第5章　内存管理

5.1　背　　景

内存(memory)也称为主存,是计算机系统的核心之一,一般是CPU能直接寻址的唯一存储器。它用于保存进程运行时的程序和数据。CPU在获取指令周期时从内存中读取指令,而在获取数据周期时在内存中对数据进行读取或写入。由于内存容量有限,无法装入所有的程序与数据,现在的计算机系统采用多层结构的存储系统。由于程序执行时存在顺序性和局部性,因此,在内存中仅装入程序当前运行所需的部分,其他部分存储在外存,在需要时再装入。内存管理的任务是:记录哪些内存正在使用、哪些内存是空闲的,在进程需要时为其分配内存,在进程使用完内存后释放内存。合理的内存管理策略不仅直接影响内存的利用率,而且对提升系统性能有重要作用。本章主要讨论内存管理问题,内容包括虚拟内存的概念、现代操作系统中常用的内存管理方法、内存的分配和释放算法、控制内存和外存之间的数据流动的方法、地址变换技术、内存数据保护和共享技术等。

5.1.1　虚拟内存

虚拟内存是内存管理的核心概念。现代计算机系统的存储器都分为内存和外存,其原因是: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;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,内存管理系统把程序中不经常被进程访问的程序和数据放入外存,待需要访问它们时再将它们调入内存。那么,对于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,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,连接程序把一个进程的不同程序段连接起来以完成程序的功能。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内存中的位置赋予物理地址。这种方法的好处是CPU执行目标代码时的执行速度高。但是,由于物理内存的容量限制,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可用物理内存容量时将无法执行。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放入内存,因此编译程序将非常复杂。

另一种方法是编译和连接程序把用户源程序经编译后连接到一个以0地址为起始地址的线性或多维虚拟地址空间。这个连接过程既可以是在程序执行以前由连接程序完成的静态连接,也可以是在程序执行过程中由于需要而进行的动态连接。而且,每一个执行过程都拥有这样一个空间(这个空间是一维的还是多维的由内存管理方式决定)。每个指令或数据

单元都在这个虚拟地址空间中拥有确定的地址，这个地址称为虚拟地址(virtual address)。显然，一个程序在执行过程中的地址排列可以是非连续的，其物理地址由虚拟地址变换得到。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地址变换如图 5.1 所示。

图 5.1 地址变换

一个程序在执行过程中的目标代码、数据等的虚拟地址组成的虚拟空间称为虚拟内存(virtual store 或 virtual memory)。虚拟内存不考虑物理内存的大小和信息存放的实际位置，只规定执行指令和数据之间互相关联信息的相对位置。与物理内存容量有限且被所有执行程序共享不一样，每个执行中的程序都有属于自己的虚拟内存，且虚拟内存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 0～64KB；而间接寻址时，其虚拟内存的容量就会大得多。

图 5.1 中的编译和连接主要是编译系统的工作，而虚拟内存到物理内存的变换是操作系统必须解决的问题。要实现这个变换，不仅要有相应的地址变换与管理算法，还要有相应的硬件支持。这些硬件和地址变换与管理算法一起完成管理内存和外存之间数据和程序的自动交换，实现虚拟内存功能。即，由于每个执行中的程序都拥有自己的虚拟内存，且每个虚拟内存的大小不受物理内存的容量限制，而系统不可能提供足够大的物理内存来存放所有程序和数据的内容，只能存放那些经常被访问的程序和数据，这就需要有相当大的外存，以存储不经常被访问或在某一段时间内不会被访问的程序与数据，待到执行过程中需要这些程序和数据时，再将其从外存中自动调入内存。至于如何具体实现和管理虚拟内存，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。在内存中，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。

怎样把不同虚拟内存的一维线性空间或多维线性空间变换到物理内存的唯一的物理地址空间呢？这涉及两个问题。

一个问题是虚拟地址空间的地址划分。例如，一个执行程序和数据应该放置在虚拟地址空间的什么地方，不同的程序模块之间如何连接，等等。虚拟空间的划分使得编译和连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)连接到一个统一的虚拟地址空间中去。虚拟地址空间的划分与计算机系统结构有关。例如，有的计算机把虚拟地址空间划分为用户空间和系统空间两大部分，而用户空间又进一步被划分为程序区和控制区。

一个地址指令为 32 位的虚拟空间容量为 2^{32} 个单元。可以划分出 2^{30} 个单元，用来存放用户程序，其存放方式可以是以 0 为起始地址(0 号单元)，动态地向高地址方向增长，最大可达 $2^{30}-1$ 号单元。控制区也可占 2^{30} 个单元，存放各种计算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和保留备用。上述实例的虚拟空间划分如图 5.2 所示。

