第8章 存储器管理

第8章 存储器管理

8.1 存储器管理的基本概念

8.2 单一连续分配

8.3 分区管理

8.4 伙伴系统

连续分配

8.5 分页存储管理

8.6 分段存储管理

8.7 段页式存储管理

非连续分配

8.1 存储器管理的基本概念

- 存储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）

- 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。

存储器管理的四个主要功能

① 存储空间的分配和回收

- OS/进程对主存的申请与释放

② 抽象与映射

- 抽象：对进程而言，占有并使用地址连续的逻辑地址空间
- 地址映射：将逻辑地址转换成物理地址

存储器管理的四个主要功能(续)

③ 保护与共享

- 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
- 通过授权进程，允许共享访问相同的地址空间。

④ 存储扩充

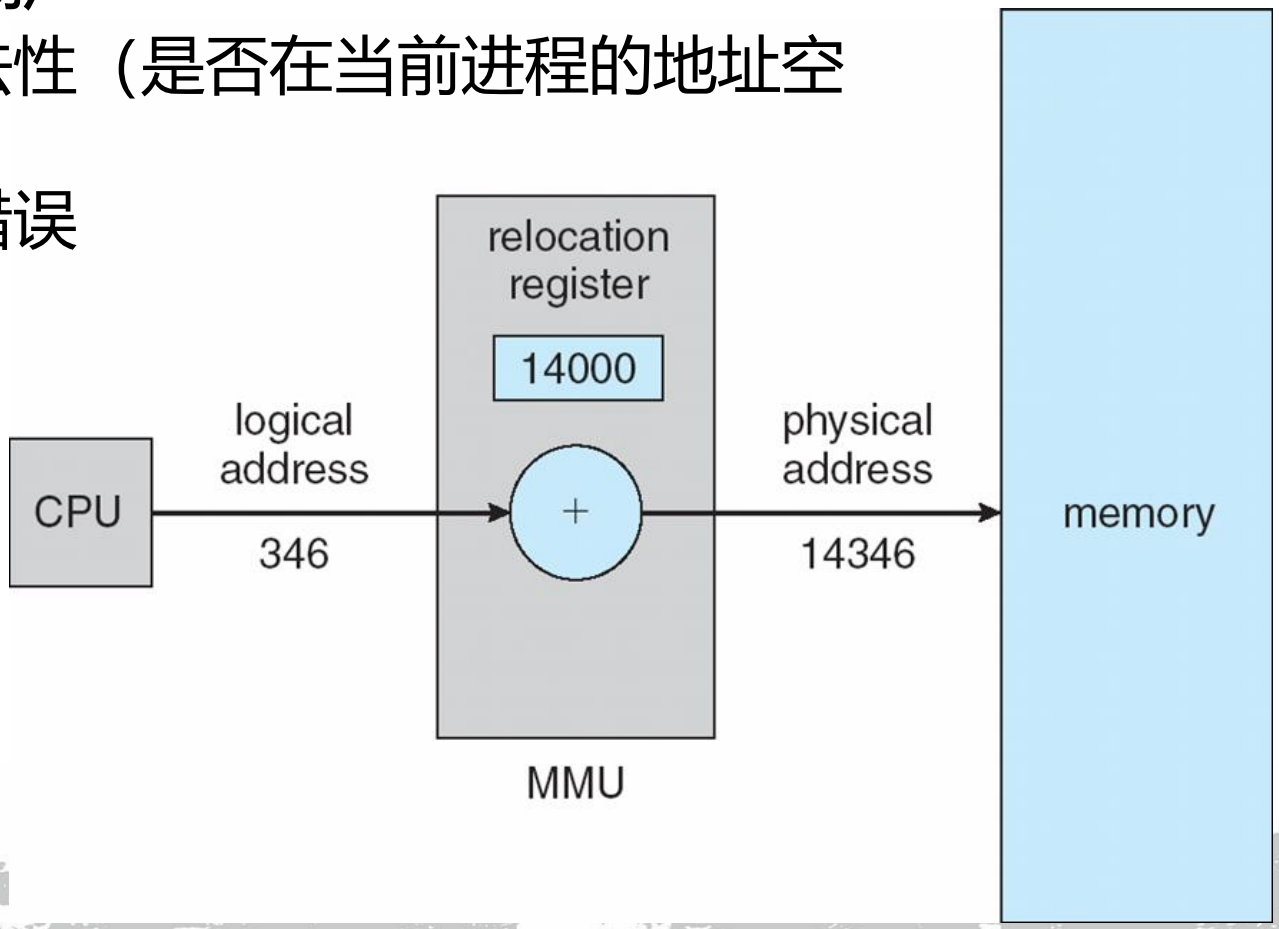
- 在逻辑上为用户提供一个比实际物理内存更大的存储空间

物理地址和逻辑地址

- 物理地址：硬件支持的地址空间，内存中的地址，绝对地址，实地址。
- 逻辑地址：在CPU运行的进程能看到的地址；编译链接时所使用的地址，又称相对地址、虚地址
- 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
  - 逻辑地址到物理地址的变换与编译、加载和运行时密切相关

