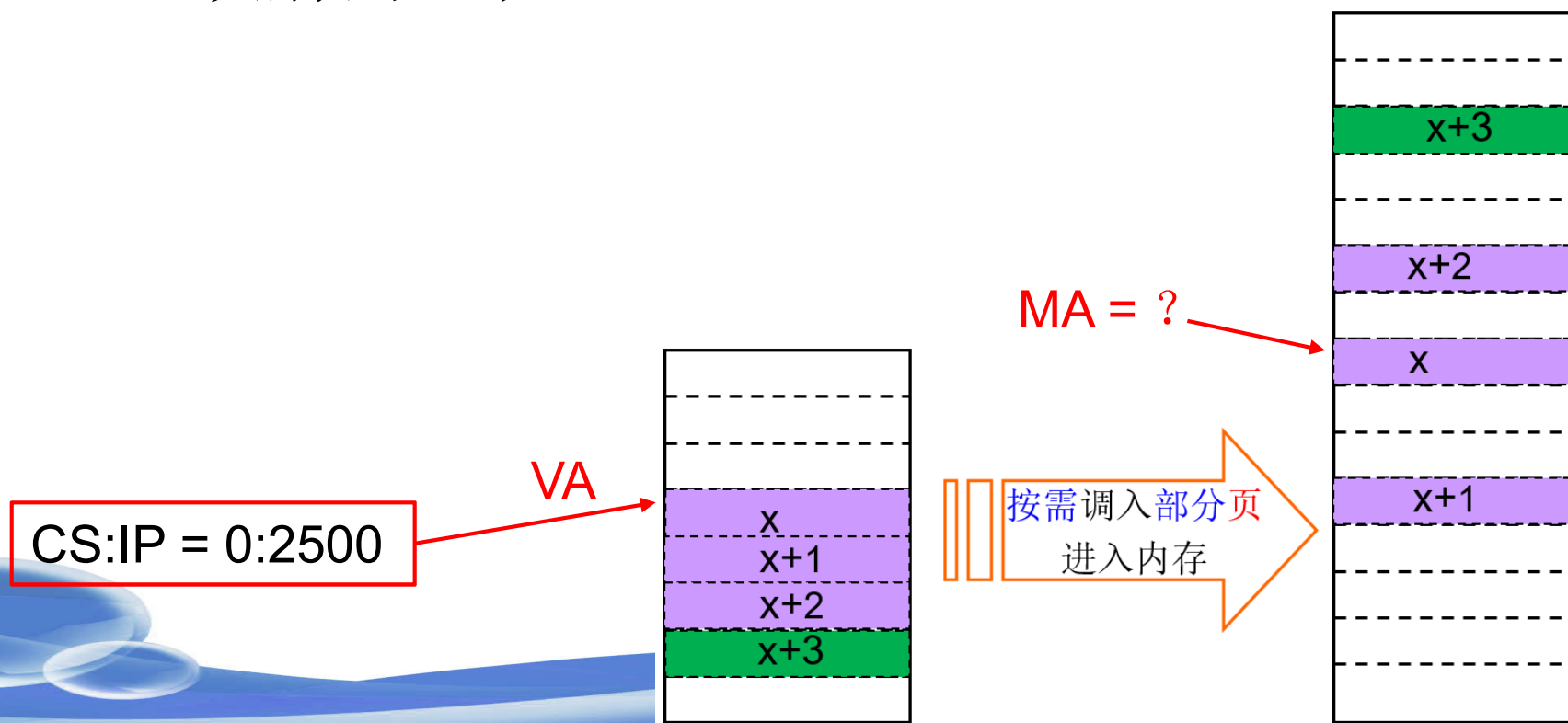
- 页式虚拟内存管理需要解决的问题

- 1. 虚拟地址如何组织或表达？

- ◆ 页式地址

- 2. 虚拟地址如何转化为物理地址（地址映射）？

- ◆ 地址映射的过程



页式系统中的地址

- 虚拟地址**VA**是线性的，从**0**开始
- **VA**分成页号**P**和页内偏移**W**
 - 页号 (P)
 - ◆ 所处页编号 = $VA / \text{页大小}$
 - 页内偏移(W)
 - ◆ 所处页内的偏移 = $VA \% \text{页大小}$
- 例子
 - $VA = 2500$; 页面大小1KB

$$P = 2500 / 1024 = 2$$

$$W = 2500 \% 1024 = 452$$

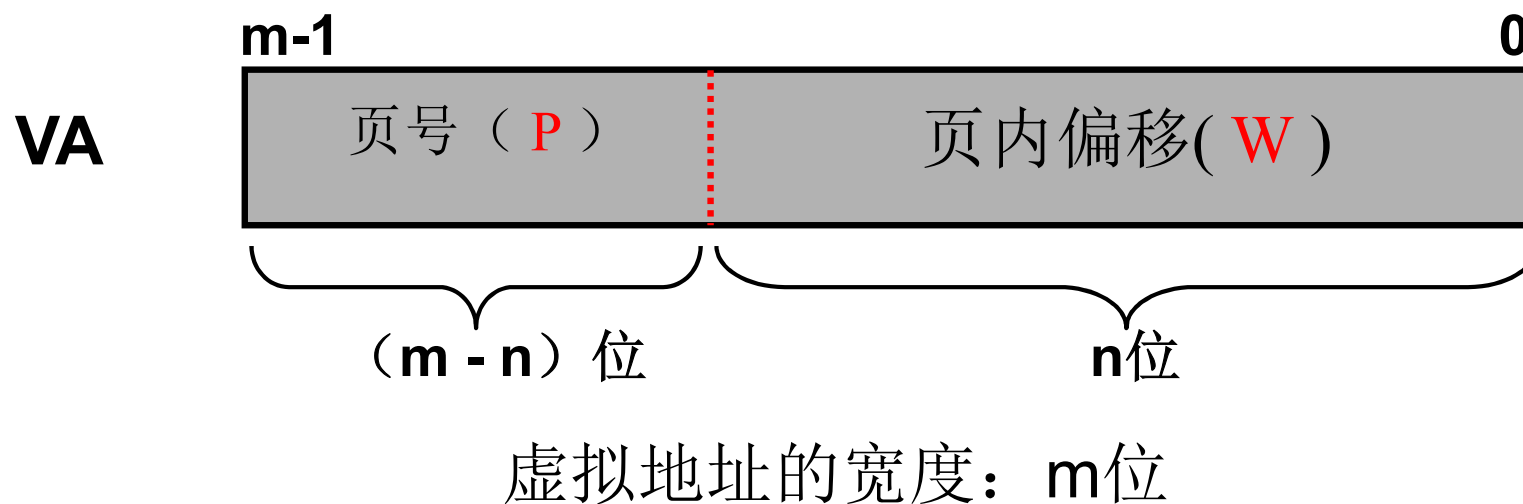
```
0 [BITS 32]
1 LATCH equ 1193180
2 SCRN_SEL equ 0x18
3 LDT0_SEL equ 0x28
4 TSS0_SEL equ 0x20
5 LDT1_SEL equ 0x38
6 TSS1_SEL equ 0x30
7 section .text vstart=0
8 start_up:
9 mov eax, task0
10 mov eax, task1
11 mov eax, print
12 mov eax, ignore_int
13 mov eax, ignore_int1
14 mov eax, system_interrupt
15 mov eax, timer_interrupt
16 mov eax, 0x10
17 mov ds, ax
18 mov es, ax
19 mov fs, ax
20 mov gs, ax
21 lss esp, [init_stack]
22 ;设置idt
23 call setup_idt
24 ;设置gdt
25 lgdt [lgdt_opcode]
26 mov eax, 0x10
27 mov ds, ax
28 mov es, ax
29 mov fs, ax
```

P和W的另一种计算方法(移位和位与)

● P和W计算(假定：页的大小： 2^n 单元)

■ 页号 $P = VA \gg n = VA / 2^n$

■ 页内偏移 $W = VA \text{ 低 } n \text{ 位} = VA \&\& (2^n - 1) = VA \% 2^n$



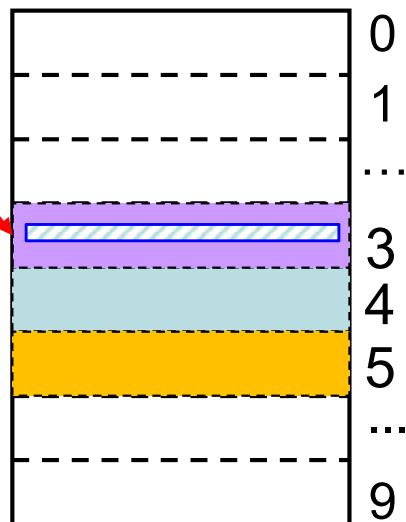
地址映射过程

● 页面映射表/页表

| 页号 | 页框号 |
|-----|-----|
| ... | ... |
| 3 | 8 |
| 4 | 11 |
| 5 | 6 |
| ... | ... |

MOV AX, [12688]

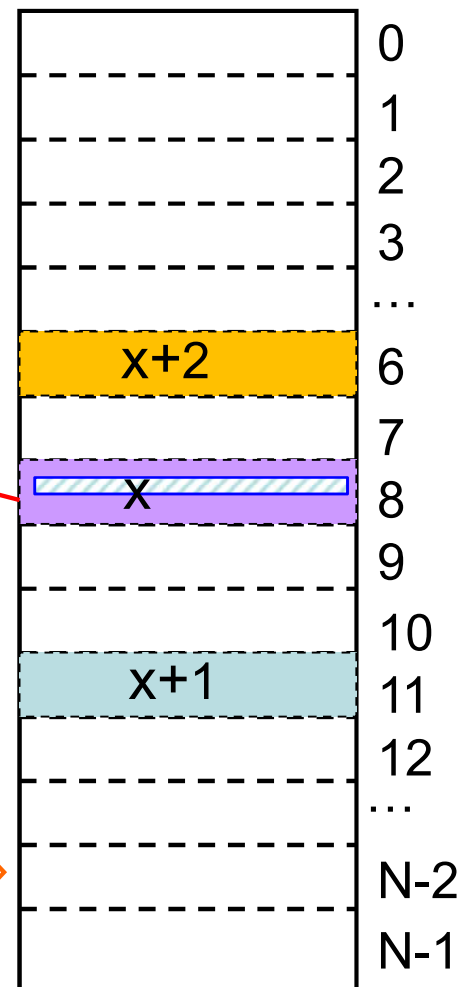
DS:12688=0:12688



虚拟内存空间

哪个页框?

按需调入部分页
进入内存



物理内存空间

地址映射过程

● 页面映射表/页表

■ 记录页面与页框(块)之间的对应关系。也叫页表。

| 页号 | 页框号 | 页面其它特性 |
|----|-----|--------|
| 0 | 5 | ... |
| 1 | 65 | ... |
| 2 | 13 | ... |

■ 页号：登记程序地址的页号。

■ 页框号：登记页所在的物理页号。

■ 页面其他特性：登记含存取权限在内的其他特性。

地址映射过程

● 地址映射

问题：MOV AX, 12688 执行时共访问了内存几次？

■ 虚拟地址（页式地址） → 物理地址

■ 例子：VA = 12688, MA=? (页 = 4KB)

■ (1) $P = VA / \text{页大小} = 12688 / 4096 = 3$

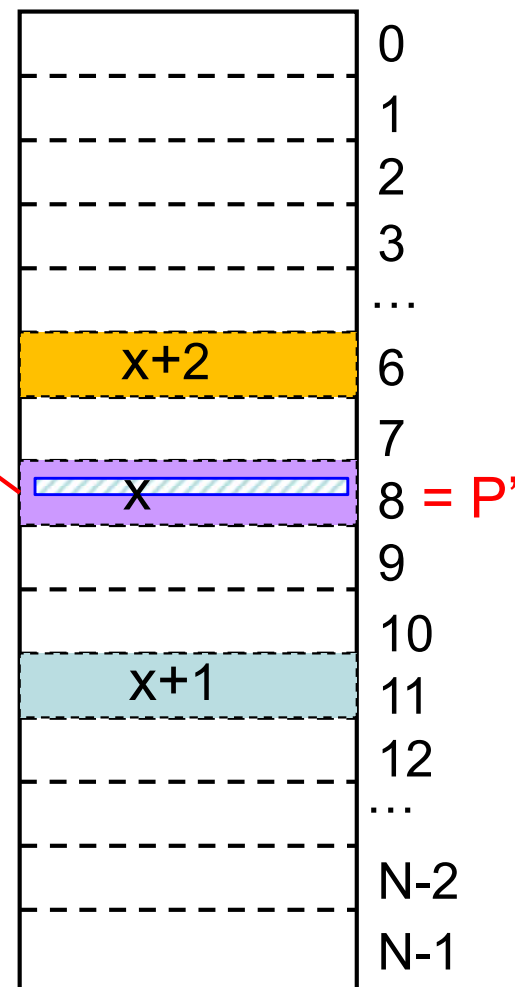
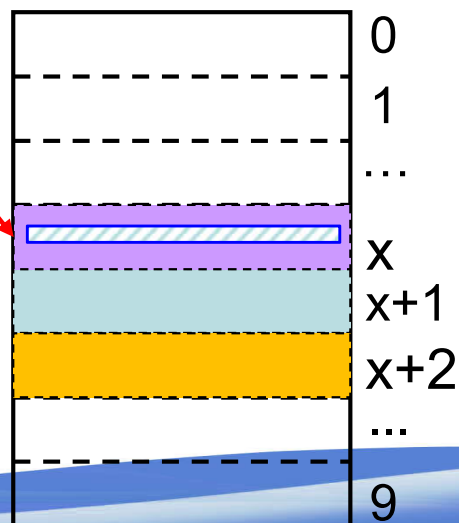
$W = VA \% \text{页大小} = 12688 \% 4096 = 400$

■ (2) 查页表, 查页框号 $P' = 8$

■ (3) 计算 $MA = P' \times \text{页大小} + \text{页内偏移} W$
 $= 8 \times 4096 + 400 = 33168$

MOV AX, [12688]

DS:BX=0:12688



物理内存空间

地址映射过程

● 地址映射

■ 虚拟地址（页式地址） → 物理地址

■ 例子：VA = 12688, MA=?

■ (1) $P = VA / \text{页大小}$

$W = VA \% \text{页大小}$

■ (2) 查页表, 查页框号 P'

■ (3) 计算 $MA = P' \times \text{页大小} + \text{页内偏移} W$

■ 注意 $= P' \ll n \parallel \text{页内偏移} W$

◆ 页面的大小

□ 4K, 2K, 1K, 16K,

◆ 页表的地址

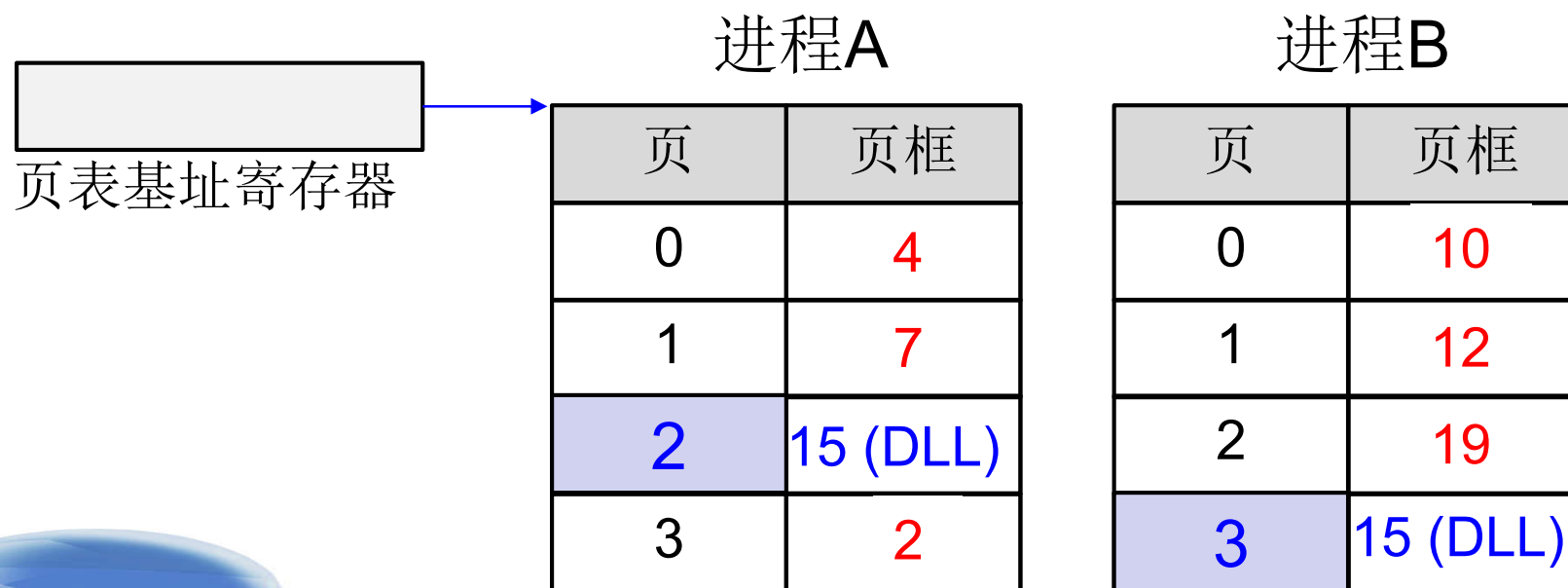
□ 页表基址寄存器

◆ CPU 计算 MA: 左移, 位或

| 页号 | 页框号 |
|-----|-----|
| ... | ... |
| 3 | 8 |
| 4 | 11 |
| 5 | 6 |
| ... | ... |

页表的建立

- 操作系统为每个**进程**建立一个**页表**
 - 页表的**基址**存放在**进程控制块**中。
 - 页表的内容由内核负责填充和更新
- 当前进程的页表驻留在内存
 - 页表基址：**页表基址寄存器**



缺页中断

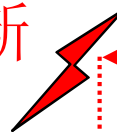
- 缺页中断的定义

- 当程序要访问的**目标页面**不在内存时，程序将被迫**临时中断**：**缺页中断**

- 缺页中断的处理

- 立即将所缺**页面**装入内存。
 - ◆ 页面从**硬盘**拷贝到内存
 - I/O操作，**耗时较长**
- 缺页中断降低了程序实时性

缺页中断



| | | |
|----|---------------------------|-----|
| 0 | [BITS 32] | 0 |
| 1 | LATCH equ 1193100 | 1 |
| 2 | SCRN_SEL equ 0x18 | 2 |
| 3 | LDT0_SEL equ 0x28 | 3 |
| 4 | TSS0_SEL equ 0x20 | 4 |
| 5 | LDT1_SEL equ 0x38 | 5 |
| 6 | TSS1_SEL equ 0x30 | 6 |
| 7 | section .text vstart=0 | ... |
| 8 | start up: | |
| 9 | mov eax, x+0 | |
| 10 | mov ecx, 0x10 | |
| 11 | mov eax, x+1 | |
| 12 | mov eax, x+2 | |
| 13 | mov ecx, ignore_int1 | |
| 14 | mov ecx, system_interrupt | |
| 15 | mov ecx, timer_interrupt | |
| 16 | mov ecx, 0x10 | |
| 17 | mov ds, ax | |
| 18 | mov es, x+7 | |
| 19 | mov fs, ax | |
| 20 | mov gs, ax | |
| 21 | lss esp, [init_stack] | |
| 22 | ;设置idt | |
| 23 | call setup_idt | |
| 24 | ;设置gdt | |
| 25 | lgdt [lgdt_opcode] | |
| 26 | mov ecx, 0x10 | 17 |
| 27 | mov ds, ax | 18 |
| 28 | mov es, ax | ... |
| 29 | mov fs, ax | |

程序

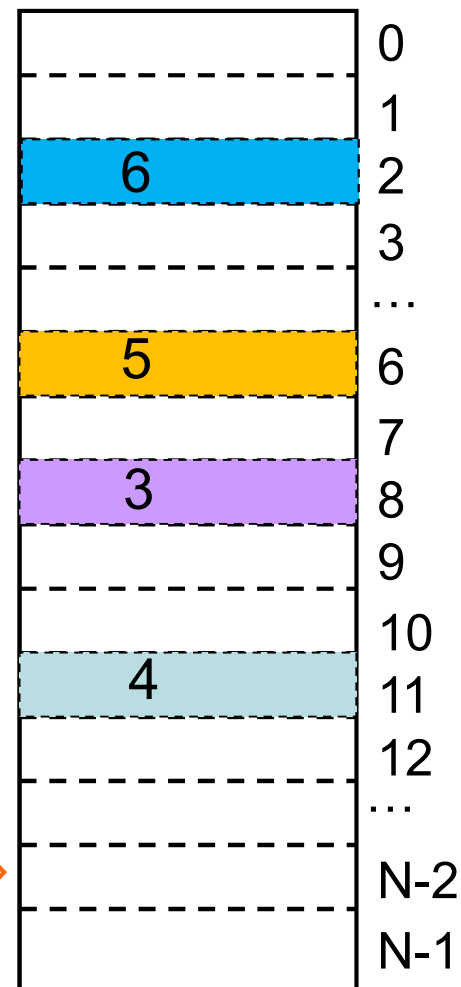
| | |
|--|-----|
| | 0 |
| | 1 |
| | 2 |
| | 3 |
| | 4 |
| | 5 |
| | 6 |
| | 7 |
| | 8 |
| | ... |

内存

缺页中断

- 例：目的页不在内存产生异常

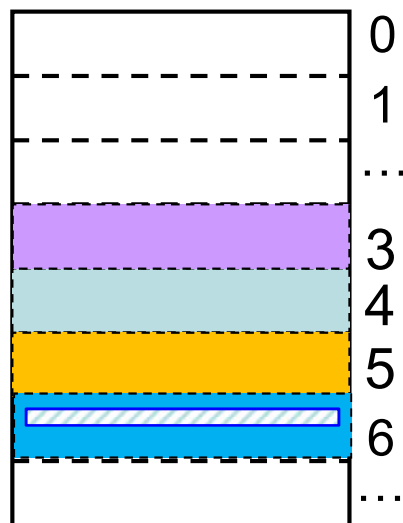
| 页号 | 页框号 |
|-----|--------|
| ... | ... |
| 3 | 8 |
| 4 | 11 |
| 5 | 6 |
| 6 | #\$*!% |



物理内存空间

MOV AX, [24776]

DS:24776=0:24776



虚拟内存空间



缺页中断

● 扩充有中断位I和辅存地址的页表

| 页号 | 页框号 | 中断位I | 辅存地址 |
|----|--------|------|------|
| 5 | 6 | 0 | 2020 |
| 6 | #\$*!% | 1 | 2022 |

■ 中断位I ——标识该页是否在内存？

◆ 若 $I = 1$ ， 不在内存

◆ 若 $I = 0$ ， 在内存

■ 辅存地址——该页在辅存上的位置

页表扩充——带访问位和修改位的页表

- 扩充有访问位（引用位）和修改位（**Dirty**）的页表

| 页号 | 页框号 | 访问位 | 修改位 |
|----|-----|-----|-----|
| | | 1 | 0 |
| | | 0 | 1 |

■ 访问位——标识该页最近是否被访问？

◆ 0 ——最近没有被访问

◆ 1 ——最近已被访问

■ 修改位——标识该页的数据是否已被修改？

◆ 0 ——该页未被修改

◆ 1 ——该页已被修改

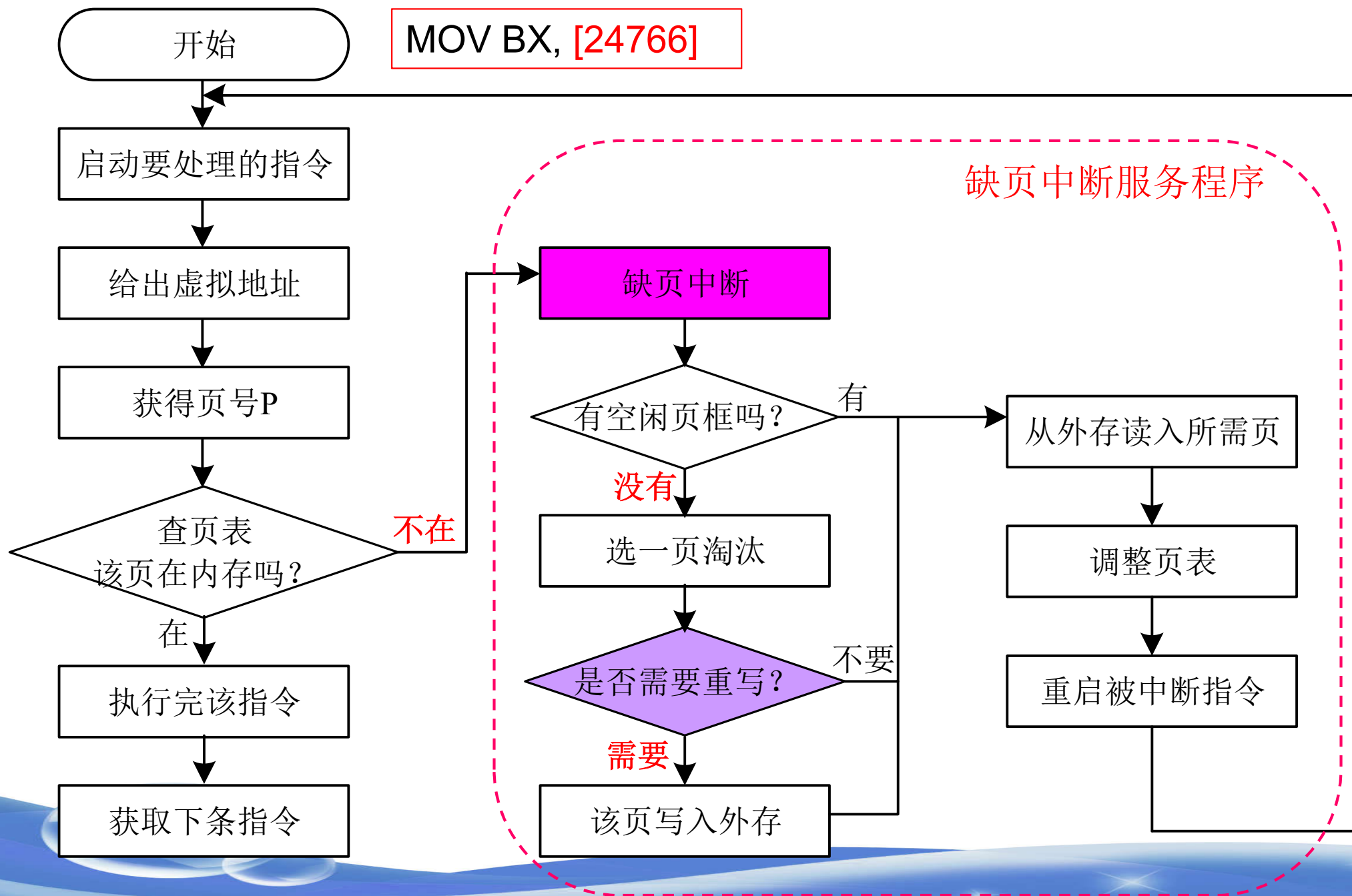
缺页中断

● 缺页中断处理程序

- 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。

| 页号 | 页框号 | 中断位I | 辅存地址 |
|----|-----|------|------|
| 5 | 6 | 0 | 2020 |
| 6 | 13 | 0 | 2022 |

访存指令的执行过程（含缺页中断处理）



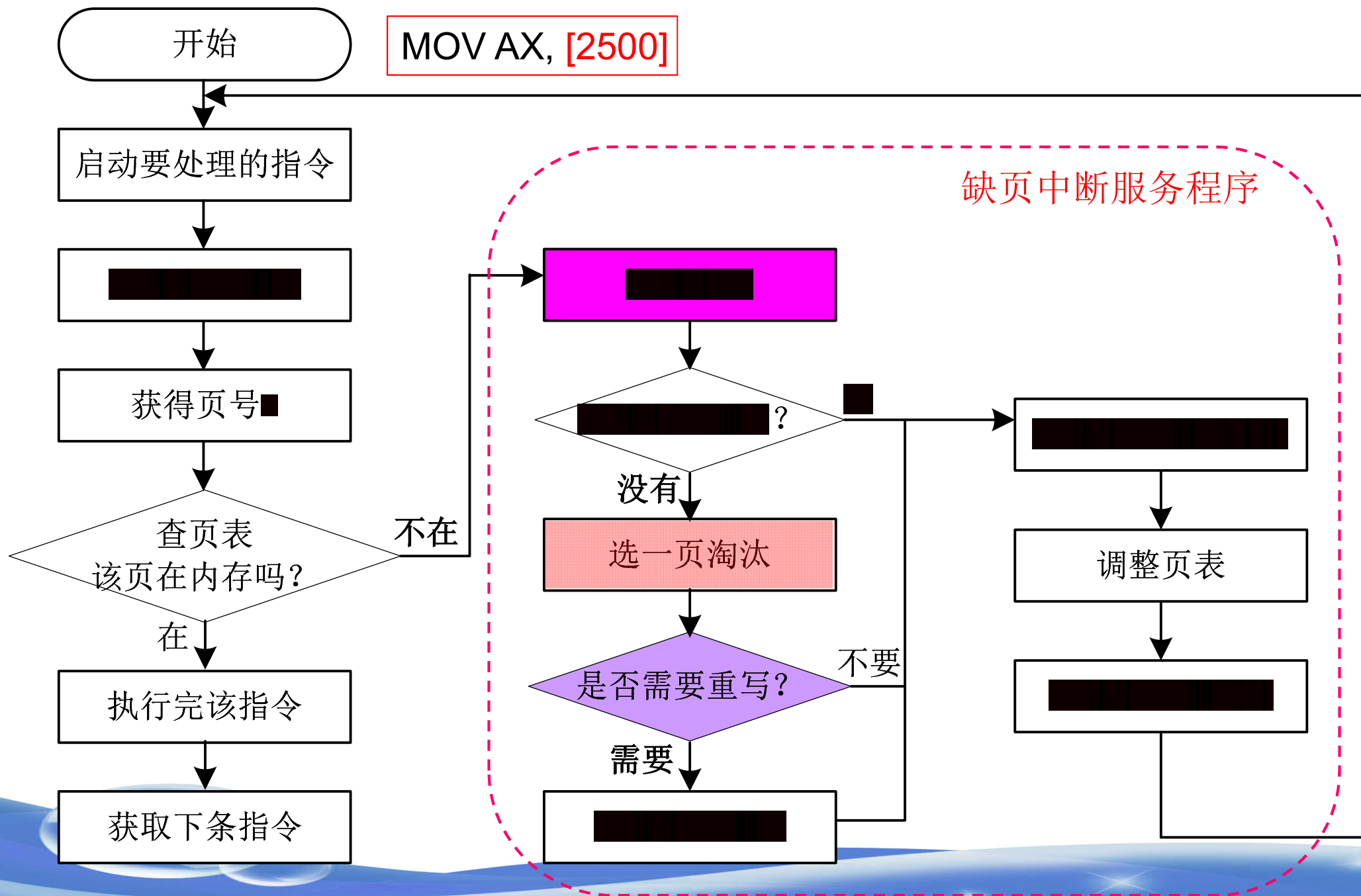
缺页中断

- 缺页（中断）率

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

- 命中率 $= 1 - f$

淘汰策略



淘汰策略

- 淘汰策略

- 选择淘汰哪一页的规则称淘汰策略。

- 页面抖动

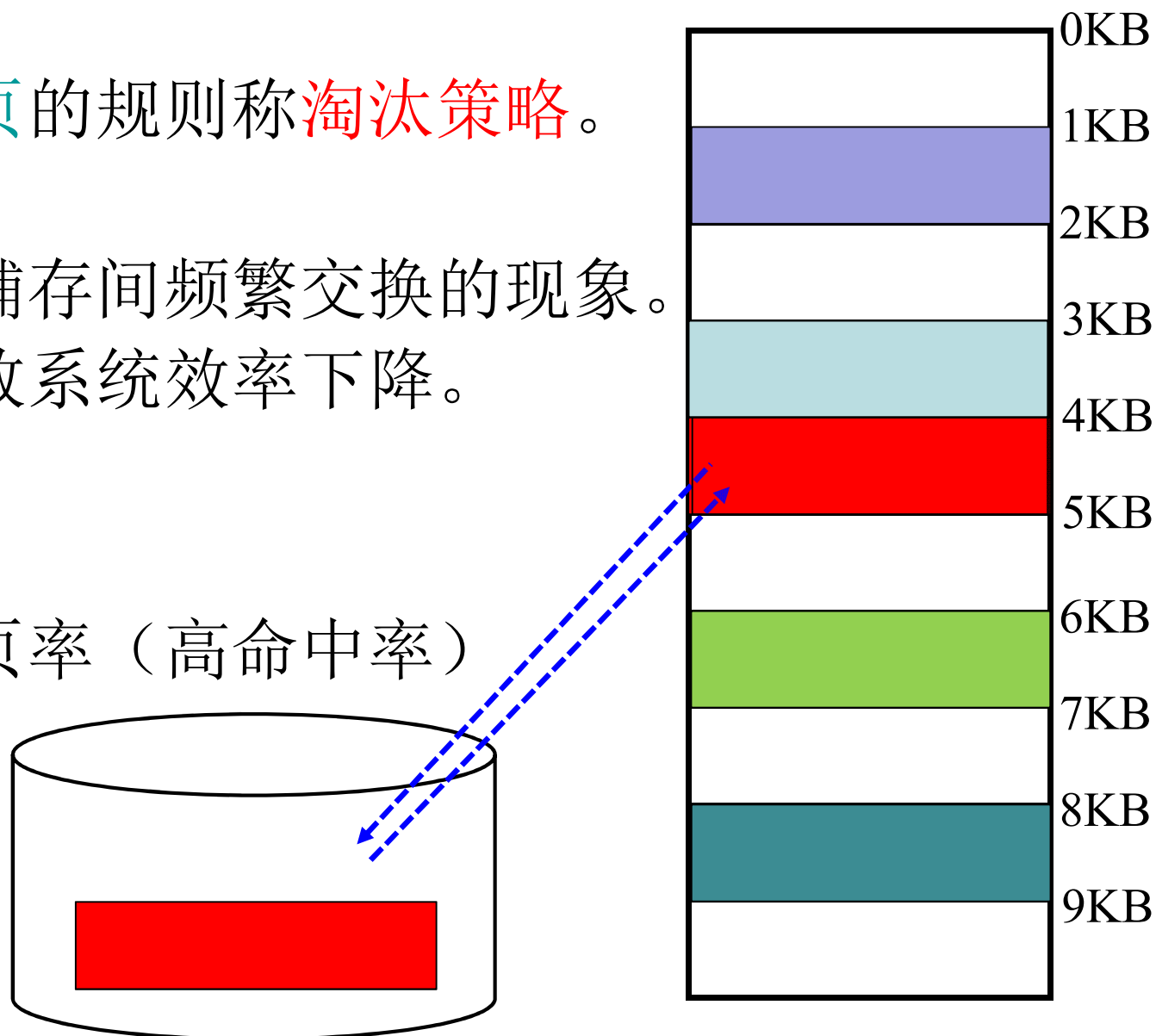
- 页面在内存和辅存间频繁交换的现象。

- “抖动”会导致系统效率下降。

- 好的淘汰策略

- 页面抖动较少

- 具有较低的缺页率（高命中率）



淘汰策略

● 常用的淘汰算法

- 最佳算法（**OPT**算法）
- 先进先出淘汰算法（**FIFO**算法）
- 最久未使用淘汰算法（**LRU**算法）
- 最不经常使用（**LFU**）算法



