

第四章 存储器管理

4.1 存储器的层次结构

4.2 程序的装入和链接

4.3 连续分配存储管理方式

4.4 对换(Swapping)

4.5 分页存储管理方式

4.6 分段存储管理方式

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，因此要求对存储器的访问速度能跟得上处理机的运行速度。或者说，**存储器的速度必须非常快**，能与处理机的速度相匹配，否则会明显地影响到处理机的运行。此外还要求**存储器具有非常大的容量**，而且存储器的**价格还应很便宜**。



4.1.1 多层结构的存储器系统

1. 存储器的多层结构

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层。如图4-1所示。

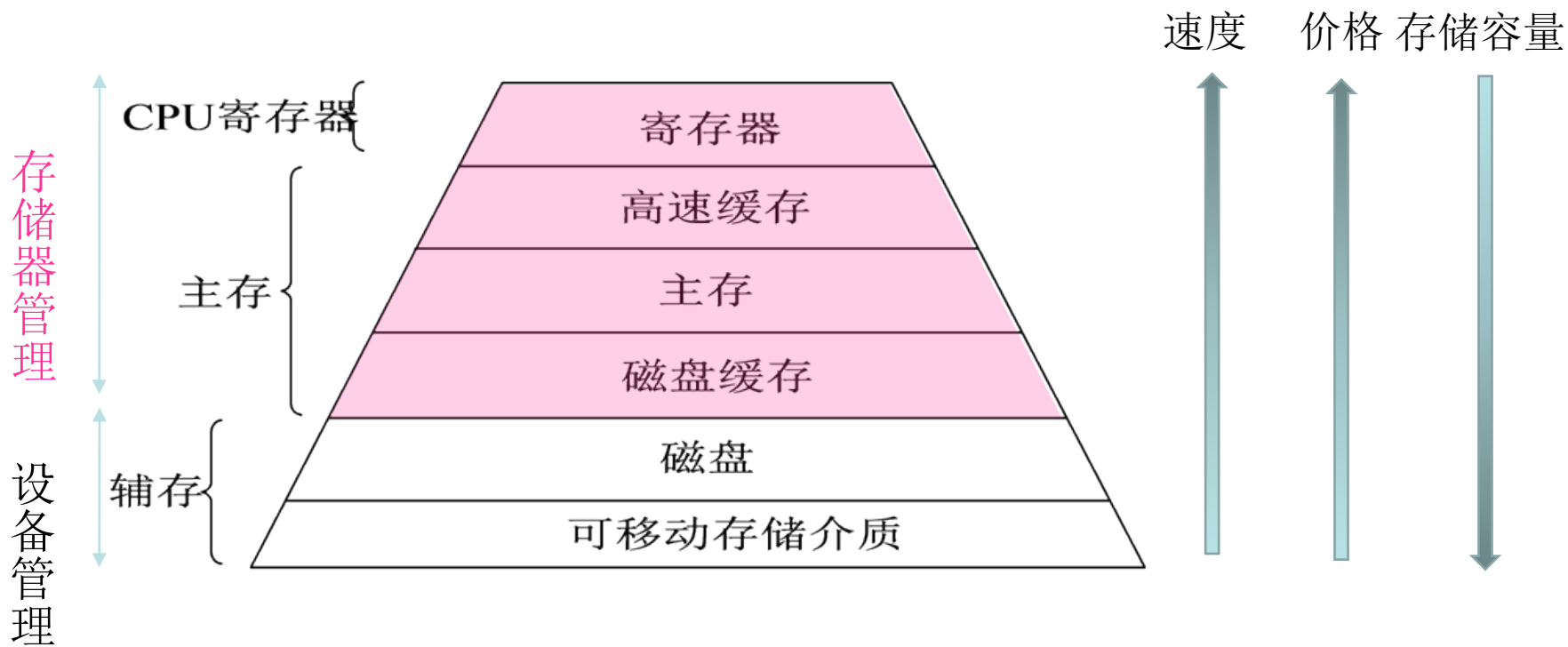


图4-1 计算机系统存储层次示意



2. 可执行存储器

在计算机系统的存储层次中，**寄存器和主存储器**又被称为**可执行存储器**。

对于存放在可执行存储器中的信息，与存放于辅存中的信息相比较而言，计算机所采用的访问机制是不同的，所需耗费的时间也是不同的。

★ 进程可以在**很少的时钟周期**内使用一条load或store指令对可执行存储器进行访问。

★ 对辅存的访问则需要**通过I/O设备实现**，因此，在访问中将涉及到中断、设备驱动程序以及物理设备的运行，所需耗费的时间远远高于访问可执行存储器的时间，一般相差3个数量级甚至更多。

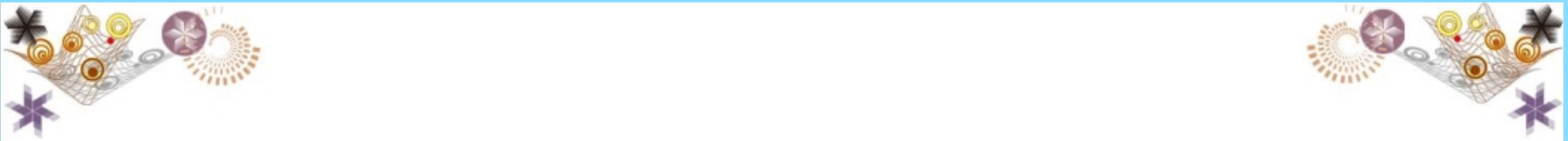


4.1.2 主存储器与寄存器

1. 主存储器

主存储器简称内存或主存，是计算机系统中的主要部件，用于保存进程运行时的程序和数据，也称可执行存储器。

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。



2. 寄存器

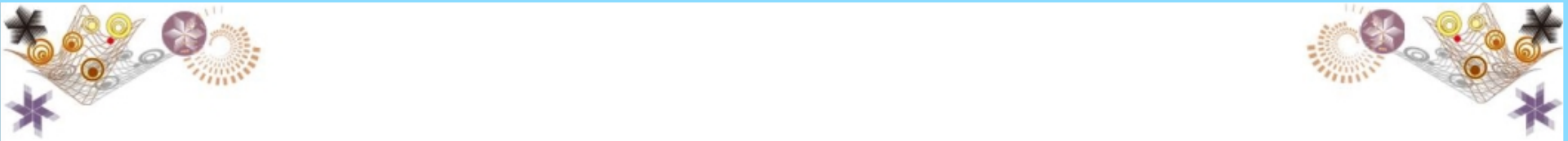
寄存器具有与处理机相同的速度，故对寄存器的访问速度最快，完全能与CPU协调工作，但价格却十分昂贵，因此容量不可能做得很大。

寄存器用于加速存储器的访问速度。例如：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。

寄存器的数目：

当前的微机系统和大中型机：几十个至上百个，字长一般是32位或者64位。

嵌入式计算机系统：一般几个到几十个，字长一般是8位。



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，它是介于寄存器和存储器之间的存储器，主要用于备份主存中较常用的数据，以减少处理机对主存储器的访问次数，这样可大幅度地提高程序执行速度。

提高执行速度的根据：**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，即在一较短的时间内，程序的执行仅局限于某个部分)。

高速缓存容量远大于寄存器，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。



通常，进程的程序和数据存放在主存储器中，**每当要访问时，才被临时复制到一个速度较快的高速缓存中。**

当CPU访问一组特定信息时，须**首先检查它是否在高速缓存**中，如果已存在，便可直接从中取出使用，以避免访问主存。否则，就须从主存中读出信息。

如大多数计算机都有**指令高速缓存**，用来暂存下一条将执行的指令，如果没有指令高速缓存，CPU将会空等若干个周期，直到下一条指令从内存中取出。

由于高速缓存的速度越高价格也越贵，故在有的计算机系统中设置了**两级或多级高速缓存**。紧靠内存的一级高速缓存的价格最高，而容量最小，二级高速缓存的容量稍大，速度也稍低。

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，为了缓和两者之间在速度上的不匹配，而设置了磁盘缓存，主要用于暂时存放频繁使用的一部分磁盘数据和信息，以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器，而是利用主存中的部分存储空间暂时存放从磁盘中读出(或写入)的信息。

主存也可以看作是辅存的高速缓存，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。

4.2 程序的装入和链接

用户程序要在系统中运行，必须先将它装入内存，然后再将其转变为一个可以执行的程序，通常都要经过以下几个步骤：

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

(2) **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)；

(3) **装入**，由装入程序(Loader)将装入模块装入内存。

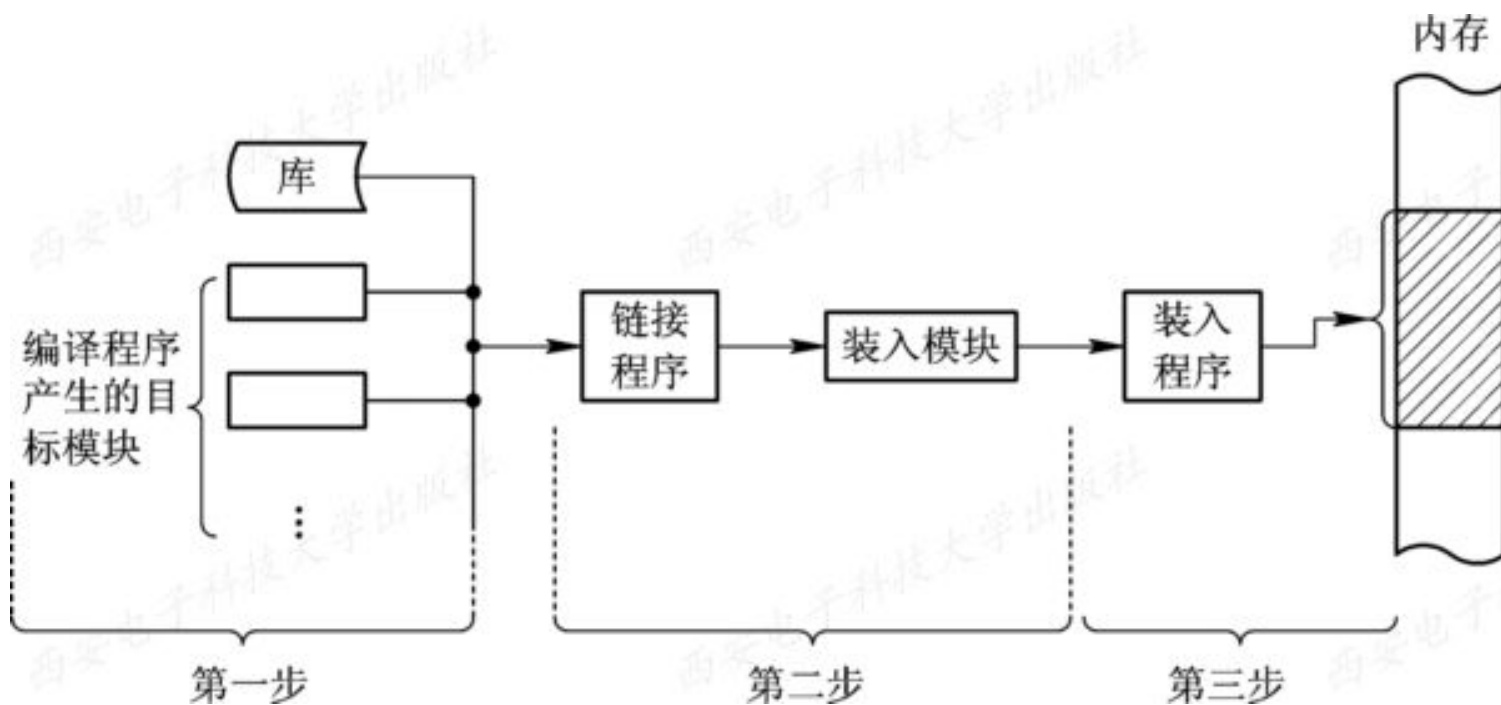


图4-2 对用户程序的处理步骤

4.2.1 程序的装入

有三种装入方式：

1. 绝对装入方式(Absolute Loading Mode)

当计算机系统很小，且仅能运行单道程序时，完全有可能知道程序将驻留在内存的什么位置。此时可以采用绝对装入方式。用户程序经编译后，将产生绝对地址（即物理地址）的目标代码。

在编译时，知道程序将驻留在内存的什么位置→产生绝对地址的目标代码→将程序和数据装入内存。

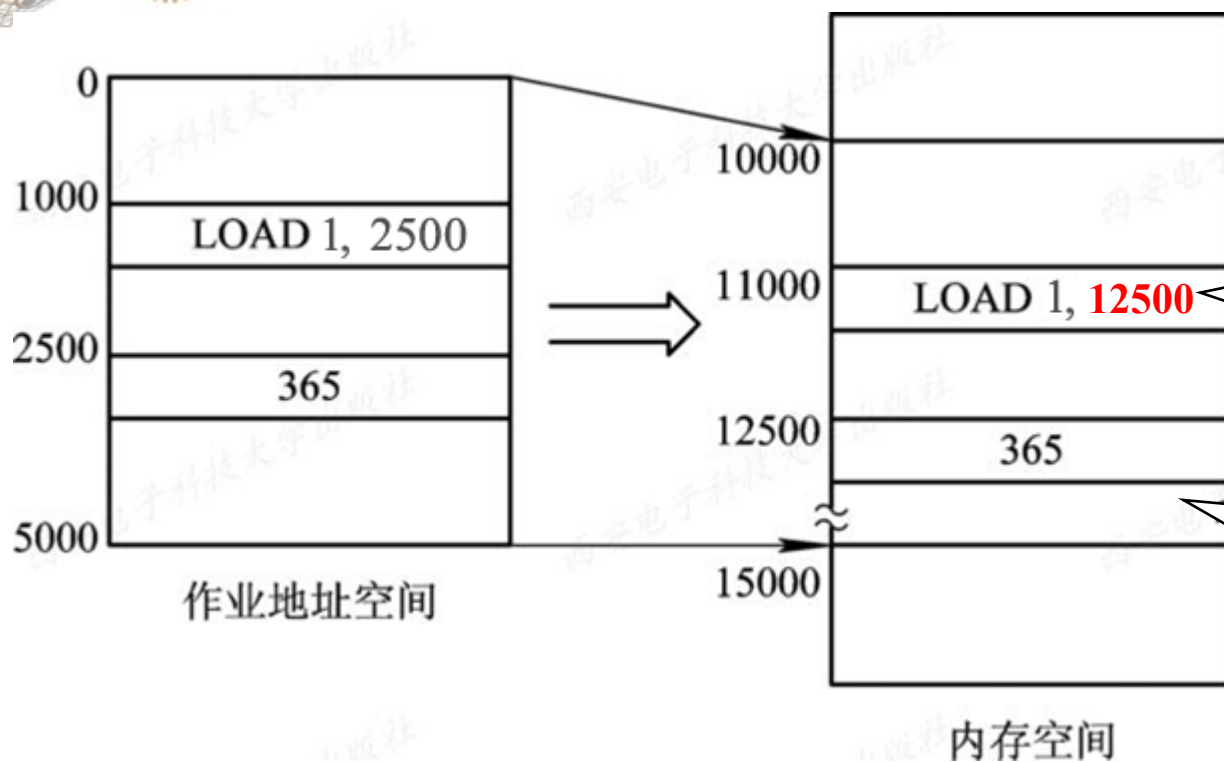
不需要对程序 and 数据的地址进行修改，程序中的相对地址（逻辑地址）即物理地址。

