第四章　存 储 器 管 理

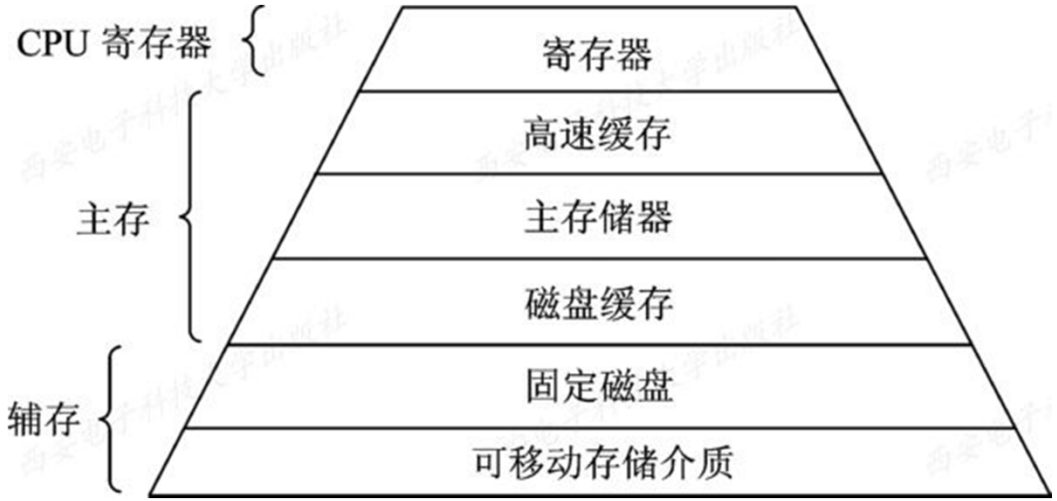
4.1　存储器的层次结构

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。



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**。

对于存放于其中的信息，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。但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。

通常，处理机都是从主存储器中取得指令和数据的，并将所取得的指令放入指令寄存器中，将所读取的数据装入到数据寄存器中；或者反之将寄存器中的数据存入到主存储器。

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，引入了寄存器和高速缓存。

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。

4.1.3　高速缓存和磁盘缓存

1. 高速缓存

在计算机系统中，为了缓和内存与处理机速度之间的矛盾，许多地方都设置了缓存。

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。

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。

寄存器通常由锁存器或触发器组成；

Cache通常由SRAM（静态随机存取存储器）组成；

内存通常由DRAM（动态随机存取存储器）组成。

通常，进程的程序和数据存放在主存中，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。这样，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。

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，直到下一条指令从主存中取出。

程序执行的局部性原理：程序在执行时呈现局部性规律，即在一较短时间内，程序的执行仅局限于某个部分。

通常，进程的程序和数据存放在主存中，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。这样，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。

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，直到下一条指令从主存中取出。

高速缓存的速度越高价格越贵，因此计算机中一般设置两级或多级高速缓存。紧靠CPU的一级高速缓存速度最高，容量最小；二级缓存容量稍大，速度稍低。

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。

磁盘缓存与高速缓存不同，它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，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。

4.2　程序的装入和链接

用gcc编译程序生成可执行文件有时候看起来似乎仅通过编译一步就完成了，但事实上，使用gcc编译工具由C语言源程序生成可执行文件的过程并不单单是一个编译的过程，完整的编译流程要经过下面的几个过程：

- 预处理（Pre-Processing）：gcc首先调用cpp命令进行预处理，主要实现对源代码编译前的预处理，比如将源代码中指定的头文件包含进来；
- 编译（Compiling）：调用cc1 命令进行编译，将源代码翻译生成汇编代码；
- 汇编（Assembling）：调用as 命令进行工作，将汇编代码生成扩展名为.o 的目标文件；目标文件包含机器代码（可直接被CPU执行）以及代码在运行时使用的数据，如重定位信息、用于链接或调试的程序符号（变量和函数的名字）等；
- 链接（Linking）：调用链接器ld处理可重定位文件，把它们的各种符号引用和符号定义转换为可执行文件中的合适信息(一般是虚拟内存地址) 的过程。

