第5章 存储管理

5.1 存储管理的功能

存储器是计算机系统的重要资源之一。因为任何程序和数据以及各种控制用的数据结构都必须占用一定的存储空间,因此,存储管理直接影响系统性能。

存储器由内存(primary storage)和外存(secondary storage)组成。内存由顺序编址的块组成,每块包含相应的物理单元。CPU 要通过启动相应的输入输出设备后才能使外存与内存交换信息。本章主要讨论内存管理问题,内容包括几种常用的内存管理方法、内存的分配和释放算法、虚拟存储器的概念、控制主存和外存之间的数据流动方法、地址变换技术和内存数据保护与共享技术等。下面先介绍存储管理的功能。

5.1.1 虚拟存储器

虚拟存储器是存储管理的核心概念。现代计算机系统的物理存储器都分为内存和外存,其理由是内存价格昂贵,不可能用大容量的内存存储所有被访问的或不被访问的程序与数据段。而外存尽管访问速度较慢,但价格便宜,适合存放大量信息。实验证明,在一个进程的执行过程中,其大部分程序和数据并不经常被访问。这样,存储管理系统把进程中那些不经常被访问的程序段和数据放入外存中,待需要访问它们时再将它们调入内存。那么,对于那些一部分数据和程序段在内存而另一部分在外存的进程,怎样安排它们的地址呢?

通常由用户编写的源程序,首先要由编译程序编译成 CPU 可执行的目标代码。然后,链接程序把一个进程的不同程序段链接起来以完成所要求的功能。显然,对于不同的程序段,应具有不同的地址。有两种方法安排这些编译后的目标代码的地址。一种方法是按照物理存储器中的位置赋予实际物理地址。这种方法的好处是 CPU 执行目标代码时的执行速度高。但是,由于物理存储器的容量限制,能装入内存并发执行的进程数将会大大减少,对于某些较大的进程来说,当其所要求的总内存容量超过内存容量时将会无法执行。另外,由于编译程序必须知道内存的当前空闲部分及其地址,并且把一个进程的不同程序段连续地存放起来,因此编译程序将非常复杂。

另一种方法是编译链接程序把用户源程序编译后链接到一个以 0 地址为始地址的线性或多维虚拟地址空间。这里,链接既可以是在程序执行以前由链接程序完成的静态链接,也可以是在程序执行过程中由于需要而进行的动态链接。而且,每一个进程都拥有这样一个空间(这个空间是一维的还是多维的由存储管理方式决定)。每个指令或数据单元都在这个虚拟空间中拥有确定的地址,这个地址称为虚拟地址(virtual address)。显然,进程在该空间的地址排列可以是非连续的,其实际物理地址由虚拟地址到实际物理地址的变换得到。由源程序到实际存放该程序指令或数据的内存物理位置的变换如图 5.1 所示。

进程中的目标代码、数据等的虚拟地址组成的虚拟空间称为虚拟存储器(virtual store

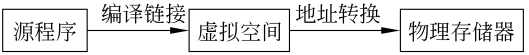


图 5.1　地址变换与物理存储器

或 virtual memory)。虚拟存储器不考虑物理存储器的大小和信息存放的实际位置,只规定每个进程中互相关连的信息的相对位置。与实际物理存储器数量有限,且被所有进程共享不一样,每个进程都拥有自己的虚拟存储器,且虚拟存储器的容量是由计算机的地址结构和寻址方式确定的。例如,直接寻址时,如果 CPU 的有效地址长度为 16 位,则其寻址范围为 0～64K。

图 5.1 中的编译和链接主要是语言系统的设计问题,而虚拟存储器到物理存储器的变换是操作系统必须解决的问题。要实现这个变换,必须要有相应的硬件支持,并使这些硬件能够完成统一管理内存和外存之间数据和程序段自动交换的虚拟存储器功能。即,由于每个进程都拥有自己的虚存,且每个虚存的大小不受实际物理存储器的限制,因此,系统不可能提供足够大的内存来存放所有进程的内容。内存中只能存放那些经常被访问的程序和数据段等。这就需要有相当大的外部存储器,以存储那些不经常被访问或在某一段时间内不会被访问的信息。待到进程执行过程中需要这些信息时,再从外存中自动调入内存。至于如何具体实现和管理虚拟存储器,将在后面有关章节介绍。

