第四章　存储器管理

BACK

4.1 存储器的层次结构

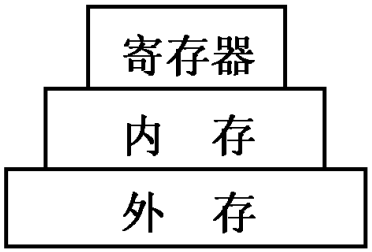
4.1.1 多级存储器结构

对于通用计算机而言，存储层次至少应具有三级：最高层为CPU寄存器，中间为主存，最底层是辅存。在较高档的计算机中，还可以根据具体的功能分工细划为寄存器、高速缓存、主存储器、磁盘缓存、固定磁盘、可移动存储介质等6层。

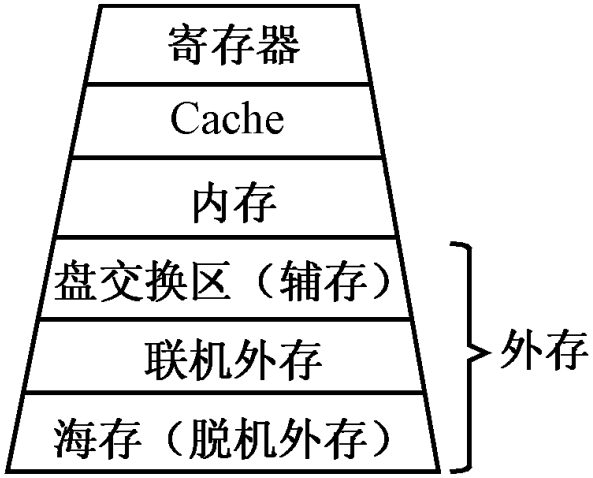
如图4-1所示，在存储层次中越往上，存储介质的访问速度越快，价格也越高，相对存储容量也越小。其中，寄存器、高速缓存、主存储器和磁盘缓存均属于操作系统存储管理的管辖范畴，掉电后它们存储的信息不再存在。固定磁盘和可移动存储介质属于设备管理的管辖范畴，它们存储的信息将被长期保存。

4.1.1 多级存储器结构

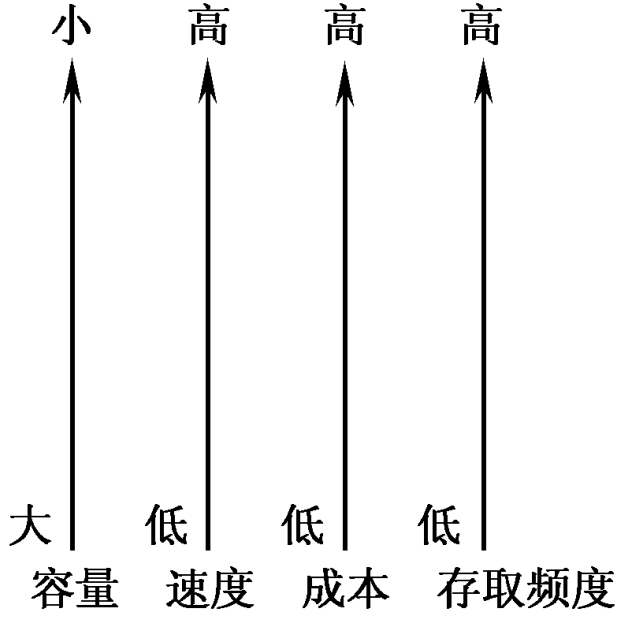
存储层次



粗层次图



细层次图



（其中 Cache 还可以再分三层：片内一级和二级，片外 Cache ）

4.1.2 主存储器与寄存器

1．主存储器

主存储器(简称内存或主存)是计算机系统中一个主要部件，用于保存进程运行时的程序和数据，也称可执行存储器，其容量对于当前的微机系统和大中型机，可能一般为数十MB到数GB，而且容量还在不断增加，而嵌入式计算机系统一般仅有几十KB到几MB。

CPU的控制部件只能从主存储器中取得指令和数据，数据能够从主存储器读取并将它们装入到寄存器中，或者从寄存器存入到主存储器。CPU与外围设备交换的信息一般也依托于主存储器地址空间。由于主存储器的访问速度远低于CPU执行指令的速度，为缓和这一矛盾，在计算机系统中引入了寄存器和高速缓存。

4.1.2 主存储器与寄存器

2．寄存器

寄存器访问速度最快，完全能与CPU协调工作，但价格却十分昂贵，因此容量不可能做得很大。寄存器的长度一般以字(word)为单位。寄存器的数目，对于当前的微机系统和大中型机，可能有几十个甚至上百个；而嵌入式计算机系统一般仅有几个到几十个。寄存器用于加速存储器的访问速度，如用寄存器存放操作数，或用作地址寄存器加快地址转换速度等。

4.1.3 高速缓存和磁盘缓存

1．高速缓存

高速缓存是现代计算机结构中的一个重要部件，其容量大于或远大于寄存器，而比内存约小两到三个数量级左右，从几十KB到几MB，访问速度快于主存储器。

根据程序执行的局部性原理(即程序在执行时将呈现出局部性规律，在一较短的时间内，程序的执行仅局限于某个部分)，将主存中一些经常访问的信息存放在高速缓存中，减少访问主存储器的次数，可大幅度提高程序执行速度。

4.1.3 高速缓存和磁盘缓存

2．磁盘缓存

由于目前磁盘的I/O速度远低于对主存的访问速度，因此将频繁使用的一部分磁盘数据和信息，暂时存放在磁盘缓存中，可减少访问磁盘的次数。磁盘缓存本身并不是一种实际存在的存储介质，它依托于固定磁盘，提供对主存储器存储空间的扩充，即利用主存中的存储空间，来暂存从磁盘中读出(或写入)的信息。

主存也可以看做是辅存的高速缓存，因为，辅存中的数据必须复制到主存方能使用；反之，数据也必须先存在主存中，才能输出到辅存。

4.1.3 高速缓存和磁盘缓存

一个文件的数据可能出现在存储器层次的不同级别中，例如，一个文件数据通常被存储在辅存中(如硬盘)，当其需要运行或被访问时，就必须调入主存，也可以暂时存放在主存的磁盘高速缓存中。

大容量的辅存常常使用磁盘，磁盘数据经常备份到磁带或可移动磁盘组上，以防止硬盘故障时丢失数据。有些系统自动地把老文件数据从辅存转储到海量存储器中，如磁带上，这样做还能降低存储价格。

存储器管理

内存管理功能

- 内存分配与回收:（空间记录表格设计、分配/回收算法）
- 地址转换：逻辑地址→物理地址（装入程序，表、地址转换机构、处理算法）
- 内存保护：防止越界、非法访问。（界地址/限长寄存器，表格、处理算法）
- 内存共享：多个进程→一个程序。（可重入代码，访问权限）
- 内存扩充：逻辑扩充，小内存运行大程序。（覆盖、交换、虚拟存储器）

存储器管理

1.内存的分配与回收

内存分配按分配时机的不同，可分为三种方式：

（1）直接分配

这种方式是指程序员采用物理内存地址编写程序。使用这种方式，必须事先划定内存的使用空间，因此，这种方式下的内存利用率不高，用户使用较困难。

（2）静态分配

指内存分配是在作业运行之前各目标模块连接后，把整个作业一次性全部装入内存，并在作业的整个运行过程中，不允许作业再申请其他内存，或在内存中移动位置。也就是说，内存分配是在作业运行前一次性完成的。

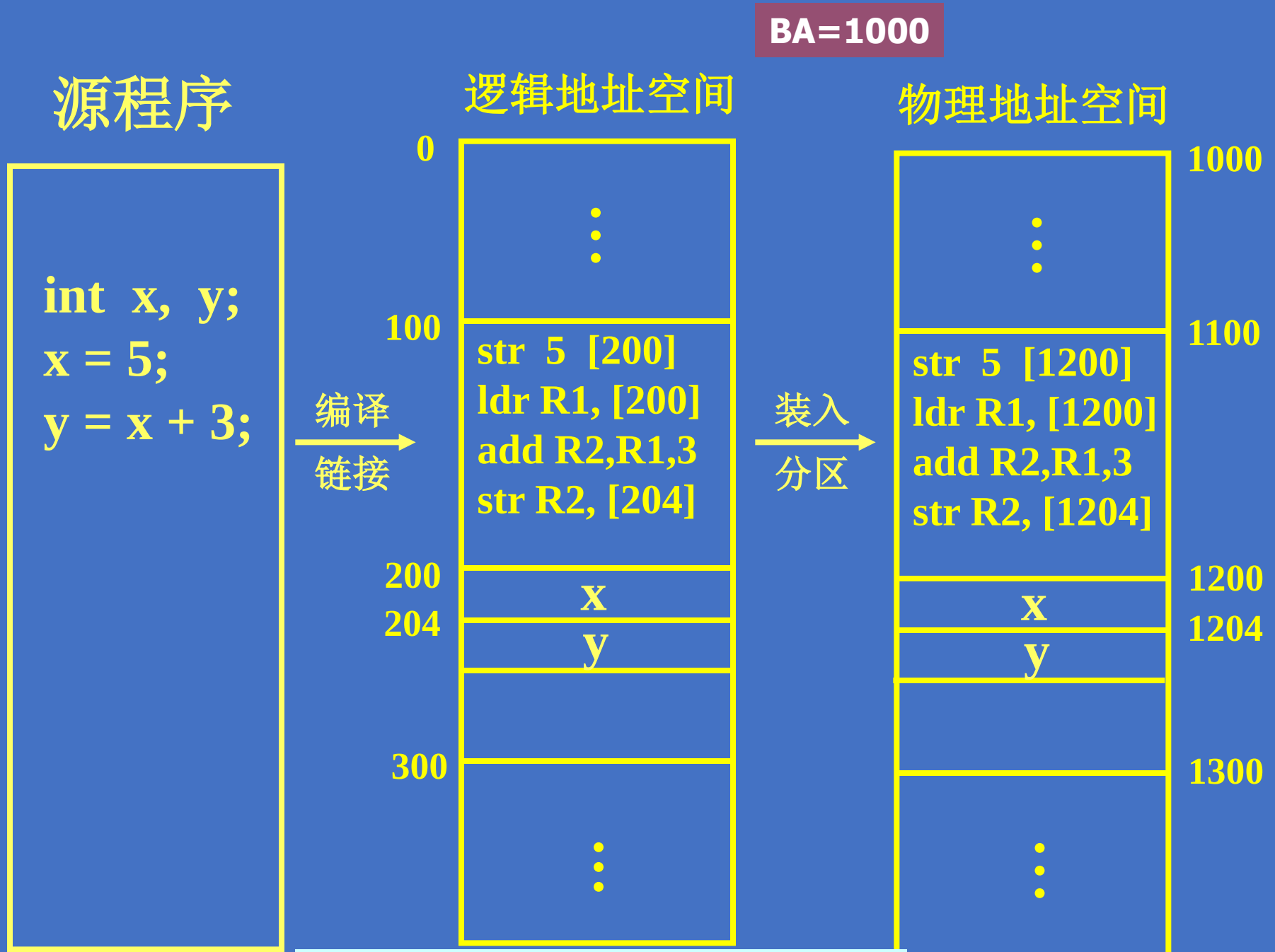
（3）动态分配

作业要求的基本内存空间是在目标模块装入内存时分配的，但在作业运行过程中，允许作业申请附加的内存空间，或是在内存中移动，即分配工作可以在作业运行前及运行过程中逐步完成。

存储器管理

2．地址转换

- 把用户程序装入内存时对有关指令的地址部分的修改定义为从程序地址到内存地址的地址转换，或称为地址重定位。
- 物理地址:内存地址，它是用于唯一标识一个内存单元的编号。所有的物理地址构成了物理空间。
- 逻辑地址:程序地址/相对地址/虚地址。它是指在源程序经过汇编或编译后形成的目标代码中，用于反映目标代码中指令或数据的相对位置关系的地址。逻辑地址都是以“0”为基址顺序进行编址的，这样生成的目标程序占据一定的地址空间，称为程序的逻辑地址空间，简称逻辑空间。
- 符号地址（符号名）：表示的程序空间称为名空间。
- 地址重定位的原因是因为程序在装入内存后，其逻辑地址和内存地址不一致。



三种空间转换例(静态重定位)

4.1.3 高速缓存和磁盘缓存

3. 内存的共享与保护

- 内存共享：多个进程共享内存中的同一段信息。
  - 优点：节省内存空间，减少I/O量，提高性能。
  - 实现：

    1. 代码必须是可重入的。

    2. 必须设置共享的数据结构（如共享计数器等），并管理这个数据结构。

存储器管理

内存的保护

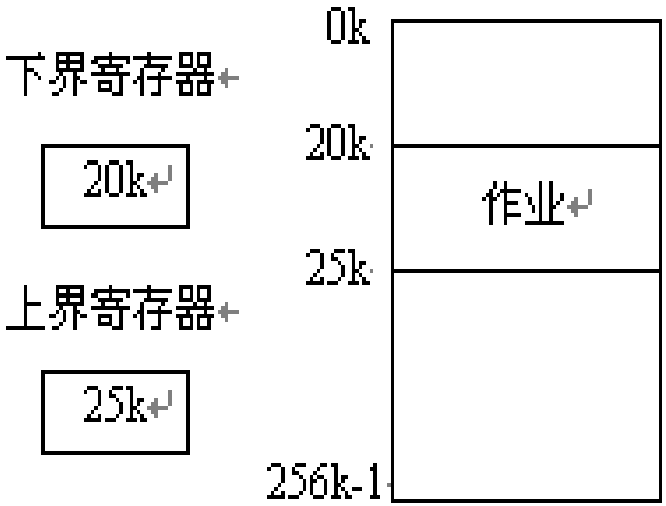
- 内存保护
  - 防止进程的数据被非法访问
  - 防止越界
- 防止非法访问：
  - 设权限位

存储器管理

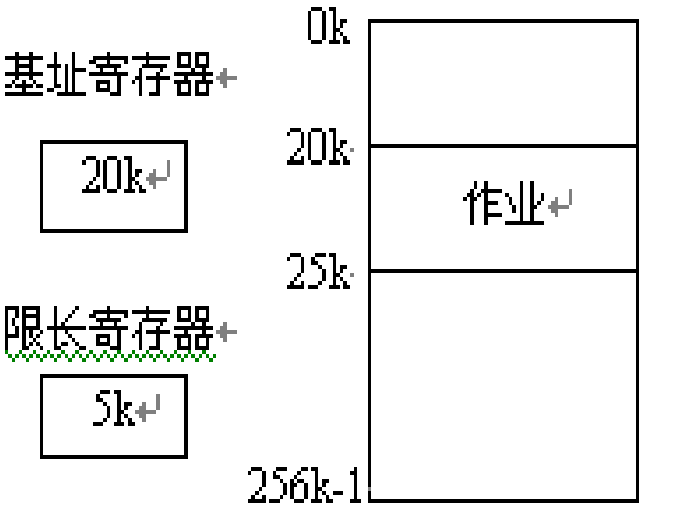
防止越界

1.上、下界存储保护：系统可为每个作业设置一对上、下界寄存器，分别用来存放当前运行作业在内存空间的上、下边界地址，用它们来限制用户程序的活动范围。

2.基址 - 限长存储保护：上、下界保护的一个变种是采用基址—限长存储保护。



（a）上、下界保护



（b）基址——限长保护

存储器管理

4．内存扩充

- 对内存进行逻辑上的扩充，现在普遍采用覆盖、交换和虚拟存储器技术。
- 虚拟存储器：程序不完整存放的自动透明实现技术。具有请求调入功能和置换功能，能仅把作业的一部分装入内存便可运行作业的存储器系统。
- 虚拟存储器的理论基础是程序的局部性原理。

  - 时间局部性：某段刚被访问的信息，可能很快会被再次访问。

  - 空间局部性：某段信息刚被访问，其相邻单元信息可能很快会被访问。

4.2 程序的装入和链接

在多道程序环境下，要使程序运行，必须先为之创建进程。而创建进程的第一件事，便是将程序和数据装入内存。如何将一个用户源程序变为一个可在内存中执行的程序，通常都要经过以下几个步骤：

首先是要编译，由编译程序(Compiler)将用户源代码编译成若干个目标模块(Object Module)；

其次是链接，由链接程序(Linker)将编译后形成的一组目标模块，以及它们所需要的库函数链接在一起，形成一个完整的装入模块(Load Module)；

最后是装入，由装入程序(Loader)将装入模块装入内存。图4-2示出了这样的三步过程。本节将扼要阐述程序(含数据)的链接和装入过程。

4.2 程序的装入和链接

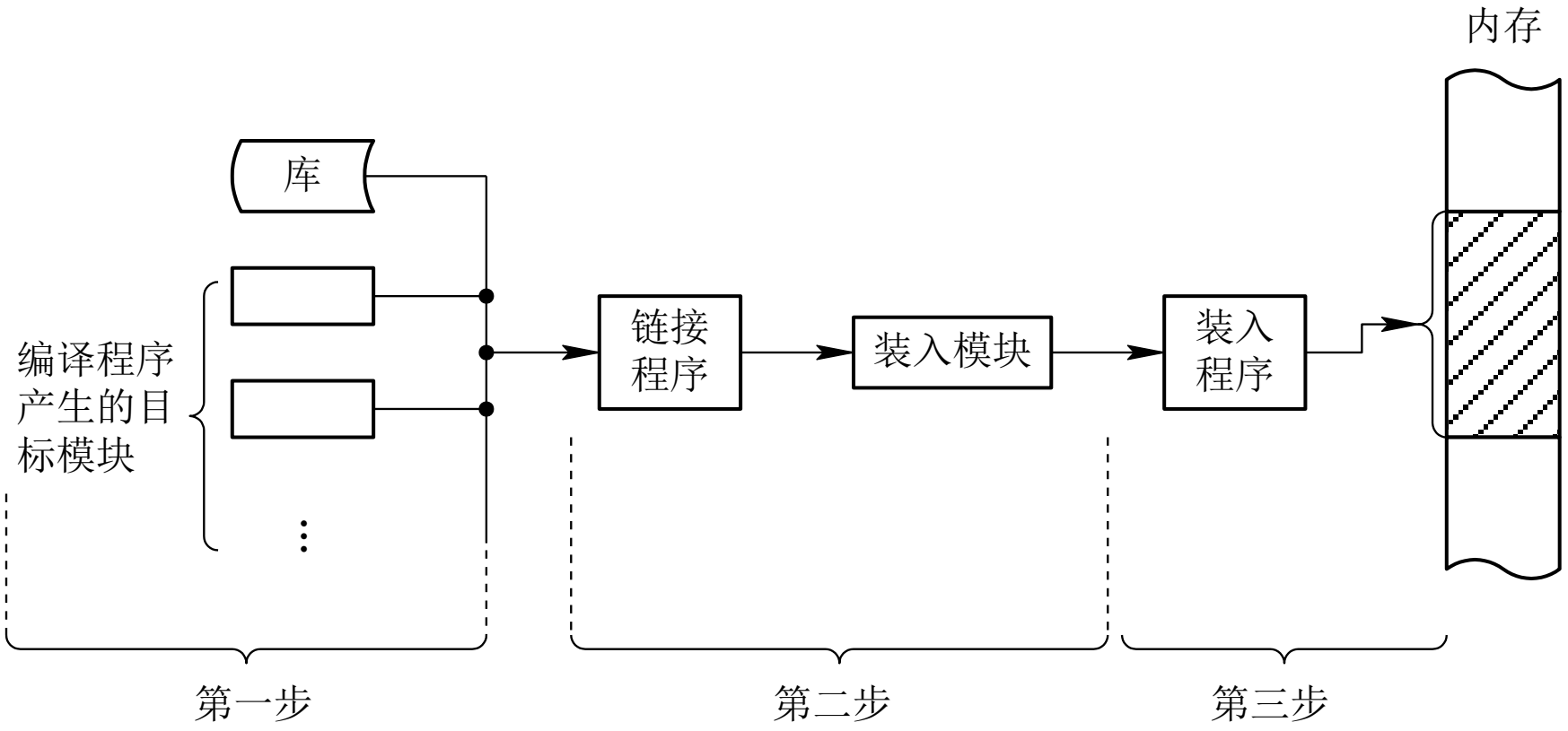


图4-2 对用户程序的处理步骤

4.2.1 程序的装入

1．绝对装入方式(Absolute Loading Mode)

在编译时，如果知道程序将驻留在内存的什么位置，那么，编译程序将产生绝对地址的目标代码。例如，事先已知用户程序(进程)驻留在从R处开始的位置，则编译程序所产生的目标模块(即装入模块)便从R处开始向上扩展。

绝对装入程序按照装入模块中的地址，将程序和数据装入内存。装入模块被装入内存后，由于程序中的逻辑地址与实际内存地址完全相同，故不须对程序和数据的地址进行修改。

4.2.1 程序的装入

2．可重定位装入方式(Relocation Loading Mode)

绝对装入方式只能将目标模块装入到内存中事先指定的位置。在多道程序环境下，编译程序不可能预知所编译的目标模块应放在内存的何处，因此，绝对装入方式只适用于单道程序环境。