最佳算法（OPT算法, **Optimal**）

- 思想

- 淘汰不再需要或最远将来才会用到的页面。



最佳算法（OPT算法, Optimal）

- 思想

- 淘汰不再需要或最远将来才会用到的页面。

- 例子

- 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 | A | C | C |
| | | B | B | B | B | B | B | B | B | B | B | D |
| | | | C | C | D | D | D | 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）

- 特点

- 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）

- 思想

- 淘汰在内存中停留时间最长的页面



先进先出淘汰算法（FIFO算法）

- 思想

- 淘汰在内存中停留时间最长的页面

- 例子

- 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 | A | D | D | D | E | E | E | E | E |
| | | B | B | B | B | A | A | A | A | A | C | C |
| | | | C | C | C | C | B | B | B | B | B | D |
| 缺页 | X | X | X | X | X | X | X | | | X | X | |

缺页次数 = 9， 缺页率 = $9 / 12 = 75\%$

先进先出淘汰算法（FIFO算法）

- 优点

- 实现简单：页面按进入内存的时间排序，淘汰队头页面。
- 进程按顺序访问地址空间时抖动较少，缺页率较低。

- 异常现象

- 对于一些特定的访问序列，分配页框越多，缺页率越高！
- 重做：页框数为4。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。



先进先出淘汰算法（FIFO算法）

● FIFO异常的例子

■ 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)

- 思想

- 淘汰最长时间未被使用的页面。

- 例子

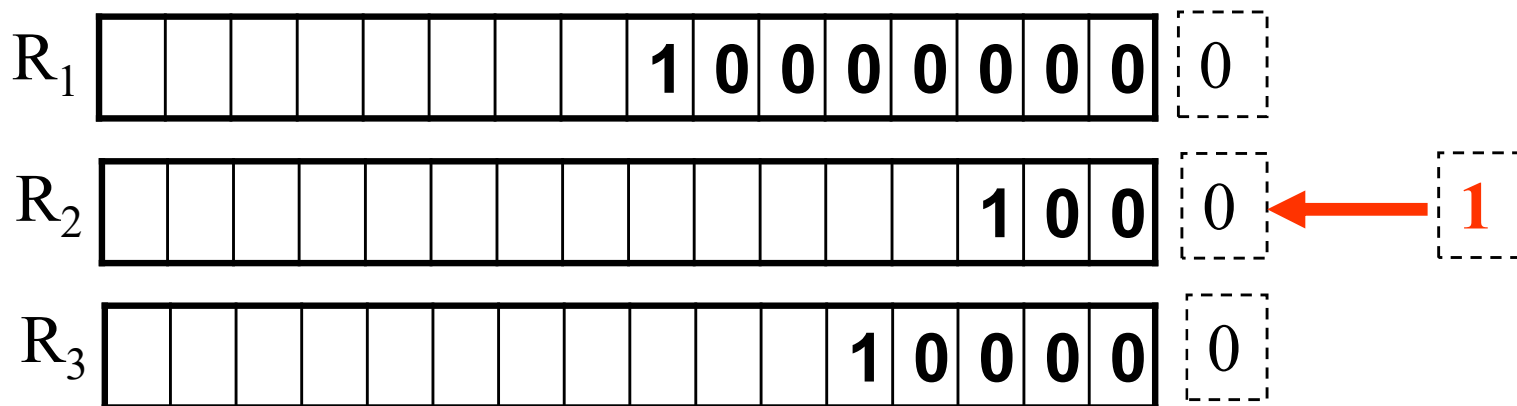
- 3个页框，页面序列：A,B,C,D,A,B,E,A,B,C,D,E。分析其按照LRU算法进行页面淘汰时的缺页情况。

| 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 | E | 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 = $10 / 12 = 83\%$

● LRU的实现（硬件方法）

- 页面设置一个移位寄存器R。页面被访问则重置1。
- 周期性地(周期很短)将所有页面的R左移1位（右边补0）



- 当需要淘汰页面时选择R值最大的页。
 - ◆ R值越大，对应页未被使用的时间越长。
- 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。
- 若R的位数太少，可能同时出现多个为0页面情况，难以比较。

最久未使用淘汰算法（LRU, Least Recently Used）

● LRU的实现（近似算法）

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

● 缺点

- 周期T难定
 - ◆ 太小，访问位为0的页过多，找不到合适的页淘汰。
 - ◆ 太大，全部页的访问位都为1，找不到合适的页淘汰。

| 页号 | 页框号 | 访问位 | ... |
|----|-----|-----|-----|
| | | 1 | ... |
| | | 0 | ... |

最不经常使用（LFU）算法

- Least Frequently Used

- 算法原则

- 选择到当前时间为止被访问次数最少的页面
- 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；
- 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。

页式内存管理

● 影响缺页次数的因素

- 淘汰算法

- 分配给进程的页框数

 - ◆ 页框越~~少~~，越容易缺页

- 页本身的大小

 - ◆ 页面越~~小~~，容易缺页



页式内存管理

- 页面的大小选择

- 页面的常见大小

- ◆ 2的整数次幂：1KB, 2KB, 4KB

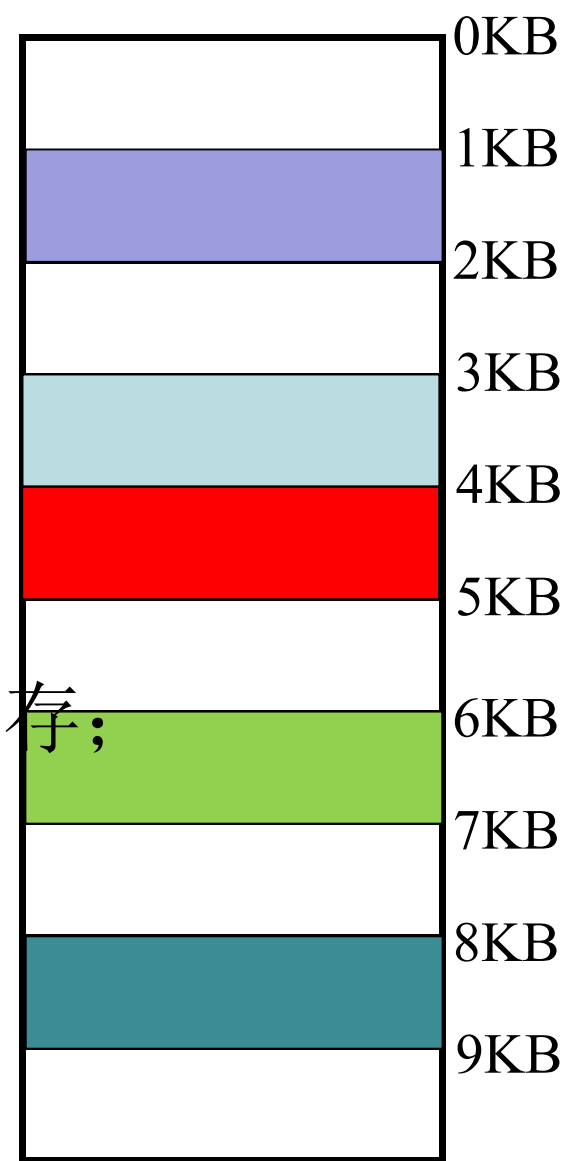
- 页面太大

- ◆ 浪费内存：极限是分区存储。

- 页面太小

- ◆ 页面增多，页表长度增加，浪费内存；

- ◆ 换页频繁，系统效率低



页式内存管理

● 影响缺页次数的因素

■ 淘汰算法

■ 分配给进程的页框数

◆ 页框越少，越容易缺页

■ 页本身的大小

◆ 页面越~~小~~，容易缺页

■ 程序的编制方法

◆



页式内存管理

● 程序的局部性

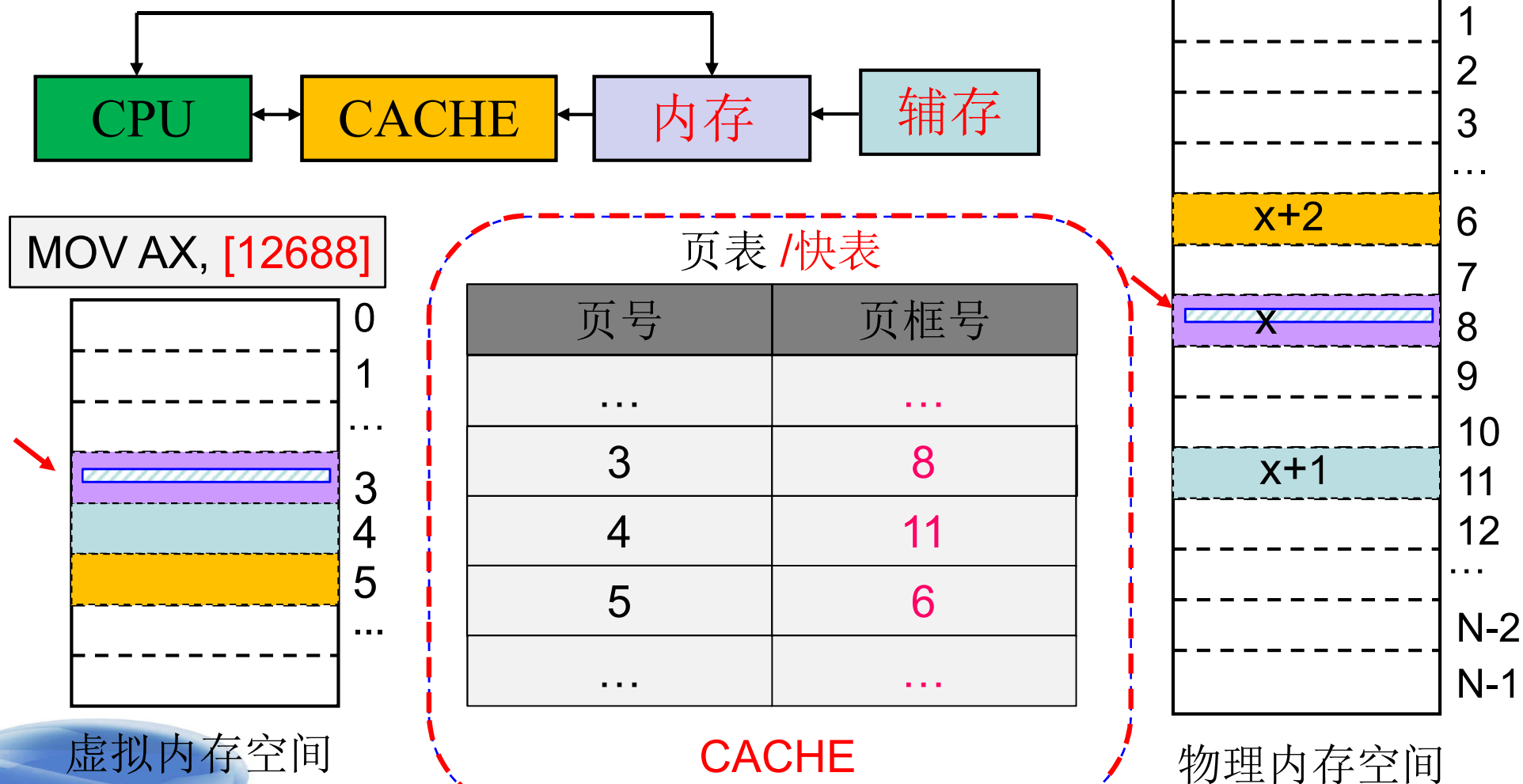
```
1 //遍历二维数组 (顺序:先行后列)
2 for( Col=0; Col<1000; Col++ )
3 {
4     for( Row=0; Row<1000; Row++ )
5     {
6         Matrix[Row][Col] = 0;
7     }
8 }
```

```
1 //遍历二维数组 (顺序:先列后行)
2 for( Row=0; Row<1000; Row++ )
3 {
4     for( Col=0; Col<1000; Col++ )
5     {
6         Matrix[Row][Col] = 0;
7     }
8 }
```

快表机制（Cache，联想存储器，TLB）的概念

● 分级存储体系

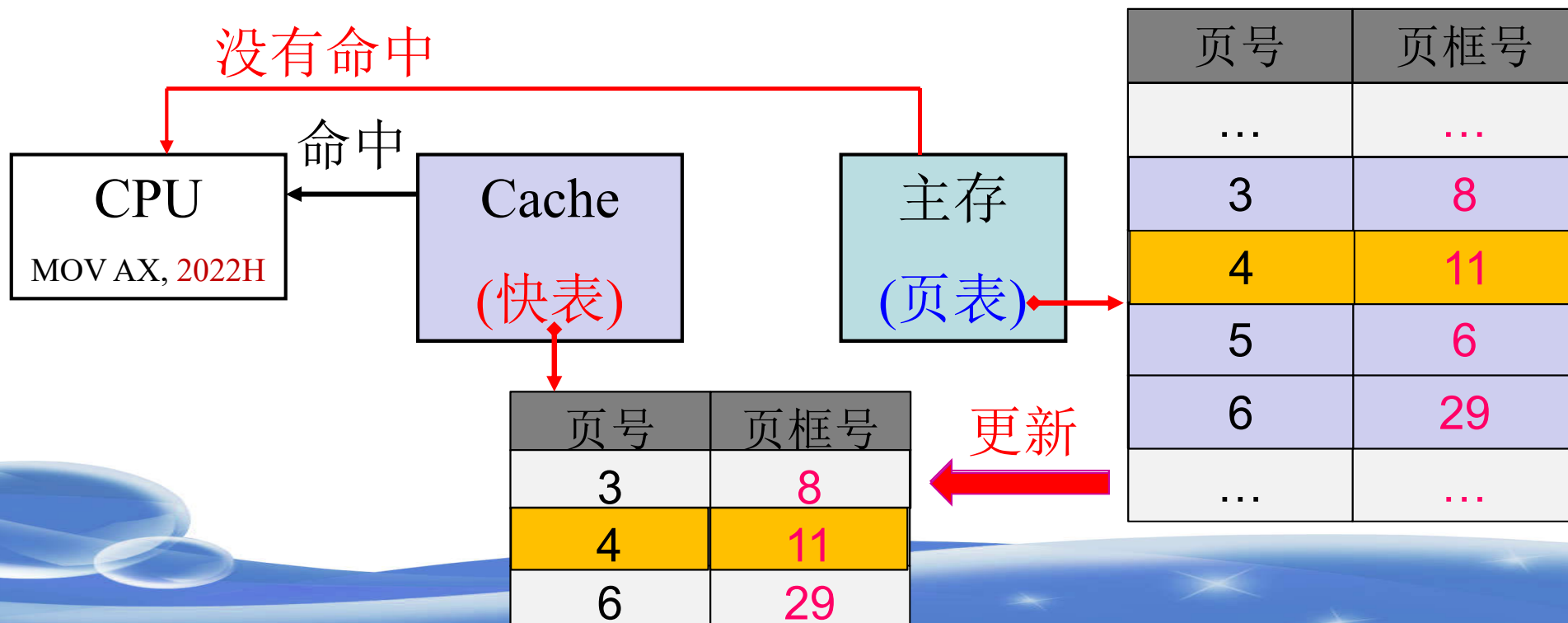
■ CACHE + 内存 + 辅存



快表机制 (Cache)

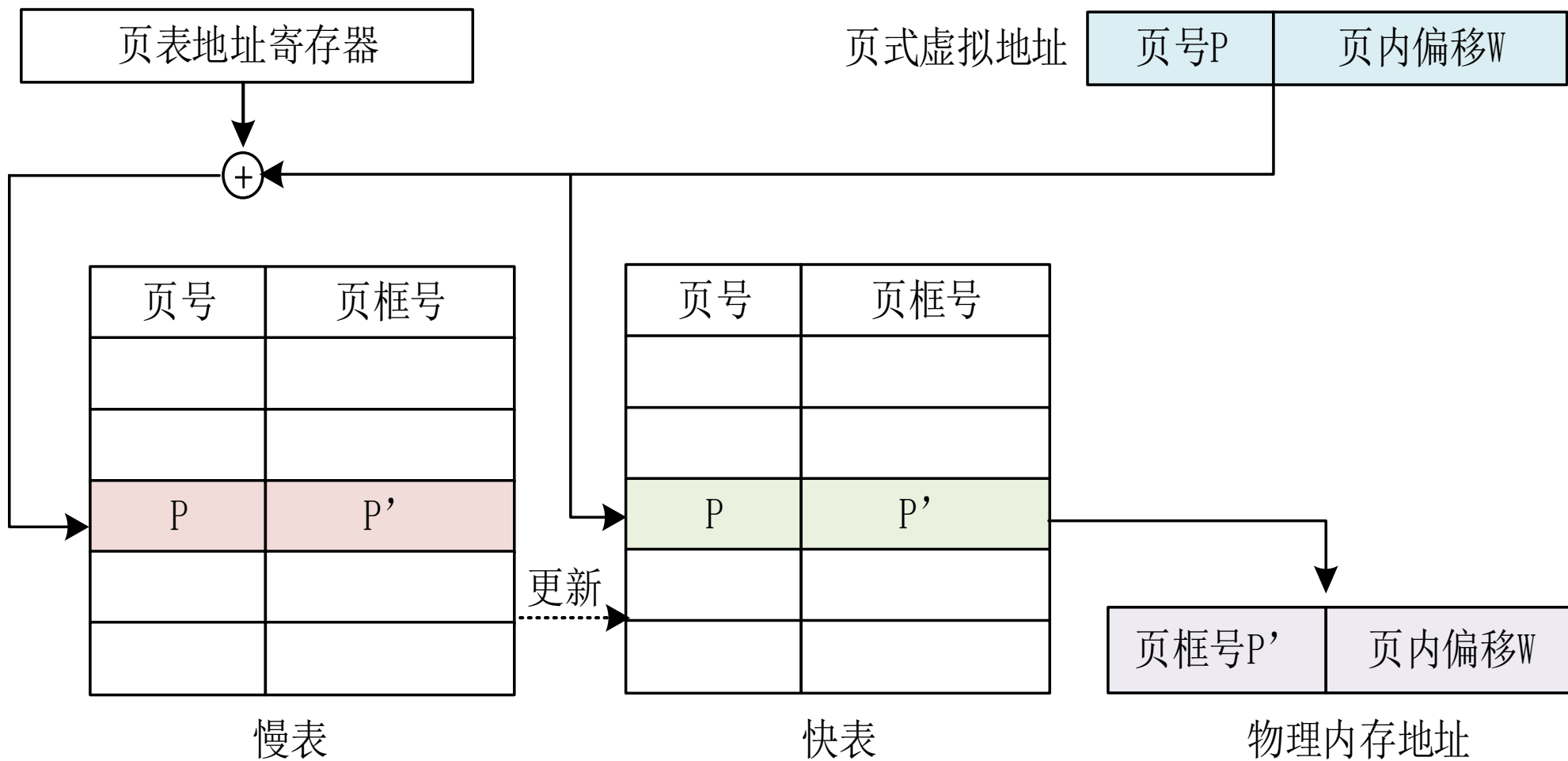
● 快表的特点

- 快表是普通页表（慢表）的**部分内容**的**复制**。
- 地址映射时**优先**访问快表
 - ◆ 若在快表中找到所需数据，则称为“**命中**”
 - ◆ 没有命中时，需要访问慢表，同时**更新快表**
- 合理的页面调度策略能使快表具有**较高命中率**



快表机制 (Cache)

● 快表机制下地址映射过程



例题

- 对于利用快表的分页系统，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期望/平均消耗时间是多少？
 $1.15\mu\text{s}$

- 分析：

- (1) 若能通过快表完成，则只需1次访问内存。

- ◆ $1\mu\text{s}$ 概率 = 85%

- (2) 若不能，则需要2次访问内存。

- ◆ $2\mu\text{s}$ 概率 = 15%

- (3) 平均时间 = $1\mu\text{s} * 85\% + 2\mu\text{s} * 15\%$
 $= \cancel{1.5} \mu\text{s}$
 $= 1.15\mu\text{s}$

页面的共享

● 代码共享的例子——文本编辑器占用多少内存？

■ 文本编辑器： **150KB**代码段和**50KB**数据段

■ 有10文本编辑器进程在并发

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

■ 如果**共享代码段**

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



页面的共享

● 页面共享机制

■ 在页表中填上被共享代码（共享页框）的页框号。

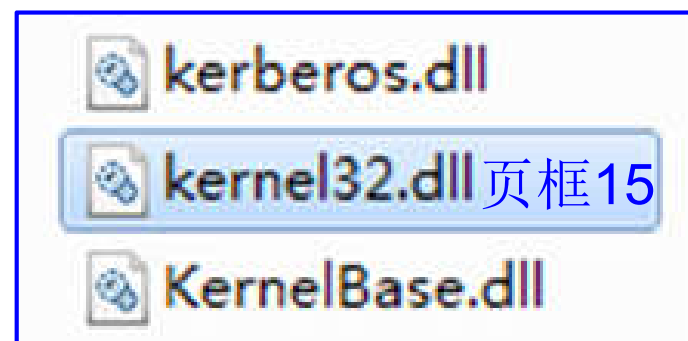
◆ 共享代码/共享页框在内存只有一份存储。

进程A

| 页 | 页框 |
|---|----|
| 0 | 4 |
| 1 | 7 |
| 2 | 15 |
| 3 | 2 |

进程B

| 页 | 页框 |
|---|----|
| 0 | 10 |
| 1 | 12 |
| 2 | 19 |
| 3 | 15 |



二级页表

● 页表实现时的问题

■ 32位OS(4G空间), 每页4K, 每个记录占4字节

- ◆ (1) 每个进程的页数 ?
- ◆ (2) 每个页表的包含的记录条数?
- ◆ (3) 每个页表所占内存是多KB或多少M?
- ◆ (4) 每个页表占几个页框?

| 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 |
| ... | ... | ... | ... | ... | ... |

二级页表

● 页表实现时的问题

■ 32位OS(4G空间), 每页4K, 每个记录占4字节

◆ 进程的页数:

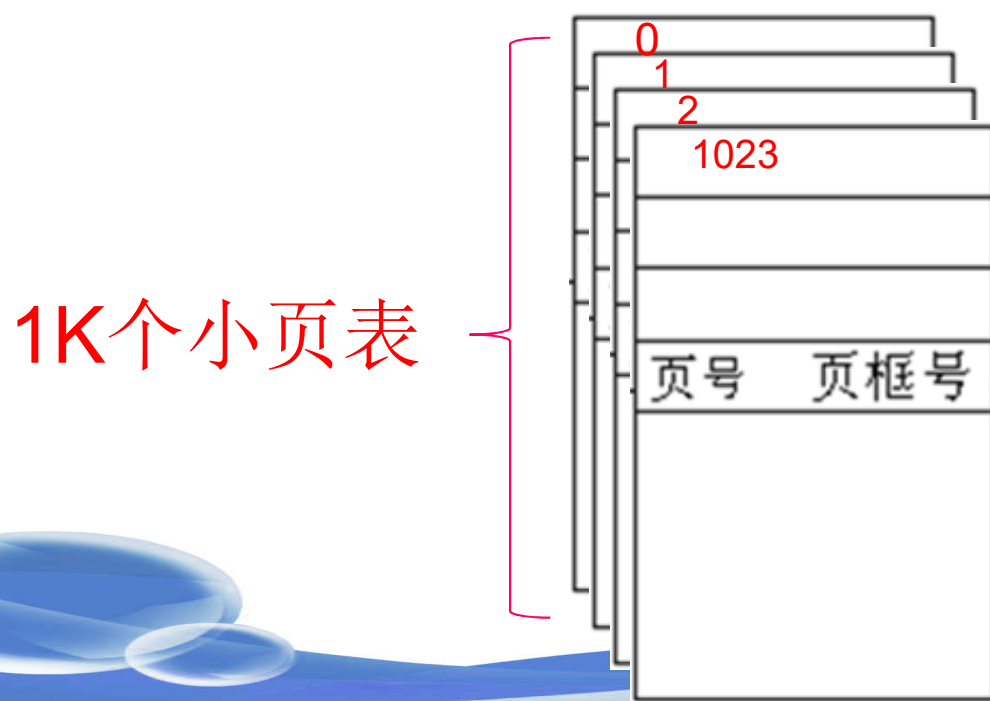
□ 页表的记录条数:

| 页号 | 页框号 | 中断位 | 外存地址 | 访问位 | 修改位 |
|------|----------|-------|------|-----|---------------|
| 0 | 8 | 0 | 4000 | 1 | 0 |
| 问题: | | | | | |
| ◆ 1) | 页表全部装入内存 | 过度消耗 | 8000 | 内存 | (4MB, 1K个页框)。 |
| ◆ 2) | 难以找到 | 连续1K个 | 页框 | 存放 | 页表。 |
| ... | ... | ... | ... | ... | ... |

二级页表

- 策略：把页表(4MB)分拆成1K个小页表(4KB)且分散存放

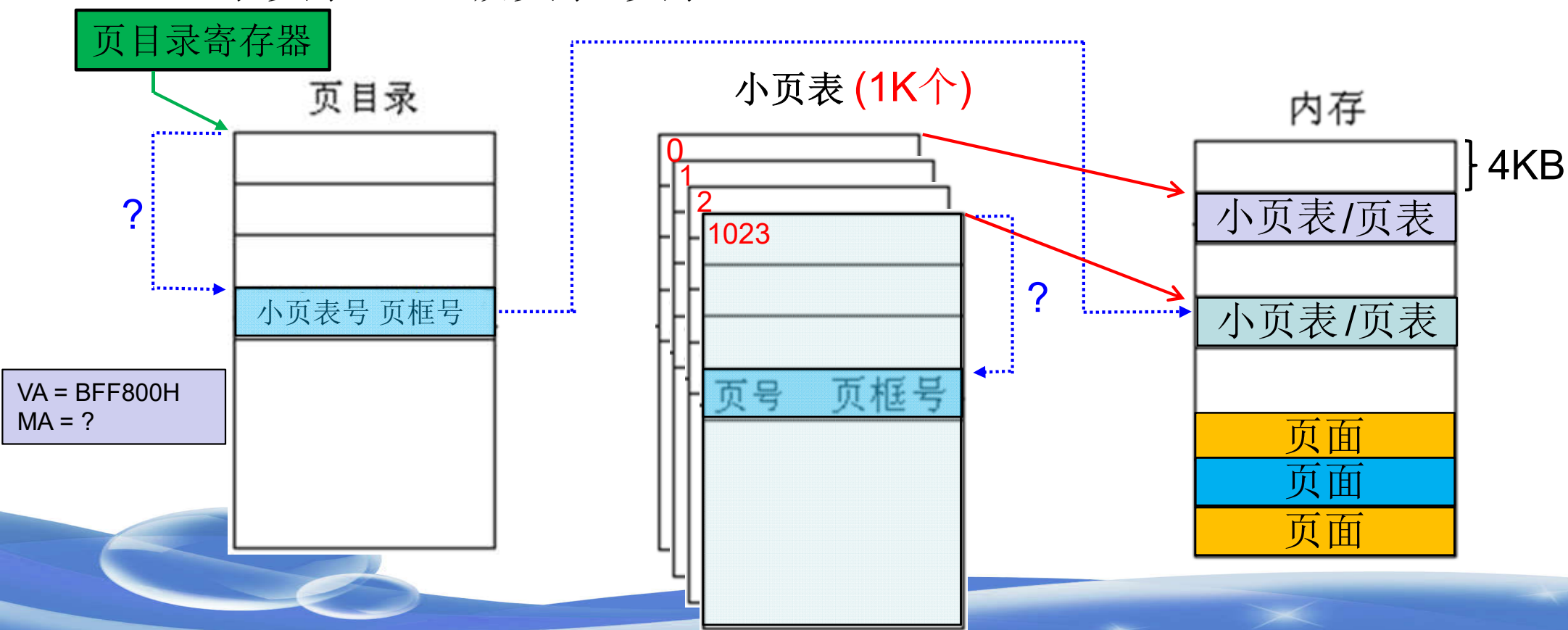
| 1 M 条 记 录 | 页号 | 页框号 | 中断位 | 外存地址 | 访问位 | 修改位 | 1K条 : 1K条 |
|-----------------------|------|------|-----|------|-----|-----|-----------------|
| | 0 | 8 | 0 | 4000 | 1 | 0 | |
| | 1 | 24 | 0 | 8000 | 1 | 1 | |
| | ... | ... | ... | ... | ... | ... | |
| | 1M-1 | 5003 | 1 | A3C4 | 0 | 1 | |



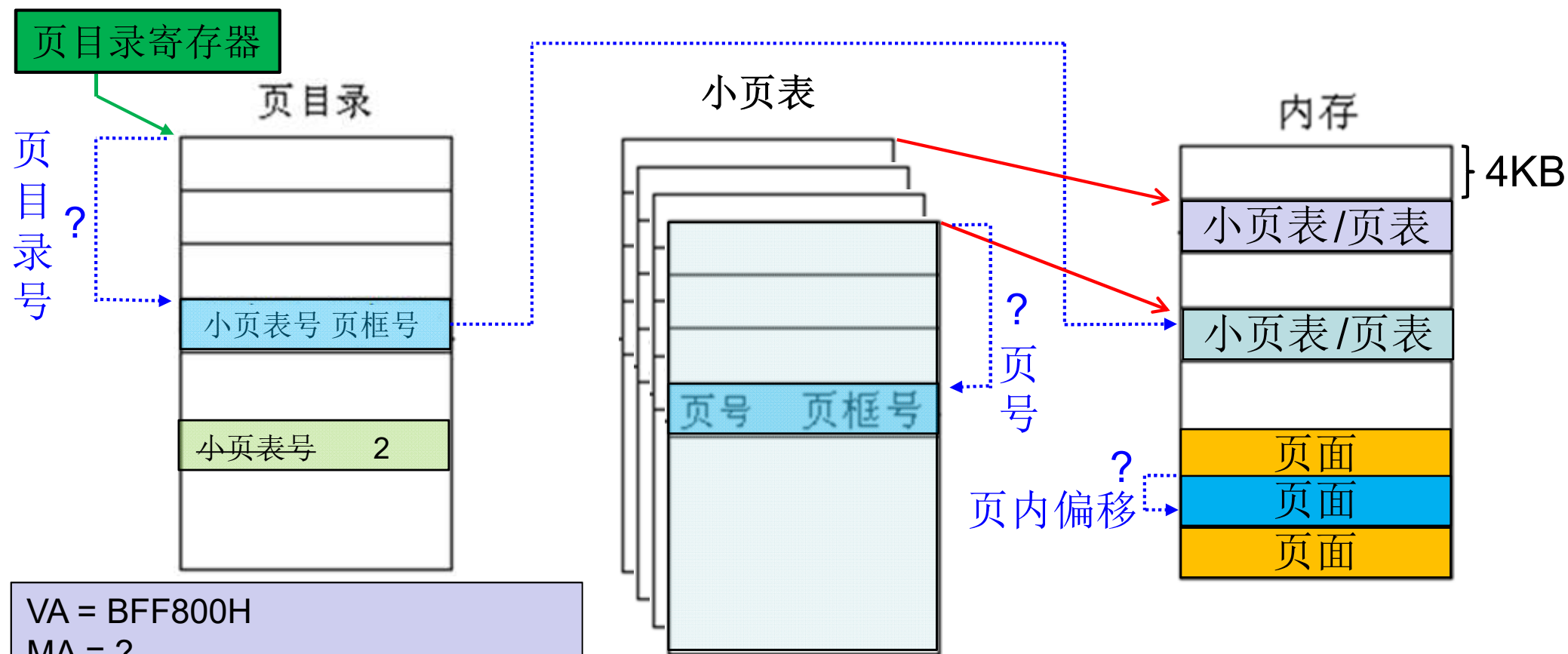
小页表含1K条记录；
小页表的大小是4K；
小页表占一个页框；

二级页表

- 策略：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**）。
 - **页目录**：1K条记录[**小页表号**：**页框号**]，4KB
 - **页目录**：一级页表或外部页表
 - **小页表**：二级页表/页表



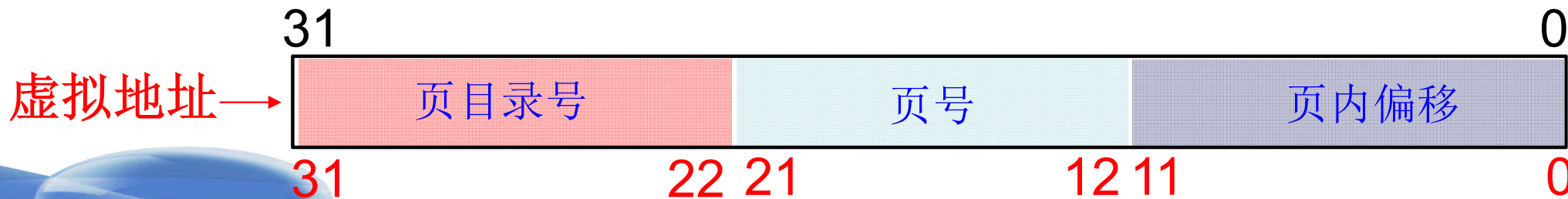
二级页表的地址映射过程



VA = BFF800H

MA = ?