5.1.2　地址变换

内存地址的集合称为内存空间或物理地址空间。内存中,每一个存储单元都与相应的称为内存地址的编号相对应。显然,内存空间是一维线性空间。

怎样把几个虚存的一维线性空间或多维线性空间变换到内存的唯一的一维物理线性空间呢? 这涉及两个问题。

第一个问题是虚拟空间的划分问题。例如进程的正文段和数据段应该放置在虚拟空间的什么地方。虚拟空间的划分使得编译链接程序可以把不同的程序模块(它们可能是用不同的高级语言编写的)链接到一个统一的虚拟空间中。虚拟空间的划分与计算机系统结构有关,例如,VAX-11 型机中的虚拟空间划分为进程空间和系统空间两大部分,而进程空间又更进一步划分为程序区和控制区。VAX-11 的虚拟空间容量为 2^{32} 个单元,其中程序区占 2^{30} 个单元,用来存放用户程序,程序段以零为基址动态地向高地址方向增长,最大可达 $2^{30}-1$ 号单元。控制区也占 2^{30} 个单元,存放各种方式和状态下的堆栈结构及数据等,其虚拟地址由 $2^{31}-1$ 号地址开始由高向低地址方向增长。系统空间占 2^{31} 个单元,用来存放操作系统程序。VAX-11 机的虚拟空间结构如图 5.2 所示。

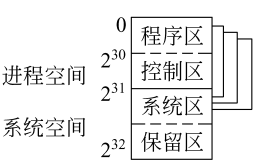


图 5.2　虚拟空间的划分

第二个问题是把虚拟空间中已链接和划分好的内容装入内存,并将虚拟地址映射为内存地址的问题,称之为地址重定位或地址映射。地址映射就是要建立虚拟地址与内存地址的关系。实现地址重定位或地址映射的方法有两种:静态地址重定位和动态地址重定位。

1. 静态地址重定位

静态地址重定位(static address relocation)是在虚拟空间程序执行之前由装配程序完

成地址映射工作。假定分配程序已分配了一块首地址为 BA 的内存区给虚拟空间内的程序段，且每条指令或数据的虚拟地址为 VA，那么，该指令或数据对应的内存地址为 MA，从而完成程序中所有地址部分的修改，以保证 CPU 的正确执行。显然，对于虚拟空间内的指令或数据来说，静态地址重定位只完成一个首地址不同的连续地址变换。它要求所有待执行的程序必须在执行之前完成它们之间的链接，否则将无法得到正确的内存地址和内存空间。

静态重定位的优点是不需要硬件支持。但是，使用静态重定位方法进行地址变换无法实现虚拟存储器。这是因为，虚拟存储器呈现在用户面前的是一个在物理上只受内存和外存总容量限制的存储系统，这要求存储管理系统只把进程执行时频繁使用和立即需要的指令与数据等存放在内存中，而把那些暂时不需要的部分存放在外存中，待需要时自动调入，以提高内存的利用率和并发执行的作业道数。显然，这是与静态重定位方法矛盾的，静态重定位方法将程序一旦装入内存之后就不能再移动，并且必须在程序执行之前将有关部分全部装入。

静态重定位的另一个缺点是必须占用连续的内存空间，这就难以做到程序和数据的共享。

2. 动态地址重定位

动态地址重定位(dynamic address relocation)是在程序执行过程中，在 CPU 访问内存之前，将要访问的程序或数据地址转换成内存地址。动态重定位依靠硬件地址变换机构完成。