在多道程序环境下，所得到的目标模块的起始地址通常是从0开始的，程序中的其它地址也都是相对于起始地址计算的。此时应采用可重定位装入方式，根据内存的当前情况，将装入模块装入到内存的适当位置。

4.2.1 程序的装入

地址重定位的方式

- 静态重定位

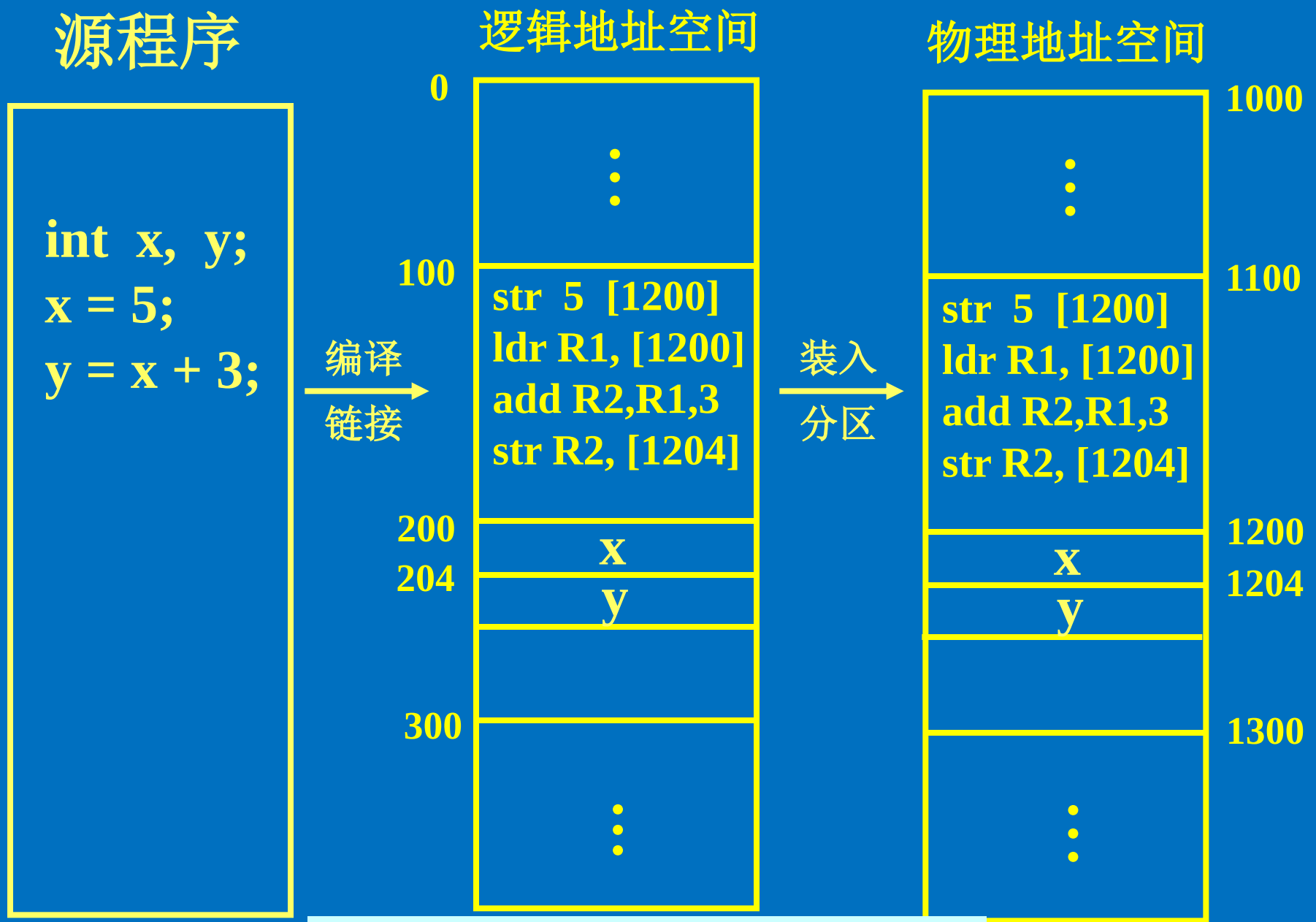
  是在程序执行之前由操作系统的连接装入程序完成地址转换

  - 优点：不需要硬件的支持。
  - 缺点：程序必须占用连续的内存空间；一旦程序装入后不能移动。

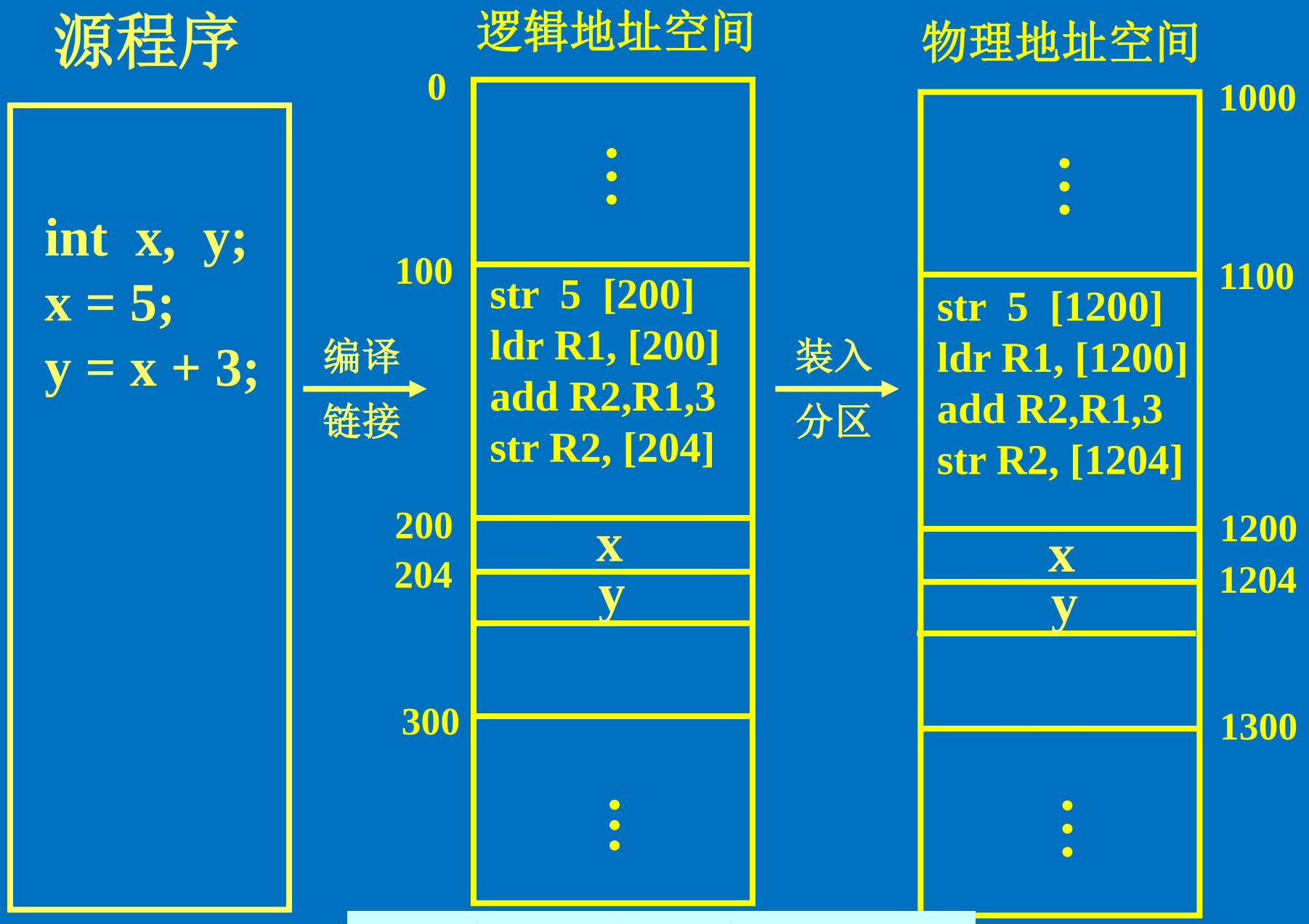
- 动态重定位

  是在程序执行期间进行的地址转换，是由专门的硬件机构来完成的。

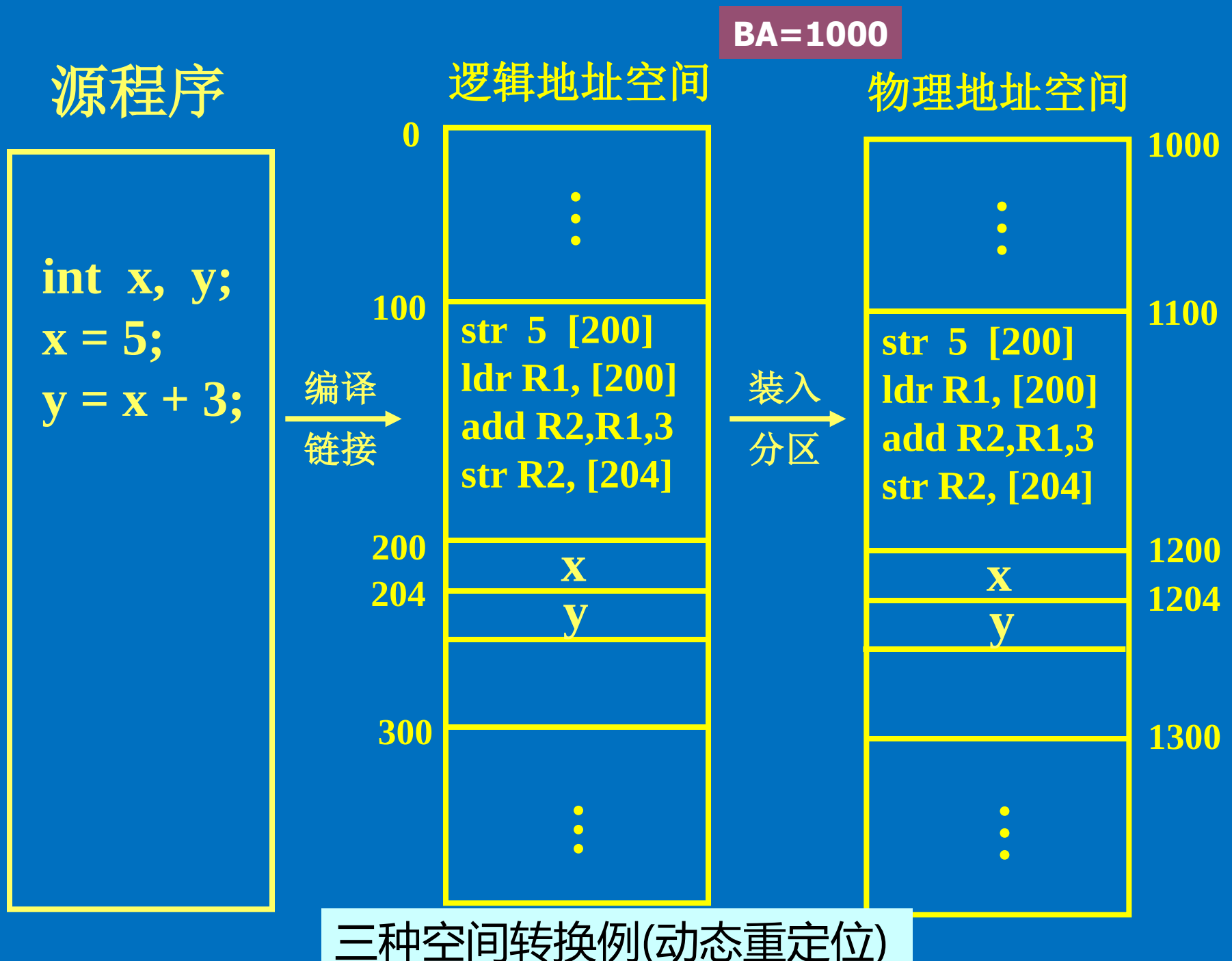
  - 优点：程序占用的内存空间是动态可变的，当程序从某个存储区移到另一个区域时，只需要修改相应的寄存器BR的内容即可。
  - 缺点：需要硬件的支持；实现存储管理的软件算法较为复杂。



三种空间转换例(绝对装入方式)



三种空间转换例(静态重定位)



三种空间转换例(动态重定位)

4.2.1 程序的装入

2．动态重定位的实现

在动态运行时装入的方式中，作业装入内存后的所有地址都仍然是相对地址，将相对地址转换为物理地址的工作，被推迟到程序指令要真正执行时进行。为使地址的转换不会影响到指令的执行速度，必须有硬件地址变换机构的支持，即须在系统中增设一个重定位寄存器，用它来存放程序(数据)在内存中的起始地址。

程序在执行时，真正访问的内存地址是相对地址与重定位寄存器中的地址相加而形成的。图4-10示出了动态重定位的实现原理。地址变换过程是在程序执行期间，随着对每条指令或数据的访问自动进行的，故称为动态重定位。当系统对内存进行了“紧凑”而使若干程序从内存的某处移至另一处时，不需对程序做任何修改，只要用该程序在内存的新起始地址，去置换原来的起始地址即可。

4.2.1 程序的装入

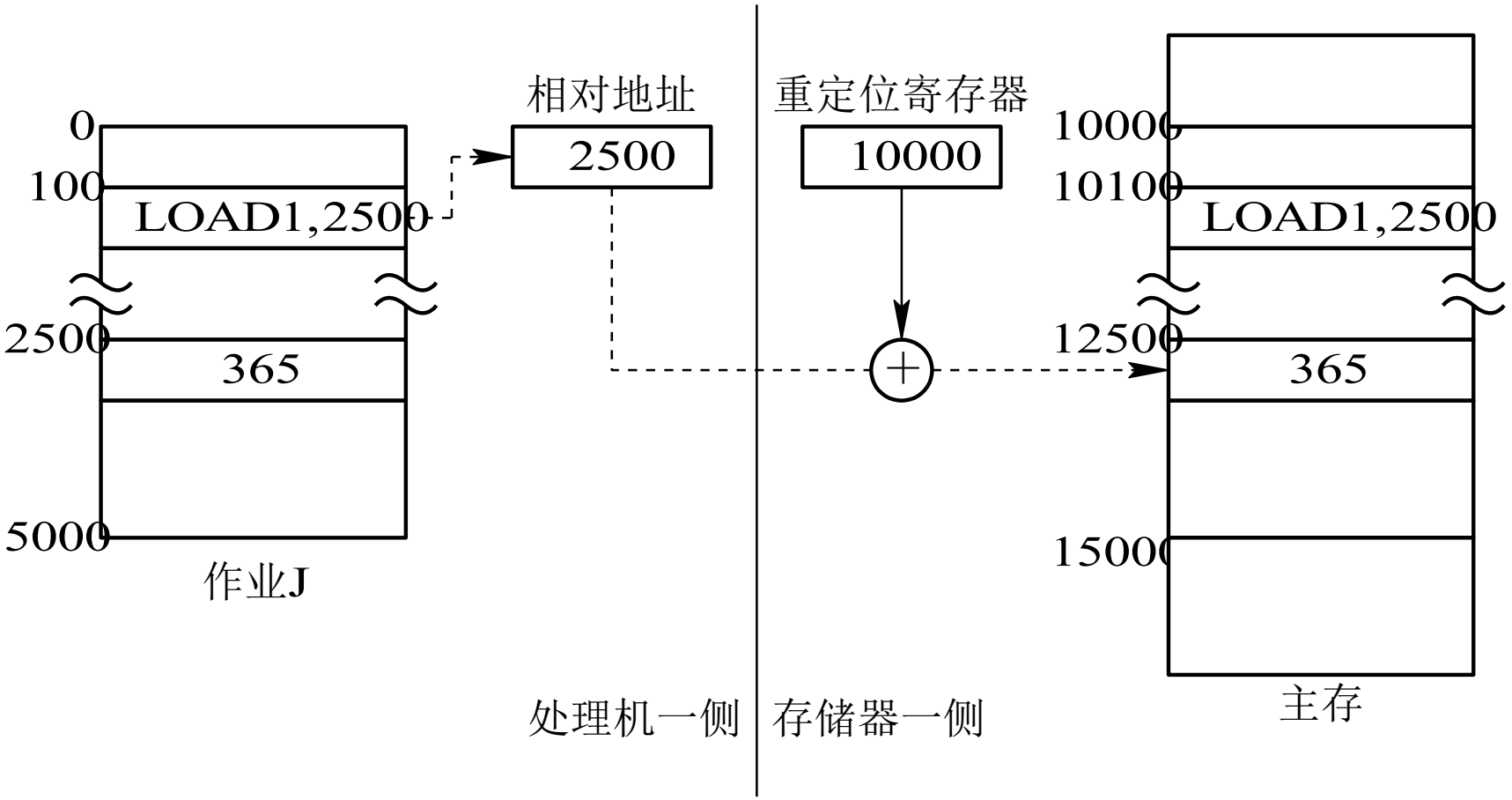


图 4-10 动态重定位示意图

4.2.2 程序的链接

根据链接时间的不同，可把链接分成如下三种：

(1) 静态链接。在程序运行之前，先将各目标模块及它们所需的库函数，链接成一个完整的装配模块，以后不再拆开。我们把这种事先进行链接的方式称为静态链接方式。

(2) 装入时动态链接。这是指将用户源程序编译后所得到的一组目标模块，在装入内存时，采用边装入边链接的链接方式。

(3) 运行时动态链接。这是指对某些目标模块的链接，是在程序执行中需要该(目标)模块时，才对它进行的链接。

4.2.2 程序的链接

1．静态链接方式(Static Linking)

(1) 对相对地址进行修改。在由编译程序所产生的所有目标模块中，使用的都是相对地址，其起始地址都为0，每个模块中的地址都是相对于起始地址计算的。在链接成一个装入模块后，原模块B和C在装入模块的起始地址不再是0，而分别是L和L+M，所以此时须修改模块B和C中的相对地址，即把原B中的所有相对地址都加上L，把原C中的所有相对地址都加上L+M。

4.2.2 程序的链接

(2) 变换外部调用符号。将每个模块中所用的外部调用符号也都变换为相对地址，如把B的起始地址变换为L，把C的起始地址变换为L+M，如图4-4(b)所示。这种先进行链接所形成的一个完整的装入模块，又称为可执行文件。通常都不再拆开它，要运行时可直接将它装入内存。这种事先进行链接，以后不再拆开的链接方式，称为静态链接方式。

4.2.2 程序的链接

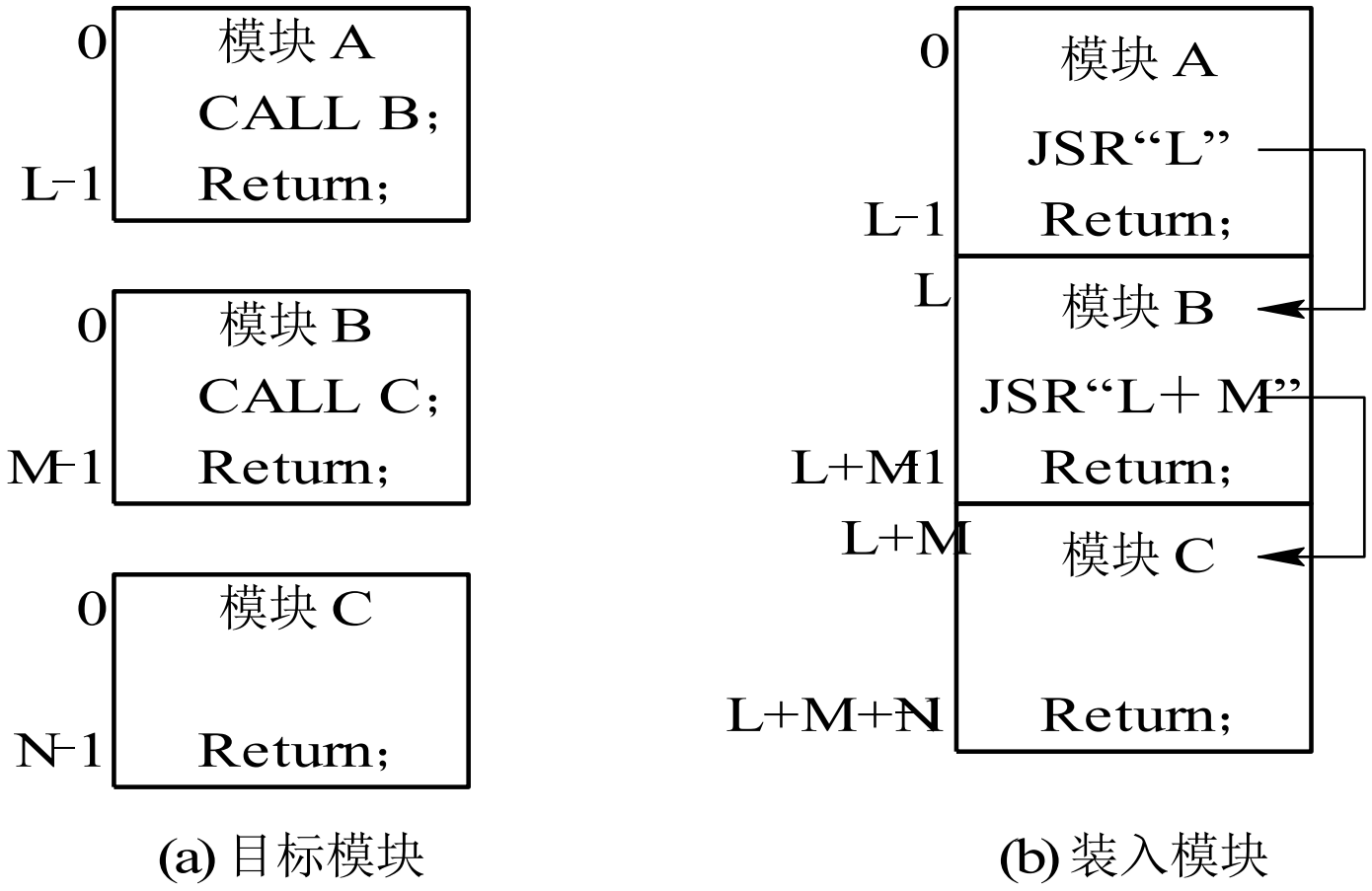


图 4-4 程序链接示意图

4.2.2 程序的链接

2．装入时动态链接(Load-time Dynamic Linking)

