《操作系统原理》

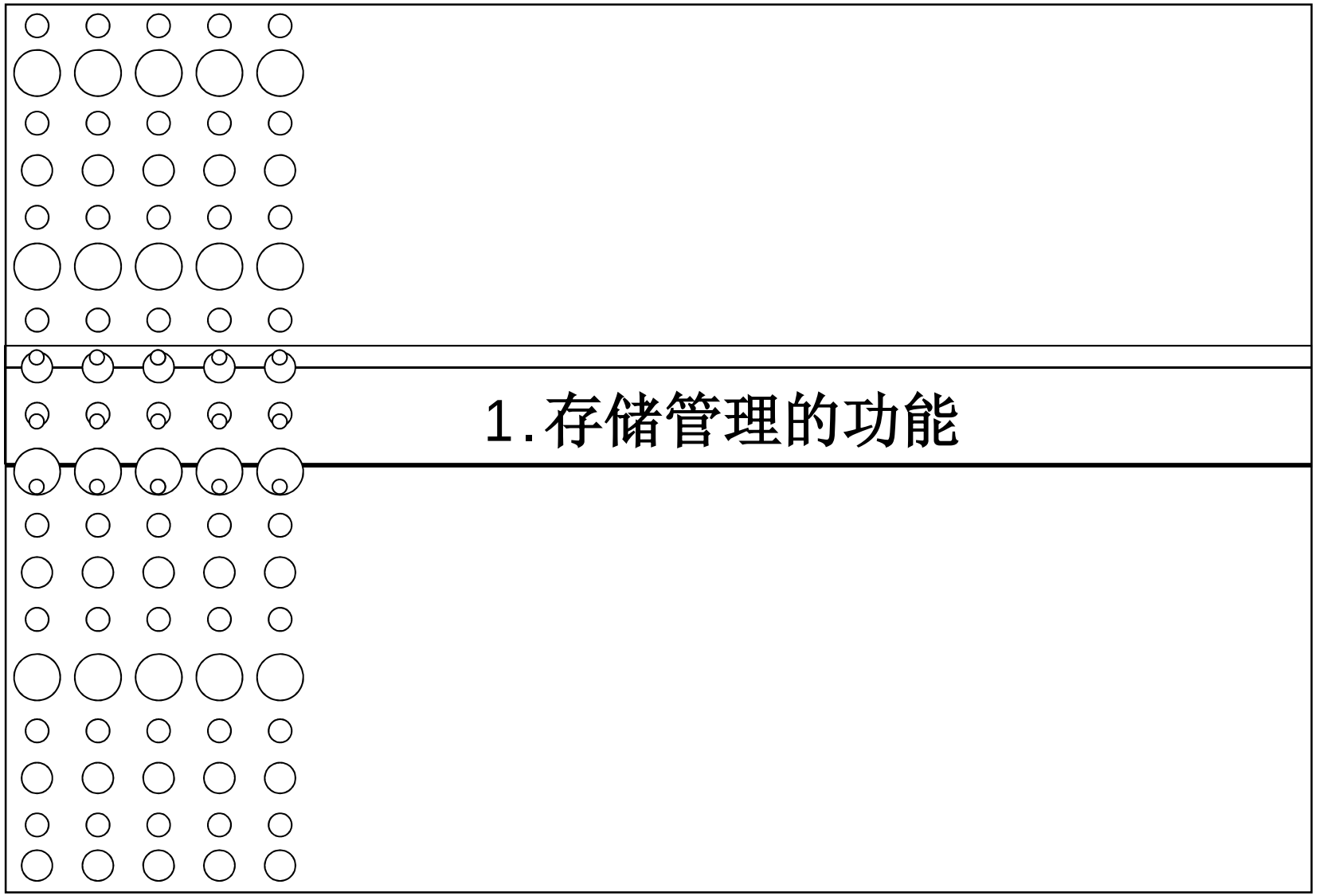
第7章 存储管理

教师：苏曙光

华中科技大学软件学院

2015年3月-5月

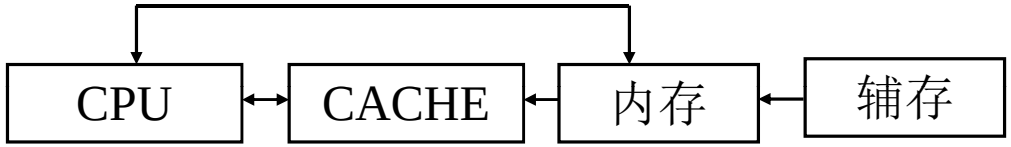
- 主要内容
  - 内存管理的功能
  - 物理内存管理
    - 分区存储管理
    - 覆盖技术
    - 对换技术
  - 虚拟内存管理
    - 页式存储管理
    - 段式存储管理
  - LINUX存储管理
- 重点
  - 地址映射概念
  - 虚拟存储概念
  - 页式存储管理原理



1.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

- **实际存储器体系**



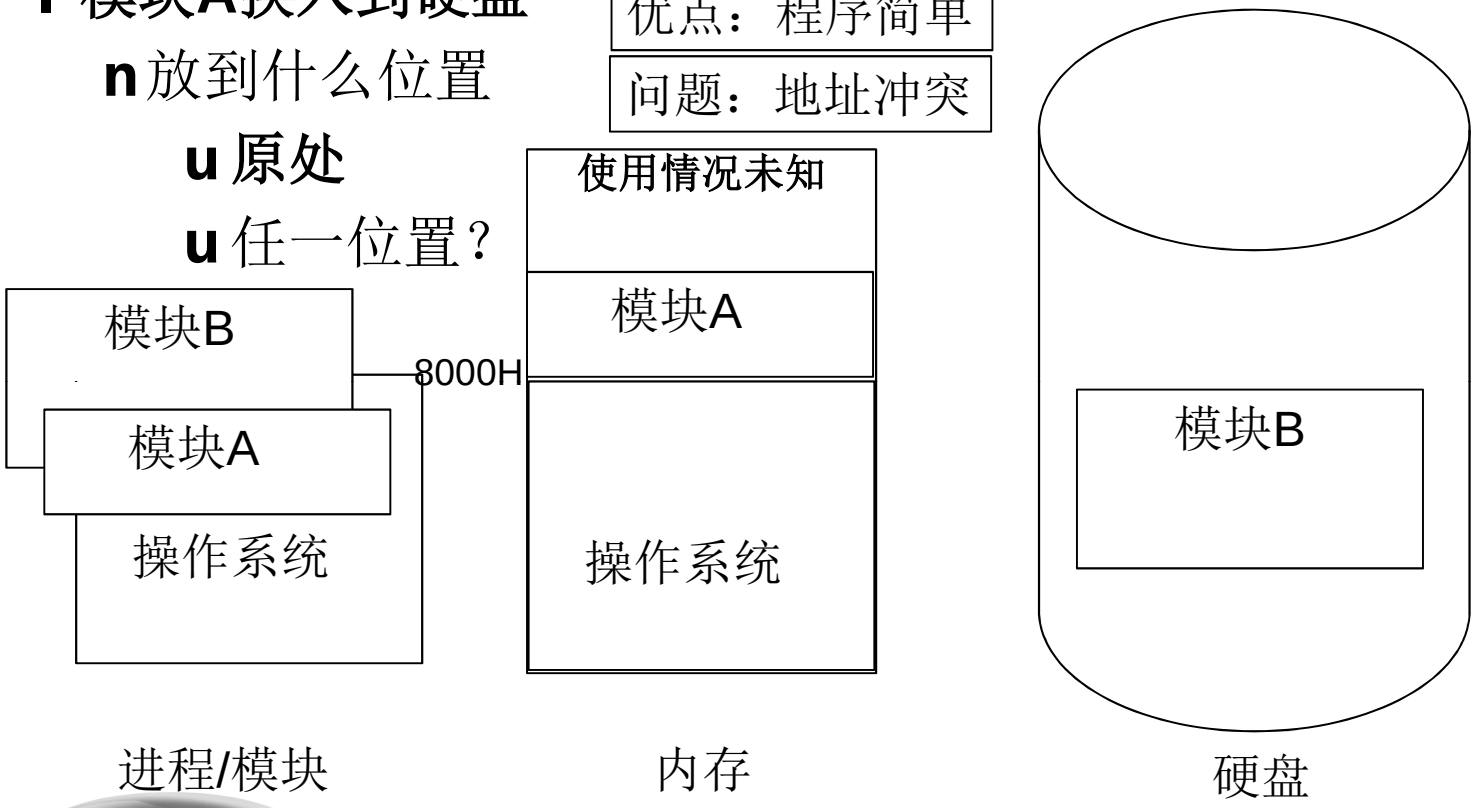
  - 三级存储体系
  - Cache(**快,小,贵**) +内存（**适中**）＋辅存（**慢,大,廉**）
  - 基本原理：
    - 当内存太小不够用时，用“辅存”来支援内存。
    - 暂时不运行的模块换出到辅存上，必要时再换入回内存。

换出与换入的讨论

- **模块A换入到硬盘**
  - **n** 放到什么位置
    - **u 原处**
    - **u** 任一位置？

优点：程序简单

问题：地址冲突

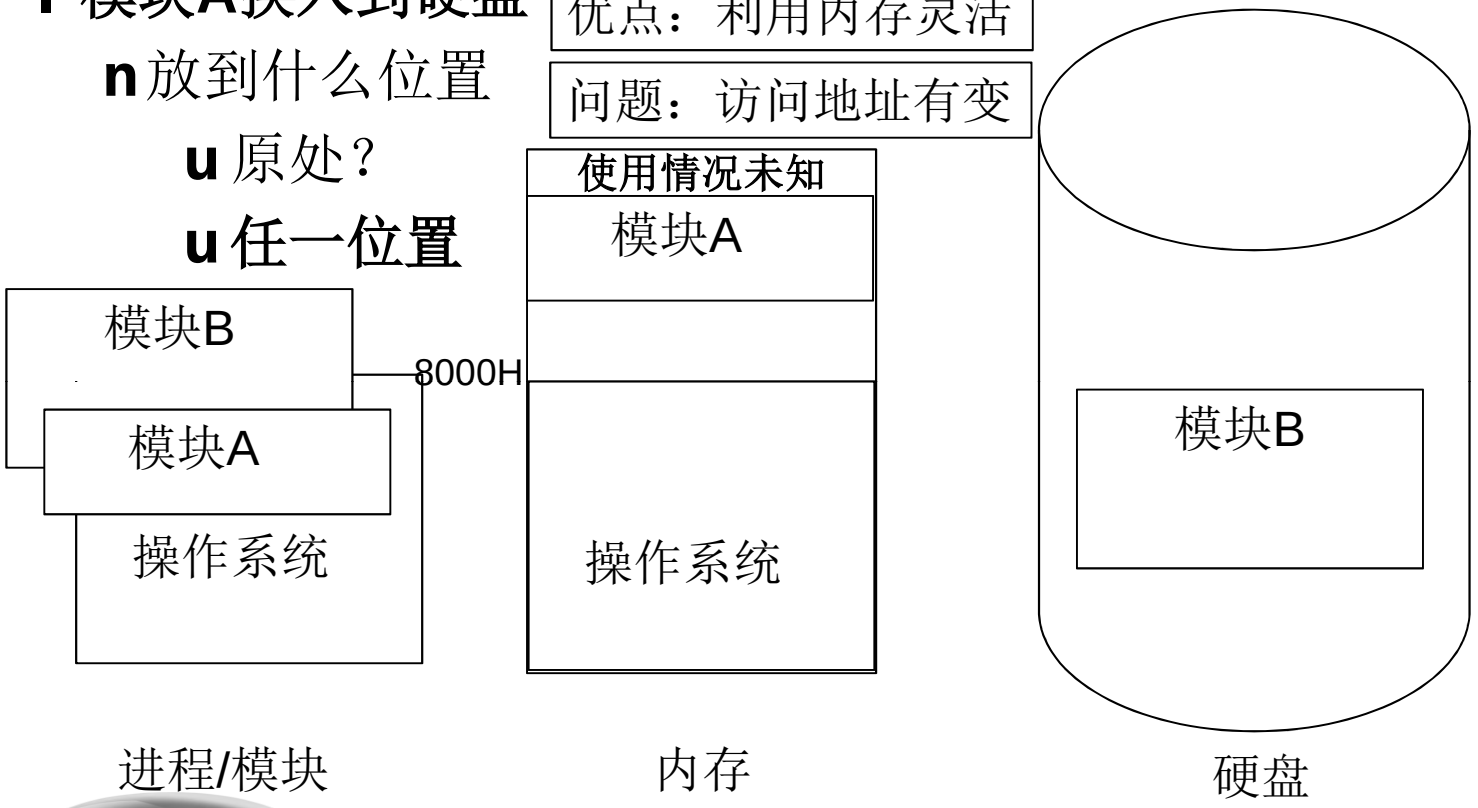


换出与换入的讨论

- **模块A换入到硬盘**
  - 放到什么位置
    - 原处？
    - **任一位置**

优点：利用内存灵活

问题：访问地址有变



- 进程或模块换出/换入需要解决的问题：访问地址有变
  - 地址重定位（地址重映射）
    - 辅存上的映像**换入**到内存时，需要重新确定放置地点，且把程序中访问的地址更新为**新地址**。

- **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

- **多道程序并行带来的问题**
  - **n** 共享
    - **u** 代码和数据共享，节省内存
  - **n** 保护
    - **u** 不允许内存中的**程序相互间**非法访问

- 存储管理的功能
  - 1）地址映射
  - 2）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
  - 把程序中的地址（虚拟地址,虚地址,逻辑地址）变换成真实的内存地址（实地址,物理地址）的过程。
  - 地址重定位，地址重映射
  - 源程序－逻辑地址－物理地址
- 方式
  - 固定地址映射
  - 静态地址映射
  - 动态地址映射

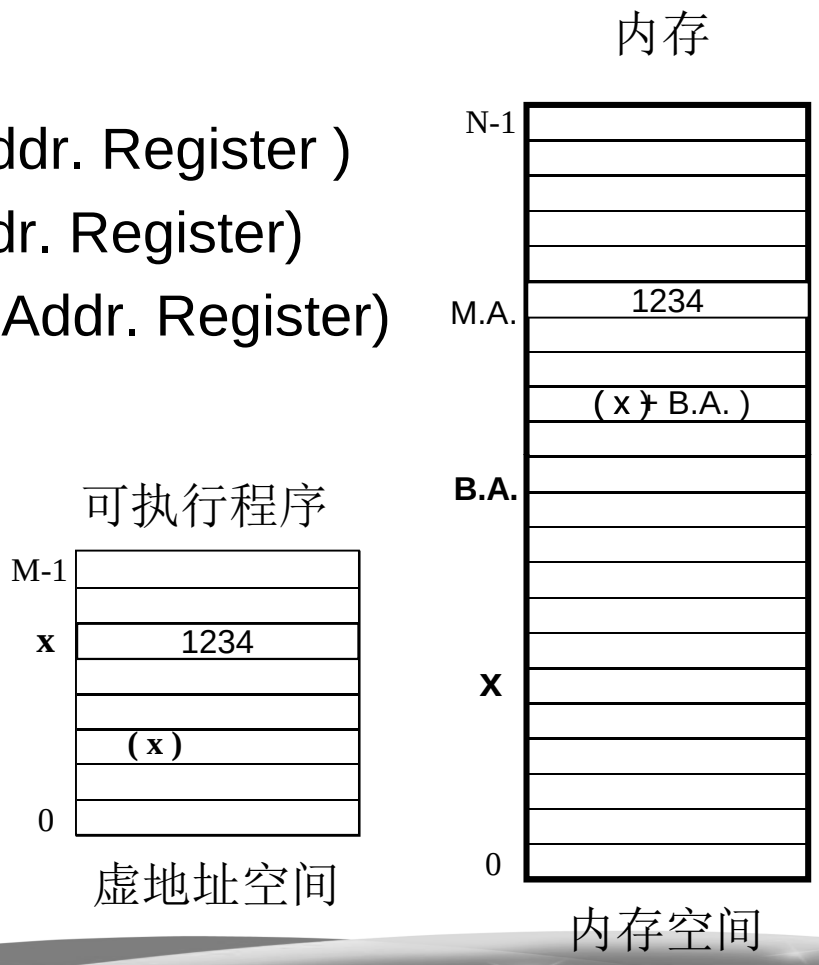
固定地址映射

- 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
  - 程序加载时必须放在指定的内存区域。
  - 容易产生地址冲突，运行失败。

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。

- **静态地址映射**
  - 逻辑地址：VA(Virtual Addr. Register)
  - 装入基址：BA(Base Addr. Register)
  - 物理地址：MA(Memory Addr. Register)
  - MA = BA + VA
- 例子
  - 逻辑地址：VA = X
    (X) = 1234
  - 装入基址：BA
  - MA = X + BA

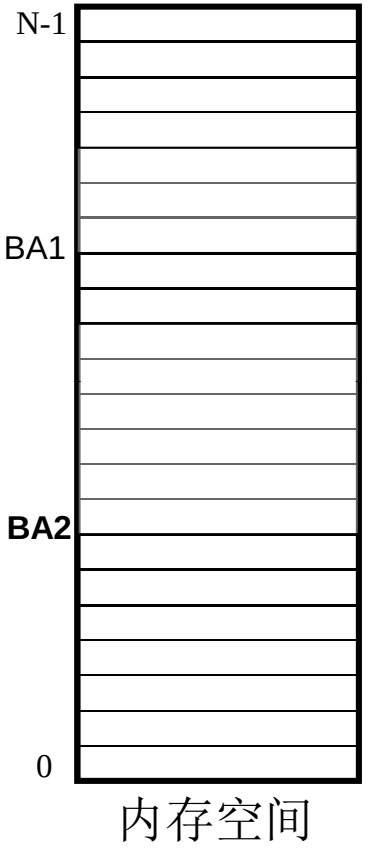


- **特点**
  - 程序运行之前确定映射关系
  - 程序装入后不能移动
    - 如果移动必须放回原来位置
  - 程序占用连续的内存空间

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。
    - 例如：MOV AX，[500]；访问500单元时执行地址转换
- 映射过程
  - 逻辑地址：VA(Virtual Addr. Register)
  - 装入基址：BA(Base Addr. Register)
  - 物理地址：MA(Memory Addr. Register)
  - MA = BA + VA
    - 注意：如果程序有移动，BA可能会有改变，自动计算新的MA。

- **特点**
  - 程序占用的内存空间可动态变化
  - 程序不要求占用连续的内存空间
    - 每段放置的基址系统应该知道
  - 便于多个进程共享代码
    - 共享代码作为独立的一段存放
  - 硬件改变后不需要重新编译程序
- **缺点**
  - 硬件支持（MMU：内存管理单元）
  - 硬件时延
  - 软件复杂



存储管理的功能：2）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（虚拟存储的基本原理）
  - 借助辅存在逻辑上扩充内存，解决内存不足
  - 把进程当前正在运行的部分装入内存（迁入），把当前不运行的部分暂时存放在辅存上（迁出），尽量腾出足够的内存供进程正常运行。
    - 辅存存放的部分当需要运行时才临时按需调入内存。
    - 进程不运行的部分往往占大部分,尤其是大进程。

- **程序局部性原理**
  - 时间局部性
    - 一条指令或数据，会在较短时间内被重复访问
      - 例如：循环语句
  - 空间局部性
    - 任一内存单元及其邻近单元会在短时间内被集中访问
    - 短时间内，CPU对内存的访问往往集中在一个较小区域内
    - 例如：表，数组的操作
  - 结论：
    - 程序在一个有限的时间段内访问的代码和数据往往集中在有限的地址范围内。因此，一般情况下，把程序的一部分装入内存在较大概率上也足够让其运行。

存储管理的功能：2）虚拟存储——

- 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2的解决方法（虚拟存储的定义）
  - 编程时不受内存容量和结构限制，认为内存是理想存储器（虚拟存储器）
    - 特点
      - 线性地址
      - 封闭空间
      - 容量足够大：2^{32}BYTE
      - 虚拟地址和虚拟地址空间
    - 工作原理
      - 运行时把虚拟地址转化为具体的物理地址：地址映射功能
      - 不同程序中的同一虚拟地址转化为不同的物理地址。
      - 内存管理单元：MMU：Memory Management Unit
    - 作用：实现虚拟地址和物理地址分离

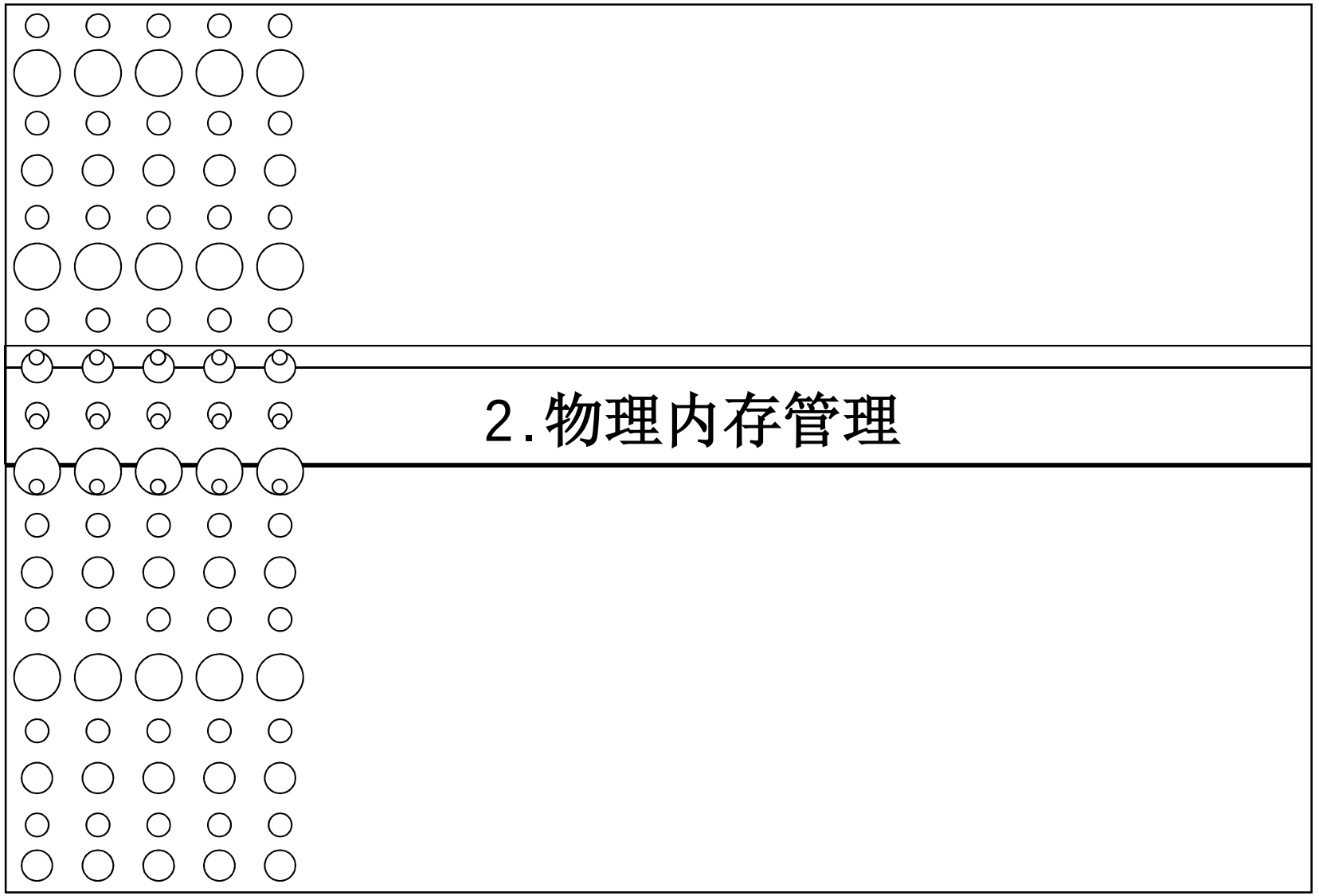
- **实现虚拟存储的前提**
  - 足够的辅存
  - 适当容量的内存
  - 地址变换机构
- **虚拟存储的应用**
  - 页式虚拟存储
  - 段式虚拟存储