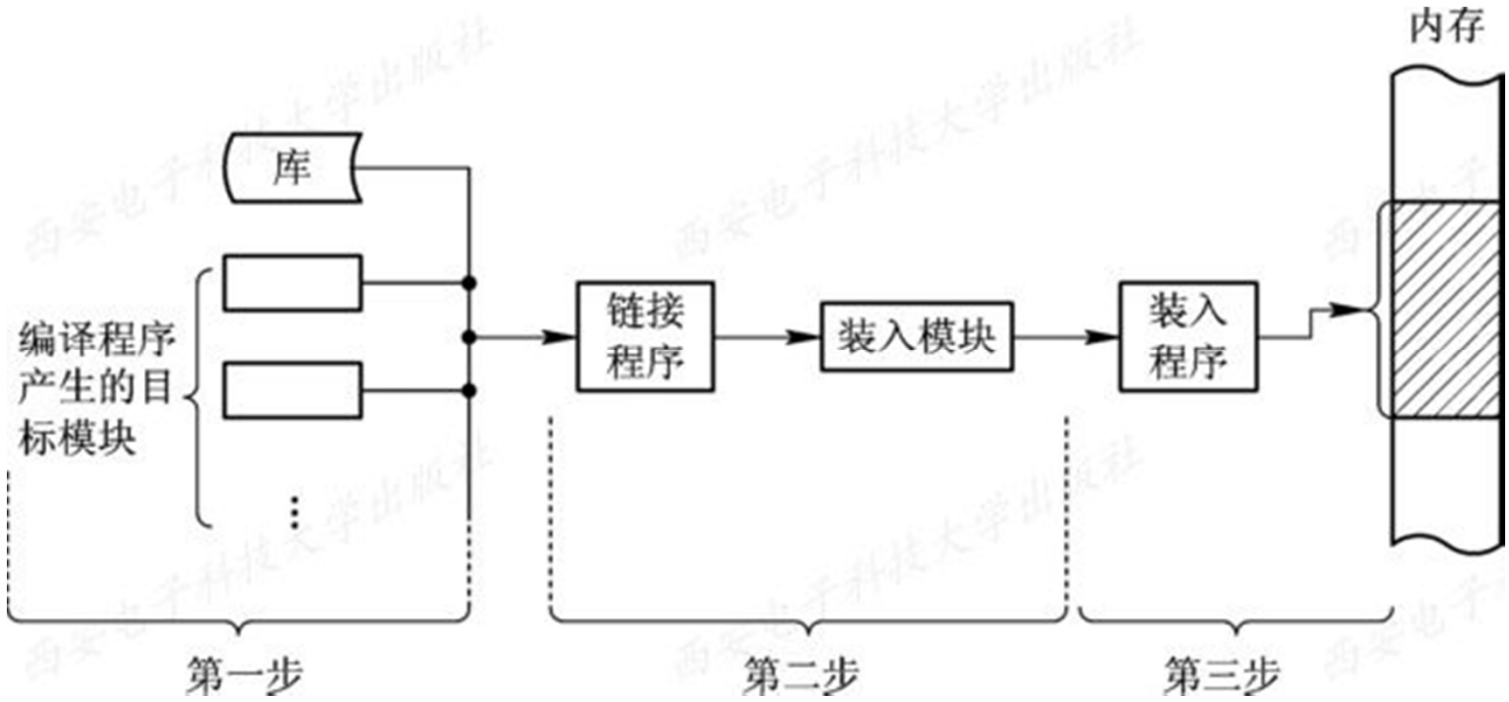


图4-2　对用户程序的处理步骤

4.2.1　程序的装入

为了阐述上的方便，我们先介绍一个无需进行链接的单个目标模块的装入过程。该目标模块也就是装入模块。在将一个装入模块装入内存时，可以有三种装入方式：绝对装入、可重定位装入和动态运行时装入。

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。

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。

程序中所使用的绝对地址，可在编译或汇编时给出， 也可由程序员直接赋予。 但在由程序员直接给出绝对地址时，不仅要求程序员熟悉内存的使用情况，而且一旦程序或数据被修改后，可能要改变程序中的所有地址。

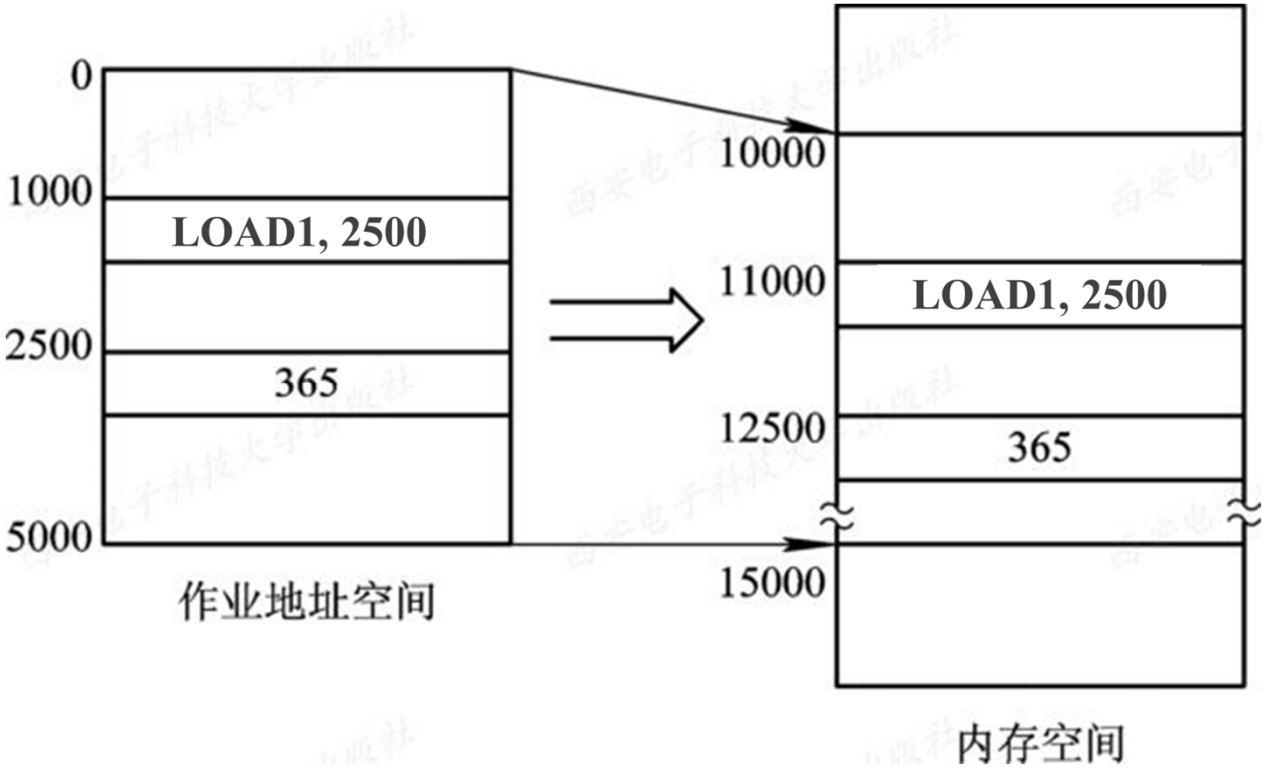
因此，通常是宁可在程序中采用符号地址，然后在编译或汇编时，再将这些符号地址转换为绝对地址。

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，这只适用于单道程序环境。

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。因此，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。

此时，应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。



在装入的时候，将装入模块中指令和数据的逻辑地址修改为物理地址，这一过程就叫**重定位**。

因为地址变换是在装入时一次性完成的，以后不会再改变，所以称为**静态重定位**。

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

可重定位装入方式可将装入模块装入到内存中任何允许的位置，但不允许程序运行时在内存中移动位置。

实际情况是，程序运行过程中，它在内存中的位置可能要经常改变，例如具有对换功能的系统中，一个进程可能被多次换出，又多次换入，每次换入后的位置通常是不同的，这种情况就应该采用动态运行时装入的方式。

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

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，但不允许程序运行时在内存中移动位置。

动态运行时的装入程序在装入模块装入内存后，并不立即把装入模块中的逻辑地址转换为物理地址，而是把这种地址转换推迟到程序真正要执行时才进行，因此装入内存后的所有地址都仍是逻辑地址。