用户源程序经编译后所得的目标模块，是在装入内存时边装入边链接的，即在装入一个目标模块时，若发生一个外部模块调用事件，将引起装入程序去找出相应的外部目标模块，并将它装入内存，还要按照图4-4所示的方式来修改目标模块中的相对地址。装入时动态链接方式有以下优点：

(1) 便于修改和更新。对于经静态链接装配在一起的装入模块，如果要修改或更新其中的某个目标模块，则要求重新打开装入模块。这不仅是低效的，而且有时是不可能的。若采用动态链接方式，由于各目标模块是分开存放的，所以要修改或更新各目标模块是件非常容易的事。

4.2.2 程序的链接

(2) 便于实现对目标模块的共享。在采用静态链接方式时，每个应用模块都必须含有其目标模块的拷贝，无法实现对目标模块的共享。但采用装入时动态链接方式，OS则很容易将一个目标模块链接到几个应用模块上，实现多个应用程序对该模块的共享。

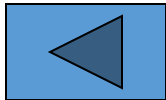
4.2.2 程序的链接

3．运行时动态链接(Run-time Dynamic Linking)

在许多情况下，应用程序在运行时，每次要运行的模块可能是不相同的。但由于事先无法知道本次要运行哪些模块，故只能是将所有可能要运行到的模块都全部装入内存，并在装入时全部链接在一起。显然这是低效的，因为往往会有些目标模块根本就不运行。比较典型的例子是作为错误处理用的目标模块，如果程序在整个运行过程中都不出现错误，则显然就不会用到该模块。

4.2.2 程序的链接

近几年流行起来的运行时动态链接方式，是对上述在装入时链接方式的一种改进。这种链接方式是将对某些模块的链接推迟到程序执行时才进行链接，亦即，在执行过程中，当发现一个被调用模块尚未装入内存时，立即由OS去找到该模块并将之装入内存，把它链接到调用者模块上。凡在执行过程中未被用到的目标模块，都不会被调入内存和被链接到装入模块上，这样不仅可加快程序的装入过程，而且可节省大量的内存空间。



4.2.2 程序的链接

存储管理方案

- 单连续
- 固定分区
- 可变分区

连续分配方式

- 页式
- 段式
- 段页式

非连续分配方式

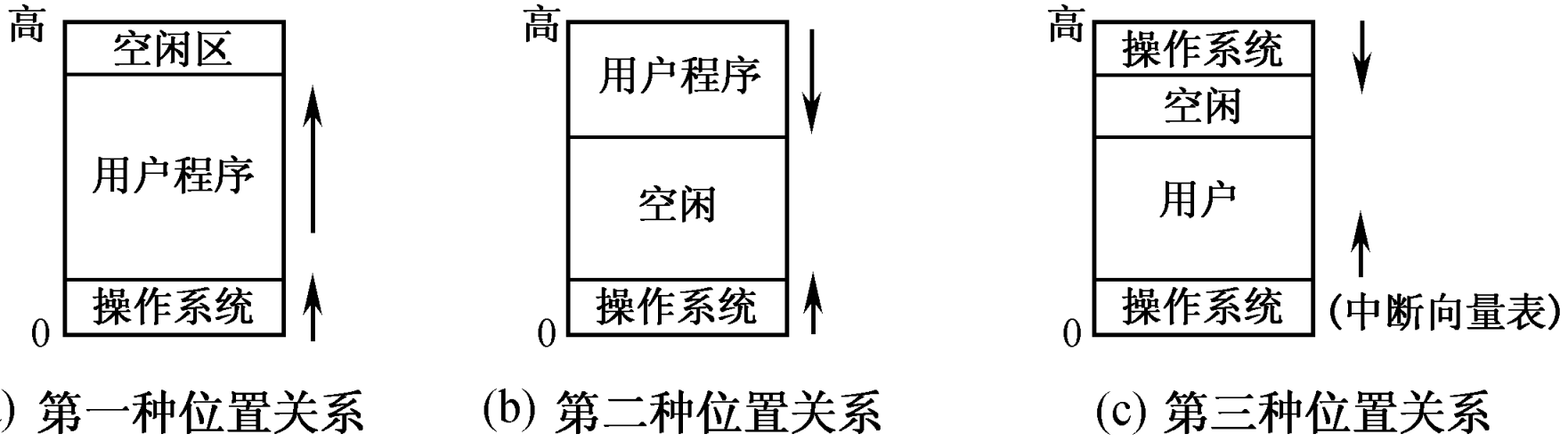
4.3 连续分配方式

4.3.1 单一连续分配

这是最简单的一种存储管理方式，但只能用于单用户、单任务的操作系统中。采用这种存储管理方式时，可把内存分为系统区和用户区两部分，系统区仅提供给OS使用，通常是放在内存的低址部分；用户区是指除系统区以外的全部内存空间，提供给用户使用。

4.3.1 单一连续分配

系统区与用户区的位置关系



(a) 第一种位置关系　(b) 第二种位置关系　(c) 第三种位置关系

4.3.2 固定分区分配

1．划分分区的方法

可用下述两种方法将内存的用户空间划分为若干个固定大小的分区：

(1) 分区大小相等，即使所有的内存分区大小相等。其缺点是缺乏灵活性，即当程序太小时，会造成内存空间的浪费；当程序太大时，一个分区又不足以装入该程序，致使该程序无法运行。尽管如此，这种划分方式仍被用于利用一台计算机去控制多个相同对象的场合，因为这些对象所需的内存空间是大小相等的。例如，炉温群控系统，就是利用一台计算机去控制多台相同的冶炼炉。

4.3.2 固定分区分配

(2) 分区大小不等。为了克服分区大小相等而缺乏灵活性的这个缺点，可把内存区划分成含有多个较小的分区、适量的中等分区及少量的大分区。这样，便可根据程序的大小为之分配适当的分区。

4.3.2 固定分区分配

2．内存分配

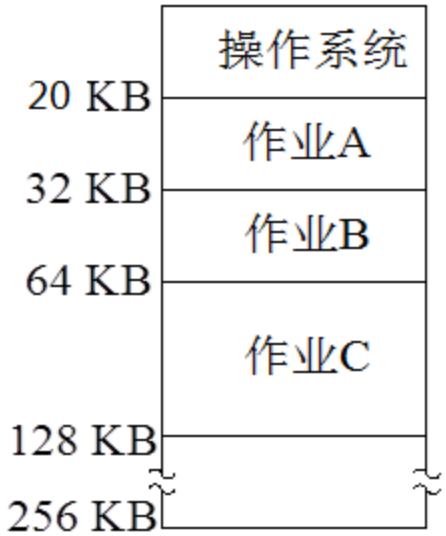
为了便于内存分配，通常将分区按大小进行排队，并为之建立一张分区使用表，其中各表项包括每个分区的起始地址、大小及状态(是否已分配)，见图4-5(a)所示。

当有一用户程序要装入时，由内存分配程序检索该表，从中找出一个能满足要求的、尚未分配的分区，将之分配给该程序，然后将该表项中的状态置为“已分配”；若未找到大小足够的分区，则拒绝为该用户程序分配内存。存储空间分配情况如图4-5(b)所示。

4.3.2 固定分区分配

分区号	大小/KB	起址/KB	状态
1	12	20	已分配
2	32	32	已分配
3	64	64	已分配
4	128	128	未分配

(a) 分区说明表



(b) 存储空间分配情况

图 4-5 固定分区使用表

4.3.2 固定分区分配

固定分区特点

- ✓ 优点：相对于单一分区，存储利用率高，可实现多道，简单。
- ◘ 缺点：

☹内存利用率低。区经常未被填满，存在内部存储碎片

☹存在因分区太小而无法装入的现象

☹多道度数受分区数所限

☹动态扩充时需要付出移动代价

☹程序间不能共享

☹整个程序装入一个分区，不能实现虚存。

4.3.3 动态分区分配

动态分区分配是根据进程的实际需要，动态地为之分配内存空间。

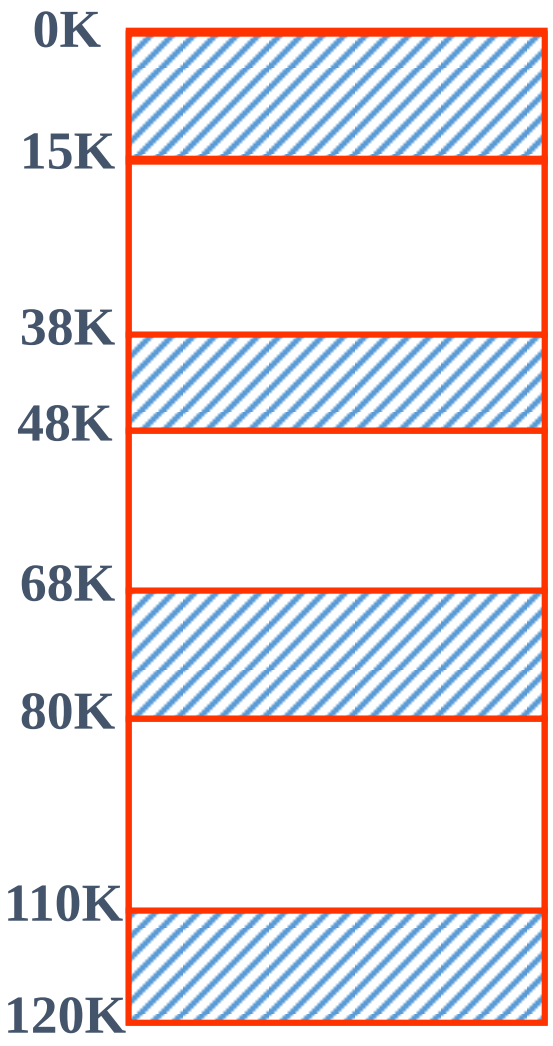
（a）可变式分区运行开始　　（b）作业1.2.3.4进入内存　　（c）作业1.3释放后内存

4.3.3 动态分区分配

1．分区分配中的数据结构

为了实现分区分配，系统中必须配置相应的数据结构，来描述空闲分区和已分配分区的情况，为分配提供依据。常用的数据结构有两种方式：

- 空闲(自由)区表
- 空闲(自由)区链



空闲区表

始址	长度	标志
15K	23K	未分配
48K	20K	未分配
80K	30K	未分配
		空
		空

已分配区表

始址	长度	标志
0K	15K	J1
38K	10K	J2
68K	12K	J3
110K	10K	J4
		空
		空

4.3.3 动态分区分配

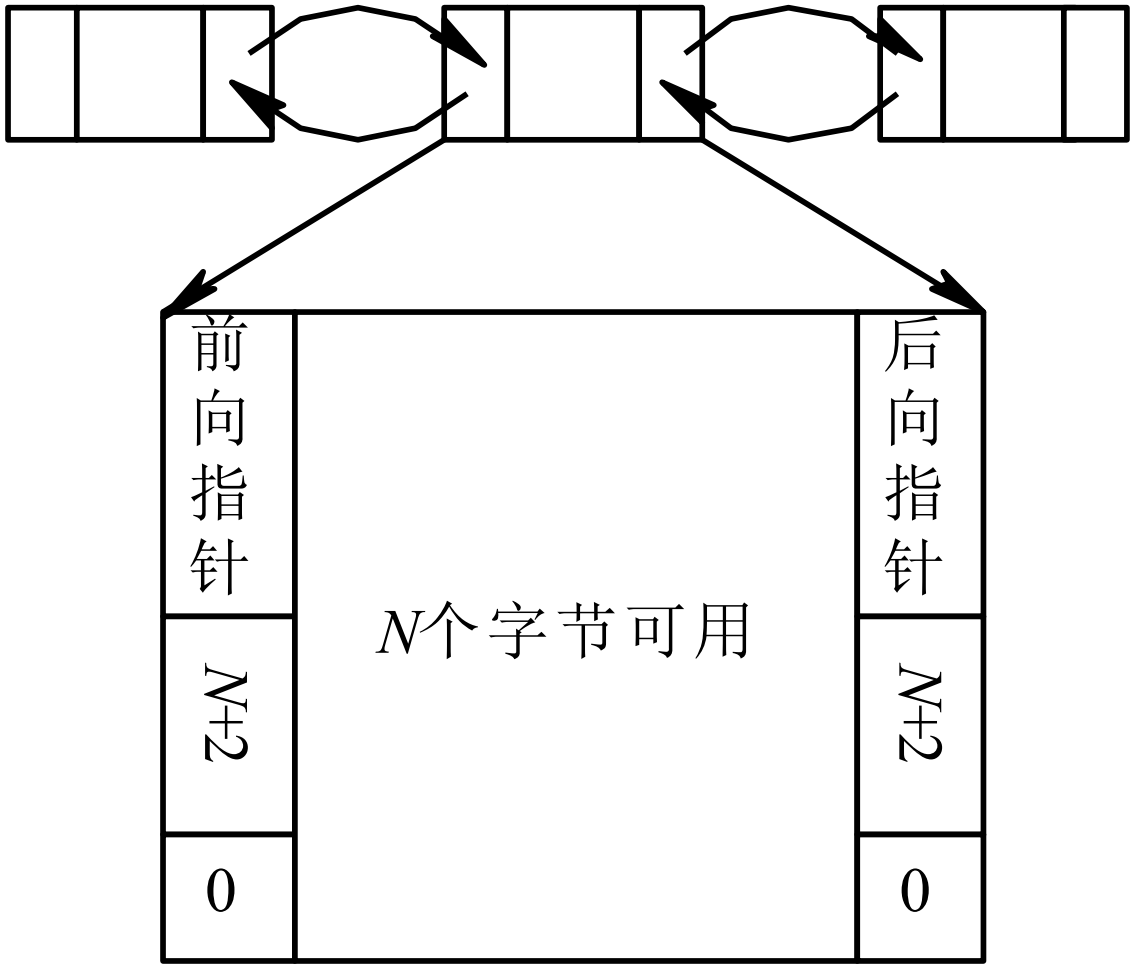


图4-6 空闲区链结构

4.3.3 动态分区分配

(1) 空闲分区表。在系统中设置一张空闲分区表，用于记录每个空闲分区的情况。每个空闲分区占一个表目，表目中包括分区序号、分区始址及分区的大小等数据项。

(2) 空闲分区链。为了实现对空闲分区的分配和链接，在每个分区的起始部分，设置一些用于控制分区分配的信息，以及用于链接各分区所用的前向指针；在分区尾部则设置一后向指针，通过前、后向链接指针，可将所有的空闲分区链接成一个双向链，如图4-6所示。为了检索方便，在分区尾部重复设置状态位和分区大小表目。当分区被分配出去以后，把状态位由“0”改为“1”，此时，前、后向指针已无意义。

4.3.3 动态分区分配

2．分区分配算法

a) 首次适应算法(first fit)

b) 循环首次适应算法(next fit)

c) 最佳适应算法(best fit)

d) 最坏适应算法(worst fit)

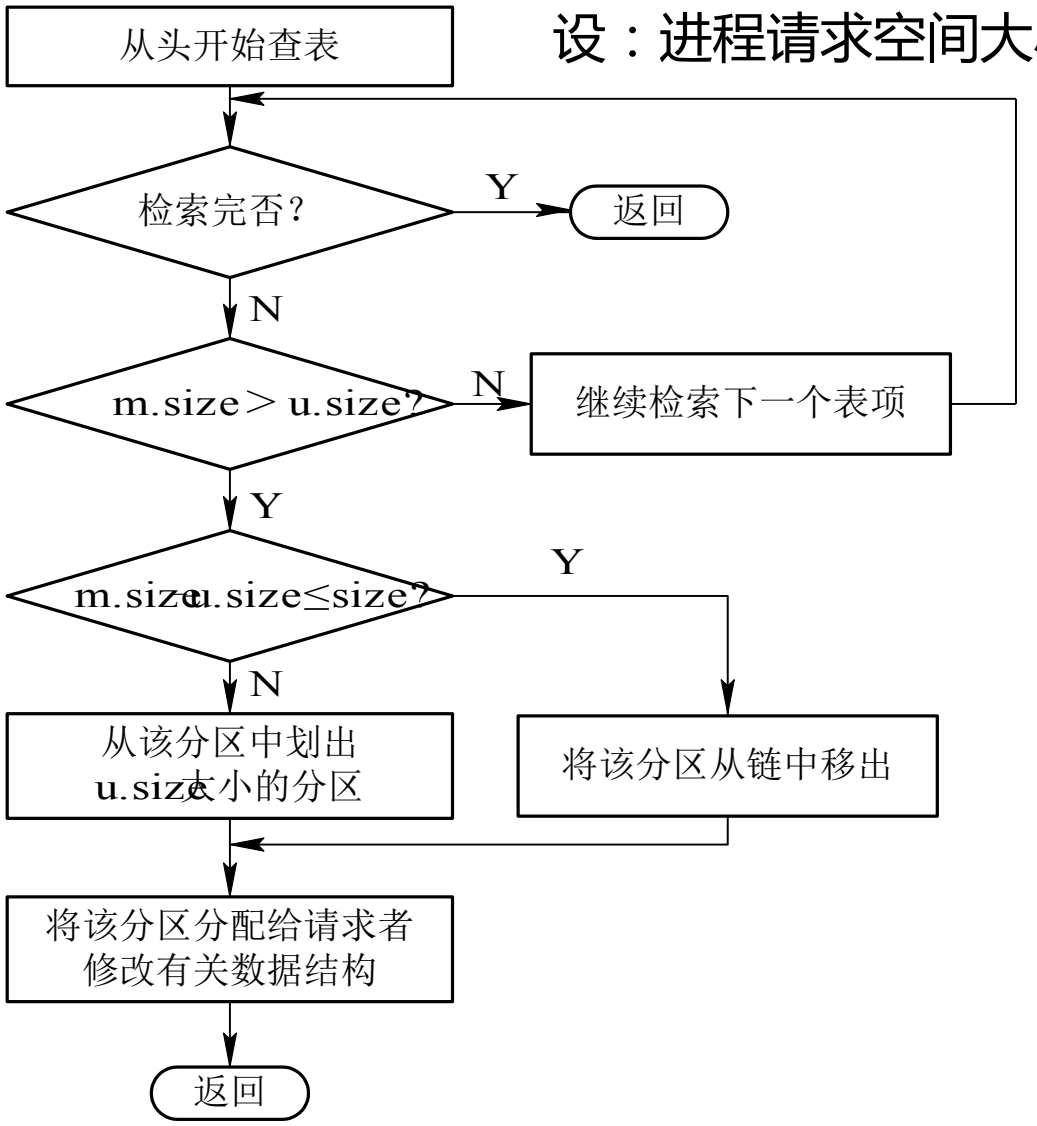

从头开始查表
设：进程请求空间大小为u.size
检索完否？
Y
返回
N
m.size＞u.size?
N
继续检索下一个表项
Y
m.size-u.size≤size?
Y
N
从该分区中划出
u.size大小的分区
将该分区从链中移出
将该分区分配给请求者
修改有关数据结构
返回

图 4-7　内存分配流程

4.3.3 动态分区分配

1) 首次适应算法(first fit)

我们以空闲分区链为例来说明采用FF算法时的分配情况。FF算法要求空闲分区链以地址递增的次序链接。在分配内存时，从链首开始顺序查找，直至找到一个大小能满足要求的空闲分区为止；然后再按照作业的大小，从该分区中划出一块内存空间分配给请求者，余下的空闲分区仍留在空闲链中。若从链首直至链尾都不能找到一个能满足要求的分区，则此次内存分配失败，返回。

该算法倾向于优先利用内存中低址部分的空闲分区，从而保留了高址部分的大空闲区。这给为以后到达的大作业分配大的内存空间创造了条件。

其缺点是低址部分不断被划分，会留下许多难以利用的、很小的空闲分区，而每次查找又都是从低址部分开始，这无疑会增加查找可用空闲分区时的开销。

4.3.3 动态分区分配

2) 循环首次适应算法(next fit)