页目录号? 页号=? 页内偏移=?





7.4 /proc文件系统

编程任务1/实验

- 例子：查看应用程序中的变量(虚拟地址)的物理地址

■ 3-3-VA-1.C

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  unsigned int gA = 100;
5
6  int main()
7  {
8      printf("PID=%d, gA = %d, VA(gA)=%x.\n",
9             getpid(), gA, &gA);
10     while(true);
11     return 0;
12 }
```

编程任务2/实验

- 例子：查看应用程序中的变量(虚拟地址)的物理地址

■ 3-3-VA-2.C

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  unsigned int gB = 200;
5
6  int main()
7  {
8      printf("PID=%d, gB = %d, VA(gB)=%x.\n",
9             getpid(), gB, &gB);
10     while(true);
11     return 0;
12 }
```

编程任务1,2/实验

- 例子：查看应用程序中的变量(虚拟地址)的物理地址

```
susg@ThinkPad: /proc/2042
susg@ThinkPad:~/devroot/OS/LAB3$ ./3-3-VA-1.out
PID=2042, gA = 100, VA(gA)= ?
^Z
[1]+  Stopped                  ./3-3-VA-1.out
susg@ThinkPad:~/devroot/OS/LAB3$ ./3-3-VA-2.out
PID=2049, gB = 200, VA(gB)= ?
^Z
[2]+  Stopped                  ./3-3-VA-2.out
susg@ThinkPad:~/devroot/OS/LAB3$ cd /proc
```

Linux的/proc文件系统

- 特点

- 内存文件系统

- 为用户访问内核信息提供接口

- ◆ 例：CPU信息

- ◆ 例：内核信息（如进程信息）

Linux的/proc文件系统

● /proc目录列表

```
susg@ThinkPad: /proc
susg@ThinkPad:/proc$ ls
1      1173  1424  1639  177   188   2082  33    43    795   99    ioports  sched_debug
10     12    1425  1641  178   189   21    330   44    796   993    irq      schedstat
100    1204  1429  1644  1782  19    2111  331   45    799    acpi    kallsyms scsi
101    1205  1477  1671  1787  190   22    332   46    8     asound  kcore    self
1010   1211  1482  1680  179   1905  22    333   47    807   buddyinfo keys      slabinfo
1011   1213  1484  1689  1792  1912  23    334   48    810   bus     key-users softirqs
1019   1234  1491  17    1798  1919  23    335   49    816   cgroups kmsg      stat
102    13    15    1711  18    1926  24    34    5    855   cmdline kpagecgroup swaps
103    1306  1570  172   180   193   249   340   50    856   consoles kpagecount sys
1030   1310  1582  173   181   194   25    341   51    857   cpuinfo kpageflags sysrq-trigger
1036   132   1584  1734  1813  195   26    342   53    858   crypto  loadavg  sysvipc
104    1320  1599  174   182   197   268   35    54    9     devices locks     thread-self
105    1337  16    1742  1825  198   27    354   55    910   diskstats mdstat   timer_list
106    1344  1602  1743  183   1993  277   36    56    912   dma     meminfo  timer_stats
107    1346  1618  175   1835  2    28    37    57    948   driver  misc     tty
1076   1350  1628  1753  1837  20    29    38    6    949   execdomains modules  uptime
11     1362  1630  1754  184   200   3    39    699  960   fb       mounts   version
111    14    1631  1755  185   2007  30    4    7    962   filesystems mtrr     version_signature
1113   1406  1632  176   1857  2024  31    40    783  97    fs       net      vmallocinfo
1131   1411  1634  1765  186   2039  311   41    788  98    interrupts pagetypeinfo vmstat
1152   1420  1638  1769  187   2046  32    42    792  983   iomem    partitions zoneinfo
```

Linux的/proc文件系统

- /proc下重要文件与子目录
 - /proc/数字 关于某个进程（PID=数字）的信息目录。
 - /proc/cpuinfo 处理器信息，如类型、制造商、型号和性能。
 - /proc/devices 当前运行的核心配置的设备驱动列表。
 - /proc/dma 显示当前使用的DMA通道。
 - /proc/filesystems 核心配置的文件系统。
 - /proc/interrupts 显示使用的中断
 - /proc/ioprots 当前使用的I/O端口。
 - /proc/kcore 系统物理内存映象。
 - /proc/kmsg 核心输出的消息。也被送到syslog。

Linux的/proc文件系统

```
susg@ThinkPad: /proc
1152 1406 1630 1754 185 20 27 332 49 816 buddyinfo kallsyms sched_debug vmstat
1173 1411 1631 1755 1857 200 2748 333 50 855 bus kcore schedstat zoneinfo
susg@ThinkPad:/proc$ cat cpuinfo
processor          : 0
vendor_id         : GenuineIntel
cpu family        : 6
model             : 142
model name        : Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
stepping          : 9
microcode         : 0x4e
cpu MHz           : 2638.745
cache size        : 4096 KB
physical id       : 0
siblings          : 4
core id           : 0
cpu cores         : 2
apicid            : 0
initial apicid    : 0
fpu               : yes
fpu_exception     : yes
cpuid level       : 22
wp                : yes
flags              : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr
sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm
pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer xsave avx f16c rdrand lahf_lm abm 3dnowprefetch epb inte
l_pt tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap
clflushopt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp
bugs              :
bogomips          : 5808.00
clflush size      : 64
cache_alignment   : 64
address sizes     : 39 bits physical, 48 bits virtual
power management:

processor          : 1
```

Linux的/proc文件系统

- /proc/**数字** 与指定进程相关的目录

目录名称

目录内容

Cmdline

命令行参数

Environ

环境变量值

Fd

一个包含所有文件描述符的目录

Mem

进程的内存被利用情况

Stat

进程状态

Status

Process status in human readable form

Cwd

当前工作目录的链接

Exe

Link to the executable of this process

Maps

内存印象

Statm

进程内存状态信息

Root

链接此进程的root目录

Linux的/proc文件系统

- /proc/数字 例子

```
susg@ThinkPad: /proc/2910
susg@ThinkPad: /proc$ ps
  PID TTY          TIME CMD
 2898 pts/1        00:00:00 bash
 2910 pts/1        00:02:16 3-3-VA-1.out
 2911 pts/1        00:02:07 3-3-VA-2.out
 2978 pts/1        00:00:00 ps
susg@ThinkPad: /proc$ cd 2910
susg@ThinkPad: /proc/2910$ ls
attr          comm          fd            map_files    net           pagemap      sessionid    status       wchan
autogroup     coredump_filter fdinfo        maps         ns            personality  setgroups    syscall
auxv          cpuset        gid_map       mem          numa_maps    projid_map   smaps        task
cgroup        cwd          io           mountinfo    oom_adj       root         stack        timers
clear_refs    environ       limits        mounts       oom_score     sched        stat         timerslack_ns
cmdline       exe         loginuid     mountstats   oom_score_adj schedstat    statm        uid_map
susg@ThinkPad: /proc/2910$
```

```
susg@ThinkPad: /proc/2910
susg@ThinkPad: /proc/2910$ cat cmdline
./3-3-VA-1.out
susg@ThinkPad: /proc/2910$
```


Linux的/proc文件系统

- /proc/**数字**/**pagemap**

- 记录虚拟页与物理页的映射关系

```
susg@ThinkPad: /proc/2910
susg@ThinkPad: /proc$ ps
  PID TTY          TIME CMD
 2898 pts/1        00:00:00 bash
 2910 pts/1        00:02:16 3-3-VA-1.out
 2911 pts/1        00:02:07 3-3-VA-2.out
 2978 pts/1        00:00:00 ps
susg@ThinkPad: /proc$ cd 2910
susg@ThinkPad: /proc/2910$ ls
attr          comm          fd            map_files    net          pagemap      sessionid     status       wchan
autogroup     coredump_filter fdinfo        maps         ns           personality   setgroups     syscall
auxv          cpuset        gid_map       mem          numa_maps   projid_map    smaps         task
cgroup        cwd           io            mountinfo    oom_adj     root          stack         timers
clear_refs    environ       limits        mounts       oom_score   sched         stat          timerslack_ns
cmdline       exe           loginuid      mountstats   oom_score_adj schedstat     statm         uid_map

susg@ThinkPad: /proc/2910$
```

/proc/pid/pagemap

- 例子：查看应用程序中的变量(虚拟地址)的物理地址

■ 3-3-VA-1.C

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  unsigned int gA = 100;
5
6  int main()
7  {
8      printf("PID=%d, gA = %d, VA(gA)=%x.\n",
9             getpid(), gA, &gA);
10     while(true);
11     return 0;
12 }
```

/proc/pid/pagemap

- 例子：查看应用程序中的变量(虚拟地址)的物理地址

■ 3-3-VA-2.C

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  unsigned int gB = 200;
5
6  int main()
7  {
8      printf("PID=%d, gB = %d, VA(gB)=%x.\n",
9             getpid(), gB, &gB);
10     while(true);
11     return 0;
12 }
```

/proc/pid/pagemap

- 例子：查看应用程序中的变量(虚拟地址)的物理地址

```
susg@ThinkPad: /proc/2042
susg@ThinkPad:~/devroot/OS/LAB3$ ./3-3-VA-1.out
PID=2042, gA = 100, VA(gA)= ?
^Z
[1]+  Stopped                  ./3-3-VA-1.out
susg@ThinkPad:~/devroot/OS/LAB3$ ./3-3-VA-2.out
PID=2049, gB = 200, VA(gB)= ?
^Z
[2]+  Stopped                  ./3-3-VA-2.out
susg@ThinkPad:~/devroot/OS/LAB3$ cd /proc
```

/proc/pid/pagemap

- 记录式文件

- 按序记录虚拟页的信息：64位值

- Bit 0-54: page frame number

- ◆ Bit 0-4: swap type if swapped

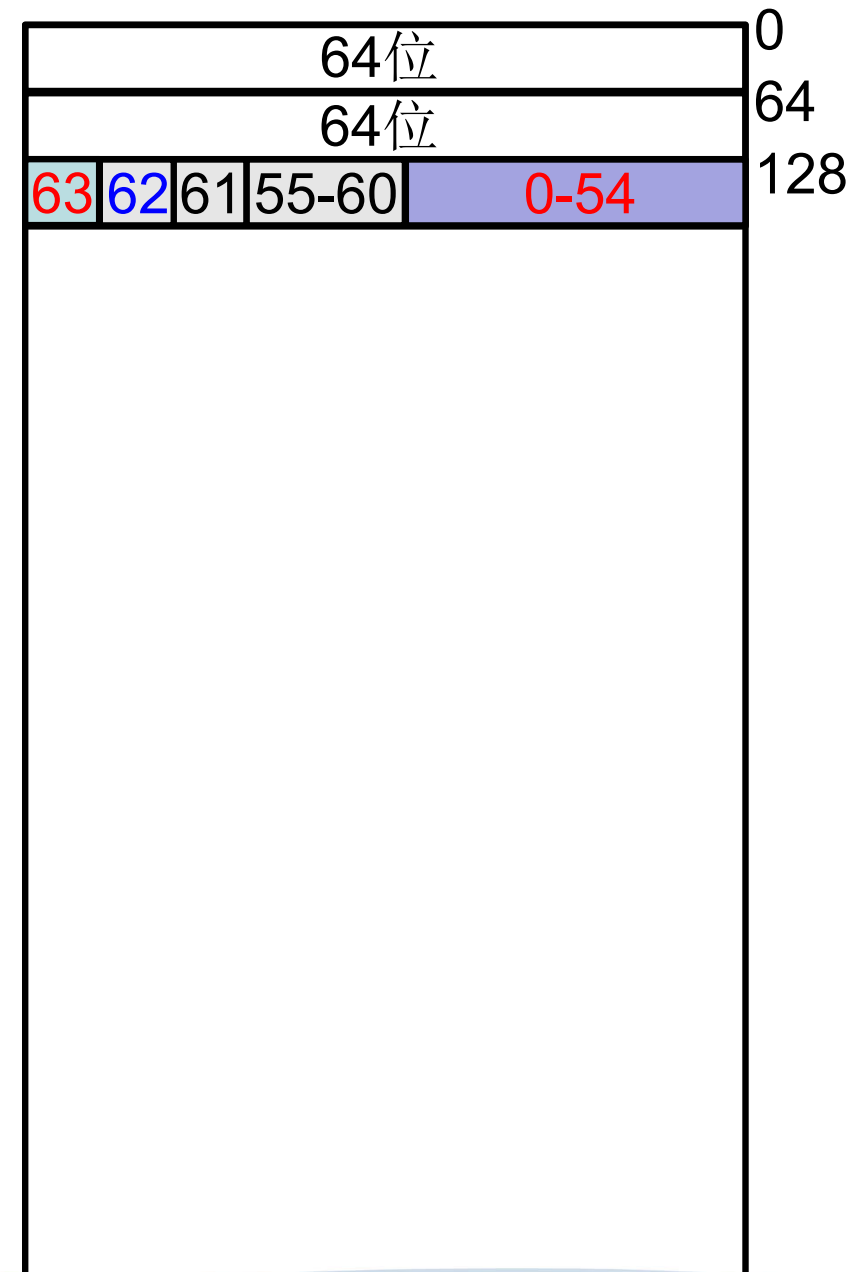
- ◆ Bit 5-54: swap offset if swapped

- Bit 55-60: page shift

- Bit 61: reserved for future use

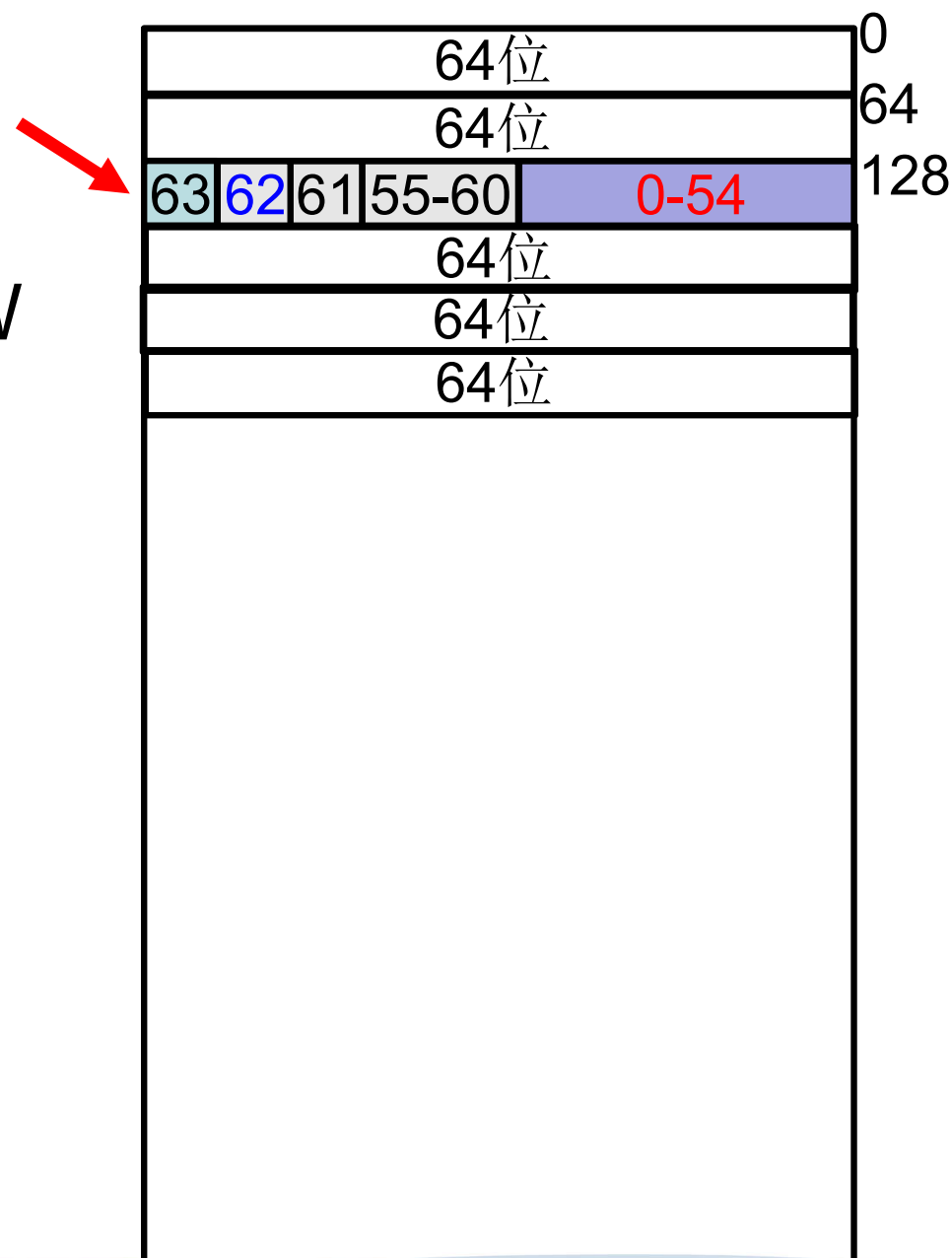
- Bit 62: page swapped

- Bit 63: page present



/proc/pid/pagemap

- 查看进程物理页面映射情况
 - 1) 打开 /proc/pid/pagemap
 - 2) 根据虚拟地址VA算P和W
 - 3) 定位记录 = $P * 8 \text{ Byte}$
 - 4) 分析64位值
 - ◆ Bit 63: Present位
 - ◆ Bit 0-54: 页框号



/proc/pid/pagemap

● 实验任务（验证）

■ 计算进程中指定虚拟地址VA的物理地址MA

◆ 两个进程的同一VA被映射到了不同的MA

```
9 //0-54为PFN(Page Frame Number), PFN掩码
10 #define PFN_MASK (((uint64_t)1)<<55)-1)
14 void main()
15 {
16     //ValueTest 测试变量
17     int ValueTest = 5;
18     //VirAddr 虚拟地址
19     unsigned long VirAddr=(unsigned long) &ValueTest;
20     //PhyAddr 物理页号
21     unsigned long PhyPageNo;
22     //PhyAddr 物理地址
```

/proc/pid/pagemap

```
9 //0-54为PFN(Page Frame Number), PFN掩码
10 #define PFN_MASK (((uint64_t)1)<<55)-1)
11 //63 为Present(page present), Present掩码
12 #define PFN_PRESENT_FLAG (((uint64_t)1)<<63)
13
14 void main()
15 {
16     //ValueTest 测试变量
17     int ValueTest = 5;
18     //VirAddr 虚拟地址
19     unsigned long VirAddr=(unsigned long)&ValueTest;
20     //PhyAddr 物理页号
21     unsigned long PhyPageNo;
22     //PhyAddr 物理地址
23     unsigned long PhyAddr;
24     int FileDesc;
25     //得到页面大小
```

/proc/pid/pagemap

```
31 //根据页号计算其物理内存信息在文件内的存放地址 = N * 64
32 unsigned long PhyMemInfoOffset = VirPageNo * sizeof(uint64_t);
33 //存放读出的物理内存信息（64位）
34 uint64_t PhyMemInfo64Bit;
35 //self 当前进程ID的别名，也可用getpid()获取后再拼接出文件名
36 FileDesc = open("/proc/self/pagemap", O_RDONLY);
37 //移动文件读指针到 PhyMemInfoOffset 位置
38 lseek(FileDesc, PhyMemInfoOffset, SEEK_SET);
39 //读出的物理内存信息（64位）
40 read(FileDesc, &PhyMemInfo64Bit, sizeof(uint64_t));
41 printf("%lx\n", PhyMemInfo64Bit);
42 //利用PFN_PRESENT_FLAG掩码检查最高位是否为0，若为0则无对应页框
43 if(0 == (PhyMemInfo64Bit & PFN_PRESENT_FLAG))
44 {
45     printf("page is not present");
46     return;
47 }
48 //利用PFN_MASK掩码获取低 54 位 作为物理页框号
49 PhyPageNo=(PhyMemInfo64Bit & PFN_MASK);
50 //计算物理地址 = 页框号 x 4K + 偏移
51 PhyAddr = PhyPageNo * PageSize + PageOffset;
```


/proc/pid/pagemap

```
42 //利用PFN_PRESENT_FLAG掩码检查最高位是否为0, 若为0则无对应页框
43 if(0 == (PhyMemInfo64Bit & PFN_PRESENT_FLAG))
44 {
45     printf("page is not present");
46     return;
47 }
48 //利用PFN_MASK掩码获取低 54 位 作为物理页框号
49 PhyPageNo=(PhyMemInfo64Bit & PFN_MASK);
50 //计算物理地址 = 页框号 x 4K + 偏移
51 PhyAddr = PhyPageNo * PageSize + PageOffset;
52 printf("PageSize=%d\n", PageSize);
53 printf("VirPageNo=%lx, PhyPageNo=%lx\n", VirPageNo, PhyPageNo);
54 printf("PageOffset=%lx\n", PageOffset);
55 printf("VirAddr=%p, PhyAddr=0x%lx\n", VirAddr, PhyAddr);
56 }
```

/proc/pid/pagemap

```
35 //self 当前进程ID, 也可用getpid()再拼接出文件名
36 FileDesc = open("/proc/self/pagemap", O_RDONLY);
37 //移动文件读指针到 PhyMemInfoOffset 位置

49 pid = getpid();
50 sprintf(buf, "%d", pid);
51 strcat(pagemap_file, "/proc/");
52 strcat(pagemap_file, buf);
53 strcat(pagemap_file, "/pagemap");
```

/proc/pid/pagemap

- 实验任务（验证）

- 计算进程中指定虚拟地址VA的物理地址MA

- ◆ 两个进程的同一VA被映射到了不同的MA

```
susg@ThinkPad: /proc/2042  
susg@ThinkPad:~/devroot/OS/LAB3$ ./3-3-VA-1.out  
PID=2042, gA = 100, VA(gA)=601040.  
^Z  
[1]+  Stopped                  ./3-3-VA-1.out  
susg@ThinkPad:~/devroot/OS/LAB3$ ./3-3-VA-2.out  
PID=2049, gB = 200, VA(gB)=601040.  
^Z  
[2]+  Stopped                  ./3-3-VA-2.out  
susg@ThinkPad:~/devroot/OS/LAB3$ cd /proc
```


/proc/pid/pagemap

● 实验任务（验证）

■ 不同进程共享的库具有相同的物理地址MA?

◆ 不同进程的不同VA被映射到同一MA(共享)?

■ /proc/pid/maps

7fa961643000

(libc-2.23.so)

```
susg@ThinkPad: /proc/2042
susg@ThinkPad: /proc/2042$ cat maps
00400000-00401000 r-xp 00000000 08:15 1356866 /home/susg/devroot/OS/LAB3/3-3-VA-1.out
00600000-00601000 r--p 00000000 08:15 1356866 /home/susg/devroot/OS/LAB3/3-3-VA-1.out
00601000-00602000 rw-p 00001000 08:15 1356866 /home/susg/devroot/OS/LAB3/3-3-VA-1.out
01b2c000-01b4d000 rw-p 00000000 00:00 0 [heap]
7fa961643000-7fa961803000 r-xp 00000000 08:15 397732 /lib/x86_64-linux-gnu/libc-2.23.so
7fa961803000-7fa961a03000 ---p 001c0000 08:15 397732 /lib/x86_64-linux-gnu/libc-2.23.so
7fa961a03000-7fa961a07000 r--p 001c0000 08:15 397732 /lib/x86_64-linux-gnu/libc-2.23.so
7fa961a07000-7fa961a09000 rw-p 001c4000 08:15 397732 /lib/x86_64-linux-gnu/libc-2.23.so
7fa961a09000-7fa961a0d000 rw-p 00000000 00:00 0
7fa961a0d000-7fa961a33000 r-xp 00000000 08:15 397704 /lib/x86_64-linux-gnu/ld-2.23.so
7fa961c12000-7fa961c15000 rw-p 00000000 00:00 0
7fa961c30000-7fa961c32000 rw-p 00000000 00:00 0
7fa961c32000-7fa961c33000 r--p 00025000 08:15 397704 /lib/x86_64-linux-gnu/ld-2.23.so
7fa961c33000-7fa961c34000 rw-p 00026000 08:15 397704 /lib/x86_64-linux-gnu/ld-2.23.so
7fa961c34000-7fa961c35000 rw-p 00000000 00:00 0
7ffc29a04000-7ffc29a25000 rw-p 00000000 00:00 0 [stack]
7ffc29bb1000-7ffc29bb3000 r--p 00000000 00:00 0 [vvar]
7ffc29bb3000-7ffc29bb5000 r-xp 00000000 00:00 0 [vdso]
ffffffffffff600000-ffffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
susg@ThinkPad: /proc/2042$
```


/proc/pid/pagemap

● 实验任务（验证）

■ 不同进程共享的库具有相同的物理地址MA?

◆ 不同进程的不同VA被映射到同一MA(共享)?

■ /proc/pid/maps

7f1f9f1a5000

7fa961643000

(libc-2.23.so)

(libc-2.23.so)

```
susg@ThinkPad: /proc/2049
susg@ThinkPad:/proc/2049$ cat maps
00400000-00401000 r-xp 00000000 08:15 1356883 /home/susg/devroot/OS/LAB3/3-3-VA-2.out
00600000-00601000 r--p 00000000 08:15 1356883 /home/susg/devroot/OS/LAB3/3-3-VA-2.out
00601000-00602000 rw-p 00001000 08:15 1356883 /home/susg/devroot/OS/LAB3/3-3-VA-2.out
00d61000-00d82000 rw-p 00000000 00:00 0 [heap]
7f1f9f1a5000-7f1f9f365000 r-xp 00000000 08:15 397732 /lib/x86_64-linux-gnu/libc-2.23.so
7f1f9f365000-7f1f9f565000 ---p 001c0000 08:15 397732 /lib/x86_64-linux-gnu/libc-2.23.so
7f1f9f565000-7f1f9f569000 r--p 001c0000 08:15 397732 /lib/x86_64-linux-gnu/libc-2.23.so
7f1f9f569000-7f1f9f56b000 rw-p 001c4000 08:15 397732 /lib/x86_64-linux-gnu/libc-2.23.so
7f1f9f56b000-7f1f9f56f000 rw-p 00000000 00:00 0
7f1f9f56f000-7f1f9f595000 r-xp 00000000 08:15 397704 /lib/x86_64-linux-gnu/ld-2.23.so
7f1f9f774000-7f1f9f777000 rw-p 00000000 00:00 0
7f1f9f792000-7f1f9f794000 rw-p 00000000 00:00 0
7f1f9f794000-7f1f9f795000 r--p 00025000 08:15 397704 /lib/x86_64-linux-gnu/ld-2.23.so
7f1f9f795000-7f1f9f796000 rw-p 00026000 08:15 397704 /lib/x86_64-linux-gnu/ld-2.23.so
7f1f9f796000-7f1f9f797000 rw-p 00000000 00:00 0
7ffd913e1000-7ffd91402000 rw-p 00000000 00:00 0 [stack]
7ffd914bc000-7ffd914be000 r--p 00000000 00:00 0 [vvar]
7ffd914be000-7ffd914c0000 r-xp 00000000 00:00 0 [vdso]
ffffffffffff600000-ffffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
susg@ThinkPad:/proc/2049$
```

/proc/pid/pagemap

● 实验任务（验证）

■ 不同进程共享的库具有相同的物理地址MA?

◆ 不同进程的不同VA被映射到同一MA(共享)?

■ /proc/pid/maps

■ 验证libc-2.23.so装入地址 MA = 9FA68000

◆ 验证VA(7fa961643000, PID=2042) MA = 9FA68000

◆ 验证VA(7f1f9f1a5000, PID=2049) MA = 9FA68000

页式系统的缺点

- 页式系统的不足
 - 页面划分无逻辑含义
 - 页的共享不灵活
 - 页内碎片

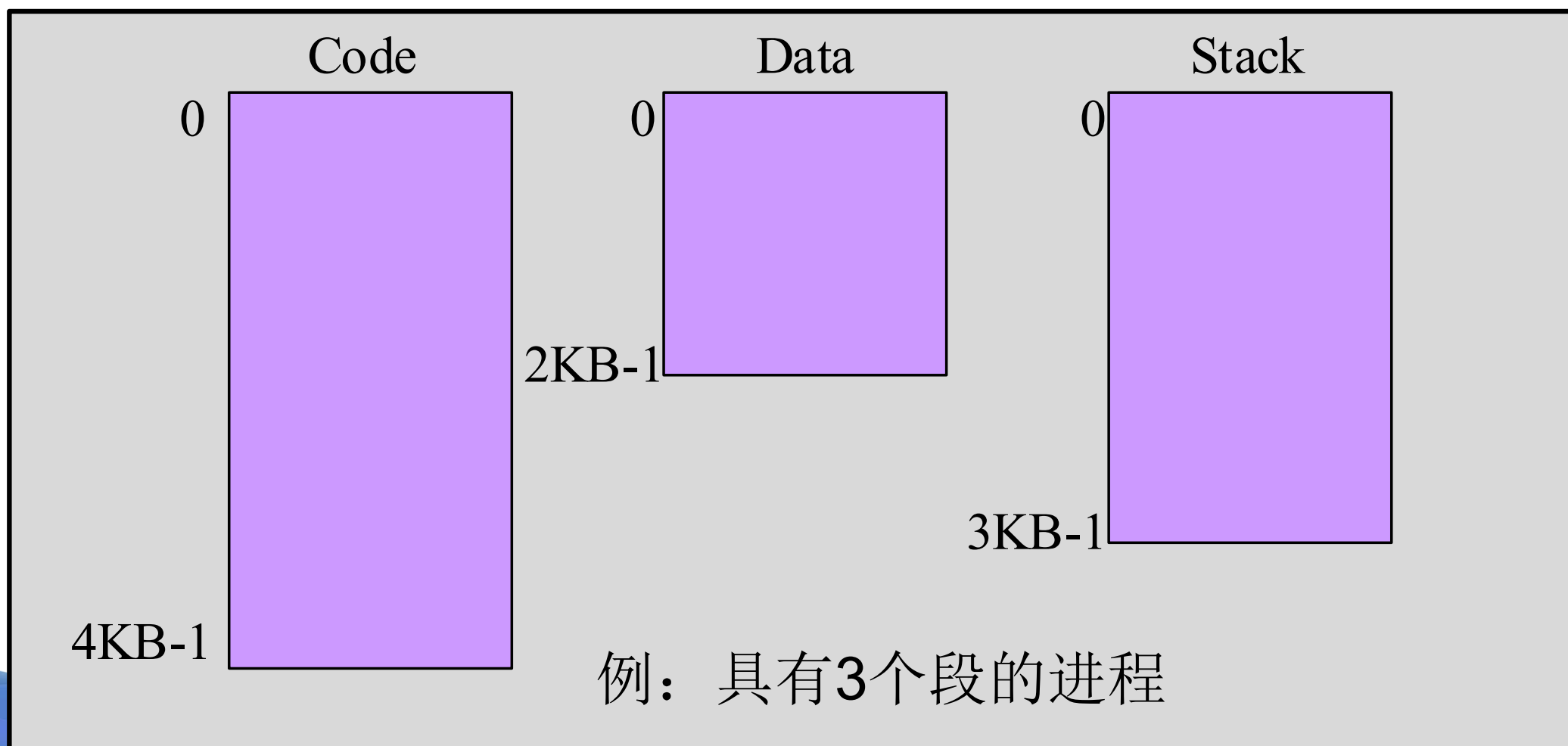


内存

段式存储管理

- 进程分段

- 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



段式存储管理

- 段式内存管理系统的内存分配
 - 以段为单位装入，每段分配连续的内存；
 - 段和段不要求相邻。
- 段表（**SMT**, **Segment Memory Table**)
 - 记录每段在内存中映射的位置
 - 段号**S**：段的编号（唯一的）
 - 段长**L**：该段的长度
 - 基地址**B**：段在内存中的地址

| 段号 | 段长 | 基地址 |
|----|----|-----|
| | | |
| | | |
| S | L | B |
| | | |
| | | |