地址重定位机构需要一个(或多个)基地址寄存器 BR 和一个(或多个)程序虚拟地址寄存器 VR。指令或数据的内存地址 MA 与虚拟地址的关系为

$$MA=(BR)+(VR)$$

这里，(BR)与(VR)分别表示寄存器 BR 与 VR 中的内容。

动态重定位过程可参看图 5.3。

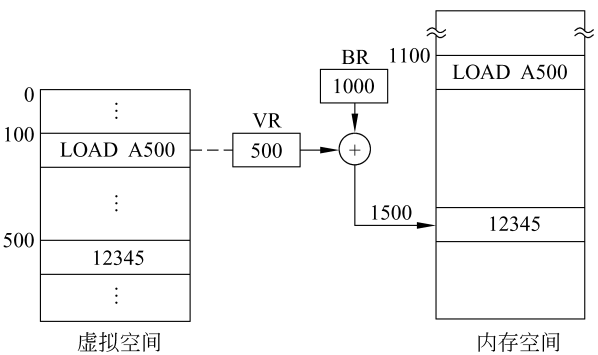


图 5.3　动态地址重定位

其具体过程如下：

(1) 设置基地址寄存器 BR 和虚拟地址寄存器 VR。

(2) 将程序段装入内存，且将其占用的内存区首地址送 BR 中。例如，在图 5.3 中，(BR)=1000。

(3) 在程序执行过程中，将所要访问的虚拟地址送入 VR 中，例如在图 5.3 中执行 LOAD A 500 语句时，将所要访问的虚拟地址 500 放入 VR 中。

(4) 地址变换机构把 VR 和 BR 的内容相加,得到实际访问的物理地址。

动态重定位的主要优点如下:

(1) 可以对内存进行非连续分配。显然,对于同一进程的各分散程序段,只要把各程序段在内存中的首地址统一存放在不同的 BR 中,则可以由地址变换机构变换得到正确的内存地址。

(2) 动态重定位提供了实现虚拟存储器的基础。因为动态重定位不要求在作业执行前为所有程序分配内存,也就是说,可以部分地、动态地分配内存。从而,可以在动态重定位的基础上,在执行期间采用请求方式为那些不在内存中的程序段分配内存,以达到内存扩充的目的。

(3) 有利于程序段的共享。

5.1.3 内外存数据传输的控制

要实现内存扩充,在程序执行过程中,内存和外存之间必须经常地交换数据。也就是说,把那些即将执行的程序和数据段调入内存,而把那些处于等待状态的程序和数据段调出内存。那么,按什么样的方式来控制内存和外存之间的数据流动呢? 最基本的控制办法有两种,一种是用户程序自己控制,另一种是操作系统控制。

用户程序自己控制内外存之间的数据交换的例子是覆盖(overlay)。覆盖技术要求用户清楚地了解程序的结构,并指定各程序段调入内存的先后次序。覆盖是一种早期的内存扩充技术。使用覆盖技术,用户负担很大,且程序段的最大长度仍受内存容量限制。因此,覆盖技术不能实现虚拟存储器。

操作系统控制方式又可进一步分为两种,一种是交换(swapping)方式,另一种是请求调入(on demand)方式和预调入(on prefetch)方式。

交换方式由操作系统把那些在内存中处于等待状态的进程换出内存,而把那些等待事件已经发生、处于就绪态的进程换入内存。

请求调入方式是在程序执行时,如果所要访问的程序段或数据段不在内存中,则操作系统自动地从外存将有关的程序段和数据段调入内存的一种操作系统控制方式。而预调入则是由操作系统预测在不远的将来会访问到的那些程序段和数据段部分,并在它们被访问之前选择适当的时机将它们调入内存的一种数据流控制方式。

由于交换方式一般不进行部分交换,即每次交换都交换那些除去常驻内存部分后的整个进程,而且,即使能完成部分交换,也不是按照执行的需要而交换程序段,只是把受资源限制、暂时不能执行的程序段换出内存。因此,虽然交换方式也能完成内存扩充任务,但它仍未实现那种所谓自动覆盖、内存和外存统一管理、进程大小不受内存容量限制的虚拟存储器。只有请求调入方式和预调入方式可以实现进程大小不受内存容量限制的虚拟存储器。有关实现方法将在后面章节中讲述。

5.1.4 内存的分配与回收

内存的分配与回收是内存管理的主要功能之一。无论采用哪一种管理和控制方式,能否把外存中的数据和程序调入内存,取决于能否在内存中为它们安排合适的位置。因此,存储管理模块要为每一个并发执行的进程分配内存空间。另外,当进程执行结束之后,存储管