2. 可重定位装入方式(Relocation Loading Mode)



在**多道程序环境**下，编译程序不可能预知经编译后所得到的目标模块应放在内存的何处。

对于用户程序编译所形成的若干个目标模块，它们的起始地址通常都是**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。

采用可重定位装入方式，它可以根据内存的具体情况将装入模块装入到内存的适当位置。



把在装入时对目标程序中指令和数据的修改过程称为重定位。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。



3. 动态运行时的装入方式 (Dynamic Run-time Loading)

可重定位装入方式并不允许程序运行时在内存中移动位置。然而，实际情况是，在运行过程中它在内存中的位置可能经常要改变，此时就应采用动态运行时装入的方式。

把地址转换的工作推迟到程序执行时进行。

装入后的所有地址都仍是相对地址。

这种方式需要重定位寄存器的支持。



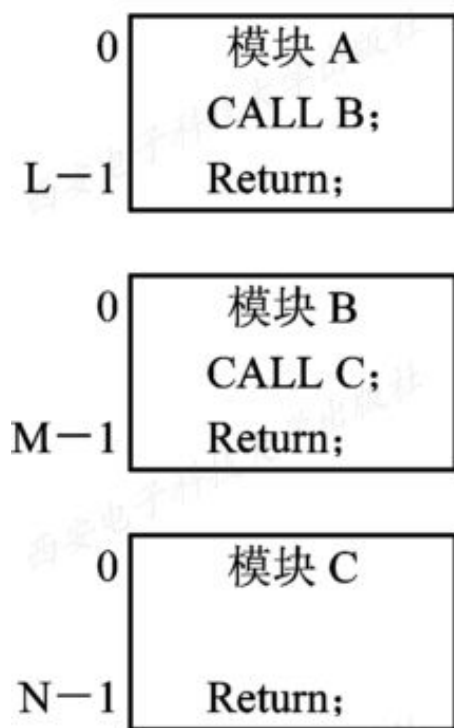
4.2.2 程序的链接

1. 静态链接(Static Linking)方式

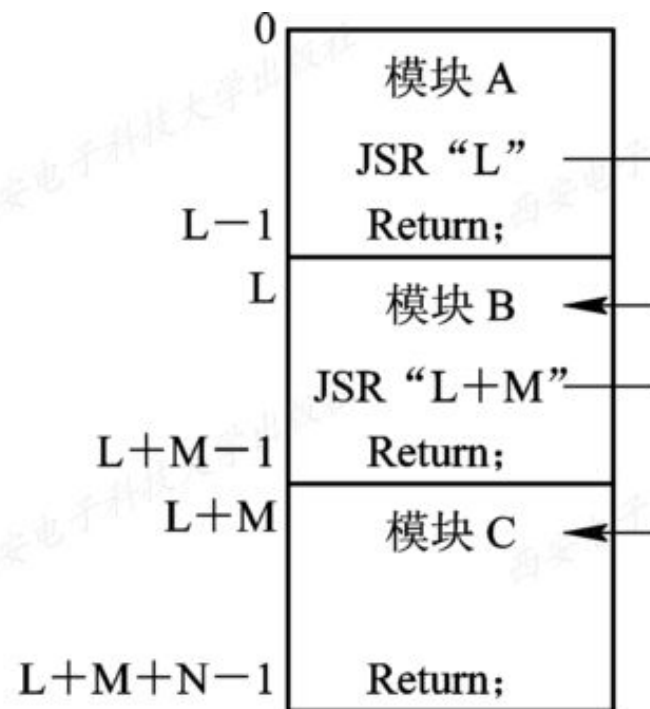
在程序运行之前，先将各目标模块及它们所需的库函数链接成一个完整的装配模块，以后不再拆开。将几个目标模块装配成一个装入模块时，须解决以下两个问题：

- (1) 对相对地址进行修改。
- (2) 变换外部调用符号。

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**。通常都不在把它拆开，要运行时可直接将它装入内存。把这种事先进行链接而以后不再拆开的链接方式称为**静态链接方式**。



(a) 目标模块



(b) 装入模块

图4-4 程序链接示意图

2. 装入时动态链接 (Load-time Dynamic Linking)

指将用户源程序编译后所得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。即在**装入一个目标模块**时，若发生一个外部模块调用事件（即装入系统发现目标模块中有对其他目标模块的调用），将引起装入程序**找出相应的外部目标模块**，并将它**装入内存**，还要**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：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：OS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)

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。

补充内容：Windows平台下的动态链接库(DLL)

动态链接库（Dynamic Link Libray, DLL）是Windows平台下的一个函数库，其中包含可同时由多个程序使用的代码和数据，使用DLL有助于促进代码重用和高效内存使用。常用的DLL有kernel.dll, user.dll, gdi.dll等系统DLL。DLL有两种类型：

(1) **加载时动态链接**：需要DLL文件，及其对应的头文件(.h)、静态库(.lib)文件。在**编译时提供头文件**，应用程序可以像调用本地函数一样显式调用 DLL中的函数。在**链接时提供静态库文件**，链接器会向系统提供DLL和其包含的函数的位置信息。当**运行程序时**，OS会自动根据可执行文件中的信息**加载DLL**。举例：**OpenCV库的调用**。

(2) **运行时动态链接**：仅需要DLL文件。应用程序中会**首先**调用API函数LoadLibrary或LoadLibraryEx以**加载 DLL**。成功加载 DLL 后，使用GetProcAddress函数**获取**要调用的 DLL **函数的地址**，然后再调用该函数。



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。而在用户区内存中，仅装有一道用户程序，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。



4.3.2 固定分区分配

固定分区分配是最简单的一种多道程序存储管理方式，它将用户空间划分成若干个大小固定的区域，在每个分区中只装入一道作业。当有空闲分区时，便可以再从外存的后备作业队列中选择一个适当大小的作业，装入该分区。

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**(指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。

(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。



2. 内存分配

为了便于内存分配，通常将分区按其大小进行排队，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)。

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。

不足：限制了活动进程的数目，当进程大小与空闲分区大小不匹配时，内存空间利用率很低。

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

(a) 分区说明表

空间	操作系统
12 KB	作业 A
32 KB	作业 B
64 KB	
128 KB	作业 C
...	...
256 KB	

(b) 存储空间分配情况

图4-5 固定分区使用表

4.3.3 动态分区分配

动态分区分配又称为**可变分区分配**，是根据进程的实际需要，动态地为之分配内存空间。

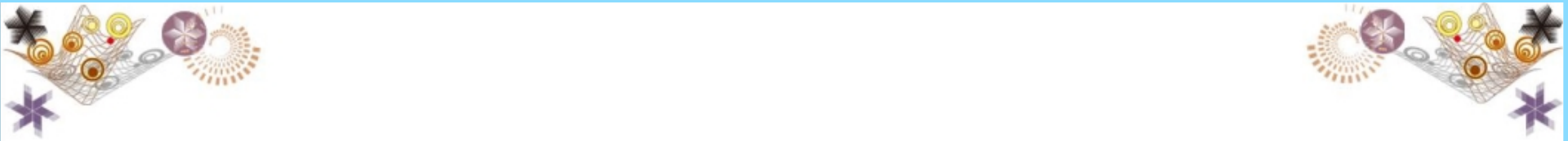
1. 动态分区分配中的数据结构

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，用于记录每个空闲分区的情况。每个空闲分区占一个表目，表目中包括分区号、分区大小和分区始址等数据项。

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**。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**。

