



## 第6章 主存储器管理

---



# 主存储器管理

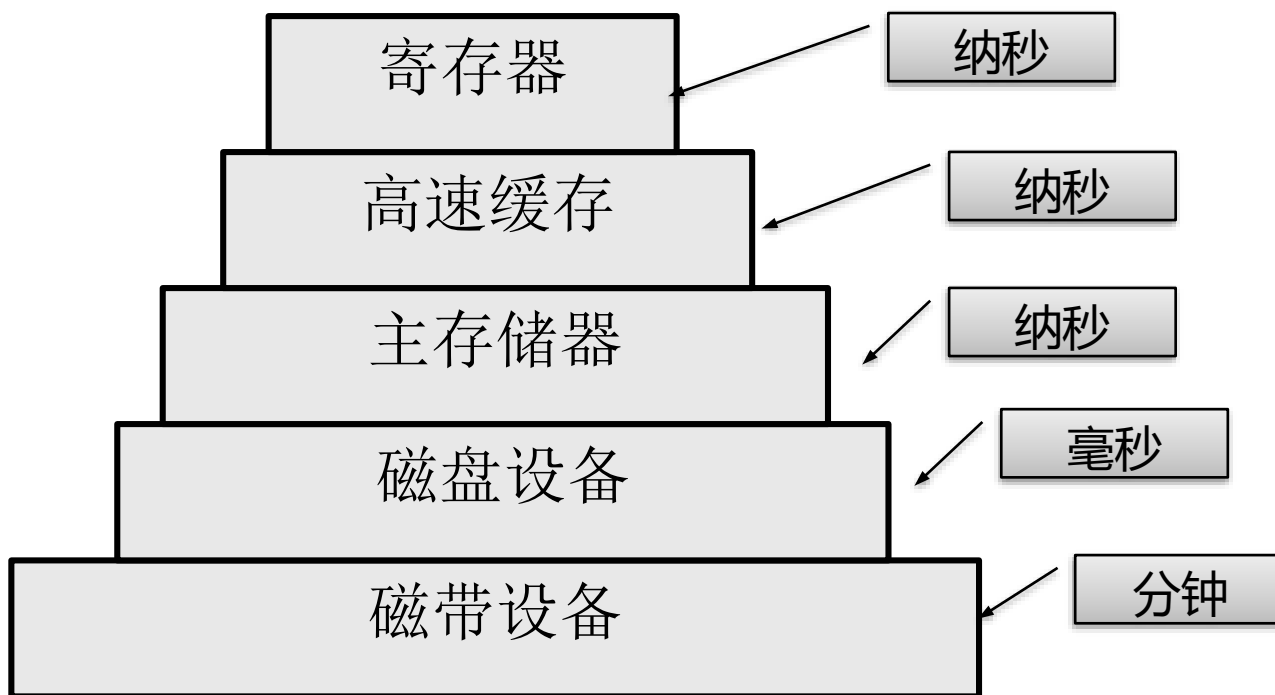
---

- 主存储器管理是操作系统的重要组成部分
- 管理对象：计算机系统主存储器（内存/主存）

# 存储器的层次结构

## ■ 发展动力：性能

- 主存储器是程序与数据的执行与处理必经载体
- 主存储器存取速度远远跟不上处理器处理速度





## ■ 主存储器分为两大部分：

- 系统区：存放操作系统内核程序与数据结构，供操作系统使用
- 用户区：存放应用程序与数据，往往被划分为一个或多个区域，供用户进程使用。

## ■ 存储器管理的主要目标：

- 为用户提供方便、安全和充分大的存储器，支持大型应用和系统程序及数据的使用。



# 主存储器管理的四个主要功能

## ① 存储空间的分配和回收

- 进程对主存的申请与释放

## ② 抽象与映射

- 存储器的抽象：相对进程而言，占有并使用一个地址连续的大数组，或者一个二维空间。
- 地址映射：逻辑地址与物理地址未必一致。



# 存储器管理的四个主要功能(续)

## ③ 隔离与共享

- 避免程序之间的相互干扰，确保进程对存储单元的独占式使用
- 在隔离的前提下，超越隔离机制，通过授权进程，允许共享访问。

## ④ 存储扩充

- 在逻辑上为用户提供一个比实际物理内存更大的存储空间



## 第6章 主存储器管理

# 6.1 存储器管理的基本概念



## 6.1 存储器管理的基本概念

- ① 逻辑地址：用户编程时所使用的地址。又称相对地址、虚地址。
- ② 地址空间：逻辑地址的集合。
- ③ 物理地址：内存中的地址。又称绝对地址、实地址。
- ④ 主存空间：物理地址的集合。
- ⑤ 地址变换：将逻辑地址转换为物理地址。又称地址映射、重定位。





## 6.1.1 程序的装入

- 为将一个用户源程序变为一个在内存中可执行的文件，通常要经历以下步骤：编译、链接、装入。
- 将装入模块装入内存有3种方式：
  - ① 绝对装入方式
  - ② 可重定位装入方式
  - ③ 动态运行时装入方式



# 1、绝对装入方式

- 编译时产生绝对地址的目标代码，绝对装入程序按照**装入模块中的地址**将程序及数据装入内存，**不需**对地址进行**变换**。
- 程序中使用的绝对地址可以在编译时给出，也可以由程序员直接赋予。



# 1、绝对装入方式(Cont.)

## ■ 特点:

- ① 知道程序驻留在内存中的确定位置，编译之后代码中包含了程序的物理地址。
- ② 装入模块之后，程序的逻辑地址与物理地址是完全相同的，不需要对程序和数据进行修改。
- ③ 只能将目标代码装入到内存中事先指定的位置，不适应多道程序环境的动态特性。
- ④ 通常在程序中采用符号地址，通过编译器，将符号地址转换为绝对地址。



## 2、可重定位装入方式

- 编译时产生相对地址的目标代码，由装入程序**根据内存当时的实际使用情况**，将装入模块装入到内存的适当地方。
- 地址变换分为两类：
  - 静态地址变换
  - 动态地址变换
- 依据地址变换进行分类
  - 静态重定位装入
  - 动态重定位装入/动态运行时装入



# 静态地址变换

## ■ 需求：

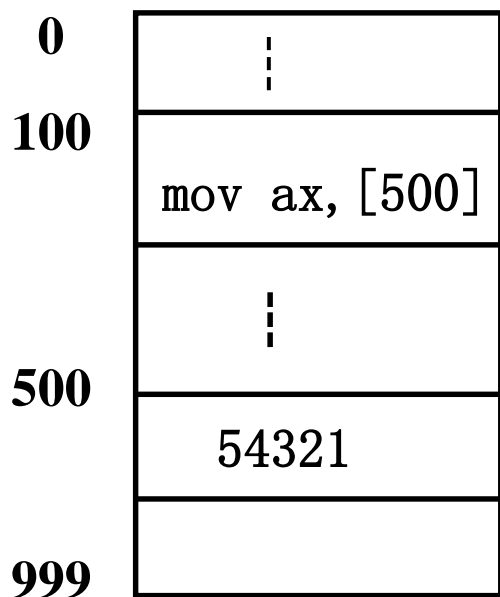
- 在作业装入时候，内存分配的空间与逻辑地址空间可能**不一致**，因此，作业访问指令和数据的实际地址与地址空间中的地址也**不一致**，因此需要相应调整

## ■ 静态地址变换

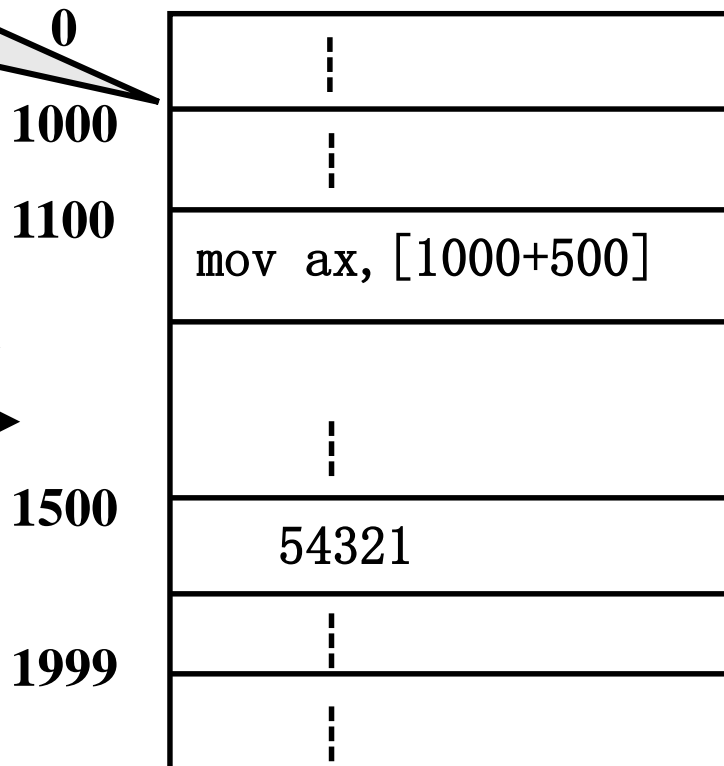
- 又称静态地址重定位，地址变换在程序装入时**一次完成**，以后不再改变。

# 静态地址变换示意图

将作业装入从**1000**开始的内存区域



重定位装入程序



作业的地址空间

注意：逻辑地址500在装入时  
转换为物理地址1500

空间



# 可重定位装入方式特点

- 物理地址 = **程序起始地址** + 逻辑地址
- 地址变换通常是在装入时一次完成的，以后不再改变，故称为**静态重定位**
- 不需硬件支持，且要求分配连续存储空间，但程序运行时**不能在内存移动**，难以实现数据与代码的共享。



### 3、动态运行时装入方式

---

- 在将装入模块**装入内存时并不进行**地址变换，而是在**程序执行过程**中进行地址变换。也称为**动态加载**。
- 特点：需要硬件支持，可以部分装入。
- 技术上需要**动态地址变换技术**支持。





# 动态地址变换

---

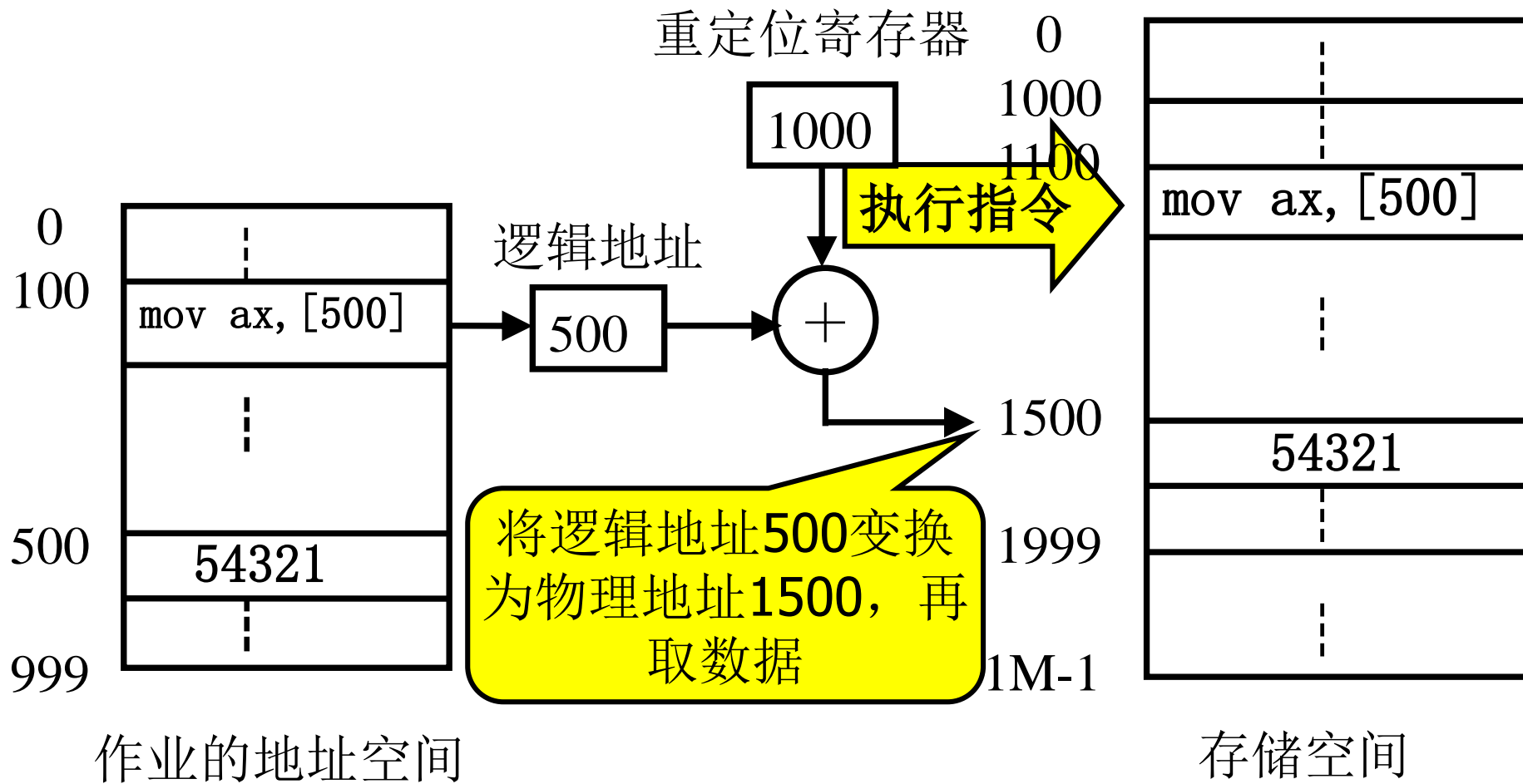
- 需求：

- 多道程序环境下，由于进程的调度，目标模块在装入内存后，可能被多次重新装入，导致每次装入的地址可能都不是相同的，造成了程序的移动。

- 动态地址变换：

- 又称动态重定位，在程序执行过程中，每次访问内存之前将待访问程序地址转换成内存地址。

# 动态地址变换示意图





# 动态运行时装入方式的具体机制

- 程序被装入内存后，程序中所引用的逻辑地址不作修改；程序的装入起始地址被加载到重定位寄存器。
- 程序运行时，当CPU引用主存地址时，硬件拦截此地址，在发送到主存储器之前，加上重定位寄存器的值。



# 动态运行装入方式的实现特点

- 多道程序中，重定位寄存器的内容存储在进程PCB中。当进程被调入时，重定位寄存器被重新设置，原有进程的重定位寄存器的内容随进程上下文切换而得到保护。
- 为了支持C语言的模型，处理器至少要有3个重定位寄存器，包括文本段、数据段和堆栈段，Intel x86有6个。



## 6.1.2 程序的链接

---

- 链接程序的功能是将经过编译或汇编后得到的目标模块以及所需的库函数装配成一个完整的装入模块。
- 实现链接的方式有三种：
  - ① 静态链接
  - ② 装入时动态链接
  - ③ 运行时动态链接



# 静态链接

---

- 静态链接：
  - 在程序运行之前，将各目标模块及其所需的库函数装配成一个完整的装入模块。
  - 部分模块的起始地址可能不再是0，需要进行二次修改
  - 这一完整的装入模块称作可执行文件
  - 链接完成后，可执行文件不会被拆开，而是在运行时直接装入内存中



# 装入时动态链接

- 装入时动态链接：
  - 源程序编译后所得到的目标模块在装入内存时边装入边链接。
  - 若在装入目标模块时，如果发生一个外部模块调用，就会引起装入程序去寻找相应的外部目标模块，并将其装入内存，同时还要修改目标模块中的相对地址
- 优点：
  - 便于软件版本的修改和更新
  - 便于目标模块的共享。



# 运行时动态链接

---

- 运行时动态链接：
  - 将某些目标模块的链接推迟到执行时才进行。
  - 在执行过程中，若发现一个被调用模块尚未装入内存时，由OS去找到该模块，将它装入内存并链接到调用者模块上。
  - 如果该模块没有被调用，则操作系统不会加载该模块
- 特点：加快了程序装入，节省了内存。





## 6.1.3 内存保护

---

- 内存保护：
  - 是防止一个进程有意或无意破坏操作系统或其他进程。
- 常用的存储保护方法有：
  - ① 界限寄存器法
  - ② 存储保护键
  - ③ 环保护机制
  - ④ 访问权限



# 1、界限寄存器法

---

- 通过对每个进程设置一对界限寄存器来防止越界访问，达到存储保护的目的。
- 界限寄存器方法有两种实现：
  - 上下界寄存器
  - 基址限长寄存器



# 上下界寄存器方法

- 上下界寄存器方法：
  - 用上、下界寄存器分别存放作业存储空间的结束地址和开始地址。
  - 在作业运行过程中，将每一个访问内存的地址都同这两个寄存器的内容进行比较，若超出了上下界寄存器的范围则产生越界中断。即：
$$\text{下界寄存器} \leq \text{地址} < \text{上界寄存器}$$



# 基址限长寄存器方法

---

- 基址、限长寄存器方法：
  - 用基址和限长寄存器分别存放作业存储空间的起始地址及作业长度。
  - 当作业执行时，将每一个访问内存的相对地址和这个限长寄存器比较，若逻辑地址超过限长则产生越界中断。



## 2、存储保护键

- 通过保护键匹配来判断存储访问方式是否合法。
- 为每个存储块分配一个保护键，相当于一把锁；
- 进入系统的每个作业赋予一个保护键，相当于一把钥匙。
- 当作业运行时，检查钥匙和锁是否匹配，若二者匹配，则允许访问。否则发出保护性中断信号



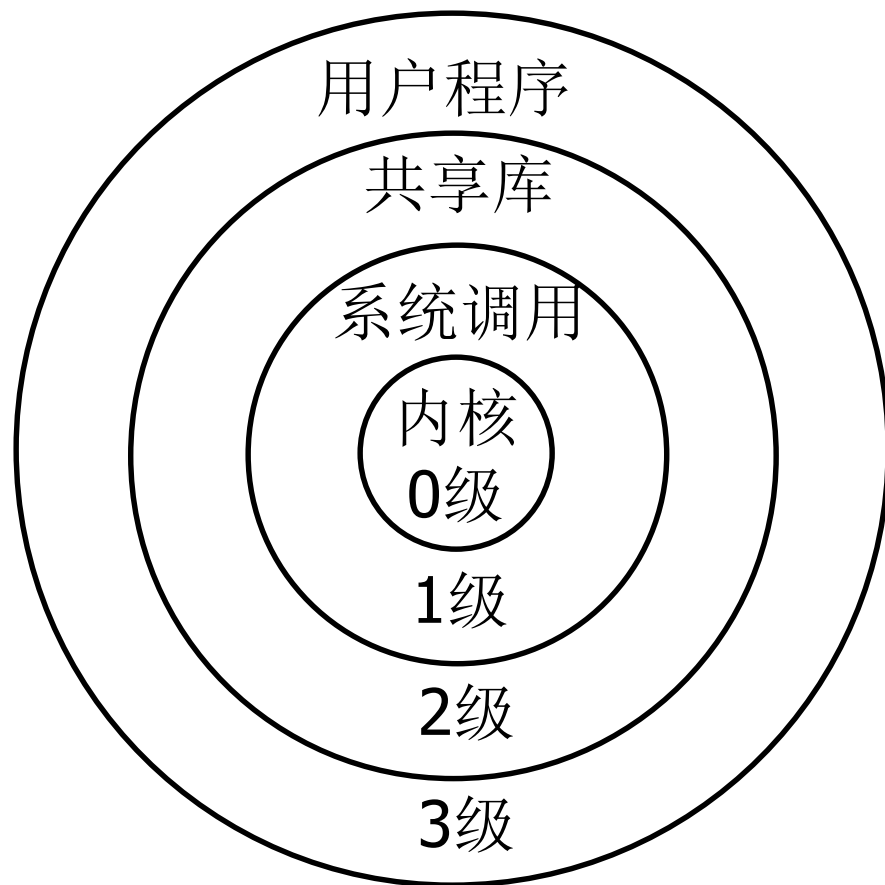
### 3、环保护机制

---

- 处理器状态分为多个环，分别具有不同的存储访问特权级，通常环的编号越小，特权级越高。
- 例如：规定低编号环具有高优先权。操作系统核心处于0环，某些重要实用程序和操作系统服务处于中间环，一般应用程序占据外环。

# 环保护的基本原则

- 环保护的基本原则是：
  - 一个程序可以**访问驻留在相同环或较低特权环**（即较高环编号）中的数据；
  - 一个程序可以**调用驻留在相同环或较高特权环**（即较低环号）中的服务。



Pentium中的环形保护结构



## 4、访问权限

---

- 除上述保护方案外，还有多种存取权限：
  - 禁止做任何操作,forbidden
  - 只能执行,execute only
  - 只能读,read only
  - 读/写,read & write
  - .....