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
- 需要解决的问题
  - 放置策略
    - 程序调入内存时将其放置在哪个/哪些内存区
  - 调入策略
    - 何时把要运行的代码和要访问的数据调入内存？
  - 淘汰策略
    - 内存空间不够时，迁出（/淘汰）哪些代码或数据以腾出内存空间。

存储管理的功能：4）存储保护功能

- 保证在内存中的多道程序只能在给定的存储区域内活动并互不干扰。
  - 防止访问越界
  - 防止访问越权
- 方法：界址寄存器
  - 在CPU中设置一对下限寄存器和上限寄存器存放程序在内存中的下限地址和上限地址
    - 程序访问内存时硬件自动将目的地址与下限寄存器和上限寄存器中存放的地址界限比较，判断是否越界。
  - 基址寄存器和限长寄存器



2.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。
    - 例如：DOS
- 优点
  - 简单，不需复杂硬件支持，适于单用户单任务OS
- 缺点
  - 程序运行占用整个内存，即使小程序也是如此
    - 内存浪费，利用率低

分区存储管理

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

固定分区

- 定义
  - 把内存固定地划分为若干个大小不等的分区供各个程序使用。每个分区的大小和位置都固定，系统运行期间不再重新划分。
  - 分区表
    - 记录分区的位置、大小和使用标志

固定分区的例子

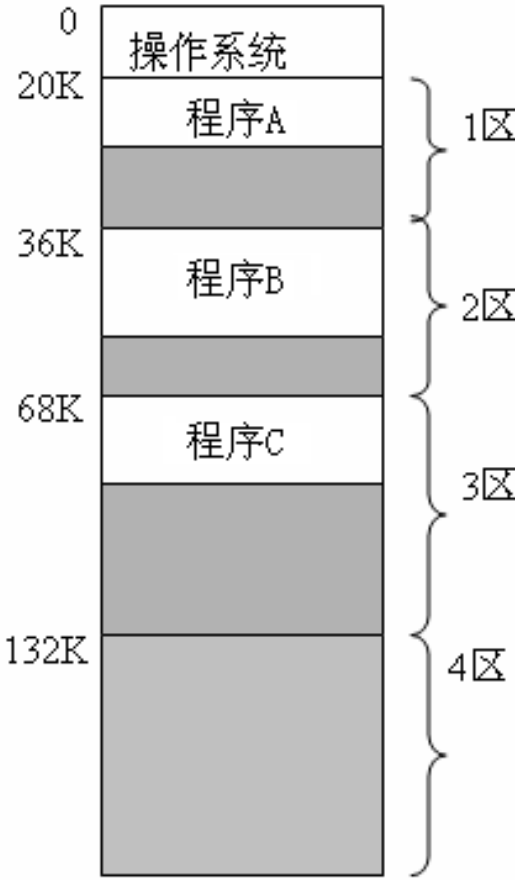
- 4个分区的例子
  - 3个程序在占用
  - 分区表

分区表

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

被占用的空间

未占用的空间



固定分区

- 使用特点
  - 在程序装入前，内存已被分区，不再改变。
  - 每个分区大小可能不同，以适应不同大小的程序。
  - 系统维护分区表，说明分区大小、地址和使用标志
  - 例如：IBM的OS/360采用了固定分区方法。
    - 具有固定任务数的多道程序系统

固定分区

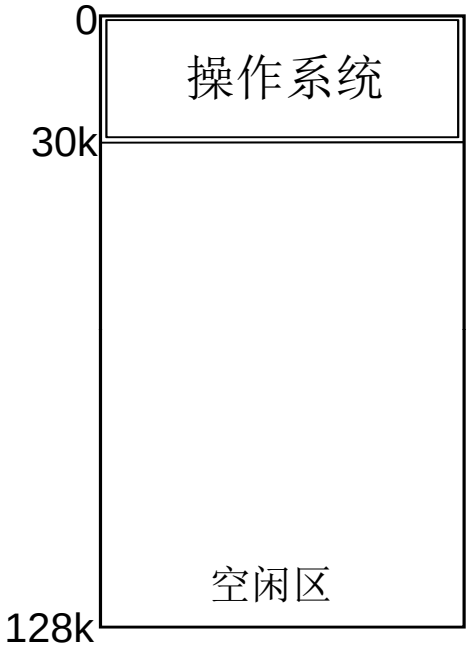
- **固定分区的性能**
  - 当程序比所在分区小时，**浪费内存**
  - 当一个程序比最大分区大时，无法装入运行
- 建议措施
  - 根据**分区表**安排程序的**装入顺序**，使得每个程序都能找到合适的分区运行。
  - 当程序的大小、个数、装入顺序等都**固定**时，内存使用**效率很高**。

动态分区

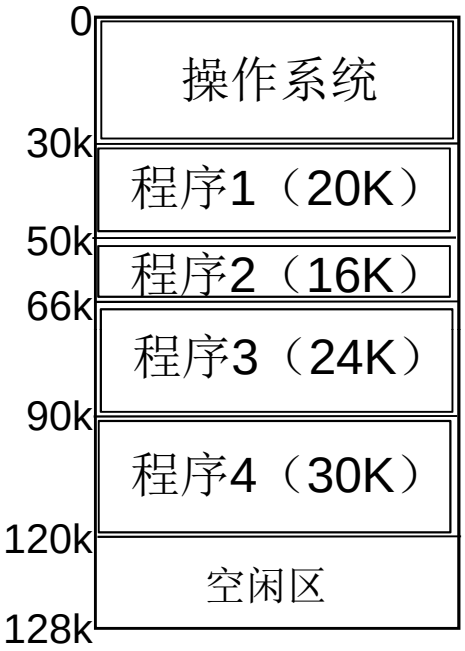
- 定义
  - 在程序装入时创建分区，使分区的大小刚好与程序的大小相等。
  - 解决固定分区浪费内存和大程序不能运行的问题。
- 特点
  - 分区动态建立

动态分区例子

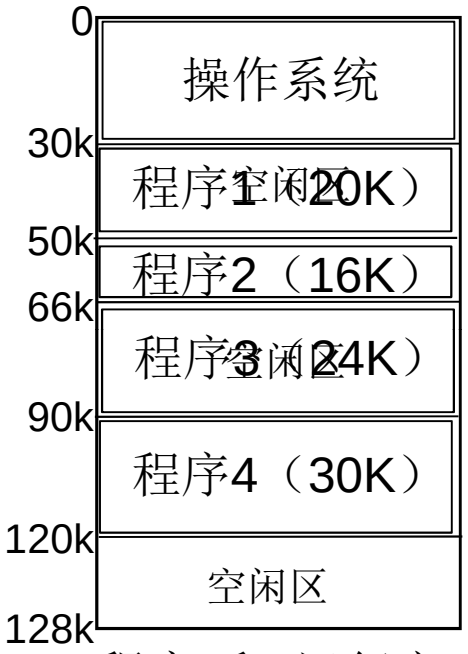
- 程序1（20K）;程序2（16k）;程序3（24k）;程序4（30K）



内存初始状态



动态划分分区

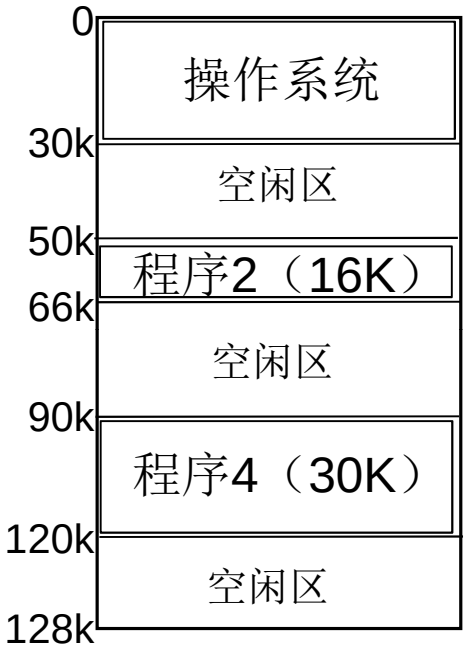


程序1和3运行完；

程序撤出收回内存

动态分区例子

- 程序1（20K）;程序2（16k）;程序3（24k）;程序4（30K）

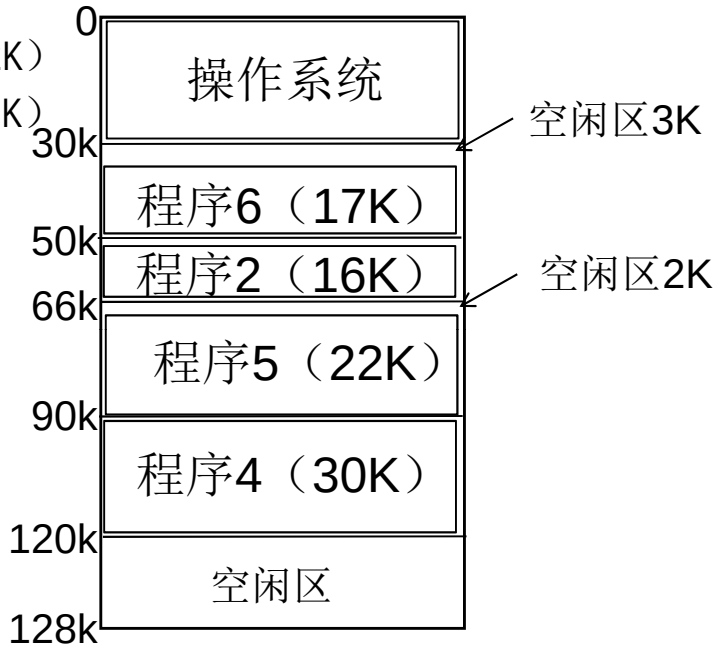


程序1和3运行完；

程序撤出收回内存

•程序5（22K）

•程序6（17K）



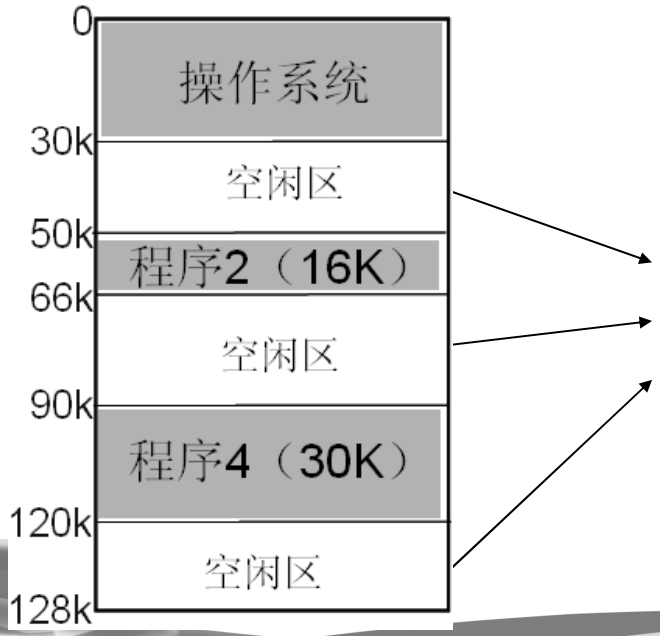
内存碎片：过小的空闲区，难实际利用

动态分区

- **特点**
  - 分区的个数和大小均可变
  - 存在内存碎片
- **动态分区需要解决的问题**
  - 分区的分配和选择？
  - 分区的回收？
  - 解决内存碎片问题？

分区的分配

- 功能
  - 在所有空闲区中寻找一个**空闲区**，分配给用户使用。
  - 基本要求：空闲区的大小应满足用户要求。
- **空闲区表：描述内存空闲区的位置和大小的数据结构**



空闲区表

位置	大小
30K	20K
66K	24K
120K	8K

分区的分配

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

空闲区表

位置	大小
30K	20K
66K	24K
120K	8K

分区的选择——放置策略

- **放置策略**
  - 选择空闲分区的策略
- **空闲区表的排序原则**
  - 按空闲区位置（首址）递增排序
  - 按空闲区位置（首址）递减排序
  - 按空闲区大小的递增排序
  - 按空闲区大小的递减排序
- **常用的放置策略**
  - 首次匹配（首次适应算法）
  - 最佳匹配（最佳适应算法）
  - 最坏匹配（最坏适应算法）

空闲区表

位置	大小
66K	24K
30K	20K
120K	8K

大小的递减

首次适应法

空闲区表

位置	大小
30K	20K
66K	24K
120K	8K

- **前提**
  - 空闲区表按**首址递增**排序
- **算法**
  - 从空闲区表的第1个分区开始查找，直到找到第一个满足用户要求的分区为止。
  - 优点尽可能地先利用低地址空间，保证高地址空间有较大的空闲区，当大程序需要较大分区时，满足的可能性很大。

最佳适应法

位置	大小
120K	8K
30K	20K
66K	24K

- **前提**
  - 空闲区表按**大小递增**排序
- **算法**
  - 从空闲区表的第1个分区开始查找，直到找到第一个满足用户要求的分区为止。
- **优点**
  - 选中分区是满足要求的最小的空闲区，尽量保留较大空闲区，当大程序需要较大分区时，满足的可能性很大。
- **缺点**
  - 容易出现内存碎片

最坏适应法

位置	大小
66K	24K
30K	20K
120K	8K

- **前提**
  - 空闲区表按**大小递减**排序
- **算法**
  - 从空闲区表的第1个分区开始查找，直到找到第一个满足用户要求的分区为止。
- **优点**
  - 最大的空闲区分割后的剩下部分还可能相当大，还能装下较大的程序。
- **特点**
  - 仅作一次查找就可找到所要分区。

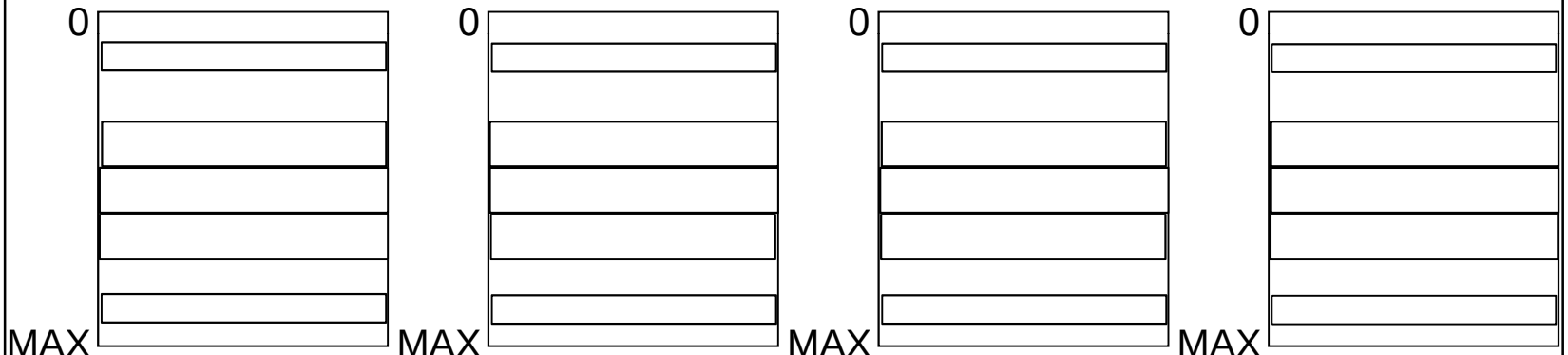
分区的回收

- 功能
  - 回收程序结束后所释放的分区（**释放区**），将其**适当处理**后登记到空闲区表中，以便再分配。
  - 要考虑**释放区**与现有**空闲区是否相邻**？
- 回收算法
  - 若**释放区**与**空闲区**不相邻，则把释放区直接插入空闲区表。
  - 若**释放区**与**空闲区**相邻，则把释放区和空闲区**合并后作为新的更大的空闲区**插入空闲区表。

位置	大小
30K	20K
66K	24K
120K	8K

l 具体的回收过程

- n □当前的空闲区
- n □释放区
- n □占用区



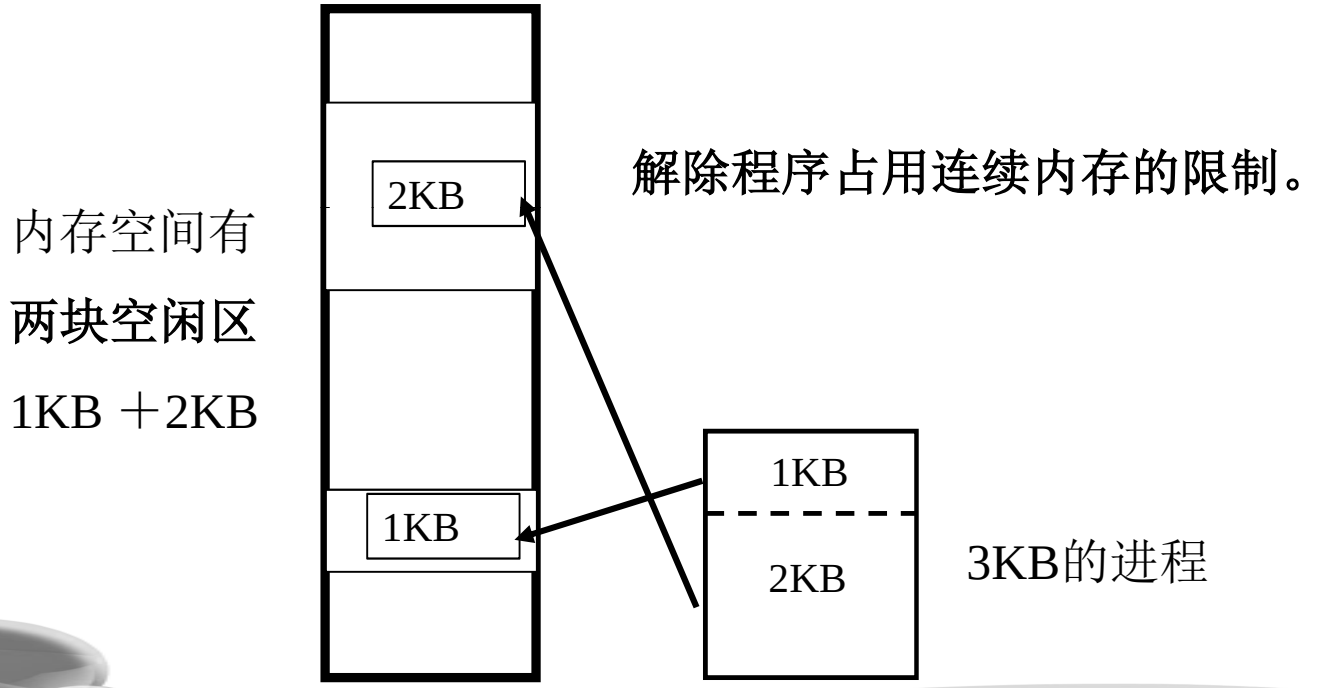
碎片问题

- 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
    - 定期清理存储空间，将所有空闲区集中一起。
    - 存储器紧缩技术

- **拼接的时机**
  - 分区回收的时候
    - 拼接频率过大，系统开销大
  - 系统找不到足够大的分区时
    - 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

l 解决碎片的办法（续）

- n 把程序分成几个部分装入不同的分区中，化整为零，以便充分利用小的碎片。



内存空间有

两块空闲区

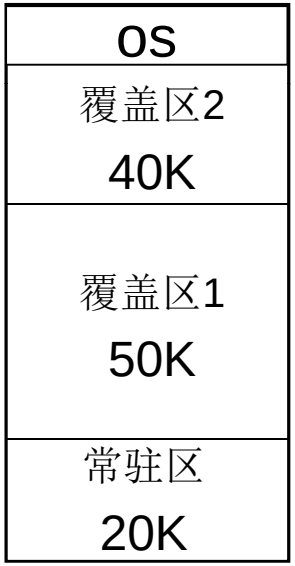
1KB ＋2KB

解除程序占用连续内存的限制。

3KB的进程

覆盖——Overlay

- 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：被某段单独占用的区域，可划分多个
  - 覆盖区： 能被多段**重复共用**（**覆盖**）的区域， 可划分多个



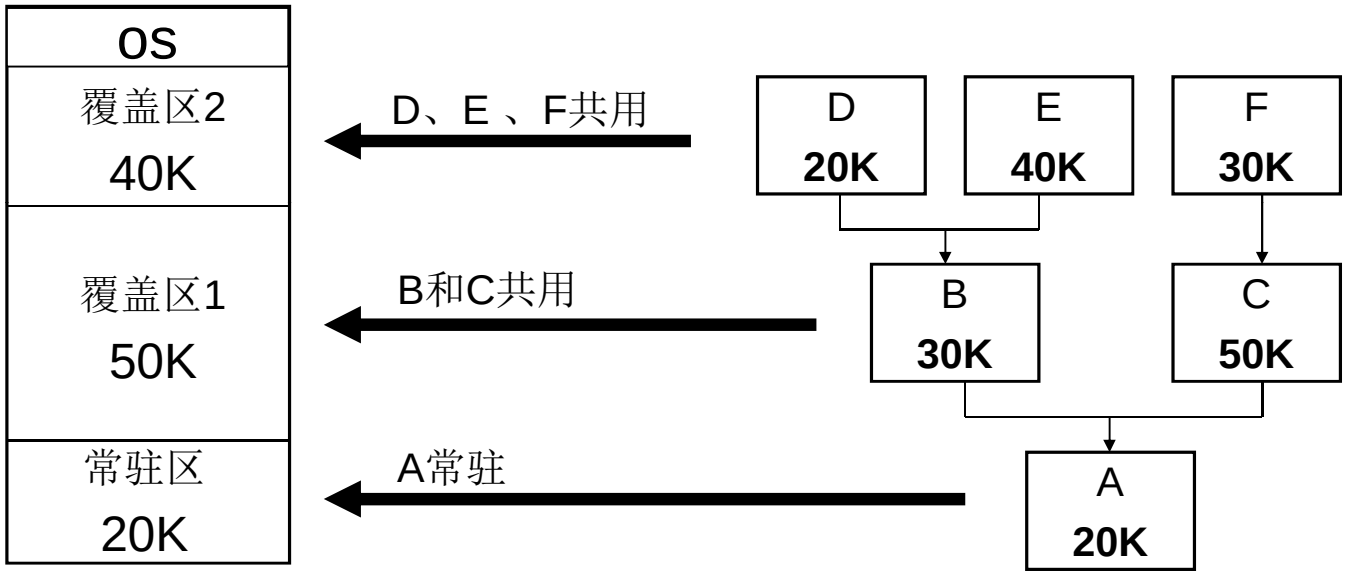
用户内存（110K）

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的内存；（核心段）
  - 目前正运行的的段处于覆盖区中；
  - 将目前不用的段或用过的段放在硬盘的特殊区域中(覆盖文件)；
  - 临时要用的段从硬盘装入**覆盖区**的内存（**覆盖**不再用的内容）；
    - 意义盖区，减少程序对内存的需求