理模块又要及时回收该进程所占用的内存资源，以便给其他进程分配空间。

为了有效合理地利用内存，设计内存的分配和回收方法时，必须考虑和确定以下几种策略和数据结构：

(1) 分配结构。登记内存使用情况以及供分配程序使用的表格与链表。例如内存空闲区表、空闲区队列等。

(2) 放置策略。确定调入内存的程序和数据在内存中的位置。这是一种选择内存空闲区的策略。

(3) 交换策略。在需要将某个程序段和数据段调入内存时，如果内存中没有足够的空闲区，由交换策略来确定把内存中的哪些程序段和数据段调出内存，以便腾出足够的空间。

(4) 调入策略。外存中的程序段和数据段什么时间按什么样的控制方式进入内存。调入策略与 5.1.3 节中所述内外存数据流动控制方式有关。

(5) 回收策略。回收策略包括两点，一是回收的时机，二是对所回收的内存空闲区和已存在的内存空闲区的调整。

5.1.5 内存信息的共享与保护

内存信息的共享与保护也是内存管理的重要功能之一。在多道程序设计环境下，内存中的许多用户程序或系统程序和数据段可供不同的用户进程共享。这种资源共享将会提高内存的利用率。但是，反过来说，除了被允许共享的部分之外，又要限制各进程只在自己的存储区活动，各进程不能对别的进程的程序和数据段产生干扰和破坏，因此必须对内存中的程序和数据段采取保护措施。

常用的内存信息保护方法有硬件法、软件法和软硬件结合法 3 种。

上下界保护法是一种常用的硬件保护法。上下界存储保护技术要求为每个进程设置一对上下界寄存器，其中装有被保护程序和数据段的起始地址和终止地址。在程序执行过程中，在对内存进行访问操作时首先进行访址合法性检查，即检查经过重定位后的内存地址是否在上下界寄存器所规定的范围之内。若在规定的范围之内，则访问是合法的；否则是非法的，并产生访址越界中断。上下界保护法的保护原理如图 5.4 所示。

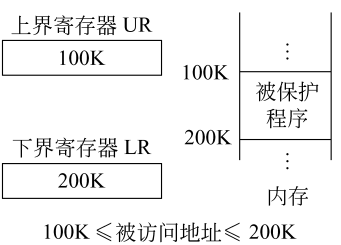


图 5.4 上下界寄存器保护法

另外，保护键法也是一种常用的存储保护法。保护键法为每一个被保护存储块分配一个单独的保护键。在程序状态字中则设置相应的保护键开关字，对不同的进程赋予不同的开关代码与被保护的存储块中的保护键匹配。保护键可设置成对读写同时保护，也可以设置成只对读或写进行单项保护。例如，图 5.5 中的保护键 0 就是对 2K 到 4K 的存储区进行读写同时保护，而保护键 2 则只对 4K 到 6K 的存储区进行写保护。如果开关字与保护键匹配或存储块未受到保护，则访问该存储块是允许的，否则将产生访问出错中断。

另外一种常用的内存保护方式是界限寄存器与 CPU 的用户态或核心态工作方式相结合的保护方式。在这种保护方式下，用户态进程只能访问那些在界限寄存器所规定范围内的内存部分，而核心态进程则可以访问整个内存地址空间。UNIX 系统就是采用的这种内存保护方式。

图 5.5 保护键保护法

5.2 分区存储管理

分区管理是把内存划分成若干个大小不等的区域，除操作系统占用一个区域之外，其余由多道环境下的各并发进程共享。分区管理是满足多道程序设计的一种最简单的存储管理方法。

下面结合分区原理来讨论分区存储管理时的虚存实现、地址变换、内存的分配与释放以及内存信息的共享与保护等问题。

5.2.1 分区管理基本原理

分区管理的基本原理是给每一个内存中的进程划分一块适当大小的存储区，以连续存储各进程的程序和数据，使各进程得以并发执行。按分区的时机，分区管理可以分为固定分区和动态分区两种方法。

1. 固定分区法

固定分区法就是把内存固定地划分为若干个大小不等的区域。分区划分的原则由一般系统操作员或操作系统决定。例如可划分为长作业分区和短作业分区。分区一旦划分结束，在整个执行过程中每个分区的长度和内存的总分区个数将保持不变。

系统对内存的管理和控制通过数据结构——分区说明表进行，分区说明表说明各分区号、分区大小、起始地址和是否是空闲区(分区状态)。内存的分配释放、存储保护以及地址变换等都通过分区说明表进行。图 5.6 给出了固定分区时分区说明表和对应内存状态的例子。

