第四章 存储器管理

关于存储器

1. 存储器是计算机重要组成部分，**问题是**：

虽然存储技术的发展，使得存储速度、容量[1]增长很快，但用户需求[2-1]及软件技术[2-2]也在快速发展，仍然需要速度更快、容量更大的内存。因此，仍然需要内存管理技术。

2. 内存管理技术的**作用**：

通过对内存的管理，来“*提高*”内存的**速度**[1]，来提高内存的**利用率/容量**[2]，同时也提高了整个系统的性能[3]。

3. 存储管理**内容**：

包括：存储器的层次管理§4.1，存储器的连续分配管理§4.3，存储器的不连续分配管理等内容§4.5、§4.6。

本章内容重点是：§4.3、§4.5、§4.6。

4.1 存储器的层次结构

在计算机执行时，几乎每一条指令都涉及对存储器的访问，因此存储器要满足如下3点要求：

① 存储速度要足够快[1]，快到能与处理机的速度相匹配，否则会严重影响处理机速度，影响整个系统的处理能力。

② 存储容量要足够大[2]，以便存放足够多的程序与数据。

③ 存储价格要足够便宜[3]，以便配置足够大的存储。

问题是：上述要求难以同时满足：

快->不便宜，大->不便宜。

解决：关于快：分层次§4.1＋管理§4.5；

关于容量、价格：虚存§5

4.1.1 多级存储器结构

对于一般的计算机，存储层次至少应具有三级/层：最高层为CPU寄存器，中间为主存，最底层是辅存。

对于较高档次的计算机，还可以细分为寄存器、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级/层**。如图4-1所示。

各层次特点：

1. 存储层次越往上，访问速度越快，价格也越高，存储容量也越小。

2. 寄存器、高速缓存、主存储器和磁盘缓存均属于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*。

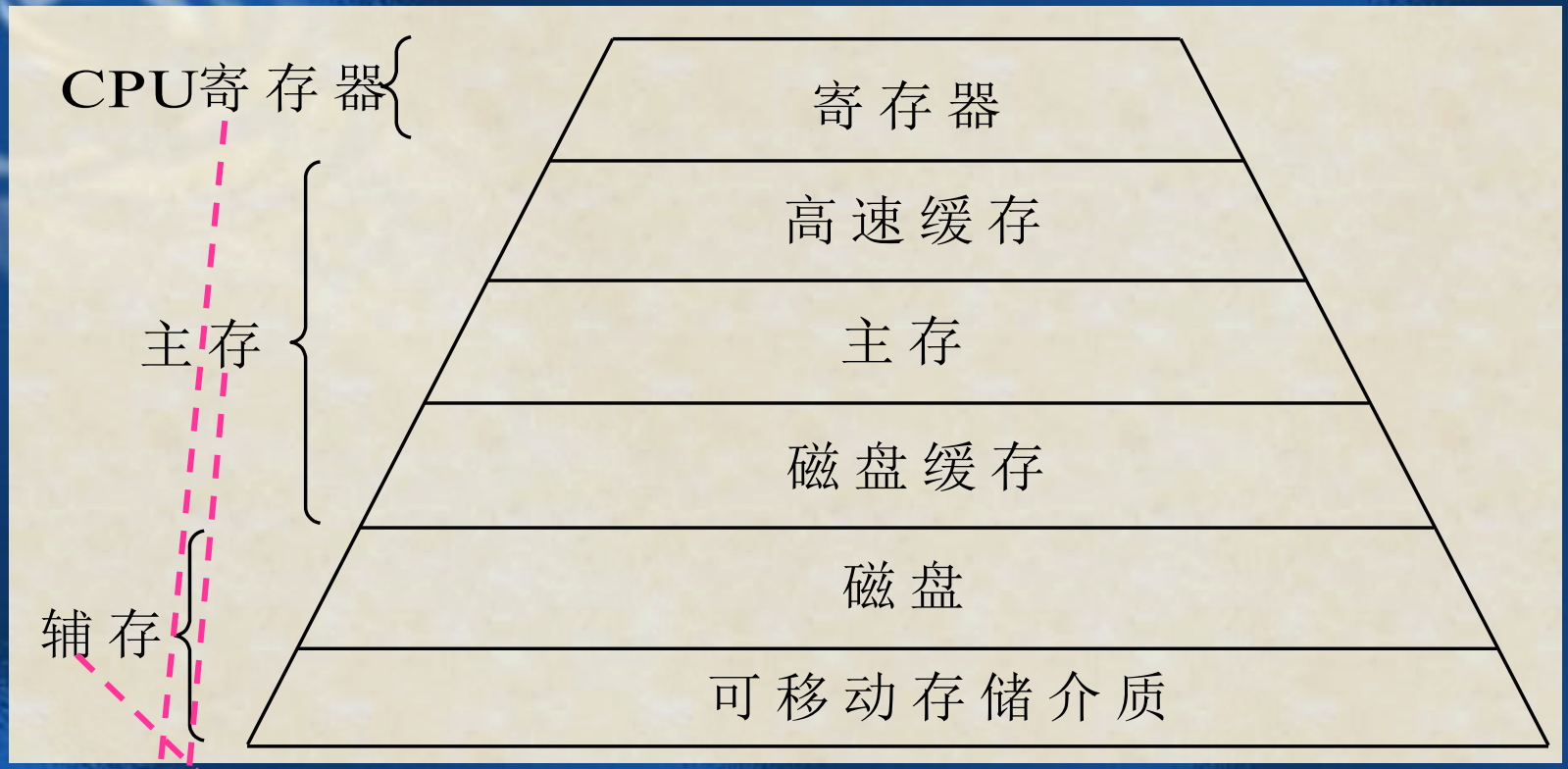


图4-1　计算机系统存储层次示意

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**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。

2. 可执行存储器

在计算机系统的存储层次中，寄存器、高速缓存和主存储器又被称为可执行存储器。

对于可执行存储器及辅存，计算机采用了不同的访问机制，所耗费的时间也不同。

- 对可执行存储器的访问，进程可以在很少的时钟周期内完成；
- 但对辅存的访问则需要很长的时间。

对辅存的访问，需要通过I/O设备来实现。会用到中断[1]、设备驱动程序[2]以及物理设备的运行[3]，在第6章中讲解。

本章讨论：可执行存储器的管理问题，包括：存储分配、回收、及在不同的存储层次间进行数据移动。

4.1.2 主存储器与寄存器(+快)

1. 主存储器

主存储器用于保存进程运行时的程序和数据[1]，也称可执行存储器，其容量因存储技术的发展而不断增加。

- CPU只能：（CPU与主存）
  - 从主存储器中读取指令和数据[2-1]，并将它们装入到寄存器中[2-2]；或者
  - 将寄存器中数据存到主存储器[3]。
- CPU与外围设备交换信息时，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。

2. 寄存器 （+快）

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，因此容量小(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 （+快）

1. 高速缓存

它是介于寄存器和存储器之间的存储器。

容量远大于寄存器[1]，而比内存约小两到三个数量级左右；访问速度快于主存储器[2]。

作用：主要用于备份主存中较常用的数据，以减少处理机对主存的访问次数，从而可大幅度地提高程序执行速度。

例：根据程序访问的局部性原理§5，将主存中一些经常访问的信息存放在高速缓存中§4.5.2 分页管理—快表，减少访问主存储器的次数，可大幅度提高程序执行速度。

2. 磁盘缓存

问题：目前磁盘的I/O速度远低于对主存的访问速度，因此，CPU不能直接访问磁盘。

解决：类似于高速缓存，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。

磁盘缓存利用主存中的存储空间，来暂存从磁盘中读出(或写入磁盘)的信息。因此，

辅存中的数据必须先复制到主存方能使用[1+why?]，所以，主存也可以看做是辅存的高速缓存。反之，计算产生的数据也必须先存在主存中[2+why?]，才能输出到辅存。

4.2 程序的装入和链接

要运行一个用户程序，必须先将它装入内存，然后再将它转变为一个可执行程序，通常都要经过以下几个步骤：

(1) **编译**，由**编译程序**对用户源程序进行编译，形成若干个目标模块；

(2) **链接**，由**链接程序**将编译后形成的一组目标模块[1]以及它们所需要的库函数[2]链接在一起，形成一个**完整**的装入模块；

(3) **装入**，由**装入程序**将装入模块装入内存。

如图4-2所示。

本节将主要讨论链接和装入过程。

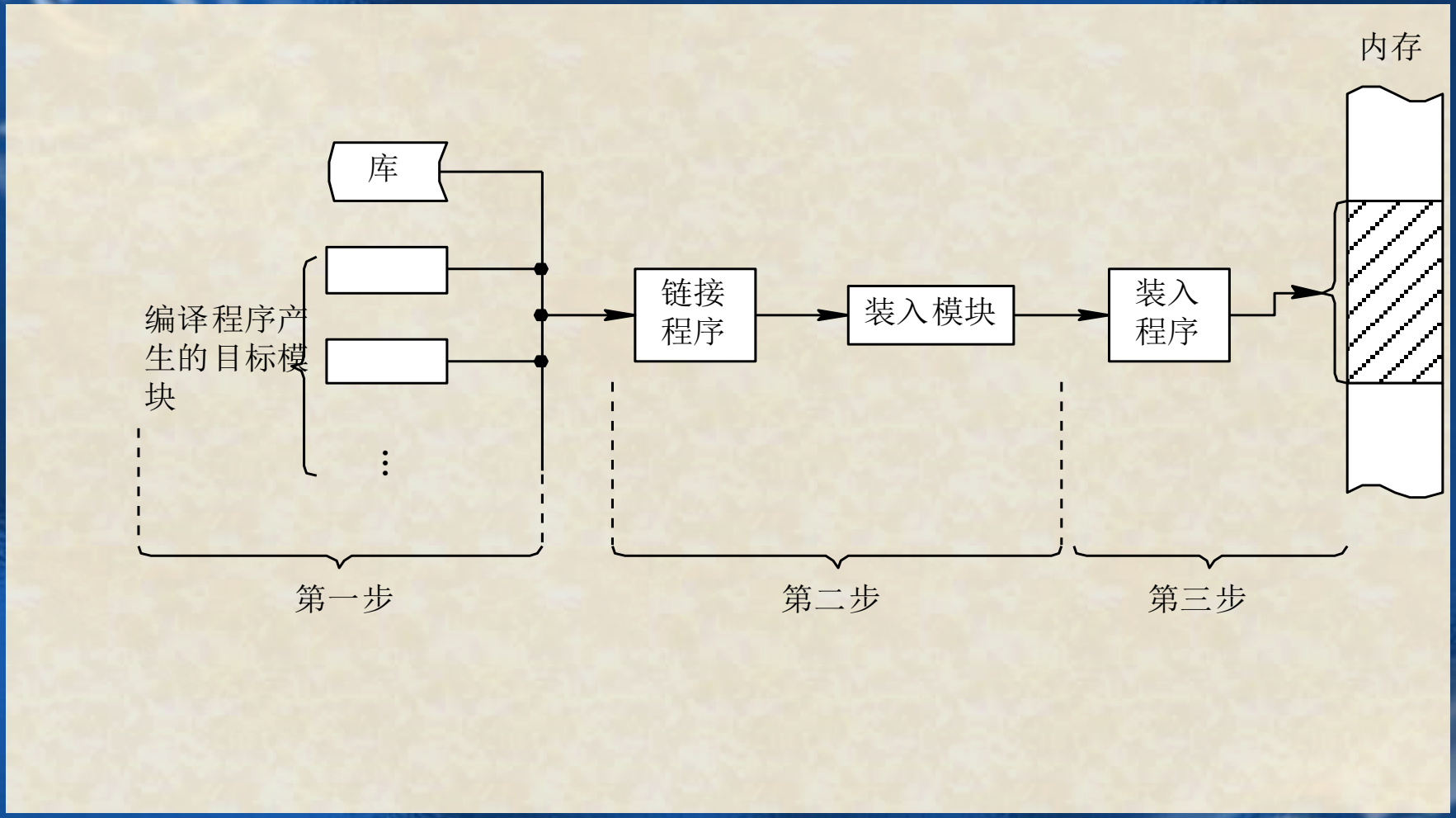


图 4-1 对用户程序的处理步骤

4.2.1 程序的装入

为方便起见，先介绍一个无需进行链接的单个目标模块(也就是装入模块)的装入过程。

可以有如下三种装入方式：----什么时候进行地址转换

1. 绝对装入方式(Absolute Loading Mode)

对于小型OS系统，假设它只能运行单道程序，用户空间起始地址为R，经编译后的**目标代码/装入模块**就可以存放在R之后的空间。目标代码中的相对地址/逻辑地址与内存地址/绝对地址相同。因此，可以采用绝对装入方式。

绝对装入程序按照装入时模块中的绝对地址，将程序和数据装入内存。装入后，因为程序中的逻辑地址与内存地址完全相同，所以，**无须对程序和数据的地址进行修改**。问题：

1. 程序和数据不能**更改、移动**，
2. 用户更希望用**符号地址**，**运行时**，再转换为**绝对地址**。
3. 多道环境问题。为此，引入了： **可重定位的装入方式**

2. 可重定位装入方式(静态重定位装入方式)

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，这只适用于单道程序环境。

问题：在多道程序环境下，有多个程序，编译程序不能够预知各目标模块到底放在内存的什么地方，因此：

需要对编译后的几个目标模块重新编址。

通常，一个程序中的起始地址从0开始，各模块中的地址也都是相对于该起始地址计算的，因此，这种地址叫**相对地址**。

很显然，相对地址不是绝对地址，需要将相对地址**映射**成绝对地址，也叫**重定位**。即：根据内存的实际情况，将装入模块装入到内存的合适位置，该装入方式叫**可重定位装入方式**。