为了使地址转换不影响指令的执行速度，这种方式需要一个重定位寄存器做支持。

4.2.2　程序的链接

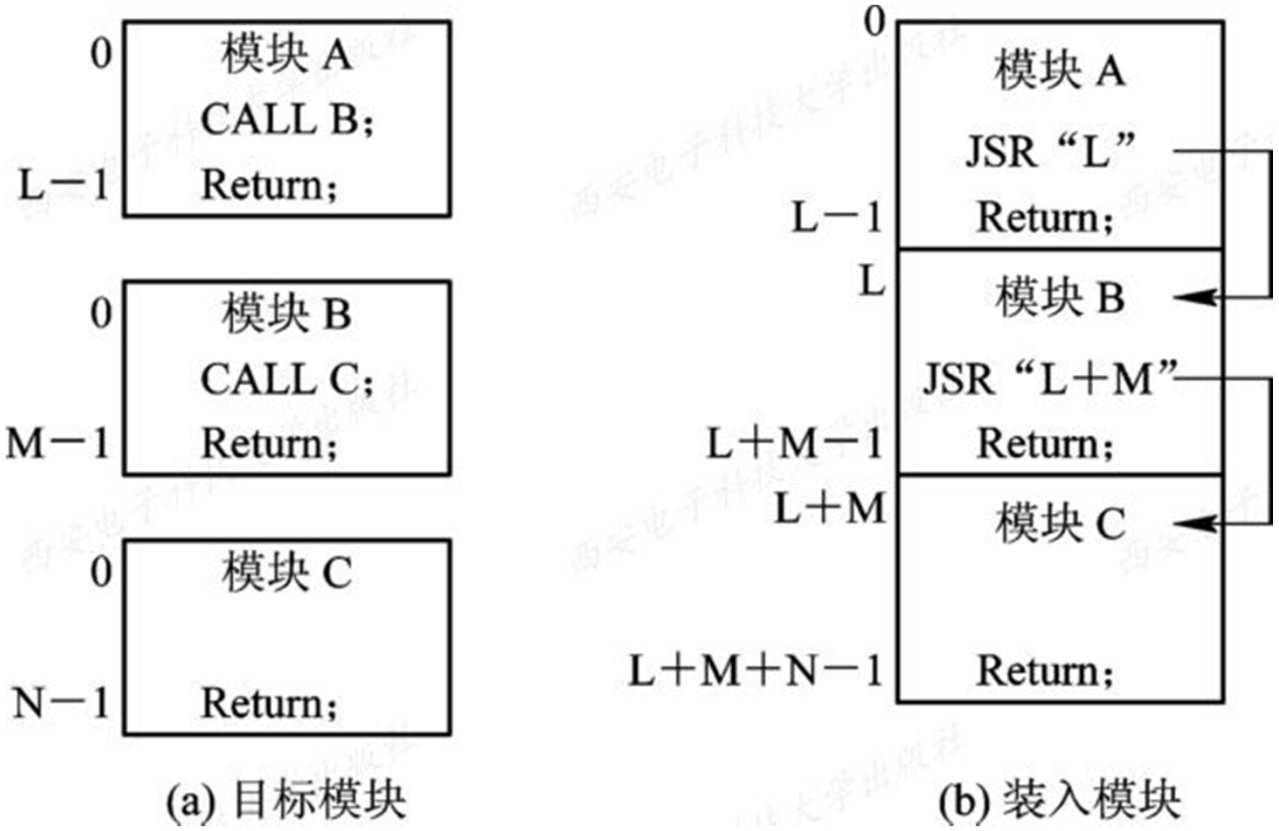
1. 静态链接(Static Linking)方式

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。

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。



这种先进行链接所形成的一个完整的装入模块称为可执行文件。

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。

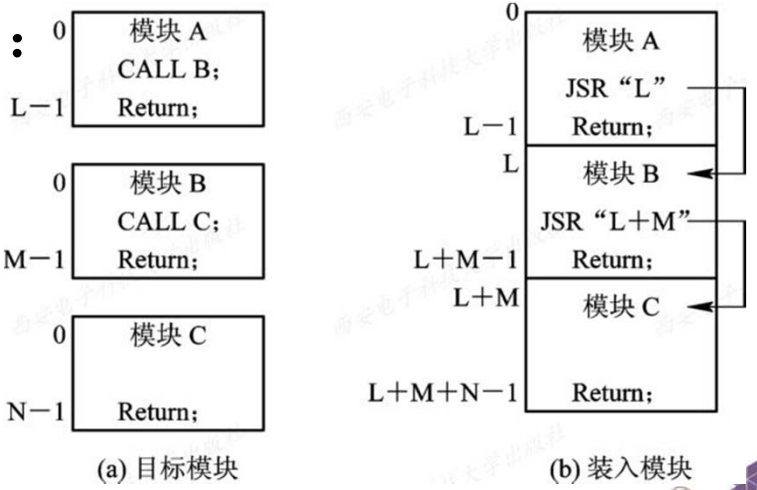
2. 装入时动态链接(Load-time Dynamic Linking)

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。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

(2) 便于实现对目标模块的共享



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全部都装入内存，并在装入时全部链接在一起。显然这是低效的，因为往往会有部分目标模块根本就不运行。

比较典型的例子是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。

运行时动态链接是将某些模块的链接推迟到程序执行时才进行。凡是在执行过程中未被用到的目标模块，都不会调入内存和被链接到装入模块上。

4.3　连续分配存储管理方式

为了能将用户程序装入内存，必须为它分配一定大小的内存空间。

＊分配方式

连续分配：为作业（进程）分配地址连续的存储空间

离散分配：为作业（进程）分配不连续存储空间

- 连续分配方式
  - 分区管理方式
    - 固定分区分配
    - 动态分区分配
    - 可重定位分区分配
- 离散分配方式
  - 分页（页式）存储管理方式
  - 分段（段式）存储管理方式
    - 段页式存储管理方式
  - 虚拟存储器
    - 请求分页存储管理方式
    - 请求分段存储管理方式

- **连续分配存储管理方式**是指一个用户程序分配一个连续的内存空间。又称**分区管理方式**。
- **分区**——是指内存中的一个连续区域。

分区管理方式曾被广泛应用于**20**世纪**60**～**70**年代的**OS**中，至今仍在内存分配方式中占一席之地。

分区分配方式可分为：单一连续分配、固定分区分配、动态分区分配、动态重定位分区分配等。

4.3.1 单一连续分配管理方式

适用于早期单用户单任务OS

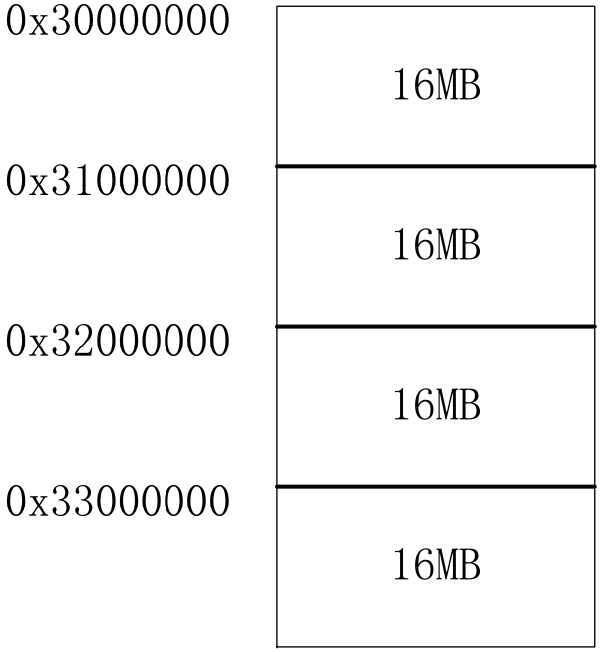
- 单一连续分配管理方式是将内存分为系统区和用户区两个区域。**系统区**提供给操作系统使用，它通常放在内存的低址部分。**用户区**提供给用户程序使用，一次只能装入一个程序运行，即整个内存的用户空间由该程序独占。