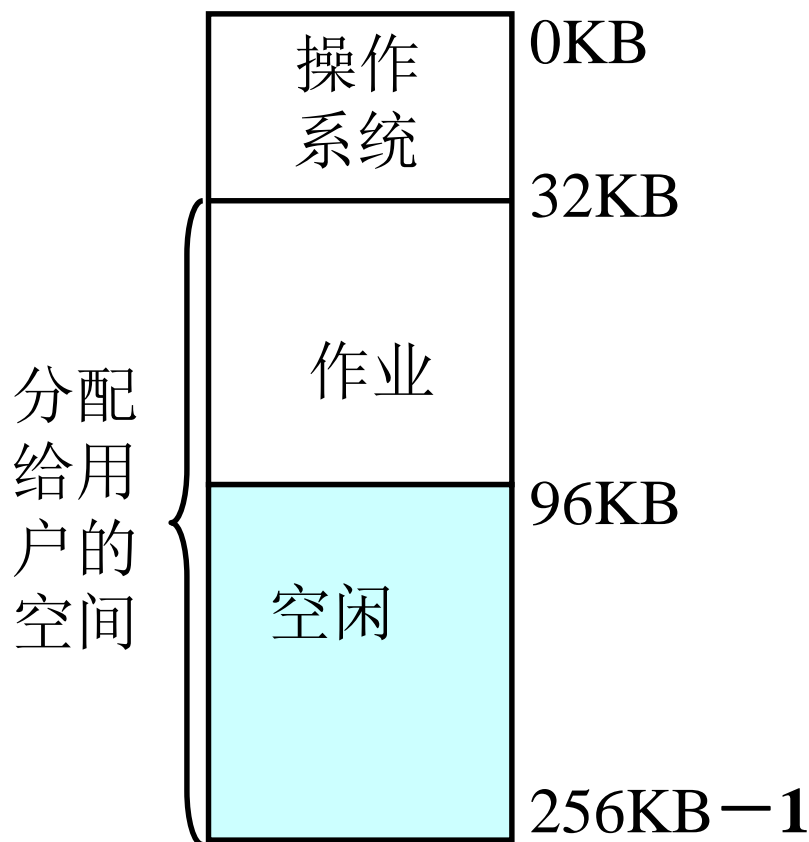
---

## 第6章 主存储器管理

# 6.2 单一连续分配

# 单一连续分配

- 单一连续分配方式（或称**单用户连续分配**），内存分为系统区和用户区。**系统区**给操作系统使用，**用户区**给一道用户作业使用。
- 特点：管理简单，只需很少的软硬件支持；但各类资源的利用率不高。



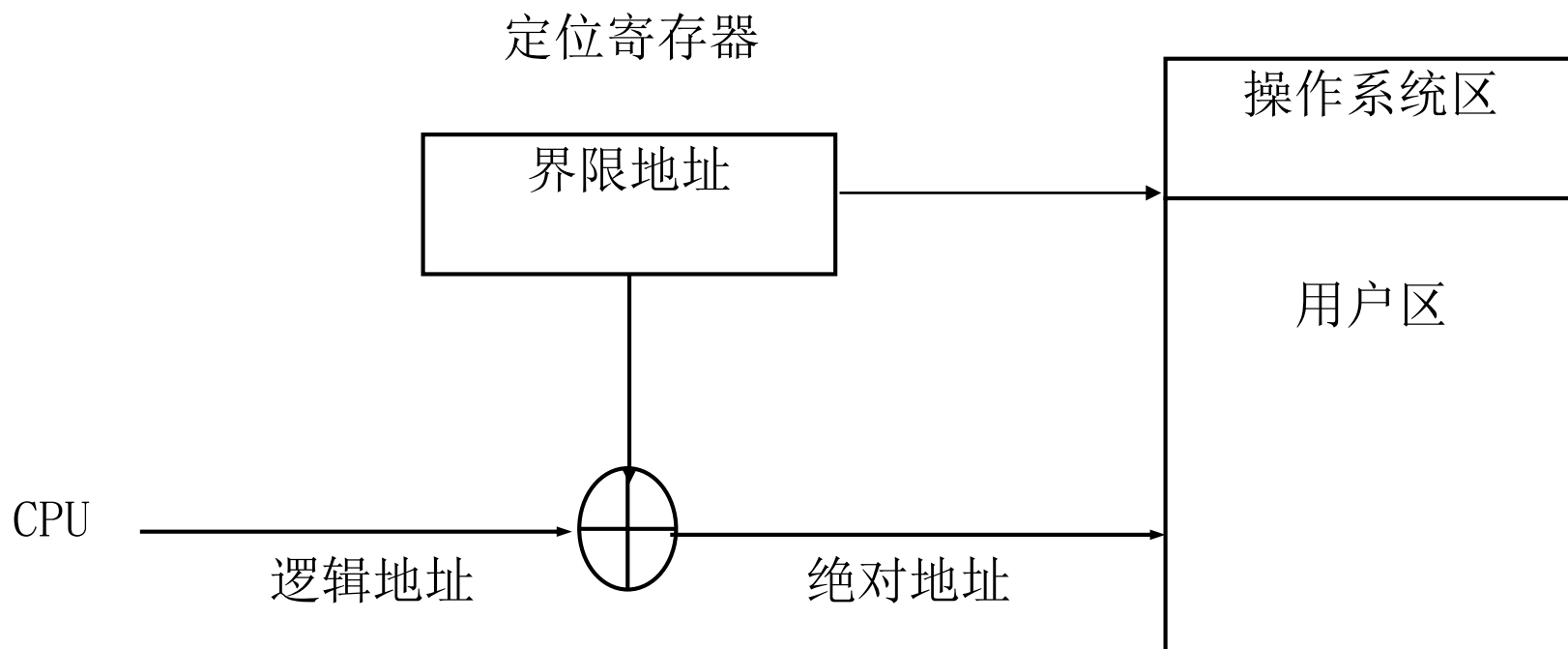


# 单一连续分配的缺点

---

- 单一连续分配为静态分配方式;
- 处理器和外部设备串行工作;
- 一个作业独占主存储空间, 降低存储空间的利用率;
- 计算机的外围设备利用率不高。

# 动态重定位单一连续分配





---

## 第6章 主存储器管理

# 6.3 分区存储管理



# 分区存储管理

- 当我们将多个进程同时放在内存中时，就需要考虑如何为调入内存的进程分配内存空间。
- 分区存储管理是多道程序系统中采用的一种最简单的方法。它把系统的内存划分为若干大小不等的区域，**操作系统占一个区域**，其他区域由**并发进程共享**，每个进程占一个区域。
- 分区存储管理分为：
  - 固定分区
  - 动态分区



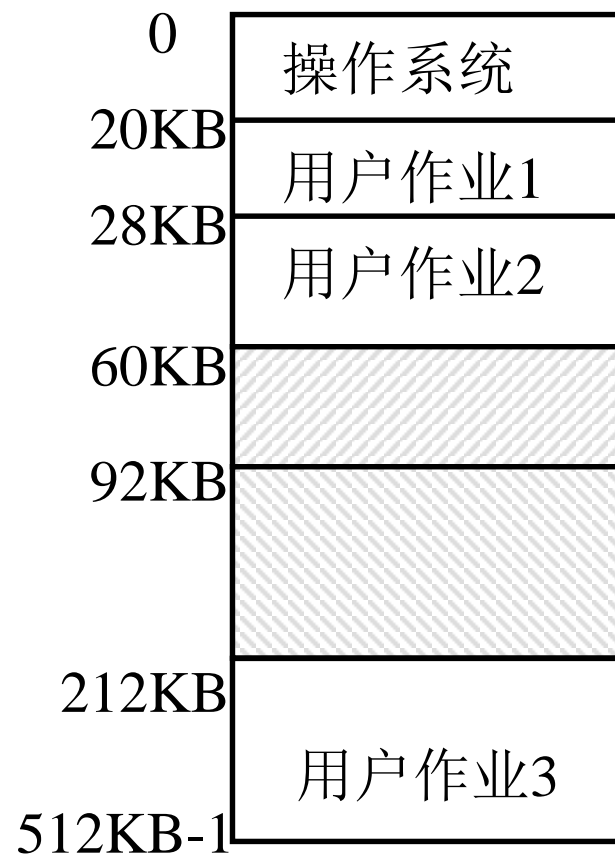
## 6.3.1 固定分区

- 固定分区存储管理方法将内存空间划分为若干个固定大小的分区，**每个分区中可以装入一道程序**。分区的位置及大小**在运行期间不能改变**。
- 为了便于管理内存，系统需要建立一张**分区说明表/分区使用表**，其中记录系统中的分区数目、分区大小、分区起始地址及状态。

# 分区说明表例

分区说明表

分区号	大小	起始地址	状态
1	8KB	20KB	已分配
2	32KB	28KB	已分配
3	32KB	60KB	未分配
4	120KB	92KB	未分配
5	300KB	212KB	已分配



内存布局图





# 固定分区的内存管理过程

- 分区分配：
  - 当有用户程序要装入时，由内存分配程序检索分区使用表，从中找出一个能满足要求的空闲分区分配给该程序
  - 然后修改分区说明表中相应表项的状态；
  - 若找不到大小足够的分区，则拒绝分配内存。
- 分区回收：
  - 当程序执行完毕不再需要内存资源时，释放程序占用的分区，管理程序只需将对应分区的状态置为未分配即可。
- 特点：最早的多道程序存储管理方式，不能充分利用内存，存在内存碎片。



## 6.3.2 动态分区存储管理

---

- 动态分区存储管理又称为可变分区存储管理
- 实现思想
  - 根据作业大小动态地建立分区，并使分区的大小正好适应作业的需要。
  - 因此系统中分区的大小是可变的，分区的数目也是可变的。

# 1、动态分区存储管理示意图

- 初始时，整个用户区是1个空闲块
- 作业1进入，成为2个分区
- 作业2进入，成为3个分区
- 作业3进入，成为4个分区
- 作业1结束，成为4个分区
- 作业3结束，成为3个分区
- 作业2结束，成为1个分区



用户区



## 2、动态分区中的数据结构

---

- 在动态分区中常用的数据结构有：
  - 空闲分区表
    - 用一个空闲分区表来登记系统中的空闲分区。其表项类似于固定分区。
  - 空闲分区链
    - 将内存中的空闲分区以链表方式链接起来，构成空闲分区链。

# 空闲分区表示意图

0	操作系统
24KB	空闲 (8K)
32KB	已分 (96K)
128KB	空闲 (12K)
140KB	已分 (108K)
248KB	空闲 (8K)
256KB-1	

内存布局图

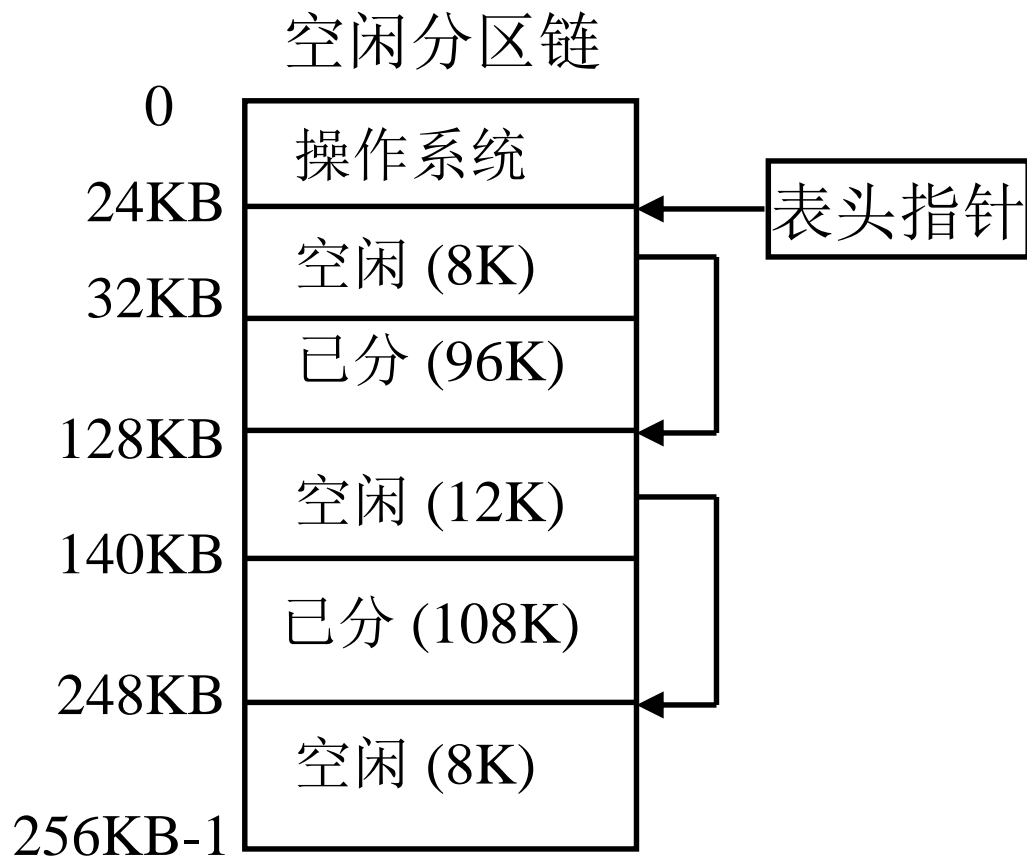
空闲分区表

分区号	大小	起始地址
1	8KB	24KB
2	12KB	128KB
3	8KB	248KB
4	...	...
5	...	...

# 空闲分区链示意图



内存布局图





### 3、分区分配算法

---

- 目前常用的分区分配算法有以下几种：
  - ① 首次适应算法
  - ② 循环首次适应算法
  - ③ 最佳适应算法
  - ④ 最坏适应算法



## ① 首次适应算法

- 首次适应算法又称最先适应算法，该算法要求空闲分区**按地址递增的次序排列**（注意：不是大小递增！）。
- 在进行内存分配时，从空闲分区表（或空闲分区链）表首（或表头节点）开始顺序查找，直到找到第一个能满足其大小要求的空闲分区为止。
- 然后，再按照作业大小，从该分区中划出一块内存空间分配给请求者，余下的空闲分区仍然留在空闲分区表（或空闲分区链）中。
- 特点：优先利用内存低地址端，高地址端有大空闲区。但低地址端有许多小空闲分区时会增加查找开销。





## ② 循环首次适应算法

- 循环首次适应算法又称下次适应算法，它是首次适应算法的变形，改进首次适应算法低地址使用频繁问题。
- 该算法在为进程分配内存空间时，从上次找到的空闲分区的下一个空闲分区开始查找，直到找到第一个能满足其大小要求的空闲分区为止。
- 然后，再按照作业大小，从该分区中划出一块内存空间分配给请求者，余下的空闲分区仍然留在空闲分区表（或空闲分区链）中。
- 特点：使存储空间的利用更加均衡，但会使系统缺乏大的空闲分区。



### ③ 最佳适应算法

- 最佳适应算法要求空闲分区按容量大小递增的次序排列。
- 在进行内存分配时，从空闲分区表（或空闲分区链）首开始顺序查找，直到找到第一个能满足其大小要求的空闲分区为止。
- 如果该空闲分区大于作业的大小，则从该分区中划出一块内存空间分配给请求者，将剩余空闲区仍然留在空闲分区表（或空闲分区链）中。
- 按最佳适应算法为作业分配内存，就能把既满足作业要求又与作业大小最接近的空闲分区分配给作业。
- 特点：保留了大的空闲区，但分割后的剩余空闲区很小，且排序需要耗费代价



## ④ 最坏适应算法

- 最坏适应算法要求空闲分区**按容量大小递减**的次序排列。
- 在进行内存分配时，先检查空闲分区表（或空闲分区链）中的第一个空闲分区，若第一个空闲分区小于作业要求的大小，则分配失败；
- 否则从该空闲分区中划出与作业大小相等的一块内存空间分配给请求者，余下的空闲分区仍然留在空闲分区表（或空闲分区链）中。
- 特点：剩下的空闲区比较大，但当大作业到来时，其存储空间的申请往往得不到满足。

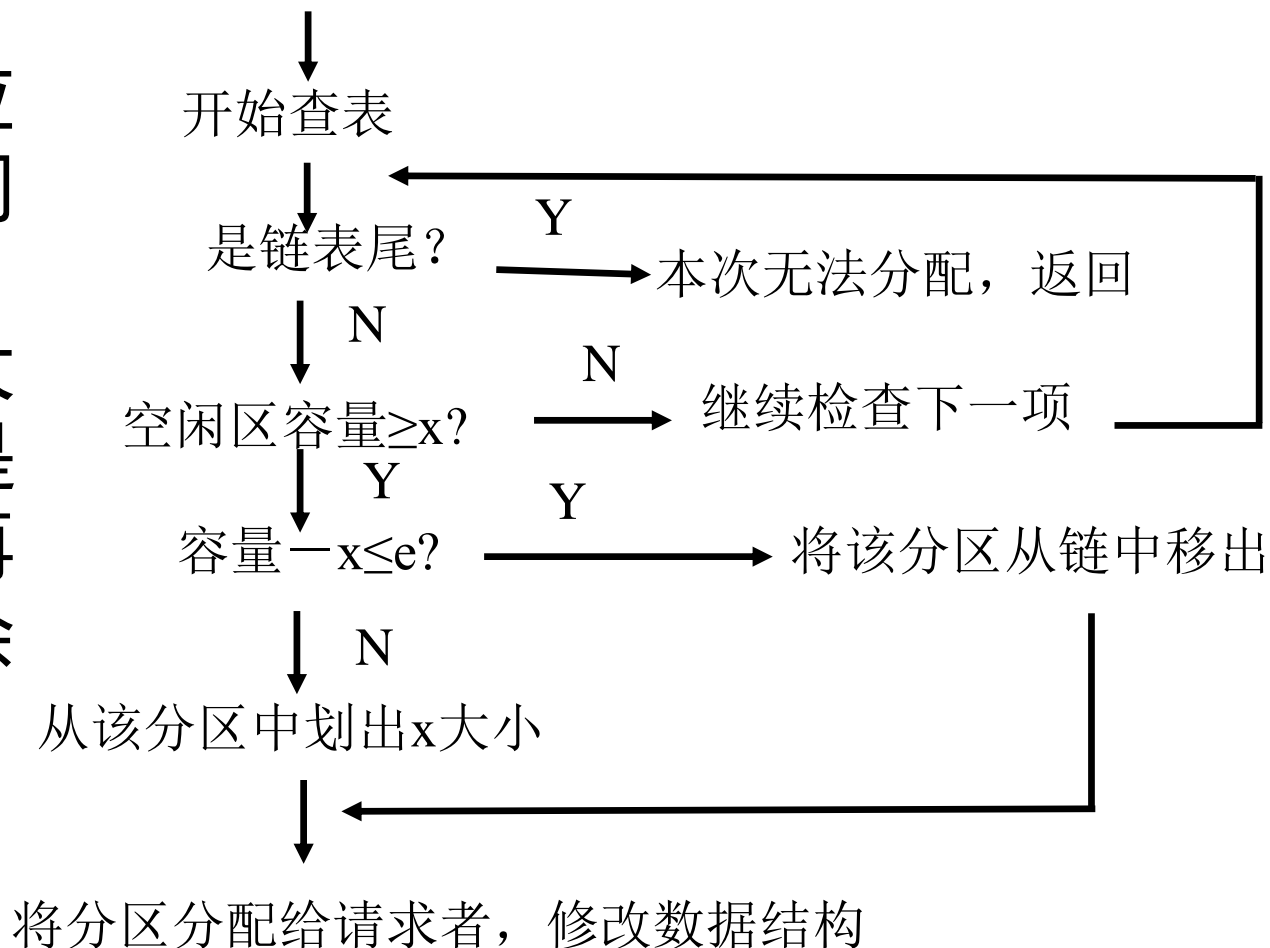


## 4、如何衡量分配算法的好坏

- 对于某一个作业序列来说，若某种分配算法能将该作业序列中所有作业安置完毕，则称该分配算法对这一作业序列合适，否则称为不合适。
- 根据模拟结果显示，首次适应和最佳适应在执行时间和利用空间方面都好于最坏适应。
- 首次适应和最优适应在利用空间方面能力想当，但首次适应更快