在图4-3中，利用可重定位装入方式装入了一个模块。

这种重定位是在装入时完成的，也叫**静态重定位**。

地址映射：绝对地址＝相对地址+10000

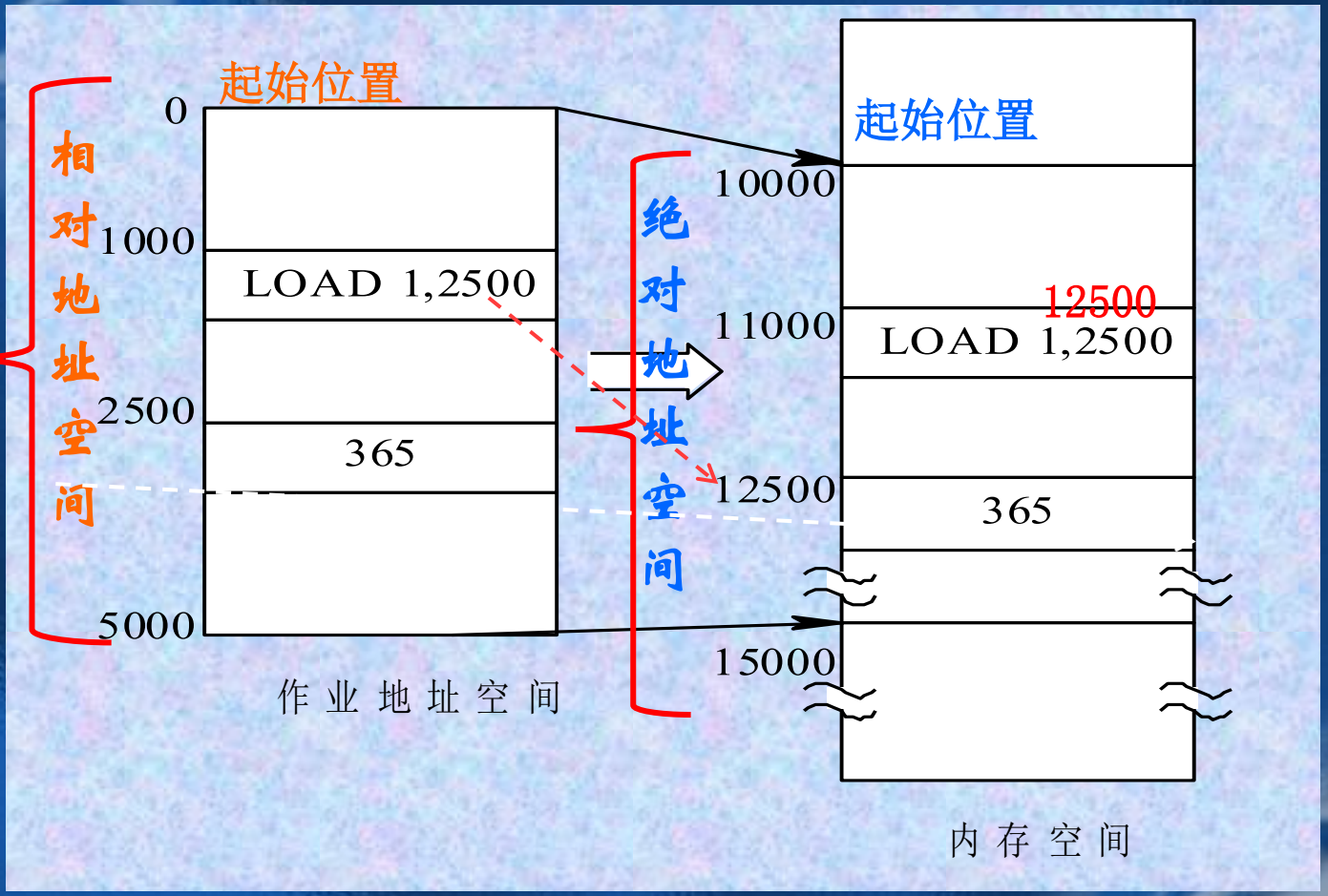


图 4-3 作业装入内存时的情况

3. 动态运行时装入方式

可重定位装入方式可将装入模块装入到内存中任何允许的位置，故可用于多道程序环境。

问题是：这种装入方式是静态的，程序的绝对地址在装入时已经确定，因此在程序运行过程中，不允许在内存中移动位置，因为移动意味着程序和数据的绝对地址发生了变化。

解决方法是：要么重新定位：重新计算绝对地址---CPU代价高新问题；要么使用**动态运行时装入方式**。

动态运行时装入方式：程序/数据装入内存后，并不进行地址转换；等到程序运行时，才进行地址转换。

4.2.2 程序的链接 （+快，讲1点，提2点）

1. 静态链接(Static Linking)方式

在程序运行之前，先将各目标模块[1]及它们所需的库函数[2]链接成一个完整的装配模块，以后不再拆开。

在图4-4(a)中示出了经过编译后所得到的三个目标模块A、B、C，它们的长度分别为L、M和N。在模块A中有一条语句CALL B，用于调用模块B。在模块B中有一条语句CALL C，用于调用模块C。B和C都属于外部调用符号，在将这几个目标模块装配成一个装入模块时，须解决以下两个问题：

(1) 对相对地址进行修改。

(2) 变换外部调用符号。

(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，要作用**统一的相对地址**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。

(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。

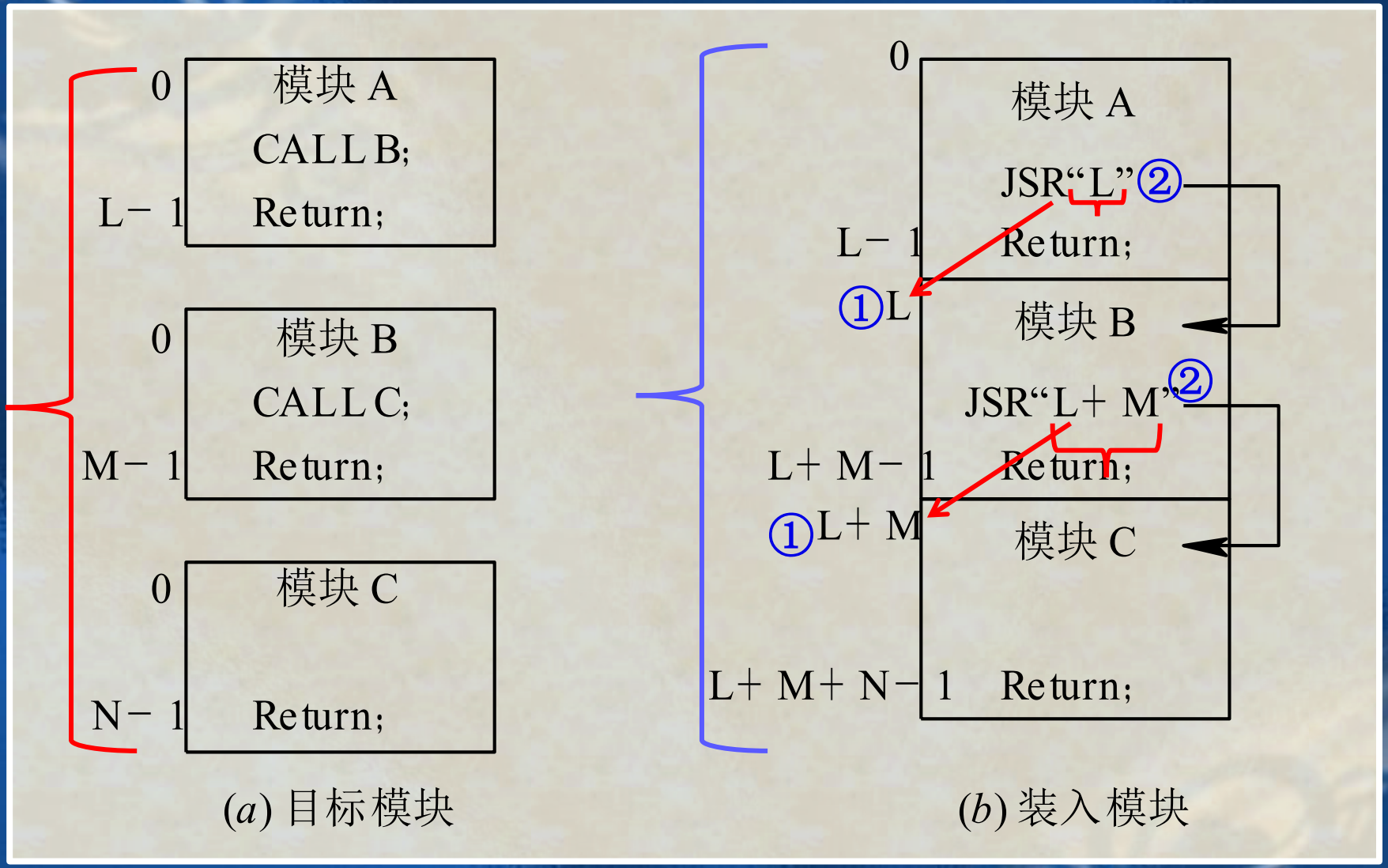


图 4-4 程序链接示意图 ①②

2. 装入时动态链接(Load-time Dynamic Linking)

目标模块在装入内存时边装入边链接的，即装入时，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，并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。不仅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。

(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。

3. 运行时动态链接(Run-time Dynamic Linking)

问题：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。

解决：**运行时动态链接方式**是将对某些模块的链接推迟到执行时才执行，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。

4.3 连续分配存储管理方式

4.3.1 单一连续分配

在**单道程序环境下**，当时的存储器管理方式是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，它通常是放在内存的低址部分。而**用户区**中，仅有一道用户程序，即整个内存的用户空间由该程序独占。

这样的存储器分配方式被称为单一连续分配方式。

转 4.3.2

虽然在早期的单用户、单任务操作系统中，有不少都配置了存储器保护机构，用于防止用户程序对操作系统的破坏，但近年来常见的几种单用户操作系统中，如CP/M、MS-DOS及RT11等，都未采取存储器保护措施。这是因为，一方面可以节省硬件，另一方面在单用户环境下，机器由一用户独占，不可能存在其他用户干扰的问题，因此这是可行的。这时可能出现的破坏行为也只是用户程序自己去破坏操作系统，其后果并不严重，只是会影响该用户程序的运行，且操作系统也很容易通过系统的再启动而重新装入内存。

4.3.2 固定分区分配

在多道OS中，要求多道程序之间不能相互干扰，为此，需要将整个用户空间划分成多个程序空间/分区，每个分区装入一道程序。怎样划分呢？

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) 按各分区大小相等来划分。

缺点：缺乏灵活性。当程序太小时，造成内存空间的浪费；当程序太大时，一个分区又装不下，程序也应无法运行.

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。

(2)按各分区大小不等来划分。

为了增加存储分配的灵活性，应该把用户空间划分成若干个大小不等的分区。通常，这些分区中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，就可以为不同大小的程序，分配不同大小的分区。

2．内存分配

通常将各分区按大小进行排队，并建立一张分区*使用表*，各表项包括：每个分区的起始地址、大小及状态，如图4-5所示。

当需要内存分配时，OS检索该表，从中找出一个能满足大小要求的、尚未分配的分区[1]；然后分配给请求程序[2]；最后将该表项中的状态置为“已分配”[3]；若未找到大小足够的分区，则拒绝为该请求分配内存。

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

这种存储分配方式，是早期多道程序系统中的存储管理方式。因为每个分区大小固定，造成了存储空间的浪费对比可变分区管理。现在已很少用。

4.3.3 动态分区分配/可变分区分配

动态分配，就是根据进程的实现需要，为进程动态地分配存储空间。

动态分区管理中，会涉及到相关的数据结构、分区的分配与回收算法等问题。

1. 动态分区分配中的数据结构

用于描述空闲分区[1]和已分配分区[2]。

常用的数据结构有以下两种形式：

① **空闲分区表**

在系统中设置一张空闲分区表（顺序表），用于记录每个空闲分区的情况。（表中“**状态**”位可分为已分配/未分配分区）

每个表项/表目记录一个空闲分区情况，包括分区号、分区大小、分区始址和状态等数据项，如图4-6所示。

分区号	分区大小(KB)	分区始址(K)	状态
1	50	85	空闲
2	32	155	空闲
3	70	275	空闲
4	60	532	空闲
5	…	…	…

图4-6 空闲分区表

② 空闲分区链

在系统中设置一张空闲分区链表双向链表，将每个空闲分区链接起来。（教材未给出已分配分区表）

在每个分区的起始部分，设置了链接各分区的前向指针[1]及分区的一些**分配信息**(状态位[2]和分区大小[3])；同理，

在分区的尾部，设置了后向指针[1]及状态位[2]和分区大小[3].