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。

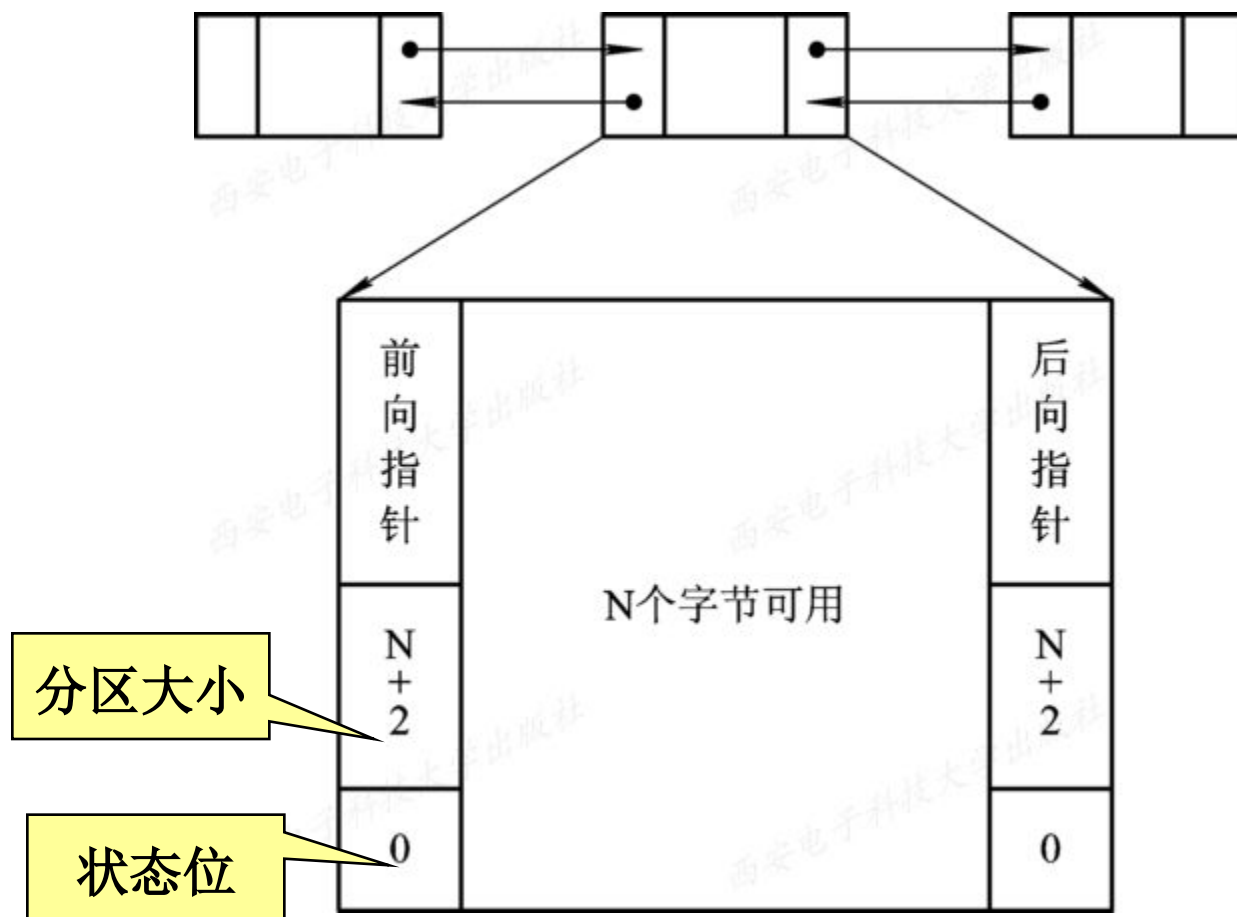


图4-7 空闲链结构



2. 动态分区分配算法

为把一个新作业装入内存，须按照一定的分配算法，从空闲分区表或空闲分区链中选出一分区分配给该作业。

由于内存分配算法对系统性能有很大的影响，故人们对它进行了较为广泛而深入的研究，于是产生了许多动态分区分配算法。

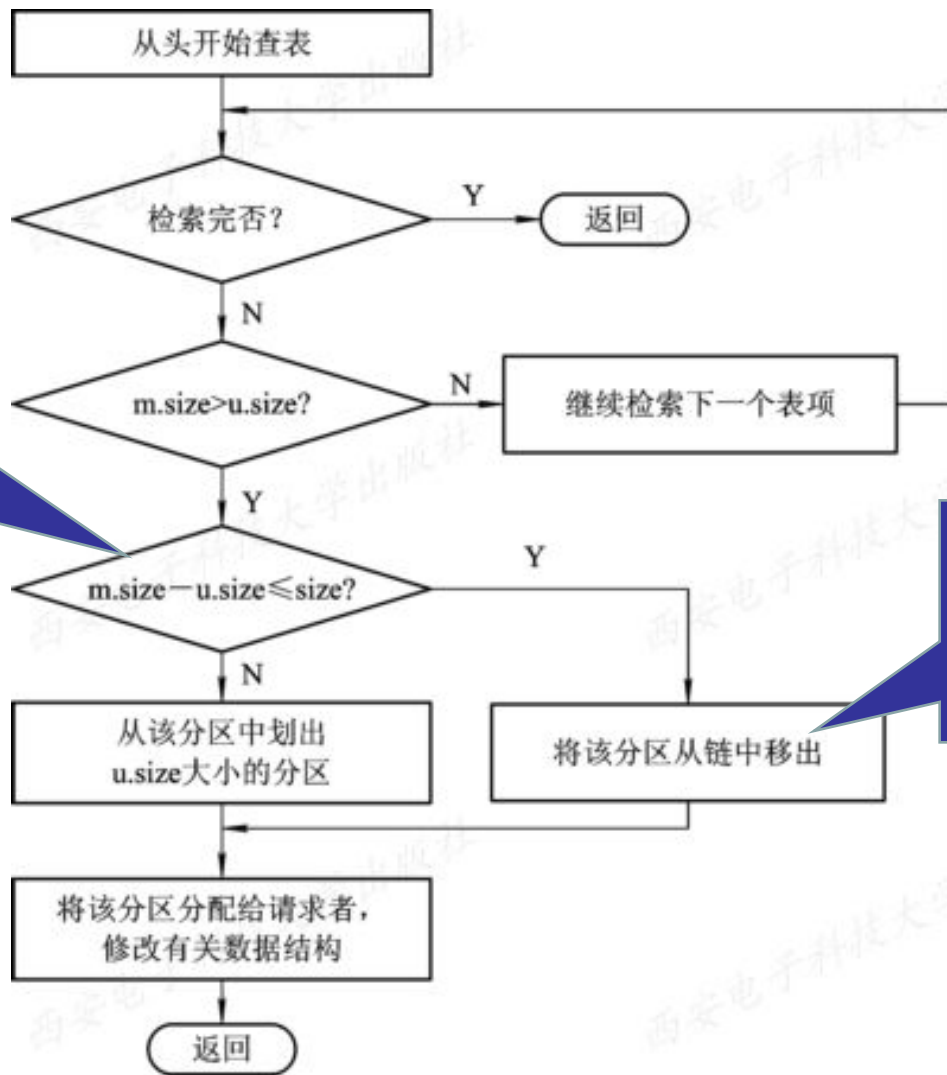


3. 分区分配操作

1) 分配内存

系统应利用某种分配算法，从空闲分区链(表)中找到所需大小的分区。

设请求的分区大小为 $u.size$ ，表中每个空闲分区的大小可表示为 $m.size$ 。



size是事先规定的
不再切割的剩余分
区的大小

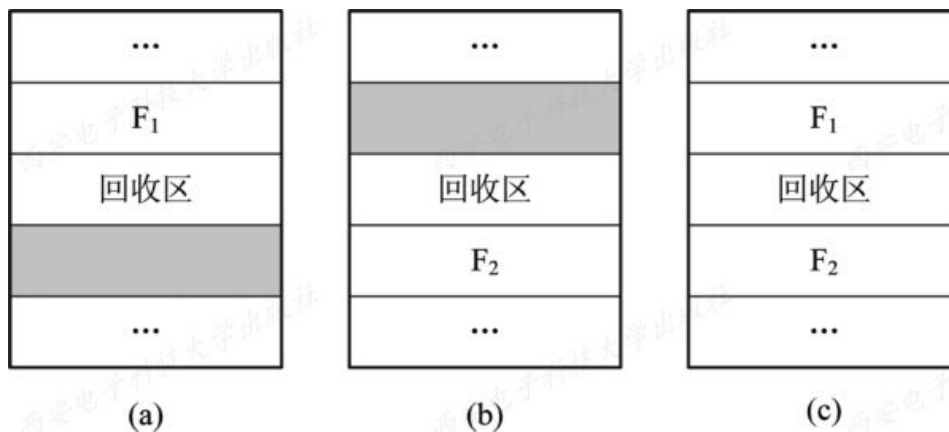
零头太小，不再继
续分割，把整个分
区都分配给进程

图4-8 内存分配流程

2) 回收内存

当进程运行完毕释放内存时，系统根据回收区的首址，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：

- (1) 回收区与插入点的前一个空闲分区 F_1 相邻接，图(a)。
- (2) 回收分区与插入点的后一空闲分区 F_2 相邻接，图(b)。
- (3) 回收区同时与插入点的前、后两个分区邻接，图(c)。
- (4) 回收区既不与 F_1 邻接，又不与 F_2 邻接。



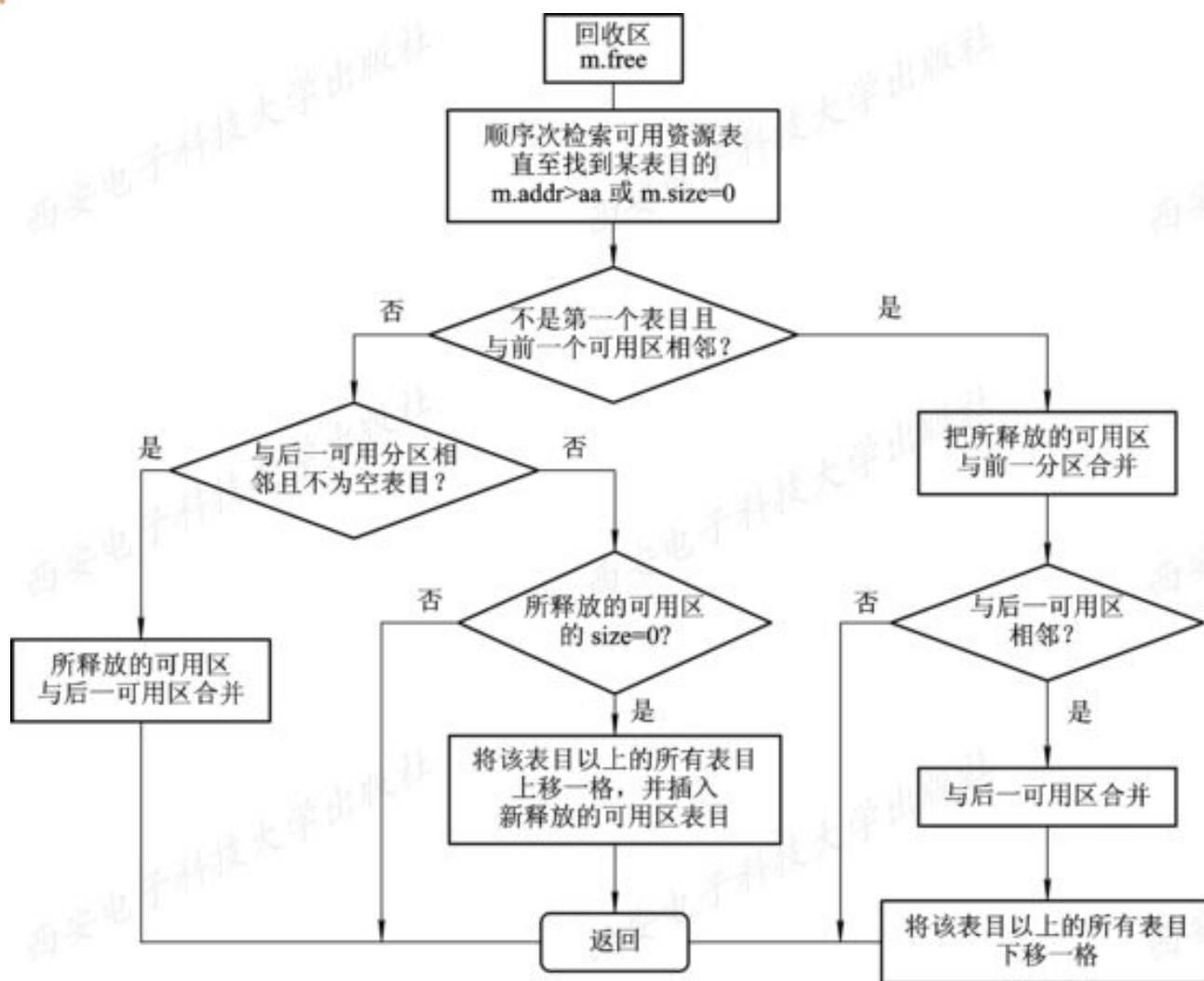


图4-10 内存回收流程

4.3.4 基于顺序搜索的动态分区分配算法

1. 首次适应(first fit, FF)算法

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。



分配过程：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。



优点：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。

缺点：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。





2. 循环首次适应(next fit, NF)算法

为避免低址部分留下许多很小的空闲分区，以及减少查找可用空闲分区的开销，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，从中划出一块与请求大小相等的内存空间分配给作业。

NF算法的特点：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。

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。

BF算法的特点：在宏观上看BF算法不一定是最佳的。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碎片。**

4. 最坏适应(worst fit, WF)算法

由于最坏适应分配算法选择空闲分区的策略正好与最佳适应算法相反：它在扫描整个空闲分区表或链表时，总是挑选一个**最大的**空闲区，从中分割一部分存储空间给作业使用，以至于存储器中缺乏大的空闲分区，故把它称为是**最坏适应算法**。

WF算法的优点：可使剩下的空闲区不至于太小，产生碎片的几率最小，**对中、小作业有利**，同时最坏适应分配算法查找效率很高。

WF算法的缺点：会使存储器中**缺乏大的空闲分区**。

动态分区分配算法比较

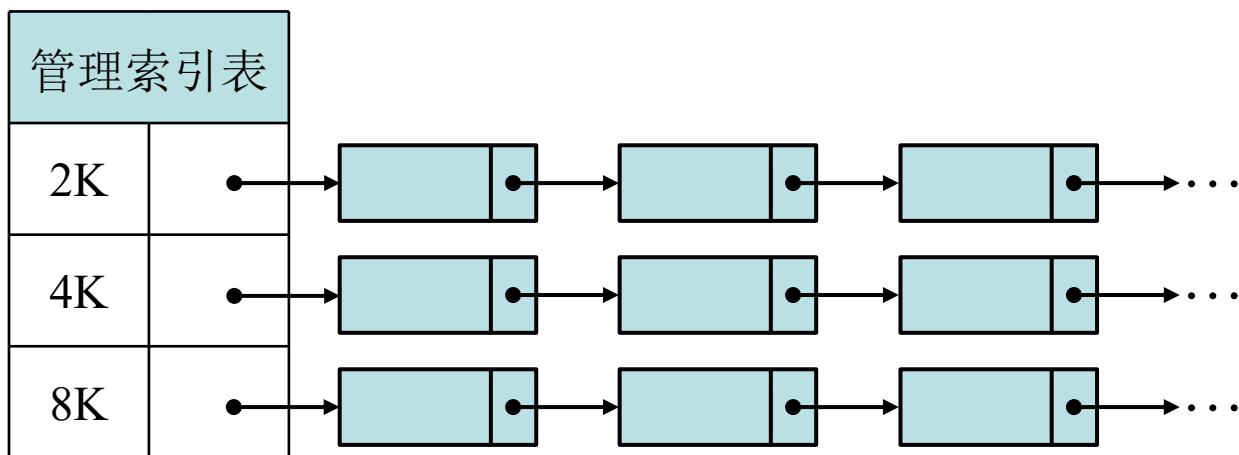
算法	算法思想	分区排列顺序	优点	缺点
首次适应	从头到尾找适合的分 区	空闲分区以地址 递增次序排列	综合看性能最好。 算 法开销小 ，回收分区 后一般不需要对空闲 分区队列重新排序	
最佳适应	优先使用更小的分 区，以保留更多大 分区	空闲分区以容量 递增次序排列	会有更多的大分区被 保留下来，更能满足 大进程需求	会产生很多太小的、难以 利用的碎片； 算法开销大 ， 回收分区后可能需要对空 闲分区队列重新排序
最坏适应	优先使用更大的分 区，以防止产生太 小的不可用的碎片	空闲分区以容量 递减次序排列	可以减少难以利用的 小碎片	大分区容易被用完，不利 于大进程； 算法开销大 (原因同上)
邻近适应 (循环首 次适应)	由首次适应演变而 来，每次从上次查 找结束位置开始查 找	空闲分区以地址 递增次序排列 (可排列成循环 链表)	不用每次都从低地址 的小分区开始检索。 算法开销小 (原因同 首次适应算法)	会使高地址的大分区也被 用完

4.3.5 基于索引搜索的动态分区分配算法

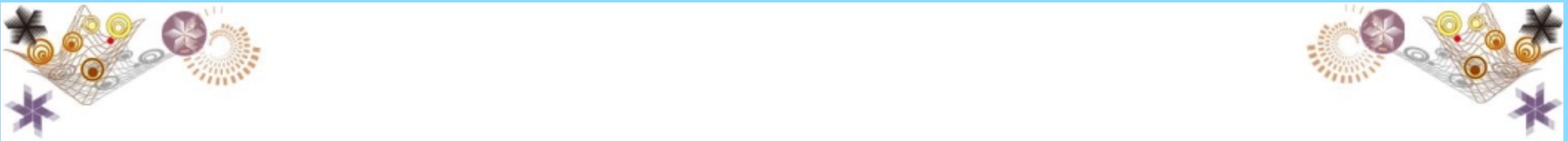
1. 快速适应(quick fit)算法

该算法又称为分类搜索法，是将空闲分区根据其容量大小进行分类，对于**每一类具有相同容量的所有空闲分区**，单独设立一个空闲分区链表，系统中存在**多个空闲分区链表**。同时，在内存中设立**一张管理索引表**，其中的每一个索引表项对应了一种空闲分区类型，并记录了该类型空闲分区链表表头的指针。

空闲分区的分类是根据**进程常用的空间大小进行划分**，如2 KB、4 KB、8 KB等，对于其它大小的分区，如7 KB这样的空闲区，既可以放在8 KB的链表中，也可以放在一个特殊的空闲区链表中。




基于索引的动态分区分配算法对应的数据结构



QF算法的优点：查找效率高，**仅需要根据进程的长度**，寻找到能容纳它的最小空闲区链表，并取下第一块进行分配即可。另外该算法在进行空闲分区分配时，**不会**对任何分区产生**分割**，所以**能保留大的分区**，满足对大空间的需求，也**不会产生内存碎片**。

缺点：为了有效合并分区，是在分区归还主存时**算法复杂**，**系统开销较大**。此外，该算法在分配空闲分区时是以进程为单位，一个分区只属于一个进程，因此在为进程所分配的一个分区中，**或多或少地存在一定的浪费**。**整体上会造成可观的存储空间浪费**，这是典型的以空间换时间的作法。