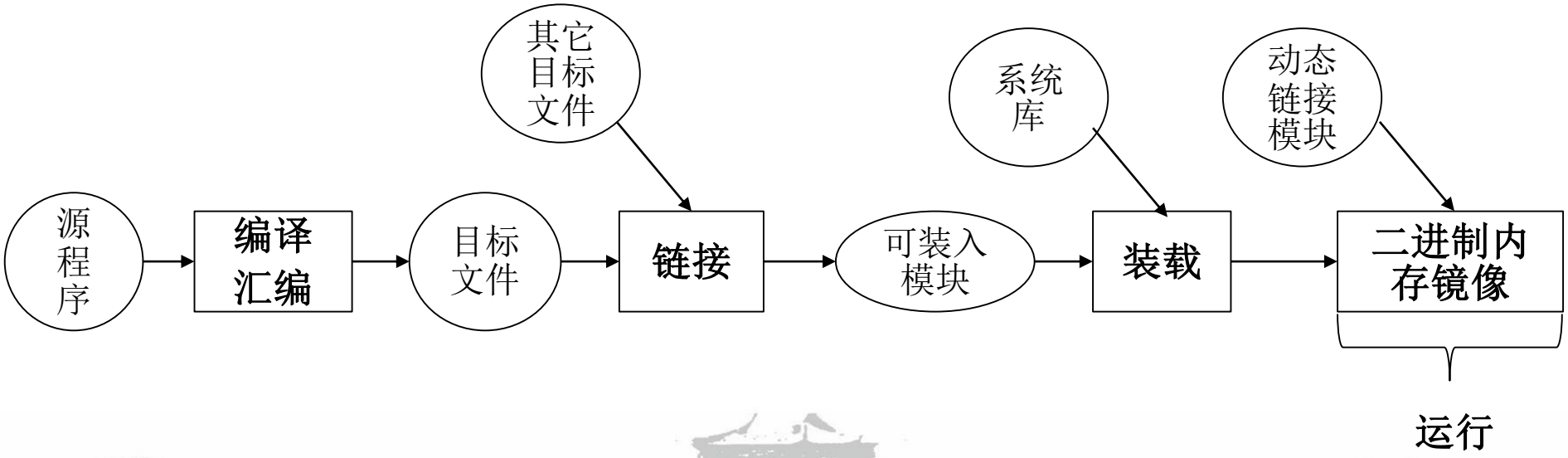
地址变换的硬件支持

- MMU通常作为硬件实现
- 实现逻辑地址到物理地址的转换
- 执行部分内存保护
  - 检查地址的合法性（是否在当前进程的地址空间内）
  - 比如越界或页错误



地址变换发生的时机

- 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、汇编、链接、装载
  - 用户程序在这个过程中用逻辑地址表示



程序装入方式

- 将程序装入内存（地址生成时机）有3种方式：
  - 绝对装入方式
    - 程序的生成使用物理地址
  - 可重定位装入方式
  - 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。
- 如果地址改变，必须重新编译

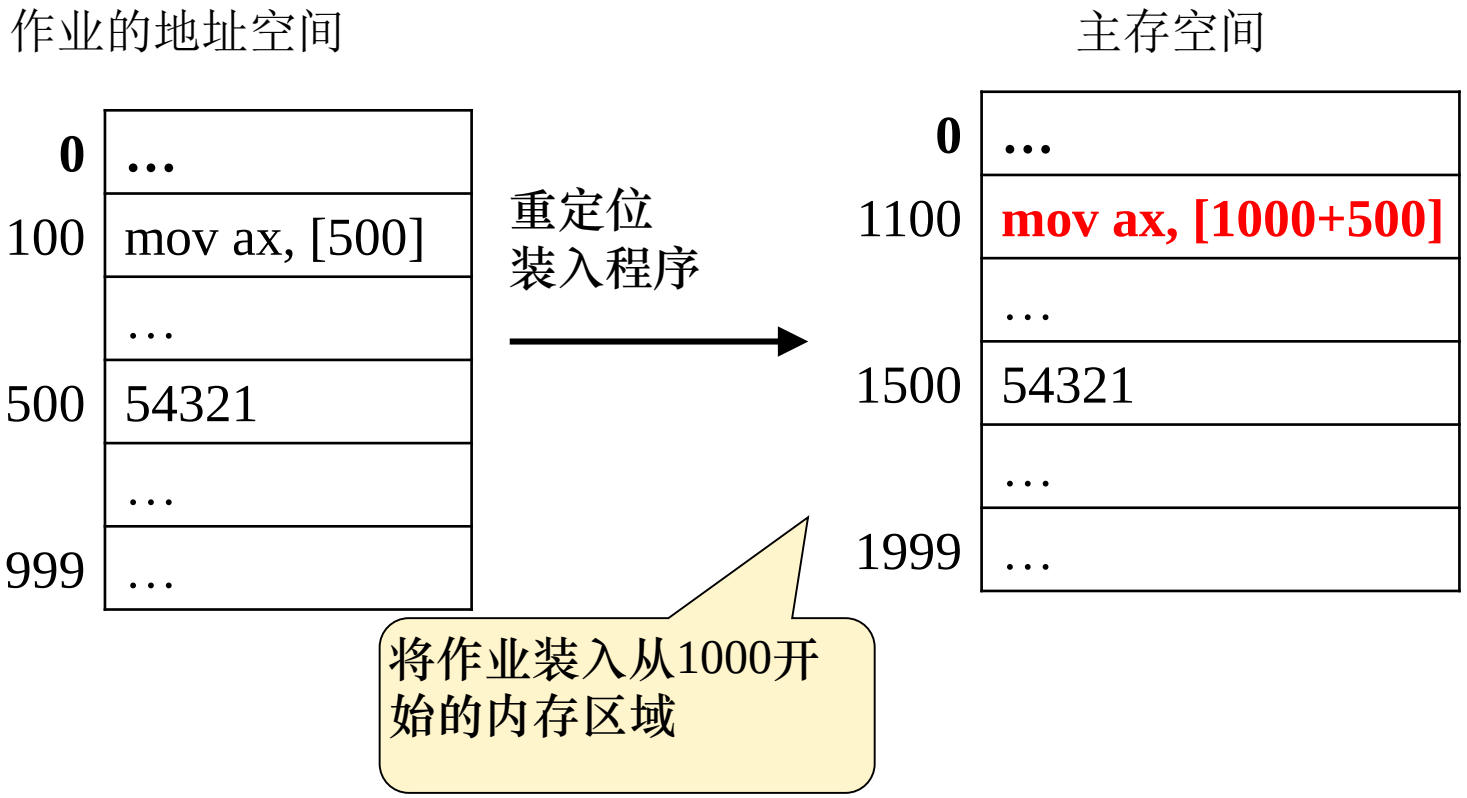
1、绝对装入方式(Cont.)