第二个问题是如何把虚拟地址空间中已连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，这称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对应关系。在内存管理中，实现虚拟地址到内存地址重定位的方法有两种：静态地址重定位和动态地址重定位。

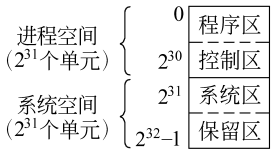


图 5.2 虚拟空间划分

1. 静态地址重定位

静态地址重定位(static address relocation)是在程序调入内存执行之前，由装配程序将虚拟地址空间中的程序模块直接装配到内存的某些单元，从而直接完成地址映射工作。小微操作系统大多采用这种方式，以减少系统开销。

假定装配程序已分配了一块首地址为 BA 的内存单元区给虚拟地址空间内的某个程序模块。设起始指令或数据的虚拟地址为 VA，该指令或数据对应的内存起始地址为 MA。装配程序只要从各自的起始地址开始，按顺序装配程序中的所有内存地址即可。显然，对于虚拟地址空间内的指令或数据来说，静态地址重定位只完成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连接，否则将无法得到正确的内存地址和内存空间。

静态地址重定位的优点是不需要特殊硬件支持。但是，使用静态地址重定位方法进行地址变换无法实现虚拟内存。这是因为虚拟内存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，这要求计算机在执行一个程序时尽可能少使用内存，使尽可能多的执行程序模块可以共享内存，从而使计算机可以同时为更多的用户服务。这就需要内存管理系统把在计算机执行时频繁使用和立即需要的指令与数据存放在内存中，而把暂时不需要的部分存放在外存中，待需要时再将其调入，以提高内存的利用率和同时执行的计算机用户数。显然，这是与静态地址重定位方法矛盾的。采用静态地址重定位方法时，程序一旦装入内存就不能再移动，并且必须在程序执行之前将有关程序和数据全部装入。

静态地址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动态转换成内存地址。动态地址重定位依靠地址变换机构(为硬件)和相应的地址管理算法完成。

地址变换机构需要一个或多个基地址寄存器(BR)和一个或多个虚拟地址寄存器(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。

动态地址重定位过程如图 5.3 所示。

动态地址重定位的具体过程如下：

(1) 设置 BR 和 VR。

(2) 将程序段装入内存，且将其占用的内存单元区的首地址送入 BR 中。例如，在图 5.3 中，(BR)=1000。

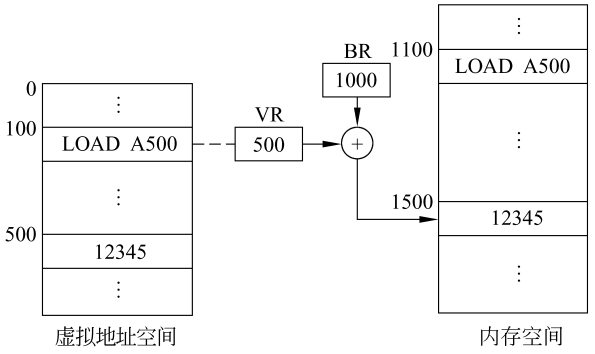


图 5.3 动态地址重定位过程

(3) 在程序执行过程中,将要访问的虚拟地址送入 VR 中,例如,在图 5.3 中,执行 LOAD A500 语句时,将要访问的虚拟地址 500 放入 VR 中。

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

动态地址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,对于完成同一任务所需的不同程序模块,只要把各程序模块在内存中的首地址存放在不同的 BR 中,就可以利用地址变换机构得到实际的内存地址。

(2) 动态地址重定位提供了实现虚拟内存的基础。因为动态地址重定位不要求在一个程序执行前为所有程序模块分配内存。也就是说,系统可以通过动态地址重定位机制部分地、动态地分配内存,从而可以在程序执行期间根据程序执行需要为不在内存中的程序模块分配内存,以达到内存扩充的目的。

(3) 有利于程序段的共享。

5.1.3 内存的管理与保护

1. 内外存数据流动控制

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交换数据。也就是说,管理系统要把即将执行的程序模块和数据段调入内存,而把不需马上执行、处于等待状态的程序模块和数据段调出内存。那么,按什么样的方式控制内存和外存之间的程序和数据流动呢? 最基本的控制办法有两种:一种是由用户程序控制;另一种是由操作系统控制。