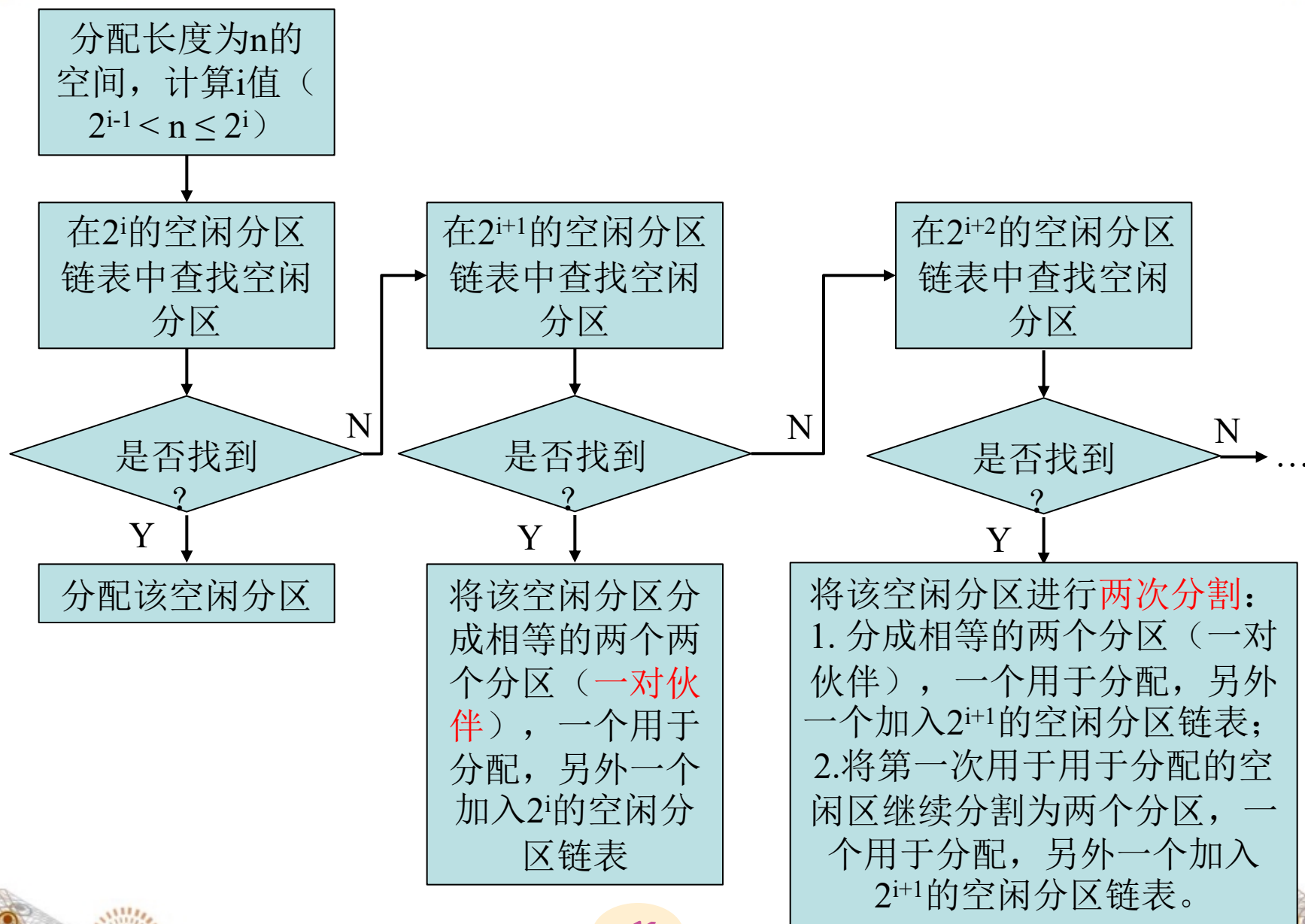
2. 伙伴系统(buddy system)

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(k为整数， $1 \leq k \leq m$)。通常 2^m 是整个可分配内存的大小(也就是最大分区的大小)。

假设系统的可利用空间容量为 2^m 个字，则系统开始运行时，整个内存区是一个大小为 2^m 的空闲分区。

在系统运行过程中，由于不断地划分，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大小进行分类。对于具有相同大小的所有空闲分区，单独设立一个空闲分区双向链表，这样，不同大小的空闲分区形成了k个空闲分区链表。

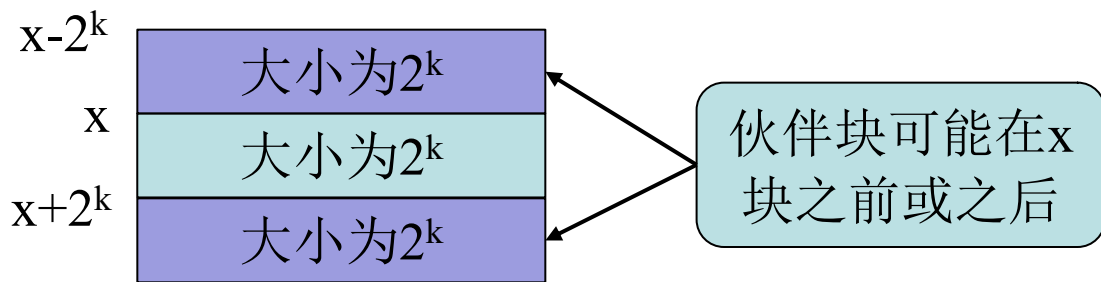
伙伴系统分配空间算法



与一次分割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，即将刚回收的分区与其空闲的伙伴分区合并为一个2倍大小的空闲分区。

在伙伴系统中，对于一个大小为 2^k ，地址为 x 的内存块，其伙伴块的地址则用 $\text{buddy}_k(x)$ 表示，其通式为：

$$\text{buddy}_k(x) = \begin{cases} x + 2^k & (\text{若 } x \bmod 2^{k+1} = 0) \\ x - 2^k & (\text{若 } x \bmod 2^{k+1} = 2^k) \end{cases}$$





★ 时间性能

分配和回收的时间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。

在回收空闲分区时，需要对空闲分区进行合并，所以其时间性能比快速适应算法差，但由于它采用了索引搜索算法，比顺序搜索算法好。

★ 空间性能

对空闲分区进行了合并，减少了小的空闲分区，提高了空闲分区的可使用率，故优于快速适应算法，比顺序搜索法略差。

3. 哈希算法

利用哈希快速查找的特点，以及空闲分区在可利用空闲区表中的分布规律，**建立哈希函数**，**构造**一张以空闲分区大小为关键字的**哈希表**，该表的每一个表项记录了一个对应的空闲分区链表表头指针。

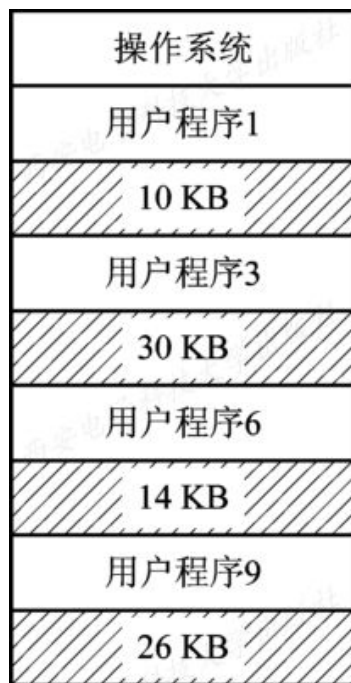
在进行空闲分区分配时，根据所需空闲分区大小，通过**哈希函数计算**，即得到在哈希表中的位置，从中得到相应的空闲分区链表，实现最佳分配策略。

4.3.6 动态可重定位分区分配

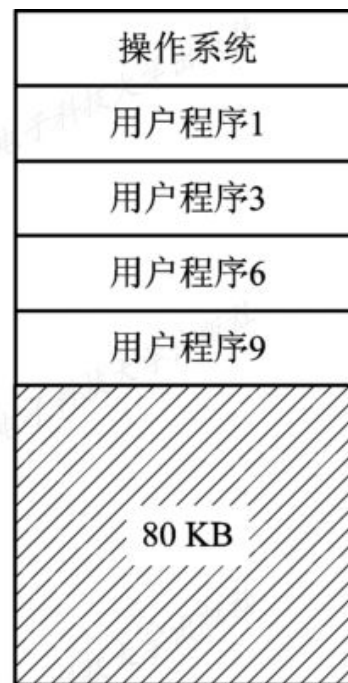
1. 紧凑

连续分配方式的一个重要特点是，一台计算机运行了一段时间后，它的内存空间将会被分割成许多小的分区，而缺乏大的空闲空间。即使这些分散的许多小分区的容量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。

若想把大作业装入，可采用的一种方法是：将内存中的所有作业进行移动，使它们都相邻接。这种通过移动内存中的作业的位置，把原来多个分散的小分区拼接成一个大分区的方法，称为“拼凑”或“紧凑”。



(a) 紧凑前



(b) 紧凑后

每次“紧凑”后，都必须对移动了的程序或数据进行重定位，大大影响了系统的效率。

图4-11 紧凑的示意

2. 动态重定位

在动态运行时装入的方式中，作业装入内存后的所有地址仍然都是相对(逻辑)地址。而将相对地址转换为绝对(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**。

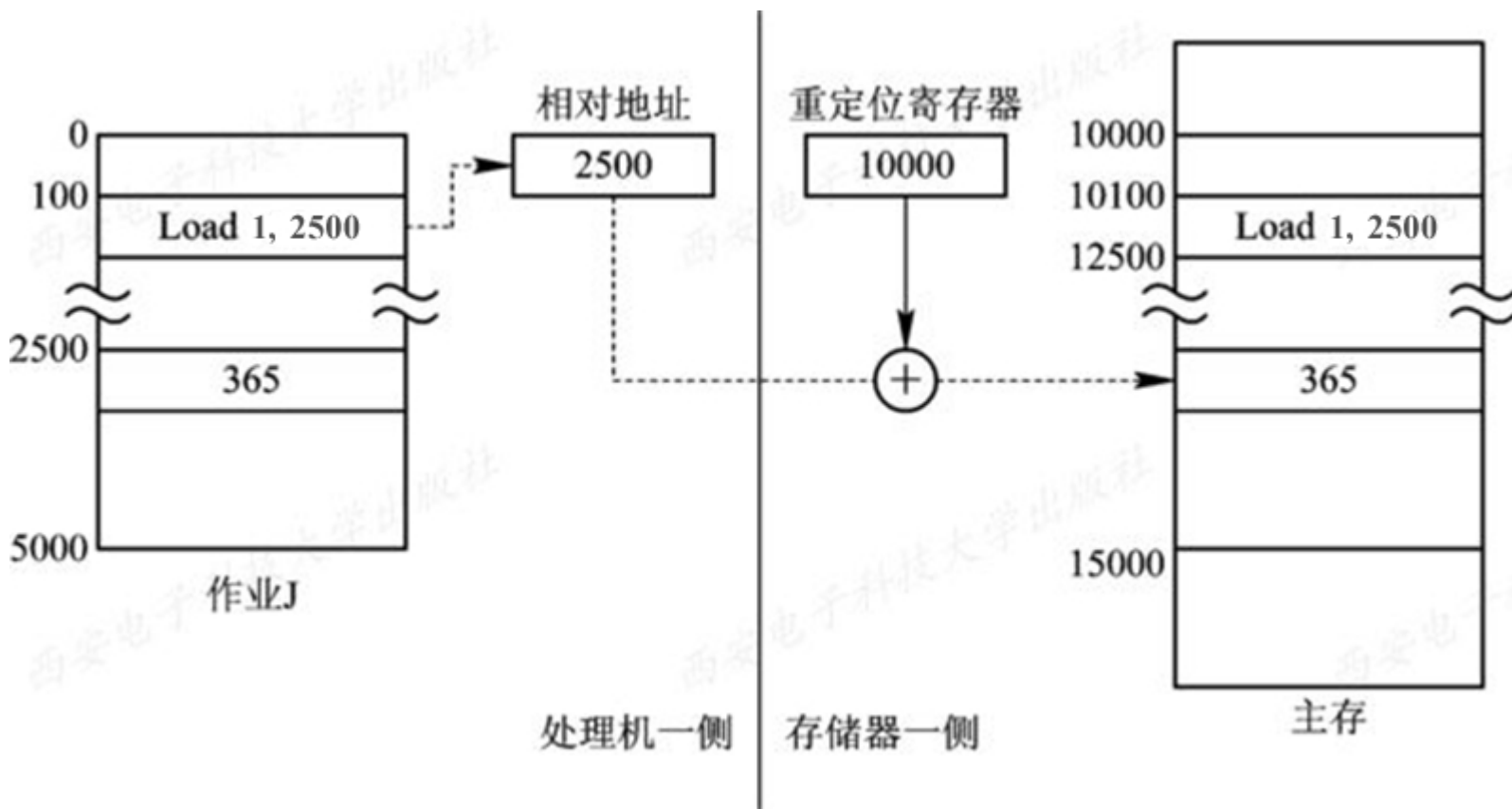


图4-12 动态重定位示意图

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。

当该算法不能找到一个足够大的空闲分区以满足用户需求时，如果所有的小的空闲分区的容量总和大于用户的要求，这时便须对内存进行“紧凑”，将经“紧凑”后所得到的大空闲分区分配给用户。如果所有的小的空闲分区的容量总和仍小于用户的要求，则返回分配失败信息。