4.3.2 固定分区分配

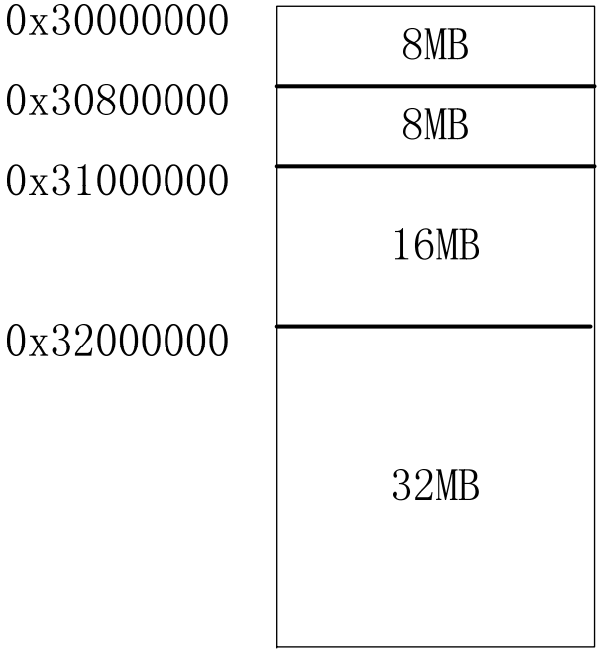
最简单的可运行多道程序的存储器管理方式。

内存可用区划分成若干个大小固定的存区，每个存区分别装入一道作业的代码（数据）。

1. 划分分区的方法



等分方式



差分方式

2. 内存分配

为了便于内存分配，通常将分区按大小排队，建立一张分区说明表。表项：分区起始地址、大小、状态（是否已分配）。

分区号	大小(K)	始址(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

图4-5(a) 分区说明表

图4-5(b) 内存分配情况

分配：查分区说明表，找到一个足够大的空闲分区分配之；

回收：将回收分区对应的分区说明表状态改为“空闲”。

3. 固定分区方式的优缺点

- 内存可同时装入多道作业代码，算法实现简单
- 存在浪费（分区一次性全部分配出去），分区内存在内碎片

4.3.3 动态分区分配（可变分区分配）

算法思想：

事先不划分分区，待作业需要分配内存时，再按需分配划分分区（分区的大小及个数不固定）。

算法实现：

建立相关数据结构，根据分配策略(或算法)设定算法程序.

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。在实现时，涉及到3个问题：数据结构、分配算法、分配及回收操作。

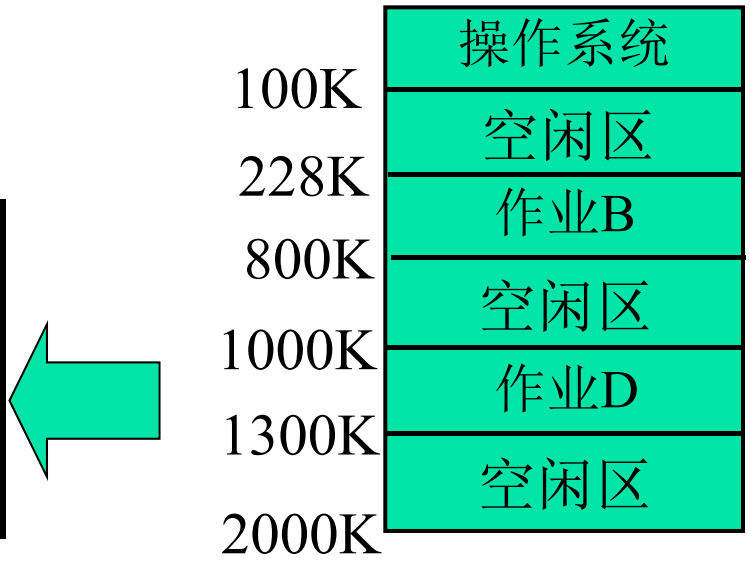
1．动态分区分配中的数据结构

两种常用形式：空闲分区表、空闲分区链

1) 空闲分区表

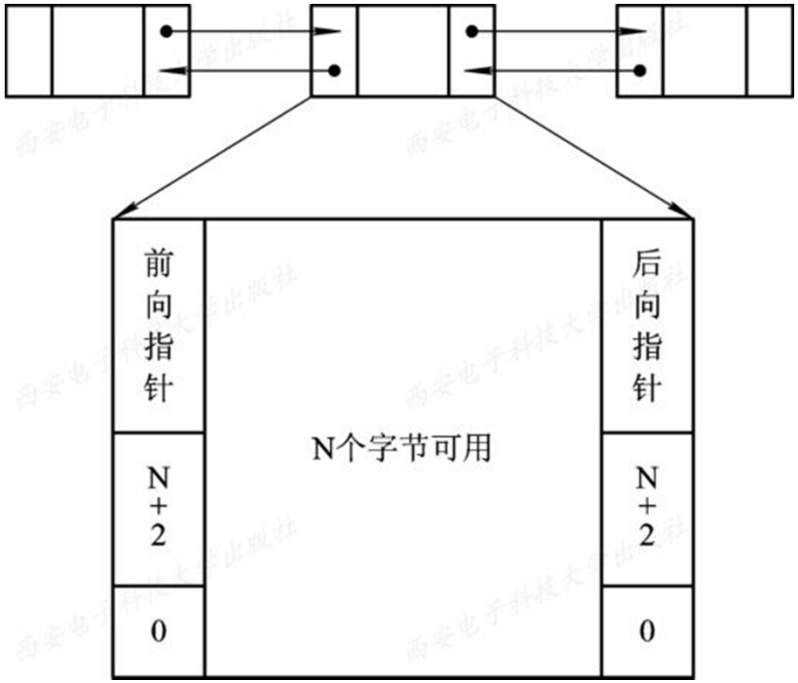
在系统中设置一张空闲分区表，记录每个空闲分区的情况。每个空闲分区占一个表目，表目包括：分区号、分区始址、分区大小等。

