2021级期末考试复习版本

《操作系统原理》

第7章 内存管理

教师：邹德清，李珍，苏曙光
华中科技大学网安学院
2023年10月-2024年01月

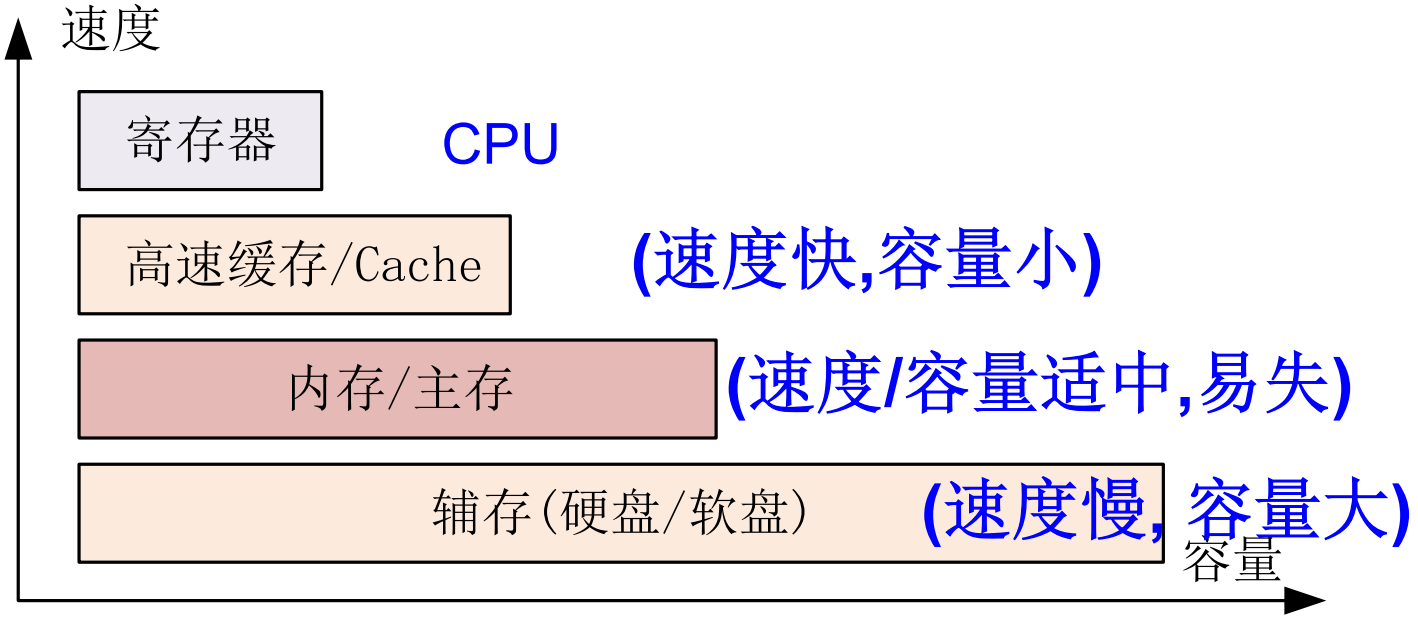
内存管理

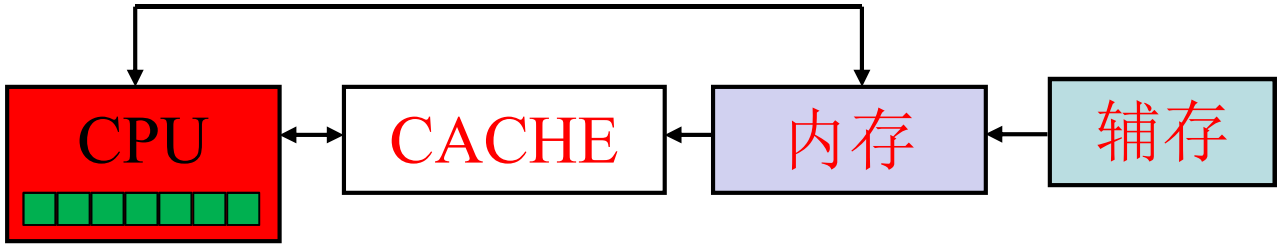
- 主要内容
  - 内存管理的功能
  - 物理内存管理
    - 分区存储管理
    - 覆盖技术
    - 对换技术
  - 虚拟内存管理
    - 页式存储管理
    - 段式存储管理
  - LINUX存储管理
  - 保护模式
    - Intel CPU保护模式
    - 段和页机制
- 重点
  - 地址映射
  - 虚拟内存
  - 页式管理
  - 段式管理
  - 保护模式内存管理机制

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



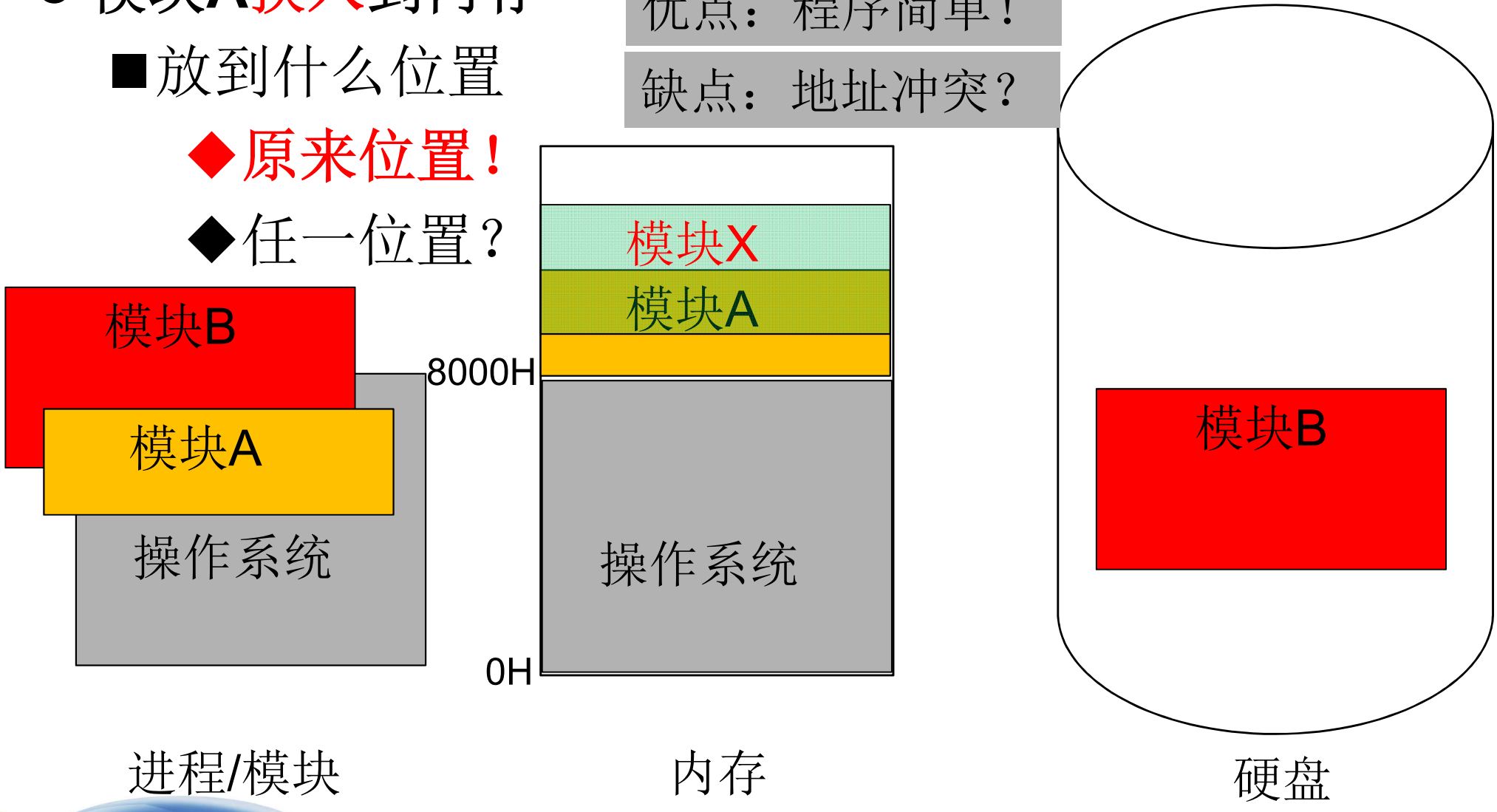


换出与换入的讨论（辅存支援内存）

- 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？

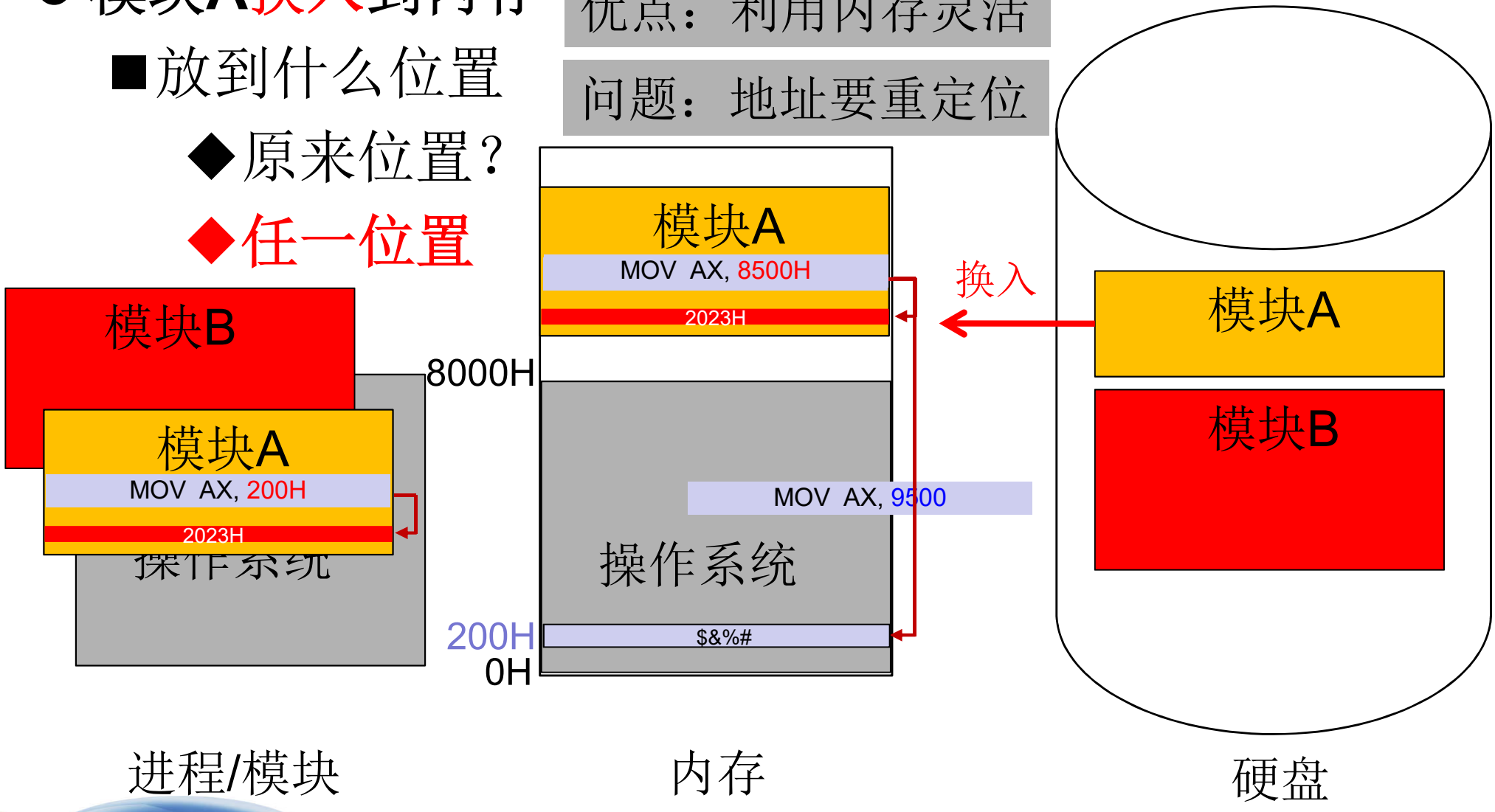


换出与换入的讨论（辅存支援内存）

- 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）

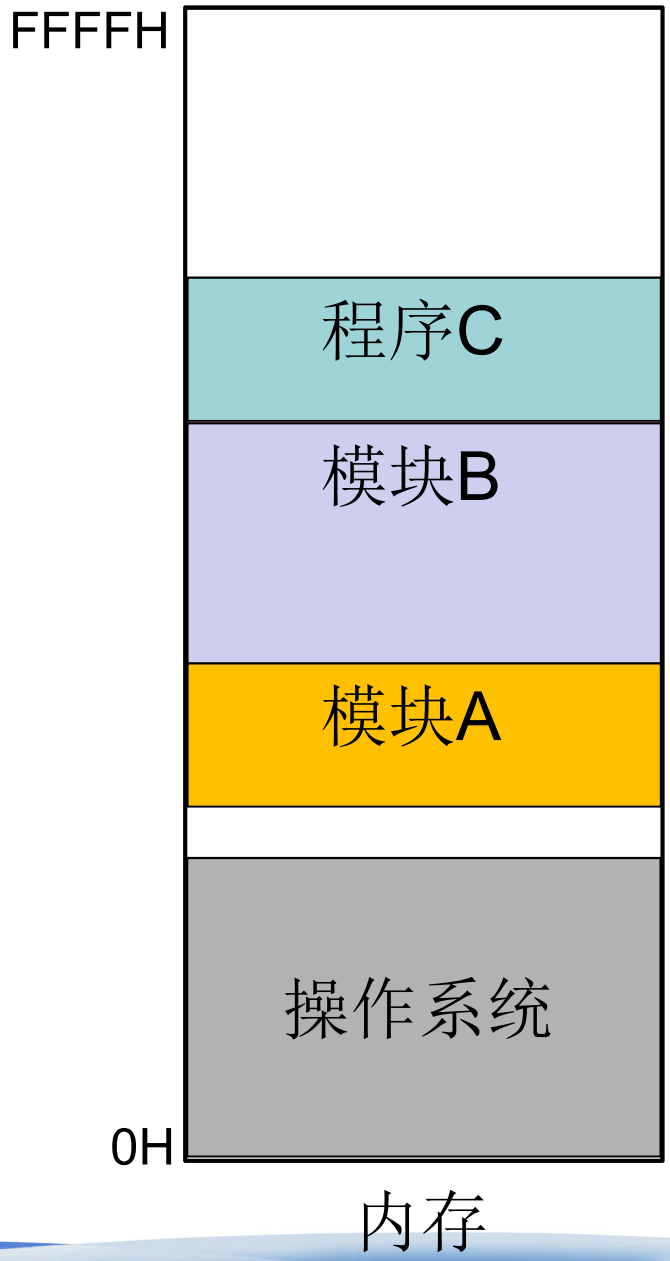
- 地址重定位（地址重映射）
  - 重新确定指令中目标数据的正确地址（更新地址）。
  - 新的值与目标程序块的实际放置位置有关

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）变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 src_path, std::vector<CString>
39  void ShowMatImageInControl(cv::Mat& matImage, UINT n
40  void DrawRectangle(HWND m_HWnd, int x1, int y1, int
```

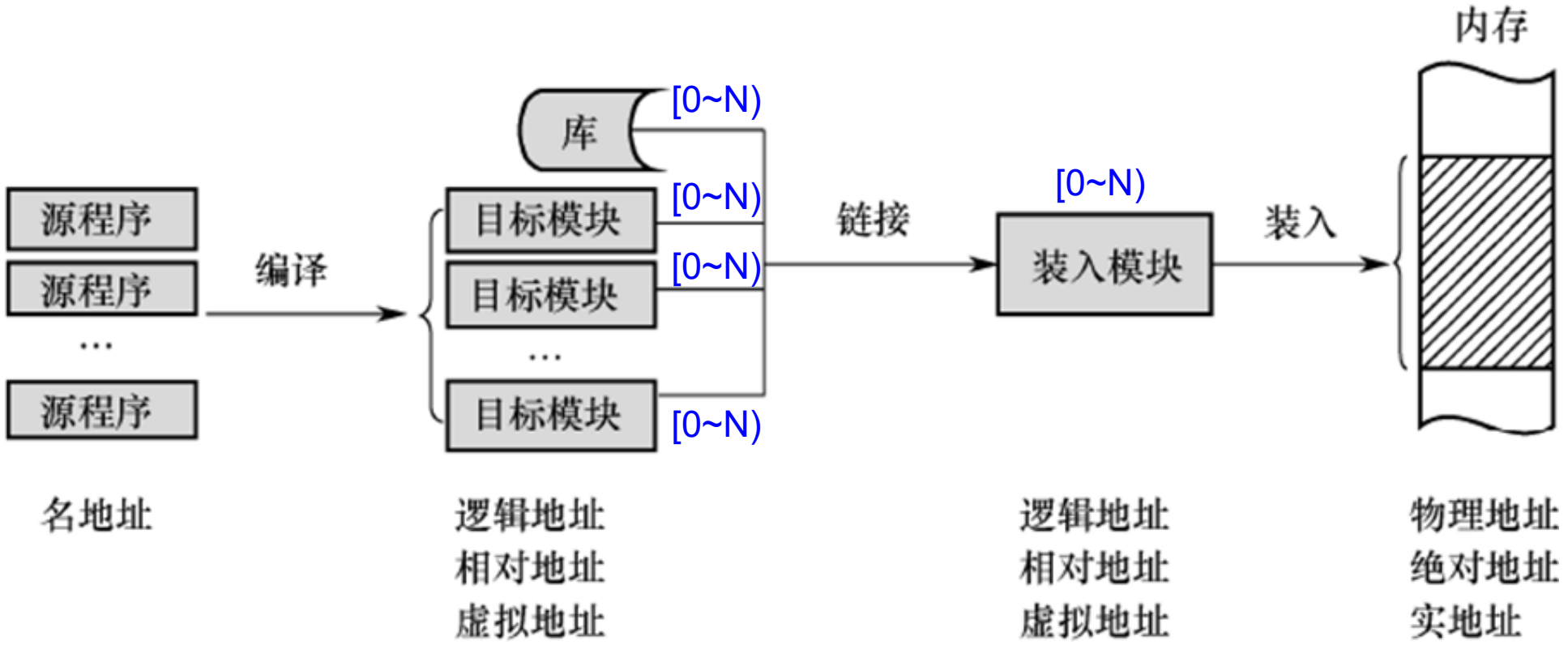
MOV AX, 200H

↓

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~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）变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_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

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 = 装入基址BA + 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_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
```

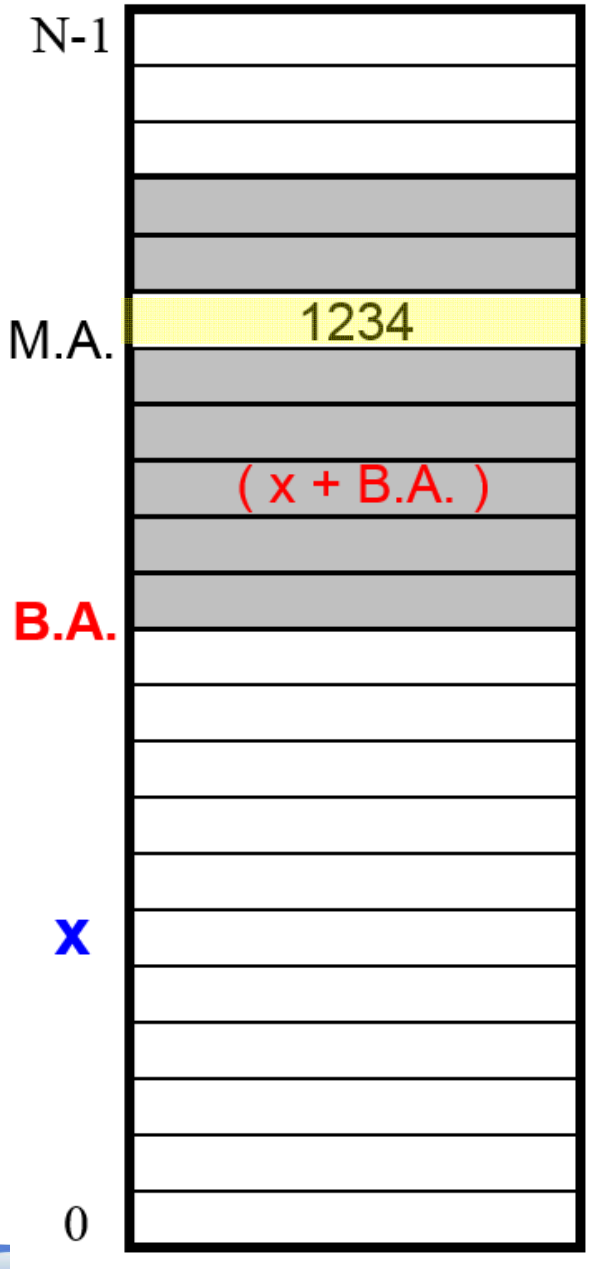
MOV AX, 200H

↓

MOV AX, 8500H

静态地址映射

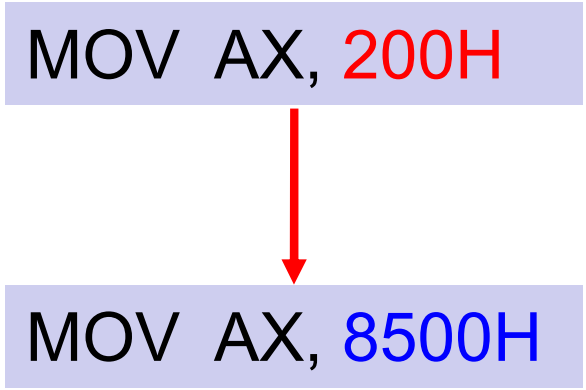
- 特点
  - ■程序运行之前确定映射关系
  - ■程序占用连续的内存空间
  - ■程序装入后不能移动
    - ◆如果移动必须放回原来位置



动态地址映射

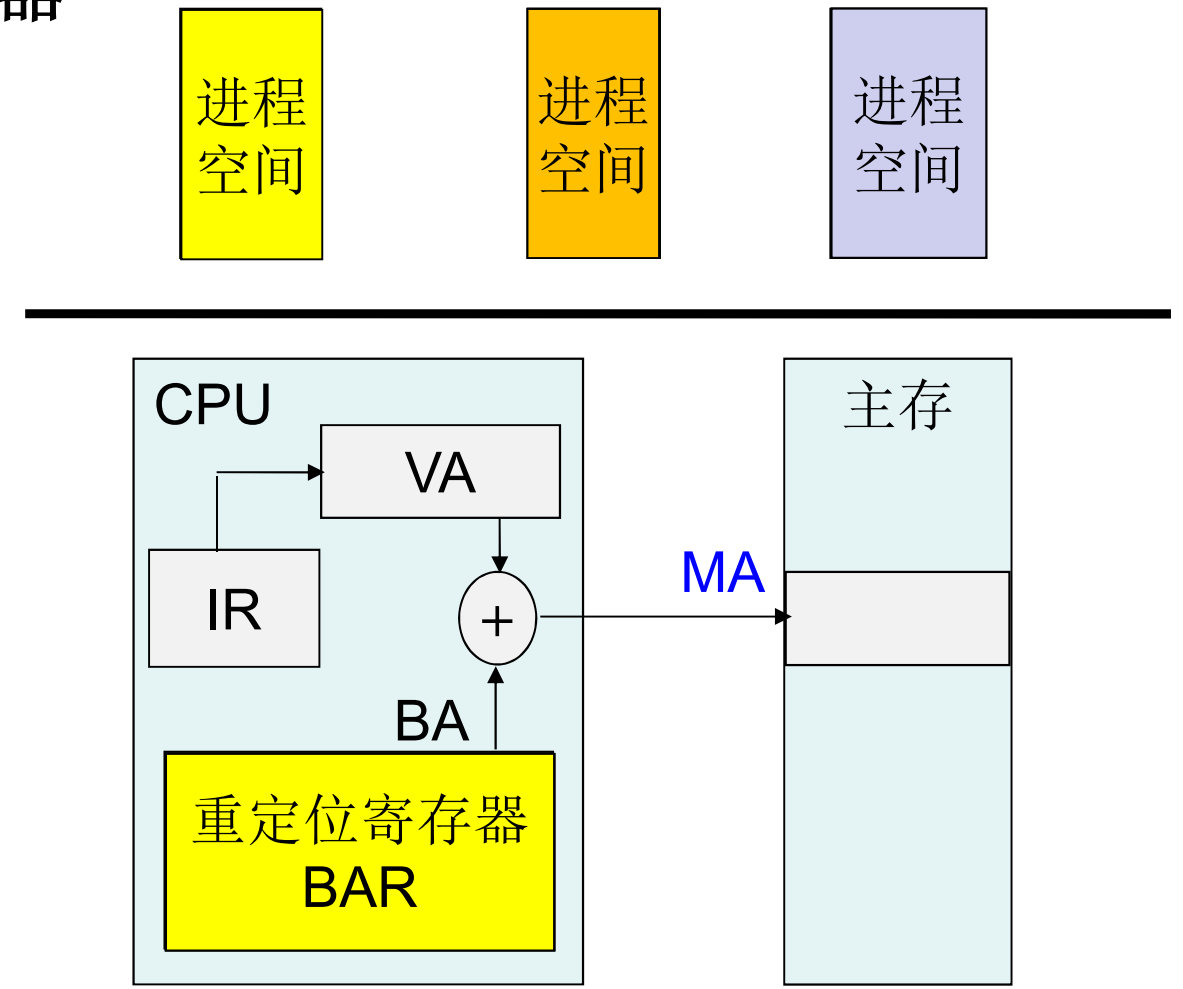
- 定义
  - 在程序执行过程中把逻辑地址转换为物理地址。
  - 映射方式
    - 物理地址MA = 装入基址BA + 虚拟地址VA
    - 装入基址：基址寄存器BAR

```
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_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
```