用户程序控制内外存之间的程序和数据交换是比较困难的。其中的一个例子是覆盖(overlay)技术。覆盖技术要求用户清楚地了解程序的结构,并需要预先定义各程序模块调入内存的先后次序。在程序执行过程中,覆盖技术用后调入内存的程序模块覆盖已经在内存中的程序模块,从而起到节约内存空间的作用。覆盖是一种早期的内存扩充技术。使用覆盖技术,用户的编程负担很大,需要对计算机系统的软硬件都比较清楚,且程序模块的最大长度仍受内存总容量限制。因此,覆盖技术不能实现虚拟内存。

操作系统控制方式又可进一步分为交换(swapping)方式和调入方式。其中,调入方式又分为请求调入(on demand)方式和预调入(on prefetch)方式。

采用交换方式时，由操作系统把在内存中处于等待状态的进程或作业换出内存，而把等待事件已经发生、处于就绪状态的进程或作业换入内存。

采用请求调入方式时，在程序执行过程中，如果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。而采用预调入方式时，操作系统预测在不远的将来执行进程要访问哪些程序和数据段，并在它们被访问之前，选择适当的时机将它们调入内存。

由于内外存数据流的差别较大，而且进行一次内外存之间的数据交换需要一定的系统开销，因此，交换方式每次交换的数据和程序段较多，可能包含或大于除去常驻内存部分后的整个进程的程序段与数据段。而且，即使交换方式能完成部分交换，也不是按照执行的需要交换执行进程的程序段或数据块。它只是把受资源限制或暂时不能执行的程序段与数据块换出内存。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内存。

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内存。有关实现方法将在后面的章节中讲述。

2. 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，内存管理模块要为每一个并发执行的进程分配内存空间；当进程执行结束之后，内存管理模块又要及时回收该进程占用的内存资源，以便给其他进程分配内存空间。

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数据结构和策略：

（1）分配结构。用于登记内存使用情况，保存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。用于确定外存中的程序段和数据段在什么时间按什么样的控制方式进入内存。调入策略与前面所述内外存数据流动控制方式有关。

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

3. 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中，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，又要限制进程在自己的内存活动，各进程不能对其他进程的程序段和数据段产生干扰和破坏，因此必须对内存中的程序段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。

上下界寄存器保护法是一种常用的硬件保护法。上下界寄存器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段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，首先要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。上下界寄存器保护法的示例如图 5.4 所示。

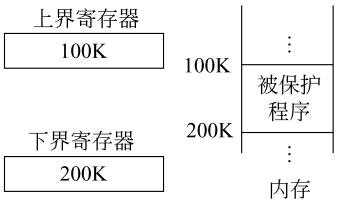


图 5.4 上下界寄存器保护法的示例

另外，保护键法也是一种常用的内存信息保护方法。保护键法为每一个被保护内存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段，对不同的进程赋予不同的开关字，与被保护的内存块中的保护键匹配。保护键可设置成对读写同时保护或只对读、写进行单项保护。例如，图 5.5 中的保护键 0 对 2K～4K 的内存区进行读写同时保护，而保护键 2 则只对 4K～6K的内存区进行写保护。如果开关字与保护键匹配或数据内存块未设保护键，则访问该内存块是允许的；否则将产生访问出错中断。

图 5.5 保护键法的示例

还有一种常用的内存保护方式是上下界寄存器与 CPU 的用户态或内核态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在上下界寄存器规定范围内的内存区，而内核态进程则可以访问整个内存地址空间。UNIX 系统就采用了这种内存保护方式。

5.2 分区管理

分区管理是把内存划分成若干大小不等的区域，除操作系统占用一个区域之外，其余区域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要求的最简单的内存管理方法。

下面结合分区原理讨论分区管理时的虚拟内存实现、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 基本原理

分区管理的基本原理是：给内存中的每一个进程划分一块适当大小的内存区，以连续存储各个进程的程序和数据，使各进程得以并发执行。

分区管理可以分为固定分区和动态分区两种方法。

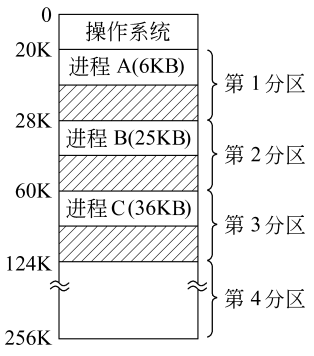
1. 固定分区法

固定分区法就是把内存区固定地划分为若干大小不等的区域。分区的原则由系统操作员或操作系统决定。例如，可将内存划分为长作业分区和短作业分区。分区一旦划定，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分区说明表进行，分区说明表中包括分区号、分区大小、起始地址和状态(是否是空闲区)。内存的分配和释放、内存信息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的内存状态的示例。