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。

通过前、后向链接指针，形成一个双向链表，如图4-7所示。

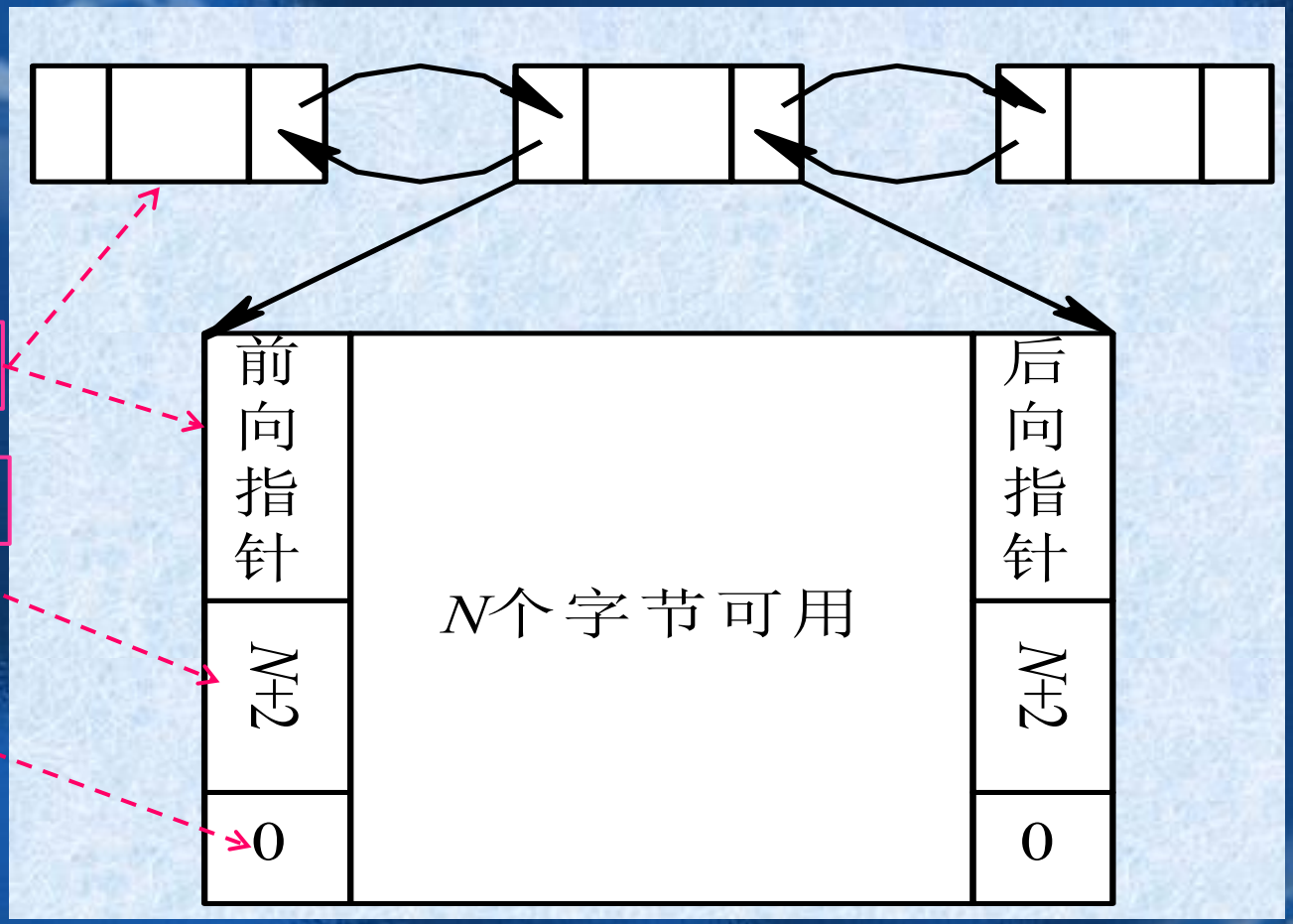


图4-7　空闲链结构

2. 动态分区分配算法（怎样分配：how?）

作业装入内存前，先按照内存分配算法，从空闲分区表或空闲分区链中选择一个分区。之后分配给作业。

动态分区分配算法较多，包括传统的四种算法和索引搜索算法。

传统的四种算法：首次适应算法、循环首次适应算法、最佳适应算法及最坏适应算法。

索引搜索算法：快速适应算法、伙伴系统及哈希算法。

这些算法内容较多，在下节§4.3.4中专门讲解。

3. 分区的分配与回收过程

在动态分区存储管理中，**主要的操作**是：分配内存/分区及回收内存/分区。

1) 分配内存过程

系统根据分配算法及所需分区的大小，从空闲分区表/链中查找，将找到的分区分配给作业。

问题：有些分区太小，难以分配出去。但如果不分配，就造成内存的浪费，这些分区叫**碎片**不能用。因为碎片是在分区之外的，所以也叫**外部碎片**。

设作业请求的分区大小为：u.size，空闲分区表中空闲分区的大小为：m.size。

图4-8是动态分区管理的分配流图。

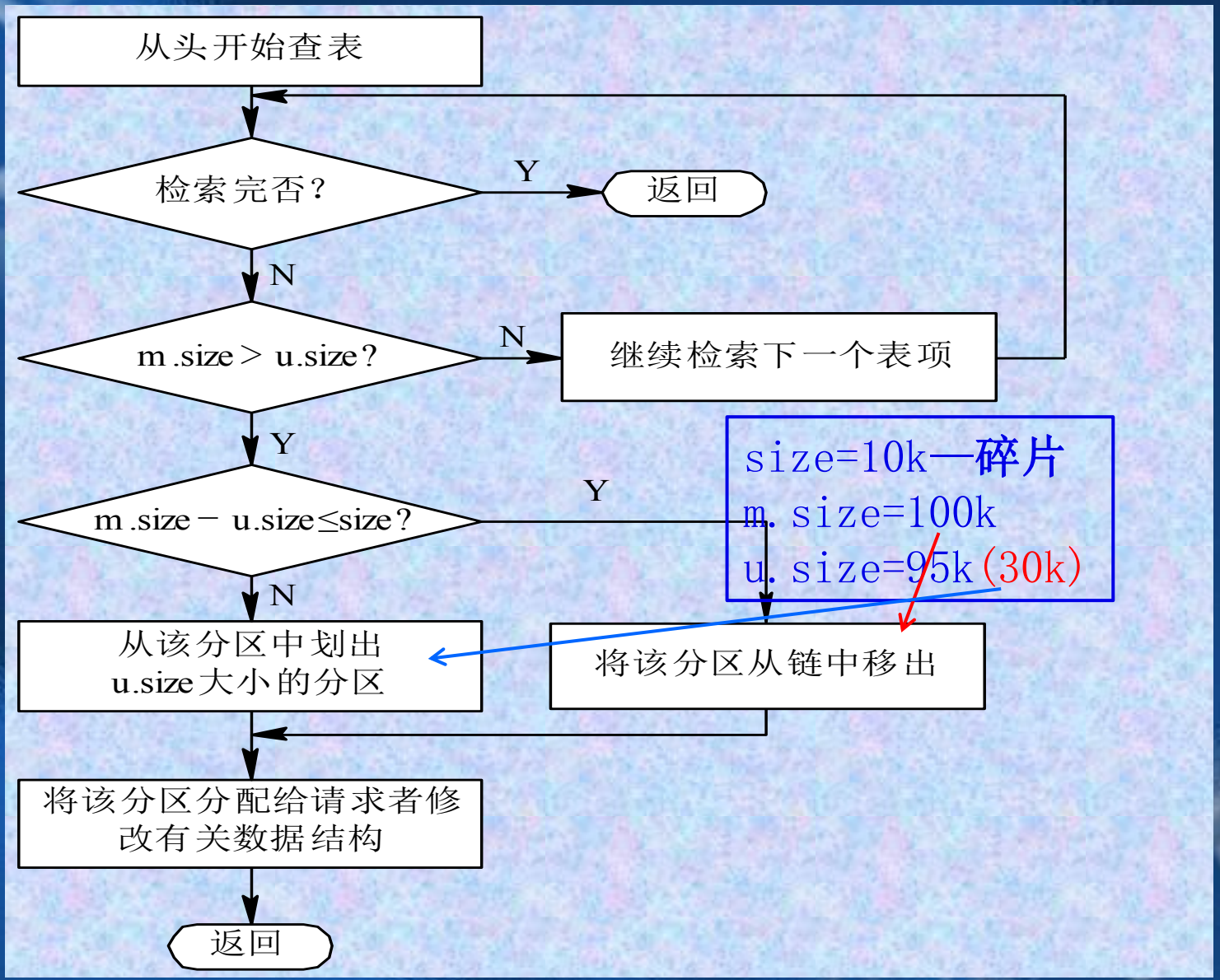

从头开始查表
检索完否？
Y
返回
N
m .size＞ u.size？
N
继续检索下一个表项
Y
m .size－ u.size≤size？
Y
size=10k—碎片
m. size=100k
u. size=95k (30k)
N
从该分区中划出
u.size大小的分区
将该分区从链中移出
将该分区分配给请求者修
改有关数据结构
返回

图4-8　内存分配流程

2) 回收内存

当进程运行完毕后，应需要释放内存。系统会根据回收区的首址，从空闲区链(表)中找到相应的插入点，此时可能出现以下四种情况之一：**如图4-9所示**。

(1) 回收区与插入点的前一个空闲分区F1相邻接，见图4-9(a)。将回收区与插入点的前一分区合并，不必为回收分区分配新表项，而只需修改其前一分区F1的大小。

(2) 回收分区与插入点的后一空闲分区F2相邻接，见图4-9(b)。将两分区合并，形成新的空闲分区，但用回收区的首址作为新空闲区的首址，大小为两者之和。

(3) 回收区同时与插入点的前、后两个分区邻接，见图4-9(c)。此时将三个分区合并，使用F1的表项和F1的首址，取消F2的表项，大小为三者之和。

(4) 回收区既不与F1邻接，又不与F2邻接。这时应为回收区单独建立一新表项，填写回收区的首址和大小，并根据其首址插入到空闲链中的适当位置。

图4-10示出了内存回收时的流程。

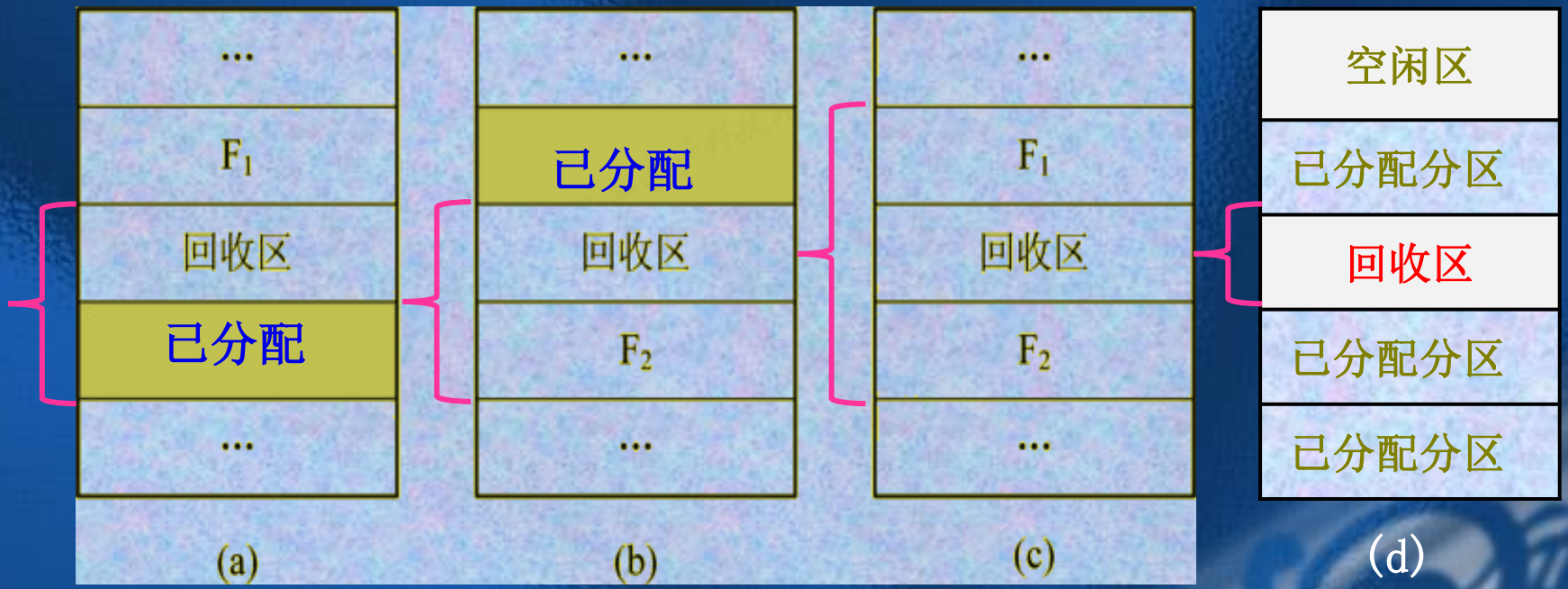


图4-9 内存回收时的情况

内存回收流程(课后学习)

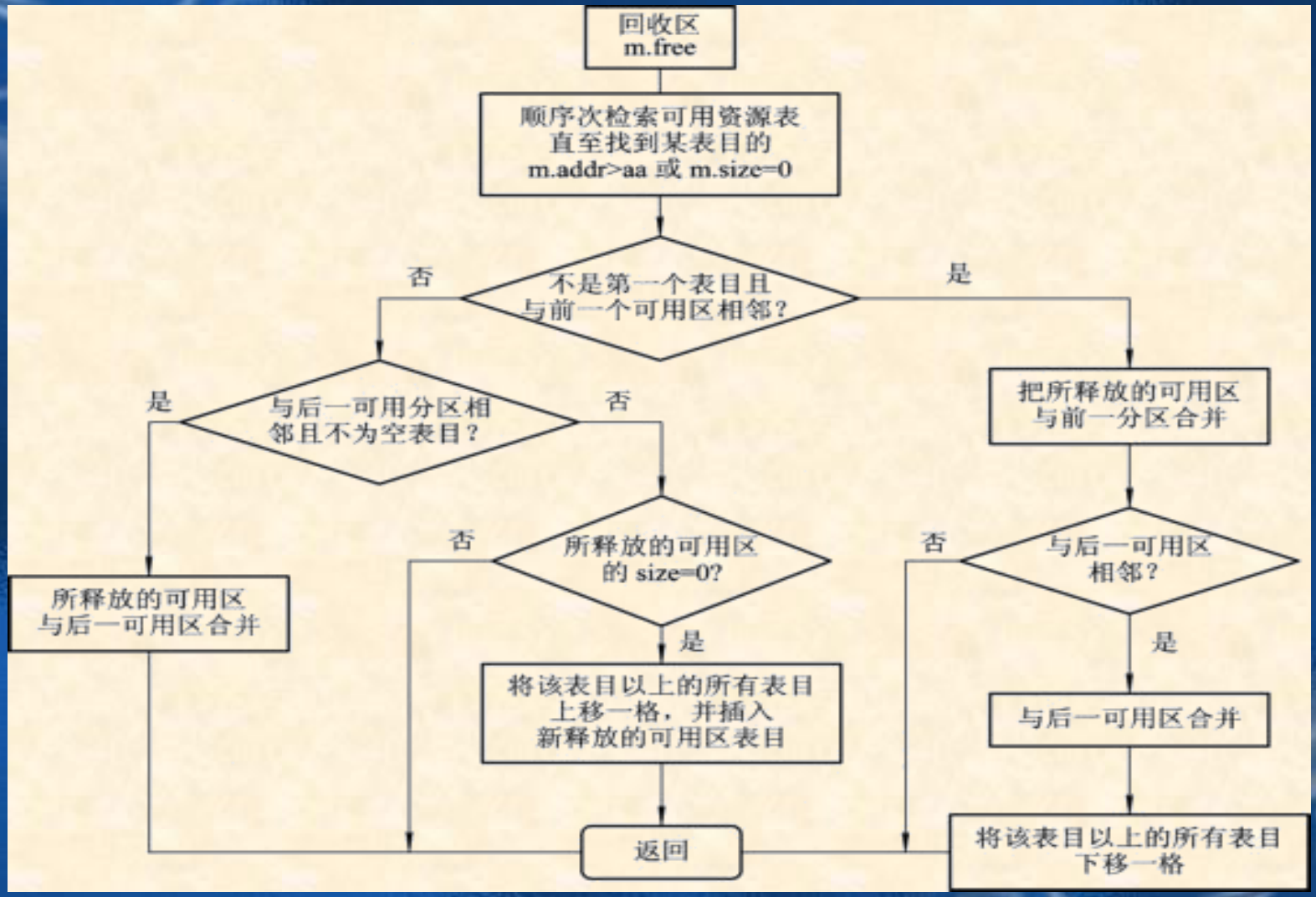


图4-10 内存回收流程

4.3.4 基于顺序搜索的动态分区分配算法

顺序搜索：依次搜索空闲分区表/链，找到一个能满足要求大小+算法的分区。

1. **首次适应(first fit，FF)算法**

以空闲分区链为例。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，重新划分分区1变2，其中一个分区分配给作业，另一个分区仍留在空闲链中。
- 若找不到，内存分配失败，返回。

优点：因为问题先分配低地址空间，所以使得高地址空间较少被划分，留下的分区较大，对大小作业都公平。

缺点：低地址部分碎片多，存储效率低、查找cpu开销大.

2. 循环首次适应(next fit，NF)算法

为避免低址部分留下许多碎片，以及减少查找开销，提出了循环首次适应算法。

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；
- 再按照作业的大小，重新划分分区1变2，其中一个分区分配给作业，另一个分区仍留在空闲链中。

技术上要求：设置一个指针，指示下次查找的起始位置。并且，采用循环的查找方式。

优点：因空闲分区分布均匀，所以减少了查找开销cpu

缺点：因空闲分区分布均匀，所以减少了大分区，对大作业不公平。

3. 最佳适应(best fit，BF)算法

所谓“**最佳**”是指每次为作业分配内存时，总是把能满足大小要求[1]、又是最小[2]的空闲分区分配给作业，避免“大材小用”。

为了加快查找，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，第一次找到的空闲分区，必然是最佳的。

优点：对大小作业都比较公平。

缺点：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碎片。

4. 最坏适应(worst fit，WF)算法

所谓“**最坏**”是指：把最大的空闲分区分配给作业，就是要“大材小用”。

为了加快查找，算法要求将所有的空闲分区按其容量由大到小的顺序形成一空闲分区链。这样，

第一次找到的空闲分区，必然是最坏的。

优点：可使剩下的空闲区不至于太小，产生碎片的几率最小，即存储效率高，且查找效率很高。对中、小作业有利。

缺点：它会使存储器中缺乏大的空闲分区。

对大作业更不利对比循环首次适应算法。

4.3.5 基于索引搜索的动态分区分配算法

顺序搜索算法，适合较小的系统。

当系统较大时，内存分区较多，分区链表较长，搜索/查找分区的速度就会慢，时间就会长 a:顺序表，排序->时间成本？ b:链表，二叉排序树?

在大、中型系统中，往往采用**基于索引搜索的动态分区分配算法**。包括快速适应算法、伙伴系统、哈希算法。

1. 快速适应(quick fit)算法/分类搜索算法

它将空闲分区根据其容量大小进行**分类**：

- 为相同容量的所有空闲分区，设立一个空闲分区**链表**。
- 为不同容量的空闲分区，设立多个空闲分区**链表**。同时，
- 在内存中设立一张索引表，每一个索引表项对应一个空闲分区**链表**及链表的**头指针**。

搜索过程：根据请求先查索引表，找到相应的链表；然后，取下链表中第一个结点所指的分区 有浪费，并**分配**给进程.