段式地址的映射机制

- 段式系统的虚拟地址

- 段式虚拟地址VA包含段号S和段内偏移W

- VA: (S, W)

| | |
|-----|-------|
| 段号S | 段内位移W |
|-----|-------|

- 段式地址映射过程

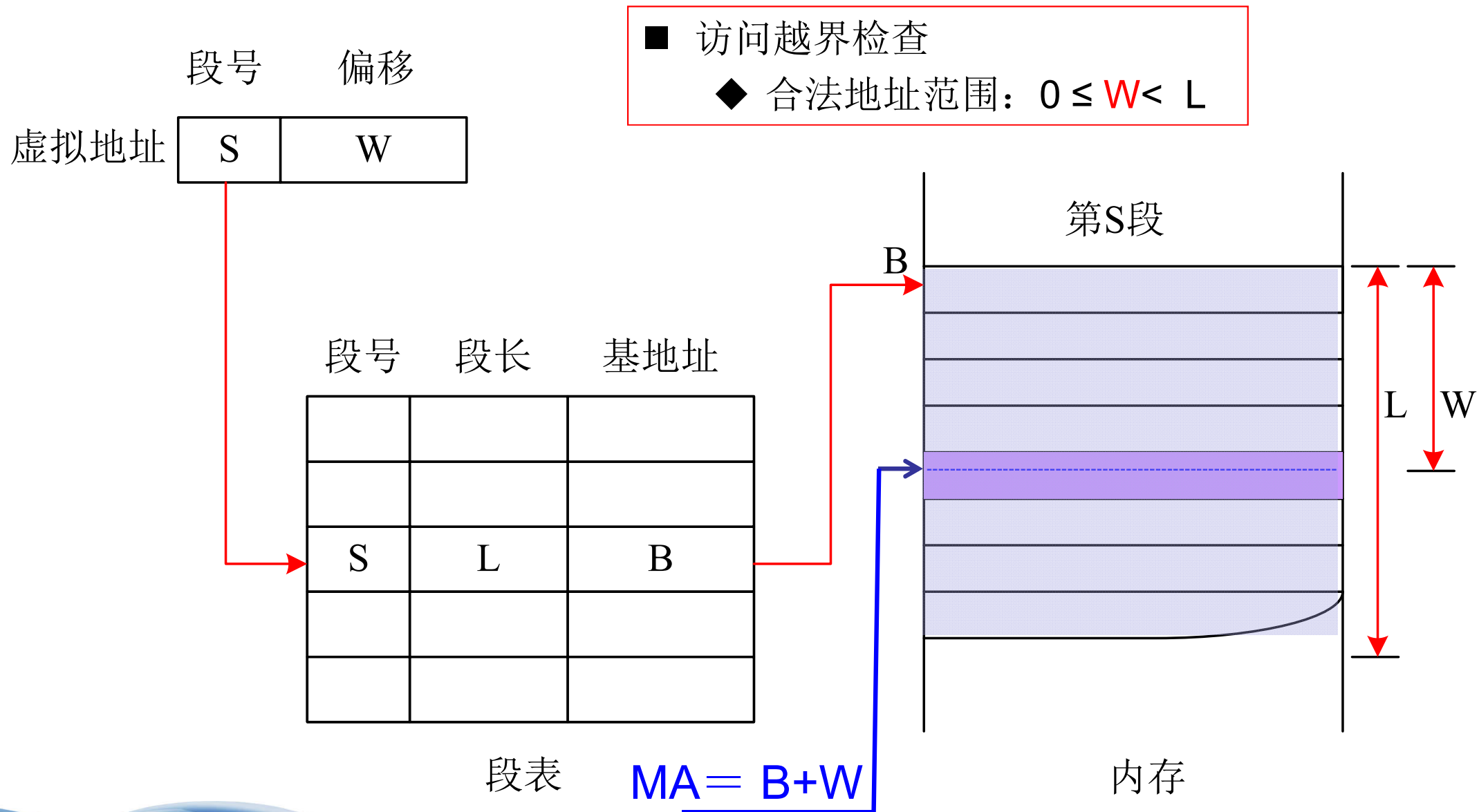
- 1.逻辑地址VA分离出(S, W);

- 2.以S为索引查询段表

- ◆ 检索段号S, 查询该段基地址B和长度L。

- 3.物理地址MA = B+W

段式地址的映射机制



段式地址的映射机制

● 段表的扩充

■ 基本字段：段号，长度，基址

■ 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 |
|----|----|----|-----|-----|-----|---|---|---|
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |

段式地址的映射机制

● 段的共享

- **共享段**在内存中只有一份存储
- 需要共享的模块都可以设置为单独的段
- 共享段写入相关进程的段表

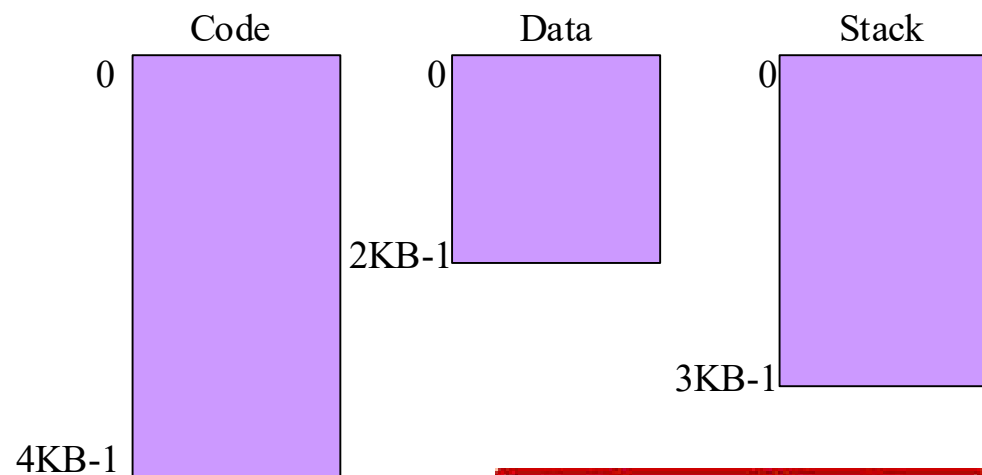
● 段式系统的缺点

- 段需要连续的存储空间
- 段的最大尺寸受到内存大小的限制
- 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

● 地址空间的区别

- 页式系统：一维地址空间
- 段式系统：二维地址空间



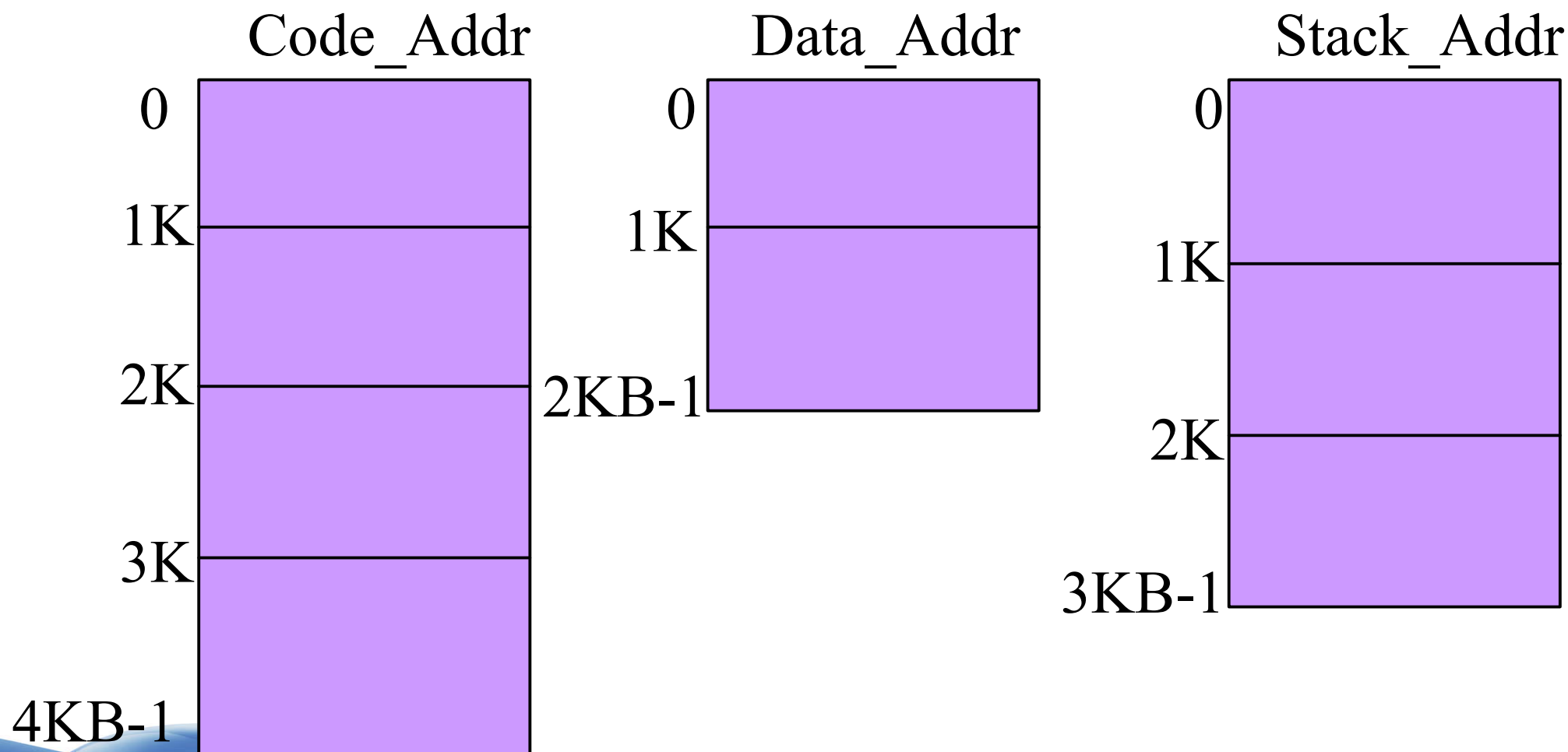
● 段与页的区别

- 段长可变 vs 页面大小固定
- 段的划分有意义 vs 页面无意义
- 段方便共享 vs 页面不方便共享（相对）
- 段用户可见 vs 页面用户不可见
- 段偏移有溢出 vs 页面偏移无溢出

| | | | |
|----|---------|--------|-----------|
| 0 | LF | EQU | 0Ah |
| 1 | CR | EQU | 0Dh |
| 2 | ESC | EQU | 01Bh |
| 3 | COM1 | EQU | 03F8h |
| 4 | COM2 | EQU | 02F8h |
| 5 | COMPORT | EQU | COM1 |
| 6 | TH8E | EQU | 020h |
| 7 | TEXT | EQU | 040h |
| 8 | DR | EQU | 001h |
| 9 | THR | EQU | COMPORT |
| 10 | RBR | EQU | COMPORT |
| 11 | LSR | EQU | COMPORT+5 |
| 12 | MCR | EQU | COMPORT+4 |
| 13 | DLL | EQU | COMPORT+0 |
| 14 | DLM | EQU | COMPORT+1 |
| 15 | LCR | EQU | COMPORT+3 |
| — | CALL | TXCHAR | |

段页式存储管理

- 在段式存储管理中结合页式存储管理技术
- 在段中划分页面



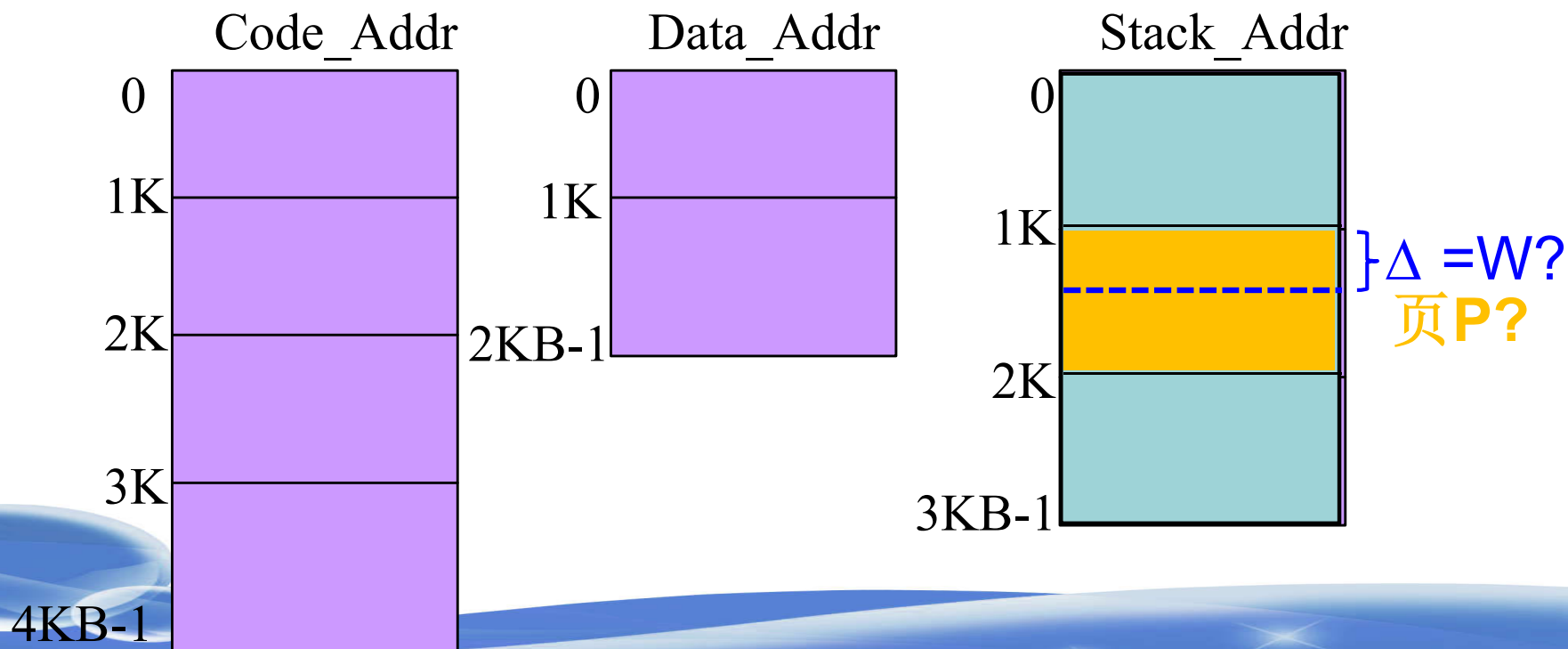
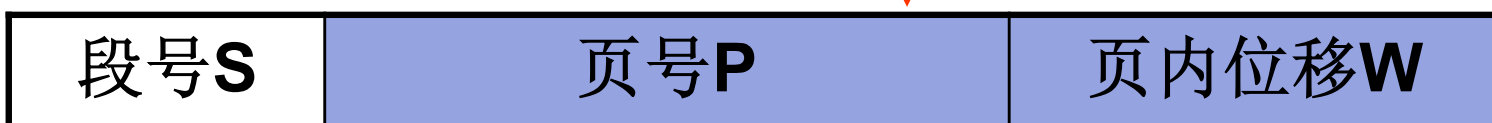
段页式地址映射

- 段页式地址

- 段号S、页号P和页内位移W



↓ 分离出P和W



段页式地址映射

● 段页式地址的映射机构

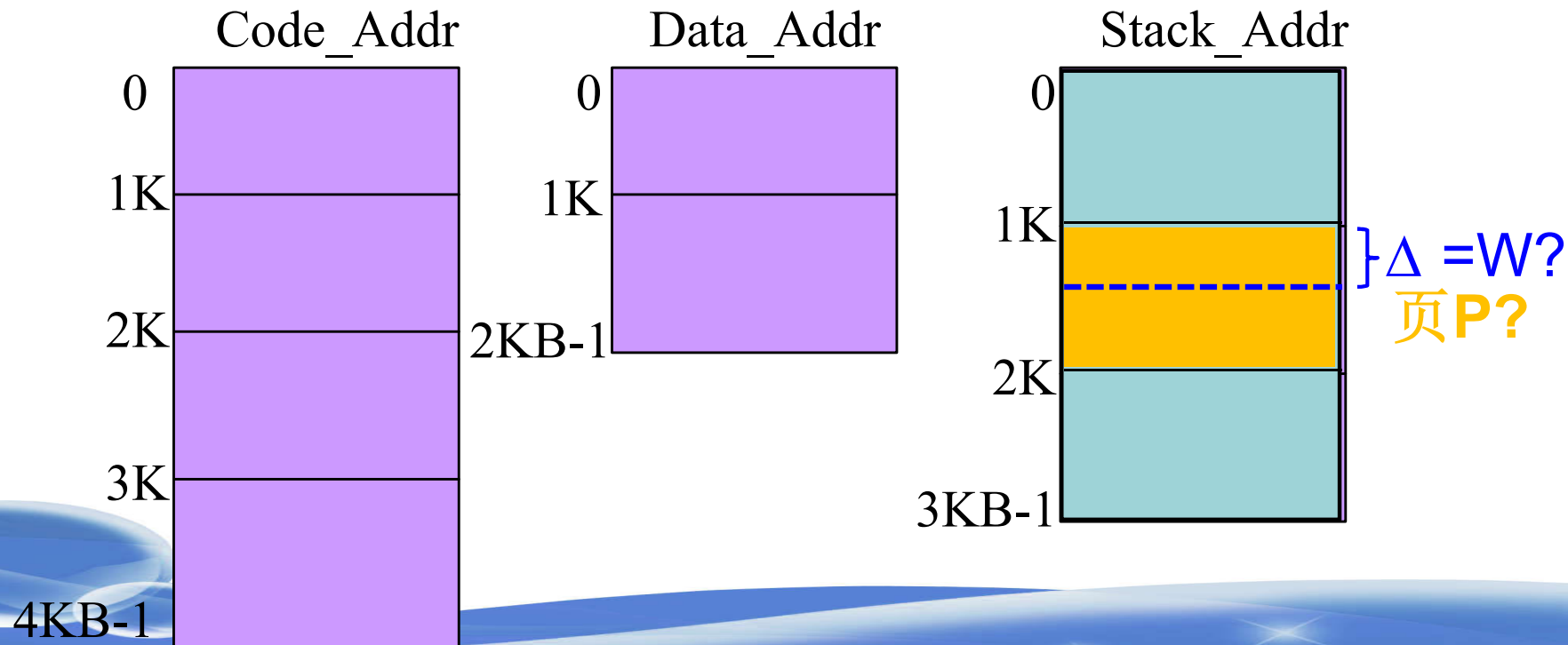
■ 同时采用段表和页表实现地址映射

◆ 系统为每个进程建立一个段表，

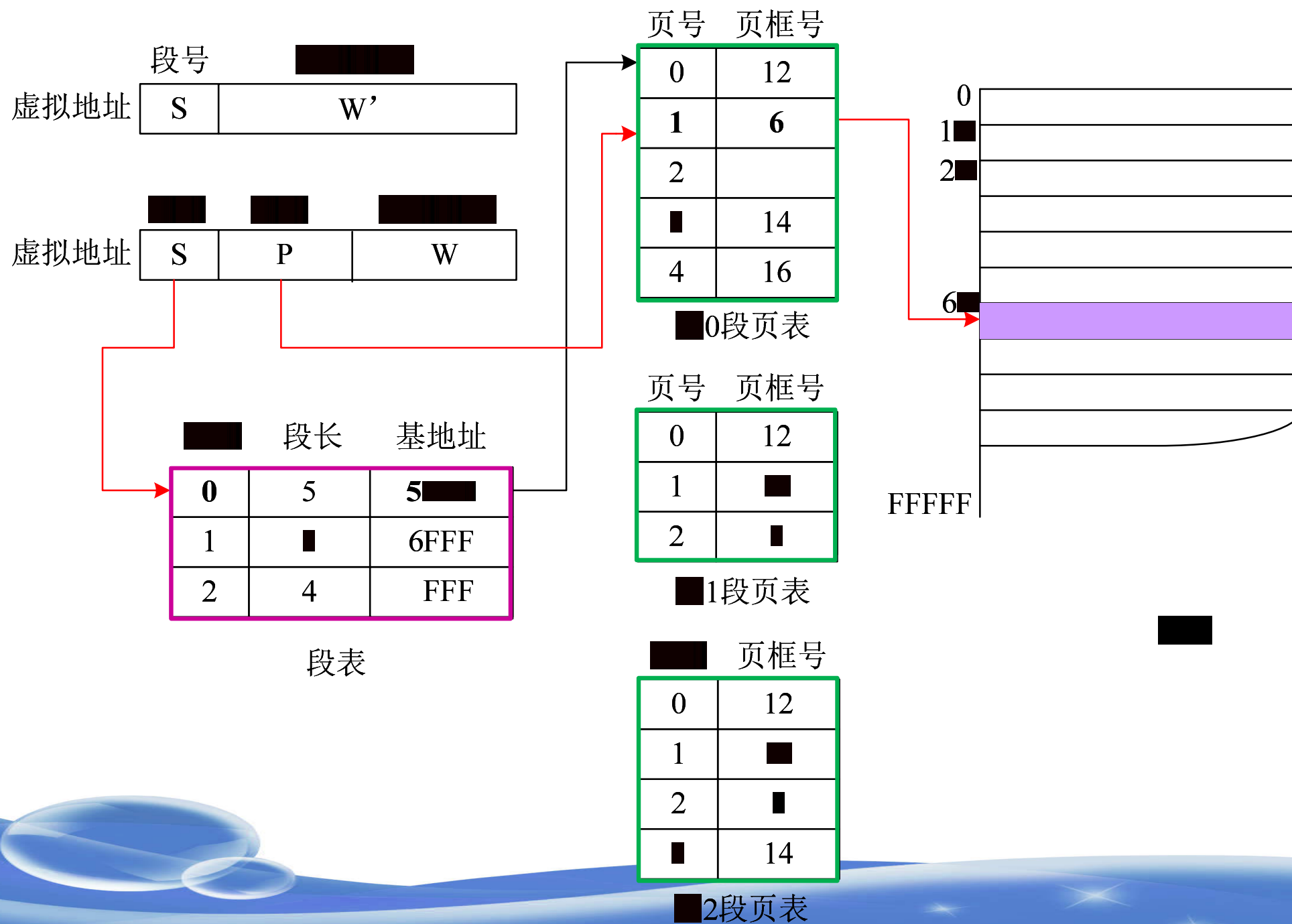
□ 每个段建立一个页表；

◆ 段表给出每段的页表基地址及页表长度

□ 页表给出段内每页对应的页框。



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

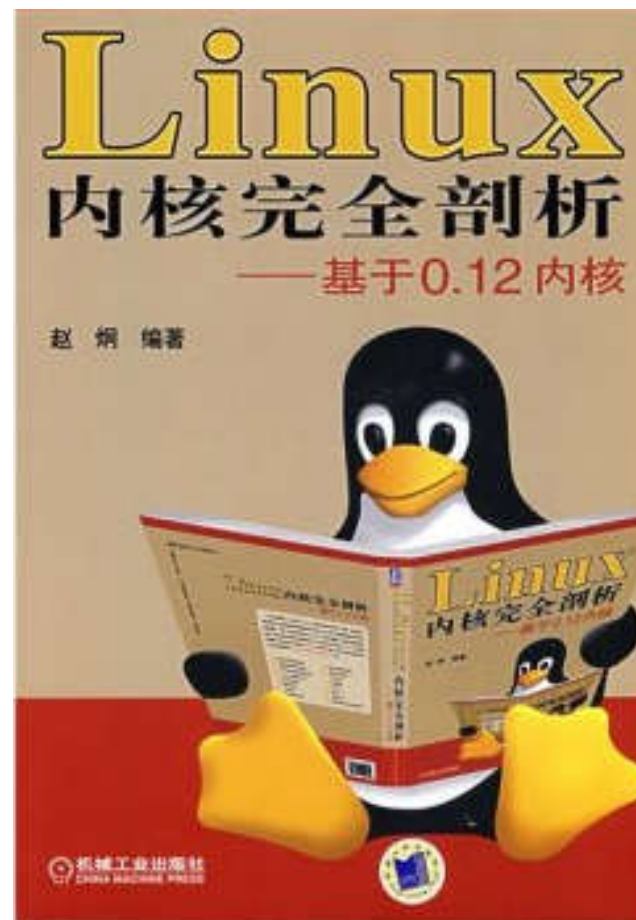
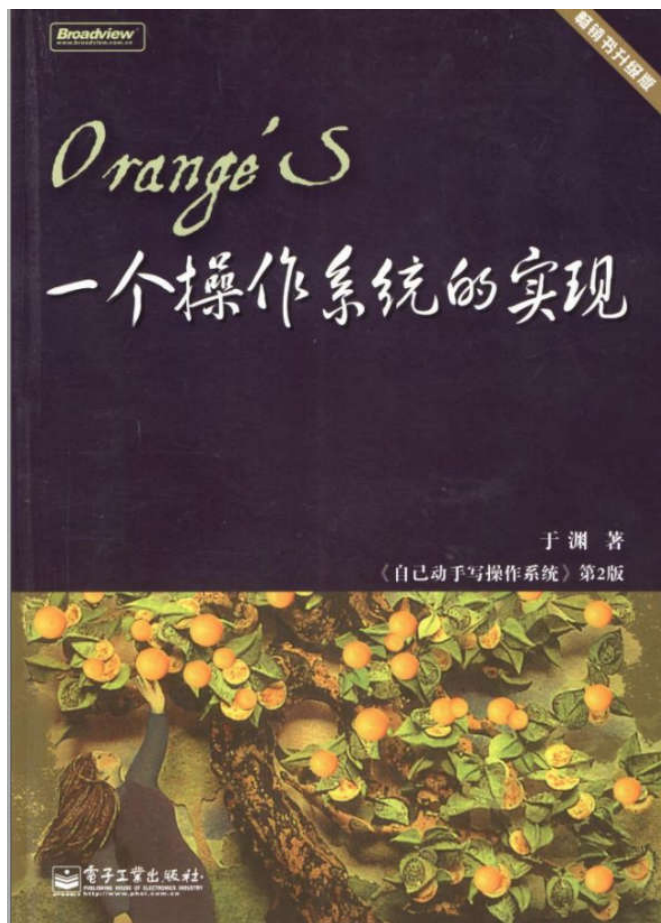




7.5 i386和Linux存储管理

本节在课堂只简单介绍，
详细原理在课程设计环节
解释！ 2023年11月

参考书



实模式 (Real Mode)

- 实模式阶段

- 计算机加电前一段时间处于实模式。

- 实模式内存空间

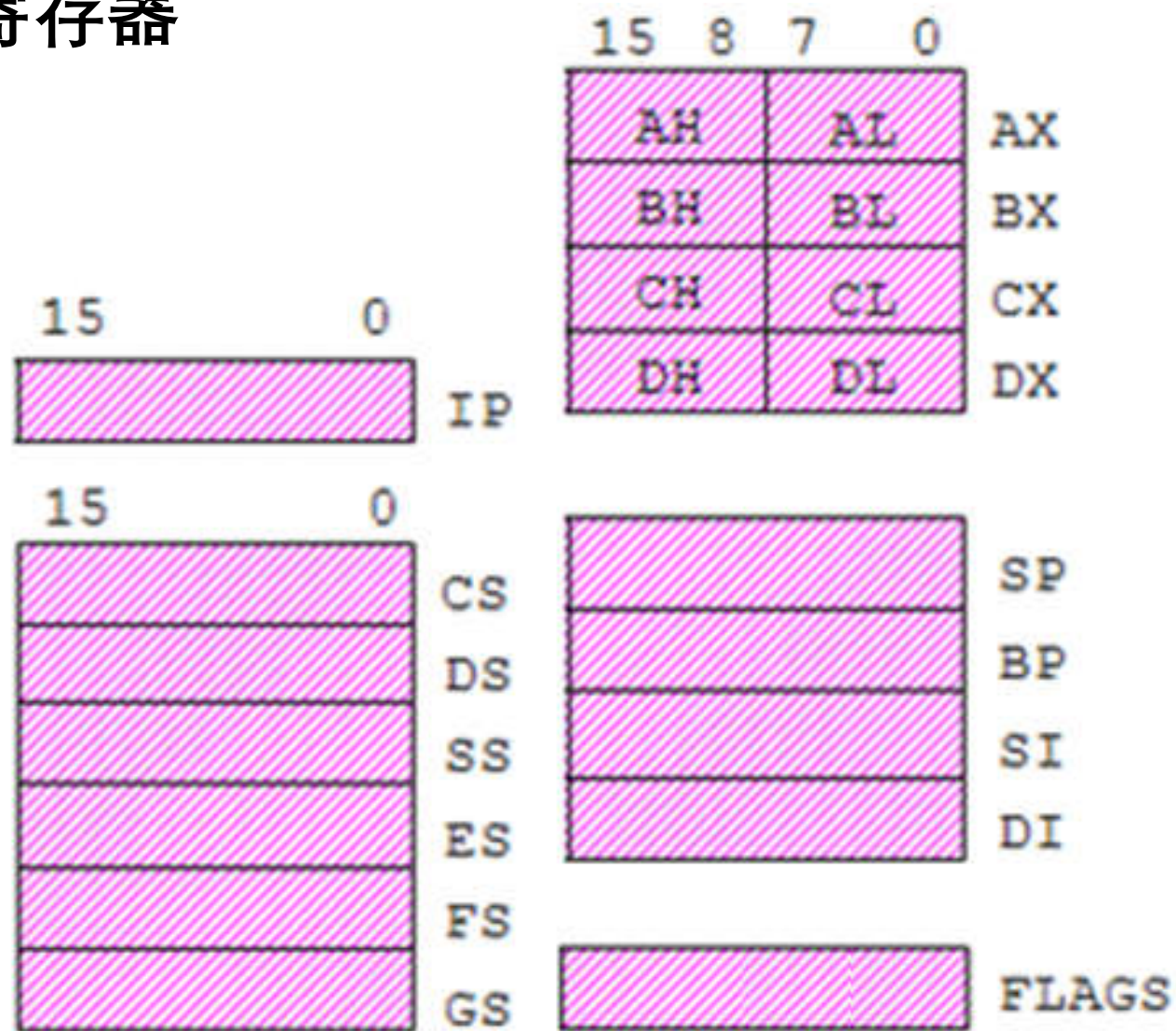
- 20位物理地址

- 1MB内存空间

- 分段机制：段地址(16位)：偏移地址(16位)

实模式 (Real Mode)

- 实模式寄存器



实模式 (Real Mode)

● 实模式寻址

■ 逻辑地址：段地址(16位)：偏移地址(16位)

■ 物理地址 = 段地址左移4位 + 偏移地址

```
MOV BX, 13H
```

```
MOV DS, BX
```

```
MOV AL, [1000H]
```

逻辑地址 = DS:1000H

= 13H:1000H

物理地址 = 1130H

130H

+ 1000H

1130H

保护模式 (Protect Mode)

● 保模式寻址

■ 逻辑地址：段地址(16位)：偏移地址(16位)

■ 物理地址 \neq 段地址左移4位 + 偏移地址

```
MOV BX, 13H  
MOV DS, BX  
MOV AL, [1000H]
```

逻辑地址 = DS:1000H
 = 13H:1000H

物理地址 = 1130H

段基址(13~~0~~H)

+ 1000H

XXXXXXXXXH

保护模式（Protect Mode）

● 保模式寻址

■ 例:逻辑地址 = DS:1000H
= 13H:1000H

■ 段基址=段基址(DS)
=段基址(13H)

■ 描述符、描述符表、类型?

■ DS含义? 段寄存器含义?