- 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布局**，将装入模块装入到内存的适当地方
  - 作业装入时，重定位作业所需访问的指令和数据，完成从相对地址到实际地址的转换；
  - 地址变换在程序装入时一次完成，又称静态地址变换（静态地址重定位）

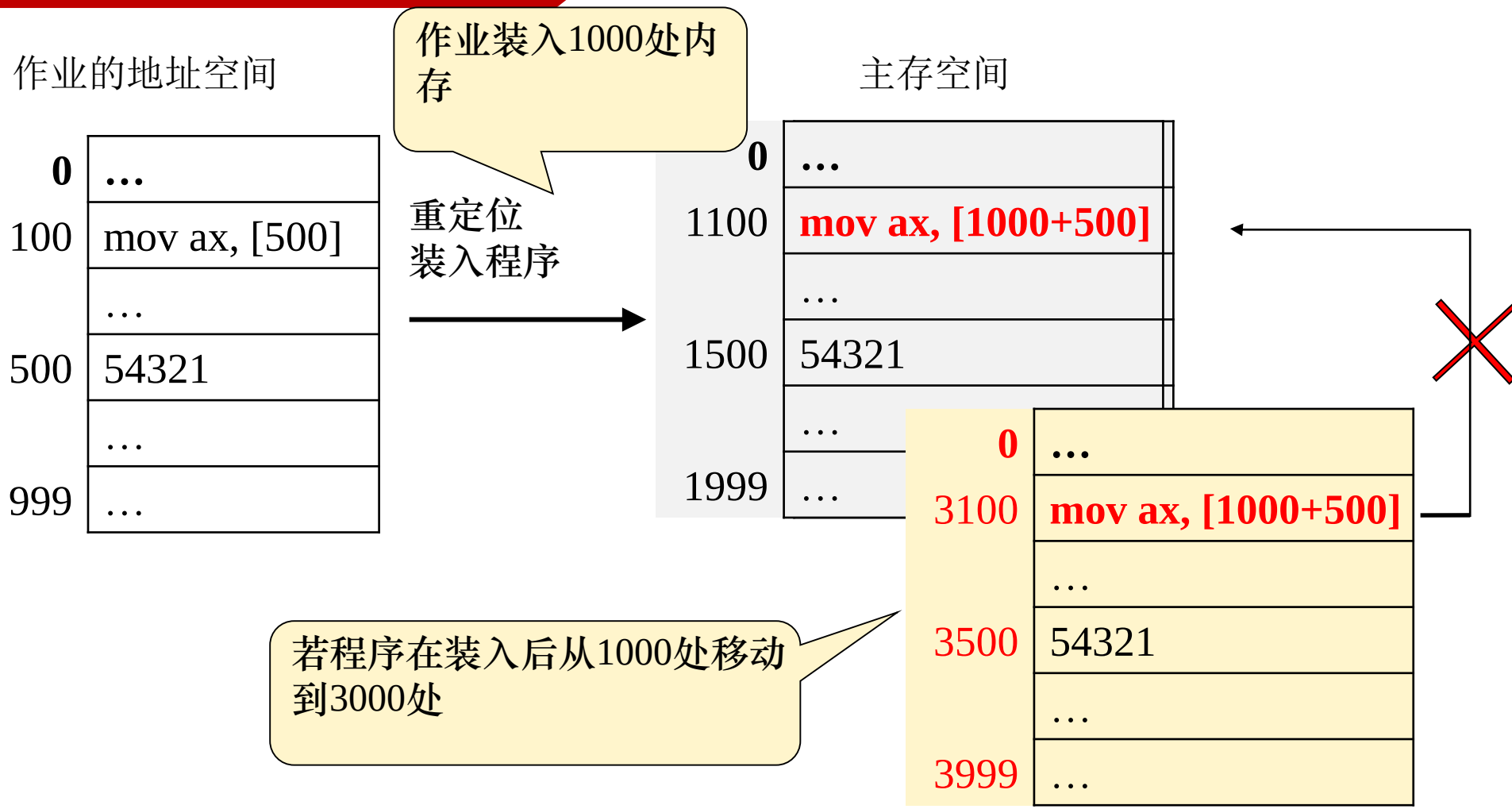
静态地址变换示意图



重定位是在装入时进行的

装入之后内存地址不再发生变化

静态地址变换示意图



程序运行时不可发生移动

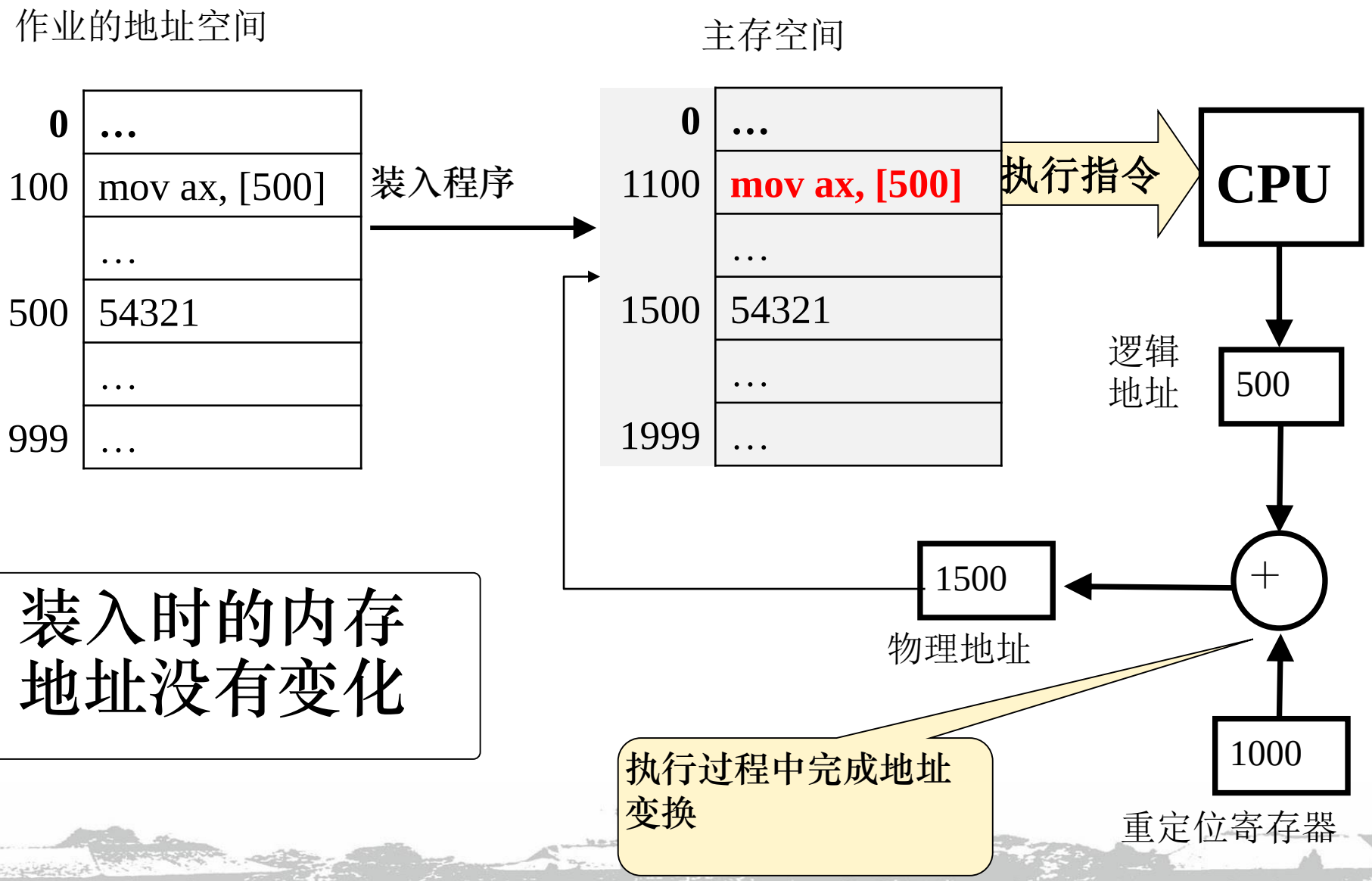