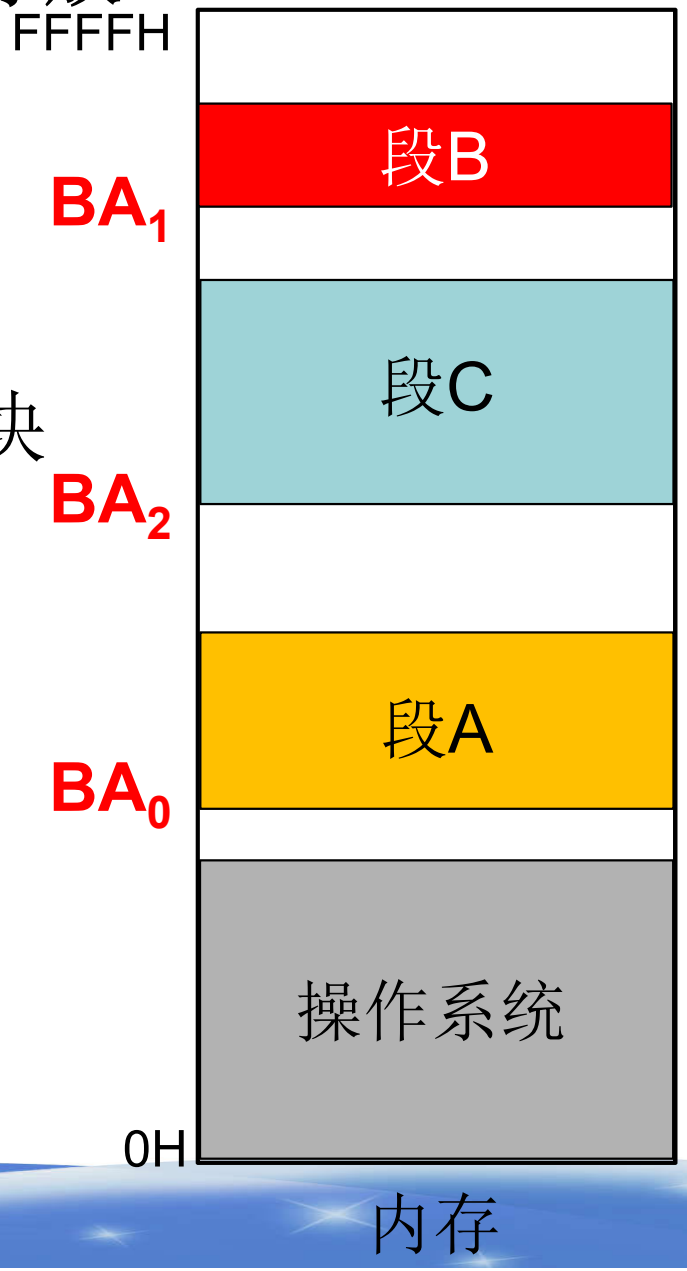
实现动态地址映射的思路

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



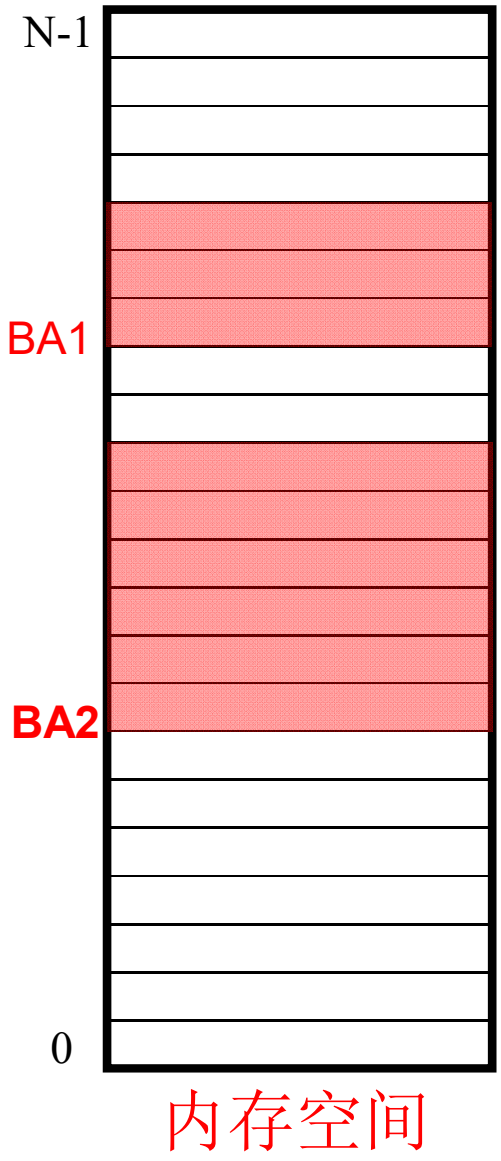
实现动态地址映射的思路

- 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

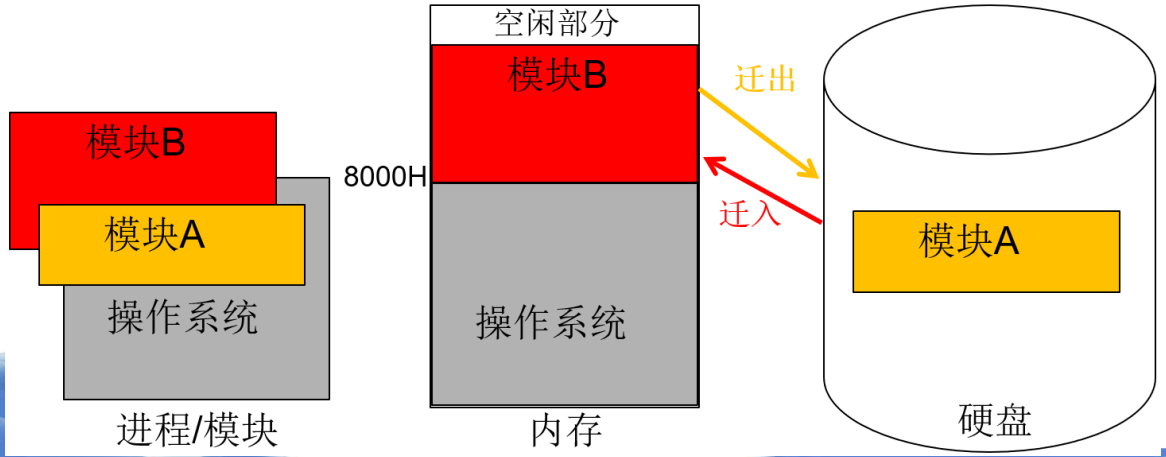


存储管理的功能

- 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+5
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+2
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  WRSPACE  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
```

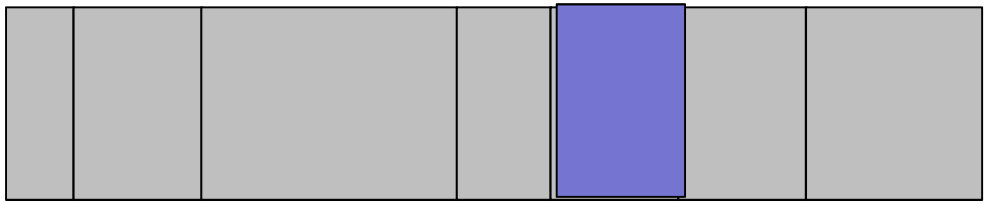
存储管理的功能：2）存储扩充（虚拟存储）

- 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+5
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+2
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  WRSPACE  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

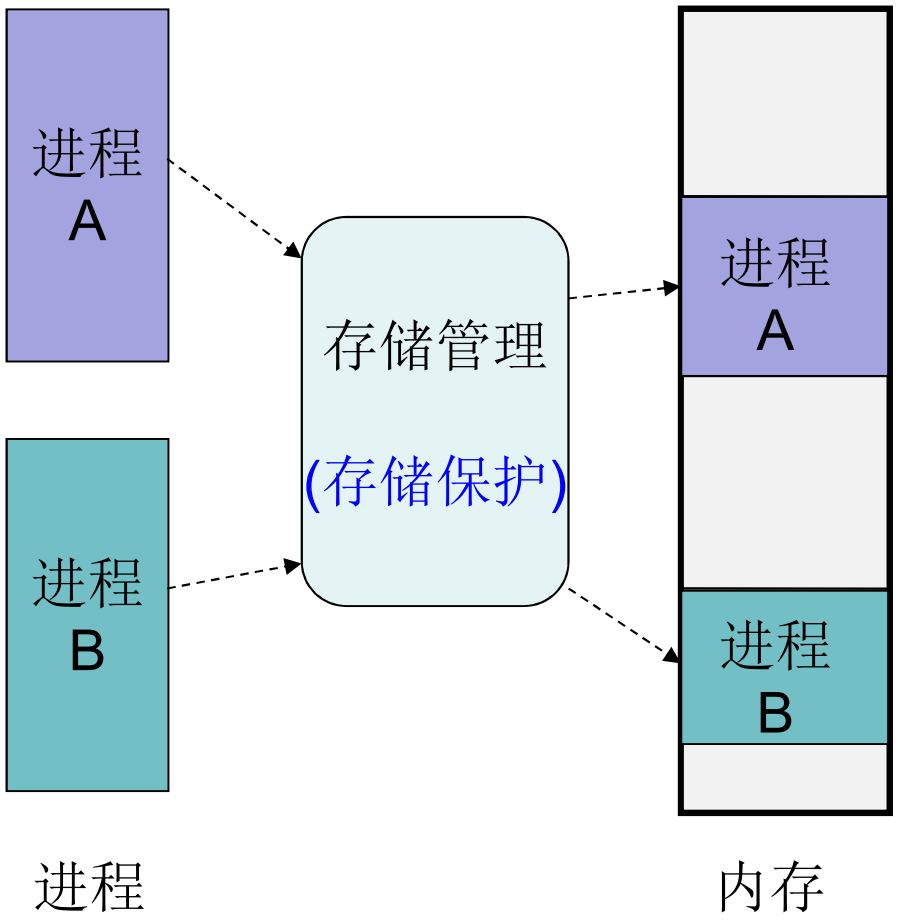
- 为程序运行分配足够的内存空间
  - 代码 | 数据 | 堆栈
- 需要解决的问题
  - 放置策略
    - 程序调入到内存哪个/哪些区域
  - 调入策略
    - 何时把要运行的程序调入内存？
    - 预调策略 | 请调策略
  - 淘汰策略
    - 迁出（/淘汰）哪些程序以腾出内存空间。



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

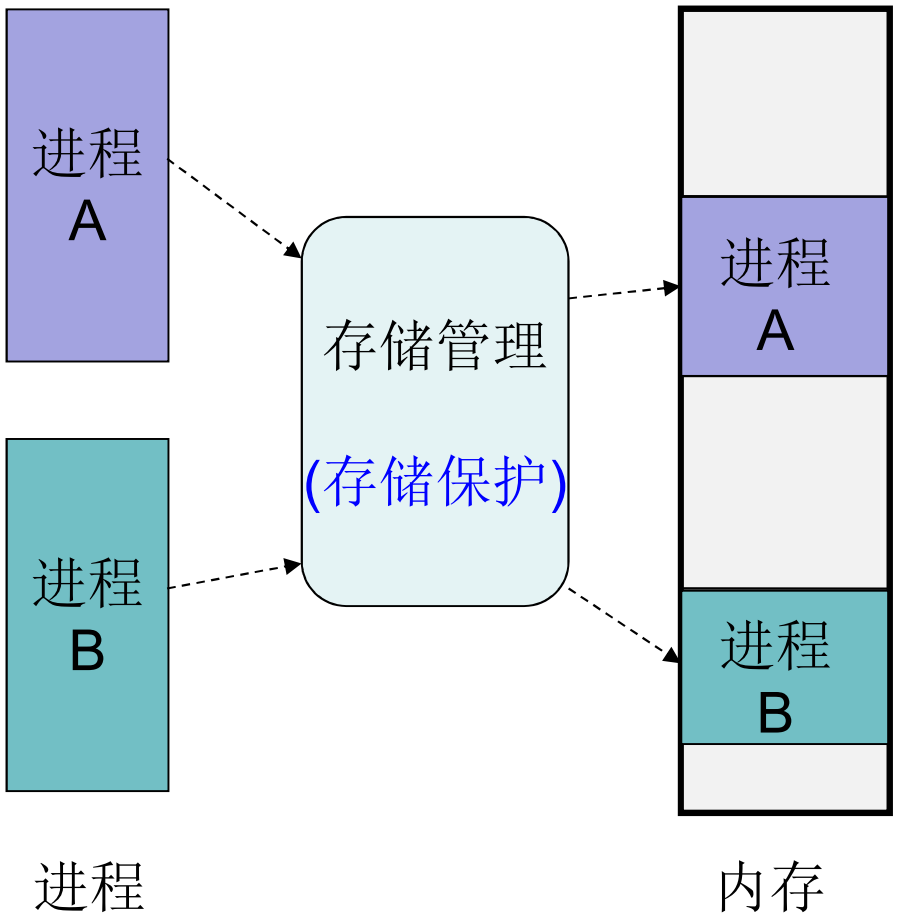


存储管理的功能：4）存储保护功能

- 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

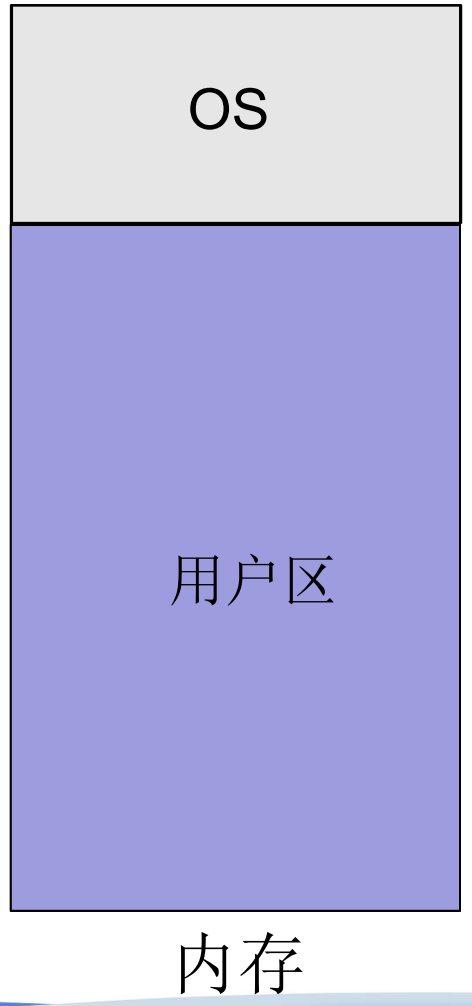
- 存储保护
  - 保证内存中的多道程序只能在给定区域活动，并且互不干扰。
    - 防止访问越界
    - 防止访问越权
  - 方法
    - 界址寄存器
    - 存储键保护
    - 段式虚拟内存管理
    - 页式虚拟内存管理



2. 物理内存管理

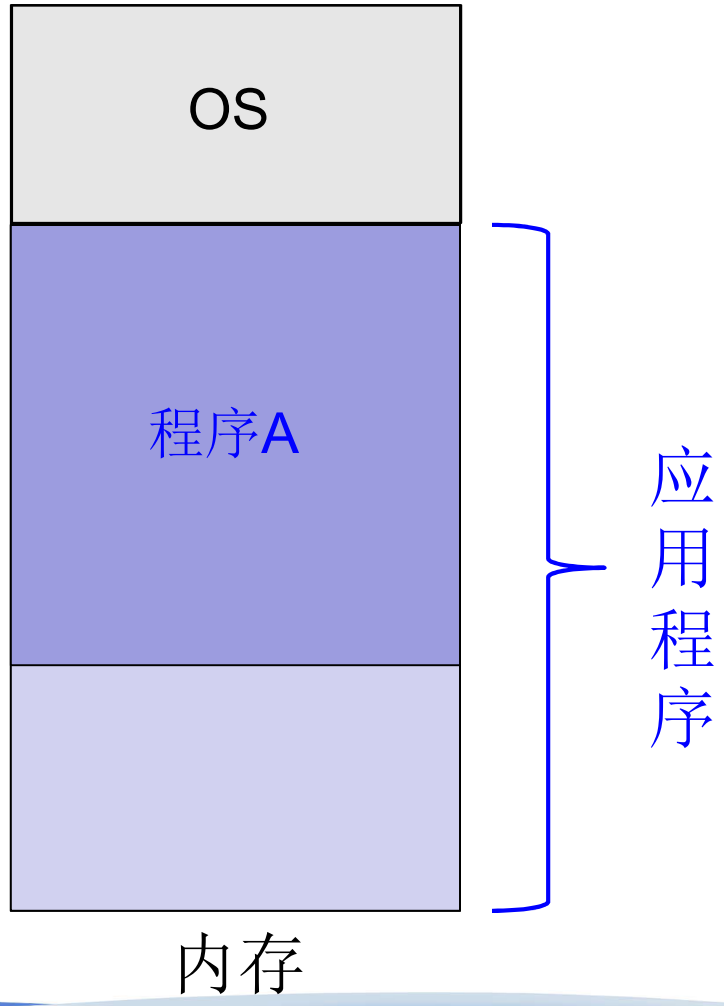
物理内存管理

- 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



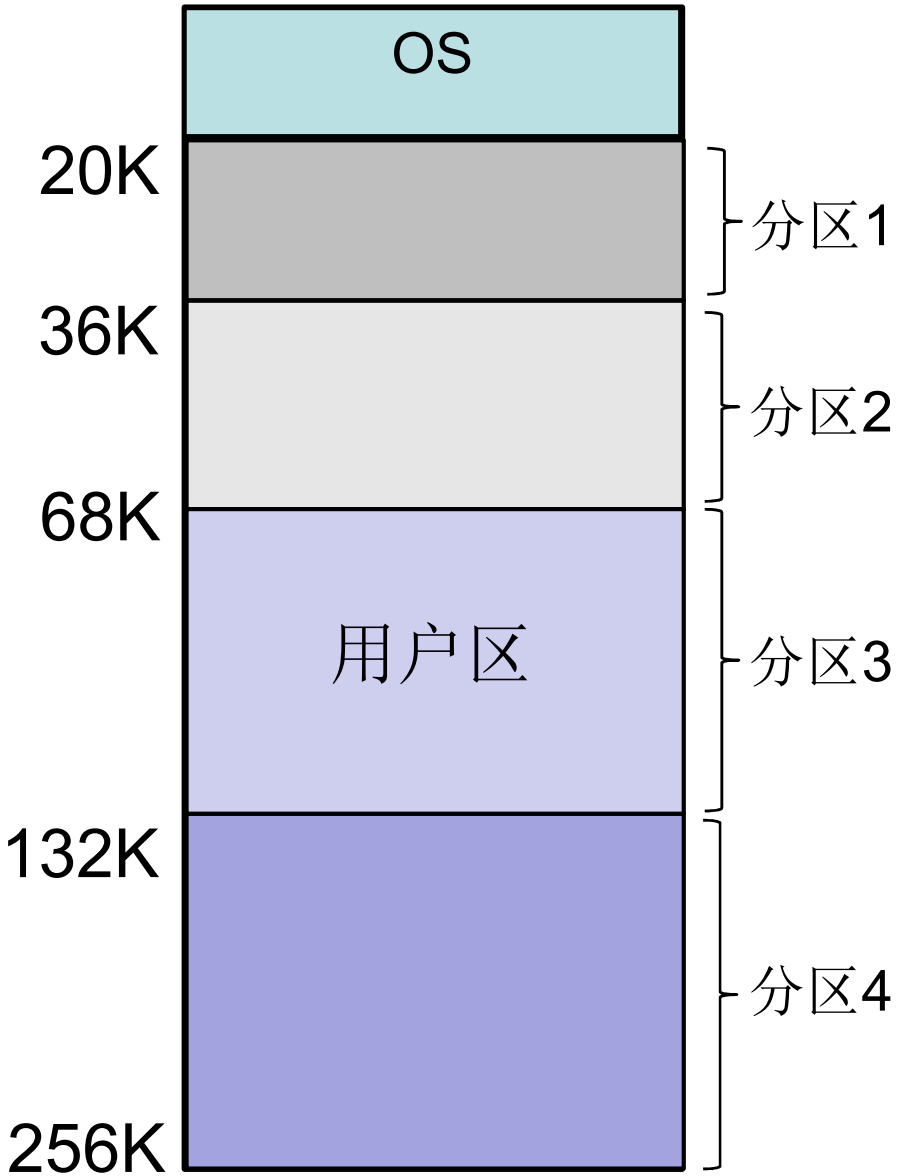
单一区存储管理（不分区存储管理）

- 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

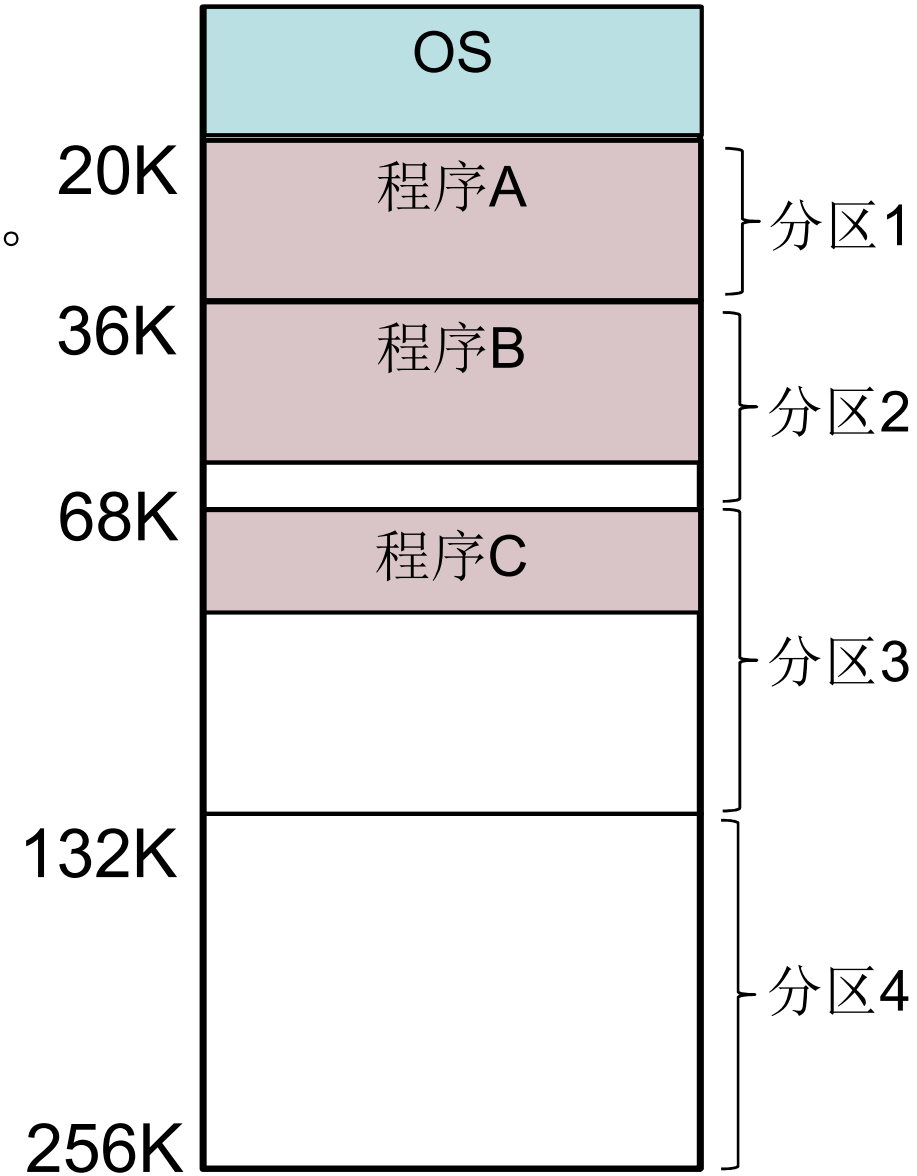
- 定义
  - 把用户区分为若干大小不等的分区，供不同程序使用。
- 分类
  - 固定分区
    - 系统初始化时分区
  - 动态分区
    - 程序装入时临时分区



固定分区

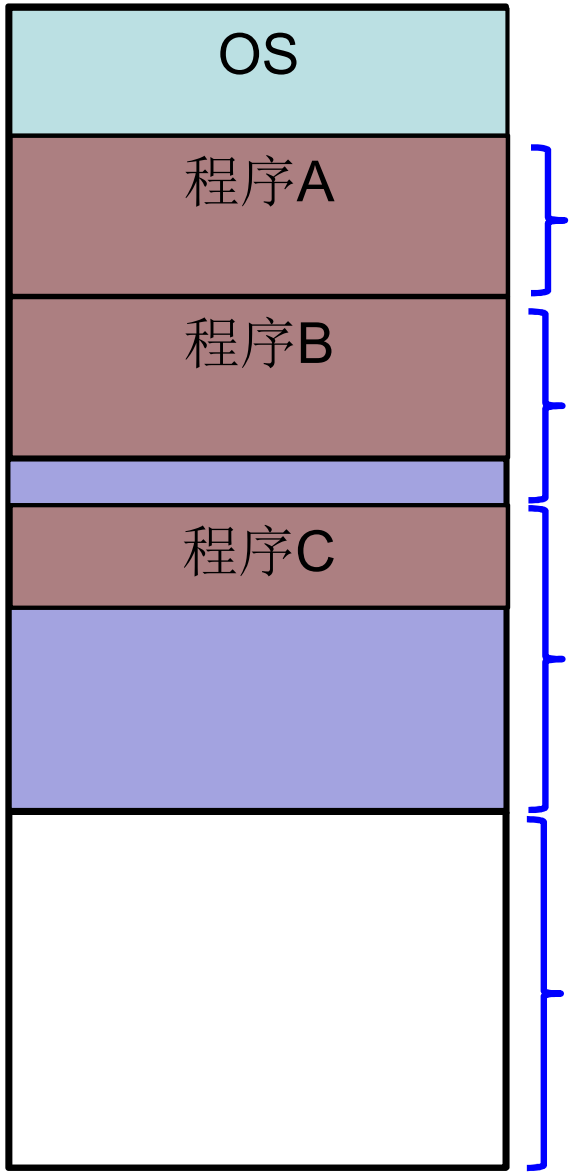
- 固定分区的特点
  - 运行时分区的大小和位置不变
  - 分区大小不同，适应不同程序需求。
- 分区表
  - 记录分区的位置、大小和占用标志

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



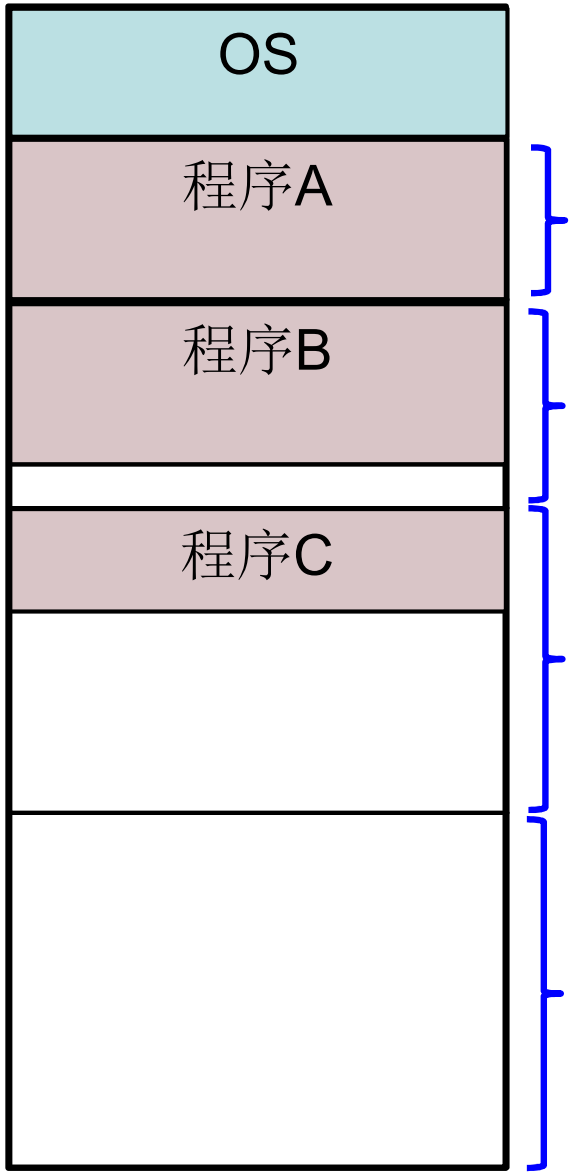
固定分区

- 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。



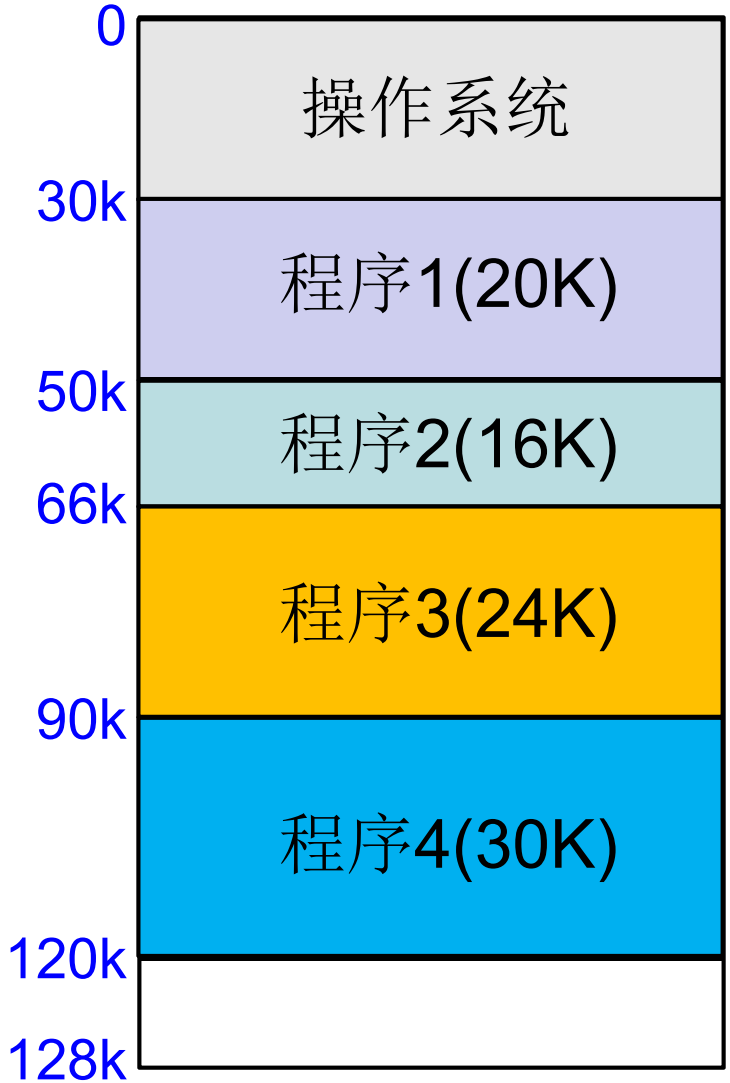
固定分区

- 例子
  - IBM OS/360
    - 多道程序系统
      - 15个分区
      - ≤15道程序



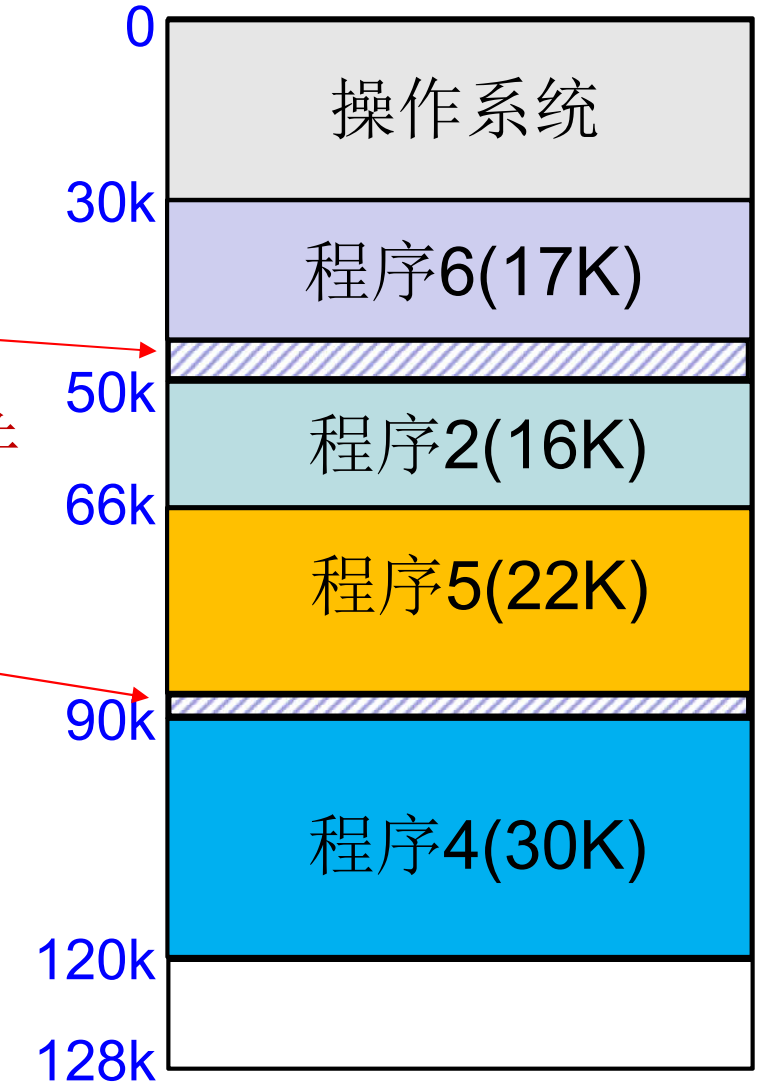