该算法是由首次适应算法演变而成的。在为进程分配内存空间时，不再是每次都从链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，直至找到一个能满足要求的空闲分区，从中划出一块与请求大小相等的内存空间分配给作业。

为实现该算法，应设置一起始查寻指针，用于指示下一次起始查寻的空闲分区，并采用循环查找方式，即如果最后一个(链尾)空闲分区的大小仍不能满足要求，则应返回到第一个空闲分区，比较其大小是否满足要求。找到后，应调整起始查寻指针。

该算法能使内存中的空闲分区分布得更均匀，从而减少了查找空闲分区时的开销，但这样会缺乏大的空闲分区。

4.3.3 动态分区分配

3) 最佳适应算法(best fit)

所谓“最佳”是指每次为作业分配内存时，总是把能满足要求、又是最小的空闲分区分配给作业，避免“大材小用”。为了加速寻找，该算法要求将所有的空闲分区按其容量以从小到大的顺序形成一空闲分区链。

这样，第一次找到的能满足要求的空闲区，必然是最佳的。孤立地看，最佳适应算法似乎是最佳的，然而在宏观上却不一定。因为每次分配后所切割下来的剩余部分总是最小的，这样，在存储器中会留下许多难以利用的小空闲区。

4.3.3 动态分区分配

4) 最坏适应算法(worst fit)

最坏适应分配算法要扫描整个空闲分区表或链表，总是挑选一个最大的空闲区分割给作业使用，其优点是可使剩下的空闲区不至于太小，产生碎片的几率最小，对中、小作业有利，同时最坏适应分配算法查找效率很高。该算法要求将所有的空闲分区按其容量以从大到小的顺序形成一空闲分区链，查找时只要看第一个分区能否满足作业要求。

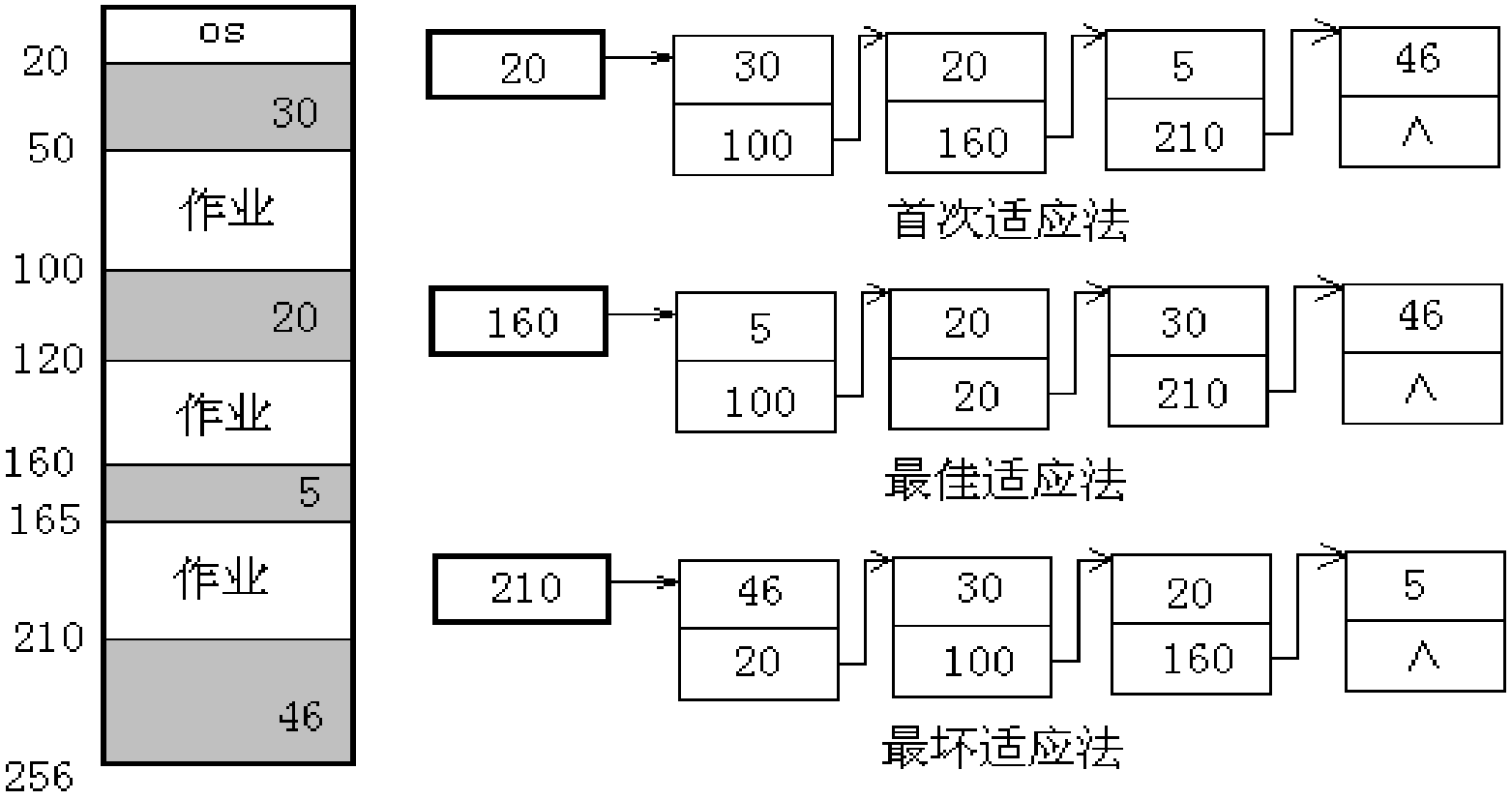
但是该算法的缺点也是明显的，它会使存储器中缺乏大的空闲分区。最坏适应算法与前面所述的首次适应算法、循环首次适应算法、最佳适应算法一起，也称为顺序搜索法。

三种主要内存分配算法及特点

- 首次适应算法(first-fit)：按分区的先后次序，从头查找，找到符合要求的第一个分区
  - 实现简单，该算法的分配和释放的时间性能较好，但随着低端分区不断划分而产生较多小分区，每次分配时查找时间开销会增大。
  - 较大的空闲分区可以被保留在内存高端。
- 最佳适应算法(best-fit)：找到其大小与要求相差最小的空闲分区
  - 实现复杂
  - 从个别来看，外碎片较小，但从整体来看，会形成较多外碎片。
  - 较大的空闲分区可以被保留以满足大程序
- 最坏适应算法(worst-fit)：找到最大的空闲分区进行分配
  - 实现复杂
  - 分割后剩下的比较大，可再用。
  - 大分区越来越少，可能会出现没有大分区满足大程序

三种分配算法例

例：有作业序列：作业A要求18K；作业B要求25K，作业C要求30K。系统中空闲区按三种算法组成的空闲区链如下，经分析可知：最佳适应法对这个作业序列是合适的，而其它两种对该作业序列是不合适的。



4.3.3 动态分区分配

3.分区回收算法

- 分区回收算法：需要将相邻的空闲分区合并成一个空闲分区。这时要解决的问题是：合并时机的选择和合并条件的判断。

➢ 合并时机：当分区被进程释放时。

➢ 合并条件 ： 1.上接空闲区

2.下接空闲区

3.上、下都接空闲区

4.上、下都不接空闲区

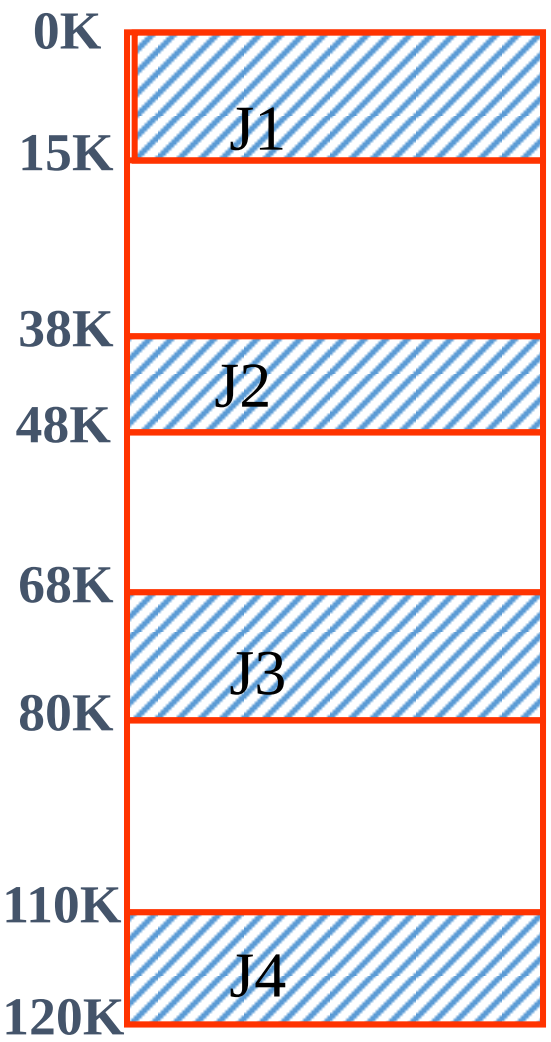
➢根据不同合并条件，有不同的合并策略。

4.3.3 动态分区分配

回收算法的实现

- 上邻空闲区。与上空闲区合并，修改其长度。
- 下邻空闲区。与下空闲区合并，修改其长度和首址。
- 上、下邻空闲区。与上、下空闲区合并，修改其长度和首址，从空闲区表中或空闲链表中删除下接的空闲区表项。
- 上下都不相邻空闲区时，在空闲区表中或空闲链表中添加新项，添上被回收分区长度和首址。

回收算法例1



J1结束？

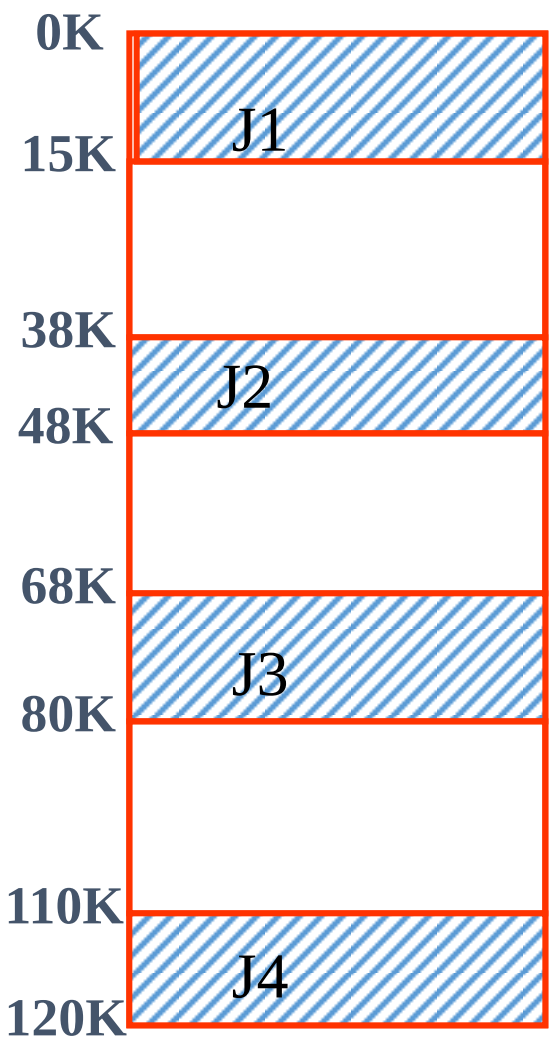
空闲区表

始址	长度	标志
0 K	38K	未分配
48K	20K	未分配
80K	30K	未分配
		空
		空

已分配表

始址	长度	标志
38K	10K	J2
68K	12K	J3
110K	10K	J4
		空
		空

回收算法例2



J2结束？

空闲区表

始址	长度	标志
15K	53K	未分配
80K	30K	未分配
		空
		空

已分配表

始址	长度	标志
0K	15K	J1
68K	12K	J3
110K	10K	J4
		空
		空

4.3.6 可重定位分区分配

1．动态重定位的引入

在连续分配方式中，必须把一个系统或用户程序装入一连续的内存空间。如果在系统中只有若干个小的分区，即使它们容量的总和大于要装入的程序，但由于这些分区不相邻接，也无法把该程序装入内存。

例如，图4-9(a)中示出了在内存中现有四个互不邻接的小分区，它们的容量分别为10 KB、30 KB、14 KB和26 KB，其总容量是80 KB。但如果现在有一作业到达，要求获得40 KB的内存空间，由于必须为它分配一连续空间，故此作业无法装入。这种不能被利用的小分区称为“零头”或“碎片”。

4.3.6 可重定位分区分配

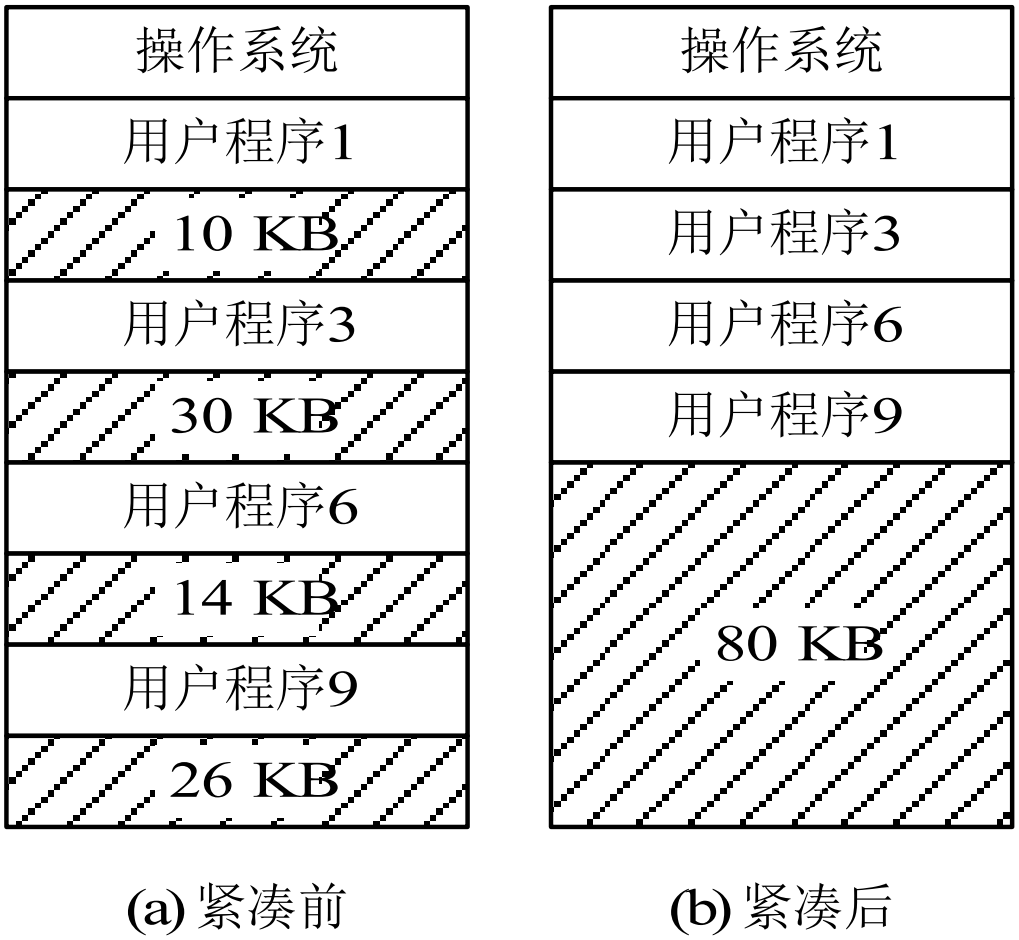


图4- 9 紧凑的示意

4.3.6 可重定位分区分配

若想把作业装入，可采用的一种方法是：将内存中的所有作业进行移动，使它们全都相邻接，这样，即可把原来分散的多个小分区拼接成一个大分区，这时就可把作业装入该区。这种通过移动内存中作业的位置，以把原来多个分散的小分区拼接成一个大分区的方法，称为“拼接”或“紧凑”，见图4-9(b)。

由于经过紧凑后的某些用户程序在内存中的位置发生了变化，此时若不对程序和数据的地址加以修改(变换)，则程序必将无法执行。为此，在每次“紧凑”后，都必须对移动了的程序或数据进行重定位。

4.3.6　可重定位分区分配

3．动态重定位分区分配算法

动态重定位分区分配算法与动态分区分配算法基本上相同，差别仅在于：在这种分配算法中，增加了紧凑的功能，通常，在找不到足够大的空闲分区来满足用户需求时进行紧凑。图4-11示出了动态重定位分区分配算法。

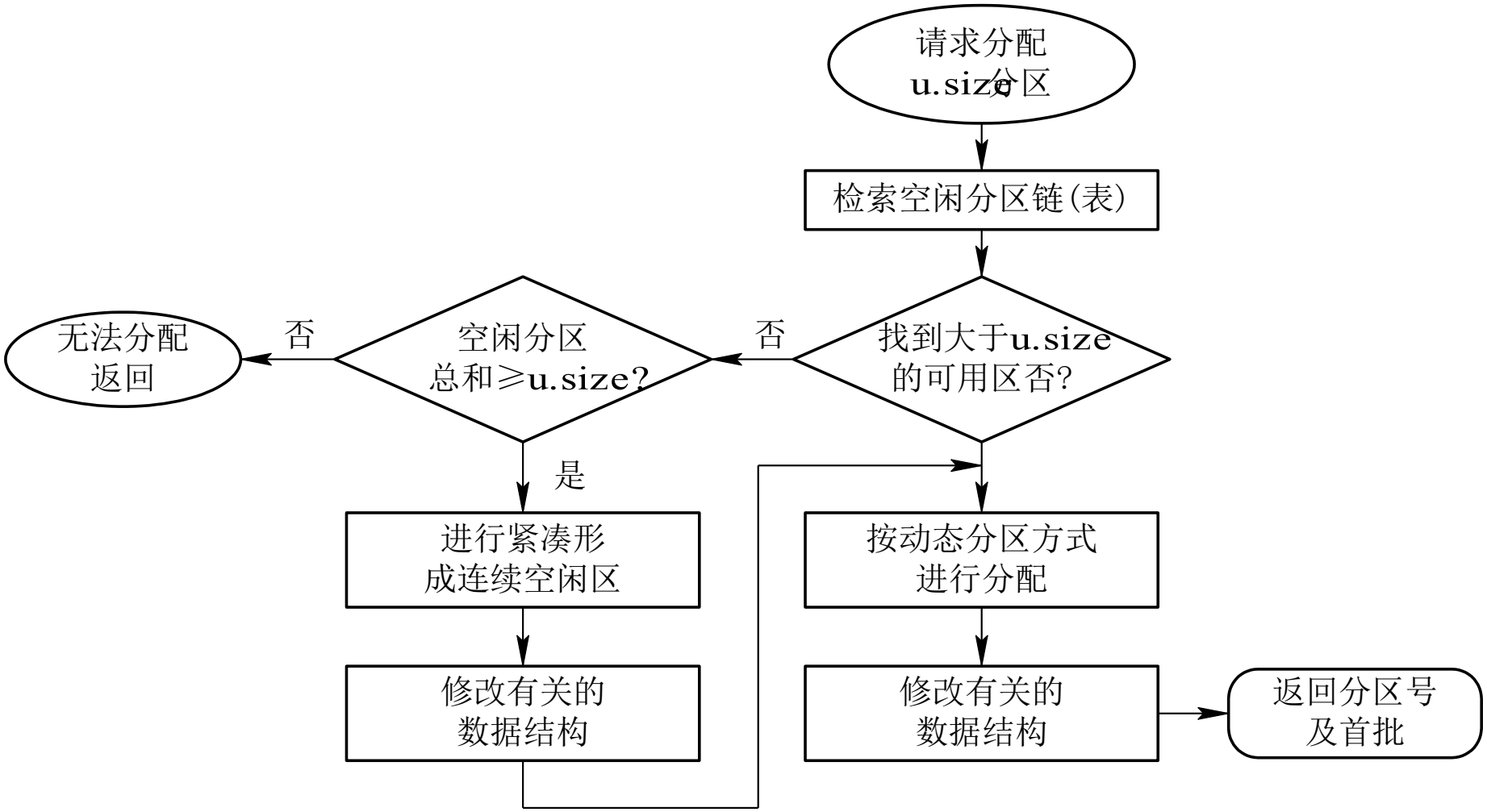


图4-11　动态分区分配算法流程图

4.3.7 对换

1．对换(Swapping)的引入

在多道程序环境下，一方面，在内存中的某些进程由于某事件尚未发生而被阻塞运行，但它却占用了大量的内存空间，甚至有时可能出现在内存中所有进程都被阻塞而迫使CPU停止下来等待的情况；另一方面，却又有着许多作业在外存上等待，因无内存而不能进入内存运行的情况。