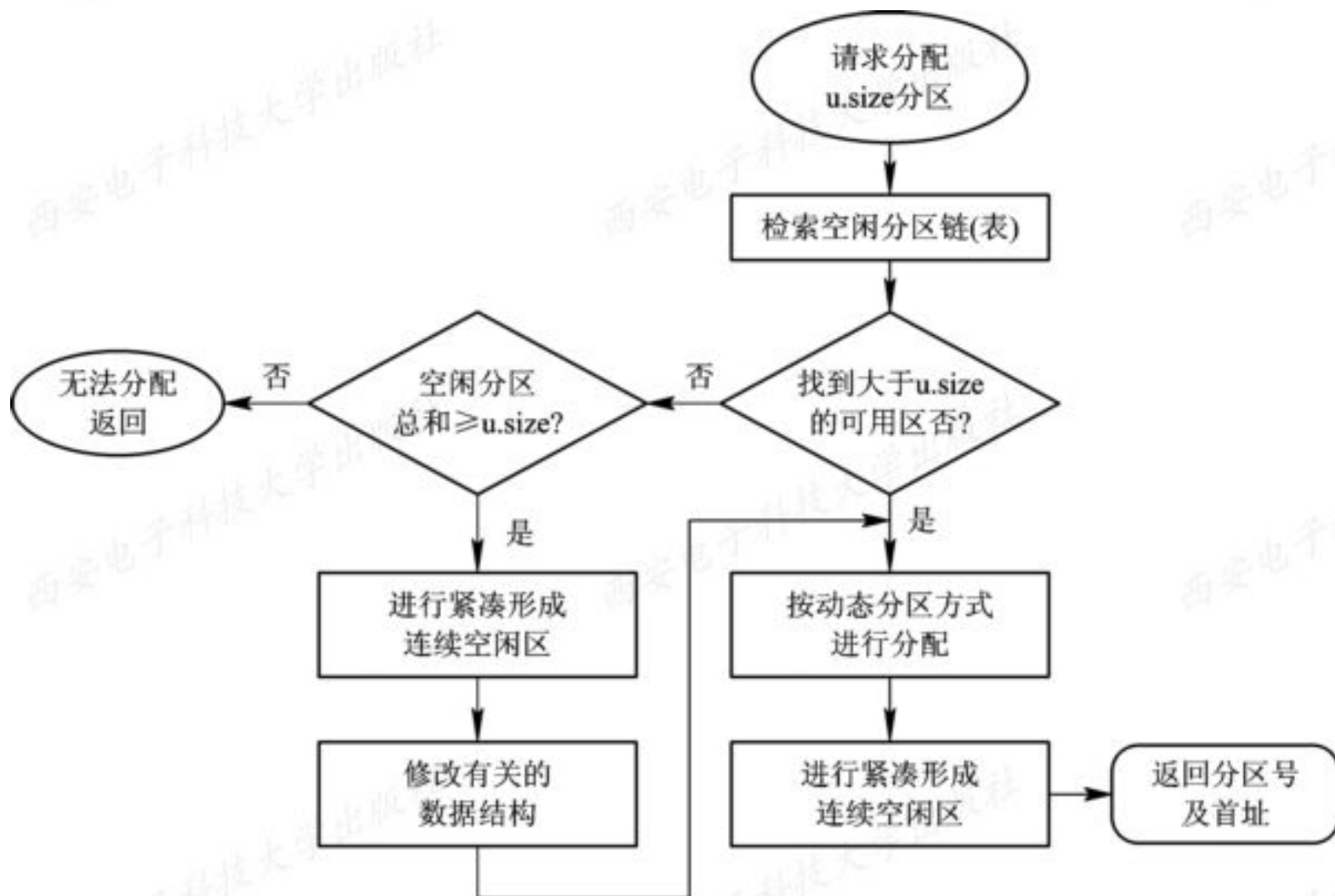


图4-13 动态分区分配算法流程图

连续分配

单一连续分配

- 只支持单道程序，内存分为系统区和用户区，用户程序放在用户区
- 无外部碎片，有内部碎片

绝对装入

固定分区分配

- 支持多道程序，内存用户空间分为若干个固定大小的分区，每个分区只能装一道作业
- 无外部碎片，有内部碎片
- 两种分区方式
 - 分区大小相等
 - 分区大小不等

可重定位装入

动态分区分配

- 支持多道程序，在程序装入内存时，根据进程的大小动态地建立分区
- 无内部碎片，有外部碎片
- 回收内存时，相邻的空闲分区要合并
- 动态分区分配算法
 - 基于顺序搜索的动态分区分配算法
 - 基于索引搜索的动态分区分配算法

可重定位分区分配

- 动态分区分配算法增加了“紧凑”的功能
- 支持多道程序，需要硬件地址变换机构提供地址转换

动态运行时装入

4.4 对换(Swapping)

对换技术也称为交换技术，最早用于麻省理工学院的单用户分时系统CTSS中。由于当时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。

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。



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，在系统中增设了对换(也称交换)设施。

“**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。

由于对换技术能够有效地改善内存的利用率，现在已被广泛地应用于OS中。

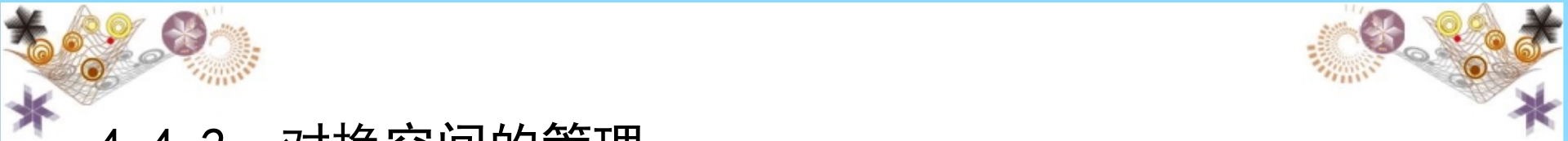
2. 对换的类型

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：对换是以整个进程为单位的，广泛地应用于分时系统中，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。

(2) 页面(分段)对换：对换是以“页”或“段”为单位进行的。是实现请求分页和请求分段式存储管理的基础，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，以及进程的换入。



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：占用磁盘空间大部分，文件区用于存放各类文件。对该区的管理的主要目标是提高文件存储空间的利用率，然后才是提高对文件的访问速度。因此，对文件区的管理采用离散分配方式。

2) 对对换空间管理的主要目标：占用磁盘空间的小部分，用于存放从内存换出的进程。进程在对换区中驻留的时间是短暂的，对换操作又较频繁，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。

2. 对换区空闲盘块管理中的数据结构

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。

对换区的首址(盘块号)	大小（盘块数）



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。



4.4.3 进程的换出与换入

1. 进程的换出

换出过程可分为以下两步：

(1) 选择被换出的进程：选择处于阻塞状态且优先级最低

(2) 进程换出过程：先申请交换空间，若申请成功，就启动磁盘，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

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2s)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

离散分配方式：

(1) 分页存储管理方式：将用户程序的地址空间分为若干个区域，称为“页”或“页面”。将内存空间也分为若干个物理块或页框(frame)，页和块的大小相同。

(2) 分段存储管理方式：把用户程序的地址空间分为若干个大小不同的段，每段可定义一组相对完整的信息。在存储器分配时，以段为单位。

(3) 段页式存储管理方式。分页和分段方式的结合。

4.5.1 分页存储管理的基本方法

1. 页面和物理块



(1) 页面。

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。

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**”。



(2) 页面大小。

在分页系统中的**页面大小应适中**。

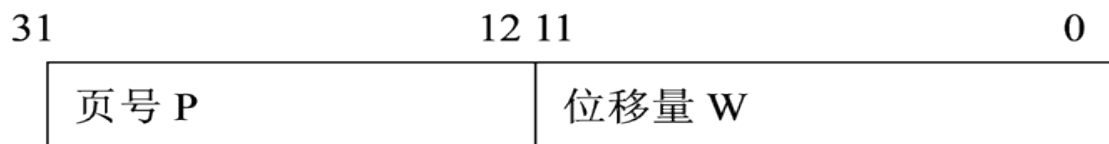
页面若太小会使每个进程占用较多的页面，从而导致进程的**页表过长**，占用大量内存；还会**降低页面换进换出的效率**。

如果选择的页面较大，会使**页内碎片增大**。

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。



2. 地址结构

分页地址中的地址结构如下：



位移量W(或称为页内地址)

图中的地址长度为32位，
0~11位为**页内地址**，即每页的大小为4 KB(2^{12} B)；
12~31位为**页号**，地址空间最多允许有1 M(2^{20} B) 页。



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：

$$P = \text{INT} \left[\frac{A}{L} \right], \quad d = [A] \text{ MOD } L$$

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**

$$A_{16} = \mathbf{1000\ 0111\ 1010}$$

则**P = 2**，**d = 122**。



3. 页表

在分页系统中，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($0 \sim n$)，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。

页表的作用是实现从页号到物理块号的地址映射

各个页表项顺序存储，
页号不占用存储空间

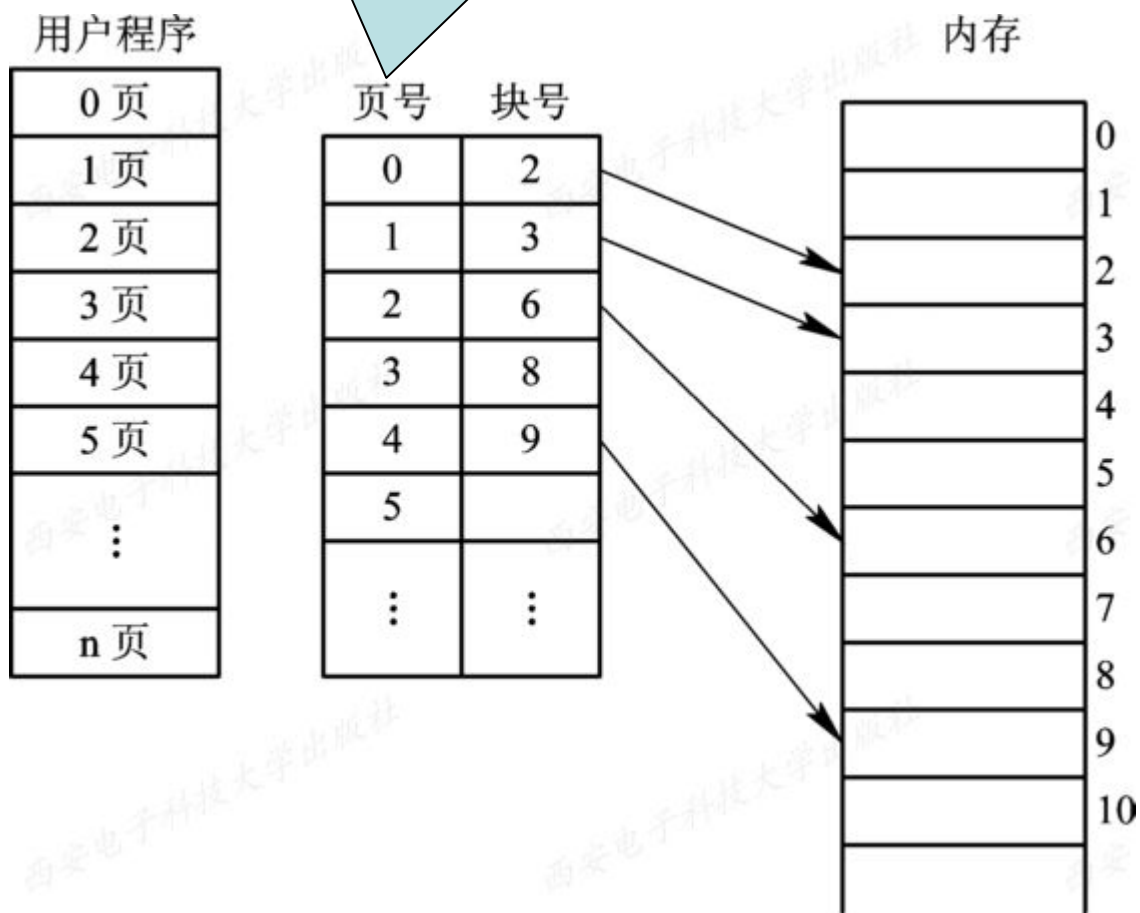


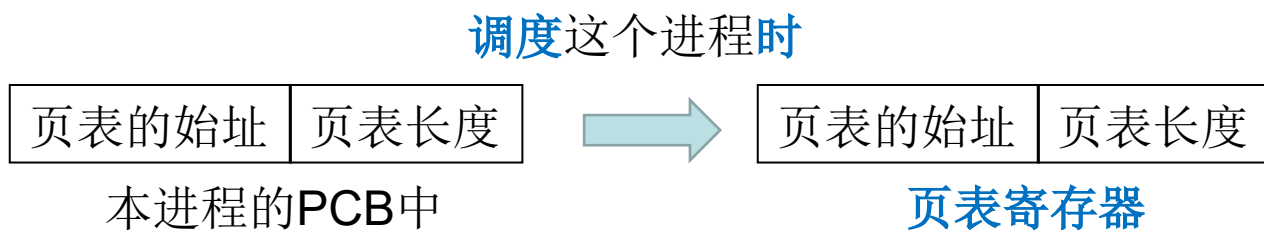
图4-14 页表的作用

4.5.2 地址变换机构

1. 基本的地址变换机构

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**。



补充知识: x86_64系统下使用控制寄存器CR3作为PTR

地址变换机构会自动的将相对地址（逻辑地址）分为页号和页内地址两部分

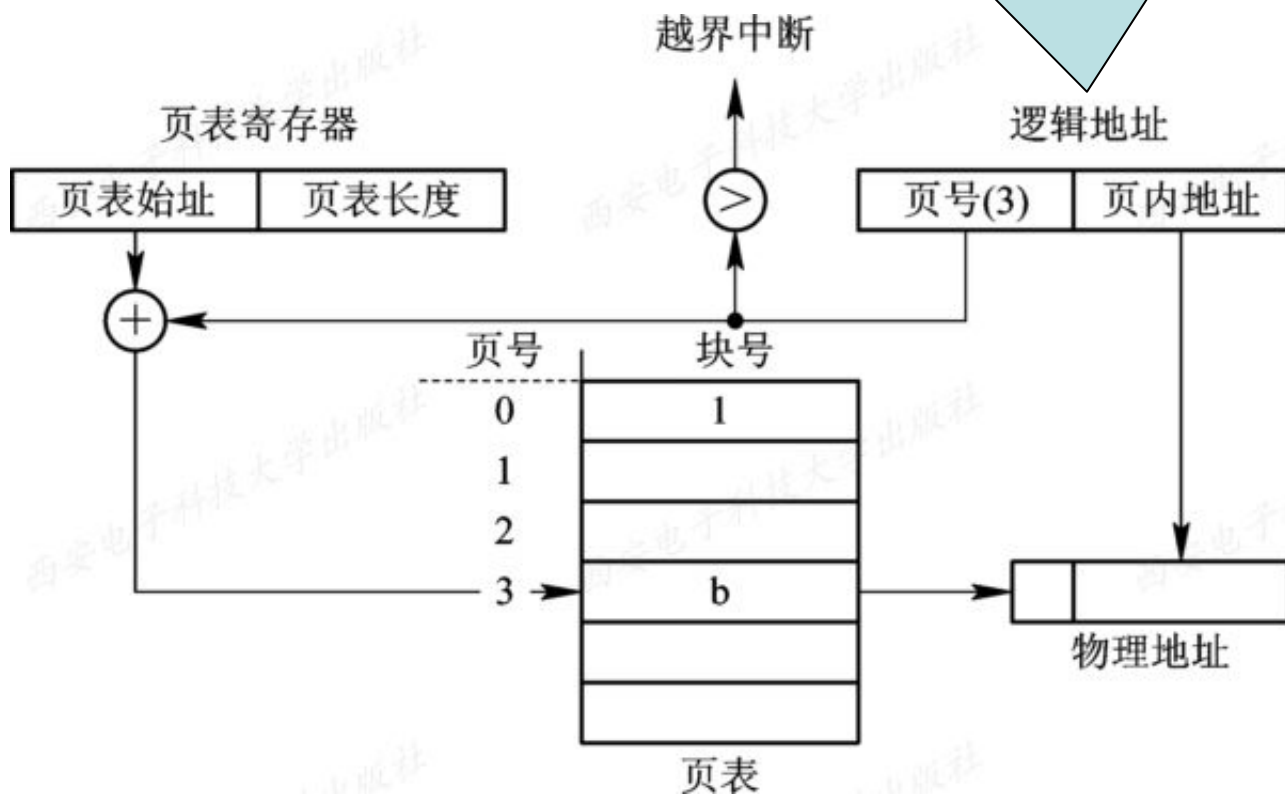
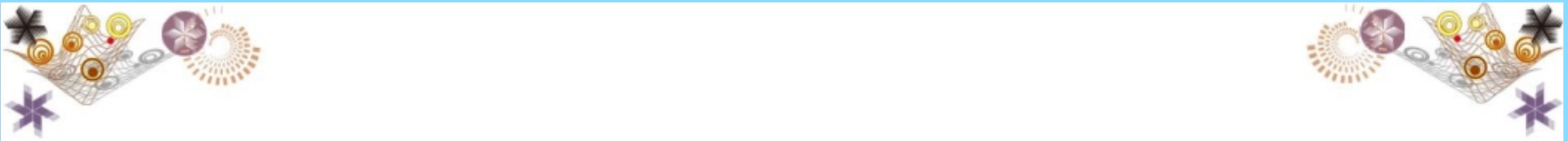