图中，操作系统占用低地址部分的 20K，其余空间被划分为 4 个分区，其中 1、2、3 号分区已分配，4 号分区未分配。

2. 动态分区法

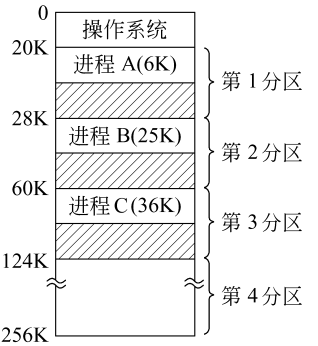
与固定分区法相比，动态分区法在作业执行前并不建立分区，分区的建立是在作业的处理过程中进行的，且其大小可随作业或进程对内存的要求而改变。这就改变了固定分区法中那种即使是小作业也要占据大分区的浪费现象，从而提高了内存的利用率。

采用动态分区法，在系统初启时，除了操作系统中常驻内存部分之外，只有一个空闲分区。随后，分配程序将该区依次划分给调度选中的作业或进程。图 5.7 给出了 FIFO 调度方式时的内存初始分配情况。

随着进程的执行，会出现一系列的分配和释放。如在某一时刻，进程 C 执行结束并释

区号	分区长度	起始地址	状态
1	8K	20K	已分配
2	32K	28K	已分配
3	64K	60K	已分配
4	132K	124K	未分配

(a) 分区说明表



(b) 内存状态

图 5.6　固定分区法

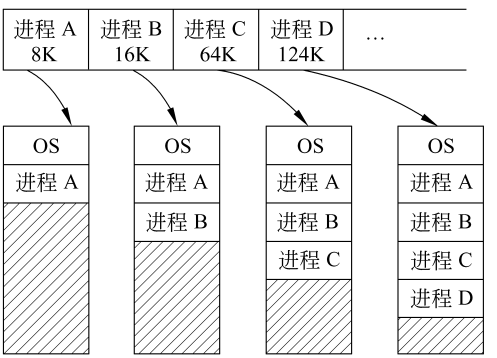


图 5.7　内存初始分配情况

放内存之后，管理程序又要为另两个进程 E(设需内存 50K)和 F(设需内存 16K)分配内存。如果分配的空闲区比所要求的大，则管理程序将该空闲区分成两个部分，其中一部分成为已分配区，而另一部分成为一个新的小空闲区。图 5.8 给出了采用最先适应算法(first fit)分配内存时进程 E 和进程 F 得到内存以及进程 B 和进程 D 释放内存的内存分配变化过程。如图 5.8 所示，在管理程序回收内存时，如果被回收分区有和它邻接的空闲分区存在，则要进行合并。

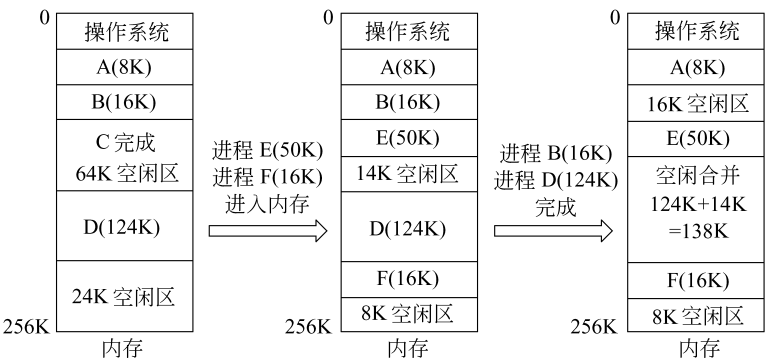
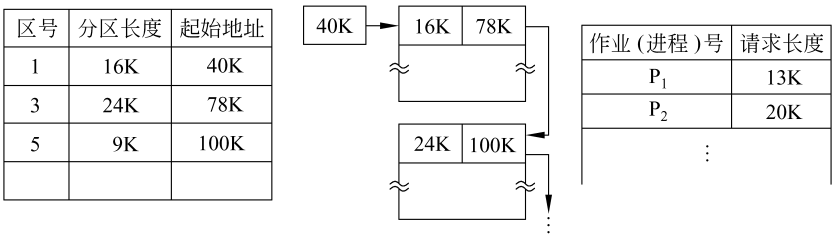