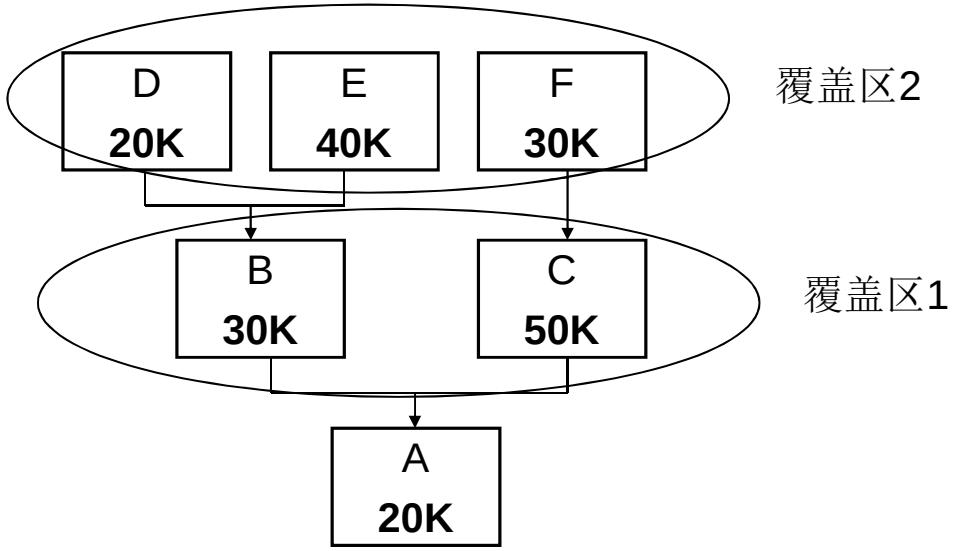
覆盖的例子

- 内存（110K）：一个常驻区，两个覆盖区
- 程序（190K）：多个模块（段）



l 覆盖的缺点

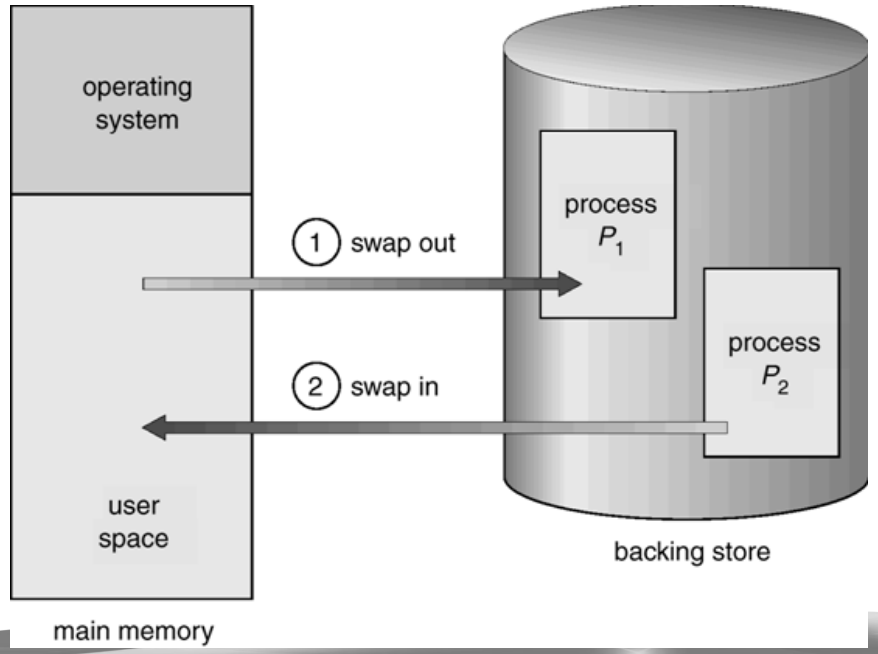
- n 编程复杂：程序员划分程序模块并确定覆盖关系。
- n 程序执行时间长：从外存装入内存耗时



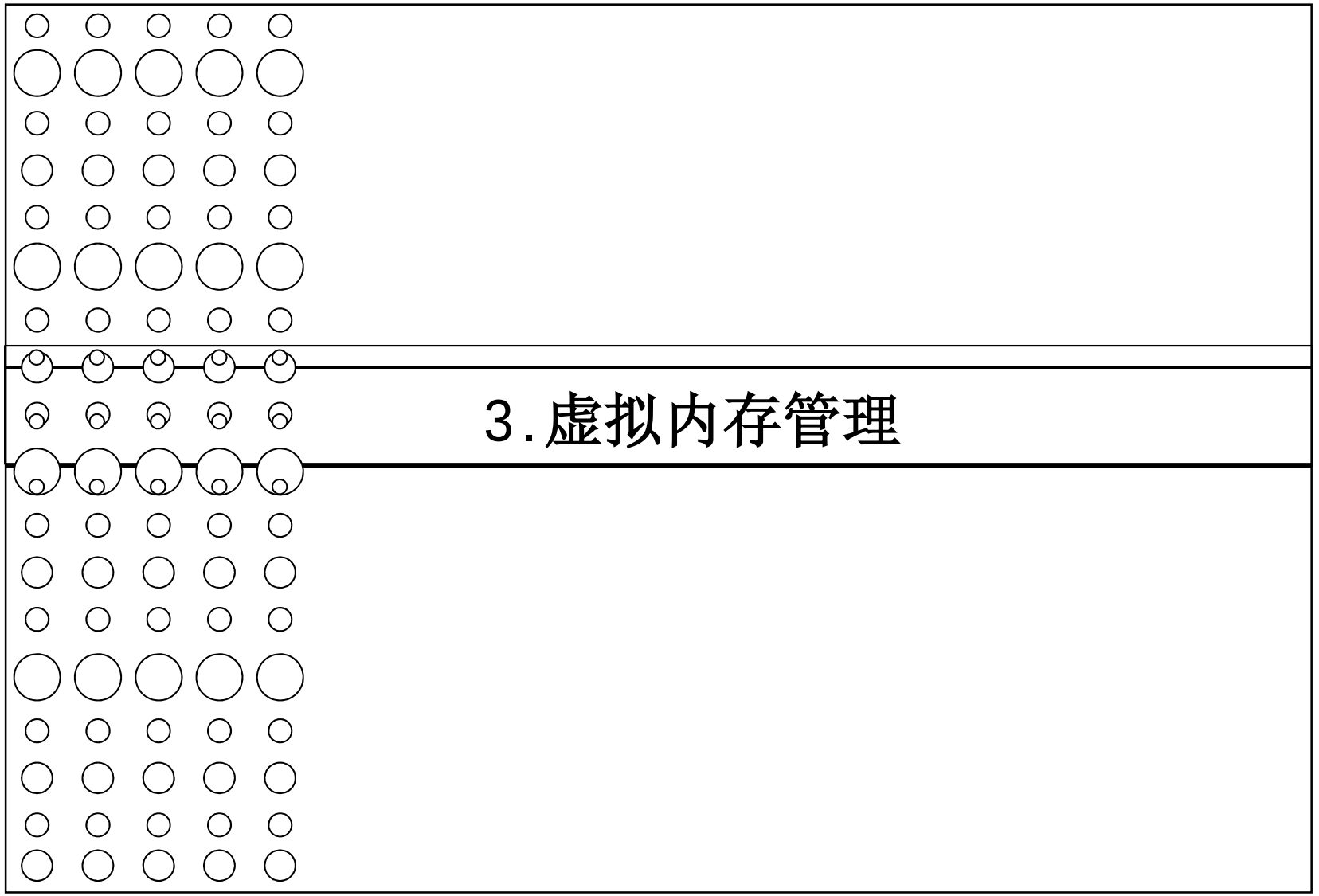
用户程序（190K）

对换技术——Swapping

- 原理
  - 当内存不够时，把进程写到磁盘上（换出，**Swap Out**），以便腾空内存。当进程要运行时，再把其重新写回到内存（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 3.1**



3.虚拟内存管理

虚拟内存的概念

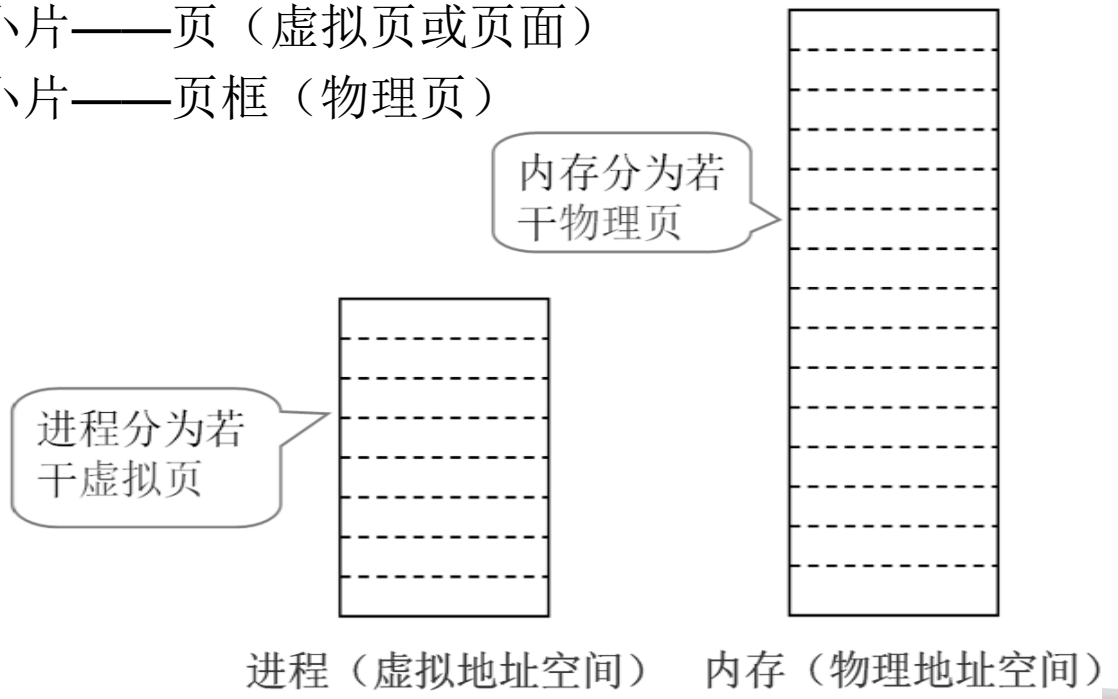
- 虚拟内存是面向用户的虚拟封闭存储空间。
  - 线性地址空间。
  - 容量4G = 2^{32} Byte
  - 封闭空间（进程空间）
  - 和物理地址分离（地址无冲突）
  - 程序员编程时使用线性虚拟地址

- **虚拟内存管理的目标**
  - 使得大的程序能在较小的内存中运行；
  - 使得多个程序能在较小的内存中运行（/能容纳下）；
  - 使得多个程序并发运行时地址不冲突（/方便,高效）；
  - 使得内存利用效率高：无碎片,共享方便
- **程序运行的局部性**
  - 程序在一个有限的时间段内访问的代码和数据往往集中在有限的地址范围内。
  - 把程序一部分装入内存在较大概率上也足够让其运行一小段时间。

- **典型虚拟内存管理方式**
  - 页式虚拟存储管理
  - 段式虚拟存储管理
  - 段页式虚拟存储管理

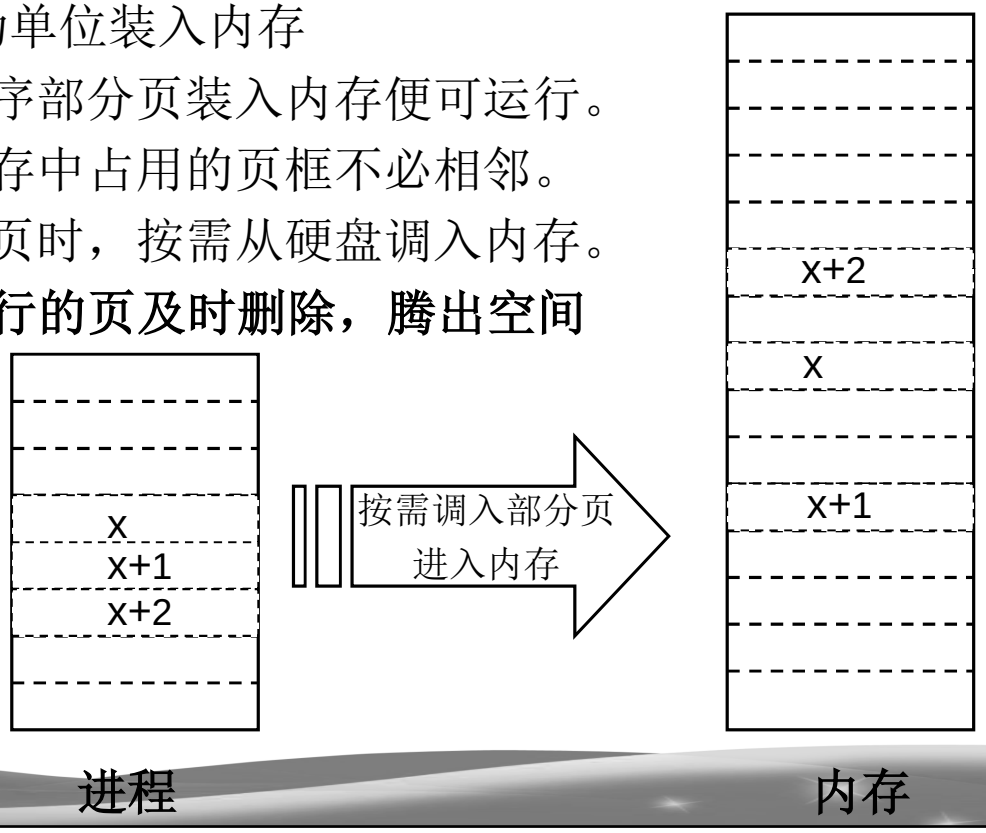
页式虚拟存储管理

- **概念**
  - 把进程空间（**虚拟**）和内存空间都划分成等大小的小片
    - 小片的典型大小：1K，2K或4K…
    - 进程的小片——页（虚拟页或页面）
    - 内存的小片——页框（物理页）



l 进程装入和使用内存的原则

- **n** 内存以**页框**为单位分配使用。
- **n** 进程以页为单位装入内存
  - **u** 只把程序部分页装入内存便可运行。
  - **u** 页在内存中占用的页框不必相邻。
  - **u** 需要新页时，按需从硬盘调入内存。
  - **u** **不再运行的页及时删除，腾出空间**



页式系统中的地址

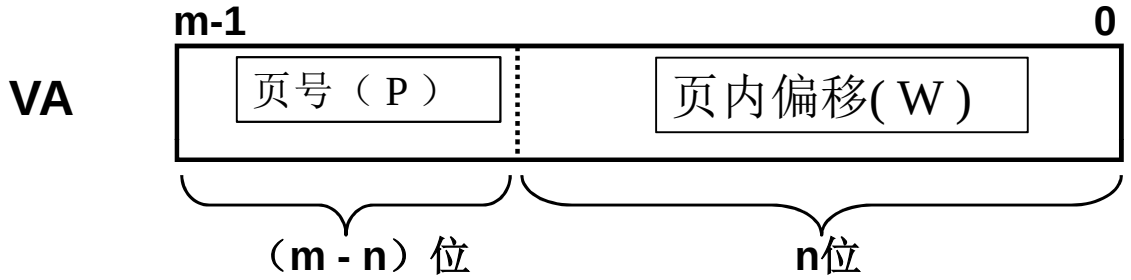
- **虚拟地址(VA) 可以分解成页号P和页内偏移W**
  - **n** 页号（P）
    - **u** VA所处页的编号 = VA / 页的大小
  - **n** 页内偏移(W)
    - **u** VA在所处页中的偏移 = VA % 页的大小
- **例子**
  - **n** VA = 2500; 页面大小1K(1012)

    P = 2500 / 1024 = 2

    W = 2500 % 1024 = 452

P和W的另一种计算方法

- 已知
  - 虚拟地址的宽度：m位
  - 页的大小：2^n单元 (显然m > n)



- P和W计算
  - 页号P = 虚拟地址的高（m - n）位 = VA >> n
  - 页内偏移W = 低 n 位 = VA && 2^n

地址映射

- 页面映射表
- 地址映射过程

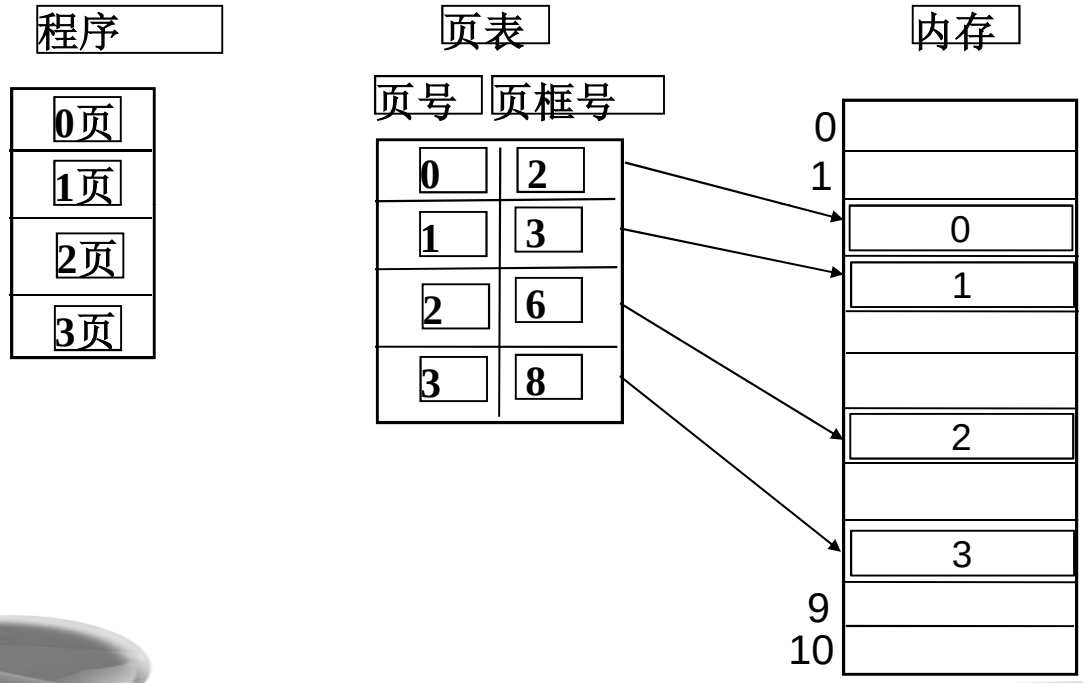
- **页面映射表**
  - 记录页与页框之间的对应关系。也叫页表。

页号	页框号	页面其它特性
0	5	…
1	65	…
2	13	…

- **页表结构**
  - 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。

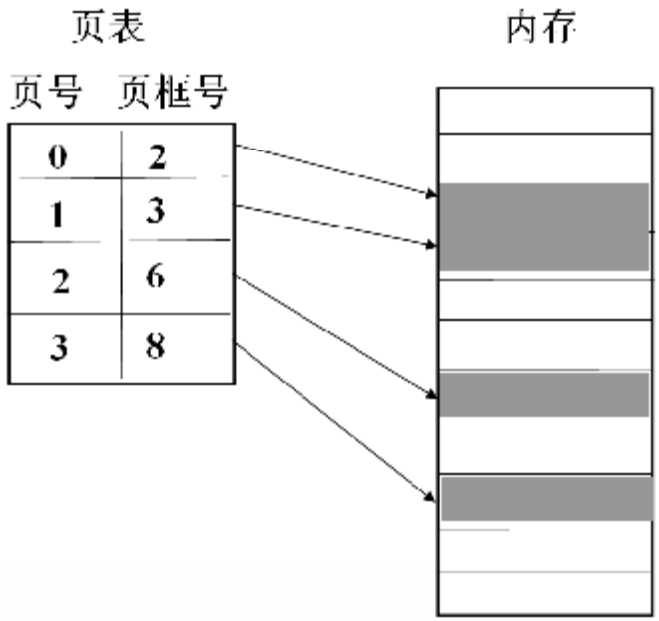
页表例子

- 一个进程：4页



页式地址映射

- 功能
  - 虚拟地址（页式地址）→物理地址
- 过程【三步】
  - 1.从VA分离页号P和页内偏移W；
  - 2.查页表：以P为索引查页框号P’；
  - 3.计算物理地址MA
    - MA = P’ x 页大小+ W



- 页式地址映射例子
  - MOV R1, [2500]
- 解：

页号	页框号	其它特性
0	4	…
1	2	…
2	7	…

1. 分离P，W.