基址32 限长16

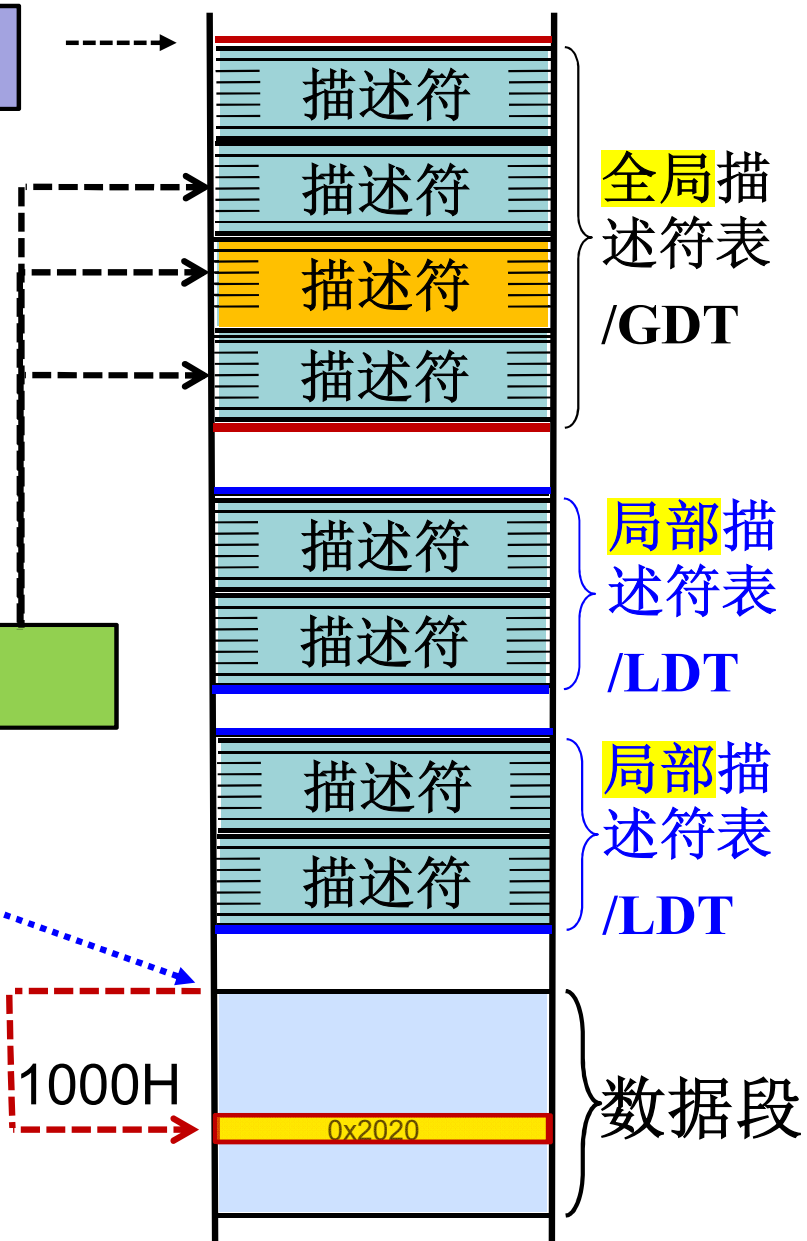
描述符

Base = Base 2 | Base1

| 字节7 | 字节6 | 字节5 | 字节4 | 字节3 | 字节2 | 字节1 | 字节0 |
|---------------------|-----------|-----|--------------------|-----|-----|---------------------|-----|
| Base 2 (31...24) | Attribute | | Base 1 (23...0) | | | Limit 1 (15...0) | |

DS

13H



保护模式（Protect Mode）

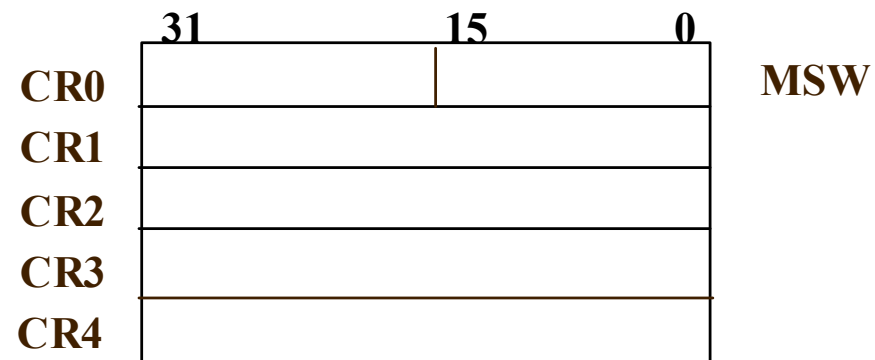
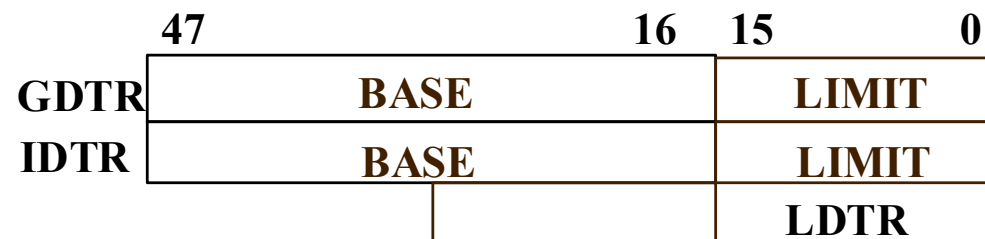
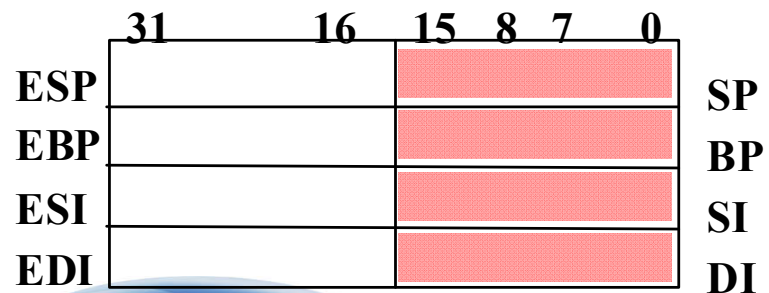
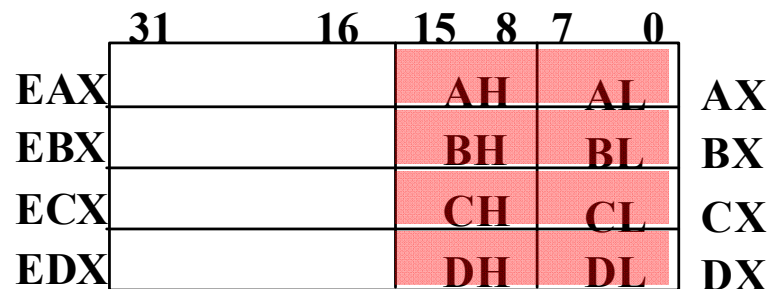
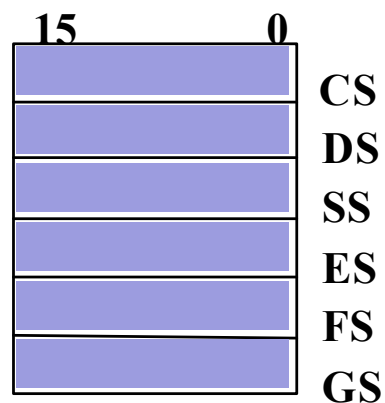
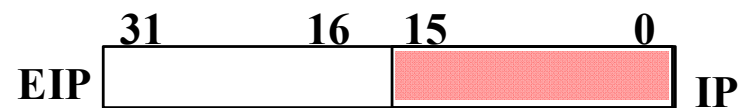
- 保模式寻址

- 优化分段管理机制
- 支持分页管理机制
- 4GB内存空间（32位）

- CPU特性

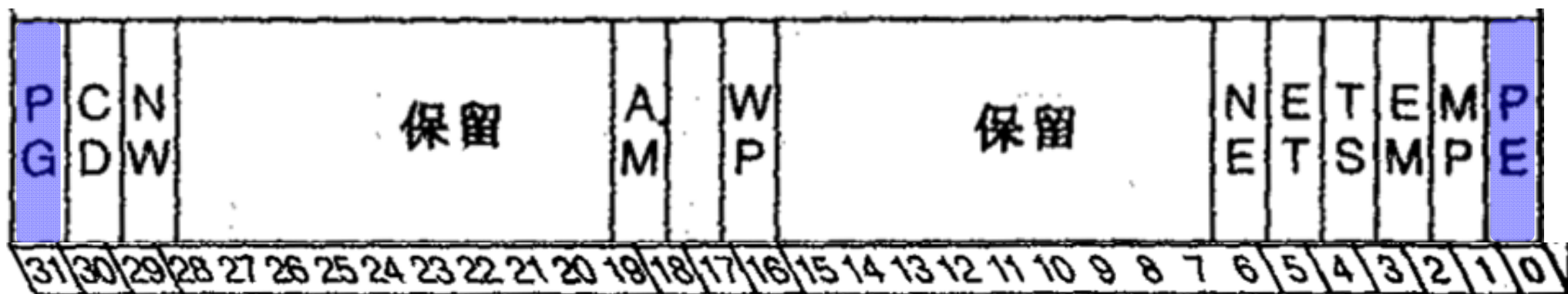
- 支持多任务
- 支持特权级机制
- 扩展寄存器和新增寄存器
 - ◆ EAX~EDX, EIP, ESP, ESI, EDI, EFLAGS
 - ◆ CR0~CR4, GDTR, LDTR, IDTR, TR

保护模式（Protect Mode）



五个控制寄存器CR0-CR4

● 控制寄存器CR0



- PE: 保护模式/实模式
- MP: 有无数学协处理器
- EM: 有无仿真协处理器
- TS: 切换任务时自动设置
- ET: 协处理器的类型
- PG: 是否允许分页

描述符 (Descriptor)

● 属性的描述

■ DPL: 描述符特权级别 **D**escriptor **P**rivilege **L**evel

■ P: Present, 是否在内存中 (**1**: 在内存)

■ G: 段的粒度 (段长计量单位)

◆ G=0, 字节 (段最长1M)

◆ G=1, 页面4KB (段最长4G)

■ S | TYPE: 描述符的类型和特性

◆ S=1 (存储段) | S=0 (系统段)

□ TYPE=4位

▲ 存取属性、特性类型

▲ 读, 写, 访问标志等

数据段

S=1

代码段

操作数

指令码



描述符 (Descriptor)

● 属性的描述

■ DPL: 描述符特权级别 **D**escriptor **P**rivilege **L**evel

■ P: Present, 是否在内存中 (**1**: 在内存)

■ G: 段的粒度 (段长计量单位)

◆ G=0, 字节 (段最长1M)

◆ G=1, 页面4KB (段最长4G)

■ S | TYPE: 描述符的类型和特性

◆ S=1 (存储段) | S=0 (系统段)

□ TYPE=4位

▲ 存取属性、特性类型

▲ 读, 写, 访问标志等

LDT/TSS
系统数据

S=0

门/Gate

代码段

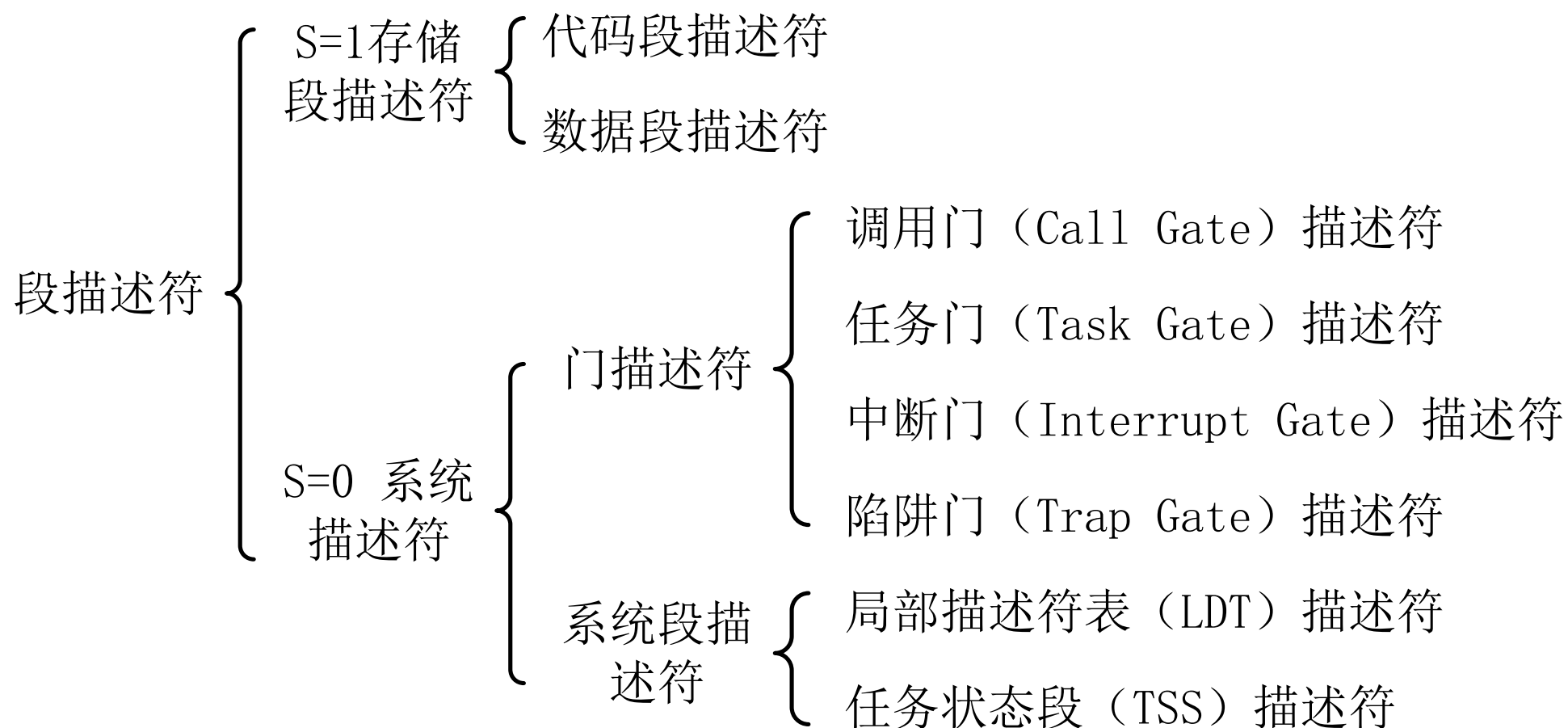
特定数据结构

指令码



描述符（Descriptor）

● 描述符的类型（S）/属性（TYPE）



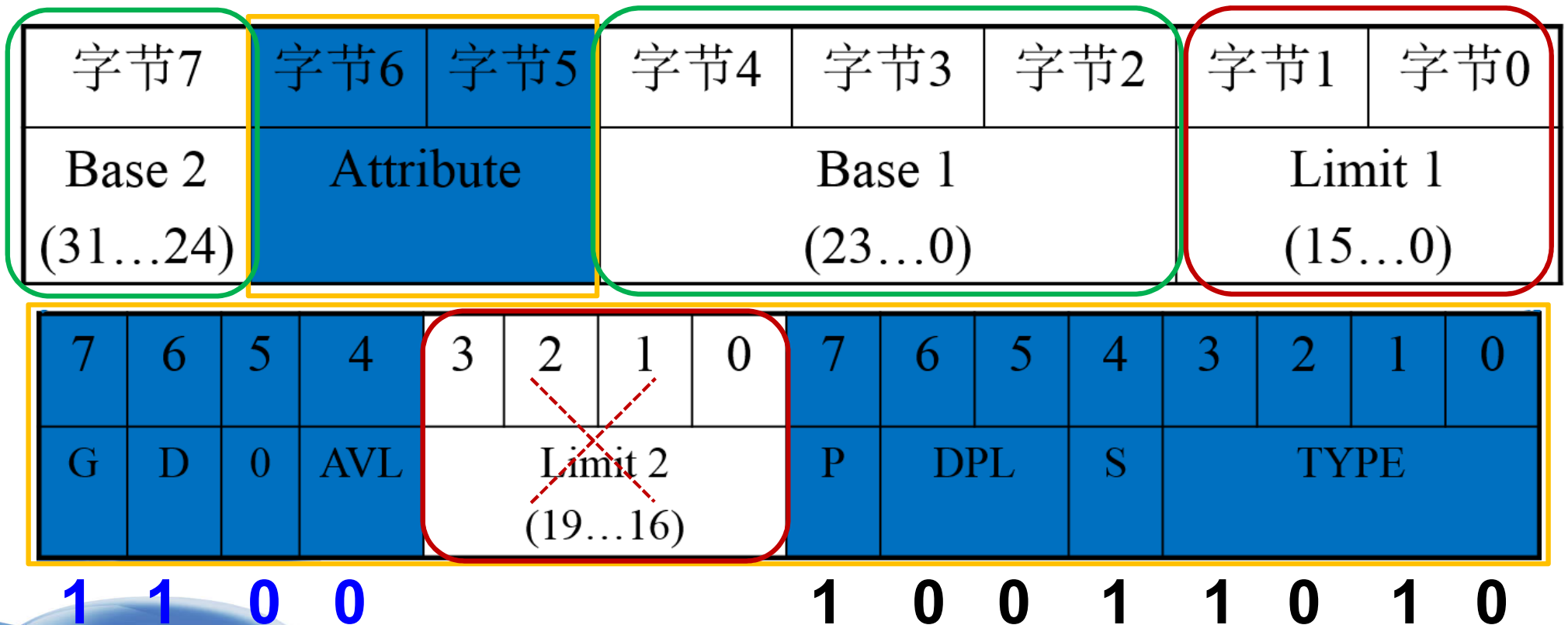
| TYPE 值 | 数据段和代码段描述符 S=1 | 系统段和门描述符 S=0 |
|--------|------------------|--------------|
| 0 | 只读 | < 未定义 > |
| 1 | 只读, 已访问 | 可用 286TSS |
| 2 | 读 / 写 | LDT |
| 3 | 读 / 写, 已访问 | 忙的 286TSS |
| 4 | 只读, 向下扩展 | 286 调用门 |
| 5 | 只读, 向下扩展, 已访问 | 任务门 |
| 6 | 读 / 写, 向下扩展 | 286 中断门 |
| 7 | 读 / 写, 向下扩展, 已访问 | 286 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 |

描述符 (Descriptor)

● 例: .quad 0x 00C0 9A80 0000 001F

■ 7~4: 0000 0000 1100 0000 1001 1010 1000 0000

■ 3~0: 0000 0000 0000 0000 0000 0000 0001 1111



描述符 (Descriptor)

● 例: .quad 0x 00C0 9A80 0000 001F

■ 7~4: 0000 0000 1100 0000 1001 1010 1000 0000

■ 3~0: 0000 0000 0000 0000 0000 0000 0001 1111

■ G = 1 (Page = 4KB)

■ BASE = 0x800000 = 8192K

■ Limit = (0x1F + 1) * 4K = 128K

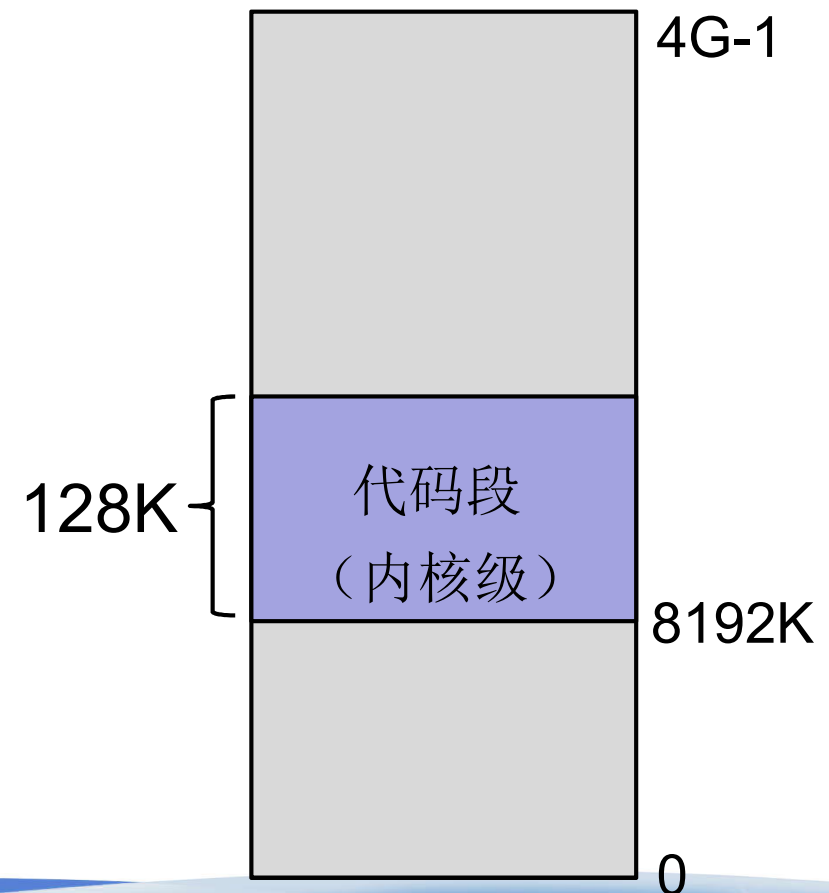
■ S = 1

■ TYPE = 1010

} 代码段

■ P = 1 (在内存中)

■ DPL = 00 (内核级)



描述符（Descriptor）C版本（S=1）

```
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
```

```
}
```

| 字节7 | 字节6 | 字节5 | 字节4 | 字节3 | 字节2 | 字节1 | 字节0 |
|---------------------|-----------|-----|--------------------|-----|-----|---------------------|-----|
| Base 2 (31...24) | Attribute | | Base 1 (23...0) | | | Limit 1 (15...0) | |

描述符（Descriptor）汇编版本（S=1）

%macro Descriptor 3 ;8字节

dw **%2** & 0FFFFh ;段界限1

dw **%1** & 0FFFFh ;段基址1

db (**%1** >> 16) & 0FFh ;段基址2 ;属性1+段界限2+属性2

dw ((**%2** >> 8) & 0F00h) | (**%3** & 0F0FFh))

db (**%1** >> 24) & 0FFh ;段基址3

%endmacro

● 用法: **Descriptor** **Base** , **Limit** , **Attribution**

例: **Descriptor** 0x800000, 0x1F, 0x9A80

| 字节7 | 字节6 | 字节5 | 字节4 | 字节3 | 字节2 | 字节1 | 字节0 |
|---------------------|-----------|-----|--------------------|-----|-----|---------------------|-----|
| Base 2 (31...24) | Attribute | | Base 1 (23...0) | | | Limit 1 (15...0) | |

描述符表 (Descriptor Table)

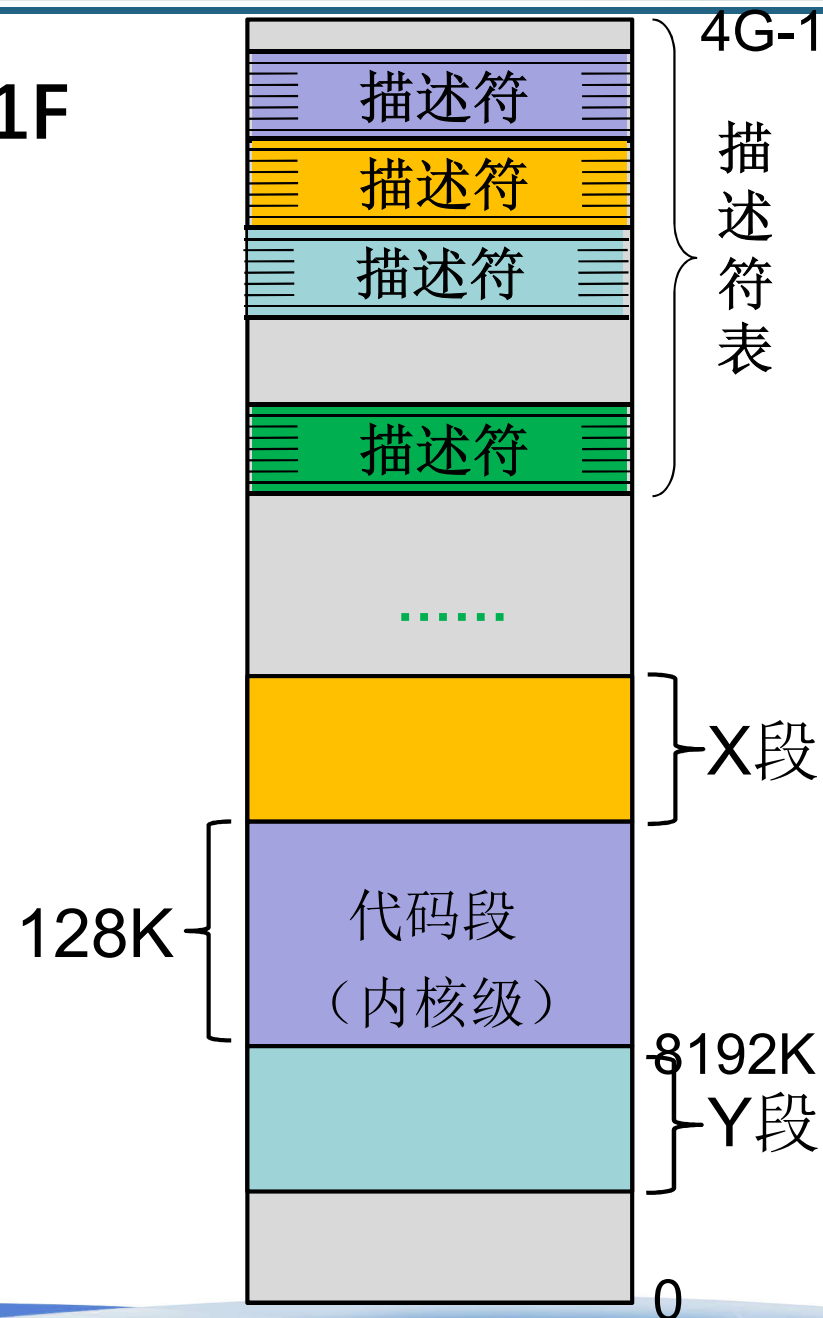
- 例: .quad 0x 00C0 9A80 0000 001F

- 描述符表

- 存放描述符的数组/线性表
- 长度: 8字节的整数倍。

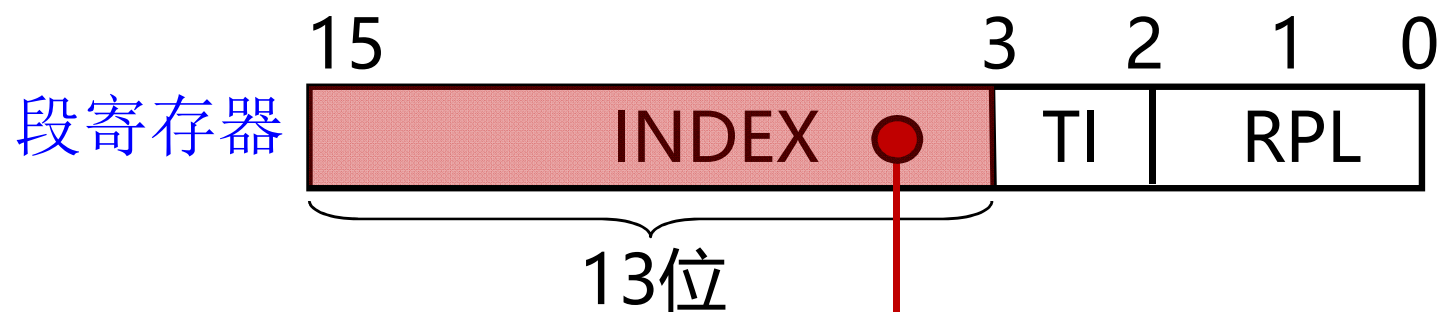
- 描述符表类型

- 全局描述符表GDT
 - ◆ Global Descriptor Table
- 局部描述符表LDT
 - ◆ Local Descriptor Table
- 中断描述符表IDT
 - ◆ Interrupt Descriptor Table



选择子(Selector) /16位/段寄存器

- 用于选择**GDT/LDT**等表中的描述符



- 索引域 (INDEX)

- ◆ 13位，描述符在描述符表中的**序号**

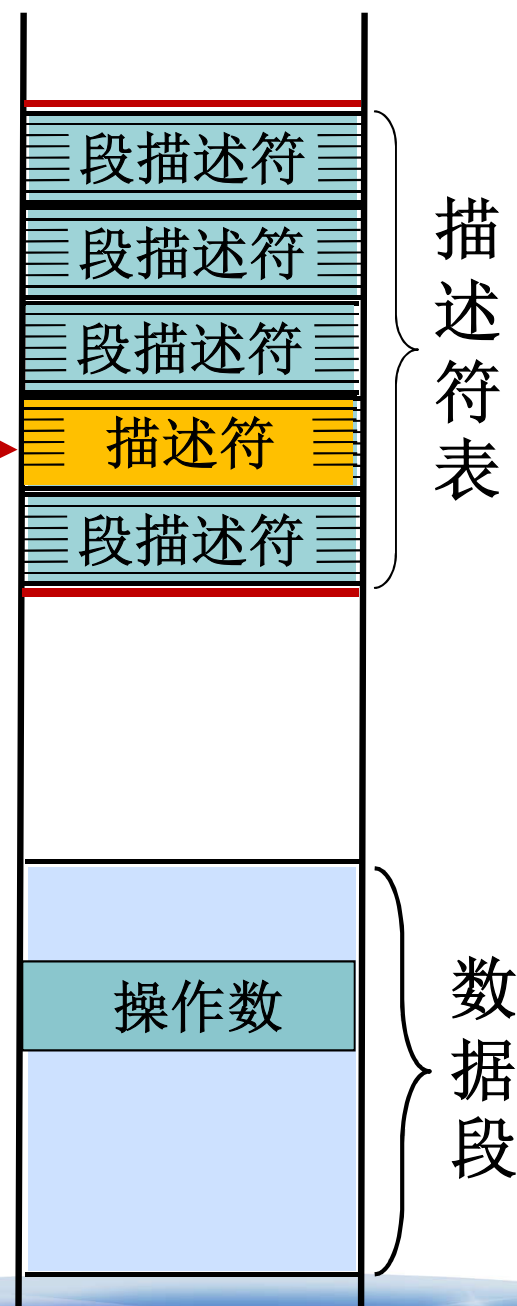
- 表标识域 (TI, Table Indicator)

- ◆ 1位：**GDT (0) | LDT (1)**

- 请求特权级域 (RPL)

- ◆ 2位，Request Privilege Level

- 选择子放在**段寄存器**中



选择子(Selector) (回顾: 保护模式内存寻址)

● 保模式寻址

■ 例: 逻辑地址 = DS:1000H
= 13H:1000H

■ 段基址 = 段基址(DS)
= 段基址(13H)

■ 描述符、描述表、类型?

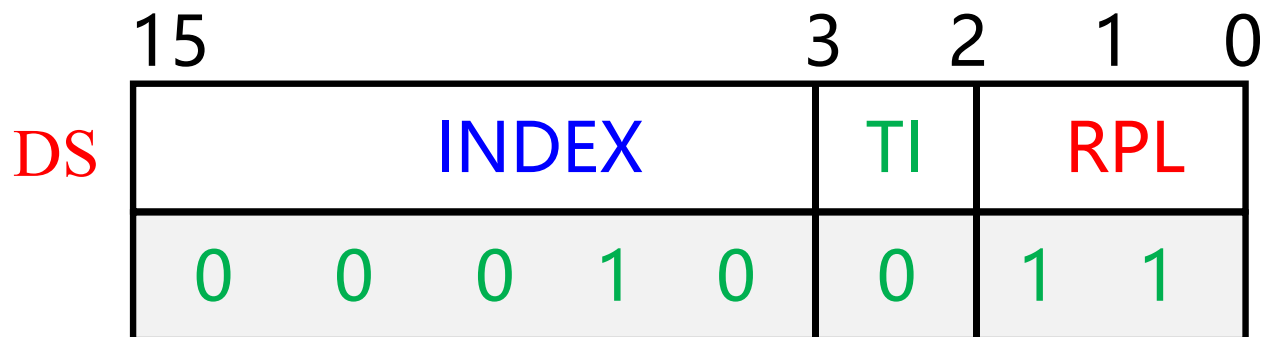
■ DS含义? 段寄存器含义?



INDEX = 2

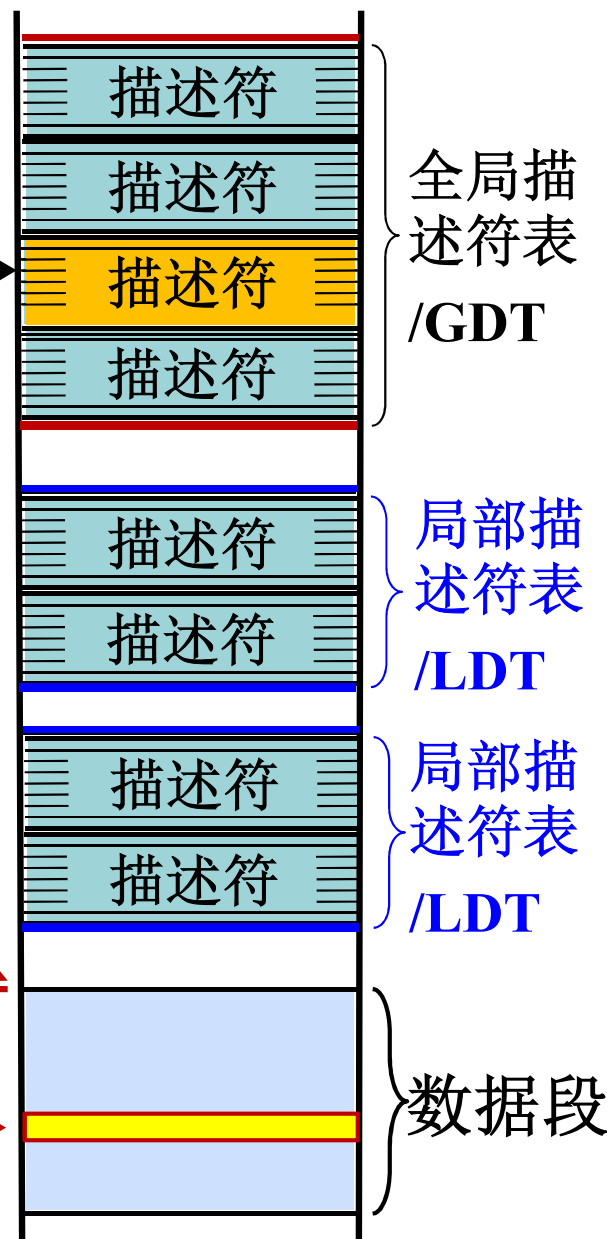
DS

13H



Base

1000H



选择子(Selector) (回顾: 保护模式内存寻址)

● 保模式寻址

■ 例: 逻辑地址 = DS:1000H
= 13H:1000H

■ 段基址 = 段基址(DS)
= 段基址(13H)

■ 描述符、描述表、类型?

■ DS含义? 段寄存器含义?