图 5.8　内存分配变化过程

与固定分区法相同，动态分区法也要使用分区说明表等数据结构对内存进行管理。除了分区说明表之外，动态分区法还把内存中的可用分区单独构成可用分区表或可用分区自由链，以描述系统内的内存资源。与此相对应，请求内存资源的作业或进程也构成一个内存资源请求表。图 5.9 给出了可用表、自由链和请求表的例子。



(a) 可用表　　(b) 自由链　　(c) 请求表

图 5.9　可用表、自由链及请求表

可用表的每个表目记录一个空闲区，主要参数包括区号、分区长度和起始地址。采用表格结构，管理过程比较简单，但表的大小难以确定，要占用一部分内存。

自由链则是利用每个内存空闲区的头几个单元存放本空闲区的大小及下个空闲区的起始地址，从而把所有的空闲区链接起来。然后，系统再设置一个自由链首指针让其指向第一个空闲区，这样，管理程序可通过链首指针查到所有的空闲区。采用自由链法管理空闲区，查找时要比可用表困难，但由于自由链指针是利用空闲区自身的单元，所以不必占用额外的内存区。

请求表的每个表目描述请求内存资源的作业或进程号以及所请求的内存大小。

无论是采用可用表方式还是自由链方式，可用表或自由链中的各个项都要按照一定的规则排列，以利查找和回收。下面讨论分区法的分区分配与回收问题。

5.2.2　分区的分配与回收

1. 固定分区时的分配与回收

固定分区法时的内存分配与回收较为简单，当用户程序要装入执行时，通过请求表提出内存分配要求和所要求的内存空间大小。存储管理程序根据请求表查询分区说明表，从中找出一个满足要求的空闲分区，并将其分配给申请者。固定分区时的分配算法如图 5.10 所示。

固定分区的回收更加简单。当进程执行完毕，不再需要内存资源时，管理程序将对应的分区状态置为未使用即可。

2. 动态分区时的分配与回收

动态分区时的分配与回收主要解决 3 个问题：

(1) 对于请求表中的要求内存长度，从可用表或自由链中寻找出合适的空闲区分配程序。

(2) 分配空闲区之后，更新可用表或自由链。

(3) 进程或作业释放内存资源时，和相邻的空闲区进行链接合并，更新可用表或自由链。

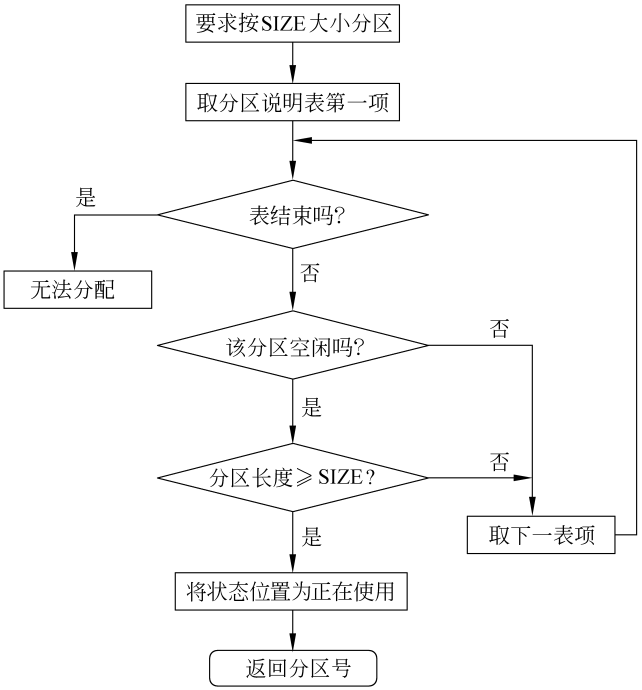


图 5.10　固定分区时的分配算法

从可用表或自由链中寻找空闲区的常用方法有 3 种：最先适应算法(first fit algorithm)、最佳适应算法(best fit algorithm)和最坏适应算法(worst fit algorithm)。这 3 种方法要求可用表或自由链按不同的方式排列。下面分别介绍这 3 种方法。