动态分区

- 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）。



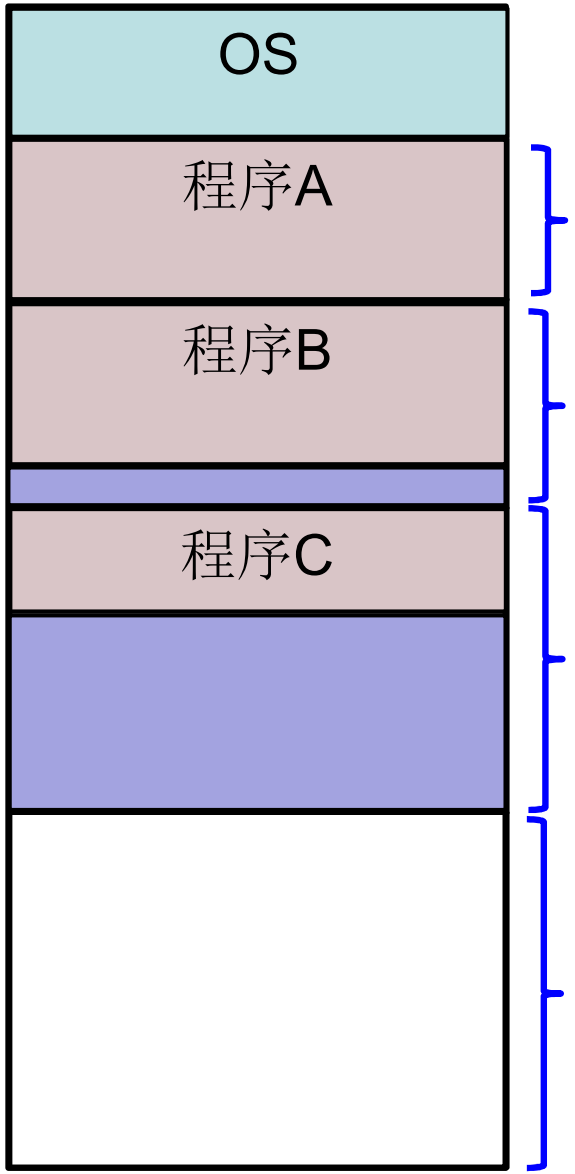
动态分区

- 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



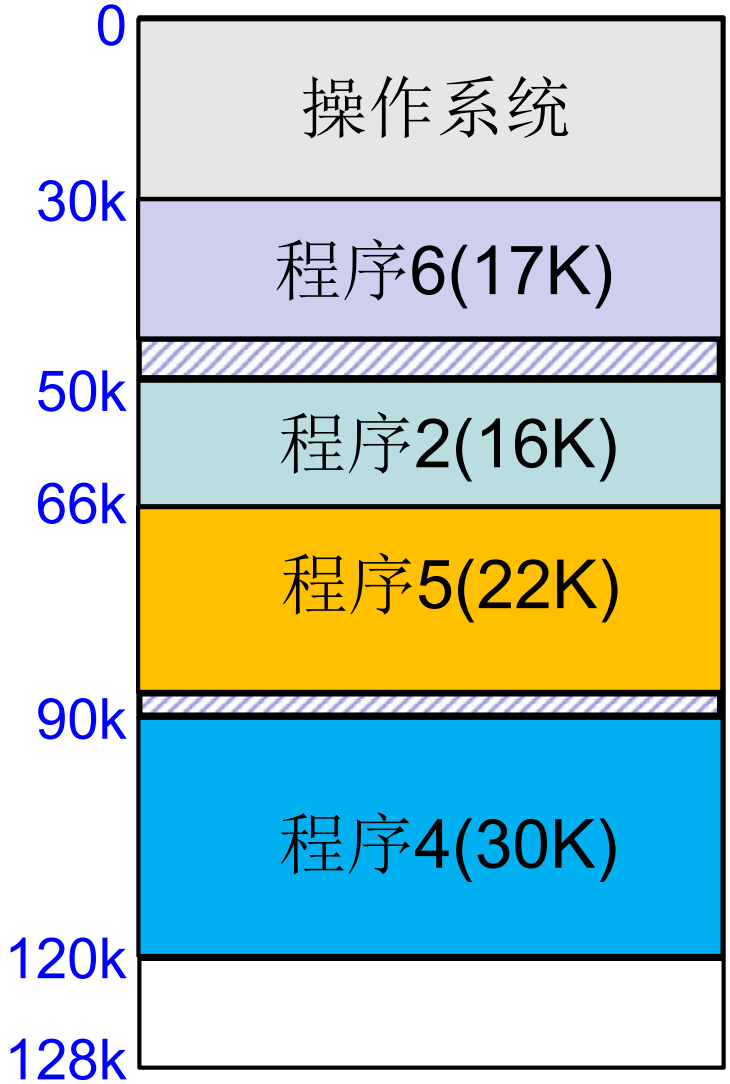
内存碎片

- **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



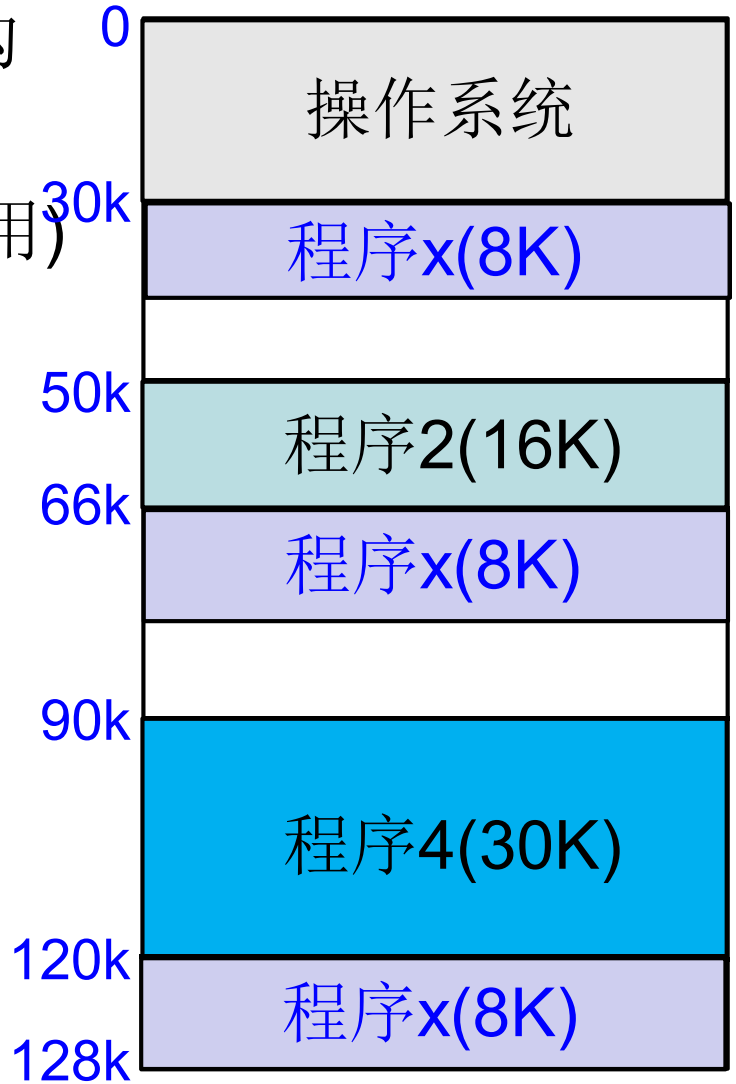
动态分区的特点

- 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

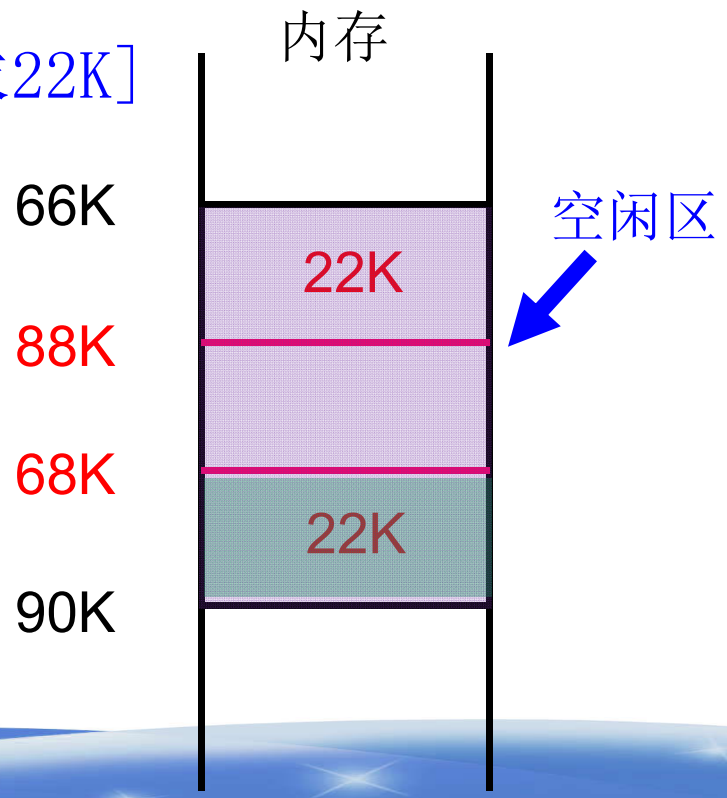
分区的分配

- 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

位置	大小
30K	20K
66K	24K
120K	8K

[例：用户要求22K]



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

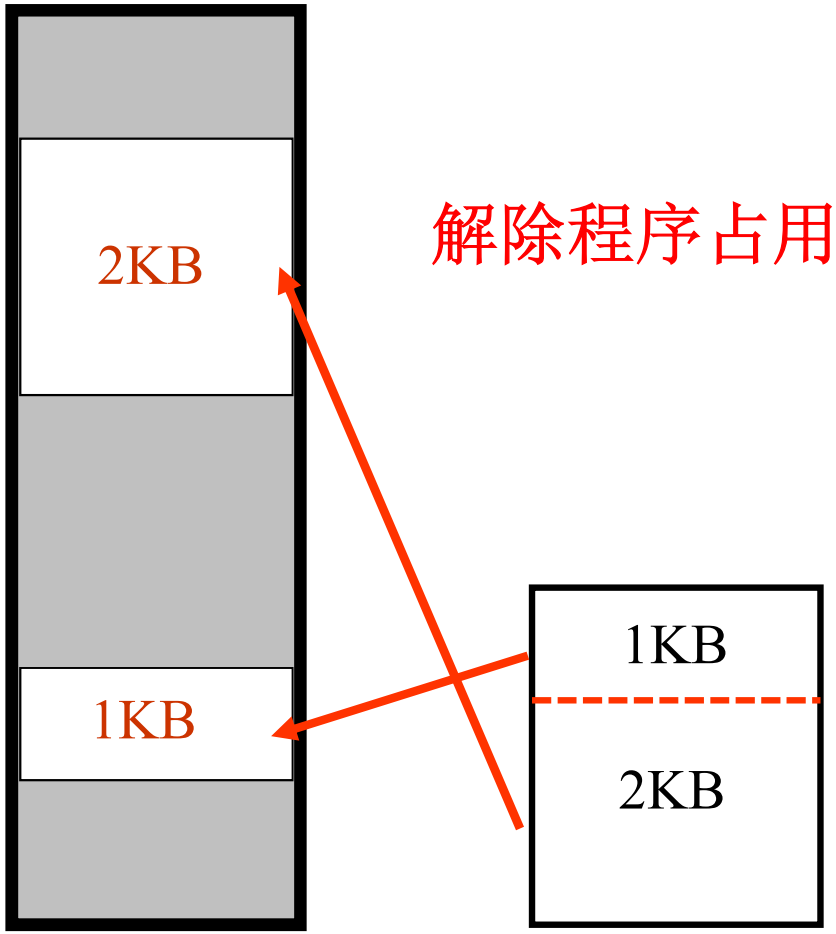
碎片问题

- 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

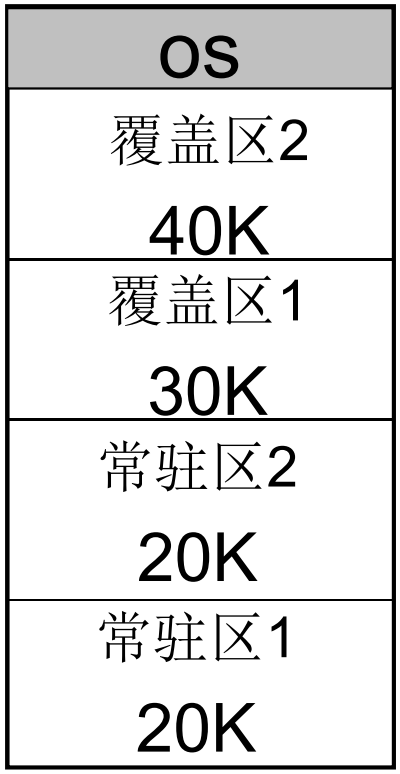


解除程序占用连续内存的限制。

3KB的进程

覆盖——Overlay

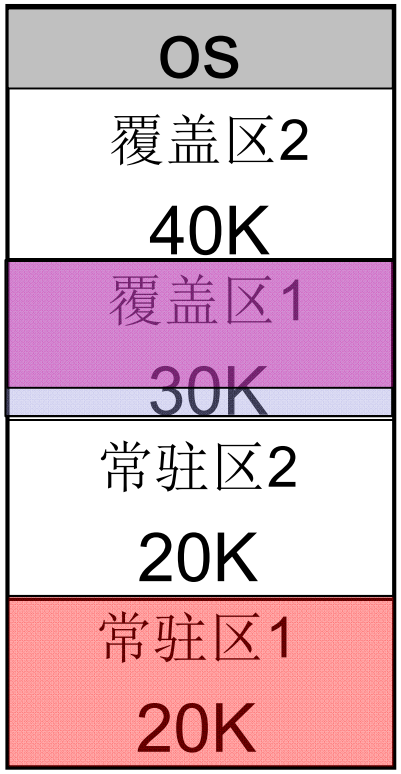
- 目的
  - 在较小的内存空间中运行较大的程序
- 内存分区
  - 常驻区：被某段单独且固定地占用的区域，可划分多个
  - 覆盖区： 能被多段共用（覆盖）的区域， 可划分多个



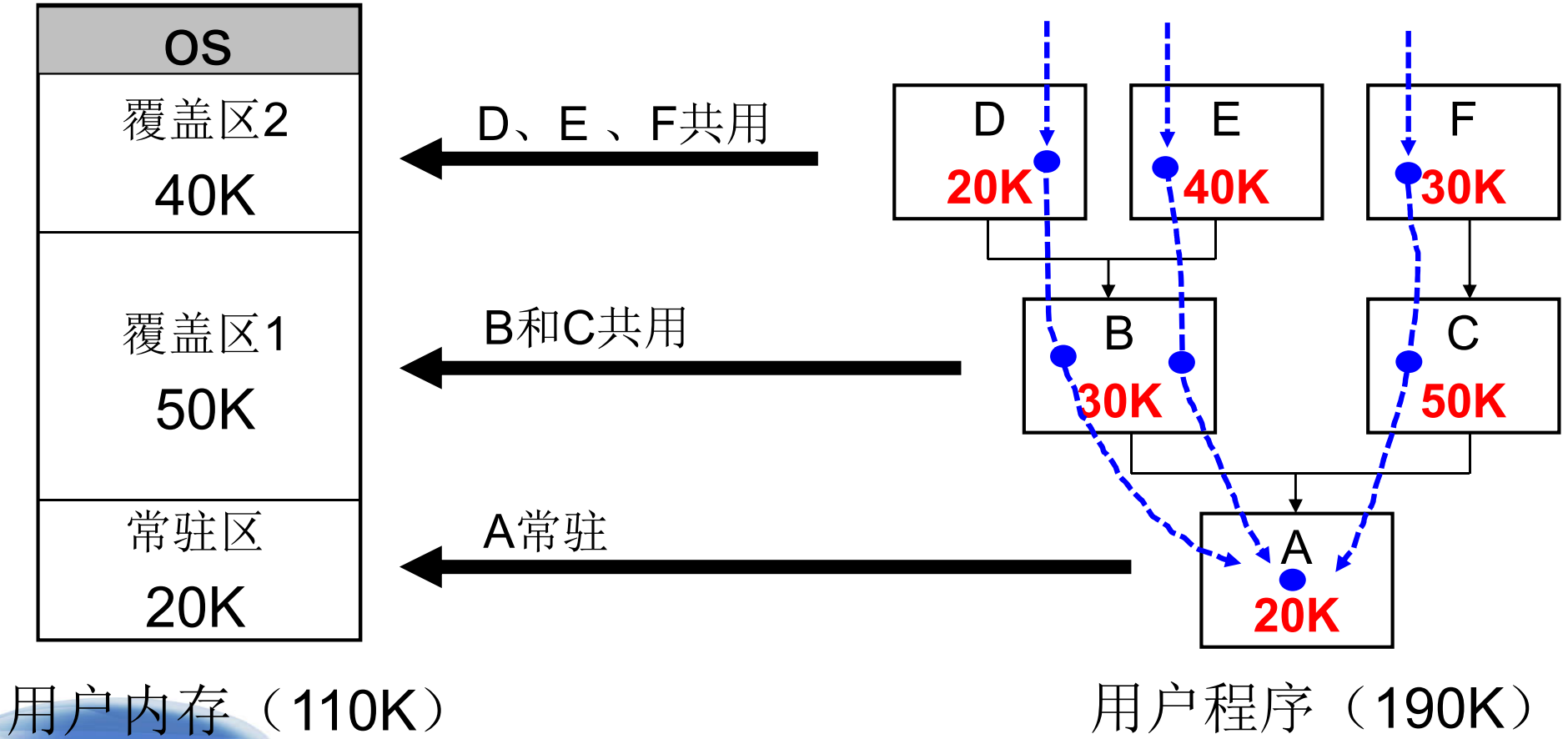
用户内存（110K）

覆盖——Overlay

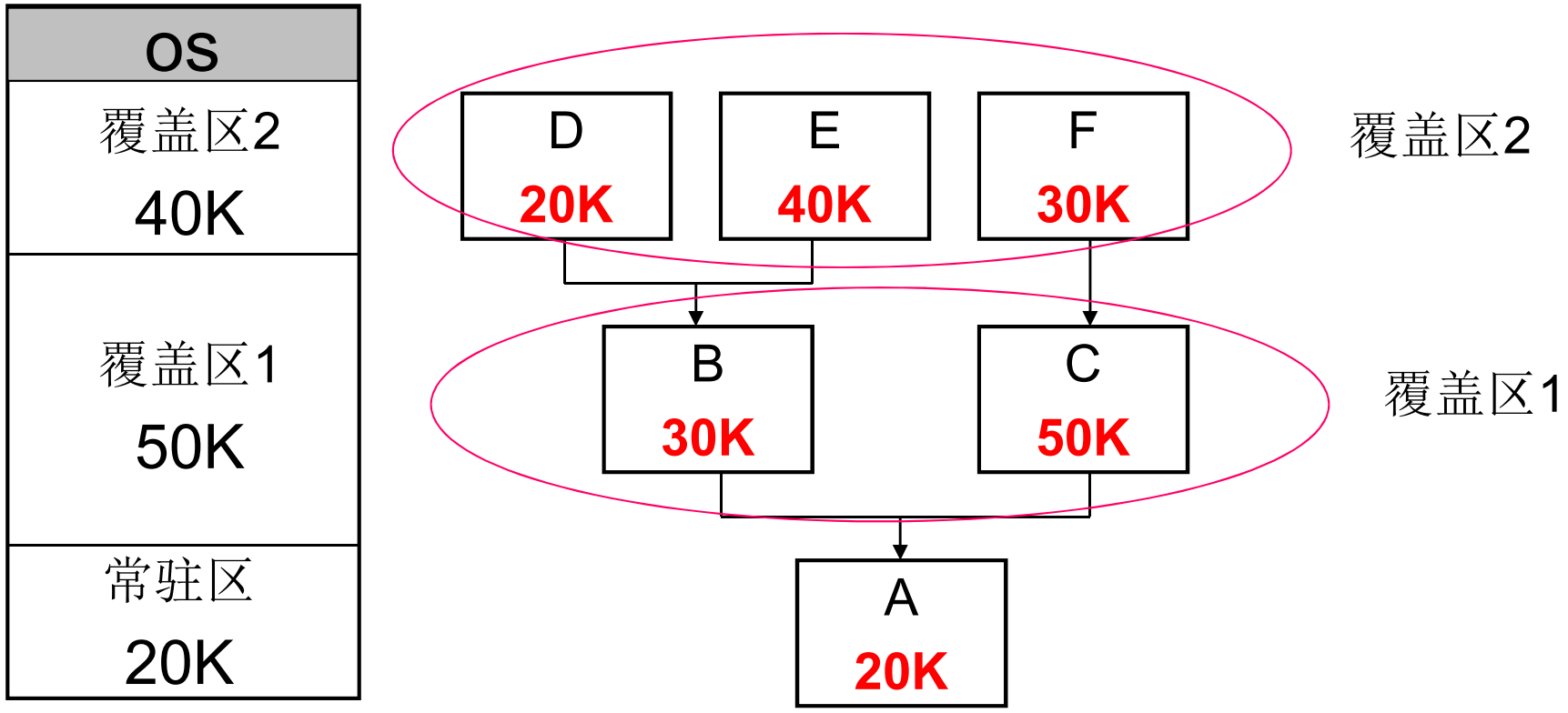
- 工作原理
  - 程序分成若干代码段或数据段
  - 将程序常用的段装入常驻区；（核心段）
  - 将程序不常用段装入覆盖区；
    - 正运行的段处于覆盖区；
    - 暂时不运行的段放在硬盘中(覆盖文件)；
    - 即将运行的段装入覆盖区（覆盖旧内容）；
- 意义
  - 减少程序对内存需求



- 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）

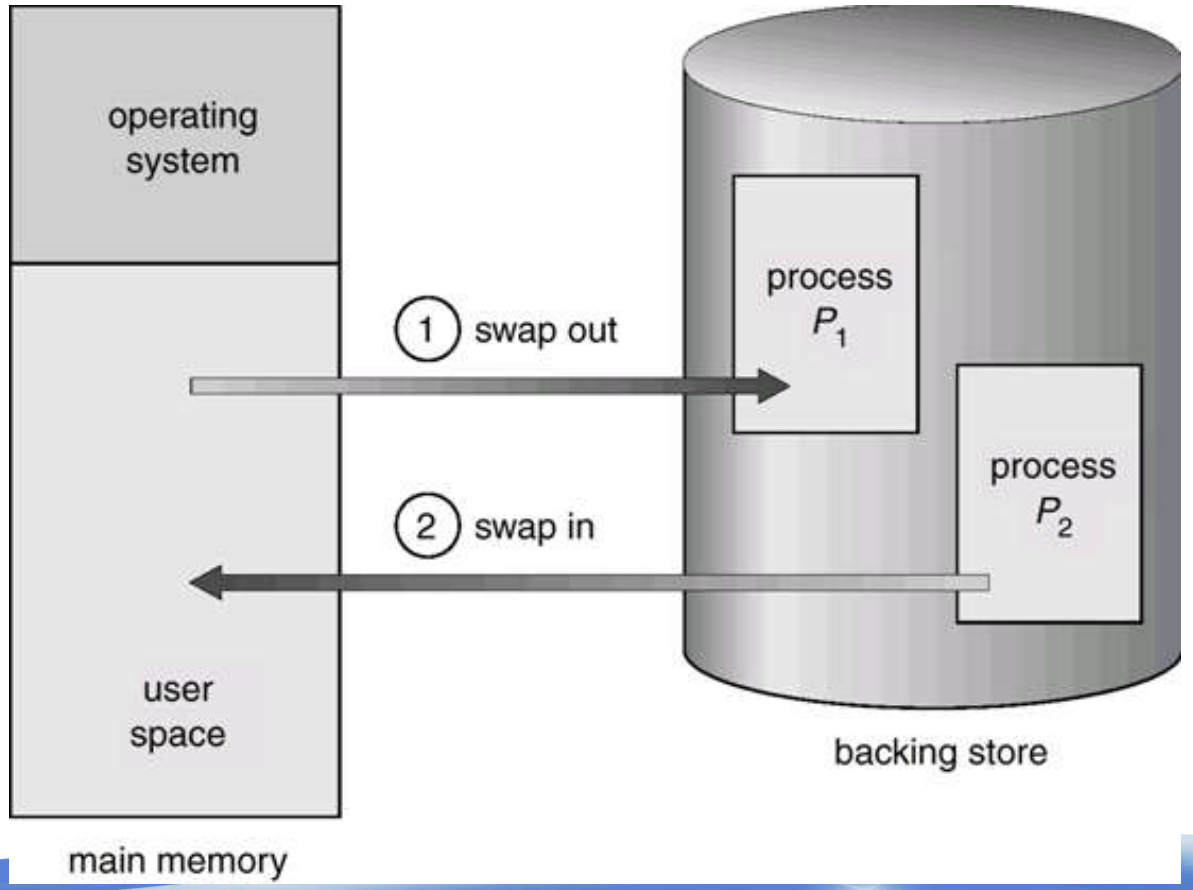


- 覆盖的缺点
  - ■编程复杂：程序员划分程序模块并确定覆盖关系。
  - ■程序执行时间长：从外存装入内存耗时



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。

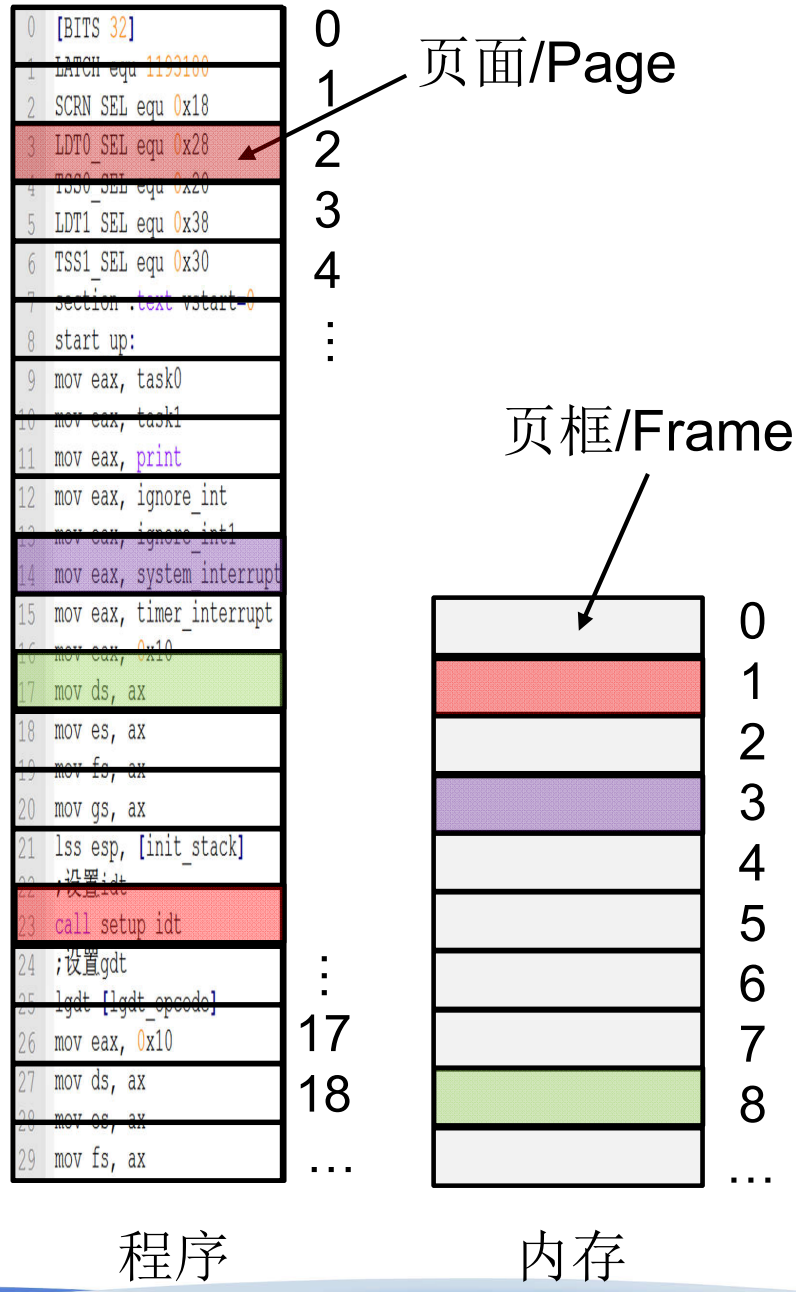


- 对换技术的缺点
  - 换入和换出增加CPU开销；
  - 对换单位太大（整个进程）
- 需要考虑的问题
  - 程序换入时的地址重定位
  - 减少对换传送的信息量
  - 外存对换空间的管理方法
- 采用交换技术的OS
  - **UNIX, Linux, Windows**

7.4 虚拟内存管理

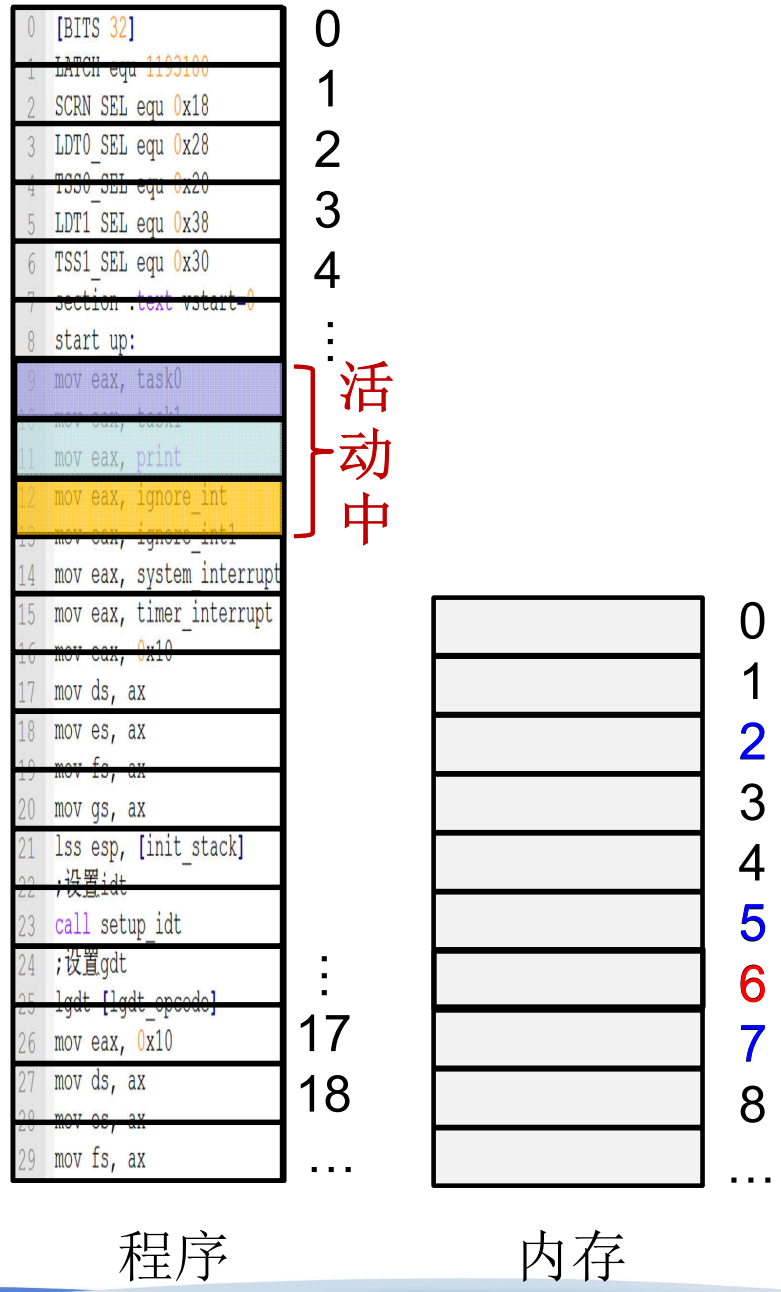
页式内存管理的概念

- 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



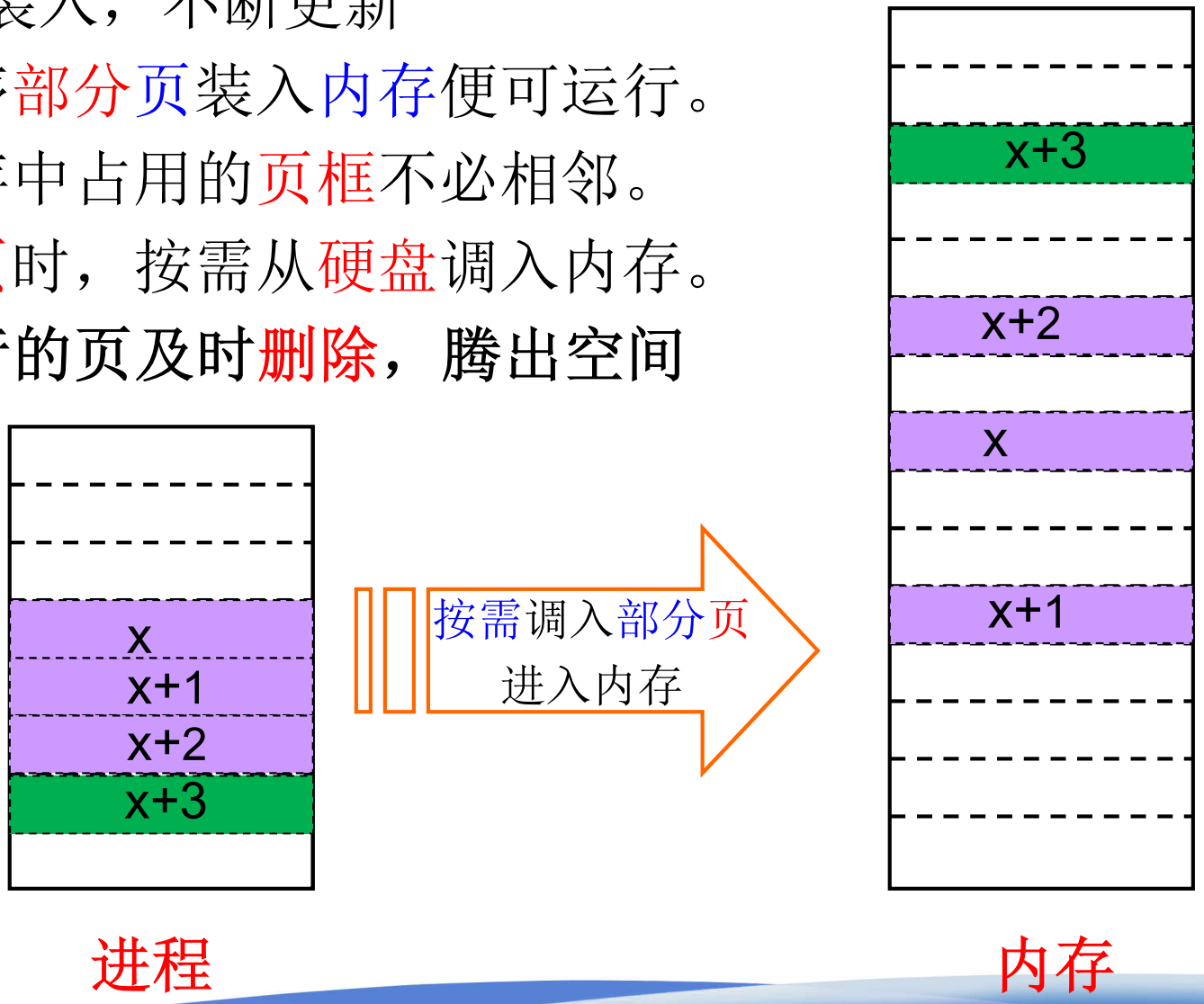
页式内存管理的原理

- 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。



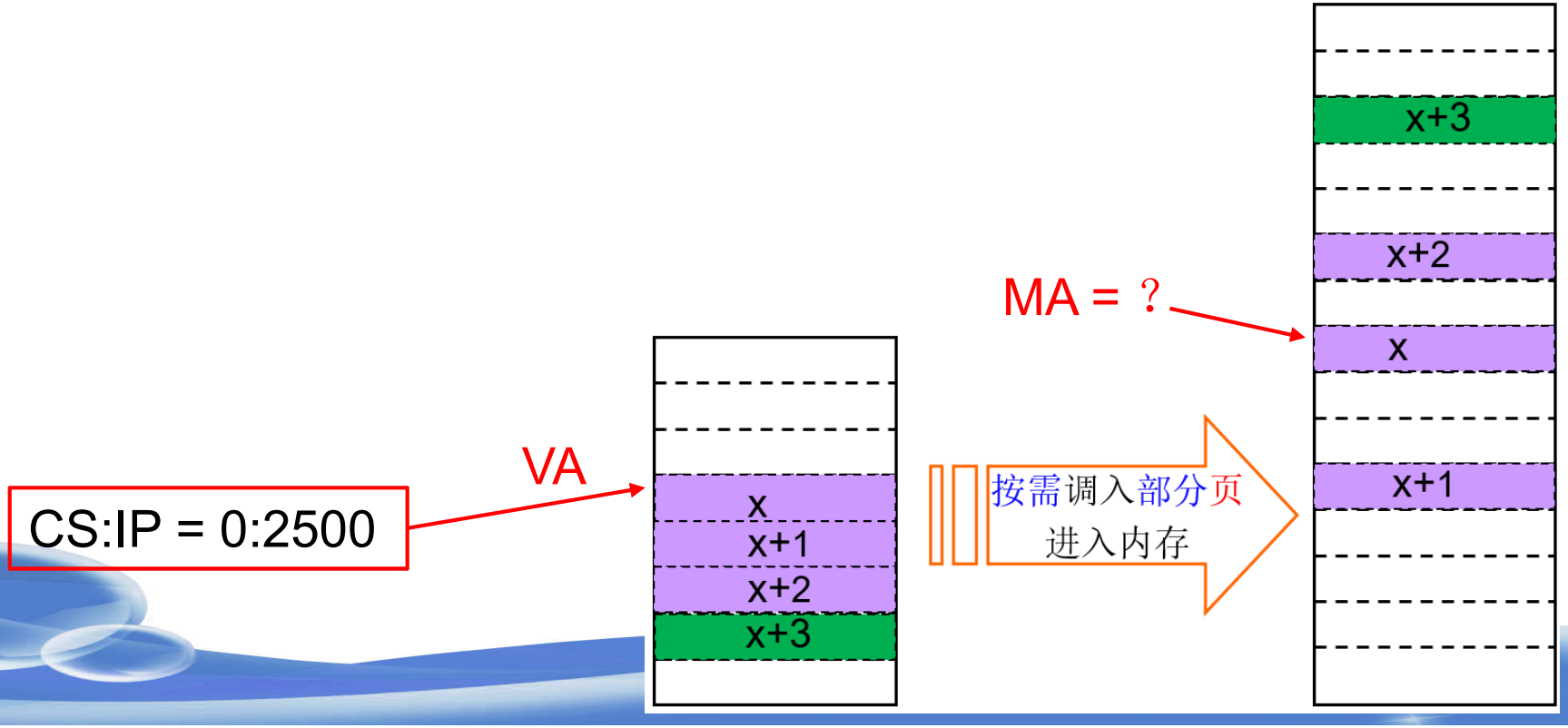
页式虚拟存储管理

- 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



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 / 页大小
  - 页内偏移(W)
    - 所处页内的偏移 =VA%页大小
- 例子
  - VA = 2500; 页面大小1KB

P = 2500 / 1024 = 2

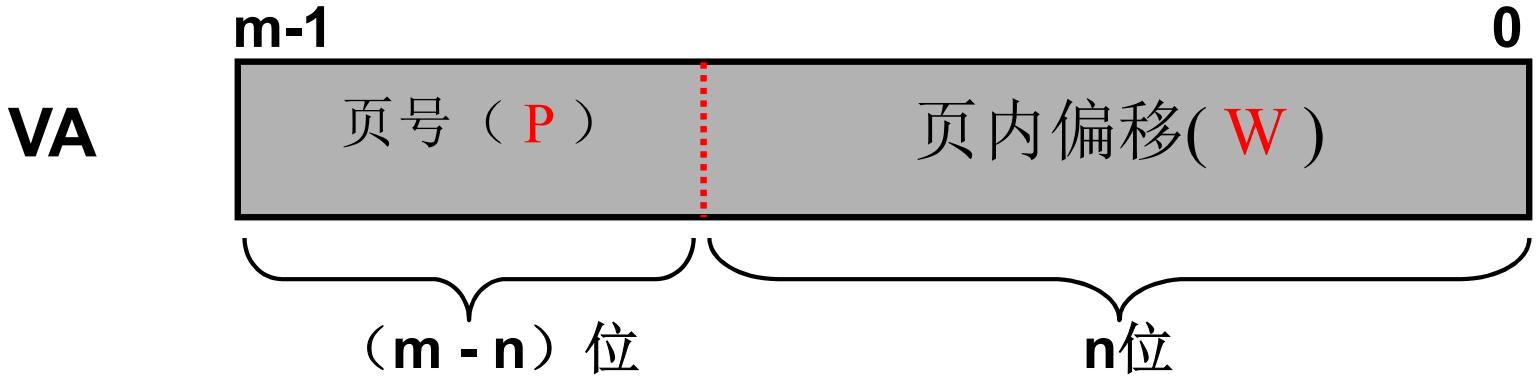
W = 2500 % 1024 = 452

VA