优点：查找速度快，不产生碎片，有利于大作业公平。

缺点：分区回收时，需要合并，系统开销大。

2. 伙伴系统(buddy system)

该算法规定，无论已分配分区或空闲分区，其大小均为2的k次幂，即：2^K(k为整数，1≤k≤m)。

通常用2^m表示整个可分配内存的大小(即最大分区的大小)，则系统刚开始运行时[1]，整个内存区就是一个2^m的空闲分区。

在系统运行过程中[2]，由于不断地分配与回收，将会形成若干个不连续的空闲分区。

将这些空闲分区按分区的容量大小进行分类。

- 对于具有相同大小(例2^i)的所有空闲分区，单独设立一个空闲分区双向链表[1]；
- 对于不同大小(例2^i，2^{i+1}，2^{i+2}，…)的空闲分区，就设立多个空闲分区链表。
- 在内存中设立一张索引表，……

分配算法：

当需要为进程分配一个大小为n的存储空间时，

1. 先计算一个i值，使$2^{i-1}<n\leq 2^i$，
2. 然后在空闲分区大小为2^i的空闲分区链表中查找。

- 若找到，即把该空闲分区分配给进程。
- 否则，表明长度为2^i的空闲分区已经耗尽，要在大小为2^{i+1}（2^i(入队) + 2^i (分配)）的空闲分区链表中寻找。

① 若找到，则把该空闲分区分为两个相等的分区，这两个分区称为一对伙伴大小相同，位置相邻，其中的一个分区用于分配，另一个加入分区大小为2^i的空闲分区链表中；

② 若找不到，则查找大小为2^{i+2}的空闲分区；

$2^{i+2}=2^{i+1}+2^{i+1}=2^{i+1}$(入队)$+2^i$(入队)$+2^i$ (分配)。

如下 所述（略）。

若找到，则对其进行两次分割：

第一次，将其分割为大小为2^{i+1}的两个分区一对伙伴，一个用于分配，一个加入到大小为2^{i+1}的空闲分区链表中；

第二次，将第一次用于分配的空闲区再分割为两个2^i的分区一对伙伴，一个用于分配，另一个加入到大小为2^i的空闲分区链表中。

若仍然找不到，则继续查找大小为2^{i+3}的空闲分区，以此类推。

（快）与一次分配可能要进行多次分割一样，一次回收也可能要进行多次合并。

如果回收大小为2^i的空闲分区时，若事先已存在2^i的伙伴分区，则应将它们合并为一个2^{i+1}的空闲分区，若事先已存在2^{i+1}的空闲分区时，又应继续与其伙伴分区合并为大小为2^{i+2}的空闲分区，依此类推（类似图4-9）。

在伙伴系统中，其分配和回收的时间性能取决于查找空闲分区的位置 和分割、合并空闲分区所花费的时间。

与前面所述的多种方法相比较：

由于在回收空闲分区时，需要对空闲分区进行合并，所以其时间性能比上述的快速适应算法差[1]；但因采用了索引搜索算法，所示比顺序搜索算法§4.3.4好[2]。

其空间性能则优于快速适应算法，比顺序搜索算法略差。

3. 哈希算法 （+快）

在上述的分类搜索算法和伙伴系统算法中，都是将空闲分区根据分区大小进行分类，对于每一类具有相同大小的空闲分区，单独设立一个空闲分区链表。在为进程分配空间时，需要在索引表中查找到所需空间大小所对应的表项，从中得到对应的空闲分区链表表头指针，从而通过查找得到一个空闲分区。如果对空闲分区分类较细，则相应的空闲分区链表也较多，因此选择合适的空闲链表的开销也相应增加，且时间性能降低。

哈希算法：找一个哈希函数：

以空闲分区大小为关键字，建立哈希表；

表中的每一个表项对应一个空闲分区链表的表头指针。

关键点：结果是**计算**出来的快，而不是通过比较找出来的。

4.3.6 动态可重定位分区分配

1. **紧凑/拼接**（技术）

连续分配方式的一个重要特点是：

程序/数据必须被装入一片连续的内存空间中。

现实是：

OS及程序在运行了一段时间后，内存空间被分割成了许多分区，这些分区可能既分散又很小。**图4-11 (a)**

问题是：

如果进程请求一个大的分区，这些分区中的任一个都不能满足要求，但它们容量的总和可以满足要求。如**图4-11 (b)所示**。由于这些分区没构成一个大的连续内存空间，所以，这些空间无法分配给一个作业。

解决方法：使用紧凑/拼接技术

将内存中作业移动到一端，各个空闲分区移动到另一端，从而拼接出一个大的空闲分区。

紧凑优点：能提供大的地址空间，对大作业有利，且存储器利用率提高。

紧凑缺点：移动使得程序/数据的绝对地址发生了变化，程序无法正常运行。如何解决这一问题呢？

解决办法：使用动态重定位技术，重新进行一次地址映射.

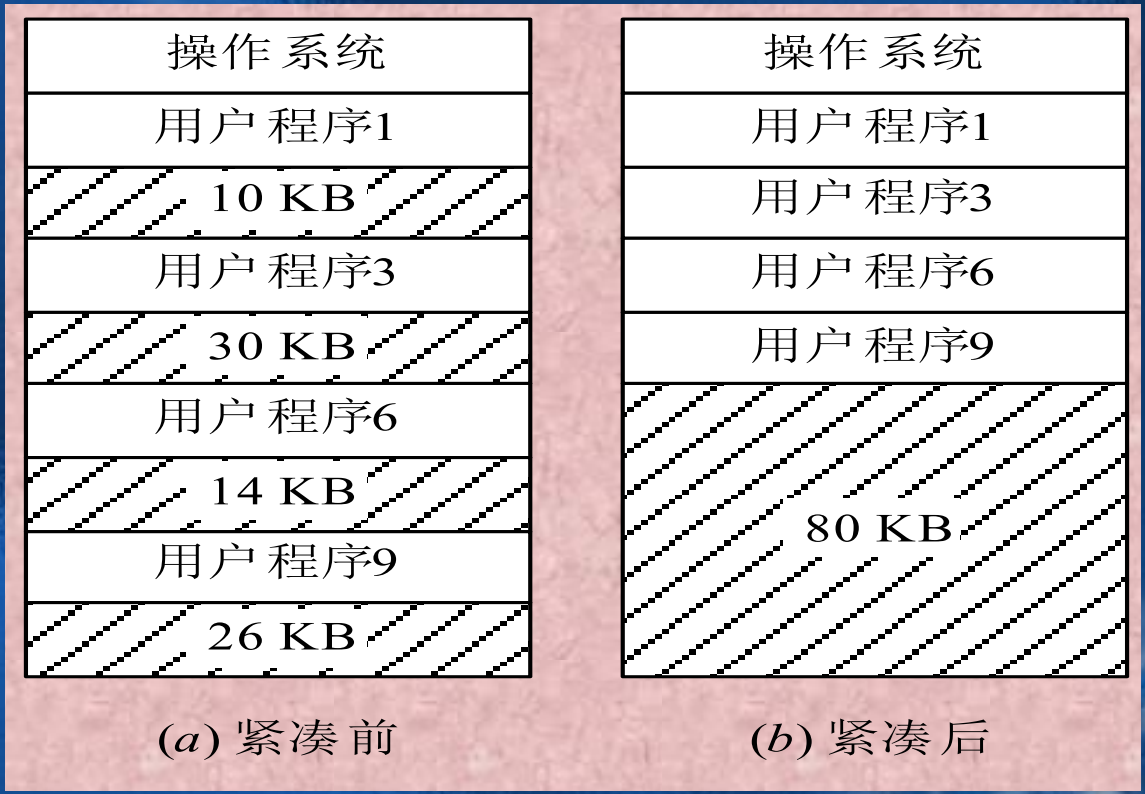


图 4-8 紧凑的示意

2. 动态重定位

在4.2.1节中所介绍的**动态运行时装入的方式**中，

- 作业装入内存后，地址仍然都是相对(逻辑)地址。而
- **地址映射**（将相对地址转换为绝对/物理地址）被推迟到程序运行时进行。
- 程序运行要求：先地址映射，后程序运行。

为避免地址映射对程序运行速度的影响，必须要有**硬件地址变换机构**的支持。即

- 在系统中设置一个**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，程序要访问的**内存地址**是：重定位寄存器中的起始地址+相对地址。 如图 4-12所示。

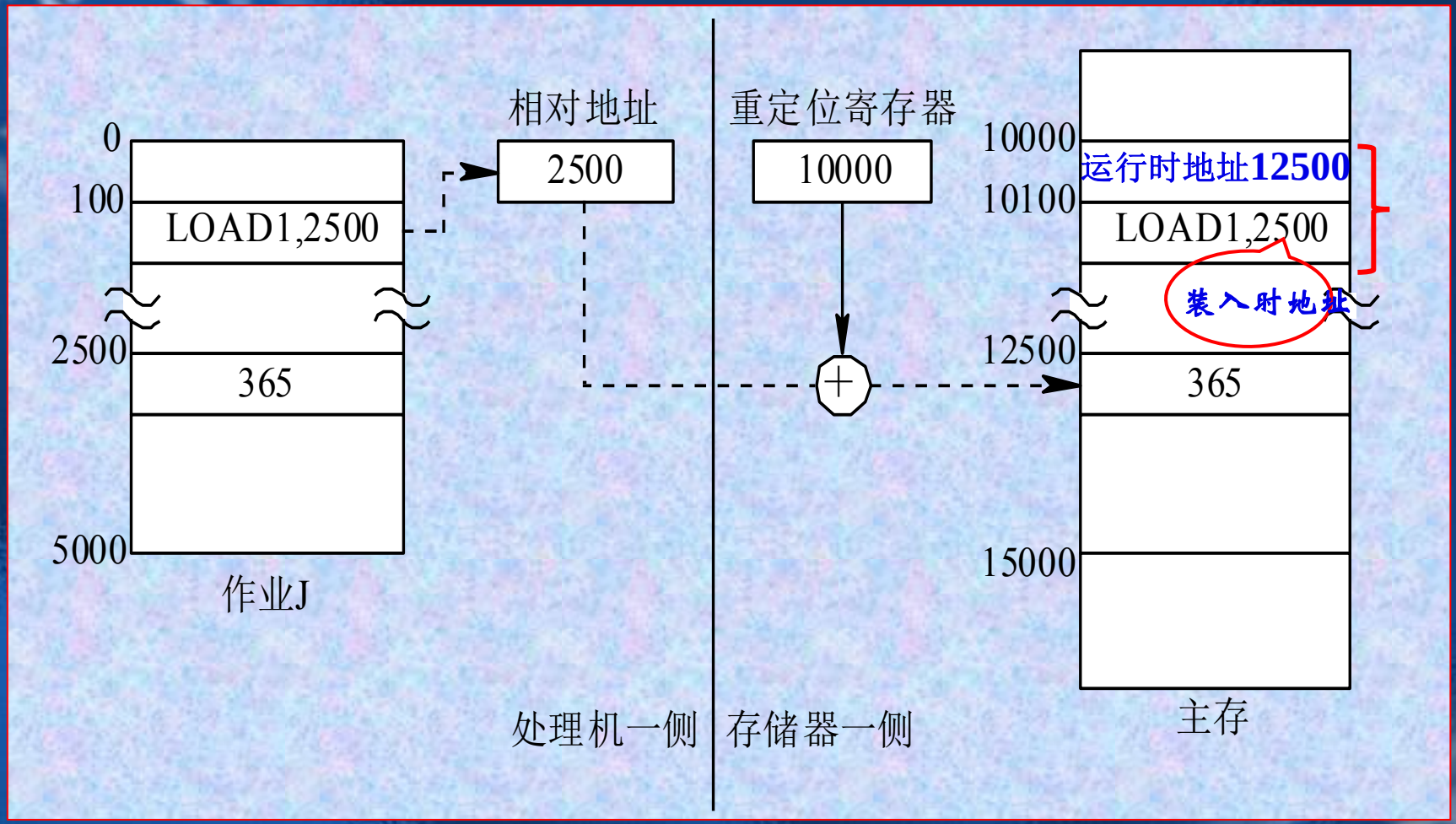


图 4-12 动态重定位示意图

地址变换是在程序运行期间进行的。

3. 动态重定位分区分配算法

动态重定位的分区分配算法

- 与动态分区的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。

如图4-13（下页）所示。

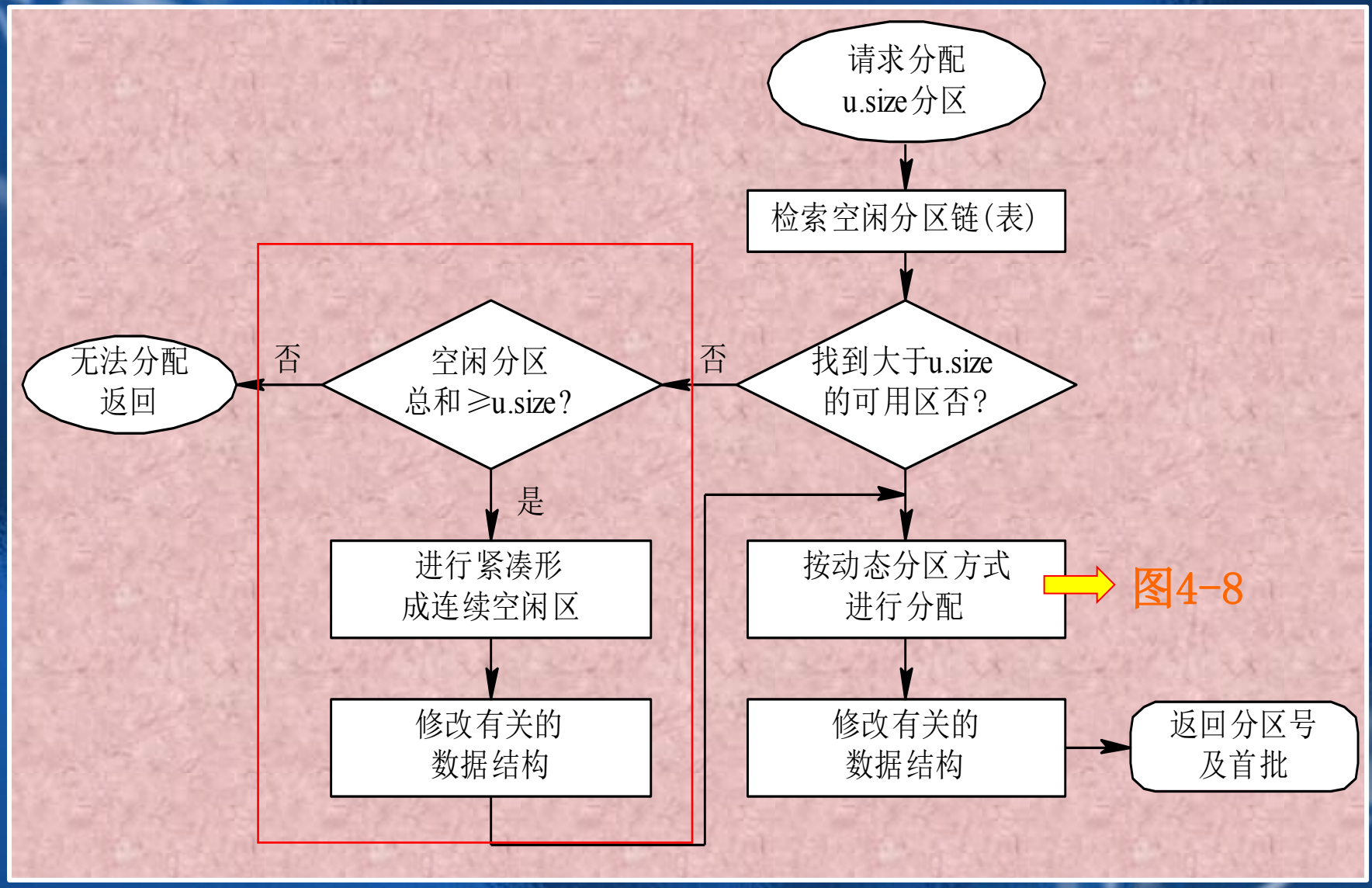


图 4-10 动态分区分配算法流程图

4.4 对换(Swapping)

对换技术也称为**交换技术**。

产生原因：由于早期计算机的内存都非常小，为了使该系统能分时运行多个用户程序而引入了对换技术。

对换的**目的**是：提高内存利用率。

对换， 是指把内存中暂时不能运行的程序/数据， 调出[1]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程序/数据， 调入[2]内存。

对换过程是：系统把所有的用户作业存放在磁盘上，每次只能调入一个作业进入内存。当该作业的一个时间片用完时，将它调至外存的后备队列上等待，再从后备队列上将另一个作业调入内存。这就是最早出现的分时系统中所用的对换技术。现在已经很少使用。

以下内容直到的4.5分页管理。（提示）

4.4.1 多道程序环境下的对换技术

1. 对换的引入

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，而无可运行之进程，迫使CPU停止下来等待的情况；另一方面，却又有着许多作业，因内存空间不足，一直驻留在外存上，而不能进入内存运行。显然这对系统资源是一种严重的浪费，且使系统吞吐量下降。