基址32 限长16

INDEX = 2

DS

13H

描述符

| 字节7 | 字节6 | 字节5 | 字节4 | 字节3 | 字节2 | 字节1 | 字节0 |
|---------------------|-----------|-----|--------------------|-----|-----|---------------------|-----|
| Base 2 (31...24) | Attribute | | Base 1 (23...0) | | | Limit 1 (15...0) | |

Base = Base 2 | Base1

Base

1000H

描述符

描述符

描述符

描述符

描述符

描述符

描述符

描述符

全局描述符表
/GDT

局部描述符表
/LDT

局部描述符表
/LDT

数据段

全局描述符表GDT与局部描述符表LDT

- 全局描述符表GDT

- Global Descriptor Table

- 全局唯一的一个，每个进程可见

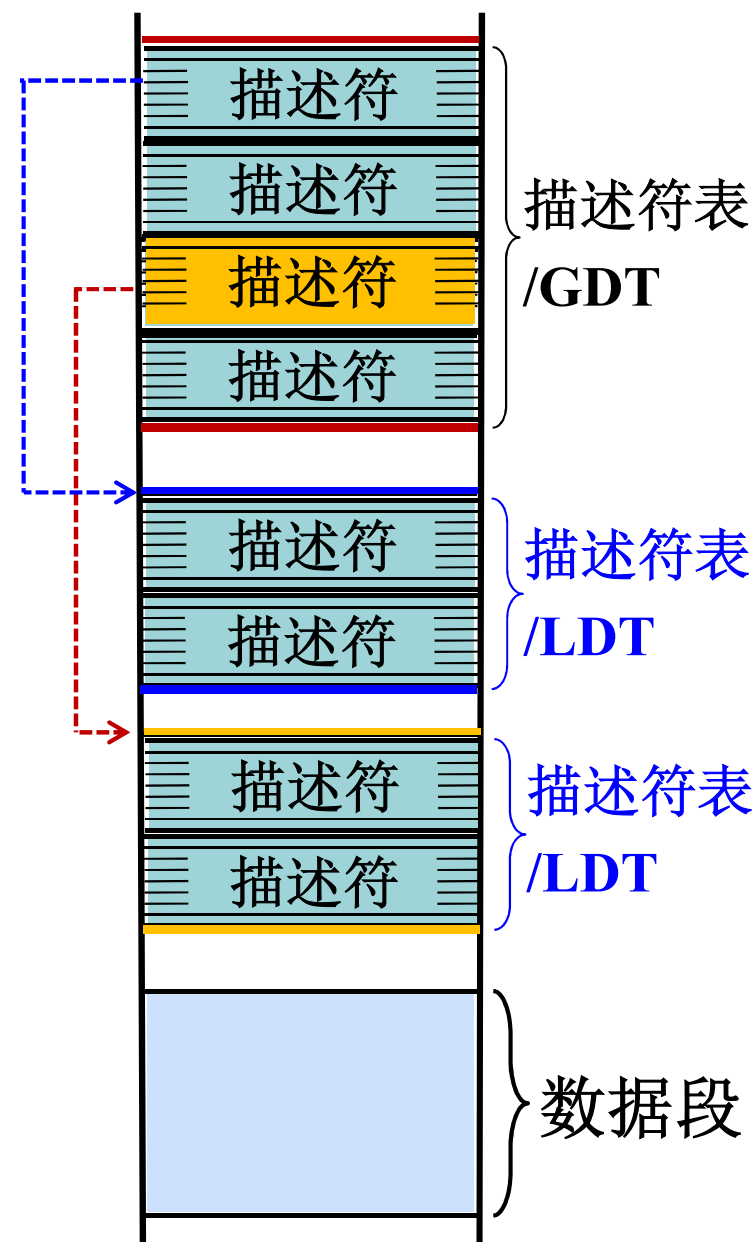
- 局部描述符表LDT

- Local Descriptor Table

- 每个进程/任务一个，表内含有进程的**各种私有段**的描述符。

- LDT的描述符位于GDT中

- ◆ LDT描述符



全局描述符表GDT与局部描述符表LDT

- GDTR (48位)

- GDT的基址 (32位)

- GDT的限长 (16位)

- ◆ GDT中描述符数量 $\leq 8K$

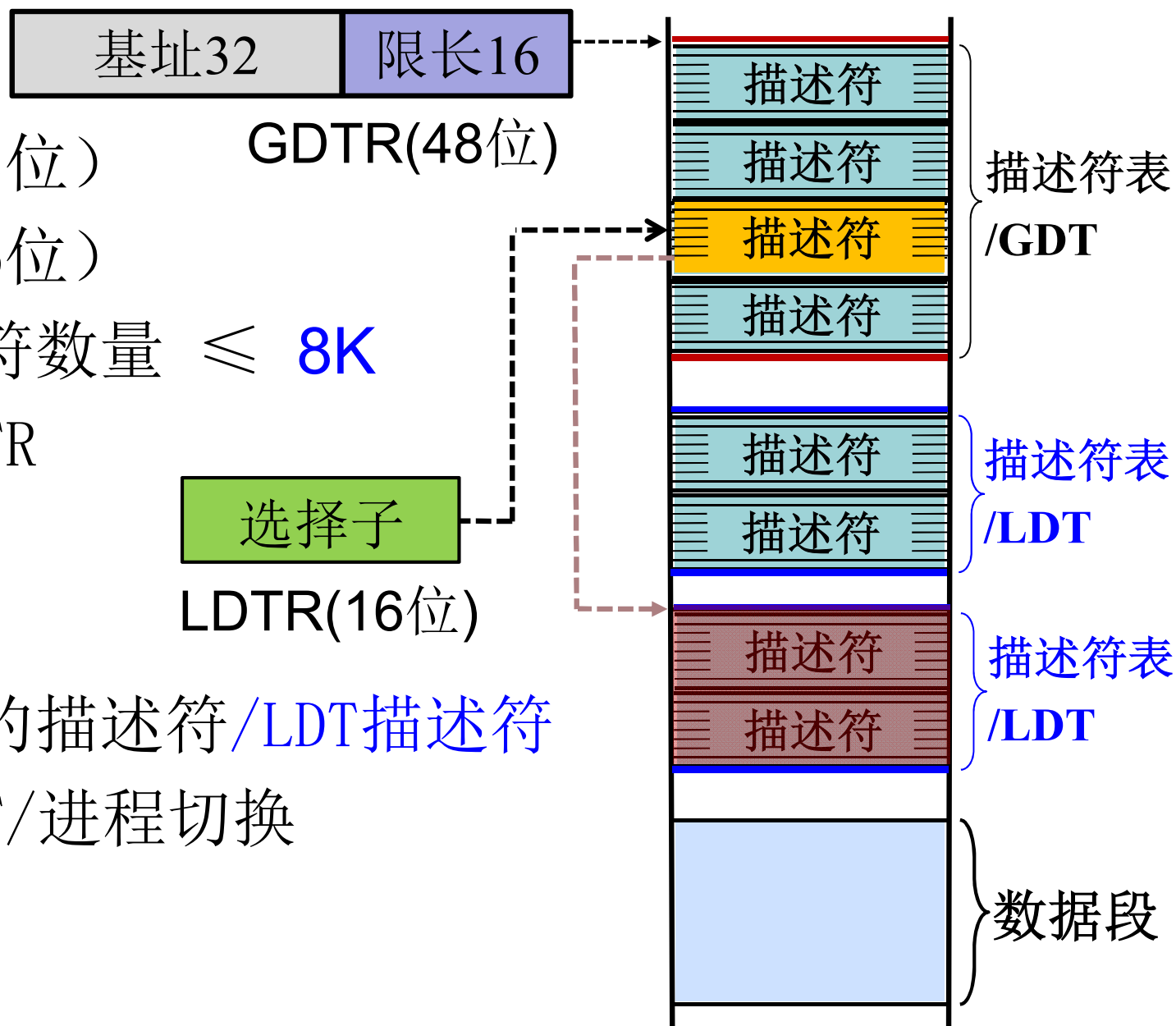
- lgdtr: 更新GDTR

- LDTR (16位)

- 选择子

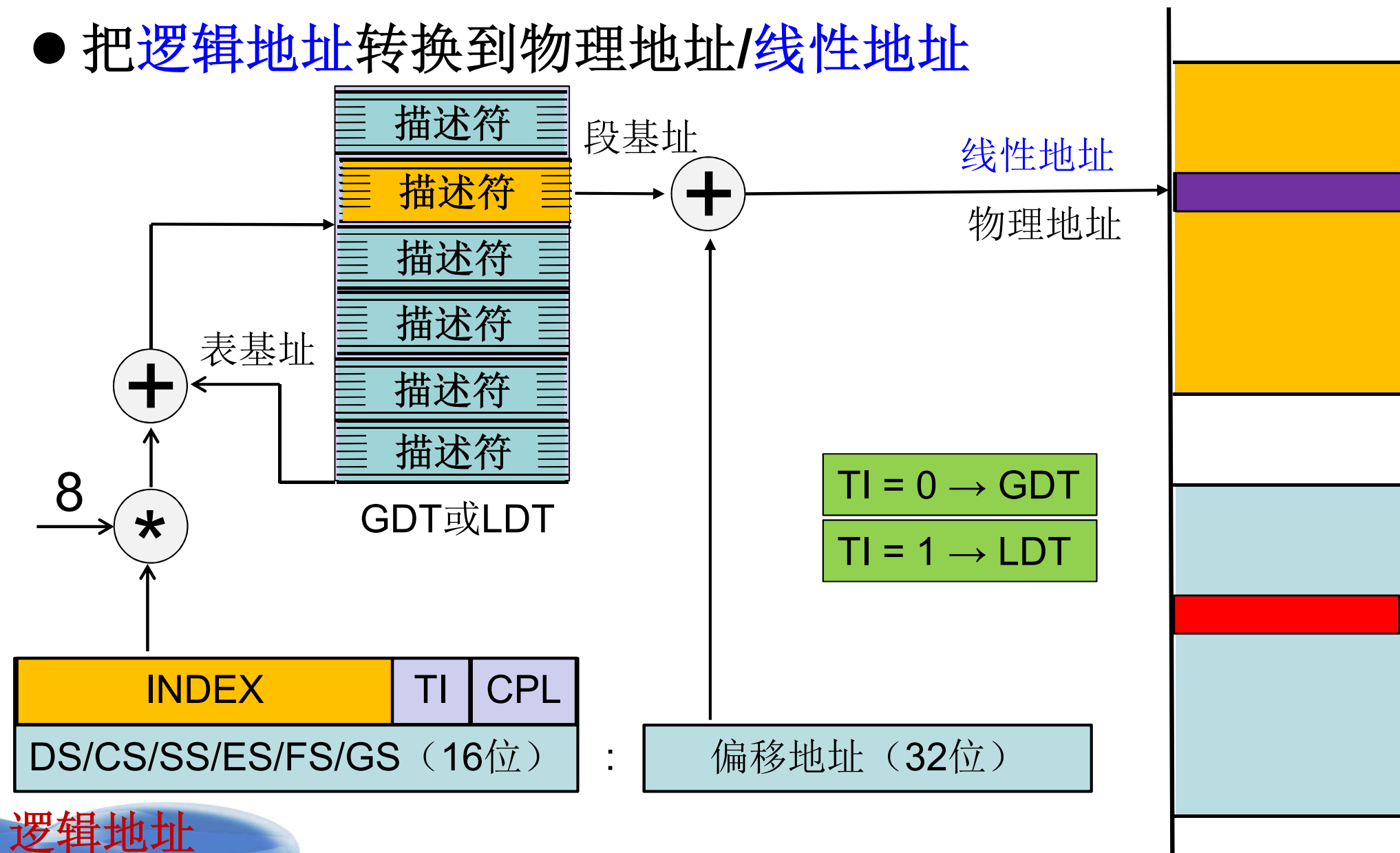
- ◆ 选择GDT中的描述符/LDT描述符

- llldr: 更新LDT/进程切换

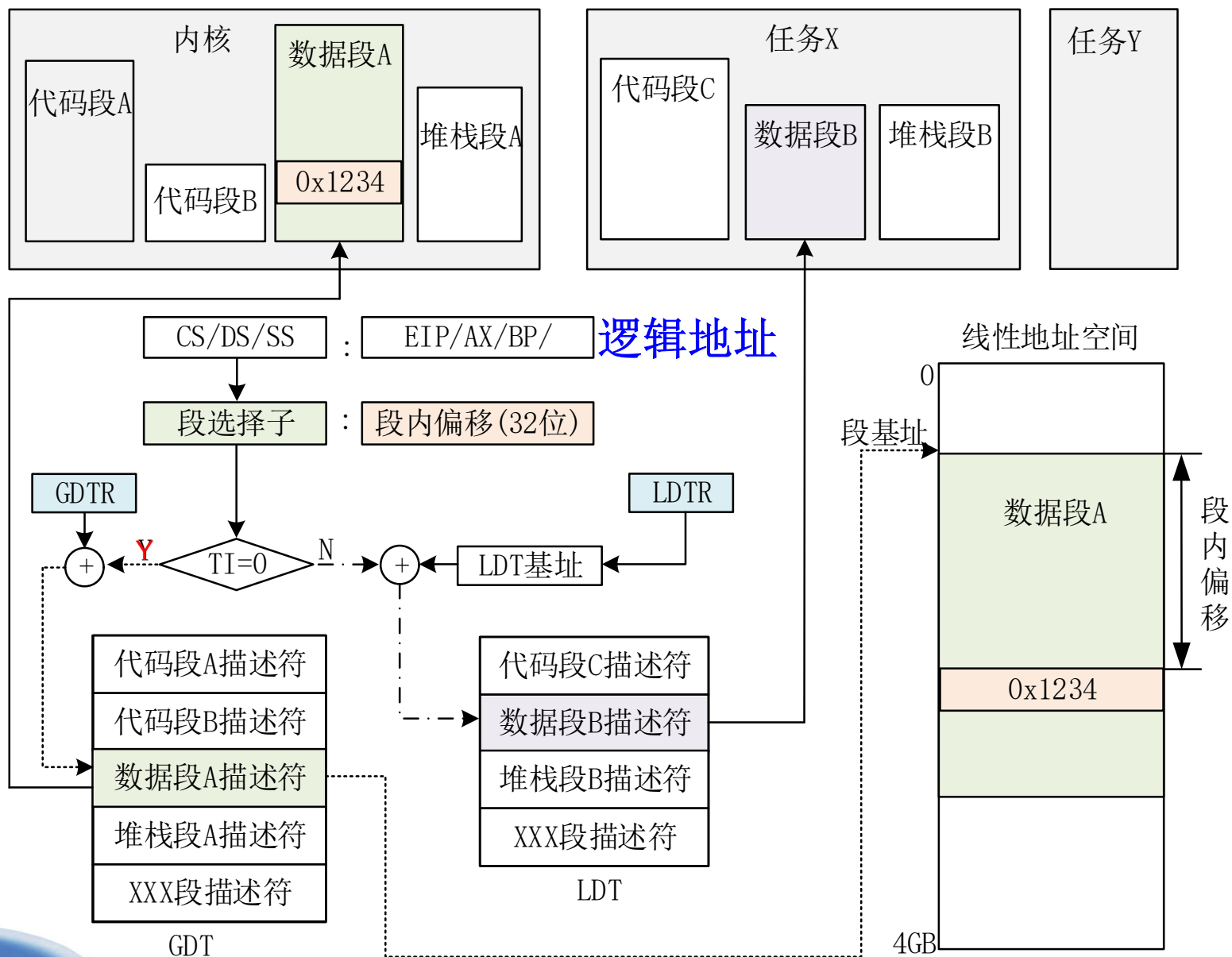


保护模式内存寻址：段式地址转换

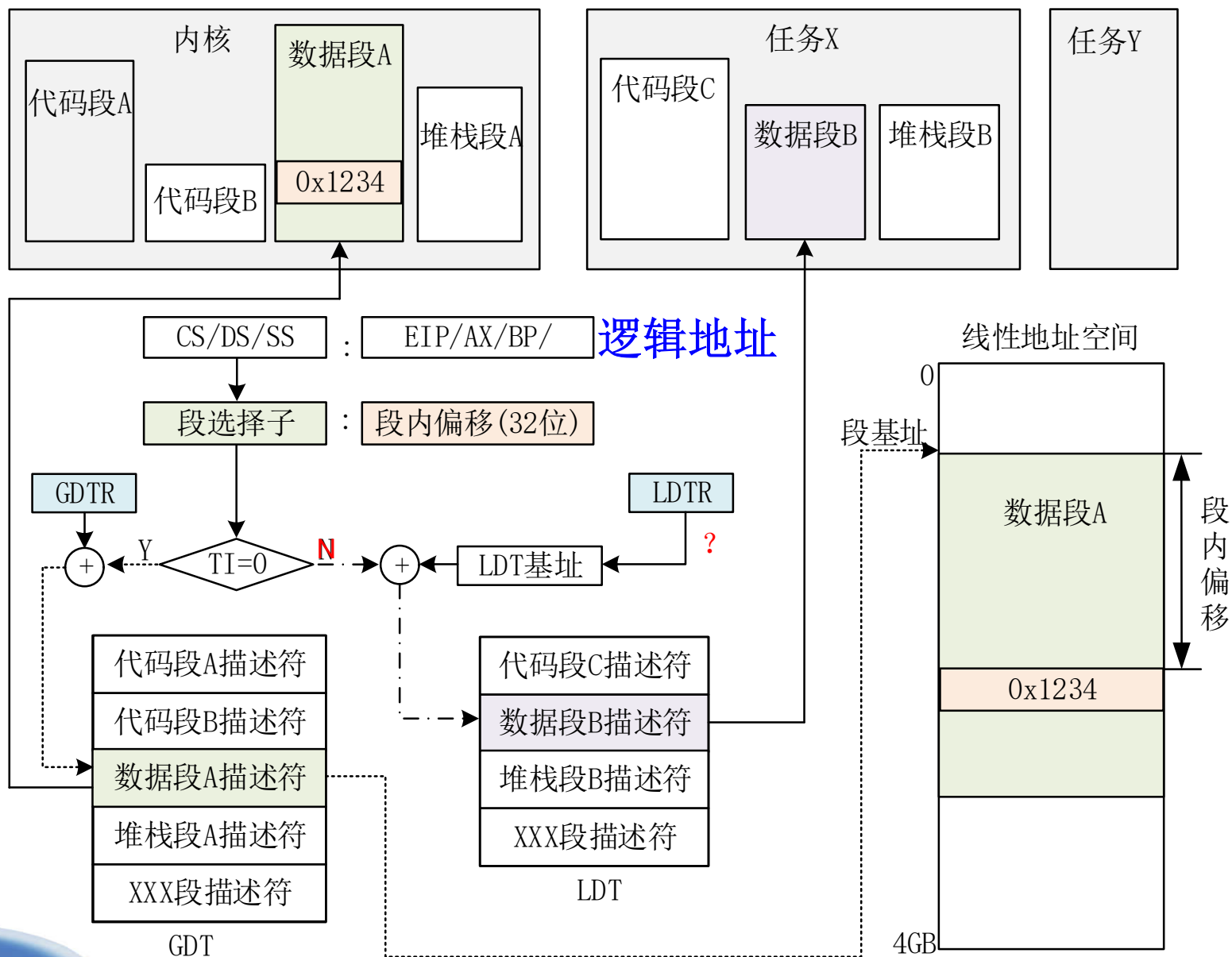
- 把逻辑地址转换到物理地址/线性地址



保护模式内存寻址：段式地址转换



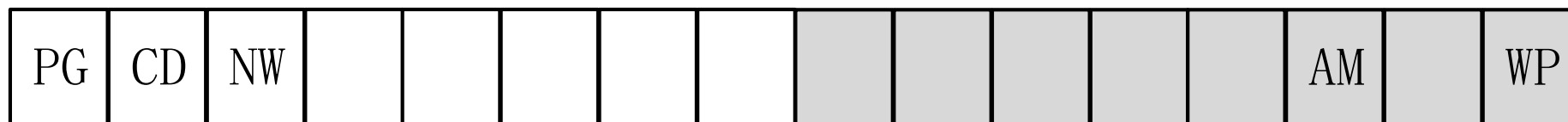
保护模式内存寻址：段式地址转换



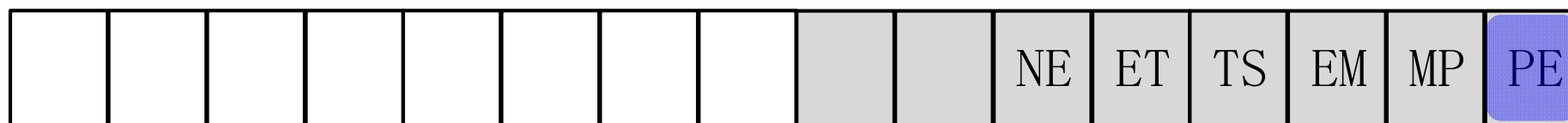
进入保护模式

- 实模式 → 保护模式
 - 控制寄存器CR0.PE

高2字节



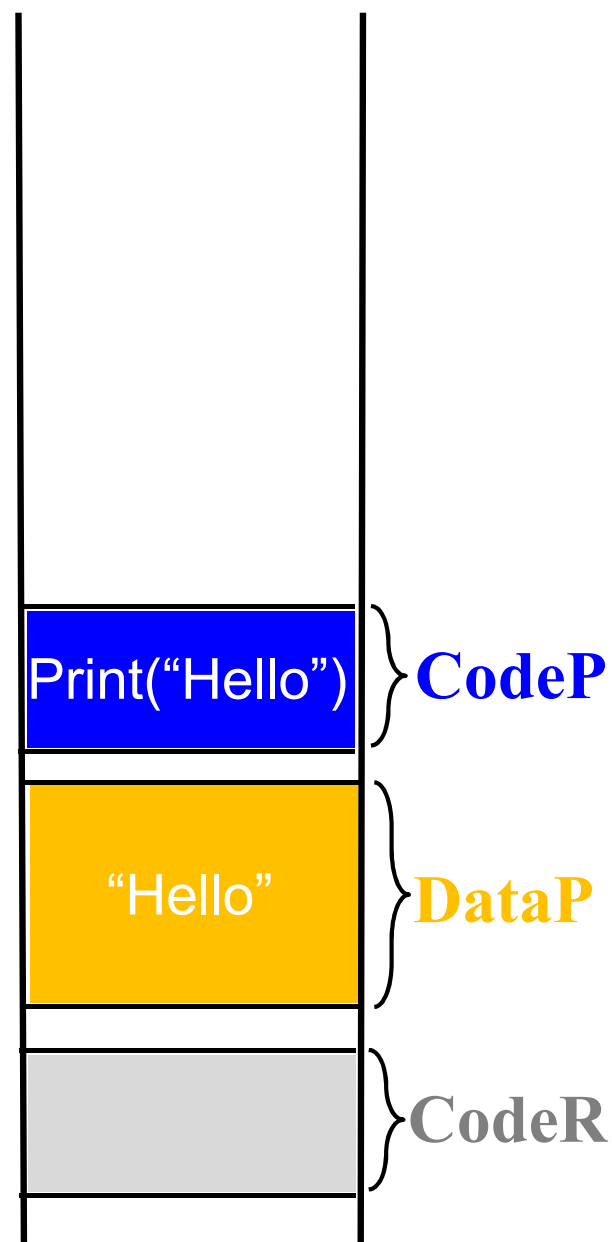
低2字节



■ CR0.PE = 1

进入保护模式

- 实模式 → 保护模式
 - CR0.PE = 1
- 程序结构(mini)
 - CodeR(实模式)
 - ◆ CodeP (保护模式)
 - DataP(数据)



进入保护模式

● CodeR

■ 定义GDT

◆ 定义描述符1

◆ 定义描述符2

■ 定义选择子1

■ 定义选择子2

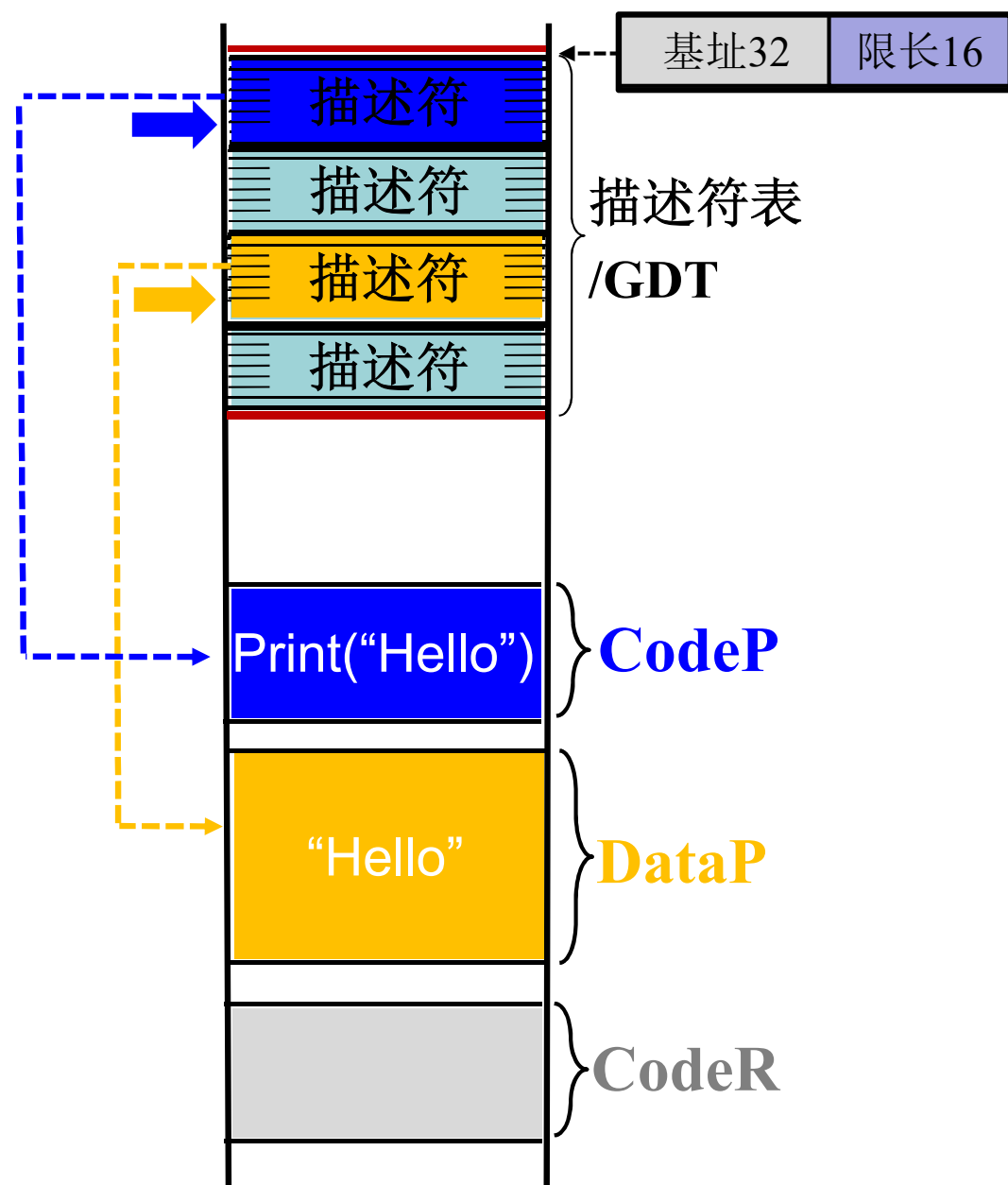
■ 更新描述符1

■ 更新描述符2

■ 装载GDTR

■ CR0.PE = 1

■ JMP CodeP



进入保护模式

- CodeP

- Print("Hello")

- ◆ MOV DataP, "Hello"

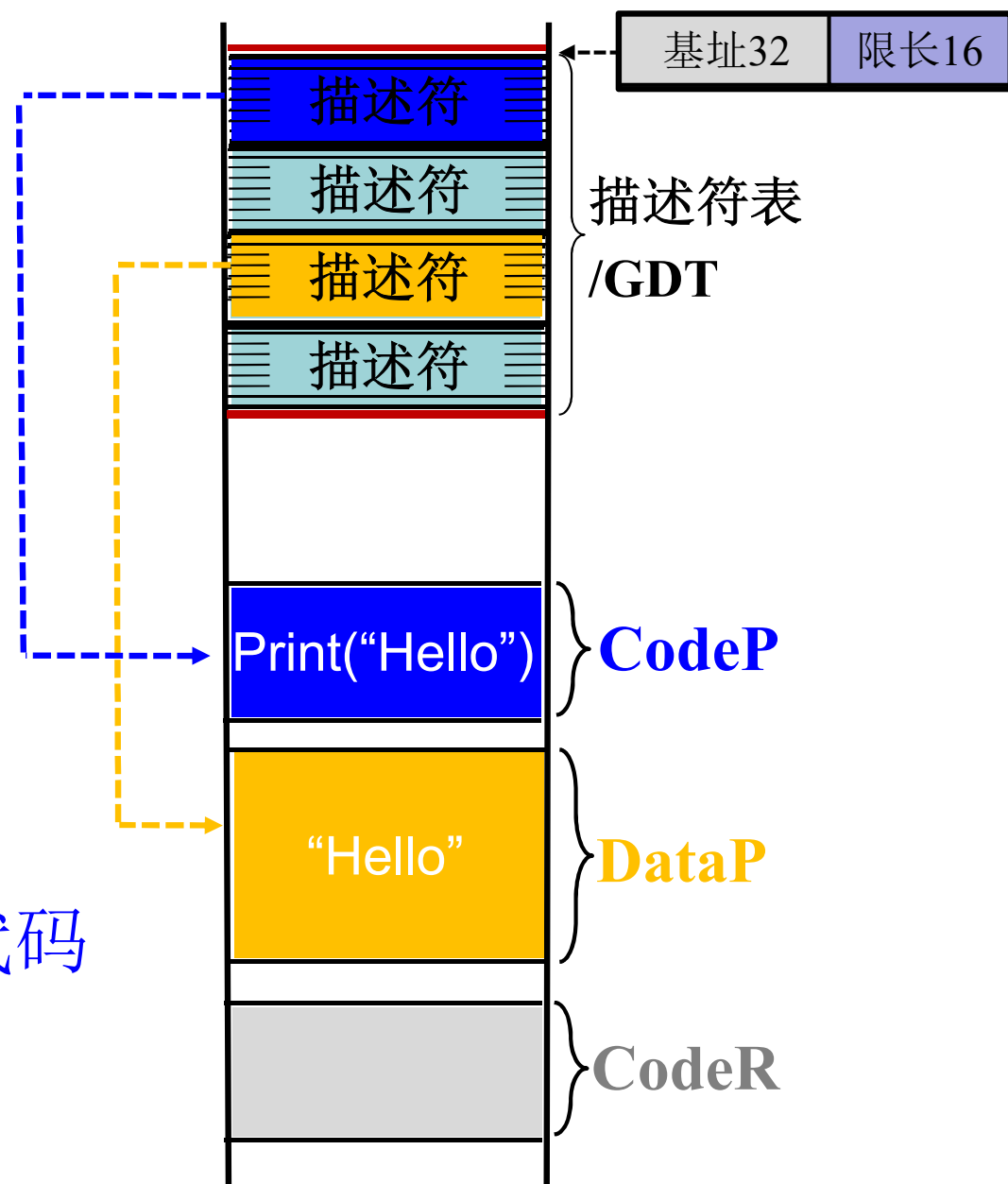
- JMP \$

- DataP

- "Hello"

- CodeR

- JMP CodeP //跳入保护代码



进入保护模式例子 《自己动手编写操作系统》[课外重现]

● 程序入口(定义GDT/描述符/选择子)

```
1 ; 编译方法: nasm pctest1.asm -o pctest1.bin
2 %include "pm.inc" ; 常量, 宏, 以及一些说明
3 ORG 07C00h
4 JMP LABEL_BEGIN
5 [SECTION .GDT]
6 ;                               段基址,                               段界限 , 属性
7 LABEL_GDT : Descriptor 0, 0, 0
8 LABEL_DESC_CODE32: Descriptor 0, SegCode32Len - 1, DA_C + DA_32
9 LABEL_DESC_VIDEO : Descriptor 0B8000h, 0ffffh, DA_DRW
10 ; GDT 结束
11 GdtLen EQU $ - LABEL_GDT ; GDT长度
12 GdtPtr DW GdtLen - 1 ; GDT指针
13 DD 0 ; GDT基址
14 ; GDT 选择子
15 SelectorCode32 EQU LABEL_DESC_CODE32
16 SelectorVideo EQU LABEL_DESC_VIDEO
17 ; END of [SECTION .GDT]
```

```
%macro Descriptor 3
    dw %2 & 0FFFFh
    dw %1 & 0FFFFh
    db ( %1 >> 16) & 0FFh
    dw (( %2 >> 8) & 0F00h) | ( %3 & 0F0FFh))
    db ( %1 >> 24) & 0FFh
%endmacro
```

进入保护模式例子 《自己动手编写操作系统》[课外重现]

● 程序入口(更新描述符)

21 [BITS 16]

22 LABEL_BEGIN:

23 MOV AX, CS

24 MOV DS, AX

25 MOV ES, AX

26 MOV SS, AX

27 MOV SP, 0100h

28

29 ; 初始化 32 位代码段描述符

30 XOR EAX, EAX

31 MOV AX, CS

32 SHL EAX, 4

33 ADD EAX, LABEL_SEG_CODE32

34 MOV WORD [LABEL_DESC_CODE32 + 2], AX

35 SHR EAX, 16

36 MOV BYTE [LABEL_DESC_CODE32 + 4], AL

37 MOV BYTE [LABEL_DESC_CODE32 + 7], AH

| | | | | | | | |
|---------------------|-----------|-----|--------------------|-----|-----|---------------------|-----|
| 字节7 | 字节6 | 字节5 | 字节4 | 字节3 | 字节2 | 字节1 | 字节0 |
| Base 1 (31...24) | Attribute | | Base 2 (23...0) | | | Limit 2 (15...0) | |

进入保护模式例子 《自己动手编写操作系统》[课外重现]

● 装载GDTR

```
38      ; 加载GDTR(48位 = 32位基址:16位限长)
39      XOR EAX, EAX
40      MOV AX, DS
41      SHL EAX, 4
42      ADD EAX, LABEL_GDT ;
43      MOV DWORD [GdtPtr + 2], EAX
44      LGDT [GdtPtr]
```

● 进入保护模式

```
53      ; 准备切换到保护模式
54      ; CR0的第0位PE=1为保护模式
55      MOV EAX, CR0
56      OR EAX, 1
57      MOV CR0, EAX
58      ; 真正进入保护模式
59      JMP DWORD SelectorCode32:0
```

进入保护模式例子 《自己动手编写操作系统》[课外重现]

● 保护模式代码（显示"Hello", "P"）

```
61 [SECTION .s32]; 32 位代码段. 由实模式跳入.
62 [BITS 32]
63 LABEL_SEG_CODE32:
64     MOV AX, SelectorVideo
65     MOV GS, AX ; 视频段选择子(目的)
66
67     MOV EDI, (80 * 11 + 79) * 2 ; 屏幕第 11 行, 第 79 列
68     MOV AH, 0Ch                ; 0000: 黑底 1100: 红字
69     MOV AL, 'P'
70     MOV [GS:EDI], AX
71     ; 到此停止
72     JMP $
73     SegCode32Len EQU $ - LABEL_SEG_CODE32
74 ; END of [SECTION .s32]
```