```
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 >> n = VA / 2^n
  - 页内偏移W = VA低 n 位 = VA && (2^n-1) = VA % 2^n

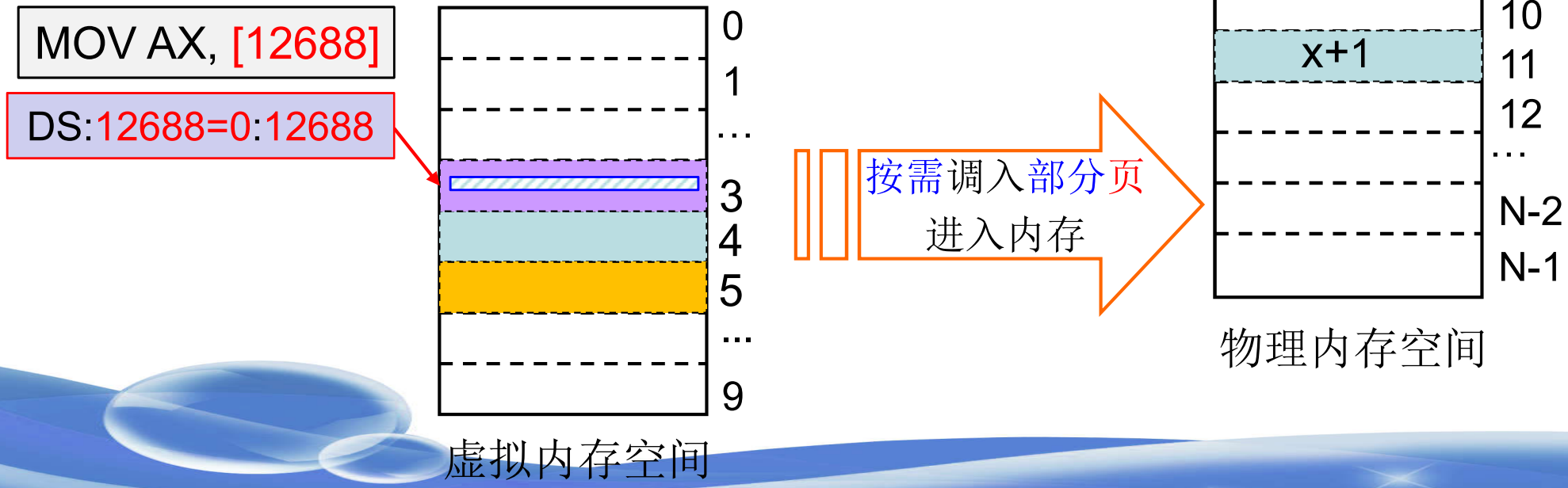


虚拟地址的宽度：m位

地址映射过程

- 页面映射表/页表

页号	页框号
…	…
3	8
4	11
5	6
…	…



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

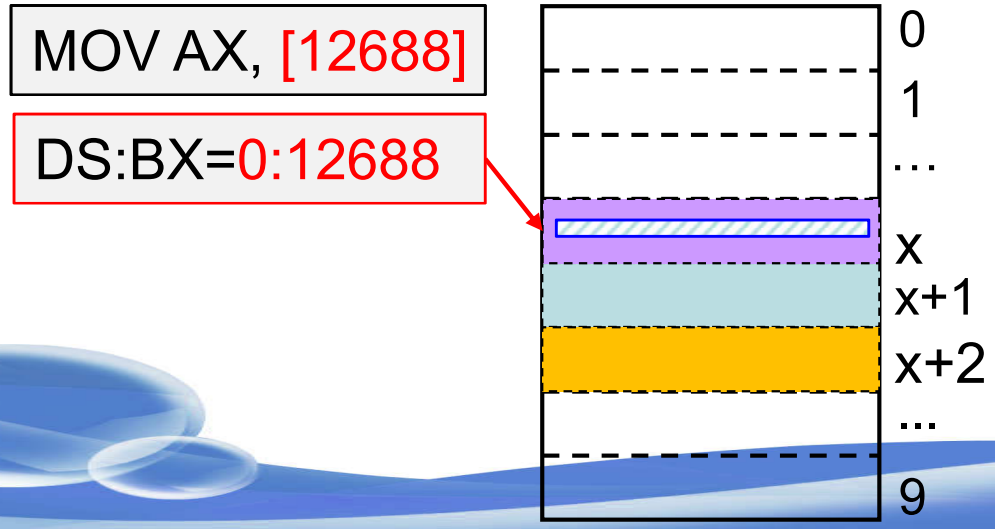
- 地址映射

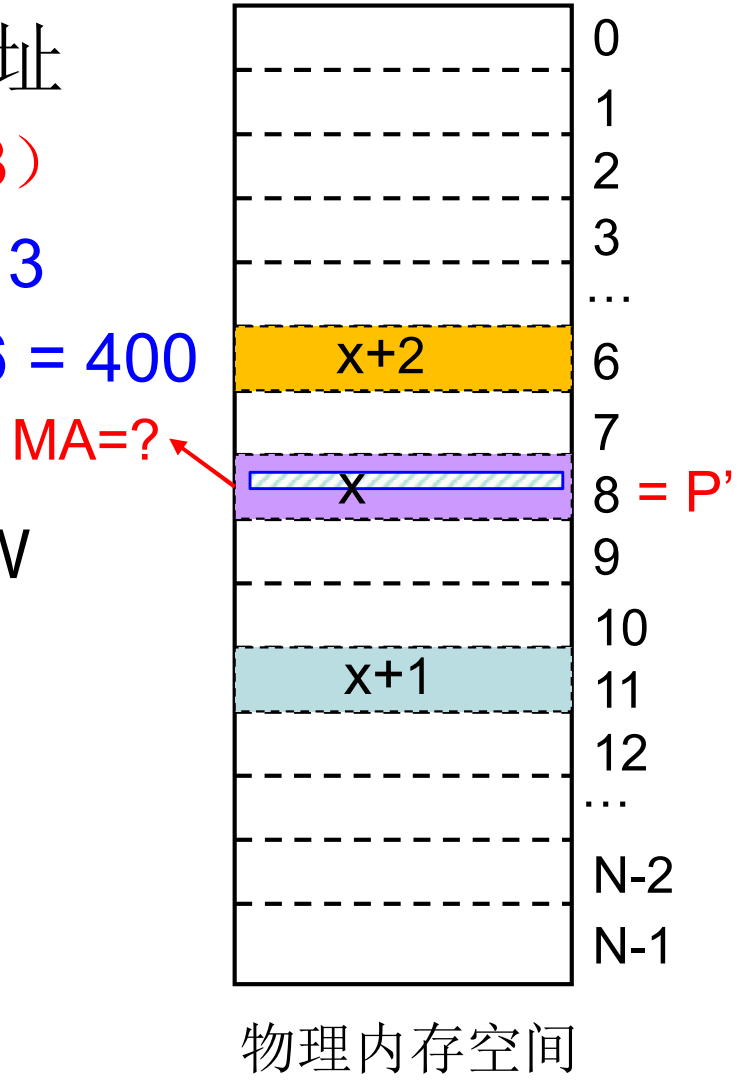
问题：MOV AX, 12688执行时共访问了内存几次？

- 虚拟地址（页式地址）→物理地址
- 例子：VA = 12688，MA=?（页= 4KB）
- (1) P = VA / 页大小 = 12688 / 4096 = 3

  W = VA % 页大小= 12688 % 4096 = 400
- (2)查页表,查页框号P’ = 8
- (3)计算MA = P’ x 页大小 + 页内偏移W

  = 8 x 4096 + 400 = 33168





地址映射过程

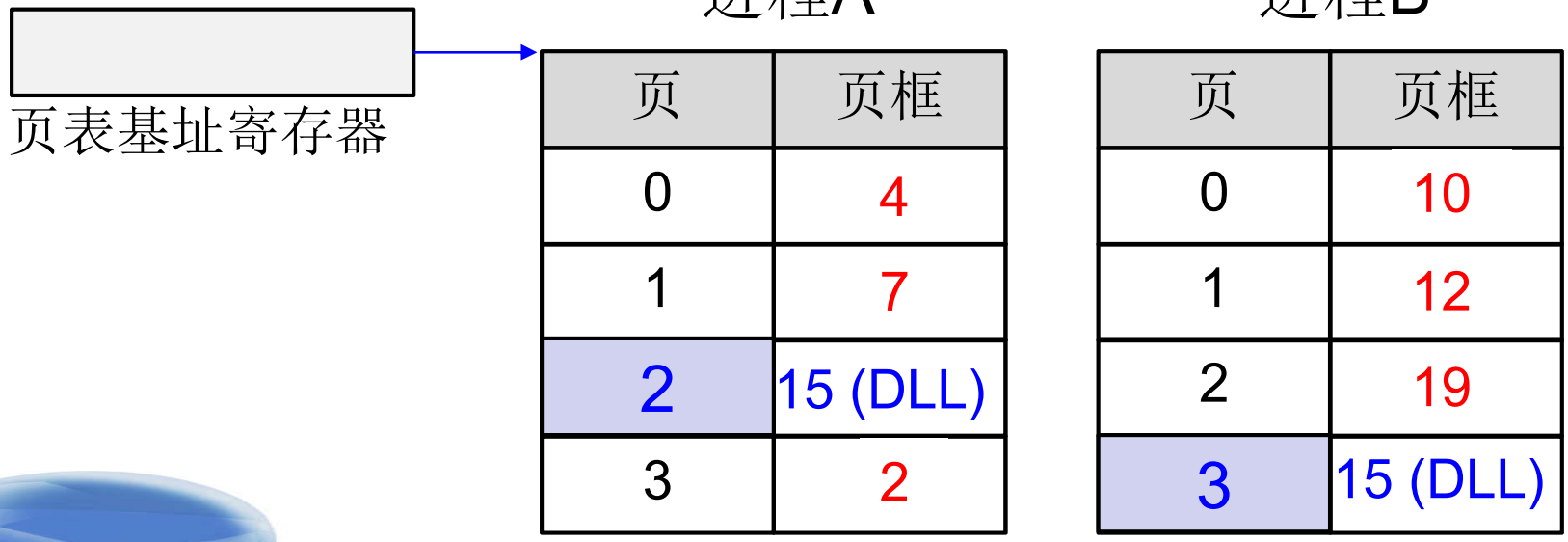
- 地址映射
  - 虚拟地址（页式地址）→物理地址
  - 例子：VA = 12688，MA=?
  - (1) P = VA / 页大小

    W = VA % 页大小
  - (2)查页表,查页框号P’
  - (3)计算MA = P’ x 页大小+ 页内偏移W
  - 注意　　= P’ << n || 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
…	…

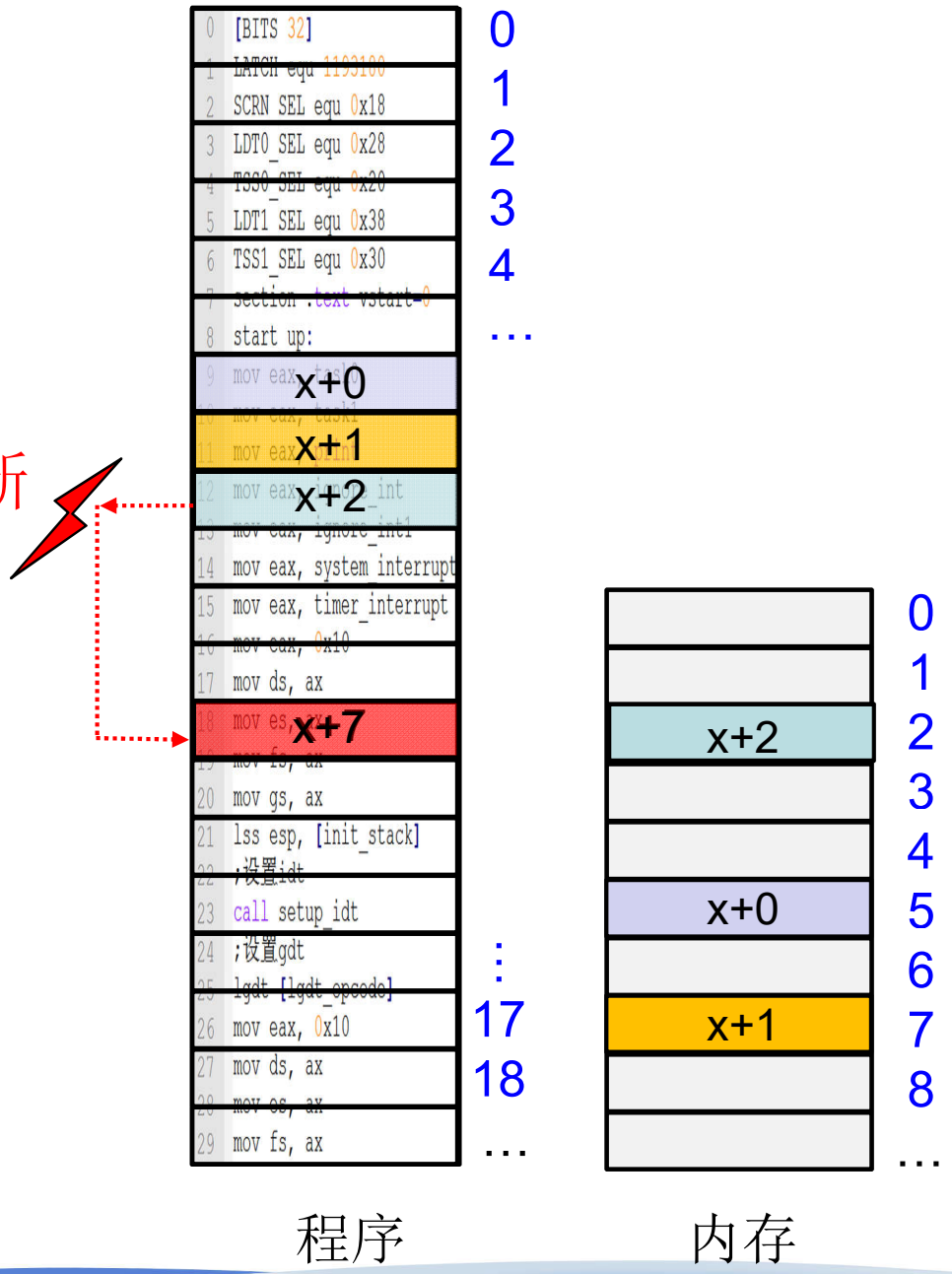
页表的建立

- 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

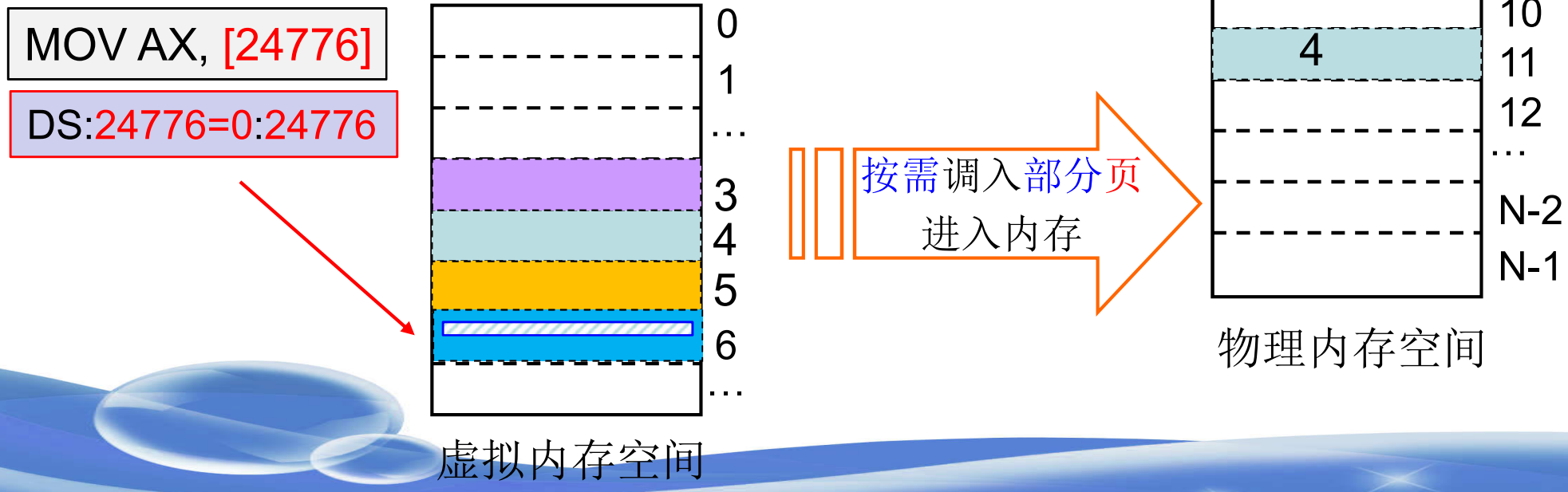
- 缺页中断的定义
  - 当程序要访问的目标页面不在内存时，程序将被迫临时中断：缺页中断
- 缺页中断的处理
  - 立即将所缺页面装入内存。
    - 页面从硬盘拷贝到内存
      - I/O操作，耗时较长
  - 缺页中断降低了程序实时性



缺页中断

- 例：目的页不在内存产生异常

页号	页框号
…	…
3	8
4	11
5	6
6	#$*!%



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

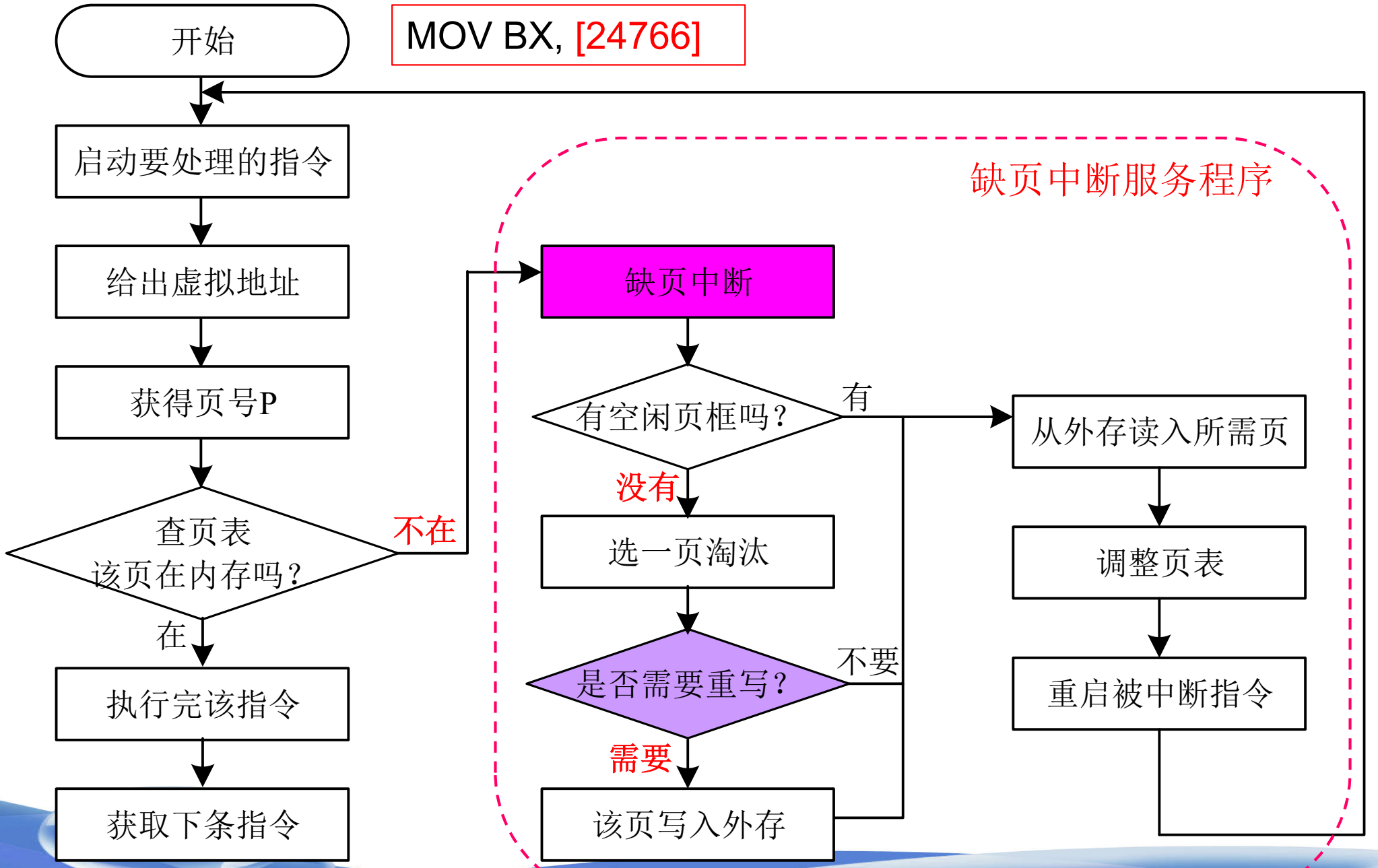
- 访问位——标识该页最近是否被访问？
  - 0 ——最近没有被访问
  - 1 ——最近已被访问
- 修改位——标识该页的数据是否已被修改？
  - 0 ——该页未被修改
  - 1 ——该页已被修改