可重定位装入方式特点

- 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，难以实现数据与代码的共享。

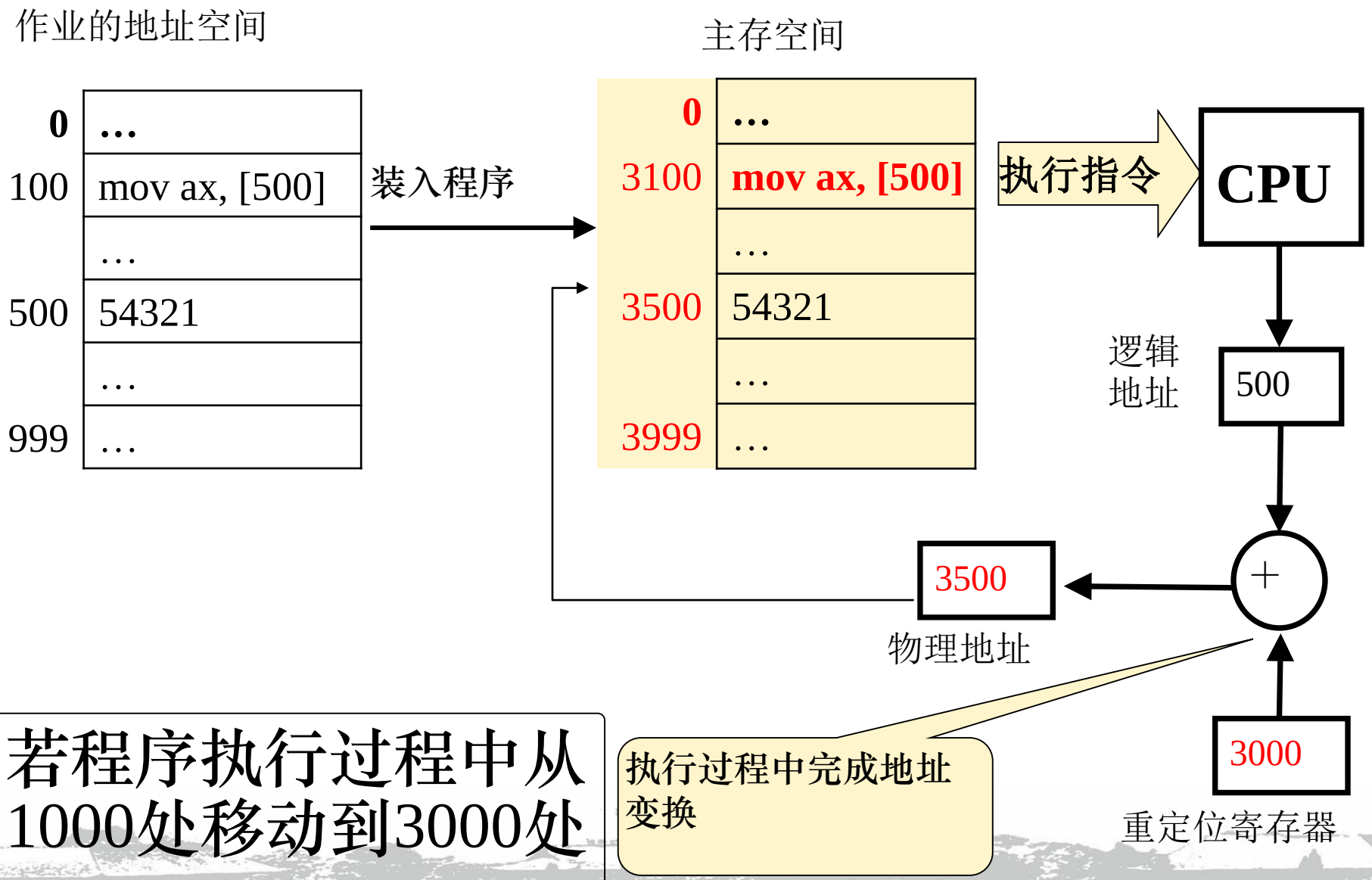
3、动态运行时装入方式

- 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
  - 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
- 程序执行过程的每次内存访问都需地址变换，又称动态地址变换（动态重定位）
  - 需要地址变换（映射）硬件支持