保护模式

- 地址映射机制
 - 段机制
 - 段页机制
- 中断机制
- 权限级与变迁
 - 权限级的定义
 - 权限级变迁
- 任务与切换
 - 任务的数据结构
 - 任务切换过程

地址映射机制

- 段机制
- 段页机制

段页机制

- 基本概念

- 页(Page): 4KB

- CR0.PG开启分页机制

- 二级页表结构

- ◆ 页目录 | 页表

- ◆ CR3=页目录基地址

- 三种地址

- 逻辑地址

- 线性地址

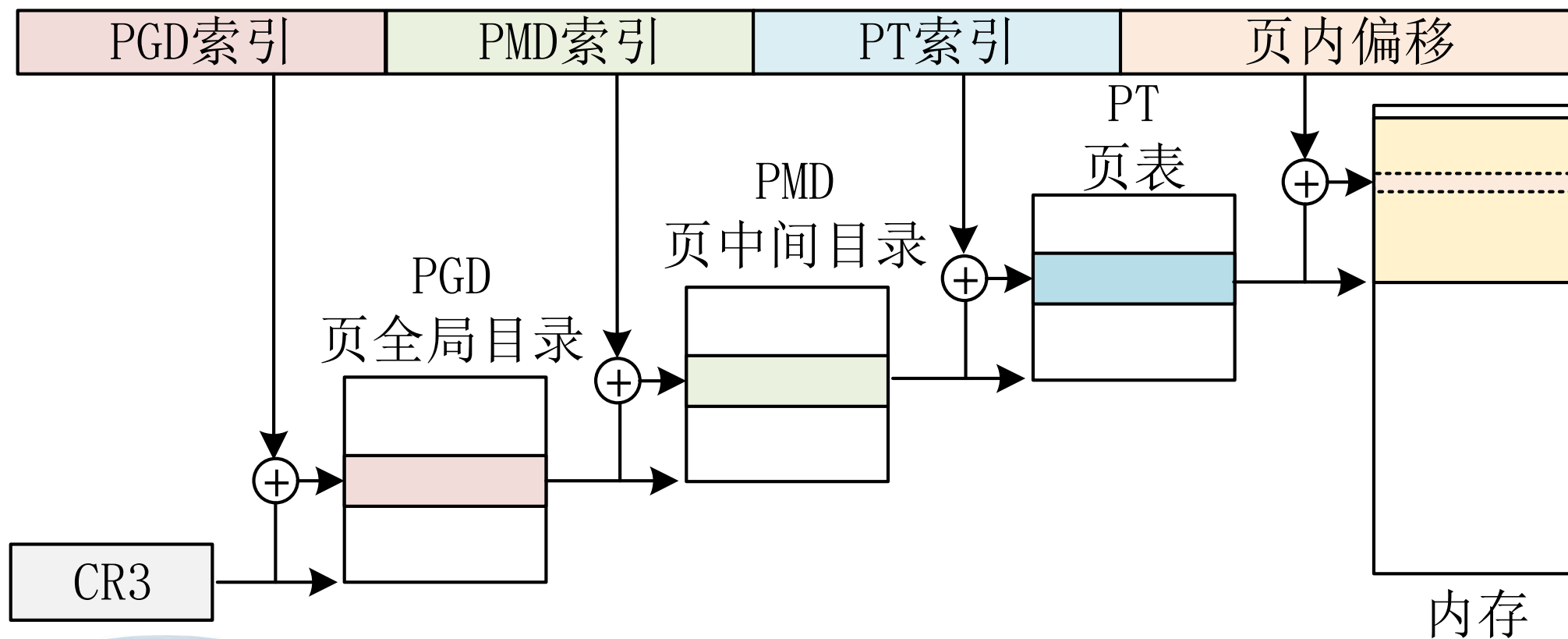
- 物理地址

段页机制

● Linux三级页表

■ PGD = 10位 | PMD = 0位 | PT=10位 | 偏移=12位

线性地址（32位）

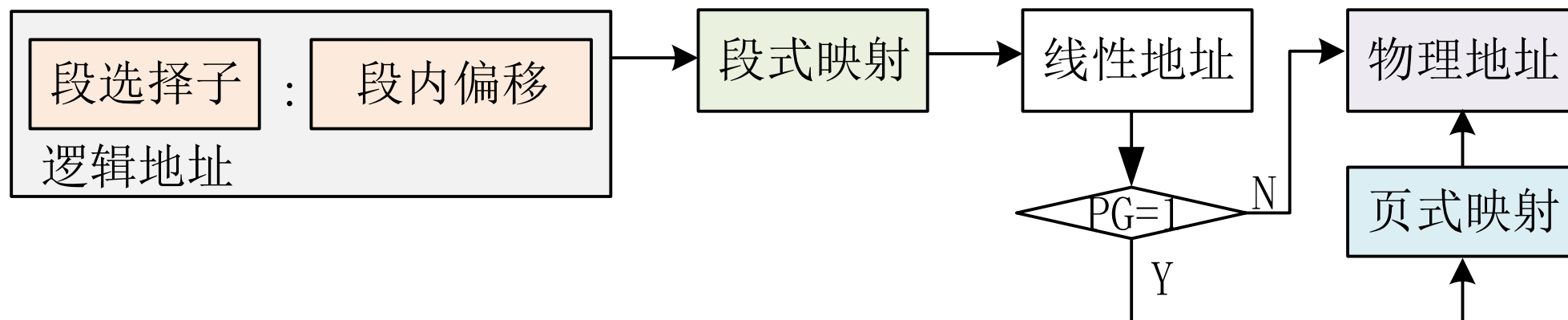


段页机制

- 两种地址映射

- 段式地址映射

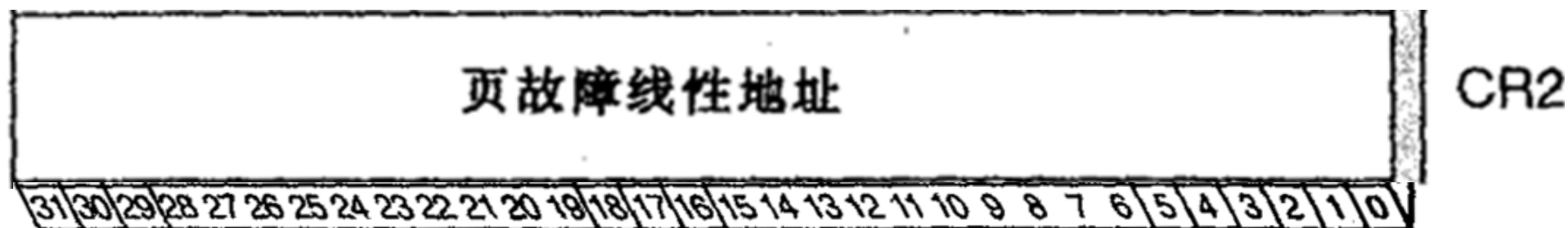
- 页式地址映射



- MMU: Memory Management Unit

段页机制

- 控制寄存器CR2
 - 存放引发缺页的线性地址



Linux段机制@Intel CPU

● 进程分段

| | | |
|---------|-----------|---------|
| ■ 内核数据段 | : 0 ~ 4 G | DPL= 00 |
| ■ 内核代码段 | : 0 ~ 4 G | DPL= 00 |
| ■ 用户数据段 | : 0 ~ 4 G | DPL= 11 |
| ■ 用户代码段 | : 0 ~ 4 G | DPL= 11 |

● 段机制作用

- 利用段隔离用户数据和系统数据
- 简化逻辑地址到线性地址转换
 - ◆ 虚拟地址 = 线性地址

Linux段机制@Intel CPU

- Linux源代码分析

//源文件: 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)
```

Linux段机制@Intel CPU

● Linux源代码分析

//源文件: include/asm-i386/processor.h

```
#define __KERNEL_CS      0x10
#define __KERNEL_DS      0x18
#define __USER_CS        0x23
#define __USER_DS        0x2B
```

ENTRY(gdt_table)

```
.quad 0x 0000 0000 0000 0000 //NULL
.quad 0x 0000 0000 0000 0000 // not used
.quad 0x 00cf 9a00 0000 ffff // Index=2
.quad 0x 00cf 9200 0000 ffff // Index=3
.quad 0x 00cf fa00 0000 ffff // Index=4
.quad 0x 00cf f200 0000 ffff // Index=5
.quad 0x 0000 0000 0000 0000 // not used
.quad 0x 0000 0000 0000 0000 // not used
```

← __KERNEL_CS

← __KERNEL_DS

← __USER_CS

← __USER_DS

Linux段机制@Intel CPU

● Linux源代码分析

ENTRY(gdt_table)