2. 对换的类型

在每次对换时，都是将一定数量的程序或数据换入或换出内存。根据每次对换时所对换的数量，可将对换分为如下两类：

(1) 整体对换/进程对换。

(2) 页面(分段)对换。

4.4.2 对换空间的管理

1. 对换空间管理的主要目标

在具有对换功能的OS中，通常把磁盘空间分为文件区和对换区两部分。

1) 对文件区管理的主要目标 第7章 文件管理

提高文件存储空间的利用率[1]，提高文件访问速度[2]。

对文件空间的管理，采取离散分配的方式。

2) 对对换空间管理的主要目标

该空间仅占磁盘空间的一小部分。

作用：用于存放从内存中换出的进程。

特点：进程在该区驻留时间短，对换频率高。因此，其

目标：提高换进换出速度首先；提高存储空间利用率其次。

方法：对存储空间采用连续分配方式。

2. 对换区空闲盘块管理中的数据结构

为了实现对对换区中的空闲盘块的管理，在系统中应配置相应的数据结构，用于记录外存对换区中的空闲盘块的使用情况。其数据结构的形式与内存在动态分区分配方式中所用数据结构相似，即同样可以用空闲分区表或空闲分区链。在空闲分区表的每个表目中，应包含两项：对换区的首址及其大小，分别用盘块号和盘块数表示。

3. 对换空间的分配与回收

由于对换分区的分配采用的是连续分配方式，因而对换空间的分配与回收与动态分区方式时的内存分配与回收方法雷同。其分配算法可以是首次适应算法、循环首次适应算法或最佳适应算法等。具体的分配操作也与图4-8中内存的分配过程相同。

4.4.3 进程的换出与换入

1. 进程的换出

对换进程在实现进程换出时，是将内存中的某些进程调出至对换区，以便腾出内存空间。换出过程可分为以下两步：

(1) 选择被换出的进程。

状态：阻塞、睡眠，+ 优先级低 + 内存中的驻留时间长

(2) 进程换出过程。 …自学

2. 进程的换入 （自学）

对换进程将定时执行换入操作，它首先查看PCB集合中所有进程的状态，从中找出“就绪”状态但已换出的进程。当有许多这样的进程时，它将选择其中已换出到磁盘上时间最久(必须大于规定时间，如2 s)的进程作为换入进程，为它申请内存。如果申请成功，可直接将进程从外存调入内存；如果失败，则需先将内存中的某些进程换出，腾出足够的内存空间后，再将进程调入。

4.5 分页存储管理方式

在前述的可变分区管理中，各分区都是连续的存储空间。

问题是：

1. 有碎片（存储效率下降）；

2. 通过“紧凑”技术可以减少碎片，但增加了系统的额外开销（CPU效率下降）。

解决思路：将程序/数据分散/离散存放。

离散分配有如下三种方式：

(1) 分页存储管理方式。

(2) 分段存储管理方式。

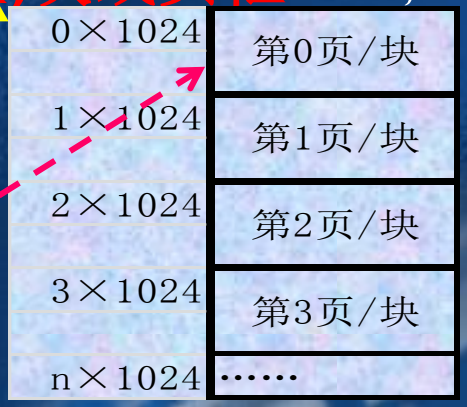
(3) 段页式存储管理方式。

4.5.1 分页存储管理的基本方法

1. 页面和物理块

（1）分页存储管理思想：

- 将一个程序的逻辑地址空间[1]分成若干个固定大小的区域，称为页面/页[2]；典型的页面大小[3]为1KB、2KB…；
  - 对各页加以编号[4]，编号/页号从0开始，如第0页、第1页等。
  - 页号对应了该页在逻辑空间中的起始位置[5]。
- 将内存空间[1]分成若干个存储块[2]，称为(物理)块或页框frame；
  - 块大小[3]与页面大小相同；
  - 各个块也有编号[4]，如0#块、1#块等等。
  - 块号对应了该块在物理空间中的起始位置[5]。
- 在为进程分配内存时，以块为单位[1]。
  - 将进程中的若干个页，分别装入到不相邻接的物理块中[2]。
  - 但进程的最后一页通常装不满一块,因此形成了页内碎片[3].



（2） 页面大小

在分页系统中，页面大小应适中。

页面若太小[1]，一方面虽然可以使页内碎片减小(1)，内存利用率提高优点。但小的页面，会使进程占用更多的页，导致进程的页表过长(2)，过长的页表又占用了大量的内存缺点(得不偿失)；此外，还会降低页面换进换出的效率缺点。

页面若太大[2]，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，浪费内存。因此，

页面的大小应选择得适中[1]，且页面大小应是2的幂[2]，通常为512 B~8 KB。

2. 地址结构

分页地址中的地址结构如下（例）：

分为两个部分：页号(20位)+页内位移(12位)/页内地址

地址空间中，可以有2^{20}个页，每个页大小为2^{12}。

对某特定机器，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：

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

$$d = [A]MODL$$

3. 页表

在分页系统中，允许将进程的各个页**离散地存储**在内存的任一物理块中。

进程运行时，需要先进行**地址映射**，将逻辑地址转换为物理地址。即：

进程空间中的哪一个页面，被放在了内存空间中的什么地方（即：哪一个物理块中），所以：

系统为每个进程建立了一张页面映像表，简称**页表**。

页表就是页面页号与物理块块号之间的**映像表**。

如图4-14所示。

另外，页表中也可以为每一项设置控制字段，用于对存储块内容的保护。例如，1位的控制字段，可用来表示：可读写或只读，2位的控制字段，可增加表示：只执行等。

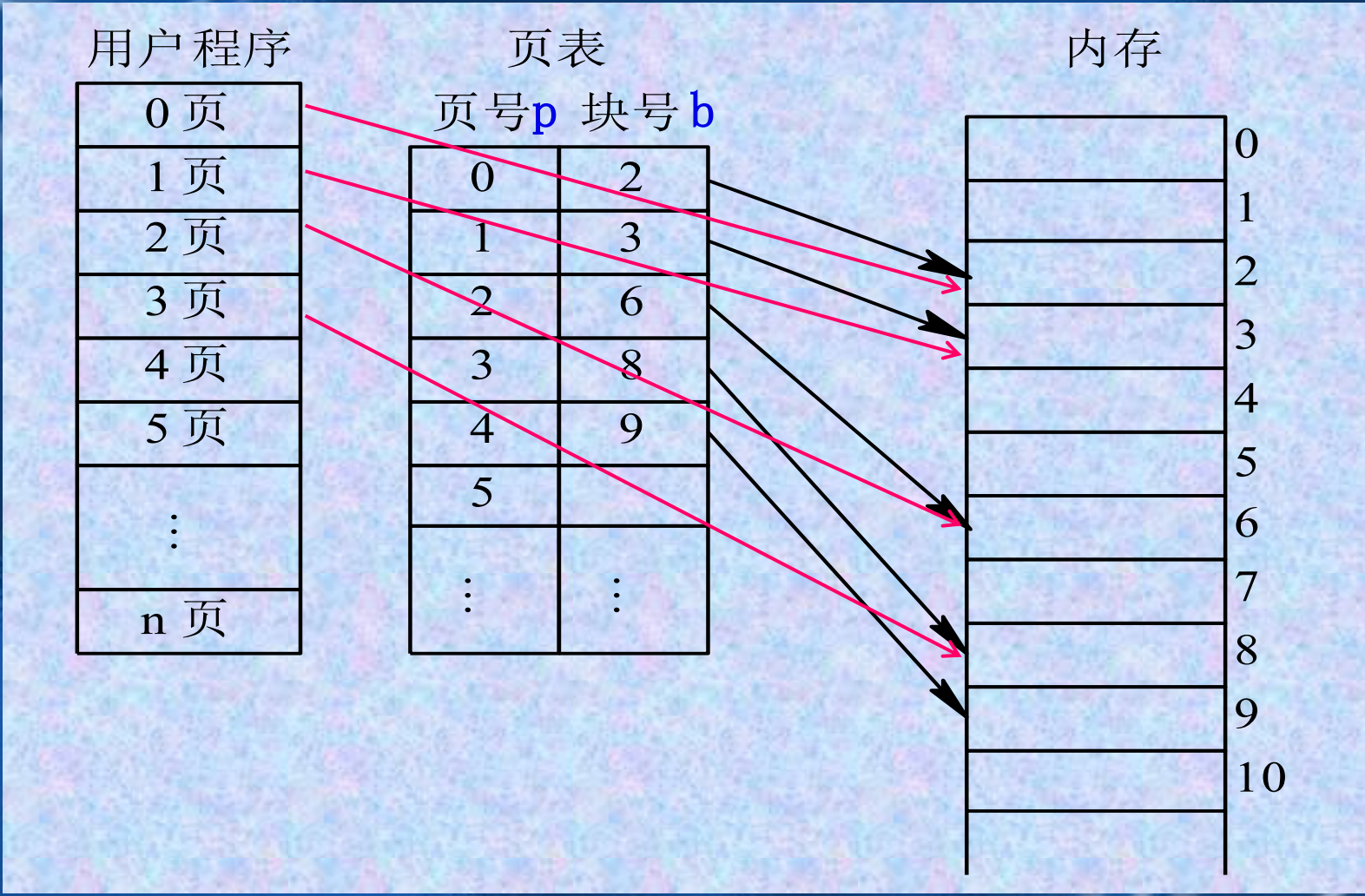

用户程序
0 页
1 页
2 页
3 页
4 页
5 页
n 页
页表
页号p 块号 b
0 2
1 3
2 6
3 8
4 9
5
内存
0
1
2
3
4
5
6
7
8
9
10

图 4-14 页表的作用

4.5.2 地址变换机构

1. 基本的地址变换机构

进程在运行期间，需要将用户地址空间中的逻辑地址变换为内存空间中的物理地址。

由于每条指令的地址都需要进行变换，变换的频率非常高，因此**需要采用硬件来实现**。

页表功能是由一组专门的寄存器来实现的。一个页表项用一个寄存器。

问题是：页表很大，需要很多寄存器；寄存器虽然快，但**成本高**，所以，需要：

页表驻留在内存中；运行时，将页表在内存的起始地址[1]及页表长度[2]（平时在PCB中）放在寄存器中。

地址映射如图4-15所示。

分页系统的地址变换机构 ---5个步骤（由硬件完成）

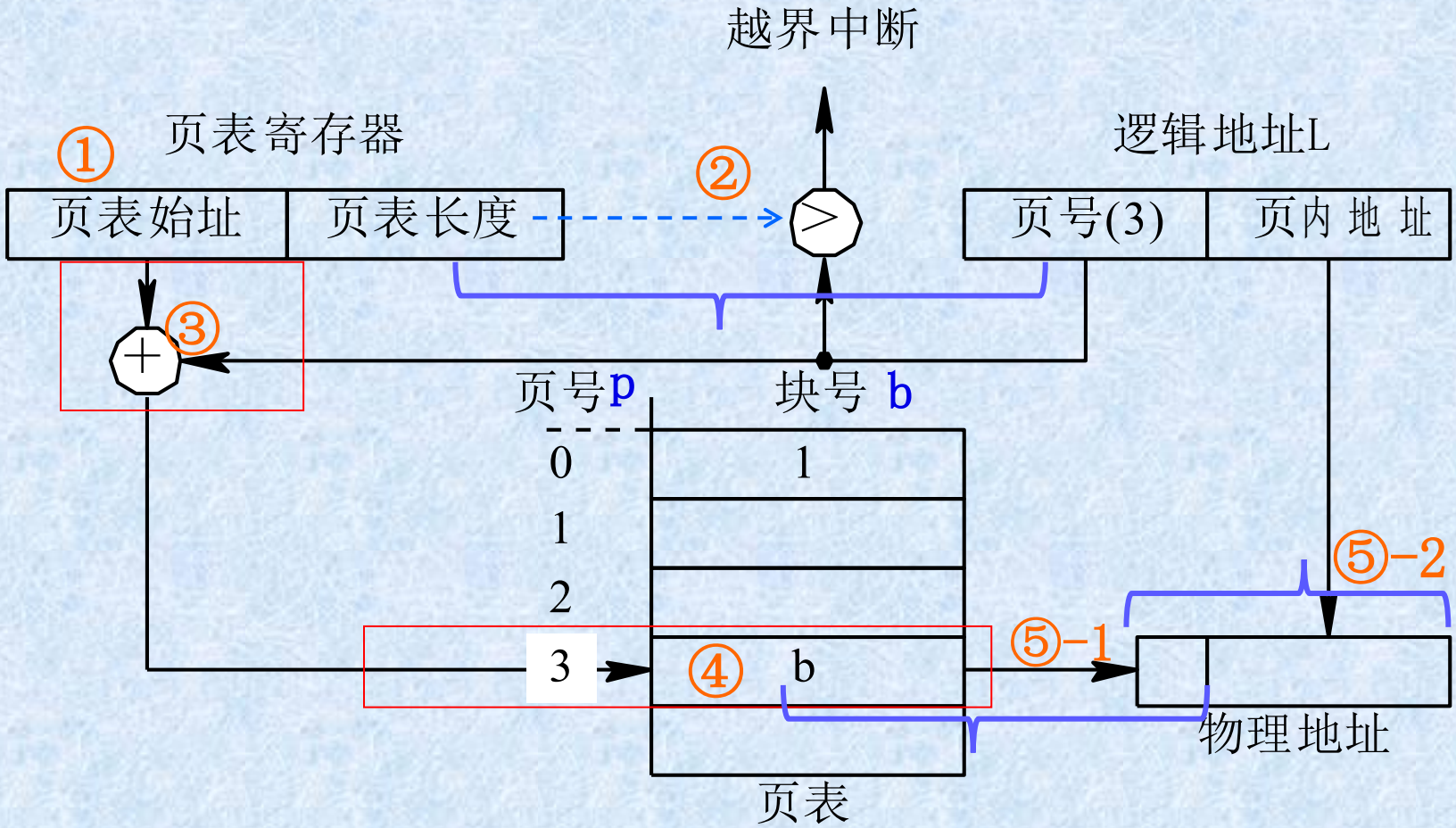


图 4-15 分页系统的地址变换机构

2. 具有快表的地址变换机构

（1）问题：

由于页表是存放在内存中的，这使得：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b（再将块号与页内偏移量W拼接），以形成物理地址。

第二次：根据物理地址访问内存，来读写指令/数据。

（2）结论：

①采用这种地址变换使得：计算机的处理速度降低到50%.