## 5、分区分配的具体方法

- 以首次适应算法及空闲链表为例，申请分区大小为 $x$ ， $e$ 是规定的不再分割的剩余区大小。



# 例

- 下表给出了某系统的空闲分区表，系统采用可变式分区存储管理策略。现有以下作业序列：96K、20K、200K。若用首次适应算法和最佳适应算法来处理这些作业序列，试问哪一种算法可以满足该作业序列的请求？

分区号	大小	起始地址
1	32K	100K
2	10K	150K
3	5K	200K
4	218K	220K
5	96K	530K

# 例--采用首次适应算法分配1

这里假设，从每个分区的外部开始切割分区，而不是起始地址来切割。

- 申请96K，
- 选中4号分区，进行分配后4号分区还剩下122K；

分区号	大小	起始地址
1	32K	100K
2	10K	150K
3	5K	200K
4	218K	220K
5	96K	530K

分区号	大小	起始地址
1	32K	100K
2	10K	150K
3	5K	200K
4	122K	220K
5	96K	530K

## 例--采用首次适应算法分配2

- 申请20K,
- 选中1号分区, 分配后剩下12K;

分区号	大小	起始地址
1	32K	100K
2	10K	150K
3	5K	200K
4	122K	220K
5	96K	530K

分区号	大小	起始地址
1	12K	100K
2	10K	150K
3	5K	200K
4	122K	220K
5	96K	530K



## 例--采用首次适应算法分配3

- 申请200K,
- 现有的五个分区都无法满足要求, 该作业等待。
- 显然采用首次适应算法进行内存分配, 无法满足该作业序列的需求。

分区号	大小	起始地址
1	12K	100K
2	10K	150K
3	5K	200K
4	122K	220K
5	96K	530K

# 例--采用最佳适应算法分配1

- 申请96K,
- 选中5号分区, 5号分区大小与申请空间大小一致, 应从空闲分区表中删去该表项;

分区号	大小	起始地址
1	32K	100K
2	10K	150K
3	5K	200K
4	218K	220K
5	96K	530K

分区号	大小	起始地址
1	32K	100K
2	10K	150K
3	5K	200K
4	218K	220K

## 例--采用最佳适应算法分配2

- 申请20K,
- 选中1号分区, 分配后1号分区还剩下12K;

分区号	大小	起始地址
1	32K	100K
2	10K	150K
3	5K	200K
4	218K	220K

分区号	大小	起始地址
1	12K	100K
2	10K	150K
3	5K	200K
4	218K	220K

## 例--采用最佳适应算法分配3

- 申请200K,
- 选中4号分区, 分配后剩下18K。

分区号	大小	起始地址
1	12K	100K
2	10K	150K
3	5K	200K
4	218K	220K

分区号	大小	起始地址
1	12K	100K
2	10K	150K
3	5K	200K
4	18K	220K

# 课下自己练

- 下表给出了某系统的空闲分区表，系统采用可变式分区存储管理策略。现有以下作业序列：申请150kb，申请50kb，申请90kb，申请80kb
- 若用首次适应算法和最佳适应算法来处理这些作业序列，试问哪一种算法可以满足该作业序列的请求？

分区号	大小	起始地址
1	300K	100K
2	112K	500K



## 6、分区回收

---

- 回收分区时，应将空闲区插入适当位置，此时有以下四种：
  - 回收分区r上面邻接一个空闲分区
  - 回收分区r下面邻接一个空闲分区
  - 回收分区r上面、下面各邻接一个空闲分区
  - 回收分区r不与任何空闲分区相邻

# 回收分区r上邻接一个空闲分区

- 此时应将回收区r与上邻接分区F1合并成一个连续的空闲区；
- 合并分区的首地址为空闲区F1的首地址，
- 其大小为二者之和。



# 回收分区r下邻接一个空闲分区

- 此时应将回收区r与下邻接分区F2合并成一个连续的空闲区；
- 合并分区的首地址为回收分区r的首地址，
- 其大小为二者之和。





# 回收分区r上下邻接空闲分区

- 此时应将回收区r与上、下邻接分区合并成一个连续的空闲区；
- 合并分区的首地址为与r上邻接空闲区F1的首地址，
- 其大小为三者之和，
- 且应将r下邻接的空闲区F2从空闲分区表(或空闲分区链)中删去。





# 回收分区r不与任何空闲分区相邻

- 这时应为回收区单独建立一个新表项，填写分区大小及起始地址等信息，并将其加入到空闲分区表(或空闲分区链)中的适当位置。

问题：空闲分区的个数在上述几种情况下如何变化？



## 6.3.3 可重定位分区分配

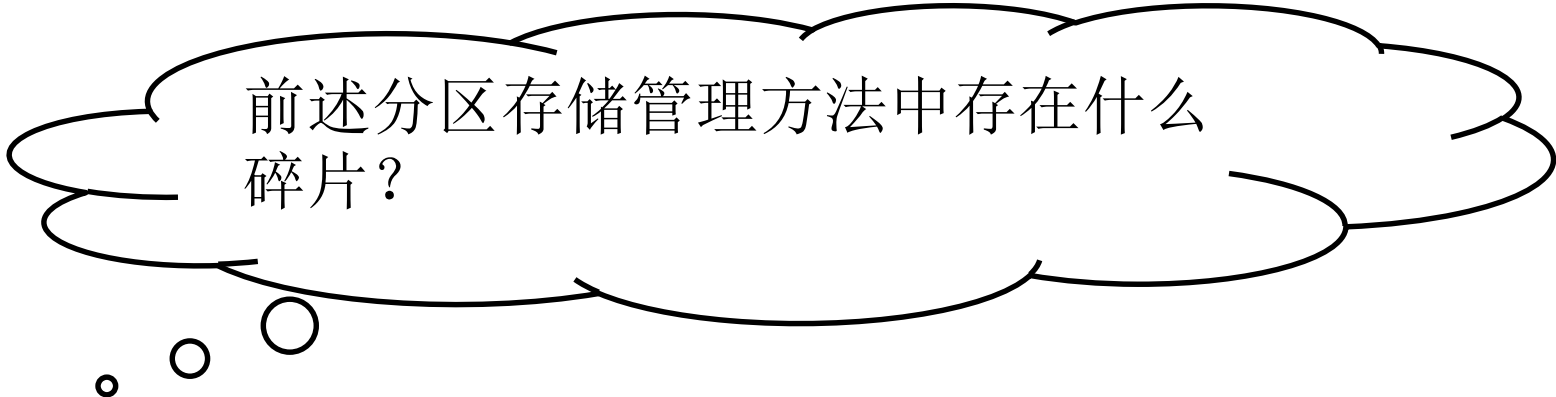
---

- 分区存储管理中，必须把作业装入到一片连续的内存空间中。这种分配方法能满足多道程序设计的需要，但存在**碎片问题**。
- 碎片也可称为零头，是指内存中无法被利用的存储空间。



# 内部碎片和外部碎片

- 内部碎片是指分配给作业的存储空间中未被利用的部分
- 外部碎片是指系统中无法利用的小存储块。



前述分区存储管理方法中存在什么碎片？

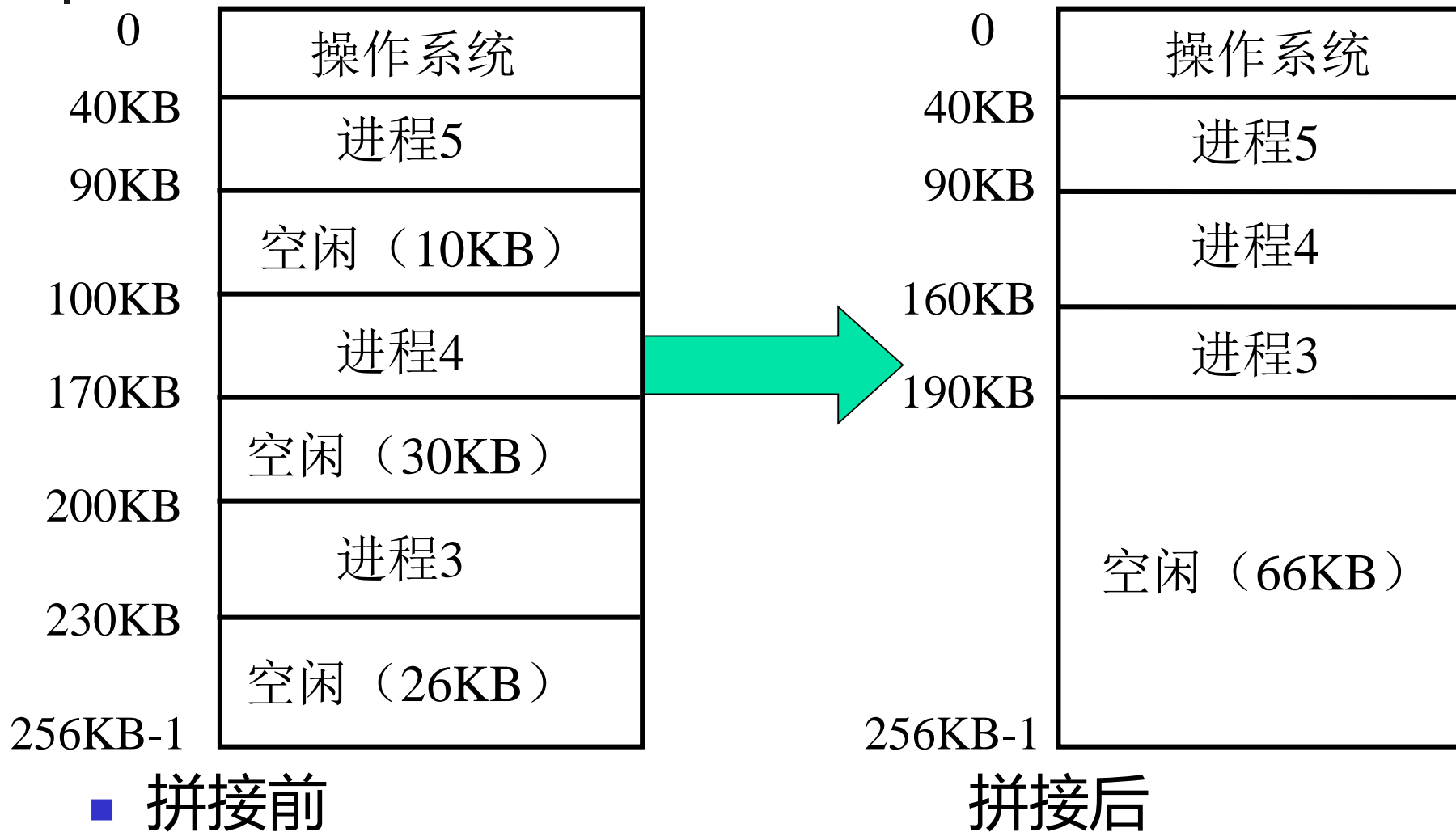


# 解决碎片问题的办法

---

- 拼接：解决碎片问题的办法之一，即通过移动把多个分散的小分区拼接成一个大分区，也可称为紧缩或紧凑。
- 不足：要耗费大量处理机时间。

# 拼接示意图





# 拼接需要的技术支持

## ① 拼接后程序在内存的位置发生变化怎么办？

- 需要**动态重定位技术支持/动态地址变换技术**。

## ② 空闲区放在何处？

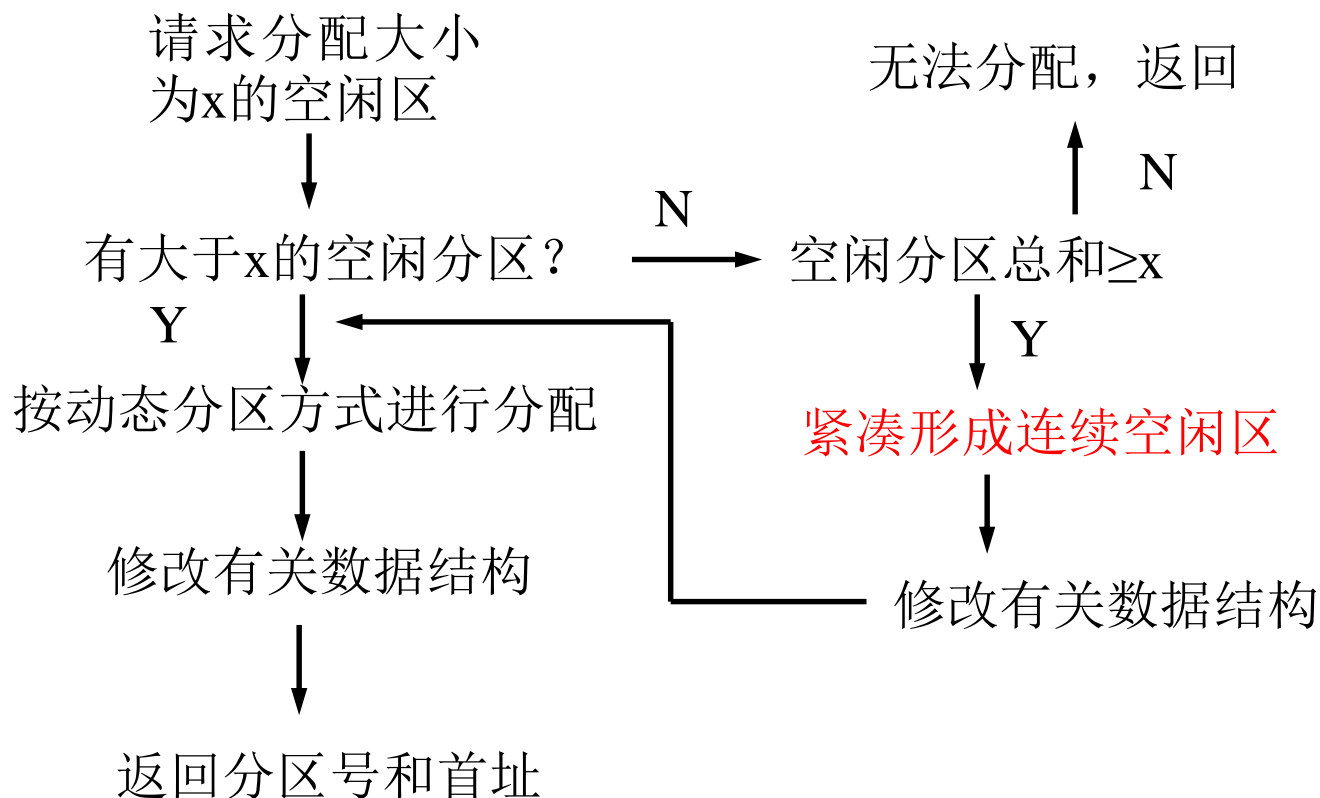
- 拼接后的空闲区放在何处不能一概而论，应根据移动进程的大小和数量多少来决定。

## ③ 拼接的时机问题？

- 时机1：回收分区时拼接：
  - 只有一个空闲区，但拼接频率过高增加系统开销。
- 时机2：找不到足够大的空闲区且系统空闲空间总量能满足要求时拼接：
  - 拼接频率小于前者，空闲区管理稍复杂，也可以只拼接部分空闲区。

# 可重定位分区分配技术

- 可重定位分区分配算法与动态分区分配算法基本相同，差别仅在于：在这种分配算法中增加了拼接功能。







---

## 第6章 主存储器管理

# 6.4 伙伴系统



# 伙伴系统(Knuth, 1973)

- 固定分区存储管理限制了内存中的进程数，动态分区的拼接需要大量时间，而伙伴系统是一种较为实用的动态存储管理办法。
- 主要思想：
  - 伙伴系统采用伙伴算法对空闲内存进行管理。
  - 该方法通过不断以 $1/2$ 的形式来分割大的空闲存储块，从而获得小的空闲存储块。当内存块释放时，应尽可能合并空闲块。



# 1、伙伴系统的内存分配

- 设系统初始时可供分配的空间为 $2^m$ 个单元。
- 当进程申请大小为 $n$ 的空间时，设 $2^{i-1} < n \leq 2^i$ ，则为进程分配大小为 $2^i$ 的空间。
- 如系统不存在大小为 $2^i$ 的空闲块，则查找系统中是否存在大于 $2^i$ 的空闲块 $2^{i+1}, 2^{i+2} \dots$ ，若找到则对其进行对半划分，直到产生大小为 $2^i$ 的空闲块为止。



## 2、伙伴系统的内存回收

- 当一块被分成两个大小相等的块时，这两块称为**伙伴**。
- 当进程释放存储空间时，应检查释放块的伙伴是否空闲，若空闲则合并。
- 如果这个较大的空闲块也存在空闲伙伴，此时也应合并。
- 重复上述过程，直至没有可以合并的伙伴为止。

### 3、伙伴地址公式

- 设某空闲块的开始地址为 $d$ ，长度为 $2^k$ ，其伙伴的开始地址为：
  - $\text{Buddy}(k, d) = d + 2^k$ ，若  $d \% 2^{k+1} = 0$
  - $\phantom{\text{Buddy}(k, d)} = d - 2^k$ ，若  $d \% 2^{k+1} = 2^k$
- 如果参与分配的 $2^m$ 个单元从 $a$ 开始，则长度为 $2^k$ 、开始地址为 $d$ 的块，其伙伴的开始地址为：
  - $\text{Buddy}(k, d) = d + 2^k$ ，若  $(d - a) \% 2^{k+1} = 0$
  - $\phantom{\text{Buddy}(k, d)} = d - 2^k$ ，若  $(d - a) \% 2^{k+1} = 2^k$



## 4、例子

---

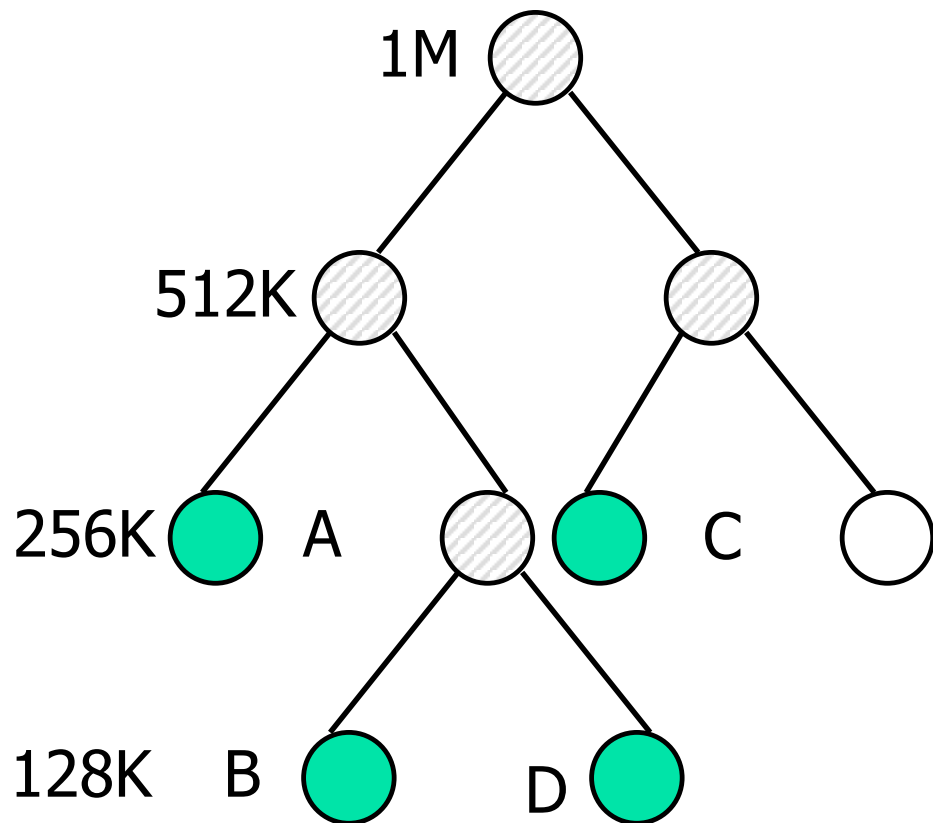
- 设系统中初始内存空间大小为1MB，进程请求和释放空间的操作序列为：
  - 进程A申请200KB； B申请120KB； C申请240KB； D申请100KB；
  - 进程B释放； E申请60KB；
  - 进程A、C释放；
  - 进程D释放； 进程E释放。