   P = VA / 页面大小 = 2500 / 1024 = 2

   W = VA % 页面大小 = 2500 % 1024 = 452

2. 查找页表

   P = 2, P’ = 7

3. 计算MA = P’ x页面大小 + W

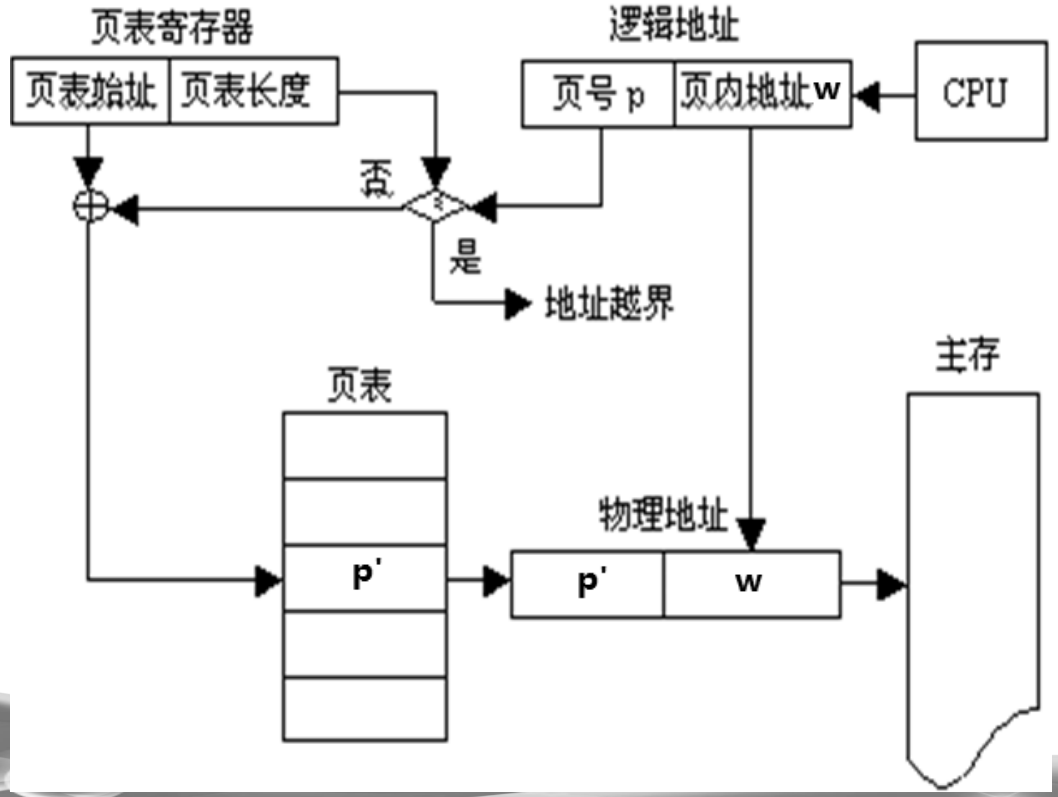
   MA = 7 * 1024 + 452 = 7620

防止越界访问页面

- **防止越界访问其它进程**
  - 检查访问的目的页号X是否在进程内？
    - 检查标准：0 < = X < 虚拟页数
    - 越界：产生越界中断

页式地址映射

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



页面的存取权限特性

- 页面映射表

页号	页框号	其它特性
0	5	…
1	65	…
2	13	…

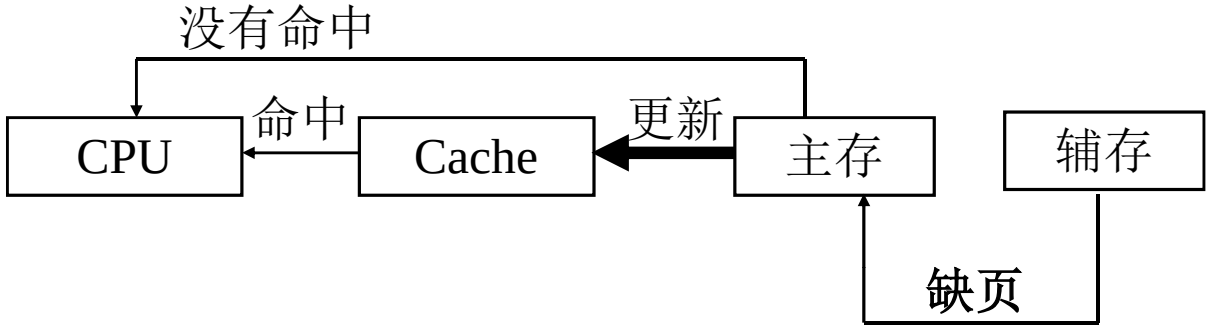
- 其它特性
  - 存取权限
  - 是否被访问过
  - 是否被修改过
  - ……

页面的存取权限

- **防止越权访问某些页面**
  - 在页表中记录每个页面的读、写、执行等权限。
  - 当访问页面时首先检查该次访问是否越权。
- **防止越界访问其它进程**
  - 检查访问的目的页号X是否在进程内？
    - 检查标准：0 < = X < 虚拟页数
    - 越界：产生越界中断

快表机制（Cache）

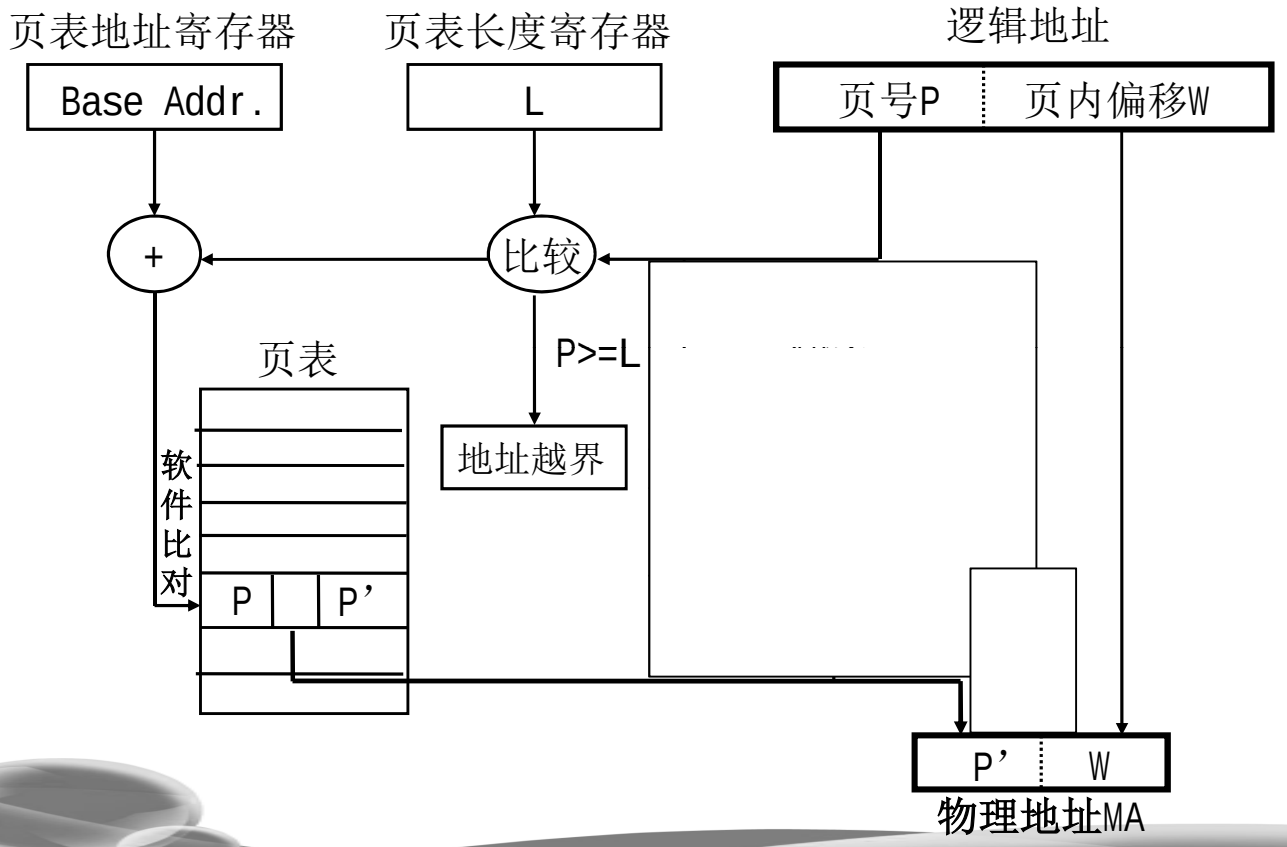
- 分级存储体系
  - CACHE + 内存 + 辅存



  - 页表放在Cache中：快表
  - 页表放在内存中：慢表

- **快表的特点**
  - 访问速度快，成本高，容量小,具有并行查寻能力。
  - 快表是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

快表机制下地址映射过程



例题

- **对于利用快表且页表存于内存的分页系统，假定CPU的一次访问内存时间为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平均有效时间是多少？**
- **分析：**
  - **（1）若直接通过快表完成，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%**

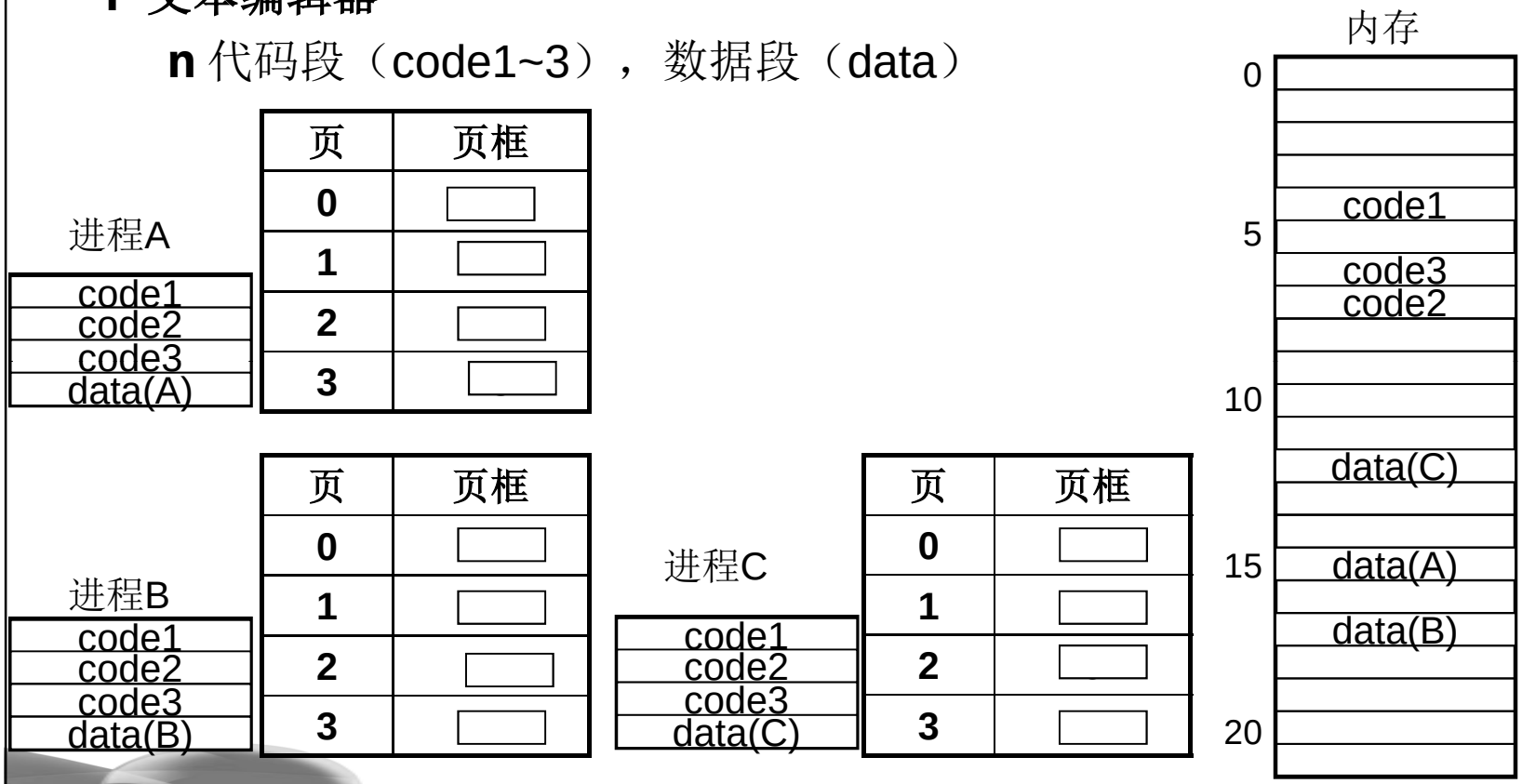
    = 1.15μs

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进程并发执行该文本编辑器。
  - **占用内存** = 10 x（**150**＋50）KB ＝ 2M
  - 如果采用代码段共享，代码段在内存只有一份真实存储
    - **占用内存** = 150 + 10 x 50 = 650KB
- **页面共享**
  - 根据页式地址转换过程：如果在不同进程的页表中填上**相同的页框号**，就能访问相同的内存空间，从而实现**页面共享**。
  - 共享页面在内存只有一份真实存储，节省内存。

页面的共享

- 文本编辑器
  - 代码段（code1~3），数据段（data）



- 页表的建立
  - 操作系统为每个进程建立一个页表
    - 页表长度和首址存放在进程控制块中。
  - 当前运行进程的页表驻留在内存
    - 页表长度和首址由页表长度寄存器和页表首址寄存器指示。
- 页表的形式
  - CACHE
    - 访问速度快，成本较高。
  - 内存
    - 成本较低，访问速度慢。

- 页面的大小选择
  - 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
  - 页面的常见大小
    - 2的整数次幂：1KB，2KB，4KB

页表扩充——带中断位的页表

l 扩充有中断位和辅存地址的页表

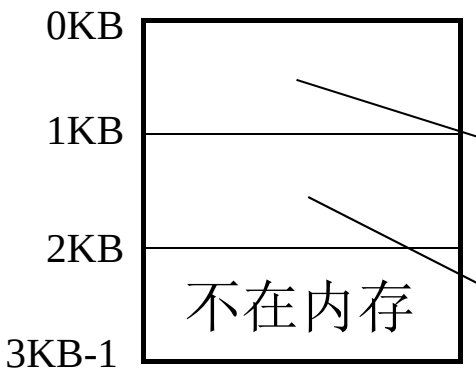
页号	页框号	中断位I	辅存地址
		1	
		0	

n 中断位I ——标识该页是否在内存?

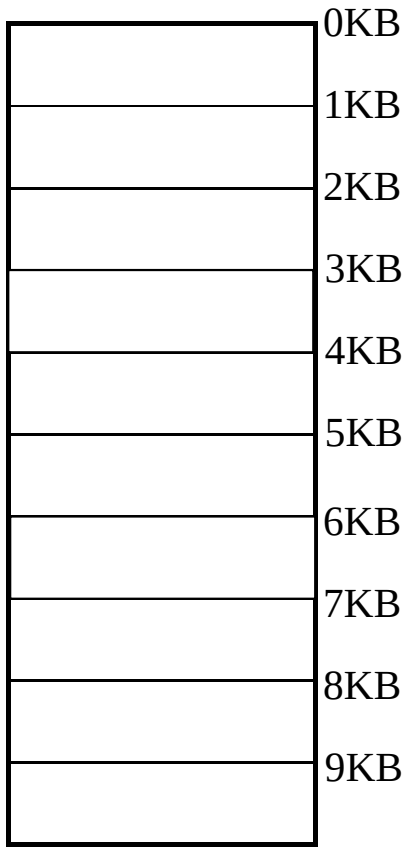
u 若I = 1， 不在内存

u 若I = 0， 在内存

n 辅存地址——该页在辅存上的位置



进程A



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

进程A的页表

页表扩充——带访问位和修改位的页表

- 扩充有访问位和修改位的页表

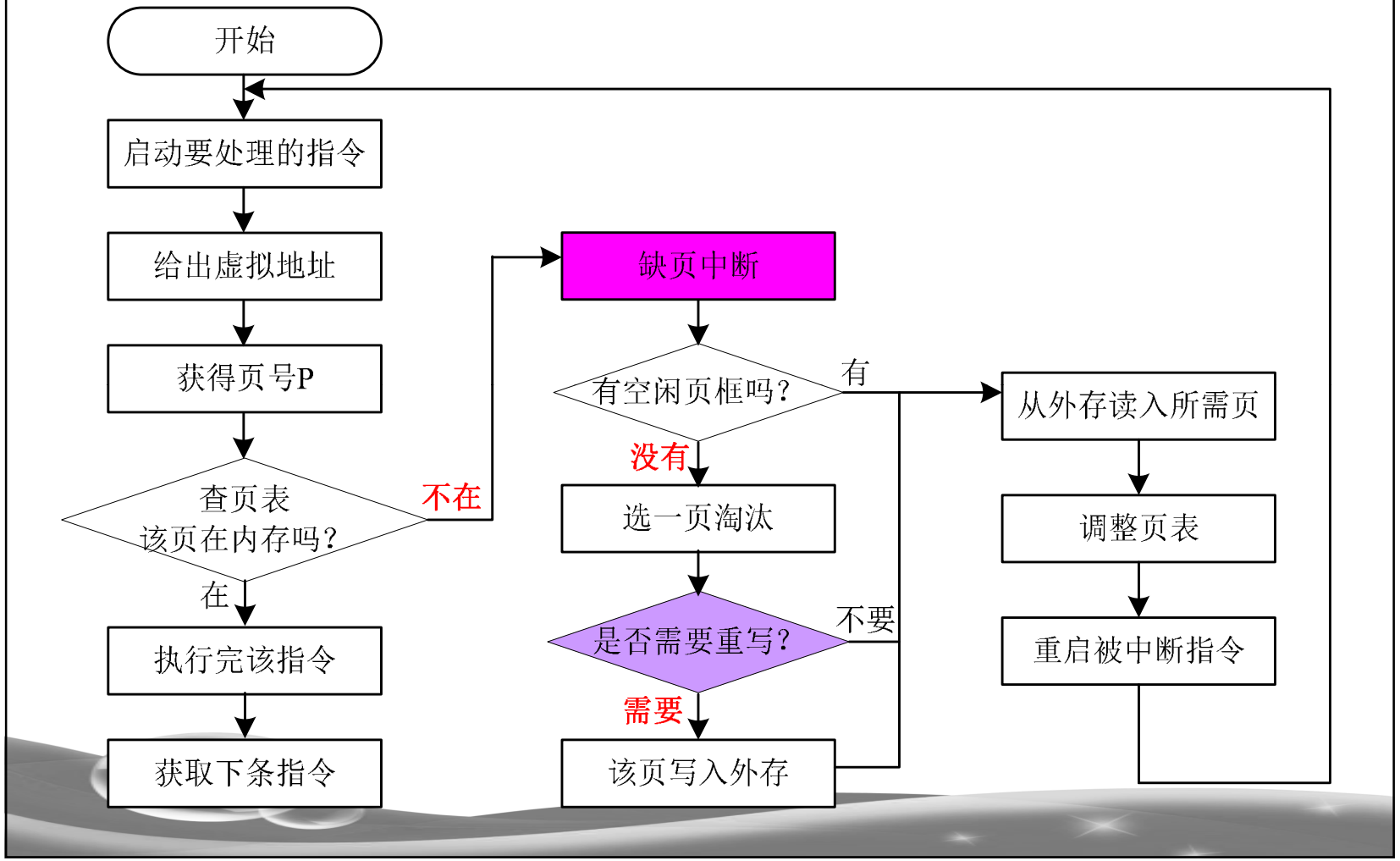
页号	页框号	访问位	修改位
		1	0
		0	1

- 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

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

访存指令的执行过程（含缺页中断处理）



页式系统的实时性

- 测量某个函数花费的时间：MuFunc( )

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

- 思考：测量的时间准不准？为什么？

- **缺页中断 vs 普通中断**
  - 相同点
    - 处理过程：保护现场、中断处理、恢复现场
  - 不同点
    - 响应时机
      - 普通中断在指令完成后响应
      - 缺页中断在指令执行过程中发生
    - 发生频率
      - 一条指令执行时可能产生多个缺页中断

二级页表

- **页表实现时的问题**
  - **n** 32位OS(4G空间)，每页4K，页表每个记录占4字节
    - **u** 进程的页数： 4G /4K = 1M
    - **u** 页表的记录数：1M
    - **u** 页表所占内存：1M * 4字节 = 4M
    - **u** 页表所占页框数：4M / 4K = **1K（且连续）**
  - **n 问题：**
    - **u 1）**页表全部装入过度消耗内存（4M）。
    - **u 2）难以找到连续**1K个页框存放页表。
  - **n** 解决办法
    - **u** ①采用离散分配解决连续内存难找的问题；
    - **u** ②仅将部分页表项调入内存解决过度内存消耗的问题。

二级页表

- 把页表以页面大小为单位分成若干个小页表，存入离散的若干个页框中。

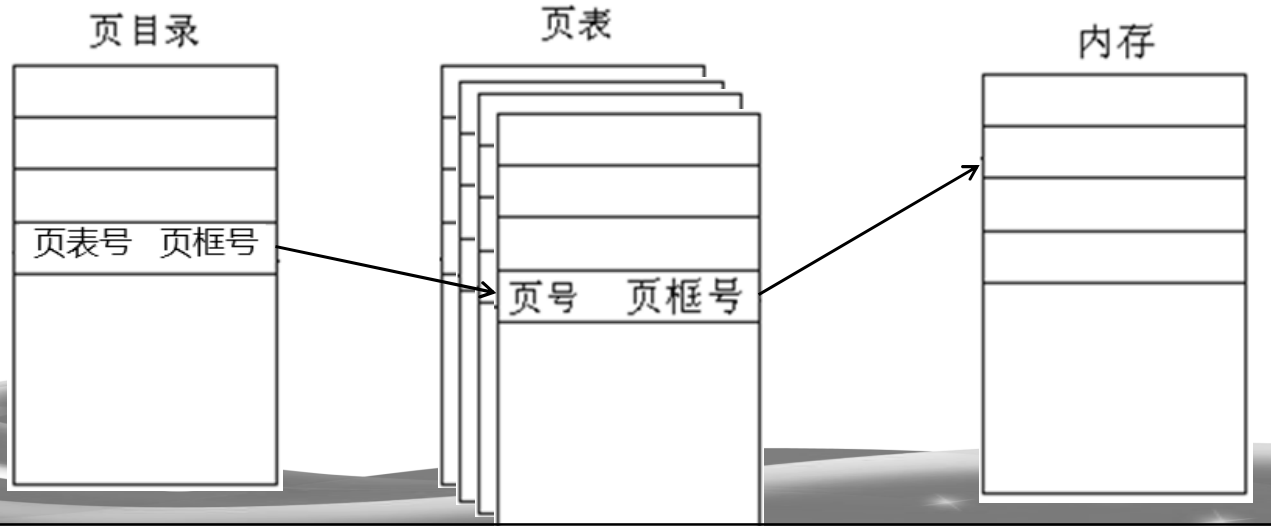
页号	页框号	中断位	外存地址	访问位	修改位
0	8	0	4000	1	0
1	24	0	8000	1	1
…	…	…	…	…	…