②分页管理使得：存储器利用率提高了，处理机效率下降了。

（3）**解决办法**：引入具有快表的地址变换机构。增加一个**快表/联想寄存器**，它是一个高速缓存，具有并行查找能力。

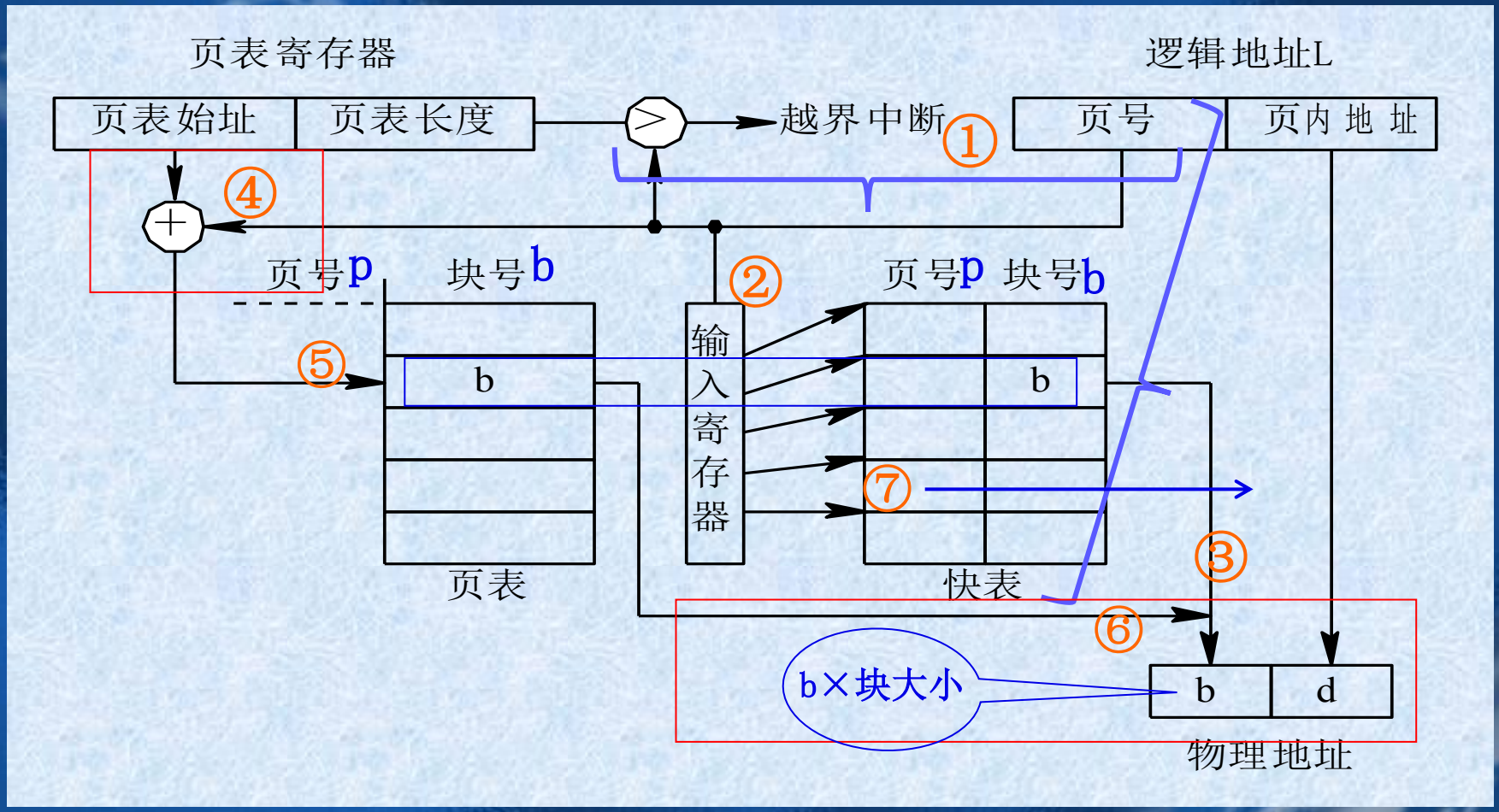


图 4-16 具有快表的地址变换机构

4.5.3 访问内存的有效时间 （+快，课后学习）

从进程发出指定逻辑地址的访问请求，经过地址变换，到在内存中找到对应的实际物理地址单元并取出数据，所需要花费的总时间，称为内存的**有效访问时间**(Effective Access Time，EAT)。假设访问一次内存的时间为t，在基本分页存储管理方式中[1]，有效访问时间分为第一次访问内存时间(即查找页表对应的页表项所耗费的时间t)与第二次访问内存时间(即将页表项中的物理块号与页内地址拼接成实际物理地址所耗费的时间t)之和：

$$EAT = t + t = 2t$$

在引入快表的分页存储管理方式中[2]，通过快表查询，可以直接得到逻辑页所对应的物理块号，由此拼接形成实际物理地址，减少了一次内存访问，缩短了进程访问内存的有效时间。但是，由于快表的容量限制，不可能将一个进程的整个页表全部装入快表，所以在快表中查找到所需表项存在着命中率的问题。所谓命中率，是指使用快表并在其中成功查找到所需页面的表项的比率。这样，在引入快表的分页存储管理方式中，有效访问时间的计算公式即为：

$$EAT=a\times\lambda+(t+\lambda)(1-a)+t=2t+\lambda-t\times a$$

上式中，λ表示查找快表所需要的时间，a表示在快表中的命中率，t表示访问一次内存所需要的时间。

可见，引入快表后的内存有效访问时间分为查找到逻辑页对应的页表项的平均时间$a \times \lambda + (t + \lambda)(1 - a)$，以及对应实际物理地址的内存访问时间t。假设对快表的访问时间λ为20 ns(纳秒)，对内存的访问时间t为100 ns，则下表中列出了不同的命中率a与有效访问时间的关系：

命中率 (%) a	有效访问时间 EAT
0	220
50	170
80	140
90	130
98	122

4.5.4 两级和多级页表

问题：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}~2^{64})。在这样的环境下，页表要占用相当大的内存空间-----新问题。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4KB/2^{12}B（页内地址占12位，页号占20位），
- 则在每个进程页表中的页表项可达1兆个（ 2^{20} 个）。
- 假设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1MB的内存空间，而且还要求是连续的。

可以采用两个方法来解决页表过大且要求连续的问题：

① 采用离散分配方式存放页表两级/多级 页表；

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，需要时再调入第5章。

1. 两级页表(Two-Level Page Table)

方法：

- 将页表再划分为若干页：页的大小与物理块的大小**相同**[1]，并为它们进行**编号**[2]，即依次为0# 页、1# 页，…，n# 页，然后
- 离散地将各个页存放在不同的物理块中。同时，
- 为这些页再建立一张页表，称为**外层页表**，它的每个表项记录了：每个页存放在哪一个物理块中。

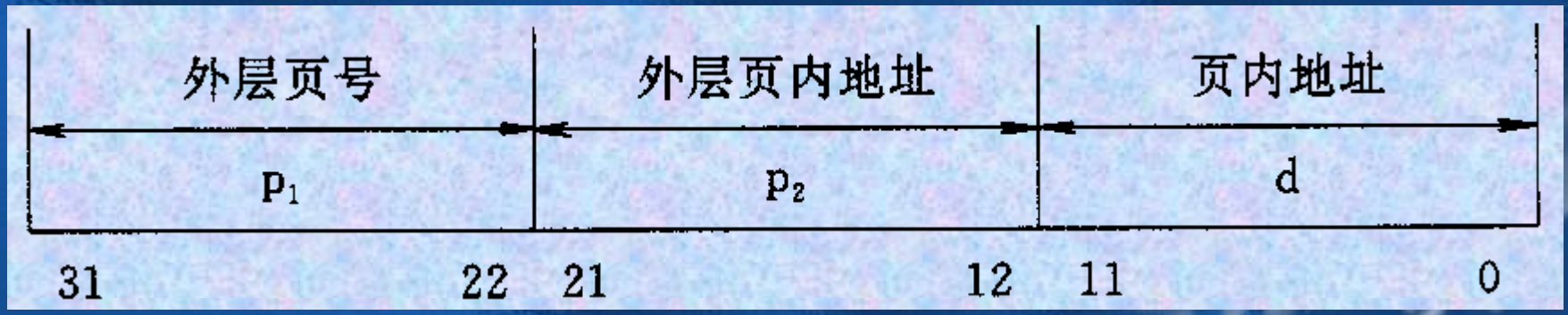


图4-17　两级页表结构　(例：32位逻辑地址)

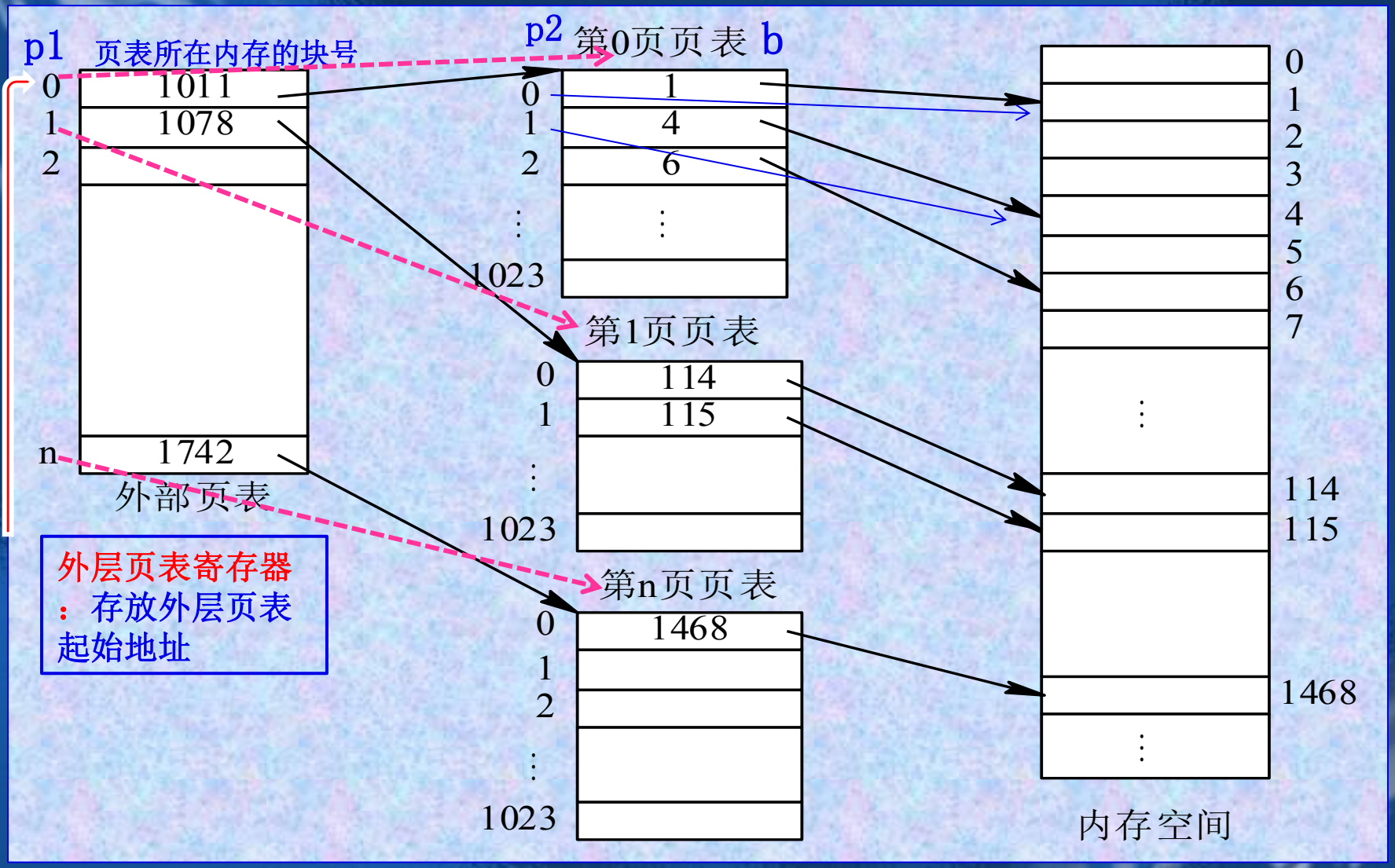


图4-18 (a)　具有两级页表的地址变换机构

优点： 页表分散存放，勿需连续的大内存空间。

问题： 但并没有节省内存。

解决： 对正在运行的进程，将外层页表[1]调入内存，将页表[2]的1页或几页调入内存节省了内存。为此，

在外层页表项中，增加状态位：0—不在内存，产生中断信号，请求OS调入此页第5章 虚存管理；1—在内存。

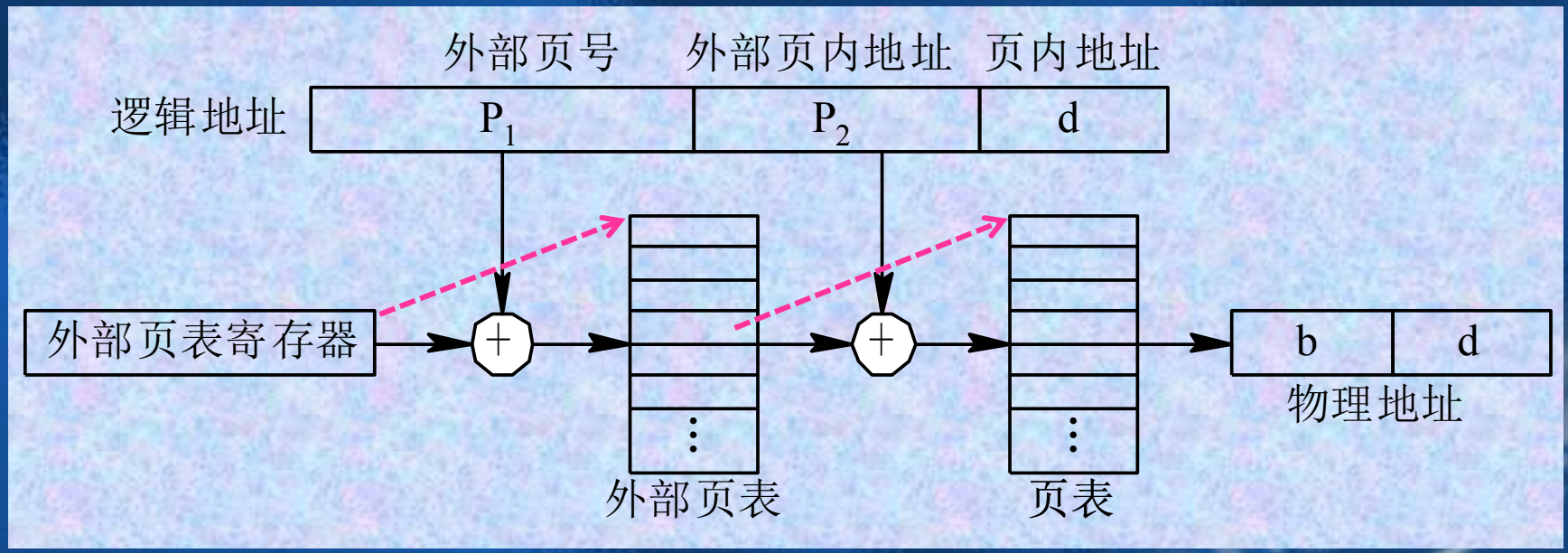


图4-18（b） 具有两级页表的地址变换机构

2. 多级页表 (课后学习)

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，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16384 GB的连续内存空间。 必须采用多级页表，将外层页表再进行分页，也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。

对于64位的计算机，如果要求它能支持2^{64}(=1844744 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。

4.5.5 反置页表(Inverted Page Table)

（课后学习，属于考试内容）

1. 反置页表的引入

在分页系统中，为每个进程配置了一张页表，进程逻辑地址空间中的每一页，在页表中都对应有一个页表项。在现代计算机系统中，通常允许一个进程的逻辑地址空间非常大，因此就需要有许多的页表项，而因此也会占用大量的内存空间。

2. 地址变换

在利用反置页表进行地址变换时，是根据进程标识符和页号，去检索反置页表。如果检索到与之匹配的页表项，则该页表项(中)的序号i便是该页所在的物理块号，可用该块号与页内地址一起构成物理地址送内存地址寄存器。若检索了整个反置页表仍未找到匹配的页表项，则表明此页尚未装入内存。对于不具有请求调页功能的存储器管理系统，此时则表示地址出错。对于具有请求调页功能的存储器管理系统，此时应产生请求调页中断，系统将把此页调入内存。

4.6 分段存储管理方式

存储管理方式随着OS的发展也在不断地发展。

当OS由单道向多道发展时，存储管理方式便由单一连续分配发展为固定分区分配。为提高内存利用率[1]，内存分配方式：

- 由固定分区分配发展到动态分区分配。（连续分配）
- 由连续分配发展离散分配/分页管理方式。（不连续分配）
- 为了满足用户/程序员的编程要求，引入了分段管理方式。

连续分配+不连续分配----是用户的需要[2]

许多高级语言都支持这种分配方式。

4.6.1 分段存储管理方式的引入原因

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**
  如：主程序段、若干子程序段、数据段、栈段……
- **每个段**都从0开始编址，并有自己的名字和长度。因此，
- 程序员希望用到的**逻辑地址格式**是：段名(段号)+段内偏移量(段内地址)。这种地址格式的