为了解决这一问题，在系统中又增设了对换(也称交换)设施。所谓“对换”，是指把内存中暂时不能运行的进程或者暂时不用的程序和数据调出到外存上，以便腾出足够的内存空间，再把已具备运行条件的进程或进程所需要的程序和数据调入内存。对换是提高内存利用率的有效措施。

4.3.7 对换

如果对换是以整个进程为单位的，便称之为“整体对换”或“进程对换”。这种对换被广泛地应用于分时系统中，其目的是用来解决内存紧张问题，并可进一步提高内存的利用率。而如果对换是以“页”或“段”为单位进行的，则分别称之为“页面对换”或“分段对换”，又统称为“部分对换”。这种对换方法是实现后面要讲到的请求分页和请求分段式存储管理的基础，其目的是为了支持虚拟存储系统。

为了实现进程对换，系统必须能实现三方面的功能：

- 对换空间的管理
- 进程的换出
- 进程的换入

4.3.7 对换

2．对换空间的管理

在具有对换功能的OS中，通常把外存分为文件区和对换区。

前者用于存放文件，后者用于存放从内存换出的进程。由于通常的文件都是较长久地驻留在外存上，故对文件区管理的主要目标，是提高文件存储空间的利用率，为此，对文件区采取离散分配方式。

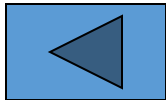
然而，进程在对换区中驻留的时间是短暂的，对换操作又较频繁，故对对换空间管理的主要目标，是提高进程换入和换出的速度。为此，采取的是连续分配方式，较少考虑外存中的碎片问题。

4.3.7 对换

3．进程的换出与换入

(1) 进程的换出。每当一进程由于创建子进程而需要更多的内存空间，但又无足够的内存空间等情况发生时，系统应将某进程换出。其过程是：系统首先选择处于阻塞状态且优先级最低的进程作为换出进程，然后启动磁盘，将该进程的程序和数据传送到磁盘的对换区上。若传送过程未出现错误，便可回收该进程所占用的内存空间，并对该进程的进程控制块做相应的修改。

(2) 进程的换入。系统应定时地查看所有进程的状态，从中找出“就绪”状态但已换出的进程，将其中换出时间最久(换出到磁盘上)的进程作为换入进程，将之换入，直至已无可换入的进程或无可换出的进程为止。



4.4 基本分页存储管理方式

- 引入页式原因：

① 分区缺点：由于进程空间逻辑上连续，物理上也连续，因此带来内存利用率低、共享难、有碎片、无法实现虚存。

② 不连续技术可以避免移动、实现共享、实现虚存、避免外部碎片，关键是地址转换如何解决？

③ 页式、段式、段页式可以解决上述问题

4.4 基本分页存储管理方式

页式实现

1. 内存分为等长页面(frame)，称为块或页帧。
2. 进程的地址空间划分成等长页(page)，页与页面通常等长。
3. 程序在内存存放时，页内连续、页间不一定连续（即相邻页不一定放在相邻页面中）。

4.4.1 页面与页表

1．页面

1) 页面和物理块

分页存储管理是将一个进程的逻辑地址空间分成若干个大小相等的片，称为页面或页，并为各页加以编号，从0开始，如第0页、第1页等。相应地，也把内存空间分成与页面相同大小的若干个存储块，称为(物理)块或页框(frame)，也同样为它们加以编号，如0#块、1#块等等。

在为进程分配内存时，以块为单位将进程中的若干个页分别装入到多个可以不相邻接的物理块中。由于进程的最后一页经常装不满一块而形成了不可利用的碎片，称之为“页内碎片”。

4.4.1 页面与页表

2) 页面大小

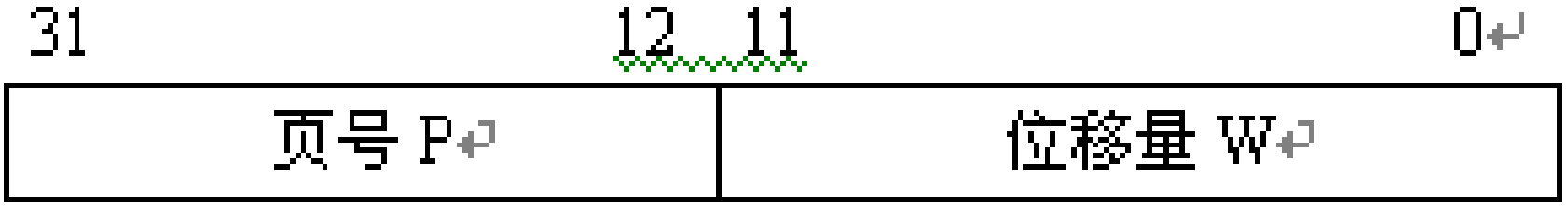
在分页系统中的页面其大小应适中。页面若太小，一方面虽然可使内存碎片减小，从而减少了内存碎片的总空间，有利于提高内存利用率，但另一方面也会使每个进程占用较多的页面，从而导致进程的页表过长，占用大量内存；此外，还会降低页面换进换出的效率。

然而，如果选择的页面较大，虽然可以减少页表的长度，提高页面换进换出的速度，但却又会使页内碎片增大。因此，页面的大小应选择适中，且页面大小应是2的幂，通常为512 B～8 KB。

4.4.1 页面与页表

2．地址结构

分页地址中的地址结构如下：



它含有两部分：前一部分为页号P，后一部分为位移量W(或称为页内地址)。图中的地址长度为32位，其中0～11位为页内地址，即每页的大小为4 KB；12～31位为页号，地址空间最多允许有1 M页。

4.4.1 页面与页表

对于某特定机器，其地址结构是一定的。若给定一个逻辑地址空间中的地址为A，页面的大小为L，则页号P和页内地址d可按下式求得：

$$P = \mathrm{INT}\left[\frac{A}{L}\right]$$

$$d = [A]\mathrm{MOD}L$$

其中，INT是整除函数，MOD是取余函数。例如，其系统的页面大小为1 KB，设A = 2170 B，则由上式可以求得P = 2，d = 122。

4.4.1 页面与页表

3．页表

在分页系统中，允许将进程的各个页离散地存储在内存不同的物理块中，但系统应能保证进程的正确运行，即能在内存中找到每个页面所对应的物理块。为此，系统又为每个进程建立了一张页面映像表，简称页表。

在进程地址空间内的所有页(0～n)，依次在页表中有一页表项，其中记录了相应页在内存中对应的物理块号，见图4-12的中间部分。在配置了页表后，进程执行时，通过查找该表，即可找到每页在内存中的物理块号。可见，页表的作用是实现从页号到物理块号的地址映射。

4.4.1 页面与页表

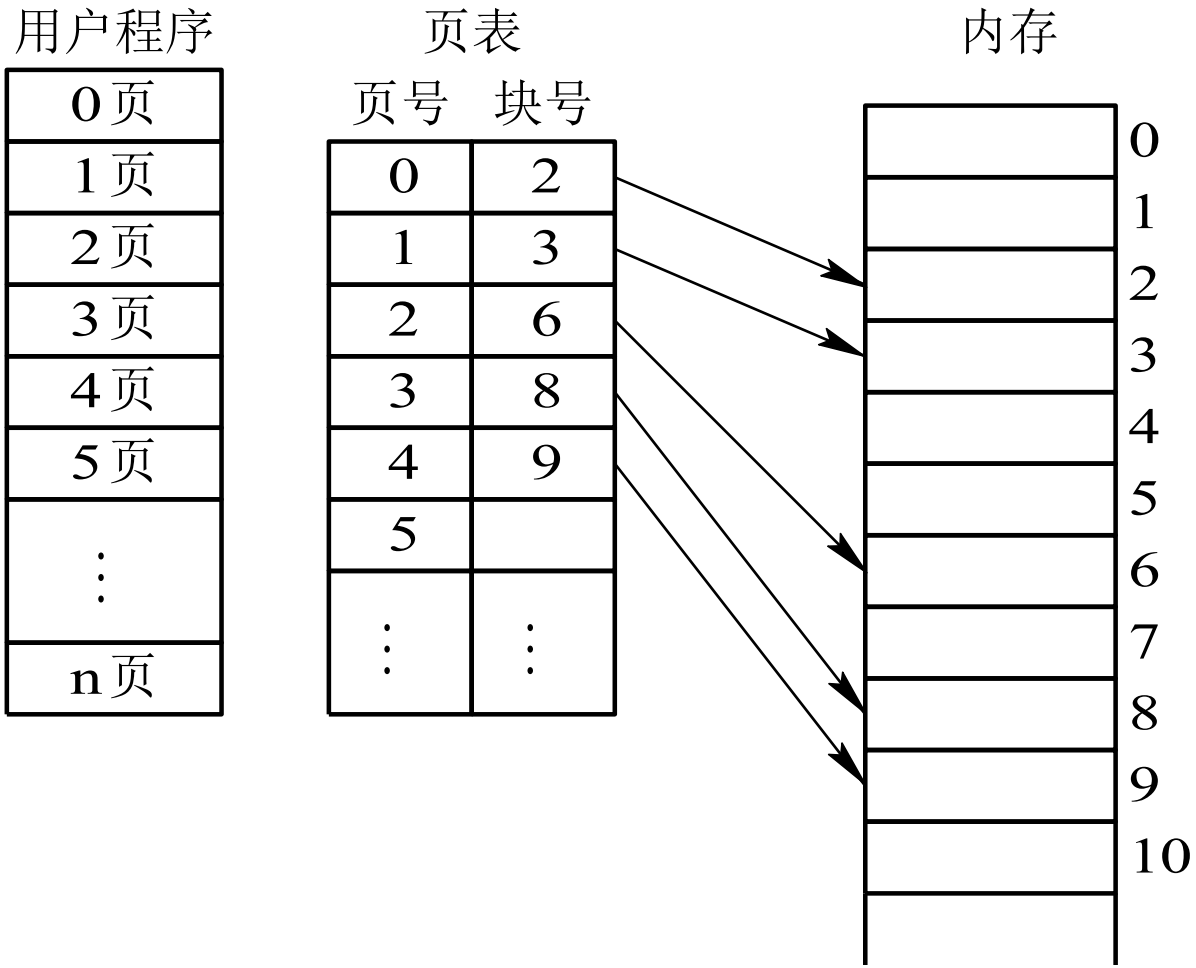


图4-12 页表的作用

4.4.2 地址变换机构

1．基本的地址变换机构

页表的功能可以由一组专门的寄存器来实现。一个页表项用一个寄存器。由于寄存器具有较高的访问速度，因而有利于提高地址变换的速度；但由于寄存器成本较高，且大多数现代计算机的页表又可能很大，使页表项的总数可达几千甚至几十万个，显然这些页表项不可能都用寄存器来实现，因此，页表大多驻留在内存中。

在系统中只设置一个页表寄存器PTR(Page-Table Register)，在其中存放页表在内存的始址和页表的长度。平时，进程未执行时，页表的始址和页表长度存放在本进程的PCB中。当调度程序调度到某进程时，才将这两个数据装入页表寄存器中。

4.4.2 地址变换机构

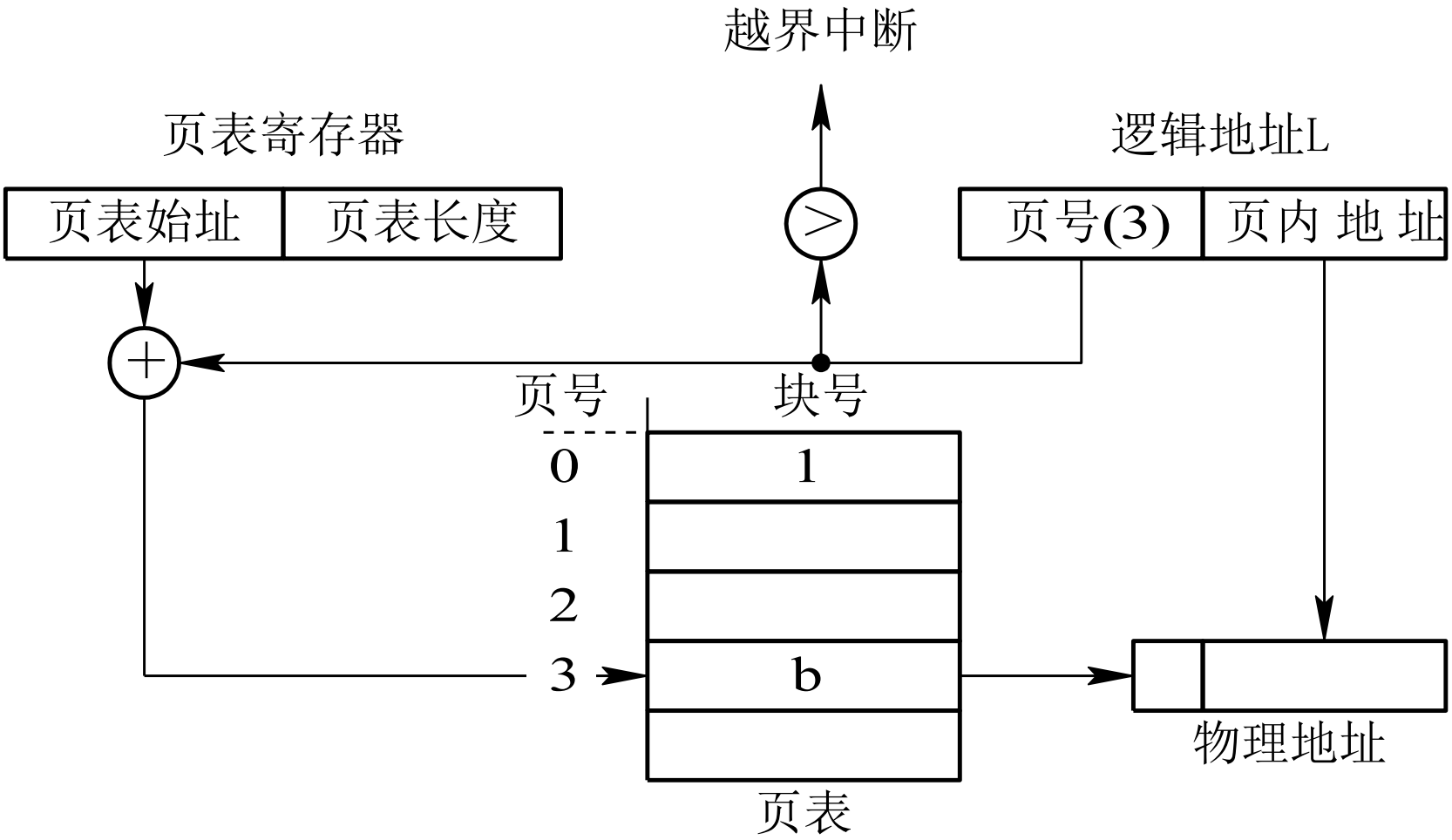


图4-13 分页系统的地址变换机构

4.4.2 地址变换机构

2．具有快表的地址变换机构

由于页表是存放在内存中的，这使CPU在每存取一个数据时，都要两次访问内存。第一次是访问内存中的页表，从中找到指定页的物理块号，再将块号与页内偏移量W拼接，以形成物理地址。第二次访问内存时，才是从第一次所得地址中获得所需数据(或向此地址中写入数据)。

因此，采用这种方式将使计算机的处理速度降低近1/2。可见，以此高昂代价来换取存储器空间利用率的提高，是得不偿失的。

4.4.2 地址变换机构

为了提高地址变换速度，可在地址变换机构中增设一个具有并行查寻能力的特殊高速缓冲寄存器，又称为“联想寄存器”(Associative Memory)，或称为“快表”，用以存放当前访问的那些页表项。

此时的地址变换过程是：在CPU给出有效地址后，由地址变换机构自动地将页号P送入高速缓冲寄存器，并将此页号与高速缓存中的所有页号进行比较，若其中有与此相匹配的页号，便表示所要访问的页表项在快表中。于是，可直接从快表中读出该页所对应的物理块号，并送到物理地址寄存器中。

如在块表中未找到对应的页表项，则还须再访问内存中的页表，找到后，把从页表项中读出的物理块号送地址寄存器；同时，再将此页表项存入快表的一个寄存器单元中。但如果联想寄存器已满，则OS必须找到一个老的且已被认为不再需要的页表项，将它换出。

4.4.2 地址变换机构

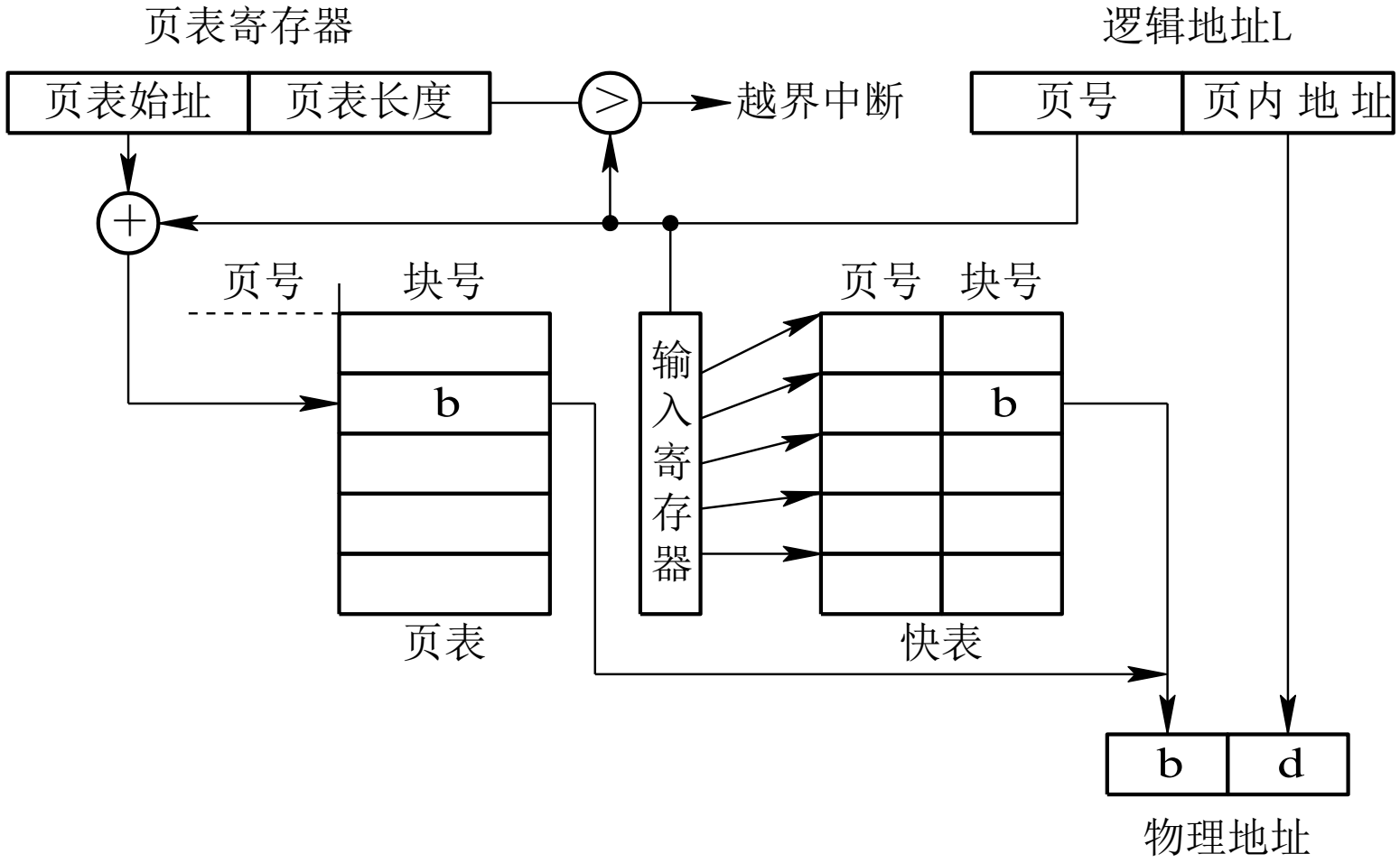


图4-14 具有快表的地址变换机构

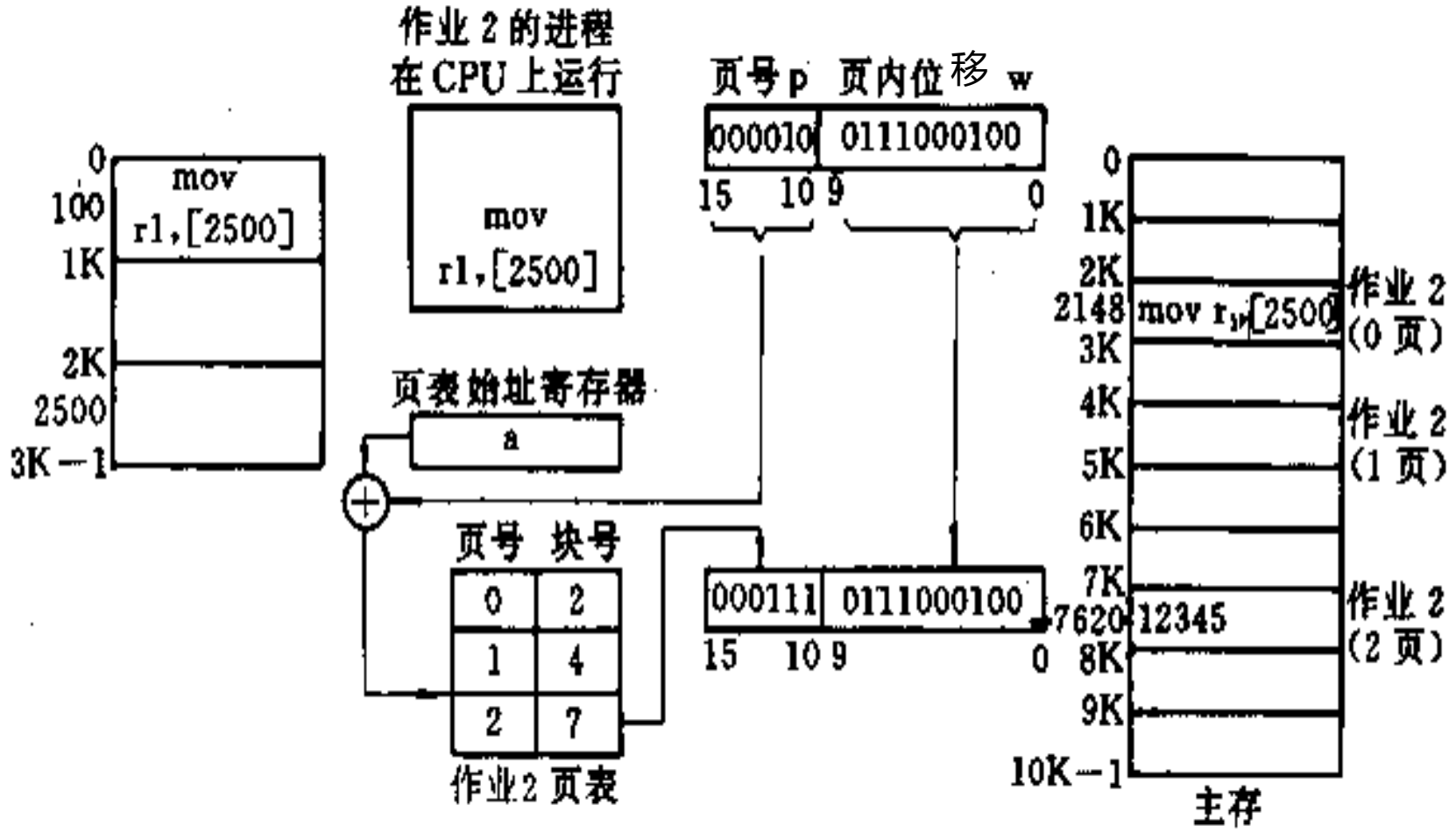
4.4.2 地址变换机构

二进制地址例（2500-->7620)