分解成若干个小页表

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

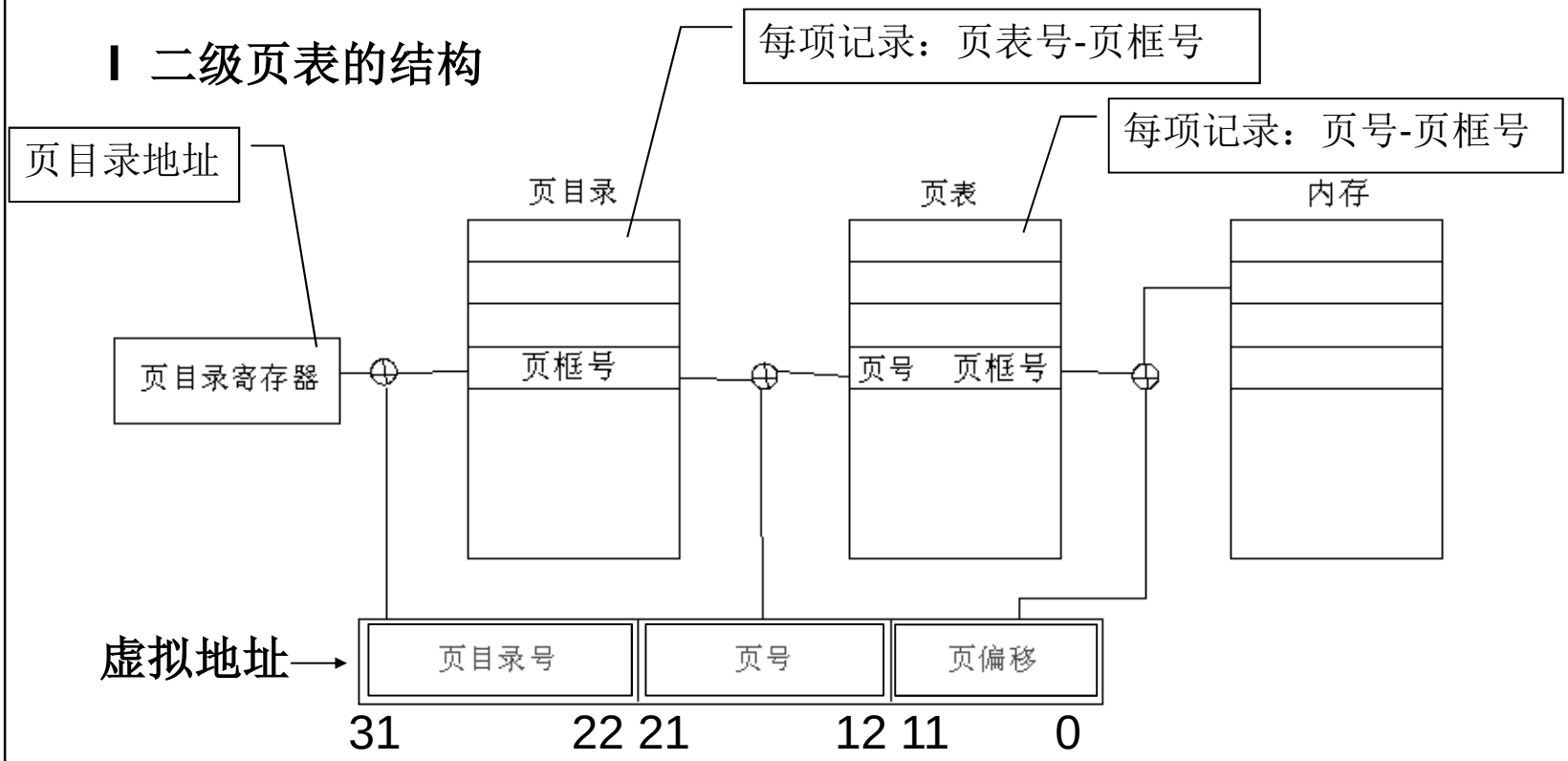
二级页表

- 把页表以页面大小为单位分成若干个小页表，存入离散的若干个页框中。
- 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）。
  - 页目录实际是一个特殊页表：每个记录存放的是小页表的序号（非页号）和所在的页框号的对应关系。
- 页目录：一级页表或外部页表；　小页表：二级页表



WINDOWS NT 二级页表的结构

I 二级页表的结构



页目录号：页表的编号（0起顺序编号） 2^{10} = 1K个页表

页　　号：页面的编号（0起顺序编号） 2^{10} = 1K个页面

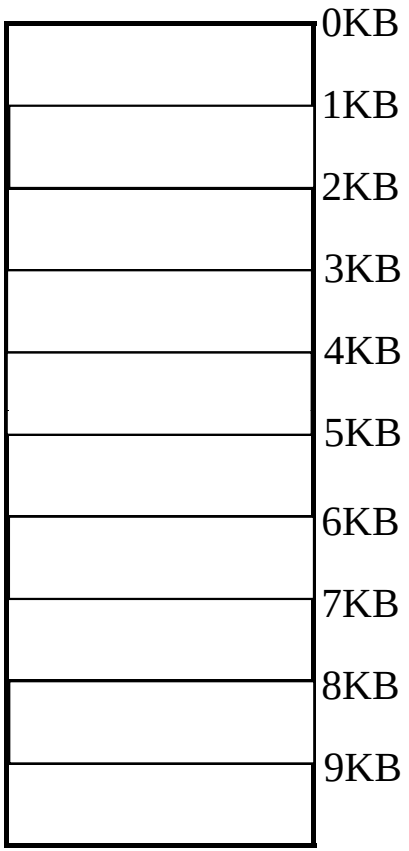
页 偏 移：页偏移　页面大小（ 2^{12} = 4K）

l 二级页表地址映射特点

- n访问数据需要 ? 次访问内存。
- n页目录调入内存
- n页表按需要调入主存
- n页面、页表，页目录的大小都刚好4K(占1个页框) 。

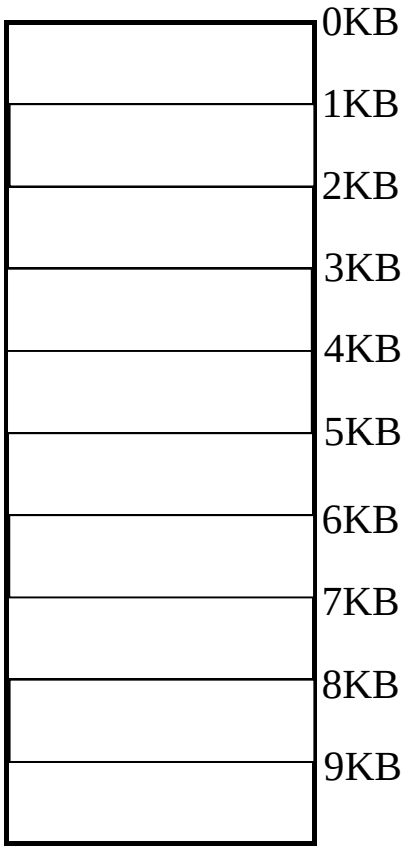
淘汰策略

- **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。
- **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
- **好的淘汰策略**
  - 具有较高的命中率
  - 和较少的页面抖动



l 常用的淘汰算法

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



最佳算法（OPT算法, Optimal）

- 思想
  - 淘汰以后不再需要或最远的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	C	C	C
		B	B	B	B	B	B	B	B	B	D	D
			C	D	D	D	E	E	E	E	E	E
缺页	X	X	X	X			X			X	X	

缺页次数 = 7　　缺页率 = 7 / 12 = 58%

最佳算法（OPT算法, Optimal）

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

缺页次数 = 9， 缺页率 = 9 / 12 = 75%

- **优点**
  - 实现简单：页面按进入内存的时间排序，淘汰队头页面。
- **缺点**
  - 进程只有按顺序访问地址空间时页面命中率才最理想。
  - 异常现象：对于一些特定的访问序列，随分配的页框增多，缺页率反而增加！

l FIFO例子

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

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

缺页次数 ＝10， 缺页率 ＝ 10 / 12 = 83%

最久未使用淘汰算法（LRU，Least Recently Used ）

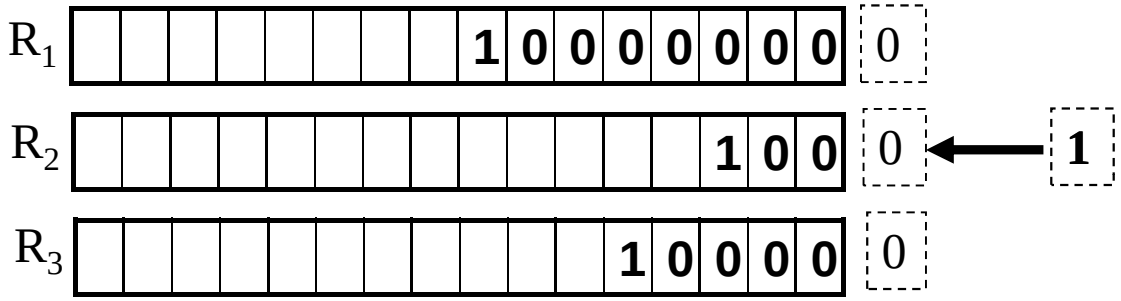
- 思想
  - 淘汰最长时间未被使用的页面。
- 例子
  - 3个页框，页面序列：A,B,C,D,A,B,E,A,B,C,D,E。分析其按照LRU算法进行页面淘汰时的缺页情况。

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	C	C	C
		B	B	B	A	A	A	A	A	A	D	D
			C	C	C	B	B	B	B	B	B	E
缺页	X	X	X	X	X	X	X			X	X	X

缺页次数 ＝10， 缺页率 ＝ 10 / 12 = 83%

l LRU的实现（硬件方法）

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



- n 当需要淘汰页面时选择R值最大的页。
  - u R值越大，对应页未被使用的时间越长。
- n R的位数越多且移位周期越小就越精确，但硬件成本也越高。

LRU近似算法

- 利用页表访问位，页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：在时间T内，该页被访问过，保留该页。
  - 访问位为0：在时间T内，该页未被访问过，淘汰该页！

缺点

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

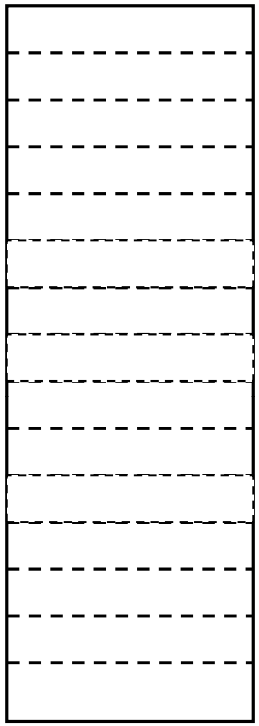
最不经常使用（LFU）算法

- 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。

- **影响缺页次数的因素**
  - 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
    - 局部性越好，越不容易缺页
    - 跳转或分支越多越容易缺页
  - 淘汰算法

l 页式系统的不足

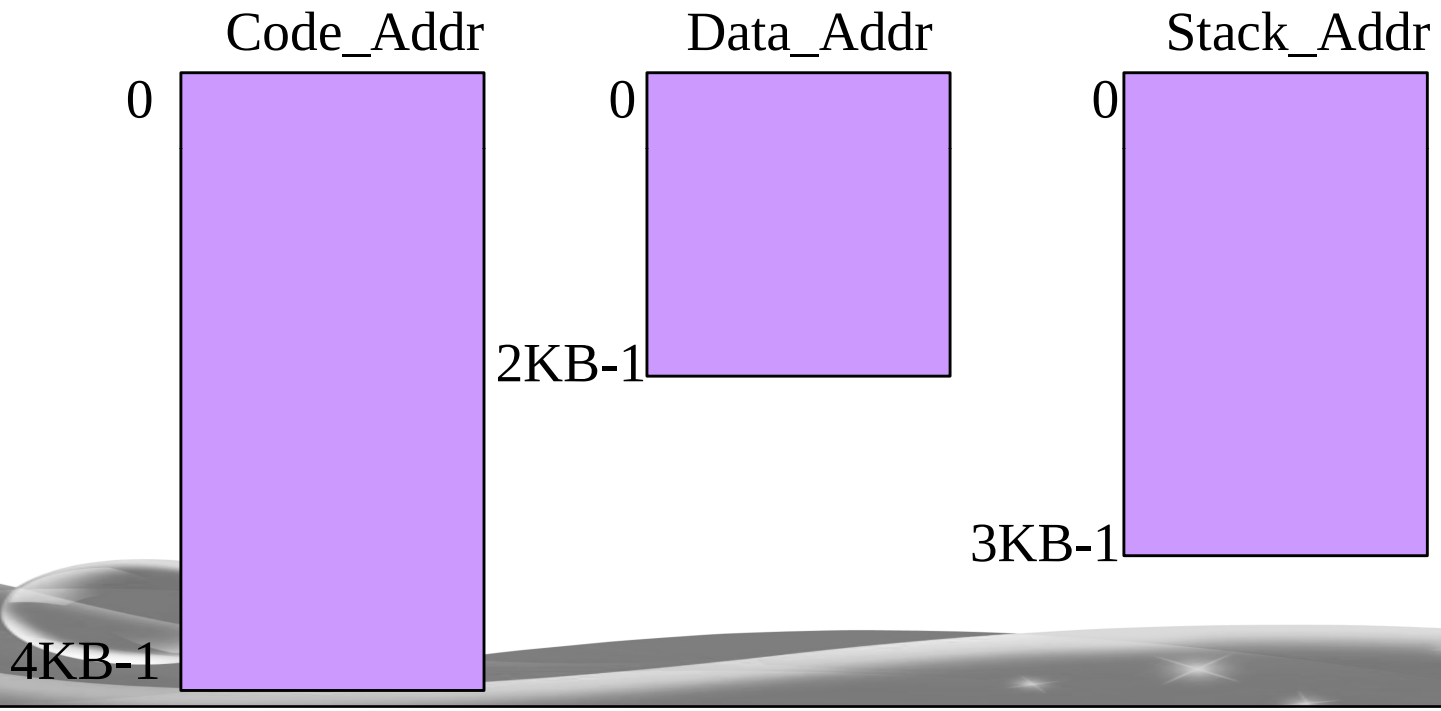
- n 页面划分无逻辑含义
- n 页的共享不灵活
- n 页内碎片



内存

段式存储管理

- 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，
  - 例：一个具有代码段、数据段、堆栈段的进程

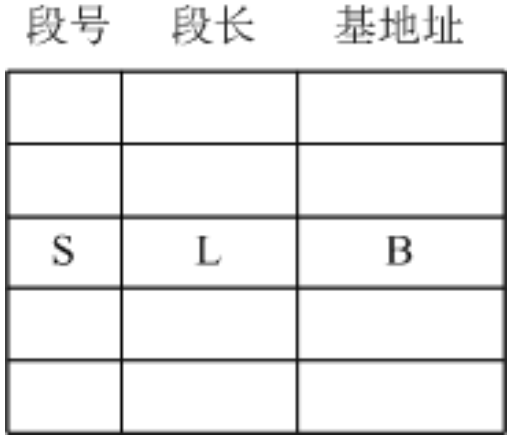


- **段式内存管理系统的内存分配**
  - 以段为单位装入，每段分配连续的内存；
  - 但是段和段不要求相邻。
- **段式系统的虚拟地址**
  - 段式虚拟地址VA包含段号S和段内偏移W
  - VA:（S，W）

段号S	段内位移W

段式地址的映射机制

- 段表（SMT，Segment Memory Table）
  - 记录每段在内存中映射的位置

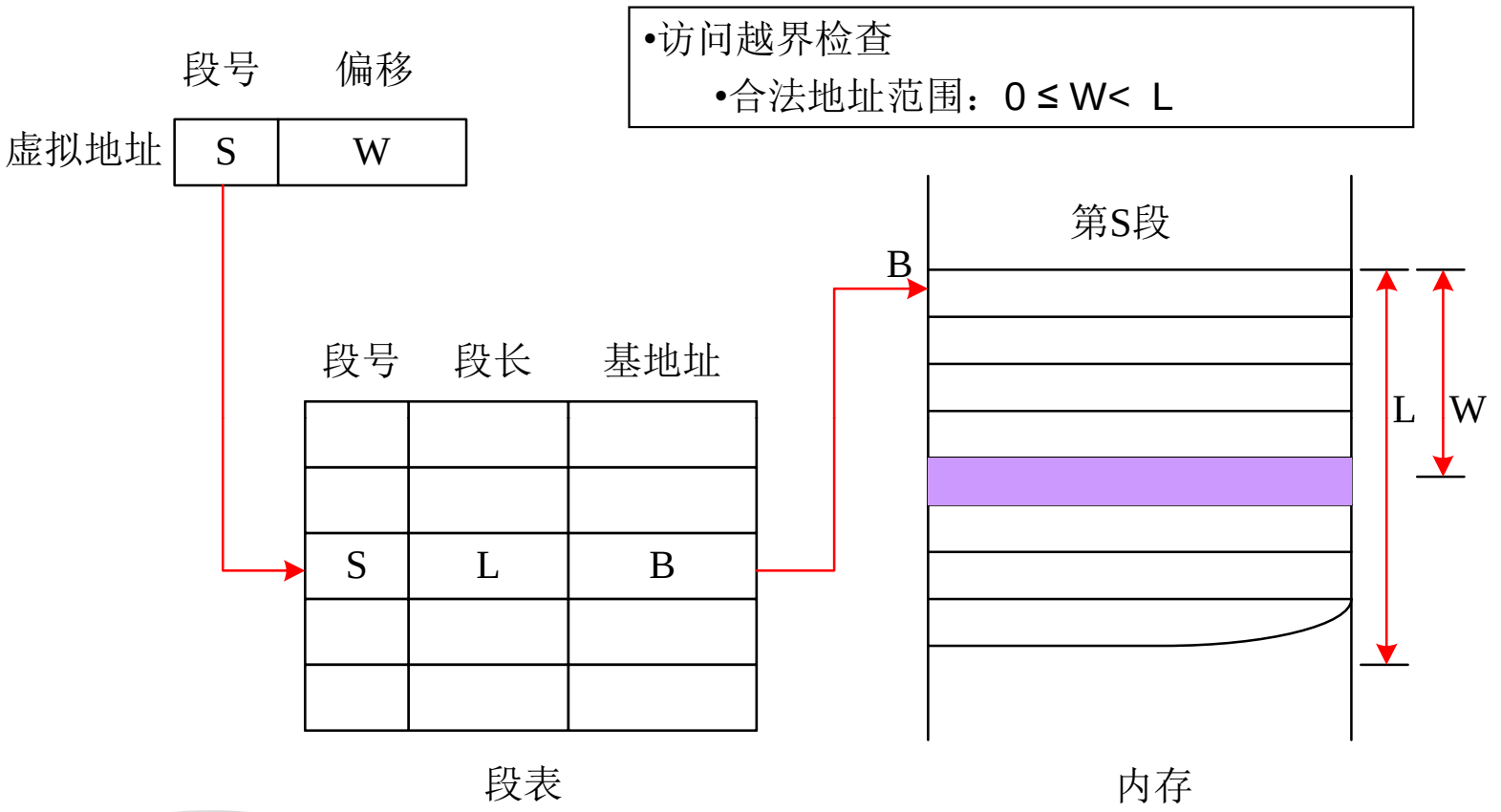


- 段表的字段
  - 段号S：段的编号（唯一的）
  - 段长L：该段的长度
  - 基地址B：该段在内存中的首地址

段式地址映射过程

- 段式地址映射过程
  - 1.由逻辑地址VA分离出(S, W);
  - 2.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

段式地址映射过程



l 段表的扩充

- n 基本字段：段号，长度，基址
- n 扩展字段：中断位，访问位，修改位，R/W/X

段号	长度	基址	**中断位**	**访问位**	**修改位**	**R**	**W**	**X**

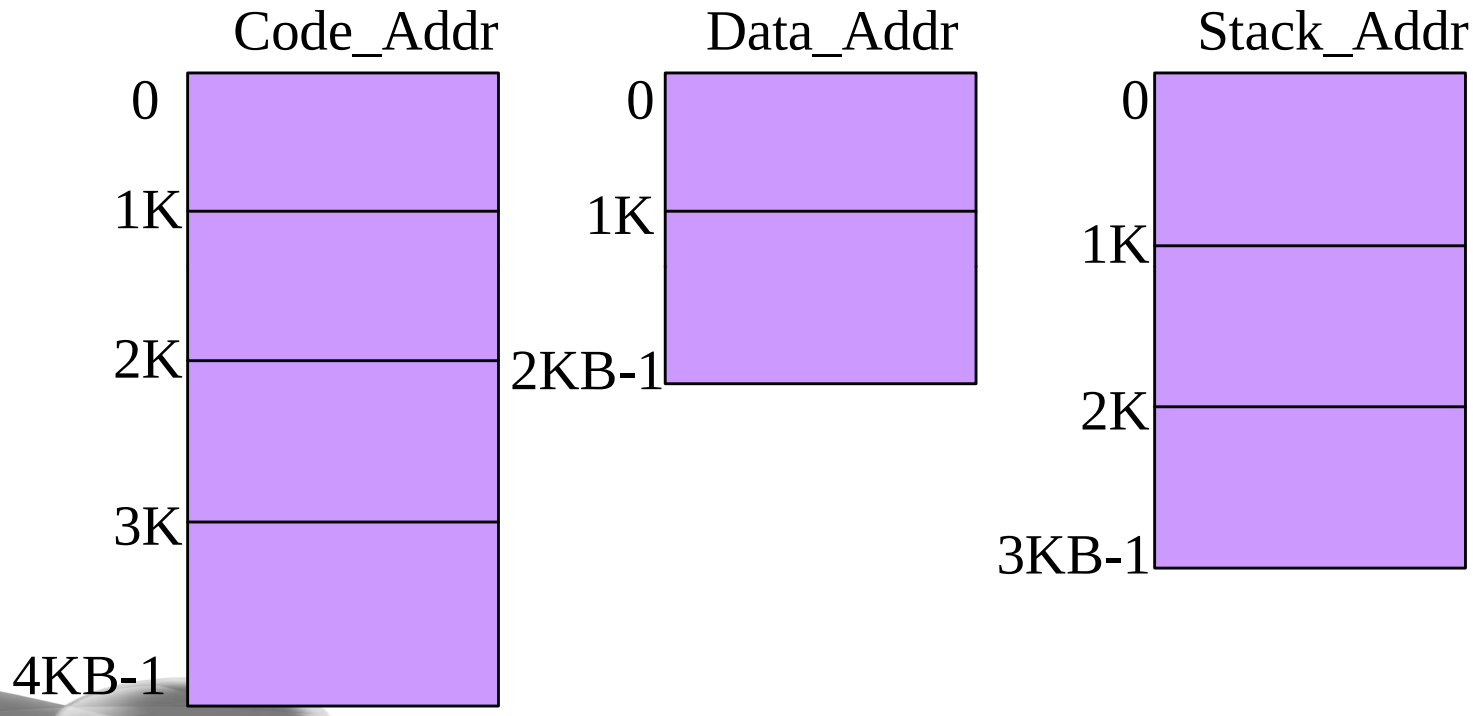
- **段的共享**
  - 共享段在内存中只有一份存储。
  - 共享段被进程映射到自己的空间（写入段表）
  - 需要共享的模块都可以设置为单独的段
- **段式系统的缺点**
  - 段需要连续的存储空间
  - 段的最大尺寸受到内存大小的限制；
  - 在辅存中管理可变尺寸的段比较困难；