## 分配过程示意图

<div>■</div>	0	128K	256K	384K	512K	640K	768K	896K	1M
初始状态	<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>								
A申请200	A		256K			512K			
B申请120	A		B	128K	512K				
C申请240	A		B	128K	C		256K		
D申请100	A		B	D	C		256K		
B释放	A		128K	D	C		256K		
E申请60	A		E	64	D	C		256K	
A释放	256K		E	64	D	C		256K	
C释放	256K		E	64	D	512K			
D释放	256K		E	64	128K	512K			
E释放									

## 5、伙伴系统的二叉树表示

- 可以用二叉树表示内存分配情况。叶结点表示存储器中的当前分区，如果两个伙伴是叶子，则至少有一个被分配。
- 右图表示A (200)、B (120)、C (240)、D (100) 分配之后的情况。







## 6、伙伴系统的不足

---

- 分配和回收时需要对伙伴进行分拆及合并。
- 存储空间有浪费。
- 伙伴系统适合于小数据量的内存管理，对于大数据量的内存管理，系统采用段页式的分配管理



---

## 第6章 主存储器管理

# 6.5 覆盖与交换技术



## 6.5 覆盖与交换技术

---

- 覆盖与交换技术是在多道程序环境下用来扩充内存的方法。

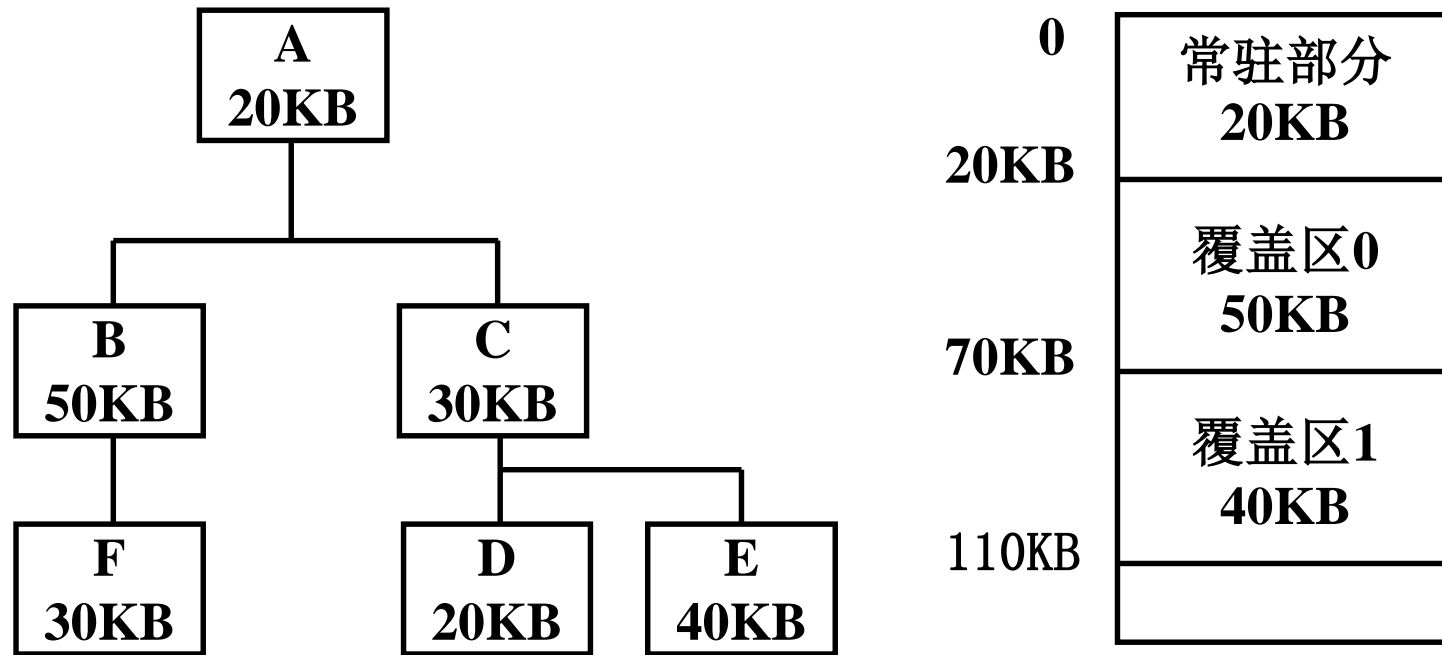


## 6.5.1 覆盖技术

- 所谓覆盖技术就是把一个大程序划分为一系列覆盖，每个覆盖是一个相对独立的程序单位；
- 把程序执行时不要求同时装入内存的覆盖组成一组，称为覆盖段；
- 将一个覆盖段分配到同一个存储区中，这个存储区称为覆盖区。
- 覆盖区的大小由覆盖段中最大的覆盖来确定。

# 覆盖示例

- B、C为一个覆盖段，D、E、F为另一个覆盖段。



- 另一种覆盖方法只需100K：B、D、E为一个覆盖段；F、C为另一个覆盖段



## 6.5.2 交换技术

---

- 在多道程序环境下
  - 内存中：存在一些阻塞进程占据大量的存储空间；
  - 外存上：有许多作业因无空闲内存而不能进入内存运行；
  - 为此引入了交换。
- 交换是指
  - 将内存中暂时不用的程序及数据换出到外存中，以腾出足够的内存空间，再将已具备运行条件的进程或进程所需的程序或数据从外存换入内存中



# 1、交换空间的管理

---

- 交换空间设置在外存交换区中，交换空间管理的主要目标是提高进程换入/换出速度。
- 交换空间采用连续分配方式，使用与动态分区分配类似的数据结构和分配回收算法。



## 2、进程的换出与换入

- 进程的换出：先选择换出进程（阻塞、优先级低、驻留时间长），再申请对换空间，然后启动磁盘写，若成功则可释放其内存空间并修改数据结构。
- 进程换入：先选择换入进程（就绪、换出时间长），再申请内存空间，然后启动磁盘读。





### 3、移动操作系统的交换问题

- 交换技术并没有得到充分支持
  - 原因：Flash存储问题
    - 存储空间有限
    - 写寿命是有限的
    - 移动平台上，Flash和内存之间的吞吐量差
- 替代方法
  - iOS 要求apps主动的放弃内存分配
    - 对于Read-only数据直接删除，将来需要再加载
    - 无法释放内存的apps会被iOS终止掉
  - Android：直接终止进程
    - 在终止前，将apps状态写入flash，用于将来重启apps



## 4、覆盖与交换的比较

- 交换技术由**操作系统自动完成**，不需要用户参与，而覆盖技术**需要专业的程序员**给出作业各部分之间的覆盖结构，并清楚系统的存储结构；
- 交换技术主要在**不同作业**之间进行，而覆盖技术主要在**同一个作业**内进行；
- 覆盖技术主要在早期的操作系统中采用，而交换技术在现代操作系统中仍具有较强的生命力。



---

第6章 主存储器管理

## 6.6 分页存储管理



## 6.6 分页存储管理

---

- 分区管理中存在碎片，而紧凑技术开销太大，若能取消作业对存储区的连续性要求，则能较好地解决碎片问题。
- 分页存储管理就是基于这一思想提出的。

## 6.6.1 分页存储管理的实现思想

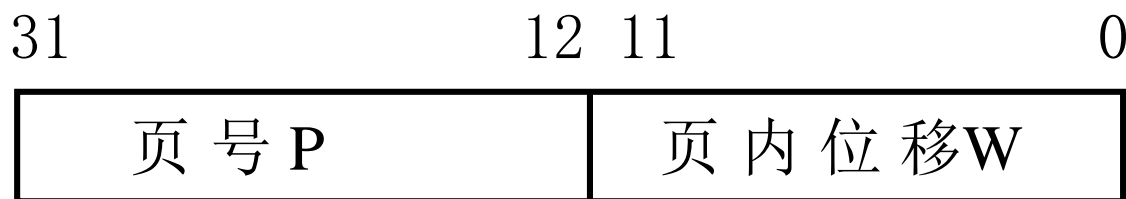
- 在分页存储管理中，将进程的逻辑地址空间划分成若干大小相等的页（或称页面）
- 相应地将主存空间也划分成与页大小相等的块（或称物理块、页框）
- 在为进程分配存储空间时，总是以块为单位来分配，可以将进程中的某一页存放到主存的某一空闲块中。

分页系统中是否有碎片？

页内碎片：由进程最后一页未装满而形成的碎片。

# 分页的逻辑地址结构

- 分页存储管理系统中，逻辑地址由页号和页内位移组成。其结构如下所示：



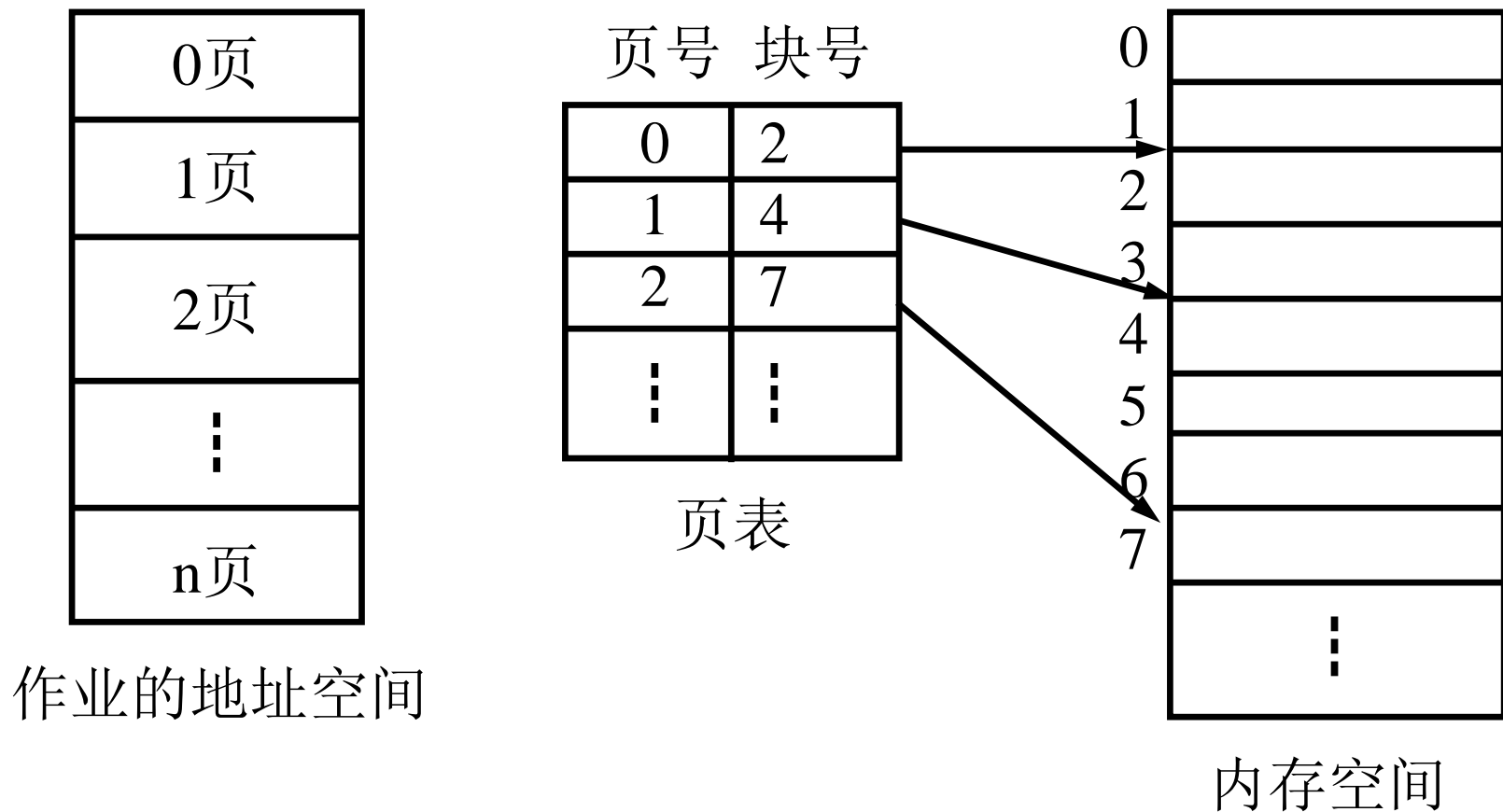
- 若A为逻辑地址，L为页面大小，则：
- 页号：  $P = \text{int}(A/L)$
- 页内位移：  $W = A \% L$



## 6.6.2 页表机制及其相关技术

- 为了在内存中找到进程的每个页面所对应的物理块，系统为每个进程建立一张**页面映象表**，简称**页表**。
- 页表：记录页面在内存中对应物理块的数据结构，页表的每一行称为页表项。
- 注意：Page frame英文词在目前国内操作系统教材中翻译的有物理块（块）、页框、页架、页帧和帧等，应加以注意，它表示的是与一个逻辑页相对应的一个物理块。

# 1、页表的作用







## 2、页面大小的选择

- 页面的大小应适中
  - 若页面太大，以至和一般进程大小相差无几，则页面分配退化为：分区分配，同时页内碎片也较大。
  - 若页面太小，虽然可减少页内碎片，但会导致页表增长。
  - 页面大小通常为2的幂，一般在512B到64KB之间。
- 页表一般存放在内存中，也可以在页表中设置存取控制字段，以实现存储保护。



### 3、存储分块表

---

- 存储分块表用来记录内存中各物理块的使用情况及未分配物理块总数。
- 存储分块表可用下述方式表示：
  - 位示图：利用二进制的一位表示一个物理块的状态，1表示已分配，0表示未分配。所有物理块状态位的集合构成位示图。
  - 空闲存储块链：将所有的空闲存储块用链表链接起来，利用空闲物理块中的单元存放指向下一个物理块的指针。



# 位示图例

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0	0	1	1	0	1	1	1	0	1	1	1	1	1
1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	1
2	1	1	1	1	1	1	0	1	1	1	1	0	0	0	0	0
3																
4	...															
⋮																

- 位示图占用的存储空间：物理块数/8（字节）



## 4、存储空间的分配及回收

- 页面分配：

- 计算进程所需页面数，然后在请求表中登记进程号、请求页面数等。如存储分块表中有足够的空闲块可供进程使用，则在系统中取得页表始址，并在页表中登记页号及其对应的物理块号，否则无法分配。

- 页面回收：

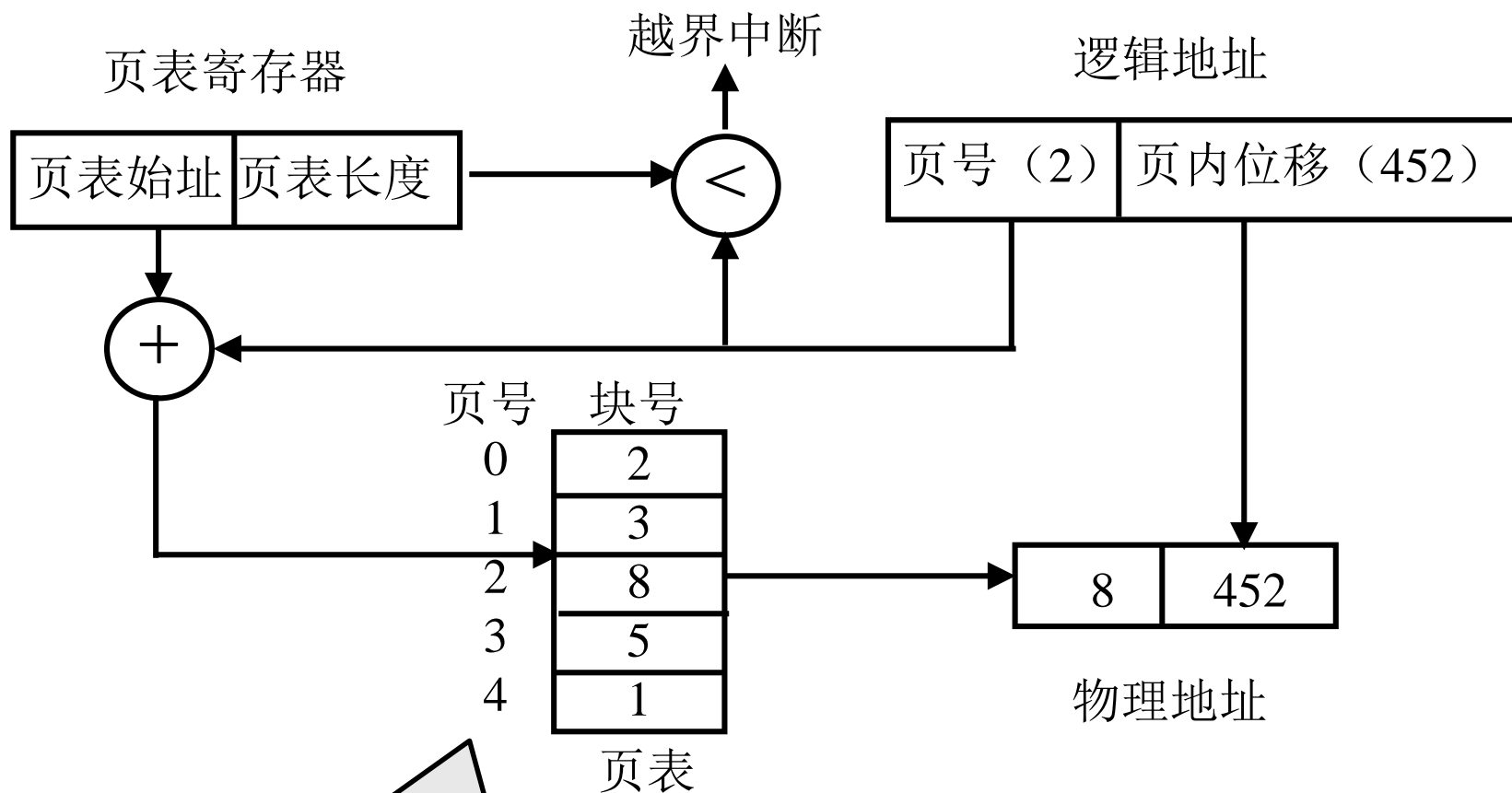
- 将存储分块表中相应的物理块改为未分配，或将回收块加入到空闲存储块链中，并释放页表，修改请求表中的页表始址及状态。



## 6.6.3 分页的基本地址变换机构

- 地址变换机构的任务是实现逻辑地址到物理地址的变换，即将逻辑地址中的页号转换为内存中的物理块号。
- 页表通常存放在内存中，为了实现方便，系统中设置了一个页表寄存器存放页表在内存的起始地址和页表的长度。
- 进程未执行时，页表的起始地址和长度存放在PCB中。当进程执行时，才将页表始址和长度存入页表寄存器中。

# 分页系统的地址变换机构图



注意这里的页号字段？



# 地址变换过程

- 分页地址变换机构自动地将逻辑地址分为页号和页内位移；
- 将页号与页表长度进行比较，如果页号超过了页表长度，则表示本次所访问的地址已超越进程的地址空间，系统产生地址越界中断；
- 若未出现越界，则由页表始址和页号计算出相应页表项的位置，从中得到该页的物理块号；
- 将物理块号与逻辑地址中的页内位移拼接在一起，就形成了访问主存的物理地址。