动态地址变换示意图


作业的地址空间
0
…
100
mov ax, [500]
…
500
54321
…
999
…
装入程序
主存空间
0
…
1100
mov ax, [500]
…
1500
54321
…
1999
…
执行指令
CPU
逻辑地址
500
+
1000
重定位寄存器
1500
物理地址
装入时的内存地址没有变化
执行过程中完成地址变换

动态地址变换示意图



若程序执行过程中从1000处移动到3000处

动态运行时装入方式的具体机制

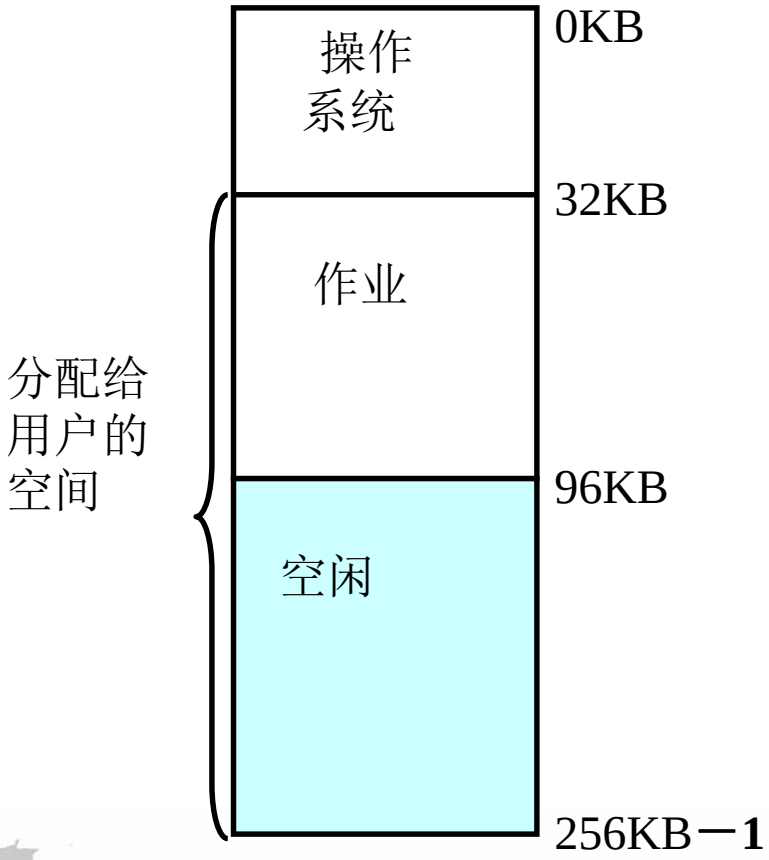
- 程序被装入内存后，程序中所引用的逻辑地址不作修改
- 程序的装入起始地址被加载到重定位寄存器。
- 程序运行时，当CPU引用主存地址时，硬件拦截此地址，在发送到主存储器之前，加上重定位寄存器的值。

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，
  - cs/ds/ss
  - 寻址：段基址+偏移

8.2 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。



8.3 分区分配

- 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

8.3.1 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。

两个固定：1、各分区的大小固定不变；
2、总分区的个数固定不变

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。

固定分区管理 - 优缺点

- 是最早采用，也是最简单的多道程序存储管理方式。
- 缺点：
  - 预先规定了分区大小，大程序无法装入。
  - 预先限制了活跃进程的最大数。
  - 主存的利用率不高：每个分区的作业不可能恰好占满该区，剩余的部分空间又不能为其它作业利用。
    - 碎片问题（内碎片）

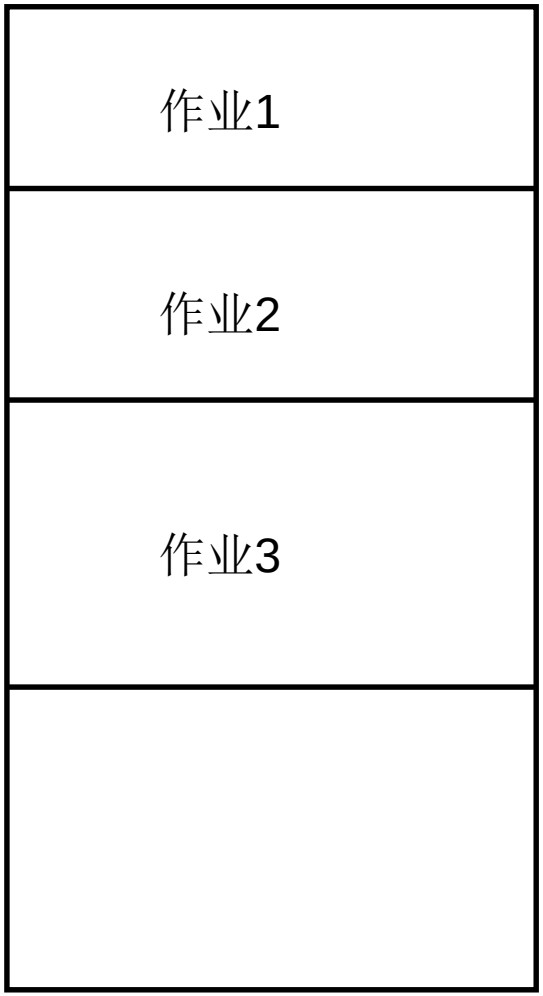
8.3.2 动态分区存储管理

- 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。

两个可变：1、各分区的大小可变；
2、总分区的个数可变。

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，2个分区
- 作业2进入，3个分区
- 作业3进入，4个分区
- 作业1结束，4个分区
- 作业3结束，3个分区
- 作业2结束，1个分区