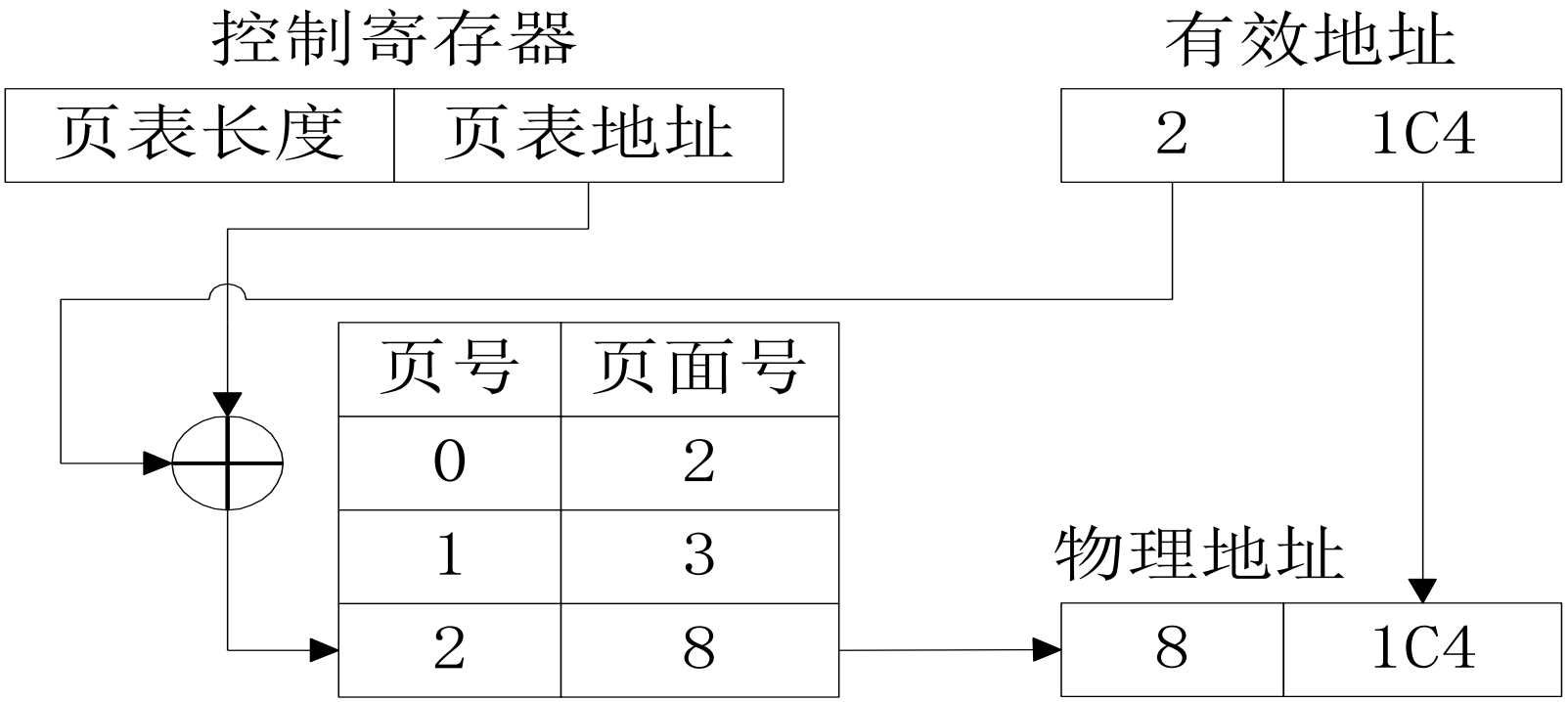
- 设页大小为2^{10}，地址长度为16位，
  则虚存大小为 2^{16} = 64K
  地址范围：0 - （ 2^{16} - 1）
- 逻辑地址 2500 = 0000,1001,1100,0100
- 页号为2，若对应页面号为7，则：
- 物理地址 7620 = 0001,1101,1100,0100

4.4.2 地址变换机构

例：设页长为1K，程序地址字长为16位，作业2空间和页表如下图所示，则逻辑地址2500转换物理地址7620示意如图：



地址转换例（21C4→81C4)



① 逻辑地址21C4分为页号2和页内位移1C4

② 根据寄存器所指页表地址与页号2找到对应的页面号8

③ 将8与页内位移1C4合并成物理地址81C4

4.4.3 两级和多级页表

现代的大多数计算机系统，都支持非常大的逻辑地址空间($2^{32} \sim 2^{64}$)。在这样的环境下，页表就变得非常大，要占用相当大的内存空间。例如，对于一个具有32位逻辑地址空间的分页系统，规定页面大小为4 KB即2^{12} B，则在每个进程页表中的页表项可达1兆个之多。又因为每个页表项占用一个字节，故每个进程仅仅其页表就要占用1 MB的内存空间，而且还要求是连续的。显然这是不现实的，我们可以采用下述两个方法来解决这一问题：

(1) 采用离散分配方式来解决难以找到一块连续的大内存空间的问题；

(2) 只将当前需要的部分页表项调入内存，其余的页表项仍驻留在磁盘上，需要时再调入。

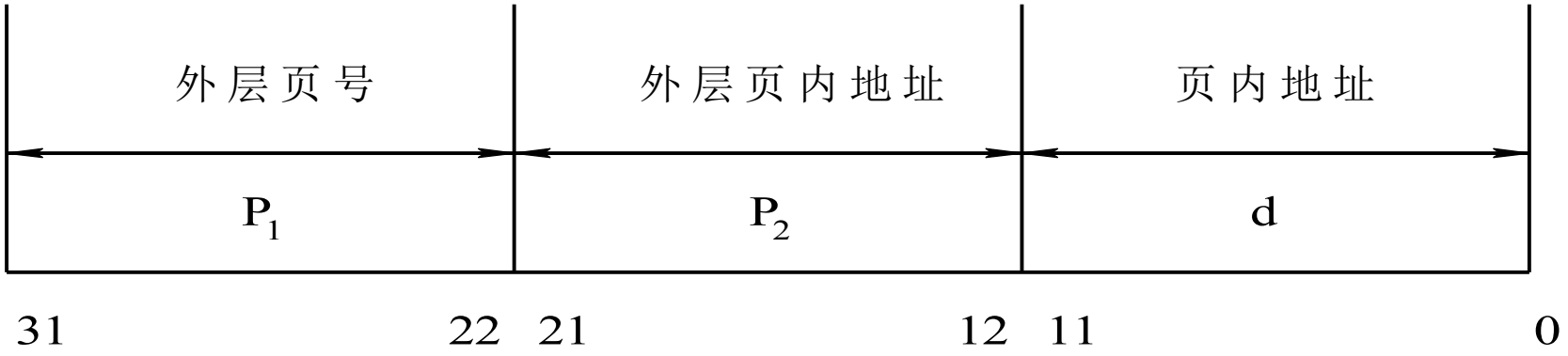
4.4.3 两级和多级页表

1．两级页表(Two-Level Page Table)

对于要求连续的内存空间来存放页表的问题，可利用将页表进行分页，并离散地将各个页面分别存放在不同的物理块中的办法来加以解决，同样也要为离散分配的页表再建立一张页表，称为外层页表(Outer Page Table)，在每个页表项中记录了页表页面的物理块号。

下面我们仍以前面的32位逻辑地址空间为例来说明。当页面大小为4 KB时(12位)，若采用一级页表结构，应具有20位的页号，即页表项应有1兆个；在采用两级页表结构时，再对页表进行分页，使每页中包含2^{10} (即1024)个页表项，最多允许有2^{10}个页表分页；或者说，外层页表中的外层页内地址P2为10位，外层页号P1也为10位。此时的逻辑地址结构可描述如下：

4.4.3 两级和多级页表



4.4.3 两级和多级页表

由图可以看出，在页表的每个表项中存放的是进程的某页在内存中的物理块号，如第0#页存放在1#物理块中；1#页存放在4#物理块中。而在外层页表的每个页表项中，所存放的是某页表分页的首址，如第0#页表是存放在第1011#物理块中。我们可以利用外层页表和页表这两级页表，来实现从进程的逻辑地址到内存中物理地址间的变换。

4.4.3 两级和多级页表

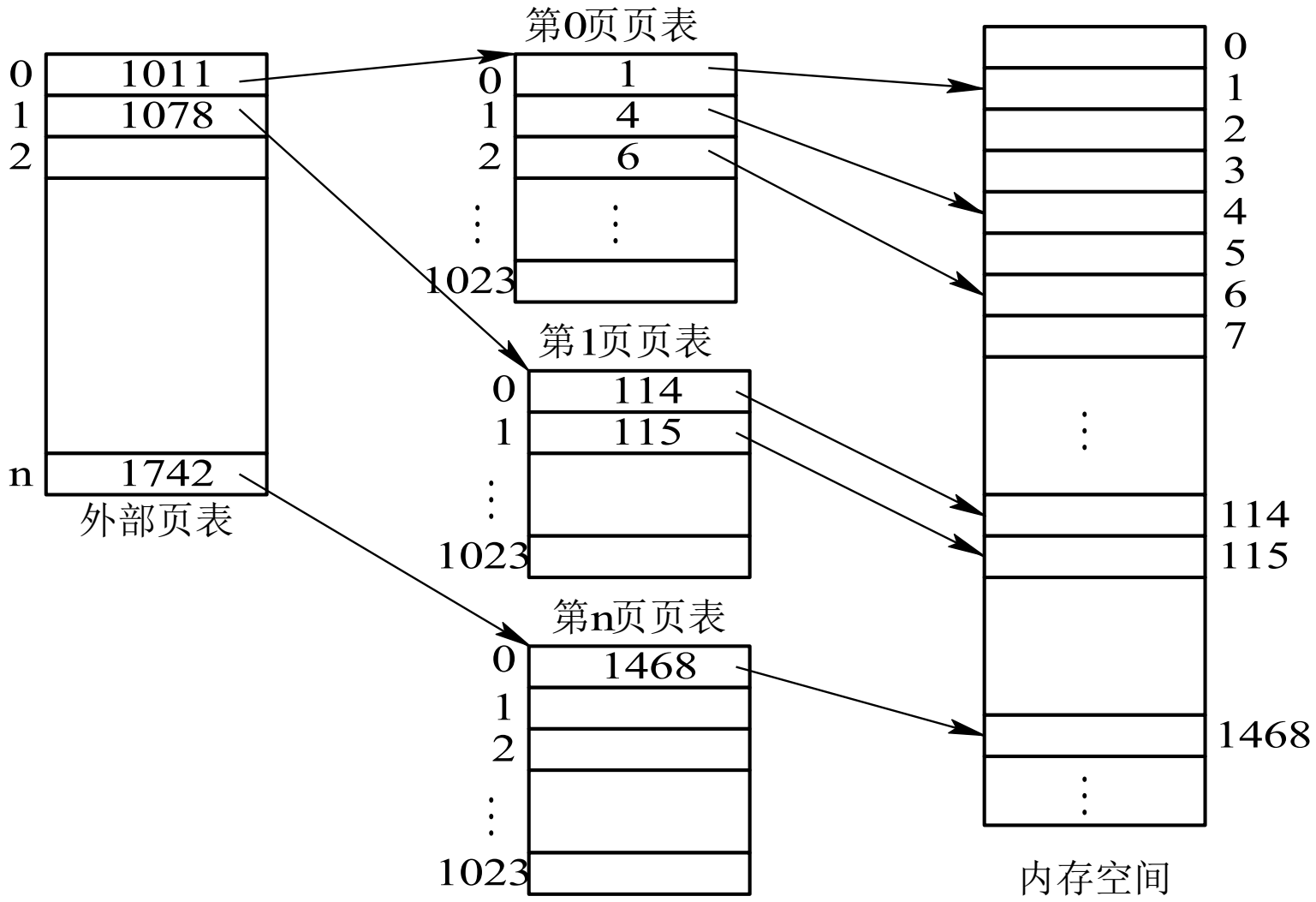


图4-15 两级页表结构

4.4.3 两级和多级页表

为了地址变换实现上的方便起见，在地址变换机构中同样需要增设一个外层页表寄存器，用于存放外层页表的始址，并利用逻辑地址中的外层页号，作为外层页表的索引，从中找到指定页表分页的始址，再利用P2作为指定页表分页的索引，找到指定的页表项，其中即含有该页在内存的物理块号，用该块号和页内地址d即可构成访问的内存物理地址。图4-16示出了两级页表时的地址变换机构。

4.4.3 两级和多级页表

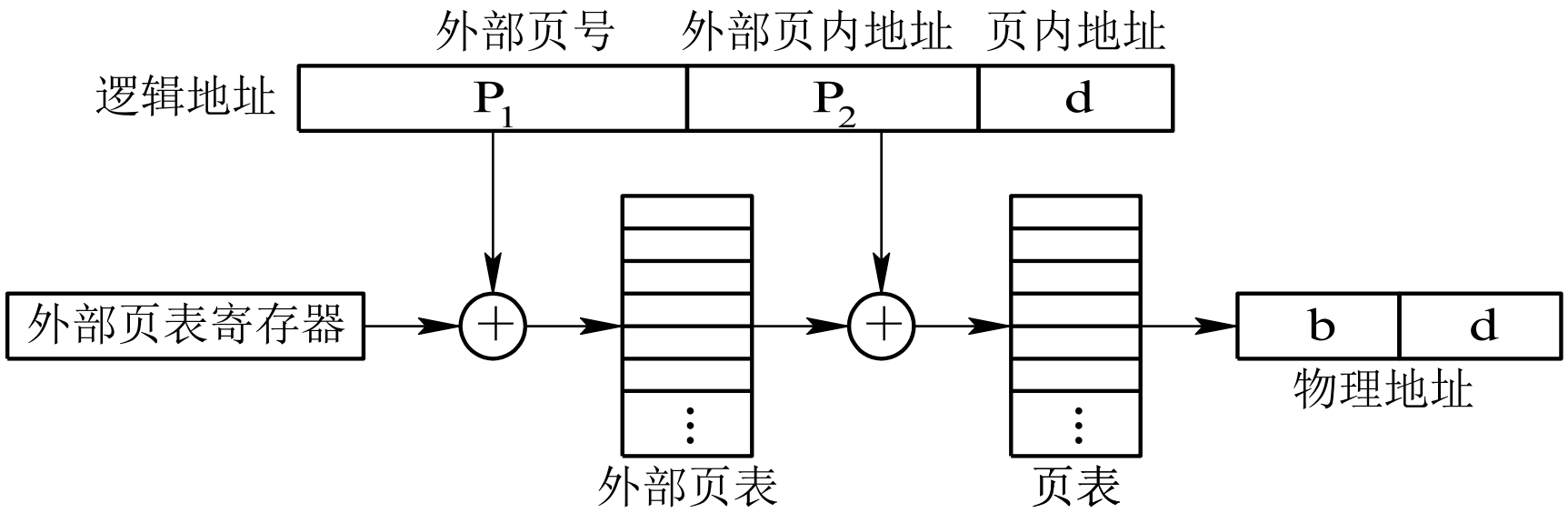


图4-16 具有两级页表的地址变换机构

4.4.3 两级和多级页表

2．多级页表

对于32位的机器，采用两级页表结构是合适的；但对于64位的机器，采用两级页表是否仍可适用的问题，须做以下简单分析。如果页面大小仍采用4 KB即2^{12} B，那么还剩下52位，假定仍按物理块的大小(2^{12}位)来划分页表，则将余下的42位用于外层页号。此时在外层页表中可能有4096 G个页表项，要占用16 384 GB的连续内存空间。

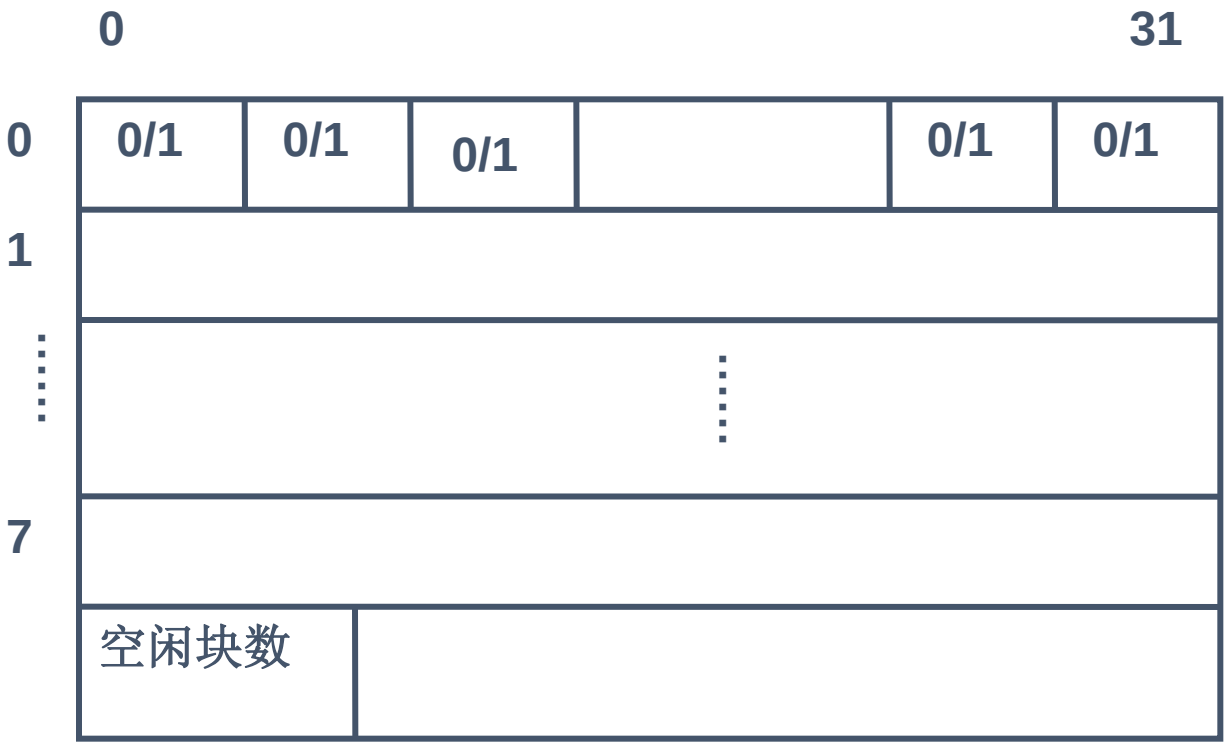
这样的结果显然是不能令人接受的，因此必须采用多级页表，将外层页表再进行分页，也就是将各分页离散地装入到不相邻接的物理块中，再利用第2级的外层页表来映射它们之间的关系。