# 地址变换例1

- 设页面大小为1K字节，作业的0、1、2页分别存放在第2、3、8块中。则逻辑地址2500的页号及页内地址为：
  - $2500/1024=2$ （页号）；
  - $2500 \% 1024 = 452$ （页内地址）；
  - 查页表可知第2页对应的物理块号为8；
  - 将块号8与页内地址452拼接得到物理地址为： $8 \times 1024 + 452 = 8644$ 。



## 地址变换例2

- 一分页系统中逻辑地址长度为16位，页面大小为1KB，且第0、1、2、3页依次存放在物理块3、7、11、10中。现有一逻辑地址0A6FH，其二进制表示如下：

页号    页内地址

000010 1001101111

- 由此可知逻辑地址0A6FH的页号为2，该页存放在第11号物理块中，用十六进制表示块号为B，所以物理地址为：
- 1011 1001101111，即2E6FH。

## 6.6.4 具有快表的地址变换机构

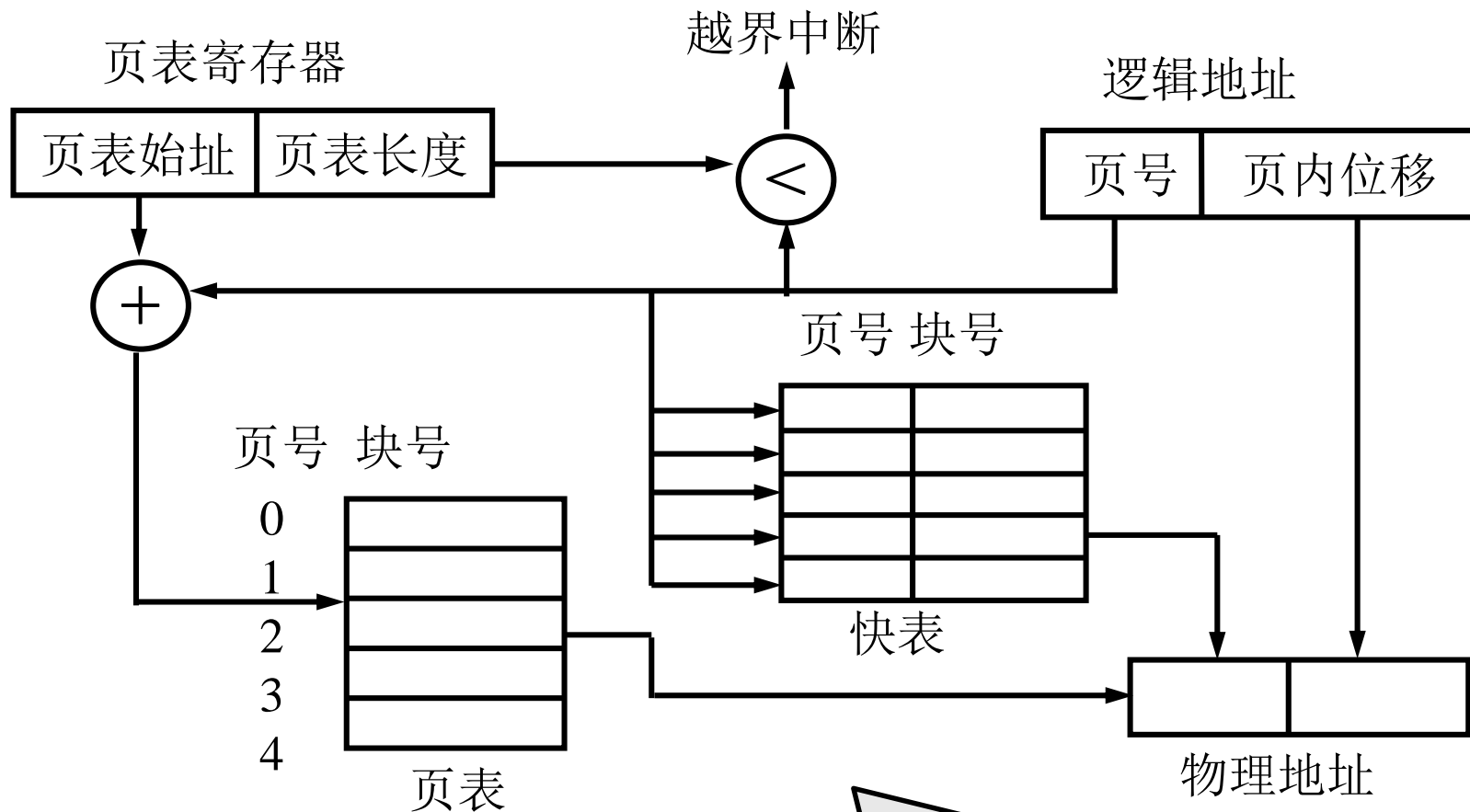
- 因页表放在主存中，故存取数据时CPU至少要访问两次主存，降低了内存访问速度。
- 为了提高地址变换速度，可在地址变换机构中增设硬件支持：
  - 具有并行查找能力的高速缓冲存储器(又称联想存储器或快表)，用以存放当前访问的那些页表项。
  - TLB (translation look-aside buffer)：转换后备缓冲区分区，即快表。



# 引入快表后的地址变换过程

- 地址变换机构自动将页号与快表中的所有页号进行并行比较，若其中有与此匹配的页号，则取出该页对应的块号，与页内地址拼接形成物理地址。
- **若页号不在快表中**，则地址变换机构会到主存页表中取出物理块号，与页内地址拼接形成物理地址。
- 同时还应将这次所查到的页表项存入快表中，若快表已满，则必须按某种原则淘汰出一个表项以腾出位置。

# 具有联想存储器的地址变换



此处页表与快表有何不同？



# 联想存储器的大小

- 由于成本关系，快表大小一般由64—1024个表项组成。由于局部性原理，联想存储器的命中率可达80%--90%。
- 局部性原理
  - CPU访问存储器时，无论是存取指令还是存取数据，所访问的存储单元都趋于聚集在一个较小的连续区域中。
  - 时间局部性(Temporal Locality)
  - 空间局部性(Spatial Locality)
  - 顺序局部性(Order Locality): 5: 1



# 有效内存访问时间

- 设快表命中率为 $p$ ，内存访问时间为 $m$ ，快表访问时间为 $n$ ，假定忽略快表更新时间
- 则内存有效访问时间=
$$p \cdot (n + m) + (1 - p)(n + 2m)$$
- 若 $p = 0.8$ ， $m = 100\text{ns}$ ， $n = 20\text{ns}$
- 则内存有效访问时间=
$$0.8 \cdot 120 + 0.2 \cdot 220$$
$$= 140\text{ns}$$



## 6.6.5 多级页表及反向页表

- 现代计算机系统都支持非常大的逻辑地址空间，致使页表很大，用连续空间存放页表显然不现实。
- 如Intel x86 CPU逻辑地址32位，页面大小4KB，当采用一级页表时，可容纳的页表项为1M个，若每个页表项占4字节，则页表共需要4MB内存空间，一张物理页能容纳的页表项为1K个，**但这仅为一个进程所需。**
- 解决方案：
  - 将页表页分级存储，用离散方式存储页表
  - 仅将当前需要的部分页表项放在内存，其余放在磁盘上，需要时产生“缺页表页”中断来调入。

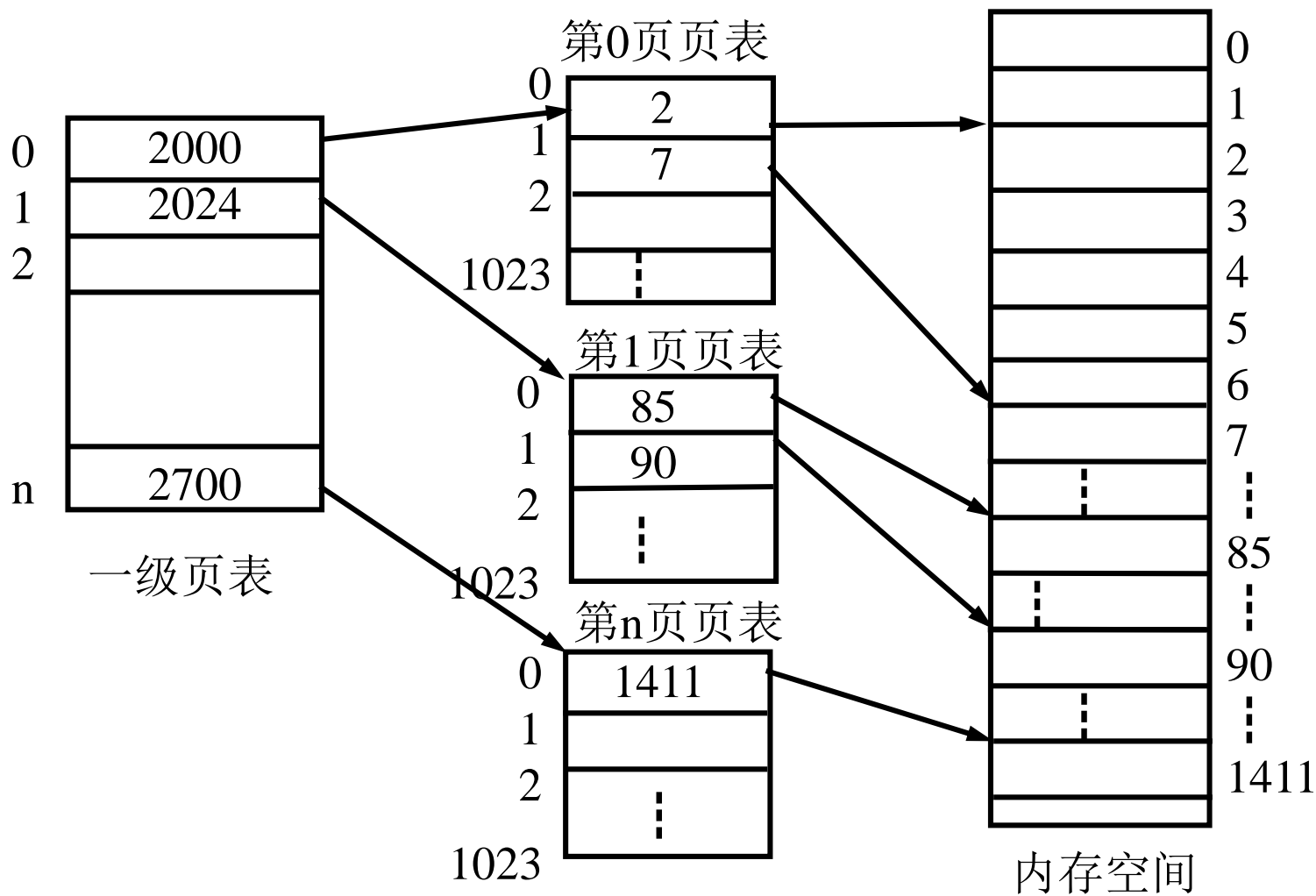
# 1、两级页表及多级页表

- 将页表再分页，使每页与内存物理块大小相同，并为它们进行编号0、1、...，同时还为离散存放的页表建立一张页表。
- 例如：32位地址可以划分为



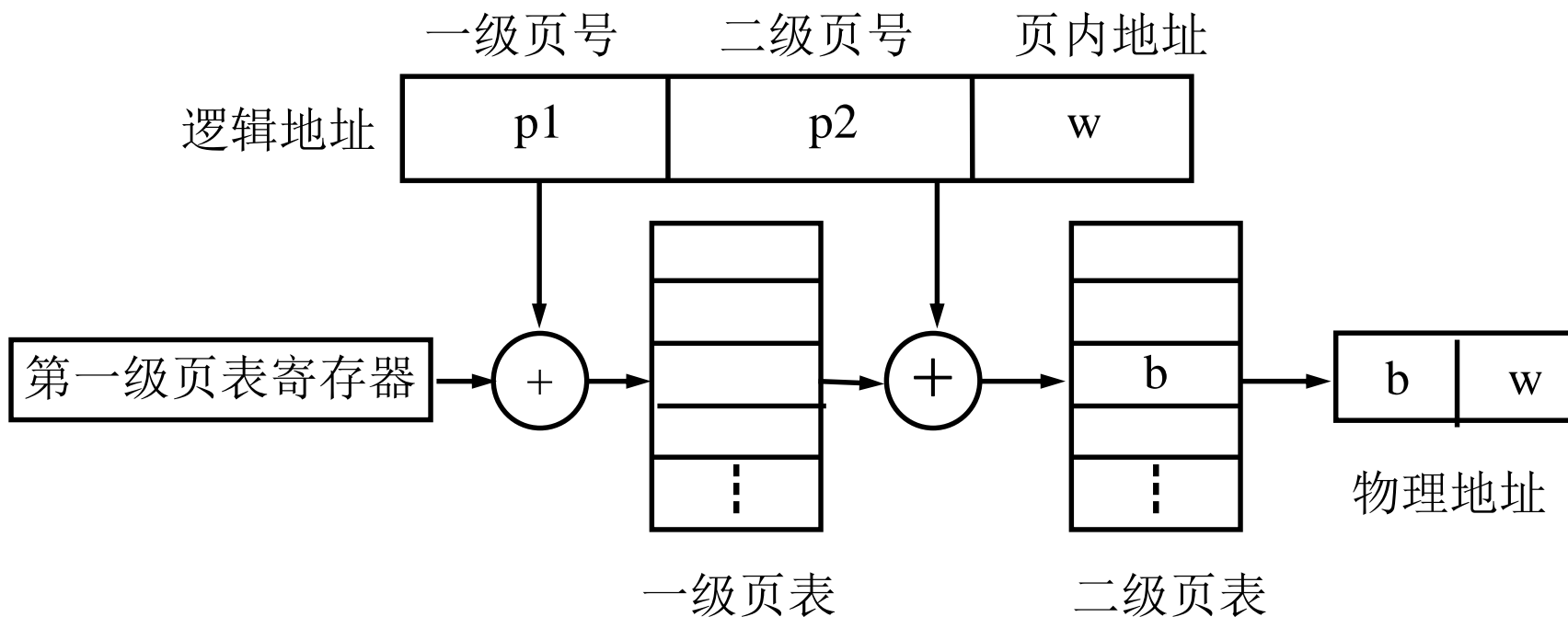


# 两级页表结构



# 具有两级页表的地址变换过程

- 利用逻辑地址中的一级页号作为索引访问一级页表，找到第二级页表的起始地址
- 再利用第二级页号找到指定页表项，从中取出块号与页内地址拼接形成物理地址。





# 多级页表

---

- 对两级页表进行扩充，便可得到三级、四级或更多级的页表。
- 多级页表的实现方式与两级页表类似。
  - Sun SPARC系统采用了3级页表，为了避免进程切换时页表页的重新装入，硬件还支持4096个上下文，每个上下文对应一个进程
  - Motorola 68030提供页表级数和每级位数的编程控制

# 多级页表例

- 为满足 $2^{64}$ 地址空间的作业运行，采用多级分页存储管理方式，假设页面大小为4KB，页表中每个页表项需占8字节，则为了满足系统的分页管理至少应采用多少级页表？
- 解：页面大小=4KB= $2^{12}$ B，每个页表项为8字节= $2^3$ B，
- 所以一个页面中可以存放 $2^{12}/2^3=2^9$ 个页表项。设有n层分页，则64位逻辑地址形式为：

第1层页号	第2层页号	...	第n层页号	页内偏移量
-------	-------	-----	-------	-------

- 其中，页面大小为 $2^{12}$ 字节，所以页内偏移量占12位。
- 由于最高层页表占一页，每页可以存放下 $2^9$ 个表项，因此分页层数： $52/9=6$ 。
- 所以为了满足系统的分页管理至少应采用6级页表。



## 2、反向页表

---

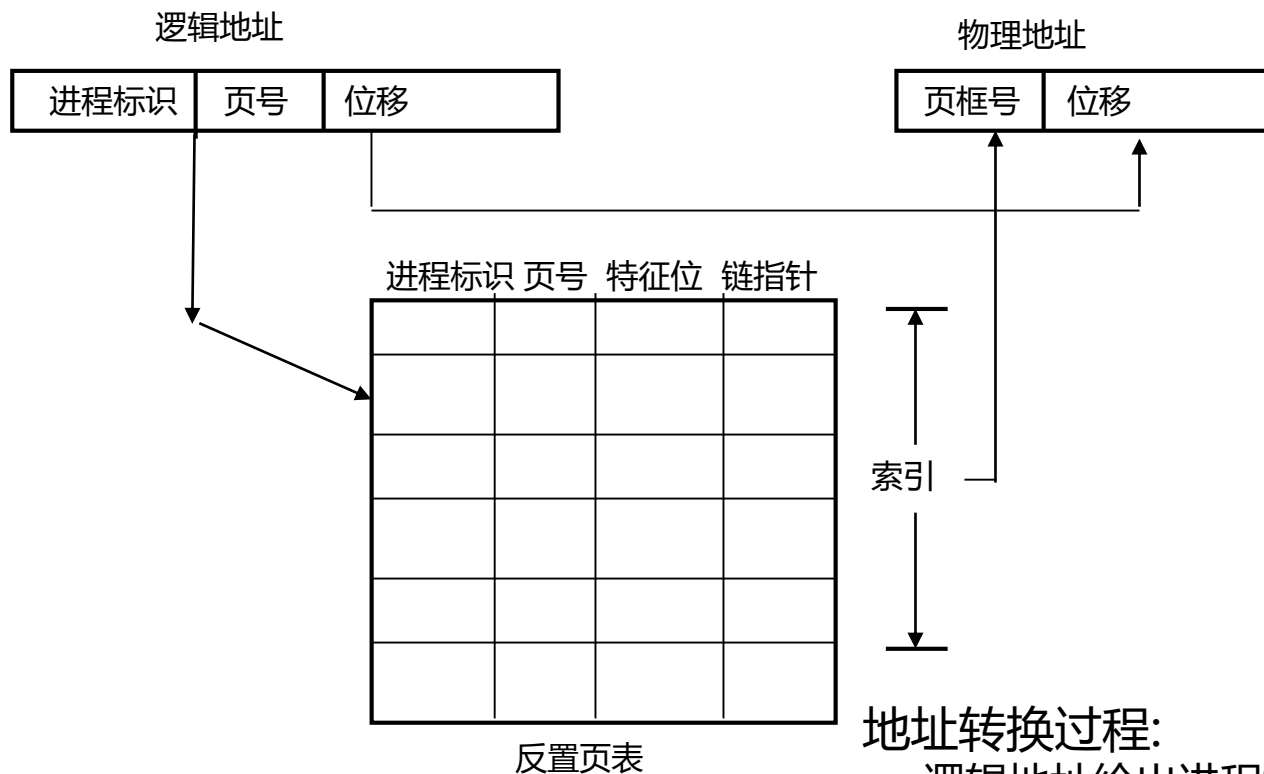
- 进程是通过页的虚拟地址来访问页的，地址转换依赖OS
- 页表排序是按照虚拟地址进行的，因此需存储页表项，跟踪物理内存的使用
- 现代操作系统一般允许大逻辑地址空间，如 $2^{64}$ ，这使得页表太大，为解决页表占用大量存储空间的问题，在PowerPC、UltraSparc等结构中引入了反向页表(Invert Page Table, IPT)，为所有进程维护一张表。
- 整个系统只有一个表，每个物理内存的页纸有一个条目对应。



# 反向页表的基本想法

- IPT是为内存中的每一个物理块建立一个页表项并按照块号排序
- 该表每个表项包含正在访问该页框的进程标识、页号及特征位
- 用途：完成主存页框到访问进程的页号、即物理地址到逻辑地址的转换。

# 反向页表的地址变换



反置页表及其地址转换

## 地址转换过程:

- 逻辑地址给出进程标识和页号,用它们去比较IPT
- 若整个反置页表中未能找到匹配的页表项,说明该页不在主存,产生请页中断,请求操作系统调入;
- 否则, 该表项的序号便是页框号,块号加上位移,便形成物理地址。