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

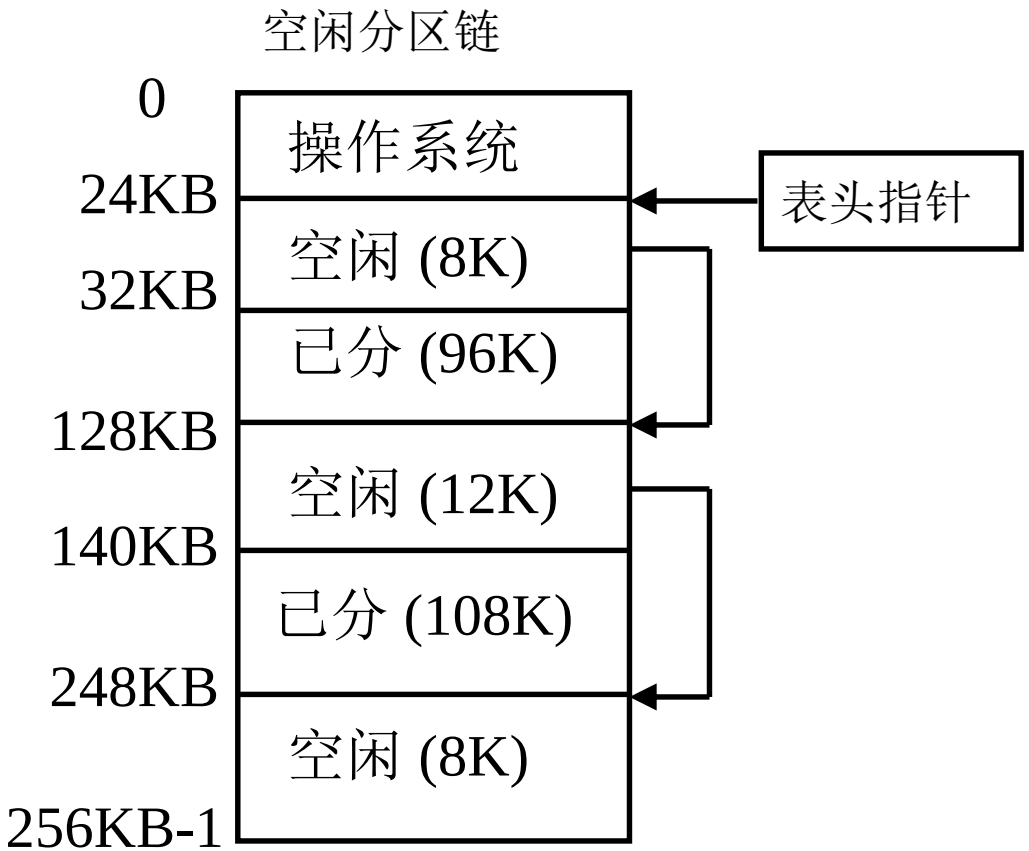
内存布局图

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



分区分配

- 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

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。
- 算法概述：
  - 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。
  - 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。

首次适应算法的特点

- 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。

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。
- 算法概述：
  - 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。

循环首次适应算法的特点

- 特点:
  - 存储空间的利用更加均衡
  - 系统缺乏大的空闲分区

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。
- 算法概述:
  - 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。

最佳适应算法的特点

- 特点：
  - 既满足作业要求又与作业大小最接近
  - 保留了大的空闲区
  - 分割后的剩余空闲区很小

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。
- 算法概述：
  - 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。

最坏适应算法的特点

- 特点：
  - 检索效率快
  - 大作业的存储空间的申请往往得不到满足

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。

例1

- 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1--采用首次适应算法分配1

- 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1--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1--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1--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

例2

- 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

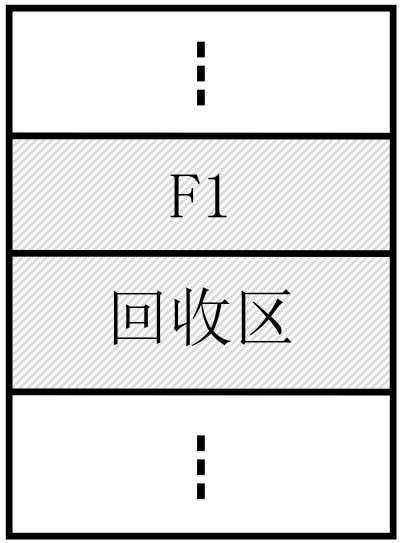
分区回收

- 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

问题：空闲分区的个数在上述几种情况下如何变化？

回收分区r上邻接一个空闲分区

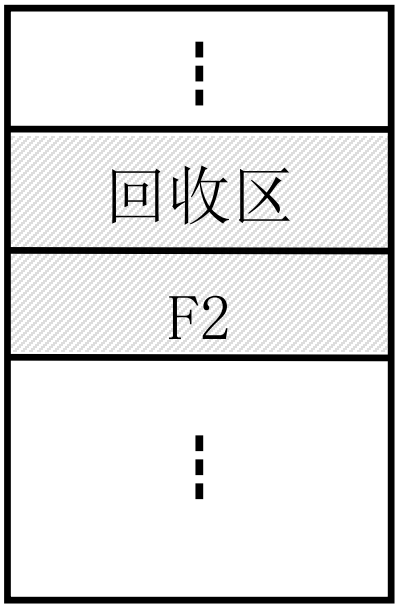
- 此时应将回收区r与上邻接分区F1合并成一个连续的空闲区。
- 合并分区的首地址为空闲区F1的首地址，
- 其大小为二者之和。