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

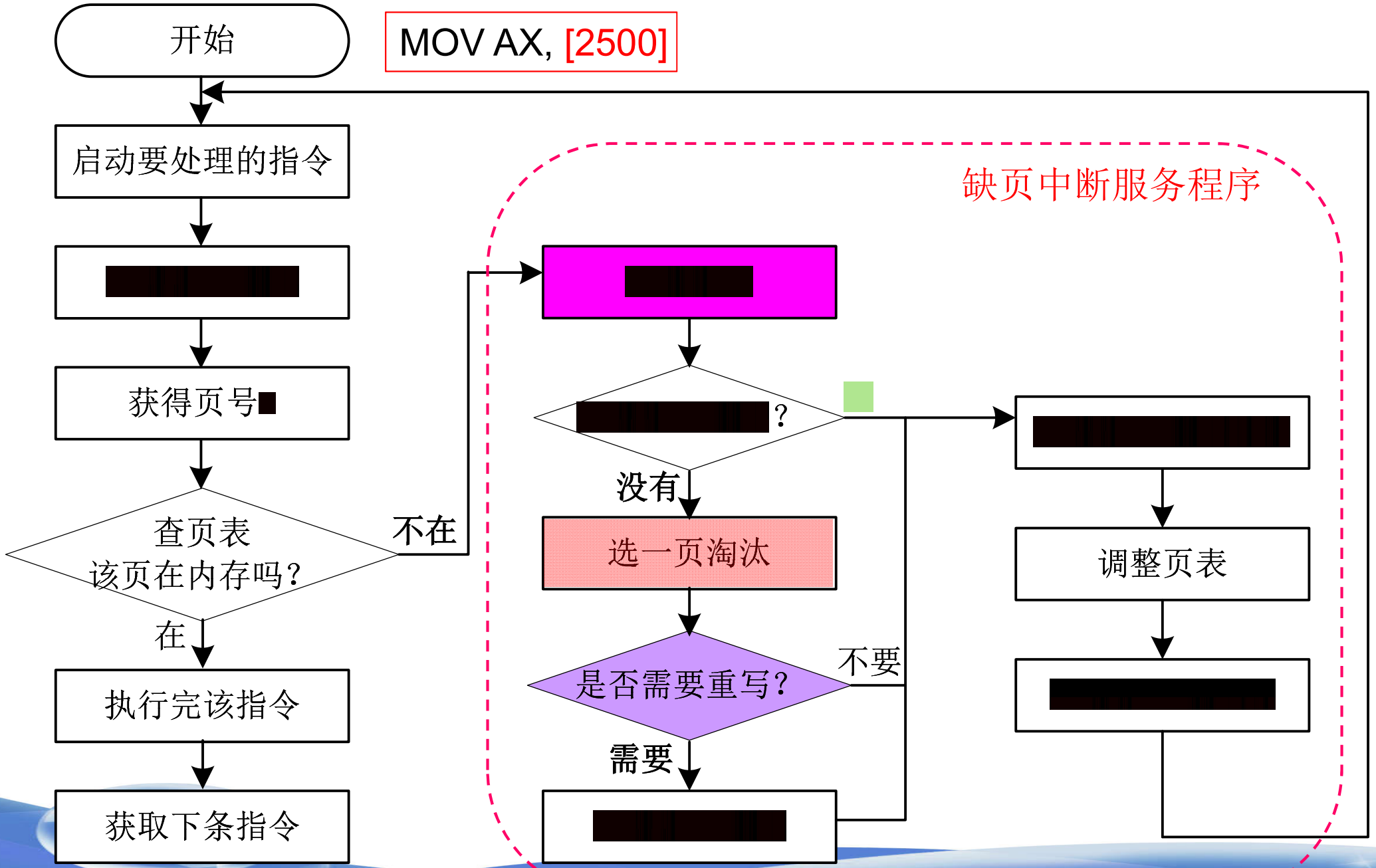
访存指令的执行过程（含缺页中断处理）



缺页中断

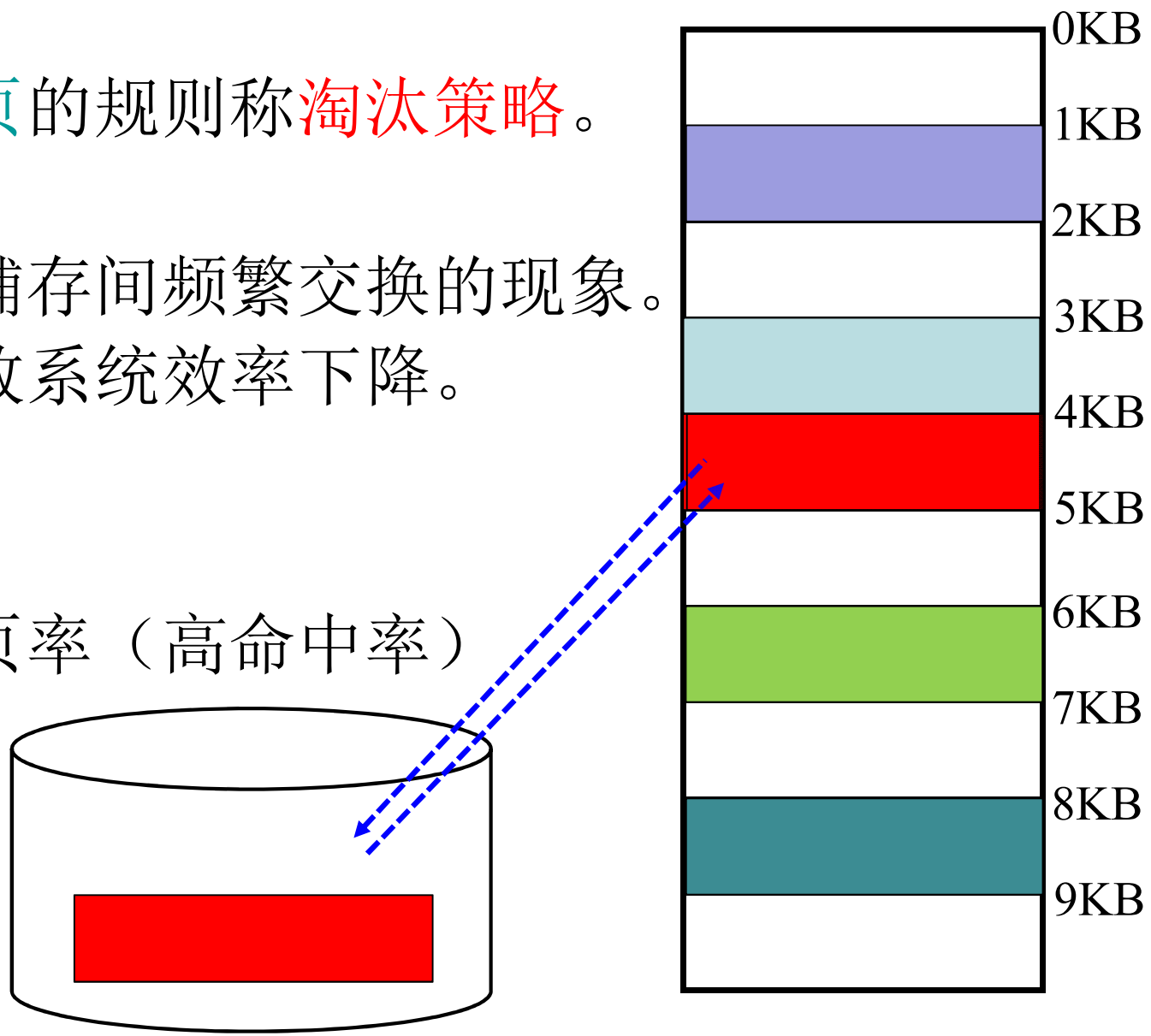
- 缺页（中断）率
  - 缺页率 f = 缺页次数 / 访问页面总次数
  - 命中率 = 1 – f

淘汰策略


开始
MOV AX, [2500]
启动要处理的指令
[redacted]
获得页号[redacted]
查页表
该页在内存吗?
不在
在
执行完该指令
获取下条指令
缺页中断服务程序
[redacted]
[redacted]?
没有
选一页淘汰
是否需要重写?
不要
需要
[redacted]
[redacted]
调整页表
[redacted]

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。
- 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
- 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%

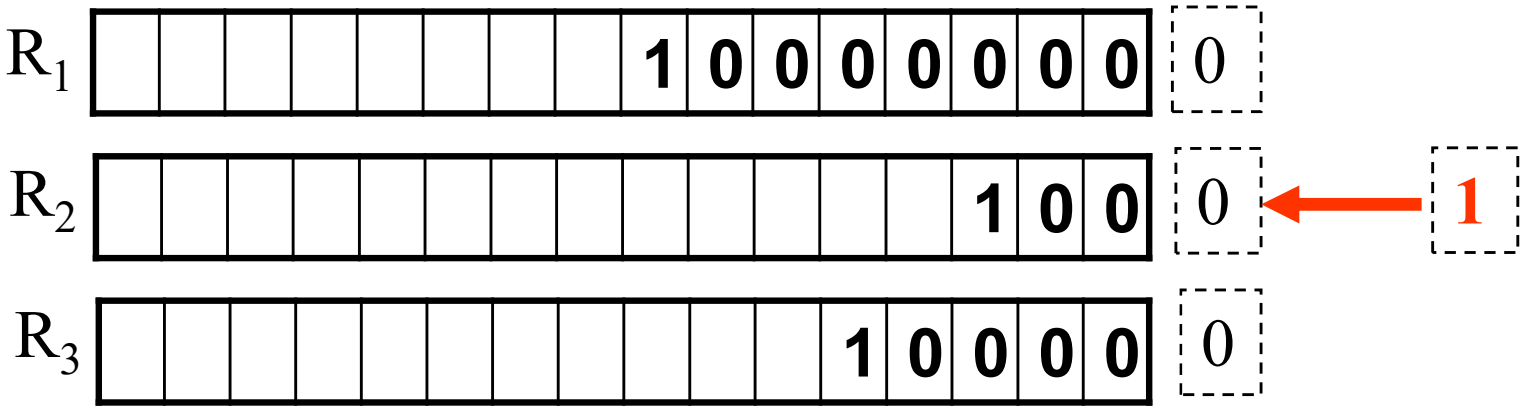
最久未使用淘汰算法（LRU，Least Recently Used）

- 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

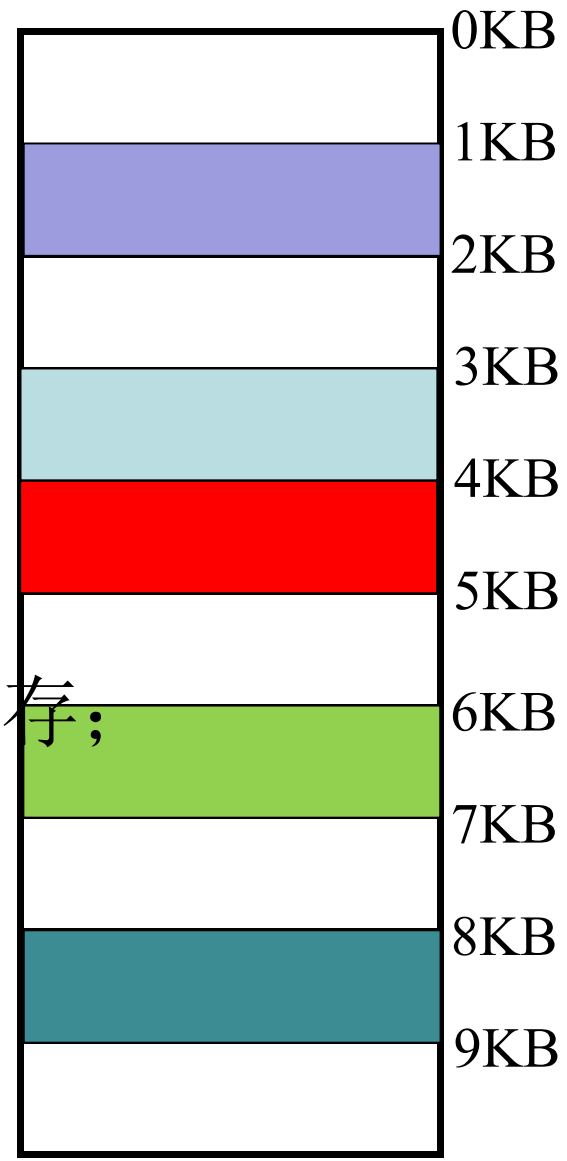
- **Least Frequently Used**
- 算法原则
  - 选择到当前时间为止被访问次数最少的页面
  - 每页设置访问计数器，每当页面被访问时，该页面的访问计数器加1；
  - 发生缺页中断时，淘汰计数值最小的页面，并将所有计数清零。