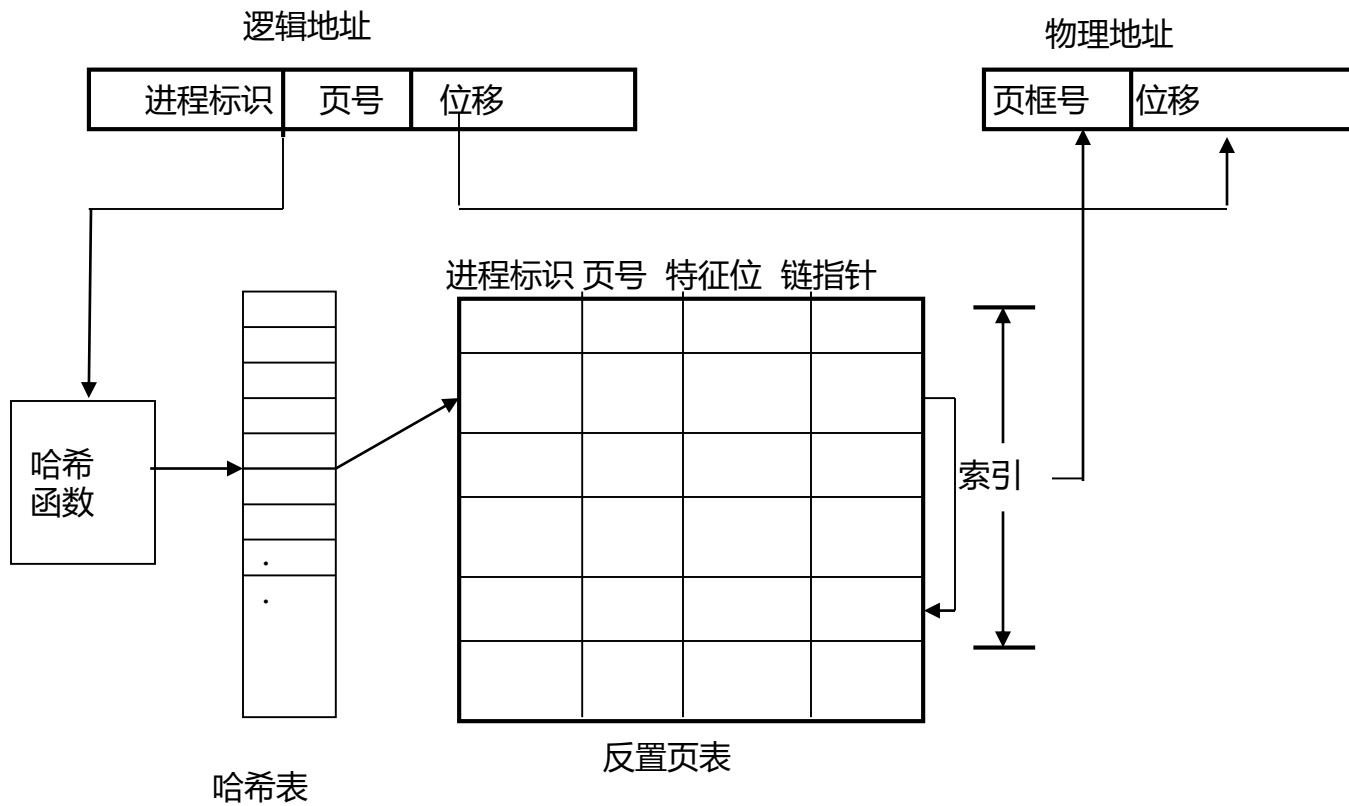


# 反向页表的不足

- 没有调入的页面不在IPT中，任然需要保存传统页表
  - 解决办法：存放到磁盘中
- 内存共享困难
  - 通常概念：多个虚拟地址映射到一个物理地址
  - IPT，每个物理地址只对应一个虚拟页条目
  - 解决办法：对于共享物理地址，只允许存在一个虚拟地址映射，其他的没有映射的虚拟地址访问会产生Page Fault
- 反向页表查找慢：
  - IPT是按照物理地址排序的，访问时是通过虚拟地址进行的，查找时需要遍历整个反向页表
  - 解决办法：
    - 采用哈希页表技术，但引入了新的内存访问代价，且可能存在哈希冲突
    - 引入快表机制



# 引入哈希页表的IPT



反置页表及其地址转换



## 6.6.7 分页存储管理的保护与共享

- 分页存储管理采用两种方式保护内存：
  - 地址越界保护：页表长度与逻辑地址中的页号比较
  - 存取控制保护：在页表中增加保护位
- 分页存储管理的共享方式
  - 数据共享：允许不同进程对共享的数据页用不同的页号来访问，即只需要页表指向共享的数据页框即可。
  - 代码共享：由于共享代码页面内包含地址，不同逻辑空间若页号不同，则导致无法访问，因此必须赋予相同页号。



---

第6章 主存储器管理

## 6.7 分段存储管理



## 6.7 分段存储管理

- 由于分页按物理单位进行，没有考虑程序段的逻辑完整性，给程序段的共享和保护带来不便，另外动态链接及段的动态增长也要求以逻辑上完整的程序段为单位管理。
- 从用户视角而言，用户并不关心程序的元素究竟在内存何处，他们只关心这是一堆数据，那是另外一组数据而已。

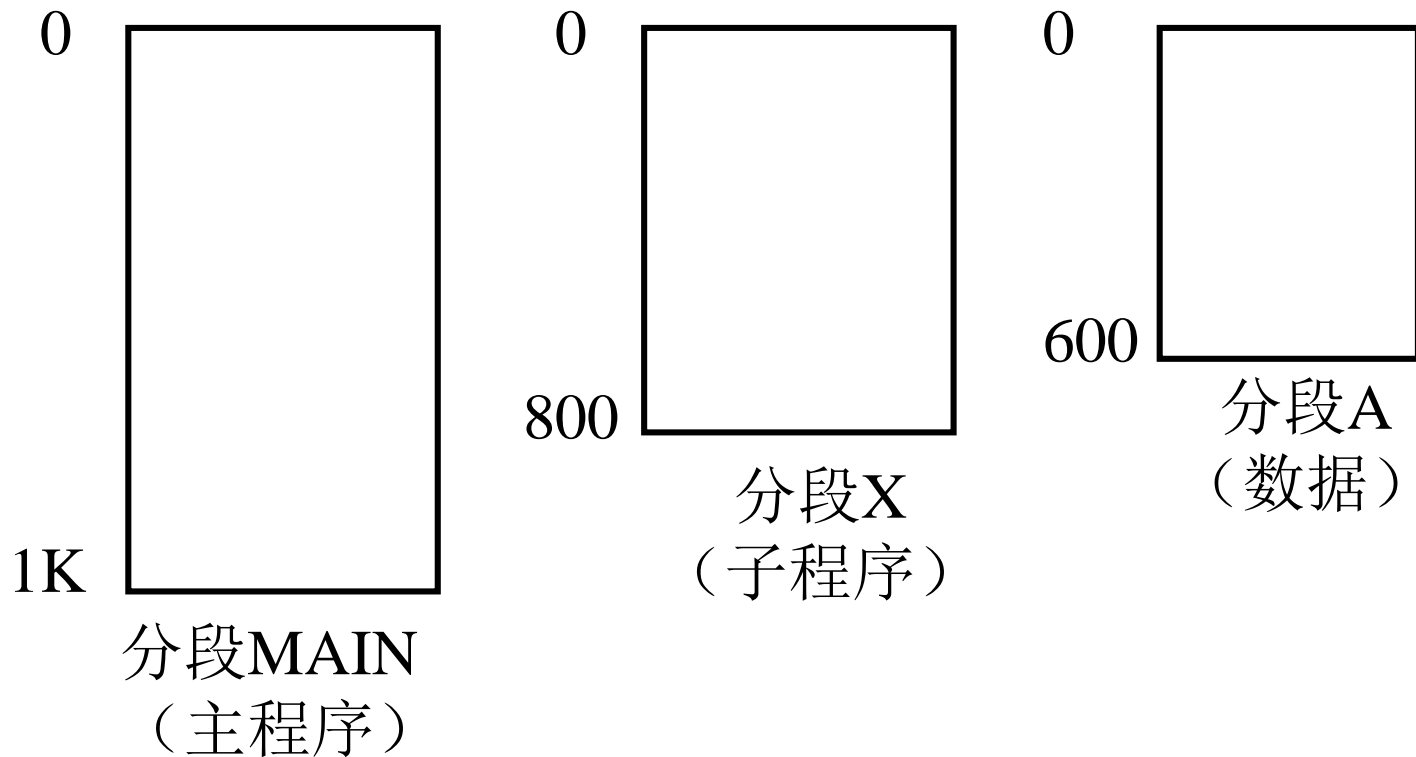


## 6.7.1 分段管理的实现思想

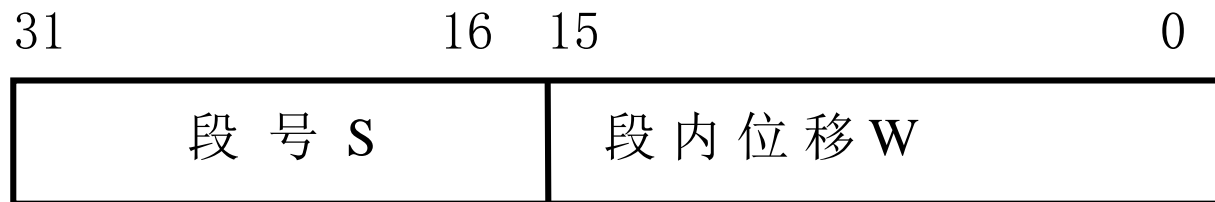
- 在分段存储管理系统中，作业的地址空间由若干个逻辑分段组成，每个分段是一组逻辑意义相对完整的信息集合，每个分段都有自己的名字，每个分段都从0开始编址并采用一段连续的地址空间。
- 在进行存储分配时，以段为单位分配内存，每段分配一个连续的内存区，但各段之间不要求连续。

# 作业的地址空间是二维的

- 作业的地址空间分为多段，每段都从0开始编址，故地址是二维的。



# 分段系统的逻辑地址结构



该地址结构最多允许多少分段?每段最大长度为多少?

该地址结构允许作业最多有64K个段，  
每段的最大长度为64KB。

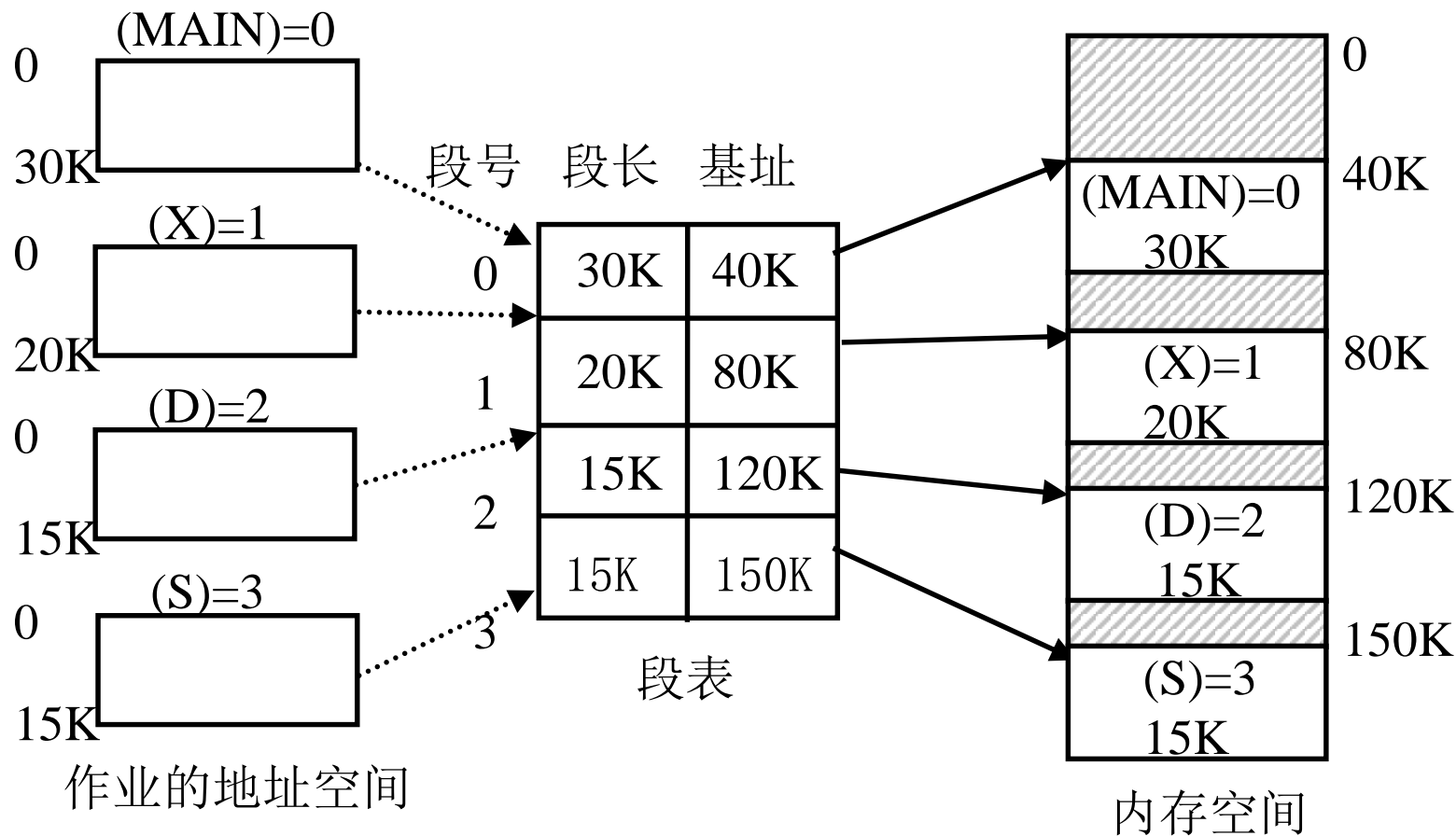


## 6.7.2 段表及地址变换

- 为了实现从逻辑地址到物理地址的变换，必须为每个进程建立一个段表，用来记录每段在内存的起始地址及相关信息。其中每个表项描述一个分段的信息，至少包含：
  - 段号
  - 段长
  - 段在内存的起始地址
  - 其他信息
- 段表一般存放在内存。



# 段表的作用



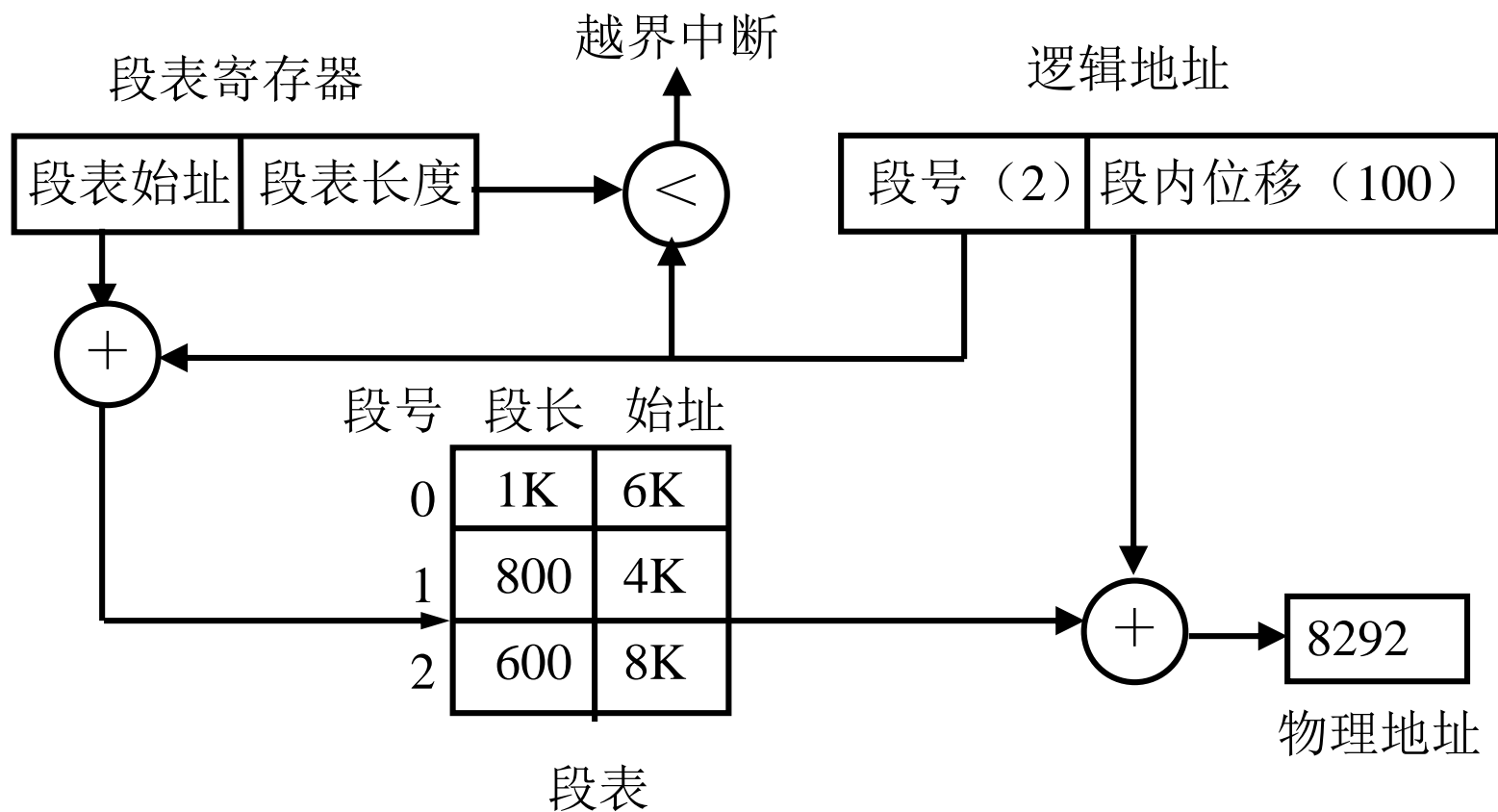


# 地址变换

---

- 为实现从逻辑地址到物理地址的转换，在系统中设置了段表寄存器，用于存放段表始址和段表长度。
- 为了提高内存的访问速度，也可以使用快表。

# 地址变换机构图





# 地址变换过程

- 进行地址变换时，系统将逻辑地址中的段号S与段表长度进行比较，若段号超过了段表长度则产生越界中断；
- 否则根据段表始址和段号计算出该段对应段表项的位置，从中读出该段在内存的起始地址，
- 然后再检查段内地址是否超过该段的段长，若超过则同样发出越界中断信号；
- 若未越界，则将该段的起始地址与段内位移相加，从而得到了要访问的物理地址。



# 分段地址变换例

- 设作业分为3段，0、1、2段长度分别为1K、800、600，分别存放在内存6K、4K、8K开始的内存区域。
  - 逻辑地址 (2, 100) 的段号为2，段内位移为100。
  - 查段表可知第2段在内存的起始地址8K。
  - 将起始地址与段内位移相加， $8K + 100 = 8292$ ，物理地址为8292。



## 6.7.3 分段与分页的主要区别

- 分页管理与分段管理有许多相似之处，但两者在概念上也有很多区别，主要表现在：
  - ① 页是信息的**物理单位**，是为了减少内存碎片及提高内存利用率，是系统管理的需要。段是信息的**逻辑单位**，它含有一组意义相对完整的信息，分段的目的是为了更好地了解用户的需要。
  - ② 页的大小**固定且由系统决定**，由硬件把逻辑地址划分为页号和页内地址两部分。段的长度**不固定且由用户所编写的程序决定**，通常由编译系统在对源程序进行编译时根据信息的性质来划分。



# 分段保护

---

- 分段保护方法有：
  - 地址越界保护：段号与段表长度的比较，段内位移与段长的比较
  - 存取控制保护：设置存取权限，访问段时判断访问类型与存取权限是否相符