总的空闲分区数不变

回收分区r下邻接一个空闲分区

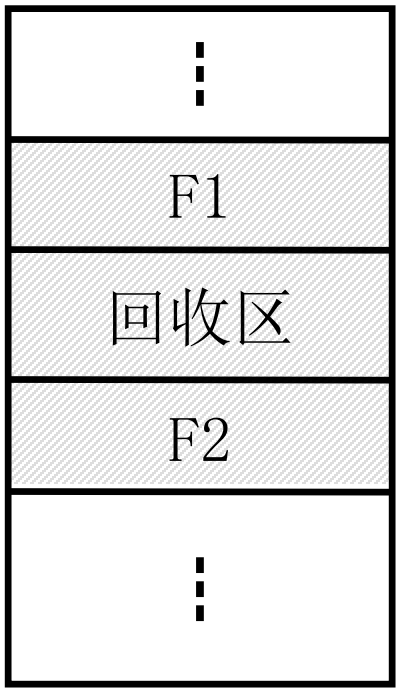
- 此时应将回收区r与下邻接分区F2合并成一个连续的空闲区。
- 合并分区的首地址为回收分区r的首地址
- 其大小为二者之和。



总的空闲分区数不变

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
- 且应将与r下邻接的空闲区F2从空闲分区表(或空闲分区链)中删去。



总的空闲分区数减少一个

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。

总的空闲分区数增加一个

动态分区管理 - 优缺点

- 相对灵活，没有固定分区中程序数目的限制和程序大小的限制。

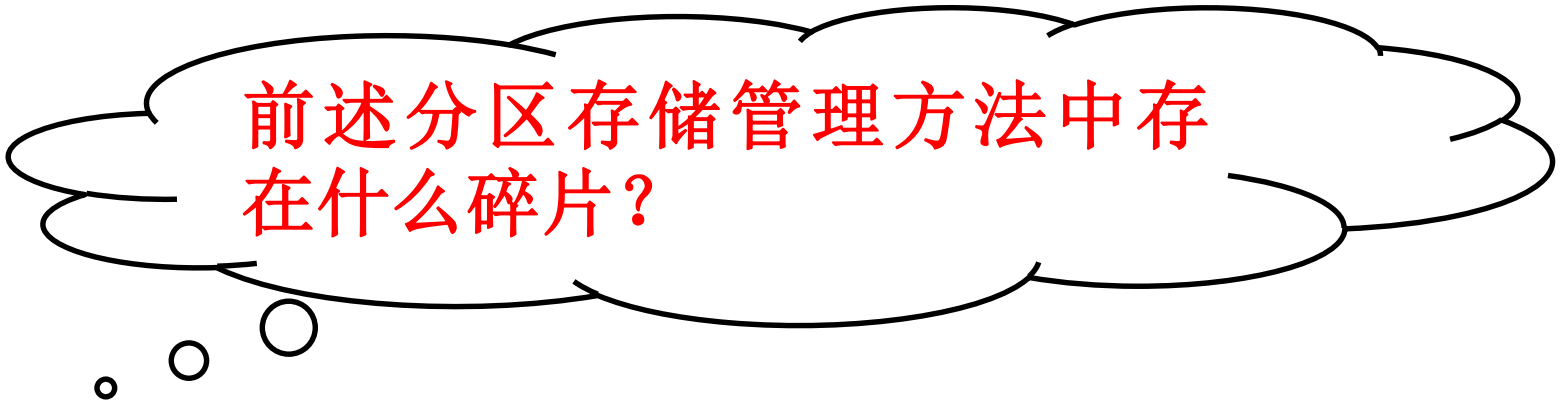
- 每道程序总是要求占用主存的连续存储区域，主存中会产生许多碎片（外碎片）。

8.3.3 可重定位分区分配

- 碎片也可称为零头，是指内存中无法被利用的存储空间。
- **碎片问题**
  - 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。