分区号	分区始址	分区长度
1	100KB	128KB
2	800KB	200KB
3	1300KB	700KB



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。



- 当分区被分配出去后，把状态位由0改为1，此时前、后向指针无意义。

2．动态分区分配和回收

1）分配内存

分配内存过程如图4-8所示。

图中假设：

- 请求分区大小u.size；
- 空闲分区大小为m.size；
- 不再切割的剩余分区大小为size。

当某空闲分区满足m.size≥u.size时，执行如下操作：

①当m.size-u.size≤size时，将整个分区分配给请求者；

②否则，按作业大小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分区表（链）中。

③将分配区的首地址返回给调用者。

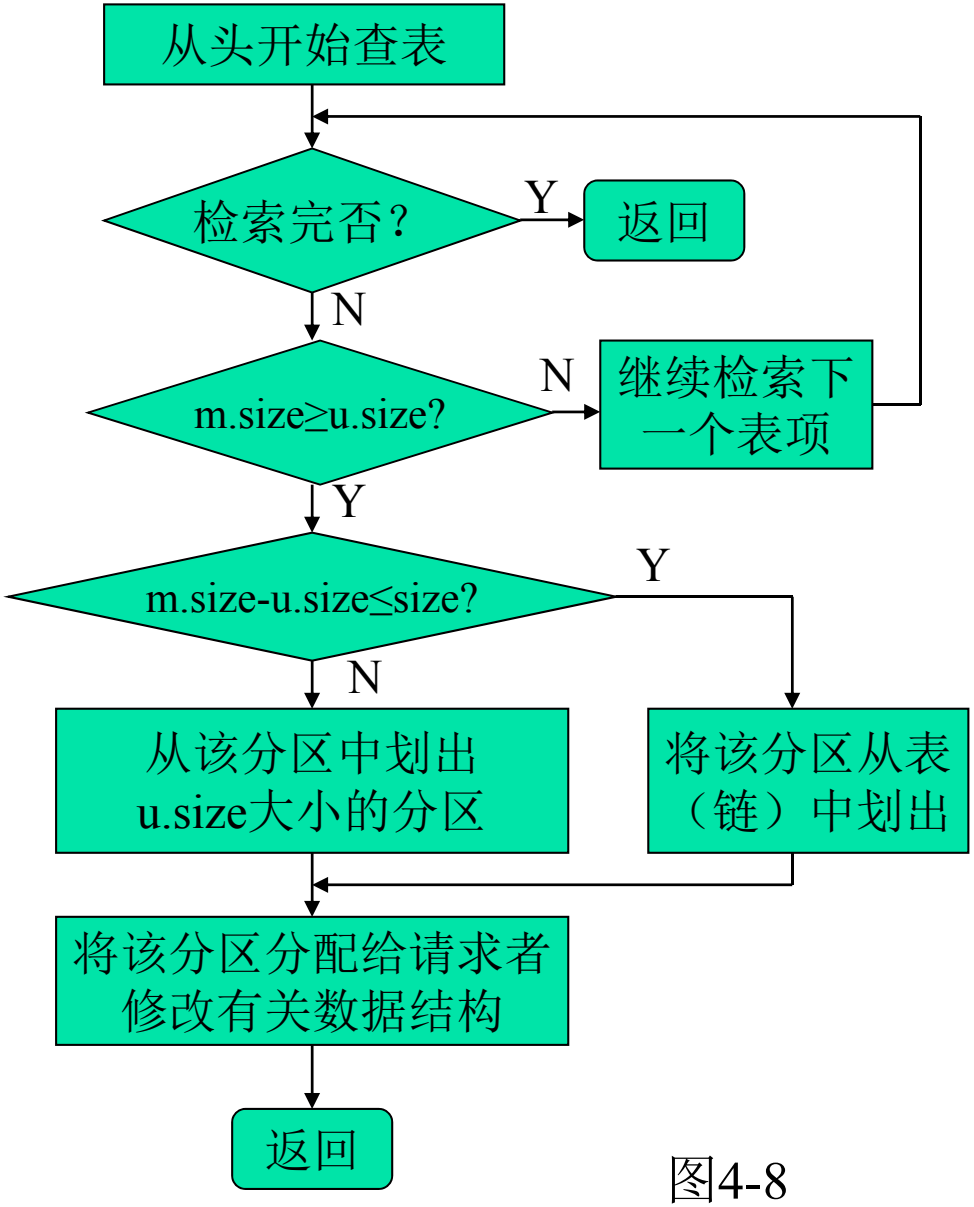


图4-8

2）回收内存

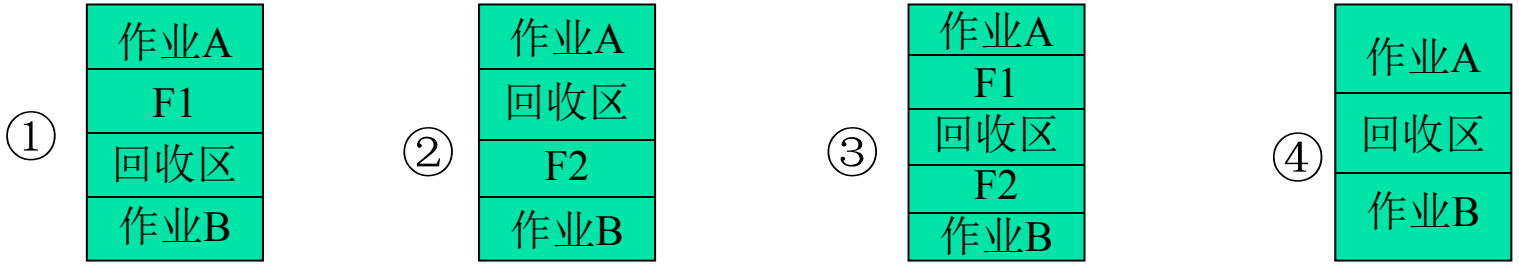
根据回收区的首地址，在空闲分区表（链）找到插入点，此时可能出现4种情况之一（假设空闲分区表按地址从低到高顺序排列）：