4.4.3 两级和多级页表

对于64位的计算机，如果要求它能支持2^{64} B(= 1 844 744 TB)规模的物理存储空间，则即使是采用三级页表结构也是难以办到的；而在当前的实际应用中也无此必要。故在近两年推出的64位OS中，把可直接寻址的存储器空间减少为45位长度(即2^{45})左右，这样便可利用三级页表结构来实现分页存储管理。

4.4.3 两级和多级页表

页式管理的分配与回收（位示图）



每位表示一个页面，0表示空闲，1表示已分配

4.4.3 两级和多级页表

页式管理的分配与回收（续）

① 计算一个进程所需要的总页数N；

② 查位示图，看看是否还有N个空闲页面；

③ 如果有足够的空闲页面，则页表长度设为N，申请页表区，把页表始址填入PCB；

④ 依次分配N个空闲页面，将块号和页号填入页表；

⑤ 修改位示图。

4.4.3 两级和多级页表

页式特点

（1）优点：

- 分配与回收简单
- 消除“碎片”问题
- 可实现虚存、共享

（2）缺点：

- 页面划分不考虑程序的逻辑结构
- 共享受限
- 二次访内，速度慢

4.5 基本分段存储管理方式

1) 方便编程

通常，用户把自己的作业按照逻辑关系划分为若干个段，每个段都是从0开始编址，并有自己的名字和长度。因此，希望要访问的逻辑地址是由段名(段号)和段内偏移量(段内地址)决定的。例如，下述的两条指令便是使用段名和段内地址：

LOAD R1，[A]|〈D〉；

STORE R1，[B]|〈C〉；

其中，前一条指令的含义是将分段A中D单元内的值读入寄存器1；后一条指令的含义是将寄存器1的内容存入B分段的C单元中。

4.5.1 分段存储管理方式的引入

2) 信息共享

在实现对程序和数据的共享时，是以信息的逻辑单位为基础的。比如，共享某个例程和函数。分页系统中的“页”只是存放信息的物理单位(块)，并无完整的意义，不便于实现共享；然而段却是信息的逻辑单位。

由此可知，为了实现段的共享，希望存储管理能与用户程序分段的组织方式相适应。

4.5.1 分段存储管理方式的引入

3) 信息保护

信息保护同样是对信息的逻辑单位进行保护，因此，分段管理方式能更有效和方便地实现信息保护功能。

4) 动态增长

在实际应用中，往往有些段，特别是数据段，在使用过程中会不断地增长，而事先又无法确切地知道数据段会增长到多大。前述的其它几种存储管理方式，都难以应付这种动态增长的情况，而分段存储管理方式却能较好地解决这一问题。

4.5.1 分段存储管理方式的引入

5) 动态链接

动态链接是指在作业运行之前，并不把几个目标程序段链接起来。要运行时，先将主程序所对应的目标程序装入内存并启动运行，当运行过程中又需要调用某段时，才将该段(目标程序)调入内存并进行链接。可见，动态链接也要求以段作为管理的单位。

4.5.2 分段系统的基本原理

1.将用户程序空间按逻辑划分为几段（segment），每个段内连续编址，每个程序段都有一个段名，且有一个段号。段号从0开始，每一段段内也从0开始编址，段内地址是连续的段间是不一定连续编址的

2. 以段为单位分配内存，段内连续完整，但各段之间可以不连续存放

3.段式逻辑地址二维:<段号><段内位移>

4.每个进程一张段表，实现地址转换、信息保护、共享、扩充、动态连接等

4.5.2 分段系统的基本原理

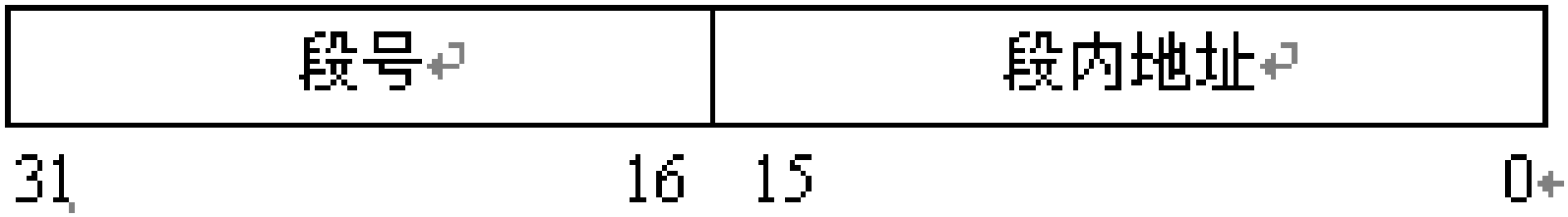
1. 分段

在分段存储管理方式中，作业的地址空间被划分为若干个段，每个段定义了一组逻辑信息。例如，有主程序段MAIN、子程序段X、数据段D及栈段S等，如图4-17所示。

每个段都有自己的名字。为了实现简单起见，通常可用一个段号来代替段名，每个段都从0开始编址，并采用一段连续的地址空间。段的长度由相应的逻辑信息组的长度决定，因而各段长度不等。整个作业的地址空间由于是分成多个段，因而是二维的，亦即，其逻辑地址由段号(段名)和段内地址所组成。

4.5.2 分段系统的基本原理

分段地址中的地址具有如下结构：



4.5.2 分段系统的基本原理

2．段表

在前面所介绍的动态分区分配方式中，系统为整个进程分配一个连续的内存空间。而在分段式存储管理系统中，则是为每个分段分配一个连续的分区，而进程中的各个段可以离散地移入内存中不同的分区中。

为使程序能正常运行，亦即，能从物理内存中找出每个逻辑段所对应的位置，应像分页系统那样，在系统中为每个进程建立一张段映射表，简称“段表”。每个段在表中占有一个表项，其中记录了该段在内存中的起始地址(又称为“基址”)和段的长度，如图4-17 所示。段表可以存放在一组寄存器中，这样有利于提高地址转换速度，但更常见的是将段表放在内存中。

4.5.2 分段系统的基本原理

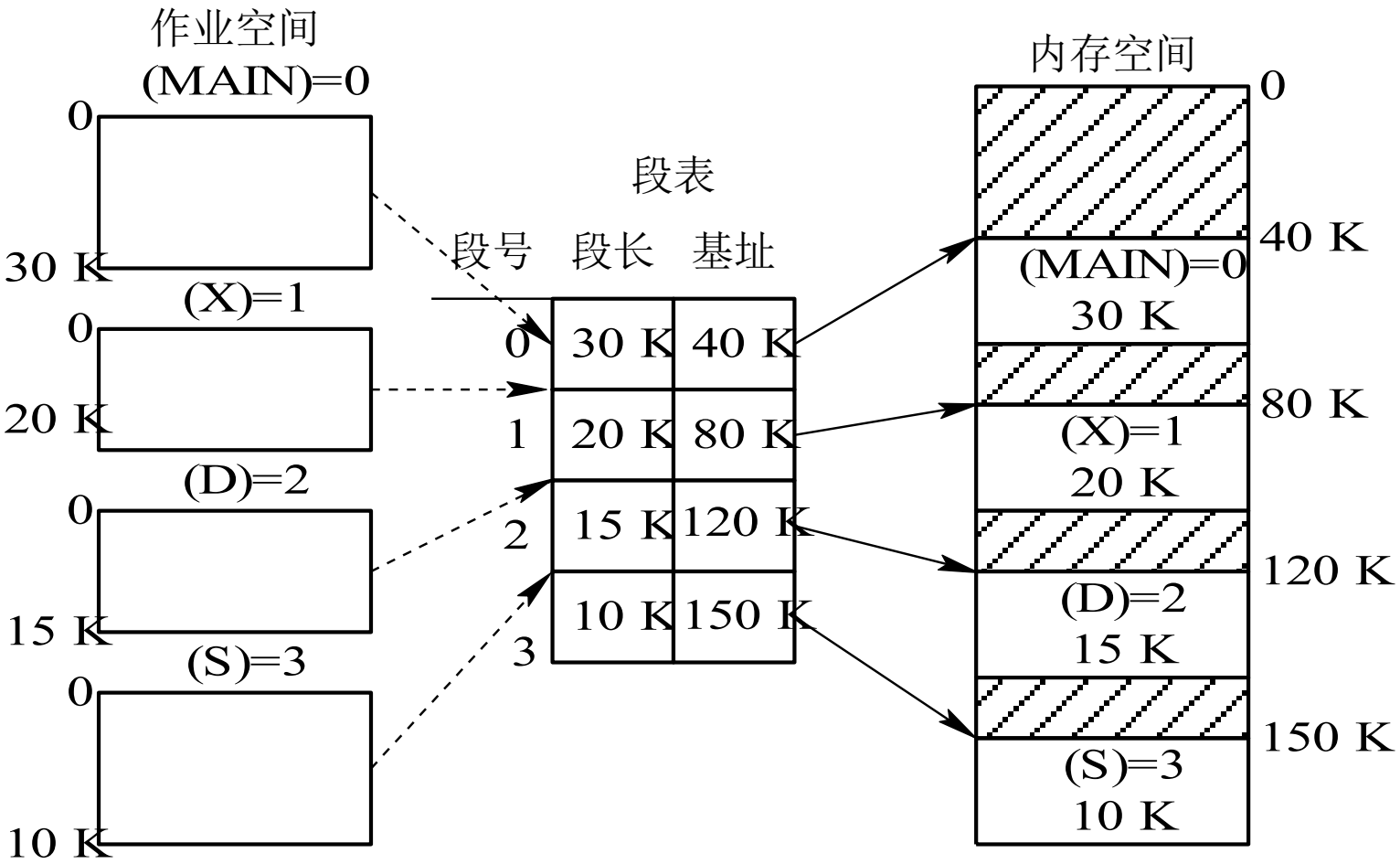


图4-17 利用段表实现地址映射

4.5.2 分段系统的基本原理

3．地址变换机构

为了实现从进程的逻辑地址到物理地址的变换功能，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TL。在进行地址变换时，系统将逻辑地址中的段号与段表长度TL进行比较。

若S>TL，表示段号太大，是访问越界，于是产生越界中断信号；若未越界，则根据段表的始址和该段的段号，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d是否超过该段的段长SL。若超过，即d>SL，同样发出越界中断信号；若未越界，则将该段的基址d与段内地址相加，即可得到要访问的内存物理地址。

4.5.2 分段系统的基本原理

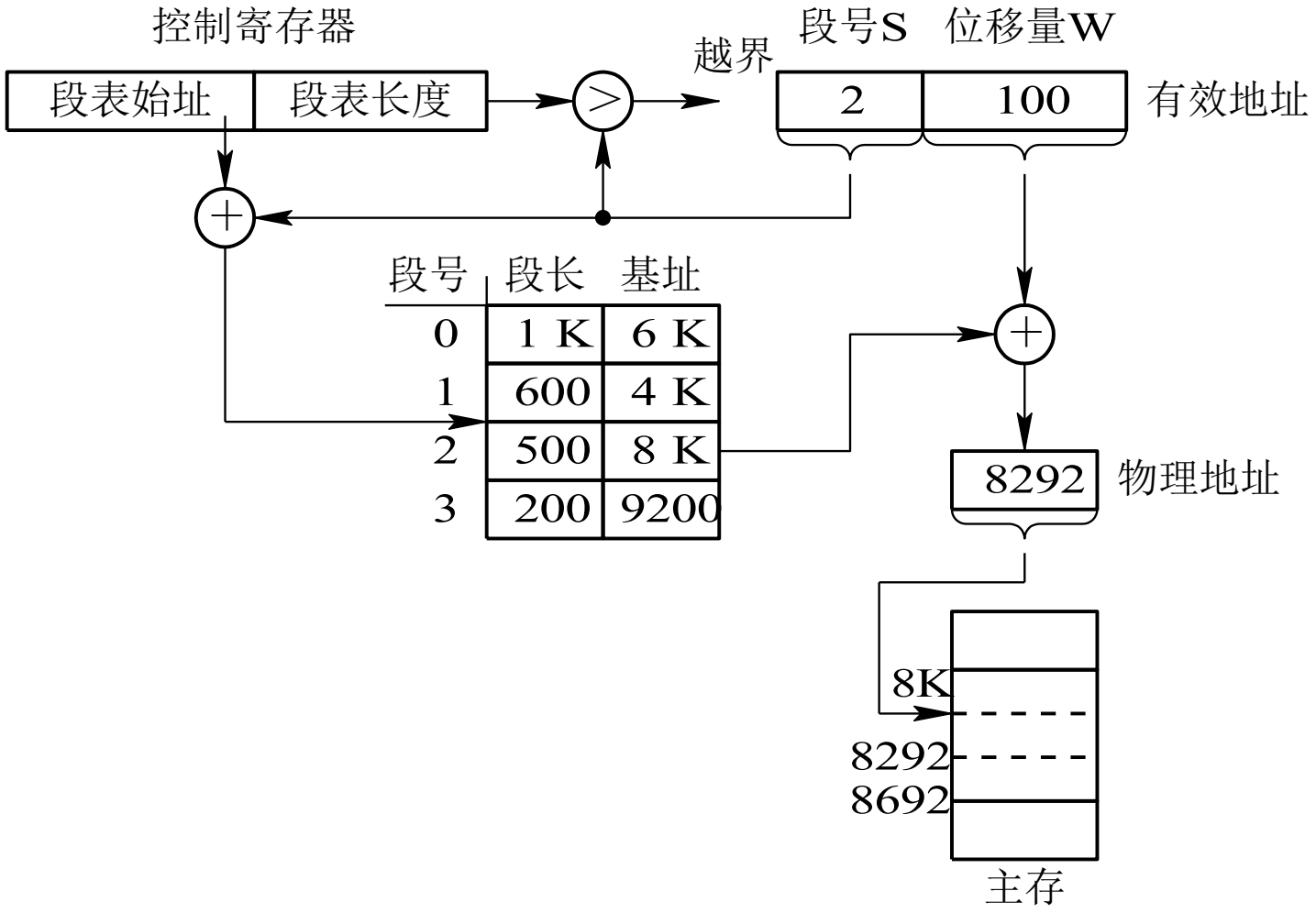


图4-18 分段系统的地址变换过程

4.5.2 分段系统的基本原理

像分页系统一样，当段表放在内存中时，每要访问一个数据，都须访问两次内存，从而极大地降低了计算机的速率。

解决的方法也和分页系统类似，再增设一个联想存储器，用于保存最近常用的段表项。由于一般情况是段比页大，因而段表项的数目比页表项的数目少，其所需的联想存储器也相对较小，便可以显著地减少存取数据的时间，比起没有地址变换的常规存储器的存取速度来仅慢约10%～15%。

4.5.2 分段系统的基本原理

4．分页和分段的主要区别

由上所述不难看出，分页和分段系统有许多相似之处。比如，两者都采用离散分配方式，且都要通过地址映射机构来实现地址变换。但在概念上两者完全不同，主要表现在下述三个方面。

(1) 页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外碎片，提高内存的利用率。或者说，分页仅仅是由于系统管理的需要而不是用户的需要。段则是信息的逻辑单位，它含有一组其意义相对完整的信息。分段的目的是为了能更好地满足用户的需要。

4.5.2 分段系统的基本原理

(2) 页的大小固定且由系统决定，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而在系统中只能有一种大小的页面；而段的长度却不固定，决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。

(3) 分页的作业地址空间是一维的，即单一的线性地址空间，程序员只需利用一个记忆符，即可表示一个地址；而分段的作业地址空间则是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。

4.5.3 信息共享

分段系统的一个突出优点，是易于实现段的共享，即允许若干个进程共享一个或多个分段，且对段的保护也十分简单易行。在分页系统中，虽然也能实现程序和数据的共享，但远不如分段系统来得方便。

我们通过一个例子来说明这个问题。例如，有一个多用户系统，可同时接纳40个用户，他们都执行一个文本编辑程序(Text Editor)。如果文本编辑程序有160 KB的代码和另外40 KB的数据区，则总共需有 8 MB的内存空间来支持40个用户。如果160 KB的代码是可重入的(Reentrant)，则无论是在分页系统还是在分段系统中，该代码都能被共享，在内存中只需保留一份文本编辑程序的副本，此时所需的内存空间仅为1760 KB(40×40+160)，而不是8000 KB。

4.5.3 信息共享

假定每个页面的大小为4 KB，那么，160 KB的代码将占用40个页面，数据区占10个页面。为实现代码的共享，应在每个进程的页表中都建立40个页表项，它们的物理块号都是21#～60#。在每个进程的页表中，还须为自己的数据区建立页表项，它们的物理块号分别是61#～70#、71#～80#、81#～90#，…，等等。图4-19是分页系统中共享editor的示意图。

4.5.3 信息共享

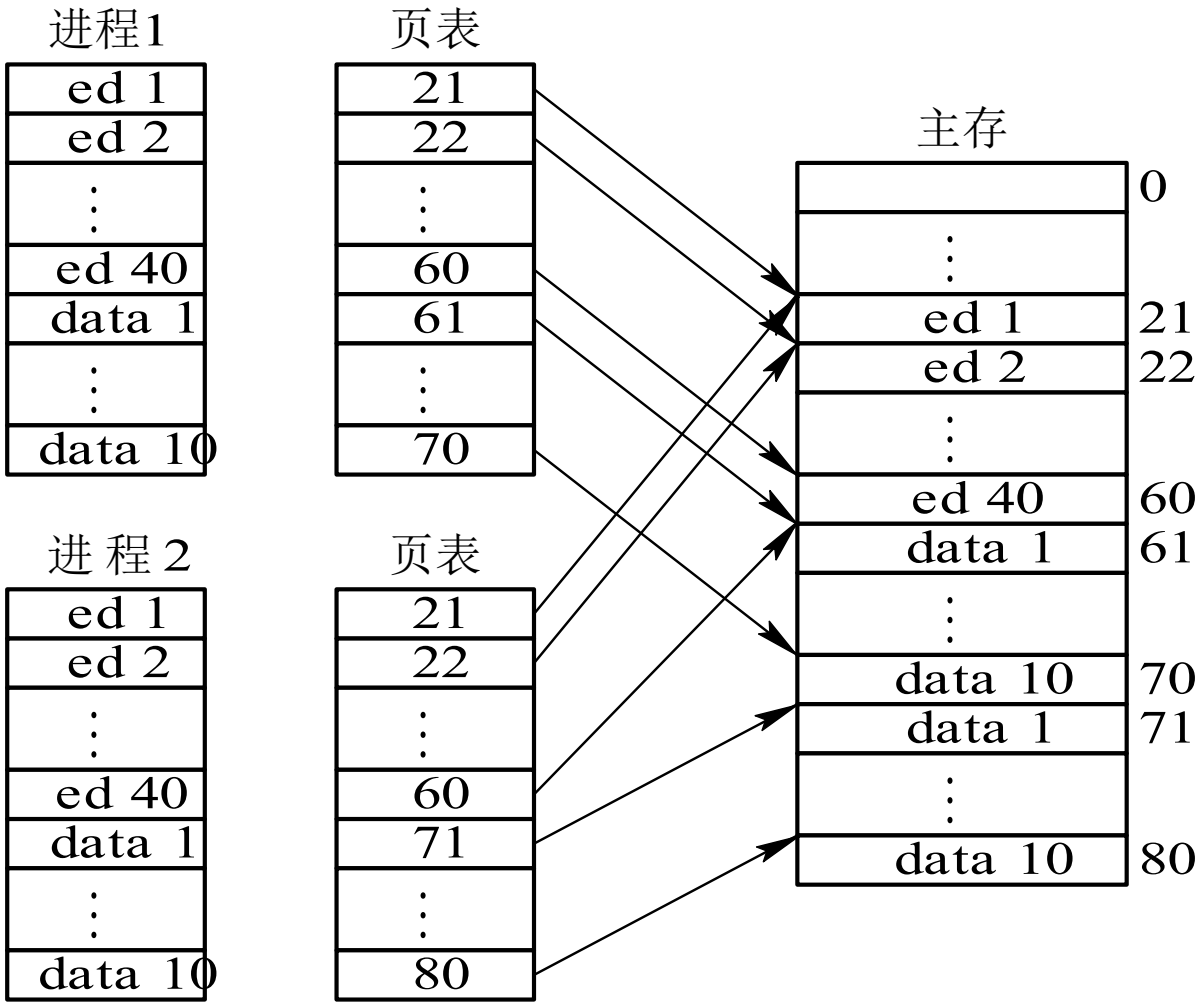


图4-19 分页系统中共享editor的示意图

4.5.3 信息共享

在分段系统中，实现共享则容易得多，只需在每个进程的段表中为文本编辑程序设置一个段表项。图4-20是分段系统中共享editor的示意图。

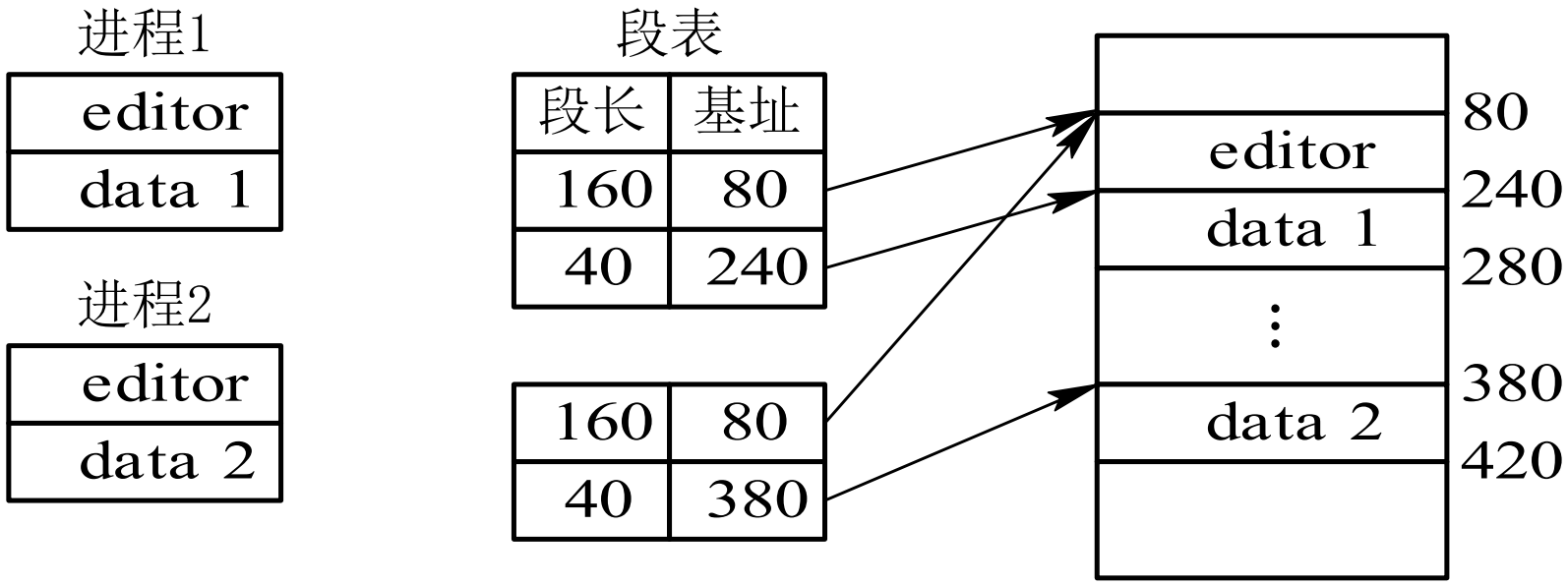


图 4-20　分段系统中共享editor的示意图

4.5.3 信息共享

可重入代码(Reentrant Code)又称为“纯代码”(Pure Code)，是一种允许多个进程同时访问的代码。为使各个进程所执行的代码完全相同，绝对不允许可重入代码在执行中有任何改变。因此，可重入代码是一种不允许任何进程对它进行修改的代码。