1) 最先适应算法

最先适应算法要求可用表或自由链按起始地址递增的次序排列。该算法的最大特点是一旦找到大于或等于所要求内存长度的分区，则结束探索。然后，该算法从所找到的分区中划出所要求的内存长度分配给用户，并把余下的部分进行合并(如果有相邻空闲区存在)后留在可用表中，但要修改其相应的表项。最先适应算法如图 5.11 所示。

2) 最佳适应算法

最佳适应算法要求按从小到大的次序组成空闲区可用表或自由链。当用户作业或进程申请一个空闲区时，存储管理程序从表头开始查找，当找到第一个满足要求的空闲区时，停止查找。如果该空闲区大于请求表中的请求长度，则与最先适应算法时相同，将减去请求长度后的剩余空闲区部分留在可用表中。

3) 最坏适应算法

最坏适应算法要求空闲区按其大小递减的顺序组成空闲区可用表或自由链。当用户作业或进程申请一个空闲区时，先检查空闲区可用表或自由链的第一个空闲可用区的大小是否大于或等于所要求的内存长度，若可用表或自由链的第一个项长度小于所要求的，则分配失败，否则从空闲区可用表或自由链中分配相应的存储空间给用户，然后修改空闲区可用表或自由链。

读者可以自行画出最佳适应法和最坏适应法的流程框图。

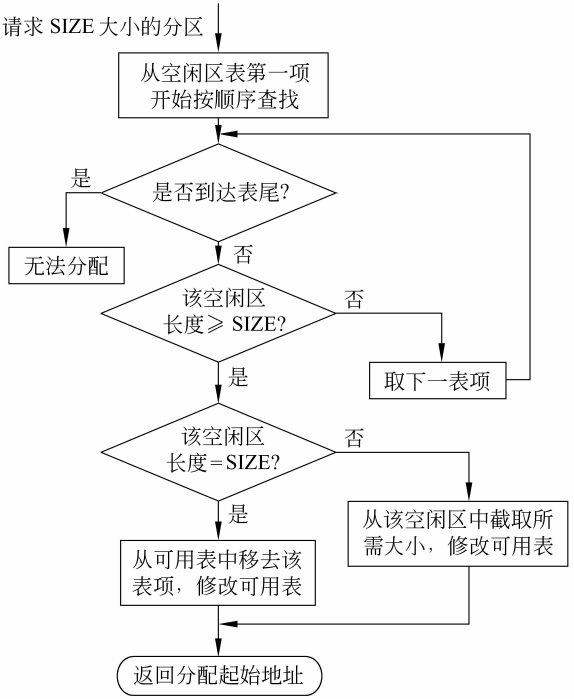


图 5.11　最先适应算法

3. 动态分区时的回收与拼接

当用户作业或进程执行结束时，存储管理程序要收回已使用完毕的空闲区（称为释放区），并将其插入空闲区可用表或自由链。这里，在将回收的空闲区插入可用表或自由链时，和分配空闲区时一样，也要遇到剩余空闲区拼接问题。如果不对空闲区进行拼接，则由于每个作业或进程所要求的内存长度不一样而出现大量分散、较小的空闲区。这就造成大量的内存浪费。解决这个问题的办法之一就是在空闲区回收时或在内存分配时进行空闲区拼接，以把不连续的零散空闲区集中起来。

在将一个新的空闲区插入可用表或自由链时，该空闲区和上下相邻区的关系是下述 4 种关系之一（如图 5.12 所示）。

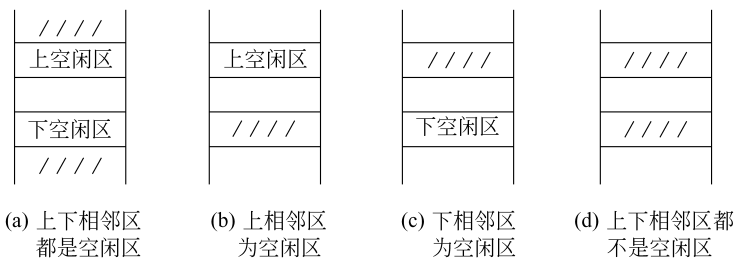


图 5.12　空闲区的合并

（1）该空闲区的上下两相邻分区都是空闲区。

（2）该空闲区的上相邻区是空闲区。