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。



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。
- 拼接的不足是要耗费大量处理机时间。

拼接示意图

- 拼接前

拼接后

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，因此需要**动态重定位技术支持**。

② 空闲区放在何处：

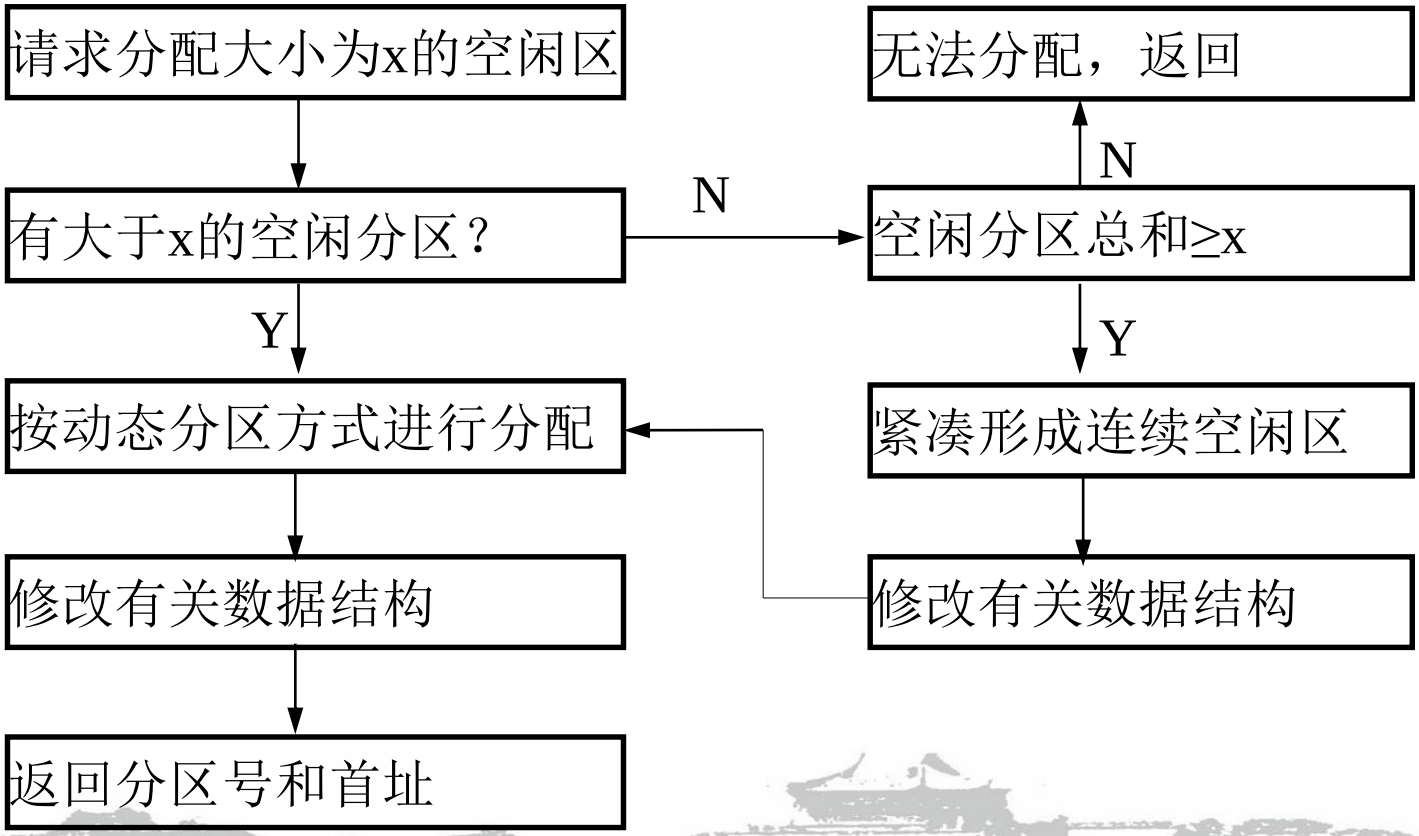
- 拼接后的空闲区放在何处不能一概而论，应根据移动进程的大小和数量多少来决定。

③ 拼接的时机问题：

- 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。
- 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。



8.4 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
- 伙伴系统主要思想：
  - 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。
- 优点：合并规则简单，易于维护

伙伴系统的内存分配

- 设进程申请大小为n
  - 系统初始时可供分配的空间为2^m。
  - 若$2^{m-1}<n\leq 2^m$，则为进程分配整个空间
  - 若$2^{i-1}<n\leq 2^i$，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}…，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。

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
  - 这个较大的空闲块也可能存在空闲伙伴，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。

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：

$$\text{Buddy}(k,d)=\begin{cases} d+2^k, & \text{若} d \bmod 2^{k+1}=0 \\ d-2^k, & \text{若} d \bmod 2^{k+1}=2^k \end{cases}$$

- 如果参与分配的2^m个单元从a开始，则对于长度为2^K、开始地址为d的块，其伙伴的开始地址为：

$$\text{Buddy}(k,d)=\begin{cases} d+2^k, & \text{若}(d-a) \bmod 2^{k+1}=0 \\ d-2^k, & \text{若}(d-a) \bmod 2^{k+1}=2^k \end{cases}$$

- 举例：块的地址为011011110000b, 若块大小分别为4和16，块的伙伴地址分别为？

伙伴系统分配及回收例

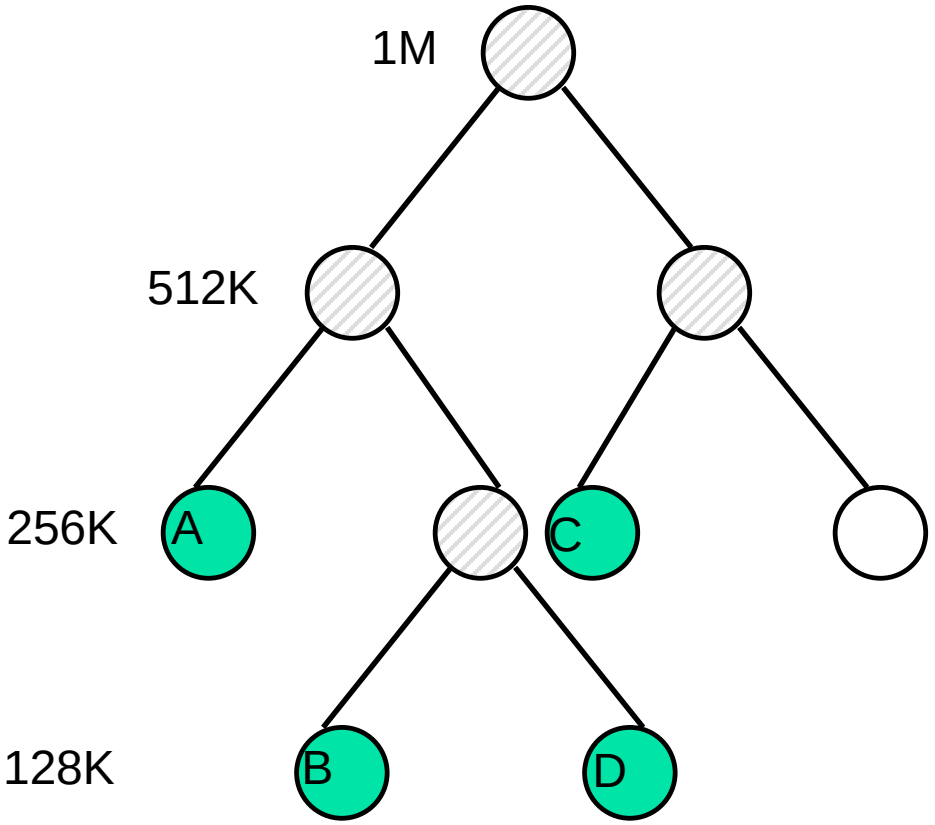
- 设系统中初始内存空间大小为1MB，进程请求和释放空间的操作序列为：
  - 进程A申请200KB；B申请120KB；C申请240KB；D申请100KB；
  - 进程B释放；E申请60KB；
  - 进程A、C释放；
  - 进程D释放；进程E释放。

分配过程示意图

	0	128K	256K	384K	512K	640K	768K	896K	1M
初始状态									
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									

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。



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