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

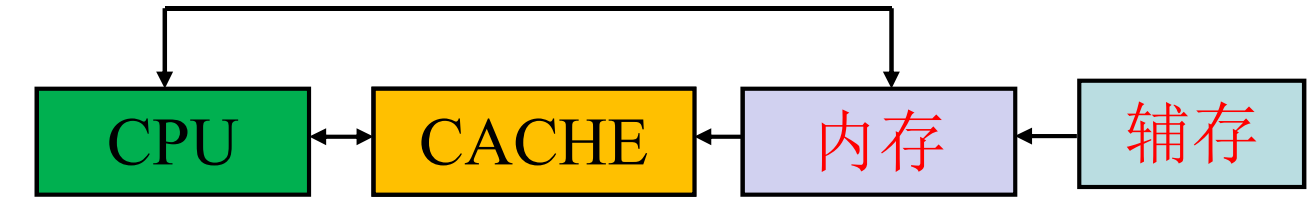
- 程序的局部性

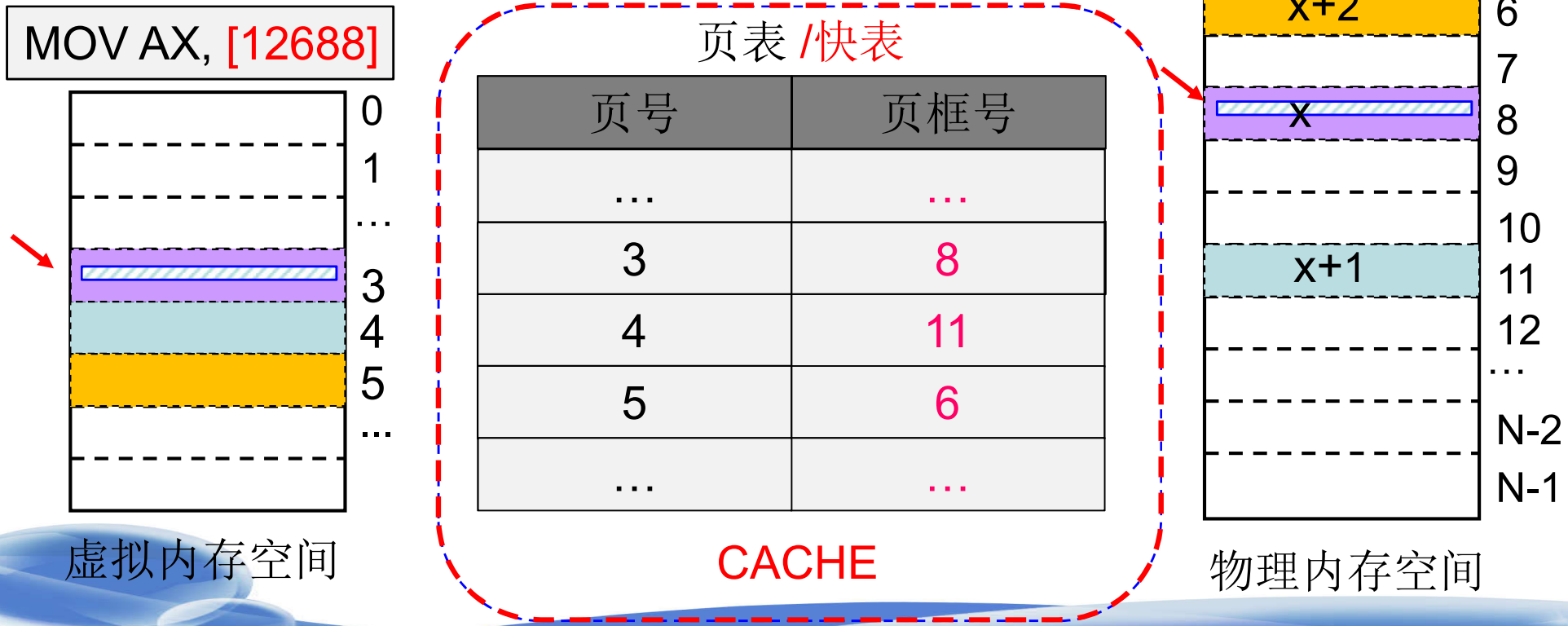
```
1  //遍历二维数组 (顺序:先行后列)
2  for( Col=0; Col<1000; Col++ )
3  {
4     for( Row=0; Row<1000; Row++ )
5     {
6        Matrix[Row][Col] = 0;
7     }
8  }
```

```
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

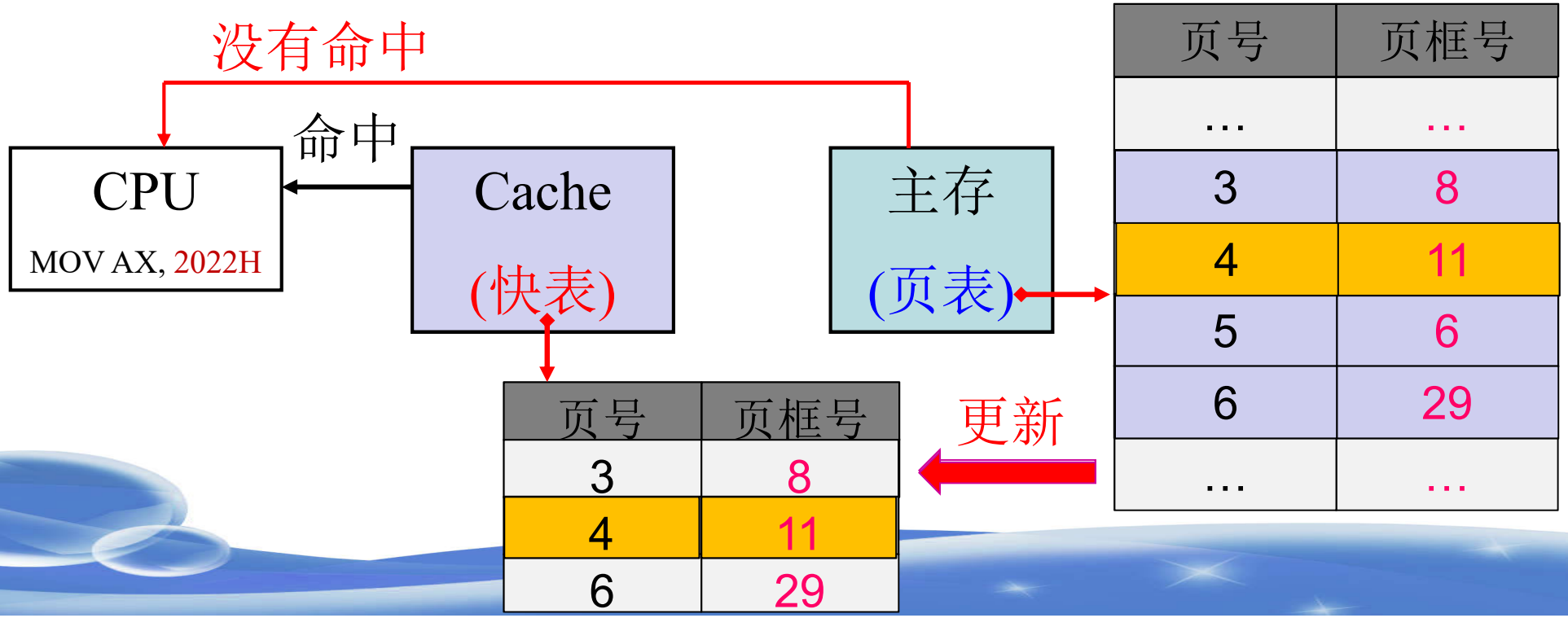
- 分级存储体系
  - ■CACHE + 内存 + 辅存





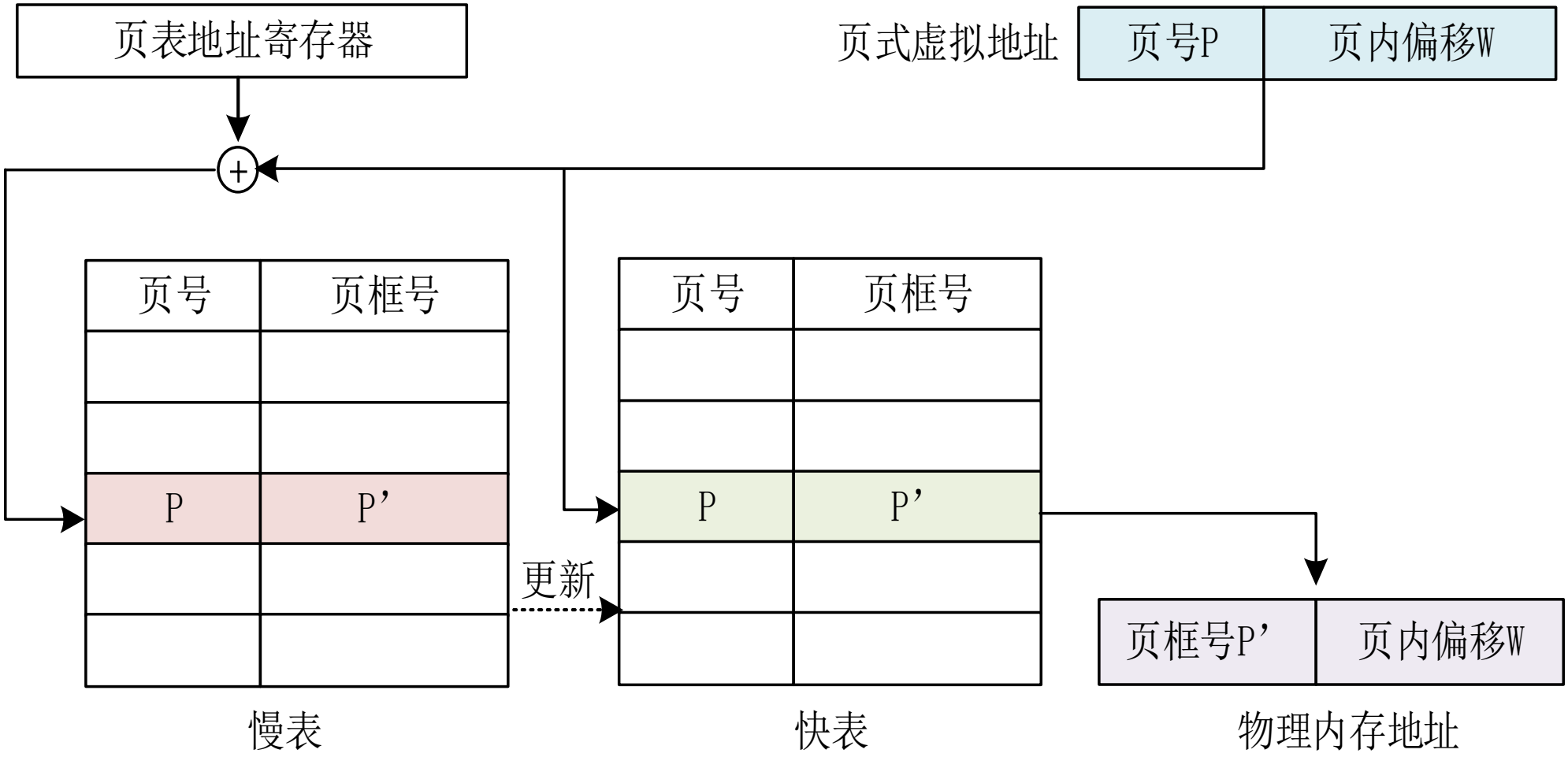
快表机制（Cache）

- 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1μs，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？ **1.15μs**
- 分析：
  - （1）若能通过快表完成，则只需1次访问内存。
    - ◆1μs 概率 = 85%
  - （2）若不能，则需要2次访问内存。
    - ◆2μs 概率 = 15%
  - （3）平均时间= 1μs *85% + 2μs *15% = 1.15μs

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 x（**150**＋50）KB ＝ 2M
  - 如果共享代码段
    - 占用内存 = 150 + 10 x 50 = 650KB

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：
        - 页表所占内存：1M * 4字节 = 4M
          - 页表占页框数：4M / 4K = **1K**页框
  - 问题：
    - **1**）页表全部装入过度消耗内存（4MB,1K个页框）。
    - **2**）难以找到连续1K个页框存放页表。

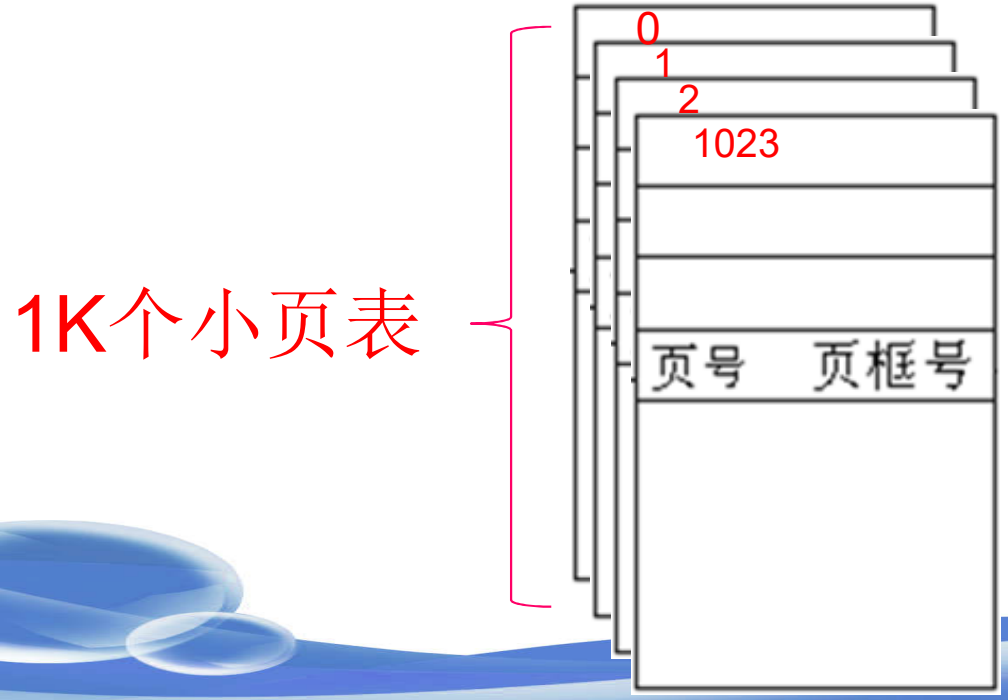
二级页表

- 策略：把页表(4MB)分拆成1K个小页表(4KB)且分散存放

1M条记录

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

1K条 … 1K条

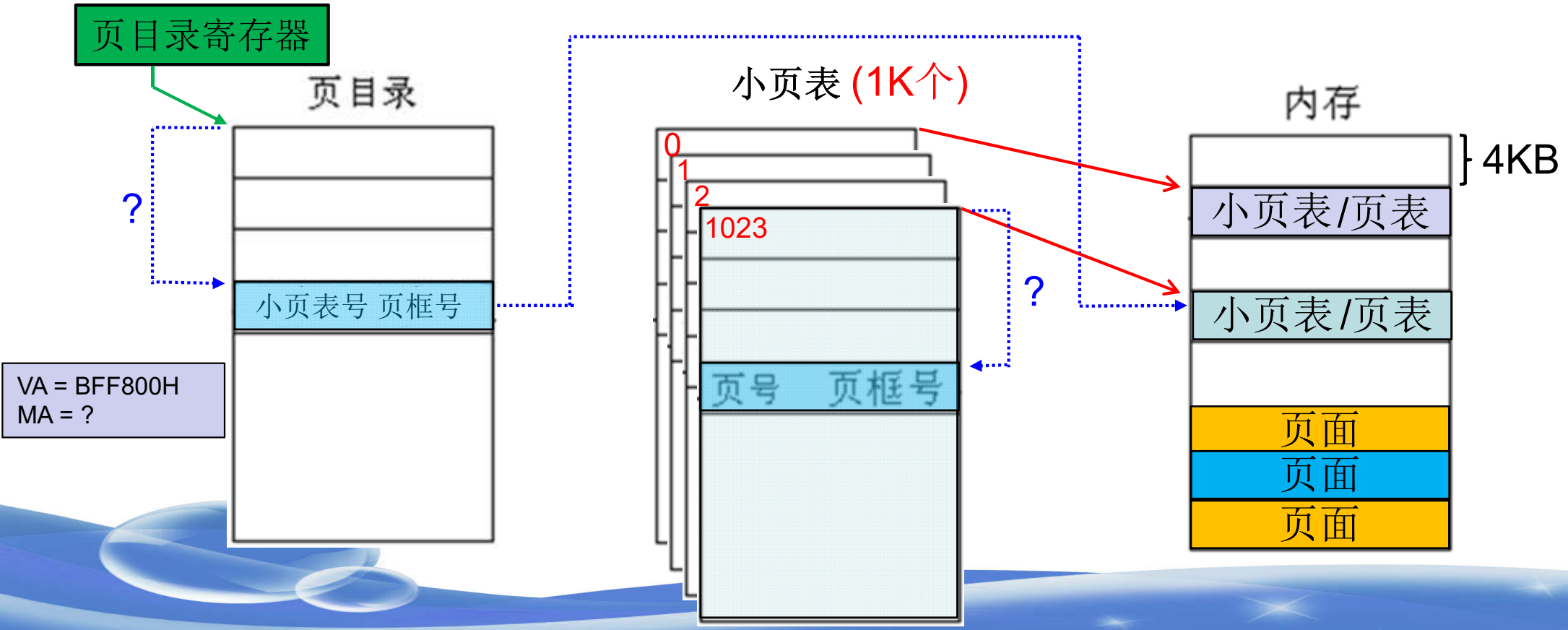


1K个小页表

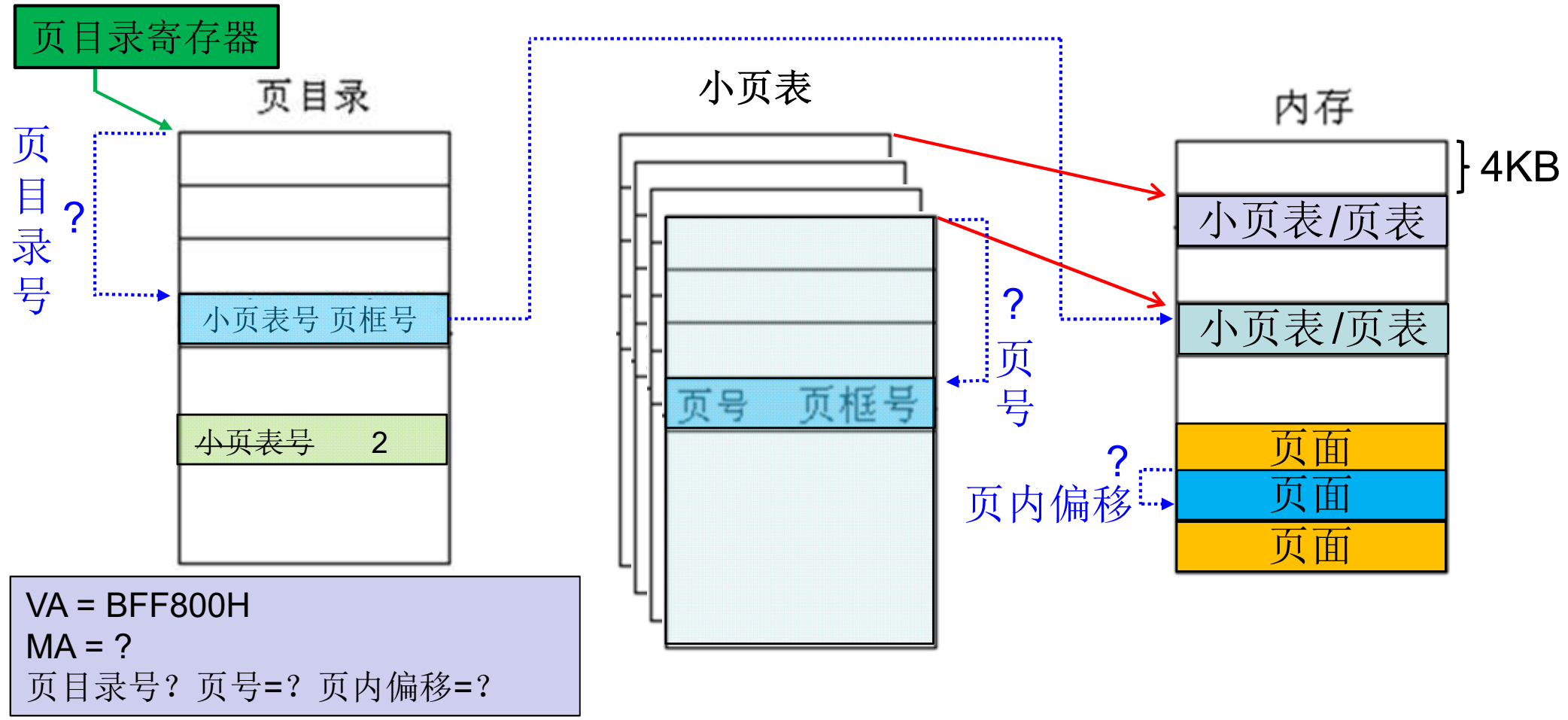
小页表含1K条记录；
小页表的大小是4K；
小页表占一个页框；

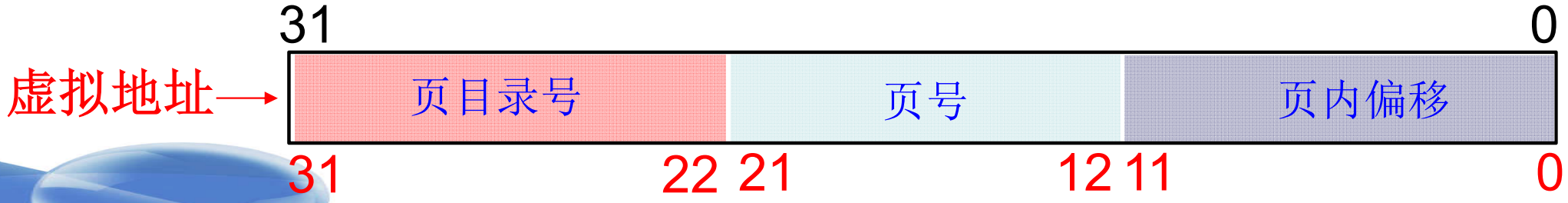
二级页表

- 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



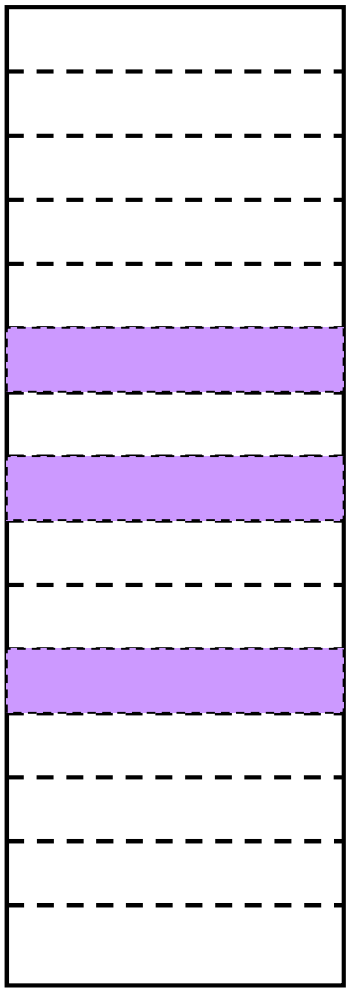
二级页表的地址映射过程





页式系统的缺点

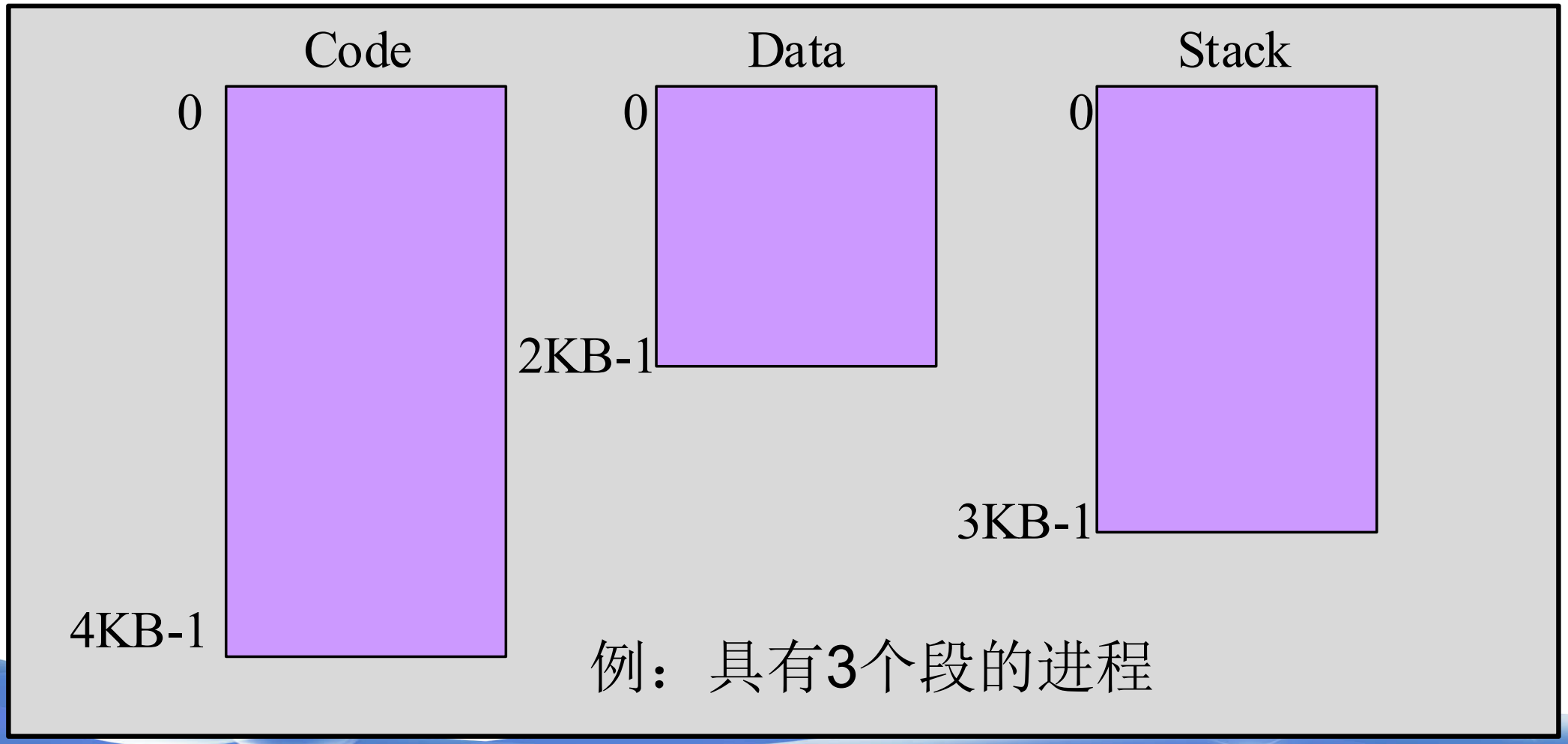
- 页式系统的不足
  - 页面划分无逻辑含义
  - 页的共享不灵活
  - 页内碎片



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）

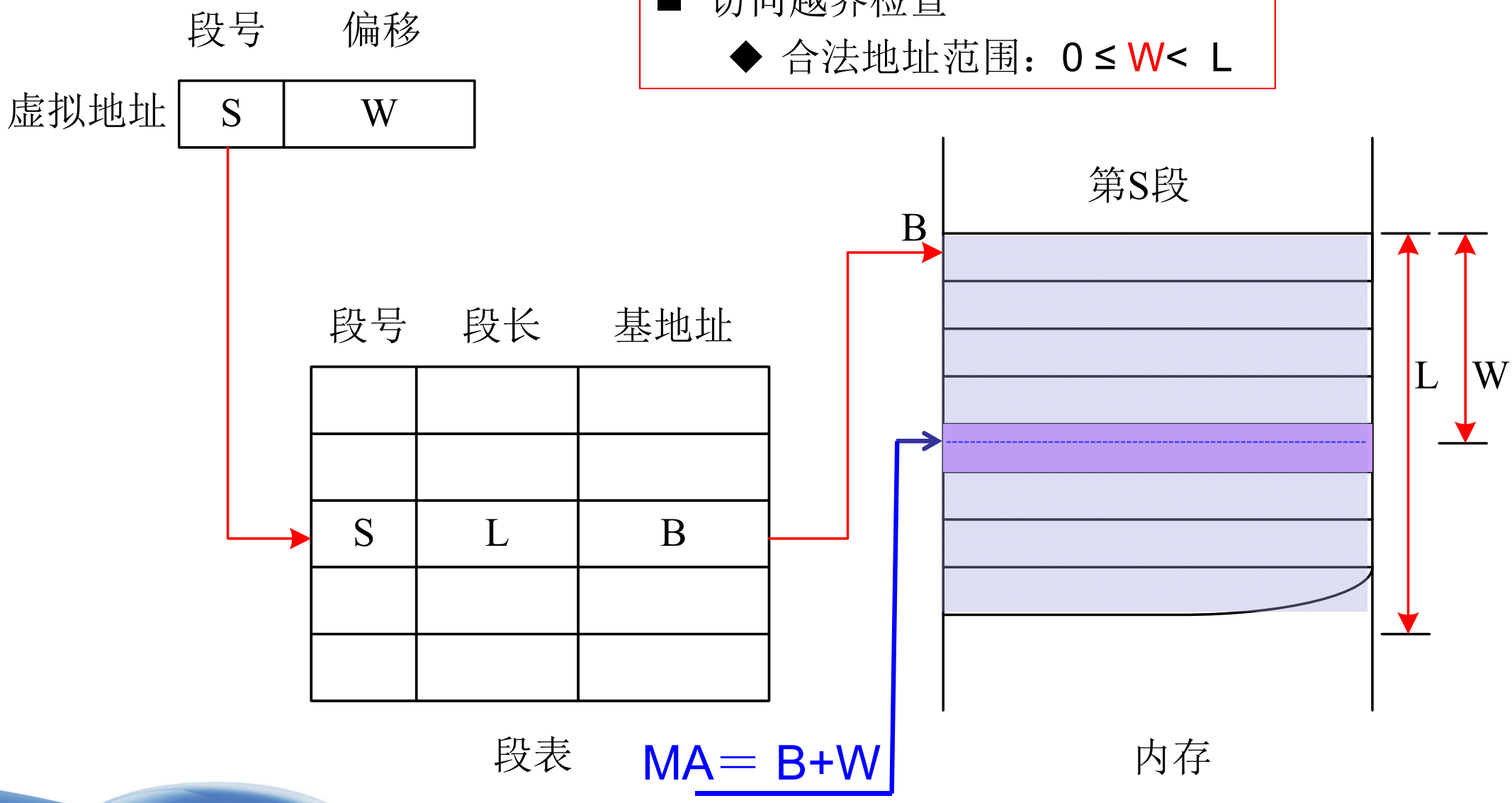
段号S	段内位移W

- 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

■ 访问越界检查

◆ 合法地址范围：0 ≤ W< L



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

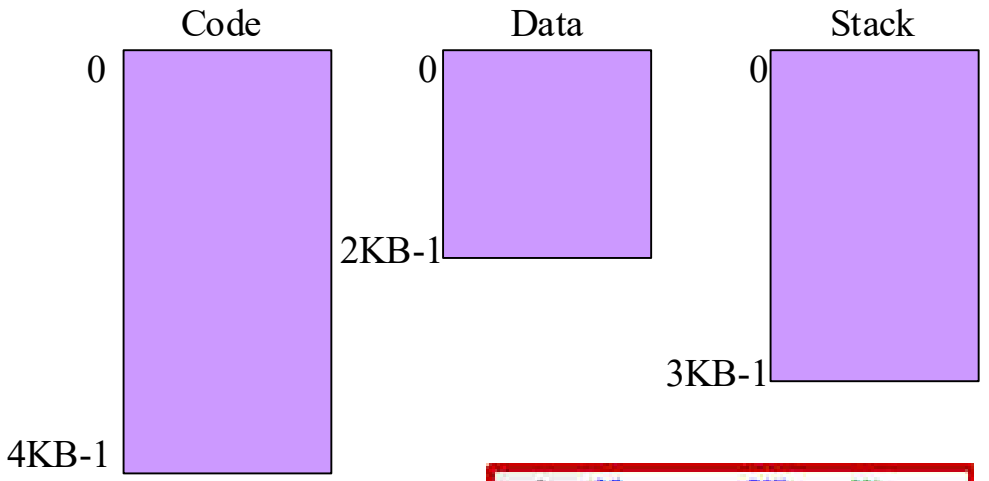
段号	长度	基址	中断位	访问位	修改位	R	W	X

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

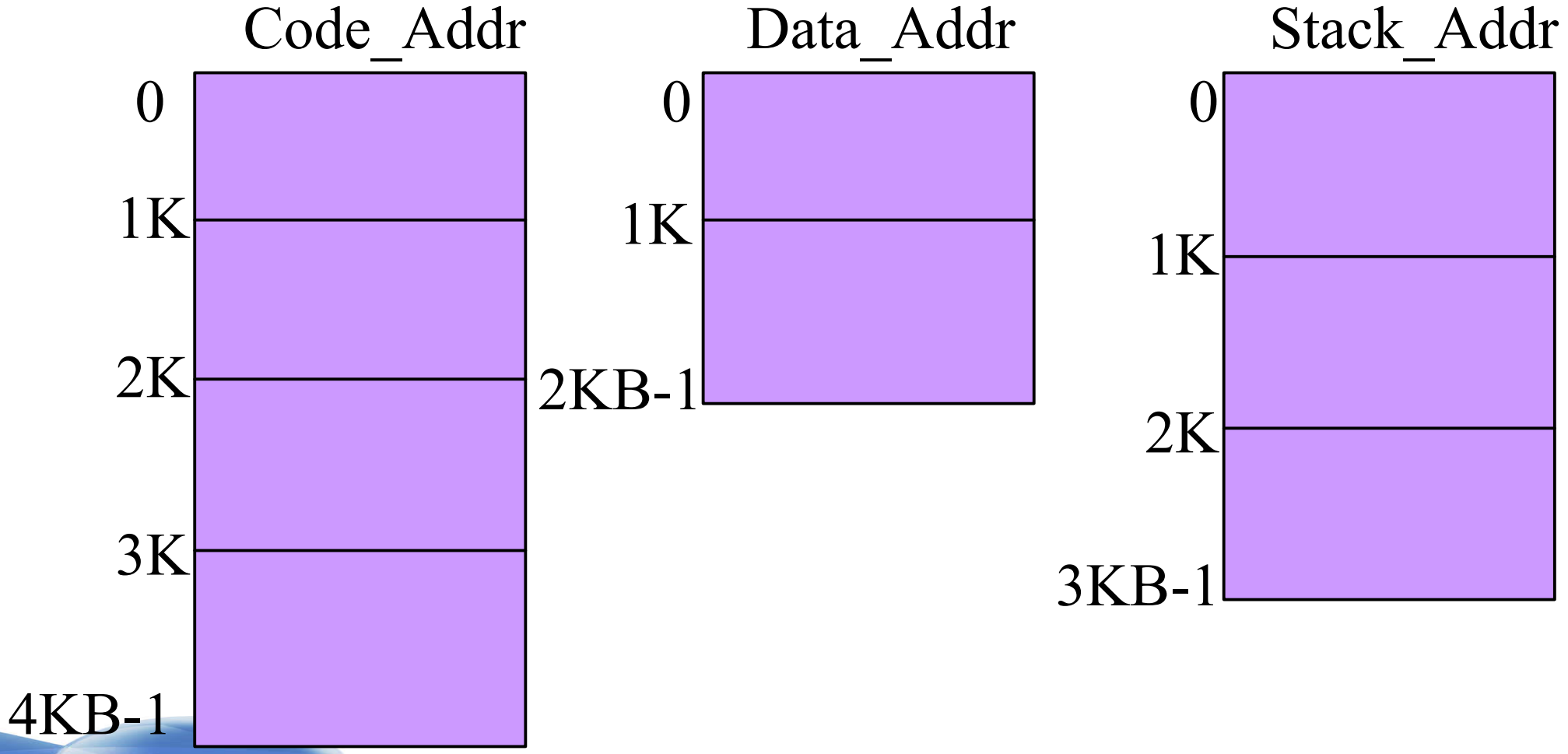
- 地址空间的区别
  - 页式系统：一维地址空间
  - 段式系统：二维地址空间
- 段与页的区别
  - 段长可变 vs 页面大小固定
  - 段的划分有意义 vs 页面无意义
  - 段方便共享 vs 页面不方便共享（相对）
  - 段用户可见 vs 页面用户不可见
  - 段偏移有溢出 vs 页面偏移无溢出