---

第6章 主存储器管理

## 6.8 段页式存储管理



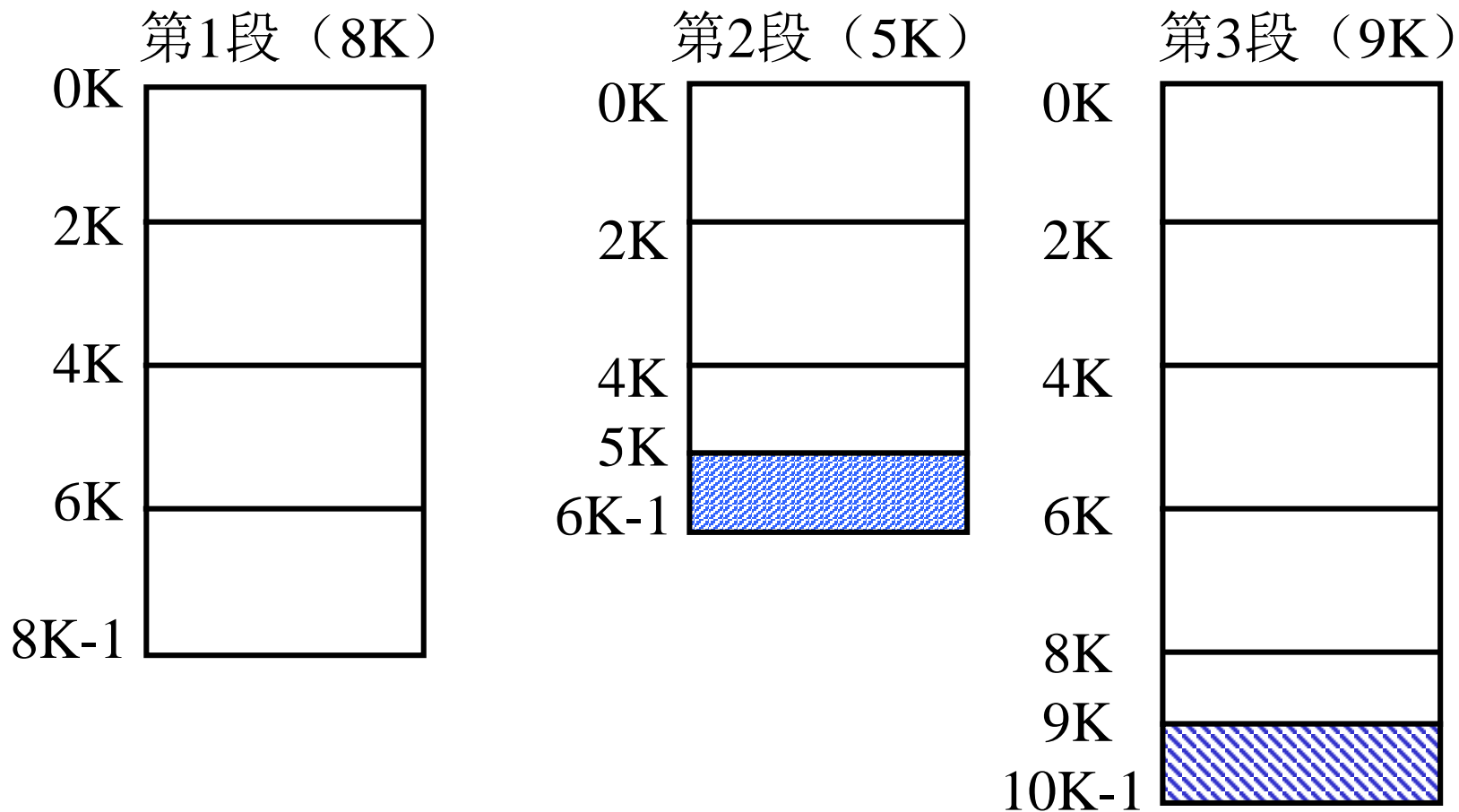


## 6.8 段页式存储管理

- 分页系统能有效地提高内存利用率，而分段系统能很好地反映用户要求。
- 如果将这两种存储管理方式结合起来，就形成了**段页式存储管理系统**。
- 基本思想：
  - 在段页式存储管理系统中，作业的地址空间**首先**被分成若干个**逻辑分段**，**然后**再将每一段分成若干个大小**固定的页面**。
  - 将主存空间分成若干个和页面大小相同的物理块，对主存的分配以物理块为单位。

# 作业的分段及分页示意图

- 设作业包含分为三段，页面大小为2K。



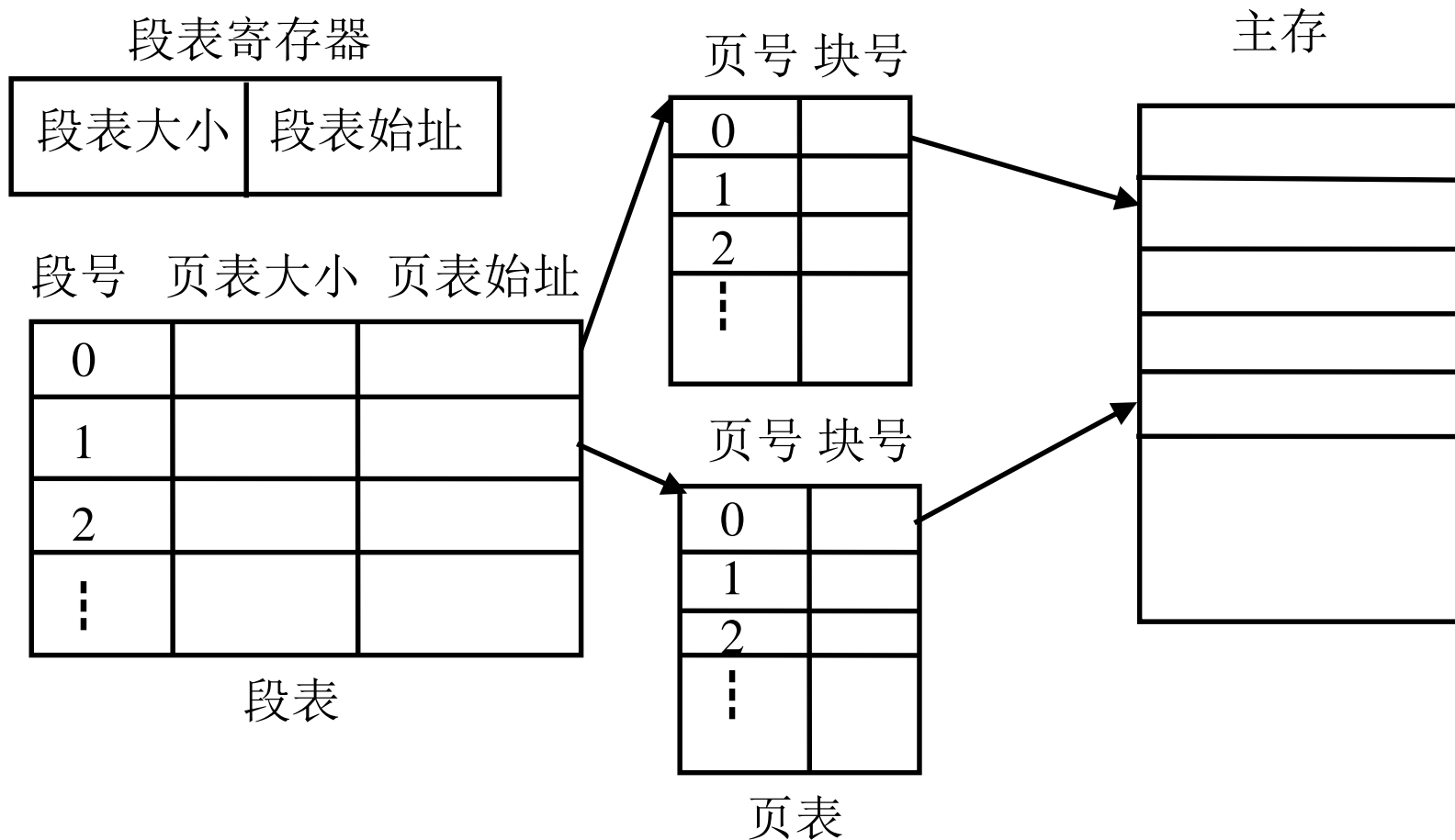
# 作业的逻辑地址结构

- 作业的逻辑地址结构:

段号S	段内页号P	页内位移D
-----	-------	-------

- 为了实现地址变换，系统中需要设立段表及页表。
- 此外，为了便于实现地址变换，还需配置一个段表寄存器，其中存放作业的段表起始地址和段表长度。

# 段表、页表及段表寄存器

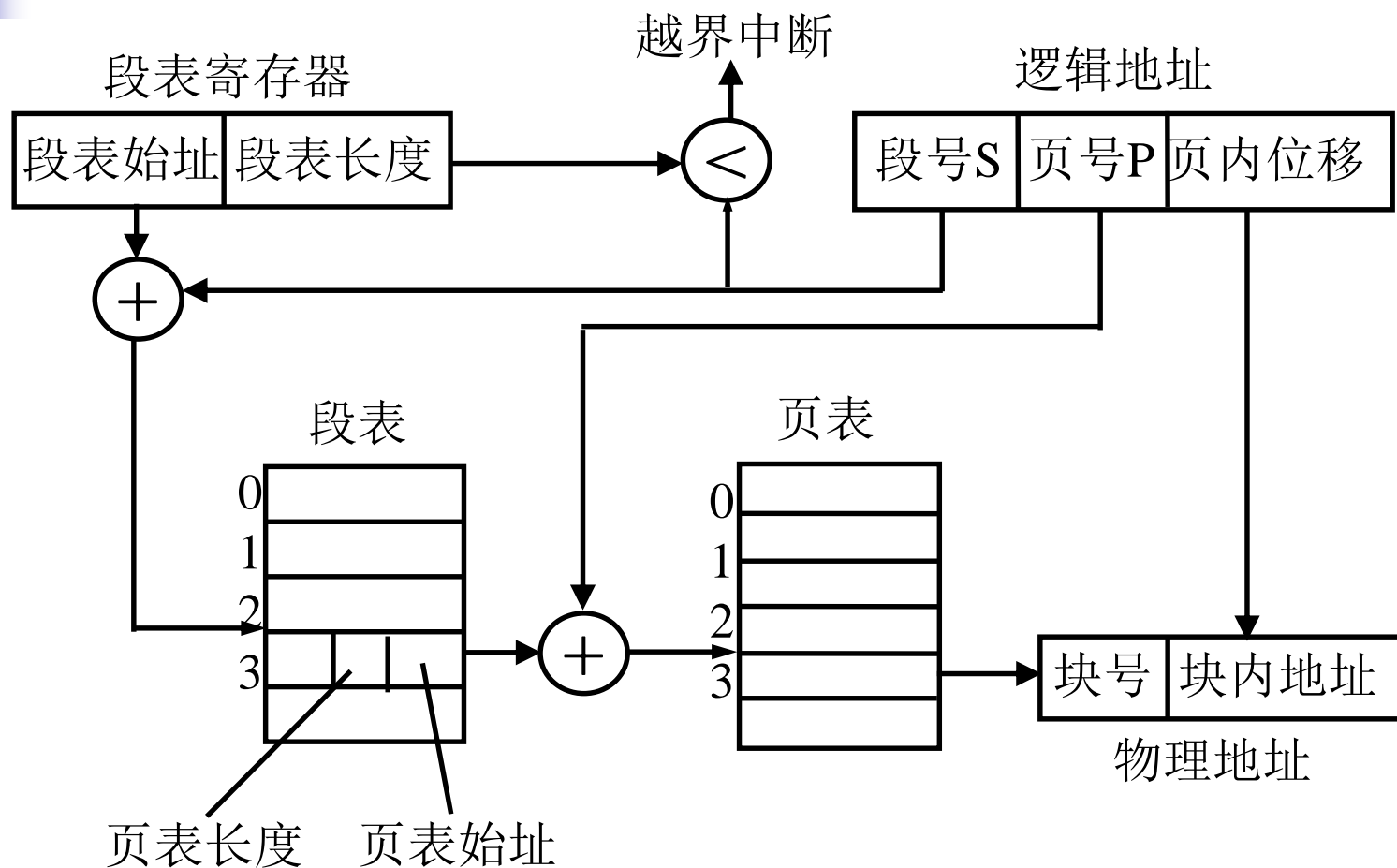




# 地址变换过程

- 在进行地址变换时，首先将段号 $S$ 与段表寄存器中的段表长度进行比较，若小于段表长度则表示未越界
- 利用段表始址和段号求出该段对应段表项的位置，从中得到该段的页表始址，
- 再利用逻辑地址中的段内页号 $P$ 获得对应页表项的位置，从中读出该页所在的物理块号，再与页内地址拼接成物理地址。

# 段页式系统中的地址变换机构





# 使用快表提高内存访问速度

- 在段页式系统中，要想存取访问信息，需要三次访问内存：
  - 第一次访问段表
  - 第二次访问页表
  - 第三次访问信息
- 为了提高访问主存的速度，应考虑使用联想寄存器。



---

第6章 主存储器管理

## 6.9 段页式存储管理举例





# 例1: Intel 内存管理

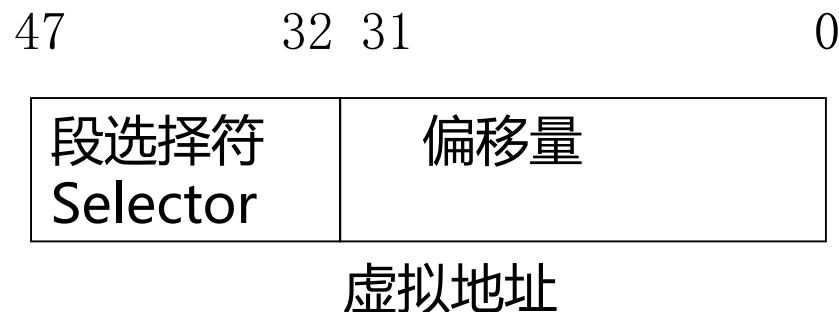
- 8086: 16-bits 处理器
  - 可寻址空间: 64KB
- 地址总线 20-bits
  - 最大寻址空间: 1MB
  - 16位如何与20位匹配
    - 引入段寄存器, 分两次读进地址: 段基址+偏移
    - 实模式寻址:  $\text{physical addr} = 16 * \text{segment} + \text{offset}$
    - CS: code segment, for EIP
    - SS: stack segment, for SP and BP
    - DS: data segment for load/store via other registers
    - ES: another data segment, destination for string ops
    - *e.g.*  $CS=f000 \quad IP=ffff0 \Rightarrow ADDR: fffff0$

# Intel IA-32 & x86-64

- 32 bit CPUs: IA-32
  - From 80386 to Pentium 家族
- 64 bit Intel CPUs: IA-64 /x86-64
  - From 酷睿家族, 安腾、至强

- IA32的段寄存器

- CS: 保存代码段选择符
- DS: 保存数据段选择符
- SS: 保存堆栈段选择符
- CR0: 保存分页标志PG、保护允许标志PE等
- CR2: 保存缺页中断时, 所缺页的线性地址
- CR3: 保存页目录表基址





# 1. Intel IA-32 Architecture

---

- 支持分段和段页式
  - 每个段最大4GB
  - 每个进程最多可以16K个段
  - 段被分成两部分
    - 进程私有部分信息：描述该进程分段信息
      - 最大8K个段
      - 存储在LDT, local descriptor table
    - 所有进程公共部分信息：描述OS的分段信息
      - 最大8K个段
      - 存储在GDT, global descriptor table
  - LDT、GDT的每个条目为8字节



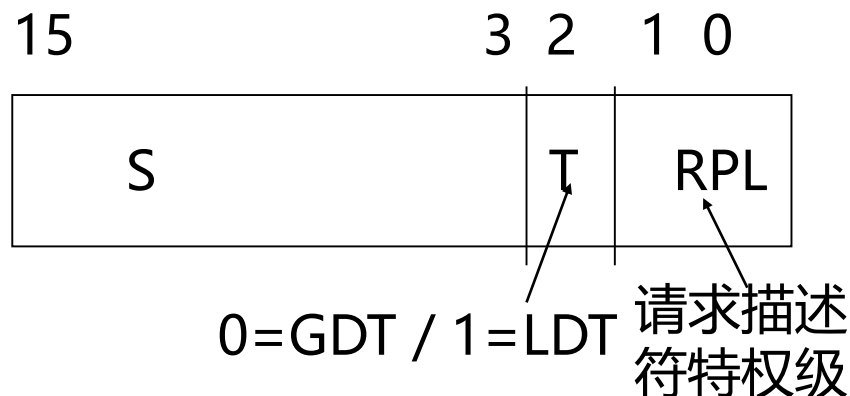
# 虚拟地址空间大小

- 虚拟地址空间共包含16K个存储器分段
  - GDT映射一半(8192): 全局虚拟地址空间
  - LDT映射另一半(8192): 局部虚拟地址空间
- 发生进程切换时, LDT更新为待执行进程的LDT, 而GDT保持不变
- 由于每段偏移量32位、即=4GB, 整个虚存地址空间  
=  $16K \times 4GB = 64TB$ 。
- 物理内存不足4GB, 含设备映射的内存

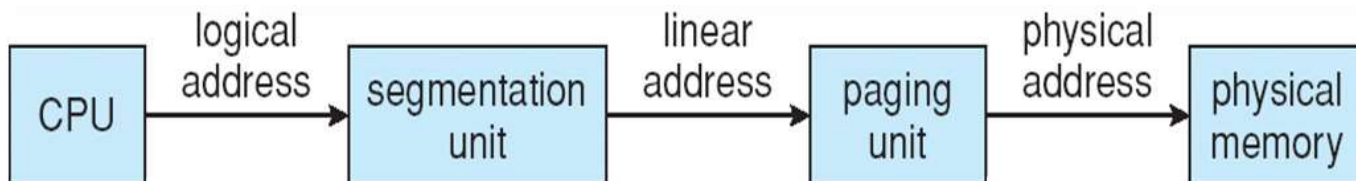
# Intel IA-32 地址映射

- 逻辑地址由<Selector, offset>组成
- Selector由s, g, p组成

- S: 段号
- T: GDT or LDT?
- RPL: 段描述符特权 DPL

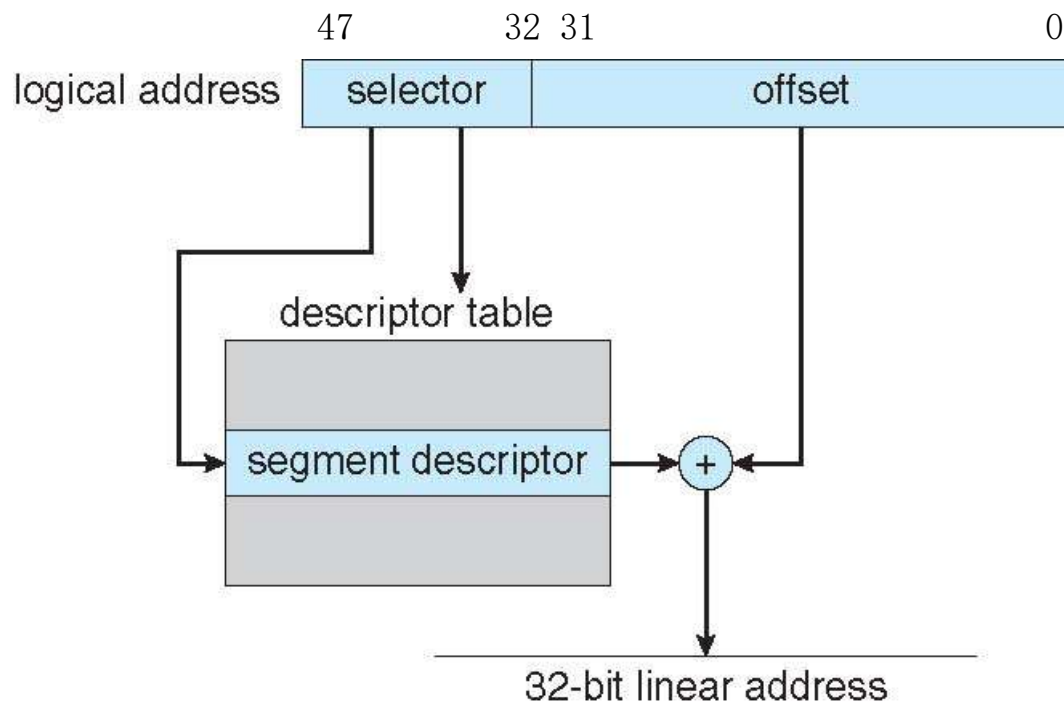


- CPU产生逻辑地址
- 选择子Selector被送到分段单元产生线性地址
- 线性地址被送到分页单元产生物理地址



# Intel IA-32 分段管理

- 逻辑地址→线性地址
  - 选择子Selector被送到分段单元
  - 分段单元产生线性地址



# 描述符Descriptor

- 描述符表中的描述符是存储管理硬件MMU管理虚存空间分段的依据。
- 一个描述符直接对应于虚存空间中的一个主存分段，定义段的基址、大小和属性。8字节

高地址.....低地址

7	6	5	4	3	2	1	0
765432107654321076543210765432107654321076543210							

<- 共 8 字节

31..24	(见下图)	段基址(23..0)	段界限(15..0)
基址2	③   ②   ①	基址1	段界限1

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
G	D	0	AVL	段界限 2 (19..16)				P	DPL		S	TYPE			
③: 属性 2				③: 段界限 2				③: 属性1							