①回收区与插入点的前一个空闲分区F1相邻接：将回收区与前一区合并，不必增加新表项，只需修改F1的大小为两者之和。

②回收区与高地址F2分区邻接：此时将回收分区与该分区合并，回收区的首地址为新分区的首地址，大小为两者之和。

③回收区与前后分区F1和F2都邻接：将此3个分区合并，F1（前邻接区）的首地址为新分区的首址，大小为三者之和，取消F2表项。

④回收区与任何空闲区都不邻接：在插入点建立一个新表项，填写回收区的首地址和大小。插入到空闲区表的适当位置（后移插入点后的各个表项）



4.3.4 基于顺序搜索的动态分区分配算法

为了实现动态分区分配，通常是将系统中的空闲分区链接成一个链。

所谓顺序搜索，是指依次搜索空闲分区链上的空闲分区，去寻找一个大小能满足要求的分区。

4.3.4 基于顺序搜索的动态分区分配算法

首次适应算法 → 空闲分区链按地址递增排序。分配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；然后按作业大小划出一块内存空间分配给请求者，余下的空闲分区仍留在链中。

循环首次适应算法 →

最佳适应算法 →

最坏适应算法

4.3.4 基于顺序搜索的动态分区分配算法

首次适应算法

空闲分区链按地址递增排序。分配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；然后按作业大小划出一块内存空间分配给请求者，余下的空闲分区仍留在链中。

- 若从链首直到链尾都找不到一个能满足要求的分区，表明系统中已经没有足够大的内存分配给该进程，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，保留高址部分的大空闲区，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，称为碎片。每次查找又都是从低址部分开始的，这无疑又会增加查找可用空闲分区时的开销。

最坏适应算法

4.3.4 基于顺序搜索的动态分区分配算法

首次适应算法

空闲分区链按地址递增排序。分配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；然后按作业大小划出一块内存空间分配给请求者，余下的空闲分区仍留在链中。

循环首次适应算法

由首次适应算法演变而成。为进程分配内存时，不再是每次从链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，直至找到一个大小能满足要求的空闲分区，从中划出一块与请求大小相等的分区分配给作业。

为实现该算法，应设置一个起始查找指针。

- 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。
- 但会缺乏大的空闲分区。

最坏适应算法

4.3.4 基于顺序搜索的动态分区分配算法

首次适应算法

空闲分区链按地址递增排序。分配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；然后按作业大小划出一块内存空间分配给请求者，余下的空闲分区仍留在链中。

循环首次适应算法

由首次适应算法演变而成。为进程分配内存时，不再是每次从链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，直至找到一个大小能满足要求的空闲分区，从中划出一块与请求大小相等的分区分配给作业。

为实现该算法，应设置一个起始查找指针。

最佳适应算法

每次分配时，总是将能满足要求的最小分区分配给请求者。将空闲分区按其容量从小到大顺序排列——加快查找。

- 孤立地看，最佳适应算法似乎是最佳的，但是在宏观上却并不是——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。

4.3.4 基于顺序搜索的动态分区分配算法

首次适应算法

空闲分区链按地址递增排序。分配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；然后按作业大小划出一块内存空间分配给请求者，余下的空闲分区仍留在链中。

循环首次适应算法

由首次适应算法演变而成。为进程分配内存时，不再是每次从链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，直至找到一个大小能满足要求的空闲分区，从中划出一块与请求大小相等的分区分配给作业。

为实现该算法，应设置一个起始查找指针。

- 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。
- 但会缺乏大的空闲分区。

最坏适应算法

每次分配时，总是将能满足要求的最大分区分配给请求者。将空闲分区按其容量从大到小顺序排列——加快查找。

4.3.5 基于索引搜索的动态分区分配算法

快速适应算法

空闲分区链按容量大小进行分类。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。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。分配时查找索引表，找到合适的链表，摘下该链表里的第一块即可。

伙伴系统

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，l≤k≤m)。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。要为进程分配一个长度为n的存储区域时，计算i值，使得$2^{i-1}<n\leq 2^{i}$然后在大小为2^{i}的空闲分区链中查找。

哈希算法

4.3.5 基于索引搜索的动态分区分配算法

- 若找到，就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，则在分区大小为2^{i+1}的空闲分区链表中寻找。若存在2^{i+1}的一个空闲分区，则把该空闲分区分为相等的两个分区，这两个分区称为一对伙伴，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。
- 若大小为2^{i+1}的空闲分区也不存在，则需要查找大小为2^{i+2}的空闲分区，若找到要对其进行两次分割：第一次，将其分割为大小为2^{i+1}的两个空闲分区，一个用于分配，一个加入大小为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分区分割为2^i的两个分区，一个用于分配，一个加入大小为2^i的空闲分区链表中。
- 若仍找不到，则继续查找大小为2^{i+3}的空闲分区，以此类推。
- 最坏的情况下，可能需要对2^k的空闲分区进行k次分割才能得到所需分区。

- 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。
- 如回收大小为2^i的空闲分区时，若事先已存在2^i的空闲分区，则应将其与伙伴分区合并为大小为2^{i+1}的空闲分区，若事先已存在2^{i+1}的空闲分区，又应继续与其伙伴分区合并为大小为2^{i+2}的空闲分区，以此类推。

4.3.5 基于索引搜索的动态分区分配算法

空闲分区链按容量大小进行分类。对于每一类具有相同容量的

- 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$buddy_k(x)$表示，其通式为：

$$buddy_k(x)=\begin{cases} x+2^k\ (若\ xMOD\ \ 2^{k+1}=0) \\ x-2^k\ (若\ xMOD\ \ 2^{k+1}=2^k) \end{cases}$$

链中查找。

- 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。
- 如回收大小为2^i的空闲分区时，若事先已存在2^i的空闲分区，则应将其与伙伴分区合并为大小为2^{i+1}的空闲分区，若事先已存在2^{i+1}的空闲分区，又应继续与其伙伴分区合并为大小为2^{i+2}的空闲分区，以此类推。

4.3.5 基于索引搜索的动态分区分配算法

快速适应算法

空闲分区链按容量大小进行分类。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。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。分配时查找索引表，找到合适的链表，摘下该链表里的第一块即可。

伙伴系统

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，l≤k≤m)。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。要为进程分配一个长度为n的存储区域时，计算i值，使得$2^{i-1}<n\leq 2^{i}$然后在大小为2^{i}的空闲分区链中查找。

哈希算法

利用哈希快速查找的优点以及空闲分区在空闲分区表中的规律，建立哈希函数，构造以空闲分区大小为关键字的哈希表，该表每个表项记录一个对应的空闲分区链表表头指针。进行空闲分区分配时，根据空闲分区大小，通过哈希函数计算，得到在哈希表中的位置，从而得到空闲分区链表的指针。