```
.quad 0x 0000 0000 0000 0000 //NULL
.quad 0x 0000 0000 0000 0000 // not used
.quad 0x 00cf 9a00 0000 ffff // Index=2
.quad 0x 00cf 9200 0000 ffff // Index=3
.quad 0x 00cf fa00 0000 ffff // Index=4
.quad 0x 00cf f200 0000 ffff // Index=5
.quad 0x 0000 0000 0000 0000 // not used
.quad 0x 0000 0000 0000 0000 // not used
```

内核数据段
内核代码段
用户数据段
用户代码段

| | |
|-----------|---------|
| : 0 ~ 4 G | DPL= 00 |
| : 0 ~ 4 G | DPL= 00 |
| : 0 ~ 4 G | DPL= 11 |
| : 0 ~ 4 G | DPL= 11 |

| xxxx | xxxx | G | 100 | hhhh | P | DPL | S | T | Y | P | E | xxxx | xxxx | xxxx | xxxx | hhhh | hhhh | hhhh | hhhh |
|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 0000 | 0000 | 1100 | 1111 | 1001 | 1010 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 1111 | 1111 | 1111 | 1111 |
| 0000 | 0000 | 1100 | 1111 | 1001 | 0010 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 1111 | 1111 | 1111 | 1111 |
| 0000 | 0000 | 1100 | 1111 | 1111 | 1010 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 1111 | 1111 | 1111 | 1111 |
| 0000 | 0000 | 1100 | 1111 | 1111 | 0010 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 1111 | 1111 | 1111 | 1111 |

权限级与变迁

- 权限级定义
- 权限级变迁

权限级定义

● 权限级定义

■ **DPL** | RPL | CPL

| 字节7 | 字节6 | 字节5 | 字节4 | 字节3 | 字节2 | 字节1 | 字节0 |
|---------------------|-----------|-----|--------------------|-----|-----|---------------------|-----|
| Base 2 (31...24) | Attribute | | Base 1 (23...0) | | | Limit 1 (15...0) | |

| | | | | | | | | | | | | | | | |
|---|---|---|-----|----------------------|---|---|---|---|-----|---|---|------|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| G | D | 0 | AVL | Limit 2 (19...16) | | | | P | DPL | | S | TYPE | | | |

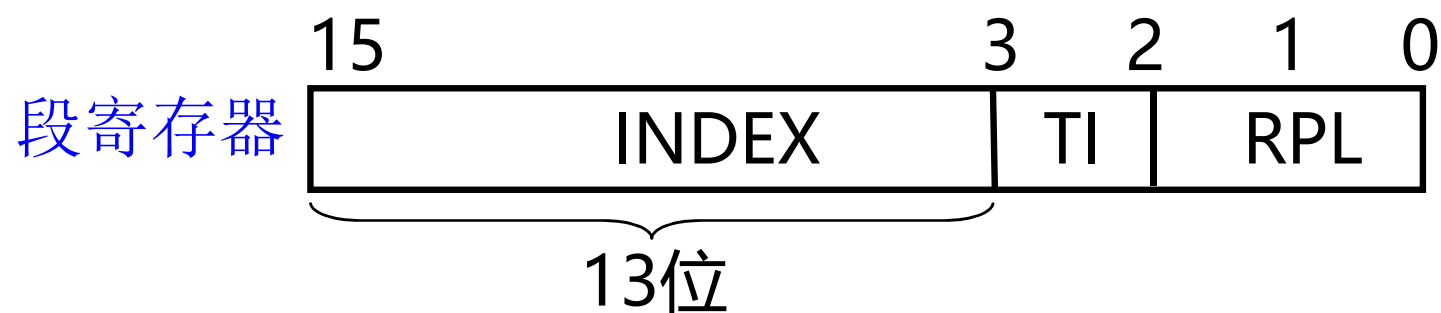
权限级定义

- 权限级定义

- DPL | RPL | CPL

- 请求特权级域

- Request Privilege Level



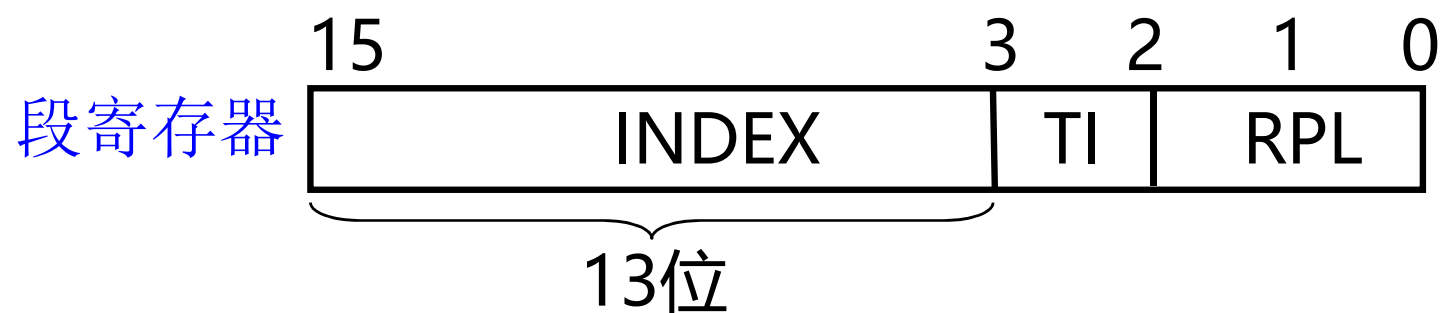
权限级定义

- 权限级定义

- DPL | RPL | **CPL**

- 当前特权级域 (CPL)

- Current Privilege Level



权限级变迁

- 权限级定义

- DPL | RPL | CPL

- 例

- JMP TestCode //CS:IP \leftarrow CS' : TestCode

- CALL TestCode //CS:IP \leftarrow CS' : TestCode

- MOV AX, Table //DS : Table

- 合法的访问

- CPL = DPL

- CPL DPL

权限级变迁

● 门

■ 控制转移的目标代码入口

■ 目的

◆ 任务内跨特权级跳转

◆ 任务间的切换

◆ 中断响应→中断服务

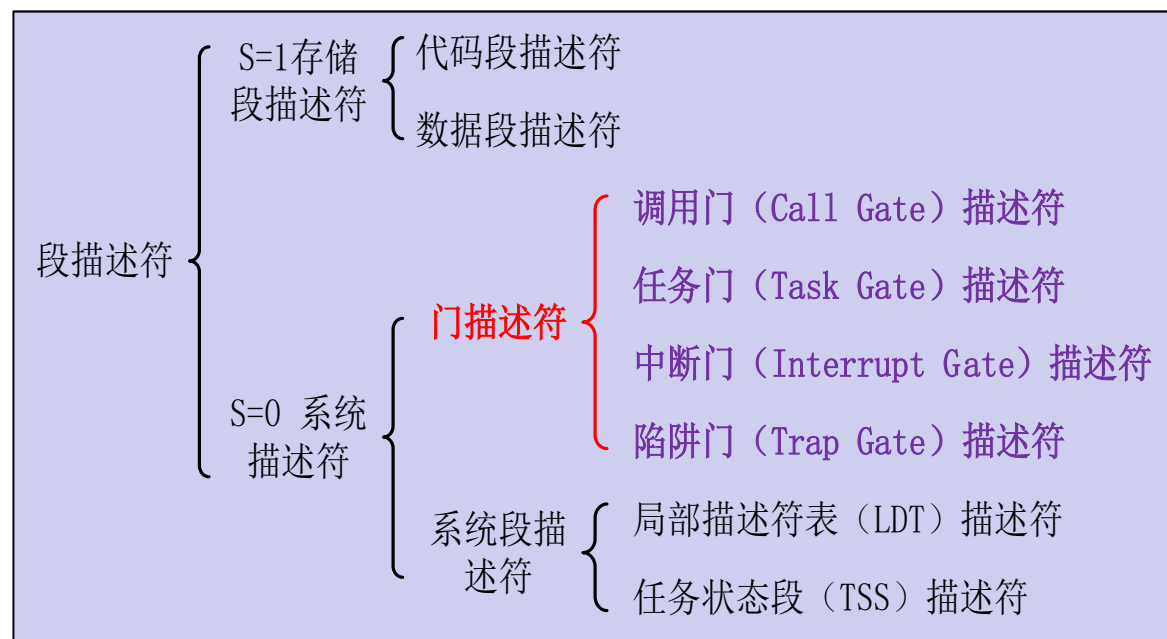
■ 类型

◆ 调用门 | 任务门 | 中断门 | 陷阱门

● 门描述符

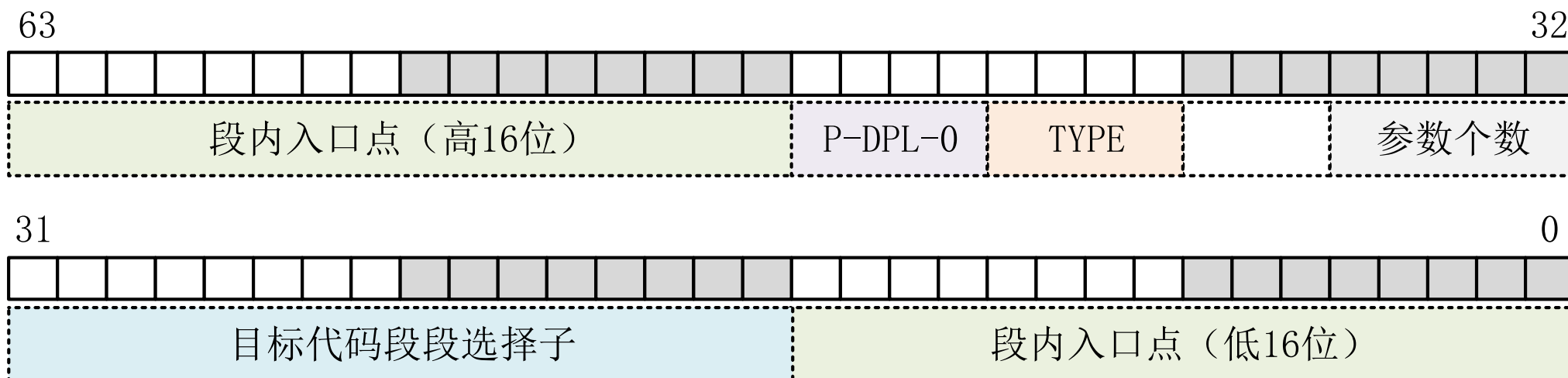
■ 描述门的相关信息

◆ 调用门描述符 | 任务门描述符 | 中断门描述符 | 陷阱门描述符



权限级变迁

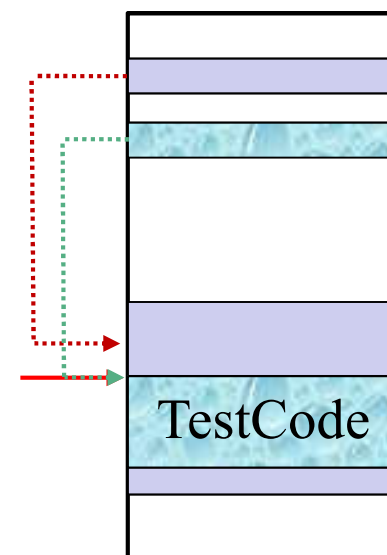
● 调用门描述符



● 例[回顾]

■ `JMP TestCode` // `CS:IP ← CS' : TestCode`

■ `CALL TestCode` // `CS:IP ← CS' : TestCode`

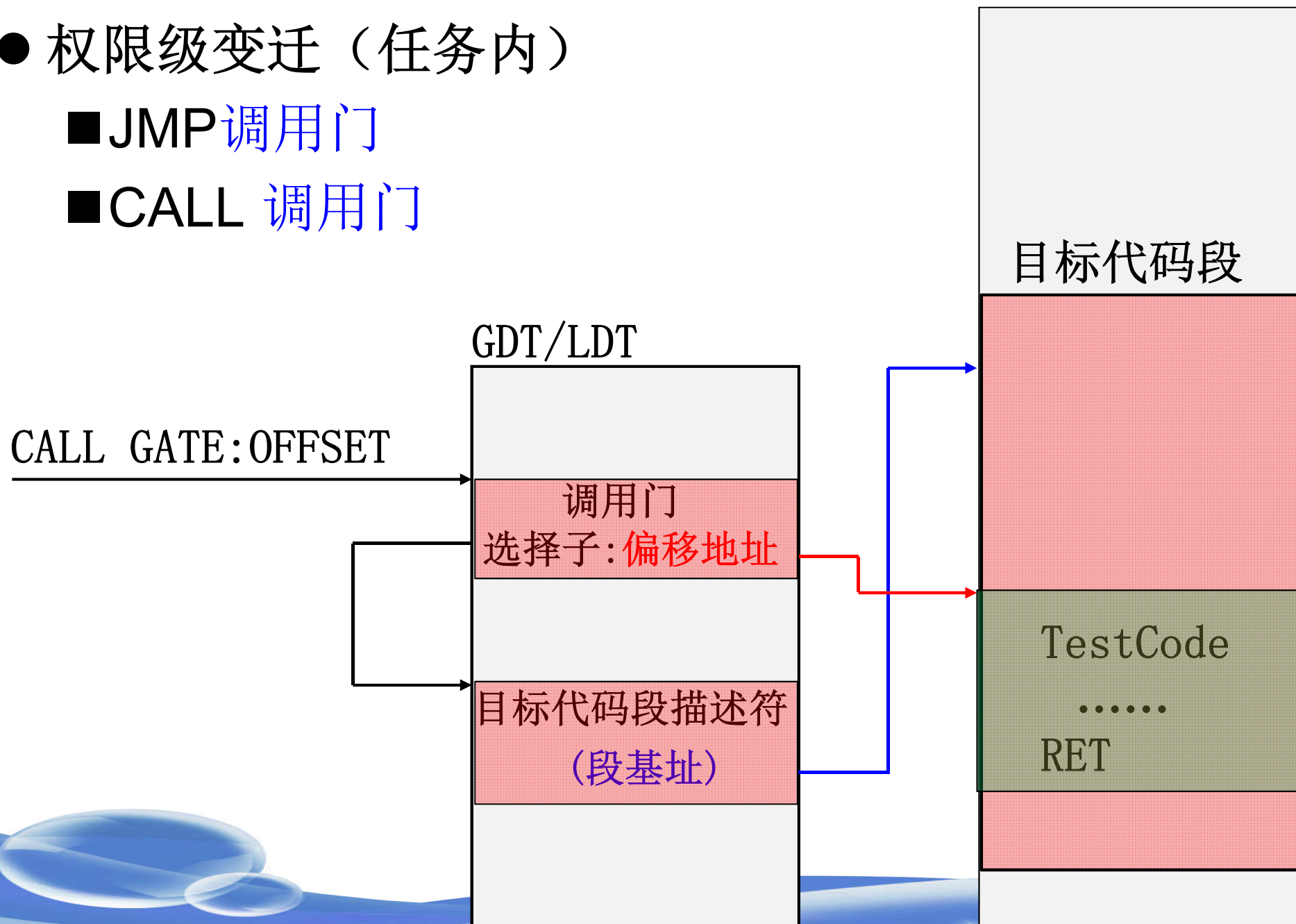


权限级变迁

- 权限级变迁（任务内）

- JMP调用门

- CALL 调用门



中断机制

● 门

■ 控制转移的目标代码入口

■ 目的

◆ 任务内跨特权级跳转

◆ 任务间的切换

◆ 中断响应→中断服务

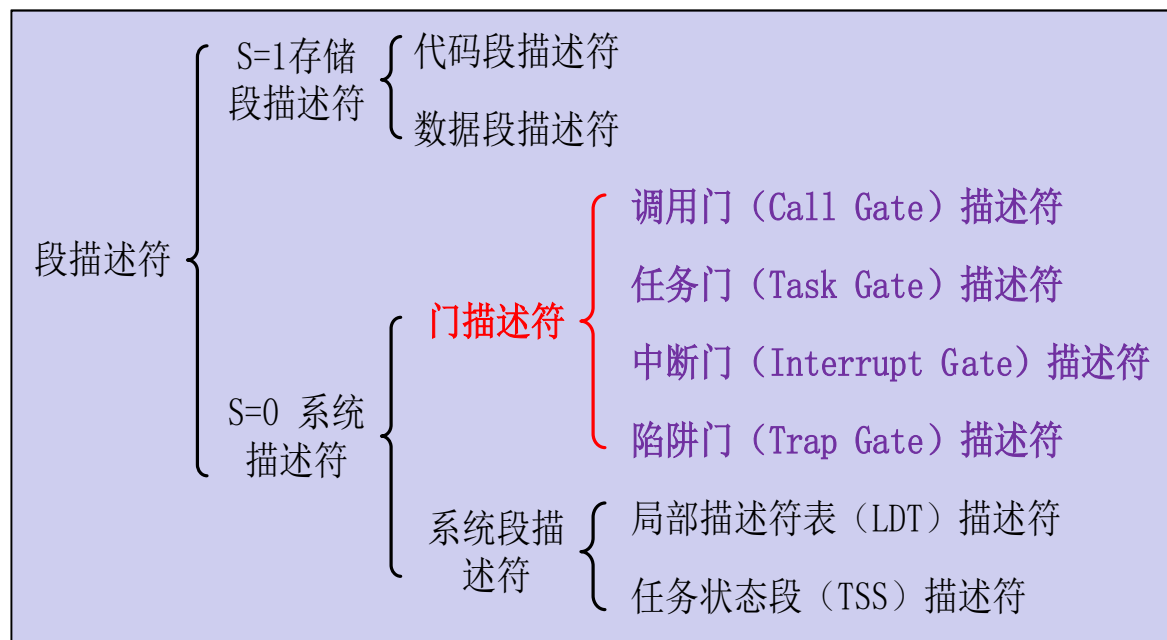
■ 类型

◆ 调用门 | 任务门 | 中断门 | 陷阱门

● 门描述符

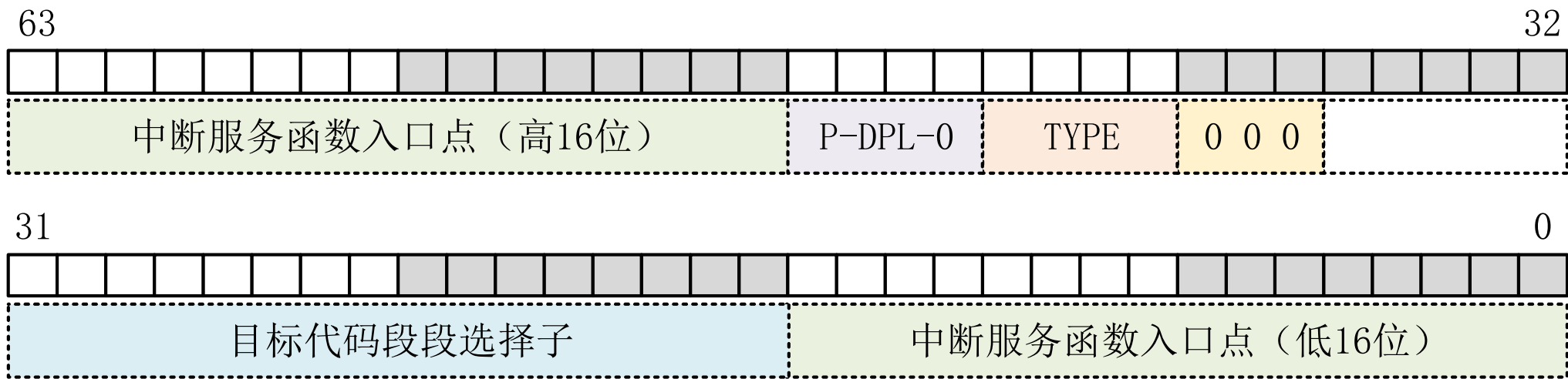
■ 描述门的相关信息

◆ 调用门描述符 | 任务门描述符 | 中断门描述符 | 陷阱门描述符

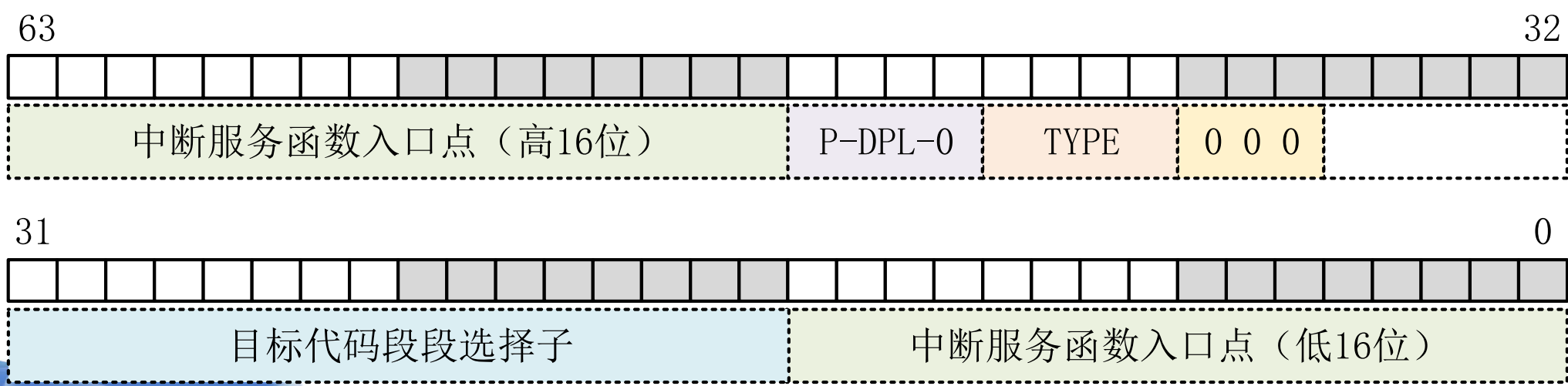


中断门描述符/陷阱门描述符

● 中断门描述符



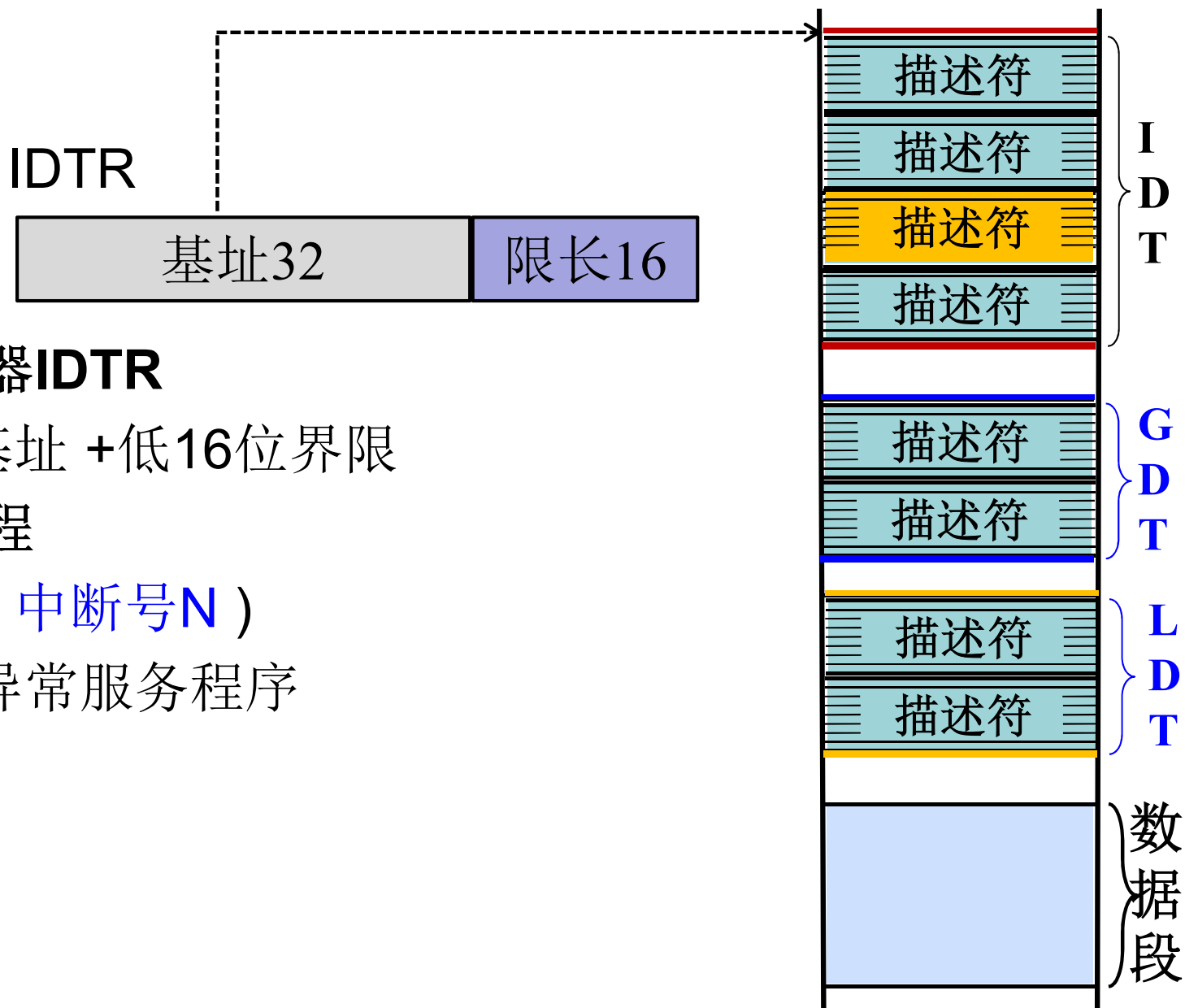
● 陷阱门描述符



中断机制

- 中断描述符表IDT

- 中断门
- 陷阱门
- 任务门



- 中断描述符表寄存器IDTR

- 48位 = 高32位基址 + 低16位界限

- 中断或异常响应过程

- 门描述符 = IDT(中断号N)
 - ◆ 进入中断或异常服务程序

[回顾]程序员处理中断的过程

- 填写中断向量表 (IVT)

- 时钟中断

- 20H中断

| |
|-----------------|
| CS:IP (X+1) |
| CS:IP (X+1) |
| CS:IP (X) (20h) |
| CS:IP (X-1) |

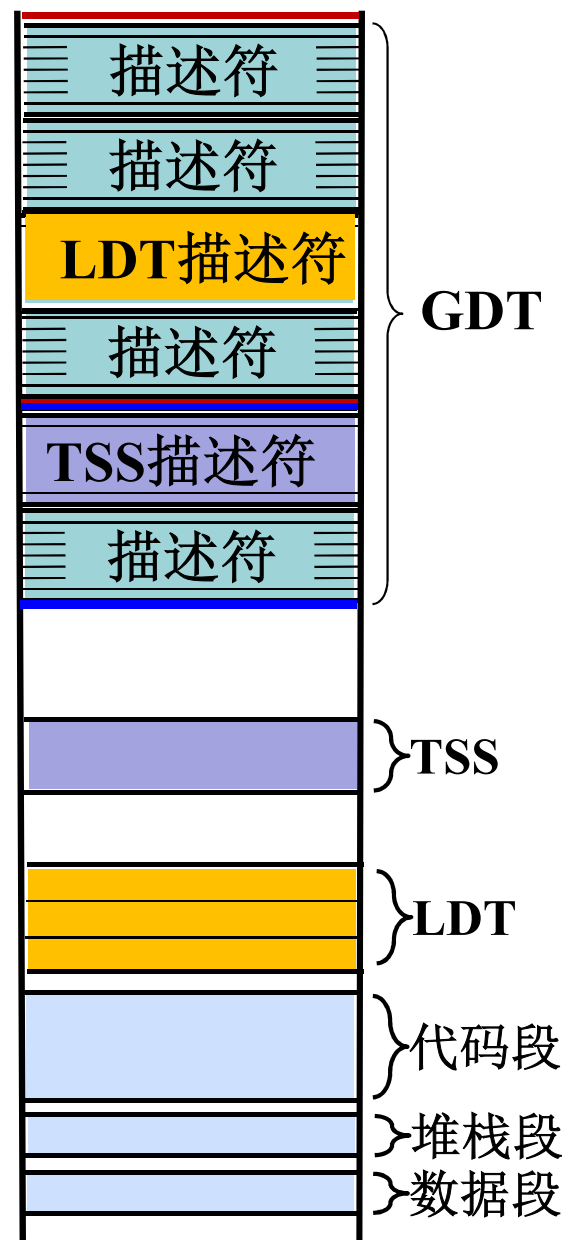
```
192 LABEL_IDT:
193 ;          门          目标选择子,          偏移,          DCount,  属性
194 ;重复填写32个无关紧要的中断
195 %rep 32
196 | Gate  SelectorCode32, SpuriousHandler,  0,  DA_386IGate
197 %endrep
198 ;20h是时钟中断
199 .020h: Gate  SelectorCode32, ClockHandler,  0,  DA_386IGate
200
201 ;重复填写95个无关紧要的中断
202 %rep 95
203 | Gate  SelectorCode32, SpuriousHandler,  0,  DA_386IGate
204 %endrep
205 ;80h是系统调用
206 .080h: Gate  SelectorCode32, UserIntHandler,  0,  DA_386IGate
```

任务与切换

- 任务的概念
- 任务的切换
- 任务与中断

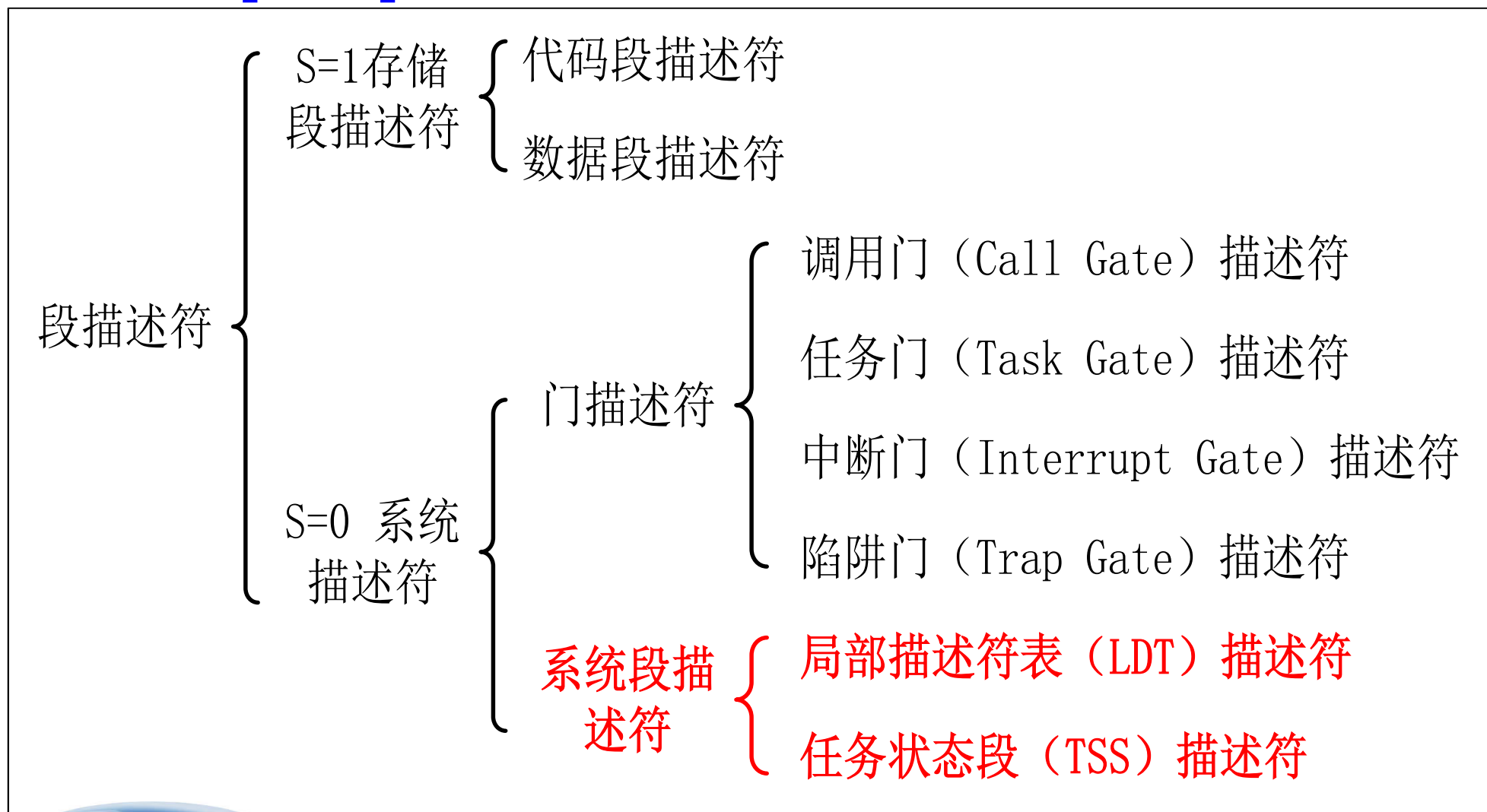
任务的概念

- 任务定义
 - 代码, 数据, 堆栈, 上下文
 - LDT, TSS
- 任务的数据结构
 - LDT (局部描述符表)
 - ◆ LDT描述符(S=0)
 - ◆ 描述符(S=1)
 - ◆ LDTR/lldtr指令
 - TSS (任务状态段)
 - ◆ TSS描述符(S=0)
 - ◆ TR/ltr指令



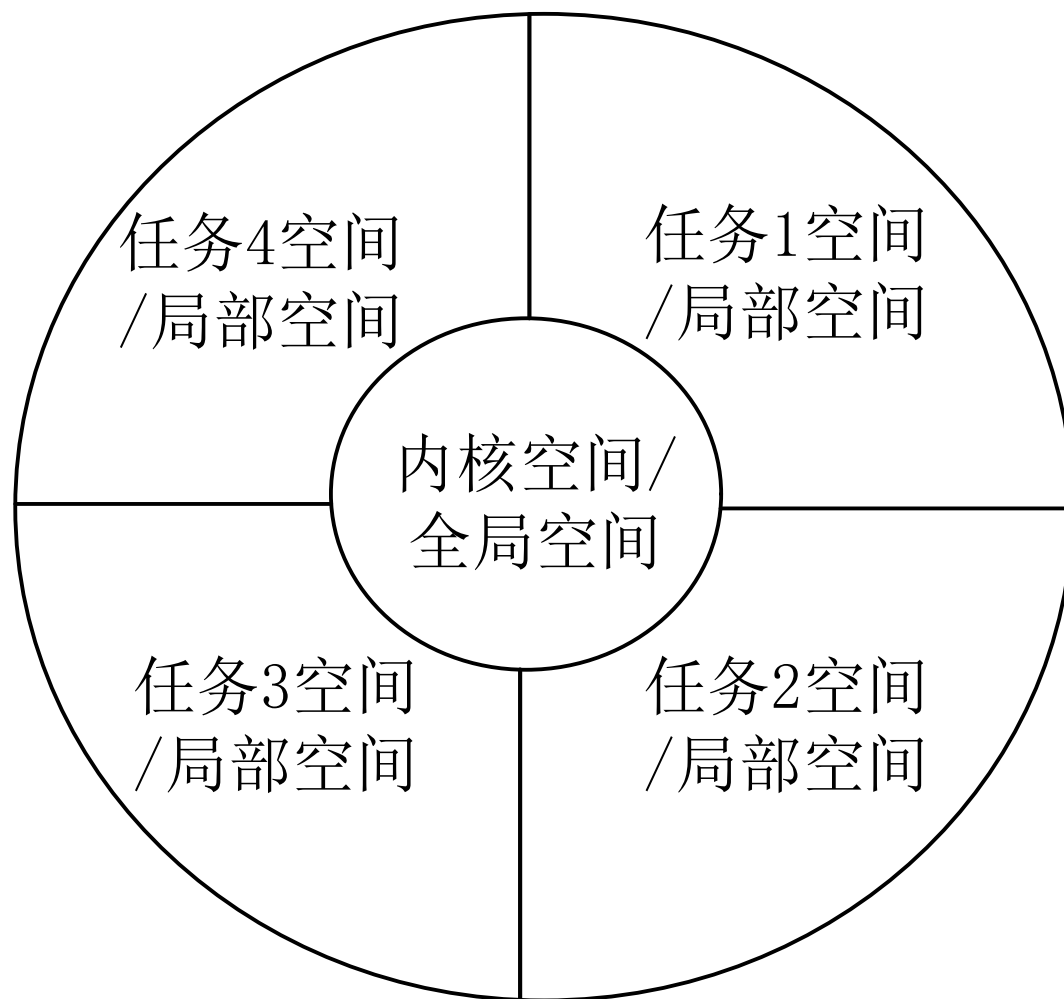
任务的概念

● 描述符[回顾]



任务的概念

● 任务的隔离



任务状态段 (Task State Segment)

(104字节)

- 保存任务上下文和基本属性

| | | | | |
|----------------------|----|----|----------------------|-----|
| 31 | 16 | 15 | 0 | |
| I/O Map Base Address | | | T | 100 |
| | | | LDT Segment Selector | 96 |
| | | | GS | 92 |
| | | | FS | 88 |
| | | | DS | 84 |
| | | | SS | 80 |
| | | | CS | 76 |
| | | | ES | 72 |
| EDI | | | | 68 |
| ESI | | | | 64 |
| EBP | | | | 60 |
| ESP | | | | 56 |
| EBX | | | | 52 |
| EDX | | | | 48 |
| ECX | | | | 44 |
| EAX | | | | 40 |
| EFLAGS | | | | 36 |
| EIP | | | | 32 |
| CR3 (PDBR) | | | | 28 |
| | | | SS2 | 24 |
| ESP2 | | | | 20 |
| | | | SS1 | 16 |
| ESP1 | | | | 12 |
| | | | SS0 | 8 |
| ESP0 | | | | 4 |
| | | | Previous Task Link | 0 |

任务状态段 (Task State Segment) - 高48字节

- 保存任务上下文和基本属性

| | | | | |
|----------------------|----|----|----------------------|-----|
| 31 | 16 | 15 | 0 | |
| I/O Map Base Address | | | T | 100 |
| | | | LDT Segment Selector | 96 |
| | | | GS | 92 |
| | | | FS | 88 |
| | | | DS | 84 |
| | | | SS | 80 |
| | | | CS | 76 |
| | | | ES | 72 |
| EDI | | | | 68 |
| ESI | | | | 64 |
| EBP | | | | 60 |
| ESP | | | | 56 |

任务状态段 (Task State Segment) - 低56字节

- 保存任务上下文和基本属性

| | | | |
|------------|-------|--------------------|----|
| 31 | 16 15 | 0 | |
| EBX | | | 52 |
| EDX | | | 48 |
| ECX | | | 44 |
| EAX | | | 40 |
| EFLAGS | | | 36 |
| EIP | | | 32 |
| CR3 (PDBR) | | | 28 |
| | | SS2 | 24 |
| ESP2 | | | 20 |
| | | SS1 | 16 |
| ESP1 | | | 12 |
| | | SS0 | 8 |
| ESP0 | | | 4 |
| | | Previous Task Link | 0 |

任务状态段 (Task State Segment)

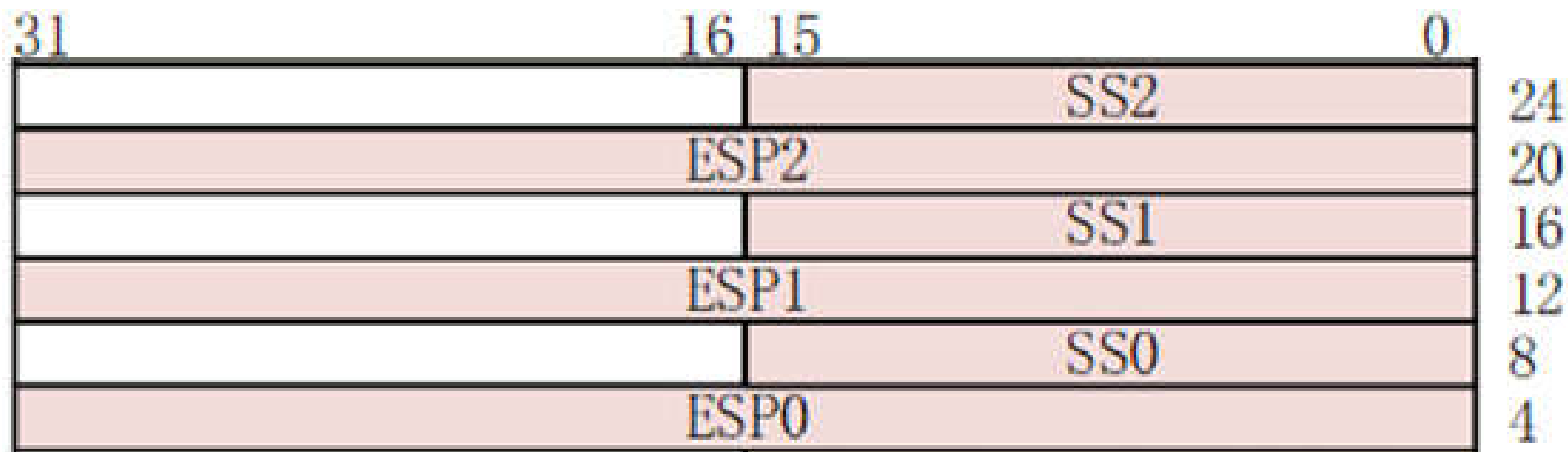
- 内层堆栈指针区域

- 0级堆栈 | 1级堆栈 | 2级堆栈的栈指针

- 特权级提升时取出相应级别的栈指针装入SS及ESP

- ◆ 特权级提升时切换到高特权级堆栈

- ◆ 低特权级SS及ESP保存在新的高特权级堆栈中。



任务状态段 (Task State Segment)

- 上下文寄存器区域

| 31 | 16 | 15 | 0 | |
|----------------------|----|----|---|----|
| LDT Segment Selector | | | | 96 |
| GS | | | | 92 |
| FS | | | | 88 |
| DS | | | | 84 |
| SS | | | | 80 |
| CS | | | | 76 |
| ES | | | | 72 |
| EDI | | | | 68 |
| ESI | | | | 64 |
| EBP | | | | 60 |
| ESP | | | | 56 |
| EBX | | | | 52 |
| EDX | | | | 48 |
| ECX | | | | 44 |
| EAX | | | | 40 |
| EFLAGS | | | | 36 |
| EIP | | | | 32 |
| CR3 (PDBR) | | | | 28 |

任务状态段 (Task State Segment)

● 任务PDBR寄存器

| 31 | 16 15 | 0 | |
|----|-------|---|-------------------------|
| | | | LDT Segment Selector 96 |
| | | | GS 92 |
| | | | FS 88 |
| | | | DS 84 |
| | | | SS 80 |
| | | | CS 76 |
| | | | ES 72 |
| | | | EDI 68 |
| | | | ESI 64 |
| | | | EBP 60 |
| | | | ESP 56 |
| | | | EBX 52 |
| | | | EDX 48 |
| | | | ECX 44 |
| | | | EAX 40 |
| | | | EFLAGS 36 |
| | | | EIP 32 |
| | | | CR3 (PDBR) 28 |

任务状态段 (Task State Segment)

● 任务PDBR寄存器

| 31 | 16 15 | 0 | |
|----|-------|---|-------------------------|
| | | | LDT Segment Selector 96 |
| | | | GS 92 |
| | | | FS 88 |
| | | | DS 84 |
| | | | SS 80 |
| | | | CS 76 |
| | | | ES 72 |
| | | | EDI 68 |
| | | | ESI 64 |
| | | | EBP 60 |
| | | | ESP 56 |
| | | | EBX 52 |
| | | | EDX 48 |
| | | | ECX 44 |
| | | | EAX 40 |
| | | | EFLAGS 36 |
| | | | EIP 32 |
| | | | CR3 (PDBR) 28 |

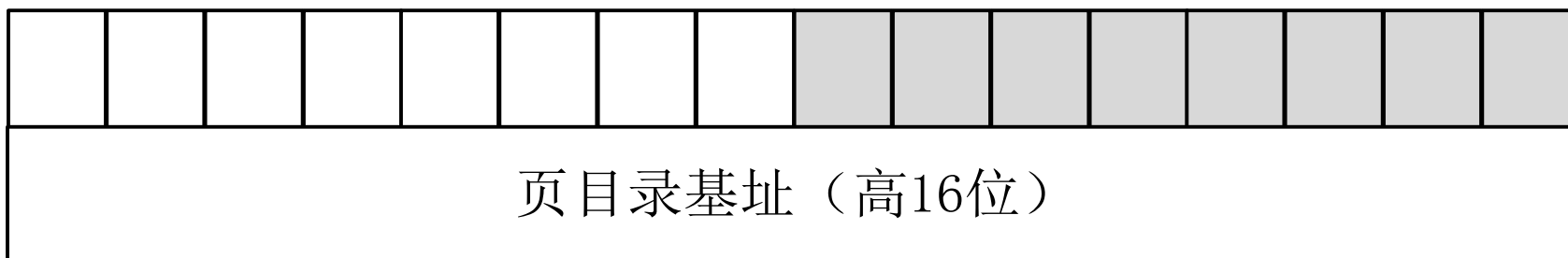
任务状态段 (Task State Segment)

● 控制寄存器CR3(PDBR)

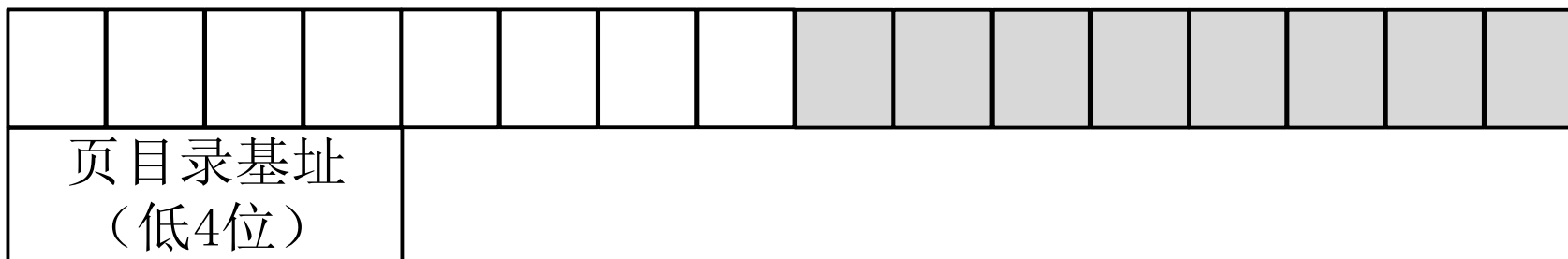
■ 页目录基址：高20位



高2字节



低2字节



任务状态段 (Task State Segment)

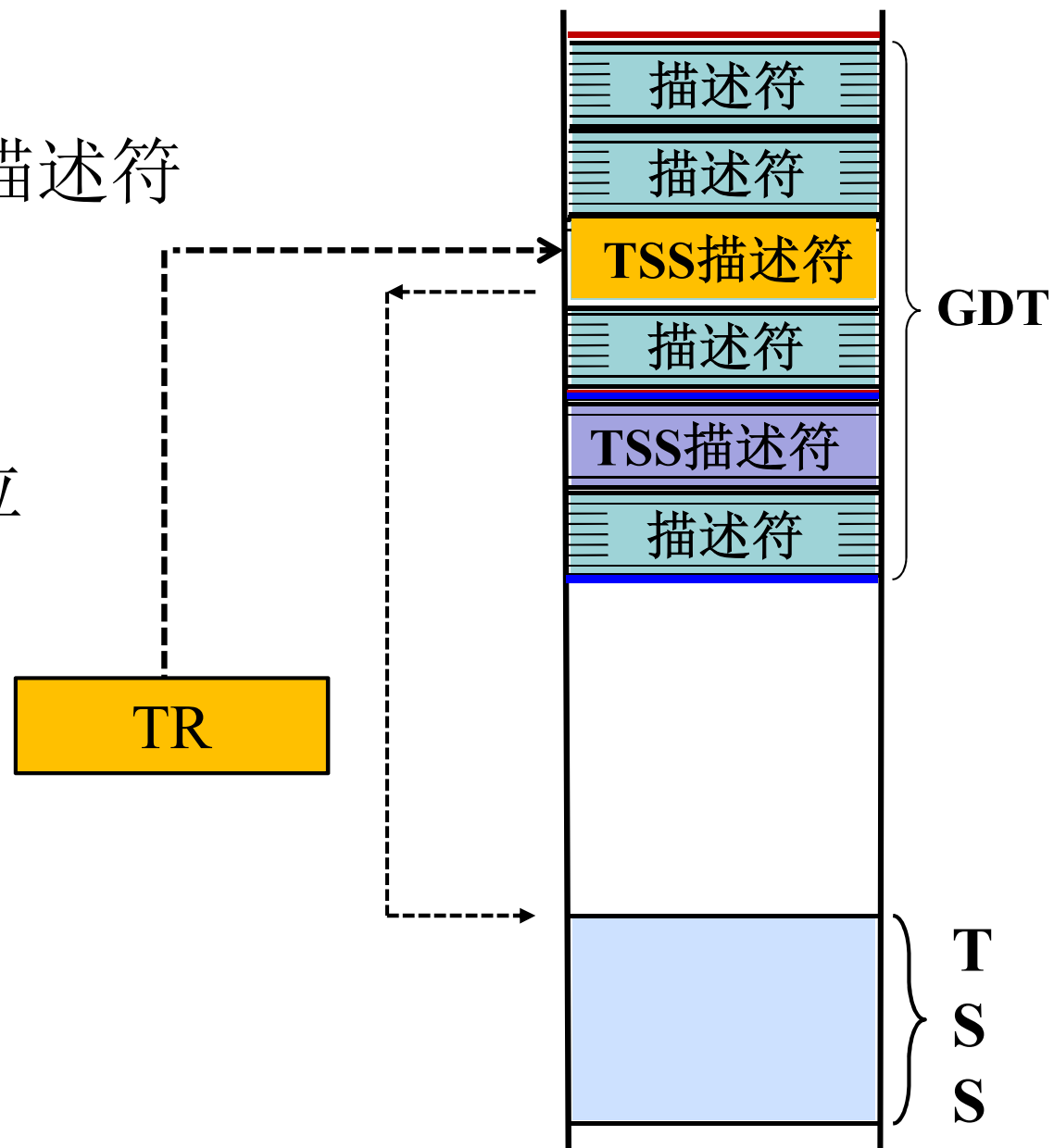
- TSS描述符

- 描述任务状态段TSS的描述符

- TSS描述符存放在GDT中

- TR

- TSS描述符选择子/16位



任务的切换

- 切换指令

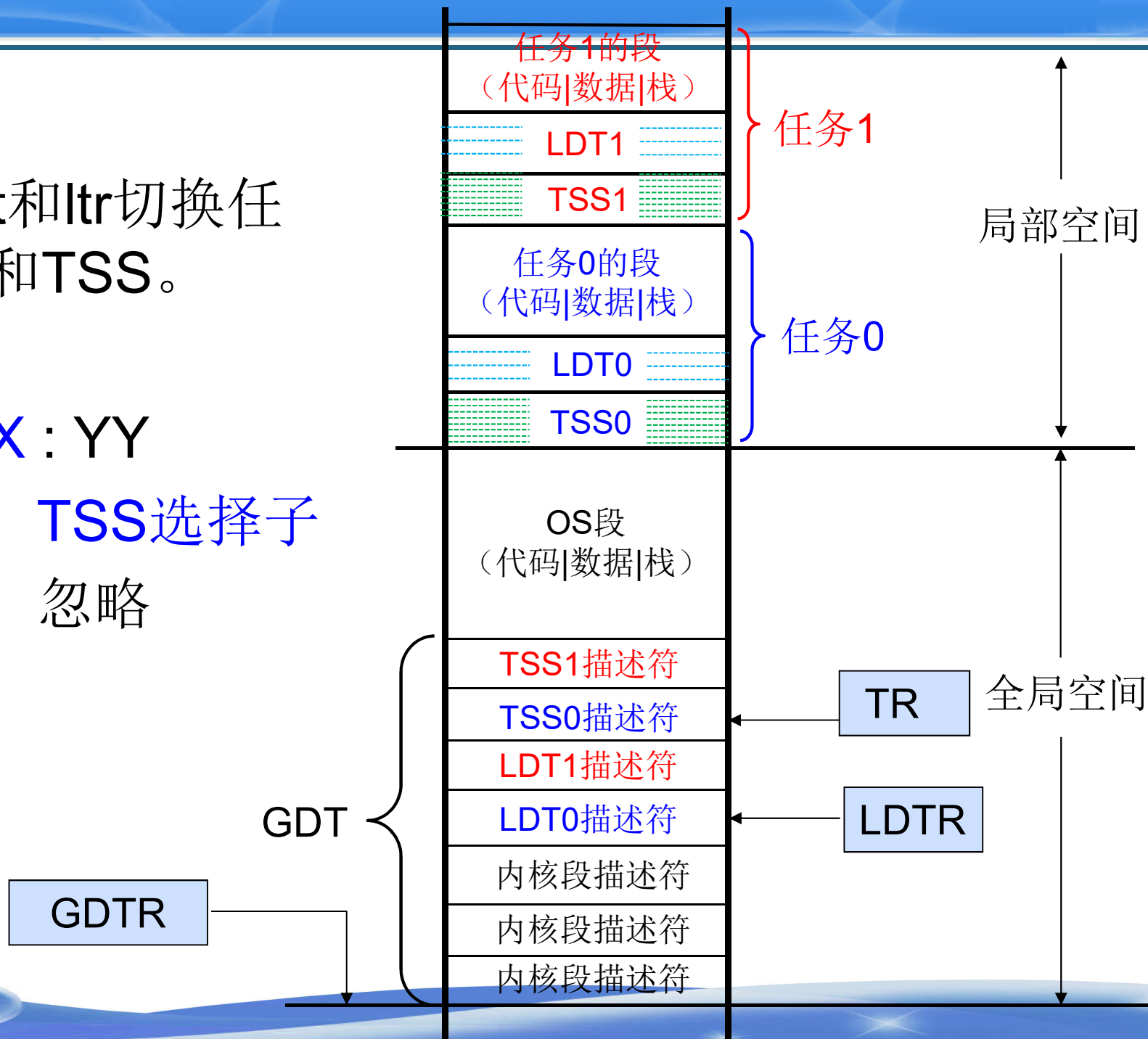
- 使用 `lidt` 和 `ltr` 切换任务 LDT 和 TSS。

- **JMP** 指令

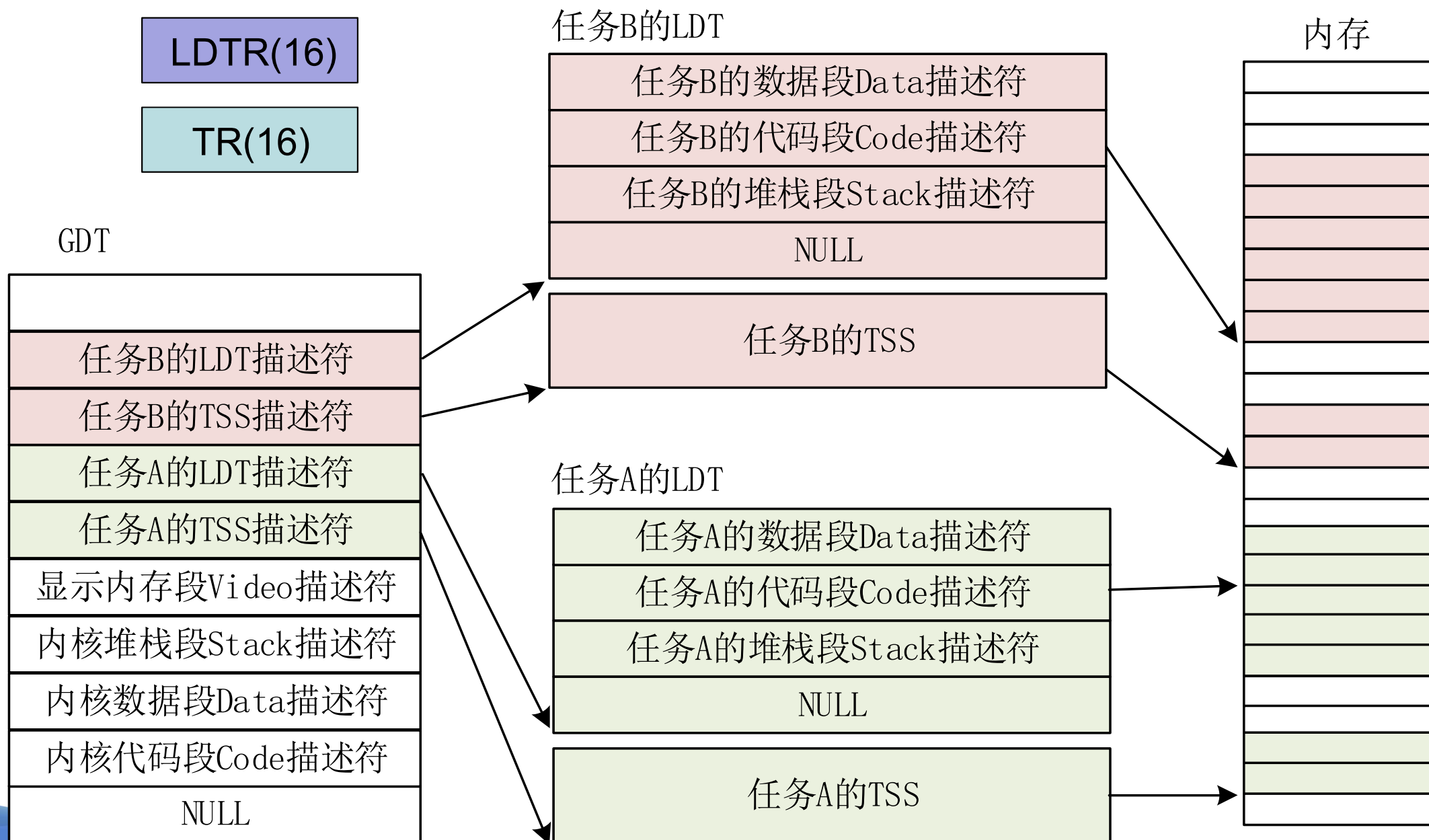
- `JMP XX : YY`

- ◆ `XX`: **TSS** 选择子

- ◆ `YY`: 忽略



任务的切换



任务的切换

- 切换时机

- 强制式任务切换

- ◆ 通过定时器中断服务程序中实施任务切换。

- ◆ 任务获得平等的执行机会

- 协同式任务切换

- ◆ 当前任务主动请求暂时放弃执行权时

- ◆ 当前任务通过调用门请求操作系统服务时

任务的切换

● 切换过程

- (1) 通过任务门完成切换
- (2) 执行JMP或CALL转移到目标TSS描述符
- (3) 执行JMP或CALL转移到目标任务门描述符
- (4) 执行IRET指令（EFLAGS的NT位已置1）