```
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   THRE    EQU   020h
7   TEMT    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    IXCHAR
```

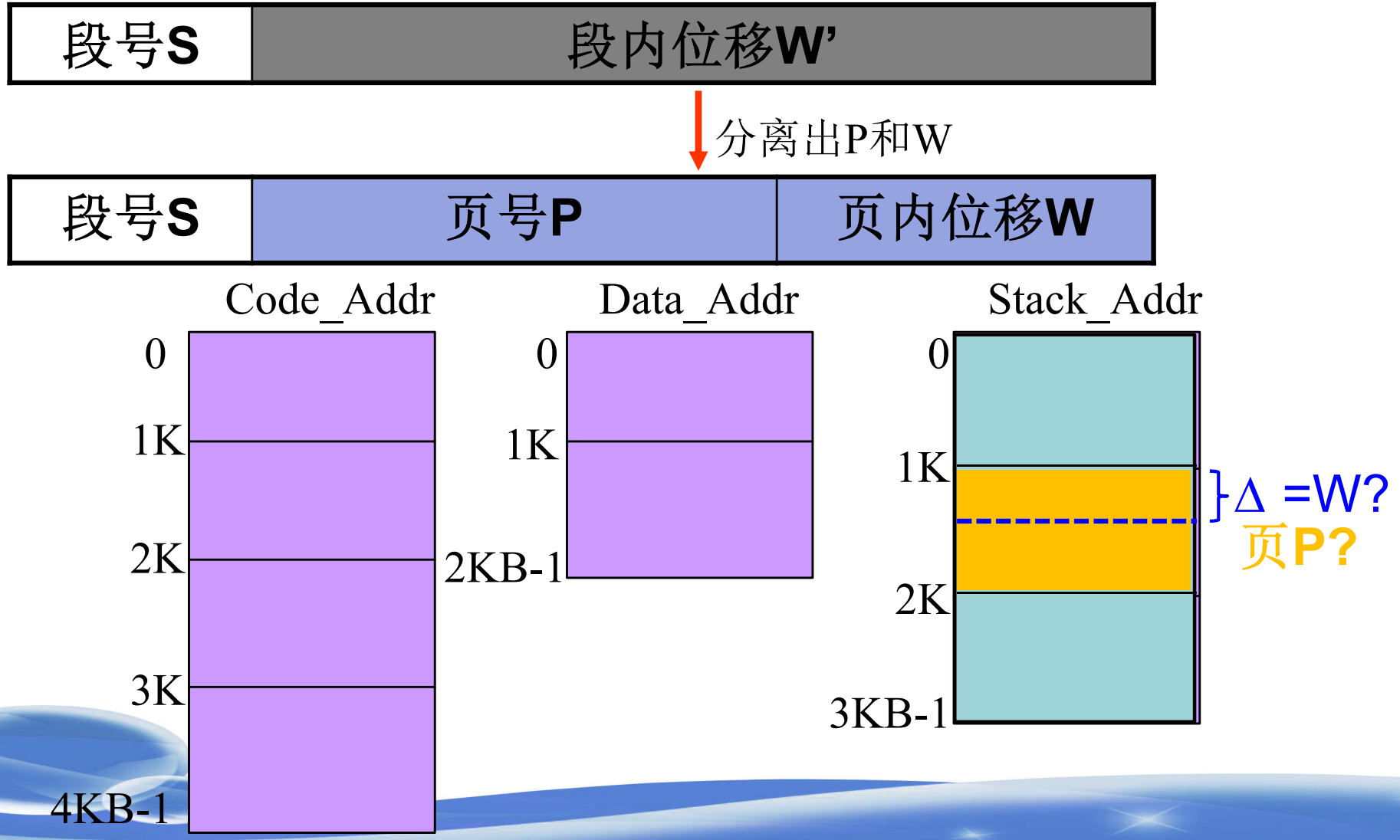
段页式存储管理

- 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



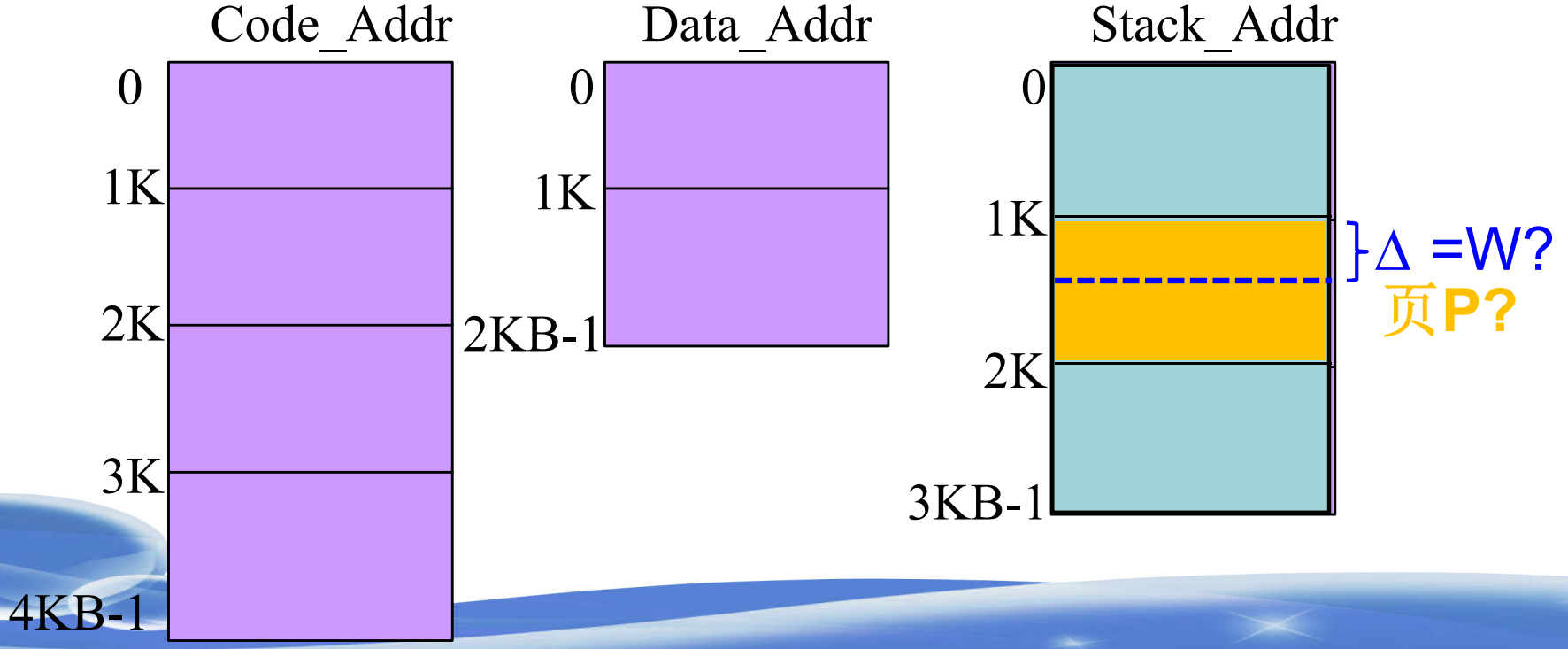
段页式地址映射

- 段页式地址
  - 段号S、页号P和页内位移W

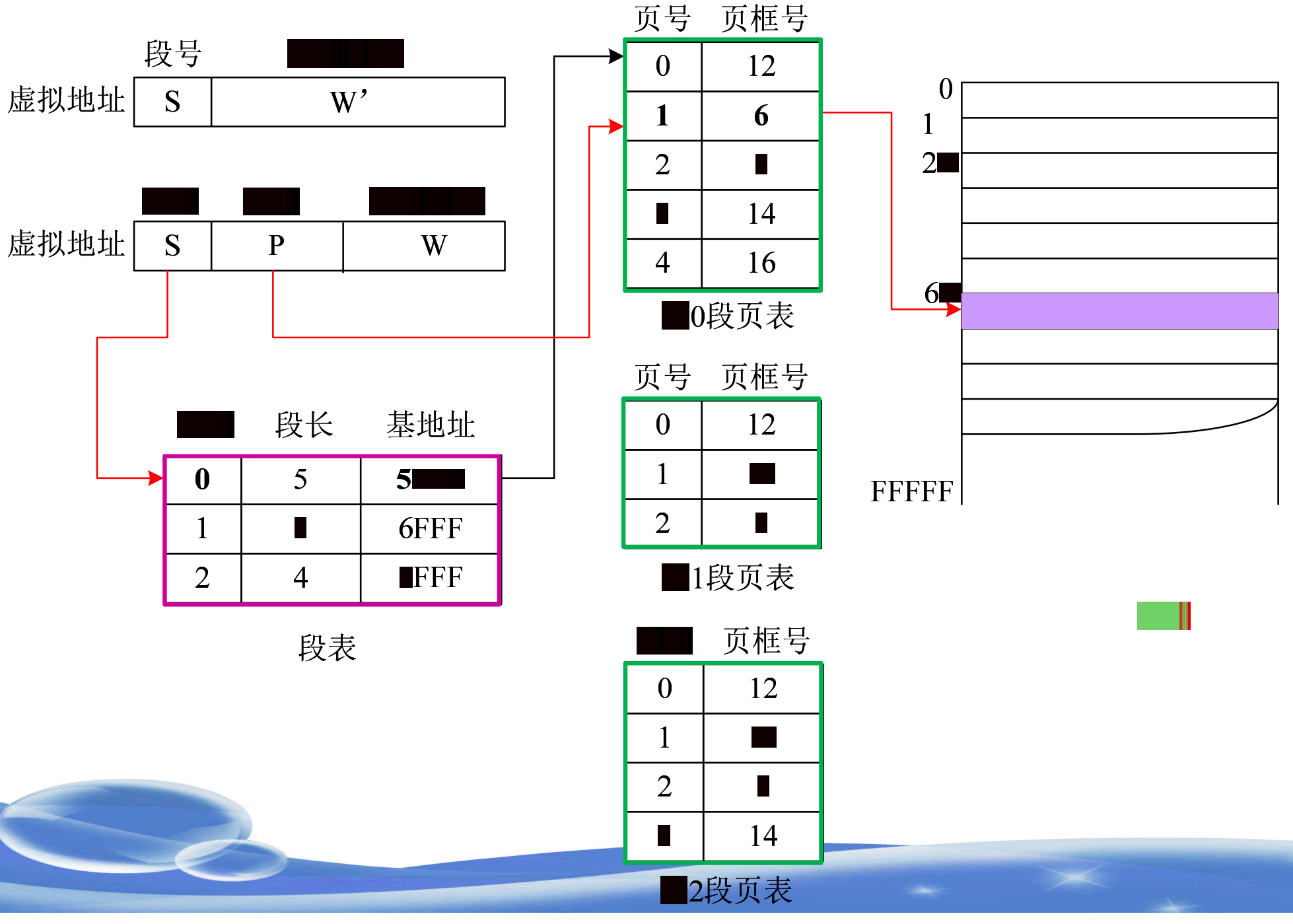


段页式地址映射

- 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') → (S, P, W) → MA



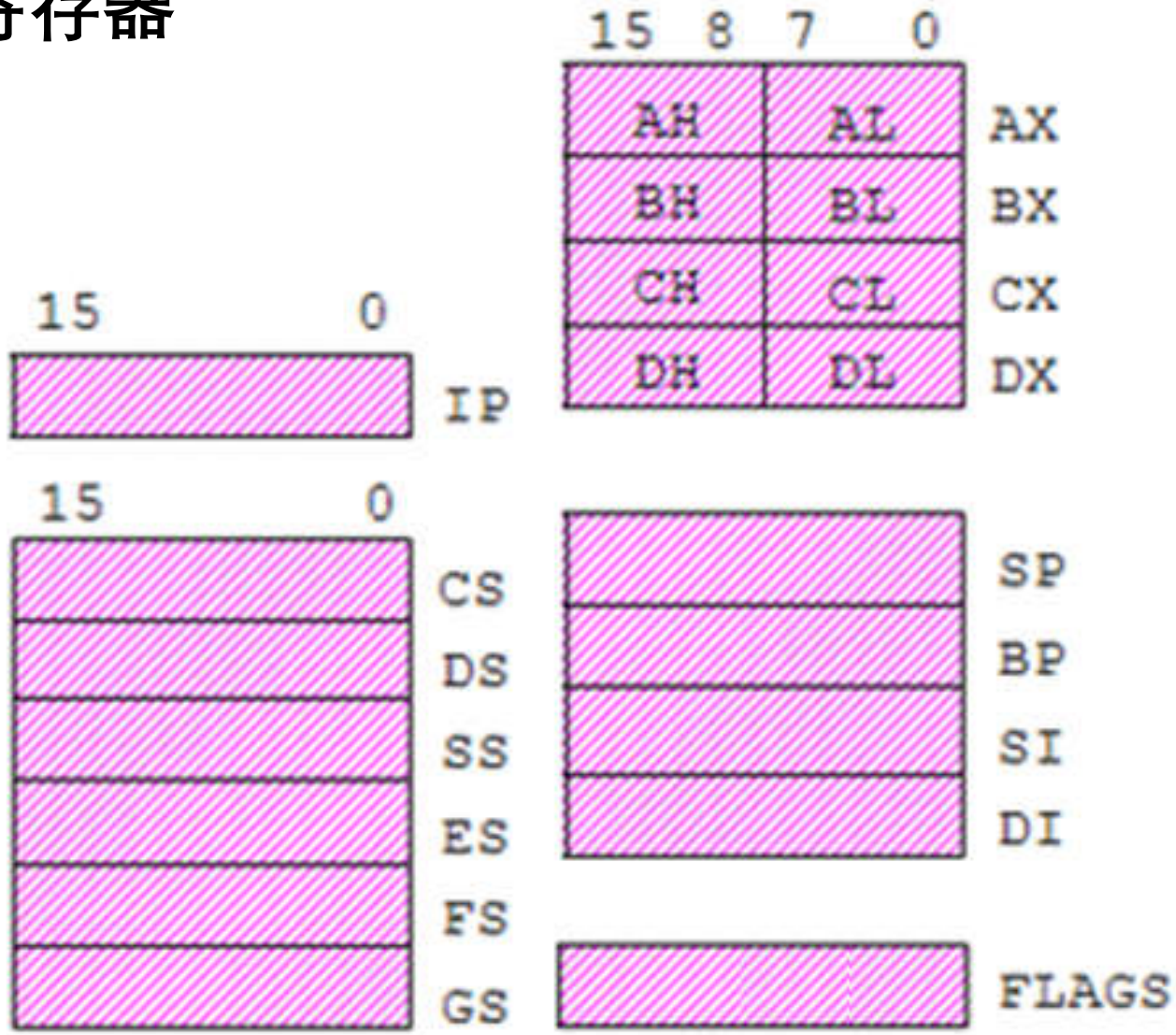
7.5 i386和Linux存储管理

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 ≠ 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

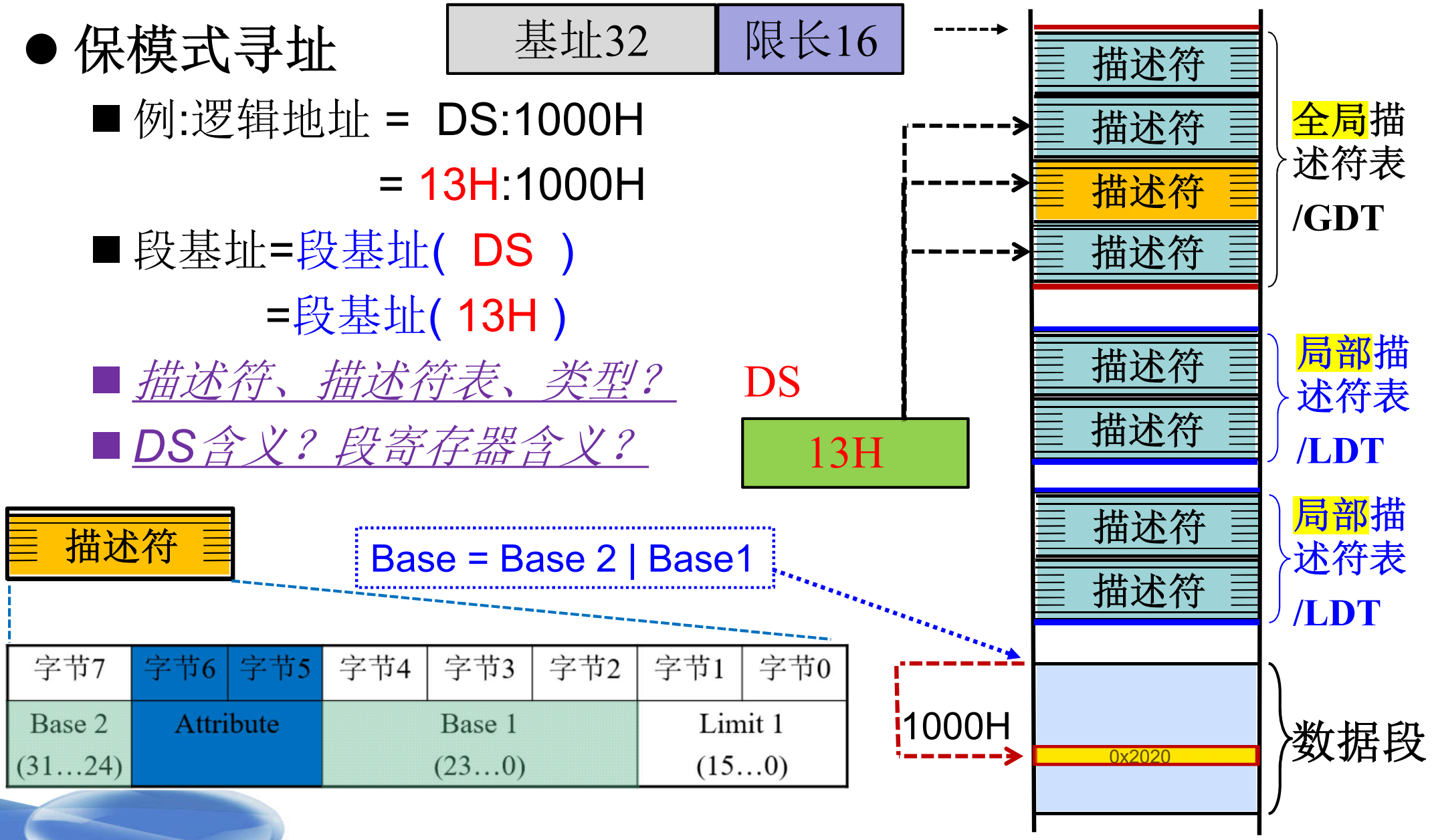
段基址(130H)

\+ 1000H

XXXXXXXXH

保护模式（Protect Mode）

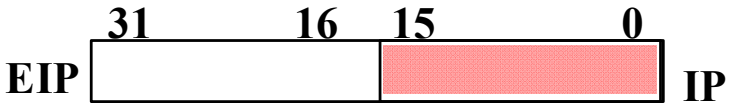
- 保模式寻址
  - 例:逻辑地址 = DS:1000H
    = 13H:1000H
  - 段基址=段基址(DS)
    =段基址(13H)
  - *描述符、描述符表、类型？*
  - *DS含义？段寄存器含义？*

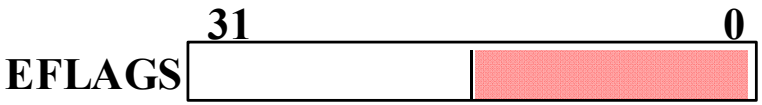


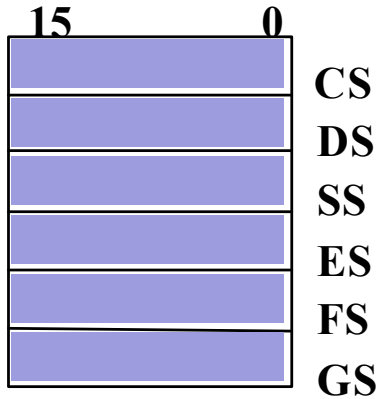
保护模式（Protect Mode）

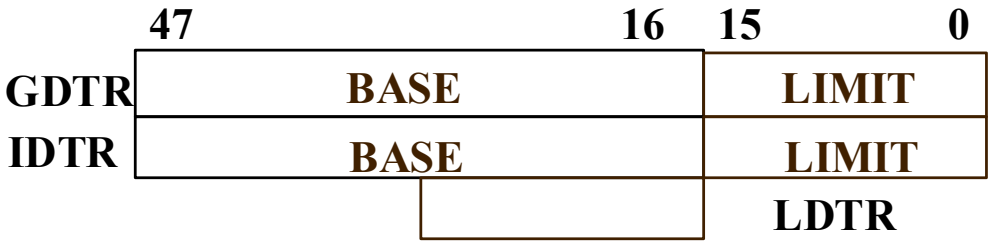
- **保模式寻址**
  - 优化分段管理机制
  - 支持分页管理机制
  - 4GB内存空间（32位）
- **CPU特性**
  - 支持多任务
  - 支持特权级机制
  - 扩展寄存器和新增寄存器
    - EAX~EDX，EIP，ESP，ESI，EDI，EFLAGS
    - CR0~CR4，GDTR，LDTR，IDTR，TR

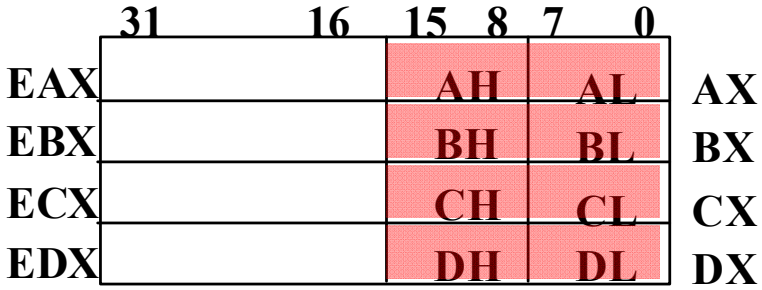
保护模式（Protect Mode）

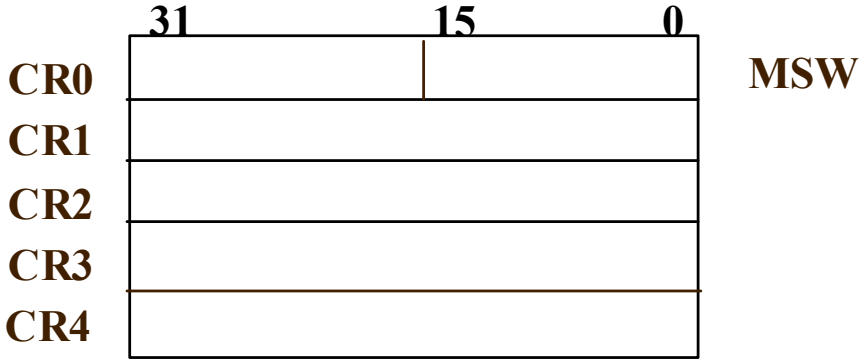


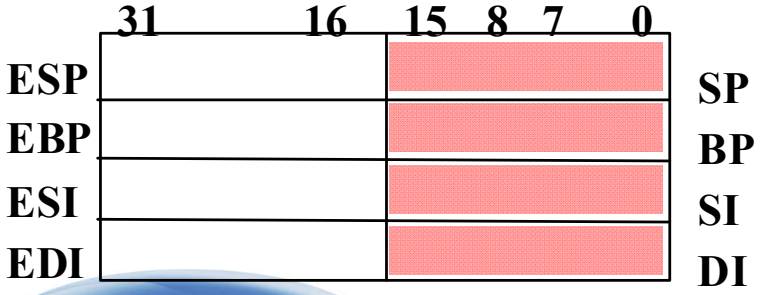


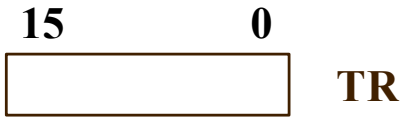






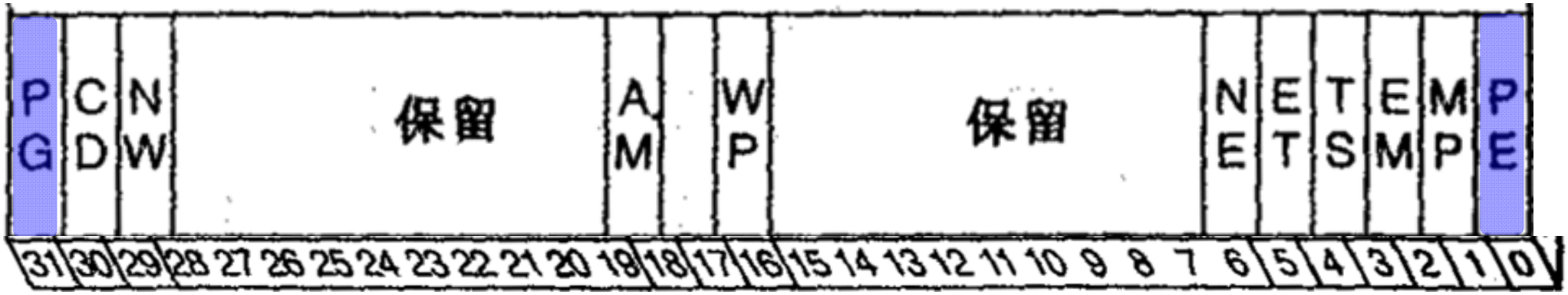






五个控制寄存器CR0-CR4

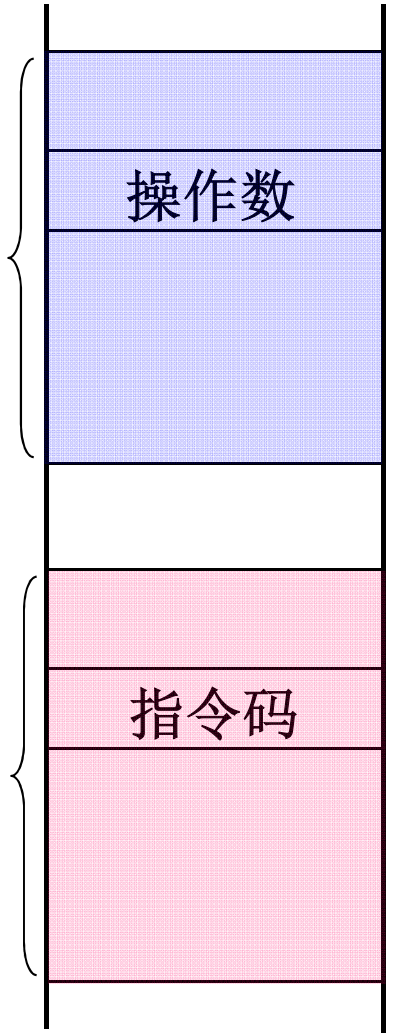
- 控制寄存器CR0



- 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

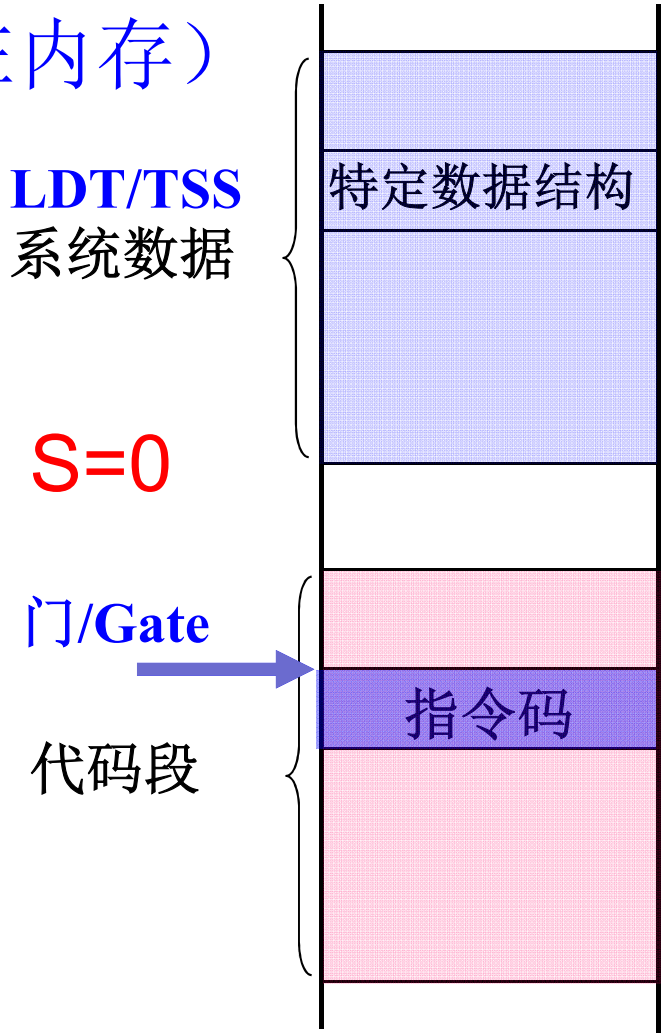
描述符（Descriptor）

- 属性的描述
  - ■DPL:描述符特权级别 Descriptor Privilege Level
  - ■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



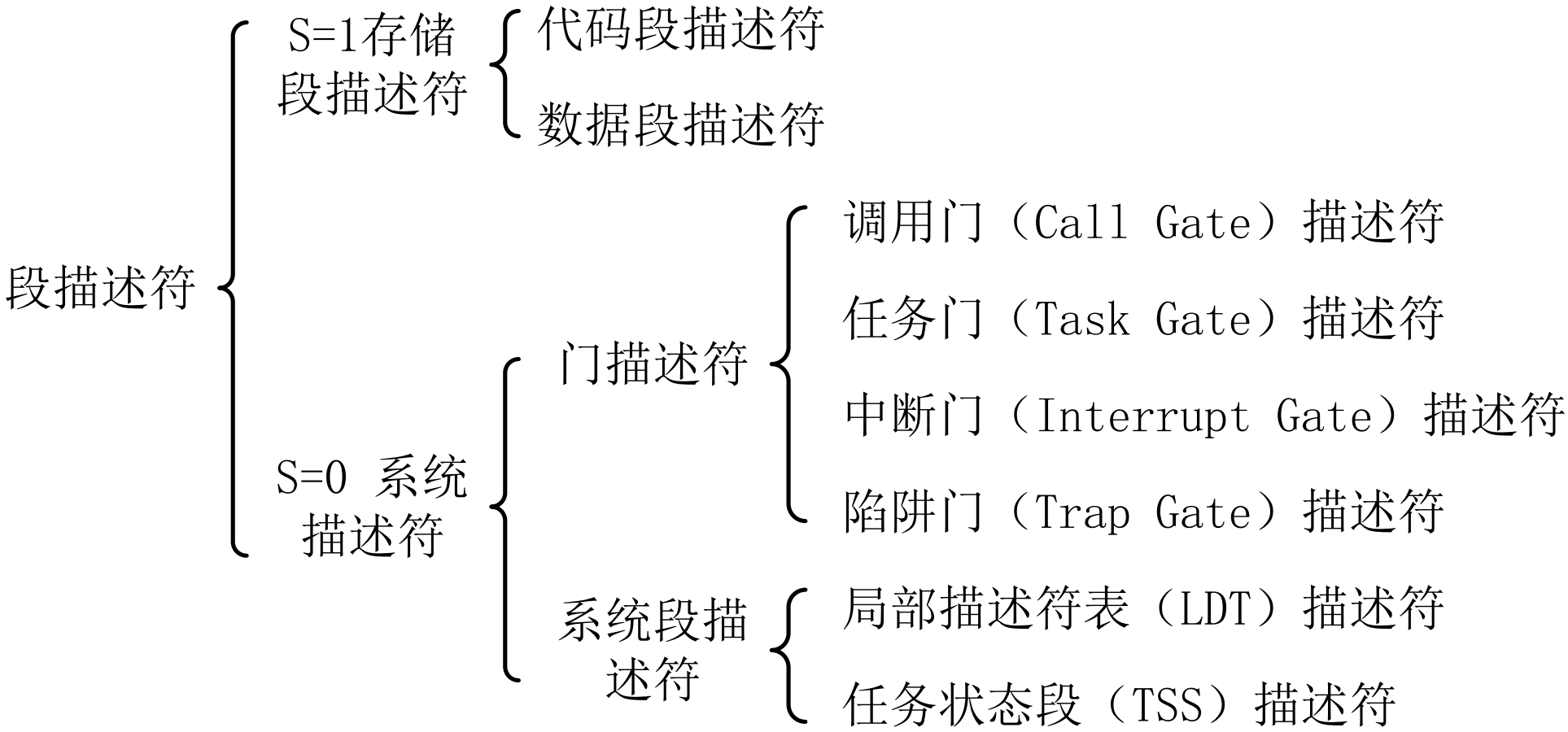
描述符（Descriptor）

- 属性的描述
  - DPL:描述符特权级别 Descriptor Privilege Level
  - 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



描述符（Descriptor）

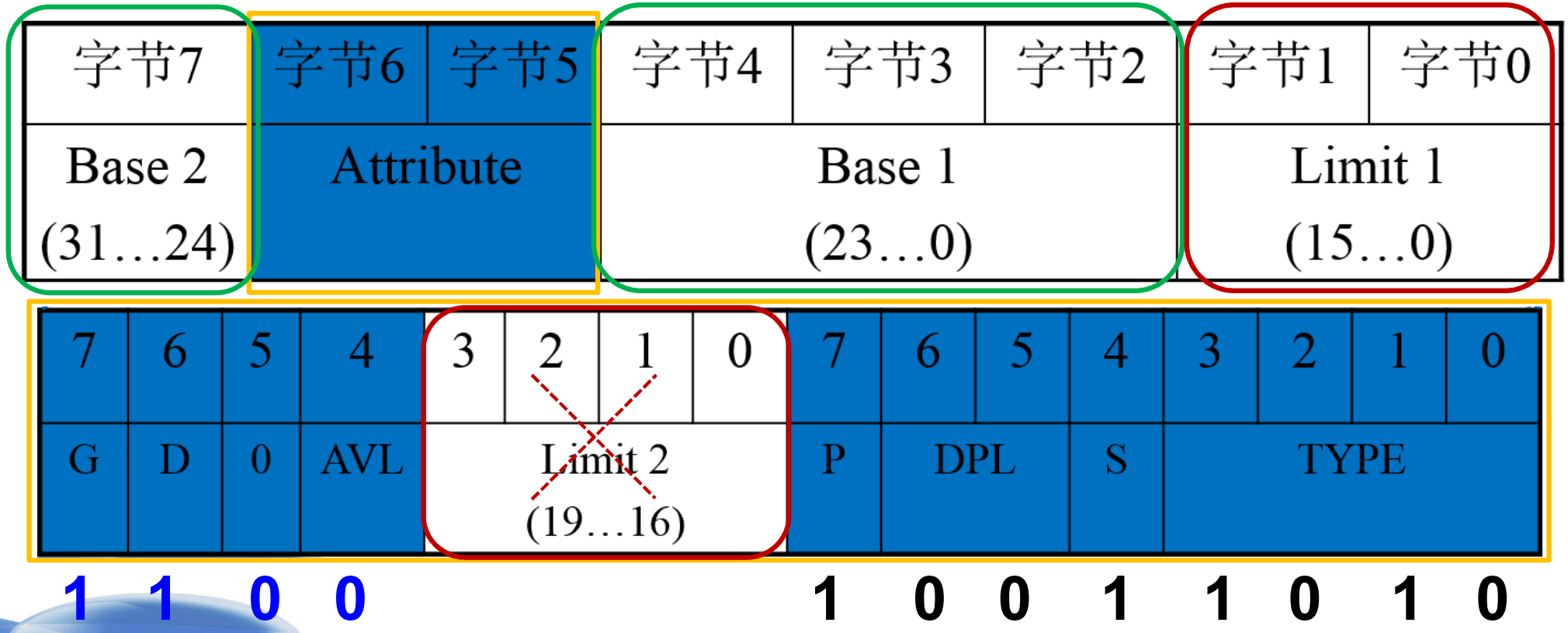