4.3.6　动态可重定位分区分配

1. 紧凑

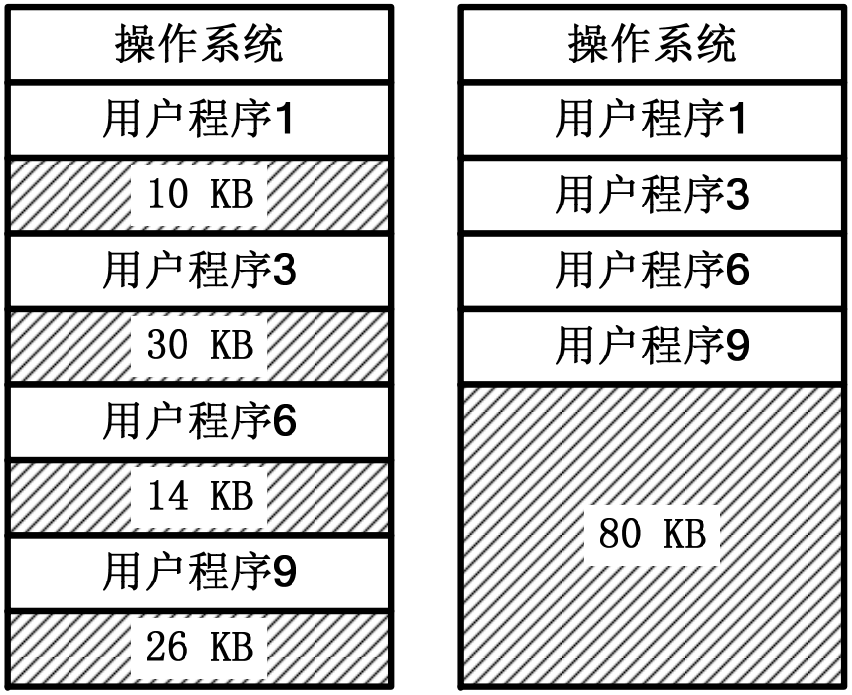
连续分配方式的一个重要特点是，一个系统或用户程序必须被装入一片连续的内存空间中。当一台计算机运行了一段时间后，它的内存空间将会被分割成许多小的分区，而缺乏大的空闲空间。即使这些分散的许多小分区的容量总和大于要装入的程序，但由于这些分区不相邻接，也无法把该程序装入内存。 如图4-11所示。