优点是：既方便程序员编程[1]，也使得程序非常直观，且可读性好[2]。

例如，下述的两条指令便使用了段名和段内地址：

LOAD 1，[A]|〈D〉；将段A中的D单元读入寄存器1中

STORE 1，[B]|〈C〉；将寄存器1中值存入段B的C单元中

2. 信息共享

对程序和数据的**共享**，是以信息的**逻辑单位**为基础的。比如，共享某个过程、函数或文件。

段就是信息的逻辑单位。但是，

分页系统中的“**页**”，（1）只是一个连续的数据块，并无完整的逻辑意义。而且

（2）共享数据可能需要占用多个页面，也可能仅占用不到一页的空间，所以，难以实现信息共享---例子：图4-21。

解决办法：采用**分段机制**来实现共享。

3. 信息保护 （课后）

信息保护也是以信息的逻辑单位为基础的，而且经常是以一个过程、函数或文件为基本单位进行保护的。

4. 动态增长 （课后）

在实际应用中，往往存在着一些段，尤其是数据段，在它们的使用过程中，由于数据量的不断增加，而使数据段动态增长，相应地它所需要的存储空间也会动态增加。然而，对于数据段究竟会增长到多大，事先又很难确切地知道。对此，很难采取预先多分配的方法进行解决。

5. 动态链接 （课后）

在4.2.2节中我们已对运行时动态链接做了介绍。为了提高内存的利用率，系统只将真正要运行的目标程序装入内存，也就是说，动态链接在作业运行之前，并不是把所有的目标程序段都链接起来。当程序要运行时，首先将主程序和它立即需要用到的目标程序装入内存，即启动运行。而在程序运行过程中，当需要调用某个目标程序时，才将该段(目标程序)调入内存并进行链接。

4.6.2 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，它占用了一个连续的地址空间。

例如，有主程序段MAIN、子程序段X、数据段D及栈段S等，如图4-19所示下页。

- 每个段有一个段号代替段名，段号从0开始，连续编址；
- 段长由逻辑信息决定；
- 段的逻辑地址可以表示为：

该地址结构表示：允许2^{12}个段，每个段的长度≤ 2^{20} B。

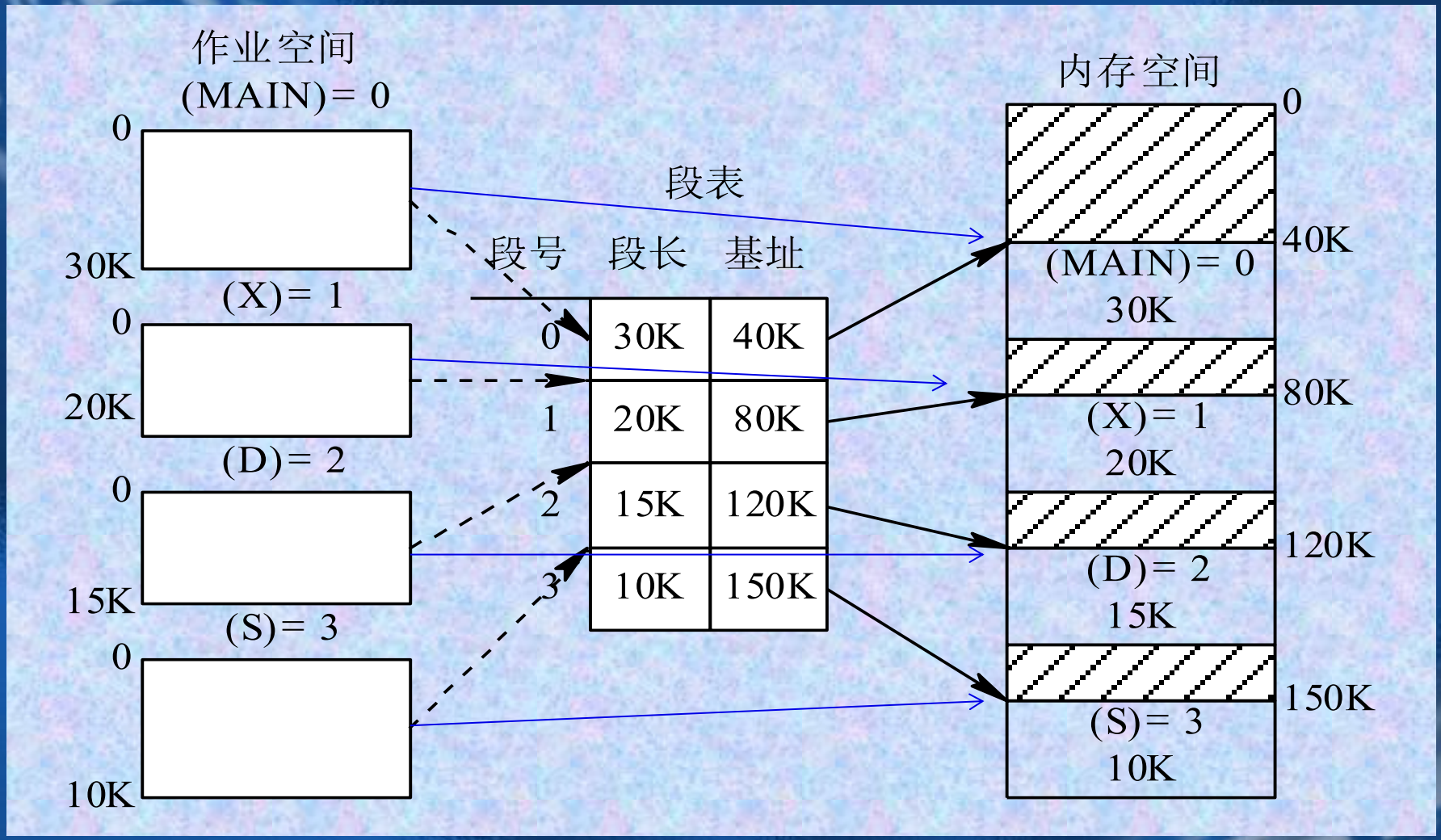


图 4-19 利用段表实现地址映射

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。

而在分段存储管理系统中，则是为每个段分配一个连续的分区。因此，

① 类似于分页管理，进程中的各个段，被离散地装入到内存中。

② 为每个进程建立一个段映射表---段表。段表可以存放在一组**寄存器**快+贵中，也可以存放在**内存**慢+便宜中。

③ 每个段占一个表项，表项中包含了：**段号、段长、基址**（段在内存中的起始位置）。如图4-19所示。

④ 程序在运行时，就可以通过查段表完成地址映射，即：从逻辑地址到物理地址的映射。

3. 地址变换机构

为了实现进程从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(完全类似于：页表寄存器)。变换过程如下：（见图4-20）

① 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

② 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

③ 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。

图4-20示出了分段系统的地址变换过程。

问题：访问一个数据需要访问内存两次。解决：高速缓存。

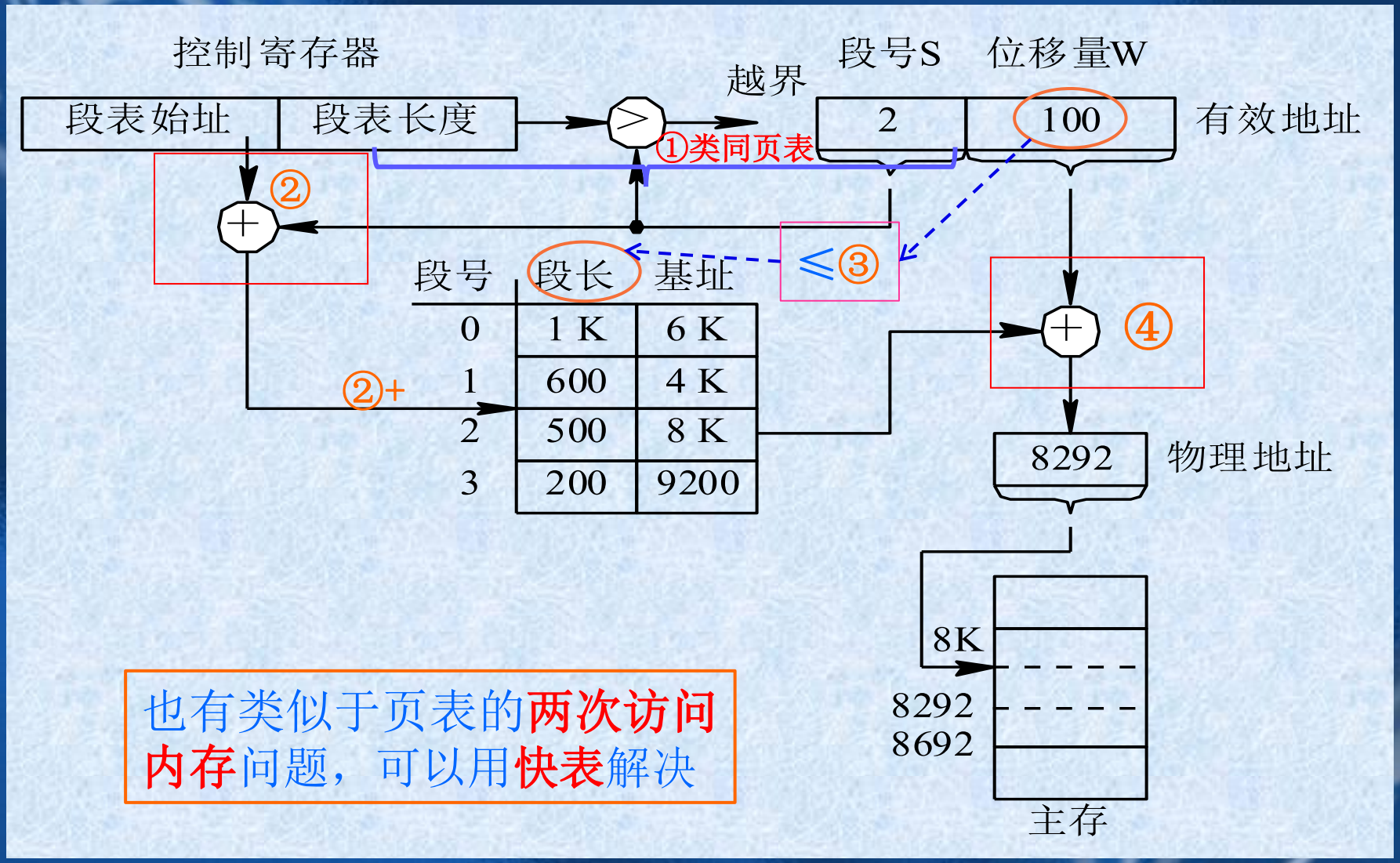


图 4-20 分段系统的地址变换过程

4. 分页和分段的主要区别 （+快---课后）

(1) 页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外零头，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**。

(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，**由用户所编写的程序决定**，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。

4.6.3 信息共享 （+快）

1. 分页系统中对程序和数据的共享

在分页系统中，虽然也能实现对程序和数据的共享，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：某多用户系统，假定：

- 有40个用户；
- 它们共享文本编辑程序（160KB）；
- 它们各自的数据区有40KB的数据；

每个进程空间占用：160KB+40KB=200KB ；

40个用户总的进程空间：40×200KB=8MB；

- 每个页4KB； 每个进程空间占用：200KB / 4KB =50页；
- 因为共享代码，40个进程所占内存空间是：

160KB(共享代码) + 40 × 40KB(数据) = 1.76MB (<8MB)

如图 4-21所示。

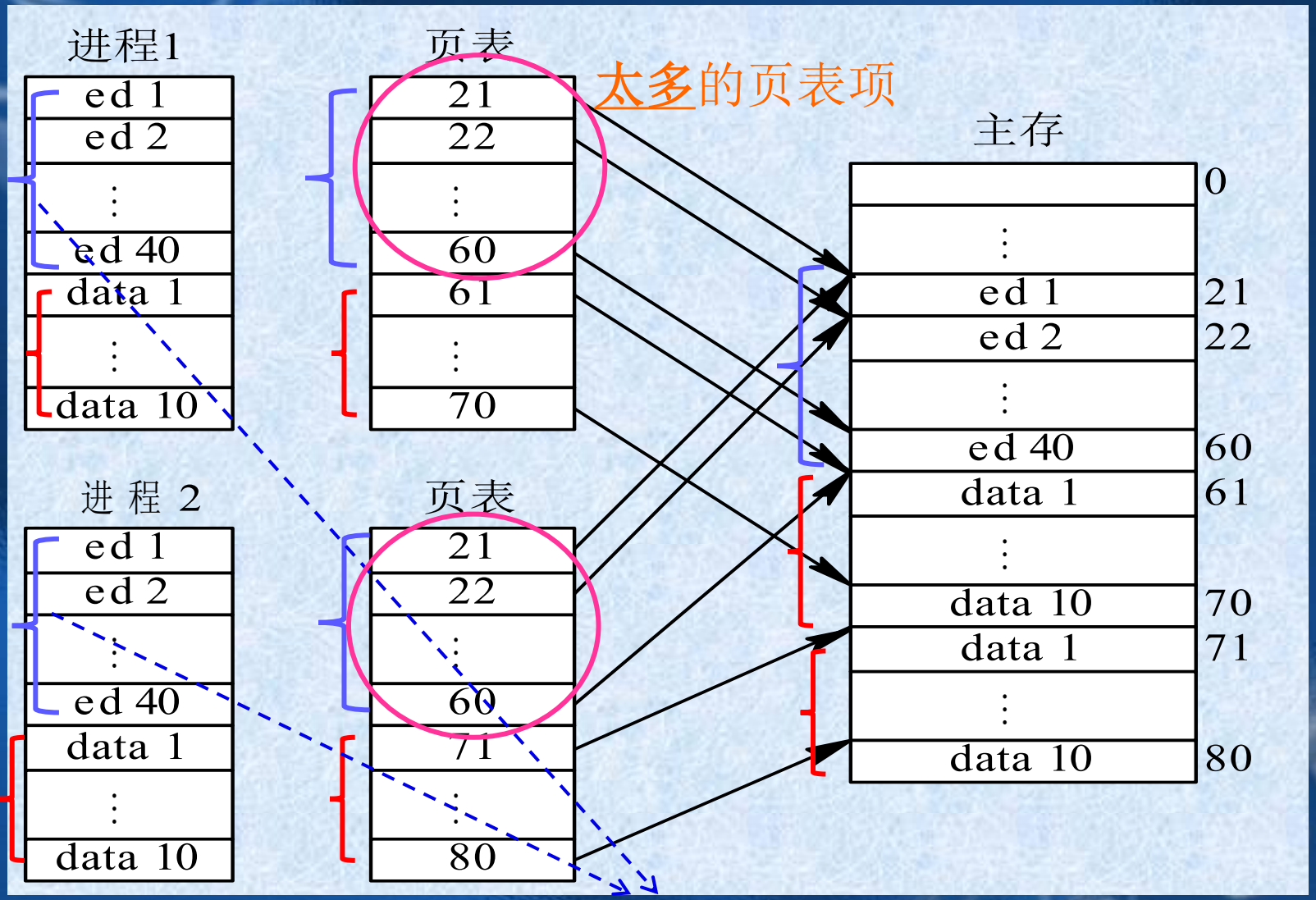


图4-21 分页系统中共享editor的示意图

2. 分段系统中程序和数据的共享

在分段系统中，由于是以段为基本单位的。不管该段有多大，只需为该段设置一个段表项，因此使实现共享变得非常容易。我们仍以共享editor为例，此时只需在(每个)进程1和进程2的段表中，为文本编辑程序设置一个段表项，让段表项中的基址(80)指向editor程序在内存的起始地址。