分区号	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 固定分区法的示例

在图 5.6 中，操作系统占用低地址部分的 20KB。其余空间被划分为 4 个分区，其中 1～3分区已分配，第 4 分区未分配。

2. 动态分区法

与固定分区法相比，动态分区法在作业或进程执行前并不建立分区，分区的建立是在作业或进程处理过程中进行的，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，分配程序将该分区依次划分给调度选中的作业或进程。图 5.7 给出了先进先出调度方式下的内存初始分配情况。

随着进程的执行，会进行一系列内存分配和释放。例如，在某一时刻，进程 C 执行结束并释放内存之后，管理程序又要为另外两个进程 E(设需内存 50KB)和 F(设需内存 16KB)分配内存。如果分配的空闲区比 E 和 F 要求的大，则管理程序将该空闲区分成两部分，其中一部分成为已经分配区，而另一部分成为新的小空闲区。图 5.8 给出了采用最先适应(first fit)算法分配内存时进程 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。

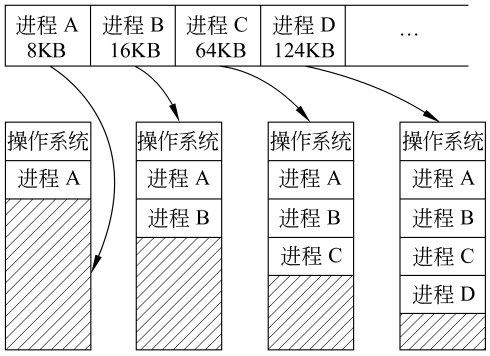


图 5.7 先进先出调度方式下的内存初始分配情况

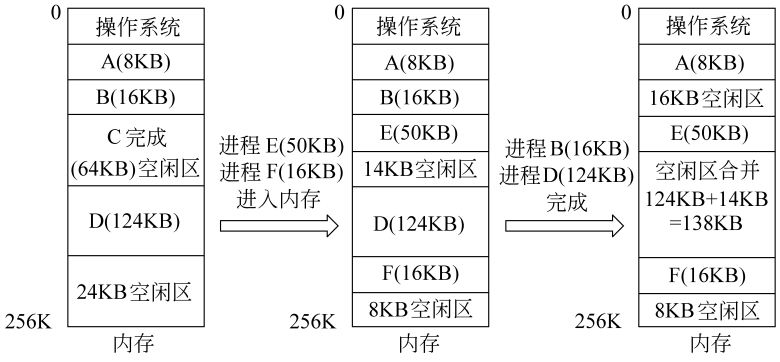
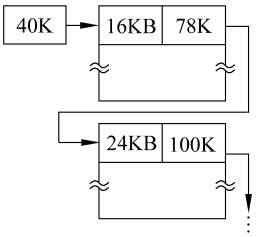


图 5.8 最先适应法内存分配变化过程

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组成可用分区表或可用分区自由链，以描述系统内的空闲内存资源。与此相对应，请求内存资源的作业或进程也组成一个内存资源请求表。图 5.9 给出了可用分区表、可用分区自由链和内存资源请求表的示例。

分区号	分区长度	起始地址
1	16KB	40K
3	24KB	78K
5	9KB	100K
⋮		

(a) 可用分区表



(b) 可用分区自由链

作业 (进程) 号	请求大小
P1	13KB
P2	20KB
⋮	⋮

(c) 内存资源请求表

图 5.9 可用分区表、可用分区自由链及内存资源请求表的示例

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

可用分区自由链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一个空闲区的起始地址，从而把所有的空闲区链接起来。然后，系统再设置一个链首指针，让其指向第一个空闲区的起始地址。这样，管理程序可通过链首指针查到所有的空闲区。采用自

由链管理空闲区，查找时要比可用表分区困难，但由于自由链的指针利用的是空闲区自身的单元，所以不必占用额外的内存空间。

内存资源请求表的每个表项描述请求内存资源的作业或进程号以及请求的内存大小。

无论采用可用分区表方式还是可用分区自由链方式，可用分区表或可用分区自由链中的各项都要按照一定的规则排列，以便于查找和回收。

5.2.2 内存的分配与回收

本节讨论分区法的分区分配与回收问题。

1. 固定分区时的内存分配

采用固定分区法时的内存分配较为简单。当用户程序要装入内存执行时，通过请求表提出内存分配请求和要求的内存空间大小。内存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，并将其分配给用户程序。固定分区法的分配算法如图 5.10 所示。