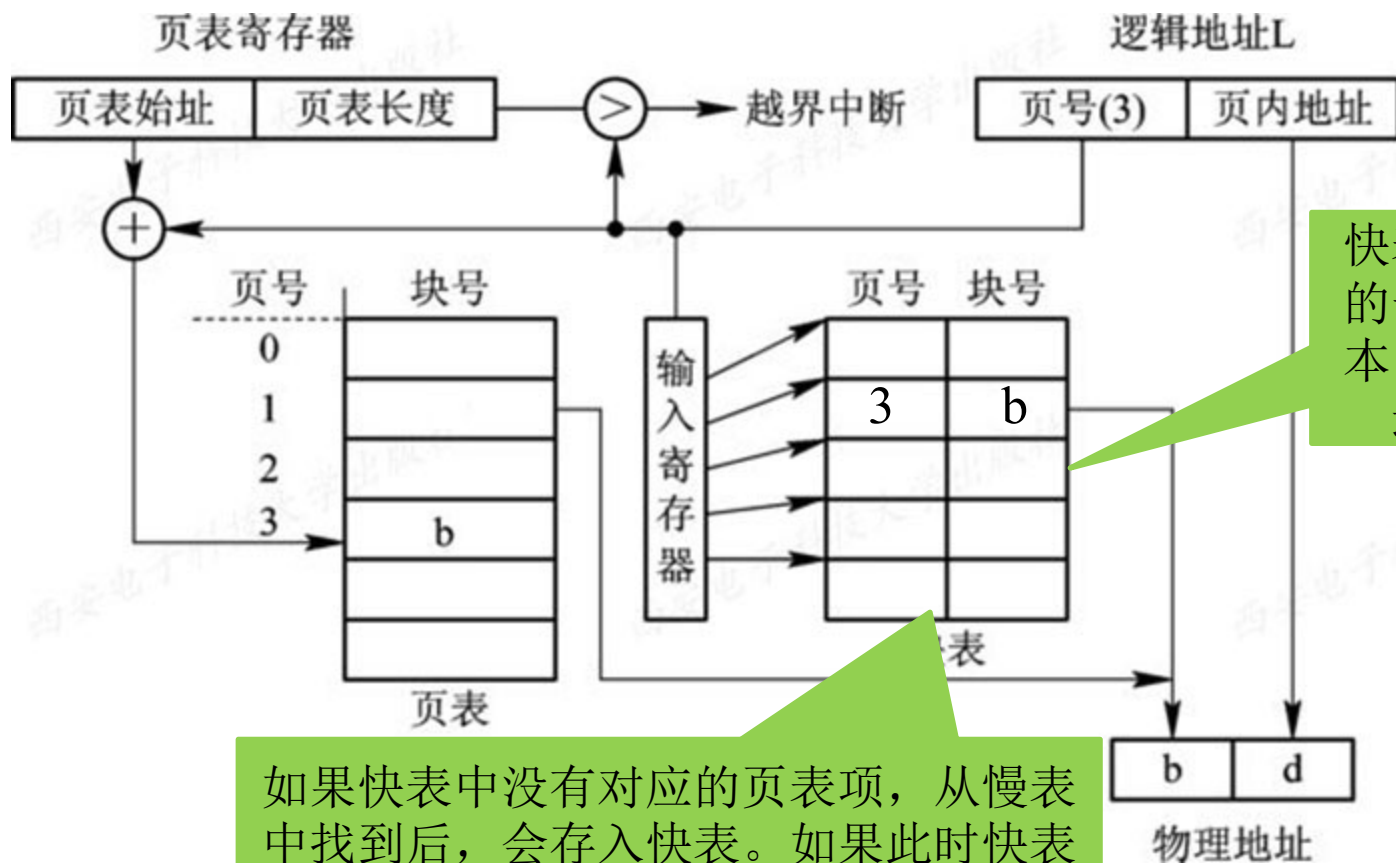
图4-15 分页系统的地址变换机构



2. 具有快表的地址变换机构

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。因此，采用这种方式将使计算机的处理速度降低近1/2。

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(存储)器，又称为“联想寄存(存储)器”(Associative Memory)，或称为“快表”，在IBM系统中又取名为TLB(Translation Lookaside Buffer)，用以存放当前访问的那些页表项。



快表是慢表的一部分副本，需要记录页号

如果快表中没有对应的页表项，从慢表中找到后，会存入快表。如果此时快表已满，会换出一个原有的页表项

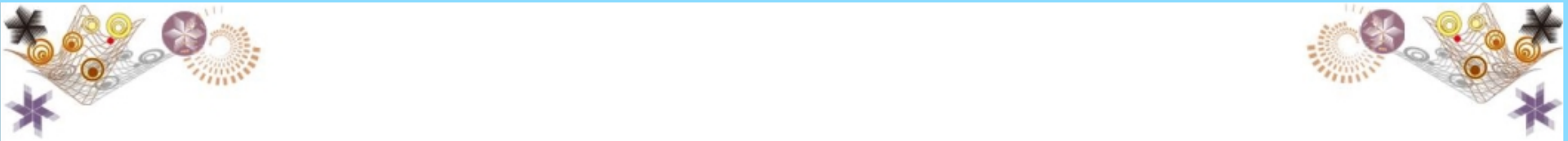
图4-16 具有快表的地址变换机构

4.5.3 访问内存的有效时间

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，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，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，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 表示命中率， t 表示访问一次内存所需要的时间。

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

现代的大多数计算机系统，都支持**非常大的逻辑地址空间**。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。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；

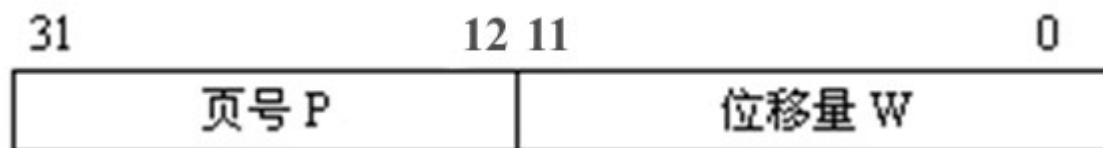
(2) 只将当前需要的**部分页表项调入内存**，其余的页表项仍驻留在磁盘上，需要时再调入。

1. 两级页表(Two-Level Page Table)

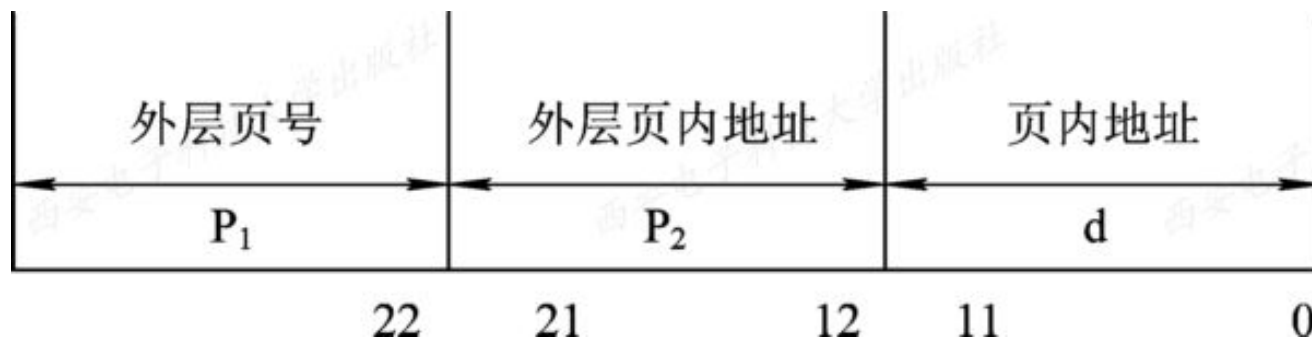
将页表进行分页的方法，使**每个页面的大小与内存物理块的大小相同**，并为他们进行编号，即依次为0#页、1#页，...，n#页，并离散地将各个页面分别存放在不同的物理块中。同样，也要为**离散分配的页表再建立一张页表**，称为**外层页表(Outer Page Table)**，在每个页表项中记录了页表页面的物理块号。

以32位逻辑地址空间为例来说明。

当**页面大小为 4 KB** (2^{12}B) 时(12位), 若采用一级页表结构, 页号有20位, 即**页表项应有1MB** (2^{20}B) 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; 外层页表中的外层页内地址 P_2 为10位, 外层页号 P_1 也为10位。