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

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。

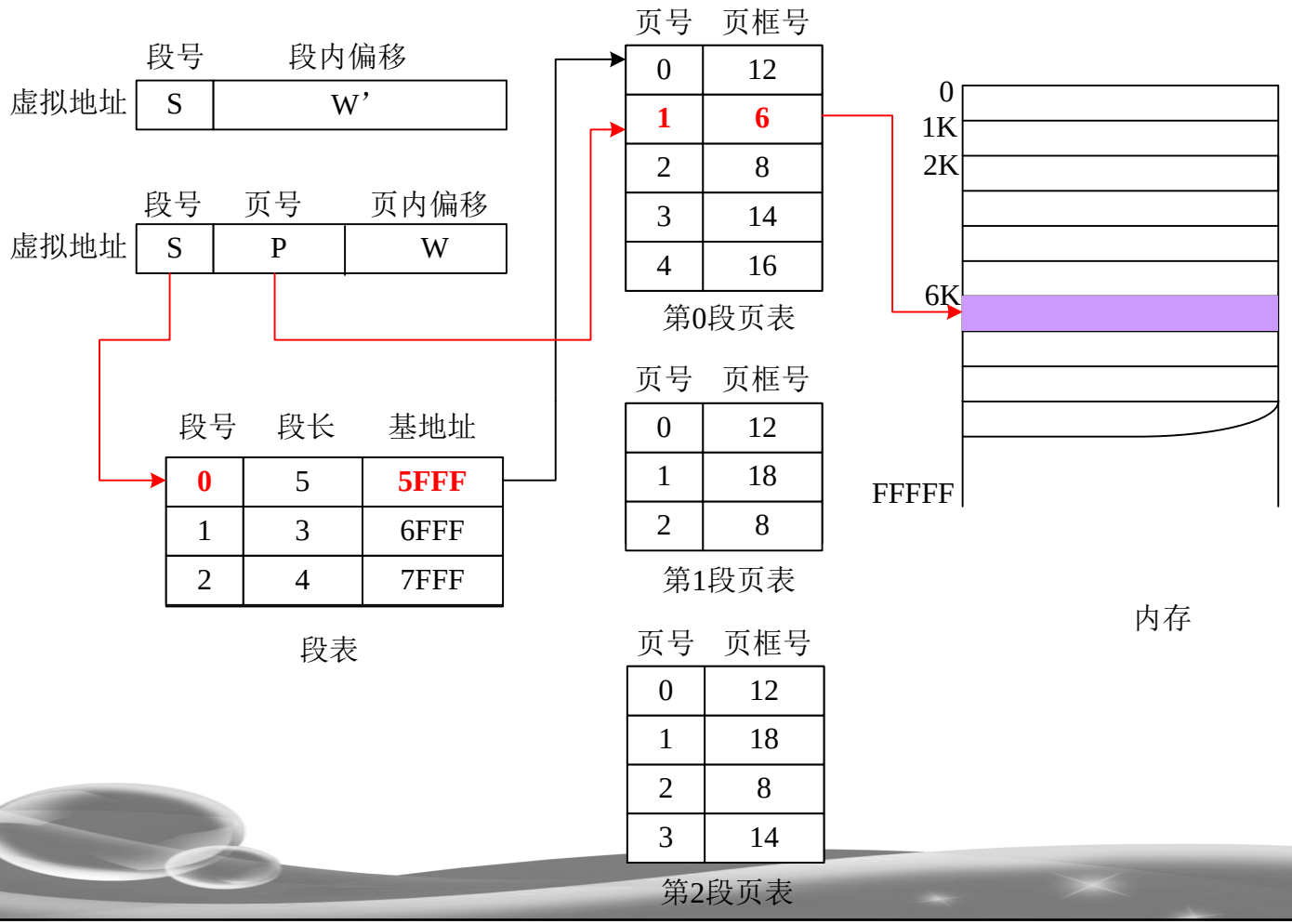


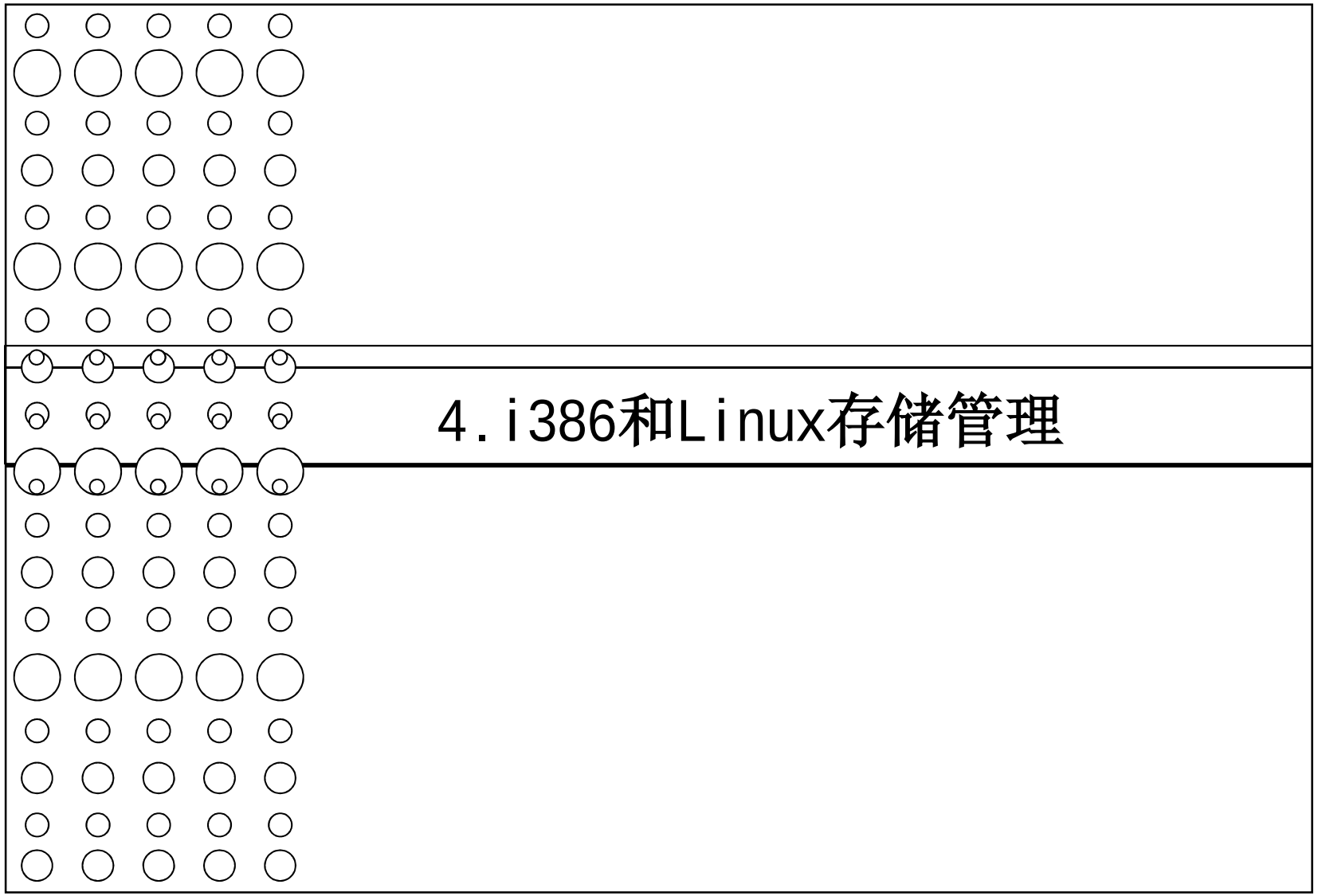
l 段页式系统的地址构成：段号，页号，页内偏移

- n 逻辑地址：段号S、页号P和页内位移W。
- n 内存按页划分，按页装入。

- **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；
    - 系统为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（**段长**）。
    - 页表给出每页对应的页框。

段页式地址映射：VA(S, W') → (S, P, W) → MA





4.i386和Linux存储管理

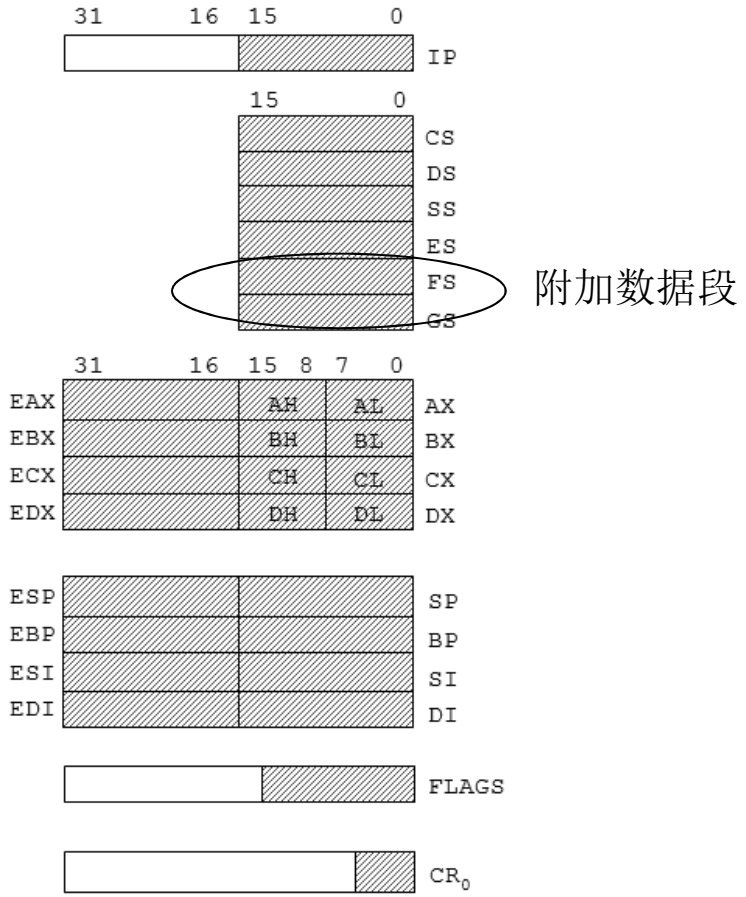
i386和Linux的存储机制

- i386的存储管理
- i386的分段和分页
- Linux存储管理

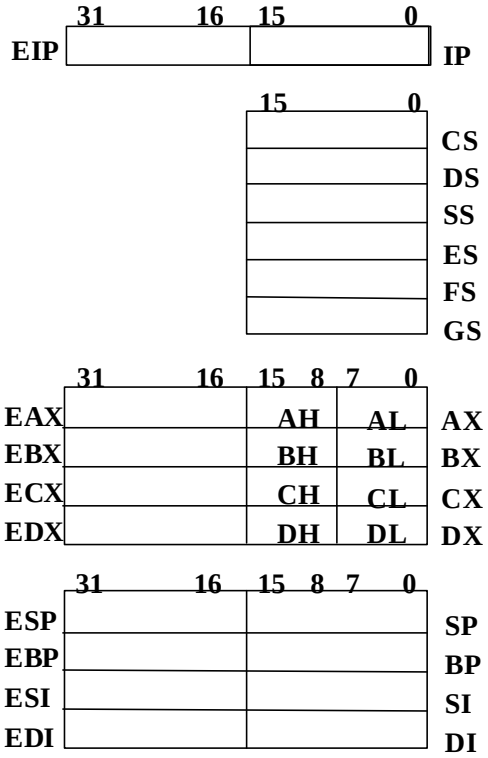
i386的存储管理

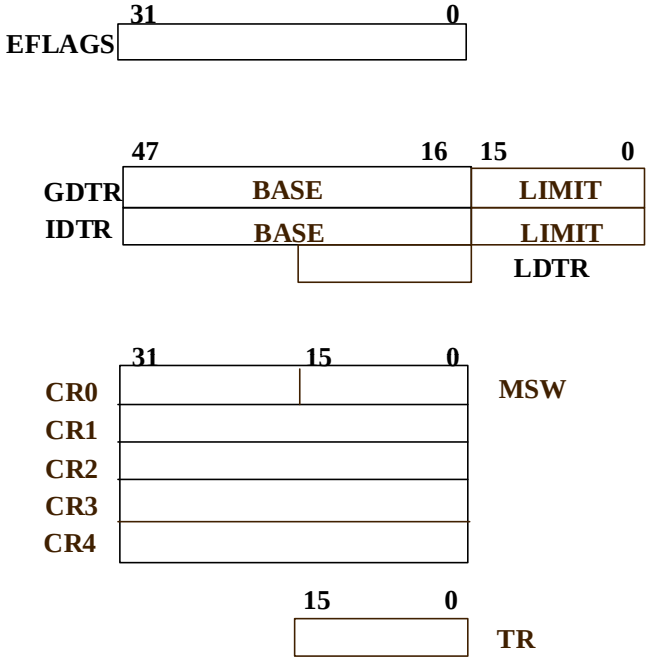
- 实模式（**Real Mode**）
  - 20位地址
    - 段地址（16位）：偏移地址（16位）
    - 段地址16字节对齐
    - 20位：1M内存空间
- 保护模式（**Protect Mode**）
  - 32位地址空间：4G物理内存
  - 支持多任务，能够快速地进行任务切换和保护任务环境
  - 支持资源共享，能保证代码和数据的安全和任务隔离
  - 支持分段机制＋分页机制
  - 新的硬件
    - EAX~EDX(32位扩展)
    - CR0~CR3, GDTR,LDTR,IDTR,

实模式的寄存器模型

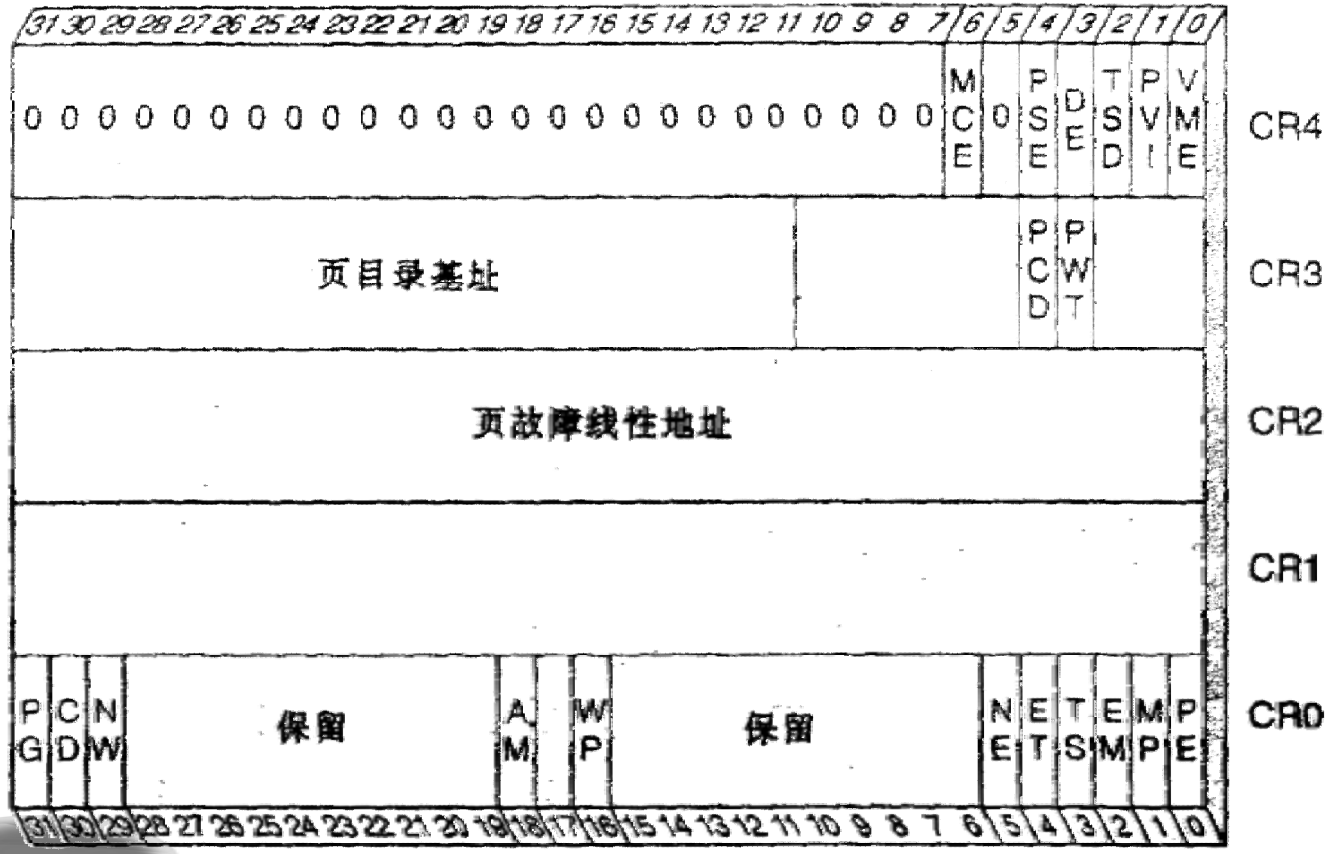

31 16 15 0
IP
15 0
CS
DS
SS
ES
FS
GS
附加数据段
31 16 15 8 7 0
EAX AH AL AX
EBX BH BL BX
ECX CH CL CX
EDX DH DL DX
ESP SP
EBP BP
ESI SI
EDI DI
FLAGS
CR0

保护模式的寄存器模型


31
16
15
0
EIP
IP
15
0
CS
DS
SS
ES
FS
GS
31
16
15
8
7
0
EAX
AH
AL
AX
EBX
BH
BL
BX
ECX
CH
CL
CX
EDX
DH
DL
DX
31
16
15
8
7
0
ESP
SP
EBP
BP
ESI
SI
EDI
DI

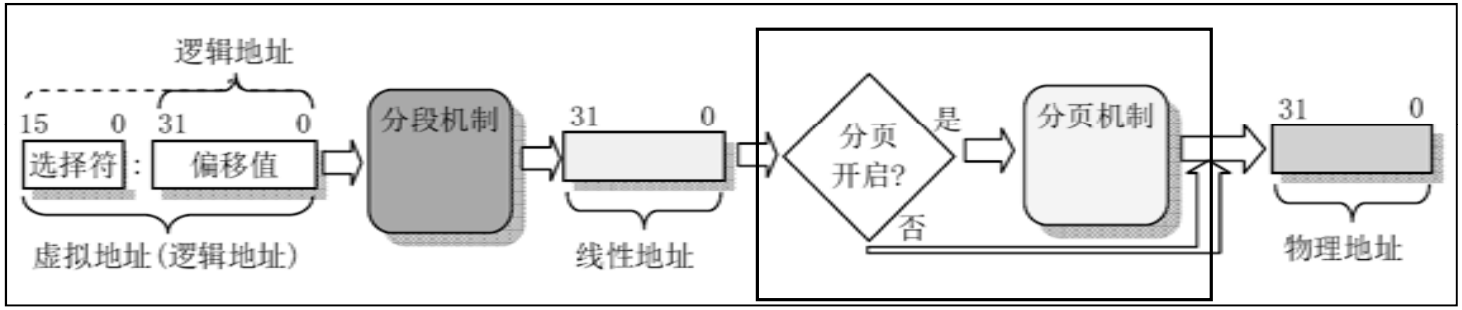

31
0
EFLAGS
47
16
15
0
GDTR
BASE
LIMIT
IDTR
BASE
LIMIT
LDTR
31
15
0
CR0
MSW
CR1
CR2
CR3
CR4
15
0
TR

控制寄存器


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
MCE
0
PSE
DE
TSD
PVI
VME
CR4
页目录基址
PCD
PWT
CR3
页故障线性地址
CR2
CR1
PG
CD
NW
保留
AM
WP
保留
NE
ET
TS
EM
MP
PE
CR0
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

i386段页式地址转换

- 第一级：段机制（逻辑地址到线性地址）
- 第二级：分页机制（线性地址到物理地址）



Intel x86 CPU 架构下的三种“地址”

- MMU：执行地址映射过程

MMU地址映射过程

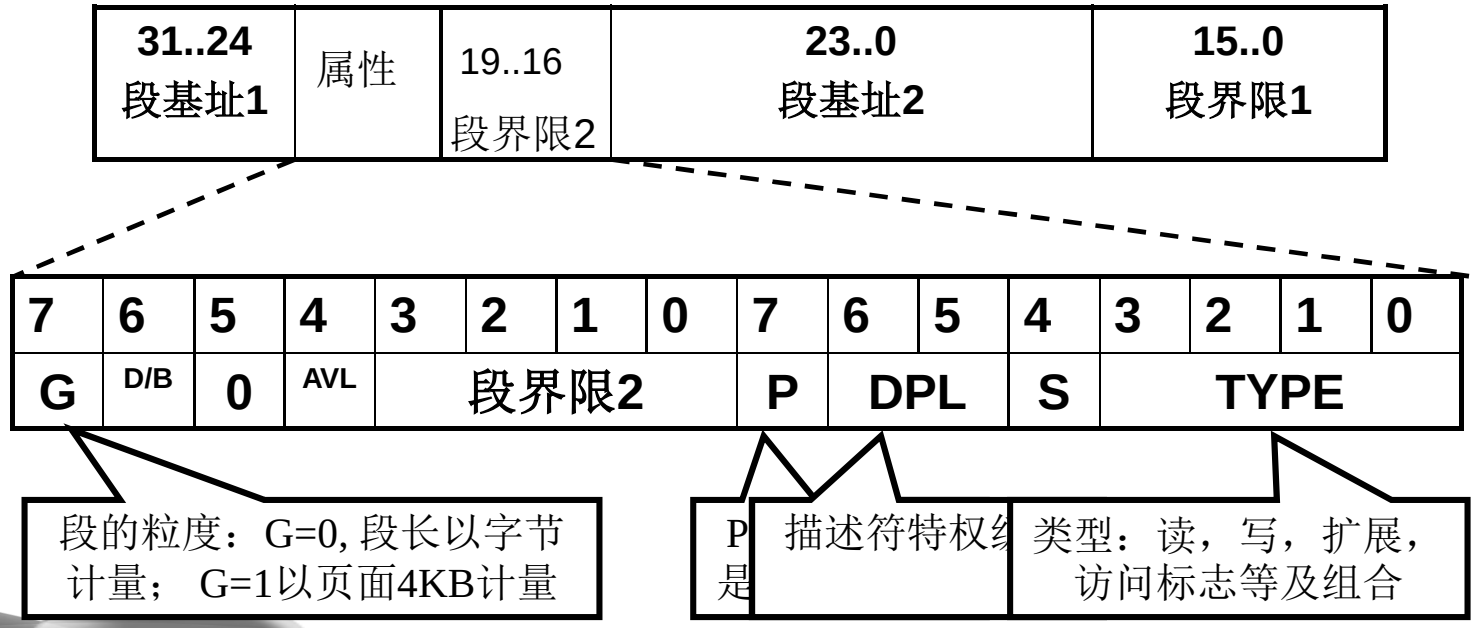
- MMU收到VA后
  - 检查该段是否在内存中（“P”位）
    - 在：段基址＋OFFSET=>PA
    - 否：产生“段不存在”异常，完成段调入
  - 访问权限检查
    - 如果RPL>=DPL，允许访问
      - 否则，产生“权限违例”异常

段描述符（Descriptor）

- 保护模式的段
  - 由段描述符（Descriptor）来描述， 8字节。
    - 段基址
    - 段界限
    - 段属性
      - 段类型
      - 访问该段所需的最小特权级
      - 是否在内存
      - …

l 段描述符（Descriptor）

- n段基址：32位（段基址1+段基址2）
- n段界限：20位（段界限1+段界限2）



- **属性的描述**
  - P：Present，是否在内存中【1在】
  - DPL: Descriptor Privilege Level 描述符特权级别
  - S: 描述符的类型
    - 【数据段，代码段S=1】
    - 【系统描述符，门描述符S＝0】
  - TYPE : 描述符的存取类型
    - 【读，写，扩展，访问标志等及其组合】
  - 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）