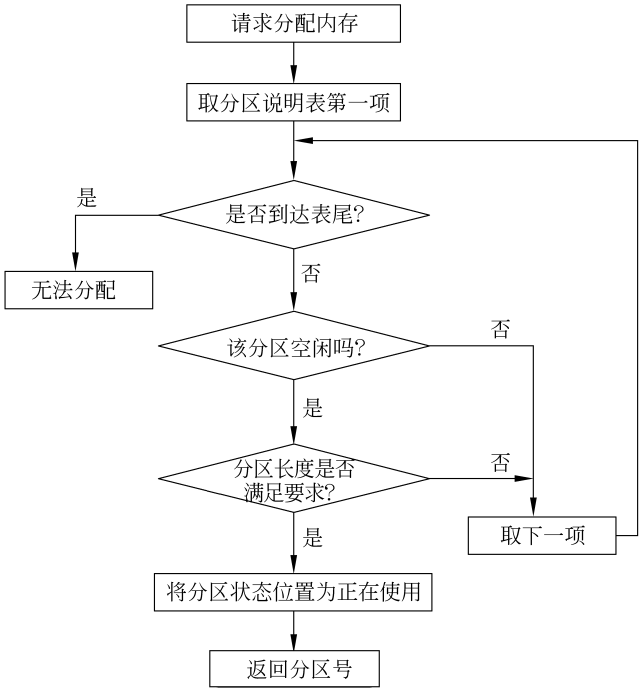


图 5.10 固定分区法的分配算法

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. 动态分区时的内存分配

动态分区时的内存分配主要解决以下 3 个问题：

(1) 对于请求表中要求的内存大小，从可用分区表或可用分区自由链中找出合适的空

闲区分配给进程或作业。

(2) 分配空闲区之后,更新可用分区表或可用分区自由链。

(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,更新可用分区表或可用分区自由链。

动态分区时分配空闲区的常用算法有 3 种,即最先适应算法(first fit algorithm),最佳适应算法(best fit algorithm)和最坏适应算法(worst fit algorithm)。这 3 种算法要求可用分区表或可用分区自由链按不同的方式排列。下面分别介绍这 3 种算法。

1) 最先适应算法

最先适应算法要求可用分区表或可用分区自由链按起始地址递增的次序排列。该算法最大的特点是一旦找到符合长度要求的分区就结束探索。然后,该算法从找到的分区中划出要求的内存空间分配给用户,并把余下的部分进行合并(如果有相邻空闲区存在)后留在可用分区表中,同时修改相应的表项。最先适应算法流程如图 5.11 所示。

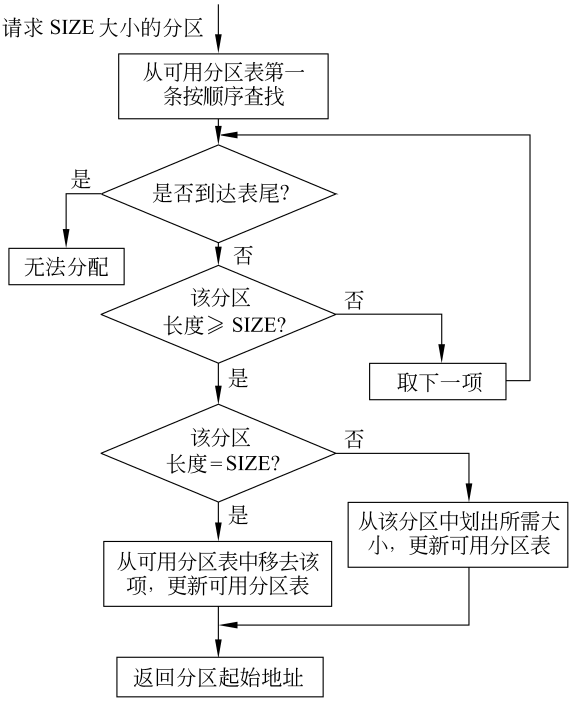


图 5.11 最先适应算法流程

2) 最佳适应算法

最佳适应算法要求将分区按长度从小到大的次序组成可用分区表或可用分区自由链。当用户作业或进程申请一个分区时,内存管理程序从表头开始查找,当找到第一个满足要求的分区时停止查找。如果该分区长度大于请求表中的请求长度,则与最先适应算法相同,将减去请求长度后的剩余空闲区留在可用分区表中。

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可用分区表或可用分区自由链。当用户作业或进程申请一个分区时,先检查可用分区表或可用分区自由链的第一个可用分区