高地址

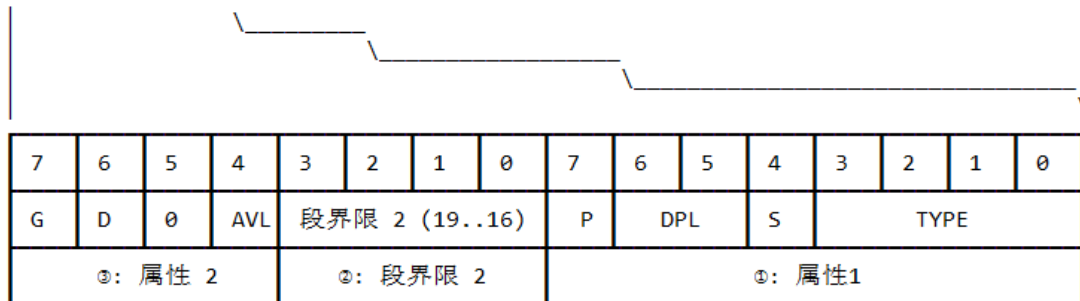
低地址

高地址.....低地址

7	6	5	4	3	2	1	0
7654321076543210765432107654321076543210765432107654321076543210							
-----							

<- 共 8 字节

31..24	(见下图)	段基址(23..0)	段界限(15..0)
基址2	③   ②   ①	基址1	段界限1



高地址

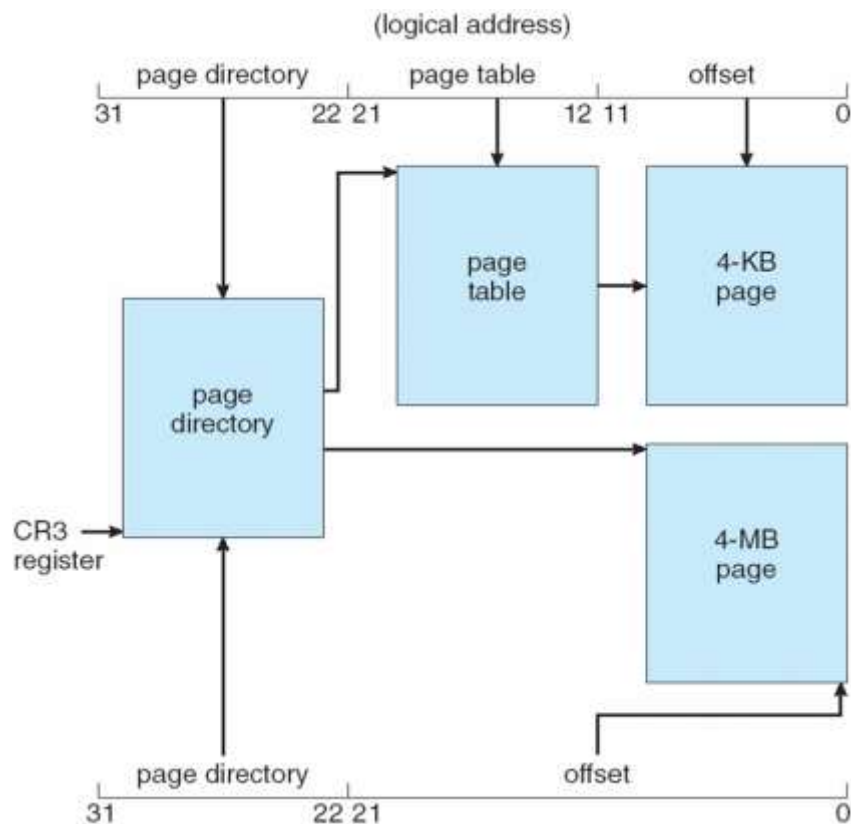
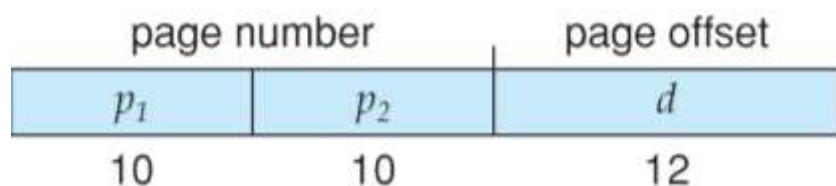
低地址

- 段基址：32位，三部分组成
- 段限长：20位，两部分组成
- P：P=1,该段在内存中；否则P=0，该段不在内存中，会引发异常
- DPL：描述符特权级(Descriptor Privilege level)，用于段访问时的特权检查
- S：段内容标志，S=1，代码和数据段描述符；S=0，系统段描述符/或门描述符
- TYPE：段类型和保护方式，如可执行数据段、只读数据段等
- G：段界限粒度，G=0，以字节为单位，G=1以页面4K为单位，故段长为 $2^{20}B$ 或者 $2^{20} \times 4KB = 4G$
- D：多种条件下，语义不同，D=1 32位方式访问、D=0 16位方式访问，
- AVL：软件可利用位，保留。

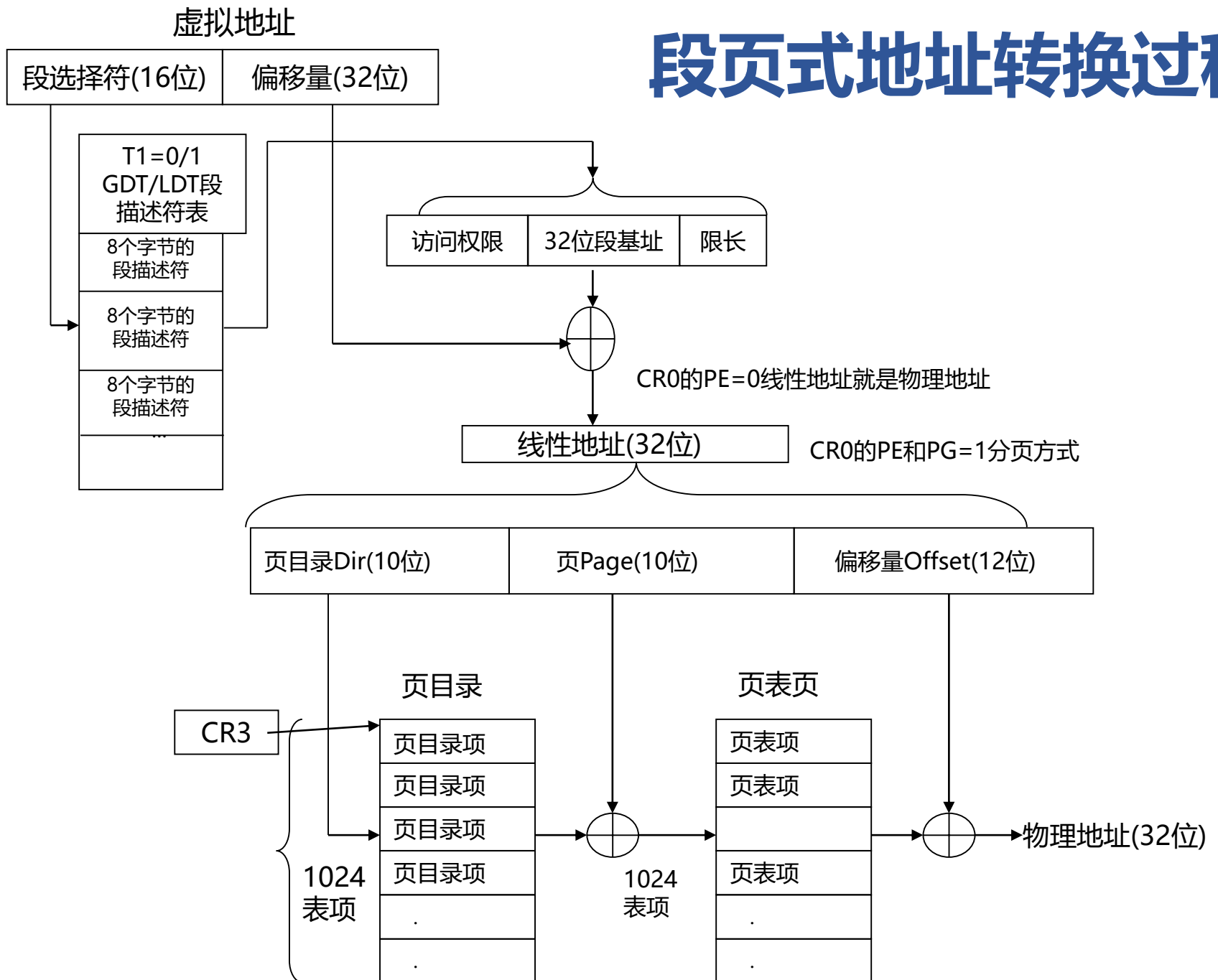


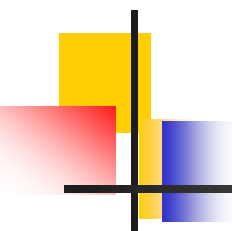
# Intel IA-32 分页管理

- 线性地址被送到分页单元产生物理地址
  - 分页单元产生物理地址
  - 分段和分页单元构成了MMU
  - 物理页大小可以为 4 KB or 4 MB



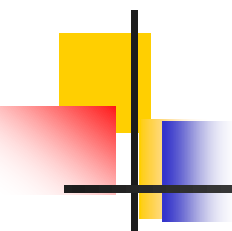
# 段页式地址转换过程





# 虚拟地址→线性地址

- MMU使用分段机制把48位虚拟地址(16位选择符+32位偏移量)转换成32位线性地址，转换过程是通过描述符表中的描述符来实现的。
  - 段选择符装入段选择符寄存器
  - 根据选择符的T位判断选的是LDT或GDT
  - 根据index段描述符下标，由硬件自动从表中取出描述符装入到段描述符高速缓存寄存器，实现16位选择符到32位段基址的转换，
  - 把描述符中的32位段基址与32位偏移量相加便形成32位线性地址。



# 线性地址→物理地址

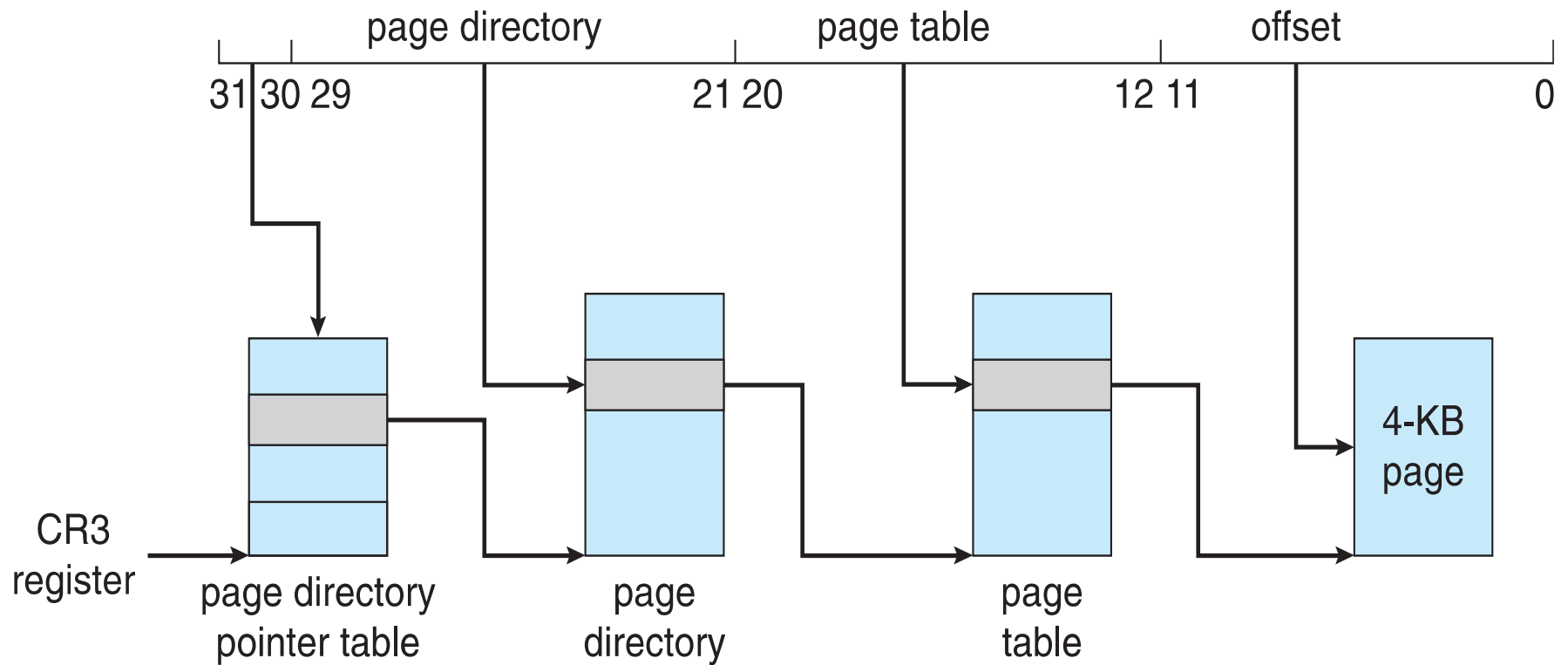
- 启用分页机制时，需要通过分页机制进行第二次地址转换。
  - 由分段得到的线性地址分成三个域：10位页目录dir、10位页page和12位偏移量offset。
  - 根据控制寄存器CR3给出的页目录表起址，用dir作索引在页目录表中找到指向页表的起址
  - 用page作索引在页表中查找到页框起址
  - 把偏移量加到页框起址上，得到访问单元的物理地址。



# Intel IA-32 页地址扩展PAE

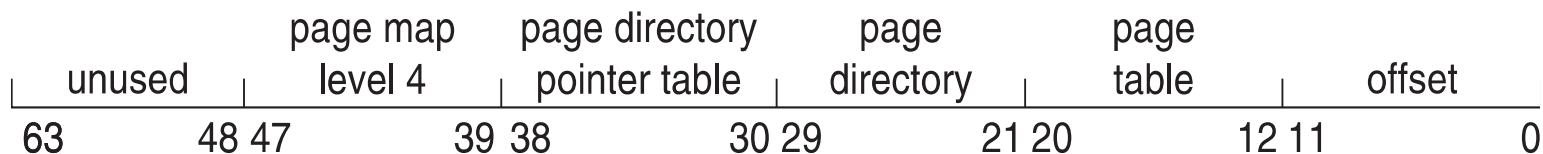
- 32位地址限制了物理内存的访问, 4GB, 因此提出 page address extension (PAE), 允许 32-bit apps 访问大于4GB的物理内存空间
  - 物理手段: Intel处理器增加管脚数(地址线), 从32 → 36, 即 4GB → 64GB
  - 页表基址、页框基址: 20bits → 24bits, 结合12位页内偏移
  - 页目录项、页表项: 12bits标志位+24bits物理地址 → 36bits → 64bits(方便对齐), 即占用8Bytes
  - 每一页包含的页目录项、页表项个数:  $4\text{KB}/8\text{B}=512$
  - 总物理页个数:  $64\text{GB}/4\text{KB}=16\text{M}$ , 总页表个数:  $16\text{M}/512=32\text{K}$ , 总页目录数:  $32\text{K}/512=64$ 个
  - 从二级分页 → 三级分页
  - 高两位指向page directory pointer table

# Intel IA-32 页地址扩展PAE



## 2、Intel x86-64

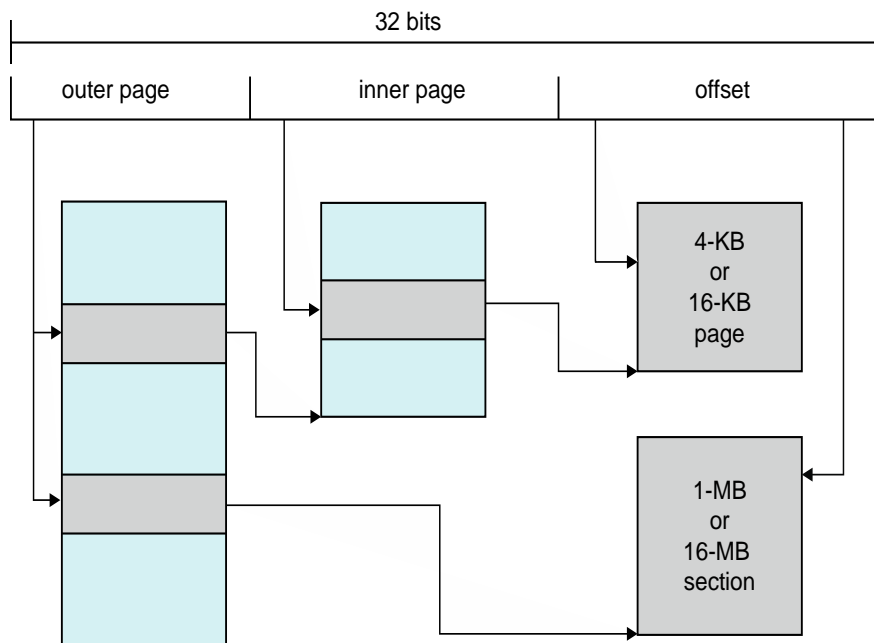
- 当前主流处理器架构
- IA-64 (Itanium)→x86-64 (amd64)
- 64 bits 地址空间惊人(> 16 exabytes)
- 实际设计虚拟寻址能力48bits
  - 页面大小: 4 KB, 2 MB, 1 GB
  - 四级分页结构



- 当采用PAE模式, 使48bits虚拟地址, 能够支持52bits物理地址

# 例子2: ARM架构

- 主导了移动计算平台芯片体系架构 (Apple iOS and Google Android)
- 能耗低, 32-bit, 64-bits
- 32bits CPU 页面大小支持:
  - 小: 4 KB & 16 KB
  - 大: 1 MB & 16 MB (称为 **sections**)
- 一级分页: 1MB & 16MB, 用于分段
- 二级分页: 用于4KB&16KB, 用于分页
- 当系统引用section or 小页面时, 分页机制相应改变
- 支持2级TLB
  - 外部页面: 2个micro TLB(数据 & 指令)
  - 内部页面: 一个main TLB
  - 优先查找micro TLB, 然后是 main TLB, 接下来是遍历页表。







# 小作业

---

1. 假设一个任务被划分成4个大小相等的段，每段有8项的页表，若页面大小为2KB。试问段页式存储系统中：
  - a. 每段最大尺寸是多少？
  - b. 该任务的逻辑地址空间最大为多少？
  - c. 若该任务访问到逻辑地址空间5ABCH中的一个数据，试给出逻辑地址的格式。
2. V9: 8.17



# 大作业

---

- 查阅资料，撰写一份调研报告，包括：
  - 调研Intel/ARM架构的内存管理机制32位/64位。
  - 以Windows 或Linux或Android等OS为例，分析OS如何实现内存管理。