目的：为了解决内存“碎片”（或称为“零头”）问题。

方法：移动作业，称为“拼接”或“紧凑”——需要动态重定位

紧凑

- 通过作业移动将原来分散的小分区拼接成一个大分区。
- 每次紧凑后，都必须对移动了的程序或数据进行重定位。

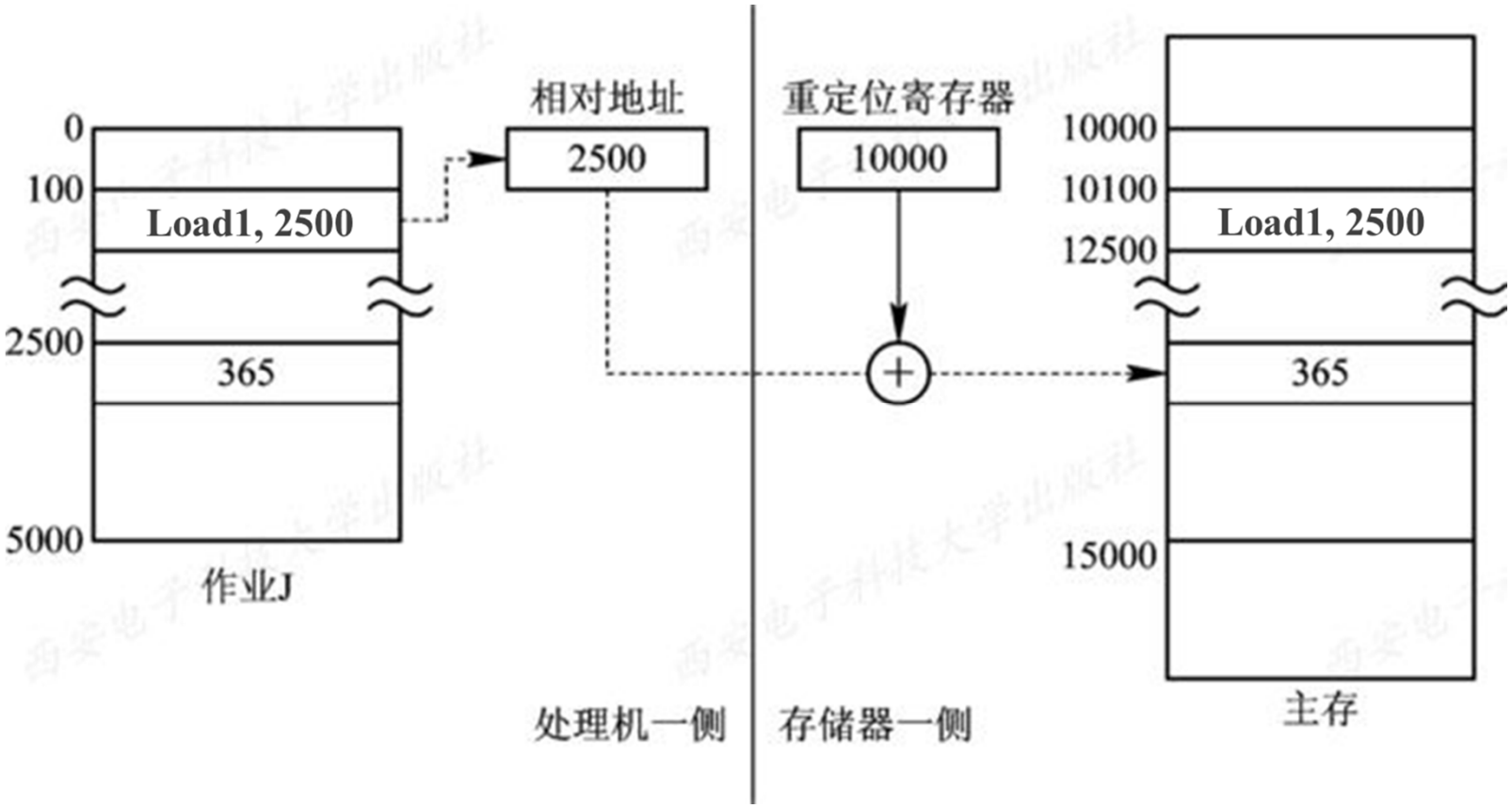


(a) 紧凑前　(b) 紧凑后

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。

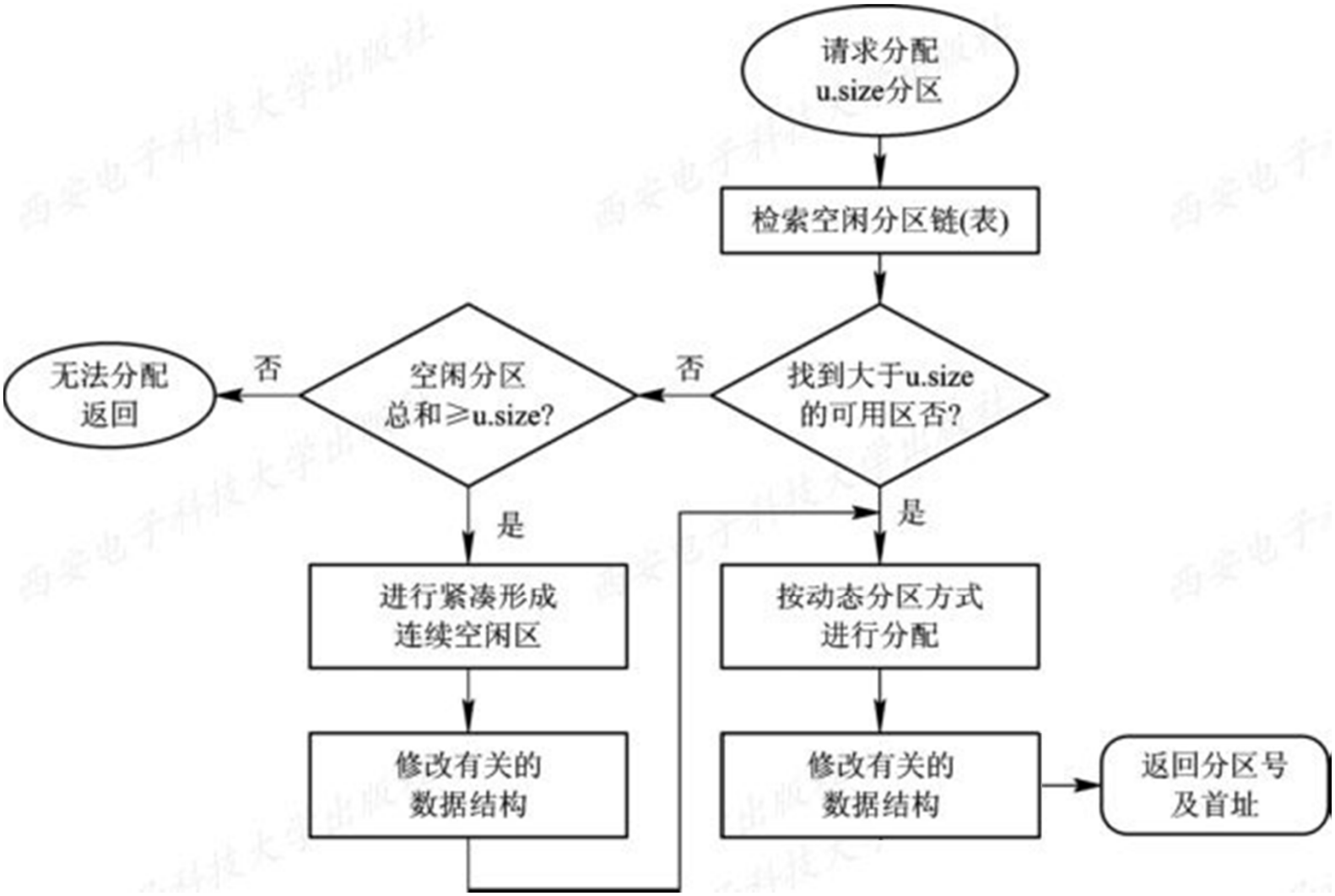


程序从内存的某处移动到另一处时，只需修改基址（重定位）寄存器的值即可。

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，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。


请求分配
u.size分区
检索空闲分区链(表)
找到大于u.size
的可用区否?
否
空闲分区
总和≥u.size?
否
无法分配
返回
是
是
进行紧凑形成
连续空闲区
按动态分区方式
进行分配
修改有关的
数据结构
修改有关的
数据结构
返回分区号
及首址

4.4 对换

1. 对换的引入

- 阻塞进程占用大量内存
- 外存中的等待作业因无内存不能调入运行

资源浪费

对换——把内存中暂时不能运行的进程或进程所需要的程序和数据，调出到外存上，以便腾出足够的内存空间，再把已具有运行条件的进程或进程所需要的程序和数据，调入内存。

整体对换（进程对换）

- 对换以整个进程为单位
- 广泛应用于分时系统中

部分对换

- 对换以“页”为单位——页面对换
- 对换以“段”为单位——分段对换

虚存

本节介绍进程对换，部分对换将在虚拟存储器中介绍。

要实现进程对换，需实现三方面功能：

- 对换空间的管理
- 进程的换出
- 进程的换入

2. 对换空间的管理

把外存分为：**文件区和对换区**

文件区

- 存放文件
- 管理主要目标：提高文件存储空间利用率
- 为此，采用离散分配方式

对换区

- 存放换出的进程
- 管理主要目标：提高进程换入和换出的速度
- 为此，采用连续分配方式

数据结构：对换区的空闲盘块管理的数据结构与动态分区分配方式采用的数据结构相似——空闲区表或空闲区链。

空闲分区表中应包含两个表项：对换区的首址及其大小，它们的单位是盘块号和盘块数。

分配算法：对换区的分配与回收，与动态分区分配雷同，其分配算法可以是首次适应、循环首次适应或最佳适应分配算法。

3. 进程的换出与换入

进程的换出

当一进程由于创建子进程而需要更多的内存空间，但无足够的内存空间等情况发生时，系统应将某个进程换出。

换出过程：

- 选择处于阻塞状态且优先级别最低的进程作为换出进程；
- 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，回收该进程所占用的内存空间，并对该进程的PCB做相应的修改。

在对进程换出时，只能换出非共享的程序和数据段，而对于那些共享的程序和数据段，只要还有进程需要，就不能被换出。

3. 进程的换出与换入

进程的换出

当一进程由于创建子进程而需要更多的内存空间，但无足够的内存空间等情况发生时，系统应将某个进程换出。

换出过程：

- 选择处于阻塞状态且优先级别最低的进程作为换出进程；
- 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，回收该进程所占用的内存空间，并对该进程的PCB做相应的修改。

进程的换入

系统应定时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的进程换入，直至已无可换入的进程或无可换出的进程为止。