外部页表占一个内存块

每个页表占一个内存块

n最多为
1024

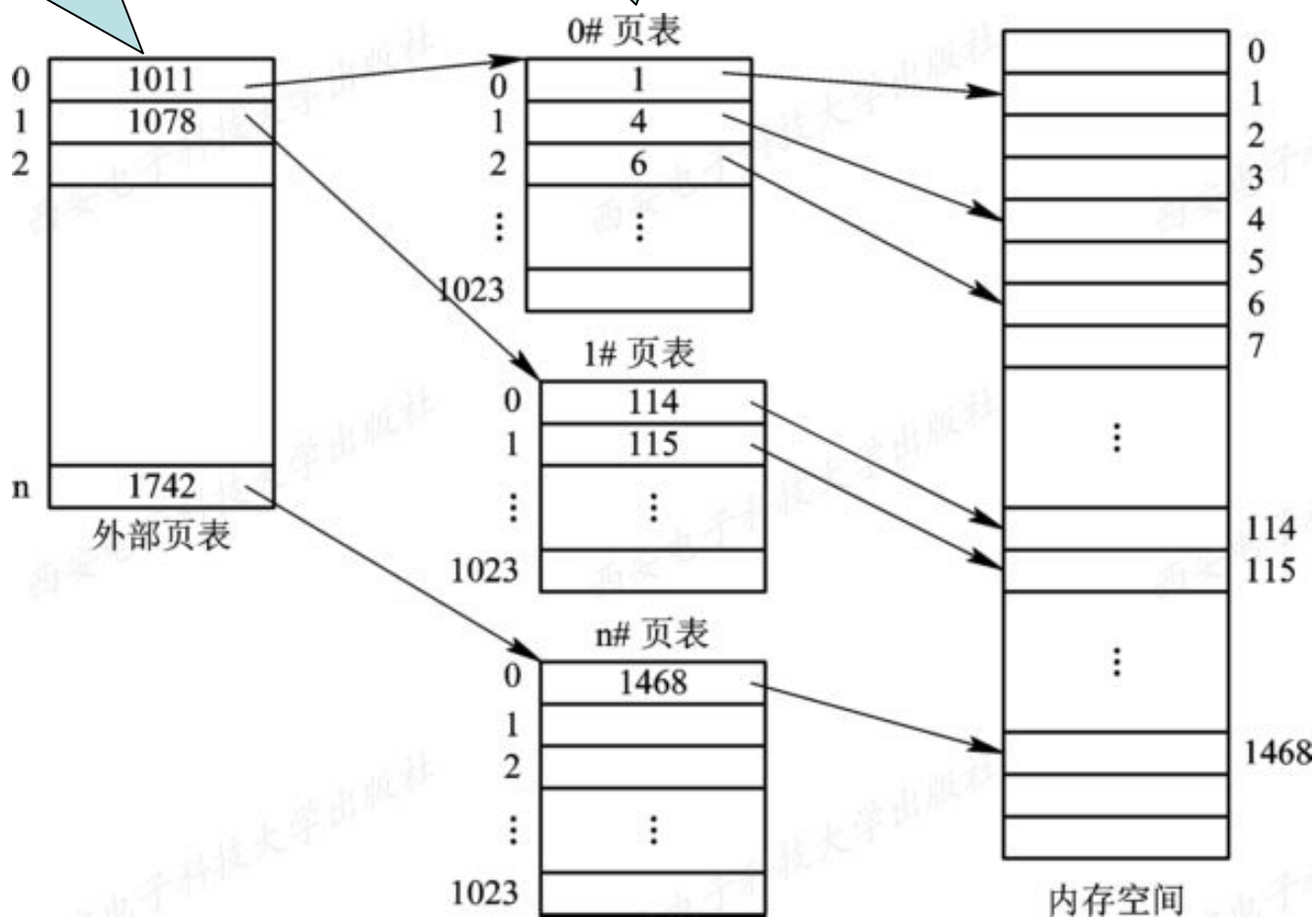


图4-18 具有两级页表的地址变换机构

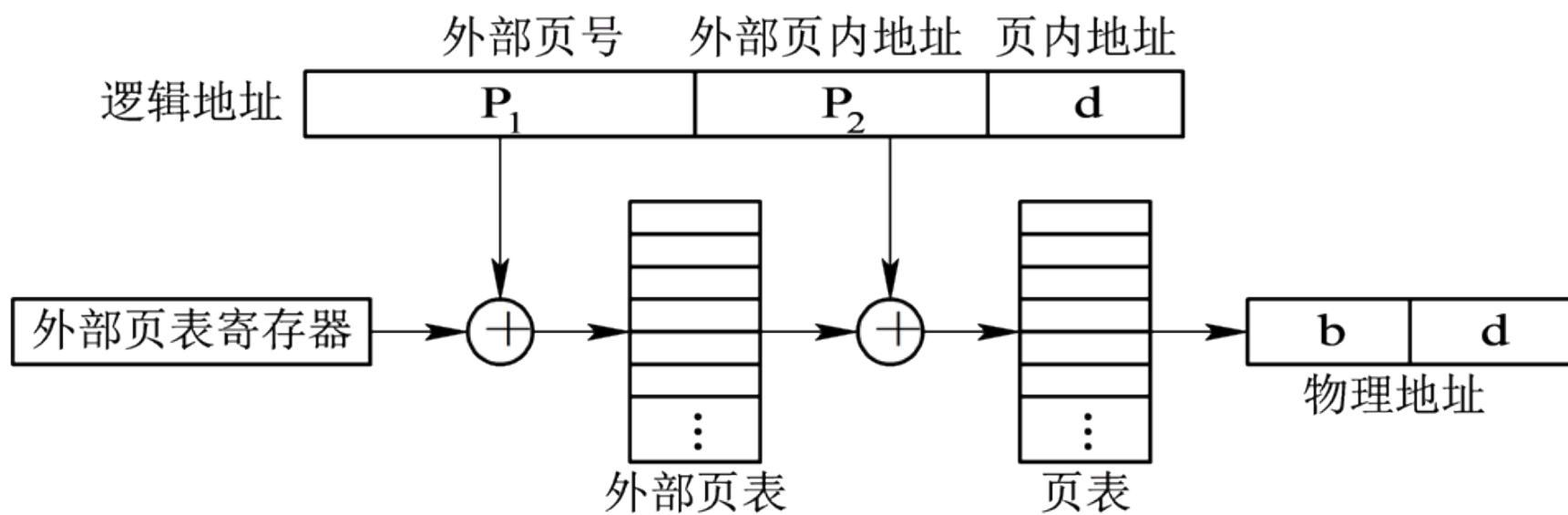


使用离散分配的方法并未减少页表所占用的内存空间。
能够用较少的内存空间存放页表的唯一方法是：**仅把当前需要的一批页表调入内存，以后再根据需要陆续调入。**

在采用两级页表结构的情况下，对于正在运行的进程，必须将其外层页表调入内存；而对于页表则只需要调入一页或几页。

为了表征某页的页表是否已经调入内存，还应**在外层页表项中增设一个状态位S**，其值若为0，表示该页表不再内存中，否则说明其分页已经调入内存。





具有两级页表的地址转换



2. 多级页表

对于32位的机器，采用两级页表结构是合适的。

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。因此，必须采用多级页表，将外层页表再进行分页，再利用第2级的外层页表来映射它们之间的关系。

目前，直接寻址的存储器空间减少到45位 (2^{45}) 长度左右，这样便可以使用三级页表结构来实现分页存储管理。



4.5.5 反置页表(Inverted Page Table)

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，会占用大量的内存空间。

为了减少页表占用的空间，引入了反置页表。反置页表为每一个物理块设置一个页表项，并将它们按物理块的编号排序，其中的内容是页号 and 其所隶属进程的标识符。

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

4. 6. 1 分段存储管理方式的引入

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。

下述的两条指令便使用段名和段内地址：

LOAD 1, [A] | 〈D〉 ;

STORE 1, [B] | 〈C〉 ;

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。



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，为了共享某个过程、函数或文件。分页系统中的“页”只是存放信息的物理单位(块)，并无完整的逻辑意义，这样，一个可被共享的过程往往可能需要占用数十个页面，这为实现共享增加了困难。因此，可以为该共享过程建立一个独立的段，这就极大地简化了共享的实现。

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。

3. 信息保护

信息保护同样是以信息的**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。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，往往存在着一些段，尤其是**数据段**（如**栈段和堆段**），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

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。

可见，动态链接也要求以段作为管理的单位。

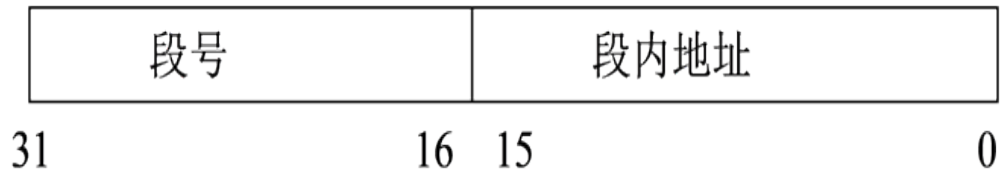


4.6.2 分段系统的基本原理



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**。例如，有主程序段MAIN、子程序段X、数据段D及栈段S等。

分段地址中的地址具有如下结构：



每个段都有自己的名字。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，其
逻辑地址由段号(段名)和段内地址所组成。



在该地址结构中，允许一个作业最长有 64 K (2^{16}) 个段，每个段的最大长度为 64 KB (2^{16}B)。

分段方式已得到许多编译程序的支持，编译程序能自动地根据源程序的情况而产生若干个段。

Pascal编译程序可以为全局变量、用于存储相应参数及返回地址的过程调用栈、每个过程或函数的代码部分、每个过程或函数的局部变量等等，分别建立各自的段。

Fortran编译程序可以为公共块(Common block)建立单独的段，也可以为数组分配一个单独的段。装入程序将装入所有这些段，并为每个段赋予一个段号。



2. 段表

在分段式存储管理系统中，为每个分段分配一个连续的分区，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，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，将段表放在内存中。

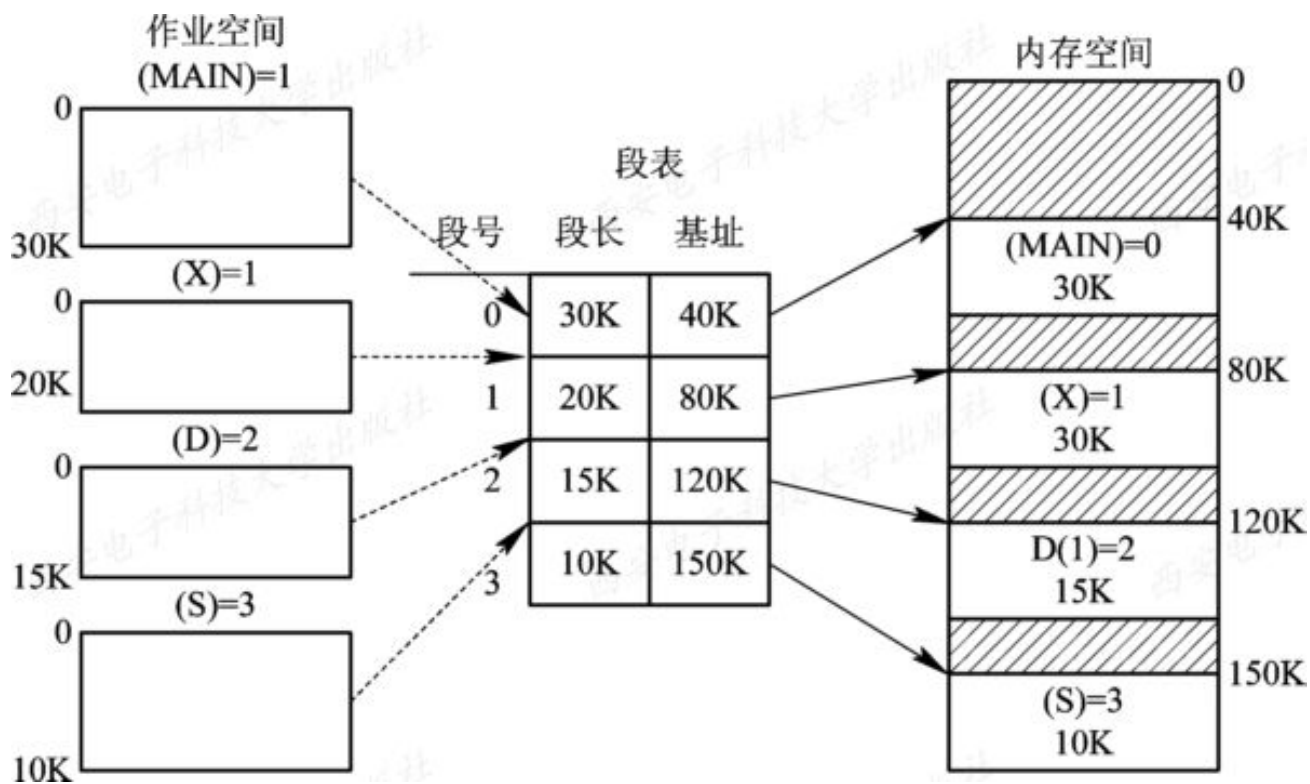


图4-19 利用段表实现地址映射

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。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。

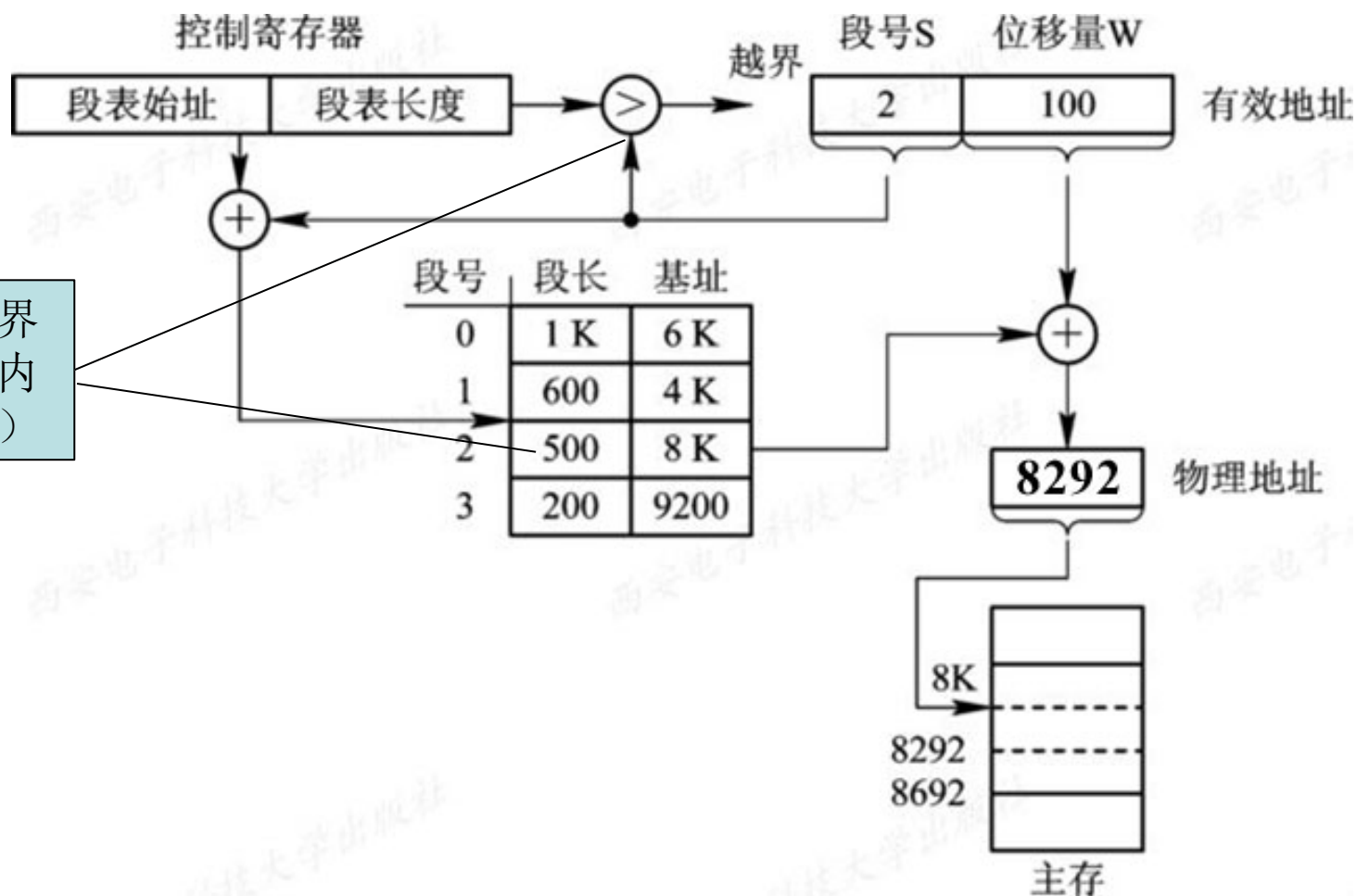


图4-20 分段系统的地址变换过程



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: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. 分页和分段的主要区别

(1) 页是信息的**物理单位**，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**，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。