段描述符（Descriptor）的实现

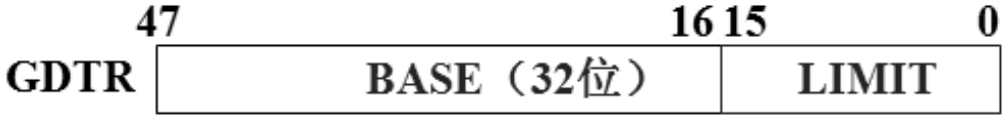
```
typedef struct Descriptor
{
    unsigned int base24_31          :8    //基地址的高8位
    unsigned int g                  :1    //段长单位，0:字节，1:4K
    unsigned int d_b                :1
    unsigned int unused             :1
    unsigned int avl                :1
    unsigned int seg_limit_16_19    :4  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

段描述符表（Descriptor Table）

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

GDTR/LDTR/IDTR/TSSR四个寄存器

- **GDTR/LDTR/IDTR/TSSR**
  - 存放GDT/LDT/IDT/TSS基址和大小的寄存器
  - GDTR（ 48位）



    - BASE：指示GDT在物理存储器中开始的位置
    - LIMIT：规定GDT的界限.
      - LIMIT有16位，GDT最大65536个字节，能够容纳65536/8=8192个描述符

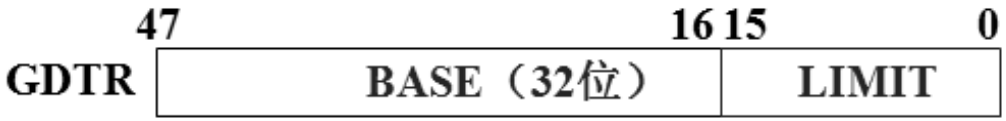
- 例：(GDTR)=001000000FFFH，求GDT在物理存储器中的起始地址，结束地址，表的大小，表中可以存放多少个描述符？

解： 起始地址：0010 0000H

结束地址: 0010 0000H+0FFFH=0010 0FFFH

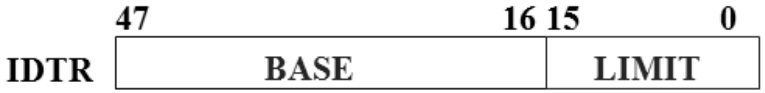
表的大小 = 0FFFH+1=4096字节

表中可以存放描述符数量 = 4096 / 8=512个



中断描述符表寄存器IDTR（ 48位）

- 在物理存储器地址空间中定义中断描述符表IDT

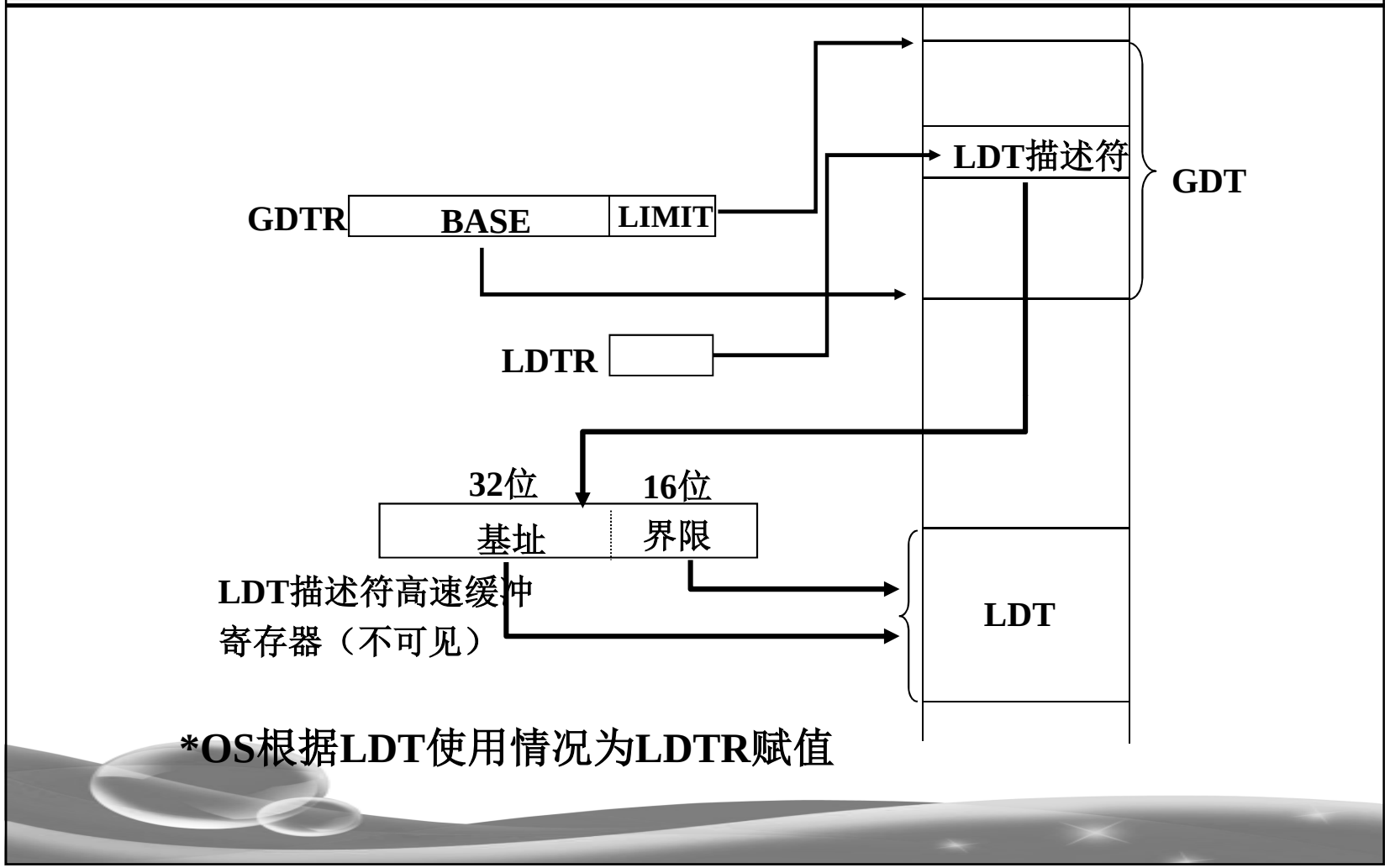


  - 由于Pentium只能支持256个中断和异常，因此LIMIT最大为（　　）？
  - IDT中的描述符类型为中断门

局部描述符表寄存器LDTR（ 16位）

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

为当前任务创建LDT的过程

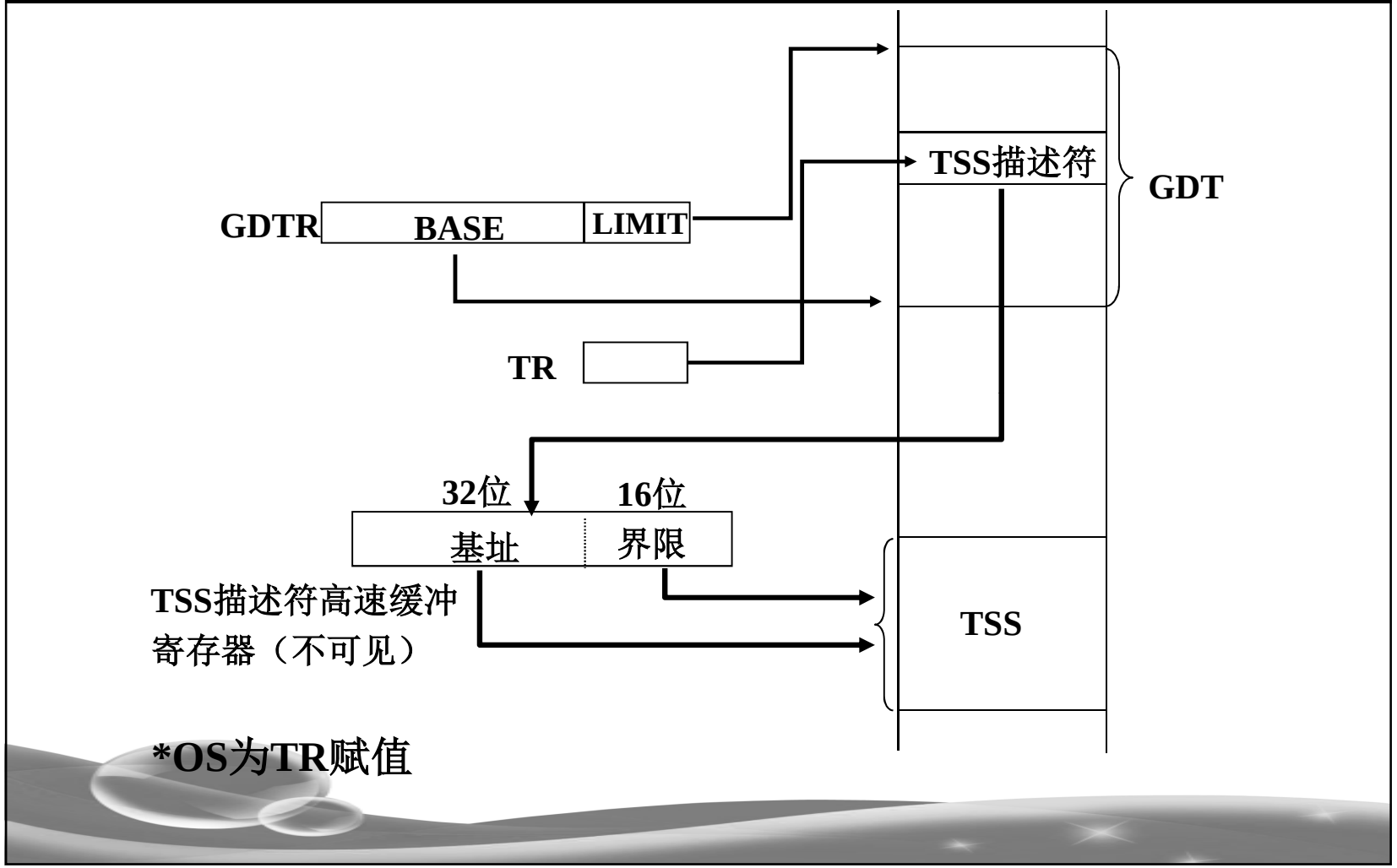


*OS根据LDT使用情况为LDTR赋值

任务寄存器TR

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

生成一个新任务的过程


TSS描述符
GDT
GDTR
BASE
LIMIT
TR
32位
16位
基址
界限
TSS描述符高速缓冲
寄存器（不可见）
TSS
*OS为TR赋值

段选择子（Selector）

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

- 构成
  - 索引域（INDEX）：13位
  - TI域（Table Indicator）：1位
  - 特权级别域（Request Privilege Level）： 2位

段选择子（Selector）

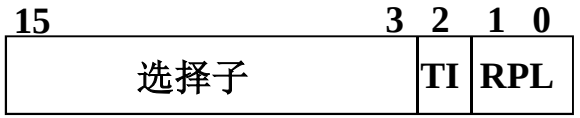
- TI域（Table Indicator）
  - TI＝1，从LDT中选择相应描述符，
  - TI＝0，从GDT中选择描述符。
- 索引域（INDEX）
  - 给出段描述符在GDT或LDT中的位置。
- 特权级别域（Request Privilege Level）
  - 请求者最低特权级的限制。
  - 只有请求者特权级高于或等于DPL，对应段才能被存取，实现段的保护。

- 例：设LDT的基址为0012 0000H，GDT的基址为00100000H，(CS)=1007H，那么：
  - ①请求的特权级是多少
  - ②段描述符位于GDT中还是LDT中
  - ③段描述符的地址是什么

解：(CS)=1007H=0001 0000 0000 0111B

① RPL=3，申请的特权级为3

②TI=1，描述符位于LDT中



③描述符相对于LDT基址的偏移量为

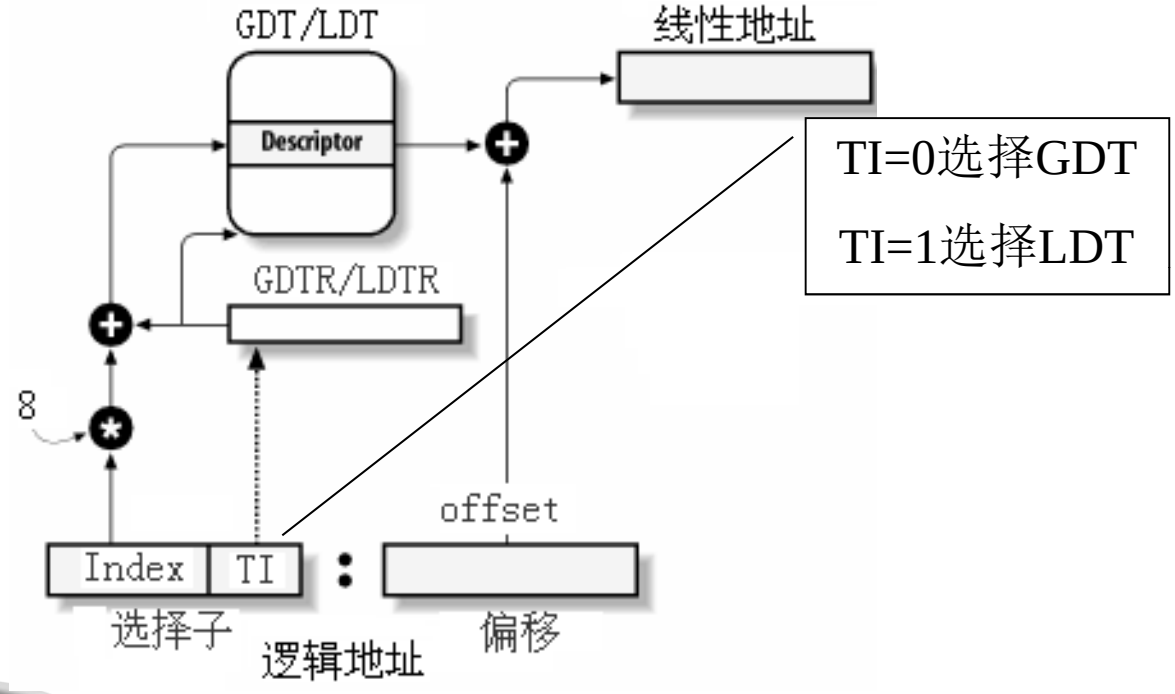
0001 0000 0000 0B´8=512 ´8=4096=1000H

段描述符的地址为

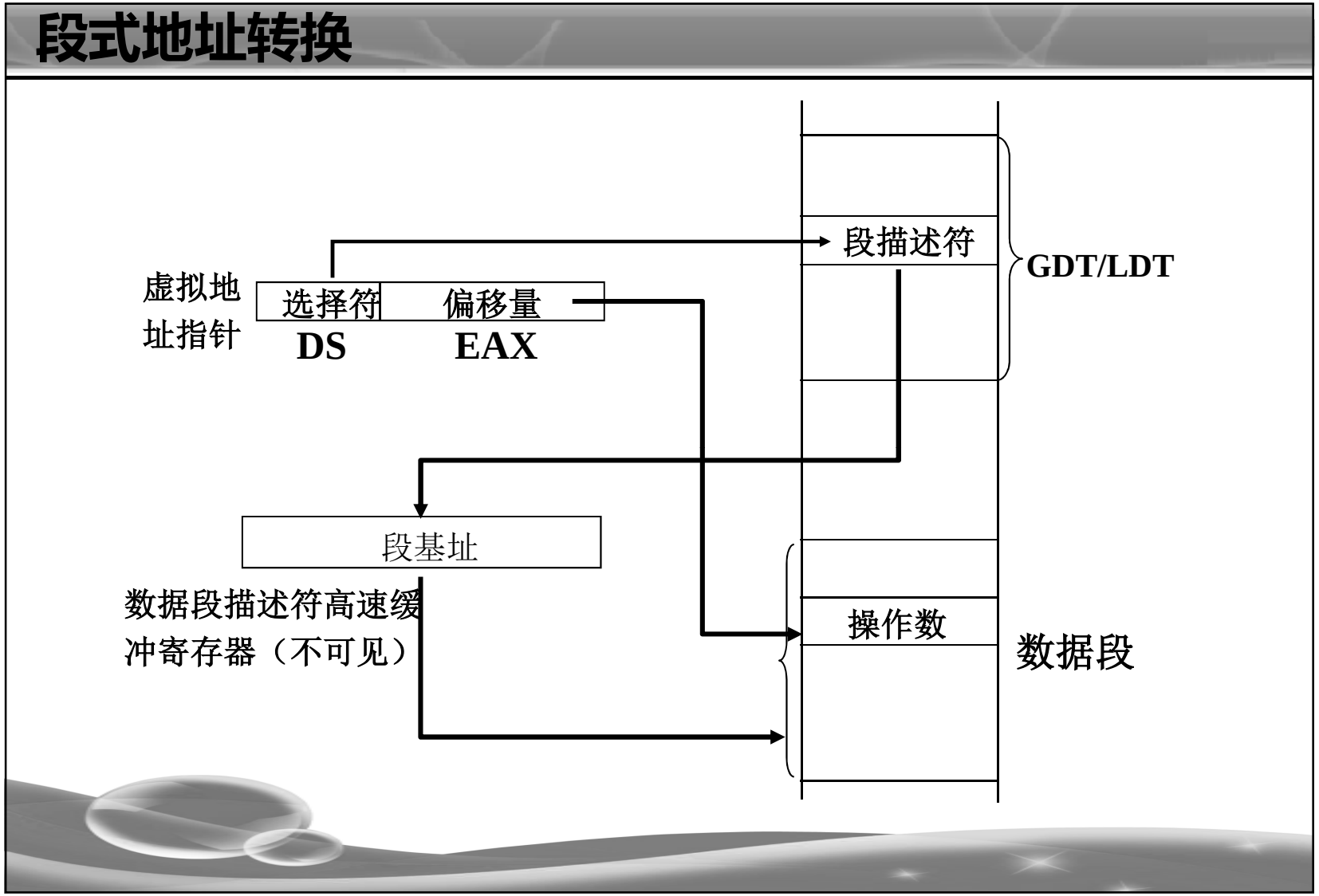
0012 0000H+1000H=00121000H

l 段机制的功能

n 把逻辑地址转换到线性地址（32位，4G）。



段式地址转换


段描述符
GDT/LDT
虚拟地
址指针
选择符
偏移量
DS
EAX
段基址
数据段描述符高速缓
冲寄存器（不可见）
操作数
数据段

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

  解：

  将此虚拟地址转换成物理地址

  ＝基地址+偏移量

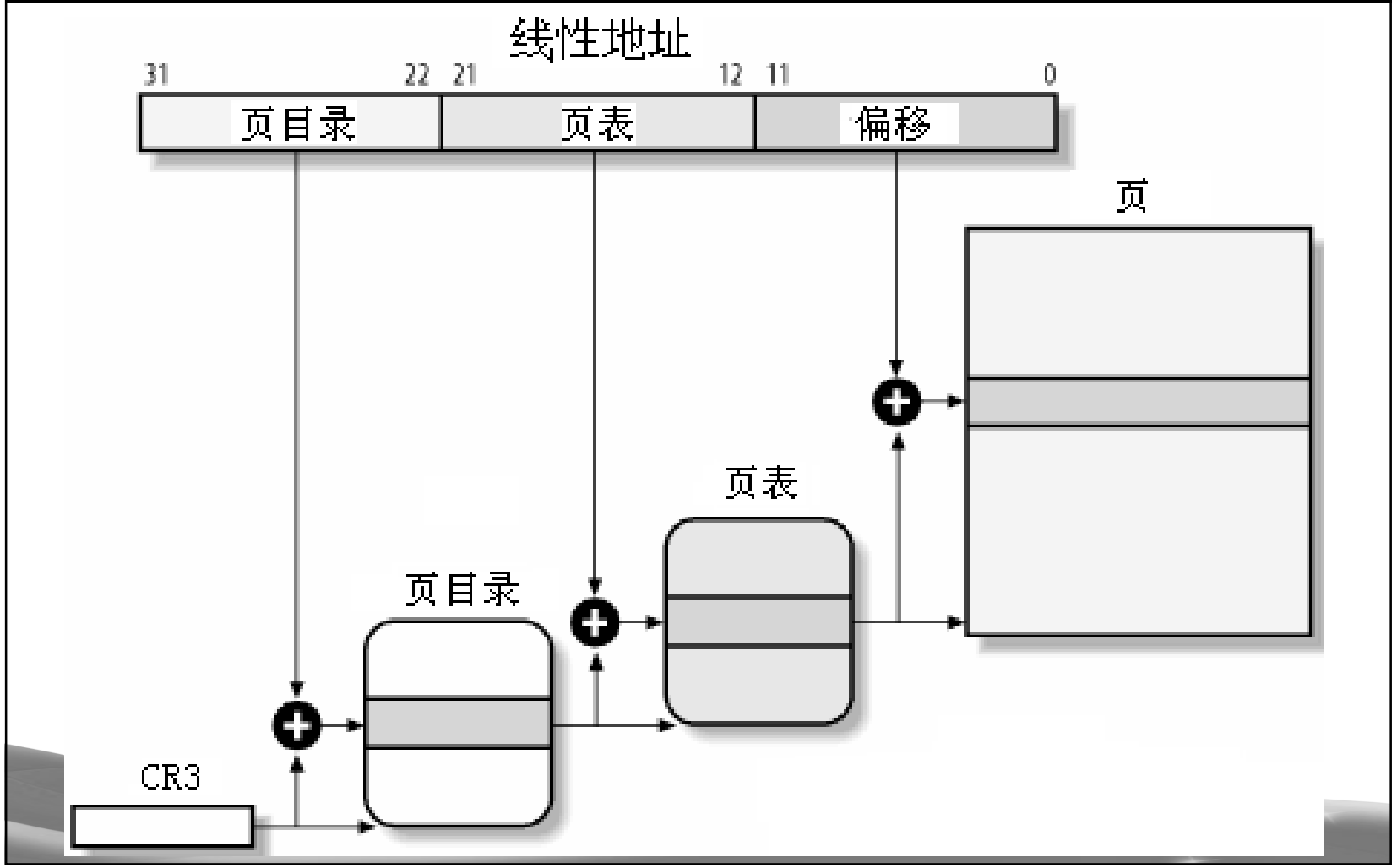
  =00030000H+00002000H

  =0003 2000H

硬件分页

- **页 vs. 页框**
  - **n** 页：虚拟页，可在主存也可在磁盘
  - **n** 页框：物理页，RAM块
- **分页**
  - **n** Intel CPU的页：4KB
  - **n** 通过设置CR3的PG位开启分页功能
  - **n** 分页发生在物理地址送地址线之前：线性地址→物理地址
  - **n** 在MMU中进行分页
- **数据结构**
  - **n** 页表
  - **n** 页目录
    - **u** 当前页目录的物理基地址在CR3中

二级页表结构（Windows）

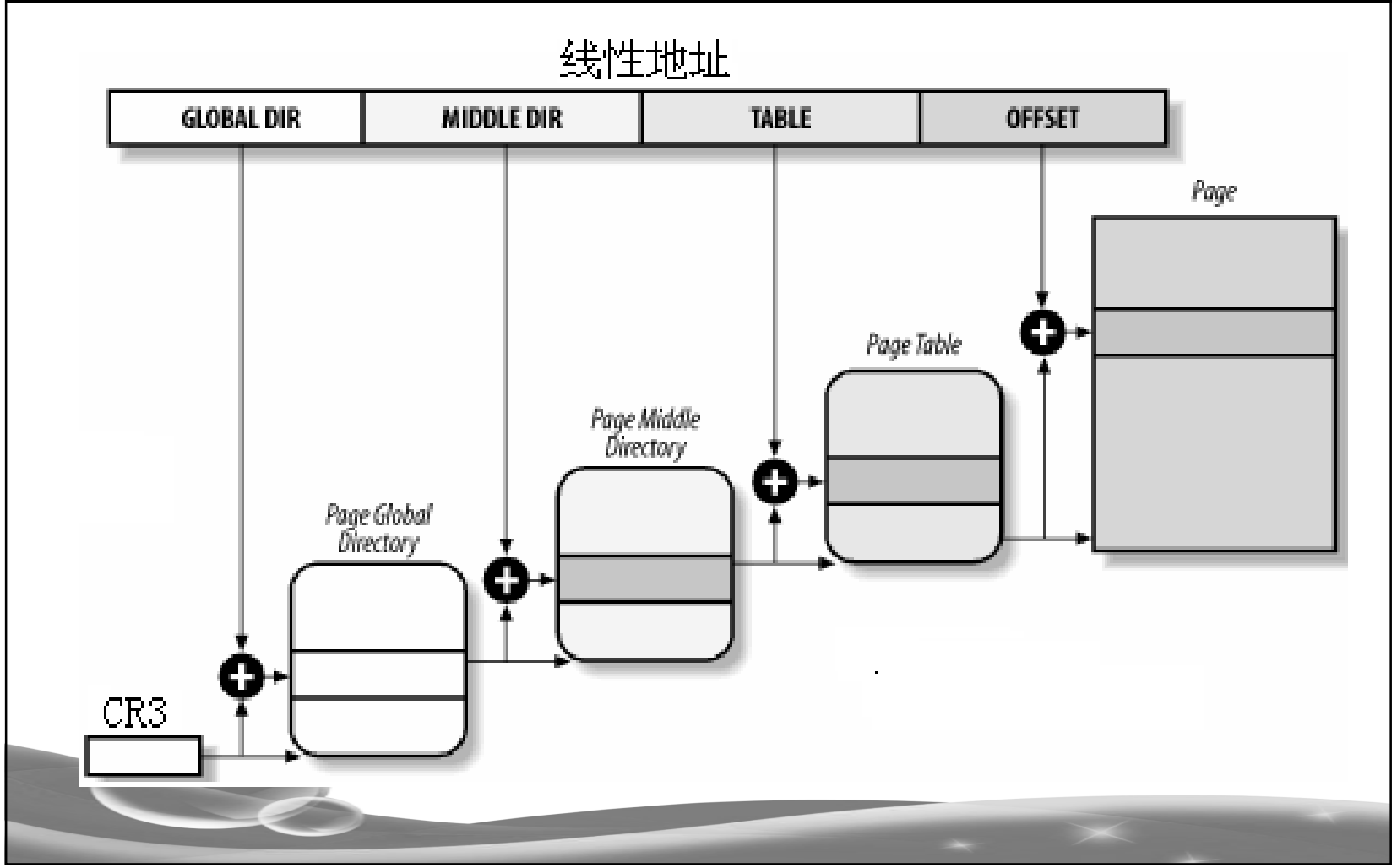


二级页表结构（Windows）

- 页目录：10bit ；页表：10bit ；偏移：12bit。
- 线性地址结构