- 描述符的类型（S）/属性（TYPE）



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）

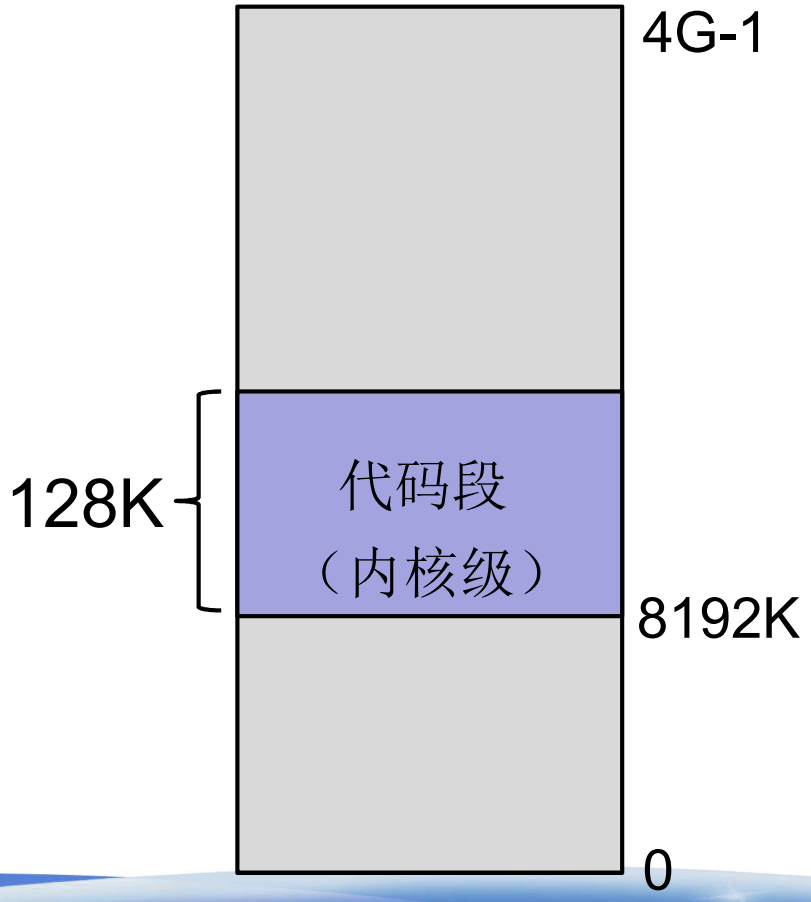
- 例：.quad 0x 00C0 9A80 0000 001F
  - 7~4：0000 0000 1100 0000 1001 1010 1000 0000
  - 3~0：0000 0000 0000 0000 0000 0000 0001 1111



描述符（Descriptor）

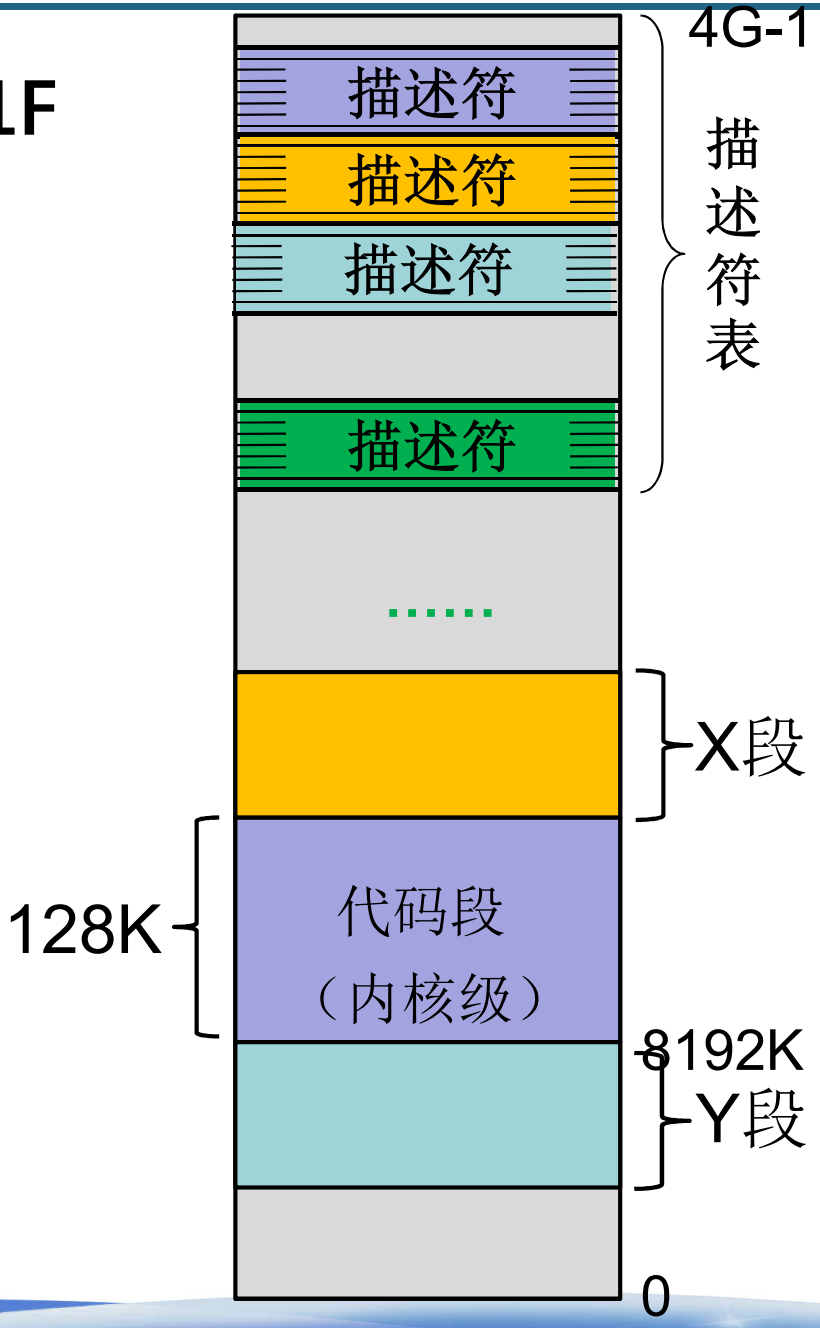
- 例: .quad 0x 00C0 9A80 0000 001F
  - 7~4: 0000 0000 1100 0000 1001 1010 1000 0000
  - 3~0: 0000 0000 0000 0000 0000 0000 0001 1111
  - G= 1（Page = 4KB）
  - BASE = 0x800000 = 8192K
  - Limit = (0x1F +1) * 4K = 128K
  - S= 1
  - TYPE = 1010

  代码段
  - P= 1 （在内存中）
  - DPL= 00 （内核级）



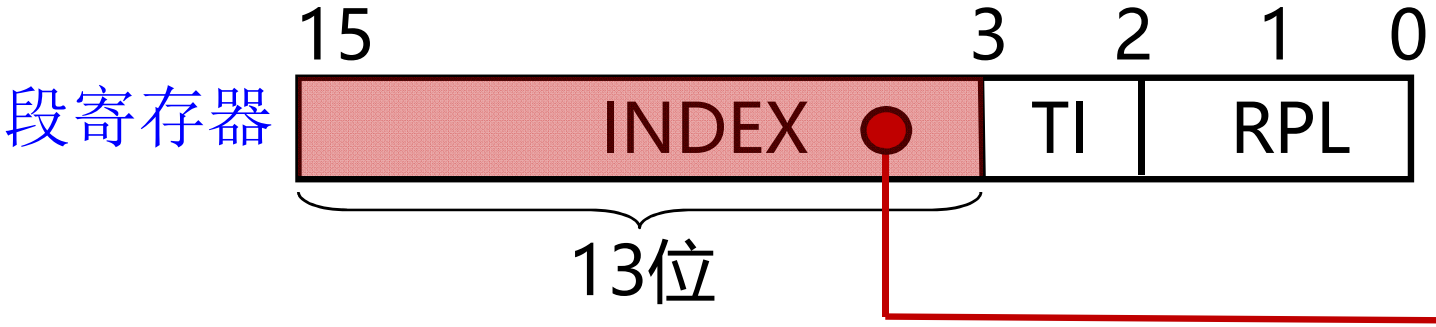
描述符表（Descriptor Table）

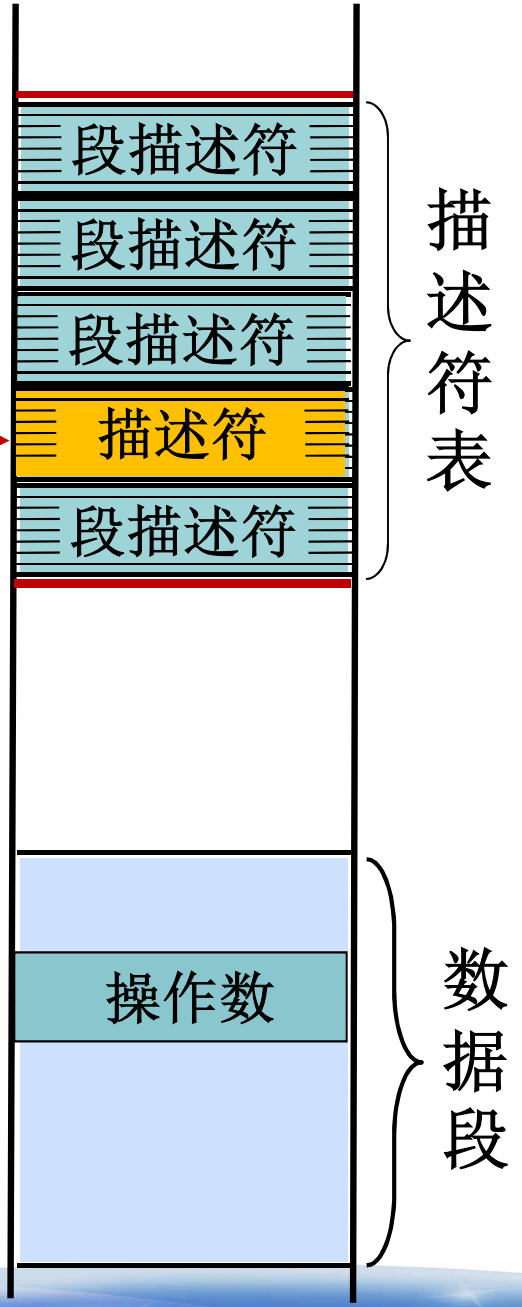
- 例：.quad 0x 00C0 9A80 0000 001F
- 描述符表
  - 存放描述符的数组/线性表
  - 长度：8字节的整数倍。
- 描述符表类型
  - 全局描述符表GDT
    - Global Descriptor Table
  - 局部描述符表LDT
    - Local Descriptor Table
  - 中断描述符表IDT
    - Interrupt Descriptor Table



选择子(Selector) /16位/段寄存器

- 用于选择GDT/LDT等表中的描述符

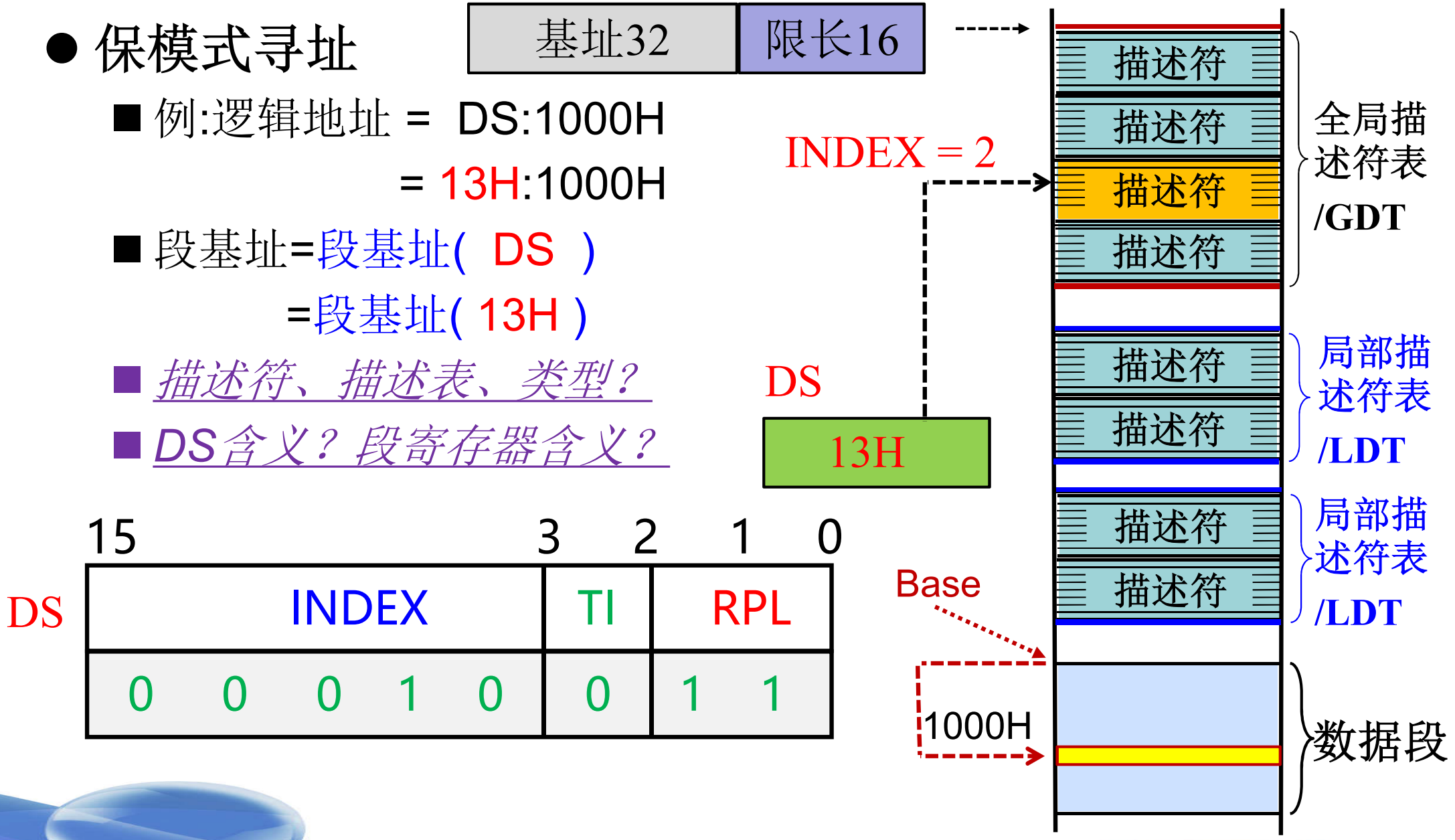




- ■索引域（INDEX）
  - ◆13位，描述符在描述符表中的序号
- ■表标识域（TI，Table Indicator）
  - ◆1位：GDT(0)|LDT(1)
- ■请求特权级域（RPL）
  - ◆2位，Request Privilege Level
- ■选择子放在段寄存器中

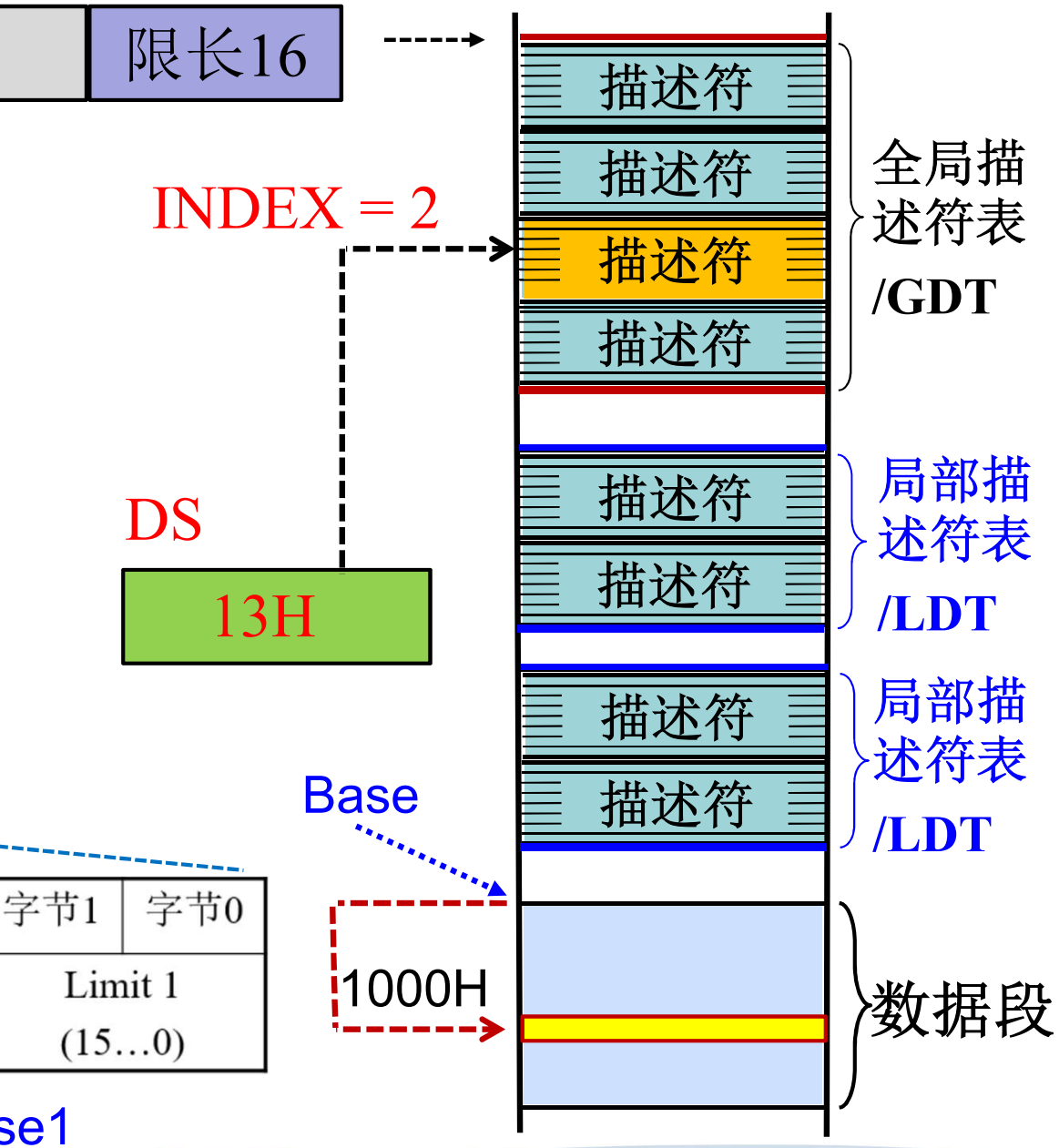
选择子(Selector)（回顾：保护模式内存寻址）

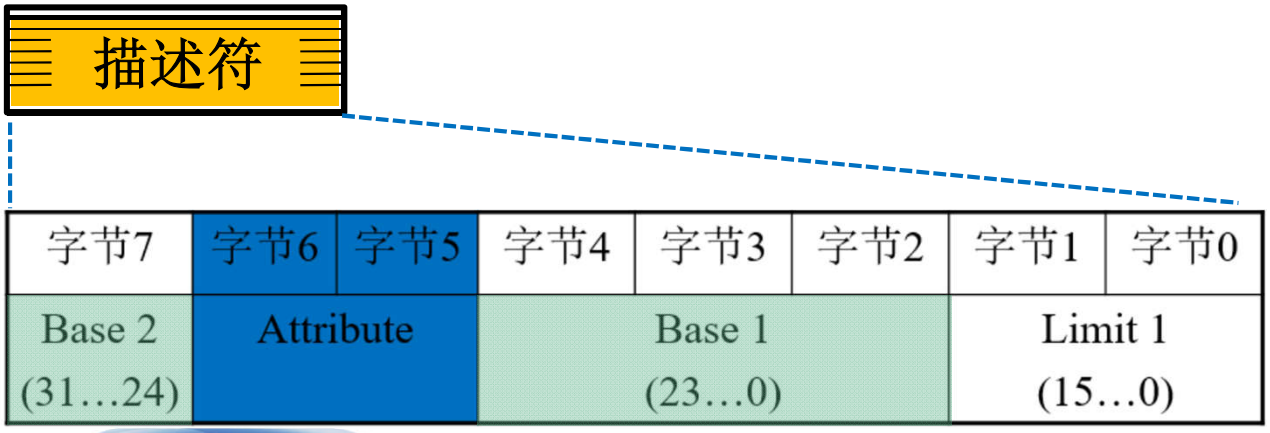
- 保模式寻址
  - 例:逻辑地址 = DS:1000H
    = 13H:1000H
  - 段基址=段基址(DS)
    =段基址(13H)
  - *描述符、描述表、类型？*
  - *DS含义？段寄存器含义？*



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？*

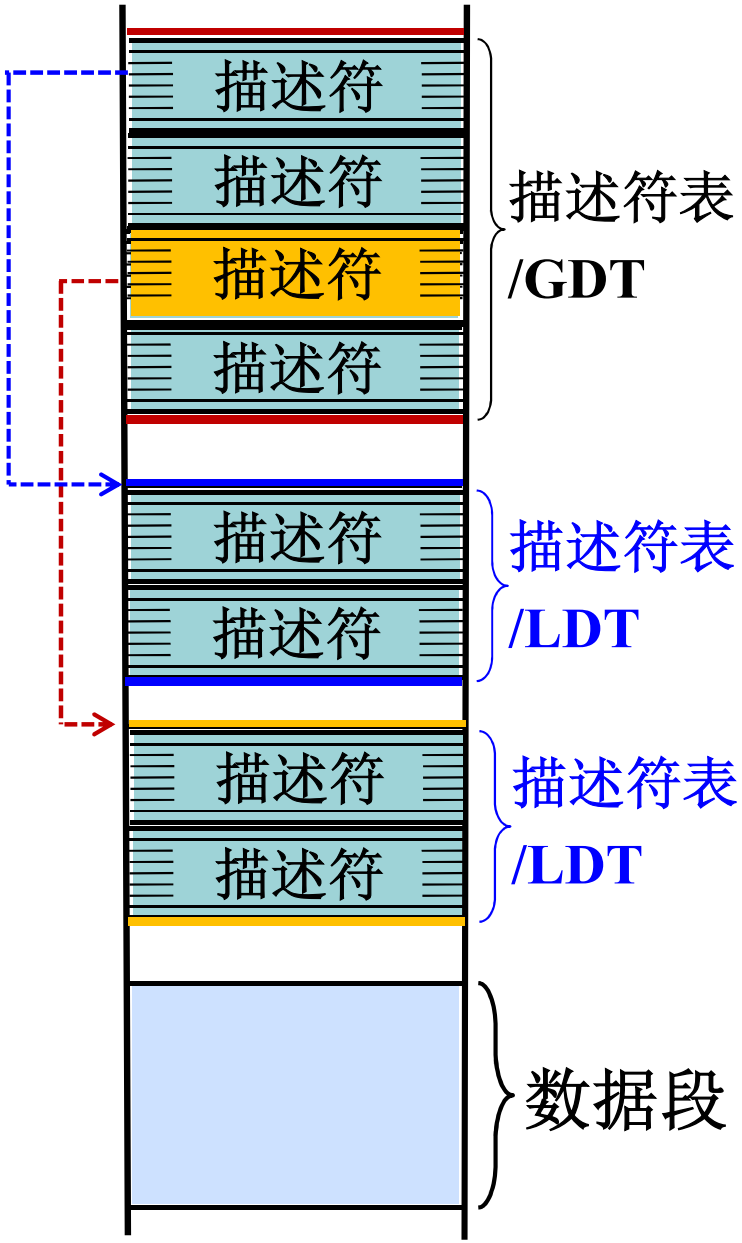




Base = Base 2 | Base1

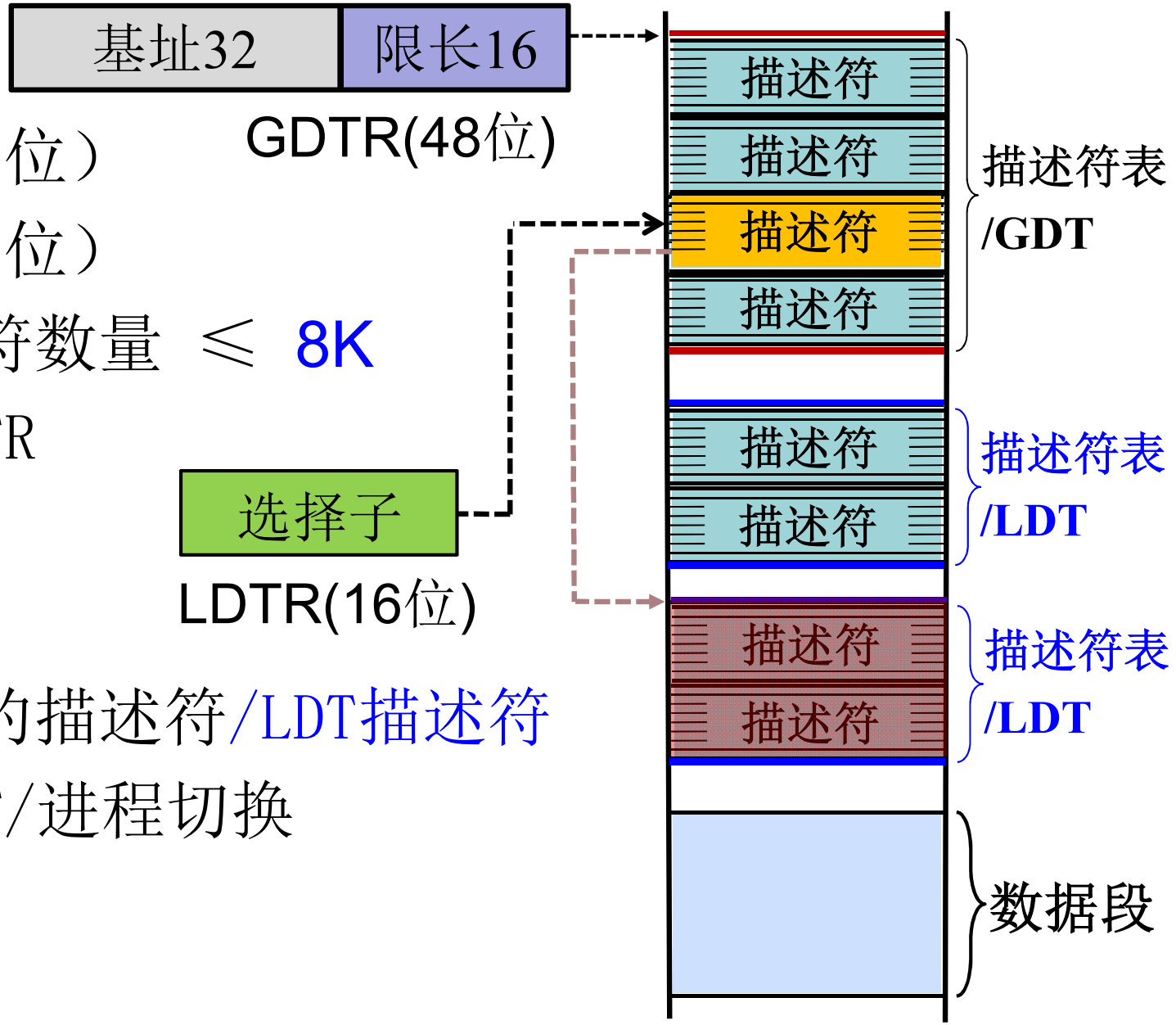
全局描述符表GDT与局部描述符表LDT

- 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



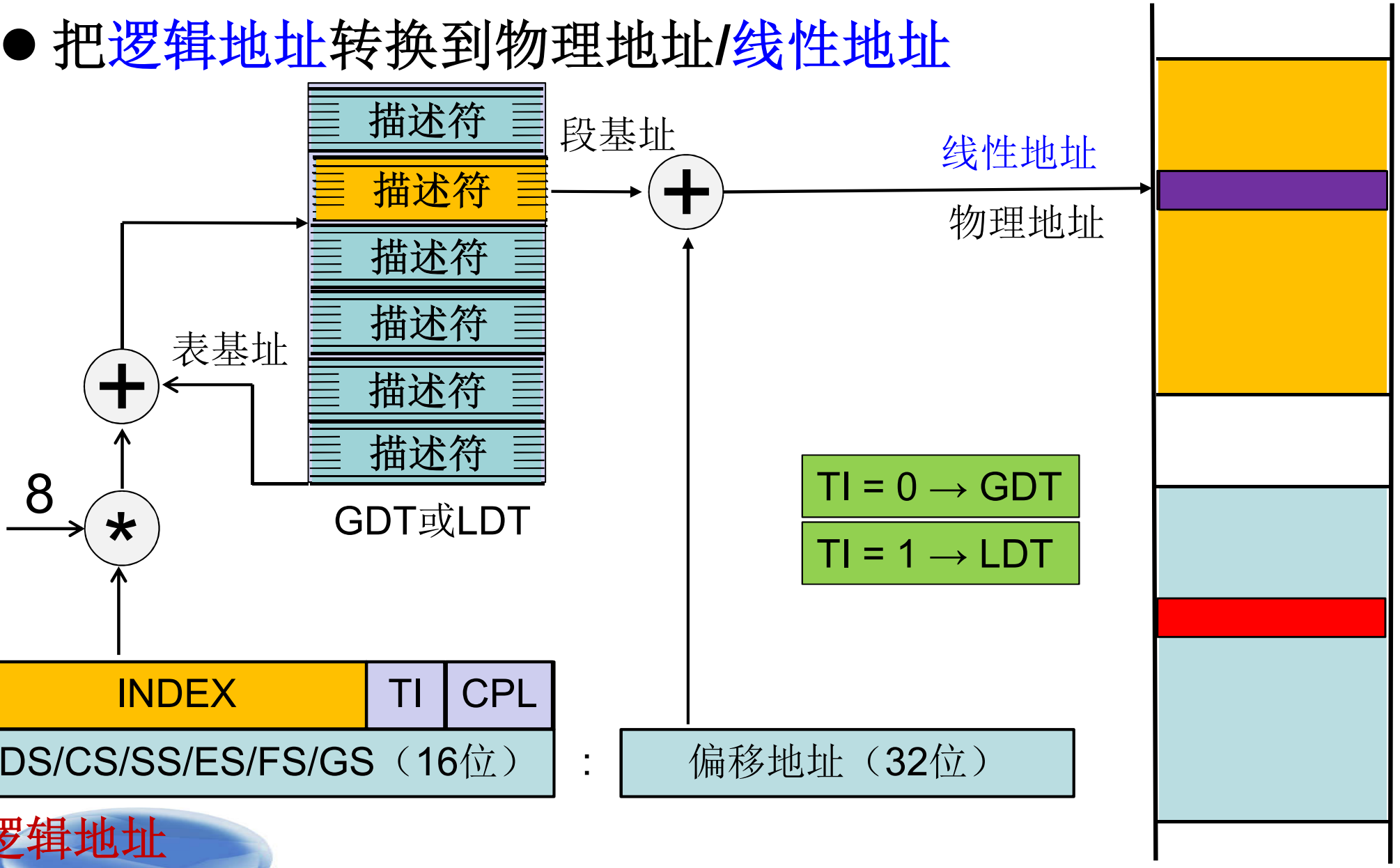
全局描述符表GDT与局部描述符表LDT

- GDTR（48位）
  - GDT的基址（32位）
  - GDT的限长（16位）
    - GDT中描述符数量 ≤ 8K
  - lgdtr:更新GDTR
- LDTR（16位）
  - 选择子
    - 选择GDT中的描述符/LDT描述符
  - lldtr:更新LDT/进程切换

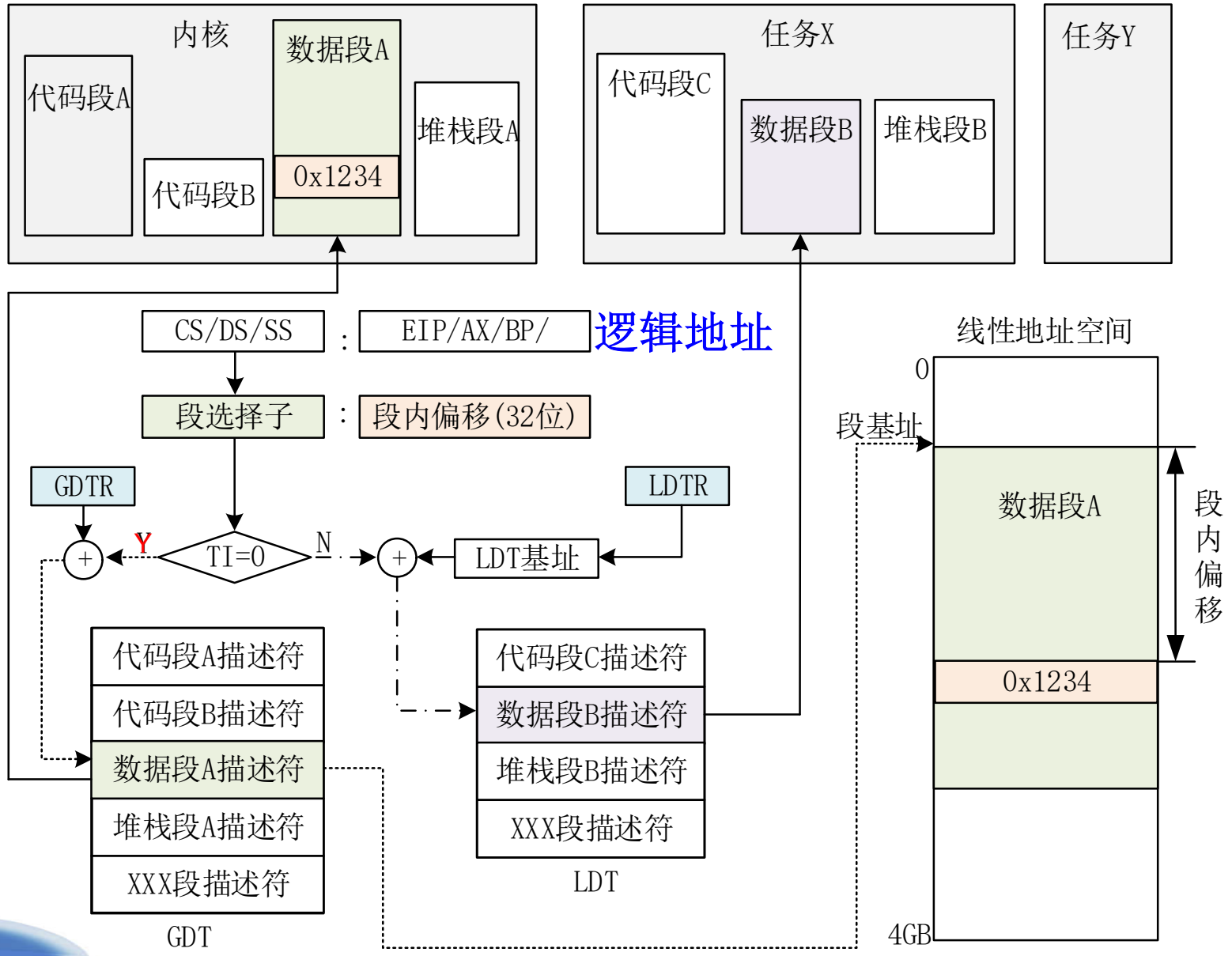


保护模式内存寻址：段式地址转换

- 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