图4-22是分段系统中共享editor的示意图。

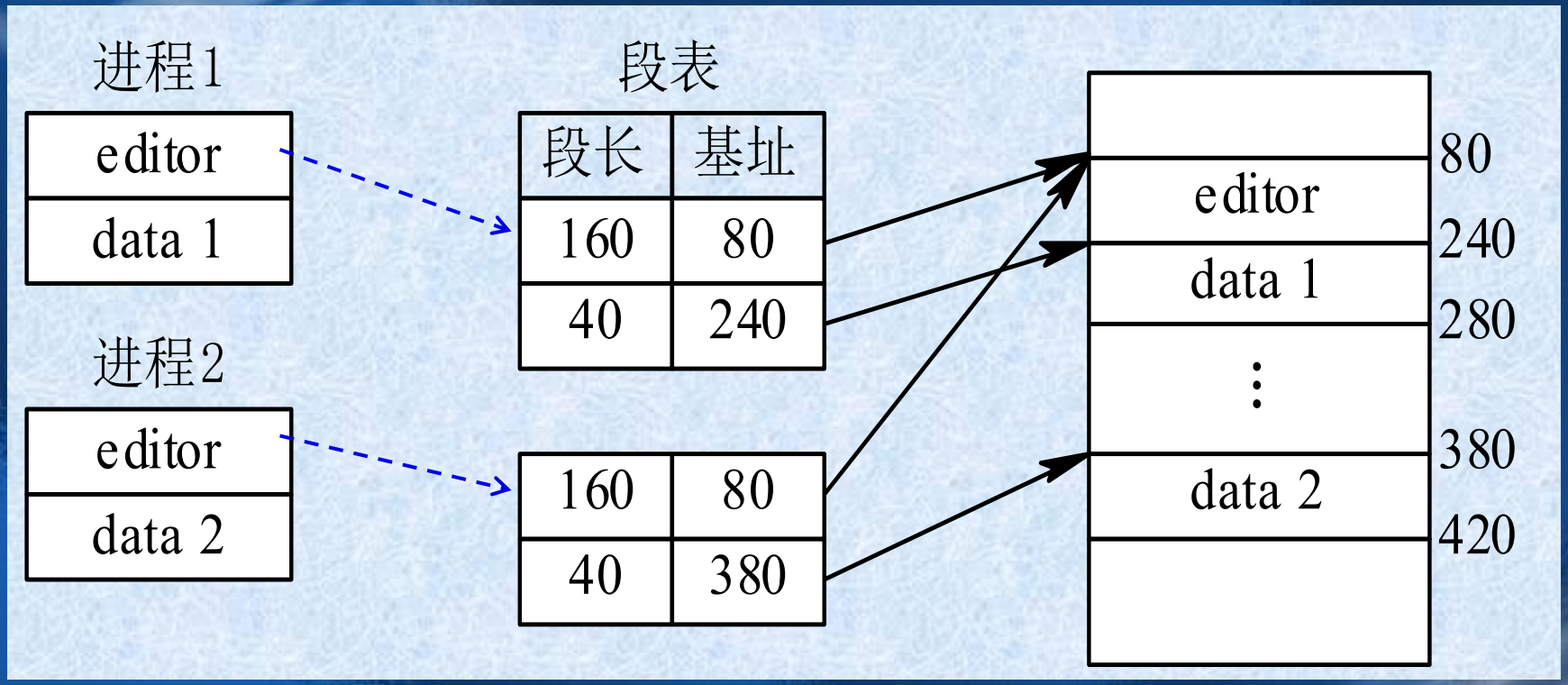


图 4-22 分段系统中共享editor的示意图

4.6.4 段页式存储管理方式

1. 基本原理

分页存储管理：以提高存储效率为目标；

分段存储管理：以方便用户为目标；

段页式存储管理：**分段和分页相结合，各取所长**。

怎样结合？

- 先将用户程序分成若干个段，每一个段有一个段名/号。
- 再把每个段分成若干个页。

图4-23(a) 是一个作业地址空间的结构。该作业有三个段：主程序段、子程序段和数据段；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4-23(b)所示。

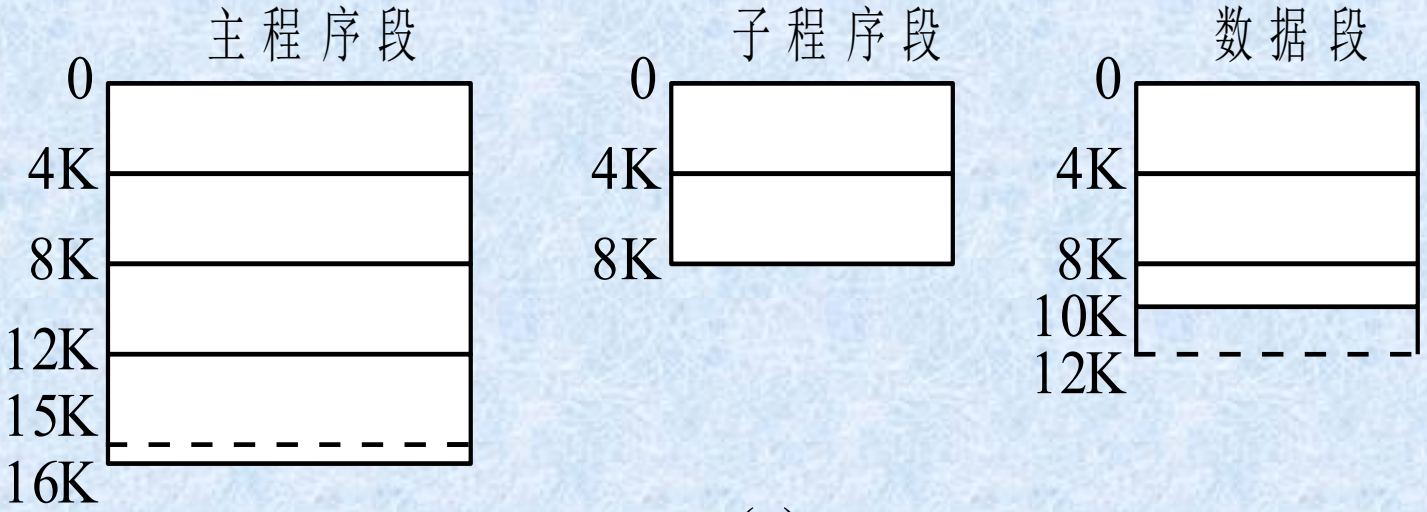


图4-23 作业地址空间和地址结构

为了实现从逻辑地址到物理地址的变换，系统中需要同时配置**段表**和**页表**。

如图4-24 所示。

段表的内容：与分段系统略有不同（它不是内存始址和段长），是页表始址和页表长度/页表大小。

地址映射：利用段表和页表可以将逻辑地址到物理地址。

如图4-25 所示。

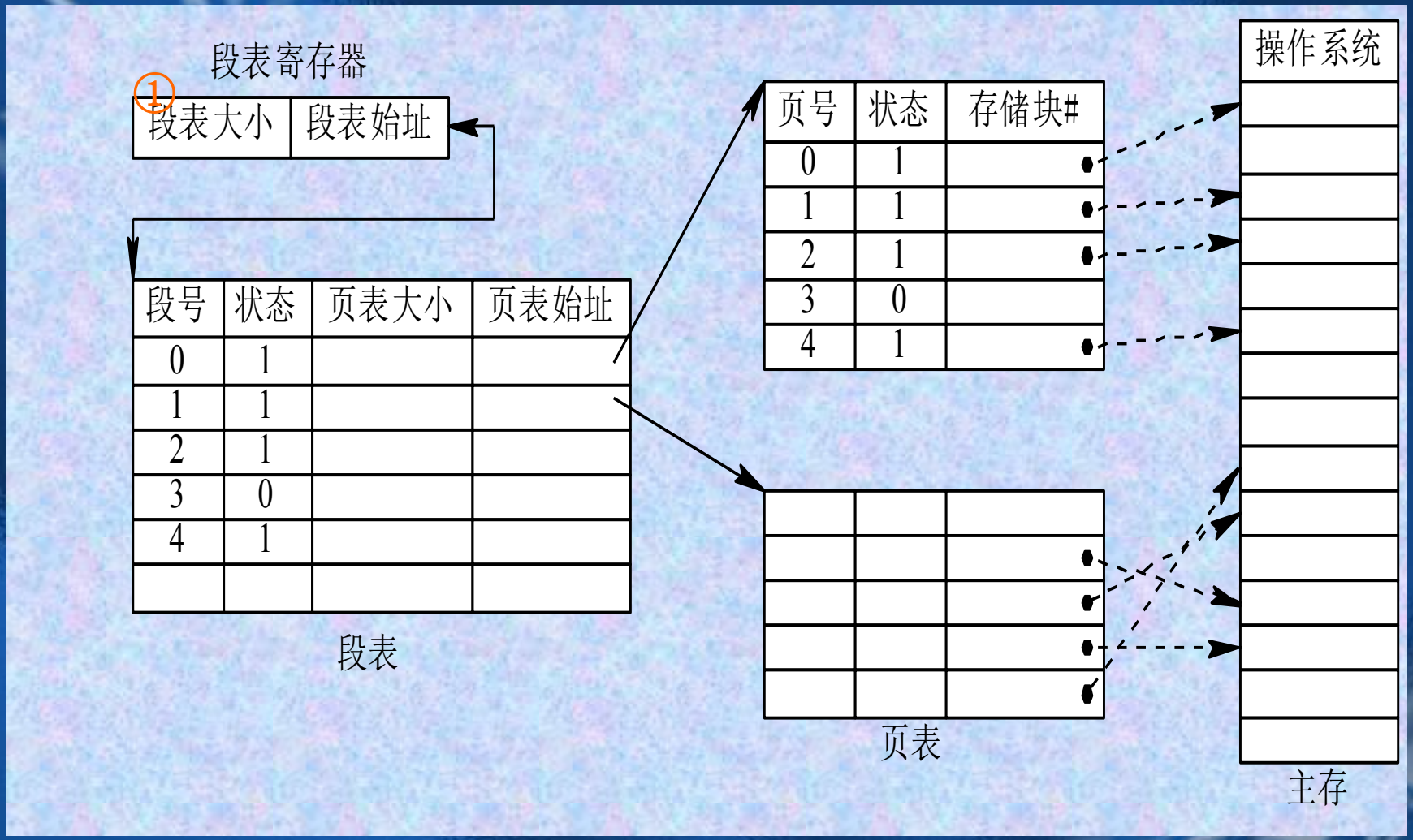


图 4-24 利用段表和页表实现地址映射

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长TL。进行地址变换时，首先利用段号S，将它与段长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4-25示出了段页式系统中的地址变换机构。

问题：访问一条指令/一个数据，需要访问内存3次。即：查段表->查页表->访问指令/数据。

解决：分别为段表、页表增加高速缓存。

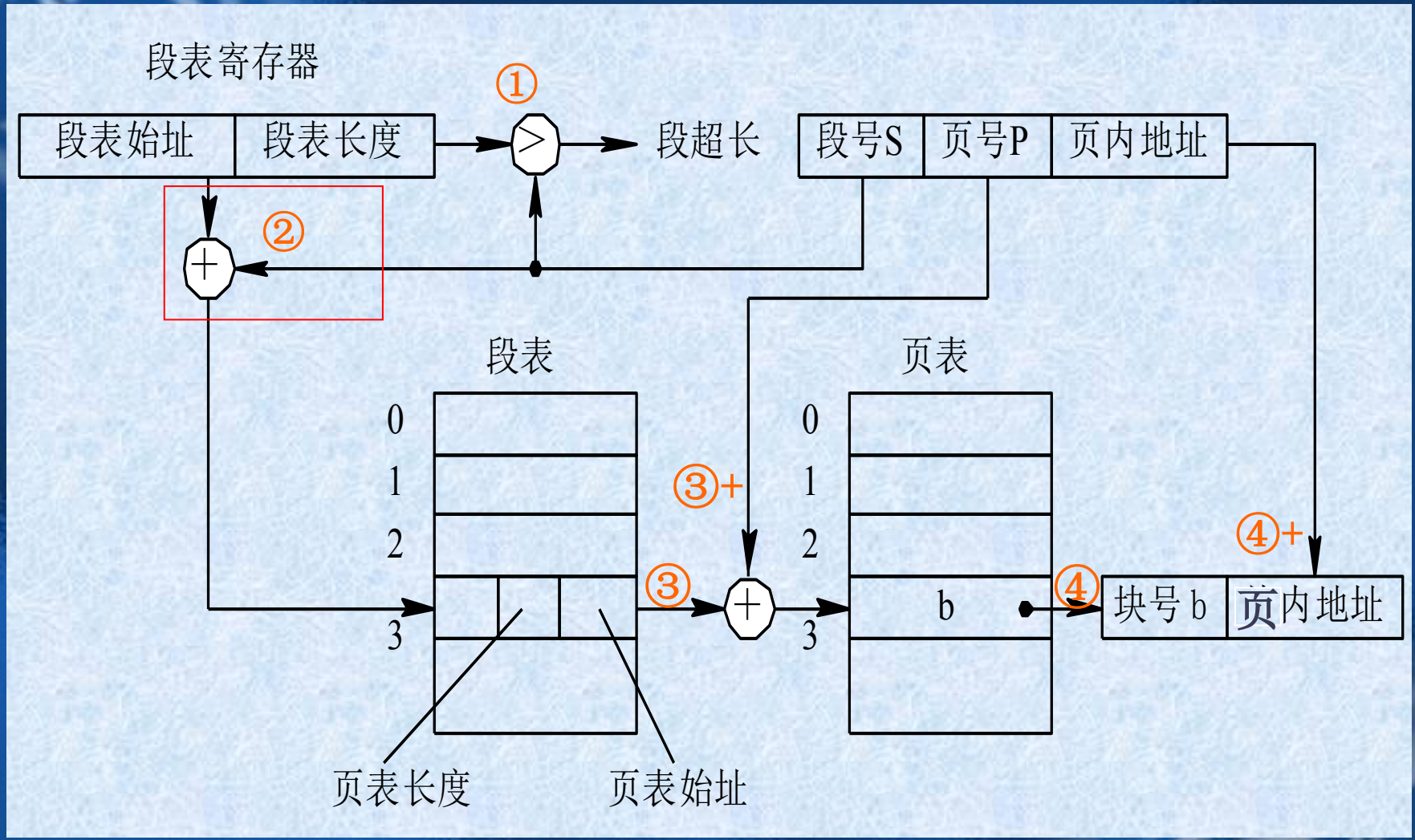


图 4-25 段页式系统中的地址变换机构

习　　题 ⏩⏭

1. 为什么要配置层次式存储器?

2. 可采用哪几种方式将程序装入内存? 它们分别适用于何种场合?

3. 何谓静态链接? 静态链接时需要解决两个什么问题?

4. 何谓装入时动态链接? 装入时动态链接方式有何优点?

5. 何谓运行时动态链接? 运行时动态链接方式有何优点?

6. 在动态分区分配方式中，应如何将各空闲分区链接成空闲分区链?

7. 为什么要引入动态重定位? 如何实现?

8. 什么是基于顺序搜索的动态分区分配算法? 它可分为哪几种?

9. 在采用首次适应算法回收内存时，可能出现哪几种情况? 应怎样处理这些情况?

10. 什么是基于索引搜索的动态分区分配算法? 它可分为哪几种?

11. 令buddyk(x)为大小为2k、地址为x的块的伙伴系统地址，试写出buddyk(x)的通用表达式。

12. 分区存储管理中常用哪些分配策略? 比较它们的优缺点。

13. 为什么要引入对换? 对换可分为哪几种类型?

14. 对文件区管理的目标和对对换空间管理的目标有何不同?

15. 为实现对换，系统应具备哪几方面的功能?

16. 在以进程为单位进行对换时，每次是否都将整个进程换出? 为什么?

17. 基于离散分配时所用的基本单位不同，可将离散分配分为哪几种?

18. 什么是页面? 什么是物理块? 页面的大小应如何确定?

19. 什么是页表? 页表的作用是什么?

20. 为实现分页存储管理，需要哪些硬件支持?

21. 在分页系统中是如何实现地址变换的?

22. 具有快表时是如何实现地址变换的?

23. 较详细地说明引入分段存储管理是为了满足用户哪几方面的需要。

24. 在具有快表的段页式存储管理方式中，如何实现地址变换?

25. 为什么说分段系统比分页系统更易于实现信息的共享和保护?

26. 分页和分段存储管理有何区别?

27. 试全面比较连续分配和离散分配方式。