```
typeded struct
{
    // 表示页目录的下标，该表项指向一个页表
      unsigned int dir  :10;
    // 表示页表的下标，该表项指向一个页框
      unsigned int page: 10;
    // 在4K字节页框内的偏移量
      unsigned int offset  :12;
}线性地址
```

Linux的三级页表结构



Linux的三级页表结构

- 与体系结构无关的三级页表结构
  - pgd，页全局目录（page global directory）
  - pmd，页中级目录（page middle directory）
  - pte，页表项（page table entry）
  - offset, 偏移

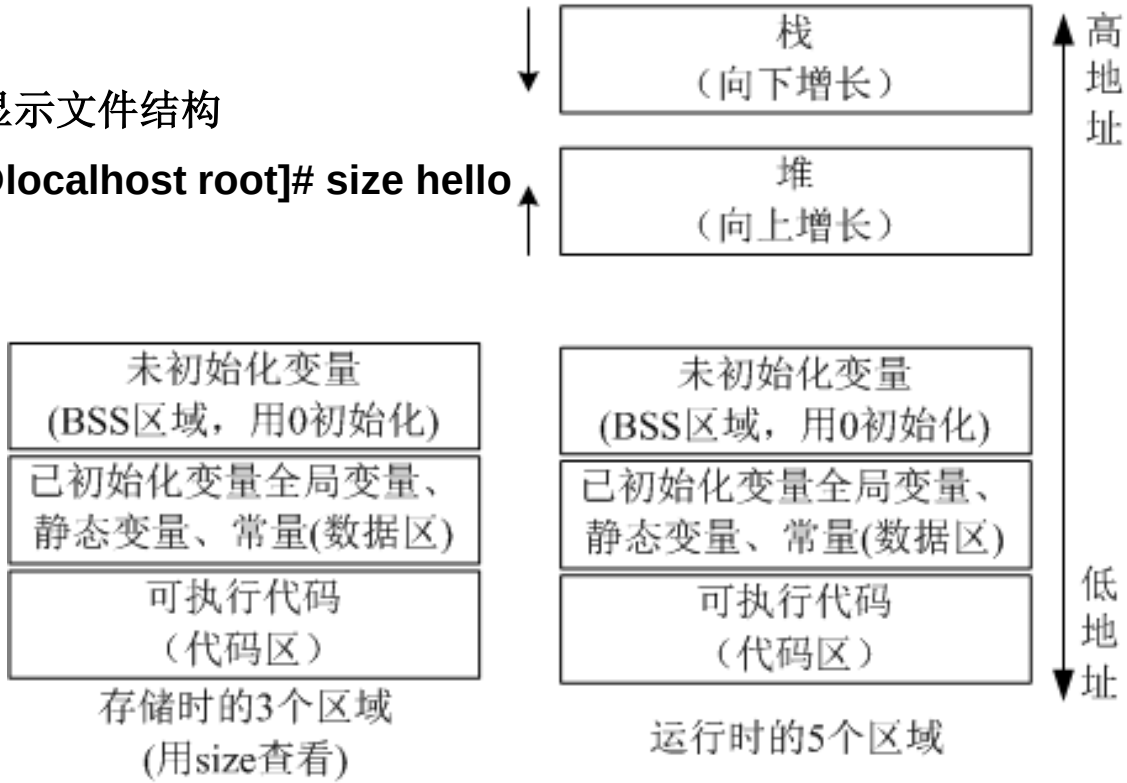
Linux虚拟内存的组织

- **Linux将4G虚拟空间划分为两个部分**
  - 用户空间与内核空间
  - 用户空间3G：从0到0xBFFFFFFF
  - 内核空间1G：从0xC0000000到4G。
  - 用户进程通常情况下只能访问用户空间的虚拟地址，不能访问内核空间。例外情况是用户进程通过系统调用访问内核空间。

用户进程地址空间(3G)

//size显示文件结构

[root@localhost root]# size hello



可执行程序存储结构

- **（1）.text**
  - 存放 CPU 执行的机器指令,代码区通常是只读的，防止程序意外地修改了它的指令。
- **（2）.data**
  - 该区包含被初始化的全局变量、静态变量.它们是在编译阶段被编译器存放在可执行目标文件的.data段中的.
- **（3）.BSS**
  - 未被初始化的全局变量和静态局部变量，编译的时候并未被分配空间，仅仅在.bss段中标记它们，当程序运行的时候才为它们在内存中分配空间，并把它们初始化为零或空指针。
- **（4）.rodata**
  - 该区包含常量数据（如字符串常量）在编译阶段被编译器存放在可执行目标文件的.rodata段中的。

编译后可执行代码

- 例:查看编译后可执行代码中几个基本段
- **#gcc -c test.c -o test**
- **#objdump –h test**

程序运行时内存结构

- （1）代码区(text)
  - 代码区指令根据程序设计流程依次执行，对于顺序指令，则只会执行一次（每个进程），如果反复，则需要使用跳转指令，如果进行递归，则需要借助栈来实现。
- （2）全局初始化数据区/静态数据区(data): 只初始化一次
- （3）未初始化数据区（BSS）:在运行时改变其值。
- （4）栈区（stack）:由编译器自动分配释放，存放函数的参数值、局部变量的值等。
- （5）堆区（heap）:用于动态内存分配。一般由程序员分配和释放，若程序员不释放，程序结束时有可能由 OS 回收。

程序执行时的内存分配情况

```
int a = 0; //a 在全局已初始化数据区
char *p1; //p1 在 BSS 区（未初始化全局变量）
main()
{
    int b; //b 在栈区
    char s[ ] = "abc";
    //s 为数组变量，存储在栈区 //“abc”为字符串常量，存储在已初始化数据区
    char *p1，p2; //p1、p2 在栈区
    char *p3 = "123456"; //123456\0  data数据区，p3 在栈区 .
    static int c =0； //c为局部静态数据，data数据区
    p1 = (char *)malloc(10);  //分配得来的 10 个字节的区域在堆区
    p2 = (char *)malloc(20); //分配得来的 20 个字节的区域在堆区
    free(p1);
    free(p2);
}
```

栈和堆的区别

- （1）管理方式不同
  - 栈编译器自动管理，无需程序员手工控制；
  - 堆空间的申请释放工作由程序员控制，容易产生内存泄漏。
- （2）空间大小不同。
  - 栈是向低地址扩展的数据结构，是一块连续的内存区域。因此，用户从栈获得的空间较小。
  - 堆是向高地址扩展的数据结构，是不连续的内存区域。堆获得的空间较灵活，也较大。

Linux的段机制

- Linux四个范围一样的段：0～4 G
  - 内核数据段
  - 内核代码段
  - 用户数据段
  - 用户代码段
- 各段属性不同
  - 内核段特权级为0
  - 用户段特权级为3
- 作用
  - 利用段机制来隔离用户数据和系统数据
  - 简化（避免）逻辑地址到线性地址转换；
  - 保留段的等级保护机制

4个GD（全局描述符）

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

K_CS:0000 0000 1100 1111 1001 1010 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111
K_DS:0000 0000 1100 1111 1001 0010 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111
U_CS:0000 0000 1100 1111 1111 1010 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111
U_DS:0000 0000 1100 1111 1111 0010 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111

- XXXX:基地址；hhhh：段界限
- **G位都是1（段长单位4KB）；P位都是1（段在内存）**
- K_CS(Index=2), kernel 4GB code at 0x0
- K_DS(Index=3), kernel 4GB data at 0x0
- U_CS(Index=4), USER 4GB code at 0x0
- U_DS(Index=5), USER 4GB data at 0x0

Linux的段机制

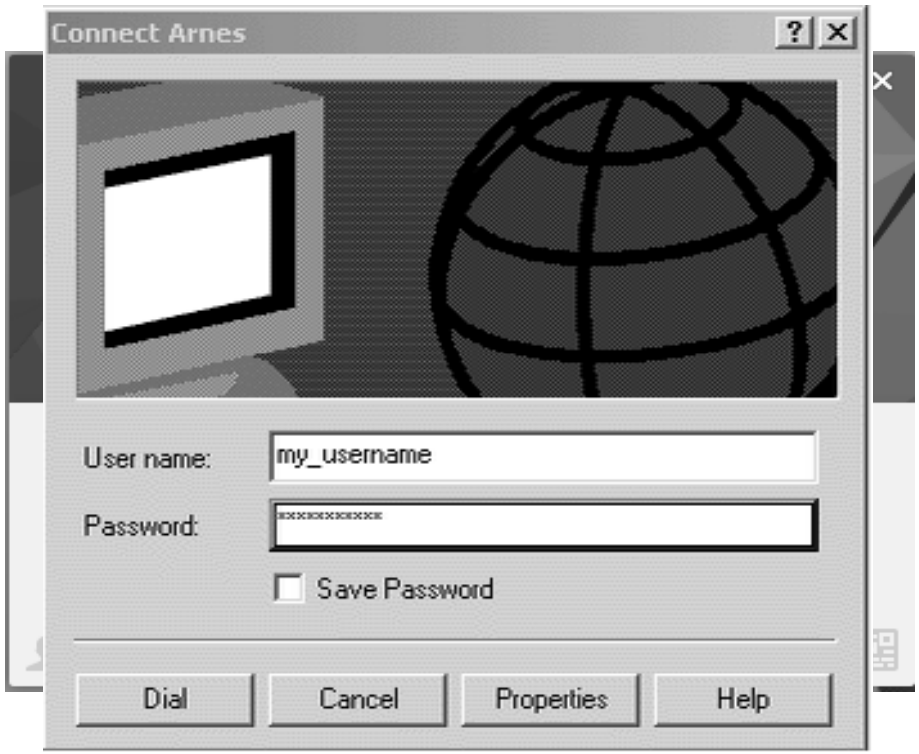
- 结论
  - 每个段是从0地址开始的4GB的虚拟空间，线性地址和虚地址保持一致。
  - 讨论和理解LINUX内核页式映射时，可以直接将虚拟地址当做线性地址，二者完全一致。
  - 段仅仅提供了一种保护机制。

课后编程练习

- 【进程，内存，Linux模块，驱动】在Linux环境下，编写一个小程序，获取进程中的某个函数或变量虚拟地址，以及其所在的物理内存地址，并指出页目录的地址和数据，页表的地址和数据，页号，页框号，页内偏移等信息。

10周第2次课

WINDOWS虚拟内存的应用

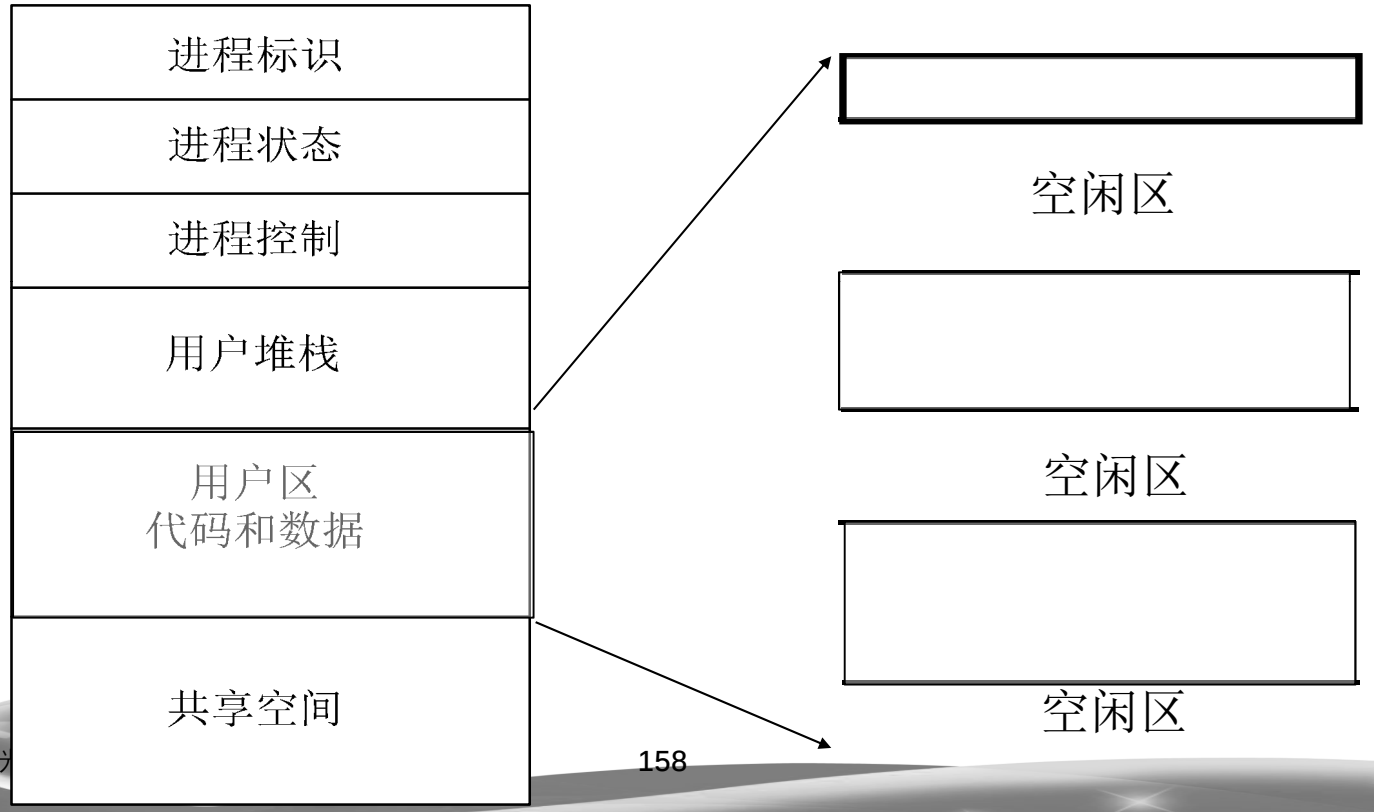


Windows进程虚拟内存空间

- 进程虚拟内存空间(32位)
  - 空间大小为4GB（2^{32}）
    - 用户区：2GB
      - 页交换区，可对换到外存
    - 系统区（2GB）
      - 在核心态可访问的存储区
  - 页面大小为4KB（2^{12}）

进程空间的分区

- 用户区内的空闲区(free, unallocated)



和虚拟内存操作相关的函数

- 获取OS系统信息[页面大小，分配粒度]
  - GetSystemInfo( )
- 分配和释放虚拟内存
  - VirtualAlloc( )和VirtualFree( )
- 获取内存状态【虚拟内存】
  - GlobalMemoryStatus( )
- 确定虚拟地址空间的状态
  - VirtualQuery( )或VirtualQueryEx( )
- 改变页面的保护属性
  - VirtualProtect( )和VirtualProtectEx( )
- 虚拟内存的读写
  - ReadProcessMemory( ) 和WriteProcessMemory( )

在应用程序中分配虚拟内存

```
LPVOID VirtualAlloc(
    LPVOID lpAddress,
    DWORD dwSize,
    DWORD flAllocationType,
    DWORD flProtect
    );
```

在应用程序中释放虚拟内存

```
BOOL VirtualFree(
    LPVOID lpAddress,
    DWORD dwSize,
    DWORD dwFreeType
    );
```

改变页面的保护属性

- **BOOL** VirtualProtect(
  PVOID pvAddress,
  DWORD dwSize,
  DWORD flNewProtect,
  PDWORD pflOldProtect
  );
- **BOOL** VirtualProtectEx**(**
  HANDLE hProcess,
  PVOID pvAddress,
  DWORD dwSize,
  DWORD flNewProtect,
  PDWORD pflOldProtect
  );

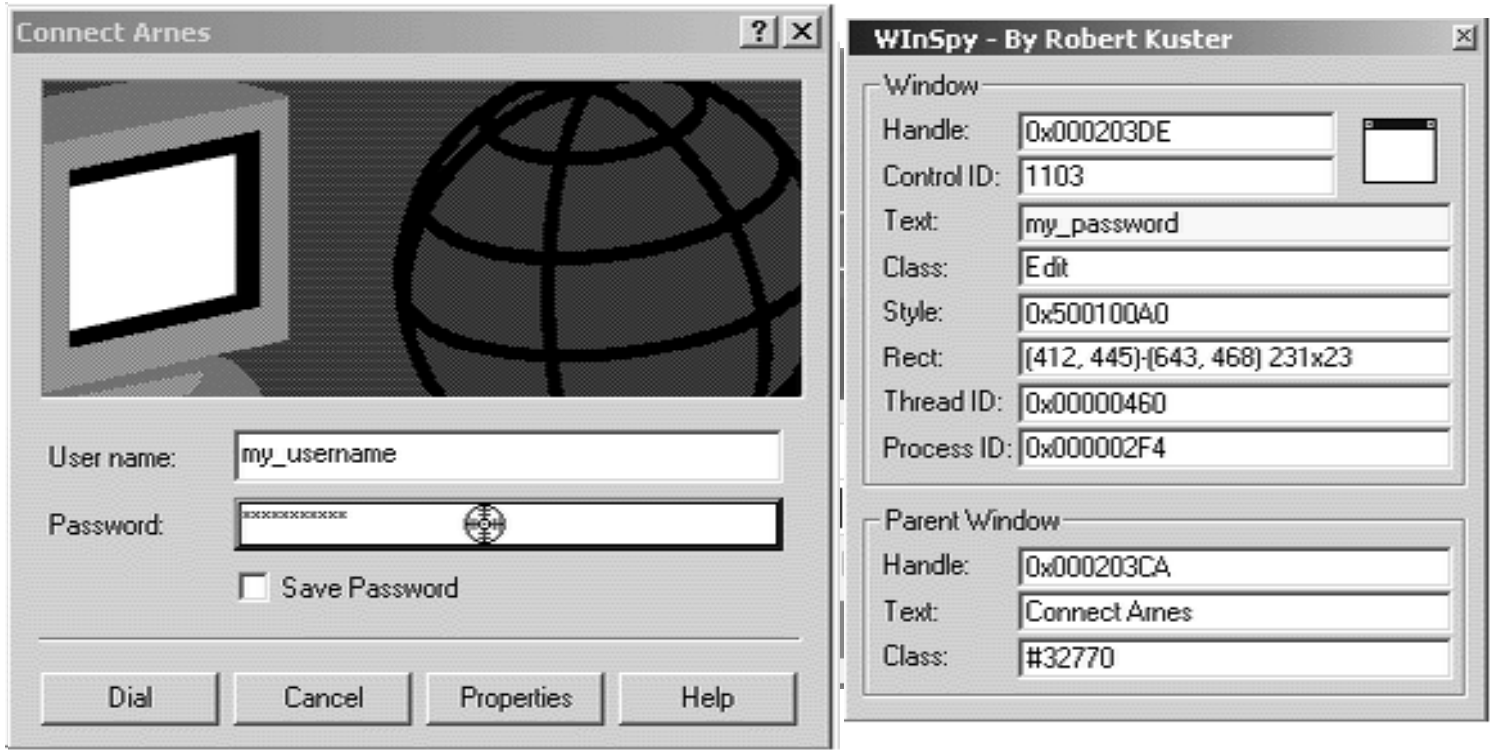
虚拟内存的读

```
BOOL ReadProcessMemory(
    HANDLE hProcess,
    LPCVOID lpBaseAddress,
    LPVOID lpBuffer,
    DWORD nSize,
    LPDWORD lpNumberOfBytesRead
    );
```

虚拟内存的写

```
BOOL WriteProcessMemory(
    HANDLE hProcess,
    LPVOID lpBaseAddress,
    LPVOID lpBuffer,
    DWORD nSize,
    LPDWORD lpNumberOfBytesWritten
    );
```

如何获取Password？



- 要“读取”某个控件的内容（例如**编辑框**）通常都是向其发送**WM_GETTEXT**消息。
  - 如果该编辑框属于远程进程（并具有 ES_PASSWORD式样，上面方法失效。
  - 所以，问题变成如何在远程进程的地址空间执行

SendMessage(hPwdEdit, WM_GETTEXT, nMaxChars, psBuffer);

- 通常有三种方法来解决这个问题
  - 1.将用户代码放入某个 DLL，然后通过 Windows 钩子映射该DLL到远程进程；
  - 2.将用户代码放入某个 DLL，通过 CreateRemoteThread 和 LoadLibrary 技术映射该DLL到远程进程；
  - 3.如果不写单独的 DLL，可将用户代码拷贝到远程进程（通过 WriteProcessMemory）并用 CreateRemoteThread 启动它的执行。
  - [请自己尝试各种方法。做出来可以替代大作业。]