但事实上，大多数代码在执行时都可能有些改变，例如，用于控制程序执行次数的变量以及指针、信号量及数组等。为此，在每个进程中，都必须配以局部数据区，把在执行中可能改变的部分拷贝到该数据区，这样，程序在执行时，只需对该数据区(属于该进程私有)中的内容进行修改，并不去改变共享的代码，这时的可共享代码即成为可重入码。

段式特点

1.优点：

- 符合用户观点和程序逻辑
- 易于保护
- 易于共享
- 易于动态链接
- 易于动态扩充
- 可以实现虚存

2.缺点：

- 分配回收同可变式分区，实现复杂
- 存在二次访内指令执行速度慢问题，需联想存储器支持
- 有外部碎片，需移动开销

4.5.4 段页式存储管理方式

◆ 段页式引入

- 段式优于页式:便于共享和保护、动态伸缩、动态链接
- 页式优于段式:消除“碎片”问题、分配与回收简单
- 段页式：结合二者优点

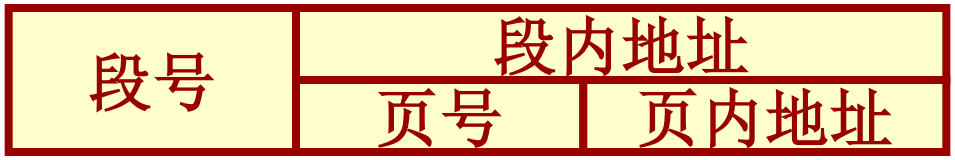
分配与回收:同页式

共享与保护、动态伸缩、动态链接:同段式

4.5.4 段页式存储管理方式

段页式存储管理实现

1. 进程分为若干段（一个进程有一张段表），每段由若干页组成（每段有一张自己的页表）。
2. 内存分页面、存储管理的分配单位是页
3. 逻辑地址：二维<段号，段内地址>
   <段号，页号，页内偏移>



4.地址变换：先查段表，再查该段的页表。请求式系统中存在缺段中断和缺页中断。

4.5.4 段页式存储管理方式

1．基本原理

段页式系统的基本原理，是分段和分页原理的结合，即先将用户程序分成若干个段，再把每个段分成若干个页，并为每一个段赋予一个段名。图4-21示出了一个作业地址空间的结构。该作业有三个段，页面大小为4 KB。在段页式系统中，其地址结构由段号、段内页号及页内地址三部分所组成，如图4-22所示。

4.5.4 段页式存储管理方式

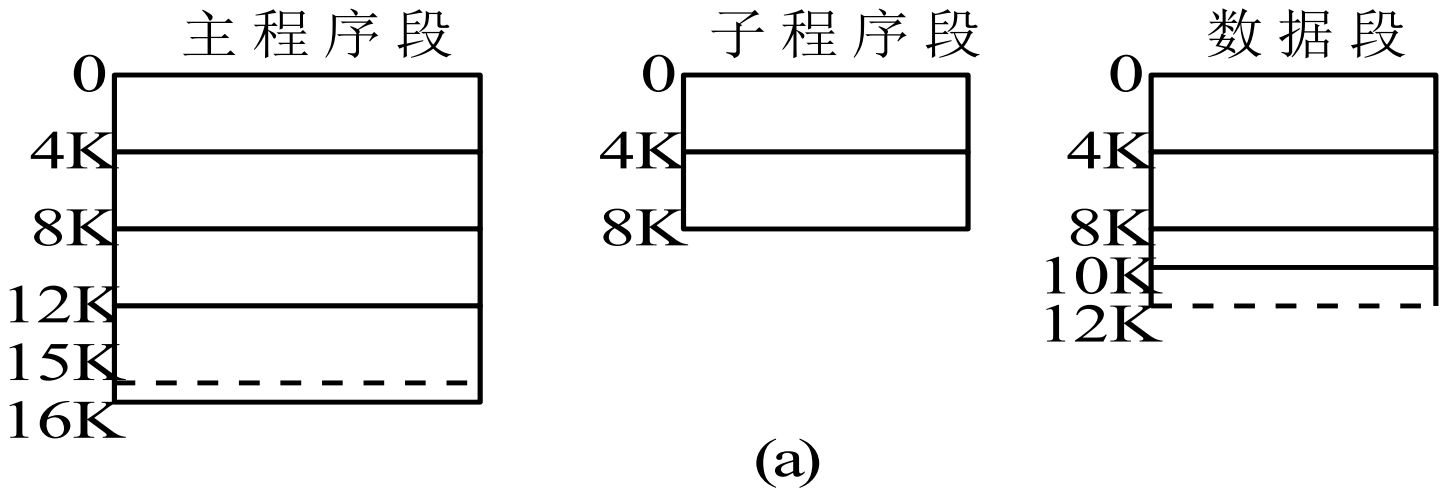


图 4-21 作业地址空间和地址结构

段页式表格例

(1) 段表：每个进程一个

段号	页表首址	页表长度
0		
…		
s	b’	*l*’
…		
l-1		

(2)页表：每个段一个

逻辑页号	页面号
0	
…	…
p	f
…	
l’-1	…

段页式寄存器及联想存取器

(3)段表基址寄存器：保存正运行程度段表首址；

b

(4)段表限长寄存器：保存正运行程序段表长度。

l

(5)快表：一组联想寄存器 (快段表+快页表)

段号	逻辑页号	页面号
s	**p**	**f**

4.5.4 段页式存储管理方式

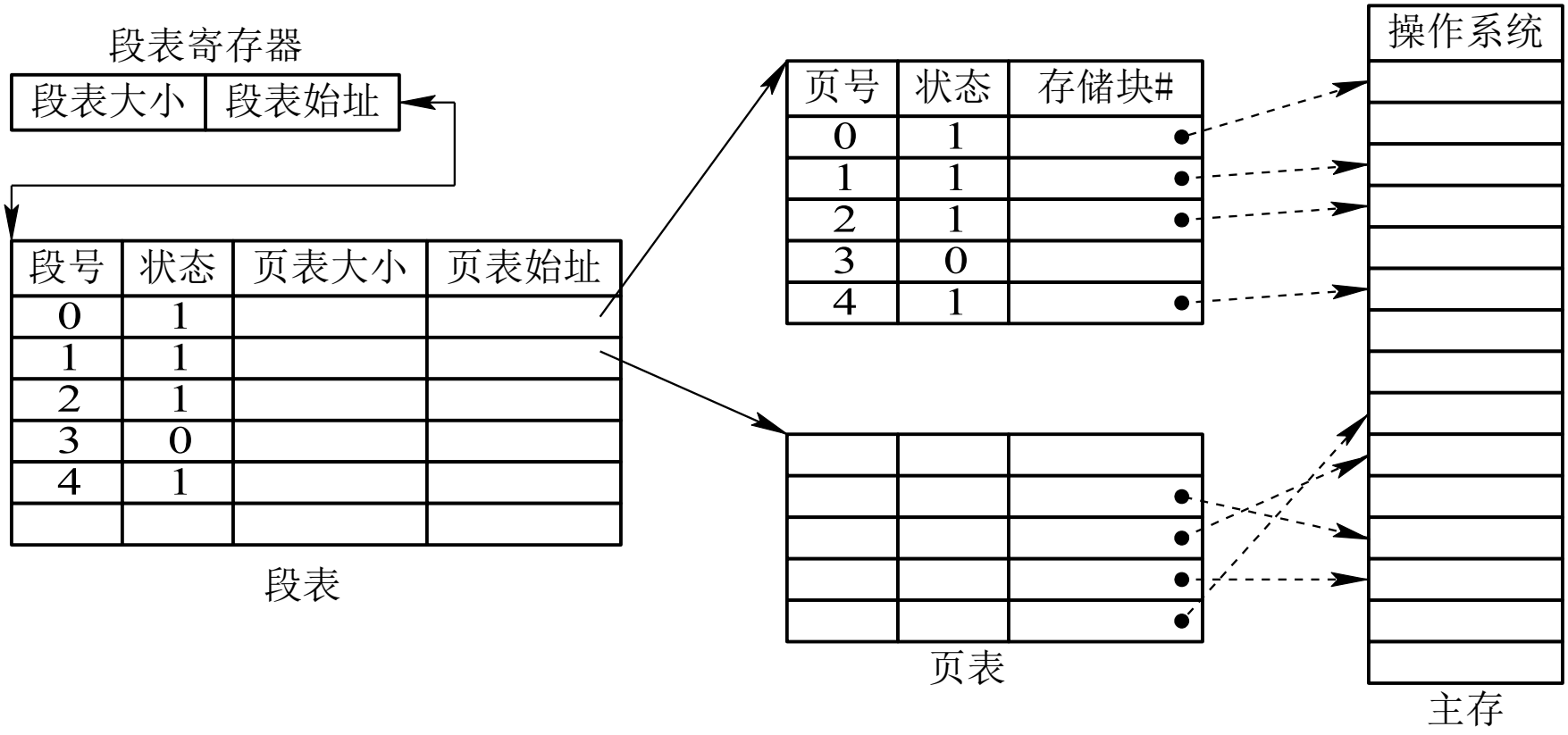


图4-22 利用段表和页表实现地址映射

4.5.4 段页式存储管理方式

2．地址变换过程

在段页式系统中，为了便于实现地址变换，须配置一个段表寄存器，其中存放段表始址和段表长TL。进行地址变换时，首先利用段号S，将它与段表长TL进行比较。

若S<TL，表示未越界，于是利用段表始址和段号来求出该段所对应的段表项在段表中的位置，从中得到该段的页表始址，并利用逻辑地址中的段内页号P来获得对应页的页表项位置，从中读出该页所在的物理块号b，再利用块号b和页内地址来构成物理地址。图4-23示出了段页式系统中的地址变换机构。

4.5.4 段页式存储管理方式

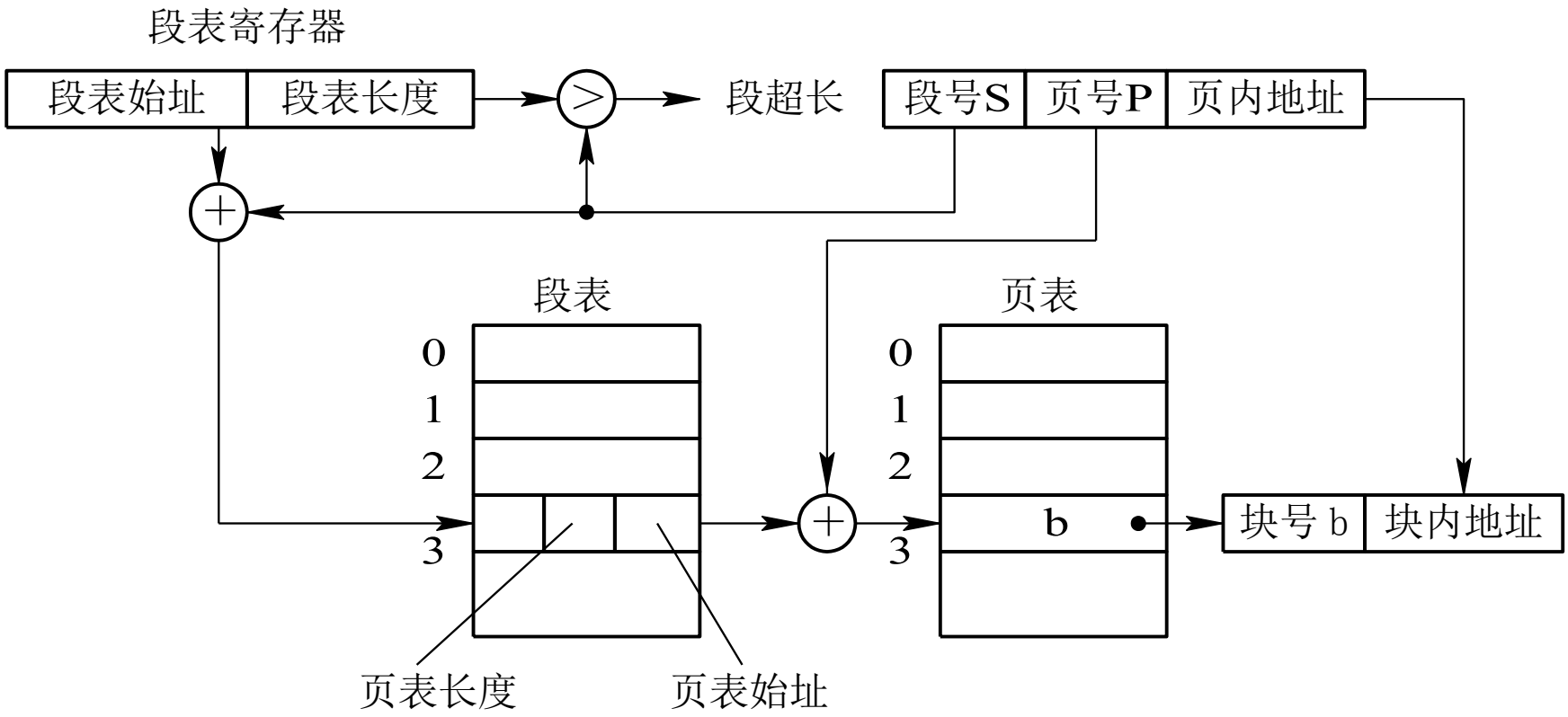
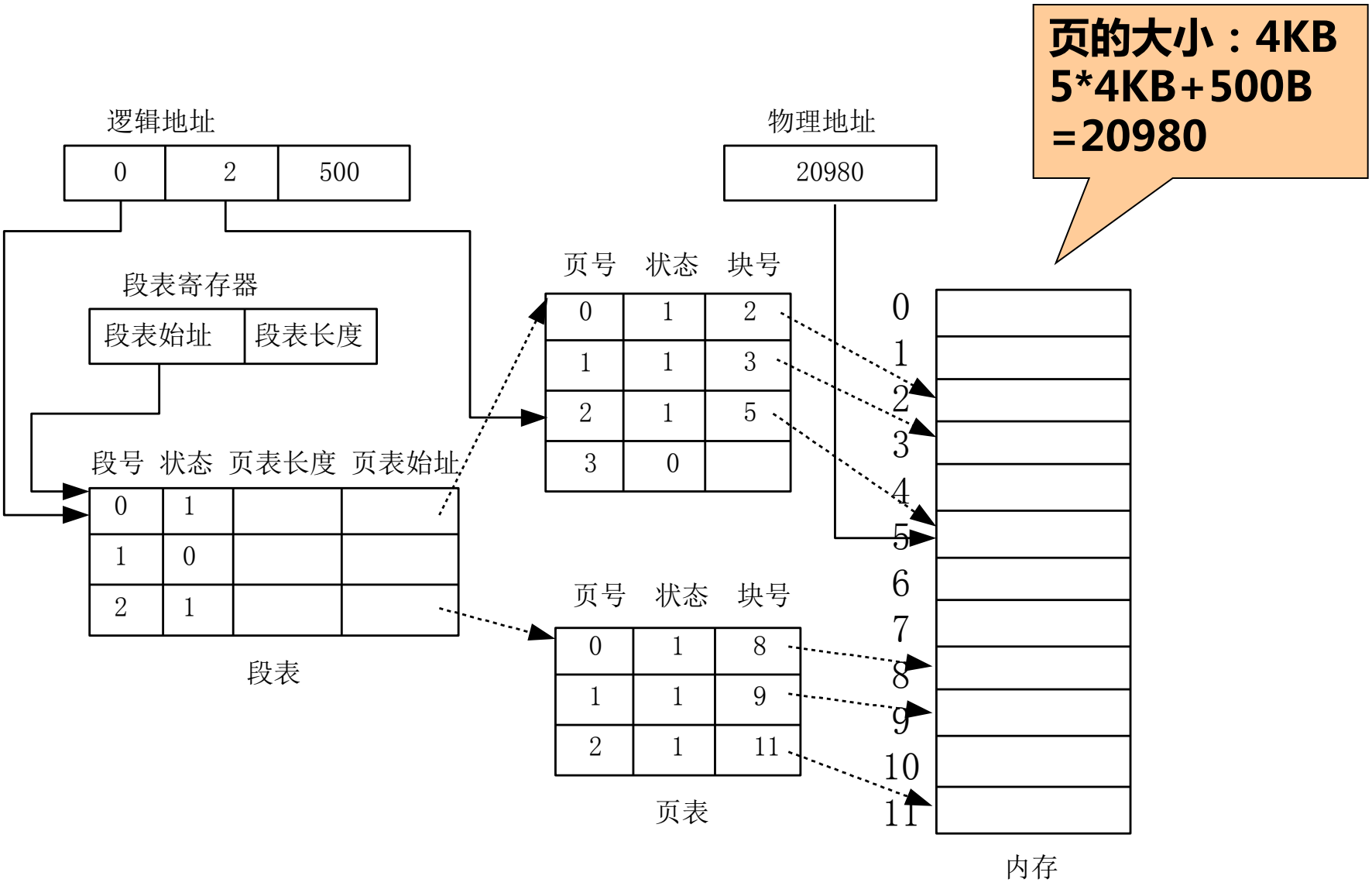


图4-23 段页式系统中的地址变换机构

4.5.4 段页式存储管理方式



4.5.4 段页式存储管理方式

在段页式系统中，为了获得一条指令或数据，须三次访问内存。第一次访问是访问内存中的段表，从中取得页表始址；第二次访问是访问内存中的页表，从中取出该页所在的物理块号，并将该块号与页内地址一起形成指令或数据的物理地址；第三次访问才是真正从第二次访问所得的地址中，取出指令或数据。

显然，这使访问内存的次数增加了近两倍。为了提高执行速度，在地址变换机构中增设一个高速缓冲寄存器。每次访问它时，都须同时利用段号和页号去检索高速缓存，若找到匹配的表项，便可从中得到相应页的物理块号，用来与页内地址一起形成物理地址；若未找到匹配表项，则仍须再三次访问内存。

4.5.4 段页式存储管理方式

段页式特点

- 分配与回收:同页式
- 共享与保护、动态伸缩、动态链接:同段式
- 三次访内，必须联想存储器支持
- 碎片：n（n是包含段的数目）倍于页式