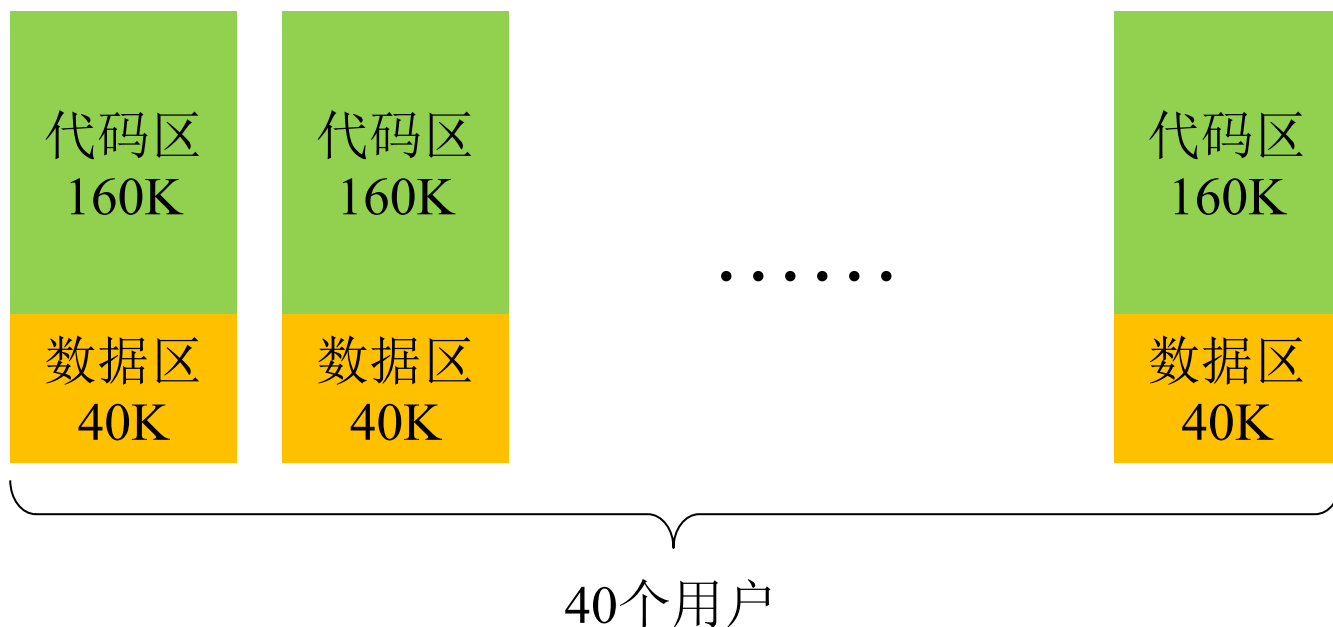
4.6.3 信息共享

1. 分页系统对程序和数据的共享

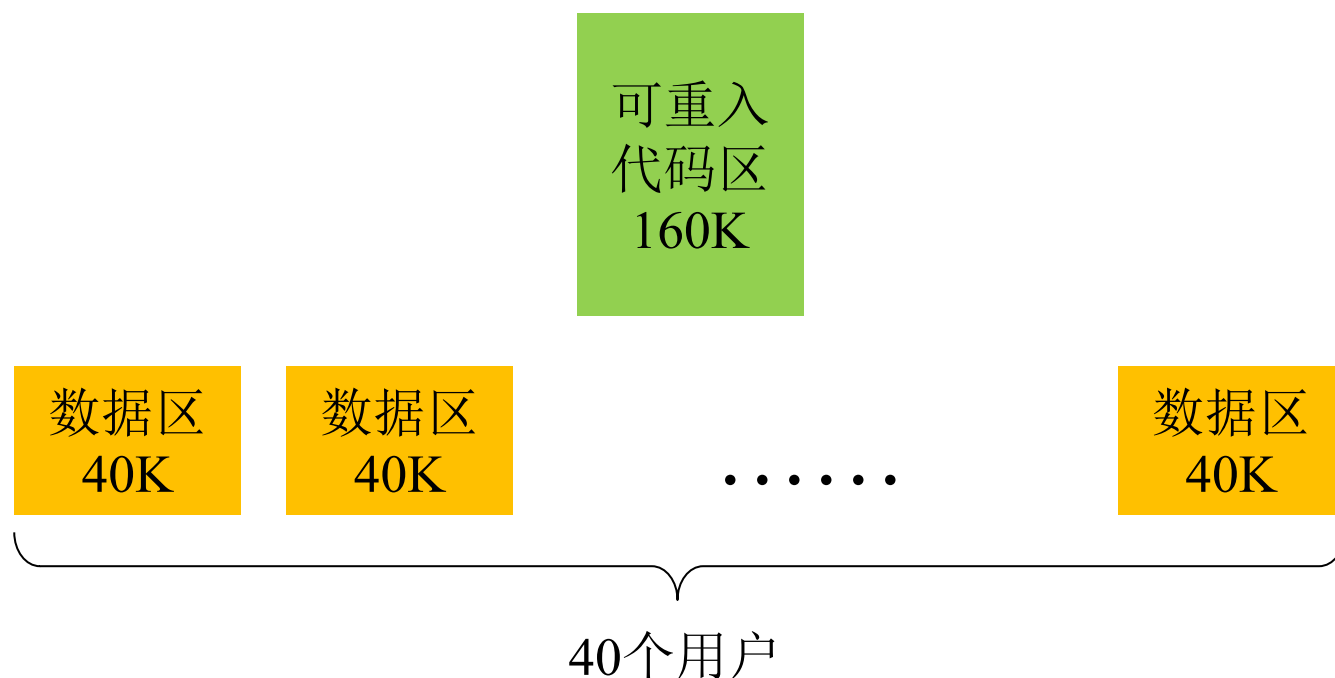
分段系统的一个突出优点，是易于实现段的共享，即允许若干个进程共享一个或多个分段，且对段的保护也十分简单易行。在分页系统中，虽然也能实现程序和数据的共享，但远不如分段系统来得方便。

举例：

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($40 \times 40 + 160$)，而不是8000 KB。





分页系统中的情况：

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（代码段的页表项），它们的物理块号都是21#~60#（相同）。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#，...，（不同）等等。

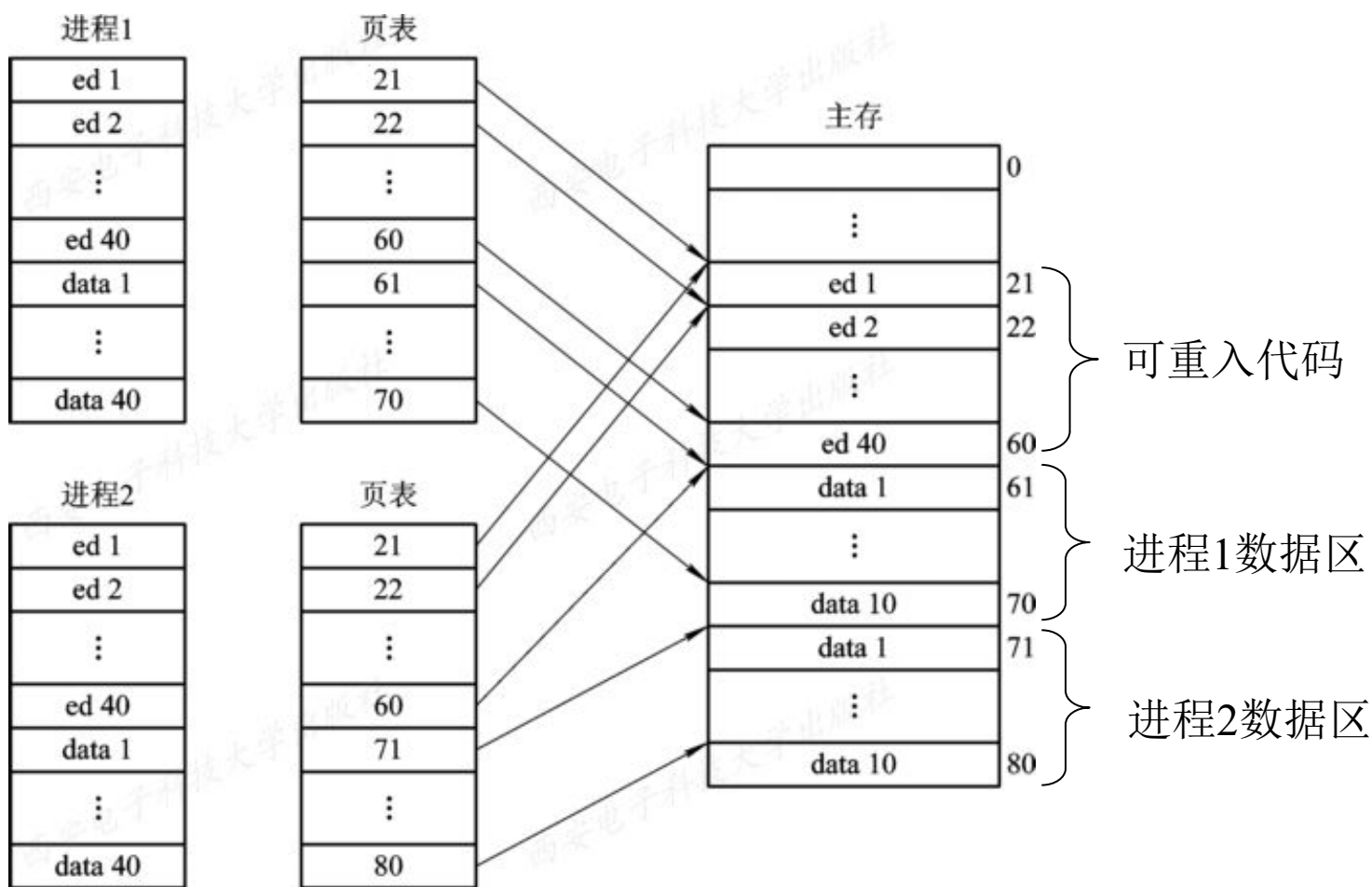



图4-21 分页系统中共享editor的示意图



2. 分段系统中程序和数据的共享

在分段系统中，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。

图4-22 分段系统中共享editor的示意图







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**。

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。

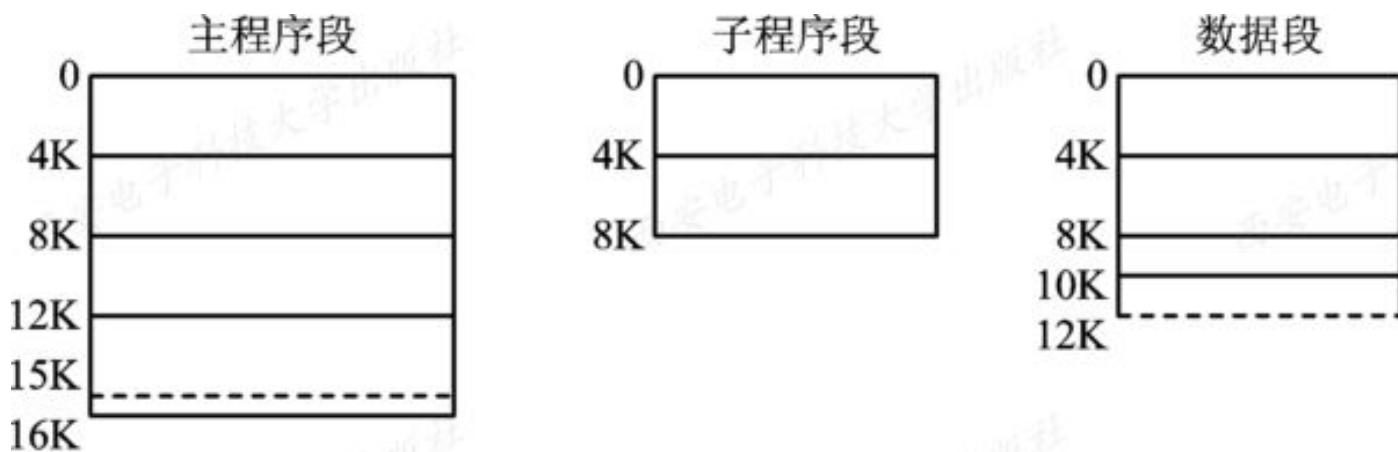




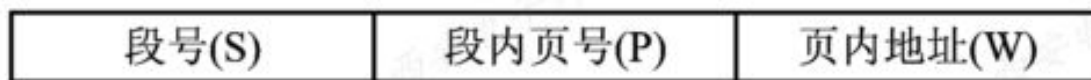
4.6.4 段页式存储管理方式

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。



(a)



(b)

图4-23 作业地址空间和地址结构



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。

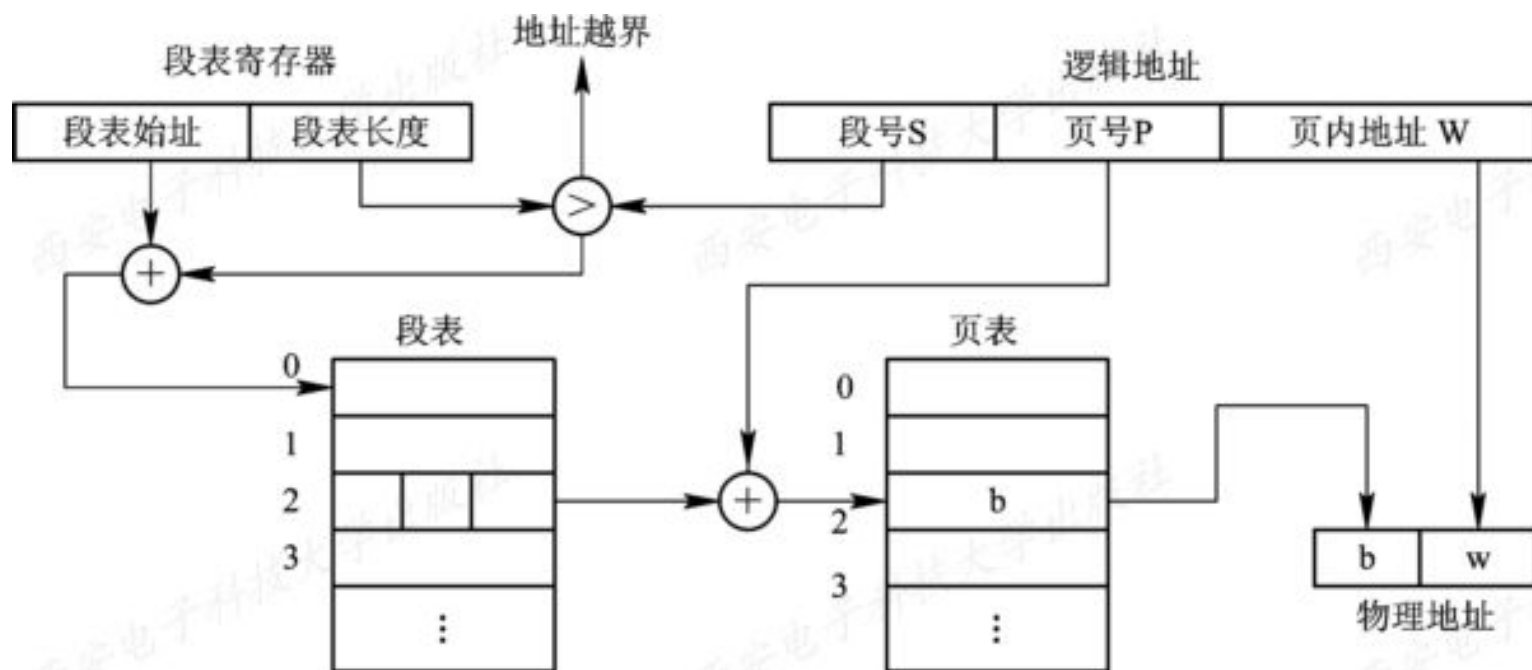


图4-25 段页式系统中的地址变换机构



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**。

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**高速缓冲存储器**。

